Brandenburg
University of Technology
Cottbus - Senftenberg

Training of Artificial Neuronal Networks with Nonlinear Optimization Techniques

Thierry Tchouto Mbatchou

Cottbus Mathematical Preprints
COMP # 15(2021)

Brandenburg University of Technology
Faculty of Mathematics, Computer Science, Electrical and Information Technology

Bachelor Thesis

Training of Artificial Neuronal Networks with Nonlinear Optimization Techniques

---

Thierry Tchouto Mbatchou

A Thesis submitted in partial fulfilment of the requirements for the degree of
BACHELOR OF SCIENCE IN BUSINESS MATHEMATICS

Supervisor: Prof. Dr. Armin Fügenschuh

Second Assessor: Prof. Dr. Carsten Hartmann

Department of Engineering Mathematics and Numerics of Optimization

January 2021

Contents

List of Figures

List of Tables

Declaration in lieu of oath

I herby declare that, to the best of my knowledge this thesis is the product of my own independent work. All content and ideas drawn from external sources, directly or indirectly, published or unpublished, are indicated as such. This thesis has neither been previously submitted in whole or in parts, for a degree at this university or any other university.

Cottbus, 06.01.2021 Signature

1 Introduction

Suppose someone back in the year 1950 wanted to write a computer code that can take an image as input, and tell whether it is a cat or a dog figuring on the image. Since it is impossible to model a cat or a dog with a mathematical equation, the task becomes even more challenging. The quest after a solution to such challenges spawned the birth of machine learning and artificial intelligence.

Machine learning is a field of computer science that studies algorithms aiming at the automation of quite intricate problems that conventional programming methods cannot solve. A conventional programming method is made of two main steps. Given a specification of the program (what the program is to do and not how it is to do it), the first step would be to meticulously sketch the design of the program, that is to say, a fixed set of steps or rules for solving the problem. The next step, would then be to transform this design into a written code in a given computer language. For many real-world problems, it can be quite tough to apply this approach, for creating a detail design can be quite puzzling despite clear specifications.

Suppose using a conventional programming method we wanted to write a computer code to detect handwritten characters in an image. Moreover, we assume we have a dataset consisting of a large number of images of handwritten characters at our disposal. In addition, each data (image) in the dataset is labelled. The goal of using labels is to help the computer know how it should behave. We recall that the aim is to come forth with a program that is very likely to recognize characters from any given image (including those not belonging to the training dataset). The pattern to tackle this challenge using a conventional approach, would be to first of all study the images in the dataset and try to understand the relation between the images and the labels. The next step would then be, to spawn a general set of rules to detect characters in any given image. Due to the fact that, a given character can be written in many different ways it can be quite tough to establish this set of rules.

In the 1980s there were some quite unsuccessful attempts to tackle complex problems using the conventional approach. Even if the attempts were successful, the conventional approach still remains very impractical. There are miscellaneous complex problems out there therefore using this approach will mean, we will have to handle each problem individually. Machine learning algorithms actually spare us that effort, because they can solve many of these hard

problems in a quite generic way. They learn the detailed design from the data and their accuracy is proportional to the dimension of the dataset [7].
The goal of any machine learning algorithm is to figure out a model, in other words, a set of rules from a labelled dataset, in order to predict with the least error possible the labels of data points (images for example) not figuring in the dataset. To achieve this, a certain cost function is to be minimized. That is often done with the stochastic gradient descent (SGD) and the backpropagation algorithm. Though this approach is a combination of two algorithms: stochastic gradient descent and backpropagation, for the sake of amenity we shall be referencing it in our work simply as the *SGD approach*. The reason why the SGD approach is often used is because trying to minimize a cost function on large scale data with common optimization methods like the gradient descent is quite costly. The crux of our work is testing a different approach with a software called AMPL. It offers a wide range of solvers to optimize non linear functions. They have gained great standing both in the research sector as in the industrial domain. This made us quite inquisitive about what the outcome will eventually be if we utilized them to minimize the cost function of a machine learning problem, precisely of a classification problem. The peculiarity of the approach we will be testing is that it neither uses the stochastic gradient descent nor the backpropagation method. So in the following, we shall firstly take a look at how to model a machine learning problem into an optimization problem. Then we shall dive into more details about the SGD approach. After that, we will not only explain in more details our approach (which shall be referenced in our entire work as *the AMPL approach*), but we shall also study it to gain some insight into factors that influence its performance. Finally we shall compare the performance of both approaches: the AMPL approach and the SGD approach on some few machine learning problems.

2 Modelling a machine learning problem into an optimization problem

In this section, we shall look at how one can transform a machine learning problem into an optimization problem using artificial neural networks.

2.1 Artificial neural networks

An artificial neural network is a medley of interconnected neurons, where each neuron takes in multiple inputs but has a single output. This output is a function of the sum of the inputs the neuron receives [1], as illustrated in figure 2.1. The function at the output of the neuron is called the *activation function*.

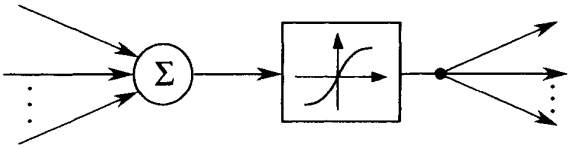

Figure 2.1: Single neuron illustration [1]

The symbol seen in figure 2.2 is used to represent a single neuron. The single output of a neuron is actually a part of the input of many other neurons, that is why many outgoing arrows can be seen on figure 2.2

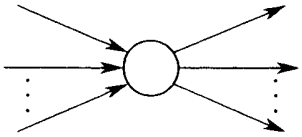

Figure 2.2: Single neuron [1]

In a neural network, the input of each neuron is the weighted output of other neurons. In a feedforward neural network, the neurons are connected in such a way that the data flow is one-directional. Each neuron receives inputs only from neurons in the preceding layer [1]. Figure 2.3 illustrates the structure of a feedforward neural network.

The first layer in such a network is the *input layer* and the last layer is the *output layer*. The other layers are called *hidden layers*. We can apprehend a neural network as a particular implementation of a map from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n is the number of inputs $x_1, \ldots, x_n$ and m the number of outputs $y_1, \ldots, y_m$. The implemented map is a function of the weights and biases in the network. The bias of a neuron can be understood as the threshold from which the neuron can send an output. For a given input, the computation of the associated output is realized by the collective effect of individual input-output characteristics of each neuron [1].

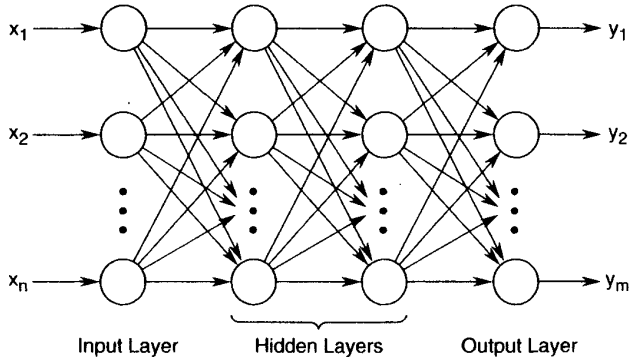

Figure 2.3: Structure of a feedforward neural network [1]

2.2 The cost function

Given a dataset on which we want to train the network, the main aim is to find the value of the weights of the interconnections and that of the bias of each neuron so that our model can predict as good as possible the label of any given image not belonging to the training dataset. In order to achieve this, we need to minimize a certain loss function.
In section 2.1, we talked about the *activation function*. To keep it simple we will be considering the sigmoid function as our activation function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{2.1}$$

We can consider the sigmoid function as being a smoothed version of a step function, the latter actually mimics the behaviour of a neuron in the brain.

In the sense that, it fires (gives an output equal to one) when the input is large enough, or remains inactive otherwise (gives an output equal to zero). Moreover, the amenity in using the sigmoid function resides in the fact that its derivate takes the simple form

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)). \tag{2.2}$$

Applying basic calculus formulas leads to the proof of (2.2).

To keep track with the notation, we interpret the sigmoid function in a vectorized sense. For a given $z \in \mathbb{R}^m$, $\sigma : \mathbb{R}^m \to \mathbb{R}^m$ is defined by applying the sigmoid function componentwisely, that is:

$$(\sigma(z))_i = \sigma(z_i).$$

For a given neural network, in each hidden layer, the output of every neuron is a unique real number, which is transmitted to each neuron in the next layer. At the next layer, each neuron forms its own weighted combination of these values, adds its bias to this combination and then squeezes it into the sigmoid function. We consider a as the output vector of a given layer, the output vector of the next layer is given through the following mathematical expression:

$$\sigma(Wa + b). \tag{2.3}$$

Here, W is a matrix and b is a vector. W contains the weights and b contains the biases. The number of columns in W is equal to the number of neurons of the previous layer, and the number of rows in W is equal to the number of neurons present in the actual layer, which is also the number of components in b. To stress the role of the ith neuron in (2.3), we can express the value of its output through the following expression:

$$\sigma(\sum_j w_{ij}a_j + b_i),$$

where the sum runs over all entries in a.

Suppose our network has L layers, with layers 1 and L being respectively the input and output layers. Moreover, we suppose layer l, for $l = 1, 2, 3, \ldots, L$, contains n_l neurons. This actually means n_1 is the dimension of the input data. Overall, the network maps from $\mathbb{R}^{n_1}$

to $\mathbb{R}^{n_L}$. We use $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ to denote the matrix of weights at layer l. Precisely, $w_{jk}^{[l]}$ is the weight that neuron j at layer l applies to the output from neuron k at layer $l-1$. In a quite similar way, $b^{[l]} \in \mathbb{R}^{n_l}$ is the vector of biases for layer l. So neuron j at layer l uses the bias $b_j^{[l]}$ [8].
We consider $x \in \mathbb{R}^{n_1}$ as the input. Furthermore let $a_j^{[l]}$ denote the output, or *activation*, from neuron j at layer l. We can then summarize the action of the network through the following:

$$a^{[1]} = x \in \mathbb{R}^{n_1}, \tag{2.4a}$$

$$a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l}, \quad \text{for} \quad l = 2, 3, \ldots, L. \tag{2.4b}$$

(2.4a) and (2.4b) amount to an algorithm for feeding the input forward through the network, such as to produce a final output $a^{[L]} \in \mathbb{R}^{n_L}$. Suppose we have N data $\{x^{\{i\}}\}_{i=1}^N \in \mathbb{R}^{n_1}$ in the training set, the cost function can take the following form:

$$Cost = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2. \tag{2.5}$$

We would like to stress that the cost function is a function of all the weights and biases present in the network. Furthermore, the cost function used in (2.5) is known as the *mean squared error*. There are other loss functions that can be used, like the: log (cross-entropy) loss, squared log loss, L_1 loss [9]. Since the mean squared error is the only loss function we effectively used in our entire work, we shall focus mainly on it in the other theoretical parts of our work.

3 Minimization of the cost function: the mainstream method

The mainstream method consists of using a combination of the stochastic gradient descent and the backpropagation algorithm. Though there is much we can say about these two algorithms, we will nevertheless restrict ourself in this section to talking about the strict minimum necessary for the understanding of our work.

3.1 Stochastic gradient descent

The stochastic gradient descent is a *gradient descent* based method which consists of computing iteratively the minimum of a given function. Suppose we want to minimize the cost function: $Cost : \mathbb{R}^s \to \mathbb{R}$, and our current vector is $p \in \mathbb{R}^s$. We are actually seeking a step Δ_p, so that $p + \Delta_p$ represents an improvement. Assuming Δ_p is small enough, we can ignore the term of order $\|\Delta_p\|^2$ in the Taylor series expansion of $Cost(p + \Delta_p)$. We then have the following

$$Cost(p + \Delta_p) \approx Cost(p) + (\nabla Cost(p))^T \Delta_p. \tag{3.1}$$

We recall that our goal is finding a step Δ_p that minimizes $Cost(p + \Delta_p)$. Considering (3.1), we see that we can achieve that in choosing Δ_p such that $(\nabla Cost(p))^T \Delta_p$ is as negative as possible. Using the *Cauchy-Schwarz inequality* which states that for any given $f, g \in \mathbb{R}^s$ we have $|f^T g| \leq \|f\|_2 \|g\|_2$. This implies that the most negative value of $f^T g$ is: $-\|f\|_2 \|g\|_2$. This scenario actually occurs when $f = -g$. Hence, based on the insight won from the *Cauchy-Schwarz inequality*, we know the direction of Δ_p is to be the same as that of $-\nabla Cost(p)$.

Not obliviating that (3.1) holds only for small Δ_p, we shall therefore limit ourself to taking a little step in that direction [8].
This leads to the following update formula:

$$p \to p - \eta \nabla Cost(p). \tag{3.2}$$

Note that η in (3.2) is considered in this context as being the *learning rate*. The steepest descent method consists in choosing a start vector $p_0 \in \mathbb{R}^s$ and iterating over and over again using (3.2) till a certain stop criterion is satisfied [8].

We can rewrite (2.5) as:

$$Cost = \frac{1}{N} \sum_{i=1}^{N} C_{x^{\{i\}}}, \tag{3.3}$$

where $C_{x^{\{i\}}}$ denotes the following:

$$C_{x^{\{i\}}} = \frac{1}{2} \| y(x^{\{i\}}) - a^{[L]}(x^{\{i\}}) \|_2^2. \tag{3.4}$$

This implies:

$$\nabla Cost(p) = \frac{1}{N} \sum_{i=1}^{N} \nabla C_{x^{\{i\}}}(p). \tag{3.5}$$

Dealing with a large number of training points and parameters makes the steepest descent method quite expensive regarding the computation cost. If we replace the mean of the individual gradients over the entire training dataset, by the gradient at a single randomly chosen training point, we can build then a cheaper version of the steepest descent method known as the *stochastic gradient descent*. A single step of the SGD may be summarized as follows:

1. Choose an integer i uniformly at random without replacement from $\{1, 2, 3, \ldots, N\}$.
2. Update
$$p \to p - \eta \nabla C_{x^{\{i\}}}(p). \tag{3.6}$$

Taking a close look at each step, we notice that the full training set is represented by a single randomly chosen point at each iteration. The more we iterate and the algorithm sees more

training data, we actually hope it convergences to a certain minimum. To understand why this method convergences, we would like to refer to [10].
A representation of the simplest version of the SGD is (3.6). There are more interesting variants like the *mini-batch* for example [10].
If we regard the stochastic gradient descent as an approximation of the mean (3.3) over all training points, then it will be quite interesting to think about a sort of compromise using a little sample average. That is exactly what the *mini-batch* is all about. So for some $m \ll N$, the updates with the *mini-batch* take the following form:

1. Choose m integers $k_1, k_2, k_3, \ldots, k_m$ uniformly at random without replacement from $\{1, 2, 3, \ldots, N\}$.

2. Update
$$p \to p - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x^{\{k_i\}}}(p). \tag{3.7}$$

3.2 Backpropagation

In the previous subsection, we saw that the update formulas to compute each new step using the SGD always required calculating the gradient of a certain function, see (3.6). This function is actually a function of p which contains our parameters (weights and biases). For the sake of clarity we will not be using p anymore, we will henceforth clearly denote the parameters $w_{jk}^{[l]}$ when talking of the weights and $b_j^{[l]}$ when talking of the biases.

