Reinforcement Learning: A Literature Survey

Matthew Chapman

January 6, 2021

1 Introduction

Reinforcement learning is the class of problems concerned with an agent learning behaviour through trial-and-error interactions with a dynamic environment [4].

An example is an aspiring tightrope artist (the agent) learns to walk from one end of a rope to another without falling (the behaviour) by repeatedly correcting their balance (the trial-and-error interactions) whenever the rope wobbles beneath them (the dynamic environment).

In this literature survey we:

1. describe the problems we want to solve with reinforcement learning, and explain why they are interesting to solve;
2. describe historical and current work in the field, including the kinds of problems solved and the approaches; as well as,
3. state what we will be using in our own project.

Much of the environments mentioned in this literature survey are of games, as they (board games in particular) are ideal testing grounds for exploring concepts and approaches in reinforcement learning and artificial intelligence [11].

2 Motivation

Reinforcement learning agents exhibit behaviour similar to intelligence, with intelligence defined as "adaptation with insufficient knowledge and resources" [12]. It is also said that humans and other animals use reinforcement learning to understand their environment and generalise past experience to new situations [7]. To study reinforcement learning, therefore, is to study a small area of exhibiting human intelligence in machines, and solving reinforcement learning problems, such as those done in robotics or chip design, can lead to developments that improve quality of life and ultimately contribute to the search for AGI [3, 5].

In the following section, we start off by introducing constraints to restrict ourselves to "simple" environments with well-defined rewards that are easier to solve mathematically.

3 Markov Decision Processes

To formalise this idea, we phrase the process of "learning" in terms of an optimisation problem. The framework in which this is done is that of Markov Decision Processes (MDPs).

The Markov structure means that

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2, \ldots, S_t),$$

which is that the next state only depends on the current state of the environment the agent is in.

An example of an environment which possesses the Markov property is a Chess board: the next move a player can make depends only on the current board state, disregarding how the game was played up to that point.

This is important because we do not need to store any history, making it easier for the agent to determine an optimal behaviour. In fact, given perfect information, there are exactly solvable equations for the optimal solution [].

Together, the agent and MDP give rise to a trajectory like this:

1. At time t, the agent senses the state of its environment, S_t.
2. The agent selects an action, A_t, which affects the environment.
3. The environment is changed, now at time $t + 1$, and returns a reward signal, R_{t+1}, to the agent, and the process is repeated.

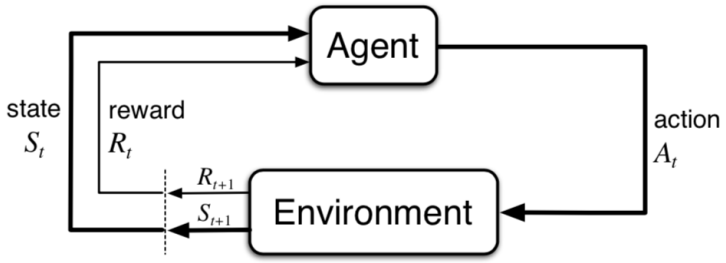

Figure 3.1: The agent–environment interaction in a Markov decision process.

Figure 1: The agent-environment interaction in a Markov decision process [10].

The agent selects actions dictated by its policy, π, and the goal of the agent is to maximise the total reward it receives from the environment over time. A policy can be deterministic, such as to always turn left at a crossroad, or stochastic, such as to either turn left, right, or carry on forward at a crossroad each with probability $\frac{1}{3}$. An example of a complex policy is the output of a neural network which takes in the state of the environment and returns the probabilities of taking each action, with higher probabilities indicating better

actions. (The policy is the resulting table of probabilities, rather than the policy network itself.)

To maximise the total reward over time, the agent must consider more than just immediate rewards. The **return**

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

is the sum of (discounted) future rewards, where $\gamma \in [0, 1]$ is the discount rate. This allows us to value states by a **value function**

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_T | S_t = s \right],$$

which is the expected return when starting in s and following behaviour, or policy, π thereafter, and actions by an **action-value function**

$$q_\pi(s, a) \doteq \mathbb{E}_\pi [G_t | S_t = s, A_t = a],$$

which is the expected return starting from s, taking the action a, and thereafter following policy π.

The best possible policy under these assumptions is given by the solution, q_* to the Bellman optimality equation,

$$q_*(s, a) = \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a \right].$$

However, the solution relies on three assumptions that in reality can rarely be achieved:

1. the dynamics of the environment are accurately known;
2. computational resources are sufficient to complete the calculation; and,
3. the states have the Markov property.

Usable RL methods provide ways of approximating solutions to q_* even when lacking complete information or having restricted computational resources. We will consider such methods in the next sections.

4 Dynamic Programming

DISCUSS DP in the context of some papers that use it

Dynamic programming (DP) refers to the collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP. Even in the case of tasks with continuous state and action spaces, the spaces can be discretised and finite-state DP methods applied to obtain approximate solutions.

DP iteratively evaluates then improves a policy based on the value function of each state; therefore, an imperfect model, such as one where a state has a negative reward despite it being positive in reality, will propagate the error and cause the algorithm to converge to a "wrong" optimal policy.

Iterative policy evaluation:

1. Initial approximation, v_0, is chosen arbitrarily.
2. Each successive approximation is obtained by using the Bellman equation for v_π as an update rule.

Policy improvement: the process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy

Value iteration

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma v_k(s')\right], \qquad \text{for all } s \in \mathcal{S} \end{aligned}$$

DP methods involve operations over the entire state set of the MDP. If the state set is very large, like in Backgammon, then each set is expensive. Therefore, DP is not useful for very big state spaces. In order to approximate things, Monte Carlo methods can be employed, which is covered in the next section.

There also exists *Asynchronous* DP algorithms, which update the value of states in any order whatsoever, using whatever values of other states happen to be available.

Policy iteration consists of making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). The result is convergence to the optimal value function and an optimal policy.

5 Monte Carlo Methods

To estimate the value of a state from experience is to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This is the basis of Monte Carlo (MC) methods for learning the state-value function for a given policy.

Defining a visit to s as each occurrence of state s in an episode, there are two main MC methods:

First-visit MC method: estimates $v_\pi(s)$ as the average of the returns following first visits to s.

Every-visit MC method: averages the returns following all visits to s.

Both methods converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity.

There are three advantages of Monte Carlo methods over DP methods:

- The computational expense of estimating the value of a single state is independent of the number of states;
- The ability to learn from actual experience; and,
- The ability to learn from simulated experience.

If a model is not available, then it is particularly useful to estimate *action* values rather than *state* values. With a model, state values alone are sufficient to determine a policy. One of our primary goals for Monte Carlo methods is to estimate q_*. To achieve this, we first consider the policy evaluation problem for action values.

The policy evaluation problem for action values is to estimate $q_\pi(s, a)$. The MC methods for this are essentially the same as just presented for state values, replacing state with state-action pair.

The complication is that many state-action pairs may never be visited. To compare alternatives we need to estimate the value of *all* the actions from each state. One way to do this is by specifying that the episodes *start in a state-action pair*, and that every pair has a non-zero probability of being selected at the start.

Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

Since the assumption of exploring starts is unlikely, the only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches:

On-policy methods: evaluate or improve the policy that is used to make decisions.

Off-policy methods evaluate or improve a policy different from that used to generate the data.

All learning control methods face a dilemma: they seek to learn action values conditional on subsequent *optimal* behaviour, but they need to behave non-optimally in order to explore all actions (to *find* the optimal actions). The on-policy approach is a compromise — it learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behaviour.

Target policy: the policy being learned

Behaviour policy: the policy used to generate behaviour

Learning is from data "off" the target policy, and the overall process is termed *off-policy* learning.

Off-policy methods utilise *importance sampling*

Importance sampling: technique for estimating expected values under one distribution given samples from another.

Apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behaviour policies, called the *importance-sampling ratio*.

