

Deep Q-Learning Agent learns to play a simplified Axie Infinity environment.

Peshang Alo^{1*†} and Rune Alexander Laursen^{1*†}

^{1*}Department of Engineering and Sciences, University of Agder,
Jon Lilletuns vei 9, Grimstad, 4879, Agder, Norway.

*Corresponding author(s). E-mail(s): peshaa18@uia.no;
runeal17@uia.no;

[†]These authors contributed equally to this work.

Abstract

In Reinforcement Learning, games are very popular for both teaching and developing new algorithms for different problems. This paper will propose a solution to create an agent that can learn to play the card-game Axie Infinity by using a Deep Q-Network. Part of the problem was to develop the environment, the group used C++ to implement it and pybind11 to access the code from python. The best result for learning was achieved with the agent playing both player one and player two, where it was able to maximize the rewards and keep a high win rate.

Keywords: Game, Axie Infinity, Reinforcement Learning, Deep Q-Learning

1 Introduction

It has become very popular to use games to learn about reinforcement learning (RL). This article will cover the implementation of an environment mimicking the NFT-based¹ card game Axie Infinity explained in appendix A and the implementation of an agent using the environment to learn how to play the game.

It would be beneficial to test teams before buying them because it costs real-life money to buy a team consisting of multiple NFTs in the Axie Infinity marketplace. By using unsupervised learning to form a policy, players could

¹None-fungable token

test teams from the marketplace before buying them. At the time of writing, there are no implementations of this environment online, which means this project will add to the research in this field. In addition, the state variables in the game are not accessible by users, making it impractical to try solving with supervised learning when there is no labeled data.

Because the cards that can be chosen are given randomly, the environment is stochastic, making the problem exponentially more difficult to solve the more different variables there are. Furthermore, the agent does not have complete visibility of the entire state of the environment because it can only see the opponents' cards after it has decided what action to take.

Axie Infinity is a fairly new game which might be one reason there are no previous solutions. There are many implementations of other card games, but they are too different, so it makes more sense to implement a new one.

The first big obstacle is that the game environment is only available in the official game client, which cannot be sped up for faster training. With that tightly locked environment, an agent is very time-constrained and cannot play multiple games simultaneously; it must play one battle at a time in real-time. Also, some data is available online from previous battles, but not round specific data, which is needed to solve with RL.

With these limitations, the group implemented an environment based on the game's rules [A](#). Moreover, with time constraints, the group decided to simplify the game quite a bit. After simplifying the game, the group implemented the game in C++ and used pybind11 [\[1\]](#) to access the code from python. The DQN algorithm was implemented in python using Tensorflow to create a training network using the environment library to train. The results from the training varied a lot depending on how the agent was rewarded and how large the reward was.

2 Background

This section will explain some essentials needed to understand better when reading this article.

2.1 Reinforcement Learning

A popular way to create models in Machine Learning is by training a network with labeled data, which is called; **supervised learning**. However, often, there is no labeled data available, and training has to be done another way. This is where reinforcement learning can be used, which is a type of **unsupervised learning**. This method creates an optimal strategy by teaching an agent to take the best actions in an environment and using rewards to create patterns between a state and the best actions for that state [\[2\]](#). The agent determines which action to take at any specific time by employing exploration and exploitation techniques, with either random actions (exploration) or prior knowledge (exploitation) as input [\[3\]](#).

2.2 Markov Decision Process

A Markov Decision Process (MDP) is when an RL problem is solved by an agent who decides what the best action is in the current state; when this step is repeated, it is called an MDP. The parts of the problem required for MDP are states, a model, actions, and rewards, as shown in Figure 1. All these variables are used and form the policy as the solution [4].

