Artificial Neural Networks

Benjamin Schoofs

August 10, 2022

Abstract

This paper pursues the issue of how neural networks learn using gradient-based approaches. The fundamental equations underlying the learning process will be derived and visualized. Based on this, the code for a neural network will be implemented and applied.

Contents

Preamble

Neural networks are algorithms capable of solving complex tasks, such as image recognition, playing go, or quantitative trading. This paper will explore how neural networks tackle these problems by the example of image recognition.

The fundamental notion underlying artificial neural networks (ANNs) is that programs can learn by altering themselves. Thus, ANNs are based on a relatively simple first set of rules which convert an input, resembling an image of a handwritten digit for instance, into a random output. This output is linked to a qualitative statement, enabling the network to claim that the image shows a handwritten 9, for example. At the same time, the algorithm is handed a second set of rules according to which it gradually alters its first set of rules. Hereby, the ANN can learn step-by-step to deal with the assigned task – in this case to classify images correctly.

The first set of rules will be outlined in section 1. The second set of rules by which a network learns will be explained in section 2. Section 3 implements an approach to solve the MNIST handwritten digit classification problem.

1 How do neural networks operate?

Assume a neural network is supposed to classify images of handwritten digits from zero to nine. In this case, the ANN should take images as an input and return a classification – $0, 1, \ldots, 9$. Mathematically, this can be phrased as a mapping ϕ from the input space χ, the space of all possible images of handwritten digits in a chosen format, to the output space $\Phi := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$:

$$\phi : \chi \to \Phi$$

For computations to become possible, the input and output spaces need to be converted into a mathematical framework. Suppose the images to be classified are in a 28×28 format. For simplicity, the images in this hypothetical example can be assumed to be black and white. We can now ascribe a brightness $b_n \in [0, 1]$ to each individual pixel, $n \in \{1, 2, \ldots, 784\}$. These values representing the brightness of the image at different points can be compactly summarized as a column vector:

$$\vec{x} = \begin{bmatrix} b_1 & b_2 & \ldots & b_{784} \end{bmatrix}^T$$

To understand neural networks, a vector such as $\vec{x}$ can simply be thought of as a list of values. $\vec{x} \in \chi$, the input of the net, is supposed to be mapped to the output space Φ. It is difficult, however, to map a 784-dimensional vector representing an image of a digit to that digit directly. This will become evident once the architecture of the mapping ϕ has been outlined. Therefore, $\{0,1,2,3,4,5,6,7,8,9\}$ requires to be translated into a vectorial representation. One approach would be to let the output space be

$$\Phi := \left\{ \begin{bmatrix} a_0 & a_1 & \dots & a_9 \end{bmatrix}^T \mid a_i \in \mathbb{R} \right\},$$

where the index of the maximum entry is interpreted as the prediction of the network. For example, if a_6 is the maximum entry, the network classifies the input as an image of a six.

The first set of rules alluded to in the introduction is a way to map any input from the input space to an output in the output space, equipping the net with an arbitrary mapping ϕ. Subsequently, the second set of rules will come into play to iteratively alter ϕ such that the net converges to a mapping ϕ^* which is capable of correctly mapping from χ to Φ.

1.1 Neurons, layers, networks

Neurons are the basic building blocks of neural networks. They take an input vector $\vec{x}$ and map it to an output number $a \in \mathbb{R}$ as follows:

$$a = a(\vec{x}) = \varphi(\vec{x} \cdot \vec{w} + b) = \varphi(z)$$

a is called the activation of the neuron. φ is the activation function of the neuron. $\vec{x} \cdot \vec{w} + b$ is the weighted sum z of the input. It can be thought of as follows:

$$z = z(\vec{x}) = \sum_i w_i x_i + b \tag{1}$$

Every entry x_i of the input vector $\vec{x}$ is multiplied by a weight w_i. These terms are added up. Furthermore, a bias b is added to this sum. The weighted sum z is mapped to the activation a by the differentiable activation function φ to introduce non-linearity to the network. This will allow the network to learn more complex mappings later on [Nie15].

