Name: Tomas Rotbauer
Student Number: 1003963124

ECE421 - Assignment 2:
Neural Networks

Part 1 - Neural Networks Using Numpy

1.1 Helper Functions

1.
```
def relu(x):
    return np.maximum(x, 0)
```
2.
```
#Accepts matrix with rows of o vectors
def softmax(x):
    #Prevent overflow by normalizing
    x -= np.amax(x, axis=1)[:, None]
    return np.exp(x) / np.sum(np.exp(x), axis=1)[:,None]
```
3.
```
def computeLayer(X, W, b):
    return W.dot(X) + b[:, None]
```
4.
```
def CE(target, prediction):
    #Avoid taking unnecessary logs by doing row-wise sum of the
matrix product
    pred_of_tar = np.sum(target*prediction, axis=1)
    return (-1/target.shape[0])*np.sum(np.log(pred_of_tar))
```

5. The following is a derivation of the gradient of the cross entropy loss function with respect to the input to the softmax function **o**. First, simplify the loss function with respect to the one non-zero term, call it the y^{th} term:

$$\begin{aligned} L &= -(1) \times log(p_y) \\ &= -log\frac{e^{o_y}}{\sum_{k=1}^{K} e^{o_k}} \end{aligned}$$

Now, the gradient must account for the partial derivatives in terms of both o_y and $o_z, z \neq y$:

$$\begin{aligned} \frac{dL}{do_y} &= \frac{-\sum_{k=1}^{K} e^{o_k}}{e^{o_y}} \times \frac{e^{o_y}(\sum_{k=1}^{K} e^{o_k}) - e^{2o_y}}{(\sum_{k=1}^{K} e^{o_k})^2} \\ &= \frac{-(\sum_{k=1}^{K} e^{o_k}) + e^{o_y}}{\sum_{k=1}^{K} e^{o_k}} \end{aligned}$$

$$= -1 + \frac{e^{o_y}}{\sum_{k=1}^{K} e^{o_k}}$$

$$\frac{dL}{do_z} = \frac{-\sum_{k=1}^{K} e^{o_k}}{e^{o_y}} \times \frac{-e^{o_z + o_y}}{(\sum_{k=1}^{K} e^{o_k})^2}$$

$$= \frac{e^{o_z}}{\sum_{k=1}^{K} e^{o_k}}$$

Noting the similarity between the two cases, the overall gradient simplifies to the following high-level expression:

$$\nabla_{\bar{o}} L = softmax(\bar{o}) - \bar{y}$$

Where $\bar{y}$ is a vector for the one-hot encoded label. The following function computes the gradient derived:

```
def gradCE(target, prediction):
    return softmax(prediction) - target
```

1.2 Backpropagation Derivation

1. Using the gradient of the cross entropy loss function with respect to the input to the softmax function **o**, the gradient with respect to the output layer weights $\mathbf{W_o}$ can be computed with the chain rule:

$$\nabla_{\overline{W_o}} L = \nabla_{\bar{o}} L \bullet \frac{do}{dW}$$

$$= \left(\frac{e^{\bar{o}}}{\sum_{k=1}^{K} e^{o_k}} - \bar{y}\right) \bullet \bar{h}^T$$

$$= (softmax(\bar{o}) - \bar{y}) \bullet \bar{h}^T$$

   Note that this gradient has the form of a K by H matrix, or equivalently can be thought of as a set of K number of H sized vectors.

2. Similarly to the derivation above, the following is a derivation of the cross entropy loss function with respect to the output layer biases:

$$\nabla_{\overline{b_o}} L = \nabla_{\bar{o}} L \bullet \frac{do}{db}$$

$$= \left(\frac{e^{\bar{o}}}{\sum_{k=1}^{K} e^{o_k}} - \bar{y}\right) \circ \bar{1}$$

$$= softmax(\bar{o}) - \bar{y}$$

Name: Tomas Rotbauer
Student Number: 1003963124

3. In order to find the gradient of the cross entropy loss function with respect to the hidden layer weights $\mathbf{W_h}$, the loss was expressed in terms of the hidden layer weights $\mathbf{W_h}$ and the derivative taken directly. The one non-zero loss term has subscript y.

$$L = -log(\frac{N}{D})$$

$$= -log(\frac{exp(W_y^{(o)} \cdot ReLU(W^{(h)}\bar{x} + b^{(h)}) + b_y^{(o)})}{\sum_{k=1}^{K} exp(W_k^{(o)} \cdot ReLU(W^{(h)}\bar{x} + b^{(h)}) + b_k^{(o)})})$$

