# 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

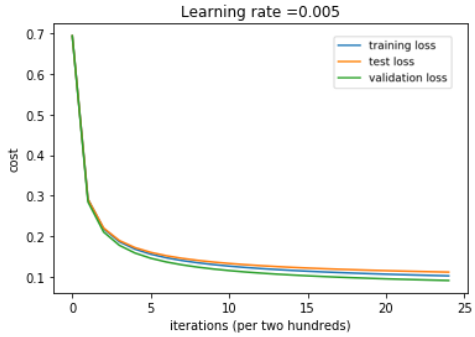
$$E(w) = -\frac{1}{N}\sum_{n=1}^{N}[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)] \tag{1}$$

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

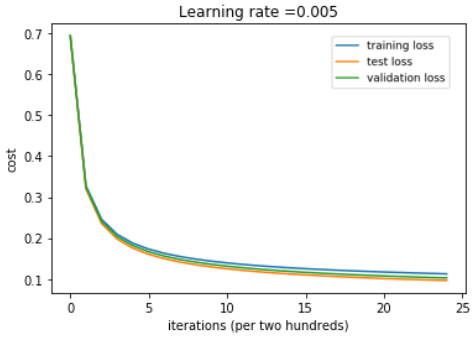
$$-\frac{\partial E^n(w)}{\partial w_j} = \frac{1}{N}\frac{\partial[t^n \ln(y^n) + (1-t^n)\ln(1-y^n)]}{\partial w_j} \tag{2}$$

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 $28 \times 28$ 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.

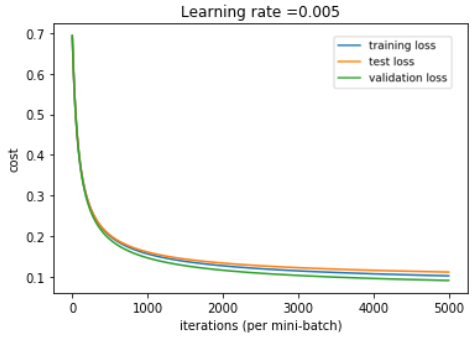
The following code shows how we can choose the example images corresponding to two specific digit classes. For the rest of the report, we choose two binary classification problems: 2 vs. 3 and 2 vs. 8.

### 2.4 Experimental Results and Discussion

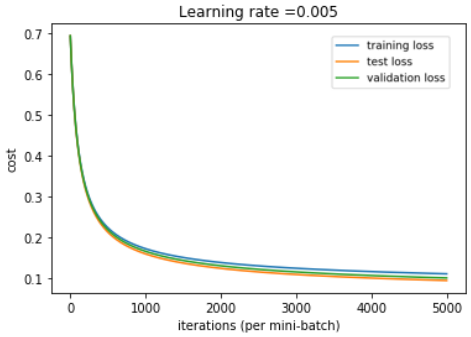
**Batch gradient descent**

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

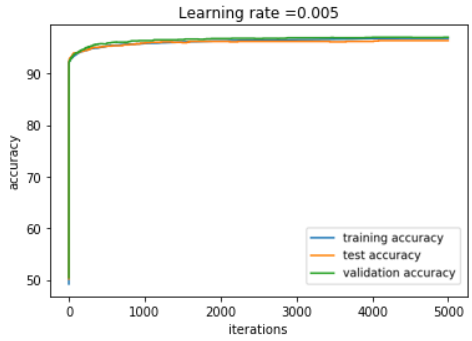



(a) Cost function with iterations for mini-batch GD on classifying 2 vs. 3

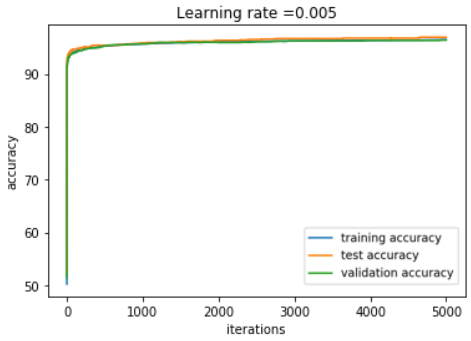



(b) Cost function with iterations for mini-batch GD on classifying 2 vs. 8

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.

