Literature review: Reinforcement Learning

Daniel Hernandez

Contents

1 Introduction

Reinforcement learning (RL) is an optimization framework. A problem can be considered a reinforcement learning problem if it can be framed in the following way: Given an enviroment in which an agent can take actions, receiving a reward for each action, find a policy that maximizes the expected cumulative reward that the agent will obtain by acting in the environment.

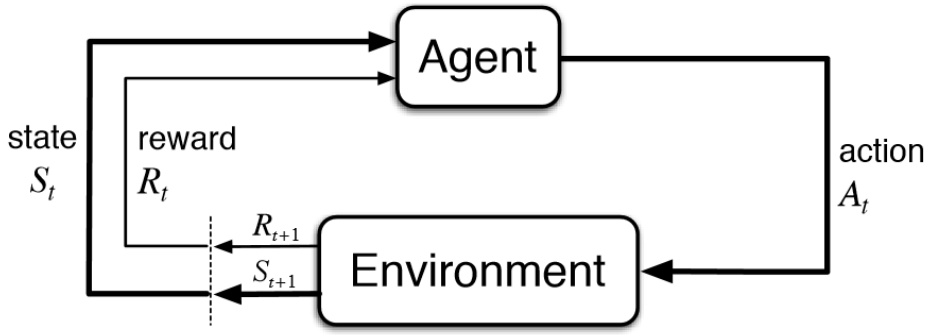

Figure 1: Reinforcement learning loop

1.1 Markov Decision Processes

The most famous mathematical structure used to represent reinforcement learning environments are Markov Decision Processes (MDP) (Bellman, 1957). Bellman introduced the concept of a Markov Decision Process as an extension of the famous mathematical construct of Markov chains. Markov Decision Processes are a standard model for sequential decision making and control problems. An MDP is fully defined by the 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \gamma)$. Whereby:

- $\mathcal{S}$ is the set of states $s \in \mathcal{S}$ of the underlying Markov chain, where $s_t \in \mathcal{S}$ represents the state of the environment at time t.
- $\mathcal{A}$ is the set of actions $a \in \mathcal{A}$ which are the transition labels between states of the underlying Markov chain. $A_t \subset \mathcal{A}$ is the subset of available actions in state s_t at time t. If an state s_t has no available actions, it is said to be a *terminal* state. Terminal states are equivalent to absorbing states in the Markov Chain literature.
- $\mathcal{P}(s_{t+1}|s_t, a_t) \in [0, 1]$, where $s_t, s_{t+1} \in \mathcal{S}$, $a_t \in \mathcal{A}$. $\mathcal{P}$ is the transition probability function[1]. It expresses a distribution over states. It defines the probability of transitioning to state s_{t+1} from state s_t after performig action a_t. Thus, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. Given a state s_t and an action a_t at time t, we can find the next state s_{t+1} by sampling from the distribution $s_{t+1} \sim P(s_t, a_t)$. The environment is said to be deterministic if $\mathcal{P}(s_t, a_t)$ is deterministic.
- $\mathcal{R}(s_t, a_t, s_{t+1}) \in \mathbb{R}$, where $s_t, s_{t+1} \in \mathcal{S}$, $a_t \in \mathcal{A}$. $\mathcal{R}$ is the reward function, which represents the immediate reward the agent will obtain after performing action a_t in state s_t and ending in state s_{t+1}. The real-valued reward[2] r_t is typically in the range $[-1, 1]$. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. If the environment is deterministic, the reward function can be rewritten as $\mathcal{R}(s_t, a_t)$.
- $\gamma \in [0, 1]$ is the discount factor, which represent the rate of importance between immediate and future rewards. If $\gamma = 0$ the agent cares only about the immediate reward, if $\gamma = 1$ all rewards r_t are taken into account. γ is often used as a variance reduction method, and aids proofs in infinitely running environments (Sutton et al., 1999).

There are also two optional parameters to an MDP which are sometimes ommited in the literature. These are the initial state distribution ρ_0 and the time horizon T:

[1] The function $\mathcal{P}$ is also known in the literature as the transition probability kernel, the transition kernel or the dynamics of the environment in model-based contexts. The word kernel is a heavily overloaded mathematical term that refers to a function that maps a series of inputs to value in $\mathbb{R}$.

[2] The reward r_t can be equivalently written as $r(s_t, a_t)$.

- $\rho_0(s_0) \in [0, 1]$, where $s_0 \in \mathcal{S}$, is the initial state distribution, representing the probability of starting an episode on a given state s_0. It is from this distribution that the initial state is sampled from $s_0 \sim \rho_0$.
- $T \in \mathbb{N}$, represents the finite time horizon, the number of steps over which the agent will try to maximize its cummulative reward. It serves a similar purporse to the more common discount factor γ in that they are both used as a variance-bias trade off and are needed for proofs of convergence over infinitely long running tasks.

The Greek letter ξ is sometimes used to represent the environment $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \gamma, \rho_0(\cdot), T)$.

Acting inside of the environment, there is the agent, and through its actions the transitions between the MDP states are triggered, advancing the environment state and obtaining rewards. The agent's behaviour is fully defined by its policy π. A policy $\pi(a_t|s_t) \in [0, 1]$, where $s_t \in \mathcal{S}$, $a_t \in \mathcal{A}$ is a mapping from states to a distribution over actions. Given a state s_t it is possible to sample an action a_t from the policy distribution $a_t \sim \pi(s_t)$. Thus, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$.

The reinforcement learning loop presented in figure 1 can be represented in algorithmic form as follows:

Algorithm 1: Reinforcement Learning loop.

Input: *Environment*: $\xi = (\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \gamma, \rho_0(\cdot), T)$
Input: *Agent Policy*: $\pi(\cdot)$
Sample initial state from the initial state distribution $s_0 \sim \rho_0$;
$\mathtt{t} \leftarrow 0$;
repeat
  Sample action $a_t \sim \pi(s_t)$;
  Sample successor state from the transition probability function $s_{t+1} \sim P(s_t, a_t)$;
  Sample reward from reward function $r_t \sim R(s_t, a_t, s_{t+1})$;
  $\mathtt{t} \leftarrow \mathtt{t} + 1$;
until $t \geq T$;

Given an environment ξ, the objective for the agent is to find an *optimal* policy π^*, which maximizes the cumulative sum of (possibly discounted) rewards.

$$\pi^* = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s \sim \xi, a \sim \pi}[\sum_{t=0}^{T} \gamma r_t] \tag{1}$$

All of the reinforcement learning research focuses on solving this optimization problem, where the exact elements of the right hand side of the equation depend on the model of the environment being used. Section 2 focuses on presenting a variety of other possible environment models.

1.2 Bellman equations and optimality principle

There are two functions of special relevance in reinforcement learning, the state value function $V^\pi(s)$ and the action value function $Q^\pi(s, a)$:

- The **state value function** $V^\pi(s)$ under a policy π, where $s \in \mathcal{S}$, represents the expected sum of rewards obtained by starting in state s and following the policy π until termination. Formally defined as $V^\pi(s) = \mathbb{E}^\pi[\sum_{t=0}^{\infty} r(s_t, a_t)|s_0 = s]$
- The **state-action value function** $Q^\pi(s, a)$ under a policy π, where $s \in \mathcal{S}$, $a \in \mathcal{A}$, represents the expected sum of rewards obtained by performing action a in state s and then following policy π. Formally defined as: $Q^\pi(s, a) = \mathbb{E}^\pi[r(s_0, a_0) + \sum_{t=1}^{\infty} r(s_t, a_t)|s_0 = s, a_0 = a]$

Bellman (1957) outlined two analytical recursive equations for the state value and action value function:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) * (r(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) * V^\pi(s')) \tag{2}$$

$$Q^\pi(s, a) = r(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) * (\sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a')) \tag{3}$$

Many algorithms in reinforcement learning research focus on approximating these two equations (Tamar et al., 2017; Watkins and Dayan, 1992; Mnih et al., 2013; Bertsekas, 2007).

The optimality principle, found in Bellman (1957), states the following: An optimal policy π^* has the property that given any initial state s_0 and initial action a_0, the remaining actions $a_{t>0}$ must constitute an optimal policy π^* with regard to the state s_1 resulting from the initial action a_0. The optimality principle, coupled with the proof of the existance of a deterministic optimal policy for any MDP as outlined in (Borkar, 1988) give rise to the optimal state value function $V^*(s) = V^{\pi^*}(s) = \text{argmax}_\pi V^\pi(s)$ and the optimal action value function $Q^*(s,a) = Q^{\pi^*}(s,a) = \text{argmax}_\pi Q^\pi(s,a)$. The optimal value functions determine the best possible performance in a MDP. An MDP is considered *solved* once the optimal value functions are found.

