# 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
email

## 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. 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, we can achieve an accuracy of $92.7\%$ 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 of Gradient

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

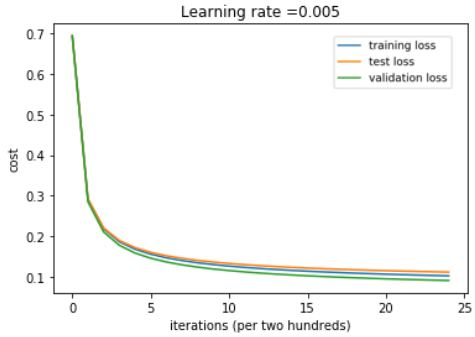
$$E(w) = -\sum_{n=1}^{N}[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)] \tag{1}$$

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

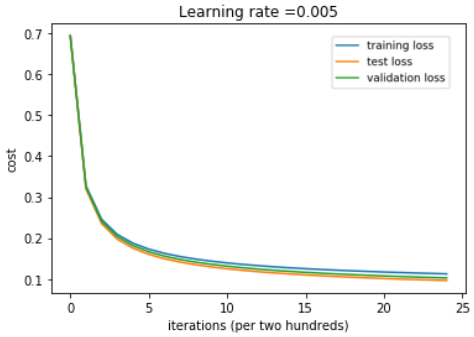
$$-\frac{\partial E^n(w)}{\partial w_j} = \frac{\partial[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)]}{\partial w_j} \tag{2}$$

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

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

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

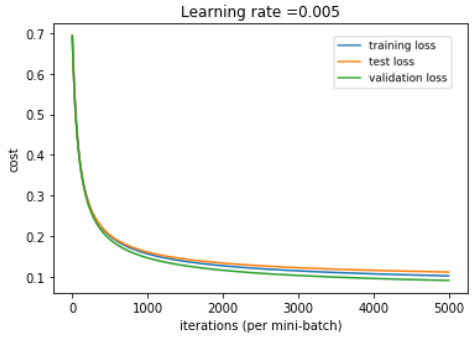
In this work we use the famous MNIST hand written digits database created by Yann LeCun. Each image in the database contains $28 \times 28$ pixels and each pixel has a grayscale intensity, so that the input feature $x \in R^{784}$ after we unroll the 2D image into an 1D vector. To include the bias term, 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.

