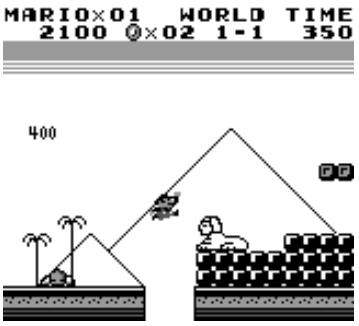
Train RL Mario AGENT

q_play

Playing Super Mario Land with Reinforcement Learning 🤖

Advances in AI - WS 2023/24 HSD

Vincent Bohm
Matriculation number: 897512



GitHub: [l4nz8/q_play]

Contents:

1. Introduction

1.1 Project description:

My project "Train RL Mario AGENT" (q_play) is a reinforcement learning (RL) model for automating and optimizing the game strategy in the video game "Super Mario Land" on the GameBoy emulator platform PyBoy. The aim of this project was to develop and implement an intelligent agent that can navigate through the game, overcome obstacles and enemies and complete the level as quickly as possible. The agent should continuously learn from its actions and their results. To achieve this, the agent uses a Double Deep Q-Network (DDQN) architecture, a further development of the Deep Q-Network (DQN), to achieve stable and efficient learning performance.

1.2 Motivation:

My motivation behind choosing this project was my great interest in the functioning, structure and operation of self-learning and optimizing networks. When, unfortunately, a video of an RL-Proximal-Policy-Optimization (PPO) model created on the PyBoy, which played Pokemon Red on the GameBoy platform, was also suggested to me on Youtube, I made my project choice to create an RL model on the Gameboy myself. After choosing the AI model and platform, all I needed was a learning environment for the agent. Since "Super Mario Land" has a complex and varied level design and a linear game structure, I finally decided on this game. It offers an ideal platform for researching and demonstrating the abilities of RL agents, which is why it has often been used as a model environment for RL models.

2. Basics and state-of-the-art

2.1 Reinforcement Learning (RL) and Deep Q-Networks (DQN):

“

Reinforcement learning (RL) is an area of machine learning in which an agent aims to independently develop an optimal policy (behavioral strategy). Agent systems try to maximize their reward for the right action through trial and error in an environment. The basic idea is that an agent learns its actions by interacting with the environment and observing the results, which are determined by a reward value. This enables it to develop situational strategies that aim to maximize the reward value. [1]

“

“

Deep Q-Networks (DQN) combine classical Q-learning with the power of deep neural networks, making it possible to solve complex problems in large state spaces. By learning an approximation of the Q-function that represents the expected value of an action in a given state, DQNs can operate in challenging environments. This method has been successfully applied in the domain of Atari video games, where DQNs showed impressive capabilities by not only equaling, but in some cases outperforming human players in several games. These breakthroughs demonstrate the enormous potential of DQNs in practice. [2]

“ *ChatGPT

2.2 State-of-the-Art in RL und Anwendung auf Videospiele

“

Since the introduction of DQNs, the field of RL has evolved rapidly. New architectures such as Double DQN, Dueling DQN, and A3C (Asynchronous Advantage Actor-Critic) have further improved the performance and

stability of RL agents. These techniques address various challenges of DQNs, such as the overestimation of Q-values or the inefficiency in the use of environmental information.
“ *ChatGPT

The application of RL to video games not only serves as a benchmark for the performance of such algorithms, but also as a research environment for the further development of RL. Platformer games such as "Super Mario" are particularly interesting as they require a combination of spatial navigation, timing and strategy. Research in this area has shown that RL agents are able to learn complex strategies and achieve high performance in these games.

2.3 Double Deep Q-Network architecture:

“
Double Deep Q-Networks (DDQN) build on the foundations of Deep Q-Networks (DQN) and aim to solve a central problem of DQNs - the systematic overestimation of Q-values. By introducing a methodological separation between the selection and evaluation of actions, DDQNs minimize the risk of overestimation. This is achieved by using two separate networks: an online network that is responsible for selecting actions and a target network that evaluates these actions. This architectural innovation helps to stabilize the learning process and accelerates convergence over traditional DQN approaches. [3]
“ *ChatGPT

Although there are now more modern RL methods and situationally better models such as Proximal Policy Optimization Algorithms (PPO) [4], DDQNs are still a very effective and commonly used architecture for agent networks. They are particularly well suited for tasks with discrete actions such as steering through the Super Mario Land level, as only a clear movement through the game is required.

3. Technology and methodology

3.1 PyBoy emulator:

To be able to train an RL model on the game Super Mario Land, you need an interface to the game, e.g. in the form of an emulator that can create this environment. The PyBoy emulator is an open source platform that makes it possible to run GameBoy games. In addition, the platform offers a Python API, which makes it much easier to read the data, which is crucial for the development of an interactive learning environment. In addition to the ability to manipulate game states and extract visual outputs, which is essential for training the RL agent, the agent can receive direct feedback on its actions and adjust its strategy to maximize in-game rewards. Additionally, the emulator offers other features such as screenshot, video recording, pause and game state saving.

3.2. Argparse:

The argument parser is used to process commands in the console and execute them in the script. I have implemented various functions to create an efficient and interactive platform. All arguments can be read in the script using `--help`. They offer the possibility to select worldlevel, activate game or training mode, load game states, activate the debug function, set hyperparameters etc. These arguments enable flexible configuration of the AI agent and the training process to take account of different scenarios and requirements.

3.3 Preprocessing techniques:

For the training process, I first prepared the output image of the game environment for the network using various so-called "wrapping processes" (preprocessing techniques):

3.3.1 Frame Skipping:

By skipping frames, the training time can be significantly reduced. This reduces the complexity of the state space with which the agent has to interact, which leads to faster decision-making.

3.3.2 Gray scaling:

Since three color channels do not have important additional information for the learning process, they are reduced to one grayscale channel. Although the game is already rendered in grayscale, the emulator still returns 3 color channels with the same black and white values. In order to reduce the dimensionality, the image had to be reduced to one color channel.

3.3.3 Resize observation:

The output image is scaled down to a smaller format for faster processing of the information. A higher resolution is not absolutely necessary to pass on the same image information.

3.3.4 Frame Stacking:

To give the agent an understanding of temporal sequences, several consecutive frames are combined into a single input stack. This enables the agent to understand movements in the game and thus make better decisions.

These pre-processing techniques are important as they give the network the opportunity to better understand the learning environment without overwhelming it. They therefore help to increase the effectiveness of the learning process.

3.4 Action space:

The action space describes the set of all possible actions that an agent can perform in its environment in order to fulfill a certain task. The larger the action space, the more complex and computationally intensive the model becomes, as more possible actions have to be evaluated. In order not to go beyond the scope of the project, I have therefore reduced the action space to what is necessary to successfully complete a level. In this project, the action space only includes forward movement, backward movement, jumping and combinations of these actions.

Possible actions: [['PRESS_ARROW_RIGHT'], ['PRESS_BUTTON_A'], ['PRESS_ARROW_LEFT'], ['PRESS_ARROW_RIGHT', 'PRESS_BUTTON_A'], ['PRESS_BUTTON_A', 'PRESS_ARROW_LEFT']]

This reduction helps to minimize the complexity and required computing power and still allow the agent to successfully complete the level.

4. Implementation of the RL agent

4.1 Network architecture:

The DDQN agent uses a Convolutional Neural Network (CNN), which is necessary for processing the pixel input of the game "Super Mario Land". The input for the network is a stack of four pre-processed frames, each scaled to a resolution of 84x84 pixels in grayscale. The architecture consists of three convolutional layers, followed by two fully connected layers.

```
        # Define the CNN architecture
        return nn.Sequential(
            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4, dtype=torch.float32),
            nn.BatchNorm2d(32), # Mitigate the problem of internal convariate shift
            nn.LeakyReLU(), # Avoid the issue of "dead neurons"
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, dtype=torch.float32),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, dtype=torch.float32),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.Flatten(), # Flatten the output for the fully connected layers
            nn.Linear(3136, 512, dtype=torch.float32), # 3136 = number of features from the last conv layer
            nn.LeakyReLU(),
            nn.Linear(512, output_dim, dtype=torch.float32) # Output layer
        )
```

- **Convolution-Layer (Conv2d):** Processes input images by extracting spatial features. The size of the filter with Stride determines how the input is reduced and which features are emphasized.
- **Batch-Normalization (BatchNorm2d):** Normalizes features, stabilizing and accelerating training. It minimizes the problem of internal covariant shift.
- **LeakyReLU-Activation:** Adds nonlinearity to avoid the "dead neuron" problem.
- **Flatten:** Converts the multidimensional feature maps into a one-dimensional vector.
- **FullyConnected-Layer (Linear):** Maps high-dimensional features to a lower-dimensional output space, which represents the possible actions.

After complete processing, the network returns a number of 512 possible situation-related actions as output.

4.2 Hyperparameters:

The most important hyperparameters for the DDQN agent include the learning rate, gamma value, exploration rate, decay batch size, replay memory size and the synchronization of the two networks.

exploration_rate = 1
exploration_rate_decay = 0.99999975
exploration_rate_min = 0.1
batch_size = 32
storage=LazyMemmapStorage(100000)
learning_rate=0.00025
gamma = 0.9
self.sync_every = 10000

- The agent initially uses an exploration rate of 1 to explore the environment by frequently performing random actions until it reaches a minimum value of 0.1.
- The exploration is decremented over time by a factor of 0.99999975 in order to execute actions with the highest Q value more frequently.
- The replay memory specifies the size of the storage of past experiences in an episode, with a batch size of 32 for training.
- The learning rate for the optimizer is 0.00025 with a gamma value of 0.9 for future rewards, which is used by the "StepLR" scheduler.

It is possible to select the "Cyclic" scheduler instead of the "StepLR", which has a step width of 2000 between the minimum (0.0001) and maximum (0.001) values.

4.3 Interaction with the environment:

The agent interacts with the game environment using a class specially modified for Super Mario Land, which is based on the OpenAI Gym interface implemented by PyBoy. This enables the agent to perform actions and receive observations and rewards back from the environment.

The `CustomPyBoyGym` class handles details of the game environment, such as loading the game, setting the level/world, and extracting relevant game information for the agent.

4.4 Components of the RL agent:

4.4.1 Action Selection:

Action selection is based on an agent epsilon-greedy policy. This means that the agent tends to choose actions that have the highest Q value according to the current model. However, a random action is chosen with a probability of Epsilon (exploration rate) to encourage exploration of the environment.

4.4.2 Reward System:

The reward system is based on four components: faster movement through the level, how far the agent has progressed in the level, a negative reward for deaths and a positive reward for completing the level.

The total reward is the sum of these four components: Time Difference, Movement Difference, Life Loss Penalty and Level Progression Reward. This system aims to encourage fast progression, survival and reaching new levels or worlds.

```
66    # Reward components
67    clock = current_mario.time_left - prevGameState.time_left
68    movement = current_mario.real_x_pos - prevGameState.real_x_pos
69    death = -15*(current_mario.lives_left - prevGameState.lives_left)
70    levelReward = 15*max((current_mario.world[0] - prevGameState.world[0]), (current_mario.world[1] - prevGameState.world[1]))
71
72    reward = clock + death + movement + levelReward
73    return reward
```

4.4.3 Memory Buffer:

The replay memory buffer stores the agent's experiences in the form of state, action, reward, next state and the end-of-episode indicator. These experiences are then used to improve the stability of the learning process and reduce the correlations between successive learning steps.

5. Experiments and evaluation

5.1 Experimental setup:

To evaluate the performance of the Mario-RL agent network, I used two different networks and compared their learning performance on one level. Two model sizes with two different learning optimizer schedulers were used:

1. *"StepLR" optimizer scheduler with a network size of 3 convolution layers and without pedding.*
2. *"Cyclic" optimizer scheduler with a network size of 4 convolution layers and a pedding of 1 on each of the 2 to 4 layers.*

I wanted to see which network performs better and whether the network size and pedding lead to significant differences.

Both models were run over a period of approximately 48 hours with a total of 6 million actions performed. The models themselves were all tested on the first level of the first world of the game (W1 / L1) and their learning progress was compared at the end. Both models were trained with the implementation of Cuda on an NVIDIA GeForce RTX 2080 Ti with 19 GB GPU memory.

5.2 Training results and discussion:

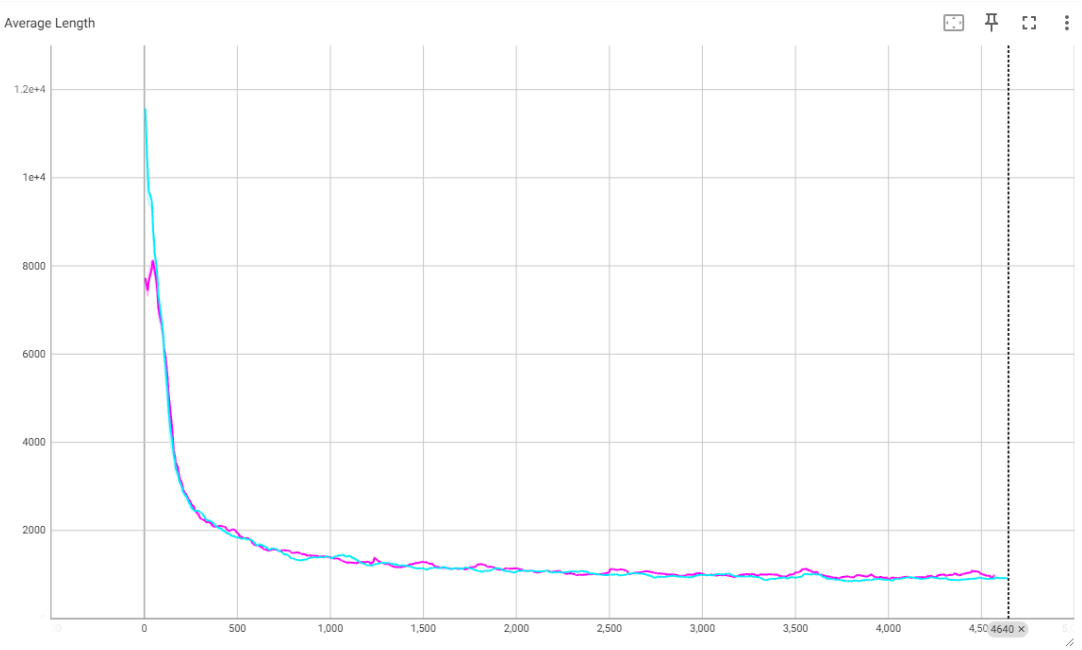
In order to observe the differences in the learning behavior of the two models, 4 metrics (Average Length, Average Loss, Average Q-Value, Average Reward) were logged during the learning process:

“

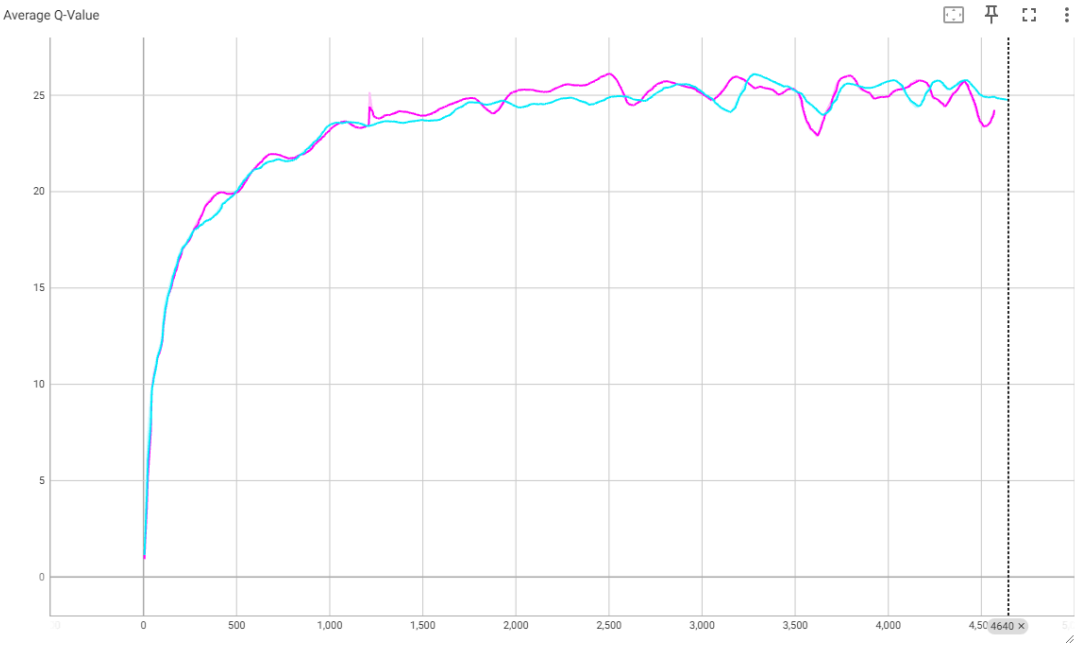
- **Average Length:** This refers to the average duration of episodes, measured in time increments or actions, before an episode ends (e.g. by reaching the goal or failing the task). Longer average episode lengths indicate that the agent is in an episode for longer, but this is only good depending on the situation.
- **Average Loss:** The average loss indicates how well the model predicts the expected rewards. A lower loss value indicates that the agent's predictions of future rewards are becoming more accurate, which indicates more effective learning.
- **Average Q-Value:** The average Q value reflects the expected reward that the agent estimates for its actions in a particular state. Higher average Q values mean that the agent expects higher rewards for its actions, which can be an indicator of learning more effective strategies.
- **Average Reward:**The average reward value shows how much reward the agent receives on average per episode. An increase in the average reward over time is a direct indicator that the agent is learning to optimize its actions in order to achieve higher rewards.

“ *ChatGPT

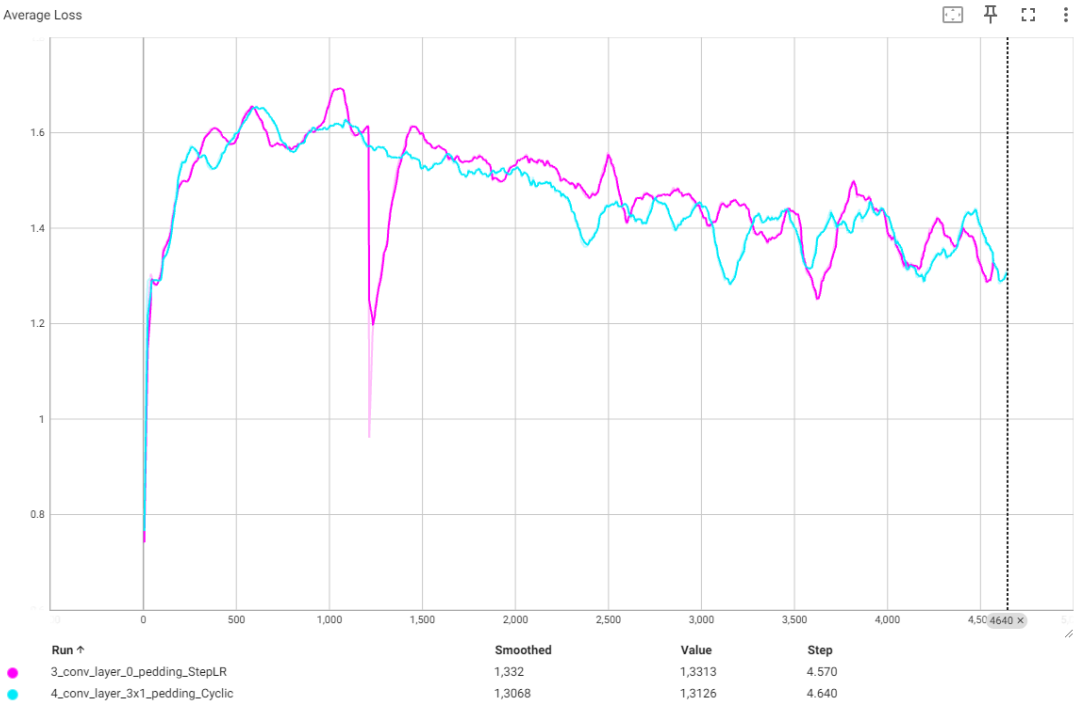
5.3 Bewertung der Leistung:

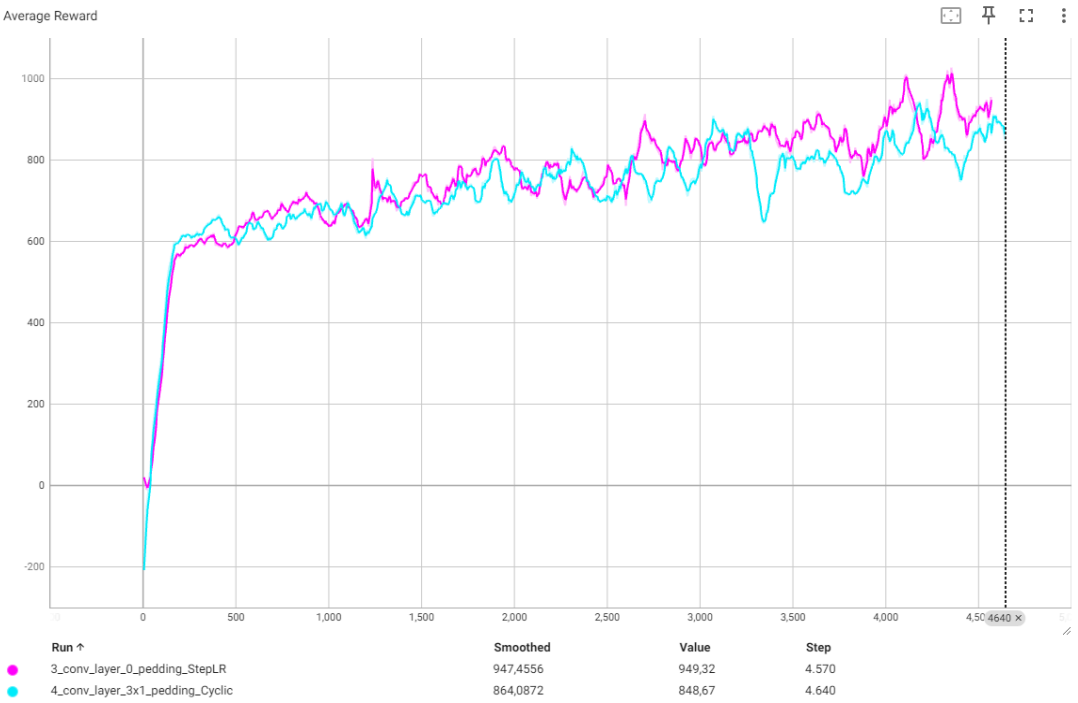


Run	Smoothed	Value	Step
3_conv_layer_0_pedding_StepLR	983,2201	983,12	4.570
4_conv_layer_3x1_pedding_Cyclic	912,3663	906,88	4.640



Run	Smoothed	Value	Step
3_conv_layer_0_pedding_StepLR	24,2308	24,2569	4.570
4_conv_layer_3x1_pedding_Cyclic	24,7665	24,7596	4.640





The two models initially show identical learning behavior with minimal deviations in loss and reward, which may have been caused by the random factor Epsilon. Nevertheless, the smaller model (1) tends to show a slightly higher reward at the end, but a lower Q-value than the larger model (2). On the whole, however, no great variance can be seen between the two networks over the 48-hour period, apart from a strong outlier in the loss at position 1250, which can be attributed to an interruption in training.

5.4 Summary of the results:

Overall, there are no major differences between Model 1 and Model 2 at one level. The learning behavior of both is roughly the same and does not differ noticeably. The only difference that stands out is the speed and

resource requirements of the two models. In comparison, the smaller model (1) uses 27% less power than the larger model (2).
However, there are other hyperparameters that have not been changed, which can also have a strong impact on the performance of the model, such as the size of the batch size.

6. Summary and outlook:

6.1 Summary of the project:

The project "Train RL Mario Agent" (q_play) demonstrates the application of a Double Deep Q-Network (DDQN) on the video game "Super Mario Land" using the PyBoy emulator. Overall, I have to say that the development process taught me a lot about building and applying such RL methods. Furthermore, by using a Python-based emulator, I was also able to understand working with different platforms. I particularly enjoyed experimenting with the different networks and working with the game environment.

One big difficulty I had at the beginning was processing the observation space of the game environment for the CNN, but I was able to solve most of this with help during the internship. The second obstacle was the Cuda implementation of my home computer, which I really wanted to use for training, but which I also managed to solve. After these two hurdles, there were hardly any problems worth mentioning that led to complex errors that I couldn't solve myself through research.

6.2 Improvements:

Although the project has become very good and extensive, I have to say that it still leaves some room for further development. Nevertheless, it is very efficient and offers the possibility of training far better models with the right hardware. In addition, the RL agent was only trained at the first level. This means that there is still plenty of scope to allow the agent to continue learning across different game levels and situations. The agent's action range, which is limited to forward, backward and jumping, can potentially be expanded with two additional actions. It is also possible to extend the project so that several agents can train the same model simultaneously in separate environments.

6.3 Outlook:

“
Future research could focus on further increasing the learning speed and efficiency of such agents through improved algorithms and network architectures. One exciting field is the investigation of cooperative and competitive multi-agent learning in multiplayer video games, which opens up new dimensions of interaction and strategic thinking.

There is also great interest in transferring the techniques and methodologies developed to other areas of application outside the games industry, for example to problems of robotics, autonomous navigation and complex decision-making.

The field of reinforcement learning (RL) is an exciting area where research continues apace. It opens the door to exciting new research directions and practical applications that have the potential to fundamentally expand our understanding and use of artificial intelligence.
“ *ChatGPT

7. Appendix Source:

A. Source code directory:

- **main.py:** Main script for initializing and executing the training or game mode of the reinforcement learning (RL) agent.
- Includes setup for the command line arguments (Argparse), initialization of the PyBoy emulator, configuration of the learning environment and the training or game logic.
- **gym_env.py:** Defines the `MarioGymAI` class, which serves as an interface between the PyBoy emulator and the RL agent.
- Responsible for managing the game state and executing actions.
- **deep_q.py:**Implementation of the Double Deep Q-Network (DDQN) architecture,
- includes the definition of the neural network and logic.
- **wrapper.py:** Contains various wrapper classes for adapting the observations from the environment.
- **console_ui.py:** Auxiliary class for user interactions, for loading models and displaying available checkpoints.
- **pyboy_gym.py:** Extension of the `PyBoyGymEnv` for customized interactions with the game "Super Mario Land".
- **qnet_interface.py:** Customization of the action space and reward calculations specific to Super Mario Land.

B. References

***** Marked sections were summarized, generated or quoted by ChatGPT.

1. C.J.C.H. Watkins and P. Dayan, "Q-Learning," in Machine Learning, vol. 8, no. 3-4, pp. 279-292, 1992.
2. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning" arXiv:1312.5602, Dec. 2013.
3. H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," arXiv:1509.06461, Sept. 2015.
4. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms" arXiv:1707.06347, Jul. 2017