2 Environment models

Even though Markov Decision Processes are the most famous mathematical structure used to model an environment in reinforcement learning, there are other types of possible models for RL environment which act as extensions to vanilla MDPs. These mainly relax assumptions about the state space, the action space, the observability of the environment state, and the reward function. This section concerns itself to defining these extensions, and making links between them. This is not an exhaustive list of all possible mathematical models used to represent environments in the RL literature. However, these are some of the most used or fundamental models in the field, on which the majority of the research is conducted, and on top of which most niche extensions are built. Table 2 features all the environment models discussed in this section as well as their differences with respect with MDPs.

Model	Partial observability	Multi-agent	Multiple Reward functions	Delayed actions
MDP	×	×	×	×
SMDP	×	×	×	✓
POMDP	✓	×	×	×
MMDP	×	✓	×	×
dec-POMDP	✓	✓	×	×
Markov Game	×	✓	✓	×

Table 1: Properties of various environment models with respect to classical Markov Decission Processes.

2.1 Semi Markov Decision Process (SMDP)

As stated in (Barto, 2003), in an MDP, only the sequential nature of the decision process is relevant, not the amount of time that passes between decision points. Semi Markov Decision Processes do not assume thatthe time elapsed in between decision points, also known as decision stages, is constant. Every action taken in an SMDP has an assigned delay τ, known as *holding time*. When an action with holding time τ is taken state s_t, the agent waits for τ time before the action is executed and the next decision point s_{t+1} is reached. The agent then recieves a cumulative reward obtained throughout the elapsed timesteps, $r_t = \int_{t'=0}^{\tau} r_{t+t'}$. The time until the next decision point τ can only depend on the action a and state t and thus τ is independent of the history of the environment. SMDPs can also be used for real-valued time systems instead of discretely timed environments. This holding time allows for a gap in time between sensorial input reaching the agent and the agent's action being executed on the environment.

This type of process is considered Semi Markovian because as the holding time is elapsing, the agent cannot know how the system is evolving. Thus, in order to determine when the next state (decision point) will be reached, it is necessary to know how much time has elapsed, introducing temporal dependency, breaking the Markov property. This is formally described as: the probability of reaching state s_{t+1} depends only on s_t and action a_t with associated holding time τ. Once the action a_t has been decided, estimating when the agent will recieve state $s_{t+\tau}$ depends on how much time has elapsed since the action a_t was decided.

A Semi Markov Decision Process is defined by a 5-element tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot,\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot,\cdot), \gamma)$:

- $\mathcal{S}, \mathcal{A}$ and γ express the same concepts as in classical MDPs.
- $\mathcal{P}(s', \tau|s, a)$, where $s', s \in \mathcal{S}, \tau \in \mathbb{N}, a \in \mathcal{A}$, is the transition probability function. Which states the probability of transitioning to state s‘ after a holding time of τ.

- $\mathcal{R}(s, a, s', \tau)$, where $s', s \in \mathcal{S}, a \in \mathcal{A}$, is the reward function. It represents the expected reward of deciding on action a on state s with an assigned holding time of τ timesteps and landing on state s'.

A useful properties of SMDPs is that they can be reduced to regular MDPs through the *data-transformation method* (Piunovskiy and Zhang, 2012). This introduces the possibility of using MDP solving methods to solve SMDPs. SMDPs have recieved a lot of attention in the field hierarchical learning, especially with regards to options (Sutton and Barto, 1998).

2.2 Partially Observable Markov Decision Process (POMDP)

In an MDP, the internal representation of the environment is the same representation that the agent receives at every timestep. POMDPs introduce the idea that what the agent observes at every timestep t is only a partial representation o_t of the real environment state s_t. This partial observation o_t alone is not enough to reconstruct the real environment state s_t, which entails that $o_t \subset s_t$. A Partially Observable Markov Decision process is defined by a 6-element tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \Omega, \mathcal{O}(\cdot|\cdot,\cdot), \gamma)$:

- $\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}$ and γ express the same concepts as in classical MDPs.
- Ω is the set of all possible agent observations. Notably $\Omega \subset \mathcal{S}$, meaning that some of the state properties may never be available to the agent.
- $\mathcal{O}(o|s, a)$, where $o \in \mathcal{O}, s \in \mathcal{S}, a \in \mathcal{A}$, represents the probability of the agent recieving observation o after executing action a in state s.

In an POMDP the goal of the environment is not to find an optimal policy π^* *conditioned* on the history of environment observations which will maximize the expected cumulative reward. The agent samples actions from its policy, which is no longer conditioned on the state of the environment, as the agent does not have access to it, but rather it is conditioned on the sequence of the observations that the agent has obtained so far, $a_t \sim \pi(o_{\leq t})$. This goal is formalized as:

$$\pi = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s \sim \xi, a \sim \pi(\cdot|o_{\leq t})}[\sum_{t=0}^{T} \gamma r_t] \tag{4}$$

A POMDP can be reduced to an MDP iff, for all timesteps t the agent's observation o_t and the environment state s_t are equal $o_t = s_t$.

2.3 Multi-agent Markov Decision Process (MMDP)

A major shortcoming of MDPs is that they assume stationary environments, which by definition entails that the environment does not change over time. This assumption makes MDPs unsuitable for modelling multi-agent environments. Agents must be considered as non-stationary parts of the environment, because the policies that define their behaviours change over time through the course of learning, breaking the environment stationarity assumption.

Boutilier (1996) introduces Multi-agent Markov Decision Processes (MMDPs) as framework to study coordination mechanisms. MMDPs feature multiple agent policies, each of them submitting an individual action every timestep, which is executed as a joint action by the environment, producing a new state via the transition probability function.

A Multi-agent Markov Decission Process featuring k agents is defined by a 5-element tuple $(\mathcal{S}, \mathcal{A}_{1..k}, \mathcal{P}(\cdot,\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma)$:

- $\mathcal{S}$ and γ express the same concepts as in classical MDPs.
- $\mathcal{A}_{1..k}$ is a collection of action sets, one for each agent in the environment, with $\mathcal{A}_i$ corresponding to the action set of the ith agent.
- $\mathcal{P}(s'|s, \mathbf{a})$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the probability transition function. It states the probability of transitioning from state s to state s' after executing the *joint* action $\mathbf{a}$. The joint action is a vector containing the action performed by every agent at a certain timestep.

- $\mathcal{A}_{1..k}$ is a collection of action sets, one for each agent in the environment, with $\mathcal{A}_i$ corresponding to the action set of the ith agent.
- $\mathcal{R}(s, \mathbf{a})$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the reward function.

MMDPs can be thought as k-person stochastic games in which the payoff function is the same for all agents.

2.4 Decentralized Markov Decision Process (dec-POMDP)

Dec-POMDPs form a framework for multiagent planning under uncertainty (Oliehoek and Amato, 2014). This uncertainy comes from two sources. The first one being the partial observability of the environment, the second one stemming from the uncertainy that each agent has over the other agent's policies. It is the natural multi-agent generalization of POMDPs, introducing multi-agent concepts analogous to that of MMDPs. They are considereded *decentralized* because there is no explicit communication between agents. Agents do not have the explicit ability of sharing their observations and action choices with each other. Every agent bases its decision purely on its own individual observations. On every timestep each agent chooses an action simultaneously and they are all collectively submitted to the environment. As in MMDPs, all agents share the same reward function, making the nature of dec-POMDPs collaborative. A decentralized Partially Observable Markov Decision Process is defined by an 8-element tuple $(I, \mathcal{S}, \mathcal{A}_{1..k}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \Omega_{1..k}, \mathcal{O}(\cdot|\cdot,\cdot), T)$:

- $\mathcal{S}$ expresses the same concepts as in classical MDPs.
- $I = \pi_i, ..\pi_k$ is the set of all agent policies.
- $\mathcal{A}_{1..k}$ is a collection of action sets, one for each agent in the environment, with $\mathcal{A}_i$ corresponding to the action set of the ith agent.
- $\mathcal{P}(s'|s, \mathbf{a})$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the probability transition function. It states the probability of transitioning from state s to state s' after executing the *joint* action $\mathbf{a}$. The joint action is a vector containing the action performed by every agent at a certain timestep.
- $\mathcal{R}(s, \mathbf{a})$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the reward function.
- $\Omega_{1..k}$ represents the joint set of all agent observations, with Ω_i representing the set of all possible observations for the ith agent.
- $\mathcal{O}(\mathbf{o}|s, \mathbf{a})$, where $\mathbf{o} = \{o_1, \ldots, o_k\}$, $o_1 \in \Omega_1, \ldots, o_k \in \Omega_k$, $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the probability of observing the joint observation vector $\mathbf{o}$, containing an observation for each agent, after executing the joint action $\mathbf{a}$ in state s.
- $T \in \mathbb{N}$ represents the finite time horizon, the number of steps over which the agents will try to maximize their cummulative reward. It serves a similar purporse to the more common discount factor γ in that they are both used as a variance-bias trade off and are needed for proofs of convergence over infinitely long running tasks.

A dec-POMDP featuring a single agent, $|I| = 1$, can be treated as a POMDP. When theenvironment features full observability, the term dec-MDP is used.

2.5 Markov Game

Owen and Owen (1982) first introduced the notion of a Markov Game. Markov Games also serve to model multi-agent environments. They came to be as a crossbreed between game theoretic structures such as extended-form games and Markov Decision Processes. A Markov Game with k different agents is denoted by a 5-element tuple $(\mathcal{S}, \mathcal{A}_{1..k}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}_{1..k}(\cdot,\cdot), \gamma)$

- $\mathcal{S}$ and γ express the same concepts as classical MDPs.
- $\mathcal{A}_{1..k}$ is a collection of action sets, on for each agent in the environment, with $\mathcal{A}_i$ corresponding to the action set of the ith agent.

- $\mathcal{P}(s'|s, \mathbf{a})$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents probability transition function. It states the probability of transitioning from state s to state s' after executing the *joint* action $\mathbf{a}$. The joint action represents all of the actions that were executed at timestep t.

- $\mathcal{R}(s_t, \mathbf{a_t}) \in \mathbb{N}^k$, where $s_t \in \mathcal{S}$, $\mathbf{a_t} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the reward function. $\boldsymbol{r}_t \in \mathbb{N}^k$ is the reward vector. The reward $r_i \in \boldsymbol{r_t}$ is the reward that the ith agent will obtain after the joint action vector $\mathbf{a}$ is executed in state s.

Each agent independently tries to maximize its expected discounted cumulative reward, $\mathbb{E}[\sum_{j=0}^{\infty} \gamma^j r_{i,t+j}]$, where $r_{i,t+j}$ is the reward obtained by agent i at time $t + j$.

Markov Games have several important properties Owen and Owen (1982); Littman (1994). Like MDP's, Every Markov game features an optimal policy for each agent. Unlike MDPs, these policies may be *stochastic*. An intuitive advantage of stochastic policies stems from the agent's uncertainty about the opponent's pending moves. On top of this, sthocastic policies make it difficult for opponents to "second guess" the agent's action, which makes the policy less exploitable.

When the number of agents in a Markov Game is exactly 1, the Markov Game can be considered an MDP. When $|\mathcal{S}| = 1$, the environment can be considered a normal-form game from game theory literature. Doning a game theory hat, γ can be thought of as the probability of the game finishing next round. If all agents shared the same reward function, the Markov Game is reduced to an MMDP.

3 Categorization of RL algorithms

Every RL algorithm attempts to learn an optimal policy π^* for a given environment ξ. So far, there is not a single algorithm which is used in every single environment to find an optimal policy. The choice of algorithm depends on many factors, such as the nature of the environment, the availability of the underlaying mechanics of the environment, access to already existing policies and more practical constraints such as the amount of computational power available. More importantly, RL algorithms are not, against common belief, black boxes. They can be modularized and composed together to overcome the weaknesses of some approaches with the strenghts of others. For this, it is important to know what type of algorithms exist. Most RL algorithms can be divided into the following categories, note that not all of them are mutually exclusive:

3.1 On-policy and off-policy algorithms

If any RL algorithm can be regarded as a learning function mapping state-action-reward sequences, also known as paths or trajectories, to a policies (Laurent et al., 2011). Esentially all that RL algorithms do is applying a learning function over paths sampled from the environment and a policy. The key and only difference between on-policy and off-policy algorithms is the following:

- On-policy algorithms use the policy that they are learning about to sample actions in the environment. A policy π is both being improved overtime[3] and also used to sample actions $a \sim \pi(s)$ inside of the environment.

- Off-policy algorithms use a behavioural policy μ to sample actions $a \sim \mu(s)$ and paths inside of the environment, and use this information to improve a target policy π. The learning that takes place in off-policy algorithms can be regarded as learning from somebody else's experience, whilst on policy algorithms focus on learning from an agent's own experience.

On-policy algorithms dedicate all computational power on learning and using a single policy π. These methods can therefore focus on spending the computational resources on applying a learning function for the sole benefit of improving this policy. With off-policy algorithms it is possible to dedicate computational time to modifying both the behavioural policy μ and the policy π being learnt. The motivation behind this being that the paths sampled from the environment using π won't necessarily yield the best paths to learn from. However, by spending some of the computational resources to using, or even changing, the behavioural policy μ, it is possible to generate more "imformative" trajectories with which we can improve π through a learning function.

By freeing computational time from directly improving the target policy π, it is possible to tackle many other tasks. Sutton et al. (2011) use sensorimotor interaction with an environment to learn a multitude

[3]The notion of improvement overtime is expressed as a monotonic increase in the expected reward of an episode $\mathbb{E}_{a\sim\pi_0}[\sum_{t=0}^{\infty} r_t] < \mathbb{E}_{a\sim\pi_1}[\sum_{t=0}^{\infty} r_t]$

of pseudoreward that are used in conjunction with the environment's reward signals. Jaderberg et al. (2016) takes this idea further by using an off-policy algorithms to learn auxiliary extra tasks: immediate reward prediction[4] and a separate policy that maximizes the change in the state representation[5].

A method to allow algorithms to perform off-policy updates to their policies is to introduce the notion of an *experience replay* (Lin, 1993), which was made famous after the success of Mnih et al. (2013). An experience replay is a list of experiences, where each experience is a 5 element tuple $< s_t, a_t, r_t, s_{t+1}, a_{t+1} >$. As an agent acts in an environment the experience replay is filled. At the time of updating the policy an experience (or batch of experiences) is sampled uniformly from the experience replay. This is not the case with algorithms such as Q-learning, where updates to a value function happen always using the latest experiences. Because these sampled experiences may have been generated using a previous policy, experience replay allows for policy updates to happen in an off-policy fashion.

The idea of the experience replay buffer has been the focus of much recent research (Schaul et al., 2015; Hessel et al., 2017). A basic improvement is to use a *prioritized* experience replay. The difference being that experiences are not sampled uniformly from the replay buffer.

Famous on-policy algorithms: Sarsa (Sutton and Barto, 1998), $Q(\sigma)$ (De Asis et al., 2017), Monte Carlo Tree search (MCTS), REINFORCE (Williams, 1992), Asynchronous Advantage estimation Actor Critic (A3C).

Famous off-policy algorithms: Q-learning, Deep Q-Network (DQN) (Mnih et al., 2013), Deterministic Policy Gradient (DPG) (Silver et al., 2014). Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015), Importance Weighted Actor-Learner Architecture (IMPALA) (Espeholt et al., 2018).

3.2 Value based

Value based, also known as critic only methods, rely on deriving a policy π from a state value function $V(s)$ or a state action value function $Q(s, a)$. These include most if not all of the traditional RL algorithms. There are various methods for extracting a policy from a value function. The simplest form of deriving a policy from a value function is to create a policy that acts greedily w.r.t a value function:

$$\begin{aligned} &\forall s \in \mathcal{S}: \quad \pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s' \in Succ(s)} P(s' \mid s, a) V(s') && \text{(Deriving policy from } V(s)\text{)} \\ &\forall s \in \mathcal{S}: \quad \pi(s) = \underset{a}{\operatorname{argmax}}\, Q_\pi(s, a) && \text{(Deriving policy from } Q(s, a)\text{)} \end{aligned} \tag{5}$$

Some algorithms such as Q-learning, covered in Section 4, and SARSA bootstrap a state action value function $Q(s, a)$ towards the value functions of an optimal policy, $Q_{\pi^*}(s, a)$. Temporal Difference (TD) algorithms bootstrap a state value function $V(s)$ towards the value function of an optimal policy, $V_{\pi^*}(s)$. These algorithms have been proved to converge to the optimal value functions, so an optimal policy π^* can be derived once the bootstraping process converges.

Other methods, like Value iteration or Policy iteration go through an iterative loop of *policy evaluation* and *policy improvement*. The policy evaluation step computes a value function $V_\pi(s)$ or $Q_\pi(s, a)$ for a given a policy π, which is randomized on initialization. The policy improvement step extracts a new policy π' from the pre-computed value functions $V_\pi(s)$ or $Q_\pi(s, a)$ using equation 5. The next iteration of the loop is computed using π'. This two step process is proved to converge to both the optimal value function and optimal policy.

Famous value based algorithms: Value iteration, Policy iteration, SARSA, TD(0), TD(1), TD(λ) (Sutton and Barto, 1998), Q-learning, DQN and it's many variants: Dueling DQN, Distributional DQN, Prioritized DQN and Double DQN. These can be found in (Hessel et al., 2017)

3.3 Policy based

Both Policy based and actor-critic methods will be covered in depth in section 5. These algorithms do not extract a policy from a calculated value function. Instead, they represent a policy π_θ through a parameter vector $\theta \in \mathbb{R}^D$. By defining both the utility of a policy's parameters $U(\theta)$ and its gradient w.r.t the policy's parameters $\nabla_\theta U(\theta)$, it is possible to iteratively update the policy parameters in a direction of utility improvement.

[4]This is different from value function estimation because the value that the off-policy algorithm is trying to predict is expected immediate reward, instead of expected future cummulative reward.

[5]Given a matrix of pixels as input, the authors define "pixel control" as a separate policy that tries to maximally change the pixels in the following state. The reasoning behind this approach is that big changes in pixel values may correspond to important events inside of the environment.

Famous policy based algorithms: vanilla policy gradient, REINFORCE(Williams, 1992) and the REINFORCE family of algorithms.

3.4 Actor-critic

Policy based only (actor only) and value based only (critic only) algorithms feature some crippling weaknesses that make them unsuitable for complex environments. (Konda and Tsitsiklis, 2000) states some of their key downsides:

- **Critic only algorithms**: The goal of any reinforcement learning problem is to find a policy. Spending all computational resources on calculating a value function (the critic) is an indirect way of approaching the problem. Furthermore, most value based algorithms only converge under strict assumptions.
- **Actor only algorithms**: These algorithms rely on estimating the gradient of a policy's performance (the actor) w.r.t the policy's parameters. Gradient estimators tend to have large variance, leading to unstable parameter updates and, potentially, lost of convergence properties. More philosophically, because new gradients are calculated independently from previous gradients, there is no "learning" in the sense of understanding and consolidation of experience.

Actor-critic methods combine the strong points of both policy based and value based algorithms, and overcome some of their individual weaknesses. The critic has an approximation architecture to compute a value function. This value function is used to update the actor's policy parameters in a direction of improvement. This dynamic leads to faster convergence than actor-only methods because of a significant decrease in variance in the estimation of the gradient of the policy's performance. It also entails better convergence properties than critic-only methods.

Konda and Tsitsiklis (2000) make the key observation that in actor-critic algorithms, the actor parameterization and the critic parameterization should *not* be independent. The choice of critic parameters should be directly prescribed by the choice of the actor parameters. That is why all real world applications that implement an actor-critic algorithm use a single parameterized model (e.g. a neural network) to represent both the policy (actor) and the value function approximation (critic). This is the most straight forward way of sharing parameters between actor and critic.

Famous actor critic algorithms: A3C (Mnih et al., 2016), PPO (Schulman et al., 2017), TRPO (Schulman et al., 2015a), ACKTR (Wu et al., 2017).

3.5 Model based and model free approaches

In RL literature the *model* or the dynamics of an environment is considered to be the transition function $\mathcal{P}$ and reward function $\mathcal{R}$. Model free algorithms aim to approximate an optimal policy π^* without explicitly using either $\mathcal{P}$ or $\mathcal{R}$ in their calculations.

Model based algorithms are either given a prior model that they can use for planning (Browne et al., 2012; Soemers, 2014), or they learn a representation via their own interaction with the environment (Sutton, 1991; Guzdial et al., 2017; Deisenroth and Rasmussen, 2011). Note that an advantage of learning a model specifically tailored for an agent, is that you can choose a representation of the environment that is relevant to the agent's decision making process. (Talk about curiosity and modelling only that you care about) (Pathak et al., 2017). Another advantage of having a model is that it allows for forward planning, which is the main method of learning for search-based artificial intelligence.

4 Q-learning

The Q-learning algorithm was first introduced by Watkins (1989), and is arguably one of the most famous, most studied and most widely implemented methods in the entire field. Given an MDP, Q-learning aims to calculate the corresponding optimal action value function Q^*, following the principle of optimality and the proof of existence of an optimal deterministic policy in an MDP as described in Section 1.1. It is model free, learning via interaction with the environment, and it is an off-policy algorithm. The latter is because, even though we are learning the optimal action value function Q^*, we can choose any *behavioural* policy to gather experience from the environment. Researchers like Tijsma et al. (2017) benchmarked the efficiency of using various exploratory policies in grid world stochastic maze environments.

Q-learning has been proven to converge to the optimal solution for an MDP under the following assumptions:

1. The Q^{π^*} function is represented in tabular form, with each state-action pair represented discretely Watkins and Dayan (1992).
2. Each state-action pair is visited an infinite number of times. (Watkins, 1989)
3. The sequence of updates of Q-values has to be monotonically increasing $Q(s_i, a_i) \leq Q(s_{i+1}, a_{i+1})$. (Thrun and Schwartz, 1993).
4. The learning rate α must decay over time, and such decay must be slow enough so that the agent can learn the optimal Q values. Expressed formally: $\sum_t \alpha_t = \infty$ and $\sum_t (\alpha_t)^2 < \infty$. (Watkins, 1989)

Algorithm 2: Q-learning

Input: *Environment*: $\xi = (\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \gamma, \rho_0(\cdot), T)$

1 Initialize Q table $\forall s \in \mathcal{S} \wedge \forall a \in \mathcal{A}$, $Q(s,a) = 0$;
2 Sample $s_0 \sim \rho_0$;
3 $s = s_0$;
4 **repeat**
5 | Select action $a = \pi(s)$, where $\pi(s) = \epsilon - greedy(Q(s,\cdot))$;
6 | Observe successor state s' and reward r after taking action a ;
7 | Update $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \text{argmax}_{a'} Q(s',a') - Q(s,a)]$;
8 **until** $t \geq T$;

Q-learning features its own share of imperfections. If there is a function approximator[6] in place, Thrun and Schwartz (1993) shows that if the approximation error is greater than a threshold which depends on the discount factor γ and episode length, then a systematic overestimation effect occurs, negating convergance. This is mainly due to the joint effort of function approximation methods and the *argmax* operator used in step 7 of the algorithm. On top of this, Kaisers and Tuyls (2010) introduces the concept of *Policy bias*, which states that state-action pairs that are favoured by the policy are chosen more often, biasing the updates. Ideally all state-action pairs are updated on every step. However, because agent's actions modify the environment, this is generally not possible in absence of an environment model.

Frequency Adjusted Q-learning (FAQL) proposes scaling the update rule of Q-learning inversely proportional to the likelihood of choosing the action taken at that step (Kaisers and Tuyls, 2010). Abdallah et al. (2016) introduces Repeated Update Q-learning (RUQL), a more promising Q-learning spin off that proposes running the update equation *multiple times*, where the number of times is inversely proportional to the probability of the action selected given the policy being followed.

5 Policy gradient methods

5.1 Policy gradient theorem

Let's assume an stochastic environment ξ from which to sample states and rewards. Consider a stochastic control policy $\pi_\theta(a|s)$[7] parameterized by a parameter vector θ, that is, a distribution over the action set $\mathcal{A}$ conditioned on a state $s \in \mathcal{S}$. θ is a D-dimensional real valued vector, $\theta \in \mathbb{R}^D$, where D is the number of parameters (dimensions) and $D << |\mathcal{S}|$. The agent acting under policy π_θ is to maximize the (possibly discounted)[8] sum of rewards obtained inside environment ξ, over a time horizon H (possibly infinitely long). Under this formulation, the optimization problem presented in Section 1.1 equation 1 becomes an optimization problem over the policy's paramter space:

$$\max_\theta = \mathbb{E}_{s_t \sim \xi, a_t \sim \pi_\theta}[\sum_{t=0}^{H} r(s_t, a_t)|\pi_\theta] \tag{6}$$

There are strong motivations for describing the optimization problem on the parameter space of a policy instead of the already discussed approaches:

[6] With neural networks being the most famous function approximators in reinforcement learning at the time of writing.
[7] Some researchers prefer the notation $\pi(\cdot, \theta)$, $\pi(\cdot \mid \theta)$ or $\pi(\cdot; \theta)$. These notations are equivalent.
[8] Williams (1992); Sutton et al. (1999) present proofs of this same derivation using a discount factor, which makes policy gradient methods work for environments with infinite time horizons.

1. It offers a more direct way of approaching the reinforcement learning problem. Instead of computing the value functions V or Q and from those deriving a policy function, we are calculating the policy function directly.

2. Using stochastic policies smoothes the optimization problem. With a deterministic policy, changing which action to execute in a given state can have a dramatic effect on potential future rewards[9]. If we assume a stochastic policy, shifting a distribution over actions slightly will only slightly modify the potential future rewards. Furthermore, Many problems, such as partially observable environments or adversarial settings have stochastic optimal policies (Degris et al., 2012; Lanctot et al., 2017).

3. Often π can be simpler than V or Q.

4. If we learn Q in a large or continuous actions space, it can be tricky to compute $\underset{a}{argmax}\ Q(s,a)$.

For an episode of length H let τ be the trajectory followed by an agent in an episode. This trajectory τ is a sequence of state-action tuples $\tau = (s_0, a_0, \ldots, s_H, a_H)$. We overload the notation of the reward function $\mathcal{R}$ thus: $\mathcal{R}(\tau) = \sum_{t=0}^{H} r(s_t, a_t)$, indicating the total reward obtained by following trajectory τ. From here, the utility of a policy parameterized by θ is defined as:

$$U(\theta) = \mathbb{E}_{s_t \sim \xi, a_t \sim \pi_\theta}[\sum_{t=0}^{H} r(s_t, a_t) | \pi_\theta] = \sum_{\tau} P(\tau; \theta) R(\tau) \tag{7}$$

Where $P(\tau;\theta)$ denotes the probability of trajectory τ happening when taking actions sampled from a parameterized policy π_θ. More informally, how likely is this sequence of state-action pairs to happen as a result of an agent following a policy π_θ. Linking equations 6 and 7, the optimization problem becomes:

$$\max_{\theta} U(\theta) = \max_{\theta} \sum_{\tau} P(\tau; \theta) \mathcal{R}(\tau) \tag{8}$$

Policy gradient methods attempt to solve this maximization problem by iteratively updating the policy parameter vector θ in a direction of improvement w.r.t to the policy utility $U(\theta)$. This direction of improvement is dictated by the gradient of the utility $\nabla_\theta U(\theta)$. The update is usually done via the well known gradient descent algorithm. This idea of iteratively improving on a parameterized policy was introduced by Williams (1992) under the name of *policy gradient theorem*. In essence, the gradient of the utility function aims to increase the probability of sampling trajectories with higher reward, and reduce the probability of sampling trajectories with lower rewards.

Equation 9 presents the gradient of the policy utility function. The Appendix section shows the derivation from equation 7 to equation 9.

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)] \tag{9}$$

A key advantage of the policy gradient theorem, as inspected by Sutton et al. (1999) (and formalized in the Appendix Section), is that equation 9 does not contain any term of the form $\nabla_\theta P(\tau;\theta)$. This means that we don't need to model the effect of policy changes on the transition probability function. Policy gradient methods therefore classify as model-free methods.

We can use Monte Carlo methods to generate an empirical estimation of the expectation in equation 9. This is done by sampling m trajectories under the policy π_θ. This works even if the reward function R is unkown and/or discontinuous, and on both discrete and continuous state spaces. The equation for the empirical approximation of the utility gradient is the following:

$$\begin{aligned} \nabla_\theta U(\theta) &\approx \hat{g} = \frac{1}{m} \sum_{i=0}^{m} \nabla_\theta \log \pi_\theta(\tau^{(i)}) R(\tau^{(i)}) \\ \nabla_\theta U(\theta) &\approx \hat{g} = \frac{1}{m} \sum_{i=0}^{m} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) (\sum_{k=0}^{H} R(s_t^{(i)}, a_t^{(i)})) \end{aligned} \tag{10}$$

The estimate $\hat{g}$ is an unbiased estimate and it works in theory. However it requries an impractical amount of samples, otherwise the approximation is very noisy (has high variance). There are various techniques that can be introduced to reduce variance.

[9] An example of this concept are *greedy* or *ϵ-greedy* policies derived thus: $\pi(s) = \text{argmax}_{a \in \mathcal{A}} Q(s,a)$.

5.2 Baselines

Intuitively, we want to reduce the probability of sampling trajectories that are worse than average, and increase the probability of trajectories that are better than average. Williams (1992), in the same paper that introduces the policy gradient theorem, explores the idea of introducing a baseline $b \in \mathbb{R}$ as a method of variance reduction. These authors also prove that introducing a baseline keeps the estimate unbiased. This proof can be found in the Appendix. It is imporant to note that this estimate is not biased as long as the baseline at time t does not depend on action a_t. Introducing a baseline in equation 9 yields the equation:

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(\tau)(R(\tau) - b)] \tag{11}$$

The most basic type of baseline is the global average reward, which keeps track of the average reward across all episodes. We can also add time dependency to the baseline, such as keeping a running average reward. Greensmith et al. (2004) derive the optimal constant value baseline.

Furthermore, it is not optimal to scale the probability of taking an action by the whole sum of rewards. A better alternative is, for a given episode, to weigh an action a_t by the reward obatined from time t onwards, otherwise we would be ignoring the Markov property underlying the environment's Markov Decission Process by adding history dependency. Removing the terms which don't depend on the current action a_t reduces variance without introducing bias. This changes equation 11 to:

$$\nabla_\theta U(\theta) = \mathbb{E}_{s_t \sim E, u_t \sim \pi_\theta}[\nabla_\theta \sum_{t=0}^{H-1} \log \pi_\theta(a_t \mid s_t)(\sum_{k=t}^{H-1} R(s_k, a_k) - b)] \tag{12}$$

A powerful idea is to make the baseline state-dependent $b(s_t)$ (Baxter and Bartlett, 2001). For each state s_t, this baseline should indicate what is the expected reward we will obtain by following policy π_θ starting on state s_t. By comparing the empirically obtained reward with the estimated reward given by the baseline $b(s_t)$, we will know if we have obtained more or less reward than expected. Note how this baseline is the exact definition of the state value function V_{π_θ}, as shown in equation 13. This type of baseline allows us to increase the log probability of taking an action proportionally to how much its returns are better than the expected return under the current policy.

$$b(s_t) = \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \cdots + r_{H-1} \mid \pi_\theta] = V_{\pi_\theta}(s_t) \tag{13}$$

The term $\sum_{k=t}^{H-1} R(s_k, a_k)$ can be regarded as an estimate of $Q_{\pi_\theta}(s_t, a_t)$ for a single roll out. This term has high variance because it is sample based, where the amount of variance depends on the stochasticity of the environment. A way to reduce variance is to include a discount factor γ, rendering the equation: $\sum_{k=t}^{H-1} \gamma^k R(s_k, a_k)$. However, this introduces a slight bias. Even with this addition, the estimation remains sample based, which means that it is not generalizable to unseen state-action pairs. This issue can be solved by using function approximators to approximate the function Q_{π_θ}. We can define another real valued parameter vector $\phi \in R^F$, where F is the dimensionality of the parameter vector. From here, we can use ϕ to parameterize the function approximator $Q^\phi_{\pi_\theta}$. This function will be able to generalize for unseen state-action pairs.

$$\begin{aligned} Q^\phi_{\pi_\theta}(s,a) &= \mathbb{E}[r_0 + r_1 + r_2 + \cdots + r_{H-1} \mid s_0 = s, a_0 = a] && (\infty\text{-step look ahead}) \\ &= \mathbb{E}[r_0 + V^\phi_{\pi_\theta}(s_1) \mid s_0 = s, a_0 = a] && (1\text{-step look ahead}) \\ &= \mathbb{E}[r_0 + +r_1 + V^\phi_{\pi_\theta}(s_2) \mid s_0 = s, a_0 = a] && (2\text{-step look ahead}) \end{aligned} \tag{14}$$

Deciding on how many steps into the future to use for the state-value function $Q^\phi_{\pi_\theta}(s,a)$ entails a variance-bias tradeoff. The more actual sampled rewards r_t used in our state-value function estimation, the more variance is introduced, whilst reducing the variance from the function approximator.

Notice how we use the parameter vector ϕ to approximate the state value function V_{π_θ}. This approach can be viewed as an actor-critic architecture where the policy π_θ is the actor and the baseline b_t is the critic (Sutton and Barto, 1998; Degris et al., 2012).

5.3 Advantage function and Generalized Advantage Function (GAE)

The advantage function is denoted as $A_t(s_t, a_t) \in \mathbb{R}$ and it denotes how much better or worse an action is compared to the policy's default behaviour. This is captured by the expression:

$$A_t(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t) \tag{15}$$

Using the advantage function inside of the policy gradient estimation yields almost the lowest variance, although this equation is not known in practice, and must be estimated. This can be done, as mentioned before, by approximating the function V_{π_θ}.

Schulman et al. (2015b) introduces a very smart idea, which generalizes the n-step lookahead of equation 14. Instead of deciding on a single value for the number of lookahead steps, it is possible to take into account *all* of them simultaneously. Let's define A_π^n as the n-step lookahead advantage function. (Schulman et al., 2015b) introduces the generalized advantage estimation (GAE), parameterized by the discount factor $\gamma \in [0, 1]$ and a special parameter $\lambda \in [0, 1]$, which is used to manually tune yet another variance-bias tradeoff.

$$A_t^{GAE(\gamma,\lambda)} = (1 - \lambda)(A_t^1 + \lambda A_t^2 + \lambda^2 A_t^3 + \dots) \tag{16}$$

By choosing low values for, we are biasing the estimation of the advantage function towards low values of n for all n-step lookahead cases, reducing variance and increasing bias. If we use a higher value for λ, we increase the weight of the higher n values of the n-step lookahead cases. The GAE can be analytically written as:

$$A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{17}$$

Where $\delta_t^V = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$ is the TD residual for a given policy π as introduced in Sutton and Barto (1998). There are two notable cases of this formula, obtained by setting $\lambda = 0$ and $\lambda = 1$:

$$\begin{aligned} GAE(\gamma, 0) : A_t &= \delta_{t+l}^V && = r_t + \gamma V(s_{t+1}) - V(s_t) \\ GAE(\gamma, 1) : A_t &= \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V && = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t) \end{aligned} \tag{18}$$

5.4 Policy gradient equation summary

In summary, policy gradient methods maximize the expected total reward by repeatedly estimating the policy's utility gradient $g = \nabla_\theta \mathbb{E}[\sum_{t=0}^{\infty} r_t]$. There are many expresessions for the policy gradient that have the form:

$$g = \mathbb{E}[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\Psi] \tag{19}$$

The variable Ψ can be one of the following:

1. $\sum_{t=0}^{\infty} r_t$: total trajectory reward.
2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t.
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baseline version.
4. $Q_\pi(s_t, a_t)$: state-action value function.
5. $A_\pi(s_t, a_t)$: Advantage function.
6. $\delta_t^V = r_t + V_\pi(s_{t+1}) - V_\pi(s_t)$: TD residual.
7. $A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$: GAE

Out of the 7 different possibilities, state of the art algorithms use GAE(γ,λ), as it has been proved empirically and loosely theoretically that it introduces an acceptable amount of bias, whilst being one of the methods that most reduces variance.

6 Self-play

Classical approaches to training a good performing policy on multi-agent environment suffer from the problem of overfitting. This is occurs when learning agents become good at operating with or against themselves, but considerably drop performance when matched against other agents that act differently to those they have previously encountered. This issue is exacerbated when one knows not of any existing good opponent strategies to test against, which is also the main reason why reinforcement learning is the only viable machine learning paradigm for these cases.

Self-play is a training scheme which arises in the context of multi-agent training. Training using self-play means that learning agents learn *purely* by simulating playing with themselves. The learning algorithm does not have access to any pre-existing dataset during the course of training. Using existing datasets containing expert knowledge for a given environment is a common paradigm in reinforcement learning (Silver et al., 2016). However, a training scheme that relies on these cannot be considered self-play. This does not mean that the learning algorithm needs to start learning *tabula rasa*, as this constraint only takes effect during training, allowing for prior knowledge to be embedded into the agent before training begins.

The notion of self-play has been present in the reinforcement learning world for over half a century. (Samuel, 1959) discusses the notion of learning a state value function (which they call generalization procedure) to evaluate board positions by playing against itself in the game of checkers. Their training scheme consisted in having a simple tree search algorithm to select moves, this search used the state value function represented as a linear polynomial function. The TD-Gammon algorithm (Tesauro, 1995) used self-play to train an agent using TD(λ) (Sutton and Barto, 1998) to reach expert level backgammon play. The researchers used a feedforward neural network to represent the state value function. (Tesauro, 1992) argues in favour of the relevance of TD-Gammon because of the non-deterministic nature of the game. More recently, AlphaGo (Silver et al., 2017b) used a version of expert iteration (Anthony et al., 2017), which is built on top of self-play. This approach was capable of beating the world champion of go, and it generalized to the games of shogi and chess (Silver et al., 2017a).

It is often assumed that a training scheme can be defined as self-play if, and only if, the learning agent plays against the latest version of itself. When the agent's policy is updated, the other agents in the environment mirror this policy update. (Bansal et al., 2017) relaxes this constraint. They create a dataset of policies by checkpointing[10] the latest policy every fixed interval during training. They allow some of the agents to sample uniformly from this policy dataset every fixed amount of episodes. Thus, the latest policy is presented with a wider variety of behaviours, increasing its robustness[11]. This is an attempt to solve the problem of catastrophic forgetting[12].

Similar ideas have also been independently discovered in other fields. In psychology Treutwein (1995) introduces the *adaptive staircase procedure*, where a learning agent is presented with a set of increasingly difficult sets of trials. Upon succeding enough trials inside a difficullty level, the agent is presented with harder trials, and if it fails continually, it is demoted to an easier set. The link with (Bansal et al., 2017) and self-play is that sometimes the agent is presented with trials beloinging to easier levels of difficulty. Such procedure ensures that the agent does not forget how to solve trials outside its current level of difficulty. This procedure was proved to work for the deep reinforcement learning architecture UNREAL (Beattie et al., 2016) in virtual visual acuity tests (Leibo et al., 2018).

Self-play can be viewed as a natural curriculum learning problem, in which lessons are generated as training occurs. A dataset of historical policies which grows over the course of training can be regarded as a curriculum to which lessons are dynamically added. State of the art algorithms sample uniformly from this dataset of historical policies, but it could be interesting to experiment with other opponent sampling distributions. There is no research exploring which opponent sampling distribution works best for different types of environments. It may even be possible to *learn* this opponent sampling distribution during training using meta reinforcement learning. Duan et al. (2016) use meta reinforcement learning to learn a "learning algorithm" to perform well on a set of MDPs which is sampled from a given distribution over MDPs. Let's assume a dataset of policies that has been generated during training. Let's also assume that each of this policies is fixed. In a 2 agent environment, sampling a new opponent using some distribution over the set of avaialable policies is analogous to sampling a new MDP. Therefore it could be possible to translate the ideas of (Duan et al., 2016) into self-play, with the addition that the set of available policies (MDPs) would grow over time.

Self-play has been empirically verified as a valid training scheme, but the theoretical proofs are lagging behind, as it is the case with many corners of the field of machine learning (Henderson et al., 2017; Lipton and Steinhardt, 2018). An underlying issue of self-play studies is the fact that multi-agent environments are non-stationary and non-Markovian (Laurent et al., 2011), resulting in a loss of convergence guarantess for many algorithms. Beyond this, there are many open questions that arise regarding the nature of self-play. Tesauro (1992) observes that it is not clear a priori that such a learning system would converge to

[10] If a neural network is used to represent the policy, creating a checkpoint just means storing the weight values of the neural network at a given timestep.

[11] The definition of robustness in this scenario is specified as: successful against a variety of opponents without hyperparameter tuning.

[12] In a multi-agent reinforcement learning context, catastrophic forgetting refers to the event of a policy dropping in performance against policies for which it used to perform favourably.

a sensible solution, and no convergence proofs exist to this date.

Although the lack of theoretical guarantess does not mean lack of experimental results. (Van Der Ree and Wiering, 2013) experimented with two different training strategies one being learning by classical self-play (only using the latest policy for all agents) and the other training against a fixed opponent. They have empirical evidence that shows that for some algorithms learning by self-play yields a higher quality policy than learning against a fixed opponent, but that this is algorithm dependant. Concretely, TD-learning (Sutton and Barto, 1998) learnt best from self play, but Q-learning performed better when learning against a fixed opponent. Similarly, (Firoiu et al., 2017) found that Q-learning based algorithms did not perform well when trained against other policies which were themselves being updated simultaneously, but otherwise performed well when training against fixed opponents.

7 Learning Environments

An unknown but potentially large fraction of animal and human intelligence is a direct consequence of the perceptual and physical richness of our environment, and this intelligence is unlikely to arise without it. The information that agents require to learn does not exist in their psycology. It exists in the affordances of the environment outside their psycology. Thus, in order to train an agent to perform well in an environment, it is important to craft an environment capable of permitting good behaviour.

In the last decade there has been a rise of virtual environments which have been specifically designed to be used for RL research. They run much faster than real time, allow for customization and some even present inbuilt metric comparisons to check how well an algorithm fares against other existing implementations. This sections presents a non exhaustive list of the most famous learning environments.

7.1 Arcade Learning Environment

The arcade learning environment (ALE) (Bellemare et al., 2015) is an Atari 2600 emulator, where games are written in C++ and they are presented through a Python interface for launching and controlling simulations of over 70 Atari games. The ALE provides visual input through pixel-based rendering and a small discrete action space which is game-dependant. Its simulation speed is orders of magnitude faster than the original console. Its popularity skyrocketted when the Deep Q-Network algorithm (Mnih et al., 2013) was released, which was the first deep RL algorithm that played at superhuman performance on various Atari games where learning happened only on pixel input. Even though most research performed on ALE focuses on learning from pixel input, some researches use the small RAM of the simulated Atari 2600 as a state representation (Sygnowski and Michalewski, 2017). It is important to note that, due to the original Atari 2600 not showcasing a random number generator, all environments present in ALE are deterministic. Visuals are pixelated and relatively simple compare against other available environments. The environments can only be configured by changing their respective source code. They present only single-agent scenarios and lack realistic physics.

7.2 ViZDoom

ViZDoom is a Doom-based AI research platform for reinforcement learning from raw visual information. It allows developing AI bots that play Doom using only the game's screen buffer (Kempka et al., 2017). ViZDoom is primarily intended for research in machine visual learning, and deep reinforcement learning, ViZDoom provides researchers with the ability to create tasks which involve first-person navigation and control (Ha and Schmidhuber, 2018). Constrained by its underlying game engine, the visual and physical complexity possible in the environments created using ViZDoom are relatively limited. It also is restricted to simulating artificial agents with only a first-person perspective, which is the point of view of the original game.

7.3 DeepMind Lab

DeepMind Lab (Lab) was released in 2016 as the external version of the research platform used by DeepMind (Beattie et al., 2016). It is built from an open source version of the famous Quake III game engine, thus presenting basic physics and multi-agent interaction. Same as ViZDoom, the state observation presented to the agent are pixel values rendered in first person, with the adition of a velocity vector and an optional depth parameter for each pixel of the pixel values. The action space includes movement in three dimensions and look direction around two axes.

The Lab integrates complex level creation tools to aide environment design and bot scripting. Featuring procedurally generation of maps, it solves the deterministic problem present in all ALE environment. Furthermore, by using a 3D game-engine, complex navigation tasks similar to those studied in robotics and animal psychology could be studied within Lab (Leibo et al., 2018).

Similarly to ViZDoom, DeepMind lab is tied to a game engine which is over a decade old. As such, the gap in quality between the physical world, or other physics simulators such as MuJoCo, and the simulation provided via Lab is relatively large. Furthermore, the first person perspective limits the availability.

7.4 Project Malmo

Malmo (Johnson et al., 2016) is a multi-agent oriented RL framework based on the popular videogame Minecraft by Microsoft. Minecraft features super flexible map creation tools that allow for extensive environment customizability. Malmo uses this to allow for the creation of not only very varied map layouts, but also very different tasks. The platform is limited by the Minecraft engine. Due to the low-polygon pixelated visuals, as well as the rudimentary physics system, Minecraft lacks both the visual as well as the physical complexity that would be desirable from a modern platform.

7.5 MuJoCo

Multi-Joint dynamics with Contact (MuJoCo) is a famous physics engine developed by OpenAI, which excels in simulating friction, gravity, joint movement interaction and collision forces. It has become a popular simulation environment for benchmarking model-free continuous control tasks. The two major factors contributing to the popularity of MuJoCo are its high quality physics simulations and the large number of benchmarks with have become standardized for continuous-control algorithms. Even though MuJoCo features a nice state and action representation for RL agents, it lacks in visual quality. This can be seen in the lack of more complex lighting techniques, texture qualities and available materials for the objects in the environment. Furthermore, the MuJoCo engine was not designed to dynamically change the objects of the environment in real-time. More dynamic environments are often necessary to pose tasks which require greater planning or coordination to solve.

7.6 Real Time Strategy Games

Real time strategy (RTS) games have been a famous testbed for many reinforcement learning algorithms for a long time. RTS games are interesting to RL research fo a variety of reasons. They present some of the biggest action spaces in the whole RL literature, yielding game trees with enormous branching factors. Many of the actions that can be carried out inside of the game feature rewards which are very temporally delayed, this being one of the biggest obstacles to overcome in the field. On one hand, RTS games allow for micro unit control, where individual units act on their own action and observation spaces corresponding to their own limited vision and actions. But they also allow for macro control (Wender and Watson, 2012), which means simultaneous control of resources, unit creation and base expansion. All of them needing action sequences that feature heavily delayed rewards.

The most popular RTS game used in AI research is *Starcraft: Broodwar*, around which many frameworks have been built to facilitate research. Some frameworks inject code into the game to allow external processses to communicate and expose information from the game at runtime. One such example is TorchCraft (Synnaeve et al., 2016). Broodwar API (Heinermann, 2009) is the most widely spread of all *Starcraft: Broodwar* frameworks. (Vinyals et al., 2017) introduces the Starcraft II Learning environment (sc2le) which opens up *Starcraft II* for RL research. It tries to mimic the interface that humans interact with in *Starcraft II*, with a single rendered screen instead of a unit centered design, which is the approach taken by the Broodwar API. On top of the pixel rendered screen, The framework also provides a series of "feature layers" which facilitate learning. These layers present the same information as the RGB pixels except that the information is decomposed and structured. Sc2le is good for benchmarking AIs against human play, but makes multi-agent interactions virtually impossible, as a single screen view must be shared across agents.

There are other frameworks that also tackle RTS games. Some of these bet on simplicity and customizability at the expense of more interesting features such as fog of war, microRTS being one such example (Ontañón, 2013). Andersen (2017) released Deep RTS, a unifying RTS framework with an emphasize on simplicity, customizability, and simulation speed.

8 Appendix

8.1 Policy gradient derivation

Take equation 7 from Section 5, representing the utility of a policy π_θ parameterized by a D-dimensional real valued parameter vector $\theta \in \mathbb{R}^D$:

$$U(\theta) = \mathbb{E}_{s_t \sim \xi, a_t \sim \pi_\theta}[\sum_{t=0}^{H} r(s_t, a_t)|\pi_\theta] = \sum_{\tau} P(\tau;\theta)\mathcal{R}(\tau) \tag{20}$$

The goal is to find the expression $\nabla_\theta U(\theta)$ that will allow us to update our policy parameter vector θ in a direction that improves the estimated value of the utility of the policy π_θ. Taking the gradient w.r.t θ gives:

$$\begin{aligned}
\nabla_\theta U(\theta) &= \nabla_\theta \sum_{\tau} P(\tau;\theta)R(\tau) \\
&= \sum_{\tau} \nabla_\theta P(\tau;\theta)R(\tau) && \text{(Move gradient operator inside sum)} \\
&= \sum_{\tau} \nabla_\theta \frac{P(\tau;\theta)}{P(\tau;\theta)} P(\tau;\theta)R(\tau) && \text{(Multiply by } \frac{P(\tau;\theta)}{P(\tau;\theta)}\text{)} \\
&= \sum_{\tau} P(\tau;\theta)\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}R(\tau) && \text{(Rearrange)} \\
&= \sum_{\tau} P(\tau;\theta)\nabla_\theta \log P(\tau;\theta)R(\tau) && \text{(Note: } \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = \nabla_\theta \log P(\tau;\theta)\text{)} \\
\nabla_\theta U(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau;\theta)R(\tau)] && (\mathbb{E}[f(x)] = \sum_x xf(x))
\end{aligned} \tag{21}$$

This leaves us with an expectation for the term $\nabla_\theta \log P(\tau;\theta)R(\tau)$. Note that as of now we have not discussed how to calculate $P(\tau;\theta)$. Let's define the probability of a trajectory under a policy π_θ as:

$$P(\tau;\theta) = \Pi_{t=0}^{H} \underbrace{P(s_{t+1}|s_t, a_t)}_{\text{dynamics models}} \underbrace{\pi_\theta(a_t|s_t)}_{\text{policy}} \tag{22}$$

From here we can calculate the term $\nabla_\theta \log P(\tau;\theta)$ present in equation 21:

$$\begin{aligned}
\nabla_\theta \log P(\tau;\theta) &= \nabla_\theta \log[\Pi_{t=0}^{H} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)] \\
&= \nabla_\theta[(\sum_{t=0}^{H} \log P(s_{t+1}|s_t, a_t)) + (\sum_{t=0}^{H} \log \pi_\theta(a_t|s_t))] \\
\nabla_\theta \log P(\tau;\theta) &= \sum_{t=0}^{H} \underbrace{\nabla_\theta \log \pi_\theta(a_t|s_t)}_{\text{no dynamics required!}}
\end{aligned} \tag{23}$$

Plugging the result of equation 23 into equation 21 we obtain the following equation for the gradient of the utility function w.r.t to parameter vector θ:

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(\tau)R(\tau)] \tag{24}$$

Sutton et al. (1999) offers a different approach to this derivation by calculating the gradient for the state value function on an initial state s_0, calculating $\nabla_\theta V_{\pi_\theta}(s_0)$.

8.2 Unbiased reward baseline derivation

(Not quite) Take an equation 11 from Section 5, representing the gradient of the utility of a parameterized policy π_θ using a real valued baseline $b \in \mathbb{R}$:

$$\begin{aligned}
\nabla_\theta U(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau;\theta)(R(\tau) - b)] \\
&= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau;\theta)R(\tau)] - \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau;\theta)b] && \text{(Linearity of expectation)}
\end{aligned} \tag{25}$$

In order to prove that substracting the baseline b leaves the estimation unbiased, we need to show that the right hand side of the substraction evaluates to 0. Which is indeed the case:

$$
\begin{aligned}
&\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau;\theta)b] \\
&= \sum_\tau P(\tau;\theta)\nabla_\theta \log P(\tau;\theta)b && (\mathbb{E}[f(x)] = \sum_x xf(x)) \\
&= \sum_\tau P(\tau;\theta)\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}b && (\text{Note: } \nabla_\theta \log P(\tau;\theta) = \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}) \\
&= \sum_\tau 1 \times \nabla_\theta P(\tau;\theta)b && (\frac{P(\tau;\theta)}{P(\tau;\theta)} = 1) \\
&= b\sum_\tau \nabla_\theta P(\tau;\theta) && (\text{Move baseline outside of summation}) \\
&= b\nabla_\theta(\sum_\tau \nabla_\theta P(\tau;\theta)) && (\text{Move gradient operator outside of summation}) \\
&= b \times 0 = 0 && (\text{Apply gradient operator})
\end{aligned} \tag{26}
$$

Thus, adding a baseline leaves the gradient of the policy utility unbiased!

References

Abdallah, S., Org, S., Kaisers, M., and Nl, K. (2016). Addressing Environment Non-Stationarity by Repeating Q-learning Updates. *Journal of Machine Learning Research*, 17:1–31.

Andersen, P.-a. (2017). Deep RTS : A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games.

Anthony, T., Tian, Z., and Barber, D. (2017). Thinking Fast and Slow with Deep Learning and Tree Search. (Il):1–19.

Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., and Mordatch, I. (2017). Emergent Complexity via Multi-Agent Competition. 2:1–12.

Barto, A. G. (2003). Recent Advances in Hierarchical Reinforcement Learning Markov and Semi-Markov Decision Processes. *Most*, 13(5):1–28.

Baxter, J. and Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. (2016). DeepMind Lab. pages 1–11.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2015). The arcade learning environment: An evaluation platform for general agents. In *IJCAI International Joint Conference on Artificial Intelligence*, volume 2015-Janua, pages 4148–4152.

Bellman, R. (1957). *Dynamic Programming*, volume 70.

Bertsekas, D. P. (2007). *Dynamic Programming and Optimal Control, Vol. II*, volume 2.

Borkar, V. S. (1988). A convex analytic approach to Markov decision processes. *Probability Theory and Related Fields*, 78(4):583–602.

Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pages 195–210.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of {Monte Carlo Tree Search} methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

De Asis, K., Ca, K., Hernandez-Garcia, J. F., Ca, J., Holland, G. Z., and Sutton, R. S. (2017). Multi-step Reinforcement Learning: A Unifying Algorithm.

Degris, T., White, M., and Sutton, R. S. (2012). Off-Policy Actor-Critic.

Deisenroth, M. P. and Rasmussen, C. E. (2011). PILCO: A Model-Based and Data-Efficient Approach to Policy Search.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). RL$ˆ2$: Fast Reinforcement Learning via Slow Reinforcement Learning. pages 1–14.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures.

Firoiu, V., Whitney, W. F., and Tenenbaum, J. B. (2017). Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning.

Greensmith, E., Bartlett, P., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *The Journal of Machine Learning ...*, 5:1471–1530.

Guzdial, M., Li, B., and Riedl, M. O. (2017). Game Engine Learning from Video. *International Conference on Artificial Intelligence (IJCAI)*.

Ha, D. and Schmidhuber, J. (2018). World Models.

Heinermann, A. (2009). BWAPI: Brood war api, an api for interacting with starcraft: Broodwar.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2017). Deep Reinforcement Learning that Matters.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2016). Reinforcement Learning with Unsupervised Auxiliary Tasks. pages 1–14.

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. (2016). The malmo platform for artificial intelligence experimentation. *IJCAI International Joint Conference on Artificial Intelligence*, 2016-Janua:4246–4247.

Kaisers, M. and Tuyls, K. (2010). Frequency Adjusted Multi-agent Q-learning. *Learning*, pages 309–316.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaskowski, W. (2017). ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computatonal Intelligence and Games, CIG*.

Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-Critic Algorithms. *Nips*, 42(4):1143–1166.

Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Perolat, J., Silver, D., and Graepel, T. (2017). A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. (Nips).

Laurent, G. J., Matignon, L., and Fort-Piat, N. L. (2011). The world of independent learners is not markovian. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 15(1):55–64.

Leibo, J. Z., D'Autume, C. d. M., Zoran, D., Amos, D., Beattie, C., Anderson, K., Castañeda, A. G., Sanchez, M., Green, S., Gruslys, A., Legg, S., Hassabis, D., and Botvinick, M. M. (2018). Psychlab: A Psychology Laboratory for Deep Reinforcement Learning Agents. pages 1–28.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

Lin, L.-j. (1993). Reinforcement Learning for Robots Using Neural Networks. *Report, CMU*, pages 1–155.

Lipton, Z. C. and Steinhardt, J. (2018). Troubling Trends in Machine Learning Scholarship. pages 1–15.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. *Machine Learning Proceedings 1994*, pages 157–163.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. 48.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. pages 1–9.

Oliehoek, F. A. and Amato, C. (2014). Best Response Bayesian Reinforcement Learning for Multiagent Systems with State Uncertainty. *AAMAS Workshop on Multiagent Sequential Decision Making Under Uncertainty, MSDM 2014*, (May).

Ontañón, S. (2013). The combinatorial multi-armed bandit problem and its application to real-time strategy games. *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 58–64.

Owen, G. and Owen, G. (1982). Game Theory. *Collection*.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-Driven Exploration by Self-Supervised Prediction. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2017-July:488–489.

Piunovskiy, A. and Zhang, Y. (2012). The Transformation Method for Continuous-Time Markov Decision Processes. *Journal of Optimization Theory and Applications*, 154(2):691–712.

Samuel, A. (1959). Some studies in machine learning using the game of checkers. *Ibm Journal*, 3(3):210.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized Experience Replay. pages 1–21.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015a). Trust Region Policy Optimization.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-Dimensional Continuous Control Using Generalized Advantage Estimation. pages 1–14.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. pages 1–12.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of {Go} with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017a). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. pages 1–19.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Van Den Driessche, G., Graepel, T., and Hassabis, D. (2017b). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.

Soemers, D. (2014). Tactical Planning Using MCTS in the Game of StarCraft. page 12.

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*.

Sutton, R. S., Mcallester, D., Singh, S., and Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *In Advances in Neural Information Processing Systems 12*, pages 1057–1063.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., and White, A. (2011). Horde : A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction Categories and Subject Descriptors. *In Practice*, 2(1972):761–768.

Sygnowski, J. and Michalewski, H. (2017). Learning from the memory of Atari 2600. *Communications in Computer and Information Science*, 705:71–85.

Synnaeve, G., Nardelli, N., Auvolat, A., Chintala, S., Lacroix, T., Lin, Z., Richoux, F., and Usunier, N. (2016). TorchCraft : a Library for Machine Learning Research on Real-Time Strategy Games arXiv : 1611 . 00625v2 [cs . LG] 3 Nov 2016. pages 1–6.

Tamar, A., Wu, Y., Thomas, G., Levine, S., and Abbeel, P. (2017). Value iteration networks. *IJCAI International Joint Conference on Artificial Intelligence*, (Nips):4949–4953.

Tesauro, G. (1992). Practical Issues in Temporal Difference Learning. *Machine Learning*, 8(3-4):257–277.

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Commun. ACM*, 38:58–68.

Thrun, S. and Schwartz, A. (1993). Issues in Using Function Approximation for Reinforcement Learning. *Proceedings of the 4th Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, pages 1–9.

Tijsma, A. D., Drugan, M. M., and Wiering, M. A. (2017). Comparing exploration strategies for Q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*.

Treutwein, B. (1995). Adaptive Psychophysical Procedures.

Van Der Ree, M. and Wiering, M. (2013). Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL*, pages 108–115.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. (2017). StarCraft II: A New Challenge for Reinforcement Learning. *arXiv preprint arXiv:1708.04782*.

Watkins, C. J. (1989). *Learning from delayed rewards*. PhD thesis, King's College.

Watkins, C. J. and Dayan, P. (1992). Technical Note: Q-Learning. *Machine Learning*, 8(3):279–292.

Wender, S. and Watson, I. (2012). Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft : Broodwar. pages 402–408.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3):229–256.

Wu, Y., Mansimov, E., Liao, S., Grosse, R., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. pages 1–14.