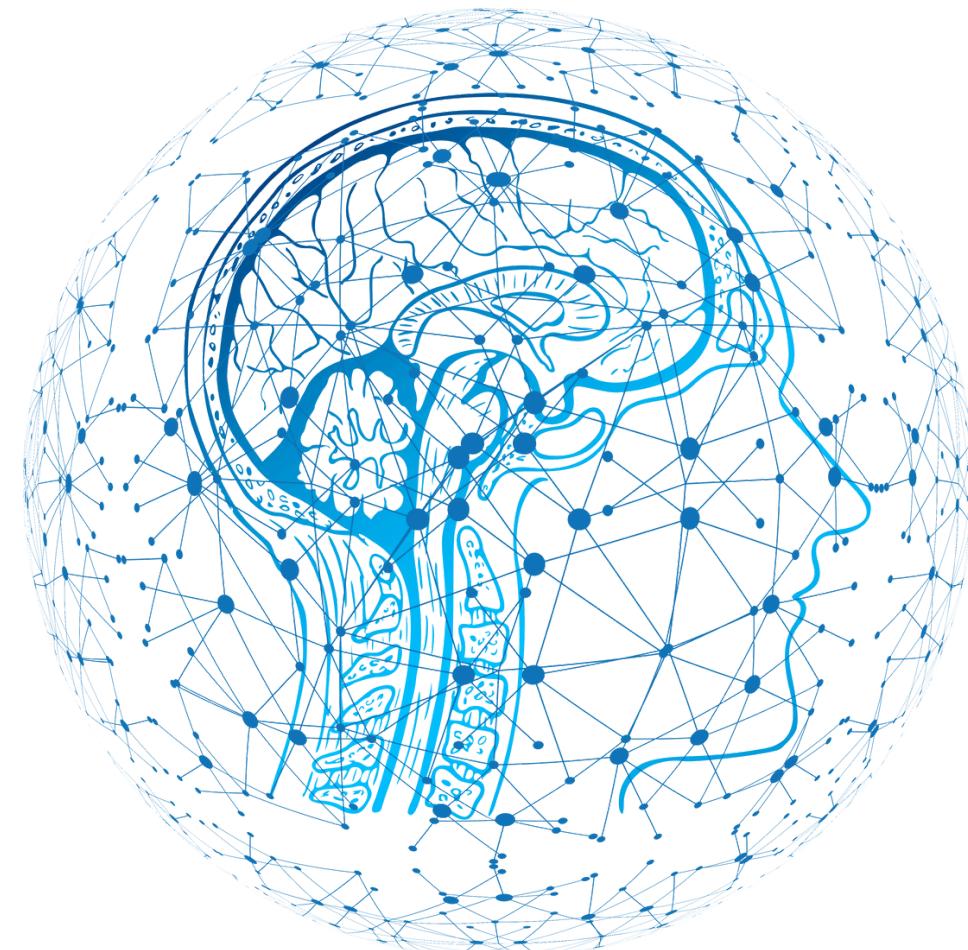




# Aprendizado por Reforço

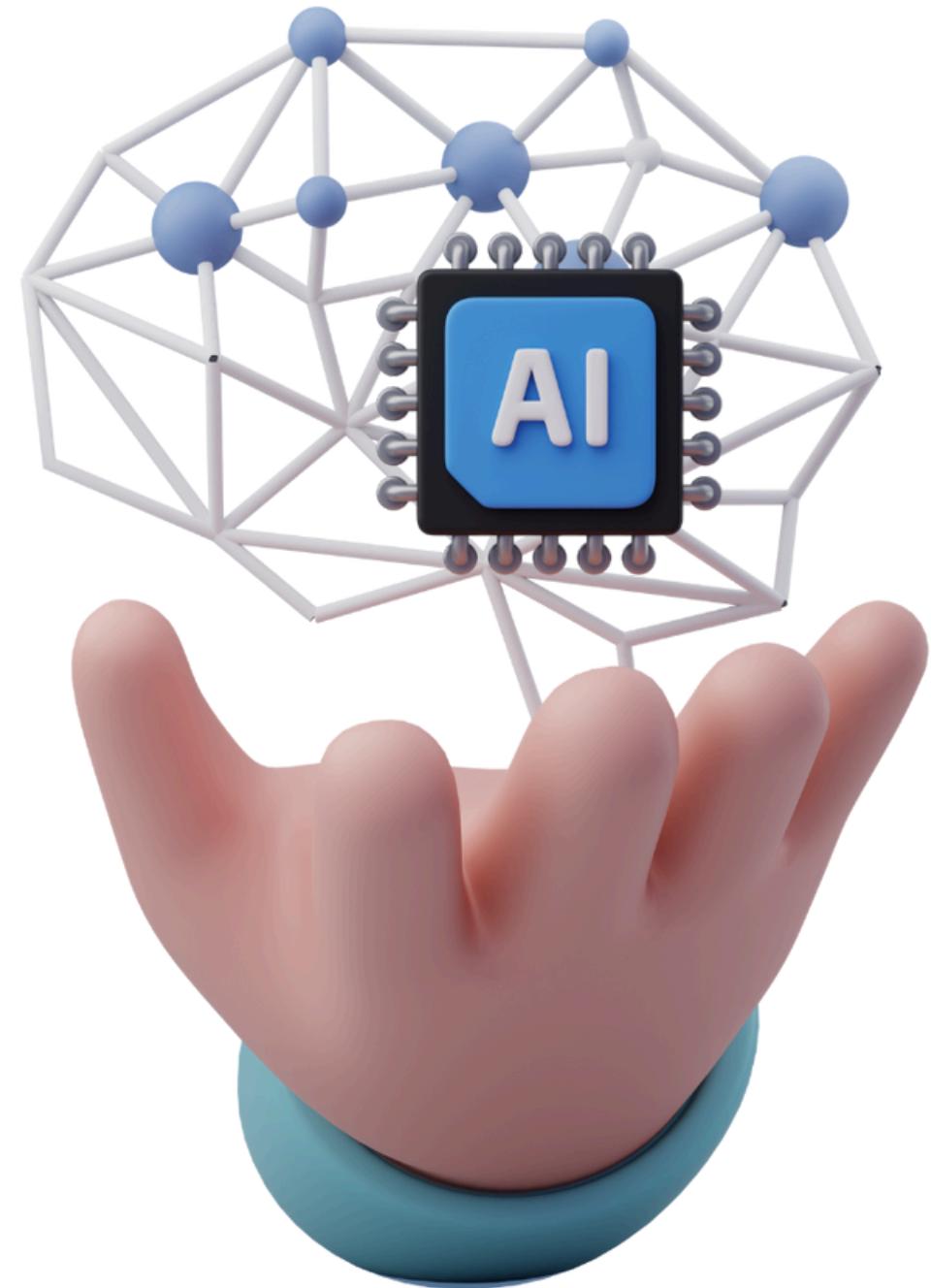
## Deep Q - Networks (DQN)





# Conceitos gerais

- Aprendizado Supervisionado x Aprendizado Não Supervisionado x Aprendizado por Reforço
- Deep Q-Network(DQN) é um algoritmo que combina redes neurais com Q-Learning, permitindo que “agentes” aprendam situações otimizadas em ambientes complexos.

