An Introduction to Deep Learning: How Neural Networks Can Be Used to Recognize Handwritten Digits

A PREPRINT

Anton Kratz [*]
contact@antontx.de

July 16, 2023

Abstract

Deep learning offers us an alternative way to approach problem-solving that differs from traditional algorithmic methods. Instead of telling the algorithm what to do, one can just show it lots of examples. This can significantly reduce development time, as it eliminates the need for complex logical rules and patterns. Furthermore, it makes it easier to scale and customize the product because just modifying the training data enables a neural network to learn and adapt new behaviors efficiently. Deep neural networks are biologically inspired computational systems composed of multiple layers that process some input data based on internal parameters. Those parameters are learned and adjusted over time using the *backpropagation* algorithm. Furthermore, the use of statistical instead of logical methods to analyze and solve problems makes it possible to quickly solve tasks that would have required significant programming effort using conventional approaches like speech, object, or handwriting recognition, which is what I will focus on in this paper.

Our approach involves revisiting the theoretical background of deep learning and its major components in part 1, followed by the application of these concepts to model and train a deep neural network capable of recognizing handwritten digits from scratch in part 2.

[*]more on `antontx.de`

Contents

1 What is a neural network and how does it work?

1.1 Biological motivation

In order to develop an intuitive understanding of artificial neural networks, it is beneficial to briefly explore the biological background of neural networks and the learning processes that occur in the human brain. This exploration helps to establish a direct connection between the biological foundation and the understanding of artificial neural networks (ANN's).

The human brain consists of approximately 86 billion neurons which are densely connected to thousands of other neurons. These connections, called synapses, get strengthened when signals are repeatedly transmitted between them. This long-lasting amplification of synaptic transmissions is called *long-term potentiation* (LTP), and is the cellular and molecular foundation of learning. Through continuous repetition of experiences and actions certain networking patterns are formed and new synapses are created or intensified while ones that were used less get dismantled over time. For example, if someone learns a new skill, such as playing the piano, certain synapses in their brain will become stronger and more efficient, allowing them to play the piano better over time. This ability to form new connections and adjust existing connections based on their importance is called *neuroplasticity*. [Groß et al. 2021]

Especially the idea of weighted connections between neurons in a network allows for the creation of complex artificial neural networks that can learn from data, recognize patterns, and make predictions, making it a fundamental concept for artificial intelligence. For example, a deep neural network used for image recognition might have hundreds of thousands or even millions of neurons arranged in layers, with each neuron connected to many others through weighted connections. By adjusting these weights during training, the network can learn to recognize patterns in images, such as identifying faces or objects. While modern ANN's are inspired by biological neural networks, it is important to note that they only loosely model the neurons in brains and do not completely mirror biological learning processes [Mitchell 1997, p. 82]. ANN's for instance have fixed structures instead of being dynamic and do often rely on supervised learning [Section 1.3 & 2.3]. Furthermore, the actual process of adjusting the weights also differs but we will focus more on that in sections 1.4 & 1.5.

1.2 The single-layer perceptron

The perceptron Before dealing with more complex artificial neural networks, it makes sense to begin by focusing on a single artificial neuron. Generally speaking, a neuron takes a set of values as input and produces a single output based on those inputs. These inputs are also called features and are numerical values representing the properties of the data in a way that a learning algorithm can process and interpret them. One of the simplest neural networks is the single-layer perceptron (SLP), introduced by McCulloch and Pitts (1943) since it only consists of a single neuron. The SLP is a binary classifier that can 100% accurately determine whether an input belongs to one specific class or not, provided that the dataset is linearly separable [Minsky and Papert 1969].

The process starts with calculating the weighted sum z of all the inputs, as shown in Equation 1. This is done by adding up each input x_i multiplied with its corresponding weight w_i.

$$z = \sum_{i=1}^{n} w_i x_i \tag{1}$$

This sum is then passed through an activation function to obtain the output of the neuron. The single-layer perceptron utilizes the Heaviside step function H as its activation function, where an output of 1 is produced if the weighted sum is larger than 0, and 0 otherwise (Equation 2). These output values can be interpreted as the probability of the input data belonging to the specified class, where 1 means that the perceptron is 100% confident that the input data belongs to the class while 0 signifies the opposite.

$$H(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{2}$$

One may also introduce a bias b by adding it to the weighted sum, effectively altering the threshold of the Heaviside Step function from 0 to b which makes the bias of the neuron another adjustable parameter of the network.

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{3}$$

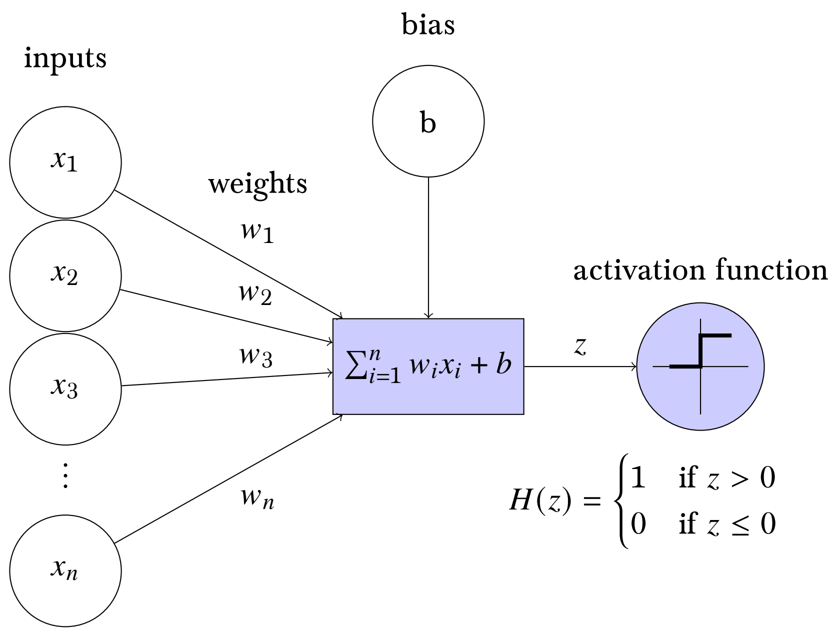

bias
b
inputs
x_1
x_2
x_3
$\vdots$
x_n
weights
w_1
w_2
w_3
w_n
$\sum_{i=1}^{n} w_i x_i + b$
z
activation function
$H(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$

Figure 1: Visual representation of a single-layer perceptron

By representing inputs and weights as vectors, we can conveniently compute the weighted sum using the dot product (Equation 4). The dot product is an operation that calculates the sum of the element-wise products of two vectors. It is an efficient method for computing the weighted inputs because modern libraries offer highly optimized matrix multiplication functions, that are easier to implement and faster to compute than calculating the sum iteratively. This will especially come in handy when dealing with neural networks that consist of multiple neurons and layers.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \qquad z = \sum_{i=1}^{n} w_i x_i + b = x \cdot w + b \tag{4}$$

Our perceptron essentially simply determines whether the input data satisfies a specific inequality. By feeding the weighted sum into the Heaviside step function, we obtain the following inequality: $x \cdot w + b > 0$. Accordingly the values assigned to the weights and bias precisely define an inequality that represents the perceptron's classification criterion, allowing it to distinguish between the different classes of input data.

