Traind AI Warfleet

Michel Schwarz
Hochschule Düsseldorf
michel.schwarz@study.hs-duesseldorf.de

Niklas Tluk
Hochschule Düsseldorf
niklas.tluk@study.hs-duesseldorf.de

ABSTRACT

The game Warfleet is a timeless classic, popular to this day in its countless iterations, ranging from the original board game to traditional video games and more recently VR applications. This is an implementation of said game in Python using neural networks and reinforcement learning. The A2C and PPO2 RL algorithms were employed to train intelligent agents, based on the OpenAI Gym toolkit and the OpenAI Baselines framework, to competently play it.

CCS CONCEPTS

• **Computing methodologies** → Reinforcement learning; Learning from critiques; Intelligent agents; Neural networks.

KEYWORDS

intelligent systems, intelligent agents, deep reinforcement learning, neural networks, A2C, PPO2

1 INTRODUCTION

Considering the current prevalence of machine and deep learning, intelligent systems are now more relevant than ever. This project is part of our participation in the intelligent systems course at the Hochschule Düsseldorf, Germany. The course served as an introduction to the design and implementation process of intelligent systems. It contained the history of machine learning, basic underlying principles, methods and algorithms aswell as current relevant topics in the media information field, like deep learning, date mining and predictive analytics.

The Project files can be found **here.**

2 MOTIVATION

Before starting this course we had already heard of Deepmind's and OpenAI's impressive and at times even mind-boggling accomplishments with AlphaGO, AlphaStar, OpennAI Five and such. So when we were given the option to train an agent to play a simple game, using reinforcement learning and neural networks, ourselves we were excited and eagerly began coming up with ideas.

3 GOAL

Our goal was the training of multiple intelligent agents, capable of competently playing the board game warfleet. We used Python to implement the reinforcement learning process. To achieve that we implemented different reinforcement learning models.

During the training the agent should show a significant improvement in playing the game. The intermediate results should show that the agents are getting a rising reward.

At the end of the training the agent should be much better then the computer. The agent's opponent targets the ships using two random variables. At the beginning of the process, both started with random variables. After a few played rounds it should turn out that the agent comprehends the structure and decides to attack a boat by its length an position on the battlefield. For example when a boat is hit at position (4/4) the agent should learn that the next field is left, right, up or down. If the ship is not destroyed after two shots the agent should learn that the ship is longer than two fields.

4 ENVIRONMENT

To achieve this we also had to develop a feasible environment for the agent to be trained in. For this purpose we chose the OpenAI Gym toolkit, which provides an easy-to-use suite of reinforcement learning tasks. The playing field or board of our game is a 10 by 10 2D array of the type integer. Possible values here are 1 for water, 2 for sections of a ship and 3 for shot positions(impacts).

```
self.action_space = spaces.MultiDiscrete([10, 10])
self.observation_space = spaces.MultiDiscrete([
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
])

self.shot = "0"
self.empty_field = "1"
self.ship = "2"
```

Listing 1: Action space and Observation space

The action space in our environment consists of all possible coordinates in said board while the observation space describes the amount of possible values, 3 in this case, for every board position.

When the game begins, the ships will be placed on the battlefield. Each player has ten ships. Table 1 shows which ships are placed before the game begins. A method which is called place_ships contains the code for the placement. This method is activated before the trained agent is activated. This means that the algorithms PPO2 and A2C are not related to the placement of the ships but only deal with the gameplay.