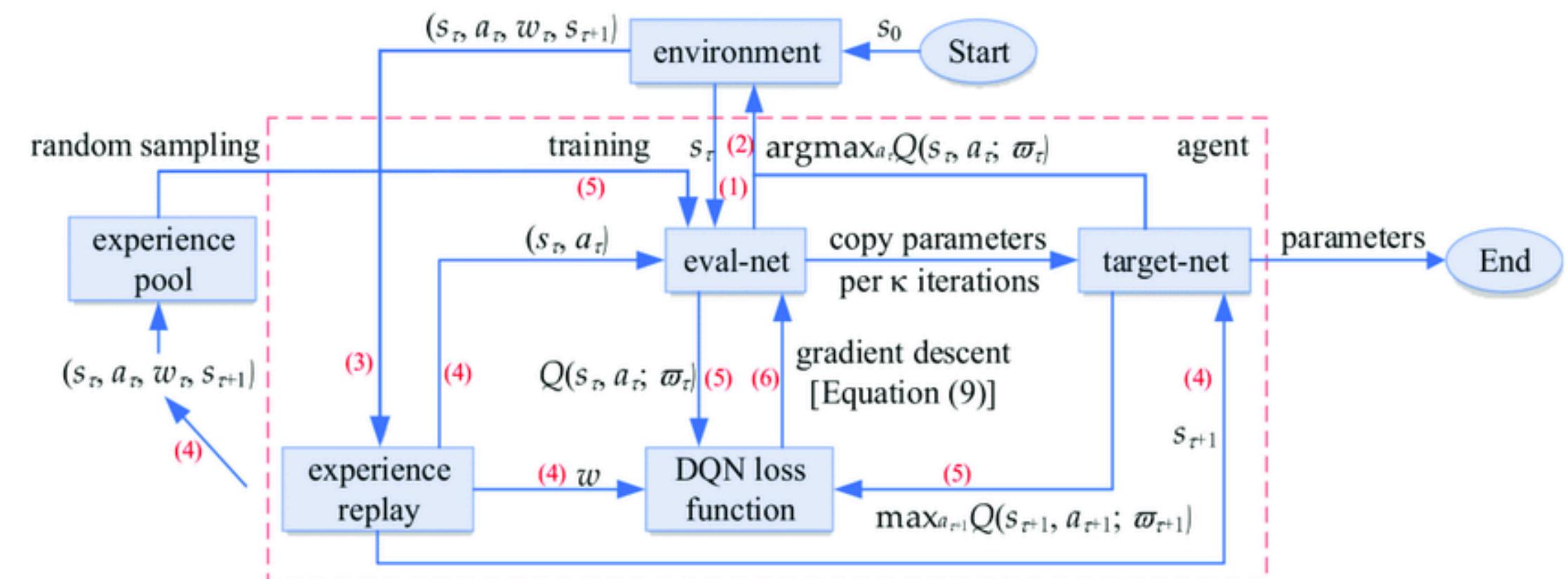




# Algoritmo passo a passo

O algoritmo DQN segue uma abordagem baseada em redes neurais profundas para aprender e otimizar funções de valor de ação. O processo de trabalho pode ser resumido da seguinte forma:

1. Representação de Estado;
2. Arquitetura de Rede Neural;
3. Repetição da experiência;
