

**ISTANBUL TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT**

**BLG 638E
DEEP REINFORCEMENT LEARNING
PROJECT REPORT**

504211549 BARIŞ CAN
504221529 ÖMÜR FATMANUR ERZURUMLUOĞLU
504211530 SADİ ÇELİK

SPRING 2023

Introduction

The project is a turn-based strategy simulation game designed for competing Deep Reinforcement Learning (DRL) agents. The objective of the game is straightforward: players aim to accumulate more resources than their opponents by the end of the game. This resource accumulation directly impacts the final score, which is calculated as the difference between the player's resources and their opponent's resources. By competing in this game, participants can explore and evaluate various DRL algorithms, architectures, and training methods to devise effective strategies and learn robust decision-making policies.

Environment

The game is implemented as a Gym environment, built on the 'gym==0.21.0' library. Gym is a widely used framework for developing and evaluating reinforcement learning algorithms. It provides a standardized interface for interacting with environments, making it easier to compare and evaluate different agents.

State Representations

The state representation plays a crucial role in enabling agents to perceive and understand the game environment. In the environment, the state consists of various components that provide information about the game state at a particular time step. These components include:

- **score:** An integer value representing the current score of both the player and the opponent.
- **turn:** An integer indicating the current turn number.
- **max turn:** An integer specifying the maximum number of turns in the game.
- **units:** A grid-like representation indicating the presence of units on the game map.
- **hps:** A grid indicating the health points of units on the game map.
- **bases:** A grid representing the locations of bases on the game map.
- **res:** A grid indicating the distribution of resources on the game map.
- **load:** A grid indicating the number of resources loaded on truck units.

Action Space

The action space defines the set of actions that agents can take in the game. In the environment, the action space is represented as a tuple of four elements: location, movement, target and train.







- **location:** Specifies the location of a unit (y, x) where the action should be performed on the game map.
- **movement:** Represents actions, with values ranging from 0 to 6. 0 means shoot or collect, 1-6 movement actions correspond to a specific direction.
- **target:** Specifies the target location (y, x) for the action, such as attacking or collecting resources.
- **train:** Indicates the type of new unit to be created, ranging from 0 to 4, with 0 representing no training and the other values representing different unit types (1 is train truck, 2 is train light tank, 3 is train heavy tank, and 4 is train drone).

Grid World and Units






The game environment is represented as a grid world, where units can move, collect resources, attack opponents, and interact with various elements of the environment. The grid world consists of hexagonal tiles, each containing different information, such as units, bases, resources, and terrain(mountain, water, mud, grass). The environment involves four types of controllable units:

- **Heavy Tank:** A powerful unit with high health points and attack capabilities.
- **Light Tank:** A relatively weaker unit with moderate health points and attack capabilities.
- **Truck:** A unit specialized in resource collection and transportation.
- **Drone:** A flying unit with unique movement abilities.

Each unit type has specific attributes, such as creation costs, health points and attack power which impact their effectiveness in different scenarios. Understanding the strengths and weaknesses of each unit is essential for developing effective strategies in the game.

Picture	Unit	Cost	Health Point (HP)	Accuracy Score
	Base	-	-	-
	Truck	1	1	-
	Light Tank	1	2	2
	Heavy Tank	2	4	2
	Drone	1	1	1
	Destroyed Unit	-	-	-

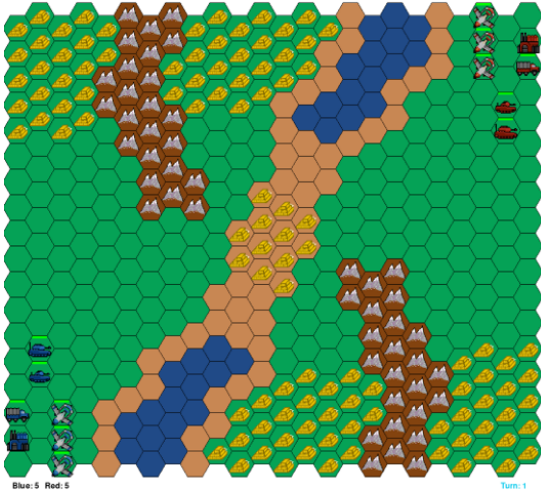
(a) Unit Details

Picture	Unit	Load Capability	Movement	Flying	Anti Air Attack	Heavy	Train Units
	Base	no	no	no	no	no	yes
	Truck	yes	yes	no	no	no	no
	Light Tank	no	yes	no	yes	no	no
	Heavy Tank	no	yes	no	no	yes	no
	Drone	no	yes	yes	yes	no	no

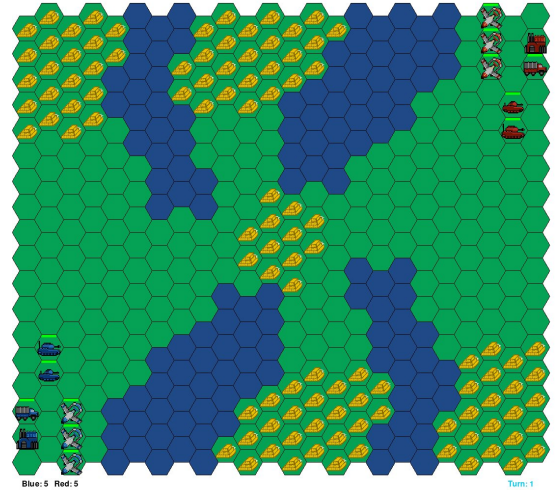
(b) Unit Abilities

Figure 1: Unit configurations available for each unit under data/config/rules.yml

Maps



(a) Risky Valley



(b) Risky Water

Figure 2: Maps consisting different terrains

Implementation

Model Code

For the training part we preferred sb3cadet.py which uses Stable-Baselines3 [3] backend due to its simplicity. Stable-Baselines3 is a Python library built on PyTorch [2] that provides reliable implementations of deep reinforcement learning algorithms. One of the key features of Stable-Baselines3 is its support for parallel (or "vectorized") environments, which can speed up training by allowing multiple environments to be run in parallel. Stable-Baselines3 also implements common environment wrappers, like prepro-

cessing Atari observations to match the original DQN experiments, to simplify training. The library contains state-of-the-art on- and off-policy algorithms, including A2C, PPO, DDPG, SAC, TD3, HER, and DQN. In addition to algorithms, the built-in Tensorboard support helped us to monitor our reward and loss functions. Since the game environment had a discrete action space, a suitable algorithm should be picked. Stable Baselines3 provides single process based DQN and multi-process based A2C and PPO. We chose A2C as our learning model. A2C (Advantage Actor-Critic) [1] combines elements of both policy-based and value-based methods. The code that imports the saved model to start learning with the pre-trained model was added.

Own Risk Valley Agent Code

In this study, RiskyValley agent inspired to us. The agent manipulates action that came from policy model where actor critic approach (A2C) output. The manipulation has some logic rules about movements. While creating the rules, we did not especially add random moves in order not to disturb the deterministic structure of the model itself. In this part we explain the rules.

We utilized A2C algorithm with MlpPolicy. Because our hardware issue, we needed to use only one environment this is actually not good ideally. Default hyper parameters did not changed. We mostly focused the manipulation rules. The rules were applied to output of MlpPolicy, the output represents action space.

In the our own risky valley agent, the action space for 34 movement and 1 train part was defined. Train part represents unit types for generating new unit. The movements represent the action of units. The original Risky Valley agent uses half of them as a target, however we used all of them as a action because we defined the targets by rules. If there are any enemy unit near our unit, we set its location as a target to defeat it. We aim that the policy model mostly learns truck action to collect resources. Therefore we started locations by trucks that means first outputs effect trucks.

We defined a movement for each unit. If our movements is less than units we added new moment and enemy. We set most frequent movement for unit type from the policy network. Sometimes our units died in this case we set as a no action and go on. We defined some rules for trucks. If the trucks on resources and their loads less than 3 the truck collect the resource or the truck on base and has any resource they bring their resources to base. If the truck is in dangerous, the truck keep their original action because the truck try to learn run away from the enemies.

If the unit is heavy tank and movement carry to regular area we keep the action otherwise we set movement to 0. We apply the rules for light tank similar to heavy tank except daub area. The light tank can move daub area.

Last and most important rule is shoot any enemies if it is possible. In this case we find the nearest enemies. The enemies distances is related the unit type. If the unit type is drone the most distance can be 1 because we see that drone can shoot only 1 distance succeeding. If the unit is tank then the distance will be 2. However we also check if our unit is heavy tank and the enemy is drone than we do not change original action because the heavy tank can not shoot to drone.

In order to truck to collect 3 golds and return to base safely we altered trucks movement and supported them with attack units to defend them. Our trucks can gather 3 golds and return to base accordingly.

We also generated train rules in addition to movement rule. In train rule, if there is no resource and the generated train is truck we generate light tank because in the case we do not need truck anymore. if the truck size more than 10 or more than the half of resources and generated unit is truck than we do not generate anything. If we do not generate truck and number of enemy heavy tank count is bigger than number of drones than we generate drones because drones can move any area and can defeat the enemy tanks and also heavy tanks cannot do anything to drones.

Learning Process

Two different ways of training the model were tried. One of models first of all was trained in RiskyValley map with 1M time steps, then it was trained in RiskyWaters map with 1M time steps. On the contrary, The other model first was trained in RiskyWaters map with 3M time steps, then it was trained in RiskyValley map with 3M time steps. While learning process, two models played against Simple Agent because it takes most intelligent actions among all given agents. The two models played 3 matches against each other.

The hyper-parameters for learning process:

seed	learning_rate	n_steps	gamma	gae_lambda	ent_coef	vf_coef
1450	7e-5	10	0.99	1	0.01	0.25
max_grad_norm	rms_prop_eps	env_seed	n_envs	log_interval		
0.5	1e-5	1	1	200		

Result

Model 1

The model firstly was trained in RiskyValley map, then RiskyWaters map with 1M time steps.

In Figure 3, grey line represents entropy loss of first step of model 1, orange line represents second step of model 1. Axis x is used for time step, and axis y is used for entropy loss value. Entropy loss is used to encourage the agent to explore by providing a loss parameter that teaches the network to avoid very confident predictions. So, in the second step the entropy loss decreased.

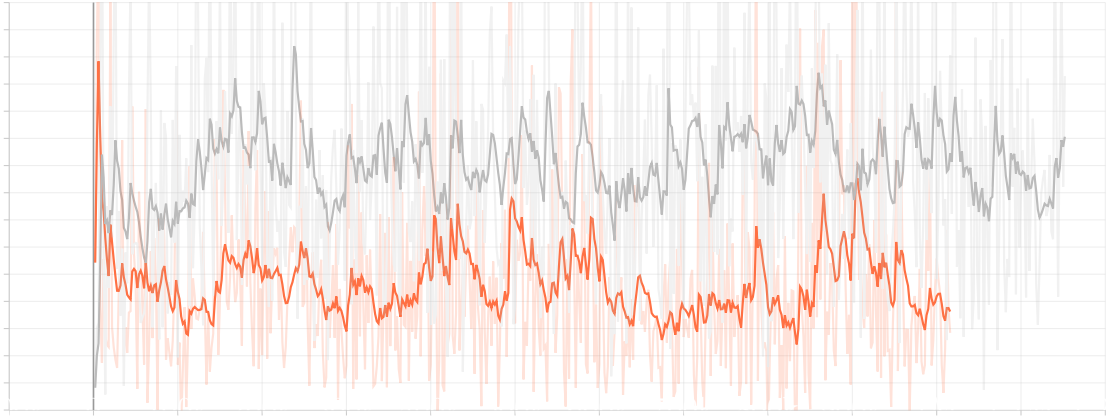


Figure 3: Entropy Loss Graph of Model 1

The grey line represents policy loss of first step of model 1, orange line represents second step of model 1. Axis x is used for time step as well. The Figure 4 was obtained by smoothing the actual policy values by 0.99. As seen in the figure, the policy tends to increase suddenly and decrease slowly.

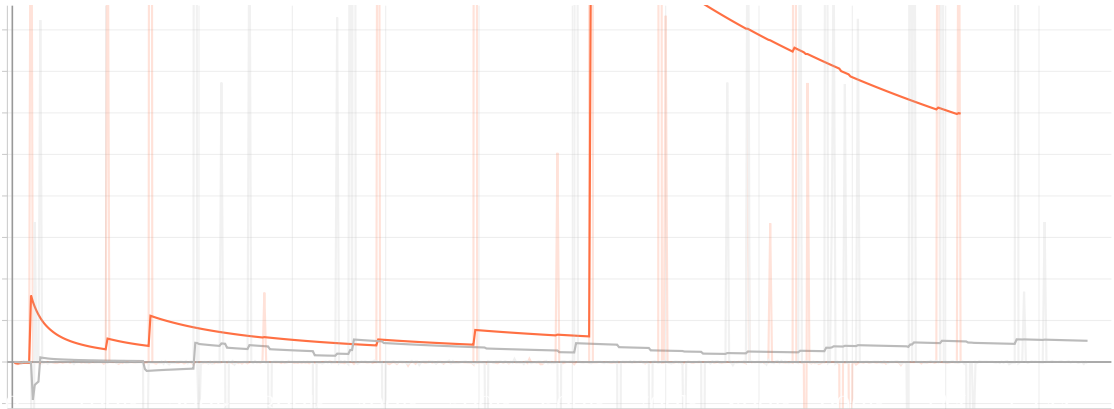


Figure 4: Policy Loss Graph of Model 1

The grey line represents value loss of first step of model 1, orange line represents second step of model 1. The Figure 5 was obtained by smoothing the actual value loss by 0.99 as well. As seen in the figure, the value loss tends to increase suddenly and decrease slowly. After the model was trained in RiskValley Map, higher poicy and value loss values were obtained in RiskWaters map than before.

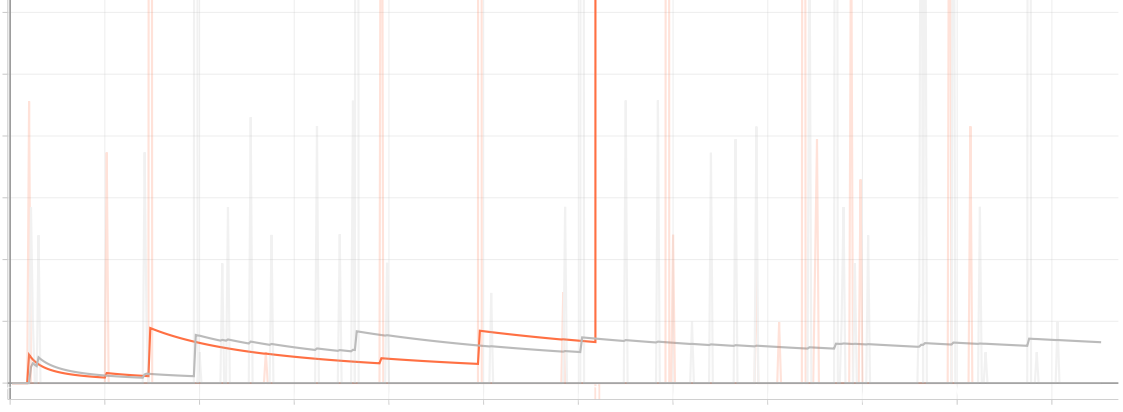


Figure 5: Value Loss Graph of Model 1

Model 2

The model firstly was trained in RiskyWaters map, then RiskyValley map with 3M time steps each time.

In Figure 6, blue line represents entropy loss of first step of model 1, orange line represents second step of model 1. Axis x is used for time step, and axis y is used for entropy loss value. Entropy loss is used to encourage the agent to explore by providing a loss parameter that teaches the network to avoid very confident predictions. So, in the second step the entropy loss decreased.

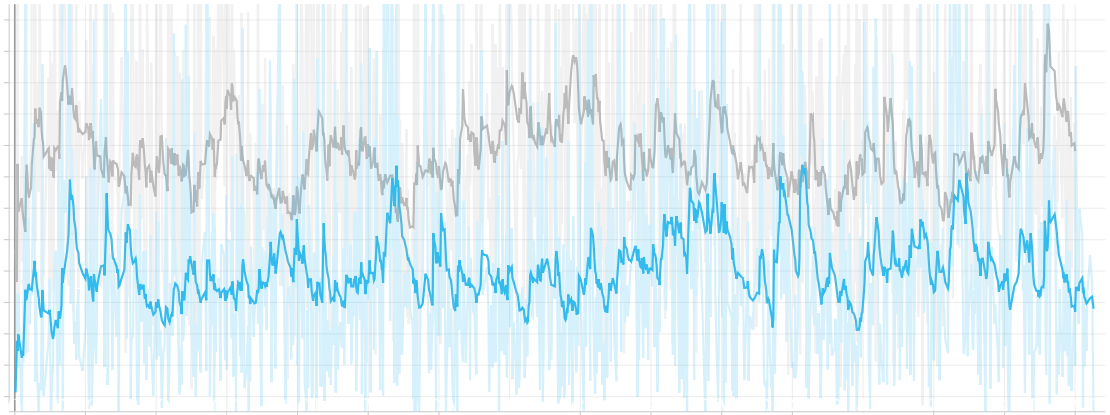


Figure 6: Entropy Loss Graph of Model 2

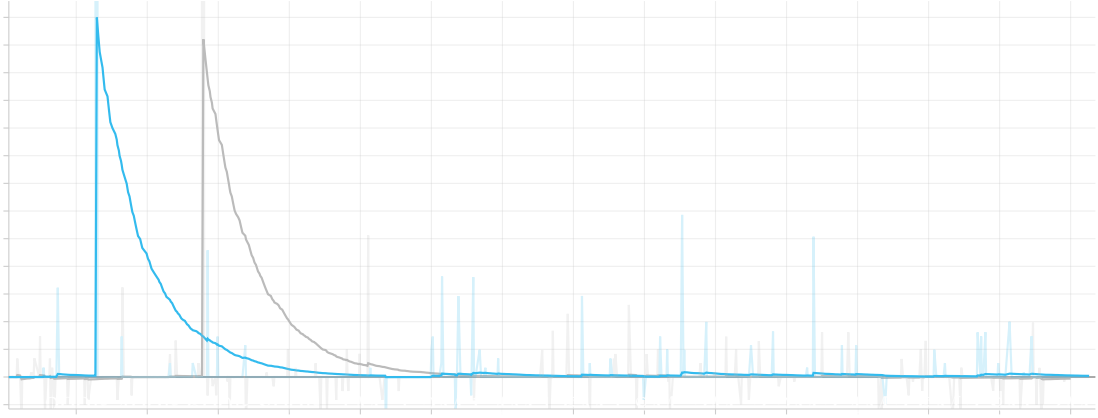


Figure 7: Policy Loss Graph of Model 2

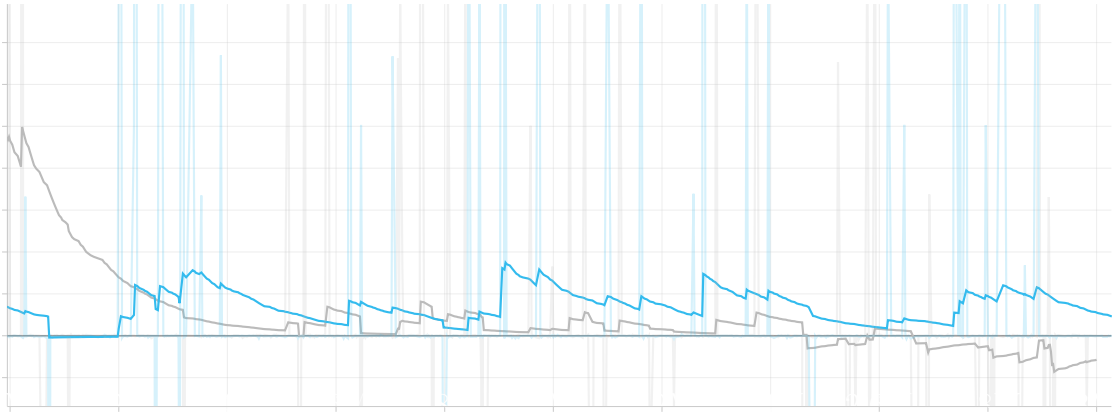


Figure 8: Policy Loss Graph of Model 2

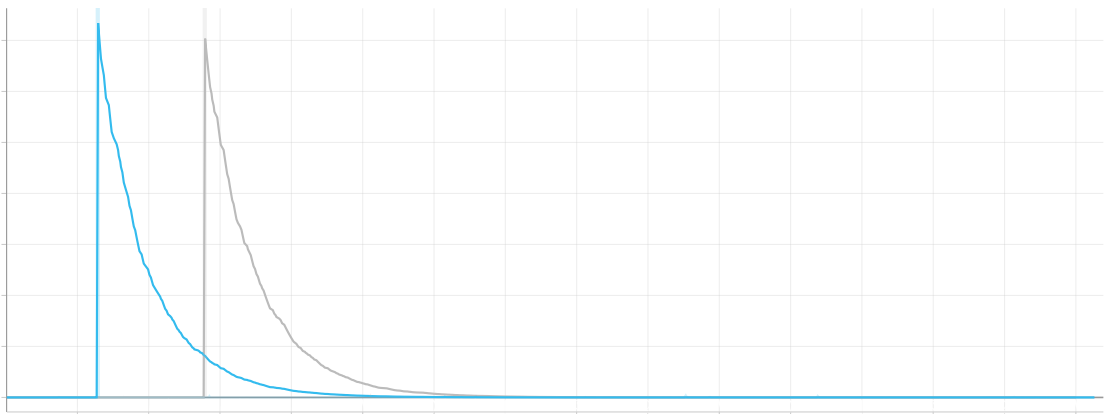


Figure 9: Value Loss Graph of Model 2

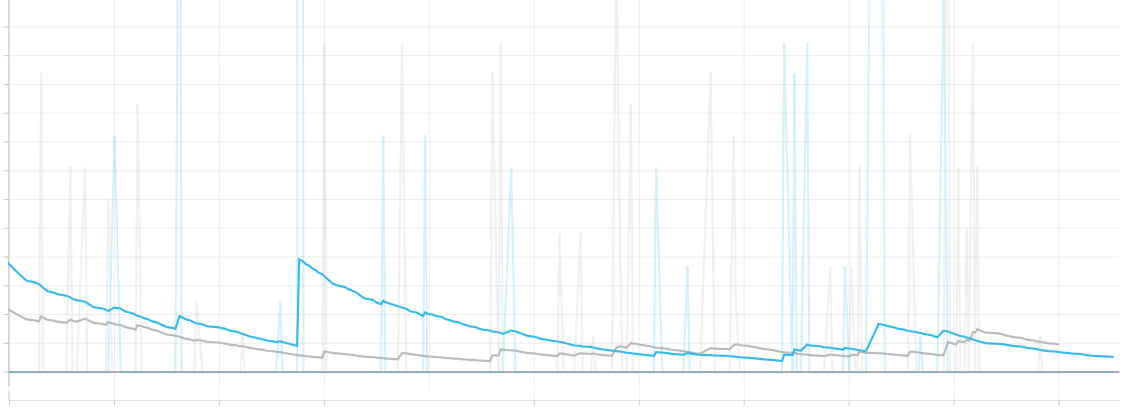


Figure 10: Value Loss Graph of Model 2

Run Config

In the competition please run "valley_3000000_steps.zip" agent.

The open source code is provided in Github Repository

Challenges

Some challenges were encountered during the implementation of the project. First of all, installing the WSL (Windows Subsystem for Linux) on Windows computers and downloading the necessary libraries to be compatible with the hardware parts of our computers occurred. We also needed to learn how to open and use efficiently Tensorboard. After providing the conditions that ensure running sb3cadet without error, the given agents were read in detail such as RandomAgent, SimpleAgent and RiskyValleyAgent. Inspired by RiskValley and Simple Agent, we set rules without disturbing the deterministic structure of the model. We made many changes to the rule list during the implementation process. After writing OwnRiskyValley agent, we got many errors in the debug process, and it helped us to close vulnerabilities in code. We had to spare 2 days for the learning process described in the Experiment part.

Conclusion

In conclusion, an agent was implemented by using deep reinforcement learning for turn-base strategy game. A2C was selected as the learning model with MlpPolicy. Instead of waiting for the agents to learn from the beginning, we gave some rules to the agent to obey before taking action. For example, truck unit must take the resource if it is on it.

With this rule, truck units do not need to learn "take the resource" action while playing. Hence, that facilitate the learning process.

References

- [1] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: <http://arxiv.org/abs/1602.01783>.
- [2] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [3] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: <http://jmlr.org/papers/v22/20-1364.html>.