Deep Leaning - Assignment 1

Tiddo Loos - 2574974

November 16, 2021

1 Part1

1.1 Question 1

The Loss and the softmax activation function are defined as follows:

$$\begin{gathered} softmax(o)_i = \frac{exp(o_i)}{\sum_j exp(o_j))} \\ Loss = \sum_j l_j \\ l_j = \left\{ \begin{array}{ll} -\log(y_c) & c = j \\ 0 & \text{otherwise} \end{array} \right. \end{gathered} \tag{1}$$

Derivative of softmax with respect to o_i and o_j:

$$i = j, \frac{\partial y_i}{\partial o_i} = \frac{\exp(o_i)\sum_j \exp(o_j) - exp(o_i)exp(o_i)}{(\sum_j \exp(o_j))^2} = yi(1 - y_i) \tag{2}$$

$$i \neq j, \frac{\partial y_i}{\partial o_j} = \frac{0 - exp(o_i)exp(o_j)}{(\sum_j \exp(o_j))^2} = -yiy_j \tag{3}$$

Derivative of the cross entropy loss function with respect to y_i:

$$\frac{\partial l}{y_i} = \begin{cases} -\frac{1}{y_c} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{4}$$

1.2 Question 2

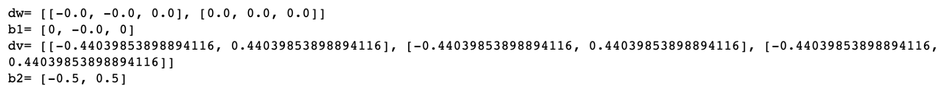
Given the derivatives of the softmax function and the cross entropy loss function it is not necessary to work out the derivative of the cross entropy loss function with respect to o_i. In Python when applying backward propagation the derivatives of the softmax and loss function are enough to apply backpropagation. To calculate the difference between a prediction and the true label simply subtract the true label from the prediction: $a_i - y_i$. When calculating the derivative of the loss with respect to o_i, we get the same result.

$$\begin{gathered} \frac{\delta l}{\delta o_i} = \frac{\delta l}{\delta y_i}\frac{y_i}{o_j} = \sum_{k=1}^{c} -\frac{t_k}{y_k}\frac{\delta y_k}{\delta o_j} = -\left[\frac{y_k}{a_i}\frac{\delta a_i}{\delta y_i} + \sum_{k=1,k\neq i}^{c}\frac{y_k}{a_k}\frac{\delta a_k}{\delta y_i}\right] = \\ -\left[\frac{y_i}{a_i}a_i(1-a_i) + \sum_{k=1,k\neq i}^{c}\frac{y_k}{a_k}(a_k a_i)\right] = -y_i + y_i a_i + \sum_{k=1,k\neq i}^{c} y_k a_i = a_i + \sum_{k=1}^{c} y_k - y_i = a_i - y_i \end{gathered} \tag{5}$$

2 Part2

2.1 Question 3

The network of the images presented in the assignment as been worked out using scalars. The code of the main class that is created for this assignment is presented in the Appendix (Figure 10. For questions 3 the goal was to show the derivatives of the weights and biases. These are presented in Figure 1, a screenshot of the output of the corresponding jupyter notebook.



```
dw= [[-0.0, -0.0, 0.0], [0.0, 0.0, 0.0]]
b1= [0, -0.0, 0]
dv= [[-0.44039853898894116, 0.44039853898894116], [-0.44039853898894116, 0.44039853898894116], [-0.44039853898894116,
0.44039853898894116]]
b2= [-0.5, 0.5]
```

Figure 1: values of the weights and biases after one forward and one backward propagation

2.2 Question 4

In this assignment a training loop was created to train the scalar network. The learning rate was set to 0.01 and the epochs to 3.0. The cross-entropy loss was investigated and is presented in Figure 2. What can be seen in Figure 2 is that the cross entropy loss decreases due to training. However, around a cross entropy loss of 0.68 training staggers and the average training loss converges.

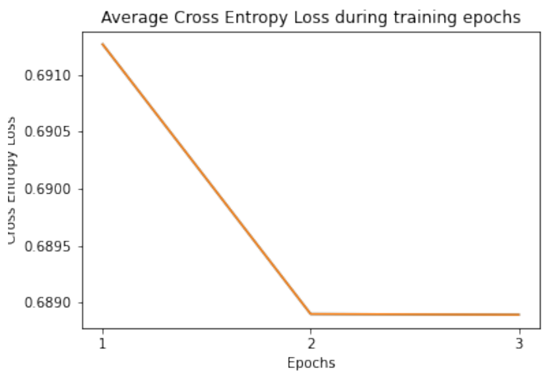


Figure 2: Cross Entropy Loss of the Scalar Neural Network over 3 epochs

3 Part3

3.1 Question 5

For this assignment a vectorized Neural Network was created. The network consisted of 784 input nodes, a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10. The vectorized neural network class can be found in Figure 11 in Appendix2. The Python package 'numpy' was used to create matrices and vectors.