The Perceptron Training Rule Single-layer perceptrons are trained using a simple algorithm called the Perceptron Training Rule. Its goal is to adjust the weights w_i of the perceptron to correctly classify input data. The rule updates the weights based on the error between the desired output $y(x)$ and the actual output o, as shown in Equations 5 & 6. The learning rate η controls the magnitude of weight updates during training and will be further explored in the section on training a multilayer perceptron. At its heart, this training rule involves updating the weights based on the value ∇w_i, which guides the necessary weight adjustments based on the difference between the expected and predicted output [Mitchell 1997, pp. 88–89]. It is important to remember this fundamental concept as we delve into more complex neural networks in subsequent chapters.

$$\nabla w_i = \eta(y(x) - o)x_i \tag{5}$$

$$w_i \longrightarrow w'_i = w_i + \nabla w_i \tag{6}$$

1.3 Artificial neural networks

Now that we have an understanding of how a single-layer perceptron works, we can expand upon this basic idea to explain the architecture of the multilayer perceptron (MLP). The MLP is a feed-forward neural network consisting of multiple static layers, each comprising several neurons. In a feed-forward network, the information flows in one direction without any loops [Nielsen 2015, p. 12].

There are three different types of layers in a deep neural network: First of all, we have the input layer, which is going to represent the features that we pass into our network without modification. This is technically the same as the input vector we had in our single-layer perceptron. The single output neuron we previously had is replaced with an output layer consisting of multiple output neurons with each neuron representing the probability of a specific output. Between the input and the output layer, we have one or more hidden layers that transform the representations of our data into the output layer by layer [Li, Zhao, and Scheidegger 2020, Nielsen 2015, p. 11].
Each neuron is directly connected to each neuron in the previous layer as well as in the next layer by a weighted connection/weight, such that the weighted outputs of each layer with the exception of the output layer serve as inputs for the following layer [Nielsen 2015, p. 12].

Still storing all network weights in vector form would pose significant challenges and lack intuitiveness since it is just not intuitive to store weights connecting each neuron from layer $i - 1$ to each neuron in layer i in a single column. A plausible and more beautiful alternative is to construct separate weight matrices w_i, that represent all inter-layer weights between layers $i - 1$ and i [Nielsen 2015, p. 25]. In addition to improving the intuitiveness and conciseness of the weight representation, the use of matrices allows for fast matrix multiplications in a single step, thereby enabling efficient computation of weighted sums as inputs to neurons in layer i.

In fact, each row of the weight matrix w_i precisely denotes the weights connecting a single neuron in layer $i - 1$ to all the neurons in layer i, while each column of w_i represents the weights connecting all the neurons in layer $i - 1$ to a single neuron in layer i.

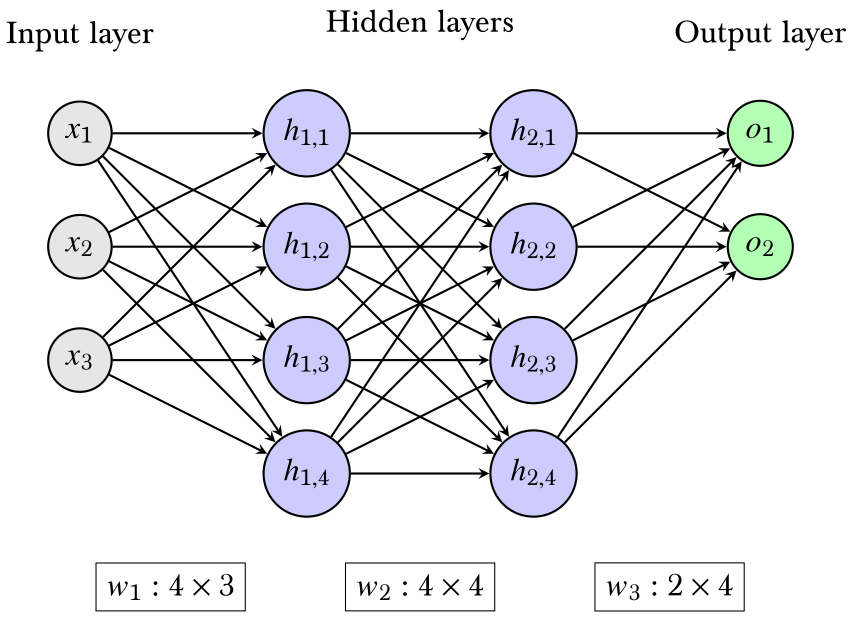


Figure 2: Visual representation of an MLP with two hidden layers. The expressions in the rectangles represent the dimensions of the weight matrices.

Let's for example consider a neural network with an input layer of size n_0 and a hidden layer of size n_1. The weight matrix w_1 connecting the input and hidden layers would have the dimensions $n_1 \times n_0$, where each column corresponds to the weights that connect a single neuron in the input layer to all the neurons in the second layer.
This is great because the calculation of the dot product between the outputs in layer i, called activations (a_i), and w_{i+1} will hence directly return a vector that stores the weighted sums for each neuron in layer $i + 1$ (Equation 7). To then obtain the actual outputs of the neurons in layer i, we can simply pass the weighted sums through our activation function f (Equation 8).

$$z_{i+1} = a_i \cdot w_{i+1} \tag{7}$$

$$a_i = f(z_i) \tag{8}$$

Note that we will set the activations of the input layer to the inputs x of the network.

$$a_0 := x \tag{9}$$

We will use the same method to store the biases for each neuron in a layer i: instead of storing a single value for each neuron, we will create multiple bias vectors for the different layers, where each bias vector b_i stores the biases of all neurons in the layer i. This is a very convenient technique because we can add the bias vector to the weighted sum without worrying about matrix dimension problems (Equation 10). The underlying reason for this is that both the activations a_i and the biases b_i will always be a column vector of size n_i, where n_i is the number of neurons in the layer i.

$$z_i = a_{i-1} \cdot w_i + b_i \tag{10}$$

Given knowledge of the activations in the input layer and the methodology for calculating activations in layer $i + 1$ based on the activations of the neurons in layer i, it is possible to determine the activations of all layers within the network from any input layer. This process is commonly referred to as "feedforward" or "forward propagation" since the data is propagated through the network from the input layer to the output layer.

Algorithm 1 performs a forward pass through a neural network, computing layer activations by calculating weighted sums of inputs and applying an activation function. The algorithm determines the final output of layer $L - 1$, but it can also provide the activations a and weighted sums z of all layers in the network. Symbols, such as weights w, biases b, activation function f, the total number of layers L, or the learning rate η, are considered attributes of the network and are assumed to be defined for all algorithms in this paper. This approach ensures a concise presentation without cluttering the **Require** section with redundant symbols that are already implied as part of the network's attributes.

