OpenAI Gym: Desenvolver e Comparar Algoritmos de Aprendizagem por Reforço

Airton Tavares[*], Bernardo Sousa[**]

[*] Universidade dos Açores – Informática
[**] Universidade dos Açores – Informática

***Abstract*-** *OpenAI Gym* é uma *framework* que fornece a capacidade de desenvolver e comparar algoritmos de aprendizagem por reforço. A bancada *Gym* proporciona múltiplos ambientes de testes — videojogos *Atari*, robótica, simuladores de físicas *2D* e *3D* — de forma a treinar agentes que possam aprender e solucionar problemas autonomamente sem qualquer colaboração humana. Dispõe de uma interface partilhada que possibilita a criação de código genérico e por virtude da sua inexistência de estruturação convencionada de dados, é possível a conciliabilidade com diversas bibliotecas de análise numérica como NumPy e TensorFlow.

***Termos de Indexação*-** Agentes Inteligentes, Algoritmos de Aprendizagem por Reforço, Inteligência Artificial, *NumPy*, *OpenAI Gym*, *TensorFlow*.

I. INTRODUÇÃO

OpenAI Gym é uma *framework* que fornece a capacidade de desenvolver e comparar algoritmos de aprendizagem por reforço. A bancada *Gym* proporciona múltiplos ambientes de testes — videojogos *Atari*, robótica, simuladores de físicas *2D* e *3D* — de forma a treinar agentes que possam aprender e solucionar problemas autonomamente sem qualquer colaboração humana. Dispõe de uma interface partilhada que possibilita a criação de código genérico e por virtude da sua inexistência de estruturação convencionada de dados, é possível a conciliabilidade com diversas bibliotecas de análise numérica como NumPy e TensorFlow.

II. AMBIENTES DE ESTUDO

SPACE INVADERS

Um dos primeiros jogos com ambiente gráfico bidimensional, lançado em 1978, este tem o objetivo principal destruir camadas de naves espaciais e ganhar o maior número de pontos possível sem esgotar as três vidas. Neste ambiente, a observação é uma imagem *RGB*, com uma matriz de formas (210, 160, 3) onde cada ação é executada repetidamente por uma duração de *k frames*, onde *k* é uniformemente amostrado de {2, 3, 4}.
A simulação deste ambiente é possível graças à *framework "Arcade Learning Environment"* [ALE], que utiliza o emulador *Stella* [Stella] Atari.

FROZEN LAKE

O agente controla o movimento de um personagem em um mundo em formato de grelha. Algumas superfícies da grelha são caminháveis e outras levam ao agente a cair na água. Além disso, a direção do movimento do agente é incerta e depende apenas parcialmente da direção escolhida. O agente é recompensado por encontrar um caminho em que possa caminhar em segurança à superfície de destino.

III. ESTUDOS E OBSERVAÇÕES

IV.

A. Aprendizagem por Reforço

Antes de mais é importante também perceber o que significa aprendizagem por reforço. Este tipo de aprendizagem trata de ter um agente autónomo que toma ações de modo a maximizar a sua recompensa num dado ambiente. Ao longo das experiências, aprende e tenta adotar o melhor comportamento possível.
Neste tipo de aprendizagem é também importante limitar a interação humana a apenas efetuar mudanças nos estados do ambiente e no sistema de recompensas e penalizações.

B. OpenAI Gym

OpenAI Gym, uma ferramenta frequentemente usada por pesquisadores para padronização e *benchmarking* de resultados. Esta possui coleções de ambientes/problemas desenvolvidos para testes de algoritmos de aprendizagem por reforço, evitando que os utilizadores tenham que criar os seus próprios ambientes.

Nestes ambientes, haverá um agente que interage com o mundo através de ações aleatórias, tendo o objetivo de maximizar a sua pontuação de recompensa como apresentado na Figura 1.

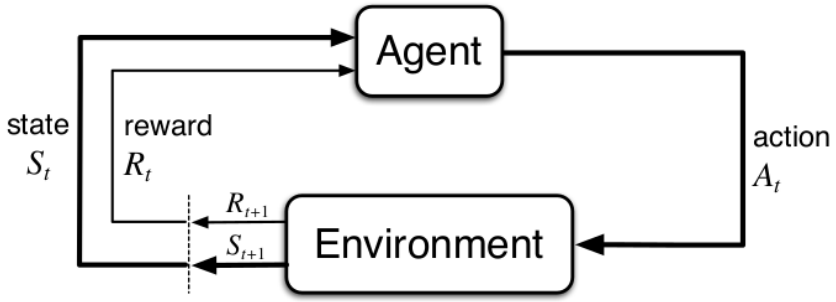

Figura 1- Aprendizagem por Reforço

Um agente no seu estado atual (S_t) efetua uma ação (A_t) no mundo no qual este responde e retorna um novo estado para o agente (S_{t+1}) e a sua correspondente recompensa (R_{t+1}). Com este novo estado e recompensa, o agente decide e efetua a sua próxima ação, este processo repete-se até que o agente chegue ao objetivo ou o ambiente seja terminado.

