Georg-August-Universität Göttingen

Master Applied Statistics - Deep Learning Seminar

Training Deep Models

Olaf Menssen

Supervisors: Dr. Benjamin Säfken, Christoph Weisser

31 March 2019

Contents

Chapter 1

Introduction

1.1 MNIST Example

Assume there is a prediction or classification problem with a given training example $(\boldsymbol{x}, \boldsymbol{y})$, with $\boldsymbol{x}$ representing the data and $\boldsymbol{y}$ the true prediction or classification corresponding to $\boldsymbol{x}$. The free database MNIST will be used in this report, it provides 60,000 of such training examples. In this context, the data of a single example to be classified is a 28x28 grayscale picture (a 28x28 matrix of brightness values between 0 and 255) of a handwritten digit. The true value corresponding to this is a single number between 0 and 9. Some data preprocessing is assumed in this report: the brightness values are normalized to be between 0 and 1 and the matrix is flattened, so that we have a vector of length 784 instead of a 28x28 matrix: $\boldsymbol{x} = (x_1, ..., x_{784})'$, $x_1, ..., x_{784} \, \epsilon \, [0, 1]$. The true value is represented by a logical vector of length 10, where each element indicates whether its index (starting from 0) was the true number, e.g. $\boldsymbol{y} = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)'$ means there was a 2 written in the picture represented by $\boldsymbol{x}$.

A neural net can be seen as one single function $f(\boldsymbol{x}) = \hat{\boldsymbol{y}}$. In this example, it has the 784 brightness values of a 28x28 grayscale picture of a handwritten digit as an input: $\boldsymbol{x} = (x_1, ..., x_{784})'$. The output are 10 values, one for each number from 0-9, which indicate how much the neural net thinks that the picture depicted the respective number: $\hat{\boldsymbol{y}} = (\hat{y_1}, ..., \hat{y_{10}})'$. When the true value was 2, the result of a somewhat trained neural net might look like: $\hat{\boldsymbol{y}} = (0.2, 0.1, 0.9, 0.0, 0.0, 0.3, 0.1, 0.0, 0.1, 0.0)'$, with 0.9 at index 2 being the biggest element. In a simple neural net with a given number of layers, the parameters of the function are the weights and biases of each neuron. The goal of loss optimization in neural networks is to find the parameters that lead to the most accurate prediction or classification.

1.2 Tensorflow Setup For The MNIST Example

The following code is a short example of how to implement a neural net to classify images of handwritten digits from the MNIST database into numbers. It is closely related to what we assumed in the introduction text (chapter 1.1) and later chapters. The typical TensorFlow implementation here in this section is slightly different: it takes matrices as an input and has a flatten activation function in the first layer to flatten the matrices to vectors, while in the rest of this report, we assume that the input neurons directly

represent the vectors (flattened matrices).

```
# Import necessary libraries
import tensorflow as tf
import numpy as np

# Load the data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Scale x so it ranges from 0 to 1
x_train = tf.keras.utils.normalize(x_train)
x_test = tf.keras.utils.normalize(x_test)

# Instantiate a standard Tensorflow gd optimizer
tf_gd_optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1)

# Instantiate, initialize and further configure the neural net
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(12, activation=tf.nn.relu),
  tf.keras.layers.Dense(12, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer = tf_gd_optimizer,
  loss='sparse_categorical_crossentropy',
  metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs = 3, verbose = 0)

# Do predictions
predictions = model.predict(x_test)

# The estimation of y corresponding to x_test[0], the x of training example 0:
print(predictions[0])
print(np.argmax(predictions[0])) # This is the estimation of y corresponding to x_test[0]
print(y_test[0]) # The true y corresponding to x_test[0]
```

This code was mostly obtained from my classmate Lars Gerdes and is closely related to the first TensorFlow tutorial: `https://www.tensorflow.org/tutorials`.

Chapter 2

Theory Of Training Deep Models

2.1 The Steps Of Training A Neural Net

Training a neural net is done in several steps, some of which are iterated repetitively. The following list outlines these steps:

1. Configuration of the neural net: Define the number of layers, layer depths, which loss function to use, etc..

2. For initialization, give all parameters $\boldsymbol{\theta}^{(it0)}$ (e.g. the weights and biases of all neurons) random values. The index $^{(it\,j)}$ denotes in which iteration of the training algorithm we are in.

3. Do a prediction or classification $\hat{\boldsymbol{y}}^{(ex0)}$ for a single training example $(\boldsymbol{x}^{(ex0)}, \boldsymbol{y}^{(ex0)})$. The prediction or classification will be bad due to the randomly chosen weights and biases. The index $^{(ex\,i)}$ refers to the specific training example, e.g. $\boldsymbol{x}^{(ex0)}$ could be the vector representation of a handwritten 5 and $\boldsymbol{y}^{(ex0)}$ the logical vector indicating that number 5 is correct.

4. Quantify how bad the prediction or classification was with a metric $J^{(ex0)}$ called the loss, it is a single scalar. It is computed with the chosen loss function in step 1. The loss function can be seen as a function that has $\boldsymbol{\theta}$ as input variables and $\boldsymbol{x}^{(ex0)}$ and $\boldsymbol{y}^{(ex0)}$ as fixed parameters (this means the roles have reversed compared to the neural net function): $J^{(ex0)} = J(\boldsymbol{\theta} \mid \boldsymbol{x}^{(ex0)}, \boldsymbol{y}^{(ex0)})$. Theoretically, this step can be omitted as the loss values are not directly required in later steps. However, it is good to be able to see whether the loss decreases between steps (which - on average - is the goal of the optimization).

5. Compute a gradient $\nabla J^{(ex0)}$ of the loss **at the point** $\boldsymbol{\theta}^{(it0)}$ by backpropagating through the neural net. Note that $\nabla J^{(ex0)}(\boldsymbol{\theta}^{(it0)})$ is a vector of real numbers and not a vector of partial derivative functions, because it gives the values of the partial derivative functions at the corresponding vector elements of of $\boldsymbol{\theta}^{(it0)}$, which is also a vector of real numbers.

6. Repeat steps 3-5 with all other training examples $(\boldsymbol{x}^{(ex\,i)}, \boldsymbol{y}^{(ex\,i)})$ in the training set or in a mini-batch. Then average the gradients to get the gradient to be later used

for one update step (one iteration) of the optimization algorithm. $\nabla J^{(it0)}(\boldsymbol{\theta}^{(it0)}) = 1/m * \nabla J^{(ex0)}(\boldsymbol{\theta}^{(it0)}) + \nabla J^{(ex1)}(\boldsymbol{\theta}^{(it0)}) + ... + \nabla J^{(ex\,m)}(\boldsymbol{\theta}^{(it0)})$

7. Compute the update for $\boldsymbol{\theta}^{(it0)}$, denoted as $\Delta\boldsymbol{\theta}^{(it0)}$, so that when moving from $\boldsymbol{\theta}^{(it0)}$ to $\boldsymbol{\theta}^{(it0)} + \Delta\boldsymbol{\theta}^{(it0)} = \boldsymbol{\theta}^{(it1)}$ we are hopefully at a point closer to a local minimum of $J(\boldsymbol{\theta} \mid \boldsymbol{x}, \boldsymbol{y})$. The update is determined by the optimization algorithm. Many algorithms use $\nabla J^{(it0)}(\boldsymbol{\theta}^{(it0)})$ for that. For the gradient descent algorithm, it is simple: $\Delta\boldsymbol{\theta}^{(it0)} = lr * -\nabla J^{(it0)}(\boldsymbol{\theta}^{(it0)})$, with lr being a constant (the learning rate).

8. Update $\boldsymbol{\theta}$: $\boldsymbol{\theta}^{(it1)} = \boldsymbol{\theta}^{(it0)} + \Delta\boldsymbol{\theta}^{(it0)}$ and use the new $\boldsymbol{\theta}^{(it1)}$ as the parameters of the neural net from now on.

9. Iterate through steps 3-8, each time with updated neural net parameters $\boldsymbol{\theta}^{(it\,j)}$. This is done until a defined maximum number of iterations is reached or the updates become so small that their norm is below a specified value, called the precision: $\|\Delta\boldsymbol{\theta}^{(it\,j)}\| < precision$. The training examples to be used in each iteration can either be repetitively all elements of the available training set. Or they can be elements of mini-batches, which are subsets of the shuffled full training set. In both cases, these iterations are the steps of the optimization algorithm: On average, each iteration should bring $\boldsymbol{\theta}^{(it\,j)}$ closer to a $\boldsymbol{\theta}$ where $L(\boldsymbol{\theta} \mid \boldsymbol{x}, \boldsymbol{y})$ has a local minimum.

10. After step 9 is done, we arrive at a certain $\boldsymbol{\theta}$ that should be better than the random values in step 2. The neural net is now considered as "fitted" or "trained", or in other words, the neural net has "learned".

As we can see, the whole model fitting process is an optimization algorithm at large. Only after step 8 was the (attempted) minimization of the loss function of the neural net $L = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = L(f(\boldsymbol{x} \mid \boldsymbol{\theta}), \boldsymbol{y}) = L(\boldsymbol{\theta} \mid \boldsymbol{x}, \boldsymbol{y})$ finished. In step 6 are the optimizations for single training examples happening. This step is influenced by step 1 - the loss function that was chosen and by step 5 - backpropagation.

2.2 Backpropagation

Backpropagation is a method to get the gradient $\nabla J^{(ex\,i)(\boldsymbol{\theta}^{(it\,j)})}$ for step 5) of the training algorithm as given in chapter 2.1. The gradient contains all partial derivatives of J with respect to each parameter of the neural net. Here we will focus on the weight parameters, while the biases are also parameters and additional parameters are possible, depending on the configuration of the neural net. In backpropagation, we take a look at the last layer first, then the second last layer, and so on. This is why it is called "back"-propagation.

In this report, we usually use a neural net configured to process the MNIST data as a practical example. However, such a neural net would be too complex to investigate backpropagation on a low level, because of its number of neurons. It is just too complex, not too complicated. Here In the backpropagation examples we will use a neural net with as many layers, just with fewer neurons per layer. And we will use only one activation function. We assume that we have a neural net with 3 neurons in the input layer, 2 neurons in the following hidden layer, 3 neurons in another following hidden layer and 2 neurons

in the output layer. This is depicted in figures 2.1 and 2.2. The following notation is used:

a_i^l: Activation of neuron i in layer l.
z_i^l: Linear combination of all neurons in the layer previous to l, which is fed into the activation function to become a_i^l. E.g. The z_i^l value corresponding to neuron 1 of layer 4 is: $z_1^4 = w_{11}^4 a_1^3 + w_{12}^4 a_1^3 + w_{13}^4 a_1^3 + b_1^4$.
w_{ij}^l: Weight that z_i^l gives to a_j^{l-1}.
b_i^l: Bias in z_i^l. The bias can control how big the weighted sum of all activations in the previous layer have to be so that z_i^l becomes so big that the activation function of z_i^l is so big that a_i^l is noticeable bigger than 0 and therefore "fires".

We assume that the input layer activations are just the x values that are fed into it. Layer 2, 3 and 4 are configured to have the ReLU function as the activation function: $a_i^l = ReLU(z_i^l) = max(0, z_i^l)$. The ReLU function is not differentiable in $z_i^l = 0$. However, a not mathematically correct derivative of ReLU can be defined, in which the derivative at $z_i^l = 0$ is defined to be 0:

$$\frac{\partial a_i^l}{\partial z_i^l} = ReLU'(z_i^l) = \begin{cases} 0, & \text{if } z_i^l \leq 0 \\ 1, & \text{if } z_i^l > 0 \end{cases}.$$

To start easy, we will have a look at a weight of a neuron in the last layer so we don't need to propagate far into the neural network, coming from behind. Let's say we are interested in the weight w_{11}^4, this is how the first neuron in layer 4 weights the activation of the first neuron of layer 3. More precisely, we are interested in $\frac{\partial J}{\partial w_{11}^4}$ - it gives how much the loss changes when w_{11}^4 is increased by a tiny amount. $\frac{\partial J}{\partial w_{11}^4}$ is one element of the gradient ∇J, which we need to find all elements of. Figure 2.1 depicts the process graphically.

