Inteligência Artificial

Raoni F. S. Teixeira

Aula 15 - Aprendizado por Reforço

1 Aprendizado por Reforço

Diferente do aprendizado supervisionado, em que o modelo aprende com exemplos rotulados, o **Aprendizado por Reforço** (*Reinforcement Learning* – RL) é baseado em tentativa e erro. O agente aprende interagindo com o ambiente, como um jogador de videogame que testa diferentes ações e melhora sua pontuação com o tempo.

2 Como Funciona

O ciclo de aprendizado por reforço acontece em quatro passos:

1. O agente escolhe e executa uma ação.
2. O ambiente responde e muda de estado.
3. O ambiente envia uma recompensa.
4. O agente ajusta sua estratégia com base no que aconteceu.

A Figura 1 mostra o ciclo de interação que se repete a cada instante t: o agente executa uma ação a_t, observa o novo estado s_{t+1} do ambiente e recebe uma recompensa R_{t+1}. O objetivo do agente é **maximizar a recompensa acumulada** ao longo da interação com o ambiente.

Para entender melhor, imagine uma criança jogando videogame pela primeira vez. Ela aperta botões aleatoriamente, erra muito, mas aos poucos percebe quais ações rendem pontos. Ao repetir esse processo, ela aprende a jogar melhor. O agente em RL faz exatamente isso: experimenta, observa o que funciona, e melhora com a prática.

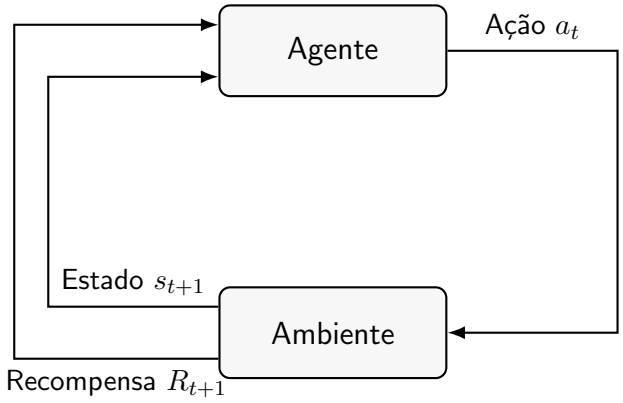

Figura 1: Ciclo de interação entre agente e ambiente.

3 Formulação Matemática

No aprendizado por reforço, um agente toma decisões em ambientes onde apenas o estado atual define o próximo estado — a chamada **Propriedade de Markov**. Essa independência do histórico permite modelar o problema como um **Processo de Decisão Markoviano (MDP)**, com cinco componentes:

- **Estados** ($\mathcal{S}$): Todas as situações possíveis do ambiente
- **Ações** ($\mathcal{A}$): Movimentos possíveis que o agente pode fazer
- **Função de Transição** ($P(s'|s,a)$): Probabilidade de alcançar o estado s' após executar a ação a no estado s
- **Função de Recompensa** ($R(s,a)$): Função que mapeia estados e ações para recompensas imediatas $R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$.
- **Fator de Desconto** (γ): Quanto valorizamos recompensas futuras ($0 \leq \gamma < 1$)

A política π é o cérebro do agente: transforma uma observação do ambiente (estado) em uma uma ação (ou uma distribuição de probabilidade sobre as ações). Existem dois tipos:

- **Determinísticas**: $\pi(s) = a$ (ação fixa para cada estado).
- **Estocásticas**: $\pi(a|s)$ (distribuição de probabilidade sobre ações).

A cada passo discreto, o agente observa o estado atual s_t, seleciona uma ação a_t conforme sua política π, e recebe do ambiente tanto a recompensa imediata $R(s_t, a_t)$ quanto o novo estado $s_{t+1} \sim P(s_t, a_t)$ — reiniciando esse ciclo para $t+1$ até o término da tarefa.

A **recompensa total** é a soma descontada das recompensas em cada passo:

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

em que $\gamma \in [0, 1]$ é o fator de desconto. Quando $\gamma \approx 0$, o agente foca em recompensas imediatas. Quando $\gamma \approx 1$, o agente planeja a longo prazo.

As tarefas que o agente enfrenta podem ser **episódicas** (fases com reinício, como jogos) ou **contínuas** (sem fim pré-definido, como controle robótico), esta última demandando balanceamento permanente entre explorar e aproveitar.

O objetivo é encontrar a **política ótima** (π^*), que define a melhor estratégia para maximizar a recompensa total ao longo do tempo.

3.1 Funções de Valor

Para avaliar uma política, usamos as funções **valor de estado** e **valor de ação**. A função valor de estado $V^\pi(s)$ estima o retorno esperado ao iniciar no estado s e seguir a política π.

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\left[G_t \mid s_t = s, \pi\right] \\ &= \mathbb{E}\Big\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid s_t = s, \pi\Big\}. \end{aligned} \tag{1}$$

Já a função valor de ação $Q^\pi(s, a)$ estima esse retorno se começarmos no estado s, tomarmos a ação a e depois seguirmos π:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}\left[G_t \mid s_t = s, a_t = a, \pi\right] \\ &= \mathbb{E}\Big\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid s_t = s, a_t = a, \pi\Big\}. \end{aligned} \tag{2}$$

Essas funções são úteis porque podem ser calculadas recursivamente usando as **equações de Bellman**, que dividem o valor de uma decisão em duas partes: a recompensa imediata e o valor esperado das decisões futuras.

Para qualquer política π, temos:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\left[G_t \mid s_t = s, \pi\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid s_t = s\right] \\ &= \underbrace{\mathbb{E}\left[R_{t+1} \mid s_t = s\right]}_{\text{Recompensa imediata}} + \underbrace{\gamma \mathbb{E}\left[G_{t+1} \mid s_t = s\right]}_{\text{Valor esperado das ações futuras}} \\ &= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) R(s,a,s') + \gamma \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) V^\pi(s') \\ &= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^\pi(s')\right]. \end{aligned} \tag{3}$$

E:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}\left[G_t \mid s_t = s, a_t = a\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a\right] \\ &= \underbrace{\mathbb{E}\left[R_{t+1} \mid s_t = s\right]}_{\text{Recompensa imediata}} + \underbrace{\gamma \mathbb{E}\left[G_{t+1} \mid s_t = s\right]}_{\text{Valor esperado das ações futuras}} \\ &= \sum_{s'} P(s'|s,a) R(s,a,s') + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid s_t = s, a_t = a\right] \\ &= \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s',a')\right]. \end{aligned} \tag{4}$$

Para calcular o valor ótimo Q^*, não seguimos uma política fixa π. Em vez disso, escolhemos sempre a melhor ação possível em cada estado. Formalmente, a política ótima π^* é:

$$\pi^*(a' \mid s') = \begin{cases} 1 & \text{para a ação } a' \text{ que maximiza } Q^*(s', a'') \\ 0 & \text{para todas as outras ações} \end{cases}$$

Substituindo o trecho na equação original de Q^π (Equação 4), obtemos:

$$\sum_{a'} \pi^*(a' \mid s') Q^*(s',a') = \max_{a'} Q^*(s',a')$$

Isso nos leva à **equação de Bellman para** Q^*:

$$Q^*(s,a) = \sum_{s'} P(s' \mid s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{5}$$

A intuição por trás dessas equações é simples: se uma política é ótima, qualquer trecho dela também deve ser. Isso significa que as decisões ideais em qualquer ponto dependem apenas do estado atual, não do caminho percorrido até ele.

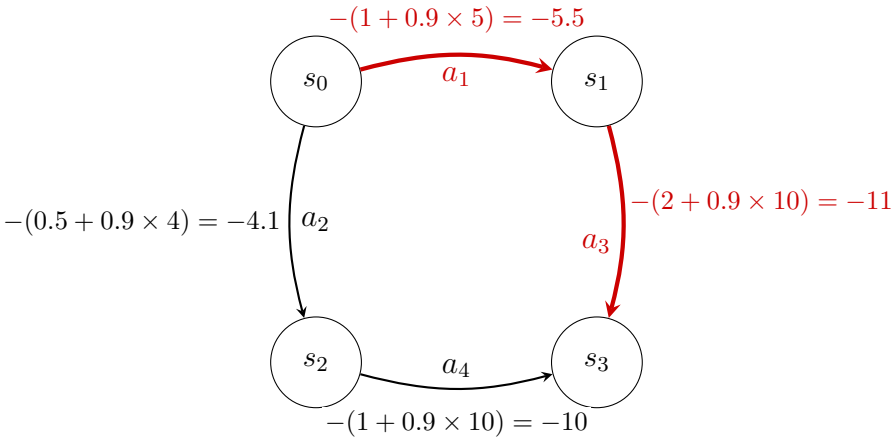

Figura 2: **Analogia MDP-Grafo**: Cada transição a_i tem um custo calculado como $-[R+\gamma V^{\pi}(s')]$. A estrela vermelha marca o estado-alvo s_3, e o caminho vermelho representa a trajetória ótima.

3.2 Analogia com caminhos mínimos

Um processo de decisão de Markov (MDP) pode ser representado como um grafo direcionado. Nele, os vértices são os estados e as arestas representam ações. O peso de cada aresta é calculado como $-[R+\gamma V^{\pi}(s')]$. Com isso, o problema de maximizar recompensas se transforma em um problema de minimizar custos — como na busca de caminhos mínimos.

A Figura 2 mostra essa analogia em um ambiente com quatro estados. Usamos $\gamma = 0.9$, valor estimado $V^{\pi}(s_3) = 10$ e recompensa imediata $R = 2$. O custo da transição de s_1 para s_3 é:

$$\text{Custo de } s_1 \to s_3 = 2 + 0.9 \times 10 = -11.$$

Minimizar a soma dos pesos no grafo equivale a maximizar o total esperado de recompensas. Como no algoritmo de Bellman-Ford, o caminho ótimo do estado inicial ($s_0 \to s_1 \to s_3$) é composto por subcaminhos ótimos ($s_1 \to s_3$). A política ótima π^* escolhe ações que minimizam o custo acumulado – o que equivale a maximizar o retorno esperado G_t.

De fato, a equação de Bellman para $Q^*(s,a)$ tem exatamente a mesma estrutura de um problema de caminho mínimo em um grafo acíclico, onde cada aresta representa uma transição ponderada pelo custo esperado, e a solução ótima é obtida pela propagação recursiva de valores como em algoritmos clássicos de grafos.

A principal diferença entre os métodos clássicos de programação dinâmica e os algoritmos de aprendizagem por reforço está na suposição do modelo do ambiente. Os primeiros exigem conhecer as probabilidades de transição e recompensas; os segundos, não. Eles funcionam

mesmo em grandes MDPs, onde os métodos exatos se tornam inviáveis.

Quando não conhecemos o modelo (ou seja, $P(s'|s,a)$ e $R(s,a,s')$), só podemos aprender interagindo com o ambiente. É o caso do método Q-Learning, que veremos na próxima seção.

4 Q-Learning

O **Q-Learning** é um algoritmo de aprendizado por reforço *off-policy* (i.e. aprende uma política seguindo ourta) que aprende uma aproximação da função ação-valor $Q(s,a)$: a recompensa total esperada que um agente pode obter ao executar a ação a no estado s, e depois seguir a melhor política possível.

Essa função é aprendida iterativamente com base nas transições observadas pelo agente. Dada uma experiência (s,a,r,s'), a atualização ocorre pela seguinte regra:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \right]$$

O termo entre colchetes é chamado de **erro temporal** (TD error), e mede a diferença entre o valor atual de $Q(s,a)$ e uma estimativa mais atualizada baseada na transição observada:

$$\delta = r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)$$

Se δ for grande, significa que a estimativa anterior de $Q(s,a)$ estava incorreta; se for pequena, a estimativa já estava próxima do valor real. Esse erro guia o aprendizado do agente ao longo dos episódios.

Normalmente, inicializamos $Q(s,a)$ com valores zeros ou aleatórios. Essa incerteza inicial estimula a exploração, especialmente ao usar políticas como a ε-greedy.

O pseudo-código é apresentado a seguir. O algoritmo recebe uma política de exploração, um conjunto de estados $\mathcal{S}$, um conjunto de ações $\mathcal{A}$ uma taxa de aprendizado α ($0 < \alpha \leq 1$), um fator de desconto γ ($0 \leq \gamma \leq 1$) e um número máximo de episódios N. A saída é a uma aproximação da função ação-valor Q.

Q-LEARNING($\mathcal{S}, \mathcal{A}, \alpha, \gamma$, política de exploração, N)

for cada par $(s, a) \in \mathcal{S} \times \mathcal{A}$
    $Q(s, a) \leftarrow$ valor arbitrário
$Q(\text{terminal}, \cdot) \leftarrow 0$
for cada episódio $\leftarrow 1$ **to** N
        Inicialize o estado s
        while s não é terminal
            Escolha ação a em s usando política de exploração (e.g., ε-greedy)
            Execute a, observe recompensa r e próximo estado s'
            if s' é terminal **then**
                $\delta \leftarrow r$
            else
                $\delta \leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$

            $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (\delta - Q(s, a))$
            $s \leftarrow s'$
    return Q

O algoritmo *Q-Learning* também pode ser interpretado como um processo de busca de caminhos mínimos em um grafo direcionado acíclico (DAG), em que os estados são nós e as ações determinam as transições. A equação de atualização de Q tem a mesma forma da equação de Bellman para caminhos mínimos: ela propaga valores ótimos de retorno a partir dos estados terminais, assim como o algoritmo de Bellman-Ford propaga distâncias a partir da origem.

