ECE421 Assignment 2 Report

Name: Deniz Akyildiz
Student number: 1003102224

1. Neural Networks using Numpy

1.1 Helper Functions

In this part, the helper functions for the implementation and training of the neural network are presented

1.1.1 ReLU

The first function to be implemented is the ReLU function where $ReLU(x) = max(x, 0)$. The implementation in Python could be seen in Figure-1.

```
def relu(x):
    return (x * (x > 0))
```

Figure-1: ReLU Python Implementation

1.1.2 Softmax

In order to prevent overflow, as suggested in the handout, the maximum values are subtracted from input values before computing the exponentials. Here the Softmax function is given by:

$$\sigma(z)_j = \frac{exp(z_j)}{\sum_{k=1}^{K} exp(z_k)} \quad j = 1, \cdots, K \; for \; K \; classes.,$$

and the Python implementation could be seen in Figure-2.

```
def softmax(x):
    exp_x = np.exp(x - np.max(x))
    return exp_x/np.sum(exp_x, axis=1, keepdims=True)
```

Figure-2: Softmax Python Implementation

1.1.3 Compute

This function computes the output (prediction) of a layer, given the input, weights, and biases. The implementation in Python could be seen in Figure-3.

```
def compute(X, W, b):
    return (np.matmul(X,W) + b)
```

Figure-3: Compute Python Implementation

1.1.4 Average CE

This function computes the average cross-entropy loss for the given dataset. The average cross-entropy loss value is given by:

$$\boxed{\text{Average CE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log\left(p_k^{(n)}\right)}$$

Figure-4: Average cross-entropy loss formula

Where $y_k^{(n)}$ is the true one-hot label for sample *n*, $p_k^{(n)}$ is the predicted class probability (i.e. softmax output for the *k*th class) of sample *n*, and *N* is the number of examples. The implementation in Python could be seen in Figure-5.

```
def averageCE(target, prediction):
    return (-1 * np.mean( target * np.log(prediction + 1e-12) ))
```

Figure-5: Average Cross-entropy Loss Python Implementation

1.1.5 Grad CE

Here, the gradient of the cross-entropy loss with respect to the softmax's inputs is calculated. The derivation could be seen in Figure-6.

- $\frac{\partial L}{\partial o}$ derivation:

Let's first find for one element, $\frac{\partial L}{\partial o_i}$

$\rightarrow \frac{\partial L}{\partial o_i} = \frac{\partial L}{\partial p_k} \cdot \frac{\partial p_k}{\partial o_i}$, start with $\frac{\partial p_k}{\partial o_i}$

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

(for ease of notation, call $\sum_{j=1}^{K} e^{o_j} = \Sigma$)

using quotient rule:

$$\frac{\partial p_k}{\partial o_i} = \frac{\frac{\partial e^{o_k}}{\partial o_i} \Sigma - \frac{\partial (\Sigma)}{\partial o_i} \cdot e^{o_k}}{\Sigma^2}$$

if $i \neq k$ we have $\frac{\partial p_k}{\partial o_i} = \frac{-e^{o_i} \cdot e^{o_k}}{\Sigma^2} = \frac{-e^{o_i}}{\Sigma} \cdot p_k = -p_i p_k$

if $i = k$, then we have $\frac{\partial p_k}{\partial o_i} = \frac{(e^{o_i} \Sigma) - e^{o_i} e^{o_k}}{\Sigma^2} = \frac{e^{o_i}}{\Sigma} - p_i p_k$

Thus:

$$\frac{\partial p_k}{\partial o_i} = \begin{cases} -p_i \cdot p_k, & i \neq k \\ p_i(1-p_k) = p_i(1-p_i), & i = k \end{cases}$$