4. Atualização do Q-Learning;
5. Exploração e Exploração;
6. Rede Alvo;
7. Repita as etapas 1 a 6.





# Frozen Lake Environment

01

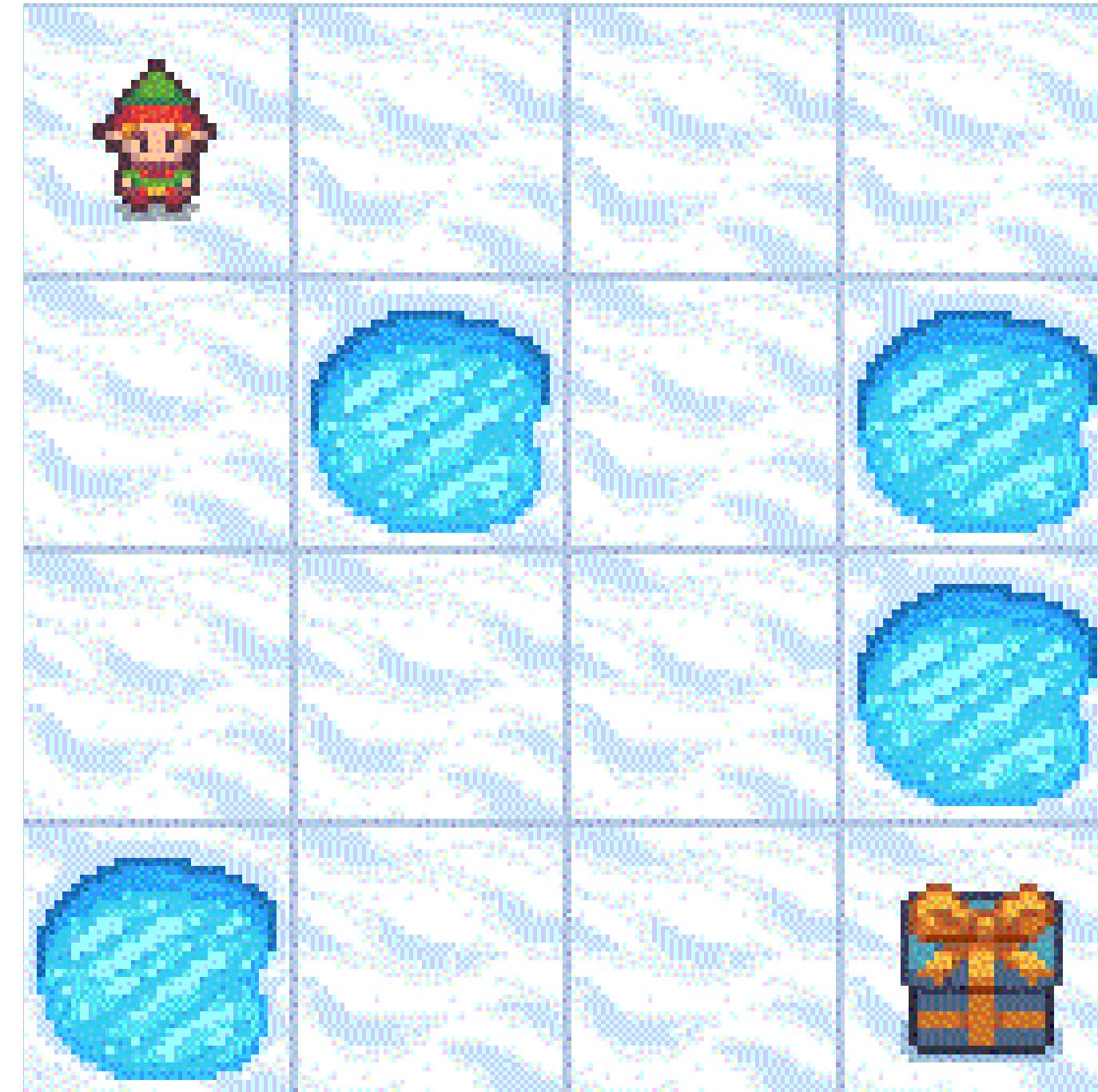
## Estado

Cada posição na grade pode ser representada de várias formas, como one-hot encoding da posição atual ou mesmo uma imagem da grade.

02

## Ações

O agente pode realizar quatro ações: mover para cima, baixo, esquerda ou direita.





# Frozen Lake Environment

03

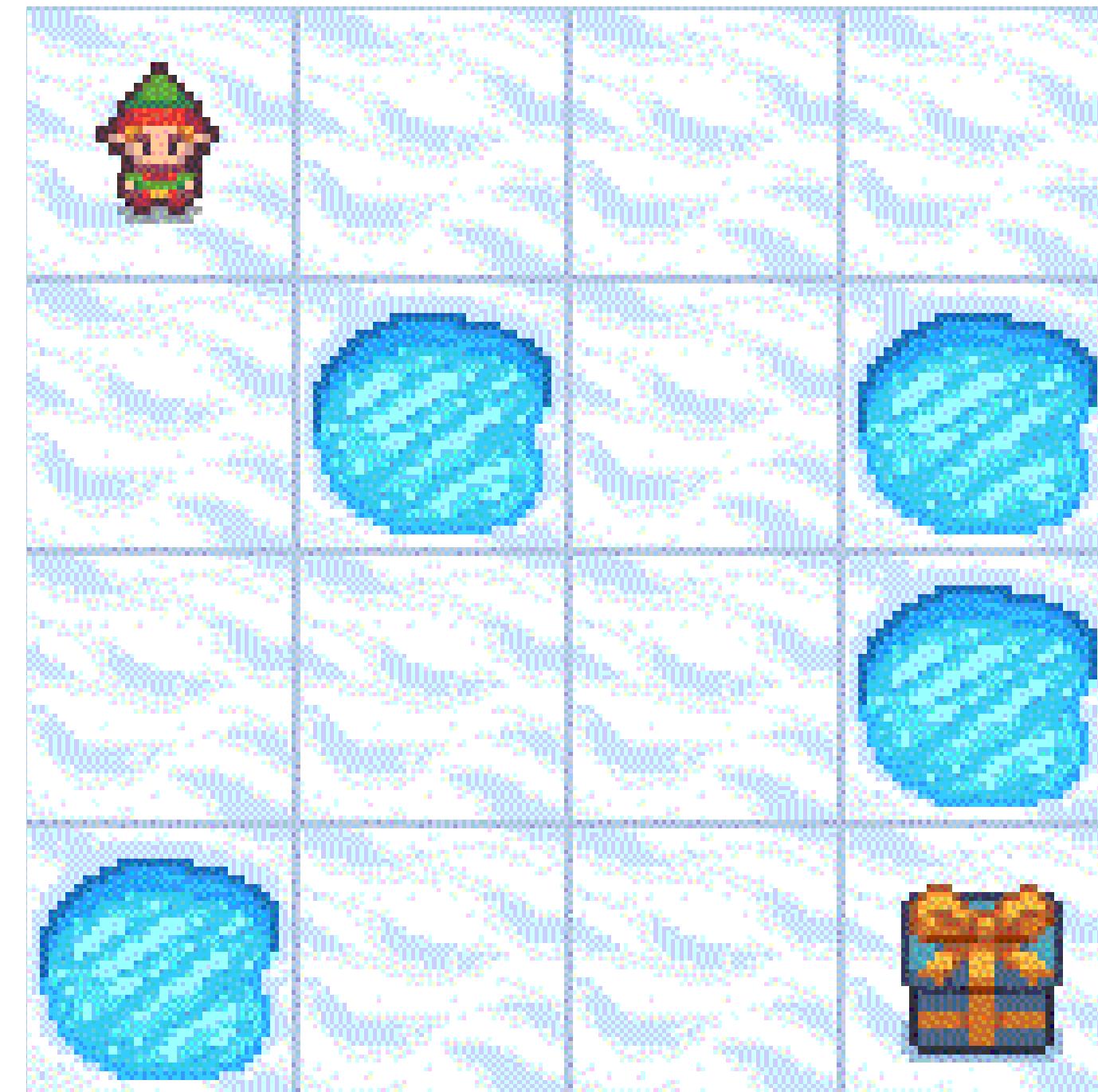
## Função de Recompensa

- Se o agente alcança o objetivo, recebe uma recompensa positiva (geralmente 1).
- Se cai em um buraco, recebe uma recompensa de zero ou negativa.