The code in listing 2 is the console output for the agent battlefield after the game has started. Each ship is placed inside the battlefield. You can see that the ships represented by the number 2 are placed side by side. At the same time, care is taken to ensure that the ships

Ship	Size	Quantity
small_ship	2	4
middle_ship	3	2
big_ship	4	2
cruiser_ship	5	2

Table 1: Ship information

do not lie on top of each other.

```
--- Player/Agent Battlefield ---
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[2, 2, 2, 2, 2, 1, 1, 1, 1, 1]
[2, 2, 2, 2, 2, 1, 1, 1, 2, 1]
[1, 2, 2, 1, 1, 1, 1, 1, 2, 1]
[1, 1, 1, 2, 2, 2, 1, 1, 1, 1]
[1, 1, 1, 2, 1, 2, 1, 1, 1, 1]
[1, 2, 1, 2, 1, 2, 2, 2, 2, 2]
[1, 2, 1, 1, 1, 2, 1, 1, 1, 1]
[1, 1, 2, 2, 2, 2, 1, 1, 1, 1]
```

Listing 2: Battlefield in start condition

After the battlefield has been filled with ships, each player starts shooting. A shot is visualized by the number 0. The code below shows the first line from the player/agent battlefield. The first shot is placed and the number 1 is replaced by the number 0.

```
[1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
```

Listing 3: A line with a hit mark

Within the environment the ships of each player are added to a list. The list contains ship objects. A parameter contains the size for each ship. If a ship is hit by a shot, the size is reduced by 1, if the size is 0, the ship is removed from the list. The game ends when one of these lists is empty.

4.1 Reward

The environment gives the agent a reward when the agent has hit a ship. This is a value of one for a simple hit, or ten if a whole ship is destroyed. The agent should notice that the reward is significantly higher when a ship has sunk.

5 METHODS

The agents experience the environment by interacting with it. We trained out agents without any given data-set of instructions. The goal for the agent is to optimize its policy to receive as much reward as possible.

Every time the agent gets a reward the policy will recognize the value of said action and therefore increase it probability in similar circumstances.

The reinforcement learning algorithms are implemented in the stable-baselines toolkit. Every algorithm has different characteristics.

5.1 Procedure

It should be clear how the agents interact with the environment and what is done to achieve the desired results. The agent goes through the environment in several steps while the game is played in turns.

The current state(S) of the environment is sent to the agent. Based on this state(S), the agent makes the next decision. This decision is called action(A). In our case the agent receives the situation on the battlefield and can see if a target is hit or not.

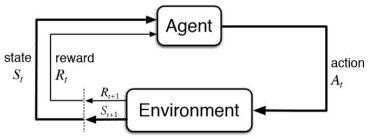

Figure 1: Procedure

5.2 Policy Gradient Methods

To realize the training we need a policy. To be able to process the tasks the agent needs a route that leads him to the target. This route is defined by a policy. The agent is told by the policy what the next action will look like. It is important to know that the policy is optimized on the basis of the self-generated data.

The Policy Gradient Methods are a bundle of reinforcement learning algorithms. The goal is to solve reinforcement learning problems. So the agent should learn a strategy which is increasingly improving. To achieve this, the reward should be increased to a maximum [1, 16-18].

There are some default policies provided by the stable-baselines project. In our project we have used the MlpPolicy. MLP stands for Multilayer Perceptrons. This type of network is called a feedforward network. One layer is always connected to the next layer. So there is only one direction to go through the network. The network consists of an input and an output layer. There can be several hidden layers between them [2, 174].

The learning methods used by us are the Proximal Policy Optimization and Advantage Actor Critic.

5.3 Proximal Policy Optimization

The idea for PPO is similar to the TRPO method but it is not as complicated. PPO is a an on-policy actor-critic algorithm which means that it is learning by its own generated data. An important feature is that the algorithm pays more attention to the results of recently completed turns than to results that are significantly older.

5.4 Advantage Actor-Critic

The A2C is an on-policy actor-critic algorithm. It is based on the idea of the A3C but in a synchronized variant. In this case the critic is a value function that gives a feedback to the policy. The policy

is the actor. While the actor observes the environment, the critic estimates values to achieve a better action. This approach shows significant improvements in the results and is more and more used in policy gradient algorithms.

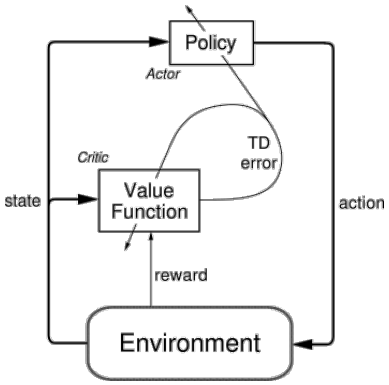

Figure 2: Actor-Critic principle

6 IMPLEMENTATION

The Open Ai Gym toolkit is imported to load our environment. Before the learning process can start the environment is loaded. To realize the implementation we imported some packages from stable-baselines. This toolbox makes it easy to implement reinforcement learning algorithms. For beginners it is easy to start reinforcement learning by using those algorithms.

```
import gym
import gym_wf
from stable_baselines.common.policies import
    MlpPolicy
from stable_baselines.common.vec_env import
    DummyVecEnv
from stable_baselines import PPO2, A2C
```

Listing 4: Imports

6.1 Algorithm implementation

The PPO2 algorithm is used with some different parameters to decrease the variance between the policies. This process is called Hyperparameter tuning. That means that the algorithm will be trained in a slightly different way by adjusting some parameters. The A2C Algorithm is used without changing any parameters [3, 5-7].

```
model = PPO2(MlpPolicy, env, verbose=1,
tensorboard_log=log_dir, cliprange=0.1,
gamma=0.99, ent_coef=0.001, vf_coef=0.2)

model.learn(total_timesteps=timesteps_input)
model.save(model_save)
```

Listing 5: PPO2 implementation

```
model = A2C(MlpPolicy, env,
verbose=1, tensorboard_log=log_dir)
model.learn(total_timesteps=timesteps_input)
model.save(model_save)
```

Listing 6: A2C implementation

6.2 Training settings

When the training starts, some information must be entered into the console. The first information is the algorithm that should be trained. The second is the number of timesteps. And the last one is the filename that contains the model.

```
Select algorithm (PPO2 or A2C only): PPO2
Choose number of timesteps: 10000000
Select model to test(input filename, eg. a2c_wf_2
    or ppo2_wf_4):a2c_wf_3
```

Listing 7: Console output before start

7 RESULTS

The aforementioned training process resulted in the creation of multiple agents based on both algorithms. Through prolonged training the agents gained proficiency in playing the game, surpassing the random AI and finally reaching a win rate of 100% with a varying number of its own ships remaining.

This is the end state of a Warfleet game played by the well trained “a2c_wp_2” agent against a random AI:

```
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 1, 0, 1, 1, 0, 0, 1]
[0, 0, 0, 1, 0, 0, 0, 0, 1, 1]
[1, 0, 1, 1, 0, 0, 0, 0, 1, 1]
[0, 0, 1, 1, 0, 0, 0, 0, 1, 1]
[0, 0, 0, 1, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 1, 0, 1, 1, 0, 1, 1]
[0, 0, 0, 1, 1, 1, 1, 0, 0, 1]
[1, 0, 0, 1, 0, 0, 0, 1, 0, 1]
```

Listing 8: Computer battlefield end-state

As you can see on the computer battlefield, all of the its ships have been destroyed by our agent quite accurately with very little shots missing their mark .

```
[0, 0, 0, 1, 1, 0, 0, 0, 1, 1]
[2, 2, 2, 0, 2, 0, 0, 1, 0, 1]
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
[1, 0, 1, 0, 0, 0, 1, 0, 0, 2]
[0, 0, 1, 2, 0, 0, 0, 2, 2, 2]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 2, 2, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 2, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 1, 0, 0, 2, 0, 0]
```

Listing 9: Agent battlefield end-state

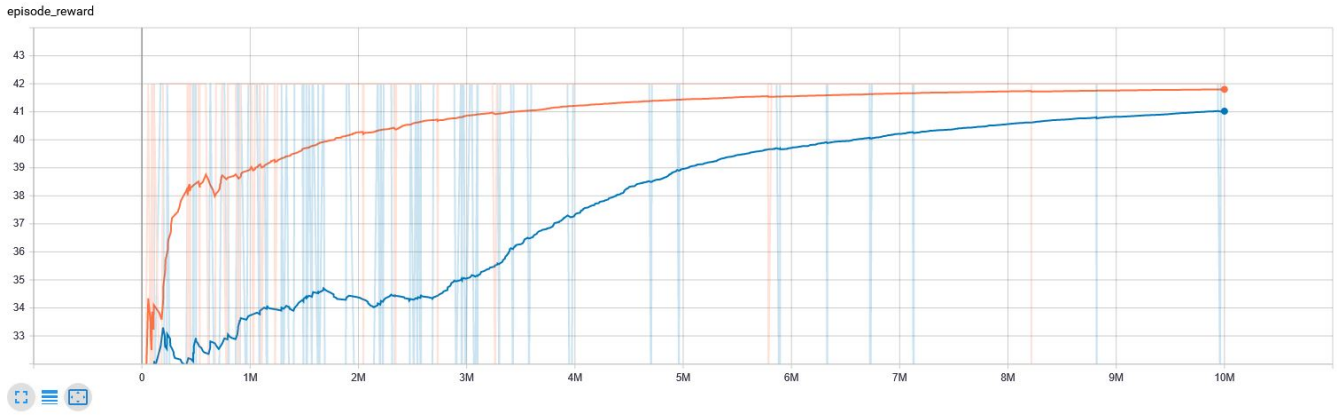

Figure 3: Episode Rewards A2C(Orange) and PPO2(Blue).

Meanwhile the computer, with its random choices, didn't fare nearly as well. It only managed to sink two of our agent's ships, mostly just firing at the open water. With eight remaining vessels our agent has bested its opponent in 121 turns, gaining a full reward of 42 points.

```
------------- End -------------
The Agent has won the game
Remaining Ships Computer: 0
Remaining Ships Agent: 8
Reward: [42.] /42
Turns: 121
```

Listing 10: Game information

Figure 3 shows a comparison of one agent of each type after a prolonged training of ten million timesteps. It appears that the A2C based agent (a2c) learned very rapidly up to a certain point and than steadily improved over time more slowly while the PPO2 model (ppo2) slowly ramped up over time eventually reaching a steady rate of improvement and slowing down again when approaching the maximum possible reward. After only about 500.000 steps our A2C based model already earned a reward of around 38-39. Meanwhile our PPO2 agent was only able to again a reward of 33 points.

At around 1.8 million time steps a2c came in with an average reward of about 40 points, thereafter steadily nearing the maximum reward of 42 points, while it took ppo2 around 6.5 million steps the achieve the same average.

With the reward per episode(figure 3) and the discounted reward (figure 6) constantly increasing the loss (figure 4) and the entropy loss (figure 5) for a2c diminished over time after first spiking to a starting value of 2.1 and 4.5 respectively. While the entropy loss of ppo2 also decreased with further training its loss hovered at a value of ~ 0.1.

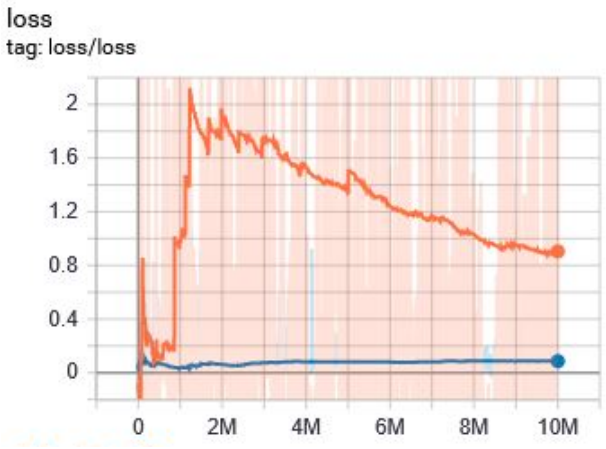

Figure 4: Loss

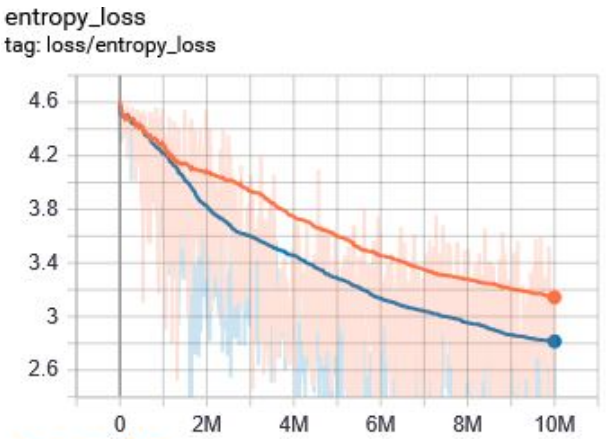

Figure 5: Entropy loss

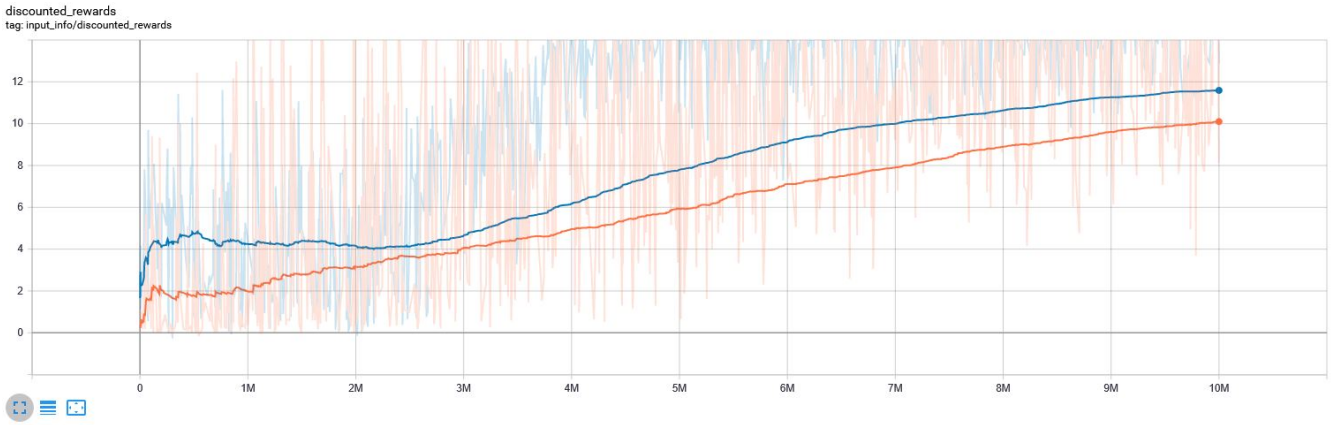

Figure 6: Discounted Rewards A2C(Orange) and PPO2(Blue).

8 CONCLUSION

From those results we can conclude that our A2C based model is more suitable to our environment and was therefore able to gain a greater proficiency at playing the game than our PPO2 based approach at a rapid pace.

9 FUTURE WORK

Currently there are no plans for further development, with leaves the future of this project as uncertain. Some possible approaches are as follows. Different types of agents with varying amounts of training could be put up against each other to display the differences in learning efficiency. An option for users to face of against there trained agents could also be implemented.

REFERENCES

[1] S. Ravichandiran, *Hands-on Reinforcement Learning with Python: Master Reinforcement and Deep Reinforcement Learning Using OpenAI Gym and TensorFlow.* Packt Publishing Ltd, 2018.

[2] J. Frochte, *Maschinelles Lernen: Grundlagen und Algorithmen in Python*, 2nd ed. München: Hanser, 2019.

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