US2HQ4 PRINCIPLES OF REINFORCEMENT LEARNING

Unit4

Approximation Solution Methods

On-policy Prediction with Approximation, On-policy control with approximation Eligibility Trace : λ-return - TD(λ) - n-step-Truncated λ-return methods- Online λ-return algorithm - True Online TD(λ); Policy Gradient Methods

* On- Policy Function Approximation

→ In order to extend Tabular RL methods to problems with arbitrarily large state spaces, function approximation (supervised learning) can be used.

→ This allows for generalization from previously encountered (similar) states when the agent encounters a state that has never been seen before.

→ used for large problems like Backgammon, Computer Go, Helicopter motion, ...

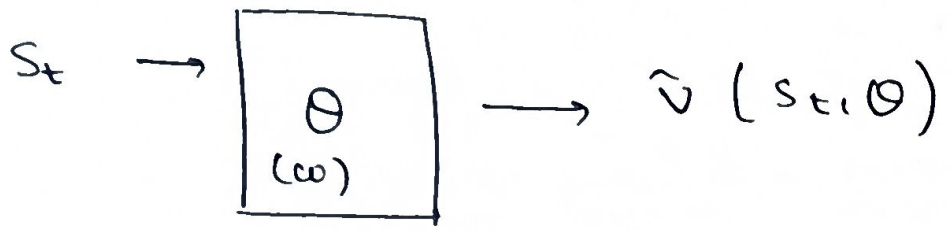
→ For large MDPs, estimate the value function with function approximation as:

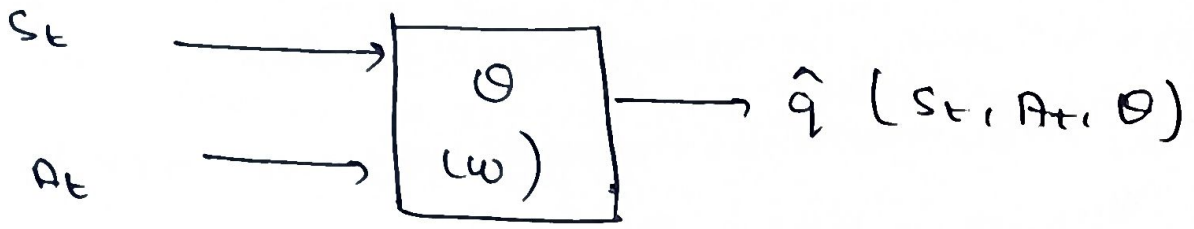
$$\hat{v}(s, w) = v_\pi(s) \quad \text{or} \quad \hat{q}(s, a, w) \approx q_\pi(s, a)$$

represents the value of state s given parameters w, under policy π

w → weights / parameters of the approximating function which might be a neural network, linear model or function

This can be diagrammatically represented as:





Where θ / w are the function approximators.

There are many function approximators. These include:

(i) Linear combination of features

(ii) Neural networks

(iii) Decision Tree

(iv) Nearest Neighbor

(v) Fourier / Wavelet based

* Value - Function Approximation in different RL methods

(i) Monte Carlo → $S_t \rightarrow G_t$

(ii) TD Update $S_t \rightarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, \underline{w_t})$

(iii) DP Update $S_t \rightarrow E_\pi [R_{t+1} + \gamma \hat{v}(S_{t+1}, \underline{w_t}) | S_t = s$

* Why all VFA don't work for RL

→ Not all VFA methods are ideal for RL. This is because certain requirements must be met. These are:

(i) online learning - agent should learn while interacting with environment

(ii) need learning methods that learn from incrementally acquired data

(iii) handle non-stationary target functions

* Prediction Objective in VFA

→ We need a way to evaluate how good our approximated value function is.

→ In the tabular case, lookup tables store the exact values for each state - this doesn't require a continuous measure of accuracy.

→ However, while using VFA, we don't have the exact values for each state. We need a measure of prediction error to know how well our approximation works.

→ To do this; define a distribution $\mu(s)$ which represents the states we care about the most.

The objective fn. is MSE

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

→ The objective is to find a set of weights w^* that minimizes the VE, ideally reaching the global optimum (lowest possible error)

$$VE(w^*) \leq VE(w)$$

→ For simple models like linear approximators, we can often find this global optimum.

→ For complex models, like neural networks, learning usually finds a local optimum (not the best solution) – This is the best that can be done.

* Linear Function Approximation

→ For each state, create a vector called the feature vector

$$x(s) = [x_1, x_2 \ldots x_d]$$

→ Each feature in this vector describes some characteristic of the state.

