Deep Reinforcement Learning for StarCraft II

Daniel Palenicek *
TU Darmstadt
daniel.palenicek@gmail.com

Marcel Hussing *
TU Darmstadt
marcel.hussing@gmail.com

Simon Meister *
TU Darmstadt
mail@simonmeister.org

Abstract

Deep reinforcement learning (RL) methods recently had increasing success in learning a wide range of toy problems directly from high-dimensional low-level percepts, including atari games and simple robot manipulation tasks. To encourage work on complex environments which require addressing, among other challenges, learning with very sparse rewards and large action spaces, as well as long-term planning, the modern StarCraft II strategy video game was recently opened up for AI research. Due to the considerable complexity and large time scale of the full game, a set of *mini-games* encompassing common sub-tasks of the full game was introduced. However, all but the simplest sub-tasks remain intractable for current deep RL agents, and solving the full game would likely require considerable breakthroughs. In this project, we therefore set out to explore the shortcomings of current approaches, with the aim of developing methods which can learn more complex sub-tasks of the game. As a first step, we implement a simple state-of-the-art deep RL agent and use it to learn public mini-games.

1 Introduction

Progress in artificial intelligence research is often bound to the availability of adequate training data and environments. One recent development is the turn towards complex consumer video games for training and evaluating AI agents [8, 10, 14, 13, 24]. Video games offer virtually unlimited potential for complexity while being inexpensive to work with, and existing games are often carefully engineered to require deliberate planning, strategic thinking, as well as fast reaction times and well-orchestrated low-level control. Additionally, many video games offer a well-defined measure of success. With the advent of deep reinforcement learning, it became possible to train agents on high-dimensional low-level input, and thus video games can often be learned from passing raw pixel observations through a deep network, with minimal game-specific pre-processing or feature engineering [10].

In this paper, we will focus on the StarCraft II Learning Environment (SC2LE), which provides programmatic access to the popular real-time strategy (RTS) game StarCraft II to enable the development and training of AI agents [24]. A session of the complete StarCraft II game often takes 30 minutes or more, during which the player needs to gather resources, develop technologies, construct buildings and units, and defeat his enemies by destroying their buildings. In the unaltered game, the binary reward (win/loss) occurs upon completion of the whole game, which is too sparse for current RL agents to make any learning progress at all. A shaping reward was introduced based on runtime game statistics, however, agents trained this way generally only converge to trivial local minima of the runtime statistics. To allow any notable progress at all, the SC2LE comes with 7 mini-games of varying difficulty, each representing a simplified aspect or sub-task of the full game. The original release of SC2LE includes results of a simple parallel advantage actor-critic [15] trained on multicore CPUs. This agent is able to learn mini-games easily which require short reaction times and basic visual inference, however it fails to develop strategies for mini-games which require acting over longer time horizons.

*Joint first authors. Ordered alphabetically by first name.

To explore the shortcomings of approaches based on state-of-the-art deep RL methods, we first implemented a similar actor-critic method and re-produced the original results on 5 out of 7 mini-games. We noted the following issues:

1. For more complex tasks, the extrinsic rewards alone often seem to be insufficient to allow the discovery of sensible strategies.
2. Due to the time scale of some tasks, exploration at the level of individual actions may make convergence to sensible strategies slow, or impossible.
3. The results are highly dependant on hyperparameter settings (e.g. learning rate), and due to the training time required, it can often be difficult to find hyperparameters that work well for a given game.
4. Sample efficiency is low, and even games which are only reaction-based may take hundreds of thousands of episodes of learning.

2 Related Work

Video games for reinforcement learning. The Atari Learning Environment (ALE) is currently highly popular in deep RL research and offers a large collection of simple video games with discrete action spaces and raw pixel observations at each game step [10]. Compared to Atari, real-time strategy games (RTS) often possess much larger state and action spaces, operate on larger timescales with very sparse rewards, and the fraction of the state that can be observed at once is smaller.

Deep reinforcement learning. One of the first deep reinforcement learning methods to learn Atari games from raw pixels was the deep Q-network (DQN), which uses a neural network for action-value approximation within a Q-learning framework [21]. In DQN, a crucial component to increase sample-efficiency are off-policy updates using experience replay, where past experiences stored in a replay buffer are randomly sampled for mini-batch learning [3].

The baseline method from the original SC2LE work employs an on-policy actor-critic method [15, 24]. In complex environments, learning a good policy may be easier than learning a good value function, and thus policy gradient methods (e.g. actor-critic) may be preferrable. Unlike Q-learning, standard policy gradient methods are used with on-policy learning. Although on-policy methods are often regarded as being more stable, multiple agents have to be simulated in parallel to increase exploration and decrease correlation between learning updates [15]. Multiple agents in parallel environments come with the benefit of a super-linear decrease in training time, however they require a larger amount of system resources for environment simulation and often imply very large update batch sizes.

Natural policy gradients. In policy gradient methods, the learning update is done in parameter space while the agent explores in policy space. Small changes in the parameters of the model might lead to a large change in the policy itself. The natural gradient [4] tries to solve this problem by putting a constraint on the parameter update to ensure the closeness of the old policy and the updated policy to stabilize learning. The idea to use this constraint in reinforcement learning has surfaced multiple times over the years, for example natural policy gradient [6], eNAC [7], REPS [9].

Trust Region Policy Optimization (TRPO) is a deep reinforcement learning algorithm following the natural gradient idea [12]. It uses an average over samples to estimate the objective function as well as the Kullback-Leibler divergence as a constraint and then solves the constraint using the conjugate gradient method to make updates to a neural network. Using natural gradients in StarCraft 2 could potentially speed up the learning process and create more stable parameter updates.

Off-policy policy gradient methods. To make policy gradient methods more sample efficient, recent work attempted to combine them with experience replay (and thus off-policy updates). The actor-critic with experience replay (ACER) employs this idea and various other innovations in policy gradient RL to increase sample efficiency [25]. For example, ACER incorporates a parallelizable structure and ideas from TRPO. To enable off-policy updates, ACER uses the Retrace algorithm [16], which is also used in a similar recent off-policy policy gradient method dubbed Retrace-actor (Reactor) [19]. In StarCraft 2, using experience replay with an actor-critic may make the training more sample-efficient, while keeping the desirable properties of policy gradient methods.

Hierarchical reinforcement learning. The goal of hierarchical reinforcement learning is to divide the sequential decision problem into smaller sub-problems (once or repeatedly), which are often easier to solve. In feudal reinforcement learning, workers execute low-level actions, while managers operate on a coarser scale, assigning goals to workers to maximize extrinsic reward [2]. Building on this framework, feudal networks employ a *temporal* manager-worker hierarchy, where goals from the manager network, represented as directions in a latent state space, modulate the actions of the worker network [23]. Due to the decreased temporal resolution of the manager, feudal networks were shown to explore and solve Atari problems with deferred rewards and longer timescale more effectively, and thus may also help in the case of StarCraft II.

Hindsight for sparse rewards. Off-policy hindsight experience replay (HER) can improve multi-goal learning when rewards are very sparse by re-playing trajectories and assuming that the final states achieved were actually the goal states [17]. This requires that goals are explicitly encoded as network inputs. Recently, an on-policy version based on importance sampling was proposed [22]. The hierarchical actor-critic (HAC) combines off-policy HER with temporal HRL, thus exploiting hindsight to more effectively train the HRL agent with sparse rewards [20]. A combination of these ideas may be useful to learn more complex mini-games in StarCraft II.

3 Background

3.1 Reinforcement Learning

In model-free reinforcement learning, at each time step t of our environment, we are given a state s_t, take an action a_t according to our policy π, and receive a reward r_t. The future *return* $R_t = \sum_{k=t}^{\infty} \gamma^{(k-t)} r_k$ is the sum of all rewards received from time t onwards, where γ is a discount factor to decay the influence of temporally distant rewards. The goal is to maximize the expected return $\mathbb{E}[R_t]$ from each state onward.

The *value* of a state s is the expected return from this state under policy π, $V^{\pi}(s) = \mathbb{E}[R_t | s_t = s]$. The *action value* of an action a in state s is the expected return when executing a and following π afterwards, $Q^{\pi}(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a]$. We can now define the *advantage* of action a in state s as $A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$.

4 Deep Reinforcement Learning for StarCraft II

4.1 The Starcraft II Learning Environment

The SC2LE allows the agent to naturally interact with StarCraft II, similar to how a human player would. Thus, the agent observes the *screen* segment of the detailed 2D game map at each time step together with a coarse *minimap* representation of the full map and subsequently performs an action.

Observations. Instead of being given as raw RGB pixels, minimap and screen observations are structured into multiple 2D feature layers with fixed resolution (Figure 1). For example, a feature layer may represent the type of unit at a specific position, or the height of the game terrain (i.e. serve as *heightmap*). Additionally, scalar observations are provided, such as the total amount of resources (e.g. *minerals*) collected. Scalar observations or individual layer values are either categorical or integer-valued. Categorical observations may have up to 1850 categories. Overall, we use all 17 layers of screen observations, all 7 layers of minimap observations, and the 11 non-spatial *player* observations.

Actions. The StarCraft II action space is structured into 524 different functions, each of which takes a fixed number of discrete spatial and non-spatial arguments, with up to thousands of choices for the value of each argument. There are 13 distinct types of arguments, and every function may have up to one argument of each type, or no arguments at all. A naive representation of the action space would require predicting intractably large policies. Therefore, the baseline work proposed to

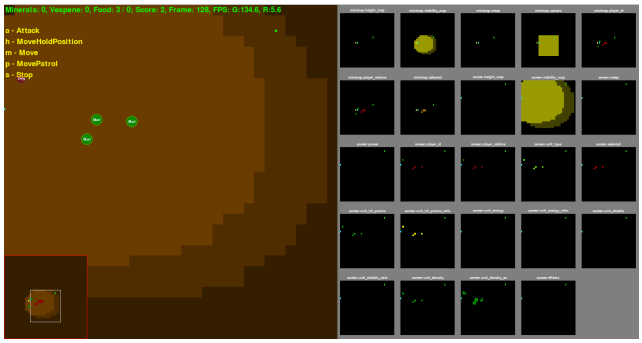


Figure 1: On the left, we show a simplified game rendering, similar to what a human player would see when playing the game. A large, high-resolution screen provides a detailed view of a small section of the map. Embedded into this view, the mini map on the bottom left provides a coarse overview of the full map. On the right, we show the spatial minimap and screen observations, which are structured into feature layers.

model actions in an auto-regressive manner. This policy models the probability of an action a as the joint probability of its function identifier a^0 and arguments a^l, $l \in \{1, ..., L\}$,

$$\pi(a|s) = \prod_{l=0}^{L} \pi(a^l|a^{<l}, s) \approx \prod_{l=0}^{L} \pi(a^l|s), \tag{1}$$

where L is the number of arguments used by the function type a^0. This approach transforms the problem of choosing an action to a sequence of choosing function arguments, where each chosen argument's probability depends on the arguments that have been chose already. Note that in this work we model arguments as being independent of each other for now, which was also proposed by DeepMind and empirically shown to suffice for most of the current mini-games. This approximation is shown in equation (1). In the future, properly conditioning on previous arguments of the argument chain within the network architecture may be helpful. As an example, consider the `select_rect` function, $a^0 = 3$, which allows the player to draw a rectangle over an area of the screen to select multiple units at once. Here, we have $L = 3$, with a non-spatial argument to determine the type of selection operation (new selection or add to existing selection), and two spatial arguments to determine the top-left and bottom-right screen locations of the selection rectangle.

4.2 Network Architecture

We pass all pre-processed observations through the *FullyConv* network proposed in [24] and obtain the predicted value of the observed state as well as the action predictions. A schematic of the network architecture is shown in Figure 2. Note that the complete network leads to a policy with approximately 20 million parameters.

Observation pre-processing. Integer values are logarithmically transformed to avoid very large input values (e.g. the integer unit hit points range up to 1600). Categorical values are first transformed to a high-dimensional one-hot representation, which is then embedded into a lower-dimensional representation using a fully connected or convolutional layer for non-spatial or spatial observations, respectively. For each value, we use the base-2 logarithm of the number of categories as the number of embedding dimensions, and the embedding is computed for each observation value or layer in isolation. This avoids an excessively large number of input channels (e.g. the unit type can have up to 1850 categories) while preserving most information.

FullyConv network. First, the pre-processed spatial minimap and screen observations are processed by two separate convolutional streams, each of which consists of 5×5 and 3×3 convolutional layers with 32 and 64 output channels, respectively. Then, the non-spatial observations are broadcast over all pixels of the spatial observation domain and depth-concatenated with the results of the

convolutional stream. The scalar state value is predicted from the flattened *state representation* using a fully-connected layer.

Action prediction. For each state, there is one indepent prediction for the function identifier a^0 as well as the 13 possible argument types. One separate fully-connected output layer coming from the flattened state representation is used for each of the non-spatial argument types and the function identifier. Spatial argument types are predicted from the 2D state representation with separate 1×1 convolutional layers with 1 output channel each, and are subsequently flattened to obtain distributions over all pixel positions. All raw action outputs are converted to valid probability distributions using the softmax function.

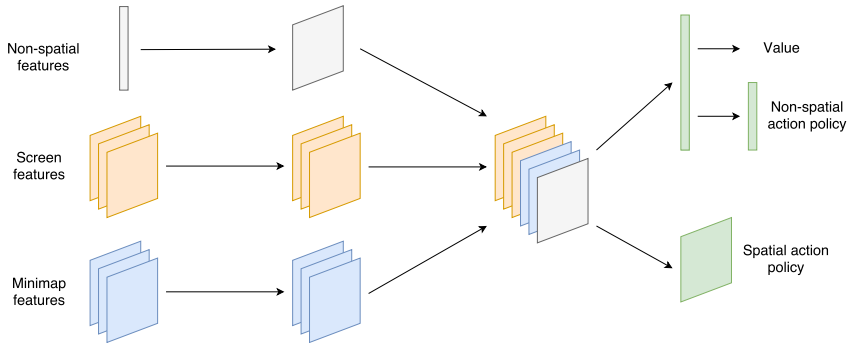


Figure 2: The FullyConv network takes spatial and non-spatial observations as input and produces the predicted value of the observed state as well as the policy as output.

4.3 Advantage Actor-Critic

As in the baseline work [24], our deep reinforcement learning agent is based on the Asynchronous Advantage Actor-Critic (A3C) introduced in [15]. Unlike the baseline work, we implement a synchronous batch variant of A3C, commonly referred to as A2C, to enable reasonably fast training on a GPU [18]. A2C collects updates from parallel agents into batches, synchronizing after each agent completed its trajectory to perform GPU-based learning steps, whereas A3C runs all agents on independent CPU threads and performs asynchronous updates. GPU-based training is much faster in general. For each training step, we sample trajectories of T state-action pairs from N parallel environments. Along each trajectory, we calculate the discounted n-step return from the single-step rewards, where the return of the last step is approximated by the predicted value of the last state. This results in a collection of $T \cdot N$ states, actions and returns, which are used to update the neural network in a single step of backpropagation.

Action sampling. When sampling actions from the predicted policy, the function identifier will be sampled first, followed by the arguments determined by the type of the sampled function. All predictions for argument types unused by this function will be ignored.

Loss. The advantage actor-critic loss is composed of a policy loss, a value loss and an entropy penalty to encourage exploration. The policy loss is based on the policy gradient theorem, which employs the likelihood-ratio policy gradient with the approximate future returns at each time step [1, 5]. As a baseline, the predicted values are subtracted from the returns to yield advantages at each step. The value loss is simply the mean-squared error between the predicted values and the approximate future returns at each time step. The complete A2C method is summarized in Algorithm 1.

5 Experiments

5.1 Implementation

The code release accompanying [24] provides an API to interact with StarCraft II, but does not include any code to replicate their reinforcement learning baseline. Thus, we implemented the baseline

Algorithm 1 Advantage actor-critic, synchronous batch variant (A2C)

Initialize network parameters θ
$t \leftarrow 0$
repeat
  Get initial state s_0
  for $i \in \{t, \ldots, t+T\}$ **do**
    for $n \in \{1, \ldots, N\}$ **do** ▷ parallel
      Perform a_i^n according to policy $\pi(a_i^n | s_i^n; \theta)$
      Receive reward r_i^n and new state s_{i+1}^n
      Reset environment if s_{i+1}^n is terminal
    end for
  end for
  for $n \in \{1, \ldots, N\}$ **do**
    $R_{t+T}^n = \begin{cases} 0 & \text{for terminal } s_{t+T}^n \\ V(s_{t+T}^n; \theta) & \text{for non-terminal } s_{t+T}^n \end{cases}$
    for $i \in \{t+T-1, \ldots, t\}$ **do**
      $R_i^n \leftarrow r_i^n + \gamma R_{i+1}^n$
    end for
  end for
  Compute policy loss $l_p = -\frac{1}{TN} \sum_i \sum_n \log \pi(a_i^n | s_i^n; \theta)(R_i^n - V(s_i^n; \theta))$
  Compute value loss $l_v = \frac{1}{TN} \sum_i \sum_n (R_i^n - V(s_i^n; \theta))^2$
  Compute entropy loss $l_e = \frac{1}{TN} \sum_i \sum_n \sum_a \log(\pi(a | s_i^n; \theta)) \pi(a | s_i^n; \theta)$
  $l = l_p + \alpha l_v + \beta l_e$
  Differentiate l wrt θ and perform SGD update on θ
  $t \leftarrow t + T$
until convergence

method from scratch and released an initial open source version [*]. To the best of our knowledge, this is the only reliable re-implementation that is publicly available at the time of writing. Other re-implementations exist, however they are very incomplete and lack generality, for example, by only supporting a subset of the action space. Our implementation of the network architecture and reinforcement learning agent is based on the TensorFlow framework, which enables automatic loss differentiation and supports GPU training [11]. We run all experiments on a Titan X Pascal GPU with 12GB of memory.

5.2 Optimization and Hyperparameters

We perform stochastic gradient descent on the combined loss using RMSprop with $\epsilon = 1 \times 10^{-5}$ and a decay of 0.99. Our initial learning rate of 7×10^{-4} follows an exponential decay by 0.94 all 10K iterations. The loss hyperparameters are fixed to $\alpha = 0.5$, $\beta = 1 \times 10^{-3}$ for all of our experiments, and we clip gradients exceeding a norm of 1.0. For reinforcement learning, we use $N = 32$ parallel environments and a trajectory length of $T = 16$. The agent observes the environment and takes actions every 8 atomic game steps, and we refer to this scaling factor as the agent time step. We use a minimap and screen resolution of 32×32.

5.3 Mini-Games

The mini-games are played in episodes, after which the game will be terminated and re-started. Mini-games terminate after a fixed time limit, and there are additional termination conditions depending on the specific mini-game. After each episode, the cumulative reward from the episode is recorded.

MoveToBeacon. The agent has a single unit which is pre-selected and has to move this unit (by placing clicks at the desired target location) towards a *beacon* that appears at a random position on the map. Upon reaching the beacon, the agent gets a reward of $+1$ and the beacon re-appears at a different, random location. The map fits onto the screen, and thus the beacon is always visible to the agent. One episodes takes 120 game seconds.

[*] https://github.com/simonmeister/pysc2-rl-agents

(a) `MoveToBeacon` (b) `CollectMineralShards` (c) `FindAndDefeatZerglings`

(d) `DefeatRoaches` (e) `DefeatZerglingsAndB.` (f) `CollectMineralsAndGas`

Figure 3: Screenshots of selected mini-games. Videos of our agents can be found at goo.gl/nE2D5H.

CollectMineralShards. The agent starts with 2 units, which have to be selected and moved in order to pick up *mineral shards* which are randomly distributed on the map. Upon reaching a shard, a reward of $+1$ is received and the shard disappears. Whenever all shards have been collected, the map is re-filled with mineral shards. One episode takes 120 game seconds. Since the shards are spread all over the map, the agent needs to collect them in an efficient order to receive maximum reward.

FindAndDefeatZerglings. The agent starts with 3 units and has to explore the map to find and defeat enemy units (*zerglings*). Defeating a zergling yields a reward of $+1$ and losing a unit is penalized with a reward of -1. Only a fraction of the map is visible at once, and the agent needs to control its camera and units to uncover new sections which may contain zerglings. Zerglings randomly re-appear and one episode takes 180 game seconds.

DefeatRoaches. The agent starts with 9 *marines* and has to defeat 4 powerful *roaches*. Whenever all roaches are defeated, the agent receives 5 new marines and 4 new roaches are spawned, until 180 game seconds have passed. The instantaneous rewards are $+10$ for defeating a roach and -1 for losing a marine.

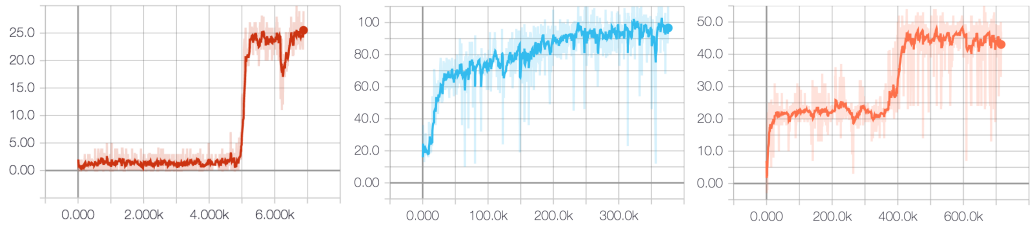
DefeatZerglingsAndBanelings. This mini-game is analogous to *DefeatRoaches*, with the difference that the enemy units are of two different types. It requires the agent to learn a strategy which is able to deal with the different abilities of these enemies. For this game, the rewards are $+5$ for defeating a unit and -1 for losing a marine.

CollectMineralsAndGas. The agent starts with a limited base and a few *worker* units, and is rewarded for collecting resources using the workers. Over time, the agent needs to recruit new units and build a second base to maximize resource collection.

BuildMarines. The agent starts with a limited base and is rewarded for building marines. It needs to create workers to collect resources, expand its base, build a facility for recruiting marines, and finally order the construction of marines.

mini-game	best mean score			episodes (ours)
	random	ours	DeepMind	
MoveToBeacon	1	26	26	8K
CollectMineralShards	17	97	103	300K
FindAndDefeatZerglings	4	45	45	450K
DefeatRoaches	1	65	100	-†
DefeatZerglingsAndBanelings	23	68	62	-†
CollectMineralsAndGas	12	12	3978	-
BuildMarines	<1	-	3	-

Table 1: For each mini-game, we show the mean score achieved by the randomly initialized agent, the best mean score achieved by our agent, the best mean score reported by DeepMind [24], and the number of mini-game episodes (in thousands) that we had to train for in order to achieve our maximum score. The mean score is computed by running the final agent over multiple episodes after it has converged. †For some of the mini-games, we had to restart the training multiple times after crashes due to a presumed memory leak in the StarCraft II simulator, which makes it difficult to read the episodes until convergence from the logs.



(a) MoveToBeacon (b) CollectMineralShards (c) FindAndDefeatZerglings

Figure 4: Episode scores during training for a number of mini-games. For MoveToBeacon, the total number of episodes corresponds to a average training time of 20 minutes, while for more complex games training can take up to 3 days on a single GPU.

5.4 Results on the Mini-Games

Table 1 compares the scores of our A2C agent to the A3C results reported in [24]. We were not able to learn CollectMineralsAndGas, which may be due to our shorter trajectories and smaller number of parallel environments compared to the baseline work [24]. For most of the other mini-games, our results are similar to the original results. The code repository contains videos of trained agents. Figure 4 shows the episode scores during training on the mini-games on which training was successfull.

Note that the score results reported in [24] were their best results after 100 experiments for each game, with a randomly sampled initial learning rate for each experiment. Thus, some of our results may be worse due to our fixed hyperparameters, as we could not afford a large random search. Additionally, experiments in [24] used $T = 40$, $N = 32$, and a screen and minimap resolution of 64×64, all of which would exceed our GPU memory capacity.

6 Conclusion

We have implemented a advantage-actor critic baseline agent for the SC2LE. Compared to the original results, our agent performs similarly on all but two mini-games, which is likely due to hardware limitations. As in the original results, our agent fails to succeed on the more complex BuildMarines mini-game. Thus, as a next step, we want to research algorithms which may improve learning for a given hardware budget, likely based on ideas outlined in the related works. Furthermore, we think that temporally hierarchical reinforcement learning, possibly combined with a hindsight mechanism, may be a promising route towards learning more complex mini-games with deferred rewards, such as BuildMarines, while additionally improving sample-efficiency.

References

[1] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8 (1992), pp. 229–256.

[2] P. Dayan and G. E. Hinton. "Feudal reinforcement learning". In: *Neural Information Processing Systems*. 1993.

[3] L.-J. Lin. "Reinforcement Learning for Robots Using Neural Networks". Dissertation. 1993.

[4] S.-I. Amari. "Natural Gradient Works Efficiently in Learning". In: *Neural Computation* 10.2 (Feb. 1998), pp. 251–276.

[5] R. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Neural Information Processing Systems*. 1999.

[6] S. M. Kakade. "A Natural Policy Gradient". In: *Advances in Neural Information Processing Systems 14*. 2002, pp. 1531–1538.

[7] J. Peters, S. Vijayakumar, and S. Schaal. "Natural Actor-critic". In: *Proceedings of the 16th European Conference on Machine Learning*. Porto, Portugal, 2005, pp. 280–291.

[8] P. Hingsto. "A Turing test for computer game bots". In: *Computational Intelligence and AI in Games*. 2009, pp. 169–186.

[9] J. Peters, K. Mülling, and Y. Altün. "Relative Entropy Policy Search". In: *AAAI Conference on Artificial Intelligence*. 2010.

[10] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[11] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`.

[12] J. Schulman et al. "Trust Region Policy Optimization". In: *arXiv preprint arXiv:1502.05477* (2015).

[13] M. Johnson et al. "The Malmo Platform for Artificial Intelligence Experimentation". In: *International Joint Conference on Artificial Intelligence*. Palo Alto, California, 2016, p. 4246.

[14] M. Kempka et al. "A Doom-based AI research platform for visual reinforcement learning". In: *Computational Intelligence and Games*. 2016, pp. 1–8.

[15] V. Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. New York City, New York, June 2016, pp. 1928–1937.

[16] R. Munos et al. "Safe and Efficient Off-Policy Reinforcement Learning". In: *Advances in Neural Information Processing Systems 29*. 2016, pp. 1054–1062.

[17] M. Andrychowicz et al. "Hindsight experience replay". In: *Neural Information Processing Systems*. 2017, pp. 5055–5065.

[18] P. Dhariwal et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[19] A. Gruslys et al. "The Reactor: A Sample-Efficient Actor-Critic Architecture". In: *arXiv preprint arXiv:1704.04651* (2017).

[20] A. Levy, R. Platt, and K. Saenko. "Hierarchical Actor-Critic". In: *arXiv preprint arXiv:1712.00948* (2017).

[21] V. Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2017).

[22] P. Rauber, F. Mutz, and J. Schmidhuber. "Hindsight policy gradients". In: *arXiv preprint arXiv:1711.06006* (2017).

[23] A. S. Vezhnevets et al. "FeUdal Networks for Hierarchical Reinforcement Learning". In: *International Conference on Machine Learning*. 2017.

[24] O. Vinyals et al. "StarCraft II: a new challenge for reinforcement learning". In: *arXiv preprint arXiv:1708.04782* (2017).

[25] Z. Wang et al. "Sample Efficient Actor-Critic with Experience Replay". In: *International Conference on Learning Representations*. Palais des Congrès Neptune, Toulon, Fr, Apr. 2017.