For a given training point, we consider $C_{x^{\{i\}}}$ in (3.4) as being a function of the weights and biases. We can simply leave out the dependence on $x^{\{i\}}$ and just write

$$C = \frac{1}{2} \| y - a^{[L]} \|_2^2. \tag{3.8}$$

The dependence of C on the weights and biases is induced by $a^{[L]}$ alone (output from artificial neural network). The reason behind this resides in the fact that the vector y, which represents the label, is a constant that is initialized before training the network. Hence, y cannot be a function of the weights and biases. We will introduce some additional variables to spawn some interesting informations from the partial derivatives. That being said let

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l} \quad \text{for} \quad l = 2, 3, \ldots, L. \tag{3.9}$$

We consider $z_j^{[l]}$ as being the *weighted input* of neuron j at layer l. Now considering (2.4b) and (3.9) we get the following

$$a^{[l]} = \sigma(z^{[l]}) \quad \text{for} \quad l = 2, 3, \ldots, L. \tag{3.10}$$

We now define $\delta^{[l]} \in \mathbb{R}^{n_l}$ as the error vector of layer l through the following expression:

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} \quad \text{for} \quad 1 \leq j \leq n_l \quad \text{and} \quad 2 \leq l \leq L, \tag{3.11}$$

where $\delta_j^{[l]}$ is a metric for the sensitivity of the cost function to the weighted input of neuron j at layer l [8].
Before moving a step forward we define the componentwise product of two vectors. For $a, b \in \mathbb{R}^n$ then $a \circ b \in \mathbb{R}^n$ is given by $(a \circ b)_i = a_i b_i$. Using then the chain rule, we obtain the following results:

Lemma 3.2.1.

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^{[L]} - y) \tag{3.12a}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \qquad \text{for} \quad 2 \leq l \leq L-1, \tag{3.12b}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for} \quad 2 \leq l \leq L, \tag{3.12c}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for} \quad 2 \leq l \leq L. \tag{3.12d}$$

Proof. We shall commence by proving (3.12a).

Using the chain rule we can express $\delta_j^{[L]}$ as

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}}. \tag{3.13}$$

With $l = L$, (3.10) shows that $a_j^{[L]} = \sigma(z_j^{[L]})$ therefore

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}) \tag{3.14}$$

$$\frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \frac{1}{2} \sum_{k=1}^{n_L} (y_k - a_k^{[L]})^2 = -(y_j - a_j^{[L]}). \tag{3.15}$$

Combining now (3.13), (3.14), and (3.15) we obtain

$$\delta_j^{[L]} = (a_j^{[L]} - y_j)\sigma'(z_j^{[L]}), \tag{3.16}$$

where (3.16) is just the componentwise form of (3.12a).

Now we shall prove (3.12b). We consider $\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}}$. We want to differentiate C with respect to $z_j^{[l]}$, that is equivalent in this case to differentiating C with respect to all neurons in layer $l+1$. Hence, we can consider C as being a function of neurons from the layer $l+1$. Taking the total derivative with respect to $z_j^{[l]}$ we obtain the following recursive expression for the derivative. We can actually express $\delta_j^{[l]}$ as:

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}. \tag{3.17}$$

From (3.9) we know we can express the following

$$z_k^{[l+1]} = \sum_{s=1}^{n_l} w_{ks}^{[l+1]} \sigma(z_s^{[l]}) + b_k^{[l+1]}. \tag{3.18}$$

Deriving then $z_k^{[l+1]}$ with respect to $z_j^{[l]}$ yields the following equation:

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \sigma'(z_j^{[l]}). \tag{3.19}$$

Combining (3.19) and (3.17) yields the following

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{kj}^{[l+1]} \sigma'(z_j^{[l]}).$$

We can rewrite it as

$$\delta_j^{[l]} = \sigma'(z_j^{[l]})((W^{[l+1]})^T \delta^{[l+1]})_j.$$

This is nothing other than then componentwise form of (3.12b).

Now let us dive into the proof of (3.12c).

From (3.9) and (3.10) we can express the following

$$z_j^{[l]} = (W^{[l]}\sigma(z^{[l-1]}))_j + b_j^{[l]}. \tag{3.20}$$

Due to the absence of any dependence between b_j and $z^{[l-1]}$, we get the following expression

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \tag{3.21}$$

From the chain rule we can express the following

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} = \delta_j^{[l]}. \tag{3.22}$$

Finally we shall prove (3.12d).
We know that we can rewrite (3.9) as

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}.$$

Deriving $z_j^{[l]}$ with respect to $w_{jk}^{[l]}$ yields the following:

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]}. \tag{3.23}$$

We obtain:

$$\frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = 0 \quad \text{for} \qquad s \neq j. \tag{3.24}$$

Considering C as a function of the neurons at layer l, we can write the total derivative of C with respect to $w_{jk}^{[l]}$ as follows:

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}}. \tag{3.25}$$

Using now (3.23) and (3.24) we obtain

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} a_k^{[l-1]} = \delta_j^{[l]} a_k^{[l-1]}. \tag{3.26}$$

Hence, the proof of Lemma 3.2.1 is complete. □

We can perform a forward pass in the network thanks to equations (2.4a), (2.4b), (3.9), and (3.10) then evaluate the final output $a^{[L]}$ in computing successively $a^{[1]}, z^{[2]}, a^{[2]}, z^{[3]}, \ldots, a^{[L]}$. That being done, we can now compute $\delta^{[L]}$ with (3.12a). (3.12b) yields through a *backward pass* the values of $\delta^{[L-1]}, \delta^{[L-2]}, \ldots, \delta^{[2]}$. Equations (3.12c) and (3.12d) spawn the values of the desired partial derivatives. Using such an algorithm to compute gradients is known as *backpropagation*.
Taking a look at the definition of a partial derivative, we understand that the entity $\frac{\partial C}{\partial w_{jk}^{[l]}}$ estimates how much change occurs with the variable C due to a little variation in $w_{jk}^{[l]}$.

In the end, training a neural network with the stochastic gradient descent and backpropagation is all about doing a forward pass in the network using the values of the weights and biases from the preceding iteration (at the first iteration the values of the weights and biases are randomly initialized) and then computing the value of the final output $a^{[L]}$. Hence, we can perform a backpropagation and readjust the values of the weights and biases keeping in view the desired output. This readjustment is performed thanks to the equations in Lemma 3.2.1. These equations actually reveal to us how we are to change the weights and biases in order to take a step in the right direction (one which minimizes the loss function).

4 Minimization of the cost function: the AMPL approach

The AMPL approach is implemented in a software called AMPL, which actually justifies the name of the approach. AMPL is a modelling language for mathematical programming that was conceived in 1985. One of the noteworthy aspects of AMPL is the fact that its arithmetic expressions are akin to customary algebraic notations. Moreover, AMPL also uses algebraic notations to express ordinary mathematical programming structures (for example network flow constraints and piecewise linearities). AMPL offers a quite supple interface, allowing the user to choose from various solvers the one that fits best to his problem.
The main disparity between our approach with AMPL and the SGD approach is the fact that, contrary to the SGD approach where the gradient of the cost function is computed at randomly chosen training points, in our approach with AMPL, we minimize the cost function either over the entire dataset when it is not too big for the capacity of the computer or we break down the dataset into small batches and minimize the cost function iteratively over each batch. In the end, we often average the different values of the weights and biases we obtained through minimizing the cost function over the different batches. We proposed also another scheme to combine the weights and biases yielded by the different batches in section 5.2.2.

4.1 The solvers in AMPL

AMPL proffers a wide range of solvers to solve miscellaneous optimization problems. Since the cost function is nonlinear, we tested some interesting solvers AMPL offers for nonlinear optimization problems: Baron, Conopt, Knitro, Lgo, Loqo, Minos, and Snopt. In order to avoid making our work unnecessarily weighty, we shall explain the functioning of just one solver: Conopt. The choice is based on the fact that it is one of the solvers we used the most.

4.1.1 Conopt

AMPL/Conopt is an algorithm that is based on the GRG (generalized reduced gradient) and was conceived with the main intention to solve large non-linear programming problems. Suppose we wanted to minimize (or maximize) a function $f(x)$ which is subject to some constraints that we express through $g(x) = b$ (this is very often a system of nonlinear equations). Conopt postulates that f and g are differentiable and have smooth first derivatives. If that is not the case, perhaps because of the use of functions like abs, min, max, round or trunk, then Conopt will have no idea about that. It will make use of an approximation of the real model using function values and first-order derivatives and it can easily get stuck in areas where any of the previously mentioned quantities is not continuous [11]. Some of the main assumptions under which Conopt properly functions are the following [12]:

1. All functions used in the problem (objective and constraint functions) are at least twice differentiable.
2. All functions used in the problem are defined for every value of the optimization variables that satisfies the bounds.
3. The functions are sparse, that is to say, there are manifold zero Jacobian elements in each equation.

In the following subsection, we will dive into the GRG which is the gist of the algorithm used by Conopt.

4.1.1.1 GRG algorithm

The generalized reduced gradient is an algorithm to solve nonlinear problems having a general structure. Suppose the nonlinear problem one wanted to solve had the form

$$\text{minimize} \quad f(X) \tag{4.1}$$

$$\text{subject to} \quad g_i(X) = 0 \quad , \quad i = 1, \ldots, m \tag{4.2}$$

$$l_j \leq X_j \leq u_j \quad , \quad j = 1, \ldots, n \tag{4.3}$$

where X is a n dimensional vector and u_j, l_j are given lower and upper bounds such that $u_j > l_j$. The Form (4.1)-(4.3) is entirely general because we can always transform inequality constraints to equalities as in (4.2), simply by adding some slack variables. Hence the components of the vector X are both the original variables of the problem and the previously mentioned slack variables [13].

We suppose $m < n$. The main idea of the GRG is to use the equalities in (4.2) to express m of the variables, called *basic variables*, in terms of the remaining $n - m$ *nonbasic* variables. This is also exactly how the simplex method of linear programming works (see [14]). Suppose $\bar{X}$ is a feasible point, $\bar{y}$ the vector of basic variables and $\bar{x}$ that of the nonbasic at $\bar{X}$, so that X is partitioned as

$$X = (y, x), \quad \bar{X} = (\bar{y}, \bar{x}) \tag{4.4}$$

then (4.2) can be written as

$$g(y, x) = 0, \tag{4.5}$$

where

$$g = (g_1, \ldots, g_m). \tag{4.6}$$

We suppose that the objective f and constraint functions g_i are differentiable. Hence, making use of the implicit function theorem [15], a sufficient prerequisite for (4.5) to have a solution $y(x)$ for all x in some neighborhood of $\bar{x}$, is that the $m \times m$ *Jacobian matrix* $\frac{\partial g}{\partial y}$, evaluated at $\bar{X}$ is nonsingular [13]. If we assume the previously mentioned condition is fulfilled, then we may express the objective as a function of x only:

$$F(x) = f(y(x), x). \tag{4.7}$$

Hence the nonlinear problem is transformed into a *reduced problem* having only upper and lower bounds for x close to $\bar{x}$:

$$\text{minimize} \quad F(x) \tag{4.8}$$

subject to

$$l_{NB} \leq x \leq u_{NB} \tag{4.9}$$

where l_{NB} and u_{NB} are the vectors of bounds for x. The GRG algorithm solves the initial problem (4.1)-(4.3) by solving a series of problems of the form (4.8)-(4.9). One can solve such

problems by applying some simple modifications to unconstrained minimization algorithms [13].

So that the reduced problem (4.8)-(4.9) yields useful results, x should freely vary about the current point $\bar{x}$. It is true the bounds in (4.9) restrain x, but it is not difficult to move x in directions which keep these bounds satisfied. The bounds on the basic variables raise a more grave problem. Suppose some components of $\bar{y}$ are already at their bounds, then any slight change in x from $\bar{x}$ may cause the violation of some bounds. To impede this from occurring and to ensure the beinghood of the function $y(x)$, we postulate that the following, which we name *nondegeneracy assumption* holds:
At any given point X satisfying (4.2)-(4.3), there is a partition of X into m basic variables y and $n - m$ non-basic variables x such that

$$l_B \leq y \leq u_B \tag{4.10}$$

where l_B and u_B are the vectors of bounds on y and $B = \frac{\partial g}{\partial y}$ is nonsingular [13].

Suppose we tried solving the reduced problem (4.8)-(4.9) starting from some feasible point $X = \bar{X}$ with basic variables y and nonbasic variables x. Considering (4.7), to evaluate the function $F(x)$ it is fundamental we know the values of the basic variables $y(x)$. It is true that apart from linear cases and a few nonlinear scenarios, the function $y(x)$ cannot be determined in closed forms. Nevertheless, we can use an iterative scheme that solves (4.5) to compute $y(x)$ for any given x. Hence an algorithm to solve the reduced problem starting from a point $X_0 \equiv \bar{X}$, is [13]

1. Set $i = 0$.
2. Replace x_i into (4.5) and determine the corresponding values of y_i through an iterative method for solving nonlinear equations.
3. Find a motion direction d_i for the nonbasic variables x.
4. Select a step size α_i such that
$$x_{i+1} = x_i + \alpha_i d_i$$
This is quite often accomplished by solving the following one dimensional search problem
$$\text{minimize} \quad F(x_i + \alpha d_i)$$

with α restrained such that $x_i + \alpha d_i$ does not violate the bounds on x. This one dimensional search will require that we reiteratively apply step (2) to evaluate F for various values of α_i.

5. Test the actual point $X_i = (y_i, x_i)$ for optimality. If not optimal, set $i = i + 1$ and return to (2)

If in step (2) the value of at least one component of y_i exceeds its bound, the procedure must be interrupted. Let us consider the simple case in which only one basic variable is out of its bounds. Hence this variable is to be transformed into a nonbasic variable and some component of x which is not on a bound is to be transformed into a basic variable. After this *basis altering*, we obtain a new function $y(x)$, a new $F(x)$, and a new reduced problem [13].