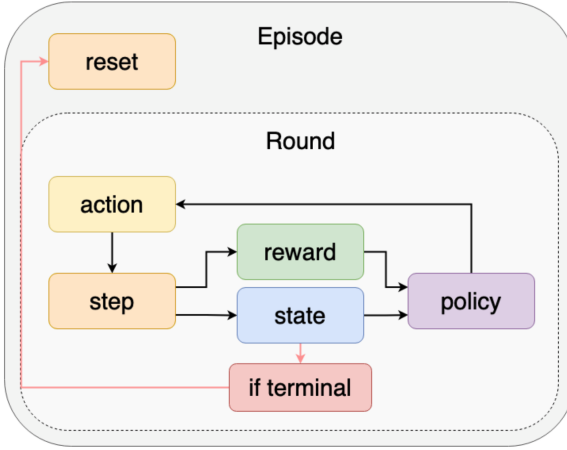


Fig. 1 Markov Decision Process

2.3 Essential elements

The following is a list of essential elements that are used to construct an RL algorithm:

- An **environment** is the logical code representation of a game or real-world problem, and it is what an agent interacts with.
- An **agent** is the algorithm that interacts with the environment and is the way that a policy can take action.
- The **Goal and Reward** are specified by the rules of the environment used. The goals are used to reward the agent based on how good it is to follow the rules.
- The **action** is a set of the possible choices that the agent can make in the environment. The policy gives the action at each step. However, it uses a value called *Epsilon-Greedy*² to choose either the given action or random action. The *Epsilon-Greedy* value decreases by some set amount for each new episode to take action less randomly.

²The Epsilon value is used to balance between exploration and exploitation.

- A **policy** is the probability distribution of the mapping between actions and states. It can be considered the brain of the algorithm or agent.
- A **value functions** predicts the value of an agent given its current state. There are two kinds of value functions in RL:
 1. **Action-value function** defines the value of taking action in some given state when following the policy, and is shown in 1 below.

$$Q(s, a) = \max_{\pi} a_{\pi}(s, a) \quad (1)$$

2. **State-value function** defines the value of being in some certain state when following the policy, and is shown in Equation 2 below.

$$V(s) = \max_{\pi} v_{\pi}(s) \quad (2)$$

- **Model-based vs. Model-free Reinforcement Learning**

- **Model-based** RL is the simplest form of reinforcement learning. The agent already knows everything about the environment (The Markov Decision Process is well known), which means that there is no need for exploration [5].
- **Model-free** RL means that the agent does not have any prior understanding of the environment, and the only way to gain knowledge is by interacting with it [6].

2.4 Bellman Equation

The Bellman equation gives the RL algorithm information about what long-term reward the agent can expect, given the state that the agent is in and assuming that the agent takes the best possible action [7]. $V(s)$ is the value of being in a particular state. $V(s')$ is the value of the next state after taking the action a . $R(s, a)$ is the reward gained after taking action a in state s . The max function makes the agent take the best action for ends up in an optimal state. γ is the discount factor for the reward the agent will get in the future [8]. The bellman equation for Deterministic Environments is shown in Equation 3 below:

$$V(s) = \max_a (R(s, a) + \gamma V(s')) \quad (3)$$

The bellman equation for Stochastic Environments is shown in Equation 4. $P(s, a, s')$ is the probability of ending up in state s' by taking action a [8].

$$V(s) = \max_a (R(s, a) + \gamma \sum_{f'} P(s, a, s') V(s')) \quad (4)$$

2.5 Deep Q-Learning and Deep Q-Network

Deep Q-Network (DQN) is a reinforcement learning algorithm integrating Q-Learning³ and deep neural networks (DNN)⁴. DeepMind proposed DQN in 2013 to bring the benefits of deep learning to reinforcement learning [9]. Q-network is an agent trained to provide the best State-Action value and learn how to take the best action in each state. The Q-network generates a memory table (Q-values) for every possible state and action combination. Deep Q-Learning is used when the Q-values are too large and insufficient computation power. Instead of remembering the entire table, DQN will approximate the Q-value function generally [10].

2.6 C++ and Python

Many popular algorithms and libraries used today are written in Python, a language that is easy to read and write, but it has some drawbacks compared to other languages. One of the drawbacks is that the language is dynamically typed, so there is no need to specify what type a variable is. This saves time when writing a program or an algorithm but is much slower to run than a statically typed language. For example, C++ is a statically typed language, where all variables must have a specified type. These facts make C++ code run a lot faster than Python. Many Python libraries, like NumPy and pandas, take advantage of this and are written in C++.

3 Approach

This section will describe the implementation that tries to solve the problem.

The original game's environment is quite complex, so the group decided to simplify the environment and remove some of the logic.

3.1 Reinforcement Learning Program

The program implemented was inspired by a game found online [11] that uses DQN to play the game paddle. The program can now run without errors with many modifications to parameters, functions, and the environment used.

