Iowa State University

Department of Electrical and Computer Engineering

Deep Machine Learning: Theory and Practice

EE 526X

Final Project Report

Author:
Vishal Deep

Instructor:
Dr. Zhengdao Wang

December 20, 2019

IOWA STATE UNIVERSITY

1 Problem Statement

The main goal of the project is to design and optimize a reinforcement learning algorithm using double Q-learning on Acrobot-v1 environment from Open AI gym package [1]. The function approximation should be done with multi-layer neural networks.

Acrobot-v1

The Acrobot-v1 environment consists of two joints. One joint is fixed and other joint is actuated as shown in fig. 1. In the initial stage the joints are hanging downwards and the goal is to make lower joint reach a certain given height. This environment is unsolved environment which means it does not have a specific reward above which we can say that it is solved.

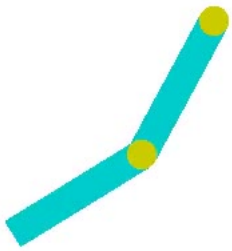

Figure 1: Acrobot-v1 from Open AI gym package

- **State**: In this environment states consists of sin and cos of the two rotational joint angles and their angular velocities. A complete state is
  [cos(theta1) sin(theta1) cos(theta2) sin(theta2) thetaDot1 thetaDot2].
- **Action**: There are three possible actions in this environment. The action is either applying +1, 0 or -1 torque on the joint between the two links.

Deep Q-learning

In the Q-learning algorithm, a virtual table of action and Q-values is used. This table could grow really fast depending on the states and actions. Therefore a better solution is to create Neural Networks (NNs) which can approximate the Q-values for every action. This is called deep Q-learning.

In case of Q-learning, taking the maximum overestimated values introduces a maximization bias in learning. To solve this problem, we use a double Q-learning algorithm which uses two separate Q-value estimators. These estimators are used to update each other and thereby providing unbiased estimation of Q-values. In this project, I will be using double Q-learning function estimators using neural networks.

2 Neural Network Model

In this project, I have used the neural network shown in fig. 2. This NN has three fully connected dense layers. The first layer consists of 48 neurons, the second one has 24 neurons, and last layer consists of 3 neurons to signify the 3 output actions. The total number of parameters are 1587.

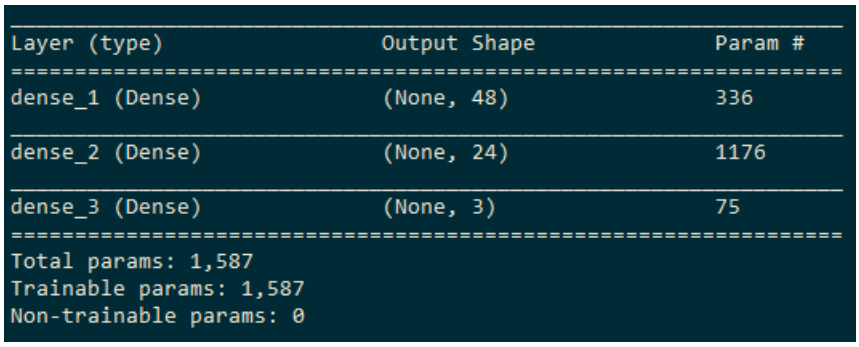

Figure 2: Neural Network Model used for function approximation in double Q-learning algorithm

3 Pseudo-code for the Algorithm

Algorithm 1 Double Q-learning Algorithm [2]

Initialize primary Q_A and target network Q_B
for each episode **do**
  for each step **do**
    Choose action a_t which maximize q-value
    Execute a_t and observe State S_{t+1} and Reward R_t
    Save (S_t, a_t, R_t, S_{t+1}) to memory
    for samples in memory **do**
      Train primary Neural Network Q_A using samples
      Compute Target Q-value
      $Q(s_t, at) \approx R_t + \gamma Q(s_{t+1}, argmax_{a'} Q^{'}(s_t, a_t))$
    end for
    Update target network Q_B at each update interval:
    $\theta^{'} \leftarrow \tau * \theta + (1 - \tau) * \theta^{'}$
  end for
end for

4 Python Code

The code is inspired from several tutorials on the internet including [3, 4]

```
import random
import gym
import time
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

class DQN:
    # Initialize env and all parameters
    def __init__(self, env):
        self.env =env
        self.state_size =env.observation_space.shape[0]
        self.action_size =env.action_space.n
        self.memory =deque(maxlen=100000)
        self.gamma =0.99
        self.epsilon =0.5
        self.epsilon_min =0.001
        self.epsilon_decay =0.9
        self.learning_rate =1e-3
        self.batch_size =64
        self.tau =1e-3
        # Create model
        # Two seperate models, one for doing predictions
        # and other for tracking target values
        self.model =self.create_model()
        self.target_model =self.create_model()

    # Neural Network model for function approximation
    def create_model(self):
        model =Sequential()
        state_shape =self.env.observation_space.shape
        model.add(Dense(48, input_dim=state_shape[0], activation="relu"))
        model.add(Dense(24, activation="relu"))
        # model.add(Dense(10, activation="relu"))
        model.add(Dense(self.env.action_space.n))
        model.compile(loss="mean_squared_error", optimizer=Adam(lr=self.learning_rate))
        return model

    # add learned states,action, and rewards to the list
    def save(self, state, action, reward, next_state, done):
        self.memory.append([state, action, reward, next_state, done])

    def train(self):
        # if memory is smaller then do nothing
        if len(self.memory) <self.batch_size:
            return

        # Take a random samples from the memory
        samples =random.sample(self.memory, self.batch_size)
```

```
        for sample in samples:
            state, action, reward, next_state, done =sample
            target =self.target_model.predict(state)
            # If at the end of trials, there are no future rewards
            if done:
                target[0][action] =reward
            else:
                Q_future =np.max(self.target_model.predict(next_state)[0])
                target[0][action] =reward +Q_future *self.gamma
            # Fit the model
            self.model.fit(state, target, epochs=1, verbose=0)

    # Updates the weights in target NN
    def target_train(self):
        # Get learned weights
        weights =self.model.get_weights()
        target_weights =self.target_model.get_weights()
        for i in range(len(target_weights)):
            target_weights[i] =weights[i] *self.tau +target_weights[i] *(1 -self.tau)
        self.target_model.set_weights(target_weights)
        # Update the epsilon to lower exploration rate
        self.epsilon *=self.epsilon_decay
        self.epsilon =max(self.epsilon_min, self.epsilon)

    # Chooses action
    def action(self, state):
        # # Update the epsilon to lower exploration rate
        # self.epsilon *= self.epsilon_decay
        # self.epsilon = max(self.epsilon_min, self.epsilon)
        if np.random.rand() <self.epsilon:
            #Choose random action
            return self.env.action_space.sample()
        # Else choose max value action
        return np.argmax(self.model.predict(state)[0])

    # Load weights
    def load_weights(self, name):
        self.model.load_weights(name)
    # Save weights
    def save_weights(self, name):
        self.model.save_weights(name)

def main(render=False):
    env =gym.make('Acrobot-v1')
    # state_size = env.observation_space.shape[0]
    # # action_size = env.action_space.n
    DQNAgent =DQN(env)
    state_size =DQNAgent.state_size
    done =False
    episodes =100
    trial_len =500

    # Print Neural Network model summary
    DQNAgent.model.summary()

    # Capture training start time
```

```
    startTime =np.round(time.time(), decimals=4)
    for episode in range(episodes):
        rewards =[]
        # reset environment
        state =env.reset()
        state =np.reshape(state, [1, state_size])

        for step in range(trial_len):
            if render:
                env.render()
            # Choose action
            action =DQNAgent.action(state)
            # Get next state and reward
            next_state, reward, done, _ =env.step(action)
            # reward = reward if not done else -10
            next_state =np.reshape(next_state, [1, state_size])
            DQNAgent.save(state, action, reward, next_state, done)

            # Train prediction Neural Networks
            DQNAgent.train()
            # Update target NN periodically
            if step%50 ==0:
                # print("Updating target Network")
                DQNAgent.target_train()
            # Append Rewards
            rewards.append(reward)
            # Update state
            state =next_state
            # Calculate score
            score =trial_len-step-1
            if score >=350:
                DQNAgent.save_weights("best-weights")
            if done:
                print("episode: {}, score: {}, epsilon: {:.6}, Rewards: {}"
                      .format(episode, trial_len-step-1, DQNAgent.epsilon, np.sum(rewards)))
                # print(f"Episode:{episode},Score:{trial_len-step}/{trial_len},Epsilon:{DQNAgent.
                                                                    epsilon}")
                break
    # Calculate time taken to train
    stopTime =np.round(time.time(), decimals=4)
    totalTime =(np.round(((stopTime -startTime)/60), decimals=4))
    print('Training time: {} minutes'.format(totalTime))

if __name__ =="__main__":
    main()
```

```
episode: 0, score: 225, epsilon: 0.265721, Rewards: −274.0
episode: 1, score: 231, epsilon: 0.141215, Rewards: −268.0
episode: 2, score: 356, epsilon: 0.102946, Rewards: −143.0
episode: 3, score: 301, epsilon: 0.0675426, Rewards: −198.0
episode: 4, score: 328, epsilon: 0.0443147, Rewards: −171.0
episode: 5, score: 353, epsilon: 0.0323054, Rewards: −146.0
episode: 6, score: 284, epsilon: 0.019076, Rewards: −215.0
episode: 7, score: 283, epsilon: 0.0112642, Rewards: −216.0
episode: 8, score: 318, epsilon: 0.00739044, Rewards: −181.0
episode: 9, score: 340, epsilon: 0.00484887, Rewards: −159.0
episode: 10, score: 341, epsilon: 0.00318134, Rewards: −158.0
episode: 11, score: 321, epsilon: 0.00208728, Rewards: −178.0
episode: 12, score: 344, epsilon: 0.00136946, Rewards: −155.0
```

```
episode: 13, score: 302, epsilon: 0.001, Rewards: −197.0
episode: 14, score: 318, epsilon: 0.001, Rewards: −181.0
episode: 15, score: 220, epsilon: 0.001, Rewards: −279.0
episode: 16, score: 247, epsilon: 0.001, Rewards: −252.0
episode: 17, score: 318, epsilon: 0.001, Rewards: −181.0
episode: 18, score: 208, epsilon: 0.001, Rewards: −291.0
episode: 19, score: 198, epsilon: 0.001, Rewards: −301.0
episode: 20, score: 192, epsilon: 0.001, Rewards: −307.0
```

Training time for 100 episodes was 54.81 minutes
Training time for 20 episodes was 19.26 minutes

5 Lessons Learned

1. **Hyper-parameter tuning**: The hyper-parameters play a really important part in helping the NN function estimators to learn and predict the Q-values. In this project, I had to tune these parameters manually to see the effect in overall performance. This manual tuning is difficult and needs a lot of effort to get good results

2. **Double Q-learning**: Double Q-learning performs much better than Q-learning algorithm and helps to converge faster. Working on this project helped me understand how double Q-learning algorithm works in practice and how it can be applied to standard benchmarks like Open AI gym package.

3. **Exploitation vs Exploration**: The exploration and exploitation plays a big role in making algorithm converge to a good predicted values. In this project, the exploration is done in first few episodes and then the algorithm starts doing exploitation. This was controlled by hyper-parameter ϵ, which was dynamically changed to get benefits from this concept.

6 Possible Improvements

1. **Prioritized Experience Replay (PER) [5]**: This is based on the concept that some experiences are more important for the training but they don't occur frequently because we are choosing samples from the memory randomly. In PER, sampling distribution is changed by having tuples of experience with priorities.

2. **Dueling DQN** [2]: In this we separate the NN function estimators into two elements: one that estimates the state value and other that estimates the advantage for each action. The advantage of DDQN is that it can learn the valuable states without having to learn the actions each state.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[2] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[3] A. Choudhary, "A hands-on introduction to deep q-learning using openai gym in python," 2019.

[4] Y. Patel, "Reinforcement learning w/ keras + openai: Dqns," 2017.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.