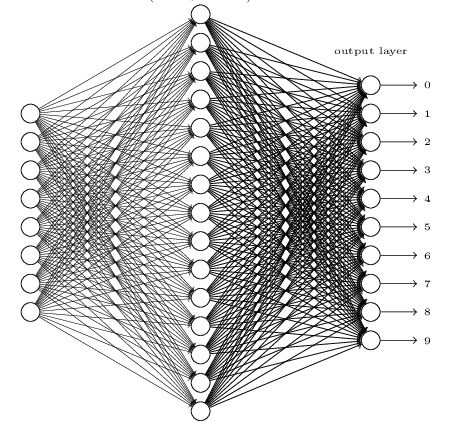
Neural Network

The Neural Network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy.



Input layer. The input layer of the network contains neurons encoding the values of the input pixels. our hand-writing samples in the MNIST data set are 28\*28 greyscale-pixel images, so we’d have 784 neurons in input layer. For every input pixel, the value of 0.0 represents white and the value of 1.0 represents black, and the between values means gradually darkening from 0.0 to 1.0.

The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by *n*, which is our hyper parameter, and we'll experiment with different values for *n*.

Output layer. There are 10 Arabic numerals in total [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Then we need 10 types of output for each number. So the output layer contains 10 neurons, when a sample is classified to a number, the corresponding neuron will be set to 1, the other 9 neurons remain 0, i.e. [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] represents 0.

So each training input *x* as a 784-dimensional vector, each entry in the vector represents the grey value for a single pixel in the image. Assuming the corresponding desired output *y=f(x)*, where *y* is a 10-dimensional vector.

To quantify how well we’re achieving this goal we define a cost function:

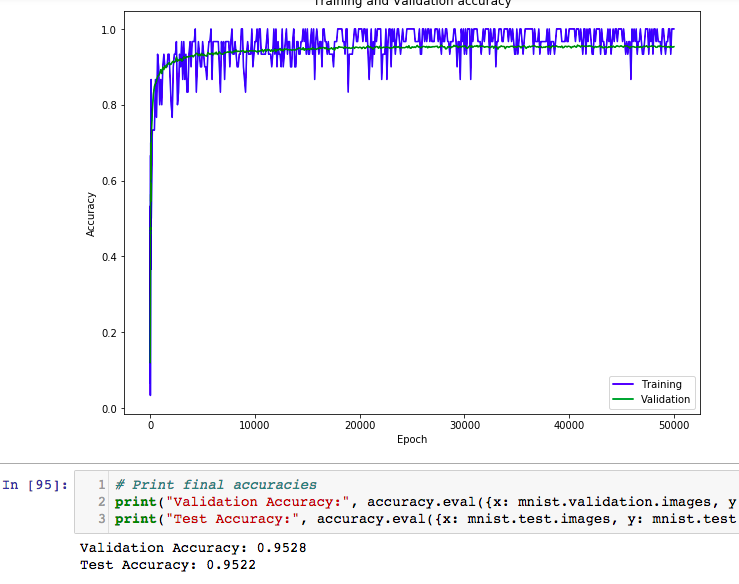
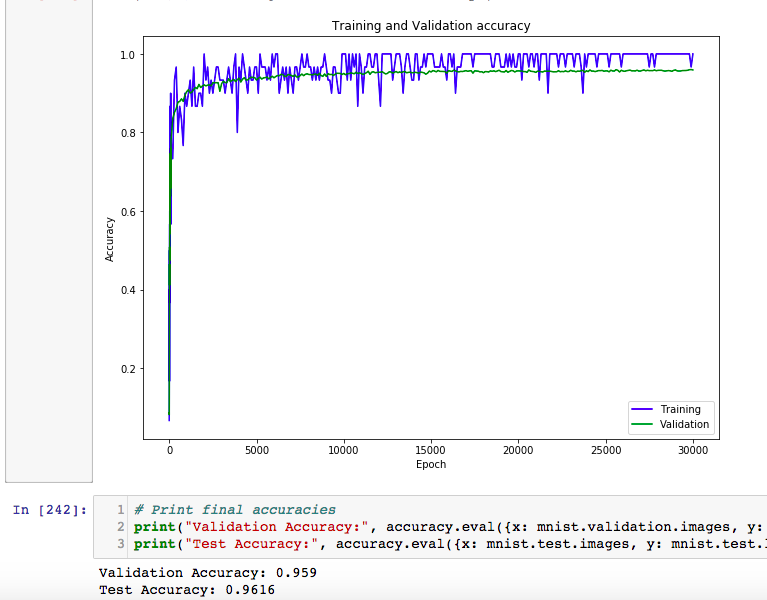
*w* donates the collection of weights in the network, *b* means the biases, *n* is the total number of training inputs, *a* is the vector of outputs from the network when *x* is input, and the sum is over all training inputs *x*.

When *C (w, b)* becomes small, i.e. when *C (w, b)* approaches to 0, *y* is approximately equal to the output *a*, for all training inputs, so our training method has been good. By contrast, it’s not doing well when *C (w, b)* is large. The aim of our training algorithm is minimizing the cost function of weights and biases. In other words, we need to find the set of key parameters (weights and biases) which make the cost function as small as possible.

We’re going to use stochastic gradient descent to find and adjust these two variables:

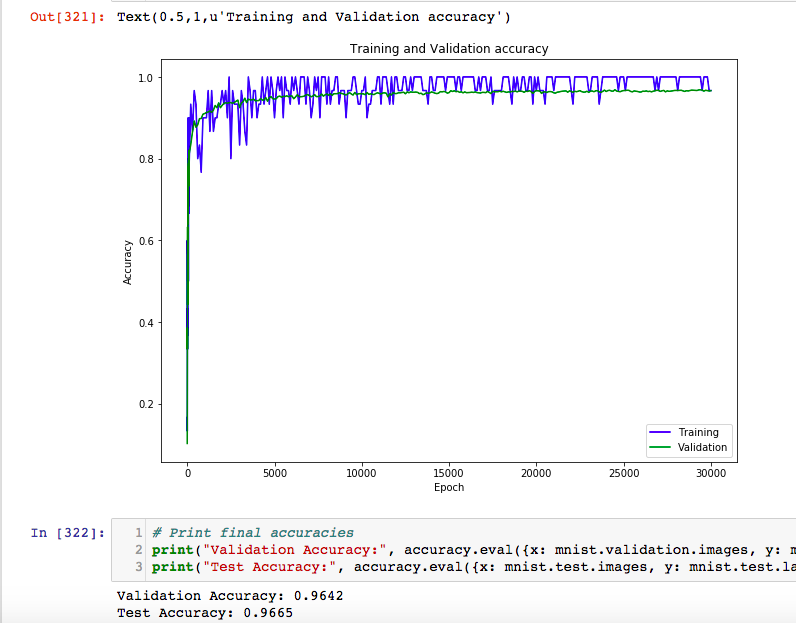
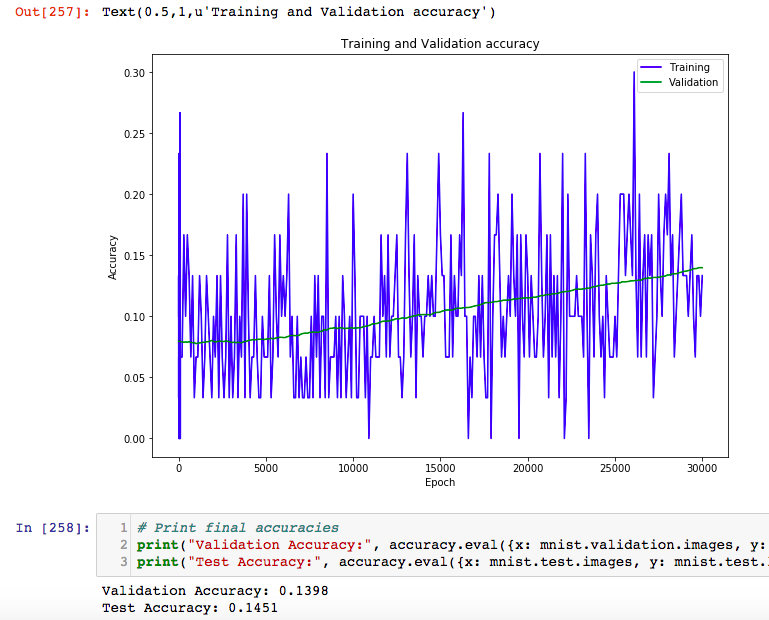
where *η* is the learningrate, *m* is the mini batch size.

Then we’re running the code and tuning the hyper parameters (hidden layer neurons *n*, learning rate *η*, mini batch size *m*).

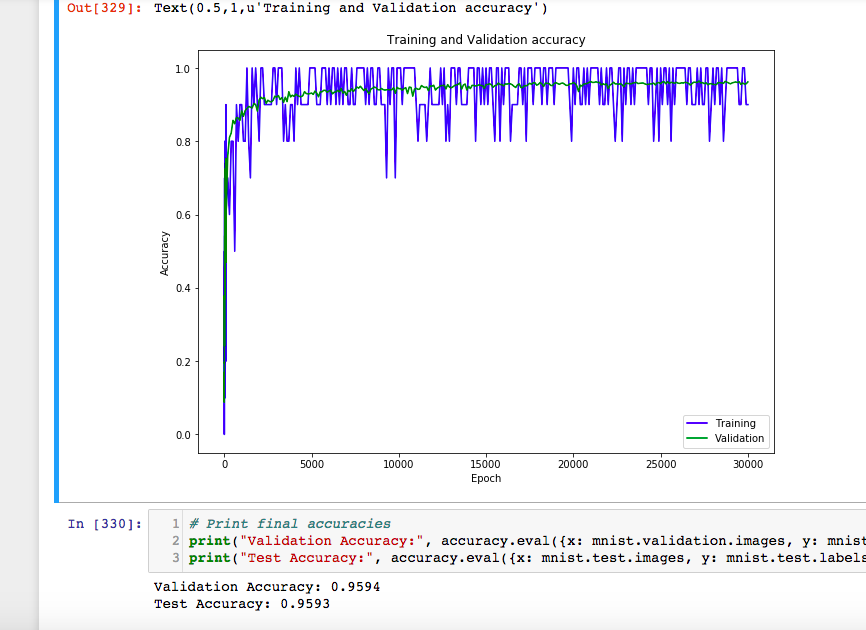
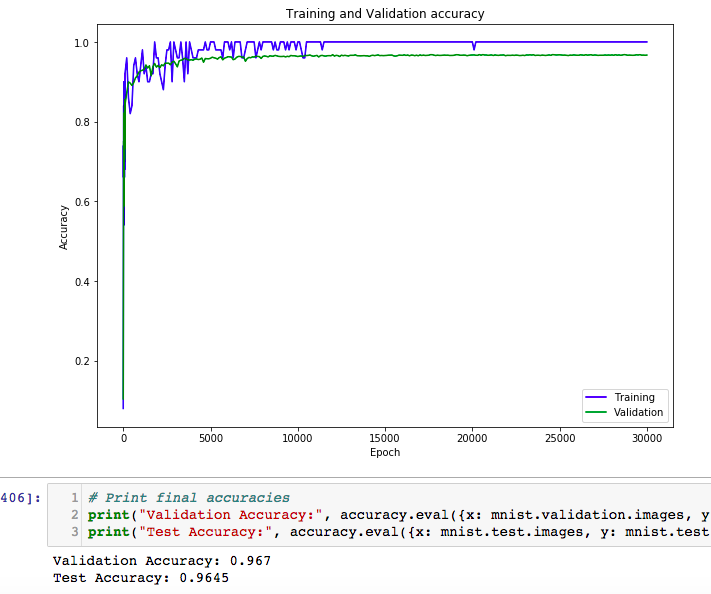
In the first running, *n* = 30, *η* = 1.0, *m* = 30 and we got the accuracy of this model is 95.2%, it seems good but when this model runs in a complex system where millions of digits are processed, this 4.8% difference would be significant. From what we have learned before, we assume that the ability of recognition of a neural network is related to the ability of its hidden layer to detect details of inputs. So, we tune the *n* to 100 to verify the assumption. As the result shows, the accuracy increases to 95.9%. The performance of our model does improve, but not too much. To avoid over-fitting and exceeded running time, we finally set the neurons of hidden layer to 300.

In the next step, we need to tune the learning rate. If it’s too small, it will take a long time to run and potentially never reaching the optimum. But if it’s too big, then the optimization may be unstable and bouncing around it.



After tuning, we found that the performance fixed between a certain threshold when learning rate is [1.5, 1.8], and we finally choose to use 1.6 as learning rate.

SGD uses a subset of training data set called mini batch to evaluate the cost function. This subset changes each iteration. When the size of a mini batch is too small, it will cause a slow convergence (the parameters may jump around a lot rather than smoothly approaching the optimum outcome). And the advantage of speed may get lost as more of the training data gets used.

The final accuracy of this model with parameters of *m*=50, *m*=300, *η*=1.6 is 96.7%. A 96.7% accuracy is not bad because it uses only one hidden layer and we could also replicate it in a proof-of-concept application. If we run the scikit-learn’s SVM classifier using default setting for recognition of handwritten digits, we will get the accuracy of 94%, which means our one hidden neural network is a little better than the default SVM.