Deep Reinforcement Learning

Assignment 3

Francesco Manzali

26/06/2021

In Reinforcement Learning (RL), an agent learns which actions to take within an environment to maximize a given reward. A common RL algorithm is Q-learning, based on finding a mapping between pairs of states and actions and their expected cumulative rewards. Since states are generally highly numerous, deep neural networks can be exploited to learn such mapping, leading to Deep Q-learning Networks (DQNs). In this homework, DQNs are applied to solving three simple tasks, using two environments from OpenAI `gym`.

Introduction

Consider an *agent* $\mathcal{A}$ acting within an environment $\mathcal{E}$. At each timestep t, $\mathcal{A}$ is in a state s_t, performs an action a_t, receives a reward r_t and moves to a new state s_{t+1}. The goal of $\mathcal{A}$ is to choose the correct actions to maximize the (discounted) total return G_t:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{+\infty} \gamma^k r_{t+k} \tag{1}$$

where $\gamma \in [0, 1]$ is the discount factor. A low γ means that the agent is interested in short-term gains, while a $\gamma \approx 1$ means that long-term rewards are important as well.

In Q-learning [1], the agent learns the *value* of each state-action pair $Q_\pi(s, a)$, i.e. the expected cumulative return resulting from choosing a_t at state s_t, and then sampling all future moves a' in each future state s' from a fixed policy $\pi(s')$:

$$Q_\pi(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a, \pi] \underset{(1)}{=} \mathbb{E}_{s'}\big[r + \mathbb{E}_{a' \sim \pi(s')}[Q_\pi(s', a')] | s, a, \pi\big] \tag{2}$$

where the expected value is over the distribution of *next states* s' reached by choosing a in s.
Choosing the best possible policy leads to the optimal Q-value function $Q^*(s, a) = \max_\pi Q^*(s, a)$, which in turn induces a definition for an optimal policy, which is just "choose the action maximizing the Q^* of that state": $a = \arg\max_{a'} Q^*(s, a')$. Substituting back in (2) leads to the Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s'}\big[r + \gamma \max_{a'} Q^*(s', a') | s, a\big]. \tag{3}$$

This equation can then be used to learn Q^* with a neural network.

In this homework, Deep Q-learning is applied to two environments, offered by OpenAI `gym` [2]:

1. **CartPole**: the goal is to vertically balance a pole attached to a moving cart. This is done in sec. 1 by using as state the phase-space coordinates of the cart and the pole. In sec. 2 the same environment is tackled by using instead an image rendering of the system.

2. **Pong**: the Atari table tennis game from 1972. The goal is to defeat a simple bot which continuously tracks the ball, by using an image rendering of the screen (sec. 3).

1 CartPole

The agent controls a cart C moving horizontally on a 2D screen, with the goal of vertically balancing a pole P attached to it. The state is a 4-tuple $(x_C, \dot{x}_C, \theta_P, \dot{\theta}_P)$ of phase-space coordinates. At every frame of the simulation, the agent can apply a fixed impulse to the cart, either in the left or right direction.

1.1 Methods

A neural network is used to return the Q-values for all actions given a state. The chosen architecture consists of just one hidden layer with 128 hidden units, followed by a Tanh activation.

To stabilize training, two copies of the network are used: one is updated at every step (*online* network), while the other (*target* network) is kept fixed for a certain amount of frames, and then its parameters are synced with the ones from the *online* counterpart. In this way, when minimizing the distance between the two terms of (3) to learn Q^*, the same network does not appear in two places at once:

$$\text{Minimize:} \quad \mathcal{L}\left[{}^{online}_{net}(s_t, a_t); r_t + \gamma \max_{a'} {}^{target}_{net}(s_{t+1}, a')\right] \tag{4}$$

where $\mathcal{L}$ is the *loss* function, chosen to be the `SmoothL1Loss` from PyTorch, which behaves like a L1 loss for errors over 1, and a quadratic (L2) loss for smaller errors, while being differentiable everywhere.

Some more considerations are done to facilitate learning:

- **Replay memory**. Training samples are 4-tuples of (s_t, a_t, s_{t+1}, r_t), which are generated by simulating the agent within the environment. However, vectors belonging to the same episode (i.e. the same run of the simulation) are highly correlated, and training directly on them would force the network to "specialize" on each single episode, without generalizing. Thus, samples are first collected in a buffer (replay memory), from which random batches are extracted to train the network. In this way, samples shown one after the other may belong to very far timesteps (or entirely different simulations), and are thus less correlated.
- **Exploration profile**. It is important that the training samples are representative of the whole state/action space. Thus, the agent is incentivized to initially explore the environment, by taking random actions.