Para o correto funcionamento e execução deste estudo utilizou-se a versão 3.7.11 do *Python* e foram necessárias as seguintes dependências:

Dependência	**Versão**
Gym & Gym[Atari]	0.19.0
Pyglet	1.5.11
Stable_baselines3	1.3.0
SWIG	3.0.12
Keras-RL2	1.0.5
Tensorflow	1.14.0
Ale-Py	0.7.3
Microsoft Visual C++	14.0 ou superior

Realça-se que este estudo foi executado com base na plataforma *Jupyter*.

C. Q-Value

Q-value é a medida, a longo prazo, do retorno de um agente num dado estado sobre uma política, que tem em conta a ação que este mesmo agente escolhe nesse estado. A ideia é mostrar que as mesmas ações em diferentes estados poderão retornar diferentes recompensas.

$$V^{\pi}(s_t, a_t) = E_{\pi}\{\sum_{k=0}^{\infty}(\gamma^k r_{t+k+1})\}$$

Figura 2 – Q-Value

Esta função cria um mapa de estados e correspondentes ações e as suas respetivas recompensas. *Q-value* é assim usada no *Q-learning* de modo realçar o quão vantajoso é uma ação com base na sua recompensa futura num dado estado.

D. Atualizar *Q-Values*

Para atualização dos valores, basicamente, é adicionada a diferença temporal ao valor atual para um par de estado-ação. Esta atualização é influenciada por parâmetros, nomeadamente:

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha * \{r_t + \gamma * max_a Q(s_{t+1}, a) - Q^{old}(s_t, a_t)\}$$

Figura 3- Atualizar Q-Values

- α ou ***alpha***, que define ao ritmo (entre 0 e 1) a que o nosso algoritmo aprende, em que 0 significa que não existe aprendizagem e 1 atualiza sempre com a informação mais recente.
- γ ou ***gamma***, que define a importância (entre 0 e 1) que uma recompense futura poderá ter sobre as ações presentes, em que 0 significa que o agente age de forma oportunista tentando obter maior recompensa atual enquanto que 1 implica que o agente tenha em mente a recompensa futura podendo sacrificar recompensa inicial por uma melhor final

E. Algoritmo *Q-Learning*

O típico algoritmo de *Q-learning* tem com base o seguinte processo:

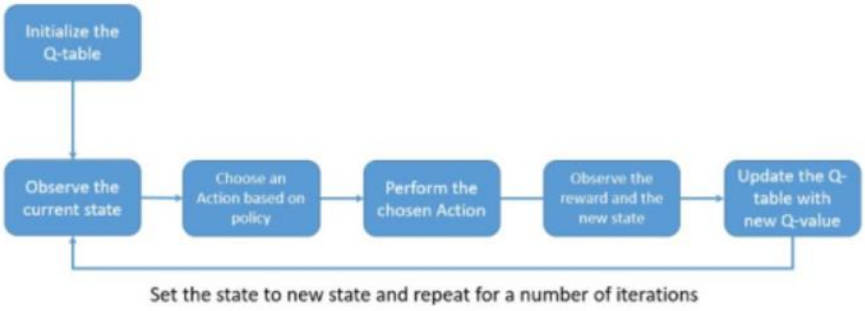

Figura 4- Processo Q-Learning

Os *Q-values* são guardados/atualizados numa *Q-table* (tabela composta por todos os *Q-values*), com dimensões correspondentes ao número de ações e estados no ambiente. A tabela é inicializada com zeros no início do processo.
Este processo começa por escolher uma ação através de uma consulta na tabela previa. Ao realizar a ação especificada, é retornada uma recompensa e o valor da ação na tabela é atualizado. Este processo poderá ser repetido inúmeras vezes, tendo em conta a eficácia do sistema e os recursos necessários para esta aprendizagem.

F. Como escolhe uma ação

Visto que a tabela é iniciada a zeros, a ação inicial tem de ser aleatória, no entanto, à maneira que a mesma começa a ser atualizada, o agente seleciona a ação que oferece melhor recompensa num determinado estado.
Porém, isto poderia significar que o agente ficaria destinado a repetir algumas das ações iniciais que não eram ótimas. Essencialmente o agente passaria de exploração para utilização do ambiente demasiado cedo. Para combater este problema é necessário introduzir uma política de seleção de ações chamada ϵ**-greedy**.

Isto implica que, é selecionado um número aleatório e se for menor que ϵ, a ação é escolhida aleatoriamente, permitindo assim ao agente explorar estados que não exploraria anteriormente e resolve o problema de repetir as mesmas ações iniciais. É importante notar que deverá gradualmente diminuir o ϵ de modo a que, com o passar dos episódios, se torne cada vez menos aleatório.

G. Rede Neuronal

Consistem de vários nós simples de processamento interligados e vagamente baseados como um cérebro humano funciona. Os mesmos são tipicamente organizados em camadas e são atribuídos pesos às suas ligações.
O objetivo é, através de várias iterações onde se efetuam correções, aprender o peso correto que cada ligação deve ter.
As camadas deste tipo de redes são compostas da seguinte forma:

- **Camada de entrada**: O número de nós de entrada desta camada é tipicamente fixo e corresponde à informação introduzida, por exemplo, o número de estados de um ambiente.