Using this: $\frac{\partial L}{\partial o_i} = \frac{\partial L}{\partial p_k} \cdot \frac{\partial p_k}{\partial o_i} = -\sum_{k=1}^{K} y_k \cdot \frac{\partial (\log p_k)}{\partial o_i} = -\sum_{k=1}^{K} \frac{y_k}{p_k} \frac{\partial p_k}{\partial o_i}$

$$= \underbrace{\frac{-y_i (1-p_i) p_i}{p_k}}_{k=i} - \underbrace{\sum_{k \neq i}^{K} \frac{y_k}{p_k} (-p_k p_i)}_{k \neq i} = -y_i (1-p_i) + \sum_{k \neq i}^{K} y_k p_i$$

$$= -y_i + y_i p_i + \sum_{k \neq i}^{K} y_k p_i = -y_i + \sum_{k=1}^{K} y_k p_i = -y_i + p_i \underbrace{\left(\sum_{k=1}^{K} y_k \right)}_{1}$$

$$= -y_i + p_i \Rightarrow \boxed{\frac{\partial L}{\partial o} = p - y}$$

Figure-6: Cross-entropy Gradient Derivation

The resulting expression, as seen from Figure-6 is:

$\frac{\partial L}{\partial o} = p - y$, where p is the prediction (output of softmax), and y is the target result (i.e. the actual value). The implementation in Python could be seen in Figure-7.

```python
def gradCE(target, o):
    # derivation could be found in the report
    return (softmax(o) - target)
```

Figure-7: Cross-entropy Gradient Python Implementation

1.2 Backpropagation Derivation

In this part, the derivations of the gradients that are used in the backpropagation algorithm as well as their Python implementations are presented. There might be some difference observed between the mathematical derivations and Python implementations, which is caused by the way inputs to the functions are shaped (to be able to perform matrix multiplication) in the Python code.

1.2.1 $\frac{\partial L}{\partial W_o}$ Derivation

The expression for the gradient of the loss with respect to the output layer weights is shown in this part. The derivation process could be seen in Figure-8 and Python implementation could be seen in Figure-9.

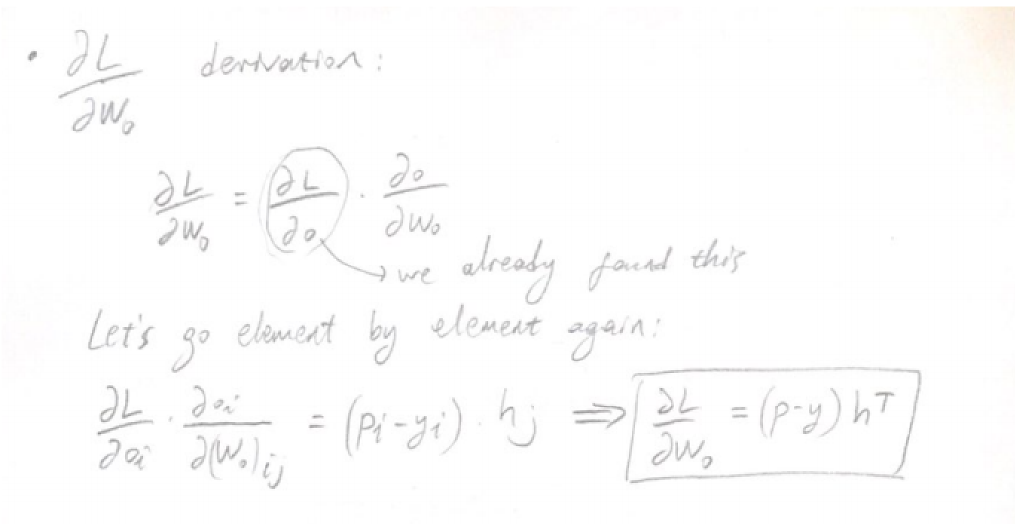


Figure-8: Gradient of the Loss With Respect to the Output Layer Weights

```python
def dL_by_dWo(target, o, h):
    # derivation could be found in the report, here the expression
    # is different than the report, in order to match the matrix size for the
    # multiplication. (i.e h: 10000xH --> hT:Hx10000)
    # Dimensions:
    # target, o : 10000x10
    # h         : 10000xH
    # output --> Hx10 (HxK)
    return np.matmul( np.transpose(h), gradCE(target, o) )
```

Figure-9: Gradient of the Loss With Respect to the Output Layer Weights Python Implementation

1.2.2 $\frac{\partial L}{\partial b_o}$ Derivation

The expression for the gradient of the loss with respect to the output layer biases is shown in this part. The derivation process could be seen in Figure-10 and Python implementation could be seen in Figure-11.

