Deep Learning 2020 **(Due: 11-11-2020)**

Submission Assignment #1

Instructor: Jakub Tomczak *Name:* Jens van Holland, *Netid:* jhd660

This report answers questions 1 - 4 first. Question 5 will be answered during the main analysis (starting from the problem statement).
At last, in the appendix are some code snippets but for a more detailed explanation about the Python code I refer to the attached Python Script.

1 Question answers

Question 1 *Work out the local derivatives of both, in scalar terms. Show the derivation. Assume that the target class is given as an integer value.*

To begin with $\frac{\partial Loss}{\partial y_i}$.
Given that,

$$Loss = \sum_i L_i \text{ with } L_i = \begin{cases} -log(y_i) & ,i = c \\ 0 & ,i \neq c \end{cases}$$

This yields,

$$\frac{\partial Loss}{\partial y_i} = \frac{\partial}{\partial y_i}\left(\sum_i L_i\right) = \begin{cases} \frac{-1}{y_i} & ,i = c \\ 0 & ,i \neq c \end{cases}$$

where c : = true class of an instance

Next, the answer of $\frac{\partial y_i}{\partial o_j}$ is shown.
Given that,

$$y_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

When filled in this yields,

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial}{\partial o_j}\left(\frac{e^{o_i}}{\sum_j e^{o_j}}\right)$$

The quotient rule, $\frac{g(x)\cdot f^{'}(x) - f(x)\cdot g^{'}(x)}{(g(x))^2}$, is used for the next derivatives.
For situation i = j, it is known that,

$$f(x) = f'(x) = e^{o_i}$$
$$g(x) = \sum_j e^{o_j}$$
$$g'(x) = e^{o_i}$$

This yields,

$$\begin{aligned}\frac{\partial y_i}{\partial o_j} &= \frac{\left(\sum_j e^{o_j}\right) \cdot e^{o_i} - e^{o_i} \cdot e^{o_i}}{\left(\sum_j e^{o_j}\right)^2} \\ &= \frac{e^{o_i} \cdot \left(\sum_j e^{o_j} - e^{o_i}\right)}{\left(\sum_j e^{o_j}\right)^2} \\ &= \left(\frac{e^{o_i}}{\sum_j e^{o_j}}\right) \cdot \left(\frac{\sum_j e^{o_j}}{\sum_j e^{o_j}} - \frac{e^{o_i}}{\sum_j e^{o_j}}\right) \\ &= y_i(1 - y_i)\end{aligned}$$

For situation, i $\neq$ j,

$$\begin{aligned}f(x) &= e^{o_i} \\ f'(x) &= 0 \\ g(x) &= \sum_j e^{o_j} \\ g'(x) &= e^{o_j}\end{aligned}$$

This yields,

$$\begin{aligned}\frac{\partial y_i}{\partial o_j} &= \frac{\sum_j e^{o_j} \cdot 0 - e^{o_i} \cdot e^{o_j}}{\left(\sum_j e^{o_j}\right)^2} \\ &= -1 \cdot \frac{e^{o_i}}{\sum_j e^{o_j}} \cdot \frac{e^{o_j}}{\sum_j e^{o_j}} \\ &= -y_i \cdot y_j\end{aligned}$$

Thus,

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_i(1 - y_i) & , i = j \\ -y_i \cdot y_j & , i \neq j \end{cases}$$

Bonus *Work out the derivative* $\frac{\partial Loss}{\partial o_i}$.
Using the chain rule, one knows that,

$$\frac{\partial Loss}{\partial o_i} = \frac{\partial Loss}{\partial y_i} \cdot \frac{\partial y_i}{\partial o_i} \tag{1.1}$$

If i $\neq$ c, 1.1 will be zero since $\frac{\partial Loss}{\partial y_{i \neq c}} = 0$. For the next situations it is known that i = c.
For situation (c =) i = j ,

$$\begin{aligned}\frac{\partial Loss}{\partial y_i} \cdot \frac{\partial y_i}{\partial o_i} &= \frac{-1}{y_i} \cdot y_i(1 - y_i) \\ &= y_i - 1\end{aligned}$$

For situation (c =) i $\neq$ j,

$$\begin{aligned}\frac{\partial Loss}{\partial y_i} \cdot \frac{\partial y_i}{\partial o_i} &= \frac{-1}{y_i} \cdot -y_i \cdot y_j \\ &= y_j\end{aligned}$$

Thus,

$$\frac{\partial Loss}{\partial o_i} = \begin{cases} y_i - 1 & , c = i = j \\ y_j & , c = i \neq j \\ 0 & , c \neq i \end{cases}$$

If one knows $\frac{\partial Loss}{\partial y_i}$ and $\frac{\partial y_i}{\partial o_j}$ it is not necessary to compute $\frac{\partial Loss}{\partial o_i}$. It is shown in 1.1 that these are the same (using the chain rule), it would be a waist of computing time to do both during the training of a neural network. However, it is much easier to compute, and program, $\frac{\partial Loss}{\partial o_i}$ directly instead of the two partial derivatives.

Question 2 *Implement the network in the image below, including the weights. Perform one forward pass, up to the loss on the target value, and one backward pass. Show the relevant code in your report. Report the derivatives on all weights W, b , V and c. Do not use anything more than plain python and the math package.*

The derivative values of the loss w.r.t. parameters after one forward-and backpropagation is shown in table 1.

(Loss w.r.t.) Parameter	Values
W	[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
b	[0.0, 0.0, 0.0]
V	[[-0.4404, 0.4404], [-0.4404, 0.4404], [-0.4404, 0.4404]]
c	[-0.5, 0.5]

Table 1: The values for the derivatives of the loss w.r.t. parameters.

The implementation of the "scalar" neural network is (partly) shown in appendix A.

Question 3 *Load the synthetic data. Implement a training loop for your network and show that the loss drops as training progresses.*
For this network, the weights are initialised with values from the standard normal distribution and the learning rate is set to 0.01[1].
As shown in figure 1, the accuracy, and loss, of the training and validation set seem to be close. Indicating that the model generalizes well.

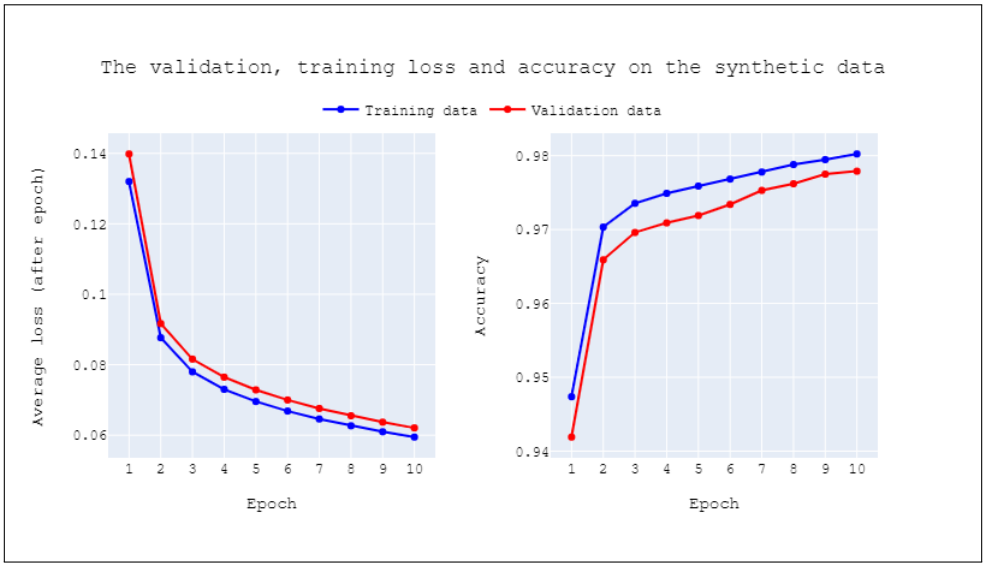

Figure 1: The loss (left) and accuracy (right) of the validation and training set on the synthetic data. Note that each point in the graphs is computed after an epoch is done.

The code in appendix A is used in a for loop, the implementation of the for loop takes a lot of space, it can be viewed in the attached Python script.

[1]The learningrate is determined by trial and error. The Glorot initialization is also tried with this network. Furthermore, a decay rate for the learningrate is also tried.

Question 4 *Implement a neural network for the MNIST data.*
Use the following architecture: 784 (input) → Linear(784, 300) → Sigmoid → Linear(300, 10) → Softmax.

The network is implemented using the Python classes: Network, Layer, Activation, Optimizer and Loss. These classes are written with Numpy. The code snippet below shows how the model is configured.

```
mod = Network()
mod.add(Layer(784,300))
mod.add(Activation("sigmoid"))
mod.add(Layer(300,10))
mod.add(Activation("softmax"))
mod.add(Loss("crossentropy"))
mod.add(Optimizer("sgd"))
```

The implementation of the classes are shown in appendix B.

Question 5 *Train the network on MNIST and plot the loss of each batch or instance against the timestep. This is called a learning curve or a loss curve. You can achieve this easily enough with a library like matplotlib in a jupyter notebook, or you can install a specialized tool like tensorboard. We'll leave that up to you.*

1. Compare the training loss per epoch to the validation loss per epoch. What does the difference tell you?

2. Run the SGD method multiple times (at least 3) and plot an average and a standard deviation of the objective value in each iteration. What does this tell you?

3. Run the SGD with different learning rates (e.g., 0.0001, 0.01, 0.05). Analyze how the learning rate value influences the final performance.

4. Based on these experiments, choose a final set of hyperparameters, load the full training data with the canonical test set, train your model with the chosen hyperparameters and report the accuracy you get.

As one can see the sub questions are indexed 1-4. I will refer to the first sub question as 1, and so on.

2 Problem statement

For this assignment, one has to build a neural network (a Multilayer Perceptron) and train it on the MNIST data set. The data set contains handwritten digits (graphs), these digits are values ranging from 0 to 9. Thus, there are 10 different classes. The data set contains 60,000 samples, the test set contains 10,000 samples. Each graph (written digit) is represented as a vector of length 784. The values of such vector are ranging from 0 to 255, indicating the intensity of a pixel.

The objective is to maximize the accuracy, and minimize the loss, meaning that the model will try to correctly predict the targets as much as possible.
Furthermore, the loss is also used to analyse the model's learning behaviour.

Next, several models are trained and evaluated on the MNIST-fashion dataset. This dataset contains 10 different classes (clothing pieces). There are 60,000 training samples and 10,000 test samples. Each graph (clothing piece) is represented as a vector of length 784. The pixel values are ranging between 0 and 255 (black and white). The goal of this second analysis is to see how more complex models perform on the dataset.

3 Methodology

Firstly, one has to determine the learningrate for the model; when does it converge? Secondly, one has to analyse the model on the training and validation data to see if the model is generic (or not). Next, a small analysis of the impact of random weights is done; how much influence do the initial weights have? Finally, one chooses the (earlier determined) hyperparameters and re-train the final model on the full dataset.
This is done for the MNIST and MNIST-fashion dataset. The training set of each dataset is split into a training set (55,000 observations) and a validation set (5,000 observations). The full training set is used in the final run.

4 Experiments

In this section there are two analyses, namely, the one on the MNIST dataset and one the MNIST-fashion dataset. The goal of the first analysis is to determine which learningrate is appropiate for the model, interpret the learning process of the model according to the accuracy and loss curves. At last, performing three runs of this model to determine if the weights are having an impact on the training process.
The goal of the second analysis is to see if certain models work better (e.g. Glorot initialization, extra layers, different activation functions) and, at last, to show the learning process of the final run of the chosen model with the accuracy and loss curves.

First analysis: MNIST

Firstly, one has to standardize [2] the training data. This is done by calculating the mean and standard deviation. Let μ_{train} and σ_{train} denote the mean and standard deviation based on the pixel values in the training set. Next, one has to subtract the mean from each pixel and divide it by the standard deviation. Since one only uses information extracted from the training set, each pixel in the test set is also normalized with μ_{train} and σ_{train}.

(Question 3)
Next, the learning rate is determined. Batches of size 2048 are used to speed up and stabilize the training process of the neural network. The batch accuracy for each learningrate is shown in figure 2, it is interesting to see that the relative high (0.1) and relative low (0.00001) learningrates are both performing worse than the learningrates between those values. It is also interesting to see that the other learningrates have the same kind of learning process (rising quickly and stabilizing). The model with learningrate 0.01 is chosen to be further analysed.

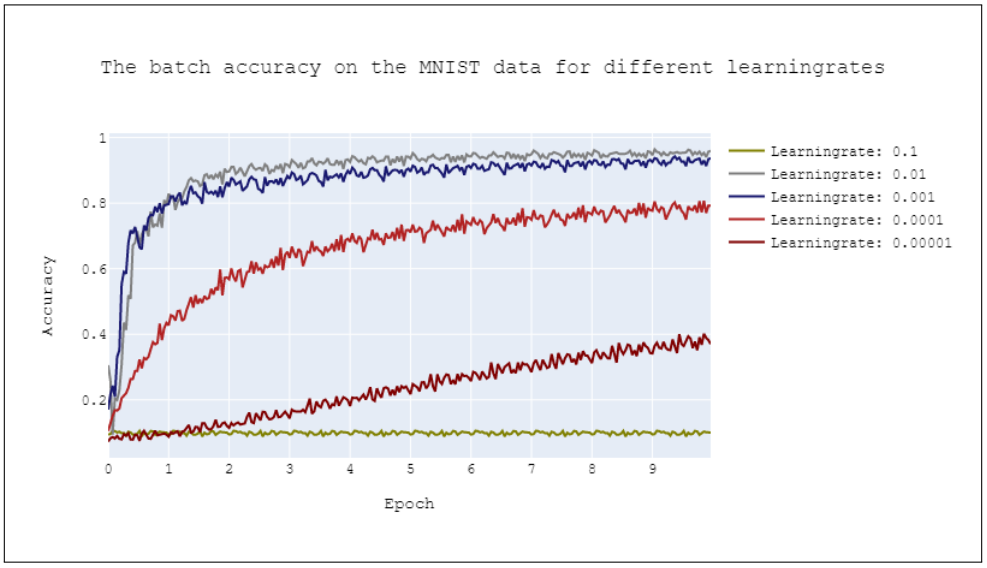

Figure 2: The batch accuracies on the MNIST dataset using several learningrates. Weights are initialized using the standard normal distribution.

[2] Normalisation is also tried, after a few runs with both methods I opted for the standardization.

(Question 1)
To analyse this model somewhat deeper, the loss and accuracy during the learning process are shown in figure 3. From the left plot, with the loss of the validation and training data, it is interesting to see that the validation loss is lower than the training loss till the end of the fourth epoch. It is also noticeable that that the losses are diverging further in the training process. This means that the model "remembers" the training set instead of generalizing. On the right graph it shown that for the validation, training en batch data the accuracy very quickly increases and stabilizes. One probably should use early stopping (around epoch 5) for a good result and minimal divergence of the losses.

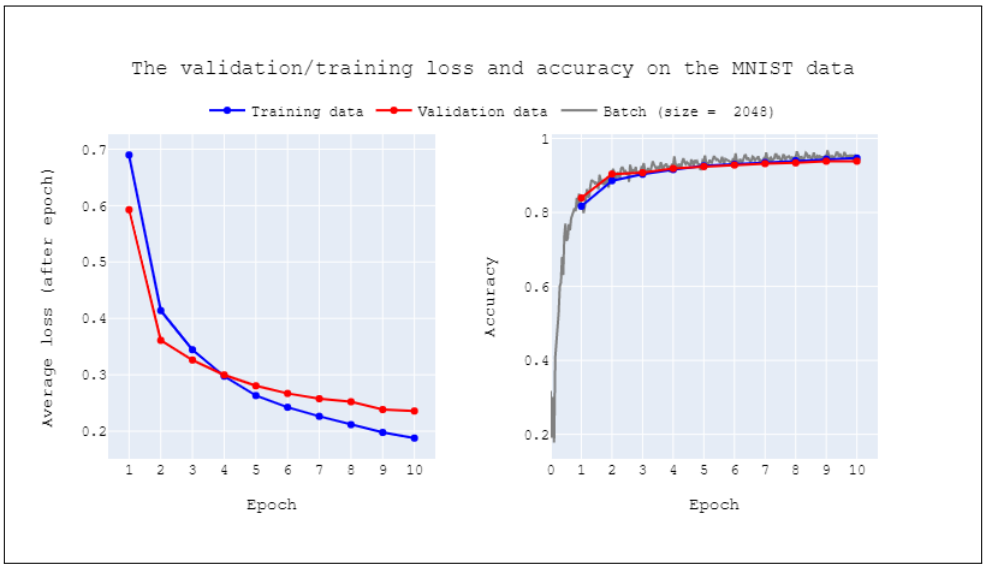

Figure 3: The loss and accuracy on the validation and training set, using standard normal initialized weights and 0.01 as learningrate.

(Question 2)
Next, it is interesting to see what the influence of the model's (random initialized) starting weights is on the performance, the accuracy, during the training process.
From figure 4, one can see in the left graph that the accuracy is not really stable in the first few epochs, meaning that the standard deviation is relative large compared to the accuracy in later epochs.
Also, from the right graph it is clear that in earlier training epochs the accuracy is depending on the initialized weights, since the deviation is sometimes very large (shaded area is big).
From the graphs, it is also clear that the final performance is (almost) the same in every run.

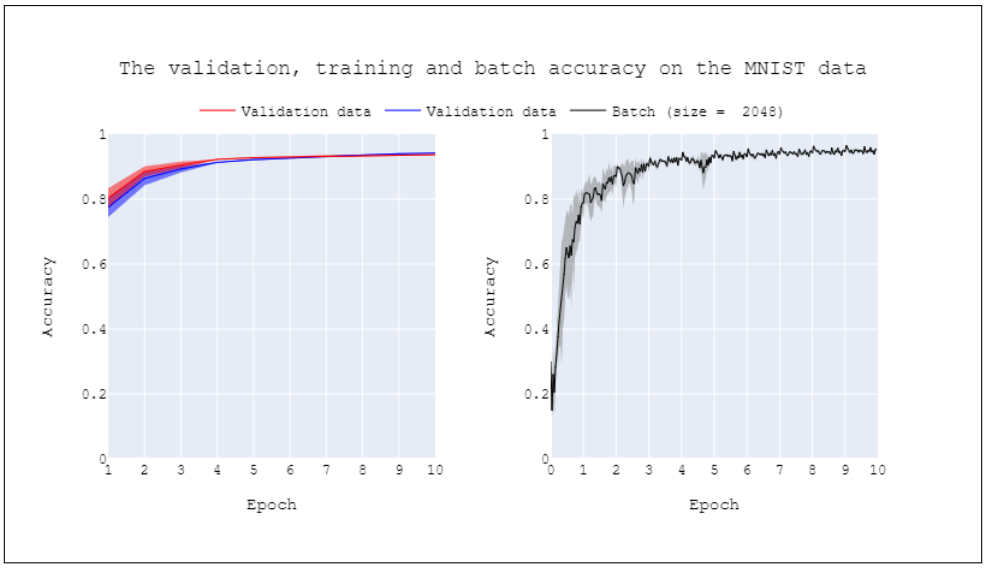

Figure 4: The accuracy of the model on the validation, training set (left) and batches (right) using a learningrate of 0.01. The shaded region is one standard deviation away from the mean accuracy, based on 3 model runs. Note that the accuracy for the validation and training data is computed after each epoch (starting at epoch 1) and the batches are computed during the epoch, hence the different x-axis range.

(Question 4)
The final model is the one with a learningrate of 0.01, this is derived from figure 2. Furthermore, the model is trained for 5 epochs, this is derived from figure 3 (the accuracy does not get very much better from that point and the loss gap between the validation and training data is small).
The training process for the final run can be seen in figure 5. As one can see, the batch accuracy is after the first epoch already at 0.8 accuracy, which is not that bad. The graph also shows that the training loss is at its lowest at the end of epoch 4. The training accuracy does not change much after epoch 3/4. One cannot change the model hyperparameters since this the final run, but otherwise this would be useful information.

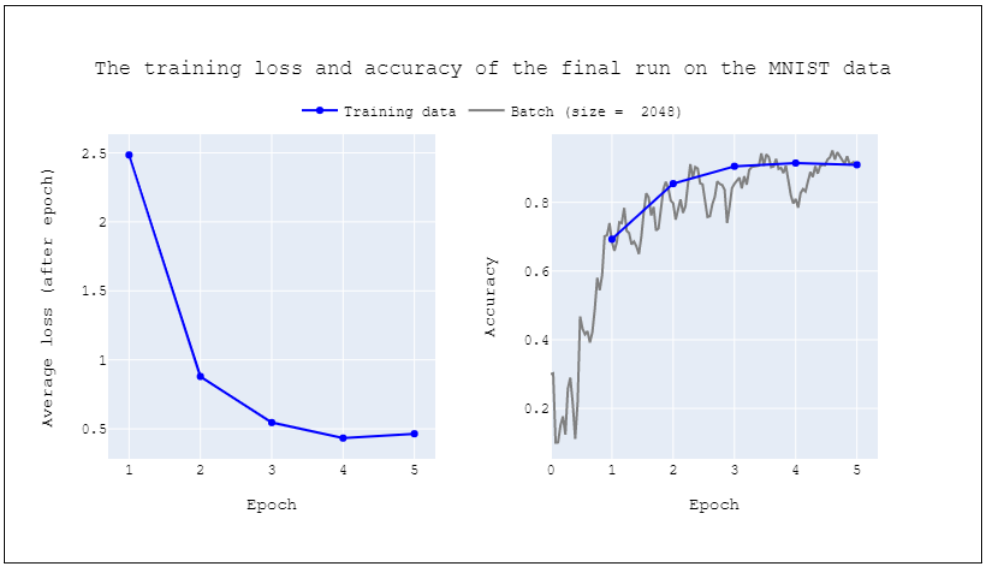

Figure 5: The training loss (left) and accuracy (right). It look likes the batch is oscillating more compared to the other graphs, this is because of the x-axis range.

As one can see in table 2, the final results of the model are not bad. The final accuracy on both datasets are almost equal, the model generalized pretty good.

Dataset	Final loss	Final accuracy
Training	0.465	0.909
Test	0.508	0.902

Table 2: The results of the last epoch (5) of the final run on the MNIST dataset.

Second analysis: MNIST-fashion

Firstly, as done in the first analysis, one standardizes the data. Let μ_{train} and σ_{train} be the mean and standard deviation of the pixels in the MNIST-fashion training set. The mean is subtracted from every pixel, in the training and test data, and is divided by the standard deviation. There are four models trained and compared in this analysis, the architectures of these models are shown in table 3.

Model	Architecture
1[3]	Linear(784,512) - Sigmoid - Linear(512,10) - Softmax
2[4]	Linear(784,512) - Sigmoid - Linear(512,256) - Tanh - Linear(256,128) - Relu -Linear(128, 10) - Softmax
3[3]	Linear(784,512) - Sigmoid - Linear(512,256) - Tanh - linear(256,10) - Softmax
4[4]	Linear(784,512) - Sigmoid - Linear(512,256) - Tanh - Linear(256,10) - Softmax

Table 3: The four different model architectures used for classifying the MNIST-fashion dataset. The learningrates for model 1, 2, 3 and 4 are respectively 0.001, 0.0001, 0.0001 and 0.0001.

The batch accuracy for the models stated in table 3 can be seen in figure 6. It is interesting to see that all models are performing quite good. Model 4 seems to be the best model during these runs, it indicates that the initialization of the weights possibly plays a roll in the performance of the model(since model 3 has the same architecture but different initialized weights). It is also noticeable that the second model during the third epoch has a dip in the training process (right graph).

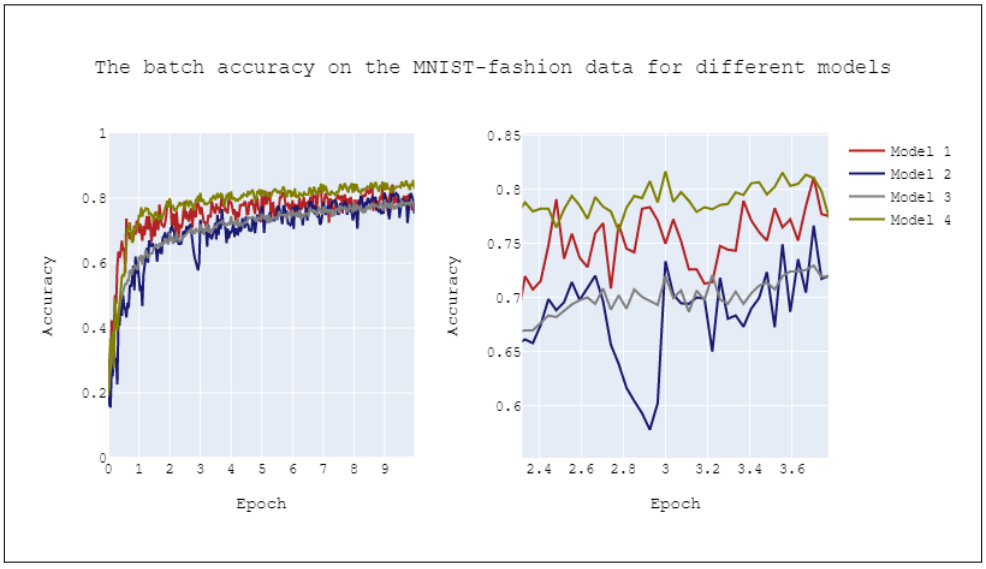

Figure 6: The batch accuracy of the models stated in table 3. The full run of all the models are displayed on the left graph and a interesting part in the learning process is displayed on the right graph.

It looks like model 4 is the best model, figure 7 shows the learning process in terms of loss and accuracy of this model. From figure 7, there is a small increase in the loss between epoch 5 and 6, it seems that the validation loss (left graph) has the same dip but the validation accuracy (right graph) is hardly influenced.

[3]Weights are initialized with the standard normal distribution
[4]Weights are initialized with the Glorot normal distribution, implementation based on McCaffrey (2019)

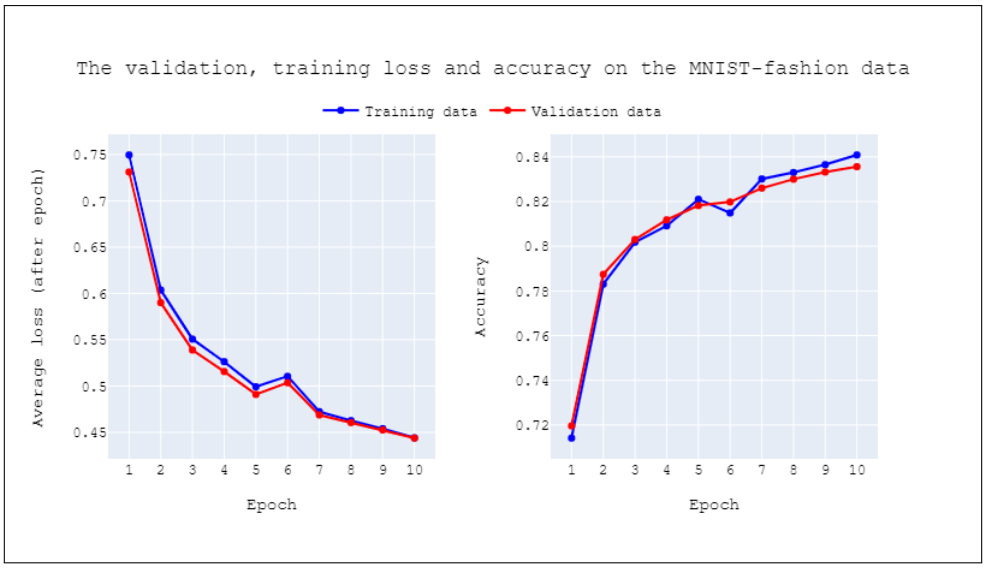

Figure 7: The learning process of the fourth model based on the loss (left) and accuracy (right).

Normally, one would show a graph (such as figure 4) to see if the initialisation of the weights influence the results. In this analyses, the mentioned graph is omitted due to the time gotten for this assignment (it takes some time to train these models).
Next, the final run of this model (on the full training data) is done with a batchsize of 2048, a learningrate of 0.001 and for 10 epochs. The learning process on the training data is shown in figure 8. It is interesting to see that, again, the loss drops relatively much between epoch 6 and 7. Also, the accuracy does not improve that much after epoch 6/7, this would probably a better cut-off point.

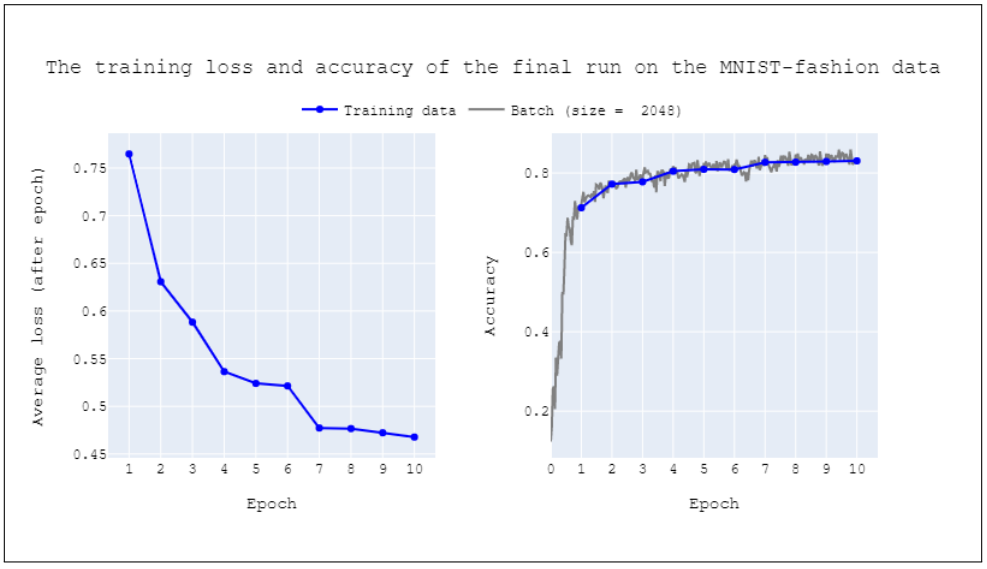

Figure 8: The learning process of the model in the final run in terms of loss and accuracy.

As can be seen in table 4, the final results of the model are not bad. The final accuracy on both datasets are (almost) equal, the model generalized pretty good.

Dataset	Final loss	Final accuracy
Training	0.524	0.809
Test	0.5472	0.799

Table 4: The results of the last epoch (10) of the final run on the MNIST-fashion dataset.

5 Results and discussion

In this report the first four questions are answered directly. These questions influenced the way of implementation (e.g. the bonus for Q1 made programming somewhat easier).
Next, there will be some small discussions and intuitions about the two main analyses.

The first analysis

Firstly, the batchsize is fixed (2048 instances per batch), it could be that the model performs better with a higher batchsize (faster convergence), but this is omitted from the analysis. My intuition tells me if a batch has more instances, the learning rate should be a bit higher. This intuition comes from the fact that the model gets more information (due to more samples in a batch) and should be able to be more confident about the direction it takes to the (local) minimal loss / optimal weights.
To be sure not to "overstep" the optimal, one could "zero in" on the optimal by multiplying the learning rate with a small factor, say a $\beta < 1$, each epoch/batch. Furthermore, to be more precise about the influence of the initial weights on the model's performance, one should run the model more than 3 times to have a better view on this.

The second analysis

Firstly, again the batchsize is fixed and a smaller or bigger batch could make some difference. Secondly, the goal of the second analysis differs from the first analysis, the goal is to see if one could get a high accuracy with a more complex model (more parameters than the models in the first analysis) instead of just answering the given questions. Also, it was interesting to use different activation functions.

The choice is made to use the best performing model according to figure 6, this is model 4. One could argue that the number of extra parameters, compared to model 1, is not justified for the accuracy that model 4 got. In practice, I would trade off the complexity of the model and its performance, but for this assignment it was more interesting to analyse the more complex model. At last, it is worth pointing out the difference between model 3 and model 4, the are the same in architecture but the initial weights are from a different distribution (same family but different parameters). One could run both models more times to see if the difference between performance is constant, meaning that model 4 out performs model 3 a significant number of times.

References

McCaffrey, J. (2019). How to do neural network glorot initialization using python. [Online; posted 09-05-2019].

A Scalar Code Appendix

The Python code below is the implementation of the backward loss and the forward and backward implementation. For more context (code), I refer to the Python script attached to this report.

```
def loss_backward(outputs, y):
    """
    Loss derivative function, according to the report. (-log(y_c), c:= true class of instance)
    """
    for i in range(2):
        if i == y: outputs[i] = -1/outputs[i]
        else: outputs[i] = 0
    return(outputs)

def forward_backward(X,y,W,V,b,c,print_params = False,
                                lr = 0.00001,
                                compute_loss = False,
                                update = False):
    """
    This function is the main function. It does the forward and back propagation.
    This function is called two times:
    1) the whole forward backward propagation
    2) the forward and loss calculation after each epoch. Hence the argument: compute_loss
    """
    ##forward pass
    o,k,h = [0]*2, [0]*3,[0]*3

    #linear forward (1)
    for j in range(3):
        for i in range(2):
            k[j] += W[i][j] * X[i]
        k[j] += b[j]

    #sigmoid forward
    for i in range(3): h[i] =  sigmoid(k[i])

    #linear forward (2)
    for i in range(2):
        for j in range(3):
            o[i] += h[j] * V[j][i]
        o[i] += c[i]

    #softmax forward (probs)
    sum_output = sum([math.exp(output) for output in o ])
    probs = [math.exp(output) / sum_output for output in o]

    #prediction for accuracy
    pred = probs.index(max(probs))

    #for computing the loss over epoch, only called when computing the loss after each epoch
    if compute_loss: return(pred,loss_forward(probs.copy(), y))

    ##backward
    dprobs, do =  [[0,0],[0,0]] , [0,0]
    #loss backward
    doutputs = loss_backward(probs.copy(), y)

    dV, dc, dh = [[0,0],[0,0], [0,0]], [0,0], [0,0,0]
    dh = [0,0,0]
    for i in range(2):
        if i == y: do[i] = probs[i] -1
        else: do[i] =  probs[i]  # "- 0"

    #gradient of V and c (2)
    for i in range(2):
        for j in range(3):
            dV[j][i] = do[i]*h[j]
            dh[j] += do[i]*V[j][i]
        dc[i] = do[i]
    dk = [None]*3

    #backward sigmoid
    for i in range(3): dk[i] = dh[i]*h[i]*(1-h[i])
```

```
    dW, db = [[0,0,0],[0,0,0]], [0,0,0]

    #gradient of W and b (1)
    for j in range(3):
        for i in range(2):
            dW[i][j] = dk[j]*X[i]
        db[j] += dk[j]

    #W, b
    for j in range(3):
        for i in range(2):
            W[i][j] = W[i][j] -lr*dW[i][j]
        b[j] += -lr*db[j]
    #V, c
    for i in range(2):
        for j in range(3):
            V[j][i] = V[j][i] -lr*dV[j][i]
        c[i] += -lr*dc[i]

    if print_params: # for Q2
        print("dW: ", dW)
        print("\ndb: ", db)
        print("\ndV: ", dV)
        print("\ndc: ", dc)
        print("\nh: ",  h)

    return(W,b,V,c)
```

B Code Classes

The code below shows how the classes Layer, Activation, Loss and Optimizer are implemented. The Network class is omitted (relative much code) but can be viewed in the attached Python script.

```
import numpy as np
class Layer():
    """
    The Layer class initializes a linear forward layer.
    It holds several objects such as the input, weights, derivatives etc.
    It also has a forward and backward function
    """
    def __init__(self, connections, neurons, initializer = "normal"):
        self.neurons = neurons

        if initializer == "normal":
            self.weights = np.random.normal(0, 1, size=(connections, neurons))
        elif initializer == "glorot_normal": #tried implementing https://visualstudiomagazine.
                                                com/articles/2019/09/05/neural-network-
                                                glorot.aspx
            term = np.sqrt(2.0/(connections+neurons))
            self.weights = np.random.normal(0, term, size=(connections, neurons))

        self.bias = np.zeros((1, neurons))
        self.name = "layer"

    def forward(self, X = None):
        self.X = X
        self.values = np.dot(X,self.weights) + self.bias

    def backward(self, grad):
        self.dweights = self.X.T @ grad
        self.dbias = np.sum(grad, axis = 0,keepdims=True)
        self.dvalues =  grad @ self.weights.T
```

```python
class Activation:
    """
    The Activation class initialises an activation (hidden and output).
    It holds several objects such as the input, weights, derivatives etc.
    It also has a forward and backward function.
    I tried to experiment with w few activation functions from: https://en.wikipedia.org/wiki/
                                                    Activation_function
    """
    def __init__(self, spec = None):
        self.spec = spec
        self.name = "activation"

    def forward(self, X):
        self.X = X
        if self.spec == "softmax":   self.values = np.exp(X) / np.sum(np.exp(X), axis=1,
                                                    keepdims=True)
        elif self.spec == "sigmoid": self.values = 1/(1+np.exp(-X))
        elif self.spec == "relu":    self.values = np.maximum(0, X)
        elif self.spec == "tanh":    self.values = (np.exp(X) - np.exp(-X)) / (np.exp(X) + np.
                                                    exp(-X))

    def backward(self,grad):
        if self.spec == "softmax":   self.dvalues = grad
        elif self.spec == "sigmoid": self.dvalues = grad * self.values * (1-self.values)
        elif self.spec == "relu":
            grad[self.X<=0] = 0
            self.dvalues =  grad
        elif self.spec == "tanh": self.dvalues = grad * (1-self.values * self.values)
```

```python
class Loss:
    """
    The Loss class initialises a loss function.
    For binary classification I also use the crossentropy but was playing around with a
                                                    seperate function.
    It holds several objects such as the input, weights, derivatives etc.
    It also has a forward and backward function
    """
    def __init__(self, spec = None, classes = None):
        self.spec = spec
        self.name = "loss"
        self.classes = classes

    def forward(self, X, y):
        if self.spec == "crossentropy":  loss = np.sum(-np.log(X[range(X.shape[0]), y])) / X.
                                                    shape[0]
        elif self.spec == "binary_entropy": loss = -1/len(y) * np.sum(np.log(X[:,y]))
        return(loss)

    def backward(self, X, y):
        # if self.spec == "binary_entropy":
        #     X[range(len(y)), y] -= 1
        #     self.dloss = X / len(y)

        if self.spec == "crossentropy":
            X[range(len(y)), y] -= 1
            self.dloss = X
```

```python
class Optimizer:
    """
    The Optimizer class initialises the optimizer (only sgd).
    It updates the weights of the Layer class after forward
    """
    def __init__(self, spec = None, learningrate = 0.01):
        self.spec = spec
        self.learningrate = learningrate
        self.name = "optimizer"

    def update_parameters(self, layer):
        if self.spec == "sgd":
            layer.weights = layer.weights - self.learningrate* layer.dweights
            layer.bias = layer.bias - self.learningrate* layer.dbias
```