Hello, TensorFlow!

In [1]:
```python
import tensorflow as tf
```

In [2]:
```python
# Simple hello world using TensorFlow

# Create a Constant op
# The op is added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.

hello = tf.constant('Hello, TensorFlow!')
```

In [3]:
```python
# Start tf session
sess = tf.Session()
```

In [4]:
```python
# Run graph
print(sess.run(hello))
```

```
b'Hello, TensorFlow!'
```

Basic Operations in TensorFlow

In [1]:
```
import tensorflow as tf
```

In [2]:
```
tf.__version__
```

Out[2]: '1.4.0'

TensorFlow

- Tensor는 동적 크기를 갖는 다차원 데이터 배열
- 수치 연산을 기호로 표현한 그래프 구조를 만들어서 처리
- 그래프의 노드는 수학 연산, 노드를 연결하는 에지는 다차원 배열을 의미
- CPU, GPU, mobile Platform(Android, iOS)등 다양한 플랫폼에서 사용 가능

Constant

In [3]:
```
a = tf.constant(2)
b = tf.constant(3)
```

In [4]:
```
# Launch the default graph.
with tf.Session() as sess:
    print("a=2, b=3")
    print("Addition with constants: %i" % sess.run(a+b))
    print("Multiplication with constants: %i" % sess.run(a*b))
```

```
a=2, b=3
Addition with constants: 5
Multiplication with constants: 6
```

Variable

In [5]:
```
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)
```

In [7]:
```
# Define some operations
add = tf.add(a, b)
mul = tf.multiply(a, b)
```

In [9]:
```
# Launch the default graph.
with tf.Session() as sess:
    # Run every operation with variable input
    print("Addition with variables: %i" %
                  sess.run(add, feed_dict={a: 2, b: 3}))
    print("Multiplication with variables: %i" %
                  sess.run(mul, feed_dict={a: 2, b: 3}))
```

```
Addition with variables: 5
Multiplication with variables: 6
```

In [10]:
```
# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
matrix1 = tf.constant([[3., 3.]])
```

In [11]:
```
# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])
```

In [12]:
```
# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
product = tf.matmul(matrix1, matrix2)
```

In [13]:
```
# The output of the op is returned in 'result' as a numpy `ndarray` ob

with tf.Session() as sess:
    result = sess.run(product)
    print(result)
```

```
[[ 12.]]
```

TensorFlow 주요 연산

연산 카테고리	연산자
math	add, sub, mul, div, exp, log, greater, less, equal
array	concat, slice, split, constant, rank, shape, shuffle
matrix	matmul, matrixInverse, matrixDeterminant
neural network	softMax, sigmoid, ReLU, convolution2D, maxPool
session	save, restore
queue	enqueue, dequeue, mutexAcquire, mutexRelease
flow control	merge, switch, enter, leave, nextIteration

제공되는 수학 함수에 대해 보다 자세한 내용은 다음의 URL을 참조하세요.

- https://www.tensorflow.org/api_guides/python/math_ops (https://www.tensorflow.org/api_guides/python/math_ops)

In [ ]:

MNIST Dataset Introduction

대부분의 예제는 손으로 쓴 숫자의 MNIST 데이터 세트를 사용합니다. 이 데이터 세트에는 교육을 위한 60,000 개의 예제와 테스트를 위한 10,000 개의 예제가 포함되어 있습니다. 숫자는 0에서 1까지의 값을 갖는 고정 크기 이미지 (28x28 픽셀)로 크기 정규화되고 중심에 배치되었습니다. 간단히하기 위해 각 이미지는 평평하게 784 피처의 1-D numpy 배열로 변환됩니다.

Overview

Usage

예제에서는 TensorFlow를 사용하고 있습니다. input_data.py (https://github.com/tensorflow/tensorflow/blob/r0.7/tensorflow/examples/tutorials/mnist/input_
스크립트를 사용하여 해당 데이터 세트를 로드하십시오. 데이터를 관리하고 처리하는 데 매우 유용합니다.

