Machine Learning Project

Solving Car Racing using Q-Learning

Federico Ambrogio 1810535

SAPIENZA
UNIVERSITÀ DI ROMA

Contents

1. Introduction

Car Racing is an environment that is part of the Box2D Gymnasium environments. In Car Racing there is an agent that tries to learn to drive a car autonomously through a random track using certain actions.

The Action Space of Car Racing can be of two types:

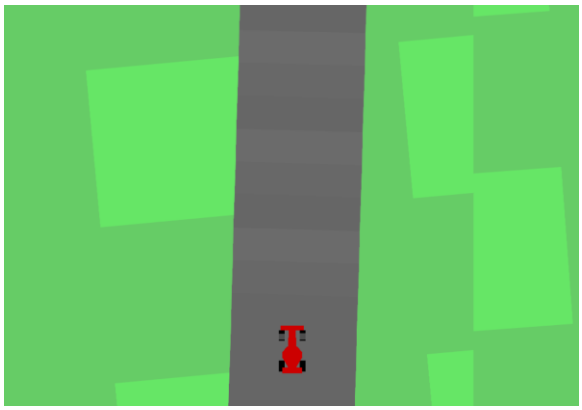
- Continuous: In this case there are three possible actions, which are steering, gas and breaking.
- Discrete: There are five possible actions to be executed, which are do nothing, steer left, steer right, gas and break.

In my project I decided to use the continuous one, but I reduced the Action Space into twelve actions.

```
#Define the Action Space, that is composed by 12 Actions of type (Steering Wheel, Throttle, Break)
action_space = [
                (-1, 1, 0.2), (0, 1, 0.2), (1, 1, 0.2),
                (-1, 1,   0), (0, 1,   0), (1, 1,   0),
                (-1, 0, 0.2), (0, 0, 0.2), (1, 0, 0.2),
                (-1, 0,   0), (0, 0,   0), (1, 0,   0)
              ]
```

All the actions are triples, in which the first value is the steering wheel (if its value is -1 the car is steering left, if it is 0 it's going straight, while if it is +1 it is steering right), the second one is the gas (whose minimum value is 0, while the maximum is 1) and the last one is the break (that can be a value between 0 and 0.2). As it is possible to see, I considered all the possible combinations of the values of the three actions to cover all the possible situations.

Now let us talk about the Observation Space, which is represented by a 96x96x3 RGB image showing the car and the track. Consider that I will manipulate in different ways these images to reach my goal, but I'll explain that topic later in the paper. Here follows an example of the Observation Space in a certain state.



As regards rewards, they are -0.1 at each frame and 1000/N for each track tile visited, where N is the total number of tiles visited. This drives the car to pass through the smaller number of track tiles possible, to reach as soon as possible the end of the circuit. Each episode will terminate in two cases, either if all the track tiles are visited or if the car goes beyond the boundary of the game: notice that in this case it will receive a -100 reward.

To solve the Car Racing problem, I used Q-Learning, which is a Reinforcement Learning policy that aims to learn the Q-Function. This function selects the best action to perform for each possible state of the environment, in such a way as making a step towards the goal.

I used two different strategies to learn this function:

- Q-Table: This is the simplest method to solve the problem, which consists of building a table whose rows are all the possible states and columns are all the possible actions. Using this table, it is possible to discover which is the action that provides the best reward for each possible state.
- Neural Network: This method is both more difficult and with better performances, and it consists in using a Neural Network to approximate the Q-Function.

2. Training using Q-Table

As I said before, the simplest way to solve Car Racing problem is building a Q-Table. The rows of this table are all the possible states, while columns are related each one to an action in that state. If we want to know which is the action that provides the highest reward in a certain state, we can simply search the row related to that state and pick the action with the highest value.

I created the Q-Table using a dictionary, in which the keys are the identifiers of the states, while the related values are arrays that contain the values associated with the twelve actions.

The hardest part in the development of this table has been the discretization of the states. Indeed, the Observation Space is composed of 96x96x3 RGB image, so each state is represented by an image. Since there is a huge number of possible states, I had to find a way to discretize images in such a way to reduce the number of states and let the Q-Table work. I had a lot of different ideas to perform this, but finally I decided to use the following strategy:

- Convert the states into grayscale images.
- Execute an 8x8 Average Pooling to reduce the dimensionality of the images.
- Discretize the values of the pixels, categorizing them into 8 sets of values.

Notice that for each state I automatically create an identifier to use as key in the dictionary: this identifier is a string composed by a character for each pixel, each one having a number between 0 and 8 referring to the set of pixel values to which the pixel belongs.

