hochschule hof
University of Applied Sciences

Angewandtes maschinelles Lernen – SS2020

„Jeder kann mit Künstlicher Intelligenz (KI) spielen“. Dieser Herausforderung haben sich Studenten der Hochschule für Angewandte Wissenschaften Hof im Sommersemester 2020 wieder gestellt. Bereits nach wenigen Vorlesungen konnten engagierte Studenten bestehende KI-Modelle für ihre Ziele anpassen und umgestalten und so eigene Aufgabenstellungen lösen.

In den letzten Jahren haben sich eine Vielzahl von Frameworks entwickelt, welche hochkomplexe und ressourcenintensive Berechnungen hinter einfach zu verstehenden Funktionen verstecken. Dies ermöglicht eine komfortable Modellierung von KI-Modellen, ohne zu technisch zu werden. Die Studenten müssen nicht zwingend verstehen, wie die jeweiligen Schichten eines Neuronalen Netzes implementiert werden, sondern sie können aus einem Katalog von Komponenten wählen und mit den jeweiligen „Produkten“ experimentieren.

In dieser Ausgabe des Semesterbandes haben die Studenten die Aussage mit dem Spielen fast zu wörtlich genommen. Ein Großteil der Arbeiten fokussiert sich auf das maschinelle Spielen. Im Zentrum steht ein Spiel, wie beispielsweise Super Mario, StarCraft oder Tetris. Diese Spiele sind ursprünglich für Menschen konzipiert. Sie verfügen über eine Bildschirmausgabe, in der mittels Computergrafik künstliche Welten geschaffen werden, mit denen der Spieler interagiert. Unter Nutzung von Tastatureingaben kann der Spieler seine Aktionen durchführen. Je nach Spiel kann er sich frei in der Spielewelt bewegen oder bestimmte Handlungen durchführen. Wenn jetzt eine Künstliche Intelligenz ein Spiel spielen soll, dann benötigt diese zum einen eine optische Komponente. Mit dieser muss sie sich ein Abbild der Spielewelt anlegen können. Sie benötigt eine Menge von möglichen Aktionen, welche durchführbar sind. Und es muss ein Ziel definiert werden, was besonders wichtig ist. Es liefert nämlich die Grundlage für eine Bewertung. Also wie gut war eine Aktion und wie viel hat die Aktion geholfen das Ziel zu erreichen, oder war die Aktion sogar schlecht und sollte nicht wiederholt werden. Die Komplexität wird noch erhöht, in dem es sein kann, dass sich zum Zeitpunkt der Aktion noch gar nicht sagen lässt, wie sich die gewählte Aktion auf das Ziel auswirkt.

Viele Fragen und komplizierte Annahmen. Der Seminarband verdeutlicht, dass diese anfänglich so komplex klingenden Aussagen gelöst werden können. Ziel dieser Veröffentlichung ist es den Leser zu motivieren die Beispiele nachzuvollziehen und selbst Erfahrungen mit KI zu sammeln.

Sebastian Leuoth (Hrsg.)
Hochschule für Angewandte Wissenschaften Hof, 2021
ISBN: 978-3-935565-18-9

Inhaltsverzeichnis

Visualisierung von Convolutional Neural Networks

Eine Einführung in die Visualisierung des Klassifikationsprozesses mit CNNs am Beispiel des VGG16 Modells

Mehmet Daglioglu
Fakultät Informatik
Hochschule Hof
Hof, Bayern, Deutschland
mehmet.daglioglu@hof-university.de

Zusammenfassung

Die vorliegende Arbeit soll einen Überblick über die Funktionsweise von CNNs verschaffen, indem der Klassifizierungsprozess eines derartigen Netzes erläutert wird. Ein weiterer Teil der Arbeit verfolgt das Ziel, Klassifizierungsentscheidungen transparenter und nachvollziehbarer zu gestalten. Der hierzu verwendete Ansatz Gradient-weighted Class Activation Mapping (Grad-CAM) wurde erstmalig von Selvaraju et. al [15] beschrieben. Einen wesentlichen Teil der Arbeit stellt das Praxisprojekt dar, in dem die Vorgänge in einem CNN schichtweise visualisiert sowie der Grad-CAM Ansatz implementiert werden. Im ersten Teil wird zunächst auf die möglichen Ausprägungen der Schichten in einem CNN für Bildklassifikation eingegangen, bevor der Klassifizierungsprozess selbst erläutert wird. Im Praxisprojekt wird hierzu ein Ansatz vorgestellt, mit dem der Inhalt eines Eingangsbildes klassifiziert und dabei schichtweise jedes Zwischenergebnis gespeichert wird. Dies ermöglicht einen interaktiven Einblick in das Modell und den Verarbeitungsstand des Bildes in den jeweiligen Schichten. Hierzu wird beispielhaft das VGG16 Modell verwendet, das von Karen Simonyan und Andrew Zisserman [17] vorgestellt wurde. Im zweiten Teil wird der Grad-CAM Ansatz vorgestellt, der es im Bereich der Bilderkennung möglich macht, die für eine Klassifizierung wichtigen Bildbereiche in Form einer Heatmap zu markieren und somit die Klassifizierungsentscheidung transparenter zu gestalten. Dieser wird im Praxisprojekt implementiert und dokumentiert. Mithilfe der Implementierung werden in dieser Arbeit verschiedene Bilder klassifiziert, Grad-CAM angewendet und die Ergebnisse evaluiert. Dabei wird auf Ergebnisse eingegangen, bei denen das Modell den Bildinhalt richtig klassifiziert hat, es werden aber auch Klassifizierungsentscheidungen beobachtet, bei denen das Bild fehlerhaft klassifiziert wurde. Sowohl bei richtigen, als auch bei falschen Entscheidungen kann analysiert werden, welche Bildbereiche für die Klassifizierung ausschlaggebend waren. Bei falsch klassifizierten Eingangsbildern kann dies unter anderem auf Mängel in den Trainingsdaten hinweisen. Im Gegensatz zum oben genannten Anwendungszweck ermöglicht der Grad-CAM Ansatz auch, dass Bildbereiche markiert werden, die gegen die Klassifizierung einer jeweiligen Klasse sprechen. Auch dieser Anwendungsfall wird in der Arbeit beschrieben und evaluiert sowie im Praxisprojekt implementiert.

1 Einleitung

1.1 Motivation

Das Themengebiet des maschinellen Lernens erfreut sich immer größerer Beliebtheit. Ein wichtiges Thema in diesem Bereich ist die Klassifizierung von Bildern anhand Neuronaler Netze. In [14] wird beschrieben, dass diesem Zusammenhang insbesondere Convolutional Neural Networks (CNN), zu Deutsch „faltende neuronale Netzwerke“ erfolgversprechend sind, weil sie eine hohe Genauigkeit in der Klassifizierung bieten können. So sind komplexe und tiefe CNNs die Basis vieler ML-Anwendungen und erzielen in der Bilderkennung durch das Erfassen räumlicher Eigenschaften und Abhängigkeiten oft bessere Ergebnisse als andere Ansätze.
Jedoch besteht ein entscheidender Nachteil: In [3] wird beschrieben, dass durch die Komplexität der Architektur tiefer neuronaler Netze die Entscheidung eines solchen Netzes nicht mehr einfach nachzuvollziehen ist.
Zhou beschreibt in [20], wie ein CNN erstellt und trainiert werden kann. Demnach wird über ein sog. Training versucht, mit vorhandenen Daten einen Algorithmus zu generieren, der richtige Klassifizierungen möglich macht. Auf den Trainingsprozess soll im Rahmen dieser Arbeit nicht eingegangen werden.
In [9] wird genannt, dass durch den komplexen Aufbau im Nachhinein nicht mehr nachvollziehbar ist, wie dieser Algorithmus/das Modell zustande kam und mit welchem kausalen Zusammenhang die Klassifizierungsentscheidung getroffen wurde. Durch die fehlende Interpretierbarkeit kann eine solche Anwendung im Betrieb als „Black-Box“ verstanden werden. Wenn Anwendungen, die darauf beruhen, fehlerhaft sind, ist es unvorhersehbar und schwer nachvollziehbar.
Außerdem beruht das durch ein Training entstandene Modell auf den für das Training verwendeten Daten. Diese Daten sind nicht immer optimal gewählt, so kann es laut [13] passieren, dass das entstandene Modell nicht in der Lage ist, die gelernten Informationen sinnvoll zu abstrahieren und richtige Entscheidungen zu treffen. Unter Umständen kann

es passieren, dass ein Modell sich zu stark an die Trainingsdaten anpasst und nicht mehr richtig abstrahiert, also nicht mehr die zugrundeliegenden Formen lernt. Dieser Effekt wird Overfitting (zu Deutsch „Überanpassung“) genannt und führt dazu, dass neue Bilder nicht mehr sinnvoll erkannt werden. Dieses Phänomen kann unter Anderem eintreten, wenn zu wenig Trainingsdaten vorhanden sind oder Netzarchitekturen gewählt werden, die keinen Mechanismus für die Regularisierung enthalten. Er beschreibt Dropout als eine Möglichkeit für die Regularisierung, die zufällig Informationen im Modell löscht und damit verhindert, dass das Modell sich zu stark an die Trainingsdaten anpasst.
In vielen Anwendungsfällen ist es nicht verantwortungsvoll, sich auf ein Ergebnis zu verlassen, das weder nachvollziehbar, noch erklärbar ist. So wäre ein Einsatz dieser effizienten und verlässlichen Technologie z.B. im medizinischen Bereich kaum denkbar, wenn die Ergebnisse nicht begründet oder nachvollzogen werden können. Ein klassisches Beispiel für ein Neuronales Netz, das mit falschen Daten trainiert und eingesetzt wurde, ist aus einem Artikel von Ribeiro et al. 2016 [12]. Das Modell sollte zwischen Wölfen und Huskys unterscheiden und funktionierte vermeintlich sehr gut. Jedoch wurde anschließend festgestellt, dass manche Huskys fälschlicherweise als Wölfe klassifiziert wurden. Der Grund hierfür war, dass auf den Trainingsbildern von Wölfen oft Schnee im Hintergrund abgebildet war und auf Bildern von Huskys nicht. Das Modell hatte nicht gelernt, Wölfe von Huskys zu unterscheiden, sondern klassifizierte Bilder mit Schnee im Hintergrund als Wölfe und Bilder mit anderem Hintergrund als Huskys.
Um die Funktionsweise von CNNs zu verstehen, soll in dieser Arbeit zunächst der Aufbau eines CNNs erläutert und die Vorgänge im Klassifizierungsprozess visualisiert werden. Für die Nachvollziehbarkeit von Entscheidungen und eine mögliche Fehlersuche für die genannten Probleme soll im Anschluss der Ansatz Gradient-weighted Class Activation Mapping (Grad-CAM) von Selvaraju et. al [15] beschrieben werden, der die für die jeweilige Klassifizierung ausschlaggebenden Bereiche im verwendeten Bild in Form einer Heatmap darstellt. Im oben genannten Beispiel würde eine solche Heatmap zeigen, dass die Bilder nicht anhand der Tiere im Vordergrund als jene klassifiziert werden, sondern anhand der Farbe des Hintergrundes.

1.2 Projektumfeld

Die vorliegende Arbeit wurde im Rahmen des fachwissenschaftlichen Wahlpflichtmoduls „Angewandtes Maschinelles Lernen“ an der Hochschule für Angewandte Wissenschaften Hof (Fakultät Informatik) im Studiengang Wirtschaftsinformatik erstellt. Laut Modulhandbuch [6] ist das Ziel des Moduls, dass die Studierenden eine Einführung in das Gebiet des maschinellen Lernens erhalten, indem wichtige Prinzipien, Techniken und Ansätze zunächst theoretisch erläutert werden. Anschließend fertigen die Studierenden ein Projekt und eine Arbeit an, die im Rahmen des Moduls präsentiert werden. Das dazugehörige Projekt wird mithilfe von Google Colab [1], einem Online Tool, mit dem es möglich ist, Python Code im Browser zu schreiben und auszuführen, erstellt. Ein entscheidender Vorteil dieses Tools ist, dass wichtige ML-Libraries bereits installiert sind und dass freier Zugriff auf GPU Leistung, die für ML-Anwendungen benötigt wird, gegeben ist. Für das Projekt wird das VGG16-Modell, das von Karen Simonyan und Andrew Zisserman [17] vorgestellt wurde, verwendet. Das Modell wurde bereits mit ImageNet Bildern trainiert. ImageNet ist ein Datensatz mit über 14 Millionen Bildern [4], die für das genannte Modell in 1000 Klassen eingeteilt wurden. Es werden außerdem die Frameworks TensorFlow [2] und Keras [3] verwendet. Tensorflow ist ein von Google entwickeltes Framework für maschinelles Lernen und Künstliche Intelligenz (KI). Es beinhaltet viele nützliche Instrumente für ML-Anwendungen. Keras ist eine nutzerfreundliche Bibliothek, die in TensorFlow enthalten ist und als high-level API dient.

2 Related Work

Die vorliegende Arbeit beschäftigt sich mit CNNs, die Klassifizierungsprobleme lösen. Einige Artikel, die auch als Quellen für diese Arbeit verwendet wurden (z.B. [2], [7], [14]), beschäftigen sich damit, wie CNNs aufgebaut sind und wie sie funktionieren. Gorner erklärt in seinem GitHub Repository [5] sehr anschaulich, wie TensorFlow verwendet werden kann, um Deep-Learning Modelle zu erstellen. Dabei wird unter Anderem erklärt, wie man eigene Modelle definiert und wie das Training funktioniert. Mit der Visualisierung der Zwischenergebnisse aus den Schichten des CNNs beschäftigt sich ein Artikel von Brownlee [3]. Dieser Artikel wurde auch für das erste Notebook im Colab Notebook des Praxisprojektes verwendet. Als Grundlage für die Erklärung von Grad-CAM dient die Arbeit von Selvaraju et. al [15], die den Ansatz erstmalig beschreibt. Meudec zeigt in [8] eine mögliche Implementierung von Grad-CAM. Diese diente auch als Hilfestellung für die Implementierung im zweiten Colab Notebook des Praxisprojektes. Verwandte Ansätze wie die in [19] und [11] beschreiben auch Möglichkeiten, mit denen Nachvollziehbarkeit der Entscheidungen in der Klassifizierung mithilfe von Heatmaps gewährleistet werden soll.

3 Aufbau und Visualisierung eines CNNs

Ein Convolutional Neural Network (zu Deutsch „gefaltetes neuronales Netzwerk“) ist ein künstliches neuronales Netz, das sehr gute Ergebnisse im Bereich der Bilderkennung liefert. Für die Bilderkennung dient in einem CNN das Bild in Form einer zweidimensionalen Matrix als Input in die

[1] colab.research.google.com
[2] www.tensorflow.org
[3] keras.io

Anwendung, die in Schichten aufgebaut ist. Im Folgenden wird auf die verschiedenen Schichten am Beispiel des CNN Modells VGG16 eingegangen. VGG16 ist ein Modell mit sehr kleinen sogenannten Convolution Filtern (3x3), das in Keras enthalten ist und mit dem ImageNet Datensatz trainiert wurde. Der grundsätzliche Aufbau eines CNNs besteht aus einem bis mehreren Convolutional Layers, an die oft ein Pooling Layer anschließt. Diese Folge kann sich mehrfach wiederholen. Im genannten Modell folgt ein Flatten Layer, bevor es mit einer Reihe aus Fully Connected Layers abgeschlossen wird. Abbildung 1 zeigt eine schematische Darstellung von CNNs. Es gibt viele weitere Möglichkeiten, wie die Architektur eines CNNs gestaltet werden kann. Als weiterer Ansatz können zum Beispiel Inception Module genannt werden, die in der GoogLeNet Architektur verwendet und in [18] beschrieben werden.

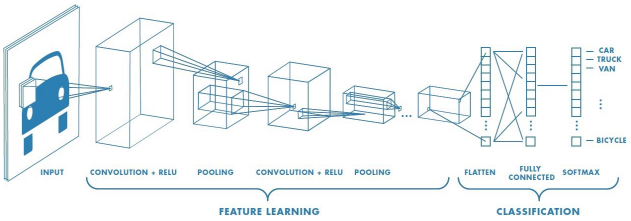


Abbildung 1. Schematischer Aufbau eines CNNs (aus [14])

Ein Bild, das klassifiziert werden soll, wird in das Modell hineingegeben und in den Schichten des Modells unterschiedlich verarbeitet. Die Ergebnisse einer Schicht dienen als Inputs für die nächste Schicht. In Abbildung 2 ist die Netzwerktopologie des verwendeten Modells zu sehen. Um ein Verständnis über die Vorgänge im Klassifizierungsprozess zu schaffen wird das Modell im Praxisprojekt an beliebiger Stelle abgeschnitten und ein Bild bis zu diesem Punkt verarbeitet. Die entstehenden Teilmodelle starten immer mit dem Input-Layer und gehen bis zu dem Punkt, an dem abgeschnitten wurde. Im Colab Projekt wird das Modell nach jeder Schicht des Modells abgeschnitten, die Zwischenergebnisse werden angezeigt und als Bilder gespeichert.

3.1 Input Layer

Der erste Layer eines Neuronalen Netzwerks ist immer der Input Layer. In einem CNN ist es, im Vergleich zu einem Perzeptron, möglich, dass eine mehrdimensionale Matrix als Eingabeelement dient [14]. Beim Klassifizieren eines Bilds ist der Input in das Modell das zu klassifizierende Bild. Die Schnittstelle des genutzten Modells in Keras erwartet ein Eingabeformat von 224x224 Pixeln mit drei Farbdimensionen. Ein Bild, das eine andere Größe hat, muss vor der Verwendung in das Zielformat transformiert werden. Die Dokumentation beschreibt, dass die Farbkanäle in die Reihenfolge „BGR“ konvertiert und null-zentriert werden müssen. Das bedeutet, dass von jedem RGB Farbwert der Mittelwert der

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000

Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0

Abbildung 2. Netzwerktopologie des VGG16 Netzwerkes

Farbwerte aus den Trainingsdaten (also allen Trainingsbildern aus dem ImageNet Datensatz) abgezogen wird.
Keras bietet für die Transformation von Bildern anderer Formate passende Methoden an. Der Output des Input Layers sind dementsprechend drei Matrizen mit 224x224 Werten. Diese Matrizen stellen die drei Farbkanäle des ursprünglichen Bilds dar und können wieder zu Bildern in Graustufen zusammengesetzt werden. Das Ergebnis ist in Abbildung 3 zu sehen. In den nächsten Schichten werden sowohl die Werte in den Matrizen als auch deren Form und Anzahl manipuliert.

3.2 Convolutional Layer

Das Ergebnis des Input Layers ist der Input in den ersten Convolutional Layer. Die Abläufe in dieser Schicht werden von Saha in [14] beschrieben. In dieser Schicht werden Matrizen mit Filtern analysiert, die selber Matrizen (sog. Kernel) mit einer bestimmten Größe sind. Diese Filter bewegen sich in einer festgelegten Schrittgröße (sog. Stride) über die zu

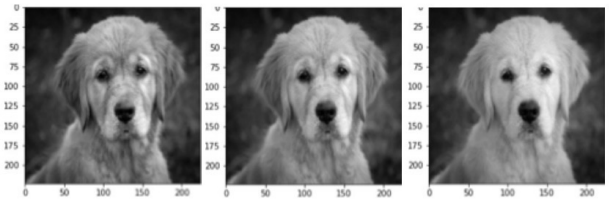


Abbildung 3. Ergebnis des Input Layers

verarbeitenden Matrizen von links nach rechts, Zeile für Zeile wie in Abbildung 4. Ein weiterer wichtiger Begriff ist das sog. Padding, das beschreibt, ob und wie viele Schritte der Filter über die Ränder der Eingabematrizen hinaus bewegt werden soll. Je nachdem, wie Filtergröße, Stride und Padding gewählt werden, unterscheidet sich das Ergebnis einer Schicht in ihrer Größe von der Eingabe.

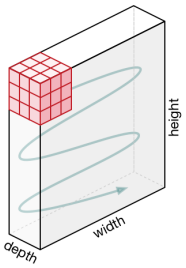


Abbildung 4. Bewegung eines 3x3 Kernels über drei Inputkanäle (aus [14])

Ein neuer Wert berechnet sich - wie in Abbildung 5 zu sehen - daraus, dass alle Werte der Input Matrix, die im aktuellen Schritt mit dem Filter überlappen, mit dem jeweilig darüber liegenden Filterwert multipliziert und alle Ergebnisse aufsummiert werden. In Abbildung 5 ist außerdem zu sehen, dass es beim Padding Methoden gibt, wie Randregionen behandelt werden. Im Bild werden die Zahlen nebenan verwendet, wenn der Filter über die Matrix hinausgeht. Die Größe des Kernels, der Stride, das Padding und die Padding Methode werden beim Erstellen eines solchen Modells fest definiert. In [14] wird außerdem beschrieben, dass die Vorgänge in einem Convolutional Layer sich am visuellen Kortex orientieren. Die Neuronen reagieren nur auf Reize innerhalb ihres begrenzten Gesichtsfelds (sog. Rezeptionsfeld). Durch die gewählte Schrittgröße werden mehrere Felder generiert, die sich entweder überlappen und damit den gesamten Bereich abdecken, oder (bei höheren Schrittgrößen) Zwischenbereiche auslassen. Wenn die Ergebnismatrizen kleiner sind als die Ursprünglichen, spricht man von „Strided Convolution“.
In Convolutional Layers besteht der Input oft aus mehreren Matrizen, wobei der jeweilige Filter über jeder Matrix angewendet wird. In Abbildung 6 wird gezeigt, dass die dabei entstehenden Werte dann addiert werden und ggf. zusätzlich

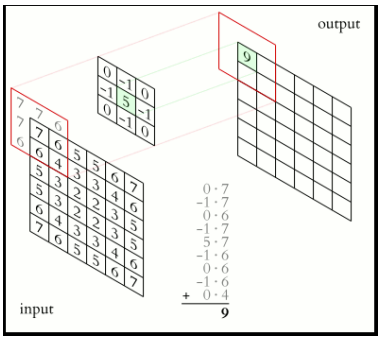


Abbildung 5. Errechnung eines neuen Wertes aus einer Inputmatrix und einem 3x3 Filter (aus [10])

noch ein fester sog. Bias-Wert hinzugefügt wird. Aus einem Filter mit mehreren Kanälen, der einen Input mit mehreren Kanälen verarbeitet, entsteht somit genau ein Output-Kanal.

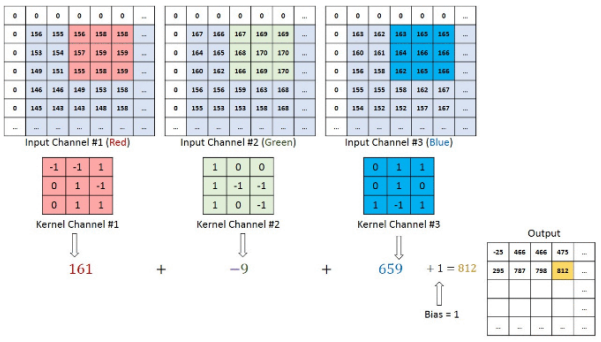


Abbildung 6. Addition der Filterergebnisse mehrerer Input-Matrizen (aus [14])

Ein weiteres wichtiges Konzept ist die Nutzung von Aktivierungsfunktionen in den Schichten. Die bisherigen Berechnungen multiplizieren Werte mit Gewichten und addieren den Bias. Damit lassen sich die Funktionen in der Form $f(x) = a*x+b$ darstellen, sie sind also linear. Die Ergebnisse einer Berechnung können Werte zwischen -∞ und +∞ sein, es gibt keine Grenzen. Sharma beschreibt in [16], dass die Entscheidung getroffen werden muss, ob ein Neuron aktiviert wird oder nicht. Er vergleicht die Aktivierung mit der der Neuronen im Hirn. Für diesen Mechanismus kann die Rectified Linear Unit (ReLU) genutzt werden. Sie ist eine nicht-lineare Aktivierungsfunktion, die negative Werte in null umwandelt und positive Werte nicht verändert.

$$f(x) = max(0, x) \tag{1}$$

Diese Funktion ist ein effizienter Weg, mit dem erreicht wird, dass nur diejenigen Neuronen aktiviert werden, deren Ergebnis der Berechnung positiv ist. Wenn Neuronen mit negativen Werten nicht aktiviert werden, wird das Netzwerk

effizienter, weil insgesamt weniger Berechnungen durchgeführt werden müssen.
Die in diesem Schritt entstehenden neuen Matrizen enthalten auch Zahlenwerte, die wieder zu Bildern zusammengesetzt werden können. Die Vorgänge der Bildverarbeitung mit Filtern in einem Convolutional Layer erlauben es, z.B. Kanten oder Formen aus den Eingangsbildern zu extrahieren. Somit ist ein Neuronales Netz durch den genannten Aufbau in der Lage, räumliche Abhängigkeiten zu erfassen und Formen zu berücksichtigen [14].
Becker beschreibt in [1] und [2], dass die Filter nicht vom Entwickler des Netzes definiert werden. Die Werte in einem Filter sind sogenannte Gewichte, die in einem Trainingsprozess schrittweise gelernt werden. Auch der genannte Bias Wert wird auf diese Weise generiert. Für ein Training wird ein Modell definiert, das initiale Gewicht-Werte (z.B. zufällige Zahlen) beinhaltet. Es werden nun disjunkte Mengen an Trainings- und Testbildern mit ihren zugehörigen Klassen benötigt. Wenn das Modell mit initialen Gewichten die Testdaten klassifiziert, ist die Fehlerrate selbstverständlich sehr hoch. Der Trainingsprozess verfolgt das Ziel, die Fehlerrate zu minimieren. Hierzu werden die Gewichte mithilfe der Trainingsdaten schrittweise an diese angepasst. Damit passt das Modell sich insgesamt an die Trainingsdaten an. Nach jedem Trainingsschritt werden die Testbilder klassifiziert und die Fehlerrate betrachtet. Ziel ist es, die Fehlerfunktion per Backpropagation zu minimieren, sodass das Modell in der Lage ist, auch trainingsfremde Bilder richtig zu erkennen. Wenn die Trainingsdaten in ausreichender Menge und Qualität vorliegen, hat das Modell nach vielen Trainingsschritten gute Gewichtswerte und ist in der Lage, die Eigenschaften auf den Bildern (Formen, Kanten, etc.) zu generalisieren und auch fremde Bilder richtig zu klassifizieren. Die genannten Gewicht-Werte können auch negative Werte annehmen.
Das im Projekt verwendete VGG16-Netz hat eine Filtergröße von 3x3 Pixeln und sowohl der Stride als auch der Padding Wert ist eins [17]. Die gewählten Werte führen dazu, dass die Output-Matrizen dieselbe Größe haben wie die Input-Matrizen (224x224 Pixel). In Abbildung 2 ist zu sehen, dass im ersten Convolutional Layer 64 Filter verwendet werden, was bedeutet, dass aus den drei Eingangsmatrizen 64 Matrizen der Größe 224x224 entstehen und an die nächste Schicht übergeben werden. In Abbildung 7 sind die ersten sechs Filter des ersten Convolutional Layers für alle drei Farbkanäle abgebildet. Die Werte wurden zwischen 0 und 1 normiert und werden in Graustufen angezeigt, während 1 weiß und 0 schwarz entspricht. Die ersten sechs Ergebnisse des sog. Faltungsprozesses dieser Schicht (sog. Feature-Maps) sind in Abbildung 8 zu sehen. Im zugehörigen Projekt wurden alle 4224 Feature-Maps aller Convolutional Layer des VGG16-Modells geloggt und können betrachtet werden.

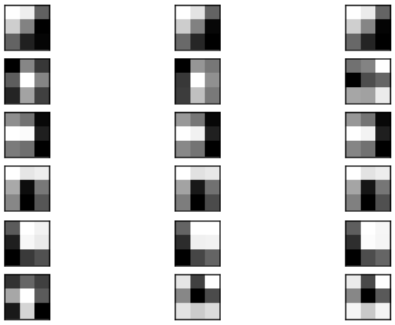


Abbildung 7. Die ersten sechs Filter des ersten Convolutional Layers des VGG16-Modells normiert und in Graustufen

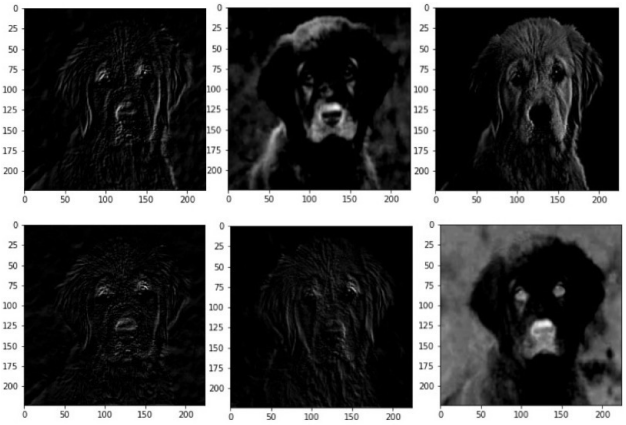


Abbildung 8. Beispiel: Die ersten 6 Feature-Maps des ersten Convolutional Layers

3.3 Pooling Layer

In Kapitel 3.2 wurde beschrieben, dass die Größe der Matrizen im Convolutional Layer des verwendeten Modells nicht verändert wird, dies aber bei anderen Modellen durchaus der Fall sein kann. Um die bei der Klassifizierung benötigte Rechenleistung zu verringern, kann durch Strided Convolutions oder Pooling die Größe der Matrizen verringert und somit eine Dimensionsreduktion erreicht werden. Saha beschreibt das Pooling in [14]. Es wird erneut ein Kernel verwendet, der sich in ähnlicher Form (ohne Padding) über die Eingangsmatrix bewegt. In diesem Fall beinhaltet der Kernel keine Werte, sondern hat lediglich eine festgelegte Größe und einen Stride-Wert. Im Average-Pooling wird ein Durchschnitt der Werte gebildet, über denen der Kernel gerade steht. Im Max-Pooling wird nur der größte Wert in die Ergebnismatrix übertragen. Beim Pooling-Prozess werden gleichzeitig dominante Merkmale im ursprünglichen Bild erkannt . Die beiden verschiedenen Pooling-Methoden

sind in Abbildung 9 visualisiert. Wie in Abbildung 2 zu sehen, wird im verwendeten VGG16-Modell ein Max-Pooling verfahren verwendet. Die Filtergröße ist 2x2 Pixel und es wurde ein Stride-Wert von zwei definiert [17]. Damit wird die Matrizengröße nach jedem Pooling Layer halbiert. Weil in den Input-Matrizen des Pooling Layers nur positive Werte enthalten sind, sind auch im Output nur positive Werte enthalten und es bedarf keiner weiteren Verwendung der ReLU-Aktivierungsfunktion. Die ersten sechs Ergebnis-Matrizen des Pooling Layers sind in Abbildung 10 zu sehen. Im zugehörigen Projekt wurden alle 1472 Feature-Maps aller Pooling Layer des VGG16-Modells geloggt und können betrachtet werden.

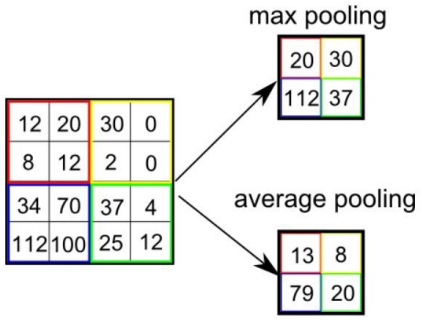


Abbildung 9. MaxPooling und AVG-Pooling aus [14]

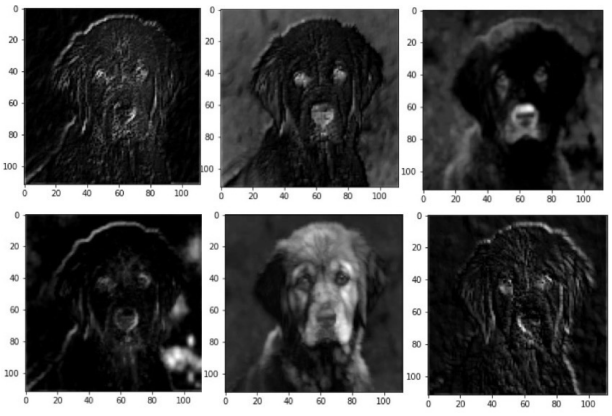


Abbildung 10. Beispiel: Die ersten 6 Feature-Maps des ersten Pooling Layers

3.4 Blöcke

In einem CNN wird der Ablauf eines oder mehrerer Convolutional Layers, denen oft ein Pooling Layer folgt, beliebig oft wiederholt. Dabei werden die verarbeiteten Matrizen immer kleiner, während immer mehr Filter verwendet, also immer mehr Matrizen verarbeitet werden. Das Kleinerwerden der Matrizen erfolgt durch Strided Convolutions oder durch die Pooling Layer. In den tieferen Schichten im Modell werden die Informationen immer abstrakter, was in den Convolutional Layers dazu führt, dass generalisierte Formen erkannt werden können. Außerdem kann in Abbildung 1 beobachtet werden, dass die Datenmenge im Verlauf kleiner, jedoch tiefer wird (die Matrizen werden kleiner, während durch die Anwendung vieler Filter mehr Matrizen entstehen). In Abbildung 2 ist ersichtlich, dass die Matrizen bis zu einer Größe von 7x7 Pixeln schrumpfen und im letzten Block 512 Filter verwendet werden. Folglich besteht der Output des letzten Blocks aus 512 Matrizen der Größe 7x7. In Abbildung 11 sind die sechs ersten Feature-Maps des letzten Pooling Layers abgebildet. Es ist aufgrund der geringen Größe kaum noch etwas vom Ursprungsbild zu erkennen. Dennoch reichen die Informationen, die nach den Filtern noch vorhanden sind aus, um mit den restlichen Schichten eine Klassifizierung mit sehr guten Ergebnissen durchzuführen. Aufgrund der großen Anzahl und der geringen Pixelgröße ist es mit dem bloßen Auge jedoch nicht möglich, Informationen darüber zu erkennen, welche Bereiche und Ausprägungen des Ursprungsbilds für die jeweilige Klassifizierung ausschlaggebend sein könnten.

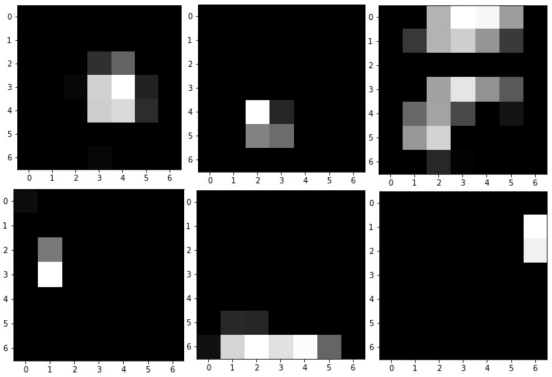


Abbildung 11. Beispiel: Die ersten 6 Feature-Maps des letzten Pooling Layers

3.5 Flatten Layer

Im Flatten Layer werden die Matrizen aus dem letzten Pooling Layer in einen Vektor umgewandelt, indem alle Werte hintereinander gesetzt werden. Dies ist notwendig, weil die folgenden Schichten keine Convolutional Layer, sondern Perzeptrons sind, und eindimensionale Vektoren erwarten. Im VGG16-Modell ist das ein Vektor mit 512x7x7=25088 Werten, der den Output des Flatten Layers darstellt. Durch die Tatsache, dass die Werte aneinandergereiht werden, werden nach dieser Schicht keine räumlichen Informationen mehr verarbeitet.

3.6 Fully Connected (FC) Layer

Den Fully Connected oder Dense Layer beschreibt [2] als eine Schicht, bei der jeder Output-Wert der vorhergehenden

Schicht als Input für jedes einzelne Neuron der betrachteten Schicht dient. Der Aufbau ist in Abbildung 12 zu sehen. In einem Neuron werden auch hier die Input Werte mit einem individuellen Gewicht multipliziert und die Ergebnisse addiert. Zuletzt wird auch hier ein Bias-Wert hinzugefügt, bevor die Aktivierungsfunktion verwendet wird. In allen FC-Layern außer dem letzten wird im VGG16-Modell die ReLU-Funktion verwendet. Der letzte FC-Layer dient der Klassifizierung. Hier gibt es so viele Neuronen, wie es Klassen gibt, denn jedes Neuron steht für eine Klasse. Die Werte, die in den Neuronen im letzten Layer enthalten sind, beschreiben die Aktivität eines Neurons in Bezug auf die Klasse, für die es steht. In dieser Schicht werden die Werte anschließend mit der Softmax-Funktion auf Werte zwischen 0 und 1 normiert werden.

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2}$$

Die Ergebnisse dieser Schicht können als Wahrscheinlichkeit dafür gesehen werden, dass der Bildinhalt die jeweilige Klasse darstellt, die Summe aller Werte der Output-Neuronen ist 1. Das Modell klassifiziert den Bildinhalt als die Klasse des Neurons, das den höchsten Wert beinhaltet. Auch die Gewicht- und Bias-Werte der FC-Layer werden im Training generiert. Die Anzahl der Neuronen in FC-Layern wird bei der Definition des Netzes festgelegt, während sich die Anzahl der Neuronen des letzten FC-Layers aus der Anzahl an möglichen Klassen ergibt. Im verwendeten VGG16-Modell wurden 3 FC-Layer definiert, wobei die ersten beiden die ReLU-Aktivierungsfunktion und der letzte Layer Softmax für die Klassifizierung nutzen. Die Klassifizierung findet im Projekt statt, indem kein Layer abgeschnitten, also das Modell vollständig verwendet wird. Das zu klassifizierende Bild wird mit der von Keras bereitgestellten Schnittstelle in das Modell gegeben und der Output der letzten Schicht ausgelesen. Die letzte Schicht liefert, wie bereits beschrieben, für jede mögliche Klasse einen Wert zwischen 0 und 1, wobei der höchste Wert der ist, der die ermittelte Klasse beschreibt. Das Bild, das als Beispiel für die obigen Erklärungen gedient hat, wurde im Projekt als „Golden Retriever“ klassifiziert.

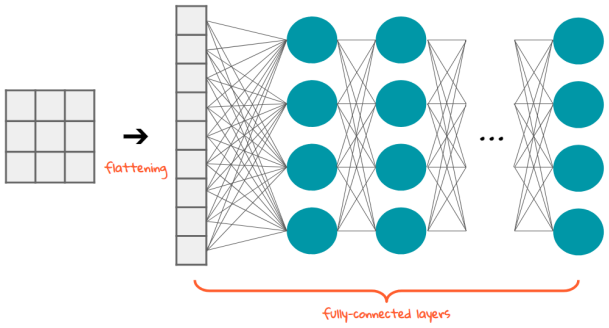


Abbildung 12. Flatten und FC-Layer aus [7]

3.7 Fazit Layer-Visualisierung

Das Ziel dieses Kapitels war es, den Aufbau eines CNNs zu beschreiben und die Vorgänge im Klassifizierungsprozess zu erläutern bzw. zu visualisieren. Es ist interessant zu sehen, was am Ende der Blöcke aus Convolutional Layers und Pooling Layers noch vom ursprünglichen Bild übrig ist und wie es Schritt für Schritt zu diesem Ergebnis kommt. Jedoch ist es aufgrund der geringen Größe von 7x7 Pixeln der letzten Bilder nicht möglich, zu sehen, aufgrund welcher Bildbereiche des Ursprungsbilds eine Klassifizierung vorgenommen wurde. Somit ist es mit dem obigen Ansatz noch nicht möglich, fehlerhafte Netzwerke (wie z.B. in 1.1) zu debuggen oder eine Nachvollziehbarkeit bzw. Erklärbarkeit der Klassifizierungen eines Netzes zu gewährleisten. Es bleibt nach wie vor ein Modell, dessen Weights und Biases in einem Training so angepasst wurden, dass die Fehler der Klassifizierung der Trainings- und Testdaten minimiert werden. Der Ansatz bietet dementsprechend auch keine Möglichkeit, Fehler in den gewählten Daten zu finden.

4 Gradient-weighted Class Activation Mapping (Grad-CAM)

Grad-CAM ist ein Ansatz, der von Selvaraju et. al in [15] vorgestellt wurde. Er macht es im Bereich der Bilderkennung möglich, für die jeweilige Klassifizierung wichtige Bildbereiche zu markieren und die Entscheidung transparenter zu gestalten. Selvaraju et. al verfolgen mit Grad-CAM drei wichtige Ziele, die Vorteile für ML-Anwendungen bieten: Erstens dient eine visuelle Darstellung dazu, die CNN-Modelle transparenter und nachvollziehbarer zu gestalten, was dazu führt, dass es ersichtlich wird, dass zum Beispiel unerwartete oder eindeutig falsche Klassifizierungen immer eine sinnvolle Begründung haben. Die erlangte Transparenz führt zum Erreichen des Ziels, dass Vertrauen in ML-Anwendungen geschaffen wird. Es wird möglich, zu verstehen, warum ein Modell das vorhersagt, was es vorhersagt. Das zweite wichtige Ziel ist, dass ein solcher Ansatz Anhaltspunkte für die Verbesserung eines Modells liefern kann. Das in 1.1 genannte Beispiel, in dem Huskys und Wölfe falsch klassifiziert wurden, kann analysiert werden. Wenn die Bildbereiche im Hintergrund als relevant markiert werden, ist ersichtlich, dass das Modell nicht anhand der Eigenschaften des Tieres im Vordergrund entscheidet, sondern aufgrund von Attributen des Hintergrunds. Fehler in Modellen und Trainingsdaten werden aufgedeckt und die Modelle können für noch bessere Ergebnisse optimiert werden. Das dritte Ziel wird als „machine teaching“ beschrieben. Ein denkbarer Anwendungsfall wäre die Nutzung eines Modells in der Medizin: Ein Modell wird mit Aufnahmen aus bildgebenden Verfahren und deren Diagnosen trainiert und ist anschließend in der Lage aus neuen Aufnahmen Diagnosevorschläge zu generieren. In einem solchen Modell ist es durchaus denkbar, dass es wichtige Zusammenhänge in den Bilddaten findet, die

den Forschern bisher noch nicht bekannt waren. Es wird sicherlich immer nötig sein, dass die Diagnose von einem Mediziner gestellt wird, jedoch wäre denkbar, dass das Modell die Aufnahme parallel untersucht. Kommt das Modell zu einem anderen Ergebnis als der Mediziner, hat es entweder einen Fehler oder die Entscheidung beruht auf einem Zusammenhang, der noch nicht erforscht wurde. In beiden Fällen ist es wichtig, dass die Entscheidung des Modells nachvollzogen werden kann. Wenn daraus ersichtlich ist, dass das Modell fehlerhaft ist, kann es verbessert werden - wenn aber ein neuer Zusammenhang sichtbar wird, kann dieser weiter untersucht und vielleicht eine neue wichtige Erkenntnis für die Diagnose einer Krankheit gewonnen werden. Damit hätte die ML-Anwendung dem Menschen etwas beigebracht und das Ziel "machine teaching"wäre erreicht.

4.1 Grad-CAM im Detail

In Kapitel 3 wurde erläutert, dass verschiedene Filter auf einem Bild angewendet und daraus generalisierte Informationen über Kanten und Formen extrahiert werden. In den FC-Layers werden die Informationen noch generalisierter, wobei die räumlichen Informationen aus den Convolutional Layers durch das Flattening verloren gehen. Im Grad-CAM Ansatz wird der letzte Convolutional Layer als Ausgangspunkt verwendet, weil dieser als Kompromiss zwischen der Semantik auf hoher Ebene (also den abstraktesten im Modell verwendeten Informationen) und den detaillierten räumlichen Informationen dient. Damit ist es möglich, für jede beliebige Klasse c eine Karte (Localization Map) der Breite u und Höhe v mit Indizien für die gewählte Klasse zu generieren.

$$L^c_{Grad-CAM} \in \mathbb{R}^{u \times v} \tag{3}$$

Auf dieser Karte sind dann die Bereiche, die für die Klassifizierung in der gewählten Klasse sprechen, zu sehen. In 3.6 wurde beschrieben, dass es im letzten FC-Layer für jede mögliche Klasse ein Neuron gibt, in dem ein Wert, der für die jeweilige Klasse spricht, steht. Diese Werte wurden mittels Softmax zwischen 0 und 1 normiert, sodass sie einen Wahrscheinlichkeitswert darstellen konnten. Im Grad-CAM-Ansatz werden die Werte y des letzten FC-Layers vor Anwendung des Softmax betrachtet. Der Wert im Neuron, das für die gewählte Klasse c spricht, wird y^c genannt.

$$y^c \tag{4}$$

Außerdem werden die Feature-Maps des letzten Convolutional Layers verwendet und jeder Wert in diesen Matrizen betrachtet. Im Beispiel im Projekt wären das für eine Matrix (auch Channel genannt) 14(i = Breite)x14(j = Höhe) Werte. Diese Werte werden Aktivierung A des Channels k genannt.

$$A^k \tag{5}$$

A^k ist somit ein anderer Ausdruck für eine Feature Map des letzten Convolutional Layers. Der genannte Wert y^c wird anschließend partiell nach der Aktivierung abgeleitet. Die Ableitung beschreibt, wie wichtig dieses Pixel eines Channels k für die Klasse c ist. Dies ist vergleichbar mit der Steigung einer Funktion $f(x)$. Bei einer geringen Steigung führt eine geringe Änderung von x zu einer geringen Änderung von y. Bei einer hohen Steigung führt eine geringer Änderung von x zu einer größeren Änderung von y. So beeinflusst die Ausprägung von x je nach Steigung das Ergebnis y. Im beschriebenen Ansatz zeigt das Ergebnis der Ableitung, ob eine geringe Änderung im Pixel eine größere oder kleinere Änderung der Klassifizierung darstellt, also wie wichtig dieses Pixel eines Channels k für die Klasse c ist.

$$w_i = \frac{\partial y^c}{\partial A^k_i} \tag{6}$$

Dies wird für jede Aktivierung in diesem Channel (Breite x Höhe) durchgeführt. Anschließend werden alle Werte (Breite i x Höhe j) aufsummiert und ein Durchschnitt gebildet (sog. Global Average Pooling). Dadurch wird eine gewichtete Aktivierung für den Channel in Bezug auf die Klasse c gebildet.

$$\alpha^c_k = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A^k} \tag{7}$$

Der entstandene Wert α^c_k beschreibt die Wichtigkeit bzw. das Gewicht einer Feature Map k für die Zielklasse c. Die Wichtigkeit eines Wertes der Feature Map wird mit dem Wert in der Feature Map multipliziert, sodass der Wert gewichtet wird. Damit wird die Wichtigkeit des konkreten Werts im Pixel für die Klasse c bestimmt.

$$\alpha^c_k A^k \tag{8}$$

Wenn das für jeden Wert in einer Feature-Map durchgeführt wird, wird dieser Wert mit der Wichtigkeit für die Klasse gewichtet. Der gesamte Prozess wird für jede Feature Map des letzten Convolutional Layers (im VGG16 Beispiel sind das 512 Feature-Maps) durchgeführt und die Werte an denselben Stellen der verschiedenen Matrizen werden aufsummiert. Die negativen Werte werden mit der ReLU-Funktion zu Nullwerten transformiert, weil nur die Bereiche betrachtet werden sollen, die einen positiven Einfluss auf die Klasse haben; die negativen Einflüsse werden somit ausgefiltert.

$$L^c_{Grad-CAM} = ReLU(\sum_k \alpha^c_k A^k) \tag{9}$$

Die entstandene Matrix ist nun genauso groß wie die Matrizen des letzten Convolutional Layers und beinhaltet die für die Klassifizierung der jeweiligen Klasse relevanten Bereiche. Die Werte werden für die Visualisierung zudem zwischen 0 und 1 normiert. Im zugehörigen Projekt ist die Matrix 14x14 Pixel groß und wird im Nachhinein an die Größe des Originalbilds angepasst.

4.2 Counterfactual Explanations

Grad-CAM wird in [15] nicht nur als Modell beschrieben, in dem analysiert werden kann, welche Bereiche des Bilds für eine ausgewählte Klasse relevant sind, sondern auch ein Ansatz vorgestellt, der Gegenteiliges möglich macht. Es ist durch folgende Manipulation möglich, Bereiche des Bilds, die gegen die gewählte Klasse sprechen, hervorzuheben:

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j -\frac{\partial y^c}{\partial A^k} \tag{10}$$

Damit kann nun bei falscher Klassifizierung eines Inhaltes analysiert werden, welche Bereiche gegen die erwartete Klasse sprechen, um zu verstehen, warum die Entscheidung getroffen wurde und ggf. Fehler im Modell zu finden. Dieses Konzept kann auch veranschaulicht werden, indem ein Bild zur Klassifizierung verwendet wird, auf dem zwei Inhalte verschiedener Klassen abgebildet sind. Wenn das Modell eine der beiden Klassen liefert, kann mit dem ersten Ansatz analysiert werden, welche Bereiche für die Klassifizierung wichtig waren. Mit dem modifizierten Ansatz ist zu sehen, welche Bereiche des Bilds gegen die Klassifizierung der jeweiligen Klasse gesprochen hätten - also die Bereiche mit dem anderen Gegenstand. In 4.3 ist ein Beispiel zu diesem Fall abgebildet.

4.3 Evaluation der Visualisierung

In diesem Kapitel werden die Ergebnisse des Teilprojekts Grad-CAM demonstriert. Hierzu wurden verschiedene Bilder durch das bereits beschriebene VGG16-Modell klassifiziert und die entsprechenden Heatmaps mit dem Grad-CAM Verfahren generiert. In Abbildung 13 ist ein Beispiel zu sehen, bei dem die Klassifizierung eines Golden Retrievers erfolgreich durchgeführt wurde. In diesem Beispiel wird die Entscheidung nachvollziehbar dargestellt: Es ist zu sehen, dass insbesondere die Ohren in ihrer Ausprägung wichtige Bereiche für die Klassifizierung als Golden Retriever sind.

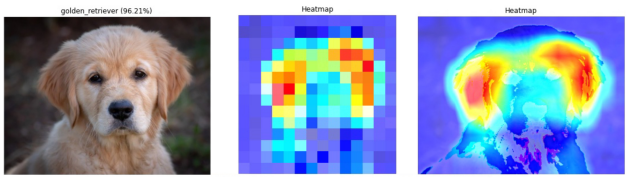


Abbildung 13. Localization Map für einen Golden Retriever vor und nach Anpassung auf Originalgröße

In Abbildung 14 ist ein ungarischer Vorstehhund (Vizsla) abgebildet. Auch dieses Bild wurde fehlerfrei klassifiziert. Die Entscheidung ist vorwiegend aufgrund der Pixel im Bereich der Schnauze des Tieres gefallen.

Als letztes Beispiel für eine Karte bei fehlerfreier Klassifizierung soll Abbildung 15 dienen. In diesem Bild wurde

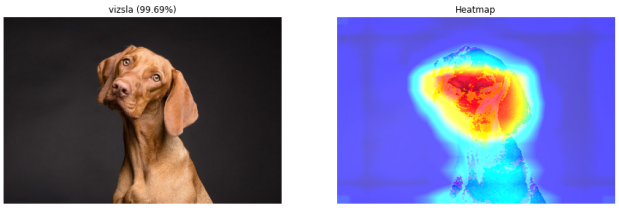


Abbildung 14. Localization Map für einen Ungarischen Vorstehhund (Vizsla)

ein Zebra fehlerfrei klassifiziert und die ausschlaggebenden Bereiche eingefärbt.

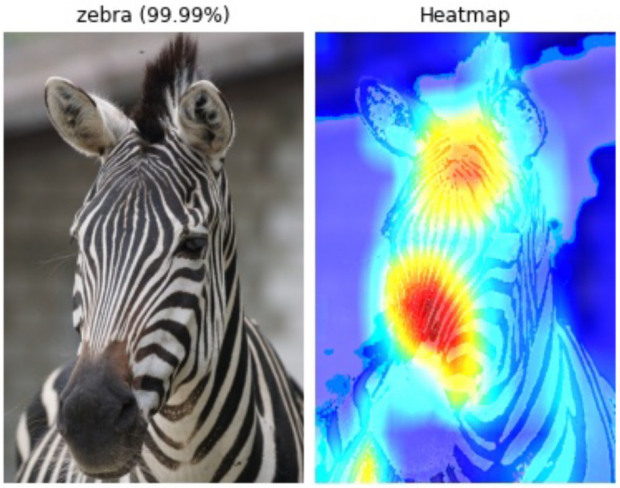


Abbildung 15. Localization Map für ein Zebra

Das Konzept „Counterfactual Explanations“ ist mit einem Bild, auf dem zwei verschiedene Objekte abgebildet sind, am besten demonstrierbar. In Abbildung 16 ist ein solches Beispiel abgebildet. Die Anwendung klassifiziert einen Golden Retriever auf dem Bild, abgebildet sind aber sowohl ein Hund als auch eine Katze. Wenn Grad-CAM mit der Klasse Golden Retriever aufgerufen wird, werden die Bereiche eingefärbt, die für die Klassifizierung als Golden Retriever ausschlaggebend sind. Wird das Vorzeichen wie in 4.2 beschrieben umgedreht, werden die Bereiche, die gegen die Klasse sprechen, eingefärbt. Interessant ist hierbei, dass ebendie Bereiche, auf denen die Katze abgebildet ist, relevant werden. Zum Vergleich wurde nur ein Ausschnitt des Bilds genommen und Grad-CAM mit der Klasse „Persian Cat“ ausgeführt. Hier sind in etwa dieselben Bereiche ausschlaggebend, die im Gesamtbild unter Counterfactual Explanations relevant waren.

Das Ziel der Fehleranalyse kann erreicht werden, wenn man falsch klassifizierte Bilder betrachtet. In Abbildung 17 ist ein Beispiel zu sehen, bei dem eine Ameise als Fliege klassifiziert wurde. Zum Vergleich zeigt Abbildung 18 eine

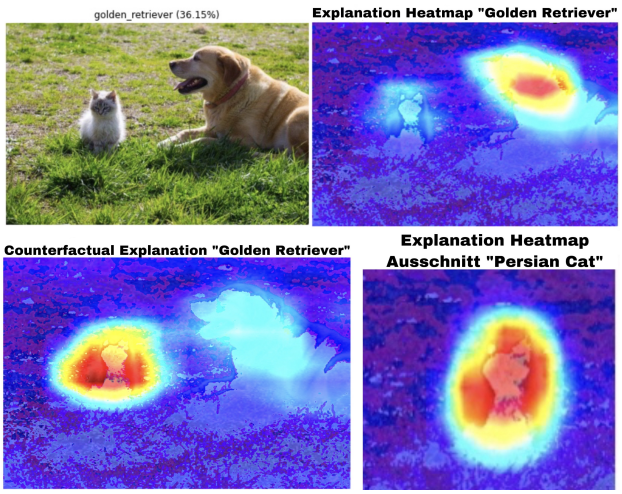


Abbildung 16. Heatmaps für ein Bild mit einem Hund und einer Katze

Fliege mit der richtigen Klassifikation. Wenn man die beiden Bilder betrachtet, fällt auf, dass derselbe Bereich des Körpers ausschlaggebend ist. Man könnte vermuten, dass das Modell in Bezug auf die Klasse „Ameise“ hauptsächlich mit Ameisen trainiert wurde, die keine Flügel haben. Aus diesem Grund könnte die Ameise mit Flügeln und ihrer Ähnlichkeit zur Fliege als solche klassifiziert worden sein. Zur Überprüfung könnten für ein weiteres Training mehr Bilder von Ameisen mit Flügeln verwendet und geprüft werden, ob das Modell anschließend sowohl geflügelte als auch Ameisen ohne Flügel von Fliegen unterscheiden kann.

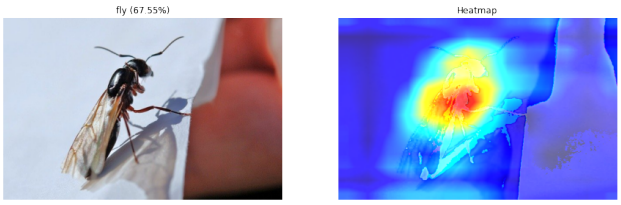


Abbildung 17. Heatmap für eine Ameise, die als Fliege klassifiziert wurde

Besonders interessant sind Beispiele, in denen Insekten von anderen Insekten nachgeahmt werden. So wird zum Beispiel eine Schwebfliege, die einer Wespe ähnelt als Biene klassifiziert. Wenn man betrachtet, welche Bildbereiche dazu geführt haben, stellt man fest, dass die Taille dieser speziellen Fliege derer der Biene ähnelt. Aufgrund der Verwandtschaft zur Fliege und der visuellen Ähnlichkeit zur Wespe ist es besonders spannend, zu sehen, was die Klassifizierung ergibt und wieso sie es tut.

Ein weiteres interessantes Beispiel zur Demonstration von „Counterfactual Explanations“ kann generiert werden, indem

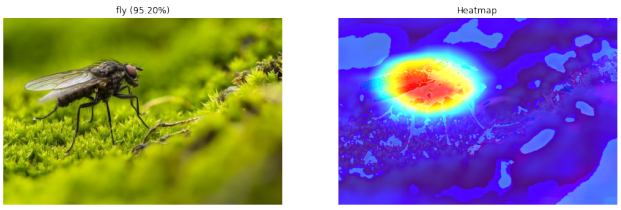


Abbildung 18. Heatmap für eine Fliege, die als solche klassifiziert wurde

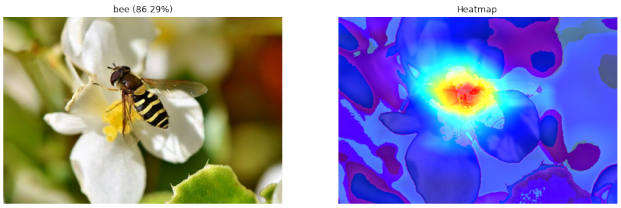


Abbildung 19. Heatmap für eine Schwebfliege, die als Biene klassifiziert wurde

Abbildung 20. Heatmap für eine Biene, die als solche klassifiziert wurde

ein Bild verwendet wird, auf dem zwei Objekte vermischt sind. Wenn zum Beispiel der Kopf des Zebras aus Abbildung 15 auf den Hundekörper aus Abbildung 14 kopiert wird und das Modell angewendet wird, klassifiziert es das Bild als Zebra. In Abbildung 21 wurde Grad-CAM mit der Klasse Zebra verwendet. Es ist zu sehen, dass die Ausprägung der Schnauze sehr wichtig für die Klassifizierung als Zebra ist. Nach dem Vorzeichenwechsel ist aber auch zu sehen, dass im Modell bekannt ist, dass der Körper des Hundes nicht charakteristisch für einen Zebrakörper ist. Die Ausprägung des Hundekörpers spricht aus diesem Grund gegen die Klassifizierung als Zebra. Außerdem wurde in Abbildung 22 Grad-CAM mit der Klasse ungarischer Vorstehhund (Vizsla) angewendet. In diesem Beispiel ist ersichtlich, dass die Körperbereiche, die schon in Counterfactual Explanations ersichtlich waren, auch für diese Hunderasse sprechen. Es ist aber auch eindeutig zu sehen, dass die Aktivierungen für das Zebra stärker sind als für die Hunderasse. Das spricht dafür, dass das Modell aufgrund des Körpers eine andere Hunderasse klassifiziert, wenn nur der Körperausschnitt verwendet wird. Die neue Klassifizierung und

deren Heatmap sind ebenfalls in Abbildung 22 abgebildet.

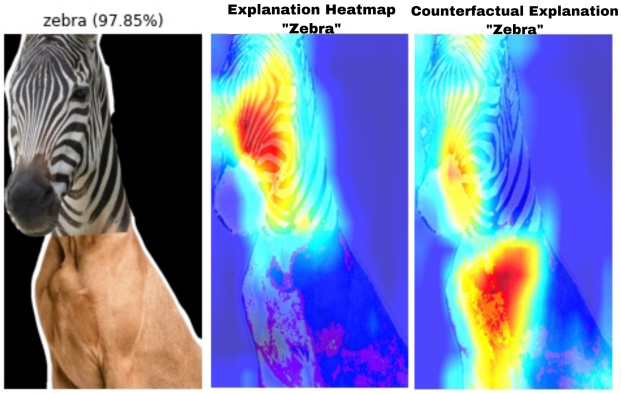


Abbildung 21. Klassifizierung und Heatmaps für ein Bild mit Zebrakopf und Hundekörper

Abbildung 22. Heatmap für die Klasse „Vizsla“ und neue Klassifizierung des Körpers mit erklärender Heatmap

5 Fazit und Ausblick

Die vorliegende Arbeit hat gezeigt, woraus ein CNN für Bildklassifizierung bestehen kann und wie der Klassifizierungsprozess funktioniert. Hierzu wurde der Aufbau in Schichten und die Funktionsweise jeder Schicht erläutert. Um den Prozess verständlicher zu gestalten, wurden Ergebnisse aus ausgewählten Schichten des VGG16 Modells in der Arbeit demonstriert und im Praxisprojekt jedes Zwischenergebnis aus jeder Schicht gespeichert. Um Klassifizierungsentscheidungen nachvollziehbarer zu machen, wurde das Verfahren Grad-CAM vorgestellt und im Praxisprojekt implementiert. Die Ergebnisse dieses Verfahrens sind Heatmaps, welche ausschlaggebende Bereiche, die für eine gewählte Klasse sprechen, einfärben. Mit dem Konzept der Counterfactual Explanations wurde zudem eine Möglichkeit dargestellt, durch die erkannt werden kann, welche Bildbereiche gegen die definierte Klasse sprechen. Zuletzt wurde Grad-CAM evaluiert, indem verschiedene Bilder klassifiziert und Heatmaps für und gegen die jeweilige Klasse erstellt wurden. Interessant sind auch Versuche mit Sonderfällen, die behandelt wurden; dazu gehören zum Beispiel mehrere Objekte auf einem Bild oder verschmolzene Gegenstände. Es konnten viele Erfahrungen in Bezug auf CNNs und die Erklärbarkeit durch Grad-CAM gesammelt werden. Es gibt viele weitere interessante Ansätze, die berücksichtigt werden können, wenn CNNs visualisiert werden sollen. Ansätze wie CNN Fixations [11] oder CAM [19] sind in diesem Zusammenhang zu nennen. Eine interessante Idee für eine Fortführung dieser Arbeit wäre es, verschiedene Methoden der Visualisierung mit unterschiedlichen CNN-Modellen anzuwenden und zu evaluieren. Auch die Analyse selbsterstellter und mit eigenen Trainingsdaten trainierter Modelle oder die Verwendung von bestehenden Modellen, mit denen ein Transfer-Learning stattgefunden hat, wären außerst interessant und könnten betrachtet werden.

6 Anhang

Literatur

[1] Roland Becker. 2018. *MACHINE / DEEP LEARNING – WIE LERNEN KÜNSTLICHE NEURONALE NETZE?* Retrieved 01.07.2020 from https://jaai.de/machine-deep-learning-529/

[2] Roland Becker. 2019. *CONVOLUTIONAL NEURAL NETWORKS – AUFBAU, FUNKTION UND ANWENDUNGSGEBIETE.* Retrieved 06.06.2020 from https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/

[3] Jason Brownlee. 2019. *How to Visualize Filters and Feature Maps in Convolutional Neural Networks.* Retrieved 06.06.2020 from https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09.*

[5] Martin Gorner. [n.d.]. Tensorflow and deep learning without a PhD. Retrieved 21.06.2020 from https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd

[6] Hochschule Hof. 2020. Modulhandbuch Wirtschaftsinformatik.

[7] Jiwon Jeong. 2019. *The Most Intuitive and Easiest Guide for Convolutional Neural Network.* Retrieved 06.06.2020 from https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480

[8] Raphael Meudec. [n.d.]. Grad CAM implementation with Tensorflow 2. Retrieved 21.06.2020 from https://gist.github.com/RaphaelMeudec/e9a805fa82880876f8d89766f0690b54

[9] Divyanshu Mishra. 2019. Demystifying Convolutional Neural Networks using GradCam. Retrieved 01.07.2020 from https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-gradcam-554a85dd4e48

[10] Julian Moeser. 2018. *KÜNSTLICHE NEURONALE NETZE – AUFBAU UND FUNKTIONSWEISE.* Retrieved 06.06.2020 from https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/

[11] Konda Reddy Mopuri, Utsav Garg, and R. Venkatesh Babu. 2017. CNN Fixations: An unraveling approach to visualize the discriminative image regions. arXiv:cs.CV/1708.06670

[12] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. arXiv:cs.LG/1602.04938

[13] Rutger Ruizendaal. 2017. Deep Learning #3: More on CNNs & Handling Overfitting. Retrieved 01.07.2020 from https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d

[14] Sumit Saha. 2018. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* Retrieved 06.06.2020 from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[15] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2019. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. *International Journal of Computer Vision* 128, 2 (Oct 2019), 336–359. https://doi.org/10.1007/s11263-019-01228-7

[16] Avinash Sharma. 2017. Understanding Activation Functions in Neural Networks. Retrieved 01.07.2020 from https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[17] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:cs.CV/1409.1556

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. arXiv:cs.CV/1409.4842

[19] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2015. Learning Deep Features for Discriminative Localization. arXiv:cs.CV/1512.04150

[20] Victor Zhou. 2019. Training a Convolutional Neural Network from scratch. Retrieved 01.07.2020 from https://towardsdatascience.com/training-a-convolutional-neural-network-from-scratch-2235c2a25754

Objekterkennung mit YOLOv3

Alexander Puchta
alexander.puchta@hof-university.de
Hochschule Hof
Hof an der Saale, Germany

ZUSAMMENFASSUNG

Im Sommersemester 2020 an der Hochschule in Hof wurden im Wahlfach Angewandtes maschinelles Lernen unterschiedliche Varianten des Machine Learnings erlernt. Ziel dieser Arbeit ist es, mit einer dieser Lernmethoden zu arbeiten. Im Bezug auf die heutige Entwicklung des autonomen Fahrens, fiel die Entscheidung, überwachtes Lernen auszuwählen und dadurch eine künstliche Intelligenz zu trainieren, die es ermöglicht, dass Verkehrszeichen erkannt werden können. Dieses Paper befasst sich mit den Frameworks Darknet und YOLO, welche aktuell weit verbreitet sind und immer wieder auftauchen, wenn es um Objekterkennung in Echtzeit geht. Darknet ermöglicht das generieren von Gewichtungen für vorbereitete Datasets, welche mittels Transferlernen erzeugt werden. Im Rahmen dieser Studienarbeit wurden Verkehrszeichen verwendet, die in Deutschland üblich sind und mittels einer Software, die Labels und Boxen erzeugt, für das Training brauchbar gemacht. YOLO verwendet die zuvor generierten Gewichtungen von Darknet und ermöglicht somit eine Angabe, wo sich Objekte, in diesem Fall Verkehrszeichen, befinden und zeigt diese mittels Bounding Boxes an. Bounding Boxes grenzen ein Objekt in etwa so ein, wie es zuvor durch die Label-Software vorhergesagt wurde. Jedoch sind Ergebnisse hier oft unterschiedlich, da es abhängig von der Bildqualität ist, wie gut eine Box vorhergesagt werden kann. Alle Ergebnisse und Fortschritte wurden durch die Verwendung des Google-Colab Jupiter Notebooks erzielt. Basis für dieses Paper ist die Arbeit 'How to train YOLO'[6] von Quang Nguyen.

1 EINLEITUNG

Oftmals wird mit dem Auto über Straßen gefahren und dabei so sehr nachgedacht, dass es in Vergessenheit gerät, welches Tempo hier erlaubt ist. Dies kann zu enormen Folgen führen, denn der Führerschein ist seit den neusten Bußgeldkataloganpassungen schneller weg, als es lieb ist. Moderne Autos haben bereits eine Funktion enthalten, die über den Boardcomputer angibt, welches Tempolimit gerade einzuhalten ist. Jedoch sind diese Angaben oft nur über das Navigationssystem eingepflegt und seltenst aktualisiert. Um hierfür eine vielversprechende Lösung zu finden, eignet sich das maschinelle Lernen sehr gut. Was ist maschinelles Lernen eigentlich? Dr. Klaus Manhart meint, dass es ganz einfach ausgedrückt das maschinelle Lernen und das selbstständige Weiterentwickeln eines Computers, oder einer Software, ist, ohne dies explizit vorzugeben.[4] Sozusagen eine eigenständige Entwicklung von Wissen aus bisherigen Erfahrungen. Dies wird durch die Vorgabe von Beispielen, bzw. Datasets ermöglicht. Ein solches Dataset sollte, um gute Ergebnisse zu liefern, eine große Menge an Informationen enthalten. Hier explizit sollte eine große Menge an Bildern von Verkehrszeichen vorgehalten sein. Aufbereitet mit Labels, sozusagen einer Angabe, wo sich auf einem solchen Bild ein Verkehrszeichen befindet, ist das Dataset einsatzbereit, so dass das Lernen starten kann. Nachfolgend werden die einzelnen Schritte dieses Prozesses erklärt.

2 GRUNDLAGEN

Machine Learning ist ein Teilgebiet der künstlichen Intelligenz. Mittels vorgehaltenen Mustern aus Datenbeständen, können Computer eigenständige Lösungen entwickeln.
Durch die Verwendung von bereits entwickelten Algorithmen können eigene Datensets, in diesem Fall Verkehrsschilder, implementiert, diese zum trainieren verwendet werden und dadurch den Algorithmus verbessern. Machine Learning wird hierbei jedoch in unterschiedliche Gruppen eingeteilt. Die Objekterkennung, hier mit YOLO, ist der Gruppe des überwachten Lernens zugewiesen. Nachfolgend werden alle Varianten aufgelistet.

Machine Learning Lernstile[3]

(1) überwachtes Lernen
(2) unüberwachtes Lernen
(3) teilüberwachtes Lernen
(4) bestärkendes Lernen
(5) aktives Lernen

3 YOLO

Ein klassisches Problem in der Bildverarbeitung ist die Objekterkennung - besonders wenn es darum geht, bestimmte Objekte in einem bestimmten Bild zu finden. Die Objekterkennung ist ein weitaus schwerwiegenderes Thema als die Klassifizierung, bei der auch Objekte erkannt werden können, allerdings ohne Angabe, wo sich diese befinden.

YOLO, oder auch "You Only Look Once", arbeitet mit einem anderen Ansatz als die klassische Klassifizierung oder Objekterkennung. Da es ein cleveres CNN (Convolutional Neural Network) ist, wendet es ein einzelnes neuronales Netzwerk auf das komplette Bild an, spaltet dieses in mehrere Regionen und kann dadurch Eingrenzungen und Wahrscheinlichkeiten für diese angeben. YOLO ermöglicht somit eine Kombination aus Klassifizierung und einer Art Objekterkennung.

Der YOLO Algorithmus wird so häufig verwendet, da zum einen eine hohe Genauigkeit erzielt wird und zum anderen das ganze in Echtzeit laufen kann. Der Algorithmus betrachtet jedes Bild nur einmal, das heißt, dass es nur einen Durchlauf im neuronalen Netzwerk benötigt, um eine Vorhersage zu treffen. YOLO gibt dann die erkannten Objekte zusammen mit den Begrenzungsrahmen und der Wahrscheinlichkeiten aus, siehe Abb. 1.

3.1 Funktionalität

Wie bereits oben genannt, sucht der YOLO Algorithmus nicht nach interessanten Gebieten oder Regionen. Der Algorithmus teilt jedes Bild in Zellen, üblicherweise wird hier ein 19 x 19 Raster verwendet. Jede dieser Zellen ist für eine Vorhersage von 5 Bounding Boxes verantwortlich.

Abbildung 1: Bounding Box Vorfahrtschild

Eine Bounding Box, siehe Abb. 2 kann mit folgenden Variablen beschrieben werden:

(1) Mittelpunkt der Box (bx, by)
(2) Breite der Box (bbreite)
(3) Höhe der Box (bhoehe)
(4) Klasse des Objekts (k)

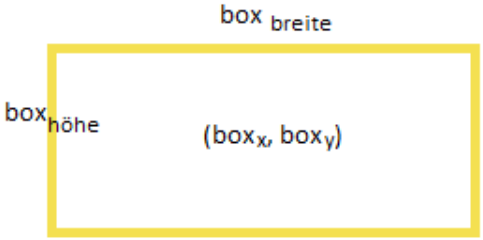


Abbildung 2: Bounding Box Definition

Des weiteren wird auch die Wahrscheinlichkeit (w) benötigt, mit der diese Box vorhergesagt wurde. Eine solche Box könnte durch die Verwendung der zuvor genannten Variablen zum Beispiel durch $objekt = (w_k, box_x, box_y, box_{breite}, box_{hhe}, k)$ bestimmt werden.

3.2 Klassifizierung

Bei der Klassifizierung möchte man mittels Beobachtung eine Entscheidung treffen, zu welcher Klasse ein beobachtetes Objekt gehört.
Als Mensch fällt man immer zu Klassifikationsentscheidungen: Ist dieses T-Shirt gelb? Was für ein Schuh ist das? Ist die Ampel rot? Ist das ein Vorfahrtsschild? Diese Fragen haben alle einen gemeinsamen Punkt: Hier gibt es keine einfache beschreibbare Herangehensweise. Hier wird ein vielschichtiger Input benötigt und oftmals kommt es zu Situationen, in denen dies nicht so einfach entschieden werden kann. Einen kleinen Einblick gibt Abb. 3, wie eine solche Entscheidung bei der Klassifizierung gefällt wird.

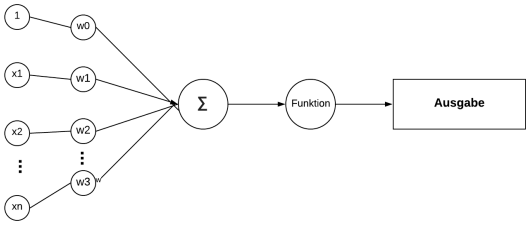


Abbildung 3: Klassifizierung

3.3 Objekterkennung

Im Gegensatz zur Klassifizierung ist das Ziel der Objekterkennung, dass nicht nur gesagt wird, welcher Klasse dieses Bild angehört, sondern auch, wo sich dieses Objekt in diesem Bild befindet. Demnach ist die Objekterkennung eine Erweiterung der Klassifizierung. Um dies jedoch vollständig leisten zu können, werden für die Objekterkennung große Mengen an Datensätzen benötigt. Hierzu gehört unter anderem ein großer Satz an Test-Daten und dazugehörige Validierungs-Daten. Ein solcher Datensatz ist unter anderem eine Bilddatei und eine dazugehörige CSV-Datei, die die Position des Objekts in der dazugehörigen Bilddatei speichert.

4 VERKEHRSZEICHENERKENNUNG

Da das Thema autonomes Fahren immer aktueller und interessanter wird, werden Maschinen benötigt, die eine Echtzeiterkennung der Verkehrszeichen ermöglichen. Der Basis-Datensatz des Frameworks Darknet[7] ermöglicht zwar, dass Ampeln erkannt werden können, jedoch ist dieser nicht auf die in Deutschland verwendeten Verkehrsschilder trainiert. Die Grundlage für diese Studienarbeit ist das vortrainierte Dataset "darknet53"[1].

4.1 Datensatz

Wie bereits zuvor genannt, basiert diese Studienarbeit auf dem vortrainierten Datensatz des Frameworks Darknet. Hierzu werden einige Verkehrszeichen[5] hinzugefügt. Aktuell wird der Datensatz auf die Verkehrsschilder 'Maximal 20 km/' und 'Maximal 30 km/h' begrenzt. Grund hierfür ist, dass der Lernvorgang eine enorme Zeit beansprucht und somit geringere Ergebnisse erzielt werden hätten können.

4.1.1 Vorbereitung. Die Bilder benötigen Bounding Boxes und die Information, wo sich diese Bounding Box befindet. Hierfür wurde das Programm Yolo Label[2] verwendet, welches frei zugänglich auf github zur Verfügung steht. Nachfolgend wird der Vorgang des Markierens beschrieben.

