Overview of DeepRL

2025 年 5 月 10 日

Deep Reinforcement Learning (Shuseng Wang)

Chapter 1: Machine Learning Basics

1 Linear Models

Linear models are the simplest class of supervised machine learning models, often considered as single-layer neural networks.

1.1 Linear Regression

This is explained using the example of house price prediction based on house attributes (features). Let $x \in \mathbb{R}^d$ be the feature vector representing d attributes of a house.

1.1.1 Model

The linear model predicts the price y as:

$$f(x; w, b) = x^T w + b$$

Here, $w \in \mathbb{R}^d$ is the weight vector, and $b \in \mathbb{R}$ is the bias. w_j represents the importance of the j-th feature, and b can be seen as a base price independent of specific features. $f(x; w, b) = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$.

1.1.2 Training

The goal is to find parameters $\hat{w}$ and $\hat{b}$ using historical data (training set) $(x_1, y_1), \ldots, (x_n, y_n)$.

1. **Optimization Problem:** Minimize the difference between the predicted price $\hat{y}_i = f(x_i; w, b)$ and the actual price y_i. The loss function is the mean squared error:

$$L(w, b) = \frac{1}{2n} \sum_{i=1}^{n} [f(x_i; w, b) - y_i]^2$$

2. **Solving:** The optimization problem is:

$$(\hat{w}, \hat{b}) = \arg\min_{w,b} L(w, b)$$

While an analytical solution exists, numerical optimization methods like gradient descent are commonly used in practice. These methods iteratively update w and b starting from an initial guess until convergence.

3. **Prediction:** Once $\hat{w}$ and $\hat{b}$ are learned, the prediction for a new house with features x is $\hat{y} = x^T\hat{w} + \hat{b}$.

1.1.3 Regularization

To prevent overfitting when the number of parameters is large or data is limited, a regularization term $R(w)$ is added to the loss function:

$$\min_{w,b} L(w, b) + \lambda R(w)$$

$\lambda > 0$ is a hyperparameter balancing the loss and regularization.

- Ridge Regression uses L_2 regularization: $R(w) = ||w||_2^2 = \sum_{j=1}^{d} w_j^2$.
- LASSO uses L_1 regularization: $R(w) = ||w||_1 = \sum_{j=1}^{d} |w_j|$.

L1 regularization produces sparse solutions and allows automatic feature selection, while L2 regularization does not produce sparse solutions and does not automatically select features The hyperparameter λ is typically chosen using cross-validation on a validation dataset

1.2 Logistic Regression

This model is used for binary classification problems where the target y is either 0 or 1. The example used is cancer detection based on blood test results ($x \in \mathbb{R}^d$).

1.2.1 Model

The linear sigmoid classifier is defined as:

$$f(x; w, b) = \text{sigmoid}(x^T w + b)$$

The sigmoid activation function maps any real number to the $(0, 1)$ interval:

$$\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}$$

The output $\hat{y} = f(x; w, b)$ represents the probability of the input belonging to the positive class ($y = 1$).

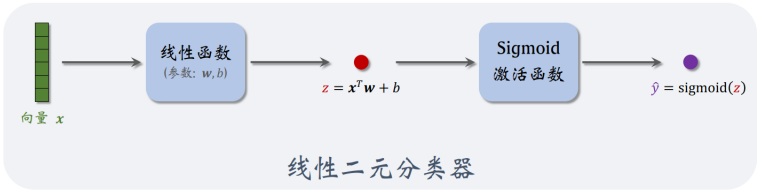


Fig. 1: the structure of linear sigmoid classify

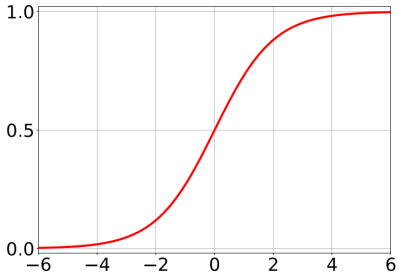


Fig. 2: sigmoid function

1.2.2 Cross Entropy Loss

Cross entropy loss function is commonly used for classification. For two discrete probability distributions $p = [p_1, \ldots, p_m]^T$ and $q = [q_1, \ldots, q_m]^T$, the cross-entropy is:

$$H(p, q) = -\sum_{j=1}^{m} p_j \ln q_j$$

Entropy is a special case: $H(p) = H(p, p)$. Kullback-Leibler (KL) divergence measures the difference between two distributions:

$$KL(p, q) = \sum_{j=1}^{m} p_j \ln \frac{p_j}{q_j} = H(p, q) - H(p)$$

KL divergence does not have symmetry, i.e.

$$KL(p, q) \neq KL(q, p)$$

Since $H(p)$ is constant with respect to q, minimizing $KL(p, q)$ is equivalent to minimizing $H(p, q)$ when p is fixed

1.2.3 Training

1. **Data:** Training set $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i \in \{0, 1\}$.

2. **Optimization Problem:** Represent the true label y_i and prediction $f_i = f(x_i; w, b)$ as proba-

bility vectors $\begin{bmatrix} y_i \\ 1-y_i \end{bmatrix}$ and $\begin{bmatrix} f_i \\ 1-f_i \end{bmatrix}$. The loss is the average cross-entropy:

$$L(w,b) = \frac{1}{n}\sum_{i=1}^{n} H\left(\begin{bmatrix} y_i \\ 1-y_i \end{bmatrix}, \begin{bmatrix} f(x_i;w,b) \\ 1-f(x_i;w,b) \end{bmatrix}\right)$$

With regularization, the problem is:

$$\min_{w,b} L(w,b) + \lambda R(w)$$

3. **Solving:** Numerical optimization methods (like Gradient Descent, SGD, Newton-Raphson, L-BFGS) are used.

1.3 Softmax Classifier

Used for multi-class classification ($k > 2$ classes). The MNIST handwritten digit recognition ($k = 10$) is used as an example.

1.3.1 One-Hot Encoding

Integer labels (0-9) are converted to k-dimensional vectors with one '1' and $k-1$ '0's. E.g., $1 \Rightarrow [0,1,0,\ldots,0]^T$.

1.3.2 Softmax Activation Function

Takes a k-dimensional vector z as input and outputs a k-dimensional vector where elements are non-negative and sum to 1:

$$\text{softmax}(z)_j = \frac{\exp(z_j)}{\sum_{l=1}^{k}\exp(z_l)}$$

The output can be interpreted as a probability distribution over the k classes. It emphasizes the largest input element while retaining information about others.

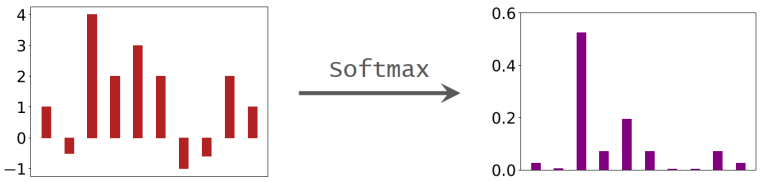


Fig. 3: softmax function

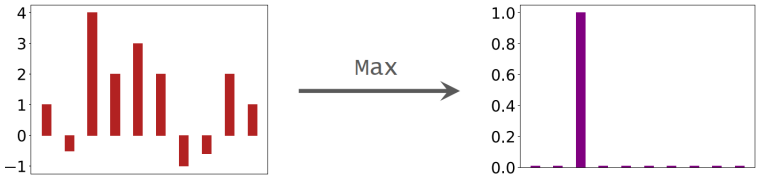


Fig. 4: max function

1.3.3 Model

The linear Softmax classifier is:

$$\pi = \text{softmax}(Wx + b)$$

where $x \in \mathbb{R}^d$, $W \in \mathbb{R}^{k \times d}$, $b \in \mathbb{R}^k$ $(k = 10)$. The j-th element of the output vector π, π_j, is the predicted probability that input x belongs to class j.

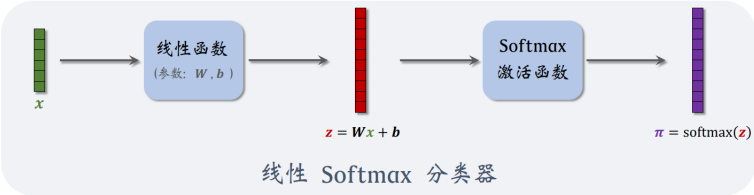


Fig. 5: linear softmax classify

1.3.4 Training

1. **Data:** n samples $(x_1, y_1), \ldots, (x_n, y_n)$, where y_i is the one-hot encoded label vector.
2. **Optimization Problem:** Minimize the difference between the prediction $\pi_i = \text{softmax}(Wx_i + b)$ and the true label y_i. The loss is the average cross-entropy:

$$L(W, b) = \frac{1}{n} \sum_{i=1}^{n} H(y_i, \pi_i)$$

With regularization:

$$\min_{W,b} L(W, b) + \lambda R(W)$$

3. **Solving:** Use gradient descent or SGD starting from random or zero initialization.

2 Neural Networks

Linear models have limited representational power. Stacking layers of "linear function + activation function" creates multi-layer networks with higher accuracy.

2.1 Fully Connected Neural Network (Multi-Layer Perceptron)

2.1.1 Fully Connected Layer

Maps input vector $x \in \mathbb{R}^d$ to output $x' \in \mathbb{R}^{d'}$:

$$z = Wx + b$$

$$x' = \sigma(z)$$

$W \in \mathbb{R}^{d' \times d}$ and $b \in \mathbb{R}^{d'}$ are the layer's parameters. $\sigma(\cdot)$ is the activation function. ReLU ($\text{ReLU}(z)_i = \max\{z_i, 0\}$) is the most common activation for hidden layers.

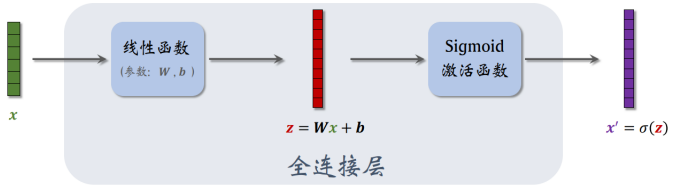


Fig. 6: fully connected layer

2.1.2 Multi-Layer Perceptron (MLP)

Stacking fully connected layers creates an MLP. For an l-layer network:

$$x^{(i)} = \sigma_i(W^{(i)}x^{(i-1)} + b^{(i)}), \quad \text{for } i = 1, \ldots, l$$

where $x^{(0)}$ is the input. Each layer i has parameters $W^{(i)}, b^{(i)}$ and activation σ_i.

- **Implementation:** Requires specifying layer width (output dimension) and activation function. Hidden layers often use ReLU; output layer activation depends on the task (Sigmoid for binary classification, Softmax for multi-class, none for regression).

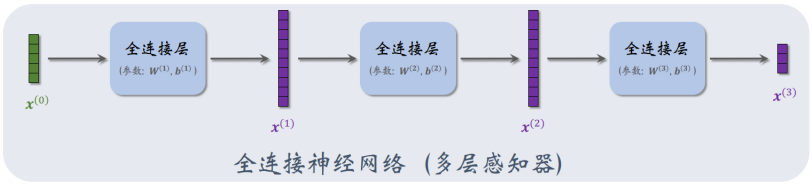


Fig. 7: multi layer Perceptron (MLP)

2.2 Convolutional Neural Network (CNN)

- **Structure:** Primarily composed of convolutional layers, often taking matrices or 3rd-order tensors (e.g., images) as input and producing tensors as output. ReLU activation usually follows

convolutional layers. The final output is typically vectorized into a feature vector. Pooling layers might also be included.

- **Function:** CNNs excel at extracting features from grid-like data such as images. The extracted features can then be fed into fully connected layers for classification or regression.

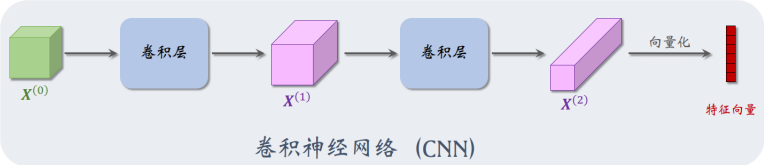


Fig. 8: CNN

Fig. 9: MNIST

3 Backpropagation and Gradient Descent

Training linear models and neural networks involves solving an optimization problem:

$$\min_{w^{(1)},\dots,w^{(l)}} L(w^{(1)},\dots,w^{(l)})$$

Gradient descent and its variants (like SGD) are the most common algorithms.

3.1 Gradient Descent

3.1.1 Gradient

The partial derivative of the loss L with respect to parameter $w^{(i)}$ is denoted as:

$$\nabla_{w^{(i)}} L \triangleq \frac{\partial L}{\partial w^{(i)}}$$

The gradient $\nabla_{w^{(i)}} L$ has the exact same shape as the parameter $w^{(i)}$ (vector, matrix, or tensor).

3.1.2 Gradient Descent (GD)

Updates parameters in the negative gradient direction to minimize the loss:

$$w^{(i)}_{\text{new}} \leftarrow w^{(i)}_{\text{now}} - \alpha \cdot \nabla_{w^{(i)}} L(w^{(1)}_{\text{now}},\dots,w^{(l)}_{\text{now}})$$

$\alpha > 0$ is the learning rate.

3.1.3 Stochastic Gradient Descent (SGD)

Used when the loss is a sum or expectation over samples, $L = \frac{1}{n}\sum_{j=1}^{n} F_j$. SGD updates using the gradient from a single randomly chosen sample j:

$$w_{\text{new}}^{(i)} \leftarrow w_{\text{now}}^{(i)} - \alpha \cdot \nabla_{w^{(i)}} F_j(w_{\text{now}}^{(1)}, \ldots, w_{\text{now}}^{(l)})$$

- **Advantages:** Preferred over GD in practice because it can escape saddle points, is computationally much cheaper (n times faster per step), and often converges faster in wall-clock time. Mini-batch SGD is also common.
- **Variants:** SGD variants like Momentum, AdaGrad, Adam, RMSProp often converge faster than basic SGD by modifying the gradient update rule.

3.2 Backpropagation

3.2.1 Purpose

An algorithm to efficiently compute the gradients of the loss function with respect to all parameters in a deep neural network, which is necessary for GD/SGD.

3.2.2 Principle

Based on Chain Rule

Consider an l-layer MLP (ignoring bias) $x^{(i)} = \sigma_i(W^{(i)}x^{(i-1)})$ and loss $z = H(y, x^{(l)})$. The goal is to compute $\frac{\partial z}{\partial W^{(i)}}$ for all i. (Conceptual dependency graph similar to Fig 1.11).

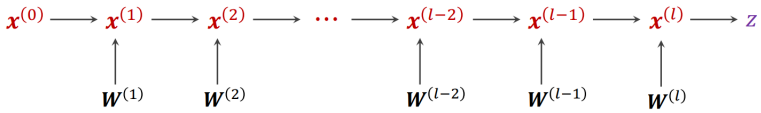


Fig. 10: relation

- **Chain Rule Recap:**
  - Single variable: If $h(x) = g(f(x))$, then $h'(x) = g'(f(x))f'(x)$.
  - Multi-variable: If $f : \mathbb{R}^d \to \mathbb{R}^m$, $g : \mathbb{R}^m \to \mathbb{R}^p$, $h = g \circ f$, the Jacobian matrices multiply:

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial x} \times \frac{\partial g}{\partial f}$$

    (Matrices/tensors are typically vectorized first).

- **Backpropagation Algorithm:**

1. Compute the gradient of the loss w.r.t. the final output: $\frac{\partial z}{\partial x^{(l)}}$.
2. Iterate backwards from $i = l$ down to 1:
   - Given $\frac{\partial z}{\partial x^{(i)}}$, compute the gradient w.r.t. layer i's parameters $W^{(i)}$ (used for updating $W^{(i)}$):
   $$\frac{\partial z}{\partial W^{(i)}} = \frac{\partial x^{(i)}}{\partial W^{(i)}} \cdot \frac{\partial z}{\partial x^{(i)}}$$
   - Compute the gradient w.r.t. layer i's input $x^{(i-1)}$ (to be passed to the previous layer):
   $$\frac{\partial z}{\partial x^{(i-1)}} = \frac{\partial x^{(i)}}{\partial x^{(i-1)}} \cdot \frac{\partial z}{\partial x^{(i)}}$$

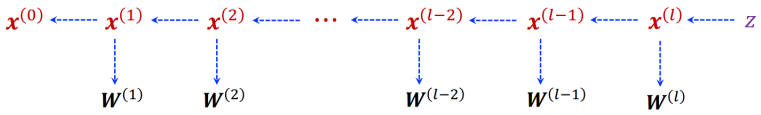


Fig. 11: backpropagation

Chapter 2: Monte Carlo

This chapter covers probability basics crucial for reinforcement learning and introduces Monte Carlo methods

4 Random Variables

4.1 Random Variable vs. Observation

A random variable (denoted by an uppercase letter, e.g., X) is an uncertain quantity whose value depends on a random event (like a coin toss, $X \in \{0, 1\}$). An observation (lowercase, e.g., x) is the actual outcome realized after the event (e.g., observing $x = 1$). Observations are deterministic numbers, while random variables possess randomness before the event occurs.

4.2 Cumulative Distribution Function (CDF)

For a random variable X, the CDF $F_X(x)$ gives the probability that X takes a value less than or equal to x:

$$F_X(x) = \mathbb{P}(X \leq x)$$

4.3 Probability Distributions (PMF & PDF)

- **Probability Mass Function (PMF):** Describes the probability distribution of a *discrete* random variable. For a variable X with possible values in a discrete set $\mathcal{X}$, the PMF is $p(x) = \mathbb{P}(X = x)$. It satisfies $\sum_{x \in \mathcal{X}} p(x) = 1$. Example: A fair coin toss has $p(0) = 0.5, p(1) = 0.5$.
- **Probability Density Function (PDF):** Describes the probability distribution of a *continuous* random variable. For a variable X taking values in a continuous set (like $\mathbb{R}$), the PDF $p(x)$ is related to the CDF by $F_X(x) = \int_{-\infty}^{x} p(u)du$. It satisfies $\int_{\mathbb{R}} p(x)dx = 1$ and $p(x) = F'_X(x)$. Example: The normal distribution $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$. Note that for continuous variables, $p(x) \neq \mathbb{P}(X = x)$ (which is 0).

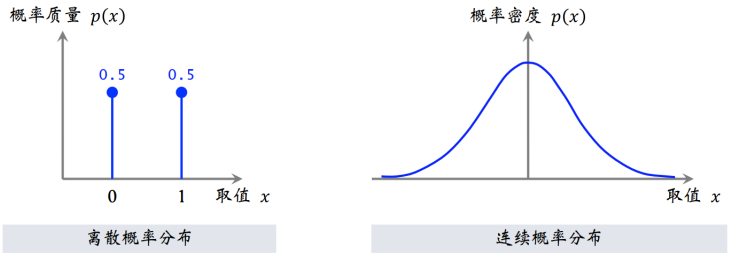


Fig. 12: probability distribution

4.4 Expectation

The expected value of a function $h(X)$ of a random variable X is its average value, weighted by the probability distribution.

- Discrete: $\mathbb{E}_{X \sim p(\cdot)}[h(X)] = \sum_{x \in \mathcal{X}} p(x)h(x)$.
- Continuous: $\mathbb{E}_{X \sim p(\cdot)}[h(X)] = \int_{\mathcal{X}} p(x)h(x)dx$.

Taking the expectation with respect to a variable eliminates that variable from the expression. For example, $\mathbb{E}_{X \sim p(\cdot)}[g(X, Y)]$ results in a function of Y only.

4.5 Random Sampling

The process of drawing an outcome according to a probability distribution. Example: Drawing a ball from a box containing 2 red, 5 green, and 3 blue balls. Before drawing, the color is a random variable X; after drawing, we observe a specific color x (e.g., red).

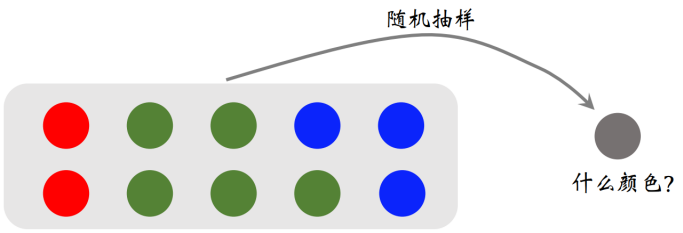


Fig. 13: box

5 Monte Carlo Estimation

Monte Carlo methods are a broad class of algorithms that use repeated random sampling to obtain numerical results, often to estimate some quantity that is difficult to compute directly.

5.1 Example 1: Approximating π

- **Concept:** Estimate π using random sampling.
- **Method:**
  1. Uniformly sample points (x, y) within the square $[-1, 1] \times [-1, 1]$ (Area $a_1 = 4$).
  2. Consider the inscribed unit circle $x^2 + y^2 \leq 1$ (Area $a_2 = \pi$). The probability of a random point falling inside the circle is $p = a_2/a_1 = \pi/4$.
  3. Sample n points. Let m be the count of points falling inside the circle.
  4. The expected number of points inside is $\mathbb{E}[M] = np = n\pi/4$. For large n, the observed count m approximates the expectation: $m \approx n\pi/4$.
  5. Estimate $\pi \approx \frac{4m}{n}$.
- **Accuracy:** The Law of Large Numbers guarantees convergence. The error decreases as $O(1/\sqrt{n})$, meaning convergence is relatively slow.