  Specifically, two exploration policies are considered:

  - **Epsilon-greedy**: at each step, the agent picks a random non-optimal action with probability ϵ, and the best action otherwise.
  - **Softmax**: the agent samples an action from a softmax distribution at temperature T on the Q-values of the available actions.

  T and ϵ are initially set to high values, and decrease exponentially during training, thus allowing the agent some time to explore the environment before settling in a learned optimal behavior.
- **Adaptive learning rate**. The learning rate is artificially reduced when the score rises. This reduces the likelihood of the model "catastrophically forgetting" a good solution, thus improving convergence.
- **Early stopping**. If the score reaches the maximum value (500) for 10 consecutive epochs, training is stopped.
- **Gradient clipping**. Gradients during training are clipped to a maximum norm of 2, to lower the risk of "catastrophic forgetting".

Small tweaks to hyperparameters highly affect the speed and quality of convergence in RL. A small scale search is manually done, with the goal of solving the CartPole environment with less than 300 episodes.

1.2 Results

The final network is trained with the Adam optimizer, with an initial learning rate $\lambda = 0.1$, which is artificially reduced to 10^{-2}, 10^{-3} and 10^{-4} if the score surpasses 100, 200 or 450 respectively. A batch size of 256 is used.
The agent follows a softmax policy with initial temperature $T = 5$, decreasing after each episode n to:

$$T(n) = 5 \exp\left(-n \frac{\ln 5}{6 \cdot 400}\right)$$

The discount factor is $\gamma = 0.98$, and a small reward of $-|x_C|$ is added to penalize the agent from moving the cart too far from the screen's center. The final learning curve is shown in fig. 1.
The agent completely solves the environment after just ~ 175 episodes, reaching the maximum score of 500. A video rendering of a test is available at this link.

