

Super Mario World AI

Siebre Cosijn

August 31, 2022

1 Introduction

This graduate work is based on the popular game Super Mario World, developed by Nintendo for the SNES¹. The game consists of many different levels, during which the objective is to reach the “goalpost” to advance to the next level before the timer runs out. To achieve this Mario will have to jump over and on top of platforms, avoid obstacles, and defeat enemies.



Figure 1: Screenshot of the Super Mario World start screen.

The goal of this work is to teach an AI how to play the game using reinforcement learning. We train an agent on some of the more basic levels and observe how it learns to deal with the various obstacles presented by the game. Afterwards we evaluate how the agent performs on those levels and see if it can successfully complete them. For this task we use OpenAI Gym² (a library for developing and comparing reinforcement learning algorithms) and Gym Retro³ (additional game integrations for Gym, including Super Mario World).

2 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning which studies how intelligent agents learn to take actions in an environment by trial and error. An agent is not told which actions to take, but must explore for itself which actions result in the highest reward. The general idea behind RL is that rewarding good behavior or penalizing bad behavior makes the agent more or less likely to repeat this behavior in the future. How the agent interacts with its environment is shown in the agent-environment loop in Figure 2.

¹Super Nintendo Entertainment System

²<https://github.com/openai/gym>

³<https://github.com/openai/retro>

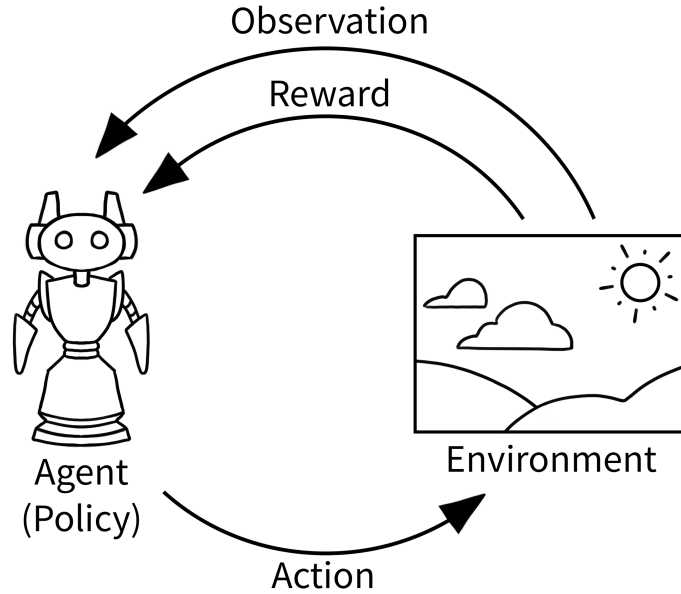


Figure 2: The agent-environment interaction loop. Image source: <https://www.gymlibrary.dev>.

The agent is trained in episodes which consist of a number of steps. At each step, the agent receives the current observation and reward. It then chooses an action to perform on the environment. The action changes the state of the environment, and the environment returns a new observation and a reward based on how the action influenced the state. Each component of the agent-environment interaction is described in more detail below.

Agent. The decision maker of the process. How the agent chooses its actions is decided by a policy that tries to maximize the some form of cumulative reward. In our case the agent controls Mario and decides which actions he will take.

Action. Agents interact with their environment through actions. In this game, an action is a combination of button presses. The set of all valid actions in an environment is called the *action space*.

Environment. World that the agent lives in (level of the game). The environment is represented by states, which are complete descriptions of the world.

Observation. Part of the state that the agent is able to see (i.e., its immediate surroundings). Observations are, in this case, “frames” of the game. A frame is an image of the game; a raster of pixels of the form $\text{height} \times \text{width} \times \text{channel}$. The *observation space* defines the structure and legitimate values an observation can take.

Reward. A numeric value that describes how good (or bad) the result of an action is on the state of the environment. The value is determined by the reward function (more on rewards in Section 4).