3.2 Question 6

The implemented minibatch gradient descent can be reviewed in the jupyter notebook corresponding to this report. The code can be found below the header 'Q6 Minibatch Gradient Descent'. Coding minibatch gradient descent basically is implementing an extra for loop with batches of array's of training data points and their corresponding output datapoints. These batches are fed to to the neural network to complete forward and backward propagation. The minibatch gradient descent was implemented with a batch size of 500 and 100 resulting in arrays of the training data of (784, 500) and for the output data (10, 500) for batch size 500. The results during the training and validation

of the implemented batch gradient descent are presented in Figure 3 and 4 (batch sizes 500 and 100 respectively). The networks trained on the batched gradient descent yield an accuracy on the MNIST validation set of 0.86 and 0.89 for batch size 500 and 100 respectively.
From the figures it can be seen that the learning occurs in batches when examining the shapes of the plotted lines. What is remarkable is that the loss on the validation set and on the training set is lower for the mini batch gradient descent of batch size 100.

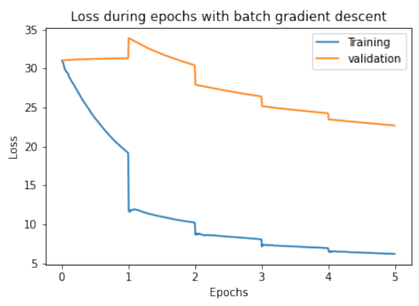


Figure 3: Loss on training and validation data with batch gradient descent, batch size = 500

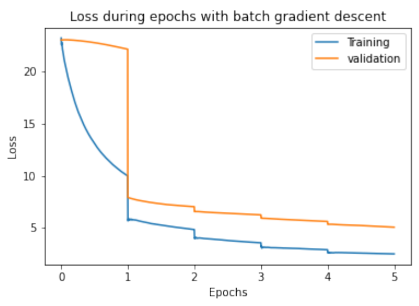


Figure 4: Loss on training and validation data with batch gradient descent, batch size = 100

3.3 Question 7.1

The cross entropy loss during training and on the validation set are presented in Figure 5. For this assignment stochastic gradient descent is used on the MNIST data set with (final=False). The learning rate was set to 0.01. Also, for all experiments with the MNIST data set in this assignment, the data was normalized by dividing all data-points with 255 (the maximum value present in the data).
The decrease of the loss over the training epochs suggests that the network is learning. Figure 5 suggest that the network is performing better on the validation set than on the training set. It is important to mention that the validation loss is always calculated after the training epoch. Therefore, it is to be expected that the average training loss for the validation set is lower than on the training set.

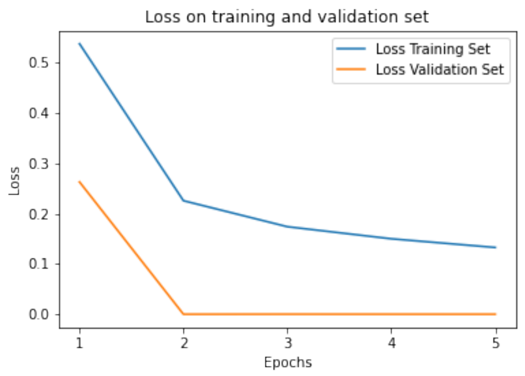


Figure 5: Loss on training and validation data

3.4 Question 7.2

Different runs are conducted to check if the outcomes are consistent. Since the weights are randomly assigned at the beginning of the training epochs it could be that the loss differs in earlier training stages. What can be seen is that the during the first epoch there is some difference between the first average value of the loss (Figure 6 and 7). When trianing is initiated the yielded loss van variate and this is shown in Figure 7 where the standard deviate during the first epoch is relatively high. However, the network trains consistently and in all 4 runs the loss converges almost to 0. The standard deviation also decreases, meaning that the average losses of the 4 runs do not differ much.

