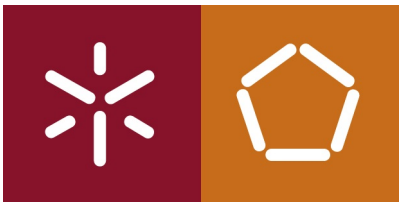

Universidade do Minho
mestrado integrado em engenharia informática

Computação Natural

Sistemas Inteligentes

(2º Semestre / 4º Ano)

Deep Reinforcement Learning Aplicado a Agentes

GRUPO 10
Diogo Braga (a82547)
Pedro Ferreira (a81135)
Ricardo Caçador (a81064)
Ricardo Veloso (a81919)

Abril, 2020

Conteúdo

1 Introdução

O presente relatório detalha a preparação de um algoritmo de *deep reinforcement learning* capaz de aprender a jogar o famoso jogo: Doom. Este projeto foi proposto no âmbito da Unidade Curricular de Computação Natural, inserida no perfil de especialização de Sistemas Inteligentes.

Um dos desafios recorrentemente ligados ao *reinforcement learning* é o de aprender a controlar agentes diretamente a partir de informações sensoriais, como a visão. As aplicações de *reinforcement learning* mais bem sucedidas que operam nesses domínios confiam em recursos criados manualmente, combinados com funções de valores lineares ou representações de políticas. De facto, o desempenho de tais sistemas depende muito da qualidade da representação do recurso [1].

O *deep learning* tornou possível extrair recursos de alto nível de dados brutos, levando a avanços na visão computacional. Esses métodos utilizam uma variedade de arquiteturas de redes neuronais, incluindo redes convolucionais e *multilayer perceptrons*. No entanto, o *reinforcement learning* apresenta vários desafios na perspectiva duma possível junção com o *deep learning*. Em primeiro lugar, verifica-se que as aplicações de *deep learning* mais bem sucedidas exigem grandes quantidades de dados de treino supervisionados. Por outro lado, os algoritmos de *reinforcement learning* devem ser capazes de aprender com valores de recompensa frequentemente escassos e com imperfeições. Outra questão passa pela maioria dos algoritmos de *deep learning* suporem que as amostras de dados são independentes, enquanto que no *reinforcement learning* são encontradas tipicamente sequências de estados altamente correlacionados. Além disso, no *reinforcement learning*, a distribuição de dados muda à medida que o algoritmo aprende novos comportamentos, o que pode ser problemático para métodos de *deep learning*.

A junção destes dois conceitos surge no sentido de possibilitar que uma rede neuronal convolucional possa superar esses desafios, de forma a aprender políticas de controlo bem sucedidas, a partir de dados digitais em ambientes complexos de *reinforcement learning*. A rede é treinada com uma variante do algoritmo *Q-learning*, em junção com a aplicação da técnica de *stochastic gradient descent*, para atualização dos pesos. Para aliviar os problemas de dados correlacionados e distribuições não estacionárias, é utilizado um mecanismo de repetição de experiência que mostra aleatoriamente transições anteriores e facilita a distribuição de treino sobre comportamentos passados.

2 Deep Reinforcement Learning

Os avanços recentes na visão computacional contam com o treino eficiente de redes neuronais profundas, em conjuntos de treino muito grandes. As abordagens mais bem sucedidas são treinadas a partir de entradas diretas, utilizando atualizações leves baseadas na descida do gradiente estocástico. Ao introduzir dados suficientes em redes neuronais profunda, é geralmente possível aprender melhores representações do que com recursos manuais [1]. O sucesso relacionado com o *deep learning* motiva a ligação com o *reinforcement learning*. O objetivo é conectar um algoritmo de *reinforcement learning* a uma rede neuronal profunda, que opere diretamente em imagens e processe dados de treino com eficiência.

2.1 Ambiente do jogo

O algoritmo em causa tem como objetivo final aprender a jogar e a vencer o jogo Doom. Neste sentido, necessita de se adaptar as variáveis e ao estado do jogo, sendo que vai ser nesse ambiente que o agente, que executa o algoritmo de *deep reinforcement learning*, vai funcionar.