The DQN uses the TensorFlow ML library[12]; the model consists of three dense layers with "Relu" activation on the first and "Linear" activation on the last [13, 14]. The optimizer used is Adam with a learning rate=0.001 and the "MeanSquareError" (MSE) loss function [15].

3.2 Environment

The environment is a simplified self-made Axie Infinity game created in C++. The simplifications made to the environment are as shown in Table 1 and

³Q-learning is a model-free reinforcement learning algorithm that seeks to find the best action to take given the current state.

⁴Deep Neural Network is a machine learning algorithm.

explained as follows; each team has one less Axie, each Axie has two fewer cards, and cards no longer have special abilities, like additional damage and restoration of health. The idea is that the agent will learn to take corrective actions based on various conditions (position of the defender Axie, availabilities of cards, and height of the axes) to win the game. The environment is stochastic due to the random choice of cards that the agent chooses. The figure below shows the environment that consists of two players, and each player has two axes.

Table 1 Differences between the original and simplified versions of the game.

	Axie Infinity	DeepAxie
no. Axes	3	2
no Cards/Axie	8	4
Total no. Cards	24	8
Special Abilities	✓	✗

```

-----
Round: 1 - (attack order : type : health)
Energy: 2
  back |          front          |  back
      |4:plant:516          |1:aqua:414
      |3:plant:486      2:plant:256|

attack done
-----
player-id 7 is choosing cards:
1- Axie 4:plant - dmg:60 - def:60
2- Axie 4:plant - dmg:115 - def:20
3- Axie 4:plant - dmg:60 - def:60
4- Axie 4:plant - dmg:115 - def:20
Choose cards, enter 0 to skip

```

Fig. 2 Environment

The player (agent) has nine actions to choose from; 0 to skip choosing any cards, and 1 to 8 to select a card. Each player starts with two energy and gains two more energy each round. The energy allows the player to select up to two cards per round. With the ability to choose up to 2 cards, the action space will be a combination of two numbers in the following list [0,1,2,3,4,5,6,7,8,9], which again gives us the action-space of 88 becomes a discreet space with 88 possible combinations (00,01,02,03,...,86,87,88).

The state-space vector consists of 88 distances [d0,d1,d2,...,d88], which represent all different variables for both player and their axes. These variables are shown in Figure 3 below.

axiel										axiel0									
card3	card2	card1	card0	type	speed	pos	health			card3	card2	card1	card0	type	speed	pos	health	energy	rank
def										def									
dmg										dmg									
0, 0, 0, 50	0, 0, 30, 30	0, 0, 0, 50	0, 0, 30, 30	0, 0, 1, 31	516	0, 0, 0, 50	0, 0, 0, 50	0, 0, 0, 50	0, 0, 0, 50	0, 0, 0, 50	0, 0, 30, 30	0, 0, 0, 50	0, 0, 30, 30	0, 0, 0, 50	0, 0, 30, 30	0, 0, 0, 50	0, 0, 30, 30	0, 0, 1, 31	516, 2, 1200

3.5 Docker

The group chose to use Docker to run this intricate system with all the different libraries and requirements because it gives the ability to execute the code across systems, which was a problem in the group at the beginning of the project. It also gives the project a much higher reproducibility if others want to use it or develop it further [16].

4 Results

This section will go through the results achieved in this project.

The first reward function used produced the graphs in Figures 5, 6, and 7. Figure 5 below compares four different training sessions between agents using policy and random function to decide what action to take. The "win-loss ratio" graph in Figure 5 gives insight into how the battle results are from player 1's perspective.



Fig. 5 Win-loss ratio for player 1

Figure 6 below shows the moving average of the rewards for player 1 over 1000 epochs.

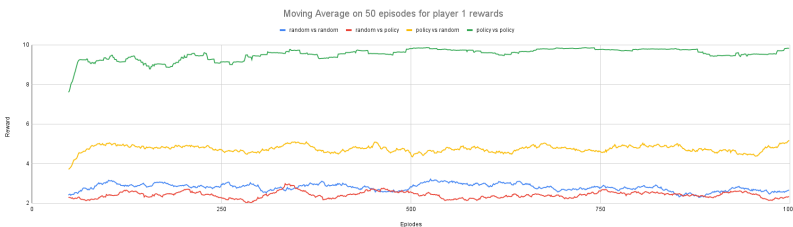


Fig. 6 Rewards for player 1

Figure 7 below shows the moving average of the rounds for player 1 over 1000 epochs.