It is possible to implement the GRG without making use of either derivatives of f or of the functions g_i. This requests schemes for solving nonlinear equations and minimizing nonlinear functions subject to bounds using no derivatives. Though there is no doubt about the beinghood of such methods, there are rather many more schemes which work using derivatives. Their efficiency is better comprehended and they could establish themselves over the years in the solving of large problems [13].

Hence we will be considering GRG algorithms that request first derivatives of f and g.
If we consider minimizing F using derivatives we therefore need a formula for ∇F. F is most certainly differentiable if f and g are and if $\frac{\partial g}{\partial y}$ is nonsingular, which also entails that the implicit function $y(x)$ is differentiable [13]. From (4.7) we get

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} + (\frac{\partial f}{\partial y})^\intercal \frac{\partial y}{\partial x_i}. \tag{4.11}$$

In order to evaluate $\frac{\partial y}{\partial x_i}$ we use the fact that, if

$$g_j(y(x), x) = 0 \quad , \quad j = 1, \ldots, m \tag{4.12}$$

for all x in some neighbourhood of $\bar{x}$, then

$$\frac{\mathrm{d}g_j}{\mathrm{d}x_i} = 0 = (\frac{\partial g_j}{\partial y})^\intercal \frac{\partial y}{\partial x_i} + \frac{\partial g_j}{\partial x_i} \quad , \quad j = 1, \ldots, m \tag{4.13}$$

or, in matrix form

$$(\frac{\partial g}{\partial y})(\frac{\partial y}{\partial x_i}) + \frac{\partial g}{\partial x_i} = 0. \tag{4.14}$$

$(\frac{\partial g}{\partial y})$ being nonsingular at $\bar{X}$ (*nondegeneracy assumption*) entails that

$$(\frac{\partial y}{\partial x_i}) = -(\frac{\partial g}{\partial y})^{-1}\frac{\partial g}{\partial x_i} \equiv -B^{-1}\frac{\partial g}{\partial x_i}. \tag{4.15}$$

Using (4.15) in (4.11)

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} - (\frac{\partial f}{\partial y})^{\intercal}B^{-1}\frac{\partial g}{\partial x_i}. \tag{4.16}$$

Let

$$\pi = (\frac{\partial f}{\partial y})^{\intercal}B^{-1}. \tag{4.17}$$

Using (4.17), the components of ∇F are

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} - \pi^{\intercal}\frac{\partial g}{\partial x_i} \tag{4.18}$$

If $\bar{X}$ is optimal for (4.1)-(4.3), and if the gradients of all tethering constraints are linearly independent (see [16]), then the *Kuhn-Tucker conditions* (see [17]) hold at $\bar{X}$ [13]. In other words we can use the *Kuhn-Tucker conditions* to check how good our current point is and therefore search further or stop the search if the current point satisfies the previously mentioned conditions.

5 Problems and models

During our entire work, we tried to model three different machine learning problems with AMPL and solve them with some solvers that AMPL offers. In this section, in a first step, we present the different machine learning problems we modelled using AMPL. Practical implementation issues like which activation function or which network structure was used will not be handled in this chapter, but in the next chapter. In a second step we shall talk about the various approaches we used in order to solve these problems.

5.1 Problems

5.1.1 State of the cell recognition problem

In this problem, we trained a network that could predict if a given image of a cell was either in the *interphase* state (period between two successive cell divisions [18]) or the *metaphase* state (stage of mitosis and meiosis in which chromosomes become arranged in the equatorial plane of the spindle [18]).
The images from the training set and test set are of size 60×60 pixels. We received them from the department of multi-parameter diagnostics of the Brandenburg University of Technology Cottbus-Senftenberg.

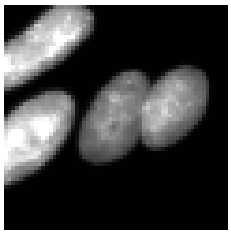

Figure 5.1: Interphase

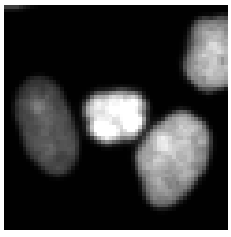

Figure 5.2: Metaphase

On both images above, the cell concerned is the cell in the center. The peculiarity of this problem is that, contrary to the others (polygon recognition and handwritten digit recognition, on a given image, we do not only have the relevant element. Cropping the picture, or taking

just part of the center of the image (considering every pixel found in a $l \times w$ rectangle sharing the same center with the picture), does not always help to isolate the relevant cell. Firstly, because this cell is not all the time perfectly fitted in the center of the image (figure 5.3 and figure 5.4 portray some extreme cases). Secondly, because the surrounding cells are sometimes too close to it (figure 5.1). Moreover, resolving to work with the entire image is not a very good option, for the images are of size 60×60. This implies our input layer will have 3600 neurons. Our system will therefore have a huge amount of variables and hence the computational cost will be tremendous.

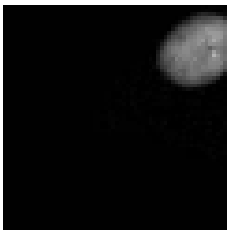

Figure 5.3: Interphase

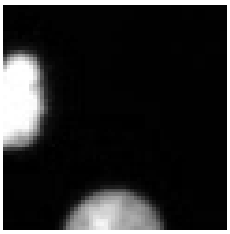

Figure 5.4: Metaphase

5.1.2 The polygon recognition problem

In the polygon recognition problem, we aimed at training the computer to predict accurately if a given image is a triangle, a four-sided polygon, or a five-sided polygon. We used a Python script to generate our dataset. The images are of size 10×10, see figures (5.5)-(5.7).

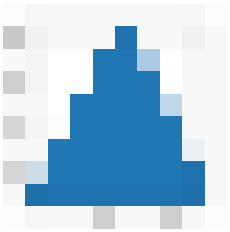

Figure 5.5: Triangle

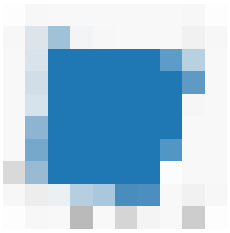

Figure 5.6: Four-sided polygon

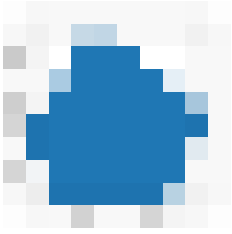

Figure 5.7: Five-sided polygon

The reason why we went for such a small image size lies in the fact that we wanted a dataset such that we could train the neural network at once on the entire training set without dreading high computational costs.

5.1.3 The handwritten digit recognition problem

The handwritten digit recognition problem is a well-known problem in the machine learning community. The aim is to train a neural network that can predict if a given image is an integer between 0 and 9. The data used for that purpose is called the MNIST dataset from the National Institute of Standards and Technology. In the training set, we have a collection of handwritten numbers from 250 different people, of which 50% are high school students and 50% are from the Census Bureau. The MNIST dataset amounts to a total of 60,000 images in the training set and 10,000 in the test set, each of size 28×28 pixels with 256 gray levels [2]. One can download it online. Figure 5.8 displays some extracts from it.

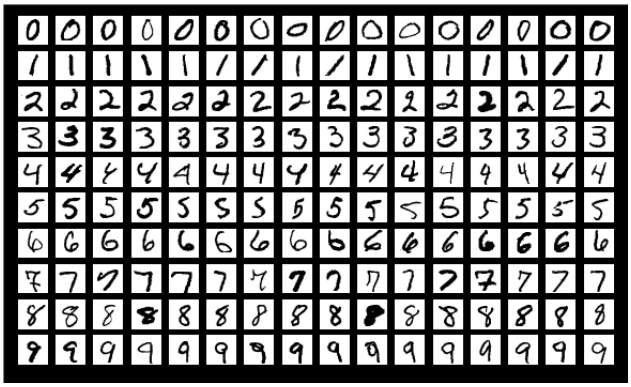

Figure 5.8: MNIST data [2]

5.2 Models

In this subsection, we shall present the different models, approaches we tested in order to solve the various machine learning problems we mentioned in the previous subsection.

5.2.1 The basic approach

The "basic approach" is the most intuitive approach one could conceive in order to model machine learning problems with AMPL and then solve them with one of the solvers AMPL offers. All our other approaches are derived from it. In the following, we describe the main steps of the basic approach.

1. In the *.mod file*:

   1.1. Declaration of the different variables that will be used in the model: the number of hidden layers, number of neurons in each hidden layer, weights, and biases etc.

   1.2. Defining the cost function. In our case, we always used the mean squared error. We nevertheless tried other loss functions such as the *binary cross-entropy*: $cost = -\frac{1}{n_L}\sum_{i=1}^{n_L} y(x^{\{i\}})\log\Big(a^{[L]}(x^{\{i\}})\Big) + (1-y(x^{\{i\}}))\log\Big(1-a^{[L]}(x^{\{i\}})\Big)$ and the *categorical cross entropy* [19]: $cost = -\sum_{i=1}^{n_L} y(x^{\{i\}})\log\Big(a^{[L]}(x^{\{i\}})\Big)$. We recall that all the terms used in both sums above (n_L,$x^{\{i\}}$, $a^{[L]}$ etc.) are to be understood as defined in section 2.2. The issue with these loss functions is that in our case (AMPL), the elements of the output vectors of hidden layers are somehow very close to zero (sometimes even equal to zero) and hence the logarithms of these elements are too great to be evaluated. So AMPL always prints out an error notification.

   1.3. Defining the constraints in our model. We do this by implementing the algorithm of a forward pass (2.4a and 2.4b) as a restriction to our model. That is to say we implement it preceded by the AMPL keyword *subject to*.

2. In the *.dat file*:

   2.1. Assignation of values to some parameters, for example, number of neurons per layer, the number of hidden layers, etc.

3. In the *.run file*:

   3.1. Loading the data. Beforehand the pixels of the images are saved in Excel sheets. In this step, we simply read from the sheets row by row and store the values in the corresponding variables we created in the *.mod file*. Actually, each row corresponds to the entire pixels of an image, and in the case of the MNIST dataset, the first element of a row corresponds to the label of the image.

3.2. Normalizing the input data. We do this by multiplying each pixel by $\frac{0.99}{255}$ and adding to it 0.01. We divide by 255 because we assume that our images have an 8-bit depth. Hence, the value of each pixel is found in the interval [0.01, 1]. Suppose we were using the sigmoid function as activation function. Then for quite big pixel values, the activation function begins to flatten since 1 is an asymptotic value of the sigmoid function. This is quite problematic because it leads to saturation in the neural network [20]. Saturation can be understood as the state in which neurons in hidden layers of a neural network more than often output values close to the asymptotic ends of the activation function range. This results in a very slow and inefficient learning [21]. The reason why we added the offset of 0.01 after scaling the input pixels with $\frac{0.99}{255}$ is because zero input values (pixels) stifle the learning rate of the neural network [20].

3.3. Initializing the label tensor $y(x^{\{i\}})$ for every $i \in 1, \ldots, N$, see (2.5). For example in the case of the polygon recognition problem, where our images are either triangles, four-sided polygons, or five-sided polygons our label vector y will be a three-dimensional vector since we just have three classes. Normally it is proposed to do the following: For example, each time the image $x^{\{i\}}$ is a triangle we set $y(x^{\{i\}}) = (1, 0, 0)$, each time it is a four-sided polygon we set $y(x^{\{i\}}) = (0, 1, 0)$, and each time it is a five-sided polygon we set $y(x^{\{i\}}) = (0, 0, 1)$. Suppose we were using the sigmoid function as activation, 0 and 1 are asymptotic values and therefore impossible to attain, this might lead to very high values of the weights and biases in the network because it will be trying to attain this asymptotic values. In the the end, it will result in saturation in our Network [21]. That is why we use 0.99 instead of 1 and 0.01 instead of 0.

3.4. Defining the training set. We select simply elements from the training set on which we want effectively to train the network.

3.5. Initialization of variables.

3.6. Choosing a solver with the AMPL command *option solver "solver name"*.

3.7. Solving the optimization problem with the AMPL *solve* command.

5.2.2 Variant 1

In the basic approach, we assume all the data from the training set is used in order to train the model (step 3.4). The problem behind this approach is, when the training set is quite huge, training the network at once on all the images in the training set is impossible. We actually noticed that either the solver computes endlessly or it displays an error notification. Therefore we came forth with another variant in which we do not train the network at once on the entire training set, but rather train it progressively on little subsets of the training set (mini-batches). It enables us to attain higher computation speed. We implement this by placing steps 3.4 to 3.7 in a for loop (we shall call this loop, loop1 for ulterior references). At the end of each iteration, we save the new values of the weights and biases. Once the program is done running, we combine the values of the weights and biases from each mini-batch to get the final values of the weights and biases of our model. We had two approaches to combine weights and biases yielded by the mini-batches.
Approach 1: It is a quite intuitive method, where we just multiple by $\frac{1}{N_b}$ each weight and bias yielded by each mini-batch. N_b denotes the number of mini-batches used.
Approach 2: Suppose we used N_b mini-batches to train the network. Let f_i be the loss function that was minimized while training the network on the ith mini-batch. The loss function varies from mini-batch to mini-batch even though it is always the mean squared error we use for each mini-batch. Before the forward pass in the network, we can consider the loss function (the general expression of the loss function) as being a function of the weights and biases and of a set of parameters that represent the pixel values of the images in the mini-batch. Once the forward pass is done, those parameters take real values and the only unknown remain the weights and biases. Therefore it is obvious that for two different mini-batches different expressions of the loss function will be spawned. Now let us assume that F is the loss function (expression of the loss function) we would have obtained, had we trained the network over the entire data set at once without splitting it into mini-batches. It is easy to notice that: $F(x) = \sum_i^{N_b} f_i(x)$ with x being a representation of the weights and biases. Let $v_{min}^{(i)}$ be the value of a local minimum of f_i. Without making any strong assumptions on our functions, there is no mathematical theorem we can use to prove that $\frac{1}{N_b}\sum_i^{N_b} v_{min}^{(i)}$ is close enough to a local minimum of F. To stress this, we shall consider a simple one dimensional case (x is a scalar) with $N_b = 2$ and $F(x) = f_1(x) + f_2(x) = x^2 + 1$. $f_2(x)$ and $f_1(x)$ respectively being equal to 1 and x^2. 0 and 10^{100} are respectively local minimums of f_1 and f_2, but $x = \frac{0+10^{100}}{2}$ is far from being a local minimum of F. This led us to conceive a second approach to combine the different weights and biases yielded by the various mini-batches. Firstly, we recall that a forward pass in our final neural network (after

the combination of the weights and biases from the different mini-batches) looks like the following:

$$a^{[1]} = x \in \mathbb{R}^{n_1}, \tag{5.1a}$$

$$a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l}, \quad \text{for} \quad l = 2, 3, \ldots, L. \tag{5.1b}$$

Written with more details, equation (5.1b) looks like the following: $a^{[l]} = \sigma(\sum_j w_{ij}^{[l]} a_j^{[l-1]} + b_i^{[l]})$. Each weight and bias is a combination of weights and biases from all the N_b mini-batches. It entails that for a given weight w_{ij} (we unheed the layer from which the weight or the bias is) and a given bias b_i we have the following: $w_{ij} = \sum_{l=1}^{N_b} \lambda_l w_{ij}^{(l)}$ and $b_i = \sum_{l=1}^{N_b} \beta_l b_i^{(l)}$, with λ_l and $\beta_l \in \mathbb{R}$, $l \in \{1, \ldots, N_b\}$. We want to underline here that $w_{ij}^{(l)}$ is the value we obtain for the weight w_{ij} after training the network on the lth mini-batch, the same goes for $b_i^{(l)}$. Suppose $cost_1$ is the value of the loss function, see (2.5), for $\lambda_l = \frac{1}{N_b}$ and $\beta_l = \frac{1}{N_b}$, $l \in \{1, \ldots, N_b\}$. In this approach, we aim at choosing λ_l and β_l, $l \in \{1, \ldots, N_b\}$ such that the value of the corresponding loss function $cost_2$ fulfills : $cost_2 \leq cost_1$. We want to underline that the values of $cost_2$ and $cost_1$ are computed over the union of all the mini-batches that were used in the training process.