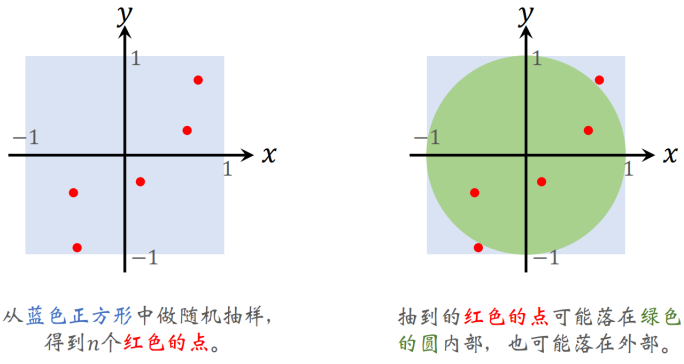


Fig. 14: monte carlo π

5.2 Example 2: Estimating Shaded Area

- **Problem:** Estimate the area of a complex shape, defined by the intersection/difference of simpler shapes.
- **Method:**
  1. Enclose the shape within a simpler region with known area a_1 (e.g., the square $[0, 2] \times [0, 2]$ with area $a_1 = 4$).
  2. Uniformly sample n points within the enclosing region.
  3. Determine the condition for a point (x, y) to be inside the shaded region (e.g., inside green circle $(x-1)^2 + (y-1)^2 \leq 1$ AND outside blue sector $x^2 + y^2 > 4$ for the book's example).
  4. Count the number of points m that fall within the shaded region.
  5. The probability of a point falling in the shaded area is $p = a_2/a_1$, where a_2 is the unknown area. Since $m \approx np = na_2/a_1$, estimate $a_2 \approx \frac{a_1 m}{n}$ (in this case, $a_2 \approx \frac{4m}{n}$).

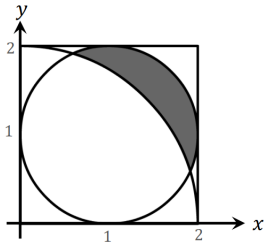


Fig. 15: square1

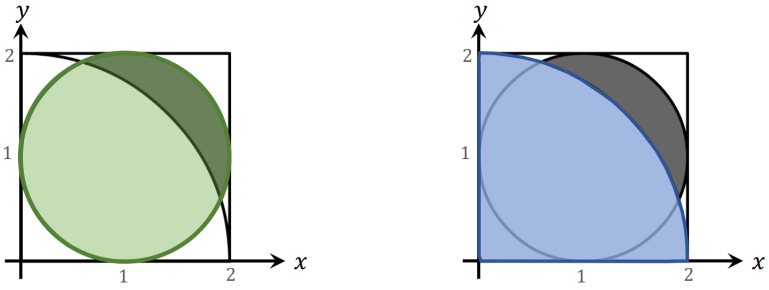


Fig. 16: square2

5.3 Example 3: Approximating Definite Integrals

- **Concept:** Estimate the value of a definite integral, especially when an analytical solution is difficult or impossible.
- **Univariate Case:** Estimate $I = \int_a^b f(x)dx$.

1. Sample $x_1, \ldots, x_n$ uniformly from $[a, b]$.
2. Estimate $I \approx q_n = (b-a) \cdot \frac{1}{n} \sum_{i=1}^{n} f(x_i)$. (This is essentially estimating the average value of $f(x)$ and multiplying by the interval length).

- **Multivariate Case:** Estimate $I = \int_\Omega f(x)dx$, where $x \in \mathbb{R}^d$.
  1. Sample $x_1, \ldots, x_n$ uniformly from the domain Ω.
  2. Calculate the volume $v = \int_\Omega dx$ of the domain Ω. (This might require another MC estimation if Ω is complex).
  3. Estimate $I \approx q_n = v \cdot \frac{1}{n} \sum_{i=1}^{n} f(x_i)$.
- **Connection to π:** Approximating π can be viewed as integrating $f(x, y) = \mathbf{1}_{x^2+y^2 \leq 1}$ over $\Omega = [-1, 1] \times [-1, 1]$ (volume $v = 4$). The integral I is π. The estimate becomes $q_n = 4 \cdot \frac{1}{n} \sum_{i=1}^{n} f(x_i, y_i) = \frac{4m}{n}$, matching the previous result.

5.4 Example 4: Approximating Expectation

- **Concept:** Estimate the expected value $\mathbb{E}_{X \sim p(\cdot)}[f(X)] = \int_\Omega p(x)f(x)dx$.
- **Method (Importance Sampling Perspective):** Instead of uniform sampling over Ω, sample directly from the distribution $p(x)$. This is generally more efficient.
  1. Draw n samples $x_1, \ldots, x_n$ from the distribution $p(\cdot)$.
  2. Estimate the expectation as the sample mean: $\mathbb{E}[f(X)] \approx q_n = \frac{1}{n} \sum_{i=1}^{n} f(x_i)$.
- **Incremental Update (Robbins-Monro Algorithm):** To save memory, update the estimate incrementally as new samples arrive:

$$q_t = (1 - \alpha_t)q_{t-1} + \alpha_t f(x_t)$$

  where $q_0 = 0$. If the learning rate $\alpha_t = 1/t$, this gives the exact sample mean $q_n = \frac{1}{n} \sum_{i=1}^{n} f(x_i)$. The algorithm converges if $\sum \alpha_t = \infty$ and $\sum \alpha_t^2 < \infty$. This form is used in Q-learning.

5.5 Example 5: Stochastic Gradient

- **Concept:** Use Monte Carlo approximation of expectation to understand Stochastic Gradient Descent (SGD).
- **Problem:** Minimize the expected loss $\min_w \mathbb{E}_{X \sim p(\cdot)}[L(X; w)]$, where w are model parameters.
- **Gradient Descent (GD):** Requires the true gradient of the expectation:

$$g = \nabla_w \mathbb{E}_X[L(X; w)] = \mathbb{E}_X[\nabla_w L(X; w)]$$

  The update is $w \leftarrow w - \alpha g$. Computing g directly is often intractable.

- **Stochastic Gradient Descent (SGD):** Approximates the true gradient g using a Monte Carlo estimate based on a mini-batch of B samples $\tilde{x}_1, \ldots, \tilde{x}_B \sim p(\cdot)$:

$$\tilde{g} = \frac{1}{B} \sum_{j=1}^{B} \nabla_w L(\tilde{x}_j; w)$$

  $\tilde{g}$ is an unbiased estimate of g ($\mathbb{E}[\tilde{g}] = g$). The SGD update uses this estimate: $w \leftarrow w - \alpha\tilde{g}$. B is the batch size.

- **Empirical Risk Minimization:** In practice, the true distribution $p(x)$ is often unknown. We use a training dataset $\mathcal{X} = \{x_1, \ldots, x_n\}$ and minimize the empirical risk $\frac{1}{n}\sum_{i=1}^{n} L(x_i; w)$. This corresponds to sampling from the empirical distribution $p(x) = 1/n$ if $x \in \mathcal{X}$ and 0 otherwise. Mini-batch SGD samples B data points uniformly *from the training set* $\mathcal{X}$ to compute $\tilde{g}$.

Chapter 3: Reinforcement Learning Basics

This chapter introduces the fundamental concepts of reinforcement learning.

6 Markov Decision Process (MDP)

RL problems are typically modeled as Markov Decision Processes.

- **Agent and Environment:** The *agent* is the entity making decisions (e.g., Mario in Super Mario, the self-driving car). The *environment* is everything the agent interacts with, defining the rules or dynamics (e.g., the game program, the physical world).

- **State (S or s):** A description of the environment at a specific time. It should ideally contain all relevant information needed for the agent to make a decision (e.g., the current screen in Super Mario, the board configuration in chess). If the agent only sees part of the true state, it's a *partial observation*.

- **State Space ($\mathcal{S}$):** The set of all possible states. It can be discrete or continuous, finite or infinite.

- **Action (A or a):** A decision made by the agent based on the current state (e.g., move left, right, jump).

- **Action Space ($\mathcal{A}$):** The set of all possible actions. It can be discrete or continuous, finite or infinite.

- **Reward (R or r):** A numerical value returned by the environment after the agent takes an action. Rewards guide the learning process. The reward function is often defined by the designer

and typically depends on the current state, action, and next state ($r(s, a, s')$), or sometimes just the state and action ($r(s, a)$). Rewards are assumed to be bounded.

- **State Transition ($p(s'|s, a)$):** Defines the dynamics of the environment. It gives the probability of transitioning to state s' at time $t + 1$, given that the agent was in state s at time t and took action a.

$$p_t(s'|s, a) = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a)$$

  Transitions can be *stochastic* (random, e.g., due to opponent moves or environment randomness) or *deterministic* (always leading to the same next state for a given state-action pair). Deterministic transitions are a special case where the probability is 1 for one specific s' and 0 for all others. RL typically assumes stochastic transitions and often *stationarity*, meaning the transition probabilities $p(s'|s, a)$ do not change over time t.

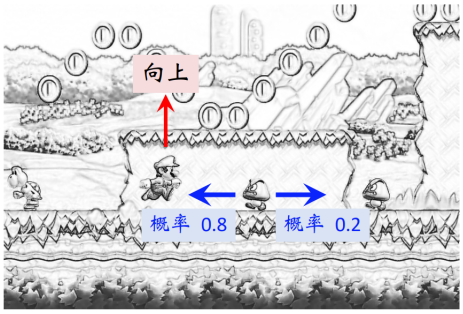


Fig. 17: transition

7 Policy

A policy defines the agent's behavior.

- **Policy (π or μ):** A function that maps states to actions (or distributions over actions). It dictates how the agent chooses an action given a state.
  - **Stochastic Policy ($\pi(a|s)$):** Outputs a probability distribution over the action space $\mathcal{A}$ given the state s.

$$\pi(a|s) = \mathbb{P}(A = a|S = s)$$

    The agent samples an action a from this distribution.
  - **Deterministic Policy ($\mu(s)$):** Directly outputs a single action a for a given state s.

$$a = \mu(s)$$

    This is a special case where the probability distribution is concentrated on one action.

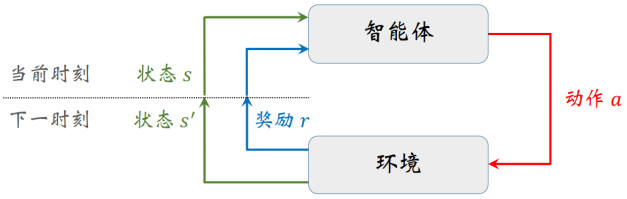


Fig. 18: interaction

- **Agent-Environment Interaction:** The core loop of RL:
  1. Agent observes state s_t.
  2. Agent selects action a_t according to its policy (e.g., $a_t \sim \pi(\cdot|s_t)$).
  3. Environment transitions to a new state $s_{t+1} \sim p(\cdot|s_t, a_t)$.
  4. Environment provides a reward r_t (which might depend on s_t, a_t, s_{t+1}).
  5. The process repeats from the new state s_{t+1}.
- **Episode:** A sequence of interactions from a starting state to a terminal state (or over a finite time horizon). RL often involves training over many episodes. (Distinguished from 'epoch' in supervised learning).

8 Sources of Randomness

Randomness in RL arises from two main sources:

1. **Action Randomness:** If the agent uses a stochastic policy $\pi(a|s)$, the action chosen in state s is random.

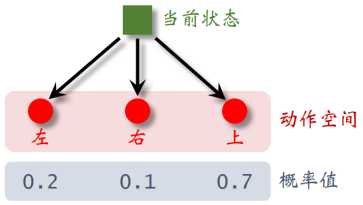


Fig. 19:

2. **State Randomness:** Even if the state s and action a are fixed, the next state s' can be random if the environment's transition function $p(s'|s, a)$ is stochastic.

- **Reward Randomness:** Since the reward R_t often depends on S_t, A_t, and potentially S_{t+1}, it is also a random variable until these quantities are observed.

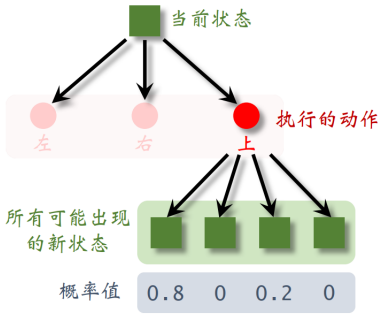


Fig. 20:

- **Markov Property:** The assumption that the next state S_{t+1} and reward R_t depend *only* on the current state S_t and action A_t, not on the history of states and actions before time t.

$$\mathbb{P}(S_{t+1}|S_t, A_t) = \mathbb{P}(S_{t+1}|S_1, A_1, \ldots, S_t, A_t)$$

- **Trajectory:** A sequence of states, actions, and rewards obtained during an episode: $\tau = (s_1, a_1, r_1, s_2, a_2, r_2, \ldots)$. At time t, the history $s_1, a_1, \ldots, a_{t-1}, r_{t-1}, s_t$ is observed, while $A_t, R_t, S_{t+1}, A_{t+1}, \ldots$ are future random variables.

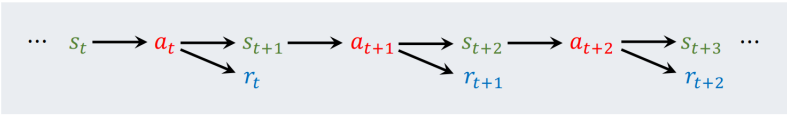


Fig. 21: trajectory

9 Return and Discounted Return

The goal in RL is not just to maximize immediate reward but the cumulative reward over time.

- **Return (U_t):** The sum of future rewards starting from time t until the end of the episode (time n). Also called cumulative future reward.

$$U_t = R_t + R_{t+1} + R_{t+2} + \cdots + R_n$$

The agent aims to find a policy that maximizes the *expected* return.

- **Discounted Return (U_t):** Future rewards are often discounted by a factor $\gamma \in [0, 1]$ per time step. This reflects the preference for immediate rewards over future rewards and ensures convergence in infinite horizon problems.

$$U_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=t}^{n} \gamma^{k-t} R_k$$

γ is the *discount factor*. $\gamma = 0$ means only immediate reward matters; $\gamma \approx 1$ means future rewards are highly valued.

- **Randomness in U_t:** At time t, before the episode ends, U_t is a random variable because it depends on the future random rewards $R_t, R_{t+1}, \ldots, R_n$, which in turn depend on future states and actions $A_t, S_{t+1}, A_{t+1}, \ldots, S_n, A_n$. Once an episode is complete and all rewards $r_1, \ldots, r_n$ are observed, the return $u_t = \sum_{k=t}^{n} \gamma^{k-t} r_k$ is a fixed numerical value.

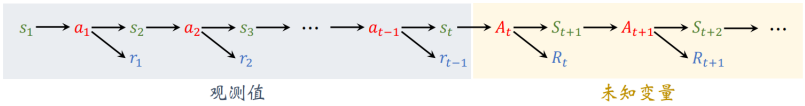


Fig. 22: trajectory reward

- **Finite vs. Infinite Horizon:**
  - **Finite Horizon:** The interaction ends after a fixed number of steps or upon reaching a terminal state. $\gamma = 1$ is permissible.
  - **Infinite Horizon:** The interaction continues indefinitely. $\gamma < 1$ is necessary to keep the discounted return finite (assuming rewards are bounded: $|R_t| < b \implies |U_t| \leq \frac{b}{1-\gamma}$).

10 Value Functions

Value functions estimate the expected return.

- **Motivation:** Estimate the expected value of the random return U_t.
- **Action-Value Function ($Q_\pi(s, a)$):** The expected discounted return starting from state s, taking action a, and subsequently following policy π.

$$Q_\pi(s_t, a_t) = \mathbb{E}_{\pi,p}[U_t | S_t = s_t, A_t = a_t]$$

The expectation is taken over future states (determined by $p(s'|s, a)$) and future actions (determined by $\pi(a'|s')$). $Q_\pi(s, a)$ depends on the state s, the action a, and the policy π. It quantifies how good it is to take action a in state s under policy π.

- **Optimal Action-Value Function ($Q_*(s, a)$):** The maximum possible action-value achievable by *any* policy π.

$$Q_*(s, a) = \max_\pi Q_\pi(s, a)$$

This represents the expected return if the agent takes action a in state s and then follows the *optimal* policy thereafter. It depends only on s and a. Knowing Q_* allows finding the optimal policy: $\pi_*(s) = \arg\max_a Q_*(s, a)$.

- **State-Value Function ($V_\pi(s)$):** The expected discounted return starting from state s and following policy π.

$$V_\pi(s_t) = \mathbb{E}_{\pi,p}[U_t|S_t = s_t]$$

It represents the average value over all actions taken from state s according to policy π:

$$V_\pi(s) = \mathbb{E}_{A\sim\pi(\cdot|s)}[Q_\pi(s, A)] = \sum_{a\in\mathcal{A}} \pi(a|s)Q_\pi(s, a)$$

Also $V_\pi(s_t) = \mathbb{E}_{A_t,S_{t+1},\ldots}[U_t|S_t = s_t]$. $V_\pi(s)$ quantifies how good state s is under policy π.

11 Experimental Environments

Chapter 4: DQN and Q-Learning

This chapter introduces the basics of value learning, focusing on Deep Q-Networks (DQN) and the Q-learning algorithm used to train them.
DQN -> Q_*
SARSA -> Q_π

12 DQN (Deep Q-Network)

This section introduces DQN as a way to approximate the optimal action-value function using neural networks.

12.1 Recap of Fundamentals

- Discounted Return: $U_t = \sum_{k=t}^{n} \gamma^{k-t} R_k$, where $\gamma \in [0, 1]$ is the discount factor.
- Action-Value Function: $Q_\pi(s_t, a_t) = \mathbb{E}[U_t|S_t = s_t, A_t = a_t]$.
- Optimal Action-Value Function: $Q_*(s_t, a_t) = \max_\pi Q_\pi(s_t, a_t)$. This represents the maximum expected return achievable starting from state s_t, taking action a_t, and following the optimal policy thereafter.

12.2 Purpose of Q_*

If Q_* is known, the optimal action in state s_t is $a_t^* = \arg\max_{a\in\mathcal{A}} Q_*(s_t, a)$. Q_* acts like an "oracle" predicting the best possible cumulative future reward for each action.

12.3 Deep Q-Network (DQN): $Q(s, a; w)$

- Since Q_* is usually unknown, we approximate it using a neural network, called DQN, denoted as $Q(s, a; w)$, where w represents the network's parameters.
- **Structure:** Typically, the DQN takes the state s as input (e.g., processed by convolutional layers if s is an image). The output layer provides Q-values for *all* possible discrete actions in the action space $\mathcal{A}$. If $|\mathcal{A}| = k$, the output is a k-dimensional vector $\mathbf{q}$. The notation $Q(s, a; w)$ refers to the specific element in $\mathbf{q}$ corresponding to action a.

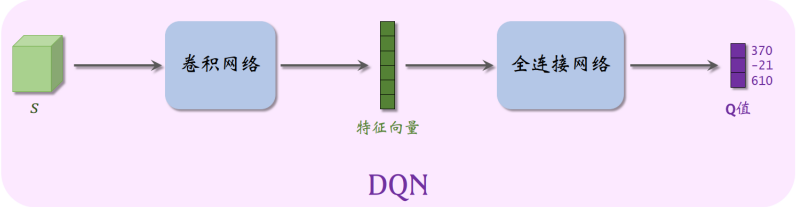


Fig. 23: DQN

- **Goal:** Train the parameters w such that $Q(s, a; w) \approx Q_*(s, a)$ for all s, a.
- **Gradient:** $\nabla_w Q(s, a; w)$ denotes the gradient of the Q-value for action a with respect to the network parameters w. It has the same shape as w.

13 Temporal Difference (TD) Algorithm

This section explains the core idea behind TD learning using an analogy of predicting driving time. TD learning allows updates based on estimates without waiting for the final outcome.

13.1 Driving Time Analogy

Imagine a model $Q(\text{start}, \text{end}; w)$ predicting driving time.

- **Initial Prediction:** Before starting from Beijing to Shanghai, the model predicts $\hat{q} = Q(\text{Beijing}, \text{Shanghai}; w) =$ 14 hours
- **Actual Outcome (Full Journey):** Upon arrival, the actual time taken is $y = 16$ hours. We can update the model using supervised learning: minimize loss $L(w) = \frac{1}{2}(\hat{q} - y)^2$ via gradient descent $w \leftarrow w - \alpha(\hat{q} - y)\nabla_w Q$.
- **Partial Journey & TD:** Suppose the trip is interrupted in Jinan after $r = 4.5$ hours of driving. At Jinan, the model predicts the remaining time to Shanghai as $\hat{q}' = Q(\text{Jinan}, \text{Shanghai}; w) = 11$ hours.

