UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Developing An AI-Powered Agent For Playing Super Mario Bros

Winter semester 2024/2025

Submission date: 19.12.2024

Marius Zimmerhackl

mnzimmer@teleco.upv.es

Study subject: Telecommunications

ID: 76098390

Teaching staff: Jorge Igual García, Vicent Pla Boscà

I hereby confirm that I have prepared this report independently. All resources used have been indicated.

Contents

1 Introduction

The following report deals with the functionality of a code for the development of an AI agent that is able to play the game "Super Mario Bros" independently. "Super Mario Bros" is a jump-and-run video game in which the player controls a character named Mario. The goal is to help Mario complete the 32 stages by making him run, jump, and duck. There are various obstacles such as cliffs and moving enemies that make solving levels difficult. The game can be played online at [Mar]. Since the levels are problems whose solutions are not yet known, reinforcement learning was used to solve them.

The code is mainly based on [Fen21] and [Kun23]. It is completely written in Python and uses Pytorch as a machine learning library. Please note that not every line of code is covered in the following. Both projects have the same basic structure and consist of several files, of which only those essential for the general understanding of the project will be discussed.

The explanation of the code is divided into four chapters. First, the creation of the environment is explained in the 2 chapter. Second, Deep Q-Learning, the reinforcement learning algorithm used in this project, is briefly explained in chapter 3. After that, chapter 4 deals with the implementation of an agent that allows Mario to act, remember, and learn the game. Finally, chapter 5 shows the main file that puts it all together.

2 Environment

2.1 Creating the environment

To set up the environment, `gym-super-mario-bros` is installed. It is based on OpenAI's Gym library, which is a widely open collection of reinforcement learning environments. The most common approach today is Gymnasium, which is a maintained fork of OpenAI's Gym library maintained by an outside team.
It allows the user to access the game and returns objects for the states along with a simple way to enter actions. The environment iterates through the stages of the game. If the agent fails (Mario dies) while solving a stage, it has to start again at the beginning of the stage. If it fails three times, it must start again from the beginning of the first stage. Only rewardable gameplay frames are sent to the agent, which means no cutscenes or loading scenes. [Kau24a]

It is also possible to import just one stage of the game and have the agent play it from the beginning. This will be used in the following project to easily understand the basic concepts of the code.
When the game was released for the Nintendo Entertainment System (NES) in 1987, players could players could issue commands using a NES controller, as shown in the figure 1. The possible commands, or actions, that Mario could perform were walking (directional cross), jumping (A Button), and shooting (B Button) [Wik24]. In this project, the pool of possible actions the agent can choose from is the same as what a human could do with the controller. Therefore, `nes_py` is imported, which provides a virtual joypad for the agent. This is also a library based on OpenAI's Gym.[Kau24b].

Figure 1: NES-controller from 1987. [Wik24]

The following code gives an example of how to set up the game and play a trial of a stage where actions are chosen randomly.

```
import gym_super_mario_bros
from gym_super_mario_bros.actions import RIGHT_ONLY

from nes_py.wrappers import JoypadSpace

ENV_NAME = 'SuperMarioBros-1-1-v0'

env = gym_super_mario_bros.make(ENV_NAME, render_mode='human',
apply_api_compatibility=True)
env = JoypadSpace(env, RIGHT_ONLY)

```

```
12 done = False
13 env.reset()
14 counter = 0
15 while not done:
16     # Choose random action
17     action = env.action_space.sample()
18
19     _, _, done, _, _ = env.step(action)
20     env.render()
```

Lines 1-4 import the libraries. Line 6 selects the environment. `'SuperMarioBros-1-1-v0'` will import only stage 1-1 of the basic Super Mario Bros. game without any chance of the ROM [Kau24a]. In line 8, the environment is created with the function `gym_super_mario_bros.make()`. Note that you need to select `render mode= 'human'` to be able to watch the game while Python is running. Line 10 shows the use of the `nes_py.wrappers JoypadSpace`. This limits the possible actions to `RIGHT_ONLY`, which means that Mario can only choose between the five actions `[['NOOP'], ['right'], ['right', 'A'],['right', 'B'],['right', 'A', 'B']]`. The boolean `done` is false while the agent is trying to solve the stage. It becomes true when the stage is solved or when Mario dies. The `env.reset()` function resets the environment to its initial state. After that, the while loop runs the game. The function `action = env.action_space.sample()` randomly chooses one of the possible actions. After each action, the function `env.step(action)` updates the parameters `next_state, reward, done, trunc, info` depending on the previously chosen action. The variable `next_state` contains the current state in the form of a tensor, which is explained in more detail in the chapter 2.2.

In `reward` the reward for the chosen action in the current state is given. It increases when Mario's speed is positive, i.e. when he moves to the right and gets closer to the flag (end of stage). It decreases for every timestep that passes or if Mario dies. `trunc` indicates if the game ended naturally (Mario cleared the level, Mario died, or Mario ran out of time) or if the game ended unnaturally (e.g. the user quit the game). The variable `info` contains various information such as coins, score, lives and position. If you need more precise information about the exact determination, you can get it from [Kau24a], but for the understanding of the project this is sufficient. To see the current state, `env.render()` renders the current frame [Kau24a; Kau18].

2.2 Preprocessing

The environment returns the state of the game (`next_state`) in a tensor of size [3,240,256], which represents one frame of the game screen. This means that we have 240x256 pixels with three values each for the amount of red, green, and blue. This may be the ideal format for humans to play the game, but not for a reinforcement learning agent. Unnecessary information leads to longer runtime during training. Therefore, the state is preprocessed before it is evaluated by our agent. The following wrappers are applied to the environment:

- GrayScaleObservation: Transforms the RGB-values into grayscalings. This reduces the tensor to [1,240,256] without any loss of (for the agent) usefull data.
- ResizeObservation: Downsamples each frame into an square-sized image. This leads to a new size of [1,84,84].
- SkipFrame: Skips a variable number of frames because consecutive frames dont varry much. This leads to an way faster processing without much loss of information.
- FrameStack: Allows us to squash m consecutive frames of the environment into a single observation point to feed to our learning model, which leads to an [m,84,84] sized tensor. This way, we can identify if Mario was landing or jumping based on the direction of his movement in the previous several frames. This is necessary for the in 2.1 mentioned reward calculation based on the velocity.

The following code shows the implementation of the wrappers.
Note that `SkipFrame` is a custom wrapper that inherits from `gym.Wrapper` and implements the step() method.

```
import numpy as np
from gym import Wrapper
from gym.wrappers import GrayScaleObservation, ResizeObservation,
FrameStack

class SkipFrame(Wrapper):
    def __init__(self, env, skip):
        super().__init__(env)
        self.skip = skip

    def step(self, action):
```

```
        total_reward = 0.0
        done = False
        for _ in range(self.skip):
            next_state, reward, done, trunc,
            info = self.env.step(action)

            total_reward += reward
            if done:
                break
        return next_state, total_reward, done, trunc,
               info

def apply_wrappers(env):
    env = SkipFrame(env, skip=4)
    env = ResizeObservation(env, shape=84)
    env = GrayScaleObservation(env)
    env = FrameStack(env, num_stack=4, lz4_compress=True)
    return env
```

The method **apply_wrappers(env)** applies all described wrappers to the environment by cascading them. The figure 2 shows the difference between the original environment and the environment after applying the method.

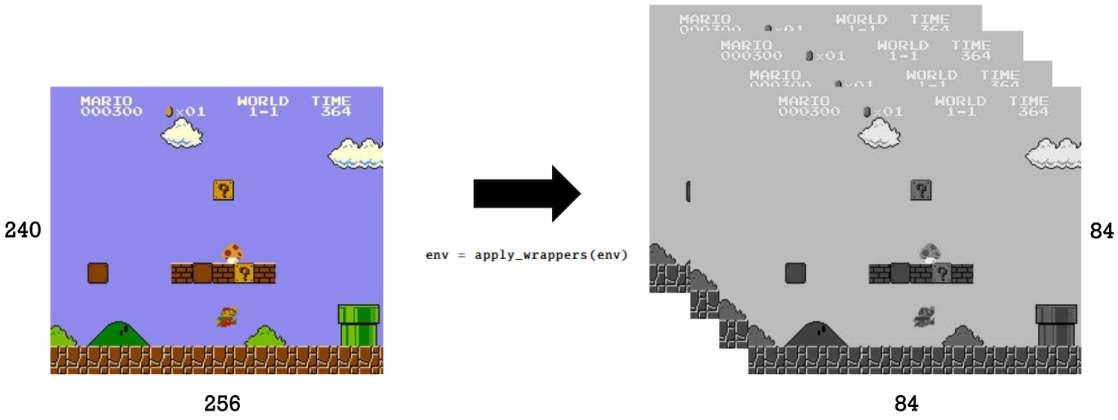

Figure 2: Impact on the state-tensor after applying the wrapper function on the environment.

3 Deep Double Q-Learning

Both projects [Fen21] and [Kun23] use Deep Double Q-Learning (DDQL) as a reinforcement learning algorithm. This choice is primarily because of two reasons, which are briefly explained below.

For comparison, the update function for the predicted value $Q(S_t, A_t)$ of state S and action A at time t in the standard Q-Learning (QL) algorithm is given in Equation 1 [SG20]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[Y_t^{QL} - Q(S_t, A_t) \right] \tag{1}$$

Here, α represents the learning rate, γ is the discount factor, and Y_t^{QL} is the target value, calculated as shown in Equation 2:

$$Y_t^{QL} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \tag{2}$$

As the name suggests, DDQL differs from standard QL in two significant aspects:
1. Function Approximation with Neural Networks: DDQL employs a multi-layer neural network, a powerful tool for approximating functions in high-dimensional spaces. This is essential because the Q-values form a large space of state-action pairs. When using a neural network, the Q-values are no longer approximated but the weight parameters of the network θ. The corresponding update function is given in equation 3.

$$\theta_{t+1} = \theta_t + \alpha \left[Y_t^{DDQL} - Q(S_t, A_t; \theta_t) \right] \nabla_{\theta_t} Q(S_t, A_t; \theta_t) \tag{3}$$

2. Double Q-Learning for Reducing Overestimations: The key idea of Double Q-Learning is to decrease overestimations by separating the max operation in the target computation into two steps: action selection and action evaluation. In the context of neural networks, this involves using two networks:

- Online Network: Evaluates the policy.
- Target Network: Estimates the values of the online network. The target network is periodically updated with the weights of the online network.

This leads to the following modification in the calculation of the target value:

$$Y_t^{DDQL} = R_{t+1} + \gamma Q\big(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t); \theta_t^-\big) \tag{4}$$

Here, θ represents the weight parameters of the online network, and θ^- denotes the weight parameters of the target network. The update rule for the Q-values remains unchanged [HGS15].
The implemention of the algorithm happens in the **class** `Agent` which is explained in 4.2.3.

4 Training Mario

4.1 Neural Network

As explained in 3, two identically structured neural networks are needed. Both codes [Fen21] and [Kun23] use a convolutional neural network (CNN). The following code shows the creation of the network structure in **class** `AgentNN`.

```
class AgentNN(nn.Module):
    def __init__(self, input_shape, n_actions, freeze=False):
        super().__init__()
        # Conolutional layers
        self.conv_layers = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU() )

        conv_out_size = self._get_conv_out(input_shape)

        # Linear layers
        self.network = nn.Sequential(
            self.conv_layers,
            nn.Flatten(),
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions))

```

```
23          if freeze:
24              self._freeze()
25
26          self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
27          self.to(self.device)
```

The input parameters of the constructor are `input_shape`, `n_actions` and `freeze`. `input_shape` defines the shape of the input data, which is the dimensions of the state. `n_actions` defines the shape of the output data, which is the number of possible actions. After training, the network should return the best action for each state inserted. The `freeze` flag decides whether the network is trained or not, which should be `true` for the target network and `false` for the online network.

The network contains three connected layers and two linear layers. This structure is strongly reminiscent of a network used for image recognition.
The convolutional layers learn local features and combine them to recognize more complex patterns. The stacked convolutional layers and the arrangement of the kernels and steps extract first simple and then increasingly complex features.
`ReLu()` is a nonlinear activation function that sets the negative values to 0.
After the convolutional layers, the output is flattened (with `nn.Flatten`) and `nn.Linear` performs a linear transformation. To configure the first linear layer correctly, we need to know the number of output elements after the convolutional layers. To do this, a dummy tensor (all values 0) with the size `input_shape` is created and sent through the convolutional layers, as shown in the code below.

```
1 class AgentNN(nn.Module):
2     def _get_conv_out(self, shape):
3     o = self.conv_layers(torch.zeros(1, *shape))
4     return int(np.prod(o.size()))
```

4.2 Agent

To create the agent that embodies the self-acting Mario, another class is needed. Mario should be able to **act** according to the optimal policy for the current state. He should also be able to **store and retrieve experience**, which contains the variables `state, action, re`–.

`forward, next_state` and `done`. The most important part, however, is that Mario should be able to **learn**, i.e. improve its action policy over time.

In order to implement the methods for the functionalities just mentioned, we first need to initialize the hyperparameters, two instances of the `AgentNN` class (for online and target network) and a replay buffer. The replay buffer serves as a limited memory for the experience that the agent collects during training. In this case, it stores the state, the selected action, the reward, the next state, and the boolean done. Using a replay buffer has two major advantages. First, the reuse of data increases data efficiency. Second, using sequential data during training can lead to instability. Random sampling from the replay buffer solves this problem. The optimizer and loss functions are also defined. An Adam optimizer and a mean squared error loss (MSELoss) are used.

4.2.1 Act

To choose an action, the agent uses the ϵ greedy technique, i.e. instead of choosing the best action every time, it sometimes chooses a random action instead. The probability of choosing a random action depends on the value of ϵ, which decreases with a higher amount of training sets, as can be seen in 4.2.3. The code to implement the act method is shown below.

```
1  class Agent:
2      def choose_action(self, observation):
3          if np.random.random() < self.epsilon:
4              return np.random.randint(self.num_actions)
5
6          observation = torch.tensor(np.array(observation),
7          dtype=torch.float32) \
8          .unsqueeze(0) \ .to(self.online_network.device)
9
10         return self.online_network(observation).argmax().item()
```

Lines 2-4 implement the ϵ greedy action selection. Line 6 unsqueezes the input state called `overservation`, which adds a dimension to make it fit the batch size. Line 10 then returns the index of the action with the highest Q-value for the input state.

4.2.2 Saving and Retrieving Experience

As mentioned above, a replay buffer is used to store the data. Since this works with tensors, the input values **state, action, reward, next_state** and **done** are converted to tensors and added to the replay buffer in the form of TensorDic.

```
class Agent:
    def store_in_memory(self, state, action, reward,
        next_state, done):
        self.replay_buffer.add(TensorDict({
            "state": torch.tensor(np.array(state),
                     dtype=torch.float32),
            "action": torch.tensor(action),
            "reward": torch.tensor(reward),
            "next_state": torch.tensor(np.array(next_state),
                          dtype=torch.float32),
            "done": torch.tensor(done)
            }, batch_size=[]))
```

4.2.3 Learn

The learning method is used to train the neural network. For this purpose, the deep double Q-learning described in 3 is implemented. Since this method is the "heart" of the code, it will be explained step by step below.

```
class Agent:
    def learn(self):
        if len(self.replay_buffer) < self.batch_size:
            return

        self.sync_networks()

        self.optimizer.zero_grad()

        samples = self.replay_buffer.sample(self.batch_size).
                  to(self.online_network.device)

```

```
13          keys = ("state", "action", "reward", "next_state", "done")
14
15          states, actions, rewards,
16          next_states, dones = [samples[key] for key in keys]
17
18          predicted_q_values = self.online_network(states)
19          predicted_q_values
20          = predicted_q_values[np.arange(self.batch_size),
21            actions.squeeze()]
22
23          target_q_values =
24          self.target_network(next_states).max(dim=1)[0]
25          target_q_values =
26          rewards + self.gamma * target_q_values * (1 - dones.float())
27
28          loss = self.loss(predicted_q_values, target_q_values)
29          loss.backward()
30          self.optimizer.step()
31
32          self.learn_step_counter += 1
33          self.decay_epsilon()
```

Lines 1-3 first check if there is enough data in the replay buffer. The training will only start if `batch_size` many transitions are available.
In line 5, the weights of the online network are copied to the target network, which is done at `sync_network_rate` intervals. The reason for this is explained in 3.
Line 8 sets the gradients of the optimizer to zero to remove old gradient values from previous optimization steps.
In lines 10 to 16, a number of `batch_size` transitions are randomly taken from the replay buffer containing the variables `states, actions, rewards, next_states` and `dones`.
The predicted Q-values are calculated in lines 18 to 21. The online network calculates Q-values for all actions for each state in the batch. The Q-values for the actually selected actions are extracted.
In lines 23 to 26, the target Q-values are determined by the target mesh. For each transition, the maximum Q value for the next action is selected. The equation used for this has already been explained in 3. If the current state is terminal, the reward is set to

0 by (1-done).
The error is determined by the loss function in line 28 using the predicted and target q-values. Then the online net is optimized. To do this, `loss.backward()` calculates the gradients of the weights in the online network based on the loss, and `self.optimizer.step()` updates the weights in the online network based on the gradients.
Finally, the learning counter is increased and the exploration rate is decreased in lines 32 and 33.

5 Summary

Now that all the essential elements of the code have been explained, we can present the main file that puts it all together and trains Mario.

```
1  import gym_super_mario_bros
2  from gym_super_mario_bros.actions import RIGHT_ONLY
3  from nes_py.wrappers import JoypadSpace
4
5  from wrappers import apply_wrappers
6  from agent import Agent
7
8  ENV_NAME = 'SuperMarioBros-1-1-v0'
9  SHOULD_TRAIN = True
10 DISPLAY = True
11 NUM_OF_EPISODES = 50_000
12
13
14 env = gym_super_mario_bros.make(ENV_NAME, render_mode='human' if DISPLAY
15 env = JoypadSpace(env, RIGHT_ONLY)
16
17 env = apply_wrappers(env)
18
19 agent = Agent(input_dims=env.observation_space.shape, num_actions=env.act
20
21 for i in range(NUM_OF_EPISODES):
22     done = False
23     state, _ = env.reset()
```

```
24      while not done:
25          a = agent.choose_action(state)
26          new_state, reward, done, truncated, info  = env.step(a)
27
28          agent.store_in_memory(state, a, reward, new_state, done)
29          agent.learn()
30
31          state = new_state
32
33  env.close()
```

We start by importing all the necessary libraries and classes.
Then the environment is created, which was explained in 2. As already mentioned, the training only covers the level `'SuperMarioBros-1-1-v0'`. The environment is then pre-processed by applying the wrapper, which was also explained in 2.
Then the agent is created, as explained in 3 and 4, and the training loop is started. The number of episodes was previously defined, and an episode lasts until done becomes false, i.e. Mario fails or the time runs out.

6 Literature

References

[HGS15] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG]. URL: https://arxiv.org/abs/1509.06461.

[Kau18] Christian Kauten. *gym-super-mario-bros*. https://github.com/Kautenja/gym-super-mario-bros/blob/master/gym_super_mario_bros/smb_env.py [Accessed: (01.12.2024)]. 2018.

[SG20] Richard S. Sutton and Andrew G.Barto. *Reinforcement Learning*. USA: Westchester Publishing Service, 2020.

[Fen21] Yuansong Feng. *MadMario*. https://github.com/yfeng997/MadMario/tree/master [Accessed: (25.11.2024)]. 2021.

[Kun23] Sourish Kundo. *Super-Mario-Bros-RL*. https://github.com/Sourish07/Super-Mario-Bros-RL/tree/main [Accessed: (1.12.2024)]. 2023.

[Kau24a] Christian Kauten. *gym-super-mario-bros*. https://pypi.org/project/gym-super-mario-bros/ [Accessed: (01.12.2024)]. 2024.

[Kau24b] Christian Kauten. *nes-py*. https://pypi.org/project/nes-py/ [Accessed: (01.12.2024)]. 2024.

[Wik24] Wikipedia. *Super Mario Bros. — Wikipedia, die freie Enzyklopädie*. [Online; Stand 2. Dezember 2024]. 2024. URL: https://de.wikipedia.org/w/index.php?title=Super_Mario_Bros.&oldid=249044376.

[Mar] Mario. *Super Mario Online*. https://supermarioplay.com/de [Accessed: (25.11.2024)].