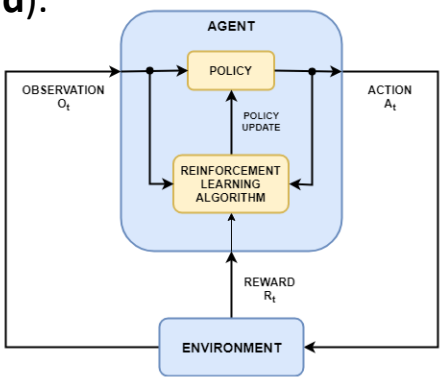
13
Aprendizagem por Reforço

CPereira IC, 22/23

Princípios

- O agente observa o ambiente (**estado**), toma uma decisão (**ação**) e recebe um reforço (**reward**).
  - **Objetivo**: Otimizar "reward".



- *https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html*

Princípios

- História
  - 1950, jogos, sistemas de controlo, …
  - 2013, DeepMind
    - Os agentes aprenderam a jogar “Atari games” do zero, superando os humanos! Nenhum conhecimento prévio!
    - Estado = Imagem representada em “pixels brutos”

» 2014, DeepMind was bought by Google (over 500 million dollars!)

Princípios

- …
  - 2016
    - AlphaGo, vitória contra o campeão mundial!

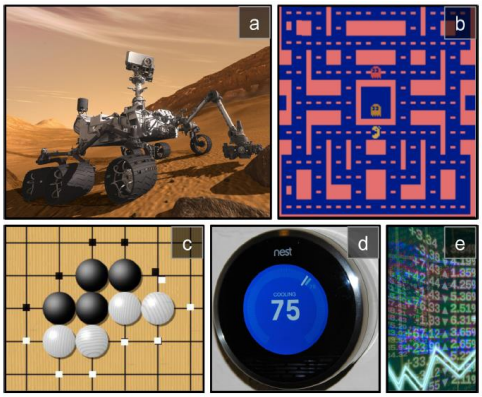
– https://www.deepmind.com/research/highlighted-research/alphago

Princípios

- Como representar o problema?
  - "Walking robot"
    - Ambiente – mundo real
    - Ações – andar em direções definidas
    - Reward – reforço ou recompensa
      - Positiva: aproxima-se do alvo
      - Negativa: vai na direção errada, perde tempo, cai, …

Princípios

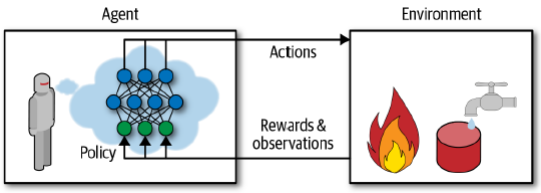
- …
  - Controlador de temperatura da água
    - Ações: Sinal de atuação (caudal de água quente)
    - reforço
      - positivo: Aproxima-se da temperatura alvo;
      - negativo: Intervenção humana,..
  - Mercado de ações
    - Ações: Comprar ou vender (a cada segundo)
    - reforço: ganhos monetários



[1]

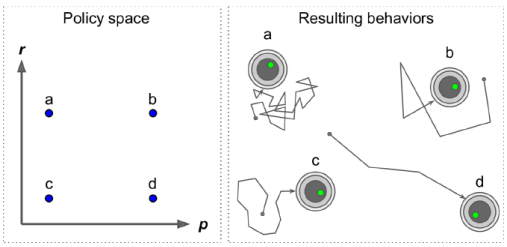
Políticas

- Política (Policy)
  - O que é
    - Algoritmo usado para determinar as ações
  - Implementação da política:
    - Regras lógicas (boolean ou fuzzy)
    - Redes neuronais
    - Qualquer outro algoritmo... determinista ou probabilístico



Políticas

- Exemplo
  - Robot Aspirador (Open AI Gym)
    - Reward: quantidade de pó que recolhe a cada 30 minutos
    - Ações (a cada segundo) – "política estocástica";
      - Mover em frente
        - » Com probabiliadade p
      - Rodar
        - » Ângulo aleatório [-r +r]
        - » com probabilidade (1-p)
    - Parâmetros de política (p, r)



Políticas

- ...
  - **Como determinar os parâmetros p e r?**
    - Força bruta – experimentar muitas combinações e escolher a melhor!
    - Algoritmo **genético**
      - 100 políticas aleatórias iniciais – individual=(p,r)
      - Selecione os 20 melhores indivíduos
      - Cada indivíduo produz 4 descendentes (cópia do pai mais alguma variação aleatória)
      - Iterar até obter uma política "boa"
    - **Gradiente** do reforço em relação aos parâmetros.
      - Se aumentarmos "p" aumenta a recompensa?

Políticas - Exercício

- Exemplo - OpenAI Gym
  - https://github.com/openai/gym
  - Permite simular ambientes

    >>Pip install –upgrade gym

    >>import gym
  - Vamos considerar um modelo 2D, o bem conhecido "cart-pole"

    >>> env = gym.make("CartPole-v0")

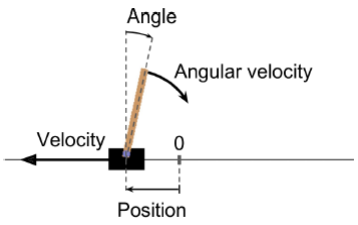
    >>> obs = env.reset()

    >>> obs

    array([-0.03799846, -0.03288115, 0.02337094, 0.00720711])

    >>> env.render()

```
In [1]: import numpy as np
   ...: import gym
   ...: env = gym.make("CartPole-v1")
   ...: obs = env.reset()
```



Name	Type	Size	Value
obs	float64	(4,)	[-0.04702661 -0.03513292 0.02138388 -0.04226916]

```
#cart's horizontal position (0.0 = center),
#its velocity (positive means right),
#the angle of the pole (0.0 = vertical),
#and its angular velocity (positive means clockwise).
```

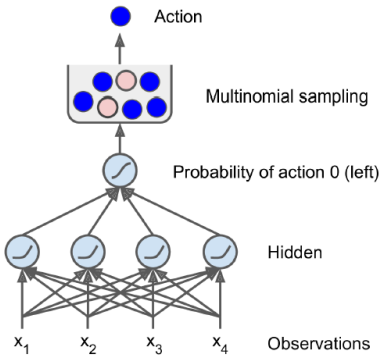
Políticas - Exercício

- …
  - Uma política básica:
    - acelera para a esquerda quando o pêndulo está inclinado para a esquerda
    - acelera para a direita quando o pêndulo está inclinado para a direita.
  - Funciona?
    - Executamos esta política, observando a média de recompensas que recebe, para 500 episódios.
    - Nunca conseguiu equilibrar o pêndulo por mais de 72 iterações consecutivas!!

```
In [15]: def basic_policy(obs):
    ...:     angle = obs[2]
    ...:     return 0 if angle < 0 else 1
    ...:
    ...:
    ...: totals=[]
    ...: for episode in range(500):
    ...:     episode_rewards=0
    ...:     obs=env.reset()
    ...:     for step in range(200):
    ...:         action=basic_policy(obs)
    ...:
    ...:         # visualize each step
    ...:         #env.render()
    ...:
    ...:         obs,reward,done,info=env.step(action)
    ...:         episode_rewards+=reward
    ...:         if done:
    ...:             break
    ...:     totals.append(episode_rewards)
    ...:
    ...: np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
Out[15]: (41.17, 8.529073806692026, 24.0, 72.0)
```

Políticas - Exercício

- …
  - Como aprender a política com uma rede neuronal?
    - Input: observação
      - posição, velocidade, angulo, velocidade angular
    - Output: ação a executar
      - "esq" ou "di"r (random probabilistic)
  - Como treinar a rede?
    - Não conhecemos a melhor ação para cada observação, assim não podemos usar supervisão!
    - "Credit assignment"



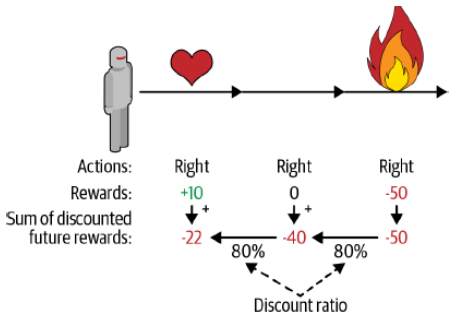

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

Políticas - Exercício

- ...
  - Treino da rede
    - "Credit assignment"
      - Avaliar um Acão com base na soma de todos os "rewards" subsequentes, aplicando "*discount rates* - r".
      - Um *discount rate* perto de zero faz com que apenas rewards imediatos sejam considerados.



Políticas - Exercício

- ...
  - Os fatores de desconto típicos variam entre 0.9 e 0.99.
    - Com um fator de desconto de **0.95**, 13 passos no futuro contam aproximadamente metade de recompensas imediatas ($0.95^{13} \approx 0,5$).
    - Com um fator de desconto de 0.99, 69 passos para o futuro valem metade das recompensas imediatas ($0.99^{69} \approx 0,5$).
    - No ambiente "Cart-Pole", as ações têm efeitos de curto prazo, assim a escolha de um fator de desconto de 0.95 parece razoável.

Políticas - Exercício

- ...
  - **Policy Gradient**
    1. Executar a rede neuronal por vários episódios, e em cada iteração calcular os gradientes que tornariam a ação tomada mais provável.
       1. Probabilidade=1 para ação “left” and p=0 se ação “rigth”
       2. Output=0.8; ação=left; erro=1-0.8=0.2
       3. Não aplicar estes gradientes, por agora!!
    2. Depois de cada episódio, calcular o reforço por ação.
       1. Usando a metodologia anterior, “credit assignment”
    3. Multiplicar o gradiente pelo reforço.
       1. Se reforço positivo, aplica-se o gradiente calculado anteriormente para fazer com que a ação seja mais provável!
       2. Se reforço negativo, aplica-se o gradiente oposto.
    4. Calcular a média de todos os vetores gradiente e executar um passo do algoritmo “Gradient Descent”

Políticas: “Policy Gradient”

- ...
  - O algoritmo de gradientes de política simples que acabamos de treinar resolveu a tarefa Cart-Pole, mas não escalaria bem para tarefas mais complexas.
  - Altamente ineficiente, pois precisa de explorar o ambiente por muito tempo antes de pode fazer progressos significativos, pois deve executar vários episódios para estimar a vantagem de cada ação.
  - No entanto pode ser combinado com outras técnicas:
    - AlphaGo baseou-se neste algoritmo, combinado com “Monte Carlo Tree Search”
  - Baseia-se no princípio:
    - Otimizar a política para aumentar os rewards.
- Princípo alternativo:
  - O agente tenta **estimar a soma esperada dos reforços** (com taxa de desconto) **para cada acção em cada estado**!

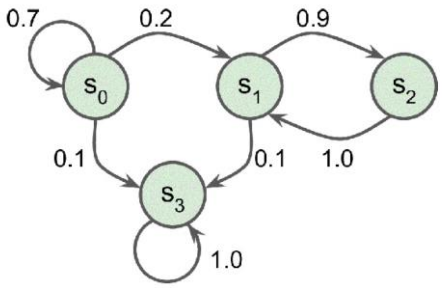
MDP – Markov Decision Processes

- …
- Princípo alternativo:
  - O agente tenta estimar a soma esperada dos reforços (com taxa de desconto) para cada acção em cada estado!
  - Como efetuar este cálculo?

- Cadeias de Markov
  - Processo estocástico - tendo uma distribuição de probabilidade aleatória ou padrão que pode ser analisado estatisticamente, mas não pode ser previsto com precisão.
  - Esteprocesso tem um número fixo de estados e evolui aleatoriamente de um estado para outro a cada passo.
  - A probabilidade de ele evoluir de um estado s para um estado s' é fixa, e depende apenas do par (s,s'), não de estados passados (o sistema não possui memória).

