Reinforcement Learning With PyBoy

Pedro Alves[†] and Andreas Høiberg Eike[†]

Faculty of Engineering and Science, University of Agder,
Grimstad, 4879, Agder, Norway.

Contributing authors: pedron18@uia.no; andree18@uia.no;
[†]These authors contributed equally to this work.

Abstract

This report describes our work in training a reinforcement learning agent to play Super Mario Land and Kirby's Dream Land using PyBoy, a GameBoy emulator for Python. We used the Double Deep Q-Networks algorithm for training the agent to pick the action given a game state. Reward functions was implemented by reading the GameBoy memory through PyBoy, taking game score, lives left, player x-position and time left into account. From our results we can see that the agent performs well in both games, Mario showed promising results, but the Mario character cannot fly like Kirby, making the environment more difficult to navigate through. the Mario Agent performed well in the first level, making it past all of the enemies, but getting stuck just before finishing the level. For Kirby's Dream Land, the agent was able to complete the first part of level 1 and reach the boss, but ultimately failing to beat it. Reward value over time showed steady increases, and with more time and more learning results could be improved further.

Keywords: Reinforcement Learning, DDQN, PyTorch, OpenAI Gym, PyBoy, GameBoy

1 Introduction

This report describes our process of creating an Reinforcement Learning agent that can play platformers, mainly Super Mario Land and Kirby's Dream Land for the GameBoy. We want our agent to focus on finishing each level as fast as possible, not necessarily with the most score. Therefore, we must create reward functions that encourages forward movement and penalizes backwards

movement and standing still. We will first look at some related work and their results, before we introduce our solution. Finally, we will discuss our results.

1.1 Background

Reinforcement Learning (RL) is the branch of machine learning that considers how agents can learn the optimal behaviour in a given environment. The optimal behaviour is decided by specifying a reward function, and allowing the agent to explore the environment with the goal of maximizing the reward function. Furthermore, the agent calculates which action to take at any given time using both techniques called exploration and exploitation; using either randomness or previous knowledge as input.

As such, an interesting environment for learning and understanding reinforcement learning is video games. Most video games include a player character that navigates in an environment to achieve some goal, which is quite similar to the description of how reinforcement learning is applied. We decided to focus on platformer video games on the GameBoy console, as they have a limited discrete action space for quicker learning, while actions may change both the reward and the state of the environment. Also in general, the goal of a platformer is to move to the right, while avoiding enemies and traps, which makes it easy to define a reward function.

1.1.1 PyBoy

PyBoy is a python based emulator for GameBoy games [1]. The emulator was created as a student project by Ynddal, Ynddal and Hansen [2], and because the core emulator is initialized as a python object, it can be manipulated by external scripts. In this case, external scripts would be able to send inputs to the emulator, read memory and screen information and load in games using GameBoy ROM files. As such, PyBoy includes all the tools needed to train a reinforcement learning agent.

1.1.2 PyTorch

PyTorch is a machine learning library for Python, which has support for various activation functions, neural network architectures and loss-functions [3]. We previously used PyTorch for projects involving neural networks, such as Generative Adversarial Networks (GAN), Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN). PyTorch also provides various tutorials for reinforcement learning, and we used their tutorial by Feng et.al as a starting point for our project [4]. This tutorial shows how to train a reinforcement learning agent to play Super Mario Bros., using a NES emulator, PyTorch and OpenAI gym.