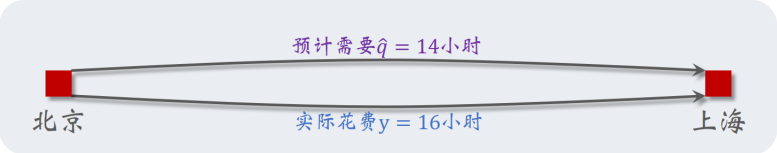


Fig. 24: predict and real

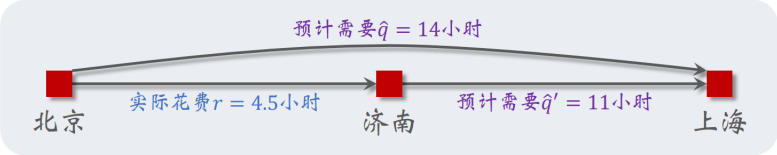


Fig. 25: TD

- **TD Target ($\hat{y}$):** Even without completing the journey, we can form a *better* estimate of the total time by combining the *observed* time r with the *estimated* remaining time $\hat{q}'$:

$$\hat{y} = r + \hat{q}' = 4.5 + 11 = 15.5 \text{ hours}$$

  This $\hat{y}$ is called the TD target. It's more reliable than the initial prediction $\hat{q} = 14$ because it incorporates the actual observed duration r.

- **TD Error (δ):** The difference between the initial estimate and the TD target:

$$\delta = \hat{q} - \hat{y} = 14 - 15.5 = -1.5$$

  Alternatively, δ is the difference between the *estimated* time for the completed segment ($\hat{q} - \hat{q}' = 14 - 11 = 3$ hours) and the *actual* observed time for that segment ($r = 4.5$ hours): $\delta = (\hat{q} - \hat{q}') - r = 3 - 4.5 = -1.5$.

- **TD Update:** Update the original prediction $Q(\text{Beijing}, \text{Shanghai}; w)$ towards the TD target $\hat{y}$. The loss is $L(w) = \frac{1}{2}(\hat{q} - \hat{y})^2 = \frac{1}{2}\delta^2$. Treating $\hat{y}$ (and thus δ) as a constant target when differentiating w.r.t w, the gradient descent update is:

$$w \leftarrow w - \alpha \cdot \delta \cdot \nabla_w Q(\text{Beijing}, \text{Shanghai}; w)$$

14 Using TD to Train DQN

This section applies the TD concept to train the DQN network $Q(s, a; w)$ to approximate $Q_*(s, a)$. The resulting algorithm is specifically Q-learning applied to a neural network.

14.1 Derivation

1. Start with the relationship between returns: $U_t = R_t + \gamma U_{t+1}$.

2. Recall the Bellman Optimality Equation:

$$Q_*(s_t, a_t) = \mathbb{E}_{S_{t+1} \sim p(\cdot|s_t, a_t)}[R_t + \gamma \max_{a'} Q_*(S_{t+1}, a')]$$

3. Use a single observed transition (s_t, a_t, r_t, s_{t+1}) to form a Monte Carlo approximation of the expectation:

$$Q_*(s_t, a_t) \approx r_t + \gamma \max_{a'} Q_*(s_{t+1}, a')$$

4. Replace the true optimal function Q_* with its approximator, the DQN $Q(s, a; w)$:

$$Q(s_t, a_t; w) \approx r_t + \gamma \max_{a'} Q(s_{t+1}, a'; w)$$

5. Define the components analogous to the driving example:

   - Prediction (at time t): $\hat{q}_t = Q(s_t, a_t; w)$
   - TD Target (formed at time $t+1$): $\hat{y}_t = r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; w)$

6. The TD error is $\delta_t = \hat{q}_t - \hat{y}_t$.

7. Define the loss function based on minimizing the TD error (squared):

$$L(w) = \frac{1}{2}[\hat{q}_t - \hat{y}_t]^2 = \frac{1}{2}[Q(s_t, a_t; w) - (r_t + \gamma \max_{a'} Q(s_{t+1}, a'; w))]^2$$

8. Compute the gradient, treating the TD target $\hat{y}_t$ as a fixed constant (this is key to TD methods):

$$\nabla_w L(w) = (\hat{q}_t - \hat{y}_t) \cdot \nabla_w Q(s_t, a_t; w) = \delta_t \cdot \nabla_w Q(s_t, a_t; w)$$

9. The gradient descent update rule for DQN parameters w is:

$$w \leftarrow w - \alpha \cdot \delta_t \cdot \nabla_w Q(s_t, a_t; w)$$

   This is the core Q-learning update for DQN.

14.2 Training Procedure

1. **Data Collection:** Use a *behavior policy* (often ϵ-greedy w.r.t the current DQN) to interact with the environment. The ϵ-greedy policy is:

$$a_t = \begin{cases} \arg\max_a Q(s_t, a; w) & \text{with probability } 1 - \epsilon \\ \text{A random action from } \mathcal{A} & \text{with probability } \epsilon \end{cases}$$

Store the experienced transitions (s_t, a_t, r_t, s_{t+1}) in a *replay buffer* (an array).

2. **DQN Parameter Update (Experience Replay):** Repeatedly:

   - Sample a mini-batch of B transitions (s_j, a_j, r_j, s_{j+1}) from the replay buffer.
   - For each transition in the batch:
     - Calculate the prediction: $\hat{q}_j = Q(s_j, a_j; w_{\text{now}})$
     - Calculate the TD target: $\hat{y}_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; w_{\text{now}})$. (Note: If s_{j+1} is a terminal state, $\hat{y}_j = r_j$).
     - Calculate the TD error: $\delta_j = \hat{q}_j - \hat{y}_j$.
   - Compute the gradient of the loss w.r.t w_{now} averaged over the mini-batch: $g = \frac{1}{B} \sum_{j=1}^{B} \delta_j \nabla_w Q(s_j, a_j; w_{\text{now}})$.
   - Update the DQN parameters: $w_{\text{new}} \leftarrow w_{\text{now}} - \alpha g$.

Data collection and parameter updates can occur concurrently.

15 Q-Learning Algorithm (Tabular)

This section describes the original Q-learning algorithm, which predates DQN and operates on tables when state and action spaces are small and discrete.

15.1 Tabular Representation

If $\mathcal{S}$ and $\mathcal{A}$ are finite, $Q_*(s, a)$ can be represented by a table (matrix) of size $|\mathcal{S}| \times |\mathcal{A}|$.

	第 1 种动作	第 2 种动作	第 3 种动作	第 4 种动作
第 1 种状态	380	-95	20	173
第 2 种状态	-7	64	-195	210
第 3 种状态	152	72	413	-80

Fig. 26: table

15.2 Policy

The optimal policy is derived by choosing the action with the highest Q-value in the current state row: $a_t = \arg\max_{a \in \mathcal{A}} Q_*(s_t, a)$.

15.3 Learning the Q-Table

Approximate Q_* with a table $\tilde{Q}$, initialized (e.g., to zeros). Update entries using TD.

- **Derivation:** Similar to DQN, start from the Bellman optimality approximation $Q_*(s_t, a_t) \approx r_t + \gamma \max_{a'} Q_*(s_{t+1}, a')$. Replace Q_* with the current table estimate $\tilde{Q}$.

- **TD Target (Tabular):** $\hat{y}_t = r_t + \gamma \max_{a' \in \mathcal{A}} \tilde{Q}(s_{t+1}, a')$.
- **Update Rule:** Move the table entry $\tilde{Q}(s_t, a_t)$ towards the target $\hat{y}_t$ with learning rate α:

$$\tilde{Q}(s_t, a_t) \leftarrow (1 - \alpha)\tilde{Q}(s_t, a_t) + \alpha \hat{y}_t$$

  This can be rewritten using the TD error $\delta_t = \tilde{Q}(s_t, a_t) - \hat{y}_t$:

$$\tilde{Q}(s_t, a_t) \leftarrow \tilde{Q}(s_t, a_t) - \alpha \delta_t$$

15.4 Training Procedure

Collect transitions (s_t, a_t, r_t, s_{t+1}) using a behavior policy (e.g., ϵ-greedy on the current $\tilde{Q}$). Store them. Sample a transition (s_j, a_j, r_j, s_{j+1}). Calculate the TD target $\hat{y}_j$ using $\tilde{Q}_{\text{now}}$. Calculate the TD error δ_j. Update the specific entry $\tilde{Q}_{\text{new}}(s_j, a_j) \leftarrow \tilde{Q}_{\text{now}}(s_j, a_j) - \alpha \delta_j$.

16 On-policy vs. Off-policy

These terms describe the relationship between the policy used for data collection and the policy being learned.

- **Behavior Policy:** The policy used by the agent to interact with the environment and generate trajectories (e.g., ϵ-greedy DQN).
- **Target Policy:** The policy that the algorithm is trying to learn and improve (e.g., the greedy policy derived from the learned Q-function: $\pi(s) = \arg\max_a Q(s, a; w)$).
- **Off-policy Learning:** The behavior policy and target policy can be different. Q-learning (both tabular and DQN) is off-policy. It learns the optimal value function Q_* (and thus the optimal target policy) even if the data was collected using a different, exploratory policy (like ϵ-greedy). This decoupling allows the use of *experience replay*, where past experiences collected under older policies can be reused efficiently.
- **On-policy Learning:** The behavior policy and target policy must be the same (or very similar). The agent learns the value function Q_π for the *same* policy π it is currently following. SARSA (covered in the next chapter) is an example. Data collected using an old policy π_{old} is typically discarded and cannot be reused for learning the current policy π_{now}. Standard experience replay is generally not applicable to on-policy methods.

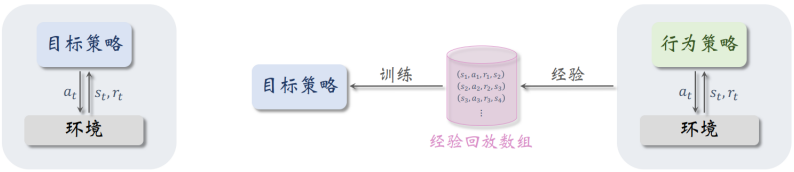


Fig. 27: on policy and off policy

Chapter 5: SARSA Algorithm

This chapter introduces the SARSA algorithm, another fundamental TD learning method. Unlike Q-learning which aims to learn the optimal action-value function Q_*, SARSA aims to learn the action-value function $Q_\pi(s, a)$ for a given policy π. SARSA is often used in actor-critic methods to train the "critic" part, which evaluates the "actor" (policy).

17 Tabular SARSA

This section discusses SARSA in its original tabular form, assuming finite discrete state ($\mathcal{S}$) and action ($\mathcal{A}$) spaces.

17.1 Representation

The action-value function $Q_\pi(s, a)$ is represented as a table (matrix) of size $|\mathcal{S}| \times |\mathcal{A}|$. The values in the table depend on the policy π being followed.

	第 1 种动作	第 2 种动作	第 3 种动作	第 4 种动作
第 1 种状态	380	-95	20	173
第 2 种状态	-7	64	-195	210
第 3 种状态	152	72	413	-80

Fig. 28: tabular SASRA

17.2 Goal

Learn a table $\tilde{q}$ that approximates Q_π.

17.3 Derivation

SARSA is derived from the Bellman equation for Q_π:

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}}[R_t + \gamma Q_\pi(S_{t+1}, A_{t+1}) | S_t = s_t, A_t = a_t]$$

The expectation is over the next state $S_{t+1} \sim p(\cdot|s_t, a_t)$ and the next action $A_{t+1} \sim \pi(\cdot|S_{t+1})$. To approximate the expectation using a single transition (s_t, a_t, r_t, s_{t+1}) and the next action a_{t+1} actually taken *according to the policy π* in state s_{t+1}, we replace the expected value with the sampled value: $r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})$.

17.4 TD Target

Replace Q_π with the current estimate $\tilde{q}$:

$$\hat{y}_t = r_t + \gamma \tilde{q}(s_{t+1}, a_{t+1})$$

17.5 Update Rule

Update the table entry $\tilde{q}(s_t, a_t)$ towards the TD target $\hat{y}_t$:

$$\tilde{q}(s_t, a_t) \leftarrow (1 - \alpha)\tilde{q}(s_t, a_t) + \alpha \hat{y}_t$$

Using the TD error $\delta_t = \tilde{q}(s_t, a_t) - \hat{y}_t$, the update is:

$$\tilde{q}(s_t, a_t) \leftarrow \tilde{q}(s_t, a_t) - \alpha \delta_t$$

17.6 Name Origin

The update uses the sequence of variables encountered: State S_t, Action A_t, Reward R_t, next State S_{t+1}, next Action A_{t+1}. Hence, SARSA.

17.7 Training Loop

1. Initialize $\tilde{q}(s, a)$ arbitrarily (e.g., to 0) for all s, a.
2. For each episode:
   (a) Initialize s_t.
   (b) Choose action a_t from s_t using policy π derived from $\tilde{q}$ (e.g., ϵ-greedy).
   (c) Loop for each step of episode:
      i. Take action a_t, observe reward r_t and next state s_{t+1}.
      ii. Choose next action a_{t+1} from s_{t+1} using policy π (e.g., ϵ-greedy).
      iii. Calculate TD target: $\hat{y}_t = r_t + \gamma \tilde{q}(s_{t+1}, a_{t+1})$.
      iv. Update the table entry: $\tilde{q}(s_t, a_t) \leftarrow \tilde{q}(s_t, a_t) - \alpha(\tilde{q}(s_t, a_t) - \hat{y}_t)$.
      v. Update state and action: $s_t \leftarrow s_{t+1}$, $a_t \leftarrow a_{t+1}$.
   (d) Until s_t is terminal.

Q 学习	近似 $Q_\star$	异策略	可以使用 经验回放
SARSA	近似 Q_π	同策略	不能使用 经验回放

Fig. 29: Q-learning and SARSA

17.8 Comparison to Q-learning (On-policy vs Off-policy)

- SARSA is **on-policy**: The update uses the action a_{t+1} that was actually chosen by the current policy π in state s_{t+1}. It learns the value of the policy being followed.
- Q-learning is **off-policy**: The update uses $\max_{a'} \tilde{q}(s_{t+1}, a')$ which represents the value of the *greedy* (optimal) action, regardless of which action was actually taken by the behavior policy. It learns the value of the optimal policy.
- **Experience Replay:** Because SARSA learns the value of the *current* policy, it cannot effectively reuse data collected under previous, different policies. Therefore, standard experience replay is generally not used with SARSA. Q-learning, being off-policy, can use experience replay.

18 Neural Network SARSA (Value Network)

This section extends SARSA to handle large or continuous state spaces using a neural network approximator for Q_π.

18.1 Value Network $q(s, a; w)$

A neural network that approximates $Q_\pi(s, a)$.

- **Structure:** Typically takes state s as input. If s is high-dimensional (like an image), convolutional layers might be used first. The network outputs a vector of Q-values, one for each discrete action $a \in \mathcal{A}$. w represents the network parameters.

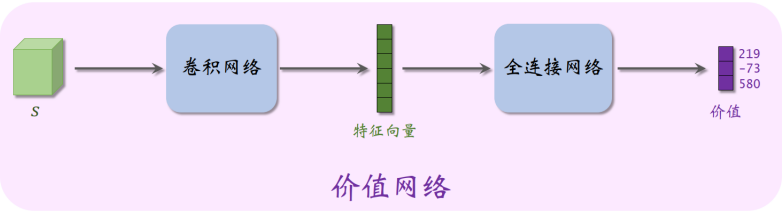


Fig. 30: value network

18.2 Goal

Learn parameters w such that $q(s, a; w) \approx Q_\pi(s, a)$.

18.3 Derivation

Analogous to tabular SARSA. Given a transition (s_t, a_t, r_t, s_{t+1}) and the next action a_{t+1} chosen by policy π in state s_{t+1}.

18.4 TD Target

Use the value network to estimate the value of the next state-action pair:

$$\hat{y}_t = r_t + \gamma q(s_{t+1}, a_{t+1}; w)$$

18.5 Loss Function

Minimize the squared difference between the current prediction and the TD target:

$$L(w) = \frac{1}{2}[q(s_t, a_t; w) - \hat{y}_t]^2$$

18.6 Gradient

Treat $\hat{y}_t$ as a constant target:

$$\nabla_w L(w) = (q(s_t, a_t; w) - \hat{y}_t)\nabla_w q(s_t, a_t; w) = \delta_t \nabla_w q(s_t, a_t; w)$$

where δ_t is the TD error.

18.7 Update

Perform gradient descent on the loss:

$$w \leftarrow w - \alpha \delta_t \nabla_w q(s_t, a_t; w)$$

18.8 Training Loop

1. Initialize network parameters w_{now} and policy parameters θ_{now} (if policy is also learned).
2. Observe state s_t.
3. Sample action $a_t \sim \pi(\cdot|s_t; \theta_{\text{now}})$.
4. Execute a_t, observe reward r_t and next state s_{t+1}.
5. Sample next action $a_{t+1} \sim \pi(\cdot|s_{t+1}; \theta_{\text{now}})$.
6. Compute prediction $\hat{q}_t = q(s_t, a_t; w_{\text{now}})$.
7. Compute target value $\hat{q}_{t+1} = q(s_{t+1}, a_{t+1}; w_{\text{now}})$ (or use a target network).
8. Compute TD target $\hat{y}_t = r_t + \gamma \hat{q}_{t+1}$ and TD error $\delta_t = \hat{q}_t - \hat{y}_t$.

9. Calculate gradient $\nabla_w q(s_t, a_t; w_{\text{now}})$.

10. Update value network parameters: $w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \delta_t \nabla_w q(s_t, a_t; w_{\text{now}})$.

11. Optionally update policy parameters θ using $q(s, a; w)$ (if using actor-critic).

19 Multi-step TD Target

Extends the single-step TD target by incorporating rewards from multiple future steps.

19.1 Motivation

The 1-step TD target relies heavily on the potentially inaccurate value estimate $q(s_{t+1}, a_{t+1}; w)$. Using more steps of actual rewards can provide a more accurate target.

19.2 Derivation

Recall the recursive relationship for the true return U_t:

$$U_t = R_t + \gamma R_{t+1} + \cdots + \gamma^{m-1} R_{t+m-1} + \gamma^m U_{t+m}$$

Taking the expectation conditioned on $S_t = s_t, A_t = a_t$ yields the Bellman equation for m-step returns:

$$Q_\pi(s_t, a_t) = \mathbb{E}\left[\sum_{i=0}^{m-1} \gamma^i R_{t+i} + \gamma^m Q_\pi(S_{t+m}, A_{t+m}) \Big| S_t = s_t, A_t = a_t \right]$$

19.3 Multi-step TD Target ($\hat{y}_t^{(m)}$)

Approximate the expectation using an observed trajectory $(s_t, a_t, r_t, s_{t+1}, a_{t+1}, \ldots, s_{t+m}, a_{t+m})$ and the current value network estimate q:

$$\hat{y}_t^{(m)} = \sum_{i=0}^{m-1} \gamma^i r_{t+i} + \gamma^m q(s_{t+m}, a_{t+m}; w)$$

This target uses m steps of actual rewards $(r_t, \ldots, r_{t+m-1})$ and bootstraps using the value estimate at step $t + m$.

19.4 Update

Use $\hat{y}_t^{(m)}$ as the target for $q(s_t, a_t; w)$. The loss is $L(w) = \frac{1}{2}[q(s_t, a_t; w) - \hat{y}_t^{(m)}]^2$. The update is:

$$w \leftarrow w - \alpha (q(s_t, a_t; w) - \hat{y}_t^{(m)}) \nabla_w q(s_t, a_t; w)$$

19.5 Training Flow

Typically, collect a full episode trajectory first. Then, for each step t in the episode (up to $n-m$), calculate the m-step target $\hat{y}_t^{(m)}$ using the collected rewards and the final value estimate $q(s_{t+m}, a_{t+m}; w)$. Calculate the TD error $\delta_t^{(m)} = q(s_t, a_t; w) - \hat{y}_t^{(m)}$. Update w based on the sum (or average) of gradients over the episode: $w \leftarrow w - \alpha \sum_t \delta_t^{(m)} \nabla_w q(s_t, a_t; w)$.

20 Monte Carlo vs. Bootstrapping

This section compares targets based purely on observed returns (Monte Carlo) versus targets that involve estimates (Bootstrapping), positioning multi-step TD as a bridge between them.

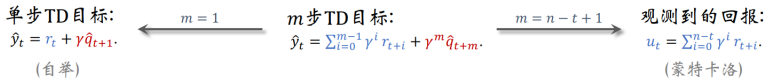


Fig. 31: one step TD, multi step TD and reward

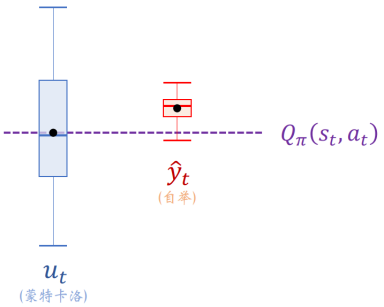


Fig. 32: boxplot

20.1 Monte Carlo (MC) Update

- Uses the full, observed discounted return $u_t = \sum_{k=t}^{n} \gamma^{k-t} r_k$ as the target for $q(s_t, a_t; w)$. Corresponds to $m \to \infty$ (or $m = n - t + 1$ for finite episode length n).
- Loss: $L(w) = \frac{1}{2}[q(s_t, a_t; w) - u_t]^2$.
- Update: $w \leftarrow w - \alpha(q(s_t, a_t; w) - u_t)\nabla_w q(s_t, a_t; w)$.
- **Pros:** The target u_t is an *unbiased* estimate of $Q_\pi(s_t, a_t)$.
- **Cons:** The target u_t has *high variance* because it depends on a potentially long sequence of random actions and state transitions. This often leads to slow convergence.

