COMP532 group submission for assignment 2

DQN - Cartpole

Dilan Aktas 201177664,

Daniel Ene 201190945,

Daniel Gardam 201166846,

Mike Hughes 201457290,

Wei Tan 201384356

April 24, 2020

Abstract

Our game of choice for this task is Cart-Pole [1], a game where you have to balance a pole atop a cart by moving the cart along the rails in the direction you think would keep the pole in the most vertical position. The aim of the game is to keep the pole in balance without going too far in either direction, the longest.

Our deep reinforcement learning model of choice is Deep-Q Learning where we use a neural network to approximate the Q-value function

While this is a simple game, it excels at highlighting the aspects of deep reinforcement learning when an DQN agent is implemented for this task.

Chapter 1

Importing OpenAI Gym

OpenAI Gym or just Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

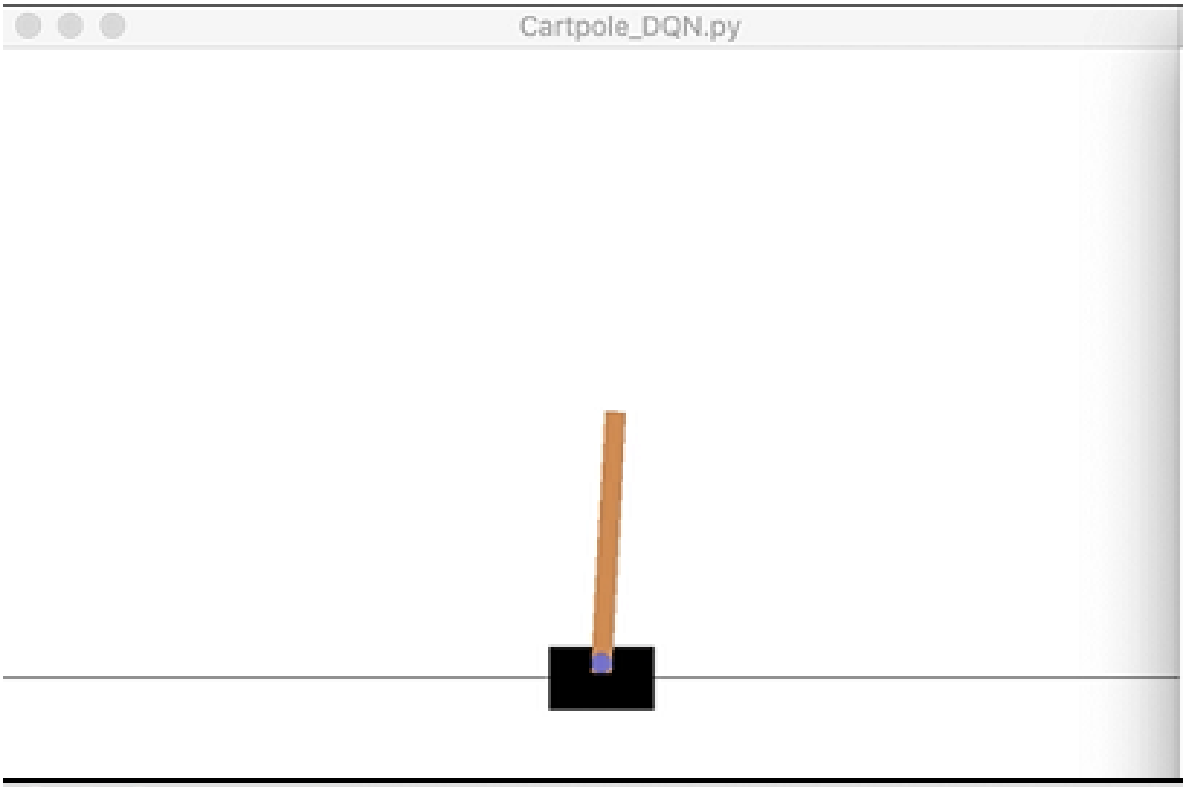
To import it, simply 'import Gym' in code. With it, we create the cart pole, i.e. a pole that is attached by an un-actuated joint to a cart, which moves along a frictionless track.

```
class DQNAgent:
    def __init__(self):
        self.env = gym.make('CartPole-v1')
        # by default, CartPole-v1 has max episode steps = 500
        self.state_size = self.env.observation_space.shape[0]
        self.action_size = self.env.action_space.n
```

In code, the environment is created by line 3 above. gym.make Cartpole facilitates the use of and takes care of the graphics and the variables the game carries, namely, the environment, the state and action space etc. A line as simple as...

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

can make an agent take random actions in this environment.



assignment2 — python Cartpole_DQN.py — 80×28

```
episode: 57/1000, score: 34, e: 0.63
episode: 58/1000, score: 146, e: 0.55
episode: 59/1000, score: 103, e: 0.49
episode: 60/1000, score: 124, e: 0.44
episode: 61/1000, score: 112, e: 0.39
episode: 62/1000, score: 148, e: 0.34
episode: 63/1000, score: 105, e: 0.3
episode: 64/1000, score: 125, e: 0.27
episode: 65/1000, score: 141, e: 0.23
episode: 66/1000, score: 268, e: 0.18
episode: 67/1000, score: 197, e: 0.15
episode: 68/1000, score: 73, e: 0.14
episode: 69/1000, score: 102, e: 0.12
episode: 70/1000, score: 141, e: 0.11
episode: 71/1000, score: 102, e: 0.096
episode: 72/1000, score: 201, e: 0.078
episode: 73/1000, score: 315, e: 0.057
episode: 74/1000, score: 202, e: 0.047
episode: 75/1000, score: 201, e: 0.038
episode: 76/1000, score: 234, e: 0.03
episode: 77/1000, score: 256, e: 0.023
episode: 78/1000, score: 9, e: 0.023
episode: 79/1000, score: 11, e: 0.023
episode: 80/1000, score: 12, e: 0.023
episode: 81/1000, score: 9, e: 0.022
episode: 82/1000, score: 10, e: 0.022
episode: 83/1000, score: 12, e: 0.022
```

Figure 1.1: Cartpole agent learning from his mistakes. OpenAI working correctly.

Chapter 2

The DNN model in use

With DQN we use a neural net to estimate a Q function as part of a classification problem. The Deep Neural Network is made up of 4 dense layers. As input to this, we have the state space, and in the 4th layer we also input the action space. The first three layers having Rectified Linear Units (ReLU) as activation functions and the last one having a simple linear activation function. The output of the network is whether the cart should move left or right in the next timestep.

As shown in chapter 1, the environment is loaded from 'Gym'. This is init-ed in class DQNAgent. Along with the environment, the state space and action space are shown to the agent. Discount rate (gamma), exploration rate (epsilon) and other hyperparameters are initialised as well. The agent also has a memory, where it appends states, actions, rewards and subsequent states to a memory and can pick an action depending on epsilon, while updating Q_max.

Chapter 3

Connection of game to network

The program is based around the following equation where s is the current state and a the chosen action that leads to the next state s'. The equation looks to maximise the Q value by selecting the most rewarding action.

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

Figure 3.1: Q-value

The program has two phases called “run” and “test” in the code.
The initial training phase ”run” uses the neural net to find the best action response for given circumstance or state. These responses are stored in memory as outlined in chapter 2 for use by the replay function. The formula used is based on the above equation and is further refined with the addition of learning and discount rates.

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

Figure 3.2: With experience this recursive equation should converge, where alpha is the learning rate.

The ”test” phase uses the stored response actions so that when the particular state is observed live that action is implemented using the replay function.

```
for i in range(self.batch_size):
    # correction on the Q value for the action used
    if done[i]:
        target[i][action[i]] = reward[i]
    else:
        # Standard - DQN
        # DQN chooses the max Q value among next actions
        # selection and evaluation of action is on the target Q Network
        # Q_max = max_a' Q_target(s', a')
        target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next[i]))
```

Figure 3.3: Code example 1

Chapter 4

Deep reinforcement learning model

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 4)                 0
_________________________
dense_1 (Dense)              (None, 512)               2560
_________________________
dense_2 (Dense)              (None, 256)               131328
_________________________
dense_3 (Dense)              (None, 64)                16448
_________________________
dense_4 (Dense)              (None, 2)                 130
=================================================================
Total params: 150,466
Trainable params: 150,466
Non-trainable params: 0
```

Throughout the 4 dense layers, the way the weights get initialised is through a 'he uniform variance scaling initialiser'. It draws samples from a uniform distribution within [-limit, limit] where limit is sqrt(6 / fan_in) where fan_in is the number of input units in the weight tensor [2].

The optimiser of choice is RMSProp, an adaptive learning rate method. It is basically a backpropagation algorithm adapted for mini-batch learning, encountered in our algorithm.

During training, steps get smaller and smaller because we keep updating the squared grads growing over training. So we divide by the larger number every time. In convex optimisation, this makes a lot of sense, because when we approach minima we want to slow down. In non-convex case it's bad as we can get stuck on saddle point. RMSprop addresses that concern a little bit.

```
def replay(self):
    if len(self.memory) < self.train_start:
        return
    # Randomly sample minibatch from the memory
    minibatch = random.sample(self.memory, min(len(self.memory), self.batch_size))

    state = np.zeros((self.batch_size, self.state_size))
    next_state = np.zeros((self.batch_size, self.state_size))
    action, reward, done = [], [], []

    # do this before prediction
    # for speedup, this could be done on the tensor level
    # but easier to understand using a loop
    for i in range(self.batch_size):
        state[i] = minibatch[i][0]
        action.append(minibatch[i][1])
        reward.append(minibatch[i][2])
        next_state[i] = minibatch[i][3]
        done.append(minibatch[i][4])

    # do batch prediction to save speed
    target = self.model.predict(state)
    target_next = self.model.predict(next_state)
```

Figure 4.1: Code example 2

As mentioned before, activation functions such as ReLU and linear are also hyperparameters that can be tweaked. As part of the network, the batch size, the number of training samples to work through before the model's internal parameters are updated, is also a hyperparameter, along with the loss function (chosen as mean squared error).

In our game, time is represented by timesteps, i.e. at each timestep the cartpole moves either left or right which compound to episodes (initialised to 1000). An episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the centre.

The discount rate (starting at 0.95), the exploration rate (starting at 1.0) and learning rate (starting at 0.00025) are the parameters of the reinforcement learning part of the problem. The learning rate evolves thanks to RMSProp, and the other 2 can be changed according to

observations.

The training set is created on the go, with the agent aiming to pick the best action given the environment, while recording state, action, rewards and the next state along the way. With a 'replay' function defined as well, training of the agent occurs over new behaviour rather than past experiences that might no longer be relevant.

Sometimes you will notice the agent moves the cart left and right in quick succession. That happens when the agent will have learnt the right moves and the pole is relatively vertical. It compensates the tiny error by making a move, eventually ruining the balance.

Chapter 5

Experimental results

See video attached.

Bibliography

[1] AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983.

[2] Francois Chollet et al, Keras: Initialisers, `https://keras.io/initializers/`, 2015.

List of Figures