Deep Learning

Convolutional Neural Network (CNN) for Handwritten Digits Recognition

Corentin RAFFLIN

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 below to load the MNIST data that comes with TensorFlow. You will use this data in Section 1 and Section 2.

In [ ]:

```
import tensorflow as tf
import numpy as np
import warnings
from tensorflow.examples.tutorials.mnist import input_data
from sklearn.utils import shuffle
from time import time
#Removing warnings
warnings.simplefilter('ignore')

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)))

epsilon = 1e-10 # this is a parameter you will use later
```

```
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 first lab.

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 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 [ ]:

```
#STEP 1

# Parameters
learning_rate = 0.01
training_epochs = 40
batch_size = 128
display_step = 1
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 Graph visualiz
ation 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
    # We use tf.clip_by_value to avoid having too low numbers in the log function
    cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(tf.clip_by_value(pred, epsilon, 1.0)), re
duction_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, shuffle=(i==0))
            # 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}".format(avg_cost
))

    print("Optimization Finished!")
    summary_writer.flush()

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

```
Epoch:  01   =====> Loss= 1.288586539
Epoch:  02   =====> Loss= 0.732474971
Epoch:  03   =====> Loss= 0.600285978
Epoch:  04   =====> Loss= 0.536354847
Epoch:  05   =====> Loss= 0.497779992
Epoch:  06   =====> Loss= 0.471399856
Epoch:  07   =====> Loss= 0.451380478
Epoch:  08   =====> Loss= 0.435802482
Epoch:  09   =====> Loss= 0.423480794
Epoch:  10   =====> Loss= 0.413272987
Epoch:  11   =====> Loss= 0.404111989
Epoch:  12   =====> Loss= 0.396864860
Epoch:  13   =====> Loss= 0.390325102
Epoch:  14   =====> Loss= 0.384214350
Epoch:  15   =====> Loss= 0.379292450
Epoch:  16   =====> Loss= 0.374498579
Epoch:  17   =====> Loss= 0.370195421
Epoch:  18   =====> Loss= 0.366471430
Epoch:  19   =====> Loss= 0.363161504
Epoch:  20   =====> Loss= 0.359470057
Epoch:  21   =====> Loss= 0.356625458
Epoch:  22   =====> Loss= 0.353857386
Epoch:  23   =====> Loss= 0.351068103
Epoch:  24   =====> Loss= 0.348683886
Epoch:  25   =====> Loss= 0.346556067
Epoch:  26   =====> Loss= 0.344203966
Epoch:  27   =====> Loss= 0.342275714
Epoch:  28   =====> Loss= 0.340429829
Epoch:  29   =====> Loss= 0.338404988
Epoch:  30   =====> Loss= 0.336588143
Epoch:  31   =====> Loss= 0.335070862
Epoch:  32   =====> Loss= 0.333266099
Epoch:  33   =====> Loss= 0.331748201
Epoch:  34   =====> Loss= 0.330545522
Epoch:  35   =====> Loss= 0.329071804
Epoch:  36   =====> Loss= 0.327823580
Epoch:  37   =====> Loss= 0.326598606
Epoch:  38   =====> Loss= 0.325235325
Epoch:  39   =====> Loss= 0.324201516
Epoch:  40   =====> Loss= 0.322875760
Optimization Finished!
Accuracy: 0.916
```

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:

- **Open a Terminal and run the command line "tensorboard --logdir=lab_2/log_files/"**
- **Click on "Tensorboard web interface" in Zoe**

Enjoy It !!

Section 2 : The 99% MNIST Challenge !

Part 1 : LeNet5 implementation

You are now familar with TensorFlow and TensorBoard. In this section, you are to build, train and test the baseline [LeNet-5](#) model for the MNIST digits recognition problem.

Then, you will make some optimizations to get more than 99% of accuracy.

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

The LeNet architecture takes a 28x28xC image as input, where C is the number of color channels. Since MNIST images are grayscale, C is 1 in this case.

Layer 1 - Convolution (5x5): The output shape should be 28x28x6. Activation: ReLU. MaxPooling: The output shape should be 14x14x6.

Layer 2 - Convolution (5x5): The output shape should be 10x10x16. Activation: ReLU. MaxPooling: 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 tf.reshape.

Layer 3 - Fully Connected: This should have 120 outputs. Activation: ReLU.

Layer 4 - Fully Connected: This should have 84 outputs. Activation: ReLU.

