Cart Pole

Reinforcement Learning Agent based on single deep Q-learning

Anna Carini, 1771784

Matteo Simonetti, 1854835

Project idea

Our project is divided into two parts: first we trained an agent on the classic control environment "Cart Pole". Then we modified the environment, to make it more complex, and we adapted our previous solution to the new environment.

Main files:

- *CartPole.py* agent file for the unmodified environment
- *CartPole_Mod_Env.py* modified environment file
- *CartPole_Mod.py* agent file for the modified environment

Problem

The agent was trained with the Python Gymnasium API, using in particular the CartPole-v1 environment.

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.
The agent can either move left or right (respectively action 0 and 1) and can observe 4 values which are: cart position, cart velocity, pole angle and pole angular velocity.
When the pole angle is not in the range [-12°, 12°] the episode terminates. The episode also terminates if the cart position is not in the range [-2.4, 2.4].
The starting state assigns to each observation a random value in [-0.05, 0.05].

Solution evolution

The solution consists in a Reinforcement Learning Agent based on Q-learning, which uses a neural network to approximate the Q-function.

First solution: We created a NN with four layers, the input layer had 5 nodes, with 4 inputs being the state and 1 being the action, then 2 hidden layers (32 and 16 units) and an output layer with one node. The output was the Q function.
This solution also included an *epsilon-greedy* strategy and an experience array (observations, action, expected reward) for the experience replay, with the expected reward being:

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

This first solution performed really badly, so we applied many changes, including:

- architecture adjustments (number of layers and nodes)
- hyper-parameters tuning (max epsilon, learning rate, epsilon reduction, episode batches)
- experience replay with more than 1 buffer
- dropout (random nodes elimination)

Solution evolution

Then we started to try some debugging. We printed the action the agent was making, we noticed that quite often it was always the same action, here we started to hypothesize that the agent was getting stuck in local minima.
The first thing we did was changing the death reward from 1 to 0, because the environment was still giving 1 as reward even if the agent died. This didn't change much.
We abandoned for a moment the local minima idea and we changed the whole NN architecture from (obs,a)->Q(obs, a) to obs->(Q(obs, a1),Q(obs, a2)), now the training was done by saving in the experience buffer (obs, reward0, reward1), with one of the rewards given by the action taken and the other reward obtained from the NN prediction. This gave better results, but was still far from perfect.
We made a change such that the learning rate would decrease during the training.
We increased epsilon decay, so that after a cyclic number of episodes epsilon would go towards 0 a little faster.

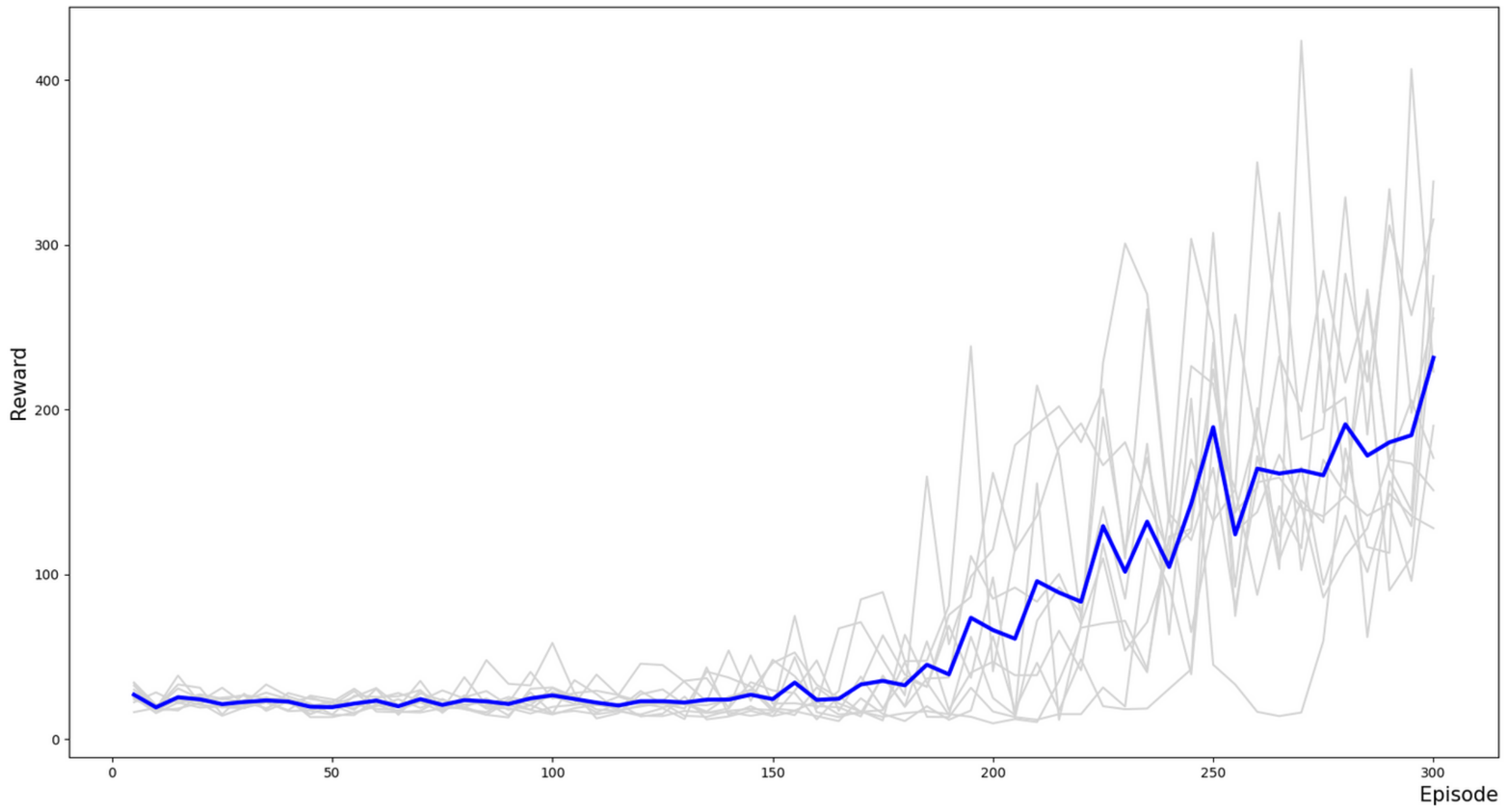
Final solution

Finally we went back to the local minima hypothesis and we substituted the reward for termination from 0 to -50, this was the game changer that lead us to a stable and high-rewarding result.

The final neural network architecture is made of five layers with the following amount of nodes: 4, 256, 256, 64, 2. It takes as input the four observations and returns the expected rewards associated to the two actions. We added two dropout layers to prevent overfitting.

The training is done every five episodes, using an experience replay buffer in which the reward associated to the action taken is -50 in case of termination, and 1 + (discount factor)*(expected future reward) otherwise.
The learning rate decreases during the training to get a better convergence.

Average reward on 10 runs



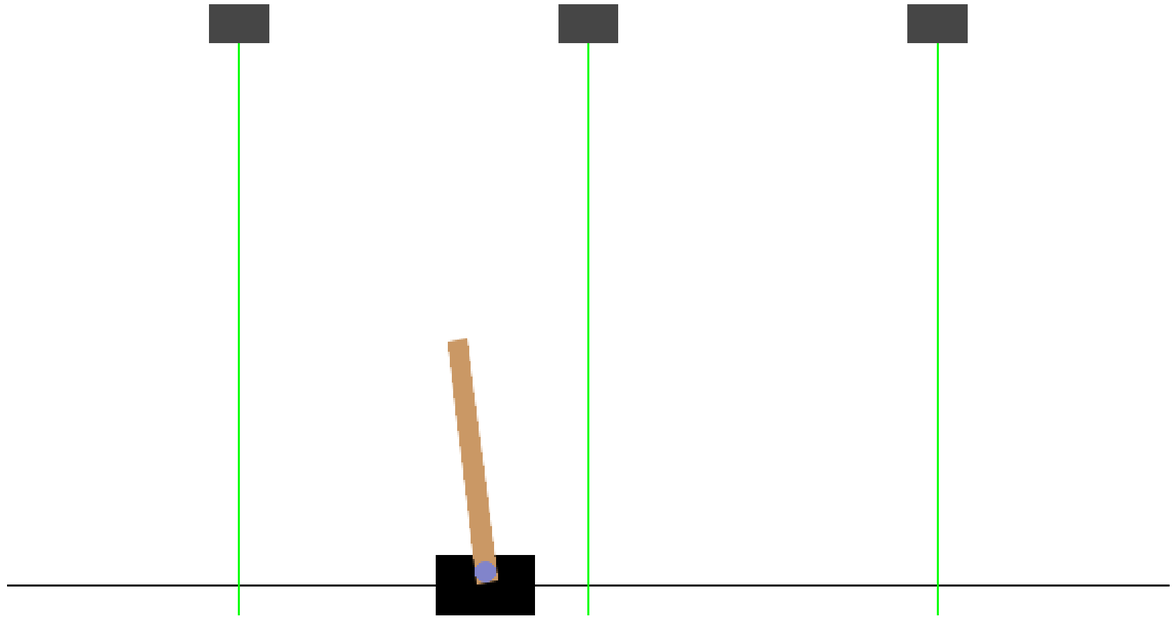
On the y axis there is the reward and on the x axis the episode number.

Each reward (in grey) was computed as the average reward for batches of 5 episodes.

In blue there is the average of all the grey rewards. After 300 episodes 250 is the usual reward.

New problem

Once we got a satisfying result with the standard cartpole, we thought of making some changes to the environment in order to make it harder for the agent to survive. We added 3 intermittent lasers which, when one of them hits the cart, terminate the round. The environment keeps track of a steps counter, which goes from 0 to 89, so that the lasers are toggled every 45 steps.



New environment

To be able to train the agent on this new environment, we added one extra observation which is called "counterNormalized", which is the normalized version of the steps counter, and it has values in the range [-1, 1].

The normalization is done in order to avoid having values too big with respect to the other 4 observations.

The reason why we added the counter as observation was to make the transition of the state deterministic, because at any step the lasers are on or off exclusively depending on the counter's value.

We also inserted the negative rewards into the environment instead of having them in the agent file, and they are -55 when the pole falls and -5 when a laser is hit.

New solution

This solution is based on the final solution presented before.
Some hyperparameters, like the learning rate, were modified to improve the rewards.

Since the environment is more complex, we had to make the neural network bigger, so the architecture was changed to five layers of 5, 512, 350, 128, 2 units.
For the same reason epsilon is also decreased more slowly and with a constant decay, to make the training last longer.

A big difference w.r.t. the previous solution is that we now use 3 experience arrays instead of 1. One of them is used to save the experience regarding the "central" section of the space (i.e. when the position of the cart is in the range (-0.09, 0.09)), the other two are used for the "left" and "right" parts.

