Part II: Reinforcement Learning in Practice

1. Value Function Approximation

Problem with large MDPs: there are too many states and/or actions to store in memory. It is too slow to learn the value of each state individually

Solution for large MDPs:

- Estimate value function with function approximation
  - $\hat{v}(s, w) \approx v_\pi(s)$
  - or $\hat{q}(s, a, w) \approx q_\pi(s, a)$
- Generalise from seen states to unseen states
- Update parameter w using MC or TD learning

A function can either approximate:

- the state value function. input s; output: approx value of v(s)
- the state action value function with:
  - EITHER input state s and action a; output: approx. value of q(s,a) (ACTIONS IN architecture)
  - OR input state s only; output: all the values of q(s,a) (ACTIONS OUT architecture)

Function approximators can be linear combinations of features, neural networks, decision tree, nearest neighbour, Fourier / wavelet bases...

Here, we'll consider differentiable function approximators: linear combinations of features and neural networks.

Furthermore, **we require a training method that is suitable for non-stationary, non-iid data** (the policy can evolve and improve during training)

Incremental methods

Gradient Descent

Gradient Descent

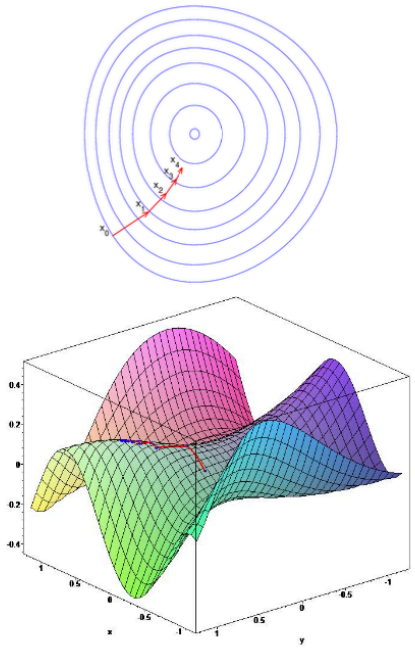
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the *gradient* of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter



Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value fn $\hat{v}(s, \mathbf{w})$ and true value fn $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2 \right]$$

- Gradient descent finds a local minimum

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- Expected update is equal to full gradient update

Feature Vector

We represent state by a feature vector (a compact way to represent the state):

$$x(s) = \begin{pmatrix} x_1(s) \\ \cdot \\ \cdot \\ x_n(s) \end{pmatrix}$$

Incremental Prediction Algorithm

Linear Value Function Approximation

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S)\mathbf{w}_j$$

- Objective function is quadratic in parameters $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \mathbf{x}(S)^\top \mathbf{w})^2 \right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$
$$\Delta \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)$$

Update = *step-size* $\times$ *prediction error* $\times$ *feature value*

Summary of targets used:

Method	Target	Update	Specificities
Generalization	$v_\pi(s)$	$\nabla w = \alpha.\,[v_\pi(s) - \hat{v}(S,w)].\,\nabla_w \hat{v}(S,w)$	/
Monte-Carlo	G_t	$\nabla w = \alpha.\,[G_t - \hat{v}(S_t,w)].\,\nabla_w \hat{v}(S_t,w)$	Monte-Carlo evaluation converges to a local optimum
TD (0)	$R_{t+1} + \gamma.\,\hat{v}(S_{t+1},w)$	$\nabla w = \alpha.\,[R_{t+1} + \gamma.\,\hat{v}(S_{t+1},w) - \hat{v}(S_t,w)].\,\nabla_w \hat{v}(S_t,w)$	Linear TD(0) converges (close) to global optimum
$TD(\lambda)$	G_{t+1}	$\nabla w = \alpha.\,[G_t^\lambda(S) - \hat{v}(S,w)].\,\nabla_w \hat{v}(S_t,w)$	xxx

Replace $\nabla_w \hat{v}(S_t,w)$ by S_t for linear functions

TD(λ) with Value Function Approximation

- The λ-return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to "training data":

$$\left\langle S_1, G_1^\lambda \right\rangle, \left\langle S_2, G_2^\lambda \right\rangle, ..., \left\langle S_{T-1}, G_{T-1}^\lambda \right\rangle$$