The importance-sampling ratio depends only on the two policies and the sequence, not on the MDP.

There are two types of importance sampling: ordinary, and weighted.

Ordinary importance sampling is unbiased whereas weighted importance sampling is biased (though the bias converges asymptotically to zero).

Monte Carlo prediction methods can be implemented incrementally, on an episode-by-episode basis. In Monte Carlo methods we average *returns*.

An advantage is that the target policy may be greedy, while the behaviour policy can continue to sample all possible actions.

These methods follow the behaviour policy while learning about and improving the target policy.

6 Temporal-Difference Learning

Temporal-difference (TD) learning gets the best of both DP methods and MC methods. Like MC methods, TD methods can learn directly from raw experience without requiring a model of the environment (it is model-free), or iterating over every state. Also like DP methods, TD methods can update their estimates without requiring the final outcome.

The main difference from MC methods is that we learn (update the value function) from incomplete episodes. Instead, we take a partial trajectory and estimate how much reward you think you'll get in place of the actual return (bootstrapping).

In TD(0) (the simplest version of TD-learning), we update our value function $V(S_t)$ towards an estimate of the return:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)),$$

where the TD error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$.

7 Function Approximation

For environments where there are too many states or actions to fit into memory, such as Backgammon with 10^{20} states or Go with 10^{170}, which are too slow to process, we must estimate the value function:

$$\hat{v}(s, \mathbf{w}) \simeq v_\pi(S).$$

For control, we'd do:

$$\hat{q}(S, A, \mathbf{w}) \simeq q_\pi(S, A).$$

Having replaced the value function with a function approximator, how do you train it?

8 Stochastic-gradient Methods

Stochastic-gradient descent (SDG) methods are learning methods for function approximation in value prediction. They are well suited to online reinforcement learning (update our estimate of the value function at every step without waiting until the episode finishes), where data becomes available in sequential order and is used to update the best predictor at each step [7].

In gradient-descent methods, the *approximate value function* $\hat{v}(s, \mathbf{w})$ is a differentiable function of a *weight vector* $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)^T$ for all $s \in \mathcal{S}$. $\mathbf{w}$ gets updated at each discrete $t = 0, 1, 2, 3, \ldots$ with each new observation, and $\mathbf{w}_t$ denotes the weight vector at each step. We try to minimise the *Mean Squared Value Error*, denoted $\overline{VE}$:

$$\overline{VE} \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right].$$

SDG methods minimise error on the observed samples by adjusting the weight vector after each vector by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2} \alpha \nabla_{\mathbf{w}} \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \\ &= \mathbf{w}_t + \alpha (v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}_t), \end{aligned}$$

where α is a positive step-size parameter.

In order to compute the ground-truth value function, $v_\pi(S_t)$, we substitute it with a target.

To learn the optimal behaviour so we have our agents learn the best way to play a game, we need the action-value function, which we approximate exactly the same as for the value function, with $\hat{q}(S, A, \mathbf{w}) \simeq q_\pi(S, A)$

Since incremental methods (learning from each experience) are not sample efficient and lead to rapidly forgetting rare experiences that would be useful alter on, we introduce a experience replay, or replay buffer. This stores experiences (such as the last 50k experiences) and reuses them. These can be sampled uniformly or prioritised to replay important transitions more frequently.

9 Deep Reinforcement Learning

Deep reinforcement learning is the field of research that combines reinforcement learning and deep learning. It has allowed machines to solve more complex decision-making tasks than ever before, due to its usefulness in problems with high dimensional state-space and/or low prior knowledge [2] For example, a deep RL agent can learn from inputs made up of pixels to play Atari 2600 games by using a deep convolutional neural network to approximate the optimal action-value function [7]. Deep RL comes with its own challenges, and so a variety of deep RL algorithms have been developed.

