Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 1

Name: Simone Colombo, *Student ID:* sco550

Please provide (concise) answers to the questions below. If you don't know an answer, please leave it blank. If necessary, please provide a (relevant) code snippet. If relevant, please remember to support your claims with data/figures.

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.

Answer The partial derivative of $\frac{\partial l}{\partial y_i}$ is calculated as shown in 0.1

$$\frac{\partial loss}{\partial y_i} = \begin{cases} \frac{\partial \sum_c l_c}{\partial y_i} & \text{if c = i} \\ 0, & \text{otherwise} \end{cases} = \begin{cases} \frac{\sum_c \partial -log(y_c)}{\partial y_i} & \text{if c = i} \\ 0, & \text{otherwise} \end{cases} = \begin{cases} \frac{-1}{y_i} & \text{if c = i} \\ 0, & \text{otherwise} \end{cases} \tag{0.1}$$

The derivative of the softmax function y_i is given by

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} \frac{\partial \frac{e^{o_i}}{\sum_j o_j}}{\partial o_j} & \text{if i } \neq \text{ j} \\ \frac{\partial \frac{e^{o_i}}{\sum_j o_j}}{\partial o_i} & \text{otherwise} \end{cases} = \begin{cases} \frac{\partial e^{o_i} (\sum_j e^{o_j})^{-1}}{\partial o^j} & \text{if i } \neq \text{ j} \\ \frac{\partial e^{o_i} (\sum_j e^{o_j})^{-1}}{\partial o^i} & \text{otherwise} \end{cases} = \begin{cases} \frac{e^{o_j}}{\sum_j e^{o_j}} \frac{e^{o_i}}{\sum_j e^{o_j}} & \text{if i } \neq \text{ j} \\ -\frac{e^{o_i}}{\sum_j e^{o_j}} \frac{e^{o_j}}{\sum_j e^{o_j}} + \frac{e^{o_i}}{\sum_j e^{o_j}} & \text{otherwise} \end{cases} \tag{0.2}$$

Hence, by applying the softmax definition to 0.2, equation 0.3 is obtained

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} -y_i y_j & \text{if i } \neq \text{ j} \\ -y_i y_j + y_i & \text{otherwise} \end{cases} = \begin{cases} -y_i y_j & \text{if i } \neq \text{ j} \\ y_i(1 - y_j) & \text{otherwise} \end{cases} \tag{0.3}$$

Question 2

Work out the derivative $\frac{\partial l}{\partial o_i}$. Why is this not strictly necessary for a neural network, if we already have the two derivatives we worked out above?

Answer The derivation of $\frac{\partial l}{\partial o_i}$ is calculated as shown in equation 0.4 (where the $\frac{\partial l}{\partial y_i}$ and $\frac{\partial l}{\partial o_j}$ are included):

$$\frac{\partial l}{\partial o_i} = \frac{\partial - log(y_c)}{\partial o_i} = \begin{cases} \frac{\partial -log(y_c)}{\partial y_c} \frac{\partial y_c}{\partial o_c} & \text{if c = i,} \\ \frac{\partial -log(y_c)}{\partial y_c} \frac{\partial y_c}{\partial o_i} & \text{if c } \neq \text{ i} \end{cases} = \ldots \tag{0.4}$$

Furthermore, if the cases $\frac{\partial y_i}{\partial o_i}$ and $\frac{\partial y_i}{\partial o_j}$ (where $i \neq j$) are considered separately, equation 0.5 follows 0.4

$$\ldots = \begin{cases} \frac{-1}{y_c}(y_c(1 - y_c)) & \text{if c = i, by applying 0.3,} \\ \frac{-1}{y_c}(-y_c y_i) & \text{if c } \neq \text{ i, by applying 0.3} \end{cases} = \begin{cases} -1 + y_c & \text{if c = i,} \\ y_i & \text{if c } \neq \text{ i} \end{cases} \tag{0.5}$$

Hence, the derivation of $\frac{\partial l}{\partial o_i}$ is independent of the class c. Furthermore, if we have the derivatives calculates in , the $\frac{\partial l}{\partial o_i}$ can be expressed with the chain rule due to the back propagation algorithm.