3 Preprocessing

Before we can train our agent a number of preprocessing steps are necessary. This includes a transformation of both the action and observation space. Most of these steps are based on papers discussing the Arcade Learning Environment (ALE) [Bel+13; Mac+18]. The ALE is a framework that allows people to develop AI agents for Atari⁴ games and is commonly used to benchmark reinforcement learning algorithms. Even though the ALE was created for Atari games, most of the preprocessing steps used are also applicable to the SNES (and Super Mario World).

⁴Atari 2600, originally known as Atari VCS (Video Computer System)

3.1 Action Space

By default, the Gym Retro environment for Super Mario World uses a MultiBinary⁵ space with 12 elements (buttons). Each action contains a value for every button in the action space where 0 (not pressed) or 1 (pressed). This results in $2^{12} = 4096$ possible combinations. Sticking to the default MultiBinary action space leads to three issues:

- **Large action space.** Exploration in RL relies on some form of random sampling. If the action space is large, it will take longer for the agent to find good actions during exploration.
- **Invalid combinations.** Some combinations of buttons cannot be pressed at the same time (e.g. left and right arrows).
- **Irrelevant buttons.** Some buttons are only used to control the game menu or pause the game and have no effect on the actual gameplay.

To solve these problems we take a look at the SNES controller in Figure 3. The game was designed to be played on this controller, so the buttons used in Gym Retro correspond to the buttons on this controller. Select, start, and the buttons on the back of the controller are not relevant for our agent. The remaining buttons can be split into two clusters, arrow keys and special keys. For each cluster, only one button can be pressed at any given time. This translates to a MultiDiscrete⁶ action space, which is the cartesian product of (in this case two) discrete spaces. The actions of both discrete spaces are described below. In this game the **X** and **Y** keys serve the same function, so we only need to include one of them. Not pressing a button is also a valid action.








- **Arrow keys.** none, left () , right () , up () , down ()
- **Special keys.** none, spin () , jump () , run/throw ()



Figure 3: SNES controller.

The resulting MultiDiscrete action space contains $5 \cdot 4 = 20$ different actions. So we have reduced the action space by a significant amount while maintaining the necessary actions to play the game.

3.2 Observation Space

3.2.1 Transform observation

To reduce the memory requirement, each observation is resized from 256×224 to 84×84 pixels. The color is also converted from RGB to grayscale. Figure 4 shows the result of this transformation. Lowering the memory requirement significantly reduces the training time of the agent.

⁵<https://www.gymnasium.dev/content/spaces/#multibinary>

⁶<https://www.gymnasium.dev/content/spaces/#multidiscrete>

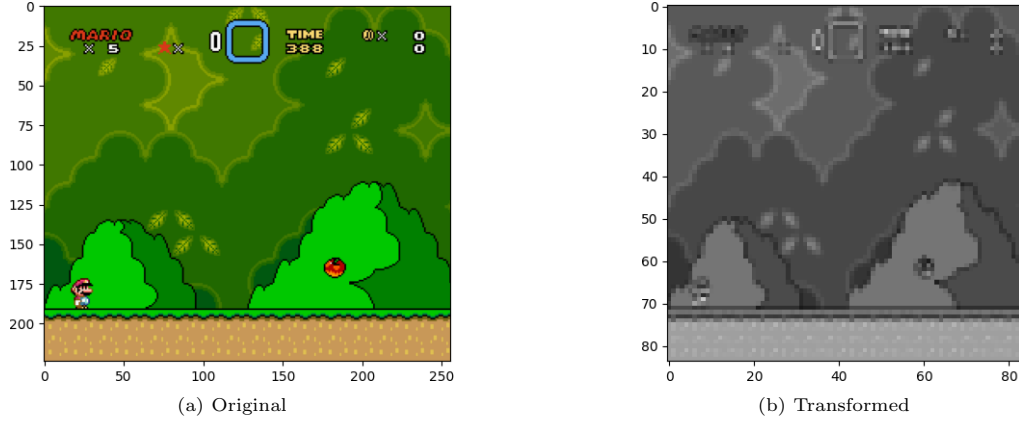


Figure 4: Transforming the observation. Resized to 84×84 pixels and converted to grayscale.

3.2.2 Skip frames

Super Mario World runs at 60 FPS (frames per second), which means there is little change in the game from frame to frame. As a result, there is no need to input a new action for each frame. Having the agent decide on a new action every few frames instead of every single frame speeds up the execution, allowing it to play more episodes during training. To create this effect of “skipping” frames each action is repeated over four frames and only the last frame is returned as observation, together with the sum of the accumulated rewards.

3.2.3 Stack frames

The agent is trained on pixels, so each observation can be seen as some sort of screenshot of the game state. By using a single frame, the agent cannot perceive the direction or velocity of movement (Is Mario going up or down? How fast is Mario moving?). To solve this we “stack” four subsequent observations and use the result as input for the agent. Each observation is concatenated with the previous three most recent observations. In Figure 5 we can see based on a stack of four observations that Mario is jumping to the right.



Figure 5: A frame stack of four subsequent observations.

3.3 Other

3.3.1 Episodic Life

In Gym environments, the current episode ends when the “done” signal is received. However, in this game the player starts with five lives, so this signal is only sent once the life counter reaches zero. Every time Mario dies the game exits the current level and returns to the level selection menu. Since no signal

has been sent, the agent does not understand it is no longer in the intended level, and as a result can get stuck in the menu or even enter another level by accident. To avoid this we want to terminate the episode after a single life is lost. Ending the episode after a life is lost also creates a positive side effect; the longer the episode continues, the more rewards the agent can gain, so the agent learns to avoid death without explicitly defining a penalty for losing a life.

3.3.2 Sticky Actions

Super Mario World is largely deterministic. Most enemies and other moving objects will always appear in the same location at the same timestamp of a level. We want the agent to learn to make good decisions in various game scenarios, not just memorize one action sequence in a single level, so we need to introduce some form of stochasticity. With “sticky actions” [Mac+18] each time the agent chooses a new action there is a 25% chance it will repeat the previous action instead. This makes it so the agent is never sure where it will end up at any given time, and forces it to act accordingly.

4 Reward

The goal of an agent in reinforcement learning is to maximize its cumulative reward. After each step the agent receives a reward from the environment, which describes how good (or bad) the action taken during this step was. The value of this reward is determined by the reward function. A well designed reward function is important in order for the agent to achieve the desired behavior.

Gym Retro’s default rewards are most often tied to the game score. In Super Mario World the player gains points by collecting coins or defeating enemies. However, the goal of a human player is usually beating the game by completing all the different levels, not maximizing the score. When trained using the score as reward, the agent will find some unintended way to maximize its score (defeating the same enemy over and over, for example) without ever finishing a level. An alternative reward is needed if we want the agent to act similar to human players.

In theory you could design a reward function that gives +1 for completing a level of the game. This reward is not practical though, because it is too “sparse”. The duration of a single episode is limited to several thousand steps due to the timer present in the game. It is very unlikely the agent will reach the end of a level by taking random actions before the timer expires, so it might never see a positive reward. The agent cannot learn which actions are good with such a restrictive reward. So we are looking for some reward that is easier to obtain but should still result in completing a level if maximized.

Mario starts each level on the left side and the flag which indicates the end of that level is always on the right side. We can reward the agent for moving right by comparing Mario’s horizontal position (x-value) before and after each step. Subtracting the x-value x_t at step t with that of the previous step x_{t-1} can result in the following three scenarios.

$$\begin{aligned} x_t - x_{t-1} > 0 & \iff \text{Mario is moving to the right} \\ x_t - x_{t-1} = 0 & \iff \text{Mario is standing in place} \\ x_t - x_{t-1} < 0 & \iff \text{Mario is moving to the left} \end{aligned}$$

The magnitude of the reward is determined by how much Mario moved in a single step. Moving faster will result in a larger reward (or penalty in case of moving left).

There is one more simple improvement we can make. By applying a small penalty (−1) on each step we incentivize the agent to try to complete the level as fast as possible. Combining both gives us the reward r_t at each step t .

$$r_t = x_t - x_{t-1} - 1$$

5 Algorithm

With the preprocessing and reward design completed, only the choice of algorithm remains before we can start training. Reinforcement learning algorithms can be divided into two branches: *Model-Based RL* and *Model-Free RL*. In model-based RL the agent uses a model (a function which predicts how the environment will react to actions) to make decisions. The model can either be known beforehand (not

the case here) or learned through interaction with the environment. The main advantage of model-based algorithms is that they are sample efficient; they learn the most out of each experience. By contrast, model-free algorithms sample purely from experience instead of using predictions of the next state. In general, model-free methods are computationally cheaper than having to learn a model of the environment, with the trade-off of being less sample efficient. In the case of Super Mario World we opt for a model-free approach because the policy is easier to learn than the model and interaction with the environment is fast.

Model-free RL can be further divided into two main approaches: *Q-Learning* and *Policy Optimization*. Q-learning methods try to learn or approximate the optimal action-value function. This means they learn the value of state-action pairs and take the action that leads to the most valuable state. Finding the optimal action-value function will also result in the optimal policy. Q-learning methods (with function approximation) tend to be less stable and are not guaranteed to converge. However, when they work they are more sample efficient than policy optimization techniques, because they can reuse old data more effectively. Policy optimization methods try to directly optimize the policy. The probabilities of actions are modeled using a neural network. Through interaction of the agent with the environment the parameters of the network are tweaked, so that good actions are more likely to be sampled in the future. The main advantage of policy optimization methods is that they are stable and reliable.

Proximal Policy Optimization (PPO) [Sch+17] is a state-of-the-art policy optimization algorithm. PPO updates policies by taking the largest possible step to improve performance, without stepping so far that it accidentally causes the performance to collapse. It achieves this by introducing a new hyperparameter, ϵ , which defines the clipping range. The use of this parameter restricts the probability ratio (a measure of the difference between old and new policy) to the interval $[1 - \epsilon, 1 + \epsilon]$. This clipping puts a constraint on how far the new policy can be removed from the old to avoid destructively large policy updates. Some advantages of PPO are: stable and reliable, easy to tune (not too reliant on finding the best hyperparameters), easy to parallelize by collecting experience from multiple copies of an environment and processing it in batches; which makes it a good choice for this project. We use the implementation of PPO from the Stable Baselines3 library⁷.

5.1 Hyperparameters

The following hyperparameters for PPO were used to train the agent. Unmentioned hyperparameters use the default value defined in the Stable Baselines library.

Hyperparameter	Value	Description
learning_rate	1×10^{-4}	Learning rate of the Adam optimizer.
n_steps	512	Number of steps of experience to collect for each environment before a policy update. Buffer size is equal to <code>n_steps</code> \times <code>n_envs</code> .
n_envs	8	Number of environment copies running in parallel.
batch_size	512	Minibatch size. How many experiences to use for each gradient update. Must be a fraction of buffer size.
n_epochs	2	Number of epochs to run for each minibatch. Less epochs will ensure more stable updates, at the cost of slower learning.
clip_range	0.1	Clipping parameter (ϵ). A larger clipping range means the new policy can diverge more from the old policy. Small values will result in more stable updates, but will also slow the learning process.
ent_coef	0.001	Entropy coefficient. Prevents the policy from converging too quickly. Larger values keep the policy random for longer, resulting in more exploration.

Table 1: PPO hyperparameters used for training the Super Mario World agent.

⁷<https://github.com/DLR-RM/stable-baselines3>

6 Training & Results

6.1 Model I

In this model the agent was trained for 25 million timesteps on the first level of the game, “Yoshi’s Island 1”. Figure 6 shows the mean episode length and reward during training.

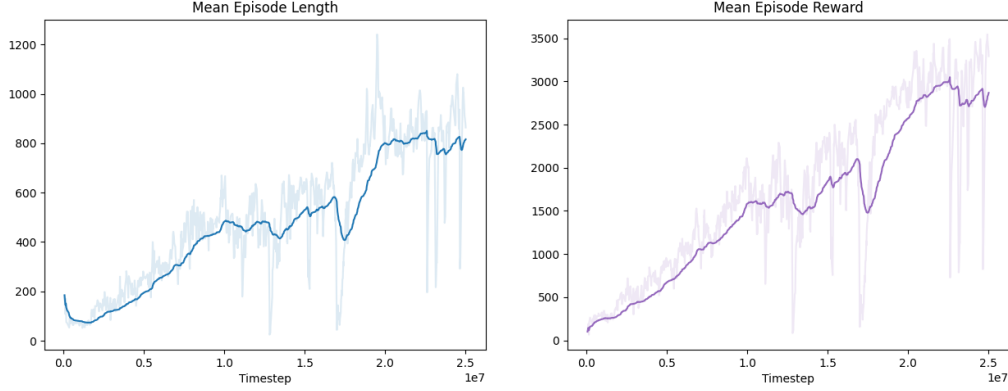


Figure 6: Results of training on the level YoshiIsland1 for 25M timesteps.

We observe a linear increase in both the episode length and reward as training goes on. The increase in length and reward follow the same trend, which means the agent is increasing its reward by staying alive longer and progressing further in the level. After 25 million timesteps the agent has learned how to complete this level.

Video of the agent playing the first level: [Yoshi’s Island 1](#)

6.2 Model II

The second model is a continuation of the first. The agent of the first model continued training for another 25 million timesteps, but this time on the second level of the game, “Yoshi’s Island 2’.

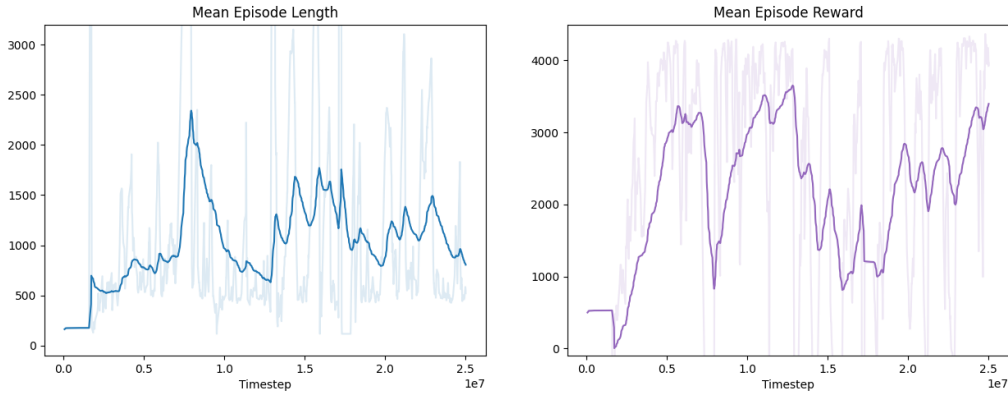


Figure 7: Results of training on the level YoshiIsland2 for 25M timesteps.

On the right side of Figure 7 we see that the agent hits a peak in reward relatively early on in the training process. As training continues the reward fluctuates and ends up at roughly the same spot towards the end. Comparing the left and right side, we see that the episode length is the lowest at those points where the reward is the highest. So we can conclude that the agent learns to complete the level faster to maximize its reward.

Video of the agent playing the second level: [Yoshi’s Island 2](#)

References

- [Bel+13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. “The arcade learning environment: An evaluation platform for general agents”. In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.
- [Mac+18] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. “Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents”. In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 523–562.
- [Sch+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).