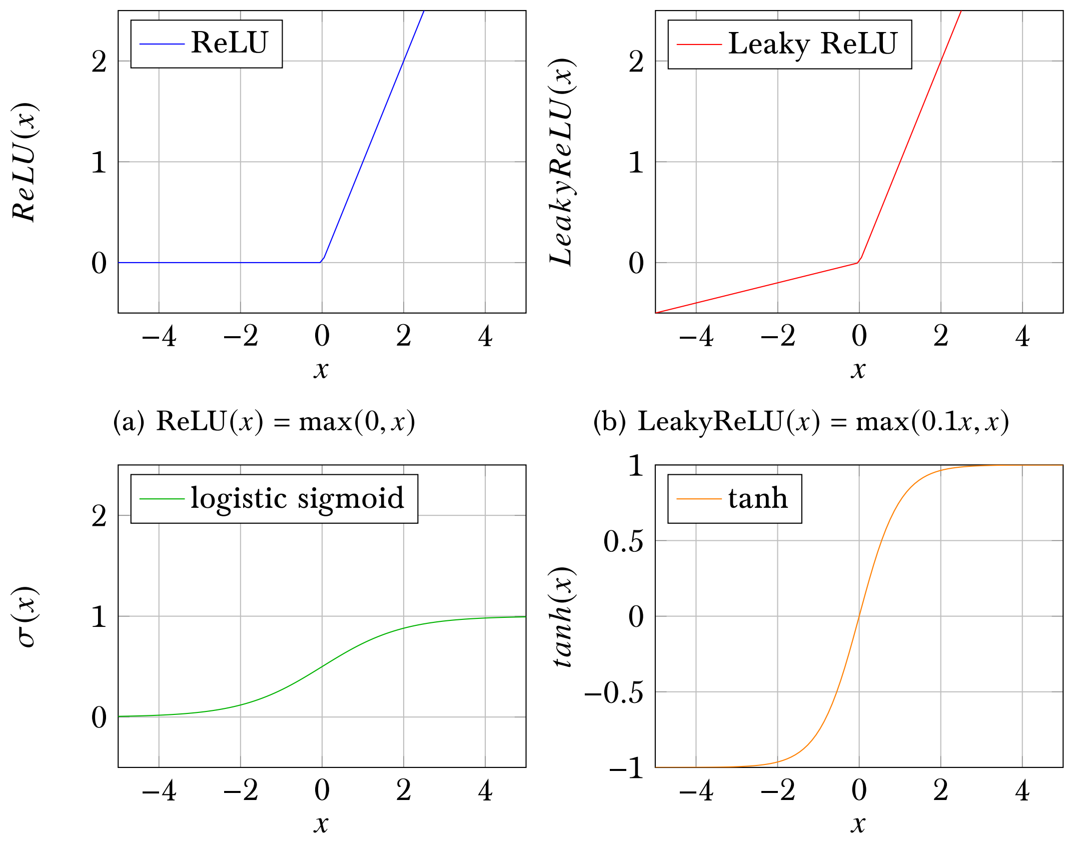
Algorithm 1 Feedforward

```
Require: $x$: input / features
Ensure: $a_{L-1}$: output layer activations
  $a_0 \leftarrow x$
  for $i = 1$ to $L - 1$ do
      $z_i \leftarrow w_i \cdot a_{i-1} + b_i$
      $a_i \leftarrow f(z_i)$
  end for
  return $a, z$
```

So far, we have simply accepted the activation function f as necessary, but have not considered its use, its necessity, or its properties. Therefore, it would be understandable to ask why the activations of each layer are not simply multiplied by the weight matrices without applying a function after carrying out each matrix multiplication. The underlying idea is that activation functions introduce non-linearity into the model, allowing it to model complex non-linear relationships and patterns in the data [Goodfellow, Bengio, and Courville 2016, pp. 165–166, S. Amidi and A. Amidi 2018].
It is therefore crucial that the function applied is a non-linear function, as this allows the model to approximate almost any function, whereas a model using linear functions

as activation functions would only be able to approximate linear functions. In other words, if a neural network did not have a non-linear activation function, it would behave akin to a single-layer perceptron. This is because combining multiple layers in the absence of non-linearity would always result in a simple linear transformation of the input, something that could be achieved without any hidden layer. Another major problem is that the backpropagation algorithm in section 1.5 requires us to use the first derivative of the activation function to compute the gradient of the cost function with respect to the weights and biases. However, when a linear activation function is used, the first derivative of the function is always a constant value, meaning that the gradient has the same magnitude and direction for all inputs, so the actual input of the neurons does not affect the updating of the weights at all [Lederer 2021, p. 3].

Popular activation functions include sigmoid functions like the *logistic* or *tanh* function but also piecewise-linear functions like the *ReLu* or *LeakyReLU* function (Figure 3).



(a) $\text{ReLU}(x) = \max(0, x)$ (b) $\text{LeakyReLU}(x) = \max(0.1x, x)$

(c) $\sigma(x) = 1/(1 + e^{-x})$ (d) $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$

Figure 3: Plots of various activation functions: ReLU, LeakyReLU, logistic sigmoid, and tanh.

Before proceeding further, it may be beneficial to summarize and reinforce our understanding by recapping all the significant points we have covered so far.
A Multilayer Perceptron consists of different layers containing different numbers of neurons. The first layer is the so-called input layer, which corresponds to the features, i.e. the input data. The last layer is the output layer, which represents the probabilities of the different possible outcomes. Between the input and the output layer, there are

usually one or two hidden layers, which transform the input data in a way that makes it easier to divide it into the output classes. From a mathematical point of view, we always have a single vector that is transformed into the output probability vector in a fixed order by relatively simple mathematical operations, such as multiplication with learned weight matrices or applying an activation function element-wise. As a result, our neural network is even deterministic since the parameters and therefore also the mathematical operations remain the same every time we feed data through the network. It should now be understood how the actual classification process of a neural network works on a mathematical level, i.e. how an output can be formed from a set of features based on the parameters of the network. The neural network itself is defined by its parameters, the weights and biases, which are learned by the *backpropagation* algorithm.

1.4 Gradient based learning

In this chapter, we will look at the training process of a simple deep neural network. In the following sections, we will refer to the features of a single training example as x, also referred to as the input, and the corresponding label as $y(x)$, also referred to as the expected output. Our primary goal is to find an algorithm that can accurately approximate $y(x)$ based on x for all training examples. In other words, our network aims to approximate a function y that maps input data x to their corresponding output values $y(x)$.

To improve the performance of our network, it is essential to evaluate its performance accurately. For this purpose, we will use a cost function that measures how poorly the model predicts the outputs. In this context and for the purpose of this paper, "model" can be used synonymously with the term 'network'. In General, a model is a mathematical construct that learns from input data to make predictions [Google 2023]. The cost is minimal when our network accurately approximates $y(x)$, but maximal when it fails to find the correct relationships between input and output. This allows us to define our training objective more precisely: We want to minimize the cost over all training examples. In other words, we need to find the values for our weights and biases that will produce the lowest possible cost [Yann LeCun, Bengio, and G. Hinton 2015, p. 1].

A possibly cost or "loss" function for a single training example would be the quadratic cost function, which calculates the squared difference between the expected output $y(x)$ and the predicted output a_{L-1} (Equation 11). Note that a_{L-1} refers to the output neuron activations of the input x. This function serves as an effective cost function because it quantifies the difference between the desired and actual output, and thus provides insight into the accuracy of the predictions. The decision to square the difference is particularly beneficial for two reasons. First, it magnifies incorrect results, thereby assigning greater weight to larger errors. Secondly, it ensures that only positive costs are obtained. This is crucial because we have defined our cost function so that higher cost corresponds to poorer network performance. If a large negative cost was possible, it would falsely indicate that the network was performing well despite a significant error. The output of the function depends on the parameters of the network and the input x. For simplicity, it makes sense to only note the dependency on the weights w and biases b since these are the parameters that can be optimized during the training process to minimize the cost function [Nielsen 2015, p. 16].

$$C(w, b) = (y(x) - a_{L-1})^2 \tag{11}$$

Knowing how good the network performs on a single training example is not meaningful about its general performance at all. Instead, we need to evaluate the average cost, which is commonly referred to as the *mean squared error* (MSE), over all the training examples since this provides a more comprehensive measure of the network's performance and its ability to classify unseen data [Equation 12].

$$C_{\text{avg}}(w, b) = \frac{1}{n} \sum_x (y(x) - a_{L-1})^2 \tag{12}$$

where:

- n: the total number of inputs x

To make derivations cleaner and easier, we will divide the sum by $2n$ instead of n so that the 2 in the power gets canceled with the $\frac{1}{2}$ multiplier (Equation 13). This won't affect the learning process since minimizing is unaffected by constants, the parameters that resulted in the smallest cost before will still produce the smallest possible cost no matter what constant our cost is multiplied with.

$$C_{\text{avg}}(w, b) = \frac{1}{2n} \sum_x (y(x) - a_{L-1})^2 \tag{13}$$

One can imagine the cost function as a multidimensional landscape that represents the cost of the network depending on the network's parameters [Yann LeCun, Bengio, and G. Hinton 2015, p. 2]. While minimizing a function might sound pretty simple at first glance, it turns out that minimizing a complex function depending on thousands of variables can not be done efficiently using regular algebraic methods.
A popular algorithm to minimize such a function is called *gradient descent*: It is an iterative method that utilizes the gradients of the cost function with respect to the parameters of the network to update the parameters in the opposite direction of their gradient (Equations 14 & 15)[Y. LeCun et al. 1998]. This update rule is based on the intuition that moving in the opposite direction of the gradient will bring us closer to the minimum of the cost function, which makes perfect sense because the gradients point in the direction of the steepest increase of the cost function, and moving in the opposite direction will therefore result in a decrease [Yann LeCun, Bengio, and G. Hinton 2015, pp. 1–2].

In the past, it was believed that simple gradient descent could become trapped in poor local minima, which are weight configurations where the average error cannot be further reduced by making small changes. This was a concern because the cost function in machine learning can have multiple minima, and getting stuck in a suboptimal solution could hinder the optimization process. However, with large networks, this concern is rarely an issue. Recent theoretical and empirical findings suggest that the landscape is primarily composed of saddle points, where the gradient is zero, and most of these points have similar objective function values. Therefore, the specific saddle point where the algorithm converges is of less significance [Yann LeCun, Bengio, and G. Hinton 2015, p. 3].

$$w \rightarrow w' = w - \eta \nabla w \tag{14}$$

$$b \rightarrow b' = b - \eta \nabla b \tag{15}$$

The learning rate, denoted as η, determines the magnitude of the weight updates during training. It is a hyperparameter that needs to be chosen carefully, as setting it too low will result in slow convergence while setting it too high can lead to overshooting the minimum [Kriesel 2006].
A *hyperparameter* is a parameter that is set outside of the learning process itself in contrast to other parameters which are derived via training [Goodfellow, Bengio, and Courville 2016, p. 96].

During the training of a neural network, the entire training set is typically processed multiple times. A complete pass over the entire training set, where each example is processed once, is commonly referred to as an *epoch*. "There are various approaches to utilize gradient descent for training a model. In a single epoch, one option is to update the network's parameters after processing each training example. Alternatively, one can compute the average gradient over a subset of examples and adjust the weights and biases accordingly. This latter approach is known as mini-batch gradient descent, which we will predominantly employ in training our model throughout this paper [G. Hinton, Srivastava, and Swersky 2012].

Algorithm 2 presents a gradient-based training algorithm with a customizable batch size for neural networks.

Algorithm 2 Gradient Descent

```
Require:
  $S$: batch size
  training data: set of input-output pairs $(x, y(x))$ used for model training.
Ensure: optimized model parameters $w$ and $b$
  randomly initialize weights $w$ and biases $b$
  for each epoch do
      shuffle and separate the training data into random batches of size $S$
      for each batch do
          for each training example $(x, y(x))$ do
              $\nabla b_i, \nabla w_i \leftarrow backprop(x, y(x))$
          end for
          $\nabla w_{\text{avg}} \leftarrow \frac{1}{S} \sum_{i=1}^{S} \nabla w_i$
          $\nabla b_{\text{avg}} \leftarrow \frac{1}{S} \sum_{i=1}^{S} \nabla b_i$
          $w \leftarrow w - \eta \cdot \nabla w_{\text{avg}}$
          $b \leftarrow b - \eta \cdot \nabla b_{\text{avg}}$
      end for
  end for
```

1.5 Backpropagation

The procedure used to compute the gradients of weights and biases is called backpropagation and was popularised by Rumelhart, G. E. Hinton, and Williams (1986). Backpropagation works by propagating the training data forward through the network, as shown in Algorithm 1, to compute the corresponding error. Next, this error is propagated backward through the layers to compute the gradients layer by layer [S. Amidi and A. Amidi 2018].

The intent of backpropagation is to compute the partial derivatives $(\partial C/\partial w)$ and $(\partial C/\partial b)$ of the cost function C with respect to the weights w and biases b in the network. The computation of partial derivatives within backpropagation is crucial as it allows us to determine the local rate of change of the cost function with respect to the weights and biases. These partial derivatives are combined to form a gradient vector, which represents the overall rate of change of the cost function with respect to all parameters in the network [Nielsen 2015, pp. 10, 42–45].

In his book, Nielsen states that backpropagation is based on four fundamental equations, which are all derived from the chain rule (Equations 17 - 20). Two equations calculate the errors δ of the neurons in the output or a hidden layer, and the other two equations calculate the two aforementioned partial derivatives of the cost function [Nielsen 2015, pp. 43–47]. For a comprehensive and comprehensible proof of these equations, I recommend taking a look at Chapter 2.5 of Nielsen's freely available book, "Neural Networks and Deep Learning."

In addition to the usual linear algebraic operations, the backpropagation algorithm also involves the use of the *Hadamard product*. The Hadamard product, denoted by $\odot$, is an element-wise multiplication of two matrices A and B, which have the same dimensions. This operation produces a resulting matrix C, which also has the same dimensions as the input matrices. For example:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \odot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{pmatrix} = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix} \tag{16}$$

Equation 17 computes the error in the output layer δ_{L-1}.

$$\delta_{L-1} = (a_{L-1} - y(x)) \odot f'(z_{L-1}) \tag{17}$$

Equation 18 computes the error in layer l (δ_l), based on the error in layer $l+1$

$$\delta_l = (w_{l+1}^T \cdot \delta_{l+1}) \odot f'(z_l) \tag{18}$$

Equation 19 computes the rates of the change of the cost with respect to the biases in layer l

$$\nabla b_l = \frac{\partial C}{\partial b_l} = \delta_l \tag{19}$$

Equation 20 computes the rates of the change of the cost with respect to the interlayer weights between layer $l-1$ and layer l

$$\nabla w_l = \frac{\partial C}{\partial w_l} = a_{l-1}^T \cdot \delta_l \tag{20}$$

Algorithm 3 outlines the backpropagation algorithm for computing the gradients of the cost function with respect to the weights and biases in a neural network.

Algorithm 3 Backpropagation

```
Require: Input data $x$, expected output $y(x)$
Ensure: Gradients $\nabla b, \nabla w$ of the cost function with respect to biases and weights
  $a, z \leftarrow \mathit{Feedforward}(x, y(x))$
  $\delta_{L-1} = (a_{L-1} - y(x)) \odot f'(z_{L-1})$
  $\nabla b_{L-1} = \delta_{L-1}$
  $\nabla w_{L-1} = a_{L-2} \cdot \delta_L - 1$
  for each layer $l$ from $L-2$ to 1 do
      $\delta_l = (w_{l+1}^T \cdot \delta_{l+1}) \odot f'(z_l)$
      $\nabla b_l = \delta_l$
      $\nabla w_l = a_{l-1} \cdot \delta_l$
  end for
  return $\nabla b, \nabla w$
```

The backpropagation algorithm may seem complex initially, but its intuition becomes clearer when examining the purpose of each component individually. Hence, prioritizing the understanding of the purpose and function of each equation is more crucial at this stage than the full comprehension of how or why the equation works.

