DQN_WorldNavigate

January 13, 2021

1 World Navigation with DQN

In this exercise you will play a world navigation game with Deep Q-Networks. The agent learn to solve a navigation task in a basic grid world. It will be built upon the simple one layer Q-network you created in Exercise 1.

In order to transform an ordinary Q-Network into a DQN you will be making the following improvements: 1. Going from a single-layer network to a multi-layer convolutional network. 2. Implementing Experience Replay, which will allow our network to train itself using stored memories from it's experience. 3. Utilizing a second "target" network, which we will use to compute target Q-values during our updates.

You will also implement two simple additional improvements to the DQN architecture, Double DQN and Dueling DQN, that allow for improved performance, stability, and faster training time. In the end you will have a network that can tackle a number of challenging Atari games, and we will demonstrate how to train the DQN to learn a basic navigation task.

```
In [ ]: from __future__ import division

        import gym
        import numpy as np
        import random
        import tensorflow as tf
        import tensorflow.contrib.slim as slim
        import matplotlib.pyplot as plt
        import scipy.misc
        import os
        %matplotlib inline
```

1.0.1 Load the game environment

```
In [ ]: from gridworld import gameEnv

        env = gameEnv(partial=False,size=5)
```

Above is an example of a starting environment in our simple game. The game environment outputs 84x84x3 color images, and uses function calls as similar to the OpenAI gym as possible. The agent controls the blue square, and can move up, down, left, or right. The goal is to move to the green square (for +1 reward) and avoid the red square (for -1 reward).

At the start of each episode all squares are randomly placed within a 5x5 grid-world. The agent has 50 steps to achieve as large a reward as possible. Because they are randomly positioned, the agent needs to do more than simply learn a fixed path, as was the case in the FrozenLake environment from Exercise 1. Instead the agent must learn a notion of spatial relationships between the blocks.

Feel free to adjust the size of the gridworld (default 5). Making it smaller provides an easier task for our DQN agent, while making the world larger increases the challenge.

1.0.2 Addition 1: Convolutional Layers

The first major addition to make DQNs work is to use convolutional layers to set up the networks. We are now familiar with convolutional layers after assignment `1-3`. For more information, see the Tensorflow documentation.

1.0.3 Addition 2: Experience Replay

The second major addition to make DQNs work is Experience Replay. The basic idea is that by storing an agent's experiences, and then randomly drawing batches of them to train the network, we can more robustly learn to perform well in the task. By keeping the experiences we draw random, we prevent the network from only learning about what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences.

Each of these experiences are stored as a tuple of *(state,action,reward,next state)*. The Experience Replay buffer stores a fixed number of recent memories, and as new ones come in, old ones are removed. When the time comes to train, we simply draw a uniform batch of random memories from the buffer, and train our network with them.

For our DQN, we build a simple class that allows us to store experies and sample them randomly to train the network:

```
In [ ]: class experience_buffer():
            def __init__(self, buffer_size = 50000):
                self.buffer = []
                self.buffer_size = buffer_size

            def add(self,experience):
                if len(self.buffer) + len(experience) >= self.buffer_size:
                    self.buffer[0:(len(experience)+len(self.buffer))-self.buffer_size] = []
                self.buffer.extend(experience)

            def sample(self,size):
                return np.reshape(np.array(random.sample(self.buffer,size)),[size,5])
```

This is a simple function to resize our game frames:

```
In [ ]: def processState(states):
            return np.reshape(states,[21168]) # 84 x 84 x 3
```

1.0.4 Addition 3: Separate Target Network

The third major addition to the DQN that makes it unique is the utilization of a second network during the training procedure. This second network is used to generate the target-Q values that

will be used to compute the loss for every action during training. Why not use just use one network for both estimations? The issue is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner.

These functions allow us to update the parameters of our target network with those of the primary network.

```
In [ ]: def updateTargetGraph(tfVars,tau):
            total_vars = len(tfVars)
            op_holder = []
            for idx,var in enumerate(tfVars[0:total_vars//2]):
                op_holder.append(tfVars[idx+total_vars//2].assign((var.value()*tau) + ((1-tau)
            return op_holder

        def updateTarget(op_holder,sess):
            for op in op_holder:
                sess.run(op)
```

With the additions above, we have everything we need to replicate the DQN.

1.0.5 Dueling DQN

In order to explain the reasoning behind the architecture changes that Dueling DQN makes, we need to first explain some a few additional reinforcement learning terms. The Q-values that we have been discussing so far correspond to how good it is to take a certain action given a certain state. This can be written as `Q(s,a)`. This action given state can actually be decomposed into two more fundamental notions of value. The first is the value function `V(s)`, which says simple how good it is to be in any given state. The second is the advantage function `A(a)`, which tells how much better taking a certain action would be compared to the others. We can then think of `Q` as being the combination of `V` and `A`. More formally:

$$Q(s,a) = V(s) + A(a)$$

The goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer. It may seem somewhat pointless to do this at first glance. Why decompose a function that we will just put back together? The key to realizing the benefit is to appreciate that our reinforcement learning agent may not need to care about both value and advantage at any given time. We can achieve more robust estimates of state value by decoupling it from the necessity of being attached to specific actions.

1.0.6 Implementing the network itself

```
In [ ]: class Qnetwork():
            def __init__(self,h_size):
                #The network recieves a frame from the game, flattened into an array.
```

```
#It then resizes it and processes it through four convolutional layers.
#We use slim.conv2d to set up our network
self.scalarInput =  tf.placeholder(shape=[None,21168],dtype=tf.float32)
self.imageIn = tf.reshape(self.scalarInput,shape=[-1,84,84,3])
self.conv1 = slim.conv2d( \
    inputs=self.imageIn,num_outputs=32,kernel_size=[8,8],stride=[4,4],padding=
self.conv2 = slim.conv2d( \
    inputs=self.conv1,num_outputs=64,kernel_size=[4,4],stride=[2,2],padding='V
self.conv3 = slim.conv2d( \
    inputs=self.conv2,num_outputs=64,kernel_size=[3,3],stride=[1,1],padding='V
self.conv4 = slim.conv2d( \
    inputs=self.conv3,num_outputs=h_size,kernel_size=[7,7],stride=[1,1],paddin

################################################################################
# TODO: Implement Dueling DQN
# We take the output from the final convolutional layer i.e. self.conv4 and
# split it into separate advantage and value streams.
# Outout: self.Advantage, self.Value
# Hint: Refer to Fig.1 in [Dueling DQN](https://arxiv.org/pdf/1511.06581.pdf)
#       In implementation, use tf.split to split into two branches. You may
#       use xavier_initializer for initializing the two additional linear
#       layers.
################################################################################
pass
################################################################################
#                                 END OF YOUR CODE
################################################################################

#Then combine them together to get our final Q-values.
#Please refer to Equation (9) in [Dueling DQN](https://arxiv.org/pdf/1511.0658
self.Qout = self.Value + tf.subtract(self.Advantage,tf.reduce_mean(self.Advanta
self.predict = tf.argmax(self.Qout,1)

#Below we obtain the loss by taking the sum of squares difference between the
self.targetQ = tf.placeholder(shape=[None],dtype=tf.float32)
self.actions = tf.placeholder(shape=[None],dtype=tf.int32)
self.actions_onehot = tf.one_hot(self.actions,env.actions,dtype=tf.float32)

################################################################################
# TODO:
# Obtain the loss (self.loss) by taking the sum of squares difference
# between the target and prediction Q values.
################################################################################
pass
################################################################################
#                                 END OF YOUR CODE
################################################################################
```

```
        self.trainer = tf.train.AdamOptimizer(learning_rate=0.0001)
        self.updateModel = self.trainer.minimize(self.loss)
```

1.0.7 Training the network

Setting all the training parameters

```
In [ ]: batch_size = 32 #How many experiences to use for each training step.
        update_freq = 4 #How often to perform a training step.
        y = .99 #Discount factor on the target Q-values
        startE = 1 #Starting chance of random action
        endE = 0.1 #Final chance of random action
        annealing_steps = 10000. #How many steps of training to reduce startE to endE.
        num_episodes = 5000 #How many episodes of game environment to train network with.
        pre_train_steps = 10000 #How many steps of random actions before training begins.
        max_epLength = 50 #The max allowed length of our episode.
        load_model = False #Whether to load a saved model.
        path = "./dqn" #The path to save our model to.
        h_size = 512 #The size of the final convolutional layer before splitting it into Advan
        tau = 0.001 #Rate to update target network toward primary network
```

1.0.8 Double DQN

The main intuition behind Double DQN is that the regular DQN often overestimates the Q-values of the potential actions to take in a given state. While this would be fine if all actions were always overestimates equally, there was reason to believe this wasn't the case. You can easily imagine that if certain suboptimal actions regularly were given higher Q-values than optimal actions, the agent would have a hard time ever learning the ideal policy. In order to correct for this, the authors of DDQN paper propose a simple trick: instead of taking the max over Q-values when computing the target-Q value for our training step, we use our primary network to chose an action, and our target network to generate the target Q-value for that action. By decoupling the action choice from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably. Below is the new DDQN equation for updating the target value.

$$Q\text{-}target = r + Q(s', \arg\max(Q(s, a, \theta), \theta'))$$

```
In [ ]: tf.reset_default_graph()
        mainQN = Qnetwork(h_size)
        targetQN = Qnetwork(h_size)

        init = tf.global_variables_initializer()

        saver = tf.train.Saver()

        trainables = tf.trainable_variables()

        targetOps = updateTargetGraph(trainables,tau)

        myBuffer = experience_buffer()
```

```
#Set the rate of random action decrease.
e = startE
stepDrop = (startE - endE)/annealing_steps

#create lists to contain total rewards and steps per episode
jList = []
rList = []
total_steps = 0

#Make a path for our model to be saved in.
if not os.path.exists(path):
    os.makedirs(path)

with tf.Session() as sess:
    sess.run(init)
    if load_model == True:
        print('Loading Model...')
        ckpt = tf.train.get_checkpoint_state(path)
        saver.restore(sess,ckpt.model_checkpoint_path)
    for i in range(num_episodes):
        episodeBuffer = experience_buffer()
        #Reset environment and get first new observation
        s = env.reset()
        s = processState(s)
        d = False
        rAll = 0
        j = 0
        #The Q-Network
        while j < max_epLength: #If the agent takes longer than 50 moves to reach eith
            j+=1
            #Choose an action by greedily (with e chance of random action) from the Q-
            if np.random.rand(1) < e or total_steps < pre_train_steps:
                a = np.random.randint(0,4)
            else:
                a = sess.run(mainQN.predict,feed_dict={mainQN.scalarInput:[s]})[0]
            total_steps += 1

            ##########################################################################
            # TODO: Save the experience to our episode buffer.
            # You will need to do the following:
            # (1) Get new state s1 (resized), reward r and done d from a
            # (2) Add experience to episode buffer. Hint: experience includes
            #     s, a, r, s1 and d.
            ##########################################################################
            pass
            ##########################################################################
            #                                 END OF YOUR CODE
```

```
                ##########################################################################

                if total_steps > pre_train_steps:
                    if e > endE:
                        e -= stepDrop

                    if total_steps % (update_freq) == 0:

                        ##########################################################################
                        # TODO: Implement Double-DQN
                        # (1) Get a random batch of experiences via experience_buffer clas
                        #
                        # (2) Perform the Double-DQN update to the target Q-values
                        #     Hint: Use mainQN and targetQN separately to chose an action
                        #     the Q-values for that action.
                        #     Then compute targetQ based on Double-DQN equation
                        #
                        # (3) Update the primary network with our target values
                        ##########################################################################
                        pass
                        ##########################################################################
                        #                                    END OF YOUR CODE
                        ##########################################################################

                        updateTarget(targetOps,sess) #Update the target network toward the
                rAll += r
                s = s1

                if d == True:

                    break

            myBuffer.add(episodeBuffer.buffer)
            jList.append(j)
            rList.append(rAll)
            #Periodically save the model.
            if i % 2000 == 0:     # i % 1000 == 0:
                saver.save(sess,path+'/model-'+str(i)+'.ckpt')
                print("Saved Model")
            if len(rList) % 10 == 0:
                print("Episode",i,"reward:",np.mean(rList[-10:]))
        saver.save(sess,path+'/model-'+str(i)+'.ckpt')
    print("Mean reward per episode: " + str(sum(rList)/num_episodes))
```

It takes about 40 minutes to train 5000 episodes in Lab 4 machines. Mean reward per episode (50 steps) should be around 20

1.0.9 Checking network learning

Mean reward over time

```
In [ ]: rMat = np.resize(np.array(rList),[len(rList)//100,100])
        rMean = np.average(rMat,1)
        plt.plot(rMean)
```

1.1 Inline Question:

Try a basic DQN without Dueling DQN and Double DQN (i.e. only one single network, no decomposition of the Q-function). You don't need to provide detailed source, just some quantitative comparison is OK (e.g. by comparing the mean reward). **

Your answer: *Fill this in*

```
In [ ]:
```

Q_learning_Basic

January 13, 2021

1 Basic Q-Learning Algorithms

In this exercise we are going be exploring a family of RL algorithms called Q-Learning algorithms. You will begin by implementing a simple lookup-table version of the algorithm, and then a neural-network equivalent using Tensorflow.

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

1.1 OpenAI Gym Environment

For this exercise we will use the FrozenLake environment from the OpenAI gym as a toy example. For those unfamiliar, the OpenAI gym provides an easy way for people to experiment with their learning agents in an array of provided toy games. The FrozenLake environment consists of a `4 x 4` grid of blocks, each one either being the start block `S`, the goal block `G`, a safe frozen block `F`, or a dangerous hole `H`. The objective is to have an agent learn to navigate from the start to the goal without moving onto a hole. At any given time the agent can choose to move either up, down, left, or right. The catch is that there is a wind which occasionally blows the agent onto a space they didn't choose. As such, perfect performance every time is impossible, but learning to avoid the holes and reach the goal are certainly still doable. The reward at every step is 0, except for entering the goal, which provides a reward of 1. Thus, we will need an algorithm that learns long-term expected rewards. This is exactly what Q-Learning is designed to provide.

1.2 Install OpenAI Gym

The provided Python package already contains basic OpenAI gym lib to work on the whole project so you don't have to install it by yourself. Otherwise, to install the OpenAI gym, simply use `pip install gym` to grab it.

1.3 Load the environment

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

For more information, please refer to OpenAI documentation

1.4 Part 1 - Q-Table learning algorithm

In it's simplest implementation, Q-Learning is a table of values for every state (row) and action (column) possible in the environment. Within each cell of the table, we learn a value for how good it is to take a given action within a given state. In the case of the FrozenLake environment, we have 16 possible states (one for each block), and 4 possible actions (the four directions of movement), giving us a 16 x 4 table of Q-values. We start by initializing the table to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly.

We make updates to our Q-table using something called the Bellman equation, which states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state. In equation form, the rule looks like this (Equation 1):

$$Q(s,a) = r + (\max(Q(s,a))$$

This says that the Q-value for a given state (s) and action (a) should represent the current reward (r) plus the maximum discounted (λ) future reward expected according to our own table for the next state (s') we would end up in.

In [ ]:
```
#Initialize table, with states as rows and actions (up, down, left, or right) as colum
Q = np.zeros([env.observation_space.n,env.action_space.n])
#Set learning parameters
lr = .8
#Set discounted factor
y = .95
num_episodes = 2000
#create lists to contain total rewards and steps per episode
rList = []
for i in range(num_episodes):
    #Reset environment and get first new observation
    s = env.reset()
    #Total reward in one episode
    rAll = 0
    d = False
    j = 0
    while j < 99:
        j+=1
        ###############################################################################
        # TODO: Implement the Q-Table learning algorithm.                             #
        # You will need to do the following:                                          #
        # (1) Choose an action by greedily (with noise) picking from Q table given s  #
        #     as input.                                                               #
        # (2) Get new state s1, reward r and done d from environment                  #
        # (3) Update Q-Table with new knowledge.                                      #
        # (4) Cumulate the total reward rAll                                          #
        # (5) Update s                                                                #
        # Note: You may use the gym interfaces env.action_space, env.step etc.        #
        #       E.g. observation, reward, done, info = env.step(action)               #
        #       Please refer to the docs for more information.                        #
```

```
            #       For (1), consider adding noise as a mean of encouraging exploration.  #
            #       For (3), calculate the new target Q-value using Bellman equation.     #
            #       Instead of directly updating toward it, we take a small step in the   #
            #       direction that will make the Q value closer to the target, i.e. use   #
            #       learning rate that controls how much of the difference between        #
            #       newly proposed Q-value and previous Q-value                           #
            ########################################################################
            pass
            ########################################################################
            #                               END OF YOUR CODE                       #
            ########################################################################

            #end of one episode
            if d == True:
                break
        rList.append(rAll)
```

The score is around 0.5 after 2000 episodes.

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

In [ ]: print("Final Q-Table Values")
        print(Q)

In [ ]: # print out the 4 x 4 grid and the current position of the agent
        env.render()
```

1.5 Inline Question 1:

In TODO(3), why not directly apply the Bellman equation for updating the Q value? (in this case lr = 1 and why?)

Your answer: *Fill this in*

1.6 Inline Question 2:

An optimal Q table will tell you the true expected discounted reward for any action given any state. If you find the maximum value of the learned table is not what you believe it should be, do you think it still make sense? Explain briefly.**

Your answer: *Fill this in*

1.7 Part 2 - Q-Network Approach

While it is easy to have a 16x4 table for a simple grid world, the number of possible states in any modern game or real-world environment is nearly infinitely larger. For most interesting problems, tables simply don't work. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. By acting as a function approximator, we can take any number of possible states that can be represented as a vector and learn to map them to Q-values.

In the case of the FrozenLake example, we will be using a one-layer network which takes the state encoded in a one-hot vector (`1x16`), and produces a vector of 4 Q-values, one for each action. Such a simple network acts kind of like a glorified table, with the network weights serving as the old cells. The key difference is that we can easily expand the Tensorflow network with added layers, activation functions, and different input types, whereas all that is impossible with a regular table. The method of updating is a little different as well. Instead of directly updating our table, with a network we will be using backpropagation and a loss function. Our loss function will be sum-of-squares loss, where the difference between the current predicted Q-values, and the "target" value is computed and the gradients passed through the network. **In this case, our Q-target value for the chosen action is the equivalent to the Q-value computed in equation 1 above.**

1.7.1 Implementing the network itself

```
In [ ]: import tensorflow as tf
        env = gym.make('FrozenLake-v0')
        tf.reset_default_graph()

In [ ]: #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 a
        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)
```

1.7.2 Training the network

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

        # Set learning parameters
        #discounted factor
        y = .99
        #chance of random action
        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()
```

```
            #Total reward in one episode
            rAll = 0
            d = False
            j = 0
            #The Q-Network
            while j < 99:
                j+=1

                ##########################################################################
                # TODO: Implement the Q-network approach.
                # You will need to do the following:
                # (1) Choose an action by greedily (with e chance of random action, e=0.1)
                #     from the Q-network
                # (2) Get new state s1, reward r and done d from environment
                # (3) Obtain the Q' values by feeding the new state through our network
                # (4) Obtain maxQ' and set our target value for chosen action.
                # (5) Train our network using target and predicted Q values
                # (6) Cumulate the total reward rAll
                # (7) Update observation s
                # Note: In (1) we need to feed a one-hot vector encoding the state space t
                #       our network. The environment represents the position in the grid-
                #       world as a number between 0 and 15, e.g. if s=11, the one-hot vect
                #       (here is inputs1) should be
                #       [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
                ##########################################################################
                pass
                ##########################################################################
                #                               END OF YOUR CODE
                ##########################################################################

                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)
            if len(rList) % 10 == 0:
                print("Episode",i,"reward:",np.mean(rList[-10:]))
        print("Percent of succesful episodes: " + str(sum(rList)/num_episodes) + "%")
```

1.7.3 Some statistics on network performance

We can see that the network beings to consistly reach the goal around the 750 episode mark.

```
In [ ]: plt.plot(rList)
```

It also begins to progress through the environment for longer than chance around the 750 mark as well.

```
In [ ]: plt.plot(jList)
```

While the network learns to solve the FrozenLake problem, it turns out it doesn't do so quite as efficiently as the Q-Table. While neural networks allow for greater flexibility, they do so at the cost of stability when it comes to Q-Learning. There are a number of possible extensions to our simple Q-Network which allow for greater performance and more robust learning. we will be exploring those additions in Exercise 2.

```
In [ ]:
```

Model_Policy_Network

January 13, 2021

1 Model-Based RL

In this exercise you will implement a policy and model network which work in tandem to solve the CartPole reinforcement learning problem.