- **Camada/s escondida/s**: Poderão existir várias camadas escondidas neste tipo de redes. O número de nós e camadas neste grupo tende a variar com a complexidade e necessidade do problema.
- **Camada de saída**: O número de nós de saída desta camada é também tipicamente fixo, conforme o resultado pretendido, por exemplo o número de ações num ambiente.

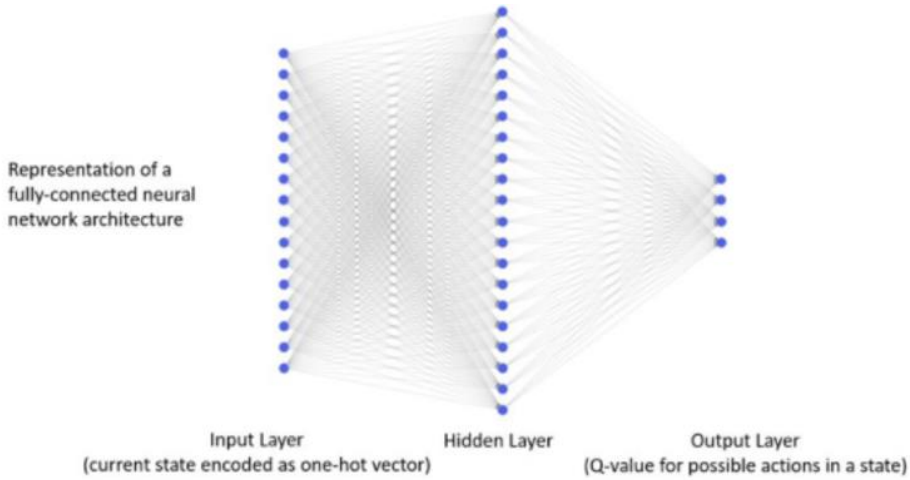

Figura 5- Rede Neuronal

No exemplo acima, podemos observar que existem 16 estados que são o número de nós na camada de entrada, existe também uma única camada escondida composta por 20 nós, responsável por efetuar a computação necessária para obter o resultado pretendido e por fim temos ainda uma única camada de saída com apenas 4 nós, ou seja, o número de ações do ambiente.

H. Como Funciona os nós da camada escondida

Os nós de processamento, ou nós da camada escondida, geram informação de saída com base na informação que recebem do nó anterior, incluindo pesos e tendências.

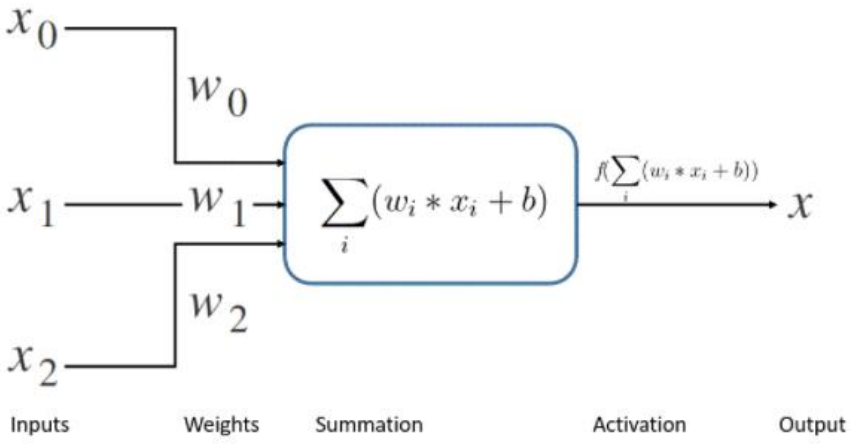

Figura 6- Nós camadas escondidas

No exemplo de um nó de processamento prévio, é utilizada também uma função de ativação, sendo a principal razão da sua necessidade, para introduzir não-linearidade no resultado, que caso contrário seria maioritariamente linear. Esta função permite este tipo de redes aprenderem problemas complexos que se possam apresentar no mundo real. Existem vários tipos destas funções e a escolha da mesma deve variar para cada rede com base na otimização da mesma e dos dados a serem explorados.

I. Função de Perda

Agora que já sabemos como as redes neuronais funcionam, sabemos com o decorrer das iterações tentam minimizar o erro das suas previsões, no entanto ainda existe erro, porque são previsões, para tal, é importante que, a um dado momento da nossa exploração, calculemos o erro do modelo. A função de perda permite-nos fazer exatamente isso, mede o valor da diferença entre o que era esperado e o que foi previsto:

$$\text{loss} = \{(\text{r} + \gamma * max_{a'} Q'(s', a')) - Q(s, a)\}^2$$

Figura 7- Função de Perda

O objetivo é, depois de calculado o erro, utilizar o mesmo e percorrer a rede pelo sentido contrário e atualizar os pesos das ligações. A isto damos o nome de retro propagação e existem vários algoritmos para o fazer, como por exemplo, ***Adam***.

J. Porquê Redes Neuronais e não Q-Learning?

Embora seja viável a criação de uma *Q-table* para ambientes simples, torna-se difícil a sua implementação em ambiente que envolvem situações reais. O número de ações e estados neste tipo de ambientes poderão ser milhares, tornando a atualização de *Q-values* extremamente ineficiente.
É aqui que entram as redes neuronais, pois permitem prever os valores para cada ação dado um determinado estado. Em vez da inicialização e atualização da tabela no processo de *Q-learning*, inicializamos e treinamos um modelo da rede neuronal.

