ECE 457C Spring 2022 Course Review

Saif K.

August 1, 2022

Chapter 1

Reinforcement Learning

This section aims to describe what reinforcement learning is and is not.

RL is simultaneously:

- A class of problems
- A set of solution methods that work well on the problem
- The field that studies the above two things

RL involves an **agent** interacting with an **environment** over time to achieve some **goal**. It does so by balancing the **exploitation** of actions it knows to be good with the **exploration** of actions it has less information about, but which may be more fruitful in the long run.

RL is **not** supervised learning. The latter involves:

- A training set of appropriate labels given data, generated by some expert
- An agent which **generalizes** from this to incorporate new data

In 'uncharted territory,' the first point is impractical, and the second is unlikely to achieve the goal.

RL is **not** unsupervised learning. The latter involves inferring some structure in unlabeled data. This may be *useful* in achieving the goal in some environment, but is not *sufficient* to do so.

Chapter 2

Multi-Armed Bandits

The k-armed bandit problem involves k slot machines each of which produces some reward according to a (hidden) probability distribution. The goal of the agent is to maximize its reward by selecting an arm to pull in each timestep.

Because the rewards for machine j follow a probability distribution (which may or may not be **stationary**), we can define the 'value' of selecting it as

$$q_*(a_j) = \mathbb{E}[R_t | A_t = a_j] \quad \forall j \in \mathbb{Z}, 0 \leq j < k \tag{2.1}$$

Note the analogy to the optimal value function Equation 3.7

2.1 Action-Value Methods

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \tag{2.2}$$

One way to estimate q_*, shown in Equation 2.2, is to replace the expectation in Equation 2.1 with an average over the actions taken. Note that this is a simple example of a **Monte-Carlo** method described in chapter 5. This estimate is not necessarily the best, but it is guaranteed to converge in the limit so long as all actions are selected infinitely often.

Given an estimate, how should we perform action selection?

- Greedy: $A_t = \text{argmax}_a(Q_t(a)) = A_t^*$
- Epsilon-Greedy: choose A_t^* with probability $1 - \epsilon$, otherwise randomly select **any** action

2.2 Incremental Implementation

$$Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n) \tag{2.3}$$

Where Q_n is the estimate $Q_t(a)$ for the action selected at time t, which gave reward R_n. This is simply a running average, but demonstrates the basic form that many tabular algorithms use to update the Q estimate:

$$\text{New Value} \leftarrow \text{Old Value} + \text{StepSize} \cdot (\text{Target} - \text{Old Value}) \tag{2.4}$$

- The value in brackets (Target - Old) is called the **error** term
- StepSize may **not** be constant over time, for example in Equation 2.3 it depends on the number of times the action has been taken by time t

2.3 Nonstationary Problems

$$Q_{n+1} = Q_n + \alpha \left(R_n - Q_n \right) \tag{2.5}$$

A constant step-size parameter weights recent rewards more than old rewards and is thus a neat way of performing an update in a non-stationary environment. As an exercise, if you express Equation 2.5 as a sum you will see that reward R_i is weighted by $\alpha(1-\alpha)^{n-i}$.

2.4 Optimistic Initial Values

- A trick to encourage exploration - set all the initial estimates high so that the agent will only switch off them if it continually gets 'disappointed' by them
- Works well on stationary k-bandit, but is **not** a general way to encourage exploration

2.5 Upper-Confidence Bound

$$A_t = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{2.6}$$

The idea here is to improve on ϵ-greediness by favouring actions that are more likely to be better in the ϵ case, which depends on two things:

- The current estimate of the action
- The uncertainty in that estimate

Correspondingly, UCB selection has the following properties:

- $N_t(a)$ indicates the number of times action a has been selected by time t
  - The higher this is, the *less* uncertainty there is in the estimate $Q_t(a)$, hence it is in the denominator
  - By convention, if $N_t(a) = 0$, we consider it a maximizing action
- $\ln t$ increases over time *monotonically*, thus increasing the probability of selection for actions that are not selected over time. This ensures that all actions will eventually be selected regardless of the initial estimates
- c is (confusingly) the confidence level - the higher it is the less confident we are in estimates, since we weight the variance term higher
- It is called Upper-Confidence bound since the $c\sqrt{\cdot}$ term is the upper bound on the estimate of a

2.6 Gradient-Bandit Algorithms

$$P[A_t = a] = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \equiv \pi(a) \tag{2.7}$$

In this approach, we simply represent the policy by recording a number $H \in \mathbb{R}$ known as the 'preference' for each action, which has no interpretation in terms of the reward. The function in Equation 2.7 simply turns these values into a valid probability distribution in a nice way - this is also known as the *softmax* distribution. The question is how to learn these values, and the answer is given by the **stochastic gradient ascent** algorithm:

$$H(A_t) \leftarrow H(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)) \tag{2.8}$$

$$H(a) \leftarrow H(a) + \alpha(R_t - \bar{R}_t)\pi_t(a) \qquad \forall a \neq A_t \tag{2.9}$$

Where $\bar{R}_t$ is the average of all the rewards up to but not including t. This is an instance of a stochastic gradient algorithm that uses a **baseline**, which comes up again in section section 9.3

Chapter 3

Finite MDPs

Markov Decision Processes, or MDPs, are a formalism for describing the types of problems we would like to solve in RL. A finite MDP involves an **agent** interacting with an **environment**. There is a finite set $\mathcal{S}$ of states that the environment can be in (which includes the state of the agent). There is a finite set $\mathcal{A}$ (which may be dependent on the state) of actions the agent can take. There is a finite set $\mathcal{R} \subset \mathbb{R}$ of rewards the agent will receive for taking that action. Time evolves in discrete steps $t = 0, 1, 2...$, and at each point the agent takes some action. The **trajectory** of an agent is thus:

$$S_0, A_0, R_1, S_1, A_1, R_2, ... \tag{3.1}$$

The **Markov property** states that the distribution of S_t and R_t depend only on the previous state and action. That is, if an environment is **'markovian'**, we can write down a distribution known as the **dynamics** of the environment:

$$p(s', r|s, a) \equiv Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{3.2}$$

It is a fundamental hypothesis of RL that everything we mean by 'goal' and 'purpose' can be thought of as the maximization of $\mathbb{E}[\sum_{t=0} R_t]$. There is considerable power in selecting exactly how to define a reward to achieve some goal. For example, when trying to win a game, we should give a reward for winning. When trying to minimize the steps taken to win the game, there should be a negative reward for each step it takes.