1.2 Theory

To find the optimal action for each state the algorithm Double Deep Q-Network(DDQN) [5] was used. This algorithm is based on the Deep Q-Network(DQN) [6] which is in turn based on the Q learning algorithm [7]. The following sections will give an overview of the Q-learning algorithm, the DQN algorithm and finally the DDQN algorithm. To explain these algorithms some variables are used for simplicity, these can be seen in the table 1.2:

α	action
s	state
s_t	state at time t
s_{t+1}, s'	next state
r	reward
$Q[s, \alpha]$	the exact reward from an action in a state
$Q^*[s, \alpha]$	expected reward from an action in a state based on possible future rewards
θ, θ_i	online network, online network at iteration i
θ'	target network

Table 1 RL important variables

1.2.1 Q-learning algorithm

In Q-learning the 'Q' stands for quality, which represents how good a certain action is. The fitness of an action is decided by a reward function, the higher the output of the reward function the better the action is. The agent maintains a table of $Q[s, \alpha]$. In this project one element in the table contains:

$$(s_t : 2dArray, s_{t+1} : 2dArray, a_t : int, r_t : int, done : bool)$$

This table must be filled by random actions at the start as the agent explores the environment. The more entries in the table the less random the agent can become by using the actions with the higher Q value. Once the table has some entries, the agent can start to create an estimate $Q^*[s, \alpha]$ of the future rewards by looking at the table. The $Q[s, \alpha]$ value can be updated in the table by using the Bellman equation 1.

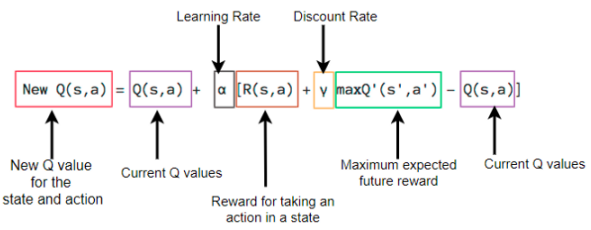


Fig. 1 Bellman equation. Equation for updating $Q[s, \alpha]$ in the table. [7]

1.2.2 Deep Q-Network (DQN)

In the Deep Q-Network algorithm, instead of updating the $Q[s, \alpha]$ values in the table, we update neural network weights. In Q-learning a table was filled with tuples (state, action, $Q[s, \alpha]$), while in DQN a neural network is used that gets state s as input and outputs $Q[s, \alpha]$ for all possible actions α. Then using the Bellman Equation we update the neural network weights such that the neural networks became better at predicting $Q[s, \alpha]$.

DQN has 2 main additions, the first addition is experience replay, which is the storing and replaying of previous game states which the agent learns from. The second addition is that it uses 2 neural networks called 'online' and 'target' where the 'target' network copies the parameters of the 'online' network every N steps. This is done to calculate the loss of the network which is used to update the network weights by using experience replay. The loss function $L_i(\theta_i)$ can be seen in equation (1.2.2). This is the equivalent of the equation in figure 1 for Q-Learning but instead of updating values in a table using $Q^*[s, \alpha]$, we use it in the loss function which is used to update the weights in a neural network.

$$
\begin{aligned}
L_i(\theta_i) &= [Q^*[s, \alpha] - Q(s, \alpha;\ \theta)]^2 \\
where \quad Q^*[s, \alpha] &= (r + \gamma max_{\alpha'} Q(s', \alpha';\ \theta'))
\end{aligned}
$$

(1.2.2) Loss equation for DQN.

1.2.3 Double Deep Q-Network (DDQN)

DDQN was created to solve a Q^* values overestimation problem from the DQN algorithm. This is because in DQN the Q^* equation always takes the highest predicted value by using $max_{\alpha'} Q(s', \alpha';\ \theta')$. In DDQN first the ighest next action is choosen by the online network $argmax_{\alpha'} Q(s', \alpha', \theta)$ and then the Q value is predicted by using the target network in $Q(s', argmax_{\alpha'} Q(s', \alpha', \theta), \theta')$.

$$
\begin{aligned}
L_i(\theta_i) &= [Q^*[s, \alpha] - Q(s, \alpha;\ \theta)]^2 \\
where \quad Q^*[s, \alpha] &= (r + \gamma Q(s', argmax_{\alpha'} Q(s', \alpha', \theta), \theta')
\end{aligned}
$$

(1.2.3) Loss equation for DDQN.

1.3 Related Work

This section presents some related work into reinforcement learning for video games.

1.3.1 Deep Q-learning for Atari 2600 games

DeepMind pioneered the first deep learning model for Atari 2600 games in 2013 [8]. Their algorithm was used on 7 Atari 2600 games, where it outperformed previous work in 6 of the games, while outperforming human ability for three of the games. They used a convolutional neural network model using the novel model-free deep Q-learning algorithm, in which the agent learns from previous experience by maintaining a replay memory, storing observations, action and reward for each experience.

What is interesting with Deepminds research is that their model works with different Atari 2600 game environments without game specific adjustments. The algorithm knows only the action space, the observed game screen, and the current game score. The results for DeepMind and other algorithms are shown in figure 2. As shown, DeepMinds DQN algorithm on average outperforms the other implementations, while also outperforming human performance in Breakout, Enduro and Pong in a best case scenario.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Fig. 2 A selection of different algorithms tested on 7 different Atari 2600 games [8]. The results are measured in reward. The row "DQN" and "DQN Best" is the DeepMind DQN algorithm.

1.3.2 DreamerV2

DreamerV2 is a model-based algorithm introduced by Hafner et. al. [9] in 2020. Because it is model based, it learns to predict the outcome of actions over time, where as model-free implementations learns by trial and error to pick the best action based on the reward function. DreamerV2 has outperformed human world records in Atari 2600 games, as well as outperforming model-free algorithms, as shown in figure 3. The score here is called gamer normalized median, and essentially, this metric shows how many times better an algorithm is compared to a human professional gamer. DreamerV2 uses multiple neural networks

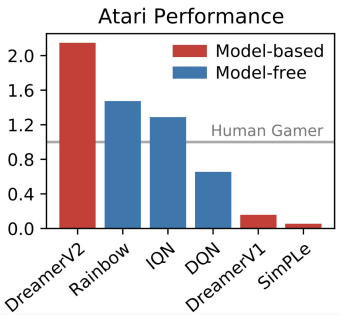


Fig. 3 Graph showing performance of DreamerV2 and other algorithms. Score is gamer normalized median [9].

to predict images, rewards and discount factors [9]. Discount factors is a parameter that says if the current episode is done or still ongoing. DreamerV2 also predicts long-term behavior using actor and critic networks. The actor chooses actions, and the critic calculates the future rewards.

2 Method

This section describes the methods we used to implement a reinforcement learning agent, to play Super Mario Land and Kirby's Dream Land. We also describe the reward function for each game.

2.1 Model used

The neural network model we choose for the input of 4x20x16 can be seen in figure 4. The reason for this input dimentions will be explained in section 2.2.1. The inputs then go trough 3 convulutional layers with kernel sizes 4x4,4x4,2x2, with ReLU as the activation function in between each convulution. Then the output is flattened and goes trough 2 linear layers to give an output of the dynamic size of number of possible actions.

```
nn.Conv2d(in_channels=c, out_channels=16, kernel_size=4, stride=1),
nn.ReLU(),
nn.Conv2d(in_channels=16, out_channels=32, kernel_size=4, stride=1),
nn.ReLU(),
nn.Conv2d(in_channels=32, out_channels=32, kernel_size=2, stride=1),
nn.ReLU(),
nn.Flatten(),
nn.Linear(3744, 512),
nn.ReLU(),
nn.Linear(512, output_dim)
```

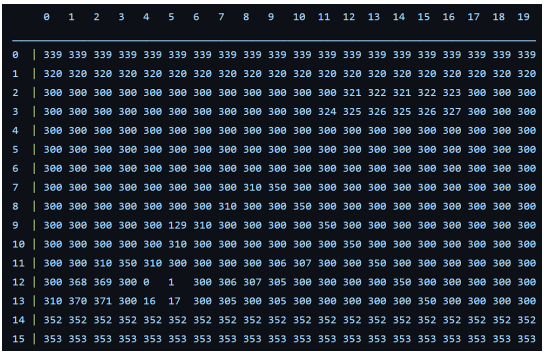
Fig. 4 Model used with input of size 4x20x16.

2.2 OpenAI Gym Enviroment & PyBoy Game Wrappers

OpenAI Gym is a library to create environments for RL training. PyBoy comes with an built in OpenAI Gym for all their games, and this is what was used. The built in Gym requires a game wrapper for each game, this game wrapper takes care of reading values from RAM memory such as score, level, x-position and so on. Super Mario Land and Kirby's Dream Land already had game wrappers implemented by the creators of PyBoy and therefor these were the games that we choose to start with.

2.2.1 Observation & Observation Wrappers

The original size of the game images is 160x144x3 which is the pixel representation of the game. But in the PyBoy game wrappers the creators of PyBoy had implemented a 2d memory representation of the game of size 20x16. This representation uses the ID's for the sprites to represent the current game state, this can be seen in figure 5.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	339	339	339	339	339	339	339	339	339	339	339	339	339	339	339	339	339	339	339	339
1	320	320	320	320	320	320	320	320	320	320	320	320	320	320	320	320	320	320	320	320
2	300	300	300	300	300	300	300	300	300	300	300	300	321	322	321	322	323	300	300	300
3	300	300	300	300	300	300	300	300	300	300	300	324	325	326	325	326	327	300	300	300
4	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300
5	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300
6	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300
7	300	300	300	300	300	300	300	300	310	350	300	300	300	300	300	300	300	300	300	300
8	300	300	300	300	300	300	300	310	300	300	350	300	300	300	300	300	300	300	300	300
9	300	300	300	300	300	129	310	300	300	300	300	350	300	300	300	300	300	300	300	300
10	300	300	300	300	300	310	300	300	300	300	300	300	350	300	300	300	300	300	300	300
11	300	300	310	350	310	300	300	300	300	306	307	300	300	350	300	300	300	300	300	300
12	300	368	369	300	0	1	300	306	307	305	300	300	300	300	350	300	300	300	300	300
13	310	370	371	300	16	17	300	305	300	305	300	300	300	300	300	350	300	300	300	300
14	352	352	352	352	352	352	352	352	352	352	352	352	352	352	352	352	352	352	352	352
15	353	353	353	353	353	353	353	353	353	353	353	353	353	353	353	353	353	353	353	353

Fig. 5 Memory view of the Super Mario Land game. In this example, Mario is '0', '1', '16' and '17'. He is standing on the ground which is '352' and '353'.

To increase the learning performance we used 3 observation wrappers:

SkipFrame(skip=4) Used to repeat the same action for N frames. This is such that our model does an action for every 4 frames instead of making an action for every frame which is unnecessary.
NormalizeObservation() Transforms the observation values into a range [-1, 1] to increase neural network performance.
FrameStack(stack=4) Stacks N frames which are fed as an input to our neural network, this is because 1 frame does not contain much information.

2.2.2 Action spaces

PyBoy has a toggle action space system, meaning that for every button there is a release button. This means that if the age wants to jump, the agent has to send JUMP_PRESS and then JUMP_RELEASE otherwise JUMP_PRESS wont work again. In the tutorial we started with mentioned in section 1.1.2, the action space consisted of simulations action such as [RIGHT], [RIGHT & JUMP]. Because of this we decided to implement simultaneous actions in PyBoy.

Super Mario Land
The 3 initial basic buttons are: [press_right, press_jump, press_left]. But we combine these into sets of 2 to provide simultaneous actions giving us $\binom{3}{2} = 3$ simultaneous actions. We remove [press_right, press_left] since it is unnecessary and we add the initial buttons, this gives us an action space of size $(3-1)+3=5$:

[press_right, press_jump, press_left, press_right & press_jump, press_jump & press_left]

Kirby's Dream Land
The 6 initial basic buttons are: [press_right, press_up, press_down, press_left,

press_a, press_b]. Using the same method as above but without removing any actions, this gives us an action space of size $\binom{6}{2} + 6 = 21$.

2.3 Super Mario Land

Super Mario Land is a 2D platformer game, consisting of 12 levels. The player character Mario has to move to the right to progress each level, in order to reach the door at the end of the level. The player character may also encounter a boss and the end of levels, where the player has to jump over the boss and reach a flag to defeat it. The player cannot progress the game until the boss is defeated. Mario can also defeat enemies or pick up items to increase the game score. However, since these pickups and defeating enemies is not required to finish the level, our reward function will not take this into consideration.

2.3.1 Reward function

The point of Super Mario Land was to beat it as fast as possible, therefore the reward function was created based on this. The final reward function can be seen in equation 2.3.1. Here x = x position of the agent, such that moving to the right will increase the x position and moving to the left will decrease it. The variable t = time change such that the more time passes the higher the value is, $\Delta lives$ is the change in lives such that if the AI loves lives it will receive a negative reward. And finally $\max(\Delta world, \Delta level)$ gives a reward based on if the agent went up a level or went up a world.

$$reward = \Delta x + \max(\Delta world, \Delta level) * 15 - \Delta t - \Delta lives * 15$$

(2.3.1) Reward function for Super Mario Land.

2.4 Kirby's Dream Land

Kirby's Dream Land is a 2D platformer game like Super Mario Land. However, it has 5 levels and you play as Kirby, which can perform different actions than Mario. Kirby can fly, inhale and shoot enemies and items to use them as attacks. Levels in Kirby's Dream Land are longer and more complex; levels require both horizontal and vertical movement to reach the end of the level. However, in the first part of level 1 you only need to move to the right to the reach the warpstar, which transports you to the first boss of level 1. This transition is shown in figure 6. Because of high level complexity and limited game information, the focus in this report is the first part of level 1 and defeating the first boss. This will in turn decrease training time as episodes are completed faster.

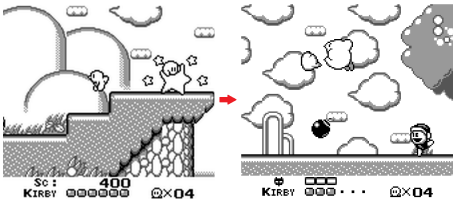


Fig. 6 Kirby has reached the Warpstar (left) and is transported to the bossfight (right).

2.4.1 Reward function

Unlike Super Mario Land, Kirby's Dream Land does not have information about the players current level progress or a game timer. Algorithm 1 shows pseudocode for the Kirby's Dream Land reward function. The function did not fit as a mathematical expression because of limited game information. The values that are being checked are stored in the games ROM memory, and is read on each calculation of the reward. This was done using PyBoy's game wrapper for Kirby's Dream Land [10], and by using a RAM map, shows how game information is stored in memory [11].

Algorithm 1 Kirby Reward Function

```
if killed_boss then
    return 10000
if boss_lost_health then
    return 1000
if kirby_lost_health and kirby_health==1 then
    return -100
if kirby_died then
    return -1000
if kirby_touched_warpstar then
    return 1000
if boss_not_active then
    if kirby_moving_left then
        return -1
    if kirby_moving_most_left then
        return -5
    if kirby_standing_still then
        return -1
    if kirby_moving_most_right then
        return 5
    return 1
else
    if current_score > previous_score then
        return 100
return 0
```

First of all, the reward function is mostly based on movement, where the agent is penalized when it is not moving to the right. Rewards for movement is given every game tick, while rewards like dropping to 1 health or reaching the warpstar is given once on the instant the event happens. Kirby's x position is relative to the screen, and it stops changing when it is moving most to the left or most to the right. Therefore, we give the agent most penalty or reward for max movement in either direction. The agent is also rewarded when it reaches the warpstar, which initiates the bossfight. The reward function then changes to fit the bossfight, removing reward based on movement and instead the agent has to focus on increasing game score by damaging the boss.

2.4.2 Training with two agents

In Kirby's Dream Land, there are two quite different environments; the platformer level and the bossfight. Initially, the training only focused on the platformer part, and the agent was able to learn how to get to the warpstar to reach the boss. The reward function must now change to fit the boss environment, so the agent can learn to defeat it. Because the reward function for the bossfight focuses on attacking the boss and not movement and level progress, a second agent is introduced that would exclusively train on bossfights, in addition to the original agent that only trains on platformer levels. Algorithm 2 shows how we implemented this in the training loop. Two agents of the same type "aiPlayer" are declared at the start of each episode, and each of them will have their own experience replay memory, target and online networks. This allows the platformer player to focus on finding the optimal state for the platformer level, without having to learn the bossfight.

Algorithm 2 Episode loop with two agents

```
1:  platformerPlayer = aiPlayer()
2:  bossPlayer = aiPlayer()
3:  player=platformerPlayer
4:  for episode in range(total_episodes) do
5:      while not game_over do
6:          if is_boss_active then
7:              player=bossPlayer
8:          else
9:              player=platformerPlayer
10:         player.train()
```

3 Results

This section describes the results from training, and how we evaluated the results. After training was done, the agents state and neural network was saved to a file to be used for evaluation.

3.1 Super Mario Land

Table A1 shows the hyperparameters that was used for training with the DDQN algorithm. The training was done for 5519 episodes, and training took about 26 hours.

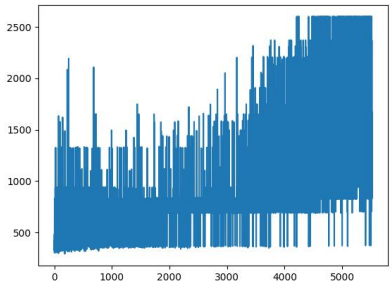


Fig. 7 The max length plot for Super Mario Land.

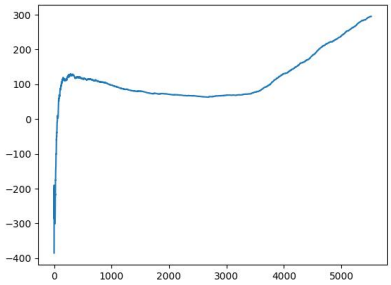


Fig. 8 The cumulative average reward for the previous 100 episodes Super Mario Land.

3.1.1 Evaluation

For Super Mario Land, the metric we wanted to evaluate was the max length the player character has traveled and time to reach this length. Unfortunately the AI did not manage to finish level 1 as it gets stuck in the last part of the level. This makes it hard to compare the AI to a human baseline.

3.2 Kirby's Dream Land

Table A2 and A3 shows the hyperparameters that was used for training with the Double DQN algorithm. Figure 9 shows the progress of reward for the agent. Since movement rewards are given each tick, this is the main factor that increases or decreases the agents reward. Because the rewards sign goes from negative to positive, this shows it has learned to walk to the right. The results for the reward after training the agent on the bossfight is shown in figure 10. The reward during the bossfight is very noisy, probably because rewards are given based on if the agent has hit the boss, which is a very specific state.

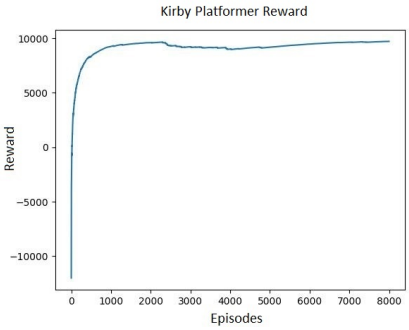


Fig. 9 The reward plot for the platformer part of Kirby's Dream Land.

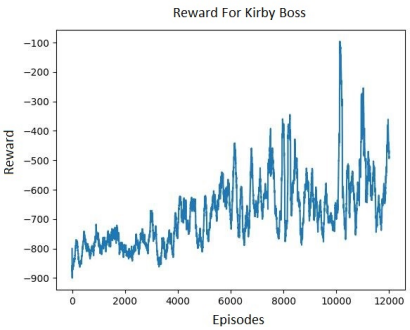


Fig. 10 The reward plot for the bossfight part of Kirby's Dream Land.

3.2.1 Evaluation

For Kirby's Dream Land, number of times the player reached the boss and average time to reach the boss was evaluated. For the bossfight, we evaluate how many times the agent was able to hit the boss in total, and the lowest health the boss reached. Evaluation was done by loading the saved platformer and bossfight model, and running the game for 100 episodes, ending when Kirby loses a life.

After 100 episodes, we got the following results:

- Number of times reached boss: 96
- Average time in seconds to reach boss: 8.77
- Number of times Kirby hit boss: 31
- Lowest boss health: 1

Time to reach boss is low because evaluation was done without rendering the emulator, to speed up evaluation. To compare how the agent compares to a human player, we compared 1 run from start to boss, each for a human and our agent. This was done by rendering the emulator, playing the game at normal speed:

- Time in seconds for human to reach boss: 23.23
- Time in seconds for agent to reach boss: 31.28

4 Discussion

This section discusses our results and problems we had during development. Finally, we describe possible future work for this project.

4.1 Super Mario Land

The results for Super Mario Land was that the agent was not able to complete the first level as the agent got stuck in the last part of the level. Although from the plots 7 and 8 we can assume that with more training the agent would be able to eventually finish the first level and move on to future levels.

4.2 Kirby's Dream Land

The results for Kirby's Dream Land was that the agent was able to complete the first part of level 1, reaching the warpstar 96 times out of 100 during evaluation. The agent also hit the boss 31 times after 100 episodes, but did not defeat it, only lowering its health to 1. If the agent were to play the bossfight perfect, hitting the boss three times per episode, number of hits required would be 300. This shows that the reward function works well for the platforming section, and with more learning it could potentially learn to defeat the boss.

4.3 Trying different reward functions

During development, the reward functions were iterated on and changed several times to improve the training.

4.3.1 Super Mario Land

The first reward function used for Super Mario Land was $reward = \Delta x - \Delta t$, this is because we only cared about speed running and time and x position are the only things that matter in speed running. The results we got were very bad as the agent would die repeatedly. Although we understand that the agent with more training could be able to find out that dying consumes repeatedly of time, we decided to include dying in the reward function as a punishment. This is to increase the learning of the agent such that the agent learns much faster that dying is bad.

4.3.2 Kirby's Dream Land

The first reward function used for Kirby's Dream Land was the one provided by PyBoy developers in the game wrapper [10]: $reward = score{*}health{*}lives\_left$

This reward function performs poorly for speedrunning, because it does not take into account the players movement in the level. Instead the agent learns to focus on increasing game score, which is not necessarily related to completing the game. Furthermore, the reward function was altered to take into account level progress, as explained in section 2.4.1. This works for the first part of level 1, since the goal is to move to the right, but the reward function will not work for more complex levels as they require vertical movement. Finally, for the bossfight we tried a reward function based on player position, rewarding it when it was in front of the boss. However, this did not work well and the

training outcome was that the agent was good at avoiding damage, but not able to damage the boss.

4.4 Acting in the environment

Early on, we had problems with the agent performing actions for just 1 frame at a time. This meant that the agent was not able to jump high enough to certain platforms, because a long jump requires the agent to perform the jump action for multiple frames. This was a big problem, as Mario and Kirby could not jump high enough to progress the game. This was fixed by changing the button-press mode in PyBoy to "toggle" buttons, as explained in section 2.2.2. Later on, we implemented simultaneous actions, and this increased the amount of actions, and the agent does not have to handle the state of a button press.

4.5 Future Work

For future work, we would like to train agents on more GameBoy games. We have generalized how a game specific agent is implemented by creating an interface called "AISettingsInterface", which is passed as an object to our custom PyBoy OpenAI gym environment. "AISettingsInterface" includes all functions that is required in an environment; GetReward, GetActionSpace, GetGameState, GetLearningHyperParameters and so on. To add a new GameBoy game, we need a GameBoy ROM file, and implement a class that inherits from "AISettingsInterface", and must implement the required functions as described. This enables us to set game specific settings like creating a reward function that fits the environment.

5 Conclusion

In conclusion, we successfully trained agents to play both Super Mario Land and Kirby's Dream Land through PyBoy, using the same DDQN networks but different reward functions and hyperparameters. The agent in Super Mario Land showed promising results, being able to complete most of level 1, but getting stuck towards the end. For Kirby's Dream Land the agent was able to reach the first boss and damage it, but it was not able to defeat it. From our results we see that reward functions keep increasing, and with more time and training results would improve for both games. From our results, we also conclude that PyBoy works well as an environment for reinforcement learning, as it integrates with OpenAI Gym and ability to read game information from GameBoy memory. However, we altered the default reward functions for each game to better focus on speed to complete games. Finally, for future work we made it possible to implement RL environments for other GameBoy games, and we encourage ourselves and curious readers to implement more agents to play GameBoy games. Finally, for future work we made it possible to implement RL environments for other GameBoy games, and we encourage ourselves and curious readers to implement more agents to play different GameBoy games.

Appendix A Hyperparameters for training

hyperparameter	value
learning_rate	0.00025
learning_rate_decay	0.99999985
memory_size	400000
batch_size	32
exploration_rate_start	1
exploration_rate_decay	0.9999975
exploration_rate_min	0.001
gamma	0.9
burnin	10000
learn_every	3
sync_every	1000
episodes	5519

Table A1 Hyperparameters for Super Mario Land.

hyperparameter	value
learning_rate	0.0002
learning_rate_decay	0.99999985
memory_size	500000
batch_size	64
exploration_rate_start	1
exploration_rate_decay	0.9999975
exploration_rate_min	0.01
gamma	0.8
burnin	1000
learn_every	3
sync_every	100
episodes	8000

Table A2 Hyperparameters for Kirby's Dream Land Platformer.

hyperparameter	value
learning_rate	0.0002
memory_size	500000
batch_size	64
exploration_rate_start	1
exploration_rate_decay	0.99999975
exploration_rate_min	0.01
gamma	0.8
burnin	1000
learn_every	3
sync_every	100
episodes	12000

Table A3 Hyperparameters for Kirby's Dream Land Bossfight.

References

[1] baekalfen: Baekalfen/PyBoy: Game Boy emulator written in Python. https://github.com/Baekalfen/PyBoy Accessed 2022-04-18

[2] Ynddal, T., Ynddal, M., Hansen, A.L.: Emulation of nintendo game boy (dmg-01) (2016). University of Copenhagen. Accessed 2022-05-01

[3] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)

[4] Feng, Y., Subramanian, S., Wang, H., Guo, S.: TRAIN A MARIO-PLAYING RL AGENT. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html Accessed 2022-05-01

[5] van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. CoRR **abs/1509.06461** (2015) 1509.06461

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015). https://doi.org/10.1038/nature14236

[7] Shyalika, C.: A Beginners Guide to Q-Learning. Model-Free Reinforcement Learning (2019). https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c Accessed 2022-05-01

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013). https://doi.org/10.48550/arXiv.1312.5602

[9] Hafner, D., Lillicrap, T., Norouzi, M., Ba, J.: Mastering atari with discrete world models. arXiv preprint arXiv:2010.02193 (2020)

[10] baekalfen: game_wrapper_kirby_dream_land.py. https://github.com/Baekalfen/PyBoy/blob/master/pyboy/plugins/game_wrapper_kirby_dream_land.py Accessed 2022-04-18

[11] DataCrystal-contributors: Kirby's Dream Land RAM map. https://datacrystal.romhacking.net/wiki/Kirby%27s_Dream_Land:RAM_map Accessed 2022-04-18