A layer l is an array of n neurons. Every neuron j in layer l is given the same input vector $\vec{x}$ and outputs a number a_j. Note, that the neurons can have

different weights and biases which is why their activations can differ. Since every neuron outputs a number and there are n neurons, layer l returns a list of n numbers – which can be summarized in a vector a^l with n entries. The j-th entry of a^l, a^l_j, is the activation of the j-th neuron of layer l. a^l serves as an input vector for the next layer $l+1$. In the very same way as layer l, layer $l+1$ will transform the input vector into a new activation vector a^{l+1}. By combining L layers, a neural network function ϕ can be built:

$$\phi(\vec{x}) = (a^L \circ a^{L-1} \circ ... \circ a^1)(\vec{x}) \tag{2}$$

This is the network equation. It captures the idea of inputting $\vec{x}$ into the first layer, using a^1 as an input for the second layer and so on. This way, an input vector, representing an image, for instance, can be mapped to a vector, incarnating the digit which is shown by the image.

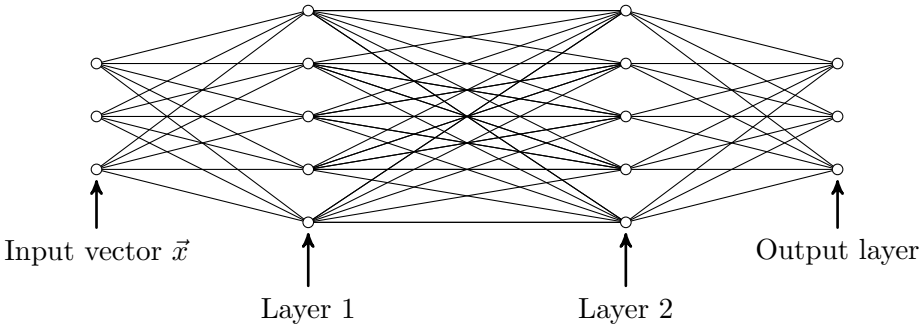

Figure 1: A neural network

Figure 1 illustrates how the neurons of one layer feed their outputs to the next layer, in the manner prescribed by equation (2): Every entry of the input vector is connected to every neuron in the first layer by a specific weight. Using these weights, one bias for every neuron and an activation function φ, the activations of the first layer can be computed and passed on to the next layer. The process is repeated sequentially for the remaining layers. As all neurons are connected to every neuron in the next layer, the network is considered to be fully connected. Note, that the input vector $\vec{x}$ may sometimes be called the input layer. From a mathematical point of view, this does not change anything. Moreover, layers 1 and 2 are called the hidden layers because they are "hidden" between the input and the output layer [Nie15].

While this way of thinking about neural networks helps to understand the big picture, it is sensible to make slight adjustments to the current mathematical

framework. This will allow for more efficient computations and a more intuitive way of thinking about ϕ.

1.2 Transitioning to a matrix-based approach

The network function ϕ iteratively transforms the input vector $\vec{x}$ by applying layer after layer. This series of transformations can be rephrased mathematically as matrix multiplications combined with potentially non-linear activation functions.

Every layer l is associated with a weight matrix $W^l \in M_{nd}(\mathbb{R})$ containing the weights of the neurons of that layer. The components of W^l are structured as follows:

$$W^l = \begin{bmatrix} w^l_{11} & \dots & w^l_{1d} \\ \vdots & \ddots & \vdots \\ w^l_{n1} & \dots & w^l_{nd} \end{bmatrix}$$

where d is the dimension of the input of that layer and n is the number of neurons. The weight of the j-th neuron for the k-th entry of the input vector is w^l_{jk}. Moreover, every layer l has a bias vector b^l containing all the biases of the neurons of that layer. Using these ideas, we can rewrite the activation a^l of a layer l as

