Proceedings of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 661–666
DOI: 10.15439/2024F1840
ISSN 2300-5963 ACSIS, Vol. 39

An environment model in multi-agent reinforcement learning with decentralized training

Rafał Niedziółka-Domański
0009-0004-0405-5508
Maria Curie-Sklodowska University in Lublin
5 M. Curie-Skłodowskiej Square, 20-031 Lublin, Poland
Email: rniedziolkad@gmail.com

Jarosław Bylina
0000-0002-0319-2525
Maria Curie-Sklodowska University in Lublin
5 M. Curie-Skłodowskiej Square, 20-031 Lublin, Poland
Email: jaroslaw.bylina@umcs.pl

***Abstract*—In multi-agent reinforcement learning scenarios, independent learning, where agents learn independently based on their observations, is often preferred for its scalability and simplicity compared to centralized training. However, it faces significant challenges due to the non-stationary nature of the environment from each agent's perspective.**

We investigate if incorporating an environment model in multi-agent reinforcement learning with decentralized training can alleviate the non-stationarity effects caused by the adaptive behaviors of other agents. To do this, we design and implement a custom model-based algorithm and compare its performance with the well-known model-free algorithm (Deep Q-Network). Our algorithm uses an environment model to plan and select actions. However, we do not require the model to be perfect for action selection, allowing it to be learned and improved during training. Our results suggest that integrating environment models into MARL offers a viable solution to mitigate non-stationarity.

Keywords: reinforcement learning, multi-agent reinforcement learning, model-based RL.

I. Introduction

IN MULTI-AGENT reinforcement learning (MARL), agents learn to make decisions in an environment where other agents are also learning and adapting. A major challenge in MARL is non-stationarity, where the perceived environment's dynamics change from the perspective of each agent due to the adaptive behaviors of other agents. This non-stationarity can significantly hinder the learning process, making it difficult for agents to converge on optimal strategies.

Decentralized learning, where agents learn independently based on their observations, is often preferred for its scalability and simplicity. However, it faces significant challenges due to the non-stationary nature of the environment from each agent's perspective. This article explores whether using an environment model in MARL with independent learning can mitigate the non-stationarity problem caused by other adaptive agents.

To investigate this, we design and implement a custom model-based method for MARL. We then compare the performance of our model-based algorithm against a traditional model-free algorithm to assess its effectiveness in reducing non-stationarity.

Our experiments and results provide insights into the potential benefits of integrating environment models in decentralized MARL, offering an alternative perspective on addressing the non-stationarity challenge.

II. Background

A. Problem Definition

We consider a sequential decision process with multiple agents. We can define it as an n-agent stochastic game [1], [2] consisting of:

- set of agents $\mathcal{N} = \{1, \ldots, n\}$,
- state space $\mathcal{S}$,
- action space $\mathcal{A}^i$ for each agent $i \in \mathcal{N}$,
- reward function $\mathcal{R}^i$ for each agent $i \in \mathcal{N}$, defined as $\mathcal{R}_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$, where $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^n$,
- state transition probability function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$.

At each time step t, agents choose their actions a_i^t according to individual policies $\pi_i : \mathcal{S} \mapsto \Delta(\mathcal{A}^i)$ for state s^t. Based on the joint action $a^t = \{a_i^t\}_{i \in \mathcal{N}}$, state transits to s^{t+1} and each agent receives reward r_i^t. In independent learning, goal of each agent i is to find optimal policy π_i^* that maximizes expected return defined as

$$\mathbb{E}_{\pi_i, \pi_{-i}} \left[\sum_{t=0}^{\infty} \gamma^t r_i^t \right],$$

where $\pi_{-i} = \prod_{j \in \mathcal{N} \setminus \{i\}} \pi_j$ is joint (Cartesian product) policy of other agents, and $\gamma \in [0, 1)$ is a discount factor. This means, that optimal policy of agent i also depends on how the other agents act. Given this goal, the solution to a stochastic game will be (ε-)Nash equilibrium, where no agent can improve its performance by unilaterally changing its policy.

B. Independent Learning

In independent learning, each agent learns its policy π_i based solely on its own observations, actions and rewards, while completely ignoring the existence of other agents [1]. This approach essentially reduces a multi-agent problem to a series of single-agent problems, for which we can use single-agent reinforcement learning algorithms.

From the perspective of agent i, the policies π_j of other agents become part of the state transition function:

$$\mathcal{T}_i(s^{t+1} | s^t, a_i^t) \propto \sum_{a_{-i} \in \mathcal{A}^{-i}} \mathcal{T}(s^{t+1} | s^t, a_i^t, a_{-i}) \prod_{j \neq i} \pi_j(a_j | s^t),$$

where $-i$ means "all agents other than i." During learning, each agent j will continue to update its policy π_j, and the action probabilities associated with π_j in each state s may change. Therefore, from the perspective of agent i, the transition function $\mathcal{T}_i$ seems to be non-stationary. However, this perceived non-stationarity is due solely to the evolving policies π_j of the other agents. Consequently, independent learning approaches can lead to unstable learning and might not converge to a stable solution in the game.

We can alleviate this problem by allowing some centralized information. For example, during training, the algorithm can utilize the shared local information from all agents to update their policies. One of the most noticeable algorithms of *centralized training and decentralized execution* (CTDE) paradigm is actor-critic method known as MADDPG [3], which uses centralized action-value function as a critic. Access to global information helps coordinate the actions of agents, which can mitigate the effect of non-stationarity and contribute to more stable learning. However, centralization introduces issues with scalability, as the number of joint actions grows exponentially relative to the number of agents.

III. Integration of Environment Model

A model of the environment refers to anything that an agent can utilize to predict the environment's response to its actions [4]. Given a state and an action, the model provides a forecast of the resulting next state and reward. In the case of a stochastic model, there are many possible future states and rewards, each with probabilities assigned to their occurrence. These models can take two forms:

- *distribution model*, which provides the complete distribution of all possibilities and their probabilities,
- *sample model*, which generates a possible state and reward by sampling according to the assigned probabilities.

In both cases, model is used to simulate the environment and generate simulated experiences.

The term *planning* can have various meanings across different fields. In our context, planning refers to any computational process that uses a model as input to generate or refine a policy for interacting with the modeled environment. Planning can be integrated into reinforcement learning in two ways: 1) we can use the model to generate simulated experiences and use these experiences to improve the policy or value function (*background planning*); and 2) we can use the model to plan in predicted states in order to select actions (*decision-time planning*) [4].

Based on how learning is used, we can distinguish three main categories of planning-learning integration [5]:

- *model-based RL with a learned model*, where we learn both model and policy/value function (e.g. Dyna-Q [6]),
- *model-based RL with a known model*, where we plan over a known model, and only use learning for value function (e.g. AlphaZero [7]),
- *planning over a learned model*, where we learn model, and use it to plan locally, without learning policy or value function (e.g. Embed2Control [8]).

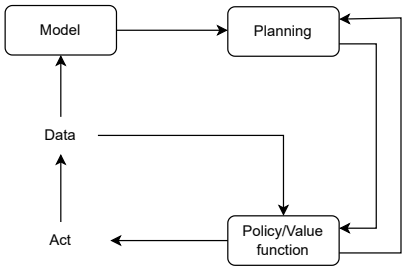

Fig. 1. Integration of planning and learning in Dyna architecture

Here, "planning over a learned model" may not be considered model-based RL, since no policy or value function is learned.

A. Dyna Architecture

The Dyna [9] architecture is an integrated framework that combines learning, planning, and reactive execution in the context of reinforcement learning (Fig. 1). Real experience is utilized by the planning agent to improve the model, making it more accurately reflect the real environment, as well as being used, similarly to model-free methods, for directly improving the value function and policy. The model is used to generate simulated experiences, which are then utilized to update the policy/function. Planning is done incrementally and can utilize world models often generated by learning processes, even if they are sometimes incorrect. If the model is accurate, planning significantly speeds up finding the optimal policy. In small tasks, it has been demonstrated to be true, even if the model also needs to be learned or if the environment changes [6].

Recent progress in Dyna-style MARL methods includes *Adaptive Opponent-wise Rollout Policy Optimization* (AORPO) [10]. AORPO explores ways to improve sample efficiency in stochastic games, where agents independently learn their policies but have the capacity to communicate with each other. It utilizes opponent models to generate model rollouts for a specific number of steps, determined by the validation error of the opponent model. Subsequent steps of the rollout involve requesting actions from the corresponding opponent through communication. Improving policies with data from environment models that predict joint agent actions helps with non-stationarity, but opponent models need full observability, which may not always be possible.

B. Heuristic Search

Classical decison-time planning methods are collectively known as *heuristic search* [4]. In heuristic search, for each encountered state, a large tree of possible future steps (based on the model) is considered. An approximate value function (typically designed by humans and never updated as a result of the search) is applied to the nodes at the ends of branches, and then these values are backed-up towards the current state, which is the root of the tree. When we eventually compute

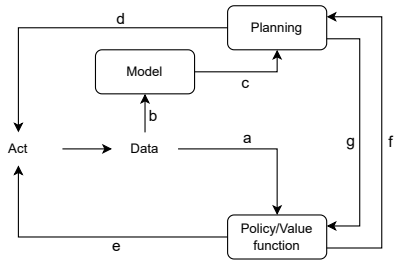

Fig. 2. Algorithmic connections between planning and learning

values of action nodes for the current state, the best of them is selected as current action. Greedy policies are like heuristic search but on a smaller scale. For example, to select a greedy action using the model and state value function, we need to predict future states for each possible action, consider the rewards and estimated values of those states, and then choose the action that yields the best outcome. Heuristic search can thus be understood as an extension of the greedy policy idea beyond a single step. Here, we assume we have a perfect model and an imperfect state value function. In such a scenario, deeper search usually leads to a better policy — if the search was carried out until the end of the episode, then indeed, the impact of the imperfect value function would be eliminated, and the action chosen in this way must be optimal. However, the deeper the search, the more computations are required.

An interesting example is the TD-Gammon program [11], created by Gerald Tesauro, which achieved a master-level performance in the game of backgammon. It uses a form of heuristic search to select moves and learns a value function over multiple self-play games. As a model, TD-Gammon utilized a priori knowledge about the probabilities of rolling specific dice outcomes and the assumption that the opponent always chose actions that TD-Gammon deemed best for it.

IV. Designing the algorithm

In our experiments, we want to explore how the environment model can affect the issue of non-stationarity in independent learning. Specifically, we want to find out whether using a model (even if imperfect) helps the agent reduce the sense of environment non-stationarity caused by other learning agents.

According to [5], there are several possible algorithmic connections between planning and learning, as shown in Fig. 2: a) learning a policy/value function from real data, b) learning the model from real data, c) planning using the model, d) acting based on the outcome of planning, e) acting based on the policy/value function, f) using the policy to improve the planning procedure, g) using the planning result to update the policy/value function. We need to consider how to use the environment model in our algorithm.

Suppose we have an ideal environment model, which implicitly must be in line with the policies of the other agents, and a well-designed approximate value function. We could use heuristic search to choose optimal actions in each state.

Algorithm 1 Designed model-based RL algorithm

// Algorithm controls agent i

Initialize predictive model $\hat{\mathcal{T}}_i$ and state-value function V^{π_i}
Repeat for every episode:
for $t = 0, 1, 2, \ldots$ **do**
  Observe current state s^t
  if explore with probability ε **then**
    Choose random action $a_i^t \in \mathcal{A}_i$
  else
    for each action $a_i \in \mathcal{A}_i$ **do**
      Obtain predicted next state $\hat{s}^{t+1}$ and predicted reward $\hat{r}_i^t$ from model $\hat{\mathcal{T}}_i$ using state s^t and action a_i
      Calculate value of the action $AV(a_i) \leftarrow \hat{r}_i^t + \gamma V^{\pi_i}(\hat{s}^{t+1})$
    end for
    Choose action $a_i^t = arg\max_{a_i} AV(a_i)$
  end if
  (meanwhile, other agents $j \neq i$ choose their actions a_j^t)
  Observe real reward r_i^t and next state s^{t+1}
  Update predictive model $\hat{\mathcal{T}}_i(s^t, a_i^t)$ using s^{t+1} and r_i^t
  $V^{\pi_i}(s^t) \leftarrow V^{\pi_i}(s^t) + \alpha[r_i^t + \gamma V^{\pi_i}(s^{t+1}) - V^{\pi_i}(s^t)]$
end for

A change in the environment dynamics (in this case, a change in someone's policy) would not affect an agent that relies entirely on its own model (assuming the model is not updated). This way, we would get rid of the non-stationarity problem. However, the model wouldn't be perfect anymore, so our future actions could be suboptimal. However, if the other agents also operate optimally, then there would be no change in dynamics in the first place.

In a situation with multiple learning agents, we cannot rely on a perfect environment model, and we must adjust it to accommodate the changing behavior of the other agents. Therefore, just like in the Dyna architecture, we need to update the model during learning. Given the inherent imperfection of the model, deep heuristic search is not feasible, as prediction errors in subsequent states will quickly accumulate. Therefore, we will plan only one step ahead.

We will approximate the values of future states using the state value function V^{π_i} for the current policy π_i of agent i (which is conditioned on the agent's environment model). This means it can only be updated using on-policy data. To make updates, we will apply one-step TD learning using experiences from interactions with the real environment. Due to the imperfection of the model, using it to acquire simulated experiences to improve V^{π_i} may be infeasible.

We hope that this way of integrating planning and learning (also presented in Fig. 3) will, at least to some extent, reduce the problem of non-stationarity. Algorithm 1 presents pseudocode of the designed model-based RL method.

V. Architectural Choices

During the experiments, we use following hyperparameters:

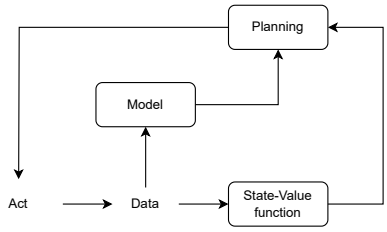

Fig. 3. Integration of planning and learning in the designed algorithm

- learning rate $\alpha = 0.0001$,
- discount rate $\gamma = 0.9$.

To encourage exploration, we use the ε-greedy method. The parameter ε will initially be $\varepsilon_{start} = 0.99$, which will gradually decrease to $\varepsilon_{end} = 0.05$ according to the formula:

$$\varepsilon = \varepsilon_{end} + (\varepsilon_{start} - \varepsilon_{end})e^{(-k/1000)}$$

where k is the total number of steps taken by the agent since the beginning of the learning process.

A. Environment Model

The environment model is a sample model which, given an input observation, returns the predicted next observations for each possible action, the predicted rewards for taking those actions, and information on whether each of these actions leads to a terminal state. It is a simple feedforward neural network with parameters θ composed of 2 hidden layers, each with 64 neuron units using the ReLU activation function.

The output layer predicting the next observations has a size equal to the number of actions multiplied by the observation size. The output layer predicting the rewards has a width equal to the number of actions. Both of these output layers do not use nonlinear activation functions.

The output layer predicting whether the next state will be terminal has a size equal to the number of actions and uses the sigmoid activation function to represent the probability, where a value of 1 indicates that the model is certain it will be a terminal state.

In the implementation, we use a memory buffer that stores the history of the last 1000 steps of the agent. At each step, a randomly selected batch of 32 experience tuples $(s, a, r, s', final)$, is retrieved from the buffer, where $final$ is an indicator whether state s' is a terminal state. Subsequently, based on this batch, the parameters of the environment model of this agent are updated using the mean squared error (MSE) loss:

$$\mathcal{L}(\theta) = MSE(\hat{s}', s') + MSE(\hat{r}, r) + MSE(\hat{final}, final)$$

For parameter updates, we employ the AdamW [12] optimizer with a learning rate α.

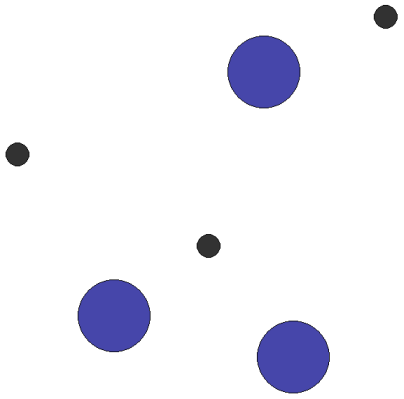

Fig. 4. *Simple Spread* environment with 3 agents (blue circles represent agents, and black dots are destination landmarks)

B. State-Value Function

The state-value function V^{π_i} or agent i's policy π_i is approximated using a feedforward neural network with parameters ϕ. This network consists of two hidden layers, similar to the environment model. It takes the state (observation) s as input, and its output layer returns a single value — the predicted state value $V^{\pi_i}(s)$ under policy π_i. The output layer does not use a nonlinear activation function. The network is updated at each timestep using the acquired tuple $(s, a, r, s', final)$ of real experience, utilizing the mean squared error loss function:

$$\mathcal{L}(\phi) = MSE(V^{\pi_i}(s), y_i)$$

where $y_i = r + \gamma V^{\pi_i}(s')$ if s' is not a terminal state, or $y_i = r$ otherwise. We use AdamW optimizer with a learning rate α for parameter updates.

VI. Experimental Setup

A. Environment

Multi-Particle Environments (MPE) [3] is a collection of environments where agents (particles) are placed in a two-dimensional space with designated landmarks. They can interact with the environment and with each other to achieve specific goals that require cooperation or competition.

One particular environment of interest is the navigation with cooperation available in the *PettingZoo* [13] library under the name *Simple Spread*. Fig. 4 illustrates this game, with blue particles representing the agents. In this environment, agents cooperate physically (without communication) to reach multiple landmarks. They observe the positions of other agents and points of interest, and their reward depends on how close they are to these points. The goal is to cover all landmarks while avoiding collisions, for which they are penalized. Agents learn which landmark to head towards and navigate there while avoiding other agents. In our experiments the game terminates after 32 timesteps.

B. Comparison Algorithm

We want to compare our model-based RL algorithm with a model-free one. Since our designed algorithm resembles Q-learning, we will compare it with the well-known Deep Q-Network (DQN) algorithm [14].

In the implementation, we use experience memory to store the agent's experiences from the last 1000 time steps. The Q-network is similar to the state-value network in our algorithm, but instead of a single value, it outputs one value for each possible action. It also uses AdamW optimizer with learning rate α. The target network is updated in every step using soft update:

$$\theta_{target} \leftarrow \tau\theta_{current} + (1-\tau)\theta_{target}$$

where $\tau = 0.005$, and θ_{target} and $\theta_{current}$ represent the parameters of the target network and the current network, respectively.

VII. Tests and Results

We trained our implemented algorithm and the DQN algorithm in the Simple Spread environment with 3 agents for 150,000 episodes. We compared the performance of the DQN algorithm with that of our model-based algorithm, illustrating the learning trends over time. The rewards for both algorithms were tracked and plotted to observe their respective improvements in performance throughout the training process.

The plot in Fig. 5 presents a moving average of the total rewards over 1000 episodes for these two algorithms. As training progresses, the plot shows the smoothed trend of the rewards received by each algorithm, highlighting their performance improvements over time. The moving average helps in reducing the noise in the reward signal, providing a clearer picture of the algorithms' learning behaviors.

In this test, our model-based method achieves better asymptotic performance than model-free DQN. DQN is an off-policy algorithm (meaning it can use the experience generated by policies other than the behavior policy to train its value function, thus enabling the reuse of old experience, which can significantly speed up the learning process) and is known for its high sample efficiency in single-agent RL. The fact that our on-policy algorithm (contrary to off-policy, it cannot use the outer experience to improve value function) achieves similar sample efficiency in the scenario with three agents must, therefore, be attributed to its more effective handling of the non-stationarity problem.

A. Single Agent

To demonstrate that DQN is indeed more sample efficient in single-agent RL, we trained both algorithms in the same environment but with only a single agent. Fig. 6 shows their performance improvements during training (also using moving average over 1000 episodes). In this test, both methods achieve similar asymptotic performance, but DQN reaches it much faster.

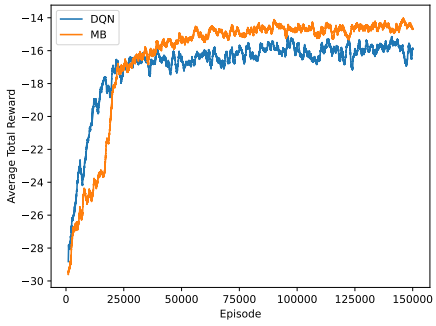

Fig. 5. Comparsion of DQN's and our model-based (MB) algorithm's learning performance in Simple Spread with 3 agents

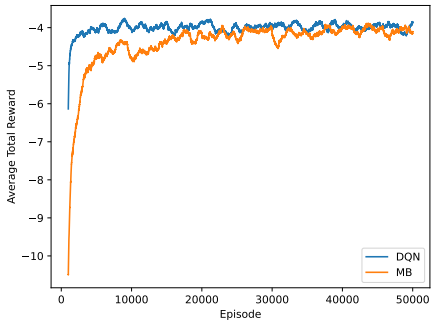

Fig. 6. Comparsion of DQN's and our model-based (MB) algorithm's learning performance in Simple Spread with 1 agent

B. More Agents

To determine if our model-based algorithm helps reducing non-stationarity, we compared it with DQN in Simple Spread environment with 4 and 6 agents. The presence of additional agents exacerbates the non-stationarity effect, allowing for a more discernible assessment of our algorithm's efficacy in addressing it. Fig. 7 and Fig. 8 show comparison plots for environments with 4 and 6 agents respectively. In both cases, the DQN agents failed to make progress in solving the problem due to the presence of non-stationarity, resulting in cyclic behavior. In contrast, our method showed signs of improved performance.

Conducted tests indicate that our model-based algorithm outperforms similar model-free method in multi-agent setups with decentralized learning. These setups are highly affected by non-stationarities, wherein our model-based approach demonstrates superior adaptability and performance.

VIII. Conclusion

We created a custom model-based algorithm for multi-agent reinforcement learning with decentralized training. Our algorithm uses the environment model to plan in predicted states in order to select actions. However we allow for the actions to be selected based on an imperfect model, enabling

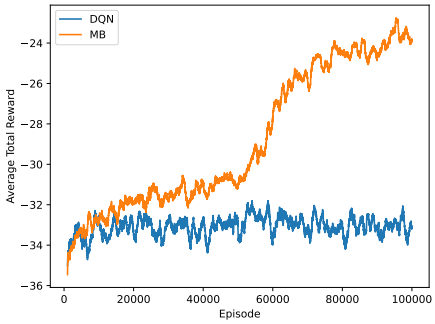

Fig. 7. Comparsion of DQN's and our model-based (MB) algorithm's learning performance in Simple Spread with 4 agent

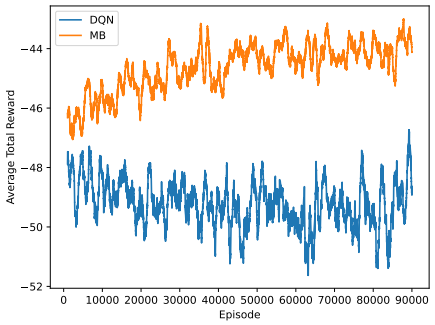

Fig. 8. Comparsion of DQN's and our model-based (MB) algorithm's learning performance in Simple Spread with 6 agent

the model to be learned and improved throughout the training process.

We tested our algorithm against the Deep Q-Network in a sample environment. As the number of agents increased, our model-based algorithm outperformed DQN. Therefore, it is reasonable to assume that this trend will continue with even more agents, although this needs to be confirmed with additional tests.

Our experimental findings suggest that utilizing an environment model can effectively help agents in mitigating the effects of non-stationarity induced by other adaptive agents. To confirm conclusively, additional experiments across a broader range of environments are required. Nevertheless, this work contributes to the understanding of how environment modeling can improve the effectiveness of multi-agent reinforcement learning algorithms. It presents an alternative approach to the challenges present in the field compared to most currently developed methods that rely on centralized training.

It may also be possible to use the model to generate simulated experience for improving the value function (as indicated by arrow g in Fig. 2) when the model is accurate enough (e.g., when model loss is sufficiently small). This could be the focus of further work.

REFERENCES

[1] S. V. Albrecht, F. Christianos, and L. Schäfer, *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024. [Online]. Available: https://www.marl-book.com

[2] L. S. Shapley, "Stochastic games*," *Proceedings of the National Academy of Sciences*, vol. 39, no. 10, pp. 1095–1100, 1953. doi: 10.1073/pnas.39.10.1095. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.39.10.1095

[3] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *CoRR*, vol. abs/1706.02275, 2017. doi: 10.48550/arXiv.1706.02275. [Online]. Available: http://arxiv.org/abs/1706.02275

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning*, 2nd ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2018. ISBN 978-0-262-03924-6. [Online]. Available: http://incompleteideas.net/book/the-book.html

[5] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, "Model-based reinforcement learning: A survey," 2022. doi: 10.48550/arXiv.2006.16712

[6] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine Learning Proceedings 1990*, B. Porter and R. Mooney, Eds. San Francisco (CA): Morgan Kaufmann, 1990, pp. 216–224. ISBN 978-1-55860-141-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558601413500304

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017. doi: 10.48550/arXiv.1712.01815. [Online]. Available: http://arxiv.org/abs/1712.01815

[8] M. Watter, J. T. Springenberg, J. Boedecker, and M. A. Riedmiller, "Embed to control: A locally linear latent dynamics model for control from raw images," *CoRR*, vol. abs/1506.07365, 2015. [Online]. Available: http://arxiv.org/abs/1506.07365

[9] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *SIGART Bull.*, vol. 2, no. 4, p. 160–163, jul 1991. doi: 10.1145/122344.122377. [Online]. Available: https://doi.org/10.1145/122344.122377

[10] W. Zhang, X. Wang, J. Shen, and M. Zhou, "Model-based multi-agent policy optimization with adaptive opponent-wise rollouts," 2022. doi: 10.48550/arXiv.2105.03363

[11] G. Tesauro, "Programming backgammon using self-teaching neural nets," *Artificial Intelligence*, vol. 134, no. 1, pp. 181–199, 2002. doi: 10.1016/S0004-3702(01)00110-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001102

[12] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. doi: 10.48550/arXiv.1711.05101

[13] J. K. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sullivan, L. Santos, R. Perez, C. Horsch, C. Dieffendahl, N. L. Williams, Y. Lokesh, and P. Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning," 2021.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. doi: 10.48550/arXiv.1312.5602