MNIST Image Classification Project

Abstract

In this project, given MNIST dataset, we modeled it as a multiclass classification problem where we implemented different classification algorithms. In addition, we also conducted feature visualization and network diagnosis using PCA and t-SNE. Six different models including Logistic Regression (Softmax), Logistic Regression with Ridge loss, Logistic Regression with Lasso loss, simple-NN, CNN and SVM were studied and their performances were measured and compared. Among which, CNN performed the best.

I. Data

1.1 Dataset Introduction

MNIST dataset is a handwritten digits dataset which has a training set of 60,000 examples and a test set of 10,000 examples. It's a subset of a larger set available from NIST. In the dataset, the digits have been size-normalized and centered in a fixed-size image (28x28 pixel box) by computing the center of mass of the pixels.

1.2 Data Preprocessing

MNIST data is stored in a simple file format designed for storing vectors and multidimensional matrices. However, when we implement classification methods, typically the `numpy` array is the easiest way for follow-up model construction, thus, we need to read in 4 files: `train-images-idx3-ubyte, train-labels-idx1-ubyte, t10k-images-idx3-ubyte, t10k-labels-idx1-ubyte` and transform them into operable format.

A)

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

B)

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	10000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

C)

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	10000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

D)

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

Fig. 1 Overview of raw MNIST dataset

A) Training set label file; B) Training set image file; C) Test set label file; D) Test set image file

From Fig.1, we can see that in the image files, pixels are organized row-wise. Values range from 0 to 255, where 0 means background (white), 255 means foreground (black). In the label files, the labels values are 0 to 9.

In `util.py`, we defined `read_image_labels` and `load_dataset` function to load data from `idx3-ubyte` and `idx1-ubyte` file. Fig.2 shows some random training and test images retrieved from raw dataset.

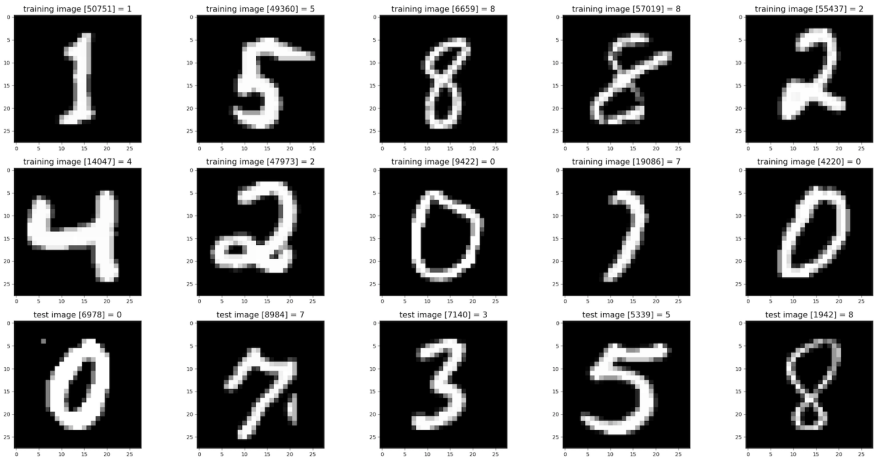

Fig. 2 Random training and test images

II. Image Classification

In this part, we implemented 6 different image classification methods: Logistic Regression (Softmax), Logistic Regression with Ridge loss, Logistic Regression with Lasso loss, simple-NN, CNN and SVM.

2.1 Logistic Regression

2.1.1 Multinomial Logistic Regression

Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes. It allows us to handle $y(i) \in \{1, \ldots, K\}$ where K is the number of classes. In the model, we assume that the conditional distribution of y given x is given by

$$\begin{aligned} p(y = i|x;\theta) &= \phi_i \\ &= \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}} \\ &= \frac{e^{\theta_i^T x}}{\sum_{j=1}^{k} e^{\theta_j^T x}} \end{aligned} \tag{1}$$

In `Logreg.py`, we built the softmax regression model from scratch without using any built-in package other than `numpy` and `matplotlib`. In this part and following two sections involving regularization, due to the long training time using complete dataset, we randomly sample $10k$ examples from the $60k$ training samples. For the first $2k$ rows, we used them as validation set. The rest $8k$ rows are used as training set. The key points in building softmax regression model is to derive the cross-entropy loss function which is given by

$$J(\theta) = -\left[\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log h_\theta(x^{(i)})\right]$$
$$= -\left[\sum_{i=1}^{m}\sum_{k=0}^{1} 1\left\{y^{(i)}=k\right\}\log P(y^{(i)}=k|x^{(i)};\theta)\right] \quad (2)$$

Thus, we wrote `one_hot_labels` and `softmax` function to assist in deriving the loss function of softmax regression. By setting the learning rate $\alpha = 0.01$, threshold for determining convergene $eps = 1e-4$, we iterated to minimize the loss function using SGD. Finally, we can get the model accuracy of 0.8905.