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.

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

In [ ]:

```
# 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., shape=shape)
  return tf.Variable(initial)
```

In [ ]:

```
def LeNet5_Model(image):
    '''image shape [batch, in_height, in_width, in_channels]'''
```

```
    #Layer 1 : Convolution 5*5
    weight1 = weight_variable([5, 5, 1, 6]) # shape [filter_height, filter_width, in_channe
ls, out_channels]
    bias1 = bias_variable([6]) # shape (out_channels)
    conv1 = tf.nn.conv2d(image, weight1, strides=[1,1,1,1], padding='SAME')
    act1 = tf.nn.relu(conv1 + bias1)
    pool1 = tf.nn.max_pool(act1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')

    #Layer 2 : Convolution 5*5
    weight2 = weight_variable([5, 5, 6, 16])
    bias2 = bias_variable([16])
    conv2 = tf.nn.conv2d(pool1, weight2, strides=[1,1,1,1], padding='VALID')
    act2 = tf.nn.relu(conv2 + bias2)
    pool2 = tf.nn.max_pool(act2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')

    #Flatten layer
    flatten = tf.reshape(pool2, [-1, 5*5*16])

    #Layer 3 : Fully Connected
    weight3 = weight_variable([5*5*16, 120])
    bias3 = bias_variable([120])
    act3 = tf.nn.relu(tf.matmul(flatten, weight3) + bias3)

    #Layer 4 : Fully Connected
    weight4 = weight_variable([120, 84])
    bias4 = bias_variable([84])
    act4 = tf.nn.relu(tf.matmul(act3, weight4) + bias4)

    #Layer 5 : Fully Connected
    weight5 = weight_variable([84, 10])
    bias5 = bias_variable([10])
    act5 = tf.nn.softmax(tf.matmul(act4, weight5) + bias5)

    return act5
```

Question 2.1.2. Calculate the number of parameters of this model

In [ ]:

```
param_layer1 = 5*5*1*6 + 6 #filter_height*filter_width*in_channels*out_channels + bias
param_layer2 = 5*5*6*16 + 16
param_layer3 = (5*5*16)*120 + 120 #input * output + bias
param_layer4 = 120*84 + 84
param_layer5 = 84*10 + 10

param = param_layer1 + param_layer2 + param_layer3 + param_layer4 + param_layer5
print("There are %d parameters in this model" % param)
```

```
There are 61706 parameters in this model
```

Question 2.1.3. Define your model, its accuracy and the loss function according to the following parameters (you can look at Section 1 to see what is expected):

```
Learning rate: 0.001
Loss Fucntion: Cross-entropy
Optimizer: tf.train.GradientDescentOptimizer
Number of epochs: 40
Batch size: 128
```

In [ ]:

```
tf.reset_default_graph() # reset the default graph before defining a new model

# Parameters
```

```
learning_rate = 0.001
training_epochs = 40
batch_size = 128
logs_path = 'log_files/'


x = tf.placeholder(tf.float32, [None, 28, 28, 1], name='InputData')
y = tf.placeholder(tf.float32, [None, 10], name='LabelData')
#Reshaping
X_train = X_train.reshape(-1, 28, 28, 1)
X_validation = X_validation.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

#Model
with tf.name_scope('Model'):
    pred = LeNet5_Model(x)
#Loss function
with tf.name_scope('Loss'):
    cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(tf.clip_by_value(pred, epsilon, 1.0)), re
duction_indices=1))
#Optimizer
with tf.name_scope('SGD'):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
#Accuracy
with tf.name_scope('Accuracy'):
    acc = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    acc = tf.reduce_mean(tf.cast(acc, tf.float32))
```

Question 2.1.4. Implement the evaluation function for accuracy computation

In [ ]:

```
def evaluate(logits, labels):
    # logits will be the outputs of your model, labels will be one-hot vectors corresponding
to the actual labels
    # logits and labels are numpy arrays
    # this function should return the accuracy of your model
    acc = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))
    acc = tf.reduce_mean(tf.cast(acc, tf.float32))
    return acc
```

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 [ ]:

```
# Initializing the variables
init = tf.global_variables_initializer()
# Create a summary to monitor cost tensor
tf.summary.scalar("Loss_LeNet-5_SGD", cost)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("Accuracy_LeNet-5_SGD", acc)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()
```