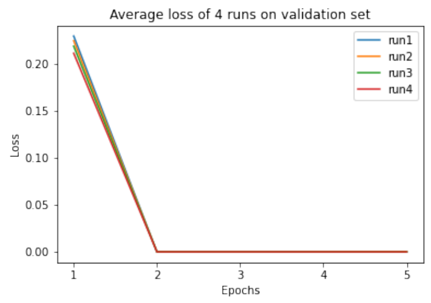


Figure 6: Average loss of 4 runs

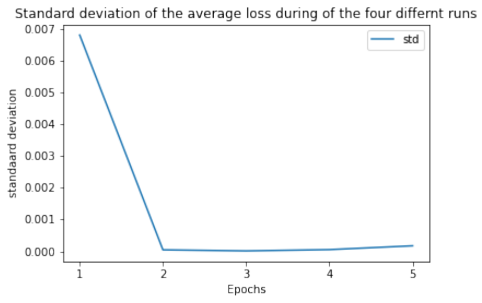


Figure 7: Std of the average loss of 4 runs

3.5 Question 7.3

For this exercise the parameter learning rate was examined. Learning rates starting from 0.002 to 0.02 were examined with a step-size of 0.002. the results are presented in Figures 8 and 9. The accuracy in Figure 8 was calculated on MNIST validation set, with the network being training during 5 epochs on the MNIST training set. The validation set consists of 5000 data points to predict. Setting the learning rate to 0.016 yields to highest accuracy on the validation set. in Figure 9 it can be seen that the highest initial average loss at epoch 1 is achieved with a learning rate of 0.016. As explained earlier, this could happen through the randomness in assigning the initial weights.

3.6 Question 7.4

For this last subquestion the training data (final = True) of the MNIST dataset was used to train and test the network. For training, a stochastic gradient descent was used as this yieled the lowest loss on the training and validation set and highest accuracy on the validation set compared to minibatch gradient descent. The learning rate was set to 0.016 as this yielded the highest accuracy on the validation set of MNIST in question 7.3. The epochs are kept to 5 for training as the assignment suggested. After training, the accuracy was calculated based on the predictions on the included test set. The trained neural network performed classification on the test set with an accuracy of 0.952 (95.2%).

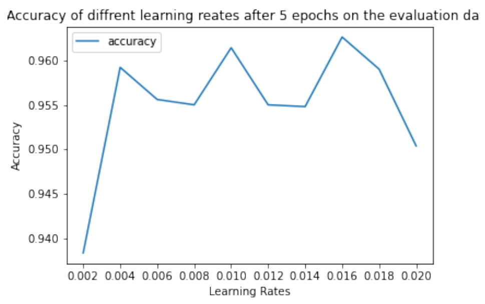


Figure 8: Accuracy versus learingrate value

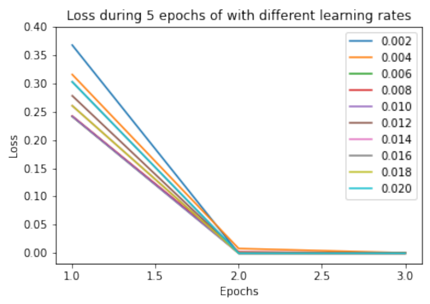
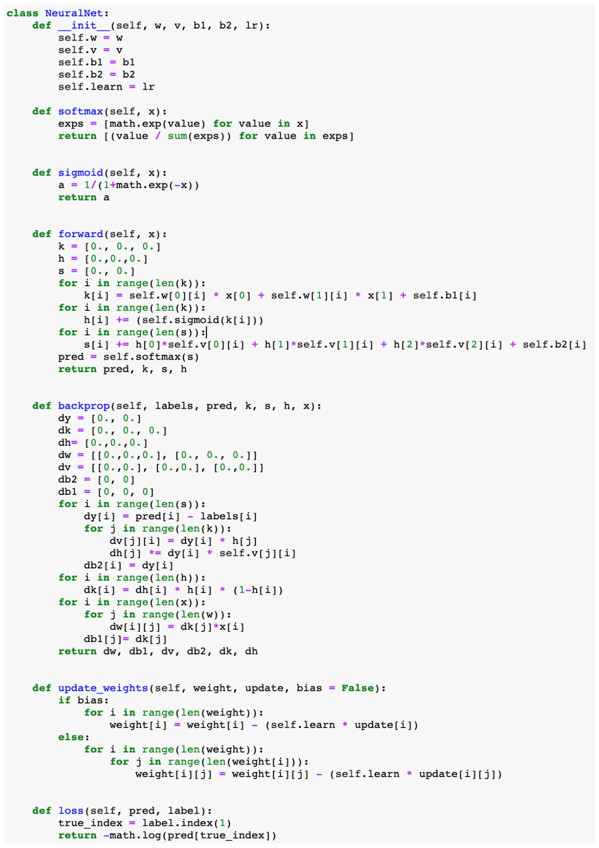


