CMPT 419/983 Fall 2019 Reinforcement Learning with Tensorflow Tutorial

This tutorial will provide a brief overview of the core concepts and functionality of Tensorflow. This tutorial will cover the following:

Part 1: Tensorflow

1. What is Tensorflow
2. How to input data
3. How to perform computations
4. How to create variables
5. How to train a neural network
6. Tips and tricks

Part 2: OpenAI

1. Introduction to OpenAI
2. Agent and Environment
3. Q-Learning
4. Deep Q Learning using OpenAI and TensorFlow

Part 1: Tensorflow

Installation: To install Tensorflow, please refer to https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html (https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html)

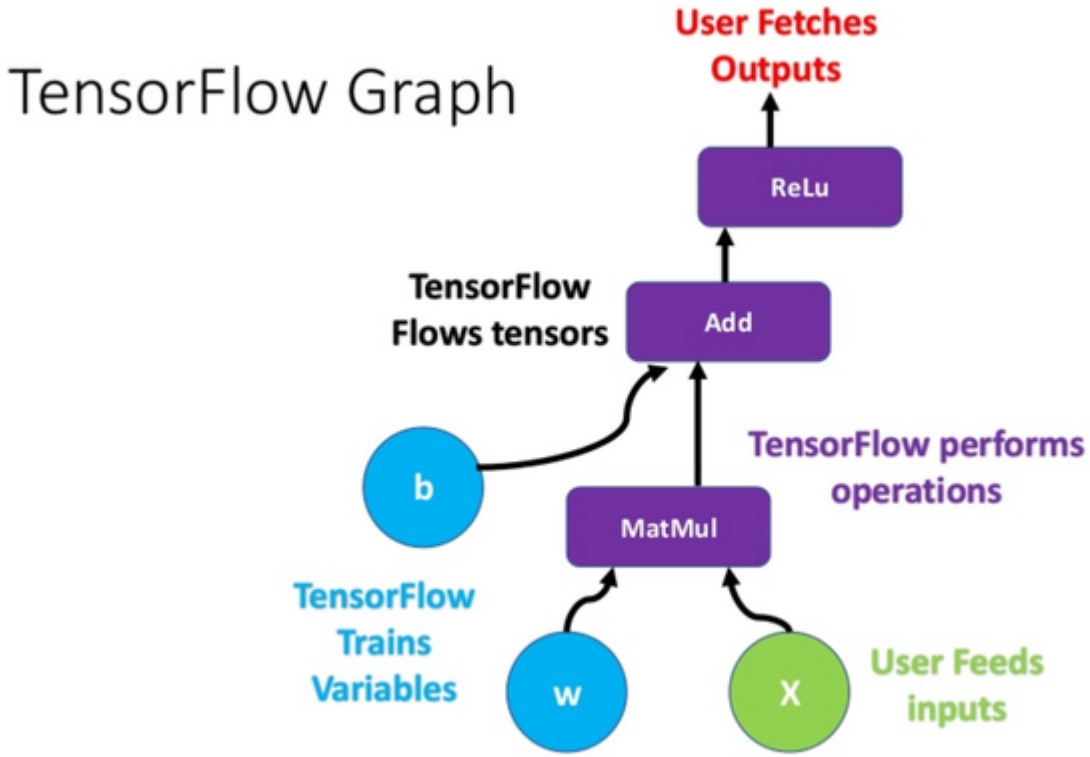
If you have a CUDA enabled GPU in your system, I highly recommend that you install the GPU version from the above website. Note: For this sepcific notebook and assignment 3, installing the GPU version is not required and you can install the CPU version.

In [46]:
```
# importing libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
# import matplotlib.cm as cm
# import matplotlib.patches as mpatches
```

In [2]:
```
# resetting tensorflow computation graph - Just housekeeping stuff
def tf_reset():
    try:
        sess.close()
    except:
        pass
    tf.reset_default_graph()
    return tf.Session()
```

1. What is Tensorflow

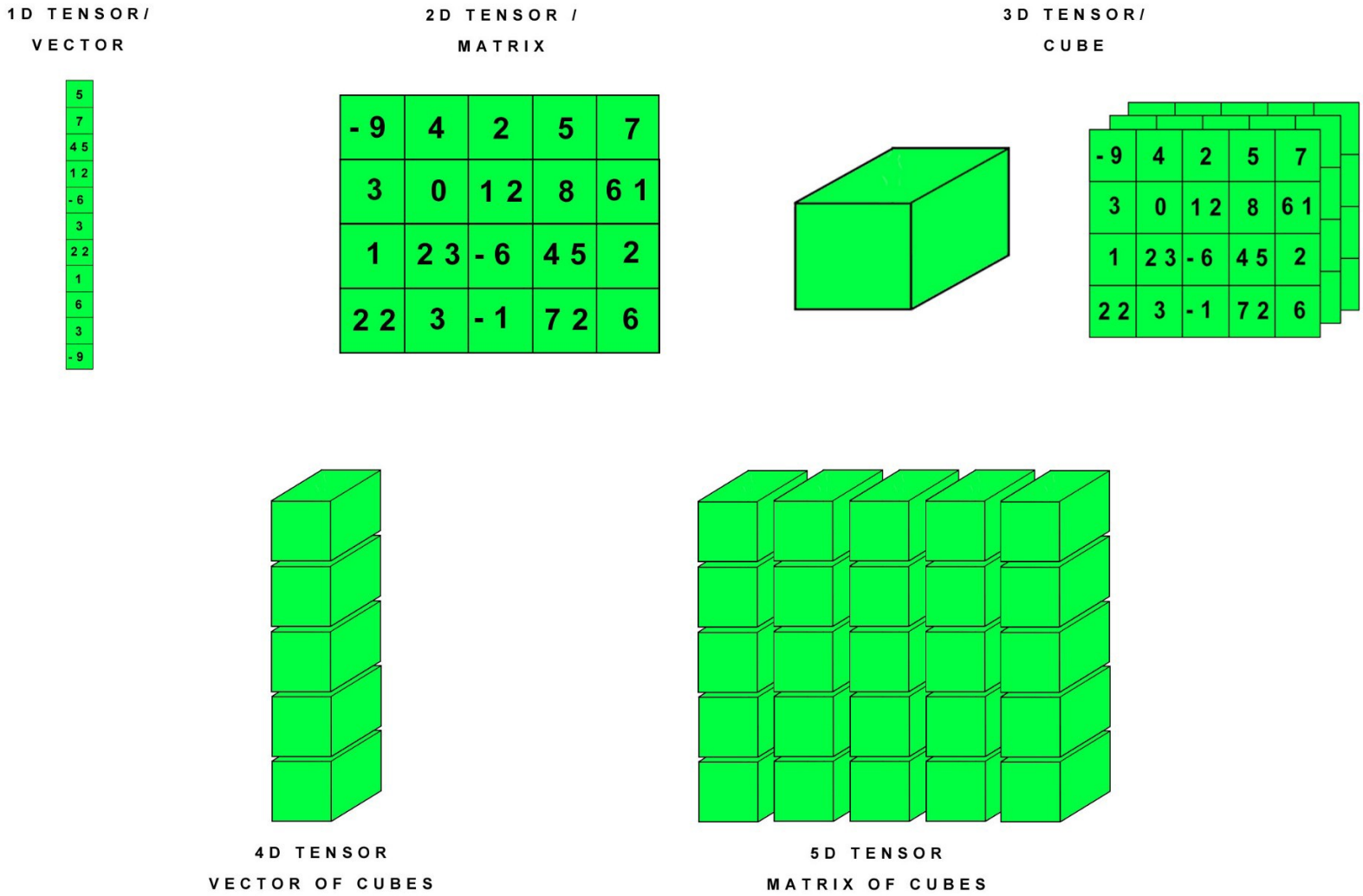
Tensorflow is a framework to define a series of computations. You define inputs, what operations should be performed, and then Tensorflow will compute the outputs for you.