5.2.3 Variant 2

In the basic approach, we said, in step 3.5 that the variables are to be initialized, but we did not disclose how they were to be initialized. During our different experiments, we used two initialization methods. In the first method (constant initialization), we initialized the variables (weights and biases) with the same constant. In the second method (random initialization) we used two well-known schemes in the machine learning community: *Xavier initialization* and *He initialization*. The goal behind the *Xavier initialization* is to preserve the variance of the activation and the back-propagated gradient as one moves forward and backward in the network [22]. Hence the back-propagated signal is preserved and the activation signal is kept from exploding to a high value or vanishing to zero as it moves through the hidden layers. Concretely the *Xavier initialization* works as following: The biases are initialized with zeros and each weight between layer l and layer $l+1$, $l = 1, \ldots, L-1$ (keeping with the notation in section 2.2) is initialized with a random number chosen from the uniform distribution $\mathcal{U}(\frac{-\sqrt{6}}{\sqrt{n_l+n_{l+1}}}, \frac{\sqrt{6}}{\sqrt{n_l+n_{l+1}}})$ [22] .

In order to derive the *Xavier initialization*, Glorot and Bengio [22] postulated that the activations are linear, which is quite untrue for the $ReLU$ activation. The goal of the *He initialization* is to propose a more sound weight initialization scheme taking into account the $ReLU$ (rectified linear unit) activation, defined as $f(x) = \max(0, x)$. The principle used to achieve that remains nevertheless the same as the one used in the *Xavier initialization*, that is to say, the preservation of the variance of the activation and the back-propagated gradient. Furthermore, it was experimentally put into evidence that the *He initialization* could enable convergence in extremely deep convolutional neural network, while the *Xavier initialization* could not [23] . Concretely this is how the *He initialization* works, just as with the *Xavier initialization* every bias is initialized with zero, but each weight between layer l and layer $l+1$, $l = 1, \ldots, L-1$ is initialized with a random number taking from a zero-mean Gaussian distribution with variance $\frac{2}{n_l}$ [23].

We used the vantages of random weight initialization schemes (namely that at each program run we obtain a different solution) to spawn a new variant to solve machine learning problems with AMPL. These are the following adjustments we brought to variant 1 so as to obtain this variant.

1. In step 3.5 a random initialization method is used. We use either the *He initialization* or the *Xavier initialization* depending on which activation function we opted for.

2. In *loop 1* after the selection of the mini-batch on which the network will be trained, we include a second for loop (*loop 2*) in which we train the network t times on the same mini-batch. Since the variables (weights) are initialized randomly, we always obtain new values for our variables each time we enter *loop 2*. In each iteration in *loop 2*, we save the yielded values of the weights and biases and test the accuracy of the network on a validation set and save the result of the test. At the end of *loop 2*, we retain the values of the weights and biases of the iteration that had the best performance on the validation set. When that is done, we move to the next iteration in *loop 1*, where we select a new mini-batch and repeat the process all over again (*loop 2*).

3. At the end of *loop 1* we combine the different values of the weights and biases that were retained (they correspond somehow to the best result of each mini-batch) using *approach 1* described in *variant 1*. We made no use of *approach 2* for we dreaded that the entire variant (variant 2) will become in the end computationally too expensive.

4. A change is also brought to step 1.3. We bring in some further constraints on the variables. Let v be a variable (either weight or bias) in our network. We denote as v_{prev}

the value of v from the previous iteration in *loop 1*. Furthermore, v_{act} denotes the value of v in the actual iteration (*loop 1*). It is actually chosen from the best iteration at the end of *loop 2*. Hence we request that $|v_{act} - v_{prev}| \leq \epsilon$. The motivation behind this constraint is to keep the system working with the same neural network and not with another permutation of the network. It strongly helped to enhance the performance of the network.

In summary, what we do in this variant is to train the network on several mini-batches using *loop 1*, and on each mini-batch we train the network several times using *loop 2*, seeking the best result (with respect to the validation set) the batch can offer. At the end of *loop 2*, we retain the best result we could obtain and move to the next mini-batch and do the same (move to the next iteration in *loop 1*). At the end of *loop 1*, we combine the different values of the weights and biases retained at each iteration in *loop 1* using *approach 1* described in variant 1.

5.2.4 Variant 3

This variant is not really a variant on its own, but rather a slight altering we can bring into the previous variants. This change is all about imposing some restrictions on the cost function, thus telling the system how good we want the cost function to be. Taking for example the case of variants where we work with mini-batches, we can either request that, for a given real number ϵ_1, we want the value of the loss function f_i for every mini-batch i to be such that: $|f_i| \leq \epsilon_1$, or else we can require that: $|f_{i+1} - f_i| \leq \epsilon_1$, $i = 1, \ldots, N_b$, with N_b being the total number of mini-batches we used and f_i is the value of the loss function associated with the ith mini-batch. The inequality is given under the assumption that the neural network was minimized in an ascending order over the mini-batches with respect to their indices.

5.2.5 Variant 4

This variant was conceived specially for the handwritten digit recognition problem, but it is also possible to generalize it so that it is applicable to similar problems, namely in cases where the dimension of the input layer is quite big (greater than 500) and that of the output layer is greater than three. We will be presenting and explaining this variant based on the

handwritten digit recognition problem. In the following we shall try to explain the approach step by step:

1. We split the handwritten digit recognition problem into ten sub-problems. For more clarity, we represent each sub-problem with an index between zero and nine. In the sub-problem number i, with $i \in \{0, \ldots, 9\}$, we try to train a neural network that can predict if a given image is number i or not. This implies that the output vector y of the neural network is such that $y \in \mathbb{R}^2$. Suppose $y = (a, b)$, for $a > b$, we consider that the prediction of the neural network regarding the input image was in favor of number i, or else we consider it was in disfavour of number i.
   For each sub-problem i, we write a subscript i (subprogram i) that solves it. We can solve each of these sub-problems using any of the approaches we described previously. For the sake clarity, we will consider that, each time the subprogram i, $i \in \{0, \ldots, 9\}$ predicted that an image of a given number between zero and nine was number i, the output of the neural network was 1, else we consider the output to be 0. This actually means that we map the output $y = (a, b)$ of the neural network to 1, when $a > b$, else we map it to 0.

2. After solving each sub-problem i, we test the neural network that was trained in it, on a validation set. The results of this different tests are saved in a matrix $A \in \mathbb{R}^{10\times10}$, such that $s_{ii} = A_{i,i}$, $i \in \{0, \ldots, 9\}$ holds the value of the success rate of subprogram i in recognizing number i. In $A_{i,j}$, $i \in \{0, \ldots, 9\}$, $j \in \{0, \ldots, 9\}$ with $i \neq j$ we save $\gamma_{ij} = 1 - s_{ij}$, where s_{ij} is the success rate of subprogram i in predicting that an image of number j is not number i. We can also understand γ_{ij} as the error subprogram i commits in recognizing an image of number j as not being i. In other words, we mean the error it commits in differentiating number j from number i.

3. Our final program is a medley of the ten subprograms we previously presented. The final program functions as following, when we present it an image of a given number between zero and nine, we pass the image to the ten different subprograms we mentioned earlier. There are actually several possible outcomes, which we will like to analyze in details in the following.

   3.1. *Only one subprogram returned a* 1 *as output.* The output of our final program in this case will simply be the index of that subprogram. Our decision is based on the fact that we assume that the $s_{ii} = A_{i,i} \geq 0.9$ (which is actually what we

observed during our experiments), that being said, it would be very likely that our output was actually the right prediction.

3.2. *All the ten subprograms returned zero as output.* Actually, in this scenario, we know that the outputs of nine of the ten subprograms are correct and that just one subprogram yielded a wrong output. This subprogram will most likely be the subprogram j, such that $A_{j,j} = \min_{i\in\{0,\dots,9\}} A_{i,i}$. Hence we return j as output (prediction) of the final program.

3.3. *More than one subprogram returned a* 1 *as output* . Let $J = \{l_1, \dots, l_p\}$, with $p \leq 10$, be the set that holds the indices of those subprograms. Moreover, we assume that among those subprograms there is exactly one that made a correct prediction. The challenge now is to try to figure out which subprogram it could be. We consider G to be a set holding the possible values of the label of our input image. Because of our previous assumption we set $G := J$ (initialize G with J). For $j \in J$: if $\exists g \in G$ such that $\gamma_{jg} = A_{j,g} = 0$, because of the definition of γ in (2), it is impossible for g to actually be the right prediction of the label of our input image. Hence we delete g from G. That is to say, we set $G := G\backslash\{g\}$. Now let us analyze the possible outcomes after this operation.

3.3.1. If $card(G) = 1$, then we return as prediction the unique element found in G

3.3.2. If $card(G) = 0$, then this implies our assumption according to which the label p of the input image was in the set G was wrong. The fact that it was deleted from the set makes it impossible to have been the right prediction just as we explained in *3.3*. Since the label p of the input image was not in the set G, it implies that the subprogram p gave a 0 as output after reading the image. It will therefore very likely be that, among all the existing subprograms, the subprogram p has the lowest s_{pp} value. In other words, p is such that $A_{p,p} = \min_{k\in\{0,\dots,9\}} A_{k,k}$.

3.3.3. If $card(G) > 1$, then we conceive two decision-making schemes.
Method 1: we return p such that: $p = \arg\max\{\min_{k\in G\backslash\{j\}} A_{k,j}, j \in G\}$, where $\arg\max$ returns the value of the index of the column holding the maximum value.
Method 2: We create a real vector t with the length 10. If $j \in G$, then $t[j] = \{\min_{k\in G\backslash\{j\}} A_{k,j}\}$, else $t[j] = 0$. In addition we create a random number generator which returns a number j from G with the probability $\frac{t[j]}{\sum_{n\in G} t[n]}$.

Our final output will be the output of our random number generator. We shall take a look at an example to explain the two methods: *method 1* and *method 2*. Suppose $G = \{1, 2, 4\}$. Furthermore, we suppose that for $i, j \in G$, our matrix A looks like the following (we only included the relevant rows and columns of A):

$A_{i,j}$	$j = 1$	$j = 2$	$j = 4$
$i = 1$	0.99	**0.01**	0.06
$i = 2$	0.09	0.95	**0.02**
$i = 4$	**0.03***	0.05	0.98

The first thing we do, regardless the method we are using is to visit each column j, and mark the row i which makes j most unlikely to be the right output label, in other words, we highlighted i such that $A_{i,j} = \min_{i \in G \setminus \{j\}} A_{i,j}$, $j \in G$. Once we have this bottleneck values, what we do in *method 1* is to choose the column with the less restrictive (highest) bottleneck value. We recall that $\gamma_{ij} = A_{i,j}$ for $i \neq j$ is the probability that the subprogram i mistakes number j for number i. Taking a look at our example: If we suppose that the right output label was 2. It implies that subprogram 1 mistook a 2 for a 1, since it gave out an output equal to 1 after reading an image of a "supposed" 2. This happens only with a probability of 0.01.
If we suppose the right output label was 4, it means subprogram 2 mistook a 4 for a 2. This happens with a probability of 0.02. Finally, if we suppose that the right output label was a 1 it means subprogram 4 mistook a 1 for a 4, this happens with a likelihood of 0.03. We recall that we only mentioned the bottleneck values since they are the problematic ones. We notice therefore that the last scenario is the one that is most likely to occur. That is why if we used *method 1* we would return 1 as output.
The truth is, the fact that $\gamma_{12} = A_{1,2} = 0.01 < 0.03 = \gamma_{41} = A_{4,1}$ only makes 2 less unlikely to be the right output label compared to 1 that has a higher (less restrictive) bottleneck value. It is not impossible that it could actually be the right label (the same goes for 4). The likelihood that it could actually be the right output label is $\frac{0.01}{0.01+0.02+0.03} = 0.16$. In the same way, we can compute the likelihood that 4 or 1 is the right output label and we get respectively 0.33 and 0.5. In *method 2*, we take what we just explained previously into consideration. We do this by creating a random number generator that returns either 1,2 or 4 with the probabilities that were computed above.

6 Experiments and discussion

In this chapter, we aim at presenting the manifold experiments we conducted and analyze their various outcomes. In our first four experiments, we wanted to bring answers to questions like: what is the best solver to solve machine learning tasks using AMPL as modelling language? Is there any activation function that outstandingly performs better than the others? Do more hidden layers and neurons lead necessarily to a better performance? Are random weight initialization schemes better than initialization schemes with a constant?
Our first wave of experiments aims at bringing answers to all these and more. It is true that some of the questions we raised previously may seem trivial to folks from the machine learning community. Since our approach is quite different from the mainstream approach to solve machine learning problems, that is to say, the stochastic gradient descent combined with the backpropagation algorithm, we wanted to probe if the conclusions drawn regarding some of those questions would be experimentally valid in our case also. The second wave of experiments is mainly to test and evaluate the different models (variants) we presented in section 5.2. All our first four experiments (section 6.1-6.4) were conducted on the polygon recognition problem. We used a training set holding 540 images and a test set with 47 images of each of the three classes, hence amounting to a total of 141 images. Furthermore, we used the basic approach (see section 5.2.1) in all our first four experiments.

