Using DQN learning on a remote server-based multiplayer game.

Multiplayer computers are extremely popular right now, and there is no reason they will stop in the future. However, I have a problem with them: I am really bad at online videogames and my dignity is on the line, so I was wondering if I could use Reinforcement Learning to solve my problem. My goal would be to make a robot in Python that learns how to play a multiplayer game situated on a remote server, by just communicating with it as if we were a normal player. Sadly, this is out of reach for modern game, so we will create a small multiplayer game as an example, based on a competitive version of blackjack. As for the reinforcement learning method, we will be using something called "Q-Learning" and especially "Deep Q-Network".

The code used here can be on Github at the following link:

Q-Learning and DQN

Q-Learning is a reinforcement learning method. The idea is the following: we can define a function Q(s,a) which describe how good is an action a in a state s. Obviously, we do not know that function. What we could do is make a table of the value of that function for every state and every action, and take the action with the highest Q score for a state. The algorithm to make that table is called Q-learning. However, it is very costly in memory if the number of states or actions are too high. Another way that expands of this idea is to approximate the Q(s,a) function using deep neural network called Deep Q-Network, or DQN for short. This is the algorithm that we will be using.

We need a game for our robot to play

To showcase DQN learning on a server based multiplayer game, we will make one of our own. The example I want to make is a competitive card games inspired by Blackjack. The rules are the following:

- Every player will draw two cards from a deck of cards.
- Each card is worth a certain amount of points: numbers are worth their numbers, figures are worth 10 and aces are worth 11.
- Each turn, a player can draw a card, or stop here.
- When all players stop, the player with the highest amount of point win. However, if a player has more than 21 points, they "burn" and lose.

In practice, we will consider the players are agents, who will be either an actual human player or our robot. We will consider the card dealer as a server, which will give cards to players, organize the beginning and end of the game, and decide who's the winner. Since the server is developed independently of the players, we can develop a special agent that uses DQN learning.

Implementations

We can implement the DQN in Python using four libraries:

- Keras and Tensorflow, which are used for the Deep Learning part
- OpenAI’s Gym, which is used to make environment that can be used for RL
- Keras-RL2, which bridges the two

First, let’s import the libraries we need:

```
from keras.layers import Dense, Activation, Flatten
from keras.models import Sequential
from rl.agents.dqn import DQNAgent
from rl.memory import SequentialMemory
from rl.policy import EpsGreedyQPolicy
from tensorflow.keras.optimizers import Adam
```

Then, we create an environment from a custom environment, that we will create later. We need to know the number of actions possible, to scale the neural network.

```
# Get the environment and extract the number of actions available
env = BlackJack()
nb_actions = env.action_space.n
```

We can then build our neural network using Keras and Tensorflow.

```
model = Sequential()
# model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(2))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
```

This is a pretty simple model, and while it could be improved, it will work for our task since our game is pretty simple. We then can use this model for Deep Q-Learning like this.

```
policy = EpsGreedyQPolicy()
memory = SequentialMemory(limit=50000, window_length=1)
dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory,
nb_steps_warmup=10,
               target_model_update=1e-2,
               policy=policy)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])
```

Finally, we fit the network.

```
dqn.fit(env, nb_steps=5000, visualize=True, verbose=2)
```

If we want to save or load the model, we can do so like this.

```
#Save model
dqn.save_weights("model.h5", overwrite=True)
```

```
#Load model
dqn.load_weights("model.h5")
```

We can use the model with the following command.

```
dqn.test(env, nb_episodes=5, visualize=True)
```

We have now a way to train an agent interacting with a custom environment. However, we need to actually make this environment, which would be our Blackjack game. For this, we need to follow Gym's environment. For this, we need to import the following.

```
import socket
import numpy as np
import gym
from gym.spaces import Box, Discrete
```

Socket is used for data communication with the server, NumPy is generally useful and gym.spaces is important for the custom environment. This is the template for a custom gym environment:

```
class CustomEnv(gym.Env):
  def __init__(self, args):
    self.action_space = spaces.Discrete(n_actions)
    self.observation_space = spaces.Box(low=0, high=255, shape=
                    shape, dtype=np.uint8)

  def step(self, action):
    #do something
    return state, reward, done, info
  def reset(self):
    #do something
  def render(self, mode='human', close=False):
    #do something
```

In the constructor __init__, we'll define the necessary tools for the environment. Pay attention to the self.action_space and self.observation_space variable. self.action_space is used to describe the amount of actions that the agent can do. Here, we have defined self.action_space to spaces.Discrete(n_actions), which means we have n_actions discrete actions. self.observation_space is what the agent obverse, which is basically the environment's state. Here, it is defined to spaces.Box(low=0, high=255, shape=shape, dtype=np.uint8), which is basically a box of continuous values. The "step" method is the bread and butter of the environment. It describes what the agents at every "step" (turns, frames, …) of the environment and returns useful information for our model to use. The "reset" method is called every time the agent is done with the environment. Finally, the "render" method is just how the environment displays itself.

For our game, here is what the constructor looks like.

```
def __init__(self):
    self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.sock.setblocking(True)
    print("connecting")
    self.sock.connect((("localhost", 4050)))
    print("connected to {}".format(self.sock))
    self.burnt = False

    self.cards = []
    self.waiting = False
    for _ in range(2):
        self.burnt = self.draw_a_card()
```

First, we create a socket to communicate with the servers. We'll see how it is implemented later. We set the blocking to True, so that the agent needs to wait for an answer. This isn't great, as it means that it will block itself and wait forever if something happens to the connection, but it will do, instead of handling timeout correctly, but it will do. We then set our action_space and observation_space.

```
self.action_space = Discrete(2)  # draw and stop
self.observation_space = Box(low=0, high=255, shape=12,))
```

There is only two actions: a player can either decide to draw a card or to wait for the game to end. For the observation_space, the player will know about the cards in their hands and that's enough. That means that the players won't be able to see the other players' hand, which might have been interesting to see but isn't an issue in the end.

```
self.sock.send(b"name")
byte_size = int(self.sock.recv(1).decode())
print(byte_size)
self.name = self.sock.recv(byte_size).decode()
print(self.name)
```

The player will just receive a name from the server. This is for readability and mostly for fluff. Pay attention that the player will receive the size of the name in byte before the name itself, because we want to receive exactly the name and its length can vary. This is something we'll do for each socket communication, as a precaution.

```
self.burnt = False

self.cards = []
self.waiting = False
for _ in range(2):
    self.burnt = self.draw_a_card()
```

We store the cards the player drew in a list. For ease of implementation, the cards will just be represented as their score, since their colour doesn't actually matters. This is the beginning of the game so we draw two cards and pay attention if the player "burnt" or not. The "draw_a_card()" method is just to a communication with the server.

```
def draw_a_card(self):
    if not self.waiting:
        self.sock.send(b"draw")
        answer = self.sock.recv(1).decode()
        byte_size = int(answer)
        card = int(self.sock.recv(byte_size).decode())
        self.cards.append(card)
        burning = self.get_BJ_score() > 21
        if burning:
            self.waiting = True
        return burning
```

We need to make sure that the player isn't trying to draw cards even though they decided to stop earlier, and we need to make sure that the player is waiting for the game to end if they burn. Also, get_BJ_score() is just a method that sums the cards.

```
def get_BJ_score(self):
    return sum(self.cards)
```

We also need to implement the step method. It needs the action taken by the player as an argument. First, we declare two variables that will be used in the learning.

```
def step(self, action):
    done = False
    reward = 0
```

Done is used to see if the game is finished or not. Reward is how the model learns if it's doing good or not: the reward will go up if the model is doing and down if not. Here, reward is equal to 0 if the game is continuing, -1 if the player loses and 1 if the player wins. A negative reward during the game's duration is common to prevent a game to carry on uselessly, but we don't need that since the game is due to end even if the agents try to make the game last. Then, we'll choose in function of the agent's action.

```
if action == 1 and not self.waiting:  # we wait
    # print("starting to wait")
    self.waiting = True
if action == 0 and not self.waiting:  # we draw a card
    self.burnt = self.draw_a_card()
    if not self.burnt:
        self.waiting = False
    else:
        self.waiting = True
```

In both cases, we ignore the agent's input if they are waiting since they cannot do anything. It shouldn't happen anyway, but we are never too sure. If the agent decides to wait, we just remember that for the future and we'll handle that later since a player can draw a card yet immediately wait for the game to end if they burn. If the player decides to draw a card, we ask the server for one and check if the player burns. Afterwards, we implement what happens if the agent is waiting.

```
if self.waiting:  # we stop
    self.sock.send(b"wait")
    answer = self.sock.recv(1)
    print(answer)
    while answer == b"c": #still waiting
        self.sock.send(b"wait")
        answer = self.sock.recv(1)
    if answer == b"w":
        print("{} won with {}".format(self.name, self.get_BJ_score()))
        reward = 1
        done = True
    elif answer == b"l":
        print("{} lost wih {}".format(self.name, self.get_BJ_score()))
        reward = -1
        done = True
    else:
        print(answer.decode())
        raise Exception()
```

