Capstone Project Joel S

Machine Learning Engineer Nanodegree January 27th, 2017

Handwritten Digit Recognizer using Deep Neural Network

I. Definition

Project Overview

Handwriting recognition is the ability of a computer to receive and interpret intelligible handwritten input from sources such as paper documents, photographs, touch-screens and other devices. The image of the written text may be sensed "off line" from a piece of paper by optical scanning (optical character recognition) or intelligent word recognition. Alternatively, the movements of the pen tip may be sensed "on line", for example by a pen-based computer screen surface, a generally easier task as there are more clues available [1].

Handwriting Recognition plays an important role in storage and retrieval of typed and handwritten information. A wealth of historical papers exists in a physical format. This includes genealogical information, old family records, written manuscripts, personal diaries and many other pieces of a shared past. Consistent review of this information damages the original paper and can lead to physical data corruption. Handwriting recognition allows people to translate this information into easily readable electronic formats [2].

Since 2009, the recurrent neural networks and deep feedforward neural networks developed in the research group of Jürgen Schmidhuber at the Swiss AI Lab IDSIA have won several international handwriting competitions. In particular, the bi-directional and multi-dimensional Long short-term memory (LSTM) of Alex Graves et al. won three competitions in connected handwriting recognition at the 2009 International Conference on Document Analysis and Recognition (ICDAR), without any prior knowledge about the three different languages (French, Arabic, Persian) to be learned. Recent GPU-based deep learning methods for feedforward networks by Dan Ciresan and colleagues at IDSIA won the ICDAR 2011 offline Chinese handwriting recognition contest; their neural networks also were the first artificial pattern recognizers to achieve human-competitive performance on the famous MNIST handwritten digit problem of Yann LeCun and colleagues at NYU [1].

Most modern tablet pc and hybrid laptops includes an active stylus and display which allows the user to write and draw on the screen. It would be very useful for the users if they can recognize, derive meaning and store the handwritten text into digital text which can be used within computer and text-processing applications. This motivated me to take the first step of implementing a Handwritten Digit Recognizer.

Problem Statement

The goal is to predict the digit handwritten in an image. The dataset contains images of handwritten digits and their respective numbers which can be used to train and test the model. This is supervised learning problem – more specifically a classification problem. The model should be able to classify which number (0 - 9) is handwritten in the image. The model can be scored for its ability to predict the number correctly over large different test data and real data.

The dataset is loaded into python using keras library as discussed in the Data exploration. The loaded data will be first explored and visualized using numpy and matplotlib library to understand the nature of the data. Exploring the data will help us in deciding how to approach and whether any preprocessing of the data is needed. Preprocessing of the data is done as required. Once the data is ready, the Deep neural network(DNN) will be built based on the architecture(ConvNet) as discussed in the Implementation Section. Once the model is built, it will be compiled to check if the architecture has any error. Then the compiled model will be trained on the training data and evaluated using accuracy score against the testing data. Then the results can be analyzed and compared with respect to the benchmark model to know the overall performance of the model. Now we have a trained model, a test is conducted against the model by loading images of digits which are not from MNIST dataset to evaluate the performance of the final model. The images are loaded and then preprocessed to match the MNIST dataset format so that we can test it. The model is then made to predict the digit in the preprocessed image. Thus, we can evaluate our model's performance over real time data.

Metrics

Accuracy is the proportion of samples predicted correctly among the total number of samples examined. For Example, in our problem, it is the ratio of the handwritten digits predicted correctly to the total number of handwritten digits evaluated. It measures the correctness of a model.

$$Accuracy = \frac{Number\ of\ samples\ predicted\ correctly}{Total\ number\ of\ samples\ examined}$$

It ranges from 0 – 1, where 0 indicates worst performance and 1 means best performance.

II. Analysis

Data Exploration

The MNIST database [4] of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.
The original black and white (bi-level) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

The dataset can be downloaded and loaded using Keras, a high-level neural networks python library using the following code

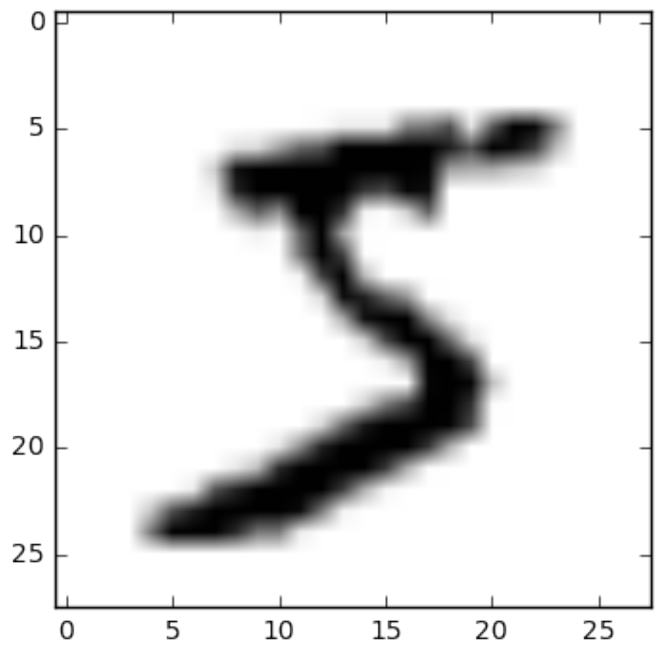
```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

- X_train, X_test: uint8 array of grayscale image split as train and test data
- y_train, y_test: uint8 array of digit labels (integers in range 0-9) split as train and test data

Thus we can use the above variables to train the model with pixel-values of the handwritten image as features(X_train) and its respective label(y_train) as target. We can also test the model by evaluating it against the test variables X_test and y_test.

Exploratory Visualization

Let's visualize an image of a digit in the training set of MNIST dataset. The image contains 28x28 pixels as represented in the image below. The image is a grayscale image i.e. it contains only one color channel whose values range from 0 – 255, where 0 represents white and 255 represents black in this case.

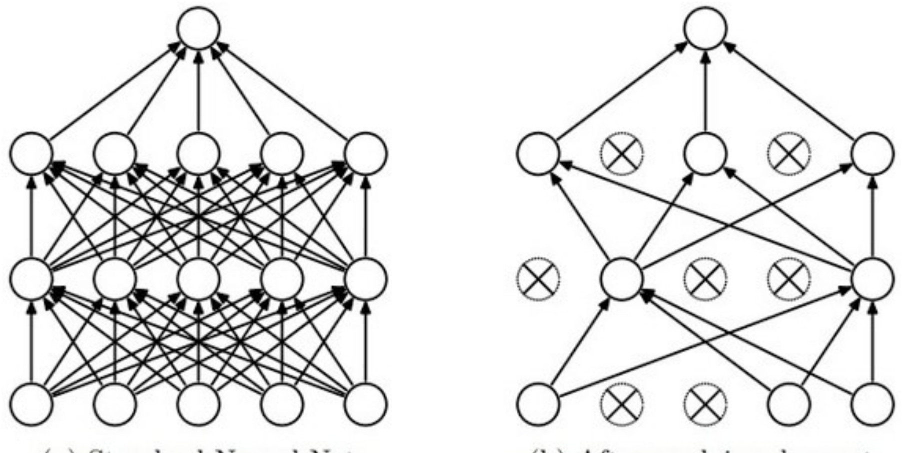


Algorithms and Techniques

Given that the problem is a supervised learning problem and more specifically a classification problem, there are lot of algorithms available for training a classifier to learn from the data. The algorithm chosen for this project is Deep Neural Network(DNN) using Convolutional Neural Network(ConvNet).

Convolutional neural network (CNN, or ConvNet) is a type of feed-forward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex [5]. Since it models animal visual perception, it can be applied to visual recognition tasks such handwritten digit recognition from images. The below are the common types of layers to build ConvNet or DNN architectures:

- **Convolutional Layer** - The input to a convolutional layer is a m x m x r image where m is the height and width of the image and r is the number of channels, e.g. an RGB image has r=3. The convolutional layer will have k filters (or kernels) of size n x n x q where n is smaller than the dimension of the image and q can either be the same as the number of channels r or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce k feature maps of size m−n+1. [6]
- **ReLU Layer** - The Rectified Linear Unit apply an elementwise activation function, such as the max (0, x) thresholding at zero.
- **Pooling Layer**-perform a down sampling operation along the spatial dimensions (width, height)
- **Fully-Connected or Dense Layer** - compute the class scores (between 0-9 digits). As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume. [7]
- **Dropout** is a technique for dealing with overfitting in neural networks. The key idea is to randomly drop units (along with their connections) from the neural network during training as shown in the diagram below. [8]



(a) Standard Neural Net (b) After applying dropout.

- **Flatten** - Flattens the input. Does not affect the batch size. Consider the example code below.[9]

```
model = Sequential()
model.add(Convolution2D(64, 3, 3, border_mode='same', input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)
model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

  After the convolutional layer the output shape is (None,64,32,32). Now when flatten is applied to the layer, the output shape is flatten by multiplying 64*32*32 = 65536.
- **Softmax layer** [16] is typically the final output layer in a neural network that performs multi-class classification (for example: object recognition). the Softmax classifier gives a slightly more intuitive output (normalized class probabilities) and also has a probabilistic interpretation that we will describe shortly. In the Softmax classifier, the function mapping $f(x_i; W) = Wx_i$ stays unchanged, but we now interpret these scores as the un-normalized log probabilities for each class and replace the *hinge loss* with a **cross-entropy loss** that has the form:

$$L_i = -\log\left(\frac{e^{f_i}}{\sum_j e^{f_j}}\right) \text{ or equivalently } L_i = -f_{yi} + \log\sum_j e^{f_j}$$

  where we are using the notation f_j to mean the j-th element of the vector of class scores f. As before, the full loss for the dataset is the mean of L_i over all training examples together with a regularization term $R(W)$. The function $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ is called the **softmax function**: It takes a vector of arbitrary real-valued scores (in z) and squashes it to a vector of values between zero and one that sum to one. The full cross-entropy loss that involves the softmax function might look scary if you're seeing it for the first time but it is relatively easy to motivate.

The ***cross-entropy*** (loss function)[16] between a "true" distribution p and an estimated distribution q is defined as:

$$H(p,q) = -\sum_x p(x)\log\ q(x)$$

The Softmax classifier is hence minimizing the cross-entropy between the estimated class probabilities ($q = e^{f_{y_i}} / \sum_j e^{f_j}$ as seen above) and the "true" distribution, which in this interpretation is the distribution where all probability mass is on the correct class (i.e. $p = [0, \ldots 1, \ldots, 0]$ contains a single 1 at the y_i position). Moreover, since the cross-entropy can be written in terms of entropy and the Kullback-Leibler divergence as $H(p,q) = H(p) + D_{KL}(p||q)$ and the entropy of the delta function p is zero, this is also equivalent to minimizing the KL divergence between the two distributions (a measure of distance). In other words, the cross-entropy objective *wants* the predicted distribution to have all its mass on the correct answer.

Stochastic Gradient Descend(SGD) Optimizer

In Gradient Descent(GD) optimization, we compute the cost gradient based on the complete training set; hence, we sometimes also call it *batch GD*. In case of very large datasets, using GD can be quite costly since we are only taking a single step for one pass over the training set -- thus, the larger the training set, the slower our algorithm updates the weights and the longer it may take until it converges to the global cost minimum (note that the SSE cost function is convex).

In Stochastic Gradient Descent (SGD; sometimes also referred to as *iterative* or *on-line* GD), we don't accumulate the weight updates as we see below for GD:

```
• for one or more epochs:
    ▪ for each weight $j$
        ◦ $w_j := w + \Delta w_j$, where: $\Delta w_j = \eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$
```

Instead, we update the weights after each training sample:

```
• for one or more epochs, or until approx. cost minimum is reached:
    ▪ for training sample $i$:
        ◦ for each weight $j$
            ◦ $w_j := w + \Delta w_j$, where: $\Delta w_j = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$
```

Here, the term "stochastic" comes from the fact that the gradient based on a single training sample is a "stochastic approximation" of the "true" cost gradient. Due to its stochastic nature, the path towards the global cost minimum is not "direct" as in GD, but may go "zig-zag" if we are visualizing the cost surface in a 2D space. [17]

Adaptive Moment Estimation Optimizer (Adam)

Adam[15] computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. $\beta_1$and $\beta_2$are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

An example of a Convolutional Neural Network is given below [7].

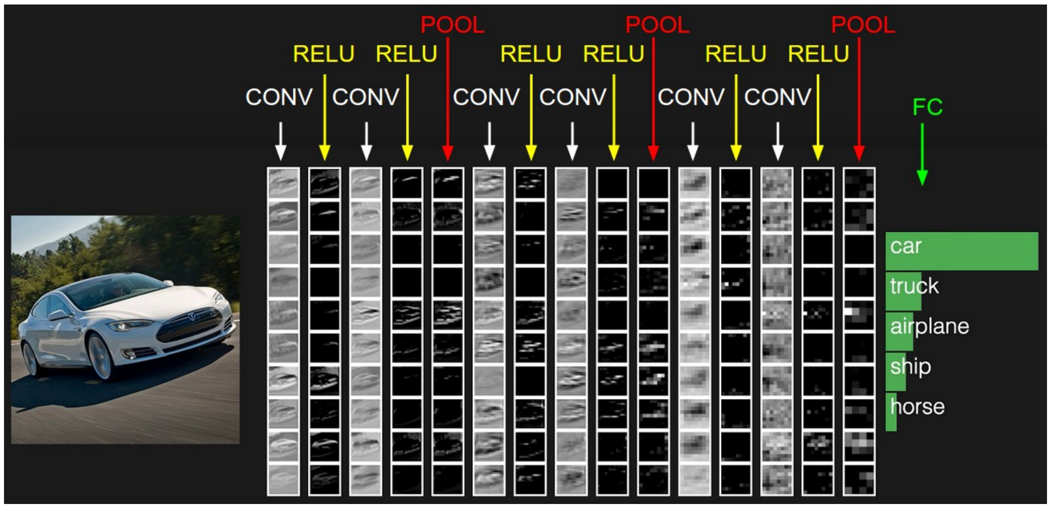


Fig 1 – An example of Convolutional Neural Network

Benchmark

The benchmark model is chosen from one of the kernels of the Kaggle Digit Recognizer competition [11]. The benchmark model solves the same handwritten digit recognition problem mentioned in this project by using a Random Forest classifier algorithm. It uses the same dataset used in this project. Thus making it a perfect benchmark model for this project. The result of the benchmark model is the accuracy score of the model. The same result can be measured for our model by calculating the accuracy score. Accuracy score of the benchmark model is 0.872.

III. Methodology

Data Preprocessing

Since Keras library can be used to download and then load the MNIST dataset directly as numpy arrays, there is only one preprocessing step needed. The MNIST dataset contains grayscale images where the color channel value of each pixels varies from 0 to 255. In order to reduce the computational load, we will map the values from 0 - 255 to 0 - 1 by dividing each pixel values by 255.

Implementation

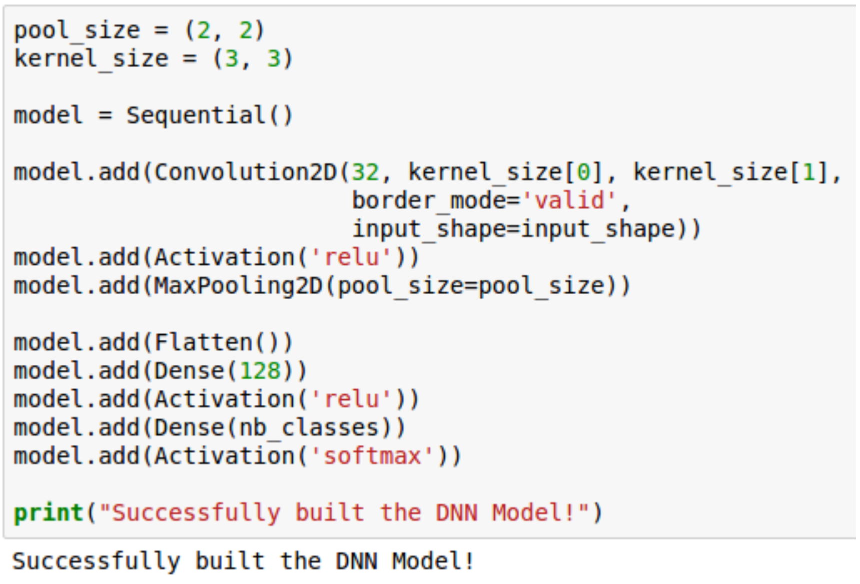
The implementation process can be split into two main stages:

1. Model training and evaluating stage
2. Model testing on real data

Model training and evaluating stage

During the first stage, the model was trained on the preprocessed training data and evaluated against the test data. The following steps are done during the first stage:

1. Load both the training and testing data into memory and preprocess them as described in the above section
2. Define the initial network architecture and training parameters as shown in the code and block diagram below.



```
pool_size = (2, 2)
kernel_size = (3, 3)

model = Sequential()

model.add(Convolution2D(32, kernel_size[0], kernel_size[1],
                        border_mode='valid',
                        input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))

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

print("Successfully built the DNN Model!")
```

```
Successfully built the DNN Model!
```

Fig – 2 Initial Model Code

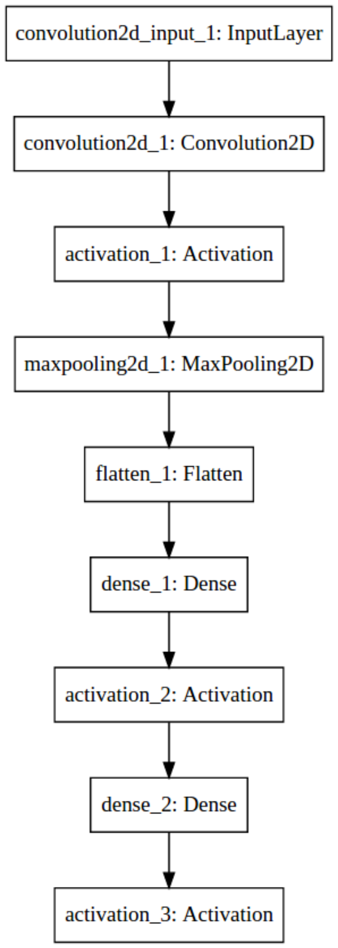


Fig – 3 Initial Model Block Diagram

3. Define the loss function initially as Stochastic Gradient Descent and metrics as Accuracy
4. The network is trained on the training data and evaluated against the test data
5. Note down the loss and accuracy score

6. If the accuracy is not high enough, repeat from step 2 by altering the architecture of the network, changing the hyper-parameters value and trying different loss function

In the first stage, step 2,3 was difficult implementing since finding the optimum architecture, hyper-parameters and loss function was time consuming because a single run takes 10-20 mins.

Initial Model Description

The first layer of the model is the Input Layer which gets the input data as arrays during the training phase. The next layer is the convolutional layer with 32 filters and 3,3 kernel size which convolutes the input image. The next layer is an activation Layer made of Rectified Linear Unit (ReLU) which acts as an elementwise activation function. The next layer is the max pooling layer where down sampling is carried out. Then a flatten layer is used to flatten the data. Then A dense or fully connected layer of size 128 is applied followed by an activation layer. Afterwards a dropout layer is used to cut off some neurons in order to avoid overfitting of the model. The next layer is fully connected layer of size 10 i.e. it is the output size followed by a softmax activation layer. The output of the final activation layer is the final output. Once the model is built using the above architecture, it is compiled by specifying the loss function as categorical cross entropy, optimizer as SGD(Stochastic Gradient Descent), and metrics as accuracy. Once the model is complied with no errors, it is trained over the training dataset X_train and Y_train. The model is then evaluated against the test dataset X_test and Y_test. The accuracy score of the initial model is measured to be 0.9582 i.e. 95.82%.

Model testing on Real Data

During the second stage, the final model is chosen and is tested against real data rather than the MNIST dataset itself. The following steps are involved in this stage:

1. Download or take photos of handwritten digits
2. Load the images into the program from a folder
3. Format the images in order to match with MNIST Dataset i.e. 28x28 pixel grayscale image
4. Make the model predict the new images and verify the performance and robustness of the model.

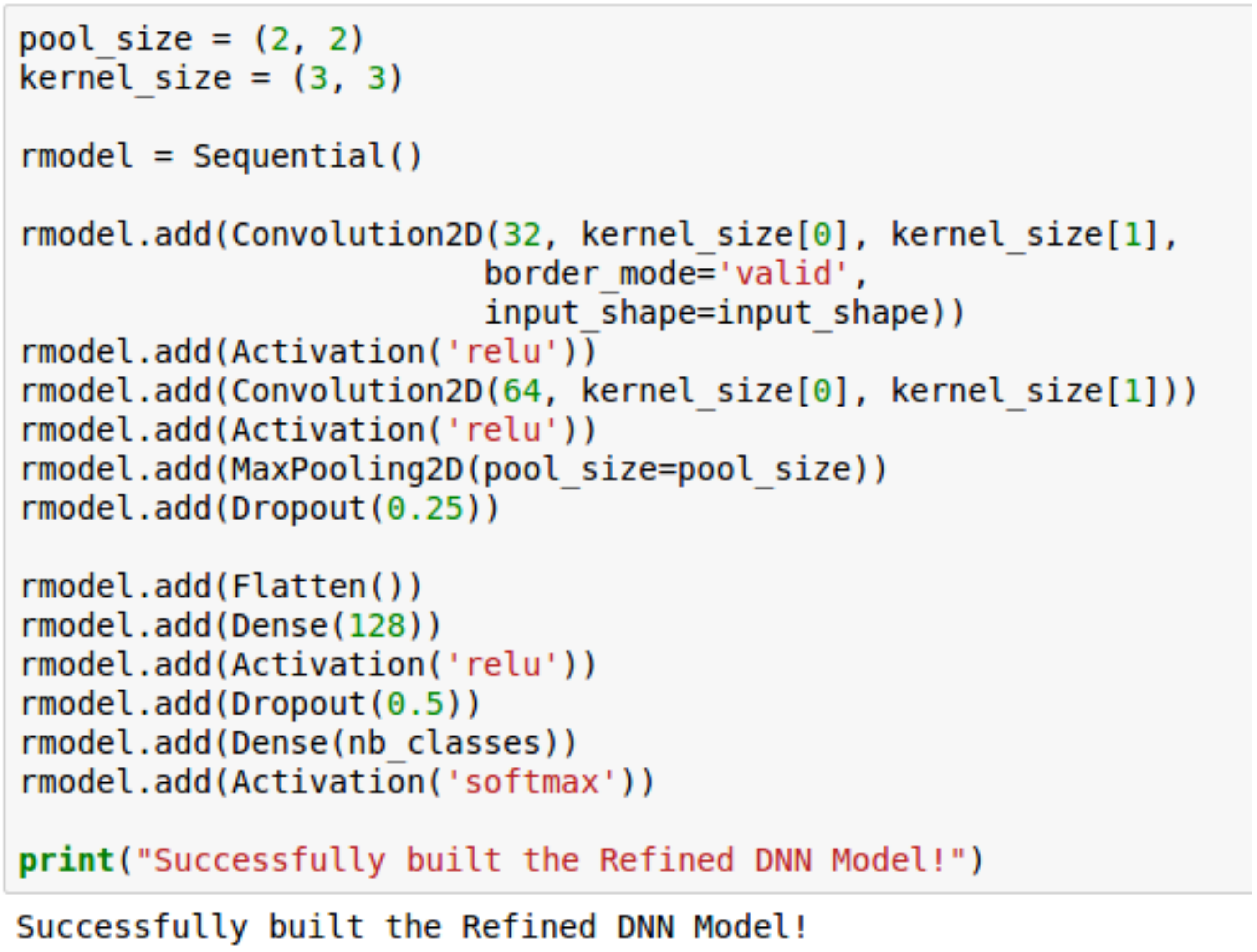
In the second stage, step 3 was a bit tough as OpenCV (advanced image processing) library is downloaded and setup for just two operations(resizing and converting to grayscale format)

Refinement

The initial model is improved by using the following technique

- It is made further deep by adding two more convolutional layer and one more extra dense (fully connected) layer.
- Two dropout layer is added to ensure that overfitting does not happen
- Using Adam optimizer instead of SGD(Stochastic Gradient Descent) to update the weights efficiently.

The refined CNN architecture and code of the model is shown below.



```
pool_size = (2, 2)
kernel_size = (3, 3)

rmodel = Sequential()

rmodel.add(Convolution2D(32, kernel_size[0], kernel_size[1],
                        border_mode='valid',
                        input_shape=input_shape))
rmodel.add(Activation('relu'))
rmodel.add(Convolution2D(64, kernel_size[0], kernel_size[1]))
rmodel.add(Activation('relu'))
rmodel.add(MaxPooling2D(pool_size=pool_size))
rmodel.add(Dropout(0.25))

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

print("Successfully built the Refined DNN Model!")
```

```
Successfully built the Refined DNN Model!
```

Fig – 4 Refined Model Code

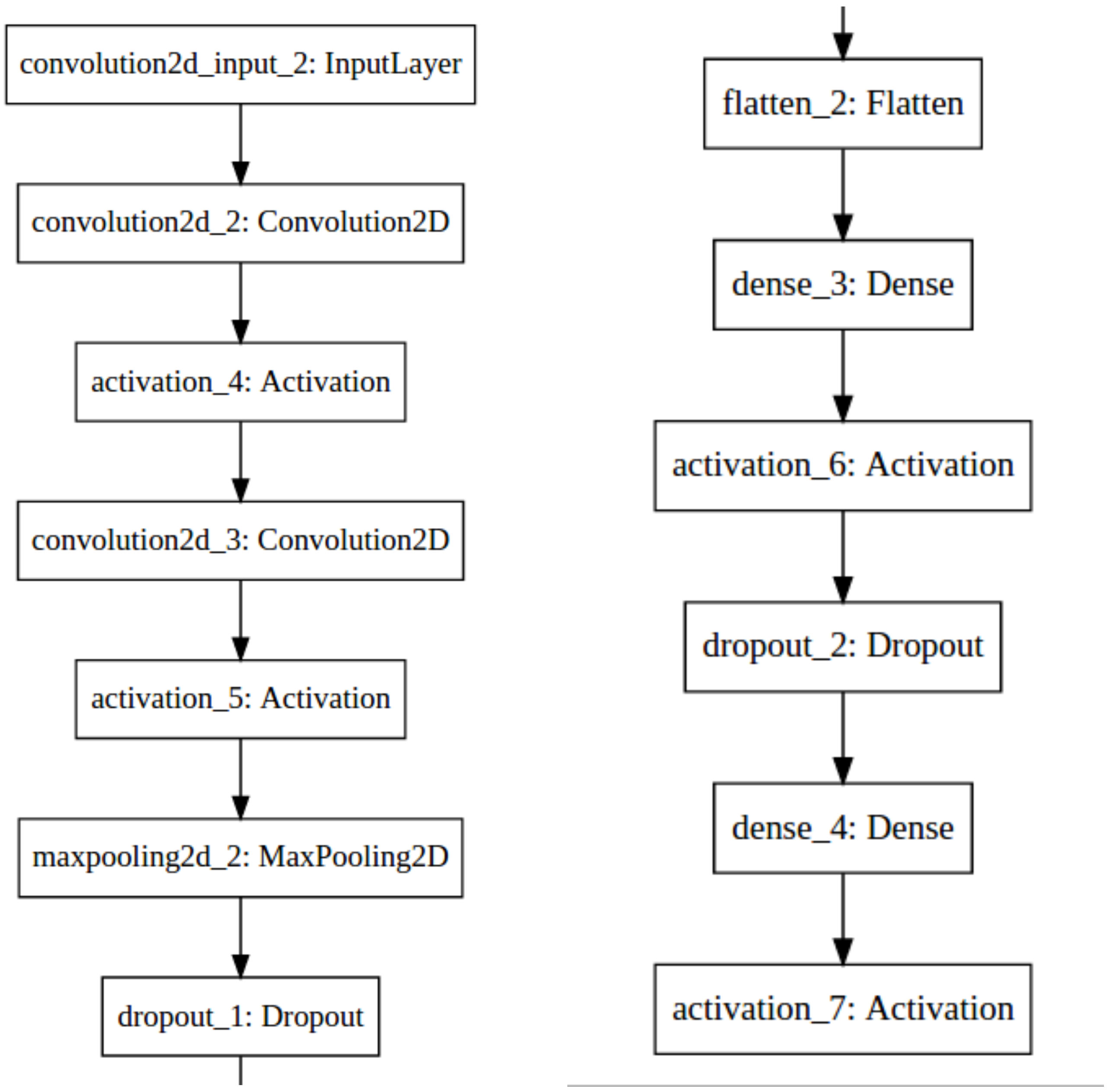


Fig – 5 Refined Model Block Diagram

Once the model is built using the above architecture, it is compiled by specifying the loss function as categorical cross entropy, optimizer as Adam, and metrics as accuracy. Once the model is complied with no errors, it is trained over the training dataset X_train and Y_train. The model is then evaluated against the test dataset X_test and Y_test.
The accuracy score for refined model is measured to be 0.9906 i.e. 99.06%.

IV. Results

Model Evaluation and Validation

The model is evaluated against the test set (X_test, Y_Test). The final architecture and hyper-parameters is chosen because they performed best out of the other models tried.
The robustness of the final model is verified by conducting a test against the final model using different images of digits other than MNIST dataset itself as shown in Fig 6.
The following observations are based on the results of the test:

- The final model has predicted the digits of 10 out 11 images correctly.
- Our model had problem in predicting digits that have thin stokes like the digit 4 with thin stroke in the test cases

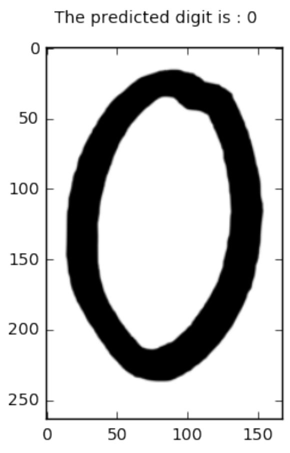
The observations of test confirms that our model is reliable and robust enough to perform well on real data. Use of Dropout layer in the final model ensures that small perturbations (changes) in training data or the input space does not affect the results greatly.

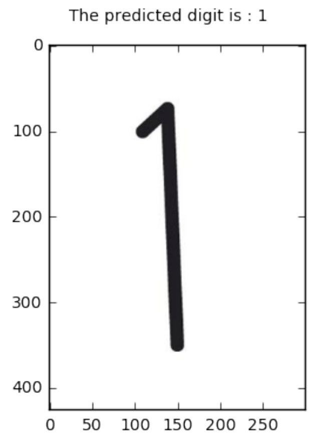
Justification

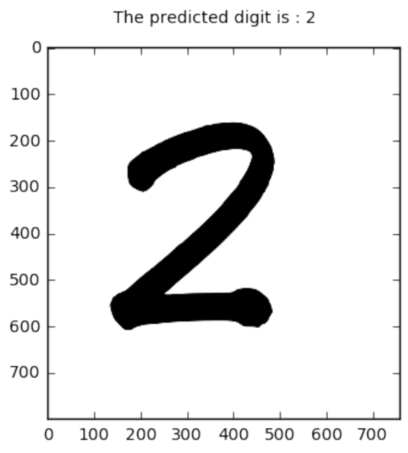
The accuracy of the final model is 99.19% whereas the benchmark model's accuracy is 87.2%. Our model has clearly outperformed the benchmark model. Although, in order to justify that our model has potentially solved the problem, we need to look at the results of the real time test conducted against model as shown in Fig 6. Our model can correctly predict 10 out of 11 tests. Thus, our model solved the problem with a good accuracy score and reliability in real time applications.

V. Conclusion

Free-Form Visualization







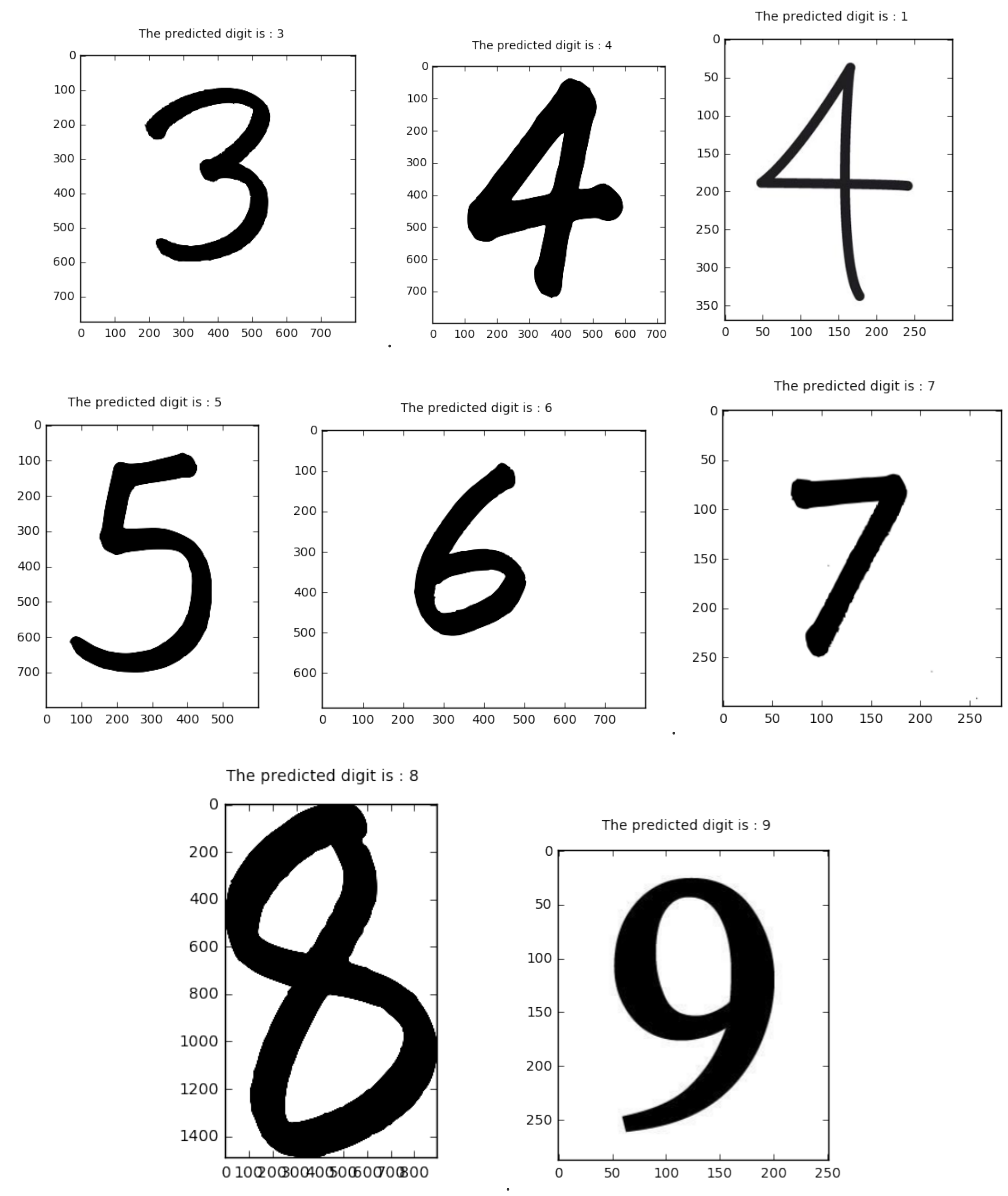


Fig – 6 Results of test conducted by loading images of digits which are not from MNIST dataset to evaluate the performance of the final model.

From the above results, we can see that our final model has predicted the digits of 10 out 11 images correctly. Thus our model has a good accuracy score of 99.19% and good reliability as seen in the above test.

Reflection

The process involved in this project can be summarized using the below steps:

1. An initial problem and a relevant public dataset were found
2. A benchmark model is chosen for the problem
3. The dataset was downloaded and loaded in the code
4. The data is preprocessed as required
5. A deep neural network model using convolutional neural network is built
6. The model is trained using the training dataset and evaluated against the test dataset multiple times in order to find an optimum model architecture with good hyper-parameters
7. A test is conducted against the refined model by loading images of digits which are not from MNIST dataset to evaluate the performance of the final model.

I found steps 5 and 6 to be tough as I was new to Deep Learning and it took a while for me to grasp the concept. However using the MNIST dataset and keras library simplified my process a lot. MNIST Dataset was a good choice for a newbie in deep learning like me as there was lot of tutorials and guides available using MNIST dataset which helped in understanding them easily. Also Keras helped me in kick-starting into deep learning without writing complex code which was really helpful in quick prototyping and experimenting in order to build a better model.

Improvement

The final model in this project uses only simple Convolutional Neural Network architecture. There are more advanced architecture like the Deep Residual Neural Network(ResNet)[12] and VGG-16[13] which could be used to improve the accuracy a little. Also we could hyperas[14] library to fine tune the hyper-parameters and architecture of the model.

References

1. https://en.wikipedia.org/wiki/Handwriting_recognition
2. http://www.ehow.com/list_7353703_benefits-advantages-handwriting-recognition.html
3. http://www.cse.unsw.edu.au/~billw/cs9444/crossentropy.html
4. http://yann.lecun.com/exdb/mnist/
5. https://en.wikipedia.org/wiki/Convolutional_neural_network
6. http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/
7. http://cs231n.github.io/convolutional-networks/
8. https://www.quora.com/What-is-dropout-in-deep-learning
9. https://keras.io/layers/core/#flatten
10. https://www.quora.com/What-are-softmax-layers

11. https://www.kaggle.com/romanroibu/digit-recognizer/random-forest-digit-classifier
12. https://arxiv.org/abs/1512.03385
13. http://www.robots.ox.ac.uk/~vgg/research/very_deep/
14. https://github.com/maxpumperla/hyperas
15. http://sebastianruder.com/optimizing-gradient-descent/
16. http://cs231n.github.io/linear-classify/
17. https://www.quora.com/Whats-the-difference-between-gradient-descent-and-stochastic-gradient-descent/answer/Sebastian-Raschka-1?srid=hhRTP