DQN Blackjack Agent

Introduction to problem
Solution for problem
    Resources
    Environment setup
    Blackjack Agent
        Constructor
        Choose action
        Update experience
        Update neural network
Performance

Introduction to problem

Blackjack is one of the simplest and played casino games. The rule are simple, in particular there is a dealer and players. Each player play against the dealer. The game use a french deck of cards (in this case the deck contain infinite card). The game start with two face up cards for each player and with one face up card and one face down card for the dealer. The goal is that the sum of the card of the player is bigger than the dealer, but the sum must be at most 21 (twenty one). The player can ask for more card but the sum of the card must less or equal of 21, otherwise the player lose. The points are the follow:

- numbers have itself as value
- figures have 10 as value
- ace can have 1 or 11 as value

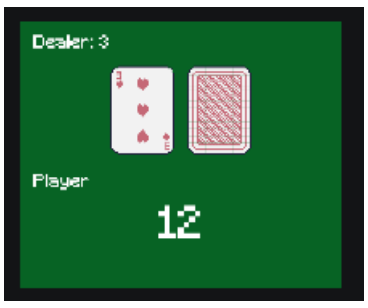
Knowing this rules we can write the Markov Decision Process MDP.

Let be $MDP =< S, A, \delta, r >$ with:

- $S = \{(s_{player}, c_{dealer}, a_b) |\ s_{player} \in N,\ c_{dealer} \in N, a_b \in \{0, 1\}\}$, in particular s_{player} is the current sum of the player, c_{dealer} is the value of the card with face up, and a_b is a boolean that is true if player hold a usable ace.
- $A = \{Stick, Hit\}$, the action space is composed by two action $Stick$ (0) and Hit (1) for take or not more card.
- δ and r are unknown. For r we know only that

$$r = \begin{cases} +1\ win \\ -1\ lose \\ 0\ draw \\ +1.5\ blackjack \end{cases}$$

So, now that we have the non deterministic MDP let implement a Blackjack player that learn to play to blackjack an maximize the reward. For this implementation I use an environment available by `gymnasium`.



Solution for problem

For create the blackjack player I implement an Reinforcement Learning Agent, in particular I use an Neural Network to approximate the **Q-function**, in particular I implement a DQN agent.

Resources

For implement the Blackjack agent I use the follow python packages:

- `gymnasium`: for create and use the blackjack environment.
- `numpy`: for evaluate some vector computation;
- `random`: for give some randomness of the algorithm, in particular for the exploration and exploitation;
- `tensorflow`: for create the Neural Network;
- `collection`: for use some useful structures;

```
# Import necesary libraries
import gymnasium as gym
import numpy as np
import random
import seaborn as sns
import matplotlib.pyplot as plt
```

```
from matplotlib.patches import Patch
from tqdm import tqdm
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from collections import deque
from collections import defaultdict
```

Environment setup

The Environment setup is the follow, in particular:

1. **Create the environment**
2. **Create the agent** that must interact with the world
3. **Interaction between environment and agent**, i.e the agent must do the follow action:
    a. **Choose an action**
    b. **Execute the action**
    c. **Observe the new state and the reward**
    d. **Update its knowledge**

In this case for more accuracy I execute the agent in the environment many times (10000), in that way it can accurate it's knowledge of the environment. After each execution I update the neural network that approximate the Q-function.

```
# Create blackjack environment
env = gym.make("Blackjack-v1", sab=True)

# Create blackjack agent
agent = BlackjackAgent(env)

n_episodes = 10000
env = gym.wrappers.RecordEpisodeStatistics(env, deque_size=n_episodes)
for episode in tqdm(range(n_episodes)):
    # Initializiate the environment to the first observation
    done = False
    obs, info = env.reset()

    while not done:
        # CHOOSE action, EXECUTE action and READ reward and state (obs)
        act = agent.get_action(obs)
        obs_next, reward, terminated, truncated, info = env.step(act)

        # update the agent
        agent.update_experience(episode, obs, act, reward, terminated, obs_next)

        # update if the environment is done and the current obs
        done = terminated or truncated
        obs = obs_next

    # Update neural network
    agent.update_nn(episode)
```

Blackjack Agent

The **Blackjack Agent** is implemented by a python class, for represent the agent as an object that interact with the world (i.e. the environment). Let see that there are intrinsic parameters of the environment that the agent must know:

```
# Blackjack environment parameters
__OBS_SPACE_SIZE=3    # obs space are tuple in N^3
__ACT_SPACE_SIZE=1    # act space is in N
__ACT_NUM=2           # act space is boolean
```

In the follow let's describe the implemented methods for the agent.

Constructor

For create the **Blackjack agent** we need by the follow parameters:

- ϵ_{max}**:** initial probability of exploration
- ϵ_{min}: final probability of exploration
- γ : discount factor

- $|D|$: experience max size
- $|T| < |D|$: batch size

```
def __init__(
       self,
       env,
       epsilon_max: int = 70,
       epsilon_min: int = 5,
       gamma: float = 0.9,
       learning_rate: float = 0.01,
       dataset_max_size: int = 10000,
       batch_size: int = 1000,
):
```

In the constructor I define the architecture of the neural network:

```
# NN architecture for aproximate Q-function
self.__model = Sequential()
self.__model.add(Dense(32, input_shape=(self.__OBS_SPACE_SIZE + self.__ACT_SPACE_SIZE,),activation='relu'))
self.__model.add(Dense(16, activation='relu'))
self.__model.add(Dense(1,activation='linear'))
self.__model.compile(optimizer='sgd', loss='mse')
```

Choose action

For choose the action we have two way: at random (**exploration**) or best action respect Q-function (**exploitation**). We choose to explore with probability ϵ and exploit with probability $1 - \epsilon$:

```
def get_action(self, obs: tuple[int, int, bool]) -> int:
        rv = random.randint(1,100)
        act = self.__env.action_space.sample()

        if rv >= self.__epsilon:
            candidates = {}
            candidates[0]= self.__model.predict_on_batch(tf.constant([[*obs, 0]]))[0][0]
            candidates[1]= self.__model.predict_on_batch(tf.constant([[*obs, 1]]))[0][0]
            act=max(candidates, key=candidates.get)
        return act
```

Update experience

The update of the experience, i.e. update of the Q-function follow the follow training rule for the stochastic models:

$$Q_n(s,a) \leftarrow Q_{n-1}(s,a) + \frac{r + \gamma * max_{a'} Q(s',a') - Q_{n-1}(s,a)}{n+1}$$

```
def update_experience(
            self,
            n,
            obs: tuple [int, int, bool],
            action: int,
            reward: float,
            terminated: bool,
            next_obs: tuple[int, int, bool],
):
        candidates_next = {}
        candidates_next[0]= self.__model.predict_on_batch(tf.constant([[*next_obs, 0]]))[0][0]
        candidates_next[1]= self.__model.predict_on_batch(tf.constant([[*next_obs, 1]]))[0][0]
        act_next=max(candidates_next, key=candidates_next.get)

        future_q_value = (not terminated) * candidates_next[act_next]
        current_q_value = self.__model.predict_on_batch(tf.constant([[*obs, action]]))[0][0]


        # Training Rule
        temporal_difference = reward + self.__GAMMA * future_q_value - current_q_value
        current_q_value = current_q_value + temporal_difference / (n+1)
```

```
        if len(self.__experience) >= self.__EXP_MAX_SIZE:
                self.__experience.popleft()

        self.__experience.append([*[*obs, act], current_q_value])
        self.training_error.append(temporal_difference)
```

Update neural network

After 1000 episodes every 10 episodes we train the neural network. In particular we consider the experience as dataset that is composed in the follow way:

S	A	$Q(s,a)$
$(s_{player}, c_{dealer}, a_b)$	a	$Q(s,a)$

with the a random samples taken form the experience dataset (a batch)

```
def decay_epsilon(self):
        self.__epsilon -= self.__epsilon/100
        if self.__epsilon <= self.__EPS_MIN:
                self.__epsilon = self.__EPS_MIN

def update_nn(
        self,
        n,
):
        if len(self.__experience) >= self.__BATCH_SIZE and n % 10 == 0:
                batch = random.sample(self.__experience, self.__BATCH_SIZE)
                trainset = np.array(batch)
                X = trainset[:,:4]
                Y = trainset[:,4]

                # train network
                self.__model.fit(tf.constant(X), tf.constant(Y), validation_split=0.2)
                self.decay_epsilon();
```

Performance

At the end I use the follow library for plot the results:

- `matplot`
- `seaborn`

I obtain the follow results