$$a^l(\vec{x}) = \varphi\left(W^l\vec{x} + b^l\right) = \varphi(z^l). \tag{3}$$

This expression captures the idea behind applying a layer to $\vec{x}$ more clearly. Multiplying $\vec{x}$ by the Matrix W^l can be thought of as rotating, stretching, and squishing the input vector linearly. The rotation can change the dimension of the vector. This makes sense, considering a case in which the input has dimension 10 and there are 20 neurons. In this case, the layer will return a vector of dimension 20 which is accounted for by the matrix multiplication (multiplying an $n \times d$ matrix by a vector of dimension d will result in a vector of dimension n). Adding the bias b^l changes the length and direction of $W^l\vec{x}$. Finally, φ transforms the weighted sum z^l in a non-linear fashion. Note, that φ maps from $\mathbb{R}$ to $\mathbb{R}$. In equations containing vectors or matrices, φ is applied component-wisely to every element of that vector or matrix.

Thus, the network equation (2) boils down to transforming an input vector $\vec{x}$ by rotating, stretching, squishing, and applying activation functions iteratively to it. This idea of transforming a vector the way a layer does is demonstrated

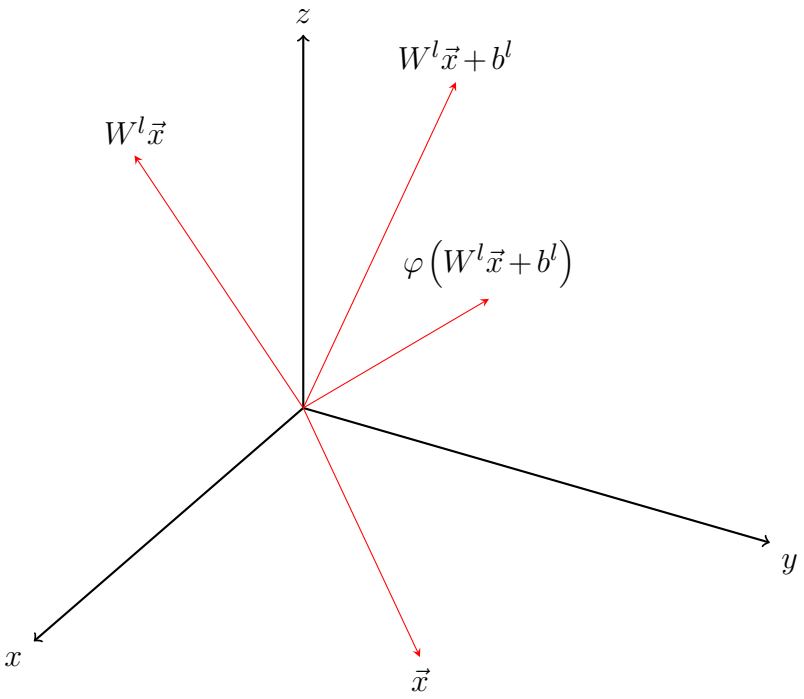

Figure 2: A layer transforms the input vector $\vec{x}$

by figure 2. To understand the network equation, imagine applying a transformation of this sort several times consecutively. Note, that the input vector $\vec{x}$, living in the two-dimensional plane, is lifted into the third dimension by W^l. This suggests that the visualized layer takes in a two-vector. Furthermore, it has three neurons as this is the dimension of the output vector $\varphi\left(W^l\vec{x}+b^l\right)$. Most networks tend to deal with higher dimensions, but this visualization conveys the basic notion of what neural networks do.

2 How do neural networks learn?

For a neural network to be able to learn, it needs to be capable of evaluating its behavior somehow. Therefore, we define a smooth cost function C which rates the current network function ϕ based on its accuracy on training data.

$$C : P \to \mathbb{R}$$

P – for parameters – is the space of all possible combinations of weights and biases of the network. If the ANN performs perfectly, C attains its global minimum, indicating that the network makes as few errors as possible. The network's learning process will revolve around finding this minimum of C. Put

differently, C will allow the ANN to find weights and biases such that ϕ maps as accurately as possible from χ to Φ.