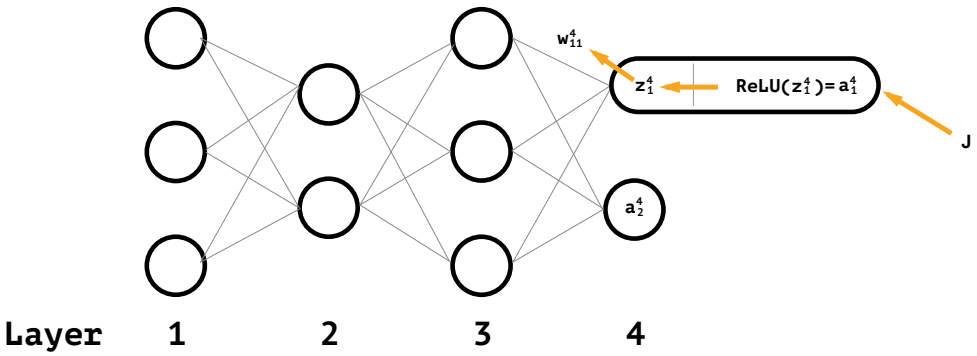

Figure 2.1: Backpropagation to get $\frac{\partial J}{\partial w_{11}^4}$

When shallowly looking at J, it is a function of the activations in the last layer and the true values of the given training example: $J(a_1^4, a_2^4, y_1, y_2)$. E.g. if J is the mean squared error, we have: $J(a_1^4, a_2^4, y_1, y_2) = 1/2 * (a_1^4 - y_1)^2 + 1/2 * (a_2^4 - y_2)^2$. The true

values y_1 and y_2 are just given values which don't depend on anything. But a_1^4 and a_2^4 are functions again. Here we focus on a_1^4, because we are only interested in how w_{11}^4 influences the loss and w_{11}^4 does not have any influence on a_2^4, so we can neglect it. The activation a_1^4 is determined by z_1^4, the linear combination of previous activations that goes into the activation function of layer 4: $a_1^4(z_1^4)$ $[= ReLU(z_1^4)]$. The weight w_{11}^4 is one of the weights of z_1^4, so $z_1^4(w_{11}^4)$. In total we have the following chain of functions when only considering inputs to J that are influenced by w_{11}^4:
$J(a_1^4(z_1^4(w_{11}^4)))$.
The partial derivative is (using the chain rule):
$\frac{\partial J}{\partial w_{11}^4} = \frac{\partial J}{\partial a_1^4}\frac{\partial a_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial w_{11}^4}$.

To demonstrate backpropagation in full length, we focus on a weight in the first layer with weights - the second layer (the input layer typically doesn't have weights, also not in this example). This means we need to propagate as deep into the neural network as possible, coming from behind. This is done in full lengths to demonstrate where repetitions occur and which partial derivatives are needed. Figure 2.2 depicts the process graphically.

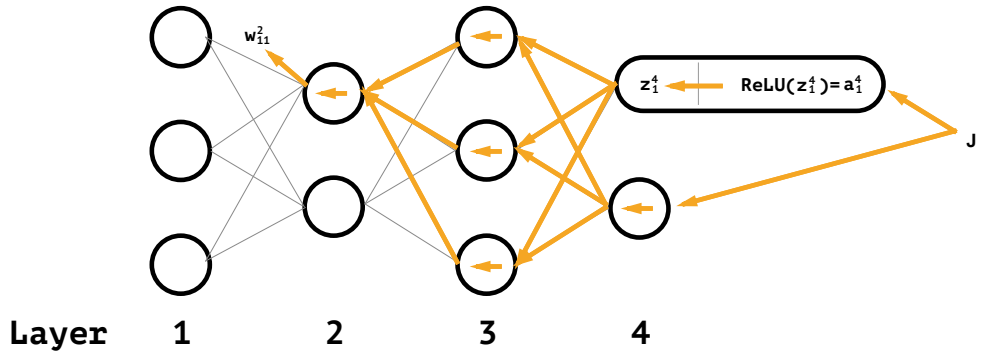

Figure 2.2: Backpropagation to get $\frac{\partial J}{\partial w_{11}^2}$

We start with the last layer: $J(w_{11}^2) = J(a_1^4, a_2^4)$, because w_{11}^2 has an influence on both of the activations a_1^4 and a_2^4 through a chain of functions. In turn, a_1^4 is a function of z_1^4: $a_1^4(z_1^4)$ since z_1^4 directly influences a_1^4 trough $a_1^4 = ReLU(z_1^4)$. Furthermore, z_1^4 is a function of all activations in the previous layer: $z_1^4(a_1^3, a_1^3, a_1^3)$ because all activations in layer 3 have a direct influence on z_1^4 trough $z_1^4 = w_{11}^4 a_1^3 + w_{12}^4 a_1^3 + w_{13}^4 a_1^3 + b_1^4$. This chain of functions continues. The full function chain is given in equation 2.1.

$$
\begin{aligned}
J(w_{11}^2) = J(a_1^4(z_1^4(\ & a_1^3(z_1^3(\ a_1^2(z_1^2(w_{11}^2)))),\\
& a_2^3(z_2^3(\ a_1^2(z_1^2(w_{11}^2)))),\\
& a_3^3(z_3^3(\ a_1^2(z_1^2(w_{11}^2)))))),\\
a_2^4(z_2^4(\ & a_1^3(z_1^3(\ a_1^2(z_1^2(w_{11}^2)))),\\
& a_2^3(z_2^3(\ a_1^2(z_1^2(w_{11}^2)))),\\
& a_3^3(z_3^3(\ a_1^2(z_1^2(w_{11}^2))))))) \qquad (2.1)
\end{aligned}
$$

