Reinforcement Learning for Super Mario World TAS Agent

Koen Looijmans ( ), Simon Klaver ( )
Federico Cucinotta ( ), Laurens Grote ( )
Kevin Bretz

May 29, 2025

Abstract

This paper investigates the application of reinforcement learning (RL) algorithms to tool-assisted speedrunning (TAS) in Super Mario World. It compares the performance of Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC) in optimizing a single level, focusing on level-specific strategies rather than generalized gameplay. Using a combination of guided learning with tailored reward functions and constrained action spaces to address the challenges of sparse rewards, the project serves as a proof of concept for creating RL agents capable of competitive level completion and potentially contributing to full game speedruns. The findings discuss why these models failed to converge on deterministic policies for optimal gameplay.

1 Introduction

Tool-assisted speedruns (TAS) use computational tools to achieve theoretically optimal game completion times through frame-by-frame input optimization[3]. While traditionally requiring extensive human expertise, deep reinforcement learning offers potential for automating this process and discovering new strategies.

This study explores the feasibility of applying several RL algorithms to create autonomous TAS agents for Super Mario World (for the SNES), a classic platformer game with a well-established speedrunning community and a lot of examples for seemingly optimal strategies[1]. We focus specifically on three prominent RL algorithms: Deep Q-Network (DQN)[15], Proximal Policy Optimization (PPO)[20], and Soft Actor-Critic (SAC)[4, 5]. Each of these algorithms approach the problem in a unique way, thereby representing different approaches within reinforcement learning.

The primary objective of this research is to evaluate and compare the performances of each of these algorithms in learning level-specific speedrunning behaviors, while also investigating the challenges and limitations of applying RL to deterministic gaming environments. Unlike previous work[2, 8, 19] that researches the application of RL on general gameplay, our approach emphasizes optimization of an individual level, trading

generalization for potential performance gains in a specific scenario.

Our methodology incorporates guided learning through carefully designed reward functions and constrained action spaces, representing a hybrid approach between pure exploration and learning with limited actions. This strategy aims to address the substantial exploration challenges inherent in the sparse-reward environment of Super Mario World while maintaining the potential for discovering novel optimal strategies.

The main focus of this project is to serve as a proof of concept to see if we can construct a reinforcement learning model that can finish a level of Super Mario World in a competitive time. To achieve this goal, the model will have to be specialized in and over-fitted to a single level. This means that the resulting model will only be able to perform well in the level it was trained on, but will fail at its task on other levels. If this can be achieved for a single level, it can be repeated for all levels (given appropriate adaptations in reward shaping and hyperparameter selection), making it a viable, yet computationally expensive tool for speedrunning games. With enough time and resources, a full sequence of models fitted to their respective levels could be able to set a new record. And with even greater resources and advances, it could be possible to train a single model to play through an entire game at a world record pace.

2 Methodology

2.1 The Environment

We selected Super Mario World (1990, SNES) for its deterministic mechanics, a common characteristic of early console games. For the environment we use Stable-Retro[16], a fork of Gym-Retro, enabling RL environment setup and direct memory address access through ROM files.

The *retro* environment does not come with any pre-made functionalities or even get-methods for significant variables that are built specifically for Super Mario World. All it provides is a *data.json* file which stores the memory addresses of specific in-game values and a *scenario.json* file for the reward function and done condition. These values returned from the *data.json* are absolute and can not be used directly to infer information about the current state of the game in relation to the previous state of the game (except for the `EndOfLevel` variable, which is a boolean value).

In order to use the information saved at the memory addresses in our done conditions and reward function, we need to convert the information to relative information, dependent on the previous frame. For example, the "death" done condition becomes true when the current number of lives is less than the number of lives in the previous frame. This approach allows us to perform many more calculations and create more complex reward functions.

To obtain these memory addresses, *stable-retro* provides a so-called "integration tool", which allows users to play the game and discover memory addresses for specific variables. However, after rigorous testing using gym-retros integration tool, we discovered that most of the memory addresses the environment was initialized with are incorrect. For example, Mario's y-coordinate did not change according to what is happening on-screen. This made it difficult to map the right variables to the right values, but after some trial

and error, we were able to find the correct memory addresses for the most important variables. Namely, Mario's x-coordinate, lives, and `EndOfLevel`.

For the reward function and done conditions, we decided to not rely on *scenario.json*, but rather make our own implementations in Python. This can be done by retrieving the needed values from *data.json* and using it to give feedback to the agent after each step. We found this approach to be easier and more straight-forward than complying to the *scenario.json* syntax.

2.2 Pre-processing

Before feeding visual data to the agents, it is important to pre-process it. This is done to limit both the amount of data used by the network and handled by the agent. The pre-processing is done as follows. The environment outputs a single *NumPy* array containing the 3 individual RGB channels, which is in turn converted into a single grayscale image. Reducing the color channels simplifies the feature extraction process and allows the network to focus on essential structural elements in the image. Lastly, the image is down-sampled to a resolution of 84x84 pixels. This compresses the input data to a much smaller form that is still well-interpretable by a CNN. These are the same input dimensions and almost the same pre-processing steps used by [11] to successfully train a RL agent to play Atari games. An example of how this looks like can be seen in Figure 2.

Figure 1: Raw screenshot of game screen

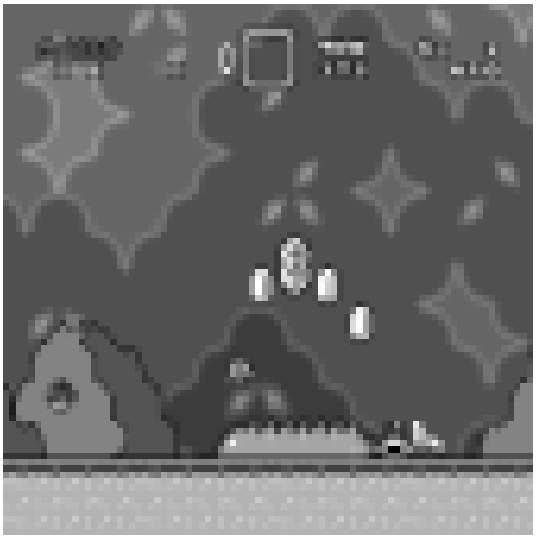

Figure 2: What the agent sees

The CNN outputs a collection of features, as seen in Figure 3, which describe the current game state, and are passed on to the actor-critic network to optimize its policy.

2.3 Guided vs Unguided RL

Depending on the goal of a project, different types of RL approaches can be taken. Reinforcement learning paradigms exist along a spectrum, ranging from purely autonomous learning to methods heavily informed by human expertise. The former takes the form of an agent learning optimal behavior solely by maximizing cumulative rewards derived

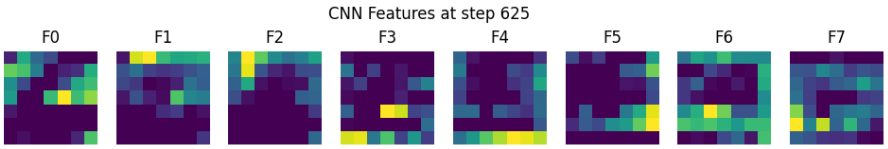

Figure 3: CNN output features

from environmental interactions, without direct human intervention in the learning loop. Instead, the latter integrates human knowledge or feedback in various forms – such as providing reward shaping tailored to the problem domain or offering demonstrations of desired actions. This is done to accelerate learning, or guide the agent towards safer or more desirable behaviors.

When choosing the goal for our project, we made the distinction between different types of agents all with their own nuances:

- A general agent able to complete most levels of SMW with no specific time constraint
- A general speedrunning agent able to independently find the end of a given level and optimize its route
- A guided speedrunning agent highly optimized for a single level

Given the time and compute constraints that we had, we chose to go for the third option. That is, a guided speedrunning agent highly optimized for a single level. This type of agent is better suited for a more guided approach in the training phase for the reasons explained below.

First, because we are trying to optimize a specific level, it is possible to greatly speed up training by only allowing the agent to perform actions that we know are necessary for completing the level as fast as possible. For example, if we know that in order to complete a given level optimally, the agent only needs to use a combination of two different actions (e.g. run right and run jump right), we can train the agent with an action space containing only these actions. This speeds up training significantly, as the agent doesn't need to first go through a long exploration phase of finding out that all other actions lead to worse rewards.

Because of the level-specific information that goes into manually defining an appropriate reward function and hand-picking the correct action space, a model trained using these metrics can inherently only perform well in an environment that rewards the resulting behavior. If a level requires Mario to move left, up, or down to finish said level, an agent trained to go only to the right will not be able to finish that level. This, however, is part of the approach of overfitting the model to the level, and if multiple levels or an entire game is to be completed, a separate agent will need to be trained, and an appropriate action space and clever reward function defined for each level.

One flaw of guiding the agent using hand-crafted action spaces and reward functions is that we are inherently limiting its exploration. We know that humans have completed this level in record times by moving to the right and jumping at precisely the correct moments, so we are manually pushing the agent to independently converge towards and mimic this behavior.

However, if the goal is to mimic a world-record run and then introduce entropy to try and find a better sequence of actions, this can be achieved far more reliably using imitation learning algorithms. Furthermore, by enforcing our agent to behave more and more like a speedrun of a level we know results in a good time, we are eliminating the chance that, by doing something completely different, we may actually get an even faster time. The agent is not independently discovering a novel sequence of actions that lead to faster level completion, it is simply mimicking and optimizing behavior we have determined to be important for achieving its goal.

3 Training the Agents

3.1 DQN

Our DQN implementation employs a convolutional neural network (CNN) to learn optimal navigation policies from preprocessed game frames. The agent uses epsilon-greedy action selection and stores experiences in a replay buffer for stable batched updates. Target networks and CNN feature extraction enhance training stability.

The agent's learning process is based on associating visual patterns with actions that result in positive rewards. It begins by receiving a raw image of the game screen. This image is resized and normalized to form a state. The CNN then extracts features from this state by identifying spatial patterns, edges, and textures. Although the network doesn't understand objects like coins, it learns to recognize visual cues that correlate with positive outcomes. The CNN's outputs are passed through fully connected layers that estimate the Q-values for each possible action, representing expected future rewards.

Learning occurs through trial and error; the agent interacts with the environment, takes actions, and observes rewards. When an action leads to a desirable outcome —such as increased x-coordinate (reward = next_x - current_x)— the agent updates its Q-value estimates. Each experience is saved to the replay buffer using the add_experience function, which records the current and next visual states. During the update step, these states are used together with the received reward to calculate a target Q-value, reinforcing the action that led to positive movement.

It is important to note that the agent does not compare consecutive frames directly to detect motion. Instead, it associates a single frame (state_raw) with the reward signal derived from the x-coordinate change. For example, think of it as receiving snapshots in a dark room: you do not see continuous motion, but you learn that a specific snapshot followed by a "moved 1 meter" signal indicates that the "forward" action is beneficial. The info dictionary, which includes the x-coordinate and lives, provides the necessary feedback for this learning.

3.1.1 Algorithm operation

DQN are Deep Q-Networks combine Q-learning with deep neural networks and can handle complex and high dimensional state spaces. It uses a neuronal network to approximate the Q-function. To reduce sequential interference between single steps, it uses a replay buffer, meaning it stores mini-batches in it's memory and samples from it during training.

To update the Q-values, the Bellman equation is used [15].

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \tag{1}$$

where: - $Q(s,a)$ represents the action-value function, which gives the expected return for taking action a in state s, - $R(s,a)$ is the immediate reward for taking action a, - γ is the discount factor, - $P(s'|s,a)$ is the transition probability from state s to s' given action a, - $\max_{a'} Q(s',a')$ represents the best possible future return from the next state s'.

To adjust the weight within the network, the loss gets computed. The loss is the difference between the predicted Q-value and the actual computed Q-value. With the help of the Adam-optimizer, a gradient descent technique, the loss gets back-propagated through the network and the weights altered.

On one hand, the agent wants to exploit it's knowledge during training, to receive the highest rewards. On the other hand it also wants to explore alternative routes and discover shortcuts. To be able to do so, the already mentioned epsilon decay approach guides the action choice for the training [18]. To decide if the action is taken randomly or model predictions are applied during training, an epsilon threshold gets defined. It has a start, end and decay value. The start value ensures that random steps for exploration are applied, in our case with a start-epsilon value of 1.0, the agent first will do 100% random exploration during the first 4 steps, after that getting reduced by epsilon decay.

3.1.2 Hyperparameters

For the DQN hyperparameters we made the following decisions:

- Learning rate α: 10^{-4}; set to a lower value to make the training more stable, even if slowing down the training process with it.
- γ: 0.99; This discount factor determines how much influence future rewards have in Q-learning; since we want Mario to jump early before he reaches an enemy and dies, we set it to 0.99, which is a middle to high value for the discount factor.
- The batch size is set to 32
- Hidden layers: 256
- ϵ-decay values
  - ϵ-start: 1.0, we want the agent to use 100% exploration first.
  - ϵ-end: 0.05; The agent should always have a little bit exploration to be able to find better solutions, while relying heavily on exploitation in the end.
  - ϵ decay: $2 \cdot 10^{-5}$; slowly reducing ϵ to get a higher exploitation rate.

3.2 PPO

The Proximal Policy Optimization (PPO) algorithm is a popular policy gradient method used to train the policy network. Unlike DQN, which is value-based, PPO directly optimizes the policy while constraining updates to prevent large deviations from the previous policy. This ensures stable learning and improved sample efficiency.

The key idea of PPO is to maximize a **clipped surrogate objective** that penalizes large changes in the policy. The PPO loss function is given by:

$$L^{\text{PPO}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2}$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the current and old policies.
- $\hat{A}_t$ is the advantage estimate at time step t.
- ϵ is a small hyperparameter (e.g., 0.2) controlling the clipping range.

The clipping ensures that the policy does not move too far from the previous policy by limiting $r_t(\theta)$ to $[1-\epsilon, 1+\epsilon]$. The overall PPO algorithm updates the policy parameters θ by maximizing this clipped objective using stochastic gradient ascent.

3.2.1 Implementation

We are using our own implementation of PPO, which was created for an assignment in a separate class on reinforcement learning with the objective of solving the Cartpole gymnasium environment [20]. Our implementation proved very effective for that task, but has not been benchmarked or tested on any other environment. All of the fundamental PPO hyperparameters, as well as various training parameters, are tunable in a dedicated config.py file. For the advantage estimation, we use the popular General Advantage Estimation (GAE), following [17].

3.2.2 Hyperparameters

For all training runs, we kept a few core hyperparameters constant:

- Hidden layer size: 512 (Good middle-ground)
- Learning rate: 10^{-4} (Slow learning rate for more stability)
- Gamma (GAE): 0.9999 (Future reward is valued highly)
- Epsilon: 0.2 (Standard PPO clipping parameter, more stable learning)

Some other hyperparameters we experimented with to see how they would affect training progress include:

- Batch size: 2048 - 32768 (Higher batch sizes lead to more stable convergence, but come at the cost of more frequent crashes due to memory problems)
- Number of epochs of policy updates: 4 - 20 (Lower values lead to more stable convergence, but also higher occurrence of catastrophic forgetting)
- Image stack size: 4 - 8 (No difference in effect on learning could be determined, but a larger image stack is slower to process)
- Number of parallel environments: 4 - 16 (Has a significant relationship with the batch size - the more environments are collecting transitions in parallel, the less transitions can be collected per environment until the batch is full.
- Maximum time per episode (in seconds): 35 - 180 (Used as a done condition so the environment knows when to reset.

Finally, regarding exploration and entropy, it is common practice to add an entropy coefficient to the PPO loss function to encourage novel states, such as done by [10].

$$L^{\text{total}}(\theta) = L^{\text{PPO}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 H[\pi_\theta] \tag{3}$$

- $L^{\text{total}}(\theta)$: Policy function loss.
- $L^{\text{VF}}(\theta)$: Value function loss.
- c_1: Coefficient for the value function loss term.

- c_2: Coefficient for the entropy bonus.
- $H[\pi_\theta]$: Entropy of the policy.

Our implementation of PPO goes one step further and introduces a decaying entropy coefficient, which starts off with a high value that diminishes over the course of training. The idea behind this is that, similar to the epsilon-greedy approach used in DQN algorithms, we want the agent to start out very curious to explore and then over the course of training and after having gathered and learned from plenty of data to slowly become more confident in its actions and converge towards the optimal action sequence. The respective entropy hyperparameters are:

- Starting Entropy coefficient: 0.1 - 0.001
- Entropy decay: 0.99 - 0.9999
- Minimum entropy coefficient: 0.00001 - 0

3.2.3 Network architecture

The PPO agent uses a CNN architecture designed for Super Mario World's visual inputs, employing a deeper network than [11] due to the game's increased complexity compared to Atari games.

The network consists of convolutional layers, with parameters (filter amount, filter size, stride); (32,8,4), (64,4,2), and two times (128,3,1) with the final layer padded. Each layer is followed by batch normalization and ReLU activation. The flattened CNN output passes through two fully connected layers with *hidden_size* units and ReLU activations before splitting into policy and value heads. The policy head uses a *hidden_size*/2 hidden layer to output action logits, while the value head produces scalar value estimates.