C can be thought of as a surface as is visualized in figure 3. The plane below this surface represents P. Note, that the network could have several thousand or millions of parameters, but that would be difficult to visualize. Thus, P is two-dimensional in figure 3. For every element of P, every set of weights and biases, there is an associated cost, measuring the accuracy of the network, when equipped with this particular set of parameters. To find the global minimum of C, the network could theoretically test a large number of possible sets of weights and biases and choose the one which performs the best. Or C could be minimized, using methods of calculus. However, when a large number of parameters is involved, these approaches become computationally infeasible. Hence, a more efficient algorithm to find the minimum of C is required.

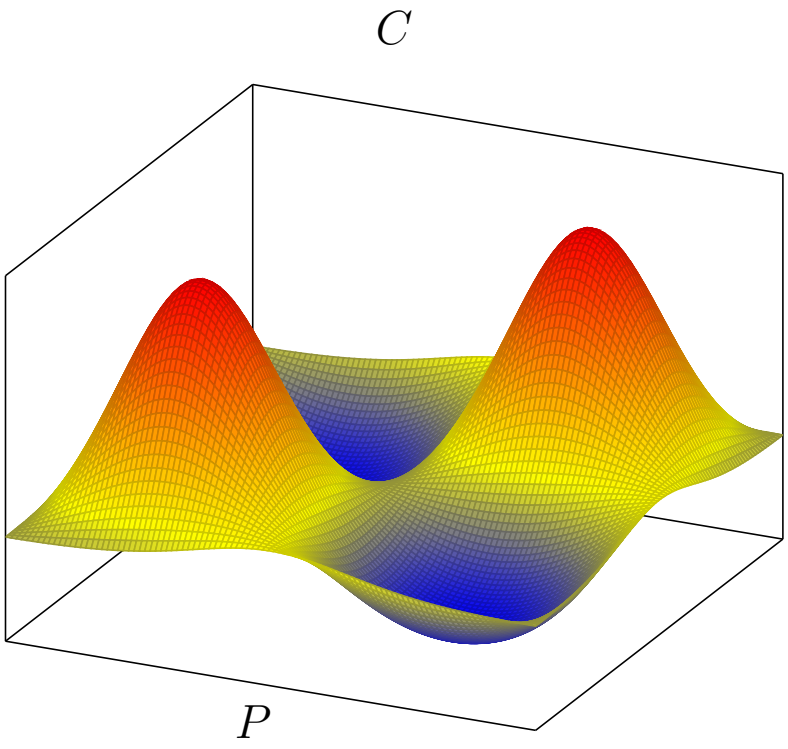

Figure 3: C traces out a graph above the parameter space P

2.1 Gradient descent

Gradient descent is a way of solving the given minimization problem efficiently by sliding down the surface of C. This process of sliding down to the minimum of the cost function plays out as follows: Initially, the network employs a

random set of parameters ρ_0. The network then slightly adjusts its parameters such that the cost function takes on a lower value. This step is repeated over and over again until the minimum is reached. As the cost function is smooth, the information about the local environment of C is encoded in the gradient of C. This allows the network to permanently find a step such that the cost function is decreased – unless $\nabla C = 0$ which happens at the desired minimum. Using some multivariable calculus, the directional derivative of C is

$$D_{\hat{v}}C = \nabla C \cdot \hat{v} = \|\nabla C\| \cos(\theta),$$

where $\hat{v}$ is a unit vector, indicating a direction, and θ is the angle between ∇C and $\hat{v}$. By moving through the parameter space in the direction minimizing $D_{\hat{v}}C$, the cost function is decreased as much as possible. The minimum of $D_{\hat{v}}C$ occurs when $\cos(\theta) = -1$, implying that $\theta = \pi$. Hence, ∇C and the direction of steepest descent $\hat{v}^*$ are at an angle of π which implies

$$\hat{v}^* = -\frac{\nabla C}{\|\nabla C\|}.$$

Tying all of this together, the network should update its parameters according to