In [ ]:

```
def train(init, sess, logs_path, n_epochs, batch_size, optimizer, cost, merged_summary_op):
    # optimizer and cost are the same kinds of objects as in Section 1
    # Train your model
```

```
    global X_train, y_train, X_validation, y_validation, X_test, y_test

    sess.run(init)

    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_graph())

    #Initialize saver
    saver = tf.train.Saver()

    total_batch = int(X_train.shape[0]/batch_size)

    start_time = time()
    # Training cycle
    for epoch in range(n_epochs):
        avg_cost = 0.

        # Shuffling the data before each epoch
        X_train, y_train = shuffle(X_train, y_train)

        # Loop over all batches
        for i in range(total_batch):
            #Take new batch
            batch_xs, batch_ys = X_train[batch_size*i:batch_size*(i+1)], y_train[batch_size
*i:batch_size*(i+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})
            # Write logs at every iteration
            summary_writer.add_summary(summary, epoch * total_batch + i)
            # Compute average loss
            avg_cost += c / total_batch

        #Compute accuracy
        val_acc = acc.eval({x: X_validation, y: y_validation})
        test_acc = acc.eval({x: X_test, y: y_test})

        # Display logs per epoch
        if (epoch+1) % 1 == 0:
            print("Epoch: ", '%02d' % (epoch+1), "  =====> Loss=", "{:.9f}".format(avg_cost
),
                  " ====> Testing accuracy =", "{:.9f}".format(test_acc),
                  " ====> Validation accuracy =", "{:.9f}".format(val_acc), )

    print("Training time :", time() - start_time)
    #Save model after training
    save_path = saver.save(sess, logs_path + 'model' + '_' + optimizer.name)
    print("Model saved in path: %s" % save_path)

    summary_writer.flush()

    # Test model
    # Calculate accuracy
    print("Final testing accuracy:", acc.eval({x: X_test, y: y_test}))
```

