MINST Handwriting Database

Nathaniel Linden
March 2018 – UW Bioengineering

Abstract

In recent years, neural networks have taken over the data science community, and shaped the focus towards a world of big data. Requiring massive datasets and enormous computing resources, neural networks provide the potential for scientists to build extremely accurate predictive models. In this paper, I will outline the basic architecture of these somewhat infamous computational tools and I will explain the basic mathematical concepts behind their functionality. I will then construct to neural networks and analyze the performances when trained on the classic MNIST dataset.

Introduction

Contemporary computing has been a field of ongoing rapid development and innovation, evolving with both the data landscape and the availability of increasingly powerful computers. In the last decade the field has seen a rise in the application and development of what is known as a neural network. First proposed in biological systems by Hubel and Wiesel[1] when they studied neuronal computation in the feline brain, neural networks, in both biology and mathematics, allow an input to be mapped to an output via a complex computation network.

One might ask what has driven the recent innovations that have lead data science to modern computational networking? In the 1980's the concepts were merely proposed biologically and hinted towards computationally. However, as mathematics moved into the late 1990s research into neural networking began to gain impact and see more efforts. However, due to limitations into data size, computing power, and execution approach, neural networks saw a major stagnation in research efforts, leading to an abandonment of field. By 2008, when the IEEE ICDM[2], published a survey of the 10 best algorithms for data mining and analysis, neural networks received no coverage and was of no interest. Despite this steep decline in neural network focus, a mere decade later, the concept of the computational network deep-learning dominates the data mining space.

In the following pages, I will introduce the MNIST Handwriting database as a tool to understand the concept of neural networks and as a sample to dataset to construct and train two simple networks. The MNSIT database is a collection of 70,000 handwritten digit samples, complied from a collection of National Institute of Standards and Technology data[3]. This easily labeled, and intuitively interpreted dataset allows discussion and interpretation of neural network analysis.

Theoretical

[1] Hubel and Wiesel won the Nobel Prize in 1981 for their proposition of the biological neural network as observed in the feline brain.

[2] IEEE International Conference on Data Mining

[3] Originally compiled by Yann LeCun in conjunction with New York University - http://yann.lecun.com/exdb/mnist/

A common question by those who have no formal data science training is: *What is a neural network?* Simply, a neural network is an algorithm which takes an input, shoves it into a large mathematical formula, and returns an intuitive output. In the case of the MNIST dataset, that input will be an image of a handwritten digit (0-9) and the output will be a numerical variable identifying which digit this is. Figure 1[4] provides a simple illustration of a generic neural network, where the inputs X are mapped by some hidden layers to the outputs Y. In data science speak, the output layer is often referred to as the perceptron layer, meaning what the humans actually perceive, the labels. This notation dates back to the 1950s when the first theories of machine classification were proposed. In reality, a neural network will have a more complex structure encompassing many hidden layers and possibly high dimensional input and output layers.

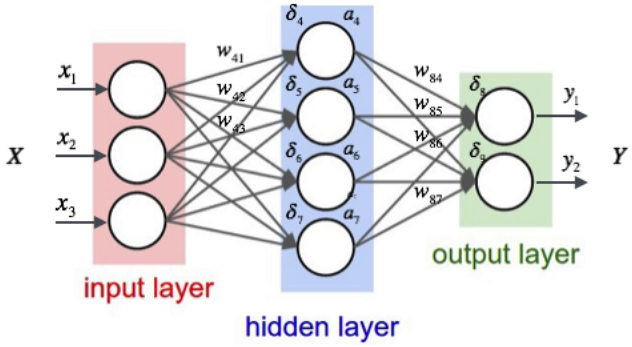

Figure 1 Generic Neural Network.

Now that the general structure of the neural network is understood, it is important to notice how the network actually maps from inputs to outputs. In order to simplify this process, consider the following one-layer network highlighted in figure 2[5].

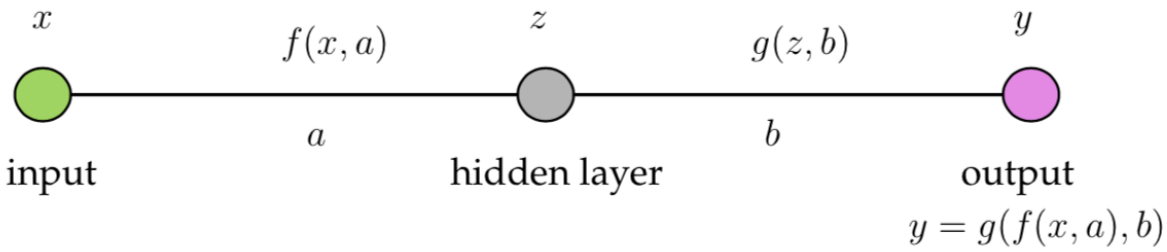

Figure 2 A simple one- layer neural network. Includes activation function examples and compositional nature of the network.

In this network, the input x is mapped to z through the function, $z = f(x, a)$ where a is matrix of parameters. From the hidden layer, z, the data is mapped to the output layer through $y = g(z, b)$. Thus, in order to get from the input layer to the output layer, the following function is defined: $y = g(f(x, a), b)$. It is quite simple to observe the compositional nature of this function, where t output depends on some function, which is a function of the input. In correct notation, these

[4] https://cdn-images-1.medium.com/max/479/1*QVIyc5HnGDWTNX3m-nIm9w.png

[5] Brunton & Kutz 2017.

functions are known as activation functions which provide the proper weighting on the inputs to map between hidden layers and the outputs. In a more genera neural network this function can be elaborated as: $Output = f_1(f_2(f_3 \dots f_n(x_n, a_n), a_3), a_2), a_1)$. This function provides the full mapping from the input layer to the output layer.

Next, it is logical to ask how the vectors, a_n are actually found in order to begin building an actual network. Generally, this step is known as the training step and it is usually executed as a large optimization problem. Since the network function is a compositional function one can execute an algorithm known as the backpropagation algorithm, to determine the exact weights for the network. First it is key to define an error function, for this I will use an $L_2\ norm$ error function defined as $E(error) = \|y_0 - y\|_2 = \frac{1}{2}(y_0 - y)^2$ where y_0 is the true labels of the data and y is the prediction generated by the network and the ½ is added for ease of calculation. Then we will want to minimize the error with respect to all of our input parameters, by differentiating and setting to zeros. So, for the simple network of figure 2:

$$\frac{\partial E}{\partial a} = -(y_0 - y)\frac{dy}{dz}\frac{dz}{da} = 0$$

Furthermore, the error with respect to b will need to be minimized as well, virtually minimizing the gradient of the error function. Following this, the backpropagation algorithm results in an optimization scheme known as gradient descent where the parameters are iteratively updated as follows:

$$a_{k+1} = a_k + \tau\frac{\partial E}{\partial a_k}$$

$$b_{k+1} = b_k + \tau\frac{\partial E}{\partial b_k}$$

Where τ is known as the learning rate. It is this iterative optimization, known as gradient descent that defines the training of neural network. Furthermore, it is this step that defines the computational complexity of training a neural network, because for very large realistic networks there are many small optimizations to perform for each training iteration and for each layer.

Now that it is clear how training of a neural network is performed it is important to understand the different choices for the hidden layers and the functions surrounding them. The hidden layer can either be simple, defined by a simple function known as the activation function, or more complex like a convolutional layer. The simple layers take the input and give an output from the defining activation function. Today there are many choices for these activation functions, some examples are ReLU (rectified linear unit), softmax, sigmoid, and tanh. In the networks trained for this paper, ReLU and softmax and sigmoid were used most dominantly so the functions for are are provided below:

$$ReLU: f(x) = \max(0, x)$$

$$softmax: \sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}\ for\ j = 1 \dots K$$

$$sigmoid: S(x) = \frac{e^x}{e^x + 1}$$

And the graphs of a sigmoid and a ReLU can be found in figure 3. These two functions are both

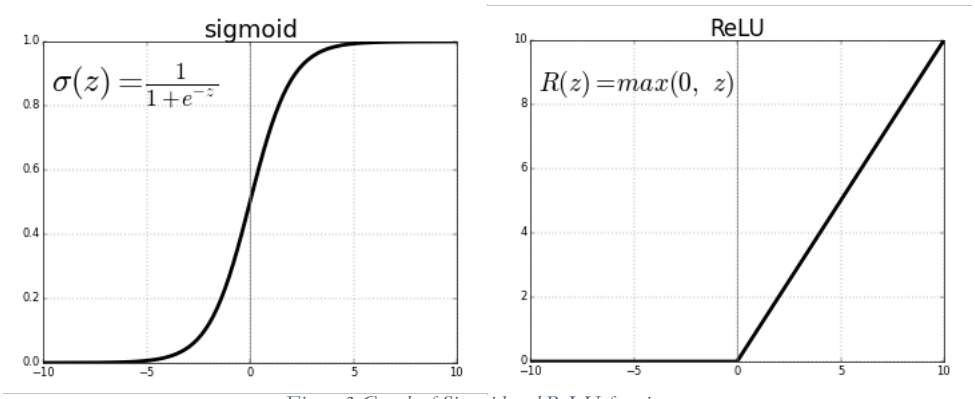

Figure 3 Graph of Sigmoid and ReLU functions.

typical activation functions used in generation of neural networks.

More complex layers, such a convolutions and pooling layers can be added to the network. Simply, a convolutional layer takes an input, usually an image, that is n x n pixels, and takes a smaller m x m window and scans across the input space. It then maps the features from these scanning regions to a high dimensional space. A flattening layer will map from the higher space back to the original dimensions. A simplified visual representation of convolutions and max pooling layers can be found in figure 4.

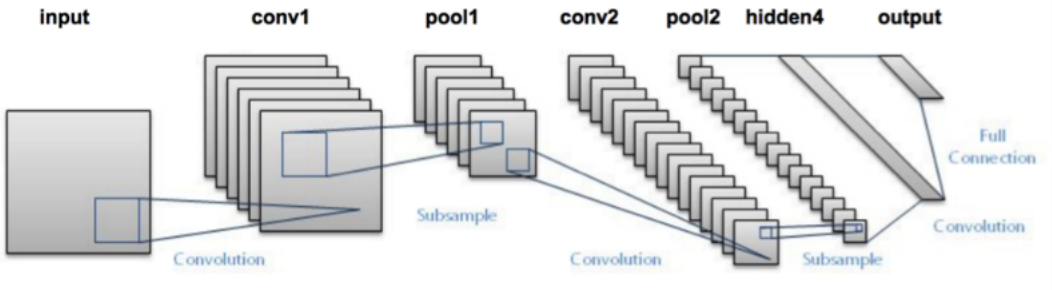

Figure 4 Convolutional Network example.

Furthermore, a pooling layer takes a larger layer and pools the max or min from a subspace into a smaller layer, this can also be observed in figure 4.

Lastly, it is key to understand some of the metrics used to measure neural network performance and training. First the simplest metric is the predictive accuracy of the network on some testing set. When the network is trained, a testing set is plugged in and the accuracy is measured by the percentage of correct predictions. Second, the loss, is another typical metric used in this paper to measure network convergence. Essentially the loss function is the error function used to perform the optimization and the loss in simply this error.

Now that I have outlined the basic structure and concepts behind neural networking I will outline the two networks generated in this paper and explain the MNIST dataset in more details. First it is important to understand the dimensionality of the MNSIT data, because it will be referenced frequently. The dataset contains 70,000 images of handwritten digits, 60,000 of which were used from training and 10,000 of which were used for testing. Each image is 28 x 28 pixels, so it spans 748 unique dimensions. Thus, the input matrix is 60000 x 784 and the perceptron layer is 60000 x 10, where a 1 in each column of the output represents a digit 0-9.

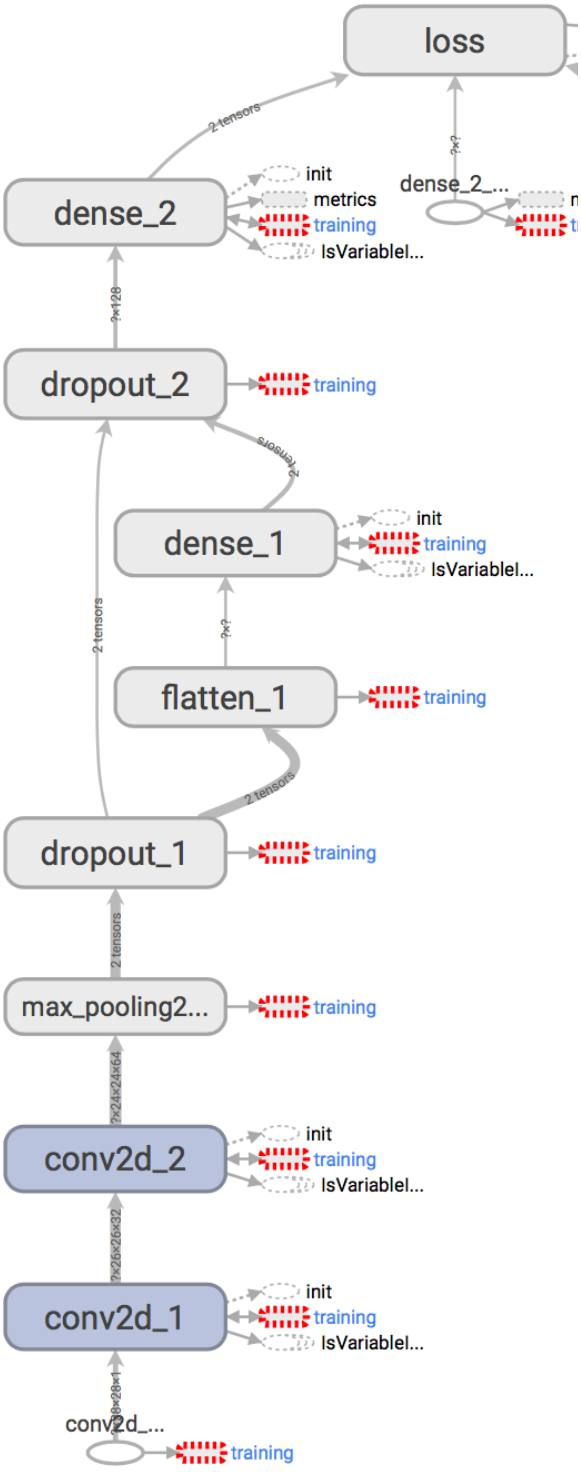

Figure 5 Convolutional Neural Network Schematic.

The first network created is a simple two-layer feedforward network that can be visualized by the chart in figure 5. This network contains two ReLU activated layers, known as Dense1 and Dense2 layers in the figure, two dropout layers and a softmax layer, Dense3 to map from the hidden layers to the outputs. I used a stochastic gradient descent optimization scheme with parameters around what was found online. And I trained for 50 epochs, iterations.

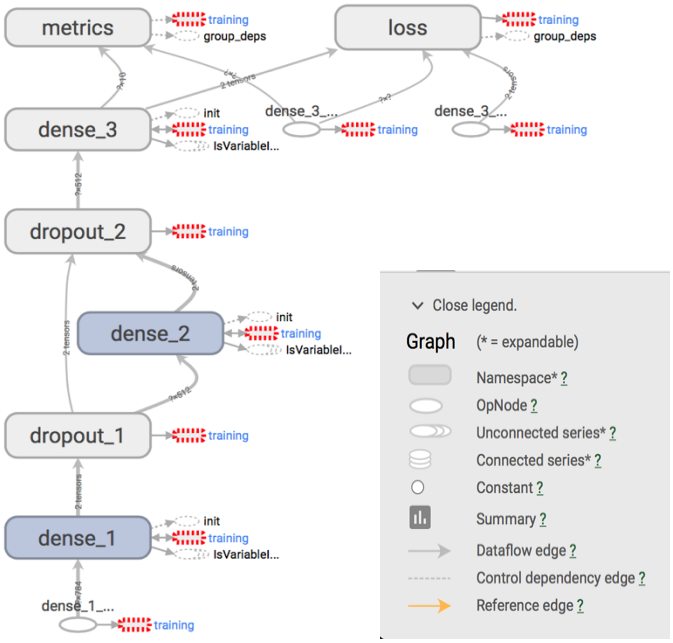

Figure 6 Simple 2-Layer Network Schematic.

The second network created is a more complex convolutional neural network. There

are two convolutional layers, a maxpooling layer which is then flattened and passed through a ReLU and a softmax function to map to the perceptron. This is optimized using a scheme known as Adadelta which adapts the gradient descent learning rate over time. This was also trained for a total of 50 epochs. The network schematic can be found in figure 6.

Computational Results

Both networks were trained and tested on the same MNIST data and the raw ending accuracies can be found in the following table, along with the total runtimes.

Network Type	Final Accuracy (%)	Total Train-Time (hr:min)
Simple 2-Layer Network	98.4	0:03
Convolutional Network	99.3	6:05

Table 1 Training accuracy and total runtime.

The loss overtime and the accuracy overtime curves can be found in figures 7 and 8 respectively.

Overall the convolution network is observed to do marginally better, with an overall validation accuracy of 99.3 % while the simpler network has an accuracy of 98.4 %. The improved accuracy comes as a sacrifice of training time, as for 50 epochs the convolutional network took 6 hrs to train while the simple network only too 3 minutes.

Summary and Conclusions

After exploring the architecture and basic mathematics of the neural network and applying the concepts to a real-world dataset the power and complexity of such a tool can be realized. As a predictive model for large scale datasets, neural networks are extremely useful and provide the potential to achieve extremely high rates of accuracy. As seen in the MNIST example, even the very simple neural network can achieve accuracies of around 98 percent. This is very promising and fairly surprising given the training time of only three minutes. While the simple network preforms well, the more complex convolutional network pushed the accuracy above the 99 percent margin to achieve even better results.

This increased performance comes at the sacrifice of efficiency, both temporally and computationally. In order to train a very complex and large neural network it takes enormous computing resources and the time availability to allow a network to train for weeks. As neural networks are applied to more and more real-world problems, large corporations are investing in massive computing clusters researching more efficient training methods. All in all, neural networks are an extremely useful tool for the analysis and generation of a predictive model of/and based on an extremely large dataset and have played a key role in shaping todays big data-oriented society.

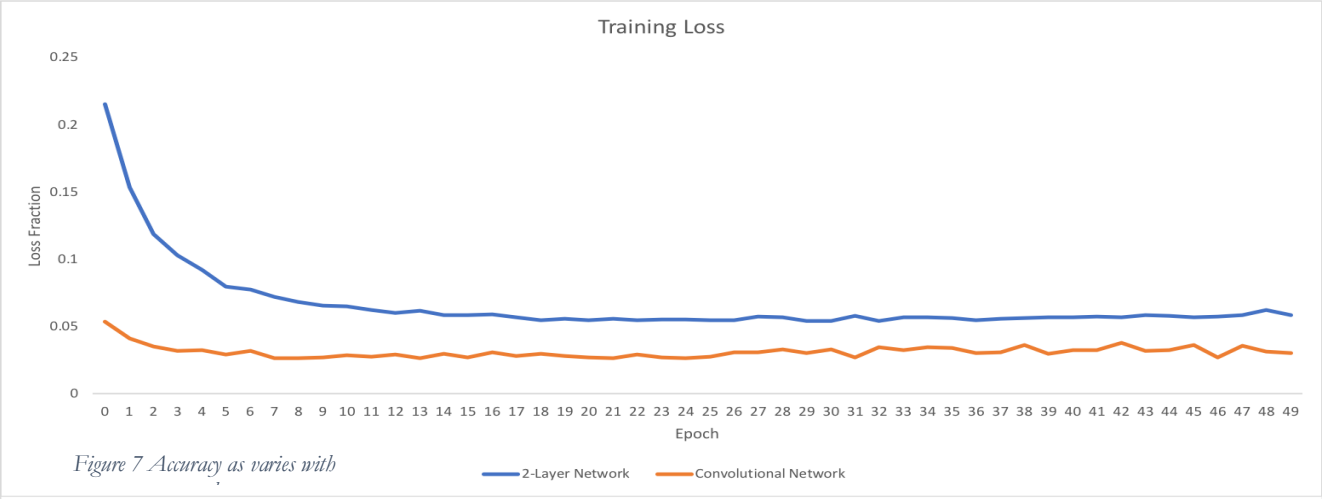

Figure 7 Accuracy as varies with

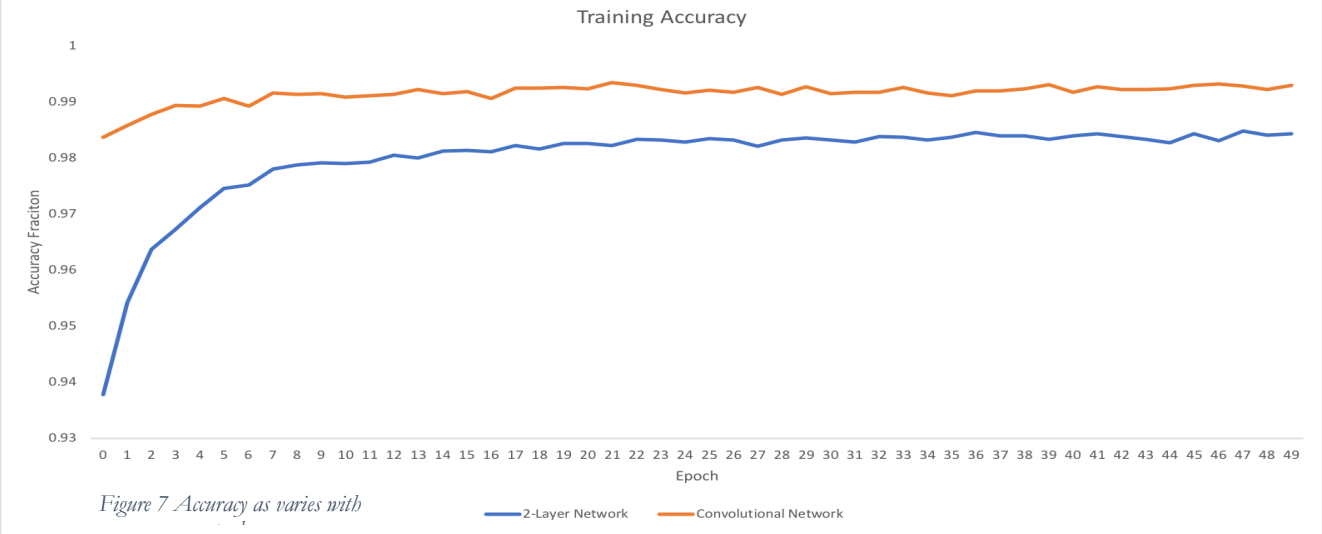

Figure 7 Accuracy as varies with

Appendix A – Python Function Repository

USED KERAS API FOR THIS PROJECT.

Keras documentation can be found here: https://keras.io

Some important neural networking functions used:

- Sqeuntial
  - Creates a feedforward network architecture.
- Add
  - Used to add a layer to the network.
- Compile
  - Compiles the Network
- Fit
  - Does the training
- Evaluate
  - Generates metrics

Appendix B – Python Code-Bank

```python
#imports
import numpy as np
from mnist import MNIST
import os
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, MaxPooling2D, Conv2D, Flatten
from keras import optimizers
from keras import backend as K
from keras.utils import plot_model
from keras.callbacks import TensorBoard
import pydot
import graphviz

#laod the MNIST Data
#downloaded from yann.lecun.com
#use the MNIST library to import images from idx1/3-ubyte data
mndata = MNIST('/Users/natelinden/anaconda3/lib/python3.6/site-packages/mnist/samples/')
Train_images, Train_labels = mndata.load_training()
Test_images, Test_labels = mndata.load_testing()

#convert to numpy array
Train_images = np.array(Train_images)
Test_images = np.array(Test_images)
Train_labels = np.array(Train_labels)
Test_labels = np.array(Test_labels)

# input image dimensions
img_rows, img_cols = 28, 28

# the data, split between train and test sets
if K.image_data_format() == 'channels_first':
    Train_imagesCNN = Train_images.reshape(Train_images.shape[0], 1, img_rows, img_cols)
    Test_imagesCNN = Test_images.reshape(Test_images.shape[0], 1, img_rows, img_cols)
    input_shapeCNN = (1, img_rows, img_cols)
else:
    Train_imagesCNN = Train_images.reshape(Train_images.shape[0],  img_rows, img_cols,1)
    Test_imagesCNN = Test_images.reshape(Test_images.shape[0], img_rows, img_cols,1)
    input_shapeCNN = (img_rows, img_cols,1)
Train_imagesCNN = Train_imagesCNN.astype('float32')
Test_imagesCNN = Test_imagesCNN.astype('float32')
Train_imagesCNN /= 255
Test_imagesCNN /= 255

Train_images = Train_images.astype('float32')
Test_images = Test_images.astype('float32')

Train_images /= 255
Test_images /= 255
#convert labels to categorical vectors
num_classes = 10
Train_labels = keras.utils.to_categorical(Train_labels,num_classes)
Test_labels = keras.utils.to_categorical(Test_labels,num_classes)
```

```
54
55  #create keras model
56  #CNN Model
57  model1 = Sequential()
58
59  model1.add(Conv2D(32, kernel_size=(3, 3),
60                    activation='relu',
61                    input_shape=input_shapeCNN))
62  model1.add(Conv2D(64, (3, 3), activation='relu'))
63  model1.add(MaxPooling2D(pool_size=(2, 2)))
64  model1.add(Dropout(0.25))
65  model1.add(Flatten())
66  model1.add(Dense(128, activation='relu'))
67  model1.add(Dropout(0.5))
68  model1.add(Dense(num_classes, activation='softmax'))
69
70  tbCallBack1 = TensorBoard(log_dir='./Graph1', histogram_freq=0, write_graph=True, write_images=True)
71  tbCallBack1.set_model(model1)
72  model1.summary()
73  plot_model(model1, to_file='model.png',show_layer_names=True, show_shapes=True)
74
75  model1.compile(loss='categorical_crossentropy', optimizer=optimizers.Adadelta(),metrics=['accuracy'])
76  history = model1.fit(Train_imagesCNN, Train_labels,
77      batch_size=128,
78      epochs=50,
79      verbose=1,
80      validation_data=(Test_imagesCNN,Test_labels),
81      callbacks=[tbCallBack1])
82  score1 = model1.evaluate(Test_imagesCNN, Test_labels,verbose=0)
83  print('Test loss:', score1[0])
84  print('Test accuracy: ', score1[1])
85  """OTHER NNN"""
86  #Simple Feed Forward Model
87  model2 = Sequential()
88
89  model2.add(Dense(512,activation='relu',input_shape=(784,)))
90  model2.add(Dropout(0.2))
91  model2.add(Dense(512,activation='tanh'))
92  model2.add(Dropout(0.3))
93  model2.add(Dense(512,activation='relu'))
94  model2.add(Dropout(0.2))
95  model2.add(Dense(num_classes,activation='softmax'))
96
97  tbCallBack2 = TensorBoard(log_dir='./Graph2', histogram_freq=0, write_graph=True, write_images=True)
98  tbCallBack2.set_model(model2)
99  model2.summary()
100 plot_model(model2, to_file='model.png',show_layer_names=True, show_shapes=True)
101 sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
102 model2.compile(loss='categorical_crossentropy', optimizer=sgd,metrics=['accuracy'])
103 history = model2.fit(Train_images, Train_labels,
104     batch_size=128,
105     epochs=50,
106     verbose=1,
107     validation_data=(Test_images,Test_labels),
108     callbacks=[tbCallBack2])
109 score2 = model2.evaluate(Test_images, Test_labels,verbose=0)
110 print('Test loss:', score2[0])
111 print('Test accuracy: ', score2[1])
```