Winter of Data Science Report
on
Reinforcement Learning

Ashwin Abraham

December 18th, 2022

Contents

Chapter 1

Introduction to Reinforcement Learning

1.1 What is Reinforcement Learning?

Reinforcement Learning is a type of Machine Learning where an agent learns to interact with it's environment in such a way as to maximize a reward signal. The agent is not told which actions are the best, but must learn this itself by trial and error.

1.2 Some Terminology

- The environment of the agent is characterized by it's **state**.
- The agent can take **actions** in order to modify the state.
- An action is taken by our agent every **timestep**.
- The state of the environment may change without any action from the agent (for example in two player games, where the other player is modelled as part of the environment).
- Every **timestep**, our agent takes an action and gets a **reward** based on the previous state and the action chosen.
- The goal of our agent is to learn a **policy/strategy** (a mapping from state to action) that maximizes the **total reward** the agent recieves.

- The **value** of a state is a measure of the maximum possible reward the agent can accumulate that state. For games that don't terminate, we can define the value of a state as the average reward per unit time that the agent can acquire from that state. In this situation it is clear that we are better of with a policy that chooses the action leading to the state of the highest value, and not the action with the highest reward.

In general, all of these may be stochastic and may change with time.

We have not defined any of these formally yet, and we shall revisit all these terms and define them when we learn about **Markov Decision Processes**, which are used to formally frame the Reinforcement Learning Problem.

1.3 Exploration vs Exploitation

The conflict between **exploration** and **exploitation** is one of the most challenging aspects of Reinforcement Learning. The problem arises due to the fact that we do not know neither the rewards associated with each action nor the value of each state beforehand. We can only estimate these from the rewards obtained so far.

To improve our estimates, we must try out all states, even the ones with low estimated rewards. This is known as **exploration**. However, we must also try to maximize the reward that we collect. Also, it is more important to know precisely the values of the higher value states than those of the lower value ones. To do this, we have to move to the states with the highest values. This is known as **exploitation**. Naturally, the two conflict with each other.

A large part of Reinforcement Learning is just finding creative ways of balancing exploration and exploitation.

1.4 Classical Methods

One situation where Reinforcement Learning is commonly used is in training an algorithm to play games. We shall soon look at Reinforcement Learning Algorithms that learn to play games. However, it is important to note that there are many classical ways of finding algorithms to win such games as well. Let us look at a few examples.

1.4.1 Game Trees

Let us look at two player *perfect information*[1] *zero-sum*[2] turn based finite games, such as Tic-Tac-Toe, Chess, etc. Each of these games ends in either a win for a player and a loss for the other or in a draw.

We can create a structure known as a game tree, which is a tree linking all the possible states the game can find itself in. Here each state corresponds to a node in the tree, with the starting position as the root. The children of each node correspond to the states reachable from the original node's state.

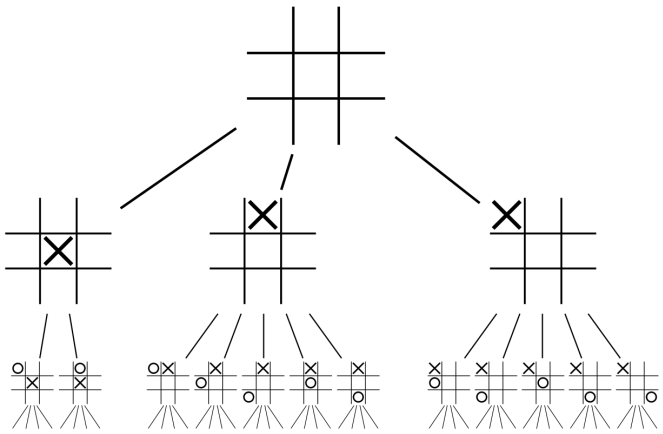

Figure 1.1: The first three levels of the game tree of Tic Tac Toe (upto symmetry)

Ideally, states should appear in pairs in the game tree, depending on which players turn it is to play. However, for multiplayer games where the players alternate in turns to play, we can identify the player if we know who the first player was and how many turns have passed since the game started. So in our tree, we do not need to have multiple copies of states differing only in player.

For two player turn based terminating games, in order to make the arguments less verbose, let us call the player who started the game **Player A**, and the other player, **Player B**. However, in any state, let us call the player whose turn it is to play, **Player 1** and the other player, **Player 2**. The introduction of these terms makes the arguments less verbose. However, it is important to note that the identities of

[1]Each player knows the entire state of the game at every stage

[2]If we associate to each outcome of the game a gain for each player (each player aims to maximize their gain), then for each outcome the sum of gains over all players is constant (taken to be 0 by convention). In the two player case, this means that each game either ends with a win for one player and a loss for the other or with a draw.

players 1 and 2 are not fixed and keep going back and forth between players A and B every turn.

Any game played corresponds to a path from the root of the tree to a leaf. At the root, player 1 is player A and player 2 is player B. As we go down the tree from one node to the next, the identities of players 1 and 2 keep getting swapped. Now, a leaf node must be either a win for the player 1 (a loss for player 2) or a loss for player 1 (a win for player 2) or a draw. Let us assign the leaf nodes a value of 0 if they correspond to draws, 1 if player 1 has won, and -1 if player 1 has lost.

In general, we assign a value of 1 to a node, if player 1 has a winning strategy, 0 if player 1 has no winning strategy, but can force a draw, and -1 if no matter what player 1 does, player 2 can play in such a way as to win.

Now, if all the children of a node have value 1, this means that no matter what player whose turn it is does, he moves into a state where the other player has a winning strategy. Therefore, this state is losing for the player, ie, it has value -1. On the other hand, if there exists a child state with value -1, then there exists a move that the player can make that puts the other player in a losing position, ie, the original state has value 1. If there exists no child state with value -1 but a child state with value 0, then the player does not have a winning strategy but can force a draw. Hence this state has value 0.

Now, we can find the values of every node in the tree from the bottom up, according to the relation:

$$V(u) = \begin{cases} 1, & \exists v \in C(u) : V(v) = -1 \\ 0, & \forall v \in C(u) : V(v) \geq 0, \exists v \in C(u) : V(v) = 0 \\ -1, & \forall v \in C(u) : V(v) = 1 \end{cases} \tag{1.1}$$

Here $V(u)$ represents the value of a node u, and $C(u)$ represents the set of children of u.

From, this we can conclude that every node in the tree has a value of either -1, 0 or 1, including the root node. This means that for every two person deterministic finite game, there exists a non-losing strategy for one of the players.

Given that we know the value of each node in the Game Tree, we can find an optimal strategy for a player. The strategy is this: If it is the players turn to play and the game is in state S, then move to the child state of S with the **lowest** value.

To prove this strategy is optimal, note that the moves $1 \to -1$ and $0 \to 0$ are the only optimal moves in the game, ie, after this move, any move by the opponent will leave us in a state atleast as good as the original state. The remaining moves ($1 \to 0$, $1 \to 1$ and $0 \to 1$) are suboptimal moves. Note that the move $-1 \to 1$ is a forced move, and so we will not consider it a suboptimal move.

Now, assume that we play the game with our strategy, and the opponent makes a suboptimal move in the duration of the game. In this case the opponent has either moved $1 \to 0$, $1 \to 1$ or $0 \to 1$. This means that we are now in a state of either 0 or 1. If we are in the state 0, no matter what the opponent does, she will not win, as we know of a strategy to force a draw. If we are in the state 1, then no matter what the opponent does, she will lose, as we know of a winning strategy.[3]

This means, that if we play using our strategy, the **only** way our opponent can win, is if she makes no suboptimal moves, ie, if she plays perfectly. This makes our strategy optimal.

Technically, an optimal strategy can be found this way for every deterministic terminating two player game, including games such as Chess. However, this method requires us to find the value of every node in the game tree, and the game trees for games such as Chess are extremely large (Chess has more than 10^{40} possible states!). This makes finding the optimal strategy in this way too computationally expensive.[4]

1.4.2 The Minimax Algorithm

This algorithm is a generalization of the game tree method for games with more than two players. In this case, our trick of having only one value function will not work. In this case, we will have k values for each state, one for each player.

We can once again build a game tree. Here we don't need to make separate states for states differing only in the player whose turn it is, as we are assigning separate value functions for them.

We assign values to the leaf nodes according to their favourabilities for the players. Each player seeks to reach a leaf with maximum value to her. We also associate each node to the player whose turn it is at that point in the game.

For non-leaf nodes, we define the value of a node u with respect to the player A as:

$$V_A(u) = \begin{cases} \min(V_A(v), v \in C(u)), & f(u) \neq A \\ \max(V_A(v), v \in C(u)), & f(u) = A \end{cases} \tag{1.2}$$

Here, $f(u)$ refers to the player whose turn it is to move in state u, and $C(u)$ refer to the child states of u.

The value at a node with respect to a player A represents the maximum possible value A can acquire at the end, if all the other players conspire against her and play perfectly so as to minimize the value A gets at the end.

The minimax strategy would simply be to always move to the state that maximizes your value. It can be shown that when every player follows a minimax strategy,

[3] In fact, if the opponent makes more than one mistake, we can always force a win.

[4] This method runs in O(number of possible states)

the game is in a *Nash Equilibrium*. The two player strategy described earlier is a minimax strategy, since in that game tree, for each node u, $V(u) = V_{f(u)}(u)$ and $V_A(u) + V_B(u) = 0$.

1.5 The Reinforcement Learning Method

Here, we shall give an example of how one could train a Reinforcement Learning algorithm to learn to play a two player game such as Chess or Tic Tac Toe.

We train the algorithm by making it play games (we usually make it play against itself). While playing games, it learns the values of the states.

Let's say our algorithm learns how to play the two player game as the first player. In the game tree, it assigns a value of 0 to all drawn leaf nodes, 1 to all leaf nodes where it wins, and -1 to all leaf nodes where it loses. It never changes the values of these leaf nodes.

During gameplay, it traverses the Game Tree of the game from the root to a leaf. During gameplay, it can either explore (by making a random move) or exploit (by moving to the state with highest value - such moves are called greedy moves). We shall see methods of balancing exploration and exploitation later, when we talk about bandits. Every time it makes a greedy move, we adjust the values of the previous nodes to make it closer. For our strategy, since we need to know only the values of the nodes where the opponent is to play, sometimes we neglect updating the values of the nodes where it is our turn. But here, we will update both. The update rule is:

$$V_A(v) \leftarrow V_A(v) + \alpha(V_A(w) - V_A(v)) \tag{1.3}$$

$$V_A(u) \leftarrow V_A(u) + \alpha(V_A(v) - V_A(u)) \tag{1.4}$$

Here, w is a node we reached by playing a greedy move, and v and u are it's parent and grandparent respectively.

Therefore, our value becomes:

$$V_A(u) = \begin{cases} \max(V_A(v), v \in C(u)), & f(u) = A \\ \mathbb{E}_{opponent}\left[V_A(v_{opponent})\right], & f(u) \neq A \end{cases} \tag{1.5}$$

Therefore, our algorithm eventually learns to play against a particular opponent.

During training our strategy would be to balance exploration (greedy moves) and exploitation (non-greedy moves) in some way, but after training we will only make greedy moves, ie, our strategy would be to just move to the state with highest value for us (same as the Minimax strategy).

We will elaborate more on this in the chapters on Markov Decision Processes and Dynamic Programming.

Chapter 2

Multiarmed Bandits

2.1 Definitions

An *n-armed bandit* is defined in terms of the following situation:

The agent faces a choice between n actions out of which it must choose one. On choosing an action, it gets a reward, that is drawn from some probability distribution that depends on the chosen action. After the choice, the agent faces the same n choices, with the same reward distributions again. We denote the actions made by the agent as A_1, A_2, $\cdots$. We say the situation after k actions is in the $(k+1)^{th}$ timestep. The aim of the agent is to act in such a way as to maximize the total expected reward it gets.

Some key points to note are:

- The agent does not know the probability distributions before hand (if it did, the optimal action would be to choose the action with the highest expected reward every time). It must estimate these distributions.
- The probability distributions do not vary with time (when we remove this restriction, we get non-stationary bandits).
- A bandit has only one state (when we remove this restriction, we get contextual bandits - however, note that even here, we impose the condition that the action taken by the agent does not affect the next state of the bandit).

The presence of only one state in a bandit, allows us to use these to explore methods of balancing exploration and exploitation more easily.

2.2 Balancing Exploration and Exploitation

Let A denote the set of possible actions. For each $a \in A$, let $q(a)$ denote the expected reward for that action. We define the value of each action as $q(a)$. If we knew the values of each state, the optimal policy would be to choose the action with the highest value each time.

During gameplay, our agent estimates the value of each action from it's experiences. At the t^{th} timestep, we denote the estimated value for action a as $Q_t(a)$. There are many ways for estimating $Q_t(a)$. Perhaps the simplest one would be the sample average:

$$Q_t(a) = \frac{\sum_{i=1}^{N_t(a)} R_i(a)}{N_t(a)}, N_t(a) > 0 \tag{2.1}$$

with $Q_t(a)$ assigned an arbitrary value when $N_t(a)$ is 0 ($N_t(a)$ represents the number of times option a has been tried by the t^{th} timestep). By the Law of Large Numbers, as $N_t(a) \to \infty$, $Q_t(a) \to q(a)$.

Noting that $Q_t(a)$ can change only when $N_t(a)$ changes, we can subscript Q with $k = N_t(a)$ instead. After doing this, we see that, for the sample average method

$$Q_{k+1}(a) = Q_k(a) + \frac{1}{k+1}(R_{k+1}(a) - Q_k(a)) \tag{2.2}$$

This is of the form

$$Q_{k+1}(a) = Q_k(a) + \alpha_k(R_{k+1}(a) - Q_k(a)) \tag{2.3}$$

where $\alpha_k = \frac{1}{k+1}$.

