Jia Ming (Carter) Huang (Student ID: 1003893245)

	P1	P2	P3	Total
Mark				

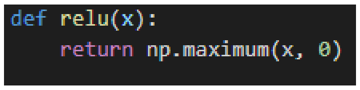
Table 1: Marking Table

Submission Date: Monday, February 28, 2022, 11:59 PM
Number of Pages: 10

Contents

1 Helper Functions [6 points]

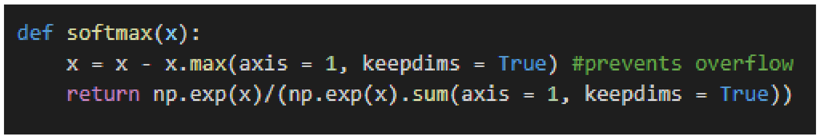
Figure 1 shows the Python implementation of the ReLU function, which returns a numpy array with ReLU activation.



```python
def relu(x):
    return np.maximum(x, 0)
```

Figure 1: ReLU Activation

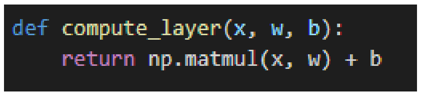
Figure 2 shows the Python implementation of the softmax function, which returns a numpy array with softmax activations.



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

Figure 2: Softmax Activation

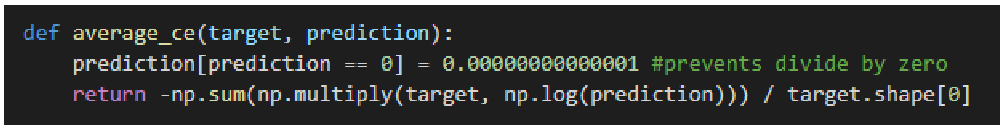
Figure 3 shows the Python implementation of the compute function, which returns the product between the weights and input, plus the biases.



```python
def compute_layer(x, w, b):
    return np.matmul(x, w) + b
```

Figure 3: Compute Layer

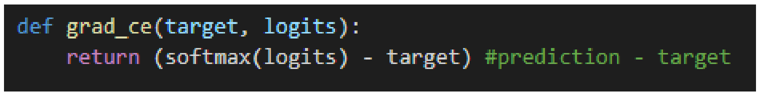
Figure 4 shows the Python implementation of the average cross-entropy loss function.



```python
def average_ce(target, prediction):
    prediction[prediction == 0] = 0.00000000000001 #prevents divide by zero
    return -np.sum(np.multiply(target, np.log(prediction))) / target.shape[0]
```

Figure 4: Average Cross Entropy Loss

Figure 5 shows the Python implementation of the gradient of the cross entropy loss function.



```python
def grad_ce(target, logits):
    return (softmax(logits) - target) #prediction - target
```

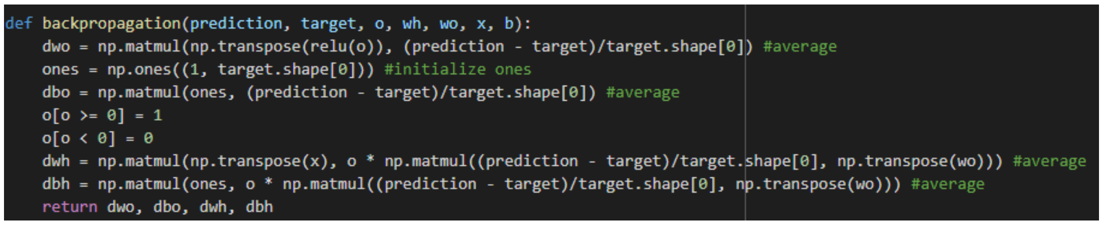
Figure 5: Gradient of Cross Entropy Loss

We know that $\mathcal{L} = -\sum_{k=1}^{K} y_k log(p_k)$, $p = \frac{e^{o_j}}{\sum_{k=1}^{K} e^{o_k}}$, and $\sum_{k=1}^{K} y_k = 1$ so the gradient of the cross entropy loss is derived below:

$$
\begin{aligned}
\frac{\partial L}{\partial o_k} &= \frac{\partial L}{\partial p_k}\frac{\partial p_k}{\partial o_k} \\
\frac{\partial L}{\partial p_k} &= -\sum_{k=1}^{K} \frac{y_k}{p_k} \\
\frac{\partial p_k}{\partial o_k} &= \begin{cases} \frac{\partial}{\partial o_k}(\frac{e^{0_k}}{\sum_{k=1}^{K} e^{o_k}}) & k = j \\ \frac{\partial}{\partial o_k}(\frac{e^{0_j}}{\sum_{k=1}^{K} e^{o_k}}) & k \neq j \end{cases} \\
&= \begin{cases} \frac{e^{o_k}\sum_{k=1}^{K} e^{o_k} - e^{o_k}e^{o_k}}{(\sum_{k=1}^{K} e^{o_k})^2} & k = j \\ \frac{0 - e^{o_j}e^{o_k}}{(\sum_{k=1}^{K} e^{o_k})^2} & k \neq j \end{cases} \\
&= \begin{cases} \frac{e^{o_k}}{\sum_{k=1}^{K} e^{o_k}}\frac{\sum_{k=1}^{K} e^{o_k} - e^{o_k}}{\sum_{k=1}^{K} e^{o_k}} & k = j \\ -p_j p_k & k \neq j \end{cases} \\
&= \begin{cases} p_k(1-p_k) & k = j \\ -p_j p_k & k \neq j \end{cases} \\
&= \begin{cases} p_j(1-p_j) & k = j \\ -p_j p_k & k \neq j \end{cases} \\
\frac{\partial L}{\partial o_k} &= -\sum_{k=1}^{K} \frac{y_k}{p_k}\frac{\partial p_k}{\partial o_k} \\
&= -\frac{y_j}{p_j}\frac{\partial p_j}{\partial o_j} - \sum_{k \neq j}^{K} \frac{y_k}{p_k}\frac{\partial p_k}{\partial o_j} \\
&= -\frac{y_j}{p_j}p_j(1-p_j) - \sum_{k \neq j}^{K} \frac{y_k}{p_k}(-p_k p_j) \\
&= -y_j + y_j p_j + p_j \sum_{k \neq j}^{K} y_k \\
&= -y_j + p_j(y_j + \sum_{k \neq j}^{K} y_k) \\
&= -y_j + p_j \sum_{k=1}^{K} y_k \\
&= p_j - y_j
\end{aligned}
$$

2 Backpropagation Derivation [8 points]

Figure 6 shows the Python implementation of the backpropagation algorithm with respect to the output layer weights, output layer biases, hidden layer weights, and hidden layer biases.



```python
def backpropagation(prediction, target, o, wh, wo, x, b):
    dwo = np.matmul(np.transpose(relu(o)), (prediction - target)/target.shape[0]) #average
    ones = np.ones((1, target.shape[0])) #initialize ones
    dbo = np.matmul(ones, (prediction - target)/target.shape[0]) #average
    o[o >= 0] = 1
    o[o < 0] = 0
    dwh = np.matmul(np.transpose(x), o * np.matmul((prediction - target)/target.shape[0], np.transpose(wo))) #average
    dbh = np.matmul(ones, o * np.matmul((prediction - target)/target.shape[0], np.transpose(wo))) #average
    return dwo, dbo, dwh, dbh
```

Figure 6: Backpropagation Algorithm

Note that $o = w_o h + b_o$, and $h = ReLU(w_h x + b_h) = (max((w_h x + b_h), 0))$.

$$\begin{aligned}\frac{\partial L}{\partial w_o} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial w_o}\\ &= (p_j - y_j)\frac{\partial o}{\partial w_o}\\ &= (p_j - y_j)h\\ \frac{\partial L}{\partial b_o} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial b_o}\\ &= (p_j - y_j)\frac{\partial o}{\partial b_o}\\ &= (p_j - y_j)\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial w_h} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial h}\frac{\partial h}{\partial w_h}\\ &= (p_j - y_j)w_o\frac{\partial h}{\partial w_h}\\ &= (p_j - y_j)w_o\frac{\partial}{\partial w_h}(max((w_h x + b_h), 0))\\ &= \begin{cases}(p_j - y_j)w_o x & w_h x + b_h \geq 0\\ 0 & w_h x + b_h < 0\end{cases}\\ \frac{\partial L}{\partial b_h} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial h}\frac{\partial h}{\partial b_h}\\ &= (p_j - y_j)w_o\frac{\partial h}{\partial b_h}\\ &= (p_j - y_j)w_o\frac{\partial}{\partial b_h}(max((w_h x + b_h), 0))\\ &= \begin{cases}(p_j - y_j)w_o & w_h x + b_h \geq 0\\ 0 & w_h x + b_h < 0\end{cases}\end{aligned}$$

3 Learning [6 points]

We initialize the weight matrices with the Xaiver initialization scheme, bias vectors to zero, gamma value of 0.9, learning rate of 0.1 with 1000 hidden units over 200 epochs. Using the backpropagation optimization technique of Gradient Descent with momentum, and softmax and ReLU activations, we observe the training and validation loss and accuracy curves in Figures 7 and 8, respectively.

