Object-Detection mit You Only Look Once (*YOLO*)

Einführung in die Objekterkennung mit YOLO sowie die Weiterentwicklung in den Versionen v2-v4

Mehmet Ali Daglioglu
Hochschule für Angewandte Wissenschaften Hof
Fakultät Informatik
Studiengang Wirtschaftsinformatik

Hof, 16. Februar 2021

Inhaltsverzeichnis

1 Grundlagen der Objekterkennung

Bereits seit Jahrzehnten wird das Forschungsgebiet der Objekterkennung, welches ein herausforderndes Teilgebiet der Bildverarbeitung darstellt, aktiv bearbeitet. Bereits im Jahr 1973 erforschten Fischler und Elschlager [10] ein Problem, das darin besteht, ein Objekt auf einem Foto mithilfe dessen Beschreibung zu finden. Sie beschreiben einen Teil der Lösung des Problems darin, ein Beschreibungsschema für ein zu erkennendes Objekt zu definieren sowie eine Metrik zu finden, die diese Erkennung bewertet. Später definieren Russakovsky et al. [42] das Problem der Objekterkennung folgendermaßen:

> „Object detection: Algorithms produce a list of object categories present in the image along with an axis-aligned bounding box indicating the position and scale of every instance of each object category.“ [42]

Ein Objekterkennungssystem hat somit die Aufgabe, Bounding Boxes zu generieren, die beschreiben, in welchem Bereich des Bildes ein relevantes Objekt abgebildet ist. Neben der Position wird außerdem ermittelt, um welches Objekt bzw. welche Objektklasse es sich handelt und mit welcher Sicherheit (Confidence) ein Objekt erkannt wurde [37]. In Abbildung 1 wird ein Beispiel für die Objekterkennung mit der jeweiligen Klasse ohne Confidences dargestellt.

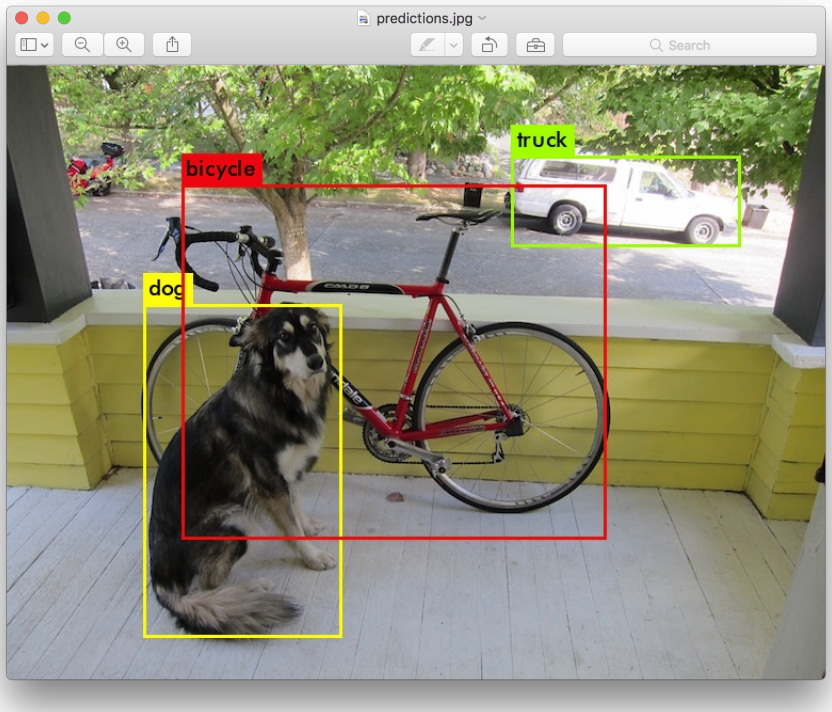


Abbildung 1: Beispiel für die Objekterkennung aus [34]

1.1 Objekterkennung mit Machine Learning Verfahren

In der Vergangenheit wurden bereits zahlreiche Methoden für die Lösung von Objekterkennungsproblemen entwickelt und verwendet. Eine wichtige Gruppe an Lösungen besteht in der Nutzung von maschinellen Lernverfahren, die sich in den letzten Jahren wachsender Beliebtheit erfreuen. Der Grund hierfür liegt unter anderem im Mooreschen Gesetz, das besagt, dass die verfügbare Rechenleistung sich etwa alle zwei Jahre verdoppelt [33]. Das Wachstum der verfügbaren Rechenleistung ermöglicht, rechenintensive Operationen, wie sie im maschinellen Lernen stattfinden, performant durchzuführen. Ein weiterer Treiber von maschinellen Lernverfahren ist die steigende Menge an verfügbaren Daten, die durch zahlreiche vernetzte Smart Devices generiert werden.

Ziel eines Objekterkennungssystems mit maschinellem Lernverfahren kann es sein, von diesen gesammelten Daten zu „lernen“ (*Supervised Learning*, vgl. S. 20ff. [12]). Elementar hierfür sind Datensätze, die aus Bildern bestehen, auf denen die „Lösung“ des Problems mit sogenannten Ground Truth Labels bereits gekennzeichnet ist (vgl. S. 20ff. [12]). Im Anwendungsfall der Objekterkennung würde dies bedeuten, dass die auf den Bildern des Datensatzes befindlichen Objekte in Form von Bounding Boxes gekennzeichnet sind. In Supervised Learning geht es somit grundsätzlich darum, mithilfe von Beispielen eine Funktion zu approximieren (vgl. S. 22 [12]). Übertragen auf den Anwendungsfall der Objekterkennung bedeutet das, dass die auf den Beispielbildern befindlichen Informationen abstrahiert und daraus Entscheidungskriterien für z.B. die Anwesenheit, die Lokalisierung sowie die Klassifizierung von Objekten auf Bildern abgeleitet werden. Bei erfolgreicher Ableitung dieser Kriterien soll ein entstehendes Objekterkennungsmodell in der Lage sein, auch neue, unbekannte Bilder zu verarbeiten und die relevanten Objekte darauf zu lokalisieren und zu klassifizieren.

Im Rahmen eines sogenannten Trainingsprozesses kann die genannte Funktion approximiert werden. Das Ziel dabei besteht darin, dass die Funktion aus Inputs in Form von Bildern den richtigen Output in Form von Predictions generieren kann. Die Outputs enthalten ermittelte Bounding Boxes, Klassen und Confidences, die idealerweise mit den Ground-Truth-Labels übereinstimmen. Eine mögliche Umsetzung von Supervised Learning besteht in der Nutzung von neuronalen Netzen. Dabei enthält die zu nähernde Funktion eine Reihe von Parametern (im Folgenden Gewichte genannt), die in einem sogenannten Trainingsprozess schrittweise angepasst werden (vgl. S. 161ff. [12]). Das Ziel dieser Anpassung besteht darin, für jeden Input aus der Menge des Trainingsdatensatzes einen Output zu generieren, der möglichst nah an den Ground-Truth-Labels des Inputs liegt. Bei einer großen Menge an Trainingsdaten ist

das Modell durch die Optimierung der Gewichte in der Lage, die Informationen der Inputs zu abstrahieren und anhand dieser abstrakten Daten Entscheidungen über das Vorhandensein und die Art von Objekten zu treffen. Die Entscheidungsgrundlagen sind idealerweise so abstrakt, dass sie auch auf fremde Inputs angewendet werden können (Generalisierung).

Mithilfe eines zweiten sogenannten Evaluationsdatensatzes kann nach jedem Trainingsschritt der Erfolg des Trainingsprozesses überprüft werden. Ein Evaluationsdatensatz gleicht einem Trainingsdatensatz, jedoch ist die Datenmenge disjunkt. Er wird nur genutzt, um den Trainingsdatensatz zu evaluieren - dabei wird kein Trainingsschritt durchgeführt. Wenn der Datensatz an Trainingsbildern und der Aufbau des Modells richtig gewählt sind, ist das Modell im Anschluss an das Training im Optimalfall in der Lage, auch Bilder des Evaluationsdatensatzes, mit denen das Modell nicht trainiert wurde, richtig zu verarbeiten. Es entsteht ein Objekterkennungssystem, das in der Lage ist, Objekte auf unbekannten, neuen Daten zu erkennen. Abbildung 2 visualisiert den genannten Ablauf vereinfacht.

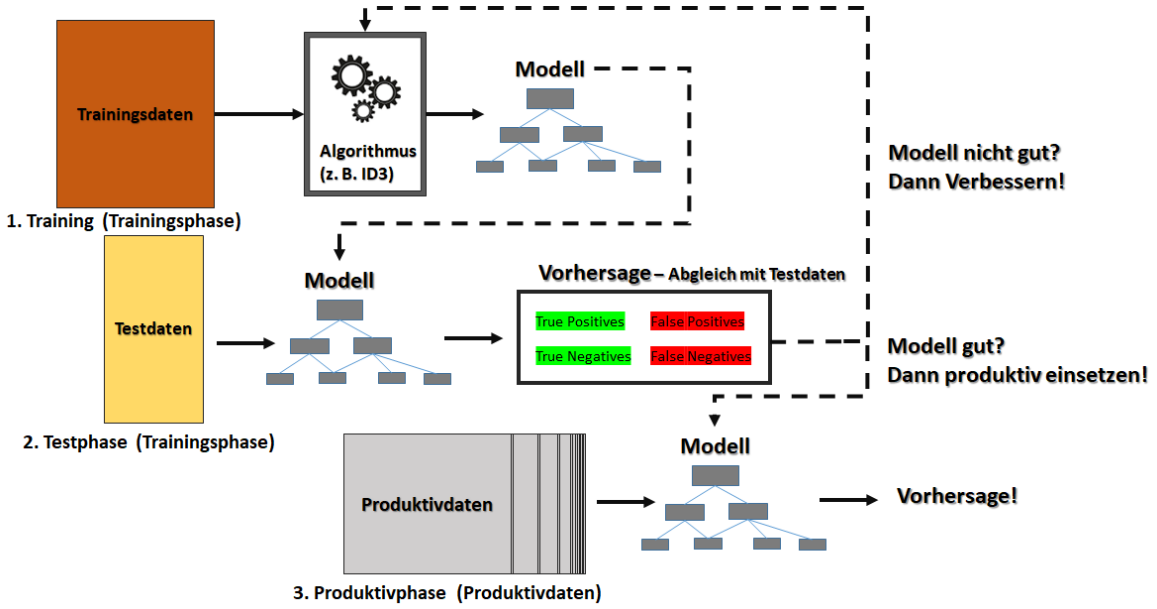


Abbildung 2: Prozess im Supervised-Learning aus [1]

1.2 Evaluation eines Objekterkennungssystems

Wie bereits beschrieben besteht eine Gruppe an Lösungen für Objekterkennungsprobleme aus Verfahren mit maschinellem Lernen. Jedoch existieren sowohl außerhalb als auch innerhalb dieser Gruppe viele verschiedene Ansätze für die es unterschiedlichste Ausprägungen von Parametern und genutzten Verfahren gibt. Die Vielzahl an Problemlösungen ist historisch bedingt: Das Themengebiet der Objekterkennung ist ein sehr aktuelles und stark diskutiertes Forschungsthema, in dem regelmäßig neue Forschungsergebnisse veröffentlicht werden. Um die Qualität eines Objekterkennungssystems zu messen und mit anderen Systemen zu vergleichen, existieren einige Metriken, die im folgenden Kapitel vorgestellt werden.

IoU

Everingham [8] beschreibt, dass die *IoU* zwischen einer vom Objekterkennungsmodell ermittelten Bounding Box und der Ground-Truth Bounding Box aus deren Überlappungsfläche geteilt durch die vereinigte Fläche errechnet werden kann (Abbildung 3).

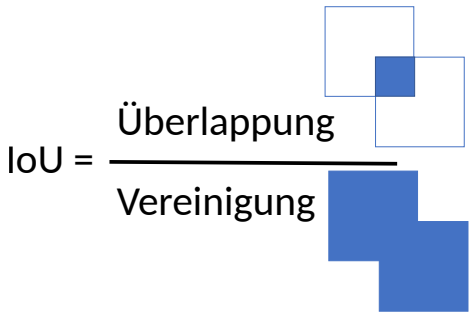


Abbildung 3: Berechnung der *IoU*

Mithilfe der *IoU* können laut Zeng [55] Metriken für die Messung der Qualität der Predictions des Objekterkennungssystems errechnet werden. Die bereits genannte Confidence spielt hierbei eine elementare Rolle. Oft werden viele Predictions erstellt, die Objekte mit einer sehr geringen Confidence erkennen. Um diese zu filtern, wird zunächst ein *Confidence Threshold* definiert. Wenn eine Prediction eine höhere Confidence aufweist, als der Wert des Thresholds, wird interpretiert, dass ein Objekt erkannt wurde. Andere Predictions werden nicht als erkannte Objekte betrachtet und ignoriert. Der *IoU Threshold* ist eine weitere Grenze, die beschreibt, ab welchem *IoU* Wert eine Prediction als „richtig“ in Bezug auf dessen Position und die Dimensionen der Bounding Box interpretiert wird. Die folgenden Metriken nutzen die genannten Thresholds.

Felder einer Konfusionsmatrix

In einer Prediction erkannte Objekte können auf dem Input-Bild entweder vorhanden oder nicht vorhanden sein. Ebenso können auf dem Input abgebildete Objekte entweder erkannt oder nicht erkannt worden sein. Die möglichen Fälle in Bezug auf die Objekterkennung beschreibt Zeng [55] folgendermaßen:

- True Positives (*TP*):
  Das System ermittelt eine Prediction, deren Klasse der Ground Truth entspricht. Damit ist die Confidence größer als der Threshold und es wird die richtige Klasse ermittelt. Ist die *IoU* zwischen Prediction Bounding Box und Ground Truth Bounding Box gleichzeitig größer als der *IoU Threshold*, beschreibt Zeng die Prediction als *True Positive*.

- False Positives (*FP*):
  Auch im Fall eines *False Positives* wird ein Objekt ermittelt, die Confidence ist größer als der *Confidence Threshold*. Wenn in der Ground Truth kein Objekt oder ein Objekt einer anderen Klasse beschrieben ist, beschreibt Zeng die Prediction als *False Positive*. Ein anderer Fall kann darin bestehen, dass der Objectness Score und die ermittelte Klasse zwar zur Ground Truth passen, jedoch die *IoU* kleiner als der definierte *IoU Threshold* ist, das Objekt also an der falschen Stelle ermittelt wurde.

- False Negatives (*FN*):
  Das System ermittelt kein Objekt, damit ist der Objectness Score kleiner als der Threshold. Jedoch beschreibt ein Ground Truth Label, dass ein Objekt vorhanden ist.

- True Negatives (*TN*):
  Das System ermittelt kein Objekt, damit ist der Objectness Score kleiner als der Threshold. Auch in den Ground Truth Labels wird kein Objekt beschrieben.

Die vier Möglichkeiten werden in folgender Tabelle vereinfacht zusammengefasst:

	Objekt vorhanden	kein Objekt vorhanden
Objekt erkannt	True Positive	False Positive
kein Objekt erkannt	False Negative	True Negative

Tabelle 1: Beispiel einer Konfusionsmatrix für die Objekterkennung

Precision und Recall

Anhand der Precision wird ermittelt, wie oft das Modell richtig liegt, wenn es ein Objekt erkennt [55]. Sie berechnet sich aus der Anzahl der True-Positives geteilt durch die Anzahl an True und False Positives.

$$precision = \frac{TP}{TP + FP} \tag{1}$$

Als Gegenstück dazu berechnet sich der Recall aus der Anzahl der True Positives durch die Summe aus True Positives und False Negatives, also der Anzahl an Ground Truth Labels insgesamt. Der Recall beschreibt damit, ob das Modell jedes Objekt erkannt hat, das es erkennen sollte [55].

$$recall = \frac{TP}{TP + FN} = \frac{TP}{n_{labels}} \tag{2}$$

Zur Beschreibung der Abhängigkeit von Precision und Recall bietet sich folgendes Beispiel aus der Rezepterkennung an: Abbildung 4 zeigt ein Kassenrezept mit 18 Ground Truth Objekten.

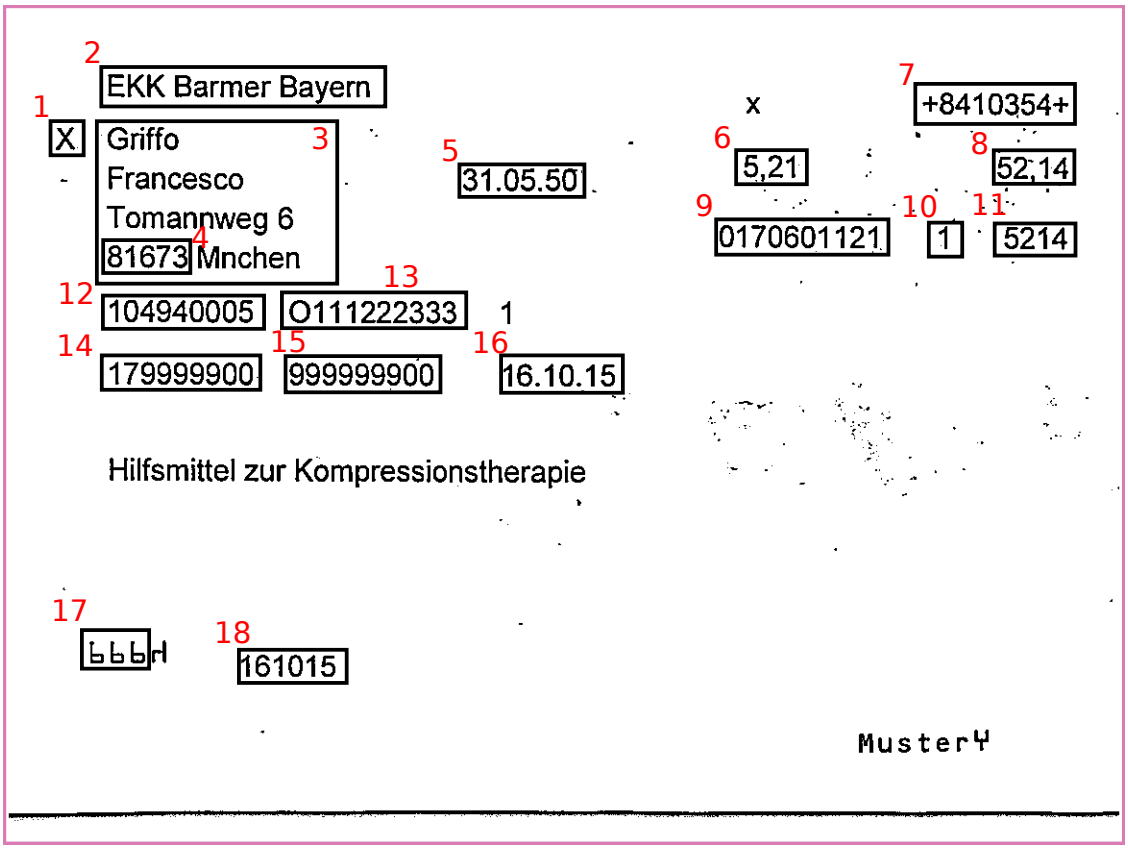


Abbildung 4: Mögliche Bounding Boxes auf einem Muster-Rezept

Erkennt ein Objekterkennungssystem nur das Feld 2 (Name der Krankenkasse), weist das System eine perfekte Precision mit dem Wert 1 auf: Jedes vom System erkannte Objekt ist auch richtig. Jedoch wird ein sehr niedriger Recall-Wert von $\frac{1}{18}$ bestimmt, da nur eines von 18 Objekten erkannt wurde. Gegenteiliges wird erreicht, wenn das System z.B. 50 Objekte auf dem Rezept erkennt, darunter alle Ground-Truth-Labels. Der Recall wäre zwar bei 1, jedoch würde eine Precision von $\frac{18}{50}$ gegen die Qualität des Systems sprechen. Zeng [55] beschreibt, dass mit einem fallenden Confidence Threshold der Recall monoton steigt und die Precision zwar schwankt, jedoch tendenziell sinkt. In Abbildung 5 ist ein Beispiel für eine derartige Precision-Recall-Kurve zu sehen.

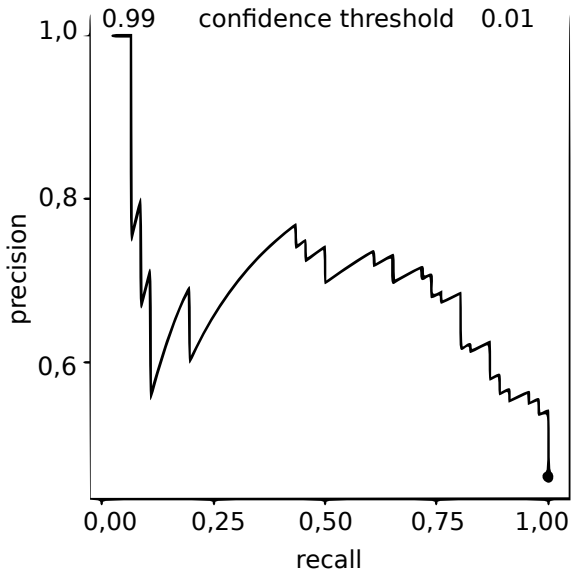


Abbildung 5: Beispiel für eine Precision-Recall-Kurve bei fallendem Confidence Threshold (abgewandelt aus [55])

Recall-*IoU*-Kurve

Neben dem Threshold für den Objectness Score beschreibt Zeng [55] auch den *IoU Threshold* als einen sehr wichtigen Wert, der die Qualitätsmessung beeinflusst. Je höher der *IoU Threshold* gewählt wird, desto niedriger wird der Recall. Wenn ein Modell auf Abbildung 4 alle 18 Objekte erkennt, aber die Bounding Boxes nicht optimal mit denen aus den Ground Truth Labels übereinstimmen, ergibt sich folgender Fall: Je höher der *IoU* Threshold gewählt wird, desto weniger dieser Predictions werden als True Positives gewertet. In Abbildung 6 ist ein Beispiel für eine derartige Kurve zu sehen.

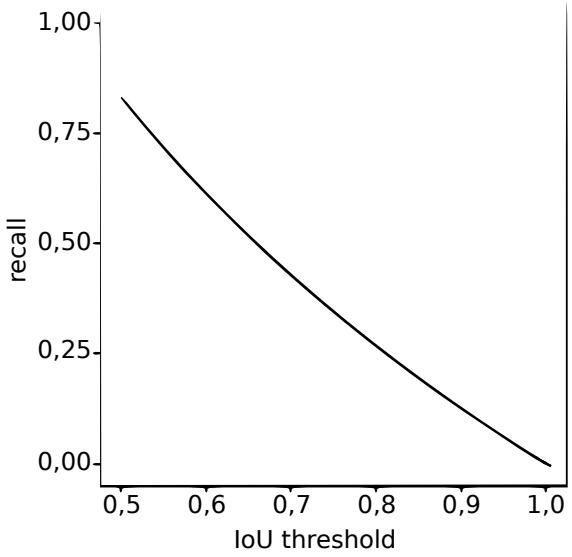


Abbildung 6: Beispiel für eine Recall-*IoU*-Kurve (abgewandelt aus [55])

Average Precision (*AP*) und Average Recall (*AR*)

Zwar kann die Precision-Recall-Kurve bei der Bewertung eines Objekterkennungssystems helfen, jedoch ist es mühsam, die Kurven verschiedener Objekterkennungssysteme zu vergleichen und deren Schnittpunkte zu analysieren. Die *AP* stellt eine Metrik dar, die auf der Precision-Recall-Kurve beruht und einen direkten Vergleich verschiedener Detektoren ermöglicht. Sie errechnet sich laut Zeng [55] prinzipiell aus dem Durchschnitt aller Precision Werte auf der Precision-Recall-Kurve. Dies kann erreicht werden, indem die Fläche unter der Precision-Recall-Kurve berechnet wird.

Um die Berechnung zu vereinfachen, kann vorher die Precision auf mehreren sog. Recall-Levels r_i interpoliert werden. Das bedeutet, dass die Funktionswerte der Precision-Recall Kurve vereinfacht mithilfe von bekannten benachbarten Werten angenähert werden. Damit wird die näherungsweise Berechnung der Fläche darunter ermöglicht. Diese Vorgehensweise wird ebenfalls von Everingham [8] vorgeschlagen. Die sogenannte Interpolated Average Precision wurde erstmals von Salton und McGill [43] eingeführt. Die Interpolation wird durchgeführt, indem die Kurve so angepasst wird, dass sie von rechts nach links stufenweise steigt, wenn ein Precision Wert erreicht wird, der größer ist, als alle bisher erreichten Precision Werte. In Abbildung 7 ist die Interpolation ersichtlich.

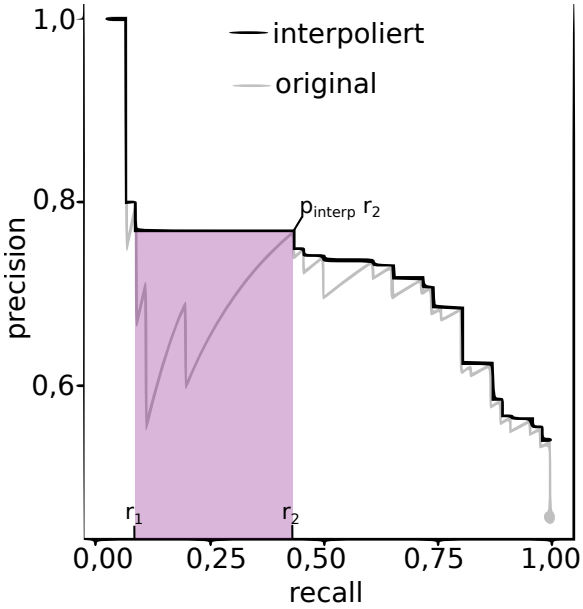


Abbildung 7: Interpolierte Precision-Recall-Kurve mit Berechnungsgrundlage (abgewandelt aus [55])

Im Anschluss kann dann der Durchschnitt der Precision Werte durch die Fläche unter der interpolierten Precision-Recall-Kurve folgendermaßen berechnet werden:

$$AP = \sum_{i=1}^{n-1} (r_{i+1} - r_i) p_{interp}(r_{i+1}) \tag{3}$$

$$p_{interp}(r) = \max_{r' \geq r} p(r') \quad (4) \tag{4}$$

Für jedes Plateau auf dem Graphen wird also die Fläche des Rechteckes darunter berechnet, indem der linke Recall Wert des Plateaus vom rechten Recall Wert abgezogen und das Ergebnis mit der Precision am Rechten Recall multipliziert wird. Verständlich wird die Berechnung bei Betrachtung von Abbildung 7, in der die violette Fläche als Teil der *AP* durch $(r_2 - r_1) \times p_{interp} r_2$ berechnet werden kann, dabei ist $p_{interp} r_2$ der Precision Wert von r_2.

Wie die *AP* ist auch der *AR* [55] eine Metrik, mit der Objekterkennungssysteme anhand eines numerischen Wertes vergleichbar werden. Der *AR* ist der Durchschnitt der Recalls für alle *IoU* Werte $IoU \in [0.5, 1.0]$, der durch die doppelte Fläche unter der Recall-*IoU* Kurve folgendermaßen berechnet werden kann:

$$AR = 2 \int_{0.5}^{1} recall(o)\, dx \tag{5}$$

mit $o = IoU$ und $recall(o) =$ zugehöriger Recall

Mean Average Precision (*mAP*) und Mean Average Recall (*mAR*)

Zeng [55] beschreibt, dass die Average Precision nur eine Klasse berücksichtigt, wobei in Objekterkennungsproblemen gewöhnlicherweise mehrere Klassen erkannt werden müssen. Die Mean Average Precision berechnet den Mittelwert aller AP's aller Klassen und ist damit eine zentrale Metrik, die beim Vergleichen von Objekterkennungssystemen Verwendung findet.

$$mAP = \frac{\sum_{i=1}^{K} AP_i}{K} \tag{6}$$

Ähnlich wie die Average Precision unterscheidet der Average Recall keine verschiedenen Klassen. Auch hier wird der Mittelwert aller AR's aller Klassen berechnet, sodass der mAR laut Zeng [55] als zweite unerlässliche Metrik für den Vergleich von Objekterkennungssystemen gilt.

$$mAR = \frac{\sum_{i=1}^{K} AR_i}{K} \tag{7}$$

Weitere Metriken

Um verschiedene Objekterkennungsmodelle zu vergleichen sowie neue Techniken und Ansätze in der Praxis zu evaluieren, werden oft mehrere Metriken mit verschiedenen Thresholds gesammelt betrachtet. Ein Beispiel dafür ist die $AP^{IoU=.50}$ Metrik, bei der die *mAP* mit dem *IoU*-Threshold 0.5 berechnet wird. In dieser Arbeit werden vereinfacht die Abkürzungen *AP* und *AR* für *mAP* und *mAR* verwendet. Außerdem ist es auch möglich, nur Objekte in einem bestimmten Größenbereich in die Berechnung einzubeziehen, um zu analysieren, ob das Objekterkennungssystem in verschiedenen Größenbereichen bessere Ergebnisse liefert, als in anderen.

Als Beispiel für eine Sammlung von unterschiedlichen Metriken kann die Common Objects in Context (*COCO*) Challenge [6] genannt werden. In dieser Arbeit werden alle Objekterkennungssysteme anhand dieser Metriken verglichen. Die Berechnung aller verschiedenen Ausprägungen der AP und AR Metriken können in [6] und der dieser Arbeit zugehörigen Praxisarbeit nachvollzogen werden. Die *COCO* Metriken kommen in der Evaluation der Verfahren aus *YOLO*v4 in **??** zum Einsatz.

2 *YOLO*

YOLO ist ein weit verbreitetes Object-Detection Verfahren, das im Jahr 2015 von Redmon et al. [37] veröffentlicht und seitdem in mehreren Schritten weiterentwickelt wurde. Sie betrachten die Objekterkennung als ein Regressionsproblem, bei dem mithilfe eines neuronalen Netzes aus den Pixeln des Eingangsbildes Objekte ermittelt werden. Hierbei wird ein erkanntes Objekt durch eine sogenannte Bounding Box beschrieben, die Position und Größe eindeutig bestimmt. Des Weiteren werden für jedes erkannte Objekt so viele Klassenwahrscheinlichkeiten ermittelt, wie es mögliche Klassen im jeweiligen Anwendungsfall gibt. Für jede Klasse wird ein Klassenwahrscheinlichkeitswert bestimmt - die Summe dieser Werte ergibt eins. Ein erkanntes Objekt wird der Klasse mit der höchsten Klassenwahrscheinlichkeit zugeordnet.

Ein zentrales Ziel von *YOLO* ist die Echtzeit-Objekterkennung. Aufgrund seiner Eigenschaften als One-Stage Object Detector wird für den gesamten Prozess der Objekterkennung lediglich ein neuronales Netz verwendet, das pro Bild durchlaufen wird - das Bild wird dabei nur einmal betrachtet (You Only Look Once). Das bewirkt, dass *YOLO* end-to-end nach seiner Leistung in der Objekterkennung optimiert werden kann.

Im Gegensatz dazu kann die Gruppe der Two-Stage Object Detectors genannt werden, zu denen unter anderem die Methoden Region Based Convolutional Neural Network (*RCNN*) [16], Fast *RCNN* [15] sowie Faster *RCNN* [38] gehören. Vereinfacht erstellen Two-Stage Methoden in einem ersten Schritt sogenannte Region-Proposals, also Vorschläge für Bereiche, in denen sich Objekte befinden könnten. In einem zweiten Schritt werden diese Regionen verwendet und der Inhalt als Klassifikationsproblem betrachtet, d.h. die Klasse des abgebildeten Objekts ermittelt [16]. Redmon et al. beschreiben, dass diese Objekterkennungssysteme keine Echtzeit-Performance liefern können, während *YOLO* aufgrund seines Designs sehr performant ist. Die folgenden Erläuterungen zu *YOLO* basieren auf den Erkenntnissen von Redmon et al.[37].

2.1 Objekterkennung mit *YOLO*

In *YOLO* wird ein neuronales Netz zum Generieren von Vorhersagen (Predictions) über die Objekte auf einem Input-Bild verwendet. Um die Vorgänge in *YOLO* zu verstehen, soll zunächst der Aufbau der Ground-Truth-Labels und der Predictions, also des Outputs des verwendeten neuronalen Netzes erläutert werden: Ein zu verarbeitendes Bild wird in ein Raster aus $S \times S$ Zellen zerteilt. Diejenige Zelle, die den Mittelpunkt eines Objektes auf dem

Bild beinhaltet, ist für die Erkennung des Objektes „zuständig“ - somit wird idealerweise jedes Objekt von genau einer Zelle bestimmt. Zu jeder dieser genannten Zellen gehören B Bounding Boxes. Die Anzahl der Zellen S und der Bounding Boxes B wird im Design des *YOLO*-Objekterkennungsmodells beliebig ausgewählt. Redmon et al. verwenden in ihrem Modell ein Raster aus 7×7 Zellen mit je zwei Boxen.

Eine Bounding Box wird durch Koordinaten (x, y, w, h) und einem Confidence-Score beschrieben. Die (x, y) Koordinaten beschreiben das Zentrum der Bounding Box als relativen Offset zu der Zelle, zu der sie gehören. Die w, h Koordinaten beschreiben die Höhe und Breite der Box relativ zum gesamten Bild. Mit den Bounding Boxes können somit Position und Größe von Objekten auf Bildern definiert werden. Der letzte Wert, durch den eine Bounding Box vollständig beschrieben ist, ist der Confidence Score. Dieser beschreibt, ob die jeweilige Bounding Box ein Objekt beinhaltet und wie gut sie sich mit diesem Objekt deckt. Formal wird der Confidence Score einer Bounding Box damit folgendermaßen definiert:

$$Confidence = Pr(Object) \times IoU_{pred}^{truth} \tag{8}$$

$Pr(Object)$ ist der Objectness-Score, der bestimmt, wie wahrscheinlich es ist, dass die Box ein Objekt beinhaltet. IoU_{pred}^{truth} bestimmt, wie genau die Größendimensionen und Position der Box verglichen zu den Dimensionen des beinhalteten Objektes sind.

Um ein Objekterkennungsmodell mit *YOLO* zu trainieren, muss für jedes Bild des Trainings- und Evaluations-Datensatzes eine Ground Truth erstellt werden. Diese beinhaltet für jede zuständige Rasterzelle die jeweilige Definition der Bounding Boxes passend zu den Objekten. Die Ground-Truth wird somit durch die gekennzeichneten Koordinaten der Objekte und folgende optimale Confidence-Scores der Boxen definiert:

- Kein Objekt in der Box: Confidence Score ist 0, da $Pr(Object) = 0$
- Objekt vorhanden: $Pr(Object) = 1 \Rightarrow$ Confidence Score = $1 \times IoU_{pred}^{truth} = IoU_{pred}^{truth}$
- Objekt genau in Koordinaten x, y, w, h der Box vorhanden:
  $Pr(Object) = 1; IoU_{pred}^{truth} = 1 \Rightarrow$ Confidence Score = $1 \times 1 = 1$

Abbildung 8 soll den Aufbau einer Ground-Truth Bounding Box verständlich visualisieren. Man nehme an, die blaue Bounding Box umrahmt ein Objekt auf einem Bild. Das Zentrum dieses Objekts fällt auf die Rasterzelle mit den Koordinaten $(x = 3, y = 7)$. Die (x, y) Koordinaten des Objekts selbst werden, wie in der Abbildung dargestellt, durch den relativen Offset berechnet. Die (w, h) Koordinaten werden relativ zum gesamten Bild modelliert.

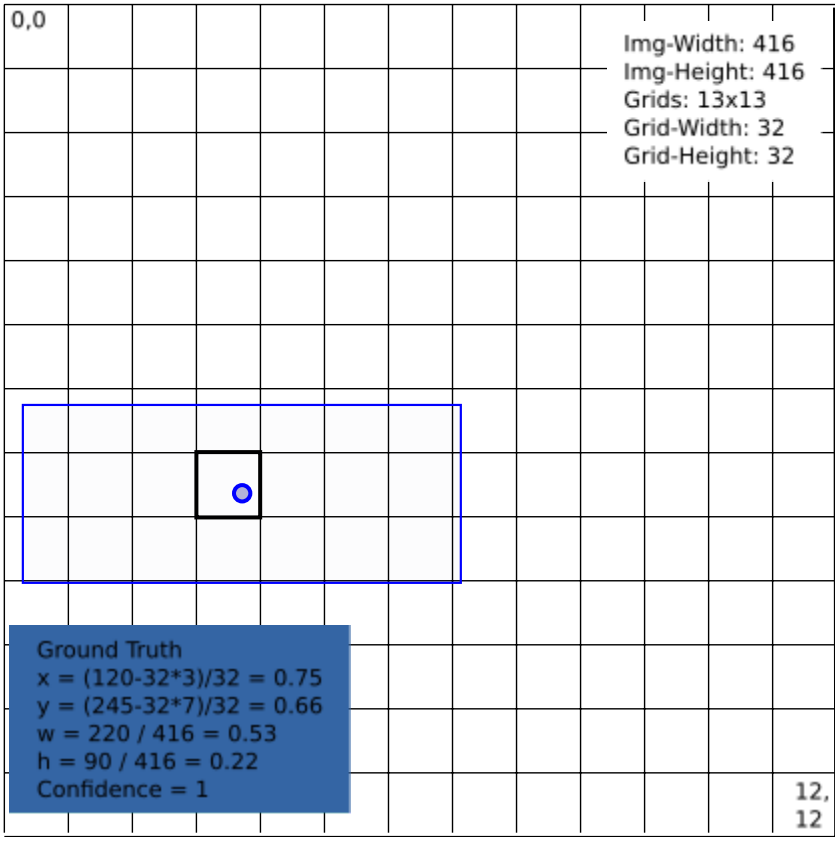


Abbildung 8: Visualisierung der Modellierung eines Ground-Truth Labels für eine Bounding-Box; Die ursprünglichen Koordinaten auf dem Input Image sind $(x = 120, y = 245, w = 220, h = 90)$

Ein weiterer Bestandteil der Ground-Truth und damit auch der Predictions, sind Klassenwahrscheinlichkeiten. Sie beschreiben, zu welcher Klasse ein erkanntes Objekt gehört und werden im Gegensatz zu den fünf bereits genannten Werten nicht pro Bounding Box, sondern pro Zelle im Raster bestimmt. In *YOLO* werden pro Rasterzelle C (= Anzahl der möglichen Klassen) Klassenwahrscheinlichkeiten definiert. Sie bestimmen, ob die Zelle ein Objekt der jeweiligen Klasse beinhaltet. Für die Ground-Truth ergeben sich diese Werte aus $Pr(Class_i|Object)$. Der Grund für die Ermittlung pro Rasterzelle und nicht pro Bounding Box ist, dass jede Zelle für die Bestimmung eines Objektes verantwortlich ist - die Bounding Boxes der Zelle sollen demnach alle dasselbe Objekt erkennen. Folgendes Beispiel soll die Klassenwahrscheinlichkeit in der Ground-Truth verdeutlichen:

- Man nehme an, es gäbe vier mögliche Klassen
- Das Objekt, das von einer Rasterzelle erkannt wird, gehört zur Klasse mit dem Index 0
- Das Label der Klassenwahrscheinlichkeiten ist somit als (1,0,0,0) definiert

  $\rightarrow Pr(Class_0|Object) = 1$

  $\rightarrow Pr(Class_{i\in 1,2,3}|Object) = 0$

Aus der Klassenwahrscheinlichkeit einer Zelle und dem Confidence Score einer Bounding Box kann ein klassenspezifischer Confidence Score für die Box errechnet werden. Dieser wird für die Berechnung des Loss Wertes (Kapitel 2.3), nach dem das Modell optimiert wird, benötigt. Der klassenspezifische Confidence Score ist folgendermaßen definiert:

$$Class\,Cond.\,Prob. = Pr(Class_i|Object) \times Pr(Object) \times IoU^{truth}_{pred} = Pr(Class_i) \times IoU^{truth}_{pred} \tag{9}$$

Mit dem beschriebenen Aufbau besteht die Ground Truth eines *YOLO*-Modells aus $S \times S \times (B \times 5 + C)$ Werten, die in Form eines Tensors (= Ground Truth Tensor) modelliert werden.

Für die Erstellung eines Objekterkennungssystems modellieren Redmon et al. ein Convolutional Neural Network (*CNN*), das in der Lage ist, einen Tensor in der beschriebenen Form als Prediction zu erstellen (*CNN*: siehe S.79ff. [30]). Durch den Trainingsprozess, in dem die Gewichte des Modells schrittweise angepasst werden, wird das Modell optimiert. Ziel dieses Prozesses ist es, dass die Prediction für jedes Input-Bild möglichst nah an dessen Ground Truth Label ist. Bei der Optimierung des Modells mithilfe von vielen Trainingsbildern „lernt“ es, Informationen zu abstrahieren und daraus Entscheidungskriterien abzuleiten, die idealerweise auch auf andere, trainingsfremde Bilder angewendet werden können. Das *CNN* reduziert die Auflösung des Eingangsbildes schrittweise auf die Größe $S \times S$, während die Channels von zum Beispiel drei Farbkanälen auf $(B \times 5 + C)$ erhöht werden. Damit erzeugt das Netz einen Output Tensor der Form $S \times S \times (B \times 5 + C)$.

2.2 Das neuronale Netz

Um das erste *YOLO* Modell zu erhalten, trainieren die Autoren erst ein *CNN* zur Bildklassifizierung (Klassifizierungsproblem: siehe S.21f. [12]) mithilfe des ImageNet 1000-class competition Datensatzes [42]. Anschließend wird dieses *CNN* zu einem Objekterkennungsmodell umfunktioniert (Transfer Learning). Der Grund hierfür wird von Redmon et al. [35] beschrieben: Datensätze, mit denen Klassifizierungsnetze trainiert werden können, beinhalten mehrere Millionen Bilder in tausenden Kategorien, während Object-Detection-Datensätze

zum Zeitpunkt der Veröffentlichung meist nur aus einigen hunderttausend Bildern in hunderten Kategorien bestehen. Somit wird in einem ersten Schritt ein Bildklassifikationssystem erstellt, das bereits in der Lage ist, Informationen zu abstrahieren und damit z.B. Kanten und Formen zu erkennen und für die Klassifikation zu deuten. Das Basismodell besteht aus 20 Convolutional Layers. Die Auflösung der Input-Bilder wird durch Max-Pooling Layer reduziert (Convolution und Pooling: siehe S.79ff. [30]).

Im Anschluss an das Training mit dem Klassifikationsdatensatz wird das Modell in ein Detection-Model konvertiert, indem der Fully Connected Layer (*FC Layer*) durch vier Convolutional Layer und zwei *FC Layer* mit zufällig initialisierten Gewichten ersetzt wird (*FC Layer*: siehe S.174 [12]). Außerdem erhöhen die Autoren die Input-Auflösung des Netzes von 224×224 auf 448×448 Pixel. Im Anschluss wird mithilfe der Datensätze PASCAL VOC 2007 und 2012 [8] ein Objekterkennungsmodell trainiert. Die Autoren verwenden die Leaky Rectified Linear Unit (*Leaky ReLU*) Aktivierungsfunktion nach jedem Convolutional Layer und im ersten *FC Layer* (*Leaky ReLU*: siehe [29]). Lediglich im letzten *FC Layer* wird eine lineare Funktion als Aktivierungsfunktion verwendet. Damit besteht die erste beschriebene Netzarchitektur aus 24 Convolutional Layers gefolgt von zwei *FC Layers*, sie ist in Abbildung 9 ersichtlich. Die von den Autoren empfohlenen und getesteten Klassifizierer-*CNNs* werden in dieser Arbeit im Folgenden Backbones und die zusätzlichen Schichten für die Objekterkennung Object-Detection-Head genannt. Die Autoren verändern und ersetzen unter anderem ihre Backbones in deren späteren Veröffentlichungen ([35], [36]) mehrfach. In ihren Experimenten definieren sie die Dimensionen des Output-Tensors als $(S,S,(B \times 5 + C)) = (7,7,(2 \times 5 + 20)) = (7,7,30)$.

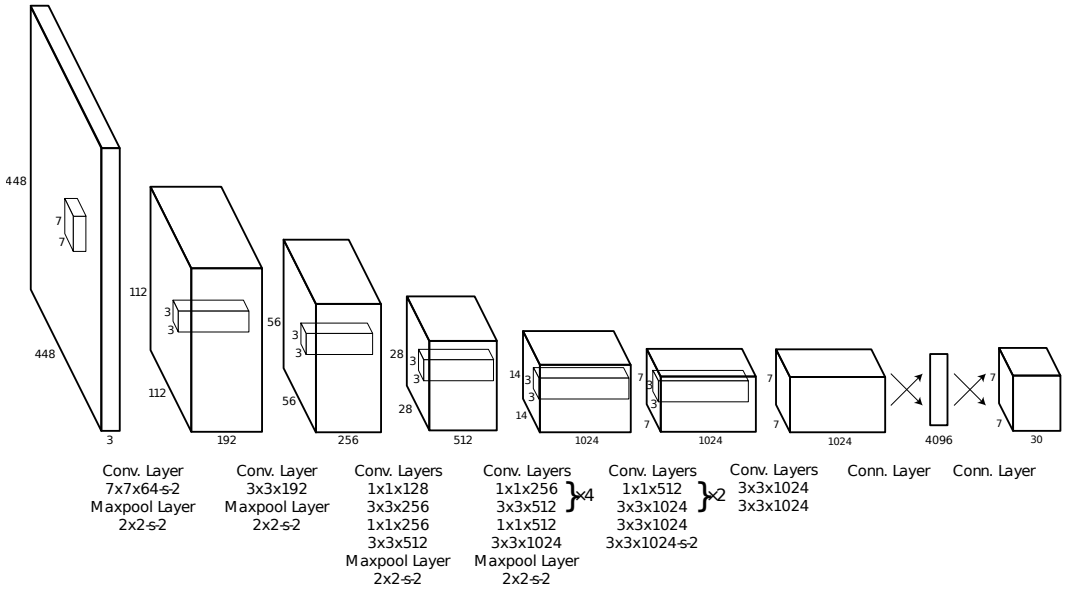


Abbildung 9: Netzwerkarchitektur aus [37] (angepasst)

2.3 Loss-Funktion

Als Output des letzten Layers entsteht ein Tensor in der Form $(S, S, (B \times 5 + C))$, dessen Inhalt die Prediction des Netzes für ein Eingangsbild darstellt. Im Training werden die Objekte der Bilder des Trainingsdatensatzes erkannt und die Predictions mit der Ground-Truth verglichen. Aus den Unterschieden zwischen Prediction und Ground-Truth wird ein Loss berechnet, der im Trainingsprozess minimiert werden soll. Damit werden die Gewichte des Modells mithilfe dieses Loss schrittweise optimiert. Der Loss kann damit als Bestrafungsterm betrachtet werden, dessen Wert im Laufe des Trainings zu minimieren ist. In *YOLO* wird der Loss aus der Residuenquadratsumme, auch Residual sum of squares (*RSS*) genannt, errechnet. Sie ist durch die Summe der Quadrate von Abweichungen ($RSS = \sum(y - \hat{y})^2$) des *YOLO*-Outputs zu den Ground-Truth Labels definiert. Die verwendete Loss-Funktion besteht aus den folgenden drei Teilen:

- Classification Loss
- Localization Loss
- Confidence Loss

Localization Loss

Dieser Teil der Loss-Funktion beschreibt die Abweichungen zwischen den Dimensionen der vom Modell bestimmten Bounding Boxes und der Ground Truth Boxes. Hierfür werden die Residuenquadratsummen für die (x, y) und (w, h) Koordinaten aller Bounding Boxes aufsummiert. Um die absoluten Fehler kleiner und großer Boxen nicht gleich zu gewichten, werden bei der Berechnung die Quadratwurzeln von w und h verwendet. Diese Modifikation bewirkt, dass z.B. ein Fehler von fünf Pixeln in einer großen Box weniger stark bestraft wird, als bei einer kleinen Box. Die Formel zur Berechnung des Localization Loss ist in Gleichung 10 beschrieben.

$$\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
&+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]
\end{aligned} \tag{10}$$

mit:

$$\mathbb{1}_{ij}^{obj} = \begin{cases} 1, & \text{wenn Bounding Box j in Zelle i für die Erkennung des Objektes zuständig ist} \\ 0, & \text{sonst} \end{cases}$$

$\lambda_{coord} = 5$

Bei der Berechnung des Localization Loss werden nur Boxen berücksichtigt, die für die Erkennung eines Objektes verantwortlich sind. Die Zuständigkeit einer Zelle ist bereits definiert (Kapitel 2.1). Jedoch ist bei der Berechnung des Loss nur die Box der Zelle, deren IoU mit der Ground-Truth am höchsten ist, für die Erkennung eines Objektes zuständig. Diese Designentscheidung führt dazu, dass in der Optimierung verschiedene Bounding Boxes in unterschiedlichen Größen generiert werden. Die verschiedenen Bounding Boxes können somit Objekte bestimmter Größen, Seitenverhältnisse oder Klassen besser vorhersagen können - sie werden spezialisert.

Eine weitere Besonderheit stellt das λ_{coord} dar. Da die meisten Boxen keine Objekte beinhalten und somit nicht für die Erkennung eines Objektes zuständig sind, entsteht eine sog. Class-Imbalance. So wird der Localization Loss nur für Boxen berechnet, die ein Objekt beinhalten. Um die Bedeutung dieser Fehler in der Lokalisierung zu erhöhen, wird der Localization-Loss mit λ_{coord} höher gewichtet. Im Gegenzug dazu wird im Confidence Loss ein λ_{noobj} eingeführt, das die Bedeutung des Losses vermindert, wenn kein Objekt erkannt wird. Abbildung 10 visualisiert eine zuständige Bounding Box (Prediction), die Ground-Truth Box sowie die Berechnung des Localization-Loss anhand eines Beispiels.

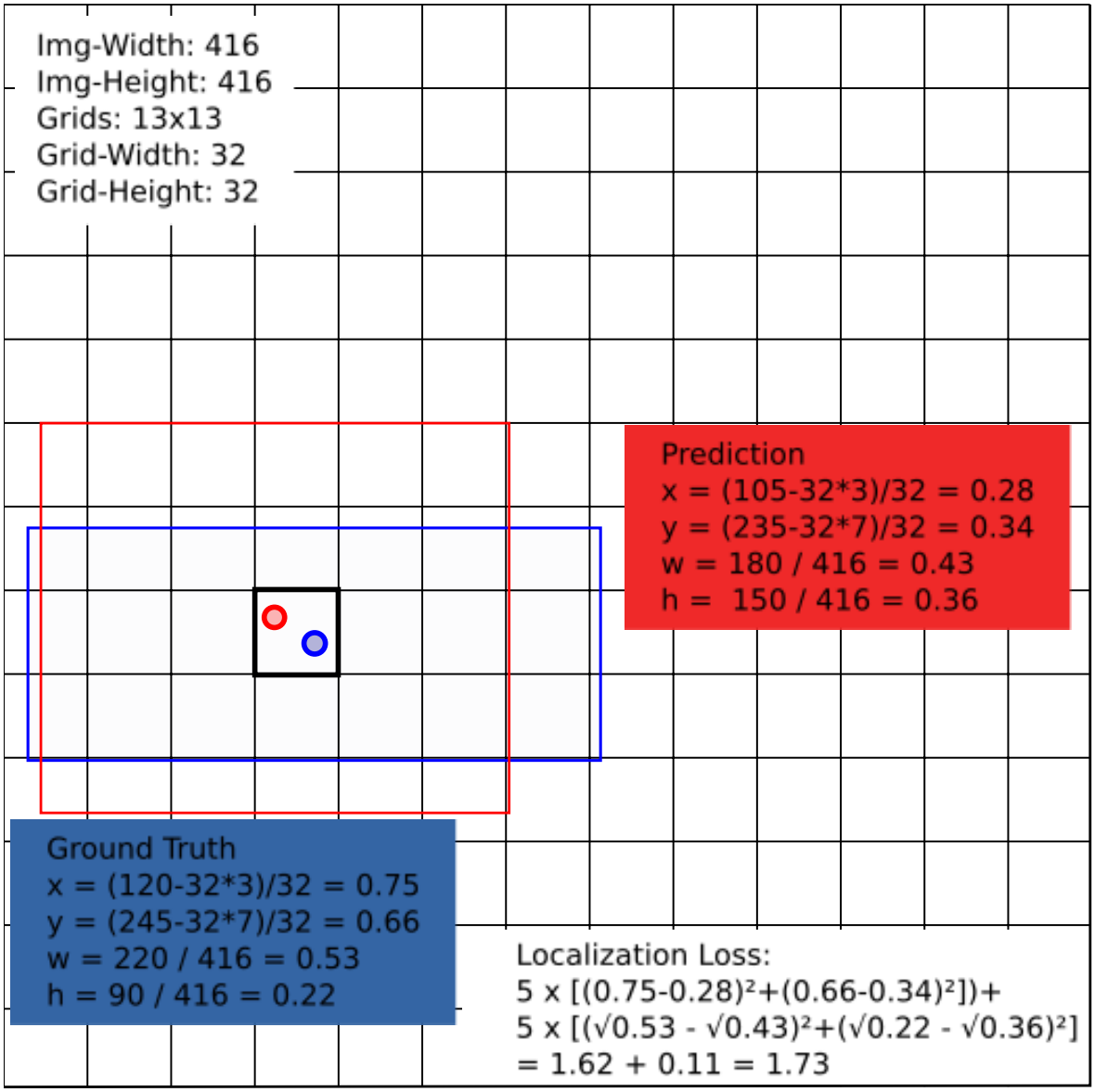


Abbildung 10: Visualisierung der Berechnung des Localization Loss

Classification Loss

Für jede Zelle, die ein Objekt beinhaltet, wird der Classification Loss berechnet. Er berücksichtigt den Fehler in der Klassifikation eines erkannten Objektes - aus diesem Grund wird hier die in Gleichung 9 beschriebene bedingte Klassenwahrscheinlichkeit (= Class-Conditional-Probability) benötigt. Wenn in einer Zelle ein Objekt vorhanden ist, berechnet sich der Classification Loss der Zelle aus der *RSS* der Class-Conditional-Probabilities aller Klassen.

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{11}$$

mit

$$\mathbb{1}_i^{obj} = \begin{cases} 1, & \text{wenn ein Objekt in Zelle i ist} \\ 0, & \text{sonst} \end{cases}$$

$\hat{p}_i(c) =$ Class-Conditional-Probability der Klasse c in Zelle i

Confidence Loss

Mit dem Confidence Loss wird der Fehler im Objectness Score einer Box berechnet. Das ist der Fehler in der Einschätzung, ob ein Objekt in der Box vorhanden ist und ob die Box für dessen Erkennung zuständig ist. In diesem Fall werden jedoch nicht ohne Weiteres alle Residuenquadratsummen berechnet, sondern in zwei Fällen unterschieden. Wie bereits im Localization Loss (Kapitel 2.3) beschrieben, wird mithilfe von λ_{noobj} Conficence Loss einer Bounding Box weniger stark gewichtet, wenn diese kein Objekt erkennen soll.

$$\begin{aligned} &\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 \\ &+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \end{aligned} \tag{12}$$

mit

$$\mathbb{1}_{ij}^{obj} = \begin{cases} 1, & \text{wenn Bounding Box j in Zelle i für die Erkennung des Objektes zuständig ist} \\ 0, & \text{sonst} \end{cases}$$

$$\mathbb{1}_{ij}^{noobj} = \begin{cases} 1, & \text{wenn Bounding Box j in Zelle i nicht für die Erkennung des Objekts zuständig ist} \\ 0, & \text{sonst} \end{cases}$$

$\lambda_{noobj} = 0.5$

$\hat{C}_i =$ Confidence Score der Box j in Zelle i

***YOLO* Loss**

Die Summe der drei beschriebenen Teile definiert den Loss, der für *YOLO* [37] festgelegt wird. Er ergibt sich aus folgender Berechnung:

$$
\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
&+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] && \text{// Localization Loss} \\
&\quad + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 \\
&\quad + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 && \text{// Confidence Loss} \\
&\quad\quad + \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 && \text{// Classification Loss}
\end{aligned}
\tag{13}
$$

2.4 Inferenz

In diesem Zusammenhang beschreibt der Begriff Inferenz die Objekterkennung auf einem Bild mithilfe eines bereits trainierten *YOLO*-Modells. Hierzu dient dieses als Input in das trainierte *YOLO* Modell. Wie bereits beschrieben, wird mit diesem Modell der gesamte Prozess der Objekterkennung in einem Durchlauf behandelt. Damit ist *YOLO* sowohl im Training, als auch in der Inferenz ein sehr performanter Ansatz. Durch den beschriebenen Aufbau wird für jede Bounding Box in jeder Rasterzelle eine Prediction erstellt. Damit generiert *YOLO* immer $(S, S, (B \times 5 + C))$ Predictions, auch wenn das Bild weniger Objekte beinhaltet. Sie können aber im Nachgang sinnvoll nach ihren Probabilities gefiltert werden, um die Anzahl zu verringern. Aufgrund der Tatsache, dass jede Zelle nur das Objekt erkennt, dessen Zentrum in der Zelle liegt, wird in den meisten Fällen nur eine Zelle pro Objekt auf dem Bild bestimmt. Die Autoren beschreiben aber, dass Objekte, deren Zentrum sich am Rand oder der Ecke einer Zelle befinden (also auch sehr nah an anderen Zellen), auch von mehreren Zellen erkannt werden können. Sie empfehlen die Nutzung von *NMS* (*NMS*: siehe [40] und Kapitel 5.2.7), um diesen Fehler zu beheben und erreichen in ihrem Anwendungsfall damit eine 2-3% bessere *mAP*.

2.5 Probleme bei der Nutzung von *YOLO*

YOLO ist zwar ein sehr schneller und weit verbreiteter Ansatz in der Objekterkennung, jedoch beschreiben die Autoren in ihrer Arbeit auch dessen Nachteile:
Durch die Definition, dass pro Zelle nur eine festgelegte Anzahl an Boxen existieren und all diese Boxen dieselbe Klasse erkennen müssen, entstehen Probleme bei der Erkennung kleiner Objekte, die sehr nah aneinander abgebildet sind. Des Weiteren ergeben sich Probleme aus dem Training des Modells mithilfe von Trainingsdaten. Dadurch kann es unter Umständen nicht in der Lage sein, Objekte auf Bildern zu erkennen, die sehr ungewöhnlich sind und sich damit zu stark von den Trainingsdaten unterscheiden. Das starke Downsampling des Originalbildes auf die Zielauflösung bringt außerdem relativ grobe Boundings Boxes mit sich. Die größte Fehlerquelle wird aber in der Lokalisierung der Boxen beschrieben. Während ein kleiner Fehler in einer großen Bounding Box kein Problem darstellen sollte, wirkt sich ein Fehler in einer kleinen Bounding Box stark auf die IoU aus. Die Verwendung der Quadratwurzeln von w und h im Localization Loss behandelt dieses Problem nicht ausreichend.

2.6 Fazit zu *YOLO*

Mit ihrer ersten Version von *YOLO* stellen die Autoren einen One-Stage Object Detector vor, den sie sowohl im Training als auch in der Inferenz als sehr performant bewerben. Es wird nur ein neuronales Netz verwendet, welches end-to-end trainiert werden kann. Damit ist zu erwarten, dass die Qualität in der Erkennung von Objekten hoch ist. Sie beschreiben zudem auch die Probleme, die sie mit ihrem Verfahren entdeckt haben und schaffen damit ein Bewusstsein für Verbesserungspotenzial, auf das sich in zukünftigen Verbesserungen von *YOLO* konzentriert werden kann. Sie veröffentlichen neben der Spezifikation in Form ihres Papers auch die Implementierung von *YOLO*, die in Darknet[1], einem Open-Source Framework für neuronale Netze, erstellt wurde. Das Modell wird in weiteren Versionen schrittweise verbessert, wobei das Grundkonzept, welches in diesem Kapitel erläutert wurde, gleichbleibend ist.

[1] https://github.com/pjreddie/darknet

3 *YOLO*v2

Nach der Veröffentlichung von *YOLO* arbeiteten die Autoren weiter an der Verbesserung ihres Object-Detectors. Sie stellten fest, dass bei *YOLO*, verglichen mit Two-Stage Ansätzen wie dem Fast-*RCNN* [15] oder Faster-*RCNN* [38], viele Fehler in der Lokalisierung (Localization Errors) stattfinden [35]. Außerdem soll der Recall im Vergleich relativ gering sein [35]. Aus diesem Grund veröffentlichen Redmon et al. mit *YOLO*v2 [35] eine verbesserte Version von *YOLO*, deren Fokus auf der Behebung der oben genannten Probleme liegt. Dabei sollen die hohe Geschwindigkeit und Qualität der Erkennung von *YOLO* beibehalten werden. Um Verbesserungen zu erreichen, experimentieren die Autoren in den folgenden Versionen mit verschiedenen Techniken, die sich in Bezug auf neuronale Netze insbesondere in der Objekterkennung als sinnvoll erwiesen haben. Das folgende Kapitel beschäftigt sich mit den Veränderungen, die sie in *YOLO*v2 [35] vornehmen.

3.1 Batch Normalization

Batch Normalization ist ein in *YOLO*v2 eingeführtes Verfahren, das auf der Arbeit von Ioffe et al. [22] basiert. Sie behandeln darin das Problem, dass die Inputs der Schichten eines neuronalen Netzes nicht normalisiert sind. Sie definieren den Begriff „Internal Covariante Shift“ als „die Änderung in der Verteilung der Netzwerkaktivierungen aufgrund der Änderung der Netzwerkparameter während des Trainings“ [22]. Demnach werden durch die Veränderung der Gewichte eines Layers während des Trainings auch die Outputs dieses Layers verändert. Folglich verändert sich die Verteilung der Eingangswerte des darauffolgenden Layers. Der folgende Layer muss seine Gewichte anschließend an die neue Verteilung anpassen, was zu einem zusätzlichen Aufwand im Training führt. Dies stellt insbesondere bei tiefen neuronalen Netzen ein Problem dar, wenn die Änderung in den Gewichten eines Layers weitreichende Folgen für viele weitere Layer hat.

Laut den Autoren ist es schon länger bekannt, dass ein Netzwerk schneller trainiert werden kann, wenn die Inputs des Netzes normalisiert werden und somit einen Mittelwert und eine Standardabweichung von 0 haben sowie dekorreliert sind. Dieser Vorgang soll nicht nur im Input des Modells, sondern auch für den Input jedes Layers sinnvoll sein. Damit kann der zusätzliche Aufwand der Anpassung an sich verändernde Verteilungen, also der Covariante Shift, vermieden werden. Die Anpassung der Gewichte eines Layers wird unabhängiger von vorhergehenden Layers. In Abbildung 11 wird der Algorithmus für die Batch Normalization beschrieben, die Herleitung ist in [22] nachzuvollziehen.

Input :Values of x over a mini-batch:$\mathscr{B} = \{x_1...m\}$;

Parameters to be learned:γ, β

Output :$\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathscr{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathscr{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathscr{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma^2_{\mathscr{B}} + \varepsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Abbildung 11: “Batch Normalizing Transform, applied to activation x over a mini-batch.“ [22]

In der Batch-Normalization wird in sogenannten Mini-Batches gearbeitet. Dabei werden Teilmengen des Trainingsdatensatzes einer festgelegten Größe (Mini-Batch-Size) gleichzeitig im Netzwerk verarbeitet. Die Nutzung von Mini-Batches wird damit begründet, dass es sehr aufwendig wäre, den gesamten Input des Modells und aller Layer auf einmal zu normalisieren. Die Funktion dazu wäre nicht an jeder Stelle differenzierbar und dieser Umstand würde ein Training, das auf Gradienten basiert, sehr aufwendig gestalten.

Indem der Mittelwert aus einer Mini-Batch berechnet, von jedem Element der Mini-Batch abgezogen und das Ergebnis durch die Standardabweichung geteilt wird, kann der Input eines Layers normalisiert werden. Bei der Berechnung der Standardabweichung wird ein kleiner Wert ε als Konstante auf die Varianz der Mini-Batch addiert, was die Autoren mit numerischer Stabilität begründen (mögliche Division durch 0 wird verhindert). Außerdem werden zwei Parameter γ und β eingeführt. Diese Werte skalieren und shiften den normalisierten Output eines Layers, sodass dieser zu der vom nachfolgenden Layer erwarteten Verteilung (Mittelwert und Varianz) passt. Die Werte γ und β sind Gewichte, die im Training gelernt und damit an die folgende Schicht angepasst werden.

Neben dem bereits genannten Vorteil führt Batch-Normalisierung auch zu einer Verbesserung der Generalisierung, also das Finden von allgemeinen Regeln aus Beispielen: Durch den Einbezug von anderen Trainingsbildern der Mini-Batch in die Berechnung wird auf jedem Trainingsbeispiel ein Rauschen erzeugt. In ihren Experimenten stellten die Autoren fest, dass dieses Rauschen zu einer Regularisierung führt (Regularisierung: siehe [14]). In vielen Modellen, so auch in *YOLO*, wurde vorher mit Dropout regularisiert, um eine Überanpassung (Overfitting) an die Trainingsdaten zu vermeiden (Dropout: siehe [29]; Overfitting: siehe Kapitel 1.3.3 in [7]). Dafür entfernt Dropout zufällig Informationen im Netz, um eine Überanpassung an die Trainingsdaten zu vermeiden. Das ist mit Batch-Normalization weniger stark oder nicht mehr nötig. Außerdem ermöglicht Batch-Normalization höhere Learning-Rates, weil es durch Normalisierung das Entstehen von verschwindend kleinen (vanish) oder sehr hohen (explode) Gradienten verhindert [22].

Ein wichtiger Hinweis der Autoren zu Batch-Normalization besteht in der Empfehlung, die Normalisierung nur während des Trainingsprozesses durchzuführen. Es macht keinen Sinn, diesen Prozess für die Inferenz zu nutzen - das Ergebnis der Inferenz eines Eingangsbildes soll nicht von anderen zu verarbeitenden Bildern, sondern ausschließlich vom Bild selbst abhängen. In den Experimenten aus [35] sind klare Vorteile durch die Nutzung von Batch-Normalization in *YOLO*v2 zu erkennen: Sie bewirkte eine Steigerung von über 2% in der *mAP* Metrik und Dropout konnte entfernt werden, ohne Overfitting zu riskieren.

3.2 High Resolution Classifier

Wie bereits in Kapitel 2.2 beschrieben, wird in *YOLO* erst ein Klassifizierungsmodell mit der Eingangsgröße 224x224 Pixel trainiert, das im Anschluss in ein Object-Detection Modell mit der Eingangsgröße 448x448 konvertiert und weiter trainiert wird. In *YOLO*v2 [35] beschreiben die Autoren, dass das konvertierte Modell im zweiten Training an die Object-Detection und gleichzeitig an die neue Auflösung angepasst werden muss. Daher schlagen sie vor, das Klassifizierungsmodell am Ende des Trainings 10 Epochen mit der hohen Auflösung zu trainieren, um die Filter der Convolutional Layers bereits an die höhere Auflösung anzupassen, bevor das Modell zu einem Objektdetektor umgebaut und erneut weiter trainiert wird.

3.3 Anchor Boxes

In *YOLO* verwenden die Autoren *FC Layers* nach den Convolutional Layers im Object Detection Head, um die Prediction zu erzeugen. In der Weiterentwicklung werden diese sowie der letzte Pooling Layer entfernt. Damit ergibt sich der Output von *YOLO*v2 direkt aus dem letzten Convolutional Layer und auf 14x14 statt 7x7 Zellen.

Außerdem ändern sie die Auflösung der Eingangsbilder erneut auf 416x416 Pixel. Wenn nun die Convolutions angewendet werden, ergibt sich ein Raster aus 13x13 statt 14x14 Zellen. Sie begründen diese Entscheidung damit, dass das Zentrum eines großen Objekts oft im Zentrum des gesamten Bildes liegt. Bei einer ungeraden Zahl an Zellen liegt das Zentrum somit genau in einer Zelle. Bei einer geraden Anzahl an Zellen kann das Zentrum des Objekts genau zwischen vier Zellen liegen, sodass alle vier Zellen für die Erkennung des Objekts zuständig wären.

Eine weitere sehr wichtige Änderung ist die Verwendung von "Anchor Boxes", die von Ren et al. für Faster-*RCNNs* [38] eingeführt werden. Eine Anchor Box ist vergleichbar mit einer Bounding Box in *YOLO*. Ihr Zentrum ist gleichzeitig das der zugehörigen Zelle und sie weist eine festgelegte Größe auf. Die Dimensionen der Anchor Boxes werden im Gegensatz zu den Bounding Boxes aus *YOLO* initial festgelegt und während des Trainings weiter optimiert. Es werden mehrere Anchor Boxes mit verschiedenen Größen und Seitenverhältnissen definiert, in denen sich Objekte befinden können. In diesem Zug legen die Autoren fest, dass in *YOLO*v2 die Klassen nicht mehr pro Zelle erkannt werden, sondern jede Anchor Box sowohl den Objectness Score als auch die Klasse selbst bestimmt. Daraus ergibt sich ein neuer Output-Tensor der Form $(S \times S \times B \times (5+C))$. Für die Bestimmung der Objectness und der Conditional Class Probability wird keine Änderung vorgeschlagen. Die Größe der Anchor Boxes wird im Training angepasst, sodass sie sich, wie auch die Bounding Boxes in in *YOLO*, auf die Dimensionen der zu erkennenden Objekte spezialisieren.

3.4 Dimension Clusters

Redmon et al. beschreiben in *YOLO*v2 [35] zwei Probleme bei der Nutzung von Anchor Boxes. Das erste Problem besteht in der initialen zufälligen Bestimmung der Breite und Höhe der Anchor Boxes. Zwar werden die Dimensionen im Training an die Größen der zu erkennenden Objekte angepasst, jedoch würde es den Trainingsprozess beschleunigen, wenn zu Beginn bereits initial passende Dimensionen bestimmt worden wären. Aus diesem Grund wird ein k-means Clustering mit den Ground-Truth Bounding Boxes der Trainingsdaten durchgeführt, um

optimale Dimensionen der Anchor Boxes zu generieren. Da nicht mit Datenpunkten, sondern mit Boxen gearbeitet wird, kann hier nicht die Entfernung von Datenpunkten verwendet werden, stattdessen kommt die *IoU* zum Einsatz. Folgende Metrik wird für die Distanz verwendet:

$$d(box, centroid) = 1 - IoU(box, centroid) \tag{14}$$

Durch die Verwendung von k-means Clustering können somit sowohl die optimale Anzahl an Anchor Boxes als auch deren Dimensionen bestimmt werden. Die Autoren von *YOLO*v2 stellen heraus, dass die Verwendung von fünf Anchor Boxes die richtige Wahl für ihren Anwendungsfall darstellt.

3.5 Direct location prediction

Ein zweites Problem, das nach der Einführung der Anchor Boxes auftritt, wird als Instabilität des Modells beschrieben. Diese liegt in der Ermittlung der (x, y) Koordinaten nach Vorbild der Region Proposal Networks aus Faster-*RCNN* [38] begründet. Darin gibt es keine Einschränkung darüber, wo eine Anchor Box auf dem Bild sein darf. Eine Prediction-Box kann überall auf dem Bild lokalisiert sein, ungeachtet zu welcher Zelle sie gehört. Ein Modell mit Anchor-Boxes ermittelt in seiner Prediction die Werte t_x und t_y als die Offsets in x und y Richtung. Die Koordinaten (x, y) des Zentrums einer Box auf dem Input-Bild können somit folgendermaßen berechnet werden:

$$\begin{aligned} x &= (t_x * w_a) - x_a \\ y &= (t_y * h_a) - y_a \end{aligned} \tag{15}$$

mit:

$w_a, h_a =$ Breite und Höhe einer Rasterzelle

$x_a, y_a =$ Koordinaten der Rasterzelle, zu der die Box gehört

Wenn das Modell z.B. $t_x = 1$ für eine Anchor Box ermittelt, ist diese um die Breite der Rasterzelle nach rechts verschoben. Es gibt keine Restriktion darüber, wie weit die Boxen verschoben werden können. In *YOLO*v2 soll eine Einschränkung über den Offset eingeführt werden, sodass sich jede Anchor Box nur in einem bestimmten Bereich bewegen kann. Das wird erreicht, indem durch Verwendung der Sigmoidfunktion die Koordinaten der Anchor Box relativ zur Position der zugehörigen Zelle ermittelt werden. *YOLO*v2 ermittelt dann in den Predictions erneut B Bounding Boxes mit je fünf Werten $(t_x, t_y, t_w, t_h, t_o)$. Die Koordinaten der Bounding Box können dann folgendermaßen berechnet werden:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h} \\
Pr(object) &\times IoU(b, object) = \sigma(t_o) \\
(\sigma(t) &= \frac{1}{1+e^{-t}})
\end{aligned}
\tag{16}
$$

mit:

b_x, b_y = Koordinaten der Bounding Box, die das erkannte Objekt beinhaltet

c_x, c_y = Koordinaten der Rasterzelle, zu der die Box gehört

p_w, p_h = Breite und Höhe der Anchor Box (aus k-Means)

b_w, b_h = Breite und Höhe der Bounding Box, die das erkannte Objekt beinhaltet

$\sigma(t_o)$ = Confidence Score der Box

Durch die Behebung der zwei genannten Probleme erreichen die Autoren eine erhebliche Verbesserung gegenüber dem Modell ohne Fehlerbehebung. Die Berechnung ist in Abbildung 12 visualisiert.

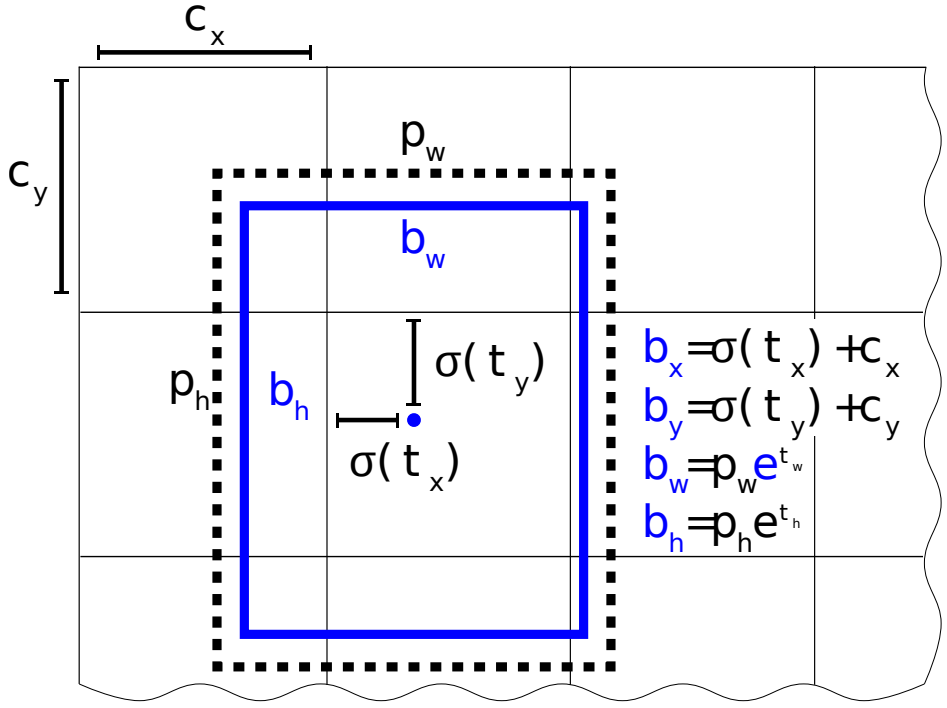


Abbildung 12: Visualisierung der Berechnung der Bounding Boxes aus [35] (angepasst)

3.6 Fine Grained Features und Multi-Scale Training

Durch die Anwendung von Filtern in den Convolutional Layers des *CNNs* sinkt die Auflösung eines Bildes schrittweise, wodurch Schwierigkeiten bei der Erkennung von kleinen Objekten auftreten [35]. Die Autoren versuchen, dieses Problem zu lösen, indem ein sogenannter *Passthrough* eingeführt wird. Statt die Erkennung per Object-Detection-Head nur auf den Outputs des letzten Convolutional Layers (auch Feature-Maps genannt) durchzuführen, verwenden sie auch die Feature-Maps einer vorherigen Schicht. Diese soll mit der höheren Auflösung 26x26 kleine Objekte besser erkennen. Hierfür wird deren Output mit den Dimensionen 26x26x512 in die Form 13x13x2048 konvertiert und an den Output des letzten Convolutional Layers mit der Form 13x13x1024 angehängt. Daraus ergibt sich ein neuer 13x13x3072 Tensor, der für den Object-Detection-Head verwendet werden kann.

In *YOLO*v2 kommen keine *FC Layer*, sondern ausschließlich Convolutional- und Pooling Layer zum Einsatz. Dadurch sind die Output-Dimensionen des Netzes nicht mehr auf die Größe der *FC Layer* festgelegt - die neue Version ist nun in der Lage, Bilder verschiedener Input-Auflösungen zu verarbeiten. Da *YOLO*v2 mit seinen Convolutions den Input um den Faktor 32 reduziert, ist die einzige Restriktion, dass die Höhe und Breite des Eingangsbildes durch 32 teilbar sein muss. Im Training verwenden die Autoren daher alle 10 Batches Eingangsbilder einer anderen zufällig gewählten Auflösung. Dieser Vorgang stellt eine Form der Augmentierung dar und soll dazu führen, dass das Modell in der Lage ist, Informationen besser zu abstrahieren und Bilder verschiedener Input-Dimensionen zuverlässig zu erkennen. Des Weiteren wird die Verwendung von kleineren sowie größeren Bildern in der Inferenz ermöglicht. Die gewählte Auflösung des Eingangsbildes resultiert dann in einem Tradeoff aus Accuracy und Inferenzzeit. Letztere steigt, wenn die Auflösung höher gewählt wird, da dann mehr Operationen, zum Beispiel bei Anwendung der Filter in den Convolutions, durchgeführt werden müssen.

3.7 Netzwerkarchitektur

Für *YOLO*v2 wird eine Netzwerkarchitektur vorgeschlagen, die die Autoren Darknet-19 nennen. In Summe werden 19 Convolutional und 5 Max-Pooling Layer verwendet. Abbildung 13 visualisiert den Aufbau des Darknet-19 Backbones beispielhaft für Eingangsbilder in der Auflösung 448x448. Sie orientieren sich dabei an der von Simonyan und Zisserman [44] vorgestellten VGG16 Architektur. Diese verwendet Convolutional Layers mit Kernels der Größe 3×3. Neben Convolutions verwenden die Autoren Max-Pooling Layers, die die Auflösung der Feature-Map jeweils halbieren. Ein Convolutional Layer, der auf einen Pooling

Layer folgt, verdoppelt jeweils die Anzahl der Channels. *YOLO*v3 orientiert sich zudem auch an der „Network in Network“ Architektur von Lin et al. [24]. Sie verwenden Convolutional Layers mit 1×1 Kernels zwischen direkt aufeinanderfolgenden 3×3 Layers um die Feature Maps zu komprimieren.

In Abbildung 13 ist eine Abgrenzung zwischen den Layers abgebildet. Der erste Teil des Netzes ist für das Klassifizierungsnetz sowie für das Objekterkennungsnetz identisch. Der zweite Teil in der Abbildung zeigt den Teil des Netzes, der für das Training des Klassifizierers verwendet wird. Eine 1×1 Convolution reduziert die Anzahl der Filter auf die Anzahl der möglichen Klassen. Anschließend wird ein Global-Average-Pooling Layer sowie Softmax verwendet, um Predictions für das Klassifizierungsnetzwerk zu erhalten. Dieses Netz wird für das Training des Klassifizierers verwendet, bevor es für das Training des Objekterkennungsmodells angepasst wird. Hierfür ersetzen die Autoren den zweiten Teil des Netzes mit drei 3×3 Convolutional Layers gefolgt von einer 1×1 Convolution, die den gewünschten Tensor für die Objekterkennung generiert.

Type	Filters	Size/Stride	Output
Convolutional	32	3 × 3	224 × 224
Maxpool		2 × 2/ 2	112 × 112
Convolutional	64	3 × 3	112 × 112
Maxpool		2 × 2/ 2	56 × 56
Convolutional	128	3 × 3	56 × 56
Convolutional	64	1 × 1	56 × 56
Convolutional	128	3 × 3	56 × 56
Maxpool		2 × 2/ 2	28 × 28
Convolutional	256	3 × 3	28 × 28
Convolutional	128	1 × 1	28 × 28
Convolutional	256	3 × 3	28 × 28
Maxpool		2 × 2/ 2	14 × 14
Convolutional	512	3 × 3	14 × 14
Convolutional	256	1 × 1	14 × 14
Convolutional	512	3 × 3	14 × 14
Convolutional	256	1 × 1	14 × 14
Convolutional	512	3 × 3	14 × 14
Maxpool		2 × 2/ 2	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	512	1 × 1	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	512	1 × 1	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	1000	1 × 1	7 × 7
Avgpool		Global	1000
Softmax			

Abbildung 13: Darknet-19 Netzwerkarchitektur aus [35]

4 *YOLO*v3

Im Jahr 2018 veröffentlichten Redmon et al. mit *YOLO*v3 [36] ihre letzten Verbesserungen für *YOLO*. Im Februar 2020 verkündete Redmon, dass er seine Forschung im Bereich Computer Vision aufgrund Datenschutzbedenken und die Nutzung seiner Erkenntnisse für militärische Anwendungen einstellt [23]. In *YOLO*v3 werden einige Veränderungen vorgenommen, die das Netz zwar vergrößern, dafür aber die Ergebnisse maßgeblich verbessern. Dabei soll *YOLO*v3 nach wie vor einen sehr schnellen Ansatz darstellen. Die beschriebenen Anpassungen werden im Folgenden zusammengefasst.

4.1 Ermittlung der Bounding Boxes und des Loss

Auch in der Berechnung des Loss schlagen die Autoren Änderungen vor, die die Leistung von *YOLO*v3 verbessern sollen. Sie verwenden hierbei den Begriff Bounding Box für die Box, die durch einen Anchor definiert ist. Die Bounding Box, die die höchste *IoU* mit einem Ground Truth Objekt hat, soll idealerweise einen Objectness Score von 1 aufweisen. Andere Bounding Boxes, die eine Überlappung (*IoU*) mit demselben Ground Truth Objekt aufweisen und deren *IoU* höher als ein Threshold (hier 0,5) ist, werden gänzlich ignoriert. Sie gehen damit nicht in die Berechnung des Loss ein. Durch diese Methode wird pro Ground-Truth-Objekt nur eine Bounding Box berücksichtigt.

Durch die Nutzung des oben genannten Thresholds bleiben letztlich Boxen übrig, die keiner einzigen Ground-Truth zugeordnet wurden. Die *IoUs* mit Ground-Truth Boxen sind geringer als der oben genannte Threshold. Für diese Boxen wird lediglich der Confidence Loss berechnet, nicht jedoch der Localization bzw. Classification Loss. Eine weitere Veränderung gegenüber der Version 2 besteht in der Verwendung der (t_x, t_y) Werte aus der Prediction für die Berechnung des Loss statt der (b_x, b_y) Werte aus Gleichung 16.

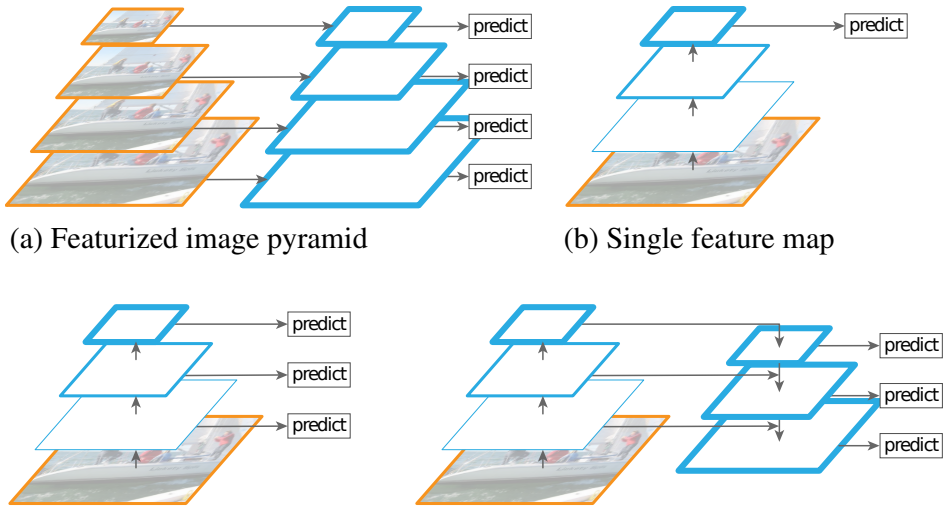
4.2 Ermittlung der Klassen

In *YOLO*v3 wird die sogenannte Multi-Label-Classification angewendet. Dadurch kann ein Objekt nicht mehr zwangsweise nur zu einer einzigen Klasse gehören, sondern auch gleichzeitig zu mehreren. Als Beispiel wird genannt, dass eine Frau auf einem Bild gleichzeitig zur Klasse Frau und zur Klasse Person gehören kann. Aus diesem Grund wird die in den bisherigen Versionen verwendete Softmax-Aktivierung im letzten Layer ersetzt. Stattdessen werden sogenannte *independent logistic classifiers* verwendet, um zu ermitteln, ob ein Objekt zu einem Label gehört. Damit wird für jede Klasse einzeln ermittelt, ob das Objekt zu dieser

gehört, oder nicht - unabhängig von den Werten der anderen Klassen. Als independent logistic classifier kann z.B. die Sigmoid Funktion verwendet werden. Darüber hinaus wird auch die Berechnung des Classification-Loss verändert: statt der *RSS* wird nun der Cross-Entropy-Loss [41] verwendet.

4.3 Feature Pyramids für Predictions auf verschiedenen Auflösungen

In Objekterkennungsmodellen wie *YOLO* wird ein Input-Image im Backbone durch verschiedene Layer verarbeitet, bevor das letzte Ergebnis für den Object-Detection-Head verwendet wird (Abbildung 14 b). In *YOLO*v2 wird bereits zusätzlich eine Feature-Map mit einer höheren Auflösung einbezogen, sodass kleinere Objekte besser erkannt werden. In *YOLO*v3 verfolgen die Autoren dieses Ziel weiter, indem sie Feature Pyramid Networks (*FPNs*) verwenden. Lin et al. [25] stellen das Konzept der *FPNs*, welches in Abbildung 14 d) abgebildet ist, vor. Sie beschreiben, dass es eine Herausforderung in der Objekterkennung darstellt, Objekte verschiedener Größen auf einem Bild zu erkennen. Abbildung 14 a) und c) zeigen verschiedene Möglichkeiten, die in der Vergangenheit genutzt wurden, um dieses Problem zu behandeln.



(a) Featurized image pyramid (b) Single feature map

(c) Pyramidal feature hierarchy (d) Feature Pyramid Network

Abbildung 14: Verschiedene Möglichkeiten für die Nutzung unterschiedlicher Auflösungen für die Objekterkennung aus [25]
a): Verwendung des Bildes in mehreren Auflösungen und mehrfaches Durchlaufen des Netzes (langsam)
b): Nutzung der letzten Feature-Map
c): Nutzung von Feature-Maps aus mehreren Schichten für den Objekt-Detection-Head
d): Feature Pyramid Network

Es wäre möglich, das zu verarbeitende Input-Bild in unterschiedlichen Auflösungen im Netzwerk zu prozessieren (Abbildung 14 a). Diese Möglichkeit beschreiben Lin et al. als sehr langsam, da mehrere Durchläufe benötigt werden.

Eine weitere Möglichkeit wird von Liu et al. in Single Shot Detectors [27] verwendet: Die pyramidenartige Hierarchie der Feature-Maps wird genutzt, um die Feature-Maps der unterschiedlichen Schichten direkt für die Objekterkennung zu verwenden (Abbildung 14 c). Bezogen auf *YOLO* könnten somit z.B. die letzten 3 Feature-Maps direkt mit dem Object-Detection Head verarbeitet werden, um damit unterschiedlich große Objekte zu erkennen. Lin et al. beschreiben jedoch, dass mit der direkten Verwendung der früheren Layers des Backbones die Informationen, die in späteren Layers ermittelt werden, nicht berücksichtigt werden, obwohl sie ohnehin berechnet werden müssen. Gleichzeitig betonen sie die Wichtigkeit ebendieser Informationen, um auch mit den Feature-Maps der früheren Layers kleine Objekte besser zu erkennen.

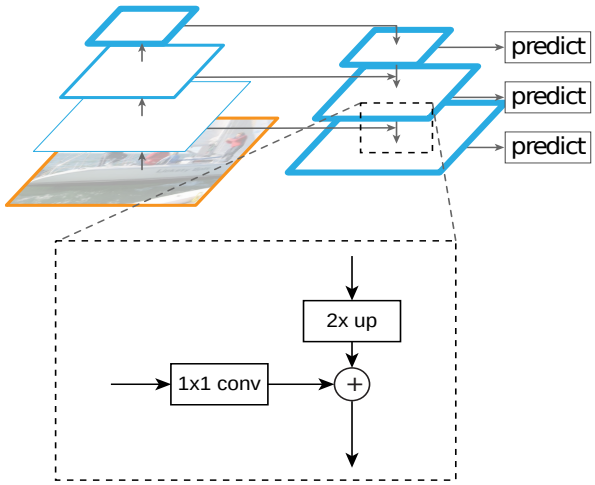


Abbildung 15: Feature Pyramid Network mit Erläuterung der Berechnung aus [25]

Um die genannten Informationen aus späteren Schichten zu nutzen, stellen sie die *FPNs*, deren Idee in Abbildung 15 visualisiert ist, vor. Sie beschreiben, dass der letzte Layer die „semantisch stärksten“ Feature-Maps erstellt, in denen die Bedeutung der Informationen des Input-Bildes am besten und gleichzeitig abstraktesten beschrieben werden. Die letzte Feature-Map wird wie gewohnt und ohne Veränderung für die Objekterkennung vom Object-Detection-Head verarbeitet. Auch vorhergehende Feature-Maps sollen für den Object-Detection Head verwendet werden. Jedoch besteht das Ziel im *FPN* darin, auch die Informationen aus den semantisch stärkeren Feature-Maps einzubeziehen. So soll zum Beispiel die vorletzte Feature-Map aus Abbildung 15 (links) auch die Informationen der letzten Feature-Map nutzen. Hierzu wird die Auflösung der letzten Feature-Map durch ein Upsamling erhöht sowie die Anzahl der

Channels durch eine 1×1 Convolution an die der vorhergehenden Feature-Map angepasst. Anschließend werden die Werte elementweise auf die mittlere Feature-Map addiert. Das Ergebnis wird anschließend für die Objekterkennung vom Object-Detection-Head verarbeitet. Derselbe Vorgang wird für die untere Feature-Map aus Abbildung 15 angewendet, hier wird die im ersten Schritt entstandene mittlere Feature-Map angepasst und die Werte elementweise addiert.

FPNs werden in *YOLO*v3 [36] verwendet, jedoch verändern die Autoren die Erstellung der Feature Pyramid. Statt die in Abbildung 15 abgebildete elementweise Addition, verwenden sie Konkatenation und erhöhen damit die Anzahl der Channels. Wie bereits beschrieben erstellt der Object-Detection-Head die Prediction der Form $(S, S, (B \times 5 + C))$. Wenn eine Feature-Map mit doppelter Auflösung mit dem Object-Detection-Head verarbeitet wird, entsteht ein $(2 \times S, 2 \times S, (B \times 5 + C))$ Tensor. Äquivalent berechnen sich die Dimensionen der Tensoren weiterer Auflösungen. Damit erzeugt *YOLO*v3 Predictions auf drei verschiedenen Auflösungen. Wichtig für die Verwendung mit *YOLO* ist, dass für die Erkennung verschieden großer Objekte durch die genannten Auflösungen auch unterschiedliche Anchor Boxes für die Feature-Maps verwendet werden sollten. Große Anchor sollen sich auf die Erkennung von großen Objekten spezialisieren und werden für die Feature-Maps mit der kleinsten Auflösung verwendet. Analog werden die mittelgroßen und kleinen Anchors auf die beiden anderen Output-Tensoren verteilt.

4.4 Netzwerkarchitektur

In der dritten Version von *YOLO* wird ein um einiges tieferes *CNN* vorgeschlagen. Im Gegensatz zu 19 Convolutional Layers werden im neuen Modell 53 Convolutions verwendet. Außerdem verwenden die Autoren erstmals sogenannte Residual-Connections. Die neue Architektur nennen sie Darknet-53, sie ist in Abbildung 16 visualisiert.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Abbildung 16: Darknet-53 Netzwerkarchitektur aus [36]

Die Idee der Residual-Connections wird von He et al. [20] beschrieben: Sie beobachten in ihren Experimenten, dass die Fehlerrate eines Netzes nach einer Schwelle an Layers nicht mehr sinkt, sondern erst stagniert und mit noch mehr Layers sogar steigt. Dieses Phänomen wird als „Degradation Problem“ bezeichnet. In Abbildung 17 ist die Fehlerrate im Training (links) und im Test (rechts) eines Netzwerks zur Klassifizierung von Bildern, das mit dem CIFAR-10 Datensatz [5] trainiert wurde, zu sehen.

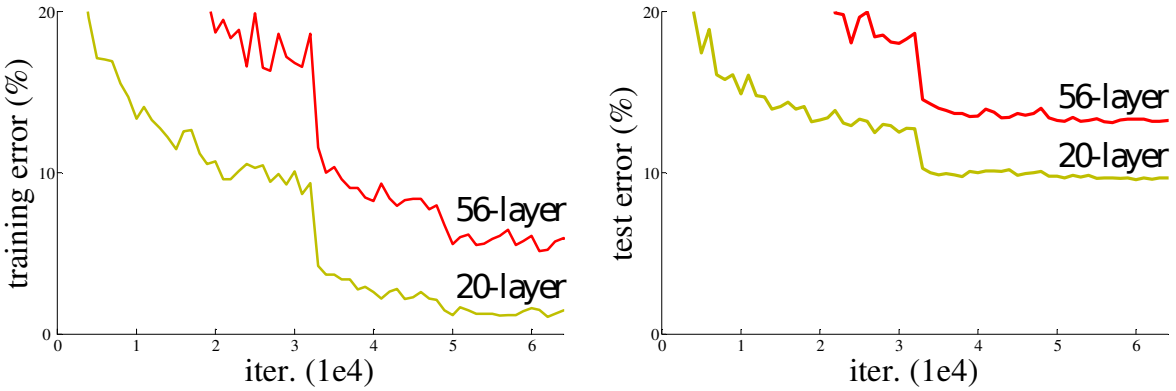


Abbildung 17: Fehlerrate im Training (links) und im Test (rechts) eines Klassifizierungsnetzes, trainiert mit CIFAR-10 ([5]), Abbildung aus [20]

Dennoch vermuten die Autoren, dass es auch nach dieser Schwelle noch tiefe Netze geben muss, die keinen höheren Fehler erzeugen, als flachere Netze. Sie beweisen dies, indem sie zuerst ein flaches Netz trainieren. Im Anschluss wird ein tieferes Netz erstellt, das die Layer des flachen Netzes erweitert. Hierfür werden die Layer des flachen, trainierten Netzes mit ihren Gewichten kopiert und neue Layer in Form von Identitätsabbildungen hinzugefügt. Damit wird ein tiefes Netz erzeugt, dessen Fehlerrate nicht höher ist, als die des flachen Äquivalents.

Es ist demnach zwar möglich, tiefere Netze ohne steigende Fehlerrate zu erzeugen, jedoch steigt die Fehlerrate im Training eines tiefen Netzes dennoch an. Das führt zu folgender Annahme: Im Training eines neuronalen Netzes werden normalerweise keine Schichten so optimiert, dass sie Identitätsabbildungen darstellen. Wenn doch, würde das Problem aus Abbildung 17 nicht auftreten, sondern das Verhalten, das im Experiment konstruiert wurde, erreicht werden.

Neben diesem konstruierten Beweis führen die Autoren die Residual-Blocks zur Lösung des Problems des Trainings sehr tiefer Netzwerke ein. Abbildung 18 zeigt die Idee hinter diesen. Sie besteht darin, eine Sequenz aus Schichten in Form eines Blocks zu modellieren und eine Skip-Connection einzuführen, die eine Art Bypass darstellt. Dieser übergibt den Input an das Ende des Blocks. Um eine Identitätsrelation zu generieren, müssen während des Trainings die Gewichte der Schichten im Block gegen 0 optimiert werden. Damit ist der Input des Residual-Blocks auch dessen Output. Diese Vorgehensweise ermöglicht jedoch keine optimale Identitätsrelation, da die Gewichte nie genau auf den Wert 0 optimiert werden. Die Ergebnisse zeigen allerdings, dass es ausreichend ist, eine annähernde Identitätsrelation zu approximieren.

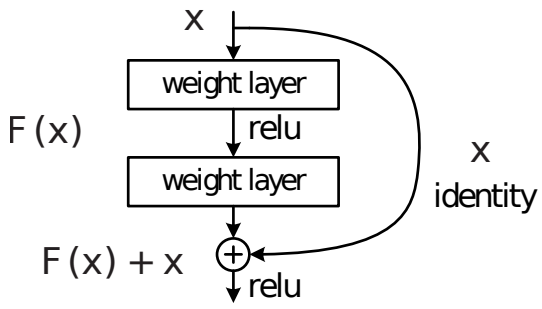


Abbildung 18: Allgemeine Darstellung eines Residual Blocks aus [20]

Formal kann die Lösung folgendermaßen beschrieben werden. Wenn eine Distribution $H(x)$ mithilfe von mehreren Layers approximiert werden soll, beschreibt x den Input und $H(x)$ den gewünschten zu approximierenden Output. Der Unterschied zwischen Output und Input wird als „Residual“ bezeichnet und folgendermaßen beschrieben:

$$F(x) = Output - Input = H(x) - x \tag{17}$$

Daraus folgt:

$$H(x) = F(x) + x \tag{18}$$

Wobei $F(x)$, wenn nötig, gegen 0 optimiert wird. Abbildung 19 zeigt schematisch einen Residual Block, der drei Convolutional Layers beinhaltet.

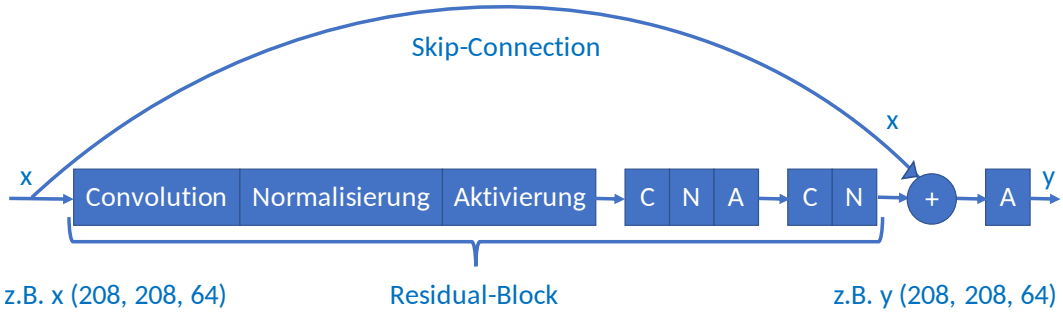


Abbildung 19: Residual Block mit drei Convolutional Layers und Skip-Connection

In *CNNs* haben nicht alle Schichten dieselben Input Dimensionen. Wenn eine Skip-Connection den Input eines Blocks an den auf den Block folgenden Layer übergibt, kann letzterer, wie in Abbildung 20, andere Dimensionen (Auflösung oder Anzahl der Feature-Maps) erwarten. Die Autoren beschreiben eine Lösung für dieses Problem darin, dass entweder zusätzliche Channels mit Nullen gefüllt und die Auflösung durch einen Stride (z.B. nur alle 2 Werte übernehmen) reduziert werden kann. Eine weitere Lösung besteht in der Nutzung einer (1×1) Convolution in der Skip-Connection (Abbildung 20). Diese kann sowohl die Anzahl der Channels als auch die Auflösung an die des nachfolgenden Layers anpassen. Ähnlich

zum einfacheren ersten Lösungsansatz wird bei einer Strided Convolution eine Schrittweite *n* (von z.B. zwei) genutzt, um nur alle n Werte für die Convolution zu verwenden und somit die Auflösung zu reduzieren.

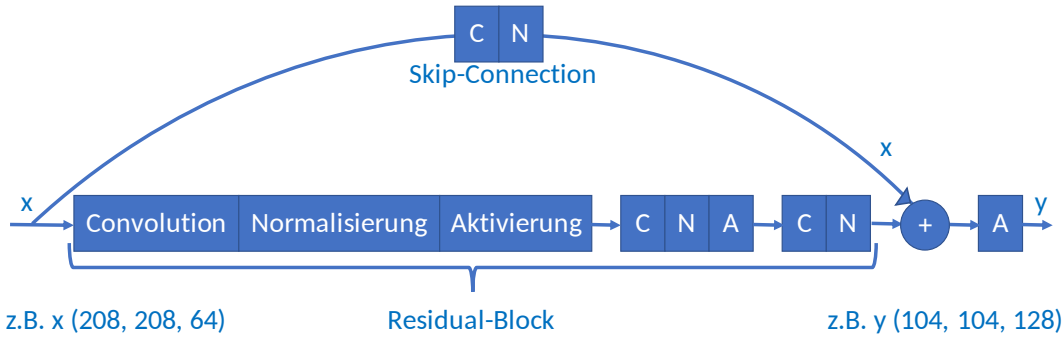


Abbildung 20: Residual Block mit drei Convolutional Layers und Skip-Connection, Dimensionen verändert

Ein weiteres Problem der Residual-Blocks ergibt sich aus der Verwendung einer hohen Anzahl an Channels im Input. Die Schichten im Residual-Block resultieren in vielen zu trainierenden Gewichten, deren Optimierung die Trainingszeit stark erhöht. Um dieses Problem zu umgehen, verwenden die Autoren ein Design, das sie Bottleneck Architektur nennen. Statt im Residual-Block nur 3x3 Convolutions zu verwenden und die Anzahl der Channels nicht zu verändern, schlagen sie Folgendes vor: Die Anzahl der Channels kann erst mit einer 1x1 Convolution verringert und anschließend die 3x3 Convolution wie gewohnt durchgeführt werden. Im Anschluss wird die Anzahl der Channels mit einer weiteren 1x1 Convolution wieder an die Ursprüngliche angepasst. In Abbildung 21 ist der Unterschied der genannten Ansätze ersichtlich.Wie in Abbildung 16 zu sehen ist, wird auch in Darknet-53 eine 1x1 Convolution verwendet, um die Anzahl der Channels in den Residual-Blocks zu verringern, jedoch wird in den 3x3 Convolutions die Anzahl dieser ohne weitere 1x1 Convolution wieder auf die Ursprüngliche erhöht.

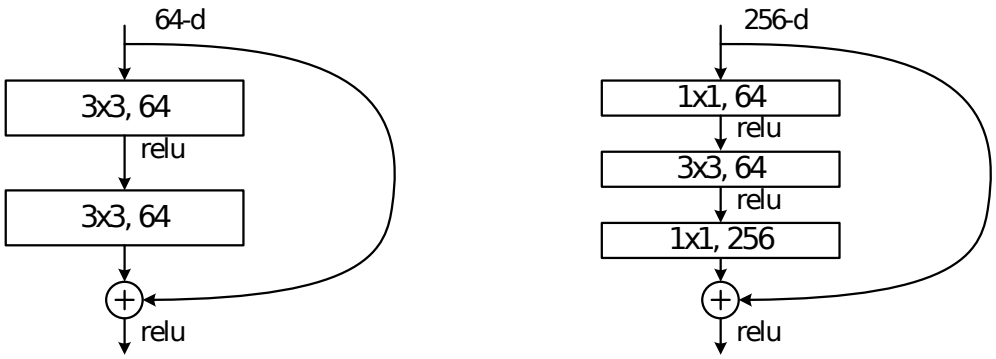


Abbildung 21: links: Residual-Block mit 64 Channels, rechts: Bottleneck Residual Block mit 256 Channels (aus [20])

5 Detailbetrachtung der Verfahren in *YOLO*v4

Nachdem Redmon seine Arbeit an *YOLO* eingestellt hatte, kam die Frage auf, ob es weitere Versionen mit Verbesserungen von anderen Autoren oder der Community geben würde. Im April 2020 veröffentlichten Bochkovskiy et al. eine Weiterentwicklung unter dem Titel *YOLO*v4 [2]. Mit der neuen Version verfolgen sie das Ziel, die Objekterkennung noch effizienter und besser zu gestalten und das Training auf einer Graphics Processing Unit (*GPU*) zu ermöglichen. Dabei bedienen sie sich an verschiedenen Methoden, die für die verbesserte Objekterkennung von anderen Autoren veröffentlicht wurden. Diese werden gegebenenfalls angepasst, um sie für das Training auf einer einzigen *GPU* zu optimieren. *YOLO* is zwar ein weit verbreitetes Verfahren zur Objekterkennung, jedoch hängt die Wirksamkeit verschiedener Methoden auch stark mit dem jeweiligen Anwendungsfall zusammen - es gibt kein „bestes“ Verfahren, das in allen Anwendungsfällen die besten Ergebnisse liefert (siehe [4]).

5.1 Bag of Freebies

In *YOLO*v4 teilen die Autoren ihre verwendeten Methoden in „Bag of Freebies“ und „Bag of Specials“ ein. Im Bag of Freebies werden Methoden zusammengefasst, die die Qualität der Objekterkennung verbessern und ggf. die Trainingszeit verlängern aber keinen Einfluss auf die Inferenzzeit haben. Das Bag of Specials fasst dagegen Methoden zusammen, die die Inferenzzeit zwar erhöhen, die Qualität des Objekterkennungsmodells jedoch maßgeblich verbessern können.

5.1.1 Mosaic Data Augmentation

Augmentierung ist ein häufig genutztes Instrument, mit dem die Menge an Trainingsdaten künstlich erhöht und die Art derer manipuliert werden kann. Durch z.B. Drehen, Verschieben bzw. hinzufügen von Filtern oder Rauschen werden neue Trainingsbeispiele erzeugt, die sich von den anderen Trainingsbeispielen unterscheiden. Im Training bewirkt dies, dass durch Unterschiede in den Daten Overfitting und ein damit verbundener Performance-Verlust bei der Verwendung auf trainingsfremden Bildern vermieden wird. In *YOLO*v4 wird mit Mosaic eine neue Art der Augmentierung eingeführt, bei der vier Trainingsbilder mit ihren Ground-Truth Labels zu einem zusammengefasst werden. Hierzu können sie skaliert und nebeneinander bzw. untereinander gesetzt werden, sodass wieder ein Bild in der fürs Training benötigten Auflösung entsteht. Das führt laut Bochkovskiy et al. [2] dazu, dass das Objekterkennnungssystem auch lernt, Objekte außerhalb des Kontexts zu erkennen, in dem sie auf den Trainingsdaten auftreten. Des Weiteren bewirkt das Zusammensetzen von vier Bildern, dass statt einem gleich vier

Bilder für die Batch-Normalisation verwendet werden. Aus diesem Nebeneffekt folgern sie die Möglichkeit einer Verringerung der Mini-Batch-Size mit geringen Einbußen in der Normalisierung. Der daraus entstehende Vorteil zielt auf die Idee ab, *YOLO*v4 auf einer einzigen *GPU* zu trainieren. Man kann vermuten, dass Mosaic bei Anwendungsfällen, in denen die zu erkennenden Objekte tatsächlich in unterschiedlichen Kontexten auftreten und die Trainingsdaten nur einen Teil dieser Kontexte abbilden, sinnvoll ist. Abbildung 22 zeigt ein Beispiel der Augmentierung für die Verwendung mit Kassenrezepten.

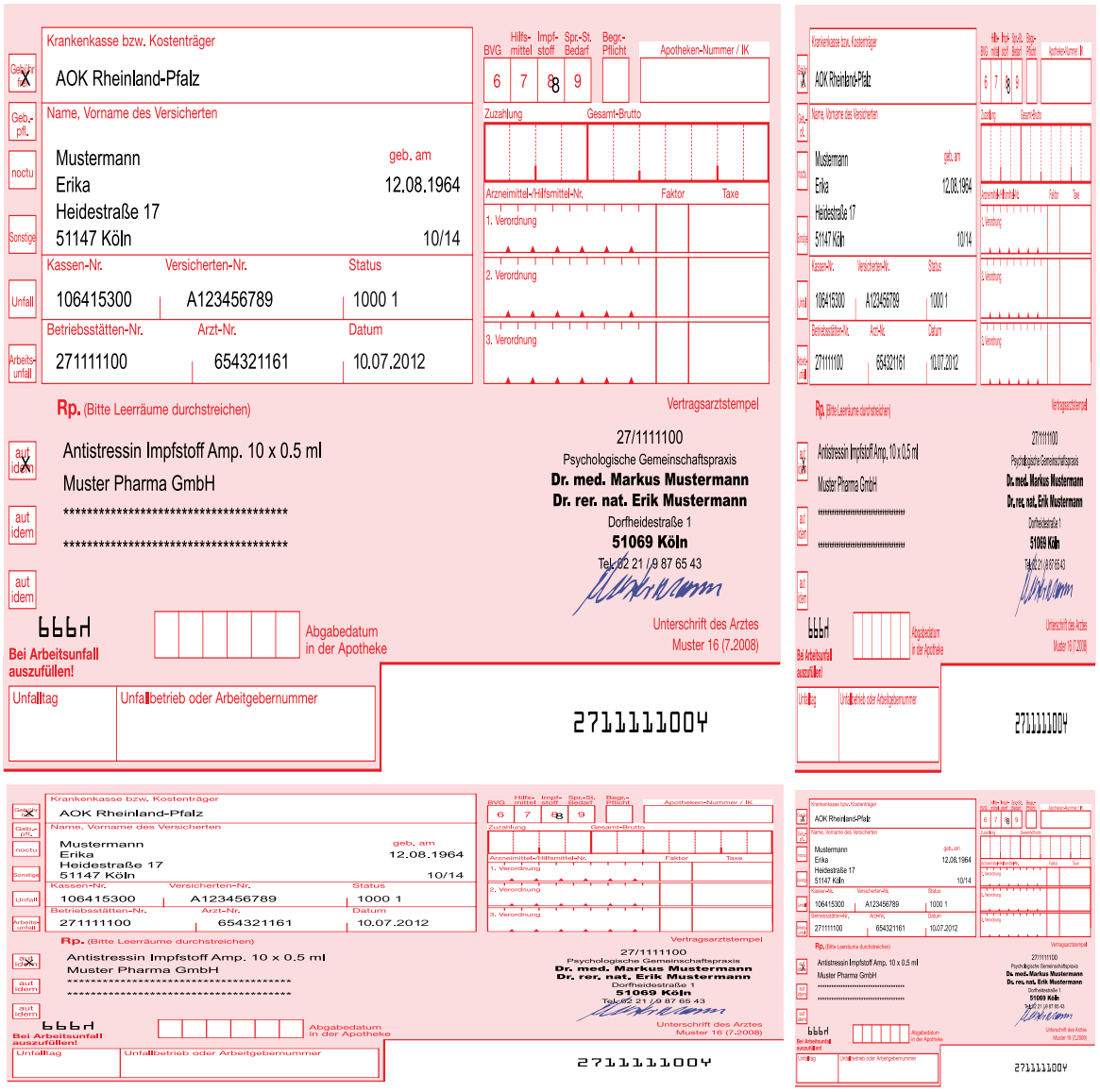


Abbildung 22: 4 identische Muster-Rezeptbilder aus [9], Mosaic augmentiert

In ihrer Veröffentlichung äußern die Autoren keine Empfehlung darüber, wie groß der Anteil der Trainingsbilder ist, die augmentiert werden und ob die Originale weiterhin im Trainingsdatensatz enthalten bleiben oder entfernt werden sollen. Wenn die Originalbilder, die zur

Augmentierung verwendet wurden, aus dem Trainingsdatensatz entfernt werden, wird mit der Mosaic-Augmentierung die Trainingszeit verkürzt, da statt vier Bildern nur noch ein Bild für das Training verwendet wird. Werden die Originale nicht entfernt, wird die Trainingszeit verlängert. In beiden Fällen hat Mosaic keinen Einfluss auf die Inferenzzeit, was die Zugehörigkeit im Bag of Freebies begründet.

5.1.2 DropBlock Regularization

In der ersten Version von *YOLO* (Kapitel 2) wurde Dropout für die Regularisierung verwendet. In *YOLO*v2 (Kapitel 3) wurde Batch Normalization eingeführt, wodurch Dropout nicht mehr verwendet werden musste. DropBlock ist eine Technik zur Regularisierung, die von Ghiasi et al. [13] vorgestellt wird und eine effektivere Form von Dropout darstellen soll. Sie beschreiben den Nachteil von Dropout darin, dass es zufällig Informationen aus dem Netzwerk löscht. Diese Vorgehensweise soll zwar in *FC Layers* sinnvoll sein, jedoch verliert der Effekt in Fully Convolutional Neural Networks, wie sie seit der zweiten Version auch in *YOLO* zum Einsatz kommen, an Bedeutung. In Convolutional Layers sind die Informationen auf Input Feature-Maps räumlich korreliert. Das Entfernen zufälliger Bildpunkte verliert an Effektivität, weil die korrelierten Informationen, z.B. die Pixel unmittelbar neben einem Entfernten, erhalten bleiben. Damit wird der Informationsgehalt nicht maßgeblich reduziert und Overfitting wird nicht effektiv verhindert. DropBlock soll dieses Problem lösen, indem nicht einzelne, sondern zufällig gewählte zusammengehörige Informationen entfernt werden. Das wird erreicht, indem gleich ein ganzer räumlicher Bereich (Block) an Informationen in den Feature-Maps der Convolutional Layer entfernt werden. Damit wird die Idee hinter Dropout auch für Convolutional Layer wieder sinnvoll. Abbildung 23 zeigt ein Beispiel beider Regularisierungen im Vergleich. Die Autoren beschreiben im DropBlock Algorithmus auch, dass, wie bei anderen Methoden der Regularisierung, die Technik nur zur Trainingszeit angewendet werden soll. Der Pseudocode ist in Abbildung 24 ersichtlich.

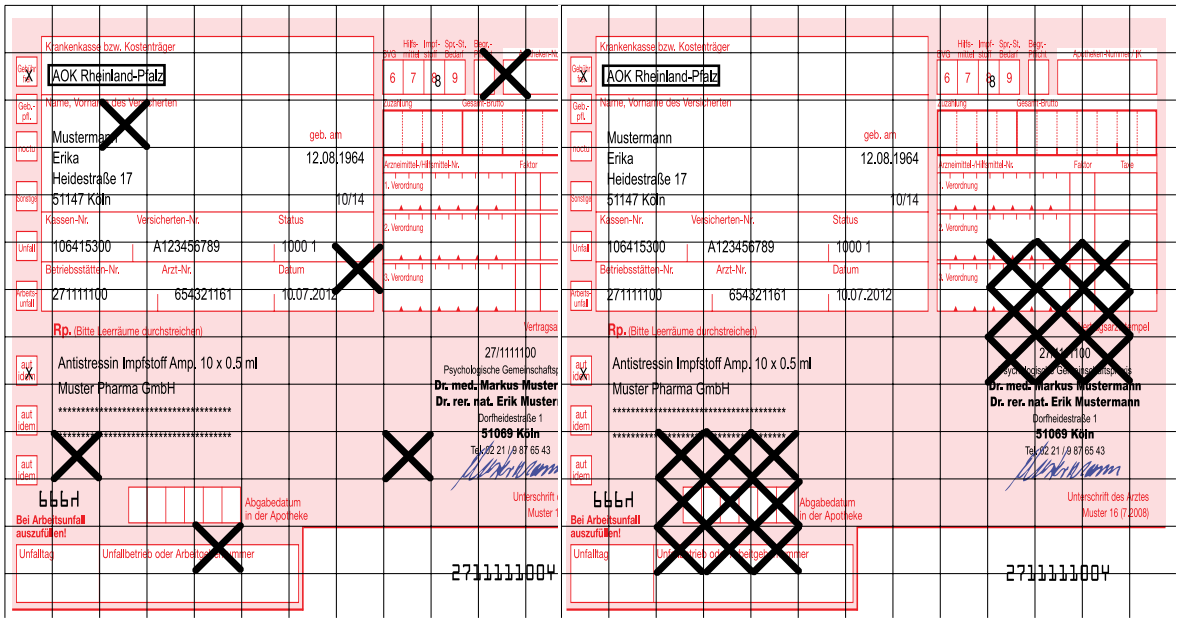


Abbildung 23: Beispielhafter Vergleich zwischen Dropout (links) und DropBlock (rechts) auf einem Kassenrezept

Data: output activations of a layer (A), $block\_size$, γ, $mode$

if $mode == inference$ **then**
  return A

end

Randomly sample mask M : $M_{i,j} \sim Bernoulli(\gamma)$;

For each zero position $M_{i,j}$, create a spatial square mask with the center being $M_{i,j}$, the width, height being $block\_size$ and set all the values of M in the square to be zero ;

Apply the mask: $A = A \times M$;

Normalize the features: $A = A \times \mathbf{count}(M)/\mathbf{count\_ones}(M)$;

Abbildung 24: DropBlock Algorithmus aus [13]

Für DropBlock werden γ Positionen $M_{i,j}$ zufällig gewählt und als Zentrum eines Blocks betrachtet. Die Informationen des Blocks mit dem Zentrum $M_{i,j}$ und der Größe $block\_size$ werden anschließend entfernt. Für den Algorithmus werden sowohl eine Größe für die zu entfernenden Blöcke $(block\_size)$ als auch ein γ als Anzahl der Blöcke, die entfernt werden sollen, benötigt. Die Autoren setzen eine feste Größe der Blöcke fest, jedoch soll γ nicht definiert werden, sondern aus der $block\_size$, der $feat\_size$, also der Größe der Feature-Map und einer anderen festzulegenden Variable $keep\_prob$ folgendermaßen errechnet werden:

$$\gamma = \frac{1 - keep\_prob}{block\_size^2} \frac{feat\_size^2}{(feat\_size - block\_size + 1)^2} \tag{19}$$

Außerdem wird *keep_prob* nicht für das ganze Training festgelegt, sondern soll während des Trainings vom Wert 1 zu einem gewünschten Zielwert reduziert werden. Die Idee, dass der Wert linear sinkt, bis der Zielwert erreicht ist, wird ursprünglich in einer anderen Regularisierungsmethode namens ScheduledDropPath [57] eingeführt. DropBlock wird nur während des Trainings verwendet. Damit hat es außer der Abfrage, ob gerade trainiert wird, oder Inferenz stattfindet, keine Auswirkung auf die Inferenzzeit und gehört somit zum Bag of Freebies von *YOLO*v4.

5.1.3 CutMix Regularization und Augmentation

Wie auch Mosaic ist CutMix eine Art der Augmentierung, bei der bestehende Bilder des Trainingsdatensatzes zusammengesetzt werden. Jedoch stellt es gleichzeitig eine Art der Regularisierung dar, da im Gegensatz zu Mosaic in CutMix die Bilder mit einem Verlust von Informationen kombiniert werden. Die Methode wird von Yun et al. [53] vorgestellt und kombiniert zwei Bilder und ihre Ground-Truth Labels miteinander, indem ein Bereich des einen Bildes herausgeschnitten und durch einen gleichgroßen Bereich eines anderen Bildes ersetzt wird. Abbildung 25 zeigt eine solche Kombination zweier Bilder.

Abbildung 25: Beispiel für die Kombination zweier Bilder mit CutMix aus [53]

CutMix ist dadurch vergleichbar mit der DropBlock Regularisierung, wenn sie auf den Input des Modells angewendet werden würde. In DropBlock werden jedoch Pixel entfernt, sodass sie keine Informationen mehr beinhalten, in CutMix werden in diesen Bereichen wieder neue Informationen hinzugefügt. Damit sollen laut der Autoren die Vorteile der Regularisierung aus dem Entfernen von Bereichen (und damit das Verhindern von Overfitting) behalten werden, während das Training effizienter wird, da die Bereiche wieder Informationen enthalten, die ins Training einfließen können. Auch in dieser Methode werden Objekte bzw. Bereiche aus verschiedenen Bildern in einen neuen Kontext gesetzt. Das bewirkt einen ähnlichen Effekt, wie in der Mosaic Augmentierung.

5.1.4 *CLS*

Eine weitere Regularisierung, die in *YOLO*v4 verwendet wird, ist die Class Label Smoothing Regularization, die von Szegedy et al. [46] veröffentlicht wurde. Sie beschreiben ein Problem in der Modellierung der Ground-Truth Labels der Trainingsdaten. Wie in Kapitel 2.1 beschrieben, werden für jede Prediction C (= Anzahl der Klassen) Klassenwahrscheinlichkeiten ermittelt - diese Werte sind auch in den Ground-Truth-Labels modelliert. Für ein Objekt O beinhaltet der Vektor C an der Stelle der Klasse c_{true}, zu der O gehört, den Wert $y = 1$ und für alle anderen Klassen den Wert $y = 0$. Diese Art der Modellierung kann laut den Autoren zu Overfitting führen, da das Modell im Training lernt, der richtigen Klasse auch in der Prediction eine Wahrscheinlichkeit nahe des Wertes 1 zuzuweisen. Sie beschreiben, dass damit keine Verallgemeinerung mehr garantiert werden kann - das Modell kann sich extrem an die Trainingsdaten anpassen.

Ein weiteres Problem besteht im maximalen Unterschied zwischen dem Wert 1 der Ground-Truth Klasse und dem Wert 0 für alle anderen Klassen. Die Werte, die maximale Unterschiede haben, werden für die Berechnung von Softmax verwendet und führen dazu, dass auch nach der Softmax-Aktivierung maximale Unterschiede vorhanden sind. Diese großen Unterschiede in Kombination mit den Gradienten, die im Training nicht beliebige Werte annehmen können, sondern begrenzt sind, sollen dazu führen, dass das Modell weniger anpassungsfähig wird. Durch die genannten Unterschiede in den Werten ist es zu stark „überzeugt“ von seinen Predictions.

Um diese Probleme zu lösen, schlagen sie vor, die Werte der Ground-Truth Labels so zu verändern, dass die Klasse c_{true} einen Wert knapp unter 1 und die anderen Klassen einen Wert knapp über 0 aufweisen. Sie nennen die neuen Ground-Truth Label „Smoothed Labels“. Durch

die Veränderung der Klassenwahrscheinlichkeiten beeinflusst *CLS* den Classification-Loss von *YOLO*. Folgende Formel beschreibt die Berechnung der y Werte mit Label Smoothing:

$$y_{ls} = (1-\alpha) * y_{hot} + \alpha/K \tag{20}$$

Der Parameter α wird Smoothing-Parameter genannt. Er bestimmt, wie stark die Werte angepasst werden sollen. Bei $\alpha = 0$ werden die Werte nicht angepasst, bei $\alpha = 1$ wird eine Gleichverteilung über alle Klassen generiert. Folgende Berechnung zeigt, wie die Labels angepasst werden, wenn es 5 Klassen gibt, bei der c_{true} die dritte Klasse ist.

Data: smoothing Factor (α)
$y_{hot} = [0.0, 0.0, 1.0, 0.0, 0.0]$;
$K = 5$;
$\alpha = 0.1$;
$y_{ls} = (1-\alpha) * y_{hot} + \alpha/K$;
$-> [0.02, 0.02, 0.92, 0.02, 0.02]$

Abbildung 26: Beispielhafte Berechnung von Smoothed Labels

5.1.5 *CIoU* Loss

Auch für die Berechnung des Loss sollen in *YOLO*v4 neue Erkenntnisse im Bereich der Objekterkennung einfließen. So beschreiben Zheng et al. [56] mit dem *DIoU* Loss und dem *CIoU* Loss neue Techniken zum Berechnen eines Localization-Loss. Ihre Arbeit basiert auf den Erkenntnissen von Rezatofighi et al. [39], die den sogenannten Generalized *IoU* (*GIoU*) Loss für die Regression von Bounding Boxes einführen. Zheng et al. beschreiben die Basis, auf der ihre Arbeit beruht, folgendermaßen: Um die *IoU* in der Berechnung des Loss zu berücksichtigen, kann der *IoU* Loss als die euklidische Distanz zwischen der Ground-Truth Bounding Box und der Prediction Bounding Box ähnlich wie in Kapitel 3.4 mit folgender Formel berechnet werden:

$$L_{IoU} = 1 - \frac{|B \cap B^{gt}|}{|B \cup B^{gt}|} \tag{21}$$

Der *IoU* Loss funktioniert jedoch nur, wenn die Bounding Boxes bereits einen Überlappungsbereich haben. Wenn dies nicht der Fall ist, kann kein Gradient berechnet werden, der eine Richtung für die Optimierung angibt. Insbesondere zu Beginn des Trainings tritt dieser Fall sehr häufig auf, ohne Gradienten wird kein Training möglich.

Rezatofighi et al. [39] lösen dieses Problem mit dem *GIoU* Loss, indem sie einen Bestrafungsterm (Penalty) für diesen Fall einführen, der bewirkt, dass die Prediction-Box sich in Richtung der Ground-Truth Box bewegt, wenn sie noch nicht überlappen:

$$L_{GIoU} = 1 - IoU + \frac{|C - B \cup B^{gt}|}{|C|} \tag{22}$$

Dafür steht C für die kleinstmögliche Fläche, die B und B^{gt} abdeckt (siehe Abbildung 27).

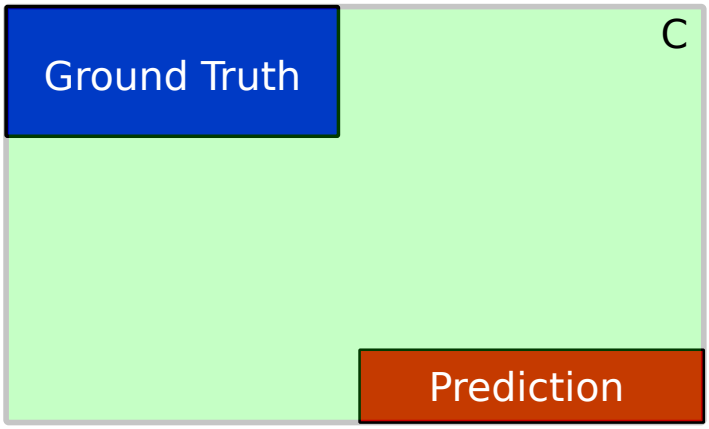


Abbildung 27: Flächen, die für die Berechnung des *GIoU* benötigt werden

Wenn die Boxen nach mehreren Schritten eine Überlappung aufweisen, berechnet der *GIoU* Loss denselben Wert wie der *IoU*-Loss. Das Problem, das Zheng et al. im *GIoU* Loss sehen, ist in Abbildung 28 demonstriert. Die Nutzung des *GIoU* Loss führt dazu, dass die Prediction-Box erst so weit vergrößert wird, bis es zu einer Überlappung mit der Ground-Truth kommt, bevor dann die Größe wieder an die Ground-Truth Box angepasst wird. Mit dieser Vorgehensweise werden sehr viele Iterationen nötig, um ein Ergebnis zu erhalten.

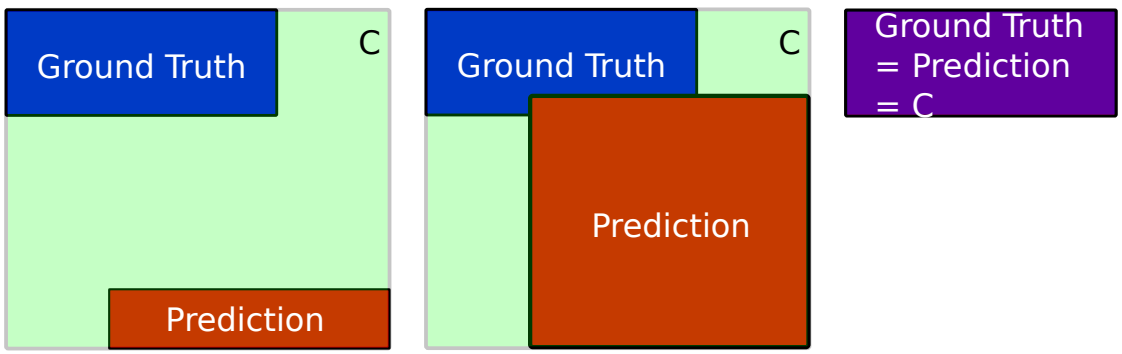


Abbildung 28: Predicton und Ground-Truth Boxes im Verlauf der Iterationen bei Verwendung des *GIoU* Loss

Zheng et al. führen aus diesen Gründen mit dem sogenannten *DIoU* Loss einen Bestrafungsterm ein, der dazu führen soll, die normalisierte Distanz der Zentrum-Punkte zweier Bounding Boxes zu minimieren. Damit sollen die Boxen viel schneller trainiert werden können, als mit den *GIoU* Loss. Der *DIoU* Loss wird folgendermaßen berechnet:

$$L_{DIoU} = 1 - IoU + \frac{\rho^2(b, b^{gt})}{c^2} \tag{23}$$

Die Werte b und b^{gt} stellen die Zentren der Boxen B und B^{gt} dar, c ist die Diagonale der kleinsten Box, die B und B^{gt} umschließt. ρ ist der euklidische Abstand, also in diesem Fall die Länge der Gerade, die die beiden Zentren miteinander verbindet. Die Punkte und Abstände werden in Abbildung 29 verdeutlicht.

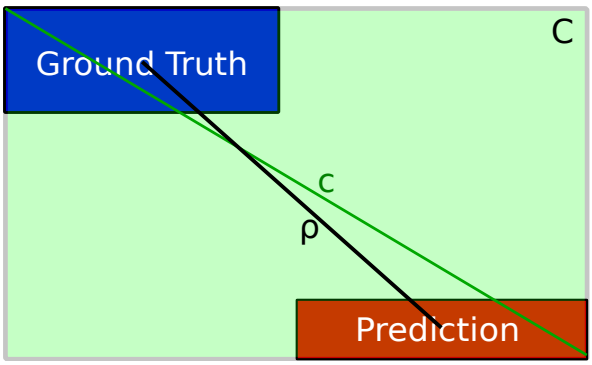


Abbildung 29: Abstände für die Berechnung des DIoU-Loss

Mit dem *DIoU* Loss kann bereits die Überlappungsfläche sowie der Abstand der Zentren optimiert werden. Laut den Autoren fehlt für eine gute Bounding Box Regression noch ein Faktor: Das Optimieren des Seitenverhältnisses. Für das Erreichen des letzten Punktes definieren sie, basierend auf dem *DIoU* Loss, den *CIoU* Loss mit einem weiteren Bestrafungsterm für die Seitenverhältnisse folgendermaßen:

$$L_{CIoU} = 1 - IoU + \frac{\rho^2(b, b^{gt})}{c^2} + \alpha v \tag{24}$$

v misst die Art des Seitenverhältnisses:

$$v = \frac{4}{\pi^2}(arctan\frac{w^{gt}}{h^{gt}} - arctan\frac{w}{h})^2 \tag{25}$$

α wird als „Trade-Off Parameter“ beschrieben, der den Faktor der Überlappungsfläche höher gewichtet, insbesondere wenn die Boxen noch nicht überlappen:

$$\alpha = \frac{v}{(1 - IoU) + v} \tag{26}$$

In *YOLO*v4 wird empfohlen, für die Berechnung des Localization-Loss den *CIoU* Loss zu verwenden. Mit diesem soll es möglich sein, schneller die Überlappungsfläche und den Abstand der Zentren von Ground-Truth und Prediction zu optimieren sowie die Seitenverhältnisse anzupassen.

5.1.6 *SAT*

Szegedy et al. [45] beschreiben eine Schwäche in der Stabilität von neuronalen Netzen bezüglich kleiner Veränderungen des Inputs. So können Predictions, wider Erwarten, durch gezielte, für das menschliche Auge nicht sichtbare Störungen auf Bildern (Adversarial Attacks) willkürlich manipuliert werden. Das gilt unter anderem auch für moderne tiefe neuronale Netze, zum Beispiel im Anwendungsgebiet der Objekterkennung. Im Gegensatz dazu würde man erwarten, dass ein Netz, das in der Lage ist, Informationen sinnvoll zu abstrahieren und unveränderte Eingangsbilder zuverlässig zu erkennen, robust gegen derartige geringfügige Störsignale ist.

Es ist möglich, derartige Störungen zu generieren, indem in einem Trainingsschritt der Fehler des Modells durch Optimierung der Pixel des Eingangsbildes (statt der Gewichte des Modells) maximiert (statt minimiert) wird. Bilder, die auf diese Weise manipuliert werden, nennen sie „Adversarial Examples“. Diese Bilder haben eine sehr wichtige Eigenschaft, die ebenfalls von Szegedy [45] ermittelt wird: Sie sind sehr robust in Bezug auf verschiedene neuronale Netze. So stellt ein Adversarial Example, das gezielt für ein neuronales Netz erstellt wurde, auch für Netze mit anderen Schichten, Hyperparametern und sogar anderen Trainingsdaten weiterhin eine Herausforderung dar. Szegedy et al. generieren in ihren Experimenten verschiedene Adversarial Examples für mehrere Datensätze sowie Netzarchitekturen und zeigen die daraus resultierenden Fehler der Netze auf. Aus den Ergebnissen schließen sie, dass Trainingsalgorithmen nicht-intuitive Eigenschaften und „blinde Flecken“ aufweisen, die bei Adversarial Examples ausgenutzt werden, um fehlerhafte Predictions zu forcieren.

Weiterhin stellen sie fest, dass die Verwendung von Adversarial Examples als zusätzliche Trainingsdaten eine Art der Augmentierung darstellt. Sie wirkt regularisierend, erhöht die Fähigkeit der Generalisierung des Modells und reduziert gleichzeitig die Anfälligkeit gegenüber Adversarial Examples. Ein Training mit Adversarial Examples nennen sie „Adversarial Training“. Es gibt verschiedene Möglichkeiten, derartige synthetisch manipulierte Bilder zu erstellen. Goodfellow et al. [18] beschäftigen sich ebenfalls mit dem Thema der Erstellung von Adversarial Examples und stellen eine Methode vor, die im Folgenden kurz erläutert

werden soll: In ihrer Arbeit behandeln sie das Problem, dass das Generieren von Adversarial Examples durch den erforderlichen Trainingsprozess und die Nicht-Linearität der Aktivierungsfunktionen sehr aufwendig ist. Aus diesem Grund schlagen sie mit der Fast Gradient Sign Method (*FSGM*) eine Möglichkeit vor, schnell und effizient Adversarial Examples zu generieren.

$$\eta = \varepsilon \times sign(\bigtriangledown_x L(\theta, x, y)) \tag{27}$$

Hierzu werden die Gradienten der Loss Funktion L in Bezug auf den Input x berechnet, um den Loss zu maximieren. θ sind die Parameter des Modells, y die Labels. Durch die Gradienten kann ermittelt werden, welchen Einfluss die verschiedenen Pixel des Inputs auf den Loss haben. Auf diese Gradienten wird die Signumfunktion angewendet, um zu ermitteln, ob sie jeweils positiv, negativ oder 0 sind. Die entstehenden Werte werden mit einem kleinen Wert ε multipliziert, sodass sie nur noch ein kleines, für das menschliche Auge unsichtbares Rauschen darstellen, bevor sie auf das Input-Bild addiert werden. Abbildung 30 visualisiert die Erstellung eines Adversarial Examples mit der Fast Gradient Sign Method. Die Autoren von *YOLO*v4 [2] empfehlen auch die Verwendung von Adversarial Examples für das Training, sie erwähnen nicht, welche Methode sie verwenden.

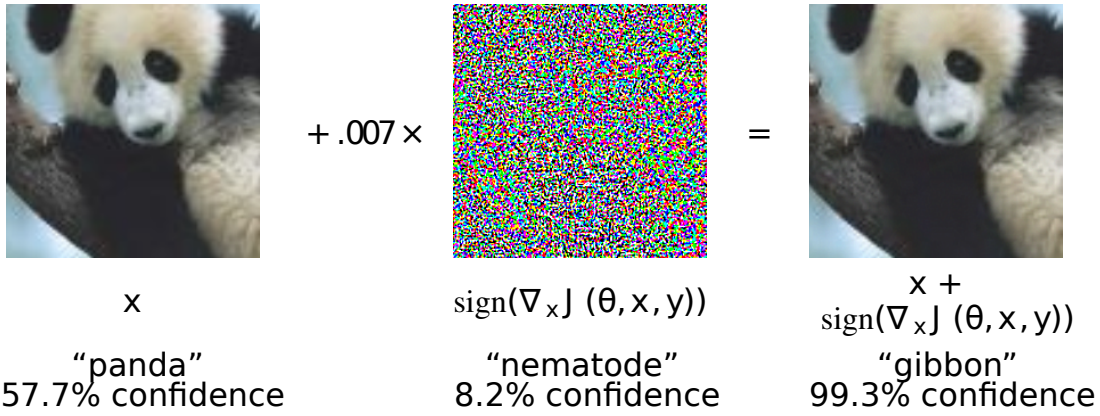


Abbildung 30: Beispiel für die Erstellung eines Adversarial Examples mit der Fast Gradient Sign Method aus [18]

5.1.7 *CmBN*

Mit der Cross mini-Batch Normalization schlagen die Autoren von *YOLO*v4 [2] eine Alternative zur in Kapitel 3.1 erläuterten Batch Normalization vor. Noch stärker als in der Mosaic Augmentierung wird in *CmBN* auf das Training mit einer einzigen gebräuchlichen *GPU*, zum Beispiel einer NVIDIA GTX 1080Ti oder RTX 2080, abgezielt. Dieses Ziel soll durch die Verwendung von kleinen Batch-Sizes ermöglicht werden.

CmBN basiert auf der Cross Iteration Batch Normalization (*CBN*), die von Yao et al. [51] vorgestellt wird. In ihrer Arbeit fassen sie zusammen, dass in der Batch Normalization davon ausgegangen wird, dass die Verteilungsstatistik der Eingangswerte einer Mini-Batch die Verteilung des gesamten Trainingsdatensatzes wiederspiegelt. Sie betonen die Wirksamkeit dieses Mechanismus bei großen Mini-Batch-Sizes, jedoch stützen sie sich auf Peng et al. [32], Wu & He [50] sowie Ioffe et al. [22], indem sie folgendes Problem beschreiben: Bei zu kleinen Mini-Batch-Sizes bleiben die Probleme, aufgrund derer Batch-Normalization eingeführt wurde, weiterhin bestehen. Wenn zu wenige Werte für die Normalisierung verwendet werden, spiegelt deren Verteilungsstatistik nicht die des gesamten Trainingsdatensatzes wieder. Die Qualität des Objekterkennungsmodells leidet dementsprechend stark unter kleinen Mini-Batch-Sizes.

In *CBN* wird in einem Trainingsschritt nicht nur der Input (= aktuelle Mini-Batch) der aktuellen Iteration t zur Berechnung von Mittelwert und Varianz verwendet, sondern auch die Inputs des betrachteten Layers aus vorherigen Iterationen $t-r$. Dadurch soll trotz geringer Mini-Batch-Sizes eine sinnvolle Normalisierung ermöglicht werden. Die Autoren beschreiben zunächst einen naiven Ansatz, in dem die Werte aus einer vorherigen Iteration $t-r$ gemeinsam mit denen der aktuellen Iteration t verwendet werden, um Mittelwert und Varianz zu berechnen. Diese Idee ist in Abbildung 31 visualisiert. Mittelwert und Varianz für eine Iteration $t-r$ sind jedoch mit den Gewichten θ_{t-r} aus dieser Iteration entstanden. Die Gewichte ändern sich durch das Training in jeder Iteration, daher macht die Verwendung des Mittelwerts $\mu_{t-r}(\theta_{t-r})$ und der Varianz $v_{t-r}(\theta_{t-r})$, die unter den Gewichten θ_{t-r} berechnet wurden, für die Verwendung in Iteration t keinen Sinn.

Während des Trainings ändern sich die Gewichte schrittweise. Aus diesem Grund verwenden die Autoren das Taylor-Polynom und approximieren den Mittelwert $\mu_{t-r}(\theta_t)$ sowie die Varianz $v_{t-r}(\theta_t)$. Hierzu verwenden sie die aktuellen Gewichte und die bereits aus den vorherigen Iterationen bekannten $\mu^l_{t-r}(\theta_{t-r})$ und $v^l_{t-r}(\theta_{t-r})$ eines Layers l. Die Berechnung des Taylor-Polynoms ist in [51] nachzuvollziehen. Mit den Mittelwerten und Varianzen von k

vorhergehenden Iterationen ermitteln sie schließlich eine durchschnittliche Varianz sowie einen Mittelwert. Diese werden für die Normalisierung der Mini-Batch der aktuellen Iteration t verwendet. Der Vorgang wird in Abbildung 32 visualisiert.

In *YOLO*v4 [2] fügen die Autoren eine Modifikation zu *CBN* hinzu: $\mu^l_{t-r}(\theta_{t-r})$ und $v^l_{t-r}(\theta_{t-r})$ werden für die letzten r Iterationen von vier Mini-Batches berechnet. Sie nennen ihre Modifikation *CmBN*. Im Gegensatz dazu werden in *CBN* nur die letzten Iterationen der aktuell betrachteten Mini-Batch verwendet.

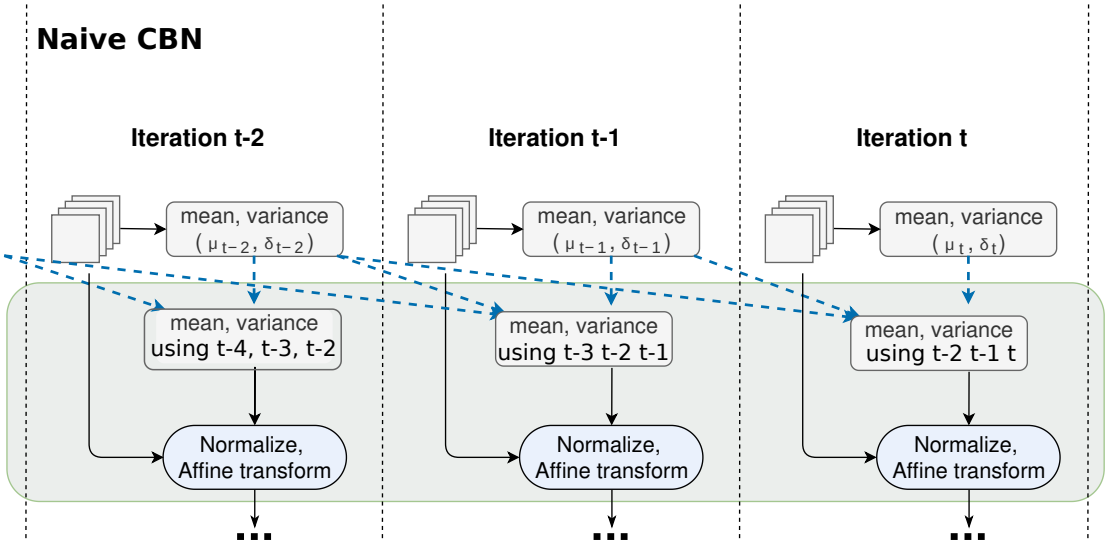


Abbildung 31: Naive Cross Iteration Batch Normalization (Abbildung aus [51] modifiziert für Naive *CBN*)

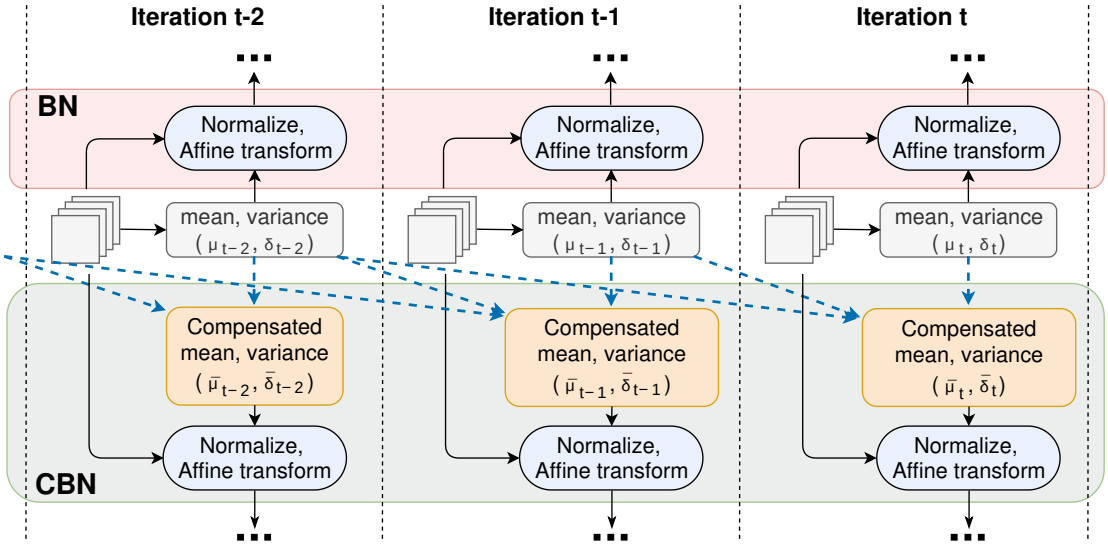


Abbildung 32: Cross Iteration Batch Normalization im Vergleich zu Batch Normalization aus [51]

5.1.8 Grid Sensitivity

*YOLO*v3 verwendet zur Berechnung der Bounding Box Koordinaten die in Gleichung 16 beschriebene Formel $b_x = \sigma(t_x) + c_x; b_y = \sigma(t_y) + c_y$. Die Werte c_x und c_y sind die Koordinaten der Rasterzellen (z.B. Zelle 1/1), also ganze Zahlen. Die Autoren stellen fest, dass für Objekte, deren Koordinaten (b_x, b_y) sehr nah an den c Werten, also sehr nah am rechten oder unteren Rand einer Rasterzelle sind, schwieriger erkannt werden. Das wird damit begründet, dass die Sigmoid-Funktion $\sigma(t_x)$ verwendet wird. Um z.B. $\sigma(t_x) \approx 1$, also den rechten Rand zu erreichen, wird ein sehr hoher t_x Wert als Prediction benötigt (analog dazu für t_y). Diese hohen t Werte werden schwer erreicht, $\sigma(t_x)$ oder $\sigma(t_y)$ werden nie den Wert 1 erreichen. Das führt dazu, dass derartige Objekte oft nicht ermittelt werden können. Dieses Problem soll gelöst werden, indem das Ergebnis der Sigmoid-Funktion mit einem Faktor größer 1.0 multipliziert wird, um damit Werte nahe der Grenzen der Rasterzellen mit kleineren t Werten zu generieren. Sie nennen diese Veränderung „Elimination of Grid Sensitivity“.

5.1.9 Multiple Anchors

Für ein Ground-Truth Objekt der Trainingsdaten wird in vorherigen Versionen immer ein Anchor ausgewählt, der für die Erkennung des Objekts zuständig ist. Die Anchor Box, die die höchste *IoU* mit der Ground-Truth Bounding Box aufweist, ist für die Erkennung dieser zuständig. In *YOLO*v4 experimentieren die Autoren mit dieser Definition, wodurch sie einen Vorteil darin erkennen, alle Anchor mit einer *IoU* über einem definierten *IoU*-Threshold als zuständig für die Erkennung zu definieren.

5.1.10 Cosine Annealing Learning Rate und Warm-Restarts

Beim Training vieler neuronaler Netze wird die Learning-Rate während des Trainings mit einer Verfallsfunktion, z.B. exponentiell reduziert. Für die Reduzierung der Learning-Rate gibt es verschiedene Ansätze. Loshchilov und Hutter [28] beschreiben eine Möglichkeit, in der die Learning Rate in mehreren sogenannten Restarts mit einem Wert initialisiert wird und anschließend ähnlich zur Kosinusfunktion in T_1 Schritten sinkt. Nach T_1 Schritten wird die Learning Rate erneut initialisiert (Restart) und in T_2 Schritten verringert. Für einen Restart i mit T_i Schritten wird die Learning-Rate η_t im Schritt t folgendermaßen definiert:

$$\eta_t = \eta^i_{min} + \frac{1}{2}(\eta^i_{max} - \eta^i_{min})(1 + cos(\frac{T_{cur}}{T_i}\pi)) \tag{28}$$

mit

$$\begin{aligned} t &= \text{Aktueller Schritt (über alle Restarts)} \\ \eta^i_{min} &= \text{minimale Learning Rate für aktuellen Restart} \\ \eta^i_{max} &= \text{maximale Learning Rate für aktuellen Restart} \\ T_i &= \text{Anzahl Schritte im aktuellen Restart} \\ T_{cur} &= \text{Aktueller Schritt im aktuellen Restart} \end{aligned}$$

Es wird empfohlen, mit einer geringen Anzahl an Schritten T_i pro Restart zu beginnen, und diese Anzahl in jedem Restart um einen Faktor T_{mult} zu erhöhen. Sie beschreiben auch, dass eine andere Möglichkeit darin bestünde, η^i_{max} und η^i_{min} in jedem Restart zu reduzieren. Die erste genannte Empfehlung wird in Abbildung 33 visualisiert.

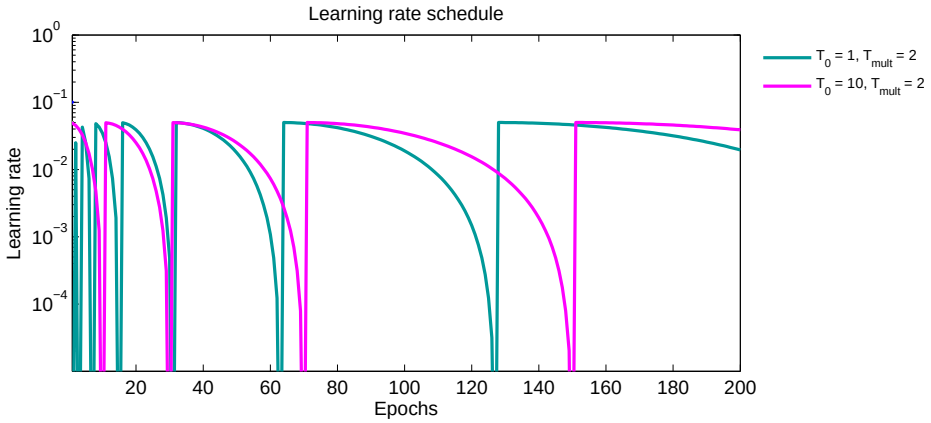


Abbildung 33: Cosine-Learning-Rate Scheduler mit unterschiedlichen T_0 Werten, angepasst aus [28]

5.1.11 Random Training Shapes

Bereits in *YOLO*v2 wurde empfohlen, verschiedene zufällige Input-Auflösungen im Training zu verwenden. In *YOLO*v4 wird diese Methode erneut vorgeschlagen. Die Begründung ist in Kapitel 3.6 beschrieben.

5.1.12 Optimale Hyperparametrisierung

Für das Training eines *YOLO*-Modells werden eine Vielzahl an Hyperparametern, wie die Learning-Rate, Anzahl der Layer, Kernel-Größen, Batch-Size und viele mehr benötigt. Breuel [4] beschreibt, dass es keinen „besten“ Lernalgorithmus gibt und eine Vorgehensweise, die für einen Anwendungsfall gut funktioniert, in einem anderen Anwendungsfall versagen kann. Durch verschiedene Wettbewerbe, wie zum Beispiel VOC [8], können unterschiedliche Objekterkennungssysteme und Parameterausprägungen mit der gleichen Basis für allgemeine Fälle verglichen werden. Daraus kann man vermuten, welche Technik wohl welchen Einfluss auf die Performance eines Modells haben könnte. Insbesondere Techniken, die in einfachen Architekturen große Fortschritte in der Performance liefern, sollen, wenn sie mit anderen vielversprechenden Techniken kombiniert werden, oft gemeinsam nicht noch besser funktionieren. Young et al. [52] stellen fest: Es gibt keine Formel für die optimalen Hyperparameter, sie können oft nur durch Erfahrung sowie das Probieren von Kombinationen ermittelt werden.

In *YOLO*v4 optimieren die Autoren die Parameter, indem sie eine Parametersuche mit genetischen Algorithmen durchführen. Hierzu gibt es mehrere Möglichkeiten - die Autoren beschreiben in ihrer Arbeit nicht, welchen genetischen Algorithmus sie verwendet haben. Ein allgemeines Beispiel für einen genetischen Algorithmus wäre der Ameisenalgorithmus [19].

Fiszelew et al. [11] beschreiben eine Möglichkeit, bei der verschiedene neuronale Netze generiert werden und mithilfe eines genetischen Algorithmus versucht wird, gute Ausprägungen der Hyperparameter zu generieren. Ihre Idee kann folgendermaßen knapp zusammengefasst werden: Es werden zunächst Netze mit zufälligen Hyperparametern erstellt und trainiert (sog. Population). In einem weiteren Schritt werden aus den entstandenen Modellen die besten gewählt („Selektion“) und die Eigenschaften der „Mutter“ und des „Vaters“ kombiniert, wodurch zwei „Kinder“ generiert werden („Rekombination“). Die Eigenschaften, die die Kinder erben, werden zufällig verändert („Mutation“) und die neuen Netze werden erneut trainiert, bevor die Kinder andere Netze der Population ersetzen. Der genannte Vorgang wird mehrfach wiederholt, sodass durch mehrfache Selektion, Rekombination und Mutation gute Hyperparameter generiert werden.

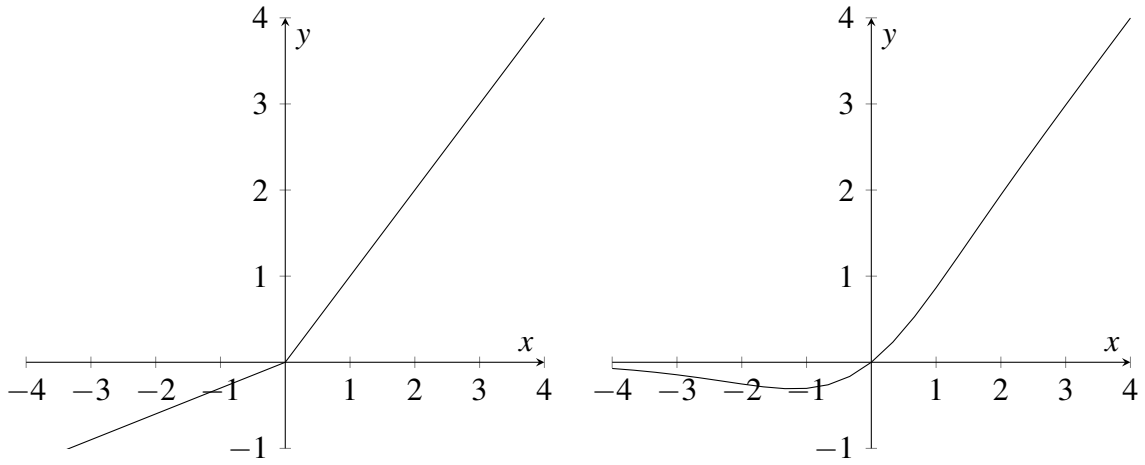
5.2 Bag of Specials

Im Bag of Specials beschreiben die Autoren von *YOLO*v4 Methoden, die im Gegensatz zum Bag of Freebies die Inferenzzeit geringfügig erhöhen, jedoch das Ergebnis erheblich verbessern sollen.

5.2.1 Mish-Aktivierungsfunktion

Bochkovskiy et al. schlagen für *YOLO*v4 Mish als Aktivierungsfunktion statt *Leaky ReLU* vor. In ihren Experimenten erzielten sie mit dieser bessere Ergebnisse im Vergleich zu *Leaky ReLU* und anderen Funktionen. Abbildung 34 zeigt den Verlauf von LeakyReLU und Mish im Vergleich. Mish [31] ist folgendermaßen definiert:

$$f(x) = x\tanh(softplus(x)) = x\tanh(\ln(1-e^x)) \tag{29}$$



(a) LeakyReLU Akivierungsfunktion $max(0.3 * x, x)$

(b) Mish Akivierungsfunktion $x * tanh(ln(1+e^x))$

Abbildung 34: LeakyReLU und Mish Aktivierungsfunktionen im Vergleich

Ein Vorteil beider Aktivierungsfunktionen besteht darin, dass sie nach oben unbegrenzt sind. Dies bewirkt, dass auch die Gradienten keine Obergrenzen haben und die Gewichte, wenn nötig, schnell stark verändern können. Damit wird das Training im Vergleich zur Nutzung von nach oben begrenzten Aktivierungsfunktionen wie z.B. Sigmoid beschleunigt [17].

Auch bieten beide Funktionen den Vorteil, dass im Vergleich zur Rectified Linear Unit (*ReLU*) Funktion ($f(x) = max(0,x)$) auch negative y-Werte generiert werden. Das verhindert das sogenannte Dying *ReLU* Problem. Dieses entsteht, wenn ein Neuron einen negativen Wert

generiert und dieser durch die *ReLU* Aktivierungsfunktion 0 wird. Die Gradienten für die Aktualisierung der Gewichte in der Optimierung nehmen in diesem Fall auch den Wert 0 an, sodass die Gewichte des Neurons nicht verändert werden. Wenn die Gewichte nicht mehr verändert werden trägt das Neuron nicht mehr zum Trainingserfolg bei - verlässt es diesen Status nicht mehr, ist es „tot“ [31].

Ein entscheidender Vorteil der Mish-Aktivierungsfunktion gegenüber *Leaky ReLU* ist ihre Begrenzung nach unten. Diese Eigenschaft hat laut Misra eine stark regularisierende Wirkung, was Overfitting reduziert. Die Regularisierung wird damit begründet, dass bei Mish die Komplexität des Modells durch die Begrenzung des Wertebereichs verringert wird.
Ein weiterer Unterschied zu *Leaky ReLU* liegt darin, dass Mish überall differenzierbar ist. Die Differenzierbarkeit verhindert Singularitäten und somit unerwünschte Effekte bei der Optimierung des Modells. Wenn die *Leaky ReLU* Funktion einen Wert nahe dem nicht-differenzierbaren Wert 0 generiert, kann die Anpassung der Gewichte dazu führen, dass der Wert im nächsten Schritt abrupt z.B. vom negativen in den positiven Bereich „springt“ (analog von positiv zu negativ), was ein unvorhersehbares Verhalten hervorrufen kann.

5.2.2 *CSPs*

In *YOLO*v4 wird das Backbone erneut verbessert. Das in Kapitel 4.4 beschriebene Darknet-53 wird mit *CSPs* ausgestattet, die die Qualität der Ergebnisse des Objekterkennungsprozesses weiter verbessern sollen. *CSPs* basieren auf der Arbeit von Wang et al. [48], die ein Problem in tiefen neuronalen Netzen sehen: Zwar liefern diese Ergebnisse hoher Qualität, jedoch steigt mit ihrer Tiefe auch die Inferenzzeit. Mit *CSP*Net finden sie eine Netzwerkarchitektur, die mit *CSPs* trotz ihrer Tiefe vergleichsweise kurze Inferenzzeiten bieten kann. Damit können zum Beispiel Objekterkennungsnetze für *YOLO* tiefer gestaltet werden, ohne große Einbußen in der Inferenzzeit zu riskieren.

Um das Ziel zu erreichen, teilen die Autoren die Feature-Maps in einer Schicht zunächst in zwei Teile auf, bevor diese verschiedene Schichten durchlaufen und letztlich wieder zusammengeführt werden. Damit besteht ihr Konzept darin, den Gradientenfluss aufzuteilen und über verschiedene Pfade im neuronalen Netz zu verarbeiten. Abbildung 35 beschreibt den Aufbau eines *CSP*-Residual-Blocks.

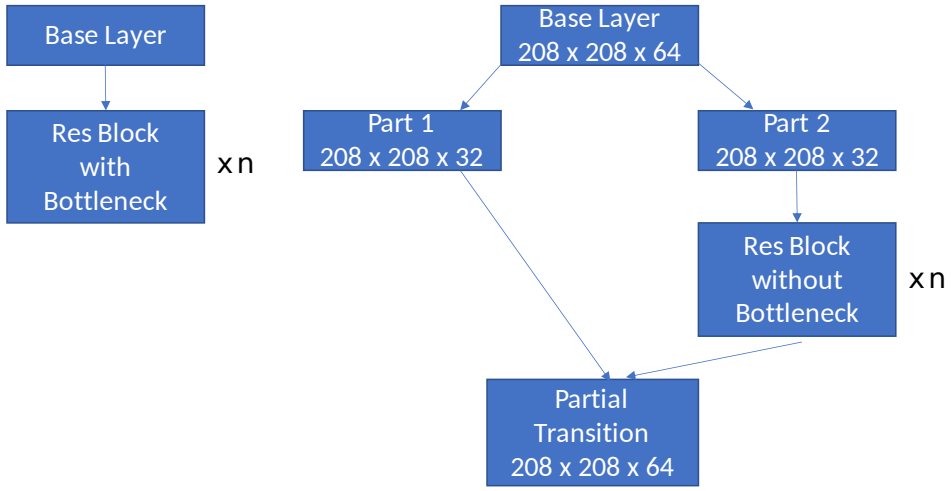


Abbildung 35: links: Residual-Block, rechts: *CSP*-Residual-Block ohne Pooling (angepasst, Idee aus [48])

Zunächst werden die Channels der Input Feature-Maps in zwei Teile aufgeteilt. Ein Teil wird an einen Residual-Block übergeben, während der andere nicht verändert wird. Da nur die Hälfte der Channels den Residual-Block passiert, soll keine Bottleneck Architektur (Abbildung 21) nötig sein. Die beiden entstandenen Teile werden schlussendlich per Konkatenation der Channels wieder zusammengesetzt. Es folgt ein sogenannter Transition-Layer. Dieser wird von He et al. [20] für Residual Connections beschrieben und besteht aus einem Convolutional Layer und einem Pooling Layer. Wang et al. [48] betonen, dass die genannte Architektur die Lernkapazität eines Netzes erhöht. Gleichzeitig wird der Rechenaufwand des Netzes im Vergleich zur reinen Nutzung von Residual-Blöcken reduziert. Zudem sollen Rechenengpässe (Computational Bottlenecks) vermieden werden, da die Berechnungen besser auf Recheneinheiten verteilt und damit die Nutzungsrate dieser erhöht werden kann. Durch das leichtgewichtigere Modell soll auch die Speichernutzung stark reduziert werden.

Zwar wird die Idee der *CSPs* von Bochkovskiy et al. [2] übernommen, jedoch verändern sie diese für die Nutzung *YOLO*v4. Abbildung 36 zeigt, wie die *CSPs* in *YOLO*v4 umgesetzt sind. Hier wird in einem Pfad der gesamte Input verwendet und eine 1×1 Convolution angewendet. In einem weiteren Pfad wird erst eine 1×1 Convolution, anschließend ein Residual-Block sowie eine weitere 1×1 Convolution angewendet. Die Ergebnisse der beiden Pfade werden konkateniert, was die Anzahl der Channels verdoppelt. Aus diesem Grund wird die Anzahl der Channels mithilfe einer weiteren 1×1 Convolution halbiert. Während Wang et al. von leichtgewichtigeren, effizienteren Modellen sprechen, werden *CSPs* in *YOLO*v4 im Bag of Specials angesiedelt. Damit ist davon auszugehen, dass die Inferenzzeit durch *CSPs* in dieser Form leicht erhöht, das Ergebnis jedoch weiterhin verbessert wird.

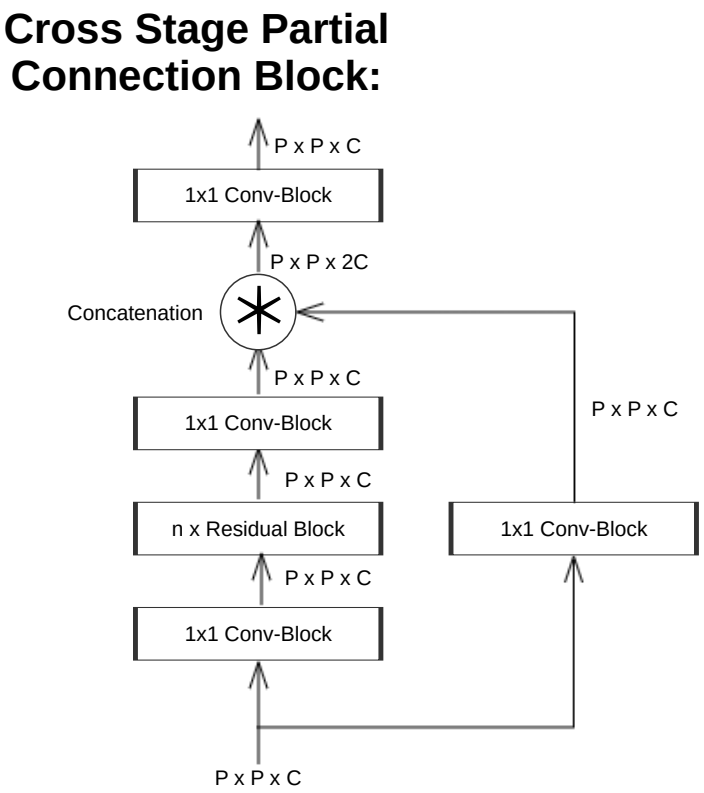


Abbildung 36: Cross Stage Partial Connection-Block in YOLOv4

5.2.3 *PANet*

In *YOLO*v3 werden, wie in Kapitel 4.3 beschrieben, Feature Pyramids für die Erstellung von Predictions auf Feature-Maps unterschiedlicher Auflösungen verwendet. Liu et al. [26] stellen darüber hinaus *PANets* vor, die in *YOLO*v4 verwendet werden. Abbildung 37 beschreibt den Aufbau von *PANet*.

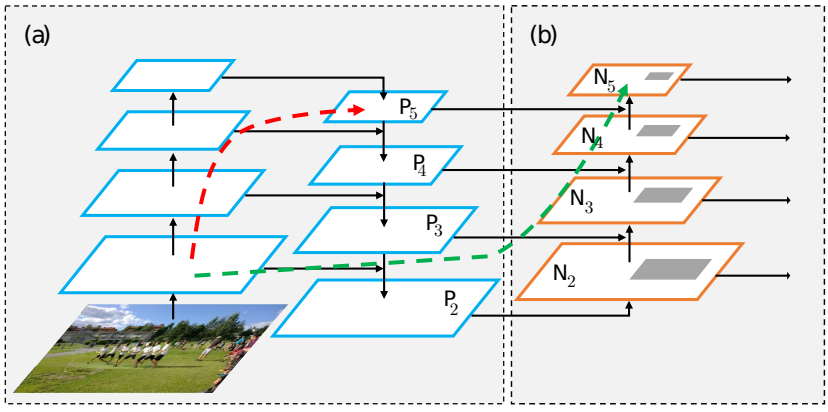


Abbildung 37: Path Aggregation Network (angepasst, Idee aus [26]); a): *FPN*, b): Bottom-Up-Path für *PANet*

In *PANet* wird die Idee des Top-Down-Paths der *FPNs* aufgegriffen und um einen Bottom-Up-Path erweitert. Dabei greifen die Autoren auf die Erkenntnisse von Zeiler und Fergus [54] zurück, die beschreiben, dass Neuronen in späteren Schichten (High-Levels) auf ganze Objekte reagieren, während sie in frühen Schichten (Low-Levels) von Mustern und Strukturen aktiviert werden. Liu et al. erkennen die Relevanz von starken Aktivierungen früher Schichten: Diese assoziieren das Vorhandensein von z.B. Kanten. Sie können ein Indiz für die Anwesenheit eines Objekts sein. Aus diesem Grund möchten sie die Informationen von Low-Levels auch stärker in High-Levels berücksichtigen. Hierzu schaffen sie Shortcut-Connections von Low-Levels in die High-Levels (grüne Verbindung auf Abbildung 37). Im Vergleich dazu beschreiben sie, wie Low-Level Informationen bei der reinen Nutzung von *FPNs* ohne Shortcut einen sehr langen Pfad zurücklegen müssten (rote Verbindung auf Abbildung 37), um in den High-Levels berücksichtigt zu werden. Daraus kann gefolgert werden, dass in *FPNs* die Low-Levels zwar mehr Informationen der Higher-Levels nutzen (durch den Top-Down-Path), dies jedoch in entgegengesetzter Richtung nicht stattfindet.

Für die Erstellung des Bottom-Up Paths soll beispielhaft die Generierung von N_3 aus Abbildung 37 beschrieben werden. Die Auflösung der Feature-Map P_2 wird, gegenteilig zu *FPN*, mit einem Downsampling Layer (3x3 Convolution mit Stride 2) verringert und ggf. die Anzahl der Channels durch eine 1×1 Convolution erhöht. Anschließend werden die Werte elementweise auf die Feature-Map P_3 addiert. Es folgt eine weitere 3x3 Convolution. Abbildung 38 beschreibt den Vorgang allgemein. N_2 in Abbildung 37 gleicht P_2, da kein Layer darunter ist.

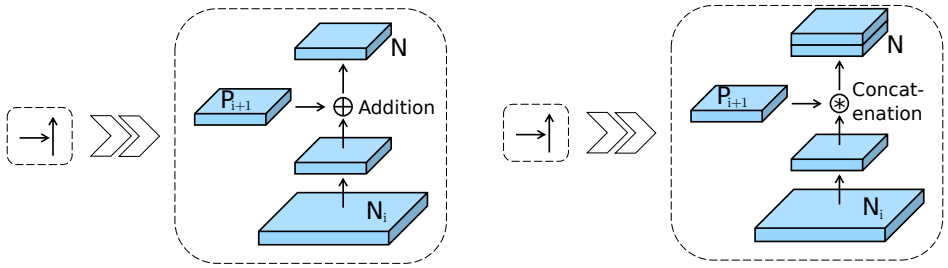


Abbildung 38: Erstellung von PANet; links PANet nach Lui et al. aus [26]; rechts aus *YOLO*v4 ([2]); beide angepasst

In *YOLO*v4 [2] verändern Bochkovskiy et al. die Vorgehensweise für die Erstellung von *PANets*, indem sie, wie ebenfalls in Abbildung 38 dargestellt, die Channels konkatenieren, statt die Werte elementweise zu addieren. Die entstandenen Feature-Maps N_i werden, wie auch in *FPN*, für den Object-Detection Head verwendet. Wie bereits in *YOLO*v3 werden in *YOLO*v4 Predictions auf drei Auflösungen erstellt.

5.2.4 *MiWRCs*

*YOLO*v4 [2] nutzt das Konzept der *MiWRCs*, das von Tan et al. in EfficientDet [47] angewendet wird. In Kapitel 4.4 wurden Residual Connections beschrieben. In diesen werden zwei Feature-Maps durch elementweise Addition kombiniert. Wenn z.B. im Residual-Block keine Identitätsrelation genähert wird, werden die Feature-Maps darin bei der Addition gleich behandelt, also nicht gewichtet. Jedoch stellt sich heraus, dass verschiedene Input-Features, wenn sie ohne weiteres verwendet werden, die Output-Features unterschiedlich stark beeinflussen. Dieses Ungleichgewicht wird behandelt, indem in *MiWRC* unterschiedliche Gewichte für die verschiedenen Features verwendet werden. Somit kann das Netzwerk auch in Residual Connections lernen, welche Features wichtiger sind und mehr Einfluss haben sollen. Dabei werden die Gewichte normalisiert, um Stabilität im Training zu erreichen. Hierzu kann zum Beispiel die Softmax Funktion verwendet werden, um die Gewichte zwischen 0 und 1 zu normalisieren. Die Autoren beschreiben jedoch, dass Softmax nicht sehr effizient ist. Aus diesem Grund verwenden die Autoren eine andere Berechnung der normalisierten Werte. In EfficientDet wird die Kombination der Feature-Maps und die Gewichtung als „Fast normalized Fusion“ beschrieben. Sie wird nach folgender Formel berechnet:

$$O = \sum_i \frac{w_i}{\varepsilon + \sum_j w_j} \times I_i \tag{30}$$

mit

$$\begin{aligned} I &= \text{Feature-Maps, die kombiniert werden sollen} \\ I_i &= \text{eine Feature-Map} \\ w_i &= \text{trainierbare Gewichte für eine Feature-Map (Gewicht-Tensor)} \\ w_i &>= 0 \text{ (durch Anwendung von ReLU)} \\ w_j &= \text{ein Gewicht-Wert aus } w_i \\ \varepsilon &= 0.0001 \end{aligned}$$

Durch den Term $\frac{w_i}{\varepsilon+\sum_j w_j}$ werden die Werte eines Gewicht-Tensors zwischen 0 und 1 effizient normalisiert, ohne Softmax zu verwenden. In *YOLO*v4 wird diese Methode für jede Residual Connection verwendet.

5.2.5 *SPP*

Mit der Verwendung verschiedener Feature-Maps mit unterschiedlichen Auflösungen in *YOLO*v2, *YOLO*v3 und *YOLO*v4 wird bereits ermöglicht, dass unterschiedlich große Objekte zuverlässiger erkannt werden, als bei der reinen Nutzung der letzten Feature-Map. He et al. erforschen mit *SPP* [21] eine Möglichkeit, die Erkennung von unterschiedlich großen Objekten auch auf nur einer Feature-Map zu erleichtern. So sind zum Beispiel auch auf dem Ergebnis des letzten Convolutional Layers unterschiedlich große Objektmerkmale abgebildet, deren Erkennung auf unterschiedlichen Skalen bisher nicht berücksichtigt wurde. Der Aufbau eines *SPP* Layers ist in Abbildung 39 ersichtlich. Die Abbildung zeigt ein *CNN* mit Convolutional Layers, „conv5“ ist die letzte Convolution, auf die ein *SPP*-Layer und anschließend *FC Layers* folgen. Der Output der letzten Convolution durchläuft parallel drei verschiedene Pooling-Layer mit verschiedenen Kernel-Größen. Pro Input Feature-Map entstehen drei Feature Maps in unterschiedlichen Auflösungen, auf denen Objektmerkmale in verschiedenen Skalen abgebildet sind.

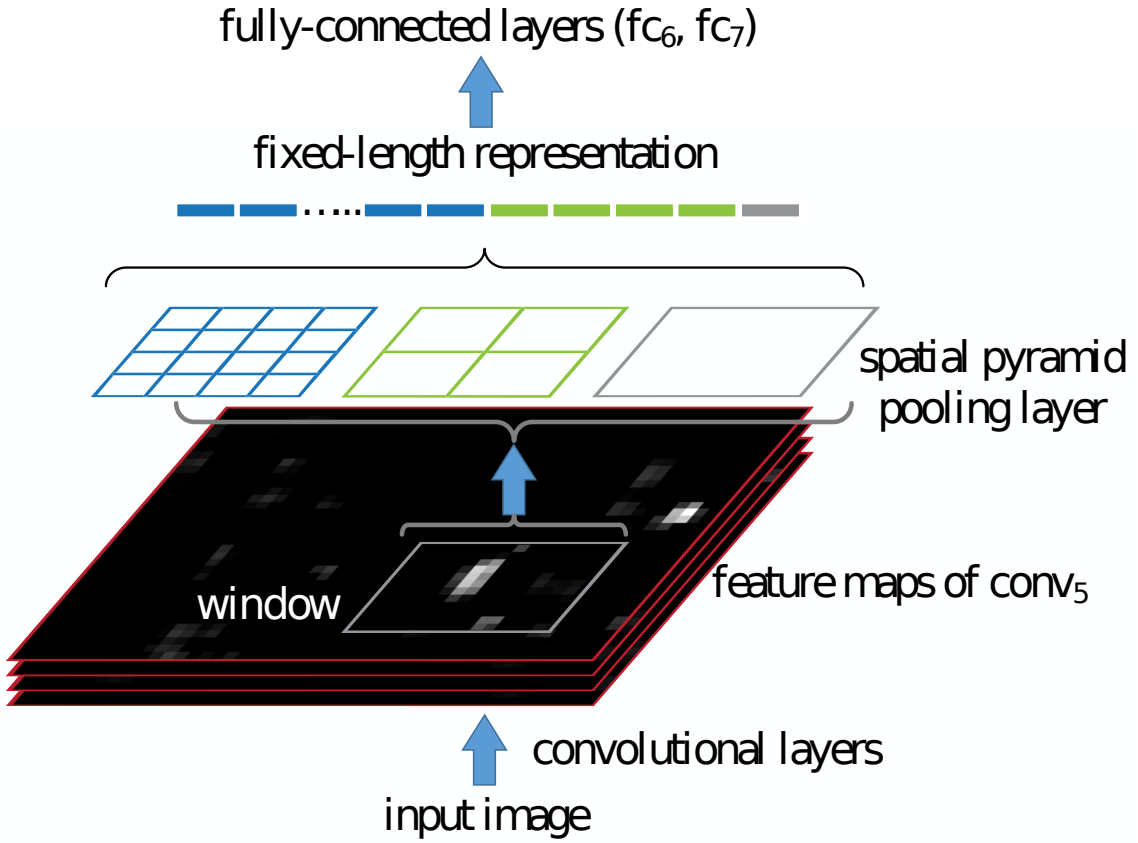


Abbildung 39: Spatial Pyramid Pooling Layer (aus [21])

In *CNNs* mit *FC Layers* werden die entstandenen Feature-Maps zusammengesetzt und in einen eindimensionalen Vektor konvertiert, der als Input in einen *FC Layer* dient. Dabei hat der *FC Layer* eine fixe Input-Größe. Damit verfolgen die Autoren von *SPP* neben der Erkennung auf unterschiedlichen Skalen auch das Ziel, in *CNNs* mit *FC Layers* Bilder in unterschiedlichen Input-Auflösungen als Inputs zu ermöglichen. Dieses Ziel wird erreicht, indem der *SPP*-Layer bei allen Input-Auflösungen Feature-Maps mit fester Größe produziert und das Ergebnis in einen *FC Layer* mit fixer Input-Größe übergeben werden kann.

Da *YOLO*v4 keine *FC Layers* verwendet, ist der letztgenannte Vorteil von *SPP* in diesem Fall irrelevant. Die drei entstehenden Feature-Maps werden in *YOLO*v4 nicht in einen eindimensionalen Vektor konvertiert. Stattdessen werden Inputs des *SPP* Layers sowie die drei enstandenen Feature-Maps konkateniert. *SPP* wird auf der letzten Feature-Map des Backbones angewendet, das Ergebnis dient als Input in den Objekt Detection Head.

5.2.6 *SAM*

In *YOLO*v4 wird erstmals auch ein Attention-Modul für die Objekterkennung verwendet. Bochkovskiy et al. verwenden für ihre neue Version das *SAM*, das Teil des Convolutional Block Attention Modules [49] von Woo et al. ist. Letztere beschreiben, dass *SAM* es ermöglicht, zu „betonen“, wo sich auf einer Feature-Map der informative Teil befindet. Damit wird beschrieben, welche Bereiche stärker bzw. weniger „betont“ und damit berücksichtigt werden sollen. Abbildung 40 a) beschreibt den Aufbau eines *SAM*.

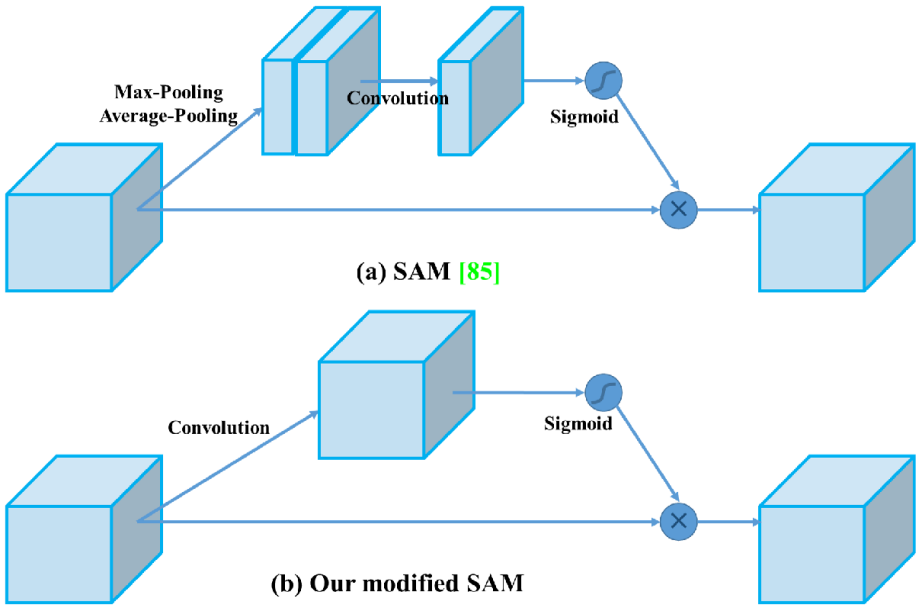


Abbildung 40: Spatial Attention Module; a): *SAM* aus [49], b) *SAM* aus [2]; Bild aus [2]

Zunächst wird sowohl Maximum-Pooling als auch ein Average-Pooling auf den Input angewendet. Das Pooling findet dabei auf die Channels bezogen statt, es entstehen zwei Feature-Maps mit den Dimensionen $1 \times W \times H$. Diese werden konkateniert und eine Convolution angewendet. Woo et al. verwenden hierbei Kernel der Größe 7×7. Zuletzt folgt die Sigmoid Aktivierungsfunktion, die den Wertebereich der Ergebnisse für die Gewichtung zwischen 0 und 1 begrenzt. Die entstandene Attention Feature-Map wird dann mit dem Input des *SAM* multipliziert, sodass der Output dieselben Dimensionen wie der Input aufweist.

In *YOLO*v4 modifizieren Bochkovskiy et al. den Aufbau von *SAM*, wie in Abbildung 40 b) zu sehen. Sie ersetzen die Pooling Layers und den Convolutional Layer mit einer einzigen 1×1 Convolution, auf die die Aktivierungsfunktion folgt. Damit wird die Aufmerksamkeit (Attention) nicht mehr räumlich über alle Channels hinweg bestimmt, sondern punktweise - also für jedes Pixel aller Channels einzeln.

5.2.7 *DIoU-NMS*

In Kapitel 2.4 wurde beschrieben, dass *NMS* verwendet wird, um in der Inferenz Boxen zu filtern, die dasselbe Objekt erkennen. Bochkovskiy et al. verwenden in *YOLO*v4 *DIoU-NMS*. Um dieses Verfahren zu betrachten, soll zunächst im Folgenden einleitend *NMS* und Soft-*NMS* erläutert werden. Abbildung 41 beschreibt den Pseudocode von *NMS*.

Wenn mehrere Boxen eine hohe *IoU* (über einem Threshold) haben, beziehen sie sich höchstwahrscheinlich auf dasselbe Objekt. In diesem Fall wird nur die Box berücksichtigt, die den höchsten Confidence Score aufweist, da die Prediction an der Position dieser Box am „sichersten“ ist. Der Vorgang wird wiederholt, bis alle stark überlappenden Boxen entfernt und damit die finale Prediction erstellt wurde [3].

Data: $B = b_1, ..., b_N, S = s_1, ..., s_N, N_t$, ($B$ is the list of initial detection boxes, S contains corresponding detection scores, N_t is the NMS Threshold)

$D \leftarrow \{\}$;

while $B \neq empty$ **do**

  $m \leftarrow$ argmax S ;

  $M \leftarrow b_m$;

  $D \leftarrow D \cup M$;

  $B \leftarrow B - M$;

  for $b_i \in B$ **do**

    if $iou(M, b_i) \geq N_t$ **then**

      $B \leftarrow B - b_i$;

      $S \leftarrow S - s_i$;

    end

    $s_i \leftarrow s_i f(iou(M, b_i))$

  end

end

return D, S

Abbildung 41: Pseudocode für Non-Maximum Suppression (angepasst aus [3]); Rote Zeilen (NMS) in Soft-NMS durch grüne Zeilen ersetzt

Ein Problem von *NMS* wird von Bodla et al. [3] beleuchtet: Wenn zwei Ground-Truth Objekte zufällig überlappen und die *IoU* größer als der *NMS* Threshold ist, wird eines der Objekte nicht erkannt. Das liegt daran, dass die Box, mit der niedrigeren Confidence durch *NMS* entfernt wird. Jedoch erkennt die Box das zweite Objekt eigentlich richtig - sie wird lediglich fälschlicherweise von *NMS* entfernt. Um dieses Problem zu beheben könnte der *NMS*-Threshold angehoben werden, jedoch würde dies zu einer höheren Anzahl an False-Positives führen, weil das Filterkriterium nicht streng genug wäre. Bodla et al. stellen mit Soft-*NMS* eine Möglichkeit vor, dieses Problem zu beheben. Zunächst wird eine andere Beschreibung für den roten Abschnitt in Abbildung 41 folgendermaßen definiert:

$$s_i = \begin{cases} s_i, & IoU(M, b_i) < N_t \\ 0, & IoU(M, b_i) \geq N_t \end{cases} \tag{31}$$

In Soft-*NMS* wird vorgeschlagen, die Boxen mit hoher *IoU* und gleichzeitig hoher Confidence nicht direkt zu entfernen, sondern die Confidences dieser Boxen proportional zur *IoU* zu reduzieren. Damit wird die Confidence der Boxen umso stärker reduziert, je mehr sie mit M, also der Box mit der höchsten Confidence, überlappen. Der oben genannte Problemfall wird damit behoben, da die überlappenden Boxen nicht direkt entfernt werden. Gleichung 31 wird mit folgender Definition ersetzt:

$$s_i = \begin{cases} s_i, & IoU(M, b_i) < N_t \\ s_i(1 - iou(M, b_i)), & IoU(M, b_i) \geq N_t \end{cases} \tag{32}$$

DIoU-NMS basiert auf derselben Idee wie Soft-*NMS*, jedoch wird hier das Problem anders gelöst: In *DIoU-NMS* wird statt der *IoU* die *DIoU* verwendet, um beim Prüfen von überlappenden Boxen auch die Distanz zwischen deren Zentren in die Entscheidung einzubeziehen. Die Idee basiert darauf, dass zwei Boxen vermutlich unterschiedliche Objekte erkennen, wenn ihre Zentren weit voneinander entfernt sind - auch wenn die *IoU* den Threshold übersteigt. Wenn dieser Abstand berücksichtigt wird, können Boxen mit hoher *DIoU* ohne Probleme entfernt werden; das Reduzieren von Confidence Scores aus Soft-*NMS* wird überflüssig. Zheng et al. ersetzen Gleichung 31 mit folgender Definition:

$$s_i = \begin{cases} s_i, & IoU(M, b_i) - R_{DIoU}(M, b_i) < N_t \\ 0, & IoU(M, b_i) - R_{DIoU}(M, b_i) \geq N_t \end{cases} \tag{33}$$

mit

$$R_{DIoU}(M, b_i) = \frac{\rho^2(M, b_i)}{c^2}$$

Herleitung und Berechnung der *DIoU* sind in Kapitel 5.1.5 und [56] nachzuvollziehen.

5.3 Fazit zu *YOLO*v4

Mit der vierten Version integrieren Bochkovskiy et al. zahlreiche neue Verfahren in *YOLO*. Damit gelingt es ihnen, das Objekterkennungssystem in ihrem Anwendungsfall, der Erkennung von Objekten auf dem *COCO* [6] Objekterkennungsdatensatz, stark zu verbessern. Neben der Beschreibung der verwendeten Techniken [2] wird unter einem Fork von Darknet[2] auch eine Implementierung von *YOLO*v4 veröffentlicht.

Abbildung 42 vergleicht die Performance von *YOLO*v4 mit anderen Objekterkennungssystemen auf dem *COCO*-Datensatz. Die *mAP* Metrik hängt bei Fully-Convolutional-Neural-Networks unter anderem mit der Auflösung der Input-Bilder zusammen. Durch geringere Auflösungen werden die Geschwindigkeit in der Inferenz erhöht (Frames per Second) und Einbußen in der *mAP* in Kauf genommen. Die Abbildung zeigt, dass die anderen betrachteten Objekterkennungssysteme nicht die Echtzeit-Geschwindigkeit von *YOLO* erreichen können. Im Gegenzug erreicht EfficientDet [47] eine höhere *mAP* bei geringerer Geschwindigkeit.

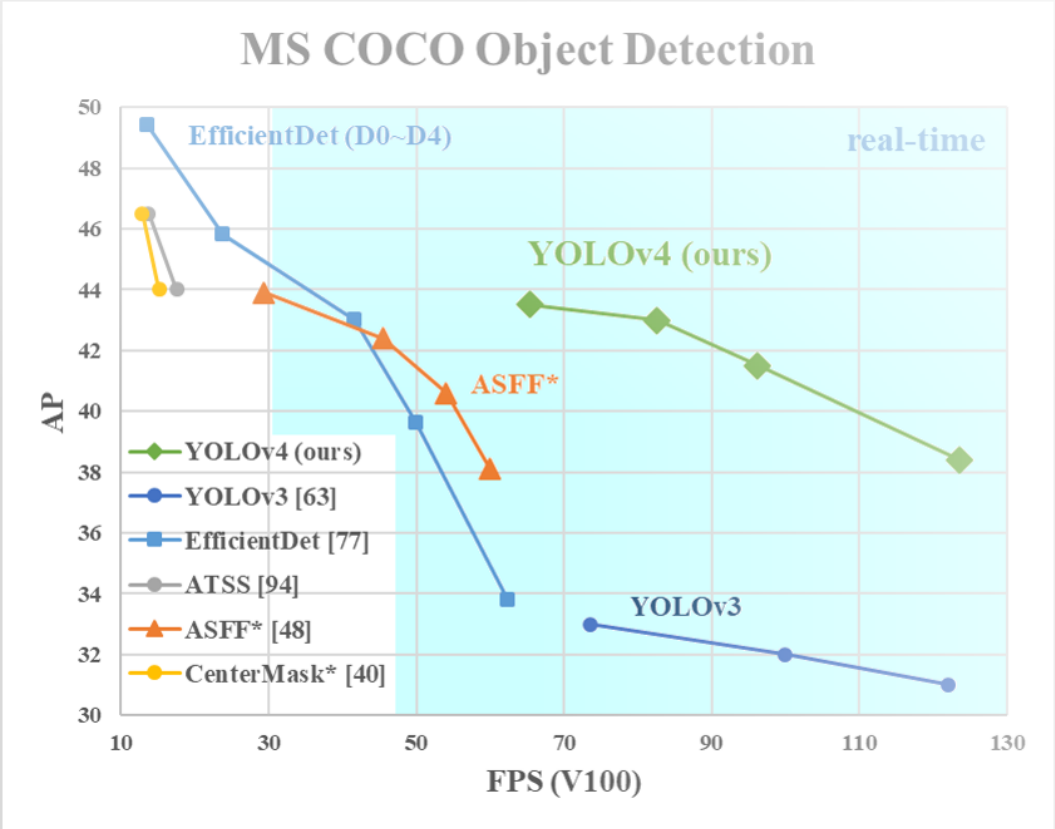


Abbildung 42: Vergleich des *mAP* von *YOLO*v4 mit *YOLO*v3 und anderen Objekterkennungssystemen in verschiedenen Frames per Second (*FPS*), Inferenz auf NVIDIA Tesla V100 *GPU*[2]

[2]https://github.com/AlexeyAB/darknet

In *YOLO*v4 werden zahlreiche Veränderungen an der Netzwerkarchitektur vorgenommen. Die verwendeten neuen Verfahren wurden in diesem Kapitel isoliert betrachtet und vorgestellt. Im Folgenden sollen sie in Zusammenhang gebracht und damit ein Überblick über die komplexe Netzwerkarchitektur von *YOLO*v4 geschaffen werden. Abbildung 43 zeigt einen Gesamtüberblick über *YOLO*v4 mit dem *CSP*-Darknet-53 Backbone. Abbildung 46, Abbildung 47 und Abbildung 48 beschreiben Blöcke, die in Abbildung 43 verwendet werden. Die *PANet* Architektur und alle darin verwendeten Blöcke (in Gesamtheit Neck genannt) wird in Abbildung 44 visualisiert. Abbildung 45 beschreibt den Object-Detection-Head und damit den letzten Teil des neuronalen Netzes für *YOLO*v4.

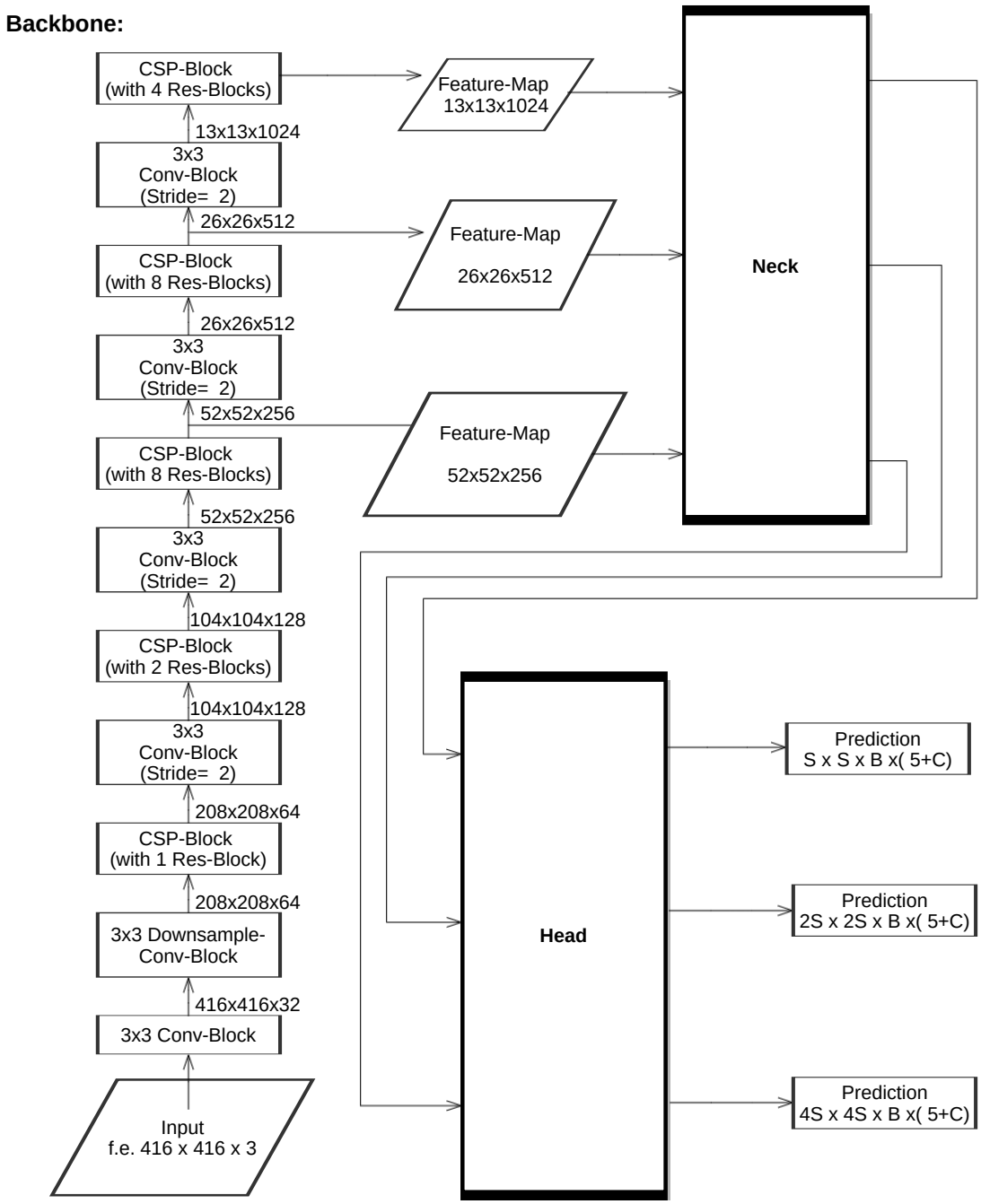


Abbildung 43: *YOLO*v4 Architektur

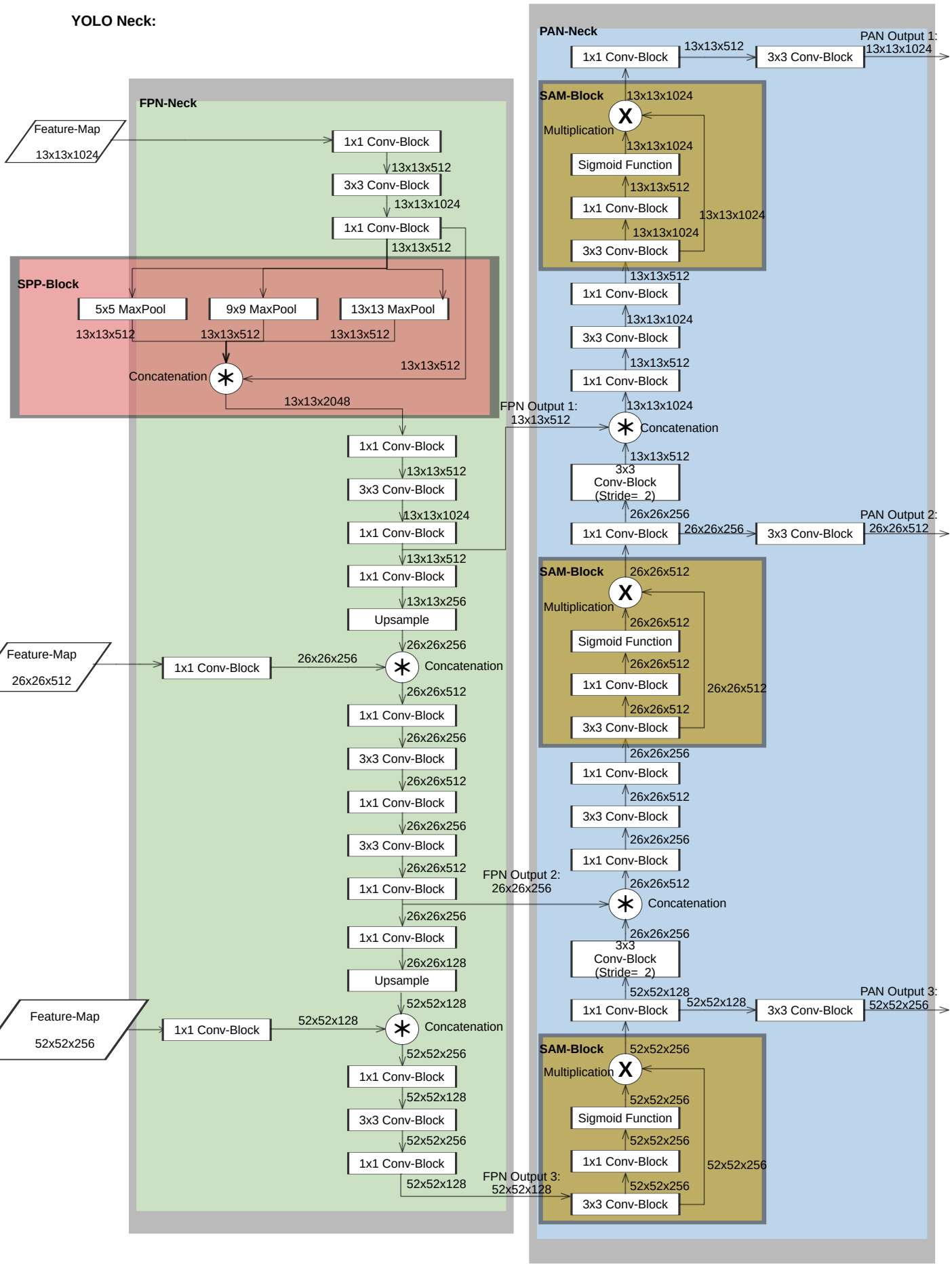


Abbildung 44: *YOLO*v4 Neck

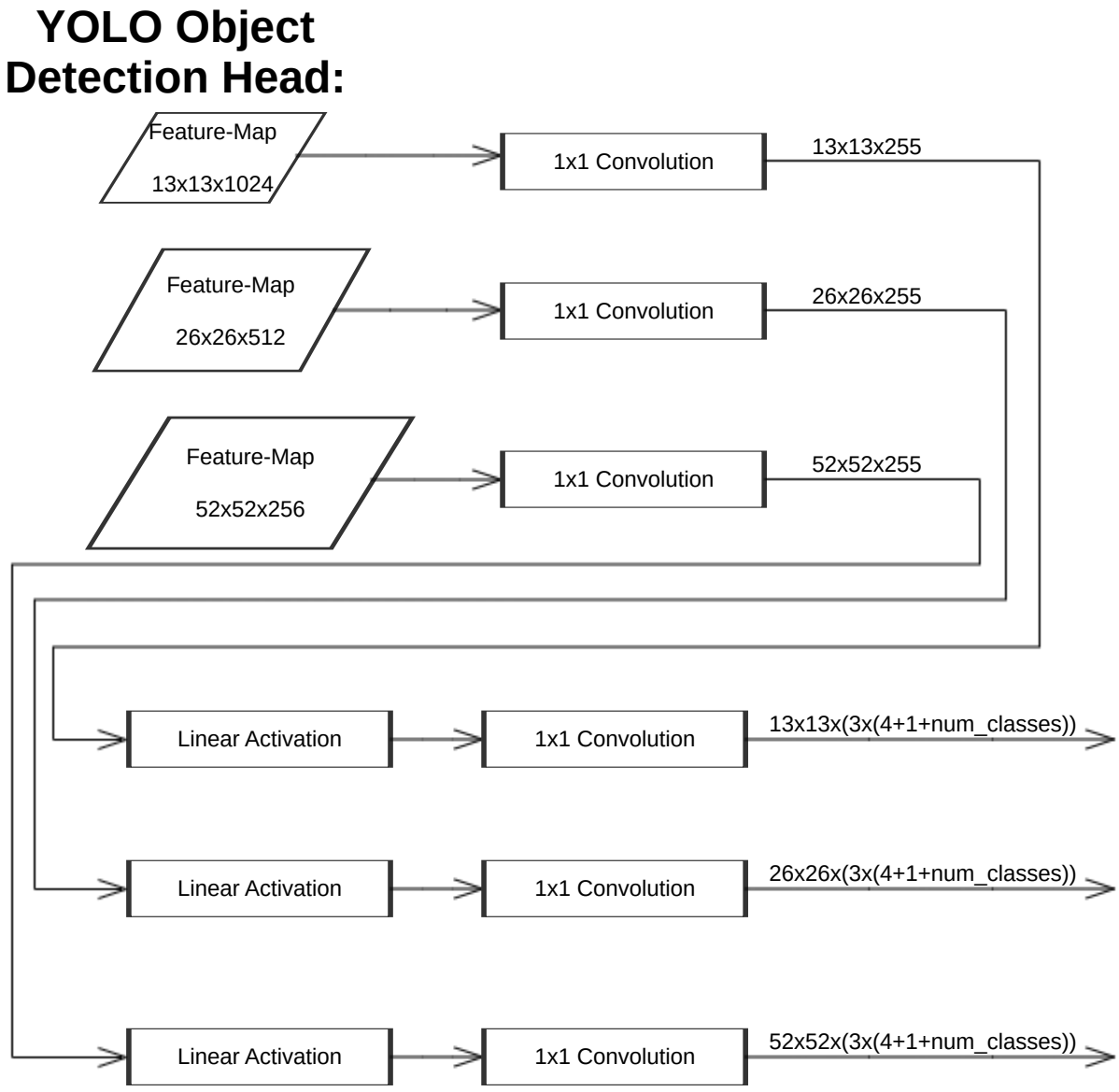


Abbildung 45: Object-Detection Head

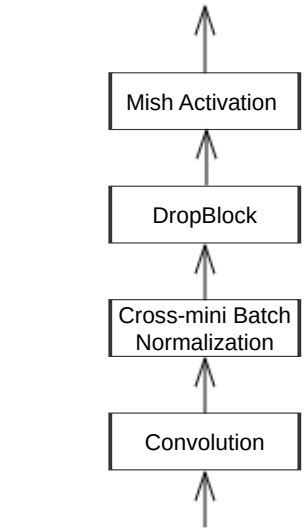


Abbildung 46: Convolution-Block

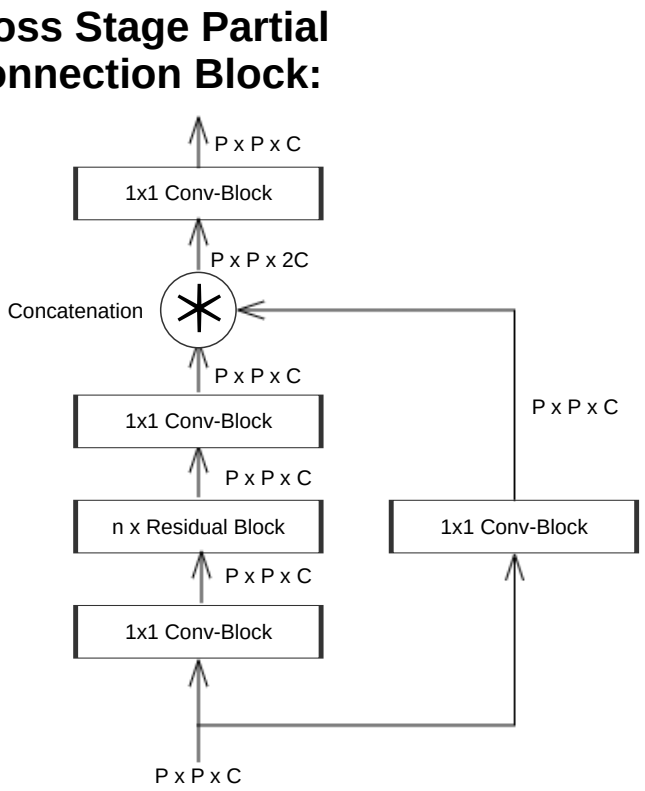


Abbildung 47: Cross Stage Partial Connection-Block

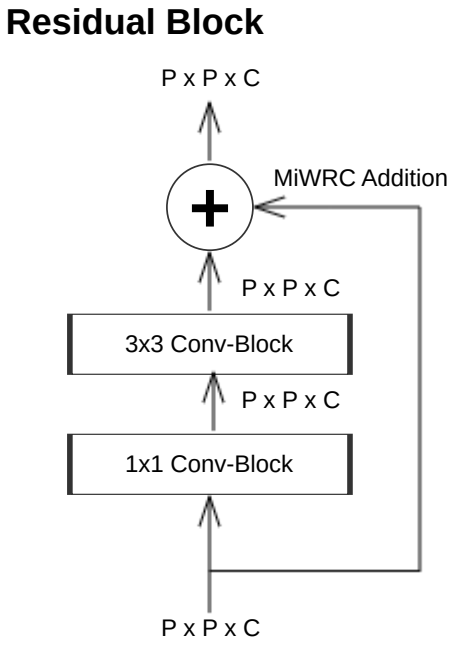


Abbildung 48: Residual-Block

Anhang A Abkürzungsverzeichnis

YOLO	You Only Look Once I
IoU	Intersection over Union II
TP	True Positive 5
FP	False Positive 5
FN	False Negative 5
TN	True Negative 5
AP	Average Precision 8
AR	Average Recall 8
mAP	Mean Average Precision 10
mAR	Mean Average Recall 10
RCNN	Region Based Convolutional Neural Network 11
CNN	Convolutional Neural Network 14
FC Layer	Fully Connected Layer 15
ReLU	Rectified Linear Unit 54
Leaky ReLU	Leaky Rectified Linear Unit 15
RSS	Residual sum of squares 16
FPN	Feature Pyramid Network 31
GPU	Graphics Processing Unit 38
CLS	Class Label Smoothing Regularization II
GIoU	Generalized *IoU* 44
DIoU	Distance *IoU* III
CIoU	Complete *IoU* II
SAT	Self Adversarial Training II
CmBN	Cross mini-Batch Normalization II
CBN	Cross Iteration Batch Normalization 49
CSP	Cross Stage Partial Connection II
PANet	Path Aggregation Network II
MiWRC	Multi Input Weighted Residual Connection III
SPP	Spatial Pyramid Pooling III
SAM	Spatial Attention Module III
NMS	Non-Maximum-Suppression III
FPS	Frames per Second 65
COCO	Common Objects in Context 10
FSGM	Fast Gradient Sign Method 48

Anhang B Literaturverzeichnis

[1] Benjamin Aunkofer. *ueberwachtes-maschinelles-lernen-prozess-modell.png (1185×617)*. 2017. URL: https://data-science-blog.com/blog/2017/07/02/uberwachtes-vs-unuberwachtes-maschinelles-lernen/ (besucht am 25. 01. 2021).

[2] Alexey Bochkovskiy, Chien-Yao Wang und Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. Techn. Ber. 2020. arXiv: 2004.10934. URL: http://arxiv.org/abs/2004.10934.

[3] Navaneeth Bodla u. a. *Improving Object Detection With One Line of Code*. Techn. Ber. arXiv: 1704.04503v2.

[4] Thomas M Breuel. *The Effects of Hyperparameters on SGD Training of Neural Networks*. Techn. Ber. arXiv: 1508.02788v1.

[5] *CIFAR-10 and CIFAR-100 datasets*. URL: https://www.cs.toronto.edu/~kriz/cifar.html (besucht am 13. 10. 2020).

[6] COCO Consortium. *COCO - Common Objects in Context*. 2016. URL: https://cocodataset.org/#detection-2020 (besucht am 07. 10. 2020).

[7] Richard O. Duda, Peter E. Hart und David G. Stork. *Pattern Classification*. 2. Aufl. Wiley-Interscience, Nov. 2000. ISBN: 9780471056690.

[8] Mark Everingham u. a. „The pascal visual object classes (VOC) challenge“. In: *International Journal of Computer Vision* 88.2 (2010), S. 303–338. ISSN: 09205691. DOI: 10.1007/s11263-009-0275-4.

[9] *File:Kassenrezept Muster 2008.svg - Wikimedia Commons*. URL: https://commons.wikimedia.org/wiki/File:Kassenrezept_Muster_2008.svg (besucht am 29. 12. 2020).

[10] Martin A. Fischler und Robert A. Elschlager. *The Representation and Matching of Pictorial Structures*. Techn. Ber. 1973, S. 67–92. DOI: 10.1109/T-C.1973.223602.

[11] A. Fiszelew u. a. *Finding Optimal Neural Network Architecture Using Genetic Algorithms*. Techn. Ber. 2007, S. 15–24.

[12] Jörg Frochte. *Maschinelles Lernen*. München: Carl Hanser Verlag GmbH & Co. KG, 2019. ISBN: 978-3-446-45996-0. DOI: 10.3139/9783446459977. URL: https://www.hanser-elibrary.com/doi/book/10.3139/9783446459977.

[13] Golnaz Ghiasi, Tsung Yi Lin und Quoc V. Le. *Dropblock: A regularization method for convolutional networks*. Techn. Ber. 2018. arXiv: 1810.12890.

[14] Benyamin Ghojogh u. a. *The Theory Behind Overfitting, Cross Validation, Regularization, Bagging, and Boosting: Tutorial*. Techn. Ber. arXiv: 1905.12787v1.

[15] Ross Girshick. *Fast R-CNN*. Techn. Ber. arXiv: 1504.08083v2. URL: https://github.com/rbgirshick/.

[16] Ross Girshick u. a. *Rich feature hierarchies for accurate object detection and semantic segmentation Tech report (v5)*. Techn. Ber. arXiv: 1311.2524v5.

[17] Xavier Glorot und Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. Techn. Ber. URL: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[18] Ian J Goodfellow, Jonathon Shlens und Christian Szegedy. *EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES*. Techn. Ber. arXiv: 1412.6572v3.

[19] Walter J Gutjahr. „A Graph-based Ant System and its convergence“. In: *Future Generation Computer Systems* 16.8 (2000), S. 873–888. ISSN: 0167-739X. DOI: https://doi.org/10.1016/S0167-739X(00)00044-3. URL: http://www.sciencedirect.com/science/article/pii/S0167739X00000443.

[20] Kaiming He u. a. *Deep Residual Learning for Image Recognition*. Techn. Ber. arXiv: 1512.03385v1.

[21] Kaiming He u. a. *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*. Techn. Ber. arXiv: 1406.4729v4.

[22] Sergey Ioffe und Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. Techn. Ber. 2015. arXiv: 1502.03167.

[23] *Joe Redmon auf Twitter: Ï stopped doing CV research because I saw the impact my work was having. I loved the work but the military applications and privacy concerns eventually became impossible to ignore. https://t.co/DMa6evaQZr/ Twitter*. URL: https://twitter.com/pjreddie/status/1230524770350817280 (besucht am 14. 01. 2021).

[24] Min Lin, Qiang Chen und Shuicheng Yan. *Network In Network*. Techn. Ber. arXiv: 1312.4400v3.

[25] Tsung-Yi Lin u. a. *Feature Pyramid Networks for Object Detection*. Techn. Ber. arXiv: 1612.03144v2.

[26] Shu Liu u. a. *Path Aggregation Network for Instance Segmentation*. Techn. Ber. arXiv: 1803.01534v4.

[27] Wei Liu u. a. *SSD: Single Shot MultiBox Detector*. Techn. Ber. arXiv: 1512.02325v5. URL: https://github.com/weiliu89/caffe/tree/ssd.

[28] Ilya Loshchilov und Frank Hutter. *SGDR: STOCHASTIC GRADIENT DESCENT WITH WARM RESTARTS*. Techn. Ber. arXiv: 1608.03983v5. URL: https://github.com/loshchil/SGDR.

[29] Andrew L Maas, Awni Y Hannun und Andrew Y Ng. „Rectifier nonlinearities improve neural network acoustic models“. In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing* 28 (2013).

[30] Umberto Michelucci. *Advanced applied deep learning: Convolutional neural networks and object detection*. Apress Media LLC, Sep. 2019, S. 1–285. ISBN: 9781484249765. DOI: 10.1007/978-1-4842-4976-5.

[31] Diganta Misra. *Mish: A Self Regularized Non-Monotonic Activation Function*. Techn. Ber. arXiv: 1908.08681v3. URL: https://github.com/digantamisra98/Mish.

[32] Chao Peng u. a. *MegDet: A Large Mini-Batch Object Detector*. Techn. Ber. arXiv: 1711.07240v4.

[33] Peter Rechenberg, Hrsg. *Informatik-Handbuch*. 2., aktual. München ; Wien: Hanser, 1999, S. 298–299. ISBN: 3-446-19601-3 and 978-3-446-19601-8.

[34] Joseph Redmon. *Screen_Shot_2018-03-24_at_10.48.42_PM.png (1760×1486)*. URL: https://pjreddie.com/darknet/yolo/ (besucht am 25. 01. 2021).

[35] Joseph Redmon und Ali Farhadi. *YOLO9000: Better, faster, stronger*. Techn. Ber. 2017, S. 6517–6525. DOI: 10.1109/CVPR.2017.690. arXiv: 1612.08242.

[36] Joseph Redmon und Ali Farhadi. *YOLOv3: An Incremental Improvement*. Techn. Ber. 2018. arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767.

[37] Joseph Redmon u. a. *You only look once: Unified, real-time object detection*. Techn. Ber. 2016, S. 779–788. DOI: 10.1109/CVPR.2016.91. arXiv: 1506.02640v5. URL: http://pjreddie.com/yolo/.

[38] Shaoqing Ren u. a. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. Techn. Ber. 6. 2017, S. 1137–1149. DOI: 10.1109/TPAMI.2016.2577031. arXiv: 1506.01497.

[39] Hamid Rezatofighi u. a. *Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression*. Techn. Ber. arXiv: 1902.09630v2.

[40] Azriel Rosenfeld und Mark Thurston. „Edge and Curve Detection for Visual Scene AnaIysis“. In: *IEEE Transactions on Computers* C-20.5 (1971), S. 562–569. ISSN: 00189340. DOI: 10.1109/T-C.1971.223290.

[41] Reuven Y. Rubinstein und Dirk P. Kroese. *The Cross-Entropy Method A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. New York: Springer, 2004. ISBN: 978-1-4419-1940-3. DOI: https://doi.org/10.1007/978-1-4757-4321-0.

[42] Olga Russakovsky u. a. *ImageNet Large Scale Visual Recognition Challenge*. Techn. Ber. 3. 2015, S. 211–252. DOI: 10.1007/s11263-015-0816-y. arXiv: 1409.0575. URL: http://image-net.org/challenges/LSVRC/.

[43] Gerard Salton und Michael J. McGill. *Introduction to modern information retrieval*. New York: McGraw-Hill, 1986. ISBN: 978-0-07-054484-0.

[44] Karen Simonyan und Andrew Zisserman. *VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION*. Techn. Ber. 2015. arXiv: 1409.1556v6.

[45] Christian Szegedy u. a. *Intriguing properties of neural networks*. Techn. Ber. arXiv: 1312.6199v4.

[46] Christian Szegedy u. a. *Rethinking the Inception Architecture for Computer Vision*. Techn. Ber. arXiv: 1512.00567v3.

[47] Mingxing Tan, Ruoming Pang und Quoc V. Le. *EfficientDet: Scalable and Efficient Object Detection*. Techn. Ber. 2020, S. 10778–10787. DOI: 10.1109/cvpr42600.2020.01079. arXiv: 1911.09070.

[48] Chien Yao Wang u. a. *CSPNet: A new backbone that can enhance learning capability of CNN*. Techn. Ber. 2020, S. 1571–1580. DOI: 10.1109/CVPRW50498.2020.00203. arXiv: 1911.11929.

[49] Sanghyun Woo u. a. *CBAM: Convolutional Block Attention Module*. Techn. Ber. arXiv: 1807.06521v2.

[50] Yuxin Wu und Kaiming He. *Group Normalization*. Techn. Ber. arXiv: 1803.08494v3. URL: https://github.com/facebookresearch/Detectron/.

[51] Zhuliang Yao u. a. *Cross-Iteration Batch Normalization*. Techn. Ber. arXiv: 2002.05712v2. URL: https://github.com/Howal/.

[52] Steven Robert Young u. a. „Optimizing deep learning hyper-parameters through an evolutionary algorithm“. In: (2015). DOI: 10.1145/2834892.2834896. URL: http://dx.doi.org/10.1145/2834892.2834896.

[53] Sangdoo Yun u. a. *CutMix: Regularization strategy to train strong classifiers with localizable features*. Techn. Ber. 2019, S. 6022–6031. DOI: 10.1109/ICCV.2019.00612. arXiv: 1905.04899. URL: https://github.com/clovaai/CutMix-PyTorch.

[54] Matthew D Zeiler und Rob Fergus. *Visualizing and Understanding Convolutional Networks*. Techn. Ber. arXiv: 1311.2901v3.

[55] Nick Zeng. *An Introduction to Evaluation Metrics for Object Detection*. 2018. URL: https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/ (besucht am 06. 10. 2020).

[56] Zhaohui Zheng u. a. *Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression*. Techn. Ber. 07. 2020, S. 12993–13000. DOI: 10.1609/aaai.v34i07.6999. arXiv: 1911.08287. URL: https://github.com/Zzh-tju/DIoU.

[57] Barret Zoph u. a. *Learning Transferable Architectures for Scalable Image Recognition*. Techn. Ber. arXiv: 1707.07012v4.