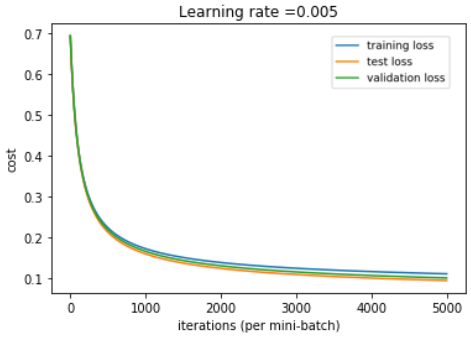
*TODO: Attach code here.*

### 2.4 Experimental Results

**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.

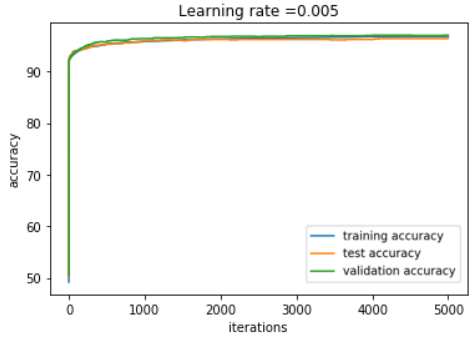



(a) cost function with iterations for mini-batch GD on classifying 2 vs. 3

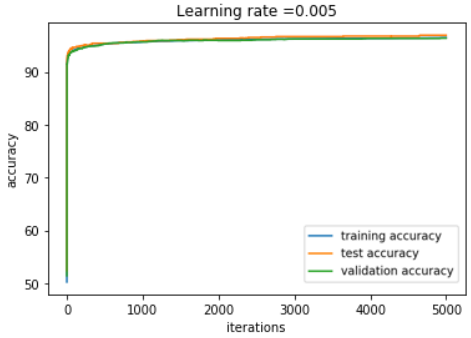



(b) cost function with iterations for mini-batch GD on classifying 2 vs. 8

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

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

**Mini-Batch gradient descent**

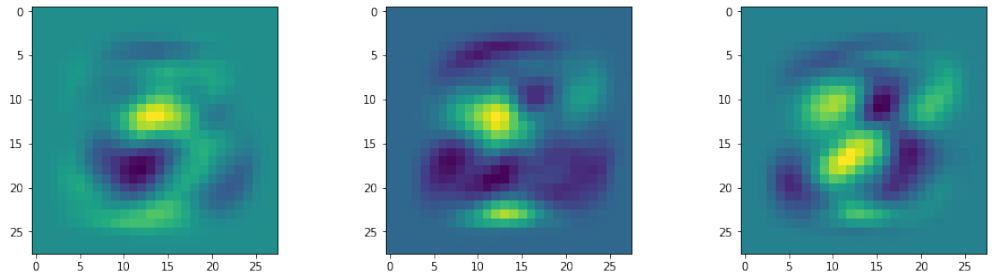
We can use mini-batch gradient descent to speed up learning process. Since the one-step optimization is done on a smaller mini-batch, the cost function will not monotonically decrease as batch gradient descent. Here we change the weights after a mini-batch of roughly $10\%$ of the entire training set. This set of mini-batch size can result in relatively smooth curve of the cost function. Fig. **??**(b) 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**

**Weight Visualization**

We can visualize the weights learned to see what logistic regression learns through the training process.

Computing the difference between these two different classifiers, we can get a new weight matrix visualized in Fig. **??**. This result shows that the new weights can be used to accurately classify digit 3 and digit 8.



(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

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

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

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

$$C(w) = ||w||^2 = \sum_{i,j} w_{i,j}^2 (L2) \tag{7}$$

$$C(w) = |w| = \sum_{i,j} |w_{i,j}| (L1) \tag{8}$$

Thus the gradient of cost function on example $n$ is

$$-\frac{\partial E^n(w)}{\partial w_j} = \begin{cases} (t^n - y^n)x_j^n - 2\lambda w_j, & \text{if } L2 \\ (t^n - y^n)x_j^n - \lambda sign(w_j), & \text{if } L1 \end{cases}$$

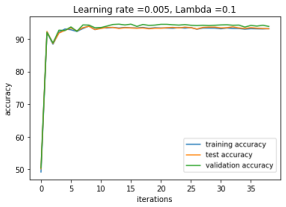
where $sign(w_j)$ is the signature function of $w_j$.

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).

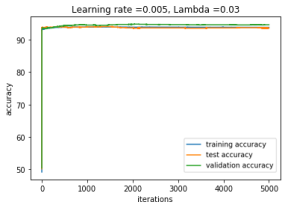
**Tuning the Regularization Parameter**
In order to choose a reasonably good regularization model, we need to tune the regularization parameter and choose the one which results in the best accuracy on the hold-out validation set. Here we try three values for regularization parameter $\lambda$: 0.0001, 0.001 and 0.01. Fig. 5 and Fig. 6 show the results with different $\lambda$ on L1 and L2 regularization.

**Reality Check on the Length of Weight Vector**
Fig. 7 shows the experimental results using L1 and L2 regularization.

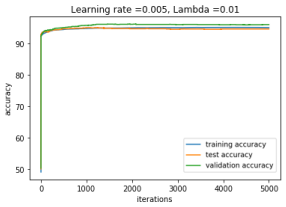
From the figures we can see that stronger regularization will result in smaller weights since the regularization penalizes large weights.