The core concept behind the backpropagation procedure is surprisingly straightforward: For every neuron i in layer l, we are calculating some error term $\delta_{l,i}$ that provides information about the extent of the output's inaccuracy. Similar to how biases are handled, we are going to store all the errors in any layer l in a vector, and call this collection of errors in layer l "the error in layer l" denoted as δ_l. Using this error information, we can compute the corresponding weight and bias gradients for the layer $(\nabla w_i \& \nabla b_i)$.

The algorithm begins by performing a forward pass of a single pair of training data through the network to obtain the activations a_l and weighted sums z_l of each layer. This step serves two purposes: firstly, it allows us to determine the error in the last layer, denoted as δ_{L-1}, which is crucial for calculating the gradients in the final layer. Secondly, it ensures that the activations and weighted sums of each layer are available for the subsequent backpropagation process outlined below.

Given that we can compute the error in a layer based on the error in the next layer, knowing the error in the last layer enables us to backpropagate the error layer by layer. This is done in the for loop which starts at the index of the second-to-last layer (layer $L-2$) and goes down to 1. As a keen eye might notice, the for loop is only executed $L-2$ times even though we have L layers in total. Since the weight and bias gradients for the output layer are calculated separately, this leaves us with one iteration off.

However, upon further reflection, this also seems logical, since our input layer has no parameters; the activations a_0 are defined by the features x and therefore do not require the calculation of z_0. The absence of z_0 also explains the absence of w_0 and b_0, which in turn explains why we do not need to calculate gradients for layer 0. Hence, the for loop computes the corresponding errors and gradients for all hidden layers (layer 1 - layer $L-2$).

And this is already it, we start by performing one forward pass to obtain δ_{L-1} using Equation 17 and then propagate the error backward based on Equation 18 to obtain δ_l for every hidden layer. Based on δ_l one can then compute ∇b_i and ∇w_i for every layer using Equations 19 & 20 to compose the weight and bias gradient of the cost function for a single training example.

2 Handwritten Digit Recognition through Neural Networks

In this chapter, we will attempt to apply the techniques and concepts previously explored to build a deep neural network capable of accurately classifying handwritten digits. The whole network will be implemented in Python with NumPy being the only external library used in the implementation. This makes sense because NumPy is the industry standard for scientific computing in Python. The complete source code, along with a brief documentation, is available at `github.com/antontx/NeuralNetwork`.

2.1 The Data

The Dataset To train our network effectively, we will feed it a substantial amount of training examples. Each example will comprise an image of a handwritten digit along with its corresponding label. By exposing the network to numerous examples, it will be able to automatically adjust its parameters over time and learn how to accurately classify digits [Nielsen 2015, pp. 10–11].

We are going to use the *MNIST* database (***M****odified* ***N****ational* ***I****nstitute of* ***S****tandards and* ***T****echnology database*) [Yann LeCun, Cortes, and Burgess 2012] which contains 60.000 labeled 28×28 training images of handwritten digits as well as 10.000 labeled testing images in the same format.

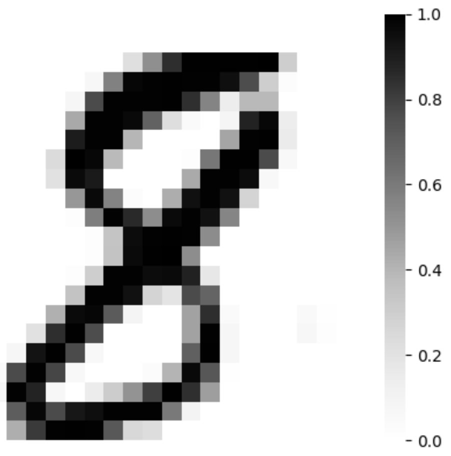


Figure 4: Example image of a handwritten digit 8 from the MNIST training dataset, preprocessed and scaled to greyscale values between 0 and 1.

Transforming the data Before training our neural network, we need to preprocess our data to make it compatible with the network's input and output shapes. Since our network will take 784-dimensional vectors as input (representing the flattened image data) and outputs a 10-dimensional vector representing the predicted probabilities for each digit, we must transform our data accordingly. We flatten the 28×28 arrays of image data into arrays of shape 784×1, and we one-hot encode the labels as arrays of shape 10×1 (where each element is either 0 or 1). Specifically, we set the element corresponding to the true label to 1, and the other elements to 0. This encoding makes it easier to compare the network's predicted probabilities to the true label since both are represented as vectors of size 10. As a further preprocessing step, we scale our features to greyscale values between 0 and 1 by dividing all pixel values by 255. This compresses the pixel values into a range where 1 represents the darkest pixel and 0 represents the brightest pixel (Figure 4). The Reason for this is that the sigmoid function used in the network involves calculating exponential terms, and computing large exponents can be computationally intensive and prone to overflow errors. By scaling the input features to a smaller range, we can reduce the likelihood of encountering numerical instability issues and improve the overall stability of the network during training.

2.2 Architecture

As already indicated, an input layer with 784 nodes is useful, where each node represents a single feature, the brightness of one particular pixel. On the other hand, it is quite obvious that we will have 10 nodes in the output layer since we want to classify our images into one of 10 different classes. Each output node therefore represents the probability that our input belongs to that class.

Determining the optimal number and size of hidden layers in a neural network is not as straightforward as it is for the input/output layers. There is no fixed rule for the hidden architecture, and it often requires experimentation and analysis to find the most effective configuration.

While a neural network with a single hidden layer theoretically has the capacity to learn any pattern, it has been observed that architectures with two hidden layers tend to outperform those with only one hidden layer, especially when dealing with more complex patterns [Heaton 2015, p. 236].

Additionally, experiments have shown that the size of the hidden layers should typically fall between the sizes of the input and output layers. It is often beneficial to gradually decrease the size of the hidden layers from the larger input layer to the smaller output layer. This gradual reduction allows for the transformation and compression of representations as they propagate toward the output activations [Heaton 2015].

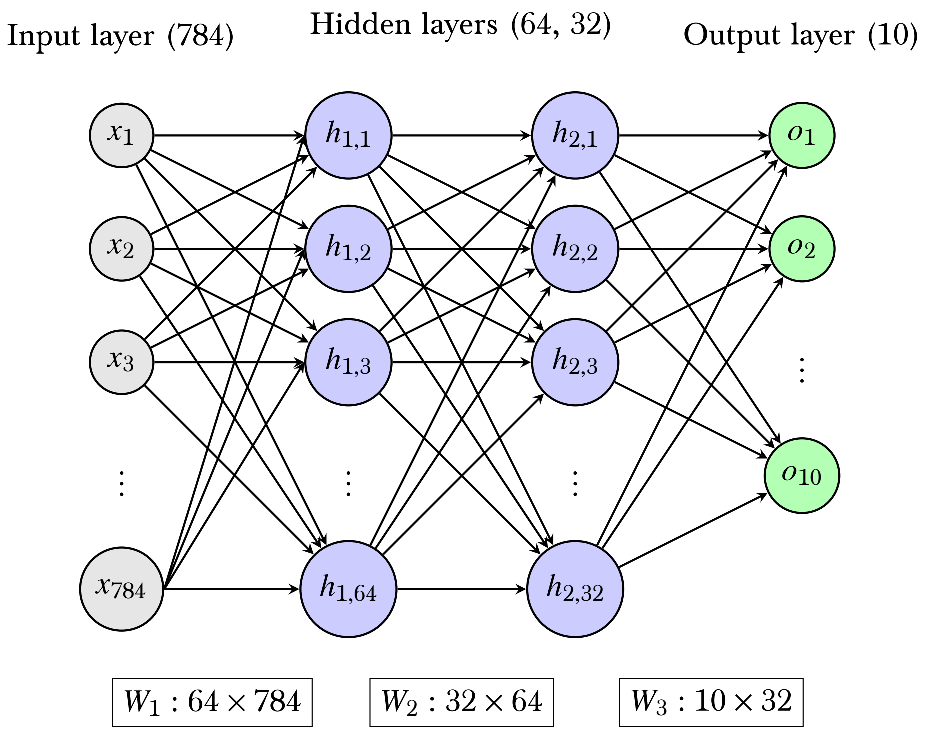


Figure 5: Multilayer Perceptron with two hidden layers. The input layer consists of 784 neurons, the first hidden layer has 64 neurons, the second hidden layer has 32 neurons, and the output layer has 10 neurons. The three dots in each layer signify that there are more neurons than graphically displayed.

Taking into account the considerations outlined above, a plausible choice would be to implement two hidden layers in the neural network architecture. The first hidden layer has a size of 64, while the second hidden layer has a size of 32 (Figure 5). These sizes are larger than the output size of 10 and smaller than the input size of 784. Moreover, following the principle of decreasing layer sizes from input to output, this configuration allows for the gradual transformation and compression of representations as they flow toward the output activations. While it is worth noting that any of the activation functions presented in Figure 3 could be used, we will adopt the classical logistic sigmoid function for the remainder of this paper as our activation function of choice.

The size of a neural network is commonly determined by its parameter count, which includes the total number of weights and biases. In our specific case, we can calculate the parameter count by summing up all the weights and biases, resulting in approximately 53,000 parameters (Equation 21).

$$\overbrace{64 \cdot 784 + 32 \cdot 64 + 10 \cdot 32}^{weights} + \overbrace{64 + 32 + 10}^{biases} = 52650 \approx 53k \tag{21}$$

Henceforth, different models will be referred to using the notation "net[size]," signifying the number of parameters in the network. Specifically, this particular network will be denoted as *net53k*. It is worth noting that the input and output layer sizes will remain unchanged, while the first hidden layer will consistently contain more neurons than the second hidden layer if two hidden layers are used.

2.3 Training

After preparing our training data in section 5 and planning a suitable network architecture in section 2.2, we can proceed with the training process. The training procedure is straightforward as we use the algorithms outlined in sections 1.4 & 1.5. To initiate training, we simply pass our training data set and the necessary hyperparameters to the mini-batch gradient descent algorithm (Algorithm 2). In particular, we need to consider three key hyperparameters: the learning rate η, the batch size S, and the number of epochs. The latter can either be determined by a preliminary training session where we monitor the network's progress, or we can continue training until the network's performance stops improving consistently over several epochs. The choice of batch size is not a critical concern at the moment since both smaller and larger batch sizes tend to yield similar outcomes in the end. However, smaller batch sizes are often recommended and tend to offer advantages [Wilson and Martinez 2003]. To maintain consistency throughout the training process of all models, a batch size of 32 will be utilized

Epoch	MSE	Accuracy
0	0.2794	1049/10000
20	0.0187	8849/10000
30	0.0143	9080/10000
40	0.0125	9202/10000
50	0.0114	9284/10000
60	0.0107	9320/10000
70	0.0103	9342/10000
80	0.0099	9360/10000
90	0.0096	9379/10000
100	0.0093	9396/10000
110	0.0091	9418/10000
120	0.009	9425/10000
130	0.0088	9428/10000
140	0.0088	9441/10000
150	0.0087	9446/10000
160	0.0086	9446/10000

Table 1: Performance metrics of net53k model trained with a learning rate of $\eta = 0.2$ over 160 epochs

unless otherwise specified.

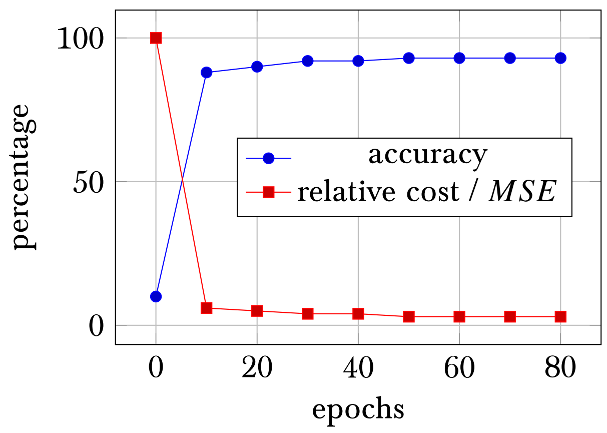


Figure 6: Relationship between cost and accuracy during the training of the Net53 model over 80 epochs using a learning rate of $\eta = 0.2$

Choosing an accurate learning rate is not as straightforward as the other hyperparameters, as it requires a good balance between accuracy and efficiency. Smaller learning rates result in slight improvements in network parameters per epoch, preventing overshooting, but lengthening the training process significantly. On the other hand, higher learning rates speed up training, especially in the early stages, but can cause problems when approaching the optimal parameter values due to the risk of overshooting caused by large steps. Table 1 presents the training progress of the net53k model over 160 epochs using a learning rate of 0.2, with updates recorded at 10-epoch intervals. In Section 1.4, our theoretical exploration of training neural networks was centered around the fundamental objective of minimizing the cost function. Figure 6 provides a visual representation of the direct association between the average cost across all training examples and digits, and the performance of the network.

2.4 Further experiments

Figure 7 highlights the importance of having a sufficient amount of training data. The model trained on 6000 training images performs significantly worse than models with the same architecture but trained on a larger portion of the dataset. This observation leads to the conclusion that having sufficient training data is crucial. However, it is important to note that increasing the dataset size does not necessarily guarantee a proportional improvement in the network's performance. In fact, using excessively large datasets can lead to longer training times and greater resource requirements without a noticeable improvement in performance. Therefore, it is essential to find a balance between the size of the dataset and the performance of the network.

Similarly, the size of the network also impacts its performance. Larger network sizes tend to enhance the network's performance, particularly in lower regions (Figure 8). However, it is important to note that larger network sizes necessitate more and larger matrix multiplications, resulting in increased computational requirements as well.

Another beautiful feature of neural networks is their adaptability; thanks to their ability to learn from training data, one can easily expose a model to different training datasets, allowing it to solve a wider range or completely different problems. By replacing the *MNIST* database with the *extended MNIST* database *(EMNIST)* and expanding the size of the output layer to accommodate a larger number of classes, the same model can be trained to classify both lowercase and uppercase handwritten characters in addition to just digits [Cohen et al. 2017].