- $\frac{\partial L}{\partial b_o}$ derivation:

$$\frac{\partial L}{\partial b_o} = \frac{\partial L}{\partial o} \cdot \frac{\partial o}{\partial b_o}$$

Similar to previous derivation of $\frac{\partial L}{\partial W_o}$

$$\frac{\partial L}{\partial b_{o_i}} = \frac{\partial L}{\partial o_i} \cdot \frac{\partial o_i}{\partial b_{o_i}} = (p_i - y_i) \cdot 1 \Rightarrow \boxed{\frac{\partial L}{\partial b_o} = (p - y) \cdot 1^T}$$

Figure-10: Gradient of the Loss With Respect to the Output Layer Biases

```python
def dL_by_dbo(target, o):
    # derivation could be found in the report
    # target, o : 10000x10
    # one_matrix: 1x10000
    # output --> 1x10 (1xK)
    one_matrix = np.ones((1, target.shape[0]))
    return np.matmul(one_matrix, (gradCE(target, o)))
```

Figure-11: Gradient of the Loss With Respect to the Output Layer Biases Python Implementation

1.2.3 $\frac{\partial L}{\partial W_h}$ Derivation

The expression for the gradient of the loss with respect to the hidden layer weights is shown in this part. The derivation process could be seen in Figure-12 and Python implementation could be seen in Figure-13.

• $\frac{\partial L}{\partial W_h}$ derivation:

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial W_h} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h}\frac{\partial h}{\partial W_h}$$

For one element:

$$\frac{\partial L}{\partial (W_h)_{ji}} = \frac{\partial L}{\partial o_k}\frac{\partial o_k}{\partial h_j}\frac{\partial h_j}{\partial (W_h)_{(j,i)}} = (p_k - y_k)\cdot(W_o)_{k,j}\cdot\frac{\partial h_j}{\partial (W_h)_{(j,i)}}$$

$$= (p_k - y_k)(W_o)_{k,j}\cdot\frac{\partial(\mathrm{ReLU}(W_{h_j}x_i + b_{h_j}))}{\partial (W_h)_{(j,i)}}$$

$$= (p_k - y_k)(W_o)_{(k,j)}\left(x_i * 1_{h_j>0}\right); \quad 1_{h_j>0} = \begin{cases} 1, & h_j>0 \\ 0, & \text{else} \end{cases}$$

$$\Rightarrow \boxed{\frac{\partial L}{\partial W_h} = (p-y)\, W_o^T\cdot\left(x_i * 1_{h_j>0}\right)}$$

Figure-12: Gradient of the Loss With Respect to the Hidden Layer Weights

```python
def dRelu(x):
    return ((x > 0) * 1)

def dL_by_dWh(target, o, x, hidden_input, Wo):
    # hidden input is WhX+bh
    # Wo is the output layer weight matrix (shape Hx10)
    # target and softmax(o) shape 1x10
    # input x is 10000x784 --> xT: 784x10000
    # hidden_input : 10000xH
    # gradCE : 10000x10
    # Wo: Hx10
    # output --> 784xH
    # this is to match the dimensions, since the relu is dependent on
    # the number of nodes, not the input.
    # this is equivalent to summing over K output nodes
    return np.matmul(np.transpose(x), \
            (dRelu(hidden_input) \
            * np.matmul((gradCE(target, o)), np.transpose(Wo))))
```

Figure-13: Gradient of the Loss With Respect to the Hidden Layer Weights Python Implementation

1.2.4 $\frac{\partial L}{\partial b_h}$ Derivation

The expression for the gradient of the loss with respect to the hidden layer biases is shown in this part. The derivation process could be seen in Figure-14 and Python implementation could be seen in Figure-15.