L. Algoritmo *DQNAgent*

O algoritmo *Deep Q-network* (DQN) é uma abordagem aproximada no algoritmo *QLearning*. Este é um método de aprendizagem por reforço *on-line*, *model-free* e *off-policy*.
Um agente DQN é um agente de aprendizagem por reforço baseado num valor que treina um crítico para estimar o retorno ou recompensas futuras. É permitido o treino em espaços de observação contínuos ou discretos como em espaços de ação discretos.
O espaço de ação é explorado através do método *epsilon-greedy*, durante cada intervalo de controlo, o agente seleciona uma ação aleatória com probabilidade ϵ ou seleciona gananciosamente uma ação tendo em conta o valor da função com probabilidade **1-ϵ**, esta ação reflete o valor da função maior.
A experiencia passada é guardada num *buffer* circular onde o agente atualiza o valor do crítico baseando-se numa pequena amostra de experiencias aleatoriamente retiradas deste mesmo *buffer*.

M. Função Crítica

Para estimar o valor da função, o agente mantem duas funções de aproximação:

- ***Q* (S, *A*|ϕ)** – O valor crítico obtém uma observação do estado *S* e da ação *A* como entradas e retorna a expectativa correspondente da recompensa a longo prazo com o auxilio do parâmetro ϕ.

- **Q_t (*S*, *A*|ϕt)** – O valor crítico objetivo é utilizado para melhorar a estabilidade da otimização, o agente irá periodicamente atualizar os parâmetros críticos objetivos alvo ϕ_t para os valores críticos mais recentes.

Quando a fase de treinos dá-se por concluída, o aproximador da função de valor é armazenado no crítico ***Q (S,A)***.

N. Space Invaders

Para começar a trabalhar no ambiente de aprendizagem por reforço será necessário por primeiramente importar a ferramenta *gym* e definir o ambiente de testes. Podemos também observar que a altura, largura e os respetivos canais do ambiente estão a ser recolhidos para a futura construção da rede neuronal como também todas as possíveis ações que o agente poderá efetuar neste espaço.

```
import gym

env = gym.make('SpaceInvaders-v0')
height, width, channels = env.observation_space.shape
actions = env.action_space.n
```

Figura 8- Importação e declaração do ambiente

Com isto feito podemos começar por examinar o ambiente em que os testes serão efetuados, este ambiente é puramente aleatório e não executa qualquer algoritmo de aprendizagem. Estes 5 episódios demonstrativos serão realizados num ciclo – enquanto o ambiente não tenha sido concluído ou não tenha esgotado o número de episódios, uma nova janela irá ser aberta com o videojogo *Space Invaders*, onde o agente irá realizar uma ação aleatória atualizando o seu estado e recompensa consoante a lógica referida anteriormente na *Figura 1*.
Por fim, será imprimido o episódio juntamente com o seu correspondente valor e o ambiente será terminado.

```
episodes = 5
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = env.action_space.sample()
        # extract info from each episode
        n_state, reward, done, info = env.step(action)
        # append episode score to total score
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
}
```

Figura 9- Demonstração Episódio

Figura 10- Ambiente SpaceInvaders

Com uma pequena noção de como o ambiente funciona, poder-se-á agora iniciar a construção de uma aprendizagem profunda utilizando a biblioteca *Keras* com o auxílio de *TensorFlow*. Será necessário a utilização do modelo de aprendizagem *Sequencial*, métodos de conversões de camadas e o algoritmo de otimização *Adam*.

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Convolution2D
from tensorflow.keras.optimizers import Adam
```

Figura 11- Imports biblioteca Keras-RL

Para a construção desta aprendizagem profunda *Sequencial* será construído um método que passará os atributos do ambiente recolhido no início do programa. Como o presente ambiente é baseado em imagens *RGB* iremos começar por convolucionar cada imagem e posteriormente achata-las.

```python
def build_model(height, width, channels, actions):