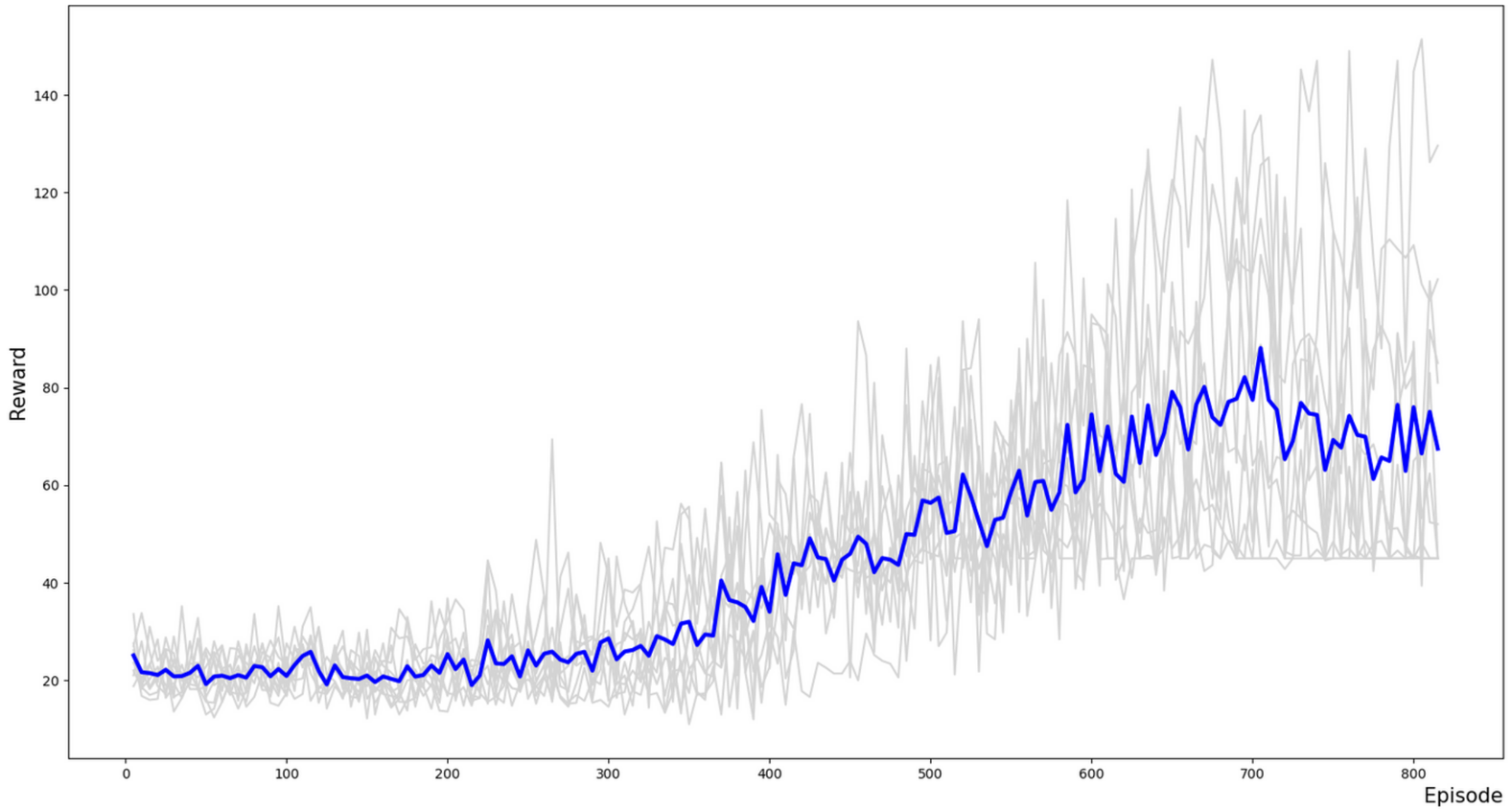
New solution

The reason is that we want the agent to balance the pole in a position different from the center, because one of the lasers is in the middle of the space.

But the cart is always initialized in the center, so if we use only one array (of limited size) then the data regarding the left and right parts of the space get constantly overwritten, and the training is mostly done on observations regarding the central part.

To avoid this, whenever we train the agent, we randomly sample data from the concatenation of these three experience arrays.

Average reward on 10 runs



The chart structure is the same as before.

We can notice that the blue reward is quite smaller. This is due to the fact that the environment is stricter.

Still there were peak performances of 150, which denote a good behavior.

Details

CartPole

Neural network:
- input layer: 4 units (ReLU)
- 3 hidden layers: 256, 256, 64 (ReLU)
- output layer: 2 units (linear)
- loss: mean squared error

Epsilon: 95 → 5 (decay: 1.5 → 1.8)

Learning rate: 0.012 → 0.0006

Rewards:
- pole fallen: -50
- else: 1 + γ*(future reward)

Experience replay:
- buffer size = 2000
- training batch size = 600

Modified CartPole

Neural network:
- input layer: 5 units (ReLU)
- 3 hidden layers: 512, 350, 128 (ReLU)
- output layer: 2 units (linear)
- loss: mean squared error

Epsilon: 95 → 5 (decay: 0.55)

Learning rate: 0.012 → 0.001

Rewards:
- pole fallen: -55
- laser hit -5
- else: 1 + γ*(future reward)

Experience replay:
- buffers size = 2000 (left, right), 900 (center)
- training batch size = 900