TensorFlow Tutorial

본 문서는 TensorFlow 를 사용하여 기초적인 Deep Learning을 구현하고 실험하기 위한 실습 자료이다. 첫 번째 파트에서는 TensorFlow에 대한 기본 원리 및 deep learning 구현 예제를 다루어보고, 두 번째 파트에서는 오픈소스를 활용한 Deep Reinforcement Learning 을 실습해보는 시간을 갖는다. 마지막으로는 TensorFlow로 구현되고 공개된 여러 오픈소스를 둘러본다.

The code and comments are written by Dong-Hyun Kwak (imcomking@gmail.com)

Part 1. TensorFlow

TensorFlow 이하, tf는 구글 주도하에 초기 개발되어, 2015년 12월 부터 오픈소스로 공개되어 널리 쓰이고 있는 기계학습(딥러닝)을 위한 라이브러리이다. Computation Graph를 사용한 Theano의 장점을 그대로 살려 automatic gradient 계산이 가능하고, Multi-GPU 환경에서도 동작가능한 아키텍처 기반으로 설계되었다.

TensorFlow는 CPU와 GPU, Multi-GPU 환경을 모두 지원하며, 함수의 적절한 추상화 수준, 직관적이고 쉬운 문법, 빠른 업데이트 속도, 구글의 강력한 지원, 넓은 사용자 층 등이 TensorFlow가 가진 큰 장점이다.

관련 사이트

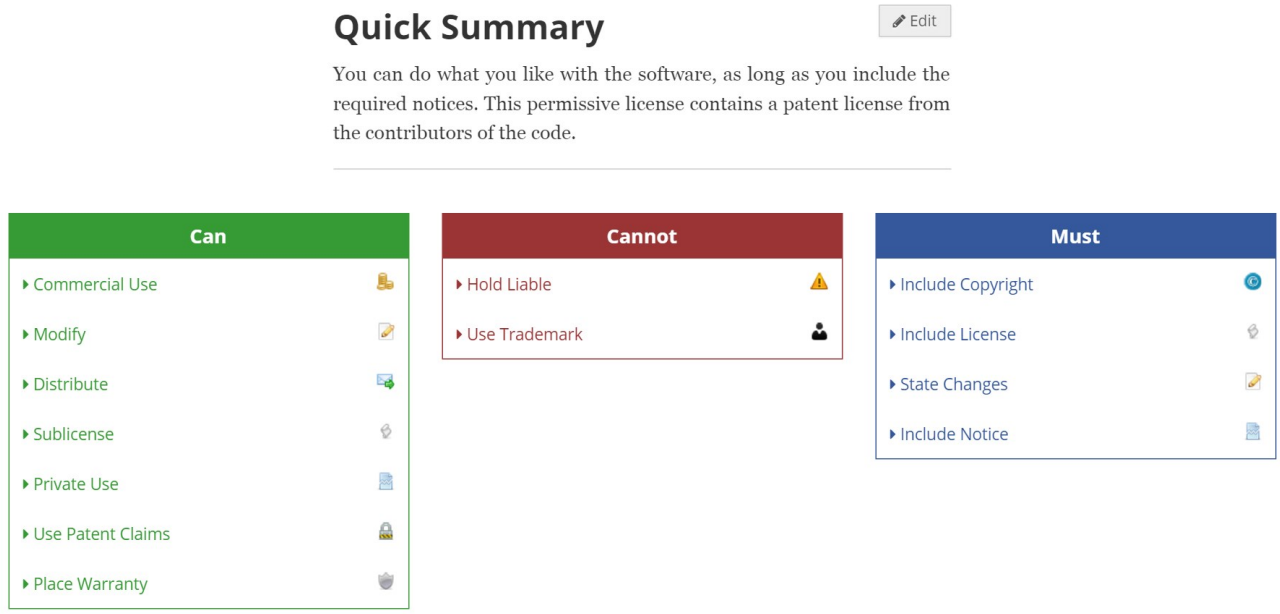
TensorFlow 공식 홈페이지 : http://tensorflow.org/
TensorFlow Github : https://github.com/tensorflow/tensorflow/releases
TensorFlow Contributors : https://github.com/tensorflow/tensorflow/graphs/contributors
TensorFlow Playground : http://playground.tensorflow.org/
딥러닝 Docker Image : https://hub.docker.com/r/imcomking/bi_deeplearning/
강의자료 : https://github.com/bi-lab/deeplearning_tutorial

TensorFlow License : Apache 2.0

https://github.com/tensorflow/tensorflow/blob/master/LICENSE

Apache 2.0 License :

자유로운 상업적 이용, 수정, 파생물 창작 및 배포가 가능하며 이 소프트웨어를 이용한 창작물을 공개하지 않아도 된다. 단, TensorFlow 로고는 따로 사용할 수 없고 반드시 저작권, 라이선스, 공지사항, 변경사항 문서를 첨부해야한다.



https://tldrlegal.com/license/apache-license-2.0-%28apache-2.0%29

TensorFlow의 핵심 원리

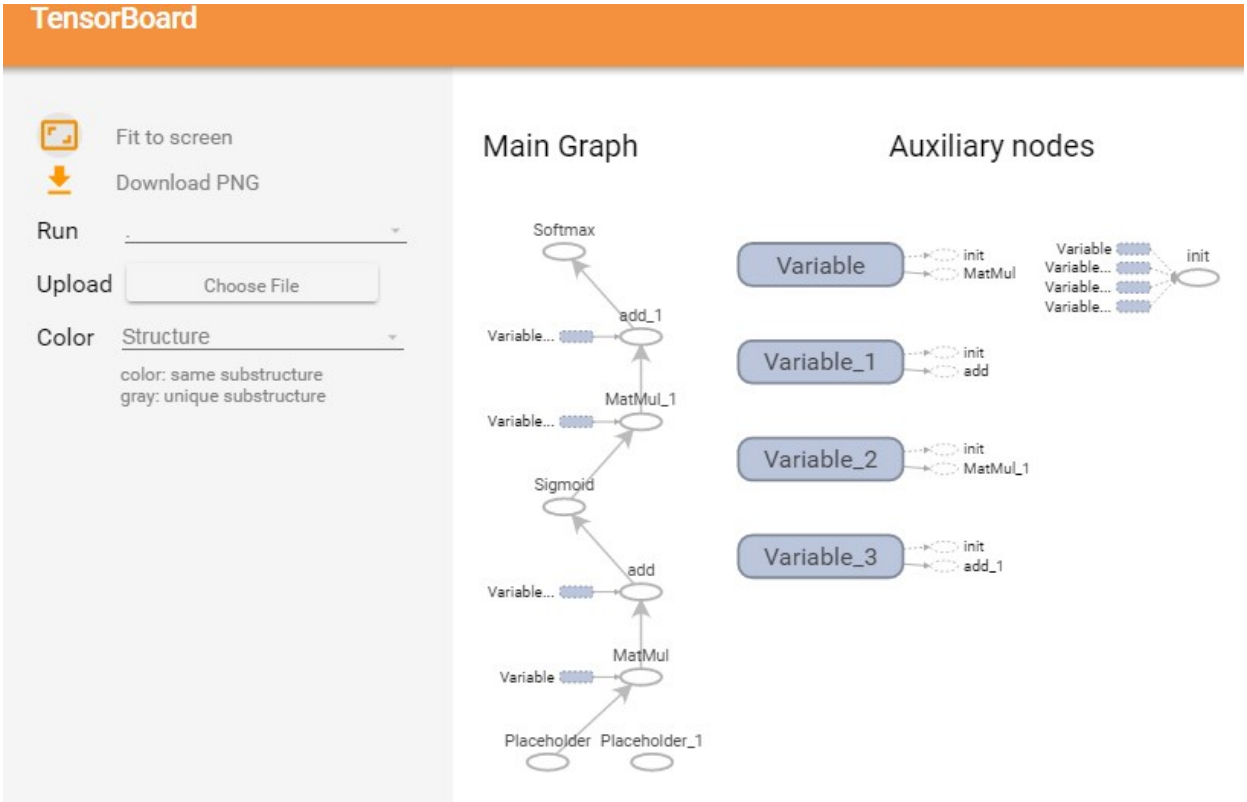
Computation Graph와 Session은 tf를 이해하는 데 가장 중요한 핵심 개념이다. 이 두가지 요소만 이해하면 전체적인 원리와 흐름을 모두 파악할 수 있다.

Computation Graph란 tf에서 실행되는 모든 알고리즘 혹은 모델을 나타내는 것이라 말할 수 있다. 보다 쉽게 설명하자면, tf에서는 하나의 거대한 함수가 실행되는데 이 함수는 내부적으로 수학적인 여러 계산을 통해 구성되어 있다. 이러한 계산 과정은 일련의 node와 edge들의 연결로써 하나의 graph를 이루게 되는데 이것이 바로 Computation Graph이다.

그래서 성공적으로 모델을 작성하게 되면 아래와 같은 Computation Graph(MLP의 feedforward 과정)가 만들어 진다. Computation Graph 기반의 프레임워크가 가진 장점은 모델 구현의 유연성이 크고, 자동화된 미분 계산이 가능하다는 점이 있다.

Session이란 한마디로 Computation Graph가 실행되는 환경 혹은 자원을 의미한다. 일반적으로 딥러닝 연산에는 GPU를 활용하게 되는데, Session은 이러한 GPU장치에서 확보한 메모리 및 프로세서 자원을 추상화시켜 표현한 것이다.

즉 이 두가지를 직관적으로 비유하자면, Computation Graph는 딥러닝 모델이고, Session은 이 모델이 실행되는 GPU라고 할 수 있다.

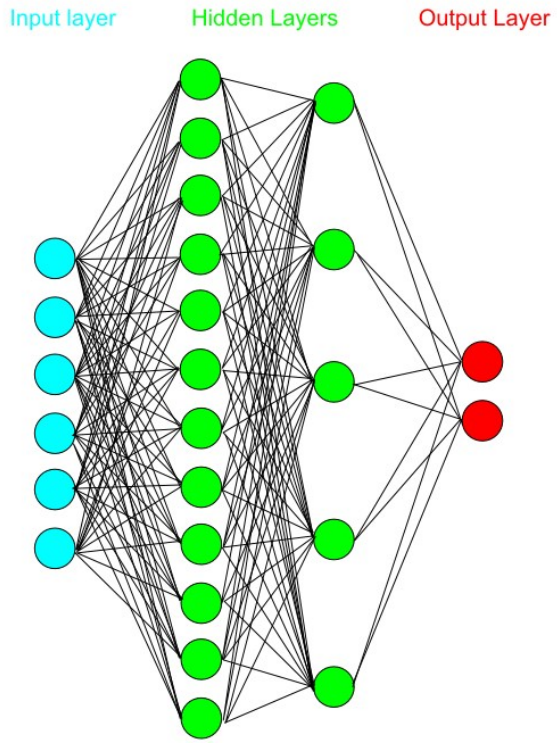


그러면 우선 기본적인 tf의 문법과 형식을 익히기 위해, 가장 기본적인 Deep Learning인 MLP의 Computation Graph를 아무것도 없는 상태에서 만들어보는 실습을 해보자.