In [ ]:

```
with tf.Session() as sess:
    train(init, sess, logs_path, training_epochs, batch_size, optimizer, cost, merged_summa
ry_op)
```

```
Epoch:  01   =====> Loss= 2.298148755  ====> Testing accuracy = 0.149900004  ====> Validatio
n accuracy = 0.158800006
Epoch:  02   =====> Loss= 2.278185468  ====> Testing accuracy = 0.224199995  ====> Validatio
n accuracy = 0.239800006
Epoch:  03   =====> Loss= 2.248713539  ====> Testing accuracy = 0.307399988  ====> Validatio
```

```
n accuracy = 0.324999988
Epoch:  04   =====> Loss= 2.194495717  ====> Testing accuracy = 0.375499994  ====> Validatio
n accuracy = 0.386200011
Epoch:  05   =====> Loss= 2.074979356  ====> Testing accuracy = 0.488999993  ====> Validatio
n accuracy = 0.507000029
Epoch:  06   =====> Loss= 1.778755422  ====> Testing accuracy = 0.669300020  ====> Validatio
n accuracy = 0.683200002
Epoch:  07   =====> Loss= 1.244942585  ====> Testing accuracy = 0.801699996  ====> Validatio
n accuracy = 0.790400028
Epoch:  08   =====> Loss= 0.803754446  ====> Testing accuracy = 0.847400010  ====> Validatio
n accuracy = 0.835200012
Epoch:  09   =====> Loss= 0.594938605  ====> Testing accuracy = 0.869899988  ====> Validatio
n accuracy = 0.860400021
Epoch:  10   =====> Loss= 0.494891864  ====> Testing accuracy = 0.887700021  ====> Validatio
n accuracy = 0.878400028
Epoch:  11   =====> Loss= 0.435313439  ====> Testing accuracy = 0.897000015  ====> Validatio
n accuracy = 0.890799999
Epoch:  12   =====> Loss= 0.394286241  ====> Testing accuracy = 0.901600003  ====> Validatio
n accuracy = 0.898599982
Epoch:  13   =====> Loss= 0.363556993  ====> Testing accuracy = 0.910600007  ====> Validatio
n accuracy = 0.909399986
Epoch:  14   =====> Loss= 0.338756653  ====> Testing accuracy = 0.914699972  ====> Validatio
n accuracy = 0.914200008
Epoch:  15   =====> Loss= 0.319165956  ====> Testing accuracy = 0.919900000  ====> Validatio
n accuracy = 0.916000009
Epoch:  16   =====> Loss= 0.302629984  ====> Testing accuracy = 0.925300002  ====> Validatio
n accuracy = 0.922399998
Epoch:  17   =====> Loss= 0.288350765  ====> Testing accuracy = 0.928399980  ====> Validatio
n accuracy = 0.925400019
Epoch:  18   =====> Loss= 0.276248129  ====> Testing accuracy = 0.931100011  ====> Validatio
n accuracy = 0.928399980
Epoch:  19   =====> Loss= 0.265580019  ====> Testing accuracy = 0.932500005  ====> Validatio
n accuracy = 0.928399980
Epoch:  20   =====> Loss= 0.255937709  ====> Testing accuracy = 0.935800016  ====> Validatio
n accuracy = 0.933399975
Epoch:  21   =====> Loss= 0.247342657  ====> Testing accuracy = 0.935999990  ====> Validatio
n accuracy = 0.935000002
Epoch:  22   =====> Loss= 0.239377569  ====> Testing accuracy = 0.937200010  ====> Validatio
n accuracy = 0.936200023
Epoch:  23   =====> Loss= 0.232008204  ====> Testing accuracy = 0.938799977  ====> Validatio
n accuracy = 0.939800024
Epoch:  24   =====> Loss= 0.224953218  ====> Testing accuracy = 0.941299975  ====> Validatio
n accuracy = 0.939999998
Epoch:  25   =====> Loss= 0.218485423  ====> Testing accuracy = 0.942399979  ====> Validatio
n accuracy = 0.942399979
Epoch:  26   =====> Loss= 0.212667487  ====> Testing accuracy = 0.944400012  ====> Validatio
n accuracy = 0.944199979
Epoch:  27   =====> Loss= 0.207095804  ====> Testing accuracy = 0.944500029  ====> Validatio
n accuracy = 0.944999993
Epoch:  28   =====> Loss= 0.202158899  ====> Testing accuracy = 0.947700024  ====> Validatio
n accuracy = 0.948000014
Epoch:  29   =====> Loss= 0.197288068  ====> Testing accuracy = 0.948000014  ====> Validatio
n accuracy = 0.946799994
Epoch:  30   =====> Loss= 0.192791463  ====> Testing accuracy = 0.950399995  ====> Validatio
n accuracy = 0.950399995
Epoch:  31   =====> Loss= 0.188420314  ====> Testing accuracy = 0.950800002  ====> Validatio
n accuracy = 0.950800002
Epoch:  32   =====> Loss= 0.184444467  ====> Testing accuracy = 0.952799976  ====> Validatio
n accuracy = 0.951799989
Epoch:  33   =====> Loss= 0.180556170  ====> Testing accuracy = 0.953999996  ====> Validatio
n accuracy = 0.952600002
Epoch:  34   =====> Loss= 0.176957561  ====> Testing accuracy = 0.954299986  ====> Validatio
n accuracy = 0.955200016
Epoch:  35   =====> Loss= 0.173188459  ====> Testing accuracy = 0.953899980  ====> Validatio
n accuracy = 0.954800010
Epoch:  36   =====> Loss= 0.169962596  ====> Testing accuracy = 0.954800010  ====> Validatio
n accuracy = 0.957199991
```

```
Epoch:  37   =====> Loss= 0.166648968  ====> Testing accuracy = 0.956499994  ====> Validatio
n accuracy = 0.956799984
Epoch:  38   =====> Loss= 0.163419584  ====> Testing accuracy = 0.956900001  ====> Validatio
n accuracy = 0.957799971
Epoch:  39   =====> Loss= 0.160550396  ====> Testing accuracy = 0.957799971  ====> Validatio
n accuracy = 0.957799971
Epoch:  40   =====> Loss= 0.157792618  ====> Testing accuracy = 0.958700001  ====> Validatio
n accuracy = 0.959200025
Training time : 711.790855884552
Model saved in path: log_files/model_SGD/GradientDescent
Final testing accuracy: 0.9587
```

Question 2.1.6 : Use TensorBoard to visualise and save loss and accuracy curves. You will save figures in the folder "lab_2/MNIST_figures" and display them in your notebook.