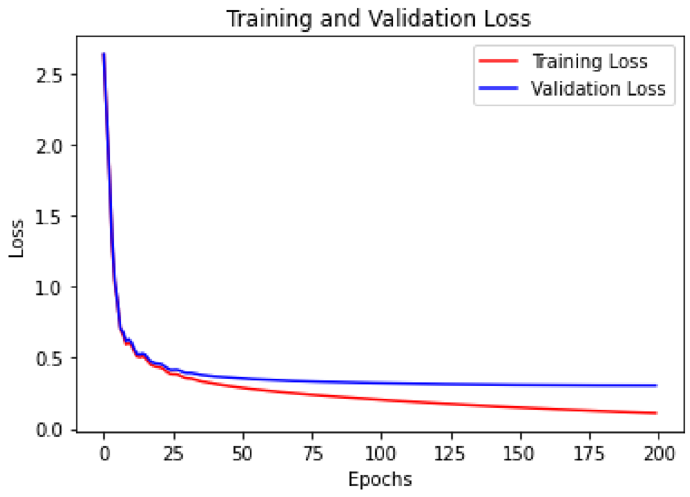


Figure 7: Training and Validation Loss

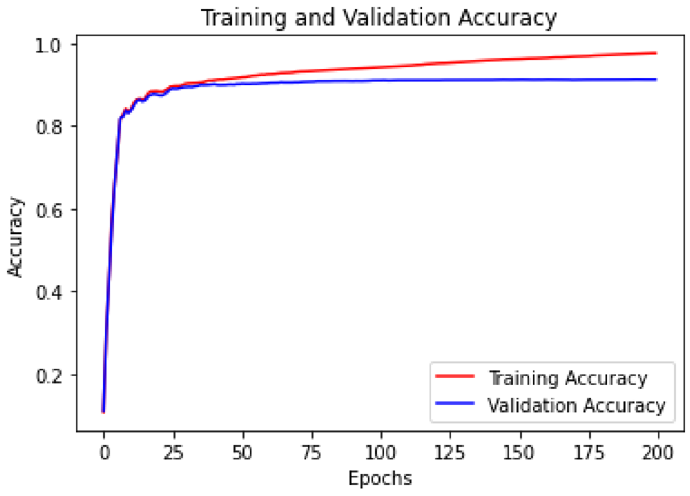


Figure 8: Training and Validation Accuracy

As a final remark, the final training accuracy is 0.9768 and the final validation accuracy is 0.9128.

