Literature review: Reinforcement Learning

Daniel Hernandez

Contents

1 Introduction

1.1 Markov Decision Processes

subfiles

Bellman (1957) introduced the concept of a Markov Decision Process (MDP) as an extension of the famous idea 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 $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma)$. Whereby:

- $\mathcal{S}$ is the set of states $s \in \mathcal{S}$, where $s_t \in \mathcal{S}$ represents the state at time t.
- $\mathcal{A}$ is the set of actions $a \in \mathcal{A}$ and $A_t \subset \mathcal{A}$ is the subset of actions available in state s_t. If an state s_t has no available actions, it is said to be a *terminal* state.
- $\mathcal{P}(s'|s,a) = PSt_{t+1} = s|s_t = sa_t = a$ where $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$ is a transition kernel which states the probability of transitioning to state s' from state s after performig action a. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. If the environment is stochastic, as opposed to deterministic, the function $\mathcal{P}$ maps a state-action pair to a distribution over states in $\mathcal{S}$.
- $\mathcal{R}(s,a)$ where $s \in \mathcal{S}$, $a \in \mathcal{A}$; is the reward function, which returns the immediate reward (typically in the range $[-1,1]$) of performing action a in state s. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$. The reward at time step t can be interchangably written as r_t or $r(s_t, a_t)$.

- $\gamma \in [0, 1]$ is the discount factor, which represent the rate of importance between the current reward 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, this is only allowed in episodic tasks, as otherwise $t \to \infty$ (Sutton et al., 1999). γ is often used as a variance reduction method, and aids proofs in infinitely running environments.

From here we can introduce the notion of an agent. (link to control theory?), An agent is an entity that on every state $s_t \in \mathcal{S}$ it can take an action $a_t \in \mathcal{A}$ in an environment transforming the environment from s_t to s_{t+1}. The beheviour of an agent is fully defined by a policy π. A policy π is a mapping from states to actions, $\pi : \mathcal{S} \to \mathcal{A}$. The agent chooses which action a_t to take in every state s_t by querying its policy such that $a_t = \pi(s_t)$. If the policy is stochastic, π will map an action to a distribution over action $a_t \sim \pi(s_t)$. The objective for an agent is to find an *optimal* policy, which tries to maximize the cumulative sum of possibly discounted rewards.

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: $Q^\pi(s, a) = \mathbb{E}^\pi[r(s_0, a_0) + \sum_{t=1}^{\infty} r(s_t, a_t)|s_0 = s, a_0 = a]$

The Bellman equations are the most straight forward, dynamic programming approach at solving MDPs (Bertsekas, 2007; Bellman, 1957).

1.2 Bellman equations and optimality principle

Note that in general it is not the case that all actions $a \in \mathcal{A}$ can be taken on every state $s_t \in \mathcal{S}$.

The optimality principle, found in Bellman (1957), states the following: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. 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 $V^*(s) = \text{argmax}_\pi V^\pi(s) = V^{\pi^*}(s)$ and the optimal action value function $Q^*(s, a) = \text{argmax}_\pi Q^\pi(s, a) = Q^{\pi^*}(s, a)$. The optimal value functions determine the best possible performance in a MDP. An MDP is considered *solved* once the optimal value functions are found.

Most of the field of reinforcement learning research focuses on approximating these two equations (Tamar et al., 2017) (Watkins and Dayan, 1992) (Mnih et al., 2013). (cite many more)

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

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) * (r(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) * V^\pi(s')) \tag{1}$$

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

Most RL algorithms can be devided into the following categories: Policy based Value based actor critic

A further categorization of algorithms is the notion of *model free* and *model based* algorithms. Consider a *model* of an environment to be the transition function $\mathcal{P}$ and reward function $\mathcal{R}$. Model free algorithms aim to approximate an optimal policy without them. 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). Note that an advantage of learning your own model is that you can choose a representation of the environment that is relevant to the agent's actions, which can have the advantage of modelling uninteresting (but perhaps complicated) environment behaviour (Pathak et al., 2017).

1.3 Categorization of RL algorithms

subfiles

Explain control vs prediction.