6.1 Experiments on weight initialization schemes

We proposed in section 5.2.3 two main weight initialization schemes: constant weight initialization and random weight initialization (Xavier and He initialization). In this section, we wanted to put into evidence which scheme experimentally offers the best results in terms of accuracy of the network. We tested each of the initialization schemes on networks with different structures (different number of hidden layers and neurons). We realized all the various trials using Conopt as solver. Regarding the constant weight initialization scheme, in all our experiments in this subsection we always used zero as initialization constant (that is to say, we initialized all our weights with zero). The reason behind this resides in the fact that when we used other real numbers very often the accuracy of the network was very poor.

We utilize the following notation to describe the structure of the network: $(p, n_1, \ldots, n_p)$, where p is the number of hidden layers and n_i, $i \in \{1, \ldots, p\}$ the number of neurons in the hidden layer number i. We summarized the results of our various trials in table 6.1.

Table 6.1: Summary of weight initialization trials

Activation function: Sigmoid					
Initialization: Xavier initialization					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	16	97.87	95.74	100	97.87
$(1, 6)$	20	97.87	97.87	100	98.58
$(1, 10)$	126	100	95.74	100	98.58
$(2, 3, 3)$	32	97.87	97.87	100	98.58
$(3, 4, 3, 5)$	103	97.87	97.87	97.87	97.87
Initialization: He initialization					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	8	100	97.87	100	99.29
$(1, 6)$	26	97.87	95.74	100	97.87
$(1, 10)$	182	97.87	97.87	100	98.58
$(2, 3, 3)$	72	97.87	97.87	100	98.58
$(3, 4, 3, 5)$	141	97.87	100	89.36	95.74
Initialization: Constant initialization					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	9	87.23	95.74	89.36	90.77
$(1, 6)$	9	87.23	95.74	93.61	92.19
$(1, 10)$	391	87.23	97.87	93.61	92.90
$(2, 3, 3)$	1	0	0	100	33.33
$(3, 4, 3, 5)$	2	0	0	100	33.33

We used the total number of hidden neurons in a given network structure as parameter to represent our results graphically since it is difficult to come forth with a graphical representation that takes into consideration the structure of the network. In case we have several structures with the same number of hidden neurons with simply average their results, which could be either the time performance or the overall success rate. We utilized this method as our default method for the graphical representation of our results.
To be candid the previously mentioned representation scheme does not unveil fine details of

the experiments, precisely the disparities in terms of performances that arise in the different network structures. It is nevertheless a good means to offer a fast-comprehensible upshot of the experiments.

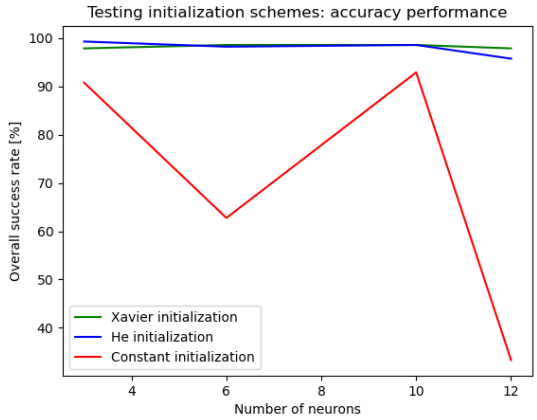

Figure 6.1: Test on initialization schemes: accuracy performance

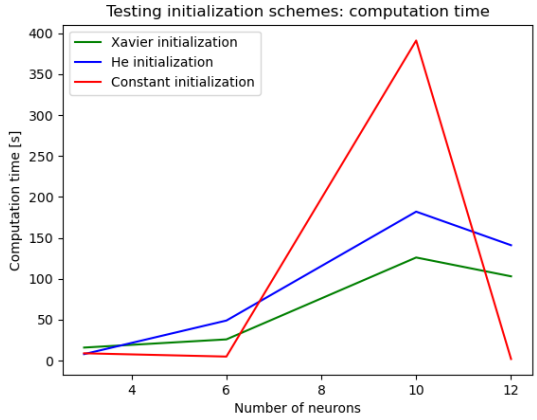

Figure 6.2: Test on initialization schemes: computation time

From table 6.1 it is clear that random weight initial schemes like *Xavier* or *He* outclass the constant weight initialization. In addition, we also noticed in other experiments that the constant weight initialization performs very poorly (overall success rate less than 40%) when one uses an activation function other than the sigmoid function. All these lead us to think that the constant weight initialization is no good choice because initializing all the weights with the same constant makes all the weights have the same value and the activation of the

neurons identical. Hence, the derivative of the loss function is selfsame for every weight in a weight matrix of a given layer. All the weights having the same value in every iteration leads to symmetric hidden layers. Since each neuron of a given hidden layer computes the same function, we hence have a model that likens a linear model [24]. Hence, the model cannot handle complex, non-linear data and is effective only one layer deep. That is exactly why each time we utilized a multi-hidden layer structure we always observed very poor results (see table 6.1). Random weight initialization schemes hence help to impede this symmetry problem [25]. Therefore it is strongly recommended to make use of them [26].

6.2 Testing solvers

In this subsection, we tried comparing some of the various solvers AMPL offers (Baron, Conopt, Knitro, Lgo, Loqo, Minos, Snopt) to see how well they perform at the task of solving machine learning challenges. Just as we said at the beginning of this chapter our test problem is the polygon recognition problem. We used four different layer structures and tested each of the solvers previously mentioned on them. For each solver, we recorded its computation time and the accuracy of the network. We used as activation function the sigmoid function and we chose the *Xavier initialization* as initialization scheme. A Summary of the various trials is found in tables 6.2 and 6.3. A graphical representation is found in figures 6.3 and 6.4.

Table 6.2: Comparing solvers part 1

Activation function: Sigmoid					
Initialization: Xavier initialization					
Solver: Baron					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 5)$	595	0	0	100	33.33
$(1, 10)$	777	100	0	0	33.33
$(2, 4, 4)$	516	97.87	97.87	100	98.58
$(3, 4, 3, 4)$	518	97.87	97.87	100	98.58
Solver: Conopt					
$(1, 3)$	8	100	97.87	100	99.29
$(1, 10)$	182	97.87	97.87	100	98.58
$(2, 4, 4)$	68	100	97.87	100	99.29
$(3, 4, 3, 5)$	244	95.74	95.74	100	97.16

Table 6.3: Comparing solvers part 2

Activation function: Sigmoid					
Initialization: Xavier initialization					
Solver: Knitro					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 5)$	0	0	0	0	0
$(1, 10)$	0	0	0	0	0
$(2, 4, 4)$	0	0	0	0	0
$(3, 4, 3, 4)$	0	0	0	0	0
Solver: Lgo					
$(1, 5)$	1	0	100	0	33.33
$(1, 10)$	2	100	0	0	33.33
$(2, 4, 4)$	1	0	0	100	33.33
$(3, 4, 3, 4)$	8	0	0	100	33.33
Solver: Loqo					
$(1, 5)$	100	97.87	97.87	100	98.58
$(1, 10)$	1281	97.87	95.74	100	97.87
$(2, 4, 4)$	113	100	97.87	100	99.29
$(3, 4, 3, 4)$	196	97.87	97.87	97.87	97.87
Solver: Minos					
$(1, 5)$	43	97.87	91.48	78.72	89.35
$(1, 10)$	75	78.72	93.61	91.48	88.17
$(2, 4, 4)$	27	100	97.87	68.08	88.65
$(3, 4, 3, 4)$	38	97.87	93.61	0	63.82
Solver: Snopt					
$(1, 5)$	270	100	95.74	100	98.58
$(1, 10)$	1334	97.87	93.61	100	97.16
$(2, 4, 4)$	508	97.87	97.87	100	98.58
$(3, 4, 3, 5)$	6	0	100	0	33.33

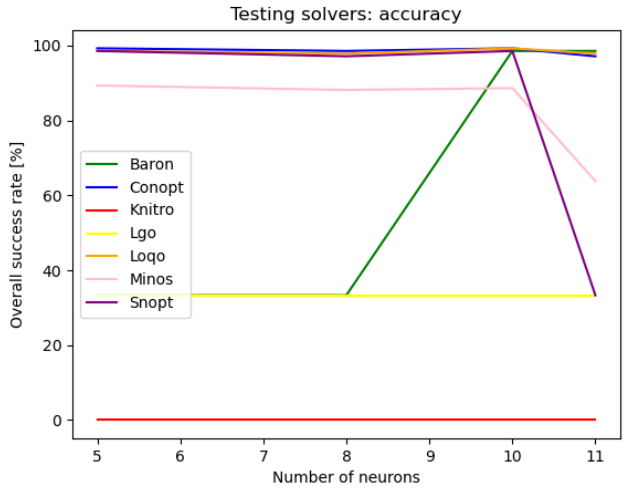

Figure 6.3: Test on solvers: accuracy performance

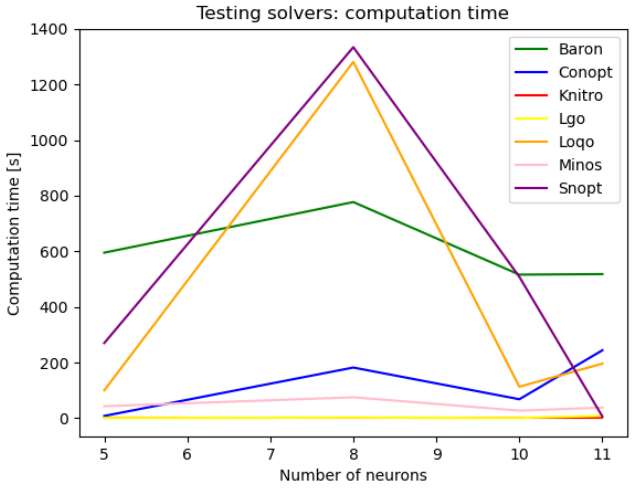

Figure 6.4: Test on solvers: computation time

From table 6.2 and table 6.3, we notice that Knitro and Lgo performed miserably compared to the other solvers. Hence, we will not recommend them for the task of solving machine learning problems. Though Baron performed very well on multi-hidden layer structures, it seems not to be efficient on single-hidden layer structures and is therefore disqualified from the league of the recommendable solvers. Minos performed quite averagely compared to top-ranking solvers like Conopt and Loqo. In some further experiments we conducted, we nevertheless noticed that it was among the three solvers that returned good results using the ReLU

as activation function, since it uses some special schemes to deal with almost everywhere differentiable functions. The other solvers are Loqo and Snopt. Snopt performed on all the network structures we used in our trials except for the last one (3,4,3,5), in terms of accuracy, almost as well as Conopt and Loqo. But just as the last network setup revealed, Snopt is not really good at handling multi-hidden layers structures, especially when the number of hidden layers is greater than two. Another downside of Snopt is its heavy computation time. For example, on the setup (1,10) it was seven times slower than Conopt. We noticed that this difference increases even more when we consider more complex problems like the handwritten digit recognition or the state of the cell recognition problem. The upside of Snopt is that just like Loqo and Minos it performs well in terms of accuracy of the network with the ReLU as activation function. In terms of accuracy, it is noteworthy that Conopt and Loqo yielded exceptionally good results in every network setup we tested. Compared to its fiercest rival Loqo, Conopt has the foredeal that it computes faster. On the setup (1,3) it was 12 times faster and on the setup (1,10) it was 7 times faster. Just as we mentioned previously this difference in the computation speed increases even more when the problem at hand is of larger size. Nevertheless, Loqo has the advantage that it performs well with the ReLU as activation function.

Taking into consideration all that we mentioned previously, we would draw as conclusion that there is no best solver. Nevertheless, we would strongly recommend the use of Conopt for most activation functions apart from the ReLU, since it has a lower computation time compared to the other solvers and yields very good results. In case one opts for the ReLU as activation function we would recommend the use of either Minos, Loqo or Snopt.

6.3 Influence of the number of hidden layers and neurons on the performance of the network

It is always quite puzzling to figure out the optimal number of neurons and hidden layers one should choose for a neural network. With optimal we mean the computation time is as small as possible and the results are as good as possible. Intuitively one will want to use a network with a great number of neurons and hidden layers to attain the best possible results the network can offer. In this subsection, we wanted to check experimentally if this intuitive idea is correct, that is to say, if more hidden layers and neurons often improve the accuracy of the network. For the tests we conducted in this subsection we used networks with an increasing number of hidden layers and neurons. We chose Conopt as solver, the sigmoid function as

activation, and the *Xavier initialization* as initialization method. We summarized our trials in table 6.4 and graphical illustrations of the outcome of the experiments are found in figures 6.5 and 6.6.

Table 6.4: Influence of the number of hidden layers and neurons on the performance of the network

Activation function: Sigmoid					
Initialization: Xavier initialization					
Solver: Conopt					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	8	100	97.87	100	99.29
$(1, 6)$	26	97.87	95.74	100	97.87
$(1, 10)$	182	97.87	97.87	100	98.58
$(1, 15)$	43	97.87	97.87	100	98.58
$(1, 26)$	254	97.87	97.87	100	98.58
$(2, 3, 3)$	68	100	97.87	100	99.29
$(2, 10, 15)$	644	97.87	100	100	99.29
$(3, 4, 3, 5)$	244	95.74	95.74	100	97.16
$(3, 10, 15, 20)$	2380	100	95.74	100	98.58
$(4, 16, 22, 18, 14)$	15571	100	97.87	100	99.29

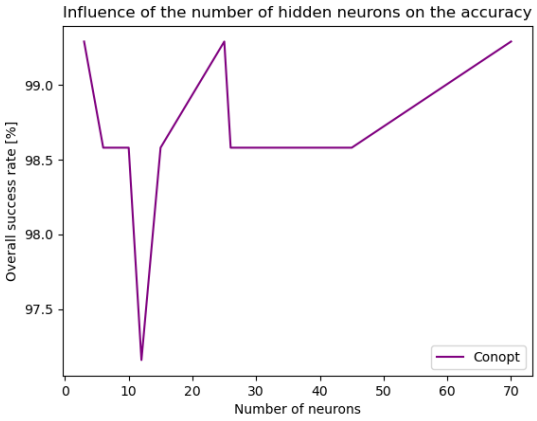

Figure 6.5: Influence of hidden neurons on accuracy

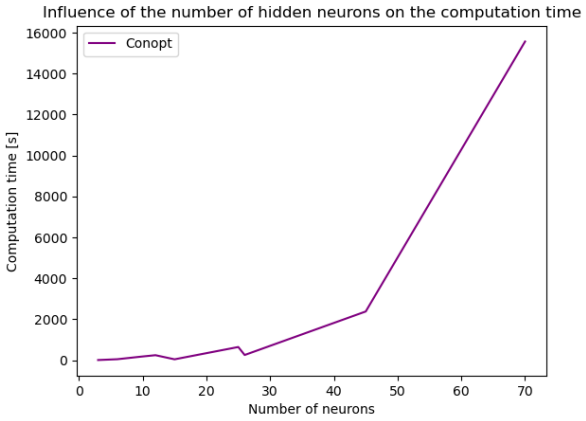

Figure 6.6: Influence of hidden neurons on computation time

We notice from table 6.4 that the best performance of the network could be attained using just a single hidden layer holding only three hidden neurons. Nevertheless, we could also achieve the same performance using more hidden layers and neurons (2,10,15). It is also noteworthy that though we used a structure of three hidden layers we could not attain an overall accuracy of 99%. The same goes for a single hidden layer structure with 26 hidden neurons. We could notice experimentally that more hidden layers and neurons do not always improve the accuracy of the network. This leads us to believe that the idea according to which more hidden neurons often improve the performance of the network is false. There is an optimal number of hidden neurons for each artificial neural network [27]. Furthermore, from a certain number of hidden neurons, the network tends to overfit the data and hence performs poorly [28].

