Training a DQN agent to play Cartpole and Pong using Reinforcement Learning

29.11.2024

Nima Hadavi[1], Borja García Pascual[2]

[1] nhadavi@student.uef.fi, [2] borja.garcia.pascual@uef.fi

1. Problem description

1.1. Deep Q-Learning

In a Reinforcement Learning (RL) paradigm, agents learn how to behave by trial and error. Each time the agent chooses the right action it is rewarded, and it is punished otherwise. After many learning cycles, intelligence emerges as agents become capable of choosing the optimal set of actions based on the past experiences it received from the environment.

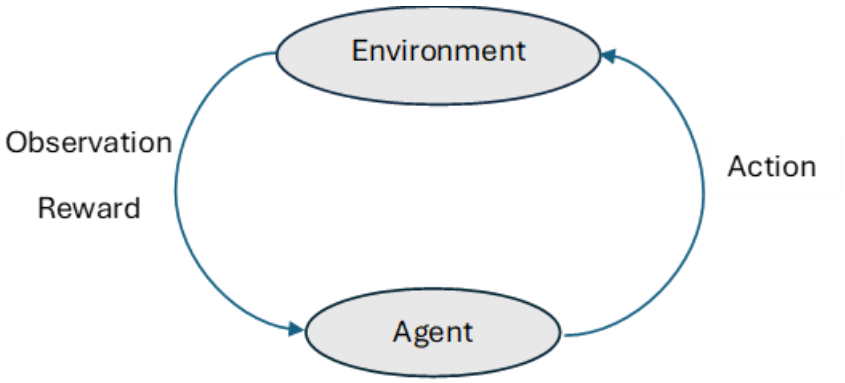

Figure 1. Reinforcement learning cycle.

First, the agent receives an observation s_t from the environment, which is the initial state that the agent is in. Then, the agent chooses an action a_t based on the current state. This action modifies the environment and generates a new state s_{t+1}, and a reward r_{t+1} for such action. This reward can be negatively valued (punishment) or positively valued (reward). Hence, the goal is to maximize the total expected future rewards, while balancing between immediate and long-term rewards.

To model how good each possible action is, we make use of the Q-value. The Q-value of an action $Q(s, a)$ in a state is the maximum sum of rewards that we expect to obtain if we start playing in sate s, take action a based on policy π, and interact with the environment until the end of the game. This value, however, is difficult to compute, especially when the optimal course of action is not known a priori.

To deal with this limitation, we resort to Deep Q-learning, which is the combination of Q-learning algorithms and neural networks, the best-known example being the DQN (Mnih et al., 2015). In Deep-Q learning, a neural network is trained to approximate the Q-value function. This way, the algorithm can handle environments with large or continuous state spaces, whereas in traditional Q-learning it would be computationally infeasible. Hence, the network takes state s as the input and outputs the Q-values for all possible actions.

1.2. Environments description: Cartpole and Pong

In **Cartpole** the agent controls a cart which has a pole attached to it and the goal is to prevent the pole from falling for as many timesteps as possible. The **state space** is a four-dimensional vector $s \in \mathbb{R}^4$ (position of the cart on the horizontal axis, the speed of the cart, the angle of the pole, and the speed at which the pole falls), and the **action space** is two-dimensional $a \in \mathbb{R}^2$ (move left, move right).

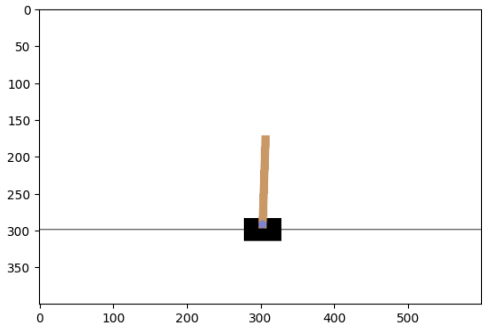

Figure 2. Initial Cartpole state, rendered as an image.

For every timestep that the agent keeps the pole up right and the cart within the boundaries, it receives a +1 **reward**. The episode terminates if the pole angle exceeds ± 12 degrees from vertical, if the cart moves more than ± 2.4 units from the center (out of bounds), or if the episode reaches a maximum of 200 timesteps (for CartPole-v0).

