NTNU | Norwegian University of Science and Technology

A GENTLE INTRODUCTION TO DEEP REINFORCEMENT LEARNING

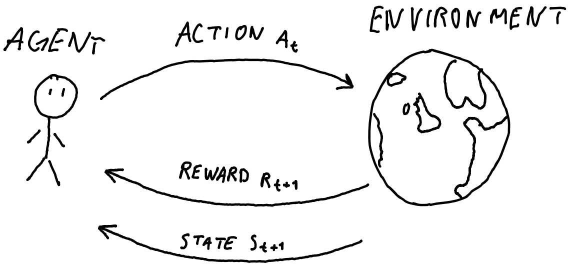
Even Klemsdal & Rudolf Mester

Outline

What is Reinforcement learning?

What is Reinforcement Learning?

- In Reinforcement Learning, an **agent** interacts with an **environment**.



What is Reinforcement Learning? (2)

- The goal of Reinforcement Learning is to **learn to interact** with the environment in an optimal way.
- This learning process is performed **without a 'teacher'**.
- Instead, the **learning is performed sequentially by interacting** with the environment.

What is Reinforcement Learning? (3)

- This is different from other types of learning
  - It is **active** rather than passive
  - Interactions are **sequential**,
    future actions can depend on earlier actions
- RL is **goal-directed**
- RL can learn **without examples** of optimal behavior
- Learning is performed by evaluating some **reward signal**

Reinforcement learning (RL) vs. Supervised Learning

- Reinforcement learning (RL) does not need labeled input/output pairs
- The output of RL are **actions**.
- There is no teacher who tells the RL agent the best action during training.
- RL receives only **hints** (rewards or penalties).
- The consequences of an action can become visible only after a delay
  $\rightarrow$ The immediate reward can be delusive / deceptive

Reinforcement learning (RL) vs. Unsupervised Learning

- RL does not learn clusters of input patterns.
- RL does not discover the structure of given data.
- RL learns to develop the best sequence of action in a **sequence** of decision problems.

Goals and rewards

As said already earlier:

- Reinforcement Learning attempts to attain a **goal** in its interaction with the environment.

RL is based on the reward hypothesis:

- *Any goal can be formalized as the outcome of maximizing a cumulative reward*

Formalizing the RL problem

Agent and environment

- We work in discrete time: $t_0, t_1, t_2 \ldots t_n, t_{n+1}, \ldots$
- At each step t, the agent:
  - receives observation O_t
  - executes action A_t
- The environment:
  - Receives action A_t
  - Emits observation O_{t+1} and reward R_{t+1}

Agent State

Agent state s_t = a representation of the information that an agent has about itself and the environment at a given time.

More specifically:

The **history** H_t is the complete sequence of observations, actions, and rewards available to the agent until time t.

The agent state s_t is usually a processed and compressed representation of that history.

This means: adding anything from the history H_t to the state s_t will not change the agent's behavior.

Rewards

- **Rewards** are feedback signals that measure the momentary success of an action in a particular situation.
- **Penalties** for inappropriate actions can be expressed by negative rewards.

- During an ongoing episode of running the agent, rewards are accumulated over time:

$$\text{Return} = \text{Total reward:} \qquad G_t = \sum_{i=t+1}^{\infty} r_i = r_{t+1} + r_{t+2} + \ldots + r_{t+n} + \ldots \tag{1}$$

Discounted Rewards

Principle: Rewards that come later in time are less valuable at the present time.
Leads to **discounted** forms of total reward:

$$\text{Discounted total reward:} \quad G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+1+k} = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 r_{t+3} + \ldots \tag{2}$$

Markov decision processes

A Markov decision process (MDP) is defined by:

- a set $\mathcal{S}$ of environment and agent states s_i ,
- a set $\mathcal{A}$ of actions a_i of the agent ,
- a stochastic model of the pcertarobability of transition (at time) from state s to state s' under action a :

$$P_a(s, s') \; := \; \Pr\left[s_{t+1} = s' \mid s_t = s \; \cap \; a_t = a\right] \tag{3}$$

- $r_a(s, s')$ is the immediate reward after the transition from s to s'
- Shorthand notation for the transition probability: $p(s', r \mid s, a)$

Note that it is not certain to which state s' the transition goes if a particular action is performed. The reward received by the agent is thus a **random entity**.

Policies

Obviously, the agent needs **rules** to decide on which action to take.

A **deterministic policy** π is a mapping from states to actions.

A **stochastic policy** π is a mapping from states to **probabilities** of selecting each possible action.

Stochastic policies

If the agent is following a stochastic policy π at time t,
then $\pi(a \mid s)$ is the probability that action a is chosen if the agent is in state s.

$\pi(a \mid s)$ is a probability distribution which is <u>conditioned</u> on the state s.

$\Rightarrow$ For each $s \in \mathcal{S}$ there is an individual probability distribution
for the actions $a \in \mathcal{A}$.

Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

Value functions

There are two different kinds of **value functions**:

1. a **state value function** $v(s)$ that estimates **how good it is to be in a particular state** s

2. an **action value function** $q(s, a)$ that estimates **how good it is to perform a given action** a **in a given state** s**.**

State Values $v(s)$

The value function $v_\pi(s)$ of a state s under a policy π, is the expected return when starting in s and following π after that.

$$v_\pi(s) \; := \; \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \tag{4}$$

The **expectation operator** $\mathbb{E}\left[...\right]$ is used here since the sequence of states that results from the action under policy π are, in general, **stochastic**.

$\mathbb{E}_\pi\left[....\right]$ means the expectation under the probabilistic model, which is given by the policy π.

Action Values $q(s, a)$

Action values $q(s, a)$ express the **expected total future reward** that an agent can receive by performing action a in state s and subsequently following policy π.

$$q_\pi(s, a) \;:=\; \mathbb{E}_\pi \left[G_t \mid S_t = s \;\cap\; A_t = a \right] \tag{5}$$

Again, this expectation value can only be formed if the policy $\pi(a \mid s)$ is fixed. We thus speak of action values $q_\pi(s, a)$ under a policy π .

Maximizing the Value Function

Assume now — for a moment — the true action value function $q(s_t, a_t)$ is known for any state s_t and any action a_t at any time t...

What would be the best choice for the agent's action?
$\Rightarrow$: Always choose the action that maximizes the expected total reward!
This yields then directly the **optimal policy** denoted as π^*:

$$\pi^*(s) = argmax_a \; q(s, a) \tag{6}$$

This means, a **learning agent** can

- either try to learn an approximation $Q(s, a)$ of the true action value function $q(s, a)$
- or try to learn a policy π.

But how do we perform the learning?

- In classical RL, 'learning' actually means building up **tables**, which are filled and modified by **exploration**.
- This means that one
  - starts with an initial choice of a policy or a value function,
  - chooses actions according to these
  - collects the rewards resulting from this
  - and updates the tables according to some **learning rule**.
- The problem that comes with this is that this may be computationally or memory-wise intractable if the action space and/or the state space are big.

But how do we perform the learning? (2)

The core of these learning rules is the **Bellman equation** (1957)

The Bellman equation for $v_\pi(s)$

We compute the state value $v_\pi(s)$ of being in a certain state s at time t:

$$
\begin{aligned}
v_\pi(s) \quad &:= \quad \mathbb{E}_\pi\left[G_t \mid S_t = s\right] && (7) \\
&= \quad \mathbb{E}_\pi\left[R_{t+1} + \gamma \cdot G_{t+1} \mid S_t = s\right] \\
&= \quad \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \cdot \left[r + \gamma \cdot \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\
&= \quad \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \cdot \left[r + \gamma \cdot v_\pi(s')\right] && (8)
\end{aligned}
$$

Once again: all this is valid for a fixed policy π

The Bellman equation (2)

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \cdot \left[r + \gamma \cdot v_\pi(s') \right] \tag{9}$$

This is a **recursive equation** that computes $v_\pi(s)$ from

- the rewards of all possible immediate successor states s'
- and the state **values** $v_\pi(s')$ that these future states have,
- considering the **stochastic transition model** $p(s', r \mid s, a)$
- and the **stochastic policy** $\pi(a \mid s)$.

With these fundamental concepts from classic reinforcement learning, we will now transition to **Deep Reinforcement Learning**.

Deep Reinforcement learning

Learning agent components

- All the components are functions
  - Policies: $\pi : \mathcal{S} \to \mathcal{A}$
  - Value functions: $v : \mathcal{S} \to \mathbb{R}$
  - Models: $m : \mathcal{S} \to \mathcal{S}$ and/or $r : \mathcal{S} \to \mathbb{R}$
  - State update: $u : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$
- We can use neural networks to represent the functions and deep learning techniques to learn them
- In order to do so, we start with value functions and/or policy functions initialized ’somehow’, run the agent, and modify these function approximators.
- Caveats with this: the state-action-reward sequences obtained this way are not uniformly distributed across the state space; subsequent states are usually ’similar’ ("correlated"). This is a problem for the learning process.

Recent DRL applications

Things you might have heard of

- DQN [MKS+13] — playing some Atari games at a superhuman level
- Alpha GO and Zero [SSS+17] — Playing perfect information games
- Alpha Star [VBC+19] — Playing complex real-time computer games and beating human players
- OpenAI Five [BBC+19] — Beating humans in the competitive 5v5 computer game Dota 2
- Emergent Tool use [BKM+19] — Implicit curricula where agents learn to build shelters

Recent DRL applications

Last year (2022)

- ChatGPT — Chatbot
- AlphaTensor [FBH+22] — Finding more efficient Matrix multiplication
- Mastering Stratego [PDVH+22] — Imperfect information game
- Architecture for tokamak magnetic controller design [DFB+22] — controlling magnetic fields of a fusion reactor

A quick deep-dive

- Value based
- Policy based
- Model based

DQN

Value based method

- We build an action value function $Q(s, a)$
- Greedy policy $\pi(s)$
- Building a value function through bootstrapping

Figure: Breakout

Q-Learning objective

- The TD update
  - $V(s_t) \leftarrow V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t))$
- For our objective using action-values we can update in the same way
  - $Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
- We can now run this update whenever we do a step in the environment
- If we reach all states with some probability we will get $Q^*(s, a)$ and π^*

Deep Q-learning

- We choose the representation of the action-value function
- This could be a simple table or a neural network
- If we are to use a neural network however, we need to address some of the problems.
  - Learning from a sparse noisy reward signal
  - Delay between reward and action
  - Correlated state visitations
  - No independent, identically distributed data
  - Single sample updates

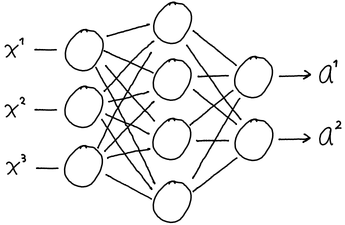


Figure: Neural Network policy

Using a replay buffer

- To stabilise the training of our Deep Q-network, [MKS+13] introduced the notion of experience replay.
- We save a tuple the tuple (s_t, a_t, s_{t+1}, r_t) for every step
- Since space might be limited, we store the tuples in a FIFO queue, often called replay buffer
- Instead of updating from a single training example, we randomly sample from the replay buffer
- This decorrelates the data w.r.t. time, and gives us larger batch updates of the network

Freezing the target network

- In Q-learning we are essentially updating our imagined value with a new imagined value
- Updating our neural network parameters to make $Q(s, a)$ closer to our desired result, will also change the value for other nearby states $Q(s^{'}, a^{'})$
- This will cause the training to be unstable.
- Therefore an often used trick is to use what is called a target network
- The target network is a copy of our Q-network, where the parameters are not trained
- Use the target network as a stable target.
- We update the target network periodically to synchronize with the real network

DQN algorithm

Initialize Replay Memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q with weights $\theta_{target} = \theta$
for $episode = 1$ to M **do**
    Get initial state s_1 from state observation x_1
    for $t = 1$ to T **do**
        With probability ϵ select random action a_t
        else select $a_t = arg\ \max_a Q(s_t, a_t; \theta)$
        Apply action a_t in environment and observe reward r_t and next s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(s_j, a'; \theta_{target}) & \text{otherwise} \end{cases}$
        Perform gradient descent step on $(y_j - Q(s_j, a'; \theta_{target}))^2$ with respect to the network parameters θ
        Every C steps set $\theta_{target} = \theta$
    end for
end for

Breakout

- Hit ball with paddle to break all colored boxes
- We consider the image as the state
- Reward every block we break
- Actions are moving the paddle left or right

Figure: Breakout

Downsides of Q-learning

The DQN algorithm achieves good results. However, there are a few downsides.

Complexity

- Its dependent on the size of the action space
- Action space must be discrete

Flexibility

- Policy comes from deterministically maximizing the Q-function
- This means we cannot learn stochastic policies

Policy Gradients

Policy Search

- Can we learn policies $\pi(a)$ directly, instead of learning values?
- One example could be if we defined action preferences $H_t(a)$ and a policy

$$\pi(a) = \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}}$$

- The preferences are not values: they are just learnable parameters
- Goal: Learn by optimizing the preferences

Moving probability mass

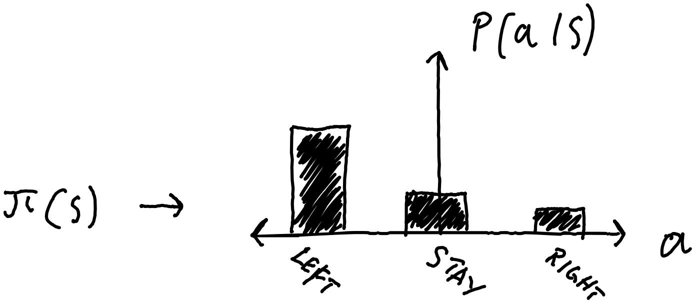


Figure: Discrete actions

Moving probability mass

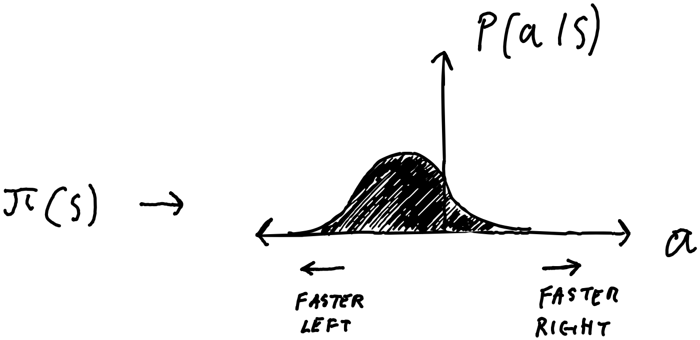


Figure: Continuous actions

Searching for policies

- We can use different methods to optimize this policy
- Bio-inspired methods such as Genetic algorithms or evolutionary strategies
- We can also use the gradients of the policy itself with regard to the RL objective

Training algorithm

1. Initialize the agent
2. Run the policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in a low reward
5. Increase probability of actions that resulted in high reward

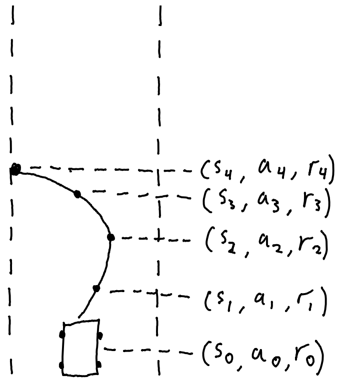


Figure: Sample policy

Training algorithm

1. Initialize the agent
2. Run the policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in a low reward
5. Increase probability of actions that resulted in high reward

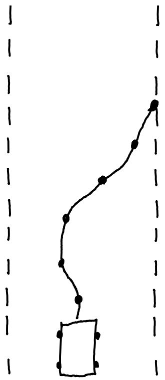


Figure: Policy improves

Training algorithm

1. Initialize the agent
2. Run the policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

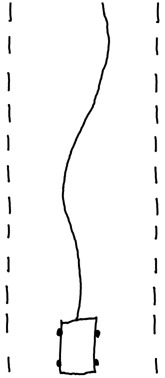


Figure: Learns not to crash

Gradients of RL objective

- Idea: Update policy parameters such that expected value increases
- We can use **gradient ascent**

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \mathbb{E}[R_t | \pi_{\theta_t}]$$

  where θ_t is the current policy parameters
- How can we compute this gradient?

REINFORCE

$$
\begin{aligned}
\nabla_\theta \mathbb{E}[R_t|\pi_\theta] =& \nabla_\theta \sum_a \pi_\theta(a)\mathbb{E}[R_t|A_t = a] \\
=& \sum_a \mathbb{E}[R_t|A_t = a]\nabla_\theta \pi_\theta(a) \\
=& \sum_a \mathbb{E}[R_t|A_t = a]\frac{\pi_\theta(a)}{\pi_\theta(a)}\nabla_\theta \pi_\theta(a) \\
=& \sum_a \pi_\theta(a)\mathbb{E}[R_t|A_t = a]\frac{\nabla_\theta \pi_\theta(a)}{\pi_\theta(a)} \\
=& \mathbb{E}\left[R_t \frac{\nabla_\theta \pi_\theta(A_t)}{\pi_\theta(A_t)}\right] = \qquad \mathbb{E}[R_t \nabla_\theta log \pi_\theta(A_t)] \qquad (10)
\end{aligned}
$$

REINFORCE

- This is known as the log-likelihood trick (also known as REINFORCE trick [Wil92]):

$$\nabla_\theta \mathbb{E}[R_t|\pi_\theta] = \mathbb{E}[R_t \nabla_\theta log \pi_\theta(A_t)]$$

- We can sample this, so our update becomes

$$\theta = \theta + \alpha R_t \nabla_\theta log \pi_\theta(A_t)$$

- We can use the sampled rewards - no need for value estimates

Downsides of Policy gradients

1. Policy gradient has a high variance
2. Convergence in policy gradient algorithms is slow
3. policy gradient is sample inefficient

PPO

- There is many ways to improve on the vanilla policy gradients
- One very popular variant is Proximal Policy Optimization (PPO) [SWD+17]
- In PPO, the main improvement is in sample efficiency and stability
- The main idea of PPO is to constrain the policy update not to wander too far from the policy that was used to generate the data.

PPO objective

- We start by defining the probability ratio

$$w_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

- so $w_t(\theta_{old}) = 1$
- PPO optimizes the following objective function

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right] \tag{11}$$

Model-based Reinforcement Learning

Model-based RL

- The idea of model-based RL is to employ a model of the world to inform the decision making.
- This allows agents to add planning to the decision-making

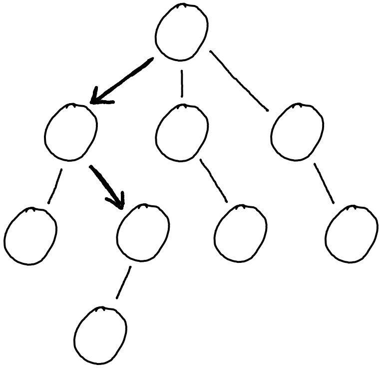


Figure: Searching with a model

Alpha Zero

- Alpha Zero is an example of a model-based RL method [SSS+17]
- The agent is given a perfect model of the environment
- This allows for use search to create better value estimates for the different actions.

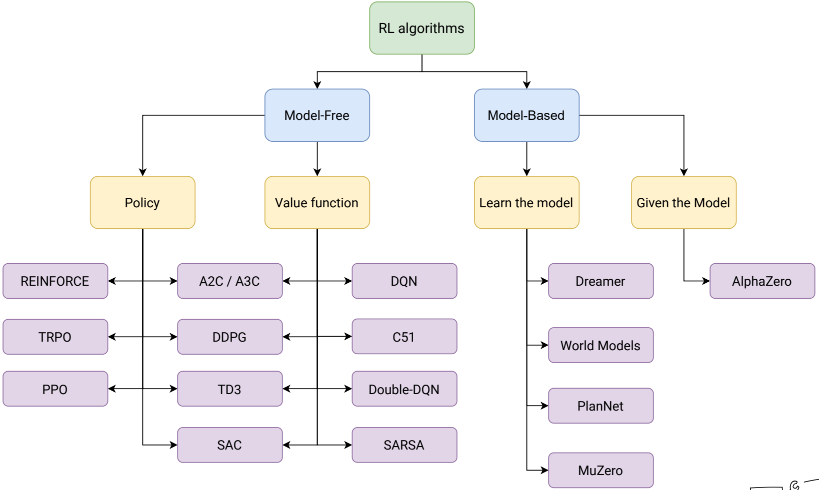
Alpha Zero

- Alpha zero uses MCTS to search the model of the game.
- This allows for a feedback loop where the search gives better value estimates
- and the better value estimates improve the rollouts in the MCTS search.

Learning the model

- We can also learn the model as we go
- An example of this is MuZero [SAH+20]
- MuZero is essentialy equivalent to Alpha Zero. However, we no longer need to provide the model
- MuZero learns its own dynamics model, then perform MCTS in the learned latent space

RL algorithms



Where to look for more materials?

- Reinforcement learning: An introduction
  - Available here
- Open AI Spinning Up
  - Available here
- UCL x Deepmind RL lecture series
  - Available here

References I

- Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al., Dota 2 with large scale deep reinforcement learning, arXiv preprint arXiv:1912.06680 (2019).
- Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch, Emergent tool use from multi-agent autocurricula, arXiv preprint arXiv:1909.07528 (2019).

References II

- Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al., Magnetic control of tokamak plasmas through deep reinforcement learning, Nature **602** (2022), no. 7897, 414–419.
- Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al., Discovering faster matrix multiplication algorithms with reinforcement learning, Nature **610** (2022), no. 7930, 47–53.

References III

- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013).
- Julien Perolat, Bart De Vylder, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer, Paul Muller, Jerome T Connor, Neil Burch, Thomas Anthony, et al., Mastering the game of stratego with model-free multiagent reinforcement learning, Science **378** (2022), no. 6623, 990–996.
- Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al., Mastering atari, go, chess and shogi by planning with a learned model, Nature **588** (2020), no. 7839, 604–609.

References IV

- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al., Mastering the game of go without human knowledge, nature **550** (2017), no. 7676, 354–359.
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv:1707.06347 (2017).
- Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al., Grandmaster level in starcraft ii using multi-agent reinforcement learning, Nature **575** (2019), no. 7782, 350–354.

References V

Ronald J Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine learning **8** (1992), no. 3, 229–256.