The code of the discretization function is the following one.

```
#This is the function that discretizes the image
def discretize_image(image):
    image = image.unsqueeze(0)

    #Execute the 3x3 Average Pooling
    pooling = nn.AvgPool2d(8, stride=8)
    image = pooling(image)
    image = image.squeeze(0)

    #Create the image identifier
    string = "S"
    for i, x in enumerate(image.numpy()):
        for j in x:
            if(j < 30):
                string = string + "0"
            elif(j < 60):
                string = string + "1"
            elif(j < 90):
                string = string + "2"
            elif(j < 120):
                string = string + "3"
            elif(j < 150):
                string = string + "4"
            elif(j < 180):
                string = string + "5"
            elif(j < 210):
                string = string + "6"
            elif(j < 256):
                string = string + "7"

    return string
```

Let’s now analyze the Algorithm I used.

```
#Q Table
if(config.USE_QTABLE):
    #Initialize the Q-Table as a Dictionary
    Q_Table = {}

    for i in range(config.NUM_EPISODES):              #Iterate over the Episodes
        state, info = env.reset()               #The state is a 96x96 Matrix

        state = cv2.cvtColor(state, cv2.COLOR_BGR2GRAY)           #Convert the state into a Grayscale Image

        #Discretize the image
        state = torch.from_numpy(state.astype('float32'))
        state = discretize_image(state)

        done = False
        j = 0

        while(True):                    #Iterate over the Steps
            action = select_action(state, epsilon)                              #Select the action to perform
            observation, reward, done, truncated, info = env.step(action)       #Execute one step with the chosen action
            observation = cv2.cvtColor(observation, cv2.COLOR_BGR2GRAY)          #Convert the state into a Grayscale Image
            image = torch.from_numpy(observation.astype('float32'))
            next_state = discretize_image(image)                                #Discretize the current state

            #If the following state is inside the Q-Table, take the maximum among the values of its entry in the dictionary
            if next_state in Q_Table:
                next_max = np.max(Q_Table[next_state])
            else:
                next_max = 0

            #Apply the Q-Learning Formula to update the Q-Table entry of the current state and action
            if state in Q_Table:
                Q_Table[state][action_space.index(action)] += alpha * (reward + config.GAMMA * next_max - Q_Table[state][action_space.index(action)])
            else:
                Q_Table[state] = [0] * 12
                Q_Table[state][action_space.index(action)] += alpha * (reward + config.GAMMA * next_max - Q_Table[state][action_space.index(action)])

            state = next_state
```

```
            #If we obtain more than 25 consecutive negative reward, we'll terminate the current episode
            tot_negative_reward = tot_negative_reward + 1 if time_frame_counter > 100 and reward < 0 else 0

            #Increment the Reward if the current Action has the maximum value for the Throttle and the minimum value fot the Break
            if action[1] == 1 and action[2] == 0:
                reward *= 1.5

            episode_reward += reward
            if j % 4 == 0:
                time_frame_counter += 1         #Update the counter of the Frames
            j += 1

            #Stop iterating if the current episode is finished, truncated, it has a negative cumulative reward or it has 25 consecutive negative rewards
            if done or truncated or tot_negative_reward > 25 or episode_reward < 0:
                epsilon = update_epsilon_nn(epsilon)            #Update the value of Epsilon
                print("Current Episode: ", i)
                print("Cumulative reward: ", episode_reward)
                print("Current epsilon: ", epsilon)
                print("\n")
                epsilon = update_epsilon(epsilon)
                cum_reward_table[i]=episode_reward
                episode_reward = 0      #Reset the total reward for each episode
                tot_negative_reward = 1
                time_frame_counter = 1
                break


        with open("test.csv", "w") as outfile:
            writer = csv.writer(outfile)      #Pass the csv file to csv.writer
            writer.writerow(Q_Table.keys())
            #Let's now iterate on each column and assign the corresponding values to the column
            for i in range(12):
                writer.writerow([Q_Table[x][i] for x in key_list])
            !cp test.csv "../drive/My Drive/"
```

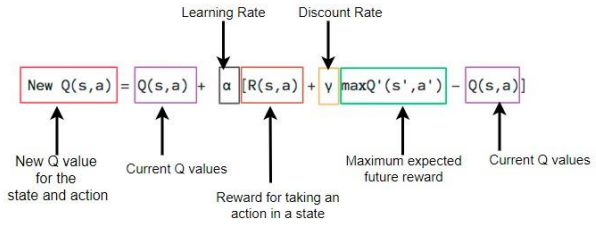
First, it initializes the empty dictionary that represents the Q-Table and then it starts the for loop that iterates over episodes. Then there is a while loop that iterates over steps: for each iteration the discretization process is executed. Each time the action to be executed is chosen using the function called “select_action”, that is the following one.

```
#Q-Table Policy
def select_action(state, epsilon):
    rv = random.uniform(0, 1)
    if rv < epsilon:
        return action_space[random.randrange(len(action_space))]          #Sample a random action from the Action Space
    else:
        if state in Q_Table:
            return action_space[np.argmax(Q_Table[state])]                #Return the action with the highest value for the current state in the Q-Table
        else:
            return (0, 1, 0)
```

This function with a probability that depends by the Epsilon value (that is decreased at each episode until it reaches its minimum possible value 0.1) decides the action to perform by sampling it randomly from the Action Space or selecting the action with the highest value from the Q-Table.