$$\rho_t \to \rho_{t+1},\ \rho_{t+1} = \rho_t - \eta \nabla C. \tag{4}$$

ρ_t – the parameters currently employed by the network – can be thought of as a vector containing all weights and biases of the network. This vector is acted upon by the negative gradient of the cost function. By applying equation (4), the network moves a small step in the parameter space P, finding weights and biases that are slightly better at performing the task assigned to the network. A single step is shown in figure 4. As the network repeatedly changes its parameters in this manner, it reaches the minimum of the cost function and learns to perform the given task. Note, that a factor of η occurs in equation (4) to control the size of the steps taken by the network. This is useful because $-\nabla C$ is only locally the direction of steepest descent. The further the network moves from the current parameters ρ_t the less accurate this estimation becomes, requiring a new calculation of ∇C at that new point. At the same time, η should not be chosen too small, as this increases the number of steps needed to approach the minimum of C.

With the intuition of sliding down the landscape of C in mind, we can now try to find a concrete formula to update the individual parameters. To start with, how should the biases of the last layer be updated? According to equation

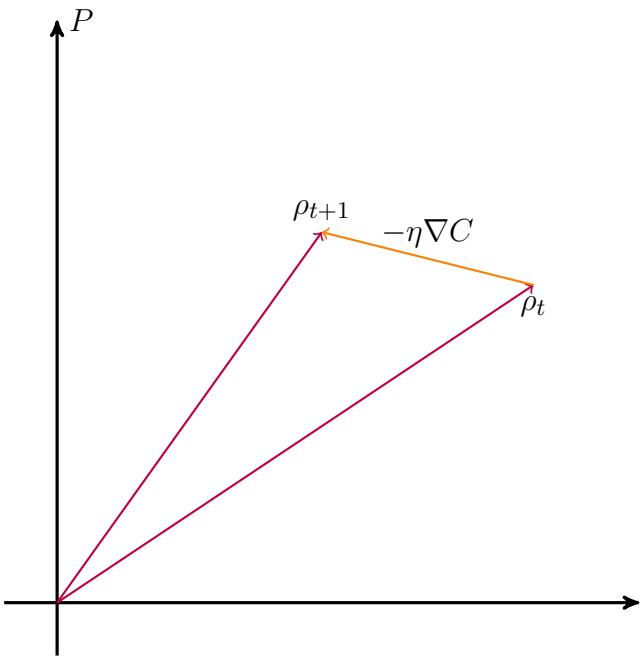

Figure 4: A step in the parameter space P

(4), updating a parameter requires the network to compute the derivative of C with respect to that individual parameter. Hence, the derivative of C with respect to any given bias b_j^L is needed to update that bias. The derivative can be found using the chain rule:

$$\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L}$$

Every term in z_j^L is constant with respect to b_j^L except for $+b_j^L$ at the very end of the weighted sum (1). Hence,

$$\frac{\partial z_j^L}{\partial b_j^L} = 1.$$

Furthermore, the derivative of a_j^L (3) with respect to z_j^L is

$$\frac{\partial a_j^L}{\partial z_j^L} = \varphi'(z_j^L).$$

Thus, the derivative of the cost function with respect to a given bias b_j^L is

$$\frac{\partial C}{\partial b_j^L} = \delta_j^L,$$

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\varphi'(z_j^L)$$

where we define δ_j^L to be the error of the j-th neuron in layer L. Merging these simplifications, biases in the last layer should be updated according to

$$b_j^L \to b_j^L - \eta \delta_j^L \tag{5}$$

To update weights, the network will analogously compute

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L}.$$

Again, the first two derivatives can be summarized as the error of the j-th neuron δ_j^L. As the network has to compute this value just once for every neuron, instead of calculating it for every single weight, this renders the process of gradient descent more efficient. Moreover, since z_j^L is a sum where each individual weight w_{jk} appears once with a factor of a_k^{L-1}, the third derivative simplifies to