Abbildung 4: Vorgang Label erstellen

Abbildung 5: Vorgang Label erstellen

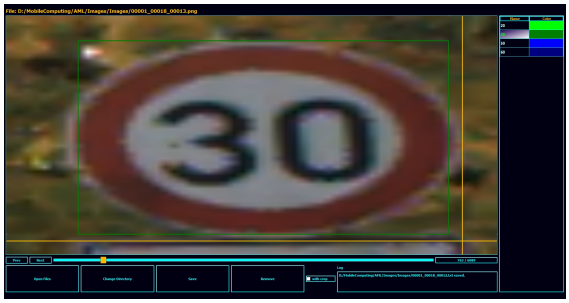


Abbildung 6: Vorgang Label erstellen

Abb. 4 zeigt, dass ein Punkt ausgewählt wird, so dass eine horizontale und vertikale Linie das gesamte Objekt einrahmen. Das Programm Yolo Label[2] ermöglicht es, dass hier nun ein Klick auf die linke Maustaste genügt, um eine Markierung zu ziehen. Auf Abb. 5 sieht man nun, dass das gesamte Objekt einen Rahmen erhält. Durch erneutes Klicken der linken Maustaste kann das Label gesetzt werden. Betrachtet man Abb. 6, so erkennt man, dass ein Rahmen in dunkelgrün erstellt wurde. Dieser entspricht einer Markierung der Klasse '30'.

4.2 Lernen

Mittels den zuvor markierten Bildern, kann nun das Framework mit diesem Datenset bestückt werden. Aktuell werden 1200 Bilder und die dazugehörigen 1200 Label verwendet. Da der in Abschnitt 4.1.1 erklärte Vorgang sehr langwierig ist, ist aktuell erst diese kleine Menge an Bildern und Labeln in diesem Datensatz. Der geschätzte Zeitaufwand für das Erstellen der Label liegt bei sieben bis acht Stunden.

4.2.1 Vorbereitung. Im Darknet Ordner muss eine .cfg erstellt werden, die dem Lernprozess die nötigen Einstellungen bereitstellt. In diesem Projekt wird eine alte yolov3.cfg kopiert und in yolov3-traffic-train.cfg umbenannt. In dieser Datei müssen einige Angaben verändert werden, siehe Abb. 7.

```
# Line 8 & 9:
width = 416, height = 416

# Line 20
max_batches = 2278 // Anzahl Bilder + Label

# Line 22:
steps = 5400

# Line 603, 689, 776:
filters = 21 // (classes + 5) * 3

# Line 610, 696, 783:
classes = 2
```

Abbildung 7: yolov3-traffic-train.cfg

Das Framework benötigt ca. 5 Prozent des Datensatzes zum Validieren, der Rest der Daten wird zum Trainieren verwendet. Hierfür wird eine train.txt und val.txt benötigt. Mittels eines kleinen Hilfsprogramms wurden die Dateien zufällig auf die beiden Text-Dateien aufgeteilt.

Zuletzt wird eine yolo.data, siehe Abb. 8, erstellt, die alle Informationen und Verweise zum Datensatz enthält.

```
classes = 2
train = data/train.txt
valid = data/val.txt
names = data/yolo.names
backup = backup
```

Abbildung 8: yolo.data

Sind all diese Schritte abgeschlossen, so kann der Ordner darknet auf github geladen, oder aber komprimiert werden, so dass er z.B. auf Google-Drive verwendet werden kann.

4.2.2 Training + Testen. Nach 1000 Lernvorgängen wurde ein erster Testversuch gestartet. Hierbei wurden jedoch noch zu häufig Fehler produziert, so dass keine Aussage getroffen werden konnte, ob und wie genau ein Objekt erkannt wurde.
Nach 1300 Lernvorgängen konnten erste Erfolge erzielt werden, welche nachfolgend dokumentiert sind. Abb. 9 und 11 zeigen, dass das Schild bereits erkannt wird, die Box um dieses Verkehrsschild jedoch noch etwas zu großzügig gezeichnet ist.

Nach absolvieren von weiteren 200 Lernvorgängen konnte die Software schon eine Verbesserung vorweisen. Nachfolgend kann in Abb. 11 und Abb. 12 gesehen werden, dass die Verkehrsschilder nun vollständig abgedeckt sind, hierbei jedoch noch ein großer Spielraum vorhanden ist. Dieser sollte durch weiteres Trainieren

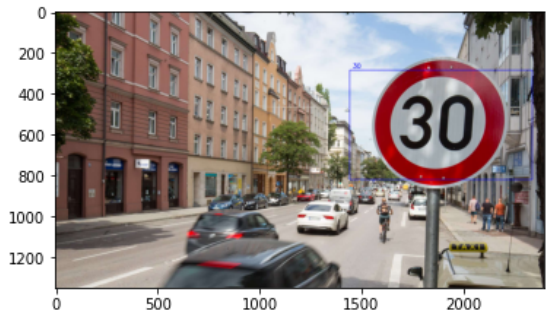


Abbildung 9: YOLO nach 1300 Lernepochen, Beispielbild 1

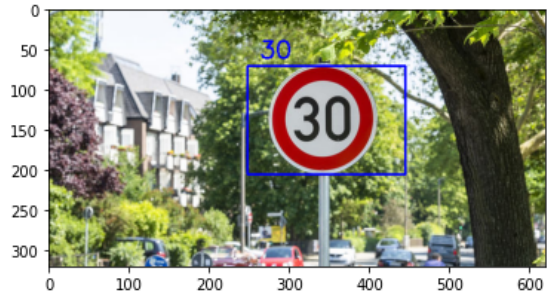


Abbildung 10: YOLO nach 1300 Lernepochen, Beispielbild 2

verbessert werden.

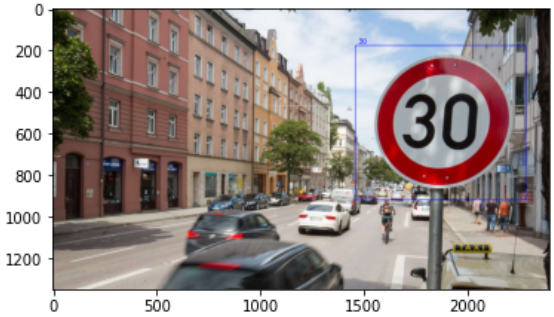


Abbildung 11: YOLO nach 1500 Lernepochen, Beispielbild 1

4.2.3 Ausgabe. Während des Lern- und Trainingsvorgangs wirft das Programm einige Informationen aus. Abb. 14 zeigt einen Ausschnitt.

Die in Abb. 13 gezeigte Ausgabe enthält Informationen über die aktuelle Trainingsepoche. Hier in diesem Fall Lernvorgang 1761. Die Zahl 0.145151 ist die Angabe des Verlusts. Die Ausgabe 0.068448 avg ist die Angabe des Durchschnittsverlusts. Die Zahl vor rate gibt die Lernrate an, welche in der Konfigurationsdatei eingestellt wurde. Die Sekundenangabe gibt an, wie lange dieser Lernvorgang

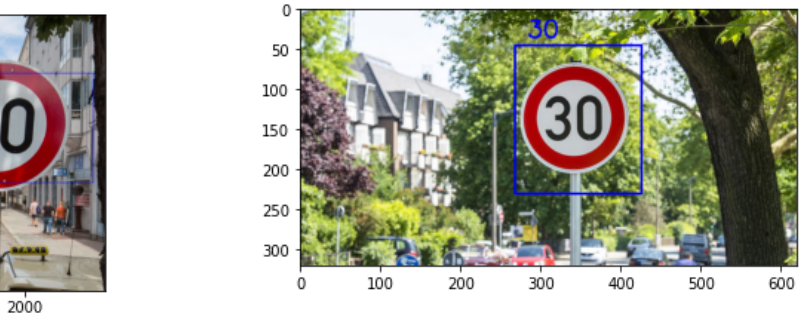
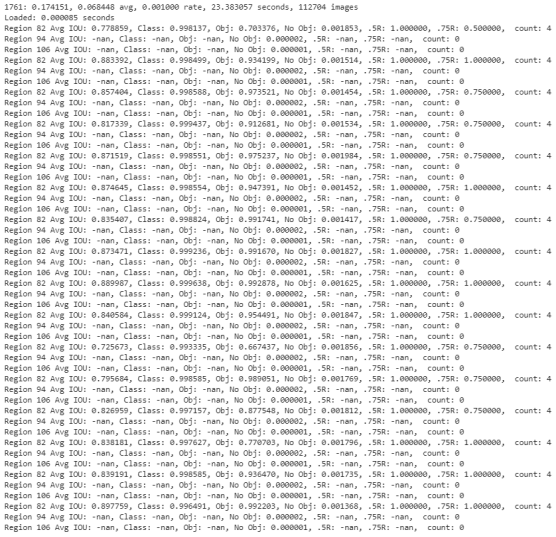


Abbildung 12: YOLO nach 1500 Lernepochen, Beispielbild 2

```
1761: 0.145151, 0.068448 avg, 0.001000 rate, 23.383057
    seconds, 112704 images
```

Abbildung 13: Code-Ausschnitt Lernvorgang

gedauert hat und abschließend wird die Menge der verwendeten Bilder ausgegeben.



```
1761: 0.174151, 0.068448 avg, 0.001000 rate, 23.383057 seconds, 112704 images
Loaded: 0.000085 seconds
Region 82 Avg IOU: 0.778859, Class: 0.998137, Obj: 0.703376, No Obj: 0.001853, .5R: 1.000000, .75R: 0.500000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.883392, Class: 0.998499, Obj: 0.934199, No Obj: 0.001514, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.857404, Class: 0.998588, Obj: 0.973521, No Obj: 0.001454, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.817339, Class: 0.999437, Obj: 0.912681, No Obj: 0.001534, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.871519, Class: 0.998551, Obj: 0.975237, No Obj: 0.001984, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.874645, Class: 0.998554, Obj: 0.947391, No Obj: 0.001452, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.835407, Class: 0.998824, Obj: 0.991741, No Obj: 0.001417, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.873471, Class: 0.999236, Obj: 0.991670, No Obj: 0.001827, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.889987, Class: 0.999638, Obj: 0.992878, No Obj: 0.001625, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.840584, Class: 0.999124, Obj: 0.954491, No Obj: 0.001847, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.725673, Class: 0.993335, Obj: 0.667437, No Obj: 0.001856, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.795684, Class: 0.998585, Obj: 0.989051, No Obj: 0.001769, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.826959, Class: 0.997157, Obj: 0.877548, No Obj: 0.001812, .5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.838181, Class: 0.997627, Obj: 0.770703, No Obj: 0.001796, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.839191, Class: 0.998585, Obj: 0.936470, No Obj: 0.001735, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: 0.897759, Class: 0.996491, Obj: 0.992203, No Obj: 0.001368, .5R: 1.000000, .75R: 1.000000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
```

Abbildung 14: Screenshot Ausgabe Training

4.3 Ergebnis

Mit Beenden dieses Papers, war der letzte Lernvorgang bei 2000 Durchgängen. Das Ergebnis wird nachfolgend geschildert.

Durch das Verwenden eines kleineren Datensatzes kann keine eindeutige Aussage getroffen werden. Jedoch wird bereits nach einigen Durchgängen klar, dass das Programm intelligenter wird. Abb. 9 zeigt, dass das Schild nicht wirklich getroffen wird, jedoch nach 200 Lernvorgängen mehr konnte ein Rahmen gezogen werden, der das Schild komplett umfasst, siehe Abb. 11.

Werden mehrere Bilder hinzugefügt, würde die Dauer für das Lernen vermutlich potenziell ansteigen, aber dafür die Ergebnisse deutlich verbessern.

```
Region 82 Avg IOU: 0.897759, ...
```

Abbildung 15: Code-Ausschnitt aus Abbildung 14

Wenn man bedenkt, dass die Angabe in Abb. 15 zeigt, dass aktuell eine Genauigkeit von ca. 89 Prozent erreicht wird und das, obwohl nur ein Datensatz von 1140 Bildern benutzt wird, sollte man annehmen können, dass ein größerer Datensatz eine noch eindeutigere Ausgabe liefert.

Nachfolgend nochmals die Testbilder mit den erlernten Gewichtungen nach 2000 Durchläufen. Vergleicht man Abb. 9 mit Abb. 16, so erkennt man einen enormen Anstieg der Genauigkeit. Abb. 17 weist auch eine höhere Genauigkeit der Box vor im Gegensatz zur Abb. 10.

Abbildung 16: YOLO nach 2000 Lernepochen, Beispielbild 1

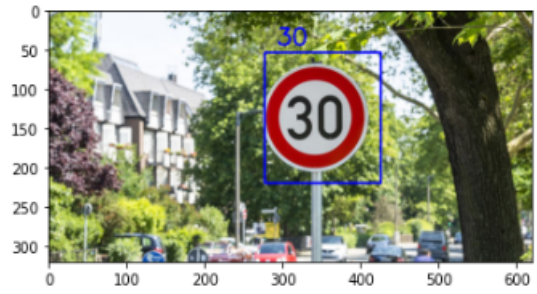


Abbildung 17: YOLO nach 2000 Lernepochen, Beispielbild 2

5 AKTUELLER STAND

Aktueller Stand liegt bei 2278 Durchgängen. Die letzten Tests wurden mit diesen erlernten Gewichtungen gemacht und haben folgende Ergebnisse erbracht.
Abb. 18 und Abb. 19 zeigen, dass die Verkehrsschilder komplett erkannt werden und das richtige Label gesetzt wird.

Abbildung 18: YOLO Final Ergebnis Bild 1

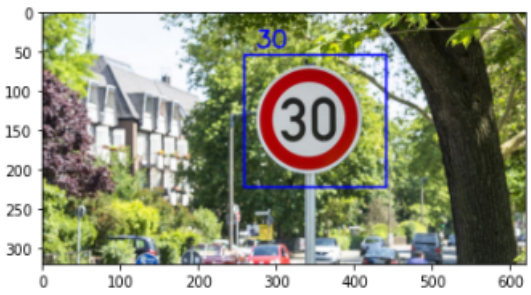


Abbildung 19: YOLO Final Ergebnis Bild 2

LITERATUR

[1] [n.d.]. Darknet pre-trained Weights. https://pjreddie.com/media/files/darknet53.conv.74. Accessed: 07.06.2020.
[2] [n.d.]. Yolo Label Github. https://github.com/developer0hye/Yolo_Label. Accessed: 07.06.2020.
[3] Dipl.-Ing. (FH) Stefan Luber. 2016. Was ist Machine Learning? https://www.bigdata-insider.de/was-ist-machine-learning-a-592092. Accessed: 16.07.2020.
[4] Dr. Klaus Manhart. 2018. Was Sie über Maschinelles Lernen wissen müssen. https://www.computerwoche.de/a/was-sie-ueber-maschinelles-lernen-wissen-muessen,3329560. Accessed: 16.07.2020.
[5] Mykola. [n.d.]. GTSRB - German Traffic Sign Recognition Benchmark. https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign. Accessed: 03.06.2020.
[6] Quang Nguyen. 2019. How to train YOLOv3 on Google COLAB to detect custom objects (e.g: Gun detection). https://medium.com/@quangnhatnguyenle/how-to-train-yolov3-on-google-colab-to-detect-custom-objects-e-g-gun-detection-d3a1ee43eda1. Accessed: 16.07.2020.
[7] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks. http://pjreddie.com/darknet/.

Abbildungsverzeichnis

Verkehrszeichenerkennung mit Keras

Maximilian Ritter
Hochschule für Angewandte Wissenschaften Hof
Angewandtes maschinelles lernen

ABSTRACT

In dieser Arbeit wird ein Einblick in die Verkehrszeichenerkennung geliefert. Diese wird mithilfe von Keras durchgeführt und verwendet ein Convolutional Neural Network. Die Arbeit beschäftigt sich hier explizit mit der Erkennung des Inhaltes eines Verkehrszeichens und nicht mit der Erkennung eines Schildes in der Umgebung. Der Datensatz für das Training des Modells stammt von der German Traffic Sign Benchmark Challenge aus dem Jahr 2011. Der für das Training verwendete Datensatz wird mithilfe eines Arguments beim Ausführen des Skripts übergeben.
Mithilfe des Python Skripts train.py wird ein Modell erzeugt und anschließend gesichert. Dieses Modell kann dem predict.py Skript übergeben werden und so auch eine Vorhersage zu eigenen Bildern beziehungsweise den Bildern eines Datensatzes treffen. Das jeweilige Ergebnis wird in ein Ausgabeverzeichnis geschrieben.

1 Vorwort

„It's simply that there are obviously two massive revolutions in the automobile industry. One is the transition to electrification and then the other is autonomy […] it came obvious to me that in the future any car that does not have autonomy would be as useful as a horse." – Elon Musk [1]
Dies ist nicht nur eine Einschätzung von Tesla CEO Elon Musk, sondern bereits für das Jahr 2030 wird insgesamt eine Beförderungsleistung von 2.450 Milliarden Personenkilometer in Europa durch autonome Privatfahrzeuge und autonome geteilte Fahrzeuge prognostiziert. [2]
Für das autonome Fahren sind allerdings einige Bestandteile notwendig. Dazu zählt die Erkennung von Objekten wie anderen Fahrzeugen, Fahrspuren und Schildern. Darauf aufbauend muss das Fahrzeug die entsprechenden Regeln kennen. Zum einen im Hinblick auf die Interpretation der Schilder, zum anderen auch Regelungen die nicht durch Schilder gekennzeichnet sind. Dazu zählen beispielsweise in Deutschland „rechts vor links" oder die maximal erlaubte Geschwindigkeit auf Landstraßen von 100 km/h. Aufbauend auf den Regeln, der Objekterkennung und weiteren Daten muss sich das Automobil im letzten Schritt korrekt verhalten. In dieser Arbeit soll einer dieser elementaren Bestandteile des autonomen Fahrens näher erläutert werden, nämlich die Verkehrszeichenerkennung. Verkehrszeichen beschreiben die notwendigen Vorschriften die ein möglichst unfallfreies und gefahrloses Fahren ermöglichen. Da mindestens mittelfristig sowohl autonom fahrende Autos als auch durch Menschen gefahrene Autos gleichzeitig genutzt werden, ist die Erkennung physischer Verkehrszeichen umso wichtiger. [2]
Bereits heute unterstützen eine Vielzahl von Fahrzeugen die Verkehrszeichenerkennung. Auf diese Weise können Fahrer kein Verkehrszeichen mehr übersehen. Außerdem ermöglicht dieses Feature bereits heute modernen Fahrzeugen ihre Geschwindigkeit basierend auf dem Verkehrszeichen automatisiert anzupassen. [3]

2 Hardware für die Verkehrszeichenerkennung

Fahrzeuge sind bereits mit einer großen Zahl an Sensoren und Kameras ausgestattet, um die Vielzahl an Assistenzsystemen anbieten zu können. [3] Für die Verkehrszeichenerkennung wird in der Regel die Frontkamera verwendet.

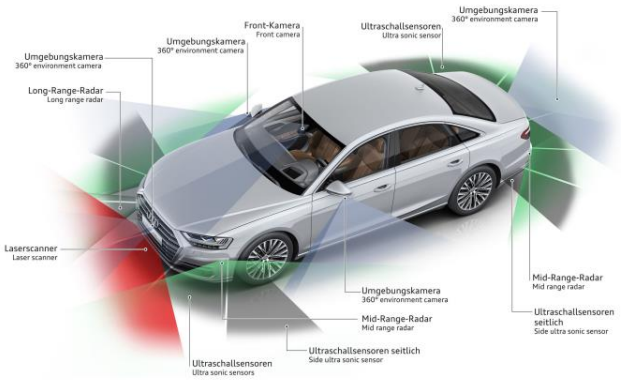


Abbildung 1 - Fahrerassistenzsysteme [3]

In der Abbildung 1 werden die verbauten Sensoren und Kameras eines Audi A8 aus dem Jahr 2017 dargestellt. Die für die Erkennung von Verkehrszeichen verwendete Kamera ist oberhalb der Windschutzscheibe positioniert. [3] Diese Position wird von einer Vielzahl anderer Hersteller ebenfalls verwendet.

3 Projektaufbau

Für die Erkennung eines Verkehrszeichens muss im ersten Schritt das Schild in einem Bild identifiziert werden und im zweiten Schritt der Inhalt des Schildes erkannt werden. Dieses Projekt setzt sich mit der Erkennung des Inhaltes auseinander und basiert auf dem Projekt zur Verkehrszeichenerkennung mit Keras von Adrian Rosebrock (Ph.D). [4]

3.1 Dataset

Der verwendete Datensatz stammt von einer Challenge zur Klassifizierung einzelner Bilder von der Joint Conference on Neural Networks 2011. Die Bilder sind in 42 Klassen unterteilt und reichen von Geschwindigkeitslimits bis hin zu einem Schild, das auf eine holprige Straße hinweist. Insgesamt 50.000 Bilder sind in diese 42 Klassen unterteilt. Dabei sind unterschiedliche Helligkeitsstufen und leicht unterschiedliche Winkel vorhanden. Ein negativer Punkt ist die ungleichmäßige Aufteilung der Bilder in die einzelnen Klassen.

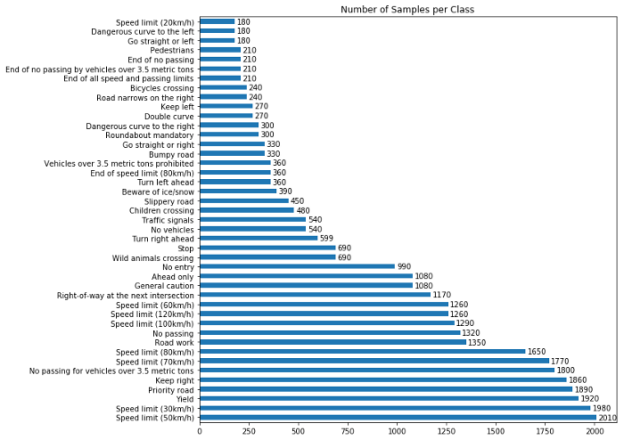


Abbildung 2 - Verteilung der Bilder [4]

Die Abbildung zeigt die Anzahl der Bilder pro Klasse und verdeutlicht so die ungleiche Verteilung. Mit 2.010 Bildern kommt die 50 km/h Geschwindigkeitsbegrenzung am häufigsten vor und die 20 km/h Begrenzung mit 180 Bildern am seltensten. Diese ungleiche Verteilung sollte beim späteren Lernen des Netzes berücksichtigt werden.

3.2 Packages

Die wichtigsten verwendeten Packages sind Tensorflow mit Keras, Sklearn, Skimages, OpenCV und Matplotlib.

Keras
Keras wird hauptsächlich für den Aufbau eines Sequential Models und für die Data Augmentation verwendet. Ein Keras Sequential Model ermöglicht den Aufbau verschiedener Ebenen mit einem Input und Output. Außerdem wird der ImageDataGenerator von Keras verwendet. Data Augmentation umfasst eine Reihe von Techniken, die den Umfang und die Qualität von Datensätzen so verbessern, dass mit ihnen bessere Modelle erstellt werden können. [5]

Skimage
Für die Bildbearbeitung, wie beispielsweise die Kontrastverbesserung, wird Skimage verwendet.

Sklearn
Sklearn ermöglicht das Aufbauen und Darstellen eines Berichts zu den wichtigsten Metriken bei der Klassifizierung. [6]

Matplotlib
Matplotlib wird verwendet, um relevante Metriken in einem Koordinatensystem mit der Anzahl der Epochs auf der x-Achse und der Accuracy auf der y-Achse zu visualisieren und anschließend in einem Bild zu speichern.

OpenCV
OpenCV wird in diesem Projekt dazu verwendet das Resultat bei der Vorhersage eines Verkehrszeichens auf das Bild geschrieben.

4 Convolutional Neural Network

Für die Verkehrszeichenerkennung wird in diesem Projekt ein CNN verwendet. CNN steht für Convolutional Neural Network und ist eine Architektur für das maschinelle Lernen, die für die Verarbeitung von Bildern entwickelt wurde. [7]

4.1 Allgemein

Convolutional Layer
Basis eines CNNs bilden die Convolutional Layers. Diese Ebene bekommt einen Input, verarbeitet diesen auf Basis verschiedener Filter und liefert einen Output. Diese Filter können Formen und Muster erkennen. Je tiefer die Ebene in dem Modell ist, desto komplexer werden auch die erkannten Muster.

Pooling Layer
Die Funktion von Pooling Layern besteht darin, die räumliche Größe der Darstellung zu verringern. Dafür wird eine pooling operation und eine Größe gewählt. In diesem Beispiel wird die Pooling Operation “Max Pooling” verwendet mit einer Größe von 2x2 Pixeln. Diese berechnet den maximalen Wert des jeweiligen Abschnitts eines Bildes. [8]

Fully Connected
Der letzte Schritt in einem CNN ist der Fully Connected Layer. Diese Ebene wird auch Dense Layer genannt und ist ein Klassifizierer für neuronale Netze. In dieser Schicht ist jeder Knoten mit jedem Knoten in der vorhergehenden Ebene verbunden. Da dieser Klassifizierer einen Feature Vector benötigt, wird ein Flattening Layer hinzugefügt, welcher die Map entsprechend umwandelt. [9]

Aktivierungsfunktionen und Optimierung
In diesem Projekt werden zwei Aktivierungsfunktionen genutzt, ReLU und softmax. Die Rectified Linear Activation Function (ReLU) ist eine einfache Berechnung, die den Wert des Inputs zurückgibt, wenn dieser größer 0 ist. Ansonsten liefert die Funktion 0 zurück. [9]

Softmax wird häufig, so auch in dieser Arbeit, in der letzten Ebene des neuronalen Netzes verwendet. Der Output ist hier ein Wert

zwischen 0 und 1 und zeigt so eine Wahrscheinlichkeitsverteilung. [10]

4.2 Verwendetes CNN

Das verwendete Modell stammt aus dem Beispielprojekt [4] von Adrian Rosebrock (Ph.D) und besteht aus einer Vielzahl verschiedener Ebenen.

Abbildung 3 - Visualisierung des CNNs

Die Abbildung zeigt die Reihenfolge und Art der verschiedenen Ebenen. Hervorzuheben ist, dass zu Beginn ein Convolutional Layer mit 8 Filtern verwendet wird und dann jeweils zwei Convolutional Layer mit je zweimal 16 und zweimal 32 Filter. Außerdem wird die Aktivierungsfunktion ReLU verwendet und eine Batch Normalization (BN) durchgeführt.

5 Training

Das Trainings-Skript lässt sich in sieben Abschnitte unterteilen, die auch in dem angefügten Code jeweils gekennzeichnet sind.

Abschnitt 1
Zu Beginn werden die Argumente für Ausführung des Python-Skripts definiert. Das erste ist hier der Pfad für das zu verwendende Dataset. Außerdem muss das Verzeichnis in dem das trainierte Modell gespeichert wird und das Verzeichnis, in welchem der Verlauf der wichtigsten Metriken gesichert wird übergeben werden.
Ein solcher Aufruf wird im Folgenden beispielhaft gezeigt:

```
python train.py
        --dataset gtsrb-german-traffic-sign
        --model output\\trafficsignnet.model
        --plot output\\plot.png
```

Abschnitt 2
Im zweiten Schritt werden die eingelesenen Bilder und das Label in zwei Arrays zwischengespeichert. Außerdem wird bereits direkt nach dem Einlesen der Contrast Limited Adaptive histogram equalization Algorithmus für eine Kontrastverbesserung angewendet. [11]

Abschnitt 3
Im dritten Schritt werden die für das Training wichtigen Metriken Epochs, Learning Rate und Batch Size festgelegt. Außerdem werden weitere Daten aus dem Datensatz eingelesen.

Epoch
Der Wert Epoch gibt die Anzahl der vollständigen Durchläufe durch den Trainingsdatensatz an. [12]

Batch Size
Die Batch Size gibt die Anzahl der Proben an, die durchgearbeitet werden müssen, bevor die internen Parameter des Modells aktualisiert werden. [12]

Learning Rate
Beim Deep Learning von neuronalen Netzen wird das stochastische Gradientenverfahren angewendet. Das stochastische Gradientenverfahren ist ein Optimierungsalgorithmus. Dieser schätzt den Fehlergradienten für den aktuellen Zustand des Modells anhand von Beispielen aus dem Trainingsdatensatz, um im nächsten Schritt die Gewichte des Modells zu aktualisieren.
Der Betrag, um den die Gewichte während des Trainings aktualisiert werden, wird als Learning Rate bezeichnet. [13]

Abschnitt 4
Im nächsten Schritt wird der ImageDataGenerator von Keras verwendet. Dieser nimmt einen Stapel an Bildern entgegen, transformiert diesen, ersetzt den ursprünglichen Stapel und gibt den neuen transformierten Stapel zurück. Dabei kann eine Vielzahl an Parameter gesetzt werden. [14]

Abschnitt 5
Nun kann bereits die build-Methode von Keras für das im Kapitel 4 gezeigte Modell angewendet werden. Danach wird das Modell mithilfe des keras.fit_generator() trainiert.

Abschnitt 6
Im Nachfolgenden wird mithilfe der predict-Methode das Modell evaluiert und das Gesamtergebnis ausgegeben. Im Gesamtergebnis werden die für die verschiedenen Schilderarten erzielten Werte angezeigt. Die Metriken lauten Precision, Recall, f1-score und support. Außerdem wird der Accuracy Wert ausgegeben.

Precision
Der Wert Precision gibt den Anteil der tatsächlich positiven Ergebnisse an, die als korrekt identifiziert wurden. [15] Die verwendete Formel ist:

$$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

[15]

Recall
Der Wert Recall gibt den Anteil der als positiv identifizierten an die tatsächlich korrekt sind:

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

[15]

F1 Score

Das F-Maß verwendet das Harmonische Mittel in Kombination mit den Werten Precision und Recall. Dies kann wie folgt berechnet werden:

$$F1 = \left(\frac{recall^{-1} + precisior^{-1}}{2}\right)^{-1}$$

[16]

Support

Der Support Wert zeigt die Anzahl des Auftretens in jeder Klasse von y_true an. [17]

Accuracy

Die Accuracy wird mittels einer Division der korrekten erkannten Bilder durch die Gesamtzahl der Versuche berechnet. Passend zu den Formeln aus Recall und Precision kann somit auch folgende Formel aufgestellt werden:

$$Accuracy = \frac{true\ positives\ (tp) + true\ negatives(tn)}{tp + tn + false\ negatives + false\ positives}$$

[17]

	precision	recall	f1-score	support
Speed limit (20km/h)	0.77	0.98	0.86	60
Speed limit (30km/h)	0.97	0.97	0.97	720
Speed limit (50km/h)	0.94	0.98	0.96	750
Speed limit (60km/h)	0.85	0.96	0.91	450
Speed limit (70km/h)	0.98	0.96	0.97	660
Speed limit (80km/h)	0.97	0.85	0.90	630
End of speed limit (80km/h)	0.98	0.80	0.88	150
Speed limit (100km/h)	0.97	0.96	0.96	450
Speed limit (120km/h)	0.94	0.93	0.93	450
No passing	0.98	0.98	0.98	480
No passing for vehicles over 3.5 metric tons	1.00	0.97	0.98	660
Right-of-way at the next intersection	0.92	0.91	0.92	420
Priority road	0.99	0.97	0.98	690
Yield	0.98	1.00	0.99	720
Stop	0.99	1.00	1.00	270
No vehicles	0.94	0.99	0.96	210
Vehicles over 3.5 metric tons prohibited	0.99	1.00	1.00	150
No entry	1.00	0.98	0.99	360
General caution	0.99	0.81	0.89	390
Dangerous curve to the left	0.55	0.72	0.62	60
Dangerous curve to the right	0.91	0.96	0.93	90
Double curve	0.87	0.66	0.75	90
Bumpy road	0.87	0.81	0.84	120
Slippery road	0.80	0.99	0.88	150
Road narrows on the right	0.80	0.96	0.87	90
Road work	0.89	0.98	0.94	480
Traffic signals	0.90	0.88	0.89	180
Pedestrians	0.52	0.78	0.62	60
Children crossing	0.92	0.97	0.94	150
Bicycles crossing	0.90	0.97	0.93	90
Beware of ice/snow	0.88	0.67	0.76	150
Wild animals crossing	0.96	0.97	0.97	270
End of all speed and passing limits	0.89	0.95	0.92	60
Turn right ahead	1.00	0.99	0.99	210
Turn left ahead	0.99	0.97	0.98	120
Ahead only	0.99	0.97	0.98	390
Go straight or right	0.98	1.00	0.99	120
Go straight or left	0.97	0.95	0.96	60
Keep right	0.98	1.00	0.99	690
Keep left	0.98	0.99	0.98	90
Roundabout mandatory	0.94	0.98	0.96	90
End of no passing	0.86	0.95	0.90	60
End of no passing by vehicles over 3.5 metric tons	0.82	0.91	0.86	90
accuracy			0.95	12630
macro avg	0.91	0.93	0.92	12630
weighted avg	0.95	0.95	0.95	12630

Abbildung 4 - Ergebnis von train.py

In der Abbildung 4 werden die konkreten Ergebnisse gezeigt, die mithilfe folgender Parameter erzielt werden konnten:
Epochs: 25
Learning Rate: 0.001
Batch Size: 94

Abschnitt 7

Im letzten Schritt wird das Modell noch in dem vorher übergebenen Verzeichnis gespeichert und ebenso das durch das Package Matplotlib erstellte Bild mit den wichtigsten Metriken.

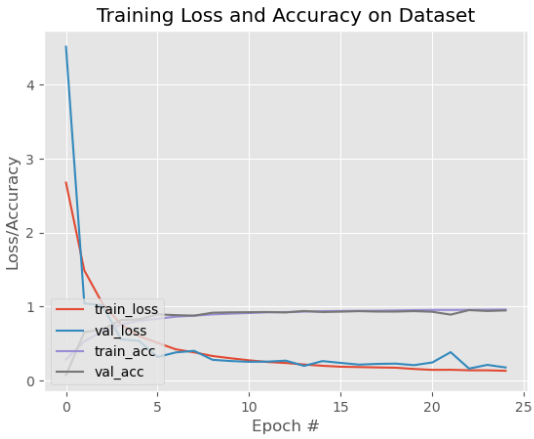


Abbildung 5 - Ergebnis durch Matplotlib

Der positive Verlauf der Accuracy und der Loss Werte wird in Abbildung 5 deutlich.

6 Prognose

Das Prognose-Skript predict.py lässt sich in vier Abschnitte unterteilen, die auch in dem angehängten Code jeweils gekennzeichnet sind.

Abschnitt 1

Für das Ausführen des Prediction Skripts werden die Argumente Modell, Bilderverzeichnis und Ausgabeverzeichnis definiert. Beim Modell wird der Pfad zu dem Modell übergeben, mithilfe dessen die Vorhersage getroffen werden soll. Der Pfad für das Bilderverzeichnis zeigt auf die zu testenden Bilder. In das Ausgabeverzeichnis werden nach der Vorhersage die Ergebnisse geschrieben.

Abschnitt 2

Die Nummerierung und der Text der Labels werden mittels der signnames.csv geladen und in einem Array zwischengespeichert.

Abschnitt 3

Für die Vorhersage werden 25 zufällige Pfade zu einzelnen Bildern in ein Array geladen. Die ersten drei Pfade werden dann von Pfaden zu eigenen Bildern überschrieben. Im Folgenden wird für jedes Bild ein Resize und eine Kontrastverbesserung durchgeführt und schlussendlich erneut die predict Methode aufgerufen.

Abschnitt 4
Im letzten Schritt wird nun das Ergebnis der Vorhersage in roter Schrift auf das Bild geschrieben und dieses in dem übergebenen Ausgabeverzeichnis gesichert.

Abbildung 6 - Beispielbild aus predict.py

Die Abbildung 6 zeigt beispielhaft ein solches Resultat.

7 Fazit

Das autonome Fahren umfasst viele unterschiedliche Aspekte, ein wichtiger ist hier das Erkennen der Verkehrszeichen und das Einhalten der damit verbundenen Vorschriften. Diese Arbeit soll einen Einblick in die Erkennung von Verkehrszeichen liefern. Allerdings gibt es zusätzlich zu der Erkennung von Verkehrszeichen mit einer Kamera weitere Ansätze. Die AUDI AG startete im Jahr 2016 eine Testphase des Traffic Light Information Systems in den USA. Dabei kommunizieren die Fahrzeuge mit der Ampel und bekommen so den momentanen Status und die Umschaltzeit mitgeteilt. Diese Umsetzung ermöglicht ein verhältnismäßig fehlerarmes System und kann durch die bekannten Umschaltzeiten dem Fahrer anzeigen mit welcher Geschwindigkeit er eine grüne Ampel erreicht. Dies senkt den Verbrauch und erhöht den Fahrkomfort und den Verkehrsfluss insgesamt. Außerdem können die anonymisierten Daten für bessere Schaltzeiten der Ampeln sorgen. [18] Ein interessanter Ansatz, der zeigt welche neuen Möglichkeiten die technologische Entwicklung eröffnet und welche Vorteile dadurch erzielt werden kann. Ebenfalls spannend ist die vehicle-to-everything (V2X) Technologie, die durch eine zellulare Verbindung zur Cloud, der Kommunikation mit anderen Fahrzeugen und der Infrastruktur mehrere Vorteile mit sich bringt. So wird erwartet, dass diese Technologie die Zukunft auf Autobahnen stark beeinflussen wird. Die Organisation Zenzik soll in Großbritannien das autonome Fahren vorantreiben. Diese hat bereits einen Bericht veröffentlicht, der auf eine Abkehr von der traditionellen Beschilderung abzielt. In-Car-Signaling in Verbindung mit Live-Informationen von der Infrastruktur die an die Fahrzeuge weitergegeben wird, soll so einen enormen Sicherheitsvorteil sowie eine Reduzierung der Wartungskosten für die Infrastruktur schaffen. [19]
Es gibt also verschiedene Herangehensweisen, um die Herausforderungen auf dem Weg in eine autonome sicherere Zukunft zu lösen.

REFERENCES

[1] Lex Fridman. 2019. Elon Musk: Tesla Autopilot | Artificial Intelligence (AI) Podcast. Letzter Zugriff: 02.08.2020. Unter: https://youtube.com/watch?v=dEv99vxKjVI

[2] Statista. (2018). Prognose der Beförderungsleistung von Personenkraftwagen in Europa nach Art der Fahrzeugnutzung im Zeitraum der Jahre 2018 bis 2030 (in Milliarden Personenkilometern). Statista. Letzter Zugriff: 04.08.2020. Unter: https://de.statista.com/statistik/daten/studie/875204/umfrage/prognostizierte-befoerderungsleistung-von-pkw-in-europa-nach-nutzungsart/

[3] AUDI AG. 2017. Fahrerassistenzsysteme. Letzter Zugriff: 05.08.2020. Unter: https://www.audi-mediacenter.com/de/technik-lexikon-7180/fahrerassistenzsysteme-7184

[4] Adrian Rosebrock. 2019. Traffic Sign Classification with Keras and Deep Learning. Letzter Zugriff: 04.08.2020. Unter: https://www.pyimagesearch.com/2019/11/04/traffic-sign-classification-with-keras-and-deep-learning/

[5] Shorten, C., Khoshgoftaar. 2019.T.M. A survey on Image Data Augmentation for Deep Learning. J Big Data. doi.org/10.1186/s40537-019-0197-0

[6] scikit-learn developers. 2020. sklearn.metrics.classification_report. Letzter Zugriff: 04.08.2020. Unter: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html

[7] Roland Becker. 2019. CONVOLUTIONAL NEURAL NETWORKS – AUFBAU, FUNKTION UND ANWENDUNGSGEBIETE. Letzter Zugriff: 04.08.2020. Unter: https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/

[8] Jason Brownlee. 2019. A Gentle Introduction to Pooling Layers for Convolutional Neural Networks. Letzter Zugriff: 04.08.2020. Unter: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks//

[9] Julia Fischer, Kevin Pochwyt. 2017. Neuronale Netze. Letzter Zugriff: 04.08.2020. Unter: user.phil.hhu.de/~petersen/SoSe17_Teamprojekt/AR/neuronalenetze.html

[10] Keras. 2020. Layer activation functions. Letzter Zugriff: 03.08.2020. Unter: https://keras.io/api/layers/activations/

[11] The MathWorks, Inc.. 2020. adapthisteq. Letzter Zugriff: 04.08.2020. Unter: https://www.mathworks.com/help/images/ref/adapthisteq.html

[12] Jason Brownlee. 2019. Difference Between a Batch and an Epoch in a Neural Network. Letzter Zugriff: 05.08.2020. Unter: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks//

[13] Jason Brownlee. 2019. Understand the Impact of Learning Rate on Neural Network Performance. Letzter Zugriff: 01.08.2020. Unter: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

[14] Adrian Rosebrock. 2019. Keras ImageDataGenerator and Data Augmentation. Letzter Zugriff: 02.08.2020. Unter: https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/

[15] Google Developers. 2020. Classification: Precision and Recall. Letzter Zugriff: 04.08.2020. Unter: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

[16] scikit-learn developers. 2020. sklearn.metrics.precision_recall_fscore_support. Letzter Zugriff: 31.07.2020. Unter: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html

[17] Google Developers. 2020. Classification: Accuracy. Letzter Zugriff: 04.08.2020. Unter: https://developers.google.com/machine-learning/crash-course/classification/accuracy

[18] Audi Mediacenter. 2019. Audi vernetzt sich mit Ampeln in Deutschland. Letzter Zugriff: 04.08.2020. Unter: https://www.audi-mediacenter.com/de/pressemitteilungen/audi-vernetzt-sich-mit-ampeln-in-deutschland-11649

[19] TU-Automotive. 2020. Planning a Future Without Road Signs. Letzter Zugriff: 04.08.2020. Unter: https://www.tu-auto.com/planning-a-future-without-road-signs/

```
# Verwendung python train.py --dataset gtsrb-german-traffic-sign --model
output\\trafficsignnet.model --plot output\\plot.png

import matplotlib
matplotlib.use("Agg")

from model_class.trafficsignnet import TrafficSignNet
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import classification_report
from skimage import transform
from skimage import exposure
from skimage import io
import matplotlib.pyplot as plt
import numpy as np
import argparse
import random
import os

# Abschnitt 2
def load_split(basePath, csvPath):
      data = []
      labels = []

      rows = open(csvPath).read().strip().split("\n")[1:]
      random.shuffle(rows)

      for (i, row) in enumerate(rows):
            if i > 0 and i % 1000 == 0:
                  print("[INFO] processed {} total images".format(i))

            (label, imagePath) = row.strip().split(",")[-2:]

            imagePath = os.path.sep.join([basePath, imagePath])
            image = io.imread(imagePath)

            image = transform.resize(image, (32, 32))
            image = exposure.equalize_adapthist(image, clip_limit=0.1)

            data.append(image)
            labels.append(int(label))

      data = np.array(data)
      labels = np.array(labels)

      return (data, labels)
```

```
# Abschnitt 1
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
      help="path to input GTSRB")
ap.add_argument("-m", "--model", required=True,
      help="path to output model")
ap.add_argument("-p", "--plot", type=str, default="plot.png",
      help="path to training history plot")
args = vars(ap.parse_args())

# Abschnitt 3
NUM_EPOCHS = 25
INIT_LR = 1e-3
BS = 94

labelNames = open("signnames.csv").read().strip().split("\n")[1:]
labelNames = [l.split(",")[1] for l in labelNames]

trainPath = os.path.sep.join([args["dataset"], "Train.csv"])
testPath = os.path.sep.join([args["dataset"], "Test.csv"])

# Aufruf Abschnitt 2
print("[INFO] loading training and testing data...")
(trainX, trainY) = load_split(args["dataset"], trainPath)
(testX, testY) = load_split(args["dataset"], testPath)
# _____________________

trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0

numLabels = len(np.unique(trainY))
trainY = to_categorical(trainY, numLabels)
testY = to_categorical(testY, numLabels)

classTotals = trainY.sum(axis=0)
classWeight = classTotals.max() / classTotals

# Abschnitt 4
aug = ImageDataGenerator(
      rotation_range=10,
      zoom_range=0.15,
      width_shift_range=0.1,
      height_shift_range=0.1,
      shear_range=0.15,
      horizontal_flip=False,
      vertical_flip=False,
      fill_mode="nearest")

print("[INFO] compiling model...")
opt = Adam(lr=INIT_LR, decay=INIT_LR / (NUM_EPOCHS * 0.5))

# Abschnitt 5
model = TrafficSignNet.build(width=32, height=32, depth=3,
      classes=numLabels)
model.compile(loss="categorical_crossentropy", optimizer=opt,
      metrics=["accuracy"])
```

```
# Abschnitt 6
print("[INFO] training network...")
H = model.fit_generator(
      aug.flow(trainX, trainY, batch_size=BS),
      validation_data=(testX, testY),
      steps_per_epoch=trainX.shape[0] // BS,
      epochs=NUM_EPOCHS,
      class_weight=classWeight,
      verbose=1)

print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=BS)
print(classification_report(testY.argmax(axis=1),
      predictions.argmax(axis=1), target_names=labelNames))

print("[INFO] serializing network to '{}'...".format(args["model"]))
model.save(args["model"])

# Abschnitt 7
N = np.arange(0, NUM_EPOCHS)
plt.style.use("ggplot")
plt.figure()
plt.plot(N, H.history["loss"], label="train_loss")
plt.plot(N, H.history["val_loss"], label="val_loss")
plt.plot(N, H.history["accuracy"], label="train_acc")
plt.plot(N, H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy on Dataset")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig(args["plot"])
```

```
# Verwendung python predict.py --model output\\trafficsignnet.model --
images gtsrb-german-traffic-sign\\Test --examples examples

from tensorflow.keras.models import load_model
from skimage import transform
from skimage import exposure
from skimage import io
from imutils import paths
import numpy as np
import argparse
import imutils
import random
import cv2
import os
# Abschnitt 1
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
      help="path to pre-trained traffic sign recognizer")
ap.add_argument("-i", "--images", required=True,
      help="path to testing directory containing images")
ap.add_argument("-e", "--examples", required=True,
      help="path to output examples directory")
args = vars(ap.parse_args())

print("[INFO] loading model...")
model = load_model(args["model"])

# Abschnitt 2
labelNames = open("signnames.csv").read().strip().split("\n")[1:]
labelNames = [l.split(",")[1] for l in labelNames]

# Abschnitt 3
print("[INFO] predicting...")
imagePaths = list(paths.list_images(args["images"]))
random.shuffle(imagePaths)
imagePaths = imagePaths[:25]
imagePaths[0] = "gtsrb-german-traffic-sign\\rechts_c_3.png"
imagePaths[1] = "gtsrb-german-traffic-sign\\v_achten_c_3.png"
imagePaths[2] = "gtsrb-german-traffic-sign\\stop_c_3.png"

for (i, imagePath) in enumerate(imagePaths):
      image = io.imread(imagePath)
      image = transform.resize(image, (32, 32))
      image = exposure.equalize_adapthist(image, clip_limit=0.1)

      image = image.astype("float32") / 255.0
      image = np.expand_dims(image, axis=0)

      # Abschnitt 4
      preds = model.predict(image)
      j = preds.argmax(axis=1)[0]
      label = labelNames[j]

      image = cv2.imread(imagePath)
      image = imutils.resize(image, width=128)
      cv2.putText(image, label, (5, 15), cv2.FONT_HERSHEY_SIMPLEX,
            0.45, (0, 0, 255), 2)

      p = os.path.sep.join([args["examples"], "{}.png".format(i)])
```

Obstacle Avoiding Robot

Rudolf Gevis
Fakultät Informatik
Hochschule Hof
Hof, Bayern, Deutschland
rudolf.gevis@hof-university.de

Zusammenfassung

In diesem Artikel wird beschrieben wie mit Transfer Learning ein Modell erstellt wird. Dieses Modell ist in der Lage Anhand eines Fotos Hindernisse zu erkennen. Das trainierte Modell wird in einem mobilen Roboter betrieben um ihn autonom fahren zu lassen. Zu Beginn werden die verwendeten Soft- und Hardwarekomponenten beschrieben. Anschließend werden Grundlagen zu neuronalen Netzen, CNN´s und die Architektur des für das Transfer Learning verwendeten Grundmodells erläutert. Es folgt die Beschreibung der Umsetzung des Transfer Learnings. Dabei wird gezeigt wie der Trainingsdatensatz erstellt, das Modell trainiert und wie es für die Verwendung auf einem Raspberry Pi konvertiert wurde. Abschließend gibt es eine kurze Erläuterung zur Funktionsweise der Webanwendung des Roboters und eine Darstellung der gesammelten Erkenntnisse.

1 Einführung

1.1 Motivation

Viele Automobil-Hersteller und branchenfremde Tech-Firmen entwickeln aktuell autonom fahrende Roboterautos. Darüber hinaus werden auch in der Produktion und Logistik bereits autonome mobile Roboter eingesetzt. Sogar im Haushalt gibt es einen Einsatzzweck für solche Systeme. So übernehmen Roboter eigenständig menschliche Tätigkeiten, wie z.B. das Staubsagen oder Rasen mähen. Um sich autonom fortbewegen zu können und mit der echten Welt ohne Schaden zu interagieren ist es notwendig das ein solcher Roboter seine Umgebung wahrnehmen kann. Für den Nahbereich wird häufig ein Ultraschallsensor zur Abstandsmessung verwendet. Damit kann man feststellen ob sich etwas in der Nähe des Sensors befindet. Der Sensor sendet eine Ultraschallwelle aus, wenn diese auf ein Hindernis trifft, wird sie reflektiert und kehrt zurück zum Sensor. Dabei wird die Zeit zwischen dem aussenden und empfangen gemessen. Da die Schallgeschwindigkeit konstant ist lässt sich daraus die Entfernung zum reflektieren Objekt berechnen[3]. Lidar-Sensoren hingegen senden ein Laserlicht aus. Dieses wird ebenfalls reflektiert. Damit kann man den Raum abtasten und Hindernisse erkennen[6]. Computer Vision, zu Deutsch "maschinelles Sehen", ist ein weiterer Ansatz für die Umsetzung von autonomer Navigation für mobile Roboter. Dabei werden die von Kameras aufgenommen Bilder verarbeitet und analysiert, um deren Inhalte und geometrischen Formen zu erkennen[4]. Dies ist eine spannende Möglichkeit, da hier neuronale Netze die Verarbeitung und Analyse übernehmen können.
Dieser Artikel beschreibt die Vorgehensweise ein neuronales Netz mithilfe von überwachtem Lernen zu entwickeln, welches eigenständig Hindernisse erkennen kann. Hierbei gilt es einem mathematischen Lernmodell eine Menge von Eingaben und Ausgaben zur Verfügung zu stellen. Daraus leitet sich das Modell eine Funktion ab, um einen Ausgabewert einem Eingabewert zuzuordnen. Mit überwachtem Lernen lassen sich insbesondere die beiden Probleme der Regression und Klassifikation lösen. Bei der Klassifikation, die hier zum Einsatz kommt, wird einem Eingabebild eine vorher bekannte Klasse zugeordnet[11]. Üblicherweise werden beim Lernen eine sehr große Menge von Werten benötigt. Diese Menge lässt sich durch das Nutzen von Transfer Learning deutlich reduzieren. Dazu wird ein bereits antrainiertes neuronales Netz als Grundlage für ein erneutes Training verwendet. Das neu-trainierte Modell wird in diesem Projekt anschließend auf einem mobilen Roboter eingesetzt, der im autonomen Modus in der Lage ist, sich selbstständig in einer definierten Umgebung zu bewegen und Hindernissen auszuweichen. Nachfolgend werden die dabei verwendeten Technologien beschrieben und erläutert.

1.2 Projektumfeld

Diese Studienarbeit wurde im Rahmen der Veranstaltung „Angewandtes maschinelles Lernen“ im Sommersemester 2020 an der Hochschule Hof angefertigt. Herr Prof. Dr. Sebastian Leuoth vermittelte die Grundlagen zum maschinellen Lernen. Dabei wurden vornehmlich die drei Bereiche Überwachtes Lernen, Unüberwachtes Lernen und Bestärkendes Lernen theoretisch erörtert und anhand praktischer Beispiele dargestellt. Außerdem begleitete er dieses Projekt als Ansprechpartner für Fragestellungen rund um die Umsetzung. Das Ziel dieser Veranstaltung ist es die erlangten theoretischen Grundlagen zu den verschiedenen Bereichen auf ein praktisches Projekt anzuwenden. Hierzu wurden verschiedene Tools, Programmiersprachen und Entwicklungsumgebungen genutzt. Das Transfer Learning wurde in Google Colab [1] mit der Programmiersprache Python in der Version 3.7 umgesetzt. Google Colab ist eine cloudbasierte Anwendung, die es ermöglicht Python Code auszuführen und Ressourcen

[1] colab.research.google.com

wie GPU-Rechenleistung von Google zu verwenden. Hierbei wurden Tensorflow in der Version 2.3[2], Tensorflow Lite[3] und Keras[4] als Machine Learning Frameworks eingesetzt. Tensorflow bildet dabei das Basis-Framework. Keras dient als high-level API für Tensorflow und vereinfacht den Zugriff auf die Funktionalität. Tensorflow Lite ist für die Konvertierung und Verwendung des Modells auf dem Raspberry Pi zuständig. Außerdem wurden zwei Webanwendungen mit Flask[5] umgesetzt. Flask ist ein Webframework mit dem es möglich ist einen Webserver mit Python zu realisieren. Für die Aufnahme und Verarbeitung von Bildern wurde das opensource Framework OpenCV[6] verwendet. Des Weiteren wurde ein Compiler und eine Classification Engine von Corals´s EdgeTPU-API verwendet[7].

1.3 Vorstellung der Hardware

Die Basis für dieses Projekt bildet ein Roboter-Bausatz namens „Devastator Tank Mobile Robot Platform“ der chinesischen Firma DFRobot. Dieses Set beinhaltet das Roboter Chassis, eine Stromversorgung mit sechs einsetzbaren AAA-Batterien und zwei Motoren. Die Motoren lassen sich in zwei Richtungen drehen und sind dadurch in der Lage den Roboter zu bewegen. Dabei gibt es insgesamt fünf möglich Zustände: Nach vorne fahren, nach hinten fahren, nach links drehen, nach rechts drehen und stehen bleiben. Ein Raspberry Pi 3B+ übernimmt dabei die Steuerung der Zustände. Um die Motoren per Raspberry Pi steuern zu können wird zusätzlich noch ein Motortreiber mit der Bezeichnung „L298N“ eingesetzt. Die Motoren und der Motortreiber werden durch die Batterien mit Strom versorgt. Eine Powerbank versorgt den Raspberry Pi mit Energie. Um Hindernisse zu erkennen ist außerdem ein Kameramodul mit dem Raspberry Pi verbunden. Damit der Roboter ausweichen kann muss die Ausführungszeit des Modells möglichst gering sein. Deswegen kommt hier Googles Coral USB Accelerator zum Einsatz. Dies ist eine Edge-TPU (Tensor Processing Unit), welche einen Coprozessor für einen Computer bereitstellt, d.h. er beschleunigt die Ausführung von maschinellen Lernmodellen auf lokalen Geräten. Abbildung 1 zeigt den fertig zusammengebauten Roboter.

Abbildung 1. Fertig zusammengebauter Roboter

2 Theoretische Grundlagen

2.1 Neuronale Netze

Bei maschinellem Lernen geht es darum eine Lösung für ein gegebenes Problem zu finden. Hierzu werden Daten mit Algorithmen kombiniert um ein Modell zu erstellen, welches in der Lage ist ein bestimmtes Problem zu lösen. In diesem Artikel soll ein Klassifizierungsproblem gelöst werden.
Ein Modell entspricht einem künstlichem neuronalen Netz welches versucht das menschliche Gehirn nachzubilden. Solch ein neuronales Netz besteht meist aus einem Input-Layer, mehreren Hidden-Layern und einem Output-Layer. Ein Layer, zu Deutsch Schicht, ist eine Menge von Neuronen, die durch gewichtete Kanten mit Knoten von anderen Schichten verbunden werden. Das Neuron empfängt also einen Eingabewert und multipliziert ihn mit dem Gewicht der Kante. Dies wird für jede Eingangsverbindung wiederholt und daraus eine Summe erstellt. Außerdem wird dazu eine Unschärfe addiert, auch Bias genannt. Dieser Wert wird in eine Aktivierungsfunktion gegeben. Diese Funktion entscheidet ob das Neuron aktiviert wird oder nicht. Am Ende werden die Werte an den Output-Layer übergeben. Dieser weist mithilfe einer Softmax-Funktion jeder Klasse eine Wahrscheinlichkeit zu. Das Modell transformiert also Eingabewerte zu Ausgabewerten. Dies passiert indem es anhand der Hyperparameter, also den Gewichten und dem Bias, eine Vorhersage abgibt. Diese Hyperparameter werden zu Beginn des Trainings zufällig initialisiert. Beim Lernen werden mehrere Eingaben als Stapel (Batch) in das Modell gegeben. Das neuronale Netz gibt zu jedem Element des Stapels eine Vorhersage ab und vergleicht dies mit der tatsächlichen Antwort. Daraus wird dann ein Klassifizierungsfehler berechnet, indem kalkuliert wird wie oft das Modell eine falsche Vorhersage getroffen hat. Daraus lässt sich die Genauigkeit des Netzes bestimmen. Das Modell lernt durch Backpropagation. Hierbei werden die Hyperparameter der Netzes abhängig von ihrem Einfluss

[2] https://www.tensorflow.org
[3] https://www.tensorflow.org/lite
[4] https://keras.io
[5] https://flask.palletsprojects.com/en/1.1.x/
[6] https://opencv.org
[7] https://coral.ai/docs/

auf den Klassifizierungsfehler geändert. Durch das stetige wiederholen des Ratens kann das Modell seine Vorhersagen den tatsächlichen Vorhersagen annähern.

2.2 Convolutionale neuronale Netze

Bei der Klassifizierung von Bildern geht es darum Muster zu erkennen. Convolutional Neural Networks (CNN), zu Deutsch faltbare neuronale Netze, eignen sich sehr gut für die Mustererkennung und werden deshalb häufig verwendet um Bilder zu Klassifizieren. Bei den vorher erläuterten neuronalen Netzen werden Bilder üblicherweise durch einen Flatten-Layer als Eingangsschicht aufgenommen. Dadurch wird ein Bild in einen eindimensionalen linearen Vektor überführt. Bei CNN´s hingegen behält das Bild seine Dimensionen bis es in einen Fully-Connected-Layer gegeben wird. Dadurch bleiben räumliche Zusammenhänge erhalten. So bestehen die Hidden-Layer üblicherweise aus einer Abfolge von einem oder mehreren Convolutional-Layern. Solch ein Layer transformiert ebenfalls einen Input zu einem Output. Da ein Bild als dreidimensionale Matrix eingegeben wird, kann darüber nun Schrittweise eine kleinere Faltungs-Matrix, auch Filter-Kernel genannt, bewegt werden. Diese Filter bewirken eine Transformation vom ursprünglichen Bild, z.B. in Form einer Drehung, Aufhellung oder Kantendetektion. Daraus lassen sich Objektinformationen (Features) extrahieren. Somit sind die Filter für die Mustererkennung zuständig. Muster können zu Beginn einfache Texturen, Formen oder Kanten darstellen. Später werden komplexe Objekte erkannt, wie z.B. ein Auge. Die Anzahl der Filter muss beim Training angegeben werden. Die Werte dieser Filter werden zu Beginn jedoch zufällig initialisiert. Bei jeder Iteration werden diese vom Modell angepasst um besser Muster zu erkennen.
Nach einer Convolution-Schicht folgt meist eine Pooling-Schicht. Ihre Aufgabe ist es die Auflösung der transformierten Bilder zu reduzieren und dadurch eine weitere Abstraktion der Features zu erreichen. Eine weitere häufig verwendete Schicht ist der Dropout-Layer. Dieser entfernt zufällig eine Menge inaktiver Neuronen aus einem Netz. Dies wird verwendet um Overfitting, zu Deutsch Überanpassung, entgegenzuwirken. Bei Overfitting geht die Fähigkeit des Modells zur Abstraktion verloren. D.h. das Modell erkennt die Trainingsbilder immer besser, während andere Bilder, die nicht Teil des Trainings sind, immer schlechter erkannt werden [5]. In Abbildung 2 wird ein häufiger Aufbau eines CNN´s dargestellt.

2.3 MobileNetV2

In diesem Artikel wird Transfer Learning verwendet. Dieses Vorgehen eignet sich besonders wenn bereits erlangtes Wissen für die Lösung eines Problems auf ein neues Problem angewendet werden soll. Zu diesem Zweck wird ein bereits angelerntes neuronales Netz als Grundlage für eine neue Netzarchitektur verwendet. Dieses Projekt verwendet MobileNetV2 von Google Inc. als Basisnetz. Das Modell wurde

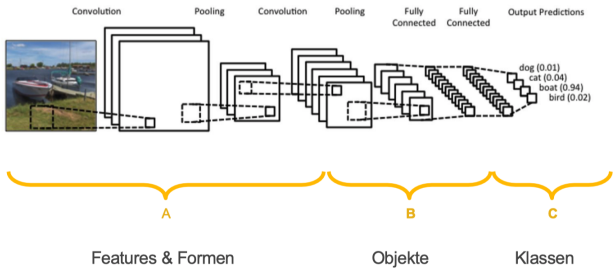


Abbildung 2. CNN - Visualisierung aus [12]

mit dem ImageNet-Datensatz [8] auf die Klassifizierung von 1000 Klassen trainiert. Der Vorteil an diesem CNN ist das es sehr effizient auf mobilen Geräten verwendet werden kann und trotzdem gute Leistungen bei der Vorhersage zeigt.
In Abbildung 3 wird die Architektur dieses Modells dargestellt. "Jede Zeile beschreibt eine Sequenz von 1 oder mehreren identischen [...] Schichten, die n-mal wiederholt werden. Alle Schichten in der gleichen Abfolge haben die gleiche Anzahl c von Ausgangskanälen. [...] Alle räumlichen Faltungen verwenden 3×3 Filter-Kernel. Der Expansionsfaktor t wird immer auf die Eingangsgröße angewandt [...]."[12]
Ein Bottleneck-Block besteht dabei aus mehreren Schichten

Input	Operator	t	c	n
$224^2 \times 3$	conv2d	-	32	1
$112^2 \times 32$	bottleneck	1	16	1
$112^2 \times 16$	bottleneck	6	24	2
$56^2 \times 24$	bottleneck	6	32	3
$28^2 \times 32$	bottleneck	6	64	4
$14^2 \times 64$	bottleneck	6	96	3
$14^2 \times 96$	bottleneck	6	160	3
$7^2 \times 160$	bottleneck	6	320	1
$7^2 \times 320$	conv2d 1x1	-	1280	1
$7^2 \times 1280$	avgpool 7x7	-	-	1
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-

Abbildung 3. MobileNetV2 - Architektur aus [12]

wie in Abbildung 4 ersichtlich. Das Ziel eines solchen Blocks ist es die Anzahl der Daten die durch das Netzwerk fliesen zu reduzieren.
Die Expansion-Schicht erweitert die Anzahl der Kanäle. Dadurch ist es möglich die Dimensionen im Eingang zu verkleinern, während die Filterung der Daten mit mehr Dimensionen durchgeführt werden kann. Die Projection-Schicht hingegen reduziert die Anzahl der Dimensionen wieder. Dieser Sachverhalt ist in Abbildung 5 dargestellt [7].
Die Depthwise-Convultion-Schicht wurden bereits im Vor-

[8] http://www.image-net.org

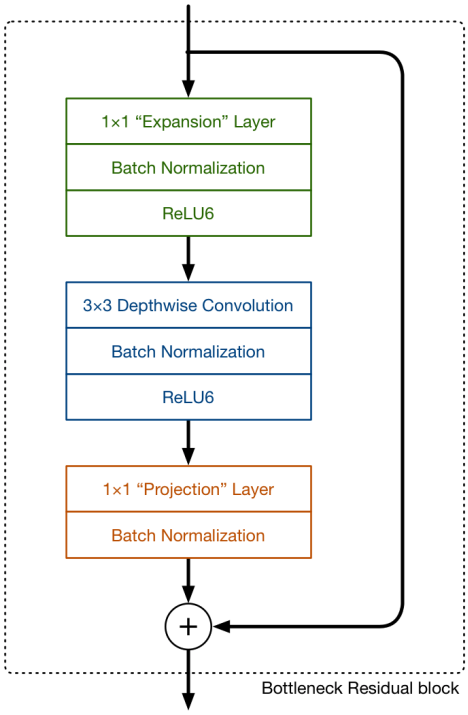


Abbildung 4. MobileNetV2 - Bottleneck [7]

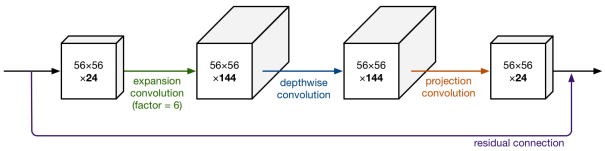


Abbildung 5. MobileNetV2 - Visualisierung Expansionsfaktor [7]

gängermodell MobileNetV1 eingeführt. Im Gegensatz zu den Convolution-Schichten aus subsection 2.2 wird bei der Depthwise-Convolution-Schicht jeder Filterkanal nur an einem Eingangskanal verwendet. D.h. das ein 3-Kanal-Filter und ein 3-Kanal-Bild auf die verschiedene Kanälen geteilt wird. Das entsprechende Bild wird anschließend mit dem entsprechenden Kanal gefiltert und dann zurück gestapelt. Abbildung 6 stellt dieses Vorgehen grafisch dar[9].

Zusätzlich gibt es pro Teilbereich eine Batch-Normalization-Schicht. Mit der Normalisierung wird die jeweilige Datenverteilung besser durch die Aktivierungsfunktion „ReLU6“ verwendet. D.h. Ausreißer in den Daten haben einen deutlich geringeren Effekt. Dadurch können die Neuronen effizienter verwendet werden. Außerdem entfällt dadurch die Verwendung eines Bias, wodurch sich die Anzahl der Hyperparameter weiter reduzieren lässt.

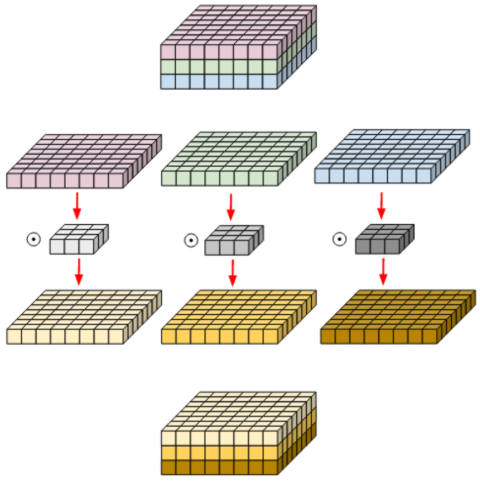


Abbildung 6. Depthwise Convolutional Layer - Visualisierung aus[9]

3 Transfer Learning

3.1 Bilder aufnehmen

Um ein maschinelles Lernmodell zu trainieren wird ein Datensatz benötigt. Dieser setzt sich aus Bildern und Klassen, auch Labels genannt, zusammen. Für dieses Projekt wurden insgesamt 274 Bilder mit einer Auflösung von 1280 x 960 Pixeln mit dem Kameramodul des Roboters aufgenommen. Um das Aufnehmen zu erleichtern wurde eine Webanwendung umgesetzt. Der Zweck dieser Webanwendung ist es Fotos in je zwei Ordner abzuspeichern. Ein Ordner namens „blocked“ nimmt Fotos auf, die für den Roboter ein Hindernis darstellen. Der zweite Ordner namens „free“ verbirgt Bilder die freie Fahrt für den Roboter bedeuten. Diese beiden Ordner entsprechen den Klassen. Abbildung 7 zeigt die Webanwendung. Abbildung 8 sowie Abbildung 10 zeigen Bilder aus dem Ordner „free“ und Abbildung 9 sowie Abbildung 11 zeigen Bilder aus dem Ordner „blocked“.

