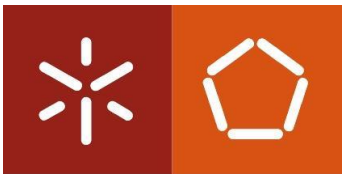

Universidade do Minho

Departamento de Informática

TP2 - DRL aplicado ao jogo DOOM

Bruno Martins (a80410), Catarina Machado (a81047),
Filipe Monteiro (a80229), Jéssica Lemos (a82061)

30 de Maio de 2021

Conteúdo

Resumo

Com bases de conhecimento nas áreas de *Reinforcement Learning* e *Deep Learning* foi proposto a utilização de *Deep Reinforcement Learning* para treinar um *bot* capaz de jogar o famoso jogo *Doom*, num pequeno mapa de ações limitadas.

1 Introdução

O trabalho proposto consiste na criação de uma rede neuronal que complemente um algoritmo de *Reinforcement Learning* aplicado ao jogo DOOM. O algoritmo utilizado foi o de *Q-Learning* que consiste num agente que aprende que ação terá de tomar em certas circunstâncias baseado num valor de recompensa.

O DOOM é um jogo FPS (*First Person Shooter*) que consiste, de forma muito simplificada, em matar monstros de forma a obter a pontuação mais elevada possível.

No contexto deste trabalho foi utilizada a biblioteca *VizDoom* que permite integrar o jogo DOOM com o algoritmo de aprendizagem escolhido.

O cenário proposto para jogar era o *basic*, que apenas consiste em três ações que o agente pode tomar: movimentar-se para a esquerda, direita e disparar. A utilização deste cenário permitiu simplificar o algoritmo de *Q-Learning* pois o número das ações é reduzido.

O objetivo proposto é a obtenção do melhor *score* possível por parte do agente durante o jogo após o seu treino recorrendo a uma *Deep Neural Network* para a obtenção dos valores das recompensas (*Q-Values*).

Desta forma, o presente relatório divide-se na explicação da criação do algoritmo de *Q-Learning* num contexto de *Deep Reinforcement Learning*, na análise de resultados e otimizações executadas e, por fim, uma breve conclusão sobre os resultados e todo o restante trabalho desenvolvido.

2 Criação do Algoritmo de DRL

No que toca ao *reinforcemente learning* foi utilizado a técnica de *Q-Learning* com duas grandes otimizações: *Double Q-Learning* e *Experience Replay Memory*. A implementação desta foi com base numa tabela *q-values* abstraída numa rede neuronal, onde recebia um *stack* de **4 frames** (para dar uma noção de movimento) e calculava, para cada ação, um *q-value* - quanto maior fosse, melhor a ação a tomar. Este seguia ainda uma política *epsilon-greedy* para realizar exploração do ambiente ou executar as melhores ações para determinado estado.

Quanto ao *Double Q-Learning*, é um facto conhecido que quando é utilizada apenas uma rede neuronal para a procura de *Q-Values* existe uma tendência de estimar um valor demasiado elevado. Isto significa que, por vezes, este algoritmo persegue valores que lhe parecem valer mais mas na verdade acaba por se prejudicar a longo prazo. A solução para este problema foi proposta em 2010 por Hado van Hasselt e atualizada mais tarde com uma nova versão. Esta consiste em criar duas redes neuronais para estimar dois valores, um para selecionar a ação a tomar de entre as disponíveis e outro para calcular o valor Q de tomar essa ação no estado seguinte. O Q representa o modelo e o Q' representa o *target*. Podemos traduzir este comportamento na seguite formula:

$$Q^*(s_t, a_t) \approx r_t + \gamma\ Q(s_{t+1}, \text{argmax}_{a'} Q'(s_t, a_t))$$

Figura 1: Fórmula de Double Q-Learnig

Depois os pesos da *target* são actualizados com os pesos da principal (sendo que principal está em constante evolução), a cada **X** episódios.