We omitted auxiliary data (Mario's coordinates and lives) after observing unfavorable behavior in earlier iterations, assuming the agent can achieve its goals through purely visual learning despite longer training requirements. Following [12], the network processes stacked consecutive frames to capture temporal relationships between states and actions.

3.2.4 Reward shaping and done conditions

For the very first iteration of the reward function, we came up with:

- Small reward if Mario's current x coordinate > Mario's last x coordinate
- 2x Small reward if Mario's current x coordinate > Mario's highest x coordinate so far
- Small penalty per (time)step
- Large penalty for dying
- Large reward for reaching the end
- Large penalty for timing out

We did, however, come up with a way to simplify the reward function, as the opposing nature of rewarding for steps but also penalizing for steps was causing us trouble with local optima.

The new reward function looks like this:

- Small reward if Mario's current x coordinate > Mario's highest x coordinate so far

- Large reward for reaching the end

With this very simple reward function we manage to sidestep all the complexities of having to account for the temporal step penalty, which we will, once again, go into greater detail on in Section 4.
The trick here is to use the maximum time per episode as the hidden time penalty by setting it low on purpose.
For example, if we set the maximum time per episode to 30 seconds, and we know, as a fact, the level we are training on can not be beaten in 30 seconds, then the agent should, in order to maximize its reward per episode, learn to get as far into the level as possible before it times out. If we set the maximum time per episode to what we know is a competitive speedrunning time for the respective level, then the agent should, in theory and if trained correctly, converge towards this behavior.

The done conditions stayed constant throughout the whole project:

- If current lives $<$ lives last step (Dead)
- If current step $>$ maximum steps per episode (Timed out)
- If abs(Mario's current x coordinate - Mario's x coordinate last step) > 100 (Running against an object)
- If EndOfLevel == True (Mario reached the end of the level)

3.2.5 Multiprocessing

Due to the game itself being made for ancient hardware, it is extremely lightweight to run an episode of a level using the retro environment. On top of this, we have the option to not render the screen and just receive an RGB array containing the values, allowing us to significantly speed up training by not being limited by its frame-rate. The parallel environments are reset synchronously.

The amount of parallel environments to be used in training is directly controllable through the agents respective config.py file. It should be noted, though, that the amount of parallel environments being run has a significant relationship with how far into the episode each agent makes it before the batch is full and the model updates, so these two parameters (num_envs and batch_size) should be designated thoughtfully. Additionally, they are limited by computational resources.

3.2.6 Entropy Regularization

Due to early iterations of the training process converging towards suboptimal deterministic behavior much too quickly, we came up with the idea of a "death zone". This is a range of x-coordinate values where Mario has experienced many deaths. We tried increasing the entropy within this death zone so that the agent can hopefully break free from a local optimum by acting more randomly. This however didn't result in the improvements we were hoping for.

The effect and implications of entropy were a topic of extensive conversation during the course of this project, starting out as simply a constant entropy coefficient chosen as an afterthought, but slowly being recognized as a significant factor in the overall performance of our models, and the crux to the question of why argmax evaluation performed so much

more poorly than stochastic sampling. We go into further detail on the topic of entropy in Section 5.

3.2.7 Curriculum Learning

To speed up training, we employed curriculum learning, which is the idea of first training the model on a simpler task, and, as the model gains confidence in this, gradually introducing further complexity. This is an already established reinforcement learning technique called "Curriculum Learning", as explained in [14].

We approached this by first having an agent learn to simply finish a level without any temporal penalty applied at all. Once the agent can confidently achieve this, we can start penalizing sub-optimal behavior to enforce convergence towards optimal behavior, without worrying that the agent unlearns its core objective of reaching the end of the level.

Regretfully, this didn't play out as we had intended. In its initial, most simple training stage, the agent does learn to reach the end of the level, but with stochastic confidence. This is because there is no single optimal solution for the agent to reach the end of the level if we ignore the temporal aspect. An agent reaching the end of the level in 100 steps will receive the same reward as an agent who reaches it in 1000 steps, meaning that there are an endless amount of potential action sequences for achieving the maximum reward, so there is no optimum for the agent to converge towards. It does still learn to maximize its reward per episode, but, as mentioned, this is with stochastic confidence, identifying a subgroup of actions as being equally conducive to achieving a high reward rather than a single action per state.

In Section 5, we propose a possible curriculum learning strategy that may lead to better results, but for which we did not have enough time to experiment with ourselves.

3.3 SAC

Soft Actor-Critic (SAC) is an off-policy actor-critic[9] deep reinforcement learning algorithm based on the maximum entropy reinforcement learning framework[4, 5]. Unlike traditional reinforcement learning approaches that solely maximize expected reward, SAC optimizes a trade-off between expected return and policy entropy (randomness). This causes the agent both to try and succeed at the task while also acting as randomly as possible (exploration). This entropy regularization provides several key advantages: improved exploration, more robust learning, and the ability to capture multiple modes of near-optimal behavior.

Just like DQN, as explained in Section 3.1.1, SAC makes used of a replay buffer, in which experiences from both the current and older versions of the policy are stored. This is what makes SAC off-policy. The replay buffer contains records of form (s, a, r, s', d), where s is the current state, a is the action taken at state s, r is the reward for performing action a, s' is the next state, and d denotes if the program is finished (done).

The core objective of SAC can be expressed as maximizing the objective function, which is the expected entropy-regularized return over all states present in the replay buffer:

$$J(\theta, \mathcal{D}) = \mathop{\mathbb{E}}_{s\sim\mathcal{D}} \left[\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)} \left[\min_{i=1,2} Q_{\phi_i}(s,a) - \alpha \log \pi_\theta(a|s) \right] \right]$$

where

- π_θ is the policy (network) with set of parameters θ,
- $\mathcal{D}$ is a given replay buffer,
- s is a state in replay buffer $\mathcal{D}$,
- a is an action in policy π_θ for state s,
- Q_ϕ is a Q-network with set of parameters ϕ,
- $Q(s, a)$ is the Q-value for performing action a in state s,
- α is the entropy coefficient or temperature that controls the trade-off between reward and entropy maximization, and
- $\log \pi_\theta(a|s)$ represents the entropy of the policy for performing action a at state s.