    model = Sequential()
    model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu', input_shape=(3, height, width, channels)))
    model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu'))
    model.add(Convolution2D(64, (3,3), activation='relu'))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model
```

Figura 12- Método Deep Learning

Na primeira convolução estamos a passar 32 filtros como parâmetro utilizando uma matriz 8x8, estes 32 filtros irão analisar e detetar na imagem acontecimentos, dito isto, será possível identificar onde uma nave inimiga está posicionada, por exemplo. Consequentemente iremos analisar e convolucionar os dados de 4 em 4 passos armazenando-os numa nova matriz 2x2. As restantes convoluções seguem a mesma logica mas com tamanhos diferentes para garantir que todo o ambiente é captado e analisado da melhor forma possível. O processo de convolução poderá ser observado na Figura 13 e 14.
Escolheu-se o método de ativação não-linear *relu* que irá retornar apenas a informação positiva introduzida, caso contrário esta será apresentada como zero e por fim indicamos ao modelo que tipo de dados serão recebidos para análise através de 3 *dataframes*.

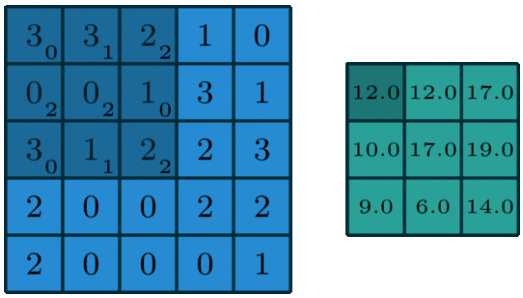

Figura 13- Exemplo Convolução

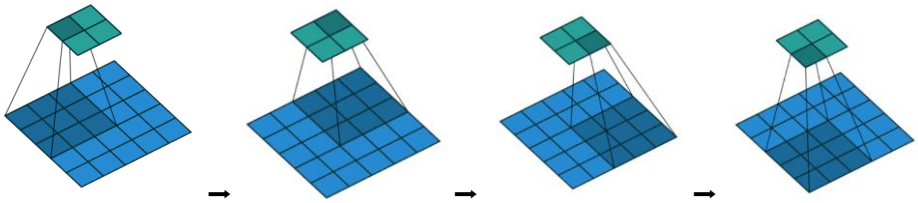

Figura 14- Exemplo Strides

Todas as camadas dos modelos de convolução serão então achadas em uma única camada através do comando *Flatten()* para que seja possível posteriormente criar camadas densas onde todas as unidades destas camadas está conectada com todas as unidades da próxima camada, como demonstrado na Figura 15. A primeira camada terá 512 unidades e a seguinte sofrerá uma compressão para 256 unidades, por fim a ultima camada terá apenas 6 unidades, estas 6 unidades refletem as possíveis ações que o agente poderá executar no mundo, a unidade que for ativada corresponderá à ação que este efetuará.

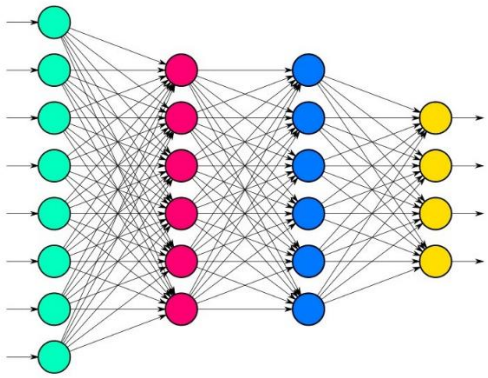

Figura 15- Exemplo Dense Layers

O. Construir Agente com *Keras-RL*

Com o modelo da nossa aprendizagem profunda realizado o passe seguinte será implementar o algoritmo *DQNAgent* com a ajuda da biblioteca *Keras-RL*.
Será necessário a importação de um *buffer* que guardará em memória a informação dos jogos anteriores (*SequentialMemory*) e de políticas de otimização para calcular a melhor estratégia e o melhor resultado das recompensas (*LinearAnnealedPolicy* e *EpsGreedyQPolicy*).

```
from rl.agents import DQNAgent
from rl.memory import SequentialMemory
from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy
```

Figura 16- Imports DQNAgent

A política *Linear Annealed* é uma técnica probabilística para aproximar o valor ótimo global de uma dada função, neste caso *Epsilon Greedy*, aqui definimos uma probabilidade ε no qual fará com que o agente possa tentar ignorar os valores Q aprendidos e tentar uma ação aleatória com probabilidade de (1 - ε).
Portanto, como o valor *épsilon* está definido como 0,2 (*value_test*) este irá maioritariamente executar as ações sugeridas pela *Q-Table*, pois um valor mais próximo de 0 para *epsilon* fará com que as ações escolhidas dependam mais dos valores Q aprendidos.
A atribuição de um valor fixo para *epsilon* é raramente usado devido às variâncias dos valores da taxa de aprendizagem do agente e da aleatoriedade do determinado ambiente, para tal, implementa-se um termo ε decrescente ao longo do tempo, começando de 1 (que não depende das previsões aleatórias iniciais) a 0,1 (dependendo dos valores Q previstos 9 de 10 vezes). Portanto, um epsilon em declínio garante que nosso agente não dependa das previsões aleatórias no início dos treinos.

```
def build_agent(model, actions):

    policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.1, value_test=.2, nb_steps=10000)
    memory = SequentialMemory(limit=1000, window_length=3)
    dqn = DQNAgent(model=model, memory=memory, policy=policy,
                   enable_dueling_network=True, dueling_type='avg',
                    nb_actions=actions, nb_steps_warmup=10000)
    return dqn
```

Figura 17- Método Agente

O *buffer* de memória do agente é guardado em 3 janelas com um limite de 1000 episódios, este parâmetro controla quantas observações são concatenadas para formar um *state*.
Posteriormente será necessário declarar o algoritmo *DQNAgent*, este receberá como parâmetros o modelo *sequencial* realizado no método anterior e o *buffer* de memória e politica de otimização definidas neste método. Será também ativada uma *dueling network* que fará com que seja configurado uma rede de "competição" e uma rede de aprendizagem por reforço, basicamente uma *dueling network* irá dividir os critérios "valor e vantagem" em dois, isto fará com que o modelo aprenda quando realizar ou não uma ação sem ter que aprender o efeito de cada ação para cada estado.
Geralmente, na aprendizagem por reforço, os valores iniciais de um treino possuem demasiados erros que podem causar oscilações nos parâmetros, para evitar isso definimos um número de passos de "aquecimento" para que o agente já possa ir com um conhecimento básico prévio, melhorando assim os resultados finais e reduzindo a estimativa de tempo.

O algoritmo encontra-se na sua fase final, podemos agora compilar o Agente utilizando novamente o algoritmo de otimização *Adam* que se baseará numa estimativa adaptativa de momentos de primeira e segunda ordem com um ritmo de aprendizagem 0.0001.

O modelo é então adaptado e definido fazendo com que o agente realize 1000 passos no mundo. Nota-se que a visualização encontra-se desativada para acelerar o processo de treino então declaramos o parâmetro *verbose=2* para recebermos o *output* de cada *epoch* em formato de texto.

```
dqn = build_agent(model, actions)
dqn.compile(Adam(learning_rate=1e-4))

dqn.fit(env, nb_steps=1000, visualize=False, verbose=2)
```

Figura 18- Implementar Agente

Após o agente ter realizado o seu treino de 1000 passos no mundo, podemos visualizar como este aprendeu ao correr um teste ao algoritmo *DQN* realizado, definimos o ambiente, número de episódios a correr e, obviamente, passar o parâmetro de visualização para *True.*

```
scores = dqn.test(env, nb_episodes=10, visualize=True, verbose=2)
print(np.mean(scores.history['episode_reward']))
```

Figura 19- Visualizar Teste Agente

Ao correr estas duas ultimas linhas de código irá abrir uma nova janela com o agente a executar as ações que aprendeu, juntamente em formato de texto a recompensa de cada episódio.

Como é óbvio, 1000 episódios não são o suficiente para treinar um agente, seria necessário milhões de iterações para que este começasse a chegar a resultados satisfatórios. Neste caso, para que o agente seja capaz de ganhar constantemente o videojogo e adquirir valores de recompensa contínuos e altos, seria necessário correr o algoritmo por 10 milhões de episódios.

Após a criação e execução do algoritmo seria conveniente poder guarda-lo para futuras análises de comparação. A biblioteca *keras-rl* permite armazenar os resultados dos testes com o seu comando *save_weights* onde somente é necessário declarar o local a guardar.

```
dqn.save_weights('SavedWeights/1k-Fast/dqn_weights.h5f')
```

Figura 20- Guardar Teste

Consequentemente também é possivel carregar treinos guardados através do comando *load_weights* onde é necessário indicar o local onde está o ficheiro de treino. Realça-se que antes de executar este comando, será necessário apagar o modelo anteriormente criado para não gerar conflitos.

```
del model, dqn
dqn.load_weights('SavedWeights/1m/dqn_weights.h5f')
```

Figura 21- Carregar Agente

P. Frozen Lake

Este ambiente consiste de um mundo em rede de tamanho maleável, onde existe sempre um começo e um objetivo. Neste mesmo mundo, poderão ainda existir, partes da rede congeladas que permitem que o agente caminhe sobre elas ou "buracos" na rede que provocam o final da iteração/episódio.
O objetivo deste ambiente é assim estudar o melhor caminho para chegar ao objetivo.
Exemplo:

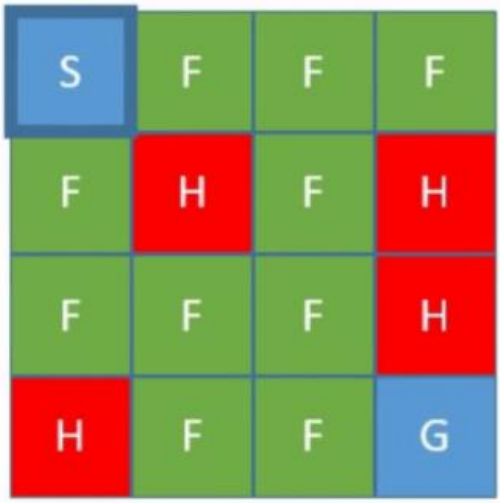

Figura 22- Exemplo Frozen Lake

Na figura anterior, temos um mundo 4x4 com 16 casas possíveis. Existe uma casa inicial (S) e uma casa objetiva (G). As casas caminháveis (F), serão as casas utilizadas para chegar ao objetivo, no entanto, terá de ter cuidado com as casas que poderão provocar a morte do agente (H). O agente deverá aprender o caminho sem intervenção humana e será recompensando quando chegar à casa objetiva.

Ficheiros:
No conteúdo anexado, encontram-se duas versões do ambiente ***FrozenLake***:

- **IA_Frozen > qlearning_frozen**, pasta que contem os ficheiros que permitem a exploração com algoritmo de *Q-learning*.
- **IA_Frozen > neural_net_frozen**, pasta que contem os ficheiros que permitem a exploração com uma rede neuronal.

No que diz respeito à pasta que contem os ficheiros do algoritmo *Q-learning*, podemos encontrar:

- "***__constants.py***" -> Ficheiro que contém constantes do ambiente, bem como uma função de geração de mapas aleatórios para este ambiente.
- "***epsilon.py***" -> Ficheiro que contém a classe que irá controlar a probabilidade de aleatoriedade (*epsilon*).
- "***qtable.py***" -> Ficheiro que como o nome indica contém a classe que irá agir como uma *Q-table* e adicionalmente uma função para treinar esta mesma tabela.
- "***frozen_lake_qlearning.py***" -> Ficheiro onde será carregado o ambiente "FrozenLake-v1" e onde será feita a aprendizagem usando a *Q-table* e a simulação do agente utilizando esta mesma tabela. É importante notar que deverá ser feita a instalação do pacote "***gym***".

```python
# step 1: loading the environment
env = gym.make("FrozenLake-v1", desc=MAPS['4x4'])

# step 2: creating the Q-table
state_size = env.observation_space.n
action_size = env.action_space.n
q = QTable(state_size, action_size)

# step 3: creating de epsilon decay
e = Epsilon(initial_epsilon=1.0, max_epsilon=1.0, min_epsilon=0.01, decay_rate=0.00005)

# step 4: Q-table training
q, rewards = train_qtable(env, q, e, Q_TOTAL_EPISODES, Q_MAX_STEPS)

print("Score over time {:.4f}".format(sum(rewards) / Q_TOTAL_EPISODES))
q.print()

# Play
env.reset()
env.render()
rewards = []

for episode in range(ACTION_EPISODES):
    state = env.reset()
    step = 0
    total_rewards = 0

    for step in range(ACTION_STEPS):
        action = q.select_action(env, state)

        # new_state -> the new state of the environment
        # reward -> the reward
        # done -> the a boolean flag indicating if the returned state is a terminal state
        # info -> an object with additional information for debugging purposes
        new_state, reward, done, info = env.step(action)

        total_rewards += reward
        state = new_state

        if done:
            break

    rewards.append(total_rewards)

    if episode % 100 == 0:
        print("=====================================")
        print("EPISODE {}".format(episode))
        print("Number of steps: {}".format(step))
        env.render()

print("Score over time {:.4f}".format(sum(rewards) / 1000))

env.close()
```

Figura 23- Código Frozen Lake

Neste ficheiro conseguimos observar os passos na aprendizagem por *Q-learning*, começamos por carregar o ambiente, depois é criada a *Q-table* e definido o *epsilon* e de seguida segue-se a aprendizagem e atualização da tabela. Por fim, inicia-se a fase da simulação, onde é utilizado o conhecimento previamente adquirido para tentar chegar ao objetivo.

Quanto à pasta que contem os ficheiros da aprendizagem com rede neuronal, podemos encontrar:

- "***__constants.py***" -> Ficheiro que contém constantes do ambiente, bem como uma função de geração de mapas aleatórios para este ambiente.
- "***main.py***" -> Ficheiro onde será carregado o ambiente "FrozenLake-v1" e onde será feita a aprendizagem usando a rede neuronal e a simulação do agente utilizando esta mesma tabela. É importante notar que deverá ser feita a instalação dos pacotes "***numpy***", "***keras***" e "***gym***" para que este ambiente funcione.

```python
# step 1: loading the environment
env = gym.make('FrozenLake-v1')

# step 2: define the algorithm variables
discount_factor = 0.95
eps = 0.5
eps_decay_factor = 0.999

# step 3: create the neural network model
model = Sequential()
model.add(InputLayer(batch_input_shape=(1, env.observation_space.n)))
model.add(Dense(env.action_space.n, activation='relu'))
# model.add(Dense(env.action_space.n, activation='linear'))
model.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

Figura 24- Código Frozen Lake

Visto que as redes neuronais, são um método de aprendizagem por reforço, terão também os mesmos parâmetros que regulam a aleatoriedade, a importância de recompense e ao ritmo que é atualizada nova informação, no entanto, existe, como já anteriormente mencionado, função de ativação, perda, retro propagação e ainda uma função para regular o desempenho do modelo. Respetivamente temos:

- Função de ativação: ***ReLU***, ou ***Rectified Linear Unit***, é uma função não-linear que irá retornar a informação introduzida se for positive, caso contrário, retornará zero. É a função utilizada normalmente utilizada por defeito pois é a mais simples e mais eficaz atualmente. Funciona da seguinte forma:

$$f(x) = max(0, x)$$

```
def ReLU(x):
  if x>0:
    return x
  else:
    return 0
```

Figura 25- Função ReLU

- Função de perda: ***MSE***, ou ***Mean Squared Error***, permite obter uma média de erros em vez de dar o número exato de erros como outras funções deste tipo, por exemplo ***(SSE)***, ou seja, quanto maior a informação processada, menor será o erro porque supostamente irá estar a aprender. É assim extremamente viável para grandes quantidades de dados e é por isso que é a função utilizada por defeito em vários algoritmos. A sua representação matemática é a seguinte:

$$\mathbf{L} = \frac{1}{\mathbf{N}}[\,\Sigma(\hat{\mathrm{Y}} - \mathrm{Y})^2]$$

Figura 26- Função de Perda

- Função de retro propagação: ***Adam***, otimizador que deriva do método de **Estimativa de Momento Adaptativo**, "é um método estocástico de gradiente descendente que se baseia na *estimative adaptative* de momentos de primeira e segunda ordem". A sua representação matemática é a seguinte:

$$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{\hat{v}_n} + \epsilon}\hat{m}_n$$

Figura 27- Função retro propagação

Pseudocódigo da mesma:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figura 28- Pseudocódigo retro propagação

E em comparação com os restantes métodos de otimização, é possível observar que é o mais eficiente, principalmente a longo prazo:

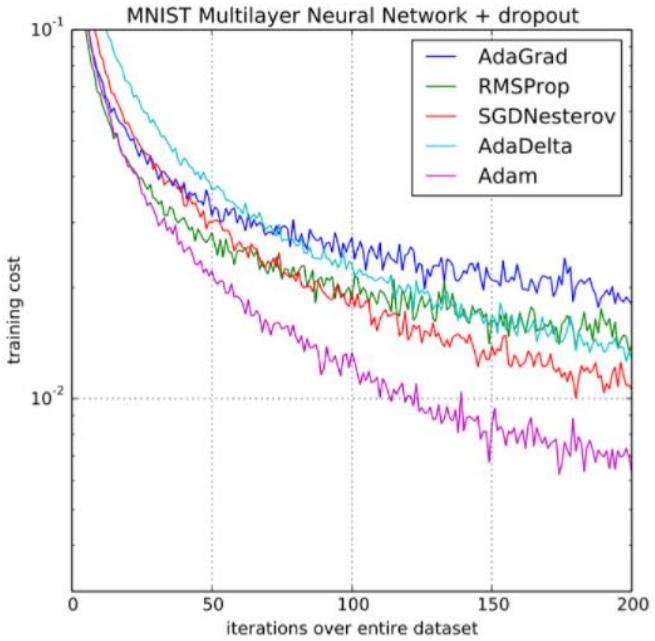

Figura 29- Comparação metodos otimização

- Função de avaliação do desempenho do modelo: ***MAE,*** ou ***Mean Absolute Error***, é uma métrica que indica a diferença absoluta média entre os valores previstos e os valores reais de um conjunto de dados. Quanto menor o valor desta, melhor está o modelo ajustado aos dados. A sua representação matemática:

$$mae = \frac{\sum_{i=1}^{n} abs\left(y_i - \lambda(x_i)\right)}{n}$$

Figura 30- Função de avaliação do desempenho do modelo

V. CONCLUSÃO

Resumidamente, explorámos vários algoritmos de inteligência artificial, nomeadamente, *DQNAgent*, *Q-learning* e Redes Neuronais utilizando ambientes pré-construídos do pacote *Gym*, respetivamente, "*SpaceInvaders-v0*" e "*FrozenLake-v1*". Visto que o objetivo deste artigo era servir como elemento de avaliação e guião para futuro alunos da cadeira de "Inteligência Artificial" da Faculdade de Ciências e Tecnologias da Universidade dos Açores, fomos mutuamente capazes de demonstrar que adquirimos os conhecimentos e efetuamos a exploração pretendida, bem como, colocamos informação suficiente para os próximos estudantes conseguirem não só perceber os algoritmos utilizados, mas também testarem os ambientes fornecidos.

REFERÊNCIAS

[1] https://www.baeldung.com/cs/reinforcement-learning-neural-network
[2] https://www.baeldung.com/cs/neural-net-advantages-disadvantages
[3] https://www.guru99.com/reinforcement-learning-tutorial.html#types-of-reinforcement-learning
[4] https://www.baeldung.com/cs/neural-networks-hidden-layers-criteria
[5] https://www.oreilly.com/radar/introduction-to-reinforcement-learning-and-openai-gym/
[6] https://gym.openai.com/docs/
[7] https://www.mathworks.com/help/reinforcement-learning/ug/dqn-agents.html
[8] https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2
[9] https://www.youtube.com/watch?v=hCeJeq8U0lo
[10] https://www.baeldung.com/cs/ml-linear-activation-functions
[11] https://keras.io/api/layers/activation_layers/relu/
[12] https://keras.io/api/optimizers/adam/
[13] https://iq.opengenus.org/relu-activation/
[14] https://towardsdatascience.com/https-medium-com-chayankathuria-regression-why-mean-square-error-a8cad2a1c96f
[15] https://keras.io/api/losses/regression_losses/#mean_squared_error-function
[16] https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be
[17] https://www.geeksforgeeks.org/intuition-of-adam-optimizer/
[18] https://www.statology.org/mae-vs-rmse/
[19] https://medium.com/@ewuramaminka/mean-absolute-error-mae-machine-learning-ml-b9b4afc63077
[20] https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1
[21] https://keras.io/api/layers/reshaping_layers/flatten/
[22] https://medium.com/appengine-ai/dense-layers-in-artificial-intelligence-b2f79cc1534a

CÓDIGO

O código utilizado e os respetivos resultados apresentados neste estudo encontram-se disponíveis na plataforma *GitHub* (https://github.com/BernardoSousa/IA-OpenAI-ProjG3).

AUTORES

Airton Tavares – Universidade dos Açores, Informática 2019109312@uac.pt
Bernardo Sousa – Universidade dos Açores, Informática 2019109334@uac.pt