Question 3

Implement the network drawn 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 (including biases). Do not use anything more than plain Python and the `math` *package.*

Answer In the Code 1, the forward pass is computed. The matrix multiplications of $Z^{[i]} = XW^{[i]} + b^{[i]}$ are computed iteratively with python lists comprehensions. More specifically,

$$\begin{aligned} Z1 &= XW^{[1]} + b^{[1]} \text{ output of the hidden layer} \\ A1 &= \frac{1}{(1+e^{-z_i})} \forall \text{i} \in Z1, \text{ activated output of the hidden layer} \\ Z2 &= A1W^{[2]} + b^{[2]} \text{ output of the output layer} \\ A2 &= \frac{e^{(z_i - max(Z2))}}{\sum_{j \in |Z2|} e^{(z_j - max(Z2))}} \forall \text{i} \in Z2 \text{ activated output of the output layer} \end{aligned} \tag{0.6}$$

Source Code 1: Forward Pass

```
import math

def forward(X,W1,b1,W2,b2):
    #Output of FCN1 before activation function
    Z1 = [X[row]*W1[row][col] + X[row+1]*W1[row + 1][col] + b1[col] for row in \
    range(len(W1) - 1) for col in range(len(W1[row]))]

    #FCN1 activated output
    A1 =[1/(1 + math.exp(-z)) for z in Z1]

    #Output of FCN2 before activation function
    Z2 = [A1[row]*W2[row][col]+ A1[row+1]*W2[row + 1][col] + \
    A1[row+2]*W2[row + 2][col] + b2[col] for row in range(len(W2) - 2) \
    for col in range(len(W2[row]))]

    #FCN2 activated output
    A2 = [math.exp(z)/sum([math.exp(z) for z in Z2]) for z in Z2]

    return Z1,A1,Z2,A2
```

In the Code 2 the gradients with respect to the loss are computed. The notation is as follows

$$\begin{aligned} dZ2 &= \boldsymbol{Z2}^{\nabla} \\ db2 &= \boldsymbol{b2}^{\nabla} \\ dW2 &= \boldsymbol{W2}^{\nabla} \\ dZ1 &= \boldsymbol{Z1}^{\nabla} \\ db1 &= \boldsymbol{b2}^{\nabla} \\ dW1 &= \boldsymbol{W1}^{\nabla} \end{aligned} \tag{0.7}$$

Source Code 2: Backward Pass

```
def backward(X, W2, A1, A2, Y):
    #Gradient of Z2
    dZ2 = [-1 + A2[c]  if c == 1 else A2[c] for c in Y]

    #Gradient of W2
    dW2 = [[A1[row]*dZ2[col] for col in range(len(dZ2))] for row in range(len(A1))]

    #Gradient of b2
    db2 = dZ2

    #Gradient of A1
```

```
    dA1 = [a1*(1-a1) for a1 in A1]

    #Gradient of Z1
    dZ1 = [[sum([dZ2[row]*col[row] for row in range(len(dZ2))]) for col in W2][i]*dA1[i] \
    for i in range(len(dA1))]

    #Gradient of W1
    dW1 = [[X[row]*dZ1[col] for col in range(len(dZ1))] for row in range(len(X))]

    #Gradient of b1
    db1 = dZ1

    return dZ1, dW1, db1, dZ2, dW2, db2
```

The Code 3 defines the cross-entropy cross function.

Source Code 3: Loss function

```
#Loss function
def criterion(A2):
    loss = sum([-math.log(a) for a in A2])
    return loss
```

In code 4 the parameters of the NN are initialized.

Source Code 4: Parameters Initialization

```

#Parameters initialization

#FCN1
W1 = [[1., 1., 1.], [-1., -1., -1.]]
b1 = [0., 0., 0.]

#FCN2
W2 = [[1., 1.], [-1., -1.], [-1.,-1.]]
b2 = [0., 0.]

#Input
X = [1., -1.]

#Output
Y = [1, 0]
```

In Code 5 the main loop of the NN is run. Thus, the forward pass is run, the cost is calculated and the backward pass is run last.

