Deep Reinforcement Learning for Stacraft II

Oliver Yates, Romain Rouvillois, Louis Mutricy and Ramy Ghorayeb
Ecole Polytechnique

January 18, 2019

Abstract

In this report, we evaluate the most recent reinforcement learning methods on the Starcraft 2 Py-SC2 mini-games, and analyze the main challenges that characterizes the difficulty of RL in Real-Time Startegy games: managing delayed and sparse rewards, designing a usable spatial action policy, and having an agent learn to chain several actions as a "path" to lead to the highest reward.

Our code implementation can be found at :
https://github.com/Aetaos/StarCraftReinforcementLearning

1 INTRODUCTION

Real-time strategy games (RTS) represent a serious challenge for state-of-the art reinforcement learning for many reasons: there are multiple agents interacting on the same map, they only have access to part of the information of the game at any given time and must take action to expand it, and rewards are sparse and delayed. These characteristics allow to benchmark a Reinforcement Learning model on long-term strategies and situational awareness (can it adapt to a slightly different situation than during training?). Starcraft 2 (SC2) is a real-time strategy game that incorporates all of the aforementioned characteristics, and benefits from a well-defined ladder of competitive players classified in standardised levels, allowing to easily compare AI models to human performance. The Google DeepMind team published an open-source framework that serializes SC2 into an easily accessible environment on python, PySc2, [1], offering a selection of mini-games of variable difficulty and designed with reinforcement learning in mind.

In this report, we review some of the most recent Deep Reinforcement Learning models that are used in this environment, what are their specificities once applied to a RTS like Starcraft 2, and analyse its main challenges: learning a policy with conceptually different type of actions (e.g. moving somewhere, selecting an entity, or building a certain type of unit or building), learning spatial actions and understanding the concept of delayed rewards. We implement reduced versions of some of these models using **Keras** [2] in order to observe the consequences of these challenges on the most basic mini-game, in which a Marine must move to a beacon that appears at a random spot on a small map.

2 GAME SPECIFICS

2.1 RTS CHARACTERISTICS

Starcraft 2 (SC2) is a real-time strategy game, where two players or more win by finding and destroying all of their opponent's bases. The game has 3 main components, which are necessary to master to beat the game:

- **Resource management**: Collecting resources to generate workers, soldiers and buildings.
- **Infrastructure improvement**: Building new infrastructure and technologies to improve the player's abilities
- **Attack/Defense**: Generate soldiers and use them efficiently to protect the player's base and defeat the opponents'

Each of these three components are substantially hard to master by a human player and often require hundreds of hours of experience to reach a decent level - and it gets even harder for a reinforcement learning agent.

The game can only be observed through a camera which needs to be moved on a larger map to get a perspective of the situation (with some higher level information being available on a small resolution map called the "minimap"). Furthermore, there is a "fog of war" mechanism, hiding the unobserved game areas which limits the player's view of the situation, and emphasizes the need for scouting. At any moment, each player control hundreds of units, dozens of buildings, and can generate many more. Each unit and building has its own specificities and possible actions, and be manipulated together or separately, creating a complex environment with an unprecedented amount of possible actions at any given state.

2.2 SPECIFICITIES OF STARCRAFT 2

Beginning to implement reinforcement learning algorithms on a particular game can be a excessively time-consuming ask if one has to prepare an adequate interface to interact with the game. Fortunately, the Starcraft franchise has historically been the RTS of reference when it comes to scripted bots or more complex AI agents. Since the launch of Starcraft Broodwar in 1998 and of Starcraft 2 in 2010, Blizzard Entertainment has made extra efforts to create an easy-to-use environment for AI researcher and players who wanted to create their own scripted bots, by providing various APIs on which the community could build upon. Detailed information about the specificities of RL in RTS and, more precisely, in Starcraft can be found in [3].

This allowed Deepmind to implement a Python environment designed to build and esaily test reinforcement learning agents on various subsections of the game, which we present in the next section.

3 PY-SC2 ENVIRONMENT

The Py-SC2 framework on python allows our RL models to easily interact with the Stacraft 2 game. The purpose of the framework is to get our model to learn from the same action space as the one available to a regular human player: our model can select units or click on the screen for example. It does not have access to information a human player would not have access to either during a regular game (the way a scripted bot would), and we can restrict the amount of actions the agent can take each second (to avoid the problem of "superhuman reflexes"). Remaining as close as possible to human-like behavior is really the philosophy behind the PySC2 framework - which is very important, as there are many ways to allow the AI to "cheat". Detailed information on the environment design can be found in [1], but we summarize in this section its most important elements.

3.1 MINI-GAMES AND REWARD STRUCTURE

Training a model on a full SC2 game is too far-fetched for state of the art RL, as the sheer complexity of available actions is too high: should the model build a bunker or a marine unit, which unit should it select and what for, should it attack now to "rush" its opponent or wait and build another base? The DeepMind team chose instead to design a selection of mini-games:

- **MoveToBeacon**: agent receives +1 when the marine reaches a beacon on the map. This is the "unit test" of the environment as the action space necessary to learn and act in this game can be greatly reduced.
- **CollectMineralShards**: agent starts with two marines that must collect minerals scattered across the map, as fast as possible. This mini-game tests the ability of a model to manipulate 2 units simultaneously.
- **FindAndDefeatZerglings**: agent starts with 3 marines, explores the map to defeat as many zerglings hidden in the fog of war as possible. This is useful to teach the agent how to use the camera and the mini-map, and how to deal with the partial availability of information. It can also chose to either split its 3 marines to find more enemies, but risking losing them in the process, or to keep them grouped.
- **DefeatRoaches**: agent starts with 9 marines, must defeat 4 roaches. Every time it defeats all of the roaches it gets 5 new marines as reinforcements and 4 new roaches spawn. The reward is +10 per roach killed and -1 per marine killed. This mini-game test what is called "unit micromanagement" in Starcraft 2 - or "micro" - namely the ability to use its troops efficiently to defeat a stronger force, by performing hit and run, focus fire, or concave engagement.
- **DefeatZerglingsAndBanelings**: this is the same as DefeatRoaches, except the opponent has Banelings and Zerglings (different types of units with different abilities), which give +5 reward when killed. The agent must learn to prioritize dangerous targets first (the banelings, which explodes on contact) and how to "split" its forces, a common tactic that is renowned to be very difficult to master, even by the best players.

- **CollectMineralsAndGas**: the agent must build new workers and extract the most resources from the map during a limited time period. This tests how the agent can optimize its resource collection rate by not overcrowding certain areas.

- **BuildMarines**: the agent is rewarded for building marines, which require a complex chain of elements to be executed in a certain order; workers must be created and used to extract resources, then to build Supply Depots, then Barracks in order for marines to be available. This mini-game corresponds more or less to the start of an actual Starcraft 2 game, and is by far the most complex mini-game available in Py-SC2.

3.2 OBSERVATION SPACE

The PY-SC2 environment provides the model with two types of features at each step:

- **Spatial features**: the map and mini-map from the game screen are decomposed into multiple visual layers that keep the same resolution as the original map/mini-map, but that contain different types of information (terrain height, visibility, selected units, zerg creep...). These layers are an abstraction of the RGB data, but keep the core spatial and graphical concepts of Starcraft 2 according to the DeepMind team.

- **Non-spatial features**: the amount of gas and minerals collected, population used, information about selected units and more (detailed information is available on the original paper [1].

3.3 ACTION SPACE

Actions in the PY-SC2 environment are supposed to mimic how a human would interact with the original game UI, like selecting units and moving them on the map by right-clicking on a point of the screen, or clicking on elements of the game before choosing a subsequent action available for that element. In order to keep the action space as simple as possible, several steps are taken:

- **Functional actions**: actions are represented by an identifier a_0, with arguments $a_1, ..., a_l$. For our implementation we kept it simple using only spatial arguments (the position of the mouse). As there are 13 possible types of arguments for 300 actions it would be computationally difficult to design a model with one output for each action-argument tuple. Here our model can have one output layer that corresponds to the action choice, and several output layers for each argument, which leads to models with much fewer weights.

- **Compound actions**: most actions in the game have shortcut, which are often used by human players to speed things up. For example, a human player can press 1 to select his main base, then Shift + E to queue up and build 5 workers. Compound actions in Py-SC2 are sequences of actions that require several atomic actions (like selecting a unit before moving it somewhere, and queuing that action with other *en-cours* actions) and are

accessed through one function only, in order to keep action-space complexity simple, especially since these atomic actions have to be called in the correct order.

Below are presented the different algorithms we implemented and studied for these mini-games. We identified them after having reviewed multiple research papers and github implementations, in order to focus on those that would yield decent results without requiring extensive hardware.

For the record, we trained our models on a GTX980M GPU, a i7-6280HK CPU and 8gb of RAM.

4 Deep-Q agent

Q-learning is based on the evaluation of the Q state action function. Deep-Q agent use neural network to approximate the Q function, and was introduced in [4] using ATARI games as a testing environment. Using a neural network to approximate the Q-function makes sense in our case for two reasons:

- it allows the model to extract visual features from the spatial inputs of the game
- the action/state space is highly dimensional, we cannot store an explicit representation of the Q-value.

DQN network does not have guaranteed convergence however implementation often use some tricks to improve the likelihood of convergence, among which is the experience replay. The Q-learning is generally based on the update :

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) \right)$$

where α is the learning rate, γ the discount factor and r_t the reward. To compute the optimal Q function, the agent will play some episodes using a Q function, then trying to estimate Q based on the set $(s_t, a_t, r_t, s_{t+1})_t$ he got. The experience replay consist on estimating the Q function only on a random sub-sample of $(s_t, a_t, r_t, s_{t+1})_t$ to reduce correlation between the observations. The Deep Q learning can be described as following :

Data: Neural Network model m
Result: Q function
$Q \leftarrow 0$;
while $N_{episodes} < Max_{episodes}$ **do**
  Generate trajectories and construct a memory $D = (s_t, a_t, r_t, s'_t = s_{t+1})_t$;
  Sample a minibatch $D_m ini$ at random from D;
  Compute the target output of the mini-batch : $y_i = r_i + \max_b Q(s'_i, b)$;
  $Q \leftarrow m.\text{fit}(Y, X)$
end

Algorithm 1: DQN

The agent also needs to use exploration i.e. play a random move at each step with a decreasing ϵ probability. We implemented a DQN on a simplified

MoveToBeacon problem where we give the agent access to prebuilt actions (such as "click directly on the beacon"). Even on that simple model convergence is quite slow and sometime the agent get stuck in a local optimum (such as doing nothing).

This approach, however, can be considered as cheating, since the agent has only access to 3 actions : "do nothing", "select the marine" and "click on the beacon" - which eludes the main difficulty of the mini-game: learning to click directly on the beacon and not somewhere else on the map.

5 Actor-critic agents

5.1 Actor-critic model

Q-learning and policy learning are not optimal in and of themselves for RTS games, especially Starcraft II. The Q-function becomes way too large when the action/state space is high-dimensional, which means the algorithm will have trouble learning the Q-values for every pair. Policy gradient methods like REINFORCE require too many samples in order to converge, as the Monte-Carlo estimate of the Value-function itself will depend on the sample trajectories, and those are extremely varied in a complex state/action environment like Starcraft II.

Both methods also have their advantages: Q-learning is sample efficient **when it converges**, while policy gradient is guaranteed to converge to a local maxima of the policy value function. **Actor-critic** methods take the best of both previous methods. The idea is to use an estimate of the Q-function $\hat{Q}$ to train the policy function estimator, using this gradient:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log(\pi_\theta(a|x))\hat{Q}(x,a)]$$

. For Starcraft II the natural estimator to use for both functions is a neural network, for the same reasons as in Q-learning. Generally, both function estimators use the same CNN in order to share visual and conceptual features (in turn these features are learned taking into account both the policy's and the value function losses). We did not implement this model as it still has too much variance, we instead used the A2C model described below.

5.2 Advantage Actor-Critic model (A2C)

Advantage Actor-Critic is a simplified version of the Asynchronous Advantage Actor-Critic model introduced in [5]. In this model, the "critic" $\hat{V}$ is a Value function approximator and a Q-function estimate used to calculate the Advantage $A^{\pi_\theta}(s,a) = \hat{Q}^{\pi_\theta}(s,a) - \hat{V}^{\pi_\theta}(s)$. This advantage function is used to train the policy estimator, using the gradient:

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}(Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t)$$

. This model has reduced variance compared to simple actor-critic models.

For our own implementation we used a simplified version of the one in [1], using $\hat{Q}(s_t, a_t) = \sum_{t' \geq t} \gamma^{t'-t} r_{t'}$. One of the main problems in adapting A2C to Starcraft II is the policy data format. One way of dealing with it is seperating Given the complexity of representing the policy as one vector (there are too many action-argument combinations), a design choice is to estimate the action function that will be chosen, and its arguments seperately. In o we cplementationhose to estimateur im only spatial arguments for design simplicity. The network architecture is as follows:

- **input**: screen layers, minimap layers
- **feature learning**, each input goes into a network with the following architecture
  - Conv layer with 16 filters, size (5,5) with ReLu activation, and Max-pooling size (2,2)
  - Conv layer with 32 filters, size (3,3) with ReLu activation, and Max-pooling size (2,2)
- **classification and regression**
  - Concatenate Screen and Minimap outputs
  - Dense layer of size 256 with ReLu activation
  - **Value function estimate**: Dense layer size 1 with linear activation
  - **Function identifier policy**: Dense layer with the size of the number of action functions available in the environment
  - **Spatial argument**: Dense layer of the same size as the screen resolution with Softmax activation

While [1] finds convergence on all mini-games with decent results, that was with a lot of computational power and time. Our simplified model did not converge even after several hours, but exposed the main challenges in the implementation of a A2C model on Starcraft II:

- The initial weight distribution in the output layer of the policy estimators has impact on the learning curve. If some actions are initialized with high probability values they will be taken by the agent, instead of trying other possibilities. Since the rewards are sparse the network does not have many occasions to learn correct behavior. This leads to a behavior we observed on *MoveToBeacon* game where the marine gets stuck at the corner of the map after one episode.
- Since the rewards are so delayed that the neural network has trouble linking specific actions to them.
- The agent doesn't understand the concept of "path", meaning it can choose to click on the screen to move a marine, but will not wait for the marine to actually reach that spot before choosing another action. Knowing to wait after trying some actions could be learned with temporal layers like LSTM (more on that later)

5.3 PPO

In order to reach convergence with the A2C faster, we took a look at the Proximal Policy Optimization algorithm, introduced in this OpenAI paper : [6]. The architecture of the PPO is mostly the same as the A2C one, but introduces a clipped surrogate objective function, as well as an associated surrogate loos function, in order to try and reduce how large the policy updates are. The ultimate goal is to reduce variability, and, ultimately, to accelerate greatly the convergence of the A2C model.

The surrogate objective function leverages the old policy prediction, on the actual state:

$$L^{CPI}(\theta) = E_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t]$$

.

We started working on the PPO algorithm after having found some very good feedbacks about it and its application to the Py-SC2 minigame, especially on this github repository : https://github.com/inoryy/reaver - which mentions an improvement of about 40x in convergence time on some runs. It is worth noting that this repository also provides an excellent overview of the performances of the different algorithms implemented on the Py-SC2 mini-games so far.

Incorporating the PPO into our A2C required us to tweak the input of our Keras network, to take into account the previous policy and the previous advantages. However, we didn't had the time to test it on a reasonable amount of episodes to check if the convergence was indeed faster.

6 Relational agents

6.1 Challenge overview

In order for Reinforcement Learning algorithms to be well defined, many assumptions are simplifying the environment and reducing the complexity of its representation. The independence of the variables and of the dataset samples enables the models to learn in isolation, chucking the data and the different objects of the representation in separated pieces to learn with. Reinforcement Learning faces many challenges in consequence:

- **Generalization**: traditional RL models do not represent the dependencies between the entities of the representation, which makes the identification of abstract classes of entities and relations difficult. Thus, the models tend to overfit the data they are trained with and fail to conceptualize the problem they are facing.
- **Transfer**: Without a proper level of abstract understanding of the problem, RL agents learning on one task fail to match their learned reasoning to different tasks from the same problem.
- **Run-Time**: The convergence of the algorithms is usually slow as the agent needs to conduct a very heavy search and exploration phase to palliate to the loss of information about the entities.

The use of recurrent neural network architectures such as RNN and LSTM have offered a partial solution to the problem: the model can be equipped with a memory that can indirectly recover parts of the relation between states and actions when close by in time but lacks of ax explicit formulation of the dependencies.

Recent advances in NLP, where the meaning of a sentence does not only depends on the meaning of each word but to their relationship have offered new interesting models explicating the dependencies between objects of a space. The **Transformer** [7], in particular, is a new model that can explicitly map the different dependencies by using attention mechanism (weight distribution on the different inputs) without the sequencing issue of recurrent models. Recent advancements in Reinforcement Learning[8] have shown that its attention mechanism, the multi-head dot product attention, can be used as a relational module to enable a solid representation of the dependencies between the objects and the different actions.

6.2 Relational Module

The relational module (Multi-head Dot-Product Attention or MHDPA) is structured as the following:

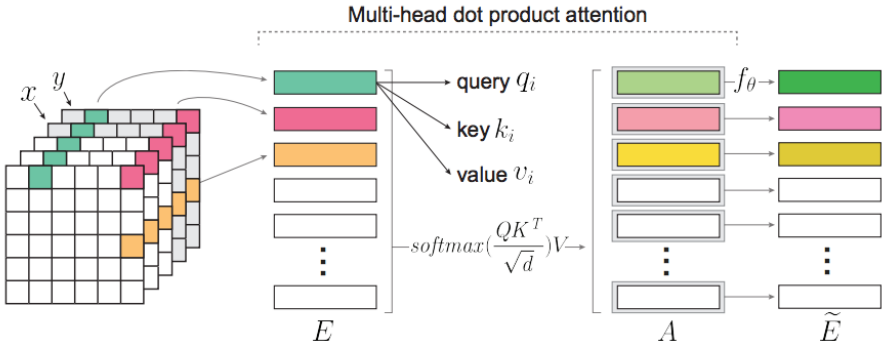

Figure 1: Relational module composed of a multi-head dot product attention

In input is passed the flattened result of the spatial processing of the state (composed of residual convolutional blocks and a 2DLSTM). The result is divided to a certain number N of vector, representing local regions of the image. Those local regions will be considered as the entities to draw the relation between. The bet is that the algorithm will identify by itself the entities within this local regions, so that the model can generalize its learned objects and use this knowledge in other problems.

Each entity ($e_{1:N}$) has an assigned query, key and value vector representation q_i k_i v_i randomized initially, and normalized at each training to have 0 mean and unit variance. For each vector, the dot product of its query with all the keys is computed (hence the explicit dependency of each entity with the others), divided by $\sqrt{d_{k_i}}$ and passed through a softmax function. We end up with the

the weight on its value. Each value is then computed with its weight to get the attention vector. The complete process can be computed in one time through the following equation:

$$A = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

We repeat the process h times and the a_i^h vectors are concatenated together and passed to a 2-layer residual feed forward network and then normalized to produce the vector output.

6.3 Network Architecture

In the first introduction of the multi-head dot product attention as a relational module for RL, DeepMind uses the architecture below and as described in [8]. For our own implementation, the architecture has been a challenge to implement as the previous states need to be recorded and sent to the LSTM module. In addition, the multi-head dot-product attention is only available on TensorFlow and not Keras. Thus, the Relational agent needed a complete readaptation of our codebase. We have decided to prioritize the implementation and experimentation on traditional agents.

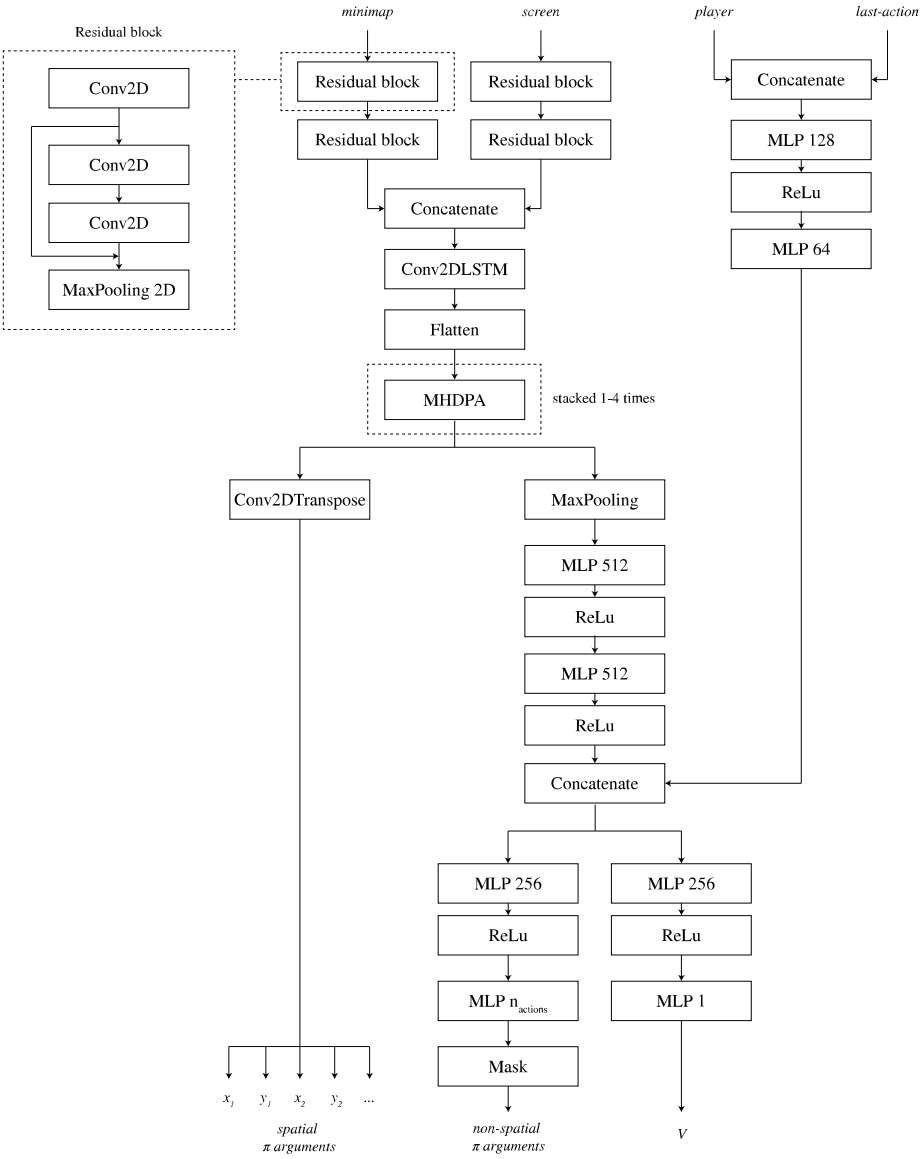

Figure 2: Relational Reinforcement Learning network architecture for Starcraft II

References

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

[2] F. Chollet *et al.*, "Keras." https://keras.io, 2015.

[3] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game ai research and competition in starcraft," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, pp. 293–311, 2013.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.

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

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[8] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals, and P. Battaglia, "Relational deep reinforcement learning," 2018.