Multi-Layer Perceptron

Multi-Layer Perceptron, MLP는 다음과 같은 구조를 가진 모델이다. Convolutional Neural Networks와 달리 굉장히 layer간의 연결이 빽

빽하게 가득 차 있어, dense layer 혹은 fully connected layer라는 이름으로도 불린다.



(출처: http://blog.refu.co/?p=931)

그러면 이러한 MLP를 이용해서 MNIST 데이터를 분류하는 코드를 작성해보자. MNIST 데이터는 다음과 같은 사람이 쓴 숫자 손글씨 7만장을 모아놓은 것으로, 기계학습에서 아주 널리 사용되는 데이터셋이다.

MNIST 데이터는 아래와 같이 28x28 크기의 이미지에 gray scale로 숫자가 그려져있고, 각 이미지가 어떤 숫자인지 label 정보가 달려있다.

그러면 가장 먼저 이를 구현하기위해서는 사용할 라이브러리를 import 해주어야 한다.

In [1]:
```
%matplotlib inline
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
print (tf.__version__)
```

```
0.10.0
```

그리고 실습에서 사용할 MNIST 데이터를 다음과 같이 다운로드 받는다.

In [2]:
```
# download the mnist 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
```

이제 mnist 데이터를 저장시킬 x와 y_target 변수를 선언해야한다. tf.placeholder는 우리가 원하는 데이터를 Computation Graph에 입력해주는 역할을 하는 변수이다. 즉 input data를 받기 위한 변수라고 생각할 수 있다.

In [3]:
```
# placeholder is used for feeding data.
x = tf.placeholder("float", shape=[None, 784]) # none represents variable length of dimension. 784 is
the dimension of MNIST data.
y_target = tf.placeholder("float", shape=[None, 10]) # shape argument is optional, but this is useful
to debug.
```

데이터를 넣을 변수를 생성했으니, 이제 실제 MLP에서 사용할 변수들을 생성하고, 이들을 이용해 Computation Graph를 그려본다.

In [4]:
```
# all the variables are allocated in GPU memory
W1 = tf.Variable(tf.zeros([784, 256]))      # create (784 * 256) matrix
b1 = tf.Variable(tf.zeros([256]))           # create (1 * 256) vector
weighted_summation1 = tf.matmul(x, W1) + b1 # compute --> weighted summation
h1 = tf.sigmoid( weighted_summation1 )      # compute --> sigmoid(weighted summation)

# Repeat again
W2 = tf.Variable(tf.zeros([256, 10]))        # create (256 * 10) matrix
b2 = tf.Variable(tf.zeros([10]))             # create (1 * 10) vector
weighted_summation2 = tf.matmul(h1, W2) + b2 # compute --> weighted summation
y = tf.nn.softmax(weighted_summation2)       # compute classification --> softmax(weighted summation)
```

위의 과정까지 완료된 경우, 변수 y는 3층짜리 MLP에서 input data에 대해 label을 예측한 결과가 저장된다. 지금까지 구현한 것은 MLP의 feed-forward process에 해당하는 Computation Graph이다.

그러면 이제 MLP를 학습시키기 위한, 수학적인 연산들을 정의해보자.

In [5]:
```
# define the Loss function
cross_entropy = -tf.reduce_sum(y_target*tf.log(y))
```

cross entorpy는 Deep Learning의 Classificatoin 모델에서 일반적으로 사용하는 에러함수이다. 간단히 말해 이 cross entropy는 MLP가 예측한 y 값이 실제 데이터와 다른 정도를 확률적으로 측정한다고 볼 수 있다.

MLP가 학습 되기위해서는 이 에러함수가 최대한 작은 값을 내도록 만들어야한다. 그래서 이 에러함수를 각각의 변수들로 편미분하여 gradient를 계산하고, 에러가 최소가 되는 변수값을 찾아가는 것이 바로 MLP의 학습 알고리즘이다.

이를 구현하려면 원래 미분 후 에러가 줄어드는 방향으로 변수를 이동시키는 과정등을 직접 코딩해야하지만, 자동 미분을 제공하는 tf에서는 아래의 단 한줄로 구현할 수 있다.

In [6]:
```
# define optimization algorithm
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

이제 학습 알고리즘의 구현이 끝났지만, 아직 더 필요한 기능이 있다. MLP가 잘 학습하고 있는지 성능을 모니터링하기 위한 정확도 계산을 정의해보자. 이 기능이 없으면 학습이 잘되고있는지를 전혀 파악할 수 없어 overfitting이 발생해도 해결할 수가 없다.

In [7]:
```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_target, 1))
# correct_prediction is list of boolean which is the result of comparing(model prediction , data)

accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
# tf.cast() : changes true -> 1 / false -> 0
# tf.reduce_mean() : calculate the mean
```

correct_prediction 는 뭔가 수식이 복잡해 보이지만 사실 매우 간단한 계산이다. 먼저 tf.argmax(y, 1) 함수는 y 벡터 중에서 가장 값이 큰 index를 알려주는 함수이다. 즉 모델이 예측한 class를 나타낸다. 그래서 모델이 예측한 class와 실제 데이터에 labeling된 class를 비교하여 같으면 true, 다르면 false를 내도록 계산한 것이 바로 correct_prediction이다.

accuracy는 앞서 계산한 correct_prediction 이라는 true/false 리스트를 1과 0으로 변환한 뒤, 이를 평균낸 것이다.

그러면 이제 필요한 모든 Computation Graph를 정의하였으니 이제 이를 session을 이용하여 실행만 시키면 된다. 일반적으로 하나의 Computation Graph는 하나의 session에 의해서 실행된다. 그럼 session을 생성해보자.

In [8]:
```
sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth =True))) # open a sessi
on which is a envrionment of computation graph.
sess.run(tf.initialize_all_variables())# initialize the variables
```

gpu_options=tf.GPUOptions(allow_growth =True)) 옵션은 session이 필요한 최소한의 자원만 할당해서 사용할 것을 강제하는 옵션이다.

tf.initialize_all_variables() 는 앞서 생성했던 변수들을 사용하기 위해서 반드시 실행해야하는 초기화 단계이다. 이 연산은 gpu 메모리

에 실제로 값을 할당하는 기능을 하기 때문에, 반드시 session에 의해서 실행되어야한다. session으로 실행을 하기위해서는 위와 같이 sess.run() 함수를 이용해 필요한 연산을 실행시킨다.

우선은 Ctrl + Enter를 눌러 다음 코드를 실행 시켜보자.

In [9]:
```
# training the MLP
for i in range(5001): # minibatch iteraction
    batch = mnist.train.next_batch(100) # minibatch size
    sess.run(train_step, feed_dict={x: batch[0], y_target: batch[1]}) # feed data into placeholder x,
y_target

    if i%500 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={x:batch[0], y_target: batch[1]})
        print ("step %d, training accuracy: %.3f"%(i, train_accuracy))

# for given x, y_target data set
print  ("test accuracy: %g"% sess.run(accuracy, feed_dict={x: mnist.test.images, y_target: mnist.test.
labels}))
sess.close()
tf.reset_default_graph()
```

```
step 0, training accuracy: 0.120
step 500, training accuracy: 0.570
step 1000, training accuracy: 0.880
step 1500, training accuracy: 0.900
step 2000, training accuracy: 0.910
step 2500, training accuracy: 0.940
step 3000, training accuracy: 0.950
step 3500, training accuracy: 0.950
step 4000, training accuracy: 0.970
step 4500, training accuracy: 0.960
step 5000, training accuracy: 0.950
test accuracy: 0.9299
```

총 5001번의 minibatch iteraction을 실행하고, 500번 마다 학습 정확도를 측정, 그리고 마지막에는 테스트 정확도를 측정하고 있다.

코드는 간단한 구조이다. for 문이 전체 iteration을 실행하고, 맨 처음 가져왔던 mnist 데이터를 100개씩 가져와서 placeholder에 넣어준다. 그리고 sess.run()을 통해 위에서 정의했던 학습 알고리즘과 정확도 계산을 실행한다.

이제 우리가 만든 MLP의 구조를 직접 눈으로 확인해보자.

TensorBoard 설정하기

TensorFlow는 TensorBoard라는 매우 강력한 visualization tool을 제공한다. 이를 사용하면 웹브라우저에서 사용자가 모델의 구조를 눈으로 확인하고, 파라미터 값의 분포를 살펴보는 등의 직관적인 분석이 가능하다. 다만, 아직은 실시간 데이터 업데이트 기능이 구현되지 않아 학습과정을 2분에 한번만 모니터링할 수가 있다. TensorBoard에는 현재도 수많은 기능이 구현중에 있고, 여러 오픈소스 개발자들의 기여를 기다리고 있다. (https://github.com/tensorflow/tensorflow/issues/2603)

그러면 이 TensorBoard를 활용해 방금 만들었던 MLP를 분석해보자. 그러려면 다음의 사항을 반영해 코드를 수정하여야 한다.

- **변수들의 이름 지어주기**
- **변수들의 Summary 생성**
- **변수들의 Summary 기록**

아래의 코드는 위의 3가지 사항을 모두 반영하고, MLP 코드를 하나의 파이썬 함수로 정리한 코드이다. 세세한 차이는 위에서 우리가 짰던 코드와 비교를 하면 파악이 가능하다.

In [10]:

```
%matplotlib inline
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
tf.reset_default_graph() # remove the previous computation graph

def MLP():
    # download the mnist data.
    mnist = input_data.read_data_sets('MNIST_data', one_hot=True)


    # placeholder is used for feeding data.
    x = tf.placeholder("float", shape=[None, 784], name = 'x') # none represents variable length of d
imension. 784 is the dimension of MNIST data.
    y_target = tf.placeholder("float", shape=[None, 10], name = 'y_target') # shape argument is optio
nal, but this is useful to debug.


