Logistic and Softmax Regression for Handwritten Digits Classification

Shilin Zhu
Ph.D. student, Computer Science
UCSD
La Jolla, CA
shz338@eng.ucsd.edu

Yunhui Guo
Ph.D. student, Computer Science
UCSD
La Jolla, CA
yug185@eng.ucsd.edu

1 Abstract

In this report, we will introduce how to use logistic regression and softmax regression to do handwritten digits classification. We did extensive experiments on the MNIST datasets [2]. For 2-way classification, we can achieve an accuracy of 98% if the targets are digit '2' and digit '3' and we can achieve an accuracy of 97% if the targets are digit '2' and digit '8' by using logistic regression. For 10-way classification for all 10 digits, we can achieve an accuracy of 93.55% by using softmax regression.

2 Logistic Regression via Gradient Descent

2.1 Problem Definition

In this work, we realize the handwritten digit classification using MNIST database. Our goal is to accurately and robustly recognize what number is present in a new image. In this section we will first use logistic regression to classify only two digit classes (binary classification), later in the next section we will use softmax regression to generalize logistic regression into N classes.

2.2 Mathematical Derivation and Methods

Derive the gradient for logistic regression:
The cross-entropy cost function can be expressed as

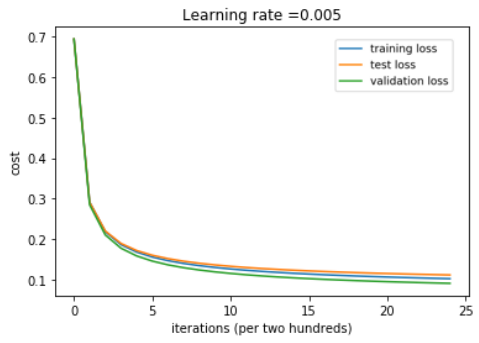
$$E(w) = -\frac{1}{N}\sum_{n=1}^{N}[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)] \tag{1}$$

where t^n is the target label for example n and y^n is our prediction for this example. To perform gradient descent, we need to first compute the gradient (derivative) of the cost function with respect to the parameters. The gradient of cost function on example n is

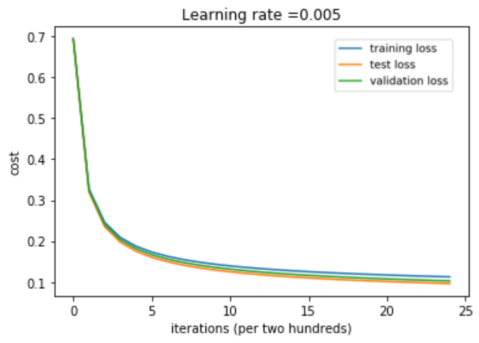
$$-\frac{\partial E^n(w)}{\partial w_j} = \frac{1}{N}\frac{\partial[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)]}{\partial w_j} \tag{2}$$

where y^n is the prediction of logistic regression as

$$y^n = g(\sum_{j=0}^{m} w_j x_j^n) \tag{3}$$



(a) Cost function with iterations for batch GD on classifying 2 vs. 3



(b) Cost function with iterations for batch GD on classifying 2 vs. 8

Figure 1: Cost function through batch GD training process

and $g(\cdot)$ is the sigmoid activation function and its derivative is

$$g'(z^n) = \frac{d(\frac{1}{1+e^{-z^n}})}{dz^n} = g(z^n)(1 - g(z^n)) \tag{4}$$

where $z^n = \sum_{j=0}^{m} w_j x_j^n$. According to the chain rule in calculus, we can then compute the gradient (derivative) of the cost function on example n with respect to the parameters as

$$-\frac{\partial E^n(w)}{\partial w_j} = -\frac{\partial E^n(w)}{\partial y^n}\frac{\partial y^n}{\partial z^n}\frac{\partial z^n}{\partial w_j} = \frac{1}{N}(\frac{t^n}{y^n} - \frac{1-t^n}{1-y^n}) \cdot y^n(1-y^n) \cdot x_j^n = \frac{1}{N}(t^n - y^n)x_j^n \tag{5}$$

Note that we can always add a factor of $1/N$ to scale the cost and gradient to somehow speed up the learning, and this will not change the optimization results (learned parameters).

2.3 Data Reading, Parsing and Feature Extraction

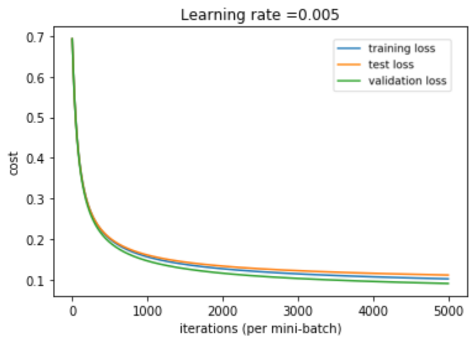
In this work we use the famous MNIST hand written digits database created by Yann LeCun. Each image in the database contains 28×28 pixels and each pixel has a grayscale intensity, so that the input feature $x \in R^{784}$ after we unroll the 2D image into an 1D vector. To include the bias term, after reading the data, we append a '1' to the beginning of each x vector so the final $x \in R^{785}$. We will use the first 20,000 training images and the last 2,000 test images. Note that for logistic regression to do binary classification, we can only use the images of two specific digit classes, so that the actual training images and test images are smaller than 20,000 and 2,000 respectively. To speed up learning, we need to apply feature normalization. For images, the most common way is to normalize the pixel values by the maximum pixel value 255. After normalization, all the values in x is now ranging from 0 to 1.

The following code shows how we can choose the example images corresponding to two specific digit classes. For the rest of the report, we choose two binary classification problems: 2 vs. 3 and 2 vs. 8.

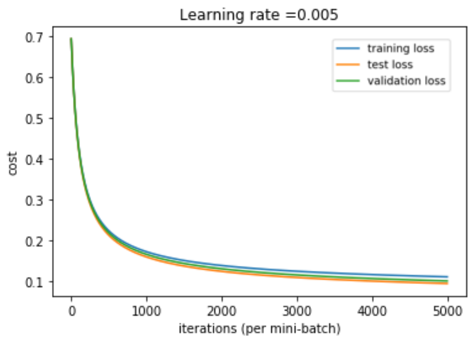
2.4 Experimental Results and Discussion

Batch gradient descent
We first apply batch gradient descent rule on logistic regression since the training set is not that huge, thus batch gradient descent can work reasonably fast.

After trying different values of learning rate, we found 0.005 is a reasonably good learning rate for this problem. Fig. 1 plots the loss function over training for the training set, the hold-out validation

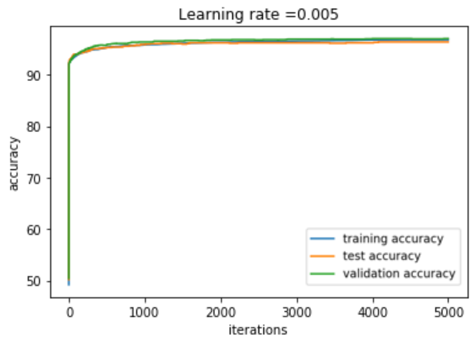


(a) Cost function with iterations for mini-batch GD on classifying 2 vs. 3

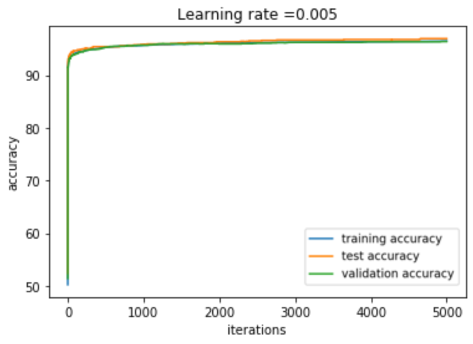


(b) Cost function with iterations for mini-batch GD on classifying 2 vs. 8

Figure 2: Cost function through mini-batch GD training process



(a) Accuracy with iterations on classifying 2 vs. 3



(b) Accuracy with iterations on classifying 2 vs. 8

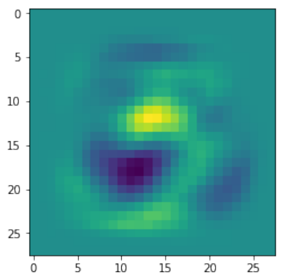
Figure 3: Accuracy through training process

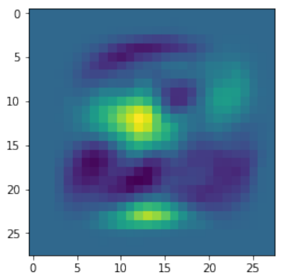
set and the test set. From the results we can see the model tries to minimize the cost and meanwhile maximize the likelihood to perform good prediction on the data. Here for both 2 vs. 8 and 3 vs. 8 cases, the test set accuracies are both around 97% so our model generalizes reasonably well given this large dataset used for the training, as shown in Fig. 3.

Mini-Batch gradient descent
We can use mini-batch gradient descent to speed up learning process. Since the one-step optimization is done on a smaller mini-batch, the cost function will not monotonically decrease as batch gradient descent. Here we change the weights after a mini-batch of roughly 10% of the entire training set. This set of mini-batch size can result in relatively smooth curve of the cost function. Fig. 2 plots the loss function over training for consecutive mini-batches. Since the validation set always has very similar errors as the test set, we can conclude the hold-out set work as a good stand-in for the test set and their underlying data distributions are same.

Accuracy of Prediction Using Logistic Regression Classifier
From batch gradient descent and mini-batch gradient descent, we can roughly achieve 97% classification accuracy on the test set for binary case. This means that our model can perform pretty well on the unseen data.

Weight Visualization
We can visualize the weights learned to see what logistic regression learns through the training process. Fig. 8 shows that the model indeed learn the representation of numbers we want it to learn. The neurons are activated and de-activated based on the structure of the handwritten digit. And since the cost function is cross-entropy loss, which means the maximum likelihood corresponds to the maximum prediction. In order to achieve maximum prediction value which is the inner product


(a) Weights learned on classifying 2 vs. 3


(b) Weights learned on classifying 2 vs. 8

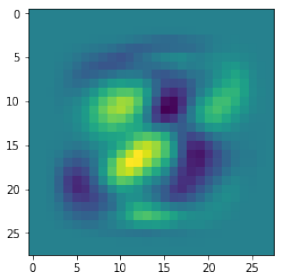

(c) Difference of the weights between the two classifiers

Figure 4: Weight visualization on binary classification using logistic regression

of input X and weights, the angle between these two vectors in the feature space should be zero so that their inner product is maximized. Thus the weights look very similar to X in the feature space and so as our 2-D visualization.

Computing the difference between these two different classifiers, we can get a new weight matrix visualized in Fig. 8. This result shows that the new weights can be used to accurately classify digit 3 and digit 8 since the neurons will be activated and de-activated based on these new weights.

In the next part of this section, we are going to add regularization to our model and analyze it via experiments.

2.5 Derive the gradient for regularized logistic regression:

To prevent potential overfitting, regularization is used in logistic regression. The cross-entropy cost function with regularization term can be computed as

$$E(w) = -\frac{1}{N}\sum_{n=1}^{N}[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)] + \lambda * C(w) \tag{6}$$

where $C(w)$ represents the complexity of the model. $L1$ and $L2$ regularizations are two most common functions people use

$$C(w) = ||w||^2 = \sum_{i,j} w_{i,j}^2, \text{if use } L_2 \text{ regularization} \tag{7}$$

$$C(w) = |w| = \sum_{i,j} |w_{i,j}|, \text{if use } L_1 \text{ regularization} \tag{8}$$

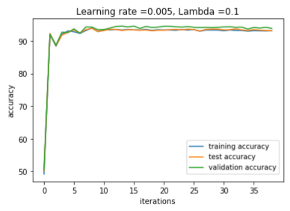
Thus the gradient of cost function on example n is

$$-\frac{\partial E^n(w)}{\partial w_j} = \begin{cases} \frac{1}{N}(t^n - y^n)x_j^n - 2\lambda w_j, & \text{if use } L_2 \text{ regularization} \\ \frac{1}{N}(t^n - y^n)x_j^n - \lambda sign(w_j), & \text{if use } L_1 \text{ regularization} \end{cases}$$

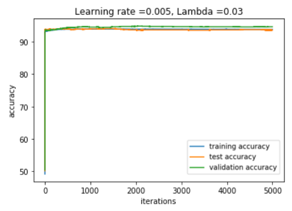
where $sign(w_j)$ is the signature function of w_j.

Note that we can always add a factor of $1/N$ to scale the cost and gradient to somehow speed up the learning, and this will not change the optimization results (learned parameters).

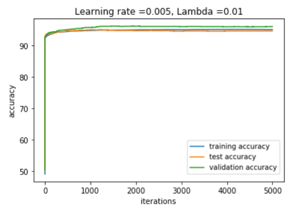
Tuning the Regularization Parameter
In order to choose a reasonably good regularization model, we need to tune the regularization parameter and choose the one which results in the best accuracy on the hold-out validation set. Here we try different values for regularization parameter λ: 0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, and



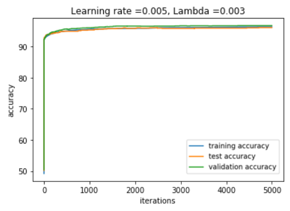
(a) Accuracy when lambda = 0.1 using L1 regularization



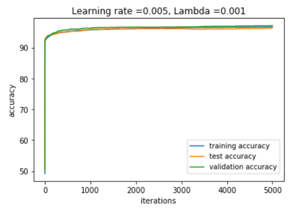
(b) Accuracy when lambda = 0.03 using L1 regularization



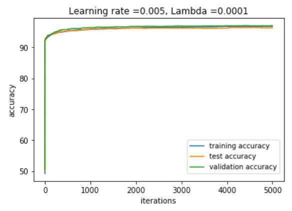
(c) Accuracy when lambda = 0.01 using L1 regularization



(d) Accuracy when lambda = 0.003 using L1 regularization

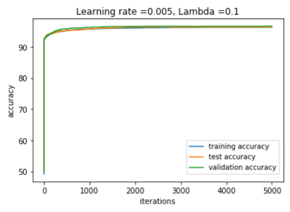


(e) Accuracy when lambda = 0.001 using L1 regularization

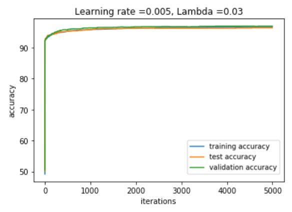


(f) Accuracy when lambda = 0.0001 using L1 regularization

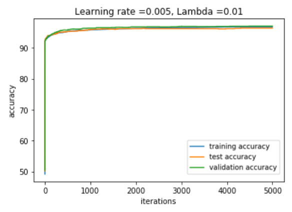
Figure 5: Accuracy using L1 regularization



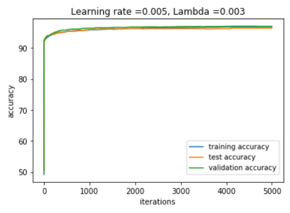
(a) Accuracy when lambda = 0.1 using L2 regularization



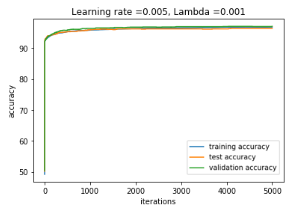
(b) Accuracy when lambda = 0.03 using L2 regularization



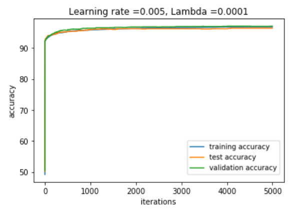
(c) Accuracy when lambda = 0.01 using L2 regularization



(d) Accuracy when lambda = 0.003 using L2 regularization



(e) Accuracy when lambda = 0.001 using L2 regularization



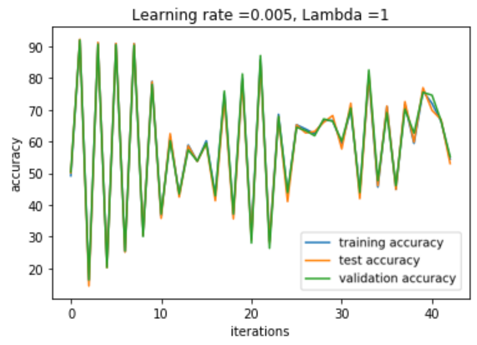
(f) Accuracy when lambda = 0.0001 using L2 regularization

Figure 6: Accuracy using L2 regularization

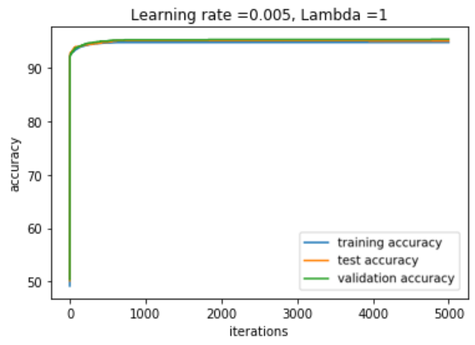
0.1. Fig. 5 and Fig. 6 show the results with different λ on L1 and L2 regularization. From the experimental results we can see as soon as we do not set λ too large, we can always get reasonably well performance (so the performance is not very sensitive to the regularization strength) if we are training our model using a large enough dataset.

If we set lambda very large ($\lambda = 1$) as shown in Fig. 7, we can see that L1 regularization will be over strong and cause the model fail to converge and may experience underfit problem. L2 regularization is more robust against large λ compared to L1 regularization. Thus it is better to use L2 regularization with carefully tuned λ such as from 0.0001 to 0.001 in this example.

Reality Check on the Length of Weight Vector
Fig. 8 shows the experimental results using L1 and L2 regularization. From the figures we can



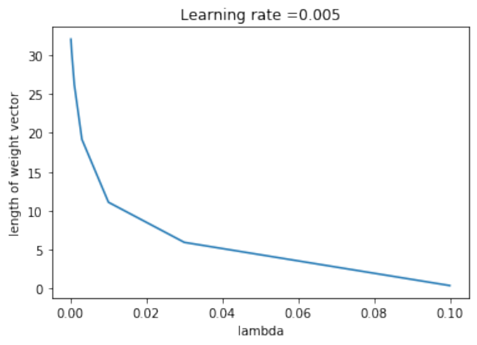
(a) Accuracy when lambda = 1 using L1 regularization



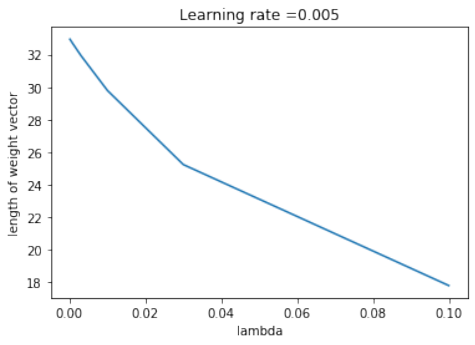
(b) Accuracy when lambda = 1 using L2 regularization

Figure 7: Setting up lambda to be very large values

see that stronger regularization will result in smaller weights since the regularization penalizes large weights. This is consistent with the mathematics of our cost function since if we use a very large λ, the optimizer will pay more attention to minimize the regularization term instead of minimizing the loss between the prediction and the ground truth labels.



(a) Length of weight vector using L1 regularization

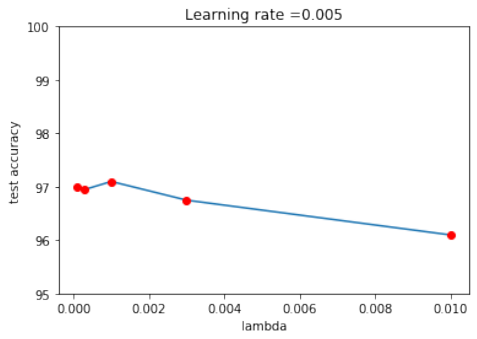


(b) Length of weight vector using L2 regularization

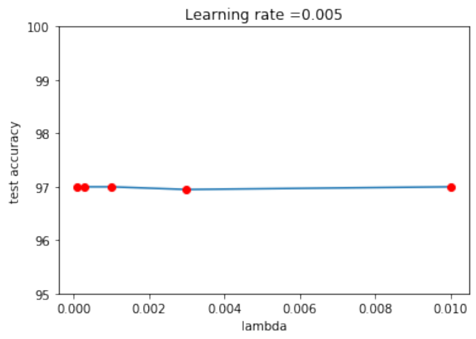
Figure 8: Reality check on the length of the weight vector

Final Test Set Error with Regularization
We also plot the final test set error with different values of the regularization parameter, as shown in Fig. 9. For L1 regularization we can see that when λ is relatively small, the test accuracy will increase as we increase λ. This proves that regularization can sometimes make the model generalize better. In this example the performance improvement is very small since the training dataset is reasonably large and with good quality, so that overfitting is not a great issue anymore. We can also observe that when λ is getting larger and larger, the test accuracy actually decreases (as well as on the training and validation set). This is because although strong regularization makes the model generalize well, over strong regularization will cause the model to be too simple and cannot even fit the data very well (as shown in Fig. 7). But we can conclude that using λ between 0.0001 and 0.001 can all work reasonably well for L1 and L2 regularized model. For L2 regularization, it is more robust and less sensitive as we change λ compared with L1 regularization.

Weight Visualization with Regularization
We further visualize the weights learned with our regularized model, which are shown in Fig. 10. From the results we can see that when λ increases, the learned parameters (weights) become simpler and cannot represent the data very well since we cannot see the structure of the digits anymore under the case with too large λ. But when λ is set a good enough value, then they can learn the structure and distribution of the data quite well. From this result we can also see that L2 regularization is less

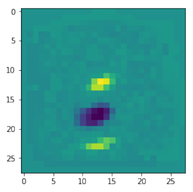


(a) Final test accuracy using L1 regularization

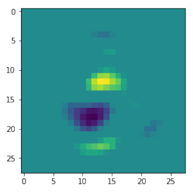


(b) Final test accuracy using L2 regularization

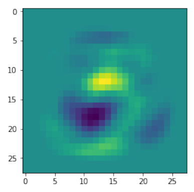
Figure 9: Final test accuracy using regularization



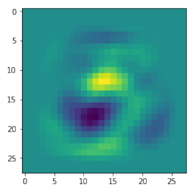
(a) Weights when lambda = 0.1 using L1 regularization



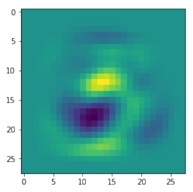
(b) Weights when lambda = 0.01 using L1 regularization



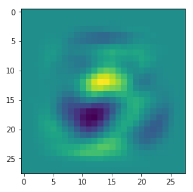
(c) Weights when lambda = 0.001 using L1 regularization



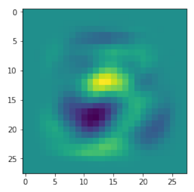
(d) Weights when lambda = 0.0001 using L1 regularization



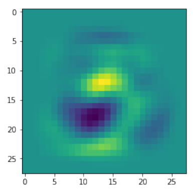
(e) Weights when lambda = 0.1 using L2 regularization



(f) Weights when lambda = 0.01 using L2 regularization



(g) Weights when lambda = 0.001 using L2 regularization



(h) Weights when lambda = 0.0001 using L2 regularization

Figure 10: Weights visualization using regularization

sensitive as we change λ compared with L1 regularization since L2 regularization can still make the model to learn good representations even if λ is large.

To conclude, our logistic regression classifier is well trained and can be generalized to predict on test data very well. Using cross-entropy cost function as the optimization objective, we can visualize the weights which reflect the structure of the digits. Regularization can be applied to the model and choosing a good λ is critical for the classifier. For large training datasets, the overfitting is not an issue with complex models. Since the logistic classifier only has one output unit, to classify all the 10 digits we need to use multiple output units where softmax regression is useful.

3 Softmax Regression via Gradient Descent

3.1 Problem definition

In this problem, we need to classify MNIST datasets using softmax regression. In the experiments, we only use the first 20,000 training images and the last 2,000 test images.

3.2 Methods

Derive the gradient for Softmax Regression:
The cross entropy cost function can be expressed as,

$$E = \sum_{n=1}^{N} \sum_{k=1}^{c} t_k^n \ln y_k^n \tag{9}$$

Where,

$$y_k^n = \frac{\exp\left(a_k^n\right)}{\sum_{k'} \exp\left(a_{k'}^n\right)} \tag{10}$$

And,

$$a_k^n = w_k^T x^n \tag{11}$$

We can calculate the gradient for softmax regression as follows,

$$\begin{aligned} -\frac{\partial E^n(w)}{\partial w_{jk}} &= -\frac{\partial E^n(w)}{\partial a_k^n} \frac{\partial a_k^n}{\partial w_{jk}} \\ &= -\sum_{k'} \frac{\partial E^n(w)}{\partial y_{k'}^n} \frac{\partial y_{k'}^n}{\partial a_k^n} \frac{\partial a_k^n}{\partial w_{jk}} \end{aligned} \tag{12}$$

And

$$\begin{aligned} \frac{\partial y_{k'}^n}{\partial a_k^n} &= \frac{\partial \frac{\exp\left(a_{k'}^n\right)}{\sum_{k'} \exp\left(a_{k'}^n\right)}}{\partial a_k^n} \\ &= y_{k'}^n \delta_{kk'} - y_{k'}^n y_k^n \end{aligned} \tag{13}$$

Where $\delta_{kk} = 1$ if $k = k'$, otherwise $\delta_{kk} = 0$. And

$$\frac{\partial E^n(w)}{\partial y_{k'}^n} = \frac{t_{k'}^n}{y_{k'}^n} \tag{14}$$

Substitute Equation (5) and Equation (6) into Equation (4) we get,

$$-\frac{\partial E^n(w)}{\partial w_{jk}} = (t_k^n - y_k^n) x_j^n \tag{15}$$

Derive the gradient for Softmax Regression with Regularizations:

With regularization, generally the loss function can be writen as,

$$J(w) = E(w) + \lambda C(w) \tag{16}$$

If we use L_1 regularization in softmax regression,

$$\lambda C(w) = \lambda C_{L_1}(w) = \lambda \sum_{j,k} |w_{jk}| \tag{17}$$

We can compute the derivate of $\frac{\partial C}{\partial w}$ as follows,

$$\frac{\partial C_{L_1}(w)}{\partial w_{jk}} = sign(w_{jk}) \tag{18}$$

Where $sign(x) = 1$ if $x > 0$, $sign(x) = 0$ if $x = 0$ and $sign(x) = -1$ if $x < 0$.

If we use L_2 regularization in softmax regression,

$$\lambda C(w) = \lambda C_{L_2}(w) = \lambda \sum_{j,k} w_{jk}^2 \tag{19}$$

We can compute the derivate of $\frac{\partial C}{\partial w}$ as follows,

$$\frac{\partial C_{L_2}(w)}{\partial w_{jk}} = 2w_{jk} \tag{20}$$

In summary,

$$-\frac{\partial J^n(w)}{\partial w_{j,k}} = \begin{cases} (t_k^n - y_k^n)x_j^n - \lambda sign(w_{jk}), & \text{if use } L_1 \text{ regularization} \\ (t_k^n - y_k^n)x_j^n - 2\lambda w_{jk}, & \text{if use } L_2 \text{ regularization} \end{cases}$$

Preprocessing: First, we extract the first 20,000 training images and the last 2,000 test images. Then normailize the images to make sure the pixel values are in the range of [0,1] by dividing each pixel value by 255. And convert the labels to one-hot vectors. Divide the training images into two parts, the first 10% are used for as a hold-out set and the rest 90% are used for training.

Experiments settings: We use standard normal distributions to initilize the weights. We use an initial learning rate $\eta(0)$ = 0.0015 and use equation $\eta(t) = \eta(0)/(1 + t/T)$ to anneal the learning rate by reducing it over time. t is used to index the epoch number and T is a metaparameter which is set to be 3. In the experiements, we find that above learning settings work best.

To determine the best type of regurization and the best λ, we try L_2 regularization and L_1 regularization seperately. For the L_2 regularizartion, we search the best λ in the set $\{0.01, 0.001, 0.0001\}$. If the error on the hold-out set increases for 5 epochs, we stop the algorithm and use the weights with the highest accuracy on the hold-out set as the final answer. For the L_1 regularization, we follow the same steps. We run the 1000 epochs for each setting. Then we compare the results got from these two regularization methods and use the best one as the final result.

We report the average cross entropy loss in the plot because the total cost function depends on the number of training examples.

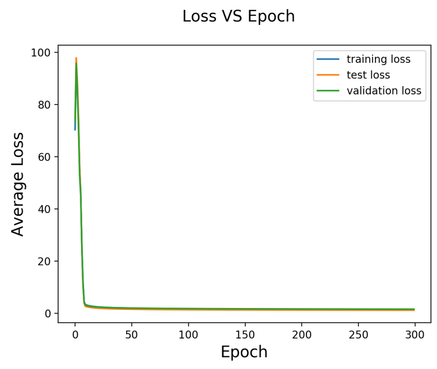
3.3 Results

(a)In the experiments we find that if using L_2 regularization with $\lambda = 0.0001$ obtains the best result on the validation set with an accuracy of 90.1%. The accuracy is 93.55% on the test set under such settings. Using L_1 regularization with $\lambda = 0.0001$ obtains an accuracy of 89.45% on the validation set and with an accuracy of 92.95% on the test set.

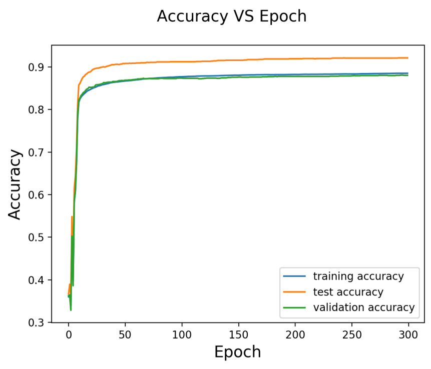
To further examine the performance of the algorithm, we try other λs. We choose λ in the set $\{0.05, 0.005, 0.0005\}$ and use L_1 regularization and L_2 regularization. If use L_1 regularization, the highest accuracy on the validation set is 88.45% with $\lambda = 0.0005$ and the test accuracy is 92.9%. If use L_2 regularization, the highest accuracy on the validation set is 89.85% with $\lambda = 0.0005$ and the test accuracy is 93.4%.

Since using L_2 regularization with $\lambda = 0.0001$ gives us the highest accuracy on the validation dataset, in the following experiments, we use L_2 regularization and fix λ to be 0.0001.

(b) In this experiement, we use L_2 regularization with $\lambda = 0.0001$. The figure is shown in Fig 11a. Note that we use the sum of loss of individual data rather than the average one.



(a) The value of the loss function over the number of training iterations for the training, hold-out, and test set.



(b) The percent correct over the number of training iterations for the training, hold-out and test set.

Figure 11: The performance of the algorithm over the number of training iterations.

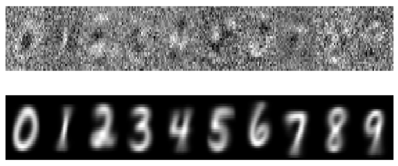


Figure 12: Images of the weights and the average examples. The images in the first row are the images of the weights. The images in the second row are the images of the average examples.

(c) In this experiement, we use L_2 regularization with $\lambda = 0.0001$. The figure is shown in Fig 11b.

(d) We plot the results in Fig 12.

3.4 Discussion

From Fig 11a and Fig 11b and we can see that loss function went up and then converges quickly and then becomes smooth. And the accuracy went up constantly accross diffrent datasets. The possible reason why the loss function went up in the first few epoches is that the learning rates maybe a little large to make to the loss function converge. But then after ”annealing”, we can see that the learining rates become reasonable enough to make the loss function to converge quickly to a local minimum.

We tried setting the learning rate smaller initially but makes the algorihtm too slow to converge or just being stucked. If we set the initial learning rate to be 0.0005, the curve of the loss function becomes smoother, but after 300 epoches, the accuracy on the test set is about 88.9% and is stucked. And if setting the learning rate too large initially makes the loss function up and down and makes it hard to converge. And we also see that there is no overfitting occurs. One possible reason for this is that the impact of the regularization. Another one is that the algorithm correctly learns the pattern underlying the data at hand.

From Fig 12 we can see that the image of the weight and the corresponding image of the average digit is similar. The reason is that we classify the images based on the inner product of the pixels with the weights as we can see from the function below,

$$y_k^n = \frac{\exp\left(w_k^T x^n\right)}{\sum_{k'} \exp\left(w_{k'}^T x^n\right)} \tag{21}$$

So for an image belongs to class k, we want our model to output a large y_k which indicates with a high probability that the image will belong to class k, we should make $w_k^T x^n$ as large as possible. And the inner product is maximized when the angle between the weight and the image is zero. so we see that the image of the weight and the corresponding image of the average digit is similar.

4 Summary

In this work, we successfully derived and implemented logistic regression (binary classification) and softmax regression (multi-class classification) for handwritten digit classification problem from scratch and achieved roughly 98% binary classification accuracy and roughly 93% 10-way classification accuracy. We also embedded regularization to prevent overfitting so that the model generalized well to the unseen test examples. Through rigorous experiments and analysis, we can systematically tune the hyper-parameters and implement model selection by looking at the error on validation set.

5 Contributions

Shilin Zhu did all the derivations, implementation codes, experiments, analysis of logistic regression part in this report (Section 2).

Yunhui Guo did all the derivations, implementation codes, experiments, analysis of softmax regression part in this report (Section 3).

Both Shilin Zhu and Yunhui Guo implemented both logistic and softmax regressions for own study purpose. Three discussions and pair programming were made before submitting this report.

Acknowledgments

We would like to thank Prof. Gary Cottrell and all TAs' efforts in preparing and grading this assignment.

References

[1] Bishop, C. M. (1995). Neural networks for pattern recognition. Oxford university press.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

6 Code

The code below shows the skeleton of our implementation of logistic and softmax regression. To do specific experiments of your own, please change the parameters and indices accordingly to meet your specific needs.

6.1 Logistic regression

```
#packages import
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

#read MNIST data
from mnist import MNIST
mndata = MNIST('./python-mnist/data')
train_images, train_labels = mndata.load_training()
test_images, test_labels = mndata.load_testing()
train_images = np.array(train_images)
test_images = np.array(test_images)
train_labels = np.array(train_labels).reshape([60000,1])
test_labels = np.array(test_labels).reshape([10000,1])

#extract examples corresponding to two specific digits and change the labels to 0 and 1
```

```
train_size = train_images.shape[0]
test_size = test_images.shape[0]
train_images_sub = np.zeros([784,train_size])
train_labels_sub = np.zeros([1,train_size])
test_images_sub = np.zeros([784,test_size])
test_labels_sub = np.zeros([1,test_size])
counter = 0
for i in range(0,train_size):
    if train_labels[i][0] == 2 or train_labels[i][0] == 8:
        train_images_sub[:,counter] = train_images[i,:]
        if train_labels[i][0] == 2:
            train_labels_sub[:,counter] = train_labels[i,:]-2
        if train_labels[i][0] == 8:
            train_labels_sub[:,counter] = train_labels[i,:]-7
        counter = counter + 1
        if counter == 20000:
            break

validation_images_sub = train_images_sub[:,0:int(counter*0.1)]
validation_labels_sub = train_labels_sub[:,0:int(counter*0.1)]
train_images_sub = train_images_sub[:,int(counter*0.1)+1:counter-1]
train_labels_sub = train_labels_sub[:,int(counter*0.1)+1:counter-1]

counter = 0
for i in range(0,test_size):
    if test_labels[i][0] == 2 or test_labels[i][0] == 8:
        test_images_sub[:,counter] = test_images[i,:]
        if test_labels[i][0] == 2:
            test_labels_sub[:,counter] = test_labels[i,:]-2
        if test_labels[i][0] == 8:
            test_labels_sub[:,counter] = test_labels[i,:]-7
        counter = counter + 1
        if counter == 2000:
            break;

test_images_sub = test_images_sub[:,0:counter-1]
test_labels_sub = test_labels_sub[:,0:counter-1]

#normalize the input features
validation_set_x = validation_images_sub/255.
validation_set_y = validation_labels_sub
train_set_x = train_images_sub/255.
test_set_x = test_images_sub/255.
train_set_y = train_labels_sub
test_set_y = test_labels_sub

#helper functions

def sigmoid(z):

    #Compute the sigmoid of z

    a = 1/(1+np.exp(-z))
    return a

def initialize_with_zeros(dim):

    #This function creates a vector of zeros of shape (dim, 1) for w and initializes b to 0.

    w = np.zeros([dim,1])
    b = 0

    return w, b

def propagate(w, b, X, Y, lambda_reg, reg_type):
    #Implement the cost function and its gradient for the propagation

    m = X.shape[1]
    A = sigmoid(np.dot(w.T,X)+b)

    if reg_type == 2:
        cost = -(1/m)*np.sum(np.multiply(Y,np.log(A))+np.multiply(1-Y,np.log(1-A)),
        keepdims=True) + (lambda_reg/2) * np.sum(np.power(w,2))
        dw = (1/m)*(np.dot(X,(A-Y).T)) + (lambda_reg) * w
    if reg_type == 1:
        cost = -(1/m)*np.sum(np.multiply(Y,np.log(A))+np.multiply(1-Y,np.log(1-A)),
        keepdims=True) + (lambda_reg) * np.sum(np.abs(w))
        dw = (1/m)*(np.dot(X,(A-Y).T)) + (lambda_reg) * np.sign(w)

    db = (1/m)*np.sum(A-Y,keepdims=True)

    cost = np.squeeze(cost)

    grads = {"dw": dw,
             "db": db}

    return grads, cost

def optimize(w, b, X, Y, X_val, Y_val, X_test, Y_test, lambda_reg, reg_type, num_iterations,
        learning_rate, print_cost = False, early_stopping = True):

    #This function optimizes w and b by running a gradient descent algorithm

    training_costs = []
    val_costs = []
    test_costs = []
    w_prev = w
    w_prevprev = w
    b_prev = b
    b_prevprev = b
    prediction_accuracy_early_train = []
    prediction_accuracy_early_val = []
    prediction_accuracy_early_test = []

    for i in range(num_iterations):
        if early_stopping == True:
            Y_prediction_test = predict(w, b, X_test)
            Y_prediction_train = predict(w, b, X)
            Y_prediction_val = predict(w, b, X_val)

            train_temp_accuracy = 100 - np.mean(np.abs(Y_prediction_train - Y)) * 100;
            val_temp_accuracy = 100 - np.mean(np.abs(Y_prediction_val - Y_val)) * 100;
            test_temp_accuracy = 100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100;

            prediction_accuracy_early_train.append(train_temp_accuracy)
            prediction_accuracy_early_val.append(val_temp_accuracy)
            prediction_accuracy_early_test.append(test_temp_accuracy)

            #three consecutive error increases
            if np.array(prediction_accuracy_early_val).shape[0] > 3:
                if prediction_accuracy_early_val[i] < prediction_accuracy_early_val[i-1] and
                prediction_accuracy_early_val[i-1] < prediction_accuracy_early_val[i-2] and
                prediction_accuracy_early_val[i-2] < prediction_accuracy_early_val[i-3]:
                    w = w_prevprev
                    b = b_prevprev
                    print('stopped!')
                    break
        grads, cost = propagate(w, b, X, Y, lambda_reg, reg_type)
        val_grads, val_cost = propagate(w, b, X_val, Y_val, lambda_reg, reg_type)
        test_grads, test_cost = propagate(w, b, X_test, Y_test, lambda_reg, reg_type)

        dw = grads["dw"]
        db = grads["db"]

        w_prevprev = w_prev
        b_prevprev = b_prev
        w_prev = w
        b_prev = b
        w = w-learning_rate*dw
        b = b-learning_rate*db



        if i % 200 == 0:
            training_costs.append(cost)
            val_costs.append(val_cost)
            test_costs.append(test_cost)

        if print_cost and i % 200 == 0:
            print ("Training cost after iteration %i: %f" %(i, cost))
            print ("Validation cost after iteration %i: %f" %(i, val_cost))
            print ("Test cost after iteration %i: %f" %(i, test_cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, training_costs, val_costs, test_costs,
        prediction_accuracy_early_train, prediction_accuracy_early_val, prediction_accuracy_e

def predict(w, b, X):

    #Predict whether the label is 0 or 1 using learned logistic regression parameters (w, b)

    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    A = sigmoid(np.dot(w.T,X)+b)

    for i in range(A.shape[1]):

        Y_prediction[0,i] = np.where(A[0,i]>0.5,1,0)

    return Y_prediction

#batch gradient descent
def model(X_train, Y_train, X_val, Y_val, X_test, Y_test, lambda_reg, reg_type,
        num_iterations = 5000, learning_rate = 0.5, print_cost = True, early_stopping = True):

    #Builds the logistic regression model by calling the helper functions

    w, b = np.zeros((X_train.shape[0],1)), 0.

    parameters, grads, costs, val_costs, test_costs, prediction_accuracy_early_train,
        prediction_accuracy_early_val, prediction_accuracy_early_test = optimize(w, b,
        X_train, Y_train, X_val, Y_val, X_test, Y_test, lambda_reg, reg_type, num_iterations,
        learning_rate, print_cost = True, early_stopping = True)

    w = parameters["w"]
    b = parameters["b"]

    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)
    Y_prediction_val = predict(w, b, X_val)

    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train))
         * 100))
    print("Hold-out validation set accuracy: {} %".format(100 - np.mean(np.abs(
        Y_prediction_val - Y_val))* 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test))
         * 100))


    d = {"costs": costs,
         "val_costs": val_costs,
         "test_costs": test_costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "Y_prediction_val": Y_prediction_val,
         "Y_prediction_train_early": prediction_accuracy_early_train,
         "Y_prediction_test_early": prediction_accuracy_early_test,
         "Y_prediction_val_early": prediction_accuracy_early_val,
         "final_test_accuracy": 100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d

#mini-batch gradient descent
def mb_model(X_train, Y_train, X_val, Y_val, X_test, Y_test, lambda_reg, reg_type,
        batch_size, num_epoch, learning_rate, T = 2, print_cost = True, weight_decay = False):

    #Builds the logistic regression model by calling the helper functions

    costs_mb = []
    val_costs_mb = []
    test_costs_mb = []
    w, b = np.zeros((X_train.shape[0],1)), 0.

    for i in range(0,num_epoch):
        if weight_decay == True:
            learning_rate = learning_rate / (1 + i / T)
        for j in range(0,train_set_x.shape[1]-batch_size+1,batch_size):
            parameters, grads, costs, val_costs, test_costs, prediction_accuracy_early_train,
            prediction_accuracy_early_val, prediction_accuracy_early_test = optimize(w, b,
            X_train, Y_train, X_val, Y_val, X_test, Y_test, lambda_reg, reg_type, 1,
            learning_rate, print_cost = True, early_stopping = False)
            w = parameters["w"]
            b = parameters["b"]
            costs_mb.append(costs)
            val_costs_mb.append(val_costs)
            test_costs_mb.append(test_costs)

    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)
    Y_prediction_val = predict(w, b, X_val)

    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train))
         * 100))
    print("Hold-out validation set accuracy: {} %".format(100 - np.mean(np.abs(
        Y_prediction_val - Y_val)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test))
        * 100))


    d = {"costs": costs_mb,
         "val_costs": val_costs_mb,
         "test_costs": test_costs_mb,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "Y_prediction_val": Y_prediction_val,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate}

    return d

#training batch gradient descent
d = model(train_set_x, train_set_y, validation_set_x, validation_set_y, test_set_x,
        test_set_y, lambda_reg = 0, reg_type = 2, num_iterations = 5000, learning_rate = 0.01,
        print_cost = True, early_stopping = False)

#training mini-batch gradient descent
d = mb_model(train_set_x, train_set_y, validation_set_x, validation_set_y, test_set_x,
        test_set_y, lambda_reg = 0, reg_type = 2, batch_size = int(0.1*train_set_x.shape[1]),
        num_epoch = 500, learning_rate = 0.005, T = 2, print_cost = True, weight_decay = False

#weight-decay
d = mb_model(train_set_x, train_set_y, validation_set_x, validation_set_y, test_set_x,
        test_set_y, lambda_reg = 0, batch_size = 500, num_epoch = 100, learning_rate = 0.01,
        T = 10, print_cost = True, weight_decay = True)

#plot accuracy during the training process
train_accuracy = np.squeeze(d['Y_prediction_train_early'])
val_accuracy = np.squeeze(d['Y_prediction_val_early'])
test_accuracy = np.squeeze(d['Y_prediction_test_early'])
plt.plot(train_accuracy)
plt.plot(val_accuracy)
plt.plot(test_accuracy)
plt.legend(['training', 'hold-out(validation)', 'test'], loc='lower right')
plt.ylabel('accuracy')
plt.xlabel('iterations')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()

#plot cost function during the training process
costs = np.squeeze(d['costs'])
val_costs = np.squeeze(d['val_costs'])
test_costs = np.squeeze(d['test_costs'])
plt.plot(costs)
plt.plot(val_costs)
plt.plot(test_costs)
plt.ylabel('cost')
plt.xlabel('iterations (per two hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()

#weight visualization after training
w = d["w"]
print(w.shape)
plt.imshow(w.reshape([28,28]))

#training with L2 regularization for different lambda
lambda_cand = [0.1, 0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001]
norm_len = []
accuracy_train_L2 = []
accuracy_val_L2 = []
```

```
accuracy_test_L2 = []
final_test_set_accuracy_L2 = []
w_L2_lambda = []

for i in range(0,7):
    d = model(train_set_x, train_set_y, validation_set_x, validation_set_y, test_set_x,
        test_set_y, lambda_reg = lambda_cand[i], reg_type = 2, num_iterations = 5000,
        learning_rate = 0.005, print_cost = False, early_stopping = True)
    w = d["w"]
    w_L2_lambda.append(w)
    accuracy_train_L2.append(d["Y_prediction_train_early"])
    accuracy_val_L2.append(d["Y_prediction_val_early"])
    accuracy_test_L2.append(d["Y_prediction_test_early"])
    norm_len.append(np.sum(np.abs(w), keepdims=True))
    final_test_set_accuracy_L2.append(d["final_test_accuracy"])

#training with L1 regularization for different lambda
lambda_cand = [0.1, 0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001]
norm_len = []
accuracy_train_L1 = []
accuracy_val_L1 = []
accuracy_test_L1 = []
w_L1_lambda = []
final_test_set_accuracy_L1 = []

for i in range(0,7):
    d = model(train_set_x, train_set_y, validation_set_x, validation_set_y, test_set_x,
        test_set_y, lambda_reg = lambda_cand[i], reg_type = 1, num_iterations = 5000,
        learning_rate = 0.005, print_cost = True, early_stopping = True)
    w = d["w"]
    w_L1_lambda.append(w)
    accuracy_train_L1.append(d["Y_prediction_train_early"])
    accuracy_val_L1.append(d["Y_prediction_val_early"])
    accuracy_test_L1.append(d["Y_prediction_test_early"])
    norm_len.append(np.sum(np.abs(w), keepdims=True))
    final_test_set_accuracy_L1.append(d["final_test_accuracy"])

#plot length of weight vector after training with regularization
plt.plot(np.squeeze(np.array([0.1, 0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001]).reshape(1,7)),
      np.squeeze(np.array(norm_len[4:11]).reshape(1,7)))
plt.ylabel('length of weight vector')
plt.xlabel('lambda')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()

#plot accuracy during training process
accuracy_train_L1_red = np.squeeze(np.array(accuracy_train_L1)[2])
accuracy_val_L1_red = np.squeeze(np.array(accuracy_val_L1)[2])
accuracy_test_L1_red = np.squeeze(np.array(accuracy_test_L1)[2])

plt.plot(accuracy_train_L1_red)
plt.plot(accuracy_val_L1_red)
plt.plot(accuracy_test_L1_red)
plt.ylabel('accuracy')
plt.xlabel('iterations')
plt.title("Learning rate =" + str(d["learning_rate"]) + ", " + "Lambda =" + str(1))
plt.show()

#plot final test accuracy with different lambda
plt.plot(np.squeeze(np.array([0.01, 0.003, 0.001, 0.0003, 0.0001]).reshape(1,5)),
        np.squeeze(np.array(final_test_set_accuracy_L1[6:11]).reshape(1,5)),
        np.squeeze(np.array([0.01, 0.003, 0.001, 0.0003, 0.0001]).reshape(1,5)),
        np.squeeze(np.array(final_test_set_accuracy_L1[6:11]).reshape(1,5)), 'or')
plt.ylabel('test accuracy')
plt.ylim((95,100))
plt.xlabel('lambda')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()

#visualize the weights after training
w = np.array(w_L1_lambda[0])
print(w.shape)
plt.imshow(w.reshape([28,28]))

#test early stopping
d = model(train_set_x, train_set_y, validation_set_x, validation_set_y, test_set_x,
        test_set_y, lambda_reg = 0, reg_type = 2, num_iterations = 5000,
        learning_rate = 0.005, print_cost = True, early_stopping = True)
```

6.2 Softmax regression

```
#!/usr/bin/python

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from numpy import linalg as LA

import sys
sys.path.append('../python-mnist/')
from mnist import MNIST

# Find best hyperparameters
def hyper_parameters_tuning(training_images, one_hot_training_labels, test_images, test_labels
validation_images, validation_labels, regularization_types, \
lambds, weights, inital_step_size, T, epoch, classes, dimensions):

best_validation_weights_L2 = np.random.randn(dimensions, classes).astype(np.float32)
best_validation_lamdb_L2 = 0
best_validation_accuracy_L2 = 0

best_validation_weights_L1 = np.random.randn(dimensions, classes).astype(np.float32)
best_validation_lamdb_L1 = 0
best_validation_accuracy_L1 = 0

early_stoping_threshholds = 5

for l in regularization_types:
if l == "L2":
for lambd in lambds:
last_accuracy  = 0.0
cnt = 0
for i in range(epoch):
# Batch graident descent
a = np.matmul(training_images, weights)
# Find the largest a, and subtract it from each a in order to prevent overflow
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)
delta = one_hot_training_labels - pred_y
grads = -np.dot(training_images.T, delta) + lambd*weights
# Batch gradient with L2 regularization and learning rate "annealing"
weights = weights - (inital_step_size/(1+i/T)) * grads

# Calculate accuracy on validation set
a = np.matmul(validation_images, weights) #
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1) # 18000, 1
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0) # 18000, 10
```

```
pred_y[pred_y == 0.0] = 1e-15
pred_class = np.argmax(pred_y, axis=1)
validation_accuracy = np.sum(pred_class == validation_labels)/(pred_class.shape[0]+0.0)

if validation_accuracy > best_validation_accuracy_L2:
best_validation_accuracy_L2 = validation_accuracy
best_validation_weights_L2 = weights
best_validation_lamdb_L2 = lambd

if validation_accuracy < last_accuracy:
cnt = cnt + 1
if cnt >= early_stoping_threshholds:
break
else:
cnt = 0
last_accuracy = validation_accuracy
elif l == "L1":
for lambd in lambds:
last_accuracy  = 0.0
cnt = 0

for i in range(epoch):
# Batch graident descent
a = np.matmul(training_images, weights)
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)

delta = one_hot_training_labels - pred_y
# Batch gradient with L1 regularization and learning rate "annealing"
grads = -np.dot(training_images.T, delta) + lambd* np.sign(weights)
weights = weights - inital_step_size/(1+i/T) * grads

# Calculate accuracy on validation set
a = np.matmul(validation_images, weights) #
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)
pred_y[pred_y == 0.0] = 1e-15
pred_class = np.argmax(pred_y, axis=1)
validation_accuracy = np.sum(pred_class == validation_labels)/(pred_class.shape[0]+0.0)

if validation_accuracy > best_validation_accuracy_L1:
best_validation_accuracy_L1 = validation_accuracy
best_validation_weights_L1 = weights
best_validation_lamdb_L1 = lambd

if validation_accuracy < last_accuracy:
cnt = cnt + 1
if cnt >= early_stoping_threshholds:
break
else:
cnt = 0
last_accuracy = validation_accuracy

print "best validation with L1 " + str(best_validation_accuracy_L1)
print "best validation with L1 when lambd = " + str(best_validation_lamdb_L1)
print "best validation with L2 " + str(best_validation_accuracy_L2)
print "best validation with L2 when lambd = " + str(best_validation_lamdb_L2)

# Claculate test accuracy using L1 regularization with the weights achieve highest
# accuracy on the validation set
a = np.matmul(test_images, best_validation_weights_L1)
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
```

```
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)
pred_class = np.argmax(pred_y, axis=1)
test_accuracy = np.sum(pred_class == test_labels)/(pred_class.shape[0]+0.0)
print "test accuracy if using L1 " + str(test_accuracy)

# Claculate test accuracy using L2 regularization with the weights achieve highest
# accuracy on the validation set
a = np.matmul(test_images, best_validation_weights_L2)
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)
pred_class = np.argmax(pred_y, axis=1)
test_accuracy = np.sum(pred_class == test_labels)/(pred_class.shape[0]+0.0)

print "test accuracy if using L2 " + str(test_accuracy)


def train(training_images, test_images, validation_images, training_labels, test_labels, \
validation_labels, one_hot_training_labels, one_hot_test_labels, one_hot_validation_labels, \
lambd, epoch, inital_step_size, T, weights):

training_losses = []
test_losses = []
validation_losses = []

training_accuracy_ = []
test_accuracy_ = []
validation_accuracy_ = []

for i in range(epoch):
print "epoch " + str(i)
# Batch graident descent of softmax regression
a = np.matmul(training_images, weights)
# Find the largest a, and subtract it from each a in order to prevent overflow
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)
delta = one_hot_training_labels - pred_y
grads = -np.dot(training_images.T, delta) + lambd*weights
# Batch gradient with L2 regularization and learning rate "annealing"
weights = weights - (inital_step_size/(1+i/T)) * grads

# Calculate training loss and accuracy
a = np.matmul(training_images, weights)
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1)
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0)
pred_class = np.argmax(pred_y, axis=1)
training_accuracy = np.sum(pred_class == training_labels)/(pred_class.shape[0]+0.0)
training_accuracy_.append(training_accuracy)
pred_y[pred_y == 0.0] = 1e-15
log_pred_y = np.log(pred_y)
training_loss = (-np.sum(one_hot_training_labels * log_pred_y)
+ lambd*LA.norm(weights)**2) / training_images.shape[0]
training_losses.append(training_loss)

# Calculate test loss and accuracy
a = np.matmul(test_images, weights) # 18000, 10
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1) # 18000, 1
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0) # 18000, 10
pred_class = np.argmax(pred_y, axis=1)
test_accuracy = np.sum(pred_class == test_labels)/(pred_class.shape[0]+0.0)
print "test accuracy " + str(test_accuracy)
test_accuracy_.append(test_accuracy)
```

```
pred_y[pred_y == 0.0] = 1e-15
log_pred_y = np.log(pred_y)
test_loss = (-np.sum(one_hot_test_labels * log_pred_y) +
lambd*LA.norm(weights)**2) / test_images.shape[0]
test_losses.append(test_loss)

# Calculate validation loss and accuracy
a = np.matmul(validation_images, weights) # 18000, 10
a_max = np.max(a,1).reshape(a.shape[0],1)
sum_exp_a = np.sum(np.exp(a - a_max),1).reshape(a.shape[0],1) # 18000, 1
pred_y = np.exp(a - a_max) / (sum_exp_a+0.0) # 18000, 10
pred_class = np.argmax(pred_y, axis=1)
validation_accuracy = np.sum(pred_class == validation_labels)/(pred_class.shape[0]+0.0)
validation_accuracy_.append(validation_accuracy)
pred_y[pred_y == 0.0] = 1e-15
log_pred_y = np.log(pred_y)
validation_loss = (-np.sum(one_hot_validation_labels * log_pred_y) +
lambd*LA.norm(weights)**2) / validation_images.shape[0]
validation_losses.append(validation_loss)

fig1 = plt.figure(1)
plt.plot(training_losses)
plt.plot(test_losses)
plt.plot(validation_losses)
fig1.suptitle('Loss VS Epoch', fontsize=15)
plt.legend(['training loss', 'test loss', 'validation loss'], loc='upper right')
plt.xlabel('Epoch', fontsize=15)
plt.ylabel('Average Loss', fontsize=15)
fig1.show()

fig2 = plt.figure(2)
plt.plot(training_accuracy_)
plt.plot(test_accuracy_)
plt.plot(validation_accuracy_)
fig2.suptitle('Accuracy VS Epoch', fontsize=15)
plt.legend(['training accuracy', 'test accuracy', 'validation accuracy'], loc='lower right')
plt.xlabel('Epoch', fontsize=15)
plt.ylabel('Accuracy', fontsize=15)
fig2.show()

# Visualize weights and digits
weights = weights[1:,:]
training_images = training_images[:,1:]

fig3 = plt.figure(figsize = (6,2))
gs = gridspec.GridSpec(2, classes)
gs.update(wspace=0, hspace=0)

for index in range(weights.shape[1]):
weight = weights[:,index]
ave_image = np.zeros((1,784))
for instance in range(training_images.shape[0]):
if training_labels[instance] == index:
ave_image += training_images[instance].reshape(1,784)

ax1 = plt.subplot(gs[0,index])
plt.imshow(np.reshape(weight, (28,28)), cmap=plt.cm.gray, aspect='equal')
plt.axis('off')
ax2 = plt.subplot(gs[1,index])
plt.imshow(np.reshape(ave_image, (28,28)), cmap=plt.cm.gray, aspect='equal')
plt.axis('off')
plt.show()

if __name__ == '__main__':
```

```
data = MNIST('../python-mnist/data')
training_images, training_labels = data.load_training()
test_images, test_labels = data.load_testing()

training_images_old = training_images[:20000]
training_labels_old = training_labels[:20000]
test_images = test_images[-2000:]
test_labels = test_labels[-2000:]
training_images_old = [[1] + image for image in training_images_old]
test_images = [[1] + image for image in test_images]

# Normalize data
training_images = np.array(training_images_old[2000:]) / 255.0
training_labels = np.array(training_labels_old[2000:])
validation_images = np.array(training_images_old[:2000]) / 255.0
validation_labels = np.array(training_labels_old[:2000])
test_images = np.array(test_images) / 255.0
test_labels = np.array(test_labels)

epoch_for_tuning_parameters = 10
inital_step_size = 0.0015
T = 3.0

# trying different lambdas
lambds_set_1 = [0.01, 0.001, 0.0001]
lambds_set_2 = [0.05, 0.005, 0.0005]
regularization_types = ["L1", "L2"]

classes = 1000
dimensions = 785
# Weight initialization
weights = np.random.randn(dimensions, classes).astype(np.float32)  #

one_hot_training_labels = np.eye(classes)[training_labels]
one_hot_validation_labels = np.eye(classes)[validation_labels]
one_hot_test_labels = np.eye(classes)[test_labels]

# Hyperparamter selction
hyper_parameters_tuning(training_images, one_hot_training_labels, test_images, test_labels, \
validation_images, validation_labels, regularization_types, lambds_set_1, weights, \
inital_step_size, T, epoch_for_tuning_parameters, classes, dimensions)

epoch = 300
# This lambd is used with L2 regularization and achieved the highest accuracy on the validatio
lambd = 0.0001
#Training based on the best hyperparameters and using L2 regularization
train(training_images, test_images, validation_images, training_labels, test_labels, \
validation_labels, one_hot_training_labels, one_hot_test_labels, one_hot_validation_labels, \
lambd, epoch, inital_step_size, T, weights)
```