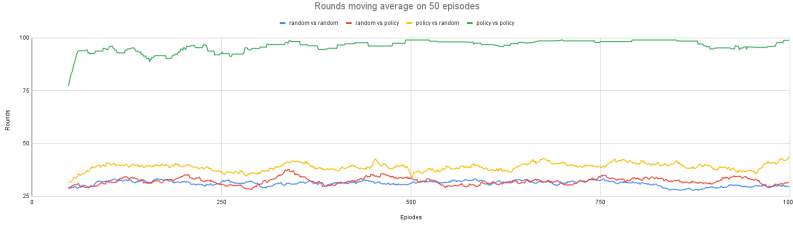


Fig. 7 Rounds for player 1

After modifying the reward function only to give rewards when an agent is winning, the results are shown in the graphs in Figures 8, 9, and 10.

The graph in Figure 8 shows the player 2 agents winning ratio is steadily increasing until the training is complete.

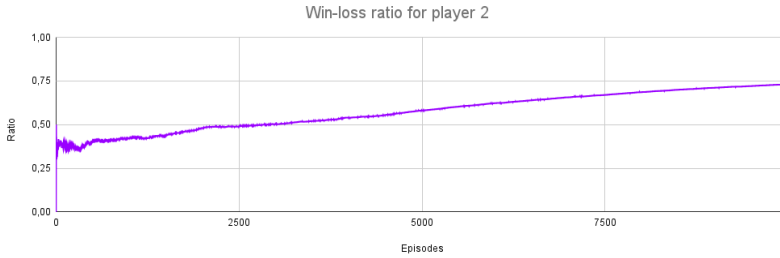


Fig. 8 Win-loss ratio for player 2

The graph in Figure 9 shows the rewards for the player 2 agent are increasing.

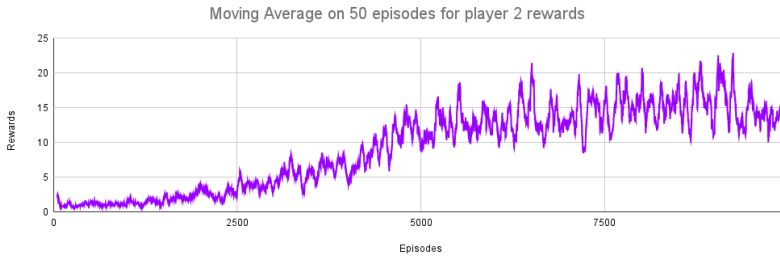


Fig. 9 Rewards for player 2

The graph in Figure 10 shows the number of rounds is decreasing in the first 5000 rounds, and then just oscillating below and around the 20-round line.

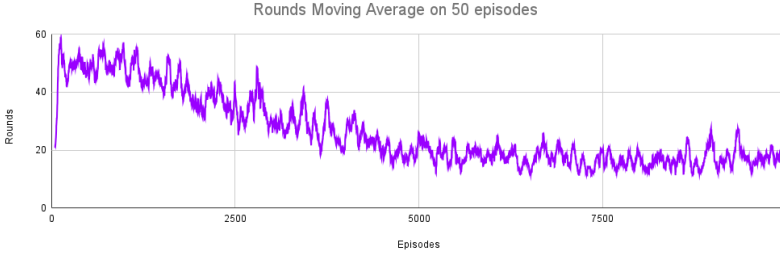


Fig. 10 Rounds for player 2

5 Discussion

Implementing the environment from scratch in C++, learning how to use Pybind to execute the code in python, and getting everything to run together without crashes and bugs took a lot of the time. Although implementing a new environment has many advantages, such as the flexibility of making adjustments and further developing it later, it also has disadvantages, such as draining the project’s time and much extra debugging when unexpected things happen while training. We had to go back into the C++ code to debug to locate the problem, then implement new solutions to fix it. Using a pre-existing environment would have given more time to fine-tune the algorithm.

The simplifications made to the environment may have inadvertently made it harder for weak teams to win by reducing possible outcomes that the player/agent could have achieved with creative combinations of card abilities.

Towards the end of the project period, we spent some time implementing an algorithm called Proxima Policy Optimization (PPO) [17]. Unfortunately, we ran into an issue where we coded some parts of the environment into the DQN algorithm, which prevented us from implementing the PPO algorithm in time.

Using Docker to containerize the library, environment, and algorithm benefits natively being run across various systems without problems, but unfortunately, we did not get all elements to work. For example, the container can compile the project as a library inside it, but it cannot run because we did not get TensorFlow to work inside the container.

When training the agent against an opponent that behaves randomly, the agent does not appear to learn due to the stochastic behavior. We avoided this issue of training against a random player by using the policy to decide actions for both players, which resulted in more deterministic behavior. This fix stabilized the learning progress and made the agent reach for higher and higher rewards. However, this solution is not ideal because the agent can see the state variable representing the opponent’s card status, which is against the game’s rules.

Rewarding valid actions was quite a challenge; we started with only giving a large negative reward for choosing incorrect ones. After a lot of trial and error,

we did not use this masking. We also tried to reward the agent for damaging the opponent and not taking any damage. This reward function led to both agents agreeing not to do any damage and harvesting the rewards until the game was self terminated.

The agent did not seem to learn anything with the first reward function. From the blue line in Figure 5, it seems that player 1's team is a bit stronger than player 2's team, while the green line shows that the agent does not care about winning. The yellow and red lines start in favor of the player using policy but lose more and more. This loss can be attributed to the random function often getting cards that will damage the opponent and win the game, whereas the "smart" player will try to maximize rewards by staying in the game as long as possible. Lastly, the green line on the graph in Figure 6 shows the rewards with a moving average of 50 that the agent tries to play as many rounds as possible to increase the rewards, as the graph in Figure 7 confirms.

After modifying the reward function to reward wins, the agent is stimulated to win as fast as possible to get a higher reward, which is a better way to get the agent to learn. The figures show that the agent is learning how to win in as few rounds as possible. As Figure 9 shows, we can assume that the agent is learning because it is getting better over time. Figure 8 tells us also that it is getting better at winning. Figure 10 is another pointer to an agent that is learning to play fewer rounds to increase its reward.

Some of the teams used are so weak that they will never win. These teams require another reward function because no wins equal no reward. Hence, it will never learn anything. It was interesting to watch how the agent changed behavior when we changed the rewards, but in the end, it seemed best to only reward the agent for winning and not give any negative rewards.

6 Conclusion

This project aims to use RL to teach an agent how to play the simplified version of the Axie infinity game. The environment, program, and algorithm created in this project were suitable for the problem and showed promising but varying results.

Implementing our own environment is a challenging task due to the complexity of the game Axie Infinity. Using C++ and creating bindings that allow python to execute the code adds to this challenge. However, the results are promising, as they show that the agent manages to learn how to win and at the same time decrease how many rounds it takes over the long run. Additionally, when both players use random action to play against each other, we can see which one of the teams is the strongest just by the numbers.

Developers should adjust the reward function in future work so the agent tries to win, not play as long as possible. When changing the reward function, it is essential to consider the strength of the teams playing against each other. Further, the function that returns the state represented as bits from C++ does not seem to work correctly and needs to be tested further. Furthermore, future

developers should modify the program to separate the environment from the algorithm to make it gym compatible, which would make it straightforward to try other algorithms. Lastly, the container needs to support TensorFlow to get everything working in Docker.

Acknowledgments. We want to thank our fellow graduate student Ole Gunvaldsen for discussing the setup of the environment, debugging, and finding issues in the environment towards the end of its development.

We also want to give a big thanks to our supervisor Per-Arne Andersen for guidance and inspiration throughout the project.









Lastly, we want to thank a few people from the internet, Kushashwa Ravi Shrimali, Shiva Verma, and Keith Whitley, for their tutorials, which sped up our development process.

Declarations

This project is created for research and educational purposes only; the terms of service of the official game state that it is not allowed to use bots to play the game. Anyone using this project to further develop anything against the terms of service of the official game is doing so under their responsibility.