Source Code 5: NN Main Loop

```
#Forward Pass
Z1,A1,Z2,A2 = forward(X,W1,b1,W2,b2)

#Cost
cost = criterion(A2)

#Backward Pass
dZ1, dW1, db1, dZ2, dW2, db2 = backward(X, W2, A1, A2, Y)
```

In the Code 6, the derivatives computed in the backward pass are shown.

Source Code 6: Print statement for derivative

```
dZ1 = [0.0, 0.0, 0.0]
dW1 = [[0.0, 0.0, 0.0], [-0.0, -0.0, -0.0]]
db1 = [0.0, 0.0, 0.0]
dZ2 = [-0.5, 0.5]
dW2 = [[-0.44039853898894116, 0.44039853898894116],
[-0.44039853898894116, 0.44039853898894116],
[-0.44039853898894116, 0.44039853898894116]]
db2 = [-0.5, 0.5]
```

Question 4

Implement a training loop for your network and show that the training loss drops as training progresses.

Answer The code 7 shows the training loop for the synthetic data and the SGD algorithm. As it is shown in the loop, the average loss over 150 iterations of the SGD algorithm is calculated to have more readable results (see conditional at line 18). In order to show that the loss decreases with the training 2 plots are generated.

Source Code 7: SGD training loop

```
1  epochs = 5
2  _iter = 150
3  #for epoch in range(epochs):#comment this line for avg. loss over epochs
4      for i in range(len(xtrain)):
5                  #Forward pass
6                  Z1,A1,Z2,A2 = forward(xtrain[i],params)
7
8                  #Calculating the loss
9                  cost = criterion(A2, ytrain[i])
10
11                 #Backward pass
12                 grads = backward(xtrain[i], params['W2'], A1, A2, ytrain[i])
13
14                 #Updating the parameters
15                 params = update_params(params, grads, learning_rate=0.01)
16                 loss.append(cost)
17                 #Averaging the loss over iter of SGD
18                 if i % _iter == 0: # Conditional for averaging the loss
19                     avg_cost = (sum(loss)/len(loss))
20                     loss = []
21                     losses.append(avg_cost)
22                     print(f'Loss epoch {i} {losses[int(i/_iter)]}')
```

Plot 1 show the avg. loss over 150 iter of SGD against the number of 150 iterations of SGD over the whole synthetic data.Plot 2 show the avg. loss over 150 iter of SGD against the number of 150 iterations of SGD over the whole synthetic data for the number of epochs (in this case 5 epochs for 60000 instances = 60000/150 * 5 = 2000).

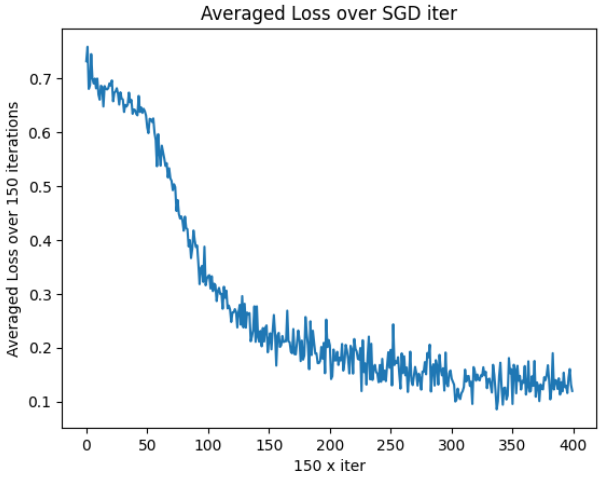

Figure 1: Loss over batches SGD

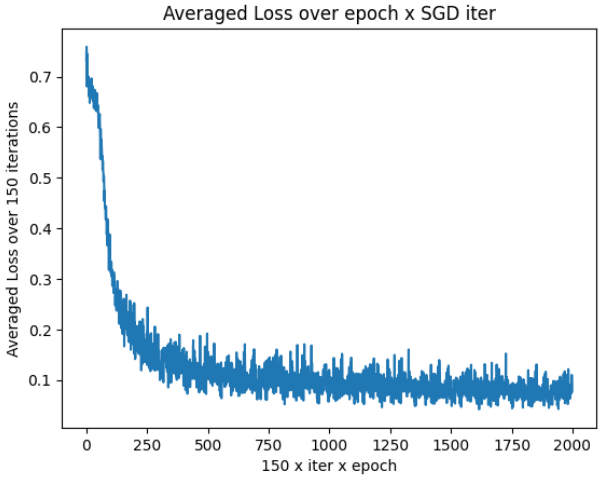

Figure 2: Loss over epochs SGD

Also note that the data is normalized as shown in Code 8.