04

## Função de Recompensa

A tabela  $Q(s,a)$  armazena o valor de cada ação  $a$  para cada estado  $s$ . Inicialmente, essa tabela é preenchida com valores zero, pois o agente ainda não aprendeu nada sobre o ambiente.





# Frozen Lake Environment

05

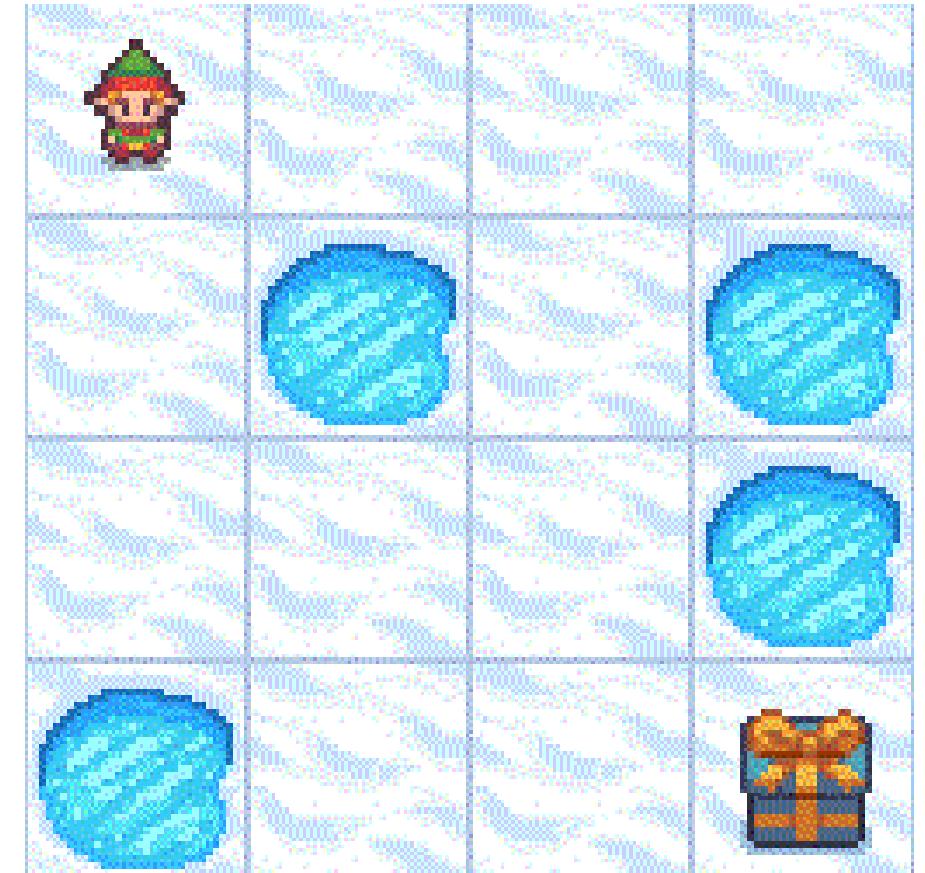
## Fórmula de Atualização do Q-Learning

$$Q(state,action) \leftarrow Q(state,action) + learning\ rate \cdot (reward + discount\_factor * maxQ(new\_state) - Q(state,action))$$

06

## Exploration vs Exploitation (Epsilon-Greedy)

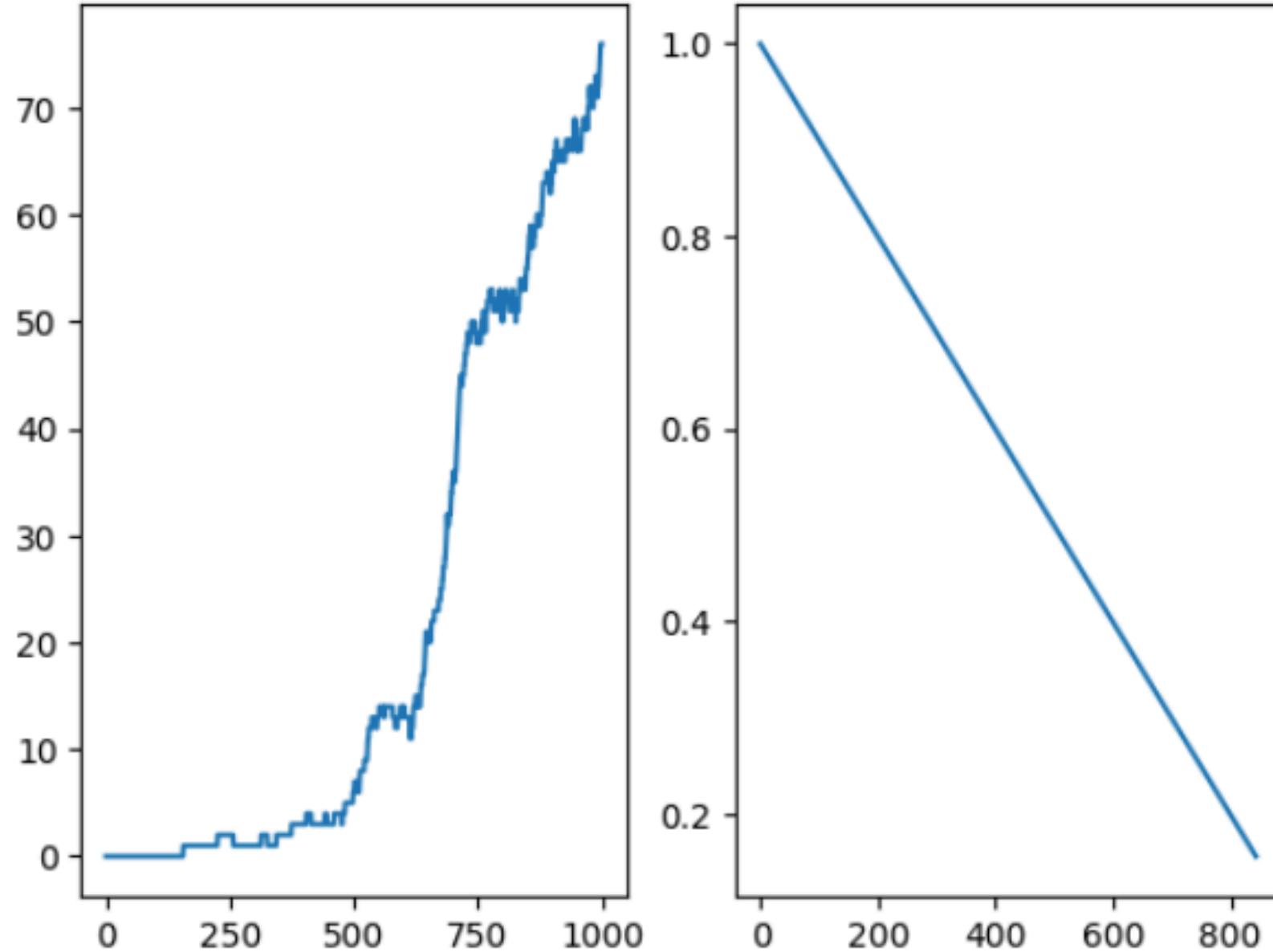
- No  $\epsilon$ -greedy, usamos uma variável  $\epsilon$ (epsilon), que controla a probabilidade de o agente explorar uma ação aleatória em vez de explorar a melhor ação conhecida.
- Com probabilidade  $\epsilon$ , o agente escolhe uma ação aleatória (exploração).
- Com probabilidade  $1-\epsilon$ , o agente escolhe a melhor ação conhecida (exploração), baseada nos valores Q que ele já aprendeu.



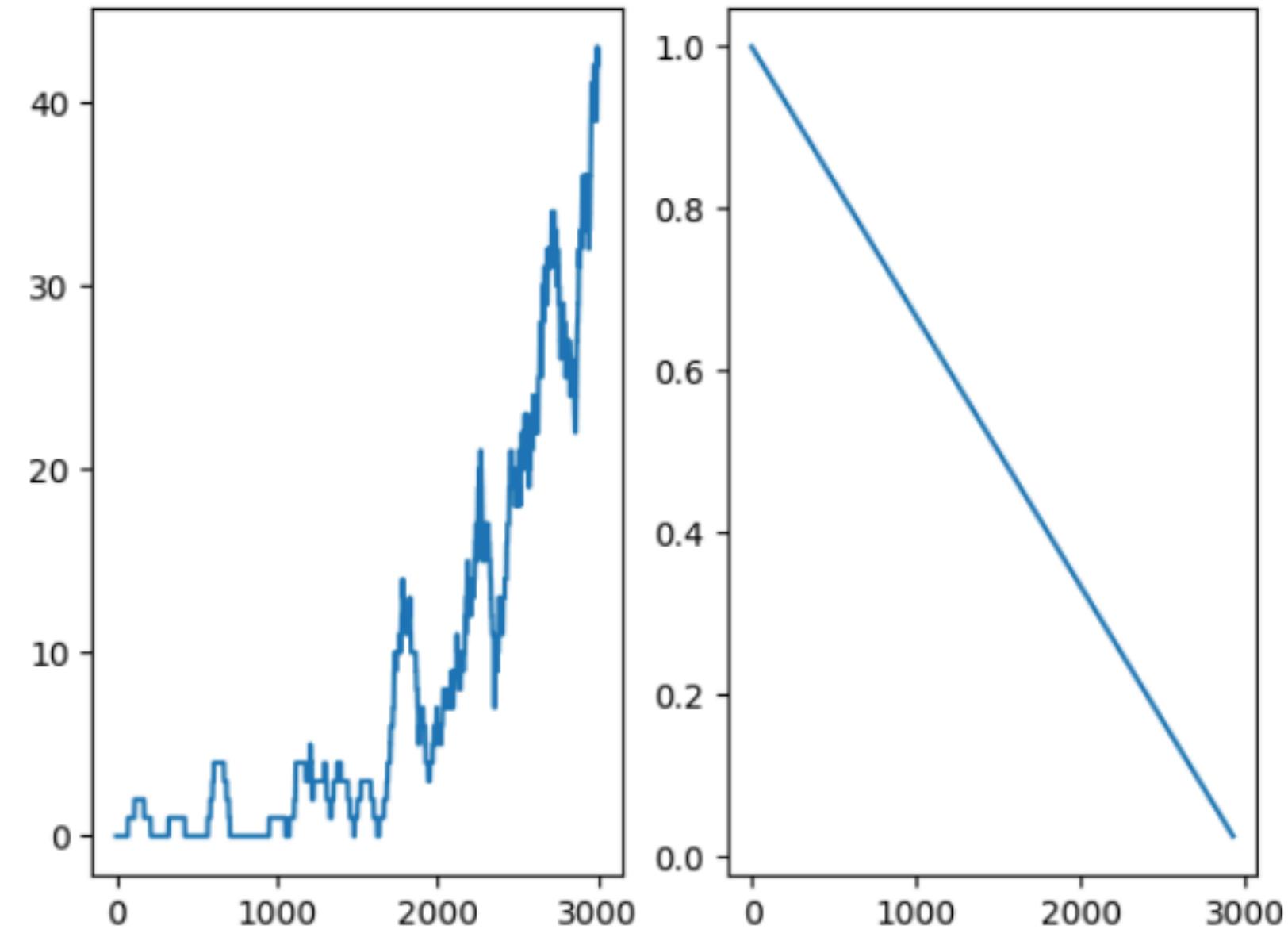


# Gráficos de teste

**is\_slippery = false**



**is\_slippery = true**





# Funções principais com Frozen Lake

```
● ● ●

# Define model
class DQN(nn.Module):
    def __init__(self, in_states, h1_nodes, out_actions):
        super().__init__()

        # Define network layers
        self.fc1 = nn.Linear(in_states, h1_nodes) # first fully connected layer
        self.out = nn.Linear(h1_nodes, out_actions) # ouptut layer w

    def forward(self, x):
        x = F.relu(self.fc1(x)) # Apply rectified linear unit (ReLU) activation
        x = self.out(x)          # Calculate output
        return x

# Define memory for Experience Replay
class ReplayMemory():
    def __init__(self, maxlen):
        self.memory = deque([], maxlen=maxlen)

    def append(self, transition):
        self.memory.append(transition)

    def sample(self, sample_size):
        return random.sample(self.memory, sample_size)

    def __len__(self):
        return len(self.memory)
```

## Classe de inicialização



# Funções principais com Frozen Lake

```
● ● ●

def state_to_dqn_input(self, state:int, num_states:int)->torch.Tensor:
    input_tensor = torch.zeros(num_states)
    input_tensor[state] = 1
    return input_tensor

# Run the FrozeLake environment with the learned policy
def test(self, episodes, is_slippery=False):
    # Create FrozenLake instance
    env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=is_slippery, render_mode='human')
    num_states = env.observation_space.n
    num_actions = env.action_space.n

    # Load learned policy
    policy_dqn = DQN(in_states=num_states, h1_nodes=num_states, out_actions=num_actions)
    policy_dqn.load_state_dict(torch.load("frozen_lake_dql.pt"))
    policy_dqn.eval()    # switch model to evaluation mode

    print('Policy (trained):')
    self.print_dqn(policy_dqn)

    for i in range(episodes):
        state = env.reset()[0] # Initialize to state 0
        terminated = False      # True when agent falls in hole or reached goal
        truncated = False       # True when agent takes more than 200 actions

        # Agent navigates map until it falls into a hole (terminated), reaches goal (terminated),
        # or has taken 200 actions (truncated).
        while(not terminated and not truncated):
            # Select best action
            with torch.no_grad():
                action = policy_dqn(self.state_to_dqn_input(state, num_states)).argmax().item()

            # Execute action
            state,reward,terminated,truncated,_ = env.step(action)

    env.close()
```

## Funções do frozen lake



# Funções principais com Frozen Lake

```
● ● ●  
class FrozenLakeDQL():  
    def train(self, episodes, render=False, is_slippery=False):  
        for i in range(episodes):  
            state = env.reset()[0] # Initialize to state 0  
            terminated = False # True when agent falls in hole or reached goal  
            truncated = False # True when agent takes more than 200 actions  
  
            # Agent navigates map until it falls into hole/reaches goal (terminated), or has taken 200  
            actions (truncated).  
            while(not terminated and not truncated):  
  
                # Select action based on epsilon-greedy  
                if random.random() < epsilon:  
                    # select random action  
                    action = env.action_space.sample() # actions: 0=left,1=down,2=right,3=up  
                else:  
                    # select best action  
                    with torch.no_grad():  
                        action = policy_dqn(self.state_to_dqn_input(state, num_states)).argmax().item()  
  
                # Execute action  
                new_state,reward,terminated,truncated,_ = env.step(action)  
  
                # Save experience into memory  
                memory.append((state, action, new_state, reward, terminated))  
  
                # Move to the next state  
                state = new_state  
  
                # Increment step counter  
                step_count+=1
```

```
● ● ●  
# Keep track of the rewards collected per episode.  
if reward == 1:  
    rewards_per_episode[i] = 1  
  
# Check if enough experience has been collected and if at least 1 reward has been collected  
if len(memory)>self.mini_batch_size and np.sum(rewards_per_episode)>0:  
    mini_batch = memory.sample(self.mini_batch_size)  
    self.optimize(mini_batch, policy_dqn, target_dqn)  
  
# Decay epsilon  
epsilon = max(epsilon - 1/episodes, 0)  
epsilon_history.append(epsilon)  
  
# Copy policy network to target network after a certain number of steps  
if step_count > self.network_sync_rate:  
    target_dqn.load_state_dict(policy_dqn.state_dict())  
    step_count=0  
  
# Close environment  
env.close()
```

## Loop do Frozen Lake



# Funções principais com Frozen Lake

```
● ● ●  
frozen_lake = FrozenLakeDQL()  
is_slippery = False  
frozen_lake.train(1000, is_slippery=is_slippery)  
frozen_lake.test(10, is_slippery=is_slippery)
```

Chamada de treino e teste



# Vantagens

01

## Deep Representation Learning

Aprendizado abstrato e com uma alta dimensionalidade de problemas, resolvendo problemas complexos.

02

## Sample Efficiency

A utilização do método de “replay” e “target” ajuda a melhorar a eficiência de análise das amostras.

03

## Generalization

O algoritmo transforma as políticas já aprendidas em cada nó em um mapeamento mais abrangente que pode ser aplicado a novos estados semelhantes



# Limitações

01

## Hyperparameter Sensitivity

Os hiperparâmetros selecionados devem ser avaliados e testados com cuidado, pois o algoritmo pode terminar sem resultados, dependendo da escolha de valores.

02

## Lack of Continual Learning

O algoritmo funciona com os treinamentos feitos em lotes, a partir das experiências coletadas na função de replay.

03

## Overestimation of Action-Values

O algoritmo faz com que o agente sempre escolha ações que o caminho que parece ter um resultado maior do que realmente tem seja escolhido, podendo gerar erro nos cálculos e uma política mais imprecisa



# Referências bibliográficas

## Explicação do algoritmo Deep Q-Network e suas funcionalidades:

- <https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871>
- <https://paperswithcode.com/method/dqn>
- [https://pytorch.org/tutorials/intermediate/reinforcement\\_q\\_learning.html](https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html)
- <https://ai.plainenglish.io/deep-q-learning-understanding-the-math-behind-the-magic-9ab03d893baf>
- <https://www.youtube.com/watch?v=Bn6kIArpd3Q>
- <https://www.youtube.com/watch?v=5u8yNEN5arQ>
- <https://www.youtube.com/watch?v=JDDqP6IZ4NQ>
- <https://www.youtube.com/watch?v=SgC6AZss478>

## Vídeo explicativo metodologia com Frozen Lake:

- <https://www.youtube.com/watch?v=EUrWGTCGzIA>

## Código exemplo da aplicação de Deep Q-Network com Frozen Lake:

- [https://github.com/johnnycode8/gym\\_solutions/blob/main/frozen\\_lake\\_dql.py](https://github.com/johnnycode8/gym_solutions/blob/main/frozen_lake_dql.py)



# Obrigado!

Alexandre Augusto Niess Ferreira  
André Mendes Rodrigues  
Arthur Martinho Medeiros Oliveira  
Caio Gomes Alcântara Glória  
Daniel Salgado Magalhães  
Rafael Maluf Araújo

Professora: Cristiane Neri Nobre