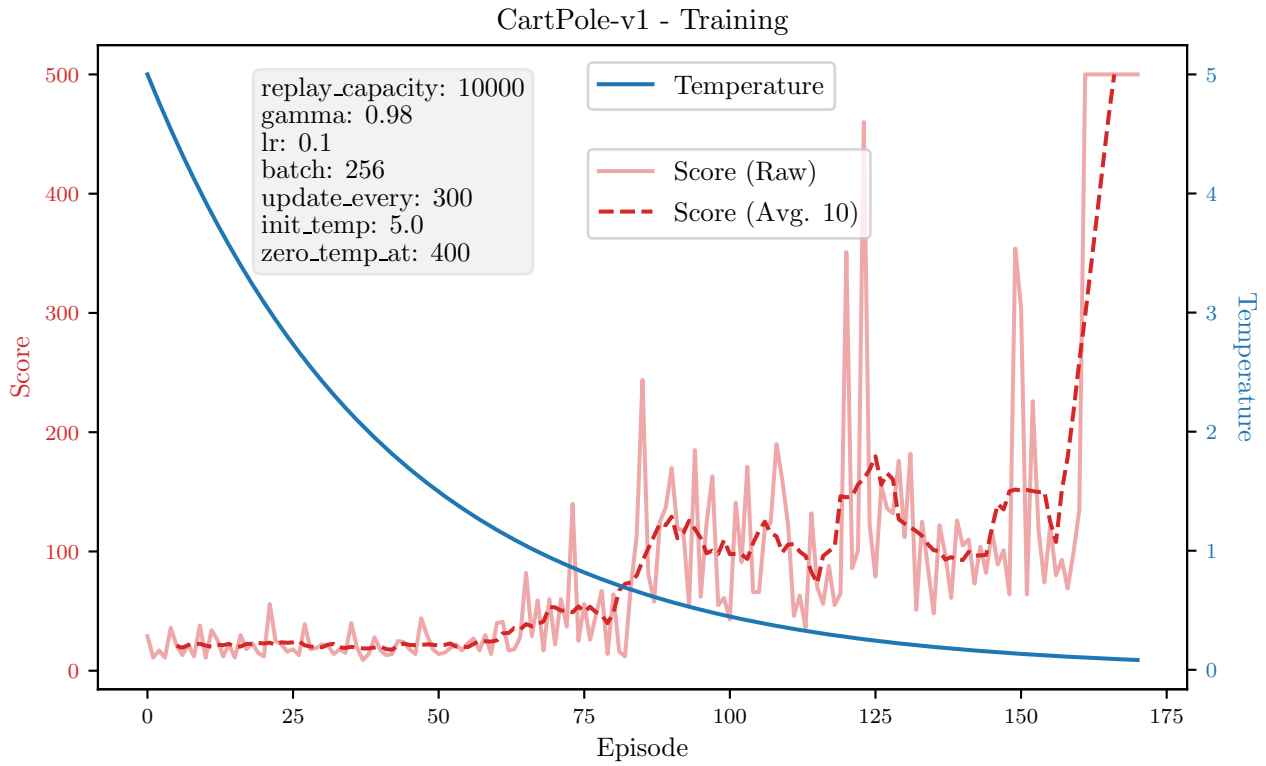

Figure (1) – Learning curve for CartPole. The temperature is updated at every episode, while the target network is synced every 300 frames.

2 CartPole with Pixels

The same environment from sec. 1 is considered. This time, however, the network must learn which features are needed to solve the task directly from an image rendering of the screen.

2.1 Methods

The CartPole environment can render the screen as a 600×400 RGB image. However, most of the pixels are useless to solve the task, since the cart occupies just a small fraction of a frame. Thus, images are preprocessed to reduce their size:

- Only the first channel is kept, and it is converted to binary values: 0 for the background, 1 for the pixels of the cartpole.
- A rectangular region of 200×125 is cropped around the position of the cart, and eventually padded with zeros to maintain a fixed dimension. In this way, the network will always see the

cart as stationary, with only the pole moving. This removes all information about x_C and $\dot{x}_C$. However, a term $-|x_C|$ is still added to the reward (as in sec. 1), while $\dot{x}_C$ can be inferred by estimating $\dot{\theta}_P$, and since the cartpole is stationary, the network can easily compute the inclination of the pole. So, effectively, this makes learning easier.

- Each sample is a stack of 4 consecutive frames. In this way, the network has the necessary information to estimate velocities and accelerations.

The chosen architecture is taken from the Deepmind paper on Atari games [3], and consists of two convolutional layers (16 filters, kernel 8 and stride 4, followed by 32 filters, kernel 4 and stride 2) and a fully-connected layer with 256 neurons. After each hidden layer a ReLU activation is applied.
A key change from the method of sec. 1 is the use of Dueling Networks [1]. The idea is to follow the convolutional segment with two instances of the fully connected layer. One computes the *advantage* $A(s,a)$ of each action, while the other outputs just a single number, estimating the *value* of the state $V(s)$. The final Q-value is then computed as:

$$Q(s,a) = V(s) + \left[A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')\right]$$

where $|\mathcal{A}|$ is the number of actions that the agent $\mathcal{A}$ can perform. Intuitively, this separates the Q-value into two terms: the goodness of being in state s, and the relative advantage of playing an action a. This allows the model to distinguish between "hopeless" states where any action will change nothing, and "critical" states for which choosing the correct actions matters a lot.
Other than this change, all the considerations from sec. 1 (e.g. replay memory, exploration profile...) still apply.

2.2 Results

The model is trained with the Adam optimizer, and a learning rate of $\lambda = 10^{-4}$, reduced to 0.5×10^{-4}, 10^{-5}, 10^{-6} or 10^{-7} whenever the score surpasses 100, 200, 300 or 400 respectively.

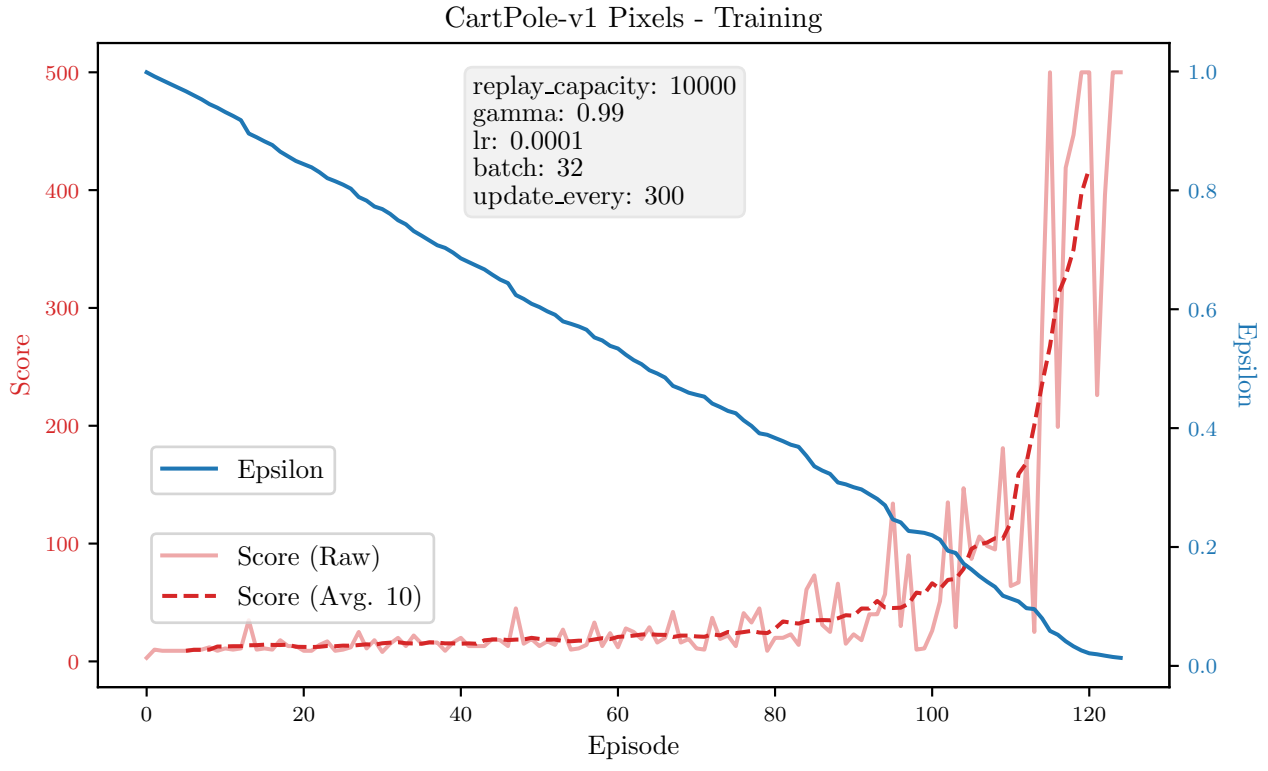

Figure (2) – Learning curve for CartPole with Pixels. ϵ is updated at every frame, while the target policy network is updated every 300 frames.

An ϵ-greedy policy is used for exploration, with ϵ starting from 1 and decaying exponentially to 0.01 with a rate of 1/2500. Differently from sec. 1, ϵ is updated at every frame, and not after each episode. The target network is synced every 300 frames, and a batch size of 32 is used.

The learning curve is shown in fig. 2. The model does not completely solve the environment, but still reaches a 475 mean score over 10 trials. Video renderings showing its behavior are available at the following link, along with the network's inputs (files with `_cnn`).

The main problem seems to be the lack of information about the x_C position, meaning that the cart tends to leave the screen (but maintains the pole upright). However, enforcing the penalty for the cart leaving the center halts the convergence. Similarly, cropping a fixed region that is not centered on the cartpole makes learning extremely difficult. Another issue is that the limited resolution of the rendering means that similar coordinates correspond to the same configuration of pixels, and so the model has access to much less information than that of sec. 1.

3 Pong with Pixels

3.1 Methods

Pong is perhaps the simplest Atari game that can be solved with a DQN. The chosen architecture is the same of sec. 2, which has been proven to work on Atari games in [3].
Screen images are used as inputs. They are first converted to grayscale, and then resized to 84×84. One in every 4 frames is actually shown to the network, with the previous chosen action repeated for the missing frames. This is because Pong games are much longer than CartPole episodes, and do not require frame-perfect moves, meaning that skipping frames is a valid strategy for reducing the computational demand of training.
Finally, each episode is terminated when the agent loses a single life, as done in [4]. This reduces the amount of frames in each episode, while also priming the agent to avoid losing lives.

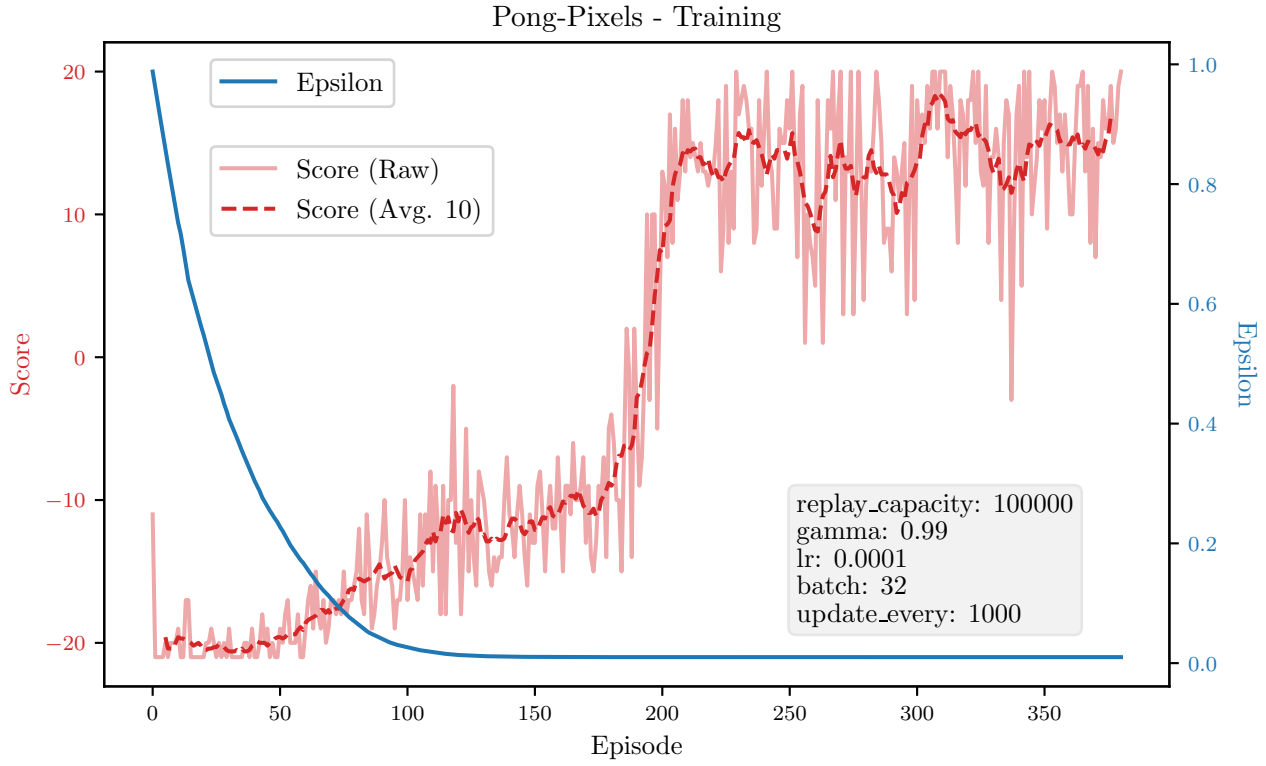

Figure (3) – Learning curve for Pong with Pixels. ϵ is updated at every frame, while the target policy network is updated every 1000 frames.

3.2 Results

The final model is trained with Adam and a fixed learning rate of $\lambda = 10^{-4}$, a batch size of 32, and an ϵ-greedy policy for the agent, with ϵ decreasing exponentially from 1 to 0.01 at a rate of $1/30\,000$. As in sec 2, ϵ is updated at every frame. The target network is synced every 1000 frames.

The agent starts to beat the baseline opponent after $\sim$ 200 episodes. By episode 300, it achieves superhuman skill, and at the end scores an average of 20 over 10 trials (the maximum score in Pong is 21). In particular, it learns how to reset the game in a specific way, which usually involves losing a life at the very start. Then, it can enter a cycle of moves which guarantees a win against the deterministic opponent. A video rendering of such behavior can be found at the following link.

References

[1] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. Apr. 5, 2016. arXiv: 1511.06581 [cs]. URL: http://arxiv.org/abs/1511.06581.

[2] Greg Brockman et al. *OpenAI Gym*. cite arxiv:1606.01540. 2016. URL: http://arxiv.org/abs/1606.01540.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. arXiv: 1312.5602 [cs]. URL: http://arxiv.org/abs/1312.5602.

[4] Volodymyr Mnih et al. "Human-Level Control through Deep Reinforcement Learning". *Nature* 518.7540 (7540 Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (visited on 06/26/2021).