20.2 Bootstrapping Update (e.g., 1-step TD)

- Uses a target that includes value estimates of future states/actions. The 1-step SARSA target $\hat{y}_t = r_t + \gamma q(s_{t+1}, a_{t+1}; w)$ is an example. The term $q(s_{t+1}, a_{t+1}; w)$ is an estimate used to update another estimate $q(s_t, a_t; w)$ –hence "bootstrapping".
- **Pros:** The target $\hat{y}_t$ has *low variance* compared to the full return u_t, as it depends on only one step of random rewards and transitions plus the function approximator. This often leads to faster convergence.
- **Cons:** The target $\hat{y}_t$ is *biased* if the value estimate $q(s_{t+1}, a_{t+1}; w)$ is inaccurate. This bias can propagate through updates, potentially leading to systematic errors or instability, especially with non-linear function approximators like neural networks.

20.3 Multi-step TD as a Trade-off

The m-step target $\hat{y}_t^{(m)} = \sum_{i=0}^{m-1} \gamma^i r_{t+i} + \gamma^m q(s_{t+m}, a_{t+m}; w)$ balances bias and variance.

- $m = 1$ is pure bootstrapping (lowest variance, potentially highest bias).
- $m \to \infty$ is pure Monte Carlo (unbiased, highest variance).
- Intermediate values of m offer a spectrum between these extremes, potentially achieving better performance than either pure method by finding a good bias-variance trade-off.

Chapter 6: Advanced Techniques for Value Learning

This chapter builds upon the basic DQN and Q-learning concepts from Chapter 4, introducing some advanced techniques to improve performance and stability

21 Experience Replay

Experience replay stores past transitions and reuses them for training updates.

21.1 Standard Experience Replay

- **Mechanism:** Store transitions (s_t, a_t, r_t, s_{t+1}) in a fixed-size replay buffer (size b). When the buffer is full, oldest transitions are discarded. For updates, randomly sample mini-batches of transitions from the buffer.
- **Benefits:**

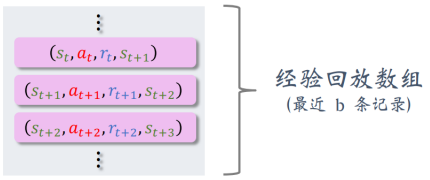


Fig. 33: replay buffer

1. **Breaks Correlations:** Random sampling breaks the temporal correlations inherent in sequential experiences, stabilizing training.
2. **Data Efficiency:** Reuses experiences multiple times, learning more from the collected data.

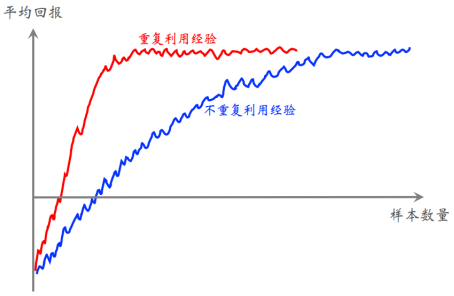


Fig. 34: curve

- **Limitation:** Applicable only to **off-policy** algorithms (like Q-learning/DQN) that can learn from data generated by policies different from the current target policy. Not suitable for on-policy algorithms (like SARSA).

21.2 Prioritized Experience Replay (PER)

- **Motivation:** Not all experiences are equally useful for learning. Transitions with high surprise or error (where the current estimate is poor) should be replayed more often.
- **Prioritization Metric:** Use the magnitude of the TD error $|\delta_j|$ as a measure of priority. A large $|\delta_j|$ indicates a large discrepancy between the prediction $Q(s_j, a_j; w)$ and the TD target y_j, suggesting the transition is "surprising" and valuable for learning.

$$\delta_j = Q(s_j, a_j; w_{\text{now}}) - y_j$$

where $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; w_{\text{target}})$ (using target network notation from next section).

- **Sampling Probability:** Assign a sampling probability p_j to each transition j based on its priority. Common forms include:

1. Proportional: $p_j \propto |\delta_j| + \epsilon$, where ϵ is a small constant ensuring non-zero probability.

2. Rank-based: $p_j \propto 1/\text{rank}(j)$, where $\text{rank}(j)$ is the rank of transition j when sorted by $|\delta_j|$.

- **Importance Sampling (IS) Correction:** Non-uniform sampling introduces bias. To correct this, scale the gradient update for sample j by an importance sampling weight w_j:

$$w_j = \left(\frac{1}{b \cdot p_j}\right)^{\beta}$$

The update rule becomes $w \leftarrow w - \alpha \cdot w_j \cdot \delta_j \cdot \nabla_w Q(s_j, a_j; w)$. $\beta \in [0, 1]$ is a hyperparameter that controls the amount of correction (annealed from an initial value to 1 during training). The term $1/b$ appears because p_j is the probability, while the uniform probability would be $1/b$.

- **Implementation:** The replay buffer stores transitions along with their TD errors (priorities). When a transition is sampled, its TD error is recalculated with the current network, the update is performed using the IS weight, and the transition's priority in the buffer is updated with the new absolute TD error.

序号	四元组	TD 误差	抽样概率	学习率
⋮	⋮	⋮	⋮	⋮
$j-1$	$(s_{j-1}, a_{j-1}, r_{j-1}, s_j)$	δ_{j-1}	$p_{j-1} \propto \lvert\delta_{j-1}\rvert + \epsilon$	$\alpha \cdot (b \cdot p_{j-1})^{-\beta}$
j	(s_j, a_j, r_j, s_{j+1})	δ_j	$p_j \propto \lvert\delta_j\rvert + \epsilon$	$\alpha \cdot (b \cdot p_j)^{-\beta}$
$j+1$	$(s_{j+1}, a_{j+1}, r_{j+1}, s_{j+2})$	δ_{j+1}	$p_{j+1} \propto \lvert\delta_{j+1}\rvert + \epsilon$	$\alpha \cdot (b \cdot p_{j+1})^{-\beta}$
⋮	⋮	⋮	⋮	⋮

Fig. 35: Prioritized Replay buffer

22 Overestimation Problem and Solutions

Q-learning, especially with function approximation like DQN, tends to overestimate the true action values Q_*.

22.1 Causes

1. **Bootstrapping and Bias Propagation:** TD updates use the network's own estimates (e.g., $Q(s_{j+1}, a'; w)$ in the TD target) to update itself. If the network overestimates the value at the next state s_{j+1}, this overestimation propagates back to the estimate at the current state s_j via the TD target y_j.

2. **Maximization Bias:** The max operator in the Q-learning target ($y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; w)$) introduces a positive bias. Even if $Q(s, a; w)$ is an unbiased estimator of $Q_*(s, a)$ (i.e., $Q =$

Q_* + noise with zero mean noise), the expectation of the maximum is greater than or equal to the maximum of the expectation:

$$\mathbb{E}[\max_{a'} Q(s_{j+1}, a'; w)] \geq \max_{a'} \mathbb{E}[Q(s_{j+1}, a'; w)] = \max_{a'} Q_*(s_{j+1}, a')$$

Thus, the TD target y_j systematically overestimates the target based on the true Q_*.

22.2 Harm

Overestimation itself isn't necessarily bad if it's uniform. However, it's typically non-uniform across actions. An action with a lower true Q_* might be significantly overestimated, leading the agent to select suboptimal actions based on the inflated Q-values.

22.3 Solution 1: Target Network

- **Idea:** Decouple the network used for estimating the target value from the network being updated. Introduce a *target network* $Q(s, a; w^-)$ with parameters w^- that lag behind the main DQN parameters w.
- **TD Target Calculation:** Use the target network to evaluate the value of the next state:

$$y_j^- = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; w_{\text{now}}^-)$$

- **Update:** Update the main DQN $Q(s, a; w)$ using the TD error $\delta_j = Q(s_j, a_j; w_{\text{now}}) - y_j^-$.
- **Target Network Update:** Update the target network parameters w^- slowly towards the main network parameters w, typically using a Polyak averaging (soft update):

$$w_{\text{new}}^- \leftarrow \tau w_{\text{new}} + (1 - \tau) w_{\text{now}}^-$$

where $\tau \ll 1$ is a hyperparameter (e.g., 0.001). Alternatively, periodically copy w to w^-.

Fig. 36: Q-learning and target network Q-learning

- **Effect:** Reduces oscillations and instability caused by bootstrapping from rapidly changing values, partially mitigating bias propagation. However, it doesn't directly address the maximization bias.

22.4 Solution 2: Double Q-Learning (DDQN)

- **Idea:** Decouple the *selection* of the best next action from the *evaluation* of that action's value, addressing the maximization bias. Requires a target network.
- **TD Target Calculation:**
  1. Use the current main DQN $Q(s, a; w_{\text{now}})$ to *select* the action that maximizes Q-value in the next state s_{j+1}:
  $$a^* = \arg\max_{a' \in \mathcal{A}} Q(s_{j+1}, a'; w_{\text{now}})$$
  2. Use the target network $Q(s, a; w^-_{\text{now}})$ to *evaluate* the value of this selected action a^*:
  $$y_j^{\text{double}} = r_j + \gamma Q(s_{j+1}, a^*; w^-_{\text{now}})$$
- **Update:** Update the main DQN w using the TD error $\delta_j = Q(s_j, a_j; w_{\text{now}}) - y_j^{\text{double}}$. Update the target network w^- as before.
- **Effect:** Since $Q(s_{j+1}, a^*; w^-)$ is generally less than or equal to $\max_{a'} Q(s_{j+1}, a'; w^-)$, the DDQN target is less prone to overestimation than the standard target network approach. DDQN is highly recommended in practice.

	选择	求值	自举造成偏差	最大化造成高估
原始 Q 学习	DQN	DQN	严重	严重
Q 学习 + 目标网络	目标网络	目标网络	不严重	严重
双 Q 学习	DQN	目标网络	不严重	不严重

Fig. 37: compare

23 Dueling Network Architecture

This technique modifies the DQN architecture itself to learn state values and action advantages separately.

23.1 Value Decomposition

Recall the state-value function $V_\pi(s)$ and the optimal state-value function $V_*(s) = \max_\pi V_\pi(s)$. The *advantage function* $D_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$ measures how much better action a is compared to the average action in state s under policy π. The *optimal advantage function* is $D_*(s, a) = Q_*(s, a) - V_*(s)$.

23.2 Relationship

The optimal action-value can be expressed as:

$$Q_*(s,a) = V_*(s) + D_*(s,a)$$

This holds because it can be shown that $\max_{a' \in \mathcal{A}} D_*(s,a') = 0$.

23.3 Dueling Network Architecture

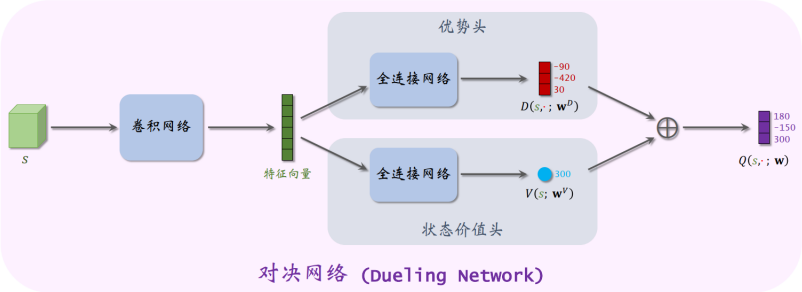


Fig. 38: Dueling network

- The network has a shared body (e.g., convolutional layers) processing the state s.
- The body's output splits into two streams (heads):
  1. A *state-value head* outputs a single scalar $V(s; w_V)$, approximating $V_*(s)$.
  2. An *advantage head* outputs a vector of advantages $D(s,a; w_D)$ for each action $a \in \mathcal{A}$, approximating $D_*(s,a)$.
- The streams are combined to produce the final Q-values. To ensure identifiability (i.e., prevent V and D from being offset by arbitrary constants), the combination uses a baseline adjustment. The standard implementation uses the average advantage as the baseline:

$$Q(s,a;w) = V(s;w_V) + \left(D(s,a;w_D) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} D(s,a';w_D) \right)$$

  where $w = (w_V, w_D)$. The subtraction term enforces that the average advantage is zero, anchoring the decomposition.

23.4 Training

Trained exactly like a standard DQN using algorithms like Double Q-learning. The loss is defined on the final $Q(s,a;w)$ outputs, and gradients propagate back through both streams.

24 Noisy Networks (Noisy Nets)

Improves exploration by adding learnable noise to the network's parameters, replacing traditional exploration strategies like ϵ-greedy.

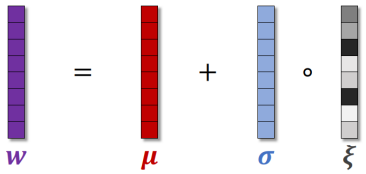


Fig. 39: noise network

24.1 Principle

Replace deterministic linear layers $y = wx + b$ with noisy linear layers. Each weight w_{ij} and bias b_i is replaced by a noisy version:

$$w_{ij} = \mu_{ij}^w + \sigma_{ij}^w \odot \xi_{ij}^w$$

$$b_i = \mu_i^b + \sigma_i^b \odot \xi_i^b$$

Here, μ (mean) and σ (standard deviation) are learnable parameters of the same shape as the original weights/biases. ξ is a random noise variable (e.g., drawn from a standard normal distribution, potentially factorized for efficiency), and $\odot$ denotes element-wise multiplication.

24.2 Noisy DQN

Apply noisy layers (typically fully connected layers) within the DQN architecture. The network output becomes dependent on the noise realization ξ: $Q_{\text{noisy}}(s, a, \xi; \mu, \sigma)$. Parameters are now μ and σ.

24.3 Exploration

The noise injected into the weights causes perturbations in the output Q-values, leading to state-dependent, stochastic action selection, which drives exploration. No external exploration mechanism (like ϵ-greedy) is needed. At each decision step, action is chosen as $a_t = \arg\max_a Q_{\text{noisy}}(s_t, a, \xi_t; \mu, \sigma)$, where ξ_t is resampled.

24.4 Training

Train using standard Q-learning (or DDQN). The TD target also uses the noisy network (with potentially different noise samples): $\hat{y}_j = r_j + \gamma \max_{a'} Q_{\text{noisy}}(s_{j+1}, a', \xi'_j; \mu_{\text{target}}, \sigma_{\text{target}})$. The loss is defined on the noisy Q-values, and gradients are computed w.r.t. the learnable parameters μ and σ.

24.5 Evaluation

During evaluation (after training), the noise is typically turned off by setting $\sigma = 0$. The agent then acts greedily according to the mean weights $Q(s, a; \mu)$. Training with noise encourages robustness to parameter perturbations.

Chapter 7: Policy Gradient Methods

This chapter focuses on policy-based reinforcement learning, where the goal is to directly learn a policy function, often represented by a policy network. It introduces the policy gradient theorem and two fundamental algorithms based on it: REINFORCE and Actor-Critic.

25 Policy Networks

This section introduces the concept of parameterizing the policy using a neural network.

- **Stochastic Policy Network (for Discrete Actions):** The policy network $\pi(a|s;\theta)$ takes the state s as input and outputs a probability distribution over the discrete action space $\mathcal{A}$. θ represents the network parameters.
  - **Output:** Typically uses a Softmax activation function in the final layer to produce probabilities that sum to 1. For example, if $\mathcal{A} = \{\text{left, right, up}\}$, the output might be $\pi(\text{left}|s;\theta) = 0.5, \pi(\text{right}|s;\theta) = 0.2, \pi(\text{up}|s;\theta) = 0.3$.
  - **Structure:** Can use convolutional layers for image-like states or fully connected layers for vector states, followed by fully connected layers and a Softmax output layer.

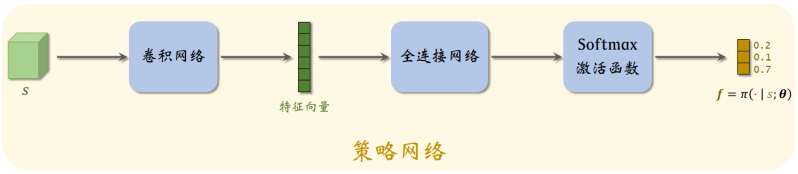


Fig. 40: policy network

  - **Usage:** To select an action a_t in state s_t, sample from the distribution given by the network: $a_t \sim \pi(\cdot|s_t;\theta)$.

26 Objective Function for Policy Learning

The goal of policy learning is to find the optimal parameters θ^* that maximize the expected return.

26.1 Value Functions Recap

- Action-Value: $Q_\pi(s_t, a_t) = \mathbb{E}[U_t | S_t = s_t, A_t = a_t]$
- State-Value: $V_\pi(s_t) = \mathbb{E}_{A_t \sim \pi(\cdot|s_t;\theta)}[Q_\pi(s_t, A_t)]$

26.2 Objective Function $J(\theta)$

The expected state value starting from some initial state distribution (or averaged over the stationary distribution of states under policy π). A common definition is:

$$J(\theta) = \mathbb{E}_{S \sim d^\pi}[V_\pi(S)]$$

(Where d^π is the state distribution induced by policy π. The text simplifies this to $J(\theta) = \mathbb{E}_S[V_\pi(S)]$). A better policy π (i.e., better θ) yields a higher $J(\theta)$.

26.3 Optimization Problem

The goal is to find parameters θ that maximize the objective function:

$$\max_\theta J(\theta)$$

26.4 Gradient Ascent

A common approach is to iteratively update the parameters using gradient ascent:

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \nabla_\theta J(\theta_{\text{now}})$$

where β is the learning rate and $\nabla_\theta J(\theta)$ is the *policy gradient*.

27 Policy Gradient Theorem

This theorem provides an expression for the policy gradient $\nabla_\theta J(\theta)$.
[Policy Gradient Theorem - Simplified Form]

$$\nabla_\theta J(\theta) = \mathbb{E}_{S \sim d^\pi, A \sim \pi(\cdot|S;\theta)}[Q_\pi(S, A) \cdot \nabla_\theta \ln \pi(A|S;\theta)]$$

- The expectation is taken over states S visited under policy π and actions A taken according to π.
- $\nabla_\theta \ln \pi(A|S;\theta)$ is often called the "score function".
- This form is widely used but often omits a scaling factor related to the discount γ and horizon length n, namely $\frac{1-\gamma^n}{1-\gamma}$ (or $1/(1-\gamma)$ for infinite horizon), which gets absorbed into the learning rate. It also assumes states S are sampled from the stationary distribution d^π.

27.1 Simplified Derivation

1. Start with $V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s;\theta) Q_\pi(s,a)$.

2. Differentiate w.r.t. θ:
$$\frac{\partial V_\pi(s)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_a \pi(a|s;\theta) Q_\pi(s,a)$$

3. Apply product rule (treating Q_π as dependent on θ):
$$\frac{\partial V_\pi(s)}{\partial \theta} = \sum_a \frac{\partial \pi(a|s;\theta)}{\partial \theta} Q_\pi(s,a) + \sum_a \pi(a|s;\theta) \frac{\partial Q_\pi(s,a)}{\partial \theta}$$

4. Use the identity $\frac{\partial \pi}{\partial \theta} = \pi \frac{\partial \ln \pi}{\partial \theta}$:
$$\frac{\partial V_\pi(s)}{\partial \theta} = \sum_a \pi(a|s;\theta) \frac{\partial \ln \pi(a|s;\theta)}{\partial \theta} Q_\pi(s,a) + \mathbb{E}_{A\sim\pi}\left[\frac{\partial Q_\pi(s,A)}{\partial \theta}\right]$$
$$\frac{\partial V_\pi(s)}{\partial \theta} = \mathbb{E}_{A\sim\pi}\left[\frac{\partial \ln \pi(A|s;\theta)}{\partial \theta} Q_\pi(s,A)\right] + \underbrace{\mathbb{E}_{A\sim\pi}\left[\frac{\partial Q_\pi(s,A)}{\partial \theta}\right]}_{\text{Term } x}$$

5. Take expectation over states $S \sim d^\pi$:
$$\nabla_\theta J(\theta) = \mathbb{E}_S\left[\frac{\partial V_\pi(S)}{\partial \theta}\right] = \mathbb{E}_{S,A}\left[\nabla_\theta \ln \pi(A|S;\theta) Q_\pi(S,A)\right] + \mathbb{E}_S[x]$$

6. The simplified theorem ignores the term $\mathbb{E}_S[x]$.

27.2 Rigorous Derivation

The book provides a more complex proof involving lemmas about recursive gradients (Lemma 7.2), the gradient as a sum (Lemma 7.3), and the stationary distribution (Lemma 7.4) to arrive at the full theorem (Theorem 7.5), which includes the discount factor term. This derivation shows that the $\mathbb{E}_S[x]$ term effectively cancels out over the long run under the stationary distribution assumption.

27.3 Stochastic Approximation

The policy gradient is an expectation, which is hard to compute. We can approximate it using Monte Carlo sampling.

1. Observe a state s (sampled from the environment, approximating $S \sim d^\pi$).

2. Sample an action $a \sim \pi(\cdot|s;\theta)$.

3. The term $g(s,a;\theta) = Q_\pi(s,a) \cdot \nabla_\theta \ln \pi(a|s;\theta)$ is an *unbiased* stochastic estimate of the (scaled) policy gradient: $\mathbb{E}_{S,A}[g(S,A;\theta)] = \nabla_\theta J(\theta)$ (ignoring the scaling factor).

4. We can use this estimate in stochastic gradient ascent: $\theta \leftarrow \theta + \beta \cdot g(s, a; \theta)$.

5. **Challenge:** We still don't know $Q_\pi(s, a)$.

28 REINFORCE Algorithm

REINFORCE (also known as Monte Carlo Policy Gradient) addresses the unknown $Q_\pi(s, a)$ by replacing it with a Monte Carlo sample: the observed return u_t.

28.1 Approximation

Since $Q_\pi(s_t, a_t) = \mathbb{E}[U_t | S_t = s_t, A_t = a_t]$, we can use the actual observed return $u_t = \sum_{k=t}^{n} \gamma^{k-t} r_k$ from a completed episode as an unbiased (but high-variance) sample estimate of $Q_\pi(s_t, a_t)$.

28.2 REINFORCE Gradient Estimate

The stochastic gradient estimate becomes:

$$\tilde{g}(s_t, a_t; \theta) = u_t \cdot \nabla_\theta \ln \pi(a_t | s_t; \theta)$$

28.3 Update Rule

Perform stochastic gradient ascent using this estimate. Typically, updates are performed after collecting a full episode:

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^{n} \gamma^{t-1} \cdot u_t \cdot \nabla_\theta \ln \pi(a_t | s_t; \theta_{\text{now}})$$

- The factor γ^{t-1} appears in the rigorous derivation to properly account for discounting and the effect of θ on the state distribution.

28.4 Training Loop

1. Initialize policy network parameters θ.

2. Loop:

   (a) Generate one full episode using the current policy $\pi(\cdot | s; \theta)$: $s_1, a_1, r_1, \ldots, s_n, a_n, r_n$.

   (b) For each step $t = 1, \ldots, n$: Calculate the return $u_t = \sum_{k=t}^{n} \gamma^{k-t} r_k$.

   (c) Compute the policy gradient estimate for the episode: $G = \sum_{t=1}^{n} \gamma^{t-1} u_t \nabla_\theta \ln \pi(a_t | s_t; \theta)$.

   (d) Update policy parameters: $\theta \leftarrow \theta + \beta \cdot G$.

28.5 Properties

REINFORCE is an **on-policy** algorithm because the returns u_t depend on the actions taken by the current policy $\pi(\cdot|s;\theta)$. Data collected under old policies cannot be reused, so experience replay is not directly applicable. It suffers from high variance due to using the full Monte Carlo return.

29 Actor-Critic

Actor-Critic methods use a separate function approximator (the Critic) to estimate the action-value function $Q_\pi(s,a)$ or state-value function $V_\pi(s)$, instead of using the Monte Carlo return u_t. The policy network $\pi(a|s;\theta)$ is the Actor.

29.1 Value Network (Critic)

Use a neural network $q(s,a;w)$ to approximate $Q_\pi(s,a)$ (like DQN but approximating Q_π). w are the critic's parameters.

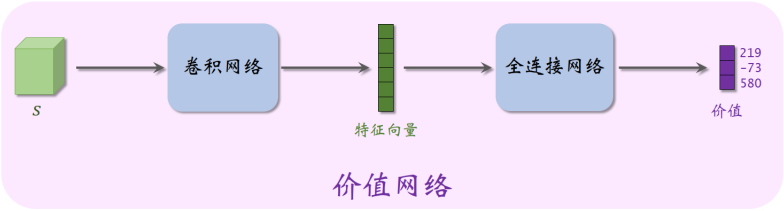


Fig. 41: value network

29.2 Actor-Critic Interaction

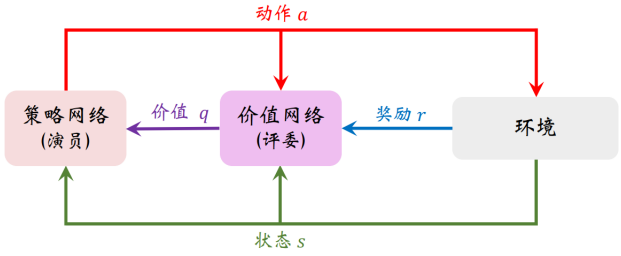


Fig. 42: actor critic network

- Actor (π) chooses action a_t based on state s_t.
- Critic (q) evaluates the chosen action by providing an estimate $q(s_t,a_t;w)$.
- Actor updates its parameters θ based on the Critic's evaluation.

- Critic updates its parameters w based on the observed reward and subsequent states/actions (using TD learning).

29.3 Actor Update (Policy Improvement)

Use the Critic's estimate $q(s_t, a_t; w)$ in place of the true $Q_\pi(s_t, a_t)$ in the policy gradient estimate:

$$\hat{g}(s_t, a_t; \theta) = q(s_t, a_t; w) \cdot \nabla_\theta \ln \pi(a_t|s_t; \theta)$$

Update the Actor parameters:

$$\theta \leftarrow \theta + \beta \cdot \hat{g}(s_t, a_t; \theta)$$

This update encourages actions that the Critic estimates to have higher values.

29.4 Critic Update (Policy Evaluation)

Use a TD algorithm like SARSA to train the value network $q(s, a; w)$ to approximate $Q_\pi(s, a)$. Given transition $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ (where $a_{t+1} \sim \pi(\cdot|s_{t+1}; \theta)$):

- TD Target: $\hat{y}_t = r_t + \gamma q(s_{t+1}, a_{t+1}; w)$
- TD Error: $\delta_t = q(s_t, a_t; w) - \hat{y}_t$
- Critic Update: $w \leftarrow w - \alpha \cdot \delta_t \cdot \nabla_w q(s_t, a_t; w)$

29.5 Training Loop (Basic Actor-Critic)

In each step t:

1. Observe s_t. Sample action $a_t \sim \pi(\cdot|s_t; \theta_{\text{now}})$.
2. Execute a_t, observe r_t, s_{t+1}.
3. Sample next action $a_{t+1} \sim \pi(\cdot|s_{t+1}; \theta_{\text{now}})$.
4. Get Critic's estimates: $\hat{q}_t = q(s_t, a_t; w_{\text{now}})$, $\hat{q}_{t+1} = q(s_{t+1}, a_{t+1}; w_{\text{now}})$.
5. Compute TD target $\hat{y}_t = r_t + \gamma \hat{q}_{t+1}$ and TD error $\delta_t = \hat{q}_t - \hat{y}_t$.
6. Update Critic: $w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \delta_t \nabla_w q(s_t, a_t; w_{\text{now}})$.
7. Update Actor: $\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \hat{q}_t \nabla_\theta \ln \pi(a_t|s_t; \theta_{\text{now}})$.

29.6 Using Target Networks

Similar to DQN, target networks can be used for the Critic $q(s, a; w^-)$ to stabilize training. The TD target becomes $\hat{y}_t = r_t + \gamma q(s_{t+1}, a_{t+1}; w^-)$.

Chapter 8: Policy Gradient Methods with Baseline

This chapter enhances the policy gradient methods by incorporating a baseline, which significantly improves their practical performance by reducing variance. This leads to variants like REINFORCE with Baseline and Advantage Actor-Critic (A2C).

30 Policy Gradient with Baseline

This section introduces the concept of a baseline and proves that subtracting it from the action-value function does not change the expected policy gradient.

30.1 Recap Policy Gradient

The objective function is $J(\theta) = \mathbb{E}_S[V_\pi(S)]$, and the policy gradient theorem states:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S\sim d^\pi, A\sim\pi(\cdot|S;\theta)}[Q_\pi(S,A) \cdot \nabla_\theta \ln \pi(A|S;\theta)]$$

30.2 Introducing the Baseline $b(S)$

A baseline b is a function that depends on the state S but *not* on the action A. A common choice is the state-value function $b(S) = V_\pi(S)$.

[Policy Gradient Theorem with Baseline] Subtracting a baseline $b(S)$ from $Q_\pi(S,A)$ does not change the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S\sim d^\pi, A\sim\pi(\cdot|S;\theta)}[(Q_\pi(S,A) - b(S)) \cdot \nabla_\theta \ln \pi(A|S;\theta)]$$

- **Proof Idea (detailed in Section 8.4):** This holds because the expected value of the term involving only the baseline is zero:

$$\mathbb{E}_{S\sim d^\pi, A\sim\pi(\cdot|S;\theta)}[b(S) \cdot \nabla_\theta \ln \pi(A|S;\theta)] = 0$$

30.3 Variance Reduction

While the expected gradient remains unchanged, the *variance* of the stochastic gradient estimate $\hat{g}(s,a;\theta) = (Q_\pi(s,a) - b(s))\nabla_\theta \ln \pi(a|s;\theta)$ is affected by the choice of $b(s)$. Choosing $b(s)$ close to the average value of $Q_\pi(s,A)$ (averaged over $A \sim \pi$) reduces the variance. The optimal choice for minimizing variance is related to $V_\pi(s)$, making $V_\pi(s)$ a standard and effective baseline.

30.4 Intuitive Explanation

Policy gradient updates increase the probability of actions with high $Q_\pi(s,a)$ and decrease it for actions with low $Q_\pi(s,a)$. Subtracting a baseline $b(s)$ shifts all Q-values for state s by the same amount. This doesn't change the *relative* desirability of actions but centers the scaling factor $(Q_\pi - b)$ around

zero. Actions better than average ($Q_\pi > b$) get positive scaling, increasing their probability, while actions worse than average ($Q_\pi < b$) get negative scaling, decreasing their probability. This centering reduces the magnitude of the updates, thus lowering variance.

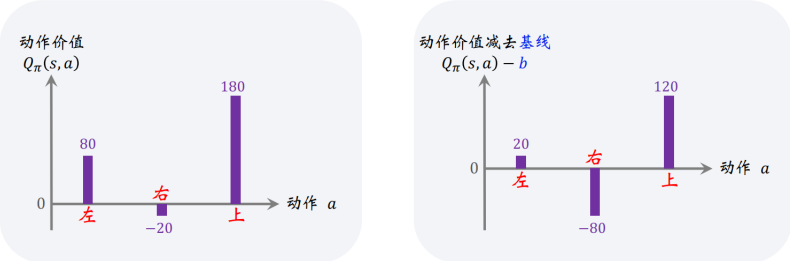


Fig. 43: baseline

31 REINFORCE with Baseline

31.1 Baseline Choice

Use the state-value function $V_\pi(s)$ as the baseline $b(s)$.

31.2 Approximations

1. Replace the true action value $Q_\pi(s,a)$ with the sampled Monte Carlo return u.
2. Approximate the baseline $V_\pi(s)$ using a neural network, the *value network*, denoted $v(s;w)$.

31.3 Stochastic Gradient Estimate

The policy gradient $\nabla_\theta J(\theta)$ is approximated by:

$$\tilde{g}(s,a;\theta) = (u - v(s;w)) \cdot \nabla_\theta \ln \pi(a|s;\theta)$$

The term $(u - v(s;w))$ is an estimate of the *advantage* $Q_\pi(s,a) - V_\pi(s)$.

31.4 Network Structures

- **Policy Network** $\pi(a|s;\theta)$**:** Same as before, outputs action probabilities.
- **Value Network** $v(s;w)$**:** Takes state s as input and outputs a single scalar value (the estimated state value $V_\pi(s)$). Often shares lower layers (e.g., CNN body) with the policy network.
- **Note:** This is *not* Actor-Critic because the value network $v(s;w)$ only serves as a baseline to reduce variance; it doesn't provide the primary learning signal for the policy network (which still comes from the Monte Carlo return u).

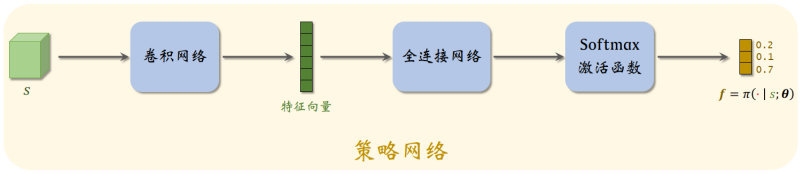


Fig. 44: policy network

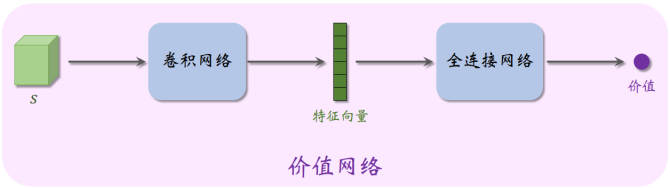


Fig. 45: value network

31.5 Training Algorithm

- **Policy Network Update:** Use the stochastic gradient estimate $\tilde{g}(s_t, a_t; \theta)$:

$$\theta \leftarrow \theta + \beta \cdot (u_t - v(s_t; w)) \cdot \nabla_\theta \ln \pi(a_t | s_t; \theta)$$

  Usually performed after an episode using the sum of gradients over the trajectory.

- **Value Network Update:** Train $v(s; w)$ to predict the Monte Carlo return u. This is a standard supervised regression problem. Minimize the mean squared error loss:

$$L(w) = \frac{1}{2n} \sum_{t=1}^{n} (v(s_t; w) - u_t)^2$$

  Update using stochastic gradient descent:

$$w \leftarrow w - \alpha \cdot (v(s_t; w) - u_t) \cdot \nabla_w v(s_t; w)$$

  (Using $\delta_t = v(s_t; w) - u_t$ as the error term for both updates simplifies implementation).

31.6 Training Loop (Full Episode Update)

1. Initialize $\theta_{\text{now}}, w_{\text{now}}$.
2. Loop:
   (a) Generate one episode using policy $\pi(\cdot|s; \theta_{\text{now}})$: $(s_1, a_1, r_1, \ldots, s_n, a_n, r_n)$.
   (b) For $t = 1, \ldots, n$: Calculate return $u_t = \sum_{k=t}^{n} \gamma^{k-t} r_k$.
   (c) For $t = 1, \ldots, n$: Predict baseline $\hat{v}_t = v(s_t; w_{\text{now}})$.
   (d) For $t = 1, \ldots, n$: Calculate error $\delta_t = \hat{v}_t - u_t$.

(e) Compute value network gradient sum: $G_w = \sum_{t=1}^{n} \delta_t \nabla_w v(s_t; w_{\text{now}})$.

(f) Update value network: $w_{\text{new}} \leftarrow w_{\text{now}} - \alpha G_w$.

(g) Compute policy network gradient sum: $G_\theta = \sum_{t=1}^{n} \gamma^{t-1}(-\delta_t)\nabla_\theta \ln \pi(a_t|s_t; \theta_{\text{now}})$.

(h) Update policy network: $\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta G_\theta$.

32 Advantage Actor-Critic (A2C)

This section applies the baseline concept within the Actor-Critic framework, using the TD error as an estimate of the advantage function.

32.1 Advantage Function

Recall the advantage function $A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$. The policy gradient theorem with baseline $V_\pi(s)$ is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S,A}[A_\pi(S, A) \cdot \nabla_\theta \ln \pi(A|S; \theta)]$$

32.2 A2C Idea

Use a value network $v(s; w)$ to approximate $V_\pi(s)$ (the Critic) and a policy network $\pi(a|s; \theta)$ (the Actor). Estimate the advantage $A_\pi(s_t, a_t)$ using TD error.

32.3 Advantage Estimation using TD Error

- We know $Q_\pi(s_t, a_t) = \mathbb{E}[R_t + \gamma V_\pi(S_{t+1})|S_t = s_t, A_t = a_t]$.
- So, $A_\pi(s_t, a_t) = \mathbb{E}[R_t + \gamma V_\pi(S_{t+1})|S_t = s_t, A_t = a_t] - V_\pi(s_t)$.
- Use a single sample (s_t, a_t, r_t, s_{t+1}) and the value network $v(s; w)$ to approximate the expectation and the true values:
$$A_\pi(s_t, a_t) \approx [r_t + \gamma v(s_{t+1}; w)] - v(s_t; w)$$
- Define the TD target $\hat{y}_t = r_t + \gamma v(s_{t+1}; w)$.
- Define the TD error $\delta_t = v(s_t; w) - \hat{y}_t$.
- Then the advantage estimate is $A_\pi(s_t, a_t) \approx \hat{y}_t - v(s_t; w) = -\delta_t$.

32.4 Actor Update (Policy Improvement)

Use the estimated advantage $(-\delta_t)$ in the policy gradient update:

$$\tilde{g}(s_t, a_t; \theta) = -\delta_t \cdot \nabla_\theta \ln \pi(a_t|s_t; \theta)$$

$$\theta \leftarrow \theta - \beta \cdot \delta_t \cdot \nabla_\theta \ln \pi(a_t|s_t; \theta)$$

32.5 Critic Update (Policy Evaluation)

Update the value network $v(s; w)$ using the TD error δ_t to minimize the difference between $v(s_t; w)$ and the TD target $\hat{y}_t$:

$$w \leftarrow w - \alpha \cdot \delta_t \cdot \nabla_w v(s_t; w)$$

32.6 Interaction

Actor π takes action a_t. Critic v calculates TD error δ_t based on r_t and its estimates $v(s_t; w), v(s_{t+1}; w)$. Actor updates θ using δ_t. Critic updates w using δ_t. The sign of δ_t indicates if the outcome was better ($\delta_t < 0$) or worse ($\delta_t > 0$) than expected at state s_t.

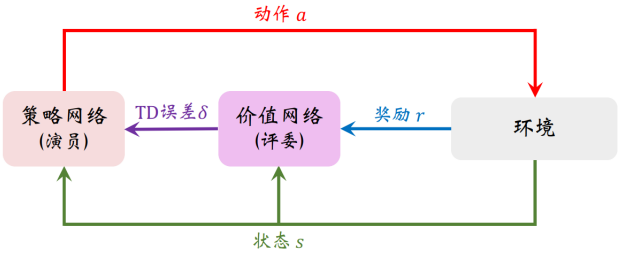


Fig. 46: A2C

32.7 Training Loop (A2C)

In each step t:

1. Observe s_t. Sample action $a_t \sim \pi(\cdot|s_t; \theta_{\text{now}})$.
2. Execute a_t, observe r_t, s_{t+1}.
3. Get Critic's estimates: $\hat{v}_t = v(s_t; w_{\text{now}})$, $\hat{v}_{t+1} = v(s_{t+1}; w_{\text{now}})$ (or use target network $v(s_{t+1}; w^-)$).
4. Compute TD target $\hat{y}_t = r_t + \gamma \hat{v}_{t+1}$ and TD error $\delta_t = \hat{v}_t - \hat{y}_t$.
5. Update Critic: $w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \delta_t \nabla_w v(s_t; w_{\text{now}})$.
6. Update Actor: $\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \delta_t \nabla_\theta \ln \pi(a_t|s_t; \theta_{\text{now}})$.

32.8 Properties

A2C is also **on-policy**. Using a target network $v(s; w^-)$ for the Critic update (computing $\hat{y}_t = r_t + \gamma v(s_{t+1}; w^-)$) is recommended for stability.

33 Proof of Policy Gradient Theorem with Baseline

This section formally proves Theorem[baseline].

Let $b(S)$ be any function that does not depend on action A. Then for any state s:

$$\mathbb{E}_{A\sim\pi(\cdot|s;\theta)}\left[b(s)\cdot\frac{\partial\ln\pi(A|s;\theta)}{\partial\theta}\right]=0$$

Since $b(s)$ does not depend on A, it can be taken out of the expectation:

$$\begin{aligned}\mathbb{E}_{A\sim\pi(\cdot|s;\theta)}\left[b(s)\cdot\frac{\partial\ln\pi(A|s;\theta)}{\partial\theta}\right]&=b(s)\cdot\mathbb{E}_{A\sim\pi(\cdot|s;\theta)}\left[\frac{\partial\ln\pi(A|s;\theta)}{\partial\theta}\right]\\&=b(s)\cdot\sum_{a\in\mathcal{A}}\pi(a|s;\theta)\frac{1}{\pi(a|s;\theta)}\frac{\partial\pi(a|s;\theta)}{\partial\theta}\\&=b(s)\cdot\sum_{a\in\mathcal{A}}\frac{\partial\pi(a|s;\theta)}{\partial\theta}\\&=b(s)\cdot\frac{\partial}{\partial\theta}\sum_{a\in\mathcal{A}}\pi(a|s;\theta)\\&=b(s)\cdot\frac{\partial}{\partial\theta}(1)\\&=b(s)\cdot0=0\end{aligned}$$

[Proof of Theorem baseline] Start with the expectation term involving the baseline:

$$\mathbb{E}_{S,A}[b(S)\cdot\nabla_\theta\ln\pi(A|S;\theta)]$$

Using the law of iterated expectations:

$$=\mathbb{E}_S\left[\mathbb{E}_{A\sim\pi(\cdot|S;\theta)}[b(S)\cdot\nabla_\theta\ln\pi(A|S;\theta)|S]\right]$$

By Lemma, the inner expectation is 0 for any $S=s$.

$$=\mathbb{E}_S[0]=0$$

Therefore,

$$\begin{aligned}\nabla_\theta J(\theta)&=\mathbb{E}_{S,A}[Q_\pi(S,A)\cdot\nabla_\theta\ln\pi(A|S;\theta)]\\&=\mathbb{E}_{S,A}[Q_\pi(S,A)\cdot\nabla_\theta\ln\pi(A|S;\theta)]-0\\&=\mathbb{E}_{S,A}[Q_\pi(S,A)\cdot\nabla_\theta\ln\pi(A|S;\theta)]-\mathbb{E}_{S,A}[b(S)\cdot\nabla_\theta\ln\pi(A|S;\theta)]\\&=\mathbb{E}_{S,A}[(Q_\pi(S,A)-b(S))\cdot\nabla_\theta\ln\pi(A|S;\theta)]\end{aligned}$$

Chapter 9: Advanced Techniques for Policy Learning

This chapter introduces advanced techniques to improve policy learning beyond the basic methods, focusing on Trust Region Policy Optimization (TRPO) and Entropy Regularization.

34 Trust Region Policy Optimization (TRPO)

TRPO is a policy learning algorithm designed to provide more stable and reliable improvements compared to standard policy gradient methods. It achieves this by restricting the amount the policy can change in each update step, keeping the new policy within a "trust region" around the old policy.

34.1 Trust Region Methods (General Optimization Concept)

Trust region methods are a class of numerical optimization algorithms for problems like $\max_\theta J(\theta)$. They work iteratively:

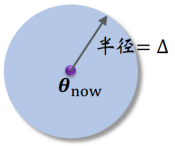


Fig. 47: region

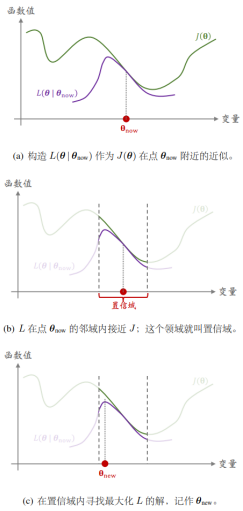
1. **Approximation:** Given the current parameters θ_{now}, construct a simpler *surrogate objective function* $L(\theta|\theta_{\text{now}})$ that approximates the true objective $J(\theta)$ within a neighborhood $\mathcal{N}(\theta_{\text{now}})$ around θ_{now}. This neighborhood is called the *trust region*. An example trust region is defined by Euclidean distance:
$$\mathcal{N}(\theta_{\text{now}}) = \{\theta \mid ||\theta - \theta_{\text{now}}||_2 \leq \Delta\}$$
where Δ is the trust region radius. The surrogate L is typically easier to optimize than J (e.g., a local quadratic approximation or a Monte Carlo estimate).

2. **Maximization:** Find the parameters θ_{new} that maximize the surrogate objective L *within* the trust region:
$$\theta_{\text{new}} = \arg \max_{\theta \in \mathcal{N}(\theta_{\text{now}})} L(\theta|\theta_{\text{now}})$$
This is a constrained optimization problem.

The process repeats until convergence. Different trust region methods vary in how they construct L and $\mathcal{N}$, and how they solve the constrained maximization.

34.2 Policy Learning Recap and Objective Transformation

- **Recap:** Policy network $\pi(a|s;\theta)$, action-value $Q_\pi(s,a)$, state-value $V_\pi(s) = \mathbb{E}_{A\sim\pi(\cdot|s;\theta)}[Q_\pi(s,A)]$, objective $J(\theta) = \mathbb{E}_S[V_\pi(S)]$.



(a) 构造 $L(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{now})$ 作为 $J(\boldsymbol{\theta})$ 在点 $\boldsymbol{\theta}_{now}$ 附近的近似。

(b) L 在点 $\boldsymbol{\theta}_{now}$ 的邻域内接近 J：这个领域就叫置信域。

(c) 在置信域内寻找最大化 L 的解，记作 $\boldsymbol{\theta}_{new}$。

Fig. 48: example

- **Objective Transformation:** Using importance sampling, the objective function $J(\theta)$ for a new policy $\pi(\cdot|s;\theta)$ can be expressed in terms of an expectation under an old policy $\pi_{\text{now}} = \pi(\cdot|s;\theta_{\text{now}})$:

$$J(\theta) = \mathbb{E}_{S\sim d^{\pi_{\text{now}}}, A\sim\pi_{\text{now}}(\cdot|S)}\left[\frac{\pi(A|S;\theta)}{\pi(A|S;\theta_{\text{now}})}Q_\pi(S,A)\right]$$

  Here, the expectation is over states visited and actions taken while following the *old* policy π_{now}. Note that Q_π still depends on the *new* policy parameter θ.

34.3 TRPO Mathematical Derivation

TRPO applies the trust region framework to the policy optimization objective $J(\theta)$.

1. **Step 1: Approximation - Constructing Surrogate L**
   - Start with the transformed objective $J(\theta)$ from Theorem 9.1.
   - Cannot compute the expectation directly. Use Monte Carlo approximation by running the *old* policy π_{now} for one (or more) full episodes to get trajectories $(s_1, a_1, r_1, \ldots, s_n, a_n, r_n)$.
   - The term inside the expectation involves $Q_\pi(S,A)$, which depends on the new θ. Make two approximations:
     - Replace $Q_\pi(s_t,a_t)$ with $Q_{\pi_{\text{now}}}(s_t,a_t)$. This is valid only when θ is close to θ_{now}.
     - Approximate $Q_{\pi_{\text{now}}}(s_t,a_t)$ using the sampled Monte Carlo return $u_t = \sum_{k=t}^{n}\gamma^{k-t}r_k$ (calculated from the trajectory generated by π_{now}).
   - Surrogate objective becomes:

$$\tilde{L}(\theta|\theta_{\text{now}}) = \frac{1}{n}\sum_{t=1}^{n}\frac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{\text{now}})}u_t$$

This $\tilde{L}$ approximates $J(\theta)$ well only when θ is near θ_{now}.

2. **Step 2: Maximization - Constrained Update**

   - TRPO maximizes the surrogate $\tilde{L}(\theta|\theta_{\text{now}})$ subject to a constraint ensuring θ stays close to θ_{now}. Uses a constraint based on the average KL divergence:

$$\max_{\theta} \tilde{L}(\theta|\theta_{\text{now}})$$

$$\text{subject to} \quad \bar{D}_{KL}(\theta_{\text{now}}||\theta) \leq \Delta$$

     where $\bar{D}_{KL}(\theta_{\text{now}}||\theta) = \frac{1}{n}\sum_{t=1}^{n}[\pi(\cdot|s_t;\theta_{\text{now}})||\pi(\cdot|s_t;\theta)]$ is the empirical average KL divergence, and Δ is the trust region size.

   - Solving this constrained optimization problem yields θ_{new}. This step is computationally involved.

34.4 TRPO Training Flow

TRPO iteratively performs the approximation and maximization steps:

1. Given current parameters θ_{now}.
2. **Approximate:**
   (a) Generate trajectory(s) using $\pi(\cdot|s;\theta_{\text{now}})$.
   (b) Compute returns u_t for all steps t.
   (c) Form the surrogate objective $\tilde{L}(\theta|\theta_{\text{now}}) = \frac{1}{n}\sum_{t=1}^{n}\frac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{\text{now}})}u_t$.
3. **Maximize:** Solve the constrained problem to find θ_{new}:

$$\theta_{\text{new}} = \arg\max_{\theta} \tilde{L}(\theta|\theta_{\text{now}}) \quad \text{s.t.} \quad \bar{D}_{KL}(\theta_{\text{now}}||\theta) \leq \Delta$$

4. Update $\theta_{\text{now}} \leftarrow \theta_{\text{new}}$ and repeat.

TRPO requires tuning Δ and parameters for the constrained optimization solver. It is generally more stable and sample-efficient than standard policy gradient methods but significantly more complex to implement.

35 Entropy Regularization

This technique encourages exploration by adding an entropy bonus to the objective function.

35.1 Motivation

Standard policy gradient methods can sometimes lead to policies that become too deterministic (concentrated on a single action) too quickly, preventing exploration. Entropy measures the randomness of a probability distribution. Higher entropy corresponds to more uniform distributions.

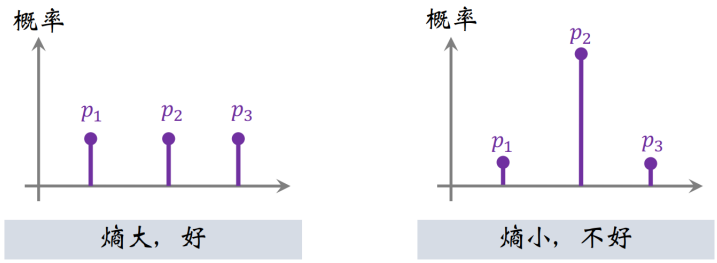


Fig. 49: entropy

35.2 Policy Entropy

The entropy of the action distribution $\pi(\cdot|s;\theta)$ for a given state s is:

$$H(s;\theta) = -\sum_{a\in\mathcal{A}} \pi(a|s;\theta)\ln\pi(a|s;\theta)$$

35.3 Regularized Objective

Modify the standard objective $J(\theta)$ to include the expected entropy over states:

$$\max_{\theta} J_{\text{reg}}(\theta) = \max_{\theta}\left[J(\theta) + \lambda\mathbb{E}_S[H(S;\theta)]\right]$$

Here, $\lambda \geq 0$ is a hyperparameter controlling the strength of the entropy bonus. Maximizing this encourages policies that achieve high returns while also maintaining some level of randomness (exploration).

35.4 Optimization (Policy Gradient Approach)

The gradient of the regularized objective is the sum of the original policy gradient and the gradient of the entropy term.

[Policy Gradient with Entropy Regularization] The gradient of the regularized objective $J_{\text{reg}}(\theta) = J(\theta) + \lambda\mathbb{E}_S[H(S;\theta)]$ is:

$$\nabla_\theta J_{\text{reg}}(\theta) = \mathbb{E}_{S,A}\left[(Q_\pi(S,A) - \lambda\ln\pi(A|S;\theta) - \lambda)\nabla_\theta\ln\pi(A|S;\theta)\right]$$

(Assuming $S \sim d^\pi, A \sim \pi(\cdot|S;\theta)$, and ignoring the overall scaling factor). [Proof Sketch] The gradient of the entropy term $\mathbb{E}_S[H(S;\theta)]$ is derived as:

$$\begin{aligned}
\nabla_\theta H(s;\theta) &= -\nabla_\theta \sum_a \pi(a|s;\theta) \ln \pi(a|s;\theta) \\
&= -\sum_a \left[(\nabla_\theta \pi(a|s;\theta)) \ln \pi(a|s;\theta) + \pi(a|s;\theta) \frac{\nabla_\theta \pi(a|s;\theta)}{\pi(a|s;\theta)} \right] \\
&= -\sum_a [\pi(a|s;\theta) \nabla_\theta \ln \pi(a|s;\theta) \ln \pi(a|s;\theta) + \nabla_\theta \pi(a|s;\theta)] \\
&= -\sum_a \pi(a|s;\theta)[\ln \pi(a|s;\theta) + 1] \nabla_\theta \ln \pi(a|s;\theta) \\
&\quad - \underbrace{\sum_a \nabla_\theta \pi(a|s;\theta)}_{=\nabla_\theta(1)=0} \\
&= -\mathbb{E}_{A\sim\pi(\cdot|s;\theta)} \left[(\ln \pi(A|s;\theta) + 1) \nabla_\theta \ln \pi(A|s;\theta) \right]
\end{aligned}$$

Adding $\lambda \mathbb{E}_S[\nabla_\theta H(S;\theta)]$ to the standard policy gradient $\mathbb{E}_{S,A}[Q_\pi(S,A) \nabla_\theta \ln \pi(A|S;\theta)]$ and combining terms inside the expectation gives the result.

35.5 Stochastic Gradient

A stochastic estimate of the gradient for the regularized objective is:

$$\tilde{g}_{\text{reg}}(s,a;\theta) = (Q_\pi(s,a) - \lambda \ln \pi(a|s;\theta) - \lambda) \nabla_\theta \ln \pi(a|s;\theta)$$

The update rule becomes $\theta \leftarrow \theta + \beta \tilde{g}_{\text{reg}}(s,a;\theta)$. This can be combined with REINFORCE or Actor-Critic methods by substituting the appropriate estimate for $Q_\pi(s,a)$.

Chapter 10: Continuous Control

Previous chapters focused on *discrete control*, where the action space $\mathcal{A}$ is a finite set (e.g., $\mathcal{A} = \{\text{left, right, up}\}$). This chapter addresses *continuous control*, where the action space is a continuous set (e.g., steering angle $\mathcal{A} = [-40°, 40°]$, or multi-dimensional joint angles $\mathcal{A} = [0, 360] \times [0, 180]$).

36 Discrete vs. Continuous Control

- **Challenge:** Methods like DQN rely on finding $\max_{a\in\mathcal{A}} Q(s,a)$, which is straightforward for discrete $\mathcal{A}$ but becomes a complex optimization problem for continuous $\mathcal{A}$.
- **Discretization:** One approach is to discretize the continuous action space into a grid. Then, standard discrete control methods can be applied.

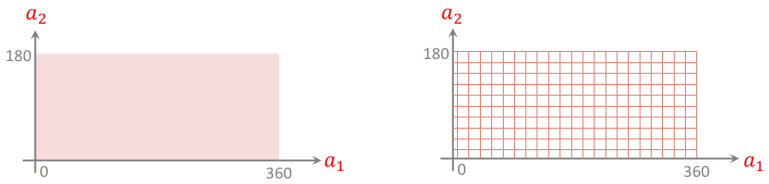


Fig. 50:

- **Limitation:** Discretization suffers from the *curse of dimensionality*. If the action space has d dimensions and each dimension is discretized into k bins, the total number of discrete actions becomes k^d, which grows exponentially. This makes learning infeasible for moderate to high dimensions d. Therefore, methods specifically designed for continuous control are needed.

37 Deterministic Policy Gradient (DPG)

DPG is a widely used actor-critic algorithm for continuous control problems.

37.1 Policy Network and Value Network

DPG uses two main networks:

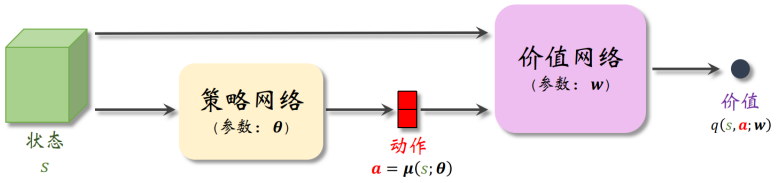


Fig. 51: DPG

1. **Deterministic Policy Network (Actor)** $\mu(s;\theta)$**:**
   - Takes state s as input and outputs a specific action vector $a \in \mathbb{R}^d$.

$$a = \mu(s;\theta)$$

   - Contrast with stochastic policy networks $\pi(a|s;\theta)$ from previous chapters, which output a probability distribution. A deterministic policy can be seen as a special case of a stochastic policy (e.g., a Gaussian with zero variance).
   - Network Structure: Typically uses CNNs for image inputs or MLPs for vector inputs. The output layer has dimension d (the action dimension) and often uses activations like 'tanh' to bound the output to a specific range (e.g., $[-1, 1]$), which is then scaled to the true action range.

2. **Value Network (Critic)** $q(s, a; w)$**:**

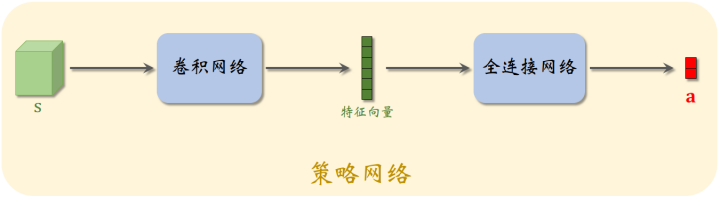


Fig. 52: policy network

- Approximates the action-value function $Q_\pi(s, a)$ for the policy π implicitly defined by the actor μ.
- Takes both state s and action vector a as input and outputs a single scalar Q-value, $q(s, a; w)$.
- Network Structure: Often processes state s and action a separately initially (e.g., s through CNNs/MLPs, a through MLPs) and then combines the features to predict the Q-value.

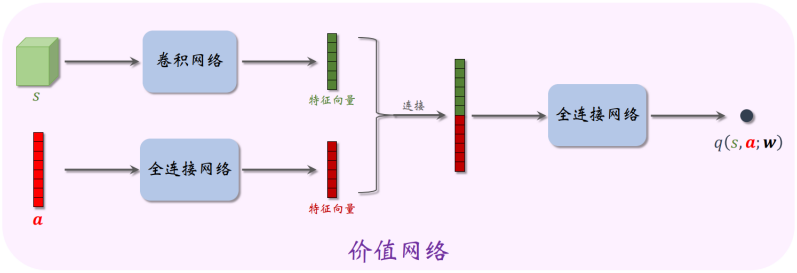


Fig. 53: value network

37.2 Algorithm Derivation

DPG is an **off-policy** algorithm, allowing the use of experience replay.

- **Data Collection:** Use a behavior policy (e.g., adding noise to the actor's output: $a_t = \mu(s_t; \theta) + \epsilon_t$, where ϵ_t is noise like Gaussian or Ornstein-Uhlenbeck) to interact with the environment. Store transitions (s_t, a_t, r_t, s_{t+1}) in a replay buffer.

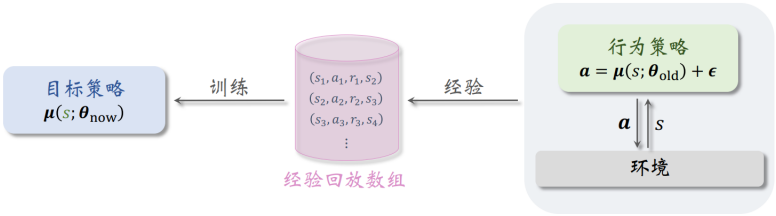


Fig. 54: DPG off policy

- **Critic Update (Value Network Training):** Learn $q(s, a; w)$ to approximate $Q_\mu(s, a)$ using TD learning.

1. Sample a mini-batch of B transitions (s_j, a_j, r_j, s_{j+1}) from the replay buffer.
2. Compute the target actions using the *target* policy network $\mu(s; \theta^-)$: $\hat{a}_{j+1} = \mu(s_{j+1}; \theta^-)$. (Target networks q^- and μ^- with parameters w^-, θ^- that lag behind w, θ are used for stability, similar to DQN).
3. Compute the target Q-value using the *target* value network $q(s, a; w^-)$: $\hat{q}_{j+1} = q(s_{j+1}, \hat{a}_{j+1}; w^-)$.
4. Compute the TD target: $\hat{y}_j = r_j + \gamma \hat{q}_{j+1}$.
5. Compute the TD error: $\delta_j = q(s_j, a_j; w) - \hat{y}_j$.
6. Update the critic parameters w by minimizing the loss $L(w) = \frac{1}{B} \sum_j \frac{1}{2} \delta_j^2$, typically via gradient descent:
$$w \leftarrow w - \alpha_w \cdot \frac{1}{B} \sum_j \delta_j \cdot \nabla_w q(s_j, a_j; w)$$

- **Actor Update (Policy Network Training):** Learn $\mu(s; \theta)$ to output actions that maximize the estimated Q-value from the critic.
  1. The objective is to maximize $J(\theta) = \mathbb{E}_{S \sim \rho^\beta}[q(S, \mu(S; \theta); w)]$ (expectation over state distribution ρ^β visited by behavior policy β).
  2. The Deterministic Policy Gradient states:
  $$\nabla_\theta J(\theta) = \mathbb{E}_{S \sim \rho^\beta}[\nabla_\theta \mu(S; \theta) \cdot \nabla_a q(S, a; w)|_{a=\mu(S;\theta)}]$$
  The gradient involves the gradient of the Q-value with respect to the *action* ($\nabla_a q$) multiplied by the gradient of the policy network output with respect to its *parameters* ($\nabla_\theta \mu$).
  3. Use the sampled states s_j from the mini-batch and the current actor network $\mu(s; \theta)$ to approximate the gradient:
  $$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_j \nabla_\theta \mu(s_j; \theta) \cdot \nabla_a q(s_j, a; w)|_{a=\mu(s_j;\theta)}$$
  4. Update the actor parameters θ using gradient ascent:
  $$\theta \leftarrow \theta + \alpha_\theta \cdot \frac{1}{B} \sum_j \nabla_\theta \mu(s_j; \theta) \cdot \nabla_a q(s_j, a; w)|_{a=\mu(s_j;\theta)}$$

- **Target Network Updates:** Update target networks w^- and θ^- slowly:
$$w^- \leftarrow \tau w + (1 - \tau) w^-$$
$$\theta^- \leftarrow \tau \theta + (1 - \tau) \theta^-$$
where $\tau \ll 1$.

37.3 Training Flow Note

The basic DPG algorithm described above often performs poorly in practice due to issues like overestimation.

38 Deeper Analysis of DPG

- **What does $q(s, a; w)$ approximate?** It approximates $Q_\mu(s, a)$, the value of the current deterministic policy μ. As training progresses and μ improves towards the optimal policy μ_*, q implicitly approaches Q_*.
- **DPG as Q_* Approximation:** DPG can be viewed as finding a pair (μ, q) such that $\mu(s; \theta) \approx \arg\max_a Q_*(s, a)$ and $q(s, \mu(s; \theta); w) \approx \max_a Q_*(s, a)$. The actor μ helps find the maximizing action for the implicit Q_* approximated by q.
- **Overestimation in DPG:** Similar to DQN, the basic DPG formulation suffers from Q-value overestimation. This arises from:
  1. **Function Approximation Error & Maximization:** The actor μ tries to exploit peaks in the learned q function. If q inaccurately has high values for some actions, the actor will learn to output those actions, leading to an inflated value in the critic's TD target calculation $q(s_{j+1}, \mu(s_{j+1}; \theta^-); w^-)$.
  2. **Bootstrapping:** The critic update uses its own (target network's) output, propagating any existing overestimation bias.

39 Twin Delayed Deep Deterministic Policy Gradient (TD3)

TD3 introduces several modifications to the DPG algorithm to mitigate overestimation and improve stability.

39.1 Solutions for Overestimation

- **Target Networks:** Use target networks $\mu(s; \theta^-)$ and $q(s, a; w^-)$ for computing TD target $\hat{y}_j = r_j + \gamma q(s_{j+1}, \mu(s_{j+1}; \theta^-); w^-)$. Helps but insufficient.

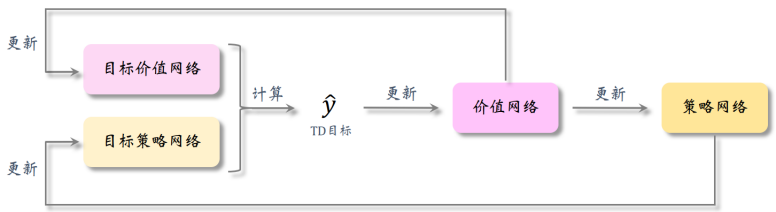


Fig. 55:

- **Clipped Double Q-Learning:**
  - Learn *two* independent Q-networks (Critics) $q(s,a;w_1)$ and $q(s,a;w_2)$, along with their respective target networks $q(s,a;w_1^-)$ and $q(s,a;w_2^-)$.
  - When computing the TD target $\hat{y}_j$, use the *minimum* of the two target Q-networks:

$$\hat{a}_{j+1}^- = \mu(s_{j+1};\theta^-)$$

$$\hat{y}_j = r_j + \gamma \min_{i=1,2} q(s_{j+1},\hat{a}_{j+1}^-;w_i^-)$$

  - Both critics q_1 and q_2 are updated using this same target $\hat{y}_j$. This helps reduce upward bias in the target value.

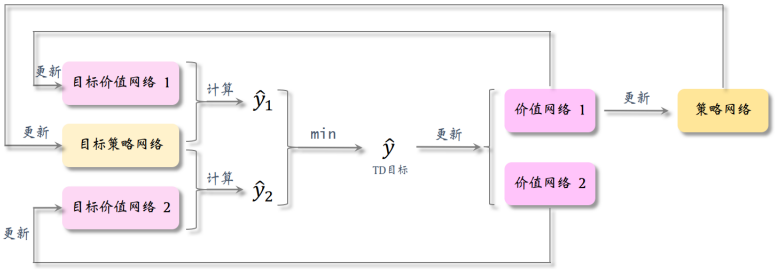


Fig. 56:

39.2 Other Improvements

- **Target Policy Smoothing Regularization:**
  - Add clipped noise to the action chosen by the *target* policy network when calculating the target Q-value. This smooths the value estimate with respect to actions.

$$\hat{a}_{j+1}^- = \text{clip}(\mu(s_{j+1};\theta^-) + \xi, a_{\text{low}}, a_{\text{high}})$$

$$\xi \sim \text{clipNormal}(0,\sigma^2,-c,c)$$

    where clipNormal is Gaussian noise clipped between $-c$ and c. c and σ are hyperparameters. The target Q-value becomes $\min_{i=1,2} q(s_{j+1},\hat{a}_{j+1}^-;w_i^-)$ using this smoothed action.

- **Delayed Policy and Target Updates:**
  - Update the policy network $\mu(s;\theta)$ and all target networks (q_1^-,q_2^-,μ^-) less frequently than the value networks (q_1,q_2). For instance, update the policy and targets only once for every k updates of the value networks (e.g., $k=2$). Allows value estimate to stabilize before updating policy.

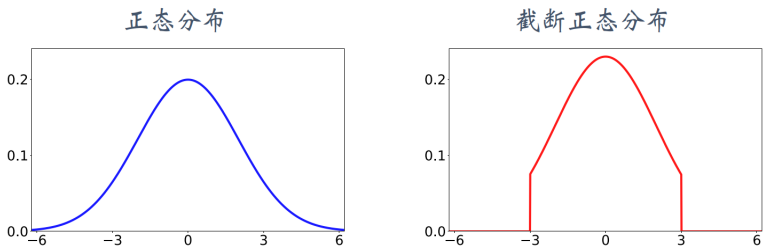


Fig. 57:

39.3 TD3 Training Flow

Combines Clipped Double Q-Learning, Target Policy Smoothing, and Delayed Policy Updates. It involves maintaining 6 networks $(q_1, q_2, \mu, q_1^-, q_2^-, \mu^-)$ and applying these techniques during the training loop based on samples from the replay buffer.

40 Stochastic Gaussian Policies

An alternative approach to continuous control using stochastic policies, specifically Gaussian distributions.

40.1 Basic Idea (d=1)

Model the policy as a Gaussian $\pi(a|s) = \mathcal{N}(a|\mu(s), \sigma^2(s))$. Use two networks to output the mean $\mu(s;\theta)$ and the log-variance $\rho(s;\theta) = \ln \sigma^2(s)$. The actual variance is $\sigma^2(s;\theta) = \exp(\rho(s;\theta))$. (Parameterizing log-variance ensures positivity of variance). Action is sampled: $a \sim \mathcal{N}(\mu(s;\theta), \exp(\rho(s;\theta)))$.

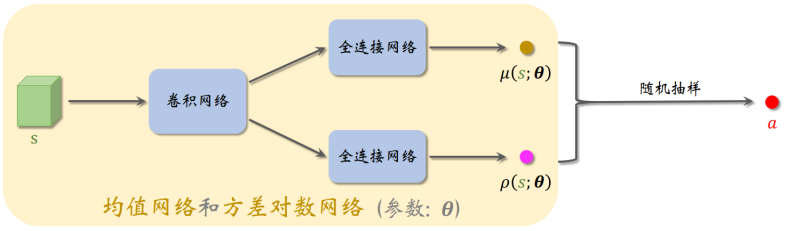


Fig. 58:

40.2 Multivariate Gaussian Policy (Diagonal Covariance)

For action $a \in \mathbb{R}^d$, use networks outputting $\mu(s;\theta) \in \mathbb{R}^d$ and $\rho(s;\theta) \in \mathbb{R}^d$. The policy is a product of independent Gaussians:

$$\pi(a|s;\theta) = \prod_{i=1}^{d} \mathcal{N}(a_i|\mu_i(s;\theta), \exp(\rho_i(s;\theta)))$$

40.3 Log-Policy / Auxiliary Network $f(s, a; \theta)$

Define $f(s, a; \theta)$ such that $f(s, a; \theta) = \ln \pi(a|s; \theta) + \text{Constant}$. For the diagonal Gaussian case:

$$f(s, a; \theta) = -\frac{1}{2} \sum_{i=1}^{d} \left(\rho_i(s; \theta) + \frac{(a_i - \mu_i(s; \theta))^2}{\exp(\rho_i(s; \theta))} \right)$$

The parameters θ are implicitly those of the μ and ρ networks.

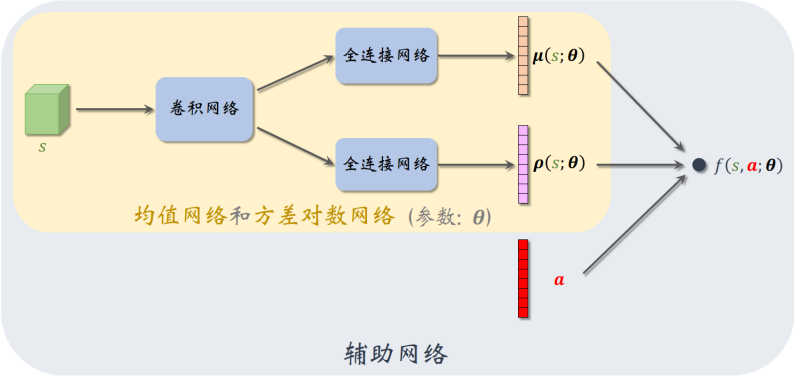


Fig. 59:

40.4 Policy Gradient

The policy gradient estimate is $g = Q_\pi(s, a) \cdot \nabla_\theta \ln \pi(a|s; \theta) = Q_\pi(s, a) \cdot \nabla_\theta f(s, a; \theta)$. Need to estimate Q_π.

40.5 Using REINFORCE

Approximate $Q_\pi(s_t, a_t)$ with Monte Carlo return u_t. Update θ using $\theta \leftarrow \theta + \beta \sum_t \gamma^{t-1} u_t \nabla_\theta f(s_t, a_t; \theta)$. On-policy. Baselines can be added.

40.6 Using Actor-Critic

Use a separate value network $q(s, a; w)$ to estimate Q_π. Actor update: $\theta \leftarrow \theta + \beta q(s_t, a_t; w) \nabla_\theta f(s_t, a_t; \theta)$. Critic update: Use SARSA or A2C principles (with appropriate target value calculations, e.g., $\hat{y}_t = r_t + \gamma q(s_{t+1}, a_{t+1}; w^-)$) to update w. Generally on-policy.

Chapter 11: Dealing with Partial Observation

Previous chapters assumed the agent can fully observe the environment's state s_t at each time step t. This chapter addresses the more realistic scenario where the agent only receives a partial observation

o_t of the true state s_t.

41 The Problem of Partial Observation

- **Definition:** In many real-world applications (e.g., robotics with limited sensors, games like Poker or StarCraft with fog of war), the agent does not perceive the complete state s_t. Instead, it receives an *observation* o_t, which contains only partial information about s_t.
- **Challenge:** Standard RL methods like DQN $Q(s, a; w)$ or policy networks $\pi(a|s; \theta)$ require the full state s as input. Directly substituting the observation o_t for the state s, i.e., using $Q(o_t, a; w)$ or $\pi(a|o_t; \theta)$, is often insufficient because o_t may lack critical information needed for optimal decision-making.
- **Need for Memory:** As illustrated by the maze example, relying solely on the current observation can lead to poor decisions (e.g., getting stuck). Humans and effective agents in partially observable environments need to remember past observations to build a more complete understanding of the underlying state.
- **History as State:** A common approach is to use the entire history of observations up to time t, denoted $o_{1:t} = [o_1, o_2, \ldots, o_t]$, as the basis for decision-making. The policy becomes $\pi(a_t|o_{1:t}; \theta)$.
- **Variable Input Length:** A key challenge is that the history $o_{1:t}$ grows in length over time. Standard feedforward networks (MLPs, CNNs) require fixed-size inputs and cannot directly process variable-length sequences like $o_{1:t}$.

(a) 对状态的完全观测 (b) 对状态的不完全观测 (c) 记忆过去的观测

Fig. 60:

42 Recurrent Neural Networks (RNN)

RNNs are designed to process sequential data of variable length, making them suitable for handling observation histories.

- **Function:** An RNN layer takes a sequence of input vectors $(x_1, x_2, \ldots, x_n)$ and produces a sequence of hidden state vectors $(h_1, h_2, \ldots, h_n)$. Crucially, the hidden state h_t at time t is

computed based on the current input x_t and the *previous* hidden state h_{t-1}. Thus, h_t serves as a summary or memory of the input sequence up to time t, i.e., $h_t = f(x_1, \ldots, x_t)$. The size of the hidden state vector h_t is fixed, regardless of the sequence length t.

- **Example Application:** To classify the sentiment of a review (sequence of words $w_1, \ldots, w_n$), each word w_t is converted to a vector x_t. These are fed sequentially into an RNN. The final hidden state h_n, which summarizes the entire review, is then passed to a classifier (e.g., an MLP with a Sigmoid output) to predict the sentiment probability $\hat{p}$.

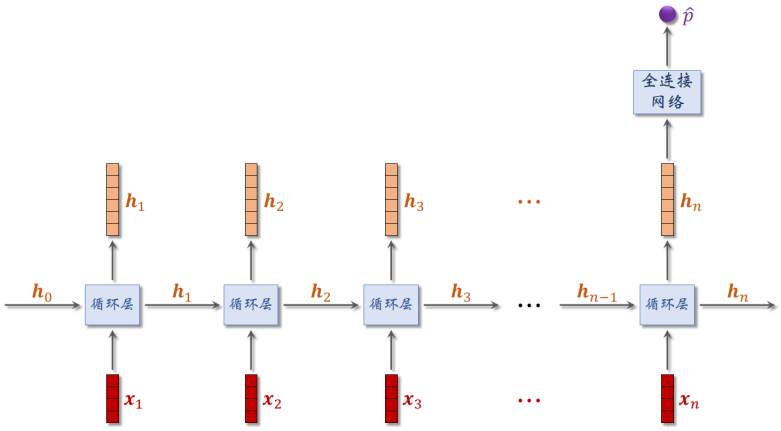


Fig. 61: RNN

- **Types of RNN Layers:** Common types include Simple RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). LSTM and GRU have more complex internal mechanisms (gates) to better handle long-term dependencies compared to Simple RNNs, but the fundamental principle of using past hidden states remains.

- **Simple RNN Layer:**

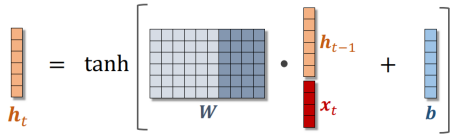


Fig. 62:

  - Input: Sequence $x_1, \ldots, x_n \in \mathbb{R}^{d_{\text{in}}}$.
  - Output: Sequence $h_1, \ldots, h_n \in \mathbb{R}^{d_{\text{out}}}$.
  - Parameters: Weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times (d_{\text{in}} + d_{\text{out}})}$ and bias vector $b \in \mathbb{R}^{d_{\text{out}}}$. Note that W and b are shared across all time steps.

– Update Rule: For $t = 1, \ldots, n$:

$$h_t = \tanh(W[h_{t-1}; x_t] + b)$$

where $[h_{t-1}; x_t]$ denotes the concatenation of the previous hidden state h_{t-1} (with h_0 typically initialized to zeros) and the current input x_t. The tanh activation function, $y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, is applied element-wise.

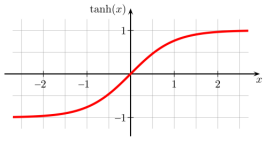


Fig. 63: tanh

43 RNN as a Policy Network

To handle partial observability in RL, an RNN can be incorporated into the policy network architecture.

- **Architecture:**

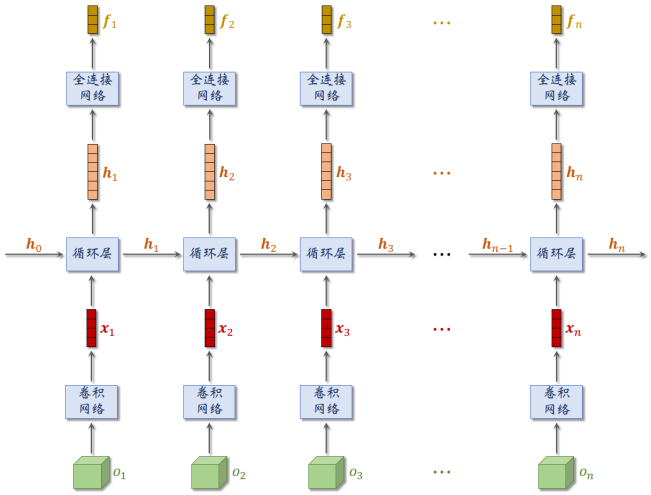


Fig. 64: policy network based on RNN

1. At time t, the current observation o_t is received.
2. (Optional) If o_t is high-dimensional (like image), it's first processed by feature extraction layers (e.g., CNNs) to produce a lower-dimensional feature vector x_t. These feature extraction layers typically share parameters across time steps.

3. The feature vector x_t and the previous hidden state h_{t-1} are fed into the RNN layer (e.g., Simple RNN, LSTM, or GRU) to compute the current hidden state h_t

$$h_t = \text{RNN}(h_{t-1}, x_t; w_{\text{rnn}})$$

4. The hidden state h_t, which summarizes the history $o_{1:t}$, is passed to an output network (e.g., MLP with Softmax activation) to produce the action probabilities $\mathbf{f}_t$

$$\mathbf{f}_t = \text{MLP}(h_t; w_{\text{out}})$$

5. The action a_t is sampled from the distribution $\mathbf{f}_t$.

- **Parameters:** The overall policy network parameters θ include the parameters of the feature extractor (if any), the RNN layer (w_{rnn}), and the output network (w_{out}).
- **Training:** Such an RNN-based policy network can be trained using policy gradient methods (like REINFORCE or A2C), where gradients are computed via backpropagation through time (BPTT).
- **Application to Value Networks/DQN:** Similar architectures can be used for value-based methods in partially observable settings. For instance, a Deep Recurrent Q-Network (DRQN) would take o_t as input at each step, update its hidden state h_t, and use h_t to predict Q-values for different actions a_t. The value network could be $Q(o_{1:t}, a_t; w)$ or $V(o_{1:t}; w)$, approximated by processing h_t through an appropriate output layer.

Chapter 12: Imitation Learning

Imitation Learning (IL) is presented not as a subfield of Reinforcement Learning (RL), but as an alternative approach with the same goal: learning a policy network to control an agent. Unlike RL, which uses environmental rewards to maximize cumulative return, IL learns by mimicking the actions of an expert (typically human).

44 Behavior Cloning (BC)

BC is the simplest form of imitation learning, essentially treating policy learning as a supervised learning problem.

- **Dataset:** Requires a dataset of expert demonstrations, consisting of state-action pairs: $\mathcal{X} = \{(s_1, a_1), \ldots, (s_n, a_n)\}$, where a_i is the action taken by the expert in state s_i.

- **Goal:** Train a policy network (either deterministic $\mu(s;\theta)$ or stochastic $\pi(a|s;\theta)$) to replicate the expert's actions given the same states.
- **No Environment Interaction:** BC is trained offline purely on the expert dataset and does not require the agent to interact with the environment during training.

44.1 Continuous Control

- **Policy Network:** Use a deterministic policy network $a = \mu(s;\theta)$, where the output a is a continuous action vector.

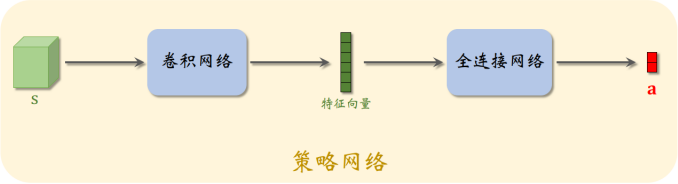


Fig. 65: policy network

- **Supervised Learning Problem:** This is treated as a regression problem where the input is state s and the target output is the expert action a.
- **Loss Function:** Typically the Mean Squared Error (MSE) between the network's output and the expert's action:

$$L(s,a;\theta) = \frac{1}{2}||\mu(s;\theta) - a||^2$$

- **Training:** Use Stochastic Gradient Descent (SGD) to update parameters θ by minimizing the loss over the dataset $\mathcal{X}$:

$$\theta \leftarrow \theta - \beta \cdot \nabla_\theta L(s_j, a_j; \theta)$$

where (s_j, a_j) is a sampled pair from $\mathcal{X}$.

44.2 Discrete Control

- **Policy Network:** Use a stochastic policy network $\pi(a|s;\theta)$, which outputs a probability distribution f over the discrete action space $\mathcal{A}$.

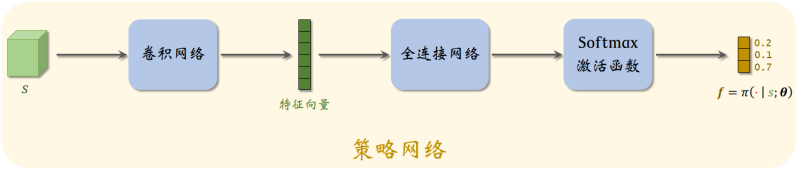


Fig. 66: policy network

- **Supervised Learning Problem:** This is treated as a classification problem.
- **Target Label:** The expert action a_j is converted into a one-hot encoded vector $\bar{a}_j$ (e.g., if $a_j =$ 'up' and $\mathcal{A} = \{\text{left, right, up}\}$, then $\bar{a}_j = [0, 0, 1]^T$).
- **Loss Function:** Typically the Cross-Entropy loss between the network's output distribution $f = \pi(\cdot|s; \theta)$ and the one-hot target $\bar{a}$:

$$H(\bar{a}, f) = -\sum_{i=1}^{|\mathcal{A}|} \bar{a}_i \ln f_i$$

  (Since $\bar{a}$ is one-hot, this simplifies to $-\ln f_k$ where k is the index of the expert action).
- **Training:** Use SGD to update θ by minimizing the cross-entropy loss over the dataset $\mathcal{X}$:

$$\theta \leftarrow \theta - \beta \cdot \nabla_\theta H(\bar{a}_j, \pi(\cdot|s_j; \theta))$$

44.3 Behavior Cloning vs. Reinforcement Learning

- **Interaction:** BC learns offline from a fixed dataset without environment interaction. RL learns online (or offline with modifications) through trial-and-error interaction with the environment, using rewards as feedback.
- **Limitations of BC:**
  - **Distribution Mismatch:** The agent trained via BC might encounter states during execution that were not present in the expert dataset. Its behavior in these novel states can be unpredictable and poor.
  - **Compounding Errors:** Small errors made by the BC policy can lead the agent into states further away from the expert distribution, potentially leading to larger errors, causing a cascade of failures.
  - **Suboptimality:** BC can only replicate the expert's behavior; it cannot surpass it. The expert might not be optimal.
- **Advantages of BC:** Low cost, no potentially harmful exploration in the real world. Can be used effectively for pre-training a policy before fine-tuning with RL.
- **Advantages of RL:** Can potentially learn policies better than the expert, adapts better to unseen states through exploration, doesn't suffer from compounding errors in the same way (errors are corrected by reward signal).

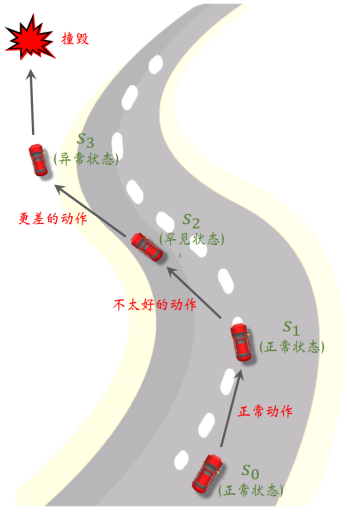


Fig. 67: example

45 Inverse Reinforcement Learning (IRL)

IRL aims to infer the underlying *reward function* R^* that explains an expert's behavior π^*, assuming the expert is acting optimally with respect to R^*. Once R^* (or an approximation) is found, standard RL can be used to learn a policy.

45.1 Setup

- Access to an expert policy π^* (often as a black box or via demonstrations).
- Ability for the agent to interact with the environment (but rewards are not provided by the environment).
- Assumption: π^* is optimal for some unknown reward function R^*, i.e.,

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{k=t}^{n} \gamma^{k-t} R^*(S_k, A_k)\right]$$

45.2 Basic Idea (Apprenticeship Learning)

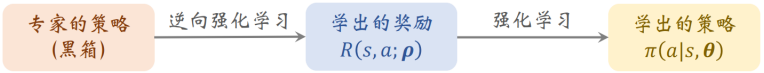


Fig. 68: policy network based on RNN

1. **Learn Reward:** Infer an approximation of the expert's reward function, $R(s, a; \rho)$, from demonstrations of π^*. This involves finding parameters ρ such that an agent trained via RL on $R(s, a; \rho)$ behaves similarly to π^*.

2. **Learn Policy:** Use the learned reward function $R(s, a; \rho)$ to train a policy $\pi(a|s; \theta)$ using standard RL methods (like policy gradients).

45.3 Inferring Reward Example

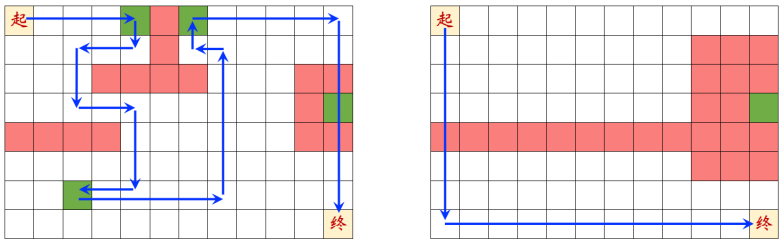


Fig. 69: example

By observing expert trajectories in a grid world (avoiding red cells, seeking green cells/goal), one can qualitatively infer properties of the reward function (e.g., negative reward for red cells, positive for green/goal, small cost per step). However, the exact reward values are often non-unique (scaling rewards doesn't change the optimal policy).

45.4 Training Policy with Learned Reward

Once $R(s, a; \rho)$ is available:

1. Agent follows its current policy $\pi(\cdot|\cdot; \theta)$ in the environment, generating (s_t, a_t, s_{t+1}) sequences.
2. Compute rewards using the learned function: $\hat{r}_t = R(s_t, a_t; \rho)$.
3. Use these computed rewards $\hat{r}_t$ in any standard RL algorithm (e.g., REINFORCE) to update the policy parameters θ. For REINFORCE, compute returns $\hat{u}_t = \sum_{k=t}^{n} \gamma^{k-t} \hat{r}_k$, then update:

$$\theta \leftarrow \theta + \beta \sum_{t=1}^{n} \gamma^{t-1} \hat{u}_t \nabla_\theta \ln \pi(a_t|s_t; \theta)$$

46 Generative Adversarial Imitation Learning (GAIL)

GAIL applies the framework of Generative Adversarial Networks (GANs) to imitation learning. It avoids explicitly learning a reward function and instead trains a policy (Generator) to produce trajectories that are indistinguishable from expert trajectories to a Discriminator network. Requires interaction.

46.1 GAN Background

- **Components:**

– **Generator $G(z;\theta)$:** Takes random noise z as input, outputs fake data x_{fake} (e.g., images).

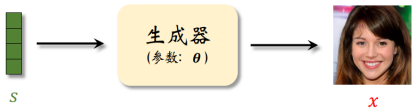


Fig. 70: generate

– **Discriminator $D(x;\phi)$:** Takes data x (real or fake) as input, outputs probability $\hat{p}$ that x is real.

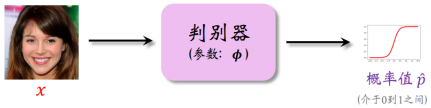


Fig. 71: Adversarial

- **Adversarial Training:** G tries to fool D, D tries to correctly classify real vs. fake.

  – **Train D:** Update ϕ to maximize $\log D(x_{\text{real}};\phi) + \log(1 - D(x_{\text{fake}};\phi))$. Sample x_{real} from dataset, generate $x_{\text{fake}} = G(z;\theta)$.

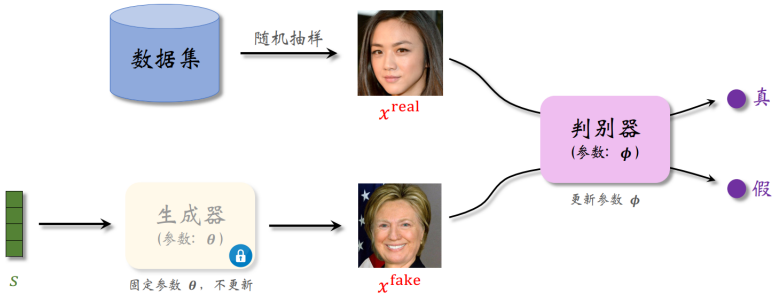


Fig. 72: train Adversarial

  – **Train G:** Update θ to maximize $\log D(G(z;\theta);\phi)$ (equivalent to minimizing $\log(1-D(G(z;\theta);\phi))$). Generate $x_{\text{fake}} = G(z;\theta)$.

Fig. 73: train generate

- Alternate training steps for D and G.

46.2 GAIL Generator and Discriminator

- **Data:** Expert trajectories $\mathcal{X} = \{\tau^{(1)}, \ldots, \tau^{(k)}\}$ where $\tau = (s_1, a_1, \ldots, s_m, a_m)$.
- **Generator:** The agent's policy network $\pi(a|s;\theta)$. It generates agent trajectories $\tau_{\text{agent}} = (s_1, a_1, \ldots, s_n, a_n)$ by interacting with the environment.

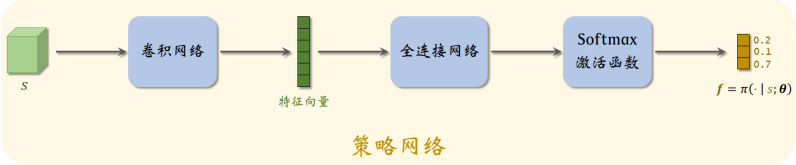


Fig. 74: policy network

- **Discriminator** $D(s, a; \phi)$**:** Takes a state-action pair (s, a) as input and outputs the probability $D(s, a; \phi)$ that this pair came from the expert distribution rather than the agent's policy.

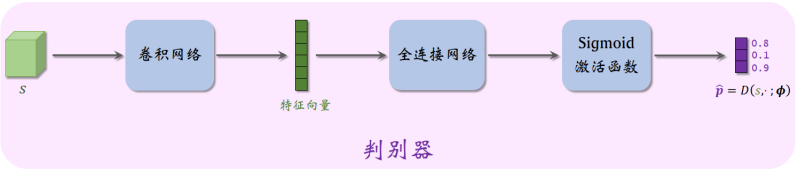


Fig. 75: adversarial

46.3 GAIL Training

Alternate training the Generator (policy) and the Discriminator.

1. **Train Discriminator** D**:**
   - Sample expert state-action pairs (s_e, a_e) from $\mathcal{X}$.
   - Generate agent state-action pairs (s_π, a_π) using the current policy $\pi(\cdot|s;\theta)$.
   - Update ϕ using SGD to maximize the GAN objective (distinguish expert from agent pairs):

$$\max_{\phi} \mathbb{E}_{(s,a)\sim\pi_{\text{expert}}}[\log D(s, a; \phi)] + \mathbb{E}_{(s,a)\sim\pi_\theta}[\log(1 - D(s, a; \phi))]$$

2. **Train Generator** π **(Policy Update):**
   - Goal: Update θ so that the generated state-action pairs (s_π, a_π) get high scores from the current discriminator D.
   - Treat $-\log(1 - D(s, a; \phi))$ or $\log D(s, a; \phi)$ as a reward signal $r(s, a)$.
   - Generate agent trajectories using $\pi(\cdot|s;\theta)$.

- Compute the "rewards" $r_t = \log D(s_t, a_t; \phi)$ for the generated state-action pairs using the fixed discriminator.
- Update the policy parameters θ using a standard policy optimization algorithm (like TRPO, as mentioned in the text, or PPO) to maximize the expected cumulative "reward" $\mathbb{E}[\sum_t \gamma^t r_t]$.

Repeat these steps. GAIL learns a policy directly without intermediate reward function learning, often achieving good results by matching state-action distributions.

Part Four: Multi-Agent Reinforcement Learning (MARL)

This part extends the concepts of reinforcement learning from single-agent scenarios to settings involving multiple agents interacting within a shared environment.

Chapter 13: Parallel Computing

- **Goal:** To accelerate RL training (reduce wall-clock time) by utilizing multiple processors or machines.
- **Basics:** Introduces parallel gradient descent using frameworks like MapReduce (broadcast, map, reduce) and concepts like data parallelism.
- **Costs:** Discusses limitations to speedup, including communication overhead (latency, bandwidth) and synchronization costs.
- **Synchronous vs. Asynchronous:**
  - **Synchronous algorithms:** All workers must wait for the slowest one before proceeding (e.g., MapReduce GD). Suffers from "straggler effect".
  - **Asynchronous algorithms:** Workers operate independently, communicating with a central server (e.g., parameter server architecture) without waiting. Improves utilization and often reduces wall-clock time, but may introduce issues due to stale gradients.
- **Parallel RL Examples:** Describes asynchronous parallel implementations for DQN and A3C (Asynchronous Advantage Actor-Critic), where multiple workers collect data and compute gradients in parallel, sending updates to a central parameter server.

Chapter 14: Multi-Agent Systems

- **MAS vs. SAS:** Defines Multi-Agent Systems (MAS) where multiple agents share an environment, and their actions mutually influence outcomes, contrasting this with Single-Agent Systems (SAS) or parallel independent SASs. MARL is RL applied to MAS.
- **Common Settings:** Categorizes MARL problems into:
  - Fully Cooperative: All agents share the same reward signal ($R^1 = \cdots = R^m$).
  - Fully Competitive: Agents have opposing goals (e.g., zero-sum games, $R^1 \propto -R^2$).
  - Mixed Cooperative-Competitive: Agents form teams that cooperate internally but compete externally.
  - Self-Interested: Each agent maximizes its own reward, irrespective of others (though actions may still affect others).
- **MARL Basic Concepts:** Extends single-agent terms: introduces joint action $a = (a^1, \ldots, a^m)$, joint action space $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$. State transition $p(s'|s, a)$ depends on joint action. Individual rewards R^i, returns U^i. Individual policies $\pi^i(a^i|s; \theta^i)$ or $\mu^i(o^i; \theta^i)$. Individual value functions $(Q^i_\pi(s, a), V^i_\pi(s))$, emphasizing that these depend on the policies of *all* agents $\pi = (\pi^1, \ldots, \pi^m)$. Covers partial observability (O^i vs S).
- **Experimental Environments:** Introduces common MARL benchmarks like Multi-Agent Particle Environments, StarCraft Multi-Agent Challenge (SMAC), and the Hanabi Challenge.

Chapter 15: MARL in Cooperative Settings

- **Setting:** Focuses on the fully cooperative case ($R^1 = \cdots = R^m = R$). This leads to a shared return U, shared value functions $Q_\pi(s, a), V_\pi(s)$, and a shared objective function $J(\theta^1, \ldots, \theta^m) = \mathbb{E}_S[V_\pi(S)]$.
- **Goal & Optimization:** All agents cooperate to find joint policy parameters $(\theta^1, \ldots, \theta^m)$ that maximize the common objective J, often via distributed gradient ascent where each agent i updates θ^i using $\nabla_{\theta^i} J$.
- **MAC-A2C (Multi-Agent Cooperative A2C):** An illustrative algorithm using a single shared value network $v(s; w) \approx V_\pi(s)$ and individual policy networks $\pi^i(a^i|s; \theta^i)$. Updates follow A2C principles using the shared reward and value function. On-policy.
- **Implementation Architectures & Challenges:** Addresses the issue that agents typically only have local observations o^i, while centralized components might need the global state $s = (o^1, \ldots, o^m)$.
  1. **Fully Centralized (Training & Execution):** A central controller manages everything. Accurate but high communication latency.

2. **Fully Decentralized (Independent Learning):** Each agent learns independently using only o^i. Fast, no communication, but often suboptimal.
3. **Centralized Training + Decentralized Execution (CTDE):** Trains centralized critics ($v(s; w)$) using global information but decentralized actors ($\pi^i(a^i|o^i; \theta^i)$). Execution is fast and local. Most practical approach.

Chapter 16: MARL in Non-Cooperative Settings

- **Setting:** Agents have individual rewards R^i, leading to distinct value functions (V_π^i, Q_π^i) and objectives $J^i(\theta^1, \ldots, \theta^m)$. General-sum games.
- **Goal & Convergence:** Each agent i maximizes its own J^i. Convergence is typically assessed via Nash Equilibrium (no agent can unilaterally improve its outcome).
- **MAN-A2C (Multi-Agent Non-Cooperative A2C):** An illustrative algorithm where each agent i has its own policy $\pi^i(a^i|s; \theta^i)$ and value network $v^i(s; w^i)$. Updates follow A2C principles using individual rewards r_t^i and TD errors δ_t^i.
- **Architectures:** The three architectures (centralized, decentralized, CTDE) apply. CTDE uses local policies $\pi^i(a^i|o^i; \theta^i)$ but central critics $v^i(s; w^i)$.
- **MADDPG (Multi-Agent DDPG):** Extends DDPG for continuous control MARL, using CTDE. Each agent i learns a local deterministic policy $\mu^i(o^i; \theta^i)$ (Actor) and a central critic $q^i(s, a; w^i)$ that takes global state s and joint action a as input. Off-policy, uses experience replay. Can be improved with TD3 techniques. Execution is decentralized.

Chapter 17: Attention Mechanism and MARL

- **Self-Attention:** Introduces the self-attention mechanism for processing sequences $(x^1, \ldots, x^m)$ into $(c^1, \ldots, c^m)$. Each output c^i is a weighted sum of transformed inputs, where weights reflect the relevance of other inputs x^j to x^i. Handles variable number of agents m. Multi-Head Attention improves robustness.
- **Application in MARL:** Suggests using self-attention within centralized components (critic or policy) in CTDE or fully centralized architectures. The input sequence can be features derived from local observations $(o^1, \ldots, o^m)$ or state-action pairs. Self-attention allows the model to dynamically weigh the importance of information from different agents when computing value functions or centralized policies, improving scalability and performance, especially for large m.

In summary, Part Four transitions from accelerating single-agent RL with parallel computing to the complexities of multi-agent interactions. It defines MARL settings, extends core RL concepts, explores cooperative and non-cooperative algorithms (like MAC-A2C, MAN-A2C, MADDPG), discusses the

crucial CTDE architecture, and introduces attention mechanisms as a way to manage information flow in large multi-agent systems.

Part Five: Applications and Outlook

This final part showcases prominent applications of Deep Reinforcement Learning (DRL) and discusses its practical challenges and future directions.

Chapter 18: AlphaGo and Monte Carlo Tree Search (MCTS)

This chapter details the workings of AlphaGo, focusing on Monte Carlo Tree Search (MCTS) as a key example of model-based RL.

47 AlphaGo Overview

Explains the significance of AlphaGo in defeating human world champions in Go. Describes its core components:

- **State Representation:** Using a 19x19x17 tensor to represent board positions, recent history, and current player.
- **Policy Network $\pi(a|s;\theta)$:** A deep convolutional network predicting the probability distribution over possible moves a given state s.
- **Value Network $v(s;w)$:** A deep convolutional network estimating the probability of the current player winning from state s.

48 Monte Carlo Tree Search (MCTS)

The primary decision-making algorithm used by AlphaGo during gameplay. MCTS builds a search tree to evaluate moves by simulating potential future game states. Each simulation involves four steps, repeated thousands of times per move:

1. **Selection:** Traverse the existing search tree from the root (current state) using a criterion (like UCT: $score(a) = Q(a) + \eta\frac{\pi(a|s;\theta)}{1+N(a)}$) balancing exploitation ($Q(a)$, estimated value) and exploration ($N(a)$, visit count; $\pi(a|s;\theta)$, policy prior).
2. **Expansion:** Expand a leaf node by adding children (next possible states). Use the policy network π for priors and to simulate the opponent's move (using π as the environment model).

3. **Evaluation (Rollout):** Estimate the value of the new node, often by combining a fast rollout simulation result ($r = \pm 1$) with the value network estimate ($v(s_{\text{new}}; w)$): $V = (r + v)/2$.

4. **Backup:** Propagate the value V up the tree, updating visit counts $N(a)$ and average action values $Q(a)$ for nodes/actions on the path.

48.1 Final Move Selection

After many simulations, MCTS selects the root action a_t with the highest visit count $N(a_t)$.

48.2 Training AlphaGo Versions

- **AlphaGo (2016):** (1) Supervised pre-training of π on human games (Behavior Cloning). (2) RL fine-tuning of π using REINFORCE self-play. 3) Train v via regression on outcomes from π's self-play games.
- **AlphaGo Zero (2017):** Trained purely from self-play using MCTS. Trains π and v jointly. Policy target for π is MCTS visit counts distribution; value target for v is game outcome (± 1). MCTS uses current π, v.

Chapter 19: Applications in the Real World

This chapter explores several real-world application domains for DRL beyond games and discusses practical limitations.

49 Neural Architecture Search (NAS)

Using RL (RNN controller trained with REINFORCE) to design neural network architectures. Actions = hyperparameters, Reward = validation accuracy. Computationally expensive; non-RL methods often preferred now.

50 SQL Generation

Using seq2seq models (policy) trained with RL to translate natural language to SQL. Reward based on execution success/correctness, not just syntax matching. Addresses limitations of supervised approaches.

51 Recommendation Systems

Framing recommendation as RL (state = user context, action = item recommendation, reward = user feedback). Aims for long-term engagement (return). RL can explore to find new user interests.

Challenges: huge action space, reward definition, cost of online exploration. Offline pre-training is crucial.

52 Ride-Sharing Dispatch (DiDi Example)

Using value-based RL to learn V_π(location, time) (expected future driver earnings). Dispatch decisions aim to maximize total driver income, considering immediate reward and future value (related to TD error). Uses bipartite matching based on these values.

53 RL vs. Supervised Learning

- RL: Suitable for sequential decisions, delayed rewards, when actions affect the environment, and exploration is needed.
- Supervised Learning: Assumes i.i.d data, decisions don't affect future inputs, focuses on immediate prediction accuracy.

54 Challenges Limiting DRL Applications

1. **Sample Inefficiency:** Requires massive amounts of data, often infeasible in the real world.
2. **Exploration Cost/Safety:** Initial random exploration can be dangerous or costly.
3. **Hyperparameter Sensitivity:** Performance heavily depends on careful tuning.
4. **Instability/Reproducibility Issues:** Training can be unstable and results vary significantly with random seeds.

Overall, Part Five illustrates DRL's power in complex tasks like Go, highlights potential applications, but also emphasizes significant practical challenges limiting its widespread adoption.