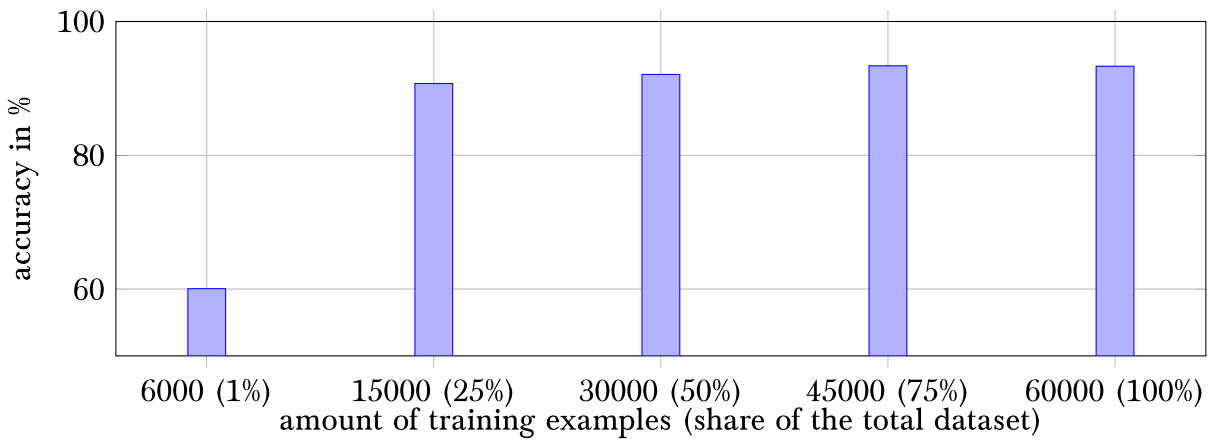


Figure 7: net13k trained of different sized portions of the training dataset over 30 epochs with a learning rate of $\eta = 3$

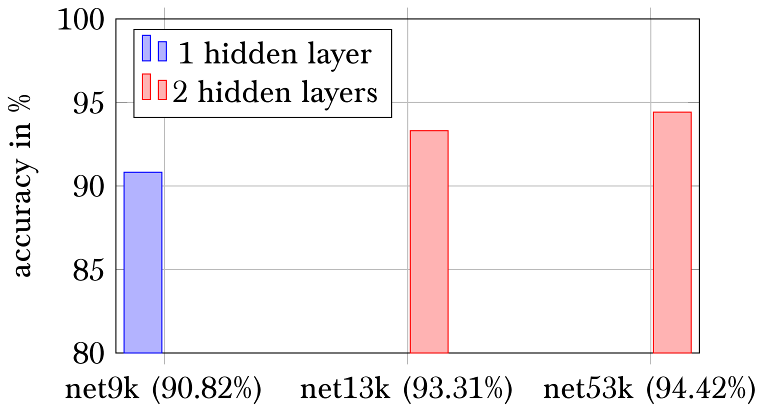


Figure 8: Comparison of accuracies achieved by models with varying sizes

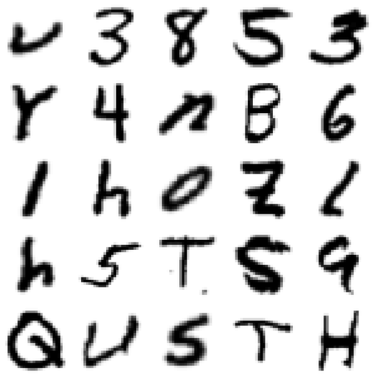


Figure 9: Randomly selected training images from the *EMNIST Balanced* database

3 Conclusion

In this paper, we have explored deep learning as an alternative approach to conventional methods for solving complex problems such as digit recognition. The basic idea behind deep learning is to approximate a function y based on different training examples consisting of input features x and their corresponding expected outputs $y(x)$. The ultimate goal is to accurately predict $y(x)$ based on x for every example.

Deep neural networks, drawing inspiration from fundamental concepts in the human brain such as neuroplasticity and long-term potentiation, are employed to tackle complex tasks. Specifically, we focused on the Multilayer Perceptron (MLP), a dense feed-forward neural network comprising multiple layers of neurons. Each neuron's activation is determined by the activations in the preceding layer and the network's parameters. The interconnections between neurons, established through weighted connections, facilitate information flow throughout the network. By calculating the weighted sum of activations from the previous layer, adjusted by a bias term, and applying a non-linear activation function, non-linearity is introduced. Initializing the activations of neurons in the input layer, representing the features x, enables the transformation of these features into output layer activations. Each activation in the output layer signifies the probability of the input data belonging to a specific class.

The weights and biases are called the parameters of the network and need to be learned through some training algorithm. In particular, we discussed gradient-based learning, which is based on the idea of having a cost function that measures how poorly our network is performing. This cost function is then minimized using gradient descent optimization. In gradient descent, the parameters are updated based on their negative gradient, since the gradient itself points in the direction of the steepest increase. In other words,it shows how we would need to modify the parameters to increase the cost the most. The gradients are obtained using an algorithm called backpropagation, which calculates the partial derivatives of the cost function with respect to the network's weights and biases, allowing an evaluation of how changes in parameters affect the overall cost.

After taking a look at the theoretical foundation of neural networks, we attempted to train a model to recognize handwritten digits in the second part of this paper. The model which was trained on the *MNIST* dataset using the explored methods was able to quickly obtain accuracies over 94% on unseen data. Experiments underlined the importance of sufficient training data and good hyperparameter choices as well as the relevance of the network's size.

Developing an algorithm based on manually defined rules and sequences to solve the same task would have been significantly more complex and challenging, and its success would likely be far less comparable to that of deep learning. Another major advantage can be found in the modifiability of neural networks: It is for example possible to modify a model initially designed for classifying handwritten digits to confidently classify all handwritten characters by just replacing the underlying training dataset and appropriately scaling the output layer size..

Deep learning techniques have revolutionized the field of handwritten digit recognition, leading to a wide range of real-world use cases across industries where the accurate identification and interpretation of handwritten digits are paramount like in postal services or banking. To further improve performance and to enable the model to confidently classify uncentered and unfiltered symbols or even longer strings, one may take a look at convolutional neural networks as well as segmentation and preprocessing algorithms that are applied to the image data before it is fed into the model.

4 Additional material

4.1 Implementation in Python: Single-layer perceptron

```python
    def step(x):
    if x > 0:
        return 1
    return 0


class Perceptron:
    def __init__(self, input_size, learning_rate = 0.01) -> None:
        self.learning_rate = learning_rate
        self.weights = np.random.randn(input_size,1)
        self.bias = np.random.randn()

    def classify(self,X):
        z = np.dot(X,self.weights) + self.bias
        return step(z)

    def train(self, training_features, training_labels, epochs = 100):
        for epoch in range(epochs):
            total_error = 0
            for X, label in zip(training_features,training_labels):
                error = label - self.classify(X)


                self.bias += self.learning_rate * error
                for i in range(len(self.weights)):
                    self.weights[i] += self.learning_rate * error * X[
                                                                    i]

                total_error += abs(error)

            if epoch % 10 == 0:
                print(f"Epoch {epoch}: Error = {total_error}")

            elif total_error == 0:
                print(f"Epoch {epoch}: Error = {total_error}")
                break
        print(f"Epoch {epoch}: Error = {total_error}")


    def eval(self ,features,labels):
        correct = 0
        for x,y in zip(features,labels):
            if self.classify(x) == y:
                correct += 1

        return f"{(correct/len(features))*100}%"
```