    # all the variables are allocated in GPU memory
    W1 = tf.Variable(tf.zeros([784, 256]), name = 'W1')    # create (784 * 256) matrix
    b1 = tf.Variable(tf.zeros([256]), name = 'b1')         # create (1 * 256) vector
    h1 = tf.sigmoid(tf.matmul(x, W1) + b1, name = 'h1')    # compute --> sigmoid(weighted summation)

    # Repeat again
    W2 = tf.Variable(tf.zeros([256, 10]), name = 'W2')      # create (256 * 10) matrix
    b2 = tf.Variable(tf.zeros([10]), name = 'b2')           # create (1 * 10) vector
    y = tf.nn.softmax(tf.matmul(h1, W2) + b2, name = 'y')  # compute classification --> softmax(weigh
ted summation)


    # define the Loss function
    cross_entropy = -tf.reduce_sum(y_target*tf.log(y), name = 'cross_entropy')


    # define optimization algorithm
    train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)



    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_target, 1))
    # correct_prediction is list of boolean which is the result of comparing(model prediction , data)


    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    # tf.cast() : changes true -> 1 / false -> 0
    # tf.reduce_mean() : calculate the mean

    # Create Session
    sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth=True))) # open a s
ession which is a envrionment of computation graph.
    sess.run(tf.initialize_all_variables())# initialize the variables


    # create summary of parameters
    tf.histogram_summary('weights_1', W1)
    tf.histogram_summary('weights_2', W2)
    tf.histogram_summary('y', y)
    tf.scalar_summary('cross_entropy', cross_entropy)
    merged = tf.merge_all_summaries()
    summary_writer = tf.train.SummaryWriter("/tmp/mlp", sess.graph)


    # training the MLP
    for i in range(5001): # minibatch iteraction
        batch = mnist.train.next_batch(100) # minibatch size
        sess.run(train_step, feed_dict={x: batch[0], y_target: batch[1]}) # placeholder's none length
 is replaced by i:i+100 indexes

        if i%500 == 0:
            train_accuracy = sess.run(accuracy, feed_dict={x:batch[0], y_target: batch[1]})
            print ("step %d, training accuracy: %.3f"%(i, train_accuracy))



            # calculate the summary and write.
            summary = sess.run(merged, feed_dict={x:batch[0], y_target: batch[1]})
            summary_writer.add_summary(summary , i)

    # for given x, y_target data set
    print  ("test accuracy: %g"% sess.run(accuracy, feed_dict={x: mnist.test.images, y_target: mnist.
test.labels}))
    sess.close()

MLP()
```

```
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
step 0, training accuracy: 0.120
step 500, training accuracy: 0.520
step 1000, training accuracy: 0.720
step 1500, training accuracy: 0.910
step 2000, training accuracy: 0.930
step 2500, training accuracy: 0.920
step 3000, training accuracy: 0.860
step 3500, training accuracy: 0.960
step 4000, training accuracy: 0.960
step 4500, training accuracy: 0.960
step 5000, training accuracy: 0.930
test accuracy: 0.9304
```

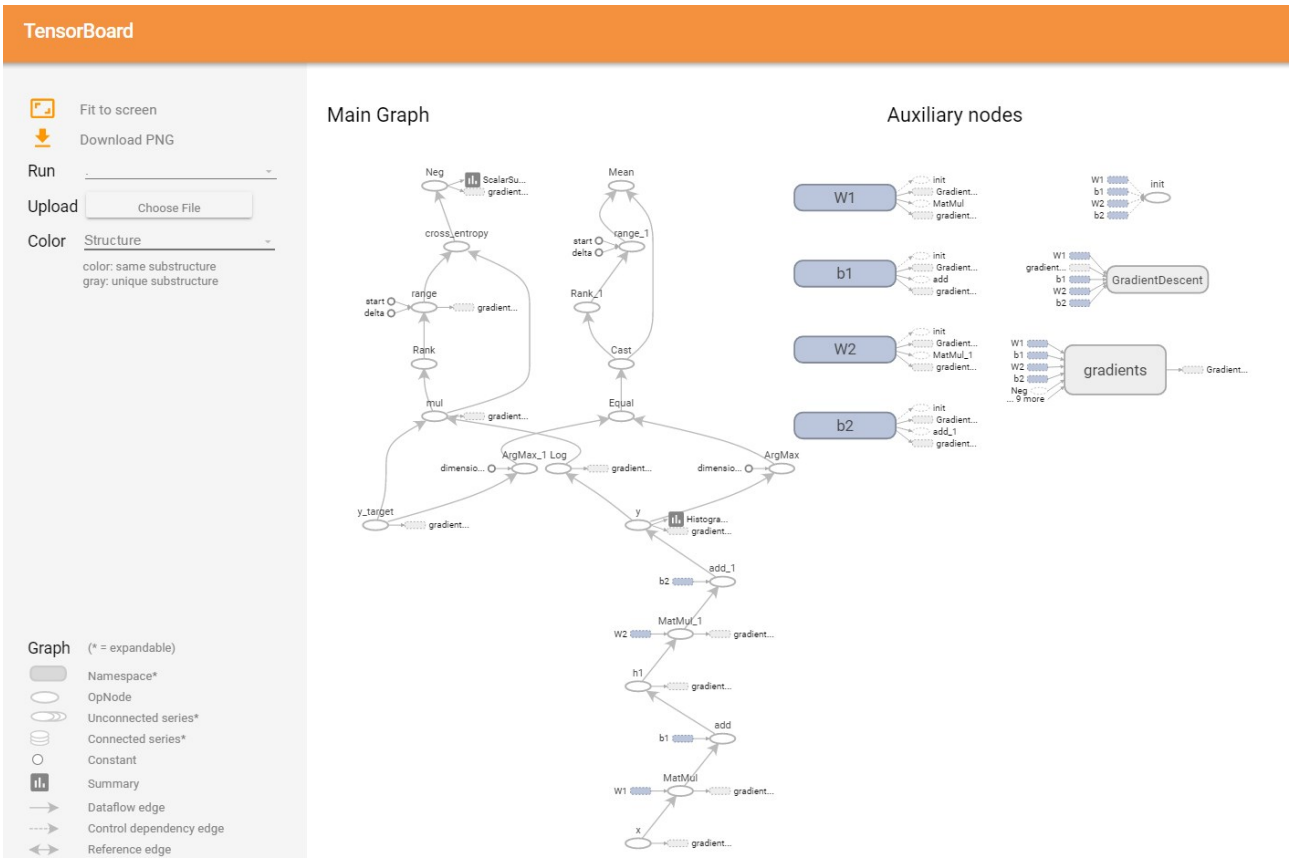
TensorBoard 실행하기

위와 같이 코드를 수정했다면, 이제 리눅스 shell로 이동한 후, tensorboard를 실행시킨다. 혹은 IPython에서 new -> terminal을 클릭하여 아래의 명령을 실행할 수도 있다. 종료는 ctrl + x를 입력한다.

tensorboard --logdir=/tmp/mlp --port=6006

(만약 위 명령어 실행시 문제가 생기는 경우 다음을 실행)
cd tensorflow/tensorflow/tensorboard
python tensorboard.py --logdir=/tmp/mlp --port=6006
https://github.com/tensorflow/tensorflow/blob/r0.10/tensorflow/tensorboard/README.md

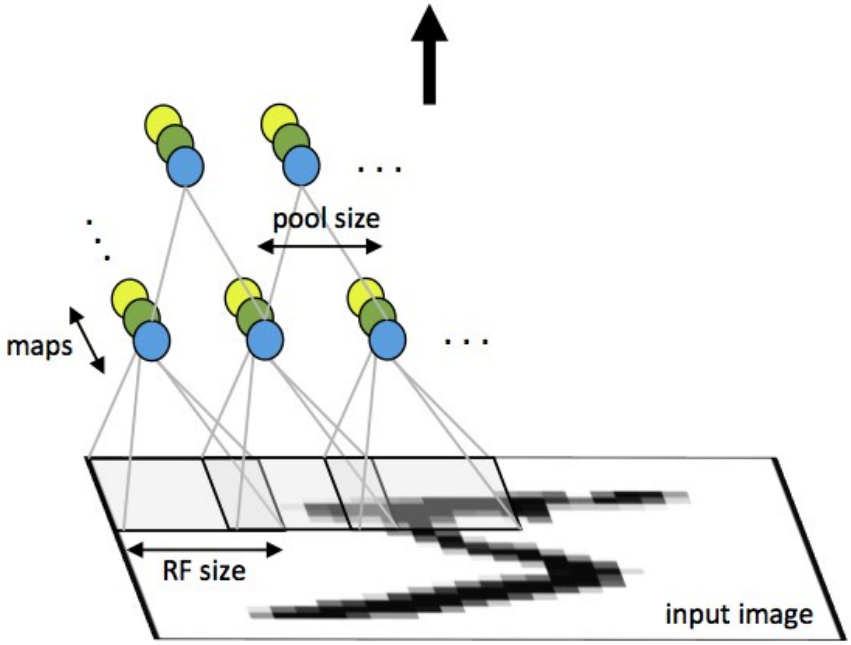
그다음 새로운 웹브라우저를 열어 IP:6006/#graphs 에 접속하면 아래와 같은 그림을 볼 수 있다.
(ex) http://192.168.99.100:6006/#graphs



이번에는 이미지 인식 분야에서 가장 성공적으로 쓰이고 있는 Convolutional Neural Networks를 실습해본다.

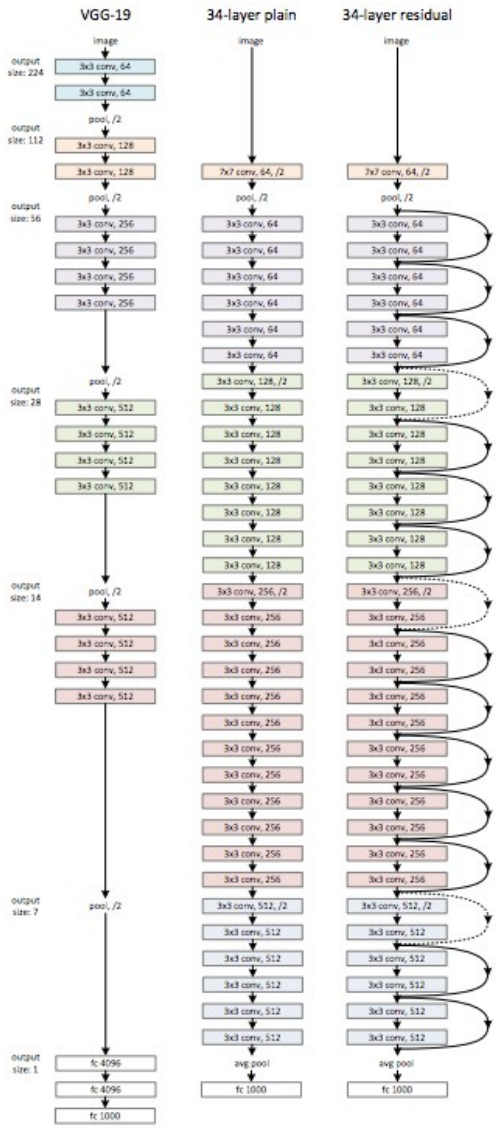
Convolutional Neural Networks

Convolutional Neural Networks, CNN은 아래와 같은 Convolutional Layer를 여러층 가진 딥러닝 모델을 뜻한다.



(출처: http://ufldl.stanford.edu/tutorial/images/Cnn_layer.png)

최근에는 아래와 같이 매우 깊은 층으로 Convolutional Layer를 쌓고 매우 방대한 량의 이미지를 학습하는 경우가 많다.



(출처: https://qph.is.quoracdn.net/main-qimg-cf89aa517e5b641dc8e41e7a57bafc2c?convert_to_webp=true)

이번에는 간단한 구조를 가진 CNN을 구현하고 방금전에 사용했던 MNIST 데이터를 학습시켜 보고, MLP와의 성능 차이를 비교해본다.

아래의 코드를 보면 MLP와 전체 구조는 매우 유사한데, 중간에 conv2d나 max-pool을 비롯해 처음 보는 여러 연산들이 추가 된 것을 알 수 있다. 또한 CNN을 효과적으로 학습하기 위해서는 Weight의 초기화를 0으로 하는 것이 아니라, 랜덤으로 해주어야하는데, 여기서는 가우시안을 이용하여 초기화 하였다. 그밖에 dropout과 relu, Adam 등이 추가로 사용되었다.

각 함수와 연산들의 자세한 설명은 아래 코드를 보면서 하나하나 분석해보자.

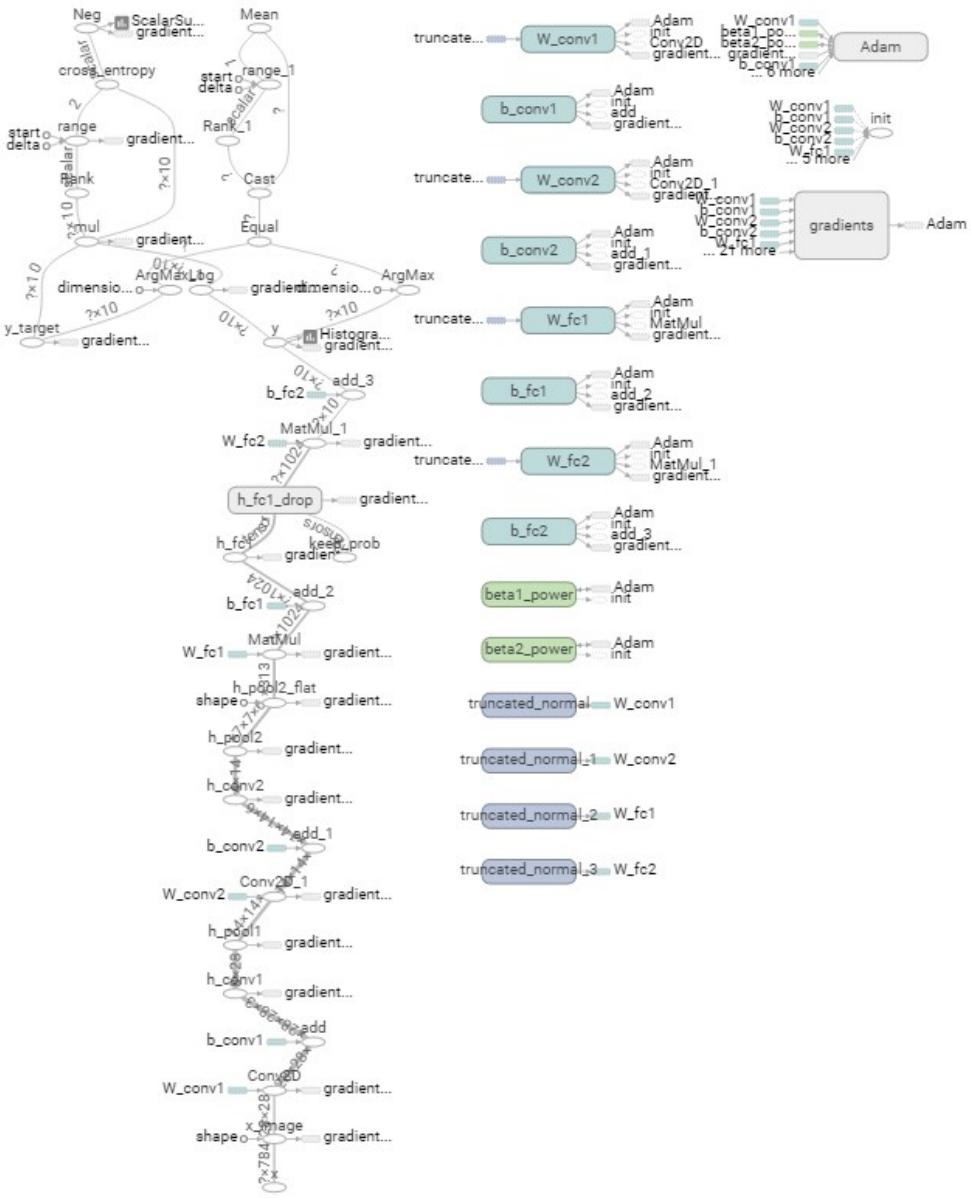
In [5]:
```
%matplotlib inline
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
tf.reset_default_graph() # remove the previous computation graph

