7.13 Results

The previously described code provided an example of how to classify Mnist images using RNNs. The code only evaluated accuracy for this example although you can evaluate for the other metrics as well. I leave that as an exercise to the reader. For comparison, I ran this code as written here an obtained classification accuracies as high as or higher than 97%-99%.

7.14 Summary

In this chapter Recurrent Neural Networks (RNNs) were presented and discussed. An example using the Mnist hand written digits data set was used for the analysis. Issues related to data representation and RNN architecture were also discussed.

CHAPTER 8: GENERATIVE ADVERSARIAL NETWORKS

In this section of the book I will cover Generative Adversarial Networks (GANs). Generative Adversarial Networks (GANs) (Goodfellow et al. 2014) are a first attempt at representing unsupervised problems in the context of games (e.g. GANs are modeled as a two player adversarial game). One of the biggest challenges we face with supervised learning is annotating the data. We cannot annotate automatically and without annotations we cannot train our learning models. But what if we could substitute the annotation of the data for something else? For instance, what if we could model the annotation task as a game or use other previous knowledge about the world as labels. These ideas are one of the main motivations for GANs.

GANs are deep neural networks that consist of a generator network connected to a discriminator network. The discriminator network has training data and the generator network only has random or noise data as input. GANs are essentially 2 player games where one player (the generator) creates synthetic data samples, while the second player (the discriminator) takes the generated sample and performs a classification.

This classification is performed to determine if the synthetic sample is similar to the distribution of the discriminator's training data. Since both networks are connected, the deep neural network (GAN) can learn to generate better synthetic samples with the help of the discriminator's output. Basically, the discriminator tells the generator how to adjust its weights to produce better synthetic samples.

Generative Adversarial Networks are methods that use 2 deep neural networks to interact with each other and generate data. Its formulation is consistent with 2 player adversarial game frameworks. One of the 2 algorithms (or networks) tries to learn a data distribution and produce new samples similar to the samples in the real data (the generator). The second algorithm (the adversary) is a classifier that tries to determine if the new samples generated by the generative algorithm are fake or real. These 2 algorithms work together to achieve an optimal outcome of producing better output samples.

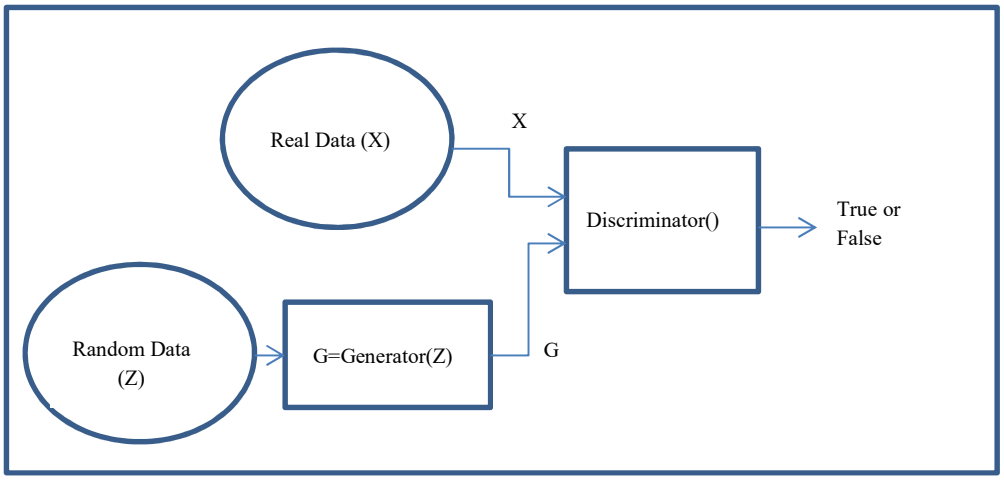


Figure. A GAN network using MNIST.

The code to implement a GAN network is presented below.

8.1 GAN code

In this section, the GAN code will be described. First we list the libraries that we need.

```
import tensorflow as tf
import numpy as np
from numpy import genfromtxt
```

Next we initialize our variables

```
batch_size = 8
hidden_size = 4

num_steps = 5000
display_step = 10

seed = 42
tf.set_random_seed(seed)
```

We will need some helper functions such as a log estimation function.

```
def log(x):
    return tf.log(tf.maximum(x, 1e-5))
```

To keep things simple, instead of reading in data, we are going to generate it automatically. We will generate samples with a normal distribution of **mean**=4 and **sigma** = 0.5. **N** is the number of samples to generate.

