Reinforcement Learning Methods

Dan Salter

Supervised by Vladislav Tadic
Level 6
20 Credit Points

May 16, 2021

Acknowledgement of Sources

For all ideas taken from other sources (books, articles, internet), the source of the ideas is mentioned in the main text and fully referenced at the end of the report.

All material which is quoted essentially word-for-word from other sources is given in quotation marks and referenced.

Pictures and diagrams copied from the internet or other sources are labelled with a reference to the web page or book, article etc.

Signed [illegible]

Date 16/5/2021

Contents

1 Introduction

Reinforcement learning is a particular paradigm of machine learning in which we seek to learn an optimal behaviour pattern, so that when presented with a specific situation the most effective action to take in that situation is known. Before we can begin designing a method for solving this particular type of optimisation problem we need to first define a structure that precisely characterises the problem at hand.

1.1 Markov decision processes

Reinforcement learning problems are best characterised by a *Markov Decision Process (MDP)*, in which an **agent** interacts with an **environment** at a series of discrete time steps by observing the environment's state then performing an action within it. Each action has two effects; it:

- prompts an evolution of the environment state, and
- results in a reward signal for the agent.

We can now begin to formally define the components of a Markov decision process. We will be doing so in keeping with the structure described in Reinforcement Learning: An Introduction second edition by Richard S. Sutton and Andrew G. Barto.[15] Let us first define some terminology we will use to discuss the evolution of MDPs over time:

Definition 1.1 (Agent) *The **agent** is the decision maker, responsible for selecting actions with the end goal of maximising the resulting reward signals over an extended period of time.*

Definition 1.2 (Environment) *The **environment** consists of everything outside of the agent; it is the space which the agent can observe and perform actions on.*

Next, we must define the sets we will use to precisely describe the cycle of events that take place in the MDP at each timestep $t = 0, 1, 2, \ldots$:

Definition 1.3 ($\mathcal{S}$) *The set of all **states** the environment can take. $S_t \in \mathcal{S}$ denotes the state at time t.*

Definition 1.4 ($\mathcal{A}(s)$) *The set of all* ***actions*** *the agent can select when the environment is in state $s \in \mathcal{S}$. $A_t \in \mathcal{A}(s)$ denotes the action executed at time t. Note, we will occasionally assume the set of available actions is identical for every state - in this case we will refer to this universal set of actions simply as $\mathcal{A}$*

Definition 1.5 ($\mathcal{R}$) *The set of numerical* ***rewards****, $\mathcal{R} \subseteq \mathbb{R}$, the agent can receive. $R_t \in \mathcal{R}$ denotes the reward received at time t.*

A MDP is initialised at $t = 0$ in a predefined state, $S_0 \in \mathcal{S}$. For each S_t the agent selects an appropriate action, $A_t \in \mathcal{A}(S_t)$. Consequently, the agent receives a reward, $R_{t+1} \in \mathcal{R}$, and the environment state is updated to $S_{t+1} \in \mathcal{S}$. This sequence of action and effect can be illustrated as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{1}$$

A MDP is a stochastic system, in the sense that, following an action the resulting reward and state values are uncertain, therefore they can be represented as random variables (RVs). We now seek to define the probability distributions that predict the values of these RVs, and hence together characterize the system's dynamics. To do so we will first define a joint distribution for the reward and new state that we can that give the individual distributions distributions in terms of.

Crucially MDPs are subject to the *Markov property*, which means that "Future is independent of the past given the present"[3], specifically in terms of the process' state. This is possible since all information relevant to the next action and consequent state transition is encapsulated within the present state. More formally, the next state and any of the states preceding the current state are conditionally independent given the current state. As a result we are able to define the desired distribution of the reward and next state, dependant solely on the current state and chosen action (often referred to as a *state-action pair*), as follows:

Definition 1.6 (Dynamics equation) *For all $s, s' \in \mathcal{S}, r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$,*

$$p(s', r \mid s, a) := \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

The *dynamics equation*, as its name suggests, entirely characterizes the dynamics of our MDP, in the sense that we can use it to compute any queries whatsoever we may have about the environment. Hence we shall now use it to construct definitions for several other useful equations:

Definition 1.7 (State-transition equation) *For all $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}(s)$,*

$$p(s' \mid s, a) := \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a)$$

Definition 1.8 (Expected reward equation) *For all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$,*

$$r(s, a) := \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a)$$

1.2 Setting goals using rewards

In reinforcement learning we will always be trying to find an optimal way of behaving in some real world situation. As the system's engineer we must have a very specific concept of what behaviour we want to train our agent to exhibit. For example we could be looking to teach a bipedal robot how to walk forwards, with an agent controlling the motion of its limbs. Here our desired behaviour is walking forwards as far as possible.

Our only method of guiding the agent's decision making behaviour towards what we believe to be optimal is through a reward scheme, where the agent is rewarded when certain desirable events take place. Similarly we can punish (reward the agent negatively) when an undesirable event takes place. This is effective since the agents' sole desire when choosing an action is to maximise the total amount of rewards it will receive. This means in certain situations forgoing the action that would result in receiving the greatest reward at the following time step, in favour of one that might be more beneficial in the long run.

Definition 1.9 (Return)

$$G_t := R_{t+1} + R_{t+2} + ... + R_T$$

where T is the final timestep.

Now we can formalise the concept of an agent choosing actions that ”maximise total rewards” as an agent choosing actions such that the *expected return* at time t is maximised. Notice, this definition assumes the existence of a finite T, at which point the environment transitions to the *terminal state* before resetting to some starting state. We call these finite subsequences *episodes*, and any task that features them is knows as an *episodic task*.

Returning to our bipedal robot example, we could choose to reward the agent +1 for each centimeter the robot walks forwards. If we tried to accomplish this by only maximising the immediately resulting reward we might naively choose actions that decrease our potential total rewards. For example the robot might learn to lunge as far as possible with its first step in order to maximise the first reward, but as a result fall to the ground halting any chance of further progress and rewards. Reinforcement learning provides a more measured approach where our agent considers the impact its action will have on its ability to earn rewards in the future by maximising the expected return. Applying this philosophy to our example would result in a robot that takes smaller steps, earning less reward in the first step, but is therefore more stable and able to remain standing. It will walk a greater total distance and earn a much greater total reward in doing so.

There is some nuance to designing an effective reward scheme. We need to ensure we are rewarding the agent for accomplishing *what* we want it to achieve without trying to influence the agent's behaviours by rewarding *how* it achieves that goal. If we try to reward a specific behaviour that we believe will help the agent accomplish its end goal, we run the risk of the agent becoming extremely efficient at exhibiting that behaviour, without actually accomplishing the end goal. For example, if we believe our bipedal robot is dragging its feet and would benefit from raising its feet higher on each step we might erroneously introduce a reward of +1 for each cm it raises its feet. This would result in the agent fully optimising this specific behaviour; after a period of learning we might observe the robot is now leaping as high as possible without ever travelling forwards at all. The agent has maximised expected return by maximising the height of its feet without the need to travel forwards.

There is an alternative formulation for return that introduces the important concept of variable *farsightedness*, which is set by γ, known as the *discount rate* in the following equation:

Definition 1.10 (Discounted Return)

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

where $\gamma \in [0, 1]$ and $T \in \mathbb{Z}$, including the possibility that $T = \infty$ or $\gamma = 1$ but not both.

Setting $\gamma = 1$ means no discount is given, with this argument the above equation defines the same non-discounted return equation as **Definition 1.9**. In this case, for this formulation of the return to be a bounded, we must have finite T. In other words we can only set a discount rate of 1 when maximising this return in an episodic task. If $T = \infty$ it means we are dealing with a *continuing task*, in which case for the infinite sum to be bounded we must set $0 < \gamma < 1$. It is worth noting that there exists an alternative formulation for returns that is undiscounted yet finite in continuing tasks, known as the *differential return*, which is covered in Subsection 6.4. In the case when $\gamma = 0$, the agent is known as *myopic* and is only concerned with maximising the immediate reward.

1.3 Policies and Value Functions

Agents "choose" actions in accordance with a *policy*, which we can define as a probability distribution over actions given states. In practice an agent picks its action by stochastically sampling from this discrete probability distribution. Informally, the chance of an agent selecting each possible action when in a certain state is determined by its policy.

Definition 1.11 (Policy) *For all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$,*

$$\pi(a \mid s) := \Pr\{A_t = a \mid S_t = s\}$$

The end goal of reinforcement learning is to find the *optimal policy*, which we can define as the policy that when followed maximises the return received by the agent. Agents learn this policy iteratively by updating the probability distribution in reflection of their experiences. Before we can formally explain this iterative process we must define the way in which we will evaluate policies, so as to draw comparisons between their effectiveness and select the most favourable. Let us define the *value function* of a state s under a policy π, denoted $v_\pi(s)$, as the expected return when starting in s and following π thereafter:

Definition 1.12 (State-value function for policy π) *For all $s \in \mathcal{S}$*

$$v_\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π.

Similarly, we can define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

Definition 1.13 (Action-value function for policy $\boldsymbol{\pi}$) *For all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$*

$$q_\pi(s, a) := \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

We will now be able to estimate the values of each state and action under a policy from the experience of an agent following that policy. To progress towards finding the optimal policy we desire, we must now define the *optimal state value function* which all optimal policies satisfy:

Definition 1.14 (Optimal state-value function) *For all $s \in \mathcal{S}$,*

$$v_*(s) := \max_\pi v_\pi(s) = v_{\pi_*}(s)$$

where π_* denotes any optimal policy.

Expectedly, we can formulate the *optimal action-value function* in a similar manner:

Definition 1.15 (Optimal action-value function) *For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$,*

$$q_*(s, a) := \max_\pi q_\pi(s, a) = q_{\pi_*}(s, a)$$

where π_* denotes any optimal policy.

If we are able to find these optimal value functions then finding an optimal policy that satisfies them is relatively straightforward. An optimal policy, π_*, is the deterministic policy such that for each state $s \in \mathcal{S}$ we choose the action $a \in \mathcal{A}(s)$ that maximises the value of $q_*(s, a)$:

$$\pi_*(a \mid s) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a \in \mathcal{A}(s)} q_*(s, a) \\ 0 & \textit{otherwise} \end{cases} \tag{2}$$

Hence, in order to find an optimal policy our job should first be to deduce the optimal value function.

There is one last representation of the value functions that will be essential when we come to solving an MDP for the optimal value function. Giving the value of the current state in terms of the value of each possible successor state allows us to construct an iterative algorithm for deducing the optimal value function. This representation is known as the *Bellman expectation equation*, and is derived from the expected return as follows:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a, s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{3}$$

As we can see, the bellman expectation equation is a decomposition of the state-value function into the sum of the immediate reward, R_{t+1}, plus the value of the successor state, $v_\pi(S_{t+1})$ with discount factor γ.[4] We can apply this same decomposition to the optimal value functions, to give *Bellman optimality equations for the state-value function and the action-value function*:

Definition 1.16 (Bellman optimality equation for state-value function)

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \mathbb{E}_\pi[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

Definition 1.17 (Bellman optimality equation for action-value function)

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

1.4 Solving MDPs using dynamic programming

When we say we are *solving an MDP* we mean we are calculating the optimal policy. We can now use our equations to construct an algorithm for solving MDPs for which *a perfect model of the environment is known.* Mathematically speaking, this is when the value of $p(s', r|s, a)$ is known for all $s, s' \in \mathcal{S}, a \in \mathcal{A}(s)$ and $r \in \mathcal{R}$.

Recall equation 3, the Bellman expectation equation; if there is a perfect model of the environment available, we can consider the set of Bellman expectation equations for each $s \in \mathcal{S}$ as a system of $|\mathcal{S}|$ linear equations in $|\mathcal{S}|$ unknowns ($v_\pi(s)$ for each $s \in \mathcal{S}$), which we can then proceed to solve using dynamic programming.

To use dynamic programming we require the knowledge of a perfect model of the MDP environment and initialise the agent to follow some random policy π_0. Our first task is to calculate v_{π_0}, which we are able to do iteratively, a process known as *iterative policy evaluation.* We begin by taking an arbitrary $v_0 \in \mathbb{R}$, then iteratively calculating approximations for the value of each state using the following update rule:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a, s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_k(s')] \end{aligned} \tag{4}$$

For all $s \in \mathcal{S}$. It is clear that $v_k = v_{\pi_0}$ is a fixed point for the update rule, and we also have that $\lim_{k\to\infty} v_k = v_{\pi_0}$. Hence by continuing this process for enough iterations we can accurately approximate the value function over the entire state space.

Our next job is to use our newly calculated value function, v_{π_0}, to determine a new, improved policy, π_1, a process known as *policy improvement.* To do so we first need to state a result that will provide certainty that our policy will always improve, and never regress; that is for all $s \in \mathcal{S}, v_{\pi_k}(s)$ increases monotonically with k.

Theorem 1 (Policy improvement theorem)

If for all $s \in \mathcal{S}$ *we have* $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ *then* $v_{\pi'}(s) \geq v_\pi(s)$

We will perform policy improvement by constructing our new policy π_1 *greedily* from v_{π_0}. This is done by setting the new policy for each state to deterministically select the action that maximises the action-value function for that state:

$$\text{For all } s \in \mathcal{S}, \text{ set } \pi'(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} \, q_\pi(s, a) \tag{5}$$

This construction satisfies the policy improvement theorem, since for all $s \in \mathcal{S}$

$$\begin{aligned} q_\pi(s, \pi'(s) &= q_\pi(s, \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} \, q_\pi(s, a)) \\ &\geq q_\pi(s, v_\pi(s)) \\ &= v_\pi(s) \end{aligned}$$

Hence, by constructing π_1 greedily according to v_{π_0} we can assert for all $s \in (S), v_{\pi_1}(s) \geq v_{\pi_0}(s)$ by the policy improvement theorem.

A process known as *generalized policy iteration (GPI)* alternates between policy evaluation and policy improvement, to achieve a monotonically improving sequence of policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \ldots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_{\pi_*}$$

Where $\xrightarrow{\text{E}}$ denotes policy evaluation and $\xrightarrow{\text{I}}$ denotes policy improvement. It can be shown that this sequence indeed converges to the optimal policy as shown in the diagram above: let π' be the greedy policy for v_π and suppose for all $s \in \mathcal{S}$ we have $v_\pi(s) = v_{\pi'}(s)$. It follows from equation 5 that for all $s \in \mathcal{S}$:

$$\begin{aligned} v_{\pi'} &= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s, A_t = a] \end{aligned}$$

The above equation is identical to the bellman optimality equation, therefore $v_{\pi'} = v_\pi = v_*$ and $\pi' = \pi = \pi*$. Consequently when performing GPI we can be certain that the improved policy will be strictly better than the previous policy, unless both policies are the optimal policy. This gives a strictly improving sequence of policies that is bounded above by the existence of an optimal policy. Hence by the monotone convergence theorem we can be certain that GPI will converge to the optimal policy.

We can develop the process of GPI further by halting the policy evaluation step early, before convergence to the true value function for the policy. Although hating early results in the next policy in the sequence being of a lower quality, we are able to perform each evaluation so much faster that convergence to the optimal policy happens faster overall. In some cases we may even halt policy evaluation after just one iteration, in which case we can express a single unified policy evaluation and improvement step for all $s \in \mathcal{S}$ as follows:

$$\begin{aligned} v_{k+1}(s) &= \max_{a \in \mathcal{A}(s)} \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_k(s')] \end{aligned} \tag{6}$$

This variation of policy iteration that implements a single policy evaluation iteration is known as *value iteration*. For arbitrary v_0, the sequence (v_k) can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

It is key to notice that every dynamic programming technique discussed so far has one crucial limitation in common: each iteration requires calculations over the entire set of states. The computational complexity of the policy iteration algorithm grows exponentially with the number of states, a problem known as the *curse of dimensionality*, hence for large state spaces dynamic programming becomes virtually intractable. This limitation befalls many optimisation algorithms that operate on large datasets.

One method of combating the curse of dimensionality is through a technique known as *asynchronous dynamic programming*. Here, instead of updating the values of all states together in one *sweep*, the values of states are updated in whatever order is considered to be the most efficient. This means some states-values may receive many updates in the time a different state is waiting for a single update. For this algorithm to converge to the optimal policy it is still required that every state receives value updates; so while this technique does open certain MDPs to the possibility of being solved, it does not eliminate the curse of dimensionality altogether.

1.5 Approximating MDP solutions

As we have just seen, some MDPs have too many states to be solved exactly; in some cases even a single value iteration sweep is intractable. However, we are still able to find near optimal policies by approximating the value function. There are many ways to go about this but they share some key concepts in common that we will go over now.

Since finding an exact value function is too computationally expensive, we can instead represent it as a parameterised function. Define $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$ as the approximate value of state s given weight vector $\boldsymbol{w} \in \mathbb{R}^d$, such that $d \ll |S|$. $\boldsymbol{w}$ fully determines the approximate value function, so our goal when approximately solving an MDP is to find the optimal $\boldsymbol{w}$. Since $\boldsymbol{w}$ has far fewer elements than there are states, it is often impossible to find a $\boldsymbol{w}$ that parameterises the exact value function, i.e. for certain MDPs for all $\boldsymbol{w} \in \mathbb{R}^d$, $\hat{v}(s, \boldsymbol{w}) \neq v_\pi(s)$.

Hence, we require some measure of accuracy for value function approximations:

Definition 1.18 (Mean squared value error ($\overline{VE}$))

$$\overline{VE}(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, \boldsymbol{w})]^2$$

where for all $s \in \mathcal{S}$ $\mu(s) \geq 0$ and $\sum_{s \in \mathcal{S}} \mu(s) = 1$.

Here $\mu(s)$ is taken to be the proportion of time that the agent spends in state s, and is used to weight the value error for that state. It essentially determines the importance of the error for each state, implementing the philosophy that we should put the most effort into accurately approximating the value of the states we encounter the most. More formally, $\mu(s)$ is usually taken to be the invariant probability mass function of $p(s' \mid s, a)$, which exists and is unique since the transition matrix, defined $T_{s,s'} = \sum_{a \in \mathcal{A}(s)} p(s' \mid s, a)$, is a stochastic matrix with all positive entries and all row vectors summing to 1.[18]

Now, to find the best approximation of the value function all we need to do is to minimise the mean squared error for $\boldsymbol{w}$. That is, to find the *local optimum* weight vector $\boldsymbol{w}^*$ such that for all $\boldsymbol{w}$ in the neighbourhood of $\boldsymbol{w}^*$, $\overline{VE}(\boldsymbol{w}^*) \leq \overline{VE}(\boldsymbol{w})$.

2 Stochastic gradient descent

2.1 Gradient descent

Generally speaking, gradient descent is a way to minimize an objective function, $f(\boldsymbol{\theta})$, parameterised by a model's parameters, $\boldsymbol{\theta} = \{\theta_1, ..., \theta_d\} \in \mathbb{R}^d$, by updating the parameters in the opposite direction of the gradient of the objective function, $\nabla f(\boldsymbol{\theta})$, with respect to the parameters.[12]

We want to prove that this operation yields a new set of parameters for which our objective function is decreased, in which case it follows that iterating this process would yield a monotonically decreasing sequence:

$$f(\boldsymbol{\theta_0}) \geq f(\boldsymbol{\theta}_1) \geq ... \geq f(\boldsymbol{\theta}^*)$$

Note, if these inequalities hold, the parameters will converge to a *local optimum* parameterisation $\boldsymbol{\theta}^*$. $\boldsymbol{\theta}^*$ is locally optimum in the sense that, for each parameter vector $\boldsymbol{\theta}$ within a neighbourhood of $\boldsymbol{\theta}^*$, we have that $f(\boldsymbol{\theta}^*) \leq f(\boldsymbol{\theta})$. Let us now formalise the gradient descent update rule described above:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha_n \nabla f(\boldsymbol{\theta}_n), \text{ for } n \geq 0 \tag{7}$$

where $\alpha \in \mathbb{R}^+$ is a *step-size parameter*.

To understand how this operation can guarantee convergence to a local optimum, we must first define the vector differential operator, ∇:

Definition 2.1 (Vector differential operator) *let $\boldsymbol{x} = \{x_1, x_2, ..., x_n\}$ and let $\boldsymbol{e}_i$ represent the ith basis vector of the standard n-dimensional basis $\{\boldsymbol{e}_1, ..., \boldsymbol{e}_n\}$, then*

$$\nabla := \sum_{i=1}^{n} \boldsymbol{e}_i^T \frac{\partial}{\partial x_i} = \left(\frac{\partial}{\partial x_1}, ..., \frac{\partial}{\partial x_n}\right)^T$$

When the vector differential operator is applied to a function of a vector, $f(\boldsymbol{\theta})$, as seen in equation 2.1, it is called the *gradient operator* and returns the vector with direction equal to the direction in which the function increases most quickly from $\boldsymbol{\theta}$ and magnitude equal to the rate of increase in that direction:

$$\nabla f(\boldsymbol{\theta}) = \left(\frac{\partial f(\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_2}, ..., \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_d}\right)^T \tag{8}$$

It should then be clear that $-\nabla f(\boldsymbol{\theta})$ is the vector pointing in the direction in which $f(\boldsymbol{\theta})$ decreases most quickly from $\boldsymbol{\theta}$. Hence we can choose some $\alpha \geq 0$ such that:

$$f(\boldsymbol{\theta}) \geq f(\boldsymbol{\theta} - \alpha \nabla f(\boldsymbol{\theta}))$$

Therefore, substituting in equation 2.1, we have shown that, given an initial minimum parameter guess, $\boldsymbol{\theta}_0$, and a sufficiently small α_n for all $n \in \mathbb{N}$:

$$f(\boldsymbol{\theta}_{\mathbf{0}}) \geq f(\boldsymbol{\theta}_1) \geq f(\boldsymbol{\theta}_2) \geq ...$$

Therefore $(\boldsymbol{\theta}_n)_{n \in \mathbb{N}}$ is a monotonically decreasing sequence. Furthermore, with certain assumptions on the objective function (e.g. f convex and ∇f Lipschitz) we can certainly say that this sequence converges to some $f(\boldsymbol{\theta}^*)$ for which $\boldsymbol{\theta}^*$ is a local optimum.

2.2 Learning rate scheduling

Notice, above we concluded that gradient descent is guaranteed to converge to a local minimum given the selection of a sufficiently small learning rate α_n at each iterative step. Scheduling refers to the way in which the learning rate is set for each iteration, so clearly pre-defining a valid schedule is imperative to ensuring convergence. Common learning rate schedules include time-based decay, step decay and exponential decay.[8]

We can now explore some properties of the sequence of learning rates $(\alpha_n)_{n \in \mathbb{N}}$ that are required to ensure convergence. An easy way to guarantee convergence is by ensuring $(\alpha_n)_{n \in \mathbb{N}}$ satisfies the following criteria laid out in the Robbins-Monro algorithm[11]:

$$\sum_{n=1}^{\infty} \alpha_n = \infty \tag{9a}$$

$$\sum_{n=1}^{\infty} \alpha_n^2 < \infty \tag{9b}$$

The first requirement ensures that, through gradient descent, our parameters $\boldsymbol{\theta}_n$ are able to change enough to take any possible value. The second requirement ensures that eventually the steps become small enough for $\boldsymbol{\theta}_n$ to converge.

One example of a sequence that satisfies both of these criteria is the sequence $\alpha_n = 1/n$, that when summed forms the harmonic series:

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ...$$

The divergence of the harmonic series is a well established result,[2] hence the first criterion holds. Moreover, the convergence of p-series, of the form:

$$\sum_{n=1}^{\infty} \frac{1}{n^p}$$

for some $p > 1$ can also easily be proved.[7] Hence the second criterion also holds, and we can conclude that gradient descent executed with a scheduled learning rate $(1/n)_{n\in\mathbb{N}}$ converges to a local optimum.

2.3 Stochastic vs batch gradient descent

So far we have established that the goal of gradient descent is to find the parameterisation $\boldsymbol{\theta}$ that minimises the objective function $f(\boldsymbol{\theta})$, but it remains to specify exactly how we apply the objective function to a training dataset with respect to the parameters.

Definition 2.2 *Let X be a training dataset such that $|X|= m$, then*

$$f(\boldsymbol{\theta}) := \frac{1}{m} \sum_{x\in X} f(x, \boldsymbol{\theta})$$

Here $f(x, \boldsymbol{\theta})$ is the objective function, parameterised by $\boldsymbol{\theta}$, applied to the data point $x \in X$. Applying this definition to equation 2.1 we get:

$$\begin{aligned} \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \alpha_n \nabla f(\boldsymbol{\theta}_n) \\ &= \boldsymbol{\theta}_n - \alpha_n \nabla \frac{1}{m} \sum_{x\in X_n} f(x, \boldsymbol{\theta}_n) \\ &= \boldsymbol{\theta}_n - \frac{\alpha_n}{m} \sum_{x\in X_n} \nabla f(x, \boldsymbol{\theta}_n) \end{aligned} \tag{10}$$

for $n \geq 0$, where X_n denotes the dataset at the nth iteration. This formulation is known as *batch gradient descent*, in which we calculate the gradient of the cost function, with respect to its parameters, to be the average of the gradients for the entire training set at each time step, known as a *training epoch*. This means that for each training epoch we must calculate the gradient of the objective function m times. Clearly the time and memory required for this calculation are linearly proportional to m, hence as m grows these factors will quickly limit the efficiency of the algorithm. This prevents batch gradient descent from being a viable *online learning method* for large data sets.

In contrast, *stochastic gradient descent (SGD)* performs a parameter update by approximating the true gradient with the gradient calculated for a single, randomly selected data point $x \in X_n$:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha_n \nabla f(x, \boldsymbol{\theta}_n) \tag{11}$$

The time and memory required for this method is constant regardless of the size of the training dataset, hence, especially for large m, a training epoch can be completed much faster, solving the problems we faced using batch gradient descent and making SGD a viable option for function approximation in online learning. Furthermore, it can be shown that when the learning rate is correctly scheduled, if some other mild assumptions on the objective function are satisfied then SGD converges *almost surely* (with probability 1) to a global optimum, and otherwise converges almost certainly to a local minimum.[13]

2.4 SGD for reinforcement learning

SGD is an especially useful technique in the context of reinforcement learning since, as discussed in section 1.5, to solve MDPs with an exceedingly large state space we need to approximate the value function by minimising an error function; SGD is designed to solve this exact type of error minimisation problem. To find an approximate solution to an MDP we first set the objective function to be the mean squared value error:

$$\overline{VE}(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \boldsymbol{w})]^2$$

$\overline{VE}$ is is a function of the weight vector $\boldsymbol{w} \in \mathbb{R}^d$ where $d \ll |\mathcal{S}|$, so here the goal of SGD to find $\boldsymbol{w}$ minimising the error function. This mean squared value error is a weighted average of all states, whereas SGD requires the calculation of the gradient for a single state, so before we can apply SGD we must define the squared value error for an individual state:

Definition 2.3 *Let $s \in \mathcal{S}$, then define*

$$\overline{VE}(s, \boldsymbol{w}) := [v_\pi(s) - \hat{v}(s, \boldsymbol{w})]^2$$

We can then substitute this definition, applied to the state at time t, $S_t \in \mathcal{S}$, into the parameter update rule outlined in equation 2.3 to acquire the following iterative formula:

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \alpha_t \nabla \overline{VE}(S_t, \boldsymbol{w}_t) \\ &= \boldsymbol{w}_t - \alpha_t \nabla [v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)]^2 \\ &= \boldsymbol{w}_t - 2\alpha_t [v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)] \nabla [v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)] \\ &= \boldsymbol{w}_t + 2\alpha_t [v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)] \nabla \hat{v}(S_t, \boldsymbol{w}_t) \\ &= \boldsymbol{w}_t + \beta_t [v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)] \nabla \hat{v}(S_t, \boldsymbol{w}_t) \end{aligned} \tag{12}$$

We find line 3 by applying the chain rule and the final line by letting $\beta_t = 2\alpha_t$ for each timestep t. So, by our previous analysis of the convergence of SGD we can conclude that, given S_t is randomly selected from $\mathcal{S}$, the learning rate β is scheduled correctly and the mild assumptions on the objective function $\overline{VE}$ that were mentioned earlier hold, $\boldsymbol{w}_t$ will almost certainly converge to a global optimum $\boldsymbol{w}^*$. If all the aforementioned conditions apart from the mild assumptions on $\overline{VE}$ hold, then convergence of $\boldsymbol{w}_t$ to a local optimum $\boldsymbol{w}^*$ is almost certain.

SGD is an example of a stochastic optimisation method; finding the optimal parameters is based on the premise that, for data points that share some common features, the gradient of the objective function at these data points will be related in some way, a property characteristic of a stochastic optimisation method. More specifically to MDPs, changing the weights vector so as to reduce the squared value error for one state will also presumably reduce the error for other similar states.

2.5 SGD with an approximate target value function

Notice in Definition 2.3 we have defined the squared value error for the state $s \in \mathcal{S}$ as the squared difference of a target function $v_\pi(s)$ and our approximate function $\hat{v}(s, \boldsymbol{w})$. This formulation relies on us already knowing the exact value function for policy π: this will clearly not be the case in practice since our motivation for performing SGD is the intractability of exactly calculating the value function for large state spaces. In reality we will be using an approximation of $v_\pi(s)$, that we have estimated through our agent's sustained interactions with the environment, as our target function for our parameterised approximation $\hat{v}(s, \boldsymbol{w})$. So for this discussion let $U_t \in \mathbb{R}$ be a possibly random approximation, such that $U_t \approx v_\pi(s)$. Substituting U_t into equation 12 yields the following iterative SGD method for state value prediction:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \beta_t[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w}_t) \tag{13}$$

If U_t is an unbiased estimate, that is, if $\mathbb{E}[U_t|S_t = s] = v_\pi(s)$ for each t, then given the correct scheduling of the step-size parameter *beta*, $\boldsymbol{w}_t$ will almost certainly converge to a local optimum.

There are a multitude of ways in which we can approximate the value function, a handful of which will be covered in the succeeding sections. For example, Monte Carlo methods produce an unbiased estimate through interaction with the environment. Other methods apply a technique known as *bootstrapping* that it is worth discussing now, due to the effect it has on the convergence of SGD. Bootstrapping uses a previous estimate to inform the updated estimate: for example, the dynamic programming update rule:

$$\hat{v}_{t+1}(S_t, \boldsymbol{w}_{t+1}) = \sum_a \pi(a, S_t) \sum_{s', r} p(s', r|S_t, a)[r + \gamma\hat{v}_t(s', \boldsymbol{w}_t)]$$

gives the new estimate in terms of the old estimate. Hence, the new update depends on the old weight vector $\boldsymbol{w}_t$, and is therefore a biased estimator. U_t's independence from $\boldsymbol{w}_t$ is essential to the derivation of equation 12, so it is not the case that this biased method will almost certainly converge to a local optimum. Although the convergence of these so called *semi-gradient* methods is less robust than using an alternative unbiased estimate, they do still converge for the majority of cases, in these cases often in less time, and offer some further advantages, such as the ability to learn online. Conversely Monte Carlo methods need to wait until the end of an episode before being able to achieve an estimation, so online learning is impractical.

3 Monte Carlo methods

3.1 Monte Carlo approximation

Most generally speaking, *Monte Carlo (MC) methods* are algorithms that approximate the expected value of a random variable by taking the *empirical mean* of many independent samples of the random variable. The empirical mean $\overline{x}$ is an estimate of the population mean μ:

Definition 3.1 (Empirical mean) *Let $X_1, X_2, ..., X_n$ be n random samples, each sharing the population distribution i.e. mean μ and standard deviation θ. The empirical mean is then defined to be:*

$$\overline{x} := \frac{1}{n}(X_1 + X_2 + ... + X_n)$$

Monte Carlo methods approximate the population mean by applying the *law of large numbers* to the empirical mean:

Theorem 2 (Law of large numbers) *Let $X_1, X_2, ..., X_n$ be independent and identically distributed (i.i.d.) random variables with expected value $\mathbb{E}(X_1) = \mathbb{E}(X_2) = ... = \mu$, then*

$$\overline{x} \longrightarrow \mu \text{ as } n \longrightarrow \infty$$

These can be an extremely useful methods if we ever wish to estimate the expected value of a random variable. For example, suppose we want to calculate the mean μ of a distribution, but do not have enough information to calculate it exactly. We can approximate the value of μ as the empirical mean of n samples from the distribution, for some large n. As n grows, the empirical mean of these samples will converge to the previously unknown mean of the distribution, μ.

The ability of Monte Carlo methods to approximate the expected value of a random variable with *no prior knowledge of the distribution of the random variable* sets them apart from other methods, such as Dynamic Programming, that could provide an exact value but require details about the dynamics of the RV to do so. While the approximate value is less accurate, the error can be reduced to a negligible size by taking enough samples.

One further advantage is MC methods produce unbiased estimators by definition:

$$\mathbb{E}(\overline{x}) = \mathbb{E}(\frac{1}{n}(X_1+X_2+...+X_n)) = \frac{1}{n}(\mathbb{E}(X_1)+\mathbb{E}(X_2)+...+\mathbb{E}(X_n)) = \mu \quad (14)$$

If we then construct a parameterised loss function with target $\overline{x}$, we can then apply SGD to this loss function and ensure almost certain convergence of the parameters to a local minimum.

3.2 Monte Carlo evaluation

Recall in section 1.4 we saw how GPI can be implemented to solve the control problem i.e. ascertain an optimal policy. GPI is, as the name suggests, an iterative process in which we alternate between policy evaluation and policy improvement processes until convergence to an optimal value function and corresponding optimal policy. However, as section 1.5 remarked, exact policy evaluation (such as calculation by dynamic programming) is intractable for environments with large state spaces - in which case function approximation provides a route for policy evaluation. We then saw in section 2.4 how, given the values of each state, we can iteratively approximate a parameterisation of the value function using SGD. Notice however, this relies on the knowledge of the value of each state, which is not derivable when the environment's dynamics are unknown. This is where Monte Carlo methods come into play: we can approximate the value of states using Monte Carlo methods, then use these approximate values as targets for a loss function that we can minimise using SGD, as described in section 2.5.

Let us now begin formalising the process of approximating state values using Monte Carlo methods. Knowledge of a model of the environment's transition probabilities, $p(s'|a, s)$ for $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, determines the type of value approximations required in order to perform the subsequent policy improvement step. If the transition probabilities are readily available it is sufficient to calculate the expected return of starting in state $s \in \mathcal{S}$ and thereafter following policy π, i.e. the *state-value* $v_\pi(s)$. We can use these approximate state-values in conjunction with the transition policies to construct a new deterministic policy that chooses the action that leads to the best combination of reward and next state. On the other hand, if the transition probabilities are *unknown*, we must calculate the expected return when starting in state $s \in \mathcal{S}$, choosing action $a \in \mathcal{A}(s)$, then following policy π thereafter, i.e. the *action-value* $q_\pi(s, a)$. For the remainder of this section we investigate the approximation of action-values, since in reality it is likely that transition probabilities will be unknown, especially since we are opting for a Monte Carlo approximation specifically to combat the lack of knowledge of the environment's dynamics.

Applying the Monte Carlo method described in the previous section, we can approximate the expected return of a state-action pair to be the empirical mean of many samples of the return experienced after visiting that state-action pair. Note, this method limits us to use in episodic environments, since the return of a state-action pair is only finalised at the termination of each episode. We then have a choice of which visits within an episode we wish to sample: the first implementation is called the *first-visit MC method* in which we restrict our sampling to the return following only the *first visit* to a particular state-action pair s, a within an episode. Without this restraint, we would sample the returns experienced after later visits to the same state-action pair within an episode, which are each summands in the calculation of the return experienced after the first visit. Clearly these returns are at most conditionally independent, hence causing the set of samples to fail to satisfy the i.i.d. condition of the law of large numbers. However, a generalisation of the theorem known as the Kolmogorov strong law of large numbers states that, for a conditionally independent sequence of random variables satisfying some conditions, the empirical mean of the observations converges to the population mean. Therefore, the alternative sampling method in which we sample every visit to a state-action pair within an episode, known as the *every-visit MC method*, converges to the true expected return of the pair as the number of samples tends to infinity.

We have established that we can approximately solve the policy evaluation problem using SGD. However, in the proof of the approximation's convergence we assumed access to unbiased estimates of the expected return for *every state-action pair*. Acquiring these estimates through Monte Carlo methods depends on having access to samples from an infinite number of episodes, which is unrealistic in practice. Furthermore, given an infinite number of episodes we still need to guarantee that *every state-action pair* will be visited an infinite number of times, i.e. the problem of *maintaining exploration*. This is not the case for every policy: a deterministic policy only visits one action for each state, neglecting visiting any alternative actions, so it will be impossible to approximate the values of the neglected actions using MC methods. Clearly using a policy π under which there exists a state $s \in \mathcal{S}$ for which some action $a \in \mathcal{A}(s)$ has a 0 probability of being selected will cause a lot of problems for Monte Carlo value approximation. These problems with such a policy can be overcome with the assumption of *exploring starts*, where the probability that the first state $S_0 = s$ and the first action $A_0 = a$ are non-zero for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. With this assumption and the assumption of infinite episodes we can guarantee all state action pairs will be visited an infinite number of times.

We are now in a position to define first-visit Monte Carlo approximation of an action-value under the assumptions of an infinite number of episodes and exploring starts:

Definition 3.2 (Monte Carlo evaluation) *Suppose we are following a policy π in an environment that guarantees exploring starts. Let $(G_n)_{n\in\mathbb{N}}$ be a sequence of returns observed after the first-visit to a state $s \in \mathcal{S}$ and action $a \in \mathcal{A}(s)$. Then we can define:*

$$Q_{\pi}^{ES}(s,a) := \overline{g}$$

Applying the law of large numbers to this definition produces the following theorem:

Theorem 3 *Let π be a policy, $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, then*

$$Q_{\pi}^{ES}(s,a) \longrightarrow q_{\pi}(s,a) \text{ as } n \longrightarrow \infty$$

This theorem compiles everything we have discussed so far and confirms that, assuming an infinite number of episodes and exploring starts, we can approximate action-values for all state-action pairs. One downside of Monte Carlo methods in general is the high variance of the approximations they produce, especially when compared to alternative bootstrapping methods such as TD learning, but we will return to this later. It is worth noting that exploring starts is quite an unrealistic assumption in practice: it is rare that we can make any guarantees on the starting state. Thankfully there are other ways we can solve the problem of maintaining exploration that do not involve assumptions about the environment.

3.3 On-policy methods

There are a number of alternative Monte Carlo methods that can be used to approximate value functions without the assumption of exploring starts. The first type of methods we will cover are known as *on-policy methods*. These methods work by restricting the policy search space to a subset of exploratory policies known as *ϵ-soft* policies:

Definition 3.3 (ϵ-soft policy) *Let π be a policy and $\epsilon > 0$, then we say π is an ϵ-soft policy if:*

$$\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|} \textit{ for all } s \in \mathcal{S} \textit{ and } a \in \mathcal{A}(s)$$

Notice how these policies satisfy $\pi(a|s) > 0$ for all state-action pairs. This means that given an infinite number of episodes each pair will be visited an infinite number of times, enabling the use of Monte Carlo methods for approximately solving the policy evaluation problem *without the assumption of exploring starts*. Our task is now to define a policy improvement method that has the following three properties:

1. It maintains the ϵ-soft property between improvements.
2. It satisfies the Policy Improvement Theorem (Theorem 1).
3. If the old and improved policies are equal then both are the optimal policy

Satisfying both of these conditions will ensure convergence to the optimal *epsilon*-soft policy, denoted $\tilde{\pi}_*$. Let us now define a subset of ε-soft policies that play a crucial role in on-policy improvement:

Definition 3.4 (ε-greedy policy) *Let π be a policy with action-value function $q_\pi(s,a)$, then we can define π', the ε-greedy policy w.r.t. π, as follows:*

$$\pi'(a \mid s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|} & \textit{if } a = \operatorname{argmax}_{a \in \mathcal{A}(s)} q_\pi(s,a) \\ \frac{\varepsilon}{|\mathcal{A}(s)|} & \textit{otherwise} \end{cases}$$

Suppose for the current policy π we have the action-value equation $q_\pi(s,a)$. Our goal is now to perform policy improvement by constructing a new policy π', such that $\pi' \geq \pi$ ($q_{\pi'}(s,a) \geq q_\pi(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$). Suppose we construct π' to be the ε-greedy policy w.r.t. π. Firstly, it is clear that $\pi'(a \mid s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ for all states and actions. Hence π' is ε-soft so the new policy maintains exploration. Secondly, it can be shown that for any $s \in \mathcal{S}$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

Then immediately $\pi' \geq \pi$ by the Policy Improvement Theorem. So from the first two properties, GPI with ε-greedy policy improvement will produce a monotonically increasing sequence of policies. Lastly, it can be shown that:

$$q_\pi(s,a) = q_{\pi'}(s,a) \implies \pi = \pi' = \tilde{\pi}_*$$

So GPI with ε-greedy policy improvement converges to the optimum ε-soft policy $\tilde{\pi}_*$.

So far we have been working with the assumption of knowing $q_\pi(s, a)$, but in practice we will be using a Monte Carlo evaluation $Q_\pi(s, a)$, defined very similarly to the exploring starts Monte Carlo evaluation:

Definition 3.5 (On-policy Monte Carlo evaluation) *Suppose we are following an ε-soft policy π. Let $(G_n)_{n\in\mathbb{N}}$ be a sequence of return observed after the first visit to a state $s \in \mathcal{S}$ and action $a \in \mathcal{A}(s)$. Then we can define:*

$$Q^{OP}_\pi(s, a) := \overline{g}$$

Let Q_n denote the value of $Q^{OP}_\pi(s, a)$ calculated at the termination of the nth episode containing a visit to s, a. We have the option to adapt on-policy Monte Carlo evaluation to an incremental evaluation of Q_n. Let G_n be the return observed after the nth first-visit to s, a. Currently, we calculate Q_n after termination of the episode containing the $(n-1)$th sample as follows:

$$Q_n = \frac{G_1 + G_2 + ... + G_{n-1}}{n-1}$$

This process can be reformulated as a running average, calculated as follows:

$$Q_{n+1} = Q_n + \frac{1}{n}(G_n - Q_n) \tag{15}$$

where Q_1 is arbitrary. This method provides benefits of constant memory and time requirements.

3.4 Off-policy methods

We now turn our attention to a second subset of Monte Carlo methods that can perform policy evaluation without the assumption of exploring starts. These methods are known as *off-policy methods* and are based on the principle of having two policies: a *behaviour policy*, which the agent will be following when generating sampled episodes, and a *target policy* which we will learn to optimise using the observed behaviour of b. So supposing π is the target policy and b is the behaviour policy such that $\pi \neq b$, our objective is to approximate q_π using Monte Carlo evaluation on sample episodes generated by b. If Q is the Monte Carlo Evaluation of π, there is one condition we must place on b to ensure almost certain convergence of Q to q_π:

Definition 3.6 (Coverage) *Let π and b be policies, then we say b satisfies coverage of π if for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$:*

$$\pi(a \mid s) > 0 \implies b(a \mid s) > 0$$

However, in order to maintain exploration we will need $b(a \mid s) > 0$ for all states and actions, known as a *soft* policy, which clearly implies b satisfies coverage of any policy π.

In comparison to on-policy methods, off-policy methods generally produce higher variance approximations, which means they take longer to converge. On the other hand, off-policy methods are generally more powerful: they allow the discovery of the optimal policy outside the set of ε-soft policies and enable learning from a non-learning controller. It is worth noticing that on-policy methods are in fact the subset of off-policy methods such that $\pi = b$.

We can now move on to our stated objective of producing an off-policy Monte Carlo approximate Q of q_π using returns experienced following the soft policy b. First of all it will be useful to think about time as being continuous across episodes, such that if T is the time at which the first episode terminates then the second episode will begin at time $T+1$. Let $T(t)$ denote the first termination following time t, let $\mathcal{T}(s,a)$ denote the set of all the times of first-visits to the state-action pair s, a across all episodes and let G_t denote the return experienced following time t until $T(t)$. Let $t_i \in \mathcal{T}(s,a)$ denote the ith visit to s, a, then $(G_{t_i})_{t_i \in \mathcal{T}(s,a)}$ is the sequence of all returns for s, a. Using this notation we can now define a formula that is crucial to off-policy Monte Carlo:

Definition 3.7 (Importance sampling ratio)

$$\rho_{t:T(t)-1} = \prod_{k=t}^{T(t)-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

To abbreviate this notation, let $w_i := \rho_{t_i:T(t_i)-1}$ denote the importance sampling ratio for return G_{t_i}. Therefore $(w_i)_{i \in \mathbb{N}} = (\rho_{t_i:T(t_i)-1})_{t_i \in \mathcal{T}(s,a)}$ is the set containing the importance sampling ratios for every G_{t_i}, the returns experienced after visiting s, a.

Since we are obtaining all sample returns by following the policy b, it is clear that:

$$\mathbb{E}[G_t | S_t = s, A_t = a] = q_b(s, a)$$

then multiplying both sides by the importance sampling ratio we get

$$\mathbb{E}[\rho_{t:T(t)-1}G_t|S_t = s, A_t = a] = q_\pi(s,a) \tag{16}$$

showing that by incorporating the importance sampling ratio we can indeed approximate the value function for policy π using only samples from another policy b that satisfies coverage of π. This type of off-policy Monte Carlo evaluation is known simply as *importance sampling*, of which there are two subcategories: *ordinary importance sampling* and *weighted importance sampling.*

Definition 3.8 (Ordinary importance sampling) *Let π and b be policies such that b satisfies coverage of π. Let G_t denote the return observed following b and $\rho_{t:T(t)}$ denote the importance sampling ratio of π and b, both between times t and $T(t)$. Then for $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:*

$$Q_\pi^{OIS}(s,a) := \frac{\sum_{t\in\mathcal{T}(s,a)} \rho_{t:T(t)}G_t}{|\mathcal{T}(s,a)|}$$

We have $\mathbb{E}[Q_\pi^{OIS}] = q_\pi$, so ordinary importance sampling produces an unbiased estimate, making this approximation ideal for use as a SGD target. However, it suffers from unbounded variance, so in practice it commonly requires a large amount of data to converge; although uncommon, in the worst case the approximation will never converge.

We can reformulate this method as an incremental calculation that can be used to improve estimates at the termination of each episode, the same technique used in equation 15. If we let Q_n denote the value of $Q_\pi^{OIS}(s,a)$ calculated at the termination of the nth episode containing a visit to s, a, then:

$$Q_{n+1} = Q_n + \frac{1}{|\mathcal{T}(s,a)|}(W_nG_n - Q_n) \tag{17}$$

This formulation maintains the aforementioned benefits of constant memory and time requirements.

We now turn our attention to the second form of importance sampling:

Definition 3.9 (Weighted importance sampling) *Let π and b be policies such that b satisfies coverage of π. Let G_t denote the return observed* *following b and $\rho_{t:T(t)}$ denote the importance sampling ratio of π and b, both between times t and $T(t)$. Then for $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:*

$$Q_\pi^{WIS}(s,a) := \begin{cases} \dfrac{\sum_{t \in \mathcal{T}(s,a)} \rho_{t:T(t)} G_t}{\sum_{t \in \mathcal{T}(s,a)} \rho_{t:T(t)}} & \textit{if} \sum\limits_{t \in \mathcal{T}(s,a)} \rho_{t:T(t)} \neq 0 \\ 0 & \textit{if} \sum\limits_{t \in \mathcal{T}(s,a)} \rho_{t:T(t)} = 0 \end{cases}$$

This time, we have $\mathbb{E}[Q_\pi^{WIS}] = q_b \neq q_\pi$, so this is an biased estimator, preventing almost certain convergence when used as a SGD target. In reality this is usually less of an issue than it sounds, since the bias converges to 0 as the number of samples increases. Furthermore, weighted importance sampling benefits from lower variance.

We can also reformulate weighted importance sampling as an incremental process in much the same way that we did for incremental importance sampling. To do so let C_n denote the cumulative sum of the first n importance sampling ratios, defined recursively as follows:

$$\begin{aligned} C_{n+1} &= C_n + W_{n+1} \\ C_0 &= 0 \end{aligned}$$

Then let Q_n denote the value of $Q_\pi^{WIS}(s,a)$ at the termination of the nth episode containing a visit to s, a. Putting this all together we can form the following incremental weighted importance sampling method:

$$Q_{n+1} = Q_n + \frac{W_n}{C_n}(G_n - Q_n) \tag{18}$$

So far we have been considering importance sampling using first-visit sampling. In the case of every-visit sampling, both normal and weighted importance sampling are biased, but again the bias converges to 0 as the number of samples increases. Every-visit methods provide further benefits of constant memory requirements (no need to store all the state-value pairs visited within each episode) and being more suited to SGD.

We can use an example to show that putting this all into practice to solve the control problem is relatively straight forward. Suppose we have a target policy π and a soft behaviour policy b, so that b satisfies coverage of π. Solving the control problem is then equivalent to finding π_*, which we can achieve by adapting GPI to use weighted importance sampling. Repeat the following process until $\pi = \pi_*$:

1. Generate an episode following the behaviour policy b.
2. Set $t = T - 1$ and repeat the steps 3-5 until $t = -1$.
3. Update the approximate action-value function $Q_\pi^{WIS}(S_t, A_t)$ using the incremental weighted importance sampling formula for π and b (equation 18).
4. Set $\pi(S_t) = \text{argmax}_{a \in \mathcal{A}(S_t)} Q_\pi^{WIS}(S_t, a)$.
5. Deduct 1 from t.

Here, step 3 is equivalent to the evaluation step of GPI and step 4 the improvement step. We have that $\pi \longrightarrow \pi_*$ as the number of episodes generated form b tends to infinity.

3.5 Inverse transform sampling

So far in this section we have investigated how sampled episodes can be used to perform Monte Carlo evaluation and control. However, one thing that has not been mentioned is how exactly those sample episodes are generated. The mechanism for sampling a deterministic policy is straight forward: for all states s there is some $a \in \mathcal{A}(s)$ such that $\pi(a|s) = 1$, while the chance of selecting any other action in zero. So in this case, when $S_t = s$ we simply take the deterministic action, $A_t = a$ such that $\pi(a|s) = 1$, then record the resulting reward and state, R_{t+1} and S_{t+1} respectively.

Things get more complicated when we want to sample a stochastic policy, such as an ε-greedy policy, in which for each state s there are a number of actions $a \in \mathcal{A}(s)$ such that $\pi(a|s) > 0$ and for all $a \in \mathcal{A}(s)$ we have $\pi(a|s) < 1$. In this scenario, given $S_t = s$, we want to randomly sample an action $A_t = a \in \mathcal{A}(s)$ according to the distribution $\pi(a|s)$, so that $A_t \sim \pi(a|s)$. It is straightforward to sample the *uniform distribution*, $\boldsymbol{U}(0,1)$, using a pseudorandom number generator, so our goal is to convert some sample $U \sim \boldsymbol{U}(0,1)$ into a sample $A_t \sim \pi(a|s)$. To sample from a distribution in this way we can use a technique known as *inverse transform sampling*, which we will first investigate generally for discrete random variables before placing the technique in the context of reinforcement learning.

Probability distributions can generally be divided into two categories: continuous and discrete distributions (although there are some distributions belonging to both). Inverse transform sampling works differently for each of these types of distributions, so they must be considered separately. In this project I am only covering finite MDPs, where there the set of states $\mathcal{S}$ is finite. These are equivalently known as discrete state MDPs, since all policies in these MDPs are discrete distributions. Therefore, it is only relevant to consider inverse transform sampling for discrete random variables taking finitely many values.

Suppose X is a discrete random variable that can take values from the set $x_1, x_2, \ldots, x_N$, then the distribution of X, f, is of the form:

$$f(x_k) := Pr(X = x_k) \tag{19}$$

for $0 < k \leq N < \infty$ where $f(x_k) \geq 0$ and $\sum_k f(x_k) = 1$. The distribution of X can be completely specified through its *cumulative distribution function (CDF)*, F, which is defined:

$$F(x_k) := \begin{cases} \sum_{i=1}^{k} f(x_i) & \text{for } k = 1, \ldots N \\ 0 & \text{if } k = 0 \end{cases} \tag{20}$$

Suppose that we are able to calculate the inverse of the CDF F, denoted F^{-1}, such that

$$F(k-1) < u \leq F(k) \iff F^{-1}(u) = k \tag{21}$$

Then we can use the following theorem[9] to sample from F by applying the CDF's inverse F^{-1} to a sample from the *uniform distribution* $\boldsymbol{U}(0,1)$:

Theorem 4 (Inverse transform sampling) *Let F be a cumulative distribution function and let F^{-1} be its inverse. If $U \sim \boldsymbol{U}(0,1)$ and $X = F^{-1}(U)$ then $X \sim F$*

We can compute $F^{-1}(u)$ by constructing a table $(k, F(k))|0 \leq k \leq N$ and then searching for the solution k satisfying Equation 21.

Now, returning to the context of sampling actions from a policy, suppose the environment is in state $S_t = s$ at timestep t and we want to sample an action A_t according to the distribution π. Our first step in this endeavor is to formulate the cumulative distribution function that uniquely identifies π. To do this we must order the set of actions, which can be done arbitrarily, so let

$a_1, a_2, \ldots, a_N = \mathcal{A}(s)$. Then we can define the cumulative policy distribution function:

$$F(a_k|s) := \begin{cases} \sum_{i=1}^{k} \pi(a_i|s) & \text{for } k = 1, \ldots N \\ 0 & \text{if } k = 0 \end{cases} \tag{22}$$

Then we can calculate the inverse of the policy's CDF by constructing a table with entries $\{(k, F(a_k|s) \mid 0 \leq k \leq N\}$. Finally, to attain a sample from $\pi(a|s)$ we must sample some $U \sim \boldsymbol{U}(0,1)$, then find the X in the table such that:

$$F(X-1) < U < F(X) \tag{23}$$

By Equation 21 we know that $X = F^{-1}(U)$ and then by Theorem 4 we know that $X \sim F$. Since F is the CDF that uniquely identifies π, we therefore have that $X \sim \pi(a|s)$, so X is a sampled action as required. We then conclude the sampling process by letting $A_t = X$.

It is worth briefly mentioning that in continuous state MDPs, where the set of states in countably infinite, it is not possible to use this method of inverse transform sampling. In these cases we must generate samples using a technique known as *Markov chain Monte Carlo (MCMC)*.

MCMC is a strategy for generating samples x_i while exploring the state space $\mathcal{X}$ using a Markov chain mechanism. This mechanism is constructed so that the chain spends more time in the most important regions. In particular, it is constructed so that the samples x_i mimic samples drawn from the target distribution $f(x)$. (We reiterate that we use MCMC when we cannot draw samples from $f(x)$ directly, but can evaluate $f(x)$ up to a normalising constant.)[1]

4 Tabular TD(0) methods

4.1 Sarsa

We now turn our attention to a third reinforcement learning paradigm, *temporal-difference (TD) learning*. These methods unify aspects of both dynamic programming and Monte Carlo methods, harnessing many of the benefits of both methods. For example, we have seen how DP methods use *bootstrapping* to incorporate old estimates in the calculation of new estimates, removing the need for randomness and therefore keeping the variance of these estimates low. We have also seen how MC methods use rewards observed from interaction with the environment to make estimates without any model of the environment's dynamics. TD-learning uses both of these techniques and is in many respects the best of both worlds.

Again, we will base our approach on GPI, beginning the construction of TD methods by solving the prediction problem. We will see that the solution is built on the same principle as dynamic programming, where we saw that we can approach the true value function by iteratively updating an approximate value function. Let us define a new *assignment* operator that will be useful in this endeavour:

Definition 4.1 *Let $Q(s, a)$ be the estimated value of some state-action pair s, a, and $x \in \mathbb{R}$. Then*

$$Q(s, a) \longleftarrow Q(s, a) + x$$

calculates the sum $Q(s, a) + x$ and stores the result as a new estimate of $Q(s, a)$.

As always with GPI we need to solve the problem of maintaining exploration, leading again to the question of learning on-policy vs off-policy. We will first define the simplest on-policy TD evaluation method, *Sarsa*:

Definition 4.2 (Sarsa) *Suppose we are following a policy π, then after choosing an action A_{t+1} and observing the resulting state S_{t+1} we can update the estimated value of the previous state-action pair S_t, A_t as follows:*

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

Sarsa is referred to as a temporal difference method because it improves action-value estimates based on the difference between the best estimates available at two neighbouring timesteps t and $t + 1$, $Q(S_t, A_t)$ and $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ respectively. In order to do this it must observe the state S_{t+1}, that the environment evolves to as a result of the action A_t, and then also observe the action A_{t+1} that is sampled from $\pi(a|S_{t+1})$. Hence, to update the value estimate $Q(S_t, A_t)$ using the Sarsa algorithm we must have sampled the sequence $\{S_t, A_t, R_t, S_{t+1}, A_{t+1}\}$, giving rise to the apt name Sarsa.

It is enlightening to notice the similarities between this update and the update used in iterative, on-policy MC, shown in equation 15. If we let $Q_t := Q(S_t, A_t)$, then these updates to the estimate Q_t are both of the form:

$$Q_t \longleftarrow Q_t + \alpha E_t$$

for some $\alpha > 0$ and an *error term* of the form $E_t = T_t - Q_t$, for some *target* T_t. This error term is used as a measure of the accuracy of our current value function estimate, and the convergence of $Q_t \longrightarrow q_\pi$ is equivalent to the convergence of $E_t \longrightarrow 0$. In the case of on-policy MC, we have $E_t = G_t - Q_t$, the difference between the observed return following the nth sample and the estimated return; $T_t = G_t$ is the target for the estimate Q_t. Now compare this to Sarsa, where the error is $E_t = R_{t+1} + \gamma Q_{t+1} - Q_t$; the estimate Q_t has target $T_t = R_{t+1} + \gamma Q_{t+1}$. Recall the Bellman expectation equation:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned} \tag{24}$$

It should be straightforward to see that on-policy MC uses the first formulation of the Bellman expectation equation as a target, while Sarsa uses the second formulation as a target. In both instances the true expectation of the target is unknown; MC circumvents this by approximating the expected return with a sample return, while TD samples only the following reward. While they may go about it in slightly different ways, this shows that both MC and TD methods learn from the experience they gain interacting with their environment.

Furthermore, notice that the Sarsa target for Q_t contains the current estimated value of the next state-action pair Q_{t+1}. This is a clear example of using an old estimate to inform a new estimate, and is therefore an example of bootstrapping, a commonality with DP. As stated above, sampling enables learning without a model and bootstrapping reduces estimate variance, and as an implementation of both techniques TD learning reaps both of these rewards.

One further advantage TD methods have over MC methods is the ability to produce a new estimate Q_t after the selection of the action at the following time-step $t + 1$, instead of needing to wait until the end of the episode to observe the full return. This attribute provides a multitude of benefits:

1. TD methods are able to learn the solution to tasks with particularly long episodes at a significantly faster rate than MC methods.

2. It enables online learning where policy improvement takes place at every timestep.

3. Since there is no need to wait until the termination of an episode to evaluate the current policy, TD learning can be used to solve *continuing*

tasks, in which episodes are infinite in length, never reaching termination (i.e. $T = \infty$).

4. Both methods converge subject to the condition of maintaining exploration, but since TD methods only consider the immediate reward, and not the entire return, the effect this sub-optimal exploration has on increasing the variance of the estimate is minimised.

As is the case for all the TD methods we will be discussing, it can proved that, for any fixed policy π that visits every state-action pair an infinite number of times and a step-size parameter α scheduled according to the usual stochastic approximation conditions, Sarsa will converge to q_π with probability 1.[5] Subsequently, it is straight forward to construct an on-policy control algorithm for which GPI converges to the optimal policy with probability 1. We can go about this in the same way we did for on-policy MC control: recall policy improvement must maintain behaviour policy exploration, so for example, if we improve π to be the ε-greedy policy w.r.t. Q, the current estimate value function, then if $\varepsilon \longrightarrow 0$ as $t \longrightarrow \infty$ (e.g. let $\varepsilon := \frac{1}{t}$), we ensure convergence to π_* with probability 1 .

4.2 Q-learning

Q-learning methods are one of the most widely used reinforcement learning methods, forming the second major class of TD-learning methods that, crucially, enable off-policy control. The Q-learning update rule inherits the same basic structure as all TD methods: an error function consisting of a sum of samples and a bootstrapped estimate. However, in this instance a unique approach to the formulation of the learning target is employed. Let us define the Q-learning update rule to begin this analysis:

Definition 4.3 (Q-learning) *Suppose we are following a policy π, then after arriving at state S_{t+1} we can update the estimated value of the previous state-action pair S_t, A_t as follows:*

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Comparing this to Sarsa we can see that the only modification lies in the Q-learning target $R_{t+1} + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a)$. We have seen how the Sarsa target models the Bellman expectation equation, so now recall the Bellman optimality equation:

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

The parallels here should be clear; in effect Q-learning works by targeting a bootstrapped estimate of the bellman optimality equation. Q-learning is considered to be an off-policy method, since Q directly approximates the greedy, optimal policy q_* instead of q_π, meaning learning can be done from experience under a very exploratory policy. It also implies we can learn the optimal policy from the experience of a non-learning controller.

It has been proved that Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely.[17] Hence, Q-learning is certainly an effective method of solving the control problem for finite, discrete MDPs. The degree of its effectiveness is situational however, and, while its ability to learn off-policy means it finds a broader range of potential applications, it is only the preferred method to Sarsa for solving certain tasks. Q-learning learns the optimal set of actions directly, even if one of the optimal actions has a slight random chance of incurring a massive negative reward. Consider a scenario where this massive negative reward represents entering a catastrophic event. If we are learning in a simulated environment, the risk of entering the state is worth it and we are able to maximise returns. However, then following the same policy in real life might not be such a good idea, since entering the catastrophic state will cause dire consequences. On the other hand, Sarsa learns a sub-optimal set of actions that then approach optimality with improvements over time, allowing the agent to learn which states are worth avoiding, resulting in convergence to a more conservative policy. This property makes Sarsa the ideal TD method if we want to learn on-policy while controlling a real life agent, with real life consequences.

4.3 Double Q-learning

In some stochastic environments the well-known reinforcement learning algorithm Q-learning performs very poorly. This poor performance is caused by large overestimations of action values. These overestimations result from a positive bias that is introduced because Q-learning uses the maximum action value as an approximation for the maximum expected action value.[6]

Q-learning is based on the principle of using an estimate of the maximum expected value as a target. We acquire this estimate as follows:

$$\text{expected maximum value} \approx \text{maximum expected value}$$

which characterised mathematically states:

$$\max_a Q_t(s_{t+1}, a) \approx \mathbb{E}\left[max_a Q_t(s_{t+1}, a)\right] \approx max_a \mathbb{E}\left[Q_t(s_{t+1}, a)\right]$$

While it is true that this technique produces a close approximation, it is also the case that the expected maximum value is biased to usually overestimate the maximum expected value. This is an example of *maximisation bias*, and is a problem widely encountered when using the max operator to calculate estimations.

Double Q-learning makes great strides in solving the problem of maximisation bias experienced with Q-learning by partitioning the set of samples into two disjoint subsets, which we can then use to produce a pair of unbiased estimates. Before we define the new algorithm, let us first identify the source of bias in Q-learning target:

$$\begin{aligned} T_t &= R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \\ &= R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a)) \end{aligned} \tag{25}$$

We can see that Q-learning takes the action that maximises Q and plugs it back into Q to form a new estimate of q_*. This means there will always be a bias towards choosing actions that maximise the current estimate Q, whereas we should be trying to choose actions that maximise the true action-value function q_*. Now suppose we have a set of sample episodes, S, from following a policy π. We can partition S into two disjoint sets S_A and S_B, such that $S = S_A \cup S_B$ and $S_A \cap S_B = \emptyset$. Then we can use these two sets to form two independent, unbiased estimates Q_A and Q_B using the following pair of equations:

$$Q_A(S_t, S_t) \longleftarrow Q_A(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_B(S_{t+1}, \operatorname*{argmax}_a Q_A(S_{t+1}, a)) - Q_A(S_t, A_t)\right] \tag{26}$$

while the update rule for Q_B is the same but with Q_A and Q_B switched. For Q_A and Q_B to be independent estimates, we must sample the reward used in the target from S_A and S_B respectively. This new formulation takes the action that maximises Q_B and plugs it back into Q_A to form a new estimate of q that is unbiased, in the sense that:

$$\mathbb{E}[Q_B(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_A(S_{t+1}, a))] = q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_A(S_{t+1}, a)) \tag{27}$$

and similar for Q_B. Suppose at each timestep we choose to update one of Q_A or Q_B such that the probability of selection is > 0 for both, then given infinite time both Q_A and Q_B will receive an infinite number of updates. Under this condition, the learning rate conditions described in the Robbins-Monro algorithm (Equations 9) and the condition of the policy maintaining exploration, in a given ergodic, finite MDP, both Q_A and Q_B, as updated by the Double Q-learning algorithm we have described, will converge to the optimal value function q_* with probability 1.[6] To solve the control problem with double Q-learning we simply take $Q = \frac{Q_A + Q_B}{2}$ (an unbiased estimate) and form π as the greedy policy according to Q. Then $\pi \longrightarrow \pi_*$ as $t \longrightarrow \infty$.

5 Approximate TD(λ) methods

5.1 n-step returns

In this section we will consider another family of reinforcement learning methods that bridge the gap between TD(0) and MC methods. Before we can describe these methods, however, we must formalise some of their components. To start, it will be useful to describe the intermediary *n-step TD* methods, which use an *n-step return* as the target for their update.

Definition 5.1 (n-step return) *Let $n \in \mathbb{N}$ such that $n \geq 1$, then:*

$$G_{t:t+n} := \begin{cases} \displaystyle\sum_{i=0}^{n-1} \gamma^i R_{t+1+i} + \gamma^n Q_{t+n-1}(S_t, A_t) & \textit{if } 0 \leq t < T - n \\ G_t & \textit{if } t \geq T - n \end{cases}$$

Here, we have given the n-step return in terms of the action-value estimate Q, but we could have just as easily given it in terms of the state-value estimate V. Which version is required is entirely context dependant and, since this context should always be clear, we will use the two interchangeably. As we can see, just like the TD(0) target this is a bootstrapped estimate, but in contrast it uses the next n sampled rewards instead of just the following reward. A 1-step return is as follows:

$$G_{t:t+1} = R_{t+1} + \gamma Q_{t+n-1}(S_t, A_t)$$

which is identical to the Sarsa target described in the previous chapter. Hence we can reformulate Sarsa as follows:

$$Q_{t+n}(S_t, A_t) := Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+1} - Q_{t+n-1}(S_t, A_t)\right] \tag{28}$$

for $0 \leq t < T$. This 1-step version of Sarsa is known as *Sarsa(0)*. We can then extend this idea to work with any n-step return, which introduces the following family of methods:

Definition 5.2 (n-step Sarsa) *Let $n \in \mathbb{N}$ such that $n \geq 1$, then:*

$$Q_{t+n}(S_t, A_t) := Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right]$$

One thing to note about n-step methods is that the estimate Q_t can only be calculated at time $t+n$, since it needs to wait until the reward R_{t+n} has been sampled and the previous estimate Q_{t+n-1} has been calculated. This delays learning slightly, meaning convergence may occur slower for a larger n. There is a comparable definition of n-step returns for any parameterised function approximator:

$$G_{t:t+n} := R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \boldsymbol{w}_{t+n-1}) \tag{29}$$

for $0 \leq t < T-n$, where $\hat{v}(s, \boldsymbol{w})$ is the approximate value of state s given the weight vector $\boldsymbol{w}$. Whenever we are working with a function approximation method, interpret $G_{t:t+n}$ as this parameterised function approximator n-step return definition.

5.2 λ-returns

It is possible to construct more complex targets as a weighted sum of other simpler targets, so long as the weights are all positive and sum to 1. An update rule with a target of this form is known as a *compound update*. For example, the target

$$\frac{1}{2}G_{t:t+1} + \frac{1}{4}G_{t:t+2} + \frac{1}{4}G_{t:t+3}$$

constitutes a valid compound update target, since the weights $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\}$ are all positive and sum to 1. Note that a compound update can only be performed once the longest of its component updates is complete. So in this case the update would be performed at timestep $t+3$.

λ-*returns* are a specific type of compound update target that work by averaging across all n-step returns:

Definition 5.3 (λ-return) *Let* $\lambda \in [0,1]$*, then*

$$\begin{aligned} G_t^\lambda &:= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n} \\ &= (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{T-t-1}G_t \end{aligned}$$

This is a valid compound update target, since $(1-\lambda)\lambda^{n-1} > 0$, for all $n \in \mathbb{N}$ such that $n \geq 1$, and the infinite series of weights is equal to 1:

$$\sum_{n=1}^{\infty}(1-\lambda)\lambda^{n-1} = 1 - \lambda + \lambda - \lambda^2 + \lambda^2 - \lambda^3 + \ldots = 1$$

In a λ-return the n-return term is weighted by $(1-\lambda)\lambda^{n-1}$, hence the weighting of each n-return decays by a factor of λ in relation to the weighting of the previous $(n-1)$-return. G_t^λ is therefore a target which gives decreasing weighting to reward samples that are observed at longer intervals after timestep t. The purpose of the λ parameter is to specify the rate at which the weighting of future samples decays. To understand this concept we will now discuss two special cases for the value of λ.

When $\boldsymbol{\lambda}$=1:

$$G_t^1 = 0 + 1G_t = G_t$$

This is the case when the weighting of future samples does not decay at all, so we consider all future samples when calculating the estimate Q_t. This is exactly the style of sampling used in MC-learning and, as expected, G_t^1 is exactly the Monte Carlo target.

When $\boldsymbol{\lambda}$=0:

$$G_t^0 = 1G_{t:t+1} + 0 = G_{t:t+1}$$

This is the case when the weighting of future samples decays to 0 immediately following the first sample, so the only sample considered when calculating Q_t

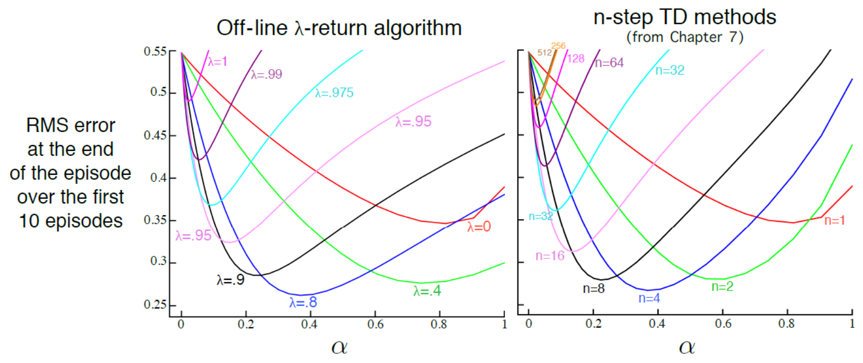

Figure 1: Performance of off-line λ-return algorithm vs n-step TD methods[15]

is R_{t+1}. This is exactly the style of sampling used in TD(0), and again, as expected, G_t^0 is exactly the TD(0) target.

We can now consider the general case of learning with the target G_t^λ for $\lambda \in [0, 1]$. The *off-line λ-return algorithm* is the name we give to SGD with the target G_t^λ. The weight update rule extends from Equation 13 in the obvious way:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \beta_t[G_t^\lambda - \hat{v}(S_t, \boldsymbol{w})]\nabla\hat{v}(S_t, \boldsymbol{w}) \tag{30}$$

for $t = 0, \ldots, T-1$. Notice, for any $\lambda > 0$ we will need to wait until the end of the episode before it is possible to update the weight vector, since only at termination will we know the value G_t^λ. This means that at termination a cascade of weight updates will occur, since each $\boldsymbol{w}_{t+1}$ depends on the previous timestep's weights, $\boldsymbol{w}_t$. This need to wait for termination is why the algorithm is known as an *off-line* method. This method effectively bridges the gap between Monte Carlo and TD methods, unifying the two as special cases of the off-line λ-return algorithm.

The similarities between this method and the n-step bootstrapping method described in the previous subsection should be clear: both methods introduce a variable that quantifies the importance of future samples when calculating a value estimate. As shown in Figure 1, this parallel is backed up by data comparing the average estimated root-mean-squared error between the ideal predictions and those found by the learning procedure across the first 10 episodes for the off-line λ-return algorithm and the n-step TD method. The root-mean-squared error is given by:

$$\sqrt{\overline{VE}(\boldsymbol{w})} := \sqrt{\sum_{s\in\mathcal{S}} \mu(s)\left[v_\pi(s) - \hat{v}(s,\boldsymbol{w})\right]^2} \tag{31}$$

Here we can clearly see that the two algorithms perform very similarly in general, with off-line λ-return performing slightly better with higher values of α. Also note that both algorithms perform best with intermediate bootstrapping parameters (λ and n respectively), but it is not possible to define a "best" parameter value, since performance varies from task to task.

5.3 TD(λ)

All the algorithms covered up to this point have approached learning with what is known as a *forward view*, meaning they make estimations by looking forward in time to future rewards and states. As previously stated, this means the algorithm but wait a number of timesteps until it can produce an accurate value estimate. An alternative approach is a *backwards view*, where new estimations are made by correcting for the TD-error of past estimates. This method provides the possibility of on-line learning, since we no longer need to wait to see the result of future actions.

To enable this backwards view approach we need to have some short-term record of the recent changes made to the weight vector $\boldsymbol{w}_t$. We can then use this record to recall how relevant each component of $\boldsymbol{w}_t$ is currently, and thus use it to determine the component's eligibility for updates. The record we use is a vector $\boldsymbol{z}_t \in \mathbb{R}^d$, called the *eligibility trace*, which is defined mathematically as follows:

Definition 5.4 (Eligibility trace) *Let* $\lambda \in [0,1]$*, then*

$$\boldsymbol{z}_t := \begin{cases} 0 & \textit{if } t < 0 \\ \gamma\lambda\boldsymbol{z}_{t-1} + \nabla\hat{v}(S_t,\boldsymbol{w}_t) & \textit{if } 0 \le t \le T \end{cases}$$

Notice that $\boldsymbol{z}_t$ and $\boldsymbol{w}_t$ are both d-dimensional vectors; each component of $\boldsymbol{z}_t$ weights the updates applied to the corresponding component of $\boldsymbol{w}_t$. In this definition γ is the usual discount rate parameter and λ is another parameter, known as the *trace decay parameter*, which plays an important role in defining how long traces from past states will maintain their relevance. $\boldsymbol{z}_t$ is initialised to 0 and is incremented by the gradient of the approximate value function, $\nabla\hat{v}(S_t, \boldsymbol{w}_t)$, at every timestep thereafter. These incrementations fade away by a factor of $\gamma\lambda$ at each subsequent timestep.

As we have stated, the eligibility trace is used to weight updates to the weight vector. But how are the updates themselves calculated? Again, we use the familiar TD-learning concept of correcting the value function estimate according to the current estimate's error. Recall the standard *one-step state-value prediction TD error*:

$$\delta_t := R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}_t) - \hat{v}(S_t, \boldsymbol{w}_t) \tag{32}$$

and recall that we must wait until timestep $t+1$ for this error the be calculated. Weighting this error by the eligibility trace $\boldsymbol{z}_t$ and the learning rate α produces the TD(λ) weight update for approximate value functions:

Definition 5.5 (TD(λ)) *Let* $\alpha \in [0, 1]$*, then*

$$\boldsymbol{w}_t := \boldsymbol{w}_t + \alpha\delta_t\boldsymbol{z}_t$$

This update is the best of both worlds: Firstly, we are able to consider a variable quantity of relevant returns, just like with the off-line λ-return algorithm, meaning we can choose an intermediate λ that produces a low relative error. Secondly, we are able to perform updates on-line, like we can with with TD(0), allowing for faster learning and learning in real world environments. It can be shown that TD(λ) converges almost certainly to the optimal policy given the correct scheduling of α and, furthermore, it matches the performance of the off-line λ-return algorithm for low values of

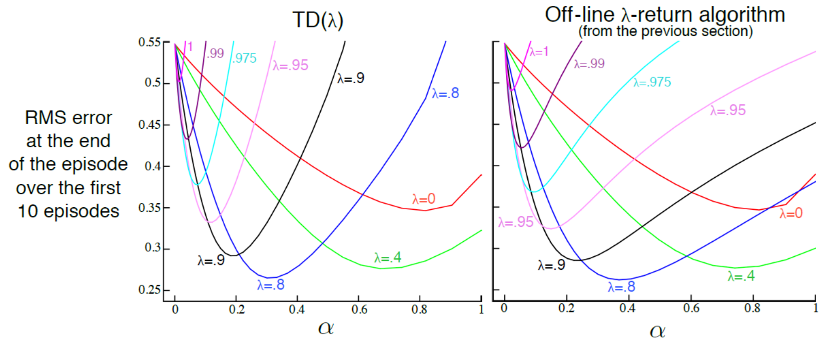

Figure 2: Performance of TD(λ) vs off-line λ-return algorithm[15]

α, while performing slightly worse for larger α (see Figure 2). Notice, for both algorithms an intermediately chosen λ is optimal ($\lambda \approx 0.8$) since higher λ results in higher variance estimates, while lower λ results in more biased estimates.

This backwards view approach provides a wide array of advantages in comparison to its forwards view counterparts, the first of which is the potential for reduced memory requirements. Off-line λ-return is a forward view algorithm that requires sampling the final return of the episode, R_t, before it can make any updates to the weight vector; this means we must store every state and return observed in the entire episode. Clearly for large T this poses a memory issue and for infinite T this renders the algorithm intractable. n-step methods truncate the returns, making them a viable option for continuing tasks. However, should we wish to use a large n we will still face significant memory issues. TD(λ) overcomes these memory challenges but only requiring the storage of the eligibility trace, a single d-dimensional vector, making it a far more scalable method.

The remaining set of advantages relate to the on-line nature of backwards view algorithms. Off-line λ-return and n-step methods must wait until episode termination and for n timesteps respectively before they can update the weight vector, while TD(λ) is able to begin updating weight vectors immediately. The removal of this delay means agents can begin improving the quality of their actions from the very first timestep, meaning more progress can be made within a single episode, and convergence to the optimal policy may come sooner. On-line learning also makes TD(λ) more suited to learning in a real world environment, where the consequences of its actions could be costly - it is far less likely to repeat the same mistake multiple times within an episode. Finally, on-line learning also means we can learn effectively in continuing tasks. While this is something it has in common with n-step methods, this is a huge advantage over the Off-line λ-return algorithm.

6 Actor-critic methods

6.1 Policy gradient methods

Every learning method we have considered so far has been based on the principle of somehow learning the value function for the current policy then, in one way or another, using that value function to construct a new policy.

We have seen that to learn on-line we must use on-policy methods, which require maintaining exploration of all state-action pairs. To achieve this we restrict the policy search space to the subset of ε-soft policies, which does not contain the true deterministic optimal policy. It is possible to approach this deterministic optimal policy by letting ε approach zero, but in reality it is very hard to schedule ε in such a way. An alternative to constructing polices from a learned value function is to learn a parameterisation of the optimal policy directly, with the value function now only used to learn the policy parameters. Let $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ denote the *policy parameter vector*, then the policy parameterised by $\boldsymbol{\theta}$ is defined:

$$\pi(a|s, \boldsymbol{\theta}) := Pr\{A_t = a | S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\} \tag{33}$$

This distribution represents the probability of taking action a at time t given that the environment is in state s and the policy parameters are $\boldsymbol{\theta}$. Since this method requires maintenance of exploration, we place the constraint on $\boldsymbol{\theta}$ that $\pi(a|s, \boldsymbol{\theta}) \in (0, 1)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. To constitute a proper policy, the probabilities of selecting all possible actions in a given state must sum to 1, so we also require that:

$$\sum_{a \in \mathcal{A}(s)} \pi(a|s, \boldsymbol{\theta}) = 1$$

for all $s \in \mathcal{S}$ and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$.

Defining a policy in terms of the parameters $\boldsymbol{\theta}$ such that the policy satisfies the aforementioned restraints is a problem in of itself. One popular solution is to use the *softmax function*, defined as follows:

Definition 6.1 (Softmax function) *Let $\boldsymbol{z} \in \mathbb{R}^K$ and z_i denote the ith component of $\boldsymbol{z}$. The standard softmax function $\sigma : \mathbb{R}^K \longrightarrow [0, 1]^K$ is then defined:*

$$\sigma(\boldsymbol{z}) = \left[\frac{e^{z_1}}{\sum_{j=1}^{K} e^{z_j}}, \frac{e^{z_2}}{\sum_{j=1}^{K} e^{z_j}}, \dots, \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \right]^T$$

The softmax function outputs a vector that represents the probability distributions of a list of potential outcomes.[16] The components of the output vector sum to 1 and are all positive, so each component can be used to represent the probability with which the corresponding component of the input vector should be selected. This equation clearly satisfies the constraints required for a parametric policy; if we define some function $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ that assigns a numerical preference value to each action a given the current state s and preference valuation parameters $\boldsymbol{\theta}$, then we can define a valid policy that is *softmax in action preferences*:

$$\pi(a|s, \boldsymbol{\theta}) := \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{b \in \mathcal{A}(s)} e^{h(s,b,\boldsymbol{\theta})}} \tag{34}$$

Therefore, our goal is to learn the parameterisation $\boldsymbol{\theta}$ of the action preference function $h(s, a, \boldsymbol{\theta})$ such that the derived softmax policy $\pi(a|s, \boldsymbol{\theta})$ is optimal. Note, policies that are softmax in action preferences are valid policies regardless of how the action preference function has been defined, so there are endless possibilities for defining h. One method is the define h as a linear combination of the state space's features:

$$h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^T x(s, a) \tag{35}$$

where $x(s, a) \in \mathbb{R}^{d'}$ is any mapping of the state-action pair to the d' dimensional feature space.

Now, in order to improve the current policy parameterisation we need some measure of the performance of the parameters. $J(\boldsymbol{\theta})$ denotes the performance of policy parameters $\boldsymbol{\theta}$, which we can define for episodic tasks as the value of the episode's starting state when following the policy parameterised by $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) := v_{\pi_{\boldsymbol{\theta}}}(S_0) \tag{36}$$

In this definition S_0 denotes the episode's starting state and $\pi_{\boldsymbol{\theta}}(a|s)$ denotes $\pi(a|s, \boldsymbol{\theta})$, so $v_{\pi_{\boldsymbol{\theta}}}$ is the true value of S_0 when following the policy parameterised by $\boldsymbol{\theta}$. For now we only consider the episodic case when returns are calculated with the discount rate parameter $\gamma = 1$ (see Definition 1.10), since discounting causes problems when performing function approximation. There is an alternative performance measure for continuing tasks which we will return to in a later subsection.

By maximising this performance measure we will directly learn the optimal policy. Recall, when performing value function approximation we minimised a performance measure using SGD, so now, in order to maximise the policy parameter performance function, we can use the closely related *gradient ascent* method to update the policy parameters:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{37}$$

Here α_t is the learning rate parameter at time t and $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is an approximation of $\nabla J(\boldsymbol{\theta}_t)$. Here, we improve the parameters by incrementing them in the direction that the performance function J grows fastest. Learning by gradient ascent is only possibly here since the policy $\pi_{\boldsymbol{\theta}}$ varies continuously with changes to $\boldsymbol{\theta}$. Conversely, the other methods considered so far are subject to discrete changes in the policy with only negligible changes in the value function: For example, suppose the first and second most valued actions (a_1 and a_2 respectively) available in a state s are close in value and we have constructed an ε-greedy policy according to this evaluation, so $\pi(a_1|s) = 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$ and $\pi(a_2|s) = \frac{\varepsilon}{|\mathcal{A}(s)|}$. Now suppose there is a small increase in the value of a_2, so that now $q_\pi(s, a_1) < q_\pi(s, a_2)$. A new policy π' constructed according to this new evaluation will have $\pi'(a_1|s) = \frac{\varepsilon}{|\mathcal{A}(s)|}$, a drastic change given only an negligibly small change in the action's valuation. This property means that policies defined by a value function do not vary continuously with changes to the value function and, since differentiation is only defined on continuous functions, are hence not differentiable and cannot be improved by policy gradient methods.

6.2 REINFORCE algorithm

Our job is now to find an approximation to $\nabla J(\boldsymbol{\theta}_t)$, which we can do in a number of ways. However, before we formalise these methods we must introduce a theorem that is the foundation of all of them, known as the *policy gradient theorem*.

Theorem 5 (Policy gradient theorem) *Let $\mu(s, \boldsymbol{\theta})$ be the invariant probability mass function (pmf) of the transition pmf $\sum_{a\in\mathcal{A}(s)} p(s'|s, a)\pi(a|s, \boldsymbol{\theta})$, then:*

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s, \boldsymbol{\theta}) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta})$$

It can be shown that this theorem holds for both episodic and continuing tasks.[14] We can deduce from this theorem that:

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \beta \sum_s \mu(s, \boldsymbol{\theta}) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \\ &= \beta \mathbb{E}\left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta})\right]\end{aligned}$$

for some $\beta \in \mathbb{R}$. Here, we acquire the second line by sampling the environment's state at time t, denoted S_t. Then, we can estimate this expected value by sampling approximate action-values from episodes generated according to $\pi_{\boldsymbol{\theta}}$:

$$\widehat{\nabla J(\boldsymbol{\theta})} = \beta \sum_a \hat{q}(S_t, a, \boldsymbol{w}_t) \nabla \pi(a|S_t, \boldsymbol{\theta})$$

Here S_t is a sampled state and $\hat{q}(s, a, \boldsymbol{w})$ is a value function parameterised by weight vector $\boldsymbol{w}$ which approximates $q_\pi(s, a)$. This is a satisfactory approximation to the gradient of J with respect to $\boldsymbol{\theta}$, so we can use it to form a gradient ascent parameter update known as the *all-actions method*. Note, β is strictly positive small quantity, so we can absorb β into α_t by letting $\alpha_t \leftarrow \alpha_t \beta$, which maintains the property that α_t is also strictly positive and small.

Definition 6.2 (All-actions method)

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \sum_a \hat{q}(S_t, a, \boldsymbol{w}_t) \nabla \pi(a|S_t, \boldsymbol{\theta}_t)$$

Notice this update still requires an approximation of the action-value equation, which can be learned using any of the previously discussed methods. This method garners its name from the estimation of the values of *all actions* available in the sampled current state S_t. An alternative method is to instead only estimate the value of the sampled action A_t, the action chosen according to $\pi_{\boldsymbol{\theta}}$. We can construct such a method by reformulating the policy gradient theorem:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}\left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta})\right] \\ &= \mathbb{E}\left[\sum_a \pi(a|S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})}\right] \\ &= \mathbb{E}[q_\pi(S_t, A_t) \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})] \\ &= \mathbb{E}[G_t \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})] \end{aligned} \tag{38}$$

Here G_t denotes the sampled discounted return experienced following timestep t. To this end, we have used the following results:

$$\mathbb{E}\left[\sum_a \pi(a|S_t, \boldsymbol{\theta}) q_\pi(S_t, a)\right] = \mathbb{E}\left[q_\pi(S_t, A_t)\right]$$

which works because the expected action according to $\pi_{\boldsymbol{\theta}}$ is the sampled action A_t. Then, by definition we have that:

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

Also notice that:

$$\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}) = \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$$

This equality follows directly from the identity:

$$\nabla \ln x = \frac{\nabla x}{x}$$

We can then use Equation 38 to form the policy parameter update for the *REINFORCE algorithm*:

Definition 6.3 (REINFORCE)

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t G_t \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$$

This update requires sampling the return G_t for all timesteps t, which all only become defined at termination when $t = T$. This means that all parameter updates are performed with a forward view at the end of each episode, making this a Monte Carlo method. Like all Monte Carlo methods, REINFORCE suffers from a high variance, and consequently relatively slow convergence. It can be shown that, given the correct scheduling of the learning rate α, REINFORCE converges almost certainly to a global optimum policy.[10]

There are some further advantages to these methods that we can cover briefly now. Suppose it is the case that the global optimum policy is a deterministic policy. While it is possible to learn this policy by finding the optimum ε-greedy policy through value function approximation and correctly scheduling ε to approach 0, correctly scheduling ε can be difficult in practice however, especially without knowledge of the environment's dynamics, often resulting in slow learning. Since policy approximation directly approaches the optimum policy, even if that policy is deterministic, it can prove to be a much faster and simpler method to use in these circumstances. Policy parameterisation also allows you to inject prior knowledge about the optimum policy with the initialisation of $\boldsymbol{\theta}$, which can also massively reduce the time it takes to learn the optimal policy.

6.3 Episodic actor-critic methods

High variance is the main pitfall of the policy gradient methods described so far, but to remedy this it is essential to understand the cause of the variance. In the majority of cases, given a state $s \in \mathcal{S}$, the action-values $q_\pi(s, a)$ will be similar for most actions $a \in \mathcal{A}(s)$, as it is likely for the remainder of the episode to constitute a similar sequence of actions, regardless of the action selected. This means that the difference in value between the best and worse actions will appear small in relation to their values, especially for actions with large subsequent returns taken near the beginning of the episode, which in turn causes the high variance of the methods. One way of making the difference in values of actions more apparent is by subtracting the expected subsequent return, which we can do with the introduction of a *baseline*.

A baseline is simply a function that is independent of a. For our purposes it will be helpful for this baseline to be a function of the current state, so we let $b : \mathcal{S} \longrightarrow \mathbb{R}$ denote the baseline function. Now, to incorporate this baseline into our update, we must first incorporate it into the policy gradient theorem:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}\left[\sum_a \left(q_\pi(S_t, a) - b(S_t)\right) \nabla \pi(a|S_t, \boldsymbol{\theta})\right] \\ &= \mathbb{E}[(G_t - b(S_t)) \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})] \end{aligned} \tag{39}$$

This formulation with a baseline is equivalent to the policy gradient theorem, since the sum of all the terms with the factor $b(S_t)$ is zero:

$$\sum_a b(s) \nabla \pi(a|s\boldsymbol{\theta}) = b(s) \nabla \sum_a \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla 1 = 0$$

So, using Equation 39 we can derive an variation of the REINFORCE algorithm that incorporates a baseline function:

Definition 6.4 (REINFORCE with baseline)

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t (G_t - b(S_t)) \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$$

Now, returning to our motivation of subtracting the expected subsequent return from the sampled return in order to reduce variance, we can let $b(s) = \hat{v}(s, \boldsymbol{w})$, which is the approximate value of a state $s \in \mathcal{S}$ given by the value function parameterised by weight vector $\boldsymbol{w} \in \mathbb{R}^d$ (See Subsection 1.5). This is a valid choice of baseline function since it is a function of states and is independent of actions. The derived REINFORCE with baseline parameter update is defined:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t(G_t - \hat{v}(s, \boldsymbol{w}_t))\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t) \tag{40}$$

Here $\boldsymbol{w}_t$ denotes the weight vector parameterising the approximate value function $\hat{v}$ at time t. Recall REINFORCE is a Monte Carlo method, so it makes sense to also learn $\hat{v}(s, \boldsymbol{w})$ using Monte Carlo methods.

So far we have been using the return G_t as our estimate of $q_\pi(S_t, A_t)$, but there are many alternative approximations that exist. The 1-step return $G_{t:t+1}$ (see Definition 5.1) is another valid estimate, so we can use it to form a new update, known as an *actor-critic method*:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t(G_{t:t+1} - \hat{v}(s, \boldsymbol{w}_t))\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t) \tag{41}$$

Notice that the following equation is equivalent to the TD(0) error, δ:

$$G_{t:t+1} - \hat{v}(S_t, \boldsymbol{w}_t) = R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}_t) - \hat{v} = \delta_t$$

Hence, the actor-critic update can be written in terms of δ:

Definition 6.5 (Actor-critic) *Let δ be the standard TD(0) error, then*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \delta_t \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$$

In these methods, the *actor* is the part of the method responsible for approximating the optimal policy parameterisation, while the *critic* is the part of the method responsible for approximating the value function, which is then used to assess the accuracy of the current policy. Alternatively, the n-step return $G_{t:t+n}$ (again see Definition 5.1) can be used as an estimate for $q_\pi(S_t, A_t)$, in which case the parameter update is of the form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t(G_{t:t+n} - \hat{v}(s, \boldsymbol{w}_t))\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t) \tag{42}$$

In this case the natural choice of algorithm for learning the approximate value function $\hat{v}(s, \boldsymbol{w})$ is n-step Sarsa. Another estimate that provides even more flexibility with the amount of sampling required is the λ-return G_t^λ (see Definition 5.3) which gives the obvious parameter update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t(G_t^\lambda - \hat{v}(s, \boldsymbol{w}_t))\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t) \tag{43}$$

Here, the natural choice of algorithm for learning the approximate state-value function $\hat{v}(s, \boldsymbol{w}_t)$ is the off-line λ-return algorithm. It is also possible to formulate a backwards view actor-critic method, where separate eligibility traces are used for the actor and critic. The critic's eligibility traces and updates are defined exactly as described in Subsection 5.3, while the actor's eligibility traces and updates are defined similarly:

$$\boldsymbol{z}_{t+1} = \gamma \boldsymbol{z}_t + \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t) \tag{44}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \delta \boldsymbol{z}_t \tag{45}$$

where δ_t is the TD(0) error and γ is the discount rate, both shared with the critic. λ, α_t and $\boldsymbol{z}_t$ are the trace decay parameter, learning rate and eligibility trace for the actor a timestep t respectively, which are all independent of the critic's respective γ, α_t and $\boldsymbol{z}_t$ parameters.

6.4 Continuing actor-critic methods

Recall, for a return in continuing task to have finite value we require that the discount rate $\gamma < 1$ and the sequence of rewards $(R_k)_{k\in\mathbb{N}}$ is bounded. But also recall we have stated that discounting is problematic with function approximation. Therefore, to adapt policy gradient methods for use in continuing tasks we need to reformulate returns without discounting rewards so that returns are still finite for continuing tasks with bounded sequences of rewards. To replace discounting we must introduce a new measure of rewards:

Definition 6.6 (Average reward) *Let $A_{0:t-1}$ denote $A_0, A_1, \ldots, A_{t-1}$. Suppose all of these actions were selected while following the policy π, then the average reward while following π is defined:*

$$\begin{aligned} r(\pi) &:= \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}\left[R_t | S_0, A_{0:t-1} \sim \pi\right] \\ &= \lim_{h\to\infty} \mathbb{E}\left[R_t | S_0, A_{0:t-1} \sim \pi\right] \end{aligned}$$

Note, the average reward is well defined for all ergodic MDPs, where "the starting state and any early decision made by the agent can have only a temporary effect; in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities."[15] A policy that maximises the average reward is considered an optimal policy. We can now use the average reward to define an alternative to traditional returns:

Definition 6.7 (Differential return)

$$G_t := R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$

Differential returns are free from any discounting and are finite for continuing tasks with bounded sequences of rewards, as required. Returning to the context of policy gradient methods, we can now define a finite policy parameter performance function for use in continuing tasks:

$$J(\boldsymbol{\theta}) := r(\pi_{\boldsymbol{\theta}}) \tag{46}$$

The policy gradient theorem holds for this $J(\boldsymbol{\theta})$, so it is possible to extend the backwards view actor-critic method to continuing tasks, where the goal is to maximise this performance function. The primary difference between the episodic and continuing cases is the definition of the TD(0) error, which must be reformulated in terms of the estimated average reward:

$$\delta_t := R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \boldsymbol{w}_t) - \hat{v}(S_t, \boldsymbol{w}_t) \tag{47}$$

Here $\boldsymbol{w}_t$ denotes the weight vector of parameters for the approximate value function $\hat{v}$ and $\bar{R}$ denotes the estimate at time t of the average reward $r(\pi)$. $\bar{R}$ is updated incrementally at each timestep as follows:

$$\bar{R}_t = \bar{R}_{t-1} + \alpha_t \delta_t \tag{48}$$

This definition of δ is used to calculate the eligibility traces for both the actor and critic at every timestep, which are updated using the same equations defined previously.

7 Conclusion

We have introduced Markov Decision Processes, explaining the goal of calculating an optimal policy that maximises expected returns. We have seen that, in the case where the MDP's dynamics equation is known we can approach an evaluation for the current policy using the bootstrapping method Dynamic Programming, before using this evaluation to improve the policy.

Alternating between policy evaluation and improvement yields the Generalised Policy Iteration algorithm, which is the basis for all the methods in sections 3-5.

We have seen why parameterised value functions can be useful for solving MDPs with large state spaces and have investigated the use of stochastic gradient descent for approximating a parameterisation of a policy's true value function. Batch gradient descent is an alternative to SGD that reduces variance. We noted that correctly scheduling the learning rate α is essential for being able to guarantee convergence to an optimum parameterisation with probability 1.

Monte Carlo methods introduced the practice of sampling episodes for to produce high-variance, unbiased estimates of the value function without knowledge of the dynamics equation. On-policy methods either rely on the assumption of exploring starts or maintain exploration by restricting the search space to ε-soft policies. Off-policy methods learn an optimal, deterministic target policy by importance sampling episodes from a soft behavioural policy under the constraint of coverage. Weighted importance sampling produces lower variance estimates than ordinary importance sampling. We finally saw how we can simulate episodes with inverse transform sampling.

Temporal Difference methods pair bootstrapping with sampling, producing biased estimates with lower variance than those produced using MC methods without the knowledge of the dynamics equation. Sarsa is an on-line method that samples the following state and action while Q-learning is an off-line method that estimates $q_\pi(s, a)$ by maximising the current approximate value function. This results in Q-learning suffering from maximisation bias, which can be solved using double Q-learning.

We have contrasted n-step Sarsa and the off-line λ-return algorithms to show that, for extreme values of λ close to 0 and 1, off-line λ-return approximates Sarsa and MC methods respectively, while intermediate settings are usually optimal. Eligibility traces enable the backwards view TD(λ) algorithm which is an on-line approximation of the off-line λ-return algorithm. Its on-line nature better suits this method to learning in real world scenarios.

Policy gradient methods are essential for parameterising stochastic policies that are not ε-soft, and often converge to the optimal policy in less steps than GPI based methods. We introduced the all-actions method and the REINFORCE algorithm before building upon REINFORCE with a variance-reducing baseline. Actor-critic methods are an instance of REINFORCE with baseline and can be extended to a backwards view with actor eligibility traces.

This project has been a thorough investigation into reinforcement learning and the major classes of methods used to execute this type of learning. While each method covered here has its computational shortcomings and is seldom the optimal RL algorithm in reality, these methods do produce policies that perform to a satisfactory level in most scenarios. Furthermore, the mathematical content of these methods is pertinent to solving more complex tasks using today's cutting edge RL methods.

References

[1] Christophe Andrieu et al. “An introduction to MCMC for machine learning”. In: *Machine learning* 50.1 (2003), pp. 5–43.

[2] J. Bernoulli and G. Cramer. *Anecdota*. Johannis Bernoulli ... Opera omnia,: tam antea sparsim edita, quam hactenus inedita. sumptibus M.M. Bousquet & sociorum, 1742. URL: https://books.google.co.uk/books?id=sxUOAAAAQAAJ.

[3] Blackburn. “Reinforcement Learning : Markov-Decision Process (Part 1)”. In: (July 2019). URL: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da.

[4] Blackburn. “Reinforcement Learning: Bellman Equation and Optimality (Part 2)”. In: (Aug. 2019). URL: https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3.

[5] Peter Dayan. “The Convergence of TD(lambda) for General lambda”. In: (May 1992). DOI: 10.1023/A:1022632907294.

[6] Hado Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[7] D Joyce. “Series Convergence Tests”. In: (). URL: https://mathcs.clarku.edu/~djoyce/ma122/posseries.pdf.

[8] Suki Lau. “Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning”. In: (June 2017). URL: https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1.

[9] Art B. Owen. *Monte Carlo theory, methods and examples*. 2013.

[10] V. V. Phansalkar and M. A. L. Thathachar. “Local and Global Optimization Algorithms for Generalized Learning Automata”. In: *Neural Computation* 7.5 (Sept. 1995), pp. 950–973. ISSN: 0899-7667. DOI: 10.1162/neco.1995.7.5.950. eprint: https://direct.mit.edu/neco/article-pdf/7/5/950/813154/neco.1995.7.5.950.pdf. URL: https://doi.org/10.1162/neco.1995.7.5.950.

[11] Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

[12] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: (Jan. 2016). URL: https://ruder.io/optimizing-gradient-descent/index.html#gradientdescentvariants.

[13] David Saad. *On-line learning in neural networks*. Cambridge University Press, 1998. ISBN: 9780511569920.

[14] Richard S Sutton et al. “Policy gradient methods for reinforcement learning with function approximation.” In: *NIPs*. Vol. 99. Citeseer. 1999, pp. 1057–1063.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning an Introduction*.

[16] Uniqtech. In: (Jan. 2018). URL: https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d.

[17] Christopher Watkins and Peter Dayan. “Technical Note: Q-Learning”. In: *Machine Learning* 8 (May 1992), pp. 279–292. DOI: 10.1007/BF00992698.

[18] BOTAO WU. “INVARIANT PROBABILITY DISTRIBUTIONS”. In: (2011).