Then:

$$\frac{dL}{dW_{ij}^{(h)}} = -\frac{D}{N} \times \left[\frac{N \times ReLU'(W^{(h)}\bar{x} + b^{(h)}) \cdot (x_j)(W_{yi}^{(o)}) \times D}{D^2} - \frac{D \times ReLU'(W^{(h)}\bar{x} + b^{(h)}) \cdot (x_j)(W_{ki}^{(o)}) \times N}{D^2}\right]$$

$$= -ReLU'(\bar{h}) \cdot (\frac{x_j W_{yi}^{(o)} \sum_{l=1}^{K} exp(o_l)}{\sum_{k=1}^{K} exp(o_k)} - \frac{\sum_{l=1}^{K} exp(o_l) x_j W_{li}^{(o)}}{\sum_{k=1}^{K} exp(o_k)})$$

$$= ReLU'(\bar{h}) \cdot x_j (\sum_{l=1}^{K} \frac{exp(o_l) W_{li}^{(o)}}{\sum_{k=1}^{K} exp(o_k)} - W_{yi}^{(o)})$$

Here is the equivalent high-level vectorized expression:

$$ReLU'(\bar{h}) \circ \left[\left((W^{o})^{T} \cdot softmax(\bar{o}) - \left(W_y^{(o)}\right)^{T}\right) \cdot \bar{x}^{T}\right]$$

4. Similarly to the derivation above, The following expression is the gradient of the cross entropy loss function with respect to the hidden layer biases.

$$\frac{dL}{db_i^{(h)}} = ReLU'(\bar{h}) \cdot (\sum_{l=1}^{K} \frac{exp(o_l) W_{li}^{(o)}}{\sum_{k=1}^{K} exp(o_k)} - W_{yi}^{(o)})$$

This is the equivalent high-level vectorized expression:

$$ReLU'(\bar{h}) \circ \left[(W^{o})^{T} \cdot softmax(\bar{o}) - \left(W_y^{(o)}\right)^{T}\right]$$

1.3 Learning

The following (Python) function was used for training the neural network. (Runtime for 200 epochs was roughly 5 minutes).

```
def gradDescent():
    #Constants
    H = 1000
    K = 10
    F = 784
```

Name: Tomas Rotbauer
Student Number: 1003963124

```
EPOCHS = 200
gamma = 0.9
alpha = 0.1
N = 10000

#Training and validation data/target extraction
Data = loadData()
data = Data[0].reshape(N, 784)
validData = Data[1].reshape(6000, 784)
Targets = convertOneHot(Data[3], Data[4], Data[5])
target = Targets[0]
validTarget = Targets[1]
del Data
del Targets

#Initialization of weights and biases
Wo = np.random.normal(0, np.sqrt(2/H), (K, H))
Wh = np.random.normal(0, np.sqrt(2/F), (H, F))
bo = np.zeros(K)
bh = np.zeros(H)

#Initialization of V matrices
VWo = np.full((K, H), 1e-5)
VWh = np.full((H, F), 1e-5)
Vbo = np.full(K, 1e-5)
Vbh = np.full(H, 1e-5)

#Data containers used for plotting
xpoints = np.arange(1, EPOCHS+1)
ytrain = []
yvalid = []

#Local function for computing a forward pass.
#Training is a boolean (set false for validation).
def forward_propagation(training):
    nonlocal data, Wh, bh, Wo, bo
    if training:
        Sh = computeLayer(data.T, Wh, bh)
    else:
        Sh = computeLayer(validData.T, Wh, bh)
```

Name: Tomas Rotbauer
Student Number: 1003963124

```
        Xh = relu(Sh)
        So = computeLayer(Xh, Wo, bo)
        Xo = softmax(So.T)

        return Sh, Xh, So, Xo

    #Main training loop
    for epoch in range(EPOCHS):
        #Forward propagation
        Sh, Xh, So, Xo = forward_propagation(True)

        #Gradients/backpropagation
        grad_bo = gradCE(target, So.T)
        grad_Wo = grad_bo.T.dot(Xh.T)/N
        grad_bo = np.sum(grad_bo, axis=0)/N
        grad_bh = (Wo.T.dot(Xo.T) - Wo.T.dot(target.T))*np.array(Xh,
dtype=bool)
        grad_Wh = grad_bh.dot(data)/N
        grad_bh = np.sum(grad_bh, axis=1)/N

        #Gradient descent with momentum
        VWo = gamma*VWo + alpha*grad_Wo
        VWh = gamma*VWh + alpha*grad_Wh
        Vbo = gamma*Vbo + alpha*grad_bo
        Vbh = gamma*Vbh + alpha*grad_bh

        #Update Weights and biases
        Wo -= VWo
        Wh -= VWh
        bo -= Vbo
        bh -= Vbh

        #Add new plotting data
        ytrain.append(accuracy(target, Xo))
        yvalid.append(accuracy(validTarget,
forward_propagation(False)[3]))

    #Plot findings. Plots shown/labeled in main() function
    plt.plot(xpoints, ytrain, label = "Training")
    plt.plot(xpoints, yvalid, label = "Validation")
```

Name: Tomas Rotbauer
Student Number: 1003963124

The following function was used for computing the accuracy for a given set of predictions and targets:

```
def accuracy(target, prediction):
    #N = number of examples
    N = target.shape[0]
    return np.sum(target[np.arange(N), np.argmax(prediction, axis=1)])/N
```

The accuracy and loss was plotted against epochs for both training and validation. The hidden unit size (H) was set to 1000, momentum parameter (γ) was set to 0.9, and learning rate (α) was set to 0.1.