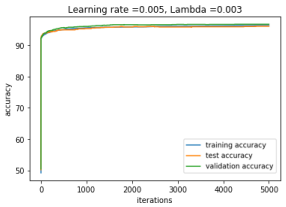

(a) accuracy when lambda = 0.1 using L1 regularization



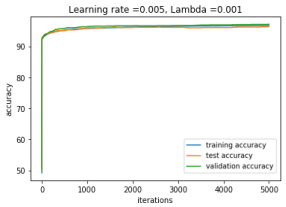

(b) accuracy when lambda = 0.03 using L1 regularization



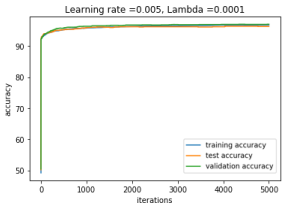

(c) accuracy when lambda = 0.01 using L1 regularization




(d) accuracy when lambda = 0.003 using L1 regularization

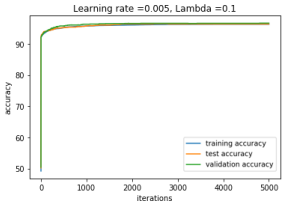



(e) accuracy when lambda = 0.001 using L1 regularization

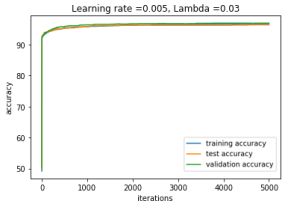



(f) accuracy when lambda = 0.0001 using L1 regularization

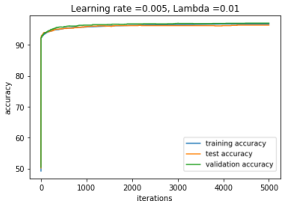
Figure 5: accuracy using L1 regularization