Big idea : Tensor + Flow; Express a numeric computation as a graph

- Graph nodes are operations which have any number of inputs and outputs
- Grapgh egdes are tensors which flow between nodes

Let's see what a Tensor is:



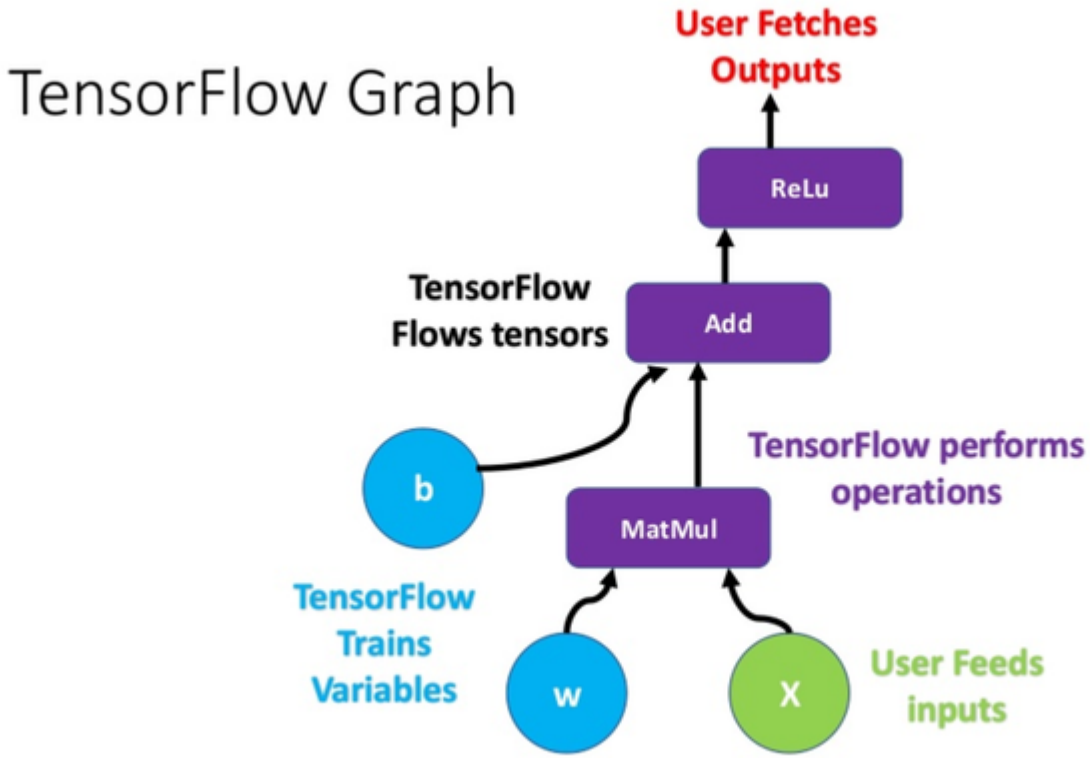
Below is a simple high-level example of a tensorflow graph for h = ReLU(Wx + b) :

In [3]:
```
# h = ReLU(Wx + b)

b = tf.Variable(tf.zeros((100,))) # all zeros
W = tf.Variable(tf.random_uniform((784, 100), -1, 1)) # W ~ Uniform(-1,1)

x = tf.placeholder(tf.float32, (100, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```



2. How to input data

Tensorflow has multiple ways for you to input data.

- One way is to have the inputs be constants: As the name suggests, the value stored in constants can't be changes later.
- Variables are stateful nodes which output their current value. For the parameters in a network, always use Variables.
- Placeholders are nodes whose value is fed in at the execution time. Training data to any network is inserted at execution time so is stored in placeholders.

In [4]:
```
# create the session you'll work in
# you can think of this as a "blank piece of paper" that you'll be writing math on
sess = tf_reset()

# define your inputs
a = tf.constant(1.0)
b = tf.constant(2.0)

# do some operations
c = a + b

# get the result
c_run = sess.run(c)

print('c = {0}'.format(c_run))
```

```
c = 3.0
```

In [5]:
```
# Let's see the nodes in computation graph for the above example
[n.name for n in tf.get_default_graph().as_graph_def().node]
```

Out[5]:
```
['Const', 'Const_1', 'add']
```

As above, having our inputs be constants is inflexible. We want to be able to change what data we input at runtime. We do this using placeholders:

In [6]:
```
sess = tf_reset()

# define your inputs
a = tf.placeholder(dtype=tf.float32, shape=[1], name='a_placeholder')
b = tf.placeholder(dtype=tf.float32, shape=[1], name='b_placeholder')

# do some operations
c = a + b

# get the result
c0_run = sess.run(c, feed_dict={a: [1.0], b: [2.0]})
c1_run = sess.run(c, feed_dict={a: [2.0], b: [4.0]})

print('c0 = {0}'.format(c0_run))
print('c1 = {0}'.format(c1_run))
```

```
c0 = [3.]
c1 = [6.]
```

But what if we don't know the size of our input beforehand? One dimension of a tensor is allowed to be 'None', which means it can be variable sized:

In [7]:
```
sess = tf_reset()

# inputs
a = tf.placeholder(dtype=tf.float32, shape=[None], name='a_placeholder')
b = tf.placeholder(dtype=tf.float32, shape=[None], name='b_placeholder')

# do some operations
c = a + b

# get outputs
c0_run = sess.run(c, feed_dict={a: [1.0], b: [2.0]})
c1_run = sess.run(c, feed_dict={a: [1.0, 2.0], b: [2.0, 4.0]})

print(a)
print('a shape: {0}'.format(a.get_shape()))
print(b)
print('b shape: {0}'.format(b.get_shape()))
print('c0 = {0}'.format(c0_run))
print('c1 = {0}'.format(c1_run))
```

```
Tensor("a_placeholder:0", shape=(?,), dtype=float32)
a shape: (?,)
Tensor("b_placeholder:0", shape=(?,), dtype=float32)
b shape: (?,)
c0 = [3.]
c1 = [3. 6.]
```

3. How to perform computations

Now that we can input data, we want to perform useful computations on the data.

First, let's create some data to work with:

In [8]:
```
sess = tf_reset()

# inputs
a = tf.constant([[-1.], [-2.], [-3.]], dtype=tf.float32)
b = tf.constant([[1., 2., 3.]], dtype=tf.float32)

a_run, b_run = sess.run([a, b])
print('a:\n{0}'.format(a_run))
print('b:\n{0}'.format(b_run))
```

```
a:
[[-1.]
 [-2.]
 [-3.]]
b:
[[1. 2. 3.]]
```

We can do simple operations, such as addition:

In [9]:
```
c = b + b

c_run = sess.run(c)
print('b:\n{0}'.format(b_run))
print('c:\n{0}'.format(c_run))
```

```
b:
[[1. 2. 3.]]
c:
[[2. 4. 6.]]
```

Be careful about the dimensions of the tensors, some operations may work even when you think they shouldn't...

In [10]:
```
c = a + b

c_run = sess.run(c)
print('a:\n{0}'.format(a_run))
print('b:\n{0}'.format(b_run))
print('c:\n{0}'.format(c_run))
```

```
a:
[[-1.]
 [-2.]
 [-3.]]
b:
[[1. 2. 3.]]
c:
[[ 0.  1.  2.]
 [-1.  0.  1.]
 [-2. -1.  0.]]
```

Also, some operations may be different than what you expect:

In [11]:
```
c_elementwise = a * b
c_matmul = tf.matmul(b, a)

c_elementwise_run, c_matmul_run = sess.run([c_elementwise, c_matmul])
print('a:\n{0}'.format(a_run))
print('b:\n{0}'.format(b_run))
print('c_elementwise:\n{0}'.format(c_elementwise_run))
print('c_matmul: \n{0}'.format(c_matmul_run))
```