3.2 Training

Nachfolgend werden ausgewählte Schritte aus dem Trainingsprozess beschrieben. Die Details zur Umsetzung und der Quellcode ist dem beigefügten Jupyter Notebook zu entnehmen. Die Grundlage hierfür bildet ein Jupyter Notebook, welches von Google´s Coral als Tutorial angeboten wird [8]. Bevor das Training beginnt müssen zunächst die Bilder in einen Trainingsdatensatz und einen Validierungsdatensatz geteilt werden. In diesem Projekt werden 20% der Bilder für die Validierung zufällig der Gesamtmenge entnommen. Mit dem Trainingsdatensatz findet das eigentliche Training statt. Das Modell sieht die Bilder und lernt von diesen Daten. Der Validierungsdatensatz hingegen wird nur indirekt beim Training verwendet. Der Validierungsdatensatz wird für die

Abbildung 7. Webseite für das Aufnehmen und Labeln von Trainings-Bildern

Abbildung 8. Aufnahme (free) - Abstand zur Wand

Feinabstimmung der Parameter des Modells verwendet. Daraus lernt das Modell seine Fähigkeiten zu verallgemeinern und auch auf andere Bilder zu übertragen. [2]
Anschließend werden die Bilder normalisiert, da viele Lernverfahren empfindlich auf eine große Distanz der Werte reagieren. D.h. die Bilddaten werden auf einen Bereich zwischen 0 und 1 gebracht. Die originalen Bilddaten setzen sich je Pixel aus RGB-Werten zwischen 0 und 255 zusammen.
Daraufhin werden für jeden Datensatz zwei Generatoren mit Keras erstellt. Der Zweck dieser Generatoren ist es bei jeder Iteration im Training einen Stapel (Batch) von Bildern zur Verfügung zu stellen. In diesem Fall wurde für die Stapel eine maximale Größe von 64 Bildern bestimmt. Zusätzlich werden die Bilder durch die Generatoren auf die vom Modell benötigte Größe von 224 x 224 Pixeln genormt.
Anschließend wird ein neues Modell erstellt, welches auf dem Basisnetz MobileNetV2 basiert. Um dieses Basisnetz für das Training verwenden zu können muss dies erst mit Keras importiert werden. Da es in diesem Artikel um Transfer

Abbildung 9. Aufnahme (blocked) - Abstand zur Wand

Abbildung 10. Aufnahme (free) - Abstand zum Hindernis

Learning geht, wird das Modell ohne die Klassifizierungs-Schicht geladen. Außerdem werden die Gewichte des Modells eingefroren. Dadurch werden die Gewichte des Basisnetzes nicht mittrainiert und man kann es als Feature-Extraktor verwenden. D.h. das Modell behält das Wissen, welches es aus seinem vorherigen Training gelernt hat.
Auf dem Grundmodell aufbauend wird ein neues sequentielles neuronales Netz erstellt. Dieses beinhaltet zusätzlich eine zwei dimensionale Convolution-Schicht, welche für Faltungen der Bilder zuständig ist. Eine Dropout-Schicht, die mit einer 20% Wahrscheinlichkeit inaktive Neuronen entfernt. Eine Pooling-Schicht welche die Bildgröße reduziert und zuletzt den Dense-Layer, auch Fully-Connected-Layer genannt, der für das Zuweisen von Klassen verantwortlich ist. In Abbildung 12 ist die neue Architektur dargestellt.
Das Modell wurde mit zehn Epochen trainiert. D.h. das eingeben der Stapel und die Klassifizierung jedes enthaltenen

Abbildung 11. Aufnahme (blocked) - Abstand zum Hindernis

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 7, 7, 1280)        2257984
_________________________________________________________________
conv2d (Conv2D)              (None, 5, 5, 32)          368672
_________________________________________________________________
dropout (Dropout)            (None, 5, 5, 32)          0
_________________________________________________________________
global_average_pooling2d (Gl (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 2)                 66
=================================================================
Total params: 2,626,722
Trainable params: 368,738
Non-trainable params: 2,257,984
_________________________________________________________________
```

Abbildung 12. Neues Modell - Architektur

Elements wurde zehn mal wiederholt. In Abbildung 13 sieht man den Trainingsverlauf. Dabei ist zu erkennen dass die Genauigkeit bereits nach drei Epochen auf über 90% ist. Auch beim Loss sind sehr schnell gute Ergebnisse zu beobachten. Der Loss gibt an wie schlecht die Vorhersage des Modells im Durchschnitt war. Die schnellen Erfolge basieren auf dem sehr guten Grundmodell MobileNetV2. Bereits nach zehn Epochen konnte das Training beendet werden, da es darüber hinaus zu Overfitting kommt.

3.3 Konvertierung

Bevor das Modell auf dem Raspberry Pi verwendet wird, muss es in das TFLite-Format konvertiert werden. Die Parameter des Modells sind standardweise im 32-Bit Floating Point Format erstellt. Da Coral´s USB Accelerator ausschließlich 8-Bit Integer Modelle unterstützt, muss dass Modell vollständig quantifiziert werden. Dies reduziert außerdem die Größe des Netzes und verkürzt die Latenz bei der Vorhersage [1]. Abschließend wurde das konvertierte Modell noch mit Coral´s Edge TPU Compiler kompiliert.

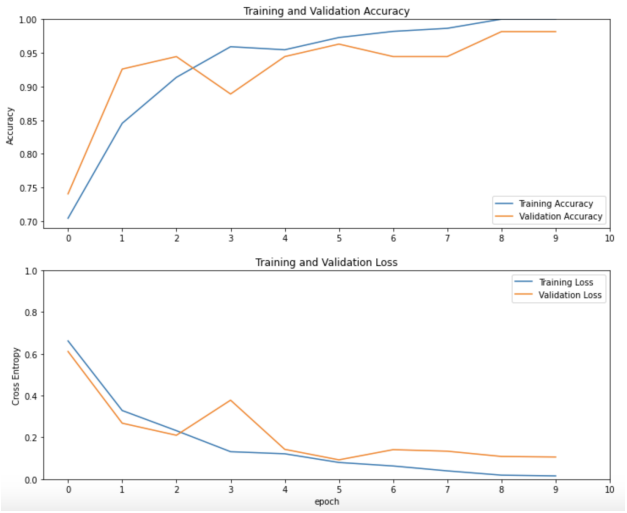


Abbildung 13. Visualisierung des Trainingsverlaufes

4 Praktischer Einsatz

4.1 Funktionsweise der Webanwendung

Das fertige Modell wird auf dem Roboter betrieben. Hierfür ist eine Webanwendung mit Flask erstellt worden. Die Basis für diese Webanwendung ist ein Projekt des GitHub Users "pengdev"[10]. Er hat einen Webserver entwickelt der dazu in der Lage ist die Videoaufnahme eines Raspberry Pi per Webstream auf einer Webseite darzustellen. Außerdem verwendet er Coral's ClassificationEngine[9] um Bilder aus diesem Stream zu klassifizieren. Die Ergebnisse der Klassifizierung werden ebenfalls via Webstream dargestellt. So ist in der oberen linken Ecke zu sehen welche Klasse, mit welcher Wahrscheinlich durch das Modell zugewiesen wurde. Außerdem sieht man die Zeit in Millisekunden, die das Modell gebraucht hat um ein Frame zu klassifizieren. In Abbildung 14 und Abbildung 15 sieht man beispielhafte Aufnahmen des Webstreams.

Zusätzliche Funktionalität wurde in diesem Projekt ergänzt. So kann man den Roboter ebenfalls mithilfe der Webseite steuern. Dazu hält man die Taste „w“ gedrückt um nach vorne zu fahren, „a“ um den Roboter nach links zu drehen, „s“ um nach hinten zu fahren und „d“ um den Roboter nach rechts zu drehen. Sobald man eine von diesen Tasten loslässt bleibt der Roboter stehen. Diese Steuerung wurde mit Javascript umgesetzt. Dabei werden GET-Requests an den Webserver gesendet, die eine Methode aufrufen. So sendet die Taste “w“ einen Request mit der URL */up* an den Webserver. Dieser ruft dann die Methode *up()* auf und der Roboter schaltet in den Zustand das er noch vorne fährt. Zusätzlich zur Basissteuerung ist ein autonomer Modus implementiert. In diesem fährt der Roboter solange gerade aus, wie die Klasse „free“ klassifziert wird. Wenn „blocked“ erkannt wird dreht sich

[9] https://coral.ai/docs/reference/edgetpu.classification.engine/

Abbildung 14. Webanwendung - Demonstration (free)

Abbildung 15. Webanwendung - Demonstration (blocked)

der Roboter zufällig nach links oder rechts, bis wieder „free“ erkannt wird.

4.2 Ergebnisse

Die Erkennung der beiden Klassen funktioniert in der Umgebung (Flur) in der auch die Trainingsbilder entstanden sind sehr gut. Auch in anderen Zimmern funktioniert die Erkennung noch gut. Jedoch sind diese der Trainingsumgebung sehr ähnlich. Die Zeit bis zur Vorhersage ist mit durchschnittlich 11 Millisekunden in Ordnung. Im autonomen Modus verzögert sich die Zeit bis zur Vorhersage auf durchschnittlich 16 Millisekunden. Dies ist gerade noch ausreichend um den Roboter autonom Entscheidungen Treffen zu lassen, ohne das er gegen ein Hindernis stößt. Es kam während der Tests jedoch auch vor das ein Hindernis zu spät erkannt wurde und der Roboter dieses berührte. Abschließend kann man sagen das Klassifizierung eine geeignete Alternative zu Ultraschall-Sensoren und Lidar-Sensoren sein kann. Dazu müssen jedoch noch weitere Optimierungen vorgenommen werden um die Zeit zur Vorhersage zu reduzieren. Außerdem kann man die Genauigkeit des Modells durch weitere Trainingsbilder aus unterschiedlichen Umgebungen weiter verbessern.

5 Anhang

Literatur

[1] [n.d.]. *TensorFlow models on the Edge TPU.* Retrieved 27.07.2020 from https://coral.ai/docs/edgetpu/models-intro#compatibility-overview

[2] 2017. *About Train, Validation and Test Sets in Machine Learning.* Retrieved 27.07.2020 from https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

[3] 2018. *Wiki: Ultraschallsensoren.* Retrieved 25.07.2020 from https://www.exp-tech.de/blog/wiki-ultraschallsensoren

[4] 2020. *Computer Vision.* Retrieved 25.07.2020 from https://de.wikipedia.org/wiki/Computer_Vision

[5] 2020. *Convolutional Neural Network.* Retrieved 26.07.2020 from https://de.wikipedia.org/wiki/Convolutional_Neural_Network

[6] 2020. *Lidar.* Retrieved 25.07.2020 from https://de.wikipedia.org/wiki/Lidar

[7] Matthijs Hollemans. 2018. *MobileNet version 2.* Retrieved 27.07.2020 from https://machinethink.net/blog/mobilenet-v2/

[8] Google LLC. 2019. *Retrain a classification model for Edge TPU using post-training quantization (with TF2).* Retrieved 28.07.2020 from https://colab.research.google.com/github/google-coral/tutorials/blob/master/retrain_classification_ptq_tf2.ipynb#scrollTo=hRTa3Ee15WsJ

[9] Atul Pandey. 2018. *Depth-wise Convolution and Depth-wise Separable Convolution.* Retrieved 28.07.2020 from https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec

[10] GitHup User pengdev. 2019. *Coral object classification model live stream.* Retrieved 28.07.2020 from https://github.com/pengdev/coral_live_stream_demo

[11] Prof. Dr. rer. nat. Jörg Frochte. 2019. *Maschinelles Lernen - Grundlagen und Algorithmen in Python, 2. Auflage.* Carl Hanser Verlag München. S. 21.

[12] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. *CoRR* abs/1801.04381 (2018). arXiv:1801.04381 http://arxiv.org/abs/1801.04381

Einführung in Convolutional Autencoder für Imagedenosing und Superesolution

Tobias Heinrich
Fakultat Informatik
Hochschule Hof
Hof, Bayern, Deutschland
tobias.heinrich@hof-university.de

Zusammenfassung — Die Vorstellung verpixelte Bilder zu rekonstruieren und scharf darzustellen ist seit langem das Ziel von vielen Wissenschaftlern der Informatik. In dieser Studienarbeit wird diesem Ziel nachgegangen anhand eines Beispiels von einem Autoencodern. Diese Studienarbeit soll einen Überblick gewähren, welche Aufgaben und Funktionalitäten ein Autoencoder hat. Speziell wird auf den Convolutional Autoencoder eingegangen, der sich am Prinzip der CNN „Convolutional Neural Networks“ bedient. Mithilfe des Datensatz LFW „Labeled Faces in the Wild Home“ und dem Framework Tensorflow wird ein CNN Autoencoder konstruiert. Das Ziel dieser Arbeit ist es verschieden Topologien eines Convolutional Autoencoders zu erstellen und zu vergleichen, um herauszufinden welche Architektur die besten Ergebnisse liefert.

I. Einleitung

In der heutigen Zeit sind die Begriffe Maschinelles Lernen, neuronale Netze, Künstliche Intelligenz und Deep Learning unabdingbar. Doch die meisten Menschen wissen nicht oder haben sogar Angst vor dieser neuen Technologie. Angst davor, dass Künstliche Intelligenz ihre Jobs wegnehmen kann. Dabei wissen die meisten nicht, dass Sie des Öfteren mit Künstlicher Intelligenz in Berührung kommen. Spamfilter von E-Mail-Anbieter beruhen auf K.I oder Chatbots von diversen Firmen, die zur Produktberatung dienen. Des Weiteren ist vielen nicht bewusst, dass ihr neuer 8K Fernseher in Wirklichkeit keine 8K Bilder zeigt, sondern SD, HD oder 4K die Mithilfe eines neuronalen Netzes hochskaliert und als 8K dargestellt werden. Nun ist die Frage zu klären, was maschinelles Lernen und künstliche neuronale Netze bedeuten.

II. Maschnielles Lernen

Mit maschinelles Lernen können wir Computern beibringen, effizient Aufgaben zu lösen, ohne sie wirklich speziell für diese Aufgabe programmiert zu haben. Das heißt der Computer generiert analog zum Menschen, eigenes Wissen aus Erfahrungen, um damit selbständig Lösungen für unbekannte Probleme zu finden. Die Vorgehensweise ist folgende: ein Programm analysiert Beispiele und versucht mit erlernten Algorithmen in den Daten gewisse Muster und Gleichheiten zu erkennen. Das Ziel von maschinelles Lernen ist es Daten effizient miteinander zu verknüpfen, Zusammenhänge zu erkennen, Rückschlüsse zu ziehen und vor allem Vorhersagen zu treffen.

III. Künstliche Neuronale Netzwerke

Ein künstliches neuronales Netz (KNN), oder auch Netzwerk, sind Netzwerke aus künstlichen Neuronen. Diese künstlichen Netzwerke aus Neuronen haben als biologisches Vorbild, die Vernetzung von Neuronen im Nervensystem eines Lebewesens. Hierbei handelt es sich aber eher um die Lösung von konkreten Problemen, weniger um die Nachbildung von biologischen Netzwerken.

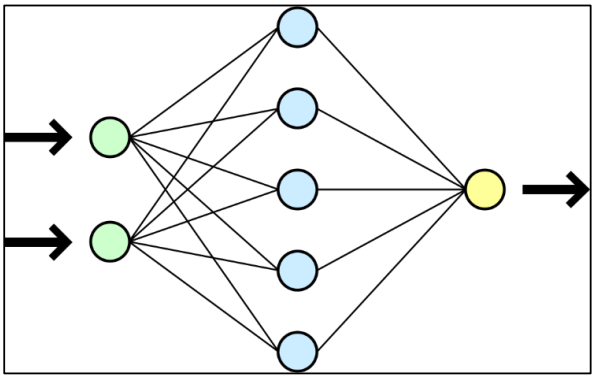


Abbildung 1: Darstellung eins einfachen künstlichen neuronalen Netzes [1]

Abbildung 1 soll hier eine vereinfachte Darstellung von einem künstlichen neuronalen Netzwerk darstellen. Die Topologie eines Netzes hängt von seiner jeweiligen Aufgabe ab. Ist das Netz einmal aufgestellt folgt nun die Trainingsphase indem das Netz lernt. (vgl. [1])

Ein KNN kann auf vielen unterschiedlichen Arten lernen. Neben der Entwicklung von neuem Verbinden können Netzwerken auch von der Löschung existierender Verbindungen profitieren. Eine der wohl wichtigsten Methoden ist die Änderung der Gewichtung. Wobei die Gewichtung der einzelnen Neuronen des Netzwerkes angepasst werden. Abbildung 2 zeigt hier einen typischen Aufbau eines künstlichen Neurons und deren Gewichtung.

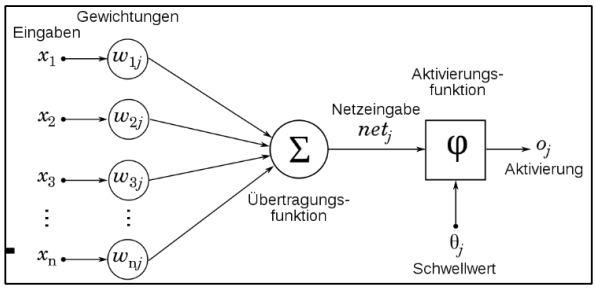


Abbildung 2: Darstellung eines künstlichen Neurons mit seinen Elementen [2]

Weiter Methoden wären unter anderem das Hinzufügen oder Löschen von Neuronen oder das Modifizieren von Aktivierung-, Propagierung- oder Ausgabefunktion. (vgl. [1]) Um ein künstliches neuronales Netz so zu modifizieren, dass es für bestimmte Muster passt, gibt es verschiedene Lernverfahren. Diese werden grob in drei Kategorien unterteilt.

A. Überwachtes Lernen

Überwachtes Lernen (engl. *supervised learning)* wird vor allem bei der Klassifikation oder Regression genutzt. Bei dieser Methode lernt der Algorithmus anhand eines beschrifteten Datensatz. Aufgrund der Beschriftung kann der Algorithmus die Ergebnisse auf die Korrektheit überprüfen und gegebenenfalls eine Anpassung der Parameter durchführen. Mit einem sehr großen Datensatz kann der Algorithmus nun Gleichheiten erkennen und Werte oder Objekte klassifizieren. (vgl. [3])

B. Unüberwachtes Lernen

Eine zweite Möglichkeit stellt das Unüberwachte Lernen (engl. unsupervised learning). Im Vergleich zum Überwachten Lernen, lernt dieser Algorithmus mit einem unbeschrifteten Datensatz. Im Gegensatz zum Algorithmus des überwachten Lernens versucht dieser, neue Muster und Beziehungen zu erkennen. Wichtig ist dabei, dass dem Algorithmus kein festes Ziel vorgegeben wird. Das Neuronale Netz passt sich entsprechend der Eingaben von selbst an. Bekannte Anwendungsgebiete sind zum Beispiel Clustering Verfahren und sogenannte Autoencoder auf denen später genauer eingegangen wird. (vgl. [3])

C. Bestärkende Lernen

Das Bestärkende Lernen (engl. reinforcement learning), unterscheidet sich gravierend von den anderen aufgezeigten Lernmethoden. Nicht wie die anderen zwei Methoden hat die Methode des bestärkenden Lernens von Beginn keine Trainingsdaten zur Verfügung. Bei dieser Lernmethode steht ein Agent im Mittelpunkt, der durch das Kommunizieren mit sein Umwelt Daten generiert. Deswegen wird das bestärkende Lernen auch als Lernen durch Interaktion bezeichnet.

Die Methodik ist am besten für Situationen geeignet, wo es bisher kaum generierte Trainingsdaten gibt und der Agent selbst Lösungen finden muss, wie etwa beim Durchspielen von einem Spiel oder das Durchqueren neuere Bereiche. Die Aktionen des Agenten werden mit Belohnungen oder Bestrafungen gewertet. Das Ziel des Agenten ist es die Belohnungen zu maximieren und Bestrafungen zu minimieren, um so eine bestmögliche Entscheidung zu treffen, die zu einer eine optimale Strategie führt. Anders als beim überwachten Lernen hat der Agent keine Richtig oder Falsch Werte, an denen er sich orientieren kann, das heißt seine Aktionen beruhen immer auf Ergebnissen aus der Vergangenheit. Dieser Vorgang wird öfters wiederholt bis die beste Strategie für das Problem ermittelt wurde. (vgl. [4])

IV. AUTOENCODER (AE)

Ein Autoencoder ist ein unüberwachtes neuronales Netzwerk das als Ziel hat, eine komprimierte Repräsentation eines Datensatz darzustellen. Autoencoder werden vor allem zur Dimensionsreduktion genutzt. Ein Autoencoder ist prinzipiell in drei Schichten aufgebaut. Einer Eingabeschicht bei der zum Beispiel, wie in dieser Studienarbeit, Pixel eins Gesichtes in den Neuronen gespeichert werden. Einem sogenannten Bottleneck, welches die essenziellen Bilddaten gespeichert hat, die wichtig für die Rekonstruktion der Neuronen sind. Und als letztes eine Ausgabeschicht, in der jedes Neuron die gleiche Eigenschaft hat, wie die entsprechenden Neurone der Eingabeschicht.

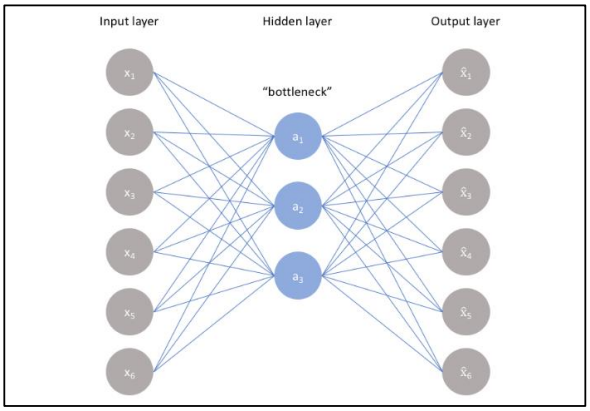


Abbildung 3: Möglicher Typologie eines Autoencoders [5]

Diese drei Schichten werden aus der Abbildung 3 nochmal verdeutlicht. Der Autoencoder lernt indem er die Ausgabedaten x mit den Eingabedaten $\hat{x}$ vergleicht:

$$L = x - \hat{x}$$

Auch genannt Rekonstruktionsfehler. Dieser Fehler berechnet die Unterschiede zwischen den Eingabedaten und den rekonstruierten Ausgabedaten. Das Bottleneck spielt hier eine besondere Rolle. Das Netzwerk könnte sonst einfach durch das Erinnern an die Eingabewerte lernen. Um das zu verhindern, gibt das Bottleneck die Menge an Informationen vor, die das Netzwerk durchlaufen kann und erzwingt dadurch eine erlernte Komprimierung der Eingabedaten. (vgl. [5])

Um optimale Ergebnisse für das Modell eines Autoencoders zu erzielen, sollten folgende Aspekte berücksichtigt werden. Zum einen sollte der Autoencoder empfindlich genug für Eingaben sein sodass eine genaue Rekonstruktion erstellt werden kann. Zum anderen unempfindlich genug für Eingaben, dass das Modell die Daten nicht einfach wiedererkennt oder überanpasst (eng. overfitting).

Dieser Kompromiss des Modells zwingt es nur die essenziellen Variationen von Daten beizubehalten, die für die Rückbildung der Eingabedaten wichtig sind. Oft wird deswegen eine Verlustfunktion (eng. Loss function) gewählt, die das Modell empfindlich genug zu den Eingabedaten hält. Um dem overfitting entgegenzuwirken wird zusätzlich zur Verlustfunktion ein Regulierer (eng. regularizer) bestimmt. (vgl. [5])

Arten von Autoencodern

Es gibt viele verschiedene Arten von Autoencoder. Wichtig für diese Studienarbeit sind vor allem der Denosing Autoencoder und der Covolutional Autoencoder.

A. Denosing Autoencoder (DAE)

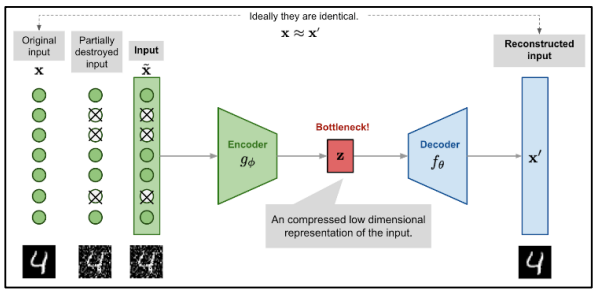


Abbildung 4: Topologie eines Denoising Autoencoder [6]

Das Design von Abbildung 4 ist durch die Tatsache motiviert, dass Menschen ein Objekt oder eine Szene leicht erkennen können, selbst wenn die Ansicht teilweise verdeckt oder verfälscht ist. Um die teilweise zerstörte Eingabe zu „reparieren“, muss der Denoising Autoencoder die Beziehung zwischen den Eingabedimensionen erkennen und erfassen, um auf fehlende Teile schließen zu können.

Bei hochdimensionalen Eingaben mit hoher Redundanz wie bei Bildern hängt das Modell von Beweisen ab, die aus einer Kombination vieler Eingabedimensionen stammen, um die denoisierte Version wiederherzustellen, anstatt eine Dimension zu overfitting. Dies bildet eine gute Grundlage für das Erlernen einer robusten latenten Repräsentation.
Bei einem Denosing Autoencoder wird eine zweite Version der Eingabedaten erstellt und diese leicht manipuliert. Wie Abbildung 4 zeigt wird den Eingabedaten rauschen (eng. noise) hinzugefügt. Die verfälschten Daten werden dann in das Netzwerk gegeben und encodiert. Aus den encodierten Daten im Bottleneck werden dann die Ausgabedaten rekonstruiert. Im letzten Schritt werden die Ausgabedaten mit den jeweiligen Eingabedaten verglichen, um so den Verlust zum Original messen zu können. Ziel ist es eine Rekonstruktion zu schaffen die dem originalem Input gleich ist. (vgl. [6])

B. Convolutional Autoencoder (CAE)

Vollständig verbundene Autoencoders und Denoising Autoencoder ignorieren beide vollständig die 2D-Bildstruktur. Das ist nicht nur ein Problem beim Umgang mit realistisch dimensionierten Eingaben, führt aber auch Redundanz in den Parametern ein, wodurch jedes Merkmal gezwungen wird, global zu sein. Der Trend in Bezug auf Vision und Objekterkennung wurde jedoch übernommen. Bei den erfolgreichsten Modellen geht es darum, lokalisierte Merkmale zu entdecken, die sich über die gesamte Eingabe wiederholen. CAEs unterscheiden sich von herkömmlichen AEs, da ihre Gewichte auf alle Stellen in der Eingabe aufgeteilt werden, wobei die räumliche Lokalität erhalten bleibt. Die Rekonstruktion beruht daher auf einer linearen Kombination grundlegender Bildfelder basierend auf dem latenten Code. (vgl. [7])

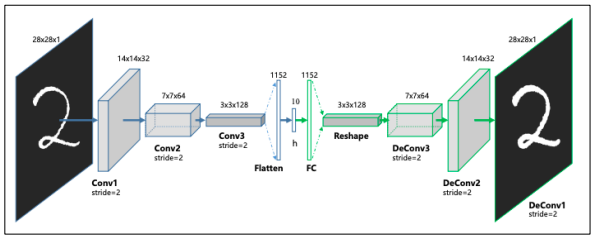


Abbildung 5: Topologie eines CAE [8]

Abbildung 5 zeigt eine typische Topologie eines CAE. Hier wird ein flaches 2D-Bild in ein dickes Quadrat (Conv1) extrahiert und dann weiterhin zu einer langen Kubik (Conv2) und letztendlich weiteren zu längeren Kubik (Conv3). Dieser Prozess soll die räumlichen Beziehungen in den Daten beibehalten. Dies ist der Kodierungsprozess in einem Convolutional Autoencoder. In der Mitte befindet sich ein vollständig verbundener Autoencoder, dessen verborgene Schicht nur aus 10 Neuronen besteht. Danach folgt der Dekodierungsprozess, der die Kubik abflacht, und dann zu einem flachen 2D-Bild rekonstruiert.

Einschub Convolutional Layer

Convolutional Layer falten die Eingabe und transferieren ihr Ergebnis an die nächste Schicht weiter. Dies gleicht der Reaktion eines Neurons im visuellen Kortex auf einen bestimmten Reiz. Jedes Faltungsneuron verarbeitet Daten nur für sein Empfangsfeld. Obwohl vollständig verbundene neuronale Netzwerke zum Erlernen von Funktionen sowie zum Klassifizieren von Daten verwendet werden können, ist es nicht praktisch, diese Topologie auf Bilder anzuwenden. Eine immense Anzahl von Neuronen wäre notwendig, selbst in einer flachen Architektur, aufgrund der sehr großen Eingabegrößen, die mit Bildern verbunden sind, wobei jeder Pixel eine relevante Variable darstellt. Beispielsweise hat ein Layer für ein „relative kleines Bild“ mit der Größe von

100 × 100 Pixel insgesamt 10.000 Gewichte für jedes Neuron in der zweiten Schicht. Die Convolution bietet eine Alternative für dieses Problem, da sie die Anzahl der freien Parameter verringert und es dem Netz ermöglicht, mit weniger Parametern tiefer zu gehen. Unabhängig von der Bildgröße erfordern ein Kachelbereiche mit einer Größe von 5 x 5 mit jeweils denselben gemeinsamen Gewichten beispielsweise nur 25 lernbare Parameter. Durch die Verwendung regulierter Gewichte über weniger Parameter werden die Probleme mit dem verschwindenden Gradienten und dem explodierenden Gradienten, die während der Rückausbreitung (eng. backpropagation) in traditionellen neuronalen Netzen auftreten, vermieden. (vgl. [9]).

V. Labeled Faces in the Wild Home Dataset

In dieser Studienarbeit wird der Labeled Faces in the Wilde Home Dataset (LFW), das von der University of Massachusetts Amherst zu Verfügung gestellt wird, zum Trainieren der verschiedenen Netzwerke genutzt. Der Datensatz enthält mehr als 13.000 Bilder von Gesichtern, die aus dem Internet gesammelt wurden. Jedes Gesicht wurde mit dem Namen der abgebildeten Person beschriftet. 1680 der abgebildeten Personen haben zwei oder mehr unterschiedliche Fotos im Datensatz. (vgl. [10]). Diese Bilder haben eine natürliche Auflösung von 256 x 256 Pixel. Aus Performancegründen wurden diese Bilder auf 80 x 80 verkleinert und an die Netztopologie angepasst.

VI. Implementation des Projektes

Im Folgenden werden verschiedene Topologien eines CAE implementiert und getestet.

A. Import

```
[ ]  import cv2
     import datetime
     import numpy as np
     import matplotlib.pyplot as plt
     from sklearn.model_selection import train_test_split

     import tensorflow as tf
     from tensorflow.keras import Model, Input, regularizers
     from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, UpSampling2D
     from tensorflow.keras.callbacks import EarlyStopping
     from keras.preprocessing import image

     import glob
     from tqdm import tqdm
     import warnings;
     warnings.filterwarnings('ignore')
```

Abbildung 6: Implementiere Bibliotheken

In der Abbildung 6 werden alle nötigen Bibliotheken importiert. Die meisten Imports beziehen sich hierbei auf Tensorflow und Keras. Zur Darstellung von Bildern und Graphen wird hier die Matplotlib verwendet. Numpy bietet ein leistungsstarkes mehrdimensionales Array-Objekt und Tools für die Arbeit mit diesen Arrays. Die OpenCv Bibliothek (cv2) wird benötigt, um Bildern formatieren zu können.

B. Herunterladen und verarbeiten Daten

Damit die heruntergeladenen Bilder von den Gesichtern aus dem TAR-Archiv verwendet werden können, müssen diese zunächst extrahiert und in die Anwendung geladen werden. Hierbei wird zunächst das TAR-Archiv geöffnet und anschließend entpackt. Abschließend wird das TAR-Archiv wieder geschlossen. Die Daten der Gesichter werden in einer globalen Variablen gespeichert. Um mit den Bilddaten arbeiten zu können, werden sie in ein Array gespeichert. Damit die Rechenleistung minimiert wird, werden die Originalbilder von 256 x 256 Pixel auf 80 x 80 reduziert und normalisiert. Darauffolgend werden die Daten im nächsten Schritt gruppiert, zu einem Trainingsdatensatz und einem Testdatensatz.

C. Daten manipulieren

```
[ ]  # function to reduce image resolution while keeping the image size constant

     def pixalate_image(image, scale_percent = 50):
       width = int(image.shape[1] * scale_percent / 100)
       height = int(image.shape[0] * scale_percent / 100)
       dim = (width, height)

       small_image = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)

       # scale back to original size
       width = int(small_image.shape[1] * 100 / scale_percent)
       height = int(small_image.shape[0] * 100 / scale_percent)
       dim = (width, height)

       low_res_image = cv2.resize(small_image, dim, interpolation = cv2.INTER_AREA)

       return low_res_image
```

Abbildung 7: pixalate_image Funktion

Im nächsten Schritt werden die Daten manipuliert. In der Abbildung 7 wird die *pixalate_image* Funktion implementiert. Diese Funktion ist dafür zuständig das die scharfen Inputbilder verpixelt werden.

D. Daten für den Input vorbereiten

```
[ ]  # get low resolution images for the training set
     train_x_px = []
     train_X = np.array(train_x)

     for i in range(train_X.shape[0]):
       temp = pixalate_image(train_X[i,:,:,:])
       train_x_px.append(temp)

     train_x_px = np.array(train_x_px)


     # get low resolution images for the validation set
     val_x_px = []
     val_X = np.array(val_x)
     print(val_X.shape)

     for i in range(val_X.shape[0]):
       temp = pixalate_image(val_X[i,:,:,:])
       val_x_px.append(temp)

     val_x_px = np.array(val_x_px)
```

Abbildung 8: Trainings- und Testdaten Anpassung

In diesem Codeblock wird der Datensatz verpixelt und in einem neuen Array abgespeichert.

E. Entwürfe der Netzwerktopologien

1) Entwurf 1

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 80, 80, 3)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 80, 80, 256)       7168
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 80, 80, 128)       295040
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 40, 40, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 40, 40, 64)        73792
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 40, 40, 64)        36928
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 80, 80, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 80, 80, 128)       73856
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 80, 80, 256)       295168
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 80, 80, 3)         6915
=================================================================
Total params: 788,867
Trainable params: 788,867
Non-trainable params: 0
_________________________________________________________________
```

Abbildung 9: Topologie des Entwurfes 1

Die Topologie für Entwurf 1 wurde sehr einfach gehalten. Sie besteht aus drei Convolutional Layer und einem Maxpooling Layer in der Encoderstruktur. Symmetrisch dazu drei Convolutional Layer und ein Upsacling Layer in der Decoderstruktur.

2) Entwurf 2

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 80, 80, 3)]       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 80, 80, 256)       7168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 40, 40, 256)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 40, 40, 128)       295040
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 20, 20, 64)        73792
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 10, 10, 32)        18464
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 10, 10, 32)        9248
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 20, 20, 32)        0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 20, 20, 64)        18496
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 40, 40, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 40, 40, 128)       73856
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 40, 40, 256)       295168
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 80, 80, 256)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 80, 80, 3)         6915
=================================================================
Total params: 798,147
Trainable params: 798,147
Non-trainable params: 0
_________________________________________________________________
```

Abbildung 10: Topologie des Entwurfes 2

Anders als beim ersten Modell wird hier (siehe Abbildung 10) auf eine tiefe Topologie gesetzt. Das heißt, statt nur einem *MaxPooling* Layer wird in diesem Modell drei Mal runterskaliert und es gibt insgesamt neun Covolutional Layer. Dieser Schritt wurde gewählt, um mehr Features aus den Bildern zu gewinnen. Diese Topologie ist wie Entwurf 1 symmetrisch aufgebaut.

3) Entwurf 3

```
Model: "model_2"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_3 (InputLayer)         [(None, 80, 80, 3)]       0
_________________________________________________________________________________________________
conv2d_16 (Conv2D)           (None, 80, 80, 64)        1792        input_3[0][0]
_________________________________________________________________________________________________
conv2d_17 (Conv2D)           (None, 80, 80, 64)        36928       conv2d_16[0][0]
_________________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D) (None, 40, 40, 64)      0           conv2d_17[0][0]
_________________________________________________________________________________________________
conv2d_18 (Conv2D)           (None, 40, 40, 128)       73856       max_pooling2d_4[0][0]
_________________________________________________________________________________________________
conv2d_19 (Conv2D)           (None, 40, 40, 128)       147584      conv2d_18[0][0]
_________________________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D) (None, 20, 20, 128)     0           conv2d_19[0][0]
_________________________________________________________________________________________________
conv2d_20 (Conv2D)           (None, 20, 20, 256)       295168      max_pooling2d_5[0][0]
_________________________________________________________________________________________________
up_sampling2d_4 (UpSampling2D) (None, 40, 40, 256)     0           conv2d_20[0][0]
_________________________________________________________________________________________________
conv2d_21 (Conv2D)           (None, 40, 40, 128)       295040      up_sampling2d_4[0][0]
_________________________________________________________________________________________________
conv2d_22 (Conv2D)           (None, 40, 40, 128)       147584      conv2d_21[0][0]
_________________________________________________________________________________________________
add (Add)                    (None, 40, 40, 128)       0           conv2d_22[0][0]
                                                                   conv2d_19[0][0]
_________________________________________________________________________________________________
up_sampling2d_5 (UpSampling2D) (None, 80, 80, 128)     0           add[0][0]
_________________________________________________________________________________________________
conv2d_23 (Conv2D)           (None, 80, 80, 64)        73792       up_sampling2d_5[0][0]
_________________________________________________________________________________________________
conv2d_24 (Conv2D)           (None, 80, 80, 64)        36928       conv2d_23[0][0]
_________________________________________________________________________________________________
add_1 (Add)                  (None, 80, 80, 64)        0           conv2d_24[0][0]
                                                                   conv2d_17[0][0]
_________________________________________________________________________________________________
conv2d_25 (Conv2D)           (None, 80, 80, 3)         1731        add_1[0][0]
=================================================================================================
Total params: 1,110,403
Trainable params: 1,110,403
Non-trainable params: 0
_________________________________________________________________________________________________
```

Abbildung 11: Topologie des Entwurfes 3

Entwurf 3 (siehe Abbildung 11) basiert auf einen Artikel von Kapil Chauhan. Wie aus Abbildung 11 ersichtlich wird, wird hier Upsampling verwendet anstelle von Conv2D Transpose. Dies liegt daran, dass Conv2D Transpose trainierbare Parameter hat, während es für diesen Entwurf von Vorteil ist, die Trainingszeit zu verlängern und Conv2D Transpose für die geringe Datenmenge nicht benötigt wird. In diesem Entwurf werden zusätzlich *residual connection* verwendet. Diese finden zwischen der verbleibenden Verbindung des Codierer und Decodierer statt. Dies dient dazu, die verlustbehafteten Ergebnisse des endgültigen rekonstruierten Bildes zu verringern. (vgl. [10])

F. Trainingsphase des Modells

```
[15] early_stopper = EarlyStopping(monitor='val_loss', min_delta=0.0001, patience=4, verbose=1, mode='auto')

     log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
     tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

     a_e = autoencoder.fit(train_x_px, train_X,
                 epochs=50,
                 batch_size=256,
                 shuffle=True,
                 validation_data=(val_x_px, val_X),
                 callbacks= [early_stopper, tensorboard_callback])
```

Abbildung 12: Definierung des Early Stopper und Trainieren des Modells

Beim Training des Modells (siehe Abbildung 12) wurde ein *early_stopper* implementiert. Zu viele Epochen können zum overfitting des Trainingsdatensatzes führen, während zu wenige zu einem underfitting des Modells führen kann. *early stopping* ist eine Methode, mit der eine beliebig große Anzahl von Trainingsepochen angeben und das Training vorzeitig beenden werden kann, sobald das Modell keine besseren Ergebnisse erzielt. (vgl. [12]).

Diese Methode bricht den Lernprozess ab, wenn die Differenz zwischen *train_loss* und *val_loss weniger* als 0.0001 wird.

Mit der Integration von Tensorbaord in den Callbacks, können Resultate über den verschiedenen Entwürfen visualisiert und dargestellt werden.

Die Anzahl der Epochen (eng. epoch) ist ein Parameter, der definiert, wie oft der Lernalgorithmus den gesamten Trainingsdatensatz durchläuft. Im Gegensatz zu Epochen beschreibt die Stapelgröße (eng. batch size) einen Parameter, der die Anzahl der zu bearbeitenden Proben bestimmt, bevor die internen Modellparameter aktualisiert werden. (vgl. [13])

Die Epochenzahl wurde für alle drei Entwurf gleich gewählt und liegt bei 50 Epochen. Zu dem bleibt auch die Stapelgröße von 256 bei allen gleich.

G. Ergebnisse der Rekonstruktion

Sobald das Netz trainiert wurde, kann nun eine Reihe von Bildern getestet werden. Als Ausgabe erhält man das verpixelte Inputbild und das rekonstruierte Outputbild (siehe Abbildung 16). Im nächsten Schritt kann ein eigenes Bild hochgeladen und getestet werden.

Demonstrator

```
[ ] #in diesem Abschnitt wird der Demonstrator erstellt
    import numpy as np
    from google.colab import files
    from keras.preprocessing import image
    import cv2
    import matplotlib.pyplot as plt

    uploaded = files.upload()

    for fn in uploaded.keys():
      #in diesem Abschnitt laden wir die Bilder rein; wir nutzen eine for Schleife, um mehrere Daten gleichzeitig einladen zu können
      path = '/content/' + fn
      img = cv2.imread(path)

      #in diesem Abschnitt formartieren wir unsere Bilder, sodass sie zu unserem neuronalen Netz passen
      img = cv2.resize(img,(80,80), 3)
      img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
      img = pixalate_image(img)

      x = image.img_to_array(img, dtype=np.float32)
      x = x / 255.0
      x = x.reshape(1, 80, 80, 3)


      #Das hochgeladene Bild wird hier Predicted
      pic = autoencoder.predict(x)

      #Ausgabe des Input und Output
      n = 1
      plt.figure(figsize= (20,10))

      for i in range(n):
        ax = plt.subplot(2, n, i+1)
        print(img.shape)
        plt.imshow(x[0])
        ax.get_xaxis().set_visible(True)
        ax.get_yaxis().set_visible(True)

        ax = plt.subplot(2, n, i+1+n)
        print(pic.shape)
        plt.imshow(pic[0])
        ax.get_xaxis().set_visible(True)
        ax.get_yaxis().set_visible(True)

