DV2586 Deep learning project: Reinforcement learning Super Mario

1st Oscar Cederlund
DVAMI19
Blekinge Institute of Technology
Karlskrona, Sweden
osce19@student.bth.se

***Abstract*—A report regarding how one might use proximal policy optimization in order to train a deep reinforcement learning agent in order to play the first level of the classic video game Super Mario Bros. Describing the challenges one might face when tying to construct and train a video game playing reinforcement learning agent. Then finally taking a dive into the inner workings of reinforcement learning, environments and algorithms to solve the problems that the environments present.**

I. Introduction

This project takes a deep dive into the reinforcement learning topic by exploring different the different concepts which together forms the topic. Starting with an introduction to the general problem of creating a deep learning solution to play and finish the first level of Super Mario, exploring what general deep learning option are available discussing their strengths and weaknesses. Going on with reinforcement learning the environment is discussed and explained in a sense that a complete beginner to the subject might understand. Further a deep dive in the inner workings of reinforcement learning concepts expands on the logic that makes reinforcement learning possible and how it interacts with the chosen algorithm proximal policy optimization which works together with convolutional networks to create a policy which the agent can follow in order to solve the problem it faces.

II. Problem Definition

A. The problem

For this project we will investigate how could use deep learning to complete the first level of the classic video game Super Mario bros [1]. This level is often called 1-1, as it is the first level of the first world that the player visits, and its starting site is familiar sight to the many that have played the game. If a proposed solution is able to complete subsequent levels within the game that's within our interest and and might be used as a metric should multiple solutions be able to complete the first level within reason.

When playing the game we want deep learning solution to have the same input as a human player would have. Thus the solution shall be able to make its decision based on what it can be seen on the frames within the game. Which would make the input for our solution vision-based, rather us gathering information from the game and presenting it to the solution to act upon.

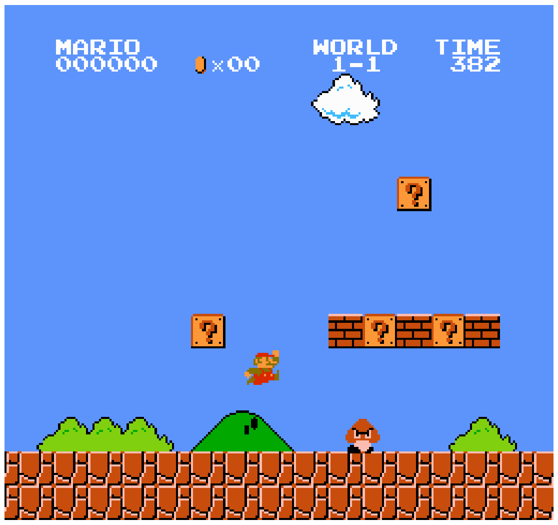


Fig. 1. The starting location for the level 1-1 in Super Mario Bros.

Which brings us to the research question for the following project:

- Research Question 1: How could use vision-based deep learning methods to play and complete the first Super Mario Bros using the frames generated as vision input?

B. Challenges

This project faces a number of challenges primarily around the method to solve the problem, which in turn rules over the how the input data shall be prepossessed and presented for the selected solution. How the vision component in a solution determines what might be an obstacle or a danger for player and how the solutions will do exploration of these elements affect training speeds.

1) Methods: There are various approaches one could could take to solve this problem such as:

- Reinforcement learning
- Supervised learning
- Evolutionary algorithms

All these approaches have their own strengths, weaknesses and manner of function.

2) Reinforcement learning: Reinforcement learning works by setting up and a reward heuristic and during the training the agent produced by model explores the action space based upon the input state in order to maximize the reward its able to gather.

3) Supervised learning: Instead of actively exploring the action supervised learning would train an agent to play Super Mario based upon a data set of expert gameplay data. This would allow the agent to learn to mimic the expert behavior based upon an input space. A consequence of this that the model would suffer in the exploration aspect.

4) Evolutionary algorithms: Such genetic algorithms or evolutionary strategies could be used to train an agent for playing Super Mario. These work by maintaining a population of agents and continually evolving them through multiple generations where the fittest agents are selected to varied through mutation and recombination.

5) Input data: Without good data there is no way that we well have a good model. This mean that we need prepare each input state from the from the model in a manner that that will help the solution to understand current state of the game by observing a visual image. How would an agent be informed that Mario is currently moving or that an enemy is moving towards Mario from a frame. There is also a discussion to be had if each frame will be necessary for the model and what trade offs are viable to make when keeping the training speed of the model in mind.

6) Vision: In each frame from the game there will be obstacles and dangers that the agent will need to manage. One way traditional way to deal with these are to flag these for the agent by detecting them in the input state, thus handling a part of the pattern recognition for the model. This way we could create a discrete observation space, helping the agent perceive the different objects as different, such as brick squares and question mark squares which often contains some sort of power-up for the player. Another approach is to let the agent learn to differentiate obstacles and players by itself through the likes of convolutional networks.

7) Training speed: As the input data is served to model on a per frame basis each frame will need to be prepossessed before heading into the model which means that we can only train our model as fast as we can prepossess the input. Besides we need to make sure that model continues to learn and wont get stuck at a local minima.

III. PROPOSED METHODS

The proposed method to solve this problem is to make use of reinforcement learning for it's suitability to be trained upon video games, along with a vast amount documentation on the topic. What makes reinforcement a suitable approach for the video game is first of its trial and error learning. Video games typically provides an interactive environment that agents receive a performance feedback in a form of reward or punishment based upon their action. By iteratively exploring exploring the game environment the agent is able to adjust its strategy based on feedback and potentially learn an optimal gameplay strategy.

Since video games consists of sequential decision making where the actions that the agent performs often has long-term consequences, being designed to learn and optimize decision making in varied sequential scenarios makes reinforcement learning very fit to this task. Being able to make actions to plan for maximizing a cumulative future rewards based upon the current dynamic of the input state.

As the input space changes when the agent acts, the agents is forced to adapt to the changing environment by continuously updating its policies. As the game progresses and training progresses the agent should learn to handle a vast amount of different scenarios and react to unexpected events and adjust its behavior accordingly. This is crucial in video games where the environment is ever changing and constantly presenting new challenges towards the player or agent.

These game environments also often involve a high-dimensional state and action spaces which makes many other planning methods quite impractical. Reinforcement learning, especially when combined with deep neural networks, can effectively handle such complexity. These can process and learn from raw pixel inputs, allowing agents to perceive and understand the game world directly from visual data.

This is because the agents are able to and encouraged to naturally explore the game environment by taking different actions and observing the resulting rewards.

A. Learning environment

In order to train a reinforcement learning you will need an environment to train the agent in, the environment serves as a simulation for the agent to navigate and train itself in. For our intentions we require an environment where can play Super Mario bros [1] with the help of python. To do that we would require an emulator for the Nintendo Entertainment System (NES), to fill this need we will use the gym-super-mario-bros [2] environment which using the package nes-py [3] in order to emulate the NES system. This environment is designed to be used with OpenAIs Gym framework, which is designed to serve as developing tool for reinforcement learning algorithms by providing basic communications between environments and learning algorithms. For learning algorithms we will use Stable-Baseline3 [4], a toolkit package for reinforcement learning algorithms in PyTorch.

1) OpenAI Gym: The Gym [5] package developed by OpenAI provides a standardized interface for developing reinforcement learning agents along with serving as a evaluation and benchmarking platform. The benchmark provided by a wide range of environments for training and testing agents where they can be compared. Because of this there are some key components that concepts that makes up the interface:

- Environment: An environment in Gym represents the task or problems that the agent will interact with, accompanied with a set of methods that allows the agent to interact with it. Environments in Gym are often very modular and follows the API, making it easy to switch or change environment.