```
a:
[[-1.]
 [-2.]
 [-3.]]
b:
[[1. 2. 3.]]
c_elementwise:
[[-1. -2. -3.]
 [-2. -4. -6.]
 [-3. -6. -9.]]
c_matmul:
[[-14.]]
```

Operations can be chained together:

In [12]:
```
# operations can be chained together
c0 = b + b
c1 = c0 + 1

c0_run, c1_run = sess.run([c0, c1])
print('b:\n{0}'.format(b_run))
print('c0:\n{0}'.format(c0_run))
print('c1:\n{0}'.format(c1_run))
```

```
b:
[[1. 2. 3.]]
c0:
[[2. 4. 6.]]
c1:
[[3. 5. 7.]]
```

Finally, Tensorflow has many useful built-in operations:

In [13]:
```
c = tf.reduce_mean(b)

c_run = sess.run(c)
print('b:\n{0}'.format(b_run))
print('c:\n{0}'.format(c_run))
```

```
b:
[[1. 2. 3.]]
c:
2.0
```

4. How to create variables

Now that we can input data and perform computations, we want some of these operations to involve variables that are free parameters, and can be trained using an optimizer (e.g., gradient descent).

First, let's create some data to work with:

In [14]:
```
sess = tf_reset()

# inputs
b = tf.constant([[1., 2., 3.]], dtype=tf.float32)

sess = tf.Session()

b_run = sess.run(b)
print('b:\n{0}'.format(b_run))
```

```
b:
[[1. 2. 3.]]
```

We'll now create a variable

In [15]:
```
var_init_value = [[2.0, 4.0, 6.0]]
var = tf.get_variable(name='myvar',
                      shape=[1, 3],
                      dtype=tf.float32,
                      initializer=tf.constant_initializer(var_init_value))

print(var)
```

```
<tf.Variable 'myvar:0' shape=(1, 3) dtype=float32_ref>
```

and check that it's been added to Tensorflow's variables list:

In [16]:
```
print(tf.global_variables())
```

```
[<tf.Variable 'myvar:0' shape=(1, 3) dtype=float32_ref>]
```

We can do operations with the variable just like any other tensor:

In [17]:
```
# can do operations
c = b + var
print(b)
print(var)
print(c)
```

```
Tensor("Const:0", shape=(1, 3), dtype=float32)
<tf.Variable 'myvar:0' shape=(1, 3) dtype=float32_ref>
Tensor("add:0", shape=(1, 3), dtype=float32)
```

Before we can run any of these operations, we must first initalize the variables

In [18]:
```
init_op = tf.global_variables_initializer()
sess.run(init_op)
```

and then we can run the operations just as we normally would.

In [19]:
```
c_run = sess.run(c)

print('b:\n{0}'.format(b_run))
print('var:\n{0}'.format(var_init_value))
print('c:\n{0}'.format(c_run))
```

```
b:
[[1. 2. 3.]]
var:
[[2.0, 4.0, 6.0]]
c:
[[3. 6. 9.]]
```

So far we haven't said yet how to optimize these variables. We'll cover that next in the context of an example.

5. How to train a simple neural network for regression probelem and a Multi Layer Perceptron neural network for MNIST classification

We've discussed how to input data, perform operations, and create variables. We'll now show how to combine all of these---with some minor additions---to train a neural network on a simple regression problem.

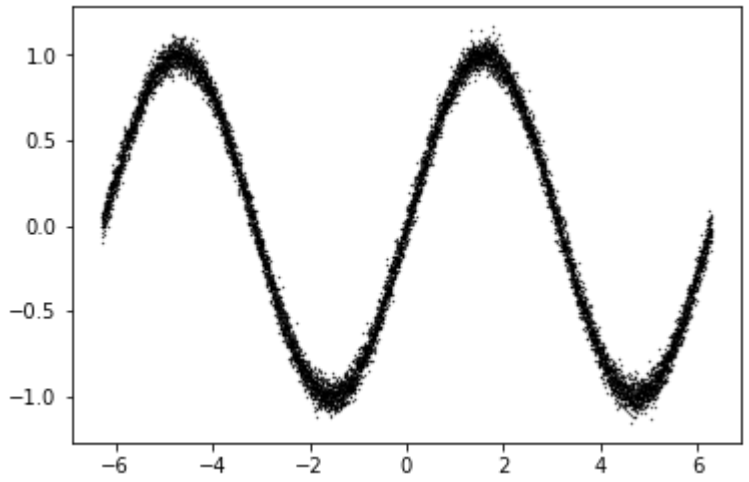
First, we'll create data for the 1-dimensional regression problem:

In [20]:
```
%matplotlib inline

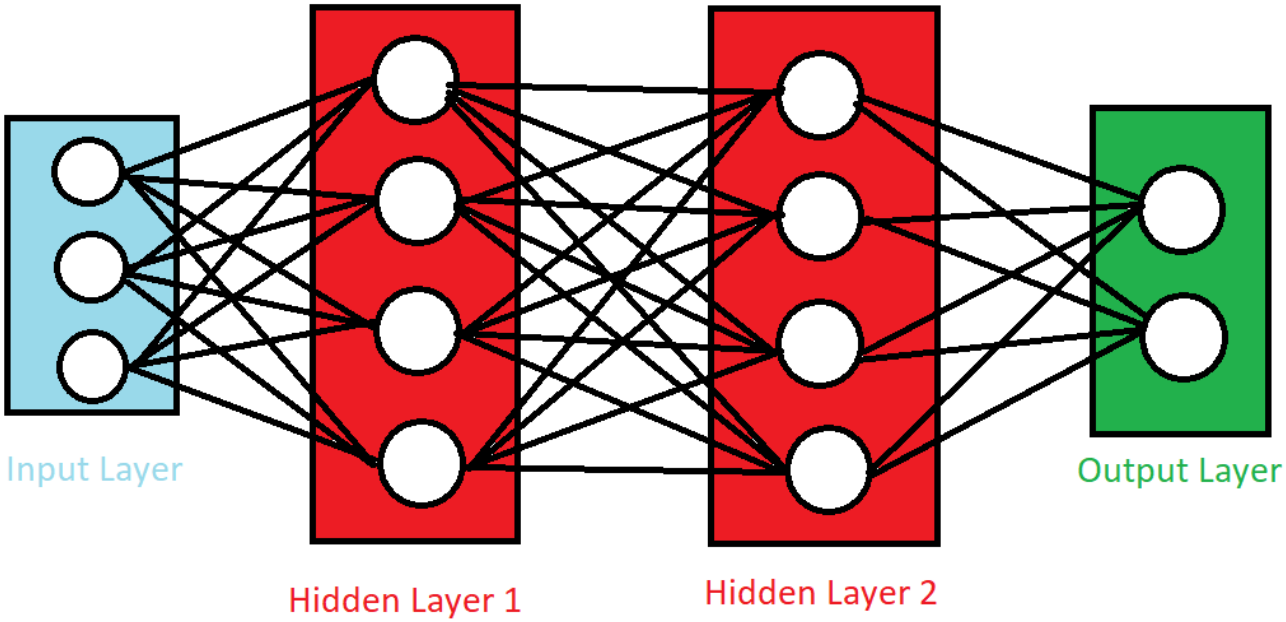
# generate the data
inputs = np.linspace(-2*np.pi, 2*np.pi, 10000)[:, None]
outputs = np.sin(inputs) + 0.05 * np.random.normal(size=[len(inputs),1])

plt.scatter(inputs[:, 0], outputs[:, 0], s=0.1, color='k', marker='o')
```

Out[20]: <matplotlib.collections.PathCollection at 0x26f0f492be0>



But first, let's see what a mulitlayer perceptron looks like



The below code creates the inputs, variables, neural network operations, mean-squared-error loss, gradient descent optimizer, and runs the optimizer using minibatches of the data.

In [21]:
```
sess = tf_reset()

def create_model():
    # create inputs
    input_ph = tf.placeholder(dtype=tf.float32, shape=[None, 1])
    output_ph = tf.placeholder(dtype=tf.float32, shape=[None, 1])