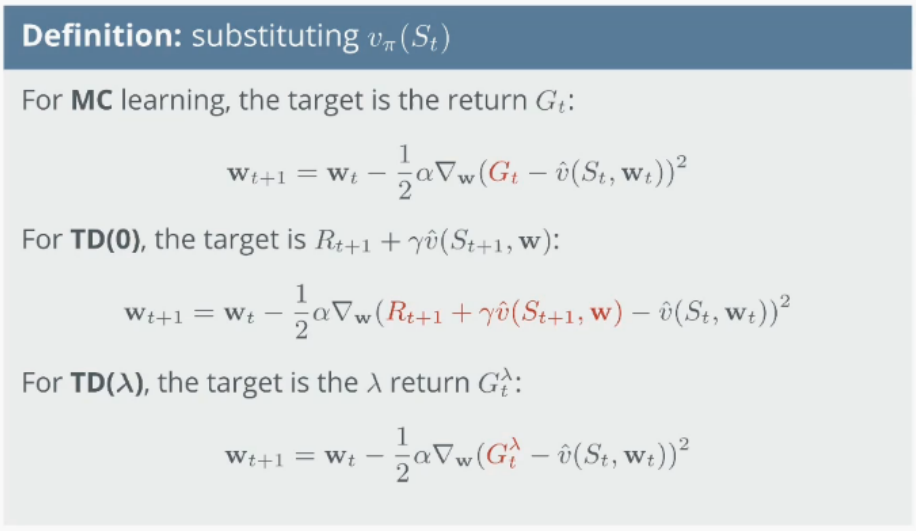

Figure 2: Slide taken from Chris Wilcock's presentation on function approximation.

The first branching point in the tree is to decide whether an RL algorithm is model-free or model-based. The algorithm is model-based if the agent has access to (or learns) a model of the environment; that is, there is a function which predicts state transitions and rewards. Otherwise, it is model-free.

Model-free methods are then divided into two families: policy optimisation, and Q-learning.

10 Policy Optimisation

Policy optimisation methods represent a policy explicitly as $\pi_\theta(a|s)$ and optimise the parameters θ either directly by gradient ascent on $J(\pi_\theta)$, the expected return when the agent acts according to it, or indirectly, by maximising local approximations of $J(\pi_\theta)$. They also involve learning an approximator for the on-policy value function, which gets used in figuring out how to update the policy [8].

10.1 Intro To Policy Optimisation

Consider a stochastic, parametrised policy, π_θ. We aim to maximise the expected return $J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$, where $R(\tau) = \sum_{t=0}^{T} r_t$ gives the finite-horizon undiscounted return, the sum of rewards obtained in a fixed window of steps. The mechanism policy gradient algorithms use to optimise the policy is gradient ascent, e.g.

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta)|_{\theta_k},$$

Definition: substituting $q_\pi(S_t, A_t)$

For **MC** learning, the target is the return G_t:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(G_t - \hat{q}(S_t, A_t, \mathbf{w}_t))^2$$

For **TD(0)**, the target is $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}_t))^2$$

For **TD(λ)**, the target is the λ return:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t))^2$$

Figure 3: Slide taken from Chris Wilcock's presentation on function approximation.

where $\nabla_\theta J(\pi_\theta)$ is called the policy gradient.

To use this function, we need an expression for the policy gradient which we can numerically compute. This involves two steps:

1. deriving the analytical gradient of policy performance, which has the form of an expected value, and then
2. forming a sample estimate of that expected value, which can be computed with data from a finite number of agent-environment interaction steps.

The simplest expression for the policy gradient, derived here [], is

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right].$$

This is an expected which we can estimate with a sample mean

$$\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau),$$

where $|\mathcal{D}| = \{\tau_i\}_{i=1,\ldots,N}$ is the number of trajectories in $\mathcal{D}$ (here, N).

Taking a step with this gradient, however, pushes up the log-probabilities of each action in proportion to the sum of *all rewards ever obtained*. We want to only reinforce actions based on rewards obtained after they are taken, by

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T}R(s_{t'}, a_{t'}, s_{t'+1})\right].$$

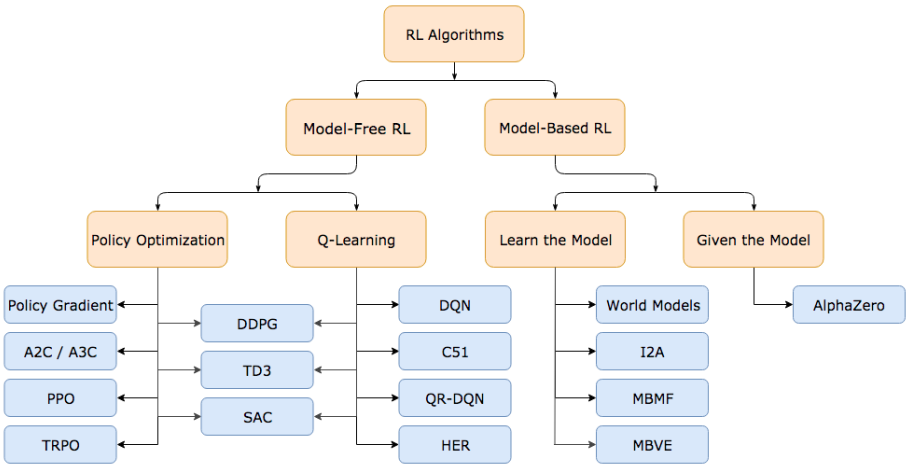

Figure 4: A taxonomy of deep reinforcement learning algorithms [8].

Furthermore, to reduce variance in the sample estimate for the policy gradient (resulting in faster and more stable learning), we can subtract the on-policy value function $V^{\pi}(s)$ (which cannot be computed exactly, so has to be approximated), which gives the expected return if we start in state s and always act according to policy $/pi$, from the expression without changing it in expectation by the GLP lemma:

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{T} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) \sum_{t'=t}^{T} R\left(s_{t'}, a_{t'}, s_{t'+1} - b(s_t)\right) \right].$$

10.2 Vanilla Policy Gradient

The key idea of policy gradients is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy.

Vanilla policy gradient (VPG) is an on-policy algorithm that can be used for environments with either discrete or continuous action spaces. The method is susceptible to the policy getting trapped in local optima.

10.3 Trust Region Policy Optimisation

Trust region policy optimisation (TRPO) updates policies by taking the largest step possible to improve performance, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of KL-Divergence, a measure of distance between probability distributions.

The theoretical TRPO update is:

$$\theta_{k+1} = arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \text{ s.t. } \overline{D}_{KL}(\theta||\theta_k) \leq \delta,$$

where

$$\mathcal{L}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right],$$

and

$$\overline{D}_{KL}(\theta||\theta_k) = \mathbb{E}_{s\sim\pi_{\theta_k}} \left[D_{KL}(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)) \right].$$

In practice, the objective and constraint are expanded around θ_k, resulting in an approximate optimisation problem. The approximate problem can then be analytically solved by the methods of Lagrangian duality; in addition, TRPO modifies the update rule by adding a backtracking line search.

10.4 Proximal Policy Optimisation

Proximal policy optimization (PPO) algorithms alternate between sampling data through interaction with the environment, and optimising a "surrogate" objective function using stochastic gradient ascent. Unlike standard policy gradient methods, multiple epochs of mini-batch updates are possible instead of performing one gradient update per data sample [9].

Similar to TRPO, PPO tries to take the biggest possible improvement step on a policy using the data we currently have while avoiding performance collapse. PPO is a family of first-order methods that are significantly simpler to implement, and perform at least as well as TRPO.

The PPO variant we consider is PPO-Clip: it does not have a KL-divergence term in the objective and does not have a constraint at all; instead, it relies on specialised clipping in the objective function to remove incentives for the new policy to get far fro the old policy.

PPO-clip updates policies via

$$\theta_{k+1} = arg\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_k}} \left[L(s, a, \theta_k, \theta) \right],$$

typically taking multiple steps of (minibatch) SDG to maximise the objective. L is given by

$$L(s, a, \theta_k, \theta) = min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right),$$

in which $/epsilon$ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old. Gradient clipping is a technique to prevent exploding gradients, such as rescaling gradients so that their norm is at most a particular value.

A simplified version is as follows:

$$L(s, a, \theta_k, \theta) = min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), g(\epsilon, A^{\pi_{\theta_k}}(s,a)) \right),$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0 \\ (1-\epsilon)A & \text{if } A < 0. \end{cases}$$

To prevent ending up with a new policy which is too far from the old policy, we can stop taking the gradient steps if the mean KL-divergence of the new policy form the old grows beyond a threshold (early stopping). The method is susceptible to policies being trapped in local optima.

11 Q-Learning

Q-learning is a value-based class of algorithms that aim to build a value function, which subsequently lets us define a policy.

Q-learning keeps a lookup table of values $Q(s, a)$ with one entry for every state-action pair. In order to learn the optimal Q-value function, the Q-learning algorithm makes use of the Bellman equation for the Q-value function whose unique solution is $Q^*(s, a)$ [2].

However, convergence to the optimal value function assumes that the state-action pairs are represented discretely, and all actions are repeatedly sampled in all states; this is often inapplicable with a high-dimensional state-action space.

12 Long Short-Term Memory (LSTM) [1]

Among the more important tasks for RL are tasks where part of the state of the environment is *hidden* from the agent. The agent now not only needs to learn the mapping from environmental states to actions, for optimal performance it needs to determine which environmental state it is in as well.

Long-term dependencies allow the agent to distinguish between identical states in its environment, such as T-junctions in a maze, by remembering observations or actions before reaching the state. LSTM is a solution to learning long-term dependencies in time series data.

LSTM is a RNN architecture that solves the problem that conventional RNN algorithms have with errors propagating back in time tending to either vanish or blow up.

RNNs can be applied to RL tasks by letting it learn a model of the environment. LSTM architecture would allow the predictions of observations and rewards to depend on information from long ago. The model-based system could then learn the mapping from environmental states to actions using standard techniques such as Q-learning. Alternatively, the RNN could directly approximate the value function of a RL algorithm.

13 Our Project

In this project, we want to look at deep reinforcement learning algorithms, implement them, and analyse their performance in those environments provided by OpenAI Gym.

Visual transfer [6]

References

[1] Bram Bakker. "Reinforcement Learning with Long Short-Term Memory". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. NIPS'01. Vancouver, British Columbia, Canada: MIT Press, 2001, pp. 1475–1482.

[2] Vincent François-Lavet et al. "An Introduction to Deep Reinforcement Learning". In: *CoRR* abs/1811.12560 (2018). arXiv: 1811.12560. URL: http://arxiv.org/abs/1811.12560.

[3] Engin Ipek et al. "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach". In: *2008 International Symposium on Computer Architecture*. IEEE, June 2008. DOI: 10.1109/isca.2008.21. URL: https://doi.org/10.1109/isca.2008.21.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* 4 (May 1996), pp. 237–285. DOI: 10.1613/jair.301. URL: https://doi.org/10.1613/jair.301.

[5] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (Aug. 2013), pp. 1238–1274. DOI: 10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721.

[6] Akshita Mittel, Sowmya P. Munukutla, and Himanshi Yadav. "Visual Transfer between Atari Games using Competitive Reinforcement Learning". In: *CoRR* abs/1809.00397 (2018). arXiv: 1809.00397. URL: http://arxiv.org/abs/1809.00397.

[7] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.

[8] OpenAI. *Part 2: Kinds of RL Algorithms¶*. 2018. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[9] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[11] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (Mar. 1995), pp. 58–68. DOI: 10.1145/203330.203343. URL: https://doi.org/10.1145/203330.203343.

[12] Pei Wang. "On Defining Artificial Intelligence". In: *Journal of Artificial General Intelligence* 10.2 (1Jan. 2019), pp. 1–37. DOI: https://doi.org/10.2478/jagi-2019-0002. URL: https://content.sciendo.com/view/journals/jagi/10/2/article-p1.xml.