• $\frac{\partial L}{\partial b_h}$ derivation: For one element

$$\frac{\partial L}{\partial (b_h)_j} = \frac{\partial L}{\partial o_k} \frac{\partial o_k}{\partial h_j} \frac{\partial h_j}{\partial (b_h)_j} = (p_k - y_k)(W_o)_{(k,j)} \underbrace{\frac{\partial h_j}{\partial (b_h)_j}}_{(1 * 1_{h_j > 0})}$$

$$\Rightarrow \boxed{\frac{\partial L}{\partial b_h} = (p-y)\, W_o^T . (1 * 1_{h_j > 0})}$$

Figure-14: Gradient of the Loss With Respect to the Hidden Layer Biases

```python
def dL_by_dbh(target, o, hidden_input, Wo):
    # hidden input is WhX+bh
    # Wo is the output layer weight matrix (shape Hx10)
    # target and softmax(o) shape 1x10
    # input x is 10000x784 --> xT: 784x10000
    # hidden_input : 10000xH
    # one_matrix = 1x10000
    # gradCE : 10000x10
    # Wo: Hx10
    # output -->1x10000 * 10000xH = 1xH
    one_matrix = np.ones((1, hidden_input.shape[0]))
    # this is to match the dimensions, since the relu is dependent on
    # the number of nodes, not the input.
    # this is equivalent to summing over K output nodes
    return np.matmul(one_matrix, \
            (dRelu(hidden_input) \
            * np.matmul((gradCE(target, o)), np.transpose(Wo))))
```

Figure-15: Gradient of the Loss With Respect to the Hidden Layer Biases Python Implementation

1.3 Learning

The training is performed over 200 epochs, with 1000 hidden units. Weights are initialized following the Xaiver initialization scheme, and biases are set to zero. To train the network, first a forward pass is performed.Then, the gradients are calculated as shown in the previous section and used in the backpropagation algorithm to update the weights and biases, with the Gradient Descent with momentum optimization method as seen in Figure-16.

Here, the γ is set to 0.99, and the α is set to$2x10^{-7}$, since the total loss is used in the training process instead of the average loss. Finally the v matrices, are initialized to the same size as the hidden and output layer weight matrix sizes, with a very small value, 10^{-5}in this case.

$$\boldsymbol{\nu}_{\text{new}} \leftarrow \gamma \boldsymbol{\nu}_{\text{old}} + \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{\nu}_{\text{new}}$$

Figure-16: Gradient Descent with Momentum

The resulting training and validation loss could be seen in Figure-17, while the training and validation accuracies are plotted in Figure-18. From these plots, it could be observed that validation accuracy tracks the training accuracy closely, so there is no significant overfitting observed. Final accuracy values are 0.9, 0.89, and 0.89 for the training, validation, and test datasets respectively. The training loop in Python could be seen in Figure-19.