    W0 = tf.Variable(tf.random_normal([1, 20]))
    W1 = tf.Variable(tf.random_normal([20, 20]))
    W2 = tf.Variable(tf.random_normal([20, 1]))

    b0 = tf.Variable(tf.random_normal([20]))
    b1 = tf.Variable(tf.random_normal([20]))
    b2 = tf.Variable(tf.random_normal([1]))

    # create computation graph
    output_pred = tf.add(tf.matmul(input_ph, W0), b0)
    output_pred = tf.nn.relu(output_pred)

    output_pred = tf.add(tf.matmul(output_pred, W1), b1)
    output_pred = tf.nn.relu(output_pred)

    output_pred = tf.add(tf.matmul(output_pred, W2), b2)

    return input_ph, output_ph, output_pred

input_ph, output_ph, output_pred = create_model()

# create loss
mse = tf.reduce_mean(0.5 * tf.square(output_pred - output_ph))

# create optimizer
opt = tf.train.AdamOptimizer().minimize(mse)

# initialize variables
sess.run(tf.global_variables_initializer())
# create saver to save model variables
saver = tf.train.Saver()

# run training
batch_size = 1024
for training_step in range(10000):
```

```
    # get a random subset of the training data
    indices = np.random.randint(low=0, high=len(inputs), size=batch_size)
    input_batch = inputs[indices]
    output_batch = outputs[indices]

    # run the optimizer and get the mse
    _, mse_run = sess.run([opt, mse], feed_dict={input_ph: input_batch, output_ph: output_batch})

    # print the mse every so often
    if training_step % 1000 == 0:
        print('{0:04d} mse: {1:.3f}'.format(training_step, mse_run))
        saver.save(sess, './tmp/model.ckpt')
```

```
0000 mse: 190.343
1000 mse: 0.082
2000 mse: 0.044
3000 mse: 0.028
4000 mse: 0.019
5000 mse: 0.010
6000 mse: 0.006
7000 mse: 0.003
8000 mse: 0.003
9000 mse: 0.002
```

Now that the neural network is trained, we can use it to make predictions:

In [22]:
```
%matplotlib inline

sess = tf_reset()

# create the model
input_ph, output_ph, output_pred = create_model()

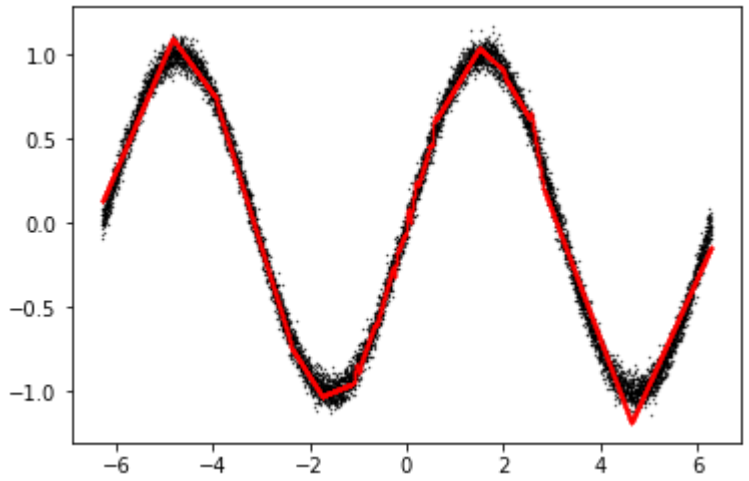
# restore the saved model
saver = tf.train.Saver()
saver.restore(sess, "./tmp/model.ckpt")

output_pred_run = sess.run(output_pred, feed_dict={input_ph: inputs})

plt.scatter(inputs[:, 0], outputs[:, 0], c='k', marker='o', s=0.1)
plt.scatter(inputs[:, 0], output_pred_run[:, 0], c='r', marker='o', s=0.1)
```

```
INFO:tensorflow:Restoring parameters from ./tmp/model.ckpt
```

Out[22]:
```
<matplotlib.collections.PathCollection at 0x26f106d59b0>
```



In [23]:
```
""" Multilayer Perceptron.

A Multilayer Perceptron (Neural Network) implementation example using
TensorFlow library. This example is using the MNIST database of handwritten
digits (http://yann.lecun.com/exdb/mnist/).

Links:
    [MNIST Dataset](http://yann.lecun.com/exdb/mnist/).

Author: Aymeric Damien
Project: https://github.com/aymericdamien/TensorFlow-Examples/
"""

from __future__ import print_function
import tensorflow as tf

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

```
WARNING:tensorflow:From <ipython-input-23-d18731ab1cbf>:19: read_data_sets (from tensorflow.contrib.learn.pytho
n.learn.datasets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorflow/models.
WARNING:tensorflow:From C:\Users\Shubam Sachdeva\AppData\Local\Continuum\Anaconda2\envs\tensorflow\lib\site-pac
kages\tensorflow\contrib\learn\python\learn\datasets\mnist.py:260: maybe_download (from tensorflow.contrib.lear
n.python.learn.datasets.base) is deprecated and will be removed in a future version.
Instructions for updating:
Please write your own downloading logic.
WARNING:tensorflow:From C:\Users\Shubam Sachdeva\AppData\Local\Continuum\Anaconda2\envs\tensorflow\lib\site-pac
kages\tensorflow\contrib\learn\python\learn\datasets\mnist.py:262: extract_images (from tensorflow.contrib.lear
n.python.learn.datasets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting /tmp/data/train-images-idx3-ubyte.gz
WARNING:tensorflow:From C:\Users\Shubam Sachdeva\AppData\Local\Continuum\Anaconda2\envs\tensorflow\lib\site-pac
kages\tensorflow\contrib\learn\python\learn\datasets\mnist.py:267: extract_labels (from tensorflow.contrib.lear
n.python.learn.datasets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
WARNING:tensorflow:From C:\Users\Shubam Sachdeva\AppData\Local\Continuum\Anaconda2\envs\tensorflow\lib\site-pac
```

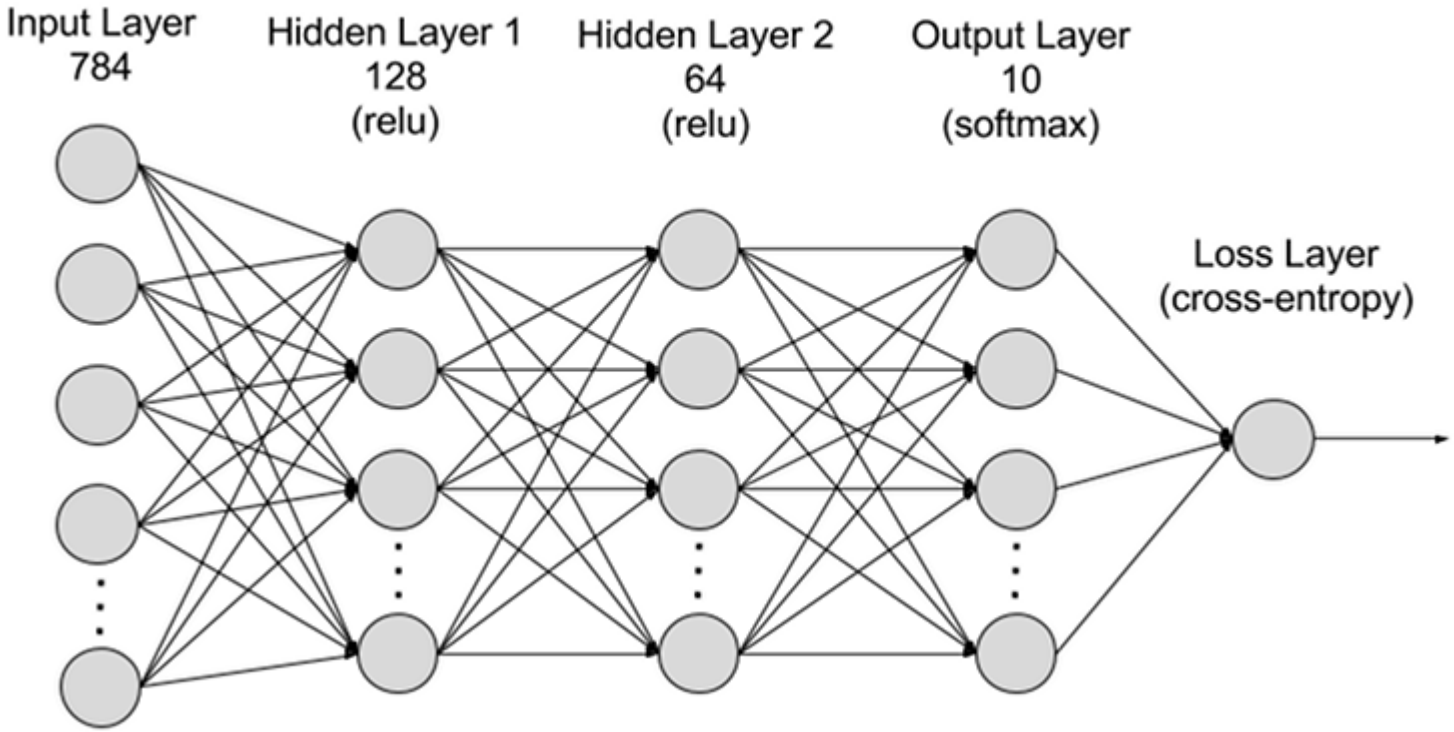
```
kages\tensorflow\contrib\learn\python\learn\datasets\mnist.py:110: dense_to_one_hot (from tensorflow.contrib.le
arn.python.learn.datasets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.one_hot on tensors.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
WARNING:tensorflow:From C:\Users\Shubam Sachdeva\AppData\Local\Continuum\Anaconda2\envs\tensorflow\lib\site-pac
kages\tensorflow\contrib\learn\python\learn\datasets\mnist.py:290: DataSet.__init__ (from tensorflow.contrib.le
arn.python.learn.datasets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorflow/models.
```

In [24]:
```
# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1

# Network Parameters
n_hidden_1 = 128 # 1st layer number of neurons
n_hidden_2 = 64 # 2nd layer number of neurons
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
```

In [25]:
```
# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])
```

In [26]:
```
# 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 [27]:
```
# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
logits = multilayer_perceptron(X)
```

In [28]:
```
# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
# Initializing the variables
init = tf.global_variables_initializer()
```

```
WARNING:tensorflow:From <ipython-input-28-6ca114f21025>:3: softmax_cross_entropy_with_logits (from tensorflow.p
ython.ops.nn_ops) is deprecated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See @{tf.nn.softmax_cross_entropy_with_logits_v2}.
```

In [29]:
```
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_x, batch_y = mnist.train.next_batch(batch_size)
            # Run optimization op (backprop) and cost op (to get loss value)
            _, c = sess.run([train_op, loss_op], feed_dict={X: batch_x,
                                                            Y: batch_y})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost={:.9f}".format(avg_cost))
    print("Optimization Finished!")

    # Test model
    pred = tf.nn.softmax(logits)  # Apply softmax to logits
    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.test.labels}))
```

```
Epoch: 0001 cost=157.289312099
Epoch: 0002 cost=43.143069897
Epoch: 0003 cost=30.213085677
Epoch: 0004 cost=23.335482633
Epoch: 0005 cost=19.076330198
Epoch: 0006 cost=16.087118572
Epoch: 0007 cost=13.692821933
Epoch: 0008 cost=12.057206927
Epoch: 0009 cost=10.625388635
Epoch: 0010 cost=9.445129234
Epoch: 0011 cost=8.607318865
Epoch: 0012 cost=7.635662027
Epoch: 0013 cost=7.052506600
Epoch: 0014 cost=6.499331078
Epoch: 0015 cost=6.021332975
```

```
Optimization Finished!
Accuracy: 0.8834
```

Not so hard after all! There is much more functionality to Tensorflow besides what we've covered, but you now know the basics.

6. Tips and tricks

(a) Check your dimensions

In [30]:
```
# example of "surprising" resulting dimensions due to broadcasting
a = tf.constant(np.random.random((4, 1)))
b = tf.constant(np.random.random((1, 4)))
c = a * b
assert c.get_shape() == (4, 4)
```

(b) Check what variables have been created

In [31]:
```
sess = tf_reset()
a = tf.get_variable('I_am_a_variable', shape=[4, 6])
b = tf.get_variable('I_am_a_variable_too', shape=[2, 7])
for var in tf.global_variables():
    print(var.name)
```

```
I_am_a_variable:0
I_am_a_variable_too:0
```

(c) Look at the tensorflow API (https://www.tensorflow.org/api_docs/python/), or open up a python terminal and investigate!

In [32]:
```
help(tf.reduce_mean)
```

```
Help on function reduce_mean in module tensorflow.python.ops.math_ops:

reduce_mean(input_tensor, axis=None, keepdims=None, name=None, reduction_indices=None, keep_dims=None)
    Computes the mean of elements across dimensions of a tensor. (deprecated arguments)

    SOME ARGUMENTS ARE DEPRECATED. They will be removed in a future version.
    Instructions for updating:
    keep_dims is deprecated, use keepdims instead

    Reduces `input_tensor` along the dimensions given in `axis`.
    Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each
    entry in `axis`. If `keepdims` is true, the reduced dimensions
    are retained with length 1.

    If `axis` is None, all dimensions are reduced, and a
    tensor with a single element is returned.

    For example:

    ```python
    x = tf.constant([[1., 1.], [2., 2.]])
    tf.reduce_mean(x)  # 1.5
    tf.reduce_mean(x, 0)  # [1.5, 1.5]
    tf.reduce_mean(x, 1)  # [1.,  2.]
    ```

    Args:
      input_tensor: The tensor to reduce. Should have numeric type.
      axis: The dimensions to reduce. If `None` (the default),
        reduces all dimensions. Must be in the range
        `[-rank(input_tensor), rank(input_tensor))`.
      keepdims: If true, retains reduced dimensions with length 1.
      name: A name for the operation (optional).
      reduction_indices: The old (deprecated) name for axis.
      keep_dims: Deprecated alias for `keepdims`.

    Returns:
      The reduced tensor.

    @compatibility(numpy)
```

```
    Equivalent to np.mean

    Please note that `np.mean` has a `dtype` parameter that could be used to
    specify the output type. By default this is `dtype=float64`. On the other
    hand, `tf.reduce_mean` has an aggressive type inference from `input_tensor`,
    for example:

    ```python
    x = tf.constant([1, 0, 1, 0])
    tf.reduce_mean(x)  # 0
    y = tf.constant([1., 0., 1., 0.])
    tf.reduce_mean(y)  # 0.5
    ```

    @end_compatibility
```

(d) Tensorflow has some built-in layers to simplify your code.

In [33]:
```
help(tf.contrib.layers.fully_connected)
```

```
Help on function fully_connected in module tensorflow.contrib.layers.python.layers.layers:

fully_connected(inputs, num_outputs, activation_fn=<function relu at 0x0000026F095AB7B8>, normalizer_fn=Non
e, normalizer_params=None, weights_initializer=<function variance_scaling_initializer.<locals>._initializer
at 0x0000026F10A92048>, weights_regularizer=None, biases_initializer=<tensorflow.python.ops.init_ops.Zeros o
bject at 0x0000026F10A8E6D8>, biases_regularizer=None, reuse=None, variables_collections=None, outputs_colle
ctions=None, trainable=True, scope=None)
    Adds a fully connected layer.

    `fully_connected` creates a variable called `weights`, representing a fully
    connected weight matrix, which is multiplied by the `inputs` to produce a
    `Tensor` of hidden units. If a `normalizer_fn` is provided (such as
    `batch_norm`), it is then applied. Otherwise, if `normalizer_fn` is
    None and a `biases_initializer` is provided then a `biases` variable would be
    created and added the hidden units. Finally, if `activation_fn` is not `None`,
    it is applied to the hidden units as well.

    Note: that if `inputs` have a rank greater than 2, then `inputs` is flattened
    prior to the initial matrix multiply by `weights`.

    Args:
      inputs: A tensor of at least rank 2 and static value for the last dimension;
        i.e. `[batch_size, depth]`, `[None, None, None, channels]`.
      num_outputs: Integer or long, the number of output units in the layer.
      activation_fn: Activation function. The default value is a ReLU function.
        Explicitly set it to None to skip it and maintain a linear activation.
      normalizer_fn: Normalization function to use instead of `biases`. If
        `normalizer_fn` is provided then `biases_initializer` and
        `biases_regularizer` are ignored and `biases` are not created nor added.
        default set to None for no normalizer function
      normalizer_params: Normalization function parameters.
      weights_initializer: An initializer for the weights.
      weights_regularizer: Optional regularizer for the weights.
      biases_initializer: An initializer for the biases. If None skip biases.
      biases_regularizer: Optional regularizer for the biases.
      reuse: Whether or not the layer and its variables should be reused. To be
        able to reuse the layer scope must be given.
      variables_collections: Optional list of collections for all the variables or
        a dictionary containing a different list of collections per variable.
      outputs_collections: Collection to add the outputs.
```

```
    trainable: If `True` also add variables to the graph collection
      `GraphKeys.TRAINABLE_VARIABLES` (see tf.Variable).
    scope: Optional scope for variable_scope.

  Returns:
     The tensor variable representing the result of the series of operations.

  Raises:
    ValueError: If x has rank less than 2 or if its last dimension is not set.
```

(e) Use variable scope (https://www.tensorflow.org/guide/variables#sharing_variables) to keep your variables organized.

In [34]:
```
sess = tf_reset()

# create variables
with tf.variable_scope('layer_0'):
    W0 = tf.get_variable(name='W0', shape=[1, 20], initializer=tf.contrib.layers.xavier_initializer())
    b0 = tf.get_variable(name='b0', shape=[20], initializer=tf.constant_initializer(0.))

with tf.variable_scope('layer_1'):
    W1 = tf.get_variable(name='W1', shape=[20, 20], initializer=tf.contrib.layers.xavier_initializer())
    b1 = tf.get_variable(name='b1', shape=[20], initializer=tf.constant_initializer(0.))

with tf.variable_scope('layer_2'):
    W2 = tf.get_variable(name='W2', shape=[20, 1], initializer=tf.contrib.layers.xavier_initializer())
    b2 = tf.get_variable(name='b2', shape=[1], initializer=tf.constant_initializer(0.))

# print the variables
var_names = sorted([v.name for v in tf.global_variables()])
print('\n'.join(var_names))
```

```
layer_0/W0:0
layer_0/b0:0
layer_1/W1:0
layer_1/b1:0
layer_2/W2:0
layer_2/b2:0
```

(f) You can specify which GPU you want to use and how much memory you want to use

In [35]:
```
gpu_device = 0
gpu_frac = 0.5

# make only one of the GPUs visible
import os
os.environ["CUDA_VISIBLE_DEVICES"] = str(gpu_device)

# only use part of the GPU memory
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=gpu_frac)
config = tf.ConfigProto(gpu_options=gpu_options)

# create the session
tf_sess = tf.Session(graph=tf.Graph(), config=config)
```

(g) You can use tensorboard (https://www.tensorflow.org/guide/summaries_and_tensorboard) to visualize and monitor the training process.

In [ ]:
```
```

Part 2: OpenAI Gym

1. Introduction to OpenAI Gym

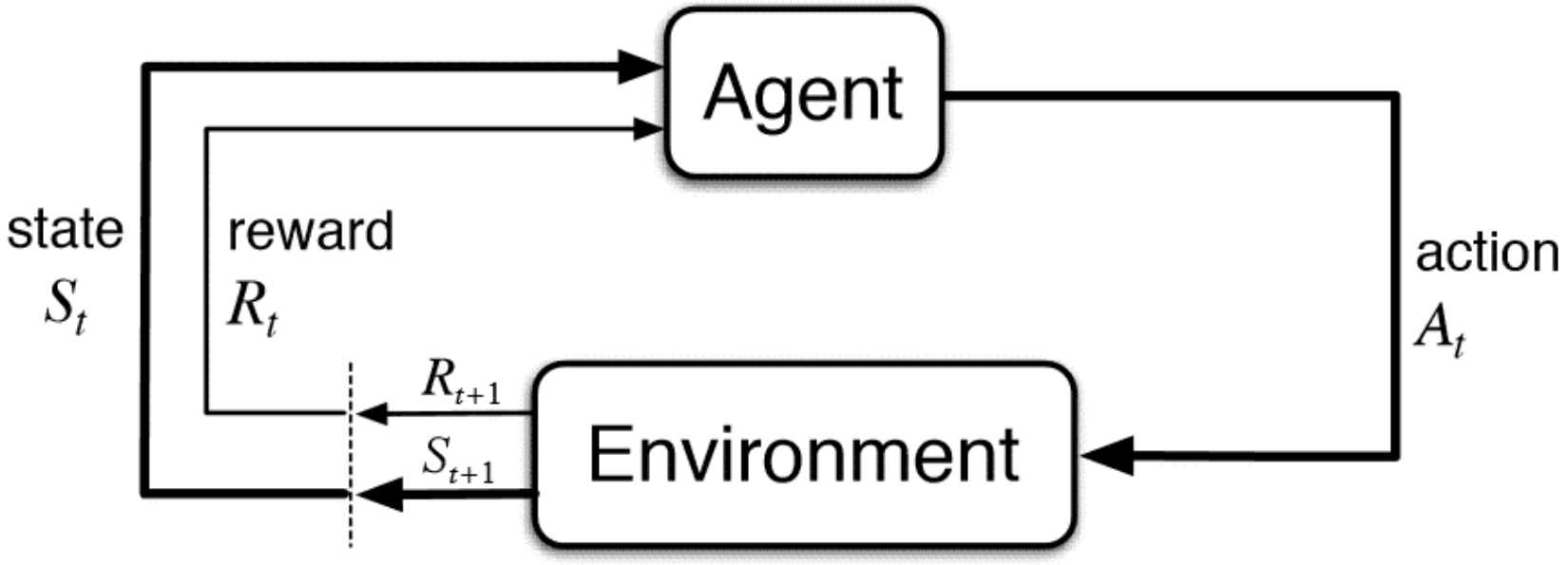
OpenAI is the for-profit corporation which conducts research in the field of artificial intelligence (AI) with the stated aim to promote and develop friendly AI in such a way as to benefit humanity as a whole.

- Founded in late 2015, the San Francisco-based organization aims to “freely collaborate” with other institutions and researchers by making its patents and research open to the public. The founders (notably Elon Musk and Sam Altman) are motivated in part by concerns about the existential risk from artificial general intelligence

OpenAI Gym provides an easy use and set up enviromnments alognwith their simple interfaces.

- User can define the action in the environment to be taken.

The basic idea of reinforcement learning can be summed up using this figure. Gym provides with both the agent and the environment



2. Agent and Environment

Let's first define concept of agent and environment formally before proceeding further for understanding technical details about RL.

Environment is the universe of agent which changes state of agent with given action performed on it.

Agent is the system that perceives environment via sensors and perform actions with actuators.

- In below situations Homer(Left) and Bart(right) are our agents and World is their environment. They performs actions on it and improve their state of being by getting happiness as reward.

NEGATIVE REINFORCEMENT

UNWANTED STIMULUS REMOVED BY BEHAVIOR

POSITIVE REINFORCEMENT

REWARDING STIMULUS PRESENTED BY BEHAVIOR

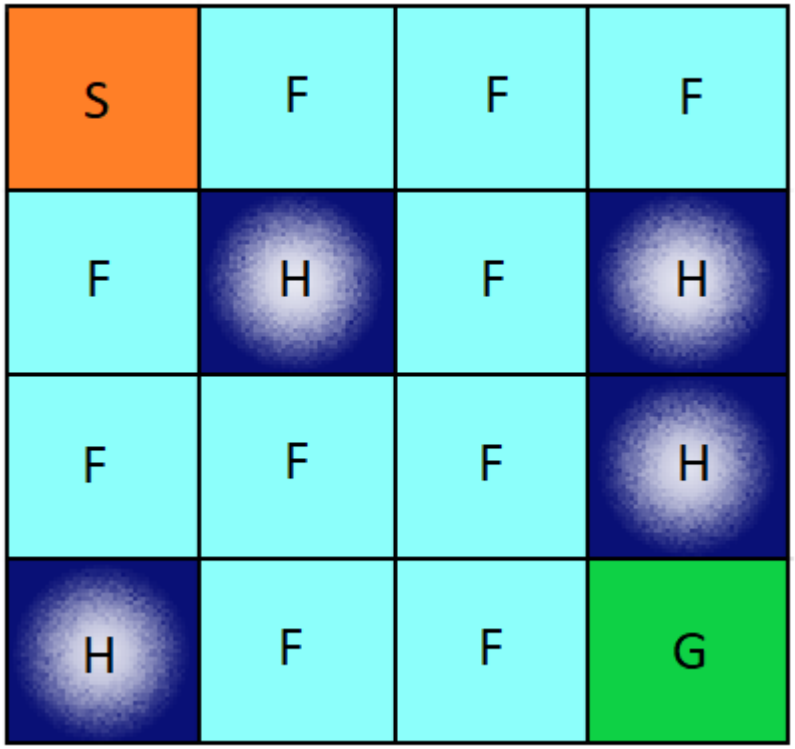
Let's see what an environment in OpenAI gym looks like

In [36]:
```
import gym
# env = gym.make('CartPole-v1')
# env = gym.make('FrozenLake-v0')
env = gym.make('Acrobot-v1')
# env = gym.make('MountainCarContinuous-v0') # try for different environements
observation = env.reset()
for t in range(1000):
        env.render()
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
#         print(observation, reward, done, info)
        if done:
            print("Finished after {} timesteps".format(t+1))
            break
env.close()
```

```
Finished after 500 timesteps
```

3. Q-Learning

To understand Q-Learning, let's look at the frozen lake environment first



Let's now look at at how to solve for the environment

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

The above equation states that the Q-value yielded from being at state s and performing action a is the immediate reward r(s,a) plus the highest Q-value possible from the next state s'. Gamma here is the discount factor which controls the contribution of rewards further in the future.

Q(s',a) again depends on Q(s",a) which will then have a coefficient of gamma squared. So, the Q-value depends on Q-values of future states as shown here:

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a) \dots\dots\dots \gamma^n Q(s''^{\dots n},a)$$

Since this is a recursive equation, we can start with making arbitrary assumptions for all q-values. With experience, it will converge to the optimal policy. In practical situations, this is implemented as an update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

where alpha is the learning rate or step size. This simply determines to what extent newly acquired information overrides old information.

In [37]:
```
import gym
import numpy as np
import time, pickle, os

env = gym.make('FrozenLake-v0')
```

In [38]:
```
# Let's declare the user defined parameters now
epsilon = 0.9
total_episodes = 100
max_steps = 100

lr_rate = 0.81
gamma = 0.96

# creating a Q table that will store the possible state action pairs
Q = np.zeros((env.observation_space.n, env.action_space.n))
```

In [39]:
```
def choose_action(state):
    action=0
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state, :])
    return action
```

In [40]:
```
def learn(state, state2, reward, action):
    predict = Q[state, action]
    target = reward + gamma * np.max(Q[state2, :])
    Q[state, action] = Q[state, action] + lr_rate * (target - predict)
```

In [41]:
```
for episode in range(total_episodes):
    state = env.reset()
    t = 0

    while t < max_steps:
#         env.render()
        action = choose_action(state)
        state2, reward, done, info = env.step(action)
        learn(state, state2, reward, action)
        state = state2
        t += 1
        if done:
            break
        #time.sleep(0.1)
# print(Q)

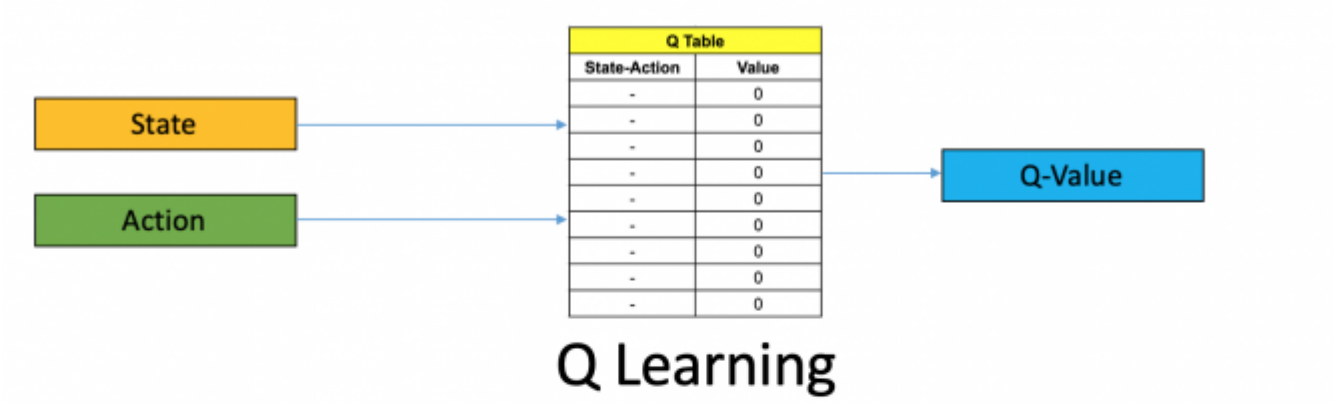
with open("frozenLake_qTable.pkl", 'wb') as f:
    pickle.dump(Q, f)
```

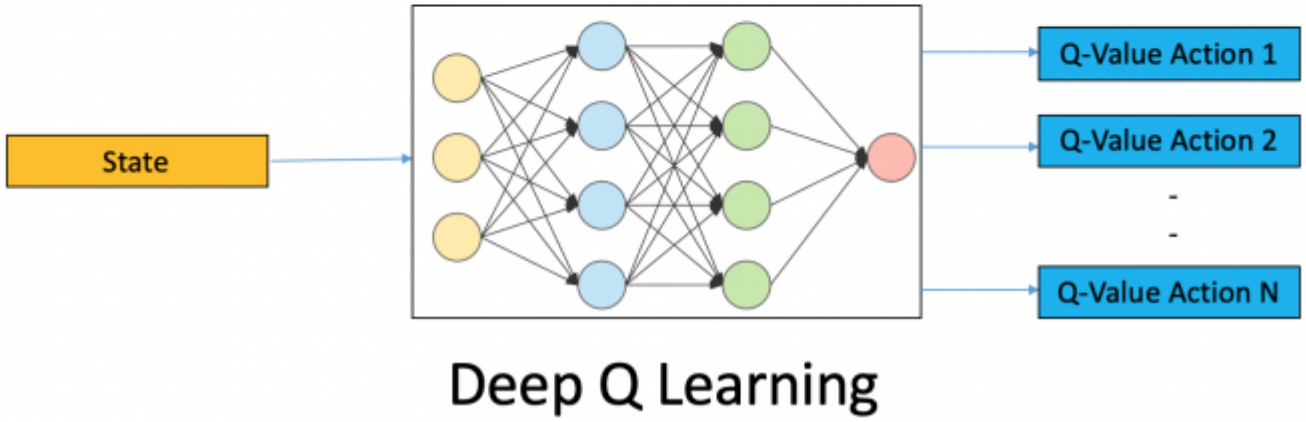
But wait, does that mean we have to remember a table so big to take an action?

4. Deep Q-Learning for Frozen Lake

As above, the actions taken by default in an OpenAI gym environment are random. Let's code the Q-learning algorithm to solve this

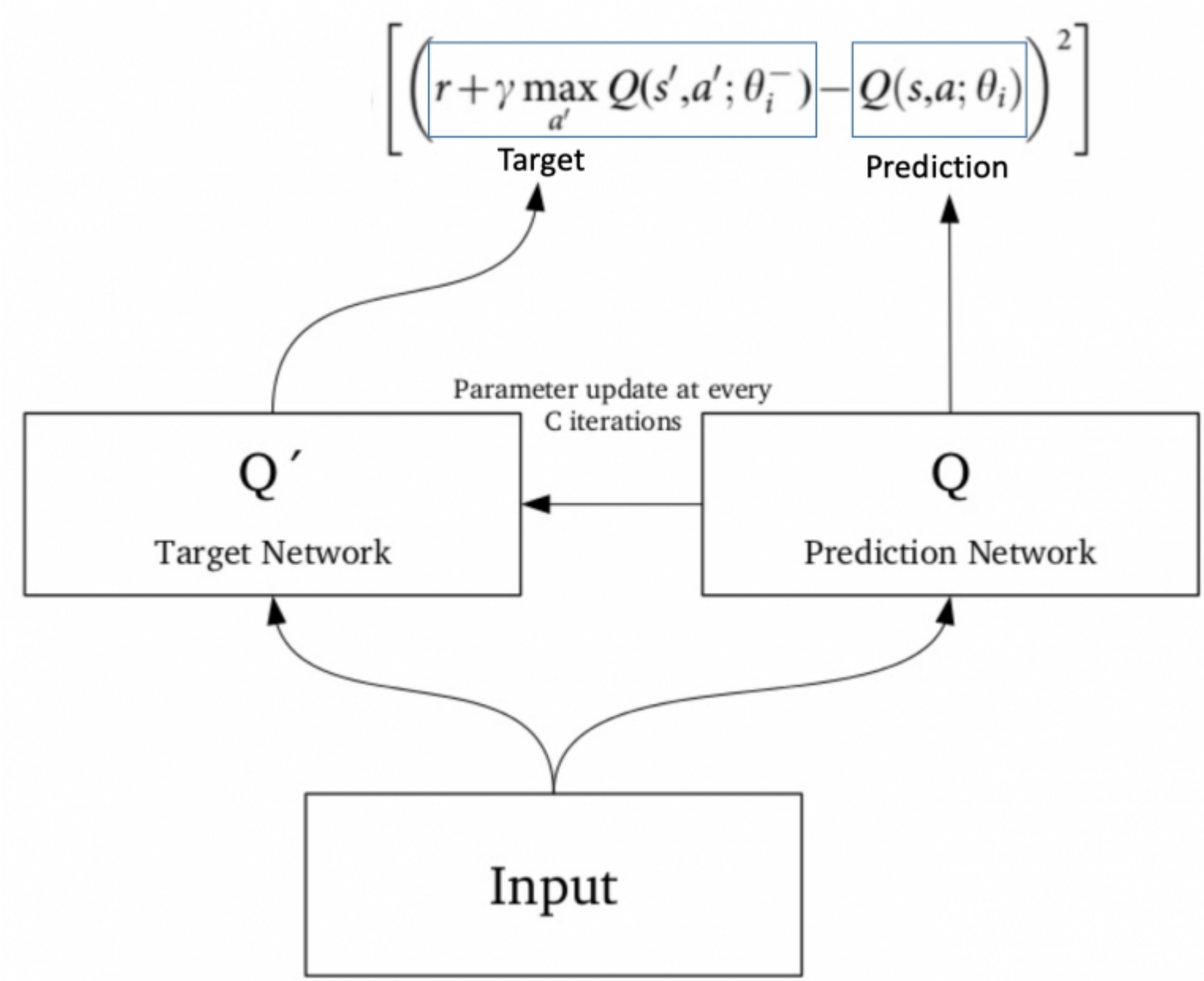
Deep Q Learning





Let's go back to the equation and see what is to be learnt by the network

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$


$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-)-Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every
C iterations
Q´
Target Network
Q
Prediction Network
Input

In [42]:
```
import gym
env = gym.make('CartPole-v0')
print(env.action_space)
print(env.observation_space)
```

```
Discrete(2)
Box(4,)
```

In [43]:
```
import gym
env = gym.make('FrozenLake-v0')
print(env.action_space)
print(env.observation_space)
```

```
Discrete(4)
Discrete(16)
```

In [44]:
```
import tensorflow as tf

tf.reset_default_graph()

#These lines establish the feed-forward part of the network used to choose actions
inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32)
W = tf.Variable(tf.random_uniform([16,4],0,0.01))
Qout = tf.matmul(inputs1,W)
predict = tf.argmax(Qout,1)

#Below we obtain the loss by taking the sum of squares difference between the target and prediction Q values.
nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32)
loss = tf.reduce_sum(tf.square(nextQ - Qout))
trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
updateModel = trainer.minimize(loss)
```

In [45]:
```
init = tf.global_variables_initializer()

# Set learning parameters
y = .99
e = 0.1
num_episodes = 2000
#create lists to contain total rewards and steps per episode
jList = []
rList = []
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        #Reset environment and get first new observation
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        #The Q-Network
        while j < 99:
            j+=1
            #Choose an action by greedily (with e chance of random action) from the Q-network
            a,allQ = sess.run([predict,Qout],feed_dict={inputs1:np.identity(16)[s:s+1]})
            if np.random.rand(1) < e:
                a[0] = env.action_space.sample()
            #Get new state and reward from environment
            s1,r,d,_ = env.step(a[0])
            #Obtain the Q' values by feeding the new state through our network
            Q1 = sess.run(Qout,feed_dict={inputs1:np.identity(16)[s1:s1+1]})
            #Obtain maxQ' and set our target value for chosen action.
            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0,a[0]] = r + y*maxQ1
            #Train our network using target and predicted Q values
            _,W1 = sess.run([updateModel,W],feed_dict={inputs1:np.identity(16)[s:s+1],nextQ:targetQ})
            rAll += r
            s = s1
            if d == True:
                #Reduce chance of random action as we train the model.
                e = 1./((i/50) + 10)
                break
        jList.append(j)
        rList.append(rAll)
```

```
print("Percent of succesful episodes: " + str((sum(rList)/num_episodes)*100) + "%")
```

```
Percent of succesful episodes: 46.9%
```

Reference

- https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ (https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/)
- https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2 (https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2)
- https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288 (https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288)

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: