Deep learning Assignment 1

Tianzheng Hu (276027)

October 2023

1 Answers

question 1 Work out the local derivatives of the softmax activation and the cross-entropy loss in scalar terms. Assume that the target class is given as an integer value.

$$y_i = \frac{\exp O_i}{\sum_j \exp O_j}$$

$$l = -\log(y_c)$$

Because of **the Quotient rule**, the derivation of the softmax activation can formalized as:

$$\frac{\partial y_i}{\partial o_j} = \frac{(\frac{\partial}{\partial o_j}\exp O_i) * \sum_j \exp O_j - \exp O_i * (\frac{\partial}{\partial o_j}\sum_j \exp O_j)}{(\sum_j \exp O_j)^2}$$

if i = j, the equation can shortened to:

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial y_i}{\partial o_i} = \frac{\exp O_i * \sum_j \exp O_j - \exp O_i * \exp O_j}{(\sum_j \exp O_j)^2}$$

$$\frac{\partial y_i}{\partial o_j} = \frac{\exp O_i}{\sum_j \exp O_j} * (1 - \frac{\exp O_j}{\sum_j \exp O_j}) = y_i(1 - y_j)$$

if i $\neq$ j, the equation can shortened to:

$$\frac{\partial y_i}{\partial o_j} = \frac{0 * \sum_j \exp O_j - \exp O_i * \exp O_j}{(\sum_j \exp O_j)^2}$$

$$\frac{\partial y_i}{\partial o_j} = \frac{-\exp O_i * \exp O_j}{\sum_j \exp O_j * \sum_j \exp O_j} = -y_i * y_j$$

Because only condition c = i is established, loss will have meaningful explaination, otherwise loss will be 0. The derivation of the cross-entropy can formalized as:

$$\frac{\partial l}{\partial y_i} = \frac{\partial}{\partial y_i}(-\log(y_c)) = -\frac{1}{y_c}$$

question 2 Since we already have $\frac{\partial y_i}{\partial o_i}$ and $\frac{\partial l}{\partial y_i}$, there is not strictly necessary to work out the derivative $\frac{\partial l}{\partial o_i}$ for a neural network. Because we can already obtain the result directly by **the chain derivation rule**.

$$\frac{\partial l}{\partial o_i} = \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial o_i}$$

if i=j:

$$\frac{\partial l}{\partial o_i} = -\frac{1}{y_i} * y_i * (1 - y_i) = y_i - 1$$

if i $\neq$ j:

$$\frac{\partial l}{\partial o_i} = -\frac{1}{y_i} * (-y_i) * y_j = y_j$$

question 3 The structure of this network is two linear layers, the first layer is activated by a sigmoid function and the output of the second layer is activated by softmax(see Fig. 1). In this report, the code in some of the code blocks has been omitted to save space and only the most important steps have been kept. The complete code is available in the appendix.

