ID: A1825
Date of submission: 23 May 2019

1 Balancing a Cartpole Using Deep Q-Network

We will develop a Deep Q network from scratch to balancing a cartpole using tensorflow.

1.1 Objective of environment

A pole is attaced by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright and the goal is to prevent it from falling over. The cart can move left or right by applying force on left or right direction.

1.2 Import necessary libraries

```
In [1]: import gym
        import random
        import numpy as np
        import tensorflow as tf
        from collections import deque
```

1.3 Import environment

```
In [2]: env = gym.make("CartPole-v0")
```

1.4 Initialization and rendering

```
In [3]: print('Action space {}'.format(env.action_space))
        print('State space {}'.format(env.observation_space))
        init_state = env.reset()
        print('Initial space {}'.format(init_state))
```

```
Action space Discrete(2)
State space Box(4,)
Initial space [0.01667933 0.02495509 0.00306789 0.02814474]
```

From the above we can see that, Action space is 2 because cartpole can move in two direction: left or right. The discrete space allows a fixed range of non-negative values.

The state space represents an n-dimentional box, so valid observatio will be an array of 4 numbers. The cartpole environment consists of 4 state (shows value in initial state result): Cart position (From the position of frame), Cart velocity (direction that it's travelling), Pole angle (from center axis) and Pole velocity at tip (angular velocity of pole).

Let's check state space size (set by environment gym):

Information	Minimum	Maximum
Cart position	-2.4	2.4
Cart velocity	-Infinity	Infinity
Pole angle	~-41.8 degree	~41.8 degree
Pole velocity at tip	-Infinity	Infinity

So state size is infinity.

The episode is over when cart postion and cart velocity is more than the range or episode length is greater than 200.

Reward is +1 for every step.

1.5 Initial agent

In [4]:
```
epochs, total_rewards = 0, 0
num_episodes = 100 #we are running for 100 times
current_episode = 0
while current_episode < num_episodes:
    reward = 0
    done = False
    while not done:

        #picking up a random action from action space and current position
```

```
            action = env.action_space.sample()

            #executing that action. This returns us next stage, the reward we got from taking that step, boolean value
            #which indicates whether our episode is done or not and some additional debugging info
            state, reward, done, info = env.step(action)

            epochs += 1

            #if the reward is -10, that means we have taken a wrong action and number of penalty is incremented by 1
            total_rewards += reward
        current_episode += 1
        env.reset()
    print("Reward achieved {}".format(total_rewards/float(num_episodes)))
    print("Avg time steps per episode {}".format(epochs/float(num_episodes)))
```

```
Reward achieved 21.81
Avg time steps per episode 21.81
```

Avg reward is very poor because the maximum reward we can from environment is 200 and also time steps are much more per episode

1.6 Build a Deep Q Network

1.6.1 Neural network architecture

A Deep Q Network is a deep neural network to model the q learning function for reinforcement learning. As the reward of environment is unknown to agent, it needs to explore environment rewards from taking different actions in various states in order build q network. It follows this equation:

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

So to model this function with q network, we need to approximate two outputs: values for each action in given state and individual q value for a single action in that state vector. So to build our network architecture we start with the state vector which we will pass through a dense layer acting as the hidden layer with a certain number of hidden nodes. Then we can pass that layer to another layer to transform it to have an output length to match the number of action available. This will be our vector of Q values for the input state. Then for the Q value for a single input action, we just want the value at the index corresponding to that action which we can get by multiplying this vector with an one hot coded action vector whichh has a 1 in the index of action and 0 elsewhere. Because as we have seen from about that action space is discrete and it has two value: 0 or 1. Then we get the sum to get a single value. So now, with our target value calculated from the above equation, we can calculate the mean squared error of the network's output with the target to get our loss.

In [5]:
```
"""Now we need to build our deep neural network model class."""
class DeepQNetwork():

    #this function takes state dimension as network size and action size as network output size
    def __init__(self, state_dimension, action_size):

        #initializing input layer
        self.state_input = tf.placeholder(tf.float32, shape=[None, *state_dimension])

        #initializing action layer
        self.action_input = tf.placeholder(tf.int32, shape=[None])
```

```python
#initializing target q
self.q_target_input = tf.placeholder(tf.float32, shape=[None])

#converting action to one hot encoded vector
action_one_hot_encoded = tf.one_hot(self.action_input, depth=action_size)


#we are initializing our hidden layer and then passing in the state to a dense layer
#with a 100 hidden units and ReLU as activation function
#because ReLU is sparsity and reduced likelihood of vanishing gradient.
self.hidden_layer1 = tf.layers.dense(self.state_input, 100, activation=tf.nn.relu)

#getting the q value for each action in a state by passing it to another dense layer
#outputting action size unit
self.q_state = tf.layers.dense(self.hidden_layer1, action_size, activation=None)

#then for our single Q value for state action comes from multiplying states Q values with the
#one hot action vector and then reducing this to a single value summing across the columns
self.q_state_action = tf.reduce_sum(tf.multiply(self.q_state, action_one_hot_encoded), axis=1)

#our loss is the squared difference between the predicted q state action and q target
#Q target will be averaged of out of the batch
self.loss = tf.reduce_mean(tf.square(self.q_state_action - self.q_target_input))

#for our optimizer,we are using adam optimizer with learning rate 0.001
self.optimizer = tf.train.AdamOptimizer(learning_rate=0.001).minimize(self.loss)
```

```python
    def update_dqnmodel(self, session, state, action, q_target):
        feed = {self.state_input: state, self.action_input: action, self.q_target_input: q_target}
        session.run(self.optimizer, feed_dict=feed)

    #to get the q state output which take tf session and state as parameter
    def obtain_q_state(self, session, state):
        q_state = session.run(self.q_state, feed_dict={self.state_input: state})
        return q_state
```

In [6]:
```python
"""To stablize the model training time and result, we need to stable our model. So for that we can implement
experience replay."""

class ExperienceReplay():

    #this function takes maximum of the buffer using collection's deque
    def __init__(self, maxlen):
        self.buffer = deque(maxlen=maxlen)

    #then we create a function to add experience to the buffer so that we can stable the time and result
    def add(self, experience):
        self.buffer.append(experience)


    #then we create a function to sample the batch of experience tuples
    def sample(self, batch_size):

        #saving batch size or minimum length of buffer in sample in case we have less buffer size then batch
```

```
            sample_size = min(len(self.buffer), batch_size)

            #getting list randomly from buffer
            sample = random.choices(self.buffer, k=sample_size)

            #to get all the result in list we unpack the sample and map it
            return map(list, zip(*sample))
```

In [7]:
```
"""Defining agent class"""
class DeepQNetworkAgent():
    def __init__(self, env):

        #getting the state size from environment
        self.state_dimension = env.observation_space.shape

        #getting the action size from the environment
        self.action_size = env.action_space.n

        #initialing instance of deep q network
        self.deep_q_network = DeepQNetwork(self.state_dimension, self.action_size)
        self.experience_replay = ExperienceReplay(maxlen=10000)
        self.gamma = 0.97
        self.epsilon = 1.0

        #initializing session
        self.sess = tf.Session()

        #initializing global variables
```

```python
        self.sess.run(tf.global_variables_initializer())

    #to get the updated action for given state using deep q network
    #so agent needs to predict the action with the highest predicted q value
    def get_action(self, state):

        #getting updated q state for certain action
        q_state = self.deep_q_network.obtain_q_state(self.sess, [state])

        #getting the highest q value for that state
        action_highest_q_value = np.argmax(q_state)

        #if we don't use epsilon model will select one action for each state and won't explore other actions
        #For that, we tell our agent to initially explore the environment by selecting actions randomly earlier
        #in training and gradually selecting the action greedily more often as the Q network moves closer to
        #true estimate

        action_random = np.random.randint(self.action_size)
        action = action_random if random.random() < self.epsilon else action_highest_q_value

        return action

    #target the q value and train the network
    def trainAgent(self, state, action, next_state, reward, done):

        #adding experience in buffer
        self.experience_replay.add((state, action, next_state, reward, done))
```

```
            #getting list of each experience type sampling from the buffer
            states, actions, next_states, rewards, dones = self.experience_replay.sample(50)

            #getting the q for next state
            q_next_state = self.deep_q_network.obtain_q_state(self.sess, next_states)

            #adjustment if there is no next state after the terminal state
            q_next_state[dones] = np.zeros([self.action_size])

            #calculate targetted q value using mentioned equation
            q_target = rewards + self.gamma * np.max(q_next_state, axis=1)

            #update the model
            self.deep_q_network.update_dqnmodel(self.sess, states, actions, q_target)

            #we need to decrease epsilon after each episode because we want our model to trust its learning more
            #after some learning
            if done:
                self.epsilon = max(0.1, 0.99*self.epsilon)

        #closing tensorflow session
        def __del__(self):
            self.sess.close()
```

In [8]:
```
dqn_agent = DeepQNetworkAgent(env)
num_episodes = 400 #running for 400 episodes
epochs = 0
```

```
for ep in range(num_episodes):
    state = env.reset()
    total_reward = 0
    done = False
    while not done:

        #getting optimal action
        action = dqn_agent.get_action(state)

        #getting all info
        next_state, reward, done, info = env.step(action)

        #training the agent
        dqn_agent.trainAgent(state, action, next_state, reward, done)
#         env.render()
        total_reward += reward
        state = next_state
        epochs += 1
    #if our reward is more than 195 from 200, we will break the loop
    if total_reward > 195:
        print("Solved!")
        break
```

```
Solved!
```

In [9]:
```
print("Reward achieved {}".format(total_reward))
print("Avg time steps per episode {}".format(epochs/float(num_episodes)))
```

```
Reward achieved 200.0
Avg time steps per episode 9.86
```

As we can see, we have achieved highest reward with a few time steps using Deep Q Network.

1.7 Discussion

1.7.1 Compare agent performance

	Initial Agent	**Deep Q-Network Agent**
Time steps	21.81	9.86
Reward achieved	21.81	200.0

As we can see from this performance table that, our agent learns better after implementing the Deep Q-Network method.

1.7.2 Improvement

- Hyperparameter tuning

1.8 Reference

- https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288
- https://dev.to/n1try/cartpole-with-a-deep-q-network