Finally, the Q-Learning Formula is applied: if the current state is already inside the Q-Table I directly apply it, otherwise first I have to create the dictionary pair referred to the state initializing it with all zero values, and then I can apply the Formula.

The Q-Learning Formula is the following one.



So, the new Q-Value related to a certain state and a certain action is computed by summing to the current Q-Values the Learning Rate Alpha multiplied by all the things that are inside the square brackets. Inside them the Reward of executing that action in that state is summed to the maximum expected future reward (that is multiplied by the Discount Rate Gamma) and then the current Q-Values are subtracted from it.

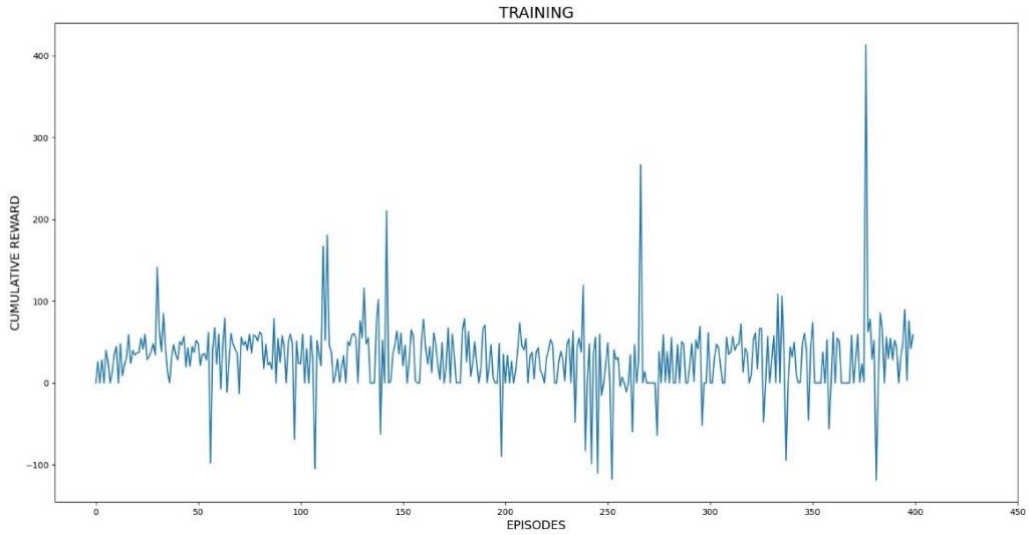
Notice that since I want the car to complete the track in the fastest way possible, I increase the reward if the agent is using the maximum value for the throttle.

Each episode is stopped if one of the following conditions happens:

- The episode is finished or truncated.
- If the agent obtains more than 25 consecutive negative rewards.
- If the cumulative episode reward is negative.

When training is finished the Q-Table is saved into a CSV File, that can be used in the testing phase.

I performed the training over 400 episodes, and the results are the following:



As is it possible to see, the cumulative reward remains more or less in the same interval [0, 100] for all the episodes, apart from some rare peaks. Since rewards don't increase, that means that this strategy is not good enough and we have to try another one.

3. Training using a Neural Network

Since using a Q-Table was not a technique with good performances, let's now consider another methodology, that consists in using a Neural Network to approximate the Q-Function.

Let's first analyze the Model that I decided to use for training.

```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=7, stride=3)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=4)

        self.dense1 = nn.Linear(432, 216)
        self.dense2 = nn.Linear(216, 12)

        self.maxpool = nn.MaxPool2d((2, 2))


    def forward(self, state):
        x0 = self.maxpool(F.relu(self.conv1(state)))
        x1 = self.maxpool(F.relu(self.conv2(x0)))
        x2 = torch.flatten(x1)
        x3 = F.relu(self.dense1(x2))
        x4 = self.dense2(x3)
        return x4
```

This Network takes in input a tensor with shape 96x96x3, that is not a RGB image, but its channels are three consecutive states, that are previously converted to grayscale images. In this way, the model can take its decisions considering more than one state and a larger time window.

The Network is composed of:

- Convolutional Layers to detect features in the images.
- Fully Connected Layers to obtain the final decision of the Network.
- Max Pooling Layers to reduce the dimensionality of the images retaining the most important information.

Notice that the activation function I used after applying convolutional layers is the ReLU. In addition, I used the Flatten operation before passing data to Fully Connected Layer, because it allowed me to reshape the multi-dimensional tensor into a one-dimensional one.

The resulting prediction is composed of 12 elements, each one representing the Q-Value related to each action.

Here it follows the algorithm:

```
#Use a Neural Network to approximate the Q-Function
else:
    for i in range(config.NUM_EPISODES):            #Iterate over the Episodes
        state, info = env.reset()                   #The state is a 96x96 Matrix
        state = cv2.cvtColor(state, cv2.COLOR_BGR2GRAY)          #Convert the state into a Grayscale Image, that is a Matrix 96x96 composed by Integer values

        frames_queue = deque([state]*3, maxlen = 3)              #Add the initial state into the Queue

        done = False

        while(True):                    #Iterate over the Steps

            current_frame = np.array(frames_queue)

            action = select_action_nn(current_frame, epsilon)         #Select the aAction with the maximum predicted Q-Value
                                                                      #The Action is composed by 3 Values, that are the steering wheel, gas and breaking


            rew = 0
            #Skip 3 Frames
            for tot in range(3):
                next_state, reward, done, truncated, info = env.step(action)
                rew += reward
                if done or truncated:
                    break

            #If we obtain more than 25 consecutive negative reward, we'll terminate the current episode
            tot_negative_reward = tot_negative_reward + 1 if time_frame_counter > 100 and reward < 0 else 0


            #Increment the Reward if the current Action has the maximum value for the Throttle and the minimum value fot the Break
            if action[1] == 1 and action[2] == 0:
                rew *= 1.5

            episode_reward += rew               #Update the cumulative episode reward
```

```
            next_state = cv2.cvtColor(next_state, cv2.COLOR_BGR2GRAY)           #Convert the next state into a Grayscale Image

            frames_queue.append(next_state)                                     #Append the next state to the Queue
            next_frame = np.array(frames_queue)

            #Remove the oldest item if the queue is full, in a way such that we can add a new one
            if len(buffer)>=config.BUFFER_SIZE:
                buffer.popleft()                #We dequeue the oldest item

            buffer.append((current_frame, action_space.index(action), reward, next_frame, done))

            #Stop iterating if the current episode is finished, truncated, it has a negative cumulative reward or it has 25 consecutive negative rewards
            if done or truncated or tot_negative_reward > 25 or episode_reward < 0:
                epsilon = update_epsilon_nn(epsilon)            #Update the value of Epsilon
                print("Current Episode: ", i)
                print("Cumulative reward: ", episode_reward)
                print("Current epsilon: ", epsilon)
                print("\n")
                break
```

```
        #Let's train the Neural Network every 4 actions and if the buffer has at least BATCH_SIZE elements
        if(len(buffer) >= config.BATCH_SIZE):
            batch = random.sample(buffer, config.BATCH_SIZE)               #Shuffle randomly the elements of the Buffer

            output_array = []                #This array will contain the predictions made by the Model with respect to the current Action
            target_array = []                #This array will contain the Rewards updated summing the prediction made by the Target Model (Using the Bellman's Equation)

            #Iterate over the shuffled elements of the Buffer
            for current_frame, action, reward, next_frame, done in batch:

                #Use the Target Model to compute the prediction over the next frame
                next_frame = torch.from_numpy(next_frame.astype('float32')).to(config.DEVICE)
                next_frame = target_model(next_frame).detach()

                #Compute the Target using the Bellman's Equation
                target = reward + (1 - done) * config.GAMMA * max(next_frame)

                #Append the Target to the Target Array
                target_array.append(target)

                #Use the Model to compute the prediction over the current frame
                current_frame = torch.from_numpy(current_frame.astype('float32')).to(config.DEVICE)
                output = model(current_frame)

                #Append the prediction of the current Action to the Output Array
                output_array.append(output[action])

            #Stack the two Arrays
            output_array = torch.stack(output_array)
            target_array = torch.stack(target_array)

            #Compute the Mean Squared Error Loss
            loss = mean_squared_error(output_array, target_array)

            optimizer.zero_grad()                   #Reset the Gradients
            loss.backward()                         #Execute the Backpropagation of the Loss
            optimizer.step()                        #Update the weights

        time_frame_counter += 1         #Update the counter of the Frames
```

```
    #Save the weights of the Network every 5 Episodes
    if (i+1) % 5 == 0:
        config.save_model(model, optimizer, i+1)

        #Save the Weights also on Google Drive
        torch.save({
            "state_dict": model.state_dict(),
            "optimizer": optimizer.state_dict(),
        }, "../drive/MyDrive/Checkpoint/" + str(i))

        print("Weigths saved in: "+ config.CHECKPOINT_FOLDER)

    #Update the weights of the Target Network every 5 Episodes
    if (i+1) % 5 == 0:
        config.load_model(config.CHECKPOINT_FOLDER, target_model, optimizer_target)
        print("Target network updated")


    cum_reward_nn[i] = episode_reward                    #Add the current episode's reward to the array

    #Reset the Variables
    episode_reward = 0
    tot_negative_reward = 1
    time_frame_counter = 1
```

The algorithm starts with a for loop that iterates over episode. Each iteration starts with resetting the environment, converting the first state into a grayscale image, and adding it to a queue called frames_queue, that will contain for each timestep the three current states to pass to the neural network. Then inside this for there is a while loop, that iterates over steps of current episode. Inside it the algorithm first chooses the action to perform using the select_action_nn function:

```
    #Neural Network Policy
    def select_action_nn(state, epsilon):
        rv = random.uniform(0, 1)
        if rv < epsilon:
            return action_space[random.randrange(len(action_space))]          #We sample a random action from the Action Space

        else:
            prediction = model(torch.from_numpy(state.astype('float32')).to(config.DEVICE)).detach().cpu().numpy()
            action = action_space[np.argmax(prediction)]                      #Select the action with the maximum predicted value
            return action
```

This function with a probability lower than epsilon chooses a random action from the action space, otherwise it exploits the prediction returned by the neural network. Notice that at each iteration of the algorithm the value of epsilon is decreased, until it reaches the minimum value that is defined in the config file. After that, three frames are skipped executing the same action and the last frame is considered as the next one: this is done to reduce redundant information and simulate decision duration.

Now there are two interesting parts to consider related to the reward. First, we want to induce the agent to go with the maximum speed possible, so to do that I increase the reward of a factor 1.5 if the maximum value for throttle is used. Then, since it is useless for the agent to drive through grass, if it receives more than 25 consecutive negative rewards the current episode is interrupted.

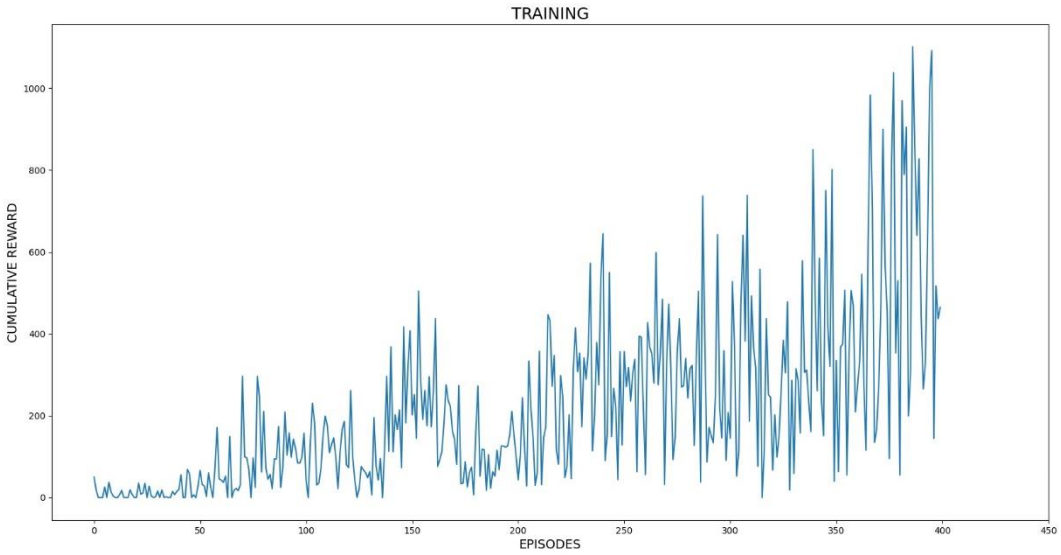
Finally, the current state, the next reached state, the reward, and the action performed are added to the buffer.

Now let's talk about the training of neural network, that is performed every four steps. Notice that two networks are trained: the actual network and the target network, that allows to increase the efficiency and stability of the training, mitigating the problem of the instability in the action selection, that is an issue that occurs when weights of the neural network are updated. So, the target model is the stable reference that reduces oscillations during training, since it is updated less frequently, and so weights variation are not propagated: indeed, the weights of the target model are updated each five episodes. The training consists in iterating over all the elements of the buffer, let's analyze them. First, we use the target network to compute the prediction over the next state, and then we exploit the Bellman's Equation:

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

It is used to compute the target, that is obtained by summing the reward with the gamma parameter multiplied by the maximum value over the actions of the prediction we made before. Then, we apply the actual model to the current state to obtain its prediction and we calculate the mean squared error loss between it and the target. Finally, the backpropagation of the loss is executed, and weights are updated.

I performed the training phase composed of 400 episodes, obtaining the following results:



Unlike training using Q-Table, in this case it's possible to see clearly that the cumulative rewards become higher and higher during iterations. Indeed, in the final 50 episodes we have that the average is more or less 500, and that in a lot of executions the rewards are higher than 1000.

