

Used Environment: CartPole-v1

We used the deep Q-learning to solve this problems by taking as an input the state and outputs Q-values for each possible action for the current state. The biggest q-value corresponds to the best action. We implemented the DQN algorithm:

- Experience replay buffer: it helps to avoid two problems which are forgetting previous experiences and the correlations in data. In reinforcement learning we receive at each time step a tuple composed by the state, the action, the reward, and the new state. In order to make our agent learn from the past policies, every tuple is stored in the experience replay buffer.
- For the policy, we used a random policy because we got to choose from 2 actions only.

Model architecture

The DQN agent has a target and local networks having the same architecture:

- 1 fully connected layer of size 24, activation: Relu
- 1 fully connected layer of size 24, activation: Relu
- 1 output layer of size 2 (the size of the action space), activation: Linear
- Optimizer : Adam , Loss : mse

Hyperparameters

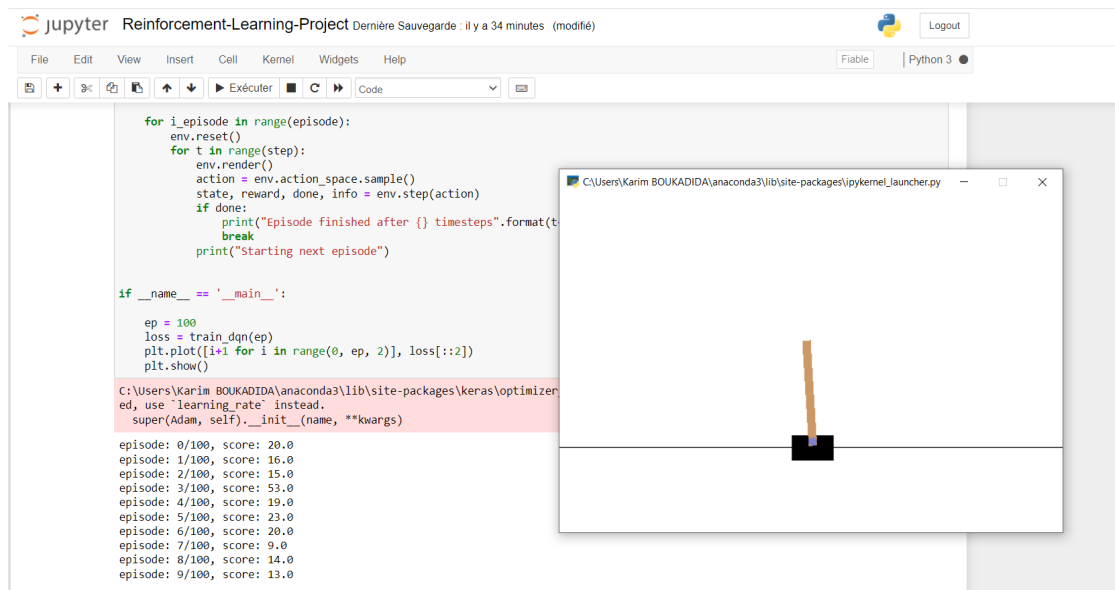
We used these hyperparameters to train our agent:

- Buffer size: the size of the experience replay buffer is 10 000
- Epsilon 1
- Epsilon min 0.01
- Epsilon decay 0.995
- Batch size: the batch size of the training is 64
- Gamma: the discount factor 0.95
- learning rate coefficient 0.001

Training our agent

For each episode, we start by giving the initial state of our environment to the agent. Then, for each time step we give our agent the current state of our environment and it will return the action that it will perform. After performing each action, the agent will save the experience in the replay buffer and return its state and reward.

We stop the training of our agent once we hit the 100 episodes.



The screenshot shows a Jupyter Notebook titled "Reinforcement-Learning-Project". The code in the notebook defines a training loop for 100 episodes. It uses a DQN model with a replay memory of size 1000. The training loop prints the score at the end of each episode. A game window titled "C:\Users\Karim BOUKADIDA\anaconda3\lib\site-packages\pygame\pygame.py" is open, showing a simple game environment with a black square and a brown vertical bar.

```
for i_episode in range(episode):
    env.reset()
    for t in range(step):
        env.render()
        action = env.action_space.sample()
        state, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
        print("Starting next episode")

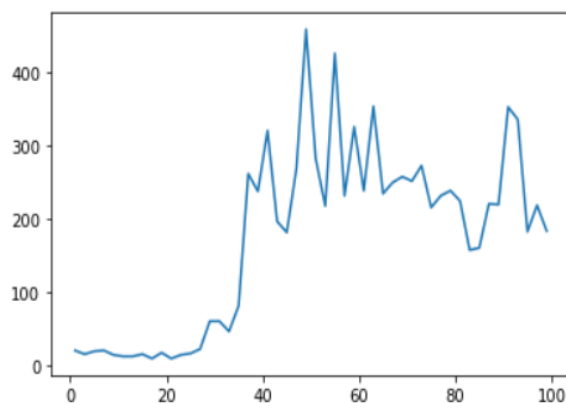
if __name__ == '__main__':
    ep = 100
    loss = train_dqn(ep)
    plt.plot([i+1 for i in range(0, ep, 2)], loss[::2])
    plt.show()

C:\Users\Karim BOUKADIDA\anaconda3\lib\site-packages\keras\optimizer.py:10: UserWarning: The `init` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)

episode: 0/100, score: 20.0
episode: 1/100, score: 16.0
episode: 2/100, score: 15.0
episode: 3/100, score: 53.0
episode: 4/100, score: 19.0
episode: 5/100, score: 23.0
episode: 6/100, score: 20.0
episode: 7/100, score: 9.0
episode: 8/100, score: 14.0
episode: 9/100, score: 13.0
```

Results

```
episode: 93/100, score: 267.0
episode: 94/100, score: 182.0
episode: 95/100, score: 177.0
episode: 96/100, score: 218.0
episode: 97/100, score: 233.0
episode: 98/100, score: 183.0
episode: 99/100, score: 208.0
```



In this graph we can see the scores of our agent vs the episodes.

In the episode: 48/100 we reached our highest score for the agent that was 458.0.

By the episode 40 we clearly have a good result for our model.