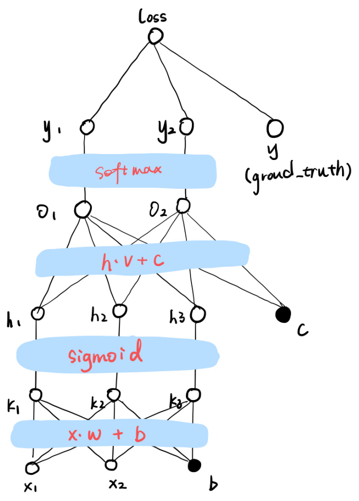


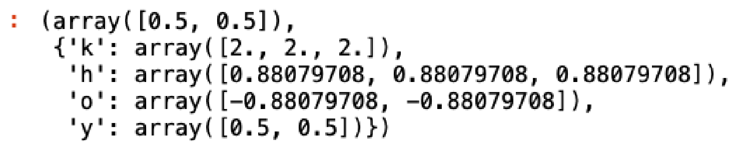
Figure 1: Two layers NN

```
# FUNCTION: forward_propagation
def forward_propagation(X, parameters):
    ...
    # Retrieve each parameter from the dictionary "parameters"
    ...
    # Implement Forward Propagation to calculate o (probabilities)
    k = np.array([sum(W[i][j] * X[i] + b[j] for i in range(2)) for j in range(3)])
    h = np.array([sigmoid(k[i]) for i in range(3)])
    o = np.array([sum(V[i][j] * h[i] + c[j] for i in range(3)) for j in range(2)])
    y = np.array(softmax(o))
    cache = {"k": k,
             "h": h,
             "o": o,
             "y": y}
    return y, cache

# FUNCTION: backward_propagation
def backward_propagation(parameters, cache, X, Y): #, update
    ...
    # initilize all the gradient
    ...
    # First, retrieve W and V from the dictionary "parameters".
    # Retrieve also k, h, o and y from dictionary "cache".
    ...
    l = -math.log(y[indices])
    # Backward propagation: calculate dw, db, dv, dc.
    for i in range(2):
        grad_y = np.where(np.arange(2) == indices, -1 / y[indices], 0)
    for j in range(2):
        grad_o = np.where(np.arange(2) == indices, y - 1, y)
    for j in range(2):
        grad_v[:, j] = grad_o[j] * h
        grad_h += grad_o[j] * V[:, j]
        grad_c[j] = grad_o[j]
    grad_k = grad_h * h * (1 - h)
    for j in range(3):
        grad_w[:, j] = np.dot(grad_k[j], X)
    grad_b = grad_k
    grads_and_loss = {"grad_w": grad_w,
             "grad_b": grad_b,
             "grad_v": grad_v,
             "grad_c": grad_c,
             "loss": l}
    return grads_and_loss
```

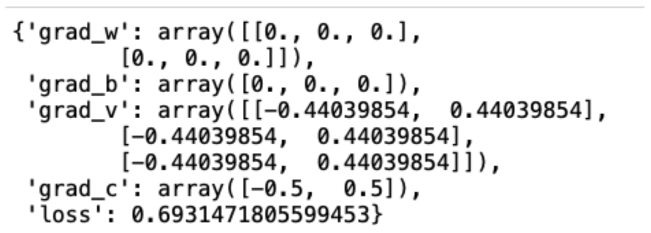
The output of the forward function includes k, h, o, y. After calculating all the derivatives.

question 4 Implement a training loop for your network and show that the training loss drops as training progresses. After 5 loops, the loss dropped



```
: (array([0.5, 0.5]),
   {'k': array([2., 2., 2.]),
    'h': array([0.88079708, 0.88079708, 0.88079708]),
    'o': array([-0.88079708, -0.88079708]),
    'y': array([0.5, 0.5])})
```

Figure 2: forward



```
{'grad_w': array([[0., 0., 0.],
        [0., 0., 0.]]),
 'grad_b': array([0., 0., 0.]),
 'grad_v': array([[-0.44039854,  0.44039854],
        [-0.44039854,  0.44039854],
        [-0.44039854,  0.44039854]]),
 'grad_c': array([-0.5,  0.5]),
 'loss': 0.6931471805599453}
```

Figure 3: backward derivatives and loss

down(see Fig. 4).

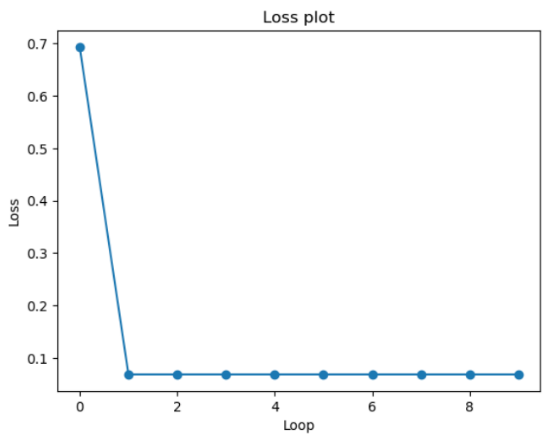


Figure 4: loss plot

question 5 and 6 It's allowed to implement question 5 and 6 together based on canvas page: `https://canvas.vu.nl/courses/72683/discussion_topics/701673`.

Implement a neural network for the MNIST data. Two linear layers have been used as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10. The Xavier's initialization has been used on weights and bias of both layers.

The for-loop used in scalar calculation can be simplified as matrix multiple in forward and backward propagation.

```
def forward(X, parameters):
    ...
     # Retrieve each parameter from the dictionary "parameters"
     ...
     # Implement Forward Propagation to calculate probabilities
     # Perform matrix multiplication and addition
     k = np.dot(W.T, X.reshape(-1, 1)) + b
     h = np.array([sigmoid(k_) for k_ in k])
     o = np.dot(V.T, h) + c
     y = softmax(o)
     cache = {"k": k,
              "h": h,
              "o": o,
              "y": y}
return y, cache

def backward(parameters, cache, X, Y_true, learning_rate):
     ...
     # First, retrieve W, b, V and c from the dictionary "parameters".
     # Retrieve also k, h, o and y from dictionary "cache".
     ...
     loss, grad_loss = compute_loss(y, Y_true)
     # Backward propagation: calculate dw, db, dv, dc.
     grad_o = grad_loss
     grad_v = np.matmul(h, grad_o.T)
     grad_h = np.dot(V, grad_o)
     grad_c = grad_o.copy()
     grad_k = np.dot(np.dot(grad_h, h.T), (1 - h))
     grad_w = np.matmul(X.reshape(-1,1), grad_k.reshape(1, -1))
     grad_b = grad_k
     # update weight parameters
     new_w = W - learning_rate * grad_w
     new_v = V - learning_rate * grad_v
     new_b = b - learning_rate * grad_b
     new_c = c - learning_rate * grad_c
...
return grads, new_parameters, loss
```

Calculate the gradient of the loss using one-hot coding.

```
def compute_loss(y, Y_true):
    ...
     # Compute the cross-entropy cost
```

```
        loss = - math.log(y[Y_true])
        y_onehot = np.zeros_like(y)
        y_onehot[Y_true] = 1
        grad_loss = y - y_onehot
    return loss, grad_loss
```

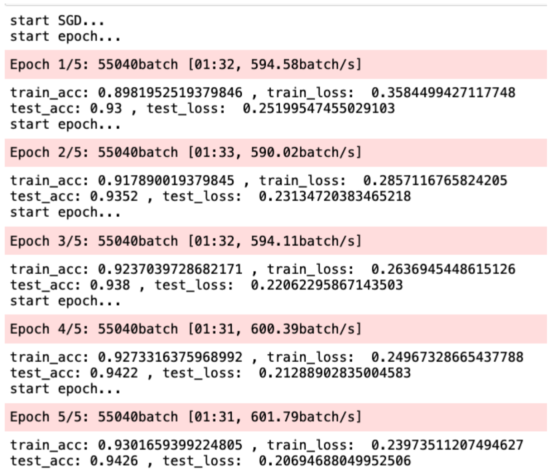
Define the training function to perform forward and backward propagation. In the test function, only forward propagation is performed to obtain the predicted y-value without backward gradient update.

```
    def train(parameters, learning_rate, xtrain_norm, ytrain):
        ...
        for i in np.arange(xtrain_norm.shape[0]):
            ...
            y, cache = forward(X, parameters)
            ...
            grads, new_parameters, loss = backward(parameters, cache,
            X, Y_true, learning_rate)
            ...
            parameters = new_parameters
    return parameters, train_loss, y_train_pred
    def test(parameters, xtest_norm, ytest):
        ...
        for i in np.arange(xtest_norm.shape[0]):
            ...
            y, cache = forward(X, parameters)
            y_pred = np.argmax(y)
            ...
    return test_loss, y_test_pred
```

Define an SGD function for experimentation, use shuffling in the epoch loop to split out the batches of the training dataset, and test it at the end of each epoch to obtain the accuracy and loss of the present model.

```
    def SGD(learning_rate, epoch_size, batch_size, output_num, X_train, Y_train, X_test, Y_t
        # Initialization
        ...
        parameters = initialize_paras(hidden_size, output_size)
        ...
        # Number of training examples
        # Implement batch
        num_examples = len(X_train)
        # Randomly shuffle the data (if needed)
        indices = np.random.permutation(num_examples)
        X = X_train[indices]
        Y = Y_train[indices]
        for epoch in range(epoch_size):
            # Training loop with batch processing
            # Reset progress bar for each epoch
            with tqdm(total=num_examples, desc=f'Epoch {epoch + 1}/{epoch_size}', unit='batc
                for i in range(0, num_examples, batch_size):
                    # Extract a batch of data and labels
                    x_train = X[i:i + batch_size]
                    y_train = Y[i:i + batch_size]
                    parameters, train_loss, y_train_pred = train(parameters, learning_rate,
                    # Calculate accurancy
                    ...
                    pbar.update(batch_size)
            test_loss, y_test_pred = test(parameters, X_test, Y_test)
            # Calculate epoch_train_loss, epoch_test_loss, train_acc, test_acc
        ...
    return All loss_and_acc_records
```

Running the SGD function when learning_rate is 0.01, epoch_size is 5 and batch_size is 64, the training proceeds normally in this situation(see Fig. 5).



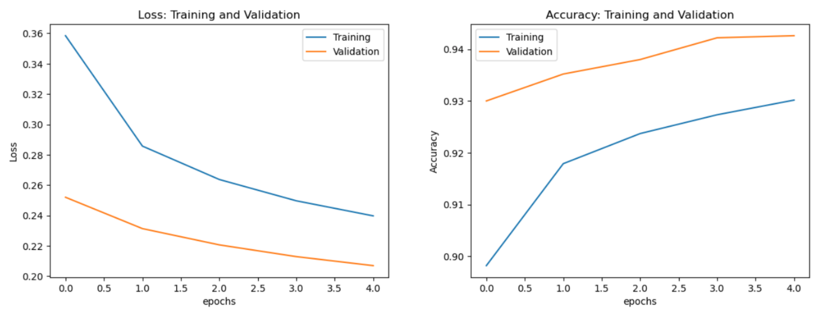
```
start SGD...
start epoch...
Epoch 1/5: 55040batch [01:32, 594.58batch/s]
train_acc: 0.8981952519379846 , train_loss:  0.3584499427117748
test_acc: 0.93 , test_loss:  0.25199547455029103
start epoch...
Epoch 2/5: 55040batch [01:33, 590.02batch/s]
train_acc: 0.917890019379845 , train_loss:  0.2857116765824205
test_acc: 0.9352 , test_loss:  0.23134720383465218
start epoch...
Epoch 3/5: 55040batch [01:32, 594.11batch/s]
train_acc: 0.9237039728682171 , train_loss:  0.2636945448615126
test_acc: 0.938 , test_loss:  0.22062295867143503
start epoch...
Epoch 4/5: 55040batch [01:31, 600.39batch/s]
train_acc: 0.9273316375968992 , train_loss:  0.24967328665437788
test_acc: 0.9422 , test_loss:  0.21288902835004583
start epoch...
Epoch 5/5: 55040batch [01:31, 601.79batch/s]
train_acc: 0.9301659399224805 , train_loss:  0.23973511207494627
test_acc: 0.9426 , test_loss:  0.20694688049952506
```

Figure 5: SGD is running

question 7 Analysis

1. Compare the training loss per epoch to the validation loss per epoch. It is interesting to note that the accuracy performance on the validation dataset is higher than on the training dataset, and the test loss is lower than the training loss. A likely reason for this is that the model is overfitting, which usually happens when the test and training data are very similar.

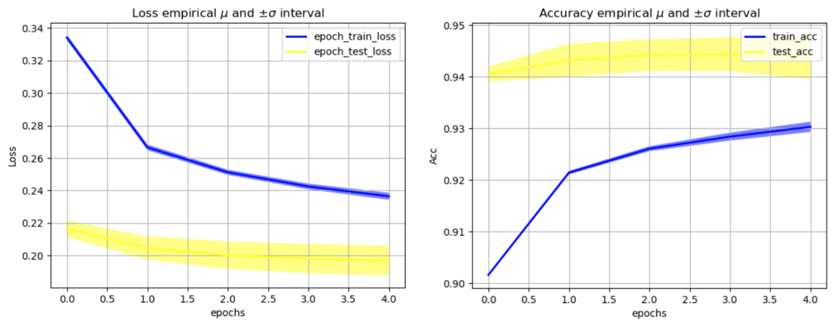
2. After four random initializations of the weights and training of five epochs, the loss for each epoch was calculated, acc the mean and variance of



(a) (b)

Figure 6: (a) epoch loss (b) epoch accuracy

the two values. Both training loss and training accuracy show relatively stable performance, while testing loss and testing accuracy have larger variance. This may be because the test dataset has data that the model has not seen before, so the predictions are unstable. Whereas the data on the training dataset is exactly where the model has been trained, so good predictions can be achieved for the data.

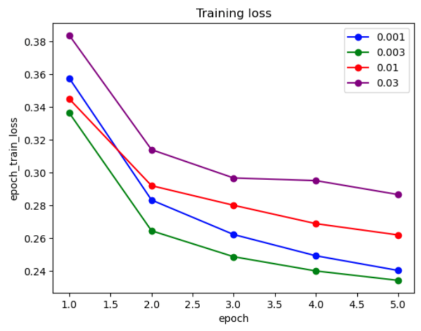


(a) (b)

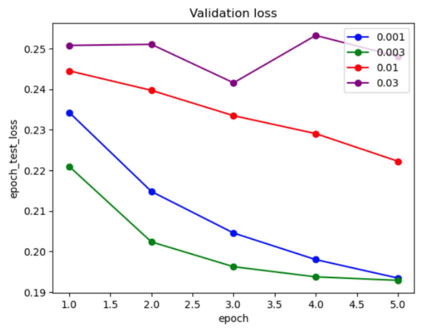
Figure 7: (a) epoch loss (b) epoch accuracy

3. Training the model with four different learning rates. It can be seen that the model performs best at a learning rate of 0.03, where the training loss drops below 0.24 and the accuracy is close to 0.95. 0.001 is not the best instead. The worst performance is seen at a learning rate of 0.03, which is possible due to the large step size, causing the gradient to cross the optimum point(see Fig. 8 and Fig. 9). This leads to a much slower decline in losses and less accuracy.

4. Final hyper hyperparameters setting is batch_size is 64, learning_rate is 0.003 and so on. The result of train_loss, test_loss, train_accuracy and test_accuracy are 0.93, 0.95, 0.24, 0.19.

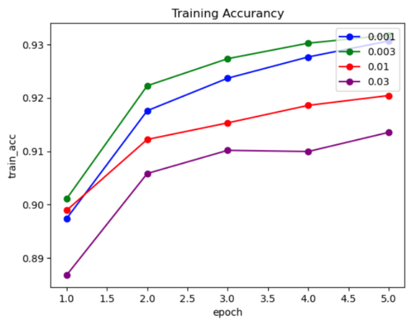


(a)

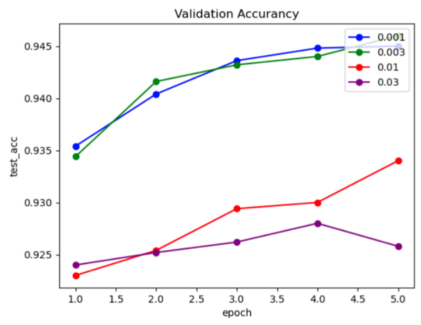


(b)

Figure 8: (a) Training loss (b) Validation loss



(a)



(b)

Figure 9: (a) Training accuracy (b) Validation accuracy

A Appendix

Part. 2

```
import numpy as np
from copy import deepcopy
import matplotlib.pyplot as plt
from copy import deepcopy
import math

# FUNCTION: initialize_parameters

def initialize_parameters():
    """
     Initialise weights and biases

    Returns:
        params -- python dictionary containing your parameters:
                        W -- weight matrix of shape (2, 3)
                        b -- bias vector of shape (3, 1)
                        V -- weight matrix of shape (3, 2)
                        c -- bias vector of shape (2, 1)
    """
    W = [
        [1., 1., 1.],
        [-1., -1., -1.]
        ]
    V = [
        [1, 1],
        [-1, -1],
        [-1, -1]
    ]
    b, c = [0, 0, 0], [0, 0]


    parameters = {"W": np.array(W),
                  "b": np.array(b),
                  "V": np.array(V),
                  "c": np.array(c)}

    return parameters

# FUNCTION: sigmoid
def sigmoid(k):
    """
    Compute the sigmoid of k

    Arguments:
    k -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(k)
    """
    s = 1/(1+(np.exp(-k)))
    return s

# FUNCTION: softmax
def softmax(h):
    """
    Compute the softmax of h

    Arguments:
    h -- A scalar or numpy array of any size.
```

```python
    Return:
    softmax_probs -- softmax(h)
    """
    # Ensure numerical stability by subtracting the maximum logit
    max_h = np.max(h)
    exp_hs = np.exp(h - max_h)

    # Calculate softmax probabilities
    softmax_probs = exp_hs / np.sum(exp_hs, axis=0)

    return softmax_probs

# FUNCTION: forward_propagation
def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing parameters
    (output of initialization function)

    Returns:
    h -- The sigmoid output of the second activation
    cache -- a dictionary containing "k", "h", "o", "y"
    """
    # Retrieve each parameter from the dictionary "parameters"
    W = parameters["W"]
    b = parameters["b"]
    V = parameters["V"]
    c = parameters["c"]

    # Implement Forward Propagation to calculate o (probabilities)
    k = np.array([sum(W[i][j] * X[i] + b[j] for i in range(2))
    for j in range(3)])
    h = np.array([sigmoid(k[i]) for i in range(3)])
    o = np.array([sum(V[i][j] * h[i] + c[j] for i in range(3))
    for j in range(2)])
    y = np.array(softmax(o))

    cache = {"k": k,
             "h": h,
             "o": o,
             "y": y}

    return y, cache
```

```
# FUNCTION: backward_propagation

def backward_propagation(parameters, cache, X, Y): #, update
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "k", "h", "o", "y".
    X -- input data of shape (2, 1)
    Y -- "true" labels vector of shape (2, 1)

    Returns:
    grads -- python dictionary containing your gradients with
    respect to different parameters
    """
    # initilize all the gradient
    m = np.array(X).shape[0]
    grad_b = np.zeros(m)
    grad_w = np.zeros((m, 3))
    grad_k = np.zeros(3)
    grad_h = np.zeros(3)
    grad_c = np.zeros(2)
    grad_v = np.zeros((3, 2))
    grad_o = np.zeros(2)
    grad_y = np.zeros(2)

    indices = [i for i in range(len(X)) if X[i] == Y[i]][0]


    # First, retrieve W and V from the dictionary "parameters".
    W = parameters["W"]
    V = parameters["V"]

    # Retrieve also k, h, o and y from dictionary "cache".
    k = cache["k"]
    h = cache["h"]
    o = cache["o"]
    y = cache["y"]

    l = -math.log(y[indices])
    # Backward propagation: calculate dw, db, dv, dc.
    for i in range(2):
        grad_y = np.where(np.arange(2) == indices, -1 / y[indices], 0)
    for j in range(2):
        grad_o = np.where(np.arange(2) == indices, y - 1, y)
    for j in range(2):
        grad_v[:, j] = grad_o[j] * h
        grad_h += grad_o[j] * V[:, j]
        grad_c[j] = grad_o[j]
    grad_k = grad_h * h * (1 - h)

    for j in range(3):
        grad_w[:, j] = np.dot(grad_k[j], X)
    grad_b = grad_k

    grads_and_loss = {"grad_w": grad_w,
             "grad_b": grad_b,
             "grad_v": grad_v,
             "grad_c": grad_c,
             "loss": l}

    return grads_and_loss

def update(parameters, grads, learning_rate):
    """
    Implement the update grads with learning rate using the
    instructions above.

    Arguments:
    parameters -- a dictionary containing all parameters
    grads -- a dictionary containing "k", "h", "o", "y".
    learning_rate -- the steps for updating weights

    Returns:
    new_parameters -- python dictionary containing new parameters
    after updating
    """

    # First, retrieve W and V from the dictionary "parameters".
    w = parameters["W"]
    b = parameters["b"]
    v = parameters["V"]
    c = parameters["c"]

    # First, retrieve W and V from the dictionary "grads".
    grad_w = grads["grad_w"]
    grad_b = grads["grad_b"]
    grad_v = grads["grad_v"]
    grad_c = grads["grad_c"]
    for j in range(3):
        for i in range(2):
            w[i][j] = w[i][j] - learning_rate* grad_w[i][j]
        b[j] = b[j] - learning_rate * grad_b[j]
    for j in range(2):
        for i in range(3):
            v[i][j] = v[i][j] - learning_rate* grad_v[i][j]
        c[j] = c[j] - learning_rate * grad_c[j]


    new_parameters = {"W": w,
                      "b": b,
                      "V": v,
                      "c": c}

    return new_parameters
```

Part. 3

```
import numpy as np
from time import time
from IPython.display import clear_output
from data import load_mnist
import math
from copy import deepcopy
import matplotlib.pyplot as plt
from tqdm import tqdm

def sigmoid(k):
    """
    Compute the sigmoid of k

    Arguments:
    k -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(k)
    """
    s = 1/(1+(np.exp(-k)))
return s

def softmax(h):
    """
    Compute the softmax of h
    Arguments:
    h -- A scalar or numpy array of any size.

    Return:
    softmax_probs -- softmax(h)
    """
    # Ensure numerical stability by subtracting the maximum logit
    max_h = np.max(h)
    exp_hs = np.exp(h - max_h)

    # Calculate softmax probabilities
    softmax_probs = exp_hs / np.sum(exp_hs, axis=0)

return softmax_probs

def initialize_paras(hidden_size, output_size):
    """
     Initialise weights and biases

    Returns:
        params -- python dictionary containing your parameters:
                    W -- weight matrix of shape (hidden_size, input_size)
                    b -- bias vector of shape (input_size, 1)
                    V -- weight matrix of shape (hidden_size, output_size)
                    c -- bias vector of shape (output_size, 1)
    """
    W = np.random.normal(0, np.sqrt(2.0 / (xtrain_norm[0].shape[0]+
    hidden_size)),
    size=(xtrain_norm[0].shape[0], hidden_size))
    V = np.random.normal(0, np.sqrt(2.0 / (hidden_size+ output_size)),
    size=(hidden_size, output_size))

    #     b, c = [0, 0, 0], [0, 0]
    b = np.random.rand(hidden_size, 1)
    c = np.random.rand(output_size, 1)

    parameters = {"W": W,
                  "b": b,
                  "V": V,
                  "c": c}

    return parameters

def forward(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing parameters
    (output of initialization function)

    Returns:
    h -- The sigmoid output of the second activation
    cache -- a dictionary containing "k", "h", "o", "y"
    """
    # Retrieve each parameter from the dictionary "parameters"
    W = parameters["W"]
    b = parameters["b"]
    V = parameters["V"]
    c = parameters["c"]

    # Implement Forward Propagation to calculate A2 (probabilities)
    # Perform matrix multiplication and addition
    k = np.dot(W.T, X.reshape(-1, 1)) + b
    h = np.array([sigmoid(k_) for k_ in k])
    o = np.dot(V.T, h) + c
    y = softmax(o)

    cache = {"k": k,
             "h": h,
             "o": o,
             "y": y}

    return y, cache

def compute_loss(y, Y_true):
    """
    Computes the cross-entropy cost

    Arguments:
    y -- The sigmoid output of the second activation, of shape (1, 10)
    Y -- "true" labels

    Returns:
    loss -- cross-entropy cost
    """
    # Compute the cross-entropy cost
    loss = - math.log(y[Y_true])
    y_onehot = np.zeros_like(y)
    y_onehot[Y_true] = 1
    grad_loss = y - y_onehot
    return loss, grad_loss

def backward(parameters, cache, X, Y_true, learning_rate):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "k", "h", "o", "y".
    X -- input data of shape (2, 1)
    Y -- "true" labels vector of shape (2, 1)

    Returns:
    grads -- python dictionary containing your gradients with respect
    to different parameters
    """
    # First, retrieve W, b, V and c from the dictionary "parameters".
    W = parameters["W"]
    b = parameters["b"]
    V = parameters["V"]
    c = parameters["c"]

    # Retrieve also k, h, o and y from dictionary "cache".
    k = cache["k"]
    h = cache["h"]
    o = cache["o"]
    y = cache["y"]

    loss, grad_loss = compute_loss(y, Y_true)

    # Backward propagation: calculate dw, db, dv, dc.
    grad_o = grad_loss

    grad_v = np.matmul(h, grad_o.T)
    grad_h = np.dot(V, grad_o)
    grad_c = grad_o.copy()
    grad_k = np.dot(np.dot(grad_h, h.T), (1 - h))

    grad_w = np.matmul(X.reshape(-1,1), grad_k.reshape(1, -1))
    grad_b = grad_k

    # update weight parameters
    new_w = W - learning_rate * grad_w
    new_v = V - learning_rate * grad_v
    new_b = b - learning_rate * grad_b
    new_c = c - learning_rate * grad_c
    grads = {"grad_w": grad_w,
             "grad_b": grad_b,
             "grad_v": grad_v,
             "grad_c": grad_c,
             }
    new_parameters = {
        "W": new_w,
        "b": new_b,
        "V": new_v,
        "c": new_c,
    }
    loss = {"loss": loss}
    return grads, new_parameters, loss

def train(parameters, learning_rate, xtrain_norm, ytrain):
    train_loss = []
    y_train_pred = []

    for i in np.arange(xtrain_norm.shape[0]):
    # for i in np.arange(5000):
        X = xtrain_norm[i]
        Y_true = ytrain[i]
        # x = np.append(x, 1)

        y, cache = forward(X, parameters)
        y_pred = np.argmax(y)
        y_train_pred.append(y_pred)

        grads, new_parameters, loss = backward(parameters, cache,
        X, Y_true, learning_rate)
        train_loss.append(loss["loss"])

        parameters = new_parameters
    #     print(loss)
    return parameters, train_loss, y_train_pred

def test(parameters, xtest_norm, ytest):
    test_loss = []
    y_test_pred = []
    paras = parameters

    for i in np.arange(xtest_norm.shape[0]):
    # for i in np.arange(500):
        X = xtest_norm[i]
        Y_true = ytest[i]
    #     x = np.append(x, 1)

        y, cache = forward(X, paras)
        y_pred = np.argmax(y)
        y_test_pred.append(y_pred)

        loss, grad_loss = compute_loss(y, Y_true)
        test_loss.append(loss)

    return test_loss, y_test_pred


def SGD(learning_rate, epoch_size, batch_size, output_num,
X_train, Y_train, X_test, Y_test):
    # Initialization
    hidden_size = 300
    output_size = output_num
    parameters = initialize_paras(hidden_size, output_size)

    epoch_train_loss = np.zeros((epoch_size))
    epoch_test_loss = np.zeros((epoch_size))
    train_acc = np.zeros((epoch_size))
    test_acc = np.zeros((epoch_size))

    # Example batch_size
    # Number of training examples
    num_examples = len(X_train)
    # Randomly shuffle the data (if needed)
    indices = np.random.permutation(num_examples)
    X = X_train[indices]
    Y = Y_train[indices]
    print("start SGD...")
    for epoch in range(epoch_size):
        # Training loop with batch processing
        print("start epoch...")
        batch_train_loss = []
        batch_test_loss = []
        batch_train_acc = []
        batch_test_acc = []
        # Reset progress bar for each epoch
        with tqdm(total=num_examples, desc=f'Epoch {epoch +
        1}/{epoch_size}', unit='batch') as pbar:

            for i in range(0, num_examples, batch_size):
                # Extract a batch of data and labels
                x_train = X[i:i + batch_size]
                y_train = Y[i:i + batch_size]

                parameters, train_loss, y_train_pred =
                train(parameters, learning_rate, x_train, y_train)

                batch_train_loss.extend([np.sum(train_loss)/len(train_loss)])
                acc = (np.array((y_train_pred) == y_train))
                acc_as_list = acc.astype(int).tolist()
                batch_train_acc.extend([np.array(acc_as_list).mean()])

                pbar.update(batch_size)

        test_loss, y_test_pred = test(parameters, X_test, Y_test)
        batch_test_loss.extend([np.sum(test_loss)/len(test_loss)])
        acc = (np.array((y_test_pred) == Y_test))
        acc_as_list = acc.astype(int).tolist()
        batch_test_acc.extend([np.array(acc_as_list).mean()])


        epoch_train_loss[epoch] = np.array(batch_train_loss).mean()
        train_acc[epoch] = np.array(batch_train_acc).mean()
        epoch_test_loss[epoch] = np.array(batch_test_loss).mean()
        test_acc[epoch] = np.array(batch_test_acc).mean()
        print("train_acc:", np.array(batch_train_acc).mean(),
        ", train_loss: ", np.array(batch_train_loss).mean())
        print("test_acc:", np.array(batch_test_acc).mean(),
        ", test_loss: ", np.array(batch_test_loss).mean())

    records = {
        "epoch_train_loss" : epoch_train_loss,
        "epoch_test_loss" : epoch_test_loss,
        "train_acc" : train_acc,
        "test_acc" : test_acc,
        "batch_train_loss" : batch_train_loss,
        "batch_test_loss" : batch_test_loss,
        "batch_train_acc" : batch_train_acc,
        "batch_test_acc" : batch_test_acc

    }
    return records
```