Table 1 Training results of Softmax Regression

Dataset	Training	Validation	Test
Accuracy	0.9511	0.9975	0.8905

2.1.2 Multinomial Logistic Regression with Ridge loss

Recall that the OLS estimates $\beta_0,\ \beta_1$ ... by minimizing

$$\text{RSS} = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2. \quad (3)$$

Similar to OLS, Ridge regression estimates $\beta_0,\ \beta_1$ ... by minimizing

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = \text{RSS} + \lambda\sum_{j=1}^{p}\beta_j^2 \quad (4)$$

The major difference lies in the regularization term, $\lambda\sum_j {\beta_j}^2$, is called a shrinkage penalty, which is small when $\beta_0,\ \beta_1$ ... are close to zero. The tuning parameter λ serves to control the relative impact of these two terms on the regression coefficient estimates. Here we set $\lambda = 1e-4$. With Ridge loss, we improve the model accuracy from 0.8905 to 0.8914.

Table 2 Training results of Softmax Regression with Ridge loss

Dataset	Training	Validation	Test
Accuracy	0.9483	0.9965	0.8914

2.1.3 Multinomial Logistic Regression with Lasso loss

Similar as Ridge regression, the Lasso regression also adds a penalty term which uses L1 norm instead of L2 norm. Here we also set $\lambda = 1e-4$. With Ridge loss, we improve the model accuracy from 0.8905 to 0.8918.

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| = \text{RSS} + \lambda\sum_{j=1}^{p}|\beta_j| \quad (5)$$

Table 3 Training results of Softmax Regression with Lasso loss

Dataset	Training	Validation	Test
Accuracy	0.9496	0.9990	0.8918

2.1.4 Basic Conclusion for Logistic Regression

According to results we got in section 2.1.1 – 2.1.3, we can see that with regularization, the test accuracy is improved to some degree. The Ridge and Lasso regularization are functioned as increasing the bias if our model overfits and suffers from high variance. Nevertheless, to much bias can lead to underfitting.

2.2 Neural Network

2.2.1 Simple-NN

As we mentioned before, in previous three sections about logistic regression with/without regularization, we used a subset data of *10k* size. Beginning from this part, we used the complete MNIST dataset of *60K* size. For the first *10k* rows, we used them as validation set. The rest *50k* rows are used as training set.

In this part, we built a simple 2-layer neural network with one hidden layer and one output layer from scratch. From the input layer to hidden layer, sigmoid function is used as activation function. From the hidden layer to output layer, softmax function is used as activation function, which can be expressed as

$$z_1 = W_1 x + b_1$$
$$a_1 = sigmoid(z_1)$$
$$z_2 = W_2 x + b_2$$
$$a_2 = \hat{y} = softmax(z_2)$$

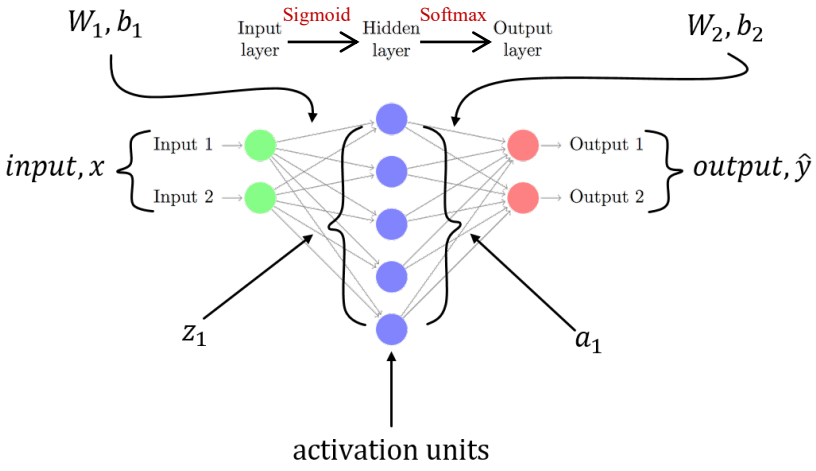

Fig. 3 A simple 2-layer neural network

The basic idea of constructing neural network consists of two elements: forward propagation and backward propagation. In forward propagation, the input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes to the successive layer. In backward propagation, we allow the information to go back from the cost backward through the network in order to compute the gradient and update the parameters in the neural network.

In `nn.py`, we implemented this neural network by minimizing the cross-entropy loss using mini-batch gradient descent. Below are some of the training parameters.

