Assignment 1 :
MNIST Handwritten Digits Classifier

This is one of the classic problems which help show the power and viability of a neural network. This involves processing small binary images with a handwritten number to figure out which digit the image represents. This is a hard problem to solve conventionally because there are a lot of variabilities which need to be addressed to ensure that the solution is robust, like the slant, handwriting, obscurity, noise and so on.

Dataset :

The data set contains small binary images of size 28 X 28 flattened out as a 784 dimensional vector inside a csv file.

The training set contains about 32000 different images with their digit labels
The test set contains about 10000 images

Approach :

To get a good baseline model I implemented a simple two hidden layers neural network with a softmax classifier at the end. The approximate architecture is as follows:

INP -> *** W1 + b1** -> ***W2 + b2** -> **Y -> softmax(Y, Y_ORIGINAL)**
[1 x 784] -> [784 x 4096] + [1 x 4096] **(1 x 4096)** -> [4096 X 10] + [1 x 10]

With this setup I was able to reach about 92% accuracy on my validation subset.

Through intuition we know that the curves and lines forming a digit are spatial in nature and hence by flattening out the image we have lost valuable information. Thus in order to train on the image itself I replaced the normal neural network with a convolutional neural network (CNN) to take full advantage of the spatial data.

The architecture I used is loosely based on an early version of LeNet which contains two hidden convolutional layers with max pooling and softmax classification after the fully connected layer.

Input :
28 X 28 X 1 image as input

Convolution layer 1 :
32 5X5X1 kernels

Non linearity :
reLU + 2 x 2 max pooling

Convolution layer 2:
64 kernels of size 5 X 5 X 32

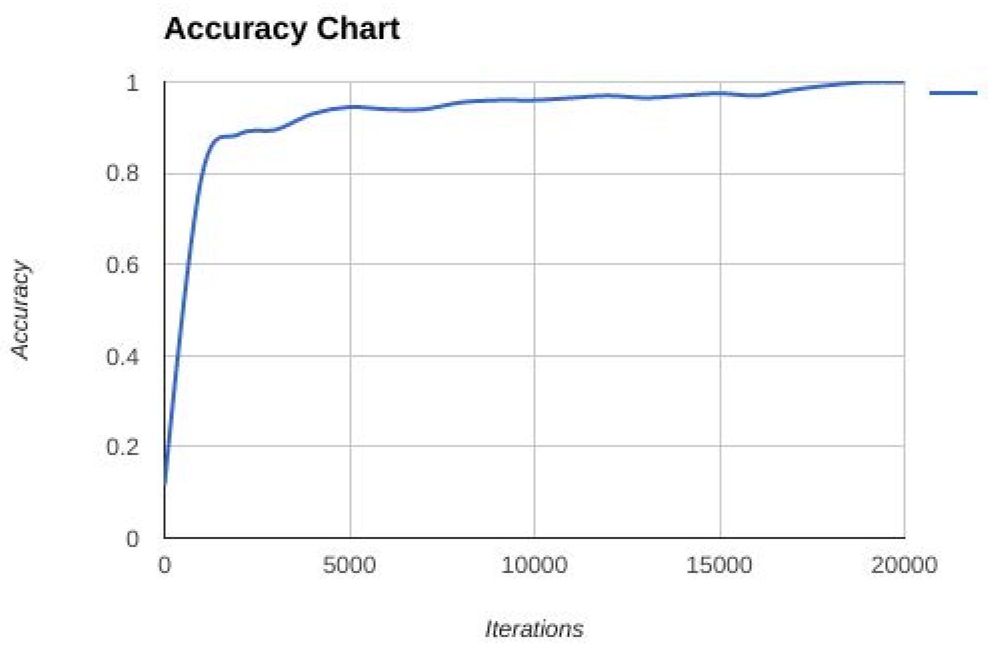
Non Linearity :
reLU + 2 x 2 max pooling
7 X 7 X 64

Fully Connected Layer 1 :
Flatten & convert to a 1024 dimensional vector

Fully Connected Layer 2 :
Convert the 1024 dimensional vector to a 10 dimensional vector which gives the probability distribution of the image belonging to one of the 10 classes (after normalization).

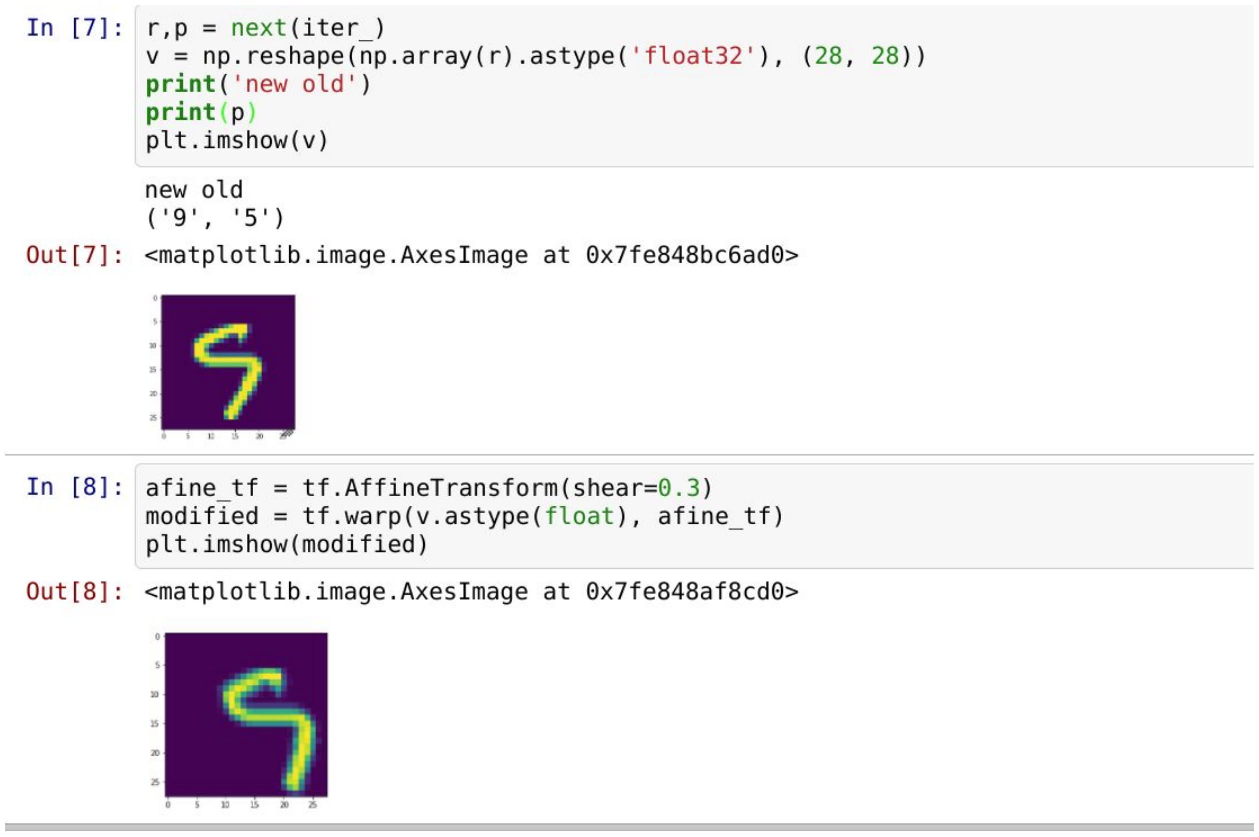
For the purposes of classification we can compare the argmax values of this representation and the given one hot encoded label.

The training can be summarized by the following chart



The architecture involved the use of few techniques to get the final observed accuracy of 99.89%. The final trained model is based on multiple iterative changes done to the pipeline which are listed below:

1) **Data Augmentation :**
   Since we had about 32000 images for training , I used few affine transformations to virtually augment the dataset. Also since MNIST is built into tensor flow I used the training set from that to pre - train the model to allow the weights to get initialized properly. The affine transformations allowed the production of many skewed cases which helped/forced the model to learn general space invariant features which helped it to identify very obscure cases.



```
In [7]: r,p = next(iter_)
        v = np.reshape(np.array(r).astype('float32'), (28, 28))
        print('new old')
        print(p)
        plt.imshow(v)

        new old
        ('9', '5')
Out[7]: <matplotlib.image.AxesImage at 0x7fe848bc6ad0>
```

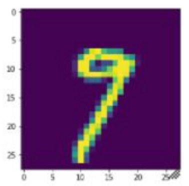
```
In [8]: afine_tf = tf.AffineTransform(shear=0.3)
        modified = tf.warp(v.astype(float), afine_tf)
        plt.imshow(modified)

Out[8]: <matplotlib.image.AxesImage at 0x7fe848af8cd0>
```

```
print('new old')
print(p)
plt.imshow(v)
```

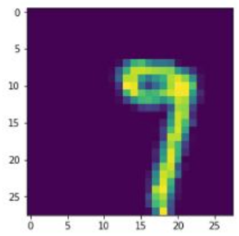
```
new old
('9', '7')
```

```
Out[9]: <matplotlib.image.AxesImage at 0x7fe848a28dd0>
```



```
In [10]: afine_tf = tf.AffineTransform(shear=0.3)
         modified = tf.warp(v.astype(float), afine_tf)
         plt.imshow(modified)
```

```
Out[10]: <matplotlib.image.AxesImage at 0x7fe848967750>
```



I ended up bloating the training set up using a combination of shear, translation and rotation.

2) Dropout :

Since the model had a relatively high capacity, there was a danger of overfitting the noisy dataset so I had to add a dropout layer at the end to randomly shut down (drive to 0) half of the neurons to force the network to learn more robust features which give a holistic representation of the dataset.

3) Mini Batch Training :

Since the dataset is large I used mini batch training to converge to the solution in a very efficient manner. By experimenting I found that the batch size of 128 seems to have the most favourable effect on the final training. This is done via the shuffle iterator I wrote based on the generator pattern advocated by python's official documentation.

Things Considered (but not implemented) :

1) **Batch Normalization instead of dropout :**

When I implemented batch norm , I faced a sudden sluggish convergence towards the optima. I suspect that my implementation has a bug and moved on to implement the dropout layer. Since the existing architecture performs in an acceptable manner I did not implement batch norm.

2) Ensemble models :

I have checkpoints of all my epochs and I was planning on using the best 3 or 4 models as an ensemble to enable better performance. But since the final model performed well I did not go for this approach.

3) Adding Noise to the image :

I wanted to add a gaussian blur to the image to see the effects but due to time/memory constraints I did not go ahead with this.

Conclusion :

Thus I built a CNN that is able to recognize handwritten digits on tensor flow. The final model achieves 1.0 on the training set and about 0.9989 on the unknown test set (50%).