6.4 Trials on activation functions

There are miscellaneous functions one can use as activation. In this subsection, we wanted to probe some popular activation functions and find out if perhaps there are any that perform outstandingly well compared to others. We considered as in our previous experiments the polygon recognition problem. We tried solving it using several setups (artificial neural networks with various number of hidden layers and neurons). On each setup, we tried the following activation functions: ReLU, sigmoid, softmax, tanh, and swish.

6.4.1 ReLU

In the year 2010 Nair and Hinton proposed the rectified linear unit (ReLU) as activation function. Hitherto it has been the mainstream activation function in the domain of deep learning applications with state-of-the-art results [29]. The ReLU activation function performs a threshold operation, setting input elements with negative values to zero, or else leaves them untouched. For a given vector $x \in \mathbb{R}^n$ the ReLU can be express by [30]

$$f(x) = \max(0, x) = \begin{cases} x_i, & \text{if} \quad x_i \geq 0 \\ 0, & \text{else} \end{cases}$$

A graphical representation of the ReLU function looks like the following image

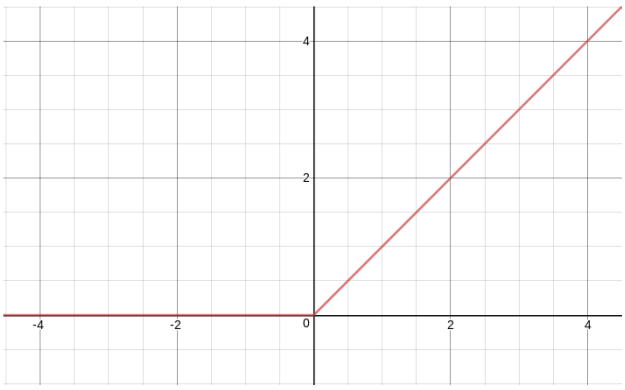

Figure 6.7: ReLU [3]

In table 6.5 is found a summary of our experiments using the ReLU as activation function.

Table 6.5: Influence of the activation function on the performance of the network: ReLU

Activation function: ReLU					
Initialization: He initialization					
Solver: Minos					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	197	95.74	95.74	100	97.16
$(1, 6)$	422	95.74	97.87	97.87	97.16
$(1, 10)$	816	95.74	97.87	100	97.87
$(2, 3, 3)$	154	95.74	97.87	97.87	97.16

6.4.2 Tanh

Simply defined the hyperbolic tangent (tanh) function is the ratio between the hyperbolic sine and the cosine function [31]. It is a smooth zero-centered function whose values reside between -1 and 1 [30]. For a given real number x, the tanh function can be expressed by the following [4]

$$tanh(x) = \frac{sinh}{cosh} = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}.$$

The curve of the tangent hyperbolic looks like the following image

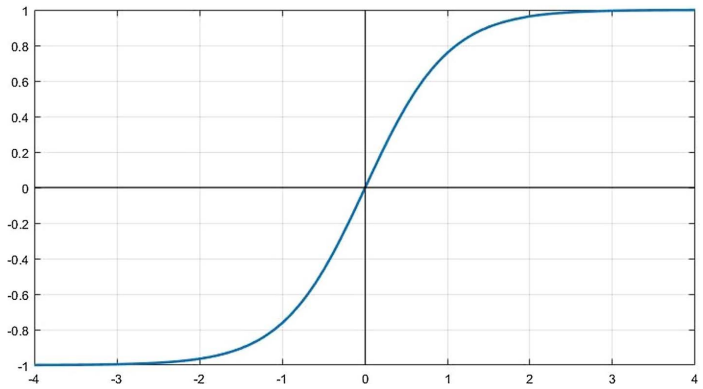

Figure 6.8: Tanh [4]

In section 5.2.1 we said that we initialize the label tensor with 0.01 and 0.99 because 0 and 1 are asymptotic values of the sigmoid function. As we can notice in figure 6.8, -1 and 1 are asymptotic values of the tanh. Hence, we initialize the label tensor with -0.99 and 0.99.
In table 6.6 is found a summary of our experiments using the tanh as activation function.

Table 6.6: Influence of the activation function on the performance of the network: Tanh

Activation function: Tanh					
Initialization: He initialization					
Solver: Conopt					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	18	97.87	95.74	100	97.87
$(1, 6)$	52	97.87	97.87	100	98.58
$(1, 10)$	79	100	95.74	100	98.58
$(2, 3, 3)$	129	97.87	97.87	100	98.58

6.4.3 Swish

Ramachandran et al. from the Google brain team presented in the year 2017 a newly conceived activation function under the name swish which was to present an alternative to the ReLU [32]. The swish function is a smooth non-monotonic function which is bounded at its lower limit and unbounded at its upper limit [30]. This function also known as the sigmoid weighted linear unit or Silu, given a real number x, is defined by [33]

$$\frac{x}{1 + \exp(-x)}$$

A graphical representation of the swish function is shown in figure 6.9.
In table 6.7 is found a summary of our experiments using the Silu as activation function.

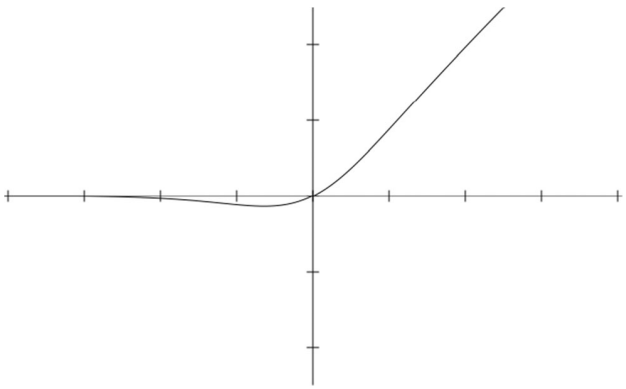

Figure 6.9: Swish [5]

Table 6.7: Influence of the activation function on the performance of the network: Swish

Activation function: Swish					
Initialization: He initialization					
Solver: Conopt					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	38	95.74	97.87	100	97.87
$(1, 6)$	168	97.87	91.48	100	96.45
$(1, 10)$	494	95.74	97.87	100	97.87
$(2, 3, 3)$	39	97.87	95.74	97.87	97.16

6.4.4 Sigmoid

The sigmoid is a nonlinear activation function often used in feed-forward neural networks [30]. It is bounded, differentiable and defined for all real values. Furthermore, it has positive derivatives and some degree of smoothness [34]. For a given real value x we can express it through the following

$$f(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

A graphical representation of the sigmoid is as shown in figure 6.10.

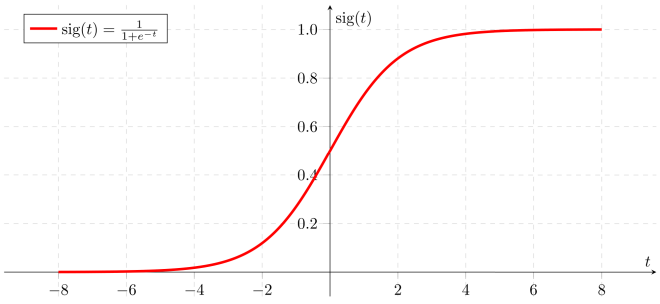

Figure 6.10: Sigmoid [6]

We summarized the various outcomes of our trials with the sigmoid function in table 6.8.

Table 6.8: Influence of the activation function on the performance of the network: Sigmoid

Activation function: Sigmoid					
Initialization: He initialization					
Solver: Conopt					
Structure of the network	Computation time [s]	Success rate triangle[%]	Success rate four-sided polygon[%]	Success rate five-sided polygon[%]	Overall success rate
$(1, 3)$	8	93.61	95.74	100	96.45
$(1, 6)$	10	100	97.87	100	99.29
$(1, 10)$	62	97.87	97.87	100	98.58
$(2, 3, 3)$	67	100	95.74	100	98.58

From what we can observe from tables 6.5 - 6.8, in terms of overall success rate there is no major difference between the various activation functions we probed. Even though the sigmoid seems to slightly perform better than the rest, we do not want to make any fast and hard conclusion regarding which activation is the best, because for a given setup there is no "definite" result. The reason behind this is the fact that if we reprise the tests for each activation function we would get different results due to the random initialization schemes we used.

There are nevertheless two important remarks we will like to make. The first is quite noteworthy. In terms of computation time the ReLU makes the network very slow. For example, compared to the scenario with the sigmoid function, the network with the ReLU was 13 times slower on the setup (1,10) and 42 times slower on the setup (1,6). This becomes a very serious issue when we handle larger problems like the handwritten digit recognition, because even Conopt takes hours, see days to handle such problems (see section 6.5). On the other hand, exponential activation functions like the sigmoid or the tanh are not flawless. We noticed something strange (*exploding activations*) during our various trials using this class of activation functions. We heeded that sometimes the weights and biases the solvers yield are such that when we try performing a forward pass in the network (or when we try to test a given data point) the weighted inputs of some neurons are such that their exponential values cannot be computed because they are too large.
This problem is even more recurrent when we try tackling larger problems like the handwritten digit recognition. This *exploding activations* problem recalls us a well-known problem in the machine learning community: *the exploding gradient problem*. The exploding gradient problem as introduced in [35], denotes a drastic increase of the norm of the gradient during the training [36]. The consequence of this is a very large update of the weights in the network.

Though the exploding gradient issue is not quite identical to the exploding activations problem, they nevertheless have some common traits: the weights are somehow too large. We noticed in some experiments we conducted and in which we were confronted with the exploding activations issue that though there were a great many weights with absolute values less than one, there were nevertheless some few with absolute values above a hundred. This prodded us to test some solutions for the exploding gradient in our own case. We discuss the outcome of these tests in section 6.5.

6.5 Models on trial

In this subsection, our aim is to see how good our different models perform on medium size machine learning problems like the handwritten digit recognition and the state of cell recognition. To achieve this, we realized multiplicious tests to put in evidence the performance of each of the models we presented in section 5.2.

6.5.1 Solving the exploding activations issue

Handling medium size machine learning challenges was not without hurdles. One of the greatest snags we confronted was the exploding activations problem. We observed it most often on the handwritten digit recognition problem independently of the size of the training set and the model used. We mentioned in section 6.4.4 that this problem likens the exploding gradient issue for which there is already a wide range of solving schemes.

A quite onefold solution is redesigning the network model to have fewer hidden layers [37]. This was actually of no help in our case for we trained our models generally with single-hidden layer structures and used in most of the cases less than 20 hidden neurons. Even when we tried altering the structure of the network by reducing the number of hidden neurons the problem was still not solved.

Another way to solve the exploding gradient problem is using *gradient clipping* [38]. We represented in section 3.1 the update formula of the stochastic gradient descent with (3.6). Gradient clipping imposes an upper bound on the update of p, by placing a constraint on the norm of the gradient [39]:

$$p \to p - \eta h_c \nabla C_{x^{\{i\}}}(p), \tag{6.1a}$$

$$h_c = \min\{\frac{\eta_c}{\|\nabla C_{x^{\{i\}}}(p)\|}, 1\}. \tag{6.1b}$$

η_c is a hyperparameter that is to be cautiously chosen. We want to mention that (6.1a) and (6.1b) present a clip by norm scheme and not a clip by value, where values of the gradient vector are scaled if they exceed a certain threshold. In the clip by norm, the whole gradient vector is scaled if its norm goes beyond a certain pre-set value. In the stochastic gradient descent this places a bound on the possible values of the step size during the training, hence preventing too large updates [39].
The previously mentioned scheme is actually not applicable in our case because we can't easily manipulate the gradient of the cost function. Even if that was possible making use of this method entails that for it to function properly we need to alter the algorithm with which Conopt works. That is actually not possible for end-users.

It is also proposed to use weight regularization to forestall the exploding gradient issue. There are two genres of weight regularization schemes: the L_1 (absolute weights) and the L_2 (squared weights) weight penalty [40]. When applying either of these weight regularization schemes the new loss function of the neural network is the sum of an unregularized objective and a regularization term. Considering a general form of the loss function expressed by $f(x)$, where x represents the weights in our model, the modified loss function $f_\lambda(x)$ using a weight penalty of type L_2 is given by the following expression [41]

$$f_\lambda(x) = f(x) + \lambda \|x\|_2^2, \tag{6.2}$$

where λ is a regularization parameter which should be properly chosen from $(0, 1)$. We want to mention that (6.2) is also known as the *Tikhonov regularization* [42]. In the same way, we can express the modified loss function using a weight penalty of type L_1 utilizing the L_1 norm in (6.2) rather than the L_2 norm. We hence obtain

$$f_\lambda(x) = f(x) + \lambda \|x\|_1, \tag{6.3}$$

which is also known under the name of *Lasso regularization* [43].
We tested the L_2 regularization with values of λ from the set $S_\lambda = \{0.1, 0.2, 0.3, \ldots, 0.9\}$.

The problem we tried solving was the handwritten digit recognition. Hence, we trained a neural network using the basic approach on a training set of 3000 images and tested it on a set holding 40 images of each number from zero to ten, thus amounting to a total of 400 images. A summary of our trials is found in table 6.9.