- Dataset downloading
- Loading the entire dataset into numpy array:

In [1]:
```
# Import MNIST
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Load data
X_train = mnist.train.images
Y_train = mnist.train.labels
X_test = mnist.test.images
Y_test = mnist.test.labels
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

- 전체 데이터 집합을 반복하고 데이터 집합 샘플의 원하는 부분만 반환하는 `next_batch` 함수입니다. (메모리를 절약하고 전체 데이터 세트를 로드하지 않습니다.)

In [2]:
```
# Get the next 64 images array and labels
batch_X, batch_Y = mnist.train.next_batch(64)
```

In [4]:
```
batch_X.shape, batch_Y.shape
```

Out[4]:
```
((64, 784), (64, 10))
```

In [7]:
```
batch_Y[0]
```

Out[7]:
```
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.])
```

Link: [http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)](http://yann.lecun.com/exdb/mnist/)

Linear Regression in TensorFlow

Simple Linear Regression

$$y = W \times x + b$$

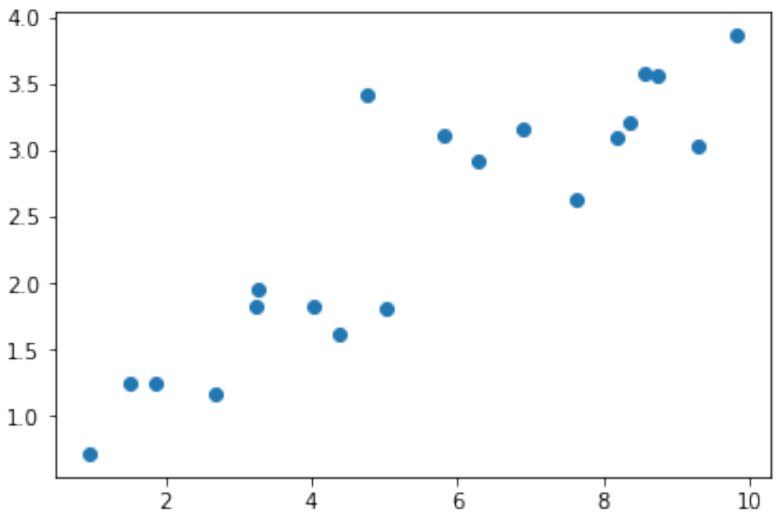
In [1]:
```
%matplotlib inline
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# Parameters
learning_rate = 0.01
training_epochs = 1000
display_step = 50
```

In [3]:
```
# Training Data
train_X = np.linspace(1, 10, 20) + np.random.randn(20) * 0.5
train_Y = np.linspace(1, 4, 20) + np.random.randn(20) * 0.5

n_samples = train_X.shape[0]
```

In [4]:
```
plt.scatter(train_X, train_Y)
```

Out[4]: <matplotlib.collections.PathCollection at 0x1251e7a20>



In [5]:
```
# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")
```

In [6]:
```
# Create Model

# Set model weights
W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
```

In [7]:
```
# Construct a linear model
activation = tf.add(tf.multiply(X, W), b)
```

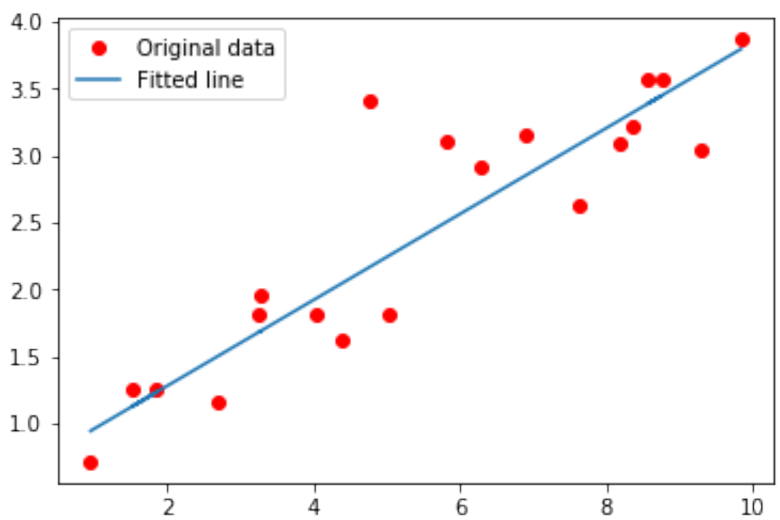
In [8]:
```
# Mean the squared errors
cost = tf.reduce_sum(tf.pow(activation-Y, 2))/(2*n_samples) #L2 loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(
```

In [9]:
```
# Initializing the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

