ECE421 Assignment 2

Student	Ⓥ Valentina Manferrari
utorID	1004796615

1. Neural Networks using Numpy

1.1 Helper Functions

ReLU

$$ReLU(x) = max(x, 0)$$

```
# This function accepts one argument and return a Numpy array with the ReLU activation.
def relu(x):
    relu_x = np.copy(x)
    relu_x = relu_x.clip(min=0)
    return relu_x
```

Softmax

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

```
# This function accepts one argument
# Returns a Numpy array with the softmax activation.
def softmax(x):
    softmax_x = np.exp(x)/np.sum(np.exp(x), axis=1, keepdims=True)
    return softmax_x
```

Compute

```
# This function accepts 3 arguments: an input vector, a weight matrix and a bias vector
# Returns the product between the weights and input plus the biases
# i.e. a prediction for a given layer.
def compute(X, W, b):
    compute_layer = np.matmul(X, W) + b
    return compute_layer
```

Average Cross Entropy

$$AverageCE = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} log(p_k^{(n)})$$

Where $y_k^{(n)}$ is the true one-hot label for sample n, p_k is the predicted class probability (i.e softmax output for the k^{th} class) of sample n, and N is the number of examples.

```
# This function accepts two arguments: the targets (labels) and predictions
# Returns the average cross entropy loss for the dataset.
def averageCE(target, prediction):
    ce = -(1/prediction.shape[0]) * np.sum(target * np.log(prediction))
    return ce
```

Cross Entropy Gradient

$$\mathcal{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} log(p_k^{(n)})$$

Where $\mathbf{o} = \mathbf{W_o x} + \mathbf{b_o}$, $\mathbf{p} = softmax(\mathbf{o}) = \frac{e^{\mathbf{o}}}{\sum_{k=1}^{K} e^{o_k}}$ and

$\mathbf{y} = [y_1, y_2, ..., y_K]^T$ is the one-hot encoded vector for the labels.

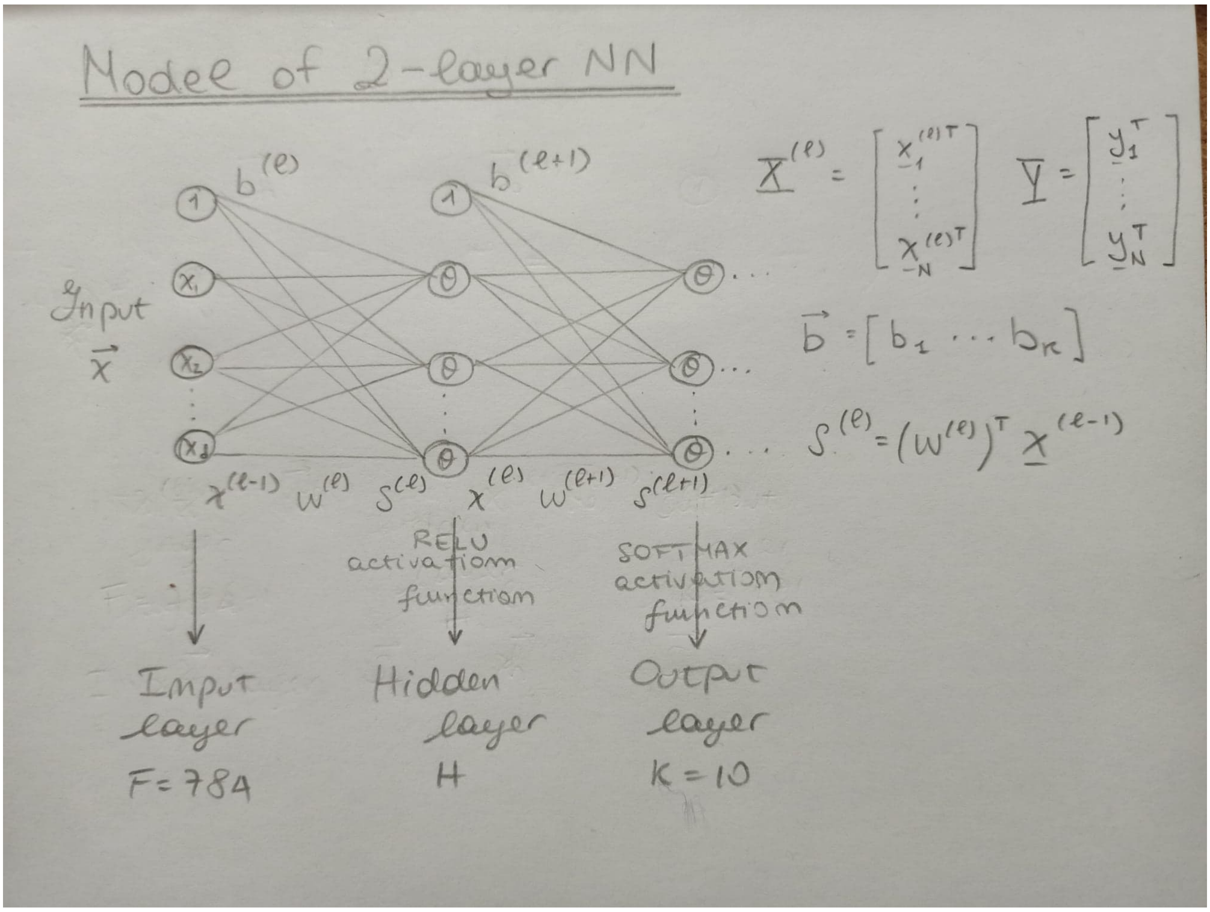
Let's derive an analytic expression for its gradient:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial p_o} &= \frac{\partial}{\partial p_o}(-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^n log(p_k^n)) \\ &= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^n \cdot \frac{\partial}{\partial p_o}(log(p_k^n)) \\ &= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^n \cdot \frac{1}{p_k^n}\end{aligned}$$

```
# This function accepts two arguments: the targets (labels)
# and the input to the softmax function.
# Returns the gradient of the cross entropy loss wrt the inputs of the softmax function.
def gradCE(target, prediction):
```

```
    grad = -(1/prediction.shape[0]) * target / prediction
    return grad
```

