**Deep Q-Learning with Python and TensorFlow 2.0**

In reinforcement learning, self-learning **agent** learns some type of **interaction**between itand the **environment**.

The agent wants to achieve some kind of **goal**within mentioned environment while it interacts with it. This interaction is divided into **time steps**. In each time step, **action** is performed by agent. This action changes the **state**of the environment and based on the success of it agent gets a certain **reward**. This way the agent learns what actions should be performed an which shouldn’t in a defined environment state.

This is oddly similar to the way we as humans learn. When we are babies, we **experiment**. We perform some actions and get a **response**from the environment based on it. If the response is positive (**reward**) we mark those actions as good, otherwise (**punishment**) we mark them as bad.

A screenshot of a computer

Description automatically generated with low confidence

**Q – Learning**



A *Q-Value* for a particular state-action combination can be observed as the **quality** of an action taken from that state. As you can see the **policy** still determines which state–action pairs are visited and updated, but nothing more. This is why [*Q-Learning*](https://rubikscode.net/2019/06/24/introduction-to-q-learning-with-python-and-open-ai-gym/) is sometimes referred to as **off-policy**.

The interesting point in the formula is *maxQ(St+1, a)*. This means that Q-value of the **current step** is based on the Q-value of the **future step**. It is confusing, I know. This means that we initialize *Q-Values* for *St* and *St+1* to some random values at first. In the first training iteration we update *Q-Value* in the state *St* based on reward and on those random value of *Q-Value* in the state *St+1*. Since **reward** is still guiding our system, this will eventually converge to the best result.

All these *Q-Values* are stored inside of the *Q-Table*, which is just the [matrix](https://rubikscode.net/2019/04/29/mathematics-for-artificial-intelligence-linear-algebra/)with the rows for states and the columns for actions:

Table

Description automatically generatedExample of Q-Table

To get it even more clear we can break down *Q-Learning* into the steps. It would look something like this:

1. **Initialize**all *Q-Values* in the *Q-Table* arbitrary, and the Q value of terminal-state to 0:  
   *Q(s, a) = n, ∀s ∈ S*, *∀a ∈ A(s)*   
   *Q(terminal-state, ·) = 0*
2. **Pick**the action *a*, from the set of actions defined for that state *A(s)* defined by the policy π.
3. **Perform**action *a*
4. **Observe** reward *R* and the next state *s’*
5. For all possible actions from the state *s’* select the one with the **highest***Q-Value* – *a’*.
6. **Update**value for the state using the formula:   
   *Q(s, a) ← Q(s, a) + α [R + γQ(s’, a’) − Q(s, a)]*
7. **Repeat**steps 2-5 for each time step until the terminal state is reached
8. **Repeat**steps 2-6 for each episode

It is important to note, that this type of learning can get stuck in the certain scenarios which might not be the best solution for the problem. For example, algorithm can learn that the best thing going from state s is performing action *a’* and going to state *s’*. However, it never performed action *a”* and ending up in the state *s”*, which could be a better option. Because of this reason, we use parameter *epsilon*. It defines will we explore new actions and maybe come up with a better solution – **exploration**, or we will go with the already learned route – **exploitation**.

Diagram

Description automatically generated

Q-Learning

The problem with the *Q-Learning* is of course **scaling**. When we are talking about complicated environments, like the planning a video game, number of states and actions can grow. Table becomes a **complicated** approach for this problem. That is where [artificial neural networks](https://rubikscode.net/2018/02/19/artificial-neural-networks-series/) come into play.

**Deep Q – Learning**

*Deep Q-Learning* harness the power of **deep learning** with so-called **Deep Q-Networks**. These are standard [feed forward neural networks](https://rubikscode.net/2017/11/13/introduction-to-artificial-neural-networks/) which are utilized for calculating *Q-Value*. In this case, the agent has to store previous experiences in a local memory and use max output of neural networks to get new *Q-Value*.

Diagram, schematic

Description automatically generated

The important thing to notice here is that *Deep Q-Networks* don’t use standard [supervised learning](https://rubikscode.net/2018/01/15/how-artificial-neural-networks-learn/), simply because we don’t have labeled expected **output**. We depend on the policy or value functions in reinforcement learning, so the **target** is continuously changing with each iteration. Because of this reason the agent doesn’t use just one neural network, but **two** of them. So, how does this all fit together? The first network, called **Q-Network** is calculating *Q-Value* in the state *St*, while the other network, called **Target Network** is calculating *Q-Value* in the state *St+1.*

Speaking more formally*,* given the current state*St*, the *Q-Network* retrieves the **action-values***Q(St,a)*. At the same time the *Target-Network* uses the next state*St+1* to calculate *Q(St+1, a)* for the [Temporal Difference target](https://rubikscode.net/2019/06/24/introduction-to-q-learning-with-python-and-open-ai-gym/). In order to **stabilize** this training of two networks, on each *N-th* iteration parameters of the *Q-Network* are **copied** over to the *Target Network*. The whole process is presented in the image below.

Diagram

Description automatically generated

We already mentioned that the agent has to **store** previous experiences. *Deep Q-Learning* goes one step further and utilizes one more concept in order to improve the agent performance – **experience replay**. It is empirically proven that neural network training process is more stable when training is done on **random** batch of previous experiences. Experience replay is nothing more than the **memory** that stores those experiences in a form of a tuple ***<s, s’, a, r>***:

* ***s*** – State of the agent
* ***a*** – Action that was taken in the state ***s***by the agent
* ***r*** – Immediate reward received in state ***s***for action***a***
* ***s’*** – Next state of the agent after state ***s***

Both networks use random batches of ***<s, s’, a, r>*** from the experience replay to calculate *Q-Values* and then do the [backpropagation](https://rubikscode.net/2018/01/22/backpropagation-algorithm-in-artificial-neural-networks/). The **loss** is calculated using the squared difference between **target** *Q-Value* and **predicted** *Q-Value*:



Note that this is performed only for the **training** of *Q-Network*, while parameters are **transferred** to *Target Network* later.

To sum it all up, we can split the whole process of Deep Q-Learning into steps:

1. Provide **the state** of the environment to the agent. The agent uses *Target Network* and Q-Network to get the *Q-Values* of all possible actions in the defined state.
2. **Pick**the action *a*, based on the **epsilon** value. Meaning, either select a random action (exploration) or select the action with the maximum Q-Value (exploitation).
3. **Perform**action *a*
4. **Observe** reward r and the next state *s’*
5. **Store** these information in the experience replay memory <s, s’, a, r>
6. **Sample** random batches from experience replay memory and perform **training** of the *Q-Network*.
7. Each *Nth* iteration, **copy** the weights values from the *Q-Network* to the *Target Network*.
8. **Repeat**steps 2-7 for each episode

**Implementation**

**Prerequisites**

In order to the code from this tutorial, you have to have *Python 3* installed on your machine. In this example, we are using *Python 3.7.* The implementation is done using *TensorFlow* 2.0. The complete guide on how to install and use *Tensorflow* 2.0 can be found [here](https://rubikscode.net/2019/04/22/ultimate-guide-to-tensorflow-2-0-in-python/).

Also, you have to install *Open AI Gym* or to be more specific *Atari Gym*. You can **install** it by running:

pip install gym[atari]

There is one more additional module you need to install in order for the code from this tutorial to work and that is the *progressbar* library. It is not doing anything essential, it just for cosmetic purposes.

**Environment**

We are using the Gym environment called Taxi-V3. This is one very **simple**environment. To sum it up, there are **4 locations** in the environment and the goal of an agent (taxi) is to pick up the passenger at one location and drop him off in another. The agent can perform **6 actions** (south, north, west, east, pickup, drop-off). You can find more **information** about this environment [here](https://gym.openai.com/envs/Taxi-v2/). https://gymnasium.farama.org/

A picture containing text, device, meter, gauge

Description automatically generated

**Code**

First, we **import** all necessary modules and libraries:

|  |
| --- |
|  |
| import numpy as np |
|  |

|  |
| --- |
| import random |
|  |

|  |
| --- |
| from IPython.display import clear\_output |
|  |

|  |
| --- |
| from collections import deque |
|  |

|  |
| --- |
| import progressbar |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| import gym |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from tensorflow.keras import Model, Sequential |
|  |

|  |
| --- |
| from tensorflow.keras.layers import Dense, Embedding, Reshape |
|  |

|  |
| --- |
| from tensorflow.keras.optimizers import Adam |

Note that apart form standard libraries and modules like *numpy*, *tensorflow* and *gym*, we imported *deque* from *collections*. We will use it for experience replay **memory**. After this we can create the **environment**:

|  |
| --- |
|  |
| enviroment = gym.make("Taxi-v3").env |
|  |

|  |
| --- |
| enviroment.render() |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| print('Number of states: {}'.format(enviroment.observation\_space.n)) |
|  |

|  |
| --- |
| print('Number of actions: {}'.format(enviroment.action\_space.n)) |

We use the *make* function to instantiate an **object**of the *Taxi-v3* environment. The current state of the environment and the agent can be presented with the *render* method. The important thing is that we can access all states of the environment using *observation\_space* property and all actions of the environment using *action\_space*. This environment has **500 states** and **6 possible actions**. Apart from these methods, Open Gym API has two more methods we need to mention. The first one is the *reset* method which **resets**the environment and returns a random initial state. Another one is the *step* method which **steps**the environment by one time-step and **performs**an action.

After this we can finally implement the **agent**. The *Deep Q-Learning* agent is implemented within the *Agent* class. Here is how that looks like:

|  |
| --- |
|  |

class Agent:

def \_\_init\_\_(self, enviroment, optimizer):

# Initialize atributes

self.\_state\_size = enviroment.observation\_space.n

self.\_action\_size = enviroment.action\_space.n

self.\_optimizer = optimizer

self.expirience\_replay = deque(maxlen=2000)

# Initialize discount and exploration rate

self.gamma = 0.6

self.epsilon = 0.1

# Build networks

self.q\_network = self.\_build\_compile\_model()

self.target\_network = self.\_build\_compile\_model()

self.alighn\_target\_model()

def store(self, state, action, reward, next\_state, terminated):

self.expirience\_replay.append((state, action, reward, next\_state, terminated))

def \_build\_compile\_model(self):

model = Sequential()

model.add(Embedding(self.\_state\_size, 10, input\_length=1))

model.add(Reshape((10,)))

model.add(Dense(50, activation='relu'))

model.add(Dense(50, activation='relu'))

model.add(Dense(self.\_action\_size, activation='linear'))

model.compile(loss='mse', optimizer=self.\_optimizer)

return model

def alighn\_target\_model(self):

self.target\_network.set\_weights(self.q\_network.get\_weights())

def act(self, state):

if np.random.rand() <= self.epsilon:

return enviroment.action\_space.sample()

q\_values = self.q\_network.predict(state)

return np.argmax(q\_values[0])

def retrain(self, batch\_size):

minibatch = random.sample(self.expirience\_replay, batch\_size)

for state, action, reward, next\_state, terminated in minibatch:

target = self.q\_network.predict(state)

if terminated:

target[0][action] = reward

else:

t = self.target\_network.predict(next\_state)

target[0][action] = reward + self.gamma \* np.amax(t)

self.q\_network.fit(state, target, epochs=1, verbose=0)

We know, that is a lot of **code**. Let’s split it up and explore some important parts of it. Of course, the whole agent is initialized inside of the **constructor**:

def \_\_init\_\_(self, enviroment, optimizer):

# Initialize atributes

self.\_state\_size = enviroment.observation\_space.n

self.\_action\_size = enviroment.action\_space.n

self.\_optimizer = optimizer

self.expirience\_replay = deque(maxlen=2000)

# Initialize discount and exploration rate

self.gamma = 0.6

self.epsilon = 0.1

# Build networks

self.q\_network = self.\_build\_compile\_model()

self.target\_network = self.\_build\_compile\_model()

self.alighn\_target\_model()

First we initialize size of the **state** and **action space** based on the environment object that is passed to this agent. We also initialize an **optimizer** and the experience reply **memory**. Then we build the *Q-Network* and the *Target Network* with the \_*build\_compile\_model* method and **align** their weights with the *alighn\_target\_model* method. The \_*build\_compile\_model* method is probably the most interesting one in this whole implementation, because it contains the **core** of the implementation. Let’s peek at it:

def \_build\_compile\_model(self):

model = Sequential()

model.add(Embedding(self.\_state\_size, 10, input\_length=1))

model.add(Reshape((10,)))

model.add(Dense(50, activation='relu'))

model.add(Dense(50, activation='relu'))

model.add(Dense(self.\_action\_size, activation='linear'))

model.compile(loss='mse', optimizer=self.\_optimizer)

return model

We see that the first layer that is used in this model is *Embedding* layer. This layer is most commonly used in a language processing, so you might be curious what is it doing here. The problem that we are facing with the *Taxi-v2* environment is that it returns **discrete value** (single number) for the state. This means that we need to reduce number of potential values a little bit. The *Embedding* layer, the parameter *input\_dimensions* refers to the number of values we have and *output\_dimensions* refers to the vector space we want to reduce them. To sum it up, we want to represent 500 possible states by 10 values and *Embedding* layer is used for exactly this. After this layer, *Reshape* layer **prepares** data for feed-forward neural network with three *Dense* layers.

The whole expropriation-exploration concept we mentioned in the previous chapter is done inside of the *act* function. Based on the epsilon value we either invoke *Q-Network* to make a prediction, or we pick a **random** action. Like this:

def act(self, state):

if np.random.rand() <= self.epsilon:

return enviroment.action\_space.sample()

q\_values = self.q\_network.predict(state)

return np.argmax(q\_values[0])

|  |
| --- |
|  |

Finally, lets take a look at the *retrain* method. In this method we pick random samples from the experience replay memory and train the *Q-Network*:

|  |
| --- |
|  |

def retrain(self, batch\_size):

minibatch = random.sample(self.expirience\_replay, batch\_size)

for state, action, reward, next\_state, terminated in minibatch:

target = self.q\_network.predict(state)

if terminated:

target[0][action] = reward

else:

t = self.target\_network.predict(next\_state)

target[0][action] = reward + self.gamma \* np.amax(t)

self.q\_network.fit(state, target, epochs=1, verbose=0)

Now, when we are aware of the *Agent* class implementation, let’s create an object of it and prepare for training:

|  |
| --- |
|  |
| optimizer = Adam(learning\_rate=0.01) |
|  |

|  |
| --- |
| agent = Agent(enviroment, optimizer) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| batch\_size = 32 |
|  |

|  |
| --- |
| num\_of\_episodes = 100 |
|  |

|  |
| --- |
| timesteps\_per\_episode = 1000 |
|  |

|  |
| --- |
| agent.q\_network.summary() |

From the output of our this sample of the code, we can see structure of networks in the agent:

Table

Description automatically generated

for e in range(0, num\_of\_episodes):

    # Reset the enviroment

    state = enviroment.reset()

    state = np.reshape(state, [1, 1])

    # Initialize variables

    reward = 0

    terminated = False

    bar = progressbar.ProgressBar(maxval=timesteps\_per\_episode/10, widgets=[progressbar.Bar('=', '[', ']'), ' ', progressbar.Percentage()])

    bar.start()

    for timestep in range(timesteps\_per\_episode):

        # Run Action

        action = agent.act(state)

        # Take action

        next\_state, reward, terminated, info = enviroment.step(action)

        next\_state = np.reshape(next\_state, [1, 1])

        agent.store(state, action, reward, next\_state, terminated)

        state = next\_state

        if terminated:

            agent.alighn\_target\_model()

            break

        if len(agent.expirience\_replay) > batch\_size:

            agent.retrain(batch\_size)

        if timestep%10 == 0:

            bar.update(timestep/10 + 1)

    bar.finish()

    if (e + 1) % 10 == 0:

        print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

        print("Episode: {}".format(e + 1))

        enviroment.render()

        print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")