These samples are for the discriminator and represent the real data.

```
class DataDistribution(object):
    def __init__(self):
        self.mu  = 4
        self.sigma = 0.5

    def sample(self, N):
        samples = np.random.normal(self.mu, self.sigma, N)
        samples.sort()
        return samples
```

The data from the distribution class looks like this:

```
array([ 3.3777126 ,  3.46725909,  3.65951541,  3.81755036,  3.81998276,
     3.92007   ,  4.28720089,  4.51158124])
```

We also provide a set of data for the generator. Think of this as noise. We set a range between [-range, range] but the values are random. The generator uses noise as input.

```
class GeneratorDistribution(object):
    def __init__(self, range):
        self.range = range

    def sample(self, N):
        return np.linspace(
          -self.range, self.range, N) + np.random.random(N) * 0.01
```

The data from the generator class looks like this:

```
>>> generatorData

array([-7.99054428, -6.2211434 , -4.43866942, -2.65729133, -0.88879437,
       0.89649839,  2.67079517,  4.45113515,  6.22879001,  8.00656172])
```

First we define the layer function for the GAN.

```
def layer_GAN(input, weight_shape, bias_shape):
        w_init = tf.random_normal_initializer(stddev=1.0)
        bias_init = tf.constant_initializer(0.0)
        W = tf.get_variable("w", weight_shape, initializer=w_init )
        b = tf.get_variable("b",  bias_shape,  initializer=bias_init )

        return tf.matmul(input, W) + b
```

With GANs we have 2 inference functions; one for the generator and one for the discriminator. The inference function for the generator is:

```
def inference_generator(input):
    h1=tf.nn.softplus(layer_GAN(input,[input.get_shape()[1], 4], [4]))
    h2 = layer_GAN(h1, [ h1.get_shape()[1], 1] , [1])

    return h2
```

The inference function for the generator with values is:

```
def inference_generator(input):
    h1=tf.nn.softplus(layer_GAN(input,[1, 4], [4]))
    h2 = layer_GAN(h1, [ 4, 1] , [1])

    return h2
```

The neural network in inference_generator() looks like the following:

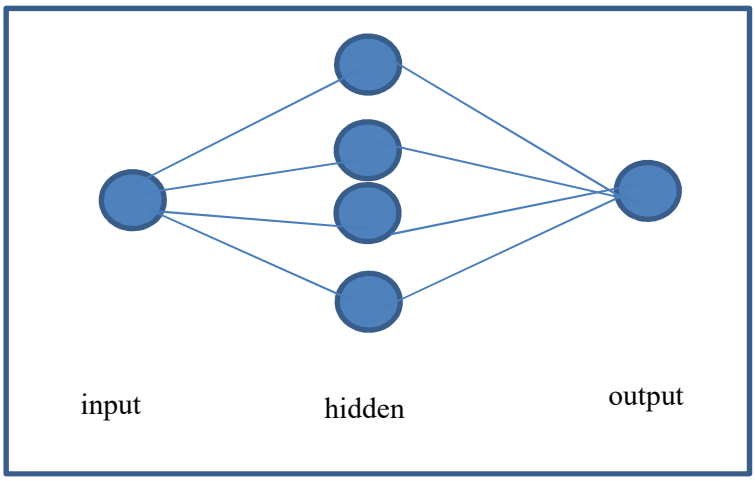


Figure. Neural network in inference_generator().

And the inference function for the discriminator is:

```
    def inference_discriminator(input):
        h1 = tf.nn.relu(layer_GAN(input, [ 1, 8 ],[8] ))
        h2 = tf.nn.relu(layer_GAN(h1, [8, 8] , [8]))
        h3 = tf.nn.relu(layer_GAN(h2, [8, 8], [8]) )
        h4 = tf.sigmoid(layer_GAN(h3, [  8, 2  ] , [2]))

        return h4
```

Notice that the discriminator has more capacity to learn.

The neural network in inference_discriminator() looks like the following:

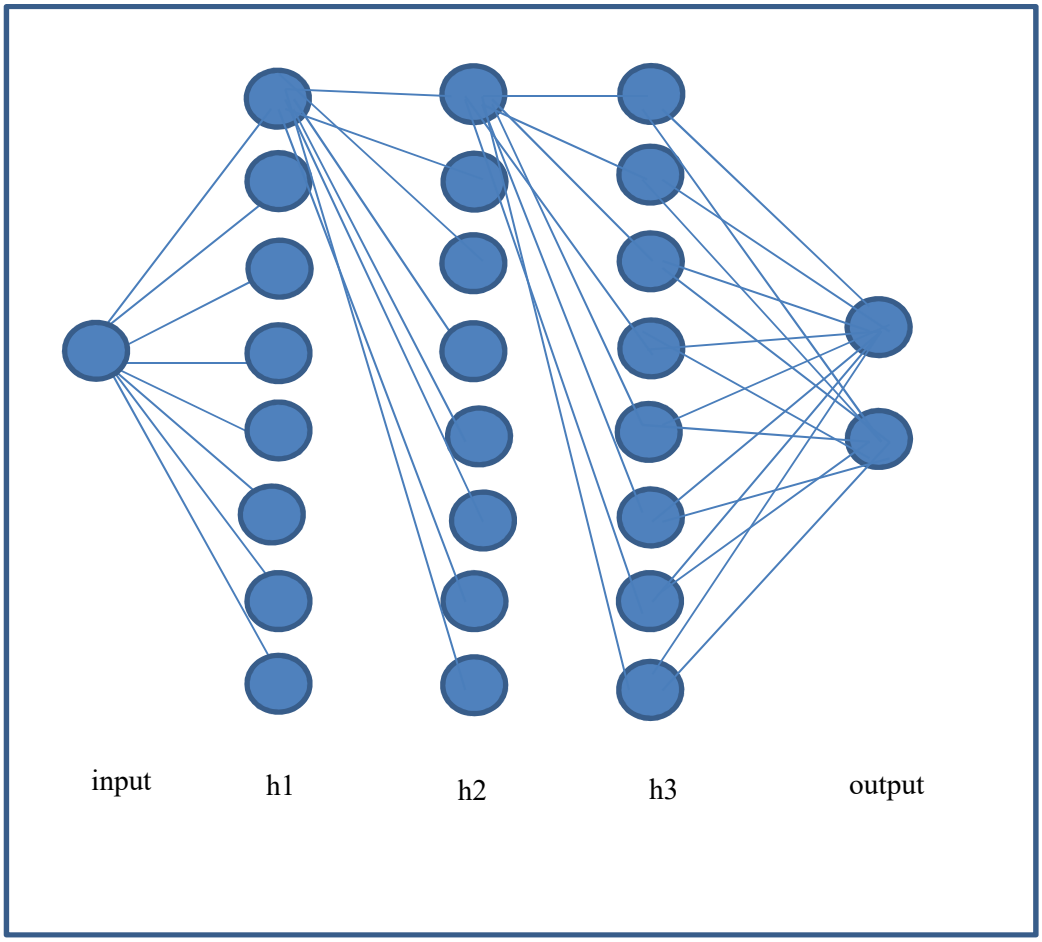


Figure. Deep neural network in inference_discriminator().

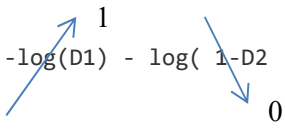
The GAN has 2 loss functions to calculate max entropy; one for the discriminator and one for the generator.

The loss for the discriminator is as follows:

```
def loss_d_GAN(D1, D2):
    loss_d = tf.reduce_mean(  -log(D1) - log( 1-D2  )   )
    return loss_d
```

Think of the 2 parameters in loss_d_GAN as tending to 1 and 0 like so

```
def loss_d_GAN(D1, D2):
    loss_d = tf.reduce_mean(  -log(D1) - log( 1-D2  )   )
    return loss_d
```

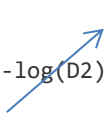


And the loss for the generator is:

```
def loss_g_GAN(D2):
    loss_g = tf.reduce_mean(   -log(D2)   )
    return loss_g
```

Think of parameter in loss_g_GAN as tending to 1 like so

```
def loss_g_GAN(D2):
    loss_g = tf.reduce_mean(   -log(D2)   )
    return loss_g
```



Next we can define the optimization function as follows:

```
def training_GAN(cost):
    step = tf.Variable(0)
    optimizer = tf.train.AdamOptimizer(0.001)
    train_op=optimizer.minimize(cost)
```

```
        return train_op
```

Once the core functions are defined, we can proceed to define our placeholders. We select a batch size of 8 so that **x** and **z** are data matrices of size [8, 1]. So they contain 8 samples with 1 feature each.

```
    ## batch size  = 8
    ## so x and z are data matrices of size 8 x 1
    ##  8 samples with 1 feature each

    x = tf.placeholder(tf.float32, shape=(batch_size  , 1))
    z = tf.placeholder(tf.float32, shape=(batch_size  , 1))
```

Now we proceed to call the core functions as shown below:

```
    with tf.variable_scope('G'):
        output_G = inference_generator(z)
    with tf.variable_scope('D'):
        output_D1 = inference_discriminator(x)
    with tf.variable_scope('D'):
        output_D2 = inference_discriminator(output_G )

    ###############################################

    cost_d = loss_d_GAN(output_D1,output_D2)
    cost_g = loss_g_GAN(output_D2)

    ###############################################

    train_op_d = training_GAN(cost_d)
    train_op_g = training_GAN(cost_g)
```

We are almost done. All that remains is to initialize the variables and create the session.

```
    init = tf.initialize_all_variables()
    sess = tf.Session()
    sess.run(init)
```

Before we call the main loop we need to have some training data. In this case we call the data generating classes we previously defined and create some data.

```
data=DataDistribution()
gen=GeneratorDistribution(range=8)
```

Finally, we can call the main loop as follows:

```
for step in range(num_steps):
    x_data = data.sample(batch_size)
    z_data = gen.sample(batch_size)

    ## reshape from row to column vector
    x_reshaped = np.reshape(x_data, (batch_size, 1))
    z_reshaped = np.reshape(z_data, (batch_size, 1))
    res_cost_d,res_train_d = sess.run(
             [cost_d,train_op_d], feed_dict={x: x_reshaped,z: z_reshaped})

    #update new data for generator
    z_data = gen.sample(batch_size)
    z_reshaped = np.reshape( z_data,  (batch_size, 1)  )
    res_cost_g, res_train_g = sess.run(
                        [cost_g,train_op_g], feed_dict={ z: z_reshaped })

    if step % display_step == 0:
        print('{}: cost_d: {:.4f}\t cost_g: {:.4f}'.format(
                                            step, res_cost_d, res_cost_g))
        print('{}: train_d {}\t train_g: {}'.format(
                                           step, res_train_d, res_train_g))
```

And that is it! We have completed our GAN model.