1.2 Backpropagation Derivation



Forward Propagation

$$s_{nj}^{(l)} = \sum_i w_{ij}^{(l)} x_{ni}^{(l-1)} + b_j^{(l)} \rightarrow S^{(l)} = [W^{(l)}]^T X^{(l-1)} + b^{(l)}$$

$$x_{nj}^{(l)} = \theta(s_{nj}^{(l)}) \rightarrow X^{(l)} = \theta(S^{(l)})$$

where θ is either ReLU or Softmax activation function depending on the layer.

Let $\delta_{nj}^{(l)} = \frac{\partial \mathcal{L}}{\partial s_{nj}^{(l)}}$ be the sensitivity of the n^{th} data point at node j layer l.

The generic formula for the gradient of the loss wrt a layer weights is:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial s_{nj}^{(l)}} \frac{\partial s_{nj}^{(l)}}{\partial w_{ij}^{(l)}}$$

$$= \sum_{n} (x_{in}^{(l-1)})^T \cdot \delta_{nj}^{(l)} = [X^{(l-1)}]^T \delta^{(l)}$$

The generic formula for the gradient of the loss wrt a layer biases is:

$$\frac{\partial \mathcal{L}}{\partial b_{j}^{(l)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial s_{nj}^{(l)}} \frac{\partial s_{nj}^{(l)}}{\partial b_{j}^{(l)}}$$

$$= \sum_{n} \delta_{nj}^{(l)} \cdot 1 = 1^T \delta^{(l)}$$

Backpropagation

Now let's calculate $\delta^{(l)}$ using backpropagation:

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial s_{nj}^{(l)}} = \frac{\partial \mathcal{L}}{\partial x_{nj}^{(l)}} \frac{\partial x_{nj}^{(l)}}{\partial s_{nj}^{(l)}}$$

$$= (\sum_{k} \delta_{nk}^{(l+1)} w_{jk}^{(l+1)}) \cdot \theta(s_{nj}^{(l)}) = (W^{(l+1)} \delta^{(l+1)}) \otimes \theta(S^{(l)})$$

Here $\theta(S^{(l)}) = sign(S^{(l)})$ since the hidden layer of out neural network has ReLu activation function.

We also need to find $\delta^{(L)}$ i.e. the sensitivity for the output layer:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial s_{nj}^{(L)}} = \sum_k \left(\frac{\partial \mathcal{L}}{\partial x_{nk}^{(L)}} \frac{\partial x_{nk}^{(L)}}{\partial s_{nj}^{(L)}}\right)$$

$$= \sum_k \frac{\partial}{\partial x_{nk}^{(L)}} \left(-\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k^{(n)} log(x_{nk}^{(L)})\right) \cdot \frac{\partial}{\partial s_{nj}^{(L)}} (\sigma(s_{nk}^{(L)}))$$

$$= -\frac{1}{N} \sum_k \frac{y_{nk}}{x_{nk}^{(L)}} \cdot x_{nk}^{(L)} (\delta_{kj} - x_{nj}^{(L)}) = -\frac{1}{N} \sum_k y_{nk} (\delta_{kj} - x_{nj}^{(L)})$$

$$= -\frac{1}{N} \sum_k (y_{nk} \delta_{kj} - y_{nk} x_{nj}^{(L)}) = -\frac{1}{N} (Y \cdot I - X^{(L)})$$

$$\delta^{(L)} = \frac{1}{N} (X^{(L)} - Y)$$

Note: here we use Softmax activation because L is the output layer.

