Assignment

Practical Assignment #4

Felix Strobel und Gabriel Cevallos

June 2, 2021

Examiner: Prof. Dr. Peter Krzystek

Contents

List of Figures

SGD	Stochastic gradient descent
RMSProp	Root mean square prop
ADAM	Adaptive momentum estimation

1 Input Data

1.1 Problem 1: Simple network

In the first problem a small neural network with 4 Layers was provided. In addition

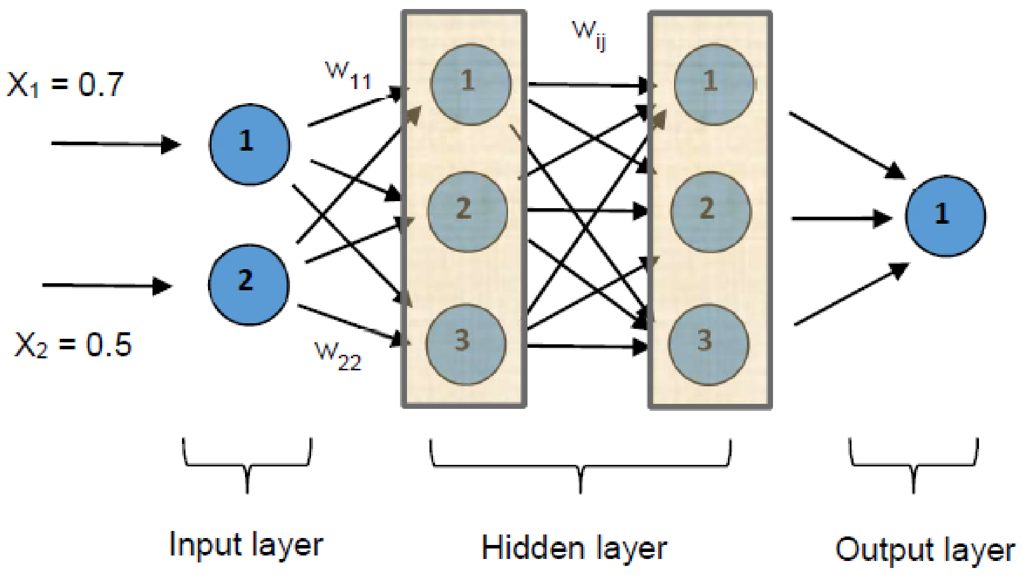
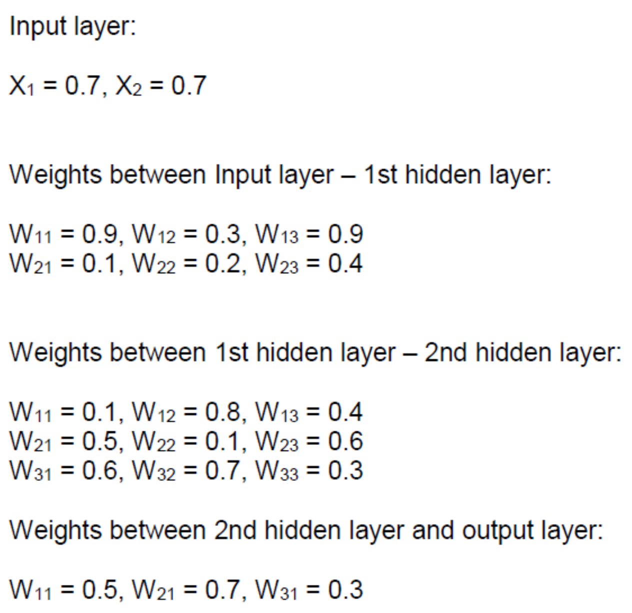


Figure 1.1: Simple neural network for problem 1

the input values and the weights for the connections of the fully connected layers were provided.

1.2 Problem 2: Backpropagation

Similarly to the first problem a small neural network was provided, in this case the network consists only of a single neuron with two inputs and an activation function Z.



Input layer:

$X_1 = 0.7$, $X_2 = 0.7$

Weights between Input layer – 1st hidden layer:

$W_{11} = 0.9$, $W_{12} = 0.3$, $W_{13} = 0.9$
$W_{21} = 0.1$, $W_{22} = 0.2$, $W_{23} = 0.4$

Weights between 1st hidden layer – 2nd hidden layer:

$W_{11} = 0.1$, $W_{12} = 0.8$, $W_{13} = 0.4$
$W_{21} = 0.5$, $W_{22} = 0.1$, $W_{23} = 0.6$
$W_{31} = 0.6$, $W_{32} = 0.7$, $W_{33} = 0.3$

Weights between 2nd hidden layer and output layer:

$W_{11} = 0.5$, $W_{21} = 0.7$, $W_{31} = 0.3$

Figure 1.2: Data provided for problem 1

1.3 Problem 3: Artificial neural network

For the third problem a small neural network with 3 Layers was provided in form of a python script using the Pytorch library, in addition to the network a dataset called **One-hundred plant species leaves data set Data Set**. This dataset contains the leaves of 100 different plant species each represented with 16 examples of real leaves, which are themselves represented by a feature vector with 64 different elements representing different features.

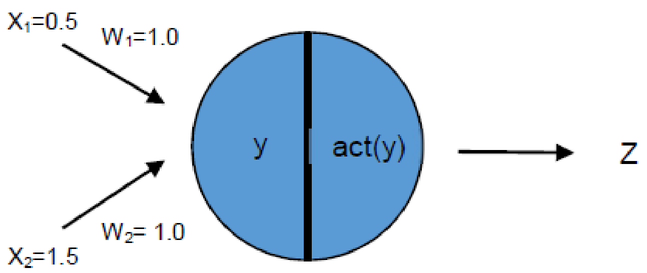
1.4 Problem 4: Gradient Descent

The input data set consists of 1000 generated points with a X value between -5 and 5. The Y value is calculated with the formula $y = mx + c$ where m and c are assumed by us. Applying a random noise leads to the dataset which is shown in Figure 3.1

Further a jupyter notebook is provided which generate these data and provides a script which only need to be completed with the calculation of the gradient and updating and weight update for m and c.

It is also given that all data points should satisfy this equation.

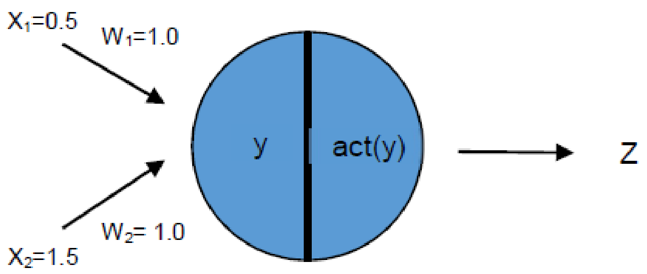
$$y = mx + c \tag{1.1}$$



$$y = w_1 \cdot X_1 + w_2 \cdot X_2 + b$$

Z = sigmoid(y)
b = 2.0
Z' = 1.0

Figure 1.3: Single neuron for problem 2



$$y = w_1 \cdot X_1 + w_2 \cdot X_2 + b$$

Z = sigmoid(y)
b = 2.0
Z' = 1.0

Figure 1.4: Generated input data of problem 4

2 Methods

2.1 Problem 1: Simple network

The task of problem 1 is to calculate the output of the given network using matrix operations. In order to execute the operations firstly the operation should be defined. It can be formulated as $x_j = f(x_i * w_{ij})$ with x_j being the output of layer j, x_i being the output of layer i, w_{ij} being the weight matrix for the connections between the layers i and j and $f()$ being the activation function of layer j which in our case is the sigmoid function.

2.2 Problem 2: Backpropagation

The task of problem 1 is to calculate the back propagation of the given network (section 1.2) for two different cost functions. This is done by first finding the gradient using the partial derivatives of the forward propagation function, taking advantage of the chain rule this can be done layer by layer from end to beginning. This is use to determine the influence of each single weight and bias to the final error, which are updated accordingly to minimize the error.

2.3 Problem 3: Artificial neural network

In problem 3 both of the tasks are combined and put to use for the training of a simple neural network with the help of the Pytorch library. The best suiting combination of network architecture and hyperparameters have to be found empirically by making assumptions and testing them by comparing the results of the different tests.

2.4 Problem 4: Gradient Descent

There are multiple versions of the gradient descent. They are based on the normal gradient descent.

2.4.1 Normal Gradient Descent

Gradient Descent is one optimizer to find better weights in the model. It is based on the loss function J which represents the classification error and the gradients to give the direction to minimize the loss. These gradients are calculated by back propagation. There is also the learning rate λ which controls the size of each step in one direction. With a low learning rate comes high computation times but a to high learning rate can lead to a miss calculation of the minimum.

The normal gradient descent is based on the complete training set and has hight costs with big data sets. The height costs are effected by only taking a single step for one pass.

In short the algorithm of the normal gradient descent is as follows:

- For each epoch:
  - For each weight j:
    - $w_j = w_{j-1} + \delta w_j$

2.4.2 Stochastic Gradient Descent

This can be optimized by using the Stochastic gradient descent (SGD) where the weights are updated after each training sample. The stochastic approximation of the true cost gradient results in a zig-zag path. This only works fine with a convex cost function.

The extension provided by SGD leads to the following algorithm:

- For each epoch:
  - For each training sample
    - For each weight j:
      - $w_j = w_{j-1} + \delta w_j$

2.4.3 Gradient Descent with momentum

The idea behind this extension is to move only for the average of the gradients. This prevent a possible oscillation and speeds up the optimization significantly.

In comparison to the normal gradient descent ($w_t = w_{t-1} - \alpha g_{t-1}$) the weight looks like $w_t = w_{t-1} - \alpha v_t$ with $v_t = \beta v_{t-1} - (1 - \beta) g_{t-1}$ and $\beta = 0.9$.

2.4.4 RMSProp

Another idea is to make the step size inversively proportional to the magnitude of the gradient which is called the Root mean square prop (RMSProp). Doing this it is necessary to introduce a new variable s which is the moving average of squared gradients.

$$s_t = \beta s_{t-1} + (1-\beta) g_{t-1}^2 \tag{2.1}$$

With the weight:

$$w_t = w_{t-1} - \alpha \frac{g_{t-1}}{\sqrt{s_t} + \varepsilon} \tag{2.2}$$

2.4.5 Adaptive moment estimation

The combination of RMSProp and Gradient descent with momentum is called Adaptive momentum estimation (ADAM).

The equations are:

$$v_t = \frac{\beta_1 v_{t-1} - (1-\beta_1) g_{t-1}}{1 - \beta_1^t} \tag{2.3}$$

$$s_t = \frac{\beta_2 s_{t-1} + (1-\beta_2) g_{t-1}^2}{1 - \beta_2^t} \tag{2.4}$$

$$w_t = w_{t-1} - \alpha \frac{v_t}{\sqrt{s_t} + \varepsilon} \tag{2.5}$$

With $\beta_1 = 0.9; \beta_2 = 0.999; \varepsilon = 10^{-8}; \alpha$ to be tuned.

Since the data points should all satisfy the equation $y = mx + c$ in this assignment the gradients for m and c are calculated with the following equations.

$$\frac{\partial l}{\partial m} = -2 \sum_{i=1}^{n} x_i (y_i - m x_i - c) \tag{2.6}$$

$$\frac{\partial l}{\partial c} = -2 \sum_{i=1}^{n} (y_i - m x_i - c) \tag{2.7}$$

Where the residuals are calculated with

$$e_i = y_i - m x_i - c \tag{2.8}$$

And the loss is calculated with

$$m_k = m_{k-1} - \lambda \frac{\partial l}{\partial m} \tag{2.9}$$

$$c_k = c_{k-1} - \lambda \frac{\partial l}{\partial c} \tag{2.10}$$

With λ used as learning parameter.

3 Results

3.1 Problem 1: Simple network

In order to be able to use our forward propagation function $x_j = f(x_i * w_{ij})$(from section 2.1), the given data (from section 1.1) were formulated as matrices resulting in the following data:

- The input vector $x_0 = \begin{bmatrix} 0.7 & 0.5 \end{bmatrix}$
- The weight matrix $w_01 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
- The weight matrix $w_12 = \begin{bmatrix} 0.9 & 0.3 & 0.9 \\ 0.1 & 0.2 & 0.4 \end{bmatrix}$
- The weight matrix $w_23 = \begin{bmatrix} 0.1 & 0.8 & 0.4 \\ 0.5 & 0.1 & 0.6 \\ 0.6 & 0.7 & 0.3 \end{bmatrix}$
- The weight matrix $w_34 = \begin{bmatrix} 0.5 & 0.7 & 0.3 \end{bmatrix}$

Thus the final equation can be summarized as:
$f(\,f\,(\,f(\,f(x_0 * w_01) * w_12) * w_23) * w_34$

The output of the neural network is 0.7451673339899871.

Alternatively the input vector $x_0 = \begin{bmatrix} 0.7 & 0.5 \end{bmatrix}$ was used, resulting in the value 0.7453676512649436.

3.2 Problem 2: Backpropagation

Using the given equations from section 1.2 the following gradients were calculated for a:

- $\partial sigmoid = sigmoid * (1 - sigmoid)$
- $\frac{\partial cost}{\partial prediction} = -1$
- $\frac{\partial prediction}{\partial y} = \partial sigmoid(y)$

- $\frac{\partial y}{\partial w_1} = x_1$
- $\frac{\partial y}{\partial w_2} = x_2$
- $\frac{\partial y}{\partial b} = 1$
- $\frac{\partial cost}{\partial w_1} = \frac{\partial cost}{\partial prediction} * \frac{\partial prediction}{\partial y} * \frac{\partial y}{\partial w_1}$
- $\frac{\partial cost}{\partial w_2} = \frac{\partial cost}{\partial prediction} * \frac{\partial prediction}{\partial y} * \frac{\partial y}{\partial w_2}$
- $\frac{\partial cost}{\partial b} = \frac{\partial cost}{\partial prediction} * \frac{\partial prediction}{\partial y} * \frac{\partial y}{\partial b}$

This results in the following weights (assuming a learning rate of 1):

- $w1 = w1 - \frac{\partial cost}{\partial w_1} = 1.0088313531066455$
- $w2 = w2 - \frac{\partial cost}{\partial w_2} = 1.0264940593199368$
- $b = b - \frac{\partial cost}{\partial b} = 2.017662706213291$

For part b only the cost function(see section 1.2 b)is different with its derivative being:

- $\frac{\partial cost}{\partial prediction} = -(z' - z)$
-

And giving the following weight values: This results in the following weights (assuming a learning rate of 1):

- $w1 = w1 - \frac{\partial cost}{\partial w_1} = 1.0001588425712256$
- $w2 = w2 - \frac{\partial cost}{\partial w_2} = 1.0004765277136765$
- $b = b - \frac{\partial cost}{\partial b} = 2.000317685142451$

3.3 Problem 3: Artificial neural network

For Problem three we defined a neural network with only one hidden layer, consisting of 11 neurons. It was trained for 10 thousand epochs with the following parameters:

- $batchsize = 8000$
- $learningrate = 1$

The training curve gives the following results: Unfortunately the testing did not work from scratch and we were not able to fix the Issue: ('CUDA error: an illegal memory access was encountered',)

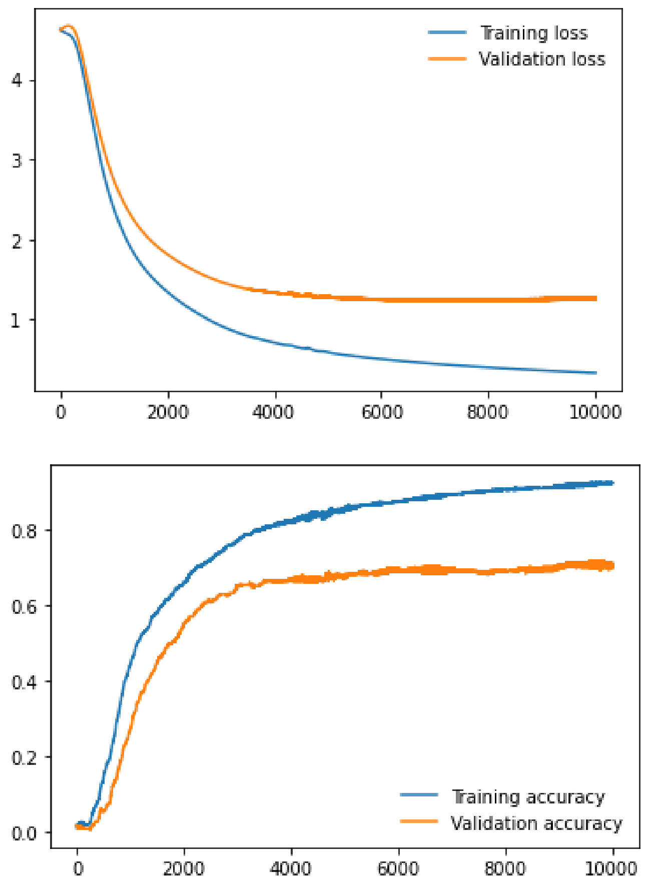


Figure 3.1: Training curves for problem 3

3.4 Problem 4: Gradient Descent

Using the given python code and applying the gradient and weight calculation described in Section 2.4 results in the line shown in Figure 3.3.

The original line is has a m of 3.30 and a c of 5.3 and the learned values are 3.28 for m and 5.27 for c.

The loss is also plotted in figure 3.4.

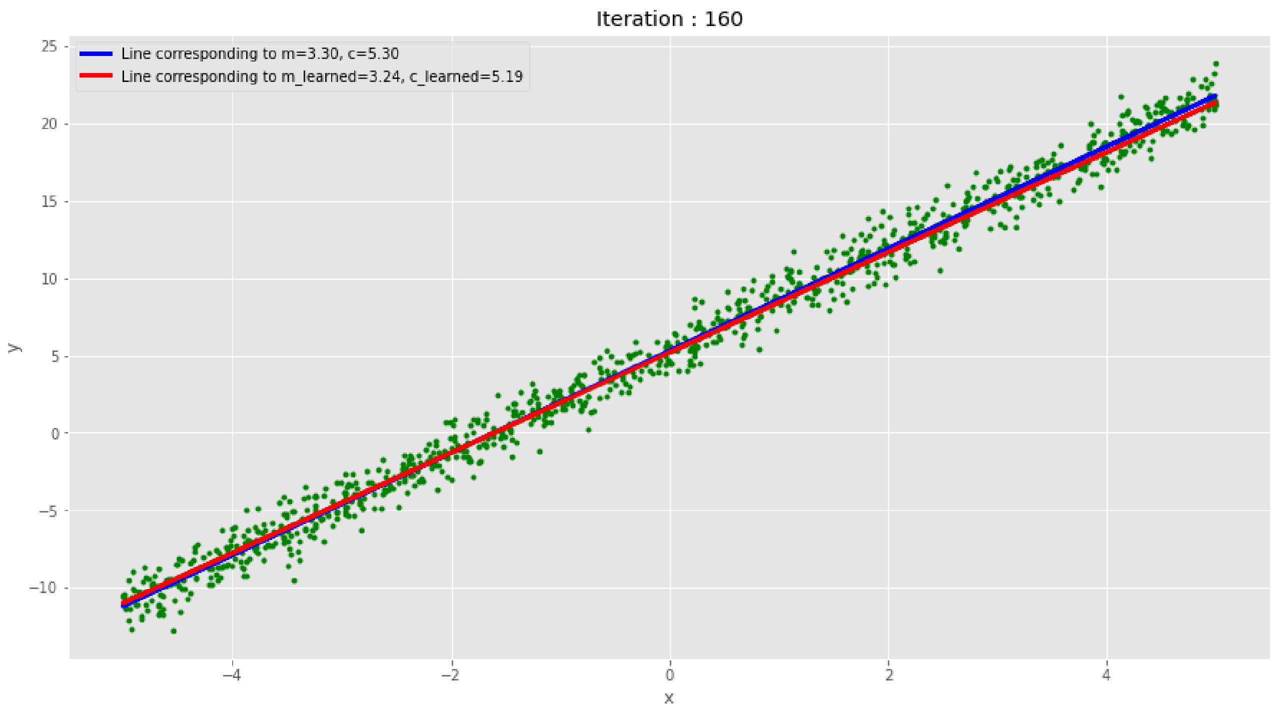


Figure 3.2: Result of gradient decent after 160 iterations

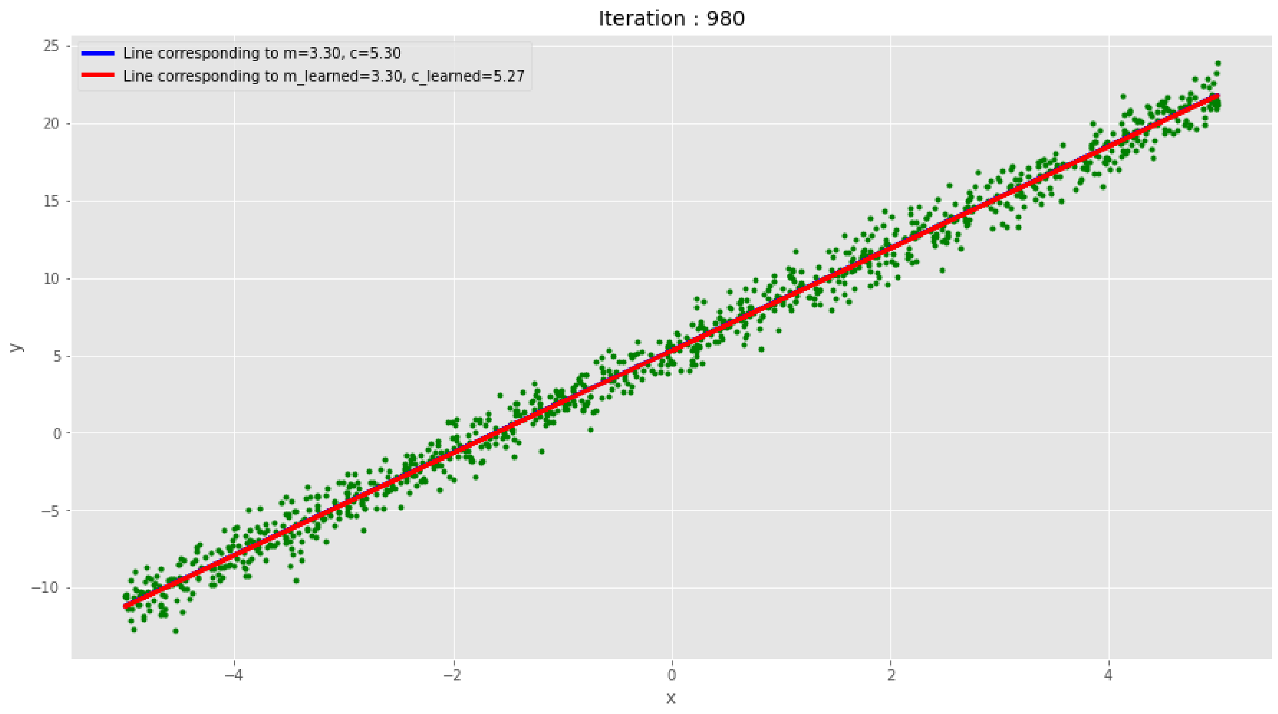


Figure 3.3: Result of gradient decent

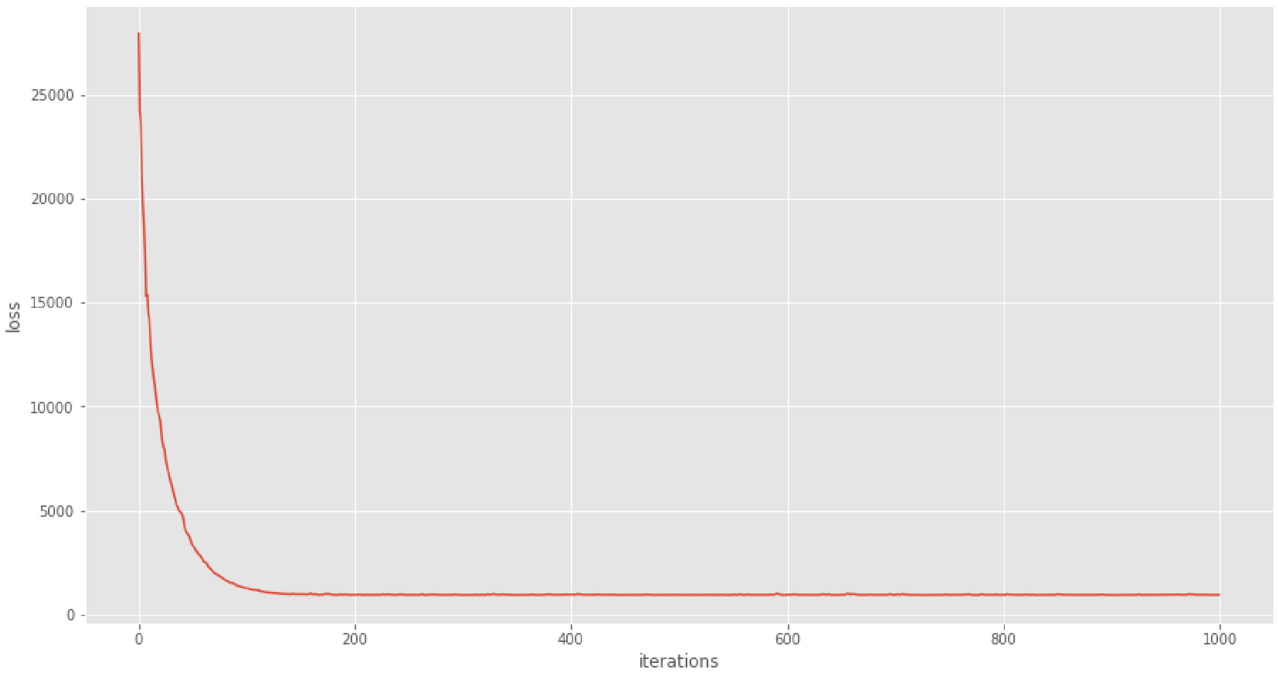


Figure 3.4: Loss of the gradient decent over the iterations

4 Discussion

4.1 Problem 1: Simple network

The concatenation of the results of the define propagation function is very similar to the Pytorch setup of a neural network with the biggest difference being the extra efficiency provided by the Pytorch implementation of the tensor operations with possible access to a GPU if available on the system.

4.2 Problem 2: Backpropagation

When comparing the two error functions given for the problem it becomes evident that the minimum of the nonlinear equation b) is more suitable to fit the model to the data than the linear function a).

4.3 Problem 3: Artificial neural network

As can be seen in the number of neurons of the network it appears that the input features are strongly correlated since they can only be accurately classified by a layer with a number of neurons which is only a fraction of the number of input features. Pruning of the input features could result in better results for the model.

4.4 Problem 4: Gradient Descent

As is can seen in 3.3 the result is very close to the actuall given line and fits very well. Over all nearly 1000 iterations are calculated but after the first 160 iterations the result is already near the true values. This can also be seen in the loss curve. All further iterations still minimize the error but only with small effects.

These results are indicators for a very well fitting model to calculate the gradient and weights.

5 Sourcecode

5.1 Problem 1

```python
from matplotlib import pyplot as plt
import numpy as np

# input data
x1 = 0.7
x2 = 0.5
data = np.array([x1,x2])

# define activation function
def sigmoid(x):
    return 1/(1+np.exp(-x))

#define matrix operation from one layer to the next
# (defining the sigmoid function as the only activation function)
def step(input, weights):
    return sigmoid(np.matmul(input, weights))

# Input Layer
hypothetical_weights = np.array([[1,0],[0,1]])
step(data,hypothetical_weights)

assert np.array_equal(step(data,hypothetical_weights), sigmoid(data))
output1 = step(data,hypothetical_weights)

#Weights Input Layer -> First Hidden Layer
w11 = 0.9 # initialize w11
w12 = 0.3 # initialize w12
w13 = 0.9 # nintialize w13
w21 = 0.1 # initialize w11
w22 = 0.2 # initialize w12
w23 = 0.4 # nintialize w13

w12 = np.array([[w11,w12,w13],[w21,w22,w23]])
output2 = step(output1,w12)
output2

# First Hidden Layer -> Second Hidden Layer
w11 =  0.1# initialize w11
w12 =  0.8# initialize w12
```

```
w13 =  0.4# nintialize w13

w21 =  0.5# initialize w21
w22 =  0.1# initialize w22
w23 =  0.6# nintialize w23

w31 =  0.6# initialize w31
w32 =  0.7# initialize w32
w33 =  0.3# nintialize w33

w23 = np.array([[w11,w12,w13],[w21,w22,w23],[w31,w32,w33]])
output3 = step(output2,w23)

# Second Hidden Layer -> Single Neuron
w11 = 0.5 # initialize w11
w12 = 0.7 # initialize w12
w13 = 0.3 # nintialize w13

w34 = np.array([w11,w12,w13])
output4 = step(output3, w34)
print(f"The output of the neural network is {str(output4)}")
```

Listing 5.1: Problem 1

5.2 Problem 2

```
%matplotlib inline

from matplotlib import pyplot as plt
import numpy as np

# define activation function and its derivative
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_p(x):
    return sigmoid(x)*(1-sigmoid(x))

T = np.linspace(-5,5,100)
plt.plot(T,sigmoid(T),c='r')
plt.plot(T,sigmoid_p(T),c='b')

## Assignment a)
data = [0.5, 1.5]
w1 = 1.0
w2 = 1.0
learning_rate = 1#?
b = 2.0
# calculate initial output
y = data[0]*w1 + data[1]*w2 + b
```

```
z = sigmoid(y)
z_ = 1.0
#cost function
cost = (z_-z)
# derivatives of the cost function w.r.t. weights
dcost_prediction = -1
dpred_dy = sigmoid_p(y)
dy_dw1 = data[0]
dy_dw2 = data[1]
dy_db = 1

# applying chain rule
dcost_dw1 = dcost_prediction * dpred_dy * dy_dw1
dcost_dw2 = dcost_prediction * dpred_dy * dy_dw2
dcost_db = dcost_prediction * dpred_dy * dy_db


# updating weights
w1 = w1 - learning_rate * dcost_dw1
w2 = w2 - learning_rate * dcost_dw2
b = b - learning_rate * dcost_db

#check output with new weights
y = data[0]*w1 + data[1]*w2 + b
z = sigmoid(y)
print(f"The error of the simple network a) with updated weights is:\n {
    str(z_-z)}")
print(f"\nThe values of the updated weights are:\n w1={str(w1)} w2={str
    (w2)} b={str(b)}")

##Assignment b)
data = [0.5, 1.5]
w1 = 1.0
w2 = 1.0
learning_rate = 1#?
b = 2.0
# calculate initial output
y = data[0]*w1 + data[1]*w2 + b
z = sigmoid(y)
z_ = 1.0
#cost function
cost = np.square((z_-z))/2
# derivatives of the cost function w.r.t. weights
dcost_prediction = (z_-z)
dpred_dy = sigmoid_p(y)
dy_dw1 = data[0]
dy_dw2 = data[1]
dy_db = 1


# applying chain rule
dcost_dw1 = dcost_prediction * dpred_dy * dy_dw1
```

```
dcost_dw2 = dcost_prediction * dpred_dy * dy_dw2
dcost_db = dcost_prediction * dpred_dy * dy_db

# updating weights
w1 = w1 - learning_rate * dcost_dw1
w2 = w2 - learning_rate * dcost_dw2
b = b - learning_rate * dcost_db

#check output with new weights
y = data[0]*w1 + data[1]*w2 + b
z = sigmoid(y)
print(f"The error of the simple network b) with updated weights is:\n {
    np.square((z_-z))/2}")
print(f"\nThe values of the updated weights are:\n w1={str(w1)} w2={str
    (w2)} b={str(b)}")
```

Listing 5.2: Problem 2

5.3 Problem 4

```
import torch
import matplotlib.pyplot as plt
%matplotlib inline

plt.style.use('ggplot')

torch.manual_seed(0)

plt.rcParams["figure.figsize"] = (15, 8)


# Generating y = mx + c + random noise
num_data = 1000

# True values of m and c
m_line = 3.3
c_line = 5.3

# input (Generate random data between [-5,5])
x = 10 * torch.rand(num_data) - 5

# Output (Generate data assuming y = mx + c + noise)
y_label = m_line * x + c_line + torch.randn_like(x)
y = m_line * x + c_line

# Plot the generated data points
plt.plot(x, y_label, '.', color='g', label="Data points")
plt.plot(x, y, color='b', label='y = mx + c', linewidth=3)
plt.ylabel('y')
plt.xlabel('x')
```

```python
plt.legend()
plt.show()

def gradient_wrt_m_and_c(inputs, labels, m, c, k):

    '''
    All arguments are defined in the training section of this notebook.
    This function will be called from the training section.
    So before completing this function go through the whole notebook.

    inputs (torch.tensor): input (X)
    labels (torch.tensor): label (Y)
    m (float): slope of the line
    c (float): vertical intercept of line
    k (torch.tensor, dtype=int): random index of data points
    '''

    # gradient w.r.t to m is g_m
    # gradient w.r.t to c is g_c

    X = (torch.take(inputs,k).numpy())
    Y = (torch.take(labels,k).numpy())

    N = k.size(0)
    sum_errors = 0

    # wrt m
    for i in range(0, N):
        sum_errors = sum_errors + X[i] * (Y[i] - m * X[i] - c)
    g_m = -2 * sum_errors

    sum_errors = 0

    # wrt c
    for i in range(0, N):
        sum_errors = sum_errors + Y[i] - m * X[i] - c
    g_c = -2 * sum_errors

    return g_m, g_c

def update_m_and_c(m, c, g_m, g_c, lr):
    '''
    All arguments are defined in the training section of this notebook.
    This function will be called from the training section.
    So before completing this function go through the whole notebook.

    g_m = gradient w.r.t to m
    g_c = gradient w.r.t to c
    '''

    updated_m = m - lr*g_m
    updated_c = c - lr*g_c

    return updated_m , updated_c

# Stochastic Gradient Descent with Minibatch

# input
X = x

# output/label
Y = y_label

num_iter = 1000
batch_size = 10

# display updated values after every 10 iterations
display_count = 20
#

lr = 0.001
m = 2
c = 1
print()
loss = []

for i in range(0, num_iter):

    # Randomly select a training data point
    k = torch.randint(0, len(Y)-1, (batch_size,))

    # Calculate gradient of m and c using a mini-batch
    g_m, g_c = gradient_wrt_m_and_c(X, Y, m, c, k)

    # update m and c parameters
    m, c = update_m_and_c(m, c, g_m, g_c, lr)

    # Calculate Error
    e = Y - m * X - c
    # Compute Loss Function
    current_loss = torch.sum(torch.mul(e,e))
    loss.append(current_loss)


    if i % display_count==0:
        #print('Iteration: {}, Loss: {}, updated m: {:.3f}, updated c:
   {:.3f}'.format(i, loss[i], m, c))
        y_pred = m * X + c
        # Plot the line corresponding to the learned m and c
        plt.plot(x, y_label, '.', color='g')
```

```python
        plt.plot(x, y, color='b', label='Line corresponding to m={0:.2f}, c={1:.2f}'.
                 format(m_line, c_line), linewidth=3)
        plt.plot(X, y_pred, color='r', label='Line corresponding to m_learned={0:.2f}, c_learned={1:.2f}'.
                 format(m, c), linewidth=3)
        plt.title("Iteration : {}".format(i))
        plt.legend()

        plt.ylabel('y')
        plt.xlabel('x')
        plt.show()

#print('Loss of after last batch: {}'.format(loss[-1]))
print('Leaned "m" value: {}'.format( m))
print('Leaned "c" value: {}'.format( c))

# Plot loss vs m
plt.figure
plt.plot(range(len(loss)),loss)
plt.ylabel('loss')
plt.xlabel('iterations')
plt.show()

# Calculate the predicted y values using the learned m and c
y_pred = m * X + c

# Plot the line corresponding to the learned m and c
plt.plot(x, y_label, '.', color='g', label='X and Y')
plt.plot(x, y, color='b', label='Line corresponding to m={0:.2f}, c={1:.2f}'.format(m_line, c_line), linewidth=3)
plt.plot(X, y_pred, color='r', label='Line corresponding to m_learned={0:.2f}, c_learned={1:.2f}'.format(m, c), linewidth=3)
plt.legend()

plt.ylabel('y')
plt.xlabel('x')
plt.show()
```

Listing 5.3: Problem 4