No entanto, ao contrário de Bellman-Ford, que conhece os pesos das arestas, o Q-Learning **não conhece previamente as probabilidades de transição** $P(s'|s, a)$ **nem as recompensas** $R(s, a, s')$. Por isso, ele precisa repetir o processo várias vezes, *interagindo com o ambiente* para estimar os valores de $Q(s, a)$ com base na experiência. Cada valor $Q(s, a)$ representa o "custo invertido" de uma aresta — isto é, um valor que queremos maximizar em vez de minimizar. Assim, aprender a função Q equivale a resolver um problema de caminho ótimo com pesos esperados e futuros.

Para equilibrar exploração e exploração, o agente geralmente segue uma política ε-*greedy*: com probabilidade ε, ele escolhe uma ação aleatória (*exploração*); com probabilidade $1 - \varepsilon$, escolhe a ação com maior valor $Q(s, a)$ (*exploração do que já aprendeu*). Isso permite que o agente descubra boas rotas mesmo quando ainda não tem informações suficientes.

Imagine que você está observando alguém explorar uma cidade aleatoriamente, mas está tentando aprender o caminho mais rápido entre dois pontos com base no que essa pessoa vê. Você não seguirá os passos dela no futuro mas está aprendendo para agir como ela.

5 Deep Q-Learning

O algoritmo **Deep Q-Learning** (ou *Deep Q-Network* — DQN) estende o Q-Learning clássico para problemas com espaços de estados muito grandes ou contínuos, onde é inviável manter uma tabela $Q(s, a)$. A principal ideia é utilizar uma **rede neural** parametrizada por θ para aproximar a função ação-valor $Q(s, a; \theta)$.

Diferenças em relação ao Q-Learning clássico

- A tabela $Q(s, a)$ é substituída por uma **rede neural**.
- As atualizações são feitas com base em um **buffer de experiência** (experience replay) e em uma **rede-alvo** (target network) separada.

Atualização dos parâmetros

A rede é treinada para minimizar a diferença entre a predição atual $Q(s, a; \theta)$ e um alvo (target) computado com a rede-alvo:

$$y = r + \gamma \cdot \max_{a'} Q(s', a'; \theta^-)$$

$$\mathcal{L}(\theta) = \left(y - Q(s, a; \theta)\right)^2$$

em que θ^- são os parâmetros da **rede-alvo**, que são atualizados periodicamente para acompanhar θ, reduzindo instabilidades no aprendizado.

Mecanismos adicionais

- **Experience Replay:** transições (s, a, r, s') são armazenadas em um buffer e amostradas aleatoriamente para quebrar a correlação entre amostras consecutivas.
- **Rede-Alvo:** usada para calcular o target y. Seus parâmetros são copiados da rede principal a cada C passos.
- **Exploração ε-greedy:** a política de exploração escolhe uma ação aleatória com probabilidade ε, que decai ao longo do tempo.

Pseudocódigo do Deep Q-Learning

DEEP-Q-LEARNING($\alpha, \gamma, \varepsilon, C, N$, capacidade do buffer)

Inicialize a rede $Q(s, a; \theta)$ com pesos aleatórios
Inicialize a rede-alvo $Q(s, a; \theta^-) \leftarrow Q(s, a; \theta)$
Inicialize o buffer de experiência $\mathcal{D} \leftarrow \emptyset$
for episódio $\leftarrow 1$ **to** N
    Inicialize o estado s
    while s não é terminal
        Com probabilidade ε, escolha ação aleatória a
        Caso contrário, escolha $a = \arg\max_{a'} Q(s, a'; \theta)$
        Execute a, observe r e próximo estado s'
        Armazene (s, a, r, s') em $\mathcal{D}$
        Amostre um minibatch de transições $(s_i, a_i, r_i, s'_i) \sim \mathcal{D}$
        Para cada transição do minibatch:
          $y_i \leftarrow r_i$ se s'_i terminal, senão
          $y_i \leftarrow r_i + \gamma \cdot \max_{a'} Q(s'_i, a'; \theta^-)$
        Atualize θ minimizando a perda:
          $\mathcal{L}(\theta) = \frac{1}{|\mathcal{B}|} \sum_i (y_i - Q(s_i, a_i; \theta))^2$
        A cada C passos, atualize $\theta^- \leftarrow \theta$
        $s \leftarrow s'$

O Deep Q-Learning combina reforço com redes neurais profundas, tornando possível aprender diretamente de observações brutas (como imagens). O uso de memória de replay e de uma rede-alvo ajuda a estabilizar e acelerar o treinamento. O agente aprende a maximizar o retorno esperado mesmo em ambientes de alta complexidade.