Appendix A Axie Infinity Game Rules

Axie Infinity Game Rules

 Created By	
 Stakeholders	
 Status	
 Type	Project Kickoff 
 Created	@May 1, 2022 2:44 PM
 Last Edited Time	@May 1, 2022 2:46 PM
 Last Edited By	

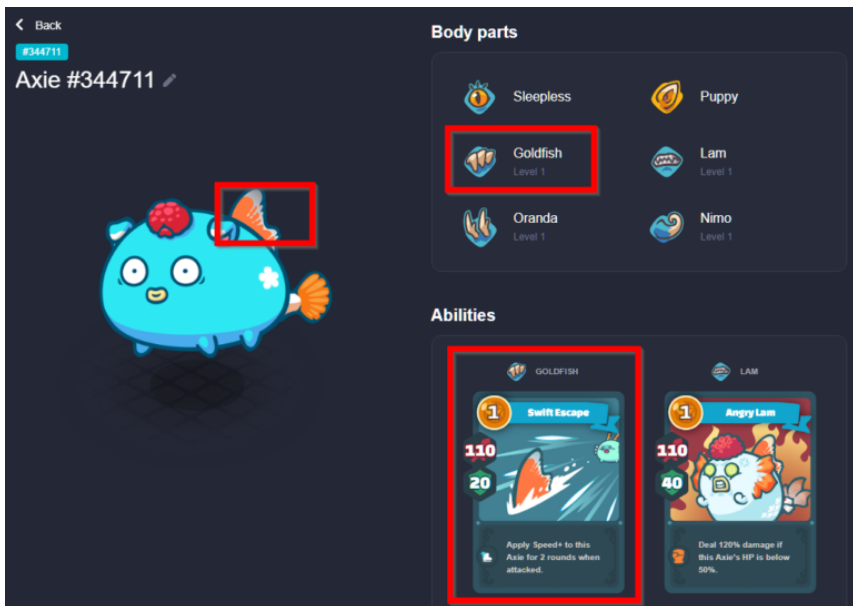
Game round:

<https://www.youtube.com/watch?v=d3GdMH2nUul&t>

How the game works:

Axie Infinity is a popular Pokemon-inspired, blockchain-based P2E (Play to Earn) game where players collect virtual creatures in the form of NFTs called Axie.

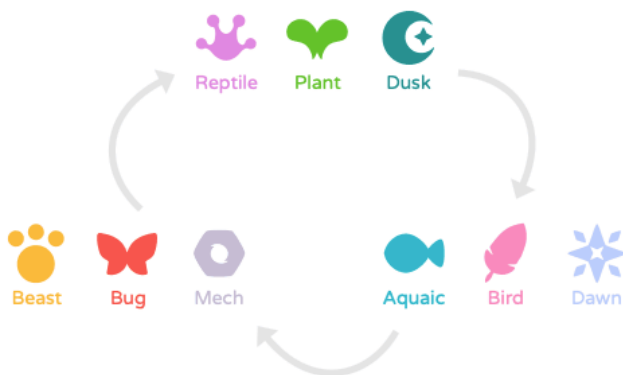
Axies have six body parts: Eyes, Ears, Horn, Mouth, Back, and Tail. Four of them (Horn, Mouth, Back, and Tail) give the Axie specific abilities that can give bonus damage, healing, and much more. Each card also has an energy cost to use the card.



Body Parts = Cards

Each Axie will have four out of 132 different cards, but it can not have two of the same.

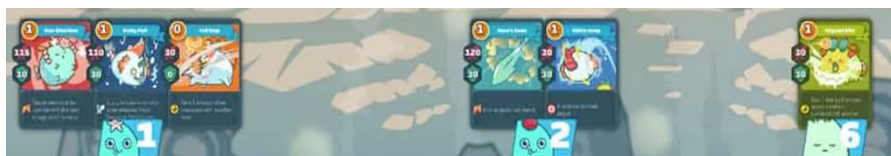
There are nine different classes, which they are either weaker or stronger against, this gives extra damage or extra shield.



Advantages and disadvantages between the classes.

In a battle, there are always two players with 3 Axes each.

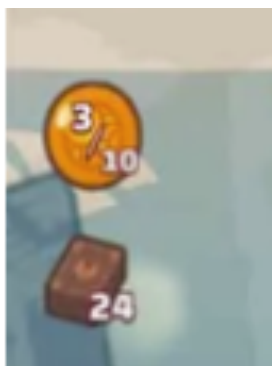
The abilities are shown in combat as cards at the bottom of the screen, like the image below shows, and are used to defeat the opponent. The opponents' cards can be seen by clicking on the Axie or by looking at its body parts.



Player Axes and the cards it can play

The players' card deck consists of 2x all cards their Axies has and is shuffled.

The first round starts with each player getting three energy and six cards out of the deck of cards.

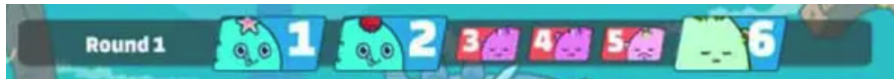


Energy and count of cards

In the following rounds, the players get two energy and x cards, when the deck of 24 cards is empty, all the cards are reshuffled. This information is available on the battle screen and is updated each round.

The players have 60 seconds to choose what cards to play before the battle starts, or the "End Turn" button is pushed.

The speed of the Axie determines the order of attack, which means the fastest will attack first.



Turn order, the player team is **blue** and opponents are **red**

The placement of the Axies on the battleground is chosen by the player before the battle starts. The placement also gives the order of which of the opponent Axies will be attacked by the player; The one in front will always be attacked first, except when the player uses cards with an ability that says otherwise.



Axie Infinity battle

Appendix D DeepAxie Code on GitHub

The code for the DeepAxie program can be found in the following git repository:

<https://github.com/cair/DeepAxie>

References

- [1] Intro — pybind11 documentation. <https://pybind11.readthedocs.io/en/stable/>
- [2] Sutton, R., Barto, A.: Reinforcement Learning: An Introduction, (2018). <https://books.google.com/books?hl=en&lr=id=uWV0DwAAQBAJoi=fndpg=PR7dq=reiF4-bbYIUG-hdW-LF18ZFVHflzxs>
- [3] Epsilon-Greedy Algorithm in Reinforcement Learning - GeeksforGeeks. <https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/>
- [4] Dodson, T., Mattei, N., . . . , J.G.-C.o.A.D., 2011, u.: A natural language argumentation interface for explanation generation in Markov decision processes. Springer **6992 LNAI**, 42–55 (2011).
- [5] Kaiser, , Babaeizadeh, M., Miłos, P., Zej Osí Nski, B., Campbell, R.H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., Mohi-uddin, A., Sepassi, R., Tucker, G., Michalewski, H., Brain, G., Ai, D.: Model-based reinforcement learning for atari. arxiv.org
- [6] Strehl, A.L., Lihong, L., Wiewiora, E., Langford, J., Littman, M.L.: PAC model-free reinforcement learning. ACM International Conference Proceeding Series **148**, 881–888 (2006). <https://doi.org/10.1145/1143844.1143955>
- [7] O’Donoghue, B., Osband, I., Learning, R.M.-.M., 2018, u.: The uncertainty bellman equation and exploration. proceedings.mlr.press (2018)
- [8] Bellman Equation and dynamic programming by Sanchit Tanwar Analytics Vidhya Medium. <https://medium.com/analytics-vidhya/bellman-equation-and-dynamic-programming>
- [9] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arxiv.org
- [10] Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning **8**(3-4), 279–292 (1992). <https://doi.org/10.1007/BF00992698>
- [11] Verma, S.: Create Your Own Reinforcement Learning Environment by Shiva Verma Towards Data Science. <https://towardsdatascience.com/create-your-own-reinforcement-learning-environment-beb12f4151ef>

- [12] Module: tf.keras TensorFlow Core v2.9.1.
https://www.tensorflow.org/api_docs/python/tf/keras
- [13] tf.nn.relu TensorFlow Core v2.9.1.
https://www.tensorflow.org/api_docs/python/tf/nn/relu
- [14] tf.layers.Linear TensorFlow Lattice.
https://www.tensorflow.org/lattice/api_docs/python/tflayers/Linear
- [15] tf.keras.losses.MeanSquaredError TensorFlow Core v2.9.1.
https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanSquaredError
- [16] Home - Docker. <https://www.docker.com/>
- [17] Schulman, J., Wolski, F., Dhariwal, P., . . . , A.R.-a.p.a., 2017, u.: Proximal policy optimization algorithms. arxiv.org