def CNN():
    # download the mnist data.
    mnist = input_data.read_data_sets('MNIST_data', one_hot=True)


    # placeholder is used for feeding data.
    x = tf.placeholder("float", shape=[None, 784], name = 'x') # none represents variable length of di
mension. 784 is the dimension of MNIST data.
    y_target = tf.placeholder("float", shape=[None, 10], name = 'y_target') # shape argument is option
al, but this is useful to debug.



    # reshape input data
    x_image = tf.reshape(x, [-1,28,28,1], name="x_image")

    # Build a convolutional layer and maxpooling with random initialization
    W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1), name="W_conv1") # W is [row,
 col, channel, feature]
    b_conv1 = tf.Variable(tf.zeros([32]), name="b_conv1")
    h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv
1, name="h_conv1")
    h_pool1 = tf.nn.max_pool( h_conv1 , ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', name
 = "h_pool1")

    # Repeat again with 64 number of filters
    W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1), name="W_conv2") # W is [row
, col, channel, feature]
    b_conv2 = tf.Variable(tf.zeros([64]), name="b_conv2")
    h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv
2, name="h_conv2")
    h_pool2 = tf.nn.max_pool( h_conv2 , ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', name
 = "h_pool2")

    # Build a fully connected layer
    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64], name="h_pool2_flat")
    W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1), name = 'W_fc1')
    b_fc1 = tf.Variable(tf.zeros([1024]), name = 'b_fc1')
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1, name="h_fc1")


    # Dropout Layer
    keep_prob = tf.placeholder("float", name="keep_prob")
    h_fc1 = tf.nn.dropout(h_fc1, keep_prob, name="h_fc1_drop")

    # Build a fully connected layer with softmax
    W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1), name = 'W_fc2')
    b_fc2 = tf.Variable(tf.zeros([10]), name = 'b_fc2')
    y=tf.nn.softmax(tf.matmul(h_fc1, W_fc2) + b_fc2, name="y")




    # define the Loss function
    cross_entropy = -tf.reduce_sum(y_target*tf.log(y), name = 'cross_entropy')


    # define optimization algorithm
    #train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)



    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_target, 1))
    # correct_prediction is list of boolean which is the result of comparing(model prediction , data)


    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    # tf.cast() : changes true -> 1 / false -> 0
    # tf.reduce_mean() : calculate the mean

    # Create Session
    sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth =True)))    # open a
 session which is a envrionment of computation graph.
    sess.run(tf.initialize_all_variables())# initialize the variables


    # create summary of parameters
    tf.histogram_summary('weights_1', W_conv1)
    tf.histogram_summary('weights_2', W_conv2)
```

```
    tf.histogram_summary('y', y)
    tf.scalar_summary('cross_entropy', cross_entropy)
    merged = tf.merge_all_summaries()
    summary_writer = tf.train.SummaryWriter("/tmp/cnn", sess.graph)



    # training the MLP
    for i in range(5001): # minibatch iteraction
        batch = mnist.train.next_batch(100) # minibatch size
        sess.run(train_step, feed_dict={x: batch[0], y_target: batch[1], keep_prob: 0.5}) # placeholde
r's none length is replaced by i:i+100 indexes

        if i%500 == 0:
            train_accuracy = sess.run(accuracy, feed_dict={x:batch[0], y_target: batch[1], keep_prob:
1})
            print ("step %d, training accuracy: %.3f"%(i, train_accuracy))

            # calculate the summary and write.
            summary = sess.run(merged, feed_dict={x:batch[0], y_target: batch[1], keep_prob: 1})
            summary_writer.add_summary(summary , i)

    # for given x, y_target data set
    print  ("test accuracy: %g"% sess.run(accuracy, feed_dict={x: mnist.test.images[0:150], y_target:
mnist.test.labels[0:150], keep_prob: 1}))
    sess.close()
CNN()
```

```
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
step 0, training accuracy: 0.060
step 500, training accuracy: 0.980
step 1000, training accuracy: 0.990
step 1500, training accuracy: 0.990
step 2000, training accuracy: 1.000
step 2500, training accuracy: 1.000
step 3000, training accuracy: 1.000
step 3500, training accuracy: 1.000
step 4000, training accuracy: 0.990
step 4500, training accuracy: 1.000
step 5000, training accuracy: 1.000
test accuracy: 1
```

학습 결과, MLP보다 성능이 훨씬 높아진 것을 알 수 있고 동시에 학습 속도가 보다 느려진 것을 확인할 수 있다. (GPU 메모리가 충분하지 않아, test 정확도는 150개의 이미지에 대해서만 측정하였다.)

이제 마찬가지로 TensorBoard를 통해서 우리가 만든 Computation Graph를 직접 눈으로 확인해보면 다음과 같이 그려진다.



Early Stopping 및 Index Shuffling

그런데 실제로 Deep Learning을 학습할 때에는 우리가 정한 횟수만큼의 iteration을 무조건 반복하는 것이 아니라, 적당히 학습이 완료되었다고 생각되면 학습을 중단하는 Early Stopping을 해야한다. 이것을 하지 않고 무조건 정해진 iteration을 하게되면, 모델이 주어진 데이터에만 과도하게 학습하여, 보지않은 데이터에 대한 일반화 성능이 떨어지는 overfitting이 일어나게 된다. 따라서 Early Stopping을 통해 이러한 일이 일어나기 전에 학습을 중단해야 한다.

또한 위의 MNIST 데이터는 이미 구현된 함수를 통해 미리 순서가 뒤섞이고, one-hot coding이 된 데이터를 필요한 개수만큼 가져올 수 있었다. 그러나 실제로 자신의 데이터를 학습시키기 위해서는 이러한 과정도 파이썬의 numpy를 이용해 구현해 주어야한다.

그래서 이번 예제에서는 아주 간단하면서 유명한 Iris 데이터를 이용해 위의 구현을 실습해보도록 한다.

Iris data : 50개*3종의 iris 꽃에서 4종류의 feature를 추출한 데이터
https://en.wikipedia.org/wiki/Iris_flower_data_set

In [16]:

```
%matplotlib inline
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.datasets.base import load_iris
import numpy as np
tf.reset_default_graph()

def MLP_iris():
    # load the iris data.
    iris = load_iris()

    np.random.seed(0)
    random_index = np.random.permutation(150)

    iris_data = iris.data[random_index]
    iris_target = iris.target[random_index]
    iris_target_onehot = np.zeros((150, 3))
    iris_target_onehot[np.arange(150), iris_target] = 1

    accuracy_list = []

    # build computation graph
    x = tf.placeholder("float", shape=[None, 4], name = 'x')
    y_target = tf.placeholder("float", shape=[None, 3], name = 'y_target')

    W1 = tf.Variable(tf.zeros([4, 128]), name = 'W1')
    b1 = tf.Variable(tf.zeros([128]), name = 'b1')
    h1 = tf.sigmoid(tf.matmul(x, W1) + b1, name = 'h1')

    W2 = tf.Variable(tf.zeros([128, 3]), name = 'W2')
    b2 = tf.Variable(tf.zeros([3]), name = 'b2')
    y = tf.nn.softmax(tf.matmul(h1, W2) + b2, name = 'y')

    cross_entropy = -tf.reduce_sum(y_target*tf.log(y), name = 'cross_entropy')

    train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_target, 1))

    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

    sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth=True)))
    sess.run(tf.initialize_all_variables())

    for i in range(500):
        sess.run(train_step, feed_dict={x: iris_data[0:100], y_target: iris_target_onehot[0:100]})

        train_accuracy = sess.run(accuracy, feed_dict={x: iris_data[0:100], y_target: iris_target_one
hot[0:100]})
        validation_accuracy = sess.run(accuracy, feed_dict={x: iris_data[100:], y_target: iris_target
_onehot[100:]})
        print ("step %d, training accuracy: %.3f / validation accuracy: %.3f" %(i, train_accuracy, va
lidation_accuracy))

        accuracy_list.append(validation_accuracy)

        if i >= 50:
            if validation_accuracy - np.mean(accuracy_list[len(accuracy_list)/2:]) <= 0.01 :
                break

    sess.close()

MLP_iris()
```

```
step 0, training accuracy: 0.360 / validation accuracy: 0.280
step 1, training accuracy: 0.310 / validation accuracy: 0.380
step 2, training accuracy: 0.330 / validation accuracy: 0.340
step 3, training accuracy: 0.360 / validation accuracy: 0.280
step 4, training accuracy: 0.360 / validation accuracy: 0.280
step 5, training accuracy: 0.350 / validation accuracy: 0.180
step 6, training accuracy: 0.130 / validation accuracy: 0.040
step 7, training accuracy: 0.010 / validation accuracy: 0.000
step 8, training accuracy: 0.010 / validation accuracy: 0.000
step 9, training accuracy: 0.020 / validation accuracy: 0.020
step 10, training accuracy: 0.040 / validation accuracy: 0.020
step 11, training accuracy: 0.060 / validation accuracy: 0.060
step 12, training accuracy: 0.090 / validation accuracy: 0.080
step 13, training accuracy: 0.100 / validation accuracy: 0.120
step 14, training accuracy: 0.110 / validation accuracy: 0.160
step 15, training accuracy: 0.470 / validation accuracy: 0.500
step 16, training accuracy: 0.640 / validation accuracy: 0.660
step 17, training accuracy: 0.660 / validation accuracy: 0.660
step 18, training accuracy: 0.660 / validation accuracy: 0.660
step 19, training accuracy: 0.660 / validation accuracy: 0.660
step 20, training accuracy: 0.660 / validation accuracy: 0.660
```

```
step 21, training accuracy: 0.660 / validation accuracy: 0.660
step 22, training accuracy: 0.660 / validation accuracy: 0.660
step 23, training accuracy: 0.660 / validation accuracy: 0.660
step 24, training accuracy: 0.660 / validation accuracy: 0.660
step 25, training accuracy: 0.660 / validation accuracy: 0.660
step 26, training accuracy: 0.660 / validation accuracy: 0.660
step 27, training accuracy: 0.660 / validation accuracy: 0.660
step 28, training accuracy: 0.660 / validation accuracy: 0.660
step 29, training accuracy: 0.660 / validation accuracy: 0.660
step 30, training accuracy: 0.660 / validation accuracy: 0.660
step 31, training accuracy: 0.660 / validation accuracy: 0.660
step 32, training accuracy: 0.660 / validation accuracy: 0.660
step 33, training accuracy: 0.660 / validation accuracy: 0.660
step 34, training accuracy: 0.660 / validation accuracy: 0.660
step 35, training accuracy: 0.660 / validation accuracy: 0.660
step 36, training accuracy: 0.660 / validation accuracy: 0.660
step 37, training accuracy: 0.670 / validation accuracy: 0.660
step 38, training accuracy: 0.680 / validation accuracy: 0.680
step 39, training accuracy: 0.690 / validation accuracy: 0.680
step 40, training accuracy: 0.700 / validation accuracy: 0.680
step 41, training accuracy: 0.730 / validation accuracy: 0.680
step 42, training accuracy: 0.730 / validation accuracy: 0.720
step 43, training accuracy: 0.730 / validation accuracy: 0.760
step 44, training accuracy: 0.730 / validation accuracy: 0.760
step 45, training accuracy: 0.730 / validation accuracy: 0.760
step 46, training accuracy: 0.750 / validation accuracy: 0.780
step 47, training accuracy: 0.770 / validation accuracy: 0.780
step 48, training accuracy: 0.770 / validation accuracy: 0.780
step 49, training accuracy: 0.770 / validation accuracy: 0.800
step 50, training accuracy: 0.780 / validation accuracy: 0.800
step 51, training accuracy: 0.780 / validation accuracy: 0.800
step 52, training accuracy: 0.790 / validation accuracy: 0.800
step 53, training accuracy: 0.790 / validation accuracy: 0.800
step 54, training accuracy: 0.790 / validation accuracy: 0.800
step 55, training accuracy: 0.790 / validation accuracy: 0.800
step 56, training accuracy: 0.790 / validation accuracy: 0.820
step 57, training accuracy: 0.800 / validation accuracy: 0.820
step 58, training accuracy: 0.810 / validation accuracy: 0.820
step 59, training accuracy: 0.810 / validation accuracy: 0.820
step 60, training accuracy: 0.810 / validation accuracy: 0.820
step 61, training accuracy: 0.810 / validation accuracy: 0.820
step 62, training accuracy: 0.810 / validation accuracy: 0.820
step 63, training accuracy: 0.810 / validation accuracy: 0.820
step 64, training accuracy: 0.810 / validation accuracy: 0.820
step 65, training accuracy: 0.810 / validation accuracy: 0.820
step 66, training accuracy: 0.820 / validation accuracy: 0.820
step 67, training accuracy: 0.830 / validation accuracy: 0.820
step 68, training accuracy: 0.830 / validation accuracy: 0.820
step 69, training accuracy: 0.830 / validation accuracy: 0.840
step 70, training accuracy: 0.830 / validation accuracy: 0.840
step 71, training accuracy: 0.830 / validation accuracy: 0.860
step 72, training accuracy: 0.850 / validation accuracy: 0.860
step 73, training accuracy: 0.850 / validation accuracy: 0.860
step 74, training accuracy: 0.850 / validation accuracy: 0.860
step 75, training accuracy: 0.860 / validation accuracy: 0.880
step 76, training accuracy: 0.860 / validation accuracy: 0.880
step 77, training accuracy: 0.870 / validation accuracy: 0.880
step 78, training accuracy: 0.870 / validation accuracy: 0.880
step 79, training accuracy: 0.870 / validation accuracy: 0.880
step 80, training accuracy: 0.890 / validation accuracy: 0.880
step 81, training accuracy: 0.900 / validation accuracy: 0.880
step 82, training accuracy: 0.900 / validation accuracy: 0.880
step 83, training accuracy: 0.900 / validation accuracy: 0.880
step 84, training accuracy: 0.900 / validation accuracy: 0.880
step 85, training accuracy: 0.900 / validation accuracy: 0.880
step 86, training accuracy: 0.900 / validation accuracy: 0.900
step 87, training accuracy: 0.910 / validation accuracy: 0.900
step 88, training accuracy: 0.920 / validation accuracy: 0.900
step 89, training accuracy: 0.920 / validation accuracy: 0.920
step 90, training accuracy: 0.920 / validation accuracy: 0.920
step 91, training accuracy: 0.920 / validation accuracy: 0.920
step 92, training accuracy: 0.920 / validation accuracy: 0.920
step 93, training accuracy: 0.920 / validation accuracy: 0.920
step 94, training accuracy: 0.920 / validation accuracy: 0.920
step 95, training accuracy: 0.920 / validation accuracy: 0.920
step 96, training accuracy: 0.920 / validation accuracy: 0.920
step 97, training accuracy: 0.920 / validation accuracy: 0.920
step 98, training accuracy: 0.920 / validation accuracy: 0.940
step 99, training accuracy: 0.920 / validation accuracy: 0.940
step 100, training accuracy: 0.920 / validation accuracy: 0.940
step 101, training accuracy: 0.920 / validation accuracy: 0.940
step 102, training accuracy: 0.920 / validation accuracy: 0.940
step 103, training accuracy: 0.920 / validation accuracy: 0.940
step 104, training accuracy: 0.930 / validation accuracy: 0.940
step 105, training accuracy: 0.930 / validation accuracy: 0.940
```

```
step 106, training accuracy: 0.940 / validation accuracy: 0.940
step 107, training accuracy: 0.940 / validation accuracy: 0.940
step 108, training accuracy: 0.940 / validation accuracy: 0.940
step 109, training accuracy: 0.940 / validation accuracy: 0.940
step 110, training accuracy: 0.940 / validation accuracy: 0.940
step 111, training accuracy: 0.940 / validation accuracy: 0.940
step 112, training accuracy: 0.940 / validation accuracy: 0.940
step 113, training accuracy: 0.940 / validation accuracy: 0.940
step 114, training accuracy: 0.940 / validation accuracy: 0.960
step 115, training accuracy: 0.940 / validation accuracy: 0.960
step 116, training accuracy: 0.940 / validation accuracy: 0.960
step 117, training accuracy: 0.940 / validation accuracy: 0.960
step 118, training accuracy: 0.940 / validation accuracy: 0.960
step 119, training accuracy: 0.940 / validation accuracy: 0.960
step 120, training accuracy: 0.940 / validation accuracy: 0.960
step 121, training accuracy: 0.940 / validation accuracy: 0.960
step 122, training accuracy: 0.940 / validation accuracy: 0.960
step 123, training accuracy: 0.940 / validation accuracy: 0.960
step 124, training accuracy: 0.940 / validation accuracy: 0.960
step 125, training accuracy: 0.940 / validation accuracy: 0.960
step 126, training accuracy: 0.940 / validation accuracy: 0.960
step 127, training accuracy: 0.950 / validation accuracy: 0.960
step 128, training accuracy: 0.950 / validation accuracy: 0.960
step 129, training accuracy: 0.950 / validation accuracy: 0.960
step 130, training accuracy: 0.950 / validation accuracy: 0.960
step 131, training accuracy: 0.950 / validation accuracy: 0.960
step 132, training accuracy: 0.960 / validation accuracy: 0.960
step 133, training accuracy: 0.960 / validation accuracy: 0.960
step 134, training accuracy: 0.960 / validation accuracy: 0.960
step 135, training accuracy: 0.960 / validation accuracy: 0.960
step 136, training accuracy: 0.960 / validation accuracy: 0.960
step 137, training accuracy: 0.960 / validation accuracy: 0.960
step 138, training accuracy: 0.960 / validation accuracy: 0.960
step 139, training accuracy: 0.960 / validation accuracy: 0.960
step 140, training accuracy: 0.960 / validation accuracy: 0.960
step 141, training accuracy: 0.960 / validation accuracy: 0.960
step 142, training accuracy: 0.960 / validation accuracy: 0.960
step 143, training accuracy: 0.960 / validation accuracy: 0.960
step 144, training accuracy: 0.960 / validation accuracy: 0.960
step 145, training accuracy: 0.960 / validation accuracy: 0.960
step 146, training accuracy: 0.960 / validation accuracy: 0.960
step 147, training accuracy: 0.960 / validation accuracy: 0.960
step 148, training accuracy: 0.960 / validation accuracy: 0.960
step 149, training accuracy: 0.960 / validation accuracy: 0.960
step 150, training accuracy: 0.960 / validation accuracy: 0.960
step 151, training accuracy: 0.960 / validation accuracy: 0.960
step 152, training accuracy: 0.960 / validation accuracy: 0.960
step 153, training accuracy: 0.960 / validation accuracy: 0.960
step 154, training accuracy: 0.960 / validation accuracy: 0.960
step 155, training accuracy: 0.960 / validation accuracy: 0.960
step 156, training accuracy: 0.960 / validation accuracy: 0.960
step 157, training accuracy: 0.960 / validation accuracy: 0.960
step 158, training accuracy: 0.960 / validation accuracy: 0.960
step 159, training accuracy: 0.960 / validation accuracy: 0.960
step 160, training accuracy: 0.960 / validation accuracy: 0.960
step 161, training accuracy: 0.970 / validation accuracy: 0.960
step 162, training accuracy: 0.970 / validation accuracy: 0.960
step 163, training accuracy: 0.970 / validation accuracy: 0.960
step 164, training accuracy: 0.970 / validation accuracy: 0.960
step 165, training accuracy: 0.970 / validation accuracy: 0.960
step 166, training accuracy: 0.970 / validation accuracy: 0.960
step 167, training accuracy: 0.970 / validation accuracy: 0.960
step 168, training accuracy: 0.970 / validation accuracy: 0.960
step 169, training accuracy: 0.970 / validation accuracy: 0.960
step 170, training accuracy: 0.970 / validation accuracy: 0.960
step 171, training accuracy: 0.970 / validation accuracy: 0.960
```

Save & Load Parameters

그러면 이번에는 이렇게 학습한 모델을 필요한 시점에서 저장하고, 다시 불러오는 기능을 구현해보자. 실제 문제를 풀 때에는 위와 같이 빠른 학습이 불가능하기 때문에 반드시 학습된 모델을 저장하고, 필요할 때 불러오는 기능이 필요하다.

In [18]:
```
%matplotlib inline
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

tf.reset_default_graph()

def MLP():
    # download the mnist data.
    mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

```
    # placeholder is used for feeding data.
    x = tf.placeholder("float", shape=[None, 784], name = 'x') # none represents variable length of d
imension. 784 is the dimension of MNIST data.
    y_target = tf.placeholder("float", shape=[None, 10], name = 'y_target') # shape argument is optio
nal, but this is useful to debug.


    # all the variables are allocated in GPU memory
    W1 = tf.Variable(tf.zeros([784, 256]), name = 'W1')    # create (784 * 256) matrix
    b1 = tf.Variable(tf.zeros([256]), name = 'b1')         # create (1 * 256) vector
    h1 = tf.sigmoid(tf.matmul(x, W1) + b1, name = 'h1')    # compute --> sigmoid(weighted summation)

    # Repeat again
    W2 = tf.Variable(tf.zeros([256, 10]), name = 'W2')      # create (256 * 10) matrix
    b2 = tf.Variable(tf.zeros([10]), name = 'b2')           # create (1 * 10) vector
    y = tf.nn.softmax(tf.matmul(h1, W2) + b2, name = 'y')  # compute classification --> softmax(weigh
ted summation)


    # define the Loss function
    cross_entropy = -tf.reduce_sum(y_target*tf.log(y), name = 'cross_entropy')


    # define optimization algorithm
    train_step = tf.train.GradientDescentOptimizer(0.01, name='GradientDescent').minimize(cross_entro
py)



    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_target, 1))
    # correct_prediction is list of boolean which is the result of comparing(model prediction , data)


    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"), name='accuracy')
    # tf.cast() : changes true -> 1 / false -> 0
    # tf.reduce_mean() : calculate the mean

    # Create Session
    sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth=True))) # open a s
ession which is a envrionment of computation graph.
    sess.run(tf.initialize_all_variables())# initialize the variables

    summary_writer = tf.train.SummaryWriter("/tmp/mlp", sess.graph)


    # Create Directory
    import os
    if not os.path.exists('Checkpoint'):
        os.makedirs('Checkpoint')

    # Create Saver
    saver = tf.train.Saver(max_to_keep=11)
    saver.restore(sess, os.path.join('Checkpoint', "mlp-5000"))

    # training the MLP
    for i in range(5001): # minibatch iteraction
        batch = mnist.train.next_batch(100) # minibatch size
        sess.run(train_step, feed_dict={x: batch[0], y_target: batch[1]}) # placeholder's none length
 is replaced by i:i+100 indexes

        if i%500 == 0:
            train_accuracy = sess.run(accuracy, feed_dict={x:batch[0], y_target: batch[1]})
            print ("step %d, training accuracy: %.3f"%(i, train_accuracy))

            saver.save(sess,  os.path.join('Checkpoint', "mlp") , global_step=i)


    # for given x, y_target data set
    print  ("test accuracy: %g"% sess.run(accuracy, feed_dict={x: mnist.test.images, y_target: mnist.
test.labels}))
    sess.close()

MLP()
```

```
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
step 0, training accuracy: 0.990
step 500, training accuracy: 0.950
step 1000, training accuracy: 0.920
step 1500, training accuracy: 0.960
step 2000, training accuracy: 0.950
step 2500, training accuracy: 0.960
step 3000, training accuracy: 0.980
step 3500, training accuracy: 0.950
step 4000, training accuracy: 0.960
step 4500, training accuracy: 0.970
step 5000, training accuracy: 0.940
test accuracy: 0.9409
```

Load Pre-defined Computation Graph and Trained Parameters

위의 예제에서는 한가지 부족한 점이 있다. 그것은 바로 학습된 parameter를 저장하고 불러올 수는 있지만, 이미 만들어진 Computation Graph를 불러오는 기능이 빠져있다. 이번에는 Computation Graph를 불러오는 부분까지 포함해서 실습해보자.

In [20]:
```
%matplotlib inline
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
tf.reset_default_graph()

sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth=True))) # open a sessi
on which is a envrionment of computation graph.
sess.run(tf.initialize_all_variables())# initialize the variables

new_saver = tf.train.import_meta_graph('Checkpoint/mlp-0.meta')
new_saver.restore(sess, 'Checkpoint/mlp-500')

train_step = sess.graph.get_operation_by_name('GradientDescent')
x = sess.graph.get_tensor_by_name('x:0')
y_target = sess.graph.get_tensor_by_name('y_target:0')
accuracy = sess.graph.get_tensor_by_name('accuracy:0')

mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

# training the MLP
for i in range(5001): # minibatch iteraction
    batch = mnist.train.next_batch(100) # minibatch size
    sess.run(train_step, feed_dict={x: batch[0], y_target: batch[1]}) # placeholder's none length is
replaced by i:i+100 indexes

    if i%500 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={x:batch[0], y_target: batch[1]})
        print ("step %d, training accuracy: %.3f"%(i, train_accuracy))

# for given x, y_target data set
print  ("test accuracy: %g"% sess.run(accuracy, feed_dict={x: mnist.test.images, y_target: mnist.test
.labels}))
sess.close()
```

```
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
step 0, training accuracy: 0.990
step 500, training accuracy: 0.950
step 1000, training accuracy: 0.970
step 1500, training accuracy: 0.960
step 2000, training accuracy: 0.930
step 2500, training accuracy: 0.960
step 3000, training accuracy: 0.950
step 3500, training accuracy: 0.970
step 4000, training accuracy: 0.970
step 4500, training accuracy: 0.970
step 5000, training accuracy: 0.980
test accuracy: 0.9417
```

TF learn

지금까지 우리는 아무것도 없는 상태에서 출발해 필요한 모델을 전부 구현하는 방법을 실습했다. 그러나 위의 예제들은 보다 추상화된 wrapping 함수들을 통해 충분히 더 간단하고 빠르게 구현이 가능하다. TensorFlow에는 이러한 wrapping 라이브러리가 굉장히 많이 존재한다. 그 중에서 가장 공식적인 secondary wrapping 라이브러리로는 바로 TF learn이 있다. 물론 TF learn은 layer를 직접 설계해서 사용하는 하드코어 유저에게는 적합하지 않다. 그러나 빠르고 단시간내에 기존에 알려진 layer로 모델을 구축할 경우 매우 편리하게 사용할 수 있다.

https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/learn/python/learn
https://github.com/tensorflow/tensorflow/tree/r0.10/tensorflow/examples/skflow

이번에는 이 TF learn을 이용해 얼마나 짧은 코드로 MLP를 구현할 수 있는 지 살펴본다.

In [21]:
```
import tensorflow.contrib.learn.python.learn as learn
from sklearn import datasets, metrics

iris = datasets.load_iris()
feature_columns = learn.infer_real_valued_columns_from_input(iris.data)# Specify that all features ha
ve real-value data. This interprets all inputs as dense, fixed-length float values.
classifier = learn.DNNClassifier(feature_columns=feature_columns, hidden_units=[100, 200, 100], n_cla
sses=3)
classifier.fit(iris.data, iris.target, steps=500, batch_size=32)
iris_predictions = list(classifier.predict(iris.data, as_iterable=True))
score = metrics.accuracy_score(iris.target, iris_predictions)
print("Accuracy: %f" % score)
```

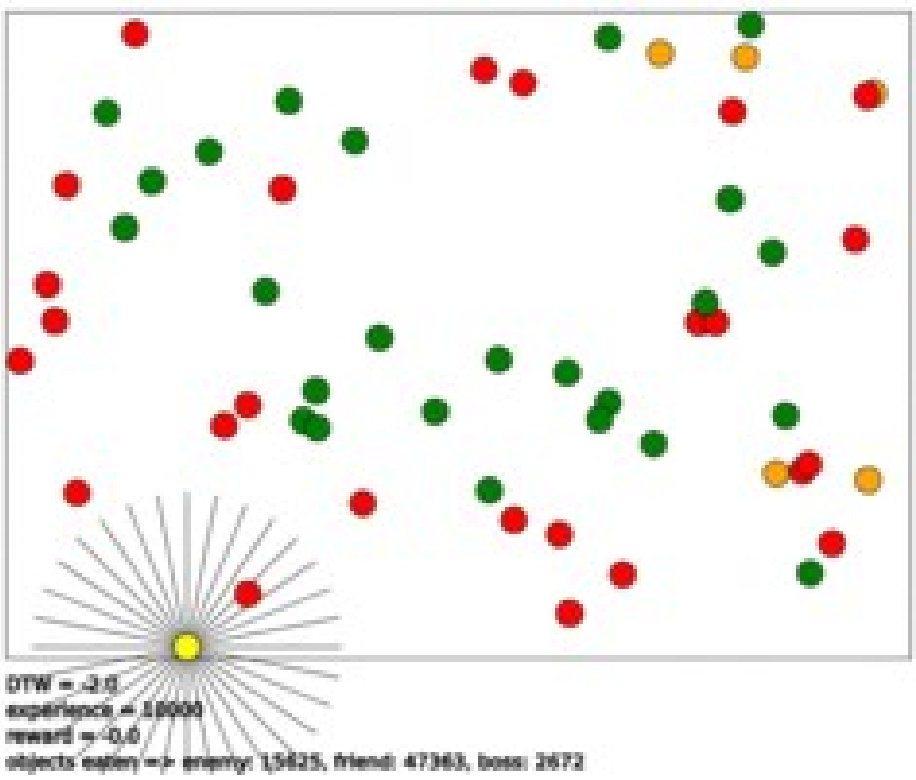
```
WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpIanytJ
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpIanytJ
WARNING:tensorflow:Using default config.
WARNING:tensorflow:Using default config.
WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
WARNING:tensorflow:Setting feature info to TensorSignature(dtype=tf.float64, shape=TensorShape([Dimensi
on(None), Dimension(4)]), is_sparse=False)
WARNING:tensorflow:Setting feature info to TensorSignature(dtype=tf.float64, shape=TensorShape([Dimensi
on(None), Dimension(4)]), is_sparse=False)
WARNING:tensorflow:Setting targets info to TensorSignature(dtype=tf.int64, shape=TensorShape([Dimension
(None)]), is_sparse=False)
WARNING:tensorflow:Setting targets info to TensorSignature(dtype=tf.int64, shape=TensorShape([Dimension
(None)]), is_sparse=False)
WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
```

```
Accuracy: 0.946667
```

Part 2. Deep Reinforcement Learing

Deep Reinforcement Learning이란, 기존의 강화학습에서의 Q function을 딥러닝으로 근사하는 모델을 의미한다. 대표적으로 구글 Deep Mind의 Atari와 AlphaGo 역시 이 Deep Reinforcement Learning 응용의 한가지이다.

이번 파트에서는 Deep Reinforcement Learning을 이용해서 간단한 2차원 게임을 플레이하고, reward로 부터 스스로 학습하는 Kaparthy의 오픈소스 예제를 실습해 본다.



(출처: https://github.com/nivwusquorum/tensorflow-deepq)

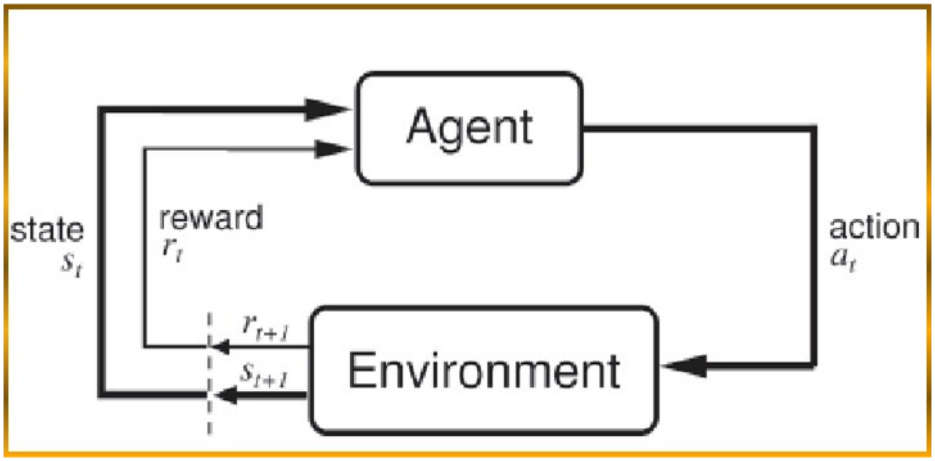
Reinforcement Learning

Reinforcement Learning, 이하 RL은 supervised learning과 달리 데이터에 대한 정확한 정답을 받지 않고, 내가 한 행동에 대한 reward feedback 만으로 학습을 수행하는 알고리즘이다. 이를 강화학습이라 부르며, 이것을 수행하는 가장 대표적인 알고리즘으로 Q-Learning 이 있다.

RL을 이해하는 것은 매우 많은 공부를 필요로 하기 때문에, 우선 2D게임을 예로 들어 아주 기본적인 개념만 살펴본다.

- **State**: 게임에서의 각 물체들의 위치, 속도, 벽과의 거리 등을 의미한다.
- **Action**: 게임을 플레이하는 주인공의 행동을 의미한다. 여기서는 4가지 방향에 대한 움직임이 이에 해당한다.
- **Reward**: 게임을 플레이하면서 받는 score. 여기서는 초록색을 먹으면 +1 , 빨간색을 먹으면 -1을 점수로 받는다.
- **Value**: 해당 action 또는 state가 미래에 얼마나 큰 reward를 가져올 지에 대한 기댓값.
- **Policy**: 주인공이 현재의 게임 State에서 Reward를 최대한 얻기 위해 Action을 선택하는 전략. 한마디로 [게임을 플레이하는 방법]을 의미한다.

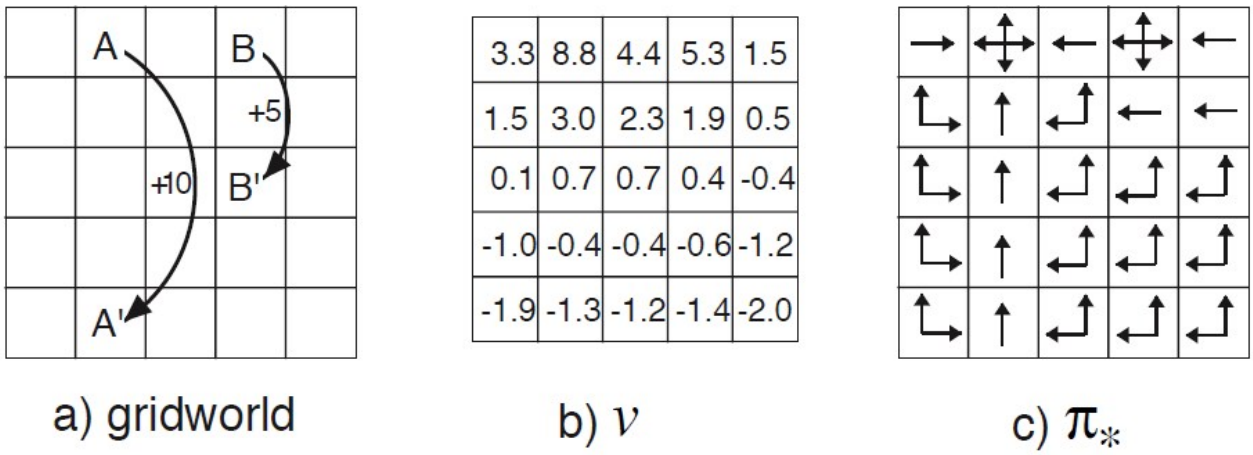
Environment Interaction(or Perception-Action Cycle) in Reinforcement Learning



(출처 : Sutton, 1998, Reinforcement Learning: An Introduction)

강화학습 학습이 다른 머신러닝 방법(supervised learning, unsupervised learning)과 다른 가장 큰 특징 중 하나는, 바로 환경과의 실시간 인터렉션을 가정한 학습 모델이라는 점이다. 강화학습은 생물체의 학습과정을 수학적으로 모델링하여 구현된 알고리즘이기 때문에 다른 방법들에 비해 보다 사람 혹은 동물의 학습과 유사하다.