4.2 Implementation in Python: Neural network

```
import matplotlib.pyplot as plt
import numpy as np
import random


def sigmoid(x, derivative=False):
    if derivative:
        return sigmoid(x)*(1-sigmoid(x))
    return 1/(1+np.exp(-x))


class DenseNeuralNetwork():
    def __init__(self, layer_sizes):
        self.layer_sizes = layer_sizes
        self.layer_count = len(layer_sizes)

        self.biases = [np.random.randn(y, 1) for y in self.layer_sizes
                                                   [1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.layer_sizes[:-1], self.
                                                               layer_sizes
                                                               [1:])]

    def feed_forward(self, a, cache=False):
        activations = [a]
        weighted_inputs = []

        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, a)+b
            a = sigmoid(z)

            activations.append(a)
            weighted_inputs.append(z)

        if not cache:
            return a
        return a, activations, weighted_inputs

    def update_parameters(self, weight_gradient, bias_gradient,
                                 learning_rate):
        for i in range(len(self.weights)):
            self.weights[i] -= learning_rate * weight_gradient[i]
            self.biases[i] -= learning_rate * bias_gradient[i]

    def train(self, training_data, gd="mini-batch", batch_size=32,
                                 learning_rate=0.1, epochs=250,
                                 test_data=None, interval=None):
        if interval == None:
            interval = epochs // 100 if epochs >= 100 else 5

        for epoch in range(epochs):
            if gd == "mini-batch":
                self.train_mini_batch(
                    training_data, batch_size, learning_rate, epochs,
                                                          test_data)
            elif gd == "batch":
                self.train_batch(training_data, learning_rate,
                                 epochs, test_data)
            elif gd == "sgd":
                self.train_sgd(training_data, learning_rate, epochs,
                                                  test_data)
            else:
                raise Exception("Invalid Gradient Descent Method")

            if epoch % interval == 0 or epoch == epochs - 1:
                print(
                    f"Epoch {epoch}: {self.evaluate(test_data)} MSE: {
                                                        np.mean(self.
                                                        MeanSquaredError
                                                        (test_data))}")

    def train_sgd(self, training_data, learning_rate=None, epochs=None
                                        , test_data=None):
        for x, y in training_data:
            bias_gradients, weight_gradients = self.backprop(
                x, y)

            self.update_parameters(
                weight_gradients, bias_gradients, learning_rate)

    def train_batch(self, training_data, learning_rate=None, epochs=
                                        None, test_data=None):
        bias_gradients = np.array([np.zeros(b.shape)
                                   for b in self.biases], dtype=object
                                                                      )

        weight_gradients = np.array([np.zeros(w.shape)
                                     for w in self.weights], dtype=
                                                                        object
                                                                        )



        for x, y in training_data:
            bias_gradients_single, weight_gradients_single = self.
                                                  backprop(
                x, y)

            bias_gradients += bias_gradients_single
            weight_gradients += weight_gradients_single

        bias_gradients /= len(training_data)
        weight_gradients /= len(training_data)

        self.update_parameters(weight_gradients, bias_gradients,
                                            learning_rate)

    def train_mini_batch(self, training_data, batch_size=32,
                                        learning_rate=0.1, epochs=250,
                                        test_data=None):
        bias_gradients = np.array([np.zeros(b.shape)
                                   for b in self.biases], dtype=object
                                                                      )

        weight_gradients = np.array([np.zeros(w.shape)
                                     for w in self.weights], dtype=
                                                                        object
                                                                        )

        random.shuffle(training_data)

        for i in range(0, len(training_data), batch_size):
            batch = training_data[i:i+batch_size]
            batch_size = len(batch)

            for x, y in batch:
                bias_gradients_single, weight_gradients_single = self.
                                                   backprop(
                    x, y)
                bias_gradients += bias_gradients_single
                weight_gradients += weight_gradients_single

            bias_gradients /= batch_size
            weight_gradients /= batch_size

            self.update_parameters(
                weight_gradients, bias_gradients, learning_rate)

    def backprop(self, x, y):
        last_layer = self.layer_count - 2
        weights_gradients = np.array(
            [np.zeros(w.shape) for w in self.weights], dtype=object)
        bias_gradients = np.array([np.zeros(b.shape)
                                  for b in self.biases], dtype=object)
        E = [0] * (last_layer+1)

        _, a, z = self.feed_forward(x, True)

        E[last_layer] = np.multiply(a[-1]-y, sigmoid(z[-1], True))

        bias_gradients[last_layer] = E[last_layer]
        weights_gradients[last_layer] = np.dot(
            E[last_layer], a[last_layer].transpose())

        for layer in range(last_layer-1, -1, -1):

            E[layer] = np.multiply(
                np.dot(self.weights[layer+1].transpose(), E[layer+1]),
                                                    sigmoid(z[layer],
                                                   True))
            bias_gradients[layer] = E[layer]
            weights_gradients[layer] = np.dot(E[layer], a[layer].
                                               transpose())

        return bias_gradients, weights_gradients

    def evaluate(self, test_data):
        correct = 0
        for x, y in test_data:
            if np.argmax(y) == np.argmax(self.feed_forward(x)):
                correct += 1
        return f"{correct}/{len(test_data)}"
```

```python
    def MeanSquaredError(self, test_data):
        mse = 0
        for x, y in test_data:
            y_hat = self.feed_forward(x)
            mse += (y_hat-y)**2
        return mse/len(test_data)
```

References

Amidi, Shervine and Afshine Amidi (2018). “Deep Learning cheatsheet”. In: *CS 229, Stanford University.*

Cohen, Gregory et al. (2017). “EMNIST: Extending MNIST to handwritten letters”. In: *2017 international joint conference on neural networks (IJCNN).* IEEE, pp. 2921–2926.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press.

Google, Developers (2023). *Machine Learning Glossary.* `https://developers.google.com/machine-learning/glossary`. Accessed: 22.5.2023.

Groß, Jorge et al. (2021). “Neurobiologie”. In: *Biosphäre: Sekundarstufe II, Gesamtband*, pp. 264–268.

Heaton, Jeff (2015). *Artificial intelligence for humans, volume 3: Deep learning and neural networks.* by Heaton Research, Inc.

Hinton, Geoffrey, Nitish Srivastava, and Kevin Swersky (2012). “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent”. In: *Cited on* 14.8, p. 2.

Kriesel, David (2006). “Neuronale Netze”. In: *Rheinische Friedrich-Wilhelms-Universität*, p. 18.

LeCun, Y. et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: `10.1109/5.726791`.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). “Deep learning”. In: *nature* 521.7553, pp. 436–444.

LeCun, Yann, C Cortes, and CJC Burgess (2012). *The MNIST Database of handwritten images.*

Lederer, Johannes (2021). “Activation functions in artificial neural networks: A systematic overview”. In: *arXiv preprint arXiv:2101.09957.*

Li, Mingwei, Zhenge Zhao, and Carlos Scheidegger (2020). “Visualizing Neural Networks with the Grand Tour”. In: *Distill.* https://distill.pub/2020/grand-tour. DOI: `10.23915/distill.00025`.

McCulloch, Warren S and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5, pp. 115–133.

Minsky, Marvin and Seymour Papert (1969). “Perceptrons”. In: *Cambridge, MA: MIT Press* 6, pp. 318–362.

Mitchell, Tom M. (1997). *Machine Learning.* IL McGraw Hill, pp. 81–108.

Nielsen, Michael A (2015). *Neural networks and deep learning.* Vol. 25. Determination press San Francisco, CA, USA. Chap. 1-2.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). “Learning representations by back-propagating errors”. In: *nature* 323.6088, pp. 533–536.

Wilson, D Randall and Tony R Martinez (2003). “The general inefficiency of batch training for gradient descent learning”. In: *Neural networks* 16.10, pp. 1429–1451.

List of Figures

List of Algorithms

List of Tables

Mainz, 25.05.2023 [illegible]