Para execução do algoritmo de DRL, os métodos do *ViZDoom* mais importantes utilizados foram os seguintes [2]:

- `game = vizdoom.DoomGame()`, para criar o jogo;
- `game.load_config(config_file_path)`, para configurar o cenário a utilizar no jogo;
- `game.set_screen*`, para definições do ecrã;
- `game.init()`, para iniciar o jogo;
- `game.get_available_buttons_size()`, para saber as ações possíveis a executar;
- `game.new_episode()`, para criar um novo episódio no jogo;
- `game.get_state()`, para saber o estado atual do jogo;
- `game.make_action`, para realizar uma ação;
- `game.is_episode_finished()`, para saber se o episódio do jogo está terminado;
- `game.get_total_reward()`, para saber a recompensa associada a um jogo;

Através da conjugação destes métodos para obter e modificar o estado do jogo, é possibilitada a interação do agente com o ambiente em causa, podendo assim este aplicar o algoritmo e aprender a vencer em diferentes cenários. Os cenários testados neste projeto foram o `basic` e o `defend_the_center`, visíveis na figura 1. No primeiro, existe um inimigo inofensivo numa determinada posição, e o agente tem que se movimentar de forma a ficar em frente a este, para depois disparar de modo a matá-lo e assim vencer o jogo. No segundo, existem variados inimigos numa área envolvente ao agente que, movendo-se para a esquerda ou para a direita numa possibilidade de $360^{\underline{o}}$, deve disparar para matar os inimigos antes que estes cheguem a si e comecem a tirar-lhe vida.

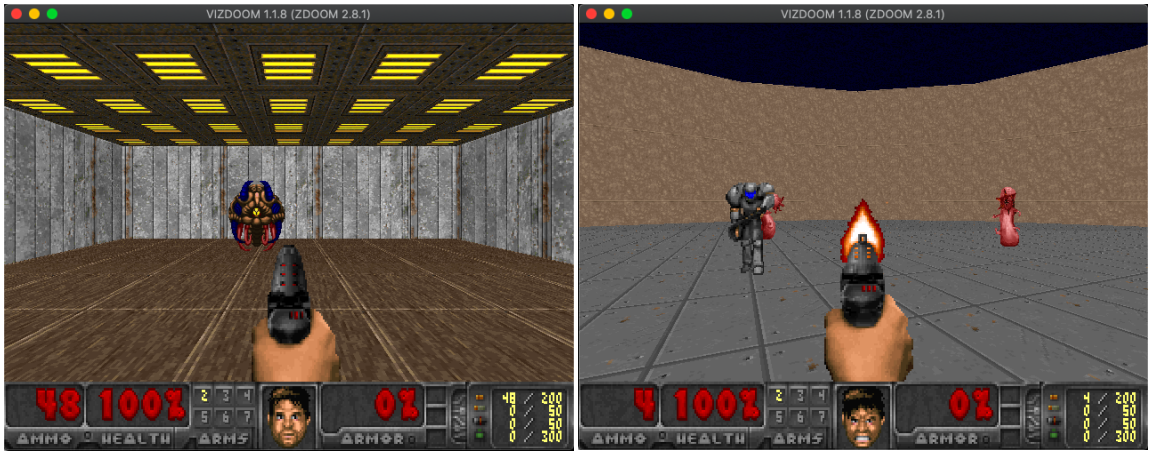

Figura 1: Cenários do jogo testados (lado esquerdo: `basic`; lado direito: `defend_the_center`)

2.2 Agente

Após definido o ambiente do jogo, surge a etapa de criar um agente que interprete o papel de jogador e que, através da evolução algoritmo de *deep reinforcement learning*, alcance o objetivo final do projeto: vencer o jogo da forma mais eficiente possível. Uma das partes mais importantes do algoritmo é a repetição de experiência, uma técnica onde são armazenadas as experiências do agente a cada etapa do tempo:

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

Em que:

- s_t: primeiro estado;
- a_t: ação executada;
- r_t: recompensa associada à ação executada;
- s_{t+1}: segundo estado, após execução da ação.

Estas experiências são agrupadas num *dataset* de episódios $D = e_1, ..., e_N$ na *replay memory*. Para a interpretação dos estados é realizado um pré-processamento da imagem em que o jogo se encontra e, após um redimensionamento, este passa a ser o próprio estado do jogo, utilizado para análise na *replay memory*.

