ECE421 - Winter 2022 Assignment 2: Neural Networks

Raymond Hong 1003942629

Code written in Google Colab:
https://colab.research.google.com/drive/1RjgqKQLAmTCNajc-Iq9aQfnFUI_W22eW?usp=sharing

1. Helper Functions

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

def softmax(x):
    x = x - (np.max(x, axis=1)).reshape(x.shape[0],1)

    return np.exp(x) / (np.sum(np.exp(x), axis=1)).reshape(x.shape[0],1)

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

def average_ce(target, prediction):
    N = target.shape[0]
    return -np.sum(target*np.log(prediction)) / N

def grad_ce(target, prediction):
    return prediction - target
```

Following is the derivation of the **grad_ce** method:

$$p = \text{softmax}(z) = \frac{e^{z}}{\sum_{k=1}^{K} e^{z_k}}$$

$$L_{CE} = -\sum_{k=1}^{K} y_k \log p_k$$

$$\frac{\partial L_{CE}}{\partial z} = \frac{\partial L_{CE}}{\partial p} \cdot \frac{\partial p}{\partial z} =$$

(1) $$\frac{\partial L_{CE}}{\partial p} = -y^{T} \frac{1}{p}$$

(2) $$\frac{\partial p}{\partial z} = \frac{e^{z}\left(\sum_{k=1}^{K} e^{z_k}\right) - (e^{z})^{T} e^{z}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{e^{z}}{\sum_{k=1}^{K} e^{z_k}} \cdot \frac{\sum_{k=1}^{K} e^{z_k} - e^{z}}{\sum_{k=1}^{K} e^{z_k}}$$

$$= p(1-p)$$

$$\frac{\partial L_{CE}}{\partial z} = \left(-y^{T} \frac{1}{p}\right)\left(p(1-p)\right)$$

$$\boxed{\frac{\partial L_{CE}}{\partial z} = p - y}$$

2. Backpropagation Derivation

```python
def output_weight(target, prediction, hidden_out):
    softmax_ce = grad_ce(target, prediction)
    hidden_out_transpose = np.transpose(hidden_out)
    return  np.matmul(hidden_out_transpose, softmax_ce)

def output_bias(target, prediction):
    softmax_ce = grad_ce(target, prediction)
    ones = np.ones((1, target.shape[0]))
    return np.matmul(ones, softmax_ce)

def hidden_weight(target, prediction, input, input_out, out_weight):
    input_out[input_out > 0] = 1
    input_out[input_out < 0] = 0

    softmax_ce = grad_ce(target, prediction)
    return  np.matmul(np.transpose(input), (input_out * np.matmul(softmax_ce, np.transpose(out_weight))))

def hidden_bias(target, prediction, input_out, out_weight):
    input_out[input_out > 0] = 1
    input_out[input_out < 0] = 0

    ones = np.ones((1, input_out.shape[0]))
    softmax_ce = grad_ce(target, prediction)
    return  np.matmul(ones, (input_out * np.matmul(softmax_ce, np.transpose(out_weight))))
```

Let output layer be $\underline{o} = W_o \underline{g} + b_o$ & $\frac{\partial L_{CE}}{\partial \underline{o}} = f - y$

① $\frac{\partial L}{\partial W_o} = \frac{\partial \underline{o}}{\partial W_o} \cdot \frac{\partial L}{\partial \underline{o}}$

$\frac{\partial L}{\partial \underline{o}} = f - y$

$\frac{\partial \underline{o}}{\partial W_o} = g^T$

$$\boxed{\frac{\partial L}{\partial W_o} = g^T (f - y)}$$

② $\frac{\partial L}{\partial b_o} = \frac{\partial \underline{o}}{\partial b_o} \cdot \frac{\partial L}{\partial \underline{o}}$

$\frac{\partial L}{\partial \underline{o}} = f - y$

$\frac{\partial \underline{o}}{\partial b_o} = \mathbb{1}^T$

$$\boxed{\frac{\partial L}{\partial b_o} = \mathbb{1}^T (f - y)}$$

Let hidden layer be $\underline{h} = W_h \underline{x} + b_h$ & $g_i = \mathrm{Relu}(h_i) = \max(0, h_i)$

$$\frac{\partial L}{\partial h_i} = \frac{\partial L}{\partial g_i} \cdot \frac{\partial g_i}{\partial h_i} = \begin{cases} \frac{\partial L}{\partial g_i} & \text{if } h_i > 0 \\ 0 & \text{if } h_i < 0 \end{cases}$$

$$\frac{\partial L}{\partial \underline{g}} = \frac{\partial L}{\partial \underline{o}} \cdot \frac{\partial \underline{o}}{\partial \underline{g}} = (f - y) \underline{W_o}^T$$

③ $\frac{\partial L}{\partial \underline{W_h}} = \frac{\partial \underline{h}}{\partial \underline{W_h}} \cdot \frac{\partial L}{\partial \underline{h}} = \underline{x}^T \cdot \frac{\partial L}{\partial \underline{h}}$

$$\boxed{\frac{\partial L}{\partial \underline{W_h}} = \underline{x}^T \otimes (f - y) \underline{W_o}^T}$$

④ $\frac{\partial L}{\partial \underline{b_h}} = \frac{\partial \underline{h}}{\partial \underline{b_h}} \cdot \frac{\partial L}{\partial \underline{h}} = \mathbb{1}^T \cdot \frac{\partial L}{\partial \underline{h}}$

$$\boxed{\frac{\partial L}{\partial \underline{b_h}} = \underline{\mathbb{1}}^T \otimes (f - y) W_o^T}$$

3. Learning

```
import time
trainData, validData, testData, trainTarget, validTarget, testTarget = load_data()
trainData = trainData.reshape((trainData.shape[0], -1))
validData = validData.reshape((validData.shape[0], -1))
testData = testData.reshape((testData.shape[0], -1))

train_target, valid_target, test_target = convert_onehot(trainTarget, validTarget, testTarget)

H = 1000
epochs = 200
gamma = 0.99
learning_rate = 0.00001

w_o = Xavier(H,10)
w_h = Xavier(784,H)
v_o = np.full((H, 10), 1e-5)
v_h = np.full((trainData.shape[1],H), 1e-5)
b_o = np.zeros((1, 10))
b_h = np.zeros((1, H))

start = time.time()

weight_o, bias_o, weight_h, bias_h, train_acc, valid_acc, test_acc, \
train_loss, valid_loss, test_loss = learning(trainData, train_target, \
validData, valid_target, testData, test_target, w_o, v_o, w_h, v_h, b_o, \
b_h, epochs, gamma, learning_rate)

end = time.time()


# plotting
iterations = range(len(train_acc))
plt.figure()
plt.plot(iterations, train_acc)
plt.plot(iterations, valid_acc)
plt.plot(iterations, test_acc)
plt.legend(['train accuracy', 'valid accuracy', 'test accuracy'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title("Accuracies")


plt.figure()
plt.plot(iterations, train_loss)
plt.plot(iterations, valid_loss)
plt.plot(iterations, test_loss)
plt.legend(['train loss', 'valid loss', 'test loss'])
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title("Losses")
```

```python
def forward(data,wh,wo,bh,bo):
    h = compute_layer(data,wh,bh)
    g = relu(h)
    o = compute_layer(g,wo,bo)
    return softmax(o)

def learning(trainData, trainTarget, validData, validTarget, testData, testTarget, w_o, v_o, w_h, v_h, b_o, b_h, epochs, gamma, learning_rate):
    w_v_o = v_o
    b_v_o = b_o
    w_v_h = v_h
    b_v_h = b_h

    train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc  = [], [], [], [], [], []
    for i in range(epochs):

        # print("Iteration:", i)
        hidden_input_train = compute_layer(trainData, w_h, b_h)
        hidden_output_train = relu(hidden_input_train)
        prediction = softmax(compute_layer(hidden_output_train, w_o , b_o))
        train_loss.append(average_ce(trainTarget,prediction))
        train_acc.append(np.sum(prediction.argmax(axis = 1) == trainTarget.argmax(axis = 1))/trainData.shape[0])


        hidden_input_valid = compute_layer(validData, w_h, b_h)
        hidden_output_valid = relu(hidden_input_valid)
        prediction_valid = softmax(compute_layer(hidden_output_valid, w_o, b_o))
        valid_loss.append(average_ce(validTarget, prediction_valid))
        valid_acc.append(np.sum(prediction_valid.argmax(axis = 1) == validTarget.argmax(axis = 1))/validData.shape[0])


        hidden_input_test = compute_layer(testData, w_h, b_h)
        hidden_output_test = relu(hidden_input_test)
        prediction_test = softmax(compute_layer(hidden_output_test, w_o, b_o))
        test_loss.append(average_ce(testTarget, prediction_test))
        test_acc.append((np.sum(prediction_test.argmax(axis = 1) == testTarget.argmax(axis = 1)))/(testData.shape[0]))

        # updating parameters
        w_v_o = gamma * w_v_o + learning_rate * output_weight(trainTarget, prediction, hidden_output_train)
        b_v_o = gamma * b_v_o + learning_rate * output_bias(trainTarget, prediction)
        w_v_h = gamma * w_v_h + learning_rate * hidden_weight(trainTarget, prediction, trainData, hidden_input_train, w_o)
        b_v_h = gamma * b_v_h + learning_rate * hidden_bias(trainTarget, prediction, hidden_input_train, w_o)

        w_o = w_o - w_v_o
        b_o = b_o - b_v_o
        w_h = w_h - w_v_h
        b_h = b_h - b_v_h

    return w_o, b_o, w_h, b_h, train_acc, valid_acc, test_acc, train_loss, valid_loss, test_loss

def Xavier(unitsin, unitsout):
  return np.random.normal(0, np.sqrt(2/(unitsin + unitsout)), (unitsin,unitsout))
```

I initialized the weight matrices using the Xaiver initialization scheme with zero-mean and 2/(units_in + units_out) variance using the 'Xavier' method. I am training my model over 200 epochs with a hidden unit H of 1000. I tuned the learning rate to 0.00001 and gamma of 0.99. I kept track of the time before and after training to determine the training duration.

During each training iteration, I used the function np.argmax() for the accuracy matrix to find the accuracies. In addition, I calculated the losses with the helper functions. I also updated the parameters.

The resulting losses and accuracies are listed in the table below:

Training Loss	0.006936827205918513
Validation Loss	0.724208224482581
Testing Loss	0.7718570778316731
Training Accuracy	0.9985
Validation Accuracy	0.9121666666666667
Testing Accuracy	0.9093245227606461

Training Time is **395.0429346561432** seconds.