Grid World



(출처 : Sutton, 1998, Reinforcement Learning: An Introduction)
State : 5x5
Action : 4방향이동
Reward : A에 도착하면 +10, B에 도착하면 +5, 벽에 부딪히면 -1, 그이외 0
Discounted Factor : 0.9

Discounted Factor

- Sum of future rewards (in episodic task)

➔ Gt := Rt+1 + Rt+2 + Rt+3 + … + RT

- Sum of future rewards (in continuous task)

➔ Gt := Rt+1 + Rt+2 + Rt+3 + … + RT + …
Gt → ∞

- Sum of discounted future rewards (in both case)

➔ $$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

(γ : discount rate , 0 <= γ < 1)

Q table

$$\hat{Q} = \begin{array}{c} state \backslash action \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} - & - & - & - & 80 & - \\ - & - & - & 64 & - & 100 \\ - & - & - & 64 & - & - \\ - & 80 & 51 & - & 80 & - \\ 64 & - & - & 64 & - & 100 \\ - & 80 & - & - & 80 & 100 \end{bmatrix} \end{array}$$

(출처 : http://gruposagama.com/pages/q-learning.html)

Q Learning Algorithm

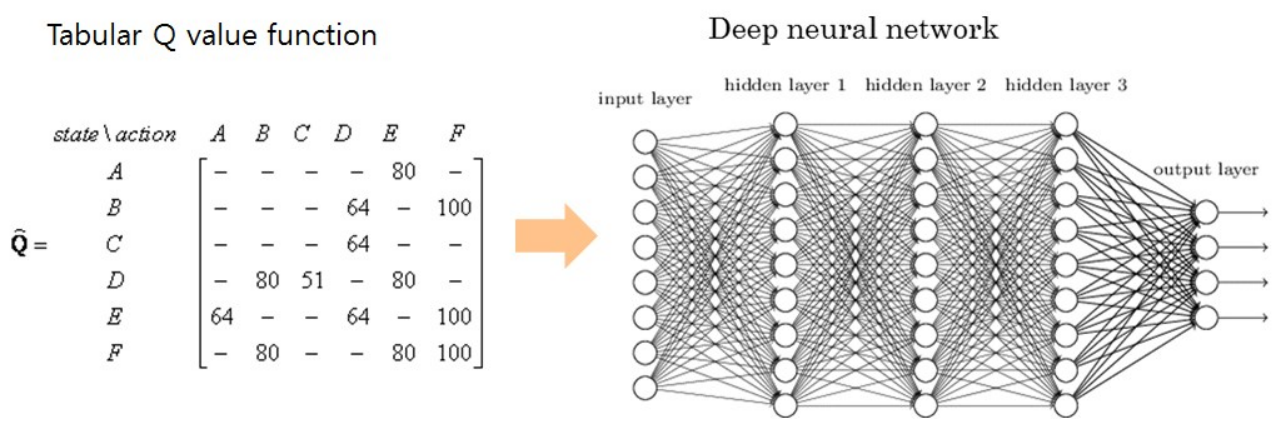
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until s is terminal

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

The New Action Value = The Old Value + The Learning Rate × (The New Information − The Old Information)

(출처 : http://www.randomant.net/wp-content/uploads/2016/05/q_learning3.jpg)
(출처 : http://people.revoledu.com/kardi/tutorial/ReinforcementLearning/Q-Learning-Example.htm)

Deep Reinforcement Learning : Q function -> Deep Learninig



$\hat{Q}$ =

state \ action	A	B	C	D	E	F
A	–	–	–	–	80	–
B	–	–	–	64	–	100
C	–	–	–	64	–	–
D	–	80	51	–	80	–
E	64	–	–	64	–	100
F	–	80	–	–	80	100

Example of Deep Reinforcement Learning

아래의 예제 코드를 실행시키기 위해서는 리눅스 shell에서 다음의 명령어를 실행해 필요한 python package를 설치해야한다.

 # pip install future euclid redis

In [ ]:
```
%matplotlib inline
import tensorflow as tf
tf.reset_default_graph()

from tf_rl.controller import DiscreteDeepQ, HumanController
from tf_rl.simulation import KarpathyGame
from tf_rl import simulate
from tf_rl.models import MLP

from __future__ import print_function
```

Environment Settings

이제 우리가 원하는 게임 환경을 설정하고, 적절한 reward와 object의 개수 및 observation 을 조절한다

In [2]:
```
current_settings = {
    'objects': [
        'friend',
        'enemy',
    ],
    'colors': {
        'hero':   'yellow',
        'friend': 'green',
        'enemy':  'red',
    },
    'object_reward': {
        'friend': 1,
        'enemy': -1,
    },
    "num_objects": {
        'friend' : 25,
        'enemy' :  25,
    },

    'hero_bounces_off_walls': False,
    'world_size': (700,500),
    'hero_initial_position': [400, 300],
    'hero_initial_speed':    [0,   0],
    "maximum_speed":         [50, 50],
    "object_radius": 10.0,
    "num_observation_lines" : 32, # the number of antennas
    "observation_line_length": 240., # the length of antennas
    "tolerable_distance_to_wall": 50,
    "wall_distance_penalty":  -0.0, # if the hero is close to wall, that receives penalty
    "delta_v": 50 # speed value
}

# create the game simulator
g = KarpathyGame(current_settings)
```

Deep Learning Architecture

이제 Q function을 근사하기 위한 딥러닝 모델을 만들어보자. 이번 예제에서는 위에서 보았던 4층짜리 MLP를 사용한다.

In [3]:
```
session = tf.InteractiveSession(config=tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth =True)))



journalist = tf.train.SummaryWriter("/tmp/drl")

# Brain maps from observation to Q values for different actions.
# Here it is a done using a multi layer perceptron with 2 hidden layers
brain = MLP([g.observation_size,], [200, 200, g.num_actions],  [tf.tanh, tf.tanh, tf.identity])
```

Make an Agent

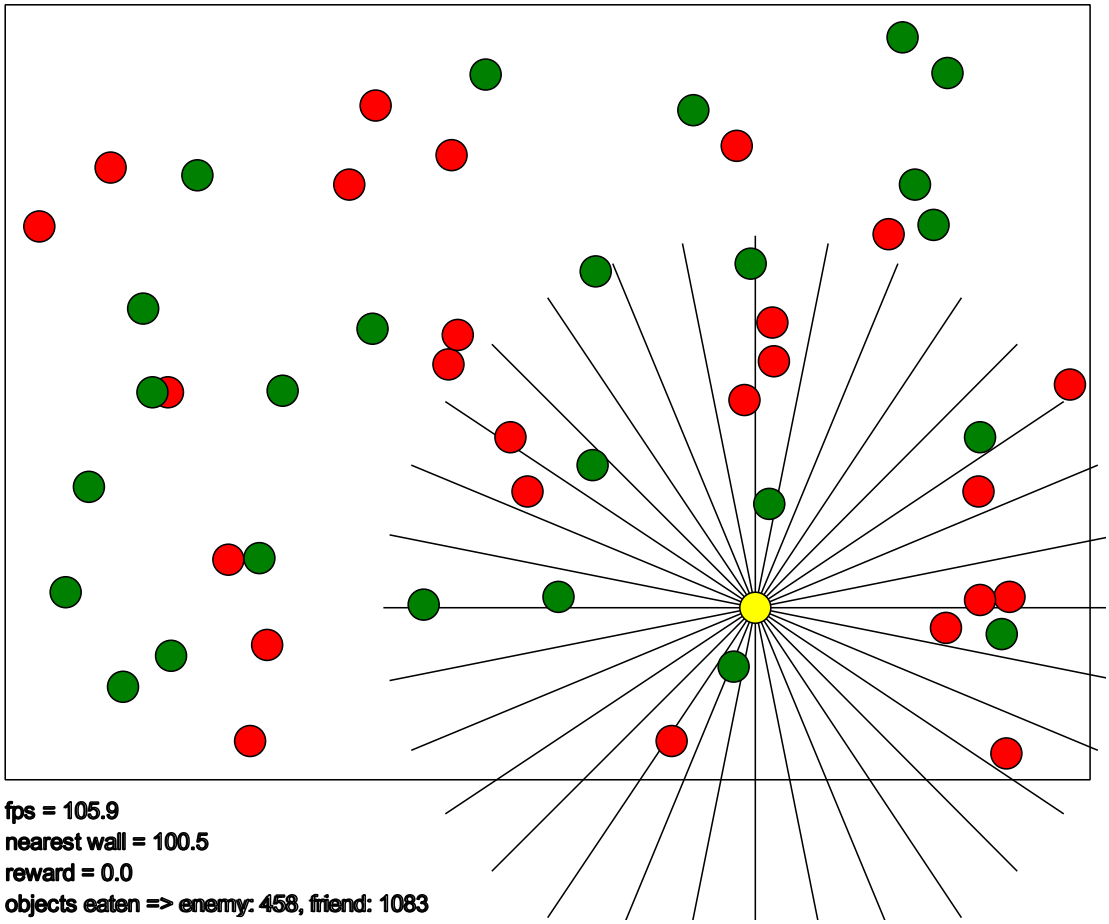
이제 Discrete Deep Q learning 알고리즘이 이 게임을 플레이하면서 학습을 하도록 agent로 설정을 한다.

In [4]:
```
# The optimizer to use. Here we use RMSProp as recommended by the publication
optimizer = tf.train.RMSPropOptimizer(learning_rate= 0.001, decay=0.9)

# DiscreteDeepQ object
current_controller = DiscreteDeepQ(g.observation_size, g.num_actions, brain, optimizer, session,
                                   discount_rate=0.99, exploration_period=5000, max_experience=10000,
                                   store_every_nth=4, train_every_nth=4,
                                   summary_writer=journalist)

session.run(tf.initialize_all_variables())
session.run(current_controller.target_network_update)
#journalist.add_graph(session.graph_def)
```

Play the Game

실제로 게임을 플레이하면서 강화학습이 일어나는 과정을 지켜보자.

In [5]:
```
FPS          = 30
ACTION_EVERY = 3

fast_mode = True
if fast_mode:
    WAIT, VISUALIZE_EVERY = False, 20
else:
    WAIT, VISUALIZE_EVERY = True, 1


try:
    with tf.device("/gpu:0"):
        simulate(simulation=g,
                 controller=current_controller,
                 fps=FPS,
                 visualize_every=VISUALIZE_EVERY,
                 action_every=ACTION_EVERY,
                 wait=WAIT,
                 disable_training=False,
                 simulation_resolution=0.001,
                 save_path=None)
except KeyboardInterrupt:
    print("Interrupted")


#session.close()
```




```
Interrupted
```

Applications

- Parameter 들을 바꾸어 enemy와 friend의 개수 차이가 최대한 많이 나도록 agent를 학습시켜본다.
- Boss object를 추가해본다.
- Deep RL에서의 tensorboard를 열어서 visualize를 해본다.

* Open-source TensorFlow Implementation

아래 링크는 TensorFlow로 구현되어 공개된 여러 오픈소스 프로젝트들을 모아서 정리해 둔 페이지들이다. 이중 본인의 연구 분야와 관련있는 프로젝트를 clone, 수정하여 사용할 경우 개발시간을 크게 단축할 수 있다.

https://github.com/tensorflow/models : Syntax Net, Magenta, Image2Txt
https://github.com/TensorFlowKR/awesome_tensorflow_implementations
https://github.com/aikorea/awesome-rl

유명한 오픈소스 몇가지를 살펴보자.