Part 2 : LeNET 5 Optimization

Question 2.2.1

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

Optimizer	Gradient Descent	AdamOptimizer
Testing Accuracy	0.9587	0.9914
Training Time	712s	710s

- **Which optimizer gives the best accuracy on test data?**

Your answer: The Adam optimizer gives a better accuracy on test data (0.9914 vs 0.9587 with Gradient Descent) for similar parameters. We also notice that the training time is quite the same with the two optimizers. We also notice that the model with Adam optimizer converges way faster. Indeed in a single epoch we obtain a loss of about 0.34, whereas we obtained a similar loss only after 14 epochs with Gradient Descent. Similarly in a single epoch the testing accuracy is already better than the one after 40 epochs with Gradient Descent.

In [ ]:

```
##Same as before, just changing the optimizer

tf.reset_default_graph()

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

#Model
with tf.name_scope('Model'):
    pred = LeNet5_Model(x)
#Loss function
with tf.name_scope('Loss'):
    cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(tf.clip_by_value(pred, epsilon, 1.0)), re
duction_indices=1))
#Optimizer
with tf.name_scope('Adam'):
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
#Accuracy
with tf.name_scope('Accuracy'):
    acc = evaluate(pred, y)

# Initializing the variables
init = tf.global_variables_initializer()
# Create a summary to monitor cost tensor
tf.summary.scalar("Loss_LeNet-5_Adam", cost)
```

```
# Create a summary to monitor accuracy tensor
tf.summary.scalar("Accuracy_LeNet-5_Adam", acc)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()
```