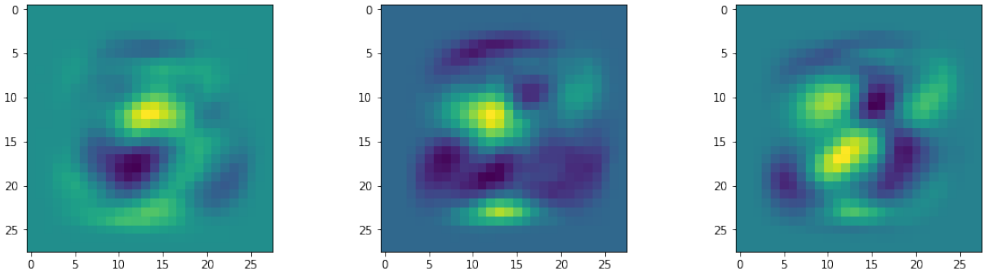
**Mini-Batch gradient descent**

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**

**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 of input $X$ and weights, the angle between these two vectors in the feature space should be zero so



(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

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

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

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

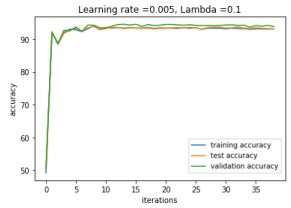
Thus the gradient of cost function on example $n$ is

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

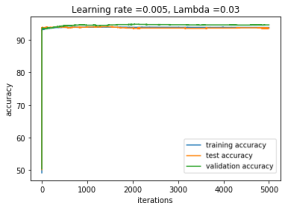
where $sign(w_j)$ is the signature function of $w_j$.

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).

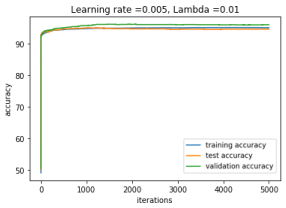
**Tuning the Regularization Parameter**
In order to choose a reasonably good regularization model, we need to tune the regularization parameter and choose the one which results in the best accuracy on the hold-out validation set. Here we try three values for regularization parameter $\lambda$: 0.0001,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



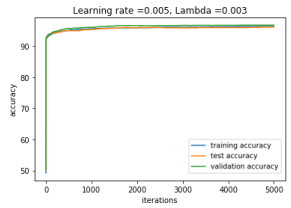

(a) Accuracy when lambda = 0.1 using L1 regularization



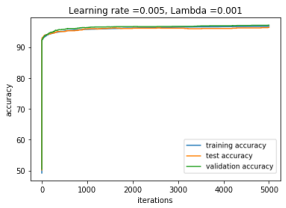

(b) Accuracy when lambda = 0.03 using L1 regularization



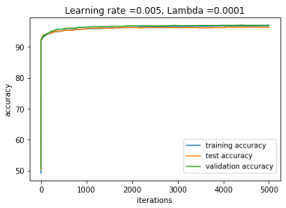

(c) Accuracy when lambda = 0.01 using L1 regularization




(d) Accuracy when lambda = 0.003 using L1 regularization

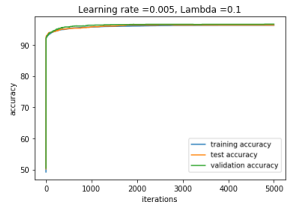



(e) Accuracy when lambda = 0.001 using L1 regularization

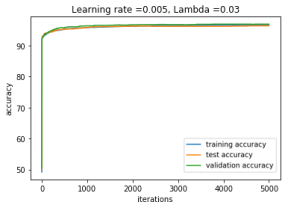



(f) Accuracy when lambda = 0.0001 using L1 regularization

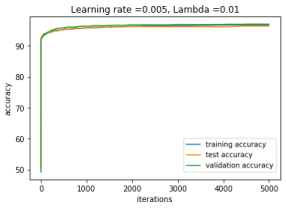
Figure 5: Accuracy using L1 regularization