What is a model and why would we want to use one? In this case, a model is going to be a neural network that attempts to learn the dynamics of the real environment. For example, in the CartPole we would like a model to be able to predict the next position of the Cart given the previous position and an action. By learning an accurate model, we can train our agent using the model rather than requiring to use the real environment every time. While this may seem less useful when the real environment is itself a simulation, like in our CartPole task, it can have huge advantages when attempting to learn policies for acting in the physical world.

How are we going to accomplish this in Tensorflow? We are going to be using a neural network that will learn the transition dynamics between a previous observation and action, and the expected new observation, reward, and done state. Our training procedure will involve switching between training our model using the real environment, and training our agent's policy using the model environment. By using this approach we will be able to learn a policy that allows our agent to solve the CartPole task without actually ever training the policy on the real environment!

1.0.1 Loading libraries and starting CartPole environment

```
In [ ]: from __future__ import print_function
        import numpy as np
        try:
            import cPickle as pickle
        except:
            import pickle
        import tensorflow as tf
        %matplotlib inline
        import matplotlib.pyplot as plt
        import math
```

```
In [ ]: import sys
        if sys.version_info.major > 2:
            xrange = range
        del sys
```

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

1.0.2 Setting Hyper-parameters

In [ ]:
```
# hyperparameters
H = 8 # number of hidden layer neurons
learning_rate = 1e-2
gamma = 0.99 # discount factor for reward
decay_rate = 0.99 # decay factor for RMSProp leaky sum of grad^2
resume = False # resume from previous checkpoint?

model_bs = 3 # Batch size when learning from model
real_bs = 3 # Batch size when learning from real environment

# model initialization
D = 4 # input dimensionality
```

1.0.3 Policy Network

In [ ]:
```
tf.reset_default_graph()
observations = tf.placeholder(tf.float32, [None,4] , name="input_x")
W1 = tf.get_variable("W1", shape=[4, H],
           initializer=tf.contrib.layers.xavier_initializer())
layer1 = tf.nn.relu(tf.matmul(observations,W1))
W2 = tf.get_variable("W2", shape=[H, 1],
           initializer=tf.contrib.layers.xavier_initializer())
score = tf.matmul(layer1,W2)
probability = tf.nn.sigmoid(score)

tvars = tf.trainable_variables()
input_y = tf.placeholder(tf.float32,[None,1], name="input_y")
advantages = tf.placeholder(tf.float32,name="reward_signal")
adam = tf.train.AdamOptimizer(learning_rate=learning_rate)
W1Grad = tf.placeholder(tf.float32,name="batch_grad1")
W2Grad = tf.placeholder(tf.float32,name="batch_grad2")
batchGrad = [W1Grad,W2Grad]

################################################################################
# TODO: Implement the loss function.                                           #
# This sends the weights in the direction of making actions that gave good     #
# advantage (reward overtime) more likely, and actions that didn't less likely.#
################################################################################
pass
################################################################################
#                                END OF YOUR CODE                              #
################################################################################

newGrads = tf.gradients(loss,tvars)
updateGrads = adam.apply_gradients(zip(batchGrad,tvars))
```

1.0.4 Model Network

Here we implement a multi-layer neural network that predicts the next observation, reward, and done state from a current state and action.

In [ ]:
```
mH = 256 # model layer size

        input_data = tf.placeholder(tf.float32, [None, 5])
        with tf.variable_scope('rnnlm'):
            softmax_w = tf.get_variable("softmax_w", [mH, 50])
            softmax_b = tf.get_variable("softmax_b", [50])

        previous_state = tf.placeholder(tf.float32, [None,5] , name="previous_state")
        W1M = tf.get_variable("W1M", shape=[5, mH],
                    initializer=tf.contrib.layers.xavier_initializer())
        B1M = tf.Variable(tf.zeros([mH]),name="B1M")
        layer1M = tf.nn.relu(tf.matmul(previous_state,W1M) + B1M)
        W2M = tf.get_variable("W2M", shape=[mH, mH],
                    initializer=tf.contrib.layers.xavier_initializer())
        B2M = tf.Variable(tf.zeros([mH]),name="B2M")
        layer2M = tf.nn.relu(tf.matmul(layer1M,W2M) + B2M)
        wO = tf.get_variable("wO", shape=[mH, 4],
                    initializer=tf.contrib.layers.xavier_initializer())
        wR = tf.get_variable("wR", shape=[mH, 1],
                    initializer=tf.contrib.layers.xavier_initializer())
        wD = tf.get_variable("wD", shape=[mH, 1],
                    initializer=tf.contrib.layers.xavier_initializer())

        bO = tf.Variable(tf.zeros([4]),name="bO")
        bR = tf.Variable(tf.zeros([1]),name="bR")
        bD = tf.Variable(tf.ones([1]),name="bD")


        predicted_observation = tf.matmul(layer2M,wO,name="predicted_observation") + bO
        predicted_reward = tf.matmul(layer2M,wR,name="predicted_reward") + bR
        predicted_done = tf.sigmoid(tf.matmul(layer2M,wD,name="predicted_done") + bD)

        true_observation = tf.placeholder(tf.float32,[None,4],name="true_observation")
        true_reward = tf.placeholder(tf.float32,[None,1],name="true_reward")
        true_done = tf.placeholder(tf.float32,[None,1],name="true_done")


        predicted_state = tf.concat([predicted_observation,predicted_reward,predicted_done],1)

        observation_loss = tf.square(true_observation - predicted_observation)

        reward_loss = tf.square(true_reward - predicted_reward)

        done_loss = tf.multiply(predicted_done, true_done) + tf.multiply(1-predicted_done, 1-t
```

```
done_loss = -tf.log(done_loss)

model_loss = tf.reduce_mean(observation_loss + done_loss + reward_loss)

modelAdam = tf.train.AdamOptimizer(learning_rate=learning_rate)
updateModel = modelAdam.minimize(model_loss)
```

1.0.5 Helper-functions

```
In [ ]: def resetGradBuffer(gradBuffer):
            for ix,grad in enumerate(gradBuffer):
                gradBuffer[ix] = grad * 0
            return gradBuffer

        def discount_rewards(r):
            ################################################################################
            # TODO: Implement the discounted rewards function                              #
            # Return discounted rewards weighed by gamma. Each reward will be replaced     #
            # with a weight reward that involves itself and all the other rewards occuring #
            # after it. The later the reward after it happens, the less effect it has on   #
            # the current rewards's discounted reward                                      #
            # Hint: [r0, r1, r2, ..., r_N] will look someting like:                        #
            #       [(r0 + r1*gamma^1 + ... r_N*gamma^N), (r1 + r2*gamma^1 + ...), ...]    #
            ################################################################################
            pass
            ################################################################################
            #                                  END OF YOUR CODE                            #
            ################################################################################

        # This function uses our model to produce a new state when given a previous state and
        def stepModel(sess, xs, action):
            toFeed = np.reshape(np.hstack([xs[-1][0],np.array(action)]),[1,5])
            myPredict = sess.run([predicted_state],feed_dict={previous_state: toFeed})
            reward = myPredict[0][:,4]
            observation = myPredict[0][:,0:4]
            observation[:,0] = np.clip(observation[:,0],-2.4,2.4)
            observation[:,2] = np.clip(observation[:,2],-0.4,0.4)
            doneP = np.clip(myPredict[0][:,5],0,1)
            if doneP > 0.1 or len(xs)>= 300:
                done = True
            else:
                done = False
            return observation, reward, done
```

1.1 Training the Policy and Model

```
In [ ]: xs,drs,ys,ds = [],[],[],[]
        running_reward = None
```

```
reward_sum = 0
episode_number = 1
real_episodes = 1
init = tf.global_variables_initializer()
batch_size = real_bs

drawFromModel = False # When set to True, will use model for observations
trainTheModel = True # Whether to train the model
trainThePolicy = False # Whether to train the policy
switch_point = 1

# Launch the graph
with tf.Session() as sess:
    rendering = False
    sess.run(init)
    observation = env.reset()
    x = observation
    gradBuffer = sess.run(tvars)
    gradBuffer = resetGradBuffer(gradBuffer)

    while episode_number <= 5000:
        # Start displaying environment once performance is acceptably high.
        if (reward_sum/batch_size > 150 and drawFromModel == False) or rendering == Tru
            env.render()
            rendering = True

        x = np.reshape(observation,[1,4])

        tfprob = sess.run(probability,feed_dict={observations: x})
        action = 1 if np.random.uniform() < tfprob else 0

        # record various intermediates (needed later for backprop)
        xs.append(x)
        y = 1 if action == 0 else 0
        ys.append(y)

        # step the  model or real environment and get new measurements
        if drawFromModel == False:
            observation, reward, done, info = env.step(action)
        else:
            observation, reward, done = stepModel(sess,xs,action)

        reward_sum += reward

        ds.append(done*1)
        drs.append(reward) # record reward (has to be done after we call step() to get

        if done:
```

```
if drawFromModel == False:
    real_episodes += 1
episode_number += 1

# stack together all inputs, hidden states, action gradients, and rewards
epx = np.vstack(xs)
epy = np.vstack(ys)
epr = np.vstack(drs)
epd = np.vstack(ds)
xs,drs,ys,ds = [],[],[],[] # reset array memory

if trainTheModel == True:

    ##############################################################################
    # TODO: Run the model network and compute predicted_state
    # Output: 'pState'
    ##############################################################################
    pass
    ##############################################################################
    #                                  END OF YOUR CODE
    ##############################################################################


if trainThePolicy == True:

    ##############################################################################
    # TODO: Run the policy network and compute newGrads
    # Output: 'tGrad'
    ##############################################################################
    pass
    ##############################################################################
    #                                  END OF YOUR CODE
    ##############################################################################

    # If gradients becom too large, end training process
    if np.sum(tGrad[0] == tGrad[0]) == 0:
        break
    for ix,grad in enumerate(tGrad):
        gradBuffer[ix] += grad

if switch_point + batch_size == episode_number:
    switch_point = episode_number
    if trainThePolicy == True:

        ##############################################################################
        # TODO:
        # (1) Run the policy network and update gradients
```

```
                # (2) Reset gradBuffer to 0
                ##############################################################
                pass
                ##############################################################
                #                                END OF YOUR CODE
                ##############################################################

            running_reward = reward_sum if running_reward is None else running_rewa
            if drawFromModel == False:
                print('World Perf: Episode %f. Reward %f. action: %f. mean reward %
                if reward_sum/batch_size > 200:
                    break
            reward_sum = 0

            # Once the model has been trained on 100 episodes
            if episode_number > 100:

                ##############################################################
                # TODO: Alternating between training the policy from the model and
                # the model from the real environment.
                ##############################################################
                pass
                ##############################################################
                #                                END OF YOUR CODE
                ##############################################################

        if drawFromModel == True:
            observation = np.random.uniform(-0.1,0.1,[4]) # Generate reasonable st
            batch_size = model_bs
        else:
            observation = env.reset()
            batch_size = real_bs

print(real_episodes)
```