In [10]:

```
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})

        #Display logs per epoch step
        if (epoch+1) % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(sess.run(cost, feed_dict={X: train_X,
                                                          Y:train_Y})),
                "W=", sess.run(W), "b=", sess.run(b))

    print("Optimization Finished!")
    print("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}),
          "W=", sess.run(W), "b=", sess.run(b))

    #Graphic display
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b),
             label='Fitted line')
    plt.legend()
    plt.show()
```

```
Epoch: 0100 cost= 0.102463506 W= 0.372069 b= 0.27407
Epoch: 0200 cost= 0.096549489 W= 0.36137 b= 0.3504
Epoch: 0300 cost= 0.092502788 W= 0.35251 b= 0.413608
Epoch: 0400 cost= 0.089735113 W= 0.345174 b= 0.465948
Epoch: 0500 cost= 0.087843239 W= 0.339098 b= 0.509289
Epoch: 0600 cost= 0.086550839 W= 0.334068 b= 0.545179
Epoch: 0700 cost= 0.085668668 W= 0.329902 b= 0.5749
Epoch: 0800 cost= 0.085067153 W= 0.326452 b= 0.599509
Epoch: 0900 cost= 0.084657431 W= 0.323596 b= 0.619889
Epoch: 1000 cost= 0.084378794 W= 0.32123 b= 0.636765
Optimization Finished!
cost= 0.0843788 W= 0.32123 b= 0.636765
```



Logistic Regression in TensorFlow

In [1]:
```
import tensorflow as tf
```

In [2]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

In [3]:
```
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1
```

In [4]:
```
# tf Graph Input
x = tf.placeholder("float", [None, 784]) # mnist data image of shape 2
y = tf.placeholder("float", [None, 10]) # 0-9 digits recognition => 10
```

In [5]:
```
# Create model

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

In [6]:
```
# Construct model
pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax
```

In [7]:
```
# Minimize error using cross entropy

# Cross entropy
cost = tf.reduce_mean( -tf.reduce_sum(y*tf.log(pred), reduction_indice

# Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(
```

In [8]:
```
# Initializing the variables
init = tf.global_variables_initializer()
```

In [9]:

```python
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # 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)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batcl
        # Display logs per epoch step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".form

    print("Optimization Finished!")

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print("Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.t
```

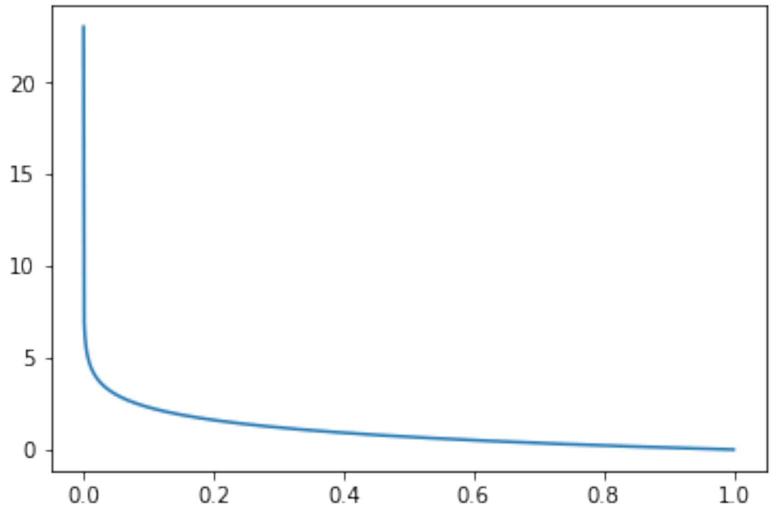
```
Epoch: 0001 cost= 1.175765760
Epoch: 0002 cost= 0.662211569
Epoch: 0003 cost= 0.550528342
Epoch: 0004 cost= 0.496682484
Epoch: 0005 cost= 0.463698466
Epoch: 0006 cost= 0.440892180
Epoch: 0007 cost= 0.423977649
Epoch: 0008 cost= 0.410613063
Epoch: 0009 cost= 0.399908744
Epoch: 0010 cost= 0.390926166
Epoch: 0011 cost= 0.383335643
Epoch: 0012 cost= 0.376794658
Epoch: 0013 cost= 0.371020695
Epoch: 0014 cost= 0.365888950
Epoch: 0015 cost= 0.361368662
Epoch: 0016 cost= 0.357248354
Epoch: 0017 cost= 0.353573571
Epoch: 0018 cost= 0.350168565
Epoch: 0019 cost= 0.347024083
Epoch: 0020 cost= 0.344156632
Epoch: 0021 cost= 0.341527212
Epoch: 0022 cost= 0.338988009
Epoch: 0023 cost= 0.336679159
Epoch: 0024 cost= 0.334483589
Epoch: 0025 cost= 0.332441212
Optimization Finished!
Accuracy: 0.9148
```

Cross entropy Cost

In [10]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

In [11]:
```
x = np.linspace(0, 1, 1000) + 1e-10
```

In [12]:
```
plt.plot(x, -np.log(x));
```



$$-\sum_i L_i \log(\hat{y_i}) => \sum_i L_i \times -\log(\hat{y_i})$$

$$L = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\hat{Y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot -\log \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} \infty \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0, \quad cost = 0$$

$$\hat{Y} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot -\log \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ \infty \end{bmatrix} = \begin{bmatrix} 0 \\ \infty \end{bmatrix} = \infty, \quad cost = \infty$$

In [ ]:
```
```

Nearest Neighbor in TensorFlow

In [1]:
```
import numpy as np
import tensorflow as tf
```

In [2]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
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
```

In [5]:
```
# In this example, we limit mnist data
Xtr, Ytr = mnist.train.next_batch(5000) #5000 for training (nn candida
Xte, Yte = mnist.test.next_batch(200)   #200 for testing
```

In [9]:
```
Xtr.shape
```

Out[9]: (5000, 784)

In [6]:
```
# tf Graph Input
xtr = tf.placeholder("float", [None, 784])
xte = tf.placeholder("float", [784])

# Nearest Neighbor calculation using L1 Distance
# Calculate L1 Distance
distance = tf.reduce_sum(tf.abs(tf.add(xtr, tf.negative(xte))),
                         reduction_indices=1)
# Predict: Get min distance index (Nearest neighbor)
pred = tf.argmin(distance, 0)

accuracy = 0.

# Initializing the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

In [7]:
```
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # loop over test data
    for i in range(len(Xte)):
        # Get nearest neighbor
        nn_index = sess.run(pred, feed_dict={xtr: Xtr, xte: Xte[i,:]})

        # Get nearest neighbor class label and compare it to its true
        if i % 10 == 0:
            print("Test", i, "Prediction:", np.argmax(Ytr[nn_index]),
                  "True Class:", np.argmax(Yte[i]))

            #print("\nnn_index :", nn_index)
            #print("Ytr[nn_index] :", Ytr[nn_index])
            #print("np.argmax(Ytr[nn_index]) :", np.argmax(Ytr[nn_inde

        # Calculate accuracy
        if np.argmax(Ytr[nn_index]) == np.argmax(Yte[i]):
            accuracy += 1./len(Xte)

    print("Done!")
    print("Accuracy:", round(accuracy,2))
```

```
Test 0 Prediction: 5 True Class: 5
Test 10 Prediction: 9 True Class: 4
Test 20 Prediction: 1 True Class: 1
Test 30 Prediction: 6 True Class: 6
Test 40 Prediction: 3 True Class: 2
Test 50 Prediction: 5 True Class: 8
Test 60 Prediction: 5 True Class: 5
Test 70 Prediction: 9 True Class: 9
Test 80 Prediction: 5 True Class: 5
Test 90 Prediction: 9 True Class: 9
Test 100 Prediction: 2 True Class: 2
Test 110 Prediction: 9 True Class: 9
Test 120 Prediction: 0 True Class: 0
Test 130 Prediction: 9 True Class: 9
Test 140 Prediction: 0 True Class: 0
Test 150 Prediction: 3 True Class: 3
Test 160 Prediction: 2 True Class: 2
Test 170 Prediction: 5 True Class: 5
Test 180 Prediction: 3 True Class: 3
Test 190 Prediction: 1 True Class: 1
Done!
Accuracy: 0.89
```