Whereas in **Pong** the agent controls the vertical position of its paddle, and the goal is to score more than the opponent. The **state space** is a matrix of size $210 \times 160 \times 3$ $s \in \mathbb{R}^{210 \times 160 \times 3}$ (the pixels of each frame in RGB), and the **action space** is six-dimensional $a \in \mathbb{R}^6$. Possible actions are:

0. No operation
1. Fire (used in games where firing is an option but here it has no effect)
2. Move paddle up
3. Move paddle down
4. Move paddle up and fire (equivalent to move up)
5. Move paddle down and fire (equivalent to move down)

If the ball passes the agent's paddle it will receive a reward of -1 and if the ball passes the opponent's paddle it will receive a reward of +1, otherwise reward 0 will be given for the state. The episode ends when one side reaches the winning score.

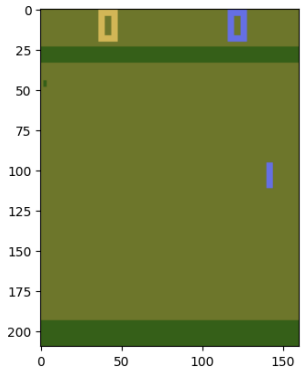

Figure 3. Initial Pong state, rendered as an image.

2. Approach to solution

Installation instructions for the packages used in this project can be found in this PyTorch tutorial https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html, and in the Gymnasium web site https://gymnasium.farama.org/.

2.1. Terminology

Some terminology that is used throughout the report:

Trajectory is the set of elements that are produced when agent moves from one state to another e.g. $(s_0, a_0, r_1, s_1, a_1, r_2, s_2)$.

Episode is a trajectory that starts from the initial state and ends in a terminal (final) state.

Reward is the immediate result that our actions produce.

Return is the sum of rewards from a certain point in time until the task is completed. The goal is to maximize the episodes' return.

Discount factor (γ): reduces the reward as time goes by. 0.99 chosen in the present work.

Policy is mapping from past experience to actions. It is a function that decides what action to take in a particular state, it takes the state as an input and outputs an action.

Replay memory is the memory that stores the state transitions that the agent experiences. The memory has a predefined capacity and when it fills up new transitions will replace the oldest ones. For pong-v4, we chose 50000 as we had limitations with the memory size.

2.2. Observations preprocessing

The **PreprocessEnv class** creates a wrapper that wraps the environment so that the values that come out of the environment can be modified to be used by PyTorch.

Regarding the **pong-v4**, as noted in the paper by Mnih et al. (2015), using the original, RGB-rendered arrays might be too computationally expensive. For that matter, the luminance is extracted from each frame by applying the mapping $\phi: \mathbb{R}^3 \rightarrow \mathbb{R}^2$

$$\phi(R, G, B) = 0.2126 \times R + 0.7152 \times G + 0.0722 \times B$$

Then, the resulting matrix is rescaled to be 84×84 pixels in size. To speed up training, $k = 4$ frames are stacked together before being fed to the model. The same action taken by the model is repeated k times, once per frame. In addition, pixel-wise maximum value between consecutive frames is used to avoid flickering. Therefore, the final size of the matrix fed to the model is $84 \times 84 \times 4$.

2.3. DQN agent

For **CartPole** we used a fully connected feed forward neural network with the 4 state dimensions as the input layer, first hidden layer with 128 neurons and ReLU activation and second hidden layer with 64 neurons and ReLU activation. Finally, we used 2 output nodes which estimated the Q-values for the 2 actions that we can choose. We call this the **q_network**.

As for the **Pong-v4** because the agent only has access to rendered images, it becomes advantageous to use a CNN instead of a fully connected network to analyse the frames. Therefore, we adopt this architecture in the present work.

DQN model architecture

The CNN model used takes the stacked frames of size $84 \times 84 \times 4$ as input, and outputs the modelled Q-values of each of the 6 possible actions. To do so, the following layers are stacked:

1. ***Convolutional layer 1*** - with 32 filters of size 8×8 applied with stride 4.
2. ***ReLU layer 1***.
3. ***Convolutional layer 2*** - with 64 filters of size 4×4 applied with stride 2.
4. ***ReLU layer 2***.
5. ***Convolutional layer 3*** - with 64 filters of size 3×3 applied with stride 1.
6. ***ReLU layer 3***.
7. ***Flatten layer***, where the matrix is stretched into a vector.
8. ***Fully connected layer 1*** - with 512 neurons.
9. ***ReLU layer 4***.
10. ***Output layer*** - with as many neurons as possible actions, that is, 6.

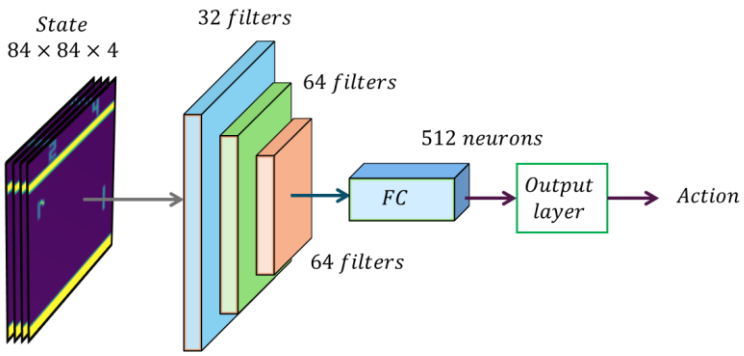

Figure 4. Architecture of the DQN model.

2.4. Training algorithm

Target-network

We also create a copy of the q-network and call it the **target_q_network** this network will be updated periodically and is used to generate stable target Q-values for training. It reduces instability by keeping its parameters fixed for a period.

Policy

At each step, the **policy** will either take a random action with probability value or use the Q-network to choose the action with the highest Q-value as $a_t = \arg\max_a Q\,(s, a|\theta)$. As for the Pong version, we chose an ε-greedy policy to have room for exploration at the beginning of the training phase. The ε value starts at 1 for maximum exploration and is reduced to 0.1 over the first 1 million frames linearly ($\epsilon_t \in [0.1, 1]$), allowing the agent to explore the action space before having to make its own decisions.

Memory

For **Cartpole** a replay memory of size 100000 is created, sampling will only be done if the number of instances available in the memory are at least 10 times the batch size to ensure there are enough instances for sampling. Samples will be used to train the neural network on each iteration.

Regarding the **Pong-v4**, contrary to the Cartpole case, each state s_t is an $84 \times 84 \times 4$ matrix, which makes it **unfeasible to store millions of transitions in a consumer-grade computer**. Therefore, we use a lower memory capacity value of 50000 during training.

Training loop

The main function **deep_q_learning** will take the **q_network**, the exploratory policy, the number of frames, alpha value, batch size, gamma, and epsilon as the input, train the **q_network** with these values, and record the MSE loss and Returns on each iteration and output them as stats. To update the parameters of the **q_network** we used the optimization algorithm AdamW.

For as long as the number of frames seen by the algorithm remains below some defined threshold, an action will be chosen by the policy and fed to the environment to get the next state, reward, and a Boolean value (which indicates if the task is done). Then these values along with the current state will be stored in the replay memory as one instance. If the memory does not have enough instances for the sampling to be done, this process will continue until it does. Then a batch of the instances in the memory will be sampled which will be split into state batch, action batch, reward batch, and next state batch.

Q-value of the present state will be calculated by calling the **q_network** and passing the states batch as input. Then, the Q-values for the next state and the next actions of each transition will be calculated by calling the **target_q_network** and passing the next state batch as input.

Next, the highest Q-value will be chosen for each transition to compute the estimated ground truth y as

$$\mathrm{y} = r + \gamma \max_{a'} Q\,(s', a'; \theta_i^-)$$

where r is the reward batch, s' is the next state batch, and a' is the next step action.

Then the loss will be calculated by calling the MSE function with the calculated Q-values for the present state $Q(s, a; \theta_i)$, and target batch values y. The calculated loss on each step, as well as the return of each episode will be stored in stats for plotting purposes.

$$\mathrm{L_i}(\theta_\mathrm{i}) = E_{(s,a,r,s')\sim U(D)}\left[\left(\mathrm{y} - Q(s, a; \theta_i)\right)^2\right]$$

Based on this loss function, the values of the neural network are updated at each step. Furthermore, every 10 episodes the **target_q_network** will be updated with the weights of the DQN agent **q_network**.

In the Cartpole case we trained the DQN agent for 500 episodes. As for Pong, the agent was fed 20.000.000 frames during the training phase.

3. Implementation

Two notebooks can be found attached to the report: one named “cart-pole.ipynb” containing the implementation for the Cartpole environment, and another called “DQN_Pong-v4.ipynb”, which contains the implementation for the environment Pong-v4. In addition, the code and trained models have been updated to a GitHub repository freely available: https://github.com/crimsonima/RL-AI.

4. Discussion on intelligence

4.1. Cartpole

As can be seen, the returns started improving significantly around episode 150 and reached the maximum of 200 by approximately episode 400. This indicates that the model was able to learn from past experiences and adapted its actions to achieve the goal of maximizing returns. This behavior demonstrates the agent's ability to act intelligently within the defined environment.

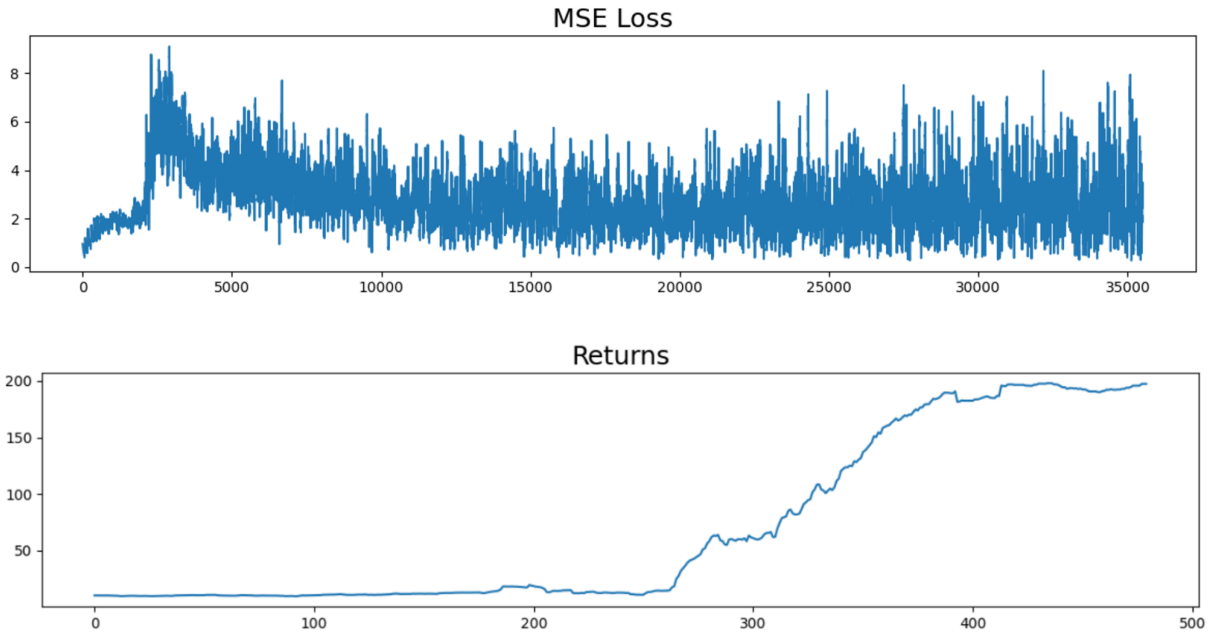

Figure 5. MSE and accumulated returns learning curves for Cartpole agent.

A display of the agent's behavior in the environment can be seen in the GIF file "cartpole_agent_play.gif".

4.2. Pong

The Mean Squared Error (MSE) loss was stored at each update cycle, and the returns were summed and stored every episode. This allowed us to compute the learning curves for the DQN agent. These curves strongly suggest the agent would have kept learning for much longer. Nonetheless, it is clear that our agent exhibits signs of intelligent behavior as accumulated reward at each episode approaches 0.

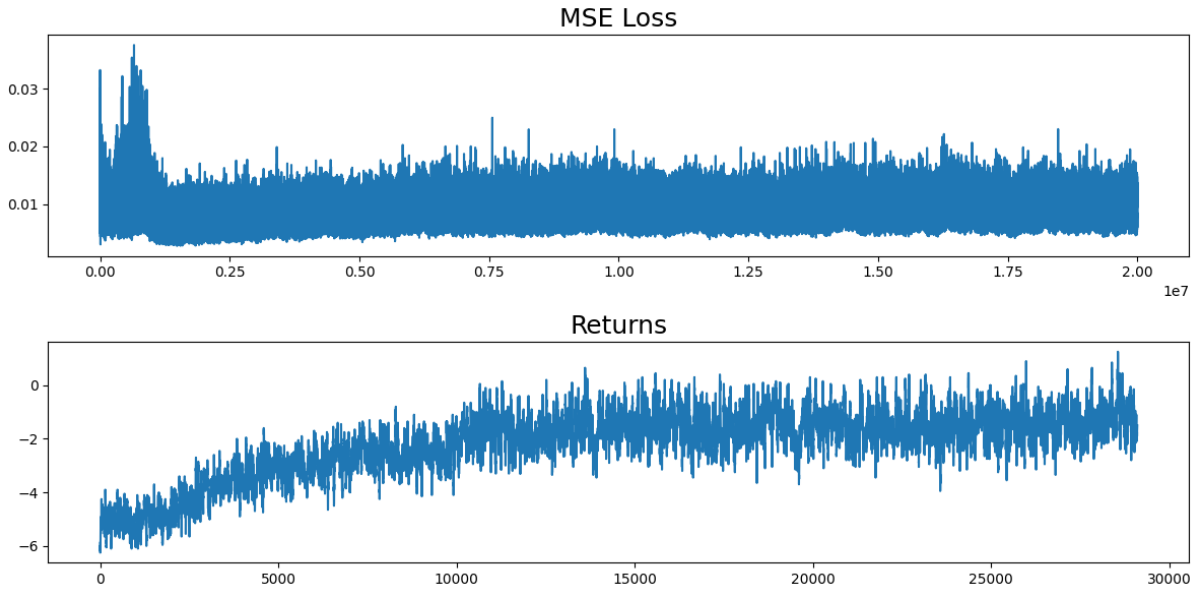

Figure 6. MSE and accumulated returns learning curves for pong agent.

Finally, the behavior of the DQN agent was visually evaluated by making it play Pong for 200 frames. As can be seen in the short gif "pong_agent_play.gif", the agent performs well and shows signs of intelligence, especially when taking the following behaviors into account:

- **Staying put and waiting for the ball** when the direction of the ball matches the current position of the agent.
- **Scouting throughout the field** when it is unclear in which direction the ball will go.
- **Correcting its position** when the ball approaches the agent's side of the screen, but it is still far from the ball.

5. Quotations

The training loop used in this project is inspired by the Udemy course "Reinforcement Learning beginner to master - AI in Python". Furthermore, we used ChatGPT to assist in the coding of helper functions. Finally, we took some inspiration and code for the CNN model in the PyTorch tutorial https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

6. Contributions

Both students contributed equally to programming, training, and testing the algorithms, as well as writing the report.

7. Reflections

The initial phase of the learning process is characterized by erratic behavior and an increase in the value of the loss function. As described earlier, during that time agents'

actions are mostly random (because of the exploration policy) and are meant to help the agent understand the action space better. After epsilon reaches its final value of 0.1, both MSE and accumulated reward start to slowly but steadily improve, meaning the agent is learning to take its own actions.

These curves indicate the model would have kept learning for much longer. In fact, the original paper recommends exposing the agent to some 50.000.000 frames, more than twice as long as we trained our DQN agent. However, consumer-grade computers are somewhat limited, so we were forced to reduce both the memory size and the training time.

8. Bibliography

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529-533.

Escape Velocity Labs (2024). Reinforcement Learning beginner to master - AI in Python [Online course]. Udemy. https://www.udemy.com/course/beginner-master-rl-1/

Paszke, A. (2024). Reinforcement Learning (DQN) Tutorial [Online tutorial]. PyTorch. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html