A competitive 3D-Volleyball-Game using Reinforcement Learning with Unity ML-Agents

Sabrina Göllner
sabrina.goellner@haw-hamburg.de
Self-optimizing Systems
Winter Term 2021/22
Department of Computer Science
University of Applied Sciences Hamburg, Germany

Abstract

This project report describes the integration of reinforcement learning into a game development scenario by creating a competitive volleyball game using the Unity ML-Agents Toolkit. The work elaborates on what reinforcement learning is, brings forth some of the challenges of adding machine learning to a game, describes the development environment Unity and its machine learning package ML-Agents. Furthermore the implementation aspects are described including the environment setup, parameter tuning, reward engineering as well as the performance analysis of the trained agent.

Keywords: Unity ML-Agents, Machine Learning, Reinforcement Learning, Game Development, Self-Play

1. Introduction

Machine Learning (ML) is a sub-field of artificial intelligence (AI) in computer science. It deals with methods and algorithms used to train an AI-model to perform a task without explicitly programming how to perform that task. These models can learn just by observing existing data or by reinforcing favorable patterns. This field has had a tremendous impact on the world due to its potential utility in various fields of study, from medicine to economy. However, ML also has applications outside of science as it can also be used for entertainment purposes, as is the case with games.
This project explores the application of ML in the games sector. The goal is to create a competitive, volleyball-inspired game and use Unity's ML agents (Juliani et al. (2020)) to train a model that can play the game. A particular focus is on fine-tuning the hyperparameters and the reward function for the agent to improve its learning progress.
The goals of this project are as follows:

1. Develop a volleyball environment in unity ml-agents suitable for training a reinforcement learning agent

2. Train the agent to be competitive at the game: the agent should be able to score points, defend at the opponents attacks and ultimately win the game using its own strategy, which depends on the hyperparameters and rewards.

3. Evaluate the performance of the agent based on metrics and visual aspects.

This document is split into three main sections structured as follows: Section 2 gives an overview of reinforcement learning, self-play and other vital concepts used in developing the game. Section 3 describes the implementation process including the agents and their environment they are trained in as well as the development of the parameters finally used for the training process. Furthermore the results are described and discussed. The last section (4) concludes the achievements and lessons learned though the project and gives an outlook for possible enhancements.

2. Background

ML contains various types of learning paradigms namely supervised, unsupervised learning as well as reinforcement learning. The naming convention of supervised and unsupervised learning is a bit misleading – it suggests that all types of learning can fit under these two categories, which is not true (Sutton and Barto (2018)). The approach used in this project is called Reinforcement Learning (RL) and it has an entirely different approach to teaching a ML model (called agent in RL) how to perform a task.

2.1 Reinforcement Learning

Unlike supervised an unsupervised learning, Reinforcement learning does not use labeled examples to fit a model, nor does it use unlabeled data akin to that of unsupervised learning. RL is teaching an agent in an open-ended environment how to perform an action by rewarding wanted behavior and optionally punishing unwanted behavior. The actions to be used and when to use them are found out by trial and error and by observing the consequences of the actions in the given environment. By performing actions, the agent aims to find a behavior (policy) that gives it the highest reward. The policy is the agent's behavior function which tells the agent which action to take in the state. The agent also receives important information about its environment using observations, that get past through an ever-improving neural network, which outputs the next action the agent will take. This process gets repeated over an extended period of time to train the agent (Sutton and Barto (2018)).

2.2 PPO

This project uses the Proximal Policy Optimization (PPO) method to optimize the neural network during training. PPO is a policy gradient method and can be used for environments with either discrete or continuous action spaces. It trains a stochastic policy in an on-policy way. Also, it utilizes the actor critic method. The actor maps the observation to an action and the critic gives an expectation of the rewards of the agent for the observation given. Firstly, it collects a set of trajectories for each epoch by sampling from the latest version of the stochastic policy. Then, the rewards-to-go and the advantage estimates are computed in order to update the policy and fit the value function. The policy is updated via a stochastic gradient ascent optimizer, while the value function is fitted via some gradient descent algorithm. This procedure is applied for many epochs until the environment is solved (Schulman et al. (2017)).

2.3 Unity ML-Agents

Unity is a 3D development software used widely for making games and other interactive experiences. As described by Juliani et al. (2020), ML-Agents is an add-on for the existing Unity platform. It is an open source toolkit and aims to provide researchers and game developers with the ability to build complex 3D environments and train intelligent agents in them. The ML-agents toolkit consists of three main components: the agent, the environment, and available actions for the agent. Agents share their experiences during the training. Their policy must access player inputs, scripts, and neural networks through Python. For this purpose, the agent uses the Communicator to connect to the trainers through the Python API (see figure 1).

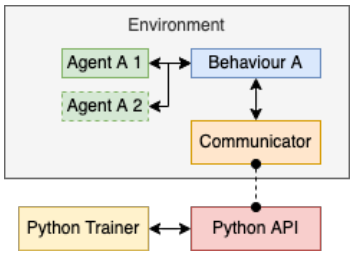

Figure 1: Architecture (based on Juliani et al. (2020))

2.4 Self Play

Giving the Agent a sufficient challenge is a common problem when using ML for adversarial play. If the opponent is too strong, then the agent will have a hard time understanding how it should play since most attempts would lead to failure. Conversely, if the agent can beat its opponent with minimal effort, it will have no reason to improve further or develop new, interesting strategies. Furthermore, one cannot assume that there will always be a human opponent at hand - apart from that, this also means an enormous effort to actively engage with the AI.

Therefore competitive self-play involves training an agent against itself. Self-play is a reinforcement learning technique used commonly in adversarial games that pits the training agent against its previous versions. After training the agent for a while, a snapshot is saved of the agent at that point. This snapshot can then be applied to the opposing player while training the agent. Furthermore, since we can choose which snapshots of the agent to save into the pool of versions to use for training, we can also ensure a balanced training environment. If we use recent iterations of the agent, both players will be of equal skill level, giving the agent the best environment for further improvement. By playing increasingly stronger versions of itself, agents can discover new and better strategies. Self-play can also combat overfitting by saving multiple snapshots of the agent (Cohen (2020)).

It was used in famous systems such as AlphaGo and OpenAI Five (Dota 2) (Silver et al. (2017, 2018))

2.5 Parallel Training

In Unity, it is possible to set up parallel training by making copies of the environment. Each simulation environment feeds data into a common training buffer, which is then used by the trainer to update its policy in order to play the game better. This new paradigm allows to collect more data without having to change the timescale or any other game parameters which may have a negative effect on the gameplay mechanics. In addition making use of asynchronous parallel environments for environments with varying step times can significantly improve sample throughput and speed up training (Erwin (2019)).

2.6 Metrics

One of the metrics and possibly the most flexible one is cumulative reward. Cumulative reward shows how much reward the agent was given during one attempt (also referred to as an episode in the ML-agents package) (Juliani et al. (2020)). However, the cumulative reward as a metric fails in specific self-play scenarios. Since self-play involves the agent playing against recent iterations, the players will be equal in skill most of the time, leading to many games ending in a tie. The agent might be improving, but it is not reflected in an increase in cumulative reward.

ELO The ELO rating system is a metric to observe in a self-play environment. It considers each player's relative skill level and is commonly used in ranking chess players or sports teams and it can also be used to predict the outcomes of matches. As the model improves during self-play, the ELO rating also increases (Juliani et al. (2020)). There are many methods for calculating ELO rating and the following will outline one of these methods: Given two arbitrary players' ELO scores R_A and R_B, calculate each players' expected result. This value ranges from 0 to 1, with 0 meaning defeat and 1 meaning victory. The expected result for player A would be:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

Once the game between players A and B has been concluded, compare the expected results to the actual result (which is either 0 or 1 for a loss or win respectively). Adjust each player's ELO rating with the following formula:

$$R'_A = R_A + K(S_A - E_A)$$

Here, S_A is the actual result of player A. K is a constant that determines the rate at which ELO changes per game played. This value is adjusted to fit the game best. A common value used in chess is K = 32 (Aversa (2019)).
However, in the case of ELO, the rating on its own does not give any substantial information about the performance of the model since it is not referencing anything – it is a relative metric. Given some arbitrary ELO rating, that rating is only useful after the respective model has been observed and its performance evaluated by a real person.

Episode Length The length of each episode is also a useful metric for determining the performance of a model, but it requires proper domain knowledge before it can be interpreted. Using the game of this project, an increase in episode length signifies an im-

provement in the model, since that means the players can keep the ball in play for longer. However, examining real life professional volleyball, each individual point is determined rather quickly – it does not take noticeably longer for a point to be earn at higher levels of play, as opposed to beginner or intermediate levels. Knowing this, it is important to consider if this also applies to our volleyball game. Perhaps as the model improves, the total length of each episode will still stay the same. An increase in episode length might not also mean the model is improving. Consider a model that is learning how to push a box to some specific location. At the start of training, the model will struggle with completing this task and the episodes will be long. Conversely, as the model trains and improves, they will complete their task faster and faster.

3. Implementation Details

The implementation in this project is built with Unity in a 3D Environment and makes use of the features mentioned in section before. The Implementation process itself consisted of several goals:

1. RL-Environment setup: Creating the game itself, set all parameters of the environment and build up a working system.
2. Hyperparameters: Setting and tuning the hyperparameters of the setup for optimal training, also the number of environments in unity to speed up training process.
3. Reward engineering: fine-tuning the reward function so that the agent plays strategically and adapts behavior to win the game.

In the following, their implementation is described.

3.1 RL-Environment Setup

The setup of the game itself is as follows: The game environment is a large rectangular room with a small green wall in the middle, that acts as the net in volleyball. Two player characters (agents) are placed on either side of the net an equal distance away. They can only move around on their side of the net. A ball is set in play that both players (agents) can interact with.
The goal of volleyball is to play the ball in a way that it touches the ground on the opponent's side of the court and scores a point for the player. Another goal is also to ensure that one's own side of the court is protected from the attempts of from the opponent's attempts to score a point. A goal was visually marked by coloring the floor in the color of the winner. The figure 2 shows the basic setup:

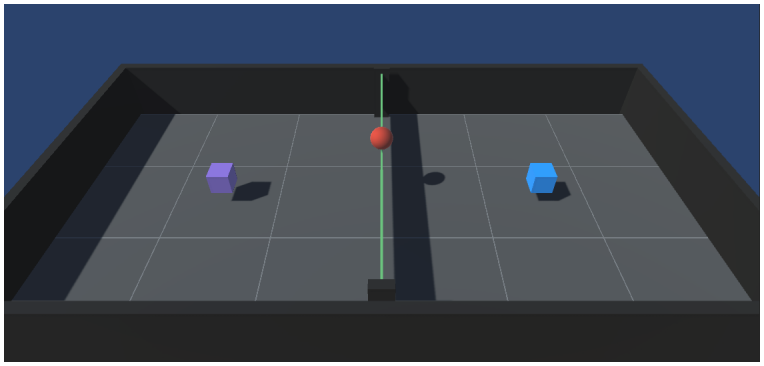

Figure 2: Image of the Environment and the agents in Unity

3.1.1 Observation Space

Observations are how the agent 'perceives' its environment. In ML-Agents, there are 3 possible types of observations: Vectors, Raycasts or camera input.
Vectors provide "direct" information about our environment (e.g. a list of floats containing the position, scale, velocity, etc of objects). In this project, the vector observations were used for simplicity reasons. The goal is to include only the observations that are relevant for making an informed decision about how the agent should act.
After some testing, this setup was found useful for observations:

- Agent's y-rotation [1 float],
- Agent's x,y,z-velocity [3 floats],
- Agent's x,y, z-normalized vector to the ball (i.e. direction to the ball) [3 floats],
- Ball's x,y,z-velocity [3 floats],

This is a total of 11 vector observations. It should be noted, that that the goal was not to replicate a real world volleyball scenario since these observations won't make sense. It would be very unlikely for a player to 'know' these direct values about the environment.

3.1.2 Action Space

The action space defines the set of possible actions an agent can perform during training. At a high level, the ML-Agents "Decision Requester" will select an action for the agent to take. In Unity these sets are called 'branches'. In this scenario we have 4 branches: Two for moving along the x and z axis, one for rotation and one for jumping.

- No movement, move forward, move backward
- No movement, move left, move right
- No rotation, rotate clockwise, rotate anti-clockwise
- No Jump, jump

3.2 Hyperparametertuning

The next step was setting up the hyperparameters of the setup for optimal training. Also the number of environments in unity play an important role to speed up the training process and the self play parameters.

3.2.1 Configuration

In ML-Agents there is the possibility of setting configurations to optimize the training process: These parameters are seperated into general confgurations, network configurations, hyperparameters, ppo-specific configurations, and self play configurations.

General Settings The agent was trained for a `max_steps`-size of `20 million` using `5` model checkpoints to keep through the whole training run.
The `time_horizon` was set to `1000` to be large enough to capture all the important behavior within a sequence of an agent's actions. This is collected per-agent before adding it to the experience buffer.

Hyperparameters The `buffer_size`, which corresponds to how many experiences should be collected before doing any learning or updating of the model, was set to `20480`. It was recommended to be multiple times larger than `batch_size`, which was set to `2048`. Typically a larger `buffer_size` corresponds to more stable training updates. The `learning_rate` defines the strength of each gradient descent update step. For reasons of training stability this value was decreased to `2e-4`. According to the recommendations of the documentation, the learning rate schedule was first set to `linear`, which decays learning rate until `max_steps` so learning converges more stably. Since this did not lead to training improvement, the learning rate was constantly kept the same during the whole run.

Network settings In the network are 2 layers present, which is suitable for moderate problems. The number of units in each of these fully connected layers of the neural network was set to `256`. Since we have more interactions between the observation variables, this was increased. The encoder type, for encoding visual observations was set to `simple`, which uses a simple encoder which consists of two convolutional layers with a kernel size of `20x20`.

PPO-specific configuration The `beta`-value, the strength of the entropy regularization, which ensures that agents properly explore the action space during training, was set to `3e-3`, since this lead to slowly decreasing entropy alongside increasing reward. `epsilon`-value influences, how rapidly the policy can evolve during training. Setting this value to a smaller value of `0.15` resulted in more stable updates, but also slowed down the training process. The regularization parameter `lambd` was decreased to `0.93` for a more stable training process.

Rewards The parameters here allow the developer to tune the balance between `intrinsic` and `extrinsic`-reward signal strengths. Extrinsic rewards signals are the reward signals set up by the developer and are specific to the model being trained and its environment. For example, an extrinsic reward signal in volleyball would be scoring a point or properly serving the ball over the net. The both variables set here were first the factor by which to multiply the reward given by the environment (`1`) and second, the gamma-value (discount factor), for how far into the future the agent should care about possible rewards. Since the

agents in this scenario should be acting in the present in order to prepare for rewards in the distant future, this value was set to maximum (`0.99`).
Since intrinsic rewards (curiosity) are only needed in sparse rewarded environments, this was not included here.

Self-play parameters The final aim and objective of this project was to train *competitive* agents that are rewarded for winning. Therefore the self play parameters (introduced in section 2) played an important role for the configurations. In this section this will be dealt with in more details:
To throw the ball over the net, the agent must take into account the trajectory of the ball flying from the opponent, and make an adjustment to the angle and speed of the flying ball, taking into account environmental conditions (gravity). However, as already mentioned considerations for a suitable opponent are not trivial therefore *self play* was taken into account. From looking at the setup in figure 2 the blue cube (right) is the defined as the learning agent, and the purple (left) is defined as its fixed policy opponent.
Additionally setting the `save_steps` parameter, a snapshot of the learning agent's existing policy was taken. Up to 10 snapshots will be stored in the `window` parameter. When a new snapshot is taken, the oldest one is discarded. These past versions of itself become the 'opponents' that the learning agent trains against. Every 10000 steps, the opponent's policy was be swapped with a different snapshot (`swap_steps`). The snapshot is sampled with a probability of 0.5 (`play_against_latest_model_ratio`), that it will play against the latest policy (i.e. the strongest opponent). This helps to prevent overfitting to a single opponent playstyle.

Parallel Environments At the beginning 10, then 20 and then 50 copies of the environment were used. After successful testing, 100 parallel environments were used in the end to actually reduce the training time by half (from 16 training hours to 8 training hours).

3.3 Reward Engineering

In reinforcement learning, the reward is a signal that the agent has done something right. The better the reward mechanism, the better the agent will learn. In Unity it is possible to allocate rewards to an Agent by calling the `AddReward()` or `SetReward()` methods on the agent. The reward assigned between each decision should be in the range [-1,1]. Values outside this range can lead to unstable training. This information was considered during the reward engineering process.

3.3.1 Aspects

Through observations in the learning environment, aspects were considered, which are shown in the following visualization. The red border shows the boundary of the environment. The purple and blue squares are representative for both agents. In the middle there is a green line representing the green wall, over which the agents have to play the ball. The ball is the black circle in this case. The possible rewards (green) or punishments (red) shown here are from the perspective of the blue agent. That is why the bottom of the blue agent is also colored red and the one on the opposing side in green.

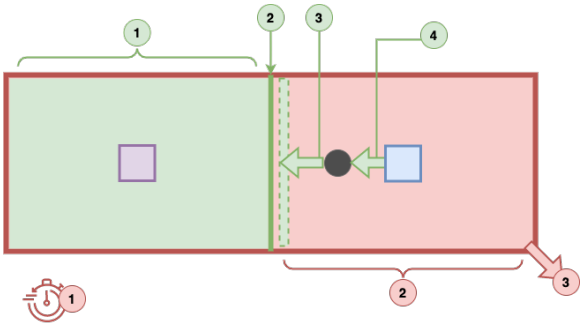

Figure 3: Reward Engineering using domain knowledge

Positive Rewards (green):

1. The ball hits the floor of the opponent's playground (winning)
2. The agent plays the ball over the wall
3. The agent plays the ball towards the wall
4. Agent hits the ball

Negative Rewards (red):

1. Adding a very small penalty over time (Unity suggested adding "an additional per-timestep penalty of -0.0003 to encourage agents to score". (Cohen (2020)))
2. The ball hits own floor (loosing)
3. The ball is hit out of bounds

After setting the simplest possible reward function (`+1` winning, `-1` losing) in the first approach several tests were performed to get the best result using the aspects from above. The best parameter set was achieved with the following settings: The agents Reward was set to `-1` for winning and `-1` for loosing. If the agent plays the ball in the are near the green wall a very small reward (`+0.02`) was added, and another, a bit bigger reward for playing over the wall itself (`+0.2`). For just hitting the ball a very small reward was set (`+0.0002`) just to encourage the hitting action and therefore to learn faster. A small penalty was added for playing the ball 'out of bounds', e.g. outside the court (`-0.001`). The values in parentheses are the very final values, which were used after intensive testing for getting the best result.

3.3.2 Results

The next figures show some diagrams of the most mentionable test runs using the tensorboard:

ELO Figure 4 shows three test runs are mentionable for comparison here:

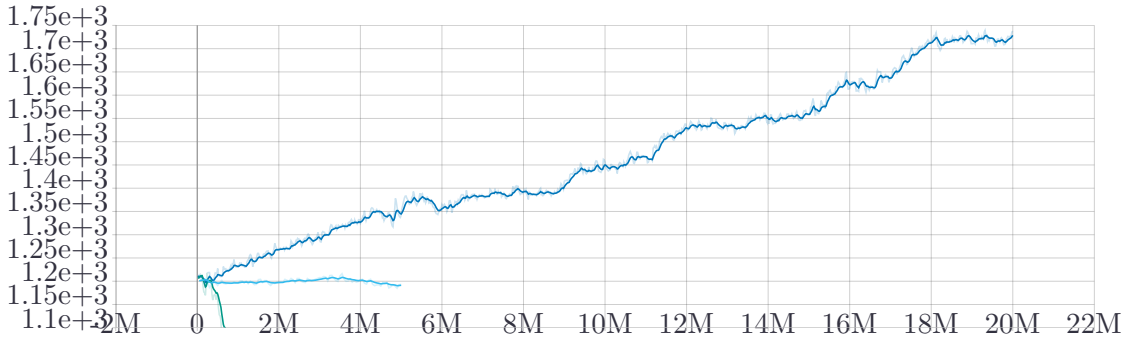

Figure 4: Agents Learning Curve using the ELO metric (per step, smoothed by 60%)

- Green line: represents a failing test run adding a negative reward each timestep and was used for testing Unity's recommendation. This was removed after the test run, since this led to very poor performance and did not work as expected.
- Light blue line: shows the initial test with the initial unity ml-agents-parameters to get an impression of the initial performance. The agent did not improve at all, so the whole parameter set needed to be reconfigured (as expected).
- Dark blue line: This is the best test run using the fine-tuned hyper-parameters mentioned in the last section showing a learning curve of a well performing agent.

Episode length The graphs in figure 5 show the episode length of each test run.

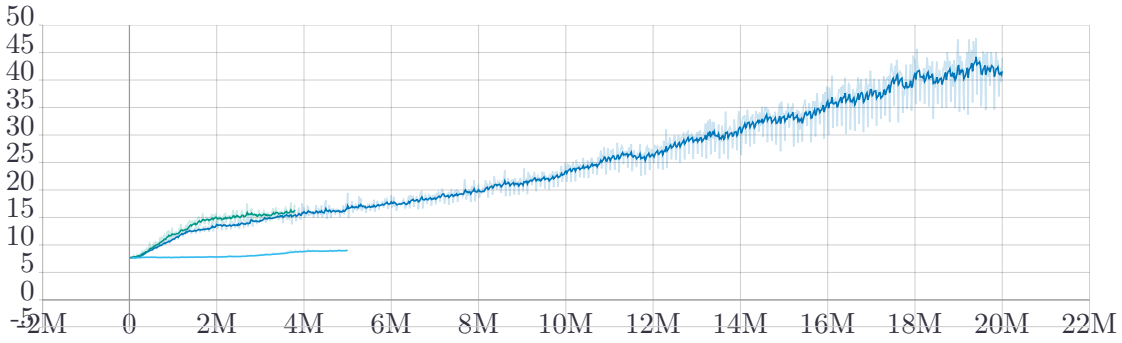

Figure 5: Episode Length (per step, smoothed by 60%)

- Green line: Even with adding penalty, the episode length increased first, but as the agent did not improve according to the ELO metric, this was not usable though.

- Light blue line: The episode length did not change over time.
- Dark blue line: In the final test run the episode length steadily increases, while the agents performance also increased (see figure 4). Despite the high performance, the variation in episode length increases towards the end. But this was interpreted in such a way, that the agents learn to serve more shots which are harder to return, leading to this effect (for example it happened that the opponent was not able to return the ball, which also happens in realistic volleyball).

Of course, more test runs were performed to improve the performance of the agent step by step, but only the most interesting test runs are shown here.

Trainingvideo The best way to see this process of improvement is also to watch the short video attached. Finally one can see that the agents try to return interesting shots and have developed some kind of 'tactics' to make the game more difficult for the opponent.
The Additional reason for watching the improvement process visually was, because the ELO rating is a relative metric and (as mentioned before in the section 2) it is only useful after the respective model has been observed and its performance evaluated by a real person. ELO was good simply for identifying that the model *is* improving, not necessarily for identifying whether the model was good or not. Therefore the training process was observed visually, too.

4. Conclusion

In this project, a competitive 3D volleyball game was designed using reinforcement learning. The agent was trained to play against its past snapshots, which provide a suitable challenge to become better. The ML-Agents library was used to support the implementation of the goal, as it provides a good learning environment for intelligent agents. First, the focus was on achieving an optimal learning result through suitable parameters, after which the implementation of the actual project was described. At the end, besides the description of the learning curve and its results, a video of the whole successful training process was recorded to show again the best improvements and milestones of the learning process.
For future improvements to the project, it can be considered that the agent can play against a real human as its opponent.
This conclusion also includes a few lessons learned, which will be listed here. It is helpful to have some experience with Unity and its behaviours itself before working with ML-agents. To come to the ML aspects, the environment must not be too complicated if you do not have enough knowledge of the matter. Giving rewards is a real difficulty of its own in RL and must be well thought out so that the learning effect is also optimal. For example, unsuitable or too few observations from an agent do not lead to learning success. Finally, the convergence process should be given time, because training a good model can often take hours, if not days. If these things are kept in mind, it is really fun to work with it and build up RL-projects.

References

Davide Aversa. The elo rating system, 2019.

Andrew Cohen. Training intelligent adversaries using self-play with ml-agents. `https://blog.unity.com/technology/training-intelligent-adversaries-using-self-play-with-ml-agents`, Accessed on 01.03.2022, 2020.

Teng Erwin. Training your agents 7 times faster with ml-agents. `https://blog.unity.com/technology/training-your-agents-7-times-faster-with-ml-agents`, Accessed on 01.03.2022, 2019.

Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. *arXiv:1809.02627 [cs, stat]*, May 2020. arXiv: 1809.02627.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. July 2017.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550 (7676):354–359, October 2017. ISSN 1476-4687. doi: 10.1038/nature24270.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018. ISSN 0036-8075, 1095-9203.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.