```
Test 0 Prediction: 7 True Class: 7

nn_index : 4444
Ytr[nn_index] : [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
np.argmax(Ytr[nn_index]) : 7

Test 1 Prediction: 2 True Class: 2

nn_index : 816
Ytr[nn_index] : [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
np.argmax(Ytr[nn_index]) : 2

...
```

K-Means Example

Implement K-Means algorithm with TensorFlow, and apply it to classify handwritten digit images.

In [6]:
```
import numpy as np
import tensorflow as tf
from tensorflow.contrib.factorization import KMeans
```

In [2]:
```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
full_data_x = mnist.train.images
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

In [12]:
```
# Parameters
num_steps = 50     # Total steps to train
batch_size = 1024  # The number of samples per batch
k = 25             # The number of clusters
num_classes = 10   # The 10 digits
num_features = 784 # Each image is 28x28 pixels

# Input images
X = tf.placeholder(tf.float32, shape=[None, num_features])
# Labels (for assigning a label to a centroid and testing)
Y = tf.placeholder(tf.float32, shape=[None, num_classes])

# K-Means Parameters
kmeans = KMeans(inputs=X, num_clusters=k, distance_metric='cosine',
                use_mini_batch=True)
```

In [13]:
```
# Build KMeans graph
(all_scores, cluster_idx, scores, cluster_centers_initialized,\
 cluster_centers_var, init_op, train_op) = kmeans.training_graph()

cluster_idx = cluster_idx[0] # fix for cluster_idx being a tuple
avg_distance = tf.reduce_mean(scores)

# Initialize the variables (i.e. assign their default value)
init_vars = tf.global_variables_initializer()
```

In [10]:

```
# Start TensorFlow session
sess = tf.Session()

# Run the initializer
sess.run(init_vars, feed_dict={X: full_data_x}) # init tf
sess.run(init_op, feed_dict={X: full_data_x})   # init kmeans

# Training
for i in range(1, num_steps + 1):
    _, d, idx = sess.run([train_op, avg_distance, cluster_idx],
                         feed_dict={X: full_data_x})
    if i % 10 == 0 or i == 1:
        print("Step %i, Avg Distance: %f" % (i, d))
```

```
Step 1, Avg Distance: 0.341471
Step 10, Avg Distance: 0.221609
Step 20, Avg Distance: 0.220328
Step 30, Avg Distance: 0.219776
Step 40, Avg Distance: 0.219419
Step 50, Avg Distance: 0.219154
Step 60, Avg Distance: 0.218940
Step 70, Avg Distance: 0.218764
Step 80, Avg Distance: 0.218614
Step 90, Avg Distance: 0.218484
Step 100, Avg Distance: 0.218373
```

In [11]:

```
# Assign a label to each centroid
# Count total number of labels per centroid,
# using the label of each training
# sample to their closest centroid (given by 'idx')
counts = np.zeros(shape=(k, num_classes))

for i in range(len(idx)):
    counts[idx[i]] += mnist.train.labels[i]

# Assign the most frequent label to the centroid
labels_map = [np.argmax(c) for c in counts]
labels_map = tf.convert_to_tensor(labels_map)

# Evaluation ops
# Lookup: centroid_id -> label
cluster_label = tf.nn.embedding_lookup(labels_map, cluster_idx)

# Compute accuracy
correct_prediction = tf.equal(cluster_label,
                              tf.cast(tf.argmax(Y, 1), tf.int32))
accuracy_op = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Test Model
test_x, test_y = mnist.test.images, mnist.test.labels
print("Test Accuracy:", sess.run(accuracy_op,
                                 feed_dict={X: test_x, Y: test_y}))
```

```
Test Accuracy: 0.7273
```

In [ ]:

```

```

tf.nn.embedding_lookup

참고자료 : [https://code.i-harness.com/ko/q/2141556 (https://code.i-harness.com/ko/q/2141556)](https://code.i-harness.com/ko/q/2141556)

embedding_lookup 함수는 params 텐서의 행을 검색합니다. 이 동작은 배열에 numpy로 인덱스를 사용하는 것과 비슷합니다. 예 :

In [42]:

```
matrix = np.eye(4)  # 64-dimensional embeddings
ids = np.array([0, 2])
print( matrix[ids] )  # prints a matrix of shape [4, 64]
```

```
[[ 1.  0.  0.  0.]
 [ 0.  0.  1.  0.]]
```

다음은 간단하게 텐서플로우로 작성한 것입니다.