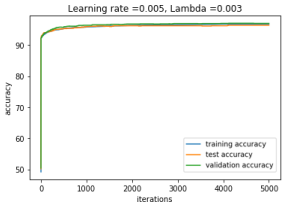

(a) accuracy when lambda = 0.1 using L2 regularization



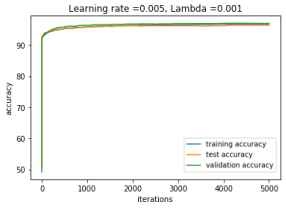

(b) accuracy when lambda = 0.03 using L2 regularization



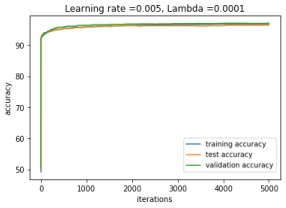

(c) accuracy when lambda = 0.01 using L2 regularization




(d) accuracy when lambda = 0.003 using L2 regularization

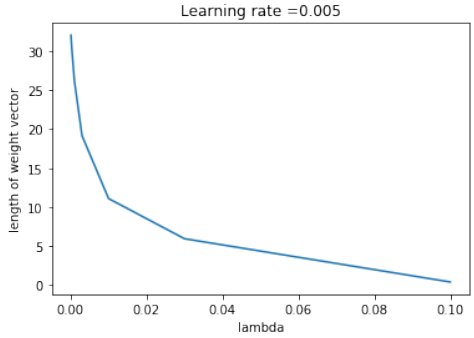



(e) accuracy when lambda = 0.001 using L2 regularization

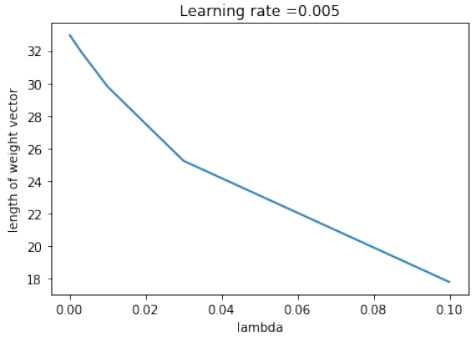



(f) accuracy when lambda = 0.0001 using L2 regularization

Figure 6: accuracy using L2 regularization

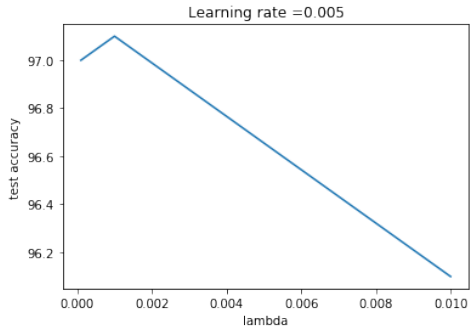



(a) length of weight vector using L1 regularization

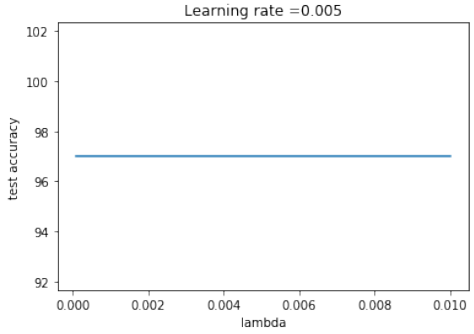



(b) length of weight vector using L2 regularization

Figure 7: Reality check on the length of the weight vector




(a) final test accuracy using L1 regularization




(b) final test accuracy using L2 regularization

Figure 8: final test accuracy using regularization

**Final Test Set Error with Regularization**
We also plot the final test set error with different values of the regularization parameter.

**Weight Visualization with Regularization**
We further visualize the weights learned with our regularized model.

# 3 Softmax Regression via Gradient Descent

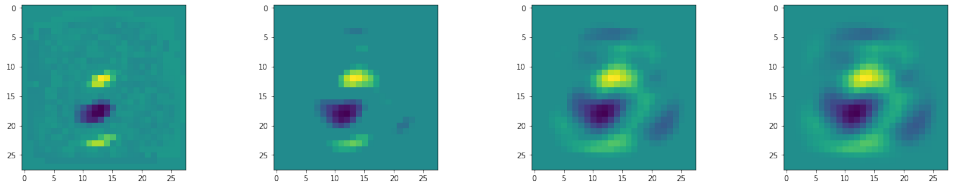
## 3.1 Problem definition

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,

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

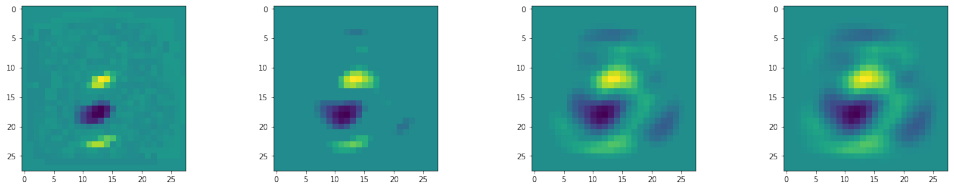


(a) weights when lambda = 0.1 using L1 regularization

(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 9: weights visualization using regularization

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

$$\frac{\partial y_{k'}^n}{\partial a_k^n} = y_{k'}^n \delta_{kk'} - y_{k'}^n y_k^n \tag{13}$$

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,

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

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

$$\frac{\partial C_{L_1}(w)}{\partial w_{ij}} = sign(w_{i,j}) \tag{18}$$

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,

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

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

$$\frac{\partial C_{L_2}(w)}{\partial w_{ij}} = 2w_{i,j} \tag{20}$$

In summary,

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

**Preprossing**: 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]. 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.

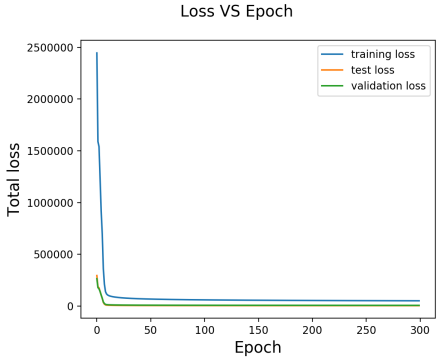
**Experiments settings:** We use standard normal distributions to initilize the weights. We use an initial learning rate $\eta(0)$ = 0.003 and use equation $\eta(t) = \eta(0)/(1 + t/T)$ to anneal the learning rate by reducing it over time. $t$ is used to index the epoch number and $T$ is a metaparameter which is set to be 2. In the experiements, we find that above learning settings work best.

To determine the best type of regurization and the best $\lambda$, we try $L_2$ regularization and $L_1$ regularization seperately. For the $L_2$ regularizartion, we search the best $\lambda$ in the set $\{0.01, 0.001, 0.0001\}$. If the accuracy on the hold-out set decreases for 3 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.

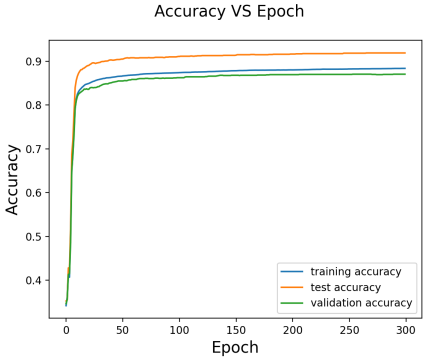
### 3.3 Results

(a) In the experiments, we find that using $L_2$ regularization with $\lambda = 0.01$ obtain the best result on the validation set with an accuracy of $90.45\%$ on the validation set. The accuracy is $92.7\%$ on the test set under such settings.

To further examine the performance of the algorithm, we try other $\lambda$s. We choose $\lambda$ 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 $93.15\%$. If use $L_2$ regularization, the highest accuracy on the validation set is $89.65\%$ with $\lambda = 0.0005$ and the test accuracy is $93.65\%$.




(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 10: The performance of the algorithm over the number of training iterations.

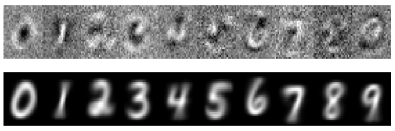


Figure 11: Images of the weights and the average examples. The images in the first row are the images of the weights. The images in the second row are the images of the average examples.

Since using $L_2$ regularization with $\lambda = 0.01$ gives us the highest accuracy on the validation dataset, in the following experiments, we use $L_2$ regularization and fix $\lambda$ to be 0.01.

(b) In this experiement, we use $L_2$ regularization with $\lambda = 0.01$. The figure is shown in Fig 10a. Note that we use the sum of loss of individual data rather than the average one.

(c) In this experiement, we use $L_2$ regularization with $\lambda = 0.01$. The figure is shown in Fig 10b.

(d) We plot the results in Fig 11.

### 3.4 Discussion

From Fig 10a and Fig 10b and we can see that the accuracy and loss function converges quickly and then becomes smooth. The plausible reason is that the learining rates are reasonble enough to make the loss function to converge quickly to a local minimum. Setting the learning rate too small initially may let the algorihtm too slow to converge or just being 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 11 we can see that the image of the weight and the corresponding image of the average digit is almost the same. The reason is that we classify the images based on the inner product of the pixels with the weights. 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 $98\%$ 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 Code

### 5.1 Logistic regression

### 5.2 Softmax regression

```
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, t

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) # 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

delta = one_hot_training_labels - pred_y # (18000, 10)
grads = -np.dot(training_images.T, delta) + lambd*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) # 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

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 # (18000, 10)
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

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)

a = np.matmul(test_images, best_validation_weights_L1) # 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 if using L1 " + str(test_accuracy)

a = np.matmul(test_images, best_validation_weights_L2) # 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 if using L2 " + str(test_accuracy)


def train(training_images, test_images, validation_images, training_labels, test_labe

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

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

for i in range(epoch):
print "epoch " + str(i)
# 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
grads = -np.dot(training_images.T, delta) + lambd*weights
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)
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)
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
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)
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('Total 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
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# (18000, 785)
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 = 300
inital_step_size = 0.003
T = 2.0

lambds_set_1 = [0.01, 0.001, 0.0001]
lambds_set_2 = [0.05, 0.005, 0.0005]
lambd = 0.01
regularization_types = ["L1"]

classes = 10
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_l

#Training based on the best hyperparameters
#train(training_images, test_images, validation_images, training_labels, test_labels
```