Source Code 8: Functions to normalize the data

```
def flatten(list_to_flatten):
    flattened_list = [item for sublist in list_to_flatten for item in sublist]
    return flattened_list

#Min-max normalization
def normalize(list_values):
    list_values_f = flatten(list_values)
    min_value = min(list_values_f)
    max_value = max(list_values_f)
    for value in list_values:
        for i in range(len(value)):
            value[i] = ((value[i] - min_value)/(max_value - min_value))
            value[i] = value[i]*(1 -(-1)) + -1
    return list_values
```

Question 5

Implement a neural network for the MNIST data. Use two linear layers as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10.

Answer The main loop for the training of the NN with mini-batch gradient descent is shown in Code 9. In order to do mini-batch gradient descent the gradient of each batched as averaged and then the validation loop runs (see lines 27 for the average of the gradients and line 35 for the validation loop). In more details, the outer loop iterates over the epochs (see line 7), while, the inner loops 11 and 16 iterate over the whole data by batches. Also note that the data is normalized by diving each value by the max pixels value (e.g. 255).

Source Code 9: Main Loop NN with MB GD

```
epochs = 5
batch_size = 50

#Parameters initialization
params = initialize_param(xtrain.shape[1], 300, num_cls)

#Iterate over the epochs
for epoch in range(epochs):
    costs_train = []
```

```
    costs_val = []
    #Traning with batches
    for i in range(0,len(xtrain), batch_size):
            dW1_batches, db1_batches, dW2_batches, db2_batches,
            costs_train, costs_val = [],[],[],[],[],[]

            #Train on the batch of images iteratively
            for batch in range(batch_size):
                    Z1,A1,Z2,A2 = forward(np.array([xtrain[i + batch]/255]), params)
                    costs_train.append(criterion(A2, ytrain[i+batch]))
                    grads = backward(np.array([xtrain[i + batch]/255]), params['W2'], \
                    A1, A2, ytrain[i + batch])
                    dW1_batches.append(grads['dW1'])
                    db1_batches.append(grads['db1'])
                    dW2_batches.append(grads['dW2'])
                    db2_batches.append(grads['db2'])

            #Averaging the gradients over the batch
            grads = {"dW1": np.mean(dW1_batches, axis = 0),
                    "db1": np.mean(db1_batches,axis = 0),
                    "dW2": np.mean(dW2_batches, axis = 0),
                    "db2": np.mean(db2_batches,axis = 0)}
            #Updating the parameters
            params = update_params(params, grads, learning_rate = lr)

            #Validate the NN with the updated parameters
            for i in range(0, len(xval), batch_size):
                    Z1,A1,Z2,A2 = forward(np.array([xval[i + batch]/255]), params)
                    costs_val.append(criterion(A2, yval[i+batch]))

    val_losses.append(np.mean(costs_val))
    losses.append(np.mean(costs_train))
    print(f'Epoch {epoch + 1} : Train -> {np.mean(costs_train)} +- {np.std(costs_train)}, \
    Valid -> {np.mean(costs_val)}')
```

The relevant functions for the code 9 are shown in Code 10, 11, 12, 13 and 14. Note that for the parameters initialization the Xavier initialization (see lines 2 and 6 in Code 10) is used such that the variance of the activations are the same across every layer [1].

Source Code 10: Initialize parameters

```
def initialize_param(input_shape, hidden_layer_shape, output_layer_shape):
    #Xavier initialization for W1
    W1 = np.random.rand(hidden_layer_shape,input_shape)\
    *np.sqrt(1/(input_shape+hidden_layer_shape))
    b1 = np.zeros((hidden_layer_shape, 1))
    #Xavier initialization for W2
    W2 = np.random.rand(output_layer_shape,hidden_layer_shape)\
    *np.sqrt(1/(input_shape+hidden_layer_shape))
    b2 = np.zeros((output_layer_shape, 1))

    parameters = {"W1": W1_temp,
                  "b1": b1_temp,
                  "W2": W2_temp,
                  "b2": b2_temp}

    return parameters
```

[1] https://cs230.stanford.edu/section/4/

In Code 11, in order to avoid higher precision floating points (so, avoiding to divide by zero in the softmax function) the *np.float128* precision is attributed to Z2.

Source Code 11: Forward Pass

```
def forward(X,parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    #Output of FCN1
    Z1 = W1@X.T +b1

    #Activated output of FCN1
    A1 = sigmoid(Z1)

    #Output of FCN2
    Z2 = W2@A1 + b2

    #Activated output of FCN2
    A2 = softmax(np.array(Z2,dtype=np.float128))
    return Z1,A1,Z2,A2
```

In Code 12, the cross-entropy loss is computed for target label Y.

Source Code 12: Cost function

```
def criterion(A2,Y):
    return -np.log(A2[Y][0])
```

In Code 13 the derivatives are computed..

Source Code 13: Backward Pass

```
def backward(X, W2, A1, A2, Y):
    dZ2 = np.array([A2[i] if i != Y else -1 + A2[i] for i in range(0,10)])
    dW2 = dZ2@A1.T
    db2 = dZ2
    dA1 = A1*(1-A1)
    dZ1 = W2.T@dZ*dA1
    dW1 = dZ1@X
    db1 = dZ1
```

In Code 14 the parameters are updated.

Source Code 14: Parameters updated

```
def update_params(parameters, grads, learning_rate):

    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    W1 = np.subtract(W1,dW1*learning_rate)
```

```
    b1 = np.subtract(b1,db1*learning_rate)
    W2 = np.subtract(W2,dW2*learning_rate)
    b2 = np.subtract(b2,db2*learning_rate)

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

Question 6

Work out the vectorized version of a batched forward and backward. That is, work out how you can feed your network a batch of images in a single 3-tensor, and still perform each multiplication by a weight matrix, the addition of a bias vector, computation of the loss, etc. in a single numpy call.

Answer In the Code 15 the data is fed to the network in batches. Hence, the input data shape is $[batches \times instances] = [55000 \times 784]$ (if the whole data set is used at once, otherwise $[batchsize \times 784]$). Therefore, the forward/backward pass are performed on the whole data set at once.

Source Code 15: Main Loop

```
(xtrain, ytrain), (xval, yval), num_cls = load_mnist()
epochs = 5
for epoch in range(epochs):
    for batch in range(0,len(xtrain), batch_size):
        grads = {"dW1": 0,"db1":  0,"dW2":  0,"db2":  0}
        Z1,A1,Z2,A2 = forward(xtrain[batch:batch+batch_size]/255,params)
        cost = criterion(A2, ytrain[batch:batch+batch_size])
        grads = backward(xtrain[batch:batch+batch_size]/255, params['W2'],\
        A1, A2,  ytrain[batch:batch+batch_size])
        params = update_params(params, grads, learning_rate=0.03)
        loss_train.append(cost)

        #Validate
        for batch in range(0,len(xval), batch_size):
            Z1,A1,Z2,A2 = forward(xval[batch:batch+batch_size]/255, params)
            cost = criterion(A2, yval[batch:batch+batch_size])
            val.append(cost)

    loss_train.append(cost)
    loss_val.append(cost_val)
    print(f'Epoch {epoch + 1} : Train -> {cost}`, Valid -> {cost_val}')
```

The parameters are initialized with the Xavier initialization as shown in Code 10, while, the forward pass is shown in Code 16 (note that here `np.matmul` is used to vectorize the multiplication over the whole batch).

Source Code 16: Forward Pass

```
def forward(X,parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    Z1 = np.matmul(X,W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(A1,W2) + b2
    A2 = softmax(np.array(Z2,dtype=np.float128))
    return Z1,A1,Z2,A2
```

In the code 17 it is shown the vectorized version of the backward pass with the use of gradient clipping to avoid exploding gradients (see line 11)

Source Code 17: Backward Pass

```
1  def backward(X, W2, A1, A2, Y):
2      dZ2 = np.array([[a2[i] if i != y else -1 + a2[i] for i in range(0,10)]\
3      for a2,y in zip(A2,Y)])
4      dW2 = np.matmul(A1.T,dZ2)
5      db2 = np.sum(dZ2, keepdims = True)
6      dA1 = A1*(1-A1)
7      dZ1 =  np.matmul(dZ2,W2.T)*dA1
8      dW1 =  np.matmul(X.T,dZ1)
9      db1 = np.sum(dZ1, keepdims = True)
10
11     dW2 = np.clip(dW2, -5, 5) #Gradient clipping
12     grads = {"dW1": dW1,
13              "db1": db1,
14              "dW2": dW2,
15              "db2": db2}
16
17     return grads
```

The rest of the other functions are the same as the ones used for the mini batch version of GD.

Question 7

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.*

Answer The NN from question 5 is now further analyzed. Plot 3 shows the decrease of the training loss with the average and standard deviation over 3 random parameter initializations. Furthermore, it is possible to notice that the standard deviation of the loss decreases as the epochs increase. This insight reflect the fact that the network is actually converging to the same minimum in the parameters space. Successively, Plot 4 shows the training of the network over 5 epochs by averaging the loss every 150 iteration of the SGD with different learning rates (0.001, 0.01, 0.003, 0.03). Hence, the best learning rate that converges the faster towards the minimum is 0.03. This results reflects the fact the the update of the weights is a bit bigger than by using the other learning rate, and that also helps to converge faster in this scenario. However, that is not sure that the network is a good predictor yet.