Table 6.9: L_2 regularization

Activation function: Sigmoid		
Initialization: Xavier initialization		
Solver: Conopt		
Structure of the network: (1,12)		
Value of λ	Computation time [s]	Overall success rate [%]
$\lambda = 0.1$	1876	58.75
$\lambda = 0.2$	3751	41.50
$\lambda = 0.3$	1689	37.75
$\lambda = 0.4$	1425	37.75
$\lambda = 0.5$	2672	40
$\lambda = 0.6$	2063	48
$\lambda = 0.7$	2423	44
$\lambda = 0.8$	2407	20
$\lambda = 0.9$	1945	45

We did not observe the exploding activations problem for any value of λ in S_λ but the accuracy of the neural network was very poor.
We also tried the L_1 regularization for $\lambda = 0.1$ on the same setup we mentioned previously. The test ran for roughly 24h and yielded no results. We can notice from table 6.9 that for every $\lambda \in S_\lambda$ the trials we conducted were always in a specific time range (1424s, 3752s). It will be very likely that trying the L_1 regularization for another value of λ might also run for 24h and yield no result. Hence, we did not probe the L_1 regularization for any other value of λ bar 0.1. A reason behind the very poor performance of both the L_1 and the L_2 regularization could be that the values of λ we probed were not optimal. There are miscellaneous methods proposed to tune hyperparameters, one of them is cross-validation [44]. It is thought to be a good way to determine an optimal value of λ [41]. Using cross-validation would still require that we choose our regularization parameter over a discrete set $\{\lambda_1, \ldots, \lambda_m\}$ and still train our network with these different values of λ. Having probed already a good number of interesting values λ can take, we were very pessimistic regarding the outcome of the cross-validation method. Hence, we did not implement it.

The last method we tried in order to solve the exploding activations issue is an approach we conceived on our own and it is quite heuristic. The idea was spawned by the fact that normalizing the data can help ward off the exploding gradient issue [45]. It is true we proposed in section 5.2.1 an approach to normalize the data. Since this approach did not help prevent the exploding activations issue we decided to try another data normalization scheme. Suppose $x \in \mathbb{R}^n$ is the pixel vector of an image in our data and n the total number of pixels contained in the image. For $i = 1, \ldots, n$ we set

$$x_i = \frac{x_i - \frac{\sum_{k=1}^{n} x_k}{n}}{\max\{x_1, \ldots, x_n\} - \min\{x_1, \ldots, x_n\}}.$$

We tested this other normalization scheme to see if it could help against the exploding activations issue and we noticed it was unsuccessful. Finally, we came up with the idea of combining both normalization schemes. That is to say, we normalize every image in our data set firstly with the normalization scheme proposed in section 5.2.1 and secondly with the recently proposed scheme in this subsection. We noticed in several experiments that it greatly contributed to ward off the exploding activations issue and contrary to the L_2 regularization yielded very good results in terms of accuracy of the network. It was thanks to it that we could successfully train neural networks on the handwritten digit recognition problem. We conducted a great many experiments with this double normalization scheme and could count just around 15 trials in which it was unsuccessful. That is not surprising since even the well-known schemes mentioned in this subsection are not always successful.

6.5.2 The basic approach

We tested the basic approach on the handwritten digit recognition problem training the network on a set of 3000 images. We tested the results on a set of 400 images holding 40 images of each number from zero to ten. We used the double normalization scheme presented in the previous subsection and the Xavier initialization. Furthermore, we used a network with 1 hidden layer and 12 hidden neurons. We chose the Sigmoid function as activation and Conopt as solver. We ran the test 5 times. It took around 10h and the best iteration yielded an overall success rate of 93.25% on the test set.

The network was trained just on 3000 data because we heeded that when we use more data for example 4000 the solver either yields very poor results (overall success rate of around 10%) because it computes for a very short period of time, or it aborts the test (training)

in less than 5 minutes without yielding any result. Though Conopt can handle successfully models with 1000000000 equations and variables [46] the previously mentioned case seems to overwhelm it.

The basic approach was also tested on the state of the cell recognition problem. In section 5.1.1, we talked about our strife to isolate the relevant element in the image. We tried two different methods. In the first method, we simply resize the images to a 28×28 format using the resize function from the PIL library in Python. In the second approach, we hew the original image into a 28×28 format by selecting every pixel found in a 28×28 square sharing the same center with the image.

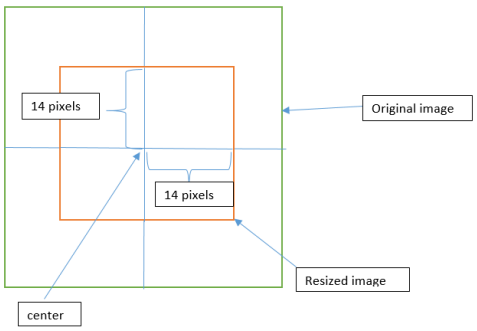

Figure 6.11: Resizing from center

We tested the basic approach on both image scaling schemes. We used a training set holding 3000 images and a test set with 200 images per category (interphase and metaphase), hence amounting to a total of 400 images. We used the standard normalization scheme proposed in section 5.2.1 since we never observed the exploding activations issue in this case study. The neural network we used had the structure (1,12). In addition, we selected the Xavier initialization as initialization scheme, Conopt as solver, and the sigmoid as activation function . Using the first scaling scheme we trained the network twice. The test ran for around 10h and the best accuracy rate we observed was roughly 81%. Regarding the second scaling method (resize from center) we trained the network 5 times. It lasted roughly 8h and the best result we observed was an overall success rate of 91.5%. The poorest iteration yielded an overall success rate of around 89%. It is noticeable that the second image scaling scheme is better than the first for it seems faster and yields far more better results. It is actually not surprising since with the second scaling scheme the rescaled images have fewer pixels of

neighboring cells and are therefore cleaner and more optimal for training. Hence, it was set as our default scaling scheme for the state of the cell recognition problem.

6.5.3 Variant 2

We recall that variant 2 is an extension of variant 1 and under given circumstances is selfsame to variant 1. It is mentioned in section 5.2.3 that the approach we use to combine the values of weights and biases yielded by the different batches is approach 1 described in variant 1. It is because we realized that using approach 2 not only yielded very poor results but was also computationally much more expensive. Most of the existing solvers for nonlinear optimization problems return as optimal value a local optimum and not a global. Hence, it is critical to initialize the variables properly [47]. This is actually the case of Conopt. It is therefore very likely that the reason why we did not observe good results with approach 2 though it is theoretically a better approach, resides in the failure to find the right way to initialize the variables in the nonlinear optimization problem spawned by approach 2.

We tested variant 2 on the state of the cell recognition problem. We used Conopt as solver, the Xavier initialization, and the sigmoid as activation function. The neural network we used had a single hidden layer and 12 hidden neurons. The data were normalized with the first normalization scheme (the one proposed in section 5.2.1). Let N_i (loop 2) be the number of times we go through a given batch and N_b (loop 1) the number of batches. The validation set as mentioned in section 5.2.3 held a total of 200 images (100 images per class).

Table 6.10: Variant 2 with batch size 500 on the state of the cell recognition

$N_b = 5$, $N_i = 2$, batch size = 500	
Batch number	Best overall success rate [%]
1	81
2	77
3	80
4	85.5
5	81.5

The entire test ran for roughly 2h30min. Using approach 1 to combine the values of the different weights, we could attain an accuracy of 90.75% on a test set of 400 images (200 images per class). Using the same main setup mentioned previously and changing the batch

size, reducing it to 50 and setting $N_b = 50$, we conducted a second test and the results are represented in figure 6.12.

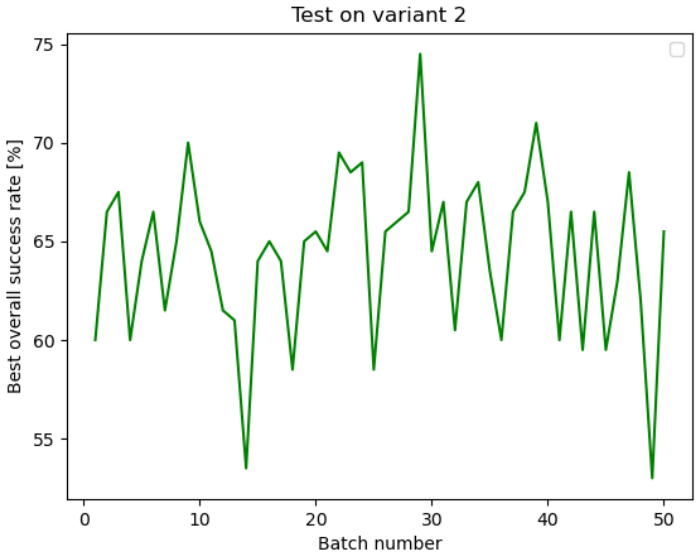

Figure 6.12: Variant 2 with batch size 50 on the state of the cell recognition

The entire test ran for around 3 hours. After combining the values of the different weights, we obtained an overall success rate of 74% on a test set holding 400 images (200 images per class).

On the handwritten digit recognition problem, we used the same setup we described in our first test in this subsection with some slight modifications. Namely the fact that we used the double normalization to ward off the exploding activations issue and our validation set held 400 images (40 per class). The final test set held 1000 images (100 per class).

Table 6.11: Variant 2 with batch size 500 on the handwritten digit recognition problem

Activation function: Sigmoid	
$N_b = 5$, $N_i = 2$, batch size = 500	
Batch number	Best overall success rate [%]
1	10
2	10
3	10
4	10
5	10

The test ran for roughly 155s and the overall success rate on the final test set was 10%. To investigate why the results were so poor we did some further testing. We increased the batch size to 1000 and ran a similar test but the results were still the same (10% on each batch and 10% overall success rate on the final test set). We then reduced the batch size to 50 and the results were still the same. We mentioned in section 5.2.3 (4th adjustment) that we brought some further restrictions on the variables. In our attempt to get the variant to perform better we took out these restrictions and altered also the way we initialized the variables. The random values that are used to initialize the weights on the first batch are exactly the same that are used on every further batch, we do not generate new random values for each batch. With these minor changes we could get variant 2 working on the handwritten digit recognition problem. Hence, using the recently mentioned test setup with a batch size of 2000 we obtained the following results

Table 6.12: Variant 2 with batch size 2000 on the handwritten digit recognition problem

$N_b = 5$, $N_i = 2$, batch size = 2000	
Batch number	Best overall success rate [%]
1	78.75
2	78
3	80.5
4	86.25
5	79.75

The entire test ran for around 24h and on the final test set of 1000 images, we could attain

an overall success rate of 78%. We performed another similar test using this time a batch size of 3000 images and 6 batches. A summary of the test is contained in table 6.13.

Table 6.13: Variant 2 with batch size 3000 on the handwritten digit recognition problem

$N_b = 6$, $N_i = 5$, batch size = 3000	
Total run time: 22h	
Result on final test set: 86.1%	
Batch number	Best overall success rate [%]
1	86.5
2	86.75
3	87
4	89.5
5	88.25
6	87.25

We performed a further test, this time using a batch size of 500 and 10 batches. The test ran for around 3h and we had on the final test set an accuracy of 80.8%.

In the end, it is noteworthy that in order for variant 2 to function properly in some cases we need to bring in some slight adjustments to it which we mentioned in this subsection. It is also remarkable that the accuracy of the network increases with the batch size and thus the computation time. Though we can train the network on more data with this approach we do not observe a better performance than when we use the basic approach. For example, in the last two tests we presented in this subsection regarding the handwritten digit recognition problem, we trained the network respectively on a total of 18000 and 5000 data. The accuracy on a test set of 1000 data was respectively 86.1% and 78%. On the other hand, we could also observe an overall success rate of around 86% using the basic approach and training the network on just 3000 images. We will conclude by saying that this variant left our expectations of attaining better accuracy rates on larger test sets after training the network on a greater amount of data unfulfilled.

6.5.4 Variant 3

We tested the two methods proposed in variant 3, that is to say : $|f_i| \leq \epsilon_1$ for every batch i and $|f_{i+1} - f_i| \leq \epsilon_1$.

We tried both approaches for different values of $\epsilon_1 \in S_\epsilon := \{10^{-3}, 10^{-2}, 10^{-1}, 1, 10^1, 10^2\}$. We tested this variant on the state of the cell recognition problem. We mentioned previously that variant 3 was not a variant on its own. Hence, we combined it with variant 2 setting $N_b = 10$ and $N_i = 1$. We used batches of size 100. We utilized the Xavier initialization, Conopt as solver, and the sigmoid as activation function. The data were normalized with the normalization scheme proposed in section 5.2.1. The tests lasted a few seconds only and we could further observe that the objective function was in the order of thousands for the first method and in the order of millions for the second. The computed weights led to the exploding activations problem since they were quite large. The reason why the results were so poor is that Conopt was unable to find a feasible point during the optimization of the loss function. It actually computed in most of the cases just two iterations.

6.5.5 Variant 4

We trained each of the ten subprograms using Variant 2 with the slight adjustments we mentioned in section 6.5.3. We set $N_b = 1$ and the size of the batch varied from subprogram to subprogram (in between 3200 and 3800). N_i varied from 1 to 5 and the computation time ranged from 2h to 8h. We used a neural network with a single hidden layer holding 12 hidden neurons. We chose Conopt as solver, the sigmoid as activation, and ran the test using the Xavier initialization. The data were normalized using the double normalization scheme presented in section 6.5.1. To obtain the values of the matrix A we used a validation set holding 1000 images (100 images per class).

$A_{i,j}$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$	$j=8$	$j=9$
$i=0$	0.95	0	0.04	0	0	0.03	0.08	0	0	0
$i=1$	0	1	0.08	0.01	0.01	0	0.03	0.06	0.05	0.01
$i=2$	0.03	0.01	0.92	0.07	0.04	0	0.06	0.08	0.13	0.03
$i=3$	0.04	0.07	0.16	0.94	0.05	0.14	0.01	0.19	0.11	0.08
$i=4$	0.03	0.05	0.09	0.07	0.93	0.11	0.10	0.04	0.12	0.18
$i=5$	0.06	0	0.04	0.19	0.08	0.92	0.2	0.07	0.10	0.05
$i=6$	0.04	0	0.06	0.05	0.02	0.02	0.93	0.02	0.01	0
$i=7$	0.05	0	0.05	0.05	0.04	0.05	0.04	0.93	0.05	0.17
$i=8$	0	0.09	0.02	0.06	0.10	0.14	0.09	0.05	0.93	0.07
$i=9$	0.05	0.01	0.03	0.09	0.15	0.13	0.02	0.13	0.16	0.89

We tested the variant on 400 images probing the two methods proposed under 3.3.3 in section 5.2.5. With method 1 we observed an accuracy of 80.75 %. Using method 2 we observed an accuracy of 79.5%. We recall that method 2 uses random numbers, it entails therefore that we get a different test accuracy each time we run the same test. The value we reported is the best accuracy we observed after running the test around 5 times.

