RL - Reinforcement Learning

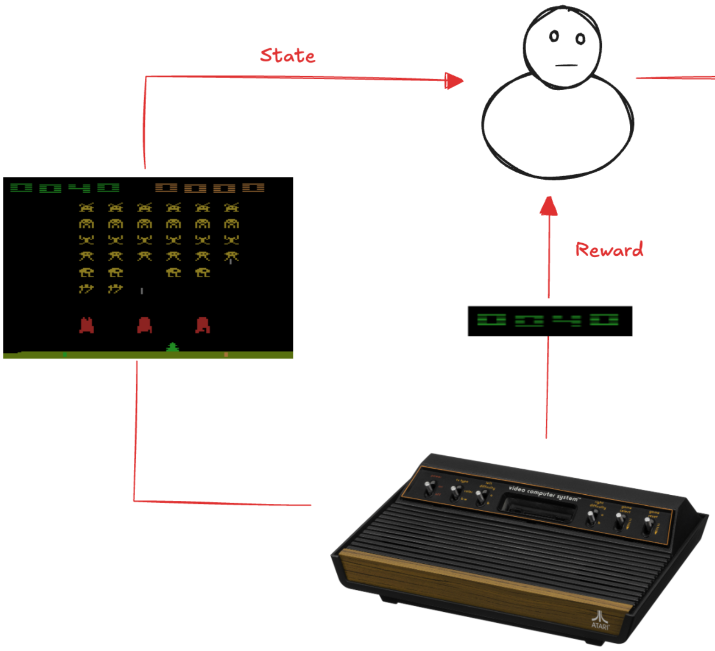
Solutions on Deep Q-Networks (DQN)

Prof. Riccardo Berta

2025.10.03

0.1 Atari Games

Atari 2600 is a video game console which provides several popular games such as pong, space invaders, pacman, break out, centipede and many more. Gymnasium includes ALE-py, the **Arcade Learning Environment (ALE)** which is a simple framework that allows researchers to develop AI agents for Atari 2600 games. It is built on top of the **Stella emulator** and separates the details of emulation from agent design. The main Gymnasium page provides a list of all the Atari environments. The objective of this notebook is to **train an agent to play a game of the Atari console** using the DQN algorithm **learning from raw pixel data**, like in the famous paper V. Mnih, et al. **"Playing Atari with Deep Reinforcement Learning"**, NIPS 2013:



In particular, we can consider **the classic Pong game**. The game is played like tennis. Using a controller, each player hits the ball by moving vertically the paddles on the game field. A player scores one point when the opponent misses a hit. The first player who scores 21 points wins the game. The action space is discrete and deterministic with 6 actions (not all meaningful for Pong): 0 (noop), 1 (fire), 2 (right) and 3 (left), 4 (right fire), 5 (left fire) The observation space is the screen image (210x160x3). The state space is really high-dimensional and to use it you need to modify the neural network architecture adding some convolutional layer in order to extract features from the image.

1 - Define a variable with the name of the Atari game you want to solve (e.g. Pong) and create an instance of the environment.

```
# YOUR CODE HERE

# Import Gymnasium and create an instance with "render_mode" set to "rgb_array". You
# need to import also "ale_py" plugin and to register it with Gymnasium
 ↪  (gym.register_envs(ale_py)).
# Set also the parameter "frameskip" to 1. Later, we will see that we want to control
# the number of frames to skip as a parameter of our code.

import gymnasium as gym
import ale_py
gym.register_envs(ale_py)

pong = gym.make('ALE/Pong-v5', render_mode="rgb_array", frameskip=1)
```

```
A.L.E: Arcade Learning Environment (version 0.10.1+6a7e0ae)
[Powered by Stella]
```