Processos de Markov

- ….
  - Exemplo de uma cadeia de Markov

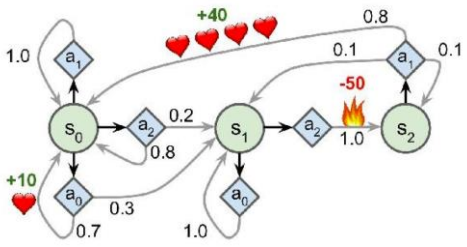


Processos de Markov

- ...
  - Processos de decisão de Markov (MDP)
    - similar a uma cadeia mas com as seguintes diferenças:
      - A cada passo **o agente pode escolher uma acção**
      - As probabilidades de transição dependem da ação tomada
      - Algumas transições oferecem reforço (positivo ou negativo)
    - Objetivo do agente:
      - Maximizar o reforço ao longo do tempo!
      - Política Ótima – recompensa acumulada máxima.

Processos de Markov

- Exemplo:
  - 3 estados e até 3 acções em cada estado.
    - Qual a melhor estratégia para otimizar a recompensa ao longo do tempo?



Value Iteration Algorithm

- ...
  - Bellman Optimality Equation
    - Calcula V*(s) - a soma de todas as recompensas futuras descontadas que o agente pode esperar em média, após atingir um estado s, supondo que ele atua de maneira ótima

Probabilidade de transição do estado s para s', considerando que o agente escolhe a ação a

$$V^*(s) = \max_a \Sigma_s T(s,a,s')\left[R(s,a,s') + \gamma \cdot V^*(s')\right] \quad \text{for all } s$$

Recompensa que o agente recebe a transitar do estado "s" para "s'", escolhendo a ação "a"

Taxa de desconto

Value Iteration Algorithm

- ...
  - A equação de Bellman conduz diretamente a um algoritmo que pode estimar o valor ótimo de cada estado possível:
    - Inicializa todas as estimativas do valor de estado a zero
    - Atualiza iterativamente o valor - *Value Iteration*:

Valor na iteração k para **s'**

Valor estimado na iteração (k+1) para **s**

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot V_k(s')\right] \quad \text{for all } s$$

    - Dado tempo suficiente, as estimativas **convergem** para os valores ótimos, correspondendo à política ótima .

Q-Value Iteration Algorithm

- ...
  - Conhecer os valores ótimos para cada estado é bastante útil, em particular para avaliar uma política, mas não informa explicitamente ao agente o que fazer - qual a ação a tomar?
    - Existe um algoritmo semelhante para estimar os valores de ação de estado ótimos, geralmente designados de Q-Values:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s'a)$$

Valor para o par **(s,a)** na iteração (k+1):
Representa a soma das recompensas futuras descontadas que o agente pode esperar depois de **atingir o estado s e escolher a ação a**, mas antes de ver o resultado da sua ação, supondo que atua de forma ótima após essa ação!

Q-Value Iteration Algorithm

- ...
  - Implementação
    - Quando o agente está no estado s0 e escolhe a ação a1, a soma esperada de recompensas futuras descontadas é de aproximadamente 17,02
    - Politica ótima <u>para desconto de 0.90</u>:
      - No estado s0 escolha a ação a0;
      - em s1, escolha a ação a0 (ou seja, fique parado);
      - em s2 escolha a ação a1 (a única ação possível).

```
transition_probabilities = [ # shape=[s, a, s']
[[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
[[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
[None, [0.8, 0.1, 0.1], None]]

rewards = [ # shape=[s, a, s']
[[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0], [0, 0, -50]],
[[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]

possible_actions = [[0, 1, 2], [0, 2], [1]]

Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions

for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # for all possible actions

gamma = 0.90 # the discount factor
for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    *(rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])

Q_values
```

```
In [21]: Q_values
Out[21]:
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,         -inf, -4.87971488],
       [       -inf, 50.13365013,         -inf]])

In [22]: np.argmax(Q_values, axis=1)
Out[22]: array([0, 0, 1], dtype=int64)
```

Temporal Difference Learning

- …
  - Observações:
    - Os problemas de Aprendizagem por Reforço com ações discretas podem ser representados como Processos de decisão de Markov,
    - **Contudo,** o agente inicialmente não conhece:
      - As probabilidades de transição (não sabe a priori T(s, a, s'))
      - As recompensas (não conhece a priori R(s, a, s')).
  - Solução
    - O agente deve "experimentar" cada estado e cada transição:
      - pelo menos uma vez para conhecer as recompensas;
      - várias vezes para obter uma estimativa razoável das probabilidades de transição.

