DQN with Experience Replay for Gymnasium Cart Pole

Pasquale Mocerino

Introduction

The following report describes my implementation of a Reinforcement Learning Agent based on Deep Q-learning. The agent was implemented with Gymnasium (formerly OpenAI Gym) project, which provides an API for all single agent RL environments. In detail, I chose the Cart Pole environment, part of the Classic Control set. The agent uses a DQN (Deep Q-Network) to approximate the Q-function, and the Experience Replay technique is implemented.

1 Problem

The Gymnasium Cart Pole environment, described in the Gymnasium official documentation at [1], refers to the version of cart-pole problem defined in the paper [2]. In this scenario, a pole is attached to a cart by an un-actuated joint. The pendulum is placed upright on the cart, which moves along a frictionless track. The goal is to balance the pole by applying proper balancing forces to the cart in left and right directions along the track.

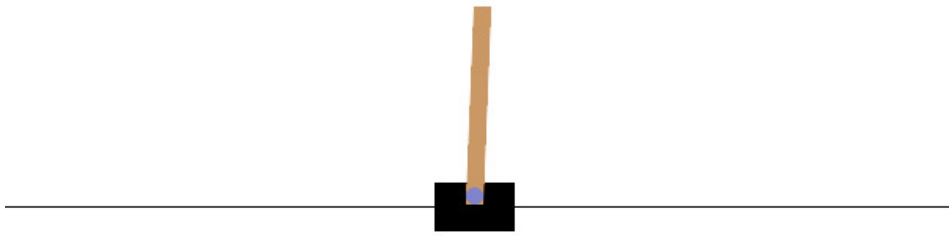

Figure 1: Cart Pole environment visualization.

1.1 Action Space

The action space is a *ndarray* with shape (1,). It can take values $0, 1$, indicating one of the two possible directions of the fixed force applied to the cart. In detail, 0 indicates a push to the left, while 1 indicates a push to the right. It is important to keep into account that if the force applied to the cart is constant, the same is not true for the velocity, whose reduction or increase depends on the current angle of the pole. In fact the position of the center of gravity of the pole determines the energy needed to move the cart underneath it.

1.2 Observation Space

The observation space is a *ndarray* with shape (4,), corresponding to 4 possible observation parameters, reported in the following table.

Num	**Observation**	**Min**	**Max**
0	Cart Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	-0.418 rad (-24°)	0.418 rad (24°)
3	Pole Angular Velocity	$-\infty$	∞

The finite ranges for Cart Position and Pole Angle denote all possible values for observation space of each element. However, they do not correspond to allowed values in a successful episode. Allowed values are in fact restricted to smaller ranges, in detail (-2.4, +2.4) for Cart Position and (-12°, +12°) for Pole Angle. So, in both cases, only half of the whole range of possible values, centered around 0, constitutes the range of allowed values.

1.3 Rewards

The goal is to keep the pole upright for as long as possible, so a reward of +1 for every completed step is considered, including the termination step. Better performing scenarios will run for longer duration, accumulating larger rewards. In my implementation, I experimented the training both on version 0 and version 1 of the environment. In the first case, the problem is considered as solved when the average reward over last 100 episodes reaches a value over 195 on a total of 200 steps. For version 1, instead, the maximum number of steps is 500, and the problem is believed to be solved when the average reward over last 100 episodes reaches 475.

1.4 Starting State

As a starting state, all observations are assigned a uniformly random value in the range (-0.05, 0.05). For uniformity across runs, I set a seed for the environment equal to 50.

1.5 Episode End

The episode ends if any one of the following condition holds:

- Pole Angle is outside the range (-12°, +12°), termination event
- Cart Position is outside the range (-2.4, +2.4), termination event
- Episode length is greater than the maximum number of steps, truncation event

For version 0 of Cart Pole environment, the truncation event is for episode length greater than 200 and the relative solution threshold is 195. For version 1, instead, the truncation event is for episode length greater than 500 and the relative solution threshold is 475.

2 Solution

Experience Replay is a technique used in RL which is based on the presence of a replay memory. Agent's experiences are stored at each time step in a dataset, pooled over many episodes into a replay memory. So memory is sampled randomly for a minibatch of experience, used to learn off-policy, as with Deep Q-Networks. This overcomes the problem of autocorrelation leading to unstable training. In fact RL is unstable or divergent when using a nonlinear function approximator as a NN to represent the Q-function. The instability is related to the correlations between observations in the sequence, hence small Q updates can significantly change the agent policy. Hence Experience Replay approach is used to remove correlations in the observation sequence and to smooth changes in data distribution. It is a biologically inspired mechanism, using a random sample of prior actions instead of the most recent action to proceed.
To perform Experience Replay, the algorithm stores all agent experiences at each time step in a dataset, so $\{s_t, a_t, r_t, s_{t+1}\}$ for a given t. In a normal Q-learner, we would run the update rule on them, while with this approach we just store them. Only later during the training process replays will be drawn uniformly from memory queue and ran through update rule.

The state space of Cart Pole environment is continuous, with real values for cart position, cart velocity, pole angle and pole angular velocity. So, as it is, it is not possible to create a Q-table. A possibility can be to create a function converting continuous state parameters into discrete values. However, the Q-table approach is not so space efficient and it cannot lead to the same results as the ones obtained with a DQN, whose structure is explained in this report.
Although the state space is continuous, there are only two possible actions. A discrete action space is suitable for a DQN solution as Q-function approximation.

3 Implementation Architecture

Since reference environment is deterministic, it is possible to formulate deterministic equations. The goal is to train a policy maximizing the discounted cumulative reward:

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$$

where γ is the discount factor, between 0 and 1, ensuring convergence. A lower γ makes rewards of the far future less important than the ones of near future, so also encouraging the collection of rewards closer in time.

The main principle of Q-learning is the possibility to construct an optimal policy $\pi^*(s)$ starting from the function $Q : S \times A \rightarrow R$, returning the reward for each pair of state and action. However, since the world is not completely known, it is not possible to obtain Q, so the same holds for the optimal policy, since:

$$\pi^*(s) = argmax_a Q(s, a)$$

Since NN are universal function approximators, the idea is to train a NN, more precisely a Deep Q-Network in this case, in order to make it similar to Q. As a training update rule, the idea is to use the Bellman equation, holding for Q:

$$Q^{\pi}(s, a) = r + \gamma \cdot Q^{\pi}(s', \pi(s'))$$

The error δ is the difference between the two sides of the equality:

$$\delta = Q(s, a) - (r + \gamma \cdot \max_a Q(s', a))$$

To minimize the error, I used the Mean Squared Error loss. An alternative is the Huber loss, which acts like MSE for small errors, but like mean absolute error for large errors, making the model more robust to outliers.

To train the DQN, I used an Experience Replay memory storing observed transitions. In this way, data can be reused later by random sampling, avoiding correlation. So, the model implemented is a DQN approximating Q-learning function and following the Experience Replay paradigm.
The environment state of CartPole agent is represented by 4 real values. These 4 values are given in input to a fully connected network with 2 outputs, one for each action.
The chosen Deep Neural Network was designed according to the will to solve the more complex version 1 of the environment. In a first phase, I tried several combinations of hyperparameters and network models, but I experienced many times a rapid decrease of the training average reward (over last 100 episodes) after reaching value that was high but not sufficient to solve the problem (the highest reached value was 450). So, after some modifications, I finally found the right hyperparameters in order to solve the problem. So, analysing the final model in detail, there are an input layer, three hidden layers and an output layer. The 4 input neurons, one for each state variable, are connected to the next hidden layer, made of 64 neurons. This layer is connected to the next hidden layer, again of size 64, while the third hidden layer is of size 32. Finally, the model returns two outputs corresponding to the number of possible actions. All layers are fully connected and, apart from last layer, ReLU activation function is always used. Model uses Mean Squared Error as loss function and Adam as optimizer. Furthermore, I used Kaiming Initialization, or He Initialization, an initialization method for weights taking into account the non-linearity of activation functions. The network is trained to predict the expected value for each action, given the input state. In fact, during the exploitation phase, the chosen action is the one associated with the maximum of the two outputs of the neural network, so the action with the highest expected value. The hyperparameters were chosen in the following way. The learning rate was set to 0.001, the discount rate γ to 0.95, the starting ϵ factor (ϵ_{max}) was defined as 1.00 and ϵ_{min} as 0.001, with a decay ϵ_{decay} of 0.995. The batch size was 64. The maximum size of the replay memory was set to 10000.
Here I report a summary of the model parameters:

Type	**Input**	**Output**	**Activation**
Dense	4	64	ReLU
Dense	64	64	ReLU
Dense	64	32	ReLU
Dense	32	2	Linear

Parameter	**Value**
Loss Function	Mean Squared Error
Optimizer	Adam
Learning Rate	0.001
Maximum Experience Size	10000
Discount Rate γ	0.95
Maximum ϵ	1.00
Minimum ϵ	0.001
ϵ Decay	0.995
Batch Size	64

Now I will analyse some important details of the implementation.

```
def action(self, state):
  if np.random.rand() <= self.eps:
      return random.randrange(self.nA)

  action_values = self.model.predict_on_batch(state)
  return np.argmax(action_values[0])
```

In the *action* method of defined *DNQ* class, the ϵ-greedy policy can be easily noticed. The agent chooses a random action (exploration) with probability ϵ and the action with max Q-value (exploitation) with probability $1 - \epsilon$. When testing, instead, only exploitation is possible (*test_action* method, trivially defined).

```
def store(self, state, action, reward, nstate, terminated):
  self.memory.append( (state, action, reward, nstate, terminated) )
```

As stated before, experience is stored in memory as a $\{s_t, a_t, r_t, s_{t+1}\}$ tuple. In the effective implementation, also *terminated* value is useful to know if the reached state is terminal or not.

```
def experience_replay(self, batch_size):

  minibatch = random.sample(self.memory, batch_size)
  np_mbatch = np.array(minibatch, dtype=object)
  st = np.zeros((0, self.nS))
  nst = np.zeros((0, self.nS))

  for i in range(len(np_mbatch)):
    st = np.append(st, np_mbatch[i, 0], axis = 0)
    nst = np.append(nst, np_mbatch[i, 3], axis = 0)

  st_pre = self.model.predict_on_batch(st)
  nst_pre = self.model.predict_on_batch(nst)
  x = []
  y = []

  for i, (state, action, reward, nstate, terminated) in enumerate(minibatch):
      if terminated:
          target = reward
      else:
          nst_act_pre = nst_pre[i]
          target = reward + self.gamma * np.amax(nst_act_pre)

      st_pre[i][action] = target
      x.append(state)
      y.append(st_pre[i])


  x_reshape = np.array(x).reshape(batch_size, self.nS)
  y_reshape = np.array(y)
  self.model.fit(x_reshape, y_reshape, verbose=0)
  self.model.save_weights(checkpoint_path)

  if self.eps > self.eps_min:
      eps_new = self.eps * self.eps_dec
      if eps_new < self.eps_min:
        self.eps = self.eps_min
      else:
        self.eps = eps_new
```

This is the implementation of Experience Replay. After sampling a random minibatch from memory, states and the next states are retrieved as Numpy arrays of batch size in order to take advantage of vectorization. If the state is terminal, the reward is simply assigned, while if the state is not terminal, the Q-learning update is performed. Since the given environment is deterministic, the update is:

$$Q(s,a) = r + \gamma \cdot max_a(Q(s',a'))$$

So, the model is finally trained on state and relative Q-function values for possible actions.

```
for e in range(MAX_EPISODES):

    observation, info = envCartPole.reset()
    state = np.reshape(observation, [1, nS])
    c_reward = 0

    for time in range(500):

        action = dqn.action(state)
        obs_next, reward, terminated, _, _ = envCartPole.step(action)
        c_reward += reward
        nstate = np.reshape(obs_next, [1, nS])
        dqn.store(state, action, reward, nstate, terminated)
        state = nstate

        if terminated or time == 499:
            rewards.append(c_reward)

            if len(dqn.memory) > BATCH_SIZE:
                  dqn.experience_replay(BATCH_SIZE)
            break

    if len(rewards) > 100 and np.average(rewards[-100:]) > 475:
        NUM_EPISODES = e
        break
```

Finally, the upper code is the most relevant extract from the main training cycle, without the code for collection of useful information for plotting. For each new episode, the cumulative reward is set to 0 and a new observation is performed. So, it is tried for 500 times to do a step by choosing the action returned from the model by applying the ϵ greedy policy. The reward is collected and the tuple is stored in memory. If the pole fell down or the maximum number of steps has been reached, the episode ends, so the final cumulative reward is collected and Experience Replay is performed (if memory contains a number of elements to randomly sample at least equal to batch size). At the end of each episode, the termination condition is checked. This code obiouvsly refers to CartPolev1 training. The modifications for CartPolev0 are trivial (200 steps and 195 as threshold value).

4 Results

4.1 CartPolev1

Here are reported the results of the training process. As specified before, the reward goal is set to the 475 threshold, considering 500 time steps for each episode. Hence the stopping condition holds when the average reward over last 100 episodes reaches this threshold. The training process of the network lasted for 646 episodes, after which the reward goal was reached. The training time was around 15 minutes. This is the reward evolution:

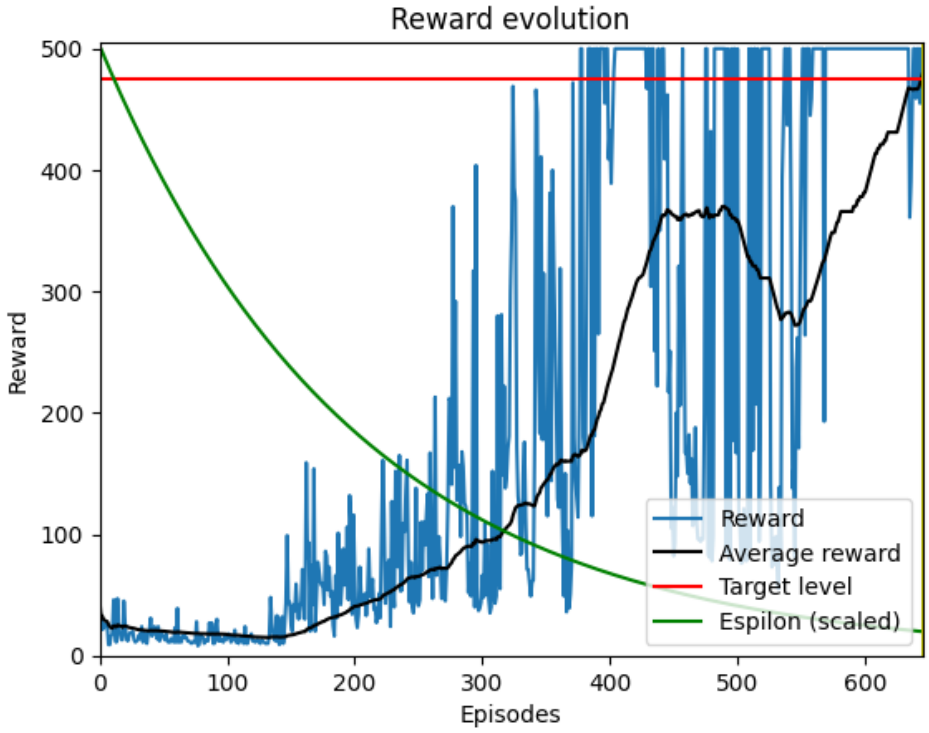

Figure 2: Training results on CartPolev1.

The general trend is largely oscillating in the first phase, but it stabilizes on higher rewards at the end, also keeping the maximum possible value of 500 for several epochs. For simplicity, I trained the model in Google Colab environment, but obviously it can be done locally as well. I had some issues with Google Colab rendering of the environment, in fact I had to use a wrapper function (with *rgb_array* as render mode instead of *human*) for video visualization. The display is easier to be executed locally. However, I managed to see visually that the agent has effectively learnt how to keep the pole upright, with a very good performance.

4.2 CartPolev0

For completeness, I also tried to use the same model to solve the previous version of the environment, which is deprecated but can be used. So, I simply modified the training process by considering the maximum number of steps as 200 and by setting the new threshold to 195. In this case, the process lasted for 618 episodes. Here it is the training evolution:

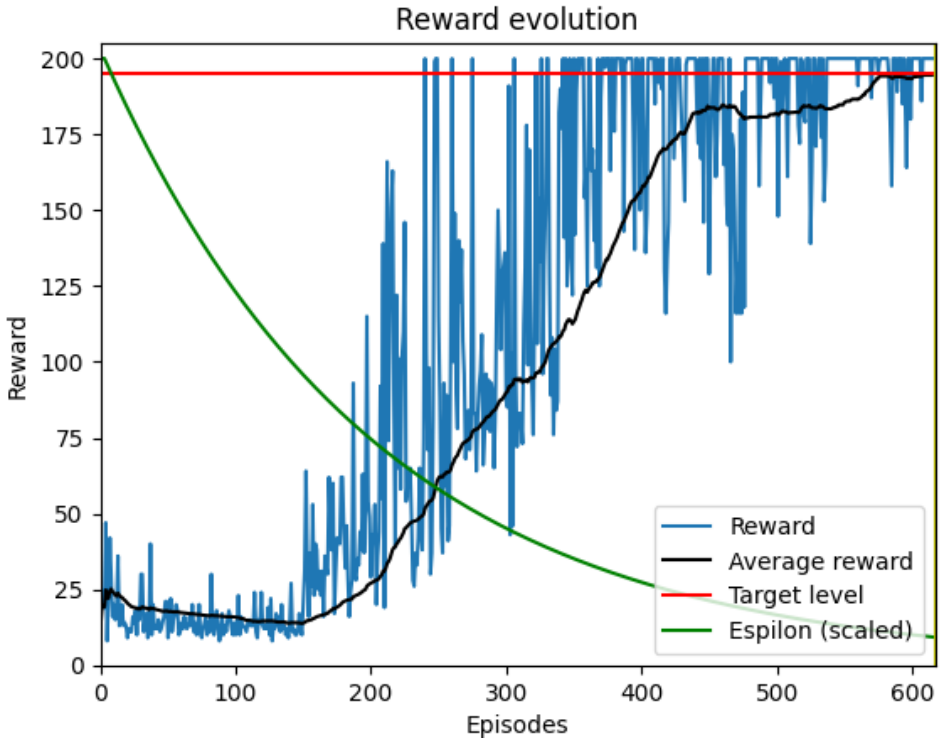

Figure 3: Training results on CartPolev0.

As expected, in this case the training process is more linear and less oscillating, because of the simplicity of the environment. Also the training time was reduced to around 8 minutes, so substantially the half of the previous case.

5 Final Remarks

I managed to solve the Gymnasium Cart Pole environment by using a Deep Neural Network to approximate the Q-function. I solved both version 0 and version 1 with the same network, obtaining slightly different training results. In both cases, the trained agent performs very well, managing to keep the pole upright for a long time.

In the *src* folder, I reported a *colab* folder containing the used Google Colab notebook and two subfolders with the final weights and the final video performance for both versions. The *code* folder, instead, contains a single source file with all used code plus the functions for local rendering and instructions to test it in a Python virtual environment.

References

[1] Farama Foundation. Cart Pole - Gymnasium Documentation. `https://gymnasium.farama.org/environments/classic_control/cart_pole/`.

[2] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.