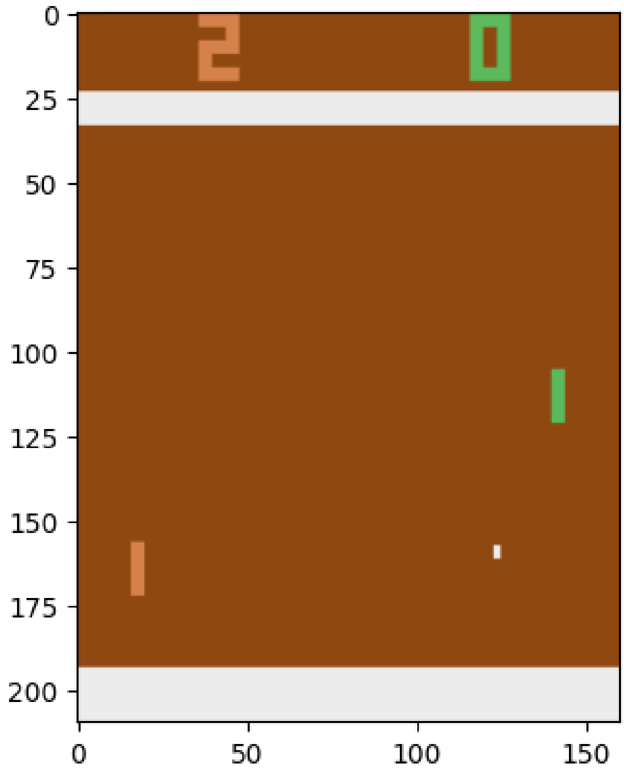
2 - Create a function to show the game in action by rendering the environment and selecting actions at random or using a policy.

```
# YOUR CODE HERE

# Define a function "random_pi" that takes as input a state and a Q-function and
# returns a random action.

import numpy as np

def random_pi(state, q):
    return np.random.randint(6);
```

```
# YOUR CODE HERE

# Define a function "show_policy" that takes as input an environment,
# a policy and a Q-function and shows the policy in action for a number of episodes

import matplotlib.pyplot as plt
from IPython.display import clear_output

def show_policy(env, pi, q, n_episodes=1, max_steps=500):
    for _ in range(n_episodes):
        state = env.reset()[0];
        done = False;
        step = 0;
        while not done:
            action = pi(state, q);
            state, reward, terminated, truncated, info = env.step(action);
            if(terminated or truncated): done = True;
            plt.imshow(env.render());
            plt.show();
            clear_output(wait=True);
            if step > max_steps: break;
            step += 1;
```

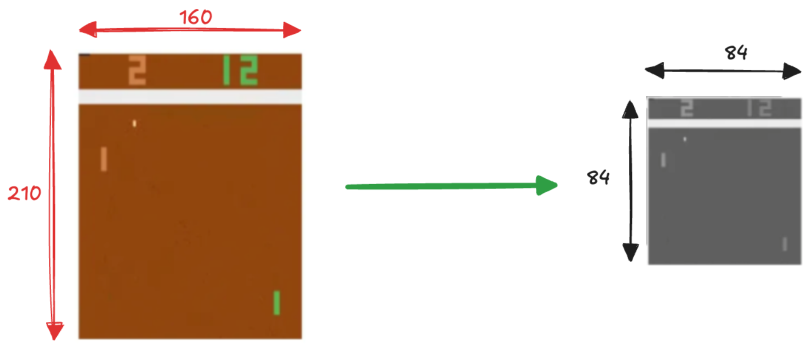
3 - Use the previous function in order to see one episode of the game with a random policy.

```
# YOUR CODE HERE

# Just call the "show_policy" function defined above passing the environment, None as
 ↪  agent and
# the number of episodes you want to visualize

show_policy(pong, random_pi, None, n_episodes=1, max_steps=500);
```



4 - We would like to use as state the raw pixels data. However, **directly feeding the raw game screen image is not efficient**, since it will be **computationally expensive**. So we need to pre-process the environment output to reduce the dimensionality of the problem. Moreover the Atari environments have some game specific issues, which are not relevant for the learning process, that we need to manage. In order to do that, you can use the **wrapper function**, which is a convenient way to modify an existing environment without having to alter the underlying code directly. Using wrappers will allow you to avoid a lot of boilerplate code and make our environment more modular. Here a list of the useful modifications that we can apply to the Atari environment:

1) The raw game screen size is 210x160 pixels with three color channels (RGB), however we don't need the color information and we can resize the image in order to reduce the dimensionality of the problem.