1.1.1 Checking model representation

Here we can examine how well the model is able to approximate the true environment after training. The green line indicates the real environment, and the blue indicates model predictions.

```
In [ ]: plt.figure(figsize=(8, 12))
        for i in range(6):
            plt.subplot(6, 2, 2*i + 1)
            plt.plot(pState[:,i])   # draw the model predictions
            plt.subplot(6,2,2*i+1)

            ##########################################################################
            # TODO: draw the real environment for comparison                         #
```

```
            ################################################################################
            pass
            ################################################################################
            #                                END OF YOUR CODE                              #
            ################################################################################
        plt.tight_layout()
```

In [ ]:

PG_CartPole

January 13, 2021

1 CartPole with Policy Gradient Method

In this exercise you will build a policy-gradient based agent that can solve the classic CartPole task where we must balance a pole on a cart for as long as possible. You will specifically be using the OpenAI Gym in order to have a reactive environment that gives you observations (state) and reward with a given action from your model. For those who are not familiar with OpenAI Gym, please check out the short tutorial to cover the basics of the different game environments.

In [ ]:
```
from __future__ import division

import numpy as np
try:
    import cPickle as pickle
except:
    import pickle
import tensorflow as tf
%matplotlib inline
import matplotlib.pyplot as plt
import math

try:
    xrange = xrange
except:
    xrange = range
import sys
import os
import pyglet
if not pyglet.version == '1.2.4':
    !{sys.executable} -m pip install pyglet==1.2.4
```

1.0.1 Loading the CartPole Environment

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

The observation is represented by a 4-dimension vector, i.e. *[position of cart, velocity of cart, angle of pole, rotation rate of pole]*. Note that good general-purpose agents don't need to know the

semantics of the observations: they can learn how to map observations to actions to maximize reward without any prior knowledge.

First, we will run the task using random actions. The cart can either move left or right in order to balance the pole. We will randomly choose which direction to move the cart.

1.1 Inline Question 1:

How well do we do with random action? (Hint: not so well.)

```
In [ ]: env.reset()
        random_episodes = 0
        reward_sum = 0
        while random_episodes < 10:
            env.render()
            observation, reward, done, _ = env.step(np.random.randint(0,2))
            reward_sum += reward
            if done:
                random_episodes += 1
                print("Reward for this episode was:",reward_sum)
                reward_sum = 0
                env.reset()
```

The goal of the task is to achieve a reward of 200 per episode. For every step the agent keeps the pole in the air, the agent recieves a +1 reward. By randomly choosing actions, our reward for each episode is only a couple dozen. Let's make that better with RL!

1.1.1 Setting up our Neural Network agent

This time we will be using a Policy neural network that takes observations, passes them through a single hidden layer, and then produces a probability of choosing a left/right movement. To learn more about this network, see Andrej Karpathy's blog on Policy Gradient networks.

```
In [ ]: # hyperparameters
        H = 10 # number of hidden layer neurons
        batch_size = 5 # every how many episodes to do a param update?
        learning_rate = 1e-2 # feel free to play with this to train faster or more stably.
        gamma = 0.99 # discount factor for reward

        D = 4 # input dimensionality
```

```
In [ ]: tf.reset_default_graph()

        #This defines the network as it goes from taking an observation of the environment to
        #giving a probability of chosing to the action of moving left or right.
        observations = tf.placeholder(tf.float32, [None,D] , name="input_x")
        W1 = tf.get_variable("W1", shape=[D, H],
                   initializer=tf.contrib.layers.xavier_initializer())
        layer1 = tf.nn.relu(tf.matmul(observations,W1))
        W2 = tf.get_variable("W2", shape=[H, 1],
```

```
            initializer=tf.contrib.layers.xavier_initializer())
score = tf.matmul(layer1,W2)
probability = tf.nn.sigmoid(score) # prediction of action

#From here we define the parts of the network needed for learning a good policy.
tvars = tf.trainable_variables()
input_y = tf.placeholder(tf.float32,[None,1], name="input_y") # label action 0,1
advantages = tf.placeholder(tf.float32,name="reward_signal")

# The loss function. This sends the weights in the direction of making actions
# that gave good advantage (reward over time) more likely, and actions that didn't les
loglik = tf.log(input_y*(input_y - probability) + (1 - input_y)*(input_y + probability)
loss = -tf.reduce_mean(loglik * advantages)
newGrads = tf.gradients(loss,tvars)

# Once we have collected a series of gradients from multiple episodes, we apply them.
# We don't just apply gradeients after every episode in order to account for noise in
adam = tf.train.AdamOptimizer(learning_rate=learning_rate) # Our optimizer
W1Grad = tf.placeholder(tf.float32,name="batch_grad1") # Placeholders to send the fina
W2Grad = tf.placeholder(tf.float32,name="batch_grad2")
batchGrad = [W1Grad,W2Grad]
updateGrads = adam.apply_gradients(zip(batchGrad,tvars))
```

1.1.2 Advantage function

This function allows us to weigh the rewards our agent recieves. In the context of the Cart-Pole task, we want actions that kept the pole in the air a long time to have a large reward, and actions that contributed to the pole falling to have a decreased or negative reward. We do this by weighing the rewards from the end of the episode, with actions at the end being seen as negative, since they likely contributed to the pole falling, and the episode ending. Likewise, early actions are seen as more positive, since they weren't responsible for the pole falling.

```
In [ ]: def discount_rewards(r):
            ##############################################################################
            # TODO: Implement the discounted rewards function                            #
            # Return discounted rewards weighed by gamma. Each reward will be replaced   #
            # with a weight reward that involves itself and all the other rewards occuring #
            # after it. The later the reward after it happens, the less effect it has on #
            # the current rewards's discounted reward                                    #
            # Hint: [r0, r1, r2, ..., r_N] will look someting like:                      #
            #       [(r0 + r1*gamma^1 + ... r_N*gamma^N), (r1 + r2*gamma^1 + ...), ...]  #
            ##############################################################################
            pass
            ##############################################################################
            #                                END OF YOUR CODE                            #
            ##############################################################################
```

1.1.3 Running the Agent and Environment

Here we run the neural network agent, and have it act in the CartPole environment.

```
In [ ]: xs,hs,dlogps,drs,ys,tfps = [],[],[],[],[],[]
        running_reward = None
        reward_sum = 0
        episode_number = 1
        total_episodes = 10000
        init = tf.global_variables_initializer()

        # Launch the graph
        with tf.Session() as sess:
            rendering = False
            sess.run(init)
            observation = env.reset() # Obtain an initial observation of the environment

            # Reset the gradient placeholder. We will collect gradients in
            # gradBuffer until we are ready to update our policy network.
            gradBuffer = sess.run(tvars)
            for ix,grad in enumerate(gradBuffer):
                gradBuffer[ix] = grad * 0

            while episode_number <= total_episodes:

                # Rendering the environment slows things down,
                # so let's only look at it once our agent is doing a good job.
                if reward_sum/batch_size > 100 or rendering == True :
                    env.render()
                    rendering = True

                # Make sure the observation is in a shape the network can handle.
                x = np.reshape(observation,[1,D])


                ##############################################################################
                # TODO: Run the policy network and get an action to take
                # Output: action
                ##############################################################################
                pass
                ##############################################################################
                #                                END OF YOUR CODE
                ##############################################################################

                xs.append(x) # observation
                y = 1 if action == 0 else 0 # a "fake label"
                ys.append(y)

                ##############################################################################
```

```
# TODO: Step the environment and get new measurements
# Output: observation, reward, done
################################################################################
pass
################################################################################
#                                 END OF YOUR CODE
################################################################################

reward_sum += reward

drs.append(reward) # record reward (has to be done after we call step() to get

if done:
    episode_number += 1
    # stack together all inputs, hidden states, action gradients, and rewards
    epx = np.vstack(xs)
    epy = np.vstack(ys)
    epr = np.vstack(drs)
    tfp = tfps
    xs,hs,dlogps,drs,ys,tfps = [],[],[],[],[],[] # reset array memory

    # compute the discounted reward backwards through time
    discounted_epr = discount_rewards(epr)
    # size the rewards to be unit normal (helps control the gradient estimator
    discounted_epr -= np.mean(discounted_epr)
    discounted_epr //= np.std(discounted_epr)


    ############################################################################
    # TODO: Run the policy network and get the gradients for this episode
    # Output: tGrad
    ############################################################################
    pass
    ############################################################################
    #                                 END OF YOUR CODE
    ############################################################################

    # Save gradients in the gradBuffer
    for ix,grad in enumerate(tGrad):
        gradBuffer[ix] += grad

    # If we have completed enough episodes
    if episode_number % batch_size == 0:

        ########################################################################
        # TODO: Update the policy network with our gradients and set gradBuffe
        ########################################################################
        pass
```

```
                    ##############################################################
                    #                                END OF YOUR CODE
                    ##############################################################

                    # Give a summary of how well our network is doing for each batch of ep
                    running_reward = reward_sum if running_reward is None else running_rewa
                    print('Average reward for episode %f.  Total average reward %f.' % (rew

                    if reward_sum//batch_size >= 200:
                        print("Task solved in",episode_number,'episodes!')
                        break

                    reward_sum = 0

                observation = env.reset()

    print(episode_number,'Episodes completed.')
```

As you can see, the network not only does much better than random actions, but achieves the goal of 200 points per episode, thus solving the task!

In [ ]: