Uppsala University

Deep learning

Hand-in assignment (1)

Tong You

April 19, 2021

Contents

1 Introduction

In this document we will be highlighting the different steps that go into building a classifier for hand-written digits from scratch. We will be focusing on the MNIST dataset. This is a well-known data set that has 60000 training samples and 10000 testing samples. The samples are digits from 0 until 9. There are therefore 10 classes. We note that the images (28 x 28 pixels) have been flattened to a vector with 784 entries.

Figure 1: 16 examples of each class from the MNIST data set. Image retrieved from https://commons.wikimedia.org/wiki/File:MnistExamples.png

Compared to the previous assignment, this is not a binary classification task anymore and we need a different approach to the problem. To this end, we will be implementing a feedforward neural network to solve a multiclass classification problem. We will be dividing the main problem into three parts. First, a softmax output needs to be added. Second, the regular gradient descent method will be replaced with mini-batch training (SGD). And finally, the full neural network will be implemented with multiple layers. Important intermediate results will be shown in their respective sections. To make development understandable, the Python code will be implemented in 3 separate scripts. Development will be done in a Github repository. All

Python code will also be attached in the Appendix.

2 Softmax regression

In order to do multiclass classification on the MNIST dataset we will need to update the linear regression model with a softmax output. Before we do so, we will need to get our dimensions in order. We will be using row-column notation and our first dimensions will be reserved for the samples. Our training data set has dimensions 600000 x 784 and our test data set has dimensions 10000 x 784. Our training labels have dimensions 60000 x 10 and test data set has dimensions 10000 x 10, where we are using one-hot encoding: the m-th column entry is "1" if it corresponds to a certain digit. Therefore we have the following indices:

$$i = 1, 2, ..., n \,(\text{number of samples}) \tag{1}$$

$$j = 1, 2, ..., p \,(\text{number of pixels}) \tag{2}$$

$$m = 1, 2, ..., M \,(\text{number of classes}) \tag{3}$$

Note that the number of samples can refer to the training and testing data. In the Python code, the distinction is made clearly. Our weight matrix will have dimensions M x p, and the bias term will have dimensions 1 x M. Therefore, our linear model for the log odds will be:

$$z_{im} = \sum_{j=1}^{p} w_{mj} x_{ij} + b_m \tag{4}$$

Since we don't want to loop over array indices we will vectorize the equation as follows:

$$\mathbf{z}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}_m \tag{5}$$

In the Python implementation we will be transposing the weight matrix in order to get the correct dimensions for the subsequent matrix multiplication (see the Appendix for implementations per task). We will be using gradient descent to update our weights and biases. Before we can implement the Python code we need to derive the update equations for gradient descent. Compared to the pre-course assignment, we will be using the cross-entropy as our loss function. The loss:

$$L_i = -\sum_{m=1}^{M} \tilde{y}_{im} \, ln(p_{im}) \tag{6}$$

In the previous equation, $\tilde{y}_{im}$ are the one-hot encoded class labels. This one-hot encoding has already been done during the loading in the provided Loading function. The one-hot encoding is defined as:

$$\tilde{y}_{im} = \begin{cases} 1, & \text{if } y_i = m. \\ 0, & \text{if } y_i \neq m. \end{cases} \tag{7}$$

The p_{im} are the probabilities that the i-th data point belongs to the m-th class (or digit). It is also called the softmax function. It is written as:

$$p_{im} = \frac{e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}} \tag{8}$$

The cost can be determined as:

$$J = \frac{1}{n} \sum_{i=1}^{n} L_i \tag{9}$$

Or, with the loss function filled in:

$$J = -\frac{1}{n} \sum_{i=1}^{n} \sum_{m=1}^{M} \tilde{y}_{im} \, ln(p_{im}) \tag{10}$$

Before we extend these equations and derive the update rules for the weights and biases, we need to remark two things. The naive implementation of the softmax function will lead to numerical overflow/underflow if we have large/small z_{im} respectively. To counteract numerical overflow, we will normalize the z_{im} by subtracting the maximum and to counteract numerical underflow, we will be calculating the softmax function and the loss in one go.

We want the derivatives for the cost function with respect to the b_m and w_{mj}:

$$\frac{dJ}{db_m} = \sum_{i=1}^{n} \frac{dJ}{dz_{im}} \frac{dz_{im}}{db_m} \tag{11}$$

$$\frac{dJ}{dw_{mj}} = \sum_{i=1}^{n} \frac{dJ}{dz_{im}} \frac{dz_{im}}{dw_{mj}} \tag{12}$$

The loss function with the softmax function filled in:

$$L_i = \sum_{m=1}^{M} \Big(\tilde{y}_{im} \, ln(\sum_{l=1}^{M} e^{z_{il}}) - \tilde{y}_{im} \, z_{im} \Big) \tag{13}$$

We also note that $\frac{dJ}{dz_{im}}$ can be written as:

$$\frac{dJ}{dz_{im}} = \frac{1}{n} \frac{dL_i}{dz_{im}} = \frac{1}{n} \frac{dL_i}{dp_{im}} \frac{dp_{im}}{dz_{im}} \tag{14}$$

Before going any further, note that we know what the dimensionality of the derivatives will be simply by inspecting the indices underneath the symbols. This will also be helpful when implementing the softmax gradient descent itself. We will do the simple derivatives first before deriving the more complicated ones.

From equation 4, we can see that:

$$\frac{dz_{im}}{db_m} = 1 \tag{15}$$

Again from equation 4, we also see that:

$$\frac{dz_{im}}{dw_{mj}} = x_{ij} \tag{16}$$

Next, using equation 6 and the derivative of the natural logarithm:

$$\frac{dL_i}{dp_{im}} = -\frac{\tilde{y}_{im}}{p_{im}} \tag{17}$$

The most complicated derivative is the derivative of the softmax function with respect to the z_{im}. To reduce the amount of equations, we will not list all steps of the derivation. Note however that we will be using equation 8 and the quotient rule for derivatives. After some algebra we get:

$$\frac{dp_{im}}{dz_{im}} = \frac{\sum_{l=1}^{M} e^{z_{il}} \, e^{z_{im}} - e^{z_{im}} \, e^{z_{im}}}{(\sum_{l=1}^{M} e^{z_{il}})^2} \tag{18}$$

This is simplified further into:

$$\frac{dp_{im}}{dz_{im}} = \frac{e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}} \frac{\sum_{l=1}^{M} e^{z_{il}} - e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}} \tag{19}$$

And further into:

$$\frac{dp_{im}}{dz_{im}} = p_{im}\left(1 - \frac{e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}}\right) \tag{20}$$

Finally, we get:

$$\frac{dp_{im}}{dz_{im}} = p_{im}\left(1 - p_{im}\right) \tag{21}$$

Now combining equations 14, 17, and 21 the derivative for the cost function becomes:

$$\frac{dJ}{dz_{im}} = -\frac{1}{n}\frac{\tilde{y}_{im}}{p_{im}}p_{im}\left(1 - p_{im}\right) = \frac{\tilde{y}_{im}p_{im} - \tilde{y}_{im}}{n} \tag{22}$$

Using equations 11, 15, 16, and 22 we get:

$$\frac{dJ}{db_m} = \sum_{i=1}^{n}\frac{\tilde{y}_{im}p_{im} - \tilde{y}_{im}}{n} \tag{23}$$

And using equations 12, 15, 16, and 22:

$$\frac{dJ}{dw_{mj}} = \sum_{i=1}^{n}\frac{(\tilde{y}_{im}p_{im} - \tilde{y}_{im})x_{ij}}{n} \tag{24}$$

Now we are ready to give the update equations for the weights and biases. Using $\leftarrow$ to denote a variable update and γ the learning rate, we get the gradient descent update rule for the biases b_m:

$$b_m \leftarrow b_m - \gamma\,\frac{dJ}{db_m} \tag{25}$$

The update rule for weight w_{mj}

$$w_{mj} \leftarrow w_{mj} - \gamma\,\frac{dJ}{dw_{mj}} \tag{26}$$

With the derivatives fully written out:

$$b_m \leftarrow b_m - \frac{\gamma}{n}\,\sum_{i=1}^{n}(\tilde{y}_{im}p_{im} - \tilde{y}_{im}) \tag{27}$$

And for w_{mj}

$$w_{mj} \leftarrow w_{mj} - \frac{\gamma}{n}\,\sum_{i=1}^{n}(\tilde{y}_{im}p_{im} - \tilde{y}_{im})x_{ij} \tag{28}$$

In the next section we will show the results for the softmax classifier on MNIST.

