STATS 202A – FINAL PROJECT TASK 3

NAME: ANOOSHA SAGAR
STUDENT ID: 605028604

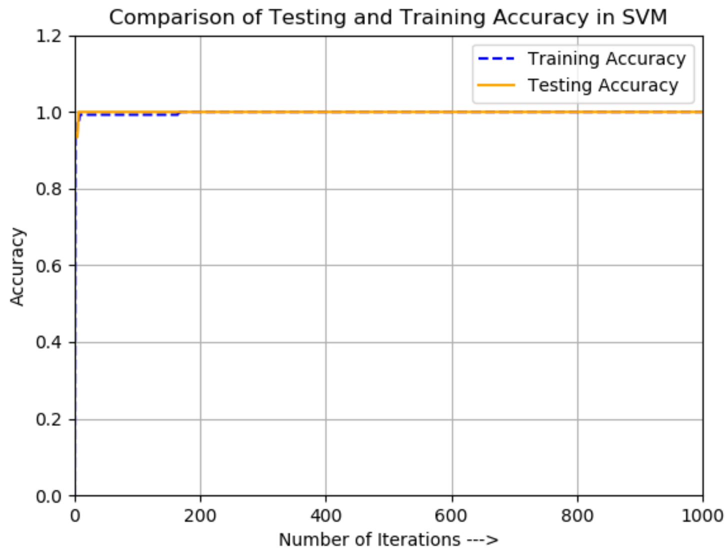
SUPPORT VECTOR MACHINES

CONSOLE OUTPUT

```
C:\Users\anoos\Anaconda2\python.exe
E:/UCLA/CourseWork/Fall2017/StatisticsProgramming/Week10/Stats202A_HW9_P2.py

Training Accuracy at Iteration  0 :  0.0

Testing Accuracy at Iteration  0 :  0.934065934066

Training Accuracy at Iteration  100 :  0.992647058824

Testing Accuracy at Iteration  100 :  1.0

Training Accuracy at Iteration  200 :  1.0

Testing Accuracy at Iteration  200 :  1.0

Training Accuracy at Iteration  300 :  1.0

Testing Accuracy at Iteration  300 :  1.0

Training Accuracy at Iteration  400 :  1.0

Testing Accuracy at Iteration  400 :  1.0

Training Accuracy at Iteration  500 :  1.0

Testing Accuracy at Iteration  500 :  1.0

Training Accuracy at Iteration  600 :  1.0

Testing Accuracy at Iteration  600 :  1.0

Training Accuracy at Iteration  700 :  1.0

Testing Accuracy at Iteration  700 :  1.0

Training Accuracy at Iteration  800 :  1.0

Testing Accuracy at Iteration  800 :  1.0

Training Accuracy at Iteration  900 :  1.0

Testing Accuracy at Iteration  900 :  1.0
```

GRAPHICAL REPRESENTATION

```
x, y, z = my_SVM(X_train, Y_train, X_test, Y_test)
plt.xlabel("Number of Iterations --->")
plt.ylabel("Accuracy")
plt.title("Comparison of Testing and Training Accuracy in SVM")
plt.plot(y, 'b--', label="Training Accuracy")
plt.plot(z, label="Testing Accuracy", color='orange')

plt.axis([0,1000,0,1.2])
plt.legend()
plt.grid(True)
plt.show()
```



In machine learning, **support vector machines** are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

From the above graph, SVM does not overfit and provides excellent results for training and testing accuracy. Also, SVM is resilient to noise.

SVM REACHES PEAK ACCURACY THE QUICKEST

ADAPTIVE BOOSTING (ADABOOST)

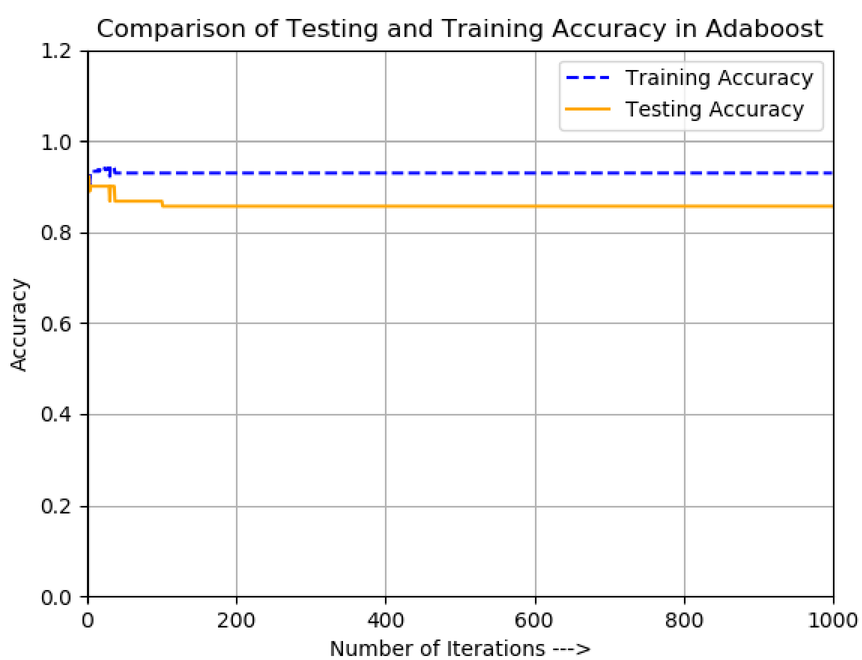
CONSOLE OUTPUT

```
C:\Users\anoos\Anaconda2\python.exe
E:/UCLA/CourseWork/Fall2017/StatisticsProgramming/Week10/Stats202A_HW9_P2.py
Training Accuracy at Iteration  0 :  0.904411764706
Testing Accuracy at Iteration  0 :  0.89010989011
Training Accuracy at Iteration  100 :  0.930147058824
Testing Accuracy at Iteration  100 :  0.868131868132
Training Accuracy at Iteration  200 :  0.930147058824
Testing Accuracy at Iteration  200 :  0.857142857143
Training Accuracy at Iteration  300 :  0.930147058824
Testing Accuracy at Iteration  300 :  0.857142857143
Training Accuracy at Iteration  400 :  0.930147058824
Testing Accuracy at Iteration  400 :  0.857142857143
Training Accuracy at Iteration  500 :  0.930147058824
```

```
Testing Accuracy at Iteration  500 :  0.857142857143

Training Accuracy at Iteration  600 :  0.930147058824

Testing Accuracy at Iteration  600 :  0.857142857143

Training Accuracy at Iteration  700 :  0.930147058824

Testing Accuracy at Iteration  700 :  0.857142857143

Training Accuracy at Iteration  800 :  0.930147058824

Testing Accuracy at Iteration  800 :  0.857142857143

Training Accuracy at Iteration  900 :  0.930147058824

Testing Accuracy at Iteration  900 :  0.857142857143
```

GRAPHICAL REPRESENTATION

```
x, y, z = my_Adaboost(X_train, Y_train, X_test, Y_test)
plt.xlabel("Number of Iterations --->")
plt.ylabel("Accuracy")
plt.title("Comparison of Testing and Training Accuracy in Adaboost")
plt.plot(y, 'b--', label="Training Accuracy")
plt.plot(z, label="Testing Accuracy", color='orange')
plt.axis([0,1000,0,1.2])
plt.legend()
plt.grid(True)
plt.show()
```



AdaBoost is a type of "Ensemble Learning" where multiple learners are employed to build a stronger learning algorithm. AdaBoost works by choosing a base algorithm (e.g. decision trees) and iteratively improving it by accounting for the incorrectly classified examples in the training set.

The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the boosted classifier. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. In some problems it can be less susceptible to the overfitting problem than other learning algorithms. The individual learners can be weak, but as long as the performance of each one is slightly better than random guessing, the final model can be proven to converge to a strong learner

AdaBoost can handle sparse datasets and therefore, can work with weak classifiers but it shows some variations after it reaches peak classification correctness. Observing the above graph, AdaBoost reaches an effective solution early but becomes sensitive to noise in further iterations.

OUT OF ALL THE TESTED TECHNIQUES, ADABOOST GIVES THE LOWEST ACCURACY.

NEURAL NETWORKS

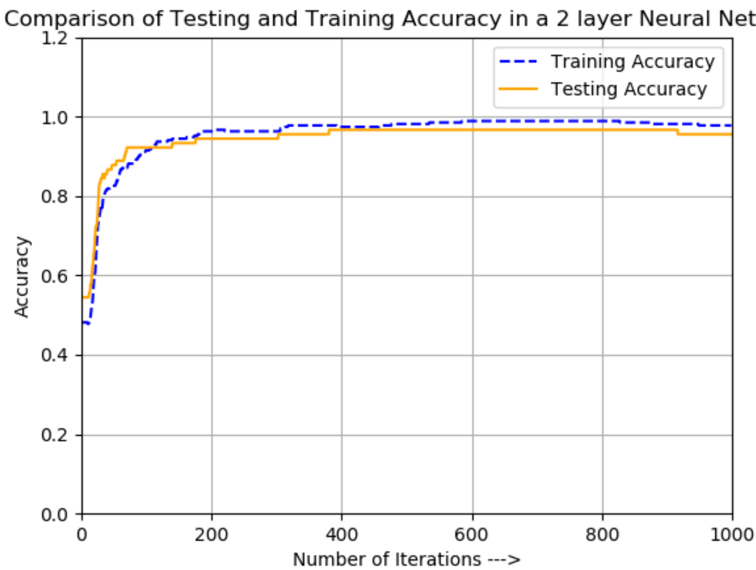
CONSOLE OUTPUT

```
C:\Users\anoos\Anaconda2\python.exe E:/UCLA/CourseWork/Fall2017/StatisticsProgramming/Final/2_layer_nn.py
Training Accuracy at Iteration  0 :  0.481481481481
Testing Accuracy at Iteration  0 :  0.544444444444
Training Accuracy at Iteration  100 :  0.914814814815
Testing Accuracy at Iteration  100 :  0.922222222222
Training Accuracy at Iteration  200 :  0.962962962963
Testing Accuracy at Iteration  200 :  0.944444444444
Training Accuracy at Iteration  300 :  0.962962962963
Testing Accuracy at Iteration  300 :  0.944444444444
Training Accuracy at Iteration  400 :  0.974074074074
Testing Accuracy at Iteration  400 :  0.966666666667
Training Accuracy at Iteration  500 :  0.981481481481
Testing Accuracy at Iteration  500 :  0.966666666667
Training Accuracy at Iteration  600 :  0.988888888889
Testing Accuracy at Iteration  600 :  0.966666666667
Training Accuracy at Iteration  700 :  0.988888888889
Testing Accuracy at Iteration  700 :  0.966666666667
Training Accuracy at Iteration  800 :  0.988888888889
Testing Accuracy at Iteration  800 :  0.966666666667
Training Accuracy at Iteration  900 :  0.981481481481
Testing Accuracy at Iteration  900 :  0.966666666667
```

GRAPHICAL REPRESENTATION

```
X_train, Y_train, X_test, Y_test = prepare_data()
alpha,beta,acc_train,acc_test=my_NN(X_train,Y_train,X_test,Y_test,num_hidden=50,num_iterations=1000,learning_rate=1e-2)
plt.axis([0,1000,0,1.2])
plt.xlabel("Number of Iterations --->")
plt.ylabel("Accuracy")
plt.title("Comparison of Testing and Training Accuracy in a 2 layer Neural Network")
```

```
plt.plot(acc_train, 'b--', label="Training Accuracy")
plt.plot(acc_test, label="Testing Accuracy", color='orange')

plt.legend()
plt.grid(True)
plt.show()
```



Neural networks process information in a similar way the human brain does. The network is composed of many highly interconnected processing elements(neurons) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

Neural network is a non-linear classifier, and the hidden layers introduce complexity. Neural networks are also heavily parametric.

ALTHOUGH NEURAL NETWORKS TAKE A LONG TIME TO CONVERGE, THEY PROVIDE GOOD ACCURACY.

POINT TO NOTE: IN THE CASE OF RELU IN NEURAL NETWORKS, IT WAS OBSERVED THAT THE PERFORMANCE OF THE NETWORK ALSO DEPENDS ON THE INITIALIZATION OF THE WEIGHTS. FOR DIFFERENT KINDS OF INITIALIZATION, DIFFERENT RESULTS WERE OBSERVED. THIS BEHAVIOR WAS OBSERVED IN TENSORFLOW AND PYTHON ALSO.

NEURAL NETWORKS USING TENSORFLOW

OVERVIEW

TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more

CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

GENERAL PRINCIPAL FOR BUILDING A NEURAL NETWORK

To build a neural network with Tensorflow, you can follow the steps below:

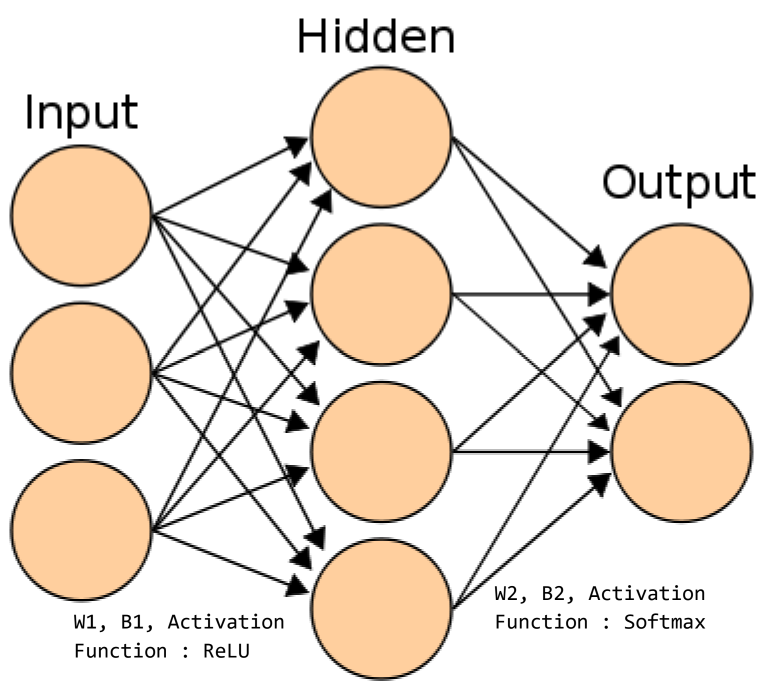
1. Import the modules you need. 2)
2. Define an add_layer function to construct layers.
3. Define the variables and initialize them.
4. Create a session to perform the operation.
5. Build the NN, and send data with placeholders and feed_dict
6. Train and test the NN, and observe the results with Tensorboard.

BUILDING A TWO LAYER NEURAL NETWORK WITH RELU AND SOFTMAX AS ACTIVATION FUNCTIONS

PROBLEM STATEMENT

```
Train a two layer neural network to classify the MNIST dataset ##
## Use Relu as the activation function for the first layer. Use Softmax as the
activation function for the second layer##
## z=Relu(x*W1+b1) ##
## y=Softmax(z*W2+b2)##
# Use cross-entropy as the loss function#
```

DIAGRAM OF THE NEURAL NETWORK



OUTPUT AT EACH LAYER

LAYER 1: RECTIFIED LINEAR UNITS (RELU)

```
W1 = tf.get_variable('w1', [784, 500],
initializer=tf.random_normal_initializer(stddev=0.3))
b1 = tf.get_variable('b1', [1,], initializer=tf.random_normal_initializer(stddev=0.3))
y1 = tf.nn.relu(tf.matmul(x, W1) + b1)
```

LAYER 2: SOFTMAX

```
W2 = tf.get_variable('w2', [500, 10],
initializer=tf.random_normal_initializer(stddev=0.3))
b2 = tf.get_variable('b2', [1, ],
initializer=tf.random_normal_initializer(stddev=0.3))
y2 = tf.nn.softmax(tf.matmul(y1, W2) + b2)
```

CHOOSING OPTIMIZATION FUNCTIONS

GRADIENT DESCENT OPTIMIZER

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point.

ADAM OPTIMIZER

The tf.train.AdamOptimizer uses Kingma and Ba's Adam algorithm to control the learning rate. Adam offers several advantages over the simple tf.train.GradientDescentOptimizer. Foremost is that it uses **moving averages of the parameters** (momentum); this enables Adam to use a larger effective step size, and the algorithm will converge to this step size without fine tuning.

The main down side of the algorithm is that Adam requires more computation to be performed for each parameter in each training step (to maintain the moving averages and variance, and calculate the scaled gradient); and more state to be retained for each parameter (approximately tripling the size of the model to store the average and variance for each parameter).

FOR MY NEURAL NETWORK I CHOSE THE GRADIENT DESCENT OPTIMIZER

TRAINING THE NETWORK USING CROSS ENTROPY

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y2,
labels=y_))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

for epoch in range(100):
  for i in range(int(mnist.train.num_examples / 100)):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
  print("Epoch Number",epoch)
```

```
  correct = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))

  accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
  print("Accuracy: ",sess.run(accuracy, feed_dict={x:mnist.test.images,
y_:mnist.test.labels})
```

RESULTS OBTAINED

After 100 epochs of training, my network was able to achieve an accuracy of 0.9671 or 96.71%

CONDENSED CONSOLE OUTPUT

```
C:\Users\anoos\AppData\Local\Programs\Python\Python36\python.exe
E:/UCLA/CourseWork/Fall2017/StatisticsProgramming/Final/tensorflow1.py

Extracting /tmp/tensorflow/mnist/input_data\train-images-idx3-ubyte.gz

Extracting /tmp/tensorflow/mnist/input_data\train-labels-idx1-ubyte.gz

Extracting /tmp/tensorflow/mnist/input_data\t10k-images-idx3-ubyte.gz

Extracting /tmp/tensorflow/mnist/input_data\t10k-labels-idx1-ubyte.gz

2017-12-14 10:06:43.623539: I C:\tf_jenkins\home\workspace\rel-
win\M\windows\PY\36\tensorflow\core\platform\cpu_feature_guard.cc:137] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: AVX AVX2

Epoch Number 0

Accuracy:  0.7319

Epoch Number 1

Accuracy:  0.7478

Epoch Number 2

Accuracy:  0.8409

Epoch Number 3

Accuracy:  0.8494

Epoch Number 4

Accuracy:  0.8547

Epoch Number 5

Accuracy:  0.8577

Epoch Number 6

Accuracy:  0.8615

Epoch Number 7

Accuracy:  0.863

Epoch Number 8

Accuracy:  0.8655

Epoch Number 9

Accuracy:  0.866

Epoch Number 10

Accuracy:  0.866

Epoch Number 11

Accuracy:  0.8673

Epoch Number 12

Accuracy:  0.8672

Epoch Number 13

Accuracy:  0.8673

Epoch Number 14

Accuracy:  0.8684

Epoch Number 15

Accuracy:  0.8683

Epoch Number 16

Accuracy:  0.8677

Epoch Number 17

Accuracy:  0.8691

Epoch Number 18

Accuracy:  0.8685

Epoch Number 19

Accuracy:  0.8692

Epoch Number 20

Accuracy:  0.8697

Epoch Number 21

Accuracy:  0.8696

Epoch Number 22

Accuracy:  0.8702

Epoch Number 23

Accuracy:  0.8699

Epoch Number 24

Accuracy:  0.8696

Epoch Number 25
```

```
Accuracy:  0.8703
Epoch Number 26
Accuracy:  0.8706
Epoch Number 27
Accuracy:  0.8708
Epoch Number 28
Accuracy:  0.947
Epoch Number 29
Accuracy:  0.9556
Epoch Number 30
Accuracy:  0.9588
Epoch Number 31
Accuracy:  0.9607
Epoch Number 32
Accuracy:  0.9611
Epoch Number 33
Accuracy:  0.9619
Epoch Number 34
Accuracy:  0.9615
Epoch Number 35
Accuracy:  0.963
Epoch Number 36
Accuracy:  0.9634
Epoch Number 37
Accuracy:  0.9637
Epoch Number 38
Accuracy:  0.9636
Epoch Number 39
Accuracy:  0.9637
Epoch Number 40
Accuracy:  0.9644
Epoch Number 41
Accuracy:  0.9642
Epoch Number 42
Accuracy:  0.9638
Epoch Number 43
Accuracy:  0.9647
Epoch Number 44
Accuracy:  0.964
Epoch Number 45
Accuracy:  0.9642
Epoch Number 46
Accuracy:  0.9644
Epoch Number 47
Accuracy:  0.9648
Epoch Number 48
Accuracy:  0.9645
Epoch Number 49
Accuracy:  0.9647
Epoch Number 50
Accuracy:  0.9643
Epoch Number 51
Accuracy:  0.9645
Epoch Number 52
Accuracy:  0.9642
Epoch Number 53
Accuracy:  0.9652
Epoch Number 54
Accuracy:  0.9654
Epoch Number 55
Accuracy:  0.9656
Epoch Number 56
Accuracy:  0.9662
Epoch Number 57
Accuracy:  0.9657
Epoch Number 58
Accuracy:  0.9661
Epoch Number 59
Accuracy:  0.967
Epoch Number 60
Accuracy:  0.9656
Epoch Number 61
Accuracy:  0.9659
Epoch Number 62
Accuracy:  0.9666
Epoch Number 63
Accuracy:  0.9665
Epoch Number 64
Accuracy:  0.9666
Epoch Number 65
Accuracy:  0.966
Epoch Number 66
Accuracy:  0.9671
Epoch Number 67
Accuracy:  0.9672
Epoch Number 68
Accuracy:  0.967
Epoch Number 69
Accuracy:  0.967
Epoch Number 70
Accuracy:  0.967
Epoch Number 71
Accuracy:  0.9666
```

```
Epoch Number 72
Accuracy:  0.9671
Epoch Number 73
Accuracy:  0.9667
Epoch Number 74
Accuracy:  0.9666
Epoch Number 75
Accuracy:  0.9668
Epoch Number 76
Accuracy:  0.9667
Epoch Number 77
Accuracy:  0.9668
Epoch Number 78
Accuracy:  0.9664
Epoch Number 79
Accuracy:  0.9666
Epoch Number 80
Accuracy:  0.9669
Epoch Number 81
Accuracy:  0.9673
Epoch Number 82
Accuracy:  0.9667
Epoch Number 83
Accuracy:  0.9671
Epoch Number 84
Accuracy:  0.9671
Epoch Number 85
Accuracy:  0.9671
Epoch Number 86
Accuracy:  0.9671
Epoch Number 87
Accuracy:  0.9675
Epoch Number 88
Accuracy:  0.967
Epoch Number 89
Accuracy:  0.9669
Epoch Number 90
Accuracy:  0.9668
Epoch Number 91
Accuracy:  0.9672
Epoch Number 92
Accuracy:  0.9675
Epoch Number 93
Accuracy:  0.9672
Epoch Number 94
Accuracy:  0.9673
Epoch Number 95
Accuracy:  0.9675
Epoch Number 96
Accuracy:  0.9673
Epoch Number 97
Accuracy:  0.9674
Epoch Number 98
Accuracy:  0.9674
Epoch Number 99
Accuracy:  0.9671
0.9671
```