2) Atari games typically run at 60 frames per second, however not every frame contains crucial information for decision-making. To reduce computation and memory requirements, and to speed up training, we can skip a certain number of frames. Moreover, some games have a flickering effect (when the game draws different portions of the screen on even and odd frames) due to the old Atari platform's limitation. For the human eye, such quick changes are not visible, but they can confuse a neural network. So, we can take the maximum of every pixel in the last two frames and use it as observation.

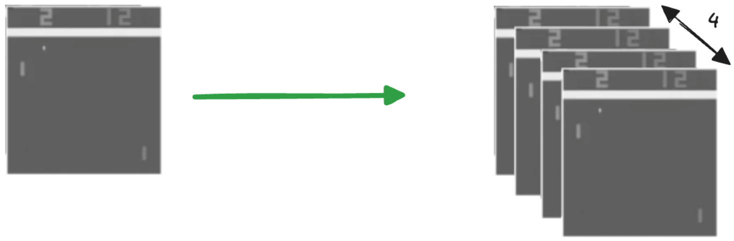
3) In order to inject variability into the initial state of each episode, promoting exploration and helping the agent discover a more robust policy across different starting conditions, we can introduce a certain number of ”no-operation” actions at the beginning of each episode in order to add stochasticity to the initial conditions of the environment.

Add a wrapper in order to apply all those modifications to the environment.

```
# YOUR CODE HERE

# Gymnasium provides a wrapper to apply those pre-porcessing to the
# observations for the Atari games, you can use the "AtariPreprocessing" wrapper
# from "gymnasium.wrappers" module.

pong = gym.wrappers.AtariPreprocessing(pong,
                                      noop_max = 30,          # the max number no-ops
 ↪   actions are taken at reset
                                       frame_skip = 4,         # the number of frames
 ↪   between new observation
                                     screen_size = 84,        # resize the image frame
                                       grayscale_obs = True,   # convert the image
 ↪   frame to grayscale
                                      scale_obs = True)       # scale the image frame
```

5 - A single raw screenshot of the game used as state is not enough to extract the information

about the game state. How can the agent learn to play just seeing a single image at a time? It is **not possible to build an idea about how the game objects are moving**. In Pong, just looking at a single frame don't provide enough information to understand if the ball is moving left to right or right to left. For example, in the Cart-pole example, the state is composed by the position of the cart, the angle of the pole, but it includes also the velocity of both. To deal with this, we need to consider as state a series of frame and not just one. In this way, the agent can learn how the objects are moving. So, we need to apply a wrapper that will stack the last N (e.g. 4) frames together and use them as state.

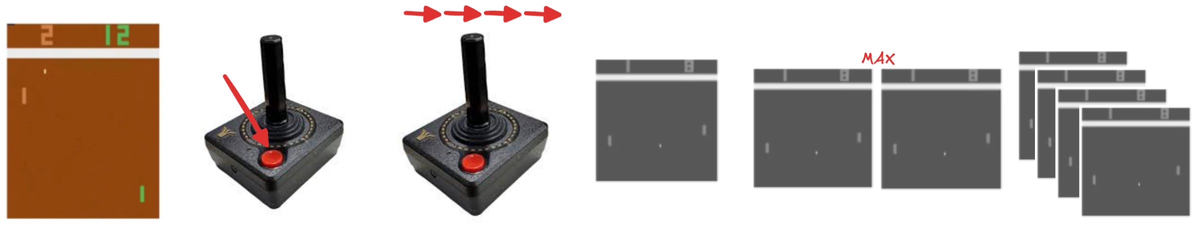


```
# YOUR CODE HERE

# Gymnasium provides the "FrameStackObservation" wrapper to stack the last n frames
 ↪  of the game. Apply it to the
# environment.

pong = gym.wrappers.FrameStack(pong, 4);
```

6 - Some games, as Pong, require a user to press the FIRE button to start the game, we need to take this action on reset. Add another wrapper to take this action on reset.

```
# YOUR CODE HERE

# This wrapper che be found in th stable_baselines3 library. It is called
 ↪  "FireResetEnv"
# Import it and apply it to the environment.

import stable_baselines3

pong = stable_baselines3.common.atari_wrappers.FireResetEnv(pong);
```

7 - Finally, we write a "make_atari_env()" function that takes as input the name of the environment and returns the wrapped environment



1 - Original Environment

2 - Press FIRE to start the game (FireResetEnv wrapper)

3 - Repeat every action for 4 frames (frame_skip=4)

4 - Resize image and convert it to grayscale

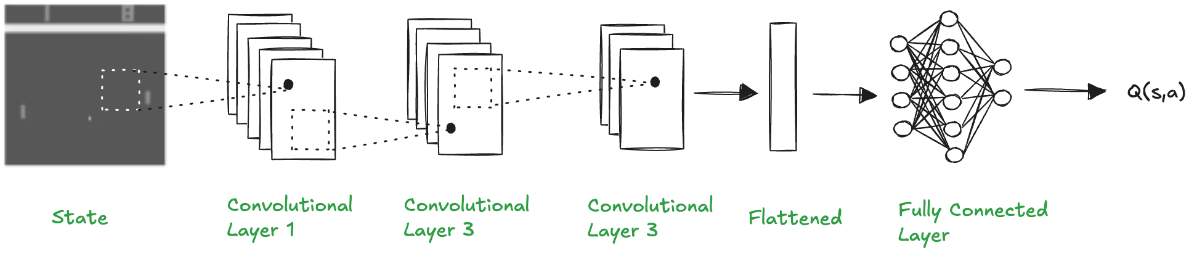
5 - Get the max of two consecutive frames to mitigate flickering

6 - Stack four frames as one observation (FrameStack wrapper)

```python
# YOUR CODE HERE

# You create an environment and then apply in order all the wrappers needed

def make_atari_env(env_name, mode="rgb_array", seed=None):

    # create the environment
    env = gym.make(env_name, render_mode=mode, frameskip=1);

    if seed is not None:
        env.np_random = np.random.Generator(np.random.PCG64(seed));

    # apply the fire reset wrapper
    env = stable_baselines3.common.atari_wrappers.FireResetEnv(env);

    # apply a wrapper to skip frames, scale the observation and convert it to grayscale
    env = gym.wrappers.AtariPreprocessing(env, noop_max=30, frame_skip=4, screen_size=84, grayscale_obs=True, scale_obs=True);

    # apply a wrapper to stack the frames
    env = gym.wrappers.FrameStack(env, 4);

    # return the environment
    return env;

# set the seed for reproducibility
seed = 42;

# set the name of the Atari environment
env_name = 'ALE/Pong-v5';

# create the environment
pong = make_atari_env(env_name, seed=seed);
```

8 - Since we are dealing with images, **instead of using a fully connected neural network, this time we need to use a convolutional neural network**. The convolutional layers extract the features from the image and output the feature maps, then the flattened feature map can be fed to the fully connected layer which returns the Q value. You can implement the following architecture:

- Convolutional layer with 32 filters, kernel size 8, stride 4, padding 0, followed by a ReLU activation function
- Convolutional layer with 64 filters, kernel size 4, stride 2, padding 0, followed by a ReLU activation function
- Convolutional layer with 64 filters, kernel size 3, stride 1, padding 0, followed by a ReLU activation function
- Flatten layer
- Fully connected layer with 512 units, followed by a ReLU activation function
- Fully connected layer with a number of units equal to the number of actions



```python
# YOUR CODE HERE

# Add the code for th neural network using the PyTorch library. And
# create two networks, one for the online network and one for the target network.

import torch

# set the seed for reproducibility
torch.manual_seed(seed);

# make device available
device = None;

# set the backend device to MPS, if available
if torch.backends.mps.is_available():
    device = torch.device("mps");
else:
    device = torch.device("cpu");

# print the used device
print(f"Using device: {device}");

def create_network(input_size, output_size):

    dnn = torch.nn.Sequential(
        #conlutional layers
        torch.nn.Conv2d(input_size[0], 32, kernel_size=8, stride=4),
        torch.nn.ReLU(),
        torch.nn.Conv2d(32, 64, kernel_size=4, stride=2),
        torch.nn.ReLU(),
        torch.nn.Conv2d(64, 64, kernel_size=3, stride=1),
        torch.nn.ReLU(),

        # flatten the output
        torch.nn.Flatten(),

        # fully connected layers
        torch.nn.Linear(7 * 7 * 64, 512),
        torch.nn.ReLU(),
        torch.nn.Linear(512, output_size));

    return dnn.to(device);

def set_optimizer(model, learning_rate):
    optimizer = torch.optim.RMSprop(model.parameters(),
                                    lr=learning_rate,   # Learning rate
                                    momentum=0.95,      # Momentum
                                 eps=0.01,          # Epsilon for numerical stability
                                   alpha=0.99);        # Smoothing constant (default: 0.99))
    return optimizer;


def update_target(target_model, online_model):
    for target, online in zip(target_model.parameters(), online_model.parameters()):
        target.data.copy_(online.data);

def pi(state, online_q):
    # convert the state into a tensor
    state = np.array(state);
    state = torch.from_numpy(state).unsqueeze(0).to(device);

    # calculate q_values from the network
    q_values = online_q(state).detach();

    # act greedy
    action = np.argmax(q_values.cpu()).data.numpy();

    return action;

state_size = pong.observation_space.shape;
action_size = pong.action_space.n;

dnn_online = create_network(state_size, action_size);
dnn_target = create_network(state_size, action_size);

learning_rate = 0.00025;
optimizer_online = torch.optim.RMSprop(dnn_online.parameters(), lr=learning_rate);
```

```
Using device: mps
```

9 - Add the Replay Memory to store the transitions (state, action, reward, next_state, done).

```
# YOUR CODE HERE

# Add the code for the replay buffer already used presented in the DQN implementation.
#  We need a larger replay buffer because the problem is more complex.

replay_memory_size = 100000;

# define the experience type
experience_type = np.dtype([
    ('state', np.float32, (state_size)),
    ('action', np.int32),
    ('reward', np.float32),
    ('next_state', np.float32, (state_size)),
    ('failure', np.int8)
])

# create the replay memory
replay_memory = {
    'size': replay_memory_size,
    'buffer': np.empty(shape=(replay_memory_size,), dtype=experience_type),
    'index': 0,
    'entries': 0
}

def store_experience(replay_memory, experience):
    # store the experience in the buffer
    replay_memory['buffer'][replay_memory['index']] = experience;

    # update the number of experiences in the buffer
    replay_memory['entries'] = min(replay_memory['entries'] + 1,
 ↪  replay_memory['size']);

    # update index, if the memory is full, start from the begging
    replay_memory['index'] += 1;
    replay_memory['index'] = replay_memory['index'] % replay_memory['size'];

def sample_experiences(replay_memory, batch_size):

    # select uniformly at random a batch of experiences from the memory
    idxs = np.random.choice(range(replay_memory['entries']), batch_size,
 ↪  replace=False);

    # return the batch of experiences
    experiences = replay_memory['buffer'][idxs];

    return experiences;
```

10 - Create a the code for exploration (decaying epsilon-greedy policy) and exploitation (greedy policy) and the code for doing the evaluation of the agent.

```
# YOUR CODE HERE

# Add the code for decaying epsilon and the epsilon-greedy policy as already
# done for the DQN implementation

import random

def decay_epsilon(max, min, decay_steps):
    values = np.logspace(start=0, stop=-2, num=decay_steps, base=10);
    values = (values - values.min()) / (values.max() - values.min());
    values = (max - min) * values + min;
    return values;

def epsilon_greedy(dnn, state, action_size, epsilon):
    if random.uniform(0,1) < epsilon:
        return np.random.randint(action_size);
    else:
        # convert the state into a tensor
        state = np.array(state);
        state = torch.from_numpy(state).unsqueeze(0).to(device);
        # calculate q_values from the network
        q_values = dnn(state).detach();
        # act greedy
        action = np.argmax(q_values.cpu()).data.numpy();
        # return the action
        return action;
```

```
# YOUR CODE HERE

# Add the code for an evaluation or the agent as already
# done for the DQN implementation.

def evaluate(env, pi, q, episodes=1):
    rewards = [];

    for episode in range(episodes):
        state, done = env.reset()[0], False;
        rewards.append(0);
        while not done:
            action = pi(state, q);
            state, reward, terminal, truncated, info = env.step(action)
            rewards[-1] += reward;
            done = terminal or truncated;

    return np.mean(rewards);
```

11 - Add the code for the optimization of the Q-function based on the experience sampled from the replay buffer. Remember to use the target network to compute the target Q-values.

```
# YOUR CODE HERE
```

```python
# Add the code of the optimization algorithm as already
# done for the DQN implementation. This time you need to to pay attention
# to states which are a stack of four images

def optimize(replay_memory, batch_size, online_model, target_model, optimizer, gamma):

    # sample a batch of experiences
    batch = sample_experiences(replay_memory, batch_size);

    # prepare the experience as tensors
    states = torch.from_numpy(batch['state'].copy()).float().to(device);
    actions = torch.from_numpy(batch['action'].copy()).long().to(device);
    next_states = torch.from_numpy(batch['next_state'].copy()).float().to(device);
    rewards = torch.from_numpy(batch['reward'].copy()).float().to(device);
    failures = torch.from_numpy(batch['failure'].copy()).long().to(device);

    # get the values of the Q-function at next state from the "target" network
    # remember to detach, we need to treat these values as constants
    q_target_next = target_model(next_states).detach();

    # get the max value
    max_q_target_next = q_target_next.max(1)[0];

    # one important step, often overlooked, is to ensure
    # that failure states are grounded to zero
    max_q_target_next *= (1 - failures.float())

    # calculate the target
    target = rewards + gamma * max_q_target_next;

    # finally, we get the current estimate of Q(s,a)
    # here we query the current "online" network
    q_online_current = torch.gather(online_model(states), 1, actions.unsqueeze(1)).squeeze(1);

    # create the errors
    td_error = target - q_online_current;

    # calculate the loss
    loss = td_error.pow(2).mean();

    # backward pass: compute the gradients
    optimizer.zero_grad();
    loss.backward();

    # update model parameters
    optimizer.step();

    return loss.detach().cpu().numpy();
```

12 - Write the code of the DQN algorithm (exploiting the new convolutional network architecture). You should modify the code in order to save the network weights after some episodes.

```
# YOUR CODE HERE

# Add the code for the DQN algorithm, exploiting the convolutional neural architecture
# already defined, the replay buffer as already implemented in DQN algorithm.
# At the begging, if a model file is already present on the disk, you should load the weights and
# sometimes, you should save the model on the disk.

def dqn(env, online_model, target_model, optimizer,
        replay_memory, batch_size, target_update_steps,
        exploration_strategy, epsilon_max, epsilon_min, epsilon_decay_steps,
        gamma, max_episodes):

    # create a score tracker for statistic purposes
    scores = [];

    # create a list of epsilon values
    epsilons = decay_epsilon(epsilon_max, epsilon_min, epsilon_decay_steps);

    # counter for the number of steps
    step = 0;

    # save the best score and episode obtained so far
    best_score = -1000;
    best_episode = -1000;

    # update the target model with the online one
    update_target(target_model, online_model);

    # train until for the maximum number of episodes
    for episode in range(max_episodes):
```

```
    # reset the environment before starting the episode
    state, done = env.reset()[0], False;

    # interact with the environment until the episode is done
    while not done:

        # calculate the epsilon value
        epsilon = epsilons[step] if step < epsilon_decay_steps else epsilon_min;

        # select the action using the exploration policy
        action = exploration_strategy(online_model, state, action_size, epsilon);

        # perform the selected action
        next_state, reward, terminal, truncated, info = env.step(action);
        done = terminal or truncated;
        failure = terminal and not truncated;

        # store the experience into the replay buffer
        experience = (state, action, reward, next_state, failure);
        store_experience(replay_memory, experience);

        # optimize the online model after the replay buffer is large enough
        if replay_memory['entries'] > batch_size:

            # optimize the online model
            optimize(replay_memory, batch_size,  online_model, target_model,
↪  optimizer, gamma);

        # sometimes, synchronize the target model with the online model
        if step % target_update_steps == 0:
            update_target(target_model, online_model);

        # update current state to next state
        state = next_state;

        # update the step counter
        step += 1;

    # After each episode, evaluate the policy
    score = evaluate(env, pi, online_model, episodes=1);

    # store the best score and episode
```

```
        if(score > best_score):
            best_score = score;
            best_episode = episode;

        # store the score in the tracker
        scores.append(score);

        # print some informative logging
        message = 'Episode {:03}, steps {:04}, current score {:05.1f}, best score
 ↪  {:05.1f} at episode {:03}';
        message = message.format(episode+1, step, score, best_score, best_episode);
        print(message, end='\r', flush=True);

    return scores;
```

14 - Apply the algorithm to the ATARI environment. Pay attention, the following code takes a lot of time to run (days on a powerful machine).

```
# YOUR CODE HERE

# Apply the previous written algorithm to the ATARI environment, please pay attention
# to the parameters.

max_episodes = 500;
batch_size = 64;
gamma = 0.99;
target_update_steps = 10;
epsilon_max = 1.0;
epsilon_min = 0.01;
epsilon_decay_steps = 500000;

# run the algorithm
scores = dqn(pong, dnn_online, dnn_target, optimizer_online,
             replay_memory, batch_size, target_update_steps,
             epsilon_greedy, epsilon_max, epsilon_min, epsilon_decay_steps,
             gamma, max_episodes);
```

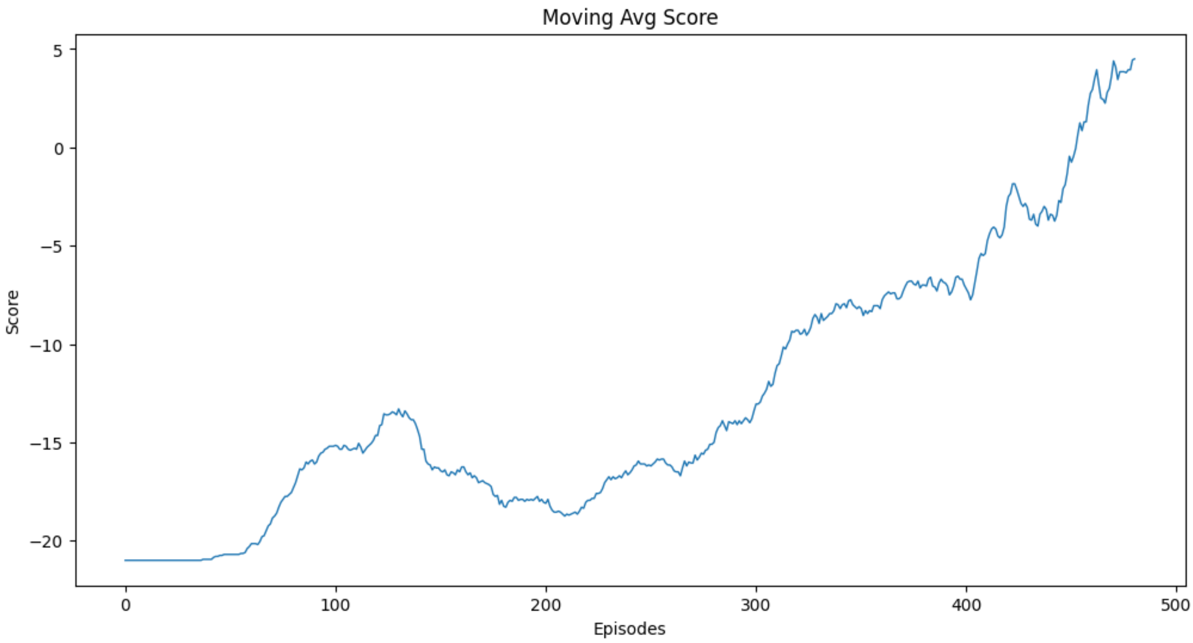
```
Episode 500, steps 983859, current score 012.0, best score 018.0 at episode 488
```

15 - Smooth the result using a sliding window

```
# YOUR CODE HERE

# smooth the result using a sliding window
sliding_windows = 20
scores = np.convolve(scores, np.ones(sliding_windows)/sliding_windows, mode='valid');
```

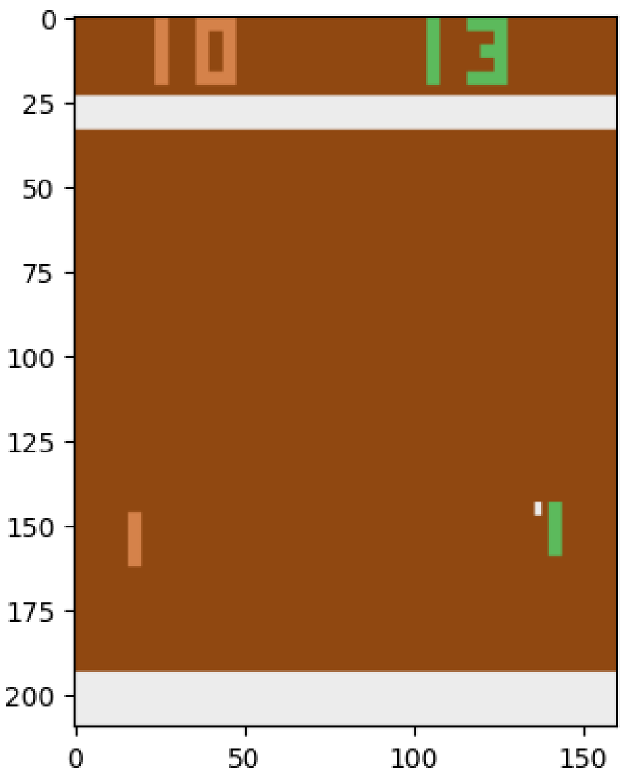
16 - Show the score curve as a function of the number of episodes

```
# YOUR CODE HERE

# Show the moving average reward during the training process

import matplotlib.pyplot as plt

plt.figure(figsize=(12,6))

plt.plot(scores, linewidth=1)
plt.title('Moving Avg Score')
plt.ylabel('Score')
plt.xlabel('Episodes')

plt.show()
```



18 - Show the agent playing the game using the greedy policy

```
# YOUR CODE HERE

# You can show the policy learned by the agent using the "show_policy" function

show_policy(pong, pi, dnn_online, n_episodes=1, max_steps=2000);
```


10
13
0
25
50
75
100
125
150
175
200
0
50
100
150