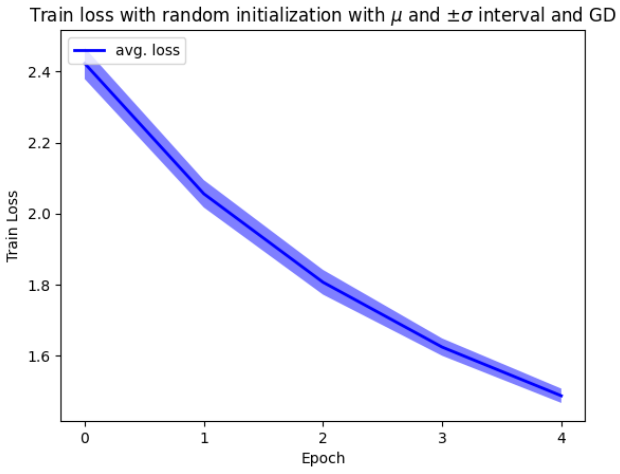

Figure 3: Loss over batches SGD

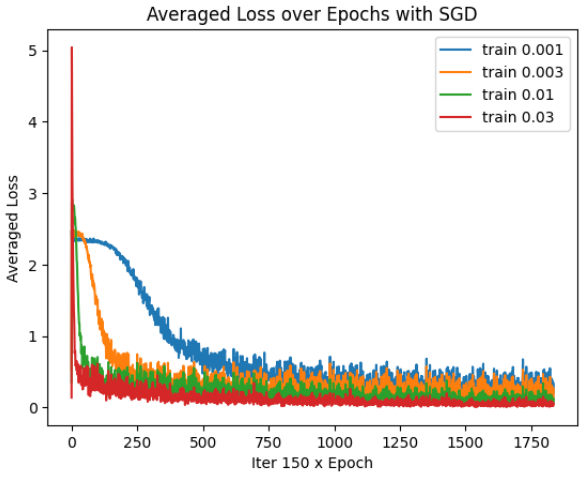

Figure 4: Loss over epochs SGD

Hence, Plots 7 and 8 show that the training loss the validation loss both decrease over epochs, while the test accuracy increases. In turn, it the network is properly learning without overfitting on the training data. For further comparisons, the training loss, validation loss and the test accuracy are reported also for the learning rate of 0.001 (see Plots 5 and 6). As it is possible to notice the learning rate really affects the rate at which the network is learning over the amount of epochs.

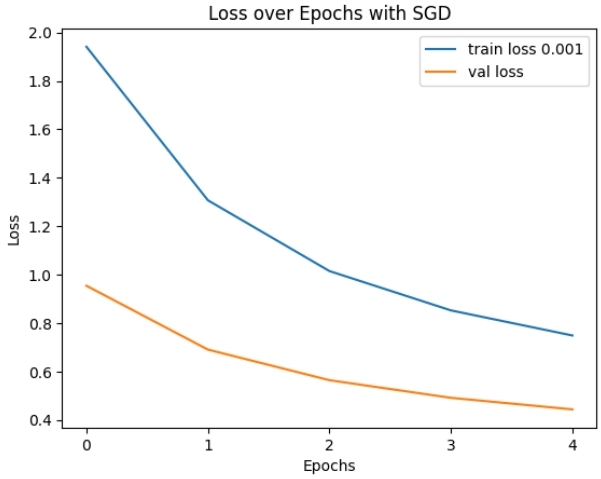

Figure 5: Train and Validation Loss over 5 epochs with 0.001 lr and SGD

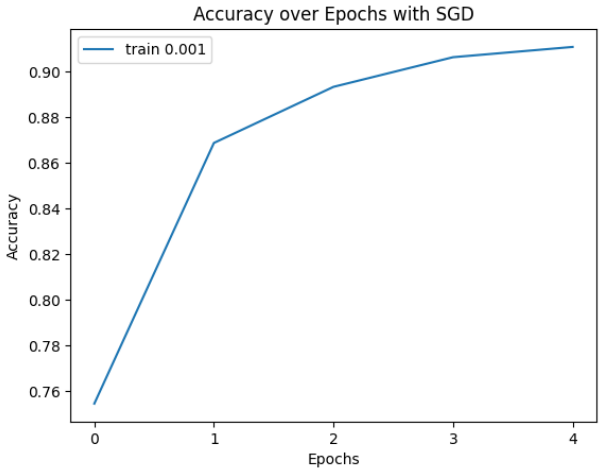

Figure 6: Accuracy over epochs with train lr 0.001 and SGD

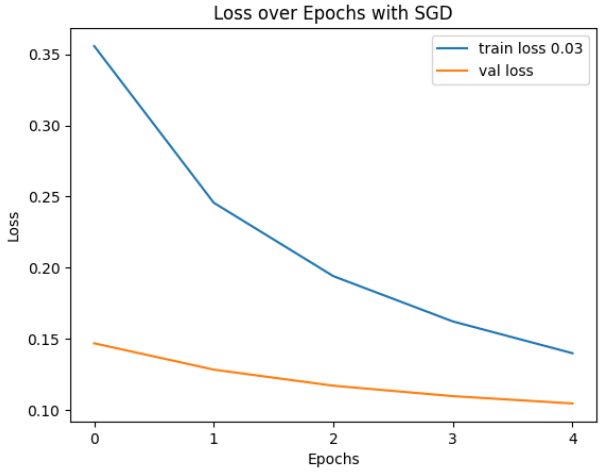

Figure 7: Train and Validation Loss over 5 epochs with 0.03 lr and SGD

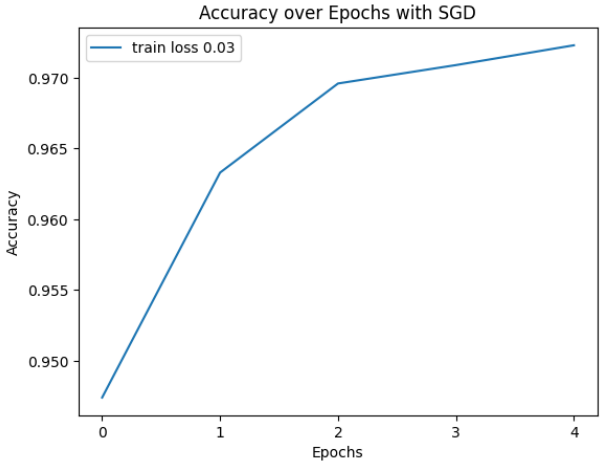

Figure 8: Accuracy over epochs with train lr 0.03 and SGD

In conclusion, the NN with SGD on the MNIST dataset reaches the best accuracy (98%) on the test set and it has the lowest bias on the train set with learning rate 0.03.