Temporal Difference Learning

- …
  - O algoritmo (TD Learning) é semelhante ao "Value Iteration", mas ajustado para levar em conta o fato de que o agente apenas possui conhecimento parcial do MDP.
    - O agente inicialmente conhece apenas os possíveis estados e ações.
    - O agente usa uma política de **exploration** - por exemplo, uma política puramente aleatória - para "explorar" o MDP e atualiza as estimativas dos valores de estado com base nas transições e recompensas realmente observadas.
    - Algoritmo do "tipo" gradiente – converge caso se reduza gradualmente o "learning rate".

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

with $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

Q-Learning

- …
  - algoritmo Q-Learning
    - é uma adaptação do "Q-Value Iteration" quando as probabilidades de transição e as recompensas são inicialmente desconhecidas:
      - Observa um agente a "jogar" (por exemplo, aleatoriamente) e melhora gradualmente as suas estimativas dos valores Q.
      - Depois de obter estimativas precisas (ou próximas), a política ideal é escolher a ação com o maior Q-Value (a política "greedy"):

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s',a')$$

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward

def exploration_policy(state):
    return np.random.choice(possible_actions[state])


alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

Q-Learning

- "Exploration vs Exploitation"
  - O algoritmo só pode funcionar se explorar suficientemente o problema MDP.
    - Embora uma política puramente aleatória garanta a visita "eventualmente frequente" a todos os estados e transições, conduz a um tempo extremamente longo para problemas de alguma complexidade!
    - Assim, opta-se por uma politica designada de **ε-greedy**:
      - » a cada passo age aleatoriamente com probabilidade ε, ou avidamente (i.e., escolhendo a ação com o valor-Q mais alto) com probabilidade 1–ε.
      - » Vantagem
        - comparada a uma política completamente aleatória, é que dedica cada vez mais tempo a "visitar" as partes mais interessantes do ambiente (**exploitation**), à medida que as estimativas melhoram, sem deixar de "visitar" as regiões desconhecidas do problema (**exploration**).
      - » Deve começar-se com um valor alto para ε (perto de 1.0) e depois reduzi-lo gradualmente (por exemplo, para 0.05).

Q-Learning

- …
  - O algoritmo Q-Learning é designado de “off-policy” porque a política que está a ser treinada não é necessariamente aquela que está a ser executada
    - no exemplo do código anterior, a política executada (a política de exploração) é aleatória, enquanto a política treinada baseia-se na escolha das ações com os maiores valores de Q!
  - Funções de “exploração”:
    - para experimentar ações que não tentou antes!
      - » implementado através de um bónus adicionado às estimativas de Q-Value.

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} f(Q(s',a'), N(s',a'))$$

Número de vezes que a ação “a” foi escolhida no estado “s’”

Função de exploração, por exemplo *f(Q,N)=Q+k(1+N)*, onde o híper-parâmetro k define a “curiosidade” – grau de atração para o desconhecido!

Q-Learning

- …
  - Algoritmos de aproximação
    - Q-Learning - não é possível escalar para problemas de média ou elevada dimensão - com muitos estados e ações.
    - Por exemplo – como treinar um agente para jogar Pac-Man?
      - Existem cerca de 150 pellets que Pac-Man pode comer, cada um dos quais pode estar presente ou ausente (ou seja, já comido) – assim o número de estados possíveis é superior a 2e150. E se adicionar todas as combinações possíveis de posições para todos os fantasmas, o número de estados possíveis torna-se maior que o número de átomos no planeta Terra,
      - Não há nenhuma forma de gerir uma tabela de Q-Values.
    - Solução:
      - Encontrar uma função $Q_\theta\ (s,a)$ que aproxime os Q-values, mantendo um número razoável de parâmetros (θ)!

Deep Q-Learning

- ...
  - Uma rede DNN para estimar valores Q é designada de rede Q profunda (DQN)
    - A aprendizagem com DQNs é designada de "Deep Q-Learning".
  - Como podemos treinar um DQN?
    - Considerando o valor Q aproximado calculado pelo DQN para um determinado par estado-ação (s, a).
    - Este valor deve estar o mais próximo possível da recompensa r observada depois de experimentar a ação a no estado s, mais o valor descontado de decidir de forma otimizada a partir de então.

$$Q_{\text{target}}\left(s, a\right) = r + \gamma \cdot \max_{a'} \; Q_{\theta}\left(s', a'\right)$$

Deep Q-Learning - exemplo

- Implementação – Cart-Pole
  - https://github.com/ageron/handson-ml
  - Em teoria, precisamos de uma rede neuronal que recebe um par "estado-ação" e gera um Q-Value aproximado.
  - Na prática é mais eficiente usar uma rede que **recebe um estado** e gera um valor aproximado "Q-Value" para cada ação possível!

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n
model = keras.models.Sequential([
keras.layers.Dense(32, activation="elu", input_shape=input_shape),
keras.layers.Dense(32, activation="elu"),
keras.layers.Dense(n_outputs)
])
```

Deep Q-Learning - exemplo

- ...
  - Para selecionar uma ação, escolhemos a ação com o maior "Qvalue" previsto pela rede. Mas, para garantir que o agente explore o ambiente, usamos uma política "ε-greedy" (ou seja, vamos escolher uma ação aleatória com probabilidade ε, ou greedy caso contrário)

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Deep Q-Learning - exemplo

- ...
  - Em vez de treinar a rede com base apenas nas experiências mais recentes, armazenamos todas as experiências numa memória de "replay", e posteriormente criamos uma amostra, aleatoriamente, em cada iteração de treino.
    - Ajuda a reduzir as correlações entre as experiências num batch de treino!
    - Cada experiência é composta por cinco elementos: estado, a ação que o agente realizou, a recompensa resultante, o próximo estado alcançado e, finalmente, uma variável booleana indicando se o episódio terminou naquele ponto (concluído).
    - Precisamos de uma função para criar uma amostra aleatória de "experiências" a partir do buffer de "replay".

```
from collections import deque
replay_buffer = deque(maxlen=2000)

def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
            np.array([experience[field_index] for experience in batch])
            for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

Deep Q-Learning - exemplo

- ...
  - Necessitamos ainda:
    - uma função que executa uma única etapa usando a política ε-greedy, e armazena a experiência resultante no buffer de repetição.

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

Deep Q-Learning - exemplo

- ...
  - uma função que irá criar a amostra (batch) de experiências do buffer e treine a rede DQN executando um passo do “gradient descent” neste batch.:

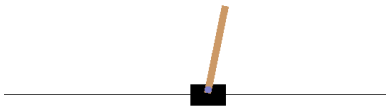
```
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards + (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Deep Q-Learning - exemplo

- ...
  - Treino do modelo

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```



Deep Q-Learning - exemplo

- ...
  - Outra variante…Deep SARSA
    - Existem duas políticas: de comportamento do agente e aprendizagem.
      - A política de comportamento é utilizada para gerar ações At+1; a política de aprendizagem é o que o agente aprende por meio de tais interações para atualizar Q. Na SARSA as duas políticas são iguais, sendo designado de método “on-policy”.
        - » $Q(s_t, a_t) = Q(s_t, a_t) + \alpha*(r_t + \gamma*Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$
    - Implementação de um agente SARSA com biblioteca Keras-rl
      - https://keras-rl.readthedocs.io/en/latest/agents/sarsa/
      - https://www.mathworks.com/help/reinforcement-learning/ug/sarsa-agents.html

Referências

- ...
  - [1] Hands on Machine Learning with scikit learn and tensorflow, Aurelien Geron, O'Reilly, 2017.
  - https://github.com/ageron/handson-ml
  - https://www.mathworks.com/help/reinforcement-learning