8.2 Some Uses of GANs

Generative Adversarial Networks (GANs) are one of the latest and most exciting developments in machine learning during the last decade (Goodfellow 2014). At this point, the use of GANs has been focused on research for image processing and synthetic generation. However, several studies have looked at the application of GANs to cyber security problems.

Currently, GANs have been used to generate works of art in the styles of Picasso, for instance, or they can potentially generate text that is similar to the styles of Shakespeare or other great authors. The application of GANs to cyber security is more recent but there already exists a body of work to highlight possible applications. In particular, the common theme is that GANs can be used by attackers to masquerade their efforts. Recent works have used GANs for password generation (Hitaj 2017) and steganography (Shi 2017). It is easy to see how this idea could also be extended to polymorphic viruses and synthetically generated network attacks.

Understanding how attackers can use GANs to masquerade their efforts is critical to understanding how to develop better intrusion or malware detection systems.

8.3 Summary

In this chapter, a description of Generative Adversarial Networks was provided. Some sample code was addressed as well as some applications of GANs to cyber security.

CHAPTER 9: REINFORCEMENT LEARNING

In this section of the book I will cover the topic of Reinforcement Learning. This is an area of machine learning somewhere between supervised learning and unsupervised learning. It has been extensively applied to recommender systems and AI-based games. Recently, it was shown that a deep Q-network, using only pixels and game scores as inputs, could surpass achieve a playing level comparable to that of professional human gamers across a set of 49 Atari games (Mnih et al. 2015). The main advantage of applying reinforcement learning to games is that games are governed by rules. You have game states (the inputs) and actions (output) that lead to new states and rewards (the objectives to maximize). Because of this, no annotation is neededand instead you rely on the rules of the game for feedback (e.g. instead of annotated labels).

There are several types of reinforcement learning techniques. In this chapter, I will focus on getting started with Q-learning since this is the technique used in the Mnih et al (2015) paper I referenced above. Here, I will try to provide a simple intuition based description of the technique. I should note that to achieve the level of Q-Learning presented in the Mnih et al (2015) paper, several additional optimizations need to be included. However, the discussion in this chapter should provide a simple way to get started with Q-Learning.

So what is Q-Learning? Q-Learning tries to learn the value of being in a given state (s), and taking a specific action from there.

As I indicated, Q-learn has been applied to games. The best way to understand the algorithm is to analyze it from the point of view of a game. Here we will use Python's OpenAI Gym module to play games. We will select the simple FrozenLake game environment.

FrozenLake is a game about crossing a frozen lake that has some cracks in the ice with holes and there is wind sometimes that pushes the person crossing it. The game is very simle and consists of a grid that is 4x4 like so.

hole	frozen	cheese	hole
frozen	frozen	hole	frozen
hole	frozen	frozen	hole
frozen	hole	frozen	start

So, the objective is to get to the cheese without falling into a hole or being pushed by the wind into a hole. There are 4 moves which are up, down, right, and left. There is only one reward and that is to get to the cheese. However, you only get that reward in the future by first taking several steps on frozen blocks without falling in a hole. Therefore, one challenge is that you have to state your objective in terms of several future moves. This is accomplished using something called the Bellman Equation.

The key to predicting these rewards is to know the associated reward given a current state and action to take. This is called a Q maping

Q (state, action) = reward

For such a simple grid, we could just use a table. In this case our table would be 16x4 because there are 16 possible states (position in the grid of 4x4) and there are 4 actions (up, down, right, left). Since we know the rules of the game and the layout of the grid, we can populate the table and learn the Q rewards for each state/action pair.

An example of the table can be seen below.

	Up	Down	Left	right
State1	Q=0.6	Q=0.8	Q=0.1	Q=0.0
State2				
State3				
State4	0		0.1	
State5				
State6			0	
State7		0.6		0
State8			1	
State9	0			
State10		0		1
State11			1	
State12	0.02			
State13			0.4	
State14		0.6		
State15		0.3	0	
State16		1		0

Figure. Q-Learn Table

Now the main challenge is that we need to learn future rewards for future actions as we move through the grid. Here the bellman equation will help. Think if the bellman equation as a type of recursive equation that looks at the future state given a current state. The Bellman equation is as follows:

Q(state, action) = reward + weight * max [Q(future_state, future_action)]

These values can be looked up from the Table.

The code discussed here can be downloaded from the course website or the github repository. In the next section, the python Q-learning code will be discussed which only uses a table to determine the rewards and the path to follow. Section 9.2 will use the same algorithm but will replace the use of the table with a neural network so that we can see how deep neural networks can improve the approach.

9.1 Q-Learning using a Table

In this section we discuss the code to implement Q-Learning using a table. This code makes use of the OpenAI gym library. The libraries used can be seen in the next code segment.

```
import numpy as np
import gym
```

The frozenLake game can be initialized by creating the **env** object as can be seen below. This object represents the game and holds all the parameters related to states, actions, rewards, and current game state.

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

The next step is to initialize the table **Q** to all zeros and of size 16x4. Here `env.observation_space.n = 16` and `env.action_space.n = 4`.

```
Q = np.zeros([env.observation_space.n,env.action_space.n])
lr = .8
y = .95
num_episodes = 2000
```

We take 2000 epochs (or episodes) and initialize some parameters **lr** and **y**. Each episode represents a game played. We use **jList** and **rList** to collect the number of steps taken per episode and the total reward per episode, respectively. These are used to collect results of each game.

```
jList = []
rList = []
```

The following code segment goes over the main loop of the Q-learn algorithm. In the next code segment, the line

```
for i in range(num_episodes):
```

indicates that we are going to play num_episodes=2000 games. During these 2000 tries we will learn the best path to take.

```
for i in range(num_episodes):
    s = env.reset()
    rAll = 0
    d = False
    j = 0
    while j < 99:
        j+=1
        zz = env.action space.n
        a=np.argmax(Q[s,:]+np.random.randn(1,zz) *(1.0/(i+1)))
        s1,r,d,_ = env.step(a)
        Q[s,a] = Q[s,a] + lr*(r + y*np.max(Q[s1,:]) - Q[s,a])
        rAll += r
        s = s1
        if d == True:
            break
    #jList.append(j)
    rList.append(rAll)
```

The line

s = env.reset()

restarts the game for every episode so we can play it again and assign the initial state to **s**. The variable **rAll** adds up the accumulated rewards for this episode. The variables **d** and **j** are control variables to indicate if the game has ended and to count the number of steps taken.

The code in the while loop is what allows the algorithm to learn or update the values in the Q table designated by the variable **Q**. To take the first step we need to pick an action to follow. We do this with the following lines of code

```
zz = env.action_space.n
a=np.argmax(  Q[s,:]+np.random.randn(1,zz) *(1.0/(i+1))  )
```

The variable **zz** is the size n of all actions in the game (up, down, left, right) which in this case is 4. The statement Q[s, :] selects the current Q values (rewards) associated with state **s**. The statement

```
np.random.randn(1,zz) *(1.0/(i+1))
```

adds randomness to the 4 Q values for the current state. Basically, you randomly increment the Q values for the current state and then select the highest one with

np.argmax()

by selecting the highest Q value you determine what action (**a**) you take given the current state.

Once the action **a** is selected, we can proceed to evaluate it in the game to obtain our new state (position) and the reward (did we fall in a hole or advanced to a frozen block). We do this with

```
s1,r,d,_ = env.step(a)
```

here, **s1** is the new state (position) and **r** is the reward. The parameter **d** indicates end of the game. Given this new information about the result of our action, we can proceed to update the Q-table with our new resuts and new knowledge about the state of the game. This is done with the statement

```
Q[s,a] = Q[s,a] + lr*(r + y*np.max(Q[s1,:]) - Q[s,a])
```

In this statement, Q[s, a] contains the current Q value (reward) associated with the state **s** and the action **a**. This is the Bellman equation which can be viewed as

```
next_s_Q = lr*(r + y*np.max(Q[s1,:]) - Q[s,a])
Q[s,a] = Q[s,a] + next_s_Q
```

next_s_Q contains the current reward for state **s** plus the maximum reward for the next state **s1**. The parameters **lr** and **y** are weights to control the importance of the next state's reward when updating the current states reward (Q value).

We can think of this parameter

```
- Q[s,a] )
```

as a regularization parameter.

At this point we are almost done and we can proceed to accumulate our results. The statement

```
rAll += r
```

accumulates the total rewards. The statement

```
s = s1
```

assigns the current state **s1** to **s**. The line

```
if d == True:
    break
```

ends the game if **d** indicates end of game. The statement

```
jList.append(j)
```

accumulates the number of steps taken to reach end of game. The statement

```
rList.append(rAll)
```

appends rewards per game to a list so that they can be viewed later.

```
print "Score over time: " +  str(sum(rList)/num_episodes)
```

That is it. We have finished our discussion of Q-learn with tables on the frozenLake game. Now we can proceed to replace the table with a neural network.

9.2 Q-Learning using a Neural Network

Now that we understand the frozenLake game with a table, we can proceed to replace the table with a neural network. It is important to note here that the weigths matrix **W** in the neural network will now represent the Q table.

In this section of the chapter I will only discuss the parts that are different from the previous implementation.

First we include the libraries as can be seen below. Notice we now add Tensorflow.

```
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
```

We create the game with the **env** object.

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

Next, we define our familiar neural network functions inference(), loss(), and train(). The function inference() creates **W** which is our new Q table. Notice the dimensions of **W** are 16x4 because we have 16 states in the game and 4 actions. **Qout** (our predicted **y** in previous chapters) is the result of a matmul operation between inputs1 (our states) and **W** (the weiths or Q values in this case).

With

```
predict = tf.argmax(Qout,1)
```

we select the action (**a**) to take. Here is the code for the inference function.

```
def inference(inputs1):
    W = tf.Variable(tf.random_uniform([16,4],0,0.01))
    Qout = tf.matmul(inputs1,W)
    predict = tf.argmax(Qout,1)
    return predict, Qout, W
```

As can be seen from the previous code, the network looks like the figure below. It is important to note that this is a basic architecture and that much more complex deep architectures with different activation functions could be used such as architectures with many hidden layers or convolutional neural networks, etc.

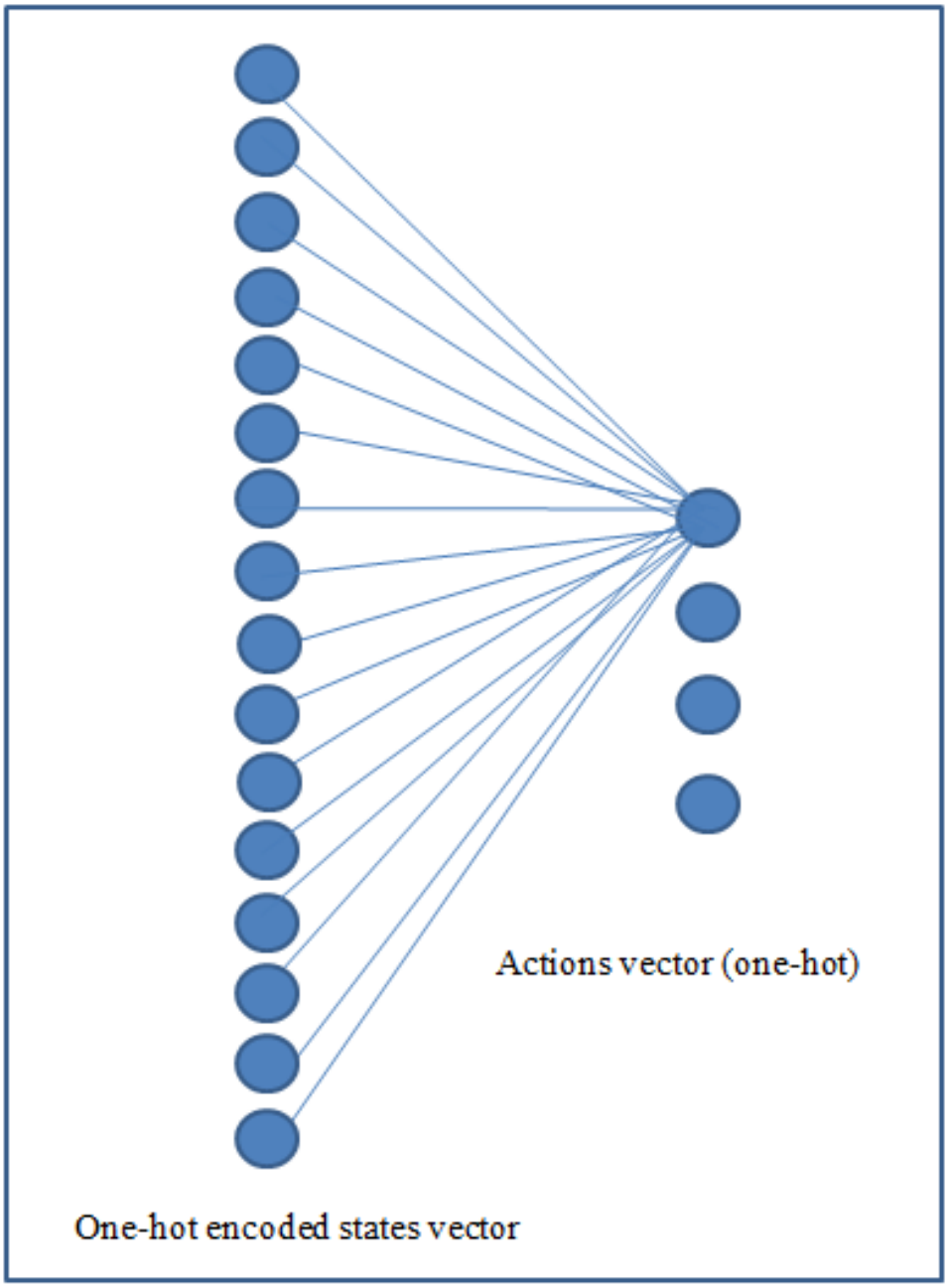


Figure. Q-Learning network.

The loss function is Least Squares Estimation which is the same as linear regression! Here, basically,we compare Current_Q to estimated_Q and try to minimize the error.

```
def loss(nextQ, Qout):
    loss = tf.reduce_sum(tf.square(nextQ - Qout))
    return loss
```

The optimization is nothing more than the very familiar Gradient Descent with a learning rate of 0.1.

```
def train(loss):
    trainer = tf.train.GradientDescentOptimizer(learning rate=0.1)
    updateModel = trainer.minimize(loss)
    return updateModel
```

I leave evaluate() for the reader to complete as an exercise.

```
def evaluate():
    print "evaluate"
```

In the next statement we initialize the placeholder to hold the data. The placeholder **inputs1** holds the one hot encoded vector representing the state of the game. The placeholder **nextQ** is used to store the one hot encoded vector of the 4 possible rewards for each action to take.

```
tf.reset_default_graph()

inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32)
nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32)
```

Next, we call the core functions like so

```
predict, Qout, W = inference(inputs1)
cost = loss(nextQ, Qout)
trainOp = train(cost)
```

Now we are ready for the main loop. We initialize the variables in the graph and a few parameters.

```
init = tf.initialize_all_variables()
y = 0.99
e = 0.1
num_episodes = 2000
```

Then we create lists to contain total steps taken per episode (game) and total rewards per game.

```
jList = []
rList = []
```

Finally, we are ready for the main loop which is shown in the next code segment below.

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(num episodes):
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        while j < 99:
            j=j+1
            a,allQ = sess.run( [predict,Qout],  feed dict=
                      {inputs1:np.identity(16)[s:s+1]})

            s1,r,d,_ = env.step(a[0])
            Q1 = sess.run(Qout, feed dict=
                      {inputs1:np.identity(16)[s1:s1+1]})

            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0,a[0]] = r + y*maxQ1
             ,W1 = sess.run( [trainOp, W], feed dict=
                      {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})

            rAll += r
            s = s1
            if d == True
                break
        jList.append(j)
        rList.append(rAll)
```

As an be seen in the code segment below, we run the main loop 2000 times (**num_episodes**) which means that we play 2000 games. Each time we play a game, we reinitialize the board (`s = env.reset()`) and initialize the rewards variable (**rAll**) to zero. The variable **j** is the counter for the current step and d is used to determine in the game is over (win or loss).

```
for i in range(num_episodes):
    s = env.reset()
    rAll = 0
    d = False
    j = 0
```

for every game iteration we run the following while loop. This while loop is the main code that helps us to learn that Q values and traverse the board (e.g. play the frozen lake game).

```
   while j < 99:
        j=j+1
        a,allQ = sess.run( [predict,Qout],  feed_dict=
                 {inputs1:np.identity(16)[s:s+1]})

        s1,r,d,_ = env.step(a[0])
        Q1 = sess.run(Qout, feed_dict=
                 {inputs1:np.identity(16)[s1:s1+1]})

        maxQ1 = np.max(Q1)
        targetQ = allQ
        targetQ[0,a[0]] = r + y*maxQ1
        _,W1 = sess.run( [trainOp, W], feed_dict=
                 {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})

        rAll += r
        s = s1
        if d == True
            break
```

We perform 99 steps since it should not take more than 99 steps to traverse the frozen lake. If it does, the game should end. The first line in the while loop is used to increment the steps

```
        j=j+1
```

after incrementing the steps, we proceed to perform our first session run operation to train the Tensorflow graph. Here we call **predict** and **Qout** from the inference() function calls.

```
a,allQ = sess.run( [predict,Qout],  feed_dict=
                {inputs1:np.identity(16)[s:s+1]})
```

The statement

```
np.identity(16)[s:s+1]
```

takes the current state in the variable **s** and converts it into a one-hot encoded representation. For instance, if the current state is 4, then the one-hot encoded representation (of size 16) looks like this

[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The next step is to take the predicted action in "**a"** and run it through the game. We use **a[0]** instead of just **a** because **a** is a tensor. Assuming the action is 1 (down), printing **a** alone will result in

[1]

Whereas, printing a[0] will result in

1

So the below statement runs the action through env.step() and this function returns the new state **s1** which is the new position in the frozen lake grid, **r** is the reward associated with the step **s** (for instance r=0.43) , and **d** indicates if the game is over (found the cheese or fell in the frozen lake).

```
s1,r,d,_ = env.step(a[0])
```

with the new state **s1**, we proceed to run the Tensorflow graph again with session run. Here we call **Qout** again using **s1**.

Recall that **Qout** is

Qout = tf.matmul(inputs1, W)

in the inference function. In this case, **inputs1** is the one-hot encoded vector of size 16 that represents state **s1**.

```
Q1 = sess.run(Qout, feed_dict= {inputs1:np.identity(16)[s1:s1+1]})
```

So **Q1** will now contain the 4 neuron vector with the Q values for all 4 actions given state **s1**. Currently, the vector **allQ** for state **s** looks like this with some values

```
allQ = [ 0.0
         0.3
         0.4
         0.02 ]
```

And **Q1** for state **s1** looks like this for some values

```
Q1 = [ 0.9
       0.1
       0.04
       0.7 ]
```

Therefore, we have predicted Q values for state "**s"** and predicted Q values for state "**s1"**.

Next, we proceed to select the highest value in **Q1**. In this case **maxQ1** gets assigned the value 0.9 from our previous example (**maxQ1**=0.9). The code is as follows

```
maxQ1 = np.max(Q1)
```

now we use a new variable **targetQ** which will be equal to the bellman equation. We assign to it **allQ**

```
targetQ = allQ
```

so that **targetQ** is now

```
targetQ  =  [   0.0
                0.3
                0.4
                0.02   ]
```

Recall that a[0] holds the index of the action taken (e.g. down or 1). Therefore, in the vector **targetQ** we select that position (targetQ[0, 1]) and add to it the reward value **r** and **maxQ1** times some **y** parameter. Recall that the Bellman equation looks like this:

Q(state, action) = reward + weight * max [Q(future_state, future_action)]

The code is as follows

```
targetQ[0,a[0]] = r + y*maxQ1
```

and with values this looks like the following

```
targetQ[0,1] = 0.43 + 0.99*0.9
```

after this update rule **targetQ** has been modified from

```
targetQ  =  [   0.0
                0.3
                0.4
                0.02   ]
```

to

```
targetQ  =  [   0.0
                1.321
                0.4
                0.02   ]
```

interestingly, only one of the 4 values in **targetQ** is updated using the bellman equation. The other values remain the same. Finally, we do a final update of the Tensorflow graph by calling **trainOp** with session run and state "**s**". Additionally, the placeholder **nextQ** is assigned the result from the Bellman equation **targetQ**.

```
_,W1 = sess.run( [trainOp, W], feed_dict=
                {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})
```

This is important because **nextQ** will be used in the loss function with the next predicted **Qout** like so

```
predict, Qout, W = inference(inputs1)
cost = loss(nextQ, Qout)
trainOp = train(cost)
```

Finally, the last peace of code adds up the rewards, assigns the new state **s1** to **s**, and checks to see if the game is over.

```
rAll += r
s = s1
if d == True
    break
```

once you exit the while loop, the last part is to append the results of the current game to **jList** and **rList**.

```
jList.append(j)
rList.append(rAll)
```

Well, that is it with the algorithm discussion. Finally, we print our results and plot them.

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

plt.plot(rList)
plt.show()
plt.plot(jList)
plt.show()
```

That is it. We have completed implementing our Q learning algorithm with a neural network. In the next section we will add a simple improvement to the code that will improve performance.

9.3 Q-Learning using a Neural Network and Randomness

In the previous section we described the code to implement Q-learning with a neural network on the frozen lake game. That was the simplest implementation of it.

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        while j < 99:
            j=j+1
            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()

            s1,r,d,  = env.step(a[0])
            Q1 = sess.run(Qout, feed_dict=
                      {inputs1:np.identity(16)[s1:s1+1]})

            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0,a[0]] = r + y*maxQ1
            _,W1 = sess.run( [trainOp, W], feed_dict=
                      {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})

            rAll += r
            s = s1
            if d == True:
                e = 1./((i/50) + 10)
                break
        jList.append(j)
        rList.append(rAll)
```

To improve the results, we can add a few lines of additional code which will allow the algorithm to better converge and learn better Q-values. The additions are simple and basically relate to adding randomness to the code. Notice in the code above that a few new statements have been added.

These new lines add randomness to the selection of the next action to take. The idea is that add the beginning of the learning process, the action prediction function may not be very good. Therefore, picking an action randomly at the beginning may be better than picking actions with the inference() function. This is reflected in the code segment below.

```
if np.random.rand(1) < e:
    a[0] = env.action_space.sample()
```

a random number is obtained and compared to **e**. If less than **e**, the action **a[0]** is selected randomly

```
a[0] = env.action_space.sample()
```

as the algorithm improves and the Q values are better, the value of e can be adjusted so that action is more often selected with the inference function and not with the random function

```
a[0] = env.action_space.sample()
```

The code can be seen here.

```
if d == True:
    e = 1./((i/50) + 10)
    break
```

where

```
e = 1./((i/50) + 10)
```

adjusts the value of "**e"**.

9.4 Summary

In this chapter we have discussed the Q learning algorithm as part of the larger topic of Reinforcement Learning using tables and neural networks.

CHAPTER 10: CONCLUSIONS AND FINAL THOUGHTS

In this book I have only begun to scratch the surface on deep learning programming and methodologies. I hope that these examples and discussions helped you to improve your deep learning coding skills and furthered your interest in machine learning in general. There are many more deep learning methodologies such as recurrent neural networks that you may want to pursue as well. The Tensorflow website at www.tensorflow.org may be a good starting point to continue your studies.

In this final chapter, I want to address a few loose ends relevant to Tensorflow and I will present a few closing thoughts.

10.1 Benchmarking Tensorflow

In this section I want to address benchmarking. Tensorflow was made to be used with GPUs and CPUs. If you want to test the performance of your processors, one way to do it is with the following code. In the following code you are performing a matrix multiplication using

```
tf.matmul(a, b)
```

The important aspect is that you can select the device to use. For instance, the following

```
with tf.device('/cpu:0')
```

tells Tensorflow to use the CPU. In contrast, using

```
with tf.device('/gpu:0')
```

would tell Tensorflow to perform the computations using the GPU.

```
import Tensorflow as tf
with tf.device('/cpu:0'):
   a = tf.zeros(shape=[10000,1000], dtype=tf.float32)
   b = tf.zeros(shape=[1000,1000], dtype=tf.float32)
   c = tf.matmul(a, b)
# Creates session with log_device_placement set to True
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

# Runs the op.
for i in range(200):
    print  i
    print sess.run(c)
```

10.2 Conclusions

Since 2007, computational power has certainly improved and today machine learning can take advantage of these more powerful processors to process large amounts of data. In the following table we can see that there are many types of processors. Some are old and traditional and some are new and still experimental. The most widely used processor before 2007 was the CPU. Now, GPUs are the most exciting and promising because they allow deep neural networks to learn the model parameters in very short periods of time.

The future may bring even more types of processors which will further improve machine learning and deep neural networks. Currently, several companies are starting to develop their own neural processors. Google, for instance, has developed the Tensor Processing Unit (TPU). This processing unit accelerates deep learning calculations on their servers.