It is noteworthy that this variant is computationally very expensive and in terms of accuracy unimpressive. We observed far much better results (93.25%) using the basic approach and a training set of just 3000 data. A reason behind the poor performance of the model is the fact that the decision-making scheme we presented in section 5.2.5 is based on the likelihood of events. Unfortunately, the fact that an event has a great likelihood does not mean it necessarily occurs. Hence, many extra errors flow into the model.

6.6 Comparing the AMPL approach with the SGD approach

In this subsection, we compare the performance of the AMPL and the SGD approach on the three different case studies we handled so far in our work. We used a Python implementation of the SGD approach, precisely with libraries as Keras and TensorFlow. We strove to make the comparison as fair as possible. Hence, we did not utilize many extra features python offers like data augmentation, Max-Pooling [48], dropout [49], convolutional neural networks [50]. In all the tests we present in this subsection, let it be for the SGD approach or AMPL approach, we always used as activation function the sigmoid and as initialization scheme the Xavier initialization. The loss function we used was always the mean squared error. Regarding the AMPL approach, we always used the basic approach and Conopt as solver.

Concerning the polygon recognition problem, we trained the network on 540 images and tested it on 141 images. For both approaches, we used the data normalization scheme proposed in section 5.2.1. Furthermore, we made use of a neural network with 1 hidden layer and 10 hidden neurons. Concerning the SGD approach, we set the batch size to 200 and the number of epochs to 1000. The test ran for 126s with the AMPL approach and we could observe an overall success rate of 98.58%. Regarding the SGD approach, the test ran for 324s and we could also observe an overall success rate of 98.58%.

On the handwritten digit recognition problem, we trained the network on 3000 images and tested it on 400. We used the normalization scheme proposed in section 5.2.1 for the SGD approach. For the AMPL approach, we used the double normalization scheme proposed in

section 6.5.1. In addition, we utilized a neural network with 1 hidden layer and 12 hidden neurons. Regarding the AMPL approach, we repeated the test 5 times. It all lasted around 10h and the best overall success rate we could observe was 93.25%. Concerning the SGD approach, we used a batch size of 200 and tested several numbers of epochs ranging from 100 to 500. The best result we observed was for a number of epochs equal to 200. The test lasted 14s and the overall success rate was 89%.

Regarding the state of the cell recognition problem, We trained the network on 3000 images and tested it on 400. We used the first normalization scheme (the one proposed in section 5.2.1.) for both approaches. Furthermore, we used a neural network with the structure (1,12). Using the AMPL approach we repeated the test 5 times. It lasted around 8h and the best overall success rate was 91.5%. Regarding the SGD approach, we used a batch size of 200 and tested a various number of epochs ranging from 100 to 500. The best result we observed was for 400 epochs. The test lasted 41s and the overall success rate was 84%.

From what we reported previously, it should be noted that under the same conditions the AMPL approach performs in terms of accuracy as well as the SGD approach and sometimes even better. There is nevertheless a chasm in between the computation speed of both approaches. The explanation to this is onefold. We mentioned in chapter 4 that the algorithm Conopt uses is gradient-based. Hence, when we train the network on n images Conopt computes the gradient at every iteration (step) over the entire n images whereas the SGD computes the gradient just over a single randomly chosen image from the n images [51], reason for which it is far much faster compared to Conopt.

7 Summary and conclusion

The objective of this work was to investigate another scheme that we called AMPL approach for the solving of machine learning problems. It was implemented in a software called AMPL. Hence the name AMPL approach. In the first part of this work, we highlighted the theoretical disparities between the AMPL approach and the SGD approach in pellucidly presenting for each scheme the theory that governs its functioning. In a further step, we showed practically through the various models, variants, and techniques we presented how the solvers in AMPL can be used to tackle machine learning challenges, which in the context of this work were: the polygon, state of the cell, and handwritten digit recognition problem. After presenting the AMPL approach and practical implementations thereof, we dived into a series of experiments, in which our first aim was to gain some insight on key thematics like the proper weight initialization scheme, the right solver, and activation function. In the end, we found out that random weight initialization schemes like the Xavier initialization or the He initialization lead to far much better results than what we called constant weight initialization. Moreover, we also noticed that Conopt stood out from the other solvers due to its outstanding performance in terms of accuracy of the neural network and computation time. The only noteworthy flaw of Conopt is the fact that it cannot function with the ReLU as activation function. Regarding the activation functions we probed, we noticed that on little scale problems like the polygon recognition problem, in terms of accuracy there is no great discrepancy between them, but in terms of computation time, we heeded that the ReLU lagged behind the other activation functions. Moreover, our attention was drawn to the fact that the several exponential activation functions (swish, sigmoid, tanh) we scrutinized in our work were not without liabilities. They often tend to lead to what we called the exploding activations problem which we could notwithstanding tackle applying a double normalization to the data. Though the solution is not flawless as it is unsuccessful sometimes, it greatly helps nevertheless quite often to forestall the problem without any tradeoff regarding the accuracy of the network.

After this first wave of tests, we went on to testing our different models (variants). We noticed that though the basic approach has a constraint on the amount of data we can train the network on, it performs nevertheless as well as variant 2 with which we can train the network on much more data. After taking a deep look into our models and gaining much insight

into how they work, we put our models on trial with the mainstream method (SGD). We paid heed that though our models perform in terms of accuracy as well as the SGD method and sometimes slightly better they are nevertheless very far from rivalling the outstanding computation speed of the SGD method, which is more than a 1000 times faster.

It is quite fascinating to see that we could attain such interesting performances in terms of accuracy with the AMPL approach without utilizing miscellaneous intricate schemes like convolution neural networks, the combination of different activation functions in the same network, max-pooling, and dropout which are often used along with the SGD method. This actually brings a glimpse of hope in the possibility of this approach challenging the mainstream approach. Notwithstanding, it will still require that much research be done on how to drastically reduce the computation time of the solvers that AMPL offers such as Conopt when it comes to solving machine learning challenges. This will ease the finding of better schemes to train neural networks effectively on larger amounts of data. Then the implementation of convolutional neural networks in order to solve even more interesting machine learning challenges shall have no hurdle on its way.

Bibliography

[1] E. K. Chong and S. H. Zak, *An introduction to optimization.* John Wiley & Sons, 2004, pp. 248-249.

[2] F. Chen, N. Chen, H. Mao, and H. Hu, "Assessing four neural networks on handwritten digit recognition dataset (MNIST)," *arXiv preprint arXiv:1811.08278*, 2018.

[3] A. F. Agarap, "Deep learning using rectified linear units (ReLU)," *arXiv preprint arXiv:1803.08375*, 2018.

[4] G. Castaneda, P. Morris, and T. M. Khoshgoftaar, "Evaluation of maxout activations in deep learning across several big data domains," *Journal of Big Data*, vol. 6, no. 1, p. 72, 2019.

[5] B. Harshanand and A. K. Sangaiah, "Comprehensive analysis of deep learning methodology in classification of leukocytes and enhancement using swish activation units," *Mobile networks and applications*, pp. 1–19, 2020.

[6] Arunava, "Derivative of the sigmoid function." `https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e`. Accessed: 05-01-2021.

[7] G. Rebala, A. Ravi, and S. Churiwala, *An introduction to machine learning.* Springer, 2019, pp. 1-2.

[8] C. F. Higham and D. J. Higham, "Deep learning: An introduction for applied mathematicians," *SIAM Review*, vol. 61, no. 4, pp. 860–891, 2019.

[9] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.

[10] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms.* Cambridge University Press, 2014, pp. 184-200.

[11] A. Consulting and Development, "Using Conopt with AMPL." `https://ampl.com/SOLVERS/conopt3.pdf`. Accessed: 19-10-2020.

[12] A. Drud, "Conopt: A GRG code for large sparse dynamic nonlinear optimization problems," *Mathematical programming*, vol. 31, no. 2, pp. 153–191, 1985.

[13] L. S. Lasdon, R. L. Fox, and M. W. Ratner, "Nonlinear optimization using the generalized reduced gradient method," *Revue française d'automatique, informatique, recherche opérationnelle. Recherche opérationnelle*, vol. 8, no. V3, pp. 73–103, 1974.

[14] R. J. Vanderbei, *Linear programming: foundations and extensions*, vol. 285. Springer Nature, pp. 12-23, 2020.

[15] O. Rio Branco de Oliveira, "The implicit and the inverse function theorems: easy proofs," *arXiv*, pp. arXiv–1212, 2012.

[16] D. G. Luenberger, *Introduction to linear and nonlinear programming*, vol. 28. Addison-Wesley Reading, MA, 1973.

[17] M. French, *Fundamentals of Optimization: Methods, Minimum Principles, and Applications for Making Things Better*. Springer, pp. 143-157, 2018.

[18] studyflix, "Mitose." `http:https://studyflix.de/biologie/mitose-1807`. Accessed: 28-12-2020.

[19] Wikipedia contributors, "Cross entropy — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Cross_entropy&oldid=993705172`, 2020. [Online; accessed 31-December-2020].

[20] T. Rashid, *Neuronale Netze selbst programmieren: ein verständlicher Einstieg mit Python*. O'Reilly, 2017.

[21] A. Rakitianskaia and A. Engelbrecht, "Measuring saturation in neural networks," in *2015 IEEE Symposium Series on Computational Intelligence*, pp. 1423–1430, IEEE, 2015.

[22] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial*

intelligence and statistics, pp. 249–256, 2010.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[24] H. Li, M. Krcek, and G. Perin, "A comparison of weight initializers in deep learning-based side-channel analysis,"

[25] Z. C. Lipton, "Stuck in a what? adventures in weight space," *arXiv preprint arXiv:1602.07320*, 2016, p. 1.

[26] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.

[27] N. Wanas, G. Auda, M. S. Kamel, and F. Karray, "On the optimal number of hidden nodes in a neural network," in *Conference Proceedings. IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No. 98TH8341)*, vol. 2, pp. 918–921, IEEE, 1998.

[28] W. Zou, Y. Li, and A. Tang, "Effects of the number of hidden nodes used in a structured-based neural network on the reliability of image classification," *Neural Computing and Applications*, vol. 18, no. 3, pp. 249–260, 2009.

[29] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *ICML*, 2010.

[30] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378*, 2018.

[31] T. Herawan, R. Ghazali, and M. M. Deris, "Recent advances on soft computing and data mining," *Sl: Springer*, 2017.

[32] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[33] W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, *Engineering*

in Dependability of Computer Systems and Networks: Proceedings of the Fourteenth International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, July 1–5, 2019, Brunów, Poland, vol. 987. Springer, pp. 499-500, 2019.

[34] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International Workshop on Artificial Neural Networks*, pp. 195–201, Springer, 1995.

[35] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[36] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR, abs/1211.5063*, vol. 2, p. 417, 2012.

[37] J. Brownlee, "A gentle introduction to exploding gradients in neural networks." `https://machinelearningmastery.com/exploding-gradients-in-neural-networks/#:~:text=Exploding%20gradients%20are%20a%20problem,learn%20from%20your%20training%20data`. Accessed: 28-10-2020.

[38] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017.

[39] P. Seetharaman, G. Wichern, B. Pardo, and J. Le Roux, "Autoclip: Adaptive gradient clipping for source separation networks," in *2020 IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6, IEEE, 2020.

[40] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, pp. 1310–1318, 2013.

[41] T. Van Laarhoven, "L2 regularization versus batch and weight normalization," *arXiv preprint arXiv:1706.05350*, 2017.

[42] L. Oneto, S. Ridella, and D. Anguita, "Tikhonov, ivanov and morozov regularization for support vector machine learning," *Machine Learning*, vol. 103, no. 1, pp. 103–136, 2016.

[43] M. Wang, Y. Sun, E. Yang, and K. Song, "Consistent model combination of lasso

via regularization path," in *Chinese Conference on Pattern Recognition*, pp. 551–562, Springer, 2016.

[44] R. Tibshirani, "Model selection and validation 1: Cross-validation." `http://www.stat.cmu.edu/~ryantibs/datamining/lectures/18-val1.pdf`. Accessed: 30-10-2020.

[45] J. Brownlee, "How to avoid exploding gradient with gradient clipping." `https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/#:~:text=Exploding%20gradients%20can%20be%20avoided,number%20of%20input%20time%20steps`. Accessed: 10-11-2020.

[46] A. Drud, "Using Conopt with AMPL." `https://www.gams.com/latest/docs/S_CONOPT.html`. Accessed: 28-10-2020.

[47] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, and J. D. Siirola, *Pyomo-optimization modeling in python*, vol. 67. Springer, pp. 111-112, 2017.

[48] Z. Qian, L. Yawei, Z. Mengyu, Y. Yuliang, X. Ling, X. Chunyu, and L. Lin, "Max-pooling convolutional neural network for chinese digital gesture recognition," in *Information Technology and Intelligent Transportation Systems*, pp. 79–89, Springer, 2017.

[49] C. Garbin, X. Zhu, and O. Marques, "Dropout vs. batch normalization: an empirical study of their impact to deep learning," *Multimedia Tools and Applications*, pp. 1–39, 2020.

[50] H. Wu and X. Gu, "Max-pooling dropout for regularization of convolutional neural networks," in *International Conference on Neural Information Processing*, pp. 46–54, Springer, 2015.

[51] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.

IMPRESSUM

Brandenburgische Technische Universität Cottbus-Senftenberg
Fakultät 1 | MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik
Institut für Mathematik
Platz der Deutschen Einheit 1
D-03046 Cottbus

Professur für Ingenieurmathematik und Numerik der Optimierung
Professor Dr. rer. nat. Armin Fügenschuh

E fuegenschuh@b-tu.de
T +49 (0)355 69 3127
F +49 (0)355 69 2307

Cottbus Mathematical Preprints (COMP), ISSN (Print) 2627-4019
Cottbus Mathematical Preprints (COMP), ISSN (Online) 2627-6100

www.b-tu.de/cottbus-mathematical-preprints
cottbus-mathematical-preprints@b-tu.de
doi.org/10.26127/btuopen-5404