      plt.show()
```

Abbildung 13: Topologie des Entwurfes 3

Im Codeblock des Demonstrators (siehe Abbildung 13) kann ein beliebiges Bild von einem Gesicht hochgeladen werden. Das Bild wird auf die Inputgröße und Farbraum angepasst, damit dieses Bild für das Testen verwendet werden kann. Dieses Bild wird anschließend verpixelt. Nachdem es verpixelt wurde, kommt es zum *predicten* des Bildes.

VII. Ergbenisse

Im Folgenden werden die Ergebnisse der einzelnen Entwürfe ausgewertet und verglichen.

A. Entwurf 1

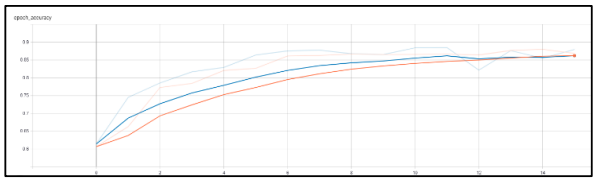


Abbildung 14: Accuracy Graph des ersten Entwurfs

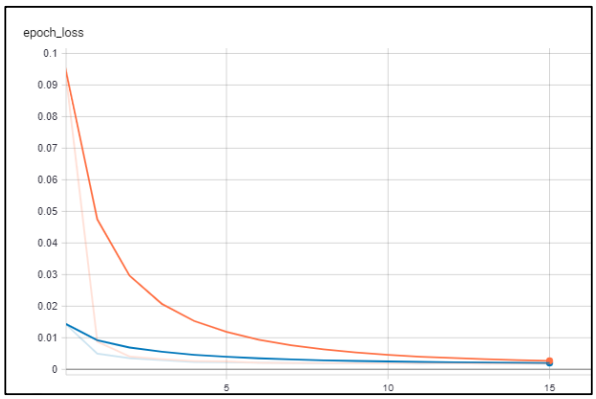


Abbildung 15: Loss Graph des ersten Entwurfs

Der erste Entwurf hat am besten abgeschnitten. Wie man in Abbildung 14 sehen kann, nimmt der *Loss* bis zur 15 Epoche ab und die *Accuracy* (siehe Abbildung 14) des Validierungsprozesses zu. Der Accuracywert liegt hier bei 0.86 und der Losswert liegt bei 0.0018. Das Modell hat nach der 15 Epoche abgebrochen, da der Callback gegriffen hat.

Wie man in Abbildung 16 sehen kann, ist das Ergebnis des Entwurfs sehr gelungen. Die Kanten vom Inhalt des Bildes bleiben gut erhalten. Die Verpixelung des Bildes wurde stark minimiert.

Abbildung 16: Ergebnis des ersten Entwurfes 1

B. Entwurf 2

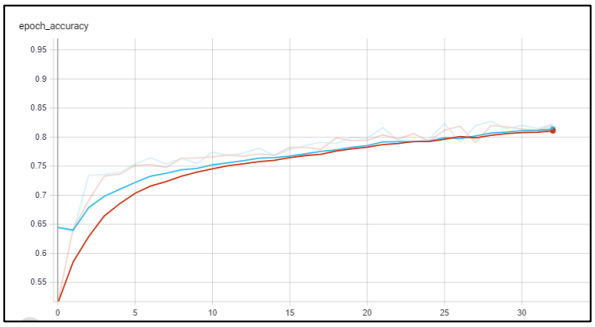


Abbildung 17: Accuracy Graph des zweiten Entwurfs

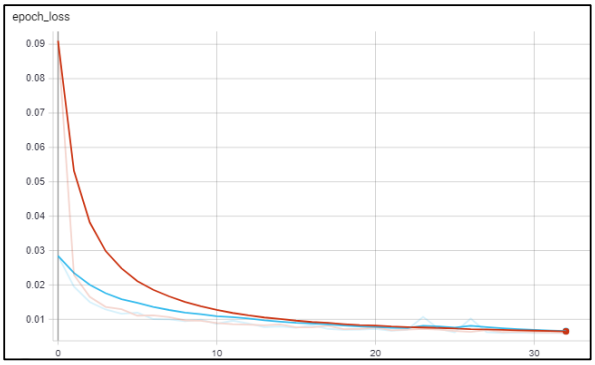


Abbildung 18: Loss Graph des zweiten Entwurfs

Der zweite Entwurf hat gut abgeschnitten. Wie man in der Abbildung 18 sehen kann, nimmt der *Loss* bis zur 35 Epoche ab und die *Accuracy* in der Abbildung 17 des Validierungsprozesses zu. Der Accuracywert liegt hier bei 0.81 und der Losswert liegt bei 0.005. Das Modell hat nach der 35 Epoche abgebrochen, da der Callback gegriffen hat.

Abbildung 19: Ergebnis des ersten Entwurfes 2

Das Ergebnis ist jedoch schlechter ausgefallen als Ergebnis von Entwurf 1. Es wird hier klar ersichtlich, dass bei dieser Methode die scharfen Kanten des Bildes vollkommen verloren gehen. Das ist zurückzuführen auf die höhere Anzahl von Convolutional Layer die mehr Features extrahieren. Ziel dieses Entwurfes war es durch ein stärkeren Informationsgewinn der Bilder ein besseres Ergebnis zu erzielen. Wie es das Ergebnis zeigt, hat dieser Ansatz zu keiner relevanten Lösung geführt.

C. Entwurf 3

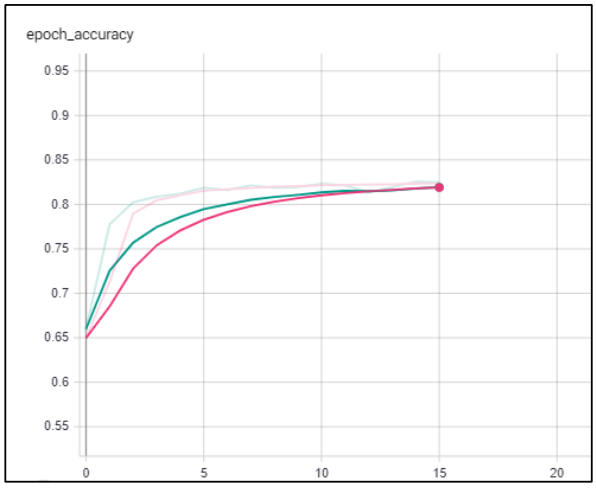


Abbildung 20: Accuracy Graph des zweiten Entwurfs

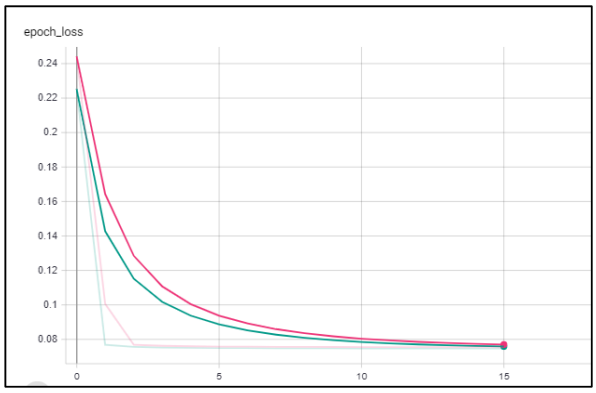


Abbildung 21: Accuracy Graph des zweiten Entwurfs

Der dritte Entwurf hat am schlechtesten von allen dreien Entwürfen abgeschnitten. Wie man in Abbildung 21 sehen kann nimmt der *Loss* bis zur 15 Epoche ab und die *Accuracy* in der Abbildung 20 des Validierungsprozesses zu. Der Accuracywert liegt hier bei 0.825, und der Losswert liegt bei 0.07, welcher somit den höchsten von allen Entwürfen ist. Das Modell hat nach der 15 Epoche abgebrochen da der Callback gegriffen hat. Dadurch dass in diesem Entwurf *residual connection* benutzt worden ist und diese die Möglichkeit besitzen Layer zu überspringen, können die Resultate dieses Entwurfes variieren. Nach mehreren Tests kamen öfters unterschiedliche Ergebnisse raus.

Abbildung 22: Ergebnis des ersten Entwurfes 3

Das Ergebnis aus der Abbildung 22 ist fast identische zum Ergebnis vom Entwurf 2. Es wird hier auch ersichtlich, dass bei dieser Methode die scharfen Kanten des Bildes verloren gehen. Das ist, wie bei Entwurf 2, zurückzuführen auf die höhere Anzahl von Convolutional Layern. Des Weiteren kann das schlechtere Ergebnis anhand des Loss erklärt werden, der bei diesem Entwurf sehr hoch war.

VIII. FAZIT

Ein tieferes Neuronales Netzwerke heißt nicht unbedingt das bessere Resultate generiert werden können, wie es diese Studienarbeit zeigt. Um jedoch bessere Resultate zu generieren, wäre ein größere Datensatz der erste Schritt. Möchte man Bilder mit einer besseren Auflösung, wie Full HD oder sogar 4K, rekonstruieren wollen, sollte der Datensatz auch mit diesen besetzt sein. Eine weitere Methode diesen Ansatz zu verbessern, wäre das Anpassen der Entwürfe. Zum Beispiel könnten die Upscaling Layers, die auf der *nearest neighbour Methode* beruhen, mit Conv2D Transpose Layer ausgetauscht werden. Conv2D Transpose Layer ermöglicht es dem Kernel während dem Training zu lernen, damit er selbst entscheiden kann, welche die beste Option der *upscaling Operation* wäre. Allerdings wird dazu ein viel größerer Datensatz benötigt, damit es effizient ist. Zusätzlich könnte an der Loss Funktion gearbeitet werden. Dadurch dass hier mit der Total by Pixel Loss gearbeitet wird, werden hier oftmals scharfe Kante ignoriert, welches zum Beispiel bei dem Arbeiten mit Gesichtern zu gravierend schlechteren Ergebnissen führen kann. Darüber hinaus könnte man den CAN auch an ein GAN (generative adversarial network) ankoppeln, um bessere Ergebnisse zu erzielen. Alles in allem konnte in dieser Studienarbeit sehr viel Erfahrung im Bereich des angewandten maschinellen Lernens, speziell im Gebiet der Autoencoder, gesammelt werden.

LITERATURVERZEICHENIS

[1] "Künstliche neuronale Netze – Wikipedia" [Online]. Available: https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz [Accessed 01.08.2020].

[2] "Kuenstliches Neuron – Wikipedia" [Online] Available: https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron [Accessed 01.08.2020]

[3] "SuperVize Me: What's the Difference Between Supervised, Unsupervised and Reinforment Learning - Nvida" [Online] Available: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/#:~:text=In%20a%20supervised%20learning%20model,and%20patterns%20on%20its%20own. [Accessed 01.08.2020]

[4] Lanquillon C. (2019) Grundzüge des maschinellen Lernens. In: Schacht S., Lanquillon C. (eds) Blockchain und maschinelles Lernen. Springer Vieweg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-60408-3_3

[5] „Introduction to autoencoders. – Jeremy Jordan" [Online]. Available: https://www.jeremyjordan.me/autoencoders/ [Accessed 01.08.2020]

[6] "From Autoencoder to Beta-VAE – Lil'Log" [Online] Available: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html#autoencoder [Accessed 01.08.2020]

[7] J. Masci, U. Meier, D. Ciresan, and J. Schmidhuber, „Stacked Coilutional Auto-Encoder for Hierarchical Feature Extraction", Istituto Dalle Molle di Studi sullÍntelligenza Artificale (IDSIA) [Online]. Available: http://people.idsia.ch/~ciresan/data/icann2011.pdf [Accessed 01.08.2020]

[8] X. Guo, X. Liu, E. Zhu and J. Yin, „Deep Clustering with Autoencoders", College of Computer and National University of Defense Technology, Chansha, China, Oktober 2017. [Online]. Available: https://xifengguo.github.io/papers/ICONIP17-DCEC.pdf [Accessed 01.08.2020]

[9] "Convolutional neural network - Wikipedia" [Online]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network#Convolutional [Accessed 01.08.2020].

[10] "Labeled Faces in the Wilde Home – Labeled Faces in the Wild" [Online] Available: http://vis-www.cs.umass.edu/lfw/#:~:text=Welcome%20to%20Labeled%20Faces%20in,name%20of%20the%20person%20pictured. [Accessed 01.08.2020]

[11] "Super-Resolution Using Autoencoder and TF2.0 – Kapil Chauhan" [Online]. Available: https://medium.com/analytics-vidhya/super-resolution-using-autoencoders-and-tf2-0-505215c1674 [Accessed 01.08.2020]

[12] "Use Early Stopping to Halt the Training of Neural Networks At the Right Time – Jason Brownlee" [Online]. Available: https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/ [Accessed 01.08.2020]

[13] "Difference Between a Batch and an Epoch in a Neural Network – Jason Brownlee" [Online]. Available: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/ [Accessed 01.08.2020]

Deep-Q-Learning mit 'Super Mario Bros' und A3C.

Jan Bernd Gaida
jan.gaida@hof-university.de
Hochschule für Angewandte Wissenschaften Hof
Hof an der Saale, Bavaria, Germany

ZUSAMMENFASSUNG

In dieser Studienarbeit wird die Performance unterschiedlicher strukturierter Reinforcement-Learning (kurz: RL) Modelle innerhalb einer *Super Mario Bros.* (1985) [7] (kurz: SMB) Umgebung hinsichtlich ihrer Lernperformance und -stabilität auf Basis eines Experiments untersucht.

Konkret werden unterschiedliche RNN-Architekturen mit unterschiedlichen *CNN*-Architekturen hinsichtlich ihres Performance bei limitierter Hardware verglichen. Dazu musste sich zunächst auf einen RL-Algorithmen geeinigt werden und anschließend das Neuronale Netzwerk und dessen Hyperparameter optimiert werden.

Abschließend wird das Ergebnisse des Projekts diskutiert.

Der Quellcode für das Projekt und sowie dessen Ergebnisse stehen öffentlich als Repository zur Verfügung [10].

1 EINFÜHRUNG

Maschinelles Lernen ist die Wissenschaft Computer indirekt zu programmieren. In der vergangen Dekade ist der Einsatz solcher Programme so allgegenwärtig geworden, das Vielen ihre eigentliche Verwendung nicht bewusst ist obwohl Sie mehrmals täglich verwendet wird. [18]

Reinforcement-Learning ist ein junger Teilbereich des Maschinelle Lernens, bei dem das Programm eine Umgebung eigenständig erkundet und in Abhängigkeit des Problems eine konkrete Lösung definiert. Dieses Verständnis erlangt das Programm mittels '*Reinforcement*' (deutsch: bekräftigende, bestätigende) Mechanismen.

Medienwirksame Aufmerksamkeit erlangt dieser Teilbereich durch beeindruckende Rekorde oder Siege in sehr komplexen Spielen wie beispielsweise '*AlphaGo*', das erste Computerprogramm das einen menschlichen Spieler im dadurch berühmten Brettspiel **Go** besiegte [8]. Oder beispielsweise die künstliche Intelligenz '*OpenAI Five*' die das Multiplayerspiel *Dota 2* auf Experten Niveau spielt [22].

Auch wenn dieser Bereich scheinbar keine konkreten Verbesserungen für das alltägliche Leben birgt, bietet das experimentieren mit solchen komplexen künstlich geschaffenen Umgebungen ein großer Forschungspotential. Dies wird langfristig dazu führen, dass vor allem Roboter in weniger komplizierten Anwendungsbereichen, wie beispielsweise dem korrekten Sortieren von Objekten von Laufbändern, großer Fortschritt machen wird [9].

Das Potential von Reinforcement-Learning ist also groß und wird uns sicherlich auch weiterhin dazu Anregen, dieses weiter zu entwickeln und im Alltag nutzbar zu machen.

2 HARDWARE

Das Training des Modells fand auf einem *Consumer-Pc* statt. Konkret standen hierfür 6 GB VRAM (*GeForce GTX 1060*) erreichbar mittels *CUDA*-Treiber, sowie 16 GB RAM (DDR3 im Dual-Channel Modus bei ca. 800 MHz Frequenz) unterstützt durch eine 4 Kern CPU getaktet auf ca. 3.8 GHz (*Intel Core i5 4690K*). Genannte Hardware wurde zum Trainigs- und Testzeitpunkt moderat übertaktet.

3 LERN- UND TESTUMGEBUNG

Als Laufzeit- und Entwicklungsumgebung wurde Python 3.8 (offizielle Releaseversion) gewählt. Nachfolgend werden die wesentlichen benutzten Frameworks kurz aufgeführt, welche allesamt mittels Pip-Paketmanager installierbar sind.

3.1 OpenAI Gym und SMB-Gym

OpenAI Gym [6] ist ein bekanntest Framework zum trainieren und testen unterschiedlicher RL-Algorithmen. Ziel der Entwickler ist es hierbei für diesen Algorithmentypus eine standardisierte Umgebung, meist als Enviorment bezeichnet, zur bessere Vergleichbarkeit mit ähnlichem anzubieten. Dadurch eignet sich dieses Framework bestens für sog. Benchmarktests mit weiteren RL-Algorithmen, mit denen Vor- und Nachteile dieser Algorithmen analysiert und interpretiert werden können.

Durch die Machine-Learning-Community gibt es eine große Anzahl an verfügbaren '*Gyms*', wie das in diesem Projekt genutzte: '*Super Mario Bros for OpenAI Gym*' [15]. Dieses ist ein von Christian Kauten zur Verfügung gestelltes SMB-Enviroment, welches die letzte Version von *OpenAI Gym* (Stand: 2020) unterstützt und weitere Features beinhaltet. So werden in diesem *Gym* unterschiedliche Render-Versionen und vordefinierte Eingabekombination (sog. ActionSpace) angeboten (vgl. 5.1).

Zur Verfügung steht dieses Enviorment aus urheberrechtlichen Gründen nur für Bildungszwecke, vgl. *Digital Millennium Copyright Act* (kurz: *DMCA*) [2].

3.2 PyTorch

Als Machine-Learning-Framework wurde *PyTorch* [4], ein bekanntes und speichereffizientes open source Framework, verwendet. Dieses wurde ursprünglich von einem *Facebook*-Forschungsteam für künstliche Intelligenz entwickelt und basiert auf der in *Lua* geschriebenen Bibliothek *Torch* [3]. [19]

3.3 Tensorboard und Ähnliche Bibliotheken

Zur Visualisierung der Lernergebnisse wurde *Tensorboard* [1] verwendet. Hierbei handelt es sich um ein *Tensorflow*-Toolkit, welches mit kleineren Einschränkungen ebenfalls mit *PyTorch* verwendbar ist. Es verfügt über einige diagrammspezifische Funktionalitäten und wird hauptsächlich für die Interpretation von ML-Experimenten verwendet [11].

Für einen größeren Funktionsumfang von *Tensorboard* mit *PyTorch* wurde die *TensorboardX*-Bibliothek verwendet. Diese bietet

neben einigen zusätzlichen Funktionen, auch Lösungen für Probleme, die aus fehlenden Abhängigkeiten zwischen diesen beiden Bibliotheken entstehen.

Für eine verbesserte Darstellung innerhalb der Commandline wurde die *TorchSummaryX*-Bibliothek verwendet. [5]

3.4 OpenCV

Für das sog. *Preprocessing* (deutsch: vorverarbeiten) (vgl. 5.3) wurde die mächtige open source Bildverarbeitungsbibliothek *OpenCV* verwendet. Diese verfügt neben der Unterstützung von unterschiedlichen Bild- und Farbformaten auch über performante Bildbearbeitungsfunktionen, wie bspw. das Umwandeln des Farbschemas, das Ändern von Helligkeit und Kontrast, sowie weiteren fortgeschritteneren Bildbearbeitungsfunktionalitäten. [14]

4 REINFORCEMENT-LEARNING-ALGORITHMUS

Für das Experiment selbst musste sich zunächst für einen Algorithmus entschieden werden. Dieser sollte möglichst unkompliziert, als auch auf der zur Verfügung stehenden Hardware (vgl. 2) möglichst performant sein, sodass der Trainingsprozess für ein SMB-Level wenige Stunden dauert. Hierfür wurde drei inkrementell komplexere Algorithmen implementiert und miteinander verglichen. Diese werden nachfolgend kurz erläutert.

4.1 Deep-Q-Network-Ansatz

Ausgehend von dem *Deep-Q-Network-Algorithmus* (kurz: DQN) zeichnete sich die Wichtigkeit der genannten Ziele bei der Auswahl des Algorithmus ab. Nach zwei Stunden Training kam es hier zu keinem Lernerfolg: Mario bewegte sich nicht, oder in die falsche Richtung. Grundlegend wird hier eine sogenannte *Q-Table* erzeugt, bei der den Zuständen die bestmöglichste Aktion zugeordnet wird (s.h. Abb. 1). [17]

Auch wenn davon auszugehen ist, dass mit diesem Algorithmus das Lernziel erreichbar ist, ist der Traingszeitrahmen für alle konkreten Netzwerke des Experimentes deutlich größer, wodurch die Verwendung dieses Algorithmus im Rahmen dieser Studienarbeit ungeeignet scheint.

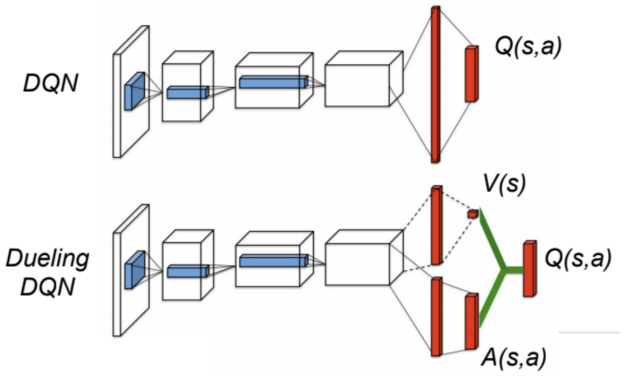


Abbildung 1: Aufbau DCN und DDCN [16]

Anschließend wurde die Tauglichkeit des *Duelling Deep-Q-Network-Algorithmus* (kurz: D-DQN) untersucht. Hierbei handelt es sich, wie der Name bereits andeutet, um eine Weiterentwicklung des vorherigen *DQN-Algorithmus*: der zuvor erläuterten *Q-Table* wird hierbei nun auch ein Wert hinzugefügt, welcher den aktuellen Zustand bewertet (s.h. Abb. 1).

Es stellte sich allerdings auch hier nach zwei Stunden Training kein sichtbarer Lernerfolg ein, wodurch auch dieser im Rahmen dieser Studienarbeit ungeeignet ist.

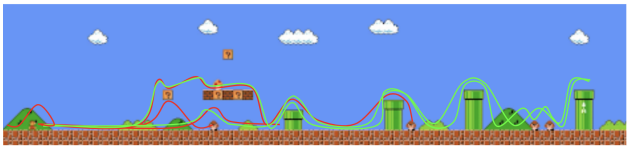


Abbildung 2: Mögliche Laufwege in SMB (Org.: [23])

Ursächlich für den geringen Lernerfolg ist hierbei die Komplexität des ursächlichen Problems bzw. der Aufgabe: Betrachtet man die Menge an Zuständen, so fällt auf, dass diese potentiell gegen unendlich läuft, ebenso ist es bei der Menge an möglichen Aktionen.

Betrachtet man Abbildung 2 so sind ein paar Laufwege von Mario eingezeichnet. Mit grünen Linien sind die erfolgreichen Laufwege markiert, in rot hingegen sind erfolglose Laufwege markiert.

Genannte gegen unendlich laufende Menge an Zuständen entspricht in der Grafik jeden Punkt auf einer Linie, die innerhalb des SMB-Enviorments auch gerendert werden kann. Die ebenfalls gegen unendlich laufende Menge an Aktionen entspricht hierbei der Anzahl an zeichenbaren Laufwegen, welche den Regeln des SMB-Enviroments folgen.

Diese beiden Mengen entsprechen den *Input* und den *Output* des Modells und definieren dadurch wichtige Eigenschaft des zur Lösung geeigneten Algorithmus.

Dies in Kombination mit der Tatsache des Trainierens auf einem einzigen Thread erklärt die Überlegenheit des nachfolgenden Algorithmus. [17], [24]

4.2 A3C-Ansatz

Ein *Reinforcement-Learning-Algorithmus*, der hingegen besser Performance bei dieser Komplexität verspricht ist der *Asynchronous-Advantage-Actor-Critic-Algorithmus* (kurz: A3C).

Im Gegensatz zur den vorher genannten Algorithmen wird hier keine *Q-Table* erzeugt. Stattdessen wird unter Berücksichtigung einer geschätzten Bewertung des aktuellen Zustandes vom sog. *Critic* (deutsch: Kritiker), eine Aktion ausgeführt vom sog. *Actor* (deutsch: Akteur). Der *Actor* und der *Critic* werden hierbei als separate Neuronale-Netzwerke definiert, die den exakt gleichen Input erhalten müssen.

Die Idee des A3C-Algorithmus lässt sich also mit folgender Metapher gut erklären.

> Let['s] imagine a small mischievous child (actor) [which] is discovering the amazing world around him, while his dad (critic) oversees him, to make sure that he does not do anything dangerous. Whenever the kid does anything good, his dad will praise and encourage

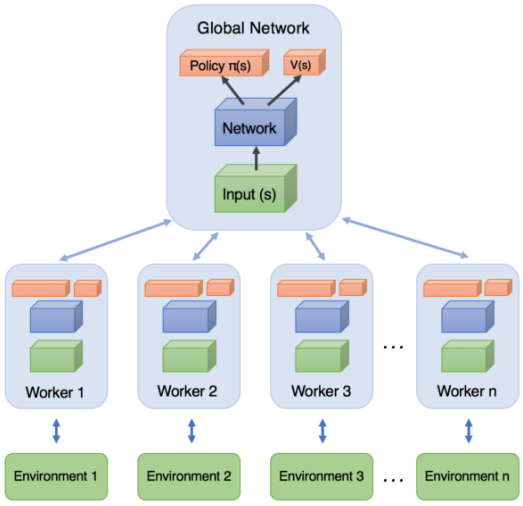


Abbildung 3: Aufbau A3C [13]

> him to repeat that action in the future. And of course, when the kid does anything harmful, he will get [a] warning from his dad. The more the kid interacts [with] the world, and takes different actions, the more feedback, both positive and negative, he gets from his dad. The goal of the kid is, to collect as many positive feedback as possible from his dad, while the goal of the dad is to evaluate his son's action better. In other word, we have a win-win relationship between the kid and his dad, or equivalently between actor and critic.[20]

Dieser Ansatz wird durch parallel agierenden Modelle erweitert, welche wiederum regelmäßig (in der Regel abhängig von der definierten Batchgröße) ein gemeinsames Model (sog. *globales Model*) aufbauen bzw. trainieren und sich auf dessen Lernstand selbst aktuell halten.

Mit diesem Algorithmus kam es bereits nach wenigen Minuten zu einem sichtbaren Lernerfolg, nach 2 Stunden wurde so das Lernziel des Experiment, das Abschließen eines Level, bereits erreicht. Es zeichnete sich also ab, dass dieser Algorithmus bestens geeignet ist für das eigentliche Experiment. [21], [13]

5 ENVIROMENT

Wie bereits erwähnt kommt in diesem Experiment ein SMB-Enviroment zum Einsatz (vgl. 3.1), nachfolgend wird die Konfiguration und der Einsatz dieses SMB-Enviroment genauer betrachtet.

5.1 Konfiguration

Auf Basis von Vorab-Experimenten wurde sich für die Standard Render-Version entschieden, welche eine minimal schlechtere Lernperformance - in diesem Fall das erstmalige Erreichen des Zieles - als die anderen bereitgestellten grafisch reduzierten Render-Versionen aufwies.

Ausgehend von den selben Vorab-Ergebnissen wurde sich für eine leicht reduzierte Eingabemenge (im '*Super Mario Bros for OpenAI Gym*' [15] 'Complex' genannt) entschieden. Diese beinhaltet im wesentlichen neben einer leeren Eingabemenge auch Kombinationen aus den Nach-Links- oder Nach-Rechts-Tasten mit der A- oder bzw. und B-Taste des *NES*-Controllers. Die weiteren reduzierten Eingabekombination hatten einen zustandsabhängigen Einfluss auf das Ergebnis, wodurch einige Level des Enviroments merklich schneller andere merklich langsamer erlernt wurden.

5.2 Weitere Modifikationen

Mittels *Wrapper-Pattern* wurde der *Output* des ursprünglichen Enviorments modifiziert. Nachfolgend werden diese Modifikationen von innen nach außen kurz erläutert.

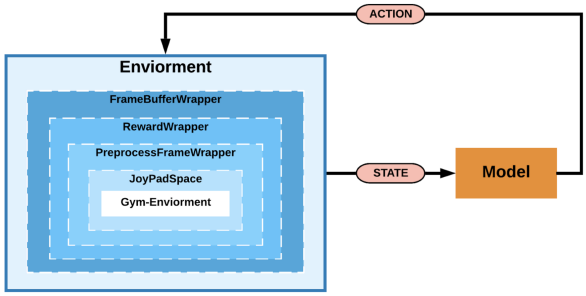


Abbildung 4: Schematischer Kontext des konkret implementierten Enviorments

5.2.1 *JoypadSpace.*

Der *JoypadSpace-Wrapper* ist eine von *OpenAI* zur Verfügung gestellte Implementation des *OneHot-Encodings* für das *NES*.

5.2.2 *PreprocessFrameWrapper.*

Der *PreprocessFrameWrapper* vorverarbeitet den ursprünglichen berechneten Frame (vgl. 5.3).

5.2.3 *RewardWrapper.*

Der *RewardWrapper* überschreibt den ursprünglich berechneten Rewardwert (vgl. 5.4).

5.2.4 *FrameBufferWrapper.*

Der *FrameBufferWrapper* fasst einige berechnete Frames in ein Array zusammen, um so dem Model die Chance zu bieten Bewegungen zu erkennen.

5.3 Preprocessing

Das *Preprocessing* dient zur Vorverarbeitung des Frames. Dadurch soll das Bild so angepasst werden, dass das zutrainierende Model damit bestmöglich rechnen kann. In der konkreten Implementation kann dieser Schritt in wiederum vier Teilschritte unterteilt werden, die nachfolgend kurz begründet werden.

5.3.1 *Zuschneiden.*

Zuerst wird der originale 240x256 Pixel große Frame auf 200x256 Pixel verkleinert. Konkret werden einige Pixel oben und unten weggeschnitten, welche (fast) keine Relevanz für den Spielverlauf haben.

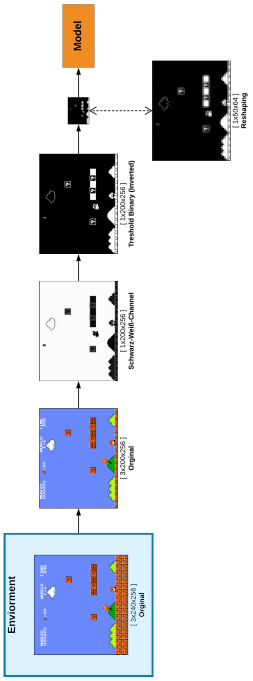


Abbildung 5: Schematische Preprocessing

5.3.2 *Schwarz-Weiß.*

Im nächsten Schritt wird der Frame zu einem verlustfreien Schwarz-Weiß-Bild vereinfacht.

5.3.3 *Schwarz-Oder-Weiß.*

Anschließend wird der Frame mittels *Binary-Treshhold* zu Schwarz- oder Weißenfarbtönen vereinfacht. Damit diese sich auch optisch von dem vorherigen Schritt leicht unterscheiden lassen, werden hierbei die Farben invertiert.

5.3.4 *Verkleinern.*

Abschließend wird das Bild im korrekten Seitenverhältnis verkleinert. Dies hat u.a. Einfluss darauf wie viel Speicher das Model in allen Ebenen benötigt und wurde im Rahmen des Experimentes auf 50x64 Pixel gesetzt (vgl. 2) - dies entspricht nahezu der minimalen informationsverlustbehafteten Verkleinerung des Bildes, die im Rahmen diverser Experimente festgelegt worden ist.

5.4 Reward

Der *Reward* definiert das mathematisch entstehende Verständnis des Modells über das Enviorment. Hierfür gibt es in der Regel zwei unterschiedliche Ansätze, wie dieser *Reward* vergeben werden kann: Per *Sparse Reward* oder mittels *Reward Shaping*. Beim sog. *Sparse Reward* wird der *Reward* nur beim Erreichen wesentlicher Ziele vergeben. Beim *Reward Shaping*, welches für das Experiment gewählt wurde, wird hingegen durch kontinuierliches Vergeben des *Reward* das mathematische Verständnis des Modells sehr stark beeinflusst, wodurch eine starke Abhängigkeit zwischen der Qualität der *Reward-Funktion* und dem Lernergebnisses ensteht.

Im der konkreten Implementation besteht der *Reward* aus 4 Werten, die nach einer Addition in einen geringeren Wertebereich verschoben werden:

$$F_{Reward} = (\Delta X + \Delta Zeit + R_{Ziel} + R_{Leben}) \ / \ 10 \tag{1}$$

Dem Delta der zurückgelegten Strecke addiert mit der verstrichenen Zeit, zusätzlich den fest definierten Belohnungswerten für das erreichen des Zieles bzw. für das verlieren eines Lebens. Diese Variablen lassen sich wie folgt definieren:

$$\Delta X = X(n+1) - X(n) \tag{2}$$

$$\Delta Zeit = T(n+1) - T(n) \tag{3}$$

$$R_{Ziel} = 45 \ if \ goal \ achieved \ else \ 0 \tag{4}$$

$$R_{Leben} = -45 \ if \ life \ lost \ else \ 0 \tag{5}$$

Ziel des Modells für die *Reward-Funktion* ist es also eine möglichst große Distanz in geringer Zeit zu erreichen, bei denen im bestenfalls das Ziel erreicht wird und idealerweise kein Leben verloren wird.

Experimente mit weiteren Faktoren wie etwa einer Belohnung für das Sammeln von Münzen oder dem Verhindern des 'Stehenbleibens' hatten meist negative Auswirkungen auf das eigentliche Hauptziel.

6 NETZWERK-ARCHITEKTUR

Die zugrundeliegende Netzwerk-Architektur für den *A3C-Algorithmus* erfordert zwei separate Netzwerke (*Actor* und *Critc*), die wiederum sich den selben *Input* teilen. Dieser *Input* entsteht durch visuelle Verarbeitung mittels *Convolutional-Netzwerkes* (kurz: CNN) und anschließend durch rückgekoppelte Verarbeitung mittels *Rekurrentem-Netzwerkes* (kurz: RNN) mit dem Ziel der reihenfolgeabhängigen Verarbeitung. Auf all diese Bestandteile des implementierten Neuronalen Netzwerkes wird nachfolgend eingegangen (vgl. Abb. 10).

6.1 Convolutional-Netzwerk

Wie erwähnt dient das *CNN* dazu grafische nicht näher definierte Muster innerhalb des Frames zu deuten. Es wurden hierbei zwei unterschiedliche Architekturen definiert, die nachfolgend kurz erklärt werden. Beide Ansätze hatten zum Ziel die *Tensorgröße* zu minimieren, um dadurch die limitierte Hardware (vgl. 2) bestmöglichst auszunutzen.

6.1.1 *Naive CNN.*

Bei dem Naiven Ansatz des *CNN* wird ein initial 320 Channel großer Tensor konstant um 80 Channel, also ursprünglich um 25%, verkleinert. Hintergrund hierfür ist, neben dem Ziel den Tensor klein zuhalten, dass mehrere Frame-Features zu einem einzigen Signal inkrementell zusammengefasst werden sollen. Als Kernelgröße wurde 3x3 gewählt, bei einem Stride von 2, welcher u.a. den Tensor konstant reduziert.

Während der Entwicklungsphase gab es keine wesentlichen Unterschiede bei ähnlich hoher Start-Channel-Anzahl und Channel-Reduktionsfaktor, weshalb die Größen so gewählt wurden, dass der verfügbare Arbeitsspeicher voll ausgelastet wurde.

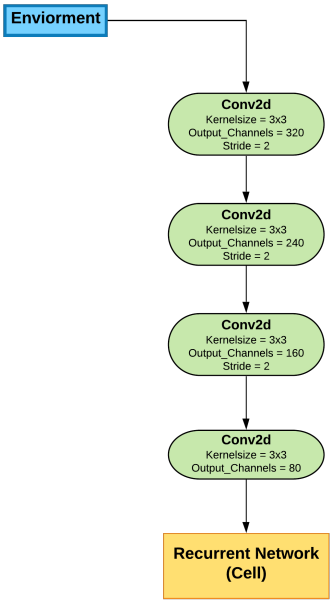


Abbildung 6: Naive CNN-Architektur

6.1.2 *Deep(ish)-CNN.*

Ein weiterer Ansatz für den *CNN*-Teil ist inspiriert von den *GoogLeNet*-Blöcken [12]. Im Gegensatz zu den vorherigen *CNN*-Ansatz kamen hier nach jedem *CNN*-Layer eine *ReLU*-Aktivierungsfunktion zum Einsatz. Wie auch in dem naiven *CNN*-Ansatz wird hier mittels Stride die Tensorgröße effektiv reduziert.

Abweichend zu *GoogLeNet* wurden hier im zweiten Branch drei 3x3 große Kernel und im dritten Branch zwei 5x5 große Kernel verwendet. Eine Reduktion der Rechenleistung durch das Ersetzen der 5x5 Kernel mittels zwei 3x3 Kernel ist hierbei aufgrund der Strides nicht möglich. Die Channelanzahl der einzelnen Branches wird unterschiedlich stark auf 32 reduziert, wodurch im letzten Verknüpfungschritt 128 Channel erreicht werden.

Die wesentlichen Abweichungen hinsichtlich der Kernelgröße und -anzahl sind hauptsächlich experimentell zu erklären. Geringe Änderungen hierbei hatten meist eine längeren Lernzeit für ähnliche Lernergebnisse zufolge. Die Anzahl an Channels ist ähnlich wie bei dem Naiven *CNN*-Ansatz mit der Hardware-Ausnutzung begründet (vgl. 2).

6.2 Rekurrent-Netzwerk

Hinsichtlich des Rekurrent-Netzwerk stellte es sich als problematisch heraus, dass zu dem Zeitpunkt, an dem ein *CNN-Output* generiert wurde das Training noch nicht abgeschlossen wurde, weshalb kein allgemeines *RNN* implementiert wurde. Stattdessen wurde

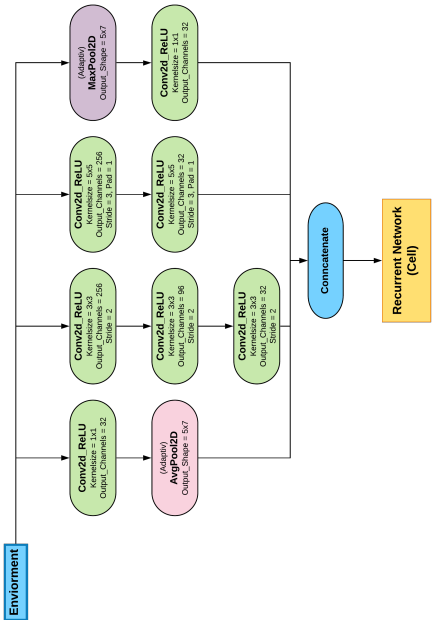


Abbildung 7: Deep(ish)-CNN-Architektur

eine Zelle implementiert, die als Teilinput den Output der selben Zelle zu dem vorherigen Schritt des Enviorments erhält.

Da der *RNN*-Teil für das absolvieren SMB ähnlicher Enviormente eine herausragende Rolle spielt, wurde die Channelanzahl innerhalb des kompletten Netzwerk maximiert. Da dies allerdings dazu führt, das bei der zur Verfügung stehenden limitierten Hardware, ein stark ansteigender Arbeitsspeicherverbrauch entsteht, wodurch diverse Abhängigkeiten innerhalb des Netzwerks und dem Enviroment ensteht, fehlten zum Zeitpunkt des Experimentes weitere Untersuchungen hinsichtlich der optimalen Werte.

6.2.1 *LSTM-RNN.*

Ein Ansatz für das *RNN* war die Verwendung der *LSTM-Zelle*. Hier wird im Vergleich zum folgenden *GRU*-Ansatz zusätzlich der *Hidden-State* ebenfalls pro Berechnungsschritt übergeben. (Die Input-Channelanzahl ist Abhängig von der Output-Channelanzahl des *CNN*-Teils.)

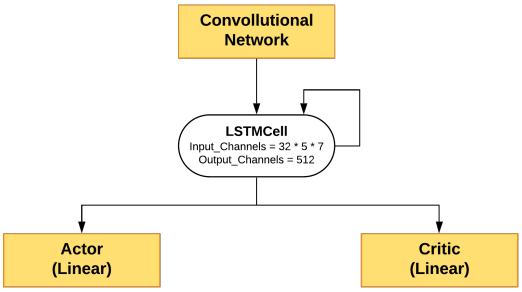


Abbildung 8: LSTM-Architektur

6.2.2 GRU-RNN.

Ähnlich wie die *LSTM-RNN*-Variante wurde die *GRU*-Zelle verwendet. (Die Input-Channelanzahl ist Abhängig von der Output-Channelanzahl des *CNN*-Teils.)

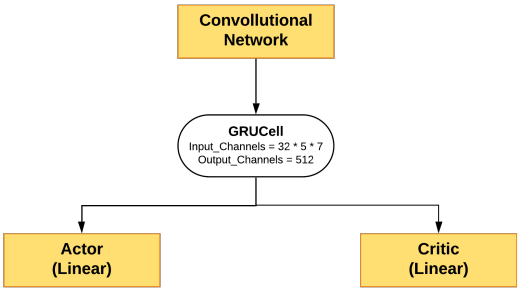


Abbildung 9: GRU-Architektur

6.3 Actor und Critic

Der *Actor*- und *Ciritc*-Teil des neuronalen Netzwerkes spielt eine entscheidende Rolle für das Bestimmen der Aktionen. Als *Linearer-Layer* implementiert unterscheiden sich die beiden hauptsächlich im Output: Der *Kritiker* berechnet einen einzigen Wert - die Bewertung des aktuellen Zustandes (vgl. 3).

Der *Akteur-Output* hingegen entspricht der Anzahl der verfügbaren Aktionen (vgl. 5.1 und beinhaltet die Wahrscheinlichkeit das diese die bestmöglichste Aktion ist. Diese wird im weiteren Verlauf nach der wahrscheinlichsten Aktion gefiltert und zurück zu einer, für das Enviorment verständlichen Aktion, umgewandelt.

6.4 Loss-Funktion

Damit sich die zuvor aufgeführten *Linearen-Layers* während des Trainings unterschiedlich entwickeln, werden diese in der Verlustfunktion mit unterschiedlichen Vorzeichen addiert abzüglich des konstanten Anteils des vorherigen Verlusts. Dieses Ergebnis wird innerhalb des Netzes schließlich back propagiert.

$$loss = -loss_{actor} + loss_{critic} - (beta * loss_{entropy}) \quad (6)$$

6.5 Hyperparameter

Für das komplette Experiment wurden im üblichen Wertebereich liegende *Hyperparamter* verwendet:

$$lr = 0.0001 \quad (7)$$

$$gamma = 0.9 \quad (8)$$

$$beta = 0.01 \quad (9)$$

7 EXPERIMENT UND ERGEBNISSE

Für das folgende Experiment wurden auf Basis der Hardware (vgl. 2) fünf parallel Trainings-Threads gestartet. Diese arbeiteten jeweils eine vorgegebene Anzahl von Epochen ab. Zusätzlich zu den

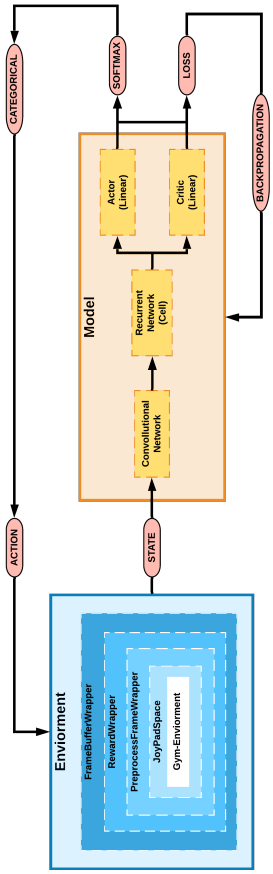


Abbildung 10: Vollständiger Kontext des Modells

Trainings-Thread, wurde ein Thread gestartet, welcher den aktuellen Zustand des globalen Modells wiederspiegelte. Während des Trainings lief zudem zur Live-Ansicht der Lernergbnisse im Hintergrund der *Tensorboard-Service*.

In den nachfolgenden Abbildungen 12, 13 und 14 sind die Farben jeweils nach der Legende (Abb. 11) gewählt. Zusehen sind hier die Ergebnisse nach 3500 Trainingsepisoden bei 5 Trainingsthread für das allererste Super Mario Bros. Level. Insgesamt wurden also 17500 Episoden pro Netzwerk trainiert. Auf den X-Achsen ist die verstrichene Zeit aufgeführt, auf der Y-Achse der für das Diagramm spezifisch erreichte Werte. Die Graphen sind zur besseren Verständlichkeit mit einem Smoothingfaktor von 85% dargestellt.

7.1 Direkter Vergleich von Netzwerken

Nachfolgend werden die Ergebnisse der jeweiligen Netzwerke kurz aufgeführt.

7.1.1 DCN + GRU.

Das *Deep(ish)-Convolutional-Netzwerk* in Kombination mit der *GRU-Zelle* erwies sich als am schnellsten lernende Kombination. Bereits nach ca. 27 Minuten wurde das Ziel erstmalig erreicht; das Lernziel war hier anschließend nach ca. 34 Minuten erreicht.

Im weiteren Verlauf verließ dieses Netzwerk allerdings zweimal den zielführenden Weg für ca. 3 Minuten und Aufgrund des Zeitpunkt nicht näher definierbaren Zeitrahmen, innerhalb des ca. einstündigen Trainingszeitraum.

Auffällig für dieses Netzwerk ist die frühe stabile starke Lernperformance mit einem Vorsprung von bis zu ca. 5 Minuten vor den anderen Kombinationen.

7.1.2 CNN + GRU.

Das einfache *CNN* in Kombination mit der*GRU-Zelle* erwies sich hingegen als langsam lernende Kombination. Auch wenn nach 29 Minuten das Ziel erstmalig erreicht wurde, wurde das Lernziel erst nach ca. 35 Minuten erreicht.

Im weiteren Verlauf verließ dieses Netzwerk allerdings dreimal den zielführenden Weg für insgesamt 7 Minuten.

Auffällig für dieses Netzwerk ist die bereits früh auftauchende Instabilität.

7.1.3 CNN + LSTM.

Als äußerst langsam lernendes Netzwerk erwies sich die Kombination aus naiven *CNN* mit *LSTM-Zelle*. Auch wenn nach 27 Minuten das Ziel erstmalig erreicht werden konnte, wurde das Lernziel erst nach ca. 36 Minuten erreicht.

Im weiteren Verlauf erwies es sich zudem als tendenziell instabil und wich dreimal von zielführenden Weg ab für insgesamt ca. 5 Minuten.

7.1.4 DCN + LSTM.

Die Kombination aus Deep(ish)-Convolutional-Netzwerk mit *LSTM-Zelle* erwies sich als langsamstes lernendes Netzwerk. Nach ca. 33 Minuten wurde das Ziel erstmalig erreicht, nach 39 Minuten wurde das Lernziel erreicht.

Im weiteren Verlauf verließ dieses Netzwerk kein einziges Mal den zielführenden Weg.

Auffällig für diese Kombination ist die für den Trainingszeitraum perfekte Stabilität und die ca. 58 Minuten auftrettende wachsende Führung des insgesamt erreichten Rewardes.

Abbildung 11: Legende für die Ergebnisse

7.2 Zwischenfazit

Ausgehend von vorherigen Test in Kombinationen mit weiteren kleineren Experimenten lässt sich feststellen, dass die Lernstabiltät eine übergeordnete Rolle zum lösen des SMB-Problemes aufweist. Denn testet man schnelle Netzwerke wie die Kombination von *DCN* und *GRU-Zelle* so lässt sich feststellen, dass nicht immer das Ziel erreicht werden kann, ähnlich gut bzw. schlecht verhalten sich die anderen beiden Netzwerkarchitekturen. Hierfür ist u.a. die Struktur der Level, bestehend aus Sackgassen, Engstellen, Fallen, Sprungpassagen oder ähnliche in Kombination mit dem Rewardshaping verantwortlich.

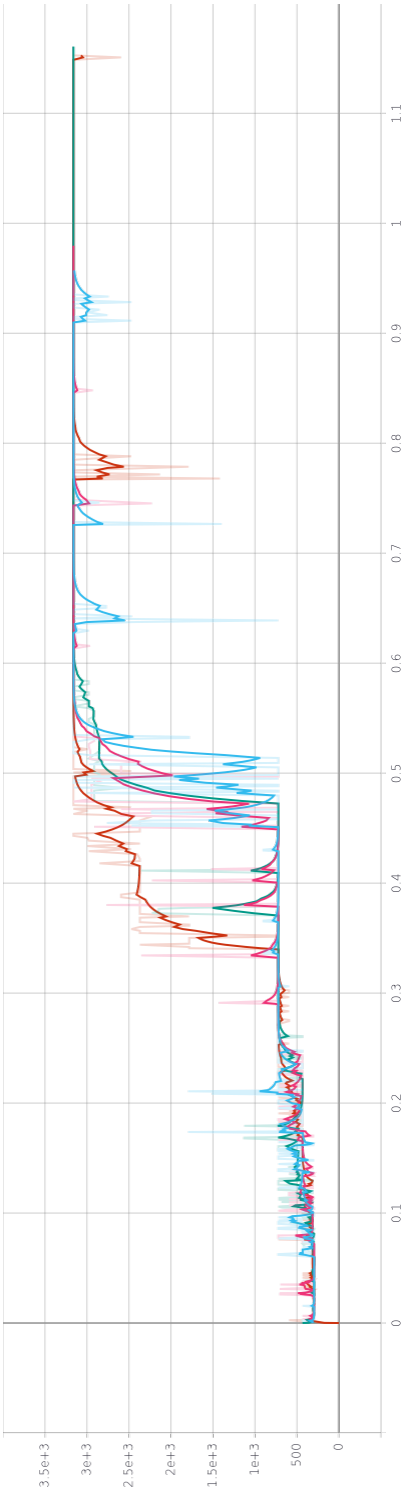


Abbildung 12: Ergebnis: Erreichte X-Position

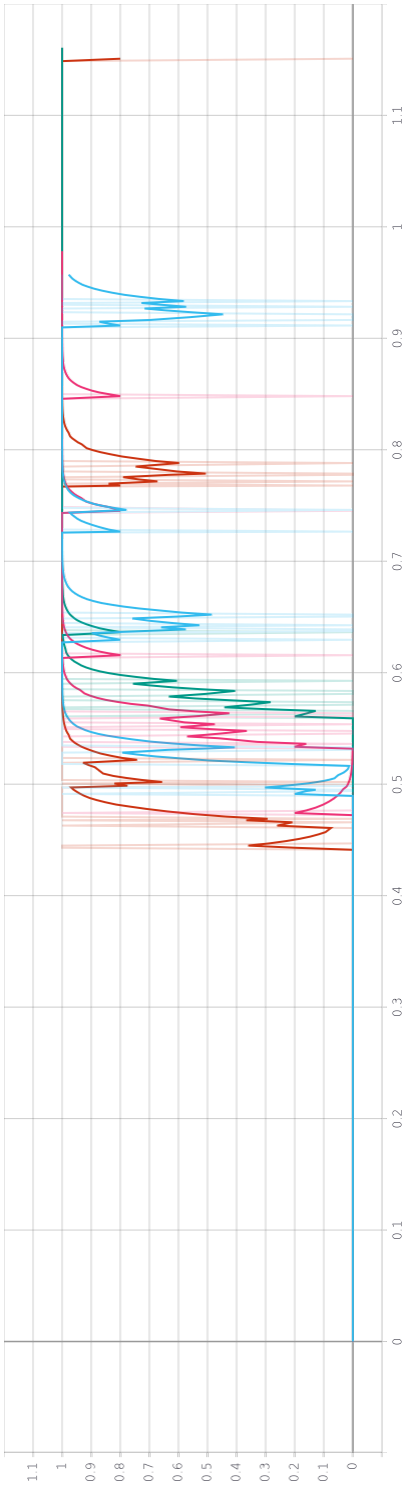


Abbildung 13: Ergebnis: Erreichte Flagge

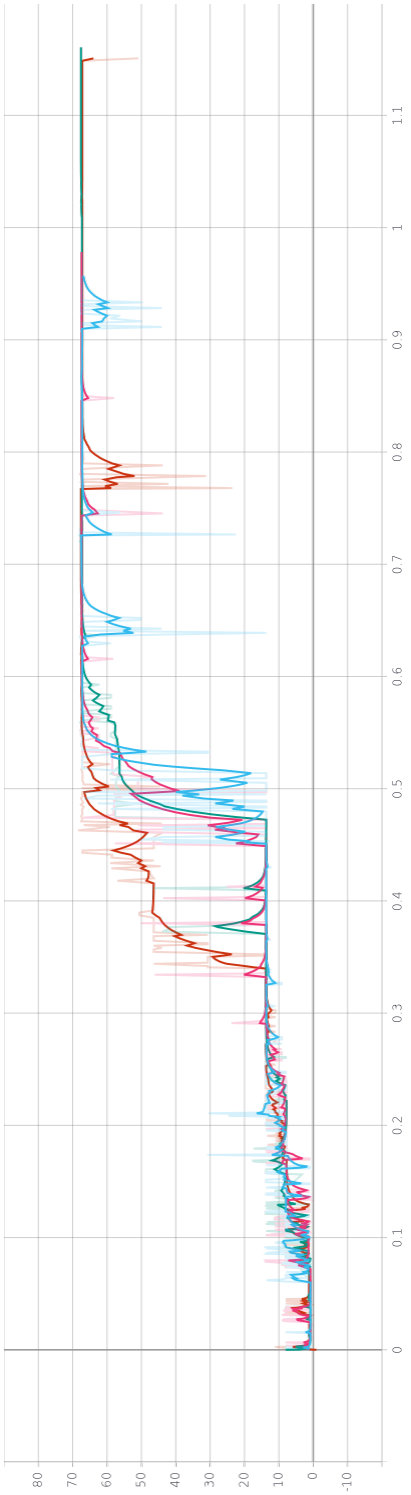


Abbildung 14: Ergebnis: Insgesamt erreichter Reward

Vergleicht man die Aufnahmen der Trainingsdurchgänge (zu finden im Projektrepository [10]) so fällt auf, dass es auch nach dem Training für alle Netzwerkkombination noch Verbesserungspotential innerhalb von wenigen Millisekunden besteht, geschuldet dadurch das Mario an einzelnen Hindernissen kurz zum stoppen kommt.

Abschließend wurden weitere Test mit dem erfolgversprechendem Netzwerk, bestehend aus *Deep(ish)-Convolutional-Netzwerk* und *LSTM-Zelle*, durchgeführt um weitere Aussagen hinsichtlich der Lernperformance zu treffen bzw. zu bestätigen.

7.3 Weiterführende Beobachtungen

Nachfolgend werden die restlichen drei Level der ersten SMB-Welt mit *DCN* und *LSTM-Zellen* aufgeführt.

7.3.1 Level 2.

Beim zweiten SMB-Leve erreichte keines der Netzwerkkombinationen aus dem ersten Experiment (vgl. 7.1) das Ziel. Alle blieben an der selben Stelle hängen: Einer Sackgasse, wo der eigentliche Lösungsweg durch mehrere Gegner für eine geringe Zeit blockiert wird.

Ursächlich hierfür ist das *Rewardshaping*, welches den Lösungsweg nicht mit längeren Wartezeiten definiert. Es zeigte sich, dass mit einem weitestgehend randomisierten Reward diese Stelle über einen entsprechend längeren Zeitraum von den meisten Kandidaten lösbar war.

7.3.2 Level 3.

Nachfolgend werden die Ergebnisse aus Abbildung 15 kurz analysiert. Es wurden 5 Threads mit jeweils 2500 Episoden trainiert.

Level 3 besteht hierbei im wesentlich aus Sprungpassagen unterschiedlicher Schwierigkeitsgrade, welche dem ersten Level tendenziell ähnelt, wodurch sich die Ähnlichkeit der Lernperformance erklären lässt.

Nach ca. 31 Minuten wurde erstmalig das Ziel erreicht, nach weiteren 3 Minuten war das Lernziel erreicht. Ähnlich wie in dem vorherigen Experiment fällt die Lernstabilität auf. Es ist auch anzumerken, dass hier ebenfalls nach Erreichen des Lernzieles ein weiterer Lernerfolg messbar ist in Form des insgesamt verdienten Rewardes.

7.3.3 Level 4.

Nachfolgend werden die Ergebnisse aus Abbildung 16 kurz analysiert. Es wurden 5 Threads mit jeweils 7500 Episoden trainiert.

Das Level 4 beinhaltet das erste Auftreten des *Bowser* Gegners und besteht aus einigen sehr schwierigen Passagen wie bspw. schwierigen Sprungpassagen, tödlichen Sackgassen, feindliche Projektile und sich bewegende Hindernisse.

Nach ca. 54 Minuten wurde das Ziel erstmalige erreicht, nach ca. 72 Minuten stellte sich das Lernziel ein, unterbrochen durch einen ca. 5 Minütigen Misserfolg.

Auffällig ist hierbei die relativ instabile Lernrate zu Beginn des Trainings, welche auf die Schwierigkeit des Levels zurückzuführen ist. Auffällig ist auch die minimal instabile Rewardrate nach dem Erreichen des Lernzieles.

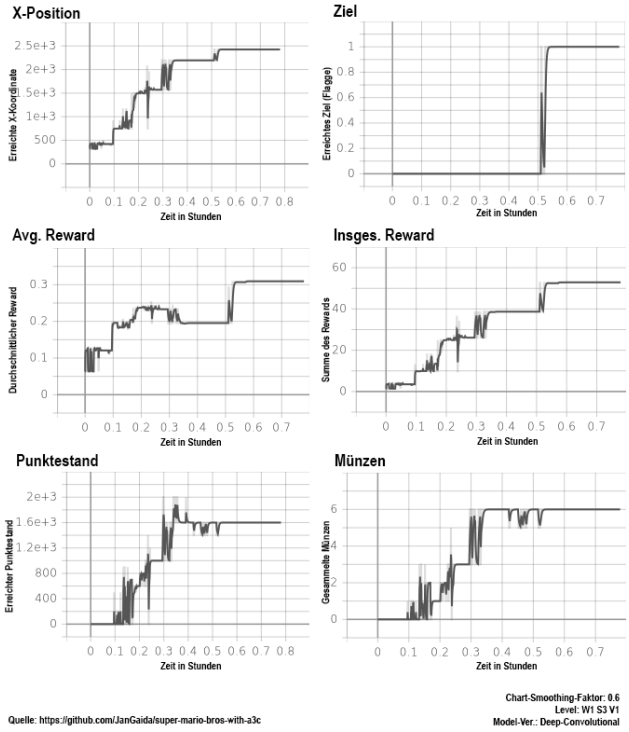


Abbildung 15: Ergebnis: Welt 1 Level 3

8 RESÜMEE

Fasst man die Ergebnisse dieser Studienarbeit also zusammen, lässt sich zunächst erst einmal feststellen, dass SMB für viele RL-Algorithmen eine Herausforderung darstellt. Es lässt sich zwar vermuten, dass mit leistungsstärkerer Hardware noch schneller das Lernziel erreicht werden kann, dennoch wird aufgrund der tendenziell geringen Explorationsrate bei den getesteten RL-Algorithmen nicht der bestmöglichste Lösungwegs zeitnah gefunden.

Für die Suche nach diesen bestmöglichen Lösungweg erwies sich die Stabilität des Lernerfolgs als äußerst wichtig, auch wenn diese dazu führt, dass das eigentliche Lernziel später erreicht wird. Desweiteren lässt sich feststellen, dass diese Stabilität des Lernerfolgs auch dazuführen kann, dass vorallem schwierige Passagen schneller gelöst werden kann.

Hinsichtlich der verwendeten neuronalen Schichten erwies sich die GRU-Zelle als schnell lernfähig, wohingegen die LSTM-Zelle sich als stabil erwies. Hinsichtlich des CNN erwies sich eine in Branchen unterteilte Variante performanter als eine Serielle Variante.

Auch wenn für ein aussagekräftiges Studienergebnis weitere Tests notwendig sind, lassen doch wichtige Charakteristika der verwendetet neuronalen Netzwerkarchitekturen feststellen, welche wiederum aktuelle Forschungsgebiete streifen.

LITERATUR

[1] [n.d.]. Tensorboard. https://www.tensorflow.org/tensorboard [Online; accessed 16-July-2020].

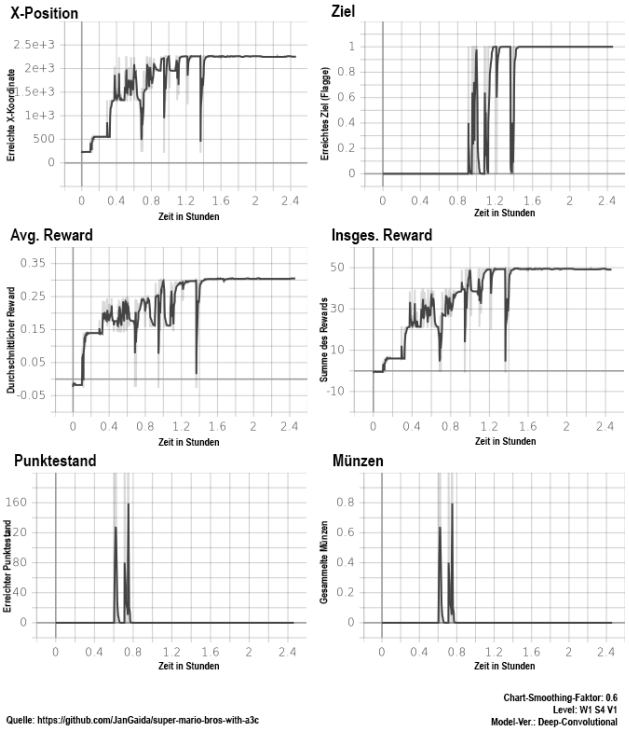


Abbildung 16: Ergebnis: Welt 1 Level 4

[2] 1998. Digital Millenium Copyright Act. https://web.archive.org/web/20000824110859/http://www.eff.org/IP/DMCA/hr2281_dmca_law_19981020_pl105-304.html [Online; accessed 16-July-2020].

[3] 2002. Torch. http://torch.ch/ [Online; accessed 16-July-2020].

[4] 2016. PyTorch. https://pytorch.org/ [Online; accessed 16-July-2020].

[5] Namhyuk Ahn. 2019. torchsummaryX. https://github.com/nmhkahn/torchsummaryX

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *ArXiv* abs/1606.01540 (2016).

[7] Wikipedia contributors. 2020. Wikipedia, The Free Encyclopedia. https://de.wikipedia.org/wiki/Super_Mario_Bros. [Online; accessed 16-July-2020].

[8] DeepMind. 2020. AlphaGo the story so far. https://deepmind.com/research/case-studies/alphago-the-story-so-far

[9] Florian Fallenbüchel. [n.d.]. Anwendungen von Reinforcement Learning. https://hci.iwr.uni-heidelberg.de/system/files/private/downloads/643877180/fallenbuechel_anwendungen-reinforcement-learning-report.pdf

[10] Jan Gaida. 2020. Deep-Q-Learning mit 'Super Mario Bros' und A3C. https://github.com/JanGaida/super-mario-bros-with-a3c

[11] Google. 2020. Why TensorFlow. https://www.tensorflow.org/about

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[13] Arthur Juliani. [n.d.]. Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C). https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2 [Online; accessed 16-July-2020].

[14] Abid K. 2013. Introduction to OpenCV-Python Tutorials. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#intro

[15] Christian Kauten. 2018. Super Mario Bros for OpenAI Gym. GitHub. https://github.com/Kautenja/gym-super-mario-bros

[16] Volodymyr Mnih. [n.d.]. Deep Q-Networks. https://drive.google.com/file/d/0BxXI_RttTZAhVUhpbDhiSUFFNjg/view [Online; accessed 16-July-2020].

[17] Parsa Heidary Moghadam. 2019. Deep Reinforcement learning: DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay. https://medium.com/@parsa_h_m/deep-reinforcement-learning-dqn-double-dqn-dueling-dqn-noisy-dqn-and-dqn-with-prioritized-551f621a9823

[18] Andrew Ng. 2020. What is Machine Learning? https://www.coursera.org/lecture/machine-learning/what-is-machine-learning-Ujm7v

[19] Chi Nhan Nguyen. 2019. Neuronale Netzwerke mit PyTorch entwickeln, trainieren und deployen. https://entwickler.de/online/machine-learning/neuronale-netzwerke-pytorch-579898880.html

[20] Viet Nguyen. 2019. [PYTORCH] Asynchronous Advantage Actor-Critic (A3C) for playing Super Mario Bros. https://github.com/uvipen/Super-mario-bros-A3C-pytorch

[21] Chris Nicholls. 2017. Reinforcement learning with the A3C algorithm. https://cgnicholls.github.io/reinforcement-learning/2017/03/27/a3c.html

[22] OpenAI. 2019. OpenAI Five. https://openai.com/projects/five/

[23] RB1196. [n.d.]. Deep Q-Networks. https://mario.fandom.com/wiki/World_1-1_(Super_Mario_Bros.)?file=SMB_World_1-1_NES_1.png [Online; accessed 16-July-2020].

[24] Chris Yoon. 2019. Double Deep Q Networks, Noisy DQN and DQN with Prioritized Experience Replay. https://towardsdatascience.com/double-deep-q-networks-905dd8325412

Abbildungsverzeichnis

Proximal Policy Optimization am Beispiel CarRacing

Hermann Mark
hermann.mark@hof-university.de
Hochschule für Angewandte Wissenschaften Hof
Hof

ZUSAMMENFASSUNG

Im Rahmen dieser Studienarbeit sollte ein eigen gewähltes Thema im Bereich des maschinellen Lernens ausgearbeitet werden. Im nächsten Abschnitt dieser Arbeit, wird zuerst die notwendige Theorie erklärt, um die Funktionsweise des verwendeten Algorithmus zu verstehen. Im Anschluss wird darauf eingegangen, welche Modifikationen am Algorithmus vorgenommen wurden, mit dem Ziel, das Training zu verbessern und die Trainingsdauer zu reduzieren. Hierbei wird anhand von Statistiken und Erklärungen über die gesammelte Erfahrung berichtet. Zuletzt gibt es noch einen Ausblick über weitere mögliche Aufgaben und Versuche, die auf Basis dieses Projekts durchgeführt werden können.

1 EINLEITUNG

Beim maschinellen Lernen handelt es sich um einen Teilbereich der künstlichen Intelligenz. Hierbei soll ein neuronales Netz trainiert werden, ein bestimmtes Problem zu lösen, wie zum Beispiel das Erlernen von Atari-Spielen oder auch das Fahren eines Autos im öffentlichen Verkehr von einem Startpunkt zu einem bestimmten Ziel. Ein Teilbereich des maschinellen Lernens ist wiederum das bestärkende Lernen. Hier muss sich ein Agent durch das Erkunden einer Umgebung die Problemlösung selbst erarbeiten. Damit der Agent aber auch erlernen kann, welche Aktionen gut oder schlecht waren, gibt es ein Belohnungssystem. Je nach Umgebung können Belohnungen und auch Strafen eigenständig vergeben werden, damit kontrolliert werden kann, was der Agent erlernen soll. Während des Trainings versucht der Agent die finale Belohnung zu maximieren. Somit kann der Agent durch vielfaches Ausprobieren erlernen, welche Aktionen für die Lösung des Problems nötig sind. Dazu gibt es viele verschiedene Algorithmen mit verschiedenen Ansätzen. Ein aktuell häufig eingesetzter Algorithmus ist der Proximal Policy Optimization Algorithmus. Dieser liefert besonders bei kontinuierlichen Steuerungsaufgaben gute Ergebnisse.

Im Rahmen dieser Studienarbeit wird deshalb der Proximal Policy Optimization Algorithmus, sowie das verwendete neuronale Netz, erklärt und versucht auf die Umgebung CarRacing zu optimieren.

2 THEORIE

2.1 Proximal Policy Optimization

Der Proximal Policy Optimization (PPO) Algorithmus [9] wurde am 20. Juli 2017 von John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford und Oleg Klimov von OpenAI veröffentlicht. PPO ist eine Policy-Gradienten-Methode und kann daher online lernen. Das bedeutet, dass direkt von den Erfahrungen des Agenten in der Umgebung gelernt wird. Eine Art Wiedergabepuffer, in dem die Erfahrungen gesammelt werden, wird also nicht benötigt. Außerdem wird ein Stapel an gesammelten Erfahrungen, sobald er für ein Gradient Update verwendet wurde, sofort wieder verworfen und die Policy wird fortgesetzt. Die Basis für PPO bildet die Trust Region Policy Optimization (TRPO) und bietet zudem noch einige Vorteile. Er ist einfacher zu implementieren, stichprobeneffizient und robust, das heißt, dass er nicht nur eine spezielle Aufgabe, sondern eine Vielzahl von Aufgaben ohne Hyperparameter-Tuning lösen kann. Ein Algorithmus ist stichprobeneffizient, wenn er aus jeder Stichprobe das Beste herausholen kann.

Folgende Formel definiert den Policy Gradient Loss:

$$L^{PG}(\theta) = \hat{E}_t \left[log \pi_\theta (a_t|s_t) \hat{A}_t \right] \quad (1)$$

$\hat{E}_t$ drückt eine Erwartung über zwei Terme aus. Beim ersten Term π_θ handelt es sich um die Policy, die entsprechend der beobachteten Zustände Aktionen liefert, die durchgeführt werden sollen. Der zweite Term ist die Vorteilsfunktion $\hat{A}_t$, die eine Schätzung abgibt, die gut oder schlecht die ausgewählte Aktion im aktuellen Zustand ist (vgl. [9]).

Diese Vorteilsfunktion ergibt sich folgendermaßen:

$$\hat{A}_t = DiscountedRewards - ValueFunction \quad (2)$$

Bei den Discounted Rewards handelt es sich um die gewichtete Summe aller Belohnungen und dem Discount-Faktor Gamma, der üblicherweise im Bereich von 0,9 bis 0,99 liegt. Jener Faktor sagt aus, dass der Agent eine Belohnung, die er schnell erhalten kann, gegenüber einer Belohnung, die er erst in 1000 Zeitschritten erhalten kann, bevorzugt und dementsprechend handelt. Die Value Function hingegen versucht lediglich den finalen Discounted Reward in dieser Episode ab dem aktuellen Stand zu erraten. Aus der folgenden Abbildung, die den Ablauf des PPO Algorithmus zeigt, kann entnommen werden, dass der vermutete Vorteil $\hat{A}_t$ erst dann berechnet wird, nachdem die Policy in der Umgebung ausgeführt wurde. Somit sind die Daten, die zur Berechnung der Discounted Rewards benötigt werden, tatsächlich vorhanden und müssen nicht geschätzt werden.

Algorithm 1 PPO, Actor-Critic Style

for iteration=1, 2, ... **do**
  for actor=1, 2, ..., N **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

Abbildung 1: Ablauf des PPO Algorithmus mit einem Actor-Critic Netzwerk [9]

Berechnet man den vermuteten Vorteil $\hat{A}_t$, so kann dieser positiv oder negativ sein. Wenn dieser positiv ist, dann ist auch der Gradient positiv und die Wahrscheinlichkeit solcher Aktionen wird erhöht.

Ist er negativ, so ist der Gradient negativ und die Wahrscheinlichkeit solcher Aktionen wird verringert.

Beim Durchführen eines Gradientenabstiegs kann es aber auch zu Problemen kommen. Sollte zum Beispiel ein Gradientenabstieg immer nur für einen Stapel gesammelter Erfahrungen gemacht werden, kann damit die eigene Policy zerstört werden, da man die Parameter im Netzwerk zu weit außerhalb des Bereichs, in dem diese Daten erfasst wurden, aktualisiert. Deshalb muss sichergestellt sein, dass man sich beim Aktualisieren der Policy niemals zu weit von der alten Policy entfernt. Im TRPO Algorithmus gibt es dafür eine sogenannte KL-Einschränkung. Allerdings belastet diese KL-Einschränkung den Optimierungsprozess. Im PPO Algorithmus wird dieses Problem so gelöst, indem diese zusätzliche Einschränkung direkt mit in die Optimierungsfunktion aufgenommen wird (vgl. [10]).

Hierbei handelt es sich um die Clip-Funktion. Die Formel dafür sieht folgendermaßen aus:

$$L^{CLIP}(\theta) = \hat{E}_t \left[min\left(r_t(\theta)\hat{A}_t, clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \quad (3)$$

Innerhalb des Erwartungsoperators wird lediglich das Minimum zweier Terme zurückgegeben. Beim ersten Term handelt es sich um das $r_t(\theta)$-fache der Vorteilsschätzung $\hat{A}_t$. Während die Vorteilsschätzung $\hat{A}_t$ (2) bereits bekannt ist, stellt die Variable $r_t(\theta)$ ein Wahrscheinlichkeitsverhältnis zwischen den neuen Policy Ausgaben und den Ausgaben der alten Policy dar. Ist $r_t(\theta)$ größer als 1, dann ist die Aktion jetzt wahrscheinlicher als vorher. Ist $r_t(\theta)$ dagegen größer 0, aber kleiner 1, dann ist die Aktion jetzt weniger wahrscheinlich als vorher. Betrachtet man nun den zweiten Term, fällt auf, dass dieser dem ersten Term sehr ähnlich ist, allerdings wird dieser mit dem Clip-Operator abgeschnitten. Zuletzt fehlt noch der Wert von Epsilon (ϵ), dieser beträgt normalerweise 0,2 oder 0,1 (vgl. [8]).

Folgende Abbildung veranschaulicht die Funktionsweise der Clip-Funktion:

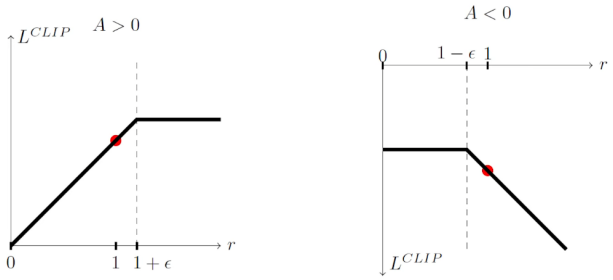


Abbildung 2: Funktionsweise der Clip-Funktion bei positiver bzw. negativer Vorteilsschätzung (roter Punkt: Startpunkt) [9]

Wie bereits in den vorherigen Abschnitten erwähnt, kann die Vorteilsschätzung sowohl positiv als auch negativ sein. Je nach Vorzeichen ändert dies auch die Wirkung des Min-Operators (3). Beim linken Diagramm ist die Vorteilsfunktion positiv, das heißt, dass die ausgewählte Aktion besser als erwartet war. Wenn r hier zu hoch wird, dann soll das Aktionsupdate nicht zu stark übertrieben werden. Deshalb wird die Funktion hier abgeschnitten, um den Effekt des Updates zu begrenzen. Beim rechten Diagramm ist die Vorteilsfunktion negativ, das bedeutet, dass die ausgewählte Aktion schlechter als erwartet war. Hier flacht die Funktion ab, wenn r gegen Null geht. Um jedoch zu verhindern, dass sich die neue Policy zu weit von der alten Policy entfernt, muss auch hier das Aktionsupdate begrenzt werden. Andernfalls würde die Aktionswahrscheinlichkeit auf Null reduziert werden (vgl. [10]).

Die endgültige Verlustfunktion, die zum Trainieren eines Agenten verwendet wird, sieht folgendermaßen aus:

$$L_t^{PPO}(\theta) = \hat{E}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \quad (4)$$

Als Resultat erhält man eine Funktion, die sich aus dem bereits bekannten Clip und zwei weiteren Termen zusammensetzt. Ersterer der beiden Terme aktualisiert das Netzwerk. Es wird außerdem eine Bewertung abgegeben, die angibt, wie gut es ist in einem gewissen Zustand zu sein, indem die finale Belohnung berechnet wird, die ab diesem Zeitpunkt erwartet wird. Beim letzten Term der Verlustfunktion handelt es sich um einen sogenannten Entropieterm. Damit der Agent zu Beginn des Trainings überhaupt in der Lage ist etwas zu lernen und herauszufinden, welche Aktionen eine bessere Belohnung erzielen, muss der Agent zufällig Aktionen durchführen. Für diese Erkundungen ist der Entropieterm zuständig. Im weiteren Verlauf des Trainings hat dieser Term immer weniger Einfluss, da der Agent nicht mehr so stark wie zu Beginn auf zufällige Aktionen angewiesen ist, um etwas zu lernen. Die beiden Parameter c_1 und c_2 sind Hyperparameter und gewichten die beiden Terme (vgl. [10]).

3 UMSETZUNG

3.1 Allgemeines

Zur Demonstration des PPO-Algorithmus wurde CarRacing [1], eine Box2D-Umgebung von Gym OpenAI [2], verwendet. Dabei handelt es sich um ein Rennen aus der Vogelperspektive. Der Agent trainiert standardmäßig so lange, bis er einen durchschnittlichen Reward von 900 erreicht hat. Diese Grenze kann allerdings auch beliebig verändert werden. Die folgende Abbildung zeigt die Oberfläche des Spiels.

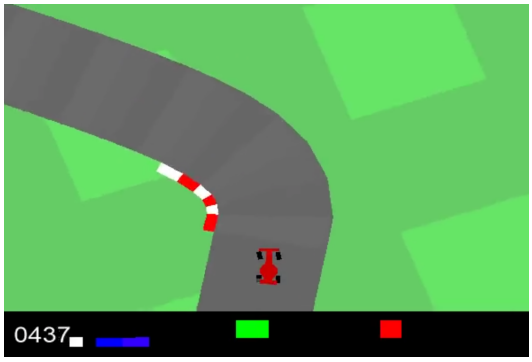


Abbildung 3: CarRacing, eine Box2D-Umgebung von Gym OpenAI

Am unteren Rand des Fensters sind verschiedene Anzeigen, die folgendes von links nach rechts angeben: Punktzahl, Geschwindigkeit, vier ABS-Sensoren, Lenkradposition, Gyroskop (Bestimmung der Lage).

3.2 Programm

Als Grundlage für die Umsetzung diente folgendes Projekt: Car Racing with PyTorch [7]. Folgende Voraussetzungen werden dafür benötigt:

- pytorch 0.4 [3]
- gym 0.10 [5]
- visdom 0.1.8 [4]

Visdom ist ein flexibles Tool zur Visualisierung und Organisierung von Live-Daten.

Um auch Videos über einzelne Trainingsdurchläufe zur Verfügung zu haben, wurde der Code um diese und noch einige andere Funktionen erweitert und umfasst nun drei Dateien mit folgenden Funktionen:

- **test.py**: Dient zum Testen eines trainierten Agenten. Dabei werden einzelne Durchläufe gerendert und im Ordner "videos fully trained" abgespeichert, sowie weitere Meta-Daten und Statistiken.
- **train.py**: Dient zum Trainieren des Agenten. Während des Trainings werden einzelne Durchläufe gerendert und im Ordner "videos" abgespeichert, sowie weitere Meta-Daten und Statistiken. Der trainierte Agent wird im Ordner "param" abgespeichert.
- **utils.py**: Beinhaltet Methoden zum Plotten des Trainingsverlaufs.

Um ein Training durchzuführen, muss zuerst der Visdom-Server gestartet werden, sofern der Trainingsverlauf visualisiert werden soll. Dies kann mit dem Befehl "python -m visdom.server" erledigt werden. Dieser ist dann unter "http://localhost:8097/" standardmäßig erreichbar. Mit dem Befehl "python train.py - -render - -vis" kann das Training dann gestartet werden. Der Parameter "- -vis" bewirkt dabei, dass der Trainingsverlauf visualisiert wird. Weitere mögliche Parameter sind in "train.py" zu finden. Um einen trainierten Agenten zu testen, wird der Befehl "python test.py - -render" verwendet.

3.3 Aufbau des Netzes

Es handelt sich bei dem verwendeten Netz um ein Actor-Critic Netzwerk. Als Input erhält das Netz ein aus 96x96 Pixeln bestehendes Frame. Um auch Informationen über die Geschwindigkeit des Wagens zu erhalten, werden vier benachbarte Frames übergeben. Es folgen sechs 2D Convolutional Layer und pro Layer wird eine ReLU zur Aktivierung verwendet. Diese sind in der folgenden Abbildung, die den Aufbau des Netzes zeigt, blau dargestellt.

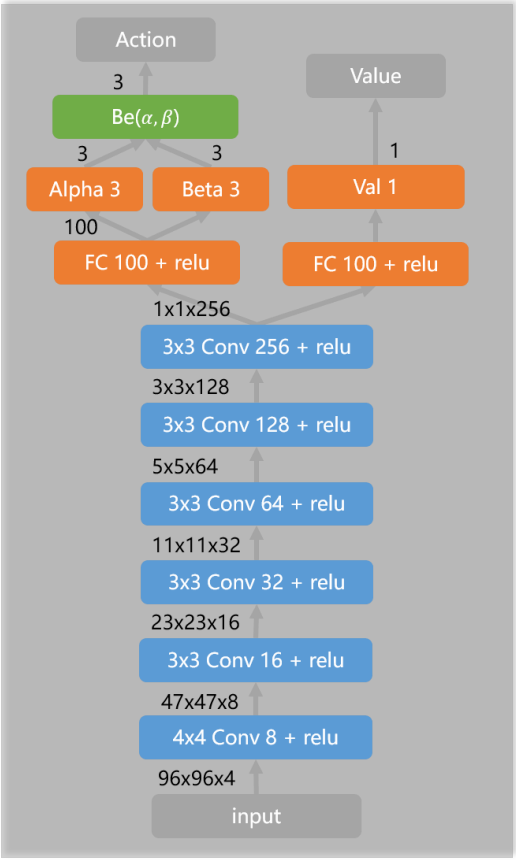


Abbildung 4: Aufbau des Actor-Critic Netzwerks [7]

Um den Actor bzw. den Critic-Teil darzustellen, wird eine Fully-Connected Layer mit zwei Köpfen verwendet. Der Actor gibt α und β für jede Aktion als Parameter einer Beta-Verteilung aus. Mögliche Aktionen sind:

- 1. Lenken: Wertebereich: [-1, 1]
- 2. Beschleunigen: Wertebereich: [0, 1]
- 3. Bremsen: Wertebereich: [0, 1]

Aufgabe der Value-Funktion (rechter oberer Teil in der Abbildung) ist es, eine Schätzung abzugeben, wie der finale Reward in dieser Episode ab diesem Zeitpunkt aussehen wird. Der Actor liefert dafür die entsprechende Aktion.

3.4 Erfahrungen

Es wurde im weiteren Verlauf dieses Projekts versucht, schneller und zuverlässiger ans Ziel zu gelangen. Zuerst wurde dabei die Verteilung des Rewards näher betrachtet. Dieser wird folgendermaßen vergeben:

- - 0,1 pro Frame
- + 1000 / N für jedes besuchte Bahnteil (N = Gesamtzahl der Bahnteile)

Wenn der Agent also beispielsweise in 732 Frames fertig ist, beträgt die Belohnung 1000 - 0,1 * 732 = 926,8 Punkte. Außerdem gibt

es eine Strafe von - 0,05 wenn der Wagen von der Bahn abkommt (pro Frame). Diese Strafe muss allerdings sehr klein sein, da sonst das Ergebnis, wie es zuvor anhand eines Beispiels berechnet wurde, verfälscht wird. Die Episode endet, wenn alle Teile besucht sind. Des Weiteren kann ein Reward vergeben werden, wenn der Agent stirbt. Es wurden deshalb drei Trainingsläufe durchgeführt, einmal mit einen Reward von +100, 0 und -100. Die folgenden drei Abbildungen zeigen die jeweiligen Trainingsergebnisse.

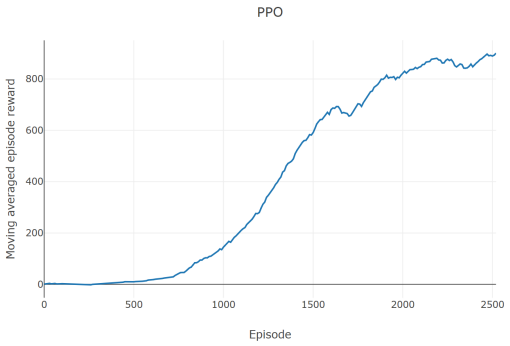


Abbildung 5: Reward beträgt +100, wenn der Agent stirbt

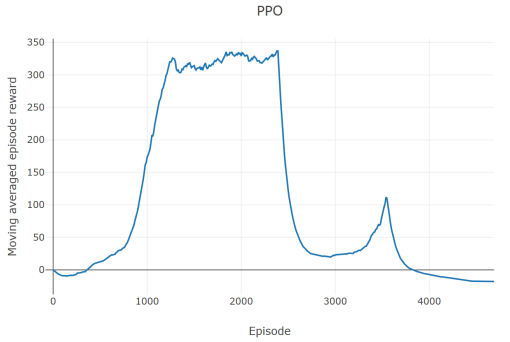


Abbildung 6: Reward beträgt 0, wenn der Agent stirbt

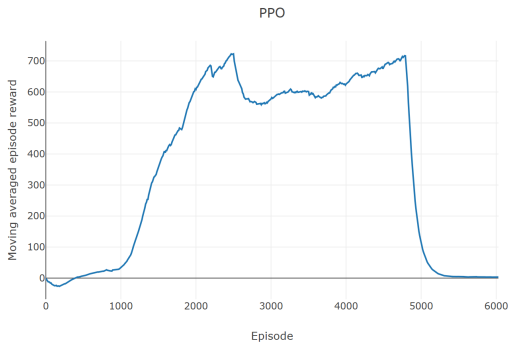


Abbildung 7: Reward beträgt -100, wenn der Agent stirbt

Interessant ist hierbei, dass der Agent, der keine Belohnung (Abb. 6) bekommt, wenn er stirbt, nicht einmal einen durchschnittlichen Reward von 400 schafft. Bei einer Bestrafung von -100 (Abb. 7) wird zumindest ein durchschnittlicher Reward von 700 erreicht. Der gewünschte durchschnittliche Reward von 900 wird nur bei einer Belohnung von +100 (Abb. 10) erreicht und diese wird somit auch beibehalten.

Als Nächstes wurden verschiedene Buffer- und Batch-Größen getestet. Der bisherige Wert für die Buffer-Größe war 2000 und für die Batch-Größe 128. Weil diese Werte bereits einen guten Trainingsverlauf haben, werden nur drei weitere Werte in diesem Bereich gewählt, und zwar:

- Buffer = 1000, Batch = 64 (orange)
- Buffer = 2000, Batch = 64 (grün)
- Buffer = 4000, Batch = 256 (blau)

Folgende Abbildung zeigt die entsprechenden Trainingsverläufe, wobei die rote Linie dem Training mit den bisherigen Werten entspricht.

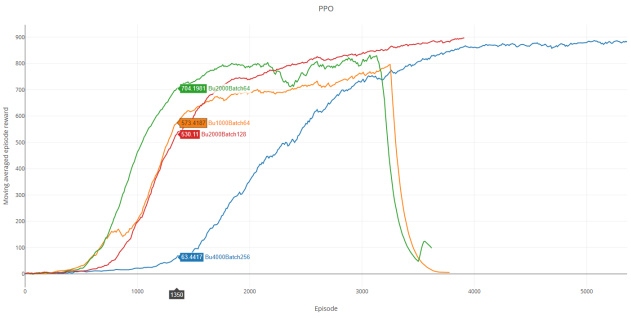


Abbildung 8: Trainingsverläufe mit verschiedenen Buffer- und Batch-Größen

Vergleicht man die verschiedenen Trainingsverläufe, fällt auf, dass man den Trainingsverlauf bei CarRacing in drei Phasen einteilen kann:

- **1. Phase:**
  Agent erreicht durchschnittlichen Reward von 0 – 100
- **2. Phase:**
  Agent erreicht durchschnittlichen Reward von 100 – 700
- **3. Phase:**
  Agent erreicht durchschnittlichen Reward von 700 – 900

In der ersten Phase lernt der Agent nur sehr wenig und macht kaum Fortschritte. In der zweiten Phase, die meistens so lange dauert wie die erste Phase, lernt der Agent dagegen sehr viel. Der schwierige Teil ist allerdings Phase 3. Diese dauert länger als die beiden Phasen davor und es kann passieren, dass der Agent keine Fortschritte mehr erzielt und sogar wieder alles verlernt. Dies ist auch bei zwei Durchläufen passiert (orange und grün). Obwohl Phase 1 und 2 bei “grün” kürzer als bei allen anderen Durchläufen gedauert hat, ist der Agent in Phase 3 gescheitert. Bei “blau” wurde auch nach über 5000 Episoden der durchschnittliche Reward von 900 nicht erreicht. Das Training wurde deshalb abgebrochen. Weil bei einer Buffer-Größe von 2000 und einer Batch-Größe von 128

der Trainingsverlauf am zuverlässigsten war und der durchschnittliche Reward von 900 auch erreicht wurde, wurden diese Werte beibehalten.

Im nächsten Schritt wurde ein Vergleich zwischen dem Adam-Optimierer und dem SGD-Optimierer (SGD = Stochastic Gradient Descent) durchgeführt. Beides sind stochastische Optimierer, wobei der Adam-Optimierer weiter verbreitet ist und häufiger zum Einsatz kommt. Bisher wurde der Adam-Optimierer verwendet. Beim SGD-Optimierer wurden auch wieder verschiedene Lernraten, Buffer- und Batch-Größen getestet und welchen Unterschied das Nesterov-Momentum macht. Auf das Nesterov-Momentum wird in dieser Studienarbeit nicht näher eingegangen. Die nächste Abbildung zeigt die Ergebnisse der verschiedenen Trainingsdurchläufe. Anhand der Farbe kann erkannt werden, welche Einstellungen beim Optimierer vorgenommen wurden:

- **Braun und Lila:**
  SGD (lr=0.001, momentum=0.9, nesterov=True), Buffer = 2000, Batch = 96
- **Rot:**
  SGD (lr=0.001, momentum=0.9, nesterov=False)
- **Grün:**
  SGD (lr=0.1, momentum=0.9, nesterov=True)
- **Blau und Orange:**
  SGD (lr=0.001, momentum=0.9, nesterov=True)

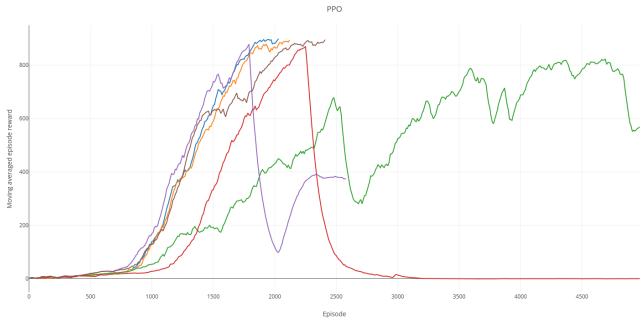


Abbildung 9: Trainingsverläufe mit SGD-Optimierer und verschiedenen Lernraten, Buffer- und Batch-Größen, sowie mit oder ohne Nesterov-Momentum

Am besten hatte der SGD-Optimierer mit einer Lernrate von 0,001, einem Momentum von 0,9 und mit dem Nesterov – Momentum (blaue und orange Linie) abgeschnitten. Es wurden noch weitere Trainingsdurchläufe durchgeführt, die aber zu keinem brauchbaren Ergebnis führten und deshalb verworfen wurden.

Folgende zwei Abbildungen zeigen das jeweils beste Ergebnis mit dem Adam-Optimierer bzw. dem SGD-Optimierer.

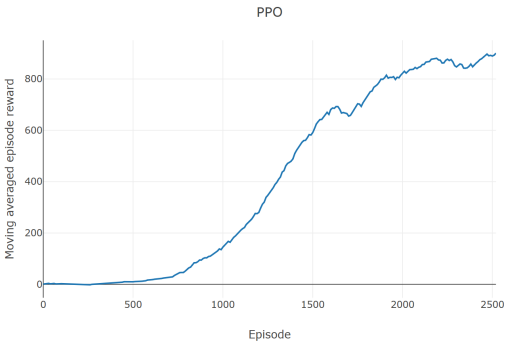


Abbildung 10: Bestes Ergebnis mit Adam-Optimierer

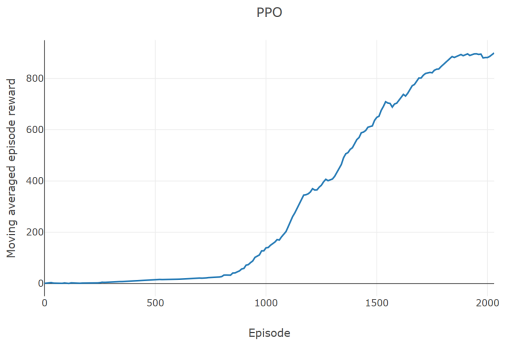


Abbildung 11: Bestes Ergebnis mit SGD-Optimierer

Beim Training mit dem SGD-Optimierer konnte der durchschnittliche Reward von 900 in nur 2031 Episoden erreicht werden. Mit dem Adam-Optimierer brauchte der Agent dafür 2520 Episoden. Außerdem war der Trainingsverlauf nicht so stabil und gleichmäßig wie mit dem SGD-Optimierer. Vergleicht man die Trainingszeiten der beiden Durchläufe, kann festgestellt werden, dass mit dem SGD-Optimierer rund 20% weniger Zeit benötigt wird als mit dem Adam-Optimierer, um einen durchschnittlichen Reward von 900 zu erreichen.

Um herauszufinden, ob mit den bisherigen Verbesserungen auch höhere Punktzahlen möglich sind, wurde versucht einen durchschnittlichen Reward von 910 zu erreichen. Die nächste Abbildung zeigt den Trainingsverlauf.

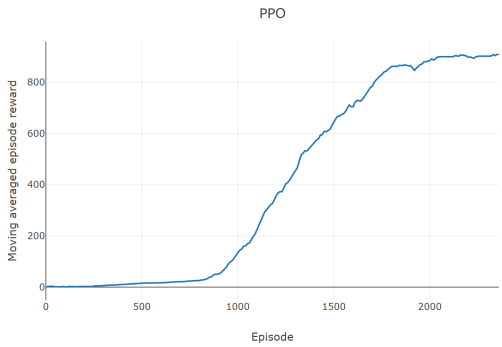


Abbildung 12: Trainingsverlauf eines Agenten, der einen durchschnittlichen Reward von 910 erreicht

Der Agent hat einen durchschnittlichen Reward von 910 nach 2361 Episoden erreicht. Noch höhere Werte könnten auch noch möglich sein, allerdings wurden dazu keine weiteren Tests durchgeführt.

Zuletzt wurde noch versucht, den optimierten PPO Algorithmus alleine auf der CPU zu parallelisieren. Hierbei wurde mit nur zwei Prozessen nach ungefähr 3600 Episoden (1800 Episoden pro Prozess) ein durchschnittlicher Reward von 900 erreicht. Damit konnte nochmal eine Verbesserung um rund 200 Episoden bewirkt werden. Weitere Trainingsdurchläufe mit mehr Prozessen waren aufgrund der zu geringen Rechenleistung nicht möglich. Allerdings sollte mit genügend Rechenleistung und mehr Prozessen ein noch besseres Ergebnis möglich sein.

4 AUSBLICK

Auf Basis dieser Arbeit können noch weitere Aufgaben und Versuche durchgeführt werden. Es können etwa noch weitere Optimierer getestet werden. Bisher wurde der Adam-Optimierer und der SGD-Optimierer verwendet. Beides sind stochastische Optimierer, allerdings wäre es interessant zu beobachten, wie gut die Performance anderer Typen von Optimierern ist.

Eine weitere Möglichkeit wäre, mit einem Scheduler die Lernrate während des Trainings zu beeinflussen. Zum Beispiel kann damit die Lernrate schrittweise oder zyklisch um den Faktor Gamma verändert werden. Außerdem kann die Anpassung auch nur nach einer bestimmten Anzahl an Episoden durchgeführt werden, beispielsweise einmal nach 100 und einmal nach 500 Episoden. Bei den bisherigen Trainingsdurchläufen hat der Agent in den ersten 800 Episoden kaum Fortschritte gemacht, mithilfe eines Schedulers kann versucht werden schneller Fortschritte zu erreichen.

Parallele Implementierungen eines Algorithmus werden beim Trainieren eines neuronalen Netzes gerne eingesetzt, um die Trainingsdauer zu reduzieren. Bisher wurde nur auf der CPU parallelisiert, allerdings kann auch die GPU mit einbezogen werden. Dabei werden auf der CPU parallel Daten in der Umgebung gesammelt und auf der GPU wird das Gradientenupdate durchgeführt.

Zuletzt kann die Performance des optimierten PPO Algoritmus mit der des World Models [6] verglichen werden. Das World Model war der erste Algorithmus, der einen sehr hohen durchschnittlichen Reward bei CarRacing erreicht hat. Da dieser auch heute noch mit aktuellen Verfahren des maschinellen Lernens mithalten kann, wäre ein Vergleich mit dem World Model sehr interessant. Außerdem kann getestet werden, mit welchem Verfahren ein höherer durchschnittlicher Reward erreicht werden kann.

5 LITERATUR

[1] *CarRacing-v0.* http://gym.openai.com/envs/CarRacing-v0/. Zugriff: 4.08.2020.
[2] *Gym OpenAI.* http://gym.openai.com/. Zugriff: 4.08.2020.
[3] *PyTorch.* https://pytorch.org/. Zugriff: 3.08.2020.
[4] *Visdom.* https://github.com/facebookresearch/visdom. Zugriff: 3.08.2020.
[5] Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang und Wojciech Zaremba: *OpenAI Gym.* https://github.com/openai/gym, 2016. Zugriff: 3.08.2020.
[6] Ha, David und Jürgen Schmidhuber: *World Models.* http://arxiv.org/abs/1803.10122, 2018. Zugriff: 4.08.2020.
[7] Ma, Xiaoteng: *Car Racing with PyTorch.* https://github.com/xtma/pytorch_car_caring, 2019. Zugriff: 3.08.2020.
[8] Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford und Oleg Klimov: *Proximal Policy Optimization.* https://openai.com/blog/openai-baselines-ppo/, 2017. Zugriff: 3.08.2020.
[9] Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford und Oleg Klimov: *Proximal Policy Optimization Algorithms.* https://arxiv.org/abs/1707.06347, 2017. Zugriff: 3.08.2020.
[10] Steenbrugge, Xander: *Policy Gradient methods and Proximal Policy Optimization (PPO): diving into Deep RL!* https://www.youtube.com/watch?v=5P7I-xPq8u8, 2018. Zugriff: 1.08.2020.

Reinforcement Learning am Beispiel des Atari Spiels Seaquest

Hannes Müller
hannes.mueller@hof-university.de
Hochschule Hof
Hof an der Saale, Germany

ZUSAMMENFASSUNG

In der Studienarbeit wurde sich intensiv mit dem Konzept von Deep Q-Learning Algorithmen auseinandergesetzt und auf Basis einer Vorlage ein eigenes Model implementiert. Es wurden anschließend Verbesserungen im Rahmen des bestehenden Netzes vorgenommen um Effizienz, Performance und Stabilität zu erzielen, wobei noch viele Verbesserungsmöglichkeiten existieren. Einige davon wurden zum Schluss grob skizziert, um einen Ausblick auf die Möglichkeiten zu geben.

1 EINLEITUNG

In dieser Studienarbeit soll beschrieben werden, wie mithilfe von bestätigendem Lernen das Atari Spiel Seaquest gespielt werden kann bzw. dort ein möglichst hoher Punktestand erzielt wird. Im ersten Abschnitt sollen grundlegende Themen wie das bestätigende Lernen oder Deep Q-Learning behandelt werden, um einen theoretischen Einstieg in das Thema maschinelles Lernen zu bieten. Der zweite Abschnitt befasst sich mit dem konkreten Programm, welches implementiert wurde und soll die erreichten Verbesserungen des Algorithmus bzw. des Models aufzeigen. Zuletzt folgt noch ein kleiner Ausblick auf mögliche Verbesserungen des Algorithmus.

2 HINFÜHRUNG UND THEORETISCHER HINTERGRUND

2.1 Gym und Keras

Zur Implementierung der KI wurde das Deep-Learning Framework Keras und die Reinforcement Learning Library Gym zur Hilfe genommen.
Die Gym Library stellt verschiedene Umgebungen bereit mit denen sowohl klassische Problemstellungen der KI-Forschung, wie z.B. das Halten eines frei schwingenden Pendels in der Senkrechten, aber auch Roboter-Simulationen oder vollständige Atari-Spiele simuliert werden können. Die Wahl für diese Studienarbeit fiel auf das Atari-Spiel „Seaquest“, indem der Spieler ein U-Boot steuert, welches Taucher aus dem Wasser retten muss. Das U-Boot wird hierbei, allerdings von Haien und feindlichen U-Booten attackiert, deren Angriffe zum Verlust von einer der vier Leben führen. Punkte erhält der Spieler, indem er Gegner beschießt und Taucher an die Oberfläche bringt. Es existiert ebenso eine Sauerstoff-Anzeige, welche nicht auf null absinken darf. Um dies zu verhindern, muss der Spieler regelmäßig auftauchen, wobei er hier immer einen Taucher an Board haben muss, ansonsten verliert er ebenfalls ein Leben. Gym bietet eine einfache Verwendung der Schnittstellen für die ausgewählte Umgebung an. So können mit der Step-Funktion von der Umgebung die Werte Observation, Reward, Done und Info abgefragt werden, welche in der Seaquest Umgebung wie folgt zusammengesetzt sind. Der Wert Observation beinhaltet die Bilddaten in Form eines zweidimensionalen Arrays. Der Wert Reward beinhaltet die erzielte Belohnung in der Spielumgebung als Integer. Der Wert Done ist ein Boolscher Wert und sagt aus, ob eine Episode, bei Seaquest bedeutet dies konkret den Verlust von vier Leben, bereits beendet ist oder nicht. Der Wert Info gibt die Anzahl an Leben zurück, welcher der Spieler bzw. Agent in der jeweiligen Episode noch zur Verfügung hat.
Das Keras-Framework ermöglicht es auf einfache und effiziente Art und Weise neuronale Netze zu erstellen und diese zu verwenden. Das Framework baut auf dem KI-Framework TensorFlow auf und bietet für dieses ein High-Level Interface an um die Komplexität der Erstellung eines Models einzugrenzen und ist vor allem häufig für Einsteiger im Bereich des maschinellen Lernens eine willkommene Hilfe.

2.2 Bestätigendes Lernen

Die Ausgangssituation des bestätigenden Lernens bzw. des bestärkenden Lernens ist eine völlig andere wie die des überwachten Lernens. Im Falle des überwachten Lernens liegt ein konkretes Ergebnis vor, welches als falsch oder richtig interpretiert werden kann. Ein Beispiel hierfür ist die Klassifizierung von Bildern, welche vorher gelabelt wurden. Der Algorithmus kann hier sofort erkennen, ob er das Bild richtig eingeordnet hat oder ob seine Interpretation falsch war. Im Bereich des bestätigenden Lernens hingegen, liegt diese Rückmeldung nicht bzw. nur bedingt vor. Die KI kann zum Zeitpunkt der Ausführung der Aktion noch nicht bewerten, ob dies eine gute oder schlechte Entscheidung war. Aufgrund dieses Umstands muss im bestätigenden Lernen ein anderer Ansatz gewählt werden. Analog zum Trainieren eines Hundes gibt es bei einer guten Aktion eine Belohnung, einen Reward. Bei einer schlechten Aktion hingegen, wird ein Negativsignal zurückgemeldet, welches dem Algorithmus dazu veranlassen soll, diese Aktion in diesem Kontext zukünftig zu vermeiden. Das Lernen basiert somit auf Interaktion und Feedback. Die Interaktion wird vom sogenannten Agenten gesteuert, wobei die Bewertung Sache der Umgebung ist, welche auch als Environment bezeichnet wird. Der Agent führt somit Aktion a

zum Zeitpunkt t aus, was den Eingang für die Umgebung darstellt. Diese verarbeitet die Folgen der Aktion und gibt dem Agenten den Zustand zum Zeitpunkt t und die Belohnung, für die ausgewählte Aktion a, zum Zeitpunkt t zurück. Der Agent erkennt nun durch die Belohnung, ob seine Aktion Sinn ergeben hat oder nicht und zudem wird die Wahl der Aktion des Agenten für den nächsten Zustand (st+1) beeinflusst. Formal betrachtet besteht Reinforcement Learning, demzufolge aus fünf wichtigen Komponenten, nämlich dem Agenten (Agent), der Umgebung (Environment), dem Zustand (State), der Aktion (Action), sowie der Belohnung (Reward).

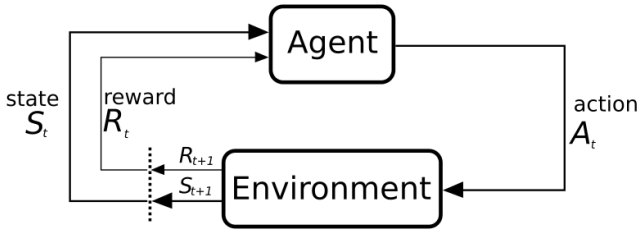


Abbildung 1: Agent und Umgebung [3]

Das bestätigende Lernen bzw. die verwendeten Algorithmen zur Bewertung von Aktionen unterteilen sich in modelbasiertes und modelfreies Lernen. In dieser Studienarbeit soll sich nur mit modelfreiem Lernen beschäftigt werden, da dies sonst den Rahmen sprengen würde. Ebenso wird das modellfreie Lernen nochmals in Q-Learning und in die Policy Optimierung aufgeteilt. Als Policy wird generell die Strategie des Agenten bezeichnet, um die Folgeaktion, auf Basis des aktuellen Zustands zu bestimmen. Bei Policy Optimierungsmethoden, wie bspw. Policy Gradients geht es darum, dass der Agent direkt die Policy Funktion erlernt, welche einen Zusammenhang zwischen Zuständen und Aktionen herstellt, vgl. [10]. Die Policy kommt ohne eine Value-Funktion aus, da hier neuronale Netze verwendet werden, um direkt die Aktionswahrscheinlichkeiten mit dem Netz abzubilden. Zu jedem Zeitpunkt, wenn der Agent mit der Umgebung interagiert, werden die Parameter des neuronalen Netzes optimiert, indem zukünftig eher „gute“ Aktionen ausgewählt werden. Ein großer Nachteil der Policy Optimierung ist allerdings, dass bei jedem Update des Netzes, die Schätzungen der Gradienten auf Basis der Daten <s,a ,r,s‘> einer einzelnen aufsummierten Runde basieren. Daher kann die Schätzung sehr verrauscht sein, sodass eine schlechte Gradientenschätzung die Stabilität des Lernalgorithmus sehr zum Nachteil beeinflussen kann. Aufgrund dieses Umstandes wurde Q-Learning präferiert, welcher eine bessere Probeneffizienz und eine stabilere Leistung aufweisen soll, vgl. [17].

2.3 Q-Learning und DQN

Q-Learning ist im Gegenteil zu Policy-Optimierungsverfahren ein „off-policy“ Algorithmus, welcher das Ziel hat, die beste Aktion zum gegebenen Zustand zu finden. Der Name „off-policy“ rührt daher, dass die Q-Learning Funktion von Aktionen außerhalb der Policy, wie z.B. Zufallsaktionen, lernt. Genauer gesagt, versucht Q-Learning eine Policy zu erlernen, welche den „Total Reward“, also die Summe der Punkte in einer Runde des Spiels, fortfolgend auch als Episoden bezeichnet, maximieren soll. Das „Q“ in Q-Learning steht für Qualität, wobei Qualität in diesem Zusammenhang bedeutet, wie nützlich die Aktion zum Erreichen einer hohen Belohnung war. Dieser Zusammenhang lässt sich durch die Q-Funktion Q(s,a) darstellen, welche den erwartenden Nutzen(Q) einer Aktion im Zustand beschreibt. Damit zu jedem Zustand die bestmögliche Aktion ausgewählt werden kann, wird eine Q-Table verwendet, in der die Q-Values als Werte eingetragen werden. Der maximale Q-Value in der Tabelle gibt somit an, welche Aktion in welchem Zustand zu wählen ist. Die Berechnung der Q-Values erfolgt mit der Bellman-Gleichung, welche als Inhalt hat, dass das Befolgen der bestmöglichen Aktion in Zustand s, dazu führt, dass insgesamt die bestmögliche Strategie gewählt wird. Nach jedem Durchlauf einer Spielrunde kann so die Q-Table mit den maximalen Q-Values aktualisiert werden, was auf längere Sicht dazu führt, dass die Tabelle immer zur bestmöglichen Aktion bzw. zum höchsten Q-Value führt. Diese Vorgehensweise wird auch als „Exploiting“, also Nutzung, bezeichnet, da vorhandene bzw. generierte Informationen genutzt werden. Das Gegenstück hierzu ist das „Exploring“, da nicht nur auf Grundlage der maximalen zukünftigen Belohnung entschieden wird, sondern ein gewisser Anteil an zufälligen Entscheidungen getroffen wird. Dieser Zufallsanteil, auch als Epsilon ϵ bezeichnet, wird Anfangs hoch angesetzt und im Laufe der Zeit immer weiter verringert, da am Anfang die Q-Table noch keine fundierte Grundlage zur Entscheidungsfindung darstellt. Dieses Herabsetzen von ϵ zur Trainingszeit, wird auch als ϵ Greedy Strategie bezeichnet und bringt den Agenten dazu sich immer mehr auf sein erlerntes Wissen zu stützen, anstatt auf Zufallsentscheidungen, vgl. [16].

Algorithm 12.1 Q-Learning als Pseudocode

procedure Q-LEARNING
  Wähle $0 \leq \gamma < 1$
  for all s, a **do** initialisiere $\hat{Q}(s, a)$
  end for
  loop
    Beobachte Zustand s
    Wähle eine Aktion a aus und führe diese durch
    Erhalte Belohnung r
    Beobachte neuen Zustand s'
    $\hat{Q}(s, a) = r + \gamma \cdot \max_{a'} \hat{Q}(s', a')$
  end loop
end procedure

Abbildung 2: Q-Learning Pseudocode [7]

Q-Learning auf Basis einer Q-Table bringt allerdings noch einen schmerzlichen Nachteil mit sich, da mit wachsender Anzahl von Zuständen bzw. Aktionen ein immenser Anstieg der Möglichkeiten einhergeht, welche durch den Explorationsmechanismus irgendwann nicht mehr mit einem reellen, praxistauglichen Zeitaufwand lösbar sind. Ein weiteres Problem beim klassischen Q-Learning ist, dass die Belohnung zeitlich sehr weit vom aktuellen Status- und Handlungsraum des Agenten entfernt liegen kann. Dadurch kann der Agent erst bei einer langen Explorationsphase weit in der Zukunft liegende Belohnungen dem tatsächlichen Zustand zuordnen,

vgl. [15]. Eine Lösung für den zeitlichen Versatz bildet die Diskontierung der Belohnungen, indem jede nachfolgende Belohnung mit einem Faktor γ^n multipliziert und dies auf den ursprünglichen Belohnungen aufsummiert wird. Angenommen wir befinden uns bei der Belohnung r0, so berechnet sich der weitere Verlauf dementsprechend:

$$r0 + \gamma r1 + \gamma^2 r2 + \gamma^3 r3, \text{ wobei für } \gamma \text{ gilt: } 0 < \gamma < 1$$

Durch diese Strategie wird erreicht, dass zurückliegende Belohnungen, welche zur Erreichung der Belohnung beigetragen haben, ebenfalls berücksichtigt werden. Wenn dies nun mit der bereits erwähnten Q-Wert-Funktion kombiniert wird, erhält man folgende Funktion

$$Q(s,a) = r + \gamma max'_a(Q(s',a'))$$

, welche es ermöglicht die beste Aktion für den aktuellen Zustand auszuwählen, vgl. [4]. Um nun auch die kombinatorische Komplexität zu bewältigen wird ein Deep Q-Learning Network verwendet (DQN) und deshalb die Q-Table gegen ein Neuronales Netz ausgetauscht, welches als Funktions-Approximator zum Erlernen der Q-Wert-Funktion fungieren soll. Das Neuronale Netz bekommt hier die Zustände als Input, anstatt der kompletten Zustands-Aktion-Tabelle. Dies stellt das Kernkonzept von DQN's dar und auch die Implementierung soll aufgrund dieser Basis erfolgen, um das anfangs vorgestellte Spiel umzusetzen, vgl. [6].

2.4 Dueling DQN

Für die letzte grundlegende Veränderung des Algorithmus bzw. des Models wird der Dueling DQN Algorithmus verwendet, welcher den Q-Value in zwei Bestandteile aufschlüsselt. Der erste Bestandteil ist der Wert des aktuellen Zustands (Value) und der zweite Bestandteil jener Vorteil (Advantage), der sich ergibt, wenn eine bestimmte Aktion in diesem Zustand ausgeführt wird, im Vergleich zu allen anderen Aktionen. Formell betrachtet bekommt man also den Nutzen einer Aktion, indem man den Wert eines Zustands vom Q-Value eines Zustands-Aktionspaars abzieht.

$$Q(s,a) = V(s) + A(s,a) \Leftrightarrow A(s,a) = Q(s,a) - V(s)$$

Die beiden Komponenten werden in jeweils separaten Schichten des neuronalen Netzes ermittelt und anschließend wieder zu einer zusammengeführt, um Q(s,a) abschätzen zu können.

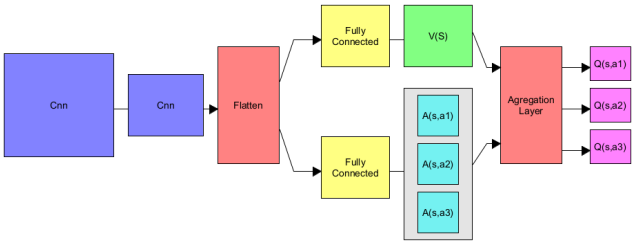


Abbildung 3: Architektur des Dueling DQN's

Die Entkopplung der Komponenten rührt daher, da so das Netz unabhängig lernen kann welchen tatsächlichen Wert die einzelnen Zustände haben, ohne auf die Folgeaktionen Rücksicht nehmen zu müssen. Ein Beispiel hierfür wäre ein einfaches Rennspiel, bei dem es darum geht, Hindernissen auszuweichen. Hier spielen die Aktionen bis kurz vor dem Zusammenstoß keine größere Rolle. Durch die Trennung der beiden Werte kann das Model nun lernen welche Zustände entscheidend zum Erzielen von Punkten sind. Unter Verwendung eines normalen DQN's muss der Wert für jede Aktion zu dem jeweiligen Zustand errechnet werden. Dies gilt auch wenn der aktuelle Zustand beim Ausführen, egal welcher Aktion, den Tod zur Folge hätte. Es wäre in dieser Situation nicht notwendig den Wert jeder einzelnen Aktion zu berechnen, da sowieso alle das gleiche Ergebnis hätten.
Die Architektur des Netzes sieht nach der Umstellung auf das Dueling DQN wie in Abbildung 4 aus, vgl. [13].

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 70, 80, 4)	0	
conv2d_1 (Conv2D)	(None, 66, 76, 128)	12928	input_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 33, 38, 128)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 31, 36, 64)	73792	max_pooling2d_1[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 15, 18, 64)	0	conv2d_2[0][0]
flatten_1 (Flatten)	(None, 17280)	0	max_pooling2d_2[0][0]
dense_3 (Dense)	(None, 64)	1105984	flatten_1[0][0]
dense_4 (Dense)	(None, 18)	1170	dense_3[0][0]
dense_1 (Dense)	(None, 64)	1105984	flatten_1[0][0]
lambda_1 (Lambda)	(None,)	0	dense_4[0][0]
dense_2 (Dense)	(None, 1)	65	dense_1[0][0]
subtract_1 (Subtract)	(None, 18)	0	dense_4[0][0] lambda_1[0][0]
add_1 (Add)	(None, 18)	0	dense_2[0][0] subtract_1[0][0]

Total params: 2,299,923
Trainable params: 2,299,923
Non-trainable params: 0

Abbildung 4: Architektur des Netzes nach Umstellung auf ein DDQN

3 MODEL UND UMSETZUNG

3.1 Allgemein

Als Ausgangsbasis für die Implementierung und das grundlegende Verständnis von DQN's diente ein Tutorial von Parsa Heidary Moghadam, welches den minimalen Aufbau eines DQN's beinhaltet

und in einem GitHub Repository [9] zu finden ist. Anschließend wurden Verbesserungen im Hinblick auf Performance und Effizienz des Netzes integriert und versucht die Reward-Funktion speziell auf die Problematik des gewählten Spiels anzupassen, vgl. [8].In diesem Abschnitt wird beschrieben, welche Verbesserungen am Netz vorgenommen wurden und welche Auswirkungen diese erzielen sollen.

3.2 Frame Skipping

Eine durch das Gym Framework bereits gegebene Funktionalität ist das sogenannte Frame Skipping, welches bestimmte Frames nicht zurückgibt, sondern diese einfach überspringt. Hierzu gibt es verschiedene Versionen des Environments für ein und das selbe Spiel, wie z.B. Seaquest-v4, welche nur jeden 4. Frame im Observation-Array zurückliefert. Der Hintergrund zu diesem Verhalten ist, dass nicht jeder Zustand relevant für das Model ist, da die meisten aufeinanderfolgenden Frames nur marginale Unterschiede aufweisen. In der Studienarbeit wird die Version Seaquest-v0 verwendet, da später noch Frame Stacking integriert werden soll und das Überspringen von drei aufeinanderfolgenden Frames hier zu Problemen führen kann. In der Version Seaquest-v0, wird eine zufällige Anzahl zwischen einem und drei Frames übersprungen.

3.3 Preprocessing

Bis jetzt muss das Model mit den rohen Frames arbeiten, die aus dem Environment entnommen werden. Dies hat den Nachteil, dass die Frames mit einer Auflösung von 210x160 Pixel und einer Farbtiefe von 3 Bit eingehen. Da eine sehr große Anzahl von Bildern verarbeitet werden muss, führt die Beschaffenheit der ursprünglichen Frames zu einem enormen Berechnungsaufwand für das neuronale Netz. Eine Reduzierung der Auflösung auf 105x80 Pixel, also die Hälfte der ursprünglichen Auflösung und die Konvertierung der RGB-Farbrepräsentation in Graustufen, führen zu einer drastischen Reduzierung der Speichergröße. Die Frames des Environments werden in einem zweidimensionalen Array mit Float als Datentyp abgerufen und können durch die Division durch 255 auf den kleinsten verfügbaren Datentyp uint8 konvertiert werden, um ebenso erheblichen Speicherplatz einzusparen. Zusätzlich werden noch unnötige Informationen aus dem Bild heraus geschnitten. Der Punktestand, die Lebensanzeige und der Rahmen bieten dem Netz keinen Nutzen im Hinblick auf das Erlernen des Spiels und werden daher entfernt. Dies führt zu einer finalen Auflösung von 70x80 Pixeln, vgl. [5].

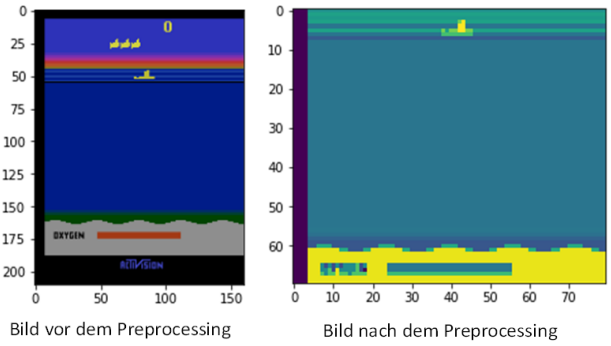


Abbildung 5: Preporcessing der Bilder

3.4 Frame Stacking

Ein überaus wichtiger Bestandteil des Algorithmus ist das Frame Stacking, indem mehrere Frames zu einem Zustand zusammengefasst werden. Ohne Frame Stacking würde das Netz jeweils ein Frame als einen Zustand ansehen. Wenn nun allerdings eine Bewegung eines Objekts wie bspw. der Torpedo eines feindlichen U-Bootes auf den Spieler zusteuert, kann das Netz nur erschwert bzw. nach langen Lernphasen einen Zusammenhang zwischen feindlichem U-Boot und Torpedo feststellen. Mit Frame Stacking hingegen werden vier Frames anstatt wie bisher einem als ein Zustand in das Netz gegeben und somit ein eventueller Zusammenhang von dynamischen Parametern wie Geschwindigkeit und Bewegungsrichtung deutlich, vgl. [4].

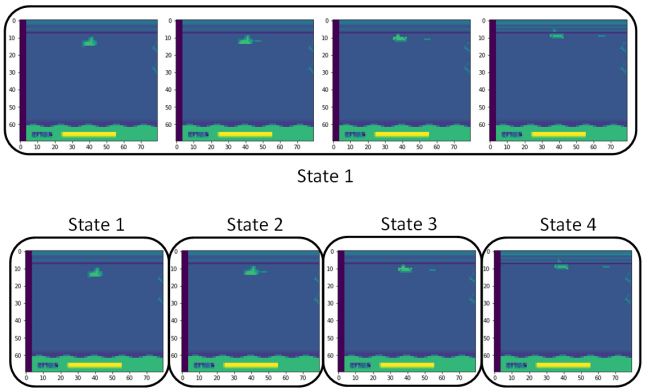


Abbildung 6: Prinzip des Frame Stackings zur besseren Bewegungserkennung

3.5 Experience Replay

Das Konzept des Experience Replay basiert darauf, dass alle erzielten Erfahrungen in einem Speicher abgelegt und zum Lernen des Models durch eine meist geringe Batchgröße verwendet werden. Bei der Speicherstruktur handelt es sich um eine Queue, in der ältere Erfahrungen durch neuere ersetzt werden. Dies soll dem

Overfitting durch korrelierte Zustände entgegenwirken, zudem der Speicher fortlaufend mit neuen Spielerfahrungen durch den Agenten versorgt wird. Somit bildet der Speicher in gewisser Weise das Gedächtnis des Agenten, welcher dazu dient, die Batch immer wieder mit neuen zufällig ausgewählten Erfahrungen zu füllen. Zuerst wurde hierfür die Deque-Struktur verwendet, diese jedoch durch eine eigene Ringbuffer-Klasse ersetzt, da sich herausstellte, dass die Deque-Struktur Schwächen im Bereich des zufallsbasierten bzw. nicht sequentiellen Zufgriffs hat. Der Ringbuffer kann dies durch Verwendung einer sequentielle Speicherstruktur und somit durch Unterstützung von Fast Random Indexing beheben, vgl. [12]. Das zufällige Befüllen der Batch, hat den Hintergrund, dass aufeinanderfolgende Zustände sich sehr ähnlich sind. Ohne Experience Replay könnte das Netz bestimmte Zustände, die es schon länger nicht mehr erreicht hat, einfach vergessen und wüsste nicht mehr, wie es sich bei jenen verhalten sollte. Durch das zufällige Auffüllen erreichen auch ältere Zustände aus dem Speicher die Batch. Hier wird allerdings schnell bewusst, dass dies einen sehr hohen Speicherplatzbedarf mit sich bringt, da die Tupel mit den Daten: Zustand (Frames), Nächster Zustand (Frames), Belohnung (Integer), Done (Boolean) im Speicher Platz finden müssen. Zu diesem Zeitpunkt zeigt sich der zusätzliche Nutzen des vorgenommenen Preprocessings, indem die Bildauflösung und Farbtiefe reduziert wurde und nun so erheblich mehr Frames im Speicher untergebracht werden können. Dennoch verbraucht ein Frame mit der Auflösung von 70x80 Pixel ca. 5,6 KB und da diese mit Experience Replay millionenfach gespeichert werden müssen, bedeutet dies eine Speicherbelegung von 5,6 GB bei einer Million Frames, wobei hier noch keine Aktionen, Belohnungen oder Python Overhead mit eingerechnet sind, vgl. [5]. Ebenfalls zu beachten ist, dass durch das zuvor angewandte Frame Stacking ein Zustand nicht aus einem Frame, sondern aus vier Frames besteht. Dementsprechend benötigt ein Tupel mit vier Frames für den Zustand und vier Frames für den Nächsten Zustand ca. 45 KB Speicherplatz. Das Experience Replay realisiert zwar das Gedächtnis des Agenten, ist jedoch zugleich eine erhebliche Belastung durch den immensen Speicherbedarf und beherbergt somit viel Potenzial zur Verbesserung.

3.6 Huber Verlustfunktion

Allgemein betrachtet hat die Verlustfunktion eines neuronalen Netzes die Aufgabe, die Vorhersage des neuronalen Netzes mit den tatsächlichen Werten zu vergleichen und diese Abweichung bzw. den Fehler hierzu zu berechnen. Der errechnete Fehler kann an die einzelnen Neuronen im Netz zurückgeführt werden, wodurch jedes Neuron seinen Einfluss auf den Ausgabefehler kennt. Danach kann eine Anpassung der Gewichte auf Grundlage des Fehlers erfolgen und das neuronale Netz kann die nächste Vorhersage präziser tätigen bzw. es wurde tatsächlich „gelernt“, vgl. [1].

Ursprünglich arbeitete das Tutorial mit der mittleren quadratischen Abweichung als Verlustfunktion, welche die Schwäche aufweist bei einer schlechten Vorhersage des Modells bzw. einem „Ausreißer“, diesen Fehler, durch die Quadrierung in der Formel, zu stark hervorzuheben. Eine Alternative hierzu wäre es den mittleren absoluten Fehler als Verlustfunktion zu verwenden, welcher in der Lage ist, „Ausreißer“ bei den Vorhersagen zu ignorieren bzw. nicht übermäßig stark zu gewichten. Diese Eigenschaft kann allerdings sogleich ein Nachteil der Verlustfunktion sein, da somit größere Fehlerausschläge genauso gewichtet werden wie kleinere Fehler. Um auch diesen Nachteil zu kompensieren, wurde die Huber Verlustfunktion gewählt, welche die beiden Vorteile der Funktionen miteinander vereint. Die Huber Verlustfunktion benutzt ein Delta, um zu entscheiden, wann es sinnvoll ist die mittlere quadratische Abweichung und wann den mittleren absoluten Fehler als Verlustfunktion zu verwenden. Somit wurde erreicht, dass größere Fehlprognosen nicht zu stark in das Model einfließen, jedoch auch nicht vernachlässigt werden, vgl. [11].

3.7 Spezialisierte Reward-Funktion

Eine der wahrscheinlich wichtigsten Verbesserungen ist das Bereitstellen einer eigenen Reward-Funktion für den Agenten. Die Atari-Umgebung liefert die Belohnung zurück, welcher auch im Spiel erzielt wird. Die Optimierung des Netzes ist an die Belohnung gekoppelt was dazu führt, dass bei komplexeren Spielsituationen eine sehr hohe Anzahl an „Erkundungen“ bzw. gespielten Episoden notwendig wird, bis das Netz den Zusammenhang erkennen kann. Im Spiel Seaquest erhält der Spieler zwar für das Abschießen von Gegnern Punkte, wird allerdings für das Befördern von Tauchern an die Oberfläche wesentlich stärker belohnt. Diesen Zusammenhang kann der Agent bzw. das Netz durch reine Erkundung selbst erlernen, jedoch ist die Dauer hierfür wesentlich länger. Durch eine beeinflusste Belohnung ist es nun möglich den Agenten zu bestimmten Aktionen zu drängen, da durch den erhöhten Reward auch höhere Q-Values für diese Aktionen generiert werden. Ein Beispiel hierfür wäre im Spiel Seaquest, dass das Auftauchen extra bepunktet wird. So könnte man durch Bonuspunkte erreichen, dass sich der Agent wie gewünscht entscheidet, wobei dies natürlich zu Lasten der Allgemeingültigkeit des Algorithmus geht. Eine spezialisierte Reward-Funktion ist auch auf ein gewisses Spiel bzw. höchstens auf eine Gruppe sehr ähnlicher Spiele oder Situationen zugeschnitten. Dies offenbart ein Problem mit der Gym-Library, welche dazu geschrieben wurde unterschiedliche Algorithmen und Lösungsansätze miteinander und zwischen den verschiedenen Simulationsumgebungen, zu vergleichen. Daraus folgt, dass die Library es nicht vorsieht speziell auf die Gegebenheiten eines einzelnen Spiels einzugehen und somit sind die Rückgabeparameter der Step-Funktion (Observation, Reward, Done, Info), die einzigen Drehschrauben, welche man beeinflussen bzw. überschreiben kann. Gym benutzt im Hintergrund zur Simulation des Spiels eine andere Komponente namens Atari Learning Environment (ALE), welche auch die eigentlichen Belohnungen zurückliefert. Die Library Gym stellt also ein Bindeglied zwischen den Frameworks für maschinelles Lernen und der Simulationsumgebung ALE dar. Dies macht auch das Überschreiben der Reward-Funktion auf Seiten der Gym-Library sehr komplex. Eine Folge davon ist, dass nur an den vorhandenen Parametern Anpassungen vorgenommen werden können. Dies sind einerseits Belohnungen und Leben des Spielers, also diejenigen die direkt aus dem Environment kommen, sowie die benötigten Schritte des Agenten bis zum Verlust aller bzw. eines Lebens (Timesteps), welche der Algorithmus selbst bereitstellen kann. Durch längeres Training des Netzes wurde z.B. deutlich, dass

der Agent dazu neigte sich an einer bestimmen Position festzusetzen und dort bis zum vollständigen Verbrauch seines Sauerstoffs abzuwarten, woraufhin versucht wurde dieser Verhaltensweise durch die Vergabe negativer Belohnungen bzw. Bestrafungen entgegenzuwirken. Auch das Erzielen von kleineren Belohnungen in bestimmten Zeitabständen wurde besonders bepunktet, um den Agenten zur Bewegung zu ermutigen. Komplexe Regeln, wie z.B. das der Spieler nur mit mindestens einem Taucher an Board auftauchen darf, um seinen Sauerstofftank aufzufüllen, hätten mit dem Zugriff auf die Position des Spielers in der Reward-Funktion angesteuert werden können. Da dies aber nicht möglich ist, muss der Agent versuchen dies mit längeren Lernphasen zu kompensieren, vgl. [2].

4 BEWERTUNG

4.1 Mögliche Verbesserungen

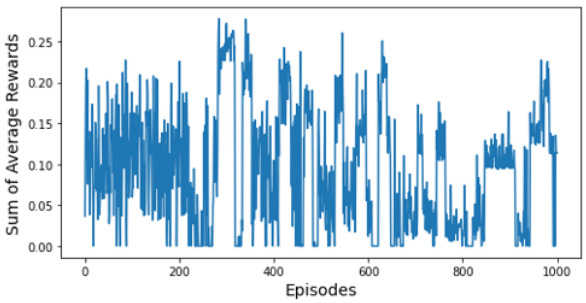
Trotz zahlreichen vorgenommenen Verbesserungen herrscht noch viel Potential zur Optimierung des Netzes bzw. des Algorithmus. In der Literatur finden sich hierzu viele Ansätze bzw. andere Möglichkeiten. Ein Beispiel hierfür sind Double DQN's. Durch diese Herangehensweise soll die Überbewertung von Q-Values verhindert werden, indem ein zusätzliches neuronales Netz, das sogenannte Target Network mit eingebracht wird. Im ersten Schritt ermittelt das DQN die zu wählende Aktion, also diejenige mit dem höchsten Q-Value. Anschließend berechnet das Target Network den Q-Value, unter Verwendung der zuvor ausgewählten Aktion, vgl. [14].

Eine andere Option stellt die Verwendung von Prioritized Experience Replay (PER) dar. Hierbei werden Trainingsdaten in der Batch nicht alle gleich gewertet, sondern Trainingsdaten bzw. Zustände die mehr zum Lernerfolg beitragen priorisiert. Die Ermittlung des Lerneinflusses der Zustände wird über die Temporal Difference Methode erreicht, welche die Q-Values zwei aufeinanderfolgender Zustände miteinander vergleicht. Liegt eine große Differenz der Beiden Q-Values vor, bedeutet dies, dass der Zustand auch eine gewisse Wichtigkeit aufweist. Die Wichtigkeit des Zustandes wird prozentual ausgedrückt und beim Befüllen der Batch berücksichtigt, so dass mehr Zustände mit einer hohen Wichtigkeit in der Batch landen, vgl. [8].

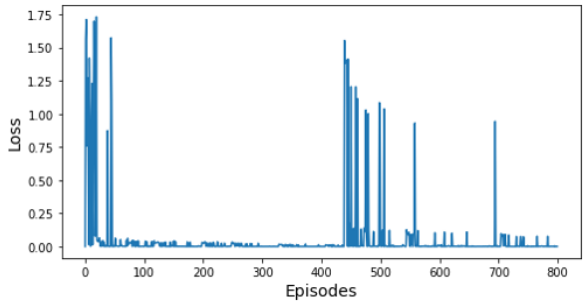
Auch Performanceverbesserungen am bestehenden Algorithmus sind definitiv noch möglich. Ein Beispiel hierfür wäre der aufgrund des Frame Stackings benötigte hohe Speicherplatz, welchen man mit einer effizienteren Speicherung der zu den Zuständen gehörenden Frames entgegenwirken könnte. Momentan werden die Frames für den aktuellen Zustand und für den folgenden Zustand in jeweils getrennten Ringbuffer gespeichert. Allerdings sind einige Frames des aktuellen Zustands identisch mit denen des nächsten Zustands. Dies könnte durch Referenzieren der Zustände auf eine einzelne Speicherstruktur effizienter gestaltet werden so, dass keine Frames mehr doppelt gespeichert werden müssen, sondern nur noch eine Referenz auf den Speicher.

4.2 Ergebnisse

Zuletzt sollen noch die erreichten Ergebnisse vorgestellt werden. Aus zeitlichen Gründen konnten abschießend allerdings nur 1000 Episoden trainiert werden. Für 500 Episoden dauerte die Berechnung bis zu 6 Stunden und aufgrund der häufigen Veränderungen am Algorithmus verblieb zum Schluss nicht mehr sehr viel Zeit für das Training des finalen optimierten Models. 1000 Episoden sind nicht sonderlich aussagekräftig für das Model, da hier das Lernen sozusagen erst wirklich beginnt. Dennoch zeigt die Entwicklung eine grundlegende Veränderung des Models bzw. dessen Arbeitsweise. Die Differenz der einzelnen Diagramme ist darauf zurückzuführen, dass die Werte der Verlustfunktionen natürlich erst vorhanden sind, wenn das Model auch wirklich trainiert. Aufgrund einer eingebauten Aufwärmphase, in welcher der Agent nur spielt um Erfahrungen zu generieren und das Model nicht optimiert wird, sind zu diesen Episoden auch keine Werte vorhanden. Es werden nun jeweils die Diagramme zu den Durchläufen ohne eine spezialisierte Reward-Funktion in Abbildung 7, mit einer spezialisierten Reward-Funktion in Abbildung 8 und nach der Umstellung von einem DQN auf ein Dueling DQN in Abbildung 9 verglichen.

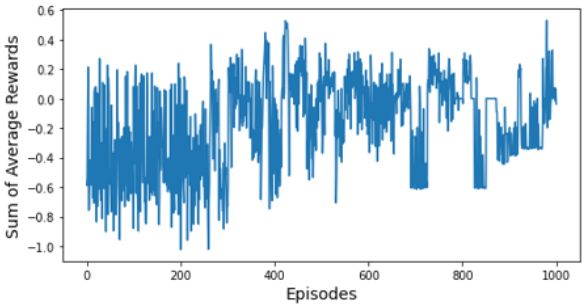


(a) Durchschnittliche Belohnungen ohne spezialisierter Reward-Funktion

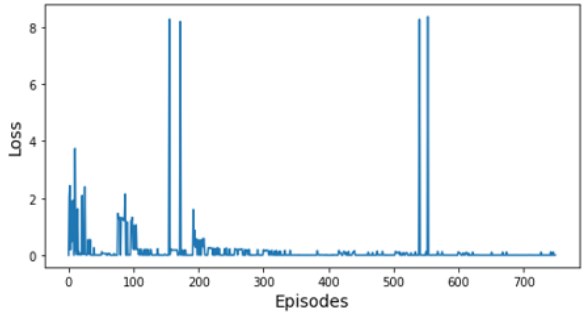


(b) Werte der Verlustfunktion ohne spezialisierter Reward-Funktion

Abbildung 7: Ergebnisse ohne spezialisierter Reward-Funktion

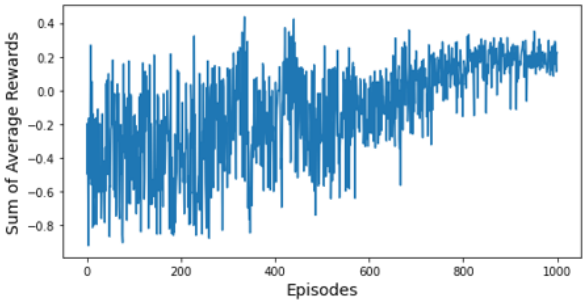


(a) Durchschnittliche Belohnungen mit spezialisierter Reward-Funktion

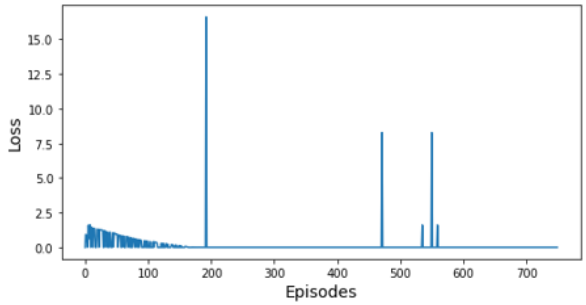


(b) Werte der Verlustfunktion mit spezialisierter Reward-Funktion

Abbildung 8: Ergebnisse mit spezialisierter Reward-Funktion



(a) Durchschnittliche Belohnungen nach der Umstellung auf Dueling DQN



(b) Werte der Verlustfunktion nach Umstellung auf Dueling DQN

Abbildung 9: Ergebnisse nach Umstellung auf Dueling DQN

Aufgrund der Entwicklungen der Diagramme kann abschließend festgestellt werden, dass sich sowohl die durchschnittlich erzielten Belohnungen als auch die Ausschläge der Verlustfunktion positiv verändert haben. Die höheren bzw. konstanteren Resultate sind auf die eingebrachten Verbesserungen zurückzuführen und zeigen, dass diese zu einem effizienteren und stabileren Algorithmus geführt haben.

LITERATUR

[1] Benjamin Aunkofer. [n.d.]. Training eines Neurons mit dem Gradientenverfahren. https://data-science-blog.com/blog/2019/01/13/training-eines-neurons-mit-dem-gradientenverfahren/. Accessed: 25.07.2020.

[2] AurelianTactics. [n.d.]. Creating a Custom Reward Function in Retro Gym and Other Utilities. https://medium.com/aureliantactics/creating-a-custom-reward-function-in-retro-gym-and-other-utilities-33c9f783bd1a. Accessed: 25.07.2020.

[3] EBattleP. [n.d.]. Markov diagram. https://commons.wikimedia.org/wiki/File:Markov_diagram_v2.svg. Accessed: 25.07.2020.

[4] Adrien Lucas Ecoffet. [n.d.]. Beat Atari with Deep Reinforcement Learning! (Part 0: Intro to RL). https://becominghuman.ai/lets-build-an-atari-ai-part-0-intro-to-rl-9b2c5336e0ec. Accessed: 25.07.2020.

[5] Adrien Lucas Ecoffet. [n.d.]. Beat Atari with Deep Reinforcement Learning! (Part 1: DQN). https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26. Accessed: 25.07.2020.

[6] Ray Heberer. [n.d.]. Why Going from Implementing Q-learning to Deep Q-learning Can Be Difficult. https://towardsdatascience.com/why-going-from-implementing-q-learning-to-deep-q-learning-can-be-difficult-36e7ea1648af. Accessed: 25.07.2020.

[7] Prof. Dr. Sebastian Leuoth. [n.d.]. AML - Bestätigendes Lernen. https://moodle.hof-university.de/pluginfile.php/213654/mod_label/intro/AML_BL.pdf. Accessed: 25.07.2020.

[8] Parsa Heidary Moghadam. [n.d.]. Deep Reinforcement learning. https://medium.com/@parsa_h_m/deep-reinforcement-learning-dqn-double-dqn-dueling-dqn-noisy-dqn-and-dqn-with-prioritized-551f621a9823#:~:text=Double. Accessed: 25.07.2020.

[9] Parsa Heidary Moghadam. [n.d.]. Dueling DQN Vorlage. https://github.com/Parsa33033/Deep-Reinforcement-Learning-DQN/blob/master/Dueling-DQN.py.
[10] Robert Moni. [n.d.]. Reinforcement Learning algorithms. https://medium.com/@SmartLabAI/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc. Accessed: 25.07.2020.
[11] George Seif. [n.d.]. Understanding the 3 most common loss functions for Machine Learning Regression. Understandingthe3mostcommonlossfunctionsforMachineLearningRegression. Accessed: 25.07.2020.
[12] Takuma Seno. [n.d.]. Welcome to Deep Reinforcement Learning. https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b. Accessed: 25.07.2020.
[13] Thomas Simonini. [n.d.]. Dueling Deep Q Networks. https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751. Accessed: 25.07.2020.
[14] Thomas Simonini. [n.d.]. Improvements in Deep Q Learning. https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/. Accessed: 25.07.2020.
[15] Andre Violante. [n.d.]. Einfuehrung in Reinforcement Learning. https://www.statworx.com/de/blog/einfuehrung-in-reinforcement-learning-wenn-maschinen-wie-menschen-lernen/. Accessed: 25.07.2020.
[16] Andre Violante. [n.d.]. Simple Reinforcement Learning: Q-learning. https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56. Accessed: 25.07.2020.
[17] Felix Yu. [n.d.]. Deep Q Network vs Policy Gradients. https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html. Accessed: 25.07.2020.

Abbildungsverzeichnis

Erstellung eines Schachagenten

Mit Reinforcement-Learning

Felix Herrmannsdörfer
Sommersemester 2020
Informatik, Hochschule Hof
Hof, Bayern, Deutschland

ABSTRACT

Schach ist ein komplexes Spiel mit langer Tradition. Sogar für Computerprogramme stellt die Analyse von Schach durch die hohe Anzahl der möglichen Zustände eine große Herausforderung dar.

Im Zuge dieser Arbeit wurden drei verschiedene Ansätze für einen Schachagenten auf Grundlage des maschinellen Lernens mit Reinforcement Learning analysiert. Jeder Ansatz basiert dabei auf einem bestehenden, öffentlichen Projekt. Anschließend wurden verschiedene Anpassungen an den Projekten getestet, um das jeweilige Projekt möglicherweise zu verbessern.

CCS CONCEPTS

• Computing methodologies → Machine Learning → Learning paradigms → Reinforcement learning

• Computing methodologies → Machine Learning → Learning settings → Learning from critiques

• Theory of computation → Theory and algorithms of application domains → Machine learning theory → Models of learning

KEYWORDS

Schach, Schachagent, bestärkendes Lernen, Maschinelles Lernen, Neuronale Netze

ACM Reference format:

Felix Herrmannsdörfer. 2020. Erstellung eines Schachagenten – Mit Reinforcement-Learning. *Hof, Bayern, Deutschland, 5 pages*

1 Einleitung

Brettspiele wie Schach oder Go zählten aufgrund ihrer hohen Komplexität lange Zeit als sehr schwierig mit Software zu lösen. Bis 2016 hielt man es sogar für unmöglich GO unter aktuellen technischen Möglichkeiten mittels maschinellen Lernens zu meistern.[1] Umso größer war die Überraschung als 2016 AlphaGo und sein späterer Nachfolger AlphaZero Weltmeister ohne Probleme besiegte.[2]

Im Rahmen des Moduls Angewandtes maschinelles Lernen von Herrn Prof. Dr. Leuoth an der Hochschule Hof befasste sich Felix Herrmannsdörfer mit dem Ziel einen Schachagenten zu erstellen. Dieser Schachagent soll durch reines Spielen mit sich selbst oder einem einfachen Gegner stetig besser werden. Ziel war es dabei allerdings nicht direkt einen besonders guten Schachagenten zu erstellen, sondern vor allem die möglichen Herangehensweisen und Konzepte zu verstehen.

1.1 Erläuterung Schach

Schach ist ein Brettspiel in dem 2 Spieler versuchen zuerst den gegnerischen König zu besiegen. Das Brett besteht aus 64 Feldern, auf denen jeweils ein Spielstein stehen kann. Spielsteine werden besiegt, indem man eigene Spielsteine auf ein Feld mit einem gegnerischen Spielstein bewegt.
Neben dem König gibt es noch 5 andere Arten von Spielsteinen mit jeweils eigenen Bewegungsmustern. In Abbildung 1 sieht man den Startzustand eines Schachspiels.

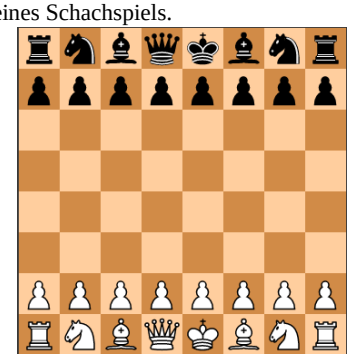


Abbildung 1: Schach im Startzustand, erstellt mit python-chess in Colab

1.2 Erläuterung bestärkendes Lernen

Allgemein gesehen gibt es beim bestärkendem Lernen 2 Komponenten: den Agenten und die Umgebung. Die Umgebung hat zu jeder Zeit einen bestimmten Zustand. Aufgrund dieses Zustands wählt der Agent eine Aktion, die ihm am besten erscheint. Durch diese Aktion ändert sich der Zustand der Umgebung in der Regel. Für jede Aktion erhält der Agent auch eine Belohnung. Diese dient dem Agenten dazu einschätzen zu können, wie gut seine Aktion wirklich war, um dann direkt oder später daraus lernen zu können.[3]

2 Erstellung des Schachagenten

Dass das Erstellen eines Schachagenten mittels maschinellen Lernens nicht einfach ist, stellt man schon nach kurzer Onlinerecherche fest. Heute gibt es einige größere Projekte deren Ausmaße enorm sind. Einen eher einfachen Ansatz zu finden stellte daher schon ein gewisses Problem dar. Schlussendlich wurden drei Projekte ausgewählt, aufgrund derer jeweils ein Jupyter-Notebook in Google Colab [1] entstand. Diese können jeweils im Anhang gefunden werden.
Ein Teil der Projekte wird immer die freie Schachumgebung python-chess [2] sein. Von dieser muss der Zustand des Boards immer noch in eine für das Model geeignete Form gebracht werden, bevor es weitergegeben werden kann. Dafür kümmert es sich allerdings um alle Schachangelegenheiten.

2.1 Erster Ansatz

Der erste Ansatz der Arbeit entspringt eines öffentlichen Projektes aus Google Colab.[3]

2.1.1 Analyse des bestehenden Models

Hier sollte das Model durch das Spielen gegen ein Schachprogramm in sehr einfacher Stufe lernen.
Wie in Abbildung 2 zu sehen wurde auch das Schachspiel deutlich vereinfacht, indem nur 5 Spielsteine zu Beginn platziert wurden.

Abbildung 2: Vereinfachter Startzustand, Erstellt mit python-chess in Colab

Für das Model wurde jedem Spielstein eine eigene Ganzzahl zugeordnet. Die Spielsteine des Gegners erhalten jeweils denselben negativen Wert.
Die Input-Ebene des Models erhält 64 Werte, also einen für jedes Feld. Das Model besteht aus 2 weiteren voll vernetzen Ebenen. Die erste Ebene davon hat 512 Ausgabewerten und wird durch die relu-Funktion aktiviert. Die letzte Ebene hat 4096 Ausgabewerte und wird durch Softmax aktiviert.
Das Model bewertet anhand des aktuellen Spielbretts, wie gut es wäre einen Spielstein von einer Position auf eine bestimmte Position zu bewegen. Durch 64 Spielfelder, auf denen ein Stein stehen kann, und wieder 64 Spielfelder, auf die der Stein bewegt werden könnte, ergibt sich in der Multiplikation die im Model festgelegte Ausgabegröße von 4096. Allerdings stellt man auch schnell fest, dass viele von diesen Ausgaben eigentlich keinen Sinn ergeben. Zum Beispiel zählt es normalerweise nicht als erlaubter Zug einen Spielstein auf demselben Feld stehen zu lassen oder wenn es an einer bestimmten Position keinen Spielstein gibt, ist ein Zug von dort aus auch nicht möglich. Dies wird aber trotzdem immer mitberechnet und ausgegeben. Diese Fälle werden später aussortiert und nur die Ausgaben für die wirklich möglichen Züge zur Bewertung herangezogen. Das dies hier notwendig ist liegt daran, dass in Schach die Anzahl und die Positionen der möglichen Züge von Zug zu Zug unterschiedlich sein werden. Daher ist es bei diesem Ansatz nicht möglich dem Model eine einfachere Ausgabegröße zu geben.

2.1.2 Eigene Umsetzung

Als erstes wurde geprüft, wie das aktuelle Model sich schlägt, wenn das Schachbrett mit dem normalen Spielbeginn gespielt wird. Wie man in Abbildung 3 sehen kann, ist auch hier das Model durchaus in der Lage sich etwas zu verbessern. Allerdings ist die Anzahl der gewonnen Spiele (in Abbildung 3 links unten) doch gleichbleibend sehr gering.

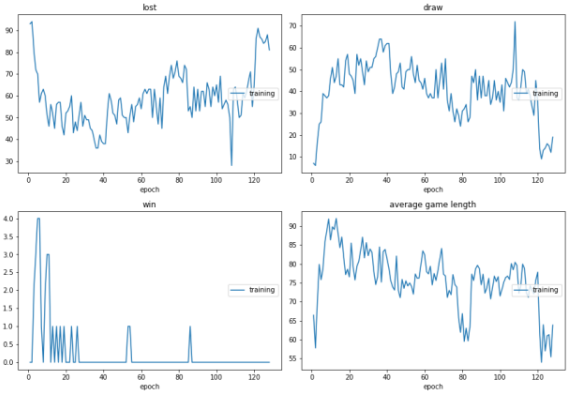


Abbildung 3: Verlauf der Spiele im ersten Ansatz, erstellt mit livelossplot in Colab

Das erste Ziel war es das Model von PyTorch [4] auf Keras zu übertragen. Dies stellte sich allerdings als sehr schwierig heraus, da

[1] https://colab.research.google.com/
[2] https://python-chess.readthedocs.io/en/latest/
[3] https://colab.research.google.com/drive/1Kir-xWX8piHPf6pDu-jbWmGz-eNQkJyy
[4] https://pytorch.org/

sich die beiden Frameworks doch deutlich voneinander unterscheidet.
Trotz vielen Tests und Onlinerecherche konnte erst später mit ein wenig Hilfe der Vorlage aus Ansatz 3 ein ähnliches Model halbwegs funktional eingesetzt werden.[4] Allerdings waren die Ergebnisse des Trainings in allen getesteten Varianten des Models schlechter als das Original. Vermutlich liegt das an der benutzerdefinierten Verlustfunktion, die dem Original mit PyTorch nicht exakt nachempfunden werden konnte.

2.2 Zweiter Ansatz

Der zweite Ansatz basiert auf einem öffentlichen Gitlab Repository[5].

2.2.1 Analyse des bestehenden Models

Als Hauptunterschied zum ersten Ansatz stellt man fest, dass hier nicht mehr die möglichen Züge bewertet werden, sondern das Spielbrett an sich. Um einen Zug auszuwählen, wird für jeden Zug das daraus entstehende Spielbrett bewertet und der Zug, aus dem das beste Brett hervorging, ausgewählt.
Für die Bewertung des Models wurde ein tiefes, voll vernetztes Model verwendet, dass am Schluss in einer Softmax-Ebene endete. Damit sollte das Board in eine von 3 Klassen unterteilt werden, die jeweils angeben ob das Board Weiß, Schwarz oder ein Unentschieden begünstigt.
Allerdings war das Projekt noch nicht vollständig, denn das komplette Belohnen und Trainieren des Models fehlte.

2.2.2 Fertigstellung und Erweiterung

Belohnungen wurden ähnlich wie im ersten Ansatz so umgesetzt, dass die letzten Spielbretter als wichtig eingestuft wurden und alle vorherigen durch Multiplikation mit einen Wert Gamma zwischen 0 und 1 abgestuft wurde. Die Klassifikation wurde auf zwei Klassen begrenzt, da es reichen sollte zu wissen, ob schwarz oder weiß für besser gehalten wird. Die Verlustfunktion konnte daher auch auf binäre Kreuzentropie geändert werden. Das finale Model kann Abbildung 4 entnommen werden.

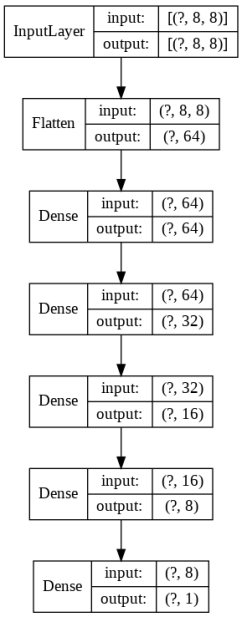


Abbildung 4:Model aus Ansatz 2, erstellt mit Keras in Colab

Da nur zwei Instanzen vom selben Model gegeneinander spielten, um zu trainieren ließ sich nach dem Training schlecht sagen, ob sich die Modelle wirklich verbesserten. Deshalb wurde später das Schachprogramm aus Ansatz 1 in dieses Projekt übertragen. Das Ergebnis war, dass der Agent definitiv besser war als im ersten Ansatz. Aber die Siegesrate lag noch deutlich unter 50%.

2.3 Dritter Ansatz

Ansatz Nummer drei basiert auf der Kaggle Serie „Reinforcement Learning Chess“ des Nutzers ArjanGroen.[5]
In dieser Serie beschreibt er sein Vorgehen bei der Entwicklung eines Schachagenten mit Keras. Sein viertes von fünf Notebooks hatte eine sehr ähnliche Funktionsweise wie das Projekt aus dem ersten Ansatz und half somit später dort ein einigermaßen funktionales Model in Keras umzusetzen. In diesem Ansatz wird allerdings das vielversprechendere letzte Notebook hergenommen. Die Notebooks aus Kaggle beinhalten allerdings nicht den eigentlich wichtigen Code, sondern nur dessen Anwendung. Der eigentliche Code für alle Notebooks befinden sich in seinem öffentlichen Gitlab[6].

2.3.1 Analyse des bestehenden Models

Grundsätzlich ist die Herangehensweise für das Model selbst ähnlich zum zweiten Ansatz, denn auch hier werden die Spielbretter bewertet. Allerdings wird das Spielbrett anders formatiert als bisher. Statt einem 8x8-Vektor wird nun ein 8x8x8-Vektor für das Model verwendet. Dieser ist so aufgebaut, dass darin die ersten sechs 8x8-Vektoren jeweils für eine bestimmte Spielsteinart alle Positionen auf dem Brett mit 1 oder -1 bestimmen.

[5] https://github.com/Ajetski/chess-ai

[6] https://github.com/arjangroen/RLC

Die übrigen beiden sind mit Spielinformationen gefüllt. Einer enthält in allen Felder den Wert 1 geteilt durch die aktuelle Zugnummer, außer in der ersten Reihe. Diese wird, abhängig davon welche Seite am Zug ist, mit dem Wert 1 oder -1 gefüllt. Der letzte 8x8-Vektor enthält immer vollständig den Wert 1.
Statt nur einem Model sind hier auch gleich eine ganze Reihe Modelle definiert. Alle Modelle sind aber grundsätzlich ähnlich aufgebaut. Von der Eingabeebene werden zuerst eine oder mehrere 2DConvolution-Ebenen unterschiedlicher Art ausgeführt. Diese werden dann in eine oder mehrere voll vernetzte Ebenen weitergeleitet, die mit der Sigmoid-Funktion aktiviert werden. Die letzte Ebene ist immer eine voll vernetzte Ebene ohne Aktivierungsfunktion. Die Ausgabe befindet sich immer zwischen 1 und -1. Dabei steht 1 für komplett weiß favorisiert und -1 für komplett schwarz favorisiert. Das initial verwendete Model namens „bignet“ besteht aus acht verschiedenen Convolution-Ebenen mit relu-Aktivierung, die alle in ein Netz aus fünf voll vernetzten Ebenen mit Sigmoid-Aktivierung münden. Der Output ist wie bereits in Ansatz 2 ein einzelner Wert zwischen 0 und 1.
Im vorherigen Ansatz wurde immer nur das Board des nächsten Zuges bewertet. Nun wird stattdessen immer ein MonteCarlo-Baum anhand der möglichen Züge aufgebaut. Durch die Baumstruktur erlernt der Agent nun die Möglichkeit voraus zu planen. Die MonteCarlo-Baumsuche hilft dabei sich zuerst auf die Zweige des Baumes zu fokussieren, die am wahrscheinlichsten wichtig sind.[6] Allerdings muss die Baumsuche trotzdem anhand einer maximalen Zeit oder Suchtiefe eingeschränkt werden. Damit soll der hohe Rechenaufwand, der durch die hohe Anzahl der möglichen Zustände einhergeht, eingedämmt werden, ohne dabei viel Genauigkeit der Vorhersage zu verlieren.
Zusätzlich zur Belohnung des Endresultats fliest hier auch noch der Zustand des Spielbretts mit in die Belohnung ein. Für die Bewertung des Boards werden die Spielsteine anhand der Reinfeld-Werte aufsummiert, die gegnerische Seite der Spielsteine zählt dabei negativ.
Zum Trainieren spielt das Model gegen einen „gierigen“-Agenten. Dieser wurde so programmiert, dass er immer die Züge wählt, die den Wert des Spielbretts (vorerst) am meisten zu seinen Gunsten optimieren. Falls es mehrere beste Züge gibt, wählt er einen zufälligen davon aus. Wenn ein Zug zum sofortigen Gewinn führen sollte, wird allerdings immer dieser bevorzugt.
Da in dem Projekt aber nicht ausgegeben wird wie viele Spiele der Agent gewinnt, ist es schwierig die Effizienz des Models zu bewerten. Auffällig ist allerdings, dass das Training sehr lange dauert. In einer Stunde schafft das Model durchschnittlich gerade einmal 35 Spiele zu spielen und das obwohl die Spiele ab 80 Zügen abgebrochen und als unentschieden gewertet werden.

2.3.2 Eigene Versuche und Anpassungen

Zuerst wurde das Model so erweitert, dass man nach jedem Spiel den Endzustand des Boards angezeigt wurde, um einen ersten Eindruck über die Spielresultate zu erhalten. Allein an den letzten Zügen ließ sich gut erkennen, dass beide Agenten besonders schlecht darin sind das Spiel effektiv zu beenden. Wenn eine Seite gewann, dann schien es meist aus Zufall.
Da das Training so lange braucht und Google Colab nach einigen Stunden Inaktivität die Ausführung abbricht, wird das Model stündlich auf einem Google-Drive-Laufwerk gesichert. Außerdem wird beim Speichern nun auch immer die durchschnittliche Gewinnrate der letzten Stunde mit ausgegeben, um die Verbesserungen leichter erkennen zu können.
Nach 100 Spielen ist die Siegesrate noch sehr gering. Nach etwa 1000 Spielen schwank sie etwa um die 50%. Wenn man nun die maximalen Züge auf 200 erhöht, bleiben sehr häufig nur noch die Könige übrig. Dies bestätigt, dass beide Spieler Probleme haben das Spiel für sich zu gewinnen, obwohl der Agent eigentlich vorausplanen kann.
Verschiedene kleinere Änderungen wie andere Optimierer oder Änderungen an den Convolution-Ebenen brachten keine ersichtlichen Verbesserungen.

3 Fazit

Schach ist und bleibt ein sehr komplexes Problem für Neuronale Netze. Jeder einfachere Ansatz konnte nicht annähernd mit dem Schachprogramm auf sehr leichter Stufe mithalten. Der letzte Ansatz ist zwar etwas vielversprechender, aber immer noch relativ schwach. Es ist nun fraglich ob das Model noch viel mehr Training benötigt oder ob das Model selbst verbessert werden muss, um eine höhere Siegesrate zu erreichen. Eine Möglichkeit wäre auch die beiden Ebenen der 8x8x8 Eingabe herauszunehmen, die nichts mit dem Schachbrett an sich zu tun haben, um sie extra über eine weitere Eingabeebene in das Model einfließen zu lassen. Möglicherweise versteht das Model den Spielstand so besser und kann trotzdem die Anzahl der bisherigen Züge mit in die Entscheidungen einfließen lassen. Außerdem könnte man versuchen zusätzliche Informationen zu übergeben, wie z.B. welcher Spielstein wird von anderen Spielsteinen gedeckt oder welche Spielsteine sind in Gefahr geschmissen zu werden.
Eventuell wäre es auch möglich das Training mehr zu parallelisieren, also mehrere Instanzen in verschiedenen Threads zeitgleich am selben Model trainieren zu lassen, um das langsame Training etwas zu verbessern. Allerdings müsste dabei auch sehr auf die Ressourcenauslastung geachtet werden, die durch die hohe Anzahl der berechneten Zustände entstehen.
Häufiger wird wegen der besseren Trainingsgeschwindigkeit das Model zum Lernen auch erst mit Daten aus einer Vielzahl von bereits gespielten Schachspielen angelernt.
Alles in allem merkt man jetzt, wie viel Arbeit für einen guten Schachagent nötig sind und dementsprechend, welch enorme Arbeit in Projekten wie AlphaZero stecken müssen.

REFERENCES

[1] Steve Connor. 2016. *A computer has beaten a professional at the world's most complex board game*. The Independent. https://www.independent.co.uk/life-style/gadgets-and-tech/news/google-alphago-computer-beats-professional-at-worlds-most-complex-board-game-go-a6837506.html. Zuletzt abgefragt August 2020

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis. 2017. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*.

[3] Chris Nicholson: *A Beginner's Guide to Deep Reinforcement Learning*, https://pathmind.com/wiki/deep-reinforcement-learning, zuletzt besucht August 2020.

[4] Arjan Grojen. 2019. *Reinforcement Learning Chess 4: Policy Gradients* . https://www.kaggle.com/arjanso/reinforcement-learning-chess-4-policy-gradients. zuletzt besucht August 2020.

[5] Arjan Grojen. 2020. *Reinforcement Learning Chess 5: Tree Search* . https://www.kaggle.com/arjanso/reinforcement-learning-chess-5-tree-search. zuletzt besucht August 2020.

[6] Ziad SALLOUM. 2019. *Monte Carlo Tree Search in Reinforcement Learning*. https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f#e407

Vergleich der beiden Ansätze (Supervised Learning & Q-Learning) in Bezug auf das Spiel StarCraft2

Kai Heinl
Hochschule Hof
kai.heinl@hof-university.de

Alexander Blankenhagen
Hochschule Hof
alexander.blankenhagen@hof-university.de

Zusammenfassung

In dieser Studienarbeit werden die Resultate festgehalten, die bei der Verwendung unserer unterschiedlichen Ansätze im Maschinellen Lernen (ML) entstanden sind. Als Basis des Experiments dient das Spiel „StarCraft2“, auf das wir sowohl das Supervised als auch das Q-Learning anwenden konnten. Für die Bilderkennung wurde ein Convolutional Neural Network (CNN) genutzt. Ideengeber war das Python AI Tutorial, an das wir uns am Anfang orientierten. (Vgl. [5]) Dabei verwendeten wir das Grundgerüst des Programms und änderten es nach unseren Anforderungen. Für die Durchführung wurden verschiedene Frameworks genutzt. Die Programmierung wurde mit Python durchgeführt und konnte mit den Erweiterungen Keras, Python-SC2 und Tensorflow für die gestellten Anforderungen angepasst werden.

1 Einleitung

1.1 Aufbau der Studienarbeit

Zunächst wird der grobe Aufbau unseres Experimentes erläutert. Das zweite Kapitel widmet sich dem Thema „Machine Learning“ und soll einen groben Überblick über die eingesetzten Verfahren geben. Kapitel drei beschäftigt sich mit der Implementierung der Verfahren Q-Learning und Supervised Learning. Am Schluss der Arbeit fasst ein Resümee die gewonnenen Erkenntnisse zusammen.

1.2 Beschreibung des Spiel StarCraft 2

StarCraft 2 ist ein Science-Fiction-Echtzeit-Strategiespiel, das im Juli 2010 von dem Entwicklerstudio Blizzard veröffentlicht wurde. (Vgl. [15]) Das Ziel des Spiels ist die Zerstörung aller Gebäude des Feindes. Für das Rekrutieren der Einheiten benötigt man zwei Arten von Ressourcen – Vespingas und Mineralien. Beides ist endlich und nicht allzu üppig vorrätig. Das wiederum zwingt den Spieler zu expandieren, das unausweichlich zur Konfrontation mit dem Gegenspieler führt. Der Spieler kann zwischen drei Rassen wählen: Protoss, Terran und Zerg. Jede dieser Nationen hat bestimmte Eigenschaften und besondere Einheiten.

Im Jahr 2017 veröffentlichte Blizzard mit DeepMind eine Machine Learning API für das Spiel. Mit PySC2 wurde eine Reinforcement Umgebung geschaffen, die eine direkte Schnittstelle zum Code von StarCraft2 darstellt. (Vgl. [13])
Neben PySC2 gibt es eine weitere Python API mit dem Namen python-sc2, die in der Studienarbeit zum Einsatz kam. Die API wurde von dem Nutzer Dentosal auf GitHub veröffentlicht und kann kostenlos verwendet werden (Vgl. [4]). Unser Ziel ist es, zwei unterschiedliche Bots zu entwickeln, die entweder Supervised Learning oder Q-Learning verwenden sollen. Um die jeweils erzielten Ergebnisse vergleichbar zu machen, wurde darauf geachtet, dass jeder Bot in allen Spieldurchläufen die gleichen Gebäude baut. Die KI sollte sich lediglich auf die Steuerung von Militäreinheiten konzentrieren.

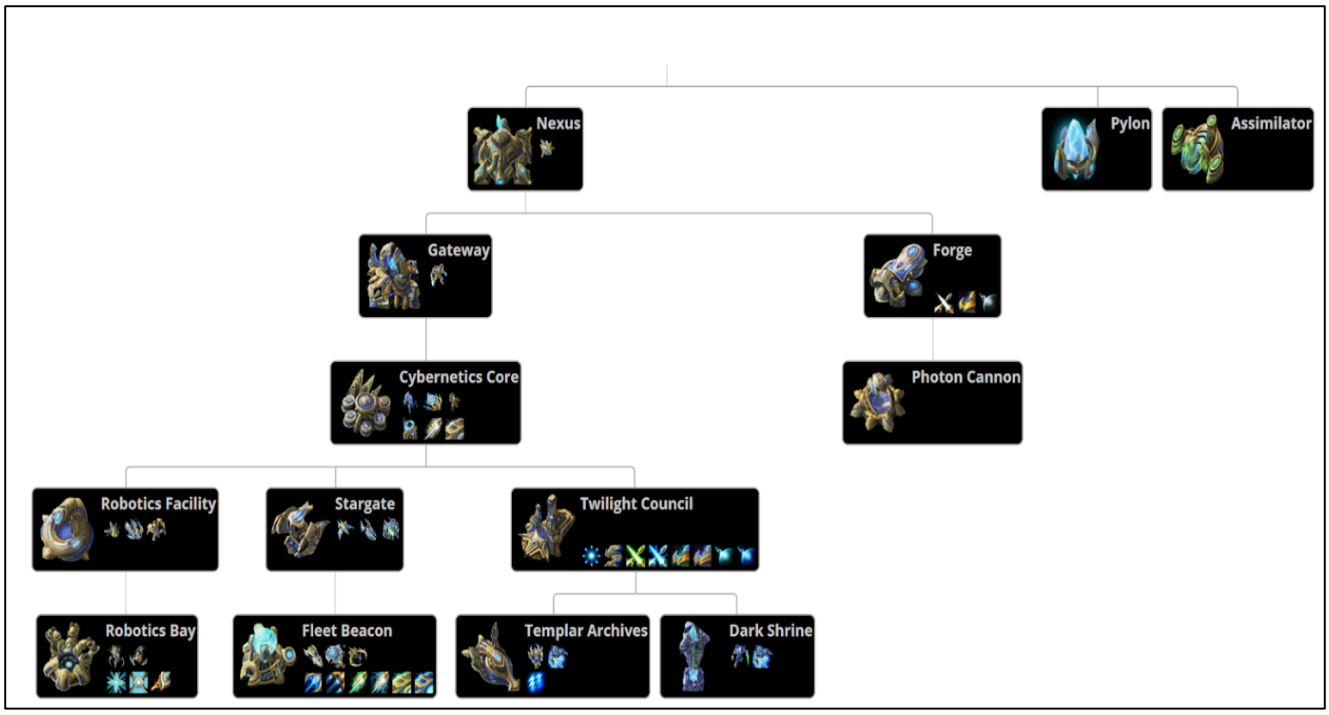


Abbildung 1: Gebäudestammbaum der Protoss (Vgl. [10])

In Abbildung 1 ist die Gebäudeauswahl der Protoss zu erkennen. Für die Studienarbeit wurde neben dem Hauptgebäude Nexus noch ein Gateway, zwei Assimilatoren, mehrere Pylons und ein Cybernetics Core gebaut. Alle anderen Gebäude wurden für den Experimentenaufbau ignoriert. Für die Ausbildung von Einheiten dienen eine Robotics Facility sowie zwei Stargates. Nachdem der Code für die Errichtung von Gebäuden erstellt wurde, konnte der Bot die erwähnten Gebäude autonom bauen. Die Erkundung der Karte und zur Observierung des Gegners entschieden wir uns für den Bau von sogenannten „Observern“. Lediglich zu Beginn eines Spieles werden aufgrund der fehlenden Baumöglichkeit für diese Aufgabe Sonden benutzt. Die Observer sind Späher, die Speziell für diese Aufgaben gedacht sind. Um den Experimentenaufbau so einfach wie möglich zu halten, entschlossen wir uns für den Bau einer einzelnen Militäreinheit – den Void Rays. Void Ray sind Flugeinheiten, die zu einer vergleichbar starken Armee zusammengeführt werden können. Die beiden KI sollen nur die Steuerung der Void Rays übernehmen. In Abbildung 2 sehen wir einen Screenshot des Void Ray und in der Abbildung 3 wird ein Observer vorgestellt.

Abbildung 2: Screenshot eines Void Rays

Abbildung 3: Observer (Vgl. [3])

2 Machine Learning

Das Maschinelle Lernen ist ein Teilgebiet der Künstlichen Intelligenz (KI) und steht immer mehr im Rampenlicht einiger IT-Fachzeitschriften, Kongresse und Lerneinrichtungen. Noch vor einigen Jahren war es lediglich möglich, an ausgewählten Forschungseinrichtungen und Universitäten etwas über „Machine Learning“ zu erfahren bzw. es zu verwenden. In der heutigen Zeit gehört „Machine Learning“ an vielen Bildungseinrichtungen zum Pflichtprogramm und erlebt auch in der Gesellschaft einen Boom.
„Machine Learning“ wird beispielsweise für die Spracherkennung der Smartphones oder Tablets eingesetzt, sowie für das Erkennen von Spam-Mails im Postfach. Die IT befindet sich im Wandel und die Anwendungen, die unser Leben erleichtern sollen, werden immer komplexer. Durch „Machine Learning“ werden komplizierte und vielschichtige Probleme vereinfacht und können dem Nutzer das Leben versimpeln. (Vgl. [2])

2.1 Supervised Learning

Supervised Learning ist ein Teilgebiet des Maschinellen Lernens und verwendet zur Erlernung und Erkennung von Regelmäßigkeiten ein Neuronales Netz. Dieses „Überwachte Lernen“ ist eine Art maschineller Lernalgorithmus, das einen validierten Trainingsdatensatz verwendet, um Vorhersagen über nachfolgende Ereignisse zu geben. Dabei enthält der Trainingsdatensatz bestimmte Eingabedaten und die jeweiligen, daraus resultierenden Ergebnisse. Nachdem die Trainingsdaten erzeugt wurden, wird anhand dieser Daten das Model mithilfe des Lernalgorithmus trainiert. Je mehr Daten dem Model zum Trainieren gegeben werden, desto höher fällt die Vorhersagekraft des Models aus. Einen groben Überblick über die verschiedenen Arten des Maschinellen Lernens soll die Abbildung 4 verschaffen. Für einen Teil der Studienarbeit wurde Supervised Learning mit einem Convolutional Neuronal Network verwendet.

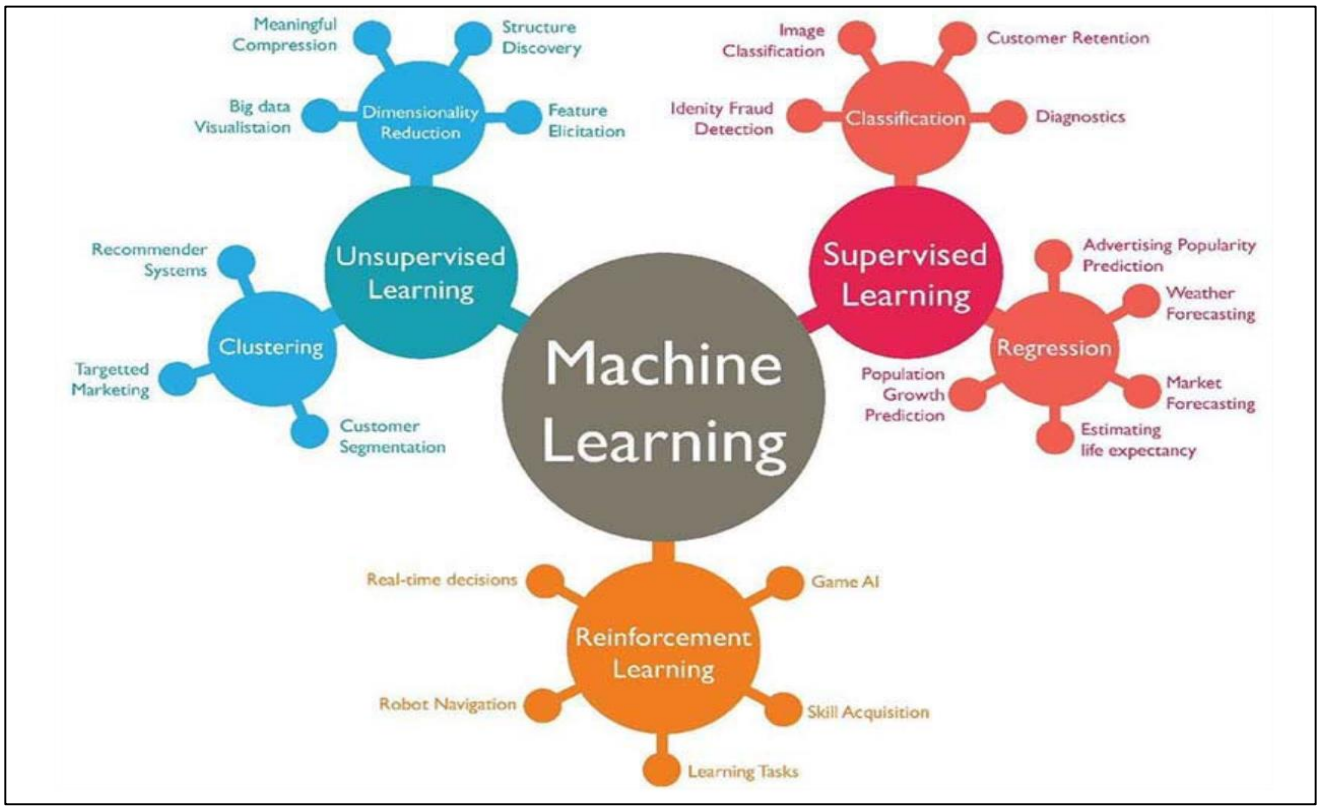


Abbildung 4: Machine Learning (Vgl. [1])

2.2 Convolutional Neuronal Network

Convolutional Neuronal Network (CNN) ist eine „Deep Learning“ Architektur, die zur Erkennung von visuellen oder auditiven Mustern entwickelt wurde. Es wird deshalb häufig für die Erkennung von Audio und Bilddateien eingesetzt, da durch das „Falten“ (Convolution) die Geschwindigkeit der Berechnungen erhöht wird und den Lernprozess beschleunigt. Ein CNN besteht aus Filtern (Convolutional Layer) und Aggregations-Schichten (Pooling Layer), die sich abwechselnd erneuern. Das Ergebnis der Lernphase besteht aus einer oder mehreren erzeugten Schichten, den sogenannten „Fully Connected Layer“. Sie stellen das herausragende Merkmal eines solchen Netzwerkes dar.

Der Input wird mithilfe einer Matrix verarbeitet. Dabei kann der Input aus Bildern bestehen, die eine feste Größe haben, wie zum Beispiel 3 x 3 x 3(Breite x Höhe x Farbkanäle).

In der ersten Schicht wird der Input von Filtern analysiert und extrahiert. Die zweite Schicht (Pooling Layer) reduziert die Auflösung und aggregiert die Ergebnisse des Convolutional Layer, indem nur das stärkste Signal gespeichert wird. Zum Schluss kommt der „Fully Connected Layer“. Bei dieser Schicht sind alle Input - und Output Daten miteinander verbunden. Wie viele Schichten das CNN am Ende besitzt, hängt von der Anzahl der Neuronen ab, die den verschiedenen Klassen entsprechen. (Vgl. [14])

2.3 Q-Learning

Bestärkendes Lernen (Reinforcement Learning) ist ein Teilgebiet des Maschinellen Lernens. Q-Learning ist wiederum ein Teilgebiet des Bestärkenden Lernen. Es wird verwendet, wenn keine Trainingsdaten, mit denen man ein neuronales Netz trainieren könnte, vorhanden sind. Um das Netz (Agent) dennoch trainieren zu können, nutzt man den aktuellen Status (state s) des Systems (Environment) und die Entscheidungen (action a), die getroffen wurden. Dies wird im Schaubild in Abbildung 5 ersichtlich. Zu Beginn werden die Entscheidungen per Zufall getroffen und anhand des Rewards (reward r) bewertet. Das neuronale Netz wird mittels des Status des Systems trainiert, indem Entscheidungen und Rewards stetig neu festgelegt werden. Mit zunehmender Anzahl der Spiele übernimmt das neuronale Netz zunehmend die Entscheidungen. Für einen kleinen festgelegten Prozentsatz (Exploration Rate) werden jedoch weiterhin die Entscheidungen per Zufall getroffen. Andere Möglichkeiten der Entscheidungsfindung werden damit nicht vollends außer Acht gelassen. (Vgl. [8])

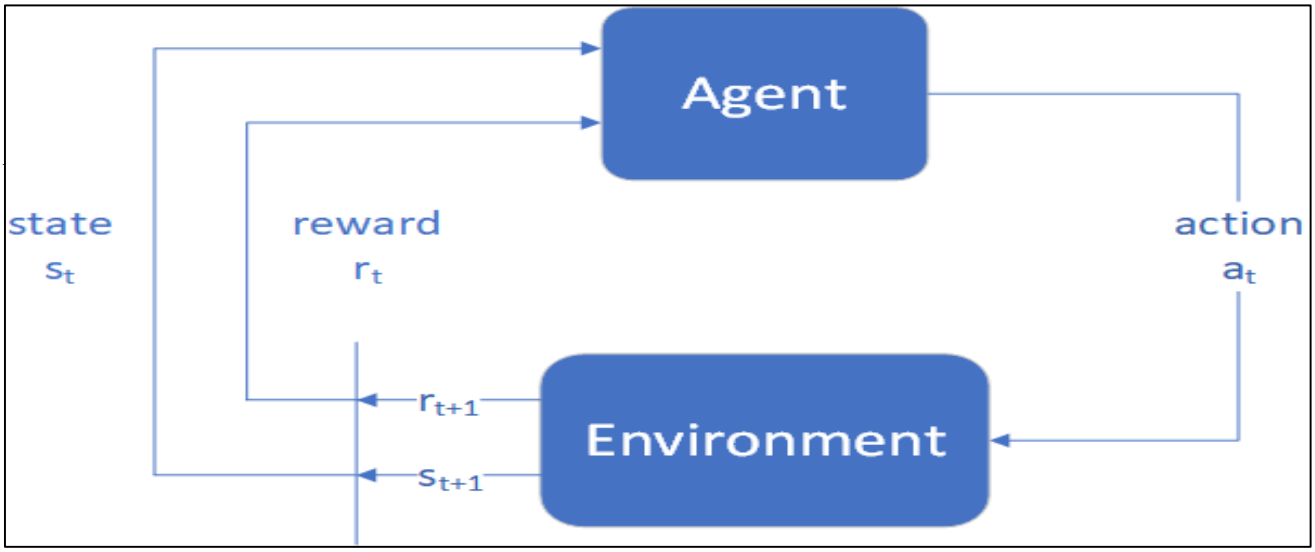


Abbildung 5: Vereinfachte Darstellung des Q-Learnings (Vgl. [6])

3. Theorie

In diesem Kapitel werden die Verfahren und Frameworks erläutert, die in der Studienarbeit zum Einsatz kamen.

3.1 python-sc2

Python-sc2 ermöglicht den einfachen Einstieg zum Schreiben von AI Bots für das Spiel StarCraft 2. Wie bereits in Kapitel 1.2 beschrieben wurde, gibt es derzeit zwei verschiedene API´s für die Interaktion mit dem Spiel. Aufgrund der einfachen Zugänglichkeit wurde für die Arbeit die API python-sc2 verwendet (anstelle der pySC2 von Blizzard und DeepMind).

3.2 Keras und TensorFlow

Keras ist eine in Python geschriebene Bibliothek für neuronale Netze und seit 2017 Teil der TensorFlow (TF) Bibliothek. Beide Anwendungen sind Open-Source und wichtiger Bestandteil des maschinellen Lernens. (Vgl. [7])
Keras Besonderheit ist unter anderem die benutzerfreundlichen APIs und bietet dem Nutzer bei Problemen strukturierte und klare Fehlermeldungen, die für den User leicht zu verstehen sind. (Vgl. [11])
Die Programme werden durch die starke Communityarbeit immer weiterentwickelt, um den Stand der Technik in ML voranzutreiben. TensorFlow verfügt über ein flexibles Ökosystem aus Tools und Bibliotheken, um sehr einfach und ohne großes Hintergrundwissen eigene Modelle zu erstellen. Dabei besteht die Hauptaufgabe des Frameworks darin, Modelle des Maschinellen Lernens zu bauen und zu trainieren. TensorFlow gehört zu Google und wird auch für die eigenen Produkte, wie beispielsweise Google Maps, verwendet. (Vgl. [12])

4 Implementierung der beiden Verfahren

In diesem Kapitel geht es um die Implementierung der beiden Arten des Maschinellen Lernens. Zur Verständlichkeit und Lesbarkeit wird hierbei nur auf die wichtigsten Teile der Implementierung eingegangen.

4.1 Allgemeine Implementierung

Um den gleichen Experimentenaufbau zu gewährleisten, müssen, wie bereits beschrieben, einige Aktionen des Bots vorgeschrieben sein.
Als Grundstein wurde der Code von Harrison Kinsley verwendet, der aber stark an die Anforderungen angepasst wurde. Zunächst wurden die nötigen Gebäude integriert. Dabei wird immer überprüft, ob genügend Ressourcen vorhanden sind, um die jeweiligen Gebäude zu errichten. Des Weiteren ist eine Funktion eingebaut, die zerstörte Gebäude wieder neu errichtet. Es gibt Gebäude, die öfter gebaut werden müssen und solche, die nur einmalig errichtet werden sollten.
Die Steuerung und Rekrutierung der Einheit „Observer“ ist ebenfalls hart in das Spiel programmiert. Dabei wird die fertige Funktion des Tutorials verwendet. Neben der Steuerung der „Observer“ werden auch die Sammler von den Standardfunktionen gesteuert und ebenfalls rekrutiert.
Erst wenn das Stargate gebaut und der erste Void Ray erstellt wurde, beginnt die Arbeit der KI und der Unterschied zwischen dem Supervised Learning und dem Q-Learning wird erkennbar.

4.1 Erstellung der fünf-Choices

Nachdem diese Infrastruktur programmiert wurde, hat das erste Model fünf verschiedene Aktionen (Choices) bekommen, das es ausführen soll. Gespielt wurde, wie in der folgenden Abbildung 6 zu erkennen ist, auf einer großen Karte, die eine Größe von 128 x 128 Pixel hatte.

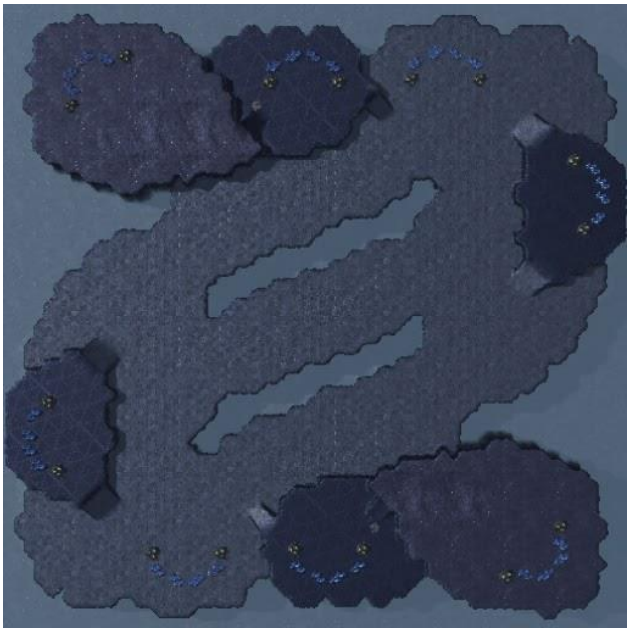


Abbildung 6: Screenshot der Karte 128 x 128

Zur besseren Übersicht sind in dem nachfolgenden Codeausschnitt 1 die fünf Choices aufgelistet.

```
if choice == 0:
    # print("No Attack")
    await self.do_nothing()

elif choice == 1:
    # print("Attack known enemy units")
    await self.attack_known_enemy_units()

elif choice == 2:
    # print("attack known enemy structures")
    await self.attack_known_enemy_stuctures()

elif choice == 3:
    # print("attack enemy start position")
    await self.attack_enemy_start_position()

elif choice == 4:
    # print("move to start location")
    await self.move_to_start_location()
```

Codeausschnitt 1: fünf verschiedene Aktionen

In unserer ersten Phase sollte das Model fünf verschiedene Aktionen verwenden können. Dabei wurde der Aufbau so schlicht wie möglich gestaltet, damit die Models schnellstmöglich den Zeitpunkt der Ausführung der Befehle erlernen. Die Choice eins bedeutet, dass die Void Rays nicht handeln sollen. Choice Nummer zwei soll bekannte gegnerische Einheiten attackieren. Aktion drei veranlasst die Void Rays dazu, die gegnerische Infrastruktur anzugreifen. Choice Nummer 4 lässt die Startposition des Gegners angreifen. Die letzte Choice bewegt die Void Rays zur eigenen Startposition. Sie dient der Verteidigung der eigenen Basis.

4.2 Erstellung der siebzehn-Choices

Um die Resultate der vorherigen Version zu verbessern, ist die Anzahl der Choices auf siebzehn gestiegen Außerdem wurde eine kleinere Spielkarte verwendet, um die Anzahl der Zugmöglichkeiten in einem berechenbaren Ausmaß zu halten. Das Weitern wurden feste Koordinaten angelegt, an denen der Bot den Gegner angreifen soll. Wie in Abbildung 7 ersichtlich wird, ist dazu das Spielfeld in vier Quadranten eingeteilt worden. Das hat den Grund, dass bei beiden Arten des ML die Anzahl der unentschiedenen Spiele stark angestiegen ist. Der Bot soll alle Gebäude des Gegenspielers zerstören, um eine Partie zu gewinnen. Werden nach einer bestimmten Zeit keine Mineralien mehr abgebaut oder Gebäude zerstört, bricht StarCraft 2 das Spiel ab und wertet dieses als unentschieden. Die trainierten Models haben zwar häufig den Gegner zur Spielaufgabe gezwungen, doch hat es uns die API nicht ermöglicht, in solchen Fällen eine

Siegeswertung zu erhalten. Im späteren Verlauf der Arbeit wird dieses Thema noch einmal aufgegriffen.

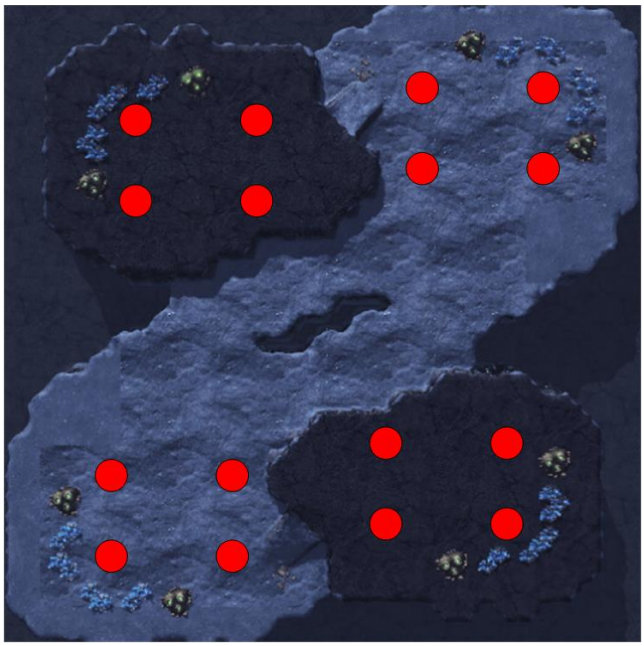


Abbildung 7: Screenshot der Karte 64 x 64

Zudem wird im Folgenden Codeausschnitt 2 ein Teil der siebzehn Choices gezeigt.

```
self.targets.append(position.Point2((28, 64)))
self.targets.append(position.Point2((37, 64)))
self.targets.append(position.Point2((28, 53)))
self.targets.append(position.Point2((40, 58)))
```

Codeausschnitt 2: Ausschnitt aus den siebzehn Choices

In dem obigen Codeausschnitt 2 sind vier verschiedene Koordinaten zu lesen. Der erste Parameter ist die X- und die zweite ist die Y-Koordinate. Zusammen ergeben diese vier Punkte ein Rechteck, in dem die Void Ray alles zerstören sollen.

4.3 Erstellung der Unitmap

Sowohl für das Q-Learning als auch für das Supervised Learning benötigen wir eine Visualisierung der Spieldaten, um die Models mit Inputs zu füttern. Dazu verwendeten wir den Code des bereits erwähnten Tutorials und änderten ihn entsprechend den Anforderungen. In der folgenden Abbildung 8 ist ein Screenshot der Unitmap zu sehen.

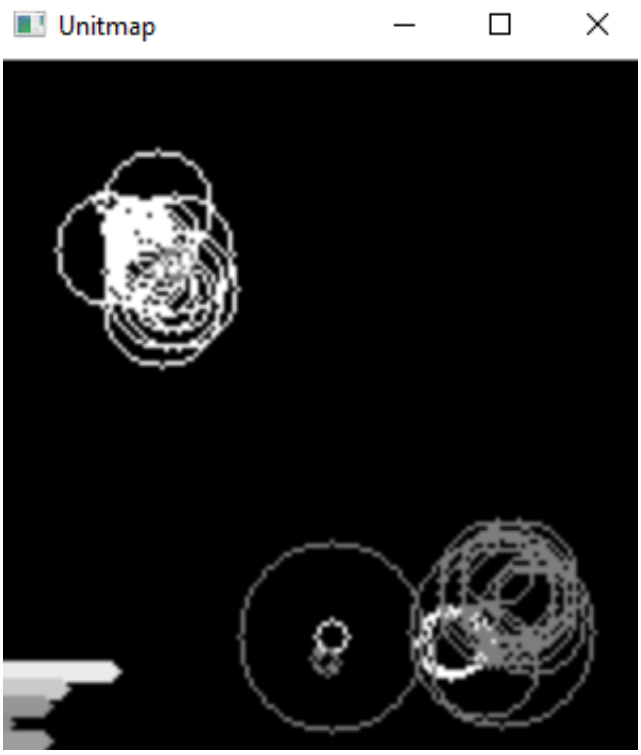


Abbildung 8: Screenshot der Unitmap

Zu erkennen ist die Visualisierung der Daten des Spiels, wie beispielsweise Einheiten und Gebäude. Links oben (weiß) ist die eigene Basis identifiziert worden. Rechts unten (grau) ist die feindliche Basis. Des Weiteren ist unten rechts ein weißer Ring ersichtlich, der einen Void Ray symbolisieren soll. Links daneben ist ein kleinerer, weißer Kreis, der für einen Observer steht. Die linken Balken im Bild sollen die verschiedenen Ressourcen visualisieren. Diese Unitmap agiert für die beiden Models als Auge und füttert die Models mit Input.

4.4 Supervised Learning

Das folgende Kapitel wird in zwei Bereiche aufgeteilt – zunächst wird das Vorgehen mit den fünf - und anschließend mit den siebzehn Choices erläutert.

Nachdem der Experimentenaufbau bereit war, konnten die Trainingsdaten erzeugt werden. Dazu wurden 1000 Spiele per Zufall ausgeführt. Die Zufallsfunktion wählte alle drei Sekunden eine der fünf Choices und führte diese aus. Nach drei Sekunden gab es wieder eine zufällige Entscheidung. Gespeichert wurden nur die gewonnenen Spiele. Dabei werden die zufälligen Schritte mit der Unitmap aufgezeichnet und in den Trainingsdaten gespeichert. Die verlorenen und unentschieden ausgegangen Spiele werden verworfen und werden für das Model nicht weiter berücksichtigt. Nach den 1000 Spielen, bei dem die Choices zufällig ausgewählt wurden, wurden ca. 100-120 Spiele

gewonnen. Die folgende Abbildung 9 zeigt den Graphen, der mit diesem Vorgehen entstand.

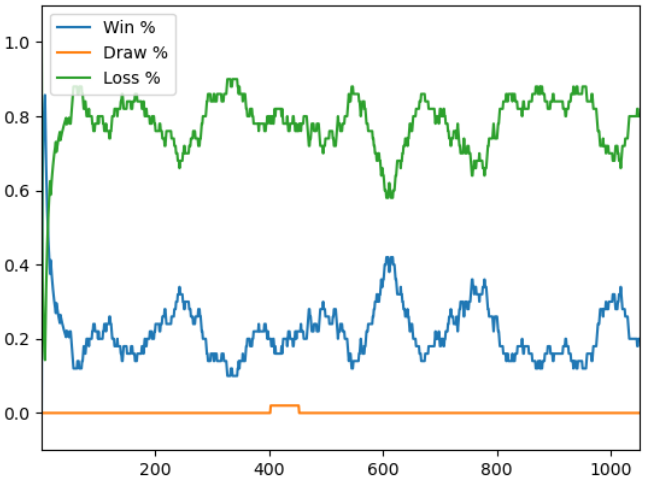


Abbildung 9: 1000 Spiele per Zufallsfunktion (128 x 128)

Mithilfe der gewonnenen Spiele und der daraus erzeugten Trainingsdaten konnte das Model erstmalig trainiert werden. Zum besseren Verständnis: Der grüne Graph gibt die Wahrscheinlichkeit für verlorene Spiele an. Der blaue dagegen, gibt die Wahrscheinlichkeit für gewonnene Spiele an. Der orangefarbene Graph visualisiert die unentschiedenen Spiele.

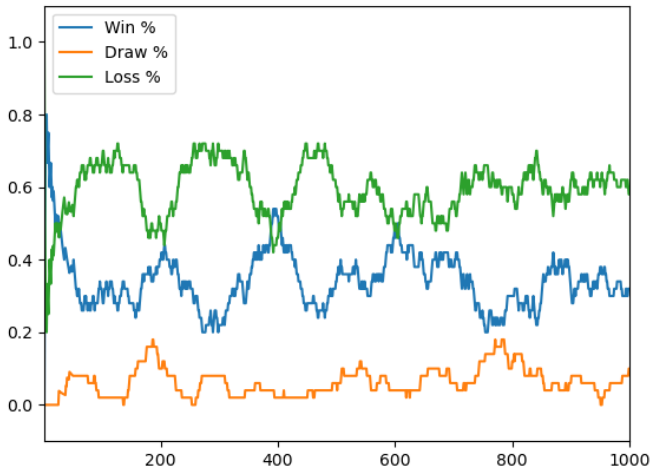


Abbildung 10: 1000 Spiele mit Model_V1 (128 x 128)

Wie aus der Abbildung 10 zu entnehmen ist, ist mit dem neuen Model (Model_V1) die Anzahl der gewonnenen Spiele gegenüber der Zufallsfunktion (siehe Abbildung 9) angestiegen. Es verliert zwar öfter, als dass es gewinnt, jedoch ist eine gewisse Verbesserung zu erkennen. Des Weiteren ist die Anzahl der Spiele gestiegen, die unentschieden ausgingen. Wie in dem vorherigen Kapitel 4.1 und 4.2 schon erwähnt, gab es allerdings Probleme mit dem Radius der Void Rays. Dieses Model zeichnet sich dadurch aus, dass es anhand der Trainingsdaten Entscheidungen (Predictions) trifft, die zum Erfolg führen sollen.

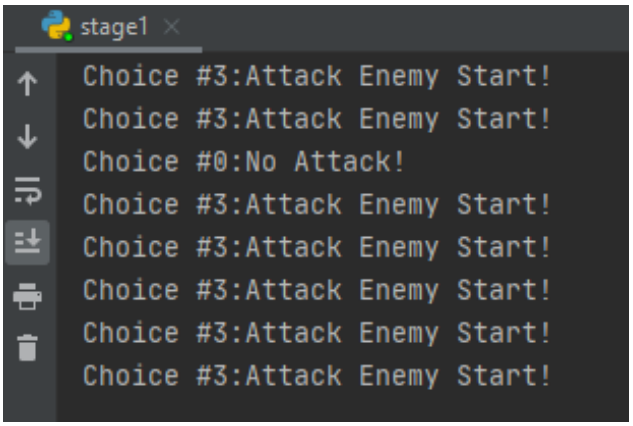


Abbildung 11: Screenshot während der Benutzung des Models_V1

Wie aus Abbildung 11 zu erfassen ist, führt das Model nicht irgendwelche zufälligen Aktionen aus, sondern macht Vorhersagen, die auf validierten Daten beruhen. Der Screenshot in Abbildung 11 ist zum Start des Spiels aufgenommen worden und demonstriert gleich die Entschlossenheit des Models, ein Spiel zu gewinnen. Nach dem Durchlauf der 1000 Spiele mit diesem Model sind weitere Trainingsdaten entstanden. Genau wie davor, wurden hier nur die gewonnenen Spiele gespeichert. Als nächstes wurde ein verbessertes Model mit den gerade erzeugten Daten trainiert. Nachfolgend sind weitere 1000 Spiele durchlaufen worden, deren Ergebnisse in Abbildung 12 zusammengefasst wurden.

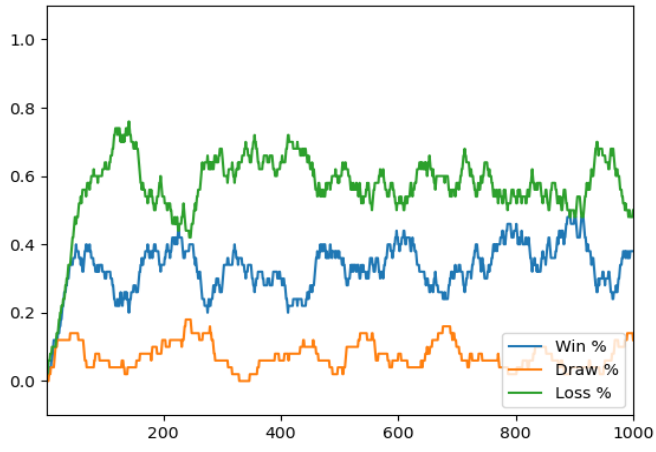


Abbildung 12: 1000 Spiele mit Model_V2 (128 x 128)

Der Unterschied zwischen Model_V1 (Abbildung 10) und Model_V2 (Abbildung 12) ist kaum zu erkennen. Bei näherer

Betrachtung fällt auf, dass die Wahrscheinlichkeit für eine unentschiedenen Partie beim Model_V2 leicht anstieg. Theoretisch sind die unentschiedenen gespielten Spiele gewonnene Spiele und müssten auch zu den gewonnenen Partien zählen. Wie bereits erwähnt, konnte aber das Aufgeben des Gegners nicht mithilfe der API abgefangen werden. Eine manuelle Eingabe hätte das Model negativ beeinflusst und somit waren wir gezwungen, die unentschiedenen Spiele nicht als gewonnen hinzunehmen.
Zur weiteren Verbesserung des Models wurde die Anzahl der Choices erhöht und die Kartengröße verringert. Zunächst wurden, wie zuvor, wieder 1000 Spiele per Zufall gespielt. Anschließend wurde mithilfe der gewonnenen Spiele das Model trainiert. Nachstehend befindet sich in der Abbildung 13 der Graph für das zufällige Auswählen der Choices.

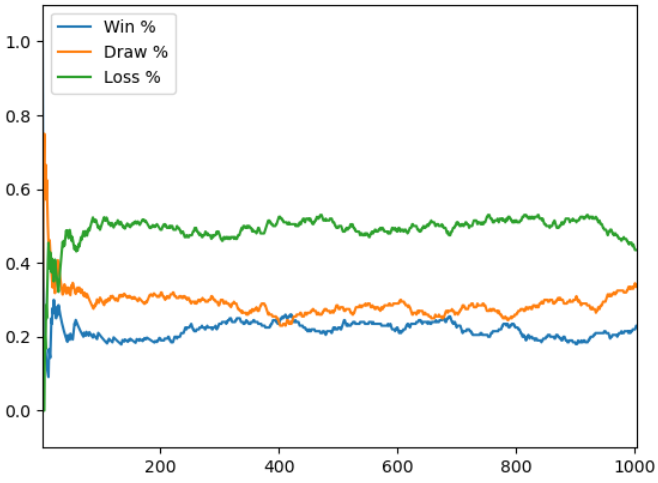


Abbildung 13: 1000 Spiele per Zufall (64 x 64)

Abbildung 13 repräsentiert einen gewöhnlichen Zufallsgraphen. Die Wahrscheinlichkeit, dass das Spiel verloren wird, ist am größten. Anzumerken bleibt jedoch, dass, obwohl die Anzahl der Möglichkeiten stark anstieg, das Model erstaunlicherweise relativ oft gewinnt. Auch ist die Anzahl der unentschiedenen Spiele auffällig. Nach dem die Trainingsdaten erzeugt wurden, wurde das Model mit den gewonnenen Daten trainiert.

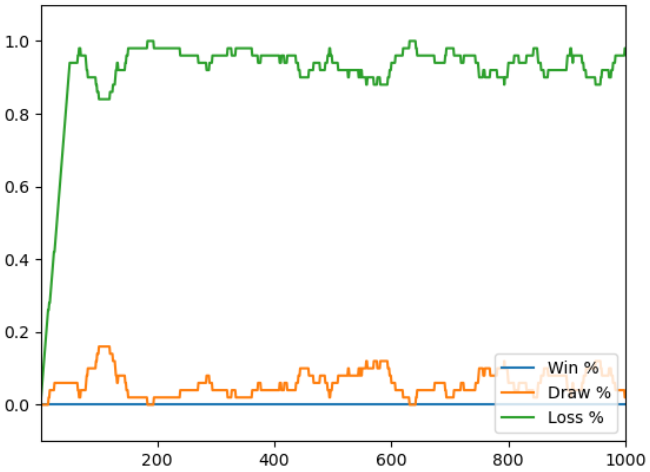


Abbildung 14: 1000 Spiele mit Model und siebzehn Choices (64 x 64)

In Abbildung 14 wird das desaströse Ergebnis von weiteren 1000 gespielten Spielen, bei denen das Model aktiv war, erkenntlich. Es wurde kein einziges Spiel gewonnen und die Wahrscheinlichkeit für eine Niederlage liegt fast bei 100%.

4.5 Q-Learning

Um das Model möglichst gut trainieren zu können, ist es notwendig, die gewählten Aktionen möglichst gut zu bewerten. Dafür muss die Reward-Funktion entsprechend des Ziels (hier: Sieg) definiert werden.
Definition des Rewards:

- Punkt 1: Sieg: +1
- Punkt 2: Niederlage/Unentschieden: -1 (Da ein Sieg den einzig akzeptablen Ausgang darstellt, sind Niederlage und Unentschieden unerwünscht. Niederlage = Verlust aller eigenen Gebäude)
- Punkt 3: Sieg = Zerstörung aller Gebäude des Gegners. Daher ist es wichtig, möglichst viele Gebäude des Gegners zu zerstören. Der Wert (in Ressourcen) der zerstörten Gebäude wird durch 1000 geteilt. (Somit erhält man einen Wert zwischen 0 und 1, z.B. 0,4 für einen Nexus)
- Punkt 4: Eine Niederlage tritt beim Verlust aller eigenen Gebäude ein. Daher ist es wichtig, so wenig eigene Gebäude wie möglich zu verlieren. Der Wert (in Ressourcen) der verlorenen Gebäude wird durch -1000 geteilt. (Somit erhält man einen Wert zwischen 0 und -1, z.B. -0,4 für einen Nexus)
- Punkt 5: Um Gebäude zerstören oder verteidigen zu können, werden Kampfeinheiten benötigt. Daher ist es zwar wichtig, selbst viele Einheiten zu produzieren und die Anzahl der gegnerischen Einheiten klein zu halten (durch Zerstörung dieser). Entscheidend jedoch ist, weniger Verluste bei Kämpfen als der Gegner zu erleiden. Der Wert (in Ressourcen) der eigenen Verluste

wird von dem Wert (in Ressourcen) der gegnerischen Verluste abgezogen. Das Ergebnis wird durch 1000 dividiert. (z.B. (400-600)/1000=-0,2)

Da über die gewählte API der Zugriff auf die eigenen Verluste (Gebäude sowie Einheiten) nicht möglich ist, scheiden Punkt 4 und 5 aus und wir erhalten folgende Reward-Funktion, siehe Abbildung 15:

- R = 1 (Sieg) oder R = -1 (Niederlage oder Unentschieden)
- ZG = zerstörte Gebäude
- ZE = eigene, verlorene Gebäude
- zG = zerstörte Einheiten
- zE= eigene, verlorene Einheiten

$$r = R + \frac{ZG}{1000} + \frac{ZE}{-1000} + \frac{zG - zE}{1000}$$

Abbildung 15: Rewardfunktion

Die Entscheidungen a, die Rewards r und die States s werden gesammelt. Nach Abschluss eines Spiels wird, wie in dem Codeausschnitt 3 zu sehen ist, das Model mit den gesammelten Daten trainiert.

```
def learn(self):
    for screen, action, reward, next_screen, done in self.memory:
        target = reward
        if not done:
            prediction = self.model.predict(next_screen)[0]
            target = (reward + self.gamma * np.amax(prediction))
        target_f = self.model.predict(screen)
        target_f[0][action] = target
        self.model.fit(screen, target_f, epochs=1, verbose=0)
    self.memory.clear()
```

Codeausschnitt 3: Ausschnitt der Lernfunktion

Der erste Versuch wird mit einer einfachen Reward-Funktion r (R = 1 (Sieg) oder R = -1 (Niederlage oder Unentschieden) und mit Categorical Crossentropy als Loss-Funktion durchgeführt.

$$r = R$$

Abbildung 16: Änderung des Rewards

Dabei werden ca. 1000 Spiele auf der Karte 128x128 (Simple128) mit den fünf Choices gespielt. Wie auf folgender Abbildung 17 ersichtlich wird, kann kein erkennbarer Lernfortschritt erzielt werden. Es werden jedoch mehr Spiele gewonnen als mit zufälligen Entscheidungen.

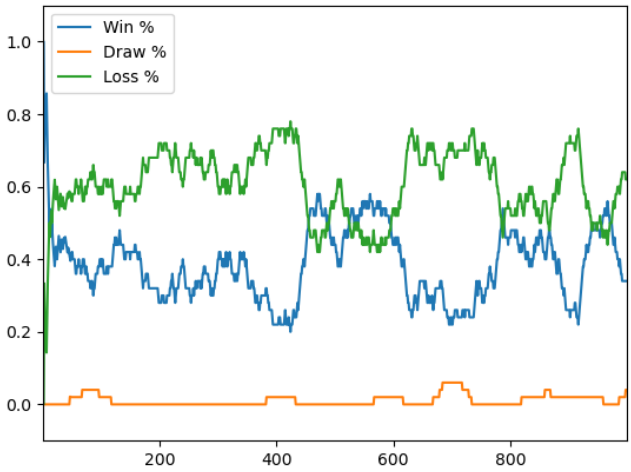


Abbildung 17: Erster Versuch mit 1000 Spielen und fünf Choices

Da das Model beim zweiten Versuch siebzehn Choices (ohne vier Quadranten, Loss-Funktion=MSE) bei demselben Reward zu treffen hat, wird auf der kleineren Karte Simple 64 gespielt. Auf der untenstehenden Abbildung 18 wird deutlich, dass, solange die Entscheidungen per Zufall getroffen werden, noch wenige Spiele gewonnen werden. Je mehr Entscheidungen das Model übernimmt, desto weniger Spiele werden allerdings gewonnen. Da ab ca. 250 Spielen keine Änderung mehr zu erkennen ist und sich das Model auf eine Auswahlmöglichkeit festgefahren hat, wird das Training bei über 400 Spielen händisch beendet.

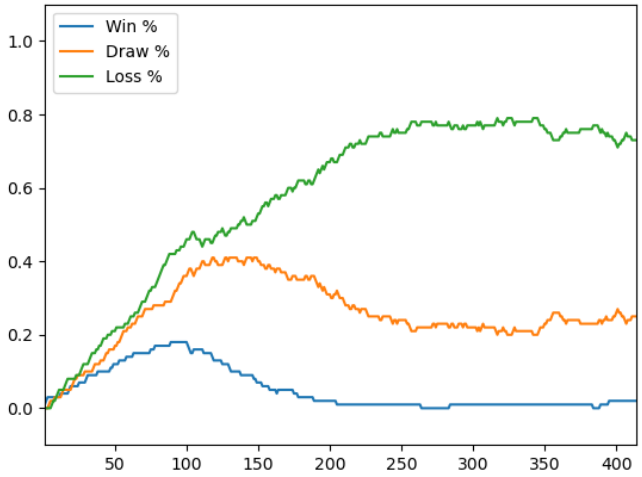


Abbildung 18: Zweiter Versuch mit 400 Spielen und siebzehn Choices

Beim dritten Versuch hat das Model 17 Choices (bei 4 Quadranten, Loss-Funktion=MSE, Reward: r=R) zu treffen. Dabei werden, wie der folgenden Abbildung 19 zu entnehmen ist, keinerlei Spiele mehr gewonnen, noch wird eines unentschieden gespielt. Auch nicht bei den ersten 100 Spielen, bei denen noch per Zufall entschieden wird. Zusätzlich hat sich das Model wieder auf eine Auswahlmöglichkeit festgefahren. Daher wird hier ebenfalls bei über 400 Spielen das Training manuell beendet.

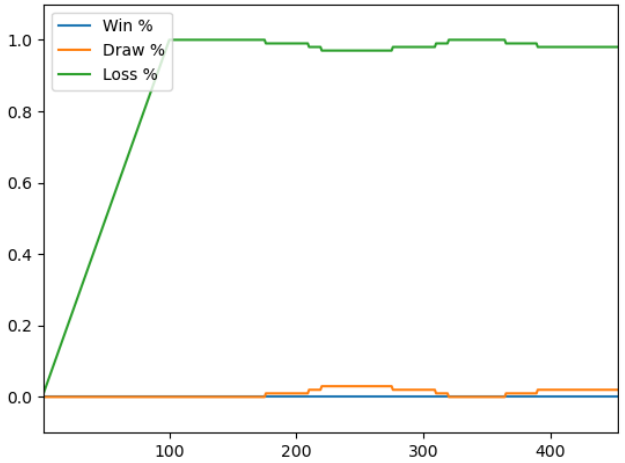


Abbildung 19: Dritter Versuch mit über 400 Spielen und siebzehn Choices

Beim vierten Versuch wird die Loss-Funktion von mse zu Categorical Crossentropy geändert und eine neue Reward-Funktion r verwendet.

$$r = \mathrm{R} + \frac{\mathrm{ZG}}{1000}$$

Abbildung 20: Änderung des Rewards

Was jedoch, wie in der folgenden Abbildung 21 zu erkennen ist, zu keiner Verbesserung führt.

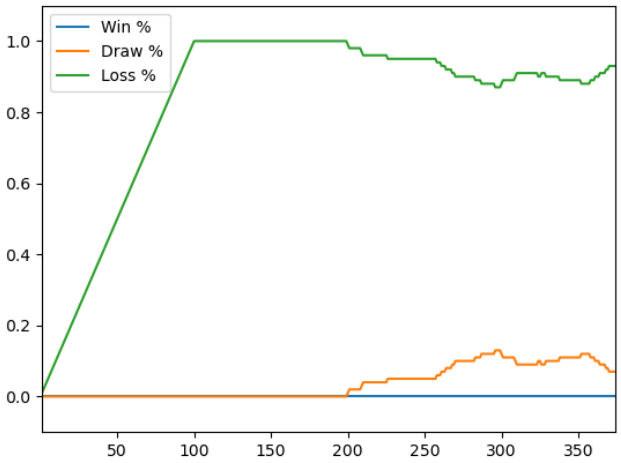


Abbildung 21: Vierter Versuch mit 400 Spielen und siebzehn Choices

Der fünfte Versuch wird auf der großen Karte Simple128 mit den 5 Choices, der neuen Reward-Funktion und MSE als Loss-Funktion durchgeführt. Wie in der Abbildung 22 zu sehen ist, werden bis zum ca. 250. Spiel Fortschritte gemacht. Zwischen dem 250. und 375. Spielen wird sogar wieder etwas verlernt und daher das Training beendet.

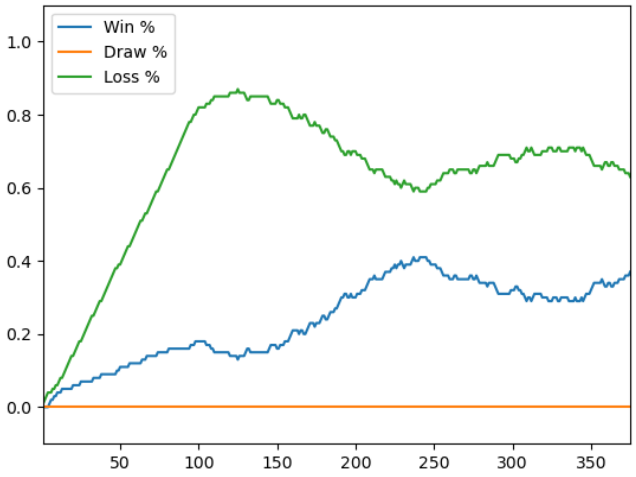


Abbildung 22: fünfter Versuch mit 375 Spielen und siebzehn Choices

Beim sechsten und letzten Versuch wurde die Loss-Funktion auf Categorical Crossentropy geändert. Wie auf in der Abbildung 23 zu erkennen ist, verbessert sich das Model bis zum 450. Spiel und wird ab diesem Zeitpunkt wieder schlechter. Das Training wird daraufhin beendet.

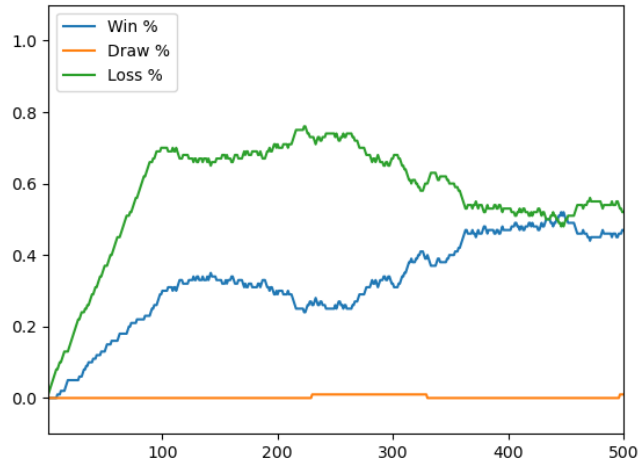


Abbildung 23: Sechster Versuch mit 500 Spielen und siebzehn Choices

Als Ergebnis wird festgehalten, dass

1. das Model schnell an seine Grenze stößt, sobald man die Entscheidungsmöglichkeiten erhöht.
2. das Model bessere Entscheidungen zu treffen lernt, je besser die Reward-Funktion ist (vgl. Versuch 6 und 1).
3. in unserem Fall mit der Loss-Funktion Categorical Crossentropy besser Ergebnisse erzielt wurden als mit MSE.

4.6 Allgemein aufgetretene Probleme

In diesem Abschnitt soll auf die Probleme eingegangen werden, die bei der Implementierung und Testung der beiden Anwendungen aufgetreten sind.
Nach dem ersten Durchlauf mit dem Model sind massive Speicherplatzprobleme aufgetreten. Nach ca. zehn bis zwölf Spielen reichte der Arbeitsspeicher der PCs nicht mehr aus und das laufende Spiel wurde abrupt abgebrochen. Nach intensiver Recherche und mehreren Versuchen, ist es uns gelungen, das Problem in den Griff zu bekommen. Der Fehler war, dass das Model jedes Mal erneut geladen wurde. Das hat Zeit und Speicherplatz gekostet. Die Lösung ist folgende: Beim Spielstart wird das Model nur einmalig hineingeladen. Die Auslastung des Arbeitsspeichers beim anschließenden Ausführen der Spiele mithilfe des Models war zwischen zehn und zwölf Gigabyte groß.
In Abbildung 10 und 12 wird das nächste Problem deutlich – Unentschiedene Spiele, die aufgrund der fehlenden Schnittstelle zum Abfangen dieses Spielergebnisses, lediglich als Niederlagen gewertet wurden.
Ein Void Ray hat einen bestimmten Angriffsradius. In diesem bestimmten Radius greift die Militäreinheit alles an, was sich in ihm befindet. Mithilfe der Funktion „attack known enemy“ kann die Maschine bei fünf Choices ebenfalls Gebäude außerhalb dieser Reichweite angreifen. Diese Entscheidungen trifft er jedoch ohne eine strategische Fallunterscheidung. Um dieses Problem, welches des Öfteren bereits erwähnt wurde, zu bereinigen, sind die 17-Choices entstanden. Mit ihnen ist es dem Model möglich, eine Strategie auszuführen. Es lernt dadurch, bestimmte Gebäudestrukturen vor allen anderen zu zerstören.
Durch die Wahl der kleineren Karte, war es möglich, die Anzahl der Choices zu erhöhen. Feste Koordinaten werden als Choice übergeben. Hierbei sind diese so gewählt, dass sie die komplette Infrastruktur des Gegners aufzeigen kann.
Durch das Erhöhen der Anzahl möglicher Choices, wird allerdings die Komplexität gesteigert. Das Model muss in beiden Fällen aus siebzehn verschiedenen Aktionen die richtige Auswählen. Das führt dazu, dass die Models überfordert sind und demensprechend große Trainingsdaten benötigen.
Ein zusätzliches Problem war der Faktor Zeit. 1000 Spiele zu spielen dauert länger als 24 Stunden. Der Autor des Tutorials, Harrison Kinsley (Vgl. [5]), hat allein für sein Model 1000 gewonnene Daten gesammelt. Um das zu erreichen, müssten die Models tagelang trainiert werden. Allerdings kosteten die Implementierung und Anpassung des Spiels bereits eine Menge Zeit.

5 Vergleich der beiden Anwendungen

In diesem Kapitel werden die Resultate beider angewendeten Methoden des Maschinellen Lernens verglichen.

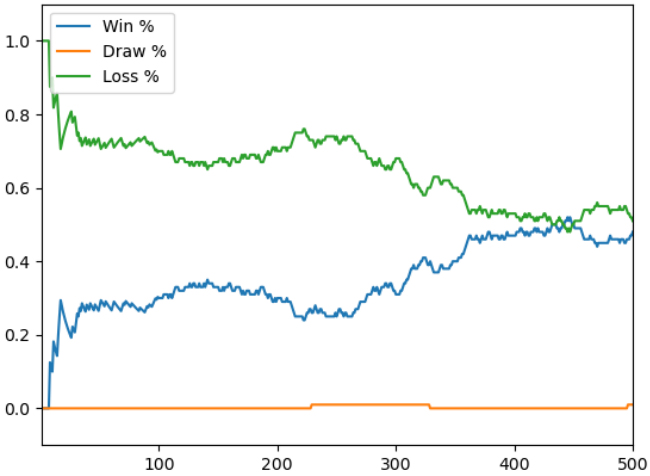


Abbildung 24: Q-Learning

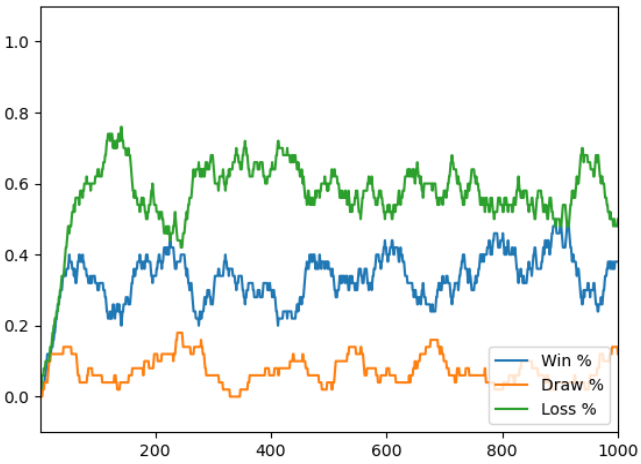


Abbildung 25: Supervised-Learning

In der Abbildung 24 ist das Ergebnis des Q-Learnings (500 Spiele, große Karte) visualisiert. Daneben ist in Abbildung 25 das Resultat mit Supervised Learning (Model_V2, 1000 Spiele, große Karte) zu erkennen. Werden die beiden Graphen miteinander verglichen, wird deutlich, dass das Q-Learning trotz geringerer Anzahl an Spielen ein besseres Ergebnis erzielt als das Supervised Learning. Die Reward Funktion des Q-Learnings lässt sich besser an die Anforderungen anpassen. Das Supervised Learning könnte mithilfe von mehr Trainingsdaten seine Genauigkeit steigern.
Im Folgenden wird der Ausgang der kleineren Karte (64 x 64) mit den siebzehn Choices dargestellt.

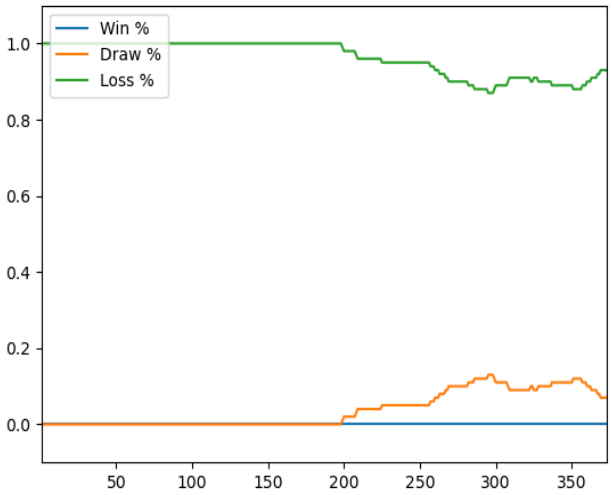


Abbildung 26: Q-Learning

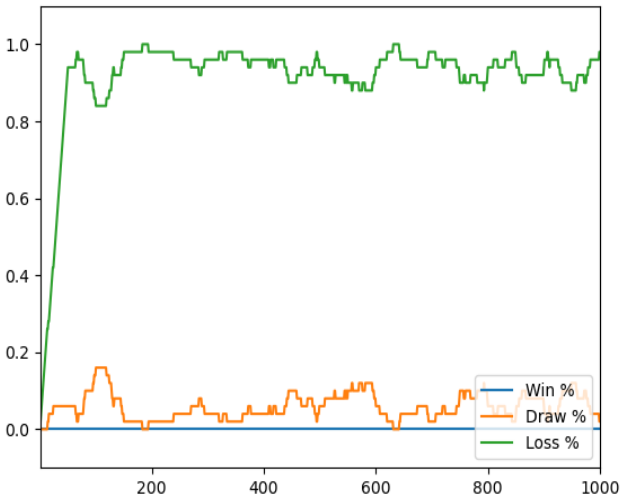


Abb. 27: Supervised-Learning

Abbildung 26 zeigt das Ergebnis des Q-Learnings. Darauffolgend ist das Resultat des Supervised Learnings (Abbildung 27). Bei beiden Verfahren ist das Ergebnis ernüchternd. Bei diesem Vergleich sind beide Arten des ML zu diesem Zeitpunkt nicht zu empfehlen.

5.1 Mögliche Verbesserungen

Allerdings stellte sich während unserer Analysen Potential zur Verbesserung heraus. Zum einen müsste die Scouting Funktion angepasst werden, damit die Militäreinheiten wissen, wo sich weitere Gebäude des Feindes befinden.
Des Weiteren müsste evaluiert werden, ob die Verwendung einer anderen API sinnvoller wäre. Die derzeitige API lässt uns, wie bereits in den vorherigen Kapiteln erklärt, nicht das Aufgeben des Gegners annehmen. Hinzufügend war es schwer, mit der derzeitig genutzten API, genug Daten für den Reward zu sammeln. So konnten einige Ideen nicht umgesetzt werden, da die benötigten Daten nicht über die Schnittstelle verfügbar waren. Die API von Blizzard und DeepMind würde diese Funktion unter Umständen unterstützen.
Die enttäuschenden Resultate in den Abbildungen 14 und 19 könnten verbessert werden, indem beim Supervised Learning die Choices mit Gewichten versehen werden und beim Q-Learning die Reward Funktion dementsprechend angepasst wird. Durch die stärkere Gewichtung einzelner Choices könnte dem Model der richtige Schubser in die richtige Richtung gegeben werden.

6 Ausblick

Diese Studienarbeit hat einen groben Überblick über das Potential des Maschinellen Lernens gegeben. Das Tutorial von Harrison Kinsley ist ein guter Einstieg in die Welt der AML. Zugleich zeigt es aber nur einen kleinen Teil dessen, was zwischenzeitlich mit KIs erreichbar wäre. So ist die Bandbreite der AML viel größer als nur der Einsatz in Spielen. Seit Jahren sind in bestimmten Autos Kameras verbaut, die mithilfe einer KI Verkehrszeichen erkennen sollen. Zudem gibt es zahlreiche Sicherheitssysteme in den Autos, die mithilfe der Künstlichen Intelligenz die Gefahr von Autounfällen reduzieren und den Personen unterstützend zur Seite stehen. (Vgl. [9])
Welche der beiden Arten des Maschinellen Lernens nun besser für das Spiel geeignet ist, ist schwer zu beantworten. Beide Models haben bewiesen, dass sie öfters gewinnen als dies mit bloßem Zufall möglich ist, jedoch steigt in beiden Fällen die Lernkurve nur langsam. Im Grunde genommen kann sowohl mit dem Supervised Learning als auch mit dem Q-Learning gute Resultate erzielt werden. Am Ende ist es wohl eine subjektive Entscheidung welche Art für das Spiel besser geeignet ist.

Quellen

[1] Amelie Smith. Learning Each Function with Machine Learning:. https://medium.com/@ameliasmithml/learning-each-function-with-machine-learning-264dbaae0e20. [Online; Stand 01.08.2020]
[2] Dr. Klaus Manhart. Was Sie über Maschinelles Lernen wissen müssen. https://www.computerwoche.de/a/was-sie-ueber-maschinelles-lernen-wissen-muessen,3329560. [Online; Stand 21.07.2020]
[3] FANDOM. StarCraft Wiki – Obeserver. https://starcraft.fandom.com/wiki/Observer. [Online; Stand 01.08.2020]
[4] Hannes Karppila. BurnySc2/python-sc2. https://github.com/BurnySc2/python-sc2. [Online; Stand 16.07.2020]
[5] Harrison Kinsley. Python AI in StarCraft 2. https://pythonprogramming.net/starcraft-ii-ai-python-sc2-tutorial/. [Online; Stand 16.07.2020]
[6] Henryk Michalewski. Playing Atari on RAM with Deep Q-learning. https://deepsense.ai/playing-atari-on-ram-with-deep-q-learning/. [Online; Stand 27.07.2020]
[7] Julia Fischer, Kevin Pochwyt. Tensorflow und Keras. https://user.phil.hhu.de/~petersen/SoSe17_Teamprojekt/AR/tensorflow.html. [Online; Stand 21.07.2020]
[8] Keon. Deep Q-Learning with Keras and Gym. https://keon.github.io/deep-q-learning/. [Online; Stand 27.07.2020]
[9] Klaus Manhart. Deep Learning – Königsdisziplin der Künstlichen Intelligenz. https://www.com-magazin.de/praxis/kuenstliche-intelligenz-ki/deep-learning-koenigsdisziplin-kuenstlichen-intelligenz-1663953.html. [Online; Stand 23.07.2020]
[10] liquipedia, 2019. Protoss Units(Legacy of the Void) https://liquipedia.net/starcraft2/Protoss_Units_(Legacy_of_the_Void). [Online; Stand 21.07.2020]
[11] Official Keras Documentation. Deep lerning for humans. https://keras.io/. [Online; Stand 21.07.2020]
[12] Official TensorFlow Documentation. Why TensorFlow. https://www.tensorflow.org/. [Online; Stand 21.07.2020]
[13] Oriol Vinyals, Stephen Gaffney, Timo Ewalds. DeepMind and Blizzard open StarCraft II as an AI research environment. https://deepmind.com/blog/announcements/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment. [Online; Stand 16.07.2020]
[14] Roland Becker. CONVOLUTIONAL NEURAL NETWORKS – AUFBAU, FUNKTION UND ANWENDUNGSGEBIETE. https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/. [Online; Stand 21.07.2020]
[15] Wikipedia. StarCraft 2. https://de.wikipedia.org/wiki/StarCraft_II. [Online; Stand 16.07.2020]

Reinforcement Learning – Eine videospielbasierte Umsetzung des Algorithmus

Burak Civak
Hochschule für angewandte Wissenschaften in Hof
Medieninformatik
burak.civak@hof-university.de

I. Einleitung

In folgender Studienarbeit wird erläutert, wie eine Künstliche Intelligenz selbstständig, durch Trainieren von Neuronalen Netzen, ein Spiel effizient löst. Inspiration zu diesem Thema waren vor allem die Videos von dem YouTuber „CodeBullet“.[1] Diese Arbeit wird sich darauf konzentrieren, ein selbst erstelltes Spiel effizient mit Hilfe des NEAT Algorithmus zu lösen.

II. Alto's Adventure

Das Handyspiel „Alto‘s Adventure“ ist von den Publishern „Noodlecake Studios Inc“ und ist ein 2D Side Scroller. Der Spieler selbst hat nur begrenzte Interaktionsmöglichkeiten, hierbei zählen das normale Tappen auf den Bildschirm für einen einfachen Sprung und das gedrückt Halten des Tapps für einen Salto. Die Spielfigur bewegt sich in der Spielumgebung vollkommen automatisch. Das Ziel des Spielers ist es möglichst weit im Level fortzuschreiten, um eine hohe Punktzahl zu erreichen. Dabei können noch Münzen oder Lamas während des Laufes gesammelt werden, um die Punktzahl zu erhöhen. Im Spiel selbst existieren vollkommen dynamische Licht- und Wettereffekte, wie beispielsweise Gewitter, Blizzards, Nebel, Regenbögen, Sternschnuppen und mehr. [2]

Die eben genannten Punkte haben auch zu Problemen geführt, weswegen sich für diese Studienarbeit dafür entschieden wurde, das gleiche Spiele vereinfacht selbst zu erstellen. Hierbei wurden folgende Sachen beachtet: Der Spieler hat nur eine Interaktionsmöglichkeit, es handelt sich hierbei um das Springen. Die Interaktionsmöglichkeit des Saltos wurde bei der Implementierung nicht berücksichtigt. Ebenfalls wurde das Einsammeln von Münzen und Lamas, um die Punktzahl zu erhöhen, nicht implementiert. Des Weiteren wurden die Objekte und Klippen, wie vom originalen Spiel übernommen. Das dynamische Wetter wurde hierbei auch weggelassen.

III. Ziel

Das Ziel der Studienarbeit ist es, dass die Künstliche Intelligenz, mithilfe des NEAT Algorithmus so trainiert wird, dass es selbst in der Spielumgebung so weit wie möglich im Level voranschreitet. Hierbei sollte die KI selbst entscheiden können, wann die Spielfigur springt und mit welcher Geschwindigkeit diese sich fortbewegen soll.

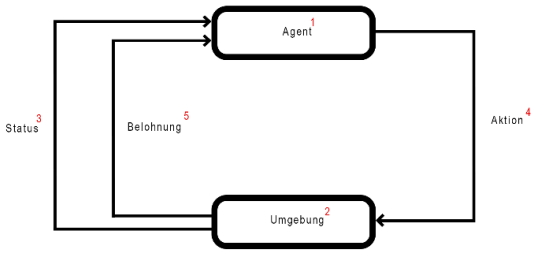
IV. Allgemeine Theorie zu den Verfahren

A. Reinforcement Learning

Reinforcement Learning, wörtlich übersetzt „Bestätigendes Lernen“, ist, neben Supervised und Unsupervised Learning, die dritte große Gruppe von Machine Learning Verfahren. Nachfolgend wird Reinforcement Learning mit RL abgekürzt. RL ist eine Methode, die sich am menschliche Verhalten orientiert. Hierbei erfolgt das menschliche Lernen vor allem in den frühen Stadien des Lebens durch Herantasten und Erkunden der Umwelt. Dies wird auch „Trial-and-Error“ genannt. Man versucht etwas und bekommt entweder ein positives (belohnendes) oder negatives (bestrafendes) Feedback.

Hierfür ein Beispiel: Auch wenn Reinforcement Learning Analogien zum Menschen darstellt, wird hierfür ein Beispiel von der Tiererziehung erläutert. Ein Welpe versucht, wenn er eine neue Umgebung sieht, sich erst langsam heranzutasten, indem er bestimmte Aktionen ausführt, welche dann wiederum Reaktionen des Herrchens hervorrufen. Bespeilsweise, wenn ein Welpe auf Toilette muss, dann wird er am Anfang nicht direkt verstehen, dass er dafür vor die Haustür muss, sondern kann auch am Anfang sein Bedürfnis an ungelegen Orten wie im Flur ausführen. Dies würde dazu führen, dass der Besitzer verärgert ist und er ihn bestraft. Durch eine Belohnung, die der Welpe erhält, beim Gassi gehen sein Bedürfnis zu erledigen, würde der Welpe verstehen und auch lernen, dass es nur beim Gassi gehen sein Bedürfnis erledigen darf.

In RL gibt es fünf wichtige Komponenten: Den Agenten (1), die Umgebung (2), den Status (3), die Aktion (4) und die Belohnung (5). Siehe Abbildung 1.



Reinforcement Learning Schaubild *Abbildung 1*

Der Ablauf ist wie folgt: Der Agent, eine Person oder eine Spielfigur, führt zu einem bestimmten Status, eine der verfügbaren Aktionen durch, welche dann zu einer Reaktion der Umgebung in Form einer Belohnung oder Bestrafung darstellt.

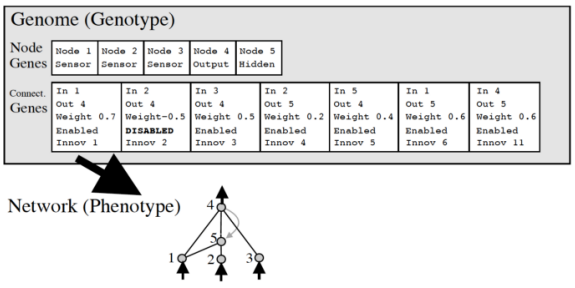
Die dementsprechende Reaktion der Umgebung auf die getätigten Aktionen des Agenten beeinflussen wiederum die Wahl der Aktion des Agenten im nächsten Status. Hierbei ist der Agent in der Lage mehrere Durchläufe zu erstellen, um einen Zusammenhang herzustellen und anschließend zu optimieren. Dabei befindet sich der Agent in dem sogenannten „Exploration-Exploitation Dilemma“. Dieses Dilemma bezeichnet auf der einen Seite die Nutzung seiner bisher erworbenen Erfahrungen und auf der anderen Seite die Exploration neuer Strategien zur Erhöhung der Belohnung.

B. NEAT Verfahren

NEAT (NeuroEvoloution of Augmenting Topologies) ist ein Algorithmus, welcher sich darauf fokussiert das Neuronale Netz von Knoten zu Knoten und Verbindung zu Verbindung zu lösen. Die Idee hierbei wurde von Kenneth O. Stanley und Risto Miikkulainen 2002 als Paper veröffentlicht. NEAT ist ein genetischer Algorithmus, welcher sich die allgemeine Biologie zu nutzen macht. Das Wichtigste hierbei sind die Prozesse, die von der Biologie hergeleitet werden können: Selection, Mutation und Crossover.

Der NEAT Algorithmus benutzt eine direkte Enkodierung, da jedes Element im Neuronales Netz direkt mit dem dazugehörigen Gegenstück vernetzt ist. In der Abbildung 2 (vom Originalpaper entnommen), ist die direkte Vernetzung des Neuronalen Netzes zu erkennen. Inputs und Outputs sind nicht entwickelt in der Gen Liste. Versteckte Knoten (Hidden Nodes) können hinzugefügt oder entfernt werden. Die Verbindungsknoten sind für die Spezifizierung der Richtung, der Gewichtung und des Biases zuständig.

Wichtig bei NEAT ist die sogenannte Mutation, diese kann existierende oder neue Strukturen im Netzwerk mit zufälligen Gewichten hinzufügen oder ändern. Wenn ein neuer Knoten hinzugefügt werden sollte, wird dieser zwischen zwei bereits existierenden Knoten eingefügt. Die Vorherige Verbindung ist zwar noch da, aber sie funktioniert nicht mehr. Der vorherige Startknoten ist verbunden mit dem neuen Knoten mit den Gewichten der alten Generation und der neue Knoten ist verbunden mit dem vorherigen Endknoten mit einem Gewicht von 1.



Grafik aus dem Original Paper *Abbildung 2*

Eine weitere wichtige Funktion bei NEAT ist das sogenannte Crossover. Die Idee hierbei ist das Kreuzen von Netzwerken. Hierbei sollte beachtet werden, dass sich nicht einfach zwei zufällige Netzwerke miteinander kreuzen, weil dies zu unvorhergesehenen Ergebnissen, wie zum Beispiel Informationsverlust, führen kann. NEAT löst dieses Problem mithilfe der Analogie zu Biologie. Jedes Mal, wenn ein neuer Knoten oder ein neuer Typ von Verbindung auftaucht, wird dieser markiert, um so die einzelnen Strukturen zu sortieren.

Als letztes kommt die Selection, welche den Zweck hat, die Besten einer Generation zu finden und diese dann in die neue Generation einfließen zu lassen.

C. Unterschied zwischen allgemeinen RL und NEAT

NEAT ist ein genetischer Algorithmus welcher künstliche Neuronale Netzwerke weiterentwickelt. Es ist eine Neuro-Evolutionäre Technik, welche die einzelnen Gewichte und Strukturen in den Netzwerken je nach Fitnesswert verbessert, umso bessere Lösungen zu finden. Der Fitnesswert ist der Wert, der jedem Netzwerk nach einem Durchlauf gegeben wird, um so die Netzwerke nach Ihrer Leistung zu sortieren.

In RL hingegen geht es um Agenten, die verschiedene Richtlinien lernen, um sich besser in der Umgebung zurechtzufinden, in der sie eingesetzt werden. RL löst vor allem komplexere Probleme. Theoretisch könnte man RL benutzen, um NEAT zu lernen, mit z. B. übergeben eines Neuronalen Netzes als State, damit es lernt, wie man es modifiziert, um bessere Ergebnisse zu erzielen.

V. Vorgehensweise

A. Das Spiel

Das Spiel soll die Anforderung an das Original halten können, das heißt es soll möglichst nah am Original bleiben und die Schlüsselelemente dessen beinhalten. Das Spiel wurde mit der Unity Engine und deren Programmiersprache C# erstellt. Wie schon vorher erwähnt, ist das Original Spiel ein 2D Side Scroller, das heißt beim Erstellen der Karte muss eine zweidimensionale Betrachtung gegeben sein.

Der erste Schritt sollte vor allem für die zufällige Generierung der Karte dienen, hierbei sollte beachtet werden, das das Spiel nicht eine feste Größe hat, sondern dass die Karte unendlich lange nachgeneriert werden soll. Das Erstellen der Karte wurde mit Hilfe eines selbst programmierten Level Generator erstellt. Der Level Generator generiert immer am Ende des jetzigen Abschnittes einen neuen, zufälligen vordefinierten, Abschnitt. Wichtig zu beachten, dass der Abschnitt nicht dauerhaft erstellt wird, sondern erst wenn der Spieler in einem bestimmten Abstand zum Ende des Abschnittes gelangt.

Einfachheitshalber wurde nicht berücksichtigt, dass man die Abschnitte wieder entfernen sollte und es wurde so gehandhabt, dass die generierten Abschnitte erst dann wieder gelöscht werden, wenn das Spiel komplett neu startet und nicht wenn der Spieler sterben sollte. Des Weiteren wurde die Generierung so gestaltet, dass es zufällig ist welcher Abschnitt als nächstes kommt, wobei garantiert ist, dass derselbe Abschnitt nicht zwei Mal hintereinander auftauchen wird.

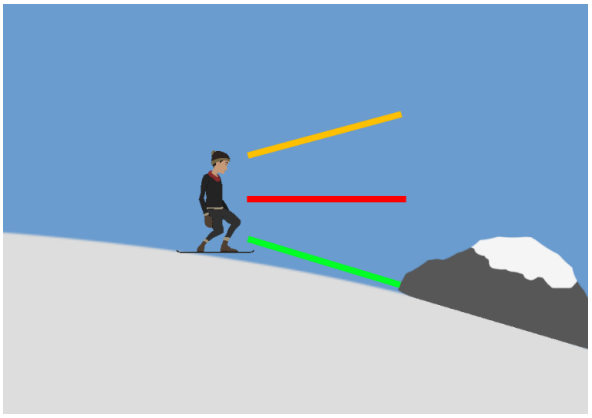
Nachdem das erledigt wurde, hat man sich auf den Spieler selbst konzentriert. Dieser sollte sich nur auf zwei Sachen konzentrieren: erstens die Geschwindelt die er fährt und zweitens die Distanz, ab wann er springen sollte. Beides sollte mit Hilfe des Neuronalen Netzes kontrolliert werden, um so einen Lernerfolg mit dem genetischen Algorithmus erzielen zu können.

Weiterhin wurde im Spieler Skript der Fitnesswert berechnet, welcher aus drei Bestandteilen besteht: der erste Teil ist die zurückgelegte Distanz des Spielers. Dieser Wert sollte die höchste Gewichtung bei der Bewertung haben, da man als Ziel hat eine hohe Distanz zurückzulegen. Der zweite Bestandteil war die Zeit, in der der Spieler die Strecke zurücklegt, um so verhindern zu können, dass langsame Spieler bei gleicher Distanz belohnt werden. Als letzter Bestandteil wurde der Wert der Abstand zu dem Objekt, indem Fall waren es Steine oder die Klippe, mitbewertet. Dies zusammen ergibt dann den Fitnesswert, welches jedes Neuronale Netz am Ende erhält.

```
Fitnesswert =   ( totalDistance *1,4 ) +
                ( avgSpeed * 0,2 ) +
                (( abstandA + abstandB + abstandC)/3
                * 0,1 ))
```

B. Neuronales Netzwerk

Beim Neuronalen Netz sollten verschiedene Punkte beim Aufbauen beachtet werden. In einem neuronalen Netz sollten vor allem die Inputs und Outputs genau bestimmt werden, um das bestmögliche Ergebnis erzielen zu können. Da beim Programmieren der Spielfigur schon darauf geachtet wurde, waren die Inputs und Outputs schnell bestimmt. Bei den Inputs wurden hierbei der Abstand zu dem Objekt gewählt, die man während des Spiels immer wieder trackt (siehe Abbildung 3). Hierbei hätte auch nur ein Sensor gereicht, aber drei Sensoren bieten ein größeren Winkel, um Objekte besser zu erkennen. Genau wie die Inputs waren die Outputs auch schon vorgegeben. Der Spieler sollte entscheiden können in welcher Geschwindigkeit er fährt (Output 1) und ab wann er springen soll, heißt wie weit ist er noch zum Objekt entfernt ist (Output 2). Für die Aktivierungsfunktion wurde hierbei Sigmoid verwendet, da die Werte nur zwischen 0 und 1 liegen sollen.



Visuelle Darstellung der Sensoren — *Abbildung 3*

C. Genetischer Algorithmus

Hier kommt der Hauptteil der Studienarbeit ins Spiel, die künstliche Intelligenz, welche mithilfe des genetischen Algorithmus lernen soll. Wie schon oben erwähnt, gibt es einige Sachen zu beachten. Als Erstes sollte die Funktion damit beginnen eine Anfangspopulation zu generieren. Hierbei ist jedes Mitglied in der Population ein Netzwerk mit Hiddenlayers und Hiddenneurons und komplett zufälligen Weights und Biases. Als nächstes kümmert man sich um die neue Generation, hierbei gibt es viele verschiedene Funktionen, die zum Einsatz kommen. Zuallererst sortiert man mit Hilfe eines einfachen Algorithmus (hier wurde Bubble Sort verwendet), die jetzige Population der Reihe nach dem Fitnesswert. Danach wählt man die besten der jetzigen Generation aus, um diese der neuen Generation zuzufügen. Anschließend kommen die zwei wichtigen Funktionen Crossover und Mutation.

Beim Crossover geht es darum mithilfe der besten Netzwerke weitere Mitglieder der neuen Generation hinzuzufügen, indem zufällige Elternpaare gebildet werden und daraus dann neue Kinder (Netzwerke) entstehen. Bei der Mutation hingegen geht es darum, dass sich einzelne Netzwerke mutieren können, um somit ein neues zufälliges Netzwerk mit komplett neuen zufälligen Weights und Biases zu erschaffen. Der Grund hierfür ist einfach zu verstehen, es kann sein, dass in der ersten Generation Netzwerke existieren, die zu nichts zu gebrauchen sind, heißt deren Fitnesswerte sagt nichts Besonderes aus. Mit der Mutation kann man mit geringer Wahrscheinlichkeit dieses Risiko verringern, indem man, wie eben erwähnt, neue zufällige Netzwerke generiert. Die Mutation muss aber auch nicht bei jeder Generation passieren, es passiert, wie eben erwähnt, nur mit einer sehr geringen Wahrscheinlichkeit (in diesem Fall mit 0,055%).Nach dem dies alles geschehen ist, werden die restliche Plätze der neuen Generation mit zufälligen Netzwerken gefüllt. Und dieser Prozess wird solange ausgeführt, bis man selbst mit dem Ergebnis des Algorithmus zufrieden ist.

VI. Code Beispiele

A. Level Generation

Im Skript Level Generator befindet sich die Funktion „SpawnLevelPart()“ (siehe Abbildung 4). Die Funktion hat mehrere Aufgaben, die es erfüllen muss, damit die Generation des Levels kompakt abläuft. In der do-while-Schleife wird mit Hilfe von zufälligen Zahlen ein vordefinierter Part entnommen, welches wiederum in einem Array gespeichert ist. Dies geschieht aber nur solange, bis ein neuer Part gefunden wurde und nicht mehr dem alten Part entspricht. Im Anschluss wird der neue Part als neuer End Part gespeichert. Daraufhin folgt eine neue Funktion, welche dann den neu gewählten Part an der Endposition des jetzigen Parts spawnt.

```
29    private void SpawnLevelPart()
30    {
31        int chosenLevelPartIndex;
32
33        do
34        {
35            chosenLevelPartIndex = Random.Range( 0, levelPartList.Count );
36        }
37        while( chosenLevelPartIndex == this.lastGeneratedPartIndex );
38
39        this.lastGeneratedPartIndex = chosenLevelPartIndex;
40
41        Transform chosenLevelPart = levelPartList[chosenLevelPartIndex];
42        Transform lastLevelPartTransform = SpawnLevelPart(chosenLevelPart, lastEndPosition);
43        lastEndPosition = lastLevelPartTransform.Find("EndPosition").position;
44    }
```

Code für SpawnLevelPart() — *Abbildung 4*

B. Crossover

Die Abbildung 5 stellt die Implementierung der Crossover Funktion dar. Diese Funktion ist mit eine der wichtigsten Funktionen, wenn es um den genetischen Algorithmus geht. Die Funktion ist wie folgt aufgebaut: zuallererst wird die Funktion mit einer for-Schleife initialisiert, welche solange durchläuft, wie oft sich Paare kreuzen sollen. Diese Zahl wird am Start festgelegt. Nach dem zwei Variablen für die Eltern angelegt wurden, werden zwei Kinder (indem Fall zwei neue Neuronale Netze) angelegt. Diese Kinder werden dann mit blanken Werten neu initialisiert. Anschließend kommt die Funktion „GetRandomParents()“ ins Spiel (siehe Abbildung 6).

```
private void Crossover()
{
    for( int i = 0; i < nrCrossover; i += 2 )
    {
        NNet parentA, parentB;

        NNet Child1 = new NNet();
        NNet Child2 = new NNet();

        Child1.Initialise( controller.LAYERS, controller.NEURONS );
        Child2.Initialise( controller.LAYERS, controller.NEURONS );

        GetRandomParents( out parentA, out parentB );

        for( int w = 0; w < Child1.weights.Count; w++ )
        {
            if( Random.Range( 0.0f, 1.0f ) < 0.5f )
            {
                Child1.weights[w] = parentA.weights[w];
                Child2.weights[w] = parentB.weights[w];
            }
            else
            {
                Child2.weights[w] = parentA.weights[w];
                Child1.weights[w] = parentB.weights[w];
            }
        }

        for( int b = 0; b < Child1.biases.Count; b++ )
        {
            if( Random.Range( 0.0f, 1.0f ) < 0.5f )
            {
                Child1.biases[b] = parentA.biases[b];
                Child2.biases[b] = parentB.biases[b];
            }
            else
            {
                Child2.biases[b] = parentA.biases[b];
                Child1.biases[b] = parentB.biases[b];
            }
        }

        SetNextChild( Child1 );
        SetNextChild( Child2 );
    }
}
```

Code für Crossover() *Abbildung 5*

```
private void GetRandomParents( out NNet ParentA, out NNet ParentB )
{
    int parentAIndex = 0;
    int parentBIndex = 0;

    parentAIndex = genePool[Random.Range( 0, genePool.Count )];
    do
    {
        parentBIndex = this.genePool[Random.Range( 0, this.genePool.Count )];
    }
    while( parentAIndex == parentBIndex );

    ParentA = this.population[parentAIndex];
    ParentB = this.population[parentBIndex];
}
```

Code für GetRandomParents() *Abbildung 6*

Diese Funktion sucht zwei Elternpaare heraus, indem es sich aus den besten Netzen zufällig welche herausnimmt. Die besten Netze wurden schon mit Hilfe einer anderen Funktion vorsortiert und in die neue Population hinein gespeichert. Dabei wird in der do-while-Schleife beachtet, dass sich beide Eltern Paare voneinander unterscheiden. Anschließend werden dann die ausgesuchten Elternpaare der Crossover Funktion zurück übergeben.

Nachdem ein Elternpaar gefunden wurde, werden den Kindern zufällig deren Gewichte und Biases übergeben. Dies geschieht mit jeweils einer for-Schleife. Dabei wird zufällig entscheiden, welches Kind welche Weights oder Biases bekommt. Nachdem beiden Kindern Biases und Weights zugeteilt wurden, werden diese dann mit Hilfe der Funktion „SetNextChild()“ der neuen Generation hinzugefügt.

C. Mutation

Mit der Abbildung 7 wird der Code von der Implementierung der Mutation Funktion dargestellt. Diese Funktion dient dazu, schon vorhandene Netzwerke mutieren zu lassen. Zuallererst beginnt auch diese Funktion mit einer for-Schleife, welche so oft durch iteriert wie viele Plätze schon in der neuen Generation belegt sind. Danach wird eine Variable angelegt, um das aktuelle Netzwerk darin zu speichern. In der darauffolgenden for-Schleife wird so oft iteriert, wie groß die Anzahl der Gewichte des aktuellen Netzwerks ist. Dann wird zufällig entschieden, ob das aktuelle Netzwerk mutiert oder es so bestehen soll. Dabei wurde beachtet, dass diese Mutationswahrscheinlichkeit sehr gering ist. Falls dieses Netzwerk mutiert, werden mit Hilfe der anschließenden for-Schleife, alle Werte des aktuellen Netzwerks zufällig neu gesetzt.

```
private void Mutate()
{
    for( int iSetChilds = 0; iSetChilds < this.iNextFreeChild; iSetChilds++ )
    {
        NNet setChild = this.population[iSetChilds];

        for( int iWeight = 0; iWeight < setChild.weights.Count; iWeight++ )
        {
            if( Random.Range( 0.0f, 1.0f ) < mutationRate )
            {
                Matrix<float> weights = setChild.weights[iWeight];

                int randomPoints = Random.Range( 1, (weights.RowCount * weights.ColumnCount) / 7 );

                for( int i = 0; i < randomPoints; i++ )
                {
                    int randomColumn = Random.Range(0, weights.ColumnCount);
                    int randomRow = Random.Range(0, weights.RowCount);

                    weights[randomRow, randomColumn] = Mathf.Clamp( weights[randomRow, randomColumn] + Random.Range( -1f, 1f ), -1f, 1f );
                }
            }
        }
    }
}
```

Code für Mutate() *Abbildung 7*

VII. Ergebnisse

In der ersten Generation kann man kaum richtige Ergebnisse betrachten. Da diese Generation komplett zufällig generiert ist, sollte man für genauere Ergebnisse die zukünftigen Generationen betrachten. Der Spielscreen (siehe Abbildung 8) zeigt hier folgende Punkte auf: Oben Links befindet sich die aktuelle Distanz, die die Spielfigur zurückgelegt hat. Darunter befindet sich der Fitnesswert, die erste Zahl beschreibt den Durchschnitt des Fitnesswerts aller bisherigen Genome (Spieler/Netzwerke) in der jetzigen Generation. Die Zahl in der Klammer steht für den Delta zwischen dem Durchschnitt der letzten Generation und der ersten Zahl. Ansonsten befindet sich noch die aktuelle Generation und das aktuelle Genom oben rechts im Spielscreen. In der Mitte kann das Spieltempo erhöht werden, um so zu einem schnelleren Ergebnis zu gelangen.

Schon ab der elften Generation, konnte man genauere Ergebnisse erkennen (siehe Abbildung 9). Der Fitnesswert hat sich stetig verbessert und die Genome sind immer weiter im Level vorangeschritten. Eine wichtige Sache ist, dass in diesem Spiel kein Genom ein höheren Fitnesswert als 10000 haben kann und somit das Genom, wenn es diesen erreichen sollte vernichtet wird. Dieses Genom wird dann auch nicht mehr für die zukünftigen Generation betrachtet. Auch

Genome mit einem sehr geringen Fitnesswert werden nicht weiter betrachtet.

InGame-Ansicht *Abbildung 8*

Verbesserter Fitnesswert *Abbildung 9*

VIII. Fazit

In dieser Studienarbeit war das Ziel mithilfe des genetischen Algorithmus, eine Künstliche Intelligenz zu trainieren, damit es das vorgegebene Problem löst. Dies wurde auch sehr schnell, von der KI absolviert. Auch wenn hierbei die Levelgeneration zufällig war, konnte die KI das Problem fast immer zur gleichen Zeit lösen.

[1] Code Bullet, zuletzt Abgerufen 02.08.2020 von https://www.youtube.com/c/CodeBullet/

[2] Snowman, zuletzt Abgerufen 02.08.2020 von http://builtbysnowman.com/press/sheet.php?p=altos_adventure

[3] Laurenz Wuttke, zuletzt Abgerufen 02.08.2020 von https://datasolut.com/reinforcement-learning/

[4] Sebastian Heinz (09. Januar 2018), zuletzt Abgerufen 02.08.2020 von https://www.statworx.com/de/blog/einfuehrung-in-reinforcement-learning-wenn-maschinen-wie-menschen-lernen/

[5] AT Redaktion (11. April 2019), zuletzt Abgerufen 02.08.2020 von https://www.alexanderthamm.com/ /blog/einfach-erklaert-so-funktioniert-reinforcement-learning/

[6] Hunter Heidenreich (04. Januar 2019), zuletzt Abgerufen 02.08.2020 von https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

[7] NEAT-Python, zuletzt Abgerufen 02.08.2020 von https://neat-python.readthedocs.io/en/latest/neat_overview.html

[8] AJTech2002, zuletzt Abgerufen 02.08.2020 von https://github.com/AJTech2002/Self-Driving-Car-Series

Reinforcement Learning am Beispiel Tetris

Daniel Kolep
Fakultät Informatik
Hochschule Hof
Hof Bayern Deutschland
daniel.kolep@hof-university.de

***Abstract*— Der Fokus dieser Arbeit liegt darin erste Erfahrungen mit künstlichen Intelligenzen, im Genaueren das Verstärkende Lernen, zu sammeln. Dafür werden die Grundlagen der Methodik an dem Spiel Tetris angewendetet. Das Ziel ist es das Spiel des Agenten, wie das eines menschlichen Spielers wirken zu lassen, wenn nicht sogar das eines Profi Tetrisspieler zu adaptieren. (*Abstract*)**

I. Einleitung

A. Tetris

Tetris wurde im Jahr 1985 von Alexey Pajitnov programmiert. Pajitnov war bekannt für seine Puzzle-Spiel Enthusiasmus bekannt und entwickelte sein Lieblingsspiel, Pentominoes, als Programm, heute bekannt als Tetris.(vgl.[1])

In dem Spiel fallen Puzzleteile von Oben nach Unten herab. Durch Verschieben und Rotieren sollen möglichst viele Steine auf das Spielfeld. Ist eine Reihe gefüllt, so lösen sich die Steine auf und es wird gepunktet. Ist das Spielfeld überfüllt, ist das Spiel verloren.

B. Konzept Reinforcement Learning

Reinforcement Learning wird dem natürlichen Lernverhalten des Menschen nachempfunden. Dabei sind unsere Handlungen im Rahmen des Lernproblems durch einen gewissen Aktionsraum definiert. Über "Trial and Error" werden die Auswirkungen verschiedener Handlungen auf unsere Umwelt beobachtet und bewertet. Als Reaktion auf unsere Handlungen erhalten wir von unserer Umgebung ein Feedback, abstrakt dargestellt in Form einer Belohnung oder Bestrafung. (vgl.[2])

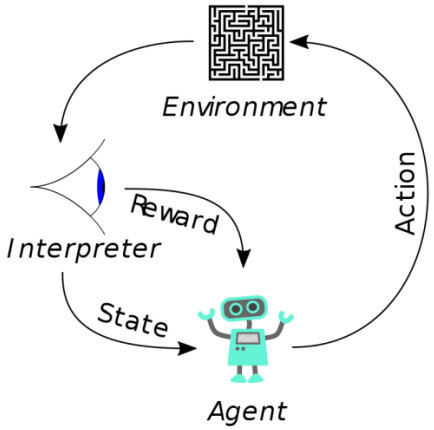


Schaubild für Reinforcement Learning[5] Abbildung 1

Reinforcement Learning wendet die Handlung aus der menschlichen Welt in der Digitalen Welt an. Dabei funktioniert das Zusammenspiel so, dass die Maschine einen State hat und auf diese hin eine Aktion ausführt, die Einfluss auf das Environment hat und den State verändert. Die Veränderung wird von einem Interpreter beobachtent und entsprechend belohnt. Die Reaktion nimmt wiederum Einfluss auf die nächsten Aktionen des Agenten. Das ganze kann für mehrere tausende Durchläufe gemacht werden bis der Agent eine langfristige Strategie für die Zukunft entwickelt. Wobei im Konflikt steht sich auf seine Gesammelte Erfahrung zu verlassen oder weiter zu Experementieren mit neuen Actions und einer daraus folgendem besseren Strategie und größeren Belohnung. Auch bekannt als "Exploration-Exploitation Dilemma".(vgl[2])

C. Q-learning

Q-Learning ist eine Methodik des Reinforcement Learning, welche die beste Action für die Aktuelle Situation findet. Dabei versucht Q-Learning eine Strategie mit dem maximalen Reward zu finden. Das Q steht dabei für Quality, was eine Aussage für die Actionen trifft, die in der Zukunft den besten Reward bringen.

Mit Q-Learning wird zuerst ein Q-Table oder auch eine Matrix erstellt. In diese Matrix wird zu jedem State mit einer Action den dazugehörige Reward gespeichert und diese als Referenz für den Agenten genutzt, um das bestmöglich Ergebniss zu erzielen. Als Nächstes folgt die Interaktion des Agenten mit dem Umfeld für diese es wieder 2 Wege gibt. Zum einen kann sich der Agent auf die Erfahrungen aus seinen bisherigen Q-Table verlassen oder er experementiert mit den Actions, um eine neue Strategie zu finden. Meistens wird ein Verhältnis dieses Dilemmas mit einem Epsilon Wert festgehalten der anfangs ziemlich hoch ist, da man ja noch keine Werte im Q-table hat und diesen mit der Zeit geringer lassen werden kann. Epsilon dient hier nur als Wahrscheinlichkeit wie zum Beispiel 0.9 was die Wahrscheinlichkeit von 90 % zufälligen Actions entspricht. Der Q-table wird mit jedem Schritt aktualisiert und endet mit einer Episode. Endet bedeutet, dass ein gewünschtes Ziel erreicht wurde oder man hat einen Checkpoint erreicht oder wie im Fall Tetris das Spiel wurde verloren. Da es viel zu viele States in Tetris gibt, empfiehlt es sich die Architektur eines Convolutional Neural Network als Policy-Funktion zu verwenden.(vgl[2],[3])

II. Umsetzung

A. Entwicklungumgebung

Das Projekt wurde in Python 3 umgesetzt. Pythongames bietet nützliche Funktionen zur Entwicklung eines Spiels und wird auch in diesem Projekt genutzt. Ein geeignetes Framework zur Entwicklung von Reinforcemente Learning in Python ist Tensorflow, welches für das Projekt installiert wurde.

Die Logik wurde in Python umgesetzt mithilfe eines YouTube Tutorials[5] und später mehrere Male angepasst.

B. Umsetzung in Python

Da das Spiel vorhanden und Menschlich spielbar ist, gilt es nun die Logik für eine Maschine umzusetzen. Dafür wurden die möglichen Actions in eine Methode umgesetzt, die ein Zahlenwert von eins bis vier erwartet. Die einzelnen Zahlen stehen jeweils für eine Action:

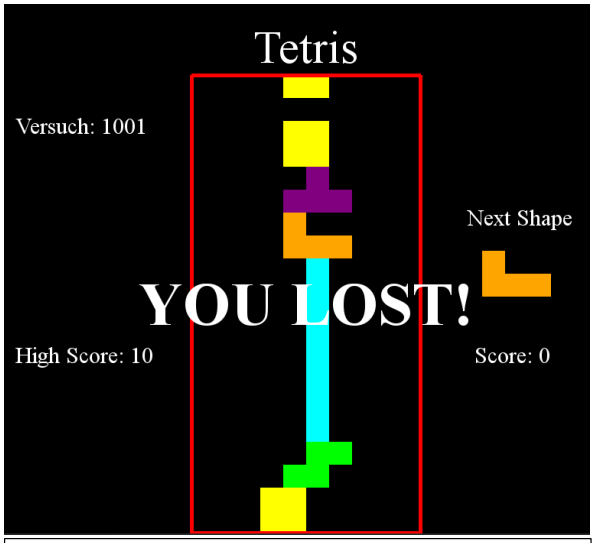
- Eins Stein nach Links bewegen
- Zwei Stein nach Rechts bewegen
- Drei Stein rotieren lassen.
- Vier Stein fallen lassen.

Danach gilt es den Agenten zu erstellen der, diese Zahlenwerte ermittelt. Dafür wird eine if-Schleife benötigt die entscheidet, ob nun eine zufällige Action folgt oder sich auf die Daten des Q-Tables verlassen wird. Die Bedingung für die if-Schleife ist die Abfrage, ob eine zufällige Zahl von 0 bis 1 kleiner als mein angegebener Epsilon Wert ist. Wenn das zutrifft, wird eine Action ausgeführt in der eine Zahl von Eins bis Vier durch die Methode zufällig bestimmt wird. Ansonsten gebe ich eine Prediction mit.

Die States sind durch die liegenden Steine und den fallenden Stein definiert. Um diese States zu übergeben, wird ein Array mit 20x10 Nullen initialisiert, was dem Spielfeld Entspricht. Sobald ein Stein sich im Spielfeld befindet, wird die Position des Steins anstelle der Nullen mit Eins aufgefüllt. So wird ein Input aufgebaut, was dem Tetrisspiel entspricht, bloß, dass das Spielfeld nicht mehr mit Farbe und Steinen visualisiert, sondern nur mit Nullen und Einsen darstellt. Das Q-Table muss aufgefüllt werden, allerdings steht noch in Raum, wann das dann geschehen soll. Dafür gab es 2 Optionen zum einen könnte man den Agenten schon während des Spielen Trainieren, zum anderen könnte man dies am Ende jedes Spiels machen. Die Entscheidung fiel für das trainieren am Ende eines Spiels, wofür die jeweiligen Actions und States in einem Array gespeichert warden, um danach den Reward zu berechnen. Für den Reward ging ich die einzelnen States und deren Actions in einer For-Schleife durch und definierte den Reward mit gescorten Ergebnissen, die Zeit wie lange der Agent durchhielt und einen Punishment der alle Leeren Felder in einem State bestrafte.

III. Ergebnisse

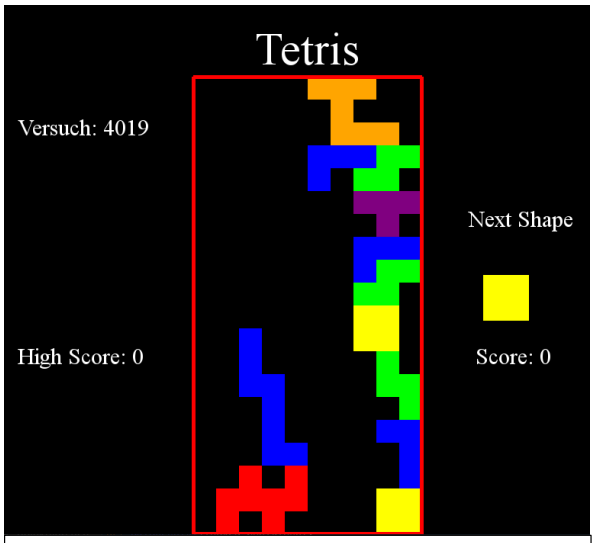
A. Version 1



Screenshoot aus dem Spiel *Abbildung 2*

Nach 1000 Epochen war das Ergebniss ernüchternt. Die Maschine hatte einen Highscore von 10 (10 Punkte entsprechen einer gelösten Linie). Auch das Spielverhalten war nicht Sonderlich bemerkenswert. Es wurde nur noch versucht das Spiel schnellst möglich zu beenden, in dem nur ein Turm gebaut wurde. (Siehe Abbildung 2)

B. Version 2



Screenshoot aus dem Spiel *Abbildung 3*

Für die 2 Versionen wurde ab nun der Code umstrukturiert. Zu einem gab es anfangs noch einen Fehler, dass die Maschine einen Stein in der Luft platzieren konnte und zum anderen wurde die Laufzeit erhöht, um die Generationen schneller durchzulaufen. Der Reward wurde auch geändert und zwar wurde die Dichte der Steine, wenn diese beispielsweise große Lücken hatte, bestraft. Des weiteren wurde ein Gamma Wert hinzugefügt, um ältere Spielzüge in den Reward mit einzubeziehen.

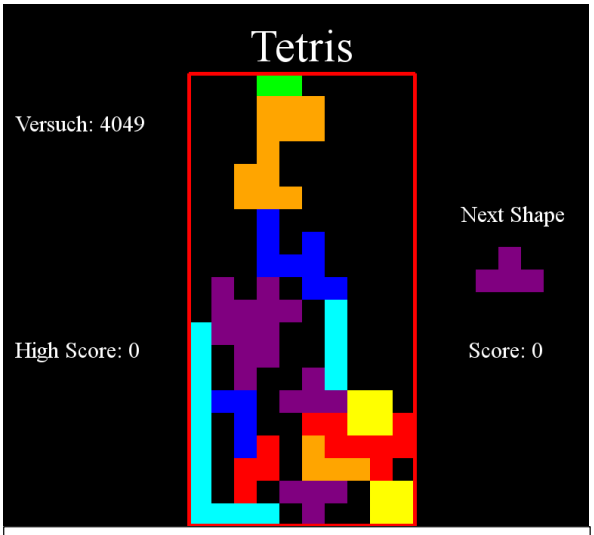
Das Ergebniss nach 4000 Generationen. Der Highscore lag bei 30, allerdings lief der Versuch nach einiger Zeit auch in eine Sackgasse, dass der Agent anfängt die Steine wie ein Turm aufeinander zu stappeln.(Siehe Abbildung 3)

C. Version 3

Das Ziel war es allerdings dem Spiel eines menschlichen Spielers näher zu kommen. Somit war das Ergebniss wieder nur ernüchternd. Einer Recherche nach sind 4 Rewards signifikant für die Maschine.(vgl.[5])

- Anzahl der gelösten Linien
- Anzahl der Löcher
- Die Höhenunterschiede der Reihen nebeneinander
- Die Durchschnittshöhe

Dabei wurden 3 dieser Rewards umgesetzt. Da die Durchschnitsshöhe mit den Höhenunterschied der Reihen nebeneinander korreliert, wurde die Durchschnittshöhe in dem Reward außen vor gelassen.



Screenshoot aus dem Spiel *Abbildung 4*

Zwar gab es nach 4000 Generationen kein besseren Highscore als in der zweiten Version, allerdings sieht das Bauen des Agenten wesentlich menschlicher aus und könnte vielleicht nach längerem Training sogar das Spiel bis unendlich spielen. (Siehe Abbildung 4)

IV. Verbesserungsmöglichkeiten

Das Projekt, einer künstliche Intellegenz, das Spiel Tetris beizubringen, hatte noch paar Stolpersteine und Verbesserungsmöglichkeiten. Zu einem wäre es sogar besser die Maschine bereits während des Spielens den Q-Table zu füllen und den Agenten zu trainieren. Zum anderen wäre es eine Überlegung wert, ob die Rewards nun wirklich zum Ziel führen oder ob es eine bessere Möglichkeit gäbe den Agenten etwas beizubringen. Beispielsweise könnte man den Reward für gelöste Zeilen niedriger halten, damit andere Schritte, die bis dahin geführt, haben nicht in Vergessenheit gelangen. Auch ob die States optimal gewählt sind, ist noch zu bedenken, man könnte den nächsten Stein in die States mit einbeziehen. Vielleicht würde aber auch eine andere Methode zum Beispiel die NEAT Methode zu einem besseren Ergebnis führen.

V. Literaturverzeichnis

[1] "Tetris Wiki – History," [Online]. Erreichbar: https://tetris.wiki/History. [Aufgerufen am 04.08.2020].

[2] Sebastian Heinz, "Einführung in Reinfocement Learning- wenn Maschinen wie Menschen Lernen", [Online]. Erreichbar: https://www.statworx.com/de/blog/einfuehrung-in-reinforcement-learning-wenn-maschinen-wie-menschen-lernen/.[Aufgerufen am 04.08.2020].

[3] Andre Violante, "Simple Reinfocement Learning: Q-learning", [Online]. Erreichbar: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56.[Aufgerufen am 04.08.2020].

[4] freeCodeCamp.org, "Python and Pygame Tutorial – Build Tetris! Full GameDev Course", [Online]. Erreichbar: https://www.youtube.com/watch?v=zfvxp7PgQ6c.[Aufgerufen am 04.06.2020].

[5] Abbildung 1, [Online]. Erreichbar: https://upload.wikimedia.org/wikipedia/commons/1/1b/Reinforcement_learning_diagram.svg

Reinforcement learning mit Unity ML-Agents am Spiel Schach

Michael Häckel
Hochschule Hof
Hof, Deutschland
michael.haeckel@hof-university.de

Zusammenfassung

Diese Arbeit behandelt, wie eine Lernumgebung in Form eines Schachspiels in der Entwicklungs-Engine Unity realisiert wurde und einige Versuche, in dieser Umgebung ein Netzwerk mit Hilfe des Unity Packages „Unity ML-Agents“ zu trainieren.

Das Spiel

Um Unity ML-Agents verwenden zu können wurde ein Schachspiel in Unity realisiert.

Komponenten

Es besteht aus einem Schachbrett, welches aus 64 Feldern und den 32 Schachfiguren besteht, einem Game Manager, der den Code für den allgemeinen Spielablauf enthält, einen Tile Manager der die einzelnen Felder enthält und mit diesen arbeitet, und zwei Player Managern die die Figuren und Werte der beiden Spieler - Schwarz und Weiß - enthalten. In der Szene befinden sich weiterhin eine Kamera einen EventManager um Klicks zu erkennen.

Sonderregeln

Im Gegensatz zum normalen Schach, bei dem der gegnerische König Matt gesetzt werden muss um zu gewinnen, muss in dieser Version des Schachspiels der gegnerische König geschlagen werden und es gibt kein Schach-setz-System, um die Programmierung des Spiels zu vereinfachen. Außerdem gibt sind En-Passant und die Rochade nicht möglich. Des Weiteren werden Bauern die in die letzte Reihe kommen automatisch zu Damen, anstatt den Spieler entscheiden zu lassen zu welcher Figur der Bauer wird. Die einzige Bedingung für ein Unentschieden ist, wenn 50 Züge lang keine Figur geschlagen wurde.

Figuren

Jede Figur hat einen Wert der den Bauerneinheiten[1] der Figur entspricht.

Jede Figur enthält ihr eigenes Script welches den Code zum Bewegen und die Werte, z.B. Farbe und Wert, enthält.
Wird ein Feld ausgewählt - entweder durch Code des Agenten oder Anklicken durch Menschliche Spieler – so wird, wenn sich eine Figur darauf befindet die sich bewegen darf, die StartMove() Funktion der Figur aufgerufen, die dann “Highlighter” auf den Feldern erstellt, auf die sich die ausgewählte Figur bewegen könnte. Wird dann ein Feld mit “Highlighter” ausgewählt, wird die Move() Funktion der vorher ausgewählten Figur aufgerufen, welche den Zug verarbeitet und die Figur bewegt. Jede Figur hat dafür ihre eigene Klasse, die von der Klasse Piece erbt.

Manager

Ist das Spiel vorbei werden durch den GameManager alle Flags resettet und alle Figuren zurück auf ihr Ursprüngliches Feld gesetzt. Der GameManager stellt außerdem auch sicher, dass die Figuren auf den richtigen Feldern angezeigt werden.
Wird – ob von Figuren die sich bewegen wollen, vom GameManager der die Figuren aufstellen will oder auch vom Agenten der Observations sammeln will – ein Feld gebraucht, wird das über den TileManager gemacht, er weist zum Beispiel den Feldern Werte zu, oder wandelt Felder in Werte um.

[1] https://schach.de/de/page/der-wert-der-figuren-im-schach

Was der TileManager für die 64 Felder macht, machen die PlayerManager für ihre jeweiligen 16 Figuren.

Learning Environment

In Unity ML-Agents befindet sich in einem Learning Environment mindestens ein Agent, der eine Komponente eines GameObjects in der Szene ist, und mindestens ein Behavior, welches die Attribute des Agenten, z.B. die Anzahl der Observations, beinhaltet und setzt. Jedes Behavior ist eindeutig mit einem "Behavior Name" festgelegt, wobei mehrere Agenten auf das gleiche Behavior zugreifen können.[2]

In dem Schachspiel ist das so realisiert, dass die beiden PlayerManager jeweils einen Agent als ChildObject haben. Diese beiden Agents haben das Behavior "Chess". Auf die Parameter des Behaviors wird in dem Kapitel "Behavior Parameter" weiter eingegangen.

Trainer Konfiguration

Unity ML-Agents hat mit Proximal Policy Optimization(PPO) und Soft Actor-Critic(SAC) zwei Algorithmen für Reinforcement Learning, wobei PPO der Standard ist.[3]

Eine weitere Funktion von Unity ML-Agents die verwendet wurde ist Self-Play. Damit können Agenten sowohl in Symmetrischen Spielen als auch Asymmetrischen Spielen trainiert werden. Da beide Seiten in einem Symmetrischen Spiel die gleichen Observations und Action Spaces haben können sie die gleiche policy verwenden. Self-Play kann zwar mit PPO und SAC verwendet werden, PPO wird jedoch empfohlen, weil es mit SAC zu Stabilitätsproblemen kommen kann.[4]

Aus diesen Gründen wurde sich auch für dieses Projekt für PPO und Self-Play entschieden.

Bei der Trainer Konfiguration wurden meistens empfohlene Werte hergenommen. Die Werte die im Folgenden nicht genauer erläutert werden wurden als default Wert belassen.

Der Wert max_steps bestimmt, wie lang trainiert wird bevor das Training beendet wird. Jeder Step von allen Agents wird dazu gezählt. Der Wert wurde trotz der empfohlenen Anzahl von 5e5 bis 1e7 auf 5e7 gesetzt damit länger trainiert wird und viele Daten gesammelt werden können.[5]

Für batch_size und buffer_size wurden die Werte 512 und 2048 gewählt. Für die batch_size wird ein kleiner Wert empfohlen, wenn ein Continuous Action Space verwendet wird. Die buffer_size sollte ein Vielfaches von der batch_size sein.[6]

Für den hidden_units Wert wurde 512 gewählt, da das der höchste empfohlene Wert war und ein hoher Wert bei komplexeren Problemen gewählt werden sollte.[7]

Um sicher zu stellen, dass immer gegen das neueste Model gespielt wird, wurde der Wert play_against_latest_model_ratio auf 1 gesetzt.[8]

Training

Im Folgenden werden vier Versuche einen Schachbot zu trainieren gezeigt. Dabei werden zuerst die Agenten erläutert, danach werden die Ergebnisse präsentiert.

Für jeden Versuch gelten jedoch die gleichen Regeln für die Vector Actions, was in

[2] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md#training-modes "Key Components"

[3] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md#training-modes "Deep Reinforcement learning"

[4] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md#training-modes "Training in Multi-Agent Environments with Self-Play"

[5] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md "Common Trainer Configurations – max_steps"

[6] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md "Common Trainer Configurations – batch_size & buffer_size"

[7] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md "Common Trainer Configurations – hidden_units"

[8] [8] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md "Self-Play – play_against_latest_model_ratio"

der OnActionReceived() Funktion des des PlayerAgent Scripts definiert ist:
Für jeden Wert von 0 bis 63 ist ein Feld zugewiesen. Wenn vom Bot eine Entscheidung verlangt wird, was nur passiert, wenn er gerade am Zug ist, wählt er eine Zahl von 0 bis 63, die auf eines der Felder passt. Daraufhin wird dieses Feld Aktiviert.

Versuch 1

Im ersten Trainingsversuch wurden noch keine Aktionen Maskiert. Die Idee war, dem Bot zuerst beizubringen Figuren zu bewegen und ihn dafür zu belohnen. Dieser Versuch besteht aus vielen kleineren Versuchen, die hier verallgemeinert zusammengefasst werden.

Agent

Maske:
Keine Maske.
Observations:
Der Status jeden Feldes. Wenn ein Feld leer ist hat es den Status 0, je nach Figur die sich darauf befindet bekommt es bei weißen Figuren einen Positiven Wert von 1-6 oder bei schwarzen Figuren einen negativen Wert von 1-6. (64)
Der bool isHighlighted jeden Feldes, um herauszufinden ob das Feld zum Bewegen markiert ist. (64)
Welche Farbe man selbst ist. (1)
Welche Farbe gerade dran ist. (1)
Insgesamt sind das 64 + 64 + 1 + 1 = 130.
Rewards:
Positive Rewards wenn er einen Zug ausgeführt hat, das heißt wenn er ein Markiertes Feld ausgewählt hat.
Einige Versuche auch mit Negativen Rewards wenn nach 2 Steps noch kein Zug ausgeführt wurde.

Beobachtung

Es konnte kein lernen beobachtet werden. Auch nach langem Training wurden noch größtenteils Felder ausgewählt die keinen Effekt haben.

Versuch 2

Im zweiten Trainingsversuch wurden Masken eingeführt, damit der Bot gezwungen ist Aktionen durchzuführen. Ab jetzt werden auch Rewards dafür vergeben, das Spiel zu beenden.

Agent

Maske:
Leere Felder und Felder mit gegnerischen Figuren werden maskiert, wenn sie nicht zum Ziehen markiert sind.
Observations:
Wie in Versuch 1.
Rewards:
+1 für Gewinnen
-1 für Verlieren
-1 für beide bei Unentschieden
-0.1 wenn kein Zug nach 2 Steps ausgeführt wird.

Beobachtung

Er hat zwar gelernt, war jedoch dabei sehr ineffektiv, weil er beim Training trotzdem sehr oft nur eigene Figuren ausgewählt hat und keine markierten Felder.

Versuch 3

Im dritten Trainingsversuch wurde die Maskierung weiter verschärft.

Agent

Maske:
Wenn noch keine Figur ausgewählt wurde können nur Felder mit Figuren die sich theoretisch bewegen könnten ausgewählt werden.
Wenn eine Figur ausgewählt wurde können nur markierte Felder ausgewählt werden.
Observations:
Wie in Versuch 1 und 2.
Rewards:
+1 für Gewinnen
-1 für Verlieren
-1 für beide bei Unentschieden

Beobachtung

Er hat etwas effektiver gelernt, da mit jedem zweiten Step ein Zug ausgeführt wurde, wodurch mehr Züge ausgeführt wurden wodurch mehr Daten gesammelt werden konnten.

Versuch 3.1

In diesem Versuch wurde ausprobiert mehr Rewards zu verteilen.

Agent

Maske:
Wie in Versuch 3.
Observations:
Wie in Versuch 3.
Rewards:
+1 für Gewinnen
-1 für Verlieren
-1 für beide bei Unentschieden
+Wert/2 wenn gegnerische Figur geschlagen
-Wert/2 wenn eigene Figur geschlagen
(Bei geschlagenen Figuren wurden verschiedene Formeln verwendet, die oben genannte war die mit der am meisten trainiert wurde.)

Beobachtung

Noch nach 20 Millionen Steps Training konnte man bei diesem Versuch nicht von lernen sprechen. Vermutlich wegen so hoher Komplexität der Umgebung und weil kein Zusammenhang zwischen Rewards und Aktionen erschlossen werden konnte.

Versuch 4

Im letzten Versuch wurden die Observations geändert da 130 Observations sehr viel sind und einige Observations nicht mehr gebraucht werden.

Agent

Maske:
Wie bei Versuch 3.
Observations:
Für jede Figur ein Int von 0 bis 64 auf welchem Feld er sich gerade befindet. 64 bedeutet die Figur wurde geschlagen.
Welche Farbe man selbst ist. (1)
Welche Farbe gerade dran ist. (1)
Insgesamt sind das 64 + 64 + 1 + 1 = 130.
Rewards:
+1 für Gewinnen
-1 für Verlieren
-1 für beide bei Unentschieden

Beobachtung

Scheinbar hat dieser Bot am besten gelernt. Das hängt höchstwahrscheinlich damit zusammen, dass weniger Observationen und mögliche Aktionen das beobachten der Umgebung und darauf zu reagieren stark erleichtern.
Als die Bots von Versuch 3 und Versuch 4 gegeneinander angetreten sind ergab sich folgendes Ergebnis:

Versuch 3: Weiß, Versuch 4: Schwarz
Nach 2811 Spielen hat Weiß 877 Spiele gewonnen, Schwarz hat 1556 Spiele gewonnen, und 378 Spiele gingen unentschieden aus.
Versuch 4: Weiß, Versuch 3: Schwarz
Nach 330 Spielen hat Weiß 19 Spiele gewonnen, Schwarz hat 311 Spiele gewonnen. Hier ist jedoch zu erwähnen, dass Schwarz eine Dominante Strategie gefunden hat, gegen die Weiß keine Lösung kennt.

Fazit

Obwohl Schach ein Extrem kompliziertes Spiel ist, konnte der Bot trotz weniger Rewards und anderer Hilfen einige Strategien entwickeln, um an den König des Gegners zu kommen. Und weil der Bot immer wieder gegen jemanden gespielt hat, der genau so spielt wie er selbst, hat er mit der Zeit gelernt wie er gegen solche Strategien vorgehen muss. Um die Ergebnisse zu verbessern hätte man einiges machen können, so hätte man zum Beispiel in der Trainer Configuration den learning_rate_schedule auf constant setzen können um länger zu trainieren. Außerdem hätte man mit Curiosity arbeiten können um mehr ausprobieren zu fördern.
Auch wenn die Bots am Ende verglichen nicht gut wurden denke ich, dass das Projekt gelungen ist, und dass man, wenn man mehr Zeit und Arbeit rein steckt, um einiges mehr mit Unity ML-Agents erreichen kann.

Nonogramm mit Reinforcement Learning lösen

Osman Minaz
Medieninformatik
Hof University of Applied Sciences
Hof, Deutschland
osman.minaz@hof-university.de

***Abstract*— Die vorliegende Studienarbeit befasst sich mit dem Trainieren eines neuronalen Netzes zur Lösung von Nonogramm-Rätseln. Dabei wurde ein Programm für das Erstellen und selbstständige Lösen von Nonogrammen implementiert. Zwar gibt es schon Solver, die ohne Reinforcement Learning funktionieren, doch diese sind nicht in jedem Fall optimal und kann je nach dem, wie groß und komplex das Nomogramm ist, sehr effizient oder gar nicht effizient sein. Das Ziel ist es also, ein Solver mittels Reinforcement Learning zu erschaffen, welche möglichst viele Varianten von Rätseln möglichst effizient lösen kann.**

Beim Erarbeiten der Studienarbeit hat die wissenschaftliche Arbeit „Solving nonogram puzzles by reinforcement learning“ von Frédéric Dandurand, Denis Cousineau und Thomas R. Shultz eine große Hilfe geleistet.

Keywords: Nonogramm, reinforcement learning, machine learning, neuronales Netz, SARSA

I. Einführung

In letzter Zeit hat das Logikrätsel Nonogramm durch moderne Apps wieder an Aufmerksamkeit gewonnen. Davor gelang dies Nintendo mit etlichen Spielen wie „Mario’s Picross“ aus dem Jahr 1995 oder „Picross DS“ aus dem Jahr 2007. Das als relativ ähnlich zu Sudoku empfundene Logikrätsel hat einen wesentlichen Unterschied zu seiner Konkurrenz. Die Endlösung ergibt meistens eine Pixelfigur, welches für eine größere Motivation sorgt. So besteht das Rätsel aus einem leeren Gitter mit beliebig vielen Zellen (5x5, 10x10, 15x15), welche durch Hinweise in Form von Zahlen im oberen und linken Teil ausgefüllt werden müssen. Dabei geben die Hinweise an wie viele Zellen in der Zeile oder Spalte ausgefüllt werden müssen. Wenn in einer Zeile oder Spalte mehrere Zahlen stehen, müssen die ausgefüllten Zellen mindestens eine nicht ausgefüllte Zelle als Lücke haben. Wenn alle Hinweise beachtet werden, wird meistens eine eindeutige Lösung resultieren.

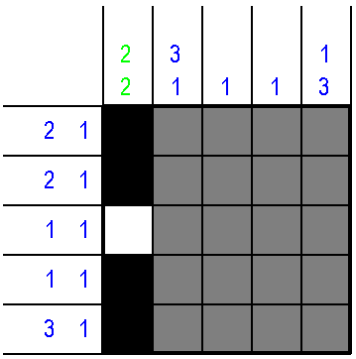


Abbildung 1 Beispiel für ein 5x5 Nonogramm mit dem ersten Lösungsweg

In Abbildung 1 kann man beispielsweise die erste Spalte eindeutig lösen, da die Spalte insgesamt nur 5 Zellen hat und man die zwei Hinweise durch eine leere Zelle trennen muss.

II. Theoretische Ansätze

Im Regelfall gibt es am Anfang immer Hinweise, die zu den eindeutigen Lösungswegen führen. Daraufhin werden weitere Zeilen und Spalten überprüft und versucht an mehr Hinweise zu gelangen. Dabei gibt es etliche Strategien und Vorgehensweisen. Am leichtesten ist es, wenn zuerst die Zeilen und Spalten gelöst werden, die in der Summe am höchsten sind.

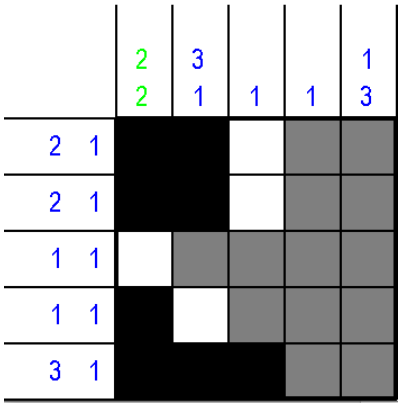


Abbildung 2 Eine mögliche Fortsetzung

Durch den Ansatz in der 1. Abbildung kann nun weitere Zeilen gefüllt oder ausgeschlossen werden.

Effizienter wäre es jedoch, wenn man, wie schon erwähnt, die Zeilen und Spalten löst, die die größte Summe haben, da diese meist unabhängig voneinander lösbar sind.

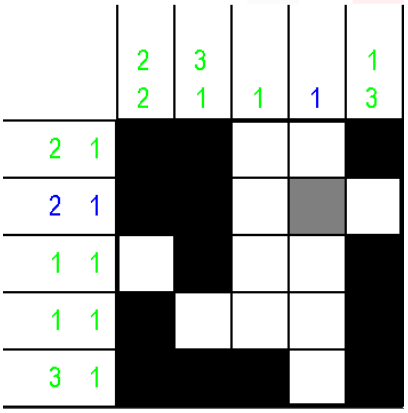


Abbildung 3 Lösung von den Zeilen und Spalten mit der höchsten Summe

Wie zu erkennen ist, wurde durch diese Methode die erste, dritte und vierte Spalte mitgelöst, wodurch die endgültige Lösung trivial geworden ist und nur noch eine Zelle auszufüllen ist.

Nun stellt sich die Frage, ob diese Methode immer am besten wäre. Wenn ein 20x20 Nonogramm vorliegt, wäre es unwahrscheinlich, dass viele Zeilen eine offensichtliche Lösung haben. Viel mehr werden nur Einzelne dadurch gelöst, sodass auf andere Methoden zurückgegriffen werden muss.

Das Rätsel kann beispielsweise Zeile für Zeile und Spalte für Spalte durchgegangen werden und einzelne Zellen dadurch gelöst werden. Oder es wird ohne System vorgegangen und zufällige Zeilen und Spalten herausgesucht und überprüft ob eine Zelle lösbar ist.

Die Überlegung bei dieser Studienarbeit ist es, ein neuronales Netzt aufzubauen, welches sich für eine Zeile oder Spalte entscheiden soll und somit das Rätsel so effizient wie möglich lösen soll.

III. Grundlagen

Bevor mit Reinforcement Learning angefangen wird, muss zuerst ein Programm implementiert werden, welches erstmal Nonogramme lösen lässt. Dafür wurde die Python-Bibliothek „Pygame“ für die Inputs und Outputs, sowie allgemein für die Oberfläche benutzt.

Außerdem wurden die meisten Logikfunktionen mit der Hilfe von Numpy implementiert.

Nachdem ein vorgegebenes Rätsel lösbar war, falsche und richtige Zeilen und Spalten erkannt worden sind, wurde Funktionen programmiert, welche zufällige Rätseln erstellt. Zudem werden Rätseln, die mehrere Lösungen haben, verworfen.

Die aktuelle Version kann nur 5x5 Nonogramme erstellen, wobei es leicht umstellbar ist. Die Zellen können per wiederholtem Mausklick entweder grau, schwarz oder weiß ausgefüllt werden. Grau steht dabei für unbekannt, schwarz für eine ausgefüllte Zelle und weiß für eine bekannte, leere Zelle. Dies wurde im Code durch jeweils einem Tupel dargestellt. Ausgefüllt ist dabei (1, 0), leer (0, 1) und unbekannt (0, 0).

Die Hinweise werden bei Fehlern rot, bei richtigen und vollständigen Linien grün und bei neutralen Linien blau markiert.

In den folgenden Punkten wird erläutert, was alles für das Reinforcement Learning verwendet worden ist.

A. SARSA

SARSA (State-Action-Reward-State-Action) ist ein ähnlicher Algorithmus wie Q-Learning. Mit der „On Policy“ Technik benutzt der SARSA Agent die ausgeführte Action durch die aktuelle „Policy“, um die Q-Werte zu lernen. Dabei interagiert der Agent mit der Umgebung und aktualisiert die Policy je nachdem für welche Action sich entschieden worden ist.

Des Weiteren sollen die States, Actions und das Rewardsystem erläutert werden:

1) States: Die States bestehen aus zwei Elementen. Zum einen sind es die Hinweise einer Linie und zum anderen die Inhalte von den Zellen der Linie. Die Inhalte können dabei entweder „unbekannt“, „gefüllt“ oder „leer“ sein.

2) Actions: Die Actions sind die Entscheidungen welche Linie als nächstes gelöst werden soll

3) Rewards: Die Actions werden durch dem „Distance-based rewards“ belohnt. Wodurch die Zeile vor und nach der Action verglichen wird und die gelösten Zellen angemerkt werden. Wenn in der Zeile davor fünf unbekannte gab und nach der Action nur zwei, gibt es für die Action 2/5 = 0,4 Rewards.

B. CNN

Als neuronales Netzwerk wurde ein „Convolutional Neural Network“(CNN) verwendet. Aufgebaut ist das Ganze mit drei Convolutional Layer ohne Pooling Layer. Für die Aktivierung wurde bei den ersten zwei Rectified Linear Unit (ReLu) verwendet. Der letzte Conv2D Layer lernt nur noch ein Filter mit ein kernel_size von (1, 1).

Der Input liegt als eine zweidimensionale Matrix vor, welcher vor dem Convolutional Layer gereshaped wird. Durch das Reshape am Ende, wird dann auf zehn Werte heruntergebrochen.

```
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 10, 14, 1)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 10, 1, 200)        3000
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 1, 20)         4020
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 1, 1)          21
_________________________________________________________________
reshape_1 (Reshape)          (None, 10)                0
=================================================================
Total params: 7,041
Trainable params: 7,041
Non-trainable params: 0
_________________________________________________________________
```

Abbildung 4 Model Zusammenfassung in der Konsole

IV. Vorgehensweise

Bevor das Model trainieren kann, muss noch eine Umgebung erstellt werden.

OpenAI Gym bietet standartmäßig verschiedene Environments an. Da aber der Agent ein anderes Problem lösen soll, wird eine eigene Umgebung erstellt.

Dabei wurden die Methoden, die Vorgeschrieben sind und jede Environment enthalten muss, implementiert:

1) *def __init__(self)*
2) *def reset(self)*
3) *def render(self, mode = "human")*
4) *def step(self, action)*
5) *def close(self)*

Dies garantiert die nötigie Funktionalität, damit der Agent später in der Lage ist zu lernen.

Im Konstruktur wird zunächst der Typ und die Shape vom „action_space“ definiert, welches alle möglichen Actions beinhaltet. Zusätzlich beinhaltet „observation_space“ alle Daten von der Umgegbung, die der Agent beziehen kann.

Die „reset“ Methode wird nach jeder Episode aufgerufen und setzt die Umgebung auf den Anfangszustand und ruft die Methode „observe(self)“ auf. Daraufhin wird dann mehrmals „step(self, action)“ aufgerufen, wodurch eine Action vom model ausgeführt werden muss. Dabei werden dann observation, reward, done und info zurückgegeben.

V. Training

Für das Trainieren wurde die „BoltzmannQPolicy()“ Implementation verwendet. Dies garantiert, dass das Explorationsverhalten vom Agenten stets zwischen zufälligen Actions und die optimalsten Actions liegt.
Es wurde sich für ein „Lernrate“ von 0,9 entschieden und einem „discount factor“ (gamma) von 0,9.

Als Optimzier kam der RMSprop Algorithmus zum Einsatz.

```
'dtype=object' when creating the ndarray
metrics = np.array(self.metrics[episode])
  21/20000: episode: 1, duration: 1.131s, episode steps: 21, steps per second: 19, episode reward: 9.7
  46/20000: episode: 2, duration: 0.322s, episode steps: 25, steps per second: 78, episode reward: 1.2
  62/20000: episode: 3, duration: 0.244s, episode steps: 16, steps per second: 66, episode reward: 9.3
  87/20000: episode: 4, duration: 0.346s, episode steps: 25, steps per second: 72, episode reward: 0.0
  96/20000: episode: 5, duration: 0.134s, episode steps: 9, steps per second: 67, episode reward: 8.50
 121/20000: episode: 6, duration: 0.421s, episode steps: 25, steps per second: 59, episode reward: 1.4
 146/20000: episode: 7, duration: 0.312s, episode steps: 25, steps per second: 80, episode reward: 8.5
 165/20000: episode: 8, duration: 0.251s, episode steps: 19, steps per second: 76, episode reward: 9.2
 182/20000: episode: 9, duration: 0.232s, episode steps: 17, steps per second: 73, episode reward: 9.5
 207/20000: episode: 10, duration: 0.333s, episode steps: 25, steps per second: 75, episode reward: 6.
 232/20000: episode: 11, duration: 0.319s, episode steps: 25, steps per second: 78, episode reward: 4.
 257/20000: episode: 12, duration: 0.409s, episode steps: 25, steps per second: 61, episode reward: 3.
 282/20000: episode: 13, duration: 4.070s, episode steps: 25, steps per second: 6, episode reward: 2.1
 302/20000: episode: 14, duration: 0.322s, episode steps: 20, steps per second: 62, episode reward: 9.
 319/20000: episode: 15, duration: 0.273s, episode steps: 17, steps per second: 62, episode reward: 8.
 336/20000: episode: 16, duration: 0.221s, episode steps: 17, steps per second: 77, episode reward: 9.
 346/20000: episode: 17, duration: 0.147s, episode steps: 10, steps per second: 68, episode reward: 8.
 357/20000: episode: 18, duration: 0.200s, episode steps: 11, steps per second: 55, episode reward: 8.
 367/20000: episode: 19, duration: 0.142s, episode steps: 10, steps per second: 70, episode reward: 8.
 383/20000: episode: 20, duration: 0.232s, episode steps: 16, steps per second: 69, episode reward: 8.
 405/20000: episode: 21, duration: 0.301s, episode steps: 22, steps per second: 73, episode reward: 10
 427/20000: episode: 22, duration: 0.289s, episode steps: 22, steps per second: 76, episode reward: 9.
 449/20000: episode: 23, duration: 0.292s, episode steps: 22, steps per second: 75, episode reward: 8.
 464/20000: episode: 24, duration: 0.211s, episode steps: 15, steps per second: 71, episode reward: 8.
 475/20000: episode: 25, duration: 0.179s, episode steps: 11, steps per second: 61, episode reward: 9.
 500/20000: episode: 26, duration: 0.437s, episode steps: 25, steps per second: 57, episode reward: 6.
```

Abbildung 5 Die ersten Schritte beim Training

VI. Ergebnisse

Nachdem die gegebene Anzahl an Schritte durchtrainiert worden sind, werden 10 Episoden getestet. Alternativ kann mit der Tastenkombination „CTRL + C“ das Training abgebrochen und direkt zum Testen übergegangen werden.

Beim Testen wird außerdem die maximale Anzahl von Schritten auf 25 beschränkt, da man davon ausgeht, dass spätestens beim 25. Schritt nicht mehr vorrangekommen wird.

```
done, took 331.964 seconds
Testing for 10 episodes ...
Episode 1: reward: 0.200, steps: 25
Episode 2: reward: 0.000, steps: 25
Episode 3: reward: 1.000, steps: 25
Episode 4: reward: 0.000, steps: 25
Episode 5: reward: 0.000, steps: 25
Episode 6: reward: 1.200, steps: 25
Episode 7: reward: 0.200, steps: 25
Episode 8: reward: 1.000, steps: 25
Episode 9: reward: 1.000, steps: 25
Episode 10: reward: 0.000, steps: 25
avg reward:  3.8867325102879278
avg moves:  9.559901234567901
```

Abbildung 6 Testergebnisse

Der durchschnittliche Reward betrug beim Testen 3,89 und die durchschnittlich benötigten Schritte betrug 9.56 Schritte.

VII. Fazit

Die Dokumentation der Bibliotheken waren optimal, um so eine Studienarbeit zu bearbeiten. Obwohl am Anfang viele Ansätze verworfen worden sind, kann das Endprodukt sich sehen lassen.

Wenn das Ganze erweitert werden sollte, könnte das Nonogramm beispielsweise vergrößert werden und ein Input Möglichkeit implementieren, damit spezifische Rätseln zum Lösen übergeben werden können.

Wünschenswert wäre auch, wenn Ergebnisse vom Reinforcment Learning geplottet somit visualisiert und mit anderen Ansätzen verglichen werden können.

Quellen

[1] https://pdfs.semanticscholar.org/d337/0379e7c9d498fc007f1aac16f781868165e2.pdf

[2] https://de.mathworks.com/help/reinforcement-learning/ug/sarsa-agents.html

[3] https://www.mikulskibartosz.name/using-boltzmann-distribution-as-exploration-policy-in-tensorflow-agent/

[4] https://www.geeksforgeeks.org/sarsa-reinforcement-learning/

[5] http://incompleteideas.net/book/ebook/node64.html

[6] https://towardsdatascience.com/creating-a-custom-openai-gym-environment-for-stock-trading-be532be3910e

[7] https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/

[8] https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

[9] https://keras.io/api/optimizers/rmsprop/

Deep-Q-Learning anhand Subway Surfers

Simon Köhler
Fakultät Informatik
Hochschule Hof
Hof, Deutschland
simon.koehler@hof-university.de

I. Zusammenfassung

In dieser Studienarbeit soll anhand von Reinforcement Learning einer KI beigebracht werden, selbstständig das Spiel „Subway Surfers“ zu spielen. In folgenden Punkten werden Erklärungen und Statistiken bezüglich der Umsetzung, der Erfahrungen und Endergebnisse, vorgestellt.

II. Einleitung

A. Spielprinzip

In dem Spiel „Subway Surfers“ läuft der Spieler vor einem Feind davon. Das Ziel des Spiels ist es, solange wie möglich vor dem Feind zu entkommen. Der Spieler hat die Möglichkeit auf drei verschiedene Bahnspuren zu laufen. Während des Spiels muss der Spieler fahrenden Zügen, Barrikaden und anderen Objekten ausweichen, um nicht gefangen zu werden. Dabei verdient er sich mittels verschiedener Interaktionen Punkte, die der Gesamtpunktzahl, dem „Score“, angerechnet werden. Der Score ist der Maßstab, welcher angibt, wie gut das Spiel gespielt wurde.

B. Reinforcement Learning

Reinforcement Learning - in Deutsch auch bestärkendes Lernen genannt - ist ein Unterbereich des Maschinellen Lernens. Reinforcement Learning ermöglicht es, einem Agenten eigenständig eine Strategie erlernen zu lassen. Damit der Agent weiß, welche Aspekte für seine Strategie relevant sind, muss dieser diese mittels einer Belohnungs-Funktion bewerten. Damit die Strategie das bestmögliche Ergebnis hervorbringen kann, versucht der Agent, die zur Aktion und Zustand zugehörigen Belohnungen zu maximieren. Die Belohnungs-Funktion beschreibt mit einem Wert, wie gut die ausgeführte Aktion zu seinem Zustand ist. [1]

C. Q-Learning

Um die Policy-Funktion (**π**) zu trainieren, gibt es verschiedene Methoden. Eine bekannte und häufig verwendete Methode ist das Q-Learning. Die Funktion, die der Methode den Namen verleiht, ist die sog. Q-Funktion $Q(s,a)$ Diese beschreibt für einen State s und einer Action a den Belohnungswert Q. Die Vielzahl der möglichen Q-Werte wird in einer Q-Matrix aus zwei Werten gespeichert. So ergibt sich für jeden möglichen State s und jeden mögliche Action a ein Wert Q.
Für jeden beliebigen Zustand wird eine Action ausgeführt, die zur Belohnung Q führt. Mit Hilfe der Bellman-Gleichung berechnet sich die Belohnung aus der Q-Funktion:

$$Q(s,a) = r + \gamma * max\Big(Q(s',a')\Big)$$

Der Wert r bezeichnet den Reward, welcher nach der Ausführung einer Action für einen State errechnet wird. Im Normalfall ist der Reward r größer, wenn die Action für den Zustand eine positivere Auswirkung hervorruft. Die Funktion

$$max\Big(Q(s',a')\Big)$$

gibt den höchsten Belohnungswert für den folgenden State und ihrer Action an. Dies heißt, dass die Belohnung höher ausfällt, wenn der nächste Zustand ein Besserer ist. Der Faktor Gamma γ gibt an, wie stark der nächste State und die dazugehörige Action in den aktuellen Belohnungswert mit hineinspielt. In manchen Umgebungen gibt es zu viele Zustandsmöglichkeiten. Sollte diese Anzahl über einen gewissen Wert, der abhängig verschiedener Faktoren ist, liegen, empfiehlt es sich keinen Q-Matrix, sondern andere Modelle zu verwenden. Für diese Studienarbeit ist dies der Fall, weshalb die Architektur eines Convolutional Neural Network CNN als Policy-Funktion zum Einsatz gekommen ist.

D. Convolutional Neural Network (CNN)

Das Convolutional Neural Network ist eine Architektur, die sich Inspiration von einem menschlichen Gehirn sucht. Das menschliche Gehirn besteht aus vielen Neuronen, die bei unterschiedlicher Aktivierung unterschiedlichste Zustände annehmen können. Gleiches versucht die Architektur CNN. Mit Hilfe von Neuronen, die zueinander verknüpft sind, werden diese bei bestimmten Ereignissen aktiviert. Diese Ereignisse sind im Falle des CNN beispielsweise Bilddaten, welche als Input in das CNN hineingegeben werden. Convolutional Neural Networks sind vor allem für die Auswertung von Bilddaten vorteilhaft geeignet, sind jedoch nicht darauf beschränkt. Für diese Studienarbeit ist die Verwendung eines CNN gut geeignet, da als Input Bilddaten (Höhe x Breite x Farbkanäle) zur Verfügung stehen. Das Convolutional Neural Network wurde mittels Keras [2], welches eine Unterkategorie von Tensorflow [3] ist, erstellt.

III. Umsetzung

A. Allgemeines

In den folgenden Punkten wird das Lernverfahren der Policy-Funktion von Anfang bis Ende vorgestellt und erläutert.

B. Umgebung

Das Spiel kann man im Browser unter Poki.de [4] spielen. Folglich verfügt die Umgebung nur über die Möglichkeit die States des Spiels über die Pixel der Bilder des Bildschirmes zu erfassen.

C. Zustandserfassung

Es gibt viele Ansätze, wie man die States ermitteln kann. So ist ein Ansatz den Abstand der zu steuernden Person zu den jeweiligen Objekten, sowie deren Typ als auch die jeweilige Bahnpositionen, zu ermitteln. Dafür benötigt man jedoch eine eigene Umgebung, die diese Parameter bereitstellt. In diesem Fall stehen diese jedoch nicht zur Verfügung. Da das Spiel über den Browser zu spielen ist, gibt es nur die Möglichkeit die Zustände als Bilder zu erfassen.

Anders als bei Schach, bei dem man genügend Zeit besitzt, die nächste Action zu berechnen und danach den zur Action gehörigen Step auszuführen, läuft das Spiel „Subway Surfers“ durchgehend weiter. Folglich verzögert sich der Step, je länger die Bilderfassung und die Auswertung des Bildes benötigt. Die Action wird somit zwar für den State ausgeführt, jedoch befindet sich das Spiel bereits in einem neuen State. Dieser Zeitunterschied bei solchen Spielen ist unvermeidbar, sollte jedoch so klein wie möglich gehalten werden. Eine weitere unvermeidbare Unannehmlichkeit ist die zeitliche Schwankung der verschiedenen Bilderfassungsmodule. Die Bilder werden daher nicht zur selben Zeit erfasst. Aufgrund dessen ist die Zeit zwischen dem State und der Action nicht gleich, was dazu führt, dass ein State nicht immer eine zugehörige Action besitzt. Dieser Fehler ist zwar sehr gering, verringert jedoch die Effektivität des Lernprozesses. Allgemein gilt, je kleiner die Zeit zum Erfassen des Bildes ist, desto kleiner auch der Schwankungsbereich dieser.

Aus diesem Grund wurden verschiedene Bilderfassungsmodule auf ihre Zeit getestet. Die Zeit bezieht sich dabei auf die benötigte Zeit zur Erfassung des Bildes.

```
Time D3D:  0.02792501449584961
Time Grab:  0.03903388977050781
Time MSS:  0.23854970932006836
Time Win32:  0.017306566238403 32
Time pyautogui:  0.04486989974975586
Time pyscreeze:  0.0460810661315918
>>>
```

Abbildung 1 – Vergleich der Bilderfassungsmethoden

Abbildung 1 zeigt die verschiedenen Methoden im Vergleich. Die kürzeste Zeit benötigt die Methode Win32 [5] Die anderen Methoden werden im weiteren Verlauf der Doku nicht weiter erläutert, weshalb diese nicht im Literaturverzeichnis erwähnt werden.

D. Datenvorverarbeitung

Trotz der Verwendung einer anderen Policy- Funktion ist es auch für ein CNN schwierig alle Pixel als Input zu berechnen und daraus einen Zusammenhang zwischen dem Output und dem Input herzustellen. Aus diesem Grund werden die vorgesehenen Inputdaten auf eine relevante Größe verkleinert. Dieses Verfahren nennt man Resize.

E. Stepausführung

Sobald das Bild verarbeitet wurde, wird dieses in passender Form in die Policy-Funktion gegeben. In diesem Fall ist die Policy-Funktion vergleichbar mit der Struktur des CNNs. Die Pixel werden in das CNN hineingeben und ausgewertet. Das Modell sagt für den Input einen Output vorher. Dieser Output wird Prediction genannt und umfasst die Anzahl der möglichen Actions.
Damit das Modell nicht einmal eine gute Action lernt und diese immer wieder verwendet, obwohl es möglicherweise eine bessere Action gibt, wird ein Zufallsfaktor in der Ausführung der Action miteinbezogen. Sollte dieser Fall eintreten, so wird eine zufällige Action ausgeführt. Dies verhindert die Erreichung eines lokalen Minimas.
Nachdem die Action ermittelt wurde, werden Zustand und Action gespeichert.
Im Falle eines Spiels, welches einem die Zeit lässt, die States und deren Actions zu ermitteln, kann der Reward pro Frame ermittelt und das Modell trainiert werden. Ein Frame beinhaltet die Erfassung des State, Berechnung der Action und Ausführung dieser. Da es sich bei Subway Surfers jedoch um ein ständig laufendes Spiel handelt, in dem sich die States jederzeit ohne Einschränkung ändern, ist es nicht möglich den Reward pro Frame, sondern erst am Ende des Spiels zu ermitteln.

F. Reward

Der Reward ist der Wert, der für einen Zustand und der dazugehörigen Action nach jedem Frame ermittelt wird. Je höher der Reward, desto besser ist die Entscheidung sich für diesen Action zu entscheiden.
Der Reward für diese Anwendung wird aus verschiedenen Faktoren berechnet:

1) Gold:

Während des Spiels kann der Spieler Gold einsammeln, welcher am Ende mit einem Faktor auf den Score addiert wird. Folglich ist es sinnvoll dem Modell beizubringen, wenn möglich Gold einzusammeln.
Das Gold wird über einen Bildausschnitt ermittelt. Dabei wird wie bei der Bilderfassung des Inputs des CNN auch ein kleiner Ausschnitt für die Goldzahl erfasst. Mit Hilfe des Moduls Tesseract OCR [6] ist es möglich aus einem Bild Text zu erkennen. Dieser Text gibt für den Zustand die jeweilige Menge an Gold an.

2) Zeit

In Subway Surfers gibt es abgesehen von dem Gold nur die Möglichkeit eine Action als gut oder schlecht zu bewerten. Gute Actions sind solche bei dem der Spieler noch im Spiel bleibt, schlechte hingegen sind solche bei dem der Spieler das Spiel verliert. Da es nur zwei Typen von Actions gibt, ist es nicht nötig, die gute Action zu bewerten, sondern nur die schlechte Action zu bewertet. In diesem Fall ist es die Action, die zum Ende führt. Im Normalfall wird einfach die letzte Action schlecht bewertet, jedoch gestaltet sich die Erfassung des Endzustandes als schwierig. Für manche

Enden endet das Spiel abrupt und die zuletzt ermittelte Action ist die tatsächlich letzte Action. Jedoch gibt es auch Enden, bei denen das Spiel für einen Bruchteil noch weiterläuft, weshalb weitere States und Actions ermittelt und ausgeführt werden. Diese Actions sind für das Spiel nicht relevant, da es eigentlich schon vorbei ist. Aus diesem Grund kann man die entscheidende Action nicht immer zuverlässig ermitteln, weshalb ein Zeitfaktor mit einbezogen wird, der die letzten Actions schlechter bewertet. Dabei werden die Actions absteigend pro Frame schwächer bewertet (am Schlechtesten wird die letzte Action bewertet).

3) Beziehung zu nächstem Frame

Es gibt Spielsequenzen, die nahezu unmöglich zu schaffen sind, wenn man den falschen Weg wählt. Damit eine Möglichkeit besteht aus dieser Sackgasse wieder herausfindet, ist es notwendig eine Beziehung zu dem nächsten State herzustellen. Das Modell lernt, dass der nächste State nur schlechte Entscheidungen bringt, weshalb er seine Entscheidung schließlich auch schlechter bewertet.

4) Nichts machen

In Subway Surfers gibt es insgesamt 5 Actions. (Links, Rechts, Springen, Rollen, Nichtstun).
Wenn der Spieler springt, kann er während er in der Luft ist keine weiteren Sprünge ausführen. Das Gleiche gilt auch für die Action Rollen. Sollte jedoch ein Springen, Rollen, Links, Rechts während der gleichen Action ausgeführt werden, so wird diese gespeichert. Folglich wird diese Action nach Beendigung der letzten Action ausgeführt. Damit dies nicht funktioniert, können diese nicht innerhalb einer bestimmten Zeit weitere Aktionen ausführen. Ausnahme bildet dabei die Action Nichtstun. Aus diesem Grund wird die Action besser bewerten, wenn es sich um Nichtstun handelt. So kann der Spieler ohne weitere Konsequenzen, so schnell wie das Modell zulässt, weitere Actions ausführen.

G. Bearbeitung der Daten

Da die Erfassung der Goldanzahl nicht zu 100% richtig verläuft, ist es notwendig, die letzten Fehler manuell über eine Funktion zu ermitteln und diese zu korrigieren.

H. Training

Nachdem das Spiel beendet ist, werden die zu den jeweiligen States und Actions relevanten Rewards ermittelt. Die neuen Predictions werden schließlich an die neuen Rewards angepasst.
Schlussendlich wird CNN mit den neuen Predictions trainiert. Dies führt zum Anpassen der Policy-Funktion.

IV. Ergebnisse

Für das Endergebnis werden verschiedene Versionen getestet, die im Folgenden vorgestellt und analysiert werden. Jede Version wird mit 1000 Durchläufen getestet.

A. Versionen

1) Version 1: Kein Limit

In dieser Version wird dem Modell keine Limits gesetzt vgl. Stepausführung. Jede Action wird so schnell, wie möglich ausgeführt, was zur Speicherung und Ausführung von Actions führt.

Abbildung 2 – Rohbild des großen Bildausschnitts

Abbildung 2 zeigt das Bild, welches pro Frame erfasst wird. Dieses Bild wird verkleinert und zur Ermittlung der Action in das CNN geben.

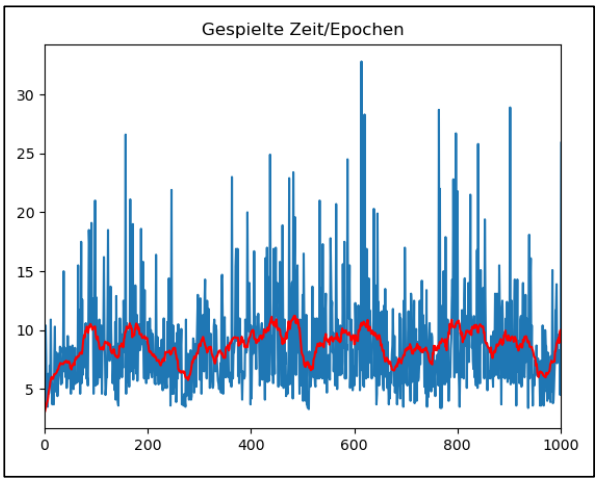


Abbildung 3 - Verlauf der Spiele bei Version 1

Nach einigen Runden ergibt sich ein für eine Analyse annehmbarer Verlauf aller Spieler. Abbildung 3 zeigt über die Runden gespielte Zeit in Sekunden an. Dabei ergibt sich eine rote Linie. Diese beschreibt je über 25 Runden den Mittelwert der gespielten Zeit. In Abbildung 3 ist zu sehen, dass sich der Spielzeit phasenweise verändert. Es gibt Phasen, in denen das Modell besser gespielt hat als in anderen Phasen. Diese Tatsache kann mit dem Lernfortschritt des CNN und der zufälligen Spielsequenzen zu erklären. Das CNN passt beim Lernen nicht alle Bereiche gleich an, sondern fokussiert sich dabei immer nur auf einen bestimmten Bereich. Für

diesen Bereich werden dann die Parameter so angepasst, dass für die Zustände und derer Actions die neuen Predictions übereinstimmen. Sollte das CNN in diesem einem Bereich nach einiger Zeit nur noch eine kleine Änderung verändern, so ändert sich der Lernbereich. Dies hat zur Folge, dass bei nächster Anpassung ein anderer Bereich priorisiert gelernt wird. Die neuen Anpassungen ändern die Parameter in diesem Bereich. So kann es vorkommen, dass der bereits gut gelernte Bereich weniger Relevanz für die Policy-Funktion hat, was zu einer temporären Verminderung der Qualität führt. [2]
Ein weiter Punkt ist der Zufallscharakter des Spiels. Der Verlauf des Spiels besteht aus mehreren Sequenzen, die unterschiedlich aufgebaut sind und nicht in immer gleicher Reihenfolge ablaufen. Manche Sequenzen sind leicht zu lernen. Hingegen anderen Sequenzen können sehr schwierig zu lernen sein.
Das zeigt sich, in der Schwankung der Zeit wieder.

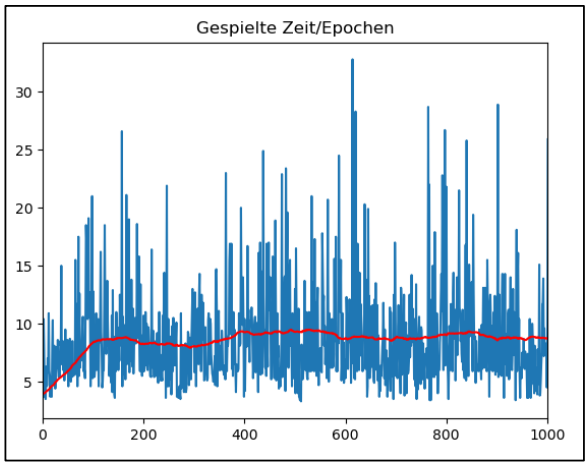


Abbildung 4 - Verlauf der Spiele bei Version 1

Abbildung 4 zeigt die gleichen Spieldaten wie in Abbildung 3, jedoch ermittelt sich der Mittelwert der Spielzeiten in dieser Abbildung über 100 Runden. Eine Einbeziehung von mehr Runden führt zu einer Glättung des Mittelwertverlaufs. Dies ist für die Analyse und Auswertung der Daten komfortabler.

Wie in Abbildung 4 zu sehen ist, steigt die durchschnittliche Spielzeit nicht an. Folglich erweist sich Version 1, welche ohne Limitierung der Zeit zur Ausführung der Actions getestet wurde, als Fehlschlag.

2) Version 2: Limit

Da sich die Variante ohne Limits als schlecht trainierbar erwiesen hat, wurde in Version 2 die Zeiten der Ausführung der Action angepasst.

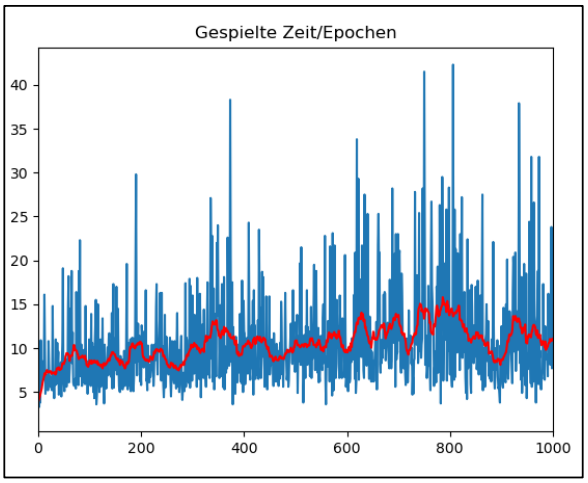


Abbildung 5 - Verlauf der Spiele bei Version 2

Im Vergleich zur vorherigen Version ist in Abbildung 5 ein leichter Anstieg der durchschnittlichen Spielzeit über die Runden zu erkennen.

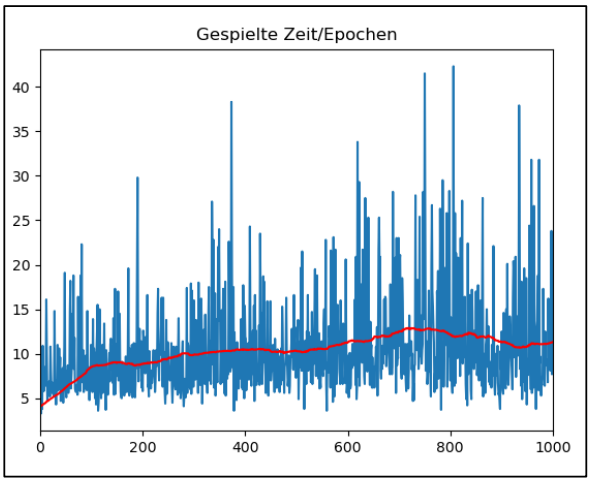


Abbildung 6 - Verlauf der Spiele bei Version 2

Wie auch in Abbildung 4 wird in Abbildung 6 zur Veranschaulichung der Mittelwert über eine größere Anzahl an Runden ermittelt.

3) Version 3: Kleinerer *Bildausschnitt*

Eine Möglichkeit das Modell zu testen, ist es die Inputdaten zu ändern. In dieser Version wurde ein kleinerer Bildausschnitt gewählt.

Abbildung 7 - Rohbild des kleinen Bildausschnitts

Im Vergleich der Abbildungen 2 und 7 ist zu erkennen, dass der Rand in Abbildung 2 nicht mehr zu sehen ist. Züge und andere Spielelemente werden in Abbildung 7 nur teilweise erfasst.

In dieser Version wird getestet, ob eine Verminderung der Inputdaten und der Erfassung teilweiser Objekte zu einer Verbesserung der Policy-Funktion führt.

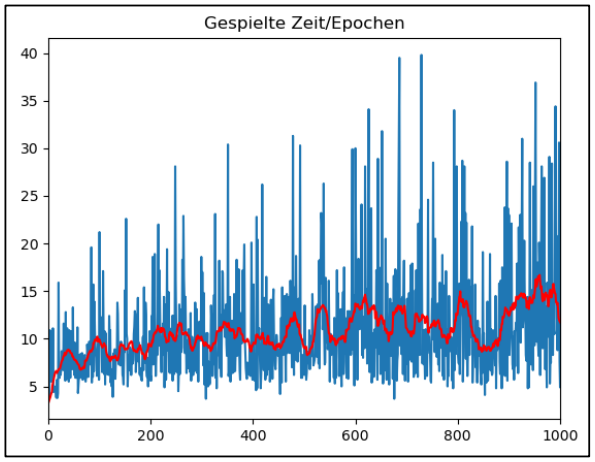


Abbildung 8 - Verlauf der Spiele bei Version 3

In Abbildung 8 werden die Schwankungen der Mittelwerte gut dargestellt. Die Schwankung ist zu Beginn des Lernprozesses kleiner als nach einigen Runden. Erklären ist dies durch die dazukommenden Sequenzen. Je länger das Spiel geht, desto mehr neue Sequenzen muss das Modell spielen. Es gibt Sequenzen, die erst nach einigen Sekunden auftreten können. Diese Sequenzen sind schwieriger zu lernen, da diese nicht immer auftreten und erst erscheinen können, wenn der Spielverlauf weiter fortgeschritten ist, was jedoch nicht immer der Fall ist.

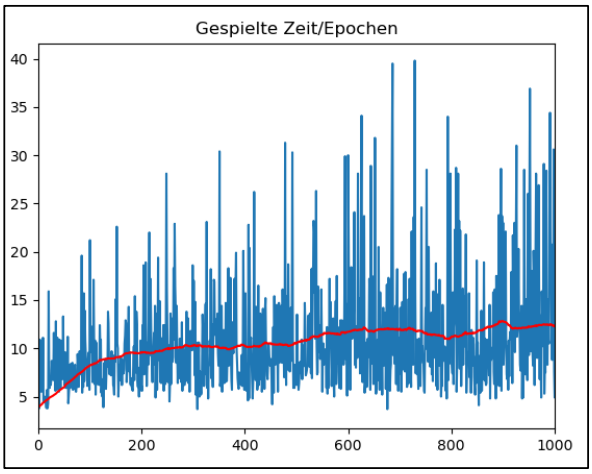


Abbildung 9 - Verlauf der Spiele bei Version 3

In Abbildung 9 ist der Anstieg der durchschnittlichen Spielzeit vergleichbar wie in Abbildung 6. Eine geringe Verbesserung ist in Version 3 trotz allem der Fall.

4) Version 4: Kleiner Bildausschnitt mit Graustufen

Die Inputdaten der ersten Versionen besitzen 3 Farbkanäle. In diese Version wird getestet, ob es besser ist, nur graustufige Bilder als Input zu verwenden.

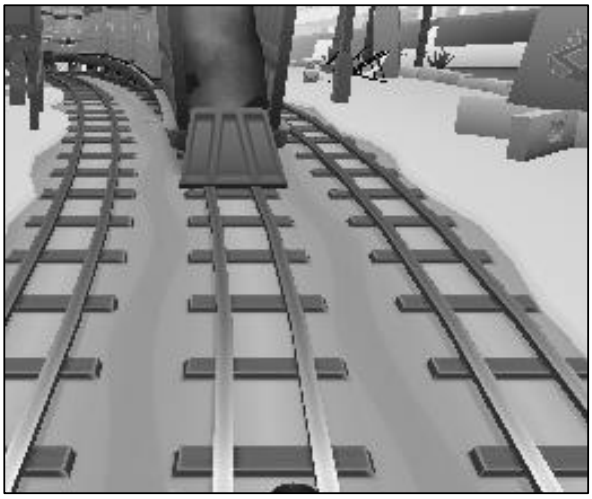


Abbildung 10 - Graustufiges Rohbild eines kleinen Bildausschnitts

Abbildung 10 zeigt das Rohbild des Modells, welches in Graustufen umgewandelt wird. Diese Bild wird verkleinert und in das CNN geben.

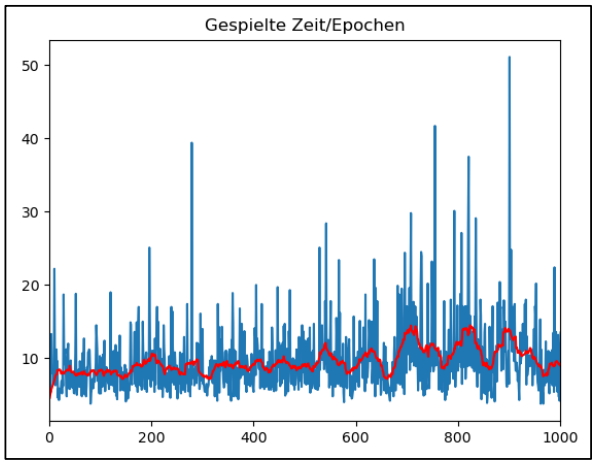


Abbildung 11 - Verlauf der Spiele bei Version 4

Der Verlauf in Abbildung 11 stellt offenkundig da, dass die Entfernung der Farbkanäle zu einem schlechten Ergebnis führt. Erklären ist dies, mit dem Kontrast des Bildes. Der Kontrast des Bildes spielt in der farblosen Variante eine große Rolle. Bei einer kleineren Differenz der Pixelwerte ist es für das Modell schwieriger unterschiedliche Zusammenhänge festzustellen. Begründet wird dies, durch die Tatsache, dass das Modell mit Zahlen rechnet.

Es gibt Runden, die über 30 Sekunden gehen, jedoch ist dies dem geringen Zufallswert zu verschulden.

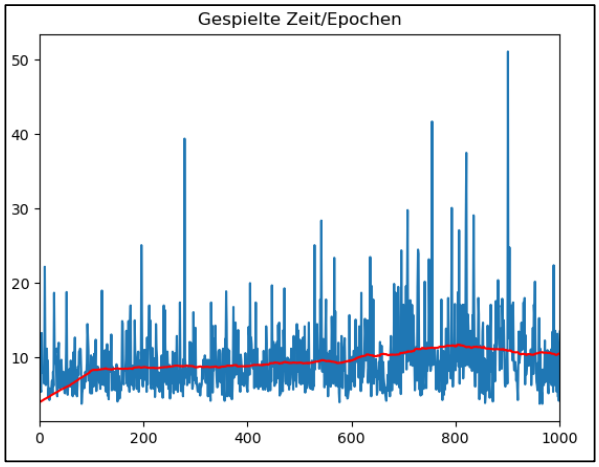


Abbildung 12 - Verlauf der Spiele bei Version 4

Wie auch in den jeweiligen Abbildungen, welche die durchschnittliche Spielzeit über 100 Runden darstellt, so zeigt auch Abbildung 12 den Verlauf der durchschnittlichen Spielzeit über 100 Runden an.

5) Version 5: Kleiner Bildausschnitt in 3er Folge

Um Bewegung im Spielverlauf zu erfassen, wurde als Input nicht nur ein Bild hineingeben, sondern 3 Bilder. Dabei sind diese Bilder, die Bilder der letzten drei Frames. Folglich startet das Spiel in dieser Version 2 Frames verzögert.

Abbildung 13 - Bearbeitetes Inputbild von Version 5

Abbildung 13 zeigt, ein Inputbild, welches in das CNN zur Berechnung der besten Action, gegeben wird. Dieses Bild ist der endgültige Input des CNNs. Dieses wurde zuvor, wie auch bei den anderen Versionen, von einem erfassten Bildausschnitt auf ein kleineres Bild herunter skaliert.

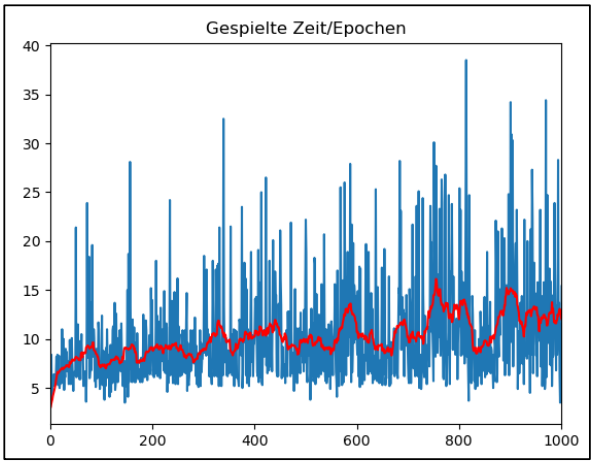


Abbildung 14 - Verlauf der Spiele bei Version 5

In Abbildung 14 sind mehrere Runden zu sehen, die es über die 30 Sekunden Grenze geschafft haben. Vor allem in den letzten Runden sind längere Spielzeiten vorhanden.

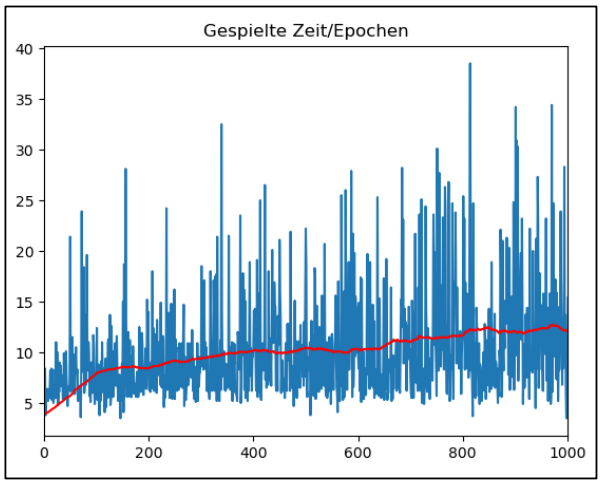


Abbildung 15 - Verlauf der Spiele bei Version 5

In Abbildung 15 ist der Anstieg der durchschnittlichen Spielzeit deutlich größer als bei den vorherigen Versionen.

V. Fazit

A. Vergleich der Versionen

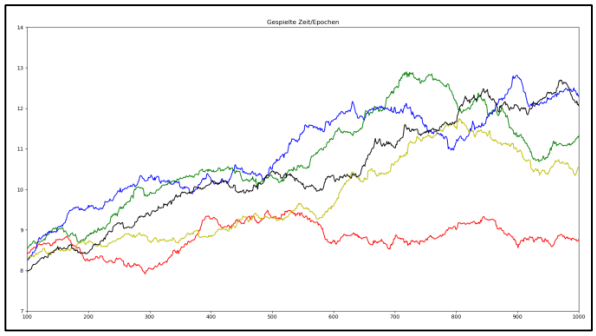


Abbildung 16 - Vergleich der Versionen

Abbildung 16 zeigt den Vergleich aller Versionen. Die verschieden farbigen Verläufe zeigen über die Runden die durchschnittliche Spielzeit der einzelnen Versionen an.

Am schlechtesten hat die Version 1 mit keiner Limitierung der Aktionszeiten abgeschnitten (rot). Gefolgt wird dieser von der graustufigen Variante (gelb). Im hinteren Bereich des Graphen weist diese Version einen Höhepunkt auf. Dieser extreme Anstieg, der schließlich zum Abfall führt, ist durch die Zufälligkeit der Spielsequenzen zu erklären. Version 2 (grün) und 3 (blau) schneiden ähnlich ab, wobei Version 3 konsistenter als Version 3 ist. Die beste Variante ist die letzte vorgestellte Variante (schwarz). Diese ist zu Beginn schlechter als die meisten anderen Varianten, was daran liegt, dass diese Version die 3-fache Menge an Inputdaten zu Verfügung steht, schneidet jedoch am Ende am besten ab. Eine mögliche Erklärung, weshalb dies der Fall ist, ist die mögliche Wahrnehmung von Bewegungen.

B. Verbesserungsvorschläge

Diese Studienarbeit hat nicht auf das perfekt spielende Modell für Subway Surfers abgezielt, sondern sollte veranschaulichen, welche Faktoren Einfluss auf das Lernverhalten des Modells nehmen können. Weitere Faktoren, die das Modell positiv hinsichtlich der Effektivität ändern könnten, sind:

1) Bilderfassung

Eine zentrale Rolle, die zur Effektivität der Policy-Funktion führt, ist die Bilderfassung. Um bereits zu Beginn Fehler verringern zu können, ist es notwendig, bei Echtzeitspielen, die Zustände so schnell wie möglich erfassen und verarbeiten zu können. Die schnellst möglichste Variante zur Bilderfassung sollte gewählt werden.

2) Input

Es ist sinnvoll, sich über die Inputdaten Gedanken zu machen.
Im Vergleich zur Version 1 und 2 bei denen der Rand der Spielumgebung miterfasst wird, welche jedoch keinen Einfluss auf das Spielverhalten haben dürfte, ist es sinnvoll, nur die Daten zu erfassen, die wirklich Einfluss auf das Spielverhalten nehmen. Sollte die Pixel eines Bilds als Inputdaten dienen, kann die Verkleinerung dieses einen erheblichen Einfluss auf die Schnelligkeit des Lernverhaltens des Modells nehmen.

3) Modelltiefe

Allgemein gilt, je mehr Schichten ein neuronales Netz hat, desto länger benötigt dieses zu lernen. Dabei ist nicht die Zeit der Veränderung der Parameter zu betrachten, sondern die Zeit, die das Modell benötigt, um gute Ergebnisse zu erzielen.

4) Modellstruktur

Ein weiterer Faktor kann die Struktur des Modells sein. Wie in dieser Studienarbeit sich ein CNN als Modell angeboten hat, so kann es für andere Umgebungen andere Modellarchitekturen geben, die besser geeignet sind.

5) Optimizer

Je nachdem, welcher Optimizer gewählt ist, kann dieser zu einem anderen Lernverhalten führen. Das Testen verschiedener Optimizer und ihrer Loss-Funktionen ist empfehlenswert.

6) Training

Einen Einfluss kann das Festlegen einer anderen Batchgröße sowie die Zahl der zu trainierenden Epochen sein.

7) Zustandserfassung

Der Faktor, der am meisten in die Effektivität des Modells hineinspielt, ist die Art der Zustandserfassung. Beispielsweise wäre es möglich, nicht die Pixel des Bildes als Input zu wählen, sondern die Gleisposition, die Entfernung und die Art des Objekts zu ermitteln. Dies ist jedoch nur möglich, wenn die Umgebung dies zulässt.

VI. LITERATURVERZEICHNIS

[1] Reinforcement Q-Learning, https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/, zuletzt aufgerufen am: 31.07.2020

[2] Keras, Offizielle Seite

[3] Tensorflow, Offizielle Seite

[4] Poki , https://poki.de/g/subway-surfers

[5] Python Win32, https://pypi.org/project/pywin32/

[6] Tesseract OCR, https://tesseract-ocr.github.io/

Angewandtes Maschinelles Lernen am Beispiel von Blackjack

Selim Demir
Medieninformatik
Hochschule Hof
Hof, Bayern
selim.demir@hof-university.de

ABSTRACT

Mit welcher Wahrscheinlichkeit kann eine Künstliche Intelligenz mehrere Partien Blackjack gewinnen. In der Studienarbeit wird darauf eingegangen, mit welcher Herangehensweise und Methodik eine KI sich darauf vorbereitet das Spiel zu erlernen und mit welcher Wahrscheinlichkeit sie das Spiel nach einer gewissen Anzahl von Testversuchen gewinnt.

• Computing methodologies ~ Machine learning ~ Learning paradigms ~ Reinforcement learning ~ Sequential decision making

1. Blackjack Einführung

Bei dem Spiel Blackjack handelt es sich um ein Glücksspiel, das in sehr vielen Kasinos zu finden ist. Das Hauptmerkmal an Blackjack ist, dass das Spiel, im Gegensatz zu vielen anderen Kartenspielen, eine sehr einfache Spielstruktur hat. Blackjack kann man in vielen verschiedenen Varianten spielen, in dieser Arbeit wird jedoch das klassische Blackjack näher betrachtet, da dieses am weitesten verbreitet und auch am einfachsten zu verstehen ist. Beim klassischen Blackjack unterscheidet man zwischen dem Dealer und dem/den Spieler/n. Vor Beginn setzt jeder Spieler einen gewissen Einsatz. Der Dealer teilt danach die Karten an jeden Spieler aus. Zunächst erhält jeder Spieler und der Dealer eine offene Karte. Beim zweiten Mal Austeilen erhält jeder Spieler jeweils wieder eine offene Karte, der Dealer jedoch eine geschlossene. Der Dealer fragt anschließend nach der Reihe jeden Spieler, ob diese eine weitere Karte ziehen möchten oder nicht ("Hit" oder "Stand"). Die Zahlen auf den Karten eines Spielers werden aufaddiert und ergeben eine Summe. Diese Summe darf die Zahl 21 nicht überschreiten, ansonsten ist das Spiel für den Spieler verloren und der Dealer bekommt seinen Einsatz. Alle Karten mit keiner Zahl (Bube, Dame, König) gelten als eine 10er Karte. Das Ass ist sowohl die 1 als auch die Zahl 11. Wenn ein Ass mit der Zahl 11 jedoch die Kartensumme 21 überschreitet, ist sie automatisch die Zahl 1. Bei dem Spiel geht es darum den Dealer mit seiner eigenen Kartensumme zu überbieten. Geschieht dies bekommt der Spieler den gesamten Einsatz vom Dealer übergeben. Der Dealer hat am Ende, wenn alle Spieler mit ihren Karten zufrieden sind auch die Möglichkeit Karten zu ziehen, zuvor aber auch seine geschlossene Karte zu öffnen, um die Spieler zu überbieten.

2. Wahl eines Lernverfahrens

Bei dem Spiel Blackjack gibt es zwei Handlungen, die man bei einer Partie ausführen kann: Hit oder Stand. Diese Handlungen müssen in Abhängigkeit von den Karten, die man selbst und die der Dealer gezogen hat, getroffen werden. Hierbei würde sich Q-Learning als Lernalgorithmus gut für den Test eignen. Beim Q-Learning werden Entscheidungen aufgrund von **Policies** getroffen, die während dem Lernen optimiert werden. Während π eine Policy beschreibt, ist $\pi*$ die optimale Policy. Auf Grundlage dieser Policies werden Aktionen durchgeführt. **Aktionen** sind alle möglichen Handlungen, die während einem Spiel getroffen werden können.

$A = \{a_1, \ldots, a_n\}$ **Menge aller Aktionen**

Jede Policy besitzt seinen eigenen Zustand. Ein **Zustand** beschreibt eine mögliche Spielkombination/Spielzug.

$Z = \{z_1, \ldots, z_n\}$ **Menge aller Zustände**

Auf dieser Grundlage wird eine Tabelle mit allen Zuständen auf der horizontalen Spalte und auf der Vertikalen die Aktionen eines Spieles erstellt. Bei einer Aktion wechselt sich auch gleichzeitig der Zustand von z auf z'. Dies nennt man auch **Übergangs-Funktion**. Bei einer Aktion muss auch eine Belohnung erfolgen, sobald ein Spiel noch nicht als verloren gilt. Dies könnte so erfolgen, dass ein gewonnenes Spiel besonders hoch belohnt wird und jeder Zustand, der das Spiel nicht zum Verlust führt, schwach belohnt wird. Bei einem Verlust erfolgt eine Bestrafung. Dies geschieht in einer Belohnungs-Funktion. Sobald eine optimale Policy nach n-Lernzügen erreicht wurde.

3. Blackjack und Q-Learning

Für das Erstellen einer Q-Table benötigt man zunächst alle nötigen Zustandswerte und Aktionen. Dafür wird das Spiel auf das Nötigste gekürzt. Zum einen besitzt der Spieler anfangs keine zwei real existierenden Karten, sondern hat nur eine Kartensumme dieser zwei Karten. Zum anderen gibt es kein Deck. Somit kann der Agent auch keine Karten zählen, zumal die Kartensummen und alle anderen Werte zufällig erzeugt werden. Den allerersten Wert aller Zustände belegt die Kartensumme, die der Spieler anfangs

ausgeteilt bekommt. Da zwei Asse als 12 gezählt werden, ist das niedrigste Kartenpaar die 4, da sie aus den Zahlen 2 und 2 gebildet werden kann. Das höchste Kartenpaar ist demnach die 21. Der zweite Wert ist die sichtbare Karte des Dealers. Dieser geht von 2 bis 11. Der dritte Wert zeigt ob der Spieler ein Ass hat oder nicht. Dieser ist demnach ein Boolean. Es gibt nur zwei verschiedene Aktionen, die ein Spieler treffen kann, "Hit" oder "Stand". Mit den Informationen kann man nun eine Q-Table erstellen und die einzelnen Zellen mit Zahlen befüllen.

		Aktionen	
	Q-Tabelle	**Hit**	**Stand**
Zustände	...	...	...
	(14,5,false)	1.0	-1.0
	(14,6,true)	0.0	0.0
	(14,6,false)	1.0	1.0
	(14,7,true)	-1.0	0.0
	(14,7,false)	-1.0	-1.0
	(14,8,true)	0.0	-1.0
	...	...	...

Bild 1: Q-Table mit zufälligen Werten

Alle einzelnen Werte in den Zellen werden je nach Trainingseinheit angepasst, sodass der Spieler eine optimale Strategie entwickelt, das Spiel so gut wie möglich zu gewinnen. Die Summe aller im Q-Table enthaltenen Kombinationen von Zuständen und Aktionen beträgt (18 * 10 * 2) * 2 = 720 (18 Kartensummen des Spielers, 10 offene Karten des Dealers, 2 Möglichkeiten für ein existierendes Ass und diese Werte mal die möglichen Aktionen also "Hit" und "Stand"). Das heißt, dass die Werte in allen 720 Zellen während des Trainings optimiert werden. Beim Testen wird dann für einen zufälligen Zustand die Aktion mit dem höheren Wert genommen. Im Hinblick auf die Wahl der besten Option wird die Epsilon-Greedy Strategie benutzt. Bei der **Epsilon-Greedy** Strategie wird zunächst ein Epsilon ε bestimmt. Dieser bekommt beispielsweise den Wert ε = 0.10. Der Agent erforscht erstmal die beste Option, um sie (1 – ε) zuzuweisen. Danach wird eine Zufallsvariable $0 \leq p \leq 1$ erstellt und je nachdem welche Zufallszahl auftaucht wird entweder die beste Option, was zu 90% eintrifft, oder eine beliebig andere Option, das nur zu 10% aller Fälle eintrifft, ausgewählt. Das Epsilon kann zur Trainingszeit je nach Bedarf konstant sein, mit der Zeit abnehmen oder anderweitig an die Gegebenheiten angepasst werden.

4. Test und Ergebnisse

Mit einer Einheit von 1.000.000 Wiederholungen wird die Künstliche Intelligenz trainiert, bis eine gute Policy durch das Training entsteht. Beim Testen werden 5.000 Wiederholungen durchgeführt und alle gewonnenen, verlorenen und zu unentschieden gespielten Spiele aufgezeichnet. Außerdem werden die Belohnungen auf einem Diagramm mit aufgenommen, damit man die Fortschritte des Agenten einsehen kann. Dabei sind folgende Ergebnisse entstanden:

```
total:
von 5000 Spielen
gewonnen: 2060 verloren: 2544 unentschieden: 396
prozentual:
gewonnen 0.412 verloren 0.5088 unentschieden 0.0792
```

Bild 2: Ergebnis des ersten Testverlaufes mit optimierter Policy

Interessant ist hierbei, dass der Spieler das Spiel öfter verloren hat. Damit man schlussendlich beweisen kann, dass der Agent etwas erlernt hat, da es sich hierbei um ein Spiel mit Zufallszahlen handelt, muss man den Test mit einer Policy testen, die zufällige Zahlen enthält und nicht trainiert wird. Damit kann man sehen welche Unterschiede es gibt zwischen einer optimierten und nicht optimierten Policy und ob dieser wirklich zwingend gravierend ist. Bei einer Policy mit zufällig erzeugten Werten bildete sich das folgende Resultat:

```
total:
von 5000 Spielen
gewonnen: 1372 verloren: 3439 unentschieden: 189
prozentual:
gewonnen 0.2744 verloren 0.6878 unentschieden 0.0378
```

Bild 3: Ergebnis des zweiten Testverlaufes mit zufälliger Policy

Man sieht, dass die Policy mit den zufällig erzeugten Werten um fast das doppelte schlechter abschneidet als die optimierte Policy. Nach zehn durchgeführten Tests hat die optimierte Policy einen Mittelwert von 2071,6.

X1	1	2	3	4	5	6	7	8	9	10	$\bar{x}$
X2	2024	2111	2057	2132	2040	2041	2079	2069	2081	2082	2071,6

Bild 4: Tabelle von zehn durchgeführten Tests mit optimierter Policy

Führt man denselben Test mit der zufällig erzeugten Policy durch, sieht man auch, dass der Mittelwert stark von dem optimierten abweicht. Um genau zu sein gewinnt der Agent mit der optimierten Policy im Schnitt ungefähr 500 Spiele mehr als die zufällig erzeugte Policy.

X1	1	2	3	4	5	6	7	8	9	10	$\bar{x}$
X2	1505	1448	1307	1506	1432	1478	1479	1612	1284	1531	1458,2

Bild 5: Tabelle von zehn durchgeführten Tests mit zufälliger Policy

Belohnungen, die auf den Diagrammen aufgezeichnet wurden, haben auch bei der Gegenüberstellung einen gravierenden Unterschied. Diese werden nämlich so gewertet, dass gute Zustände, die zum Gewinnen führen, den Wert 1.0, normale Zustände, die das Spiel weiterspielen lassen, den Wert 0.0 und schlechte Zustände, die zum Verlieren führen, den Wert -1.0 bekommen. Das Diagramm zeigt die Zu- und Abnahme der Belohnungen nach n-Spielzügen. Bei dem Diagramm mit der optimierten Policy kann man eine gewisse Linearität erkennen,

jedoch fällt die Linie bzw. fallen die Werte nicht so steil wie bei dem Diagramm mit der zufällig erzeugten Policy. Bei diesem Diagramm führen die Werte fast parallel auf beiden Achsen, nur dass die x-Achse um eine Hochzahl höher ist als die y-Achse. Belohnungen, die auf den Diagrammen aufgezeichnet wurden, haben auch bei der Gegenüberstellung einen gravierenden Unterschied. Diese werden nämlich so gewertet, dass gute Zustände, die zum Gewinnen führen, den Wert 1.0, normale Zustände, die das Spiel weiterspielen lassen, den Wert 0.0 und schlechte Zustände, die zum Verlieren führen, den Wert -1.0 bekommen. Das Diagramm zeigt die Zu- und Abnahme der Belohnungen nach n-Spielzügen. Bei dem Diagramm mit der optimierten Policy kann man eine gewisse Linearität erkennen, jedoch fällt die Linie bzw. fallen die Werte nicht so steil wie bei dem Diagramm mit der zufällig erzeugten Policy. Bei diesem Diagramm führen die Werte fast parallel auf beiden Achsen, nur dass die x-Achse um eine Hochzahl höher ist als die y-Achse. Auf dem anderen Diagramm sieht man, dass die Policy unangepasst ist, zumal die Linie ohne eine wirkliche Schwankung nach unten führt und sie viel steiler fällt als die Kurve der optimierten Policy.

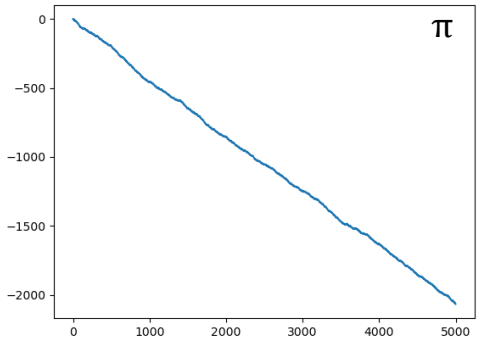


Bild 6: Graph einer Belohnungsfunktion einer zufällig erzeugten Policy

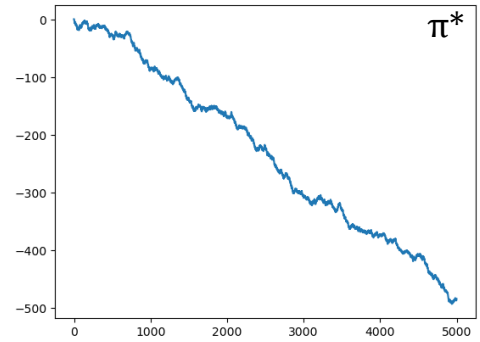


Bild 7: Graph einer Belohnungsfunktion einer optimierten Policy

5. Fazit

Im Großen und Ganzen schneidet der Agent mit der optimierten Policy besser ab, ist jedoch nicht in der Lage den Dealer zu überbieten, auch wenn zufällige Zustände erzeugt werden. Dies heißt auch, dass der Dealer im Schnitt besser abschneidet als der Spieler. Macht dieses Ergebnis Blackjack zu einem unfairen Spiel? Diese Frage kann man nicht leicht beantworten, da es ein gewisses Glück erfordert und die Zufallswerte einen entscheidenden Faktor ausmachen. Auch sind Häufungen nicht auszuschließen und diese können das Spielgeschehen stark beeinflussen. Genauso spielt die Auswahl eines anderen Lernalgorithmus eine große Rolle, die wohlmöglich eine bessere Strategie finden könnte. Jedoch kann man in diesem Beispiel im Durchschnitt sagen, dass der Spieler weniger gewinnt, wenn man davon ausgeht, dass das Blackjack mit denselben Regeln, wie sie in dieser Ausarbeitung definiert wurden, gespielt wird.

REFERENCES

[1] Roulette with Monte Carlo Policy, (Stand: 24.10.2018). https://mc.ai/learning-machine-learning-roulette-with-monte-carlo-policy/ [05.08.2020]

[2] Patrick Dammann, Einführung in das Reinforcement Learning. https://hci.iwr.uni-heidelberg.de/system/files/private/downloads/541645681/dammann_reinfocement-learning-report.pdf [05.08.2020]

Space Invaders mit Bestärkendes Lernen spielen

Thorsten Jost
thorsten.jost@hof-university.de
Hochschule Hof
Hof an der Saale, Germany

ZUSAMMENFASSUNG

Ziel dieser Arbeit ist es, mit Hilfe von Keras und OpenGym, ein neuronales Netz aufzubauen und dann so zu trainieren, dass der Agent im Atari Spiel "Space Invaders" die Aktionen so wählt, das die höchstmögliche Belohnung einer Runde des Spiels erzielt wird. Zuerst werden die genutzten Frameworks vorgestellt und deren Nutzen erläutert. Es werden Grundlegende Themen in Reinforcement Learning behandelt, sowie Anpassungen erläutert die notwendig sind, um das genannte Ziel zu erreichen.

1 EINLEITUNG

1.1 Atari 2600

Atari 2600 ist die erfolgreichste Spielkonsole in den 70er und 80er, die im Jahre 1977 von dem Unternehmen Atari eingeführt wurde. Die Spielekonsole wurde durch die große Anzahl der zu Verfügung stehenden Spielauswahl (mehr als 1200 Spiele) zur erfolgreichsten Spielkonsole zu dieser Zeit. Noch heute stehen viele Emulatoren zur Verfügung, die es ermöglichen, Atari 2600 Spiele auf dem Computer zu spielen, vgl. [9].

1.2 SpaceInvaders

Space Invarders ist ein „Shoot'em-up-Computerspiel“, dass für die Atari 2600 Konsole im Jahre 1978 entwickelt worden ist. Die Aliens sind in sechs Spalten und je sechs Zeilen angeordnet und bewegen sich von links nach rechts und von rechts nach links. Wird eine Seite erreicht, bewegen sich alle Aliens eine Position nach unten. Sollte ein Alien die Spielfigur erreichen, ist die Runde des Spiels verloren. In zufälligen Abständen kommen auch am oberen Spielfeldrand Aliens, die abgeschossen werden können, um Bonuspunkte zu erhalten. Das Ziel ist es, durch die Abschüsse der „Aliens“ eine bestmögliche Punktzahl zu erreichen, ohne von den Aliens selbst getroffen zu werden. Die bestmögliche Punktzahl kann durch Abschüsse aller erscheinenden Bonusziele sowie der Aliens erreicht werden, ohne zu Sterben.

2 FRAMEWORKS

2.1 Open AI Gym

Open AI Gym ist eine Bibliothek das Speziell für Entwicklung und Reproduzierbarkeit von bestätigendes Lernen Algorithmen entwickelt und zur Verfügung gestellt wird. Mit OpenAI Gym lassen sich verschiedene Umgebungen importieren, wie die Atari Umgebung. Diese stellt Spiele von der Atari 2600 Konsole zur Verfügung. [7]

2.2 Atari Umgebung

Bereitgestellt wird die Atari Umgebung von der Arcade Learning Environment (ALE). Diese stellt eine öffentliche Schnittstelle, mit hunderten von Atari 2600 Spiele, zur Verfügung. Das Hauptaugenmerk der ALE liegt darin, mit Atari 2600 Spiele, bestärkendes Lernen Algorithmen zu erstellen und das Ergebnis zu bewerten.
Die Atari Umgebung bietet je Spiel zwei Versionen an. Eine Version stellt die verschiedenen Zustände als Bild, von der Spielumgebung, zur Verfügung. Diese Version findet in dieser Arbeit Verwendung. Die andere Version stellt die Zustände direkt vom Arbeitsspeicher der emulierten Atari 2600 Konsole zur Verfügung, vgl. [1].
Die Umgebung stellt vier wesentliche Variablen zu jeder durchgeführten Aktion zur Verfügung.

Observation. Repräsentiert den aktuellen Status der Umgebung mit Pixel-Informationen das ein Bild des Spiels repräsentiert.

Done. Ob das aktuelle Spiel beendet (true) ist oder nicht.

Reward. Gibt die Belohnung der aktuellen Beobachtung an.

Info. Gibt zusätzliche Informationen der Umgebung an, wie z. B. das vorhandene Leben der Spielfigur.

Die Umgebung wird nicht als Bild, sondern als Vector mit Pixel Daten bereitgestellt. Wird die Umgebung initialisiert oder eine Action durchgeführt, gibt die Atari Umgebung, die genannten vier Variablen zurück. Diese Informationen können dann verarbeitet werden und für das neuronale Netz zum Trainieren verwendet werden. Insgesamt bietet die Atari Umgebung 18 mögliche Aktionen an.

Für das Spiel Space Invaders stehen aber nur sechs der möglichen 18 Aktionen zur Verfügung.
- Noop, Fire, Right, Left,
- Rightfire, Leftfire

2.3 Keras

Keras ist eine modernes Deep Learning Open Source Framework, das seit TensorFlow Version 2.0 fester Bestandteil von TensorFlow geworden ist. Keras kann aber auch weiterhin in anderen Bibliotheken frei verwendet werden. Mit Keras ist es möglich, in kürzester Zeit, eigene Modelle und Experimente, im Bereich Deep Learning, zu erstellen und zu nutzen, vgl. [4].

3 THEORI

3.1 Reinforcement Learning

Bestärkendes Lernen (Reinforcement Learning) ist ein Teil von maschinellem Lernen, dass darum handelt, das ein Agent in einer Umgebung die höchstmögliche kumulierte Belohnung mit gewählten Aktionen erhält. Die Herausforderung ist es, das Netz dynamisch zum Spielgeschehen zu trainieren, mit dem Problem, das die Aktion

nicht sofort bewertet werden kann, da die Belohnungen erst später im Spielverlauf erlangt werden.

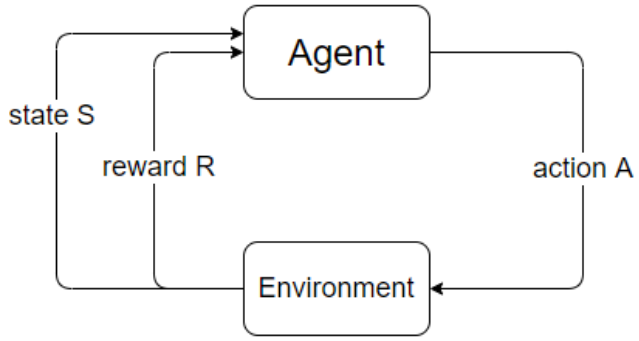


Abbildung 1: Ablauf von Reinforcement Learning

Der Agent führt eine Aktion in der Umgebung aus. Die Umgebung gibt daraufhin einen neuen Zustand, der von der ausgeführten Aktion ausgelöst wurde, zurück. Zusätzlich wird auch die Belohnung, die durch die Aktion erzeugt wurde, zurück an den Agenten übertragen, vgl.[5]. Diesen Ablauf stellt die Abbildung 1 grob dar.

3.2 Q-Learning

Um das Ziel, Erreichung einer kumulierten höchstmöglichen Belohnung, zu erzielen, kann Q-Learning verwendet werden. Der Name Q-Learning kommt von der Q-Funktion Q(s,a), die den erwarteten Nutzen aus einem Status (s) und gewählten Aktion (a) zurückgibt. Um dieses Ziel zu erreichen, wird eine Optimale „Action-Value-Funktion (Q(s,a)) definiert, die der Bellmann Optimalitätsprinzip folgt.
Die Q-Value Function gibt an, dass pro Zustand und pro möglicher Aktion, ein Q-Value gespeichert wird. Dieser Q-Value besteht aus dem Reward vom Zustand (s) und aus der diskontierten Belohnung zukünftiger Zustände, wie Abbildung 2 zeigt. Vgl. [5].

$$Q(s, a) = r + \gamma \max_{a'}(Q(s', a'))$$

Abbildung 2: Q-Learning

Diese bedeutet, wenn die Q-Value für den Zustand (s) für den nächsten Schritt für alle möglichen Actions (a) bekannt ist, wähle die optimale Action, die den erwarteten Wert maximiert.
Das Optimalitätsprinzips von Bellman gibt an, dass bei Optimierungsproblemen jede optimale Lösung aus optimalen Teillösungen besteht.
In Q-Learning wird eine sogenannte Q-Table erstellt. Diese enthält zu jeder Beobachtung den Q-Value zur jeweiligen ausführbaren Aktion. Dies ist ausreichend, in sehr einfach gehaltenten Umgebungen, mit sehr wenigen Beobachtungen und möglichen Aktionen. Sobald die Anzahl der Beobachtungen und Aktionen steigt, wird es sehr schwierig dies noch mit einer Tabelle zu handhaben.
Diese Tabelle wird in komplexen Umgebungen durch ein Neuronales Netz ersetzt. Das Ergebnis daraus ist Deep Q-Learning.

3.3 Discount Factor

Wie zukünftige Belohnungen für den aktuellen Zustand bewertet werden, gibt der Discount Factor an. Der Discount Factor ist ein Wert zwischen 0 und 1. Ist der Wert zu niedrig, besteht die Gefahr der Kurzsichtigkeit des Netzes. Da die Belohnungen, die nah in der Zukunft liegen, stark bewertet werden und Belohnung die weit in der Zukunft liegen, sehr schwach bewertet werden. Ziel ist es jedoch eine langfristige hohe kumulierte Belohnung zu erreichen, somit wird der Discount Factor sehr hoch angesetzt, vgl. [8].
In dieser Arbeit ist der Discount Factor mit 0,99 festgelegt.

3.4 Greedy

Am Anfang des Trainings werden zufällige Aktionen (Exploration) gewählt. Im Laufe des Trainings soll das Netz vermehrt auf das erlangte Wissen zurückgreifen und durch Ausnutzung (Exploitation) seines Wissens die Aktionen gezielt wählen. Diese Anforderung wird mit Greedy gelöst.
Der Faktor Greedy ist zum Start auf 1 festgelegt und wird nach jeder Episode, um einen festgelegten Faktor, reduziert. Der minimale Wert darf nicht 0 betragen sondern muss > 0 sein. Den Greedy soll verhindern, das der Agent im späteren Verlauf des Trainierens nicht immer die gleichen Aktionen durchführt und durch zufällige Aktionen in neue Situationen kommt, vgl. [8].

3.5 Deep Q-Network

Deep Q-Network baut auf Q-Learning auf und ersetzt die Q-Table durch ein neuronales Netz mit mehreren Schichten. Dieses Konstrukt erlaubt auch den Umgang mit komplexen Strukturen, wie zum Beispiel einem Bild. Das Neuronale Netz schätzt die entsprechende Aktions-Wert Funktion, welche die entsprechende berechnete Belohnung für die gemachte Aktion der jeweiligen Beobachtung zurückgibt, vgl. [6].

3.6 Double Deep Q-Network

Der konzeptionelle Aufbau eines DQN, bringt Probleme mit sich. In *Deep RL with Double Q-learning* [3] wird „Überoptimismus“ dadurch begründet, dass die Auswahl sowie Bewertung der Aktionen von den gleichen Gewichten eines Neuronalen Netzes durchgeführt wird. Dies macht es wahrscheinlicher überschätzte Werte auszuwählen, was zu überoptimistischen Wertschätzungen führt. Dies löst Double DQN mit Trennung der Gewichte zur Auswahl und Bewertung der Aktionen. Hierfür wird ein zweites Neuronales Netz in die vorhandene Umgebung eingebaut. Wie Abbildung 2 zeigt, gibt das Deep Q-Network die gewählte Aktion zurück und das zweite Deep Q-Network nimmt die gewählte Aktion und gibt die Q-Values zurück.

$$Q(s,a) = r(s,a) + \gamma Q(s', argmax_a Q(s',a))$$

Target Network berechnet Q Value DQN Network wählt Aktion

Abbildung 3: Double Deep Q-Network

4 UMSETZUNG

4.1 Pooling Layer

In der klassischen Bildklassifizierung geht es um die Detektion der Objekte auf einem Bild. Im Bereich der Gesichtserkennung ist es nicht wesentlich, ob das Gesicht auf einer exakten Position liegt, je nach Einsatzszenario, sondern das Gesicht muss erkannt werden als solches.
Pooling Layer verwerfen Informationen, wie einen Teil des räumlichen Standortes des Objektes, durch Verwendung z. B. von Maxpooling Schichten. Im Maxpooling werden die höchste Werte einer Matrix verwendet und die restlichen Werte verworfen. Zum Beispiel wird aus einer 2x2 Matrix, der höchste Wert gepickt und die anderen drei Werte verworfen. Was dazu führt das ein Teil des räumlichen Standortes verworfen werden kann. Durch Verwendung von Pooling reduziert sich somit auch die Bildgröße und Speicherbedarf des Bildes an die Umgebung. Da keine exakte Position notwendig ist, spielt der Verlust dieser Informationen keine größere Rolle in der Gesichtserkennung.

Würde man Pooling Layer speziell in dieser Arbeit, in der Atari Umgebung, verwenden, würde es in bestimmten Zuständen dazu kommen, das das Model nicht die Möglichkeit hat, die Positionen der Gegner, sowie eigenen Spielfigur sehr exakt interpretieren zu können, vgl. [8]. Das neuronale Netz sieht zwar die Schüsse, der gegnerischen Spielfiguren, aber nicht, ob der Schuss direkt auf die Spielfigur zugeht oder minimal daneben liegt. Somit hätte das Netz dementsprechend die Chance, falsch auf diesen Zustand zu reagieren. Aus diesem Grund wird der Einsatz von Pooling Layer verzichtet, um die Chance der Fehlinterpretierung zu minimieren.

4.2 Preprocessing

Mit den rohen Atari Bildern (210x16 Pixel) zu arbeiten, würde den Aufwand für das Neuronale Netz stark erhöhen, ohne jeglichen Nutzen dafür zu generieren. Deshalb werden die Bilder bearbeitet bevor diese ins Netz geladen werden.
Die Auflösung der Zustände wird auf 84x84 Pixel reduziert, sowie auf Graustufen reduziert, vgl. [6].
Bestimmte Bereiche der Zustände spielen für das Lernen des Netzes keine Rolle, wie zum Beispiel im Spiel Space Invaders der obere Punktebereich, da die Punkte von der Atari Umgebung geliefert werden. Das Bild wird so zugeschnitten, dass Bereiche, die für das Trainieren nicht relevant sind, entfernt werden, um die Effizienz des Lernens zu erhöhen.
Dies muss je nach Umgebung separat entschieden werden und kann nicht auf alle Atari 2600 Spiele verallgemeinert werden.
Ein weiterer Teil des Preprocessing ist die Art der Speicherung der einzelnen Zustände. Im Standard haben die Zustände ein Format von float64, was den Speicherbedarf eines Zustandes 8x erhöhen würde, als ein Datentyp int8. Das würde zur Folge haben das der Speicherbedarf von 100.000 Zuständen ca. 70 GB betragen würde, was in den Arbeitsspeicher geladen werden müsste, vgl. [2]. Um diesen massiven Bedarf einzuschränken, werden die Bilder im Format int8 konvertiert und in den Replay Memory gespeichert.

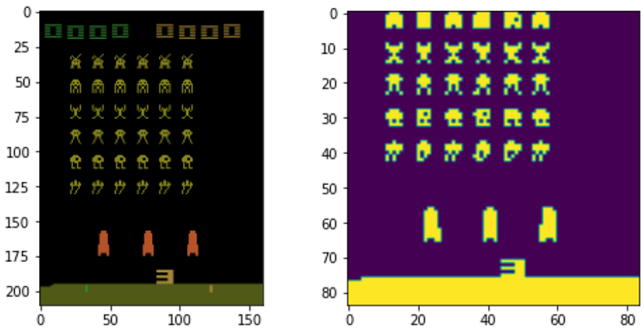


Abbildung 4: Verlgeich vor und nach Bearbeitung

Abbildung Links zeit das Bild des Spiels komplett unbearbeitet an. Die Abbildung rechts zeigt das Ergebniss der Bearbeitung von Bild links.

4.3 Replay Memory

Da eine getätigte Aktion die nächsten Zustände beeinflusst, korrelieren die nachfolgenden Beobachtungen stark. Wenn als Beispiel das Netz dem Agenten vorgibt, sich zu bewegen, sind nachfolgende Beobachtungen von dieser Vorgabe stark beeinflusst bzw. sehr ähnlich. Würde somit das Netz nach jeder Beobachtung trainieren, würde das zu Schleifen führen. Da das Netz die gleichen Bilder zum trainieren vorgesetzt bekommt.

Um dieses Problem, des ineffizienten Lernens, zu lösen, wird ein Erinnerungsspeicher (Replay Memory) eingeführt. Dieser Speicher speichert zu jeder Beobachtung den Zustand(s), Belohnung (r), getätigte Aktion (a) sowie den nächsten Zustand (s‘), somit entsteht folgende Tupel: « s, a, r, s‘ ».
Der Speicher speichert bis zu einer vorgegebenen Größe und löscht dann die älteste Tupel wieder, um die neue Tupel (s,a,r,s‘) aufzunehmen. Hier ist die Auswahl der Größe entscheidend für das Trainieren des Netzwerkes. Wird der Speicher zu klein definiert, ist die Wahrscheinlichkeit höher, dass für das Trainieren korrelierte Beobachtungen herangezogen werden. Zu Groß darf der Speicher auch nicht gewählt werden, da der Speicher einen massiven Speicherbedarf für den Arbeitsspeicher auslöst.
Zum Trainieren wird ein Minispeicher (Batch), mit zufällig gewählten Beobachtungen, aus dem Speicher gefüllt. Durch die zufällige Ziehung der Beobachtungen wird die Wahrscheinlichkeit zusammenhängende Beobachtungen zu greifen, abhängig von der Größe des Batches, verringert, vgl. [5].

Es existieren mehrere Arten von „Memory‘s“. In dieser Arbeit wird der sequenzielle Speicher genutzt. Dieser Speicher speichert alles in der eingehende Reihenfolge und macht keine weiteren Änderungen an den gespeicherten Beobachtungen.
Eine weitere Art von Speicher hängt an die einzelnen Beobachtungen noch Gewichte der Beobachtungen als zusätzliche Daten dran. Das Netz wird dann nur mit den Beobachtungen, mit der höchsten Gewichtung, trainiert.

4.4 Frame Skipping

Nach der wissenschaftlichen Arbeit *Playing Atari with Deep Reinforcement Learning*[5] ist es nicht erforderlich für jeden Zustand eine Aktion durchführen zu lassen, da sich nicht in jeder Zustand eklatant ändert. Aus diesem Grund wird Frame Skipping in die Umgebung eingebaut.

Mit der Funktion Frame Skipping wird die letzte Aktion, die vor den zu überspringenden Zustand ausgeführt wurde, für den nachfolgenden Zustand wiederholt.

Die Funktionalität ist in den Standard der Atari Umgebung integriert. Die Anzahl der zu überspringenden Bilder wird mit der Version des Spieles gesteuert. Beispiel „SpaceInvaders-V4“ überspringt immer drei Zustände. Die Version-0 überspringt die Bilder zufällig von 1-4 Zuständen. Es wurde sich gezielt gegen die V4 entschieden, da die Laser der Aliens und Spielfigur nicht konstant sichtbar sind. Deshalb kann es sein, das die Laser dann durch das "Frame Skipping"nicht gesehen werden und das Netzwerk nicht dementsprechend reagieren kann.

4.5 Frame Stacking

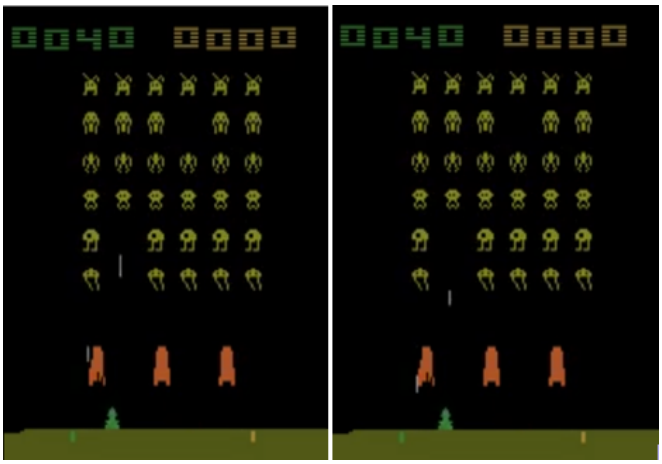


Abbildung 5: aufeinanderfolgende Zustände der Umgebung

Betrachtet man nur die linke Spielumgebung in der Abbildung 5, zeigt es deutlich, das ein Zustand alleine nicht aussagekräftig ist, was den aktuellen Zustand des Spieles beschreiben könnte. Die Geschosse der Aliens können nach oben oder nach unten verlaufen. Die gegnerischen Spielfiguren können sich nach links oder rechts bewegen.

Deshalb benötigt das neuronale Netz eine Abfolge an Bilder. Die geschieht durch gestapelte Bilder (Stacked Frames), um die aktuelle Beobachtung richtig einschätzen zu können, vgl. [5]. Der Stapel hat eine Größe von 4 Bilder, um genügend Informationen der nachfolgenden Bilder generieren zu können. Zum Start jeder Episode wird der Stapel mit dem ersten Bild, das von der Methode reset() geliefert wird, gefüllt. Speziell zum Start einer Episode wird das erste Bild dreimal kopiert und in den Stapel geladen. Für jede weiteren Schritt wird das aktuelle Bild in den Stapel gepackt und das älteste Bild wird aus dem Stapel gelöscht.

Der Stapel wird mit jeder Beobachtung im Replay Memory gespeichert und vom neuronalen Netz zum trainieren verwendet.

4.6 Reward Function

Zur Optimierung des Deep Q-Networks wird eine Belohnungsfunktion eingebaut, die die erspielten Belohnungen optimiert. Folgende Kriterien werden zur Berechnung herangezogen.

Figur stirbt. Wenn die Figur stirbt werden immer eine bestimmte Menge an Punkten von der Belohnung abgezogen.

Anzahl Beobachtungen pro Leben. Stirbt die Figur zu schnell in einem Spiel, werden Punkte von der Belohnung abgezogen. Überlebt die Figur hingegen eine gewisse Zeit, werden Bonuspunkte gutgeschrieben.

Anzahl Punkte pro Leben. Erzielt das Netzwerk nur wenige Punkte, werden eine definierte Anzahl an Punkten von der Belohnung abgezogen. Erzielt die Figur aber eine gewisse Menge an Punkten, werden Bonuspunkte der Belohnung hinzugefügt.

5 DURCHFÜHRUNG

5.1 Model

Verwendet wurden im neuronalen Netz, Convoultion Layer, wie Abbildung 6 den Aufbau des Neuronalen Netzes zeigt. Es werden drei Convolution Layer eingesetzt zur Bildklassifizierung. Verwendet man stattdessen aber die ATARI RAM Version würde man die Convolution Layer durch Fully Connected Layer ersetzen.
Am Ende ist ein Fully Connected Layer mit 6 Neuronen implementiert. Diese sechs Neuronen stellen die sechs möglichen Aktion in der Spielumgebung dar.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 84, 84, 4)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 32)        3232
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 64)          18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 9, 9, 64)          4160
_________________________________________________________________
flatten_1 (Flatten)          (None, 5184)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               2654720
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 3078
=================================================================
Total params: 2,683,686
Trainable params: 2,683,686
Non-trainable params: 0
_________________________________________________________________
None
```

Abbildung 6: Aufbau des Neuronalen Netzes

5.2 Ablauf

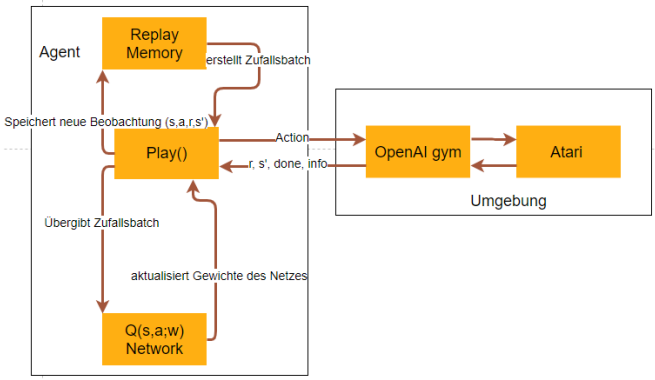


Abbildung 7: Ablauf der Umgebung mit den Agenten

Abbildung 7 zeigt wie das Programm in groben Zügen funktioniert. Am Anfang des Trainierens wird der Replay Memory mit Daten gefüllt ohne das Netz selbst zu trainieren. Würde man sofort mit dem Trainieren beginnen, wäre der Speicher nur sehr klein und die Wahrscheinlichkeit, korrelierte Daten zu greifen, wäre sehr hoch. Deshalb wird mit "Warm-Up"Phasen der Speicher bis zu einer gewissen Menge an Beobachtungen gefüllt.

Ist die Warm-Up Phase beendet, beginnt der Agent die Aktionen entweder selbst zu wählen oder per Zufall zu entscheiden, je nach dem wie groß Greedy ist. Danach wird die Aktion an die Umgebung übergeben und diese gibt eine Beobachtung mit erzeugter Belohnung sowie den nächsten Zustand zurück.
Die zurückgegebene Belohnung wird zusätzlich durch eine eigene definierte Belohnungsfunktion angepasst. Wie in Kapitel 4.6 *Reward Function* erläutert, wird die Belohnung durch bestimmte Faktoren wie die Anzahl der getätigten Schritten sowie erlangte Belohnungen, gut oder schlecht bewertet. Die Summe der einzelnen Bewertungen werden dann dementsprechend der eigenlichen Belohnung, von der Umgebung erhalten, abgezogen oder hinzufügt.
Die Beobachtung wird mit der ausgeführten Aktion, vorherigen Zustand, Belohnung und neuen Zustand, in den Replay Memory gespeichert. Das wird so lange wiederholt bis kein Leben mehr vorhanden ist und somit die Episode beendet ist.
Ist die Episode zu Ende, wird eine zufällig gewählte Menge an Beobachtungen aus dem Replay Memory in einen MiniBatch geladen. Der erzeugte MiniBatch wird dann zum Trainieren des neuronalen Netzes verwendet. Das Netzwerk passt daraufhin die Gewichte an. Dann beginnt die nächste Episode und der Durchlauf startet von vorn.

5.3 Google Colab

Da die Verwendung des Atari Frameworks unter Windows offiziell nicht unterstützt wird, wäre das Framework nur durch Anpassungen und zusätzlicher Software lauffähig geworden, aber dies nicht garantiert. Aus diesem Grund wurde das Projekt in der Umgebung Google Colab aufgebaut und ausgeführt.

Google Colab ist eine Entwicklungsumgebung die es erlaubt einfach und schnell Programmcode in Python zu schreiben, auszuführen und zu dokumentieren. Es sind viele Frameworks, wie TensorFlow und Keras, von Grund auf integriert und müssen nicht separat integriert werden.

Um Google Colab optimal für das Projekt zu nutzen, wurde hierfür auf die Pro Version hochgestuft, um auf die schnellere GPU P100 Zugriff zu bekommen. In der kostenlose Variante ist ein Verbindungsabbruch nach 12h Verwendung der Laufzeit eingebaut, in der Pro Version findet der Verbindungsabbruch erst nach 24h statt. Zusätzlich erhöhte die Pro Version den verfügbaren Arbeitsspeicher auf 26 GB, was es erlaubte, den Erinnerungsspeicher zu erhöhen.

Google Colab erlaubt auch einen direkten Zugriff auf die Google Drive Umgebung. Mit diesem Zugriff ist man in der Lage die Zwischenstände des Trainings abzuspeichern, da zufällige Zusammenbrüche sowie Abbrüche der Umgebung nicht selten waren.

Die Umgebung, genannt Notebook, kann jederzeit aus der Google Colab Umgebung exportiert und heruntergeladen werden. Dieses Notebook kann dann in vergleichbaren Umgebungen verwendet werden, die die gleiche Funktionalität beinhalten wie Google Colab.

6 ERGEBNISS

Abbildung 8 zeigt das Ergebnis von 25000 trainierten Episoden in der Google Colab Umgebung. Das Training wurde für ca. 20 Stunden durchgeführt.

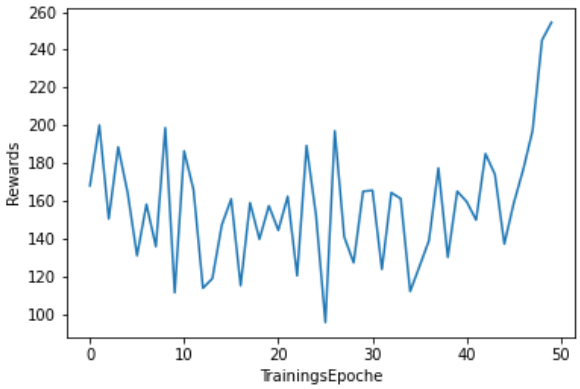


Abbildung 8: Durchschnittliche Belohnung pro Epoche

Die Ergebnisse wurden in Epochen zusammengefasst. Eine Epoche enthält 500 Episoden.

Um den Wachstum des Lernens zu zeigen, wurde der Durchschnitt mit den Belohnungen der Episoden gebildet. Wie man auf der Abbildung 8 sehen kann, ist ein langsamer Anstieg der durchschnittlichen Belohnungen in den Epochen zu beobachten.

Lässt man sich die Spielumgebung im jetzigen Zustand anzeigen,

sieht man das der Agent auf die Umgebung reagiert. Am Anfang des Trainings war dies nicht der Fall, entweder wurde die Spielfigur sofort von den gegnerischen Spielfiguren getroffen oder der Agent verharrte in der Ecke bis er dort abgeschossen wurde.
Mit 50 trainierten Epochen reagiert der Agent schon deutlich häufiger auf die feindlichen Aliens sowie deren Bewegungen und Schüssen. Die, in verschiedenen Abständen, erscheinenden Bonusziele wurden vom Agent auch vermehrt getroffen, was den Anstieg der Belohnungen mit verursacht hat.

7 FAZIT

Durch die modernen Frameworks wie OpenGym und Keras stehen eine Vielzahl von Funktionen zur Verfügung, die zum Aufbau und zur Verwendung eines eigenes Neuronales Netz verwendet werden können und die Erstellung des Netzes erleichtern.

Trotz anfänglicher Schwierigkeiten in der Studienarbeit, die Entwicklungsumgebung sowie Ausführungsumgebung funktionsfähig zu machen, konnte mit mehrfachen Anpassungen und vielen Stunden Training eine Umgebung mit einem Neuronalen Netz aufgebaut werden, das über die Trainingsdauer gewisse Lernerfolge erzielt.

Um das Ziel, regelmäßig über 400 Punkte pro Episode zu generieren, zu erreichen, muss das Netz weiter stark trainiert und zum teil verbessert werden.

Es zeigt, trotz starker Hardware (GPU P100, die auf solche Operationen optimiert/konzipiert ist) und einigen Unterstützungsfunktionen wie Frame-Skipping, das Trainieren sehr viel Zeit beansprucht.

LITERATUR

[1] Marc Bellemare. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. https://arxiv.org/abs/1207.4708. Accessed: 27.07.2020.
[2] Adrien Lucas Ecoffet. 2017. Beat Atari with Deep Reinforcement Learning! (Part1:DQN). https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26. Accessed: 01.06.2020.
[3] Silver Hasselt, Guez. 2015. Deep Reinforcement Learning with Double Q-learning. https://arxiv.org/abs/1509.06461. Accessed: 01.07.2020.
[4] Keras. [n.d.]. Keras API Reference. https://keras.io/api/. Accessed: 01.05.2020.
[5] Mnih. 2013. Playing Atari with Deep Reinforcement Learning. https://arxiv.org/pdf/1312.5602.pdf. Accessed: 01.05.2020.
[6] Mnih. 2015. Human-level control through deep reinforcement learning. https://deepmind.com/research/publications/human-level-control-through-deep-reinforcement-learning. Accessed: 01.05.2020.
[7] OpenAI. [n.d.]. Gym Documentation. https://gym.openai.com/docs/. Accessed: 01.05.2020.
[8] Niloy Purkait. 2019. Neural Networks with Keras. https://learning.oreilly.com/library/view/hands-on-neural-networks/9781789536089/. Accessed: 01.07.2020.
[9] Michael Vogt. 2020. ATARI 2600. http://www.atari-computermuseum.de/2600.htm. Accessed: 25.07.2020.

Abbildungsverzeichnis

Erstellung eines KI-Chatbots

zur Erlernung musiktheoretischer Vokabeln

Sophie Dehmer
Medieninformatik
Hochschule Hof
Hof Deutschland
sophie.dehmer@hof-university.de

ABSTRACT

Wie es im Schwimmen die Abzeichen „Seepferdchen, Bronze, Silber, Gold“ gibt, gibt es bei Blasorchestern die Musikerleistungsabzeichen „Kleine Stimmgabel, D1, D2, D3“ für die Grundausbildung und noch einige mehr für die Weiterbildung.

Typischerweise werden (zumindest in Hessen, dort habe ich meine Ausbildungen gemacht) zweimal im Jahr so genannte "D-Wochen" der Landesmusikjugend angeboten. In dieser Woche lernen Kinder die benötigte Theorie und Praxis für die Arbeit in Blasorchestern oder Spielmannszügen. Am Ende der Woche legen sie eine theoretische und eine praktische Prüfung ab.

Auf der Webseite des Bayerischen Blasmusikverbandes heißt es folgendermaßen zum Musikerleistungsabzeichen:

> „Die Musikerleistungsabzeichenprüfungen des Bayerischen Blasmusikverbandes sind das Erfolgsmodell im Bereich der Blasmusik. Jährlich absolvieren ca. 5.000 Musikerinnen und Musiker die Theorie und/oder Praxisprüfungen in D1 (Bronze), D2 (Sil- ber), D3 (Gold). Diese wurden vor einigen Jahren um die Juniorprüfung als Einstieg ergänzt.“ [1]

„Die Musikerleistungsabzeichenprüfungen des Bayerischen Blasmusikverbandes sind das Erfolgsmodell im Bereich der Blasmusik. Jährlich absolvieren ca. 5.000 Musikerinnen und Musiker die Theorie und/oder Praxisprüfungen in D1 (Bronze), D2 (Silber), D3 (Gold). Diese wurden vor einigen Jahren um die Juniorprüfung als Einstieg ergänzt.“ [1]

1 Allgemeines

1.1 Idee

Um Kindern das Lernen der Vokabeln zu erleichtern und da mittlerweile ja viele Dinge digitalisiert werden, wollte ich einen Chatbot erstellen, der auf die Eingaben der Kinder reagieren und ihnen die gestellte Frage beantworten kann.

1.2 Umsetzung

In der folgenden Studienarbeit wird ein Chatbot umgesetzt, der auf einer Tutorialreihe für einen Python Chatbot mit Deep Learning von „Tech with Tim“ [11] basiert.

1.3 Vokabeln

Folgende Vokabeln werden dem Chatbot zum Lernen zur Verfügung gestellt:

Lento, Grave, Adagio, Larghetto, Andante, Andantino, Moderato, Allegretto, Vivo, Allegro, Vivace, presto, prestissimo, retardant, ritenuto, rallentando, Accelerando, Stringendo, ad libitum, senza tempo, Piu mosso, Pocco meno, A Tempo, Pianissimo, piano, mezzopiano, mezzoforte, forte, Fortissimo, Crescendo, Decrescendo, Diminuendo, Sforzato, Fortepiano, Non legato, Legato, Staccato, Simile, Portato, Tenuto, Sostenuto, marcato, martellato, Phrasierung, Phrasierungsbogen, Wiederholungszeichen, Faulenzer, Da Capo, Da Capo al Fine

2 Umsetzung

2.1 Allgemeines

Der Chatbot wurde in der Entwicklungsumgebung PyCharm [5] umgesetzt und anschließend in ein Jupyter Notebook [6] auf Google Colaboratory[4] übertragen.

2.2 Verwendete Module

Folgende Vokabeln werden dem Chatbot zum Lernen zur Verfügung gestellt:

2.2.1 Tensorflow[10]

In diesem Projekt wird die Tensorflow 1.x-Version verwendet, da die neue 2.x Version das Modul tflearn noch nicht unterstützt.

2.2.2 hanTa[12]

Dieses Modul ermöglicht die Lemmatisierung in der deutschen Sprache. „Das Lemma ist der Ein- trag oder das Stichwort in

einem Wörterbuch (Lexikon, Enzyklopädie). Man bezeichnet es sowohl als Grundform eines Wortes als auch als Zitier- oder Grundform eines Lexems.“[7]

2.2.3 TFLearn[2]

TFLearn ist ein Deep Learning Modul, das für Tensorflow eingesetzt werden kann.

2.2.4 NLTK[9]

NLTK wird in Python für die Verwendung mit menschlicher Sprache eingesetzt.

2.2.5 json

Um die Vokabeln in diesem Projekt einlesen zu können, wird JSON und dementsprechend auch das dafür vorgesehene Modul importiert.

2.2.6 numpy[8]

Numpy wird für verschiedene Bereiche in „Science“ und „Engineering“ verwendet.

2.2.7 pickle[3]

Pickle ist ein Modul, dass binäre Protokolle in Python zur „Serialisierung“ und „De-Serializing“ zur Verfügung stellt.

2.2.8 random

Damit der Bot am Ende unterschiedliche Antworten auf dieselben Fragen geben kann, werden sie mit Hilfe des Moduls random ausgewählt.

3 Dokumentation

3.1 Rohdatenverarbeitung

Das Wörterbuch (die Json-Datei) wird Wort für Wort durchgegangen, eingegebene Wörter werden auf ihren Wortstamm zurückgeführt und Satzzeichen entfernt.

Alle Wörter werden in Tokens umgewandelt - in logisch zusammenhängende Einheiten. Außerdem wird die Sprache auf Deutsch festgelegt.

3.2 Modell erstellen und speichern

3.2.1 Erstellen der Layer des Neuronalen Netzes

```
#Graph-Daten werden zurückgesetzt
tensorflow.reset_default_graph()

#Input-Shape wird festgelegt, das vom Model
erwartet wird, in diesem Fall die Länge von
"training"
net = tflearn.input_data(shape=[None,
len(training[0])])

#Fully connected Layer wird zum Neuronalen
Netzwerk hinzugefügt mit 8 Neuronen für die
Hidden Layer
net = tflearn.fully_connected(net, 8) #1.
Hidden Layer mit 8 Neuronen
net = tflearn.fully_connected(net, 8) #2.
Hidden Layer mit 8 Neuronen

#Ermöglicht es, Wahrscheinlichkeiten für
jeden Output zu bekommen
#Softmax geht Liste durch und gibt
Wahrscheinlichkeiten an (Output des
Neuronalen Netzwerks).
net = tflearn.fully_connected(net,
len(output[0]), activation="softmax")

#Das Regression-Layer wird in TFLearn
verwendet um auf das zur Verfügung gestellte
Input eine Regression (Vorhersage)
anzuwenden
net = tflearn.regression(net)

#DNN steht für Deep Neural Network,
spezifiziert die Art von Netzwerk und
speichert das Netzwerk in der Variable model
model = tflearn.DNN(net)
```

In diesem Code-Teil wird der komplette KI-Teil des Bots erstellt. Es besteht aus 4 Fully Connected Layer (Input (in diesem Fall Training), zwei Hidden Layer mit jeweils 8 Neuronen und Output (in diesem Fall die Tags bzw. Labels, die in der JSON Datei festgelegt wurden).

Bevor der Bot eine Vorhersage (Regression) auf das Netz treffen kann, berechnet er die mit der Softmax-Funktion die Wahrscheinlichkeiten für die Tags/Labels (= Output) und legt anschließend den Typ des Netzes auf DNN (Deep Neural Netwerk) fest.

Da die Vorhersagen des Models immer so um die 0.98 lagen (mit 1000 Epochen), habe ich mich dazu entschieden 2 Hidden Layer zu verwenden und diese nicht aufzustocken, da der Bot auch mit diesen relativ zuverlässig arbeitet.

3.3 Ausgaben des Codes

Um den Code nachvollziehbar zu gestalten, gibt er ab und zu aus, was er gerade bearbeitet hat.

Beispielsweise werden die Labels und Wörter, die er aus der JSON-Datei ausgelesen hat, in se- parate Listen gespeichert, auf ihren Wortstamm zurückgeführt und alphabetisch sortiert.

3.3.1 Ausgabe von „print(labels)“

Da die Roh-Daten der Labels sowohl mit Groß- als auch Kleinbuchstaben beginnen, werden hier erst die Wörter nach dem Alphabet sortiert, die einen großen Anfangsbuchstaben haben, danach die klein geschriebenen Wörter.

[’A Tempo’, ’Accelerando’, ’Adagio’, ’Allegretto’, ’Allegro’, ’Andante’, ’Andantino’, ’Crescendo’, ’Da Capo’, ’Da Capo al Fine’, ’Decrescendo’, ’Dimi- nuendo’, ’Faulenzer’, ’Fortepiano’, ’Fortissimo’, ’Grave’, ’Larghetto’, ’Largo’, ’Legato’, ’Lento’, ’Moderato’, ’Non legato’, ’Phrasierung’, ’Phrasierungsbogen’, ’Pianissimo’, ’Piu mosso’, ’Pocco meno’, ’Portato’, ’Sforzato’, ’Simile’, ’Sostenuto’, ’Staccato’, ’Stringendo’, ’Tenuto’, ’Vivace’, ’Vivo’, ’Wiederholungszeichen’, ’ad libitum’, ’begruessung’, ’forte’, ’marcato’, ’martellato’, ’mez- zoforte’, ’mezzopiano’, ’piano’, ’prestissimo’, ’presto’, ’rallentando’, ’ritardanto’, ’ritenuto’, ’senza tempo’, ’verabschiedung’]

3.3.2 Ausgabe von „print(words)“

Alle enthaltenen Worte im Wörterbuch in Kleinbuchstaben umgewandelt und im Alphabet geordnet.

[’a’, ’accelerando’, ’ad’, ’adagio’, ’al’, ’allegretto’, ’allegro’, ’and’, ’andantino’, ’capo’, ’cre- scendo’, ’da’, ’decrescendo’, ’diminuendo’, ’faulenz’, ’fin’, ’fort’, ’fortepiano’, ’fortissimo’, ’geht’, ’grav’, ’gut’, ’hallo’, ’heißt’, ’hey’, ’hi’, ’ich’, ’larghetto’, ’largo’, ’legato’, ’lento’, ’libit’, ’mar- cato’, ’martellato’, ’meno’, ’mezzofort’, ’mezzopiano’, ’moderato’, ’mosso’, ’non’, ’phrasierung’, ’phrasierungsbogen’, ’pianissimo’, ’piano’, ’piu’, ’pocco’, ’portato’, ’prestissimo’, ’presto’, ’quit’, ’rallentando’, ’ritardanto’, ’ritenuto’, ’senz’, ’sforzato’, ’simil’, ’sostenuto’, ’spiel’, ’staccato’, ’strin- gendo’, ’tag’, ’tempo’, ’tenuto’, ’vivac’, ’vivo’, ’was’, ’wie’, ’wiederholungszeich’]

3.3.3 Lernmodell speichern

Dem Model werden alle notwendigen Daten übergeben. Training steht hier für den Input, also die Worte, die der Bot lernen soll, Output steht für die Tags/Labels die im Zusammengang damit verwendet wurden (beispielsweise "Begrüßung" oder die einzelnen Vokabeln). n_epoch = 1000 legt fest, dass das Model die vorgelegten Daten 1000 mal durchgehen muss um sie zu lernen. Zu Testzwecken hatte ich die Epochen auf 100 runtergesetzt. Der Bot lernte zwar sehr schnell, verlor aber auch sehr viele Infos wieder und gab dementsprechend schlechte (oder gar keine) Antworten, da die accuracy sehr niedrig war. Ich lies den gleichen Code mit n_epoch = 100 mehrfach durchlaufen. Jeder Durchlauf hatte ein komplett unterschiedliches Ergebnis. Hier habe ich zwei verschiedene Runden abgebildet:

Training Step: 1999 | total loss: 1.99670 | time: 0.085s | Adam | epoch: 100 | loss: 1.99670 - acc: 0.3886 -- iter: 152/157 Training Step: 2000 | total loss: 1.96101 | time: 0.090s | Adam | epoch: 100 | loss: 1.96101 - acc: 0.3998 -- iter: 157/157

Training Step: 3999 | total loss: 0.42836 | time: 0.110s | Adam | epoch: 200 | loss: 0.42836 - acc: 0.9771 -- iter: 152/157 Training Step: 4000 | total loss: 0.39531 | time: 0.115s | Adam | epoch: 200 | loss: 0.39531 - acc: 0.9794 -- iter: 157/157

Am Ende wird das Model als "model.tflearn" abgespeichert.

```
model.fit(training, output,
n_epoch=1000, batch_size=8,
show_metric=True)
model.save("model.tflearn")
```

3.3.3 Training des Models

Das Model gibt während es trainiert seine Genauigkeit, Schritte, den Lernverlust und wie lange es für jeden Schritt gebrauch hat, aus. Dies ist die Ausgabe dazu:

Training Step: 19999 | total loss: 0.00199 | time: 0.109s | Adam | epoch: 1000 | loss: 0.00199 - acc: 1.0000 – iter: 152/157

Training Step: 20000 | total loss: 0.00197 | time: 0.114s | Adam | epoch: 1000 | loss: 0.00197 - acc: 1.0000 – iter: 157/157

Diese Ausgaben sind bei einem beispielhaften Durchlauf die letzten zwei Trainingsrunden des Bots, bevor der Benutzer mit ihm chatten kann.

QUELLEN

[1] Bayerischer Blasmusikverband. Musikerleistungsabzeichen. url: https : / / www . bbmv - online.de/Musikerleistungsabze.250.0.html.
[2] TFLearn Contributors. TFLearn: Deep learning library featuring a higher-level API for TensorFlow. url: http://tflearn.org/.
[3] Python Software Foundation. pickle — Python object serialization. url: https : / / docs . python.org/3/library/pickle.html.
[4] Google. Google Colaboratory. url: https://colab.research.google.com/notebooks/intro.ipynb.
[5] Jetbrains. PyCharm. url: https://techwithtim.net/tutorials/ai-chatbot/part- 1/.
[6] Project Jupyter. Jupyter Notebook. url: https://jupyter.org/.
[7] Lemma (Lexikographie) – Wikipedia. (Accessed on 08/01/2020). url: https://de.wikipedia.
org/wiki/Lemma_%28Lexikographie%29.
[8] Inc. NumFOCUS. The fundamental package for scientific computing with Python. url:
https://numpy.org/.
[9] Edward Loper Steven Bird Ewan Klein. Natural Language Toolkit. url: http : / / www .
nltk.org/index.html.
[10] Tensorflow. TensorFlow: Large-scale machine learning on heterogeneous systems. url:
https://www.tensorflow.org/.
[11] Tech with Tim. Python Chat Bot Tutorial - Chatbot with Deep Learning. 2019. url: https:
//techwithtim.net/tutorials/ai-chatbot/part-1/.
[12] Christian Wartena. Hannover Tagger: Morphological Analysis and POS Tagging. url: https: //pypi.org/project/HanTa/.

Music Recommendation Service

Musikempfehlungen mit k-Means und Spotify

Jan Degen
Sommersemester 2020
Informatik, Hochschule Hof
Hof, Bayern, Deutschland

ABSTRAKT

Im Rahmen dieser Arbeit wurde eine Anwendung entwickelt, die mittels k-Means Clustering und Spotify benutzerspezifische Musikempfehlungen generiert.
Dabei werden die meisten gehörten Songs des Benutzers aus seinem Spotify Profil abgerufen und ein k-Means Clustering durchgeführt. Im Nachgang werden die Cluster identifiziert, welche den Musikgeschmack den des Nutzers am besten treffen.
Für die Empfehlungen dienen, die bereits in der Datenbank abgelegten Songs, welche durch das Clustering eingeordnet und wenn sie einen der bevorzugten Cluster angehören, dem Benutzer vorgeschlagen werden.

CCS CONCEPTS

• Computing methodologies~Machine learning~Machine learning approaches~Factorization methods~Non-negative matrix factorization

• Computing methodologies~Machine learning~Learning paradigms~Unsupervised learning~Cluster analysis

• Human-centered computing~Visualization~Visualization application domains~Information visualization

• Applied computing~Arts and humanities~Sound and music computing

KEYWORDS

Recommendation Service, Recommendation Engine, Music, K-Means, Clustering, Unsupervised Learning

1 Einführung

Machine Learning findet mittlerweile viele Anwendungen in unterschiedlichen Bereichen. So können mittels Klassifikation von Patientendaten Diagnosen getroffen werden oder in Computerspielen die nicht-menschlichen Gegner vorbereitet und trainiert werden. Recommendations Services oder Engines werden verwendet, um Voraussagen für bestimmte Daten zu treffen. Beispielsweise können Blumen anhand ihrer Eigenschaften klassifiziert werden. Eine andere Variante ist eine Prognose mittels Clustering. Beim Clustering ist vorher nicht bekannt wie das Ergebnis aussieht, sondern es wird nach ähnlichen Gruppen gesucht. Anhand dieser Gruppen können Voraussagen für weitere Daten getroffen werden.
So ist es möglich, Film oder Musikempfehlungen anhand eines Clusterings zu treffen, in dem auf den bisherigen gewerteten Daten ein Clustering durchgeführt wird. Anschließend wird identifiziert, welcher Cluster dem Benutzer gefällt und weitere Elemente gesucht, die in diesen Cluster fallen.

Anwendungsarchitektur

Die gesamte Applikation ist in Komponenten aufgeteilt, um eine einfache und übersichtliche Anwendung bereitzustellen.

Die recommendation API ist ein ASP.NET REST-Service, welcher die Grundfunktionen für die Musikempfehlung liefert. Hier werden REST-Endpunkte bereitgestellt, um Informationen über einen Benutzer von Spotify abzufragen und auf Basis dieser Daten eine Empfehlung zu generieren.

Die mittels ML.net trainierten Modelle werden auf dem Dateisystem für jeden Benutzer abgelegt, um so eine schnelle Wiederverwendung zu ermöglichen.

Das recommendation Frontend basiert ebenfalls auf ASP.NET und dient der Kommunikation mit dem End User. Hier werden die Authentifizierung für Spotify erfragt und die Ergebnisse aus der recommendation API dem Benutzer dargestellt.

Der Identity Server 4 verwaltet die Benutzerkonten und Autorisierung zur Verwendung der APIs und Frontends, außerdem werden die Zugangstoken für die Spotify API hier abgelegt, sodass andere Applikationen dieses bei Bedarf abfragen können.

Verfügbare Daten

Alle Daten für die Erstellung der Cluster werden von Spotify abgerufen. da die Spotify API ein Zugriffslimit besitzt, werden diese Daten in einer Datenbank abgelegt, um die Zugriffe auf die Spotify API zu minimieren.

Für jeden Song stellt Spotify allgemeine Informationen bereit, die beim Abruf des Songs mitgeliefert werden.

- Title
- Album
- Artist
- Genre
- DurationsMs

Das Genre wird zu diesem Zeitpunkt nicht verwendet, da hierfür eine komplexere Auflösung mit den Artisten nötig wäre. Könnte aber in Zukunft für eine Optimierung des Clusterings verwendet werden.

Des Weiteren stellt Spotify eine Songanalyse bereit, die den Song hinsichtlich verschiedener Merkmale bewertet und eine Grundlage für ein Clustering bietet.

Accousticness	Ein Konfidenzmaß von 0,0 bis 1,0, das angibt, ob die Spur akustisch ist. 1,0 steht für ein hohes Konfidenzniveau.[11]
Mode	Modus gibt die Modalität (Dur oder Moll) eines Stücks an, die Art der Skala, von der sein melodischer Inhalt abgeleitet ist. Dur wird durch 1 und Moll durch 0 dargestellt.[11]
Danceability	Tanzbarkeit beschreibt, wie geeignet ein Stück zum Tanzen ist, basierend auf einer Kombination von musikalischen Elementen wie Tempo, Rhythmusstabilität, Taktstärke und allgemeiner Regelmäßigkeit. Ein Wert von 0,0 ist am wenigsten tanzbar und 1,0 ist am besten tanzbar.[11]
Energy	Energie ist ein Maß von 0,0 bis 1,0 und stellt ein Wahrnehmungsmaß für Intensität und Aktivität dar. Typischerweise fühlen sich energetische Spuren schnell, laut und geräuschvoll an. Zu den Wahrnehmungs-merkmalen, die zu diesem Attribut beitragen, gehören der Dynamikbereich, die wahrgenommene Lautsärke, das Timbre, die Anstiegsrate und die allgemeine Entropie.[11]
Instrumentalness	Sagt voraus, ob ein Titel keinen Gesang enthält. "Ooh"- und "aah"-Klänge werden in diesem Zusammenhang als instrumental behandelt. Rap- oder Spoken-Word-Tracks sind eindeutig "vokal". Je näher der Instrumentalitätswert bei 1,0 liegt, desto größer ist die Wahrscheinlichkeit, dass der Track keinen Gesangsanteil enthält. Werte über 0,5 sollen Instrumentalspuren darstellen, aber die Wahrscheinlichkeit ist höher, je näher der Wert bei 1,0 liegt.[11]
Liveness	Erkennt die Anwesenheit eines Publikums in der Aufnahme. Höhere Liveness-Werte stellen eine erhöhte Wahrscheinlichkeit dar, dass der Titel live aufgeführt wurde.[11]
Loudness	Die Gesamtlautheit einer Spur in Dezibel (dB). Die Loudness-Werte werden über den gesamten Track gemittelt und sind nützlich zum Vergleich der relativen Lautsärke von Tracks.[11]
Speechiness	Speechiness erkennt das Vorhandensein von gesprochenen Worten in einer Spur. Je ausschließlich sprachähnlicher die Aufnahme (z.B. Talkshow, Hörbuch, Lyrik), desto näher an 1.0 liegt der Attributwert. Werte über 0.66 beschreiben Tracks, die wahrscheinlich vollständig aus gesprochenen Worten bestehen. Werte zwischen 0,33 und 0,66 beschreiben Tracks, die sowohl Musik als auch Sprache enthalten können, entweder in Abschnitten oder geschichtet, einschließlich solcher Fälle wie Rap-Musik. Werte unter 0,33 stehen höchstwahrscheinlich für Musik und andere nicht sprachähnliche Tracks.[11]
Tempo	Das geschätzte Gesamttempo eines Tracks in Schlägen pro Minute (BPM). In der musikalischen Terminologie ist das Tempo die Geschwindigkeit oder das Tempo eines bestimmten Stücks und leitet sich direkt von der durchschnittlichen Taktdauer ab.[11]
TimeSignature	Eine geschätzte Gesamtzeit-Signatur einer Spur. Die Taktart (Meter) ist eine Notationskonvention, die angibt, wie viele Schläge in jedem Takt enthalten sind.[11]
Valence	Ein Maß von 0,0 bis 1,0, das die musikalische Positivität beschreibt, die von einem Titel vermittelt wird. Tracks mit hoher Valenz klingen positiver während Tracks mit niedriger Valenz negativer klingen.[11]

Des Weiteren kann für einen Spotify User seine meistgehörten Lieder in drei wählbaren Zeitspannen abgefragt werden.

Short Term Mid Term Long Term

Die zurückgelieferten Songs enthalten ein von Spotify kalkuliertes benutzerspezifisches Ranking, welches für jeden User und Song abgespeichert wird.
Dieses Ranking dient zur Identifizierung von Songs, die dem Benutzer gefallen.

Benutzerspezifische Clustermodelle

Die recommendation API erstellt für jeden Benutzer ein eigenes Cluster Modell. Das Cluster Model basiert auf den meistgehörten Songs des Benutzers und verwendet die Features:

- Accousticness
- Instrumentalness
- Mode
- Liveness
- Danceability
- Valence

Einige weitere Features müssen vor der Verwendung über eine MinMax-Normalisierung vorbereitet werden.

- Loudness
- Tempo
- DurationMs
- TimeSignature

k-Means Clustering

Das Clustering gehört zur Kategorie des unüberwachten Lernens. Dabei werden in einem gegebenen Datensatz Gruppen ähnlicher Elemente gebildet.
Beim k-Means Clustering muss die Anzahl der gewünschten Cluster vorher definiert werden. Während der Initialisierung werden dann die entsprechende Anzahl an Clusterzentren zufällig aus den Datenpunkten gewählt und alle Datenpunkte dem ihm nächstem Zentrum zu gewiesen. In den weiteren Iterationen werden die Clusterzentren so verschoben, dass die Varianz der Cluster sich am wenigsten erhöht und die Cluster sich stabilisieren. [3, 4, 6]

ML.net unterstützt 3 verschiedene Varianten des Initialisierungsalgorthmuses für k-Means Clusterings:

- k-Means++
- k-Means YingYang
- Random

Alle drei Varianten wurden in der recommendation API getestet.

k-Means++

Die Wahl der Clusterzentren bei k-Means++ für die Startpositionen ist nicht zufällig, sondern folgt einem festen Schema bei der Initialisierung. Diese läuft dabei in den folgenden Schritten ab:

1. Wähle das erste Element zufällig aus dem Datensatz
2. Wähle ein weiteres Element x aus dem Datensatz, wobei für die Wahrscheinlichkeit der Wahl für das Element gilt $\frac{D(x)^2}{\sum_{x \in X} D(x)}$. Wobei $D(x)$ die kürzeste Distanz des Datenpunkts zum nächsten Zentrum darstellt
3. Wiederhole 2. bis die entsprechende Anzahl von Zentren erstellt wurde

Die Wahl der Startpunkte hängt damit von ihrer Entfernung zum am nächsten liegenden Clusterzentrum ab. Je weiter weg ein Datenpunkt vom Clusterzentrum ist, desto höher ist die Wahrscheinlichkeit, dass dieser Punkt als Zentrum gewählt wird. Da dieses Verfahren in der Regel bereits nach wenigen Schritten konvergiert, ist die Ausführung des Algorithmus deutlich schneller als der normale k-Means Algorithmus. [4]

k-Means YingYang

Im Gegensatz zu anderen k-Means Algorithmen berechnet die Yinyang Variante nicht die Distanz zwischen allen Datenpunkten, sondern versucht, durch die Anwendung von Filtern, die benötigten Berechnungen zu minimieren.
Diese Filter basieren auf der Dreiecks-Ungleichheit und bestehen aus dem globalen Filtern, Gruppenfiltern und lokalen Filtern.
Das globale Filtern erkennt, ob ein Punkt einen Cluster verändern würde, in dem ein Vergleich mit der globalen Ober- und Untergrenzen stattfindet.
Die Gruppenfilter stellen eine Verallgemeinerung des globalen Filters dar und teilt die Cluster bei jeder Iteration in Gruppen ein. Für jede dieser Gruppen wird ebenfalls eine Ober- und Untergrenze berechnet.
Nach dem Gruppenfilter könnte ein Datenpunkt ein neues bestes Zentrum sein. Statt nun wie beim herkömmlichen k-Means die Distanzen aller Datenpunkte zu berechnen, werden nur bestimmte Punkte berechnet und der zweit weit entfernteste Punkt gewählt. [1, 12]

Random

Bei diesem Verfahren folgt dem ursprünglichen k-Means Algorithmus und initialisiert die Clusterzentren zufällig. Dabei kann es aber zu schlechten Annäherungen zum optimalen Cluster kommen. [9]

Mit den verfügbaren Varianten wurden jeweils für unterschiedliche Personen Empfehlungen erstellt. Dabei schnitt Random am schlechtesten ab. k-Means++ lieferte nach den Erfahrungen der Nutzer die besten Empfehlungen, während k-Means Yinyang mit deutlich niedrigerem Speicherverbrauch auffiel.

Erstellung der benutzerspezifischen Clustermodelle

Die recommendation API lädt alle Songs, die mit dem User in der Datenbank assoziiert sind und verwendet diese Daten für das Training des Clustermodells.

Nach dem Training ist es durch das Framework ML.net möglich die ermittelten Clusterzentren aus dem Model abzurufen, jedoch ist es nicht möglich die ermittelten Clustereinstufung für die Trainingsdaten abzurufen. Daher werden die Trainingsdaten im Nachhinein dem Model für eine Clustereinordnung eingespeist. So kann für jeden Song der zugewiesenen Cluster abgerufen werden und ermöglicht es so eine Auswahl zu treffen, welcher Cluster verwendet werden kann, um Empfehlungen zu identifizieren.

Eine der größten Problematiken war, die korrekte Anzahl an Clustern zu wählen mit dem das Model erstellt werden soll. Da die Songs sich von Benutzer zu Benutzer unterscheiden, konnte hier keine Konstante ermittelt werden, stattdessen wurde ein Vorgehen entwickelt, welches verschiedene Anzahlen von Clustern verwendet und anhand des Davies Bouldin Index bewertet.
Der Davies Bouldin Index ermittelt die Streuung innerhalb der Cluster und die Abstände zwischen den Clustern. [8, 10]
Das Verfahren startet mit drei Clustern und iteriert bis eine theoretische Verteilung pro Cluster über alle Daten 5% beträgt. Dabei wird als Optimum die Variante mit dem niedrigsten Davies Bouldin Index gewählt. Dies wird in neun Epochen wiederholt, um die Variante mit der besten Trennung der Cluster zu erhalten.
Der Speicherverbrauch dieser Variante liegt jedoch deutlich höher, als bei der herkömmlichen Methode, was einen Einsatz in einer produktiven Umgebung mit vielen gleichzeitigen Zugriffen deutlich erschwert.

Auswahl der Favoriten Cluster

Für die Musikempfehlung ist es wichtig zu wissen, welche Eigenschaften ein Song besitzen muss, damit er dem Benutzer gefällt. Durch das Clustering wurden ähnliche Songs in Clustern zusammengefasst und es lassen sich diese Gruppen verwenden, um eine Empfehlung auszusprechen.
Die recommendation API ruft von Spotify zwar die meistgehörten Lieder ab, jedoch sind in dieser Auflistung meistens auch Songs mit einem niedrigen Ranking enthalten. Anhand des von Spotify gelieferten Rankings ist es möglich die Cluster zu identifizieren, die dem Benutzer verstärkt gefallen.
Die Cluster, welche für die Empfehlung relevant sind, werden als Favoritencluster markiert.

Es wurden zwei unterschiedliche Möglichkeiten für die Ermittlung eines Favoritencluster getestet.

1. Höchster Rankingmittelwert

Für jeden Cluster wird der Mittelwert des Rankings der enthaltenen Songs aus dem Trainingsdatensatz ermittelt. Der Cluster mit dem höchsten Mittelwert wird als favorisierter Cluster markiert.

2. Meisten Songs über Schwellwert

Für jeden Cluster wird ermittelt wie viele Songs mit einem Ranking höher oder gleich einem bestimmten Schwellwert enthalten sind. Der Cluster mit den meisten Songs, die diesen Schwellwert erreichen bzw. übersteigen wird als ein favorisierter Cluster markiert.

Fällt ein Song in einen dieser Favoritencluster, wird er dem Benutzer als Empfehlung vorgeschlagen. Die Entfernung des Songs zum Clusterzentrum wird in eine Punktzahl umgerechnet. Bei dieser Umwandlung gilt, je näher ein Song zu seinem Zentrum liegt, desto höher ist die erreichte Punktzahl.

Anhand dieser Punktzahl werden die Songs dann absteigend sortiert und ausgegeben.

Die Architektur der recommendation API würde es ermöglichen mehr als einen Favoritencluster zu markieren. Dabei gäbe es folgende Möglichkeiten, die jedoch nicht evaluiert worden sind:

- Kombination aus höchstem Mittelwert und meisten Songs über Schwellwert
- Ranking aus den Clustern mit den Songs über Schwellwert mit einer alternativen Punkteberechnung

Für den Trainingsdatensatz werden eine Reihe von Visualisierungen generiert.

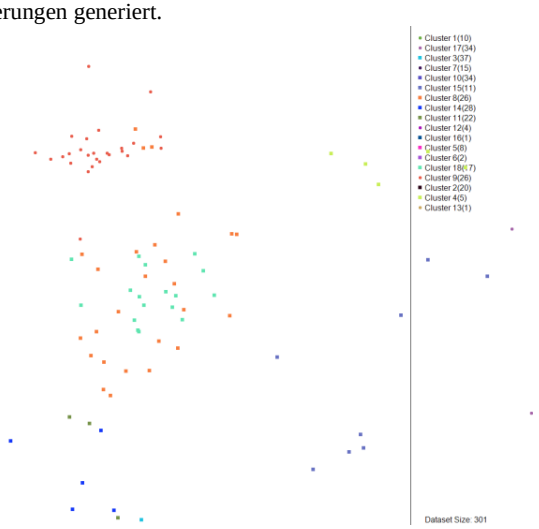


Abbildung 1: Trainingsvisualisierung - Gesamt

Abbildung 1 zeigt die Verteilung des Trainingsdatensatzes. Da es sich um einen Datensatz mit 12 Features handelt wurden die Datenpunkte in den 2-dimensionalen Raum übersetzt:

$$x = \sum_{Feature\ 0}^{Feature\ 5} Feature.Value$$

$$y = \sum_{Feature\ 6}^{Feature\ 11} Feature.Value$$

Für jedes Feature wird außerdem eine Visualisierung in Abhängigkeit des Rankings von Spotify erstellt. Abbildung 2 zeigt eine solche Visualisierung für das Feature Accousticness.

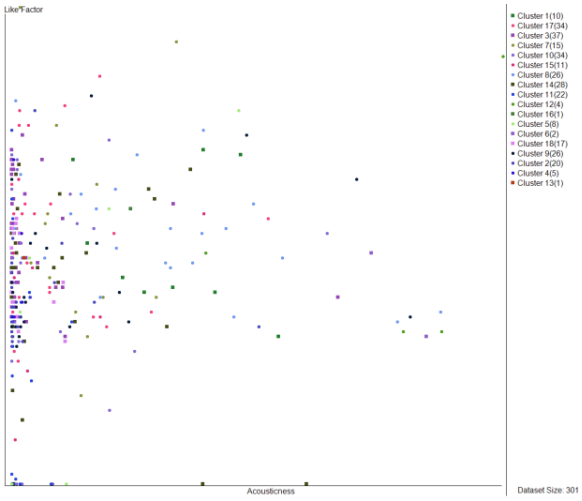


Abbildung 2: Trainingsvisualisierung - Accousticness

In den ersten Schritten der Entwicklung der Anwendung wurde ebenfalls eine One-Class Matrix Faktorisierung implementiert. Diese basiert auf dem von Spotify gelieferten Ranking.
Eine Matrix Faktorisierung basiert auf dem Prinzip

„Wenn Person A Produkt B mag und Person C Produkt B und D mag, gefällt Person A Produkt D möglicherweise“ [2, 5, 7]

In manchen Fällen ist mit dem aktuellen Datensatz nicht möglich eine Bewertung mittels Matrix Faktorisierung zu berechnen. In diesen Fällen werden 0 Punkte angenommen

Fazit

Die Musikempfehlungen des Models stimmen in vielen Fällen mit dem Musikgeschmack des Benutzers überein, könnten jedoch durch eine Verbesserung der Auswahl der Favoritencluster optimiert werden. Für eine weitere Optimierung könnten die Ergebnisse aus dem k-Means Clustering und der Matrix Faktorisierung gegeneinander validiert werden und eine gemeinsame Bewertung berechnet werden, um sowohl eine datenbasierte als auch eine kollaborativ-basierte Empfehlung zu generieren.
Ein großer Nachteil, welcher die aktuelle Lösung für produktive Systeme unbrauchbar macht, ist der massive Speicherverbrauch der Trainingsvariante bei der Erstellung der Clustermodelle.
Solange der Speicherverbrauch des Modeltrainings nicht reduziert wurde, kann das System nur mit vielen Hardwareressourcen umgesetzt werden.

References

[1] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, Sergei Vassilvitskii. Scalable K-Means++.

[2] Briacht. 2020. *Tutorial: Build a movie recommender - matrix factorization - ML.NET*. https://docs.microsoft.com/en-us/dotnet/machine-learning/tutorials/movie-recommendation. Accessed 3 August 2020.

[3] Clustergruppe. 2020. *K-Means*. https://www-m9.ma.tum.de/material/felix-klein/clustering/Methoden/K-Means.php. Accessed 3 August 2020.

[4] David Arthur, S. V. k-means++: The Advantages of Careful Seeding.

[5] Eric Le. 2017. *How to build a simple song recommender system*. https://towardsdatascience.com/how-to-build-a-simple-song-recommender-296fcbc8c85. Accessed 3 August 2020.

[6] Eugene Seo, H.-J. C. *MOVIE RECOMMENDATION WITH K-MEANS CLUSTERING AND SELF-ORGANIZING MAP METHODS*. Department of Computer Science, Korea Advanced Institute of Science and Techonology (KAIST). DOI=10.5220/0002737603850390.

[7] Mark Farragher. 2019. *Build A Product Recommender Using C# and ML.NET Machine Learning*. https://medium.com/machinelearningadvantage/build-a-product-recommender-using-c-and-ml-net-machine-learning-ab890b802d25. Accessed 3 August 2020.

[8] Microsoft. 2020. *ClusteringMetrics.DaviesBouldinIndex Property (Microsoft.ML.Data)*. https://docs.microsoft.com/en-us/dotnet/api/microsoft.ml.data.clusteringmetrics.daviesbouldinindex?view=ml-dotnet. Accessed 4 August 2020.

[9] Microsoft. 2020. *KMeansTrainer Class (Microsoft.ML.Trainers)*. https://docs.microsoft.com/en-us/dotnet/api/microsoft.ml.trainers.kmeanstrainer?view=ml-dotnet. Accessed 4 August 2020.

[10] Philipp Plöhn. 2014. Kategorisierung von Indizes zur Clustervalidierung.

[11] Spotify. 2020. *Spotify API Audio Features Object*. https://developer.spotify.com/documentation/web-api/reference/object-model/. Accessed 4 August 2020.

[12] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, Todd Mytkowicz. 2015. *Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup* 37. Department of Computer Science, North Carolina State University, France.