In [ ]:

```
with tf.Session() as sess:
    train(init, sess, logs_path, training_epochs, batch_size, optimizer, cost, merged_summa
ry_op)
```

```
Epoch:  01   =====> Loss= 0.337192377  ====> Testing accuracy = 0.971199989  ====> Validatio
n accuracy = 0.971400023
Epoch:  02   =====> Loss= 0.093284122  ====> Testing accuracy = 0.975899994  ====> Validatio
n accuracy = 0.973999977
Epoch:  03   =====> Loss= 0.069581516  ====> Testing accuracy = 0.982100010  ====> Validatio
n accuracy = 0.983200014
Epoch:  04   =====> Loss= 0.056758887  ====> Testing accuracy = 0.987299979  ====> Validatio
n accuracy = 0.985199988
Epoch:  05   =====> Loss= 0.044753444  ====> Testing accuracy = 0.987500012  ====> Validatio
n accuracy = 0.987200022
Epoch:  06   =====> Loss= 0.036672540  ====> Testing accuracy = 0.988099992  ====> Validatio
n accuracy = 0.987999976
Epoch:  07   =====> Loss= 0.033269419  ====> Testing accuracy = 0.990499973  ====> Validatio
n accuracy = 0.986999989
Epoch:  08   =====> Loss= 0.029368611  ====> Testing accuracy = 0.988200009  ====> Validatio
n accuracy = 0.988600016
Epoch:  09   =====> Loss= 0.024178660  ====> Testing accuracy = 0.987699986  ====> Validatio
n accuracy = 0.986999989
Epoch:  10   =====> Loss= 0.022274069  ====> Testing accuracy = 0.989199996  ====> Validatio
n accuracy = 0.988600016
Epoch:  11   =====> Loss= 0.018577916  ====> Testing accuracy = 0.990400016  ====> Validatio
n accuracy = 0.989000022
Epoch:  12   =====> Loss= 0.015998311  ====> Testing accuracy = 0.989099979  ====> Validatio
n accuracy = 0.985800028
Epoch:  13   =====> Loss= 0.015998260  ====> Testing accuracy = 0.990599990  ====> Validatio
n accuracy = 0.990400016
Epoch:  14   =====> Loss= 0.014665965  ====> Testing accuracy = 0.991400003  ====> Validatio
n accuracy = 0.988600016
Epoch:  15   =====> Loss= 0.011703846  ====> Testing accuracy = 0.988699973  ====> Validatio
n accuracy = 0.990199983
Epoch:  16   =====> Loss= 0.011374031  ====> Testing accuracy = 0.987399995  ====> Validatio
n accuracy = 0.986400008
Epoch:  17   =====> Loss= 0.011589384  ====> Testing accuracy = 0.989300013  ====> Validatio
n accuracy = 0.988200009
Epoch:  18   =====> Loss= 0.008555054  ====> Testing accuracy = 0.989799976  ====> Validatio
n accuracy = 0.988399982
Epoch:  19   =====> Loss= 0.010774759  ====> Testing accuracy = 0.989499986  ====> Validatio
n accuracy = 0.987399995
Epoch:  20   =====> Loss= 0.006664998  ====> Testing accuracy = 0.989899993  ====> Validatio
n accuracy = 0.989799976
Epoch:  21   =====> Loss= 0.008171073  ====> Testing accuracy = 0.989799976  ====> Validatio
n accuracy = 0.989199996
Epoch:  22   =====> Loss= 0.010182011  ====> Testing accuracy = 0.987299979  ====> Validatio
n accuracy = 0.986400008
Epoch:  23   =====> Loss= 0.006053188  ====> Testing accuracy = 0.990800023  ====> Validatio
n accuracy = 0.988600016
Epoch:  24   =====> Loss= 0.005813461  ====> Testing accuracy = 0.990800023  ====> Validatio
n accuracy = 0.988799989
Epoch:  25   =====> Loss= 0.004524751  ====> Testing accuracy = 0.991100013  ====> Validatio
n accuracy = 0.988799989
Epoch:  26   =====> Loss= 0.004561770  ====> Testing accuracy = 0.987699986  ====> Validatio
n accuracy = 0.986000001
Epoch:  27   =====> Loss= 0.010814430  ====> Testing accuracy = 0.987800002  ====> Validatio
n accuracy = 0.988200009
Epoch:  28   =====> Loss= 0.003521892  ====> Testing accuracy = 0.990499973  ====> Validatio
n accuracy = 0.990999997
Epoch:  29   =====> Loss= 0.004938369  ====> Testing accuracy = 0.990100026  ====> Validatio
```

```
n accuracy = 0.991999984
Epoch:  30   =====> Loss= 0.004605746  ====> Testing accuracy = 0.989700019  ====> Validatio
n accuracy = 0.988799989
Epoch:  31   =====> Loss= 0.007255645  ====> Testing accuracy = 0.988900006  ====> Validatio
n accuracy = 0.989600003
Epoch:  32   =====> Loss= 0.004653554  ====> Testing accuracy = 0.988499999  ====> Validatio
n accuracy = 0.989000022
Epoch:  33   =====> Loss= 0.005368224  ====> Testing accuracy = 0.990599990  ====> Validatio
n accuracy = 0.988799989
Epoch:  34   =====> Loss= 0.002434701  ====> Testing accuracy = 0.991599977  ====> Validatio
n accuracy = 0.990400016
Epoch:  35   =====> Loss= 0.004841189  ====> Testing accuracy = 0.988799989  ====> Validatio
n accuracy = 0.987200022
Epoch:  36   =====> Loss= 0.005518572  ====> Testing accuracy = 0.991299987  ====> Validatio
n accuracy = 0.986999989
Epoch:  37   =====> Loss= 0.005487071  ====> Testing accuracy = 0.990800023  ====> Validatio
n accuracy = 0.987999976
Epoch:  38   =====> Loss= 0.002733897  ====> Testing accuracy = 0.987500012  ====> Validatio
n accuracy = 0.986599982
Epoch:  39   =====> Loss= 0.004490448  ====> Testing accuracy = 0.988900006  ====> Validatio
n accuracy = 0.990199983
Epoch:  40   =====> Loss= 0.005449366  ====> Testing accuracy = 0.991400003  ====> Validatio
n accuracy = 0.991400003
Training time : 709.6574301719666
Model saved in path: log_files/model_Adam/Adam
Final testing accuracy: 0.9914
```

Question 2.2.2 Try to add dropout (keep_prob = 0.75) before the first fully connected layer. You will use tf.nn.dropout for that purpose. What accuracy do you achieve on testing data?

Accuracy achieved on testing data: With the Adam optimizer (best of the two tested), we obtain an accuracy of 0.9877 on testing data. The dropout did not improve the model, the accuracy is slightly worse (0.9877 vs 0.9914). Even the training time is similar to the one without dropout. The training does not converge faster as we could expected, but as it already converged really fast without dropout it would be difficult to realize that in this model.

In [ ]:

```
def LeNet5_Model_Dropout(image):
    keep_prob = 0.75

    #Layer 1 : Convolution 5*5
    weight1 = weight_variable([5, 5, 1, 6]) # shape [filter_height, filter_width, in_channe
ls, out_channels]
    bias1 = bias_variable([6]) # shape (out_channels)
    conv1 = tf.nn.conv2d(image, weight1, strides=[1,1,1,1], padding='SAME')
    act1 = tf.nn.relu(conv1 + bias1)
    pool1 = tf.nn.max_pool(act1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')

    #Layer 2 : Convolution 5*5
    weight2 = weight_variable([5, 5, 6, 16]) # shape [filter_height, filter_width, in_chann
els, out_channels]
    bias2 = bias_variable([16]) # shape (depth_image_out)
    conv2 = tf.nn.conv2d(pool1, weight2, strides=[1,1,1,1], padding='VALID')
    act2 = tf.nn.relu(conv2 + bias2)
    pool2 = tf.nn.max_pool(act2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')

    #Flatten layer
    flatten = tf.reshape(pool2, [-1, 5*5*16])
    drop = tf.nn.dropout(flatten, keep_prob)

    #Layer 3 : Fully Connected
    weight3 = weight_variable([5*5*16, 120])
    bias3 = bias_variable([120])
    act3 = tf.nn.relu(tf.matmul(drop, weight3) + bias3)
```

```
    #Layer 4 : Fully Connected
    weight4 = weight_variable([120, 84])
    bias4 = bias_variable([84])
    act4 = tf.nn.relu(tf.matmul(act3, weight4) + bias4)

    #Layer 5 : Fully Connected
    weight5 = weight_variable([84, 10])
    bias5 = bias_variable([10])
    act5 = tf.nn.softmax(tf.matmul(act4, weight5) + bias5)

    return act5
```

In [ ]:

```
tf.reset_default_graph()

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

#Model
with tf.name_scope('Model'):
    pred = LeNet5_Model_Dropout(x)
#Loss function
with tf.name_scope('Loss'):
    cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(tf.clip_by_value(pred, epsilon, 1.0)), re
duction_indices=1))
#Optimizer
with tf.name_scope('Adam'):
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
#Accuracy
with tf.name_scope('Accuracy'):
    acc = evaluate(pred, y)

# Initializing the variables
init = tf.global_variables_initializer()
# Create a summary to monitor cost tensor
tf.summary.scalar("Loss_LeNet-5_Adam", cost)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("Accuracy_LeNet-5_Adam", acc)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()
```

