Rishabh Indoria
21F3001823

Reinforcement Learning

March 26, 2025

1 Introduction to Reinforcement Learning

- A trial and error learning paradigm, rewards and punishments.
- Learn about a system through interaction.
- The idea is inspired by behavioral psychology.
- We use RL when the system has complex dynamics, we have a complex workspace
- Stochastic sensing and actuation is required
- Human-in-the-loop learning
- Customization/Personalization, Recommendation systems
- RL goes beyond human knowledge, it can learn through self play.
- RL can be used to improve heuristics.
- Book to be used is "Reinforcement Learning: An Introduction", 2^{nd} edition, this can be found here.

2 Fundamental Concepts in RL

2.1 Games and Learning

- Consider the game of Tic-tac-toe
- We can set up a supervised learning paradigm based on current position and the correct move.
- Then we can use any model to train and predict the move.
- In RL we learn from evaluation, so if the model wins then it gets 1 point, if it loses then it gets -1 point and if it draws then it gets 0 points.
- RL will learn from repeated play.
- **MENACE**: Matchbox based learning for tic-tac-toe.
- Each matchbox has a position drawn on it, and inside it are different colored beads which correspond to different moves the computer/matchbox can make.
- Let's say we play a game, then we would have chosen some number of matchboxes and picked some beads.
- If matchbox wins then add two of the beads back in the matchbox, and if it looses then we throw away the beads.
- By doing this, we are essentially increasing the probability of winning for the matchbox.
- We had to wait till the end of the game to update, but we don't need to wait till the end, we can check intermediate states based on some rules.
- We can also remember previous plays and say that a certain position has a higher or lower chance of winning. Now, for every move, we would want the chance of winning to go up.
- **Temporal Difference Learning**: Look at successive states and change action probabilities based on chance of winning.
- TD is a simple rule to explain complex behaviors.
- **Intuition**: Prediction of outcome at time $t+1$ is better than the prediction at time t. Hence, use the later prediction to adjust the earlier prediction.

2.2 Bandit Problems

- At the core of RL is exploration vs exploitation.
- **Immediate Reinforcement**: The payoff accrues immediately after an action is chosen.
- One key question they try to solve is the dilemma between exploration vs exploitation.
- **Bandit problems** encapsulate 'Explore vs Exploit'.
- We are trying to find profitable actions.
- We exploit to act according to the best observations already made.
- Always exploring might not be optimal, always exploiting might not be optimal either.
- **Multi-arm Bandits**: n-arm bandit problem is to learn to preferentially select a particular action(arm) from a set of actions $(1, 2, 3, ..., n)$.
- Each selection results in Rewards derived from the respective probability distribution.
- Arm i has a reward distribution with mean μ_i and $\mu^* = \max\{\mu_i\}$.
- We are trying to identify the arm with mean μ^*.
- Objective is to identify the correct arm eventually.
- We first look at traditional approaches
- Let $r_{i,k}$ be the reward sample acquired when i^{th} arm is selected for the k^{th} time.
- $Q(a_i)$ is the average or expected reward that we get by choosing a_i.
- Define

$$Q(a_i) = \frac{\sum_k r_{i,k}}{\sum_{\{k:r_{i,k}\}} 1} \quad Q(a^*) = \max\{Q(a_i)\}$$
$$Q_{k+1}(a_i) = Q_k(a_i) + \alpha(r_{i,k} - Q_k(a_i))$$

- Setting $\alpha = \frac{1}{k_i+1}$ yields the average.
- Setting α to a constant makes the expression a weighted average.
- **Epsilon Greedy**: Select arm $a^* = \arg\max_i\{Q_k(a_i)\}$ with probability $1 - \epsilon$ and select any arbitrary arm with probability ϵ. Usually ϵ is very small.
- This is an exploration method, works fairly well in real world scenarios.
- **Softmax**: Select arms with probability proportional to the current value estimates

$$\pi_k(a_i) = \frac{\exp\left(Q_k(a_i)/\tau\right)}{\sum_j \exp\left(Q_k(a_j)/\tau\right)}$$

- τ is called temperature, low temperature corresponds to greedy or exploitation and high temperature corresponds to uniform or exploration.

2.3 Regret and PAC Frameworks

- So far, we maximized the total rewards obtained
- But we can also try to minimize **regret**(loss) while learning.
- We can also use **Probably Approximately Correct** frameworks(PAC)
  1. Identification of an ϵ-optimal arm with probability $1 - \delta$
  2. ϵ-optimal: Mean of the selected arm satisfies

$$\mu > \mu^* - \epsilon$$

  3. Minimize sample complexity: Order of samples required for such an arm identification.

- Practical algorithms tend to go the regret route rather than PAC optimality.
- **Median Elimination** achieves PAC optimality, it is a round based algorithm.
- Due to this algorithm, more and more algorithms are developed based on round based method.
- **Upper Confidence Bounds** achieves regret optimality, idea is to reduce the constants hidden in big O notation.
- The arm with the best estimate r^* so far serves as a benchmark, and other arms are played only if the upper bound of a suitable confidence interval is at least r^*.
- **Simple Approach**: Be greedy with respect to upper confidence bounds.
- Sub-optimal arm j is played fewer than $(8/\Delta_j)$ in n times.
- Consider a deterministic policy $UCB1$
- **Initialization**: Play each arm once.
- **Loop**: Play arm j that maximizes $\bar{x}_j + \sqrt{\frac{2\ln n}{n_j}}$ where $\bar{x}_j$ is the average reward obtained from arm j, n_j is the number of times arm j has been played so far, and n is the overall number of plays done so far.
- Details regarding this algorithm can be found in this paper.
- **Thompson Sampling** also achieves regret optimality, this is a Bayesian approach.

2.4 Contextual Bandits

- Different ads for different users, we can use one bandit for each user.
- However, this is hard to train, needs several rounds of experience with same user.
- Assume that the parameters of the reward distributions themselves are determined by a set of hyperparameters.
  item Typical assumption is a linear parameterization of the expectation.
- Essentially grouping users based on some context, age, gender, last queries, etc.
- Assume that each user is represented by a set of features, can be joint features of user and arm.
- The "static" used for choosing arms is now dependent on these features.
- Could correspond to the presence or absence of different signals.
- We could also have features for arms, in case there are way too many arms.
- **LinUCB** is one of the more popular contextual bandit algorithms, whose paper can be found here.
- Idea is that **Predicted expected reward** is assumed to be a linear function of the features.
  1. Use ridge regression to fit parameters
  2. Can derive an upper confidence bounds for the regression fit
  3. Use UCB like action selection
  4. Gives better performance with lesser "training" data
- Bandits have only actions
- Contextual bandits have actions, context, but no sequence.
- A full RL problem will have all three.
- I have implemented multiple methods for the multi armed bandit problem, which can be found at `https://colab.research.google.com/drive/15vgC170tQthc2DVUM3ZN_-jqJiyclGHl?usp=sharing`

2.5 Full RL Problem

- Action at a Temporal Distance
- Learning an appropriate action at state 1 is dependent on the actions at future states.
- There is **no immediate** feedback.
- We have an agent and environment.
- Learn from close interaction with a stochastic environment having noisy delayed scalar evaluation with a goal to maximize a measure of long term performance.

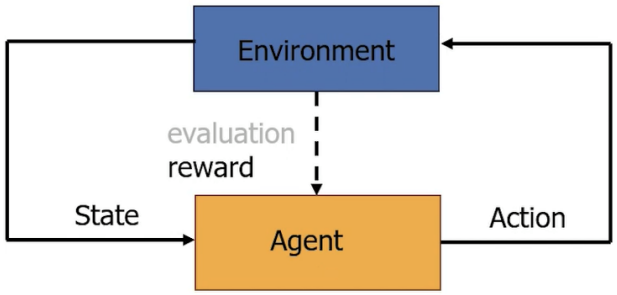
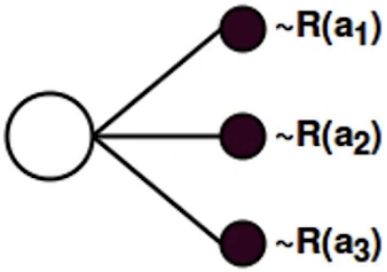
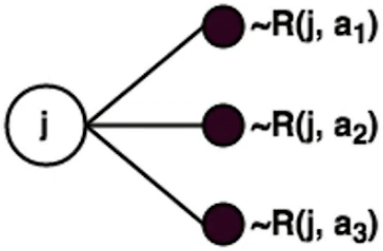


Figure 1: Simplified RL framework

- Bandit problem can be viewed as having a single state, and multiple actions that lead to certain rewards.



(a) Bandit Problem



(b) Contextual Bandit Problem

- **Idea**: Use prediction as **surrogate** feedback. This allows us to get immediate reward.

3 Markov Decision Processes

3.1 Introduction to MDP

- Putting together the complete RL problem, we define the **Agent-Environment Interface**.

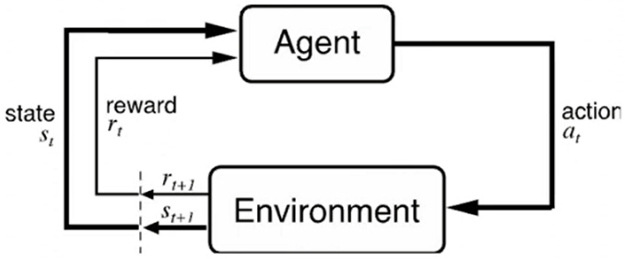


Figure 3: The Agent-Environment Interface

- Agent and Environment interact at discrete time steps: $t = 0, 1, 2, ...$
- Agent observes state at step t: $s_t \in S$

- produces action at step t: $a_t \in A(s_t)$
- gets resulting reward: $r_{t+1} \in \mathcal{R}$
- and resulting next state: S_{t+1}

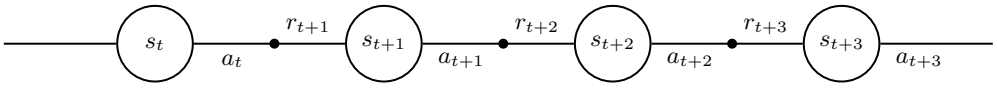


Figure 4: Rolled out Interface

- "the state" at step t, means whatever information is available to the agent at step t about its environment.
- The state can include immediate "sensations", highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations to retain all "essential" information, i.e., it should have the **Markov Property**

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ...\} = Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t\} \ \forall s', r$$

- **Markov Decision Processes**, M, is the tuple: $M = \langle S, A, p, r \rangle$
  1. S: set of states
  2. A: set of actions
  3. p: $S \times A \times S \to [0, 1]$, probability of transition
  4. r: $S \times A \times S \to \mathbb{R}$, expected reward
- Goal of the agent is to learn an **optimal** policy $\pi : S \times A \to [0, 1]$, can be deterministic.
- We want to learn a policy such that expected reward is maximized.
- The policy that achieves maximum total expected reward is the **optimal** policy.
- **States**: Enough information to take decisions, Raw inputs, are often not sufficient.
- **Actions**: The control variables, can be discrete or continuous.
- **Rewards**: Defines the goal of the problem.

3.2 Returns and Value Functions

- **Policy** at step t, π_t: A mapping from states to action probabilities,
  $\pi_t(s, a)$ = probability that $a_t = a$ when $s_t = s$
- Reinforcement learning methods specify how the agent changes its policy as a result of experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.
- Suppose the sequence of rewards after step t is: $r_{t+1}, r_{t+2}, r_{t+3}, ...$
- We want to maximize the **return**, G_t, for each step t.
- **Episodic tasks**: Interaction breaks naturally into episodes, have some end point, e.g., plays of a game, trips through a maze.

  $$G_t = r_{t+1} + r_{t+2} + ... + r_T$$

  where T is a final time step at which a **terminal state** is reached, ending an episode.
- **Continuous tasks**: Interaction does not have natural episodes. In this case we use **Discounted return**

  $$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

  where $\gamma, 0 \leq \gamma \leq 1$, is the **discount rate**. shortsighted $0 \leftarrow \gamma \to 1$ farsighted.
- In general, we want to maximize the **expected return**, $E\{G_t\}$, for each step t.

- If we are sure that states will end at some point then take $\gamma = 1$ else γ is strictly less than 1.
- **Value Functions**: Expected future rewards, starting at a state or state-action pair and following policy π.
- **State-value function for policy π**

$$v_\pi(s) = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] = E_\pi[G_t | S_t = s]$$

- **Action-value function for policy π**

$$q_\pi(s,a) = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] = E_\pi[G_t | S_t = s, A_t = a]$$

- Both of these are connected as follows

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) = E_\pi[q_\pi(s,a)]$$

- We can use either but q_π ends up being better than v_π in a lot of the problems.

4 Stochastic Dynamic Programming

4.1 Bellman Equations

- **Bellman Equation for a Policy π**

$$\begin{aligned} v_\pi(s) =& E_\pi[G_t | S_t = s] \\ =& E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ =& \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)[r + \gamma E_\pi[G_{t+1} | S_{t+1} = s']] \\ =& \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a)[r + \gamma v_\pi(s')], \text{ for all } s \in \mathcal{S} \end{aligned}$$

- From s we can take a number of actions based on policy π
- For each action, there could be a number of states that it could lead and their corresponding reward.
- We can write $E_\pi[G_{t+1} | S_{t+1} = s'] = E_\pi[G_t | S_t = s'] = v_\pi(s')$ because of Markov property and stationary.
- This is a linear equation in $|S|$ variables.
- A unique solution for this exists.
- **Equiprobable Random Policy**: Same probability for all actions.
- For finite MDPs, policies can be partially ordered.
- $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$
- There is always at least one (possibly many) policies that is better than or equal to all the others. This is an **optimal policy**. We denote them all as π^*
- There is at least one policy that is deterministic.
- Optimal policies share the same optimal state-value function

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S} \\ q_*(s,a) &= \max_\pi q_\pi(s,a) \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \end{aligned}$$

- Try to derive optimal value function, from this derive the optimal policy.

- **Bellman Optimality Equation for v_***: The value of a state under an optimal policy must equal the expected return for the best action from that state.

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi^*}(s,a) \\ &= \max_a E_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a E_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')] \end{aligned}$$

- **Bellman Optimality Equation for q_***

$$\begin{aligned} q_*(s,a) =& E_{\pi *}[G_t | S_t = s, A_t = a] \\ =& E_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ =& E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ =& E[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ =& \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned}$$

- Any policy that is greedy with respect to v_* is an optimal policy.
- Therefore, given v_*, one-step lookahead search produces the long-term optimal actions.
- Given q_*, the agent does not even have to do a one-step lookahead search.

$$\pi_*(s) = \arg\max_{a \in \mathcal{A}(s)} q_*(s,a)$$

4.2 Policy and Value Iteration

- DP is the solution method of choice for MDPs
- Requires complete knowledge of system dynamics(transition matrix and rewards), Computationally expensive, Curse of dimensionality however it is guaranteed to converge.
- **Policy Iteration**: We solve for v_π iteratively using its bellman equation.
- We keep iterating until v_{k+1} and v_k are sufficiently close.