The partial derivative is (using the chain rule):

$$
\begin{aligned}
\frac{\partial J}{\partial w_{11}^2} = \frac{\partial J}{\partial a_1^4}\frac{\partial a_1^4}{\partial z_1^4}(&\frac{\partial z_1^4}{\partial a_1^3}\frac{\partial a_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}+\\
&\frac{\partial z_1^4}{\partial a_2^3}\frac{\partial a_2^3}{\partial z_2^3}\frac{\partial z_2^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}+\\
&\frac{\partial z_1^4}{\partial a_3^3}\frac{\partial a_3^3}{\partial z_3^3}\frac{\partial z_3^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2})\\
+&\\
\frac{\partial J}{\partial a_2^4}\frac{\partial a_2^4}{\partial z_2^4}(&\frac{\partial z_1^4}{\partial a_1^3}\frac{\partial a_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}+\\
&\frac{\partial z_1^4}{\partial a_2^3}\frac{\partial a_2^3}{\partial z_2^3}\frac{\partial z_2^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}+\\
&\frac{\partial z_1^4}{\partial a_3^3}\frac{\partial a_3^3}{\partial z_3^3}\frac{\partial z_3^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}) \qquad (2.2)
\end{aligned}
$$

We can see that the following 4 types of derivatives are required to compute $\frac{\partial J}{\partial w_{11}^2}$ or similar derivatives of J with respect to a weight:

1. We need to derive the loss function J with respect to all activations of the output layer, here: $\nabla_{\boldsymbol{a}^4}\boldsymbol{J} = (\frac{\partial J}{\partial a_1^4}, \frac{\partial J}{\partial a_2^4})'$. If the loss function was chosen to be the mean squared error: $J(a_1^4, a_2^4, y_1, y_2) = \frac{1}{2}(a_1^4 - y_1)^2 + \frac{1}{2}(a_2^4 - y_2)^2$, this would mean:

$$\nabla_{\boldsymbol{a}^4}\boldsymbol{J} = (a_1^4 - y_1,\ a_2^4 - y_2)'.$$

2. We need the derivatives of z_i^l with respect to the activations of the previous layer. These are just the weights.

$$\frac{\partial z_i^l}{\partial a_j^{l-1}} = w_{ij}^l$$

3. We need $\frac{\partial a_i^l}{\partial z_i^l}$, which are the derivatives of the activation function of layer l, so they are identical at least for each layer. Since we chose the same activation function for all layers in this example, we have:

$$\frac{\partial a_1^4}{\partial z_1^4} = \frac{\partial a_2^4}{\partial z_2^4} \quad = \quad \frac{\partial a_1^3}{\partial z_1^3} = \frac{\partial a_2^3}{\partial z_2^3} = \frac{\partial a_3^3}{\partial z_3^3} \quad = \quad \frac{\partial a_1^2}{\partial z_1^2} = \frac{\partial a_2^2}{\partial z_2^2} = ReLU'(z_i^l) = \begin{cases} 0, & \text{if } z_i^l \leq 0 \\ 1, & \text{if } z_i^l > 0 \end{cases}.$$

4. And we need the derivative of z_i^l with respect to the weight of interest: $\frac{\partial z_i^l}{\partial w_{ij}^l}$. This is just the activation corresponding to that weight. Here:

$$\frac{\partial z_1^2}{\partial w_{11}^2} = a_1^2.$$

These 4 types of derivatives are everything required to partially derive J with respect to all weights in the neural network. While equation 2.2 looks like a lot of functions are required, only MSE' and ReLU' are required to be coded once. The other derivatives are just constant values - either a weight or an activation. Furthermore, ReLU' appears 26 times in equation 2.2 in the form of $\frac{\partial a_i^l}{\partial z_i^l}$. These derivatives can only take the value 1 or 0, which doesn't complicate the products as much as it might seem at first. When the value is 0 sometimes (a neuron didn't "fire"), the corresponding products can be kicked out, further simplifying $\frac{\partial J}{\partial w_{11}^2}$. This is done in the following example to compute an actual value for $\frac{\partial J}{\partial w_{11}^2}$.

If we take equation 2.2 and imply that $\frac{\partial a_2^3}{\partial z_2^3} = \frac{\partial a_3^3}{\partial z_3^3} = \frac{\partial a_2^4}{\partial z_2^4} = 0$ because the ReLU didn't fire in those neurons, then the equation simplifies to $\frac{\partial J}{\partial w_{11}^2} = \frac{\partial J}{\partial a_1^4}\frac{\partial a_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial a_1^3}\frac{\partial a_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{22}^2}$. In such a case, $\frac{\partial J}{\partial w_{11}^2}$ is computed as follows:

- $\frac{\partial z_1^2}{\partial w_{11}^2} = a_1^1 = 0.3$, assuming $a_1^1 = x_1^{(ex\,i)} = 0.3$ in this example. In the MNIST example, this would mean the first pixel of the handwritten digit image is dark gray.
- $\frac{\partial a_1^2}{\partial z_1^2} = 1$, we skip the calculation of z_1^2 and just assume that the linear combination of all activations of layer 1 (in MNIST example the brightness values of pixels) is $z_1^2 = 42.0$ so $ReLU'(42.0) = 1$.
- $\frac{\partial z_1^3}{\partial a_1^2} = w_{11}^3 = 2.5$, assuming this is the weight in the current training algorithm iteration j. It is an element of $\boldsymbol{\theta}^{(it\,j)}$.
- $\frac{\partial a_1^2}{\partial z_1^3} = 1$, assuming $z_1^3 > 0$ so that $ReLU'(z_1^3) = 1$.
- $\frac{\partial z_1^4}{\partial a_1^3} = w_{11}^4 = 4.2$, assuming this is the weight in the current training algorithm iteration j. It is an element of $\boldsymbol{\theta}^{(it\,j)}$.
- $\frac{\partial a_1^3}{\partial z_1^4} = 1$, assuming $z_1^4 > 0$ so that $ReLU'(z_1^4) = 1$.
- $\frac{\partial J}{\partial a_1^4} = 0.2$ assuming $a_1^4 - y_1 = 0.2$

$\Rightarrow \underline{\underline{\frac{\partial J}{\partial w_{11}^2}}} = 0.3 * 1 * 2.5 * 1 * 4.2 * 1 * 0.2 = \underline{\underline{0.63}}$.

If we are at the point $\boldsymbol{\theta}^{(it\,j)}$, evaluate the training example $(\boldsymbol{x}^{(ex\,i)}, \boldsymbol{y}^{(ex\,i)})$ and increase w_{11}^2 by 0.1, then the loss approximately increases by 0.063. Judging by only this specific training example j, w_{11}^2 should therefore be decreased. If we would train the neural net always with only one training example i, it would just learn to always output something

close to $\boldsymbol{y}^{(ex\,i)}$, which would be a strong over fitting. Therefore, the gradients in a given iteration j corresponding to many training examples is computed by averaging the gradients that correspond to single training examples. $(\frac{\partial J}{\partial w_{11}^2})^{(it\,j)} = \frac{1}{m} * ((\frac{\partial J}{\partial w_{11}^2})^{(ex\,1)} + ... + (\frac{\partial J}{\partial w_{11}^2})^{(ex\,m)})$. Using some example values, this could be $(\frac{\partial J}{\partial w_{11}^2})^{(it\,j)} = \frac{1}{100} * (0.63 + ... + (-0.42)) = -0.34$. When evaluating several training examples, we can see that it makes more sense to increase w_{11}^2 so that the loss becomes lower on average.

Chapter 3

Optimization Algorithms

Optimization is the process of finding a maximum or minimum of a function. In the case of neural networks, the loss function J has to be optimized, more precisely, it has to be minimized. This is done iteratively as given by all steps in chapter 2.1. We will focus on step 7), the computation of the update of theta $\Delta\boldsymbol{\theta}^{(it\,j)}$, to be used within the given iteration j.

3.1 Stochastic Gradient Descent

"Nearly all of deep learning is powered by one very important algorithm: stochastic gradient descent (SGD). Stochastic gradient descent is an extension of the gradient descent algorithm" (Goodfellow, 2016). Gradient descent is a first order optimization algorithm, meaning it only needs the first order derivative of J - the gradient $\nabla_{\boldsymbol{\theta}} J$ - and no Hessian is required. It is the basis of stochastic gradient descent, which in turn is the basis of many other first order optimization algorithms such as RMSProp, Adam or AdaGrad.

Gradient Descent Step 7) in chapter 2.1 generally describes a step of an optimization algorithm. In gradient descent, the update $\Delta\boldsymbol{\theta}^{(it\,j)}$ is computed in the following way:

$$\Delta\boldsymbol{\theta}^{(it\,j)} = lr * -\nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)}).$$

$\nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)})$ was defined in step 6 in chapter 2.1. It is the average of all gradients of the loss functions with respect to $\boldsymbol{\theta}$ with each of the training examples as the parameters respectively: $\nabla J^{(it0)}(\boldsymbol{\theta}^{(it0)}) = 1/m * \nabla J^{(ex0)}(\boldsymbol{\theta}^{(it0)}) + \nabla J^{(ex1)}(\boldsymbol{\theta}^{(it0)}) + ... + \nabla J^{(ex\,m)}(\boldsymbol{\theta}^{(it0)})$. It points to the direction of the steepest ascent of the averaged loss function (average of the loss functions with each of the training examples as the parameters respectively) when "standing" at point $\boldsymbol{\theta}^{(it\,j)}$. Therefore, the negative of that gradient points to the steepest descent of a differentiable function. The norm of the gradient, as well as the freely defined constant "lr", called the learning rate, determine how far the current step (of iteration j) goes in that direction.

The update is applied by adding it to the current point $\boldsymbol{\theta}^{(it\,j)}$ in order to create the updated point $\boldsymbol{\theta}^{(it\,j+1)}$ for the next iteration:

$$\boldsymbol{\theta}^{(it\,j+1)} = \boldsymbol{\theta}^{(it\,j)} + \Delta\boldsymbol{\theta}^{(it\,j)}.$$

Code implementation of gradient descent can be found in the file "gd_rmsprop.ipynb" belonging to this report (Appendix A). Furthermore, an inspection of how a gradient descent optimizer is implemented in tensorflow and keras is given in chapter 4.

Stochastic Gradient Descent In deterministic gradient descent ("gradient descent") we assume that all training examples of a training set are used in each update step to compute the loss and the gradient of it. In stochastic gradient descent, the training set is shuffled randomly (hence "stochastic") and divided into subsets, called mini-batches. For each update step, a new mini-batch is used to compute the loss and the gradient of it (unless all mini-batches have been used and the circle starts again with a new epoch). Calculating $\nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)})$ with less training examples is computationally cheaper, which is the advantage of stochastic gradient descent over gradient descent. The disadvantage is that $\nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)})$ as derived from the loss according to a mini-batch of length m_1 - $J(\boldsymbol{\theta}^{(it\,j)} \mid \boldsymbol{x}^{ex\,0}, ..., \boldsymbol{x}^{ex\,m_1})$ - does not necessarily point at the direction of the steepest descent of the loss according to all m_{all} training examples ($m_{all} > m_1$): $J(\boldsymbol{\theta}^{(it\,j)} \mid \boldsymbol{x}^{ex\,0}, ..., \boldsymbol{x}^{ex\,m_{all}})$. The advantage of being computationally cheaper usually outweighs the disadvantage of not precisely going in the direction of the steepest descent. With each update step of stochastic gradient descent being computationally cheap, many update steps can be done in the same time as a single update step of gradient descent, which usually leads to a faster conversion to a local minimum of J.

3.2 RMSProp

RMSProp is one example of an extension of stochastic gradient descent. In each optimization step, we first need to compute the accumulated square gradient (Goodfellow, 2016):

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\ \nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)}) \odot \nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)})$$

ρ is the decay rate. The smaller ρ, the less weight do the elementwise squared gradients of past iterations have and the more weight does the elementwise square of the gradient of the current iteration have.
The update is calculated in the following way (Goodfellow, 2016):

$$\Delta\boldsymbol{\theta}^{(it\,j)} = -\frac{lr}{\sqrt{\delta + \boldsymbol{r}}} \odot \nabla J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)}).$$

$\frac{lr}{\sqrt{\delta+\boldsymbol{r}}}$ is applied element-wise. lr is the learning rate. δ is a small constant, i.e. 0.00000001, to stabilize the division, since some elemnts of $\boldsymbol{r}$ might be zero. We can see that RMSProp is identical to gradient descent, except that there is not only one constant learning rate multiplied with all elements of $J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)})$. Instead, $\frac{lr}{\sqrt{\delta+\boldsymbol{r}}}$ as a whole can be seen as a vector of learning rates which contains separate learning rates for each element of $J^{(it\,j)}(\boldsymbol{\theta}^{(it\,j)})$. And these learning rates also change between the update steps, whereas the learning rate in gradient descent doesn't change over the iterations. Therefore, RMSProp allows for more flexibility in order to determine the lengths along each axes of a single update step. Furthermore, information of previous mini-batches can be

made use of through $\boldsymbol{r}$, wich is not possible in stochastic gradient descent.