In [ ]:

```
with tf.Session() as sess:
    train(init, sess, logs_path, training_epochs, batch_size, optimizer, cost, merged_summa
ry_op)
```

```
Epoch:  01   =====> Loss= 0.374580437  ====> Testing accuracy = 0.962800026  ====> Validatio
n accuracy = 0.958400011
Epoch:  02   =====> Loss= 0.105710727  ====> Testing accuracy = 0.976199985  ====> Validatio
n accuracy = 0.977400005
Epoch:  03   =====> Loss= 0.078360710  ====> Testing accuracy = 0.980300009  ====> Validatio
n accuracy = 0.980599999
Epoch:  04   =====> Loss= 0.062998356  ====> Testing accuracy = 0.981500030  ====> Validatio
n accuracy = 0.981400013
Epoch:  05   =====> Loss= 0.053353163  ====> Testing accuracy = 0.984899998  ====> Validatio
n accuracy = 0.984399974
Epoch:  06   =====> Loss= 0.048859623  ====> Testing accuracy = 0.984399974  ====> Validatio
n accuracy = 0.980599999
Epoch:  07   =====> Loss= 0.043629460  ====> Testing accuracy = 0.986199975  ====> Validatio
n accuracy = 0.986400008
Epoch:  08   =====> Loss= 0.040206590  ====> Testing accuracy = 0.984600008  ====> Validatio
n accuracy = 0.981199980
Epoch:  09   =====> Loss= 0.035232335  ====> Testing accuracy = 0.986400008  ====> Validatio
n accuracy = 0.985400021
Epoch:  10   =====> Loss= 0.031702766  ====> Testing accuracy = 0.985800028  ====> Validatio
n accuracy = 0.986599982
```

```
n accuracy = 0.986599982
Epoch:  11   =====> Loss= 0.032478882  ====> Testing accuracy = 0.984899998  ====> Validatio
n accuracy = 0.982800007
Epoch:  12   =====> Loss= 0.028045211  ====> Testing accuracy = 0.987100005  ====> Validatio
n accuracy = 0.986199975
Epoch:  13   =====> Loss= 0.025580297  ====> Testing accuracy = 0.988200009  ====> Validatio
n accuracy = 0.987600029
Epoch:  14   =====> Loss= 0.025265909  ====> Testing accuracy = 0.989000022  ====> Validatio
n accuracy = 0.990199983
Epoch:  15   =====> Loss= 0.025295069  ====> Testing accuracy = 0.988099992  ====> Validatio
n accuracy = 0.986000001
Epoch:  16   =====> Loss= 0.021733447  ====> Testing accuracy = 0.986699998  ====> Validatio
n accuracy = 0.986400008
Epoch:  17   =====> Loss= 0.021563134  ====> Testing accuracy = 0.985800028  ====> Validatio
n accuracy = 0.987200022
Epoch:  18   =====> Loss= 0.020839919  ====> Testing accuracy = 0.987500012  ====> Validatio
n accuracy = 0.987800002
Epoch:  19   =====> Loss= 0.018534759  ====> Testing accuracy = 0.988799989  ====> Validatio
n accuracy = 0.989000022
Epoch:  20   =====> Loss= 0.017443275  ====> Testing accuracy = 0.985800028  ====> Validatio
n accuracy = 0.987800002
Epoch:  21   =====> Loss= 0.018232929  ====> Testing accuracy = 0.988799989  ====> Validatio
n accuracy = 0.988200009
Epoch:  22   =====> Loss= 0.015170505  ====> Testing accuracy = 0.986699998  ====> Validatio
n accuracy = 0.987800002
Epoch:  23   =====> Loss= 0.017768484  ====> Testing accuracy = 0.988399982  ====> Validatio
n accuracy = 0.989400029
Epoch:  24   =====> Loss= 0.016556849  ====> Testing accuracy = 0.987900019  ====> Validatio
n accuracy = 0.988399982
Epoch:  25   =====> Loss= 0.015419104  ====> Testing accuracy = 0.987900019  ====> Validatio
n accuracy = 0.988799989
Epoch:  26   =====> Loss= 0.014873846  ====> Testing accuracy = 0.988799989  ====> Validatio
n accuracy = 0.988399982
Epoch:  27   =====> Loss= 0.014391605  ====> Testing accuracy = 0.987900019  ====> Validatio
n accuracy = 0.988399982
Epoch:  28   =====> Loss= 0.013937918  ====> Testing accuracy = 0.987200022  ====> Validatio
n accuracy = 0.988799989
Epoch:  29   =====> Loss= 0.013180870  ====> Testing accuracy = 0.988600016  ====> Validatio
n accuracy = 0.988799989
Epoch:  30   =====> Loss= 0.012467145  ====> Testing accuracy = 0.986599982  ====> Validatio
n accuracy = 0.989000022
Epoch:  31   =====> Loss= 0.012077014  ====> Testing accuracy = 0.987999976  ====> Validatio
n accuracy = 0.989400029
Epoch:  32   =====> Loss= 0.012803988  ====> Testing accuracy = 0.988200009  ====> Validatio
n accuracy = 0.987999976
Epoch:  33   =====> Loss= 0.011257896  ====> Testing accuracy = 0.987500012  ====> Validatio
n accuracy = 0.988399982
Epoch:  34   =====> Loss= 0.012967548  ====> Testing accuracy = 0.988099992  ====> Validatio
n accuracy = 0.990199983
Epoch:  35   =====> Loss= 0.011118847  ====> Testing accuracy = 0.987399995  ====> Validatio
n accuracy = 0.988600016
Epoch:  36   =====> Loss= 0.010818945  ====> Testing accuracy = 0.987699986  ====> Validatio
n accuracy = 0.990999997
Epoch:  37   =====> Loss= 0.010403670  ====> Testing accuracy = 0.987900019  ====> Validatio
n accuracy = 0.989799976
Epoch:  38   =====> Loss= 0.009915944  ====> Testing accuracy = 0.989000022  ====> Validatio
n accuracy = 0.989000022
Epoch:  39   =====> Loss= 0.010704457  ====> Testing accuracy = 0.988300025  ====> Validatio
n accuracy = 0.989199996
Epoch:  40   =====> Loss= 0.010647823  ====> Testing accuracy = 0.987200022  ====> Validatio
n accuracy = 0.987600029
Training time : 715.9265096187592
Model saved in path: log_files/model_Adam/Adam
Final testing accuracy: 0.9877
```

All in all, CNN allows us to reach an accuracy of 99% (0.9914 with Adam optimizer and without dropout in 40 epochs) where MLP gave us an accuracy of about 97%.