Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

One Intelligent Agent to Rule Them All

Bachelor Thesis

Laurin Paech

lpaech@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

August 21, 2018

Acknowledgements

I would like to thank my Supervisor Manuel Eichelberger for his ideas, inputs and support, the PySC2 Community for their support and DeepMind that created such a great opportunity for research.

Abstract

In this work, we explore the possibility of training agents which are able to show intelligent behaviour in many different scenarios. We present the effectiveness of different machine learning algorithms in the *StarCraft II Learning Environment*[1] and their performance in different scenarios and compare them. In the end, we recreate DeepMind's *FullyConv* Agent with slightly better results.

Contents

Chapter 1

Introduction

One of the biggest questions for mankind is the question of intelligence.
The key to this seems to be learning itself.

While animals, when trained, can learn also quite impressive feats, even rudimentary speech[2], they lack the generalized learning nature of humans. Thats why the field of Artificial Intelligence (AI) studies learning and how to build algorithms that can even transcend human capabilities. The applications are endless. From medical diagnosis, robot control to even education and finance [3][4][5] [6].

In the history of AI, games were commonly used as benchmarks for intelligence. There are several advantages of using them. First, repeatability. We can repeat every experiment in precisely the same environment. Second, games have a clearly defined objective. Third, games are difficult even for humans. In total, they offer a unique means of measuring intelligence through task-based comparisons.

One of the first self-learning programs and an early demonstration of the concept of artificial intelligence dates back to Arthur Samuel developing a Checkers program in 1959 that trained itself using reinforcement learning[7]. In 1995, Gerry Tesauro built a program called TD-Gammon[8] that, with little backgammon knowledge, and also using reinforcement learning, learned through self-play to play near the level of the world's strongest grandmasters and in 1997 IBM's Deep Blue defeated World Chess Champion Garry Kasparov[9].

After chess, the next challenge became the game of Go. With its enormous state space an AI being able to compete against humans has to be able to learn in a different way compared to Deep Blue that used mere brute force when searching for the optimal action in a certain state.

Next, DeepMind developed novel Deep Reinforcement Learning algorithms, that generalize Reinforcement Learning and thus was able to defeat Lee Sedol, world's top Go player with their new algorithm called *AlphaGo*[10].

With these recent developments and the development of novel algorithms achieving human-like or even super-human results, Deep Reinforcement Learning has gained a lot attention of the artificial intelligence research community.

Subsequently, DeepMind announced StarCraft II [11], a popular real-time strategy video game, as their next research target. StarCraft II is considered highly complex. Even for human players becoming able to remotely master the game takes years of experience. While Backgammon, Chess and Go are games of perfect information, as each player has instantaneous knowledge of all moves in the game, StarCraft II is not. Additionally, the game consists of a vast state- and action-space, the need to adapt fast to complex situations and requires a high level of logical inference.

By creating the StarCraft II Learning Environment (SC2LE) DeepMind opened up a new benchmark for reinforcement learning research. Together with SC2LE, DeepMind published a paper describing a baseline and several mini-games that were created to represent subsets of the full game[1].

In this thesis, we use SC2LE to explore the possibility of training agents, which are able to show intelligent behaviour in many different scenarios utilizing the mini-games. We start off with vanilla and dueling Deep Q-Network (DQN)[12]. Next, we moved on to recreate the baseline and improve on it, trying to find its flaws and strengths as early DQN versions are to inefficient. As part of the evaluation of the baseline agent, we created a scripted agent for the minigame *CollectMineralShards* that achieves near optimal results.

Chapter 2

Background

In this chapter, we give the technical background that is necessary to understand the rest of the thesis. We start off with a general overview over Reinforcement Learning and reinforcement methods. Then we briefly talk about Function Approximators like Artificial Neural Networks and Convolutional Neural Networks and how to combine them with Reinforcement Learning to the area of Deep Reinforcement learning. In the end we give a brief summary of StarCraft II and the Python API PySC2 [13] we used.

2.1 Reinforcement Learning

Reinforcement Learning is a behavioristic-psychology-inspired subdomain of Machine Learning and can loosely be defined as learning from interactions to achieve a goal [14].

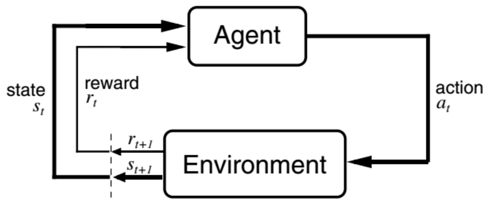


Figure 2.1: Agent interacting with environment [14].

It consists of an agent that interacts with an environment that is in a specified state by taking actions to maximize a cumulative reward over time (see Figure 2.1). We define state as the information about the environment that is available to the agent. By interacting with the environment, the agent constructs a mapping from states to probabilities of taking action a in state s. This is called the policy and is defined for a time step t as

$$\pi_t(s, a) = P(a_t = a | s_t = s)$$

The policy defines which action a is chosen in a state s.

In general, the objective of a Reinforcement Learning Problem is to maximize the expected reward by learning an optimal policy π^*. At each timestep the reward is a number returned from the environment after an action has been taken. The return R_t is defined as the cumulative reward from a state s_t until a terminal state s_T is reached.

$$R_t = r_{t+1} + r_{t+2} + ... + r_T$$

Here, we will only focus on finite, episodic tasks but continous tasks are also possible. We also introduce the discount factor γ, that discounts future rewards, such that the agent prefers earlier over future rewards:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

$$0 \leq \gamma \leq 1$$

Therefore a reward k time steps in the future is only worth $\gamma^k - 1$ times as much.

We usually assume the Markov Property for an environment, that is the probabilistic behaviour of the future states does only depend on the present state [15]. For the Markov Property holds:

$$P(s_{t+1} = s, r_{t+1} = r | s_t, a_t, r_t, ..., r_1, s_0, a_0) = P(s_{t+1} = s, r_{t+1} = r | s_t, a_t)$$

where $s_t, a_t, r_t, ..., r_1, s_0, a_0$ are past states, actions and rewards. A sequence of states, actions and rewards is also called a trajectory.

Intuitively, the Markov Property can be described as that the information of the current state combines all the information of the past states. This enables the agent to predict the next state and the expected reward based on the current state. A task that has the Markov Property is called Markov Decision Process (MDP) [16]. MDPs are defined by their finite set of states and actions, their transition probabilities $\mathfrak{P}^a_{s,s'} = P(s_{t+1} = s' | s_t = s, a_t = a)$, the expected reward $\mathfrak{R}^a_{s,s'} = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s]$, and the discount factor γ. Additionally, the notion of the MDP can be generalized to the partially observable Markov decision process (POMDP), which assumes that the process is modeled after an MDP but the agent can only partially observe the state. Thus the agent has to take actions with uncertainty of the full state of the environment.

To evaluate policies, the expected return has to be known. We define the notion of state-value function (Value Function). The expected return when starting in state s and following policy π is defined as the value of the state s:

$$V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s]$$

Informally, the value function describes how good it is to follow the policy.

A relationship between the value of a state and the values of its successor states can be defined in terms of the expected reward and transition probabilities. This is called the Bellman Equation [17] of the Value Function:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s}^{\prime} \mathfrak{P}^{a}_{ss'} [\mathfrak{R}^{a}_{ss'} + \gamma V^{\pi}(s')]$$

The Bellman Equation can be used to learn the Value Function of a Policy.

Similiarly, the notion of action-value function (Action Function) can be defined, where the agent takes an action a in a state s and thereafter follows the policy:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a] = \mathbb{E}_{\pi}[r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a]$$

Informally, the action-value function defines the value of taking an action a in state s.

An optimal policy π^* is defined as having the highest expected return compared to other policies for all states. In other words, the optimal policy has the optimal Value Function $V^*(s)$. The Bellman Equation of the optimal state-value function is defined independent of a specific policy and instead in terms of the best action that can be taken in that state.

$$V^*(s) = max_a Q^*(s,a) \tag{2.1}$$

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}_{s'}[r + \gamma max_{a'} Q^*(s',a') | s,a] && (2.2) \\ &= \sum_{s'} \mathfrak{P}^{a}_{ss'} [\mathfrak{R}^{a}_{ss'} + \gamma max_a Q^*(s',a')] && (2.3) \end{aligned}$$

Every policy that acts greedily with respect to the optimal state-value function is then an optimal policy.

Reinforcement Learning suffers from 2 distinct problems [14]:
First, the Exploration-Exploitation-Trade-Off refers to the problem of choosing between an seemingly rewarding action (exploitation) or deviating to explore more of the reward structure of the environment (exploration).
Second, the Credit Assignment Problem refers to the problem of delayed rewards. In environments with large action- and state-space, an action taken early can result in an reward far into the future. Difficulties arise to assign the reward to the correct action.

In Reinforcement Learning, we either optimize the policy directly by parameterizing it or we optimise the policy by maximizing the Value Function and subsequently evaluate a policy. These are Policy Gradient Methods and Value-based Methods and in the following subsections we will briefly describe them.

2.1.1 Value-based Methods

In general, Value-based methods attempt to find a policy that maximizes the return by first learning a state-value function or action-value function and then improving the current policy.

For example, if a model is available, the Bellman Equation for the Value Function can be used to subsequently approximate the Value Function for a policy by iteratively updating [18].

The Q Values can then be determined and the policy improved by always taking the action-value a that is highest in state s. By the Policy Improvement Theorem, this leads to an optimal Policy [17].

In practice there is commonly an absence of a concrete mathematical model and therefore no transition probabilities and expected rewards are available and a Value Function is not sufficient to get a policy. Instead, the action-value function is used. An agent learns the values of state-action pairs from experiences by interacting with the environment under a policy. The downfall is that many state-action pairs are never explored if a policy is deterministic. To ensure a stochastic policy, exploration is encourage by using for example an ϵ-greedy policy, which is greedy in the selection of an action but chooses an alternative action with a probability of ϵ [19].

Additionally, instead of computing values of states over the whole episode, that based on the return of the episode, the value can be computed based on an estimate of the value of following states. This bootstrapping process is called Temporal-Difference (TD) Learning [8]. The advantage is that it does not require a model of the environment of the reward and the next state probabilities and it can be used with continous tasks. Further empirical evidence suggests that TD methods seem to converge faster.

One example of a model-free value-based TD method is Q-Learning [20]. The learned action-value function $Q(s, a)$ directly approximates the optimal action-value function Q^* and a policy can then be generated directly by using a greedy strategy. In its simplest form the algorithm can be described as

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

The full algorithm can be seen below [14].

Algorithm 1 Q-Learning

```
1: *Initialize* $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3:     Initialize $s$
4:     **repeat** for each step in episode:
5:         Choose $a$ from $s$ using policy derived from $Q$
6:         Take action $a$, observe $r, s'$
7:         $Q(s,a) \leftarrow Q(s,a) + \alpha\,(r + \gamma max_{a'} Q(s', a') - Q(s,a))$
8:         $s \leftarrow s'$
9:     **until** $s$ is terminal
```

2.1.2 Policy Gradient Methods

Instead of working with Value Functions, Policy Gradient Methods parameterize the policy function and optimize it directly:

$$\pi_\theta(a, s) = P[a|s, \theta]$$

The objective is the expected return of the episode and the optimisation in terms of the objective is done with gradient descent methods.

Advantages of policy gradient methods are better convergence properties and the efficiency of not needing to compute an exact value of every state. Furthermore in aliased states in an POMDP a deterministic policy could not differentiate between similiar but for the observer equal states while a stochastic policy could factor in the ambiguity of partial observed states. A disadvantage of this approach is high variance and the convergence in local rather than global optima [21].

One of the earliest examples of policy gradient methods is REINFORCE [21]. It is a family of algorithms using Monte-Carlo Policy Gradient. It makes use of the Policy Gradient Theorem to get the gradient and samples the expectation by using the return R_t as an unbiased sample of the action-value functions.

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) R_t$$

The disadvantage of using REINFORCE is that it has high variance.

To further reduce variance in the unbiased estimate R_t of the action-value function a baseline can be subtracted $R_t - b_t(s_t)$. The value function can be used as a baseline and due to R_t being an estimate of $Q(s_t, a_t)$ this leads to

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

This is the Advantage function, that is the advantage of using action a_t in state s_t instead of following the policy. We will come back to this in Section 2.4.2 about Advantage-Actor Critic (A2C).

2.1.3 Actor-Critic Methods

Actor-Critic Methods combine the value-based and policy-based approach. Policy gradient methods also include Actor-Critic Methods which learn both the policy and the state-value function. Informally, the agent uses the current policy (actor) to act in an environment, while the critic evaluates the action (see Figure 2.2).

An example of Actor-Critic methods, based on REINFORCE, is the Action-Value Actor-Critic. A parameterized action-value function $Q_w(s, a)$ is introduced that replaces the return to approximate the policy gradient as

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)]$$

$$\Delta\theta = \alpha \nabla_\theta log \pi_\theta(s, a) Q_w(s, a)$$

where $J(\theta)$ is the policy objective function.

To update the critic, after each action a the critic evaluates if the action was better in respect to the expectation by the state-value function. Depending on the outcome the probability of taking action a should in- or decrease. This is the TD error already mentioned before:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

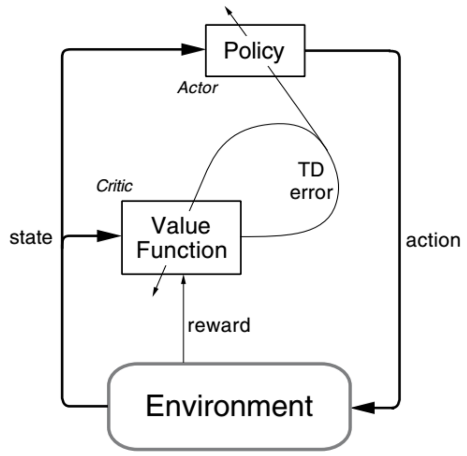


Figure 2.2: actor-critic architecture[14].

2.2 Artificial Neural Networks

Artificial Neural Networks (ANN) are universal function approximators, that can be trained without any prior knowledge on input data and corresponding targets to learn the underlying function or pattern [22][23].

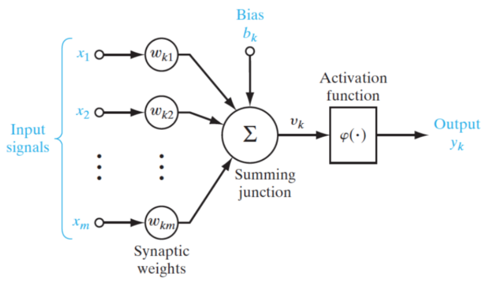


Figure 2.3: Schematic of a neuron[24].

A neural network is made up of interconnected layers, starting with an input layer, followed by several hidden layers and ending with an output layer. These layers consist of computational units (also called neurons) that are parametrized with a set of weights. They compute a weighted sum with a bias and the output is then transformed by an activation function.

$$\nu = \sum_{i=0}^{m} w_i x_i + b$$
$$y = \phi(\nu)$$

Depending on which activation function is chosen, distinct mathematical properties are achieved. The most common ones are the sigmoid and the ReLU[25].

A sigmoid unit uses the logistic function as activation and maps its input to a range of values monotonically increasing between (0, 1) or (-1, 1) [26]. It is differentiable and defined for all real inputs and the derivative is always non-negative. The logistic function is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The Rectifier Linear Unit (ReLU) [27] is the most commonly used activation function because of its favorable mathematical properties. ReLU is defined as

$$(z)_+ := max(0, z)$$

Compared to the sigmoid for instance, it suffers from fewer vanishing gradients, is computationally faster and has a more sparse activation [28]. When $|z|$ is large, sigmoids reach saturation and the derivative vanishes which leads to no updates for the weights.

To train an ANN, a forward pass is performed on the input data, meaning the data is fed into the input layer and computation occurs successively in every hidden layer until the output layer is reached. Due to the network being parameterized by random initialized weights, it will have output values which differ from the target values. This prediction error can be measured by a loss function, which needs to be defined depending on the task.

The goal is to minimize the loss function given a set of input values and corresponding target values. This is achieved by determining the weights by maximizing the likelihood of the target values. This requires the evaluation of gradients with respect to the network parameters. After a forward pass, local gradients are computed by iteratively propagating them backwards. The process of computing gradients is called backpropagation. The gradients are then applied to update the weights by using gradient or stochastic gradient descent methods:

This concept can be generalized to multiple layers and lead to success in multiple areas such as medical diagnosis, computer vision, speech recognition, finance, autonomous cars and data mining. As an example, Facebook achieved an accuracy of 97.35% on the Labeled Faces in the Wild (LFW) dataset, with DeepFace, a program that uses neural networks to identify faces, rivalling the performance of humans [29][30]

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are special end-to-end ANN architectures made up of preprocessing layers (Convolutional, Pooling) and Fully-Connected Layers and are commonly used for image processing [31].
Images consist of 3-dimensions: width, height, depth, where the depth stands for the colour channels (RGB). The design is based on the organization of receptive fields in the visual cortex.

While regular ANNs consist of only fully-connected layers, this is suboptimal for images in that it would lead to an explosion in network parameters and would not take the spatial structure into account. For example, a 64x64x3 image results in $64 * 64 * 3 = 12,288$ weights. Instead the benefit of CNNs is that it can take full advantage of the spatial information of the input with less weights.

Now we will give a short description of the layers of a CNN. A CNN consists of an input layer, multiple preprocessing layers made up of a sequence of Convolutional and Pooling Layers and at the end a Fully-Connected Layer.

The purpose of the Convolutional Layer is to extract features by training so called filters (also known as kernels). Like the name of the layer implies they do so by appyling convolutions on the input data. Among others, one benefit of convolutions is that spatial dependencies in the data are preserved, although, mathematically speaking, the convolution is a cross-correlation.

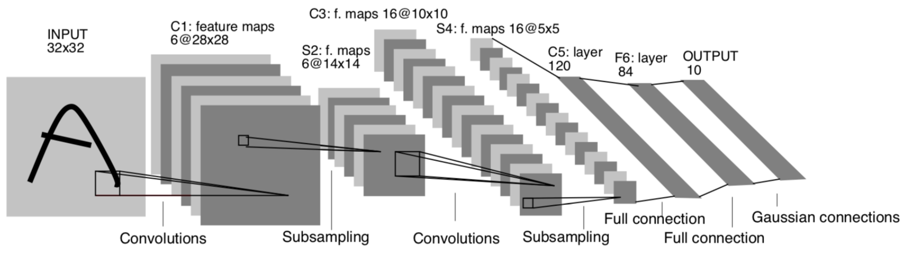


Figure 2.4: Example: Architecture of LeNet5 [31].

The Convolutional Layer consists of filters (essentially weight matrices) and corresponding receptive fields (essentially the filter size). The receptive field enforces local connectivity by exploiting spatially local correlations through each neuron only receiving input from small subparts of the input image.

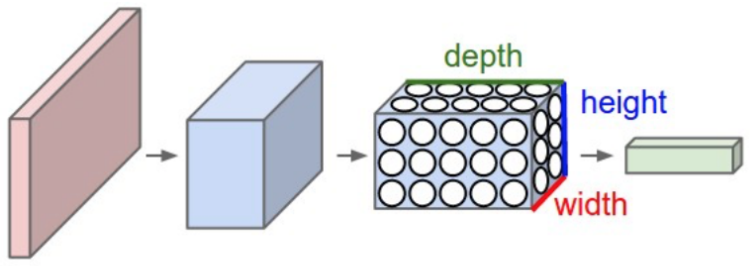


Figure 2.5: Convolutional Layer [32].

Properties of a filter are its stride and zero-padding. The stride determines the pixel distance between receptive fields per applying the filter and therefore how much the receptive fields overlap. Zero-padding determines the size of the border of zeroes around the input data, which allows controlling the size of the activation maps.

When trained, a filter activates when features such as specific edges or curves are detected at some position in the input. The outputs are 2-dimensional activation maps that essentially mark the spatial position of the feature. The output of the layer is then the stacked activation maps.

A special case are 1x1 convolutional filters that are used to reduce dimensionality along the channel dimension [33].

Usually between the Convolutional and Pooling Layer a element-wise non-linear activation function such as ReLU is used to introduce non-linearity.

The purpose of the Pooling Layer is the downsampling of spatial dimensions to avoid overfitting and reducing the input dimension. There are several types of pooling like Max and Average Pooling. In the case of Max Pooling, the most commonly used type, the largest value of a receptive field is taken.

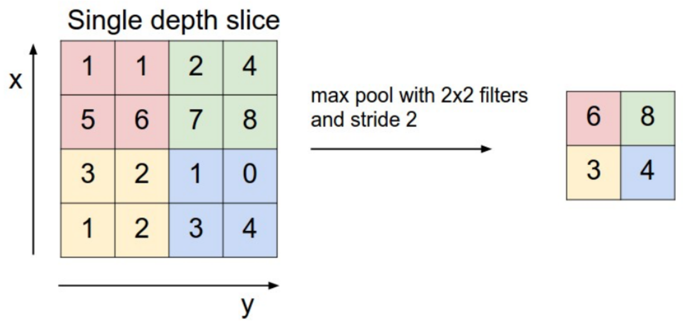


Figure 2.6: Max Pooling Layer [32].

The CNN architecture ends with a fully-connected (FC) layer. As the Convolutional and Pooling Layers output high-level features, the purpose of the FC Layer is classifying the input image. As with ANNs, backpropagation and gradient descent methods are used to train the CNN.

One example of the successful usage of CNNs is AlexNet, a CNN architecture that won ImageNet Large Scale Visual Recognition Challenge in 2012 with a significant jump in classification accuracy [34].

2.4 Deep Reinforcement Learning

As Backgammon has 10^{20} states, Go 10^{170} states and tasks with real-world complexity have commonly a continuous state- and action-space, it is unpractical to differentiate every state. Deep Reinforcement Learning combines Reinforcement Learning with Function Approximators like artificial neural networks or convolutional neural networks to generalize seen states to unseen states. This is particularly useful for only partially observed environments. Furthermore function approximators can utilize non-i.i.d. data like highly correlated trajectories and non-stationary problems that are typical for Reinforcement Learning problems [35].

In this section, we examine Deep Reinforcement Learning methods, in particular the Deep Q-Network (DQN) and Advantage Actor-Critic (A2C).

2.4.1 Deep Q-Learning

Deep Q-Learning combines Q-Learning with a convolutional neural network as function approximator [12].

The Convolutional Neural Network is used to approximate the optimal action-value function. In particular

$$Q^*(s,a) = max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

The network parameterizes the action-value function as $Q^*(s', a'; \theta)$.

As input, the CNN gets a preprocessed high-dimensional visual input to extract features and outputs the values for all actions. Then the action is selected by an ϵ-greedy policy.

The Q-Network is trained to reduce the mean-squared error in the Bellman Equation (Eq. 2.2). With approximated target values as

$$y = r + \gamma max_{a'} Q(s', a'; \theta)$$

using most recent parameters θ. The loss can be defined as

$$\begin{aligned} L(\theta) &= \mathbb{E}_{s,a,r}[(\mathbb{E}_{s'}[y|s,a] - Q(s,a;\theta))^2] \\ &= \mathbb{E}_{s,a,r,s'}[(y - Q(s,a;\theta))^2 + \mathbb{E}_{s,a,r}[\mathbb{V}_{s'}[y]]] \end{aligned}$$

This leads us to the following gradient

$$\nabla_\theta L(\theta) = \mathbb{E}_{s,a,r,s'}[(r + \gamma max_{a'} Q(s', a'; \theta) - Q(s,a;\theta)) \nabla_\theta Q(s,a;\theta)]$$

An issue that arises is the instability based on naturally occuring correlation in trajectories and therefore high variance. To solve this, Experience Replay, a biological inspired process to randomize trajectories, is introduced. The Experience Replay Buffer stores experiences as $e_t(s_t, a_t, r_t, s_{t+1})$ at each timestep t over several episodes. For learning a sample from the buffer is drawn uniformly at random. This reduces the correlation and leads to greater data efficiency as experiences are sampled several times.

To further improve stability a separate duplicate Q-Network is introduced that is used as a target network while the original network gets updated. After several updates, the improved Q-Network replaces the old target network and the process repeats.

There exists several improvements of DQN, most notably the Dueling DQN, which approximates the state-value function and advantage separately and combines those estimates to get the action-values. This reduces overestimation and improves the stability [36].

DeepMind's Deep Q-Network is considered a major breakthrough and started off major research interest in the field of Deep Reinforcement Learning. DQN surpassed all other algorithms in classic Atari 2600 games and achieved human level performance on only raw visual inputs without prior knowledge.

2.4.2 Advantage Actor-Critic

While DQN had a lot of success, it has several drawbacks, notably Experience Replay. Experience Replay is memory and computation intensive and updates the network on data generated from a target network that uses outdated parameters.

One proposed solution is Asynchronous Advantage Actor-Critic (A3C) [37]. As the name already suggests, A3C belongs to the Actor-Critic Methods. It is based on General Reinforcement Learning Architecture (Gorila) [38], which performs asynchronous training in a distributed setting, computes gradients locally and sents them asynchronously to a central parameter server. The updated policy parameters are then sent to the local clients.

Instead of using Experience Replay, A3C utilizes agents (workers) on multiple environments in parallel by executing them asynchronously. The workers compute gradients locally and send those to a global target network that updates the parameters and sends them back to the worker.

The parallelism decorrelates the experience and reduces variance. Further the reduction in training time is roughly linear in the number of parallel workers. Due to no longer relying on Experience Replay on-policy methods like Actor-Critic can be used.

As an Actor-Critic method, the algorithm has a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta)$. An update is performed every t steps or when the terminal state is reached. The policy gradient is then defined as

$$\nabla_\theta \log \pi(a_t|s_t;\theta) A(s_t, a_t;\theta)$$

and the Advantage estimate $A(s_t, a_t;\theta)$ as

$$A(s_t, a_t;\theta) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+1};\theta) - V(s_t;\theta)$$

where k is the length of the trajectory. Adding the entropy of the policy π further encourages exploration and helps prevent converging to local minima.

This defines the objective function as

$$\begin{aligned} J(\theta) &= \nu \log \pi(a_t|s_t;\theta) A(s_t, a_t;\theta) + \alpha(R_t - V(s_t;\theta))^2 - \beta H(\pi(s_t;\theta)) \\ &= \nu \log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta)) + \alpha(R_t - V(s_t;\theta))^2 - \beta H(\pi(s_t;\theta)) \end{aligned}$$

where the hyperparameters α and ν trade off the importance of the different loss components and β controls the strength of the entropy regularization term.

As a result of most workers using a version of the global network with old parameters non-optimal updates are computed. As an alternative a synchronous implementation was proposed called Advantage Actor-Critic (A2C). In A2C the workers are only used for communicating with the environments and do not have a local network. Workers act synchronously and gather experience which is then sent to the global network. One advantage of this approach is that the work load is on the global network and thus specialized hardware like GPUs can be used more efficiently, further reducing training time.

The adapted algorithm for A2C from A3C can be found below:

Algorithm 2 Advantage Actor-Critic (A2C)

```
1: Input: max updates $T_{max}$, learning rate $\nu$, number of training steps $t_{max}$
2: Initialize all environments and receive initial states $s_0$
3: Initialize the global network with random weights
4: repeat
5:     Reset states $s$, actions $a$, rewards $r$, values $v$
6:     $t \leftarrow 0$
7:     for $t \leq t_{max}$ do
8:         Perform $a_t$ according to policy $\pi(a_t|s_t;\theta)$
9:         Receive reward $r_t$ and new state $s_{t+1}$
10:        $t \leftarrow t+1$
11:    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t,\theta) & \text{for non-terminal } s_t \end{cases}$ // Bootstrap from last state
12:    for $i \in \{t_0, ..., t_{max}\}$ do
13:        $R_i \leftarrow r_i + \gamma R_{i+1}$
14:    Compute gradient: $\theta \leftarrow \theta + \nabla_\theta J(\theta)$
15:    Perform synchronous update of $\theta$
16: until max updates is reached
```

2.5 StarCraft II

StarCraft II (SC2) is the sequel to the real-time strategy videogame StarCraft, developed and published by Blizzard Entertainment and known as one of the most popular and competitive videogames of all time [11].

The StarCraft II Learning Environment (SC2LE) [1] is the successor to the community-made Brood War API (BWAPI) [39], an open-source C++ framework for StarCraft: Brood War [40], created by reverse engineering the game and relying on reading and writing to memory in order to interact with the game.

BWAPI already attracted some research interest [41][42] and several tournaments have formed, where AI bots compete against each other. Among others, one tournament is part of the annual IEEE Conference on Computational Intelligence and Games [43][44][45].

DeepMind and Blizzard developed the StarCraft II API and PySC2, a Python environment wrapper which is optimized for Reinforcement Learning agents [13]. It follows in the footsteps of video game benchmarks like the OpenAI Gym [46] and the Broodwar API. In general, benchmarks are critical to advance and test algorithms and fulfil the need of having a mutual standard.

The game is usually played between two players with each controlling an army of one of three distinct races competing for influence and resources and can last from a few minutes to an hour. Each race comprises of unique units and structures, and consequently unique strengths and weaknesses. The goal of the game is to defeat the enemy player by destroying his base and his units. A game of SC2 consists of longterm high-level planning and short-term tradeoffs. Small early decision making can have game deciding long-term consequences.

Figure 2.7: StarCraft 2 Gameplay [47].

Compared to games like Chess or Go, SC2 can capture the continuous nature of the real world and posseses a high-dimensional, continuous action- and state-space. The player has to be able to deal with the concepts of macro- and micromanagement, dealing with the tradeoff of gathering and spending resources, controlling units in an effective manner and adopting the right combat tactics. Furthermore due to the only partially observed map (also known as 'fog of war'), the players are only in a partially-observed state, that means imperfect information over a long time horizon, leading to difficult decision making. Something that is even considerably challenging for a human players. Additionally, this makes StarCraft II a POMDP. With its sparse ternary rewards (tie, win, lose), the game is a perfect training ground for state-of-the-art reinforcement algorithms.

2.5.1 PySC2

PySC2 is a Python environment wrapper for the StarCraft II API and part of the SC2LE [46].

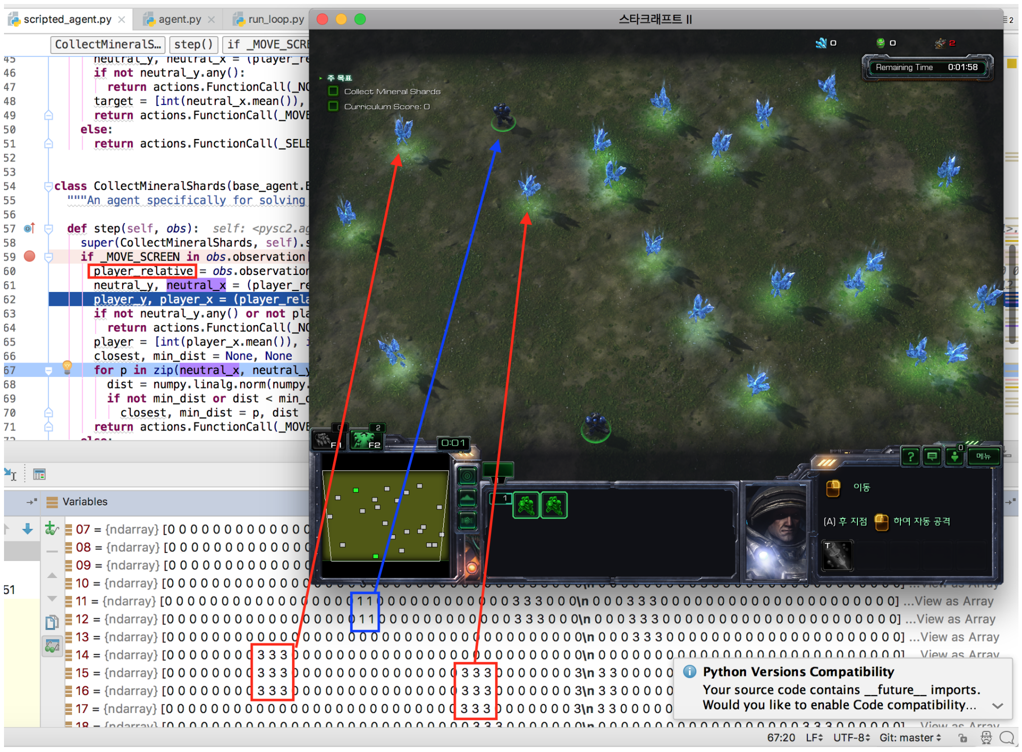
The API provides ways to create new environments, set properties like the resolution of spatial observations, take certain actions and exposes spatial and structural features in isolated layers. Further, it is used to communicate with SC2 programmatically.

Every time an action in an environment is taken it returns an observation, a tensor of spatial and structural feature layers, ranging from general player information and in-game alerts to decomposed and structured screen and minimap features like which units are friendly and hostile and which units are selected. For example in an observation, spatial features are exposed as `feature_screen` and `feature_minimap`, which are multidimensional tensors with height and width of their respective resolutions and different feature layers as channels. An element in the feature layer corresponds with a game pixel. In the `player_relative` feature layer, pixels can take on values between 0 and 4, which denote background, self, ally, neutral and enemy units respectively. Non-spatial features are individually exposed and can not be read from pixel values. As an example, the general player information is a tensor with 11 elements: player_id, minerals count, vespene count, supply used, supply cap and more.

An action is taken by calling the step function on the environment and passing a single FunctionCall in pysc2.lib.actions with all its arguments filled. To be precise actions are exposed as nested lists of action id and corresponding arguments. For example `move_camera` is the action with action id 1 of moving the ingame perspective to a position on the minimap. That means the action takes as argument coordinates on the minimap.

