Analysis of Competition Part

Link to model -
https://drive.google.com/open?id=1dlUVcJSb5plHPrDorKAwJsDhCNUCkaiH

Set of Libraries Used:
The only main library used was **Keras** which was running tensorflow at backend.

Details of Architecture:

1. In preprocessing, only naive normalization was performed. All the dimensions were divided by **255**.
2. The insights for architecture came along as I trained.
3. Initially, I started with 2 convolutional layers.
4. Soon, I changed layers from **3** to **4** to **5** and **6(the best)** (across days) and got increasing accuracy, but with diminishing returns.
5. The optimizer that worked best was **Adam** optimizer.
6. One thing that was penalizing was **overfitting**. For that, I exploited dropout parameter, tweaking it and its placements.
7. Also, once I imagined **max pooling** to be of little help because all it's doing was reduce image size with **losing** details of data. It turned out to be a bad choice. Model started severely **overfitting**. Performance degraded. **Training time** also jumped high.
8. Pooling coupled with dropout prevent model's overfitting.
9. **Dense layers** were not much played with, but slowly decreasing number of nodes over 2 layers just turned right.

Training Architecture:

1. Training this wasn't easy.
2. Following approach to imitate results are:

a. First train the final model using batch of **256**. Leave learning rate default. ~ 60 epochs. (~91.2 - 92 %)

b. Then, when we say no substantial improvements, we increase batch size to 512 --> 1024 --> 2048 progressively, training for 20-25 epochs for all. (92- 93)%

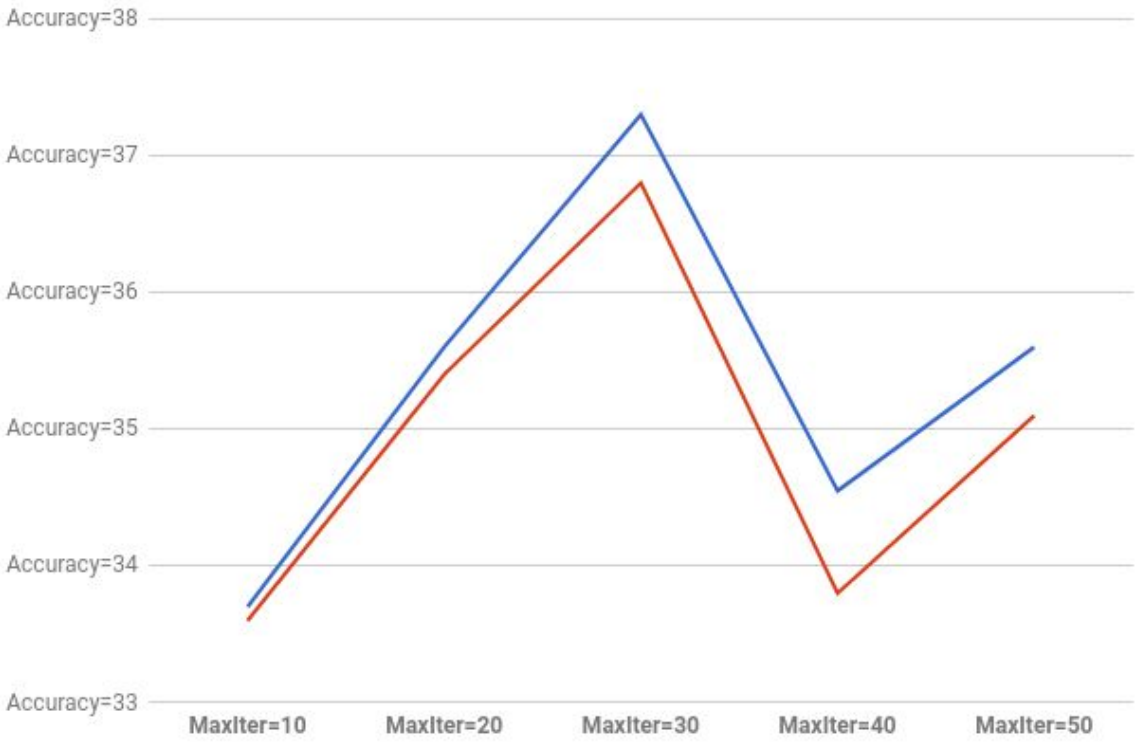
c. One final thing was fine tuning model with random shuffled data cut from original data, each time for **4-6** epochs. This is done **2-3** times. This might just completely fit the data but it produces **quality predictions**.

Also, I tried an **ensemble hack** by downloading all solutions and combining all by majority, but no improvements were observed.

Just few no.s of conv layers vs test accuracy:
6 layers: 93.83
5 layers: 92.45
2 layers : 89.5
3 layers: 90.91

Analysis for K Means:

Test:0.36417 Train:0.36806 for first part. This shows that k means is not the right choice because it might that clusters are not well separated in space, which causing K means to confuse. Also, in a supervised setting, it is not a wise choice to throw in unsupervised algorithms as we are not fully exploiting the potential of the problem.



Graph of Train Test Accuracy with varying max_iter value

The variation in graph with changing iteration limit indicates instability in learning, indicating again that K-means might not be a very good learner for this task.

Analysis for SVM:

Linear SVM did not much variation in accuracy when its hyperparameter **C** was varied. The best hovered around 1, with Train Accuracy=0.656 Test Accuracy=0.654. Better performance was expected since it exploit label to split space with hyperplanes, but no considerable change with hyperparameters reflect that underlying space is indeed non-linear, and there is only so much, a linear separator can do about it. More on it in comparison section.

Analysis for Neural Networks:

Hidden units:		Train | Valid | Test
70	->	81 74 73.1
100	->	85 74.2 75.2
200	->	95 75 75
20	->	75 67.5

100 in hidden layer seems to be good fit. This is because it appears to generalize well to validation as well as test set. **200** seems to have **overfit** the training data, which may fail miserably in testing environment. Maybe somewhere between 100 and 150 neurons would be **optimal**.

Analysis for Convolutional Neural Networks(CNN):

Best set of parameters obtained were for this model:

```
model.add(Conv2D(64, (5, 5), padding='same', activation='relu',
input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
```

Accuracy: Train 95 Test 85

One thing to observe while training CNN was overfitting. Even this single models was overfitting on training data. Changing parameters in local neighbourhood of above failed to give any promising benefits. Removing pooling caused severe overfitting and increased training time per epoch.

Overall Comparison:

The sudden increase in accuracy using even a naive CNN reflects why CNN are **outperforming** in current world. Their feat is because of the way the network is designed exploiting **structural properties** of an image. Other algorithms are general algorithms, meaning they can be applied to many-many settings, whereas CNN is in particular designed for image based tasks.

What a distinct observation we see is just a single convolutional layer neural networks has already surpassed results of previous mentioned algorithms by a margin. Also, it didn't require any sort of preprocessing of data (as in SVM). This shows how powerful CNN's are in fitting arbitrary functions and require no *"feature engineering"* on input.

Some experiments were performed trying to fit ***Gaussian Kernel*** to data but with cross validation, they at best gave 82-83% which is again not as bad as others, but lower compared to CNN's. Also, the training time for such SVM was non-trivial and CNN in same limit of time produced pretty good results(competition part).