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = \frac{\partial}{\partial w_{jk}^L} \left(\sum_i w_{ji}^L a_i^{L-1} + b_j^L \right) = a_k^{L-1}.$$

All further terms drop off as a constant when differentiating. Putting the pieces back together, the derivative of the cost function with respect to a weight w_{jk}^L is

$$\frac{\partial C}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}.$$

Hence, the weights of the last layer should be changed according to

$$w_{jk}^L \to w_{jk}^L - \eta \delta_j^L a_k^{L-1}. \tag{6}$$

Using, equations (5) and (6) the network can learn the optimal set of weights and biases for the last layer. To learn the ideal parameter configuration, however, the network needs to be able to compute the derivative of the cost function with respect to weights and biases in any layer. Following equation (4) and a similar approach to the one taken to find equation (5) and (6), this naturally leads to the back-propagation algorithm.

2.2 Back-propagation

The core question guiding the way to the back-propagation algorithm is, what are

$$\frac{\partial C}{\partial b_j^l} \quad \text{and} \quad \frac{\partial C}{\partial w_{jk}^l},$$

the derivatives required to perform gradient descent (4). To tackle this problem, it is sensible to recall the process of finding those derivatives for the last

layer. Glancing at equations (5) and (6), the error δ_j^L seems to play an important role. However, when defining δ_j^l for the last layer $l = L$ and using it to find (5) and (6), no special properties of the last layer were used. Hence, (5) and (6) remain valid, reducing the problem to finding the error δ_j^l for neurons in an arbitrary layer l. Using the multivariable chain rule, the error can be rewritten as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}.$$

Note, that the multivariable chain-rule requires summing over all intermediate variables z_k^{l+1}. Furthermore notice, that the first factor within the sum is simply the error of the k-th neuron in layer $l+1$. Combining this with the definition of the weighted sum z results in

$$\delta_j^l = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l} \left(\sum_i w_{ki}^{l+1} \varphi\left(z_i^l\right) + b_k^{l+1} \right) = w_{kj}^{l+1} \varphi'\left(z_j^l\right)$$

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \varphi'\left(z_j^l\right). \tag{7}$$

Once again, all terms independent of the differentiation variable drop off. This leaves an expression that allows the network to compute the error of a neuron in layer l using the error of layer $l+1$.

That is where the notion of back-propagation comes in: Instead of repeating a complete error calculation for every layer, the computations are framed recursively by back-propagating the error, layer by layer, for the sake of computational efficiency [Nie15].

Based on equations (1) to (7), the subsequent section will outline a way of implementing neural networks to solve the MNIST classification problem.

3 Implementation

The MNIST dataset consists of 70 000 samples of handwritten digits from zero to nine with associated classifications. The goal of this implementation is to train a neural network using 60 000 samples and predict the labels of the remaining 10 000 samples with an accuracy of above 90%. The split ensures that the ANN does not simply learn the images by heart, but rather learns

to understand patterns and generalize. Equations (1) to (7) are completely sufficient to build a network, equipped to meet this task. From the standpoint of computational efficiency, however, it is sensible to slightly modify the equations and transition to a fully-matrix-based approach. This will allow the algorithm to harness the efficiency of linear algebra libraries such as NumPy.

3.1 Fully matrix-based approach

The idea behind transitioning to a fully matrix-based approach is to enable the network to handle several inputs simultaneously by efficiently computing matrix-matrix multiplications. Inputs will be fed forward through the network by an array of matrix multiplications. Furthermore, errors will be calculated and backpropagated by a sequence of efficient matrix operations. This will speed up the algorithm significantly. The fully matrix-based approach requires representations of the inputs, biases, errors, and gradients as matrices. Assume, there is a batch of m training samples – m is called the batch size. Then the input vectors $\vec{x}_1$, ..., $\vec{x}_m$ are compactly summarized in an input matrix X of the following form:

$$X = \begin{bmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_m \end{bmatrix} \tag{8}$$

