Deep Learning

Lab Session 2 - 3 Hours

Convolutional Neural Network (CNN) for Handwritten Digits Recognition

Student 1: Daniele Reda
Student 2: Matteo Romiti

The aim of this session is to practice with Convolutional Neural Networks. Answers and experiments should be made by groups of one or two students. Each group should fill and run appropriate notebook cells.

Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an pdf document using print as PDF (Ctrl+P). Do not forget to run all your cells before generating your final report and do not forget to include the names of all participants in the group. The lab session should be completed by May 29th 2017.

Send you pdf file to benoit.huet@eurecom.fr and olfa.ben-ahmed@eurecom.fr using **[DeepLearning_lab2]** as Subject of your email.

Introduction

In the last Lab Session, you built a Multilayer Perceptron for recognizing hand-written digits from the MNIST data-set. The best achieved accuracy on testing data was about 97%. Can you do better than these results using a deep CNN ? In this Lab Session, you will build, train and optimize in TensorFlow one of the early Convolutional Neural Networks: **LeNet-5** to go to more than 99% of accuracy.

Load MNIST Data in TensorFlow

Run the cell above to load the MNIST data that comes with TensorFlow. You will use this data in **Section 1** and **Section 2**.

In [1]:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
X_train, y_train           = mnist.train.images, mnist.train.labels
X_validation, y_validation = mnist.validation.images, mnist.validation.labels
X_test, y_test             = mnist.test.images, mnist.test.labels
print("Image Shape:    {}".format(X_train[0].shape))
print("Training Set:   {} samples".format(len(X_train)))
print("Validation Set: {} samples".format(len(X_validation)))
print("Test Set:       {} samples".format(len(X_test)))
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Image Shape:    (784,)
Training Set:   55000 samples
Validation Set: 5000 samples
Test Set:       10000 samples
```

Section 1 : My First Model in TensorFlow

Before starting with CNN, let's train and test in TensorFlow the example : **y=softmax(Wx+b)** seen in the DeepLearing course last week.

This model reaches an accuracy of about 92 %. You will also learn how to launch the tensorBoard https://www.tensorflow.org/get_started/summaries_and_tensorboard (https://www.tensorflow.org/get_started/summaries_and_tensorboard) to visualize the computation graph, statistics and learning curves.

Part 1 : Read carefully the code in the cell below. Run it to perform training.

In [2]:

```
#STEP 1

# Parameters
learning_rate = 0.01
training_epochs = 100
batch_size = 128
display_step = 2
logs_path = 'log_files/'  # useful for tensorboard

# tf Graph Input:  mnist data image of shape 28*28=784
x = tf.placeholder(tf.float32, [None, 784], name='InputData')
# 0-9 digits recognition,  10 classes
y = tf.placeholder(tf.float32, [None, 10], name='LabelData')

# Set model weights
W = tf.Variable(tf.zeros([784, 10]), name='Weights')
b = tf.Variable(tf.zeros([10]), name='Bias')

# Construct model and encapsulating all ops into scopes, making Tensorboard's Gr
aph visualization more convenient
```

```
with tf.name_scope('Model'):
    # Model
    pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax
with tf.name_scope('Loss'):
    # Minimize error using cross entropy
    cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
with tf.name_scope('SGD'):
    # Gradient Descent
    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
with tf.name_scope('Accuracy'):
    # Accuracy
    acc = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    acc = tf.reduce_mean(tf.cast(acc, tf.float32))

# Initializing the variables
init = tf.global_variables_initializer()
# Create a summary to monitor cost tensor
tf.summary.scalar("Loss", cost)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("Accuracy", acc)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()


#STEP 2


# Launch the graph for training
with tf.Session() as sess:
    sess.run(init)
    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logs_path,
graph=tf.get_default_graph())
    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Run optimization op (backprop), cost op (to get loss value)
            # and summary nodes
            _, c, summary = sess.run([optimizer, cost, merged_summary_op],
                                     feed_dict={x: batch_xs, y: batch_ys})
            # Write logs at every iteration
            summary_writer.add_summary(summary, epoch * total_batch + i)
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            print("Epoch: ", '%02d' % (epoch+1), "  =====> Loss=", "{:.9f}".form
at(avg_cost))

    print("Optimization Finished!")

    # Test model
    # Calculate accuracy
    print("Accuracy:", acc.eval({x: mnist.test.images, y: mnist.test.labels}))
```

```
Epoch:  02   =====> Loss= 0.733064874
Epoch:  04   =====> Loss= 0.536358523
Epoch:  06   =====> Loss= 0.471889261
Epoch:  08   =====> Loss= 0.435504827
Epoch:  10   =====> Loss= 0.412166757
Epoch:  12   =====> Loss= 0.397256630
Epoch:  14   =====> Loss= 0.384323012
Epoch:  16   =====> Loss= 0.374909195
Epoch:  18   =====> Loss= 0.365859221
Epoch:  20   =====> Loss= 0.358819640
Epoch:  22   =====> Loss= 0.353986612
Epoch:  24   =====> Loss= 0.349985992
Epoch:  26   =====> Loss= 0.345158857
Epoch:  28   =====> Loss= 0.340375543
Epoch:  30   =====> Loss= 0.337398454
Epoch:  32   =====> Loss= 0.333335278
Epoch:  34   =====> Loss= 0.330760923
Epoch:  36   =====> Loss= 0.326252774
Epoch:  38   =====> Loss= 0.325419832
Epoch:  40   =====> Loss= 0.322225989
Epoch:  42   =====> Loss= 0.320298648
Epoch:  44   =====> Loss= 0.319281198
Epoch:  46   =====> Loss= 0.318291789
Epoch:  48   =====> Loss= 0.314709458
Epoch:  50   =====> Loss= 0.313845383
Epoch:  52   =====> Loss= 0.309665868
Epoch:  54   =====> Loss= 0.311834381
Epoch:  56   =====> Loss= 0.306622827
Epoch:  58   =====> Loss= 0.308776800
Epoch:  60   =====> Loss= 0.305952971
Epoch:  62   =====> Loss= 0.303234257
Epoch:  64   =====> Loss= 0.304481421
Epoch:  66   =====> Loss= 0.302969264
Epoch:  68   =====> Loss= 0.304459532
Epoch:  70   =====> Loss= 0.303972692
Epoch:  72   =====> Loss= 0.298687273
Epoch:  74   =====> Loss= 0.300435235
Epoch:  76   =====> Loss= 0.297799206
Epoch:  78   =====> Loss= 0.299489916
Epoch:  80   =====> Loss= 0.295029076
Epoch:  82   =====> Loss= 0.297068079
Epoch:  84   =====> Loss= 0.296208043
Epoch:  86   =====> Loss= 0.293802273
Epoch:  88   =====> Loss= 0.290398892
Epoch:  90   =====> Loss= 0.294975000
Epoch:  92   =====> Loss= 0.291894806
Epoch:  94   =====> Loss= 0.292634473
Epoch:  96   =====> Loss= 0.292533482
Epoch:  98   =====> Loss= 0.290901574
Epoch:  100   =====> Loss= 0.289058641
Optimization Finished!
Accuracy: 0.9203
```

Part 2 : Using Tensorboard, we can now visualize the created graph, giving you an overview of your architecture and how all of the major components are connected. You can also see and analyse the learning curves.

To launch tensorBoard:

- Go to the **TP2** folder,
- Open a Terminal and run the command line **"tensorboard --logdir= log_files/"**, it will generate an http link ,ex http://666.6.6.6:6006 (http://666.6.6.6:6006),
- Copy this link into your web browser

Enjoy It !!

Section 2 : The 99% MNIST Challenge !

Part 1 : LeNet5 implementation

Once you are familar with **tensorFlow** and **tensorBoard**, you are in this section to build, train and test the baseline LeNet-5 (http://yann.lecun.com/exdb/lenet/) model for the MNIST digits recognition problem.

In more advanced step you will make some optimizations to get more than 99% of accuracy. The best model can get to over 99.7% accuracy!

For more information, have a look at this list of results :
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html)

<img src="lenet.png",width="800" height="600" align="center">
Figure 1: Lenet 5

The LeNet architecture accepts a 32x32xC image as input, where C is the number of color channels. Since MNIST images are grayscale, C is 1 in this case.

Layer 1: Convolutional. The output shape should be 28x28x6 **Activation.** sigmoid **Pooling.** The output shape should be 14x14x6.

Layer 2: Convolutional. The output shape should be 10x10x16. **Activation.** sigmoid **Pooling.** The output shape should be 5x5x16.

Flatten. Flatten the output shape of the final pooling layer such that it's 1D instead of 3D. You may need to use **flatten* from tensorflow.contrib.layers import flatten

Layer 3: Fully Connected. This should have 120 outputs. **Activation.** sigmoid

Layer 4: Fully Connected. This should have 84 outputs. **Activation.** sigmoid

Layer 5: Fully Connected. This should have 10 outputs **Activation.** softmax.

Question 2.1.1 Implement the Neural Network architecture described above. For that, your will use classes and functions from https://www.tensorflow.org/api_docs/python/tf/nn (https://www.tensorflow.org/api_docs/python/tf/nn).

We give you some helper functions for weigths and bias initilization. Also you can refer to section 1.

In [2]:

```
# Helper functions  for weigths and bias initilization
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d(x, W, stride, padding_):
    return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding=padding_)
```

In [3]:

```
def LeNet5_Model(data, transfer="sigmoid", keep_prob=1.):
    # your implementation goes here

    transferFuncs = {"sigmoid" : tf.sigmoid, "ReLU": tf.nn.relu}

    #first convolutional layer
    W_conv1 = weight_variable([5, 5, 1, 6]) ## [filter_width, filter_height, dep
th_image_in, depth_image_out]
    b_conv1 = bias_variable([6])
    h_conv1 = transferFuncs[transfer](conv2d(data, W_conv1, 1, 'SAME') +
b_conv1)
    pool1 = tf.nn.pool(h_conv1, [2,2], "MAX", 'VALID', strides=[2,2])

    #second convolutional layer
    W_conv2 = weight_variable([5, 5, 6, 16])
    b_conv2 = bias_variable([16])
    h_conv2 = transferFuncs[transfer](conv2d(pool1, W_conv2, 1, 'VALID') + b_con
v2)
    pool2 = tf.nn.pool(h_conv2, [2,2], "MAX", 'VALID', strides=[2,2])

    #first fully connected layer
    s = pool2.get_shape().as_list()
    flattened_length = s[1] * s[2] * s[3]
    pool2_flat = tf.reshape(pool2, [-1, flattened_length])
    W_fc1 = weight_variable([1*5*5*16, 120])
    b_fc1 = bias_variable([120])
    h_fc1 = transferFuncs[transfer](tf.matmul(pool2_flat, W_fc1) + b_fc1)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

    #second fully connected layer
    W_fc2 = weight_variable([120, 84])
    b_fc2 = bias_variable([84])
    h_fc2 = transferFuncs[transfer](tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
    h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)

    #third fully connected layer
    W_fc3 = weight_variable([84, 10])
    b_fc3 = bias_variable([10])
    h_fc3 = tf.nn.softmax(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)

    return h_fc3
```

Question 2.1.2. Calculate the number of parameters of this model

In [5]:

```
# first conv
pconv1 = 5*5*1*6 # filter_height * filter_width * channels_in * num_feature_maps
# second conv
pconv2 = 5*5*1*16 # filter_height * filter_width * channels_in * num_feature_map
s
# first fcl
pfcl1 = 5*5*16*120 # fcl_input_size * fcl_output_size
pfcl1# second fcl
pfcl2 = 84*120 # fcl_input_size * fcl_output_size
# third fcl
pfcl3 = 84*10 # fcl_input_size * fcl_output_size
pbias = 6+16+120+84+10 # all the biases
total = pbias + pfcl1 + pfcl2 + pfcl3 + pconv2 + pconv1
print(total)
```

```
59706
```

Question 2.1.3. Start the training with the parameters cited below:

```
Learning rate : 0.1
Loss Function : Cross entropy
Optimizer: SGD
Number of training iterations : 10000
Batch size : 128
```

In [8]:

```
learning_rate = 0.1
training_epochs = 200 # as suggested in the email
batch_size = 128
display_step = 10
logs_path = 'log_files/'
```

Question 2.1.4. Implement the evaluation function for accuracy computation

In [4]:

```
def evaluate(model, y):
    correct = tf.equal(tf.argmax(model, 1), tf.argmax(y, 1))
    return tf.reduce_mean(tf.cast(correct, tf.float32))
```

Question 2.1.5. Implement training pipeline and run the training data through it to train the model.

- Before each epoch, shuffle the training set.
- Print the loss per mini batch and the training/validation accuracy per epoch. (Display results every 100 epochs)
- Save the model after training
- Print after training the final testing accuracy

In [5]:

```
def train(learning_rate, training_epochs, batch_size, display_step = 1, \
          logs_path='log_files/', optFunction="SGD", verbose=True, transfer="sig
```

```
moid", keep_probability= 1.0):

    optFunctions = {"SGD":tf.train.GradientDescentOptimizer, "Adam":tf.train.Ada
mOptimizer}

    # Erase previous graph
    tf.reset_default_graph()

    x = tf.placeholder(tf.float32, [None, 28, 28, 1], name='InputData')
    y = tf.placeholder(tf.float32, [None, 10], name='LabelData')
    keep_prob = tf.placeholder(tf.float32)

    # Construct model
    with tf.name_scope('Model'):
        pred = LeNet5_Model(x, transfer=transfer)

    # Define loss and optimizer
    with tf.name_scope('Loss'):
        cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred),
reduction_indices=1))
        #cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pr
ed, labels=y))

    with tf.name_scope(optFunction):
        if transfer is "sigmoid":
            optimizer = optFunctions[optFunction](learning_rate).minimize(cost)
        else:
            opt = optFunctions[optFunction](learning_rate)
            gvs = opt.compute_gradients(cost)
            capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var i
n gvs]
            optimizer = opt.apply_gradients(capped_gvs)

    # Evaluate model
    with tf.name_scope('Accuracy'):
        accuracy = evaluate(pred, y)

    # Initializing the variables
    init = tf.global_variables_initializer()

    # Create a summary to monitor cost tensor
    tf.summary.scalar("Loss", cost)
    # Create a summary to monitor accuracy tensor
    tf.summary.scalar("Accuracy", accuracy)
    # Merge all summaries into a single op
    merged_summary_op = tf.summary.merge_all()

    x_val, y_val = mnist.validation.images.reshape(-1, 28, 28, 1), mnist.validat
ion.labels
    x_test, y_test = mnist.test.images.reshape(-1, 28, 28, 1), mnist.test.labels

    with tf.Session() as sess:
#         acc_history = []
        test_history = []
        val_history = []
#         train_history = []

        sess.run(init)
        if verbose is True:
            print("Start Training!")
```

```
        # op to write logs to Tensorboard
        summary_writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_g
raph())
        saver = tf.train.Saver()
        #Training cycle
        for epoch in range(training_epochs):
            avg_cost = 0.
            total_batch = int(mnist.train.num_examples/batch_size)
            #Loop over all batches
            for i in range(total_batch):
                batch_xs, batch_ys = mnist.train.next_batch(batch_size)
                batch_xs = batch_xs.reshape(-1, 28, 28, 1)
                # Run optimization op (backprop), cost op (to get loss value)
                # and summary nodes
                _, c, summary = sess.run([optimizer, cost, merged_summary_op],
                                         feed_dict={x: batch_xs, y: batch_ys, ke
ep_prob: keep_probability})
#                 _, c, summary = sess.run([optimizer, cost, merged_summary_op],
#                                          feed_dict={x: batch_xs, y: batch_ys,
 keep_prob:keep_prob_})
                # Write logs at every iteration
                summary_writer.add_summary(summary, epoch * total_batch + i)
                # Compute average loss
                avg_cost += c / total_batch
            # Display logs per epoch step
#             train_acc = accuracy.eval({x: batch_xs, y:batch_ys})
            val_acc = accuracy.eval({x: x_val, y:y_val, keep_prob:1.0})
            test_acc = accuracy.eval({x: x_test, y:y_test, keep_prob:1.0})
#             acc_history.append(acc)
#             train_history.append(train_acc)
            val_history.append(val_acc)
            test_history.append(test_acc)

            saver.save(sess, 'Models/model_' + str(learning_rate) + '_' + str(ba
tch_size) + '_' + optFunction)
            if verbose is True and (epoch+1) % display_step == 0:
                print("Epoch: ", '%02d' % (epoch+1), \
                      "  =====> Loss=", "{:.9f}".format(avg_cost), \
                      " Validation accuracy=", val_acc, " Test accuracy=", test_
acc)
            if val_acc>=0.99:
                if verbose is True:
                    print("Validation Accuracy over 99%% reached after %d epoch
s" %(epoch+1))
                break

        if verbose is True:
            print("Training Finished!")
            # Test model
            # Calculate accuracy
            print("Test accuracy:", accuracy.eval({x: x_test, y:y_test, keep_pro
b:1.0}))

    return val_history, test_history
```

`train.next_batch()` has shuffle parameter set to True by default.

In [9]:

```
val_hist, test_hist = train(learning_rate, training_epochs, batch_size)
```

```
Start Training!
Epoch:  01   =====> Loss= 2.306711101  Validation accuracy= 0.1126
 Test accuracy= 0.1135
Epoch:  02   =====> Loss= 2.305636428  Validation accuracy= 0.1126
 Test accuracy= 0.1135
Epoch:  03   =====> Loss= 2.304683939  Validation accuracy= 0.099  T
est accuracy= 0.1009
Epoch:  04   =====> Loss= 2.304435188  Validation accuracy= 0.0986
 Test accuracy= 0.101
Epoch:  05   =====> Loss= 2.302418652  Validation accuracy= 0.1126
 Test accuracy= 0.1135
Epoch:  06   =====> Loss= 2.300473815  Validation accuracy= 0.0868
 Test accuracy= 0.0892
Epoch:  07   =====> Loss= 2.294925442  Validation accuracy= 0.1582
 Test accuracy= 0.1736
Epoch:  08   =====> Loss= 2.272766478  Validation accuracy= 0.2568
 Test accuracy= 0.2669
Epoch:  09   =====> Loss= 1.972761237  Validation accuracy= 0.5516
 Test accuracy= 0.5554
Epoch:  10   =====> Loss= 1.132301843  Validation accuracy= 0.7514
 Test accuracy= 0.7543
Epoch:  11   =====> Loss= 0.731297972  Validation accuracy= 0.8326
 Test accuracy= 0.8345
Epoch:  12   =====> Loss= 0.539246344  Validation accuracy= 0.8642
 Test accuracy= 0.8697
Epoch:  13   =====> Loss= 0.425185097  Validation accuracy= 0.8948
 Test accuracy= 0.8911
Epoch:  14   =====> Loss= 0.347723823  Validation accuracy= 0.9178
 Test accuracy= 0.9133
Epoch:  15   =====> Loss= 0.296964134  Validation accuracy= 0.9294
 Test accuracy= 0.9254
Epoch:  16   =====> Loss= 0.254887692  Validation accuracy= 0.9402
 Test accuracy= 0.9359
Epoch:  17   =====> Loss= 0.227199897  Validation accuracy= 0.9464
 Test accuracy= 0.9392
Epoch:  18   =====> Loss= 0.201696811  Validation accuracy= 0.9488
 Test accuracy= 0.9474
Epoch:  19   =====> Loss= 0.183556086  Validation accuracy= 0.9548
 Test accuracy= 0.9534
Epoch:  20   =====> Loss= 0.167839546  Validation accuracy= 0.9588
 Test accuracy= 0.9571
Epoch:  21   =====> Loss= 0.156564979  Validation accuracy= 0.9612
 Test accuracy= 0.9588
Epoch:  22   =====> Loss= 0.144359405  Validation accuracy= 0.9646
 Test accuracy= 0.9632
Epoch:  23   =====> Loss= 0.132515675  Validation accuracy= 0.9664
 Test accuracy= 0.9654
Epoch:  24   =====> Loss= 0.127158645  Validation accuracy= 0.9676
 Test accuracy= 0.967
Epoch:  25   =====> Loss= 0.120277383  Validation accuracy= 0.9696
 Test accuracy= 0.9693
Epoch:  26   =====> Loss= 0.112007220  Validation accuracy= 0.97  Te
st accuracy= 0.9707
Epoch:  27   =====> Loss= 0.109625914  Validation accuracy= 0.9704
 Test accuracy= 0.9716
Epoch:  28   =====> Loss= 0.103575878  Validation accuracy= 0.971  T
est accuracy= 0.972
Epoch:  29   =====> Loss= 0.098601557  Validation accuracy= 0.973  T
est accuracy= 0.9739
Epoch:  30   =====> Loss= 0.093816171  Validation accuracy= 0.974  T
```

```
est accuracy= 0.9744
Epoch:  31   =====> Loss= 0.092868912  Validation accuracy= 0.9726
 Test accuracy= 0.9741
Epoch:  32   =====> Loss= 0.089477438  Validation accuracy= 0.9738
 Test accuracy= 0.9749
Epoch:  33   =====> Loss= 0.084379114  Validation accuracy= 0.976  T
est accuracy= 0.9769
Epoch:  34   =====> Loss= 0.083824713  Validation accuracy= 0.9772
 Test accuracy= 0.9784
Epoch:  35   =====> Loss= 0.079959069  Validation accuracy= 0.9754
 Test accuracy= 0.9791
Epoch:  36   =====> Loss= 0.079306742  Validation accuracy= 0.9772
 Test accuracy= 0.9762
Epoch:  37   =====> Loss= 0.076031795  Validation accuracy= 0.9766
 Test accuracy= 0.9787
Epoch:  38   =====> Loss= 0.075798994  Validation accuracy= 0.9764
 Test accuracy= 0.9804
Epoch:  39   =====> Loss= 0.071249567  Validation accuracy= 0.976  T
est accuracy= 0.9806
Epoch:  40   =====> Loss= 0.071117505  Validation accuracy= 0.9784
 Test accuracy= 0.9807
Epoch:  41   =====> Loss= 0.068700804  Validation accuracy= 0.9788
 Test accuracy= 0.9808
Epoch:  42   =====> Loss= 0.068370116  Validation accuracy= 0.978  T
est accuracy= 0.9817
Epoch:  43   =====> Loss= 0.066696880  Validation accuracy= 0.9776
 Test accuracy= 0.9817
Epoch:  44   =====> Loss= 0.065036479  Validation accuracy= 0.9788
 Test accuracy= 0.9818
Epoch:  45   =====> Loss= 0.062617577  Validation accuracy= 0.9798
 Test accuracy= 0.9827
Epoch:  46   =====> Loss= 0.061715125  Validation accuracy= 0.98  Te
st accuracy= 0.9815
Epoch:  47   =====> Loss= 0.062542747  Validation accuracy= 0.9806
 Test accuracy= 0.9814
Epoch:  48   =====> Loss= 0.058317994  Validation accuracy= 0.9822
 Test accuracy= 0.9827
Epoch:  49   =====> Loss= 0.061187078  Validation accuracy= 0.9804
 Test accuracy= 0.9834
Epoch:  50   =====> Loss= 0.057029585  Validation accuracy= 0.98  Te
st accuracy= 0.9812
Epoch:  51   =====> Loss= 0.055684161  Validation accuracy= 0.9798
 Test accuracy= 0.9822
Epoch:  52   =====> Loss= 0.055113544  Validation accuracy= 0.9818
 Test accuracy= 0.9833
Epoch:  53   =====> Loss= 0.056086201  Validation accuracy= 0.9808
 Test accuracy= 0.9834
Epoch:  54   =====> Loss= 0.053827283  Validation accuracy= 0.9816
 Test accuracy= 0.9838
Epoch:  55   =====> Loss= 0.051751721  Validation accuracy= 0.9824
 Test accuracy= 0.9839
Epoch:  56   =====> Loss= 0.051553623  Validation accuracy= 0.982  T
est accuracy= 0.9843
Epoch:  57   =====> Loss= 0.052797941  Validation accuracy= 0.9828
 Test accuracy= 0.9846
Epoch:  58   =====> Loss= 0.048138580  Validation accuracy= 0.983  T
est accuracy= 0.9839
Epoch:  59   =====> Loss= 0.048249521  Validation accuracy= 0.9834
 Test accuracy= 0.985
Epoch:  60   =====> Loss= 0.048580303  Validation accuracy= 0.9846
```

```
 Test accuracy= 0.9849
Epoch:  61   =====> Loss= 0.047425265  Validation accuracy= 0.984  T
est accuracy= 0.9862
Epoch:  62   =====> Loss= 0.048077195  Validation accuracy= 0.9836
 Test accuracy= 0.9852
Epoch:  63   =====> Loss= 0.046083331  Validation accuracy= 0.9834
 Test accuracy= 0.9853
Epoch:  64   =====> Loss= 0.043985134  Validation accuracy= 0.9836
 Test accuracy= 0.9851
Epoch:  65   =====> Loss= 0.045851600  Validation accuracy= 0.9836
 Test accuracy= 0.9838
Epoch:  66   =====> Loss= 0.043868013  Validation accuracy= 0.9842
 Test accuracy= 0.985
Epoch:  67   =====> Loss= 0.043159742  Validation accuracy= 0.982  T
est accuracy= 0.9845
Epoch:  68   =====> Loss= 0.043726347  Validation accuracy= 0.9844
 Test accuracy= 0.9862
Epoch:  69   =====> Loss= 0.041593596  Validation accuracy= 0.9836
 Test accuracy= 0.9856
Epoch:  70   =====> Loss= 0.042528965  Validation accuracy= 0.9846
 Test accuracy= 0.9854
Epoch:  71   =====> Loss= 0.040201591  Validation accuracy= 0.9854
 Test accuracy= 0.9858
Epoch:  72   =====> Loss= 0.040041049  Validation accuracy= 0.9838
 Test accuracy= 0.9848
Epoch:  73   =====> Loss= 0.039296127  Validation accuracy= 0.985  T
est accuracy= 0.9856
Epoch:  74   =====> Loss= 0.039682834  Validation accuracy= 0.9846
 Test accuracy= 0.9852
Epoch:  75   =====> Loss= 0.037695736  Validation accuracy= 0.9848
 Test accuracy= 0.9848
Epoch:  76   =====> Loss= 0.038444788  Validation accuracy= 0.9846
 Test accuracy= 0.9856
Epoch:  77   =====> Loss= 0.037482553  Validation accuracy= 0.9864
 Test accuracy= 0.987
Epoch:  78   =====> Loss= 0.037332472  Validation accuracy= 0.9854
 Test accuracy= 0.9867
Epoch:  79   =====> Loss= 0.036921862  Validation accuracy= 0.9864
 Test accuracy= 0.986
Epoch:  80   =====> Loss= 0.036638310  Validation accuracy= 0.9852
 Test accuracy= 0.9864
Epoch:  81   =====> Loss= 0.034755491  Validation accuracy= 0.9868
 Test accuracy= 0.9867
Epoch:  82   =====> Loss= 0.036031529  Validation accuracy= 0.9862
 Test accuracy= 0.9865
Epoch:  83   =====> Loss= 0.034996243  Validation accuracy= 0.9854
 Test accuracy= 0.9857
Epoch:  84   =====> Loss= 0.032668294  Validation accuracy= 0.9874
 Test accuracy= 0.9868
Epoch:  85   =====> Loss= 0.035465714  Validation accuracy= 0.9864
 Test accuracy= 0.9859
Epoch:  86   =====> Loss= 0.032586588  Validation accuracy= 0.9874
 Test accuracy= 0.9878
Epoch:  87   =====> Loss= 0.033481765  Validation accuracy= 0.985  T
est accuracy= 0.9864
Epoch:  88   =====> Loss= 0.031420817  Validation accuracy= 0.9864
 Test accuracy= 0.9863
Epoch:  89   =====> Loss= 0.033429292  Validation accuracy= 0.9878
 Test accuracy= 0.9878
Epoch:  90   =====> Loss= 0.031617118  Validation accuracy= 0.9862
```

```
 Test accuracy= 0.9859
Epoch:  91   =====> Loss= 0.031943024  Validation accuracy= 0.9868
 Test accuracy= 0.9872
Epoch:  92   =====> Loss= 0.031059507  Validation accuracy= 0.9872
 Test accuracy= 0.9863
Epoch:  93   =====> Loss= 0.030869834  Validation accuracy= 0.9878
 Test accuracy= 0.9871
Epoch:  94   =====> Loss= 0.030361671  Validation accuracy= 0.9872
 Test accuracy= 0.9877
Epoch:  95   =====> Loss= 0.030616136  Validation accuracy= 0.987  T
est accuracy= 0.9871
Epoch:  96   =====> Loss= 0.028895149  Validation accuracy= 0.988  T
est accuracy= 0.9874
Epoch:  97   =====> Loss= 0.028924264  Validation accuracy= 0.9882
 Test accuracy= 0.9877
Epoch:  98   =====> Loss= 0.029672429  Validation accuracy= 0.9878
 Test accuracy= 0.9879
Epoch:  99   =====> Loss= 0.028289247  Validation accuracy= 0.9872
 Test accuracy= 0.9868
Epoch:  100   =====> Loss= 0.028785688  Validation accuracy= 0.9878
Test accuracy= 0.988
Epoch:  101   =====> Loss= 0.027101249  Validation accuracy= 0.9876
Test accuracy= 0.9867
Epoch:  102   =====> Loss= 0.027941262  Validation accuracy= 0.988
 Test accuracy= 0.9878
Epoch:  103   =====> Loss= 0.026916288  Validation accuracy= 0.9884
Test accuracy= 0.9884
Epoch:  104   =====> Loss= 0.025974303  Validation accuracy= 0.9884
Test accuracy= 0.988
Epoch:  105   =====> Loss= 0.028418892  Validation accuracy= 0.9866
Test accuracy= 0.9867
Epoch:  106   =====> Loss= 0.026032649  Validation accuracy= 0.9874
Test accuracy= 0.9873
Epoch:  107   =====> Loss= 0.026324572  Validation accuracy= 0.987
 Test accuracy= 0.9868
Epoch:  108   =====> Loss= 0.025792525  Validation accuracy= 0.9872
Test accuracy= 0.9877
Epoch:  109   =====> Loss= 0.024758348  Validation accuracy= 0.9874
Test accuracy= 0.9878
Epoch:  110   =====> Loss= 0.026273310  Validation accuracy= 0.988
 Test accuracy= 0.9875
Epoch:  111   =====> Loss= 0.024954320  Validation accuracy= 0.9868
Test accuracy= 0.9874
Epoch:  112   =====> Loss= 0.024047538  Validation accuracy= 0.988
 Test accuracy= 0.9878
Epoch:  113   =====> Loss= 0.024775138  Validation accuracy= 0.9878
Test accuracy= 0.9883
Epoch:  114   =====> Loss= 0.024530545  Validation accuracy= 0.9878
Test accuracy= 0.9885
Epoch:  115   =====> Loss= 0.024088073  Validation accuracy= 0.987
 Test accuracy= 0.988
Epoch:  116   =====> Loss= 0.023213179  Validation accuracy= 0.988
 Test accuracy= 0.9875
Epoch:  117   =====> Loss= 0.023963250  Validation accuracy= 0.9878
Test accuracy= 0.988
Epoch:  118   =====> Loss= 0.021890021  Validation accuracy= 0.9874
Test accuracy= 0.9874
Epoch:  119   =====> Loss= 0.023650439  Validation accuracy= 0.9874
Test accuracy= 0.9879
Epoch:  120   =====> Loss= 0.022322848  Validation accuracy= 0.9876
```

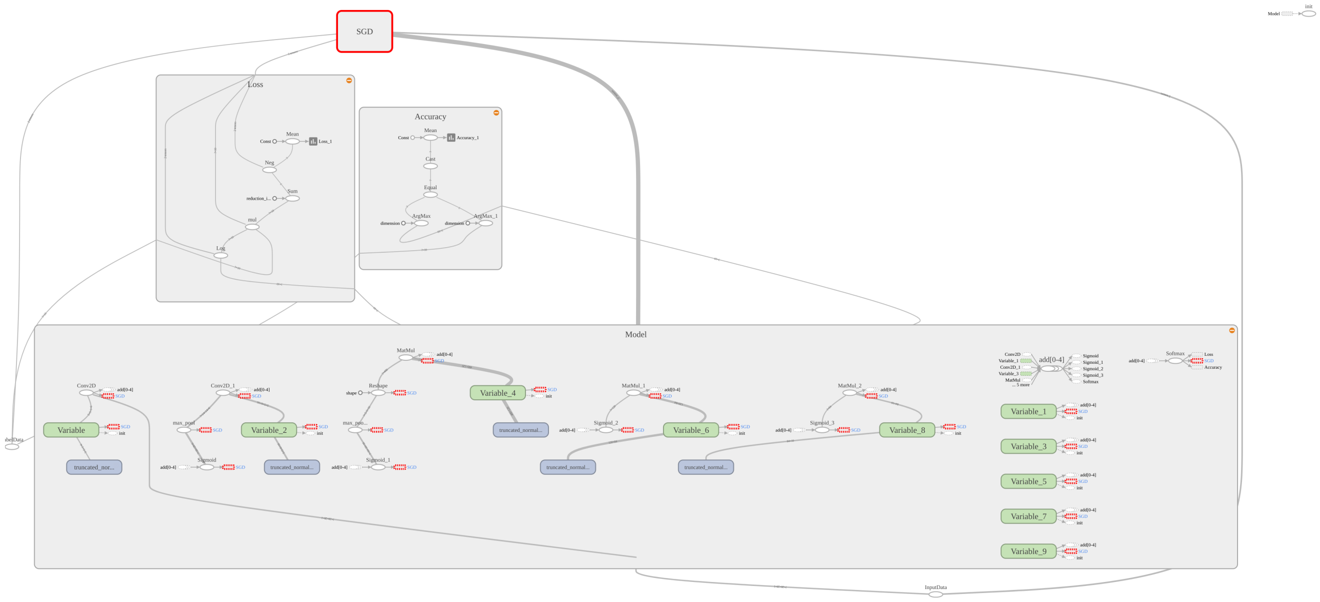
```
Test accuracy= 0.988
Epoch:  121   =====> Loss= 0.023320520  Validation accuracy= 0.9888
Test accuracy= 0.9883
Epoch:  122   =====> Loss= 0.021696907  Validation accuracy= 0.9876
Test accuracy= 0.9879
Epoch:  123   =====> Loss= 0.022045037  Validation accuracy= 0.9884
Test accuracy= 0.9887
Epoch:  124   =====> Loss= 0.021735647  Validation accuracy= 0.9878
Test accuracy= 0.9881
Epoch:  125   =====> Loss= 0.021409377  Validation accuracy= 0.9874
Test accuracy= 0.9877
Epoch:  126   =====> Loss= 0.021471361  Validation accuracy= 0.9884
Test accuracy= 0.9889
Epoch:  127   =====> Loss= 0.020926045  Validation accuracy= 0.9872
Test accuracy= 0.9883
Epoch:  128   =====> Loss= 0.020402798  Validation accuracy= 0.987
 Test accuracy= 0.9881
Epoch:  129   =====> Loss= 0.020623742  Validation accuracy= 0.9878
Test accuracy= 0.9886
Epoch:  130   =====> Loss= 0.020756424  Validation accuracy= 0.9886
Test accuracy= 0.9886
Epoch:  131   =====> Loss= 0.019686457  Validation accuracy= 0.9886
Test accuracy= 0.9888
Epoch:  132   =====> Loss= 0.020024373  Validation accuracy= 0.988
 Test accuracy= 0.9883
Epoch:  133   =====> Loss= 0.020844101  Validation accuracy= 0.9872
Test accuracy= 0.9886
Epoch:  134   =====> Loss= 0.018982376  Validation accuracy= 0.9888
Test accuracy= 0.9892
Epoch:  135   =====> Loss= 0.019082260  Validation accuracy= 0.988
 Test accuracy= 0.9879
Epoch:  136   =====> Loss= 0.019065446  Validation accuracy= 0.9876
Test accuracy= 0.9891
Epoch:  137   =====> Loss= 0.019546648  Validation accuracy= 0.9886
Test accuracy= 0.9892
Epoch:  138   =====> Loss= 0.018960108  Validation accuracy= 0.9876
Test accuracy= 0.9892
Epoch:  139   =====> Loss= 0.018055711  Validation accuracy= 0.9886
Test accuracy= 0.9886
Epoch:  140   =====> Loss= 0.018643468  Validation accuracy= 0.988
 Test accuracy= 0.9889
Epoch:  141   =====> Loss= 0.018267117  Validation accuracy= 0.9882
Test accuracy= 0.9891
Epoch:  142   =====> Loss= 0.018018327  Validation accuracy= 0.9882
Test accuracy= 0.9895
Epoch:  143   =====> Loss= 0.018379094  Validation accuracy= 0.9876
Test accuracy= 0.9892
Epoch:  144   =====> Loss= 0.017671612  Validation accuracy= 0.9874
Test accuracy= 0.989
Epoch:  145   =====> Loss= 0.016998068  Validation accuracy= 0.9878
Test accuracy= 0.9884
Epoch:  146   =====> Loss= 0.017596262  Validation accuracy= 0.9892
Test accuracy= 0.9892
Epoch:  147   =====> Loss= 0.017063023  Validation accuracy= 0.9882
Test accuracy= 0.9884
Epoch:  148   =====> Loss= 0.016875218  Validation accuracy= 0.9888
Test accuracy= 0.9902
Epoch:  149   =====> Loss= 0.017785960  Validation accuracy= 0.9876
Test accuracy= 0.989
Epoch:  150   =====> Loss= 0.016966669  Validation accuracy= 0.9878
```

```
Test accuracy= 0.9895
Epoch:  151   =====> Loss= 0.016676137  Validation accuracy= 0.9888
Test accuracy= 0.9898
Epoch:  152   =====> Loss= 0.015744936  Validation accuracy= 0.989
 Test accuracy= 0.9894
Epoch:  153   =====> Loss= 0.016398569  Validation accuracy= 0.9884
Test accuracy= 0.9895
Epoch:  154   =====> Loss= 0.015774943  Validation accuracy= 0.9882
Test accuracy= 0.9892
Epoch:  155   =====> Loss= 0.015730022  Validation accuracy= 0.9894
Test accuracy= 0.9892
Epoch:  156   =====> Loss= 0.015811218  Validation accuracy= 0.9882
Test accuracy= 0.9889
Epoch:  157   =====> Loss= 0.016122183  Validation accuracy= 0.9884
Test accuracy= 0.9884
Epoch:  158   =====> Loss= 0.015315281  Validation accuracy= 0.9878
Test accuracy= 0.9886
Epoch:  159   =====> Loss= 0.015055184  Validation accuracy= 0.9882
Test accuracy= 0.9895
Epoch:  160   =====> Loss= 0.015479419  Validation accuracy= 0.988
 Test accuracy= 0.9893
Epoch:  161   =====> Loss= 0.014629433  Validation accuracy= 0.9888
Test accuracy= 0.9896
Epoch:  162   =====> Loss= 0.014732498  Validation accuracy= 0.9884
Test accuracy= 0.9897
Epoch:  163   =====> Loss= 0.014117643  Validation accuracy= 0.9886
Test accuracy= 0.9895
Epoch:  164   =====> Loss= 0.014545544  Validation accuracy= 0.9884
Test accuracy= 0.9899
Epoch:  165   =====> Loss= 0.014448897  Validation accuracy= 0.9886
Test accuracy= 0.9901
Epoch:  166   =====> Loss= 0.014977406  Validation accuracy= 0.9884
Test accuracy= 0.9895
Epoch:  167   =====> Loss= 0.014063026  Validation accuracy= 0.989
 Test accuracy= 0.9898
Epoch:  168   =====> Loss= 0.013537712  Validation accuracy= 0.988
 Test accuracy= 0.99
Epoch:  169   =====> Loss= 0.014221393  Validation accuracy= 0.988
 Test accuracy= 0.9893
Epoch:  170   =====> Loss= 0.014508333  Validation accuracy= 0.988
 Test accuracy= 0.9886
Epoch:  171   =====> Loss= 0.012804839  Validation accuracy= 0.989
 Test accuracy= 0.9894
Epoch:  172   =====> Loss= 0.013299452  Validation accuracy= 0.9882
Test accuracy= 0.9892
Epoch:  173   =====> Loss= 0.013026615  Validation accuracy= 0.9898
Test accuracy= 0.9895
Epoch:  174   =====> Loss= 0.013447625  Validation accuracy= 0.988
 Test accuracy= 0.9898
Epoch:  175   =====> Loss= 0.012404188  Validation accuracy= 0.9882
Test accuracy= 0.9895
Epoch:  176   =====> Loss= 0.013155411  Validation accuracy= 0.988
 Test accuracy= 0.9899
Epoch:  177   =====> Loss= 0.013614445  Validation accuracy= 0.988
 Test accuracy= 0.9901
Epoch:  178   =====> Loss= 0.012569072  Validation accuracy= 0.9888
Test accuracy= 0.9896
Epoch:  179   =====> Loss= 0.012858638  Validation accuracy= 0.9884
Test accuracy= 0.9897
Epoch:  180   =====> Loss= 0.012033841  Validation accuracy= 0.9886
```

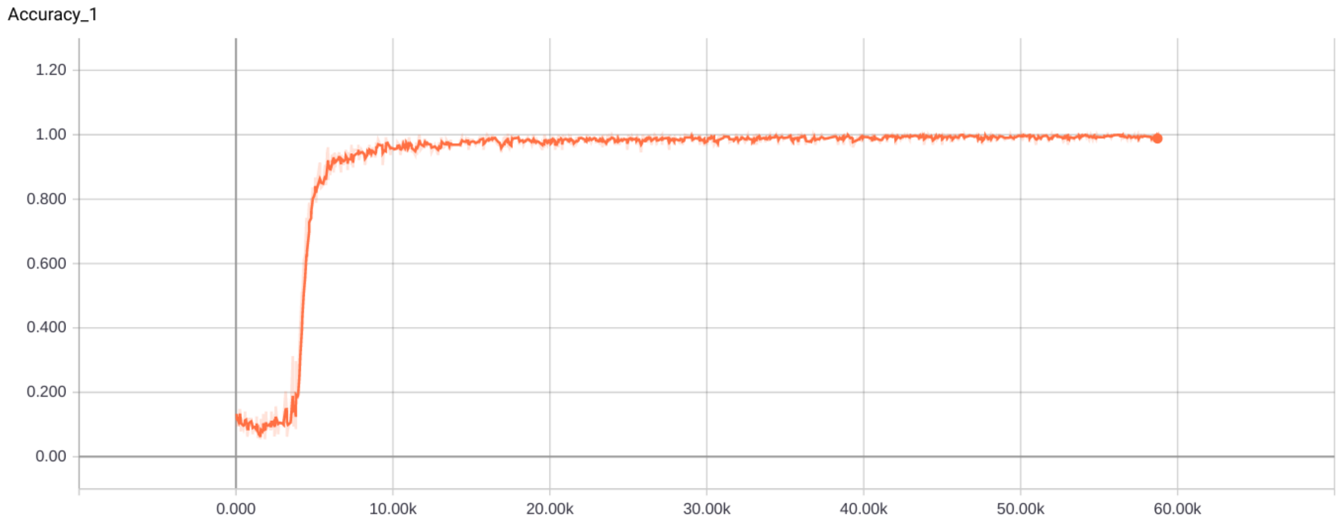
```
Test accuracy= 0.9895
Epoch:  181   =====> Loss= 0.012924304  Validation accuracy= 0.9882
Test accuracy= 0.9891
Epoch:  182   =====> Loss= 0.011285193  Validation accuracy= 0.989
 Test accuracy= 0.9904
Epoch:  183   =====> Loss= 0.012636733  Validation accuracy= 0.988
 Test accuracy= 0.99
Epoch:  184   =====> Loss= 0.011654639  Validation accuracy= 0.9886
Test accuracy= 0.9898
Epoch:  185   =====> Loss= 0.011734363  Validation accuracy= 0.987
 Test accuracy= 0.9902
Epoch:  186   =====> Loss= 0.012098399  Validation accuracy= 0.9884
Test accuracy= 0.9896
Epoch:  187   =====> Loss= 0.011165604  Validation accuracy= 0.9884
Test accuracy= 0.9889
Epoch:  188   =====> Loss= 0.012288753  Validation accuracy= 0.989
 Test accuracy= 0.9901
Epoch:  189   =====> Loss= 0.010551964  Validation accuracy= 0.9886
Test accuracy= 0.99
Epoch:  190   =====> Loss= 0.012254596  Validation accuracy= 0.9882
Test accuracy= 0.9893
Epoch:  191   =====> Loss= 0.010705987  Validation accuracy= 0.9884
Test accuracy= 0.9905
Epoch:  192   =====> Loss= 0.010890607  Validation accuracy= 0.9892
Test accuracy= 0.9898
Epoch:  193   =====> Loss= 0.011106896  Validation accuracy= 0.988
 Test accuracy= 0.9899
Epoch:  194   =====> Loss= 0.010891080  Validation accuracy= 0.989
 Test accuracy= 0.9906
Epoch:  195   =====> Loss= 0.010817055  Validation accuracy= 0.9888
Test accuracy= 0.9895
Epoch:  196   =====> Loss= 0.011130803  Validation accuracy= 0.9898
Test accuracy= 0.9894
Epoch:  197   =====> Loss= 0.009879846  Validation accuracy= 0.9884
Test accuracy= 0.99
Epoch:  198   =====> Loss= 0.010734334  Validation accuracy= 0.9886
Test accuracy= 0.9904
Epoch:  199   =====> Loss= 0.010990067  Validation accuracy= 0.9884
Test accuracy= 0.9896
Epoch:  200   =====> Loss= 0.009702761  Validation accuracy= 0.989
 Test accuracy= 0.9901
Training Finished!
Test accuracy: 0.9901
```

Question 2.1.6 : Use tensorBoard to visualise and save the LeNet5 Graph and all learning curves. Save all obtained figures in the folder **"TP2/MNIST_99_Challenge_Figures"**

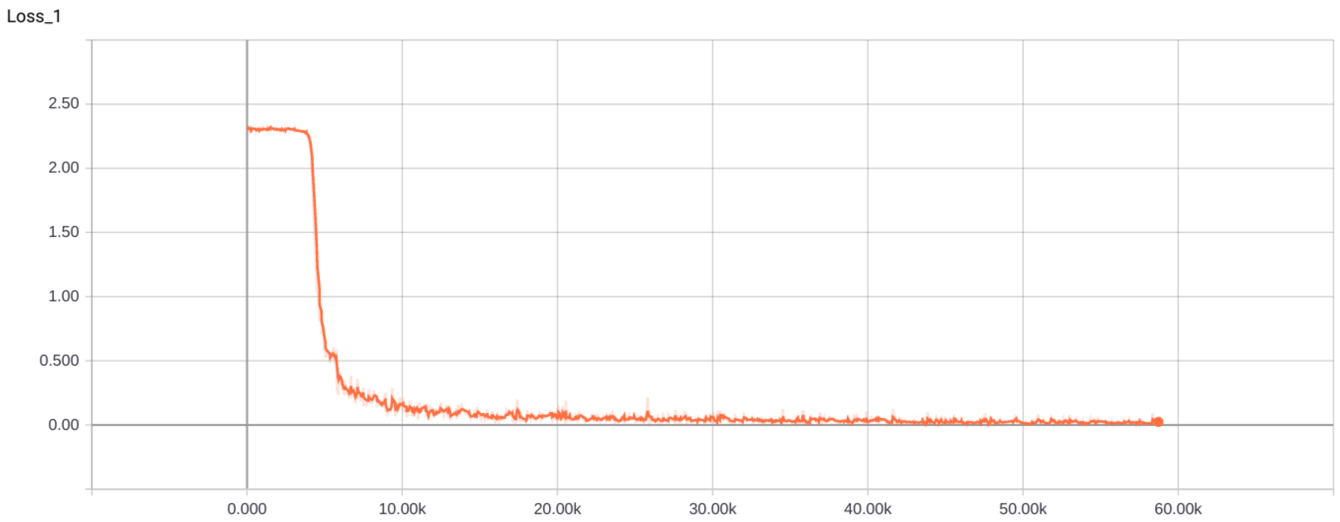
Graph Model



Accuracy



Loss



Part 2 : LeNET 5 Optimization

Question 2.2.1 Change the sigmoid function with a ReLU :

- Retrain your network with SGD and AdamOptimizer and then fill the table above :

Optimizer	**Gradient Descent**	**AdamOptimizer**
Validation Accuracy	0.9858	0.992
Testing Accuracy	0.9842	0.9892
Training Time	6176s	465s

- Try with different learning rates for each Optimizer (0.0001 and 0.001) and different Batch sizes (50 and 128) for 20000 Epochs.
- For each optimizer, plot (on the same curve) the **testing accuracies** function to **(learning rate, batch size)**

- Did you reach the 99% accuracy ? What are the optimal parametres that gave you the best results?

Below, we print the results for the different training models, their parameters and the corresponding accuracies.

In [6]:

```
import time
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

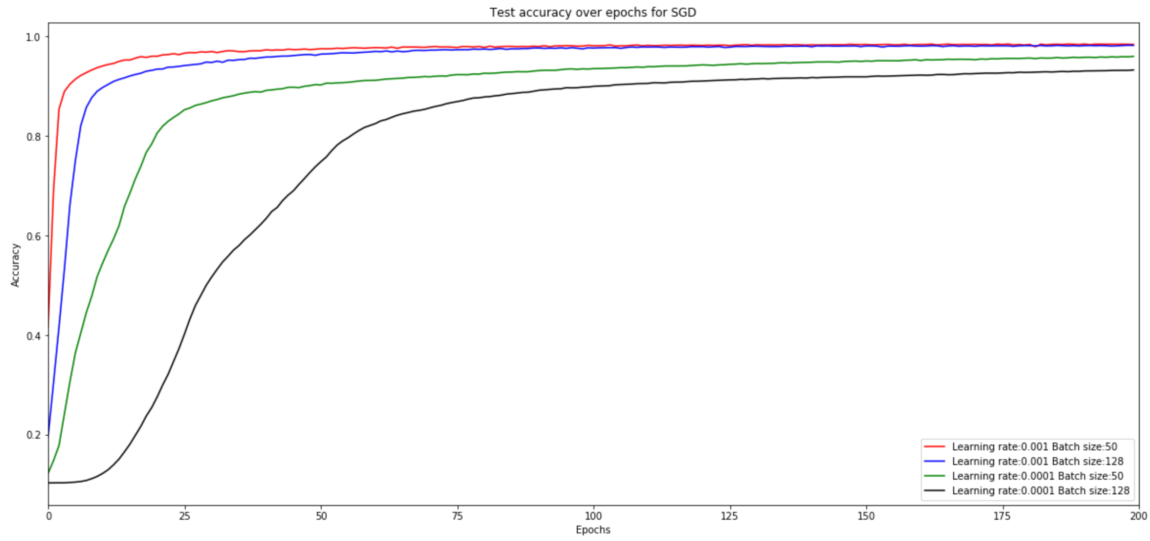
In [11]:

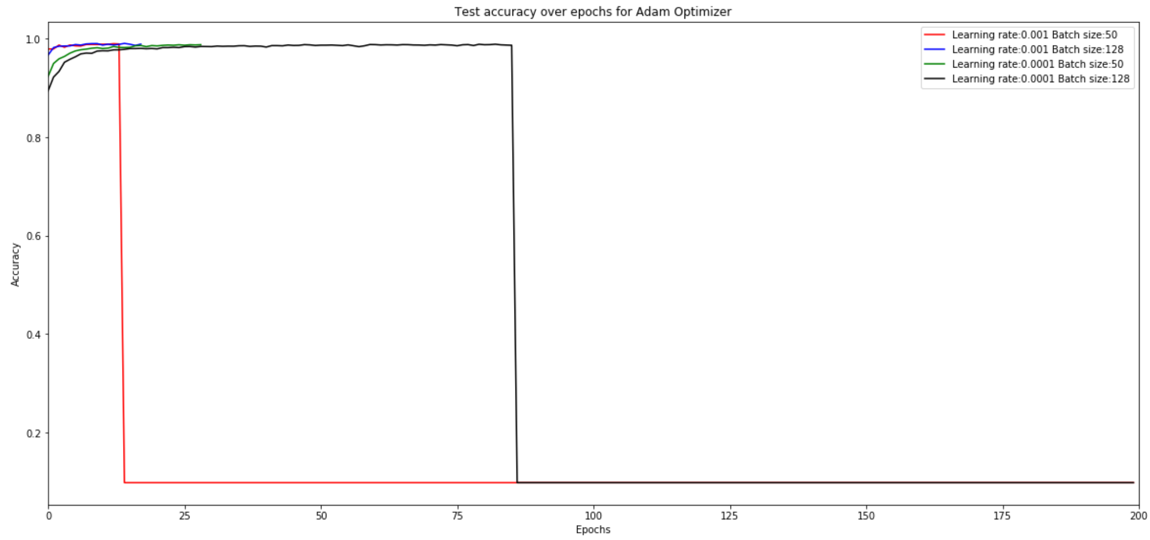
```
learning_rates = [0.001, 0.0001]
batch_sizes = [50, 128]
optNames = ["SGD", "Adam"]
training_epochs = 200
disp_step = 20
# training_epochs = 3
# disp_step = 1

results = []
for on in optNames:
    for lr in learning_rates:
        for bs in batch_sizes:
            print("Learning rate:", lr, "Batch size:", bs, "optimizer:", on)
            t1 = time.time()
            val_history, test_history = train(learning_rate=lr, \
                                              training_epochs=training_epochs, bat
ch_size=bs, \
                                              display_step=disp_step,
optFunction=on, verbose=False, transfer="ReLU")
            t2 = time.time() - t1
            print("\t====> Time:", t2, "Validation accuracy:", val_history[-1],
"Test accuracy:", test_history[-1], \
                  "\n---------------------------------------------")
            results.append((lr, bs, on, t2, test_history, val_history))
print("Optimization Finished!")
```

```
Learning rate: 0.001 Batch size: 50 optimizer: SGD
        ====> Time: 6176.0356233119965 Validation accuracy: 0.9858 T
est accuracy: 0.9842
----------------------------------------------
Learning rate: 0.001 Batch size: 128 optimizer: SGD
        ====> Time: 5185.878306150436 Validation accuracy: 0.9814 Te
st accuracy: 0.9822
----------------------------------------------
Learning rate: 0.0001 Batch size: 50 optimizer: SGD
        ====> Time: 6140.959002494812 Validation accuracy: 0.9626 Te
st accuracy: 0.9602
----------------------------------------------
Learning rate: 0.0001 Batch size: 128 optimizer: SGD
        ====> Time: 5140.468836069107 Validation accuracy: 0.9352 Te
st accuracy: 0.9329
----------------------------------------------
Learning rate: 0.001 Batch size: 50 optimizer: Adam
        ====> Time: 6156.195225954056 Validation accuracy: 0.0958 Te
st accuracy: 0.098
----------------------------------------------
Learning rate: 0.001 Batch size: 128 optimizer: Adam
        ====> Time: 465.2290246486664 Validation accuracy: 0.992 Tes
t accuracy: 0.9892
----------------------------------------------
Learning rate: 0.0001 Batch size: 50 optimizer: Adam
        ====> Time: 904.2078680992126 Validation accuracy: 0.9904 Te
st accuracy: 0.988
----------------------------------------------
Learning rate: 0.0001 Batch size: 128 optimizer: Adam
        ====> Time: 5160.4019911289215 Validation accuracy: 0.0958 T
est accuracy: 0.098
----------------------------------------------
Optimization Finished!
```

In [ ]:

```
plt.figure(figsize=(20,20))
plt.subplot(211)
max_epochs = max([len(results[0][4]), len(results[1][4]), len(results[2][4]), le
n(results[3][4])])
plt.plot(np.arange(len(results[0][4])), results[0][4], c="r", \
         label="Learning rate:" + str(results[0][0]) + " Batch size:" + str(resu
lts[0][1]))
plt.plot(np.arange(len(results[1][4])), results[1][4], c="b", \
         label="Learning rate:" + str(results[1][0]) + " Batch size:" + str(resu
lts[1][1]))
plt.plot(np.arange(len(results[2][4])), results[2][4], c="g", \
         label="Learning rate:" + str(results[2][0]) + " Batch size:" + str(resu
lts[2][1]))
plt.plot(np.arange(len(results[3][4])), results[3][4], c="k", \
         label="Learning rate:" + str(results[3][0]) + " Batch size:" + str(resu
lts[3][1]))
plt.legend()
plt.title("Test accuracy over epochs for SGD")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.xlim((0, max_epochs))

plt.subplot(212)
max_epochs = max([len(results[4][4]), len(results[5][4]), len(results[6][4]), le
n(results[7][4])])
plt.plot(np.arange(len(results[4][4])), results[4][4], c="r", \
         label="Learning rate:" + str(results[4][0]) + " Batch size:" + str(resu
lts[4][1]))
plt.plot(np.arange(len(results[5][4])), results[5][4], c="b", \
         label="Learning rate:" + str(results[5][0]) + " Batch size:" + str(resu
lts[5][1]))
plt.plot(np.arange(len(results[6][4])), results[6][4], c="g", \
         label="Learning rate:" + str(results[6][0]) + " Batch size:" + str(resu
lts[6][1]))
plt.plot(np.arange(len(results[7][4])), results[7][4], c="k", \
         label="Learning rate:" + str(results[7][0]) + " Batch size:" + str(resu
lts[7][1]))
plt.legend()
plt.title("Test accuracy over epochs for Adam Optimizer")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.xlim((0, max_epochs))
plt.show()
```





We sometimes obtain these drops in accuracy probably caused by vanishing gradients.

Question 2.2.2 What about applying a dropout layer on the Fully connected layer and then retraining the model with the best Optimizer and parameters (Learning rate and Batch size) obtained in *Question 2.2.1* ? (probability to keep units=0.75). For this stage ensure that the keep prob is set to 1.0 to evaluate the performance of the network including all nodes.

In [8]:

```
lr = 0.001
bs = 128
opt = "Adam"
training_epochs = 100
disp_step = 10
kp = 0.75

t1 = time.time()
val_history_do, test_history_do = train(learning_rate=lr, training_epochs=traini
ng_epochs, batch_size=bs, \
                    display_step=disp_step, optFunction=opt, verbose=True, trans
fer="ReLU", keep_probability=kp)
t2 = time.time() - t1
print("====> Time:", t2, "Validation accuracy:", val_history_do[-1], "Test accur
acy:", test_history_do[-1])
print("Optimization Finished!")
```

```
Start Training!
Epoch:  10   =====> Loss= 0.020439351  Validation accuracy= 0.9876
 Test accuracy= 0.99
Validation Accuracy over 99% reached after 16 epochs
Training Finished!
Test accuracy: 0.9899
====> Time: 277.9252426624298 Validation accuracy: 0.99 Test accurac
y: 0.9899
Optimization Finished!
```

With this configuration, we are able to reach 99% accuracy in a shorter time. Dropout is a useful technique and we see it here.