In [34]:

```
params = tf.constant([10, 20, 30, 40])
ids = tf.constant([0,1,2,1])
print( tf.nn.embedding_lookup(params, ids).eval(session=sess) )
```

```
[10 20 30 20]
```

In [ ]:

```

```

Multilayer Perceptron in TensorFlow

In [1]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
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
```

In [2]:
```
import tensorflow as tf
```

In [3]:
```
# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1
```

In [4]:
```
# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 256 # 2nd layer num features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
```

In [5]:
```
# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])
```

In [6]:
```
# Create model
def multilayer_perceptron(_X, _weights, _biases):
    #Hidden layer with RELU activation
    layer_1 = tf.nn.relu(tf.add(tf.matmul(_X, _weights['h1']), _biases
    #Hidden layer with RELU activation
    layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, _weights['h2']), _b
    return tf.matmul(layer_2, _weights['out']) + _biases['out']
```

In [7]:
```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

In [8]:
```
# Construct model
pred = multilayer_perceptron(x, weights, biases)
```

In [9]:
```
# Define loss and optimizer
# Softmax loss
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y
                                                               logits=p

# Adam Optimizer
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimi
```

In [10]:
```
# Initializing the variables
init = tf.global_variables_initializer()
```

In [11]:
```
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # 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)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batcl
        # Display logs per epoch step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".form

    print("Optimization Finished!")

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print("Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.t
```

```
Epoch: 0001 cost= 162.539960483
Epoch: 0002 cost= 38.502977509
Epoch: 0003 cost= 23.416618775
Epoch: 0004 cost= 15.683095744
Epoch: 0005 cost= 11.245915335
Epoch: 0006 cost= 8.056293164
Epoch: 0007 cost= 5.871352453
Epoch: 0008 cost= 4.246478563
Epoch: 0009 cost= 3.219774899
Epoch: 0010 cost= 2.251065161
Epoch: 0011 cost= 1.617439716
Epoch: 0012 cost= 1.073335351
Epoch: 0013 cost= 0.834915558
Epoch: 0014 cost= 0.535639277
Epoch: 0015 cost= 0.411491749
Optimization Finished!
Accuracy: 0.9474
```

In [ ]:

AlexNet in TensorFlow

In [1]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
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
```

In [2]:
```
import tensorflow as tf
```

In [3]:
```
# Parameters
learning_rate = 0.001

#training_iters = 300000
training_iters = 50000
#batch_size = 64
batch_size = 128

display_step = 100
```

In [4]:
```
# Network Parameters
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
dropout = 0.8 # Dropout, probability to keep units
```

In [5]:
```
# tf Graph input
x = tf.placeholder(tf.float32, [None, n_input])
y = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32) # dropout (keep probability)
```

In [6]:
```
# Create AlexNet model
def conv2d(name, l_input, w, b):
    return tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(l_input, w,
                                                  strides=[1, 1, 1, 1]
                                                  padding='SAME'),b),
                                                  name=name)

def max_pool(name, l_input, k):
    return tf.nn.max_pool(l_input, ksize=[1, k, k, 1], strides=[1, k, 
                          padding='SAME', name=name)

def norm(name, l_input, lsize=4):
    return tf.nn.lrn(l_input, lsize, bias=1.0, alpha=0.001 / 9.0,
                     beta=0.75, name=name)

def alex_net(_X, _weights, _biases, _dropout):
    # Reshape input picture
    _X = tf.reshape(_X, shape=[-1, 28, 28, 1])

