ECE421 - Winter 2022
Assignment 2: Neural Networks

Faranak Dayyani, Student Number: 1002373674

Part 1: Neural Networks using Numpy

1.1: Helper Functions

1. relu():

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

Figure 1: relu() function code snippet

2. softmax():

```
def softmax(x):

    x = x - x.max(axis=1, keepdims=True)
    z = np.exp(x) / np.exp(x).sum(axis=1, keepdims=True)

    return z
```

Figure 2: softmax() function code snippet

3. compute_layer():

```
def compute_layer(x, w, b):

    product = x @ w + b

    return product
```

Figure 3: compute_layer() function code snippet

4. average_ce():

```
def average_ce(target, prediction):

    avg_ce = -np.mean(target*np.log(prediction))

    return avg_ce
```

Figure 4: average_ce() function code snippet

5. grad_ce():

Analytical expression:

$$L = -\sum_{k=1}^{k} y_k \log(p_k) \quad , \quad \frac{\partial L}{\partial O} = ? \ , \ \frac{\partial L}{\partial O} = \frac{\partial L}{\partial P_i} \times \frac{\partial P_i}{\partial O_i}$$

$$p_i = softmax(O_i) = \frac{e^{O_i}}{\sum_{k=1}^{k} e^{O_k}} \ , i = 1, \ldots, k \ for \ k \ classes$$

$$\frac{\partial L}{\partial P_i} = -\sum_{k=1}^{K} y_i \frac{1}{p_i}$$

$$for \ i = j => \ \frac{\partial P_i}{\partial O_i} = \frac{e^{O_i}\left(\sum_{k=1}^{k} e^{O_k}\right) - e^{O_i}e^{O_i}}{(\sum_{k=1}^{k} e^{O_k})^2}$$

$$\frac{\partial P_i}{\partial O_i} = \frac{e^{O_i}}{(\sum_{k=1}^{k} e^{O_k})} \frac{\left(\sum_{k=1}^{k} e^{O_k}\right)}{(\sum_{k=1}^{k} e^{O_k})} - \frac{e^{O_i}e^{O_i}}{(\sum_{k=1}^{k} e^{O_k})^2}$$

$$\frac{\partial P_i}{\partial O_i} = p_i - p_i^2 = p_i(1 - p_i)$$

$$for \ i \neq j => \ \frac{\partial P_i}{\partial O_j} = \frac{0 - e^{O_i}e^{O_j}}{(\sum_{k=1}^{k} e^{O_k})^2}$$

$$\frac{\partial P_i}{\partial O_j} = -\frac{e^{O_i}}{(\sum_{k=1}^{k} e^{O_k})} \frac{e^{O_j}}{(\sum_{k=1}^{k} e^{O_k})}$$

$$\frac{\partial P_i}{\partial O_j} = -p_i p_j$$

$$\frac{\partial L}{\partial O} = \frac{\partial L}{\partial P_i} \times \frac{\partial P_i}{\partial O_i} + \frac{\partial L}{\partial P_i} \times \frac{\partial P_i}{\partial O_j}$$

$$\frac{\partial L}{\partial O} = \left(-\sum_{i=j}^{k} y_i \frac{1}{p_i}\right)\left(p_i(1 - p_i)\right) + \left(-\sum_{i \neq j}^{k} y_i \frac{1}{p_i}\right)(-p_i p_j)$$

$$\frac{\partial L}{\partial O} = \sum_{i} -y_i + p_i y_i + p_i y_i = p_i \sum_{i} y_i - y_i = p_i - y_i$$

$$\frac{\partial L}{\partial O} = \frac{1}{N}(p - y)$$

```python
def grad_ce(target, logits):

    gr_ce = (softmax(logits) - target)/target.shape[0]

    return gr_ce
```

Figure 5: grad_ce() function code snippet

1.2: Backpropagation Derivation

1. The gradient of the loss with respect to the output layer weights ($\frac{\partial L}{\partial W_o}$):

$$O = w_O h + b_O => \frac{\partial O}{\partial W_o} = h$$

$$\frac{\partial L}{\partial W_o} = \frac{\partial O}{\partial W_o} \times \frac{\partial L}{\partial O} = h^T \times (p - y)$$

```python
def grad_wrt_wO(h, p, y):

    p_y = grad_ce(y, p)
    h_tran = np.transpose(h)
    gradwO = np.matmul(h_tran, p_y)

    return gradwO
```

Figure 6: grad_wrt_wO function code snippet

2. The gradient of the loss with respect to the output layer biases ($\frac{\partial L}{\partial b_o}$):

$$O = w_O h + b_O => \frac{\partial O}{\partial b_o} = 1$$

$$\frac{\partial L}{\partial W_o} = \frac{\partial O}{\partial b_o} \times \frac{\partial L}{\partial O} = 1^T \times (p - y)$$

```python
def grad_wrt_bO(p, y):

    p_y = grad_ce(y, p)
    ones_shape = np.ones((1,y.shape[0]))
    gradbO = np.matmul(ones_shape, p_y)

    return gradbO
```

Figure 7: grad_wrt_bO function code snippet

3. The gradient of the loss with respect to the hidden layer weights ($\frac{\partial L}{\partial W_h}$):

$$O = w_O h + b_O \Rightarrow \frac{\partial O}{\partial h} = w_O$$

$$\frac{\partial L}{\partial O} = p - y$$

$$z = w_h x + b_h \Rightarrow \frac{\partial z}{\partial w_h} = x$$

$$\frac{\partial h}{\partial z} = \begin{cases} 0, & if\ z_i < 0 \\ 1, & if\ z_i > 0 \end{cases}$$

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial O} \times \frac{\partial O}{\partial h} \times \frac{\partial h}{\partial z} \times \frac{\partial z}{\partial w_h} = (p - y) w_O^T \times \frac{\partial h}{\partial z} x^T$$

```python
def grad_wrt_wh(p, y, x, h_wh, w_O):

    h_wh [h_wh > 0] = 1
    h_wh [h_wh < 0] = 0

    x_tran = x.transpose()
    p_y = grad_ce(y, p)

    part1 = np.matmul(p_y, np.transpose(w_O))
    part2 = h_wh * part1

    grad_wh = np.matmul(x_tran, part2)

    return grad_wh
```

Figure 8: grad_wrt_wh function code snippet

4. The gradient of the loss with respect to the hidden layer biases ($\frac{\partial L}{\partial b_h}$):

$$O = w_O h + b_O \Rightarrow \frac{\partial O}{\partial h} = w_O$$

$$\frac{\partial L}{\partial O} = p - y$$

$$z = w_h x + b_h \Rightarrow \frac{\partial z}{\partial b_h} = 1$$

$$\frac{\partial h}{\partial z} = \begin{cases} 0, & if\ z_i < 0 \\ 1, & if\ z_i > 0 \end{cases}$$

$$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial O} \times \frac{\partial O}{\partial h} \times \frac{\partial h}{\partial z} \times \frac{\partial z}{\partial b_h} = (p-y)w_O^T \times \frac{\partial h}{\partial z} 1^T$$

```
def grad_wrt_bh(p, y, x, h_wh, w_O):

    h_wh [h_wh > 0] = 1
    h_wh [h_wh < 0] = 0

    p_y = grad_ce(y, p)
    ones_shape = np.ones((x.shape[0], 1))
    ones_shape_tran = ones_shape.transpose()

    part1 = np.matmul(p_y, np.transpose(w_O))
    part2 = h_wh * part1

    grad_bh = np.matmul(ones_shape_tran, part2)

    return grad_bh
```

Figure 9: grad_wrt_bh function code snippet

1.3: Learning

The model was initialized with the values below:

- The number of hidden units are 1000 (H=1000).
- The hidden layer weights are initialized with zero mean gaussian distribution with variance of $\frac{2}{units\ in + units\ out} = \frac{2}{784+1000}$.
- The output layer weights are initialized with zero mean gaussian distribution with variance of $\frac{2}{units\ in + units\ out} = \frac{2}{1000+10}$.
- The biases are initialized to zero.
- The learning rate (α) is set to 0.1.
- γ is set to 0.99.
- The number of epochs that the model was run for is 200.

Table 1: The table below shows the loss and accuracy values for training, validation and testing data sets:

	Loss	Accuracy
Training set	0.054638	89.13 %
Validation set	0.060305	87.88 %
Testing set	0.060478	88.36 %

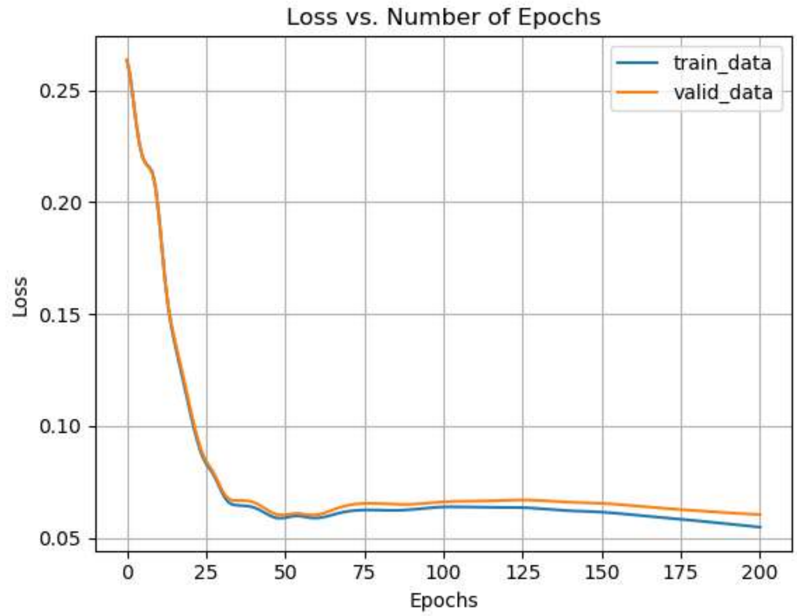

Figure 10: Loss vs. Number of Epochs for training and validation set

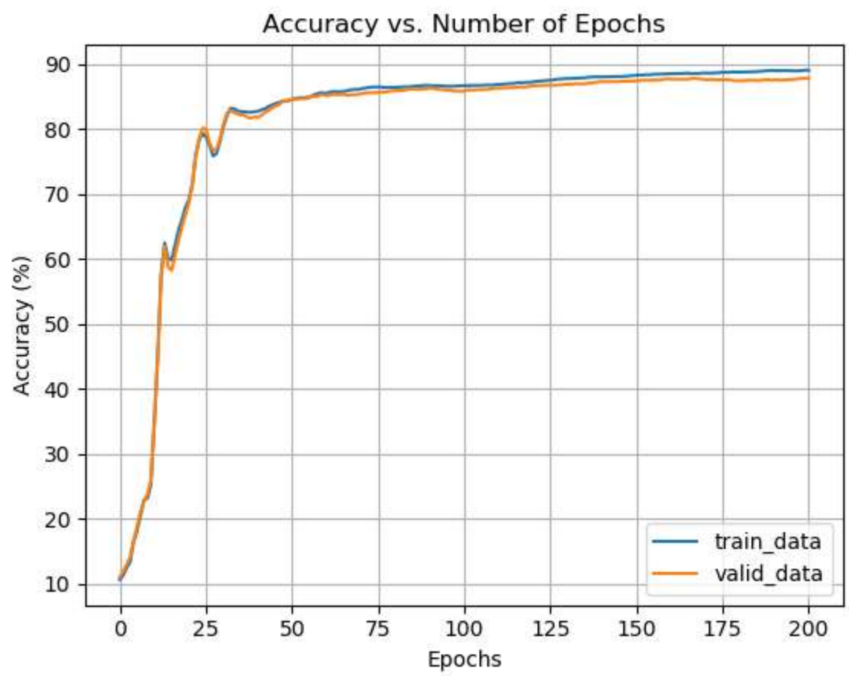

Figure 11: Accuracy vs. Number of Epochs for training and validation set

The code snippet for this part can be seen below:

```python
def learning(train_data, train_target, valid_data, valid_target, w_o, b_o, v_o, w_h, b_h, v_h, epochs, gamma, alpha):

    global i, b_o_1, b_h_1

    # creating array variables for training and validation loss and accuracies
    train_loss = []
    train_acc = []
    valid_loss = []
    valid_acc = []

    b_o_1 = b_o
    b_h_1 = b_h

    for i in range(epochs):

        # ------------ forward propagation ---------------------
        # training data: calculating loss and accuracy
        #                hidden layer
        producttrain_h = compute_layer(train_data, w_h, b_h)
        re_lu_train_h = relu(producttrain_h)
        #                output layer
        producttrain_o = compute_layer(re_lu_train_h, w_o, b_o)
        y_hat_train = softmax(producttrain_o)
        avg_ce_train = average_ce(new_train, y_hat_train)
        train_loss.append(avg_ce_train)
        acc1 = np.argmax(y_hat_train, axis=1)
        acc11 = np.argmax(new_train, axis=1)
        acc111 = np.sum(acc1 == acc11)/train_data.shape[0]
        acc111 = acc111*100
        train_acc.append(acc111)

        # validation data: calculating loss and accuracy
        #                hidden layer
        productvalid_h = compute_layer(valid_data, w_h, b_h)
        re_lu_valid_h = relu(productvalid_h)
        #                output layer
        productvalid_o = compute_layer(re_lu_valid_h, w_o, b_o)
        y_hat_valid = softmax(productvalid_o)
        avg_ce_valid = average_ce(new_valid, y_hat_valid)
```

Figure 12: learning() function code snippet – 1

```python
        avg_ce_valid = average_ce(new_valid, y_hat_valid)
        valid_loss.append(avg_ce_valid)
        acc2 = np.argmax(y_hat_valid, axis=1)
        acc22 = np.argmax(new_valid, axis=1)
        acc222 = np.sum(acc2 == acc22)/valid_data.shape[0]
        acc222 = acc222*100
        valid_acc.append(acc222)


        # ----------- Backpropagation --------------------
        #                Output layer
        gradw0 = grad_wrt_w0(re_lu_train_h, producttrain_o, new_train)
        v_o = (gamma * v_o) + (alpha * gradw0)
        w_o = w_o - v_o

        gradb0 = grad_wrt_b0(producttrain_o, new_train)
        b_o_1 = (gamma * b_o_1) + (alpha * gradb0)
        b_o = b_o - b_o_1

        #                Hidden layer
        grad_wh = grad_wrt_wh(producttrain_o, new_train, train_data, producttrain_h, w_o)
        v_h = (gamma * v_h) + (alpha * grad_wh)
        v_h = w_h - v_h

        grad_bh = grad_wrt_bh(producttrain_o, new_train, train_data, producttrain_h, w_o)
        b_h_1 = (gamma * b_h_1) + (alpha * grad_bh)
        b_h = b_h - b_h_1

        print("Going through the Loop: Epoch ", i+1,"/", epochs)

    return w_o, b_o, w_h, b_h, train_loss, train_acc, valid_loss, valid_acc
```

Figure 13: learning() function code snippet – 2

```python
if __name__ == "__main__":
    train_data, valid_data, test_data, train_target, valid_target, test_target = load_data()
    train_data = train_data.reshape([illegible], [illegible])
    valid_data = valid_data.reshape([illegible], [illegible])
    test_data = test_data.reshape([illegible], [illegible])

    new_train, new_valid, new_test = convert_onehot(train_target, valid_target, test_target)

    # ---------------------------------------- section 1.3 ----------------------------------------------------
    epochs = [illegible]
    H = [illegible]
    gamma = [illegible]
    alpha = [illegible]

    # initializing weights with Xaiver initialization scheme:
    # bias set to 0
    # initial momentum value of 1e-5

    # -------- output layer initialization ------------
    limit_o = np.sqrt([illegible]/(H+[illegible]))
    w_o = np.random.normal([illegible], limit_o, (H,[illegible]))
    v_o = np.full((H,[illegible]), [illegible])
    b_o = np.zeros(([illegible],[illegible]))

    # -------- hidden layer initialization ------------
    limit_h = np.sqrt([illegible]/(train_data.shape[[illegible]]+H))
    w_h = np.random.normal([illegible], limit_h, (train_data.shape[[illegible]],H))
    v_h = np.full((train_data.shape[[illegible]], H), [illegible])
    b_h = np.zeros(([illegible], H))

    w_o, b_o, w_h, b_h, train_loss, train_acc, valid_loss, valid_acc = learning(train_data, new_train, valid_data, new_valid, w_o, b_o,
                                                                              v_o, w_h, b_h, v_h, epochs, gamma, alpha/[illegible])
```

Figure 14: main() function code snippet – 1

```
    # Training data result:
    # ------ hidden layer ------
    product_h_train = compute_layer(train_data, w_h, b_h)
    relu_h_train = relu(product_h_train)
    # ------ output layer ------
    product_o_train = compute_layer(relu_h_train, w_o, b_o)
    yhat_train = softmax(product_o_train)
    avgce_train = average_ce(new_train, yhat_train)
    train_loss.append(avgce_train)
    acctrain1 = np.argmax(yhat_train, axis=1)
    acctrain11 = np.argmax(new_train, axis=1)
    acctrain111 = np.sum(acctrain1 == acctrain11)/train_data.shape[0]
    acctrain111 = acctrain111*100
    train_acc.append(acctrain111)
    print(" -------- Training Data ----------")
    print("Training loss is:", train_loss[-1])
    print("Training accuracy is:", train_acc[-1]," %")

    # Validation data result:
    # ------ hidden layer ------
    product_h_valid = compute_layer(valid_data, w_h, b_h)
    relu_h_valid = relu(product_h_valid)
    # ------ output layer ------
    product_o_valid = compute_layer(relu_h_valid, w_o, b_o)
    yhat_valid = softmax(product_o_valid)
    avgce_valid = average_ce(new_valid, yhat_valid)
    valid_loss.append(avgce_valid)
    accvalid1 = np.argmax(yhat_valid, axis=1)
    accvalid11 = np.argmax(new_valid, axis=1)
    accvalid111 = np.sum(accvalid1 == accvalid11)/valid_data.shape[0]
    accvalid111 = accvalid111*100
    valid_acc.append(accvalid111)
    print("\n -------- Validation Data ----------")
    print("Validation loss is:", valid_loss[-1])
    print("Validation accuracy is:", valid_acc[-1]," %")
```

Figure 15: main() function code snippet – 2

```
    # Testing data result:
    test_loss = []
    test_acc = []
    # ------ hidden layer ------
    product_h_test = compute_layer(test_data, w_h, b_h)
    relu_h_test = relu(product_h_test)
    # ------ output layer ------
    product_o_test = compute_layer(relu_h_test, w_o, b_o)
    yhat_test = softmax(product_o_test)
    avgce_test = average_ce(new_test, yhat_test)
    test_loss.append(avgce_test)
    acctest1 = np.argmax(yhat_test, axis=1)
    acctest11 = np.argmax(new_test, axis=1)
    acctest111 = np.sum(acctest1 == acctest11)/test_data.shape[0]
    acctest111 = acctest111*100
    test_acc.append(acctest111)
    print("\n -------- Testing Data ----------")
    print("Testing loss is:", test_loss[-1])
    print("Testing accuracy is:", test_acc[-1]," %")



    # plotting loss for training and validation data:
    xaxis = np.linspace(0, epochs, num=epochs)
    plt.plot(xaxis, train_loss[0:epochs])
    plt.plot(xaxis, valid_loss[0:epochs])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss vs. Number of Epochs')
    plt.legend(['train_data', 'valid_data'])
    plt.grid()
    plt.show()
    plt.figure()
```

Figure 16: main() function code snippet – 3

```
    # plotting accuracy for training and validation data:
    plt.plot(xaxis, train_acc[0:epochs])
    plt.plot(xaxis, valid_acc[0:epochs])
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy vs. Number of Epochs')
    plt.legend(['train_data','valid_data'])
    plt.grid()
    plt.show()
```

Figure 17: main() function code snippet - 4