For the training of the SAC algorithm, the following loss function is used:

$$L(\phi_i, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_i}(s, a) - y(r, s', d) \right)^2 \right]$$

where $y(r, s', d)$ is defined as

$$y(r, s', d) = r + \gamma(1 - d) \cdot \left(\mathop{\mathbb{E}}_{a'\sim\pi_\theta(\cdot|s')} \left[\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a') - \alpha \log \pi_\theta(a'|s') \right] \right)$$

where

- γ is a discount factor
- a' an action taken from next state s', and
- $Q_{\phi_{\text{targ},i}}$ is the target Q-network with set of parameters ϕ corresponding to Q-network Q_{ϕ_i}

Do note d is a boolean value of either 0 or 1; meaning that if for a certain next state s' the algorithm is done, $y(r, s', d) = r$. Another note is that this definition of $y(r, s', d)$ only works for discrete action spaces, which is relevant for the environment our implementation is used in (as described in Section 2.1). For a continuous action space sampling would occur, replacing the expectation $\mathbb{E}$ as no full expectation can be considered over an infinite amount of actions.

These formulas effectively means the Mean Squared Bellman Error (MSBE) is taken over the Q-networks: the mean squared error with regard to the Q-networks and, in the case of SAC, also with regard to the entropy. As the action space is very limited, the full expectation over all actions can be easily computed.

3.3.1 Implementation

SAC is implemented with three main neural networks: a stochastic policy network (the actor) and two Q-networks (the critics) with their corresponding target networks. The algorithm uses clipped double Q-learning[6, 7]. This effectively means that out of the calculated Q-values the lower, more conservative, value is being used to prevent overestimation bias. The policy is trained to maximize the expected Q-value while maintaining high entropy, while at the same time the Q-networks are updated using the MSBE with regard to their target networks and the entropy as described in Section 3.3.

The implementation as described in this paper is an adaptation of the original SAC, mainly in regard to discrete action spaces such as, in this case, Super Mario World. As

described above, this means the full expectation over all possible actions is calculated, rather than sampling from a large or infinite amount of actions.

Another adaptation is the incorporation of Prioritized Experience Replay with importance sampling. This means the algorithm prioritizes learning from experiences in the replay buffer which have higher loss, potentially speeding up convergence in environments with sparse or delayed rewards.

Thirdly, there is dynamic hyperparameter scheduling. This means that during the course of the algorithm the hyperparameters are adapted (at preset intervals) to lessen the amount of exploration, shifting from a lot of exploration to less exploration as more experience has been collected.

3.3.2 Reward Shaping

The rewards used for the SAC implementation are focused on moving Mario to the right and finishing the level, while explicitly not ending the level by dying. This is implemented as the agent getting a positive reward for moving to the right, increasing the reward for the run by $2\Delta x$, with $x \in \mathbb{N}$. For not moving to the right, or even moving to the left (in some way) the agent incurs a penalty of 1 (a reward of -1). For reaching the end of the level, thereby successfully finishing said level, the agent will receive a reward bonus of 1000, while for dying and therefore unsuccessfully finishing the level the agent incurs a penalty of 5000. Each run or episode is capped at 5000 steps, as after that point the agent would finish with a too low reward to consider useful, if it would finish at all. This effectively causes the agent to receive a negative total reward for a run where Mario does not reach the finish, and a positive total reward for a run ending with a finish.

3.4 Alternative approaches

3.4.1 State-based

After encountering challenges using visual data to teach the agent, we decided to experiment with a state-based approach. That is, omit the visual data entirely. This approach works by taking the game variables from *data.json* and give feedback to the agent simply based on the values of the changing variables as the agent performs actions in the environment. As an example, the agent can learn to go right by just receiving a reward when the x-value of the environment is increased, without having to rely on any visual data. This approach also had the advantage of significantly speeding up training time as there is no processing of the visual data to be performed. An analogy to this approach would be guiding a person to the bathroom while being blindfolded and telling the person whether it is going in the right direction or not. We found that this method helped to speed up training, but given the limitations of the variables given in the *data.json*, the agent was very limited in what it could learn.

3.4.2 Brute force

The project aimed to develop a machine learning model capable of replicating successful speedruns in a video game level, though consistency is generally less important for speedruns: for competitive speedrunning, where single record-breaking runs matter most,

the optimal approach should be able to make a record run at least once rather than always.

The training process inherently relies on intelligent brute-force sampling: agents randomly attempt actions until achieving high-reward runs. While backpropagation progressively biases the action probabilities of short-term beneficial choices, the agent does not focus on those, creating a probability distribution that slowly converges toward a global optima. This method enables capturing individual record-setting runs during training phases, even without producing a consistently high-performing model.

Practically, limiting the action space (e.g., to only run-right and jump-right) accelerates moving in the right direction, as demonstrated in our submitted recording. For speedrunning applications, this targeted brute-force approach proves more viable than training fully autonomous agents.

4 Results

4.1 DQN

The DQN agent has 42 different actions available, meaning in the first episodes he just tires out random trajectories. After a while the agent learns that for getting a higher reward he has to take an action that gets him a better x-position. As discussed in section 3.1, the only input the agent gets is a prepared gray-scaled and down scaled picture, to recognize enemies, items and obstacles (See also Figure 2). Eventually, the agent is not able to recognize any pattern in the picture. Its executions seem to be random random, except in the data that he gets explicitly through the rom addresses (x-position, dying and finishing the level). Because the level is so easy, the agent finishes it sometimes.

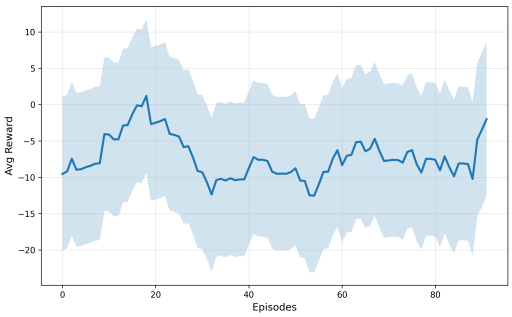

Figure 4: Smoothed Average Reward Per Episode (DQN)

Figure 4 shows what reward the agent gets after a time; No real learning behavior can be observed. The light-blue areas of steps that returned a high or vary low sore.

4.2 PPO

As detailed in Section 3, multiple reward functions were tested. Most agents demonstrated rapid early progress, learning to prioritize rightward movement within an action space of 42 options after minimal training. When episode durations exceeded two minutes,

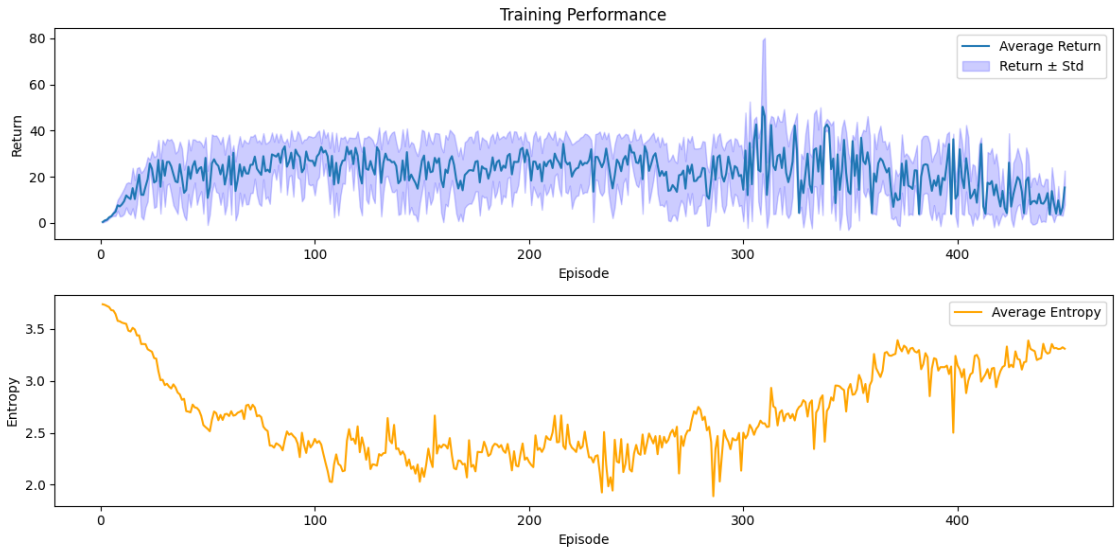

Figure 5: PPO training progress over the course of 450 episodes

agents occasionally reached the level's end by chance during initial exploration phases. However, prolonged training (hours to days) revealed systemic failure modes: models exhibited catastrophic forgetting, regressing to suboptimal policies where agents ceased meaningful action. Early progress stalled once agents encountered the level's endpoint, with models unable to optimize action sequences for consistent completion.

The simplified reward strategy from Section 3 produced distinct learning dynamics (Figure 5). Episode returns stopped increasing after about 50 training iterations, followed by 300 episodes of stagnation and eventually started decreasing. Brief performance spikes at 300 episodes reflected intermittent high-reward trajectories, but these were not retained. Entropy values remained above 2.0 throughout training, indicating low policy confidence, and began rising post-300 episodes, mirroring the return decline. Only five episodes terminated via successful level completion, underscoring the strategy's limitations.

All models suffered from stochastic confidence rather than deterministic policy optimization. While action sampling produced occasional successful runs (evidenced in simulation files), argmax evaluation consistently yielded repetitive failure sequences (e.g., continuous rightward movement into enemies). This divergence confirms models identified action subgroups with marginally higher rewards but failed to develop precise confidence hierarchies. The inability to learn context-sensitive responses (e.g., jumping vs. moving right near enemies) persisted across all configurations, preventing reliable level completion with deterministic evaluation.

4.3 SAC

The SAC algorithm was run with the following parameters: a learning rate of $3 \cdot 10^{-4}$, an (initial) α of 0.8, a γ (discount factor) of 0.95, and a hidden layer size of 512. It did 25 updates every 100 environment steps. The replay buffer was of size $1 \cdot 10^5$ from where batches of size 2048 were taken. Training did only start from the point where there were at least 5000 experiences in the replay buffer.

As can be seen in Figure 6, the training process of the algorithm is rather erratic. The

reward fluctuates between thousands in the plus and in the minus. Do note this graph already had smoothening applied with a rolling window of 5: the original graph was basically a barcode, fluctuating between 10000 in the plus (for a successful run) and 4000 in the minus (for dying), and other values in between as well. Even though this model had already been trained for thousands of episodes before as well, it shows no sign of convergence.

Another interesting aspect from Figure 6 is that the amount of steps is very closely tied to the score. This is intuitive, as the more steps the agent is alive for, the more chance it has to move to the right (giving a positive score, see Section 3.3.2). At the same time, a higher step count can also mean that the agent is running against a pipe, which does incur significant negative rewards. So it is interesting to see a higher number of steps taken results in a higher reward across most if not all episodes.