    # Convolution Layer
```

```
    # Convolution Layer
    conv1 = conv2d('conv1', _X, _weights['wc1'], _biases['bc1'])
    # Max Pooling (down-sampling)
    pool1 = max_pool('pool1', conv1, k=2)
    # Apply Normalization
    norm1 = norm('norm1', pool1, lsize=4)
    # Apply Dropout
    norm1 = tf.nn.dropout(norm1, _dropout)

    # Convolution Layer
    conv2 = conv2d('conv2', norm1, _weights['wc2'], _biases['bc2'])
    # Max Pooling (down-sampling)
    pool2 = max_pool('pool2', conv2, k=2)
    # Apply Normalization
    norm2 = norm('norm2', pool2, lsize=4)
    # Apply Dropout
    norm2 = tf.nn.dropout(norm2, _dropout)

    # Convolution Layer
    conv3 = conv2d('conv3', norm2, _weights['wc3'], _biases['bc3'])
    # Max Pooling (down-sampling)
    pool3 = max_pool('pool3', conv3, k=2)
    # Apply Normalization
    norm3 = norm('norm3', pool3, lsize=4)
    # Apply Dropout
    norm3 = tf.nn.dropout(norm3, _dropout)

    # Fully connected layer
    # Reshape conv3 output to fit dense layer input
    dense1 = tf.reshape(norm3, [-1, _weights['wd1'].get_shape().as_lis
    # Relu activation
    dense1 = tf.nn.relu(tf.matmul(dense1, _weights['wd1']) + _biases['

    # Relu activation
    dense2 = tf.nn.relu(tf.matmul(dense1, _weights['wd2']) + _biases['

    # Output, class prediction
    out = tf.matmul(dense2, _weights['out']) + _biases['out']
    return out
```

In [7]:
```
# Store layers weight & bias
weights = {
    'wc1': tf.Variable(tf.random_normal([3, 3, 1, 64])),
    'wc2': tf.Variable(tf.random_normal([3, 3, 64, 128])),
    'wc3': tf.Variable(tf.random_normal([3, 3, 128, 256])),
    'wd1': tf.Variable(tf.random_normal([4*4*256, 1024])),
    'wd2': tf.Variable(tf.random_normal([1024, 1024])),
    'out': tf.Variable(tf.random_normal([1024, 10]))
}
biases = {
    'bc1': tf.Variable(tf.random_normal([64])),
    'bc2': tf.Variable(tf.random_normal([128])),
    'bc3': tf.Variable(tf.random_normal([256])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'bd2': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

In [8]:
```
# Construct model
pred = alex_net(x, weights, biases, keep_prob)
```

In [9]:
```
# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimi
```

In [10]:
```
# Evaluate model
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

In [11]:
```
# Initializing the variables
init = tf.global_variables_initializer()
```

In [12]:
```
import time
```

In [13]:

```
# Launch the graph

start = time.time()
with tf.Session() as sess:
    sess.run(init)
    step = 1
    # Keep training until reach max iterations
    while step * batch_size < training_iters:
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        # Fit training using batch data
        sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys, keep_
        if step % display_step == 0:
            # Calculate batch accuracy
            acc = sess.run(accuracy, feed_dict={x: batch_xs,
                                                y: batch_ys, keep_prob
            # Calculate batch loss
            loss = sess.run(cost, feed_dict={x: batch_xs,
                                             y: batch_ys, keep_prob: 1
            print("Iter " + str(step*batch_size) + ", Minibatch Loss=
                  + "{:.6f}".format(loss) + ", Training Accuracy= " +
            print("Elapsed Time : {:.3f}".format(time.time()-start))

        step += 1
    print("Optimization Finished!")
    # Calculate accuracy for 256 mnist test images
    print("Testing Accuracy:", sess.run(accuracy, feed_dict={x: mnist.
                                                              y: mnist.
                                                              keep_prob
```

```
Iter 12800, Minibatch Loss= 34608.140625, Training Accuracy= 0.54688
Elapsed Time : 103.070
Iter 25600, Minibatch Loss= 13450.774414, Training Accuracy= 0.78125
Elapsed Time : 203.184
Iter 38400, Minibatch Loss= 8710.173828, Training Accuracy= 0.77344
Elapsed Time : 315.355
Optimization Finished!
Testing Accuracy: 0.792969
```

In [ ]: