MNIST Dataset Classification

Rishab Gulati
A53219165
r1gulati@ucsd.edu

Abstract

In this project report I present the experiments that I ran with different classifiers over the MNIST dataset in order to classify the images. I've used Convolution Neural Network, K-Nearest Neighbor and Support Vector Machines in order to classify the images to their respective classes and to analyze which classifier gives the best results over the dataset.

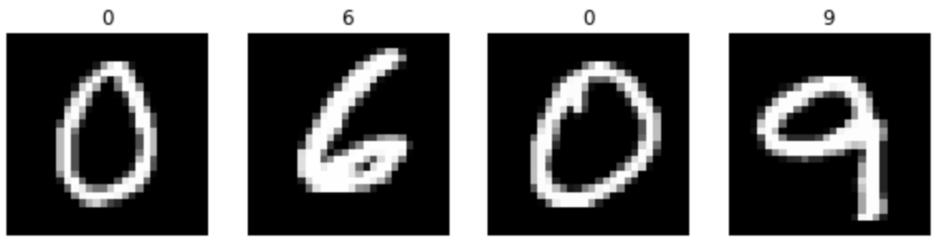
1 Dataset and Preprocessing

The dataset used in this project is the MNIST dataset which contains about 60,000 train images and about 10,000 test images. Each image is a 28x28x1 grayscale image.
For the CNN, I loaded the dataset from the in-built functionality in PyTorch using torchvision. The mean and standard deviation over all the images in the dataset is found to be 0.1307 and 0.3081 respectively. So each image is normalized by subtracting the mean and dividing by the standard deviation.

For KNN and SVM, I used the function "fetch_mldata" from the sklearn library which loads the MNIST dataset.

A few of the images from the train data is shown in figure 1

Figure 1: Train Data Sample. Label of each image is provided above it.



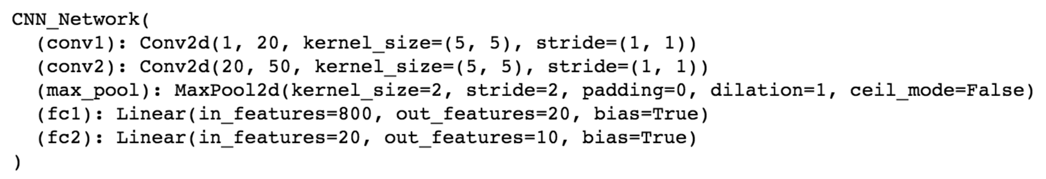
2 Convolution Neural Network

The Convolution Neural Network or CNN is one of the most common ways to analyze image data. Usually it's a combination of convolution layers, max pooling layers and fully connected layers that is used in order to analyze image data and predict their classes. For this part of the project, I build my own CNN architecture that analyzed the digits of the MNIST and classified it to one of the 10 available image classes.
One main reason why I preferred writing a smaller network than perform transfer learning was because the pre-trained networks like VGG16, resnet etc. have a lot of layers. This is primarily

because they are trained over ImageNet data which may as well be one of the largest collection of RGB images and hence requires a large number of layers to detect the intricacies in the images. But here we have a single channel grayscale images that do not require enormously many filters. So even with smaller amount of filters the network is able to analyze the images properly.

For this network I added 2 Convolution Blocks, where each Convolution block had a Convolution Layer with ReLU activation followed by a MaxPool Layer, which were then connected to a Fully Connected Network with 20 nodes which was further connected to another Fully Connected Network with nodes equivalent to number of classes, i.e 10 . The different layers used in the network are given in Figure 2. I experimented by running the network with 3 different optimizers the details of which are given in Section 2.1

Figure 2: Architecture of CNN used on MNIST



```
CNN_Network(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))
  (max_pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=800, out_features=20, bias=True)
  (fc2): Linear(in_features=20, out_features=10, bias=True)
)
```

2.1 Experiments with different Optimizers

2.1.1 Stochastic Gradient Descent with Cross Entropy Loss

Ran the network with Stochastic Gradient Descent(SGD) as the optimizer and Cross Entropy Loss as the loss function. The loss per iteration is given in Figure 3

The loss per epoch on training data is plotted in Figure 4 . **Note that iterations refer to epoch in the plots.**

This network gave 99.09% Accuracy on test data

2.1.2 Adam with Cross Entropy Loss

Reinitialized the network and ran it again with Adam as the optimizer and Cross Entropy Loss as the loss function. The loss per iteration is given in Figure 5

The loss per epoch on training data is plotted in Figure 6 . **Note that iterations refer to epoch in the plots.**

This network gave 98.89% Accuracy on test data

2.1.3 RMSProp and Cross Entropy

The third optimizer I tried using was RMSProp along with Cross Entropy Loss. The results given by this optimizer were disastrous compared to Adam and SGD. The loss per iteration is given in Figure 7

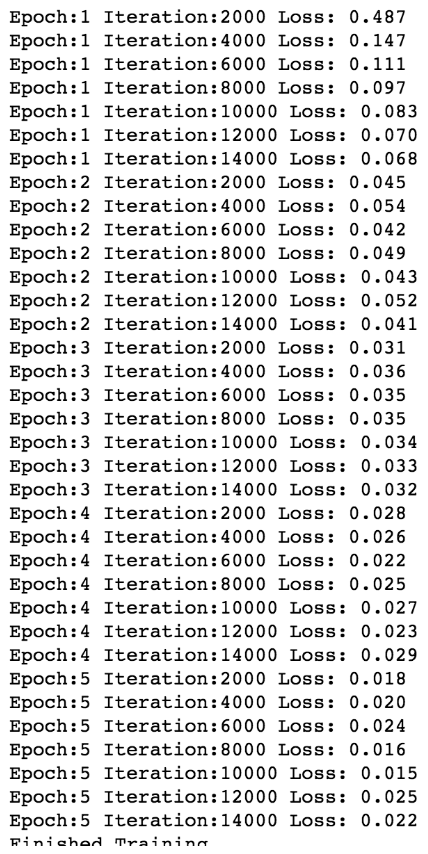
The loss per epoch on training data is plotted in Figure 8 . **Note that iterations refer to epoch in the plots.**

The summary of all the Optimizers are given in Table 1. From this table we can see that SGD provides the best results on test data.

2.2 Analyzing Predictions on Images

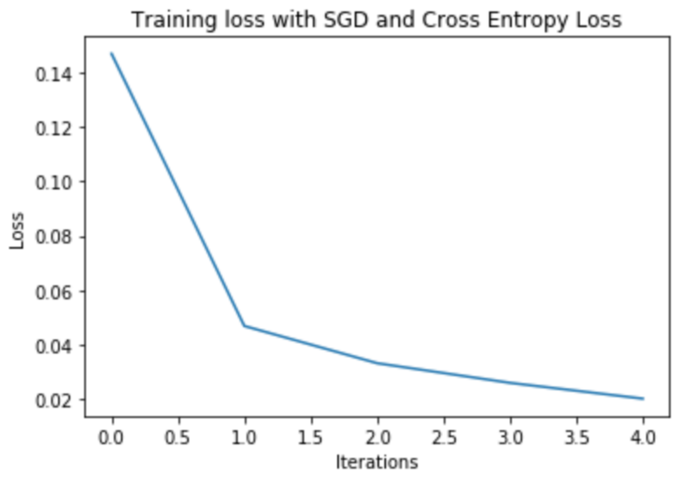
Since SGD provides the best results, I ran it over the test images and tried to see how well it predicts and tried to analyze the predictions.

Figure 3: Loss per iteration when using SGD and Cross Entropy Loss



```
Epoch:1 Iteration:2000 Loss: 0.487
Epoch:1 Iteration:4000 Loss: 0.147
Epoch:1 Iteration:6000 Loss: 0.111
Epoch:1 Iteration:8000 Loss: 0.097
Epoch:1 Iteration:10000 Loss: 0.083
Epoch:1 Iteration:12000 Loss: 0.070
Epoch:1 Iteration:14000 Loss: 0.068
Epoch:2 Iteration:2000 Loss: 0.045
Epoch:2 Iteration:4000 Loss: 0.054
Epoch:2 Iteration:6000 Loss: 0.042
Epoch:2 Iteration:8000 Loss: 0.049
Epoch:2 Iteration:10000 Loss: 0.043
Epoch:2 Iteration:12000 Loss: 0.052
Epoch:2 Iteration:14000 Loss: 0.041
Epoch:3 Iteration:2000 Loss: 0.031
Epoch:3 Iteration:4000 Loss: 0.036
Epoch:3 Iteration:6000 Loss: 0.035
Epoch:3 Iteration:8000 Loss: 0.035
Epoch:3 Iteration:10000 Loss: 0.034
Epoch:3 Iteration:12000 Loss: 0.033
Epoch:3 Iteration:14000 Loss: 0.032
Epoch:4 Iteration:2000 Loss: 0.028
Epoch:4 Iteration:4000 Loss: 0.026
Epoch:4 Iteration:6000 Loss: 0.022
Epoch:4 Iteration:8000 Loss: 0.025
Epoch:4 Iteration:10000 Loss: 0.027
Epoch:4 Iteration:12000 Loss: 0.023
Epoch:4 Iteration:14000 Loss: 0.029
Epoch:5 Iteration:2000 Loss: 0.018
Epoch:5 Iteration:4000 Loss: 0.020
Epoch:5 Iteration:6000 Loss: 0.024
Epoch:5 Iteration:8000 Loss: 0.016
Epoch:5 Iteration:10000 Loss: 0.015
Epoch:5 Iteration:12000 Loss: 0.025
Epoch:5 Iteration:14000 Loss: 0.022
Finished Training
```

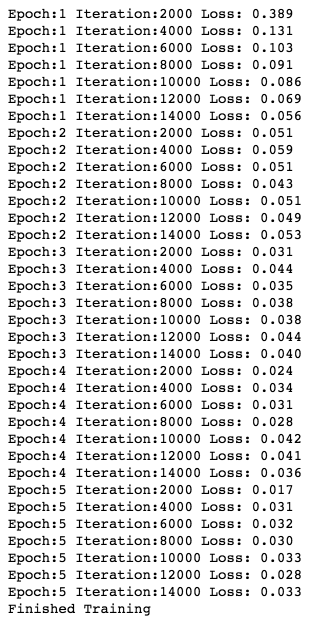
Figure 4: Training Loss per Epoch when using SGD and Cross Entropy.



Correct Predictions

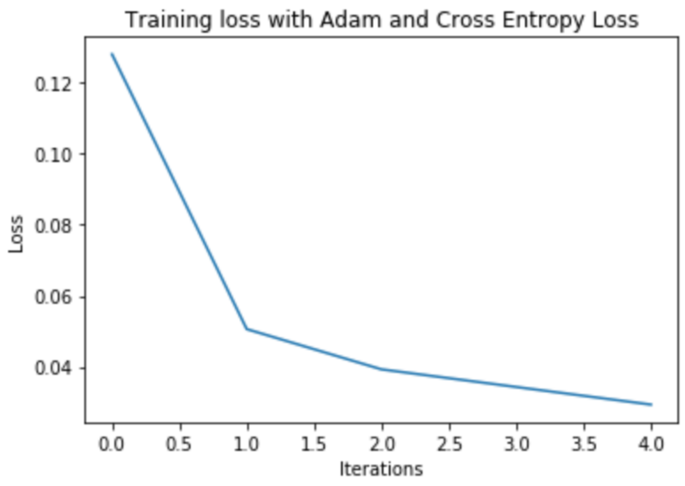
Out of 10,000 images the network correctly predicts for 9909 images. Random 4 images out of those 9909 are shown in figure 9. The labels of the images are the classes the network has predicted for them.

Figure 5: Loss per iteration when using Adam and Cross Entropy.



```
Epoch:1 Iteration:2000 Loss: 0.389
Epoch:1 Iteration:4000 Loss: 0.131
Epoch:1 Iteration:6000 Loss: 0.103
Epoch:1 Iteration:8000 Loss: 0.091
Epoch:1 Iteration:10000 Loss: 0.086
Epoch:1 Iteration:12000 Loss: 0.069
Epoch:1 Iteration:14000 Loss: 0.056
Epoch:2 Iteration:2000 Loss: 0.051
Epoch:2 Iteration:4000 Loss: 0.059
Epoch:2 Iteration:6000 Loss: 0.051
Epoch:2 Iteration:8000 Loss: 0.043
Epoch:2 Iteration:10000 Loss: 0.051
Epoch:2 Iteration:12000 Loss: 0.049
Epoch:2 Iteration:14000 Loss: 0.053
Epoch:3 Iteration:2000 Loss: 0.031
Epoch:3 Iteration:4000 Loss: 0.044
Epoch:3 Iteration:6000 Loss: 0.035
Epoch:3 Iteration:8000 Loss: 0.038
Epoch:3 Iteration:10000 Loss: 0.038
Epoch:3 Iteration:12000 Loss: 0.044
Epoch:3 Iteration:14000 Loss: 0.040
Epoch:4 Iteration:2000 Loss: 0.024
Epoch:4 Iteration:4000 Loss: 0.034
Epoch:4 Iteration:6000 Loss: 0.031
Epoch:4 Iteration:8000 Loss: 0.028
Epoch:4 Iteration:10000 Loss: 0.042
Epoch:4 Iteration:12000 Loss: 0.041
Epoch:4 Iteration:14000 Loss: 0.036
Epoch:5 Iteration:2000 Loss: 0.017
Epoch:5 Iteration:4000 Loss: 0.031
Epoch:5 Iteration:6000 Loss: 0.032
Epoch:5 Iteration:8000 Loss: 0.030
Epoch:5 Iteration:10000 Loss: 0.033
Epoch:5 Iteration:12000 Loss: 0.028
Epoch:5 Iteration:14000 Loss: 0.033
Finished Training
```

Figure 6: Training Loss per Epoch when using Adam and Cross Entropy.

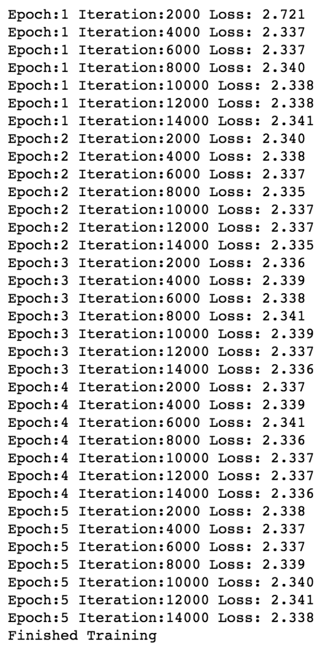


As we can see, these images are clear and there's no shape of any kind that may confuse the classifier for between multiple classes. For example, '2's are clearly '2's and '4' is clearly a '4', there's no bars or lines over the top of 4 that may confuse the classifier whether it's a 4 or 9.

Table 1: **Optimizers Summary**

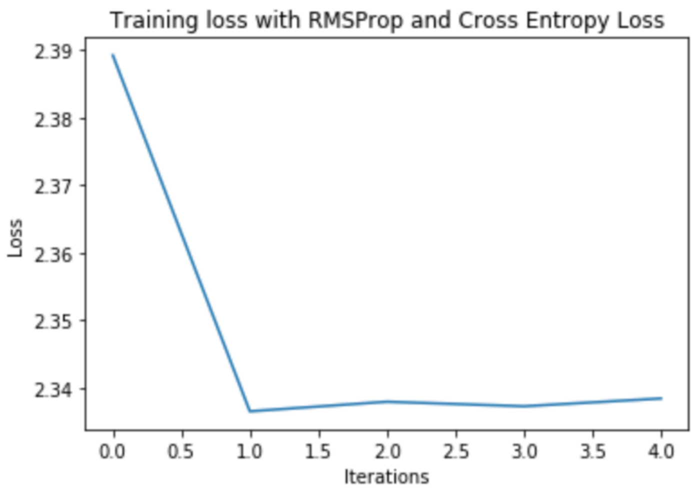
Optimizer	**Final Loss on Training Data**	**Test Accuracy**
Stochastic Gradient Descent	0.022	99.09%
Adam	0.033	98.89%
RMSProp	2.338	11.35%

Figure 7: Loss per iteration when using RMSProp and Cross Entropy.



```
Epoch:1 Iteration:2000 Loss: 2.721
Epoch:1 Iteration:4000 Loss: 2.337
Epoch:1 Iteration:6000 Loss: 2.337
Epoch:1 Iteration:8000 Loss: 2.340
Epoch:1 Iteration:10000 Loss: 2.338
Epoch:1 Iteration:12000 Loss: 2.338
Epoch:1 Iteration:14000 Loss: 2.341
Epoch:2 Iteration:2000 Loss: 2.340
Epoch:2 Iteration:4000 Loss: 2.338
Epoch:2 Iteration:6000 Loss: 2.337
Epoch:2 Iteration:8000 Loss: 2.335
Epoch:2 Iteration:10000 Loss: 2.337
Epoch:2 Iteration:12000 Loss: 2.337
Epoch:2 Iteration:14000 Loss: 2.335
Epoch:3 Iteration:2000 Loss: 2.336
Epoch:3 Iteration:4000 Loss: 2.339
Epoch:3 Iteration:6000 Loss: 2.338
Epoch:3 Iteration:8000 Loss: 2.341
Epoch:3 Iteration:10000 Loss: 2.339
Epoch:3 Iteration:12000 Loss: 2.337
Epoch:3 Iteration:14000 Loss: 2.336
Epoch:4 Iteration:2000 Loss: 2.337
Epoch:4 Iteration:4000 Loss: 2.339
Epoch:4 Iteration:6000 Loss: 2.341
Epoch:4 Iteration:8000 Loss: 2.336
Epoch:4 Iteration:10000 Loss: 2.337
Epoch:4 Iteration:12000 Loss: 2.337
Epoch:4 Iteration:14000 Loss: 2.336
Epoch:5 Iteration:2000 Loss: 2.338
Epoch:5 Iteration:4000 Loss: 2.337
Epoch:5 Iteration:6000 Loss: 2.337
Epoch:5 Iteration:8000 Loss: 2.339
Epoch:5 Iteration:10000 Loss: 2.340
Epoch:5 Iteration:12000 Loss: 2.341
Epoch:5 Iteration:14000 Loss: 2.338
Finished Training
```

Figure 8: Loss per Epoch when using RMSProp and Cross Entropy.



Incorrect Predictions

The network predicted the wrong classes for only 91 images out of 10,000. 4 of these 91 images selected randomly are shown in Figure 10. The labels of the images are what the network has predicted the classes to be. The actual classes are shown in the top left corner of the image.

We can see here that the images are very confusing to the human eye so it's no surprise that even the network is not able to recognize it. For example, the first image is a '9' but it's skewed in such a way that it looks like a '5' that has the top of the digit touching the middle area. Similarly in the second image '7' does look like a '1' and in the third image '9' is not completely joined and the open top makes it look like a '4'.

Figure 9: 4 Correctly Predicted Images Selected At Random

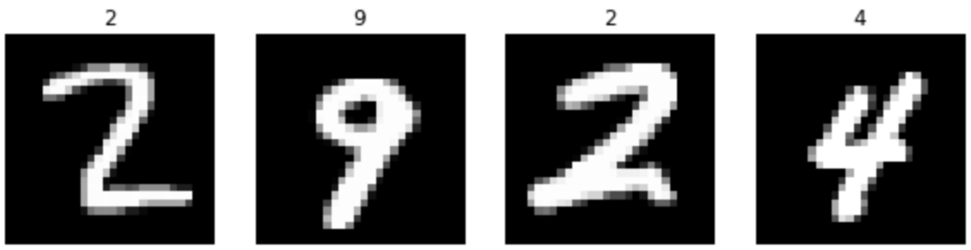
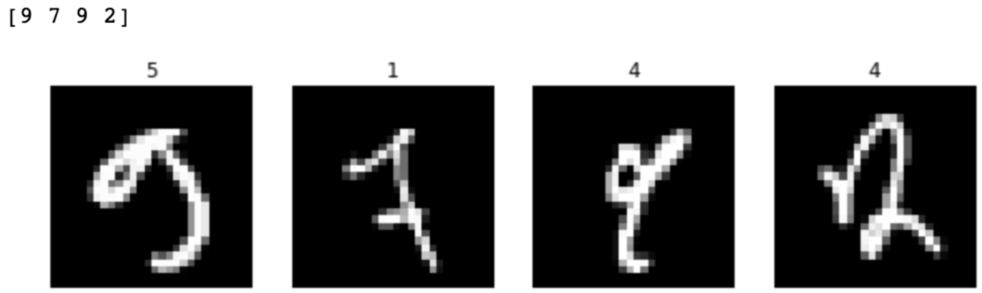


Figure 10: 4 Incorrectly Predicted Images Selected At Random

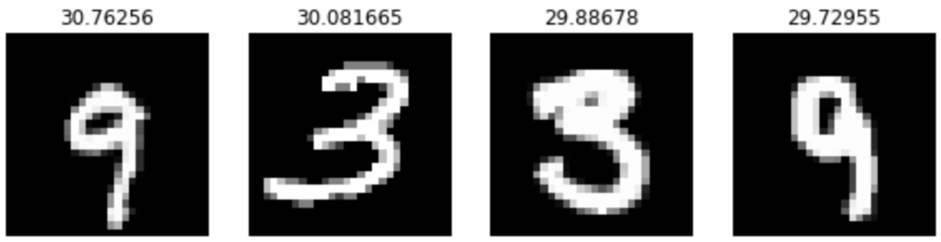


```
[9 7 9 2]
```

Most Correct Predictions

Taking all the correct images, I ordered them on the basis of the score which represents how confident the classifier is about that prediction. Higher the score on a correct prediction, higher is the classifier's confidence that it is correct. The 4 most correctly classified images are shown in Figure 11. The labels of the images represent the score given by classifier.

Figure 11: 4 Most Correctly Predicted Images



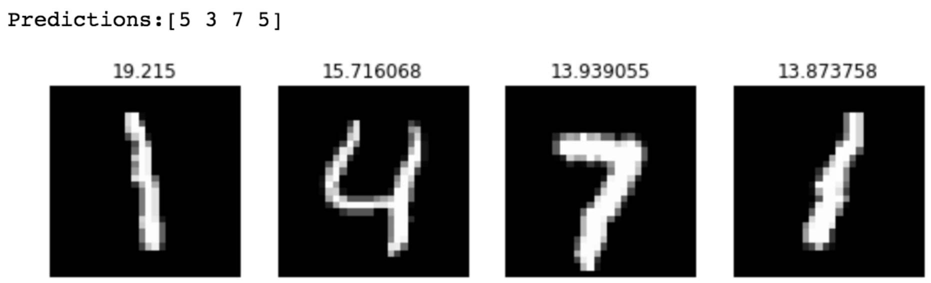
We can see here that the top 4 images that the classifier has the most confidence about correctly classifying are of either '9' or '3'.

Most Incorrect Predictions

Similarly, I ordered all the incorrect images on the basis of the score given by the classifier. Higher the score on the image, the more confidence the classifier had in it's correct prediction and more incorrect is the classification. The 4 most incorrectly classified images are shown in Figure 12.

The labels represent the score given by the classifier to the image and the classes predicted by the classifier for the images are given on top left. We can clearly see that ’1’ is nowhere close to being ’5’ and ’4’ is nowhere close to ’3’. But these predictions have higher scores than others depicting that classifier has higher confidence in their correct prediction. So these are more incorrect than others. Here we see that ’7’ has been predicted as ’7’ but the actual label is given as 2. So this is an anomaly here that I think might be a case of a mis-labeled data point.

Figure 12: 4 Most Incorrectly Predicted Images



```
Predictions:[5 3 7 5]
```

3 K-Nearest Neighbors

The next classifier that I use on the MNIST dataset is the K Nearest Neighbor Classifier. Instead of using just 1 Nearest Neighbor I tried checking the different values of K amongst 1,3,5,7 and 9 to see which classifier gave the best result.

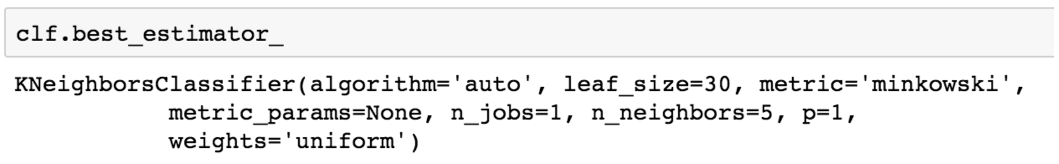
3.1 Methodology

For implementing KNN classifier I used the library function ”fetch_mldata” from sklearn which provided me with MNIST dataset. I split the data into train and test set but because the classifier was going to run on a CPU and not a GPU, I reduced the number of training points to 52,500 from 60,000 and increased the number of test-points to 17,500 from 10,000 in order to train faster and to test on more data.

Initially I used the dataset from PyTorch similar to CNN in Section 2. but the dataset was by default normalized and all the pixel values were between 0 and 1. So the points were really close to each other in a 784 dimensional space. This made it really difficult for the classifier to find the closest neighbor or neighbors to the test points. So I used the unnormalized dataset given by sklearn where all the pixel values are between 0 and 255.

I performed Cross Validation in order to find the best hyper-parameters for the model. For this, I used the L1 distance and L2 distance as the different possible distance measures and the number of nearest neighbors were to be considered from 1,3,5,7 and 9 neighbors. The model gave the best results when L1 distance was used and 5 nearest neighbors were considered. The parameters of the best model are summarized in Figure 13. Based on these results I built a new KNN classifier with the best parameters mentioned above and fit the training data to it. Then I ran it on test data to get the predictions.

Figure 13: Parameters of the best estimator



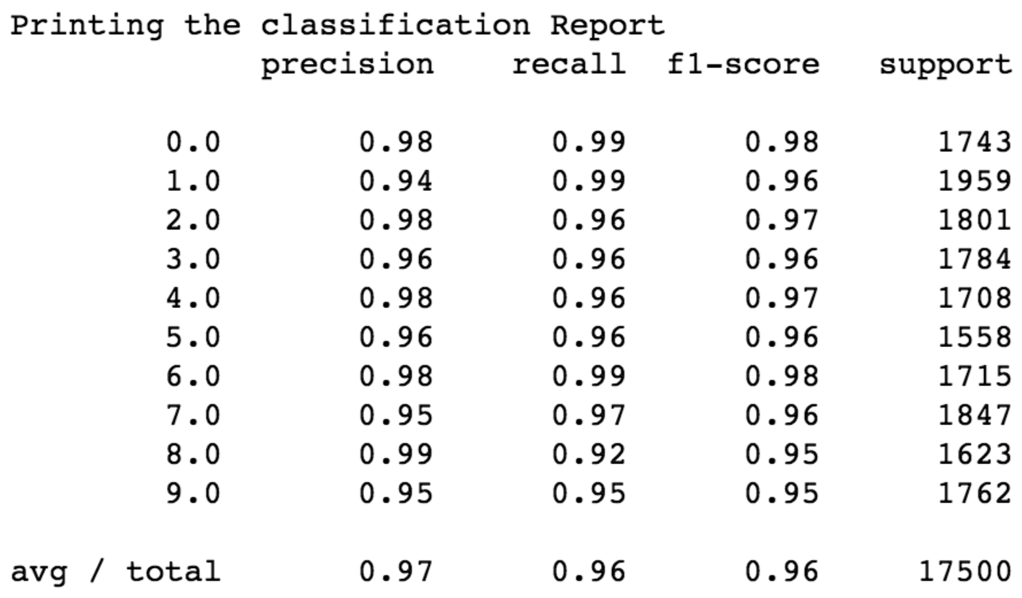
```
clf.best_estimator_
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=5, p=1,
           weights='uniform')
```

3.2 Results

A KNN classifier that takes into consideration for every data point 5 nearest neighbors and uses L1 distance gave an accuracy of 96.49% on the test dataset.
The other details about the performance of the classifier are given in Figure 14. This gives us a lot of information about each class. For example, we can see that we have the highest precision for digit 8. This tells us that out of all the digits that were recognized as 8, 99% of them were accurate. 8 also has the lowest recall which tells us that out of all the images that belong to class 8, only 92% of them were correctly classified as 8. So we can come to the conclusion that class 8 has a least number of False Positives and has highest number of False negatives.

Figure 14: Performance of KNN on test dataset



```
Printing the classification Report
             precision    recall  f1-score   support

        0.0       0.98      0.99      0.98      1743
        1.0       0.94      0.99      0.96      1959
        2.0       0.98      0.96      0.97      1801
        3.0       0.96      0.96      0.96      1784
        4.0       0.98      0.96      0.97      1708
        5.0       0.96      0.96      0.96      1558
        6.0       0.98      0.99      0.98      1715
        7.0       0.95      0.97      0.96      1847
        8.0       0.99      0.92      0.95      1623
        9.0       0.95      0.95      0.95      1762

avg / total       0.97      0.96      0.96     17500
```

The confusion matrix for the results obtained for different classes is shown in Figure 15. This confusion matrix tells us which classes the classifier messes up with. For example, it misclassifies 2 as 7 most of the time which is understandable as they have slightly similar shape. This is not a symmetric matrix. So when it misclassifies 2, it classifies it as 7 most of the time but it misclassifies 7 mostly as 1!. Similarly, the classifier confuses 4 with 9, 8 with 5 and 9 with 7. We can also see that digit 1 has the least number of incorrect classifications amongst all the classes. This makes sense because the digit '1' has such a basic shape that the classifier would confuse other digits, like 7 or 4(as seen from the confusion matrix),with being 1 but not 1 being other digits. **other words '1' has a lot more False Positives than False Negatives**.

4 Support Vector Machines

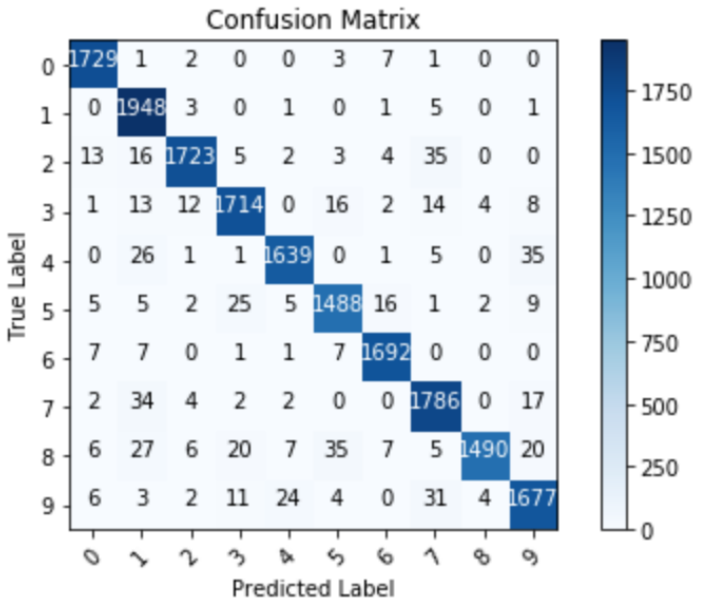
The third classification algorithm that I run over MNIST is the Support Vector Machine. The data I use for SVM is exactly the same as used in KNN classifier in Section 3. It has 52,500 training data points and 17,500 test data points.

4.1 Methodology

Here, I've tried to compare the results of running the classifier through SGDClassifier and SVC classes of sklearn library. First I create a classifier using SGDClassifier over the entire training data. The SGDClassifier implements a Regularized Linear SVM if hinge loss is used along with l2 penalty.

Figure 15: Confusion Matrix for KNN classifier

Confusion Matrix without normalization

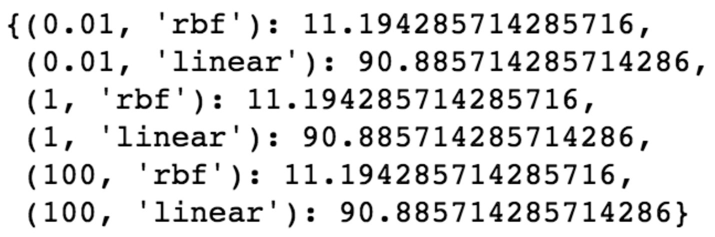


Then I created separate classifiers using SVC class of sklearn with C taking the values of 0.01, 1.0 or 100.0 and kernels being either rbf or linear. For fitting the train data on these classifiers, it takes a long time, so I reduced the number of training data points from 52,500 to 20,000 by randomly selecting the data points. The size of the test data remained the same.

4.2 Results

When a Linear SVM is created using SGDClassifier it gives an accuracy of 87.82% on the test data. The results of running the SVM classifier through SVC on the test data are shown in Figure 16.

Figure 16: Test Accuracy for different values of C and Kernel



```
{(0.01, 'rbf'): 11.194285714285716,
 (0.01, 'linear'): 90.885714285714286,
 (1, 'rbf'): 11.194285714285716,
 (1, 'linear'): 90.885714285714286,
 (100, 'rbf'): 11.194285714285716,
 (100, 'linear'): 90.885714285714286}
```

Observe here that the value of C for a particular kernel doesn't have any affect on the test accuracy. So I used the default value of C in SVC of 1 and Linear kernel as the best hyper-parameters for the SVM to classify MNIST data and consider this as the final SVM. As we can see from Figure 16, the SVM classifier with C = 1.0 and Linear kernel gives an accuracy of 90.89%. The Confusion Matrix for SVM is given in Figure 17.

Figure 17: Confusion Matrix for SVM classifier

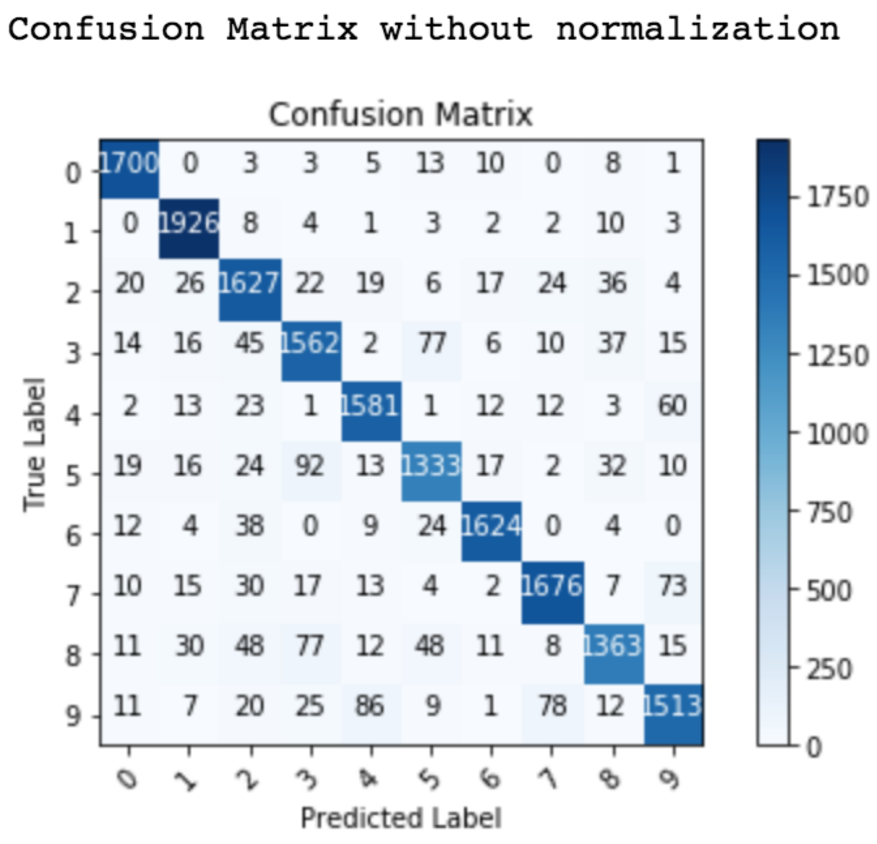


Table 2: **Models Summary**

Model	**Hyper Parameters**	**Test Accuracy**
CNN	SGD and CE	99.09%
CNN	Adam and CE	98.89%
CNN	RMSProp and CE	11.35%
KNN	neighbors = 5, L1 distance	96.49%
SVM	SGDClassifier	87.82%
SVM	C = 0.01, rbf	11.19%
SVM	C = 0.01, Linear	90.89%
SVM	C = 1, rbf	11.19%
SVM	C = 1, rbf	90.89%
SVM	C = 100, rbf	11.19%
SVM	C = 100, Linear	90.89%

5 Conclusion

The results of different models used in this project are given in Table 2. We can conclude that CNN with SGD and Cross Entropy Loss is the most effective in classifying handwritten digits of the MNIST dataset with an accuracy of 99.09%.

References

[1] Pytorch Tutorials: `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`

[2] KNN Classifier: `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`

[3] SVM through SVC: `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

[4] SVC through SGDClassifier: `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html`

[5] GridsearchCV: `http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`

[6] Code for Confusion Matrix: `http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py`