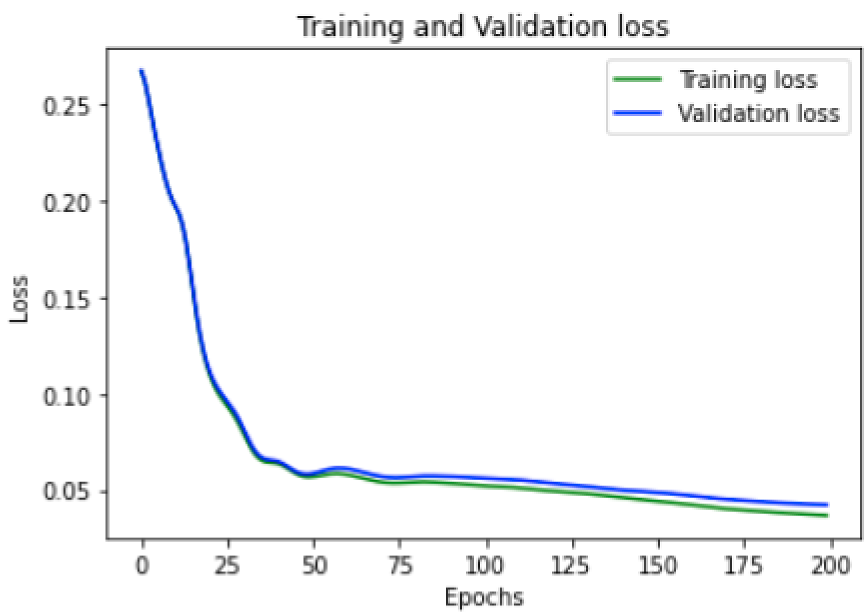
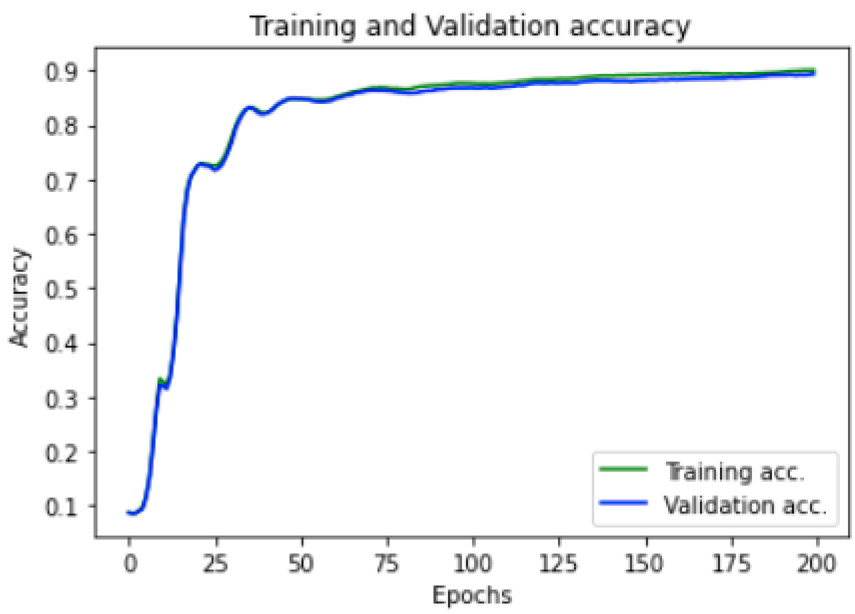


Figure-17: Training and Validation Loss




```
Training accuracy =  0.9005
Validation accuracy =  0.8938333333333334
Test accuracy =  0.893906020558003
```

Figure-18: Training and Validation Accuracy

```python
for _ in range(epochs):
  # forward pass for training
  prediction, o, h, hidden_input = forward_propagation(train_data, Wh, bh, Wo, bo, train_target)
  # calculate loss and accuracy
  loss.append(averageCE(train_target, prediction))
  prediction_idx  = prediction.argmax(axis = 1)
  target_idx = train_target.argmax(axis=1)
  check_equal = (prediction_idx==target_idx)
  accuracy.append(np.mean(check_equal))

  # forward pass for validation
  val_prediction, val_o, val_h, val_hidden_input = forward_propagation(val_data, Wh, bh, Wo, bo, val_target)
  # calculate loss and accuracy
  val_loss.append(averageCE(val_target, val_prediction))
  val_prediction_idx  = val_prediction.argmax(axis = 1)
  val_target_idx = val_target.argmax(axis=1)
  val_check_equal = (val_prediction_idx == val_target_idx)
  val_accuracy.append(np.mean(val_check_equal))

  # calculate gradients
  dL_dWo = dL_by_dWo(train_target, o, h)
  dL_dbo = dL_by_dbo(train_target, o)
  dL_dWh = dL_by_dWh(train_target, o, train_data, hidden_input, Wo)
  dL_dbh = dL_by_dbh(train_target, o, hidden_input, Wo)

  # update weights and biases
  vWh = (gamma * vWh) + (alpha * dL_dWh)
  vWo = (gamma * vWo) + (alpha * dL_dWo)
  vbh = (gamma * vbh) + (alpha * dL_dbh)
  vbo = (gamma * vbo) + (alpha * dL_dbo)
  Wh = Wh - vWh
  Wo = Wo - vWo
  bh = bh - vbh
  bo = bo - vbo

return Wh, bh, Wo, bo, accuracy, loss, val_accuracy, val_loss
```

Figure-19: Training Loop in Python