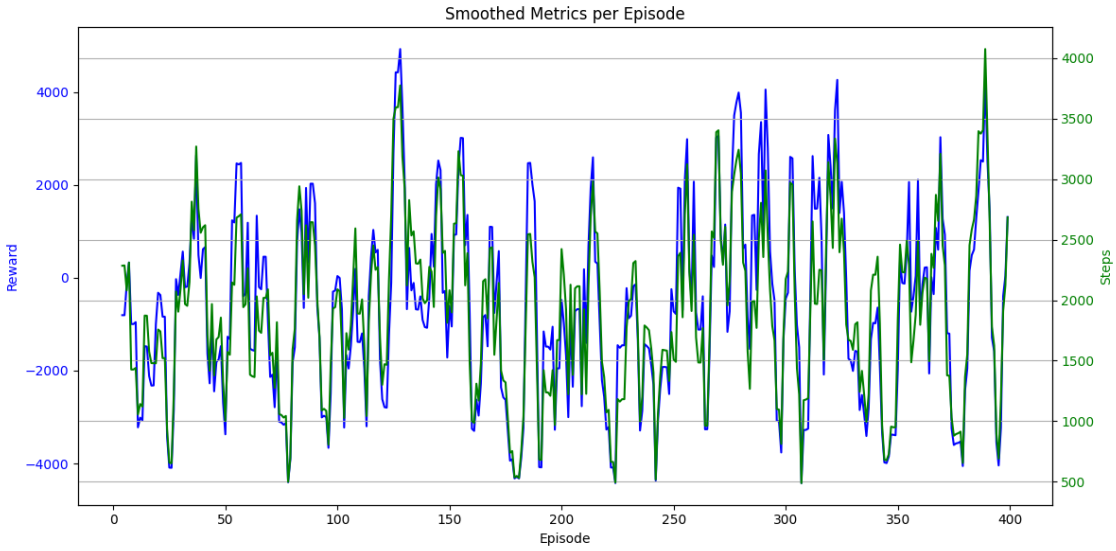

Figure 6: SAC training progress over the course of 400 episodes

5 Discussion

5.1 DQN

DQN has demonstrated success in various game-playing scenarios in the past. However, SMW comes with its own challenges. DQN's reliance on approximating the Q-function, which estimates the expected cumulative reward for each action, can be problematic in environments with sparse or delayed rewards, as is sometimes the case in speedrunning. The agent struggles to effectively propagate rewards back through time, leading to slow learning or suboptimal policies. Additionally, DQN's tendency to overestimate Q-values can result in unstable training and exploration, particularly when dealing with high-dimensional state spaces like the visual input from Super Mario World. To try and mitigate this, we used techniques such as experience replay and target networks. However, this also increases model complexity and more difficult hyperparameter tuning. Furthermore, given our results DQN's exploration strategy, which is based on epsilon-greedy, did not seem to be sufficient for discovering novel strategies or overcoming local

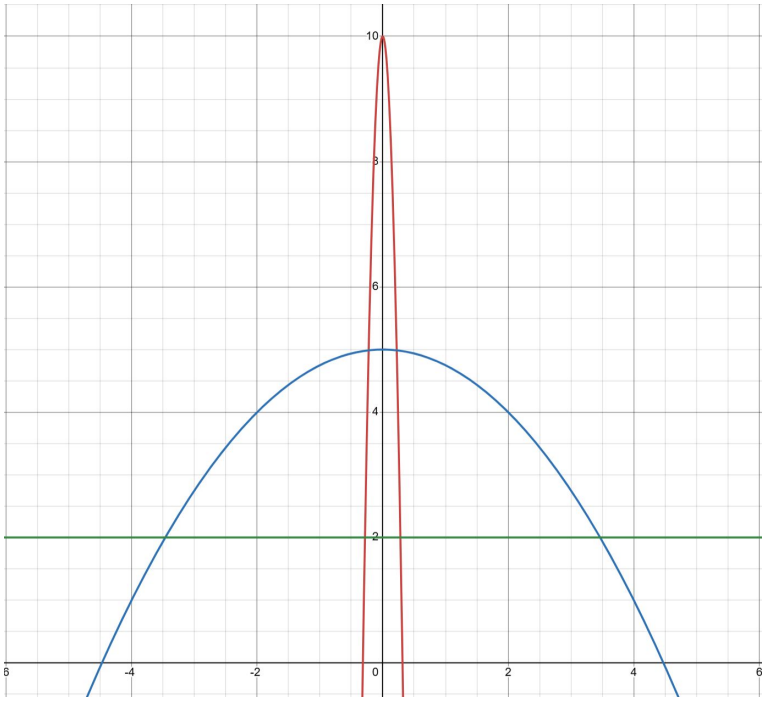

Figure 7: Red: High Deterministic Confidence, Blue: High Stochastic Confidence, Green: High Entropy

optima. Out of the three models that we have tested, we found that DQN provided the least satisfactory rewards.

5.2 PPO

Although we were regrettably not able to achieve our goal for this project of obtaining a model that can confidently speedrun a level of Super Mario World, we did learn a lot and come to many new realizations in the process. Mainly, we gained plenty of practical experience working with and tuning the PPO algorithm, and were introduced to the challenges of working in an environment with sparse rewards and a fragile reward function with many local optima.

5.2.1 Thoughts on entropy

The agent does not behave completely randomly anymore (Green line in Figure 7), but also has not been able to single out a single best action for every possible state it could encounter (Red line in Figure 7). It behaves somewhere within these boundaries, having been able to filter out actions that definitely don't result in a high reward, but between those that do lead to a potentially better reward, it is unable to learn which specific action is truly better than the others for which state (Blue line in Figure 7).

5.2.2 Thoughts on curriculum learning

Although curriculum learning did not enable us to break out of the local optima and entropy hell we were facing, we do think that it has high potential of aiding us at our specific goal.

Our agent was quite successful at finding the end of the level, but struggled a lot at optimizing its route to the end, usually resulting in catastrophic forgetting. This is a product of the stochastic nature of such agents, where if a trajectory is not being reproduced in training often enough, its behavior and effect on the model will over time be forgotten and the model will regress. Also, if there are many different sequences that lead to the same result, which applies more and more the longer the level is, then there is no true optimum for the model to converge towards and it will only gain stochastic confidence. A practical solution for this could be to set the end of the level much closer, say at x coordinate 1000, or just after the first challenging part (e.g. enemy) of the level. By shortening the section that needs to be completed, training can be exponentially sped up because there are far fewer local optima to get stuck in, and there are fewer optimal actions for each state. After the agent has learned to confidently complete the first section, it should in theory be able to learn to complete consecutive sections faster as well, as some of its learned behavior will also be relevant for the next sections.

5.3 SAC

There are a few issues with implementing SAC for Super Mario World. First of all, the algorithm is generally designed for a continuous action space, yet our action space is discrete. There are things you can do however to adapt SAC to a discrete action space [21], which we have done.

We suspect however that the main drawback of this approach is in the usage of Q-functions. These give the expected total reward for a certain action, but this is mostly correlated with how far along the level Mario is, and with that how much "potential reward" he has left. This means that the Q-functions effectively memorize where Mario is along the level, rather than what he should do, and this is not a useful metric for the agent to know. It should just judge which action is best locally, as getting through the currently visible part of the level as fast as possible, will also result in the entire level being completed as fast as possible, at least for the simple level we trained our agent on. This means that SAC is probably not suited well for this task.

6 Conclusion

6.1 Comparison

DQN achieves the lowest scores due to challenges in handling the sparse rewards and Q-value overestimation. Its epsilon-greedy exploration strategy proves insufficient for the environment's complexity.
PPO demonstrates rapid initial progress in directional movement, but then experiences a long period of stagnation, followed by catastrophic forgetting leading to a steady recession of the models abilities. While some promising models were trained, their evaluation highlights the difficulties of implementing a stochastic model into a deterministic environment with a large action space, and how tricky it can be to get the reward function right.
SAC shows extreme training instability with the reward varying by the thousands between episodes. The Q-networks memorizing the position rather than learning optimal actions seems to lead to focus on potential reward rather than optimal actions.

With all the algorithms failing to develop deterministic policy confidence, none can reliably complete the level or even achieve speedrun-level performance.

6.2 Challenges and Limitations

One notable challenge we encountered is the difference in granularity between the agent x-step and environment x-axis. This means that for the change in position of the agent to register in the environment, the agent needs to take more than one step. This is problematic, because the reward function is updated after each step. If the environment does not return an x-change after each action, then it is hard to give appropriate feedback to the agent, as the change in the x-direction is an integral part of the reward function. It was very difficult to find this issue, since the connection with the *data.json* file made it hard to read in-game variables.

This granularity mismatch presented a problem with the training times rather than the agent learning properly. This is because the agent would also learn that going left is a viable strategy for ultimately going right, provided that the agent takes more random steps to the right triggering an environment x-change. In turn, this meant that the agent had to perform many more actions before learning that going right was the right thing to do.

To fix this issue, it is possible to directly incentivize the agent to move in the right direction, rather than let it figure out that going right is the right move. We unfortunately found this out quite late in the project, which made us try many unnecessary things which we thought were the cause of the agent not learning properly. Once this was implemented, we noticed that our agent was learning faster than before.

6.3 Future work

Future research could explore the application of imitation learning techniques, wherein the agent is trained on human-generated or expert demonstrations of Super Mario World gameplay. This approach could bypass the exploration challenges inherent in reinforcement learning, rapidly converging towards competent strategies by emulating successful play patterns [13].

Future research should also consider the development of a more general agent capable of speedrunning various Super Mario World levels, rather than being optimized for a single specific level. This would require addressing the challenges of generalization across different level layouts, enemy placements, and goal locations. Potential approaches include using more robust feature extraction methods, such as convolutional neural networks trained on a diverse set of levels, and incorporating techniques like meta-learning or transfer learning to enable the agent to quickly adapt to new environments.

References

[1] Super mario world - speedrun.com. `https://www.speedrun.com/smw`. Accessed: 2025-05-28.

[2] Dobrovsky, A., Borghoff, U. M., and Hofmann, M. Improving adaptive gameplay in serious games through interactive deep reinforcement learning. In *Cognitive infocommunications, theory and applications*. Springer, 2018, pp. 411–432.

[3] Gross, M., Zühlke, D., and Naujoks, B. Automating speedrun routing: Overview and vision. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)* (2022), Springer, pp. 471–486.

[4] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR abs/1801.01290* (2018).

[5] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., and Levine, S. Soft actor-critic algorithms and applications. *CoRR abs/1812.05905* (2018).

[6] Hasselt, H. Double q-learning. In *Advances in Neural Information Processing Systems* (2010), J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23, Curran Associates, Inc.

[7] Jiang, H., Xie, J., and Yang, J. Action candidate driven clipped double q-learning for discrete and continuous action tasks. `https://arxiv.org/abs/2203.11526`, 2022.

[8] Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374* (2019).

[9] Konda, V., and Tsitsiklis, J. Actor-critic algorithms. In *Advances in Neural Information Processing Systems* (1999), S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press.

[10] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. *International conference on machine learning* (2016), 1928–1937.

[11] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[13] Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W., and Abbeel, P. Overcoming exploration in reinforcement learning with demonstrations.

[14] Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E., and Stone, P. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research 21*, 181 (2020), 1–50.

[15] Olekhnovich, S. Walking through original DQN paper, Dec. 2019.

[16] Poliquin, M. Stable retro, a maintained fork of openai's gym-retro. `https://github.com/Farama-Foundation/stable-retro`, 2025.

[17] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

[18] Sewak, M. Double dqn in code: Coding the ddqn with epsilon-decay behavior policy, 06 2019.

[19] Sriram, V. *Automated playtesting of platformer games using reinforcement learning*. Northeastern University, 2019.

[20] Srivastava, P., Swed, Y., and Bretz, K. A comparison of ppo and sac agents in the cartpole environment, 2025. Unpublished report.

[21] Zhou, H., Wei, T., Lin, Z., junyou li, Xing, J., Shi, Y., Shen, L., Yu, C., and Ye, D. Revisiting discrete soft actor-critic, 2024.