Figure 9: Loss of the different learning rates over 5 epochs

3.7 Appendix

3.7.1 Appendix1: Scalar Neural Network



```python
class NeuralNet:
    def __init__(self, w, v, b1, b2, lr):
        self.w = w
        self.v = v
        self.b1 = b1
        self.b2 = b2
        self.learn = lr

    def softmax(self, x):
        exps = [math.exp(value) for value in x]
        return [(value / sum(exps)) for value in exps]


    def sigmoid(self, x):
        a = 1/(1+math.exp(-x))
        return a


    def forward(self, x):
        k = [0., 0., 0.]
        h = [0.,0.,0.]
        s = [0., 0.]
        for i in range(len(k)):
            k[i] = self.w[0][i] * x[0] + self.w[1][i] * x[1] + self.b1[i]
        for i in range(len(k)):
            h[i] += (self.sigmoid(k[i]))
        for i in range(len(s)):
            s[i] += h[0]*self.v[0][i] + h[1]*self.v[1][i] + h[2]*self.v[2][i] + self.b2[i]
        pred = self.softmax(s)
        return pred, k, s, h


    def backprop(self, labels, pred, k, s, h, x):
        dy = [0., 0.]
        dk = [0., 0., 0.]
        dh= [0.,0.,0.]
        dw = [[0.,0.,0.], [0., 0., 0.]]
        dv = [[0.,0.], [0.,0.], [0.,0.]]
        db2 = [0, 0]
        db1 = [0, 0, 0]
        for i in range(len(s)):
            dy[i] = pred[i] - labels[i]
            for j in range(len(k)):
                dv[j][i] = dy[i] * h[j]
                dh[j] *= dy[i] * self.v[j][i]
            db2[i] = dy[i]
        for i in range(len(h)):
            dk[i] = dh[i] * h[i] * (1-h[i])
        for i in range(len(x)):
            for j in range(len(w)):
                dw[i][j] = dk[j]*x[i]
            db1[j]= dk[j]
        return dw, db1, dv, db2, dk, dh


    def update_weights(self, weight, update, bias = False):
        if bias:
            for i in range(len(weight)):
                weight[i] = weight[i] - (self.learn * update[i])
        else:
            for i in range(len(weight)):
                for j in range(len(weight[i])):
                    weight[i][j] = weight[i][j] - (self.learn * update[i][j])


    def loss(self, pred, label):
        true_index = label.index(1)
        return -math.log(pred[true_index])
```

Figure 10: Appendix1: The main class for the scalar Neural Network

3.7.2 Appendix2: Vectorized Neural Network

```
class NeuralNet:
    def __init__(self, W1, W2, b1, b2, lr):
        self.W1 = W1
        self.W2 = W2
        self.b1 = b1
        self.b2 = b2
        self.learn = lr

    def softmax(self, x):
        return np.exp(x) / np.sum(np.exp(x), axis=0)


    def sigmoid(self, X):
        X = np.clip(X, -500, 500 )
        return 1 / (1 + np.exp(-X))


    def forward(self, x, nn):
        Z1 = np.dot(self.W1, x) + self.b1
        A1 = nn.sigmoid(Z1)
        Z2 = np.dot(self.W2, A1) + self.b2
        A2 = nn.softmax(Z2)
        return Z1, A1, A2


    def backprop(self, A1, A2, x, y):
        m = x.shape[1]
        dz2 = A2 - y
        dW2 = np.dot(dz2, A1.T) * (1/m)
        db2 = np.sum(dz2, axis =1, keepdims = True) * (1/m)
        dZ1 = np.dot(self.W2.T, dz2) * (1 - np.power(A1, 2))
        dW1 = np.dot(dZ1, x.T) * (1/m)
        db1 = np.sum(dZ1, axis =1, keepdims = True) * (1/m)
        return dW1, dW2, db1, db2

    def update_weights(self, dW1, dW2, db1, db2):
        self.W1 = self.W1 - self.learn * dW1
        self.b1 = self.b1 - self.learn * db1
        self.W2 = self.W2 - self.learn * dW2
        self.b2 = self.b2 - self.learn * db2

    def loss(self, y_pred, y):
        return -np.log(y_pred[np.where(y)])
```

Figure 11: Appendix2: The main class for the vectorized Neural Network