Table 4 Training parameters of simple-NN

Parameters	#Hidden units	Batch size	#Epochs	Learning rate	Lambda
Value	300	1000	30	5	0.001

In addition to the baseline model, we added a regularization term to the cross-entropy loss to get the regularized model. Fig 4 shows the loss vs epoch and accuracy vs epoch in baseline and regularized model.

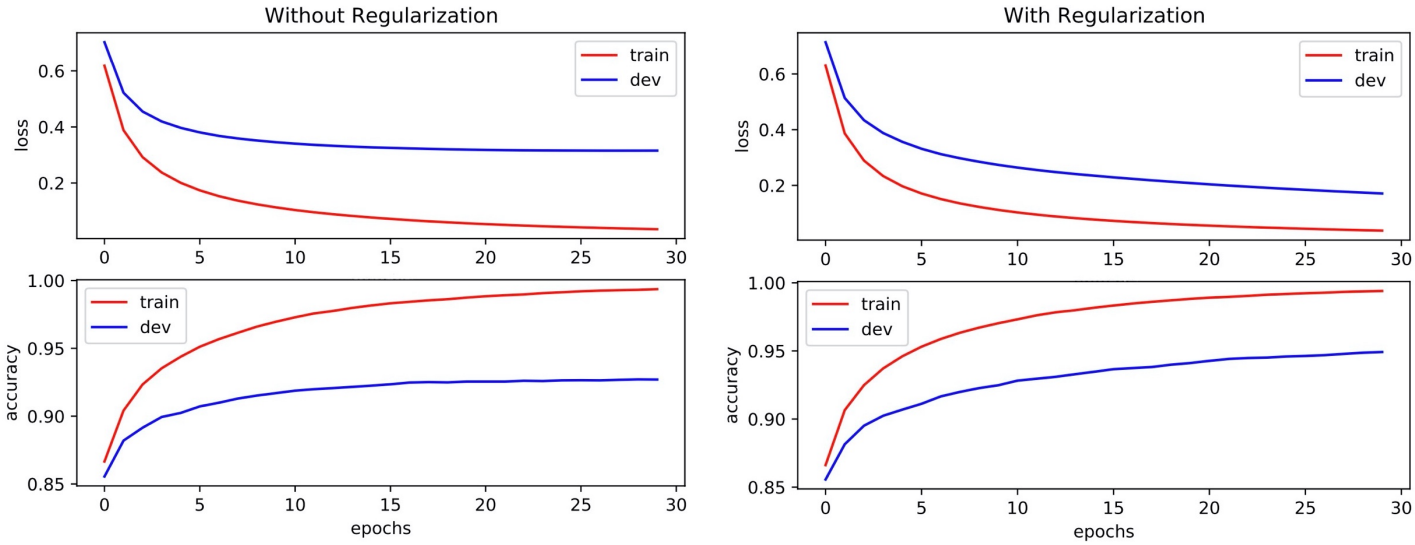

Fig. 4 Loss vs epoch, Accuracy vs epoch

Left: without regularization; Right: with regulatization

Table 5 Comparison of baseline and regularized model

Model	Baseline	Regularized
Accuracy	0.9310	0.9493

Apparently from table 4, the regularized model improved the model accuracy from 0.9310 to 0.9493. We can interpret the better performance by looking at fig 4. Without regularization, the overfitting is apparent as there is big difference between the performances on the the training and validation data in terms of both cross-entropy loss and accuracy. In contrast, with regularization, the performance on validation data follows that on training data closely, does much better than without regularization.

2.2.2 CNN

1. Build Model

1) Different Model Architecture

In this part, we used `keras` package in `tensorflow` to build two CNN models. These two model network architectures are generated by `keras` as fig 5 shows.

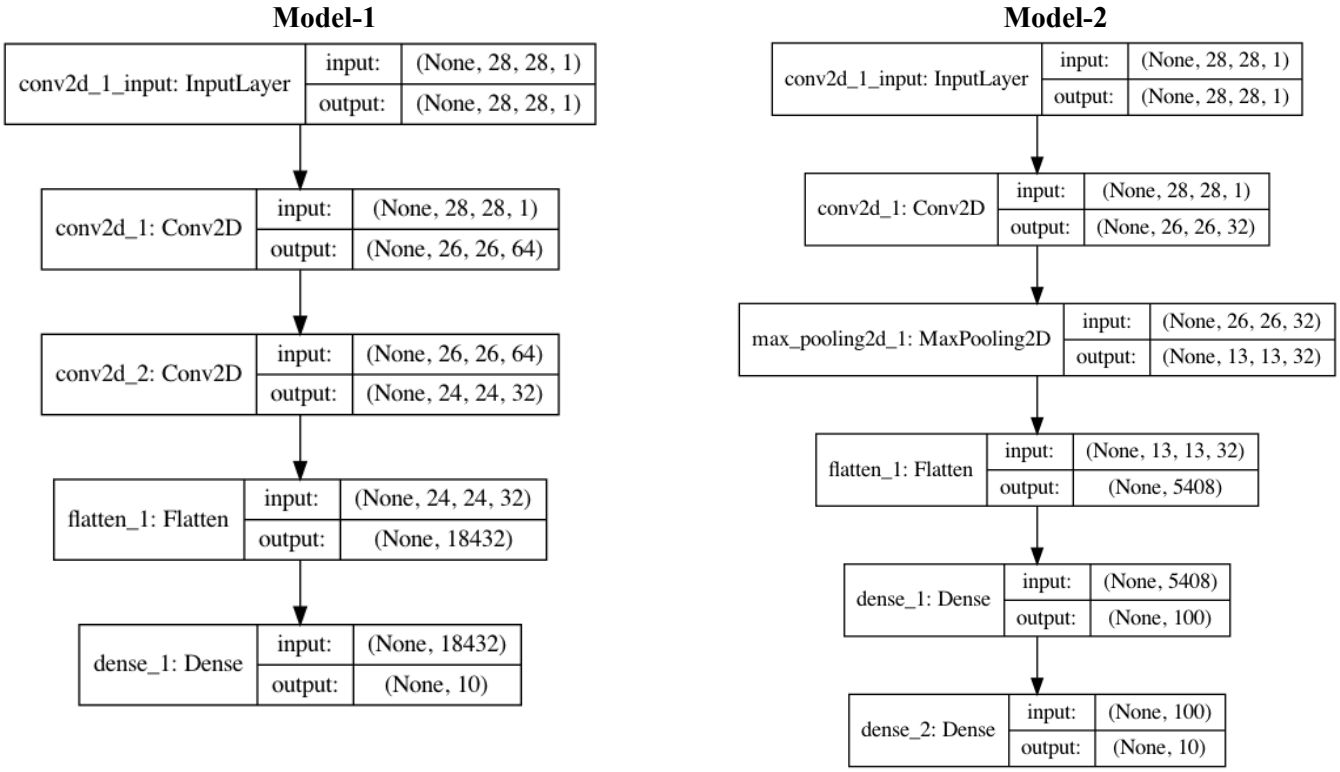

Fig. 5 CNN model-1 and model-2 architecture

Left: CNN model-1; Right: CNN model-2

For the first model, we used Conv2D as the first two layers to deal with our input images which are seen as 2-dimensional matrices. We designed the number of nodes in each layer to be 64 in the first one and 32 in the second one. In the meantime, the kernel size 3 represents for the 3x3 filter matrix. And we used ReLU as the activation function for these two layers. A flatten layer used to reshape and build connection between convolution and dense layer is implemented before the output layer, which is a dense layer with 10 nodes using softmax as activation function.

Similar as the first one, the second CNN model is just a little more complicated because we added more layers and tuned some of the parameters to see the effect. For the first two layers, we still used Conv2D but we changed the nodes in the second layer from 32 to 64. Also, before the flatten layer, we added a MaxPooling2D layer to down sample the input. Before the final dense layer same as model 1, we added another 100-node dense layer to interpret the features.

In this part, we used the SGD with a learning rate of 0.01 and a momentum of 0.9 as the optimizer for both model 1 and 2.

2) Different Optimizer

In the training process of CNN model 2, we also changed the optimizer from SGD to Adam to see if the optimizer has an impact on the result and model performance.

2. Evaluate model

After model construction, we used 5-Fold cross validation method to split training and validation set (K=5). First we split dataset into 5 consecutive folds with shuffling. Each fold is then used once as a validation while the 4 remaining folds form the training set. The value of K is appropriate to avoid an overwhelmingly long running time as well as a non-repeatable evaluation.

3. Model Analysis Result

1) Different Model Architecture

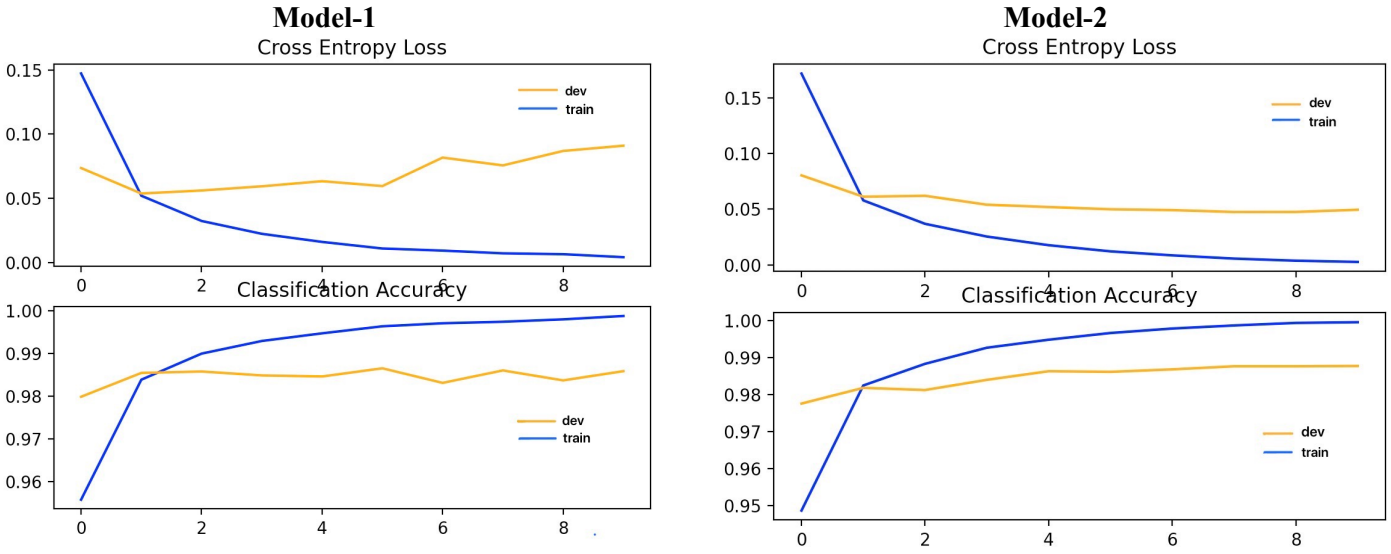

Fig. 6 CE loss vs epoch, Classification Accuracy vs epoch (optimizer: SGD)

Left: CNN model-1; Right: CNN model-2

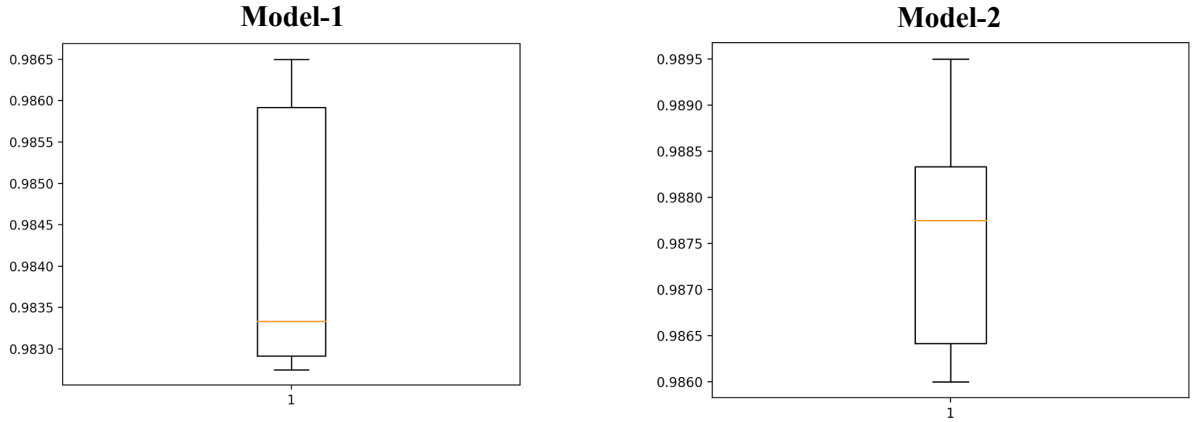

Fig. 7 Boxplot of test accuracy using k-fold (k=5) (optimizer: SGD)

Left: CNN model-1; Right: CNN model-2

Table 6 Test accuracy using k-fold (k=5) (optimizer: SGD)

	K-Fold	1	2	3	4	5	Mean
Model 1	Accuracy	0.9828	0.9833	0.9829	0.9865	0.9859	0.9843
Model 2		0.9860	0.9883	0.9864	0.9895	0.9878	0.9876

We can see that the mean accuracy of model 2 is slightly higher but there is no apparent difference between two models' performance in our case. However, we noticed that model 2 can train model in significantly shorter time period than model 1. Although two models have same amount of convolutional layers, model 2 has 'fatter' convolutional layer with more units, which speed up its training process. What's more, the MaxPooling2D layer also reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation by down-sampling the input. Thus, model 2's faster training speed can be well explained.

2) Different Optimizer

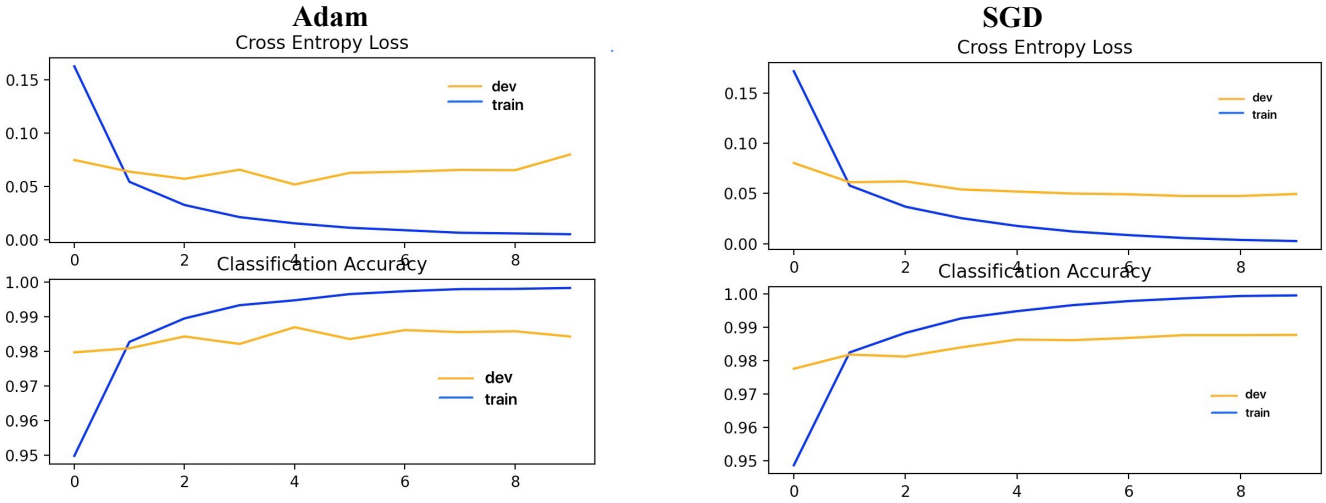

Fig. 8 CNN model-2 CE loss vs epoch, Classification Accuracy vs epoch using different optimizers

Left: Adam; Right: SGD

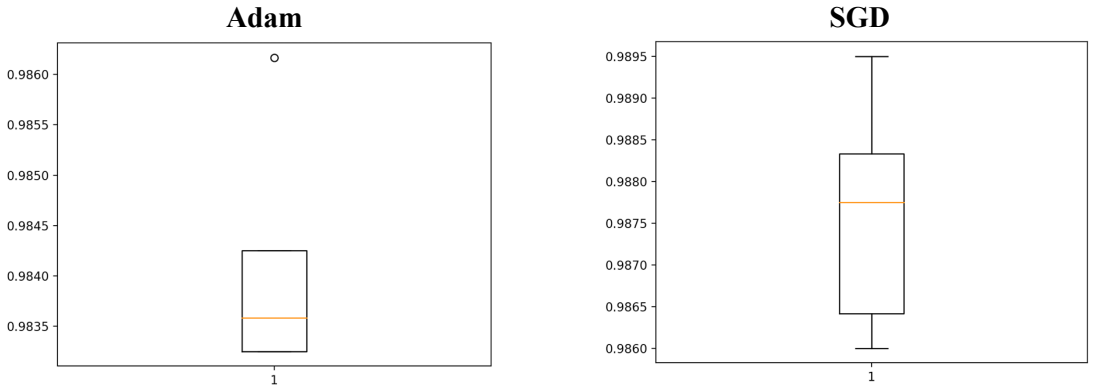

Fig. 9 Boxplot of CNN model-2 test accuracy using k-fold (k=5) with different optimizers

Left: Adam; Right: SGD

Table 7 CNN model-2 Test accuracy using k-fold (k=5) with different optimizers

	K-Fold	1	2	3	4	5	
Adam	Accuracy	0.9833	0.9833	0.9836	0.9862	0.9843	0.9841
SGD		0.9860	0.9883	0.9864	0.9895	0.9878	0.9876

We can see that there is no significant difference in accuracy using different optimizers in our case. SGD result is just slightly higher. But we do need to consider the choice of optimizers when building CNN. Taking Adam and SGD as an example, the latter one requires a relatively low memory requirements but the former one usually works better with little tuning of hyperparameters.

2.3 SVM

SVM intends to construct a hyperplane or set of hyperplanes that has the largest distance to the nearest training data point of any class to reduce the classifier's generalization error. In other words, the main object of SVM is to maximize the geometric margin.

In this part, we just used the built-in model of SVM in `sklearn`.

Table 8 Training results of Softmax Regression with Lasso loss

Dataset	Training	Test
Accuracy	1.0	0.9837

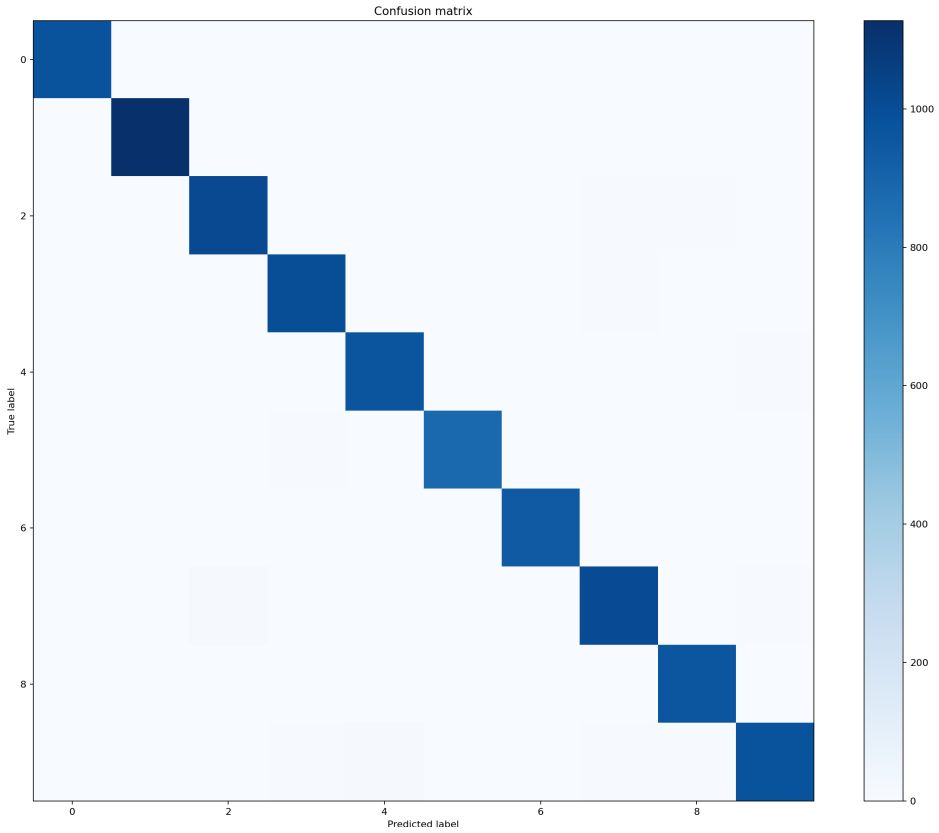

Fig. 10 Dimensionality reduction using PCA

Fig 10 shows the confusion matrix. We know that the diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. The higher the diagonal values of the confusion matrix the better, indicating many correct predictions. For this SVM model, all of the diagonal values are beyond 850, which corresponds to the high accuracy of the model.

2.4 Model Comparison

According to the previous experiment with 6 different algorithms (9 models in total), we summarized the model performance in table 9.

Table 9 Different model performance

Model	Accuracy (test)
Softmax	0.8905
Softmax + Ridge	0.8914
Softmax + Lasso	0.8918
Simple-NN-baseline	0.9310
Simple-NN-regularized	0.9493
CNN-model1-SGD	0.9843
CNN-model2-SGD	0.9876
CNN-model2-Adam	0.9841
SVM	0.9837

All of the models we built have a relatively high accuracy, among which, CNN model performs the best, and the Softmax regression without regularization performs the worst. If we take training time into consideration, simple-NN has the quickest training speed.

III. Feature Visualization

3.1 PCA

Principal Components analysis (PCA), which tries to identify the subspace in which the data approximately lies, is extensionally used for dimensionality reduction for the visualization of high dimensional data. The main object of PCA is to provide a minimum number of variables that keeps the maximum amount of variation or information about how the original data is distributed using the correlation between some dimensions.

From fig 11, the first two components account for about 25% of the variation in the entire dataset. If we do scatter plot to show the first two components as fig 12 shows, we can see that they do contain some information especially for digits like 0.0. However, it's not enough to separate different classes apart clearly. Thus, next we will use another technique to see if it can provide us with better feature visualization results.