4. Training without using Double Q-Learning

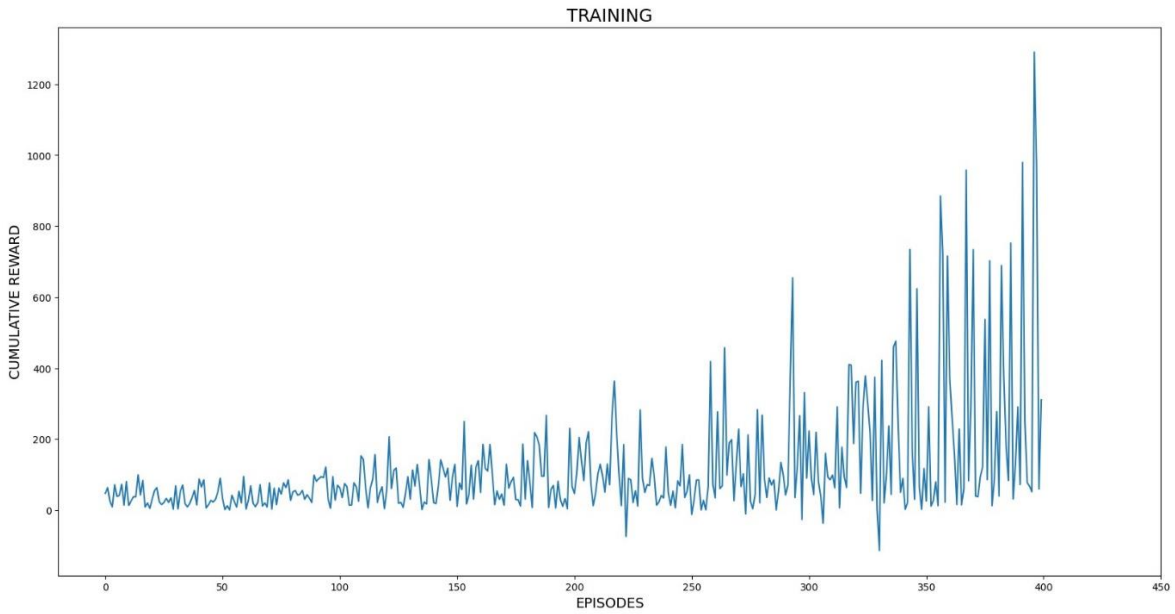
In the previous chapter I've shown an optimization of the solution of the problem called Double Q-Learning, in which I've used two different Networks (the actual model and the target model) to obtain a better result. Now let's analyze another solution, in which I use only the actual model to solve the same issue.

The code is more or less the same, but in this case obviously I don't initialize it and I don't update the weights of the target model, since I'm not using it. The code changes in the training part.

```
#Let's train the Neural Network every 4 actions and if the buffer has at least BATCH_SIZE elements
if(len(buffer) >= config.BATCH_SIZE):
    batch = random.sample(buffer, config.BATCH_SIZE)                #Shuffle randomly the elements of the Buffer

    output_array = []                #This array will contain the predictions made by the Model with respect to the current Action
    target_array = []                #This array will contain the Rewards updated summing the prediction made by the Target Model (Using the Bellman's Equation)

    #Iterate over the shuffled elements of the Buffer
    for current_frame, action, reward, next_frame, done in batch:

        #Use the Model to compute the prediction over the next frame
        next_frame = torch.from_numpy(next_frame.astype('float32')).to(config.DEVICE)
        next_frame = model(next_frame)

        #Compute the Target using the Bellman's Equation
        target = reward + (1 - done) * config.GAMMA * max(next_frame)

        #Append the Target to the Target Array
        target_array.append(target)

        #Use the Model to compute the prediction over the current frame
        current_frame = torch.from_numpy(current_frame.astype('float32')).to(config.DEVICE)
        output = model(current_frame)

        #Append the prediction of the current Action to the Output Array
        output_array.append(output[action])

    #Stack the two Arrays
    output_array = torch.stack(output_array)
    target_array = torch.stack(target_array)

    #Compute the Mean Squared Error Loss
    loss = mean_squared_error(output_array, target_array)

    optimizer.zero_grad()                   #Reset the Gradients
    loss.backward()                         #Execute the Backpropagation of the Loss
    optimizer.step()                        #Update the weights
```

As it is possible to see, in this case I don't use the Target Model to compute the prediction over the next_frame, but I directly use the actual Model. After that, I use its prediction to compute the target using the Bellman's Equation, and finally I compute the Mean Squared Error loss between this target and the prediction made by the actual Model over current_frame. In this way, the training phase works in a very similar way with respect to the previous one, the only difference is that the actual Model is used also to compute the target.

Also in this case I performed a training phase composed of 400 episodes, and the results are the following.



The results, as expected, are not satisfactory like in Double Q-Learning, but they are still good: indeed, after episode 350 the average value for cumulative rewards is roughly 400, that is a good value. Notice that in some peaks the Rewards are also higher of 1000, so there are some episodes in which the reward is very high: I'm pretty sure that training for other 100/200 episodes results can become even better.

5. Testing

For the testing phase I used the following slightly modified version of the code I used for training:

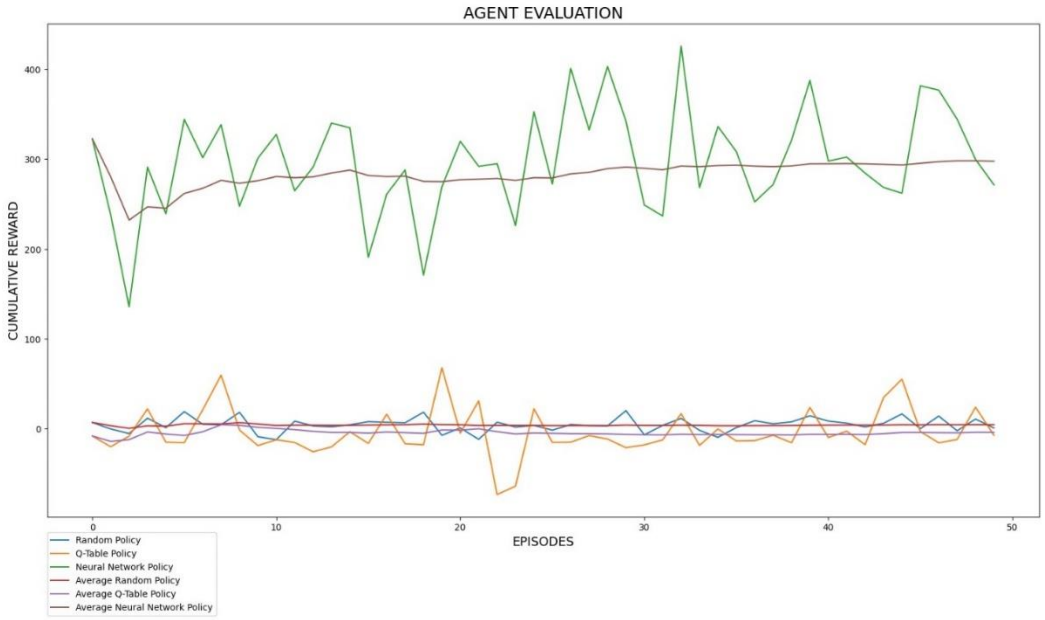
```
#Average rewards
average_random = np.zeros(config_test.NUM_EPISODES)
average_QTable = np.zeros(config_test.NUM_EPISODES)
average_nn = np.zeros(config_test.NUM_EPISODES)

#Cumulative reward
cum_reward_random = np.zeros(config_test.NUM_EPISODES)
cum_reward_QTable = np.zeros(config_test.NUM_EPISODES)
cum_reward_nn = np.zeros(config_test.NUM_EPISODES)

#Initialize the Model and load weights
model = Model().to(config_test.DEVICE)
optimizer = optim.Adam(model.parameters(), lr=config_test.LR)
config_test.load_model(config_test.CHECKPOINT_FOLDER, model, optimizer)

#Load the Q-Table csv file
Q_Table = {}
with open('QTable/QTable.csv', 'r') as csvfile:
    csvreader = csv.reader(csvfile)
    headers = next(csvreader)
    for header in headers:
        Q_Table[header] = []

    for row in csvreader:
        for col_idx, value in enumerate(row):
            Q_Table[headers[col_idx]].append(value)


#Policies
def policy(p, state):
    if(p == "Random"):
        return action_space[random.randrange(len(action_space))]          #Sample a random action from the Action Space

    elif(p == "Q-Table"):
        if state in Q_Table:
            return action_space[np.argmax(Q_Table[state])]                #Return the action with the highest value for the current state in the Q-Table
        else:
            return (0, 1, 0)

    elif(p == "nn"):
        prediction = model(torch.from_numpy(state.astype('float32')).to(config_test.DEVICE)).detach().cpu().numpy()
        action = action_space[np.argmax(prediction)]                 #Select the action with the maximum predicted Q-Value
        return action
```

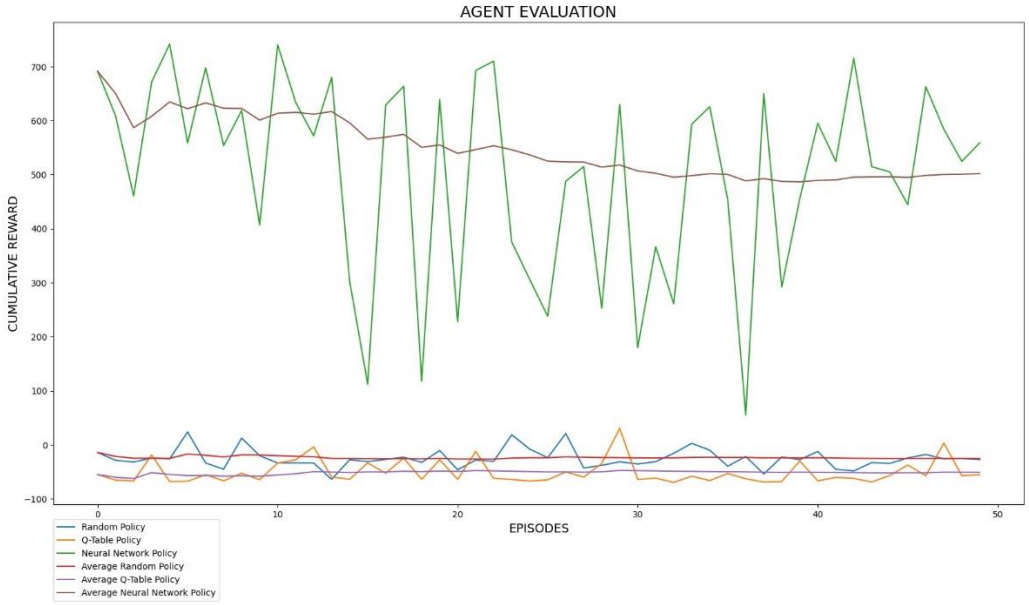
```
#Initialize the Car Racing Environment
env = gym.make("CarRacing-v2", render_mode="human")
observation = env.reset()

#Testing
def test(strategy):
    c_reward = 0                #Cumulative reward
    total_reward = 0
    episode = 0
    for i in range(config_test.NUM_EPISODES):
            observation = env.reset()

            observation = cv2.cvtColor(observation, cv2.COLOR_BGR2GRAY)          #Convert the state into a Grayscale Image, that is a Matrix 96x96 composed by Integer values

            if(strategy == "Neural Network policy"):
                print("NN POLICY")
                frames_queue = deque([observation]*3, maxlen = 3)
            elif(strategy == "Q-Table policy"):
                print("Q-TABLE POLICY")
            else:
                print("RANDOM POLICY")

            for j in range(500):
                env.render()

                if(strategy == "Random policy"):
                    action = policy("Random", observation)
                elif(strategy == "Q-Table policy"):
                    observation = torch.from_numpy(observation.astype('float32'))
                    observation = discretize_image(observation)
                    action = policy("Q-Table", observation)
                elif(strategy == "Neural Network policy"):
                    current_frame = np.array(frames_queue)
                    action = policy("nn", current_frame)

                observation, reward, done, truncated = env.step(action)
                c_reward += reward

                observation = cv2.cvtColor(observation, cv2.COLOR_BGR2GRAY)

                if(strategy == "Neural Network policy"):
                    frames_queue.append(observation)

                if done or truncated or j == 499:
                    total_reward += c_reward
                    break
```

```
            if(strategy == "Random policy"):
                cum_reward_random[episode] = c_reward # save the cum_reward for the relative episode
                average_random[episode] = total_reward/(episode+1)
            elif(strategy == "Q-Table policy"):
                cum_reward_QTable[episode] = c_reward
                average_QTable[episode] = total_reward/(episode+1)
            elif(strategy == "Neural Network policy"):
                cum_reward_nn[episode] = c_reward
                average_nn[episode] = total_reward/(episode+1)

            print("The cumulative reward is:", c_reward)
            c_reward = 0 # reset the current cumulative reward
            print("Episode: ", episode)
            episode += +1

    print(f"Tests for {strategy} finished after {config_test.NUM_EPISODES} episodes")



test("Random policy")
test("Q-Table policy")
test("Neural Network policy")

env.close()
```

With this code I performed 50 epochs of testing with the three different possible strategies, that are: random policy, Q-Table Policy and Neural Netowk Policy. Notice also that I performed two testing phases: one in which all the episodes have a duration of 500 steps, and one in which I executed all episodes until their termination. Furthermore, the Q-Table I used is the one obtained in the previous training and the Neural Network model has the weights returned by the previous training.

The results are the following:

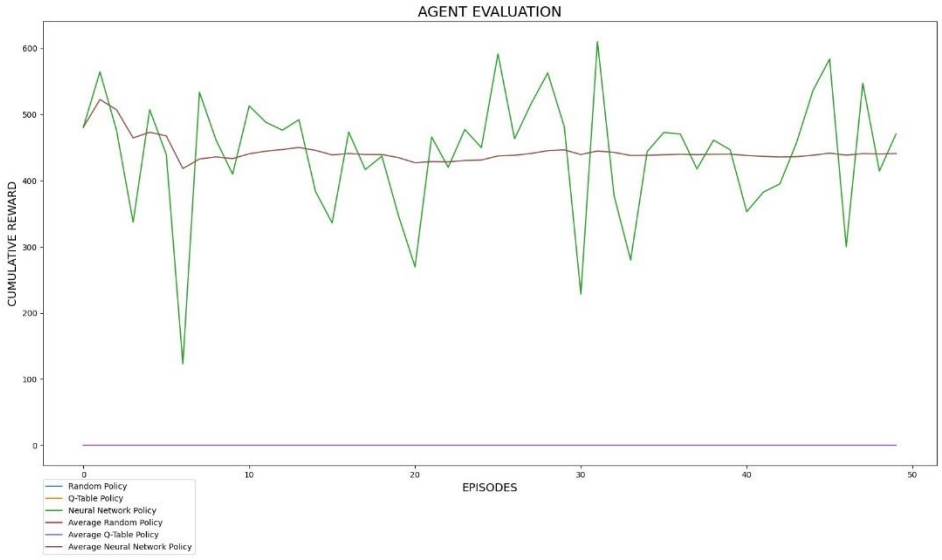




This shows again that both Random and Q-Table policies are not good strategies, indeed their rewards are on average near to zero.

Instead, the average cumulative rewards of Neural Nework policy are clearly better: apart from some rare cases their values are a way higher with respect to the other two strategies.

Now, let's analyze the results of the testing using Neural Network Policy, but without Double Q-Learning.



It's clear that results also in this case are good, but they are slightly worse than the results of Double Q-Learning. Indeed, using Double Q-Learning the cumulative rewards are on average roughly 550, while without using it they are roughly 450, that is still a good result, but not the best one.