Algorithm 1 Iterative Policy Evaluation Algorithm

Input π, the policy to be evaluated
Algorithm parameter: A small threshold $\theta > 0$ determining accuracy of estimation.
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)[r + \gamma V(s')]$
    ▷ If s' is terminal state then replace $[r + \gamma V(s')]$ with $[r]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$

- One issue or feature of the above algorithm is that once $V(s_1)$ is updated, then all other states will use the updated value rather than the previous value.
- Suppose we have computed v_π for an arbitrary deterministic policy π.
- The value of doing a in state s is

$$\begin{aligned} q_\pi(s,a) =& E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ =& \sum_{s',r} p(s', r | s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

- It is better to switch to action $a \neq \pi(s)$ for state s if and only if $q_\pi(s,a) > v_\pi(s) = q_\pi(s,\pi(s))$
- Doing this for all states to get a new policy π' that is greedy with respect to v_π

$$\begin{aligned}\pi'(s) &= \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]\end{aligned}$$

- This will guarantee $v_{\pi'} \geq v_\pi$
- If we keep doing this then at some point we will find $v_{\pi'} = v_\pi$ for all states

$$v_{\pi'}(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

- But this is the Bellman Optimality Equation
- So, $v_{\pi'} = v_*$ and both π and π' are optimal policies.
- **Policy Iteration**: $\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_*$
  E corresponds to policy evaluation, I corresponds to policy improvement, "greedification".

Algorithm 2 Policy Iteration Algorithm

▷ 1. Initialization
$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

loop
    ▷ 2. Policy Evaluation
    repeat
        $\Delta \leftarrow 0$
        for each $s \in \mathcal{S}$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]$
            ▷ Again if s' is terminal, then replace $[r + \gamma V(s')]$ with $[r]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end for
    until $\Delta < \theta$

    ▷ 3. Policy Improvement
    policy-stable $\leftarrow$ *true*
    for each $s \in \mathcal{S}$ **do**
        old-action $\leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s',r} |p(s',r|s,a)[r + \gamma v_\pi(s')]$
        ▷ Break ties consistently so that we don't oscillate between ties.
        if *old-action* $\neq \pi(s)$ **then**
            policy-stable $\leftarrow false$
        end if
    end for
    if *policy-stable* **then**
        return $V \approx v$, and $\pi \approx \pi_*$
    end if
end loop

- The above algorithms utilize the bellman optimality equation for action-value function.
- We can also use the equation for state-value function.
- state-value function can be updated iteratively without having to calculate policy at each iteration.
- We will calculate deterministic policy only once, at the end of the loop.
- **Value Iteration Algorithm**

Algorithm 3 Value Iteration Algorithm

Algorithm parameters: A small threshold $\theta > 0$ determining accuracy of estimation.
Initialize $V(s)$, for all $s \in \mathcal{S}^+$ arbitrarily except that $V(terminal) = 0$

repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$ such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

- In these algorithms, we have to do the updates over the entire state set.

4.3 Asynchronous Dynamic Programming

- In asynchronous DP, the updates are not done over the entire state set at each iteration.
- Have to ensure that every state is visited sufficiently often for convergence.
- Gives flexibility to choose order of updates.
- Can intertwine real time interaction with the environment and DP updates.
- Can focus updates on parts of state space relevant to agent.
- **Real-Time DP**: Idea is that we get a trajectory by following some policy π_k, then update only those states that were part of the trajectory and then do greedification to get a new policy π_{k+1}.
- Generally we do multiple trajectories and do ϵ-greedy.
- If policy evaluation step is stopped after one update of each step, then we apply greedification to that one step, we get value iteration.
- **Generalized Policy Iteration**: Refers to the idea of letting policy evaluation and policy improvement interact, independent of their granularity.
- Almost all RL methods can be viewed as GPI.
- Policy iteration has evaluation running to completion before improvement begins.
- Value iteration has only one step of evaluation done before improvement step.
- Asynchronous DP has the two interleaved at a finer granularity.

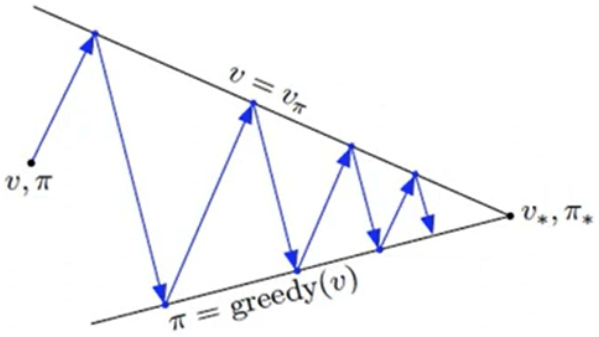


Figure 5: Interleaving

5 Monte Carlo and Temporal Difference Methods

5.1 Monte Carlo Methods

- Learning directly from sample episodes of experience.
- Does not use a known model and is **model-free**.
- Monte Carlo does not use bootstrapping.
- Value functions are calculated as mean of discounted returns(G_t)
- **First-visit MC** method estimates $V(s)$ as the average of the returns following first visits to s.

Algorithm 4 Monte-Carlo Prediction: First Visit MC

```
▷ Initialize
π ← policy to be evaluated
V ← an arbitrary state-value function
Returns(s) ← an empty list, for all s ∈ S
loop
    Generate an episode using π
    for each state s appearing in the episode do
        G ← return following the first occurrence of s
        Append G to Returns(s)
        V(s) ← average(Returns(s))
    end for
end loop
```

- **Every-visit MC** method averages returns following all visits to s.
- **Bootstrapping**: Update using an estimate, DP bootstraps.
- **Sampling**: Update calculates using samples without models, Monte Carlo samples.
- **Temporal Difference Learning** does both bootstrapping and sampling.

5.2 Temporal Difference Learning

- Simple rule to explain complex behaviors.
- **Intuition**: Prediction of outcome at time $t+1$ is better than the prediction at time t. Hence, use the later prediction to adjust the earlier prediction.
- Simplest "TD" Method, TD(0)

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

- $r_{t+1} + \gamma V(s_{t+1})$ is the estimate of $V(s_t)$ at time $t+1$.
- The α term is called the **TD error**.
- TD methods (like MC) do not require a model of the environment, only experience(sampling).
- TD methods can be fully incremental (bootstrapping)
- We can learn **before** knowing the final outcome, less memory and less peak computation.
- We can learn **without** the final outcome, for incomplete sequences.
- TD methods thus combine individual advantages of DP and MC.
- If there were infinite samples, then MC and TD converge to the same result.
- MC converges to the least squares estimate of the return.
- TD converges to certainty equivalence estimate.
- Policy Evaluation: Use $TD(0)$ to evaluate value function.
- Policy Improvement: Make policy greedy w.r.t current value function.
- Note that we estimate **action values** rather than state values in the **absence of a model**.

5.3 On-Policy Learning

- **On-Policy Control**: We sample transitions from the world according to the policy we want to evaluate.
- **SARSA**: After every transition from a non-terminal state s, do

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- SARSA: State - Action - Reward - State - Action
- If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1}) = 0$

Algorithm 5 SARSA Algorithm

```
Initialize Q(s, a) arbitrarily
for each episode do
    Initialize s
    Choose a from s using policy derived from Q
    repeat
        Take action a, observer, s'
        Choose a' from s' using policy derived from Q
        Q(s, a) ← Q(s, a) + α[r + γQ(s', a') − Q(s, a)]
        s ← s'
        a ← a'
    until s is terminal
end for
```

- Convergence is guaranteed as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy.

5.4 Off-Policy Learning

- In off-policy control, we have two policies
  1. **Behavior policy**: used to generate behavior
  2. **Estimation policy**: The policy that is being evaluated and improved
- One-step **Q-learning**

$$\hat{q}(s_t, a_t) \leftarrow \hat{q}(s_t, a_t) + \alpha[rt + 1 + \gamma \max \hat{q}(s_{t+1}, a) - \hat{q}(s_t, a_t)]$$

- Again, if s_{t+1} is terminal, then $Q(s_{t+1}, a) = 0$

Algorithm 6 Q-learning Algorithm

```
Initialize Q(s, a) arbitrarily
for each episode do
    Initialize s
    repeat
        Choose action a from s using policy derived from Q
        Take action a, observe r, s'
        Q(s, a) ← Q(s, a) + α[r + γ max Q(s', a') − Q(s, a)]
        s ← s'
    until s is terminal
end for
```

- Q-learning always updates greedily, a small fraction of updates will be different from SARSA.
- Cliff Walking Example
- In Off-policy learning, we want to evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$
- However we follow behavior policy $\mu(a|s)$, this is done due to a multitude of reasons, π can be deterministic, have less exploration,...
- Assumption of coverage

1. Every action taken under π is also taken, at least occasionally, under μ.
2. $\pi(a|s) > 0$ implies $\mu(a|s) > 0$

- **Importance Sampling**: A general technique for estimating expected values under one distribution given samples from another.

$$\begin{aligned} E_{X\sim P}[f(X)] =& \sum P(X)f(X) \\ =& \sum Q(X)\frac{P(X)}{Q(X)}f(X) \\ =& E_{X\sim Q}[\frac{P(X)}{Q(X)}f(X)] \end{aligned}$$

- **Off-Policy MC with weighted importance sampling**

1. **Importance-sampling ratio**

$$\rho_t^T = \frac{\prod_{k=t}^{T-1}\pi(A_k|S_k)p(S_{k+1}|S_k,A_k)}{\prod_{k=t}^{T-1}\mu(A_k|S_k)p(S_{k+1}|S_k,A_k)} = \prod_{k=t}^{T-1}\frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$$

2. **Weighted average return for** V

$$V(s) = \frac{\sum_{t\in\mathcal{J}(s)}\rho_t^{T(t)}G_t}{\sum_{t\in\mathcal{J}(s)}\rho_t^{T(t)}}$$

6 Advanced Prediction and Control Techniques

6.1 N-Step and TD-Lambda

- We saw $TD(0)$ where the target contains the next step reward.
- Alternatively, we can consider the rewards received in the next n steps

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1}r_{t+n} + \gamma^n V(s_{t+n}) - V(s_t)]$$

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{n-1}r_{t+n} + \gamma^n V_t(s_{t+n})$$

- The extreme would be to consider rewards till the end of the episode(Monte Carlo).
- Downside is that we need to wait n time steps to make an update.
- The variance can also become very high.
- **TD(λ)**: Instead of using n-step backups, we can consider an average of n-step returns.

$$\text{Example: } G_t^{avg} = \frac{1}{2}G_t^{(2)} + \frac{1}{2}G_t^{(4)}$$

- In TD(λ), the average contains all the n-step backups each weighted proportional to λ^{n-1}, where $0 \leq \lambda \leq 1$
- Define λ-return

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$

- Putting $\lambda = 0$ we get our usual 1-step return and putting $\lambda = 1$ we get an iterative version of MC return.
- This is called the **forward view** of TD(λ)
- **Eligibility Traces**: It is a **backward view** of TD(λ)
- These are variables associated with each state denoted by $e_t(s)$.
- They indicate the degree to which each state is eligible for undergoing learning changes.

- On each step

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1, & \text{if } s = s_t \end{cases}$$

for all $s \in S$, where γ is the discount rate.

Algorithm 7 TD(λ) Algorithm

Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$
for each episode **do**
    Initialize s
    repeat
        $a \leftarrow$ action given by π for s
        Take action a, observe reward r, and next state s'
        $\delta \leftarrow r + \gamma V(s') - V(s)$
        $e(s) \leftarrow e(s) + 1$
        for all s **do**
            $V(s) \leftarrow V(s) + a\delta e(s)$
            $e(s) \leftarrow \gamma\lambda e(s)$
        end for
        $s \leftarrow s'$
    until s is terminal
end for

- Implementing this in a non-neural network setting is easy, however when using neural networks this becomes tricky.

6.2 Double-Q Learning

- **Maximization Bias**: Using maximum overestimate as an estimate of the maximum.
- Maximization Bias does not mean that the algorithm will not reach optimal.
- It simply means that at the start, there is a chance to deviate from the optimal action.
- We are essentially using the same samples both to determine the maximizing action and to estimate its value.
- Simple solution is to use different estimates for maximizing the action and estimating its value.
- We learn 2 Q functions, one with first half of samples and second with other half.

Algorithm 8 Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q$

Algorithm parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$

for each episode **do**
    Initialize S
    repeat
        Choose A from S using the policy ϵ-greedy in $(Q_1 + Q_2)$
        Take action A, observe R, S'
        $p \leftarrow$ a random number between $[0, 1]$
        if $p < 0.5$ **then**
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha(R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A))$
        else
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha(R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A))$
        end if
        $S \leftarrow S'$
    until S is terminal
end for

- We can show that both Q_1 and Q_2 will converge to q^*
- This doesn't eliminate maximization bias completely, it reduces it quite a bit.

7 Function Approximations

- Issues with large state/action spaces
  1. Tabular approaches are not memory efficient
  2. Suffers from data sparsity
  3. Continuous state/action spaces
  4. Generalization
- Use a parameterized representation, Value functions, Policies, Models.
- **Value function approximation**: Least squares $Q(s_1, a_t) = f(s_t, a_t; w_T)$

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla_{w_t}[q_*(s_t, a_t) - Q(s_t, a_t)]^2$$

- But we don't know the target.
- To solve this we can use the TD target

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla_{w_t}[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]^2$$

- One issue here is that we are using f or w to generate both target and prediction.
- Gradient Descent and all advancements can be used.
- While computing the gradient of the TD error in Q-learning, we typically ignore the gradient w.r.t the TD target. Hence, it is a **semi gradient** method, i.e., we are computing an approximation of the true gradient.

$$\begin{aligned}
&Q(s_t, a_t) = \phi^T(s_t, a_t) \times w_t \\
&\delta_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \\
&\nabla_{w_t}[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]^2 = -\delta_t\phi(s_t, a_t) \\
&w_{t+1} = w_t + \alpha\delta_t\phi(s_t, a_t)
\end{aligned}$$

- Here ϕ is some representation of the state action pair, it should be a vector of numbers. Essentially a transformation from $\mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$
- Assume the task is to learn a policy on a big grid world.
  Naive method: Divide the grid into smaller 10×10 grid. Abrupt change in Q-value of states that lie on the boundary.
- **Coarse coding**: Avoids abrupt changes in the value of the state. Smoothens the Q-values during transition from one cluster to another. However, there is no uniformity in the number of 'ON' bits used to represent a state.
  Tile Coding: Form of coarse coding but systematic. Number of 'ON' bits $\equiv$ Number of tiles used.
- Linear function approximaters are restrictive. Can only model linear functions. Basis expansion does help to generate non-linear functions in the the original input space.
- Non-linear approximators can model complex functions and are very powerful.
- The features are learnt on the fly and are not hardcoded, as is the case with tile and sparse coding.
- Can generalize to unseen states.
- However, these would require a lot of data and compute.

8 Deep Reinforcement Learning

8.1 Value-Based Methods

- **TD-Gammon**: Learnt completely by self play. New moves not recorded by humans in centuries of play.
- **Deep Q-Learning Network (DQN)**: `https://arxiv.org/abs/1312.5602`, Q function is modeled as a CNN.

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla_{w_t}[r_{t+1} + \gamma \max_a \hat{q}(s_{t+1}, a) - \hat{q}(s_t, a_t)]^2$$

  Divergence is an issue since the current network is used to decide its own target.
- Correlations between samples, Non-stationarity of the targets.
- These issues are addressed by **Replay Memory**, and **Freeze target Network**
- **Replay Memory**: To remove correlations, build dataset from the agent's experience
- **Freeze Target Network**: Sample experiences from dataset; freeze w^- to address non-stationarity.

$$[r_{t+1} + \gamma \max_a \hat{q}(s_{t+1}, a; w^-) - \hat{q}(s_t, a_t; w)]^2$$

- Convolutional layers act as feature extractors.
- These features are then passed through a series of fully connected layers.
- The output layer has $|A|$ number of nodes which are used to calculate the Q-value for each action.
- This network is updated using Huber loss and not regular least squares loss.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2, & \text{for } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

- Recall Double Q-Learning 8, different Q functions for selecting an action and estimating its value.
- DQN can be extended **Double DQN (DDQN)**, `https://arxiv.org/abs/1509.06461`
- We can use the target network to estimate the value while the online network is used for selecting the action.
- Unlike the Q-learning algorithm, where both the networks are updated with equal probability, in Double DQN, only the online network will be updates since the target network is updated towards online network as according to the DQN.

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1,a;\theta_t}), \theta_t^-)$$

- Different versions of a single estimator of Q value is being used. A target model Q' is used for action selection and Q is used for action evaluation.

Algorithm 9 Deep Double Q-learning

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  end for
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value

$$Q^*(s_t, a_t) \approx r_t + \gamma Q_\theta(s_{t+1}, \arg\max_a Q_{\theta'}(s_{t+1}, a))$$

    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters

$$\theta' \leftarrow \tau * \theta + (1 - \tau)\theta'$$

  end for
end for

- Updating θ' can be a direct copy of θ or **Polyak averaging**, used above.
- **Dueling DQN**: Sometimes it is not necessary to estimate the value of each action choice in many states, `https://arxiv.org/pdf/1511.06581.pdf`.
- Value network attends on both the horizon of track and score.
- The attention network concentrates only when there are inbound cars. Only in these states there is a need to estimate the value of either moving left/right.
- Estimate the Q-value using both Advantage $A(s,a)$ and $V(s)$, calculates the benefit of taking an action.
- Explicitly separating two estimators, the dueling architecture can learn which states are valuable, without having to learn the effect of each action for each state.
- Separate aggregation layer is required to combine $V(s)$ and $A(s,a)$. Simple addition wouldn't work, will fall into the issue of identifiability.
- Average advantage is calculated

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha))$$

- This aggregation gives us the Q-function, so the network is essentially a Q-network. Thus, we can directly apply Q-learning techniques such as DDQN and PER with the dueling architecture.
- In SARSA update, a_{t+1} introduces variance which slows convergence.
- **Expected SARSA**: Using the expectation of Q value reduces the variance in the update, we can increase the rate of learning(α), `http://www.cs.ox.ac.uk/people/shimon.whiteson/pubs/vanseijenadprl09.pdf`

$$\begin{aligned} Q(S_t,A_t) \leftarrow & Q(S_t,A_t) + \alpha[R_{t+1} + \gamma E_\pi[Q(S_{t+1},A_{t+1}|S_t)] - Q(S_t,A_t)] \\ = & Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1},a) - Q(S_t,A_t)] \end{aligned}$$

Algorithm 10 Expected SARSA

Initialize $Q(s,a)$ arbitrarily for all s,a
for each episode **do**
  Initialize s
  for each step in the episode **do**
    Choose a form s using policy π derived from Q
    take action a, observe r and s'
    $V_{s'} = \sum_a \pi(s',a)Q(s',a)$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma V_{s'} - Q(s,a)]$
    $s \leftarrow s'$
  end for
end for

8.2 Policy-Based Methods

- **Policy search Methods**: Instead of maintaining estimates of value function, search in the space of policies.
- Simpler description, Better convergence, Robust to partial observability.
- Direct policy search via genetic algorithms
- **Policy Gradient Methods**: Policy depends on some parameters θ, action preferences, mean, variance, weights of a neural network.
- Modify policy parameters directly instead of estimating the action values.

$$\max J(\theta) = E(r_t) = \sum_a q_*(a)\pi_\theta(a)$$

$$\text{Gradient Ascent } \theta \leftarrow \theta + \alpha \nabla J(\theta)$$

- We compute the gradient of the performance $J(\theta)$ w.r.t the parameters θ

$$\nabla J(\theta) = \sum_a q_*(a) \nabla \pi_\theta(a) = \sum_a q_*(a) \frac{\nabla \pi_\theta(a)}{\pi_\theta(a)} \pi_\theta(a)$$

- Estimate the gradient from N samples

$$\hat{\nabla} J(\theta) = \frac{1}{N} \sum_{i=1}^{N} r_i \frac{\nabla \pi_\theta(a_i)}{\pi_\theta(a_i)}$$

- **REINFORCE**: `https://people.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf`
- Incremental version

$$\Delta\theta_t = \alpha r_t \frac{\nabla \pi_\theta(a_t)}{\pi_\theta(a_t}$$
$$\Delta\theta_t = \alpha r_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta}$$

- REINFORCE with baseline

$$\Delta\theta_t = \alpha (r_t - b_t) \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta}$$

- Generalize L_{R-I}, consider binary bandit problems with arbitrary rewards.

$$\pi(\theta, a) = \begin{cases} \theta, \text{ if } a = 1 \\ 1 - \theta, \text{ if } a = 0 \end{cases}$$
$$\frac{\partial \ln \pi}{\partial \theta} = \frac{a - \theta}{\theta(1 - \theta)}$$
$$b = 0 \text{ and } \alpha = \rho\theta(1 - \theta)$$
$$\Delta\theta = \rho r (a - \theta)$$

- Set baseling to average of observed rewards

$$b_t = \bar{r}_t = \bar{r}_{t-1} + \beta(r_t - \bar{r}_{t-1})$$

- Using softmax action selection

$$\Delta\theta_i = \alpha(r - \bar{r})(1 - \pi(\Theta, a_i))$$

- Use a Gaussian distribution to select actions

$$\pi(a, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(a-\mu)^2}{2\sigma^2}}$$
$$\Delta\mu = \alpha(r - \bar{r})(a - \mu)$$
$$\Delta\sigma = (\frac{\alpha}{\sigma})(r - \bar{r})((a - \mu)^2 - \sigma^2)$$

- **Policy Gradient Theorem**: The gradient of the performance can be expressed in terms of the gradient of the policy

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

- The policy gradient theorem provides an analytic expression for the gradient of performance with respect to the policy parameter θ
- **REINFORCE: MC Policy Gradient Algorithm**

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$

- The algorithm computes an unbiased estimate of the gradient
- Can be very slow due to high variance in the estimates
- Variance is related to the "recurrence time" or the episode length.

- For problems with large state spaces, the variance becomes unacceptably high.

Algorithm 11 REINFORCE: Monte-Carlo Policy Gradient Algorithm

Input: a differentiable policy parameterization $\pi(a|s,\theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$

for each episode **do**
    Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$
    for each step of the episode $t = 0, 1, ..., T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\theta \leftarrow \theta \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\theta)$
    end for
end for

- The policy gradient theorem can be generalized to include a comparison of the action value to an arbitrary baseline $b(s)$

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s) \nabla \pi(a|s,\theta)$$

- The baseline should be a function or a random variable that does not depend on the action a, in which case, the subtracted quantity is 0.

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

- Update rule of REINFORCE with baseline

$$\theta_{t+1} = \theta_t + \alpha (G_t - b(S_t)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

- Doesn't change the expected value, but has an effect on the variance.

9 Actor-Critic Methods

9.1 Actor-Critic Methods

- Actor-Critic methods learn both a policy and a state-value function simultaneously.
- The policy is referred to as the actor that suggests actions given a state.
- The estimated value function is referred to as the critic. It evaluates actions taken by the actor based on the policy.

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha (G_t - \hat{v}(S_t,w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\
&= \theta_t + \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1},w) - \hat{v}(S_t,w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\
&= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}
\end{aligned}$$

- The G_t term although unbiased causes high variance in the algorithm.
- If we had an estimate of $E_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t)$ with less variance, then we can use that instead of G_t
- In the one-step AC algorithm, we use $\hat{v}$ for both estimating $q_\pi(S_t,A_t)$ and as the baseline.
- The bootstrapping in the update introduces bias but decreases the variance.
- This reduced variance can accelerate learning.

Algorithm 12 One-step Actor-Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s,w)$
Parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$

for each episode **do**
  Initialize S (first state of episode)
  $I \leftarrow 1$
  while S is not terminal (for each time step) **do**
    $A \sim \pi(\cdot|S,\theta)$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$ ▷ if S' is terminal, then $\hat{v}(S',w) = 0$
    $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S,w)$
    $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla \ln \pi(A|S,\theta)$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$
  end while
end for

- **Actor**: Computes the policy π_θ and updates θ
- **Critic**: Typically compute an estimate $\hat{v}(s,w)$ of the state value function. Updates the parameter w

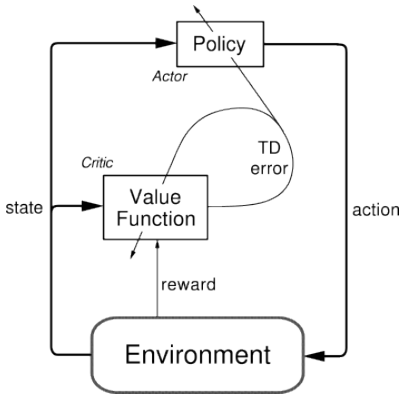


Figure 6: Actor-Critic Architecture

- Basic Actor-Critic Algorithm
  1. Take action $a \sim \pi_\theta(a|s)$ and receive (s,a,s',r)
  2. Update value of parameter w using data $(s, r + \gamma\hat{v}(s',w))$
  3. Compute $\hat{\delta}(s,a) = r + \gamma\hat{v}(s',w) - \hat{v}(s,w)$
  4. $\theta \leftarrow \theta + \alpha\hat{\delta}(s,a)\nabla_\theta \log \pi_\theta(a|s)$
  5. Go back to step 1
- Step 2 of this usually happens in batches. We get multiple data points of the form (s,y)
- Minimize the squared loss, to update critic

$$L(w) = \frac{1}{N}\sum_i \|\hat{v}(s_i,w) - y_i\|^2$$

- We have two design choices as well
  1. **Two Network Design**: Actor and Critic are two different networks
  2. **Shared Network Design**: Actor and Critic are the same network
- The **advantage function** is the difference between the q-value and the value function

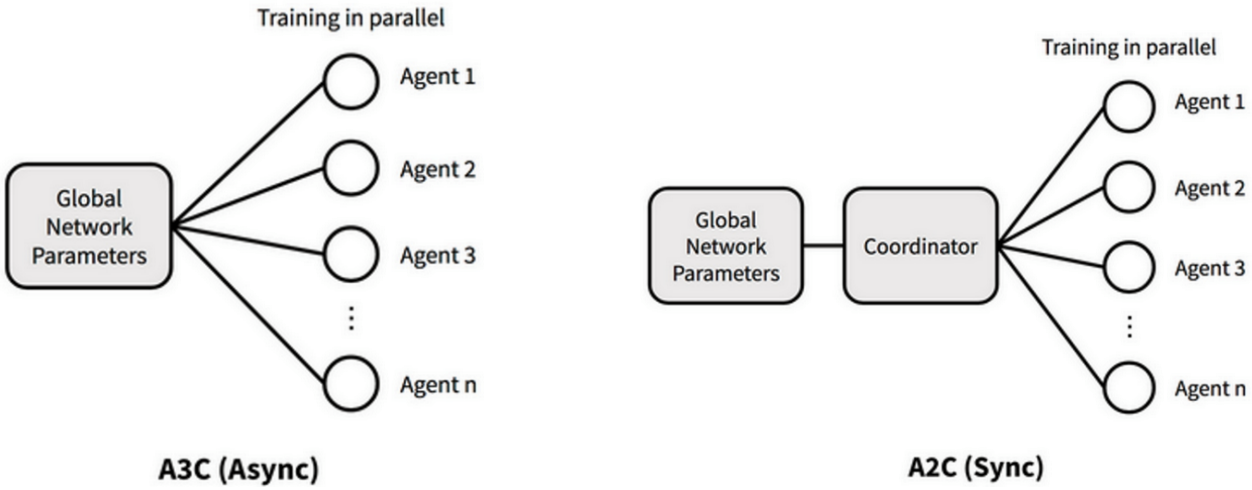
- It can be interpreted as a measure of the advantage of taking action a in state s as compared to following policy π

$$\delta_\pi(s,a) = A_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$$

- **Asynchronous Advantage Actor-Critic (A3C)** and **Synchronous Advantage Actor-Critic (A2C)**: `https://arxiv.org/abs/1602.01783`
- The updates to the global parameters are executed only after all the threads have finished their computation.

Algorithm 13 Asynchronous Advantage Actor-Critic - Pseudocode for each Actor-Learner Thread

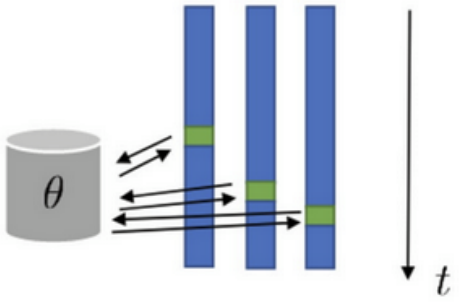
▷ Assume global shared parameters θ and w and global shared counter $T = 0$
▷ Assume thread-specific parameters θ' and w'
Initialize thread step counter $t \leftarrow 1$
repeat
  ▷ **Reset Thread parameters, Update local parameters with global parameters**
  Reset Gradients: $d\theta, dw \leftarrow 0, 0$
  Synchronize thread-specific parameters $\theta' = \theta$ and $w' = w$
  ▷ **Gather Experience**
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t or $t - t_{start} == t_{max}$
  ▷ **Compute the gradients for this thread**
  $R = \begin{cases} 0, & \text{for terminal } s_t \\ V(s_t, w'), & \text{for non-terminal } s_t \text{ (Bootstrap from last state)} \end{cases}$
  for $i \in \{t-1, ..., t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients w.r.t θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R - V(s_i;w'))$
    Accumulate gradients w.r.t w': $dw \leftarrow dw + \frac{\partial (R - V(s_i;w'))^2}{\partial w'}$
  end for
  ▷ **Update Global Parameters**
  Perform asynchronous update of θ using $d\theta$ and w using dw
until $T > T_{max}$



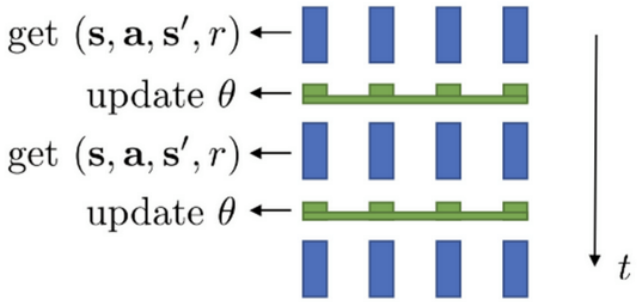
(a) A3C Architecture

(b) A2C Architecture

Figure 7: A3C vs A2C



(c) A3C Update method



(d) A2C Update method

Figure 7: A3C vs A2C

- Substituting the approximation $\hat{q}(s, a, w)$ instead of the true value $q_\pi(s, a)$ may introduce bias.
- It can be proved that there is no bias if the function approximator has a **"compatible" parameterization** with the policy parameterization.

  Condition 1: $\hat{q}(s, a, w) = \nabla_\theta \log \pi_\theta(a|s)^T w$

  Condition 2: w minimizes mean squared error

$$w = \arg\min E_{s\sim\rho^{\pi_\theta}, a\sim\pi_\theta}[(\hat{q}(s, a, w) - q_\pi(s, a))^2]$$

9.2 Advanced Actor-Critic

- Policy Gradient Theorem over Discrete Actions

$$\begin{aligned}\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \theta) \\ &= E_\pi[\sum_a q_\pi(s, a) \nabla \pi(a|s, \theta)]\end{aligned}$$

- Policy Gradient Theorem over Continuous Actions

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \int_S \mu^\pi(s) \int_A \nabla_\theta \pi_\theta(a|s) q_{\pi_\theta}(s, a) da ds \\ &= E_{s\sim\mu^\pi(s), a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) q_{\pi_\theta}(s, a)]\end{aligned}$$

- It is hard to implement differentiable continuous controllers in many problems.
- Continuous actions take expectation over both states and actions.
- If we can reduce the expectation to only over states, this might simplify gradient estimation.
- Let $\pi_\theta : S \to A$ be deterministic, π_θ will output the action to be taken at state s.
- The performance objective, now contains a single integral

$$J(\theta) = \int_S \mu^\pi(s) q_{\pi_\theta}(s, \pi_\theta(s)) ds$$

- Now our gradient also becomes an expectation over only states

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_S \mu^\pi(s) \nabla_a q_\pi(s, a) \nabla_\theta \pi_\theta(s)|_{a=\pi_\theta(s)} ds \\ &= E_{s\sim\mu^\pi}[\nabla_a q_\pi(s, a) \nabla_\theta \pi_\theta(s)|_{a=\pi_\theta(s)} ds]\end{aligned}$$

- We avoided max over actions by moving in the direction of the gradient of q_π.
- For a wide class of stochastic policies, the **deterministic policy gradient** is a limiting case of the stochastic policy gradient

- Update equations in actor critic framework are:

$$\delta_t = r_t + \gamma \hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)$$
$$w \leftarrow w + \alpha_w \delta_t \nabla_w \hat{q}(s_t, a_t, w)$$
$$\theta \leftarrow \theta + \alpha_\theta \nabla_a \hat{q}(s_t, a_t, w) \nabla_\theta \pi_\theta(s_t)|_{a=\pi_\theta(s)}$$

- We can ensure exploration by using off-policy actor critic, where the behavior policy differs from the estimation policy.
- This requires compatible parameterization.
- **Deep DPG**: Combines DPG with DQN
- The algorithm is off-policy with behavior policy being

$$\pi'(s) = \pi_\theta(s) + \mathcal{N} \leftarrow \text{Noise}$$

- Uses replay buffer: alleviates moving target problem.
- Maintain separate target network parameters θ', w' and uses soft updates
- Use Batch Normalization to normalize input state features and minimize covariate shift.
- Critic Network equations

$$\delta_t = r_t + \gamma \hat{q}(s_{t+1}, \pi_{\theta'}(s_{t+1}), w') - \hat{q}(s_t, a_t, w)$$
$$w \leftarrow w + \alpha_w \delta_t \nabla_w \hat{q}(s_t, a_t, w)$$

- Policy Network equation

$$\theta \leftarrow \theta + \alpha_\theta \nabla_a \hat{q}(s_t, a_t, w) \nabla_\theta \pi_\theta(s_t)|_{a=\pi_\theta(s)}$$

Algorithm 14 DDPG Algorithm

Randomly initialize critic network $\hat{q}(s, a, w)$ and actor $\pi_\theta(s)$ with weights w and θ
Initialize target network parameters $w' \leftarrow w$ and $\theta' \leftarrow \theta$
Initialize Replay Buffer $\Re$
for Episode $= 1, ..., M$ **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation s_1
    for $t = 1, ..., T$ **do**
        Select action $a_t = \pi_\theta(s_t) + \mathcal{N}$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\Re$
        Sample a random minibatch of N transitions from $\Re$
        Set $y_i = r_i + \gamma \hat{q}(s_{i+1}, \pi_{\theta'}(s_{i+1}), w')$
        Update critic by minimizing the loss $L = \frac{1}{N} \sum_i (y_i - \hat{q}(s_i, a_i, w))^2$
        Update the actor policy using the sampled policy gradient

$$\nabla_\theta J \approx \frac{1}{N} \sum_i \nabla_a \hat{q}(s_i, a_i, w) \nabla_\theta \pi_\theta(s_i)|_{a=\pi_\theta(s_i)}$$

        Update the target parameters, soft update

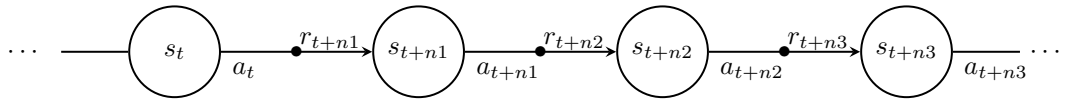
$$w' \leftarrow \tau w + (1 - \tau) w'$$
$$\theta' \leftarrow \tau \theta + (1 - \tau) \theta'$$

    end for
end for

10 Advanced Topics

10.1 Hierarchical Reinforcement Learning

- **Hierarchies**: Natural problem abstraction for humans, divide and conquer
- Can be scaled up easily, ease of reuse, aggressive abstraction possible, and is more explainable.
- HRL introduces a hierarchy of decision-making, where higher-level policies decide abstract actions (options), and lower-level policies execute them. This reduces the search space and allows learning long-term strategies efficiently.
- Essentially, the agent learns skills and reuses them.
- Many RL frameworks: Options, MaxQ, HAM, Airports, etc.
- **Semi-Markov Decision Process(SMDP)**: Generalization of MDP
- The time between decision is a random variable.
- Consider the system remaining in each state for a random waiting before instantaneously transitioning to the next state, aka **holding time**.
- Traditionally modelled as a product of marginals



- Bellman Equation

$$V^*(s) = \max_{a \in A_s} \left[R(s,a) + \sum_{s',\tau} \gamma^\tau P(s', \tau | s, a) V^*(s') \right]$$

$$Q^*(s,a) = R(s,a) + \sum_{s',\tau} \gamma^\tau P(s', \tau | s, a) \max_{a' \in A_{s'}} Q^*(s', a')$$

- For SMDP, one-step Q-learning becomes

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+\tau} + \gamma^\tau \max_a Q(s_{t+\tau}, a) - Q(s_t, a_t)]$$

- We can define three notions of optimality
- **Hierarchically Optimal** policies that give the best solution yet satisfies the proposed hierarchies
- **Recursive Optimal** policies are the best policies formed by putting together component-wise policies obtained by optimizing for each component.
- **Flat Optimal** policies are the same as a regular optimal policy that does not consider any hierarchy.
- **Options Framework**: A generalization of actions to include temporally-extended courses of actions, `https://people.cs.umass.edu/~barto/courses/cs687/Sutton-Precup-Singh-AIJ99.pdf`
- An option is a triple, $o = \langle I, \pi_o, \beta \rangle$
  1. $I \subseteq S$ is the set of states in which o may be started
  2. $\pi_o : \Psi \to [0, 1]$ is the stochastic policy followed during o
  3. $\beta : S \to [0, 1]$ is the probability of terminating in each state
- **Generalizing over tasks**: Each task has a different reward structure in the state space.
- Options provide a model for subtasks, semi Markov Process.
- Options help in transfer learning, long-term planning, and faster convergence in RL.
- We can use generalization of TD, Q-Learning, SARSA, etc. with options.

10.2 Monte Carlo Tree Search

- In online planning, planning is undertaken immediately before executing an action.
- Once an action (or perhaps a sequence of actions) is executed, we start planning again from the new state.
- As such, planning and execution are interleaved such that:
  1. For each state s visited, the set of all available actions $A(s)$ are partially evaluated
  2. $Q(s, a)$ is approximated by averaging the expected reward of trajectories over S obtained by repeated simulations.
  3. The chosen action is $\arg\max_a Q(s, a)$.
- Consider the following tree representing a MDP, where black circles represent actions and white represent states.

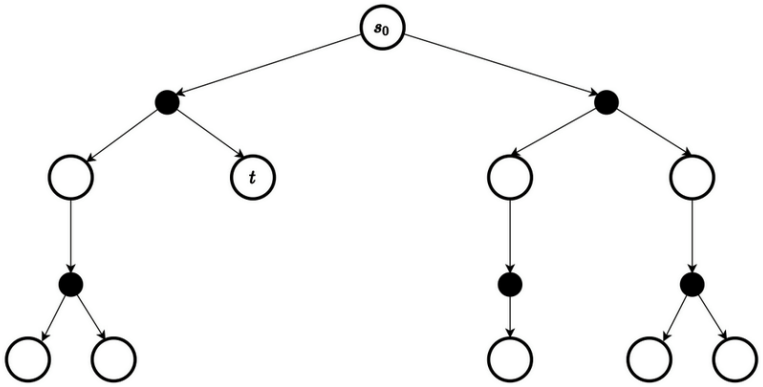


Figure 8: Tree Representing MDP

- **Selection**: Start at the root node, and successively select a child until we reach a node that is not fully expanded.

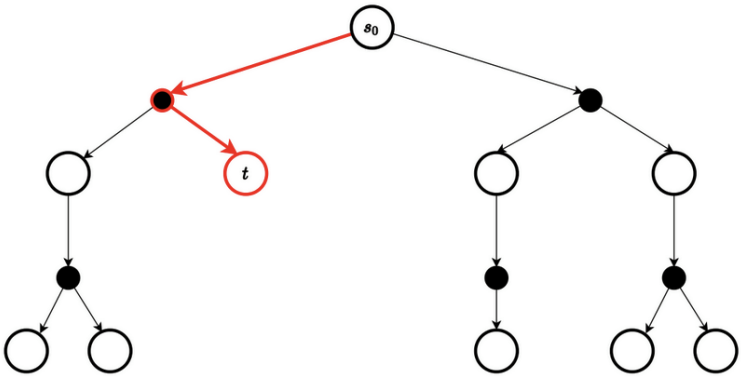


Figure 9: Selection Process

- **Expansion**: Expand the children of the selected node by choosing an action with selection policy or tree policy and creating new nodes using the action outcomes.

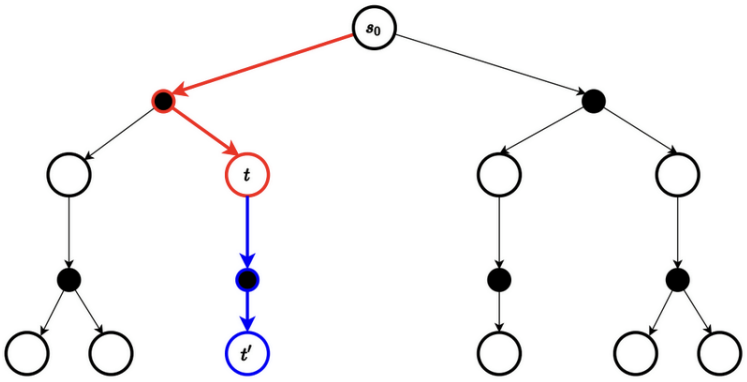


Figure 10: Expansion Process

- **Simulation**: Choose one of the new nodes and perform a random simulation of the MDP to the terminating state with rollout policy

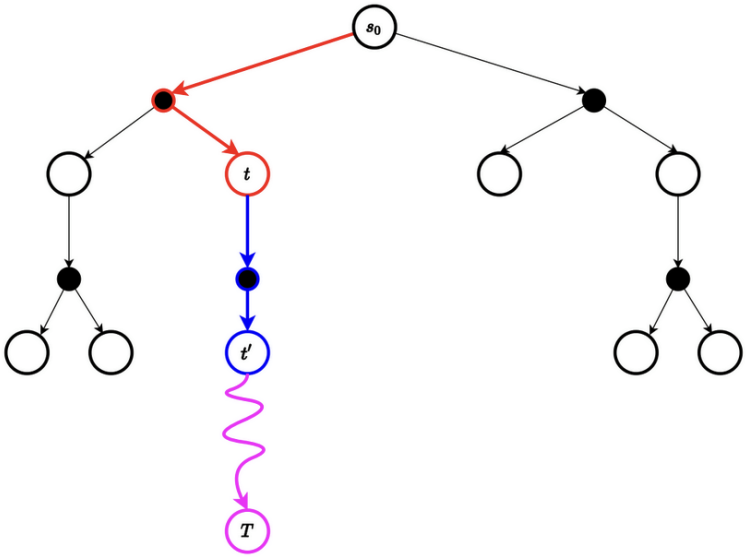


Figure 11: Simulation Process

- **Backup**: Given the reward r at the terminating state, backup the reward to calculate the value $Q(s, a)$ at each state along the path.

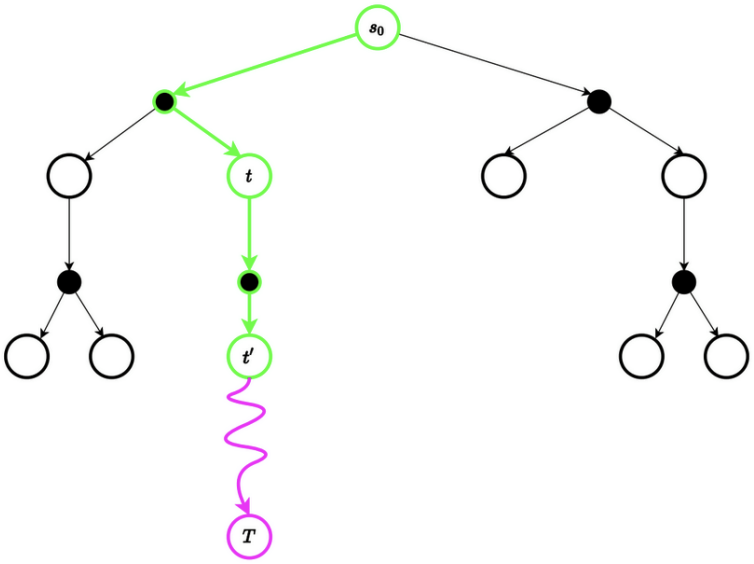


Figure 12: Backup Process

Algorithm 15 Monte Carlo Tree Search

```
function MCTSearch(s_0)
    while within computational budget do
        selectedNode ← Select(s_0)
        child ← Expand(selectedNode)
        G ← Simulate(child)
        Backup(child,G)
    end while
    return arg max_a Q(s_0, a)
end function

function Select(s)
    while s is fully expanded do
        Select action a to apply in s according to tree policy
        Get next state s'
        s ← s'
    end while
    return s
end function

function Expand(s)
    Select action a to apply in s according to tree policy
    Get next state s' and reward r
end function

function Backup(s,G)
    repeat
        N(s, a) ← N(s, a) + 1
        G ← r + γG
        Q(s, a) ← Q(s, a) + 1/N(s,a) [G − Q(s, a)]
        s ← parent of s
    until s ≠ s_0
end function
```

- **Aheuristic**: MCTS doesn't need domain-specific knowledge, making it readily applicable to any domain that may be modelled using a tree.
- **Asymmetric**: the tree selection allows the algorithm to focus on promising nodes, leading to an asymmetric tree expansion.
- **Anytime**: MCTS backups the outcome of each simulation immediately, which ensures all values are always up to-date following every iteration of the algorithm. This allows the algorithm to return an action from the root at any moment in time.
- **UCT**: Upper Confidence bound for Trees, UCB1 + MCTS
- The UCT selection strategy is similar to UCB1 selection strategy

$$\underset{a \in A(s)}{\arg\max} \left[Q(s,a) + 2c\sqrt{\frac{2\ln N(s)}{N(s,a)}} \right]$$

- $N(s)$: The number of times a state node has been visited
- $N(s,a)$: The number of times action a has been selected from this node
- $c > 0$ is the exploration constant
- A few applications for this include Alpha Go Zero, Chess, Shogi, Real Time Strategy games, and Planning and scheduling