4 Code

```
# -*- coding: utf-8 -*-
"""Assignment_2

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/18
        WD4ZTZxynhbjXZcl351tcHvncogr2zS
"""

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

# Load the data
def load_data():
    with np.load('/content/gdrive/My_Drive/ECE421/Assignment_2/notMNIST
        .npz') as data:
        data, targets = data["images"], data["labels"]

        np.random.seed(521)
        rand_idx = np.arange(len(data))
        np.random.shuffle(rand_idx)

        data = data[rand_idx] / 255.0
        targets = targets[rand_idx].astype(int)

        train_data, train_target = data[:10000], targets[:10000]
        valid_data, valid_target = data[10000:16000], targets
            [10000:16000]
        test_data, test_target = data[16000:], targets[16000:]
    return train_data, valid_data, test_data, train_target,
        valid_target, test_target


def convert_onehot(train_target, valid_target, test_target):
    new_train = np.zeros((train_target.shape[0], 10))
    new_valid = np.zeros((valid_target.shape[0], 10))
    new_test = np.zeros((test_target.shape[0], 10))

    for item in range(0, train_target.shape[0]):
        new_train[item][train_target[item]] = 1
    for item in range(0, valid_target.shape[0]):
        new_valid[item][valid_target[item]] = 1
    for item in range(0, test_target.shape[0]):
        new_test[item][test_target[item]] = 1
    return new_train, new_valid, new_test



def shuffle(data, target):
    np.random.seed(421)
    rand_idx = np.random.permutation(len(data))
    return data[rand_idx], target[rand_idx]


# Implementation of a neural network using only Numpy - trained using
    gradient descent with momentum
def relu(x):
    return np.maximum(x, 0)

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

def compute_layer(x, w, b):
    return np.matmul(x, w) + b

def average_ce(target, prediction):
    prediction[prediction == 0] = 0.00000000000001 #prevents divide by
        zero
    return -np.sum(np.multiply(target, np.log(prediction))) / target.
        shape[0]

def grad_ce(target, logits):
    return (softmax(logits) - target) #prediction - target

def backpropagation(prediction, target, o, wh, wo, x, b):
    dwo = np.matmul(np.transpose(relu(o)), (prediction - target)/target
        .shape[0]) #average
    ones = np.ones((1, target.shape[0])) #initialize ones
    dbo = np.matmul(ones, (prediction - target)/target.shape[0]) #
        average
    o[o >= 0] = 1
    o[o < 0] = 0
    dwh = np.matmul(np.transpose(x), o * np.matmul((prediction - target
        )/target.shape[0], np.transpose(wo))) #average
    dbh = np.matmul(ones, o * np.matmul((prediction - target)/target.
        shape[0], np.transpose(wo))) #average
    return dwo, dbo, dwh, dbh

def learning():
    #data cleaning
    train_data, valid_data, test_data, train_target, valid_target,
        test_target = load_data()
    train_data = train_data.reshape(np.shape(train_data)[0], np.shape(
        train_data)[1]*np.shape(train_data)[2])
    valid_data = valid_data.reshape(np.shape(valid_data)[0], np.shape(
        valid_data)[1]*np.shape(valid_data)[2])
    test_data = test_data.reshape(np.shape(test_data)[0], np.shape(
        test_data)[1]*np.shape(test_data)[2])
    new_train, new_valid, new_test = convert_onehot(train_target,
        valid_target, test_target)

    #initializations
    epoch = 200
    H = 1000
    F = train_data.shape[1]
    K = 10
    gamma = 0.9
    alpha = 0.1 #without average use 0.1/target.shape[0] scaling
    wo = np.random.normal(0, np.sqrt(2/(H + 10)), (H, 10)) #variance of
         2/(units in + units out), zero-mean Gaussians
    wh = np.random.normal(0, np.sqrt(2/(F + H)), (F, H))
    bo = np.zeros((1, 10)) #bias to zero
    bh = np.zeros((1, H))
    train_loss, train_acc, valid_loss, valid_acc = ([] for idx in range
        (4))

    vwh = np.full((F, H), 1e-5)
    vwo = np.full((H, 10), 1e-5)
    vbh = np.full((1, H), 1e-5)
    vbo = np.full((1, 10), 1e-5)

    for i in range(epoch):
        o = compute_layer(train_data, wh, bh)
        h = relu(o)
        logits = compute_layer(h, wo, bo)
        prediction = softmax(logits)
        train_loss.append(average_ce(new_train, prediction))
        compare = np.equal(np.argmax(prediction, axis = 1), np.argmax(
            new_train, axis = 1))
        train_acc.append(np.sum((compare==True))/(train_data.shape[0]))

        #back propagation
        dwo, dbo, dwh, dbh = backpropagation(prediction, new_train, o,
            wh, wo, train_data, bh)

        h_valid = relu(compute_layer(valid_data, wh, bh))
        logits_valid = compute_layer(h_valid, wo, bo)
        prediction_valid = softmax(logits_valid)
        valid_loss.append(average_ce(new_valid, prediction_valid))
        compare_valid = np.equal(np.argmax(prediction_valid, axis = 1),
             np.argmax(new_valid, axis = 1))
        valid_acc.append(np.sum((compare_valid==True))/(valid_data.
            shape[0]))

        #update rule
        vwh = gamma * vwh + alpha * dwh
        vwo = gamma * vwo + alpha * dwo
        vbh = gamma * vbh + alpha * dbh
        vbo = gamma * vbo + alpha * dbo
        wo = wo - vwo
        wh = wh - vwh
        bo = bo - vbo
        bh = bh - vbh

    plt.title('Training and Validation Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epochs')
    plt.plot(range(epoch), train_loss, 'r', range(epoch), valid_loss, '
        b')
    plt.legend(["Training Loss" ,"Validation Loss"], loc = 'best')
    plt.show()

    plt.title('Training and Validation Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epochs')
    plt.plot(range(epoch), train_acc, 'r', range(epoch), valid_acc, 'b'
        )
    plt.legend(["Training Accuracy" ,"Validation Accuracy"], loc = '
        best')
    plt.show()

    print("Final Training Accuracy: ", train_acc[199])
    print("Final Validation Accuracy: ", valid_acc[199])
```

```
learning()
```