3.1 Return

Many tasks are **episodic** in that the task ends at $t = T$. In these cases, we define a **return**. In the episodic case this can simply be the sum of rewards until the episode end, however in the case of continuing tasks we need the sum to be finite and thus add a factor $0 \leq \gamma \leq 1$ known as the **discounting factor**:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1} \tag{3.3}$$

3.2 Policies and Value Functions

Almost all RL algorithms involve learning value functions which indicate *how good* it is to be in a given state or perform an action in a given state. This is defined in terms of the expected return (or expected future discounted reward) that we would get in a given situation.

Since the expected return depends on how the agent will act in the future, reward is also computed with respect to a **policy**, which can be described as a function $\pi(a|s)$ that gives the probability of taking an action in a state. The policy an agent uses may change over time.

The state value is given by

$$v_\pi(s) \equiv \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{3.4}$$

The state-action (or just action) value is given by

$$q_\pi(s,a) \equiv \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{3.5}$$

Note that in the expectation, the sequence R_t depends on π_t (or just π if the policy is constant) through the **dynamics** (Equation 3.2). The dynamics may or may not be known, and this will be very important when it comes to algorithms that need to compute either of these value functions.

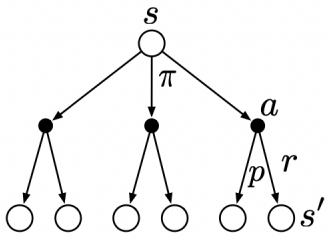
3.3 Bellman Equation

Intuitively, the state value depends on the reward we get for moving away from this state and then on the value of the state we end up in. This intuition leads us to derive the **Bellman equation**:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \tag{3.6}$$

3.4 Backup Diagrams

We can visually describe Equation 3.6 with the following diagram:



Backup diagram for v_π

The diagram expresses how the value of a state is related to the value of subsequent (possible) states. Each action can lead to one of many possible states and the value of the root state is the weighted average of these values. It is called a backup diagram because of how information flows from successor states back to the root state.

3.5 Optimal Policies and Value Functions

We need some way to compare policies; this is done through the ordering $\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \forall s \in \mathcal{S}$. For finite MDPs there is a finite number of policies and thus some policy π_* such that $\pi_* \geq \pi \forall \pi$. There may be multiple, but they all share the same state-value function (due to uniqueness under Bellman) called v_*. Similarly, there exists a unique q_*:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{3.7}$$

Using Equation 3.6:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_*(s') \right] \tag{3.8}$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right] \tag{3.9}$$

Intuitively, this is because the optimal value is the expected return of taking the optimal action from s plus the optimal value of the state you would end up in. This is represented in a backup diagram by drawing an arc across the edges to represent 'max'.

Once we have v_*, the optimal policy is easy to find - any policy that assigns nonzero probabilities only to the actions that give the maximum in Equation 3.6. Similarly, if we have q_* the optimal policy is that which assigns nonzero probability only to actions a_* such that $a_* = \text{argmax}_a q(s, a)$.

Problems with directly computing v_*:

- Dynamics $p(s', r|s, a)$ may not be known
- If $|\mathcal{S}|$ is large then it may be infeasible to solve all the linear equations
- The problem may not be Markovian

Chapter 4

Dynamic Programming

Assuming we have a perfect model of the environment, we can essentially use the Bellman equation Equation 3.6 as an update rule for some estimate of v_* or q_* respectively. This is the theoretical basis for many practical algorithms even though it is not very practical on its own.

There are three problems that need to be solved. The first **prediction** or **evaluation**: given a policy π, what is v_π? Second is **improvement** given π and v_π how do we derive $\pi' \geq pi$? Finally the **control** or **iteration** problem - how do we manage the process of evaluation and iteration?

4.1 Evaluation (or Prediction)

The most straightforward method is called **iterative policy evaluation**. We initialize $v_0(s)$ arbitrarily for all states except set the value of terminal states to 0 and then do:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{4.1}$$

This is called a 'sweep' through the state space. The update can be done with two value vectors or in-place; in-place usually converges faster. We detect convergence by computing $\delta = \max(\delta, |v_{k+1}(s) - v_k(s)|$ for each state and stopping when it goes below a threshold.

4.2 Policy Improvement

Note that this section only considers deterministic policies, but analogous results are possible for stochastic ones. If we have $\pi, v_\pi(s)$ we can apply the **policy-improvement theorem** (Equation 4.2) to prove the following intuitive result: If it would be better to select some action a in state s once and thereafter follow π than it would be to follow π forever, then it would be better to always select a in state s.

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \;\; \forall s \in \mathcal{S} \implies \pi' \geq \pi \tag{4.2}$$

Therefore, we can construct a new policy π' by being *greedy* with respect to q_π and are guaranteed that this is better than the original policy.

4.3 Policy Iteration (or Control)

Combining the processes in the previous two sections allows us to converge towards an optimal policy. Simply start with any policy, evaluate it, improve it, evaluate the improved policy, and continue. We can check for policy

convergence by seeing if any actions have changed (in the deterministic case) or that the KL divergence has bottomed out (in the stochastic case).

A special case of policy iteration, known as **value iteration** combines the evaluation and improvement by replacing π in Equation 4.1 with a max over the actions and performing only one state-space sweep (or, equivalently, by turning the Bellman *optimality* in Equation 3.8 into an update):

$$v_{k+1}(s) \equiv \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{4.3}$$

4.4 Asynchronous DP

Asynchronous DP is simply DP that does not perform systematic sweeps over the state space. Instead, it updates the values of states in some arbitrary order - maintaining convergence so long as it eventually keeps updating every state. For example, asynchronous value iteration uses Equation 4.3 only once in each iteration step (e.g. on a state that was actually visited by some random policy). The main benefit is that we have some more flexibility and can focus on states that are more important - and update the policy while actually experiencing the MDP.

4.5 Generalized Policy Iteration

GPI is an abstraction of the processes described above. In GPI we have two processes: one updating the value function for a policy, and the other making the policy greedy with respect to the current value function.

Chapter 5

Monte-Carlo Methods

Monte-Carlo methods allow us to learn value functions and compute optimal policies *without* the dynamics of the environment. We average returns received along trajectories and use this information to update state value functions.

5.1 Prediction

Note that the expectation of some function over random variables can be approximated by the sample mean of that function over samples of those random variables. In this case, the random variables are the sequence R_t whose distribution depends on π and the function is the (discounted) return G_t in some starting state s_0:

$$V_\pi(s_0) = \mathbb{E}[G_t | S_t = s_0] = \frac{1}{N} \sum_{i=1}^{N} G_i \tag{5.1}$$

Where G_i is the return we got in iteration i. An iteration involves running through the MDP according to π starting at s_0 until we reach a terminal state.

When we perform an iteration, we may visit s_0 multiple times. In **first-visit** MC we get exactly 1 G_i in each iteration, corresponding to the overall return of that iteration (which started in s_0). In **every-visit** MC we may get multiple G_i in the iteration, corresponding to the returns following each visit to s_0 in the iteration. **Both** first and every visit MC converge to v_π so long as we visit every state infinitely often - this is because the sample mean is an unbiased estimator of G_t.

The backup diagram for Monte-Carlo would be the full trajectory up to the terminal state and starting at s_0.

Monte-Carlo methods do not **bootstrap** because the estimate for one state does not depend on the estimate for any other state (it only uses the actual experienced rewards starting at a given state).

Advantages over DP:

- Even when we know the full description of the environment, the exact probabilities $p(s', r|s, a)$ may be difficult to compute (e.g. in blackjack). Monte-Carlo does not require these
- Monte-Carlo can learn from actual and simulated experience
- The computational expense does not depend on $|\mathcal{S}|$

5.1.1 Exploring Starts

Deterministic policies may make it difficult to use MC methods to compute $q_*(s, a)$ as they cannot visit every possible action in a given state. To address this, we can specify that episodes start in a random (s_0, a_0) pair to ensure that every state and action pair is visited infinitely often. This is just a trick, and the better way to do this would be to use stochastic policies.

5.1.2 On-Policy Prediction

Removing the assumption of exploring starts introduces the concepts of **off-policy** and **on-policy** methods. In **on-policy** methods, we still evaluate $v_p i$ but ensure that we still visit every action. For example, we may use 'eternally soft' $pi(a|s) > 0$ policies. We may also use ϵ-soft policies (note that $A^* = \text{argmax}_a Q(s,a)$):

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}(s)|} & a \neq A^* \\ 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & a = A^* \end{cases} \tag{5.2}$$

Note that using an ϵ-greedy policy in this method will converge to the optimal *e-soft* policy (**not** the overall optimum).

5.1.3 Off-Policy Prediction - Importance Sampling

In **off-policy** control, we seek to learn the values for an optimal policy, but must behave in a non-optimal way in order to explore all actions. On-policy is a compromise - it behaves close to optimally to avoid addressing this problem.

We have *two* policies: the behaviour policy $b(a|s)$ and the target policy $\pi(a|s)$. b is usually some exploratory policy while π is usually the deterministic greedy policy. Note the following

$$\mathbb{E}_p[f(X)] = \sum_{x \in \text{Val}(X)} p(x) f(x) = \sum_{x \in \text{Val}(X)} q(x) \frac{p(x)}{q(x)} f(x) = \mathbb{E}_q \left[\frac{p(X)}{q(X)} f(X) \right] \tag{5.3}$$

Similarly, we can compute $v_\pi(s)$ using $v_b(s)$ using:

$$\rho_{t:h} \equiv \prod_{k=t}^{\min(h, T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{5.4}$$

$$v_\pi(s) = \mathbb{E}_b \left[\rho_{t:T-1} G_t | S_t = s \right] \tag{5.5}$$

By behaving according to b and computing the expectation in Equation 5.5 using sample means weighted by the product, we can converge to v_π.

5.2 Improvement

We simply make π greedy with respect to Q_π as before.

5.3 Control

On policy control is the same idea as GPI. Off-policy control is achieved in much the same way, but using the estimate of V_π as discussed in subsection 5.1.3

5.4 General Remarks

Monte-Carlo is sort of like DFS. Information flows from root states to leaf nodes one step at a time, so the best case for updating the value of the tree is proportional to the depth of the tree.

Chapter 6

TD-Learning

Temporal-Difference (TD) learning combines concepts from DP and MC. TD updates value estimates based on other estimates - that is, it **bootstraps** like DP. TD does not require the environment dynamics and can learn from experience like MC.

6.1 TD Prediction

The simplest update, known as TD(0) or **one-step** TD:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{6.1}$$

This is in contrast with Equation 5.1 (the target is $R_{t+1} + \gamma V(S_{t+1})$ instead of G_t). DP is an estimate because it uses the estimate $V(S_{t+1})$. MC is an estimate because it estimates $\mathbb{E}[G_t]$ using the sample mean. TD is an estimate for both reasons. It combines the **sampling** of MC with the **bootstrapping** of DP.

TD is guaranteed to converge to v_π provided sufficiently small α. TD is implemented as an **online** algorithm and thus can update values more often. It is not clear whether TD converges faster than MC in theory, but TD methods are generally preferable.

6.2 On-policy TD Control: SARSA

- On-policy TD: estimate q_π for current π while updating π
- Convergence: depends on how π relates to Q
  - π must converge to greedy in the limit (can be acheived by slowly decreasing ϵ)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{6.2}$$

6.3 Off-policy TD Control: Q-Learning

- Off-policy TD: Q-learning directly learns q^* while behaving via π
- Convergence: behaviour π eventually visits all (S,A) infinitely often
  - ϵ-greedy not the only possibility

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{6.3}$$

The idea here is that we run this algorithm for N episodes, or until we detect that Q has converged. Then we have 'learned' the optimal policy π_* and can begin acting according to it. This is important as this is also what DQN (see section 9.2) does.

6.4 Expected SARSA

- Can be used on or off policy
  - When π_{target} is greedy and $\pi_{\text{behaviour}}$ is exploratory, it is the same as Q-Learning (as an exercise, confirm this with Equation 6.4)
- Eliminates the **variance** added by randomly selecting $A_{t+1} \sim \pi$
- Can set $\alpha = 1$ in deterministic environments as randomness is only from the policy which is removed through taking the mean
- More computational complexity, but likely to outperform Q-Learning and SARSA

This algorithm subsumes Q-Learning and improves upon SARSA by removing the variance associated with the random selection of the next action. Because of this, it is likely to perform better and on a wider range of α while providing the same or better convergence guarantees than the other two.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_{a \in \mathcal{A}} \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{6.4}$$

6.5 Maximization Bias

Using the maximum over estimates as the estimate of the maximum (as done in Q-learning and Sarsa) can cause significant *positive* bias known as maximization bias. The problem is that we are using the same samples to determine the max and compute the value. Instead, in **Double Q-Learning** we store two estimates, $Q_1(s, a)$ and $Q_2(s, a)$ and use the maximum over the values of one to update the values of the other:

$$Q_x(S, A) \leftarrow Q_x(S, A) + \alpha \left[R + \gamma Q_y(S, \text{argmax}_a Q_x(S, a)) - Q_x(S, A) \right] \tag{6.5}$$

Where we randomly swap between $(x, y) = (1, 2)$ and $(2, 1)$. We can do the same thing to make double SARSA or double Expected-SARSA.

Chapter 7

N-step Bootstrapping

We can view Monte Carlo methods (chapter 5) and One-step TD (chapter 6) as being on a spectrum. MC is sort of like TD done over the whole trajectory of an episode. Since the number of steps that MC uses, it can be considered '∞-step TD'.

7.1 Prediction: N-Step TD

As an intermediate between One-step TD and ∞-step TD (MC), in N-step TD we use the first N observed rewards plus the *estimated* value of the state N steps later. To do this, it is useful to define the N-step return:

$$G_{t:t+n} = \gamma^n V_{t+n-1}(S_{t+n}) + \sum_{k=0}^{n-1} \gamma^k R_{t+1+k} \tag{7.1}$$

Note that V_t is the estimate of v_π at time t. You can think of the V_{t+n-1} term as 'accounting' for the missing observed rewards $R_{t+n+1}...R_T$ for an episode that ends at time T. Using this, we can write down the n-step TD update:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)] \quad 0 \leq t < T \tag{7.2}$$

N-step TD can be proved to converge for all N. As a reminder, this is just about estimating v_π since we are considering the prediction problem. We must now look at how to perform control.

7.2 Control: N-Step SARSA

This is a generalization of SARSA (Equation 6.2 is 'SARSA(0)'). As expected, it is **on-policy**. We use a modified definition of N-step return to use action values instead, and then use a policy that is ϵ-greedy with respect to Q:

$$G_{t:t+n} \equiv \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) + \sum_{k=0}^{n-1} \gamma^k R_{t+1+k} \tag{7.3}$$

Therefore our update is

$$Q_{t+n}(S_t, A_t) \equiv Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{7.4}$$

We can also do N-step Expected SARSA by redefining $G_{t:t+n}$:

$$G_{t:t+n} \equiv \gamma^n \left(\sum_a \pi(a|S_{t+n}) Q_{t+n-1}(S_{t+n}, a) \right) + \sum_{k=0}^{n-1} \gamma^k R_{t+1+k} \tag{7.5}$$

7.3 Control: Off-Policy

We simply use the important sampling ratio from Equation 5.4:

$$Q_{t+n}(S_t, A_t) \equiv Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{7.6}$$

We can then use the policy behaviour b for K episodes and train the value function for the greedy policy π.

7.4 Eligibility Traces

Eligibility traces are a clever way to unify TD and MC in a computationally efficient manner. They solve the credit assignment problem by keeping a memory of visited states whose *eligibility* or share in credit decrements with every step they are not revisited.

7.4.1 Credit-Assignment Problem

When we reach some state and receive a nice reward, how do we determine which pairs (S_i, A_i) are most responsible for getting us here? This is the **credit assignment** problem. Monte Carlo gives **equal credit** to all the pairs that got us here. N-step TD only immediately gives credit to the pair that occurred N steps before a reward (e.g. TD(0) only credits the (s,a) that happened right before the reward).

7.4.2 Lambda-Return and Lambda-TD

What if instead of selecting a fixed N for N-step algorithms we took some sort of weighted average of them? Turns out this is good and **guaranteed** to converge so long as the weights are positive and sum to 1. TD(λ) uses weights $(1-\lambda)\lambda^{n-1}$, $\lambda \in [0, 1)$ to weight each n-step return; that is, its target is the λ-return

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{7.7}$$

7.4.3 Eligibility Trace Vector

In the lectures, we use the notation that we have an array $\epsilon[s]$ that stores the 'credit' for each state. In every iteration, we set $\epsilon[s] = 1$ for the current state s, and set $\epsilon[s'] = \lambda \epsilon[s']$ for every other state s'. Then, the reward to apportion to each state in ϵ is $R_t \cdot \epsilon$. When an element in the vector goes below a threshold, we delete it from the vector.

Chapter 8

Value Function Approximation

Thus far we have assumed we can represent the values of Q_π or V_π for all states $s \in \mathcal{S}$. This is not feasible for enormous spaces or high-dimensional ones. Instead we use the approximations

$$\hat{v}_\pi(s;\vec{w}) \approx v_\pi(s) \tag{8.1}$$
$$\hat{q}_\pi(s,a;\vec{w}) \approx q_\pi(s,a) \tag{8.2}$$

Technically $\vec{w}$ doesnt really have to be a vector - it could be any information that affects the value of $\hat{v}$, but a vector is usually used since it allows us to apply calculus to minimize the error in this estimation.

The goal is to *learn* the weight vector that minimizes some error between $\hat{v}_\pi(s;\vec{w})$ and $v_\pi(s)$ (the latter is the *true* state value). In general, the approximated function may be any function of the weights however there are a few common cases that work well. In particular, we normally use **differentiable** functions since a typical assumption is that the value of *nearby* states is related to the value of the current state. Thus when we modify a weight, it affects a *range* of states in a smooth way.

We can conceive of all the algorithms hitherto considered as producing *training examples* $s \to u$. For example, in first-visit MC we are training on $s \to G_t$ - in the tabular case we would directly move the value of the state towards G_t, but in VFA we smoothly adjust the weight to reduce the error on this example.

8.1 Linear VFA

We assume that the state is some vector of **features** $\vec{s}$. For example in 3-D space the features could be the x, y, and z coordinates. $\vec{w}$ is a vector of parameters for the function. In **linear value-function approximation** we use the general form given by Equation 8.3.

$$\hat{v}_\pi(\vec{s};\vec{w}) = \vec{w}^T\vec{s} = \sum_{i=1}^{n} w_i s_i \tag{8.3}$$

8.2 Prediction

8.2.1 Objective Function

The implicit goal of tabular methods was to eventually converge - reduce the error $max|V_\pi(s) - v_\pi(s)|$ to 0. Since we by definition have fewer weights than states (otherwise there would be no point in approximating), we **cannot** generally expect there to be a way to reduce this error to 0. Instead we define a new **prediction objective** to minimize:

$$\mathcal{J}(\vec{w}) = \mathbb{E}_\pi \left[\left(v_\pi(s) - \hat{v}(s;\vec{w}) \right)^2 \right] \tag{8.4}$$

$$= \sum_s \mu(s) \left[\left(v_\pi(s) - \hat{v}(s;\vec{w}) \right)^2 \right] \tag{8.5}$$

Note that Equation 8.5 includes a term $\mu(s)$ which is simply the density at which the state s is visited if we follow $\pi(a|s)$. It is sort of like an 'importance' measure of the value of that given state (states we visit often should have lower error).

The textbook (equation 9.1) uses the form of Equation 8.5 without reference to Equation 8.4, as it assumes $\mu(s)$ can be any measure of how important state values are.

There is no reason to believe that this is the best objective function. This goal just seems like a good one - the real goal is to find a better policy. Regardless, it is the implicit objective of all methods considered from here.

For non-linear $\hat{v}$, the best we can usually hope for is to converge to a **local minima** of $\mathcal{J}(\vec{w})$, though ideally it would be global.

8.2.2 Optimization

Now the question is how to actually reach a local minima? Obviously this will involve some differentiation - but what datapoints should we use and when, and what should the step size be?

Stochastic Gradient Descent (SGD)

Recall that the gradient of a function $\nabla F(\vec{x})$ is a vector $\langle \frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, ..., \frac{\partial F}{\partial x_n} \rangle$. Like the ordinary derivative, this vector points in the direction that would locally **increase** the value of $\vec{x}$, so to minimize F we should step in the negative gradient direction.

Equation 8.7 shows us the update to use. Note that the $\frac{1}{2}$ factor could have just been absorbed into α, it's just there so the result has a nice form.

$$\vec{w} \leftarrow \vec{w} - \frac{1}{2}\alpha \nabla_{\vec{w}} \left[v_\pi - \hat{v}_\pi \right]^2 \tag{8.6}$$

$$= \vec{w} - \alpha \left[v_\pi - \hat{v}_\pi \right] \nabla_{\vec{w}} \hat{v}_\pi \tag{8.7}$$

What makes it stochastic? This update is performed on a single $(s, v_\pi(s))$ pair. Note that we can replace v_π with some unbiased estimate of it, called U_t - with first-visit Monte Carlo this would be $U_t = G_t$.

Performing the derivative of Equation 8.7 in the linear case gives us a simple form because we simply need to multiply the error term by the example's state vector:

$$\nabla_{\vec{w}} \hat{v}_\pi(\vec{w}) = \nabla_{\vec{w}} \vec{w}^T \vec{s} = \vec{s}\, \nabla_{\vec{w}} \vec{w}^T = \vec{s} \tag{8.8}$$

8.2.3 Semi-Gradient Methods

Bootstrapping targets are not unbiased estimators. We cannot achieve convergence guarantees using DP or TD(λ) bootstrapping targets since their values depend on the current weight vector - making them biased. Using these estimators in Equation 8.7 only factors in how the weight affects the error but not the value estimate - thus if we do it anyway it is called a **semi-gradient** method.

8.3 VFA Coding ('Shallow' Learning)

The feature vector $\vec{s}$ (also written $\vec{x}(S_t)$ in the textbook, as a way to decouple it from the state) is actually something we choose prior to letting the agent learn. It is a useful way for us to inject prior knowledge about a given problem, especially in the case of linear VFA. In linear VFA, we have no way to represent *interactions* between state features because there are no 'cross-terms' in Equation 8.3. Instead, we *choose* the features to represent any state interactions we think are useful for the problem.

An example is the pole-balancing problem. Here, the angular velocity at which the pole is moving is good *depending* on the angle. If the angle is low, a high velocity is fine (it is self-correcting). If the angle is high, a high velocity is bad since it is likely falling. So we might choose a feature $s_i = \omega \cdot \theta$ to represent the product of these two whose weight can then be adjusted.

8.3.1 Coarse Coding

One way to select features in a continuous space (e.g. 2D or 3D) is to have n possibly overlapping circles. When the *state* (which is technically the exact location of the point in space) is in a circle c_i, then $s_i = 1$ otherwise $s_i = 0$. Features like this are called **coarse coding**, though they may be as fine-grained as you like.

Remember, the goal is to make the number of features much less than $|\mathcal{S}|$. We also want to inject some prior information through the selection of features. Coarse coding provides both of these. Making the circles bigger essentially tells the agent to **generalize** over a wider region, and we can even choose irregular circles if we know a particular spot needs finer-grained discrimination

8.3.2 Tile Coding

Tile coding is a type of coarse coding, where instead of circles we use n layers of grids each with a slight offset relative to the others. On a digital computer, it is much more computationally efficient to find the singular active region in each grid than to compute the value of c_i in circular coding.

8.3.3 Hand-Coded Tilings

In general, we can choose whatever shapes we like to make the agent generalize in particular directions (or give the agent the *option* to generalize in a given direction) or couple regions together. This should be done in some sensible way based on the problem at hand.

8.4 Neural Networks

This section is sort of optional so long as you get the basic idea of neural networks. From the perspective of this course, they are simply a biologically-inspired nonlinear value-function approximator.

8.4.1 Neuron

The basic unit is the neuron, which takes in features $\{x_1 ... x_d\}$, computes $net = \sum w_i x_i$ and outputs $\sigma(net)$ where σ is the **activation function**. Usually σ is the sigmoid function given by Equation 8.9 (with the nice property given by Equation 8.10)

$$\sigma(x) = \frac{1}{1 - e^{-x}} \tag{8.9}$$

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x)) \tag{8.10}$$

8.4.2 Net Activation

If we compose layers of neurons where layer j has weights w_{ij} we can decide what values of a layer get propagated to which neurons in the next and we thus have a neural network.

8.4.3 Universal Approximation Theorem

Don't need to know the details, but essentially guarantees that a fully-connected neural network can approximate any function (under some conditions on the size).

8.4.4 Gradient Descent

Given an input-output example, we can use an optimizer to tell us how to update the weights. The updates can be batched for efficiency. Note that Stochastic Gradient-Descent isn't the only option - Adam and LBFGS are some other names.

8.4.5 Problems

- Overfitting
- Inefficiency with time-series or image data
- Overfitting
- Overfitting
- Good at interpolation but not extrapolation. A few techniques can address this
  - Separate data into $\sim 70\%$ training data, 10% validation to see how well it is doing, and use the remaining data to test at the end
  - Dropout: randomly drop edges in the network. This creates some sets of nodes that 'specialize' and allows information to be spread. Usually done by specifying a probability p that any given edge will be dropped
- Catastrophic Forgetting: if we suddenly start sampling from a different distribution for example data (e.g. nonstationary problem), we will forget how to solve the original problem. How can we allow for learning new distributions? Segmented networks might be an option

Regularization methods aim to decrease the generalization error while not affecting the training error. That is, they punish overfitting. Dropout is one example of this, some other methods include dataset augmentation, and L1/L2 norm regularization.

8.4.6 Architecture

There are quite a few different approaches to designing the connectivity of networks, which usually depends on the *type* of data we are looking at (images? time-series? spatially-structured data?). LSTM (Long Short-Term Memory) uses time as a variable to identify patterns over and is thus used in ordered data (time-series or language). CNN (Convolutional Neural Networks) perform a convolution which mixes local spatial data together, which is good for images or other spatially structured data.

Chapter 9

Deep Reinforcement Learning

9.1 Overview

As discussed in section 8.4, we can use neural networks as value-function approximators. In general, NNs minimize some **loss function**, which in our case is the difference $Target - Estimate$.

9.2 Deep Q-Network (DQN)

Recall the Q-Learning of Equation 6.3. It uses the experienced reward and the *maximum* of the reward estimates of adjacent states to update the current (S, A) pair. Deep Q-Network uses a neural network to approximate Q. For Atari, the network is two convolutions followed by two dense layers the last of which outputs a one-hot encoding of the direction to move the joystick.

Recall also in subsection 8.2.3 that bootstrapping targets cannot be used for gradient optimization. DQN addresses this issue in two ways: replay buffers and minibatching

9.2.1 Replay Buffer

DQN accumulates a buffer D containing (S_t, A_t, R_t, S_{t+1}) tuples. Note that we don't need A_{t+1} because Q-Learning is off-policy and uses the update of Equation 6.3. The size of this buffer is a **hyper-parameter**. At a time t this buffer is denoted $D_t = \{e_1, ...e_t\}$ where $e_k = (S_k, A_k, R_k, S_{k+1})$.

This buffer will be randomly sampled to perform the update Equation 8.7. Note that $\nabla_{\vec{w}} \hat{v}_\pi$ has a known form, and in practice this uses something like `autograd` to handle arbitrary architectures.

Why does this help? One problem with using bootstrapping targets in gradient descent is that successive samples are *biased*. This is because the update target depends on the policy which is derived indirectly through the current estimate $Q(s, a; \vec{w})$ and the update of Equation 8.7 doesn't factor this in. By taking *uncorrelated* samples from D we mitigate this bias.

9.2.2 Batching or 'Mini-Batching'

Instead of directly performing Equation 8.7 on each timestep, we maintain two networks. One has stable parameters $\vec{w}^-$ while the other is updated every step using that equation. Every C steps, we set $\vec{w}^- = \vec{w}$. This prevents us from having a 'moving target' which increases the stability of this semi-gradient method. The number of steps C at which this occurs is also a hyperparameter.

9.2.3 Double DQN

We still have the issue of maximization bias for which section 6.5 was developed. If we use DQN to approximate both Q_1 and Q_2 we get double DQN which has been shown to be effective in practice.

9.2.4 Objective

The exact objective (or **loss**) function to minimize to use is given by Equation 9.1 (note the connection to Equation 8.4):

$$\mathcal{J}(\vec{w}) = \mathbb{E}_{(S,A,R,S')\sim U(D)}\left[\left(R + \gamma \max_{a'} Q(S', a'; \vec{w}^-) - Q(S, a; \vec{w})\right)^2\right] \tag{9.1}$$

Note that $x \sim U(D)$ means sampling from a discrete uniform distribution over the set D.

9.2.5 Prioritized Experience-Replay

Instead of sampling from D uniformly, why not prioritize samples that have a high error? We can use the absolute value of the inner term of Equation 9.1 to do this. This value is also called the **advantage** function:

$$\mathcal{A}(a, s) = |r + \gamma \max_{a'} Q(s', a'; \vec{w}^-) - Q(s, a; \vec{w})| \tag{9.2}$$

9.3 Policy Gradient Methods

So far we have only looked at methods that compute or estimate action or state-action values based on some policy. For example, in on-policy control (section 6.2) the policy was ϵ-soft with respect to the value estimate. In deep RL we use gradients (or semi-gradients) to minimize some loss function that is proportional to the error between the estimate and true state value. We then feed the (parametrized) value function to one of our control algorithms (SARSA, Q-Learning, etc.) which determines how actions are selected and policies changed.

What if instead of adjusting $\vec{w}$ to minimize this error and then feeding it to a control algorithm, our control algorithm directly manipulated the parameters of the *policy*? If that is unclear, just recall that the policy is simply some probability distribution which is just a function and can thus be approximated like any other. This gives rise to **policy gradient methods**.

As a matter of notation, we use $\vec{w}$ to denote value function parameters while θ is a vector representing policy parameters. The parameterized policy function is now written $\pi(a|s, \theta)$.

Instead of minimizing a loss function as in chapter 8, we maximize a performance measure (or 'objective function') $J(\theta)$ and perform gradient *ascent* using Equation 9.3. Note the connection to Equation 8.7.

$$\theta \leftarrow \theta + \alpha \widehat{\nabla J(\theta)} \tag{9.3}$$

$\widehat{x}$ means some unbiased estimate of x (the exact value is also an unbiased estimate). Some methods approximate the policy and the value function. These are the **actor-critic** methods of section 9.4.

9.3.1 Policy Approximation

Learning the policy has a few Advantages

1. The policy may converge to an optimal deterministic one. This cannot be said of e.g. SARSA (section 6.2)
2. The policy may converge to an optimal stochastic one. This cannot be said of any of the other methods discussed so far (they always get the value function for a deterministic one)
3. Policies may be simpler to approximate than value functions
4. Can learn continuous actions
5. Can inject prior information about the problem by selecting a parameterization

A few good options for parametrizing a policy include soft-max'ed preferences (Equation 9.4 - note the connection to Equation 2.7) for discrete actions and normal distributions (Equation 9.5) for continuous ones.

$$\pi(a|s,\theta) = \frac{e^{h(s,a;\theta)}}{\sum_b e^{h(s,b;\theta)}} \tag{9.4}$$

$$\pi(a|s,\theta) \sim \mathcal{N}(a|\mu(s;\theta),\sigma(s;\theta)) \tag{9.5}$$

9.3.2 Policy Gradient Theorem

In Equation 9.3 we need to know $\widehat{\nabla J(\theta)}$. Note that $J(\theta) = v_{\pi(a|s;\theta)}(s_0)$ for the starting state s_0 (for simplicity, assume that there is some fixed starting state that all episodes begin with, e.g. in mazeworld we always start at some coordinate for the task).

The problem is if we try to use Equation 3.6 to find the derivative we will get stuck because we do not know the dynamics function. The policy gradient theorem gives us a way to **approximate** $\widehat{\nabla J(\theta)}$ in this case without needing to know the dynamics function. The theorem is given by Equation 9.6.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\theta) \tag{9.6}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla_\theta \pi(a|S_t,\theta) \right] \tag{9.7}$$

The proportionality constant does not matter as it can be absorbed into the α of Equation 9.3. $\mu(s)$ is the state distribution under π as described in subsection 8.2.1

9.3.3 REINFORCE Algorithm

REINFORCE is a Monte-Carlo approach to performing policy gradient ascent. It uses the form of the policy gradient theorem given by Equation 9.7. Since this is an expectation, we can get an unbiased estimate of it by sampling the inner value and taking the average. This would involve trying all actions in each state (which would be the **all-actions** method) - REINFORCE omits this by only using A_t. Details of the derivation are in the textbook (section 13.3), but the result is Equation 9.8.

$$\nabla J(\theta) \propto \mathbb{E}_\pi \left[G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] \tag{9.8}$$

So given π_θ, and hopefully find a nice form for $\nabla_\theta \pi_\theta$, the process is as follows. We perform an episode according to π_θ and compute G_t by averaging (with or without discounting) the received returns starting at t for each $t \in [0,T)$. Then, we compute the value $\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$ for each $t \in [0,T)$. We thus have T samples of the thing inside the expectation of Equation 9.8 which we can use to update θ using Equation 9.3.

Since REINFORCE is a Monte-Carlo method, it may have high variance (unbiased does not mean zero variance!). It also may converge slowly. This can be improved using **baselines**, discussed in the next section.

Baselines

As long as we subtract some function that only depends on the **state**, we add no bias to the estimate and can improve the variance and thus get faster convergence. A good choice for this is some approximation of the state value function. Given such a baselines $b(S_t)$ we use the update of Equation 9.9. Note that since $b(S_t)$ could be 0 this is the most general form of REINFORCE.

$$\nabla J(\theta) \propto \mathbb{E}_\pi \left[(G_t - b(S_t)) \nabla_\theta \ln(\pi(A_t|S_t, \theta)) \right] \tag{9.9}$$

Note: $\nabla \ln(x) = \frac{\nabla(x)}{x}$ is just a trick so we can work with log-probabilities that have better numerical properties. If we use the parameterized state value function Equation 8.1 we can also learn *its* value by using its gradient every step using $v_\pi \approx G_t$ in Equation 8.7 (δ is the TD error):

$$\delta \leftarrow G_t - \hat{v}_\pi(S_t; \vec{w}_t) \tag{9.10}$$
$$\vec{w}_{t+1} \leftarrow \vec{w}_t + \alpha_w \delta \nabla_{\vec{w}} \hat{v}_\pi(S_t; \vec{w}_t) \tag{9.11}$$
$$\theta_{t+1} \leftarrow \theta_t + \alpha_\theta \delta \nabla_\theta \ln(\pi(A_t|S_t, \theta)) \tag{9.12}$$

Parameter Update

To get the full update we combine Equation 9.9 with Equation 9.3:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \left(G_t - b(S_t) \right) \nabla_\theta \ln(\pi(A_t|S_t, \theta)) \tag{9.13}$$

9.4 Actor-Critic Methods

Instead of simply using a learned state-value as a baseline (as in REINFORCE with baselines - section 9.3.3), we can use it to point the policy parameter update of Equation 9.3 in a direction that will tend to increase the value of S_{t+1}. We can also think of this as using the estimated state value to assess which actions are better to take from a given state.

Notice how in REINFORCE with baselines, Equation 9.12 only uses the estimate of the value of S_t. In actor-critic methods, it will use the estimate of S_{t+1} as well. Simply put, we can redefine the TD error to use the value estimate of the next state:

$$\delta = \delta_t = R_t + \gamma \hat{v}_\pi(S_{t+1}; \vec{w}) - \hat{v}_\pi(S_t; \vec{w}) \tag{9.14}$$

More generally, we can use any function for δ_t that tells us the advantage of a in s. For example, $A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$ tells us how action a compares to the average value of actions in the given state.

9.4.1 A3C and A2C

Asynchronous Advantage Actor-Critic (A3C) was the first one to be developed [1], and later it was found that the asynchronous part wasnt needed which lead to Advantage Actor-Critic (A2C).

Both algorithms use some sort of neural network to parameterize the value function and policy.

A3C

In A3C, we have a bunch of actors running in parallel. Each gets an initial copy of θ and $\vec{w}$ to work with. They perform the usual algorithm (combining Equation 9.14 with Equation 9.11 and Equation 9.12) but instead of actually updating θ and $\vec{w}$, they accumulate the value on the right hand side of the latter equations. Every once in a while the actor sends these values to the root 'controller' which adds them to the global parameters. It then synchronizes its own parameters with the global ones and repeats this process.

[1] Original paper available at https://arxiv.org/pdf/1602.01783.pdf

A2C

Instead of allowing updates to occur asynchronously, the controller just waits for all agents to finish the current iteration, updates the global parameters, then runs the agent again with the new parameters. This 'batched' training works better with hardware accelerators (e.g. GPUs) where IO is the bottleneck - updates can be accumulated and then sent at once to the controller.

9.5 TRPO and PPO

Trust-Region Policy Optimization (TRPO) relies on the idea that taking too large of a step in policy space may result in a bad policy that prevents convergence - instead we take 'small' steps. The question is how to quantify 'small steps' and the answer is through KL divergence. Consider two policies $\pi_\theta(a|s)$ and $\pi_{\theta'}(a|s)$. They are just probability distributions, and KL divergence is a way to measure how different one probability distribution is from another (it is technically not a distance as it is not symmetric). The KL divergence is given by Equation 9.15.

$$\mathrm{D}_{\mathrm{KL}}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{9.15}$$

The objective in TRPO is to find $\text{argmax}_\theta J(\theta)$ subject to the constraint Equation 9.16

$$\mathbb{E}\left[\mathrm{D}_{\mathrm{KL}}(\pi_{\theta_{\text{old}}}||\pi_\theta)\right] \leq \delta \tag{9.16}$$

What is $J(\theta)$ in this case? It is essentially the importance sampling ratio of using parameters θ for the policy instead of θ_{old}:

$$J(\theta) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\theta_{\text{old}}}(s,a)\right] \tag{9.17}$$

9.5.1 PPO

TRPO uses a slightly complex $J(\theta)$ - **Proximal Policy Optimization** (PPO) simplifies this in a way that makes the constraint easier to solve. It does so by clipping the value inside the expectation of Equation 9.17.

9.6 Deterministic Policy Gradients

In this approach, we work with a deterministic policy μ_θ. The idea is that this reduces the dimensionality of the space the gradient is integrated over (no longer state and action, just state) which gives us a computational advantage over stochastic policy gradients. We must work with off-policy methods here to get exploration though.

9.6.1 Deterministic Policy Gradient Theorem

This is a deterministic analog to Equation 9.6:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{\mu_\theta}\left[\nabla_\theta \mu_\theta(s) \left(\nabla_a Q_{\mu_\theta}(s,a)\right)|_{a=\mu_\theta(s)}\right] \tag{9.18}$$

9.6.2 Deterministic Policy Gradient (DPG)

Using Equation 9.18 the update we use, analogous to Equation 9.3 is

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta \mu_\theta(s) \left(\nabla_a Q_{\mu_\theta}(s,a;\vec{w})\right)|_{a=\mu_\theta(s)} \tag{9.19}$$

Note that since the gradient is taken with respect to the actions, this only works for continuous action spaces. This is fine because it is large action spaces for which this method is useful in the first place.

Deep Deterministic Policy Gradient (DDPG)

In DDPG we add some noise to the stochastic behaviour policy and use neural networks for estimation.

Soft Actor-Critic (SAC)

In this method, we include the policy entropy in $J(\theta)$ which helps with convergence and exploration. The main idea is to parameterize π with respect to Q using a softmax distribution.

1. Update Q using a gradient step
2. Take a gradient step in θ' space to minimize Equation 9.20
3. Repeat until convergence

$$\mathrm{D_{KL}}\left(\pi_{\theta'}||\frac{1}{Z}e^{Q_{\pi_\theta}(s,a)}\right) \tag{9.20}$$

Where Z is the normalizing constant for softmax.

9.7 Monte Carlo Tree Search (MCTS)

In this method, we maintain a tree of states and their values and update it for a fixed number of steps before taking an action. There are 4 steps that are repeated N times:

1. Selection: starting at the root node R, recursively traverse selecting the child with highest value at each step until a leaf L is reached
2. Expansion: given a (non-terminal) leaf, add $|\mathcal{A}|$ children corresponding to the avilable actions at this leaf, and select the first one (M)
3. Simulation: simulate the environment from M until a terminal state is reached, which gives us a value (e.g. 1 for a win state or 0 otherwise)
4. Backpropagation: update the values of the nodes M through R

9.7.1 Alpha-Beta Search

At each step during simulation, alternatingly choose the best action for yourself and the best action your opponent can take (i.e. the worst action for you). This must be based on some heuristic function - the accuracy is not that important, just a way of hinting which states may be better for a given agent.

This method is difficult for games like Go where no good heuristic can be determined.

9.7.2 Game Tree

We can use something like UCB (section 2.5) or a value neural network to store probabilities of actions by training on human or other AI players. We can use this to guide the growth of the 'game tree' and backpropagate the right values.