Code implementation of RMSProp can be found in the file "gd_rmsprop.ipynb" belonging to this report (Appendix A).

3.3 Second Order Optimization Algorithm

The second order Newton method, which has the Taylor Expansion as the basis, is the theoretical foundation for many second order optimization algorithms. A code implementation of Taylor Expansions and Newton methods can be found in the file "taylor_newton.ipynb" belonging to this report (Appendix B).

The problem is that the Hessian, or more precisely, the inverse of the Hessian of J is required for second-order optimization algorithms, which is not computed by a standard backpropagation as described in step 5) in chapter 2.1. It is very computationally expensive to obtain the inverse of the exact Hessian. This is why there is a group of algorithms which are called Quasi-Newton Methods (e.g. BFGS). They only use an approximation of the inverse of the Hessian, which isn't as computationally expensive as the inverse of the exact Hessian. However, even the computation of an approximate inverse of the Hessian is expensive and is an additional expense to obtaining the gradient. While a single step of a second-order method might be a more accurate (and more strategic) step towards a minimum, first-order methods might be able to do several (less accurate) steps in the same time. Neither TensorFlow, nor Keras have a second order optimizer implemented.

Chapter 4

Implementation Of Optimization Algorithms In TensorFlow And Keras

4.1 Motivation And Background

The software libraries TensorFlow and Keras are commonly used to create and use neural nets. Both of them are open source and have a repository on GitHub. Both of them have optimizer classes defined in Python that can be used to minimize the loss function of a neural net. This chapter tries to close the gap between the mathematical and verbal definitions of the optimization algorithms in chapter 3 and their practical use in optimizer classes in two common machine learning libraries. Seeing the code implementations might lead to a better understanding of the algorithms in chapter 3. Furthermore, this chapter is meant to be a first orientation when the goal is to write a custom optimizer subclass to be fluently used with one of these libraries, as the official documentation of TensorFlow and Keras optimizers is sparse. It is recommended to read this chapter in the PDF version of this report, as clickable hyperlinks are given.

We refer to the the code in chapter 1.2. In that example, optimizers have to be specified in line 22 in the model.compile() function. The specified optimizer is used by the fit() function of a tf.keras.Sequential model (in line 27).

4.2 TensorFlow Implementation

TensorFlow's GitHub repository can be found at `github.com/tensorflow/tensorflow`. Any file directories given in this section imply this GitHub repository as the parent directory. This report refers to the commit bd36b48c555b2d46c41a179ed9f27a04806e9e66 from February 16 2019. The base class "Optimizer" is defined in tensorflow/train/optimizer.py. Subclasses of that are each defined in a seperate file, e.g. the "GradientDescentOptimizer" subclass of "Optimizer" is defined in tensorflow/train/gradient_descent.py.

What happens when model.fit() in line 27 in the code in chapter 1.2 is run and the "GradientDescentOptimizer" optimizer was chosen? The chain of function calls is as follows: When a model is fitted, the "apply_gradients()" function of a GradientDescentOptimizer instance is called. Depending on the context, this happened either through a call of the "minimize()" function of that instance or directly. These functions are defined in the superclass "Optimizer" (optimizer.py) starting in the lines 511 and 354. "apply_gradients()" calls the function "update_op()" of an instance of one of the processor classes which are also defined in optimizer.py, in lines 104 - 193. Depending on the processor, "update_op()" calls either "_apply_dense()" or "_resource_apply_dense()". These functions have to be overwritten in all "Optimizer" subclasses. In the case of GradientDescentOptimizer, this is done in gradient_descent.py. Both of these overwritten functions call training_ops.apply_gradient_descent(). tensorflow/train/training_ops.py is just a Python wrapper file for the C++ code in tensorflow/tensorflow/core/kernels/training_ops.cc (CPU Implementation) and tensorflow/tensorflow/core/kernels/training_ops_gpu.cu.cc (GPU Implementation). The struct ApplyGradientDescent in these files finally gives the code for a gradient descent update. Excerpt from training_ops.cc:

```
47 namespace functor {
48 template <typename T>
49 struct ApplyGradientDescent<CPUDevice, T> {
50   void operator()(const CPUDevice& d, typename TTypes<T>::Flat var,
51                   typename TTypes<T>::ConstScalar lr,
52                   typename TTypes<T>::ConstFlat grad) {
53     var.device(d) -= grad * lr();
54   }
55 };
```

In line 53, we can see the gradient descent update. Many other optimization algorithims are also in these training_ops*.cc files, e.g. as given in the structs "ApplyAdagrad" or "ApplyRMSProp".

4.3 Keras Implementation

Keras' GitHub repository can be found at `github.com/keras-team/keras` . Any file directories given in this section imply this GitHub repository as the parent directory. This report refers to the commit 9bd69784e31f17ee3501167c267dad5c64f25de8 from March 20 2019. The base class "Optimizer", as well as subclasses of it like "SGD", are defined in keras/optimizers.py.

What happens when model.fit() in line 27 in the code in chapter 1.2 is run and the "SGD" optimizer was chosen? The chain of function calls is as follows: When a model is fitted, the "get_updates()" function of an SGD instance is called. The "get_updates()" function directly contains python code that represents a (stochastic) gradient descent step. The function also supports a number of extensions to the gradient descent algorithm.

```
183 def get_updates(self, loss, params):
184   grads = self.get_gradients(loss, params)
185   self.updates = [K.update_add(self.iterations, 1)]
186
187   lr = self.lr
188   if self.initial_decay > 0:
```

```
189   lr = lr * (1. / (1. + self.decay * K.cast(self.iterations,
190   K.dtype(self.decay))))
191   # momentum
192   shapes = [K.int_shape(p) for p in params]
193   moments = [K.zeros(shape) for shape in shapes]
194   self.weights = [self.iterations] + moments
195   for p, g, m in zip(params, grads, moments):
196     v = self.momentum * m - lr * g  # velocity
197     self.updates.append(K.update(m, v))
198
199     if self.nesterov:
200       new_p = p + self.momentum * v - lr * g
201     else:
202       new_p = p + v
203
204     # Apply constraints.
205     if getattr(p, 'constraint', None) is not None:
206       new_p = p.constraint(new_p)
207
208     self.updates.append(K.update(p, new_p))
209   return self.updates
```

In line 184 we can see how the gradient is obtained with the "get_gradients()" function from the "Optimizer" superclass. The function "get_gradients()" is an implementation of backpropagation. Line 196 is where the optimization algorithm's update is calculated.

```
196     v = self.momentum * m - lr * g
```

becomes

```
    v = - lr * g
```

when we just use the basic gradient descent algorithm and don't use momenti. The update is added to the current point in line 202. This represents step 8 in chapter 2.1: $\boldsymbol{\theta}^{(it\,j+1)} = \boldsymbol{\theta}^{(it\,j)} + \Delta\boldsymbol{\theta}^{(it\,j)}$.

Bibliography

Goodfellow I., Bengio Y., C. A. (2016). *Deep Learning*. The MIT Press Cambridge, Massachusetts, London, England.

Appendices

Appendix A

Python Implementation: Gradient Descent And RMSProp

Please see the next page for a print of the Jupyter notebook file "gd_rmsprop.ipynb" which belongs to this report.

gd_rmsprop

March 30, 2019

1 Simple Code Implementation Of Gradient Descent And RMSProp

1.1 Loss Function, Gradient, Starting Values

```
In [4]: import numpy as np
```

In the following is a simple loss function for example purposes only: J(**theta**) = theta_1 * theta_2 * ... (multiplication of all elements). The loss function J is determined by the real values **y** and the predictions/classifications **y_hat**, which is a function of the predictors **x** (the data), and the parameters **theta** of the prediction model. E.g. loss J = mean(abs(f(**x**, **theta**) - **y**)) would be the Mean Absolute Error. To keep the example simple, we just assume a certain influence of **theta** without explaining how exactly **theta** effects the loss through the model f(**x**, **theta**) = **y_hat**. The given loss funciton does not converge to a minimum, but it has a saddle point at (0,0,...). In this example, the goal is to get close to the saddle point (while the real life goal with more complex loss functions is to get close to a minimum).

```
In [1]: # Loss Function J
        def J(theta):
            loss = np.prod(theta) #multiply all elements of theta, e.g. theta1*theta2*...
            return loss
```

```
In [2]: # Gradient Of J
        def dJ(theta):
            gradient = theta * 0
            for i in range(len(theta)):
                gradient[i] = np.prod(theta[np.arange(len(theta)) != i])
            return gradient
```

```
In [5]: # Setting The Starting Values Of Theta, Getting Loss And Gradient At Theata_Start
        theta_start = np.array([2., 2.])
        print(J(theta_start))
        print(dJ(theta_start))
```

```
4.0
[2. 2.]
```

1.2 Algorithms: Gradient Descent And RMSProp

```
In [6]: def gradient_descent(J, dJ, theta_start, steps = 100, lr = 0.001):
            theta = np.copy(theta_start)
            loss = []
            for _ in range(steps):
                gradient = dJ(theta)
                theta -= lr * gradient
                loss.append(J(theta))

            return theta, loss
```

```
In [7]: def rmsprop(J, dJ, theta_start, steps = 100, lr = 0.001, decay = .9, delta = 1e-6):
            theta = np.copy(theta_start)
            loss = []
            gradient_mean_sqr = np.zeros(theta.shape, dtype=float) # vector full of zeros

            for _ in range(steps):
                gradient = dJ(theta)
                gradient_mean_sqr = decay * gradient_mean_sqr + (1 - decay) * gradient ** 2
                # **2 means elementwise ^2
                theta -= lr * gradient / (np.sqrt(gradient_mean_sqr) + delta)
                # np.sqrt meanselementwise square roots
                # / means elementwise division
                loss.append(J(theta))

            return theta, loss
```

"gradient_mean_sqr" is always positive, but might have values equal or close to zero. "delta" is only there to stabilize the divisions by "gradient_mean_sqr".

1.3 Usage And Comparison

```
In [8]: theta_opt1, loss1 = gradient_descent(J, dJ, theta_start, steps = 5, lr = .5)
        print(loss1)
        print(theta_opt1)
```

```
[1.0, 0.25, 0.0625, 0.015625, 0.00390625]
[0.0625 0.0625]
```

```
In [9]: theta_opt2, loss2 = rmsprop(J, dJ, theta_start, steps = 5, lr = .5)
        print(loss2)
        print(theta_opt2)
```

```
[0.17544677397202935, 0.006086804167495761, 0.00012448839791948842, 1.1634232222190242e-06, 2.65
[5.15687064e-05 5.15687064e-05]
```

In both cases, 5 iterations were done. We can see the loss getting smaller and smaller with each iteration. RMSProp reached a point closer to the saddle point (0,0) compared to Gradient Descent.

Appendix B

Python Implementation: Taylor Expansion And Newton Algorithm

Please see the next page for a print of the Jupyter notebook file "taylor_newton.ipynb" which belongs to this report.

taylor_newton

March 30, 2019

1 Taylor Expansion And Newton's Method

1.1 Taylor Expansion

Estimation of a function value at a given point, coming from a starting point with known function value. E.g. we are at x = (2, 3)' and know f(2, 3) = 20, we wonder what f(3, 4) is.

1.1.1 First-Order Taylor Expansion (repetition from last year's Statistical Inference lecture, chapter 2)

For linear functions (planes), an exact determination of a function value at a given point is possible with a first-order Taylor expansion

In [42]:
```
import numpy as np
```

In [43]:
```
x_start = np.array([2, 3])
```

In [44]:
```
# Some linear function: 5 + 3x1 + 3x2 + ...
def lin_f(x):
    function_value = 5 + np.sum(3 * x) # slope is 3 in all dimensions
    return function_value

# example output
lin_f(x_start)
```

Out[44]:
```
20
```

In [45]:
```
# Gradient: (3, 3, ...)'
def dlin_f(x):
    return x * 0 + 3 # x * 0 is only to get a 0 vector of same dimension as x

# example output
dlin_f(x_start)
```

Out[45]:
```
array([3, 3])
```

In [46]:
```
# Taylor expansion (first-order)
x_sought = np.array([3, 4])
```

```
def taylor1(x_start, f, df, x_sought):
    f_sought_est = f(x_start) + df(x_start) @ (x_sought - x_start)
    return f_sought_est

#check
estimation = taylor1(x_start, lin_f, dlin_f, x_sought)
print(estimation)
print(lin_f(x_sought))
```

```
26
26
```

We could exactly "estimate" lin_f(3, 5) with a first-order Taylor expansion.

1.1.2 Second-Order Taylor Expansion (this is used in the Goodfellow 2016 book)

For quadratic functions, an exact determination of a function value at a given point is possible with a second-order Taylor expansion

In [47]:
```
# Some quadratic function: 5 + 3x1^2 + 3x2^2 + ...
def quad_f(x):
    function_value = 5 + np.sum(3 * x**2)
    return function_value

# example output
quad_f(x_start)
```

Out[47]:
```
44
```

In [48]:
```
# Gradient: (6x1, 6x2, ...)'
def dquad_f(x):
    return x * 6

# example output
dquad_f(x_start)
```

Out[48]:
```
array([12, 18])
```

In [49]:
```
# Hessian:
def ddquad_f(x):
    identity_mat = np.eye(len(x), dtype=float)
    return identity_mat * 6

# example output
ddquad_f(x_start)
```

Out[49]:
```
array([[6., 0.],
       [0., 6.]])
```

```
In [50]: # Taylor expansion (second-order)
         x_sought = np.array([3, 4])

         def taylor2(x_start, f, df, ddf, x_sought):
             delta = (x_sought - x_start)
             f_sought_est = (f(x_start) + df(x_start) @ delta
                             + 0.5 * delta @ ddquad_f(x_start) @ delta)
             return f_sought_est

         #check
         estimation = taylor2(x_start, quad_f, dquad_f, ddquad_f, x_sought)
         print("second-order Taylor estimation: " + str(estimation))
         print("real value: quad_f(x_new) = : " + str(quad_f(x_sought)))

         #estimate quadratic function with a first-order taylor expansion
         print("first-order Taylor estimation: " + str(taylor1(x_start, quad_f,
                                                              dquad_f, x_sought)))
```

```
second-order Taylor estimation: 80.0
real value: quad_f(x_new) = : 80
first-order Taylor estimation: 74
```

We could exactly "estimate" quad_f(3, 5) = 80 with a second-order Taylor expansion. The result of the first-order Taylor expansion is just an approximation.

-> for higher-order polynomials or non-plynomial-functions the Taylor estimation can be off, especially when delta is big, e.g. if the function value of a point is estimated that is far away from the current/starting point.

1.2 Newton's Method

The Newton algorithm seeks the root, we want an x_sought where f(x_sought) = 0. We can set the Taylor estimation of f(x_sought) to zero and then analytically determine the x_sought point where f(x_sought) is estimated to be zero according to the Taylor expansion.

If the target function is a polynomial and the Taylor expansion has at least the order of that polynomial, the Newton Algorithm finds the exact root with one step. But for most target functions (non-plynomial or polynomial of higher-order), the point where the root was estimated to be turns out to not (exactly) be the root of the function. The estimated x_sought is now seen as the new staring point x_start and another step of the newton algorithm is started. We do as many steps until the estimations for the root don't change much between steps.

```
In [51]: # Example on the given quadratic function
         # Determination of the exact root of a second-order Taylor series
         # (as used in the Goodfellow 2016 book)
         # step 1 (all we need for a quadratic function):
         x_sought = x_start - np.linalg.inv(ddquad_f(x_start)) @ dquad_f(x_start)
         x_sought
```

```
Out[51]: array([0., 0.])
```

```
In [52]: quad_f(np.array([0, 0]))
```

```
Out[52]: 5
```

The second-order Newton algorithm only needed one step for the best possible result for the given quadratic function. 5 is not 0, so (0, 0)' is not a root of quad_f, but it is the lowest possible point due to the intercept of 5. The full second-order Newton algorithm that allows for multiple steps is as follows:

```
In [53]: def newton2(x_start, df, ddf, stop_delta = 0.1):
             # Stopping criterion: stop when the last two estimated points are close
             # to each other. E.g. when the Euclidian distance between the last two
             # estimated points (estimation_delta) is smaller or equal to a defined
             # stop_delta

             # Initialization, just to get the while loop running
             estimation_delta = stop_delta + 1

             while estimation_delta > stop_delta:
                 # a step
                 print("step")

                 x_sought = x_start - np.linalg.inv(ddf(x_start)) @ df(x_start)

                 # result comparison with previous step
                 estimation_delta = np.sqrt(x_start @ x_sought) # here: Euclidian distance

                 # x_sought becomes the x_start of the next step:
                 x_start = x_sought

             return x_sought
```

```
In [54]: newton2(np.array([5, 3]), dquad_f, ddquad_f)
```

```
step
```

```
Out[54]: array([0., 0.])
```

After only 1 step, the second-order Newton algorithm reached the global minimum of the quadratic function quad_f().

Appendix C

Used Resources And Disclaimer

I hereby declare that this report was created by myself. Work of others has been referenced. For many parts of the report, the book "Deep Learning" by Goodfellow et. al 2016 was used as a source of orientation or direct quotation (indicated in the report). The series "Neural Networks" from the YouTube channel "3Blue1Brown" was also often used as a source of orientation. The following software libraries were used: keras, tensorflow, numpy.

Göttingen, 31 March 2019: