Exploration of Model-based Methods on StarCraft II

Anonymous

Abstract

StarCraft II provides an extremely challenging platform for reinforcement learning. In this paper, we explore model-based methods on StarCraft II for the first time. By experimenting with three different model building methods, we investigate a simple but effective model-based method.

1 Introduction

In recent years, reinforcement learning [Sutton and Barto, 1998; Kaelbling *et al.*, 1996] has received more and more attention from the research communities. Unlike traditional supervised learning, reinforcement learning is applied to the environment. The agent collects the data itself in the environment instead of using the preprocessed data. This training mode is more like the process of human learning from scratch. As a result, the artificial intelligence community has begun to pay more attention to the reinforcement learning. The combination of reinforcement learning and deep learning enabled AI to achieve much better results in Atari games [Mnih *et al.*, 2015; Hado *et al.*, 2016; Ziyu *et al.*, 2015] and Go [Silver *et al.*, 2016; David *et al.*, 2017], demonstrating the effectiveness of reinforcement learning. After the challenge of Go, many large companies and research institutes have moved the application environment of reinforcement learning to more complex games, such as real-time strategy (RTS) games. StarCraft II [Vinyals *et al.*, 2017] has become the platform for many companies to challenge.

Applying reinforcement learning to StarCraft II is facing many difficulties and challenges. For example, StarCraft II has a state space that is much larger than Go. Players need to control hundreds of units in the game and the game has a very large action space. StarCraft II is a game with imperfect information. Players know almost nothing about the state of their opponents, so that they need to scout then speculate on the opponents' potential behaviors. Moreover, a full-time StarCraft II game contains thousands of steps, which makes the problem of credit assignment much worser. Because of these difficulties and challenges, the StarCraft II environment can be regarded as an environment closer to the real world. If reinforcement learning makes breakthroughs in this environment, it will be of great benefit to the application of reinforcement learning in the real world.

The previous work, whether it was a hierarchical or modular architecture agent or a highly resource-intensive agent using several new technologies, used a model-free reinforcement learning method. The advantage of the model-free reinforcement learning methods is that the agent does not need to know the state transition model in the environment while learning, but the disadvantage is that it requires huge number of samples. Therefore, when using the model-free reinforcement learning method, it requires a lot of computing resources and training time. In contrast, the model-based reinforcement learning method first obtains the model of the environment in some way, and then learns based on this model. Since the model is obtained, we can use several planning methods to enhance the exploration, or use the samples generated by the model to reduce the number of samples from the environment, thus accelerating training phase. Different from all previous work on StarCraft II, this paper explores the application of model-based reinforcement learning method in the StarCraft II environment for the first time.

Model-based reinforcement learning is explored in the following ways. First, we built a subproblem in a StarCraft II environment. The environment for this subproblem is based on the StarCraft II platform, but only focuses on one of the most fundamental aspects, namely, how to maximize the efficiency of resource collection. We evaluated the performance of using regression-based method to learn the model through this sub-environment. We found that although the prediction error of the model is small, the cumulative error will be very large due to the continuity of the model prediction, which brings difficulties to subsequent learning. In response to this problem, we implemented state transition rules to predict some states, and using machine learning methods to predict some other features.

Based on the ideas validated in the sub-environment, we explored using this method in the original StarCraft II environment. Unlike the sub-environment, it is difficult to implement the complete state transition model of StarCraft II. Therefore, we have taken advantage of a simpler idea: instead of building the original StarCraft II environment, we build a mind-game environment of StarCraft II which called 'strategy-SC2'.This method is simple but reasonable. While manipulating the units in the game, humans will also con-

struct a simple model in their minds to judge the result of the game or speculate on the outcome of a battle. This model is not part of a complete game, but a simplified simulation of the game. Since there is a simple mini environment and all state transitions in this environment are known, we can do various learning based on the mini environment. It will bring a number benefits: First, we can do planning in the mini environment, so we can effectively consider the situation after several steps; Second, the learning speed will be very fast because the mini environment is relatively simple; Third, since we control the mini environment, the curriculum we design can be smooth and easy to learn; Last but not the least, it is easy to migrate from the mini environment to the original environment because of the similarity between the mini environment and the original environment.

Additionally, humans do not learn to play StarCraft II from scratch. By building a mini environment, historical strategic information can be integrated into this environment, such as the advantage of attacking low places from high places, the importance of logistics of war and so on, thus simulating the way humans learn. Since there is a simple mini environment and all state transitions in this environment are known, we can do various learning based on the mini environment. It will bring a number benefits: First, we can do planning in the mini environment, so we can effectively consider the situation after several steps; Second, the learning speed will be very fast because the mini environment is relatively simple; Third, since we control the mini environment, the curriculum we design can be smooth and easy to learn; Last but not the least, it is easy to migrate from the mini environment to the original environment because of the similarity between the mini environment and the original environment.

After building the mini environment, we first train an agent in the mini environment. We use an ACRL (Adaptive Curriculum Reinforcement Learning) learning algorithm to efficiently train an effective agent. It is not necessary to manually design reward due to the smoothness of ACRL for difficulty control, and we can just use outcome as reward to learn an effective policy. Then we migrated the agents trained in the mini environment to the original StarCraft II environment.

There are many ways to migrate agents. For example, the agent in the mini environment can be used as a teacher and the agents in the source environment can be trained by imitation learning. We use a simple transfer learning method here, which is to adjust the state as in the source environment so that the state feature is the same as in the mini environment and keep the action space in the source environment consistent with that in the mini environment at the same time (this means that macro-actions may need to be used as action space in the source environment). We can then use the agent policy in the mini environment as the initial value of that in the source environment, and then fine-tune it.

2 Methods

In this section, we will introduce the Model-Based Reinforcement Learning (MB-RL) methods we explored. First, we discuss four possible model-based learning methods. After that, we analyze each of the three possible methods. For the last one, which is we apply to the StarCraft II game environment, we introduce the three steps of its implementation: Firstly, we design the strategy model which employs human knowledge; Secondly, we train a policy on the strategy model; Finally, we map the strategy policy to the source domain.

2.1 Four cases of MB-RL

Before going into the details, we divide model-based reinforcement learning into the following four categories:

Model is known: The state transition function of environment is known. For example, in Go games, when the current position is known, the next position is clear if certain move is taken. Therefore, a model-based method such as MCTS can be applied to Go. However, for many games other than Go, the model is unknown. These includes some RTS games that are currently focused on the research communities, such as StarCraft: Brood War, Dota 2, Honor of Kings and StarCraft II. If we want to apply model-based reinforcement learning to these games, the common way is to learn a model.

The learning phase is not required because of the model is known. MCTS can enhance the exploration of reinforcement learning, leading to better further strategies. The effectiveness of combining reinforcement learning with MCTS has been successfully verified in Go.

Model is learned: The most common idea of model-based reinforcement learning is to first learn a model. The state transition function predicts s_{t+1} when given the input (s_t, a_t), i.e.

$$s_{t+1} = f_\rho(s_t, a_t) \tag{1}$$

The states s_t and s_{t+1} are continuous in most settings, which means that it is a regression problem. Since the state contains many dimensions, it is a multivariate regression problem. We can use normal supervised learning methods to solve them. There are quite a few tricks to learn a state transition model. This problem is more difficult than the usual regression problem because of the continues state transitions, which will be described in detail later.

Model is implemented: Simplicity and generalization are the benefits of learning a model. However, the problem is that the errors in learning continue to increase with the continuous decision process, which leads to cumulative error problems. Cumulative errors make the difficulty of learning an effective model increase dramatically. In addition, the data collection process required to learn a model is also a problem. The collection of data requires agents to explore in the environment, and exploration can be time consuming. The time it takes to collect data is even more than the time saved by learning the model.

The opposite of learning a model is to implement a model. Some actions in the state transition can cause a fixed change. If these changes are known, then we can implement them as a set of rules and then put them into the model. For stochastic parts of the state transition, such as changes in state features that are difficult to predict, can be supplemented by learning. The model implemented by the rule and learning method can effectively reduce the error caused by learning the complete model and save the required data. The formula is as follows:

$$s^d_{t+1} = f_\omega(s_t, a_t) \tag{2}$$

$$s^r_{t+1} = f_\rho(s_t, a_t) \tag{3}$$

In the next state s_{t+1} predicted, some features s^d_{t+1} are given by rules while others s^r_{t+1} by learning algorithms. We will give such a practical case in the prototype experiment later.

There are more deterministic state features in some environments, so there are more rules. In other environment, there are more stochastic state features, so there is more learning. If there are more deterministic parts in some environment and the rules of state transition are simple, the way to implement a model will increase the accuracy of prediction and the speed of training.

Model is abstracted: Implementing a model yields a accurate model, but the complete implementation of a model is too expensive for large and complex games. For example, StarCraft II and Dota 2 are developed by a team of hundreds of people over a few years. Although there are certain rules that can be implemented, these rules are too many and complex. In terms of the action of the game unit, the difficulty of predicting using a pure regression model (e.g. predicting the next second image from the current one) is very large due to the physical collision involved in the image. Therefore, we present our solution in this paper, which is to implement a reduced abstract model. This is a simplified version of the original environment, including the core part of the original environment, but removing the details of the part. An abstract model can be mapped to the original environment through a mapping function. That is

$$z_t = f_{m_s}(s_t) \tag{4}$$

$$a_t = f_{m_a}(\pi_m(z_t)) \tag{5}$$

Although in some historically long games, such as Atari, deep neural networks have been able to predict images for the next frame. However, those games were born about 40 years ago, with few pixels and simple rules, so the predictions are not as difficult. In modern high-resolution, high-complexity games, the computational and training resources required to predict the next frame of image are much larger than before. It is not realistic to completely implement a model of a modern complex game.

There are two immediate benefits to the model implemented in this way. The first is that it is simple, because there is no need to implement a complete model, so the cost of obtaining the model is greatly reduced. The second benefit is that although it is only a simplified original model, the planning and training on this model can be fed back to the original model because it retains the most core part. We will show how to preserve the core part of the model in a later experiment. There are several ways to feed back into the original model, and we will give the simplest but effective way in this work.

After obtaining the model, there are several ways to perform model-based reinforcement learning. Firstly, we can plan directly with the model. Planning is essentially a search method with a huge amount of computation, but the advantage is that it can be run directly, without the learning process. For example, we can do MCTS based on the model. The benefit of MCTS is that it can greatly enhance the depth of thinking. This will lead to longer-term decision. Secondly, we can still use the previous model-free reinforcement learning method, but use the model to generate some simulated samples, which reduces the amount of sample that needs to be collected in the real environment. Thirdly, we can also learn to get an initial policy π in the model and fine-tune the policy π in the original environment.

2.2 Learning a Model

Suppose we have such a problem and we need to predict $f(s, a) \rightarrow s'$. In order to fit such a function, we first need to collect data. We let the agent run in the environment through a random policy to collect data. The form of these data is as follows (s, a, s'). Then we use neural network to construct a regression model, the input is (s, a), and the output is s'. Here the dimensions of s and s' are the same, and the feature of each dimension is continuous. Action a is the action we defined, and here we are using discrete action spaces. When s and a are concatenated into a vector, we first use one-hot encoding to turn a into a vector of $|A|$ dimensions. Our neural network structure is a fully connected network with two hidden layers. We use MSE (Mean squared error) as the loss function which is defined as follow:

$$loss = \frac{1}{M}\frac{1}{L}\sum_{i=1}^{M}\sum_{j=1}^{L}(s_{i,j} - s'_{i,j})^2, s'_i = f_\theta(s_i, a_i) \tag{6}$$

$$loss = loss + \beta * \|\theta\|^2 \tag{7}$$

where β is a hyper-parameter to control over-fitting.

After collecting enough data, we used the Adam optimizer for back-propagation. Referring to the method in [Nagabandi *et al.*, 2018], we use several tricks to improve the model learning: 1. Predicting the difference of states. Like [Nagabandi *et al.*, 2018], we tried to predict $(s - s')$. Due to predicting the difference of states, the range of the label space becomes smaller, and the number of samples predicting the same label increases, meaning that the same category of data is increased, which can alleviate the problem of sample imbalance. we found that the prediction difference will bring better experimental results. 2. Standardize the input space. Since the scale of each dimension may be different, we first standardize the input feature as follows:

$$x_i = \frac{(x_i - mean(x_i))}{std(x_i)} \tag{8}$$

3. Change the way of evaluation. We found that using MSE to evaluate the model is of little significance due to the strong fitting ability of the neural network. At the end of the training, both the MSE on the training set and validation set are very small. For model learning, one of the most important attributes is H-step-error, which is defined as follow.

$$loss = \frac{1}{H}\sum_{z=1}^{H}\|s'_{t+z} - s_{t+z}\|^2 \tag{9}$$

$$s'_{t+z} = f_\theta(s'_{t+z-1}, a_{t+z-1}) \tag{10}$$

which means how much the final error of the states is when predicting the next state with the predicted state. This characterizes the magnitude of the cumulative error.

Algorithm 1 Step-Dyna

Input: max episodes for update M, max iteration steps Z, max game steps T
Parameter: initial simulated steps H, decrease count S
Output: π_θ

Initialize policy π_θ. Let $h \leftarrow H$.
for $z = 1$ to Z **do**
  Let $h \leftarrow H - z/S$.
  if $h < 0$ **then**
    Let $h \leftarrow 0$.
  end if
  Clear global buffer D_g.
  for $m = 1$ to M **do**
    Clear local buffer D_c.
    for $t = 1$ to T **do**
      Collect experience by using π_θ .
      $D_c \leftarrow D_c \cup \{(s_t^m, a_t^m, R_c(s_t^m, a_t^m), s_{t+1}^m\}$
      Clear simulated buffer D_s.
      for $j = 1$ to h **do**
        Simulate experience by using π_θ and f_ϕ.
        $D_s \leftarrow D_s \cup \{(s_j^t, a_j^t, R_s(s_j^t, a_j^t), s_{j+1}^t\}$
      end for
      $D_g \leftarrow D_g \cup D_s$
    end for
    $D_g \leftarrow D_g \cup D_c$
  end for
  Use D_g to update θ to maximize expected return of M episodes.
end for

After learning a model, we use this model for reinforcement learning. Here we use the algorithm Dyna []. When the agent samples in the environment, each time the data (s, a, s', r) is acquired, the following state is predicted by the model $f_\rho(s, a) \rightarrow s'$, and then the policy $f_\theta(s) \rightarrow a$ is used to select the action. This simulation process continues h steps. The agent then proceeds to the next step, so that when the agent finishes running in the environment, the sample size originally acquired is n, and the sample size simulated by Dyna is $h * n$. The total sample size is expanded to $(h + 1) * n$. The reinforcement learning algorithm we use is PPO []. During the update process of the PPO algorithm, we update with the samples of size of $(h + 1) * n$.

Such an update strategy poses a problem. The inaccuracy of the model leads to a rapid learning of PPO at the beginning, but the final effect will be degraded. To solve this problem, we propose step-Dyna algorithm Algorithm 1. As the number of iterations increases, the sample step h of the Dyna is reduced. When a certain stage is reached, h is reduced to 0, meaning that Dyna is not used to increase the sample size. Using this algorithm, learning often learns better in the first sever steps, and the subsequent learning can still rise faster. The results are shown in the experiment section.

2.3 Implementing a Model

The H-step-error of the trained model is still high. This is mainly due to the following problems: First, The systematic error of the model, because the accurate model also has an error, which is unavoidable; Second, the huge difference in state caused by some minor errors. For example, for a given environment, the number of mines 45 and 48 may not cause too much difference, but Crystal Tower data 7 and 8 will have a large impact. Although these errors are small, for states, the distance between the correct and error states can be very large, corresponding to two completely different strategies.

We use "implementing a model" to alleviate this problem. Implementing a model means dividing the state that needs to be predicted into two parts—deterministic state feature s_d and random state feature s_r that are not easily inference. For s_d, several rules can be used to predict changes in these values. For example, when performing the action of producing farmers, if the conditions for farmers' production are met, the number of farmers is increased by one. An action a may affect several s_d, while an s_d may also be affected by several actions. Whether or not an action is executed successfully can also be judged by rules. Some actions can only bring about a change in state if they are successfully executed. Due to the certainty of the rules, the state changes in this way are very accurate. Using rules to control state changes can result in a significant drop in H-step-error for unseen states.

For s_r, we still use the previous learning method to predict it. However, since we predict $s_{d_1}, s_{d_2}, ...$, the acquisition of these values is not affected by the learning model and can be given before the learning model, so the feature value of the rule prediction s_d can be obtained first, then we get a model prediction

$$f(s, a, s_d) \rightarrow s_r \tag{11}$$

Using the method of implementing a model, the hyperparameter limits for step-Dyna can be more relaxed because the prediction model is accurate even in unseen states. Using this implemented model, we found that step-Dyna achieved faster convergence and better final performance than the original PPO in the setting of few samples, which are shown in experiment section.

2.4 Mapping a Model

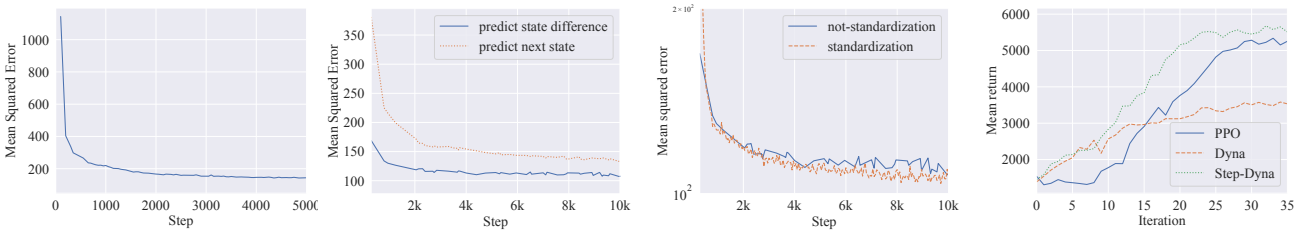
The method of implementing a model and the step-Dyna based on it has achieved better results than the PPO in the prototype environment. But in the face of a larger and more complex environment, the way to implement a complete model becomes costly. Here, we introduce one of our ideas, which draws on the way human players think in battles in the game—building an abstract simplified model of the game, planning and learning in the model, and mapping the results to the original game.

In modern games, due to the complexity of their implementation, very precise models are used on the screen and physics, which makes it very difficult to implement the underlying models of these environments directly by rules.

How to mapping a model will be explored in future.

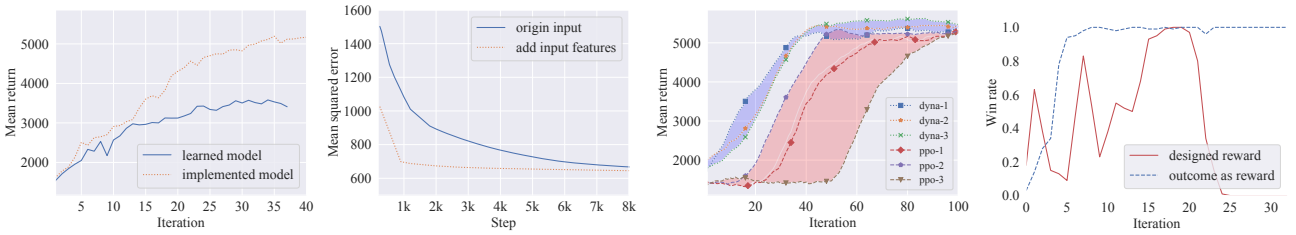
3 Experiment

In this section, we first show the effects and problems of learning the model in the prototype environment, and then give the improvements brought by the implementation model.



(a) Learning models (b) Predict difference (c) Standardization (d) Step-Dyna

Figure 1: Experiments in learning a model.



(a) Implemented models (b) Expand input feature (c) Dyna vs PPO (d) Effect of reward

Figure 2: Experiments in implementing a model.

3.1 Effects of Learning a Model

In this section we show the effect of learning a model in a prototype environment.

We first introduce the prototype environment. This prototype is one of the tasks in StarCraft II—the development of the economy. The goal of this task is to maximize the number of minerals owned by the agent at a fixed time point T. In this prototype environment, the agent has only three actions, including building farmers, building pylons, and doing nothing. In detail, the construction of farmers will increase the amount of crystal mine growth per unit time but will consume 50 crystal mines and 1 food supplies. Building pylon will increase food supplies by 8, but it will also consume 100 crystal mines. The maximum number of farmers collected per unit time in each mining area is limited. When this limit is exceeded, increasing farmers will not increase the growth of crystal mines per unit time. Therefore, the agent needs to properly control the three actions. It is necessary to build farmers to maintain production at the beginning. When the population is insufficient, pylon needs to be built. When the collection resources are saturated, it is necessary to stop building farmers and pylon to save crystal mines. Only three conditions to coordinate reasonable resources will be maximized.

Experiment 1: Can the prediction function $f(s, a) = s'$ be learned using supervised learning, neural networks, and Adam back-propagation algorithms, so that the validation error is reducing?

Some hyper-parameters are set as follows. Neural network architecture is a MLP, with two hidden layers and 256 ans 128 units for each hidden layer. Training batch-size is set to 10000. Adam's initial learning rate is set to 10^{-4}

As in Fig.1 (a), we can see that MSE is reduced through training, which means our method is effective.

Experiment 2: Can the prediction difference make the learning effect better? As in Fig.1 (b), prediction of state difference can make the learning faster.

Experiment 3: What is the impact of standardizing input features? As in Fig.1 (c), standardizing input features has some effects, but the effect is not very obvious..

Experiment 4: What are the different effects between Dyna, PPO and step-Dyna?

The settings of the hyper-parameters in the game are as follows. The maximum duration of a game is 10000 steps. The settings of the hyper-parameters in the PPO are as follows. The epoch-num is set to 1. The setting of the parameter in step-Dyna is as follows. H is set to 2 and S is set to 5.

As we can see in Fig.1 (d), Dyna is better than PPO at first, but worse in final performance. The step-Dyna exceeded the PPO at the beginning and convergence, indicating the effectiveness of the model learning.

3.2 Effects of Implementing a Model

Although using step-Dyna can lead to better learning effects than the original PPO, the hyperparameter step requires careful setting and the learning curve is not stable. The reason for these problems is that although the DM model is more accurate after predicting one or two steps, the accuracy is greatly reduced after several states after continuous prediction. At the same time, at the end of the study, the inaccurate model predicts that the resulting sample will interfere with the original PPO learning. Therefore, a method of implementing a model can be employed.

In the implementation model, the total number of rules we implement is 12, and the state that needs to be learned is *diff-mineral*.

Experiment 1: Can implementing a model bring stability to Dyna? Fig.2 (a) shows the Dyna using rules to predict and

the previous Dyna. It can be seen that the realization of a model has a great improvement on the effect of Dyna.

Experiment 2: What is the effect of using ruled feature as input? Through Fig.2 (b), we can find that when the feature predicted by rule is added to the input and then predicted, it can speed up the convergence of learning.

Experiment 3: Comparison of step-Dyna and PPO. Fig.2 (c) shows learning curves of step-Dyna and PPO. All algorithm is run for three times. The update-num of the PPO is set to 20 in this setting. The setting of the parameter in step-Dyna is as follows. H is set to 3 and S is set to 5.

It can be seen that step-Dyna can be significantly better than PPO when the number of games used for each update is small, which confirms the characteristics of sample-efficient of model-based RL.

4 Conclusion

This report explores methods of model-based reinforcement learning in a complex environment StarCraft II.

References

[David *et al.*, 2017] Silver David, Schrittwieser Julian, Simonyan Karen, Antonoglou Ioannis, Huang Aja, Guez Arthur, Hubert Thomas, Baker Lucas, Lai Matthew, Bolton Adrian, et al. Mastering the Game of Go without Human Knowledge. *Nature*, 550(7676):354, 2017.

[Hado *et al.*, 2016] Van Hasselt Hado, Guez Arthur, and Silver David. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of 30th AAAI Conference on Artificial Intelligence*, 2016.

[Kaelbling *et al.*, 1996] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level Control through Deep Reinforcement Learning. *Nature*, 518(7540):529–533, 2015.

[Nagabandi *et al.*, 2018] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. In *Proceedings of IEEE Transactions on International Conference on Robotics and Automation (ICRA)*, pages 7559–7566, Brisbane, Australia, 2018.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*, volume 135. MIT Press Cambridge, 1998.

[Vinyals *et al.*, 2017] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[Ziyu *et al.*, 2015] Wang Ziyu, Schaul Tom, Hessel Matteo, Van Hasselt Hado, Lanctot Marc, and De Freitas Nando. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv:1511.06581*, 2015.