The game speed is determined by how fast steps are taken using the API. This gives a significant speed-up compared to the normal game. Bottlenecks are scene and unit complexity. Actions per minute (APM) are determined by a variable called `step_mul`, which determines how many game frames are skipped between actions. For example, if an action is taken every 8 frames the agent has an APM of 180. While an AI could take an action in every frame that is not comparable to humans, who have an average APM ranging from 10 to 300 depending on the player skill. While SC2 is mostly deterministic, there are small sources of randomness, such as weapon speed and update order, to provide random outcomes in fair settings to make the game feel more natural. A random seed to mitigate this can be set for evaluation purposes.

See the environment documentation for a complete description of observational and action space [49].



```
        neutral_y, neutral_x = (player_re
        if not neutral_y.any():
          return actions.FunctionCall(_NO
        target = [int(neutral_x.mean()),
        return actions.FunctionCall(_MOVE
      else:
        return actions.FunctionCall(_SELE


class CollectMineralShards(base_agent.B
  """An agent specifically for solving

  def step(self, obs):  self: <pysc2.ag
    super(CollectMineralShards, self).s
    if _MOVE_SCREEN in obs.observation
      player_relative = obs.observation
      neutral_y, neutral_x = (player_re
      player_y, player_x = (player_rela
      if not neutral_y.any() or not pla
        return actions.FunctionCall(_NO
      player = [int(player_x.mean()), i
      closest, min_dist = None, None
      for p in zip(neutral_x, neutral_y
        dist = numpy.linalg.norm(numpy.
        if not min_dist or dist < min_d
          closest, min_dist = p, dist
      return actions.FunctionCall(_MOVE
```

```
07 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0
08 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0
09 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0
10 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0
11 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0\n 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...View as Array
12 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0\n 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...View as Array
13 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\n 0 0 0 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...View as Array
14 = {ndarray} [0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\n 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...View as Array
15 = {ndarray} [0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0 0 0 0 0 3\n 3 3 0 0 0 0 0
16 = {ndarray} [0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0 0 0 0 0 3\n 3 3 0 0 0 0 0
17 = {ndarray} [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 0 0 0 0 0 0 0 3\n 3 3 0 0 0 0 0
```

Python Versions Compatibility
Your source code contains __future__ imports.
Would you like to enable Code compatibility…

Figure 2.8: PySC2 Feature Layers [48].

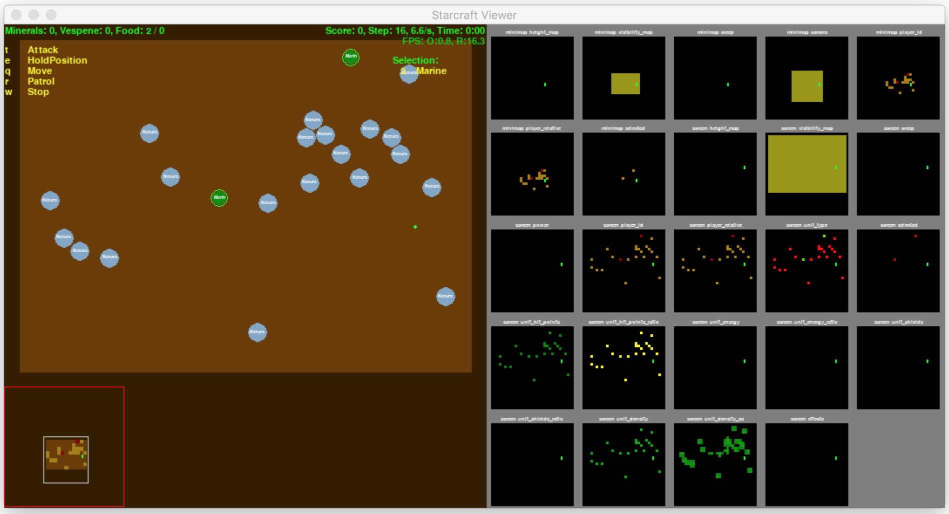


Figure 2.9: PySC2 Visualization with feature layers in headless Linux.

Chapter 3

Implementation

In this chapter we describe the implementation details of the agents that were used to examine the effectiveness of different machine learning algorithms. First, we created a simple vanilla Deep Q-Network with slight improvements like Prioritized Experience Replay. Second, to calibrate the task difficulty DeepMind created several baseline agents, particularly an Atari-net agent, that is a close adaptation of the architecture used for their Atari benchmark, and 2 fully convolutional agents, one with and one without convolutional LSTM module [50]. We recreated the fully convolutional architecture from DeepMind's baseline agent using Advantage Actor-Critic, which is examined in Section 3.2. Third, we created, for evaluation purposes, an algorithm that reaches near optimal results in the *CollectMineralShards* minigame.

All of our agents are written in the programming language Python. We use the PySC2 Library [13] to interact with the game engine and utilize OpenAI's Baseline Library, a set of high-quality implementations of reinforcement learning algorithms [51]. Particularly, our DQN agent is based on OpenAI's DQN example and uses its model implementation [52]. The DQN and A2C agent both use the `logger` class of the Baseline Library for logging, extending it where needed. Furthermore, the Tensorflow Library [53] is used to build Neural Network archictecture as it provides an easy interface and takes care of a large part of computational setup. Additionaly, as pre-processing is needed to ensure the modularity for different algorithm, the NumPy Library [54] was used, which supports a variety of functions to operate easily on arrays.

3.1 Deep Q-Network

The DQN agent consists of main.py, where the environment and model is created, deepq_runner.py, where the main execution loop and logging is located, and deepq_preprocess.py, where observations are pre-processed.

In main.py the DQN model and the SC2 environment is created. As a model we use OpenAI's end-to-end convolutional architecture (cnn_to_mlp). The preprocessed screen observation is passed through a 2-layer convolutional network with 16 and 32 filters of size 8 x 8 and with stride 4 and 4 x 4 with stride 2 respectively, followed by a fully-connected layer with 256 units. We evaluated the agent with and without dueling. Model and environment are then passed on to the main execution loop in deepq_runner.py.

deepq_runner.py consists of the ActWrapper class, which is used for serializing the model parameters and the function `learn`, which trains the DQN. `learn` first creates a placeholder for the input data and builds the network and target network using the model.

We use the LinearSchedule from the Baseline Library to create a Prioritized Experience Replay buffer with linear interpolated beta schedule and a specified linear interpolated exploration schedule [55]. The original Experience Replay samples the experience transitions uniformly from a replay memory. Instead, we use Prioritized Experience Replay samples based on importance. Importance for a transition is defined as the magnitude of the TD error. The higher the TD error the more frequent the transition is sampled. The drawback of this greedy approach is severe overfitting and sensibility to noise spikes. To mitigate this we use stochastic priorization. The probability of sampling transition i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$
$$p(i) = \frac{1}{rank(i)}$$

where $rank(i)$ is the position by TD error in the replay buffer and α is the strength of the prioritization. Due to introducing a bias, weighted importance-sampling is added

$$w_i = \left(\frac{1}{N} * \frac{1}{P(i)}\right)^\beta$$

Using the linear schedule on β, we anneal the amount of importance-sampling correction over time. The gradient is multiplied by the weight w_i and reduces the bias and also the magnitude of the gradient and thus the learning rate, which leads to better convergence.

Next the main execution loop is a sequence of retrieving observation, pre-processing, acting in the environment and storing the transition in the replay buffer.

After a specified consecutive number of steps, the agent samples a batch of experience from the replay buffer, trains the network on the sample, updates the priorities and periodically updates the target network's parameters.

The pre-processing consists of extracting reward and `StepType`, indicating if an observation is the first or last step in an episode, and `screen`, the player's on-screen view extracted features in form of a tensor. Due to the modular structure of the pre-processing, evaluating on different combinations of `screen` features is possible.

3.2 Advantage Actor-Critic

In this section we introduce the Advantage Actor-Critic agent. We will first examine the basic architecture laid out in the reference paper [1] and then dive deeper giving an overview over our codebase, the implementation details and advances we made.

3.2.1 Basic Architecture

The basis for our agent is DeepMind's fully convolutional architecture, which is visualized in Figure 3.1. In essence, it is a convolutional architecture adapted to the needs of StarCraft II, in particular the action space.

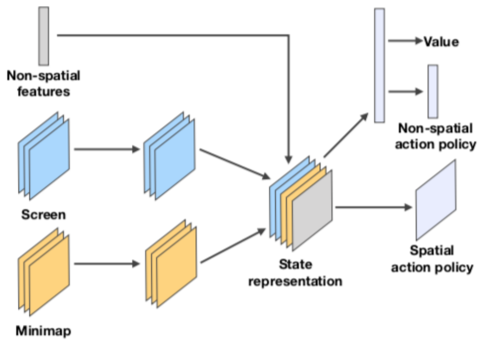


Figure 3.1: FullyConv architecture [1]

In StarCraft II the input feature layers consist of 2-dimensional tensors representing the same perspective view as a human would see in the game but instead of raw RGB values, features are directly exposed. For example, units can take up several elements ('pixels') in the tensor, having the size of the resolution, and every element contains various information about these units, such as their type (e.g. Marine or Zergling) or player affiliation (e.g. friendly, hostile).

As these features are most often categorical, they have to be preprocessed. Usually this is dealt with by one-hot encoding. This can cause massive expansion in dimensionality. For example there are currently 1914 different unit types and an one-hot encoding of the `unit_type` feature layer with size NxN, would lead to a new dimensionality of NxNx1914. For that reason all categorical features are embedded into a continous space by using a one-hot encoding and subsequently 1 x 1 convolutional filters. These reduce the depth by essentially 'squashing' the one-hot encoded tensor together along the channel dimension by applying a dot product.

Numerical features are transformed to a logarithmic scaling to combat the value explosion of some values such as minerals or hit-points.

Unlike usual reinforcement learning agents, which tend to reduce spatial dimensionality to extract spatial information from the input, the fully convolutional architecture preserves the spatial structure in the observation (screen and minimap) as it is passed through 2 layers with 16, 32 filters of size 5 x 5, 3 x 3 respectively with no stride or padding. This is needed as spatial actions act in the same space as the inputs and reducing the dimensionality might be detrimental.

Next, the spatial network outputs and non-spatial features are concatenated to a state representation. For this the non-spatial features are broadcasted along the channel dimension as they do not have matching shapes.

To compute the state-value, the state representation is passed through a fully-connected layer (FC1) with 256 units and ReLU activations. To further get the non-spatial, that is categorical like right-click or button-press, action policy, the output of the fully-connected layer FC1 is followed by another one with softmax outputs.

Lastly, to obtain the spatial action policy the state representation is passed through a layer of 1x1 convolutions and a subsequent softmax layer for each spatial action argument.

As described in Section 2.5.1, actions are exposed as nested list of an action and its arguments. For example `Move_screen`, which defines the move action for currently selected units in the screen space, takes two arguments: a bolean value, specifying if this action should be deferred or not, and a tuple of coordinates that represent the location on the screen. This would mean that we would have to model a joint distribution of action and arguments but as the action space in StarCraft II is very large, this is impractical. Instead the assumption is made that actions and arguments are conditionally independent and thus the softmax output layers can determine the probability distribution individually. Additionally, the argument types that are not allowed for a specific action are masked, that is setting all probabilities to zero that are unavailable for this action. For example `Move_screen` action does not take the argument `unload_id`. We will come back to this in the next subsection

3.2.2 Implementation Details

A general difference in DeepMind's baseline compared to our implementation is the use of A2C as DeepMind uses A3C [37]. As we laid out in Section 2.4.2, this is computationally more efficient. Furthermore, due to non existing official A2C publications, we used several reference implementations of A3C and A2C. Notably NVIDIA's GA3C, OpenAI's baseline A2C and several others [56][57][58][59][60][61][62][63]. These are also referenced in the specific code sections.

Further, as a result of DeepMind not making their code publicly available, assumptions had to be made based on the reference publication.

We chose a modular design such that the code is easily extendable to use different architectures or networks. First, `main` sets up all the individual components of the agent. The A2C network, environments and logger are created that are passed on to the `Runner`, which contains the main execution loop. All hyperparameter and general configuration, such as type of model or number of environments, is setup in `main`.

The SC2 environments are all initialized in the `EnvWrapper` class, that further creates an environment pool `EnvPool` that is used as interface for communication. The environment pool creates separate worker processes for every environment and uses pipes to communicate with them. While the environment pool is an abstract vectorized environment and therefore acts as interface between the individual environments and the wrapper, the EnvWrapper acts as interface between network and EnvPool and splits upon receiving, environment observations into rewards, StepType and raw observations, which are then preprocessed to states.

The `Runner` is used as main execution loop and lets the network and environments interact. This is done by first resetting the environments and receiving the initial states. Second, it passes the states as input to the network, which returns the actions that the individual environments should take. Third, the actions are taken in the environments and new states are received. This repeats until the maximum train-loop steps are reached. Afterwards the network is trained on all states and actions.

After every train-loop, the `Logger` is called, which utilizes the baseline implementation to achieve extensive logging, like the elapsed time, number of samples, important current and past rewards. General configuration and hyperparameters are logged at the start of every log file at the beginning of every execution.

To preprocess the raw observations to states, the spatial and non-spatial features that are used are extracted. This is done for each environment to build raw state. As raw observations in StarCraft II are in channels-first format (NCHW), we transform the observation to channels-last format (NHWC) because the non-spatial observations need to be broadcasted later.

The `ac_network` class builds the fully-convolutional network and is used to run the Tensorflow sessions.

The `fully_conv` model in `models` implements the basic architecture described in Section 3.2.1. The non-spatial input is split in non-spatial features and available actions. Available actions are only used to mask our actions. This saves time compared to making the network learn masking itself. Masking is done by computing a dot product between the available actions and the action output and subsequently normalized such that the total probability is 1.

While the numerical features are only transformed to logarithmic scaling, the categorical features are passed through a layer of 1x1 convolutions. As number of filters we use a logarithmic transformation of the dimensionality of the feature scaling. For example this reduces the output dimensionality of unit types from 1921 to only 11.

Next, the non-spatial features are broadcaste to match the shape of spatial features. The combined state representation is then transformed from NHWC back to NCHW, as NCHW is the optimal format for training with Tensorflow on NVIDIA GPUs [64].

The flattened state representation is then passed through the subsequent fully-connected and convolutional layers, as already mentioned in Section 3.2.1., to create policy and value outputs. To obtain an action and arguments, we then sample from the probability distributions received by the policy output.

To train the network, we compute the loss function over the mean of the batch:

$$\begin{aligned} J(\theta) &= \nu \log \pi(a_t|s_t;\theta) A(s_t,a_t;\theta) + \alpha(R_t - V(s_t;\theta))^2 - \beta H(\pi(s_t;\theta)) \\ &= \nu \log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta)) + \alpha(R_t - V(s_t;\theta))^2 - \beta H(\pi(s_t;\theta)) \end{aligned}$$

As an additional enhancement, we used Generalized Advantage Estimation [65], which is a more sophisticated form of Advantage. This can tremendously reduce variance but unfortunatetly comes at the cost of introducing bias.

We defined Advantage in Section 2.1.2 as

$$A(s_t,a_t) = Q(s_t,a_t) - V(s_t)$$

Estimating $Q(s_t,a_t)$ gives us

$$A_t^1(s_t,a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$A_t^2(s_t,a_t) = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$$

$$A_t^\infty(s_t,a_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... - V(s_t)$$

If the time frame of estimation is small, then the estimation has low variance but high bias, whereas with a large time frame the estimation has high variance but low bias. This is because the amount of dependency of our estimation.

Instead of using a specific time frame, Generalized Advantage Estimation uses all of them. For this we define the Bellman residuals:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

This defines the Generalized Advantage Estimation as

$$\begin{aligned} A_t^{GAE(\delta,\lambda)} &= (1-\lambda)\left(A_t^1 + \lambda A_t^2 + \lambda^2 A_t^3 + ...\right) \\ &= (1-\lambda)\left(\delta_t + \lambda\left(\delta_t + \gamma\delta_{t+1}\right) + \lambda^2\left(\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2}\right) + ...\right) \\ &= (1-\lambda)\left(\delta_t\left(1+\lambda+\lambda^2+...\right) + \gamma\delta_{t+1}\left(\lambda+\lambda^2+...\right) + ...\right) \end{aligned}$$

Using geometric series formula:

$$\begin{aligned} &= (1-\lambda)\left(\delta_t\frac{1}{1-\lambda} + \gamma\delta_{t+1}\frac{\lambda}{1-\lambda} + ...\right) \\ &= \sum_{i=0}^{\infty}(\gamma\lambda)^i\,\delta_{t+i} \end{aligned}$$

where the hyperparameter λ and γ adjust the bias-variance tradeoff.

We compute the loss and use the RMSProp algorithm to optimize our network. Further cliping gradients at 1 seemes to have stabilize the network more.

3.3 TSM Scripted Agent

For evaluation purposes we also implemented an agent that reaches near optimal score on the minigame *CollectMineralShards*.

In the minigame *CollectMineralShards* the agent has two Marines (game units) and the goal is to collect 20 Mineral Shards randomly spread (at least 2 units away from all Marines) on the map. For every collected Shard the agent receives +1 reward. Whenever all 20 Shards have been collected, a new set of 20 Mineral Shards are spawned at random locations. This repeats until a time limit of 120 ingame seconds is reached.

CollectMineralShards can be viewed as a generalization of the Traveling Salesmen (TSM) Problem. To be precise, as a non-fixed destination multi-depot multiple traveling salesman problem. The more efficiently the units move, the higher the score. An optimal strategy needs to exploit that the units can be controlled separately.

As with the DQN and A2C agents, the TSM Scripted agent is written in the Python programming language and uses the PySC2 library to interact with the game engine. It consists of 2 files main.py and mtsm.py.

In main.py an SC2 environment is created with a `step_mul` of 1 (we explain later why this is the case). After the environment is reset and the agent received the initial observation, the `mtsm` function in mtsm.py is called with the observation as parameter to calculate the game actions and paths for the Marines.

mtsm.py consists of two functions `mtsm` and `held_karp`. mtsm preprocesses the observation to extract the `player_relative` feature layer of `screen`. `player_relative`

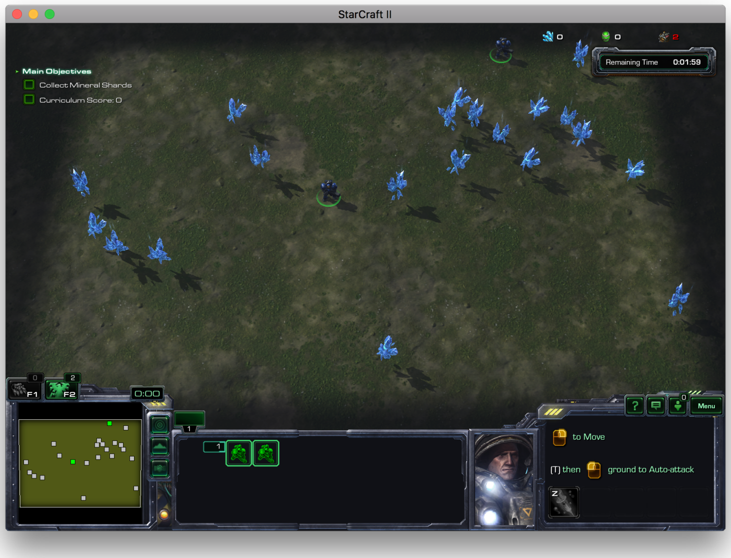


Figure 3.2: CollectMineralShards.

is used to get the coordinates of the Marines and Shards. The distances are then calculated and inserted into the list dist.

After this preprocessing `held_karp` is called. `held_karp` uses the Held-Karp algorithm [66] to calculate the optimal path for a TSM problem. It takes the list of distances and returns the optimal cost and optimal path (TSM path) to `mtsm`. The optimal cost is used to compare different initializations (seeds) of the minigame.

The agent creates the list `action_list`, which is used later as a bootstrap of actions, and adds the FunctionCall to select the first Marine (`Marine1`). The order of Marines is based on their location in the environment, the first Marine is always the left most.

As a result of the Marines only being recognizable by their location, the agent needs to keep track of their latest coordinates. When the agent wants to select a specific Marine the locations are compared to the latest coordinates and the one with the shortest deviation in distance is selected.

Next, the closest Shard to `Marine1` is found, the `move_action` is added to the `action_list` and the coordinates are added to list `path_marine1`. The same is done for `Marine2`. The closest, non-identical, Shard is found and the location and `move_action` are added to the action and corresponding path (`path_marine2`) list.

Corresponding to the closest Shards the Marines are now on two different positions in the TSM path. This leads to two distinct cases for selecting the rest of the paths optimally. If the Marines are on positions next to each other, they follow the path in opposite directions.

If the Marines are not next to each other, both follow the path in the same direction until one reaches the starting position of the other Marine. It then turns around and goes in the other direction of its starting position. See Figure 3.3 for an example.

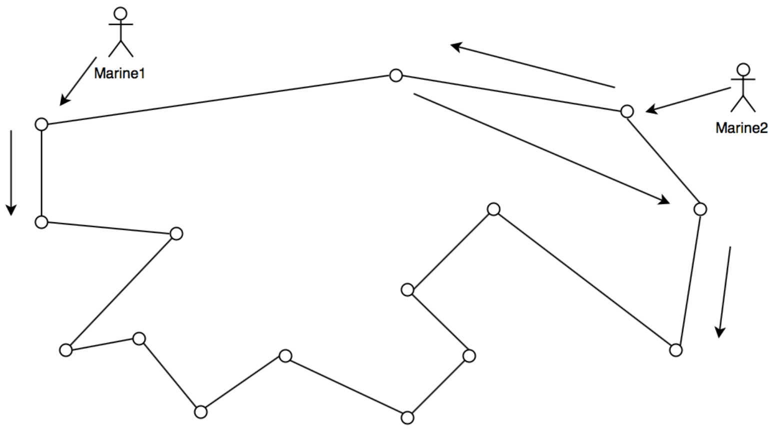


Figure 3.3: Example of Scenario 2.

After the optimal distinct paths for both Marines are computed, the mtsm function returns to the main function. Now the bootstrapping actions are executed in the environment. Afterwards turns are taken in selecting the next Marine and queueing the `move_action` to the next Shard on their corresponding paths. We do this so that even when Marines and Shards are close to each other on the path we reduce the potential waiting time between queuing the next actions. In StarCraft II actions can be queued for a unit such that the unit will carry them out in sequence.

A minor bug arises when the Marines are too close to each other and the wrong unit is selected. Hence the other is not selected anymore. This rarely happens in practice when the environment is not skipping frames between actions (`step_mul`=1) and therefore the Marines do not move considerably between turns.

When all 20 Shards are collected `mtsm` is called on the latest observation and the process repeats until the end of the episode is reached. We believe this to be near optimal.

Chapter 4

Evaluation

In this chaper we evaluate the agents on a set of tasks, examine our setup and lastly discuss our results. DeepMind provides seven minigames to divide the full-game into subsets and to test agents on different game mechanics and scenarios with different reward structures. Unlike the full-game where the reward is ternary, that is win, tie or lose. In Section 4.1.1 we give a brief overview over the minigames and in Section 4.2 we examine the results of our agents.

4.1 Setup

We tried to stay as close as possible to DeepMind's setup to be able to compare the results, although some changes had to be made. Our computational setup consists of an AMD Threadripper 1950X 16-Core Processor with Hyperthreading and 2x GeForce GTX 1080 Ti with 11GB RAM. Despite this being above average consumer hardware, it does not match the computational capabilities of DeepMind. Therefore it was infeasible to randomly sample hyperparameters. All experiments were run on Linux with the headless StarCraft II game.

Furthermore, in contrast to DeepMind we evaluated both agents for every minigame on a (32, 32) resolution, the same for `screen` and `minimap`, while DeepMind uses a (64, 64) resolution. To make sure that the results on lower resolution are accurate, we let every agent additionally run on the (64, 64) resolution at least once. However, it takes at least 10x as long as the state space grows by a factor of 4. Our empirical evidence supports that a (32, 32) resolution leads in fact to the same results, although at a reduction in wall clock time.

For both agents, we use a fixed `step_mul` of 8 as this complies with DeepMind and matches human APM levels. This means that the agent acts in the environment every 8 frames, which leads to an APM of 180. Further, we did not fix our maximum number of training updates as this widely varied depending on the minigames and agent.

The DQN agent is not as computational expensive to run as the A2C agent, due to the need of only a single CPU and GPU. Therefore hyperparameter tuning was done and we determined a `learning rate` of $3e-4$ and a `discount factor` of 0.99. Furthermore, we ran experiments with and without dueling but settled for dueling as it significantly reduced the training time. The same is true for the Prioritized Experience Replay.

With our A2C agent, we tried to stay as close as possible to the original `fully-conv` baseline agent, only differences are mentioned here.

We used 20 environments, that is environment worker threads that send and receive observations and actions respectively. As the cutoff for the trajectory, we settled for 16 forward steps (`n_steps`) per train loop or if a terminal signal is received, compared to DeepMind, which used 40. This does not seem to affect results but makes training more computational feasible. We used a `learning rate` of $7e-4$ and a `discount factor` of 0.99. Further, we use an entropy penality of $2e-3$ and 0.25 as value coefficient.

DeepMind does not mention, which spatial or non-spatial features they use. As using all features is computational expensive und simply impractical, we decided on a set of features that we used for all experiments with A2C. For `screen` feature layers, we use `visibility`, `player_relative`, `unit_type`, `selected`, `unit_hit_points`, `unit_hit_points_ratio`, `unit_density` and for `minimap` feature layers, we use `visibility`, `camera`, `player_relative`. These were decided as the lowest common denominator as they were used on all minigames. Some of them are redundant for specific minigames. For example for the minigame *CollectMineralShards*, `unit_hit_points_ratio` does not convey any information as the unit hit points do not change. For exploration tasks like `FindAndDefeatZerglings`, `camera` and `visibility` are important, while for minigames with battles are unit related features important. Additionally, we used as an guide-line other implementations.

DQN, however, uses a restricted set of features, as they massively extend training time. For this agent we use only spatial features, in particular `player_relative` and `selected` of the `screen` feature layers, and no non-spatial features.

4.1.1 Minigames

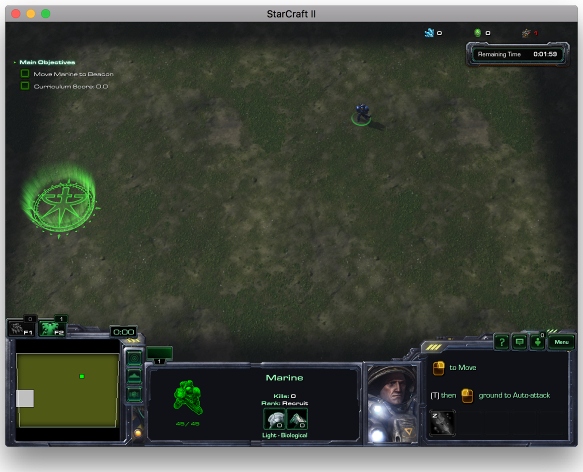
We will now give a brief summary of the minigames:

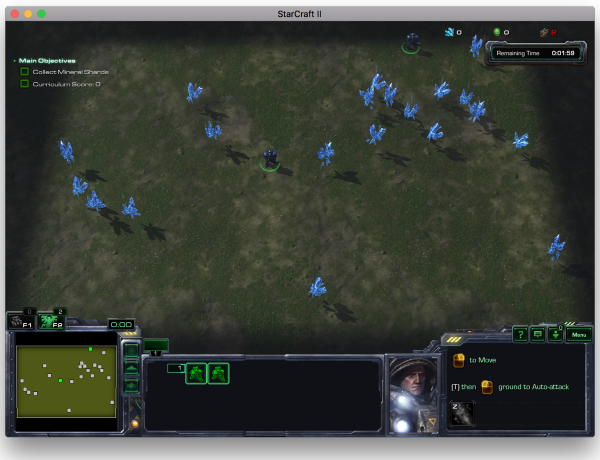
- MoveToBeacon: The agent has one unit and the goal is to go to a beacon that appears in random positions. It has a simple greedy strategy as solution and should be viewed as a simple unit test.
- CollectMineralShards: The agent has two units and the goal is to collect 20 mineral shards that are scattered on the map. It can be viewed as a generalization of the Traveling Salesmen (TSM) Problem.

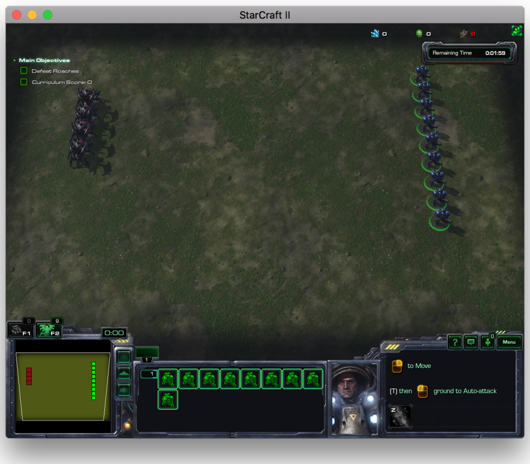
- FindAndDefeatZerglings: The agent has to explore the map to find and defeat Zerglings. This minigame mainly tests exploration.
- DefeatRoaches: The agent has a small army and has to defeat Roaches. For every successful round, the agents army gets reinforcements.
- DefeatZerglingsAndBanelings: The agent starts with a small army and has to defeat an army of Zerglings and Banelings. The agent's army is weaker than the enemies and a greedy solution is not optimal.
- CollectMineralsAndGas: The agent starts with a small base and has to collect resources to increase its score.
- BuildMarines: The agent starts with a base and units to gather resources and has to build an army of marines. It is the biggest subset of the full-game and therefore the most complex and hardest to learn for any agent.

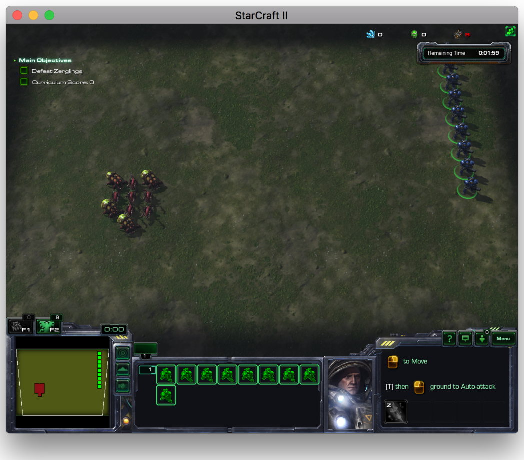
See the minigame documentation for a complete description of reward and game structure and Figure 4.1 for an overview of the minigames [67].

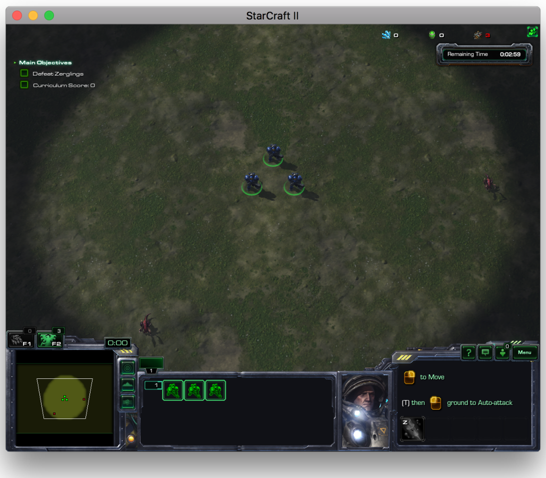
To better evaluate the minigame `CollectMineralShards`, we introduced in 3.3 the TSM Scripted Agent, an almost optimal scripted agent solution. We let both the TSM and A2C agent run on the same random seed to compute the efficiency difference.


(a) MoveToBeacon


(b) CollectMineralShards


(c) DefeatRoaches


(d) DefeatZerglingsAndBanelings


(e) FindAndDefeatZerglings

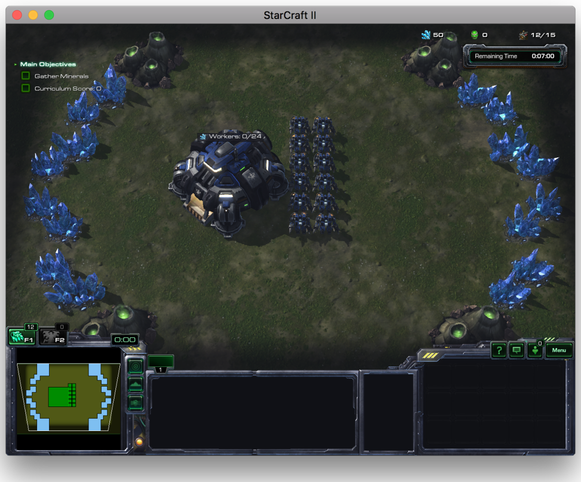

(f) CollectMineralsAndGas

Figure 4.1: Minigames ingame perspective

4.2 Discussion

Agent	Metric	MoveToBeacon	CollectMineralShards	DefeatRoaches	DefeatZerglingsAndBanelings	FindAndDefeatZerglings	CollectMineralsAndGas	BuildMarines
Human GrandMaster	best mean	28	177	215	727	61	7566	133
	max	28	179	363	848	61	7566	133
DeepMind Fully-Conv	best mean	26	103	100	62	45	3978	3
	max	45	134	355	251	56	4130	42
DQN Agent	best mean	25	63	-	-	-	-	-
	max	28	85	-	-	-	-	-
A2C GAE Agent	best mean	26.6	105	167	341	36	3436	0.5
	max	38	126	343	425	48	3878	14

Overall we ran a minimum of 5 experiments for each minigame on each agent and computed the mean over all experiments to obtain the results. The number of maximum game steps varied per minigame depending on the complexity. For example *MoveToBeacon* takes around 10 minutes wall clock time to reach an optimal mean reward, while *CollectMineralsAndGas* and *BuildMarines* have far longer episode times and the state space is far larger. Thus more samples are needed for training.

Our results are visible in the table above. The DQN agent was significantly slower as the other agents even though we used Prioritzied Experience Replay and Dueling. In particular, even on *MoveToBeacon* the DQN agent needed roughly 6 million samples, while the A2C agent achieved better results with around 1 million samples. As A2C scales and performs significantly better than DQN, it should be preferred over it. Unfortunately, due to the time constraint, we could not test more advanced DQN implementations like Rainbow DQN [68].

We can also see the difference between DeepMind's agent and ours. We clearly beat DeepMind's agent in *DefeatZerglingsAndBanelings* and *DefeatRoaches*, while the other results are similar. We also compared our agent with A2C with Generalized Advantage Estimation (GAE) and vanilla A2C and we get results that match the comparision between DeepMind's and our agent. Although our vanilla agent also performed slightly better than DeepMind's agent on the respective maps, Generalized Advantage Estimation clearly lead to a boost in mean scores.

In Figure 4.2 to 4.7, sample plots can be seen of our agents. Unfortunately, due to a bug the plotting ended being cut off prematurely. The y-axis is the mean score and the x-axis the number of episodes. We want to note that 'jumps' in performance always coincide with a strategy change of the agent. We will go into more detail in a task by task result analysis:

MoveToBeacon
: Due to this map being a form of unit testing, the expected the agent found the optimal solution very easily and but suprisingly fast compared to the other minigames as it takes around 10 minutes in wall clock time. Off particular interest is that the agent's results jump very suddenly as one can see in Figure 4.2. Our DQN agent takes longer and there is no 'jump' in results.

CollectMineralShards

: CollectMineralShards is a variant of the TSM Problem and therefore NP-hard. It is suprisingly hard to learn in terms of complexity for an agent compared to other minigames. Ongoing research in cognitivie psychology observed that humans are able to produce near optimal solutions to the TSM problem and even an order of magnitude better than well-known heuristics [69][70]. We assume that Deep Reinforcement Learning agents can provide more efficient ways of computing NP-complete problems in the future. Our DQN agent's strategy is to move in the approximate direction of the Shards, despite behaving rather randomly sometimes. The 'optimal' strategy of our A2C agent is to move both units together, this leads to a decent score but is far from the perfect strategy. Compared with TSM Scripted Agent, we achieved only an accuracy of around 50% as the TSM agent reached a score of around 198, while the DQN agent only achieves 25%.

FindAndDefeatZerglings
: Our agent achieved similar results to DeepMind's but far from optimal. We assume this is due to the nature of the partial observed state due to the fog-of-war.

DefeatZerglingsAndBanelings
: We expected the minigame to be quite challenging because the enemy army consists of several units with different abilities, strength and weaknesses but our agent was suprisingly good and learns to reduce exposure to Banelings but lacks efficient micromanagement. We assume this to be the reason that the agent did not achieve human level scores.

DefeatRoaches
: As DefeatZerglingsAndBanelings, we have better results than DeepMind

but lack a human level score. The most likely optimal strategy would be to efficiently micromanagement the units.

CollectMineralsAndGas
Our results were slightly below DeepMind's. However, we assume this to a lack of training time. The general solution of our agent was to send the workers to gather resources.

BuildMarines
As this minigame is the biggest subset of the full-game, it is clearly the most complex. Our expectations were confirmed by our results that this map is most likely not beatable without a form of long term strategy as could be achieved with LSTMs. The best strategy our agent has learned, is to send the work to gather resources and then do nothing for the remaining time.

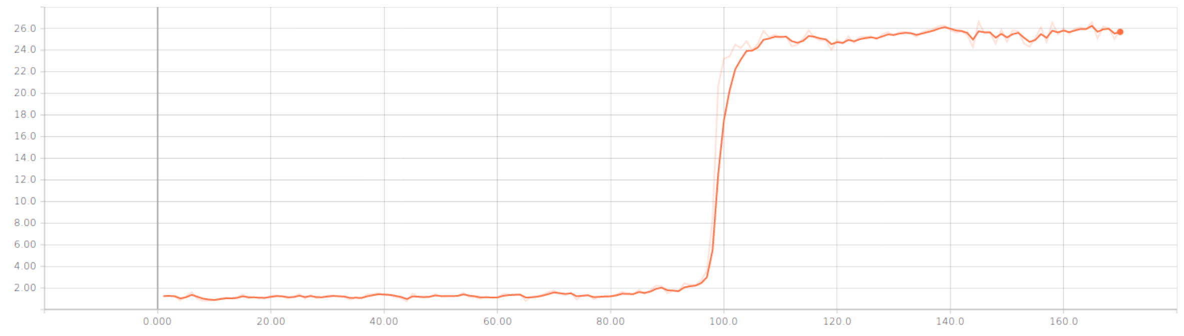


Figure 4.2: MoveToBeacon

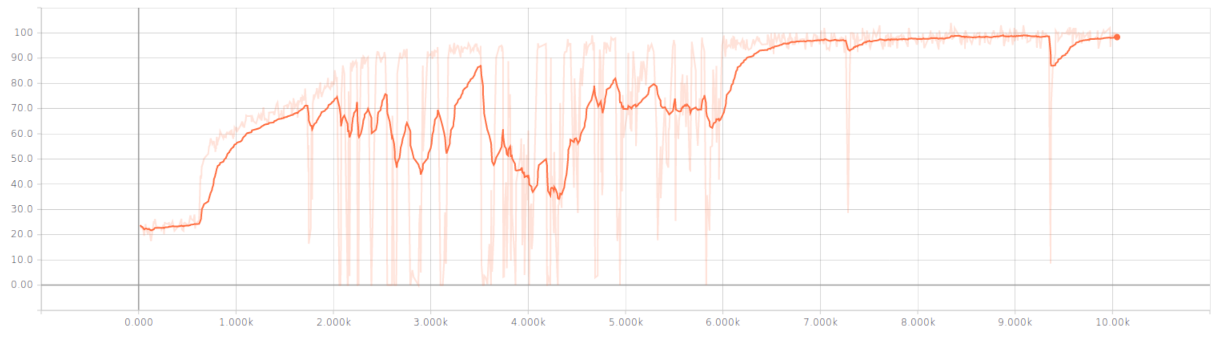


Figure 4.3: CollectMineralShards

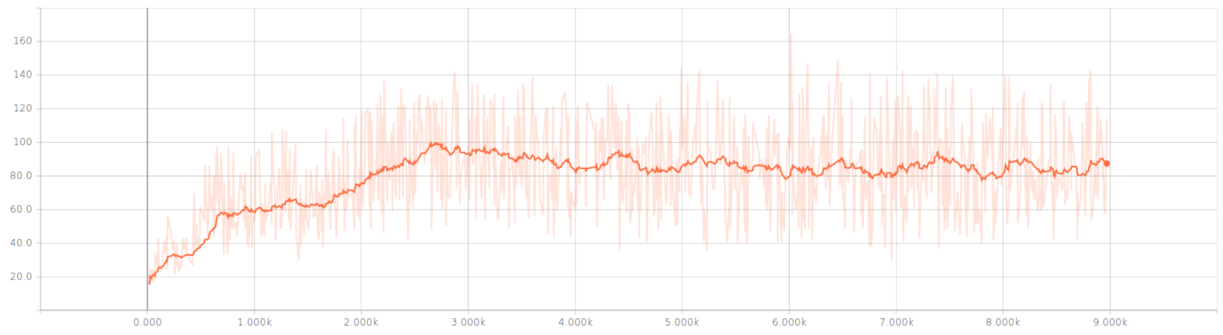


Figure 4.4: DefeatRoaches

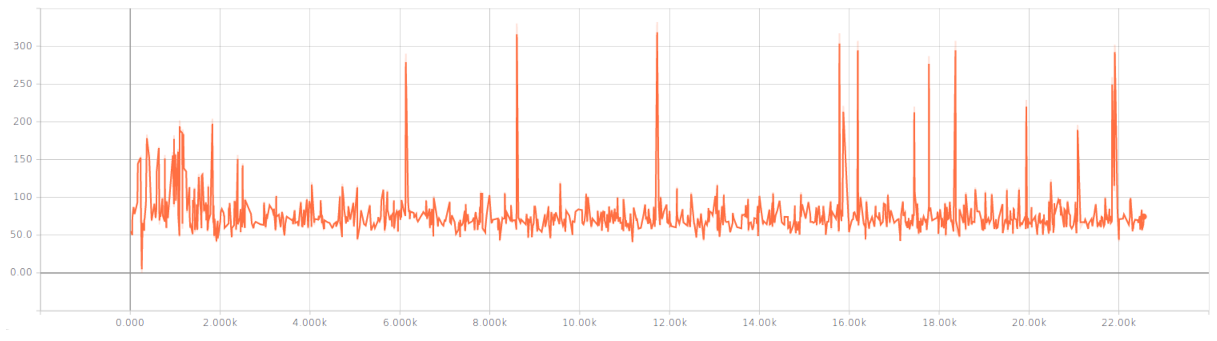


Figure 4.5: DefeatZerglingsAndBanelings

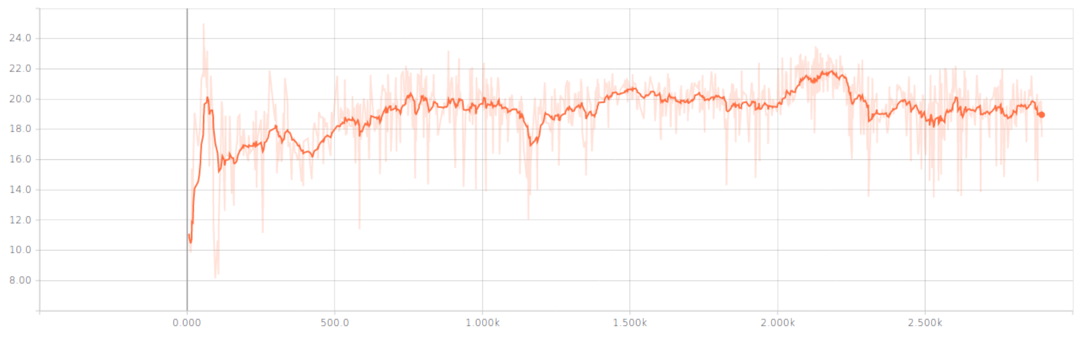


Figure 4.6: FindAndDefeatZerglingsbig

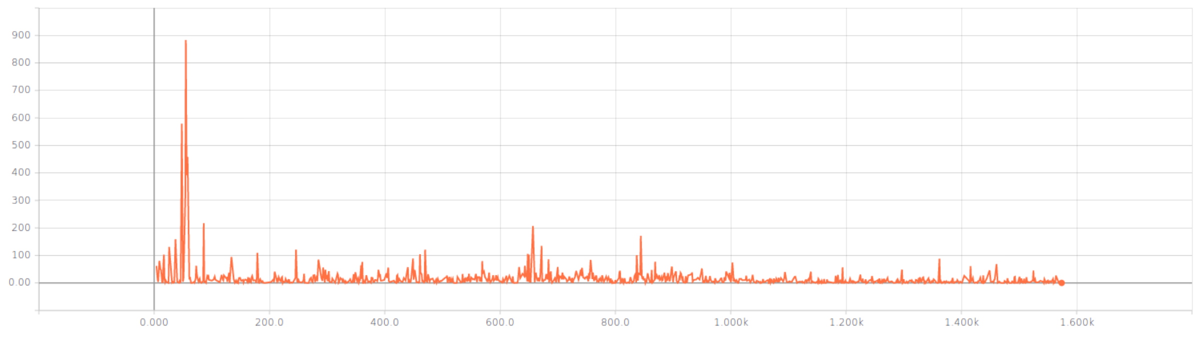


Figure 4.7: CollectMineralsAndGasbig

Chapter 5

Conclusion

In this thesis, we presented the effectiveness of different machine learning algorithms in the StarCraft II Learning Environment. At first we created a Deep Q-Network agent that had decent results but it is a computationally impractical approach due to the complexity of the benchmark environment. Unfortunately, due to time constraints and lack of prior experience, we were not able to investigate every idea and settled for a more conservative approach of testing 2 of the most prevalent agents in reinforcement learning.

Recently DeepMind published their Relational Deep Reinforcement Learning (RDRL) model, which exceeded the baseline results and also the professional player in 6 out of 7 minigames. While this is significant, our improvements of A2C in form of Generalized Advantage Estimation suggest that current algorithms are not nearly at their limits. We have shown that our implementation is capable of surpassing the baseline in 2 minigames with minor improvements. An alternative successful example of this is Rainbow DQN, which is a combination of improvements to the DQN algorithm [68]. Additionally, our TSM Scripted agent seemes to be an almost optimal approach as DeepMind's RDRL agent achieved the same mean score of around 198 in *CollectMineralShards*.

A video comparing results, that is untrained and trained A2C agent, will be found at `https://www.youtube.com/channel/UCo72ShAMnW9OZhI5YrGlvdA`.

5.1 Future Work

As our A2C agent is marginal better or on par with DeepMind's baseline, we would like to see an extension of our approach in future works. In particular, the replacement of vanilla policy gradient by Proximal Policy Optimization (PPO) or Trust Region Policy Optimization (TRPO) [71][72]. Additionally to make A2C more sample efficient Experience Replay could be added (ACER) [73]. Due to the nature of StarCraft II, RNNs and especially LSTMs are also of interest. In particular for the full game as long-term planning needs some form of memory.

At the beginning of this year DeepMind also presented Importance Weighted

Actor-Learner Architectures (IMPALA) [74], which is even more scalable and sample efficient than A2C and could be a viable replacement. Utilizing IMPALA DeepMind's Relational Deep Reinforcement Learning [75] approach is currently the most promising approach to solve the game of StarCraft II, we would like to see improvements and further investigations of that.

Instead of working with feature layers, an agent can also learn from raw RGB pixels as the newly updated PySC2 allows. This resembles the learning of a human more closely and should be a longterm goal. To transition to the full game, it might make sense to limit the action space significantly, thus only allowing a specific unit composition as there are countless examples of players reaching a high level in the game by mastering a single strategy. For example the player 'pseudomota' moved from Bronze to Diamond League, which is among the highest leagues in StarCraft II, by only building Marines [76].

Hierarchical Reinforcement Learning [77] seems also to be an interesting approach as that it closely models how real players approach the game by macro and micro strategies. For that reason we propose the following approach. The first agent would operate on a meta level, deciding 'high-level' strategies and a second agent uses those strategies to act in the game and decide on 'low-level' strategies, such as short-term trade-offs like building an army or focussing on gathering resources.

Furthermore, the evaluation of semi-supervised approaches using the provided replays to additionally train networks should be investigated. This is also one of the most common training strategies among human players as there is no 'fog-of-war' and the player can analyse the opponents behaviour easily.

Bibliography

[1] Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A.S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T.P., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., Tsing, R.: Starcraft II: A new challenge for reinforcement learning. CoRR **abs/1708.04782** (2017)

[2] Patterson, F.G.P., Cohn, R.H., Paiterson, G.P., Cohn, R.: Language acquisition by a lowland gorilla: Koko's first ten years of vocabulary development. (2015)

[3] Rajpurkar, P., Irvin, J., Zhu, K., Yang, B., Mehta, H., Duan, T., Ding, D., Bagul, A., Langlotz, C., Shpanskaya, K., Lungren, M.P., Ng, A.Y.: Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. CoRR **abs/1711.05225** (2017)

[4] OpenAI: Learning Dexterous In-Hand Manipulation. Technical report

[5] Popenici, S.A.D., Kerr, S.: Exploring the impact of artificial intelligence on teaching and learning in higher education. Research and Practice in Technology Enhanced Learning **12**(1) (Nov 2017) 22

[6] Trippi, R.R., Turban, E., eds.: Neural Networks in Finance and Investing: Using Artificial Intelligence to Improve Real World Performance. McGraw-Hill, Inc., New York, NY, USA (1992)

[7] Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal of research and development **3**(3) (1959) 210–229

[8] Tesauro, G.: Temporal difference learning and td-gammon. Communications of the ACM **38**(3) (1995) 58–68

[9] Campbell, M., Hoane Jr, A.J., Hsu, F.h.: Deep blue. Artificial intelligence **134**(1-2) (2002) 57–83

[10] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529** (2016)

[11] Entertainment, B.: Starcraft ii. https://starcraft2.com/en-us/

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540) (2015) 529

[13] DeepMind: Pysc2 - starcraft ii learning environment. https://github.com/deepmind/pysc2 (2017)

[14] Sutton, R.S., Barto, A.G., et al.: Reinforcement learning: An introduction. MIT press (1998)

[15] Markov, A.A.: The theory of algorithms. (1953)

[16] van Otterlo, M., Wiering, M.: Reinforcement learning and markov decision processes. In: Reinforcement Learning. Springer (2012) 3–42

[17] Angel, E., Bellman, R.: Dynamic programming and partial differential equations. Elsevier (1972)

[18] Puterman, M.L., Shin, M.C.: Modified policy iteration algorithms for discounted markov decision problems. Management Science **24**(11) (1978) 1127–1137

[19] Tokic, M., Palm, G.: Value-difference based exploration: adaptive control between epsilon-greedy and softmax. In: Annual Conference on Artificial Intelligence, Springer (2011) 335–346

[20] Watkins, C.J.C.H.: Learning from delayed rewards. PhD thesis, King's College, Cambridge (1989)

[21] Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning **8**(3-4) (1992) 229–256

[22] Hornik, K.: Approximation capabilities of multilayer feedforward networks. Neural networks **4**(2) (1991) 251–257

[23] Hoffmann, T.: Learning and intelligence systems script. (2017)

[24] Haykin, S.S., Haykin, S.S., Haykin, S.S., Haykin, S.S.: Neural networks and learning machines. Volume 3. Pearson Upper Saddle River, NJ, USA: (2009)

[25] LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. nature **521**(7553) (2015) 436

[26] Verhulst, P.F.: NOUVEAUX MÉMOIRES DE L'ACADÉMIE ROYALE DES SCIENCES ET BELLES-LETTRES DE BRUXELLES

[27] Hahnloser, R.H., Sarpeshkar, R., Mahowald, M.A., Douglas, R.J., Seung, H.S.: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature **405**(6789) (2000) 947

[28] Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. (2011) 315–323

[29] : Labeled faces in the wild. http://vis-www.cs.umass.edu/lfw/

[30] Taigman, Y., Yang, M., Ranzato, M., Wolf, L.: Deepface: Closing the gap to human-level performance in face verification. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2014) 1701–1708

[31] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11) (1998) 2278–2324

[32] : Cs231n convolutional neural networks for visual recognition. http://cs231n.github.io/convolutional-networks/

[33] Lin, M., Chen, Q., Yan, S.: Network in network. arXiv preprint arXiv:1312.4400 (2013)

[34] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. (2012) 1097–1105

[35] Silver, D.

[36] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., De Freitas, N.: Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581 (2015)

[37] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. (2016) 1928–1937

[38] Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., et al.: Massively parallel methods for deep reinforcement learning. arXiv preprint arXiv:1507.04296 (2015)

[39] : Brood war api - api for interacting with starcraft: Broodwar. https://bwapi.github.io

[40] Entertainment, B.: Starcraft: Broodwar. https://starcraft.com/en-us/

[41] : Bwapi in academia. https://github.com/bwapi/bwapi/wiki/Academics

[42] : Bwapi research papers. https://github.com/Eric-Wallace/starcraft-research-papers

[43] : Aiide starcraft ai competition. http://www.cs.mun.ca/~dchurchill/starcraftaicomp/

[44] : Student starcraft ai tournament. https://sscaitournament.com

[45] : Ieee conference on computational intelligence and games. http://www.ieee-cig.org

[46] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)

[47] : Sc2 ingame screenshot. https://news-cdn.softpedia.com/images/news2/Quick-Look-StarCraft-2-Legacy-of-the-Void-with-Gameplay-Video-and-Screenshots-4
jpg

[48] : Chris-chris.ai image of feature layers of old pysc2 version. http://chris-chris.ai/2017/11/06/pysc2-tutorial2/

[49] : Documentation of pysc2. https://github.com/deepmind/pysc2/blob/master/docs/environment.md

[50] Hochreiter, S., Schmidhuber, J.: Lstm can solve hard long time lag problems. In: Advances in neural information processing systems. (1997) 473–479

[51] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y.: Openai baselines. https://github.com/openai/baselines (2017)

[52] : Baselines dqn example. https://github.com/openai/baselines/blob/master/baselines/deepq/deepq.py

[53] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015) Software available from tensorflow.org.

[54] Oliphant, T.E.: A guide to NumPy. Volume 1. Trelgol Publishing USA (2006)

[55] Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. arXiv preprint arXiv:1511.05952 (2015)

[56] Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., Kautz, J.: Reinforcement learning thorugh asynchronous advantage actor-critic on a gpu. In: ICLR. (2017)

[57] : Nvidia's ga3c repository. https://github.com/NVlabs/GA3C

[58] : Openai's a2c repository. https://github.com/openai/baselines/tree/master/baselines/a2c

[59] : A2c reference implementation from awjuliani. https://github.com/awjuliani/DeepRL-Agents

[60] : A2c reference implementation from xhujoy. https://github.com/xhujoy/pysc2-agents

[61] : A2c reference implementation from pekaalto. https://github.com/pekaalto/sc2aibot

[62] : A2c reference implementation from inoryy. https://github.com/Inoryy/pysc2-rl-agent

[63] : A2c reference implementation from simonmeister. https://github.com/simonmeister/pysc2-rl-agents

[64] : Tensorflow performance guide. https://www.tensorflow.org/performance/performance_guide

[65] Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P.: High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438 (2015)

[66] Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied Mathematics **10**(1) (1962) 196–210

[67] : Documentation of minigames. https://github.com/deepmind/pysc2/blob/master/docs/mini_games.md

[68] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D.: Rainbow: Combining improvements in deep reinforcement learning. arXiv preprint arXiv:1710.02298 (2017)

[69] MacGregor, J.N., Ormerod, T.: Human performance on the traveling salesman problem. Perception & Psychophysics **58**(4) (1996) 527–539

[70] Dry, M., Lee, M.D., Vickers, D., Hughes, P.: Human performance on visually presented traveling salesperson problems with varying numbers of nodes. The Journal of Problem Solving **1**(1) (2006) 4

[71] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

[72] Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P.: Trust region policy optimization. In: International Conference on Machine Learning. (2015) 1889–1897

[73] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., de Freitas, N.: Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224 (2016)

[74] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al.: Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. arXiv preprint arXiv:1802.01561 (2018)

[75] Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., et al.: Relational deep reinforcement learning. arXiv preprint arXiv:1806.01830 (2018)

[76] : Bronze to diamond league making only marines. `www.twitch.tv/zeezdrdevice`

[77] Kulkarni, T.D., Narasimhan, K., Saeedi, A., Tenenbaum, J.: Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In: Advances in neural information processing systems. (2016) 3675–3683