→ To estimate the value of a state, use this formula

$$\hat{v}(s, w) = \sum_{i=1}^{d} w_i x_i \qquad \text{(inner product)}$$

$$= w_1 x_1(s) + w_2 x_2(s) + \ldots w_d x_d(s)$$

→ Update the weights using Stochastic Gradient Descent. $\nabla \hat{v}(s,w) = x(s)$

→ Each weight is updated as:

$$w_{i,new} = w_{i,old} + \alpha \left(\text{Target} - \hat{v}(s,w)\right) \times x_i(S_t)$$

i.e $$w_{t+1} = w_t + \alpha \left(v_t - \hat{v}(s,w)\right) \times x(S_t)$$

→ By gradually adjusting the weights, the estimated value becomes closer to the actual value.

* How different algorithms update weights with Linear Function Approximation

1. Gradient Monte Carlo - wait till end of each episode

2. TD(0) : The update rule is:

$$w_{i,new} = w_{i,old} + \alpha \left(R_{t+1} + \gamma \hat{v}(s_{t+1}, w) - \hat{v}(s,w) \right) \times x(s_t)$$

TD(0) converges under Linear Function approximation

→ There is a specific set of weights where the updates stop changing the weights. This is called the fixed point. At this point the estimate has become stable.

→ Even if TD(0) doesn't find the best weights, it gets close. The bound is given by

$$VE(w_{TD}) \leq \frac{1}{1-\gamma} \times \text{minimum error}$$

* Non-Linear Function Approximation

→ Some methods include :

(i) ANNs

(ii) Memory-based non-parametric functions

(iii) Kernel-based functions.

→ ANNs are universal function approximators. They generate a hierarchical representation of features, and learn by SGD methods.

* Eligibility Traces

→ Eligibility traces in RL are a mechanism that helps speed up the learning process by allowing updates to a broader range of states and actions even if they were visited in the past.

→ In standard RL, when an agent receives an award, only the current (s,a) pair is updated.

→ However, in many cases, the steps leading to that reward are also important.

→ Eligibility traces allow an RL algorithm to remember and assign credit to all previously visited (s,a) pairs during an episode.

Working of Eligibility Traces

→ Eligibility trace is a value that represents how much credit should be assigned to a particular state or state-action

pair at a given time.

→ When a state/action, its eligibility trace is increased.

→ As time progresses, the eligibility trace decays (exponentially) if the s/a is not visited again.

→ When a reward is obtained, all state action pairs that have non-zero eligibility traces are updated, meaning past actions that lead to the reward are credited.

→ Eligibility traces are updated as:

$$E(s,a) = \gamma \lambda E(s,a) + 1$$

- γ (discount factor): how much future rewards are considered
- λ (trace decay parameter)
- $+1$: added when (s,a) is visited

* Types of Eligibility Traces

1. **Accumulating Traces** → Every time a (s,a) pair is visited, its trace is incremented by 1, and it decays over time

$$E(s,a) = \gamma \lambda E(s,a) + 1$$

2. **Replacing Traces** : When an (s,a) is visited, the trace is reset to 1 instead of being incremented. This method forgets the past, and focuses on the most recent visit

$$E(s,a) = 1$$

→ used when it is important to focus on the latest info & prevent the traces from getting too large.

* Algorithms using Eligibility Traces

1. **TD(λ)** - an extension of TD(0) that introduces eligibility traces
   - updates (s,a) pairs using the decayed eligibility traces
   - λ controls how much weight past states / actions get.

2. **SARSA(λ)** - extends SARSA w/ eligibility traces, allowing it to update a series of (s,a) pairs that lead to a reward, not just the last one.

* Forward and Backward Views of Eligibility Traces of TD(λ)

FORWARD VIEW

→ Forward view focuses on how future rewards affect the current state and action.

→ It describes the process of temporal credit assigned by considering the entire sequence of future rewards and assigning a portion of the expected reward to each past (s,a) pair.

→ λ-return - future rewards up to a certain no. of steps are taken into account

1-step → update value of current state based on the immediate next reward

2-step → considers the reward 2 steps ahead.

λ-return is computed as:

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

Example : Given the reward sequence

$R = [1, 0, 2, -1]$

$\lambda = 0.9$

$\gamma = 0.95$

Compute the λ-return for s_0

$R_0^{(1)} = R_0 = 1$

$R_0^{(2)} = R_0 + \gamma R_1 = 1 + (0.95)(0) = 1$

$R_0^{(3)} = R_0 + \gamma R_1 + \gamma^2 R_2 = 1 + (0.95)(0) + (0.95)^2(2)$

$= 1 + 1.805$

$= 2.805$

$R_0^{(4)} = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma 3 R_3$

$= 1 + (0.95)(0) + (0.95)^2(2) + (0.95)^3(-1)$

$= 2.805 + -0.857$

$= 1.948$

The λ-return formula is:

$$R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{4} \lambda^{n-1} R_t(n)$$

$$R_t^{\lambda} = (1-\lambda) \sum_{i=1}^{4} \lambda^{n-1} R_t^{(n)}$$

$$R_t^{\lambda} = (1-0.9) \times \left((0.9)^0 R_0^1 + (0.9)^1 R_0^2 + (0.9)^2 R_0^3 + (0.9)^3 R_0^4 \right)$$

$$= (0.1) \left(1 + 0.9 + 0.81 \times (2.805)^3 + (0.9)^3 (1.948)^4 \right)$$

$$=$$

BACKWARD VIEW

→ The backward view of TD(λ) provides a causal and incremental mechanism for assigning credit to past actions and states.

→ The forward view relies on knowledge of future events and it isn't feasible for real-time applications.

→ The backward view works by maintaining eligibility traces for each state (s,a) pair.

→ A memory variable traces how recently & frequently each state was visited, allowing past states to receive credits when a reward is observed.

→ The credit assigned decays over time based on the trace decay parameter λ.

Mathematically, the eligibility traces are calculated as

$$e(s) = \begin{cases} \gamma \lambda e(s) + 1 & s = s_t \\ \gamma \lambda e(s) & s \neq s_t \end{cases}$$

(γλ: decay; γ: discount)

→ Next, the Q values are updated in the TD(λ) algorithm

$$Q(s) \leftarrow Q(s) + \alpha \delta e(s)$$

where $\delta = r + Q(s') - Q(s)$

Algorithm

Initialize $V(s) = 0$ & $e(s) = 0$ $\forall s \in S$

Repeat

Initialize s

$a \leftarrow$ action given by π for s

Take action a, observe reward r, and next state s'

$\delta \rightarrow r + \gamma V(s') - V(s)$

$e(s) \leftarrow e(s) + 1$

For all s

$$V(s) \leftarrow V(s) + \alpha \delta e(s)$$

$$e(s) \leftarrow \gamma \lambda e(s)$$

$s \leftarrow s'$

until s is terminal

* **True Online TD(λ)**

→ a method used to improve upon TD(λ), especially when using function approximations.

→ This method introduces a correction term, that is aimed at reducing bias.

→ It also uses a Dutch Trace, which is a specialized version of the eligibility that decays in time in a way that reduces bias even further.

→ Online TD(λ) is also computationally efficient, since it doesn't require backward traces over previous experiences, making it suitable for online learning where updates happen step-by-step.

~~The update rule is:~~

~~$w_{t+1} = w_t + \alpha \delta_t z_t + \alpha (w_t^T s_t - w_{t-1}^T s_t) z_t$~~

Key Modifications in True Online TD(λ)

1. Correction Term: True Online TD(λ) introduces a correction to the parameter update to reduce bias that occurs in regular TD(λ) when using approximations.

2. Eligibility Traces: True Online TD(λ) maintains an eligibility trace, but modifies the way it decays over time, ensuring that the updates reflect the current state of the system more accurately.

3. Efficiency - Unlike standard TD(λ), True Online TD(λ) performs updates more efficiently, without requiring backward passes over previous experiences, making it more suitable for online learning.

The parameter update rule is:

$$\theta_{t+1} = \theta_t + \alpha \delta_t z_t + \alpha \overbrace{(\theta_t^T \phi_t - \theta_{t-1}^T \phi_t)}^{\text{dutch trace.}} z_t$$

α = learning error z_t = eligibility trace

δ_t = TD error

* Policy Gradient Methods

→ Policy Gradient Methods are a class of RL algorithms, where we directly optimize the policy to maximize the expected return.

→ Instead of learning a value-function (as in Q-learning), we parameterize the policy $\pi_\theta(a|s)$, which is the probability of taking an action a in a state s given parameters θ.