- Gym interface: The two common components of the interface consists of the following:
  - Reset(): Resets the environment to the initial starting state and returns the initial observation. This is used all the time to revert the environment to same starting point between episodes.
  - step(action): Takes an action as input and performs one step, iteration within the environment. The return values from this methods allows us to make a decision for the following step. They are as follows: next observation, reward value, done flag(indicating if the episode is over, e.g if Mario steps on an enemy.) and finally various additional information regarding the environment such as game score.
- Action space: The action space specifies the set of possible actions that the agent can take in the environment and pass to the step function. An action space can either be discrete or continuous, where discrete cases are comparable to button presses while a continuous could be how much pressure to apply on a gas pedal.
- Wrapper: Gym allows the use of wrappers to modify or extend the behavior of an environment. Wrappers have many use cases such as preprocessing observations, modifying rewards or add any additional functionality within reason to the environment. This allows the user to customize and tailor the environment and its outputs to their needs.

2) Super Mario gym: This environment [2] for gym that makes use of the NES-py emulator [3] to simulate Super Mario Bros. and Super Mario Bros. 2 and features multiple types of environments to as seen in table I

B. *Stable-Baseline3*

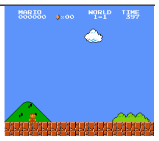
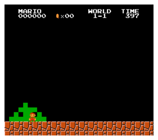
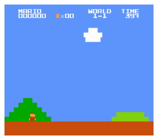
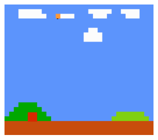
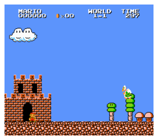
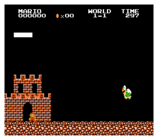
Stable-Baseline3 [4] is a python library that provides a collection of implementations of state of the art reinforcement learning algorithms, and can be compared as the scikit-learn of reinforcement learning. It is built upon OpenAIs Gym [5] and Pytorch in order to offer a a user-friendly and efficient framework for developing and training reinforcement learning agents. This provided a powerful and efficient framework, Its collection of implemented algorithms, user-friendly API, performance optimizations, and seamless integration with OpenAI Gym make it a popular choice.

C. *action space*

The action space for NES-py is a large discrete space consisting of 256 different actions, the gym-super-mario-bros [2] only makes use of 7 of these. Thus in order ease the training for agent we reducing this number by using a wrapper from. Which gives the agents the following actions to pass into the step function:

- No operation
- Right
- Right + A
- Right + B
- right + A + B
- A
- left

TABLE I
SUPER MARIO GYM ENVIRONMENTS

Environment	Game	ROM	Screenshot
SMB-v0	SMB	Standard	
SMB-v1	SMB	Downsample	
SMB-v2	SMB	Pixel	
SMB-v3	SMB	Rectangle	
SMB2-v0	SMB2	Standard	
SMB2-v1	SMB2	Downsample	

Those are all the significant moves that the agent should be able to perform. There is a case to be made whether or not to reduce the action space to:

- No operation
- Right
- Right + A

This is because these move are least varied action space set that can be used, and can be boiled down to two moves, move right, and move right and jump at the same time. The agent should technically be able to complete the level by just using these.

D. *State spaces*

The observations that the environment provides is an image with size (240, 256, 3) and it is a usable state space by itself but we can do some modifications to it in order to better our performance, both quality and time wise. To do this we are using wrappers to update the output of the step function.

- Skipframe(4)

- WarpFrame(120, 128)
- DummyVecEnv()
- VecFrameStack(4)

The Skipframe wrapper makes the environment only return every n-th frame and accumulate the reward that happens in the skipped frames. This together with the VecFrameStack that stack the latest 4 frames in a list-like fashion helps the agent get a sense of motion. The WarpFrame is resizing the output state to smaller size and removes the color channels, turning the image into a gray scale one. This significantly helps the performance of the model by reducing the data amount of data that the model is working with. DummyVecEnv is used to package the environment within a vector.

E. Reward function

The reward function of the gym-super-mario-bros environment assumes that the agents objective is to move as far right as possible, as fast as possible and without dying. To capture this the reward consists of three parts:

- Reward function: R = V + C + D
- V: velocity
$$V = X_1 - X_0 \quad (1)$$
Where X_1 is the X-position of the agent after the step and X_0 is the X-position of agent after the step. Thus we get a reward for moving towards the right and a penalty when moving to the left. Standing still will have no effect on the reward.
- C: The clock difference between frames
$$C = C_0 - C_1 \quad (2)$$
Where C_0 is the clock reading within the game before the step, and C_1 is the clock reading after the step. If there is no time difference it will have no effect on the reward, but if there is a penalty will be applied on the reward for that specific step.
- D: death penalty
  - If the agent is alive, reward it with 0 score.
  - If the agent is dead, penalize it with -15 Score

  The death penalty penalizes the agent for dying.