2.1 Results

Previously, we have derived the gradient descent update equations for the weights and biases. This is very important, but how we choose to implement the equations in Python is also important. The full code to generate the plots can be found in Softmax regression. We will highlight the most important part and discuss some brief implementation details before showing the results. In the main optimization loop, the weights and biases are updated as follows:

```
        z_im = xtrain @ w_mj.T + b_m

        y_im = ytrain

        z_im_norm = z_im - np.max(z_im, axis = 1, keepdims = True)

        p_im = np.exp(z_im_norm) / np.sum(np.exp(z_im_norm), axis = 1,
        keepdims = True)

        dJdzim = (1/n_train) * (y_im * p_im - y_im)

        dJdbm = np.sum(dJdzim, axis = 0)
        dJdwmj = dJdzim.T @ xtrain

        b_m = b_m - lr * dJdbm
        w_mj = w_mj - lr * dJdwmj
```

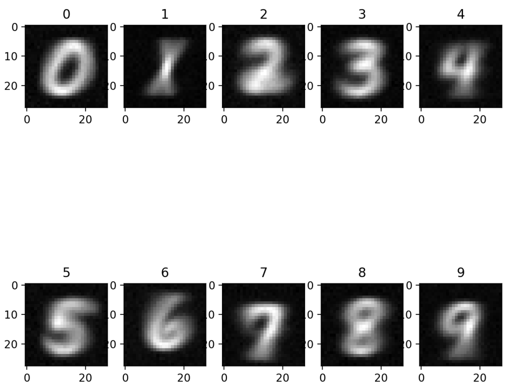
The choice has been made to try to stay as close as possible to the mathematical equations we have used previously. The normalization of the z_{im} is explicit in the Python code. The numerical-stable version of the loss was also implemented:

```
    L_i = np.sum(y_im * np.log(np.sum(np.exp(z_im_norm), axis = 1,
    keepdims = True)) - y_im * z_im_norm, axis = 1)
```

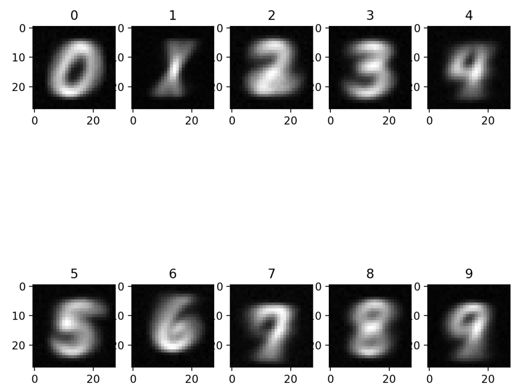
The testing loss is implemented in a similar way, with the only difference being that we use testing data to evaluate (not training data!!!) our model. Before we show the results, we will briefly talk about parameter initialization. For the biases b_m it was sufficient to initialize all 10 values to zero. For the weights w_{mj} we will initialize the entire matrix with normally-distributed values with a mean of 0 and a standard deviation of 0.01.

An important value to tune is the learning rate γ. Too high a value means there will be no convergence, whereas a low value means that the learning will be slow. We will show results for $\gamma = 0.01$, $\gamma = 0.02$, and $\gamma = 0.03$. These learning rates resulted in accuracies close to 90%. We ran the training for a maximum of 3000 iterations. In Figures 2 and 3 the weights and accuracy/cost plots are shown for the three different learning rates. From the weight matrices themselves no clear distinction can be made between the choice of the three different learning rates. It is only when we look at the cost/accuracy plots that a clear distinction can be made between the choice of the various learning rates. First of all, we note that for all the learning rates the training accuracy is always slightly below the testing accuracy. In the cost plots, this is reverse where the training cost is higher than the testing cost. This is consistent with each other. We also see that the testing accuracy does increase going from a learning rate of 0.01 to 0.03. However, from testing (results not shown) it was difficult to reach a testing accuracy of 90%. Finally, we note that the cost increases after arriving at a minimum value for both the training and testing data when learning rates of 0.02 and 0.03 are used. Note that we don't train on the testing data (can also be seen in the code). We only evaluate the testing data using the weights and bias that is being trained on the training data. Since we have more training than testing data it is also no unusual for the testing accuracy to be slightly higher than the training accuracy.

(a) $\gamma = 0.01$



(b) $\gamma = 0.02$



(c) $\gamma = 0.03$

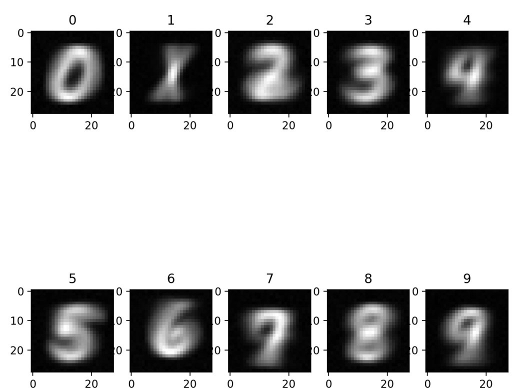
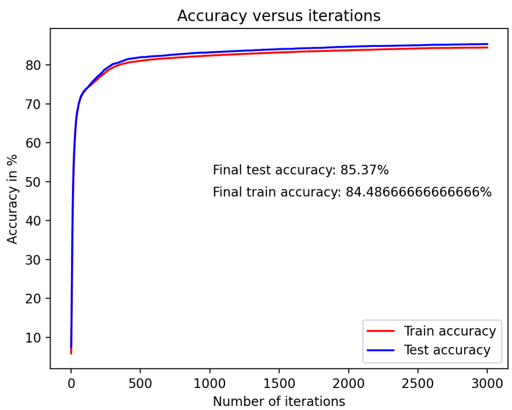
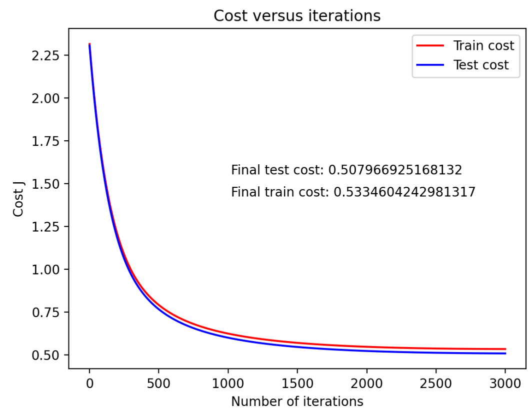


Figure 2: All 10 weight matrices for three different learning rates. The learning rates are indicated in the subcaptions. No post-processing was performed on the weights obtained. The weight matrices were only reshaped from 784 to 28 x 28 images.

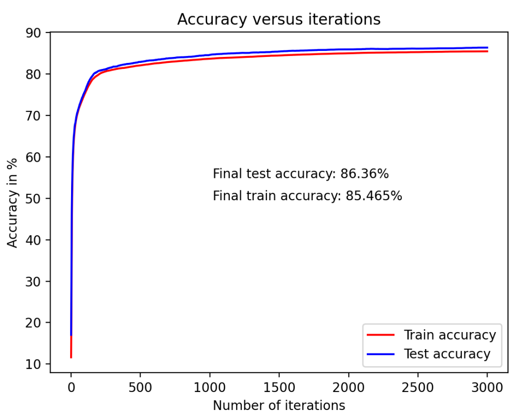
(a) $\gamma = 0.01$



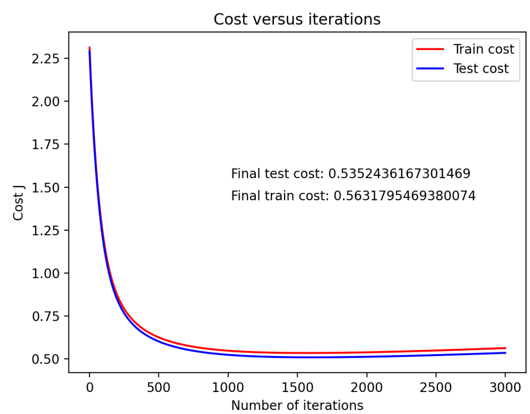
(b) $\gamma = 0.01$



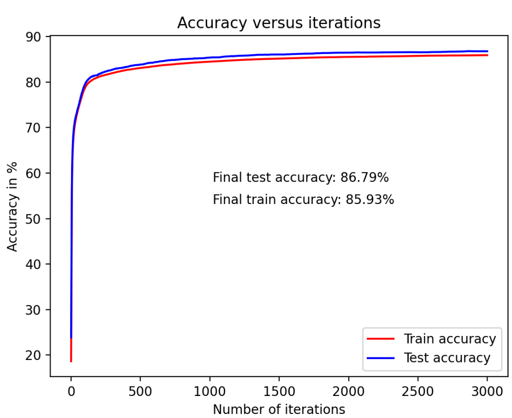
(c) $\gamma = 0.02$



(d) $\gamma = 0.02$



(e) $\gamma = 0.03$



(f) $\gamma = 0.03$

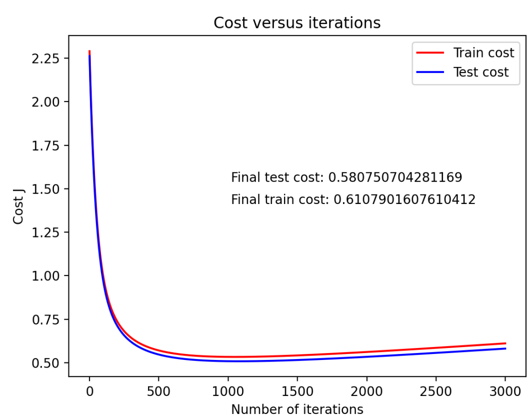


Figure 3: Accuracy and cost for the corresponding weights in Figure 2.

3 Softmax regression with mini-batch SGD

There are no major differences compared to the softmax-only implementation. We will use the same update equations for the weights and biases. There will be minor changes to the equations and Python code used previously. The first one will be in the gradient calculations. Instead of calculating gradients over the entire training set, we will calculate gradients over a smaller batch (or subset) of the training data. This means that the gradient updates will be noisier, but our training should be significantly faster! We need to make sure that the gradients are properly scaled with the batch size. This means that equations 27 and 28 need to be updated to:

$$b_m \leftarrow b_m - \frac{\gamma}{n_{batch}} \sum_{i \in batch} (\tilde{y}_{im} p_{im} - \tilde{y}_{im}) \tag{29}$$

$$w_{mj} \leftarrow w_{mj} - \frac{\gamma}{n_{batch}} \sum_{i \in batch} (\tilde{y}_{im} p_{im} - \tilde{y}_{im}) x_{ij} \tag{30}$$

The final thing to keep in mind is that we need to decrease the learning rate when using mini-batch SGD. The learning rate needs to decrease over time so that convergence is guaranteed. We will be using a linear schedule:

$$\gamma_{it} = (1 - \frac{it}{\tau})\gamma_0 + \frac{it}{\tau}\gamma_\tau \tag{31}$$

Here γ_0 is the initial learning rate that we start with, γ_τ is the learning rate that we use after τ iterations (it is iteration counter). The values for the learning rates and important parts of the Python implementation will be given below.

3.1 Results

We will highlight major differences between the softmax-only and mini-batch softmax implementations before showing the results. One difference is that we have two loops instead of one loop. In the current implementation we have a while-loop over the epochs, while the optimization loop is a for-loop. Each epoch, we will loop $\frac{n_{train}}{n_{batch}}$ times over the training data (an iteration is a gradient update using a mini-batch). The most important code is the mini-batching code. We shuffle our training data each epoch randomly. We also need to shuffle our training labels. The shuffling is done as follows:

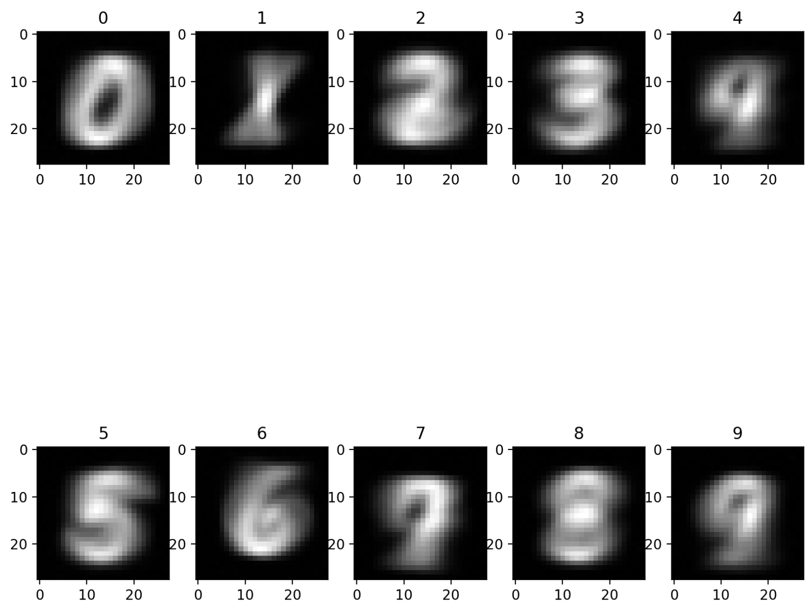
```
        ### Shuffling indices
        ind = np.arange(n_train)
        np.random.shuffle(ind) # shuffle indices

        ### Shuffling training data and labels
        xt = xtrain[ind, :]
        yt = ytrain[ind, :]

        ytrue_shuff = np.argmax(yt, axis = 1)
```

The next important implementation detail is the mini-batching done each iteration. This is done as follows:

```
        mini_batch = np.random.randint(0, n_train, size = nb) # batch indices
```

The rest of the Python implementation is similar, where we need to be careful to only calculate the z_{im} and etc. for the current mini-batch. For more details, please refer to the full Softmax regression with mini-batching code. Since a learning rate of 0.01 was found to be desirable we will be using this value for γ_0. For γ_τ we will use a rule-of-thumb to set it to 1% of γ_0. The value of τ is slightly more difficult to set. The choice has been made to keep the learning rate constant for the final 5 iterations. For the mini-batch size of 1500 that was settled on, each epoch lasts 40 iterations. We will train for 300 epochs and calculate/plot the training and testing cost/accuracy every 5 iterations. In order to compare differences, we will look at a smaller batch-size of 150. For a batch-size equal to the training data size we expect similar results to the softmax-only implementation.

(a) $\gamma = 0.01$ and batch-size of 150



(b) $\gamma = 0.01$ and batch-size of 1500

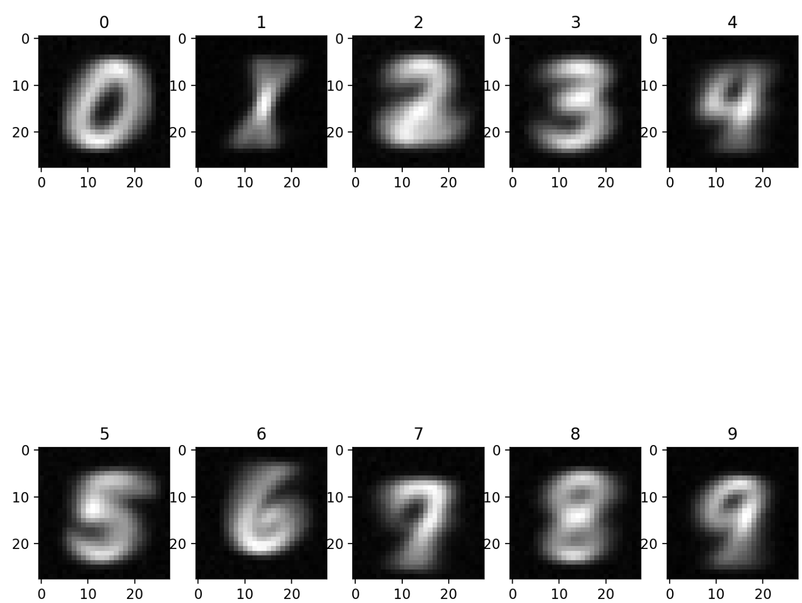
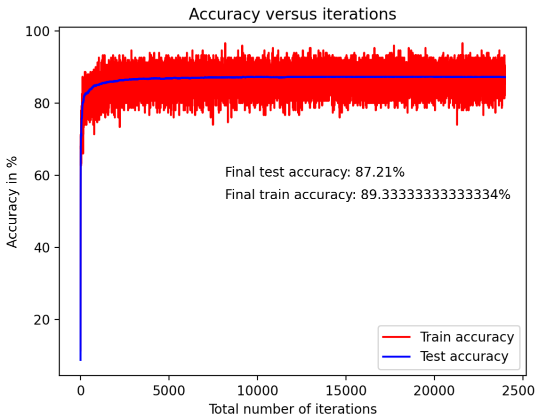
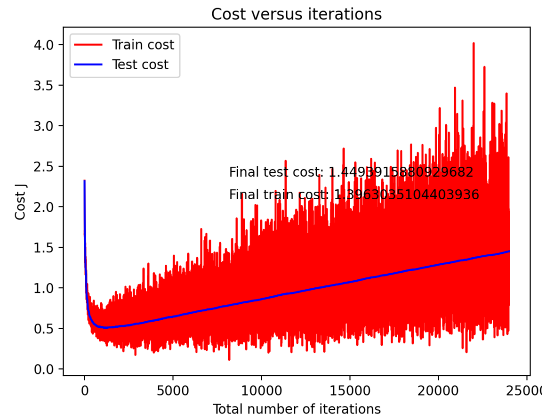


Figure 4: All 10 weight matrices for two different learning rates. The learning rate and batch-size are indicated in the subcaptions. No post-processing was performed on the weights obtained. The weight matrices were only reshaped from 784 to 28 x 28 images.

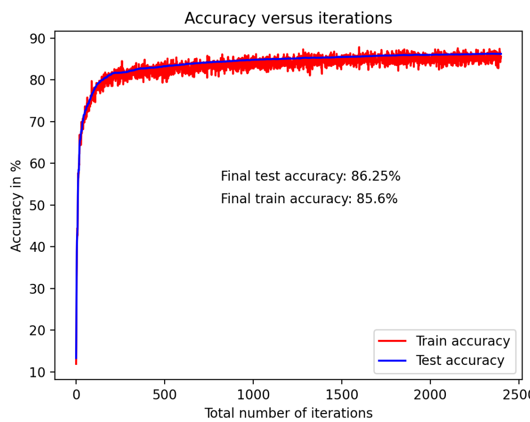
(a) $\gamma = 0.01$ and batch-size of 150



(b) $\gamma = 0.01$ and batch-size of 150



(c) $\gamma = 0.01$ and batch-size of 1500



(d) $\gamma = 0.01$ and batch-size of 1500

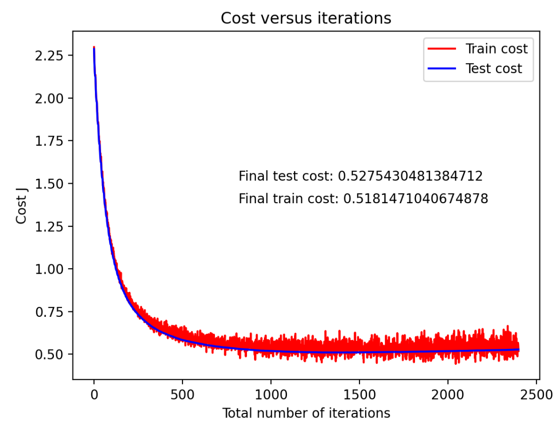


Figure 5: Accuracy and cost for the weights in Figure 4.

Compared to the softmax-only implementation we can see with the same parameters except batch-size there is a large difference between using a batch-size of 150 and 1500. In Figure 4 we can see that for most of the 10 classes, the weights are blurrier for a batch-size of 150 versus one of 1500. However, for the digit "4" the differences are not as clear for a batch size of 150 and 1500. There is a much clearer difference between the two batch sizes when we look at Figure 5. In this we can see that both the training cost and accuracy fluctuates much more for a batch size of 150 compared to 1500. This is to be expected since we know that for a smaller batch-size the gradient updates are much noisier. This was confirmed by looking at the evolution of the gradient of the bias versus iterations. However, even though the training accuracy fluctuates a lot it does converge to a final value of about 89.3%. The final test accuracy for a batch-size of 150 was 87.21%. Compare this to the final training accuracy of 85.6% and final test accuracy of 86.25% for a batch-size of 1500. Although the test accuracy is higher than the softmax-only result, it is still not able to reach the 92% target. Maybe doing a parameter search using a function like GridSearchCV might result in optimal values for the mini-batch softmax implementation. Next, we will look at the full neural network implementation.

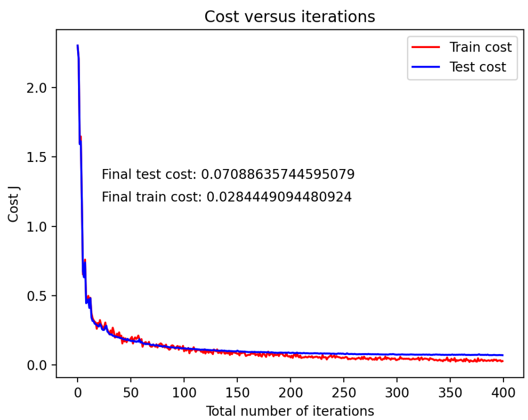
4 Full Neural Network

In this section we will implement an L-layer neural network (note that the previous two networks were one-layer networks). The process is similar to the softmax implementations. One of the major differences is that the mathematics is more involved. We need to obtain equations for the forward and backward propagation. During the forward propagation we just propagate the input sequentially through all layers and calculate the cost/loss. During the backward propagation we evaluate all the gradients from the last layer back to the input with respect to the hidden units. We will be using the results from the Supervised Machine Learning book in our Python implementation. We will follow backpropagation Algorithm 6.1 and use the derived equations in our implementation. This algorithm only gives us the cost function and the gradient(s) of the cost function. We will update the weights and biases using gradient descent (stochastic since we are using mini-batch) like previously.

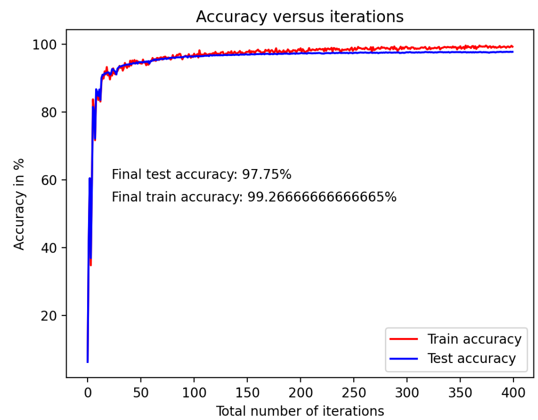
4.1 Results

Similar to the previous two sections, the full Python code can be found in the Full Neural Network Appendix section. The main optimization loop looks quite similar to the code for the previous two sections. The current implementation of the neural network only has code implemented for 2-layer and 4-layer neural networks. We will therefore make comparisons between the 2-layer and 4-layer results. We want to start by stating that an optimal network architecture for this task seems to be a 2-layer network. Also note that we will be showing results only for ReLU activations. We will also accumulate the cost/accuracy values every 5 iterations and use a batch-size of 1500 for both the 2-layer and the 4-layer network. The number of hidden units was also kept the same at 100 and 100,100, and 100 for the 2-layer and 4-layer network respectively. The 2-layer network seemed stable with various hidden units. The 4-layer network was much more difficult to configure, but the results that will be shown are the "optimal" ones. Before showing the results we also want to note that the learning rate was kept at a constant value for both the 2-layer and 4-layer network. Using a decaying learning rate we saw that both networks exhibited oscillations in the cost/accuracy. They did manage to converge to a "reasonable" final value, but these oscillations are clearly not desirable. The learning rate was set to a constant value for both networks. We ran both networks for 50 epochs. Note that for the various settings this means we have plots over 400 iterations. Both networks had their weight matrices initialized according to a normal distribution with mean 0 and standard deviation 0.01. The biases were all initialized to 0.

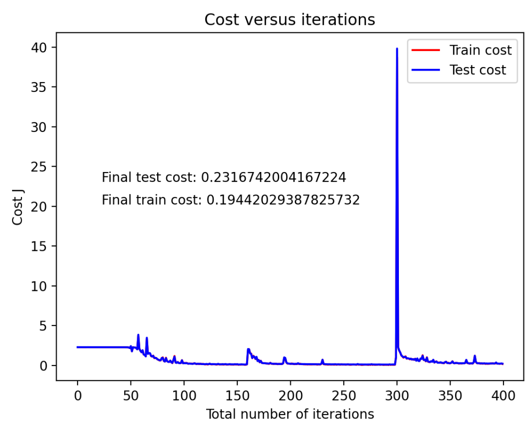
(a) $\gamma = 0.8$ for 2-layer network



(b) $\gamma = 0.8$ for 2-layer network



(c) $\gamma = 0.8$ for 4-layer network



(d) $\gamma = 0.8$ for 4-layer network

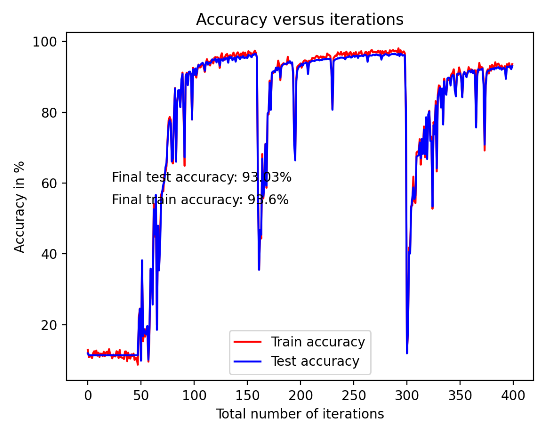


Figure 6: Accuracy and cost plots for 2-layer and 4-layer networks.

4.2 Discussion

One of the big differences between the 2-layer and 4-layer network is that for the same number of iterations we can see that the cost and accuracy for the 2-layer network converge nicely. The final testing accuracy for the 2-layer network is 97.75%, which is close to the target of 98%. The final test accuracy for the 4-layer network is around 93%. However, note that the 4-layer network exhibits a much less stable cost/accuracy evolution. An increase at around 300 iterations occurs in both the training and testing cost, with a corresponding decrease in the training and testing accuracy. Despite this instability, the 4-layer network does manage to converge to a reasonable final accuracy. It is in fact much more than the softmax-only/mini-batch softmax implementations. The 4-layer network is also much more sensitive to initialization since running the 4-layer network with the same parameters resulted in wildy-varying final testing accuracies. Again, a parameter-search for the 4-layer network might result in a much more well-behaved network for this specific data set. A final thing of note is that the 2-layer network also exhibits slight fluctuation in the testing accuracy and testing cost. For the softmax-only and mini-batch softmax implementations this effect is not visible.

5 Appendix

5.1 Loading function

```python
import numpy as np
import imageio
import glob

# Small modification: added absolute path of Test and Train
↪ folders

def load_mnist():
    # Loads the MNIST dataset from png images

    NUM_LABELS = 10
    # create list of image objects
    test_images = []
    test_labels = []

    for label in range(NUM_LABELS):
```

```
        for image_path in glob.glob("/Users/tongyou/Desktop/Deep-learning-course/Assignment-1/MNIST/Test/"+ str(label) + "/*.png"):
            image = imageio.imread(image_path)
            test_images.append(image)
            letter = [0 for _ in range(0,NUM_LABELS)]
            letter[label] = 1
            test_labels.append(letter)

    # create list of image objects
    train_images = []
    train_labels = []

    for label in range(NUM_LABELS):
        for image_path in glob.glob("/Users/tongyou/Desktop/Deep-learning-course/Assignment-1/MNIST/Train/" + str(label) + "/*.png"):
            image = imageio.imread(image_path)
            train_images.append(image)
            letter = [0 for _ in range(0,NUM_LABELS)]
            letter[label] = 1
            train_labels.append(letter)

    X_train= np.array(train_images).reshape(-1,784)/255.0
    Y_train= np.array(train_labels)
    X_test= np.array(test_images).reshape(-1,784)/255.0
    Y_test= np.array(test_labels)

    return X_train, Y_train, X_test, Y_test
```

5.2 Softmax regression

```
import numpy as np
from load_mnist import load_mnist
from matplotlib import pyplot as plt

x_train, y_train, x_test, y_test = load_mnist()
```

```python

M = 10
n_train = x_train.shape[0] # number of training examples - 60000
p = x_train.shape[1] # number of input pixels - 784 (flattened 28x28 image)

n_test = x_test.shape[0]

def softmax_gd(xtrain, ytrain, xtest, ytest, n_train, n_test, lr, maxit):

    w_mj = np.random.normal(scale = 0.02, size = (M, p)) # weight matrix
    b_m = np.zeros(shape = (1, M)) # offset vector
    z_im = np.zeros(shape = (n_train, M)) # model in (n x M)
    dJdbm = np.zeros(shape = (1, M))
    dJdwmj = np.zeros(shape = (M, p))

    J = np.zeros(shape = (maxit, 1)) # cost vector
    acc_train = np.zeros(shape = (maxit, 1)) # classifcation accuracy

    J_test = np.zeros(shape = (maxit, 1)) # cost vector
    acc_test = np.zeros(shape = (maxit, 1)) # classifcation accuracy

    it = 0

    while it != maxit:

        z_im = xtrain @ w_mj.T + b_m

        y_im = ytrain

        z_im_norm = z_im - np.max(z_im, axis = 1, keepdims = True)
```

```python

        p_im = np.exp(z_im_norm) / np.sum(np.exp(z_im_norm), axis = 1, keepdims = True)

        dJdzim = (1/n_train) * (y_im * p_im - y_im)

        dJdbm = np.sum(dJdzim, axis = 0)
        dJdwmj = dJdzim.T @ xtrain

        b_m = b_m - lr * dJdbm
        w_mj = w_mj - lr * dJdwmj

        # Cost and accuracy for training mini-batch
        L_i = np.sum(y_im * np.log(np.sum(np.exp(z_im_norm), axis = 1, keepdims = True)) - y_im * z_im_norm, axis = 1)
        J[it] = (1/n_train) * np.sum(L_i)

        ypred_train = np.argmax(p_im, axis = 1)
        ytrue_train = np.argmax(y_train, axis = 1)
        acc_train[it] = np.array([1 for i in range(0,n_train) if ypred_train[i] == ytrue_train[i]]).sum()

        # Cost and accuracy for testing data
        z_test = xtest @ w_mj.T + b_m
        z_test_norm = z_test - np.max(z_test, axis = 1, keepdims = True)
        L_i_test = np.sum(ytest * np.log(np.sum(np.exp(z_test_norm), axis = 1, keepdims = True)) - ytest * z_test_norm, axis = 1)
        J_test[it] = (1/n_test) * np.sum(L_i_test)

        p_test = np.exp(z_test_norm) / np.sum(np.exp(z_test_norm), axis = 1, keepdims = True)

        ypred_test = np.argmax(p_test, axis = 1)
```

```
        ytrue_test = np.argmax(y_test, axis = 1)
        acc_test[it] = np.array([1 for i in range(0,n_test) if ypred_test[i] == ytrue_test[i]]).sum()

        it += 1
        print("Iteration: (%s/%s)" % (it, maxit))

    return J, acc_train * (1/n_train) * 100, J_test, acc_test * (1/n_test) * 100, it, w_mj, b_m , z_im, p_im

lr = 0.01
iters = 5000

J, acc_train, J_test, acc_test, it, wmj, bm, zim, pim = softmax_gd(x_train, y_train, x_test, y_test, n_train, n_test, lr, iters)

plt.figure(1)
plt_J, = plt.plot(J, 'r')
plt_J_test, = plt.plot(J_test, 'b')
plt.legend([plt_J, plt_J_test], ['Train cost', 'Test cost'])
plt.annotate("Final train cost: %s" % (float(J[-1])), xycoords = 'figure fraction',xy = (0.4,0.5))
plt.annotate("Final test cost: %s" % (float(J_test[-1])), xycoords = 'figure fraction',xy = (0.4,0.55))
plt.xlabel('Number of iterations')
plt.ylabel('Cost J')
plt.title('Cost versus iterations')
print("Final train cost: %s" % float(J[-1]))
print("Final test cost: %s" % float(J_test[-1]))

plt.figure(2)
plt_acc_train, = plt.plot(acc_train, 'r')
plt_acc_test, = plt.plot(acc_test, 'b')
plt.legend([plt_acc_train, plt_acc_test], ['Train accuracy', 'Test accuracy'])
```

```
plt.annotate("Final train accuracy: %s%%" % (float(acc_train[-1])), xycoords = 'figure fraction',xy = (0.4,0.5))
plt.annotate("Final test accuracy: %s%%" % (float(acc_test[-1])), xycoords = 'figure fraction',xy = (0.4,0.55))
plt.xlabel('Number of iterations')
plt.ylabel('Accuracy in %')
plt.title('Accuracy versus iterations')
print("Final train accuracy: %s%%" % float(acc_train[-1]))
print("Final test accuracy: %s%%" % float(acc_test[-1]))

figw, axw = plt.subplots(2, 5, figsize = (8,8))
axw[0,0].imshow(wmj[0, :].reshape(28,28), cmap = 'gray')
axw[0,0].set_title('0')
axw[0,1].imshow(wmj[1, :].reshape(28,28), cmap = 'gray')
axw[0,1].set_title('1')
axw[0,2].imshow(wmj[2, :].reshape(28,28), cmap = 'gray')
axw[0,2].set_title('2')
axw[0,3].imshow(wmj[3, :].reshape(28,28), cmap = 'gray')
axw[0,3].set_title('3')
axw[0,4].imshow(wmj[4, :].reshape(28,28), cmap = 'gray')
axw[0,4].set_title('4')
axw[1,0].imshow(wmj[5, :].reshape(28,28), cmap = 'gray')
axw[1,0].set_title('5')
axw[1,1].imshow(wmj[6, :].reshape(28,28), cmap = 'gray')
axw[1,1].set_title('6')
axw[1,2].imshow(wmj[7, :].reshape(28,28), cmap = 'gray')
axw[1,2].set_title('7')
axw[1,3].imshow(wmj[8, :].reshape(28,28), cmap = 'gray')
axw[1,3].set_title('8')
axw[1,4].imshow(wmj[9, :].reshape(28,28), cmap = 'gray')
axw[1,4].set_title('9')

plt.show()
```

5.3 Softmax regression with mini-batching

```python
import numpy as np
from load_mnist import load_mnist
from matplotlib import pyplot as plt

x_train, y_train, x_test, y_test = load_mnist()

M = 10
n_train = x_train.shape[0] # number of training examples - 60000
p = x_train.shape[1] # number of input pixels - 784 (flattened 28x28 image)

n_test = x_test.shape[0] # number of testing examples - 10000
ytrue_test = np.argmax(y_test, axis = 1)

def softmax_gd_minibatch(xtrain, ytrain, xtest, ytest, ep, nb, lr_init, tau, n_train, n_test, k):

    w_mj = np.random.normal(scale = 0.01, size = (M, p)) # weight matrix
    b_m = np.zeros(shape = (1, M))
    z_im = np.zeros(shape = (n_train, M))
    dJdbm = np.zeros(shape = (1, M))
    dJdwmj = np.zeros(shape = (M, p))

    J_train = np.zeros(shape = (ep, 1))
    acc_train = np.zeros(shape = (ep, 1))

    J_test = np.zeros(shape = (ep, 1))
    acc_test = np.zeros(shape = (ep, 1))

    e_p = 0 # epoch counter
    lr0 = lr_init # initial learning rate
    lrt = 0.01 * lr0 # final learning rate
    t_tau = tau
```

```

    tot_it = 0
    it_k = 0

    Jtrainiter = np.zeros(shape = ((((n_train//nb)//(k)) * ep), 1))
    Jtestniter = np.zeros(shape = ((((n_train//nb)//(k)) * ep), 1))

    acctrainiter = np.zeros(shape = ((((n_train//nb)//(k)) * ep), 1))
    acctestiter = np.zeros(shape = ((((n_train//nb)//(k)) * ep), 1))

    while e_p != ep:

        ### Shuffling indices
        ind = np.arange(n_train)
        np.random.shuffle(ind) # shuffle indices

        ### Shuffling training data and labels
        xt = xtrain[ind, :]
        yt = ytrain[ind, :]

        ytrue_shuff = np.argmax(yt, axis = 1)

        it = 0

        Jtrainaccum = []
        acctrainaccum = []

        Jtestaccum = []
        acctestaccum = []

        while it != n_train//nb:

            lr = (1 - (it/t_tau)) * lr0 + (it/t_tau) * lrt

```

```
            mini_batch = np.random.randint(0, n_train, size = nb) # batch indices

            z_im[mini_batch, :] = xt[mini_batch, :] @ w_mj.T + b_m

            y_im = yt[mini_batch, :]

            z_im_norm = z_im[mini_batch, :] - np.max(z_im[mini_batch, :], axis = 1, keepdims = True)

            p_im = np.exp(z_im_norm) / np.sum(np.exp(z_im_norm), axis = 1, keepdims = True)

            dJdzim = (1/nb) * (y_im * p_im - y_im)
            dJdbm = np.sum(dJdzim, axis = 0)
            dJdwmj = dJdzim.T @ xt[mini_batch, :]

            b_m = b_m - lr * dJdbm
            w_mj = w_mj - lr * dJdwmj

            # Calculate the cost and accuracy every k-th iteration for averaging per epoch
            if it % k == 0:
                #Cost and accuracy for training data
                L_i = np.sum(y_im * np.log(np.sum(np.exp(z_im_norm), axis = 1, keepdims = True)) - y_im * z_im_norm, axis = 1)
                ypred = np.argmax(p_im, axis = 1)

                Jtrainaccum.append((1/nb) * np.sum(L_i))
                acctrainaccum.append((1/nb) * np.sum(ypred == ytrue_shuff[mini_batch]))

```

```
            # Cost and accuracy for testing data
            z_test = xtest @ w_mj.T + b_m
            z_test_norm = z_test - np.max(z_test, axis = 1, keepdims = True)
            p_test = np.exp(z_test_norm) / np.sum(np.exp(z_test_norm), axis = 1, keepdims = True)
            L_i_test = np.sum(y_test * np.log(np.sum(np.exp(z_test_norm), axis = 1, keepdims = True)) - y_test * z_test_norm, axis = 1)
            ypred_test = np.argmax(p_test, axis = 1)

            Jtestaccum.append((1/n_test) * np.sum(L_i_test))
            acctestaccum.append((1/n_test) * np.sum(ypred_test == ytrue_test))

         # Calculate the cost and accuracy every k-th iteration and accumulate over all epochs
        if tot_it % k == 0:
            #Cost and accuracy for training data
            L_i = np.sum(y_im * np.log(np.sum(np.exp(z_im_norm), axis = 1, keepdims = True)) - y_im * z_im_norm, axis = 1)
            ypred = np.argmax(p_im, axis = 1)
            if nb == n_train:
                Jtrainiter = []
                acctrainiter = []
                Jtrainiter.append(((1/nb) * np.sum(L_i)))
                acctrainiter.append(((1/nb) * np.sum(ypred == ytrue_shuff[mini_batch])))
            else:
                Jtrainiter[it_k] = ((1/nb) * np.sum(L_i))
                acctrainiter[it_k] = ((1/nb) * np.sum(ypred == ytrue_shuff[mini_batch]))
```

```python
            # Cost and accuracy for testing data
            z_test = xtest @ w_mj.T + b_m
            z_test_norm = z_test - np.max(z_test, axis = 1, keepdims = True)
            p_test = np.exp(z_test_norm) / np.sum(np.exp(z_test_norm), axis = 1, keepdims = True)
            L_i_test = np.sum(y_test * np.log(np.sum(np.exp(z_test_norm), axis = 1, keepdims = True)) - y_test * z_test_norm, axis = 1)
            ypred_test = np.argmax(p_test, axis = 1)
            if nb == n_train:
                Jtestniter = []
                acctestiter = []
                Jtestniter.append(((1/n_test) * np.sum(L_i_test)))
                acctestiter.append(((1/n_test) * np.sum(ypred_test == ytrue_test)))
            else:
                Jtestniter[it_k] = ((1/n_test) * np.sum(L_i_test))
                acctestiter[it_k] = ((1/n_test) * np.sum(ypred_test == ytrue_test))
            it_k += 1

        Jtrainaccum_av = np.mean(Jtrainaccum)
        acctrainaccum_av = np.mean(acctrainaccum)
        Jtestaccum_av = np.mean(Jtestaccum)
        acctestaccum_av = np.mean(acctestaccum)

        it += 1
        tot_it += 1

        print("Epoch: (%s/%s), iteration: %s" % (e_p + 1, ep, it))

```

```python
        J_train[e_p] = Jtrainaccum_av
        acc_train[e_p] = acctrainaccum_av
        J_test[e_p] = Jtestaccum_av
        acc_test[e_p] = acctestaccum_av

        e_p += 1

    return J_train, 100 * acc_train, J_test, 100 * acc_test, Jtrainiter, Jtestniter, 100 * acctrainiter, 100 * acctestiter, it, w_mj, b_m , z_im, p_im

n_batch = 1500 # batch size ---> 30 iterations per epoch
epochs = 300 # epochs
lr0 = 0.01 # initial learning rate
tau_it = (n_train//n_batch) - 5 # decay
k_plot = 5 # storing accuracy/cost values each k-th iteration

Jtrain, acc_train, Jtest, acc_test, J_trainiter, J_testniter, acc_trainiter, acc_testiter, it, wmj, bm, zim, pim = softmax_gd_minibatch(x_train, y_train, x_test, y_test, epochs, n_batch, lr0, tau_it, n_train, n_test, k_plot)

# Costs and accuracies averaged over k-th iteration per epoch
plt.figure(1)
plt_J, = plt.plot(Jtrain, 'r')
plt_J_test, = plt.plot(Jtest, 'b')
plt.legend([plt_J, plt_J_test], ['Train cost', 'Test cost'])
plt.annotate("Final train cost: %s" % (float(Jtrain[-1])), xycoords = 'figure fraction', xy = (0.4,0.5))
plt.annotate("Final test cost: %s" % (float(Jtest[-1])), xycoords = 'figure fraction', xy = (0.4,0.55))
plt.xlabel('Number of epochs')
plt.ylabel('Cost J')
plt.title('Average cost versus epochs')
print("Final train cost: %s" % float(Jtrain[-1]))
print("Final test cost: %s" % float(Jtest[-1]))
```

```python

plt.figure(2)
plt_acc_train, = plt.plot(acc_train, 'r')
plt_acc_test, = plt.plot(acc_test, 'b')
plt.legend([plt_acc_train, plt_acc_test], ['Train accuracy', 'Test accuracy'])
plt.annotate("Final train accuracy: %s%%" % (float(acc_train[-1])), xycoords = 'figure fraction', xy = (0.4,0.5))
plt.annotate("Final test accuracy: %s%%" % (float(acc_test[-1])), xycoords = 'figure fraction', xy = (0.4,0.55))
plt.xlabel('Number of epochs')
plt.ylabel('Accuracy in %')
plt.title('Average accuracy versus epochs')
print("Final train accuracy: %s%%" % float(acc_train[-1]))
print("Final test accuracy: %s%%" % float(acc_test[-1]))

### Costs and accuracies per k-th iteration
plt.figure(3)
plt_Jtrain_it, = plt.plot(J_trainiter, 'r')
plt_Jtest_it, = plt.plot(J_testniter, 'b')
plt.legend([plt_Jtrain_it, plt_Jtest_it], ['Train cost', 'Test cost'])
plt.annotate("Final train cost: %s" % (float(J_trainiter[-1])), xycoords = 'figure fraction', xy = (0.4,0.5))
plt.annotate("Final test cost: %s" % (float(J_testniter[-1])), xycoords = 'figure fraction', xy = (0.4,0.55))
plt.xlabel('Total number of iterations')
plt.ylabel('Cost J')
plt.title('Cost versus iterations')
print("Final train cost: %s" % float(J_trainiter[-1]))
print("Final test cost: %s" % float(J_testniter[-1]))

plt.figure(4)
plt_acc_train_it, = plt.plot(acc_trainiter, 'r')
plt_acc_test_it, = plt.plot(acc_testiter, 'b')
```

```
plt.legend([plt_acc_train_it, plt_acc_test_it], ['Train accuracy', 'Test accuracy'])
plt.annotate("Final train accuracy: %s%%" % (float(acc_trainiter[-1])), xycoords = 'figure fraction', xy = (0.4,0.5))
plt.annotate("Final test accuracy: %s%%" % (float(acc_testiter[-1])), xycoords = 'figure fraction', xy = (0.4,0.55))
plt.xlabel('Total number of iterations')
plt.ylabel('Accuracy in %')
plt.title('Accuracy versus iterations')
print("Final train accuracy: %s%%" % float(acc_trainiter[-1]))
print("Final test accuracy: %s%%" % float(acc_testiter[-1]))

figw, axw = plt.subplots(2, 5, figsize = (10,10))

axw[0,0].imshow(wmj[0, :].reshape(28,28), cmap = 'gray')
axw[0,0].set_title('0')

axw[0,1].imshow(wmj[1, :].reshape(28,28), cmap = 'gray')
axw[0,1].set_title('1')

axw[0,2].imshow(wmj[2, :].reshape(28,28), cmap = 'gray')
axw[0,2].set_title('2')

axw[0,3].imshow(wmj[3, :].reshape(28,28), cmap = 'gray')
axw[0,3].set_title('3')

axw[0,4].imshow(wmj[4, :].reshape(28,28), cmap = 'gray')
axw[0,4].set_title('4')

axw[1,0].imshow(wmj[5, :].reshape(28,28), cmap = 'gray')
axw[1,0].set_title('5')

axw[1,1].imshow(wmj[6, :].reshape(28,28), cmap = 'gray')
axw[1,1].set_title('6')

axw[1,2].imshow(wmj[7, :].reshape(28,28), cmap = 'gray')
```

```
axw[1,2].set_title('7')

axw[1,3].imshow(wmj[8, :].reshape(28,28), cmap = 'gray')
axw[1,3].set_title('8')

axw[1,4].imshow(wmj[9, :].reshape(28,28), cmap = 'gray')
axw[1,4].set_title('9')

plt.show()
```

5.4 Full Neural Network

```
import numpy as np
from load_mnist import load_mnist
from matplotlib import pyplot as plt

x_train, y_train, x_test, y_test = load_mnist()

def relu(x):

    return np.maximum(0, x)

def sigmoid(x):

    return (np.exp(x)) / (1 + np.exp(x))

def relu_deriv(x):

    x[x <= 0] = 0
    x[x > 0] = 1

    return x

def sigmoid_deriv(x):

    return sigmoid(x) * (1-sigmoid(x))

```

```
def softmax(x):

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

    return p_x

def init_params(M, p, n_hidden):

    W1 = np.random.normal(scale = 0.01, size = (n_hidden[0], p))
    W2 = np.random.normal(scale = 0.01, size = (n_hidden[1], n_hidden[0]))
    W3 = np.random.normal(scale = 0.01, size = (n_hidden[2], n_hidden[1]))
    W4 = np.random.normal(scale = 0.01, size = (M, n_hidden[2]))

    b1 = np.zeros(shape = (n_hidden[0], 1))
    b2 = np.zeros(shape = (n_hidden[1], 1))
    b3 = np.zeros(shape = (n_hidden[2], 1))
    b4 = np.zeros(shape = (M, 1))

    return W1, b1, W2, b2, W3, b3, W4, b4

def calc_cost(nb, mini_batch, batchBool, y_L, z_L, sz):
    if batchBool == True:
        z_L_norm = z_L - np.max(z_L, axis = 1, keepdims = True)
        loss = np.sum(y_L[mini_batch, :] * np.log(np.sum(np.exp(z_L_norm), axis = 1, keepdims = True)) - y_L[mini_batch, :] * z_L_norm, axis = 1, keepdims = True)
        cost = (1/nb) * np.sum(loss, axis = 0, keepdims = True)

        dz_L = - y_L[mini_batch, :] + sz
```

```python
        return cost, dz_L
    else:
        z_L_norm = z_L - np.max(z_L, axis = 1, keepdims = True)
        loss = np.sum(y_L * np.log(np.sum(np.exp(z_L_norm), axis = 1, keepdims = True)) - y_L * z_L_norm, axis = 1, keepdims = True)
        cost = (1/nb) * np.sum(loss, axis = 0, keepdims = True)

        return cost

def forward(xt, mb, batchBool, w1, b1, w2, b2, w3, b3, w4, b4):

    if batchBool == True:
        z_1 = xt[mb, :] @ w1.T + b1.T
        q_1 = relu(z_1)
        z_2 = q_1 @ w2.T + b2.T
        q_2 = relu(z_2)
        z_3 = q_2 @ w3.T + b3.T
        q_3 = relu(z_3)
        z = q_3 @ w4.T + b4.T
        softmax_z = softmax(z)
    else:
        z_1 = xt @ w1.T + b1.T
        q_1 = relu(z_1)
        z_2 = q_1 @ w2.T + b2.T
        q_2 = relu(z_2)
        z_3 = q_2 @ w3.T + b3.T
        q_3 = relu(z_3)
        z = q_3 @ w4.T + b4.T
        softmax_z = softmax(z)

    return z_1, q_1, z_2, q_2, z_3, q_3, z, softmax_z

def backward(q1,q2,q3, z1, z2, z3, dzl, w2, w3, w4, xt, mb):
```

```python

    dq_3 = dzl @ w4

    dz_3 = np.multiply(dq_3, relu_deriv(z3))
    dq_2 = dz_3 @ w3

    dz_2 = np.multiply(dq_2, relu_deriv(z2))
    dq_1 = dz_2 @ w2

    dz_1 = np.multiply(dq_1, relu_deriv(z1))

    dW_4 = (1/n_batch) * dzl.T @ q3
    dW_3 = (1/n_batch) * dz_3.T @ q2
    dW_2 = (1/n_batch) * dz_2.T @ q1
    dW_1 = (1/n_batch) * dz_1.T @ xt[mb, :]

    db_4 = (1/n_batch) * np.sum(dzl, axis = 0, keepdims =
     ↪  True)
    db_3 = (1/n_batch) * np.sum(dz_3, axis = 0, keepdims =
     ↪  True)
    db_2 = (1/n_batch) * np.sum(dz_2, axis = 0, keepdims =
     ↪  True)
    db_1 = (1/n_batch) * np.sum(dz_1, axis = 0, keepdims =
     ↪  True)

    return dW_1, db_1.T, dW_2, db_2.T, dW_3, db_3.T, dW_4,
     ↪  db_4.T

def init_params_2(M, p, n_hidden):

    W1 = np.random.normal(scale = 0.01, size = (n_hidden[0],
     ↪  p))
    W2 = np.random.normal(scale = 0.01, size = (M,
     ↪  n_hidden[0]))

    b1 = np.zeros(shape = (n_hidden[0], 1))
    b2 = np.zeros(shape = (M, 1))
```

```python

    return W1, b1, W2, b2

def backward_2(q1, z1, w1, w2, xt, mb, dzl):
    dq_1 = dzl @ w2
    dz_1 = np.multiply(dq_1, relu_deriv(z1))

    dW_2 = (1/n_batch) * dzl.T @ q1

    db_2 = (1/n_batch) * np.sum(dzl, axis = 0, keepdims = True)

    dW_1 = (1/n_batch) * dz_1.T @ xt[mb, :]

    db_1 = (1/n_batch) * np.sum(dz_1, axis = 0, keepdims = True)

    return dW_1, db_1.T, dW_2, db_2.T

def forward_2(xt, mb, batchBool, w1, b1, w2, b2):

    if batchBool == True:
        z_1 = xt[mb, :] @ w1.T + b1.T
        q_1 = relu(z_1)
        z = q_1 @ w2.T + b2.T
        softmax_z = softmax(z)
    else:
        z_1 = xt @ w1.T + b1.T
        q_1 = relu(z_1)
        z = q_1 @ w2.T + b2.T
        softmax_z = softmax(z)

    return z_1, q_1, z, softmax_z

def neural_network(epochs, nb, M, p, k, xtrain, ytrain, xtest, ytest, ntrain, ntest):

```

```
    ytrue_test = np.argmax(ytest, axis = 1) # labels for
    ↪  testing data

    e_p = 0 # epoch counter
    #lr0 = 0.8 # initial learning rate
    #lrt = 0.01 * lr0 # final learning rate
    #t_tau = 30 # iterations until learning rate is set to
    ↪  constant lrt value

    tot_it = 0 # total iteration counter
    it_k = 0 # k-th iteration counter

    #n_hidden_4 = np.array([50, 50, 50]) # hidden units per
    ↪  layer ---> L - 1 hidden layers
    #w1,b1,w2,b2,w3,b3,w4,b4 = init_params(M, p ,
    ↪  n_hidden_4)

    n_hidden = np.array([100]) # hidden units per layer --->
    ↪  L - 1 hidden layers
    w1,b1,w2,b2 = init_params_2(M, p , n_hidden)

    acctrain = np.zeros(shape = (((n_train//nb)//k) *
    ↪  epochs,1))
    costtrain = np.zeros(shape = (((n_train//nb)//k) *
    ↪  epochs,1))

    acctest = np.zeros(shape = (((n_train//nb)//k) *
    ↪  epochs,1))
    costtest = np.zeros(shape = (((n_train//nb)//k) *
    ↪  epochs,1))

    while e_p != epochs:

        ### Shuffling indices
        ind = np.arange(n_train)
        np.random.shuffle(ind)

        ### Shuffling training data and labels
```

```
        xt = xtrain[ind, :]
        yt = ytrain[ind, :]

        ytrue_train = np.argmax(yt, axis = 1) # labels for
        ↪ training data

        it = 0 # iteration counter for an epoch

        while it != n_train//nb:

            #lr = (1 - (it/t_tau)) * lr0 + (it/t_tau) * lrt
            lr = 0.8

            mini_batch = np.random.randint(0, n_train, size =
            ↪ nb) # batch indices

            #2-layer code
            z1, q1, z, softz = forward_2(xt, mini_batch, True,
            ↪ w1,b1,w2,b2)
            train_cost, dzL = calc_cost(nb, mini_batch, True,
            ↪ yt, z, softz)
            dw1, db1, dw2, db2 = backward_2(q1, z1, w1, w2,
            ↪ xt, mini_batch, dzL)
            _, _, z_test, softz_test = forward_2(xtest,
            ↪ mini_batch, False, w1,b1,w2,b2)

            # 4-layer code
            #z1, q1, z2, q2, z3, q3, z, softz = forward(xt,
            ↪ mini_batch, True, w1,b1,w2,b2,w3,b3,w4,b4)
            #train_cost, dzL = calc_cost(nb, mini_batch,
            ↪ True, yt, z, softz)
            #dw1, db1, dw2, db2, dw3, db3, dw4, db4 =
            ↪ backward(q1, q2, q3, z1, z2, z3, dzL, w2, w3,
            ↪ w4, xt, mini_batch)
            #_, _, _, _, _, _, z_test, softz_test =
            ↪ forward(xtest, mini_batch, False, w1, b1, w2,
            ↪ b2, w3, b3, w4, b4)

```

```
            test_cost = calc_cost(n_test, mini_batch, False, ytest, z_test, softz_test)

            w1 = w1 - lr * dw1
            w2 = w2 - lr * dw2
            b1 = b1 - lr * db1
            b2 = b2 - lr * db2

            #w3 = w3 - lr * dw3
            #w4 = w4 - lr * dw4
            #b3 = b3 - lr * db3
            #b4 = b4 - lr * db4

            if tot_it % k == 0:
                y_predtrain = np.argmax(softz, axis = 1)
                acctrain[it_k] = 100 * ((1/nb) * np.sum(y_predtrain == ytrue_train[mini_batch]))
                costtrain[it_k] = train_cost

                y_predtest = np.argmax(softz_test, axis = 1)
                acctest[it_k] = 100 * ((1/n_test) * np.sum(y_predtest == ytrue_test))
                costtest[it_k] = test_cost

                it_k += 1

            tot_it += 1
            it += 1

            print("Epoch: (%s/%s), iteration: %s" % (e_p + 1, epochs, it))

        e_p += 1

    #return w1,w2,w3,w4,b1,b2,b3,b4, costtrain, acctrain, costtest, acctest
```

```
    return w1,w2, softz, costtrain, acctrain, costtest, acctest

M = 10 # number of classes/ digits
p = x_train.shape[1] # number of input pixels - 784 (flattened 28x28 image)

n_train = x_train.shape[0] # number of training examples - 60000
n_test = x_test.shape[0] # number of testing examples - 10000

n_batch = 1500 # batch size
epochs = 50 # number of epochs

k_acc = 5

w1,w2,sz, costtrain, acctrain, costtest, acctest = neural_network(epochs, n_batch, M, p, k_acc, x_train, y_train, x_test, y_test, n_train, n_test)
#w1,w2,w3,w4,b1,b2,b3,b4, costtrain, acctrain, costtest, acctest = neural_network(epochs, n_batch, M, p, k_acc, x_train, y_train, x_test, y_test, n_train, n_test)

plt.figure(1)
plt_Jtrain_it, = plt.plot(costtrain, 'r')
plt_Jtest_it, = plt.plot(costtest, 'b')
plt.legend([plt_Jtrain_it, plt_Jtest_it], ['Train cost', 'Test cost'])
plt.annotate("Final train cost: %s" % (float(costtrain[-1])), xycoords = 'figure fraction', xy = (0.2,0.5))
plt.annotate("Final test cost: %s" % (float(costtest[-1])), xycoords = 'figure fraction', xy = (0.2,0.55))
plt.xlabel('Total number of iterations')
plt.ylabel('Cost J')
plt.title('Cost versus iterations')
print("Final train cost: %s" % float(costtrain[-1]))
print("Final test cost: %s" % float(costtest[-1]))
```

```

plt.figure(2)
plt_acc_train_it, = plt.plot(acctrain, 'r')
plt_acc_test_it, = plt.plot(acctest, 'b')
plt.legend([plt_acc_train_it, plt_acc_test_it], ['Train accuracy', 'Test accuracy'])
plt.annotate("Final train accuracy: %s%%" % (float(acctrain[-1])), xycoords = 'figure fraction', xy = (0.2,0.5))
plt.annotate("Final test accuracy: %s%%" % (float(acctest[-1])), xycoords = 'figure fraction', xy = (0.2,0.55))
plt.xlabel('Total number of iterations')
plt.ylabel('Accuracy in %')
plt.title('Accuracy versus iterations')
print("Final train accuracy: %s%%" % float(acctrain[-1]))
print("Final test accuracy: %s%%" % float(acctest[-1]))

plt.show()
```