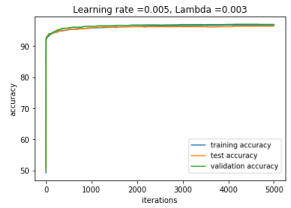

(a) Accuracy when lambda = 0.1 using L2 regularization



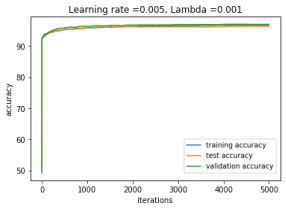

(b) Accuracy when lambda = 0.03 using L2 regularization



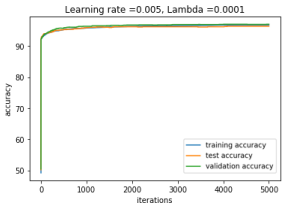

(c) Accuracy when lambda = 0.01 using L2 regularization




(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

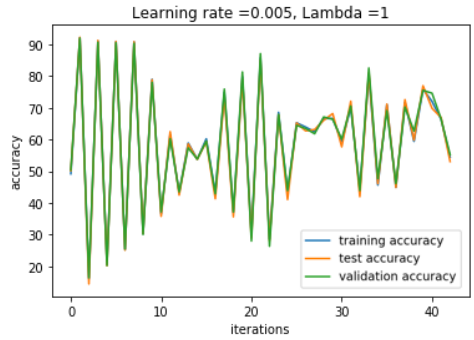
Figure 6: Accuracy using L2 regularization

as soon as we do not set $\lambda$ 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.

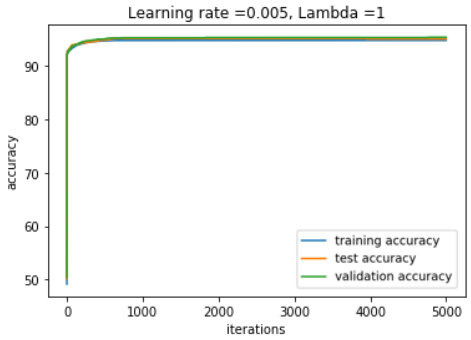
If we set lambda very large as shown in Fig. 7, we can see that L1 regularization will be over strong and cause the model fail to converge and may experience underfit problem. L2 regularization is more robust against large $\lambda$ compared to L1 regularization.

**Reality Check on the Length of Weight Vector**

Fig. 8 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. This is consistent with the mathematics of our cost function since if we use a very large $\lambda$,



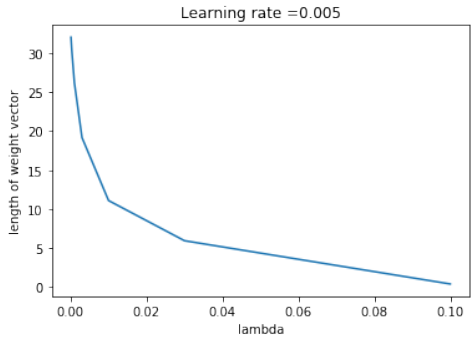

(a) Accuracy when lambda = 1 using L1 regularization



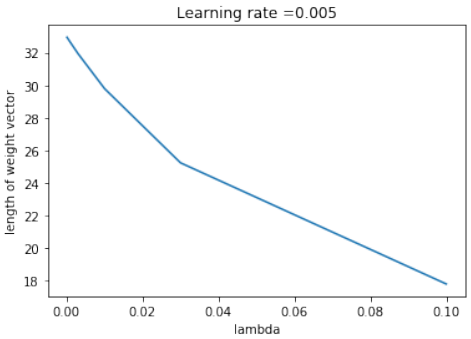

(b) Accuracy when lambda = 1 using L2 regularization

Figure 7: Setting up lambda to be very large values

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

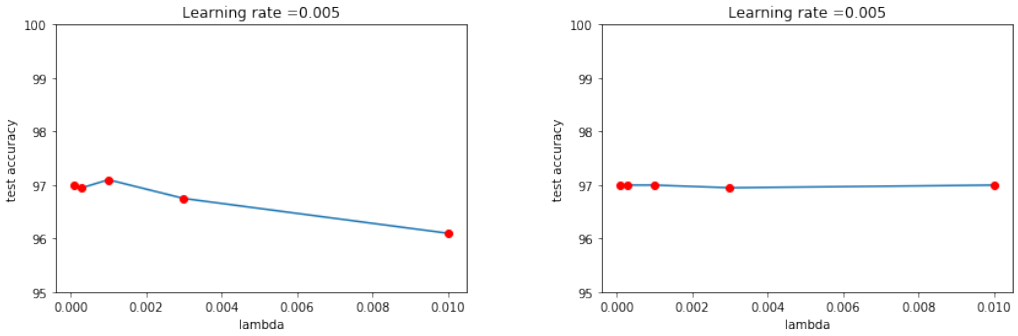
Figure 8: Reality check on the length of the weight vector

**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 $\lambda$ is relatively small, the test accuracy will increase as we increase $\lambda$. 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 $\lambda$ 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 $\lambda$ 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 $\lambda$ compared with L1 regularization.

**Weight Visualization with Regularization**

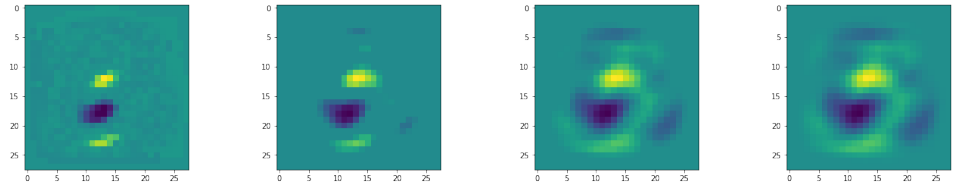
We further visualize the weights learned with our regularized model, which are shown in Fig. 10. From the results we can see that when $\lambda$ increases, the learned parameters (weights) become simpler and cannot represent the data very well since we cannot see the structure of the digits anymore under the case with too large $\lambda$. But when $\lambda$ 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 sensitive as we change $\lambda$ compared with L1 regularization since L2 regularization can still make the model to learn good representations even if $\lambda$ is large.




(a) Final test accuracy using L1 regularization

(b) Final test accuracy using L2 regularization

Figure 9: Final test accuracy using regularization

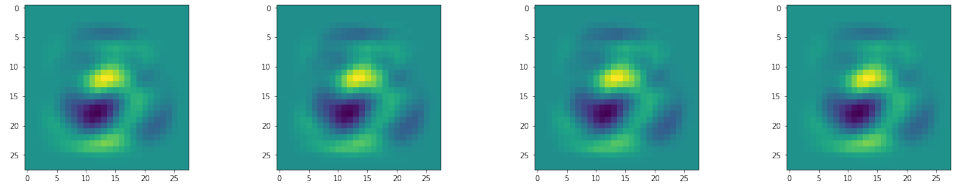


(a) Weights when lambda = 0.1 using L1 regularization

(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

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 $\lambda$ 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,

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

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

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

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 $\lambda$, 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 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.

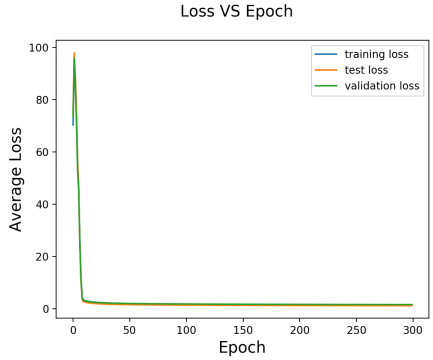
### 3.3 Results

(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.

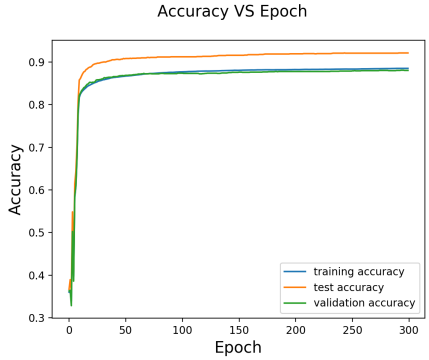
To further examine the performance of the algorithm, we try other $\lambda$s. We choose $\lambda$ in the set $\{0.05, 0.005, 0.0005\}$ and use $L_1$ regularization and $L_2$ regularization. If use $L_1$ regularization, the highest accuracy on the validation set is $88.45\%$ with $\lambda = 0.0005$ and the test accuracy is $92.9\%$. 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 $\lambda$ 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 $98\%$ 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

## 6.1 Logistic regression

## 6.2 Softmax regression