Key Characteristics

① **Model-Free** - do not need knowledge of environment dynamics i.e the transition probability

② **Direct Optimization** - focus on finding the optimal policy by maximizing the expected cumulative return

Return & Rewards in Policy Gradient Methods

The return $R(\gamma)$ is the sum of rewards an agent receives over a trajectory $\gamma = \{s_0, a_0, s_1, a_1 \ldots\}$

$$R(\gamma) = \sum_{t=0}^{T-1} R(s_t, a_t)$$

T is the finite time horizon (considering episodic tasks). The goal is to maximize this return over all possible trajectories

Objective Function in Policy Gradient

→ The objective is to find parameters Θ of the policy $\pi_\theta(a|s)$ such that the expected return is maximized

$$J(\pi_\theta) = E_{\pi_\theta}[R(\gamma)]$$

→ This involves:

(a) Defining a stochastic policy - The policy is parameterized (eg. using a neural network) and $\pi_\theta(a|s)$ outputs probabilities

(b) Maximizing $J(\pi_\theta)$ - This is done by using the gradient ascent algorithm.

Policy Gradient Theorem

→ The gradient of the objective $J(\pi_\theta)$ can be computed as.

$$\nabla_\theta J(\pi_\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) R(\gamma)]$$

→ This is the policy gradien theorem, which states that we can compute the gradient of the objective using the gradient of the log probabilities of actions weighted by the return

$\log \pi_\theta(a|s)$ → measures how changes in θ affect the action probabilities

$R(\gamma)$ → acts as the weight / importance of each trajectory

Gradient Ascent For Policy Optimization

→ Once we have the gradient $\nabla_\theta J(\pi_\theta)$, we can update the policy parameters θ using gradient ascent:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

→ This iteratively improves the policy to maximize the expected return.

* Advantages and Disadvantages of the Policy Gradient Method

Advantages → works on continuous action spaces

→ allows for exploration by optimizing probabilistic policies

→ can optimize non-linear and complex policies

@hallenges → high variance in gradient estimates - the return R(J) can vary significantly across trajectories, leading to noisy gradients

→ long time to converge - PGMs need a large numbers of interactions with the environment to converge

Reinforcement Learning Unit 4

CAT 2 Lambda-Return Numerical

Construct the **λ-return** for the state S_0 based on the given reward sequence $R = [1, 0, 1, -1]$, with $\lambda = 0.9$ and $\gamma = 0.95$.

1. Definition of λ-Return

The **λ-return** G_t^λ at time t is a weighted average of the **n-step returns** $G_t^{(n)}$:

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

Where:

- $G_t^{(n)}$ is the **n-step return** at time t:

$$G_t^{(n)} = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^{n-1} R_{t+n-1}$$

- G_t^λ accounts for all possible future rewards with an exponentially decaying weighting factor λ.

2. Step-by-Step Calculation

Reward Sequence:

Given $R = [1, 0, 1, -1]$, we calculate G_0^λ for S_0.

Step 1: Calculate $G_0^{(n)}$ for Different n

We compute the **n-step returns** $G_0^{(n)}$ for $n = 1, 2, 3, 4$:

- **For $n = 1$:**

$$G_0^{(1)} = R_0 = 1$$

- **For $n = 2$:**

$$G_0^{(2)} = R_0 + \gamma R_1 = 1 + 0.95(0) = 1$$

- **For $n = 3$:**

$$G_0^{(3)} = R_0 + \gamma R_1 + \gamma^2 R_2 = 1 + 0.95(0) + (0.95^2)(1) = 1 + 0.9025 = 1.9025$$

- **For $n = 4$:**

$$G_0^{(4)} = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 = 1 + 0.95(0) + (0.95^2)(1) + (0.95^3)(-1)$$

$$G_0^{(4)} = 1 + 0 + 0.9025 + (-0.857375) = 1 + 0.045125 = 1.045125$$

Step 2: Compute G_0^λ

Using the λ-return formula:

$$G_0^\lambda = (1-\lambda)\sum_{n=1}^{4}\lambda^{n-1}G_0^{(n)}$$

Substitute the values:

$$G_0^\lambda = (1-0.9)\left[1(1)+0.9(1)+0.9^2(1.9025)+0.9^3(1.045125)\right]$$

Expand the terms:

$$G_0^\lambda = 0.1\left[1+0.9+(0.81)(1.9025)+(0.729)(1.045125)\right]$$

Calculate each term:

- $0.81 \times 1.9025 = 1.541025$
- $0.729 \times 1.045125 = 0.761892$

Add them:

$$G_0^\lambda = 0.1\left[1+0.9+1.541025+0.761892\right]$$

$$G_0^\lambda = 0.1 \times 4.202917 = 0.420292$$