We communicate with the server our intention to stop. If the game is still on for other players, the server will tell us, and we'll continue to communicate that we are waiting. If all the players are waiting for the end, then the server would communicate if we won or not. The agent will set its reward in function and in both case, we remember that the game is done. The step method is almost done, we just need to format a few data that the method returns.

```
# Setting the placeholder for info
info = {}

#Formatting the state
state = np.pad(self.cards, (0, 12 - len(self.cards)))

# Returning the step information
return state, reward, done, info
```

The info is used for debugging but we're not using it so it's empty. The state of the environment in the agent's view is the list of cards we have. Since the observation_space with a specific length, we need to pad it. We just add 0, the model should learn to ignore that.

The two remaining methods are fairly simple. The reset method is just setting the class variable to the starting one and drawing two new cards. We also warn the server about the reset.

```
def reset(self, *, seed=None, return_info=False, options=None, ):
    self.cards = []
    self.waiting = False
    self.burnt = False
    self.sock.send(b"rese")
    for _ in range(2):
        self.burnt = self.draw_a_card()
    return np.pad(self.cards, (0, 12 - len(self.cards)))
```

And the render method is just a print.

```
def render(self, mode="human"):
    print("Player {} has those cards: {}".format(self.name, self.cards))
```

Now, the last important thing we need to implement is the server. It only needs the socket library, the threading library and random, so we can import those. The name library is just a file containing a single list of names.

```
import random
import socket
import threading

from names import names
```

We then set up the socket for communication.

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(("localhost", 4050))
sock.setblocking(True)
sock.listen(5)
```

We declare a constant and two variables that the server uses. Those are

```
NBR_PLAYERS_NEEDED = 2
resolving_game = False
players = {}
nbr_players = 0
```

NBR_PLAYERS_NEEDED is used to set the number of players needed to start a game, resolving_game is used in the game's completion, while the server communicate who's the winner and who are the losers, players is a list of the players and nbr_players is the number of players who connected.

```
class Player:
    def __init__(self, cnn, addr, id):
        self.cards = 0 #just the sum here
        self.name = "undefined"
        self.waiting = False

        self.rcards = []
        self.cnn = cnn
        self.addr = addr

        self.id = id
```

For a player, we store the sum of their cards, their names, if they are waiting or not, their list of cards, their socket connection information and an id.

A way the server can work is with two threads: one that handles connections and one that handles the game itself.

```
if __name__ == "__main__":
   print("=====================================================")
   threading.Thread(target=running).start()
   threading.Thread(target=connection).start()
```

First, let's look at the running function.

```
def connection():
    global players, nbr_players, NBR_PLAYERS_NEEDED
    print("Connection starting")

    while 1:
        cnn2, addr2 = sock.accept()
        print(f"New player from address : {addr2}")
        new_player = Player(cnn2, addr2, nbr_players % NBR_PLAYERS_NEEDED)
        players[nbr_players % NBR_PLAYERS_NEEDED] = new_player
        nbr_players += 1
```

It continuously accepts connection from players, create an object with the players' information and adds it to the players list.

The running method is a bit more complex. As the running thread, it also works in an infinite loop. In this loop, we first copy the list of players to work on it. Then we iterate of it, to ensure everyone gets their turn.

```
def running():
    global nbr_players, players, resolving_game
    print("Main starting")
    cards = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]
    while True:
        if len(players) != 0:
            copy_of_players = players.copy()

            for key, player in copy_of_players.items():
                print(copy_of_players.items())
                print(f"Turn of {player.name}")
                cnn = player.cnn
                addr = player.addr
```

We then wait for all the players to connect.

```
waiting = True
while nbr_players < NBR_PLAYERS_NEEDED:
    if waiting:
        print("waiting")
        waiting = False
```

Then, in a “try-except” clause to prevent communication errors, we receive data from the turn’s player. A player will always send a bytes object of length 4 here, for implementation ease. Also, don’t forget that the sockets are set to blocking. As for the players, this isn’t great since that if a player stop being active for any reason without announcing its departure, the server will stop, waiting for them. Using timeout is a better way of doing this.

```
data2 = cnn.recv(4)
data = data2.decode()
```

We then handle the message using if/else in the true Python way. Here is the several messages it handles.

```
if data == "draw":
    card = random.choice(cards)
    cnn.send(str(len(str(card).encode())).encode())
    cnn.send(str(card).encode())
    players[key].cards += card
    players[key].rcards.append(card)
    if players[key].cards > 21:
        players[key].cards = -1
        players[key].waiting = True
        all_players_waiting = [i.name for i in copy_of_players.values() if
not i.waiting]
        if len(all_players_waiting) == 0:
            game_completed = True
```

We randomly select a card from the cards list, send it to the player and save it on the server side. We also need to handle the player burning if they go over 21. You might be thinking that it is unrealistic, as we don’t actually remove a card from the deck. While it is true, it isn’t so farfetched since real-life casino shuffles several decks of cards to prevent the players from counting the cards, so we can consider it as a casino that shuffles an infinite amount of decks.

```
elif data == "name":
    name = random.choice(names)
    name_b = name.encode()

    cnn.send(str(len(name_b)).encode())

    cnn.send(name_b)
    players[key].name = name
```

This is just to give the player a name.

```
elif data == "stop":
    players = {k: v for k, v in copy_of_players.items() if v != player}

    nbr_players -= 1
```

This handle a player leaving the game.

```
elif data == "rese":
    players[player.id].rcards = []
    players[player.id].cards = 0
    players[player.id].waiting = False
    copy_of_players[player.id].rcards = []
    copy_of_players[player.id].cards = 0
    copy_of_players[player.id].waiting = False
```

This resets a player to the default value, in both the general players list and in the iteration list since we need to update both. We don’t need to draw cards here because the player will ask it themselves.

```
elif data == "wait":
    players[key].waiting = True
    copy_of_players[key].waiting = True
    all_players_waiting = [i.name for i in copy_of_players.values() if not
i.waiting]
    if len(all_players_waiting) == 0:

        winner = max(copy_of_players, key=lambda x: players[x].cards)

        if player.id == winner:
            print("{} won with a score of {} and the cards
{}".format(player.name,

player.cards,

player.rcards))
            player.cnn.send(b"w")
        else:
            print("{} lost with a score of {} and the cards
{}".format(player.name,

player.cards,

player.rcards))
            player.cnn.send(b"l")
    else:
        player.cnn.send(b"c")
```

This one is a bit complicated. First, we set the player to waiting in both the global player list and the iteration list. Then, we check if all the players are waiting. If they aren't, we just communicate that the game is still going on. However, if they are, we determine who's the winner and communicate to everyone their status.

```
else:
    raise Exception()
```

Every other message raises an exception. Speaking of, we close the "try" with the "except" clause, in which we just leave the running thread.

```
except:
    return
```

With all that, we can train two agents to compete. However, we forgot about the fleshy human player. This is easy to implement, since we already did all the work. We can just reuse the custom environment we made for Gym and strip it from the data only used by Gym. Since it's controlled by action, we can just call the step method with the action the human player chooses. It would look like something like this.

```
if __name__ == "__main__":
    playing = True
    env = BlackJack()
    while playing:
        cards = env.reset()
        waiting = False
        done = False
        reward = 0
        while not done:
            if not waiting:
                print("Your cards are {}".format([card for card in cards if 
card != 0]))
                choice = input("0 if you want to draw a card or 1 if you 
want to stop here")
                if choice == '0' or choice == '1':
                    cards, reward, done, info = env.step(int(choice))
                    if choice == '1':
                        waiting = True
                else:
                    print("You need to input 0 or 1")
            else:
                n_state, reward, done, info = env.step(1)
        if reward == 1:
            print("You've won")
        else:
            print("You've lost")

        choice = input("Want to replay y/N ?")
        if choice != "y":
            playing = False
```

Since it is relatively light, we can imagine an implementation of this on a small ARM-based chip, like a Raspberri Pi. I myself modified the above code to run on a Pynq card, so I could control the game using buttons. In any case, now our super AI can wreck human players at this game, which is obviously great.

Conclusion

DQN is a very interesting tools and it’s interesting to see that it can work to talk to a server without that server being adapted for deep learning. As we can see, the server we made doesn’t do specific computation that the learning method needs: this is all on the client side. Also, this is pretty performant: our simple model will beat an agent playing randomly 70% of the time, which is great for a game that is pretty luck based.