Further the reward clipped into a the span $[-15, 15]$ which means that the agent can never be rewarded for dying.

F. Reinforcement learning

The key concept of reinforcement learning is that there is an agent that interacts with environment of some sort. At every step the agent takes it will see a new observation of the world and decides upon the next action. For each action the agent receives a reward or penalty that informs the agent of the current state of the observation state it has received. The agents goal is to maximize this cumulative reward. In order to get a deeper understanding of the topic we will take dive in the following topics:

- States and Observations spaces
- Action Spaces
- Policies
- Trajectories
- Reward
- Reinforcement learning
- Value Functions
- Advantage functions

1) States and Observations spaces: The state s is a complete description the world within the environment while an observation o is what is being presented to an agent and may have information from s omitted. In deep reinforcement learning these both s and o are often represented as matrices, in our case for in the gym-super-mario-bros environment the states represented by RGB matrices that makes up an image for that selected state. For many of the equations following states and observations can be user interchangeably.

2) Action Spaces: As mentioned in section III-C there are different types of action spaces and within gym-super-mario-bros we are making use of discrete ones as there is a finite number of moves available for the agent, while other environment may have continuous action spaces. This is very important to keep in mind when choosing and algorithm as different algorithm can only be applied to certain action spaces.

3) Policies: Policies are the rules which the agent uses to decide upon which action to take, we will focus upon parameterized policies as those are the what we're dealing with within deep reinforcement learning. Here the outputs are computable functions that depend on parameters from the neural network which we are constantly updating and adjusting. We denote these parameters with θ and the policy with the following equation:

$$a_t \approx \pi(\cdot|s_t) \quad (3)$$

4) Trajectories: Denoted by τ, a trajectory or episodes as it is sometimes called is a sequence of stats and actions within the environment.

$$\tau = (s_0, a_0, s_1, a_1, ..., s_n, a_m) \quad (4)$$

The absolute first action s_0 are sampled from the start-state distribution and can be denoted as

$$s_0 \approx \rho_0(\cdot) \quad (5)$$

Transitions between the states s_t and s_{t+1} are entirely up to the environment and a function of the latest action passed to it. Stochastically it can be described as:

$$s_{t+1} = P(\cdot|s_t, a_t) \quad (6)$$

The action a_t is provided by the agents policy.

5) Reward: The reward function R is the most essential part of reinforcement learning as the whole concept relies upon the rewards and penalties it provides. It is a result of the current environment state and action compared to the next state of the environment:

$$r_t = R(s_t, a_t, s_{t+1}) \quad (7)$$

The agents goal is the maximize a cumulative reward of a trajectory (III-F4), which mean that we can denote it as $R(\rho)$.

The kind of return that we use in proximal policy optimization that well be described later in section III-G uses a reward function that follows:

$$R(\tau) = \sum_{k=0}^{\inf} \gamma^k r_t + k \tag{8}$$

Where the discount factor γ accounts for the fact that the agent cares more about rewards it can gather quickly compared to future rewards. This can be compared to the concept of interest where getting more money tomorrow will carry more value then getting the same amount of money a year later.

6) Reinforcement learning: The primary goal within reinforcement learning is to select or create a policy(III-F3) which maximizes the expected return when the agent acts accordingly. In order to get there we will first define the probability distributions over the trajectories. The probability of a step thus becomes:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_(t+1)|st, at)\pi(a_t|s_t) \tag{9}$$

The expected return of these probabilities denoted by $J(\pi)$ thus becomes:

$$J(\pi) = \int_\tau P(\tau|\pi)R(\tau) = \underset{T\approx\pi}{E}[R(\tau)] \tag{10}$$

The core problem in reinforcement learning, finding the optimal policy $\pi*$ is finally then expressed by:

$$\pi* = \underset{\pi}{argmax} J(\pi) \tag{11}$$

7) Value function: Knowing the value of a sate and action pair is crucial. The value for our case is the expected return when you pass the pair to the policy and act upon it. We will be looking at the On-Policy Action-value Function $Q^\pi(s, a)$ which gives the expected return for the starting sate s and action taken a, and afterwards continuing to act upon the the policy pi:

$$Q^\pi(s, a) = \underset{\tau\approx\pi}{E}[R(\tau)|S_0 = s, a_0 = a] \tag{12}$$

8) Advantage functions: In reinforcement learning it is often difficult the describe how good an action is in an absolute sense, but by using advantage functions to find the how but better it is than others on average, thus finding the relative advantage of that action. an advantage function $A^\pi(s, a)$ corresponding to the policy π describes how much better the action a is in the state s over selecting a random action to put into s, $\pi(\cdot|s)$. The function itself can be defined as:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{13}$$

As the advantage function in a deep reinforcement learning case consists of self adjusting neural network, the output can become quite noisy.

G. Proximal policy optimization

Reinforcement learning faces many challenges such as that the training data that is generated is itself dependent on its current policy since the agent is generating its own training data by its interaction with the environment. In comparison a supervised learning method would be relying on its static data set. This becomes a major cause for instability in the training process for reinforcement learning methods in conjunction with reinforcement learning high sensitivity to hyper parameter tuning. If the data is always collected under a poor policy, the agent might get stuck struggle to recover. As a counter measure to this OpenAI developed a reinforcement learning algorithm that is called Proximal Policy Optimization [6].

The main goals of Proximal Policy Optimization are ease of implementation, sample efficiency and ease of tuning. Proximal Policy Optimization is based on the method of policy gradient method, which means that does not learn from stored ”offline” data which many other popular methods does, such DQN [7]. This is very helpful when working on workstations where you have limited amounts of memory. The however means that Proximal Policy Optimization learns ”online” and does not rely on a replay buffer, but instead the encounters of the agent. As a result of this Proximal Policy Optimization is less sample efficient then DQN and other Q-learing based methods.

1) Policy gradient law: The policy optimization method starts by defining the policy gradient laws as the expectation of the log of the policy actions $\pi_t heta$ times an estimate of the advantage function as seen in equation 14.

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{14}$$

This is quite intuitive since if the advantage estimate function $\hat{A}_t$ is positive, meaning that the action the agent took in the trajectory resulted in an better then average return, what will do is to increase the probability of selecting that action again in future if we were to encounter the same state.

2) Trust region: In order to prevent the network to update its parameter outside the range of the data collection and ruining the policy, Proximal Policy Optimization uses a type of method called trust-region methods that makes sure that the algorithm will not stray to far away form the previous policy. In order to do this Proximal Policy Optimization applies a constraint on its policy gradient, which would cause some extra overhead on the calculation on our optimization process. Proximal Policy Optimization avoid this by including this constraint directly into the optimization objective

3) Central optimization objective: As the central optimization objective objective function in Proximal Policy Optimization is an expectation of trajectory batches and is calculated as a minimum between to terms. The first term represents the default objective for normal policy gradients which yields a high positive advantage over the baseline. The second term is quite similar but is clipped between the values -1 and 1.

$$\hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{15}$$

As the advantage estimate can be both positive and negative it changes the function of the min operator slightly. The Proximal Policy Optimization paper [6] demonstrates this quite well with the figures 2 and 3.

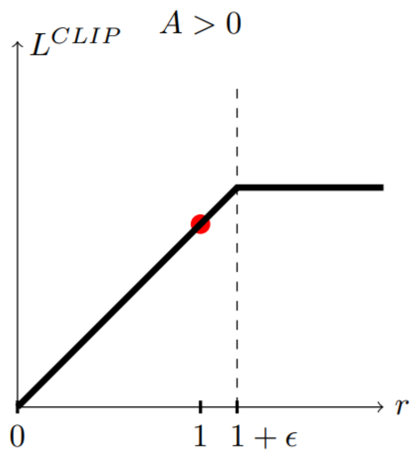


Fig. 2. All the cases when the selected action yielded a better than expected return

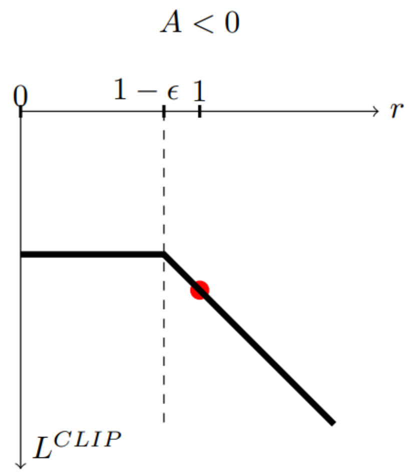


Fig. 3. All the cases when the selected action yielded a worse than expected return

In 2 we can see oh the loss function flattens when the return gets too high, this is a result of that action being more likely under the current policy than it was under the old policy. To prevent it from overdoing the action update the objective function gets clipped in order to limit the effect of the gradient update. The opposites applies if the advantage is negative, if the action is becomes less probable do not reduce the likelihood too much. This is very important since the neural network provides us with a noisy advantage function and we don't want to destroy the by doing a single faulty estimate.

4) Algorithm: With the most important parts covered we can take a look at the final algorithm [6]:

Algorithm 1 PPO, Actor-Critic Style

```
for iteration $1, 2, ...$ do
    for actor$1, 2, ..., N$ do
        Run policy $\pi_{\theta_o}ld$ in environment for T timesteps
        Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
    end for
    Optimize surrogate L wrt $\theta$, with K epochs and minibatch
    size $M \leq NT$
$\theta_{old} \leftarrow \theta$
end for
The second for-loop corresponds to were we are interacting with the environment and generating episodes sequences for which we calculate the advantage function using the fitted baseline estimate for the state values. Afterwards in the first for-loop we are going to collect all that experience and run gradient descent upon the policy network using the clipped proximal policy optimization objective. =0
```

IV. Results and discussion

A. The Super Mario gym

As presented in subsection III-A2 the gym-super-mario-bros was used as the training environment and would correspond to the selection of data set. It featured a very familiar video game environment which made it easy to relate to the agents actions and differences in the agents performance could easily be detected by looking of it was playing the game. A common and satisfying example of this was that the agent was very prone to sporadically jumping in early iterations which made it occasionally skip obstacles and avoid contact with enemies. When an agent had passed sufficiently many iterations the sporadic jumping was greatly reduced and became reserved for when the agent encountered obstacles.

The environment and was surprisingly easy to work with as it was very well integrated within the rest of the openAI gym.

B. Evaluation metrics

As the goal was to make the agent be able to reliably reach the end of the first level the absolute evaluation metric that was used was the mean reward given during the last 1000 steps that the agent took. This made very clear how far the agent would get in the level since the only source of reward was to move towards the end, by calculating the mean we also measured how reliable the agent was at reaching that goal. With some occasional visual inspection it was easy to identify what obstacles the agent faced correlated to which mean score. This is demonstrated in figure .

Occasionally one could compare to the mean episode length to when the mean reward was being reduced during training as there was often a correlation between them. If the agent would get stuck on obstacle and be unable jump across it would be visible in the mean episode length graph since the episode length would be increasing. This is demonstrated in figure .

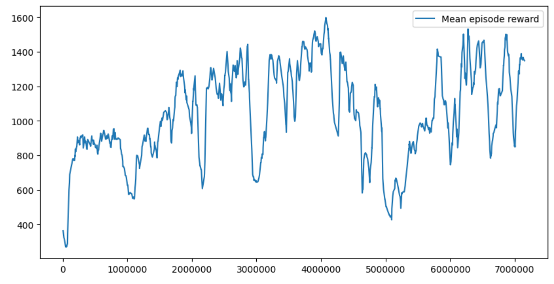


Fig. 4. The mean reward per step of

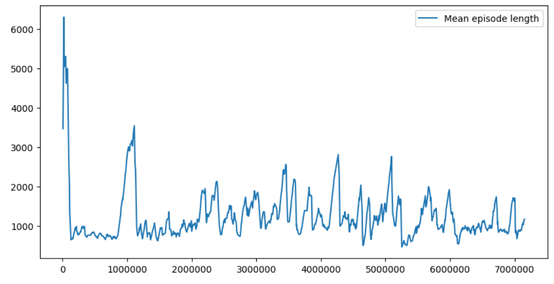


Fig. 5. Caption

C. Proximal policy optimization result

Why your method performs better or worse than others (a clear explanation and discussion is needed)? Alas we were not able to make agent complete the whole level and the agent would very rarely be able the a specific location that located halfway across the first level. This became quite visible in mean reward graph as the agents were never able to reliably pass the mean reward score of 1600, but instead tended to remain within the value span $1400 - 1600$. Inside the environment this corresponded to reaching the location shown in figure 6. This location challenges the agent with a very complex environment, not only are the multiple obstacles float in the air hindering the agent from jumping on top of the enemies to defeat them, there are also seven different enemies who are all moving to the left. This creates a difficult situation for the agent that during near to all tries managed to get into contact with.

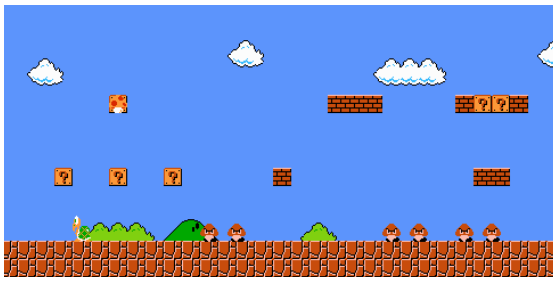


Fig. 6. The location which the agent rarely managed to move across. As can be seen it's an area of higher complexity with multiple obstacles in the air preventing the player from jumping above the seven enemies that moves towards the agent.

There were multiple agents trained with different hyper parameters in order to find a way for the agent to pass this location. The agents that were tested where the ones that achieved the highest mean reward score during the training. This because as the agent learns mean reward score can swing quite a lot, sometimes going down below a mean reward of 200 which corresponds do being defeated by the first enemy or getting stuck at first obstacle.

During the online research enthusiasts claim to have their agents complete the level after $2500000 - 5000000$ steps with similar agents setups. during our training we tested various amount of steps various parameters as seen in table II. Training the agents became very time consuming as training 1000000 steps taking approximately 1 hour and 30 minutes to take and could vary a lot depending on the Frame skip parameter since it would determine how many states the environment would skip before giving the agent an observation. As table II also shows three agents made it the $1400 - 1600$ span where they tended to stagnate relatively quickly after 2000000 steps.

TABLE II
RESULTS FROM AGENT TRAINING WITH PARAMETERS INCLUDED.

Steps	Learning rate	n$_{steps}$	Frame skip	Best mean reward
1000000	1e-5	512	2	924.56
3000000	1e-4	512	0	548.37
5000000	1e-5	512	4	1567.32
7000000	1e-5	1024	4	1623.46
10000000	1e-5	512	4	1549.81

V. Conclusion

In the end there is a lot of further experimentation that can be done with proximal policy optimization with the gym-super-mario-bros environment. We may not have gotten the desired results from the project but this could probably be achieved with prolonged training times, wrapper and parameter tuning. Other possible option to deal with this overfittness could be to introduce a check point for systems the agent which could allow to train on the segment it struggles with, this might cause the agent to ”forget” about the earlier stages of the level, but definitely something worth that could be investigated. Other deep learning methods are also possible to use, as mentioned earlier there are methods like Deep Q-Networks. During development a Double Deep Q-Network (DDQN) was implemented but ultimately abandoned in favor of proximal policy optimization. This decision was made because of the large memory requirements of DDQN and a much slower training time compared to proximal policy optimization. With more time at hand one might try to apply that type of algorithm to the problem instead.

References

[1] V. wikipedia contributors, “Super mario bros.” wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Super_Mario_Bros.
[2] C. Kauten, “Super Mario Bros for OpenAI Gym,” GitHub, 2018. [Online]. Available: https://github.com/Kautenja/gym-super-mario-bros

[3] G. user Kautenja, "Nes-py emulator." [Online]. Available: https://github.com/Kautenja/nes-py

[4] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

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