Most RL algorithms can be devided into the following categories:

1.3.1 Policy based

These algorithms tend to represent a policy through a parameter vector $\theta \in \mathbb{R}^D$. The goal then becomes to improve the choice of parameters $\theta_i \in \theta$ to improve the expected sum of rewards $\mathbb{E}[\sum_T r(t)]$. Policy based algorithms are the main focus on Section 3.

Famous policy based algorithms: vanialla policy gradient, REINFORCE family of algorithms.

1.3.2 Value based

Value based, or critic only methods, rely exclusively on value function approximation and aim at learning an approximate solution to the Bellman equation. The underlying assumption is that from the value function, a near optimal policy can be computed.

Famous value based algorithms: Value iteration, Policy iteration, Sarsa, Q-learning

1.3.3 Actor critic

Actor critic methods combine the strong points of both policy based and value based algorithms, and overcome some of their individual weaknesses. The critic assumes the role of learning a value function, which is then used as part of the update for the actor's policy. The individual critic is analogous to value based algorithms, and the actor to policy based methods.

Famous actor critic algorithms: A3C, A2C, PPO, TRPO, ACKTR.

1.3.4 Model free, model based

Consider the *model* or the dynamics of an environment to be the transition function $\mathcal{P}(s_{t+1}|s_t, a_t)$ and reward function $\mathcal{R}(s_t, a_t, s_{t+1})$. 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). Note that an advantage of learning your own model is that you can choose a representation of the environment that is relevant to the agent's actions, which can have the advantage of modelling uninteresting (but perhaps overly complicated) environment behaviour (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 (throw mcts papers here).

1.3.5 on-policy, off-policy

On policy methods use a policy π to gather experience on an environment in order to improve that same policy π. Off policy methods a behavioural policy μ to carry out actions in an environment, and use this information to improve a target policy π. The learning that takes place in off policy methods can be regarded as learning from somebody else's experience, whilst on policy methods focus on learning from an agent's own experience.

A method to alllow 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, in the same fashion as in the reinfocement learning loop presented in (reference), the experience replay is filled. At the time of updating the policy, the agent does not choose to update its policy using the last action function that was taken, as it is the case with Q-learning. Instead, the agent samples an experience (or batch of experiences) from the experience replay. 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.

(see what to include) In an off-policy setting, on the other hand, an agent learns about a policy or policies different from the one it is executing Unlike on-policy methods, off-policy methods are able to, for example, learn about an optimal policy while executing an exploratory policy. Or learn from demostration (Smart and Kaelbling, 2002). and learn multiple tasks in parallel from a single sensorimotor interaction with an environment (Sutton et al., 2011).

Some notable on policy algorithms: Sarsa, Monte Carlo estimation for prediction, Monte Carlo Tree search methods, Many vanilla policy gradient methods such as REINFORCE(Williams, 1992) family of algorithms, asynchronous advantage actor critic method (A3C and A2C).

Some notable off policy algorithms are. Q-learning, deep Q-learning, deep deterministic policy gradient

1.4 Common problems in Reinforcement Learning

1.4.1 Credit assignment problem

As we have previously discussed, one of the difficulties of reinforcement learning is that feedback for actions is often delayed. The credit assignment problem focuses on the question: if a sequence of actions led to a reward, how much credit should each action take for obtaining that reward? It is against common sense to assume that every action should be equally rewarded or punished. This is commonly known as the credit assignment problem.

1.4.2 Exploration vs exploitation

The exploration vs exploitation dilemma corresponds to finding a good middle point between exploting the best action available at any given time (exploitation), or performing a sub optimal action in the hopes that the agent will learn more about the environment that could lead to higher expected rewards in the long run. A common approach in stochastic environments, the trade off between exploration and exploitation is tackled by the following idea. For each state $s_t \in \mathcal{S}$ we want to maximize the expected sum of rewards $\mathbb{E}_\pi[V_\pi(s_t)]$ and reduce the variance $\sigma^2_{s_t}$. The Upper Bound Confidence Monte Carlo Tree Search algorithm UCT-MCTS has a remarkable example of this tradeoff as part of its selection policy:

2 Q-learning

subfiles

The Q-learning algorithm was first introduced by Watkins (1989), and is arguably one of the most famous and 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 proof of the existence of an optimal policy as described in Section 1.1. It is model free, learning via interaction with the environment, and it is an offline algorithm. The latter is because, even though we are learning the optimal action value function Q^*, we can choose any to gather experience from the environment with any policy of our choosing. This policy is often named an *exploratory* policy. 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 some assumptions:

1. Each state-action pair is visited an infinite number of times. (Watkins, 1989)
2. 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).
3. 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 1: Q-learning

```
1 repeat
2     Select action a from policy ;
3     Observe successor state s' and reward r after taking action a ;
4     Update $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \text{argmax}_{a'} Q(s',a') - Q(s,a)]$ ;
5 until done;
```

Q-learning features its own share of imperfections, Watkins and Dayan (1992) tell us that the algorithm converges to optimality with probability 1 if each state-action pair is represented discretely, that is, if it is implemented in tabular form. If there is a function approximator[1] 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, which happens mainly due to the joint effort of function approximation methods and the max operator. 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.

3 Policy gradient methods

subfiles

3.1 Policy gradient theorem

In order to comply with notation used in the field of direct optimization, we shall use u for actions.

Consider a stochastic control policy $\pi_\theta(s)$ parameterized by a parameter vector θ, that is, a distribution over the action set $\mathcal{A}$ given 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 $D << |\mathcal{S}|$. This prameterized policy function will be denoted by π_θ.[2].

There are strong motivations for using policy gradient approaches versus the already discussed RL methods:

1. A more direct way of approaching the 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 do in a given state can have a dramatic effect on potential future rewards[3]. 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).
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 $\underset{\text{u}}{argmax}\ Q(s, u)$.

(they are not, though. REINFORCE is, but dqn, ddpg arent!) Policy gradient methods are on-policy. In them, the agent acts with using a policy which is improved gradually over time. This contrasts with off-policy algorithms, such as the Q-learning algorithm introduced in Section 2, which allows the agent to interact with the environment with a policy while it is simultaneously learning another policy. There is ongoing research looking at off-policy variants of policy gradient methods (Mnih et al., 2013, 2016).

Let's assume an stochastic environment E from which to sample states and rewards, and an stochastic policy π_θ parameterized by a vector θ from which to sample actions. The agent acting under policy π_θ is to maximize the (possibly discounted)[4] sum of rewards on environment E, over a time horizon H (possibly infinitely long). We reach the following optimization problem:

$$\max_\theta = \mathbb{E}_{s_t \sim E, u_t \sim \pi_\theta}[\sum_{t=0}^{H} r(s_t, u_t) | \pi_\theta] \tag{3}$$

[1]With neural networks being the most famous function approximators in reinforcement learning at the time of writing.

[2]Some researchers prefer the notation $\pi(\cdot, \theta)$, $\pi(\cdot \mid \theta)$ or $\pi(\cdot; \theta)$. These notations are equivalent.

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

[4]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.

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: $\mathcal{R}(\tau) = \sum_{t=0}^{H} r(s_t, u_t)$, indicating the total accumulated reward in the trajectory τ. We will also use $r(s_t) \in \mathbb{R}$ to refer to the scalar reward obtained at timestep t in the trajectory. From here, the utility of a policy parameterized by θ is defined as:

$$U(\theta) = \mathbb{E}_{s_t \sim E, u_t \sim \pi_\theta}[\sum_{t=0}^{H} r(s_t, u_t)|\pi_\theta] = \sum_{\tau} P(\tau; \theta)\mathcal{R}(\tau) \tag{4}$$

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 3 and 4, our optimization problem becomes:

$$\max_\theta U(\theta) = \max_\theta \sum_{\tau} P(\tau; \theta)\mathcal{R}(\tau) \tag{5}$$

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 is 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 6 presents the gradient of the policy utility function. The Appendix section shows the derivation from equation 4 to equation 6.

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)] \tag{6}$$

A key advantage of the policy gradient theorem, as inspected by Sutton et al. (1999) (and formalized in the Appendix Section), is that equation 6 does not contain any term of the form $\nabla_\theta \mathcal{P}(\tau; \theta)$. This means that we don't need to model the effect of policy changes on the distribution of states. 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 6. 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:

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=0}^{m} \nabla_\theta \log \pi_\theta(\tau^{(i)}) R(\tau^{(i)}) \tag{7}$$

The estimate $\hat{g}$ is unbiased estimate and it works in theory. However it requries an impractical amount of samples, otherwise the approximation is very noisy. In order to overcome this limitation we can do the following tricks:

- Add a baseline
- Add temporal structure (advantage function)
- Use trust region and natural gradient.

3.2 Baselines

Intuitively, we want to reduce the probability of 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 as a method of variance reduction, where $b \in \mathbb{R}$. These authors also prove that introducin a baseline keeps the estimate unbiased (have proof in 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 u_t. Introducing a baseline in equation 6 yields the equation:

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(\tau)(R(\tau) - b)] \tag{8}$$

The most basic type of baseline is the global average reward, which keeps track of the average reward across all episodes. Greensmith et al. (2004) derives the optimal constant value baseline. We can also add time dependency to the baseline. It is not optimal to scale the probability of taking an action by the whole sum of rewards. A better idea is, for a given episode, to weigh an action u_t by the reward obatined from time t onwards, otherwise we would be ignoring the Markov property underlying the environment's Markov Decission Process. This changes equation 8 to:

$$\nabla_\theta U(\theta) = \mathbb{E}_{s_t \sim E, u_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(u_t \mid s_t)(\sum_{k=t}^{H-1} R(s_k, u_k) - b)] \tag{9}$$

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. 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 10. 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.

$$b(s_t) = \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \cdots + r_{H-1}] = V_{\pi_\theta}(s_t) \tag{10}$$

Consider a further improvement: the term $\sum_{k=t}^{H-1} R(s_k, u_k)$ can be regarded as an estimate of $Q_{\pi_\theta}(s_t, u_t)$ 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: $\sum_{k=t}^{H-1} \gamma^k R(s_k, u_k)$. However, this still keeps the estimation 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.

$$\begin{aligned} Q^\phi_{\pi_\theta}(s,u) &= \mathbb{E}[r_0 + r_1 + r_2 + \cdots + r_{H-1} \mid s_0 = s, u_0 = u] && (\infty\text{-step look ahead}) \\ &= \mathbb{E}[r_0 + V^\phi_{\pi_\theta}(s_1) \mid s_0 = s, u_0 = u] && (\text{1-step look ahead}) \\ &= \mathbb{E}[r_0 + +r_1 + V^\phi_{\pi_\theta}(s_2) \mid s_0 = s, u_0 = u] && (\text{2-step look ahead}) \end{aligned} \tag{11}$$

Notice how we use 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) (read these 2 papes). Konda and Tsitsiklis (2000) make the key observation that in actor critic methods, the actor parameterization θ and the critic parameterization ϕ should *not* be independent. The choice of critic parameters should be directly prescribed by the choice of the actor parameters.

3.3 Advantage functions

Let the advantage function $A_\pi(s_t, a_t) \in \mathbb{R}$ be the numerical advantage of taking action a_t in state s_t under policy π. The advantage function is often depicted as:

$$A_\pi(t) = A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t) \tag{12}$$

3.4 Trust region optimization, and natural gradient?

3.5 Off-policy policy gradient methods

Off-PAC, The first off-policy policy gradient method introduced by Degris et al. (2012) used importance sampling techniques to weigh the actor gradient update against the behavioural policy being used (look this up on paper). They also used eligibility traces for a critic with linear function approximator, similar to a TD(λ). Reply buffer, introduced in Lin (1993), has seen a lot of use recently (Mnih et al., 2013, 2016).

4 Learning Environments

- OpenAI Gym (Brockman et al., 2016)
- Real time strategy games have always been a famous testbed for many reinforcement learning algorithms. Talk about Broodwar (Heinermann, 2009) and sc2le (Vinyals et al., 2017) for Starcraft. (Which one was this again?) (Synnaeve et al., 2016). Followed by microRTS (**?**), finishing by saying that (**?**) Creates a unifying and highly customizable RTS environment that allows for POMDP with variyng degrees of partial observability. Compare simulation speed.
  - Real time strategy games are really appealing for reinforcement scenarios because a good agent should be able to perform effective search and meaningful planning, over environments with enormous branching factors (Soemers, 2014)(Wender and Watson, 2012).
- Arcade Learning Environment (ALE) (Bellemare et al., 2015)
- VizDoom (Kempka et al., 2017)
- Unity ML-agents (Unity, 2017). Currently only performance boost of 100x.

5 Appendix

subfiles

Take equation 4 from Section **??**, representing the utility of a policy π_θ parameterized by a D-dimensional real valued parameter vector $\theta \in \mathbb{R}^D$

$$U(\theta) = \sum_\tau P(\tau;\theta)R(\tau) \tag{13}$$

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:

$$\begin{aligned}
\nabla_\theta U(\theta) &= \nabla_\theta \sum_\tau P(\tau;\theta)R(\tau) \\
&= \sum_\tau \nabla_\theta P(\tau;\theta)R(\tau) && \text{(Move gradient operator inside sum)} \\
&= \sum_\tau \nabla_\theta \frac{P(\tau;\theta)}{P(\tau;\theta)} P(\tau;\theta)R(\tau) && \text{(Multiply by}\frac{P(\tau;\theta)}{P(\tau;\theta)}) \\
&= \sum_\tau P(\tau;\theta)\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} R(\tau) && \text{(Rearrange)} \\
&= \sum_\tau P(\tau;\theta)\nabla_\theta \log P(\tau;\theta)R(\tau) && \text{(Note:}\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = \nabla_\theta \log P(\tau;\theta)) \\
\nabla_\theta U(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau;\theta)R(\tau)] && (\mathbb{E}[f(x)] = \sum_x x f(x))
\end{aligned} \tag{14}$$

$$\tag{15}$$

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:

$$P(\tau;\theta) = \Pi_{t=0}^{H} \underbrace{P(s_{t+1}|s_t,u_t)}_{\text{dynamics models}} \underbrace{\pi_\theta(u_t|s_t)}_{\text{policy}} \tag{16}$$

From here we can calculate the term $\nabla_\theta \log P(\tau;\theta)$ present in equation 14

$$
\begin{aligned}
\nabla_\theta \log P(\tau;\theta) &= \nabla_\theta \log[\Pi_{t=0}^{H} P(s_{t+1}|s_t,u_t)\pi_\theta(u_t|s_t)] \\
&= \nabla_\theta[(\sum_{t=0}^{H} \log P(s_{t+1}|s_t,u_t)) + (\sum_{t=0}^{H} \log \pi_\theta(u_t|s_t))] \\
&= \sum_{t=0}^{H} \underbrace{\nabla_\theta \log \pi_\theta(u_t|s_t)}_{\text{no dynamics required!}}
\end{aligned} \tag{17}
$$

Plugging the result of equation 17 into equation 14 we obtain the following equation for the gradient of the utility function w.r.t to parameter vector θ:

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)] \tag{18}$$

We can compute an empirical approximation of that expresion by taking m sample trajectories (or paths) under the policy π_θ. This works even if the reward function R is unkown and/or discontinuous. This works in discrete state spaces. The likelihood ratio changes the probability of experienced paths. That is, the probability of sampling trajectories. Thus we use a Monte Carlo approach to approximate the gradient of the utility of π_θ: (REPHRASE) Which, if we plug into the original equation, we get:

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m}\sum_{i=0}^{m} \nabla_\theta \log P(\tau^{(i)};\theta) R(\tau^{(i)}) \tag{19}$$

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m}\sum_{i=0}^{m}\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})(\sum_{k=0}^{H} R(s_t^{(i)}, u_t^{(i)})) \tag{20}$$

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 $\nabla_\theta V_{\pi_\theta}(s_0)$.

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.

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

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.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. pages 1–4.

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.

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

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).*

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

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.

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

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.

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.

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., 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.

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.

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.*

Unity (2017). Unity Machine Learning Agents.

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.