Deep Learning on SVHN dataset using Convolution Neural Networks

Project Overview:

The aim of this project is to recognize numbers in an image. The images used are called “The Street View House Numbers (SVHN) Dataset” [1].

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images. [2]

The dataset has 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data. The extra dataset is not being used for this project as its size is more than 1GB and gets really heavy to process even on GPU.

The SVHN dataset comes in two formats:

1. Original images with character level bounding boxes.

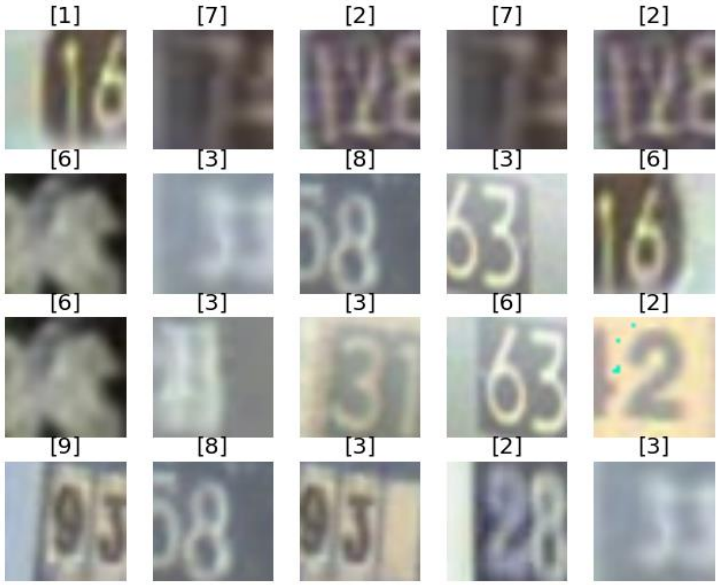
2. 32-by-32 images centered around a single character (many of the images do contain some distractors at the sides).

This project makes use of the second format.

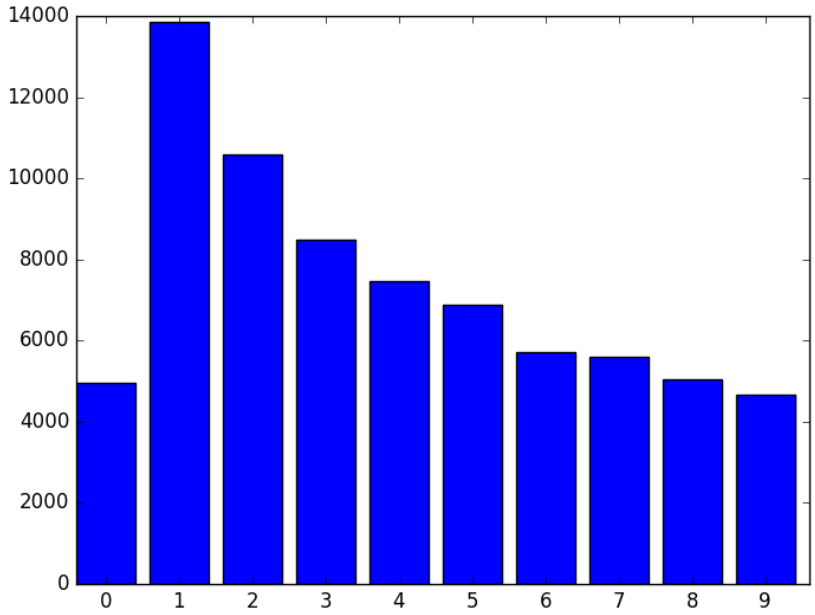
Some sample images in the dataset are shown below:

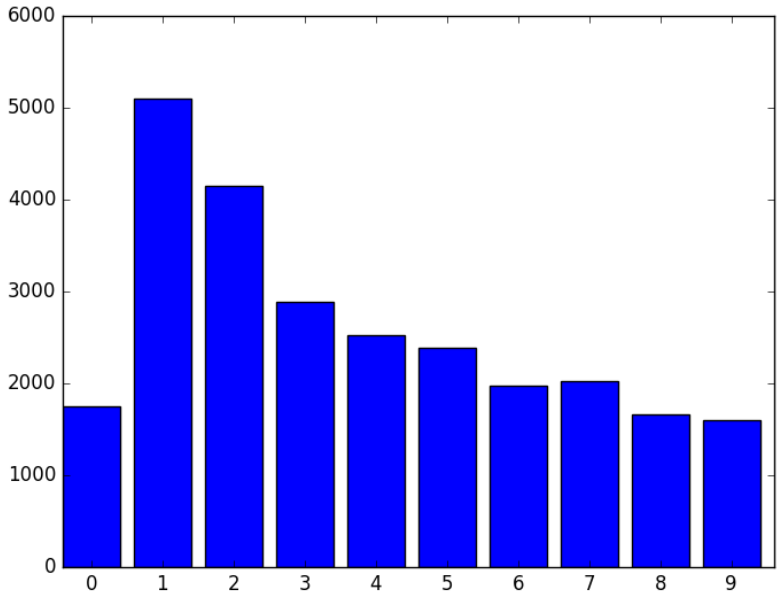
Each image is of dimension 32x32

Below are some sample images with labels from train_32x32.mat



Below are the histograms of class distribution in train_32x32.mat and test_32x32.mat





As above histogram shows that classes are not evenly distributed in the dataset. This may or may not cause a problem. Using trial and error method, on trying the unevenly distributed data for training, the model was able to achieve considerable accuracy, hence the data was left as it is.

Problem Description:

As shown in the sample images, each image contains a number. The other numbers on either side are to be ignored and not to be considered. Only the center numbers in the images are relevant.

Making use of a convolution neural network, a model is trained to identify the numbers in each images. Once the model is trained, the model will be able to identify the number in an unseen image.

A Convolutional Neural Networks are very similar to ordinary Neural Networks. Neural Networks are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they have a loss function (e.g. Softmax) on the last (fully-connected) layer. [3]

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a Convolutional Neural Network have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in SVHN are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). [3]

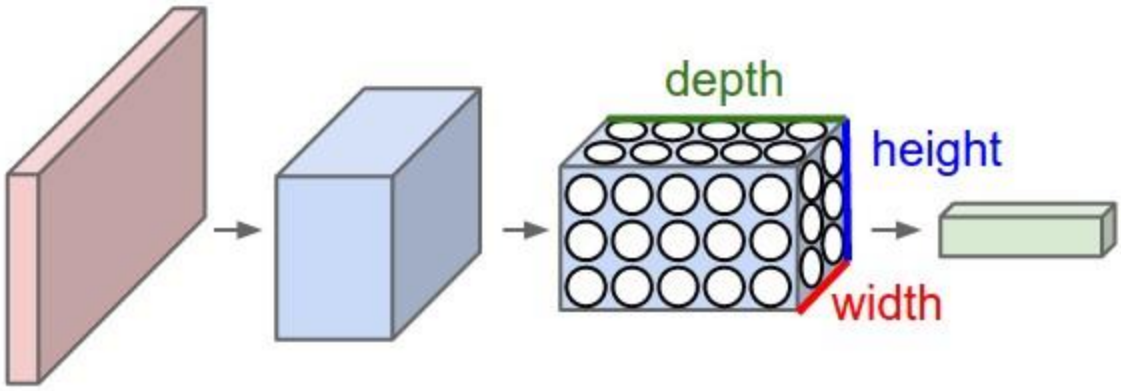


Figure 1 A regular 3-layer Neural Network Source: http://cs231n.github.io/convolutional-networks/

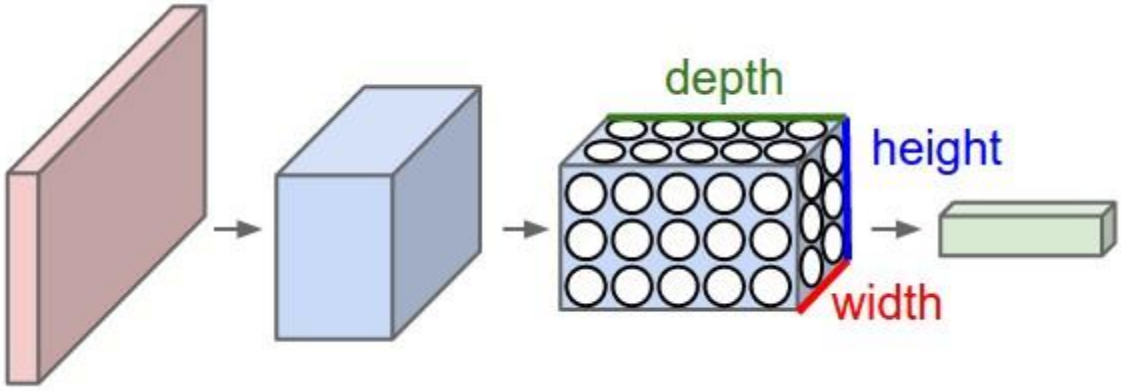


Figure 2 A Convolutional Neural Network Source: http://cs231n.github.io/convolutional-networks/

In Figure 2, A Convolutional Neural Network arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). [3]

A training dataset is used to train the model. The data from test_32x32.mat are divided into two datasets called Validation data and Test data. Validation data is used to estimate the performance of the model during each epoch and Test data is used to evaluate the model performance after training on unseen data.

A convolutional neural network with six layers is designed to create a model that can recognize numbers in the images of SVHN dataset. Max pooling and dropout has been applied after the second layer and again after the fourth layer, also one more dropout has been applied after fifth layer. These measure are applied to prevent overfitting.

Metrics:

Accuracy is the metric that is used to calculate the performance of the model. This is a classification problem and the dataset is also large. In this case it will be best to use accuracy as it will be fast and efficient.

A simple neural network model is trained and it achieves validation accuracy of 80.95% and test accuracy of 81.24%. This is set as a benchmark for the convolution neural network trained and the target is set as an accuracy of at least 90%.

Methodology:

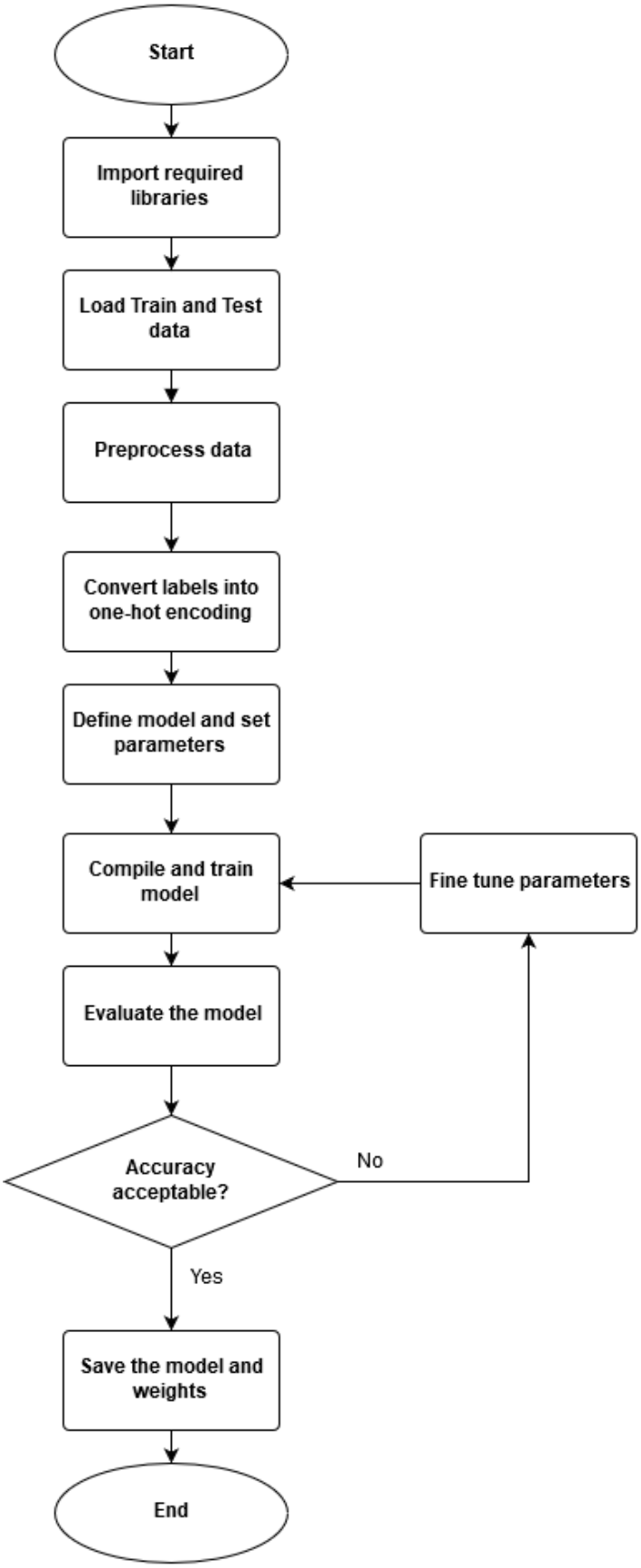


Figure 3 Flow chart for keras_svhn_training.py

To design the convolution neural network in python, first all the required libraries is imported. Then the train and test dataset is loaded. This training and testing datasets are in '.mat' format. To load this the loadmat() function from scipy library is used.

```
X_Train = loadmat('train_32x32.mat')['X'].astype('float64')
y_Train = loadmat("train_32x32.mat")['y'].astype('int8')
X_Test = loadmat("test_32x32.mat")['X'].astype('float64')
y_Test = loadmat("test_32x32.mat")['y'].astype('int8')
```

The features and the labels are loaded separately into separate arrays. When the features are loaded, it is in the shape (32, 32, 3, 73257). The data is needed in the shape (73257, 3, 32, 32), so a transpose of the array is taken to convert the shape (32, 32, 3, 73257) into (73257, 3, 32, 32).

```
X_Train = X_Train.transpose()
X_Test = X_Test.transpose()
```

The data from test_32x32.mat file is randomly split 50/50 into test and validation data. The test_32x32.mat contains 26032 samples, so after split both test and validation data will contain 13016 samples.

```
X_Test, X_Valid, y_Test, y_Valid = train_test_split(X_Testing, y_Testing, test_size=0.5)
```

Now, the training, validation and testing data should be scaled. To do this it is divided by 255, which is the maximum value a pixel can have.

```
X_Train /= 255
X_Test /= 255
X_Valid /= 255
```

There are 10 classes in the label array. Class name 1 corresponds to number 1 and class name 9 corresponds to the number 9. But the class name 10 corresponds to the number 0. This will not allow to create one-hot encoding of the label data. So to overcome this, the class name 10 is replaced by 0, so now each class name represents its corresponding number.

```
np.place(y_Train, y_Train == 10, 0)
np.place(y_Test, y_Test == 10, 0)
np.place(y_Valid, y_Valid == 10, 0)
```

It is then converted into one hot encoding scheme.

```
y_TrainB = np_utils.to_categorical(y_Train, nb_classes)
y_TestB = np_utils.to_categorical(y_Test, nb_classes)
y_ValidB = np_utils.to_categorical(y_Valid, nb_classes)
```

This concludes the preprocessing of the data and the model is defined next.

The model is defined by:

```
model = Sequential()
```

A sequential model in keras is a model with a linear stack of layers.

After defining the model, layers will be added to the model using the following code:

```
model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='valid',
                        input_shape=(3, img_rows, img_cols)))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, nb_conv, nb_conv))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(nb_pool, nb_pool)))
model.add(Dropout(0.25))

model.add(Convolution2D(nb_filters, nb_pool, nb_pool, border_mode='valid'))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, nb_pool, nb_pool))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(nb_pool, nb_pool)))
model.add(Dropout(0.25))
```

```
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

Convolution2D function is a convolution operator for filtering windows of two-dimensional inputs.

- nb_filters is the number of convolution filters to use.
- nb_conv is the convolution kernel size.
- nb_pool is the size of pooling area for max pooling.
- input_shape specifies the size of the input array.
- Activation function specifies which activation function to use on the corresponding layer.
- MaxPooling2D is used for pooling operation for spatial data.
- Pooling is used for non-linear down-sampling to reduce computational cost.
- Flatten layer is used to flatten the input. This flatten layer is set as a dense layer with output dimension set to 128.
- Dropout function applies dropout to the input. Dropout consists in randomly setting a fraction x of input units to 0 at each update during training time, which helps prevent overfitting. [4]
- Dropout is a regularization technique, which aims to reduce the complexity of the model with the goal to prevent overfitting.
- A final dense layer is added with output dimension set to number of classes, which is 10 in this case.

Once the model is defined, it needs to be compiled. Compile means to configure the defined model for training. The following code is used to compile the model:

```
model.compile(loss='categorical_crossentropy', optimizer='adadelta',
    metrics=['accuracy'])
```

During compiling the model, the loss function, the optimizer and the performance metrics needs to be defined. For this model, the categorical crossentropy is selected

as the loss function, the optimizer selected is adadelta and the accuracy is used as performance metrics.

After compiling the model, it should be trained using the following command:

```
model.fit(X_Train, y_TrainB, batch_size=batch_size, nb_epoch=nb_epoch,
        shuffle=True, verbose=1, validation_data=(X_Valid, y_ValidB))
```

In the above code it is specified to use X_Train as training features and y_TrainB as training labels and X_Valid are validation features and y_ValidB are validation labels. Also setting shuffle to True makes the model shuffle the data before selecting data of the size of batch size specified. This allows for random selection of data for training, which increases the accuracy and reduces overfitting and also prevents underfitting.

After training is done, the model needs to be evaluated for how well it performs on the testing data. To do this the following code is used:

```
score = model.evaluate(X_Test, y_TestB, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

This will return the scalar test loss and the accuracy of the model on the testing data.

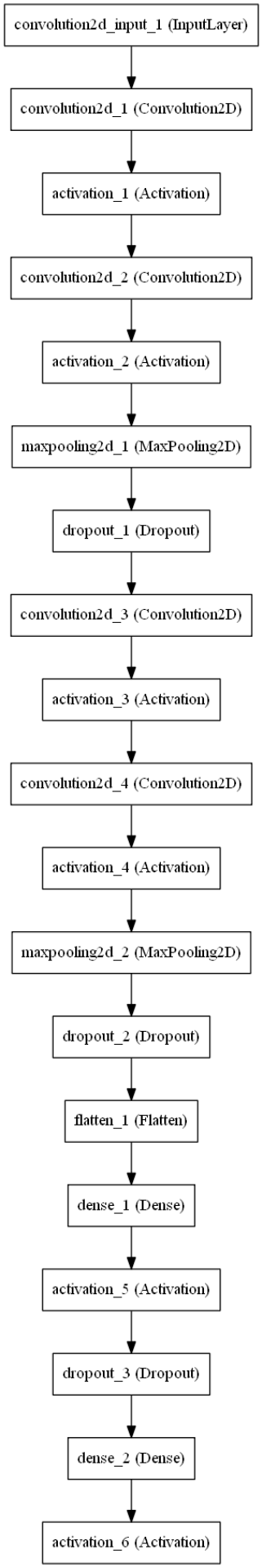
If the results are acceptable, the model can be saved for future. Using the saved model numbers in new unseen images can be recognized. The model can be saved with the following code:

```
json_string = model.to_json()
open('keras_svhn_architecture.json', 'w').write(json_string)
model.save_weights('keras_svhn_weights.h5')
```

This will save the model architecture as well as the weights of the trained network, which can be loaded directly without the need to define or train the model again.

During the early stages of model defining a Convolution Neural Network was designed with 4 layers. However, this model was not very efficient as it was taking very long to learn i.e. it took more than five epochs on average for the validation accuracy to go above 20%. By trial and error method, tweaks were made in dropout values as well as number of layers. Finally, a six-layer model was defined which obtained a decent validation accuracy of 60% or above on the first epoch.

The graph of the convolution network is shown in next figure.


convolution2d_input_1 (InputLayer)
convolution2d_1 (Convolution2D)
activation_1 (Activation)
convolution2d_2 (Convolution2D)
activation_2 (Activation)
maxpooling2d_1 (MaxPooling2D)
dropout_1 (Dropout)
convolution2d_3 (Convolution2D)
activation_3 (Activation)
convolution2d_4 (Convolution2D)
activation_4 (Activation)
maxpooling2d_2 (MaxPooling2D)
dropout_2 (Dropout)
flatten_1 (Flatten)
dense_1 (Dense)
activation_5 (Activation)
dropout_3 (Dropout)
dense_2 (Dense)
activation_6 (Activation)

Results:

As discussed earlier a simple neural network was designed and its accuracy was 81.24% on the test dataset. A new convolution neural network was designed and it was trained on the training dataset, validated on validation dataset and tested on testing dataset.

Using the trained model an accuracy of 90.32% was obtained on the test data. The model was run for 20 epochs. The validation accuracy for 1st epoch was 69.82% and for 20th epoch it was 90.13%. Each epoch ran for an average of 240.3 seconds. Total time all the epochs took to run was 4806 seconds on a GPU. It was fast because it was run on GPU, it will take 10 times more time on a CPU.

Conclusion:

The main obstacle in this project was to increase the accuracy of the model while maintaining a reasonable computation time. So setting up the environment to train the model using GPU was the main crisis during the early stage of development. Once the system was setup for utilizing GPU, training and fine tuning the model became effortless and less time consuming. Using the model designed, I was able to successfully train it to predict the number in unseen images. This model can be further fine-tuned to increase the accuracy even more. Using a faster GPU and adding more layers and more epochs a higher accuracy can be obtained.

Also rather than dropping samples from train and test data to make all class equal, data from extra_32x32.mat can be taken and a train and test data can be created with equal number of samples for all the classes.

To further increase the training speed, the script can be run on multiple GPUs set up in parallel or multi-core GPUs.

References:

[1] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*.

[2] http://ufldl.stanford.edu/housenumbers/

[3] http://cs231n.github.io/convolutional-networks/

[4] http://keras.io/layers/core