Furthermore, the bias b^l of a layer is extended to

$$B^l = \begin{bmatrix} b^l \\ \vdots \\ b^l \end{bmatrix}, \tag{9}$$

where B^l contains m rows. Using X and B^l, equation (3) can be rewritten as

$$A^l(X) = \varphi\left(X\left(W^l\right)^T + B^l\right) = \varphi\left(Z^l\right). \tag{10}$$

Note, that the resulting matrix will still have m rows. This is nice as it keeps the training samples seperate. It can be verified that this equation is equivalent to applying equation (3) to every single input and summarizing the outputs in a matrix, analogously to the way X was assembled. The error can be represented as a matrix

$$\delta^l = \begin{bmatrix} \delta^l_{11} & \dots & \delta^l_{1n} \\ \vdots & \ddots & \vdots \\ \delta^l_{m1} & \dots & \delta^l_{mn} \end{bmatrix}, \tag{11}$$

where n is the number of neurons of layer l and m is the batch size. Biases can be updated by replacing b_j^l and δ_j^l with B^l and δ^l in equation (5) and averaging over the training samples. Similarly, weights can be updated according to equation (6), using W^l and a weight-update matrix ΔW^l,

$$\Delta W^l = \frac{1}{m} \left(A^{l-1}\right)^T \delta^l. \tag{12}$$

A quick calculation reveals that the matrix multiplication and division by m accounts for taking an average over the training samples. Finally, the error can be back-propagated as follows:

$$\delta^l = \left(\delta^{l+1} W^l\right) \odot \varphi'(Z^l) \tag{13}$$

The operation $\odot$ represents element-wise multiplication, also known as the Hadamard product. Using equations (8) to (13), which I found by solving a problem posed at the end of chapter 2 of [Nie15], I implemented a fully matrix-based neural network. When testing it on the MNIST dataset, however, I found it to perform poorly at first. In the following, I will delineate some of the typical issues, early implementations struggled with.

3.2 Problems of neural networks

The seemingly malfunctioning network consists of one input layer with 784 neurons, one hidden layer with 64 neurons, and one output layer with 10 neurons. Initially, the learning rate was set to 1.0, all neurons employed the sigmoid activation function and gradient descent used the quadratic cost function. Weights and biases were initialized randomly over a gaussian distribution with a standard deviation of 1.0. According to all the work outlined in this paper, this approach should work. When running the network, however, the algorithm classified the images randomly and remained unable to learn. As it turns out, the initial weights were too large which led the sigmoid neurons to saturate. This slowed down the learning process significantly. Hence, I switched to initializing the weights of a layer randomly over a gaussian distribution with a standard deviation of $n^{-\frac{1}{2}}$, where n is the dimension of the input [LBOM98]. After making this change, the network started to act in a non-random fashion. The results were still quite poor though as the learning process continued to be slow. A closer look at the combination of sigmoid neurons and the quadratic cost function reveals that the derivatives, needed to perform gradient descent, tend to become small when the outputs of the sigmoid neurons are close to 0 or

1 [Nie15]. Switching to ReLU neurons further increased performance. Moreover, the initial learning rate of 1.0 was poorly chosen. To find a better one, a bayesian hyper-parameter optimization approach was employed [SLA12], the code for which can be found in a GitHub repository by Thomas Huijskens [Hui16]. Finally, I implemented L2-regularization and an algorithm to artificially expand the training data [Nie15]. These slight modifications enhanced the ANN's accuracy to 90.8%.

References

[Hui16] Thomas Huijskens. Bayesian optimization with gaussian processes, 2016. https://github.com/thuijskens/bayesian-optimization.

[LBOM98] Yann LeCun, Leon Bottou, Genevieve B. Orr., and Klaus-Robert Müller. Efficient backprop, 1998.

[Nie15] Micheal A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. 2012.

Ich versichere, dass ich diese Facharbeit selbst angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt, und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.

Datum: 09.10.2022 Unterschrift: Benjamin Schogs