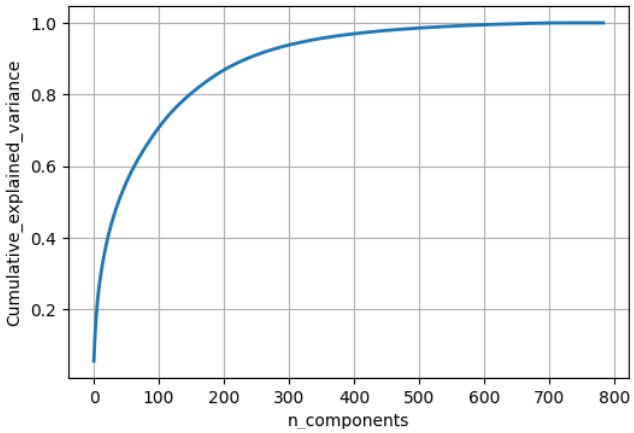

Fig. 11 Dimensionality reduction using PCA

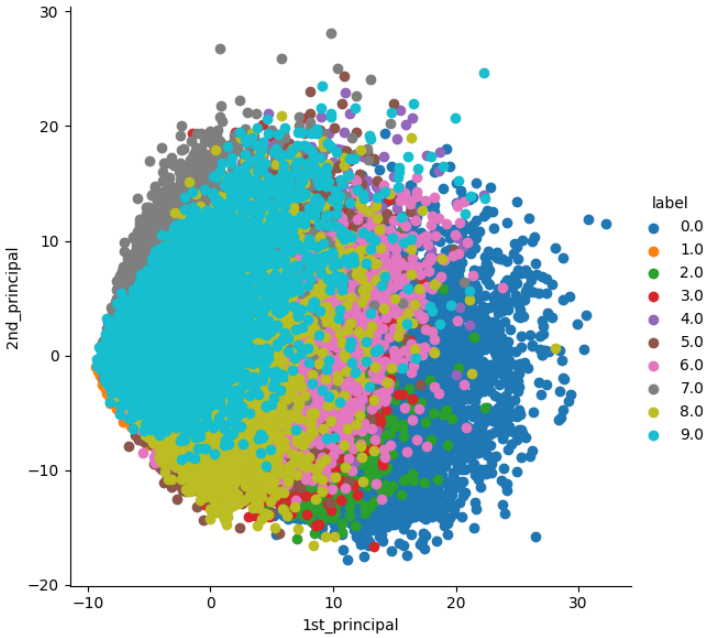

Fig. 12 PCA using `sklearn`

3.2 t-SNE

t-SNE stands for t-Distributed Stochastic Neighbor Embedding, which is another method for dimensionality reduction. Its main goal is to minimize the divergence between two distributions, a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding.

Due to the fact that the entire dataset needs very long time for t-SNE to process, we used a subset data of *10k* size. As fig 13 shows, the digits are clearly clustered in their own sub groups. It would be easier to assign new points to its label now.

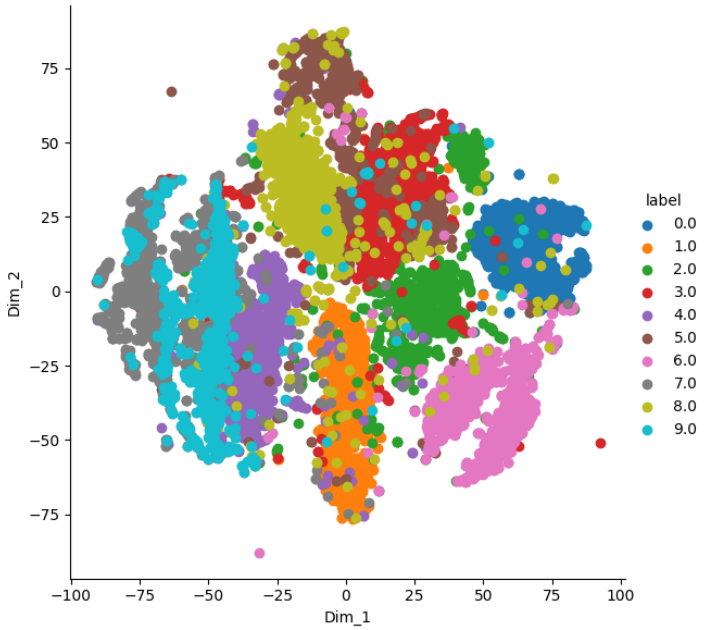

Fig. 13 t-SNE using `sklearn`

IV. Conclusion

In project 1, we implemented MNIST digit image classification using six different algorithms including Logistic Regression (Softmax), Logistic Regression with Ridge loss, Logistic Regression with Lasso loss, simple-NN, CNN and SVM. We built 9 models in total with some parameter tuning and regularization. Among all the models, CNN performs the best with highest accuracy while Softmax regression without regularization performs the worst. In general, all these algorithms can fulfill our requirements for model accuracy. However, some of the models can take 2-3 hours to train each time. Thus, we do need to consider memory and time cost for training different models, from which aspect, the simple-NN is the best. In the third part, we also conducted feature visualization using PCA and t-SNE. For future work, we think that the optimal model doesn't only require the stable and outstanding classification ability but also the capacity to avoid suffering from the accuracy-training cost tradeoff too much.