Durante o *loop* interno do algoritmo, são aplicadas atualizações de Q-learning a amostras de experiência, extraídas aleatoriamente da pool de amostras armazenadas. Após executar a repetição da experiência, o agente seleciona e executa uma ação de acordo com a política definida. Como utilizar históricos de comprimento arbitrário como entradas para uma rede neuronal pode ser difícil, a função Q funciona com uma representação de comprimento fixo das histórias produzidas por uma outra função. O algoritmo de aprendizagem é apresentado de seguida:

1. Criar rede neuronal, com arquitetura antes definida;
2. Inicializar a *replay memory*;

3. Para X rondas, em cada ronda:
   (a) Criar novo episódio;
   (b) Até episódio terminar, em cada ação:
      i. Pré-processar estado atual;
      ii. Calcular *exploration rate* a aplicar na tomada de decisão;
      iii. Selecionar melhor ação retornada pela rede ou ação aleatória (dependendo do *exploration rate*);
      iv. Realizar ação e calcular a recompensa associada;
      v. Pré-processar novo estado;
      vi. Adicionar a transição de estados realizada e atributos associados na *replay memory*;
      vii. Treinar a rede neuronal com aplicação da `Q-function`, utilizando também a nova experiência presente na memória.
   (c) Após episódio terminado, obter recompensa total.

Na seguinte figura é ilustrado o processo de aprendizagem aplicado, com especial destaque para o papel da *replay memory*:

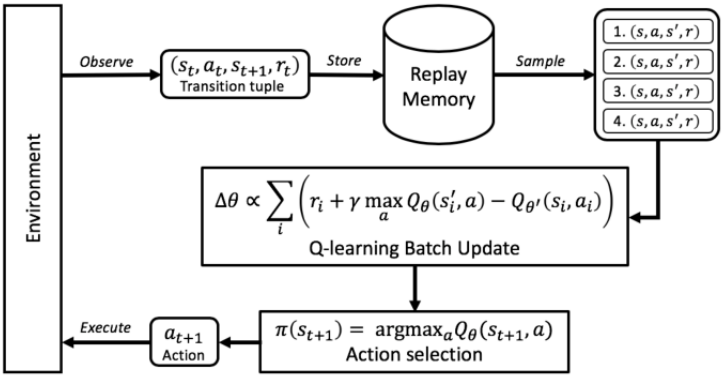

Figura 2: Replay memory no algoritmo DRL [3]

Esta abordagem apresenta várias vantagens. Primeiro, cada etapa da experiência é potencialmente usada em muitas atualizações de peso, o que permite maior eficiência dos dados. Segundo, aprender diretamente com amostras consecutivas é ineficiente, devido às fortes correlações entre as amostras; a aleatoriedade das amostras diminui essas correlações e, portanto, reduz a variação das atualizações. Terceiro, ao aprender a política, os parâmetros atuais determinam a próxima amostra de dados na qual os parâmetros são treinados. Por exemplo, se a ação maximizadora for mover para a esquerda, as amostras de treino serão dominadas por amostras do lado esquerdo; se a ação maximizadora for mover para a direita, a distribuição do treino também será alterada. É possível ver como os loops de *feedback* indesejados podem surgir e os parâmetros podem ficar presos num mínimo local negativo ou até divergir. Ao usar o *replay* da experiência, a distribuição do comportamento é calculada em média sobre muitos dos seus estados anteriores, facilitando a aprendizagem e evitando oscilações ou divergências nos parâmetros.

Na prática, este algoritmo armazena apenas os últimos tuplos de experiência na memória de reprodução e é amostrado aleatoriamente de maneira aleatória ao executar atualizações. Essa

abordagem é limitada em alguns aspectos, já que o *buffer* de memória não diferencia transições importantes e substitui sempre as transições recentes devido ao tamanho finito da memória. Da mesma forma, a amostragem uniforme dá igual importância a todas as transições na memória de reprodução.

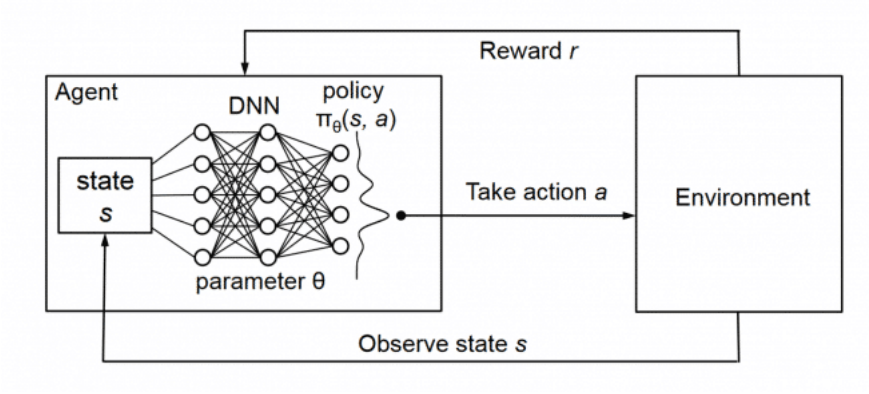

Figura 3: Deep Reinforcement Learning [4]

Em relação ao modelo utilizado no agente, este foi desenvolvido em *tensorflow* e, para o cenário `basic`, possui a seguinte arquitetura base:

- Duas camadas `Convolution`;
- Uma camada `Flatten`;
- Duas camadas `FullyConnected`, sendo uma de saída;

Os parâmetros, como o número de nodos utilizados em cada camada, função de ativação e *learning_rate*, são definidos posteriormente pela execução do algoritmo de otimização: `Particle Swarm Optimization`.

Devido à maior complexidade do cenário `defend_the_center`, foi concluído, através de alguns testes, que a arquitetura base apresentada, e utilizada no cenário simples, não era suficiente para apresentar bons resultados para as variáveis do estado do jogo. Desta forma, foi desenvolvida uma outra, mais complexa:

- Três camadas `Convolution`;
- Uma camada `Flatten`;
- Três camadas `FullyConnected`, sendo uma de saída;

3 Particle Swarm Optimization

De forma a encontrar os melhores parâmetros para a rede neuronal utilizada no cenário simples, recorremos à meta-heurística *particle swarm optimization* (PSO).

O PSO apresenta-se como um algoritmo baseado em sistemas sociais, como o comportamento em grupo das aves, e normalmente é utilizado para otimização de problemas. Trata-se de um método computacional que otimiza um problema, iterativamente, ao tentar melhorar a solução candidata com respeito a uma dada medida de qualidade. Este resolve um problema ao ter uma população de soluções candidatas, denominadas partículas, e movendo-as no espaço de pesquisa de acordo com fórmulas matemáticas que incidem sobre a posição e a velocidade da partícula. O movimento de cada partícula não só é influenciado pela melhor solução local conhecida, mas também pelas melhores soluções conhecidas globalmente. Assim, como em cada iteração as partículas atualização a sua posição, é espectável que as posições comecem a ficar mais concentradas em torno de uma determinada área do espaço de pesquisa. Isto faz com que todo o enxame se mova em direção às melhores soluções.

No problema retratado neste relatório, o espaço de pesquisa era composto por quatro variáveis:

- *Learning rate*
- Função de ativação das camadas de convolução
- Número de *feature maps* resultantes das camadas de convolução
- *Discount factor* do algoritmo de *Q-learning*

Ao contrário do que geralmente acontece na aplicação desta meta-heurística, optamos por utilizar um espaço de procura discreto, sendo que apenas o *discount factor* poderia assumir qualquer valor pertencente ao intervalo $[0, 1]$. As restantes variáveis deviam assumir um dos valores pertences aos conjuntos que de seguida se apresentam:

- *Learning rate*: [0.1, 0.01, 0.001, 0.0001]
- Função de ativação: [relu, swish, selu, sigmoid]
- Número de *feature maps*: [4, 8, 12, 16]

Em cada iteração deste algoritmo, cada partícula treina a rede neuronal apresentada anteriormente durante 7 épocas, dispondo de 1500 ações por época. No final da época, são jogados 100 episódios que servem como validação. Deste conjunto de episódios são recolhidas métricas como pontuação média, mínima e máxima, sendo que a atualização da posição das partículas é realizada tendo por base a melhor pontuação média obtida na validação da última época de treino da rede.

Devido ao elevado tempo necessário para este processo, optamos por apenas utilizar 5 partículas e realizar 5 iterações. Isto resulta num total de 25 combinações de parâmetros utilizados e testados ao longo da execução do PSO. É de notar que também o número de épocas foi reduzido devido ao tempo despendido. Esta opção foi tomada pois em experiências anteriores verificamos que quando os parâmetros de rede conseguem alcançar bons resultados, estes são visíveis após poucas épocas de treino, dada a simplicidade do cenário.

Uma vez que foram utilizadas poucas partículas e iterações, a vertente social deste algoritmo foi prejudicada, nomeadamente no que diz respeito à convergência para uma solução ótima. Quantas mais iterações mais visível seria o processo de convergência. Por outro lado, quantas mais

partículas mais soluções seriam experimentas. Não obstante, as melhores soluções encontradas por iteração foram as seguintes:

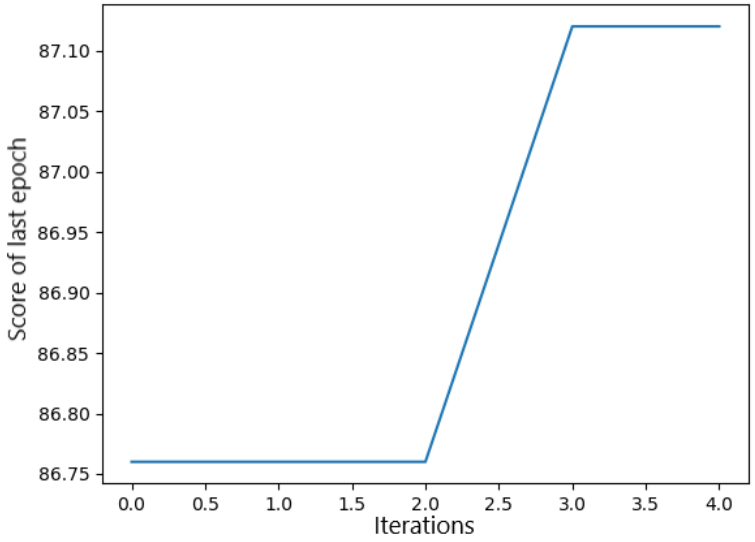

Figura 4: Evolução do algoritmo de PSO

A combinação de parâmetros escolhida foi a seguinte:

- *Learning rate*: 0.001
- Função de ativação: relu
- Número de *feature maps*: 12
- *Discount factor*: 0.408

4 Avaliação de Resultados Obtidos

Uma vez selecionados os parâmetros através do PSO, procedemos a reconstruir a rede utilizada na melhor solução alcançada. Deste vez, no que diz respeito ao período de treino, aumentamos o número de épocas para 10, realizando no final de cada uma 200 jogos de teste.

Em relação à pontuação média alcançada por época, os resultados registados foram os seguintes:

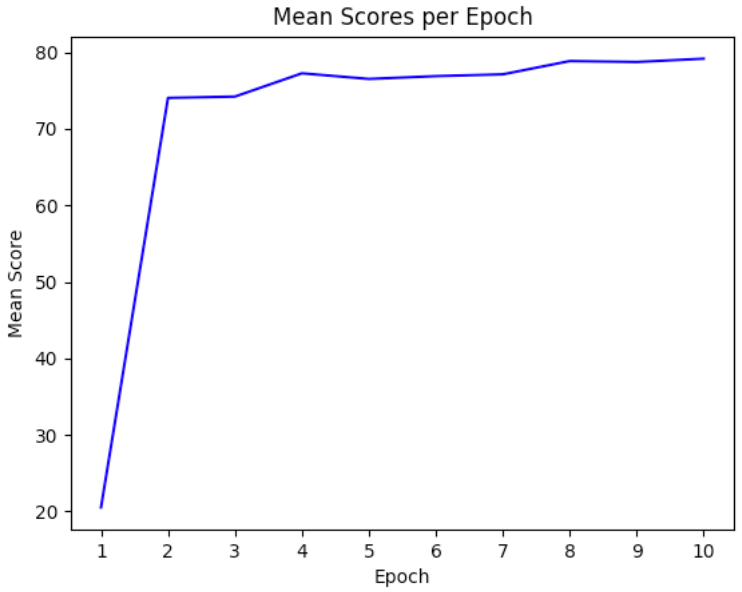

Figura 5: Pontuação média por época

Outra métrica que é interessante analisar, e que comprova que a o agente efetivamente aprende a jogar o cenário, é o número total de episódio concluídos por época. Como o agente apenas dispões de 1500 movimentos por época, quanto melhor for a sua aprendizagem mais episódios conseguirá completar, uma vez que irá minimizar o número de ações desnecessárias, realizando apenas aquelas que são precisas para completar o episódio.

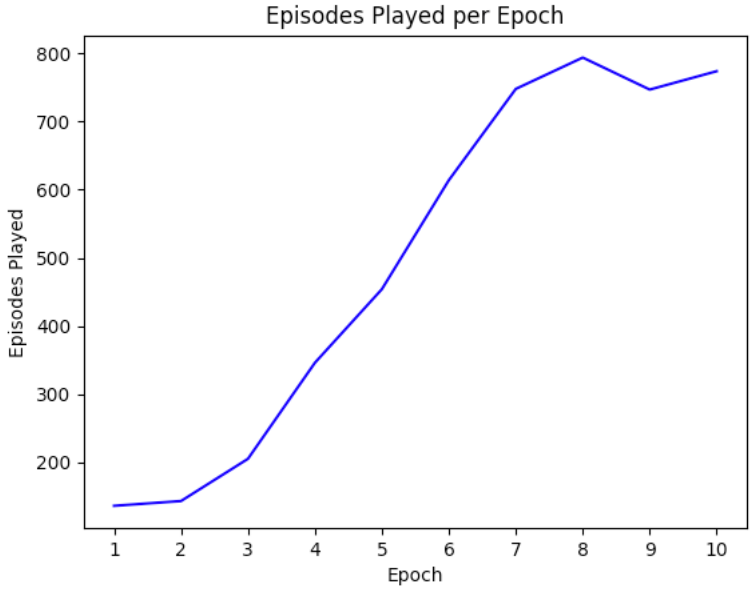

Figura 6: Episódios completos por época

Pela análise dos gráficos anteriores podemos concluir que o agente aprende a interagir corretamente com o ambiente, de acordo com o objetivo pretendido. Além disso, essa aprendizagem é contínua, sendo que à medida que as épocas aumentam, o *score* médio obtido também sofre melhorias.

5 Conclusão e Trabalho Futuro

Dado por concluído o trabalho, surge a necessidade de o contextualizar com a matéria abordada nas aulas de Computação Natural.

A criação de ambientes é uma etapa imprescindível para que o agente possa aprender tendo em conta as suas experiências nestes. No contexto deste trabalho, esta fase tornou-se, principalmente, numa fase de exploração e análise uma vez que o próprio ambiente já é fornecido pelo *ViZDoom*, tendo sido necessário aprender como explorar esta biblioteca. Ainda assim permitiu ao grupo entender o tipo de performance que é esperada pelo agente no ambiente em causa.

A fase de criação do agente é, sem dúvida, a principal fase no que diz respeito à finalidade do trabalho. É aqui que são criados todos os algoritmos e funções que permitem ao agente um bom desempenho. No que diz respeito ao projeto, a utilização de *layers* que permitem o processamento de imagens como *convolutional layers*, foi a novidade mais impactante para o grupo. A par disso, a introdução do mecanismo de *replay memory*, e todo a lógica de treino inerente a este, também foi uma nova experiência.

Segue-se a fase de otimização do modelo e para esta decidimos utilizar *Partice Swarm Optimization*. Como sabemos, PSO permite maximizar funções objetivo e, na realidade, é isso que pretendemos aqui sendo que a nossa função objetivo é o score do agente após realização do treino.

Após a utilização do PSO para *benchmarking* dos parâmetros da rede neuronal, surge a necessidade de avaliar e testar a solução ótima gerada pelo PSO. Podemos verificar que realmente o PSO gerou uma solução extremamente otimizada, já que os valores obtidos pelo agente foram muito bons, revelando assim uma boa performance do agente no jogo.

Relativamente a trabalho futuro, uma possível evolução para o modelo seria a implementação do *double deep reinforcement learning*, no qual existe uma separação entre duas redes, sendo uma para registar os valores de recompensa e outra para realizar as ações, evitando assim a possibilidade de surgir tendências na rede. Quanto ao processo de otimização, poderíamos aplicar o PSO ao segundo cenário de jogo (`defend_the_center`). Outra possibilidade seria aumentar o número de partículas e de iterações do PSO para que a solução fosse ainda melhor mas, devido ao gasto de recursos computacionais que este tipo de processo implica, não foi possível realizá-lo.

Referências

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis, AntonoglouDaan Wierstra, Martin Riedmiller: Playing Atari with Deep Reinforcement Learning (2013). DeepMind Technologies.

[2] ViZDoom: https://github.com/mwydmuch/ViZDoom. Accessed 31 May 2019.

[3] Aswin Raghavan, Jesse Hostetler, Sek Chai: Generative Memory for Lifelong Reinforcement Learning (2019). SRI International, Princeton, NJ, USA.

[4] Parsa Heidary Moghadam: Deep Reinforcement learning - DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay (2019). Accessed 31 May 2019.