We can create many value estimators from the recurrence relation mentioned above by changing α_k.

The condition for $Q_k(a)$ to converge to $q(a)$ with probability 1 (for a stationary distribution) is given by:

$$\sum_{i=1}^{\infty} \alpha_i = \infty \tag{2.4}$$

$$\sum_{i=1}^{\infty} \alpha_i^2 < \infty \tag{2.5}$$

The first condition ensures that α_i are big enough to overcome the arbitrarily decided initial value of Q and overcome the noise in the earlier values of $R_i(a)$, and the second condition ensures that they are small enough to overcome the noise in the later values of R_i.

For non-stationary bandits, we often want to give more weight to the newer values of R_i, as these more closely reflect the actual expectated reward at the current time. Therefore, for non-stationary bandits, we often choose α_i to violate the second condition. A common choice is to keep α_i constant, ie, $\alpha_i = \alpha$.

Now, given we've estimated $Q_t(a)$, we need to find strategies to balance exploration and exploitation.

2.2.1 The Greedy Policy

Here, we always choose the action with the highest estimated value.

$$A_t = \arg\max_{a \in A} Q_t(a) \tag{2.6}$$

Very little exploration happens here. However, we can increase the amount of exploration, at least in the initial stages, by exploiting the fact that the initial values $Q_0(a)$ are set by us. This is known as **Optimistic Initial Value Selection**. Here, we set $Q_0(a) >> q(a)$. Therefore, for the first few actions, $Q_t(a)$ will only decrease. This means that the greedy policy will choose actions that haven't been chosen earlier, encouraging exploration. We can use this method with the other policies as well, including the ones described later.

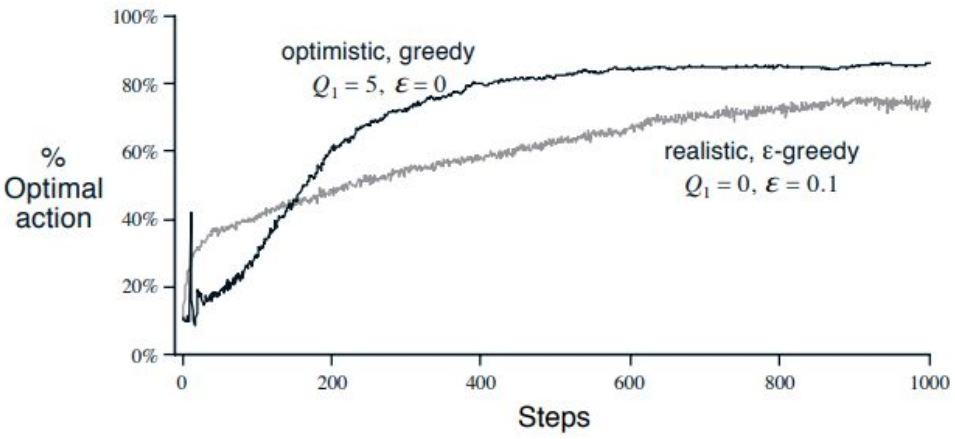

Figure 2.1: Here we have prepared a testbed of 100 identical 10 armed bandits, and we test multiple strategies on this testbed. `% Optimal Action` refers to the percentage of bandits in the testbed choosing the action with the highest value. One can see that the optimistic greedy policy sometimes outperforms realistic versions of even non-greedy algorithms.

2.2.2 The ϵ-Greedy Policy

Here, our policy becomes stochastic.

$$A_t = \arg\max_{a \in A} Q_t(a), \text{with probability } 1 - \epsilon \tag{2.7}$$

$$A_t = \text{random choice from } A, \text{ with probability } \epsilon \tag{2.8}$$

Note that even in the second case, it is possible for $\arg\max_{a \in A} Q_t(a)$ to be chosen. All ties in the first case are broken randomly.

The advantage of this policy is that, as $t \to \infty$, $N_t(a) \to \infty, \forall a \in A$. If we use the sample average to estimate Q_t (and if the bandit is stationary), this means that $\forall a \in A, Q_t(a) \to q(a)$.

If $\epsilon = 0$, then this strategy reduces to the greedy policy. For small values of ϵ, the agent learns slowly, and for larger values it learns more quickly. However, asymptotically, the agent performs better with smaller values of ϵ than with larger values.

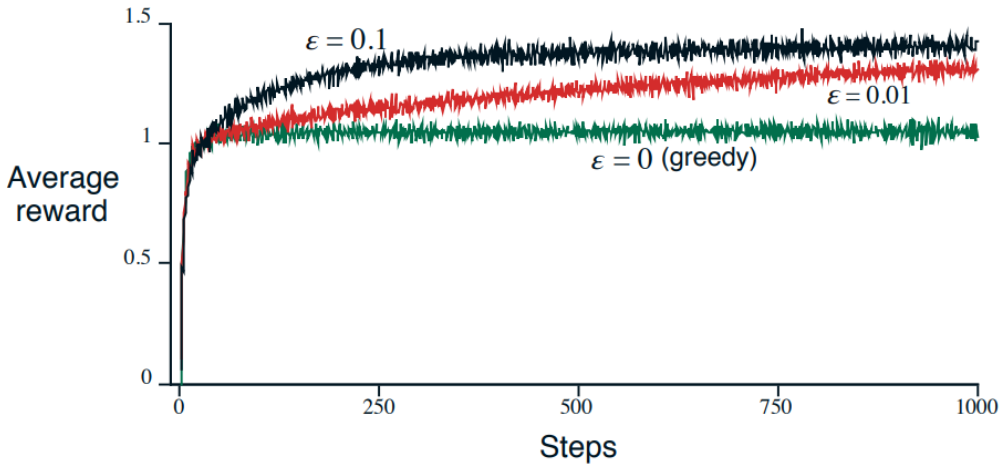

Figure 2.2: We again use the same testbed described earlier and compare greedy and ϵ-greedy policies. `Average Reward` here refers to the average reward each agent obtained at a particular timestep. As we can see, the ϵ-greedy policies perform better than the greedy policies. Among the ϵ-greedy policies, the one with the lower ϵ learns slower, but asymptotically performs better than the one with higher ϵ.

2.2.3 Upper Confidence Bound Action Selection

One problem with ϵ greedy policies are that, during exploration, it gives no preference to actions with higher values, or to relatively underexplored actions over low value,

well explored actions. The UCBA selection method chooses the action with the largest upper confidence bound, where the upper confidence bound of an action is the largest value of $q(a)$ that we are reasonably confident is possible for the action a. For actions tried many times, this upper bound will be close to the estimated Q_t, whereas for untried actions it'll be much larger than it. Formally, speaking the policy is:

$$A_t = \arg\max_{a \in A} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{2.9}$$

When $N_t(a)$ is 0, we set the argument to ∞. The reason we keep the additive term proportional to $\frac{1}{\sqrt{N_t(a)}}$ is because of the Law of Large Numbers, which states that the uncertainty in an estimate after n measurements is proportional to $\frac{1}{\sqrt{n}}$. The $\sqrt{\ln t}$ term appears because that is a function that, although grows extremely slowly, is unbounded. This ensures that as $t \to \infty$, $N_t(a) \to \infty, \forall a \in A$.

The UCBA policy is one of the most effective ways of dealing with stationary bandits[1], however, it does not generalize very well to non-stationary bandits.

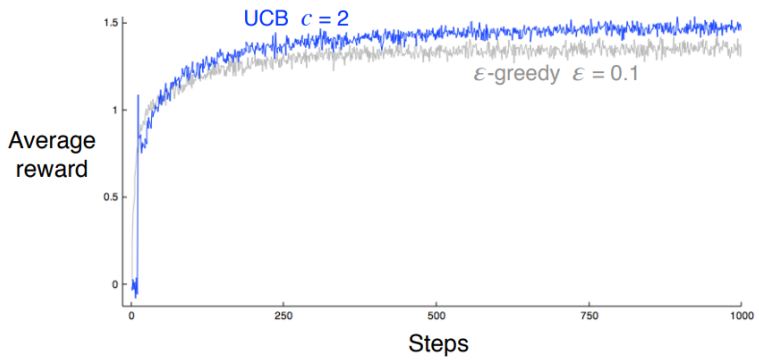

Figure 2.3: We use the same testbed again to compare the UCBA and ϵ-greedy policies. As you can see, the UCBA policy is better.

2.2.4 Gradient Bandits

Here we associate to each action a preference $H_t(a)$, such that the policy becomes:

$$\mathbb{P}(A_t = a) = \pi_t(a) = \frac{\exp H_t(a)}{\sum_{b \in A} \exp H_t(b)}, \forall a \in A \tag{2.10}$$

[1]This is because it can be shown the total regret with the UCBA policy at the t^{th} timestep grows as $O(\log t)$. Here the regret at a particular move is defined as $\max(q(a), a \in A) - q(A_t)$.

This distribution is known as the Softmax distribution. We try to find the values of $H_t(a)$ that maximize $\mathbb{E}_{A_t}\left[\mathbb{E}_{R|A_t}[R]\right] = \mathbb{E}_{A_t}[q(A_t)]$ (the expected reward at each timestep). We find the optimal values of $H_t(a)$ via Gradient Ascent.

$$H_{t+1}(a) = H_t(a) + \alpha \frac{\partial}{\partial H_t(a)} \mathop{\mathbb{E}}_{A_t}\left[\mathop{\mathbb{E}}_{R|A_t}[R]\right] \tag{2.11}$$

However, we cannot calculate this derivative directly as we do not know the value of $\mathbb{E}_{R|A_t}[R] = q(A_t)$. However, it can be shown that

$$\mathop{\mathbb{E}}_{A_t}\left[\mathop{\mathbb{E}}_{R|A_t}\left[R - \overline{R_t}\right](\mathbb{I}(a = A_t) - \pi_t(a))\right] = \frac{\partial}{\partial H_t(a)} \mathop{\mathbb{E}}_{A_t}\left[\mathop{\mathbb{E}}_{R|A_t}[R]\right] \tag{2.12}$$

where $\mathbb{I}$ is the indicator function, which is 1 if it's argument is true, and 0 if it is false, and $\overline{R_t}$ is the average reward obtained so far, ie

$$\overline{R_t} = \frac{\sum_{i=1}^{t-1} R_i}{t-1} \tag{2.13}$$

Therefore, we can use Stochastic Gradient Ascent, where

$$H_{t+1}(a) = H_t(a) + \alpha(R_t - \overline{R_t})(\mathbb{I}(a = A_t) - \pi_t(a)), \forall a \in A \tag{2.14}$$

where A_t is the chosen action and R_t is the obtained reward.

Equation (2.12) remains valid if $\overline{R_t}$ is replaced by any constant X_t. However, putting $X_t = \overline{R_t}$ improves the performance of the algorithm greatly by improving the speed of convergence of the gradient ascent and ensuring that below average rewards are punished immediately, as we can see in the following graph.

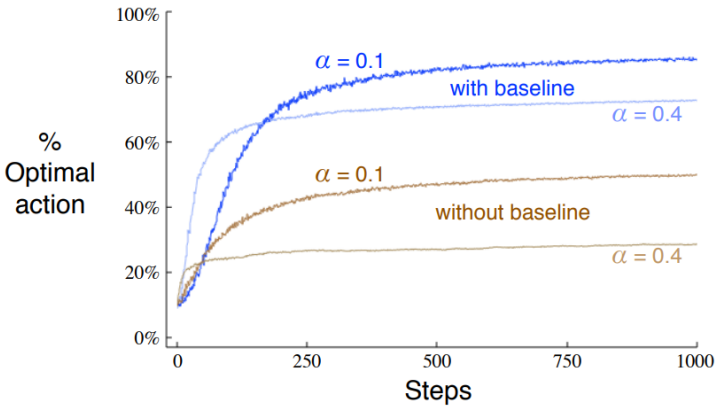

Figure 2.4: We use the previously discussed testbed to compare different types of Gradient Ascent algorithms.

2.2.5 Thompson Sampling

This is a Bayesian approach to the multiarmed bandit problem. It can be used when the class of the reward distributions for the actions is known or can be reasonably guessed. We assume that $R(a) \sim P(R; \theta_a)$, ie, the rewards for each action are drawn from the same family of probability distributions, differing only in parameter θ.

We use Baye's Theorem to get the posterior distribution of θ_a for each $a \in A$, assuming a suitable prior (usually the conjugate prior, to make our calculations easier)

$$P(\theta_a | R_1, R_2 \dots) = \frac{P(R_1, R_2 \dots | \theta_a) P(\theta_a)}{P(R_1, R_2 \dots)} \tag{2.15}$$

We then sample[2] θ_a^* from each posterior distribution. We then choose the action with the highest value ($\mathbb{E}_{\theta_a^*}[R]$), where the values are calculated according to the sampled parameters.

This policy is often even more efficient then the UCBA policy, as this - like the UCBA policy - focuses exploration on actions that are nearly optimal, or actions with a high variance.

A disadvantage of this method is that it is more computationally expensive than the others discussed so far.

2.3 Comparing Policies

Note that all of these policies have parameters that change the effectiveness of the policies. For each policy it is the case that the parameters have optimal values that make the policies as optimal as possible. Considering the policies with their parameters optimized, it seems that for stationary bandits, the Upper Confidence Bound Action (UCBA) Selection Policy[3] and Thompson Sampling[4] perform the best.

However, among these policies, only the ϵ-greedy policies and the gradient bandit policies can be readily generalized to more complex RL problems.

2.4 Contextual Bandits

One of the defining properties of bandits were that they possess only one state. The situation is therefore **non-associative** (ie we don't deal with multiple states), and

[2] We sample θ instead of estimating it from the posterior as estimating can make this policy too greedy

[3] This may be because it can be shown the total regret with the UCBA policy at the t^{th} timestep grows as $O(\log t)$. Here the regret at a particular move is defined as $\max(q(a), a \in A) - q(A_t)$.

[4] Not shown in this graph

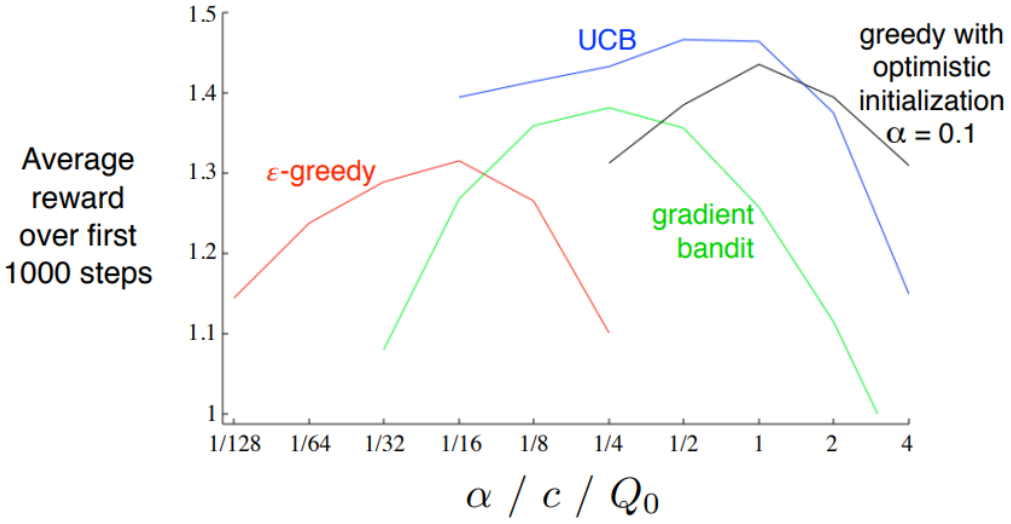

Figure 2.5: We compare all the different policies on our testbed over different values for their parameters. `Average Reward` here refers to the average reward obtained over the first 1000 timesteps. The characteristic inverted U shape of these curves shows that an optimal value exists for the parameters.

therefore we could easily design and test algorithms that balance exploration and exploitation.

When we define contextual bandits, we remove the restriction that there is only one state but still impose that the actions taken by the agent do not affect the next state of the bandit. In this case, we can train separate RL algorithms over each state of the bandit.

For example, say we have k normal bandits. Each time we take an action on a bandit, the bandit is replaced with a different one, in a manner that is **independent** of the action taken. Also, the bandits are **distinguishable**, ie, before we take an action we know which bandit we are dealing with. This situation behaves exactly like a contextual bandit with k states. In this scenario we can learn a separate policy for each bandit.

The conditions of **independence** and **distinguishability** allow us to decouple this problem into k single state problems.

Chapter 3

Markov Decision Processes

3.1 Formalization of the Reinforcement Learning Problem

Here, we will formalize the Reinforcement Learning problem we have stated informally so far, ie, where an agent selects an action based on the state of the environment in order to maximize the cumulative reward it recieves.

We assume the agent and the environment interact with each other in discrete time steps $t = 0, 1, 2 \ldots$. At each time step, the agent recieves the state of the environment ($S_t \in S$) and makes a choice ($A_t \in A(S_t)$) based on that state and earns a reward R_{t+1}[1] and transitions to state S_{t+1}.

The agent follows a policy π, where $\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$[2]. Our agent must learn the optimal policy.

3.2 Return and Discounting

However, we have not yet formally defined what it is that we must optimize. So far we have been talking about optimizing the expected cumulative reward $R_{t+1} + R_{t+2} + \ldots$. However, for tasks that do not terminate eventually, this becomes an infinite sum, which may not converge. Therefore, we define a quantity known as the return G_t in

[1] R_t represents the reward before the t^{th} timestep. We set $R_0 = 0$ by convention.

[2] In general, the agents policy will be a function of time, $\pi_t(a|s) = \mathbb{P}(A_t = a|S_t = s)$ which changes as the agent learns. However, we assume the agent has a stationary policy in order to calculate certain values associated with each state. After each timestep, we will use these values (calculated with the old policy) to update our policy. Towards the end our policy will reach a stationary optimal policy.

different ways for different tasks. It is the expected return $\mathbb{E}(G_t)$ that the agent seeks to maximize at each timestep.

Tasks that always terminate after a finite number of timesteps are known as episodic tasks. For such tasks, we can define G_t as the cumulative reward obtained from the t^{th} timestep onwards.

$$G_t = \sum_{k=0}^{T-k-1} R_{t+k+1} \tag{3.1}$$

For non-episodic tasks however, this sum may not converge. Therefore, we introduce the concept of discounting, where we value rewards obtained later less than those obtained sooner. To be precise, we define G_t as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

where $0 \leq \gamma \leq 1$. If $\gamma < 1$ and the rewards are bounded then this sum will always converge. G_t will satisfy the recurrence $G_t = R_{t+1} + \gamma G_{t+1}$ (this recurrence will prove useful later on).

γ is known as the *discount rate*, and determines how far sighted our agent is. If $\gamma = 0$, then our agent is said to be greedy (or) myopic, as in this $\mathbb{E}(G_t) = \mathbb{E}(R_{t+1})$, and therefore the optimal policy would be to choose the action with the highest expected reward without worrying about future rewards.

We can unify the notation for episodic and non-episodic tasks by having the terminal states of episodic tasks have only a single action, that always gives 0 reward and never changes the state of the environment. We can then define the return for all tasks using equation (3.2), with $\gamma = 1$ for episodic tasks. Note that for episodic tasks, if S_t is a terminal state, $G_t = 0$.

3.3 Markov Decision Processes and the Markov Property

The current state should provide enough information to determine the probability distributions of the rewards and the transition probabilities to the next state (given the next state). At the most it can contain an entire description of the history of the environment upto the present moment. This is quantified mathematically by saying the states and rewards must satisfy the *Markov Property*, ie

$$\mathbb{P}(S_{t+1} = s, R_{t+1} = r | S_t, A_t, R_t, S_{t-1}, A_{t-1} \dots) = \mathbb{P}(S_{t+1} = s, R_{t+1} = r | S_t, A_t) \tag{3.3}$$

for all r, $s \in S$. This means that the distributions of the rewards and the next state depend only on the current state and the action chosen.

For notational convenience, we will use $p(r, s'|s, a)$ to denote this distribution, ie

$$p(s', r|s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{3.4}$$

We say a task is a *complete-knowledge* task if the distribution $p(s', r|s, a)$ is known before hand, and an *incomplete-knowledge* task if it is unknown, and must be learnt by the agent.

$p(s', r|s, a)$ along with the set of states S, set of actions for each state $A(s)$ (for $s \in S$) completely specifies the task at hand. Tasks that can be formulated this way in terms of states and actions are known as **Markov Decision Processes**. Along with the agents policy $\pi(s|a)$, the entire interaction between the Agent and the Environment is specified.

We can define R_{sa} as the expected reward given a state-action pair.

$$R_{sa} = \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a\right] = \sum_{r, s' \in S} rp(s', r|s, a) \tag{3.5}$$

The state transition probability $P^a_{ss'}$ is defined as the probability of transitioning from state s to state s' given action a was chosen.

$$P^a_{ss'} = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a) = \sum_r p(s', r|s, a) \tag{3.6}$$

We can also define $R^a_{ss'}$ as the expected reward given a triplet of the current state s, the chosen action a, and the next state s'.

$$R^a_{ss'} = \mathbb{E}\left[R_{t+1}|S_t = s, S_{t+1} = s', A_t = a\right] = \frac{\sum_r rp(s', r|s, a)}{\sum_r p(s', r|s, a)} \tag{3.7}$$

We can diagramatically represent Markov Decision Processes via Transition Diagrams, which are directed multigraphs in which each node represents a state, and between every two nodes representing states $s, s' \in S$, there is an edge from node s to node s' for each action $a \in A(s)$. These edges are associated with the state transition probabilities $P^a_{ss'}$ and the expected rewards $R^a_{ss'}$. The sum of $P^a_{ss'}$ for all edges leaving a node s is always 1.

3.4 The Bellman Equations

For every state s in the MDP, we can define it's value under the policy π as

$$v_\pi(s) = \mathbb{E}\left[G_t|S_t = s\right] \tag{3.8}$$

v_π is known as the state-value function.

We can also define q_π (known as the action-value function) as

$$q_\pi(s,a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right] \tag{3.9}$$

From the total probability theorem, we can conclude that

$$v_\pi(s) = \sum_{a \in A(s)} q_\pi(s,a)\pi(a|s) \tag{3.10}$$

Now, we can also find q_π in terms of v_π.

$$q_\pi(s,a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right] = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \tag{3.11}$$

Hence,

$$q_\pi(s,a) = \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma\, \mathbb{E}\left[G_{t+1} | S_{t+1} = s', S_t = s, A_t = a\right]) \tag{3.12}$$

By the Markov Property,

$$\mathbb{E}\left[G_{t+1} | S_{t+1} = s', S_t = s, A_t = a\right] = \mathbb{E}\left[G_{t+1} | S_{t+1} = s'\right] = v_\pi(s') \tag{3.13}$$

Therefore,

$$q_\pi(s,a) = \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{3.14}$$

Using equations (3.10) and (3.14) we can get recursive relations for q_π and v_π.

$$q_\pi(s,a) = \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma \sum_{a' \in A(s')} \pi(a'|s') q_\pi(s', a')) \tag{3.15}$$

$$v_\pi(s) = \sum_{\substack{a \in A(s) \\ r,s' \in S}} \pi(a|s) p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{3.16}$$

These equations are known as the **Bellman Equations** for the Action Value and State Value functions respectively. Using these equations, for any policy π, we can find all the $v_\pi(s)$ and $q_\pi(s,a)$ for all s, a up assuming we know the dynamics of the MDP ($p(s', r|s, a)$). This is because we usually know the values of some states of the MDP already, or we can assign values to these based on some heuristic. For example, in episodic MDPs, the terminal states have 0 value. From these states, we can work from the bottom up to find all the values.

3.5 Bellman Optimality Equations

We say a policy π_1 is better than a policy π_2 for an MDP if $\forall s \in S, v_{\pi_1}(s) \geq v_{\pi_2}(s)$ and $\forall s \in S, a \in A(S), q_{\pi_1}(s,a) \geq q_{\pi_2}(s,a)$ (note that due to equations (3.10) and (3.14) these two conditions are equivalent)[3]. It can be shown that there will always exist an optimal policy π^* that satifies $v_{\pi^*}(s) \geq v_\pi(s)$ and $q_{\pi^*}(s,a) \geq q_\pi(s,a)$, $\forall s \in S, \forall$ policies π. Although there may be many optimal policies, it is easy to show that v_{π^*} and q_{π^*} are unique[4]. To show this uniqueness, we write $v_* = v_{\pi^*}$ and $q_* = q_{\pi^*}$. Clearly, $\forall s \in S, a \in A(S)$ $q_*(s,a) = \max_\pi(q_\pi(s,a))$ and $v_*(s) = \max_\pi(v_\pi(s))$.

We can show that the optimal value functions must satisfy the following equations, which are known as the **Bellman Optimality Equations**.

$$q_*(s,a) = \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma \max_{a \in A(s')} q(s', a)) \tag{3.17}$$

and

$$v_*(s) = \max_{a \in A(s)} q_*(s,a) = \max_{a \in A(s)} \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v_*(s')) \tag{3.18}$$

We can intuitively justify this by stating that in the optimal policy we will always choose an action that leads to maximum return, ie,

$$v_*(s) = \max_{a \in A(s)} q_*(s,a) \tag{3.19}$$

Using this along with equation (3.14), we get the Bellman Optimality Equations, and we can also show that and policy π satisfying $\pi(a|s) \neq 0$ only if $a \in \arg\max_{a \in A(s)} q_\pi(s,a)$ is an optimal policy (the optimal policy is greedy with respect to the action value function).

A formal proof[5] would go as follows: Assume we have some function $v(s)$ that satisfies

$$v(s) = \max_{a \in A(s)} \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v(s')) \tag{3.20}$$

Now, clearly (by equation (3.16)) this corresponds to an action-value function for a policy π such that $\pi(a|s) \neq 0$ only if $a \in \arg\max_{a \in A(s)} \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v(s'))$.

[3] There may be policies which are not comparable, ie Now, if $v_{\pi_1}(s) \geq v_{\pi_2}(s)$ for some s and $v_{\pi_1}(s) < v_{\pi_2}(s)$ for other s. We have defined in this way a *partial order* on policies.

[4] If v_1 and v_2 are the state value functions for two optimal policies, then $\forall s \in S$, $v_1(s) \geq v_2(s)$ and $v_2(s) \geq v_1(s)$.

[5] This part is not essential reading for the project

By equation (3.14), the corresponding action value function would be $q(s,a) = \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v(s'))$, ie $\pi(a|s) \neq 0$ only if $a \in \arg\max_{a \in A(s)} q(s,a)$.

Now, we shall prove that $v(s) \geq v_{\pi'}(s), \forall s \in S$ for all policies π'. This makes π an optimal policy[6], which makes v **the** optimal state value function. Therefore, due to uniqueness, the optimal state value function must satisfy the equation (3.20), which is nothing but the Bellman Optimality Condition for the state value function, from which we can derive the Bellman Optimality Condition for the action value function (using equation (3.14) along with the fact that $v(s) = \max_{a \in A(s)} q(s,a)$).

To prove that $v(s) \geq v_{\pi'}(s), \forall s \in S$, for all policies π', we proceed by induction.

For the terminal states $v(s) = v_{\pi'}(s) = 0$, therefore, $v(s) \geq v_{\pi'}(s)$. Now, assume for some state s, every s' with $p(s', r|s, a) > 0$ for some r and some a has $v(s') \geq v_{\pi'}(s')$. Now,

$$v(s) = \max_{a \in A(s)} \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v(s')) \geq \max_{a \in A(s)} \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v_{\pi'}(s')) \quad (3.21)$$

Therefore,

$$v(s) \geq \sum_{\substack{a \in A(s) \\ r,s' \in S}} \pi'(a|s) p(s', r|s, a)(r + \gamma v_{\pi'}(s')) = v_{\pi'}(s) \quad (3.22)$$

Hence,

$$v(s) \geq v_{\pi'}(s) \quad (3.23)$$

Therefore, by the inductive principle $\forall s \in S, v(s) \geq v_{\pi'}(s)$. Using this, we can prove the Bellman Optimality Conditions via the method described above.

Note, that we here we have proved that the Bellman Optimality Equations impose a sufficient condition on the value functions for them to be optimal. The uniqueness of the value functions leads to this being a necessary condition as well.

3.6 Solving the Bellman Optimality Conditions

In principle, if the dynamics of the MDP ($p(s', r|s, a)$) are known (the MDP is complete knowledge), then we can solve the Bellman Optimality Equations from the bottom up, and use these to find the optimal policy, ie, we can solve the Reinforcement Learning Problem.

[6]This means that any deterministic policy that is greedy with respect to it's own action value function is an optimal policy, as it is trivial to show that any such policy must satify the Bellman Optimality Equations.

The two principal difficulties with this are that, often $p(s', r|s, a)$ are not known beforehand and must be learnt, and that in most situations the number of states is too large for the previously described method to be computationally feasible. Therefore, we develop various methods and hueristics in order to approximately solve these equations.

Chapter 4

Dynamic Programming

4.1 Introduction

Dynamic Programming[1] Algorithms are a collection of iterative algorithms that can be used to compute the optimal policy for **complete-knowledge**[2] finite MDPs.

For complete knowledge finite MDPs, it is possible to explicitly solve the Bellman Optimality Equations to find the optimal value and action value functions. The optimal policy would then be greedy with respect to the optimal action value function. However, solving the Bellman Optimality Equations explicitly may be computationally infeasible, especially when the number of possible states is extremely large. Hence, we develop methods that iteratively converge to the Optimal Policy[3].

Broadly speaking, this class of algorithms starts out with an arbitrarily chosen policy and value function. The value function is then made to fit the policy more accurately, after which the policy is changed to be greedy with respect to this value function. This process will eventually converge to the optimal policy and value functions for the MDP[4].

4.2 Policy Evaluation

Policy Evaluation refers to a class of algorithms that can be used to find the value functions for a given policy π. This can also be done explicitly by solving the Bellman

[1] The name arises due to the fact that most algorithms in this class cache the values of the states encountered instead of recomputing them each time.

[2] $p(s', r|s, a)$ is known to the agent.

[3] An analogy with Gradient Descent may be made here. This analogy can be extended further as we discuss variations on Dynamic Programming.

[4] Note that the optimal policy and value functions are fixed points of this algorithm, as the optimal policy is greedy with respect to it's own value function.

Equations, but, as mentioned earlier, this becomes computationally infeasible when the number of possible states ($|S|$) is extremely large. Therefore, here too we use iterative algorithms that converge to the actual value functions.

These algorithms function by converting the Bellman Equations into update rules. We start with a guess value function v_0[5]. We then repeatedly apply the following update rule:

$$v_{k+1}(s) = \sum_{\substack{a \in A(s) \\ r, s' \in S}} \pi(a|s) p(s', r|s, a)(r + \gamma v_k(s)), \forall s \in S \tag{4.1}$$

It can be shown that $\lim_{k \to \infty} v_k = v_\pi$. Note, that v_π itself is a fixed point of this update rule (it remains unchanged by this update), due to the Bellman Equations. This algorithm is known as iterative policy evaluation. Once we find v_π, we can find q_π via the Bellman Equations:

$$q_\pi(s, a) = \sum_{r, s' \in S} p(s', r|s, a)(r + \gamma \sum_{a' \in A(s')} \pi(a'|s') q_\pi(s', a')) \tag{4.2}$$

It is also possible to iteratively find q_π instead of v_π using it's Bellman Equation as an update rule, ie we start with an initial guess q_0, and update it according to:

$$q_{k+1}(s, a) = \sum_{r, s' \in S} p(s', r|s, a)(r + \gamma \sum_{a' \in A(s')} \pi(a'|s') q_k(s', a')) \tag{4.3}$$

Here also $\lim_{k \to \infty} q_k = q_\pi$, and we can therefore find v_π using $v_\pi(s) = \sum_{a \in A(s)} \pi(a|s) q_\pi(s, a)$. This algorithm is less efficient than the previously described one, as during each update we must iterate over the space of states and the space of actions ($S \times A$), rather than S alone.

Each update to the value of a state s is known as a backup to s. We can do our backups in-place, ie, once we calculate the new value of $v(s)$ via the update rule, we immediately use this new value to update $v(s')$ for other states s'. This generally leads to faster convergence as we use the updated information as soon as it becomes available. Since, in this algorithm, to go from v_k to v_{k+1}, we update $v(s)$ for every state s, this update is known as a full backup. Full backups run in $O(|S|^2)$ and therefore can be extremely computationally expensive. *Asynchronous Dynamic Programming* and *Generalized Policy Iteration* techniques (discussed later) provide ways for us to manage this cost.

[5]Note that the $v_0(s)$ must be 0 for all terminal states s. This ensures that $v_k(s) = 0, \forall s \in S^T$. This basically means our initial guess must follow the boundary conditions that v_π follow $\forall \pi$.

4.3 Policy Improvement

We can use the computed value functions to improve our policy. This is done by changing our policy to be greedy with respect to the newly computed value function, ie

$$\pi'(s) = \arg\max_{a \in A(s)} q_\pi(s, a) \tag{4.4}$$

with ties broken arbitrarily[6].

This new policy will be better than (or at least as good as) the previous policy. This can be formalized via the *Policy Improvement Theorem*.

4.3.1 The Policy Improvement Theorem

The statement of the theorem is as follows:

If π and π' are two policies such that

$$\mathop{\mathbb{E}}_{a \sim \pi'} [q_\pi(s, a)] \geq v_\pi(s), \forall s \in S \tag{4.5}$$

then

$$v_{\pi'}(s) \geq v_\pi(s), \forall s \in S \tag{4.6}$$

ie, π' is at least as good a policy as π.

The proof of this theorem is as follows:

Firstly, note that $\mathbb{E}_{a \sim \pi'} [q_\pi(s, a)] = \sum_{a \in A(s)} \pi'(a|s) q_\pi(s, a)$. Therefore, our inequality becomes

$$\sum_{a \in A(s)} \pi'(a|s) q_\pi(s, a) \geq v_\pi(s), \forall s \in S \tag{4.7}$$

Our proof is by induction. Note that for all terminal states s, $v_{\pi'}(s) = v_\pi(s) = 0$. For some state s, assume that every s' with $p(s', r|s, a) > 0$ for some r, a has $v_{\pi'}(s') \geq v_\pi(s')$. Now,

$$v_{\pi'}(s) = \sum_{\substack{a \in A(s) \\ r, s' \in S}} \pi'(a|s) p(s', r|s, a)(r + \gamma v_{\pi'}(s)) \tag{4.8}$$

Since, $v_{\pi'}(s) \geq v_\pi(s)$

$$v_{\pi'}(s) \geq \sum_{\substack{a \in A(s) \\ r, s' \in S}} \pi'(a|s) p(s', r|s, a)(r + \gamma v_\pi(s)) \tag{4.9}$$

[6]Note that π' here is a deterministic policy, ie, $\mathbb{P}(A_t = a|S_t = s)$ is 1 for $a = \pi'(s)$ and 0 for all other $a \in A(s)$. We abuse notation for deterministic policies by using π to stand for both $\mathbb{P}(A_t = a|S_t = s)$ ($\pi(a|s)$) and for the action that is chosen ($\pi(s)$).

The RHS here can be rewritten as $\sum_{a \in A(s)} \pi'(a|s) \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v_\pi(s))$ which is nothing but $\sum_{a \in A(s)} \pi'(a|s) q_\pi(s, a)$.

Therefore,

$$v_{\pi'}(s) \geq \sum_{a \in A(s)} \pi'(a|s) q_\pi(s, a) \geq v_\pi(s) \tag{4.10}$$

Hence, $v_{\pi'}(s) \geq v_\pi(s)$. By induction, this means that $v_{\pi'}(s) \geq v_\pi(s), \forall s \in S$.

4.3.2 Improving our Policy

If we change our policy in the manner described earlier, ie

$$\pi'(s) = \arg \max_{a \in A(s)} q_\pi(s, a), \forall s \in S \tag{4.11}$$

then for all states s,

$$\mathop{\mathbb{E}}_{a \sim \pi'} [q_\pi(s, a)] = \max_{a \in A(s)} q_\pi(s, a) \geq \sum_{a \in A(s)} \pi(a|s) q_\pi(s, a) = v_\pi(s) \tag{4.12}$$

and therefore, by the Policy Improvement Theorem,

$$v_{\pi'}(s) \geq v_\pi(s), \forall s \in S \tag{4.13}$$

ie, π' is at least as good as π.

If π' is exactly as good as π, ie

$$v_{\pi'}(s) = v_\pi(s), \forall s \in S \tag{4.14}$$

then the inequality in equation (4.12) must be an equality. Therefore,

$$v_\pi(s) = \max_{a \in A(s)} q_\pi(s, a) = \max_{a \in A(s)} \sum_{r,s' \in S} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{4.15}$$

ie, v_π (and therefore $v_{\pi'}$) both satisfy the Bellman Optimality Equations, and therefore both π and π' are optimal policies.

This means that, given the value function of a policy, we can modify the policy in such as a way as to make it strictly better, unless it is already an optimal policy (in which case it can only change to another optimal policy).

This leads to the idea of *policy iteration*.

4.4 Policy Iteration

Policy Iteration refers to a class of algorithms that alternate Policy Evaluation and Policy Improvement in order to find the optimal policy for the MDP.

We start with an arbitrary policy π_0, and compute it's value function v_{π_0} via policy evaluation. The update rule we then follow is:

- Given value function $v_{\pi_{k-1}}$, get policy π_k that is greedy with respect to $v_{\pi_{k-1}}$ by policy improvement.
- Given policy π_k, compute v_{π_k} via policy evaluation.

We keep doing this until we reach an optimal policy π_*, which occurs when there is no improvement in the policy between the k^{th} and $(k+1)^{th}$ iterations.

Doing this, we get a monotonically increasing sequence of policies and value functions during policy iteration which eventually converges to an optimal policy and the optimal value function.

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

However, Policy Iteration can be extremely computationally intensive, as it involves multiple iterations of policy evaluation, each of which involves multiple iterations of full backups, each of which involves a traversal of over each possible state of the MDP.

Therefore, we develop methods to reduce the amount of computation we need to do. One such method is *value iteration*.

4.5 Value Iteration

The key idea in value iteration is that we don't need to find the exact value function v_{π_k} in order to get an improvement by making our new policy greedy with respect to the value function. We can just do a few full backups and then move on to policy improvement.

In value iteration, we do exactly one full backup, and move on to the policy improvement step immediately afterwards. The update rules then become:

$$v_{k+1}(s) = \max_{a \in A(s)} \sum_{r, s' \in S} p(s', r|s, a)(r + \gamma v_k(s)) \tag{4.16}$$

where, at every iteration, $\pi_k(s) = \arg\max_{a \in A(s)} \sum_{r, s' \in S} p(s', r|s, a)(r + \gamma v_k(s))$.

Note, that equation (4.16) is nothing but the Bellman Optimality Condition converted into an update rule, and therefore the optimal value function will be a fixed point of this update rule.

4.6 Asynchronous Dynamic Programming

Sometimes, for large state spaces, the cost of doing even a single full backup every iteration may be too large. In these situations, it is often the case that the most states are very unlikely to occur in the agent's experience of playing the game. Therefore, in asynchronous dynamic programming, we update only a subset of the states[7], ideally the ones that are most relevant.

Asynchronous Dynamic Programming is often done by training the agent while it is experiencing the MDP. We backup only the states that the agent has encountered so far. This causes learning to be focused on the most relevant states.

4.7 Generalized Policy Iteration

Starting from Policy Iteration, our strategy to find the optimal policy has been to both update the value function for the states (policy evaluation) and to update the policy to make it greedy with respect to the current value function (policy improvement). Such algorithms are known as Generalized Policy Iteration algorithms. To make these algorithms less computationally intensive we have spread the policy evaluation around and intermixed it more with policy improvement. This allows us to not have to wait for long policy evaluations in order to improve our policy.

All Generalized Policy Iteration algorithms will eventually converge to an optimal policy, as, by the Bellman Optimality Equations, these policies are greedy with respect to their own value functions.

[7] An analogy may be drawn with stochastic gradient descent here.