- Forward view linear TD(λ)

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t,\mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t,\mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t,\mathbf{w}))\mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD(λ)

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}) \\ E_t &= \gamma\lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

Incremental Control Algorithm

Incremental Control Algorithms

- Like prediction, we must substitute a *target* for $q_\pi(S, A)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For forward-view TD(λ), target is the action-value λ-return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For backward-view TD(λ), equivalent update is

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \\ E_t &= \gamma\lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t \end{aligned}$$

Convergence

TD does not follow the gradient of any objective function This is why TD can diverge when off-policy or using non-linear function approximation. Gradient TD follows true gradient of projected Bellman error.

Gradient Temporal-Difference Learning

- TD does not follow the gradient of *any* objective function
- This is why TD can diverge when off-policy or using non-linear function approximation
- Gradient TD follows true gradient of projected Bellman error

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

Batch Methods

Gradient descent is simple and appealing but it is not sample efficient. Batch methods seek to find the best fitting value function given the agent's experience ("training data").

Least Squares Prediction

Least Squares Prediction

- Given value function approximation $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
- And *experience* $\mathcal{D}$ consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$$

- Which parameters $\mathbf{w}$ give the *best fitting* value fn $\hat{v}(s, \mathbf{w})$?
- Least squares algorithms find parameter vector $\mathbf{w}$ minimising sum-squared error between $\hat{v}(s_t, \mathbf{w})$ and target values v_t^π,

$$\begin{aligned} LS(\mathbf{w}) &= \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2 \\ &= \mathbb{E}_{\mathcal{D}} \left[(v^\pi - \hat{v}(s, \mathbf{w}))^2 \right] \end{aligned}$$

There exists a very simple solution to find the value function approximator (i.e the parameter vector w so that $w^\pi = \underset{w}{argmin}\, LS(w)$). It consists in sampling state,value from experience, i.e. randomizing the state value (instead of going step by step in the incremental method - the effect is the decorrelation across time-steps) and re-using experience data multiple times.

Stochastic Gradient Descent with Experience Replay

Given experience consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$$

Repeat:

1. Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

Converges to least squares solution

$$\mathbf{w}^\pi = \underset{\mathbf{w}}{\text{argmin}}\ LS(\mathbf{w})$$

DQN (Deep Q-Networks)

DQN uses **experience replay** and **fixed Q-targets**.

Experience Replay in Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

- Using variant of stochastic gradient descent

Note that:

- for every batch, we compute the Q-learning target w.r.t old, fixed parameters w^- , which stabilizes the learning process. If we use the same network, it would be unstable. Every x iterations, we replace the old network (with fixed parameters w^-) by the current network.
- we optimize MSE between Q-network (live parameters) and Q-learning targets (old parameters).

The robustness of the approach combining experience replay and fixed Q-learning target was demonstrated on about 50 Atari games (same algorithm).

How much does DQN help?

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

Linear Least Squares Prediction

Experience replay finds least squares solution, but it may take many iterations. Using linear value function approximation $\hat{v}(s, w) = x(s)^{\intercal} w$. We can solve the least squares solution directly

Linear Least Squares Prediction (2)

- At minimum of $LS(\mathbf{w})$, the expected update must be zero

$$\mathbb{E}_{\mathcal{D}}[\Delta \mathbf{w}] = 0$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)(v_t^{\pi} - \mathbf{x}(s_t)^{\top}\mathbf{w}) = 0$$

$$\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi}$$

- For N features, direct solution time is $O(N^3)$
- Incremental solution time is $O(N^2)$ using Shermann-Morrison

However, we do not know true values v_t^{π} so in practice, In practice, our "training data" must use noisy or

biased samples of v_t^π

- LSMC Least Squares Monte-Carlo uses return G_t
- LSTD Least Squares Temporal-Difference uses TD target $R_{t+1} + \gamma . \hat{v}(S_{t+1}, w)$
- LSTD(λ) Least Squares TD(λ) uses λ-return G_t^λ

In each case solve directly for fixed point of MC / TD / TD(λ)

Linear Least Squares Prediction Algorithms (2)

LSMC

$$0 = \sum_{t=1}^{T} \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)\mathbf{x}(S_t)^\top \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t) G_t$$

LSTD

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)(\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t) R_{t+1}$$

LSTD(λ)

$$0 = \sum_{t=1}^{T} \alpha \delta_t E_t$$

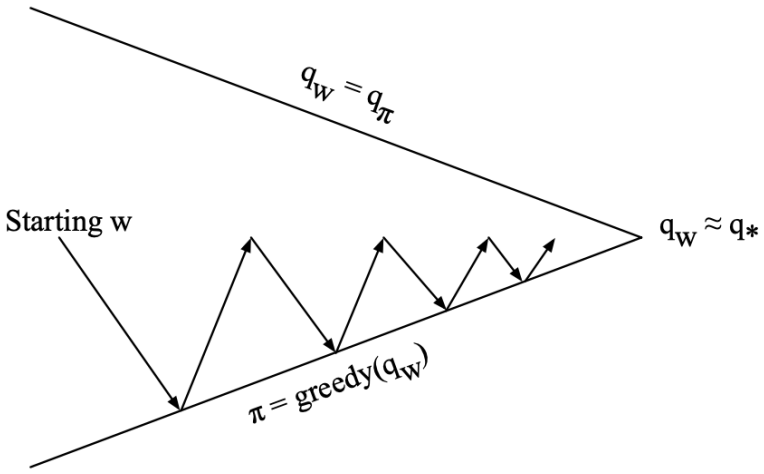
$$\mathbf{w} = \left(\sum_{t=1}^{T} E_t(\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top \right)^{-1} \sum_{t=1}^{T} E_t R_{t+1}$$

Convergence of Linear Least Squares Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Least Squares Control

Least Squares Policy Iteration



Policy evaluation Policy evaluation by least squares Q-learning

Policy improvement Greedy policy improvement

Least Squares Action-Value Function Approximation

- Approximate action-value function $q_\pi(s, a)$
- using linear combination of features $\mathbf{x}(s, a)$

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \mathbf{w} \approx q_\pi(s, a)$$

- Minimise least squares error between $\hat{q}(s, a, \mathbf{w})$ and $q_\pi(s, a)$
- from experience generated using policy π
- consisting of $\langle (state, action), value \rangle$ pairs

$$\mathcal{D} = \{\langle (s_1, a_1), v_1^\pi \rangle, \langle (s_2, a_2), v_2^\pi \rangle, ..., \langle (s_T, a_T), v_T^\pi \rangle\}$$

Least Squares Control

- For policy evaluation, we want to efficiently use all experience
- For control, we also want to improve the policy
- This experience is generated from many policies
- So to evaluate $q_\pi(S, A)$ we must learn off-policy
- We use the same idea as Q-learning:
  - Use experience generated by old policy
    $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
  - Consider alternative successor action $A' = \pi_{new}(S_{t+1})$
  - Update $\hat{q}(S_t, A_t, \mathbf{w})$ towards value of alternative action
    $R_{t+1} + \gamma \hat{q}(S_{t+1}, A', \mathbf{w}))$

Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
LSPI	✓	(✓)	-

(✓) = chatters around near-optimal value function

2. Policy Gradient Methods

Introduction

We will directly parametrise the policy instead of approximating the value function.

$\pi_\theta s, a) = \mathbb{P}[a/s, \theta]$

Advantages of policy-based RL (vs. value-based RL):

- Better convergence properties
- Effective in high-dimensional or continuous action spaces (in value-based RL, there is a max to calculate, which may be computationnally expensive)
- Can learn stochastic policies

Disadvantages:

- Typically converge to a local rather than global optimum
- Evaluating a policy is typically inefficient and high variance

Aliased Gridworld Example

When state-aliasing occurs (i.e. the agent cannot differentiate 2 states, i.e. same feature vector for 2 different states), a stochastic policy does better than a deterministic policy. Policy-based RL can learn the optimal stochastic policy.

Partial observation (POMDP) can be due to the limit of the feature vector representation.

Note that we call aliasing an effect that causes different signals to become indistinguishable when sampled.

Policy Search

The main question is how we measure the quality of a policy π_θ.

First, we need to define the **policy objective function** $J(\theta)$ - several cases:

- In episodic environments: we can use the start value - $J_1^\theta = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$
- In continuing environments:
  - we can use the average value for all the states - $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$
  - or the average reward per time-step - $J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a). R_s^a$

with $d^{\pi_\theta}(s)$ the stationary distribution of Markov chain for π_θ

All follow the same policy gradient.

Now we want to optimize the objective function, i.e. find θ that maximizes $J(\theta)$. Policy-based RL is an

optimization problem.

Approaches

without gradient	with gradient
Hill climbing	Gradient Descent
Simplex / amoeba / Nelder Mead	Conjugate gradient
Genetic algorithms	Quasi Newton

Here, we'll consider **gradient ascent** (maximization).

Finite Difference Policy Gradient

Policy gradient algorithms search for a local maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ $\nabla\theta = \alpha\nabla_\theta J(\theta)$ where $\nabla_\theta J(\theta)$ is the policy gradient and α a step-size parameter.

To evaluate policy gradient of $\pi_\theta(s, a)$, we can apply the following method:

- For each dimension k ∈ [1, n]
  - Estimate kth partial derivative of objective function w.r.t. θ
  - By perturbing θ by small amount ϵ in kth dimension $\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta+\epsilon u_k)-J(\theta)}{\epsilon}$

  where u_k is unit vector with 1 in kth component, 0 elsewhere
- Uses n evaluations to compute policy gradient in n dimensions

This method is simple, noisy, inefficient - but sometimes effective. It works for arbitrary policies, even if policy is not differentiable.

Monte Carlo Policy Gradient

Likelihood Ratios

We now compute the policy gradient analytically. We assume policy π_θ is differntiable almost everywhere. The goal is to compute $\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_d[v_{\pi_\theta}(S)]$. We will use MC samples to compute this gradient.

Score Function

- We now compute the policy gradient *analytically*
- Assume policy π_θ is differentiable whenever it is non-zero
- and we know the gradient $\nabla_\theta \pi_\theta(s, a)$
- Likelihood ratios exploit the following identity

$$\begin{aligned}\nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)\end{aligned}$$

- The score function is $\nabla_\theta \log \pi_\theta(s, a)$

By rewriting the gradient with the score function, we can now easily calculate the expectation.

The gradient of the policy is equal to the policy times the score function. The score function is the gradient of the log of the policy.

Let's use 2 examples:

- Softmax policy

Let's assume a feature vector $\phi(s, a)$ and θ some parameters. Each action is given a weight using linear combinations of features $\phi(s, a)^\intercal \theta$. Then, we convert these weights into probabilities by exponentiating them and dividing them by a nornalizing factor (sum of exponentiated action weights).

$\pi_\theta(s, a) = e^{\phi(s,a)^\intercal \theta} / \sum_b e^{\phi(s,b)^\intercal . \theta}$

Score function = $\nabla_\theta log \pi_\theta(s, a) = \phi(s, a) - \mathbb{E}_{\pi_\theta}[\phi(s, .)]$

(formula explained here here)

Interpretation: The score function, for a given state, is equal to the feature of the action we took minus the average of all the features of the actions we might have taken.

- Gaussian policy

In continuous action spaces, a Gaussian policy is natural. Mean is a linear combination of state features $\mu(s) = \phi(s)^\intercal \theta$ Variance may be fixed σ^2, or can also parametrised. The policy is Gaussian, $a \sim N(\mu(s), \sigma^2)$

We pick the mean and add some noise to be stochastic.

Score function = $\nabla_\theta log \pi_\theta(s,a) = \frac{(a-\mu(s)).\phi(s)}{\sigma^2}$

Again, the score function is quite intuitive : it is the action we actually took minus the mean, multipled by the feature, scaled by the variance.

Policy Gradient Theorem

One-step MDP

One-Step MDPs

- Consider a simple class of one-step MDPs
  - Starting in state $s \sim d(s)$
  - Terminating after one time-step with reward $r = \mathcal{R}_{s,a}$
- Use likelihood ratios to compute the policy gradient

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\
&= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \mathcal{R}_{s,a} \\
\nabla_\theta J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \mathcal{R}_{s,a} \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r]
\end{aligned}
$$

Here, we can see how easy it is to express the gradient of the objective function with an expectation of the policy, thanks to the "trick" of the likelihhod ratio (multiplying and dividing by the policy)

This one-step MDP shows that the adjustment goes in the direction of the reward times the score function

Theorem and Reinforce algorithm

Policy Gradient Theorem

- The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs
- Replaces instantaneous reward r with long-term value $Q^{\pi}(s, a)$
- Policy gradient theorem applies to start state objective, average reward and average value objective

Theorem

For any differentiable policy $\pi_\theta(s, a)$,
for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$,
the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s, a) \; Q^{\pi_\theta}(s, a) \right]$$

Compared to the one-step MDP, we replaced the reward by the long-term value $Q^{\pi}(s, a)$

Monte-Carlo Policy Gradient (REINFORCE)

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

The main issues of this MC policy gradient algorithm is its high variance and very low speed. How to make it more efficient? First, we'll reduce the variance using a critic.

Actor-Critic Policy Gradient

Actor-critic algorithms maintain two sets of parameters:

- **Critic**: Updates **action-value function** parameters w (instead of calculating directly the return)
- **Actor**: Updates policy parameters θ, in direction suggested by critic

Actor-critic algorithms follow an approximate policy gradient.

$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log_{\pi_\theta}(s, a). Q_w(s, a)]$

$\nabla_\theta = \alpha. \nabla_\theta log_{\pi_\theta}(s, a). Q_w(s, a)$

The critic is solving a familiar problem: policy evaluation, i.e. how good is policy π_θ for current parameters θ?. Here, we do not aim at improving the action-value function. In order to estimate the Action-value function, we can use Monte-Carlo policy evaluation Temporal-Difference learning, $TD(\lambda)$ or least-squares policy evaluation.

Action-Value Actor-Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s, a) = \phi(s, a)^\top w$

  Critic Updates w by linear TD(0)

  Actor Updates θ by policy gradient

function QAC

  Initialise s, θ

  Sample $a \sim \pi_\theta$

  for each step **do**

    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$

    Sample action $a' \sim \pi_\theta(s', a')$

    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$

    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$

    $w \leftarrow w + \beta \delta \phi(s, a)$

    $a \leftarrow a', s \leftarrow s'$

  end for

end function

Compatible Function Approximation

Advantage Function Critic

Introducing the advantage function helps reduce the variance. The advantage function represents the benefit of taking an action at a given state vs. the average value of the state. If the value is positive, we will move towards this direction.

Reducing Variance Using a Baseline

- We subtract a baseline function $B(s)$ from the policy gradient
- This can reduce variance, without changing expectation

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) B(s)\right] &= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s,a) B(s) \\ &= \sum_{s\in\mathcal{S}} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(s,a) \\ &= 0\end{aligned}$$

- A good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$
- So we can rewrite the policy gradient using the advantage function $A^{\pi_\theta}(s,a)$

$$\begin{aligned}A^{\pi_\theta}(s,a) &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^{\pi_\theta}(s,a)\right]\end{aligned}$$

The critic should estimate the advantage function, for example, by:

- estimating $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s,a)$
- using 2 function approximators and 2 parameter vectors
- updating both value functions by e.g. TD-learning

There is an easier way (and more used): we can use the TD-error to compute the policy gradient (because TD-error is an unbiased estimate of the advantage function - in other words, the expected value of the TD-error is equal to the advantage function). The main benefit is that we don't need to calculate Q, just V.

Estimating the Advantage Function (2)

- For the true value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ}

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

- is an unbiased estimate of the advantage function

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta}|s,a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s')|s,a\right] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s,a)\end{aligned}$$

- So we can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta^{\pi_\theta}\right]$$

- In practice we can use an approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

- This approach only requires one set of critic parameters v

Critics at Different Time-Scales

- Critic can estimate value function $V_\theta(s)$ from many targets at different time-scales From last lecture...
  - For MC, the target is the return v_t

$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$

  - For TD(0), the target is the TD target $r + \gamma V(s')$

$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$

  - For forward-view TD(λ), the target is the λ-return v_t^λ

$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$

  - For backward-view TD(λ), we use eligibility traces

$$\begin{aligned}\delta_t &= r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\\ e_t &= \gamma\lambda e_{t-1} + \phi(s_t)\\ \Delta\theta &= \alpha\delta_t e_t\end{aligned}$$

Actors at Different Time-Scales

- The policy gradient can also be estimated at many time-scales

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^{\pi_\theta}(s,a)\right]$$

- Monte-Carlo policy gradient uses error from complete return

$$\Delta\theta = \alpha(v_t - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

- Actor-critic policy gradient uses the one-step TD error

$$\Delta\theta = \alpha(r + \gamma V_v(s_{t+1}) - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

Policy Gradient with Eligibility Traces

- Just like forward-view TD(λ), we can mix over time-scales

$$\Delta\theta = \alpha(v_t^\lambda - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

- where $v_t^\lambda - V_v(s_t)$ is a biased estimate of advantage fn
- Like backward-view TD(λ), we can also use eligibility traces
  - By equivalence with TD(λ), substituting $\phi(s) = \nabla_\theta \log \pi_\theta(s, a)$

$$\begin{aligned} \delta &= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \\ e_{t+1} &= \lambda e_t + \nabla_\theta \log \pi_\theta(s, a) \\ \Delta\theta &= \alpha\delta e_t \end{aligned}$$

- This update can be applied online, to incomplete sequences

Deterministic Policy Gradient Theorem (Actor-Critic)

Continuous action spaces Scales well to high-dimensions

Natural Policy Gradient

- Gradient ascent algorithms can follow any ascent direction
- A good ascent direction can significantly speed convergence
- Also, a policy can often be reparametrised without changing action probabilities
- For example, increasing score of all actions in a softmax policy
- The vanilla gradient is sensitive to these reparametrisations

Summary of Policy Gradient Algorithms

- The policy gradient has many equivalent forms

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ v_t\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q^w(s,a)\right] && \text{Q Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^w(s,a)\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta\right] && \text{TD Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta e\right] && \text{TD}(\lambda)\text{ Actor-Critic} \\
G_\theta^{-1}\nabla_\theta J(\theta) &= w && \text{Natural Actor-Critic}
\end{aligned}
$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$

3. Integrating Learning and Planning

Introduction

In previous lessons, we learnt policy directly from experience or learnt value function directly from experience. Now we'll learn model directly from experience and use planning to construct a value function or policy. We'll aslo integrate learning and planning into a single architecture. Learning about a model is about learning the transitions from state to state and the rewards.

Model-Free RL:

- No model
- Learn value function (and/or policy) from experience

Model-Based RL:

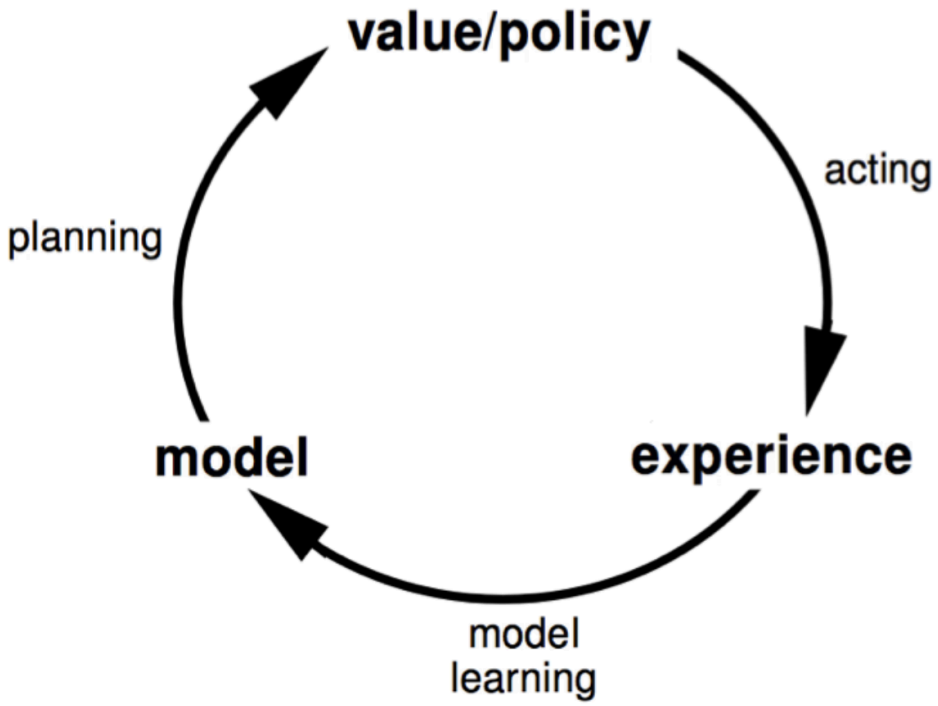
- Learn a model from experience
- Plan value function (and/or policy) from model

At some stage, the model enables the agent to improve its policy without interacting with the environment.

Model-Based Reinforcement Learning

Learning a model

Model-Based RL



Advantages:

- Can efficiently learn model by supervised learning methods
- Can reason about model uncertainty

Disadvantages:

- First learn a model, then construct a value function => two sources of approximation error

What is a Model?

- A *model* $\mathcal{M}$ is a representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, parametrized by η
- We will assume state space $\mathcal{S}$ and action space $\mathcal{A}$ are known
- So a model $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$ represents state transitions $\mathcal{P}_\eta \approx \mathcal{P}$ and rewards $\mathcal{R}_\eta \approx \mathcal{R}$

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

- Typically assume conditional independence between state transitions and rewards

$$\mathbb{P}\left[S_{t+1}, R_{t+1} \mid S_t, A_t\right] = \mathbb{P}\left[S_{t+1} \mid S_t, A_t\right] \mathbb{P}\left[R_{t+1} \mid S_t, A_t\right]$$

The difference between learning about a reward function and a value function is that a value function would look at all the states and the optimal behavior for all the states, which is not the same as rewards which may only appear in the terminal states.

Learning a model is similar as a supervised learning problem:

- learning $s, a \rightarrow r$ is a **regression problem**
- learning $s, a \rightarrow s'$ is a **density estimation problem**

Pick loss function, e.g. mean-squared error, KL divergence, ... Find parameters η that minimise empirical loss

Examples of model:

- Table Lookup Model
- Linear Expectation Model
- Linear Gaussian Model
- Gaussian Process Model
- Deep Belief Network Model
- ...

Process:

- Learn a model
- Use the model only to generate samples

- Sample experience from model

$$S_{t+1} \sim P_\eta(S_{t+1}/S_t, A_t)$$

$$R_{t+1} = R_\eta(R_{t+1}/S_t, A_t)$$

Apply model-free RL to samples, e.g.: Monte-Carlo control, Sarsa Q-learning

The main advantage is that sample-based planning methods are often more efficient.

Performance of model-based RL is limited to optimal policy for approximate MDP $\langle S, A, P_{\cdot\eta}, R_\eta \rangle$, i.e. **Model-based RL is only as good as the estimated model**. When the model is inaccurate, planning process will compute a sub-optimal policy.

Solution 1: when model is wrong, use model-free RL **Solution 2:** reason explicitly about model uncertainty

Planning with a model

Dyna

Simulation-Based Search

Monte Carlo search

MCTS in Go

Temporal-Difference Search

4. Exploration and Exploitation

5. Case study - RL in games