Note: δ_{kj} is the Dirac delta function.

Derivation for 2-Layer NN

1. The gradient of the loss with respect to the output layer weights

$$\frac{\partial \mathcal{L}}{\partial W_o} = \frac{\partial \mathcal{L}}{\partial W^{(2)}} = [X^{(1)}]^T \delta^{(2)} = \frac{1}{N} [X^{(1)}]^T (X^{(2)} - Y)$$

2. The gradient of the loss with respect to the output layer biases

$$\frac{\partial \mathcal{L}}{\partial b_o} = \frac{\partial \mathcal{L}}{\partial b^{(2)}} = 1^T \delta^{(2)} = \frac{1}{N} 1^T (X^{(2)} - Y)$$

3. The gradient of the loss with respect to the hidden layer weights

$$\frac{\partial \mathcal{L}}{\partial W_h} = \frac{\partial \mathcal{L}}{\partial W^{(1)}} = [X^{(0)}]^T \delta^{(1)} = \frac{1}{N} [X^{(0)}]^T (\delta^{(2)} [W^{(2)}]^T) \otimes sign(S^{(1)})$$

4. The gradient of the loss with respect to the hidden layer biases

$$\frac{\partial \mathcal{L}}{\partial b_h} = \frac{\partial \mathcal{L}}{\partial b^{(1)}} = 1^T \delta^{(1)} = \frac{1}{N} 1^T (\delta^{(2)} [W^{(2)}]^T) \otimes sign(S^{(1)})$$

1.3 Learning

Computing a forward pass of the training data and implementing the backpropagation algorithm using gradients derived in Section 1.2. The optimization technique used for backpropagation is Gradient Descent with momentum.

```
def forward_propagation(dataIN, W, b):
    X, S = [None]*3, [None]*3
    X[0] = dataIN
    # updating hidden layer
    S[1] = np.matmul(X[0], W[1]) + b[1]
    X[1] = relu(S[1])
    # updating output layer
    S[2] = np.matmul(X[1], W[2]) + b[2]
    X[2] = softmax(S[2])
    return X, S

def backpropagation(X, S, W, label):
    delta = [None]*3
    N = label.shape[0]
    delta[2] = (1/N) * (X[2] - label)
    derivative_relu = np.zeros_like(S[1])
    derivative_relu[S[1] > 0] = 1
    delta[1] = np.matmul(delta[2], (W[2]).T) * derivative_relu # backpropagation
    return delta

def compute_gradients(dataIN, label, W, b):
    gradW = [0]*3
    gradb = [0]*3
    N = dataIN.shape[0]
    # run forward propagation
    X, S = forward_propagation(dataIN, W, b)
    # run backpropagation
    delta = backpropagation(X, S, W, label)
    # calculate gradient of output layer weiths and biases
    gradW[2] = np.matmul((X[1]).T, delta[2])
    gradb[2] = np.sum(delta[2], axis=0)
    # calculate gradient of hidden layer weiths and biases
    gradW[1] = np.matmul((X[0]).T, delta[1])
    gradb[1] = np.sum(delta[1], axis=0)
    return gradW, gradb
```

Using the above implemented functions construct NN training function:

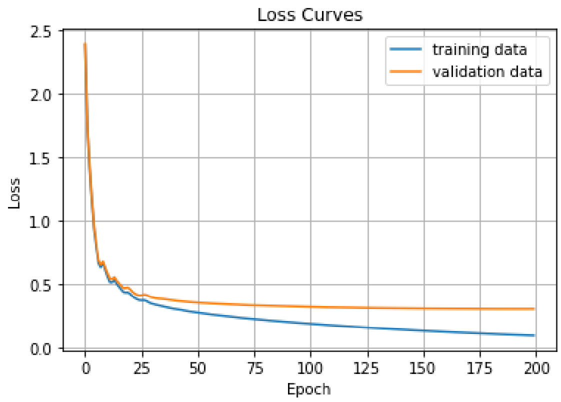
```
def training(data_train, label_train, epochs, alpha, gamma, hidden_units):
    input_neurons = data_train.shape[1]
    output_neurons = label_train.shape[1]

    # Initialize weights
    W = []
    W.append(None)
```

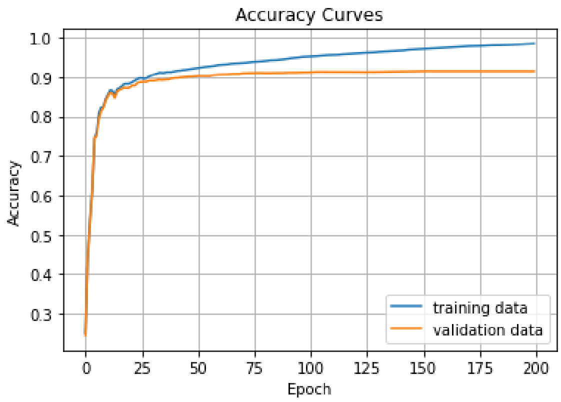
```python
    W.append(xavier_init(input_neurons, hidden_units, output_neurons))
    W.append(xavier_init(hidden_units, output_neurons, 1))
    # initialize biases
    b = []
    b.append(None)
    b.append(np.zeros((1, hidden_units)))
    b.append(np.zeros((1, output_neurons)))

    W_o = np.ones_like(W[2]) * 1e-5
    W_h = np.ones_like(W[1]) * 1e-5
    b_o = np.ones_like(b[2]) * 1e-5
    b_h = np.ones_like(b[1]) * 1e-5

    # Initialize Loss/Accuracy vectors
    loss_train, loss_valid = [], []
    acc_train, acc_valid = [], []

    for i in range(epochs):
        gradW, gradb = compute_gradients(data_train, label_train, W, b)
        # Update hidden layer
        W_h = gamma * W_h + alpha * gradW[1]
        W[1] = W[1] - W_h
        b_h = gamma * b_h + alpha * gradb[1]
        b[1] = b[1] - b_h
        # Update output layer
        W_o = gamma * W_o + alpha * gradW[2]
        W[2] = W[2] - W_o
        b_o = gamma * b_o + alpha * gradb[2]
        b[2] = b[2] - b_o

        # Measuring performance
        train_loss, train_acc, valid_loss, valid_acc = measure_performance(W, b)
        loss_train.append(train_loss)
        acc_train.append(train_acc)
        loss_valid.append(valid_loss)
        acc_valid.append(valid_acc)

    # Print out final training and validation loss/accuracy
    print("Training Loss: ", loss_train[-1])
    print("Training Accuracy: ", acc_train[-1])
    print("Validation Loss: ", loss_valid[-1])
    print("Validation Accuracy: ", acc_valid[-1])

    # Plot training an validation loss in one figure
    plt.figure()
    plt.plot(loss_train, label='training data')
    plt.plot(loss_valid, label='validation data')
    plt.legend()
    plt.title('Loss Curves')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(loc="best")
    plt.grid()
    plt.draw()
    # Plot training an validation accuracy in one figure
    plt.figure()
    plt.plot(acc_train, label='training data')
    plt.plot(acc_valid, label='validation data')
```

```
    plt.legend()
    plt.title('Accuracy Curves')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(loc="best")
    plt.grid()
    plt.draw()
```

Loading the data from notMNIST.npz and calling training function to train the neural network for 200 epochs with a hidden unit size of H=1000. Average loss is set to 0.1 and momentum to 0.9.

```
# --- MAIN --- #
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
X_train = trainData.reshape(trainData.shape[0], -1)
X_valid = validData.reshape(validData.shape[0], -1)
X_test = testData.reshape(testData.shape[0], -1)
Y_train, Y_valid, Y_test = convertOneHot(trainTarget, validTarget, testTarget)

training(X_train, Y_train, epochs=200, alpha=0.1, gamma=0.9, hidden_units=1000)
```






```
Training Loss:      0.0918319125408706
Training Accuracy: 0.9842
Validation Loss:      0.30177785629824094
Validation Accuracy: 0.9143333333333333
```

```
        """
        Extra helper functions used in Training function
        """

def xavier_init(neurons_in, n_units, neurons_out):
```

```python
    shape = (neurons_in, n_units)
    var = 2./(neurons_in + neurons_out)
    W = np.random.normal(0, np.sqrt(var), shape)
    return W

def accuracy(label, label_pred):
    j = np.argmax(label_pred, axis=1)
    i = np.arange(label.shape[0])
    return np.mean(label[i, j])

def measure_performance(W, b):
    Y_pred, S = forward_propagation(X_train, W, b)
    train_loss = avgCE(Y_train, Y_pred[2])
    train_acc = accuracy(Y_train, Y_pred[2])
    Y_pred, S = forward_propagation(X_valid, W, b)
    valid_loss = avgCE(Y_valid, Y_pred[2])
    valid_acc = accuracy(Y_valid, Y_pred[2])
    return train_loss, train_acc, valid_loss, valid_acc
```