Training a Reinforcement Learning agent to Play Mario game

Arina Sadeghi Khiabanian   Maxence Murat

Deep Learning 3 Credit Project
Professor: Andrea Asperti

1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is a subset of machine learning where an agent learns to make decisions by performing certain actions and receiving feedback from those actions in the form of rewards or penalties. The goal of the agent is to maximize the cumulative reward over time. Key concepts in RL include:

- **Environment**: The world with which the agent interacts.
- **State**: A representation of the current situation of the environment.
- **Action**: The decisions or moves the agent can make.
- **Reward**: The feedback received from the environment after performing an action.
- **Policy**: A strategy used by the agent to determine the next action based on the current state.
- **Value Function**: A function that estimates the expected reward for a given state or state-action pair.

In RL, the agent iteratively interacts with the environment, updating its policy based on the rewards received, thereby learning the optimal way to achieve its goals.

2 Training a Mario-playing RL Agent

2.1 Project Overview

In this project, we developed an AI agent capable of playing the classic game Super Mario Bros. using Reinforcement Learning. The agent was trained using the Double Deep Q-Networks (DDQN) algorithm, which is an extension of the Deep Q-Networks (DQN) that mitigates the overestimation bias commonly found in Q-learning. Below is a detailed description of the development process.

2.2 Project Structure

1. main.py

   This is the main loop connecting the environment and the Mario agent.

2. Marioagent.py

   Defines how the agent collects experiences, makes actions based on observations, and updates its action policy.

3. Mariowrapper.py

   Handles environment pre-processing logic such as observation resizing and conversion from RGB to grayscale.

4. neural.py

   Defines Q-value estimators using a convolutional neural network.

5. replay.py

   To evaluate a trained Mario we use this file

6. Mariometrics.py

Contains a MetricLogger to track training and evaluation performance.

2.3 Key Metrics

- Episode: Current episode number.
- Step: Total number of steps Mario has taken.
- Epsilon: Current exploration rate.
- MeanReward: Moving average of episode rewards over the past 100 episodes.
- MeanLength: Moving average of episode lengths over the past 100 episodes.
- MeanLoss: Moving average of step losses over the past 100 episodes.
- MeanQValue: Moving average of predicted Q values over the past 100 episodes.

2.4 Experience Replay

Experience replay allows the agent to store and reuse past experiences to stabilize training. It involves the following steps:

- **Store experiences**: Store the current state, action, reward, and next state.
- **Sample experiences**: Randomly sample from the memory for training.
- **Batch learning**: Use batches of experiences for training to reduce variance.

2.5 Setting Up the Environment

The environment for training the agent was set up using the `gym-super-mario-bros` package, which provides a customizable Mario game environment compatible with OpenAI's Gym. This allowed for seamless integration of the environment into the RL framework.

2.6 Limiting the Action Space

Listing 1: Limiting the action space

```
# Limit the action-space to
#   0. walk right
#   1. jump right
env = JoypadSpace(env, [["right"], ["right", "A"]])
```

Explanation:

- The action space is limited to two specific actions using the `JoypadSpace` wrapper:
  - Action 0: Walk right (`["right"]`)
  - Action 1: Jump right (`["right", "A"]`)

2.7 Environment Reset and Taking an Action

Listing 2: Environment reset and taking an action

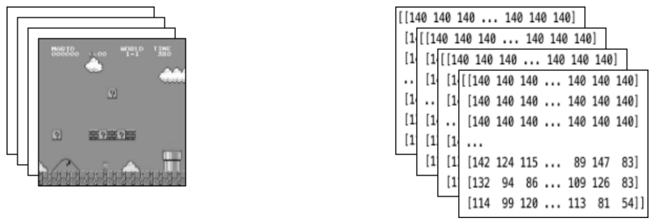
```
env.reset()
next_state, reward, done, trunc, info = env.step(action=0)
print(f"{next_state.shape},\n {reward},\n {done},\n {info}")
```

Explanation:

- The environment is reset to its initial state using the `env.reset()` function.
- An action (in this case, `action=0`, which corresponds to walking right) is performed using the `env.step(action=0)` function.
- This function returns:
  - `next_state`: The state of the environment after the action is taken.
  - `reward`: The reward received for taking the action.
  - `done`: A flag indicating whether the episode has ended.
  - `trunc`: A flag indicating if the episode was truncated (ended prematurely).
  - `info`: Additional information about the environment.
- The shape of the next state, the reward received, the done flag, and additional info are printed to the console.

2.8 Preprocessing the Environment

To ensure the agent received relevant information without unnecessary complexity, the environment observations were preprocessed. This involved converting the images to grayscale, resizing them, and stacking consecutive frames to give the agent a sense of motion.



```
[[140 140 140 ... 140 140 140]
 [140 140 140 ... 140 140 140]
 [140 140 140 ... 140 140 140]
 ...
 [142 124 115 ...  89 147  83]
 [132  94  86 ... 109 126  83]
 [114  99 120 ... 113  81  54]]
```

Figure 1: Preprocessed Environment

Explanation:

- Grayscale conversion was done to reduce the complexity of the image input.
- Images were resized to 84x84 pixels to standardize the input size.
- The `GrayScaleObservation` and `ResizeObservation` wrappers were used for these preprocessing steps.

2.9 Building the Agent

We create a class `Mario` to represent our agent in the game. Mario should be able to:

- Act according to the optimal action policy based on the current state (of the environment).
- Remember experiences. Experience = (current state, current action, reward, next state). Mario caches and later recalls his experiences to update his action policy.
- Learn a better action policy over time.

Explanation:

- The neural network consisted of three convolutional layers to extract features from the input images.
- Fully connected layers were used to map these features to Q-values for each possible action.
- The architecture was designed to handle the high-dimensional input and produce meaningful action-value predictions.

2.10 Core Components of the Agent

2.10.1 Caching Experiences

The agent stores its experiences in a replay buffer to facilitate efficient learning. This involves saving the current state, next state, action taken, reward received, and whether the episode ended.

2.10.2 Recalling Experiences

The agent samples a batch of experiences from the replay buffer. This batch is used to update the Q-values during the learning phase.

2.10.3 Temporal Difference (TD) Estimate & TD Target

- The TD Estimate is the current estimate of the Q-value for a given state-action pair.
- The TD Target is the target value used to update the Q-value, calculated using the reward received and the maximum Q-value of the next state.

2.11 Learning

The agent updates its Q-values by minimizing the loss between the TD Estimate and the TD Target. This is done using gradient descent, with the loss function defined as the squared difference between the TD Target and TD Estimate.

The key equations used in training are as follows:

Q-value Update Equation:

$$Q(s_t, a_t) = r_t + \gamma \max_{a'} Q'(s_{t+1}, a') \tag{1}$$

Where:

- $Q(s_t, a_t)$ is the Q-value for state s_t and action a_t
- r_t is the reward received after taking action a_t in state s_t
- γ is the discount factor

- $\max_{a'} Q'(s_{t+1}, a')$ is the maximum Q-value for the next state s_{t+1} over all possible actions a'

Loss Function:

$$L = \left(r_t + \gamma \max_{a'} Q'(s_{t+1}, a') - Q(s_t, a_t)\right)^2 \tag{2}$$

Explanation:

- The Q-value update equation helps the agent to learn the expected future rewards.
- The loss function minimizes the error between the predicted Q-values and the actual rewards received.

3 Outputs

Initially, when executing the ‘main.py‘ file, Mario’s gameplay is visibly flawed, with frequent mistakes and suboptimal movements. This initial behavior is expected as Mario operates without prior experience or learning. However, upon running the ‘replay.py‘ file, the system begins to leverage past experiences through the mechanism of experience replay. This iterative learning process allows Mario to refine his actions over time. With each successive episode, noticeable improvements in Mario’s movements and overall gameplay performance are observed, demonstrating the efficacy of the learning algorithm.

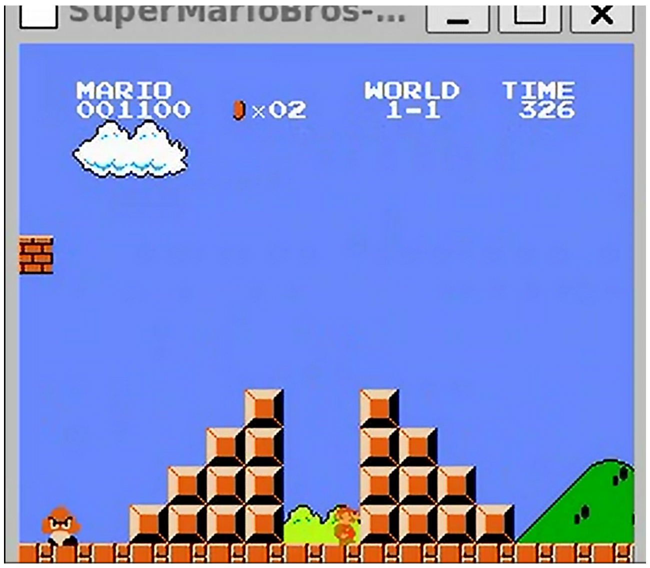


Figure 2: Mario playing in main.py file

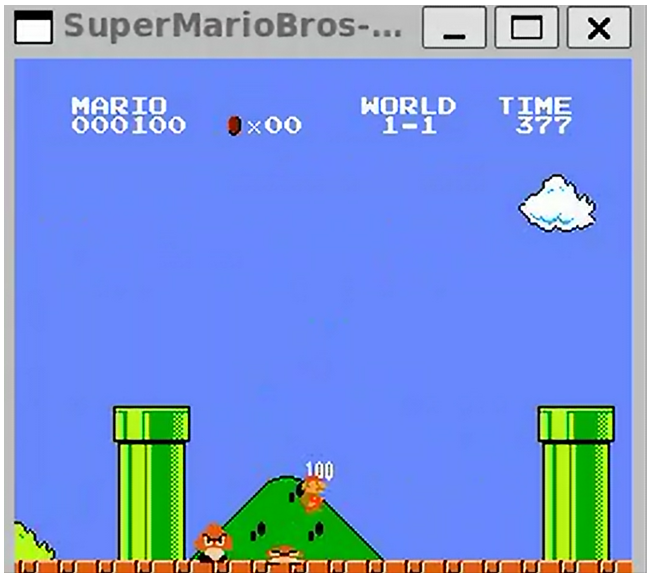


Figure 3: Mario playing in replay.py file

In the given images, Mario initially exhibits erratic and ineffective behavior, leading to frequent losses in the game. This ”dumb” behavior is indicative of the early stages of training, where Mario has not yet learned optimal strategies. However, as the training progresses and Mario learns from past experiences, his gameplay improves significantly. The later images show Mario navigating the game more skillfully, avoiding obstacles, and surviving for longer periods, highlighting the successful application of the learning algorithm.

4 Conclusion

By following the steps outlined above, we successfully developed and trained an agent capable of playing Mario using reinforcement learning techniques. Through iterative training and continuous interaction with the game environment, the agent was able to refine its strategies and enhance its performance progressively. This improvement underscores the effectiveness of reinforcement learning in navigating and mastering complex decision-making tasks.

Our project not only highlights the practical application of reinforcement learning but also demonstrates its potential in handling intricate and dynamic environments. By adapting to the varied challenges of Super Mario Bros., the agent showcases how advanced RL methods can be employed to solve problems that require both strategic planning and real-time adaptation. This endeavor serves as a compelling example of how reinforcement learning can be harnessed to develop sophisticated solutions for gaming and beyond, paving the way for further exploration and innovation in this exciting field.

Resources

1. Hado V. Hasselt, Arthur Guez, David Silver. *Deep Reinforcement Learning with Double Q-learning.* NIPS 2015. Available at: `https://arxiv.org/abs/1509.06461`

2. OpenAI. *Spinning Up in Deep Reinforcement Learning.* Available at: `https://spinningup.openai.com/en/latest/`

3. Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction.* Available at: `https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf`

4. Sebastian Heinz. *super-mario-reinforcement-learning.* GitHub repository. Available at: `https://github.com/sebastianheinz/super-mario-reinforcement-learning`

5. Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet.* Available at: `https://www.alexirpan.com/2018/02/14/rl-hard.html`