O *Replay Memory* consiste em guardar a experiência do agente durante o treino, utilizando tuplos. Estes contêm o estado do ambiente *st*, a ação *at* tomada no estado *st*, a recompensa *r(t)* dada ao agente no fim de executar a ação e o estado do ambiente *s(t+1)* no instante após a ação.

$$et = (st, at, r(t+1), s(t+1))$$

Figura 2: Tuplo de *replay*

Neste contexto apenas guardamos 50000 experiências, onde apenas começamos a treinar a rede quando obtivermos no mínimo 5000 destas. Para isto utilizamos uma *deque* - uma lista de tamanho

máximo n onde, na inserção de valores novos, se estiver cheia, retira os valores mais antigos. Para o treino pegamos aleatoriamente em 64 experiências (*batch size*) deste *buffer*, para cada *step* (no máximo 100 por episódio). A razão pela qual é usado este método consiste no facto de tentar que a rede não aprenda com os episódios consecutivos do ambiente. Se isto acontecesse iria existir uma grande correlação entre as amostras e isso iria traduzir-se numa má aprendizagem por parte do modelo. A escolha aleatória já referida estanca essa correlação. Por fim, a cada 5 episódios (hiperparâmetro), a rede *target* é atualizada com os pesos da rede principal.

2.1 Rede Neuronal e Pré-processamento dos *inputs*

Para as redes neuronais, desenvolvemos duas de diferentes complexidades para comparação de resultados:

Rede simples:

- Convolucional com 8 *kernels* de 6x6 com ativação *relu* e *padding*;
- Convolucional com 8 *kernels* de 6x6 com ativação *relu* e *padding*;
- Flatten;
- Dense de 512 nodos com ativação *relu*;
- Dense com 3 nodos com ativação linear.
- Optimizador Adam com *lr* de 0.001 e *loss mean squared error*.

Rede complexa:

- Convolucional com 32 *kernels* de 8x8 e *stride* 2x2 com ativação *elu* e *padding*;
- BatchNormalization com *epsilon* 1e-5;
- Convolucional com 64 *kernels* de 4x4 e *stride* 2x2 com ativação *elu* e *padding*;
- BatchNormalization com *epsilon* 1e-5;
- Convolucional com 128 *kernels* de 4x4 e *stride* 1x1 com ativação *elu* e *padding*;
- BatchNormalization com *epsilon* 1e-5;
- Flatten;
- Dense de 512 nodos com ativação *elu*;
- Dense com 3 nodos com ativação linear.
- Optimizador Adam com *lr* de 0.001 e *loss mean squared error*.

Para *input* estas recebem 4 imagens 84x84 píxeis, juntas (4x84x84), já em preto e branco (opção do *Vizdoom* e normalizadas (valores entre 0 e 1)). Escolhemos 4 imagens para dar uma noção de movimento dos estados nas imagens.

2.2 Comportamento do agente

As escolhas a tomar por parte do agente durante o jogo traduzem-se facilmente em dois conceitos, *exploitation* e *exploration*.

Caso a escolha da ação a tomar seja do tipo *exploitation* significa que o agente vai tomar uma ação tendo em conta os cenários passados de forma a retirar a melhor recompensa possível.

Caso a escolha da ação seja do tipo *exploration* o agente irá tomar uma ação aleatória dentro das possibilidades que tem no momento. No início é mais provável que o agente tome a rota da exploração pois o valor associado a ela (*epsilon*) vai ser elevado, no entanto, à medida que o tempo passa vai havendo uma deterioração desse valor até que a probabilidade de exploração seja mínima. A fórmula utilizada é a seguinte:

$$expl_{probs} = epsilon_{min} + (epsilon_{inicial} - epsilon_{min}) * e^{(} - epsilon_{desc} * step_{decay})$$

, onde o *step decay* é um contador de passos de execução, começando a 0 no ínicio do treino e aumentando 1 unidade a cada *step* executado. No fim, ele fica, no mínimo, com **X**% de probabilidade de explorar.

Um facto que é necessário referir é que os valores de recompensa são todos dados pela biblioteca, não sendo necessária a nossa preocupação quanto esse a ponto.

2.3 Hiperparâmetros e extras

Como hiperparâmetros, visto que não havia grande variação dos resultados na mudança destes (foram testadas os parâmetros usados e testados já em artigos e códigos de outros), decidimos estabilizar com os seguintes:

- Gamma: 0.95
- Epochs: 1000
- *Steps* máximo: 100
- *Batch size*: 64
- Tamanho mínimo do *buffer* de experiências: 1000
- Tamanho máximo do mesmo: 50000
- *Steps* para atualizar a *target network*: 5
- *Epsilon* inicial: 1
- *Epsilon final*: 0.1
- *Epsilon degradation*: 0.00005

Para guardar os resultados obtidos durante o treino, foi utilizado a plataforma *tensorboard*, onde guardámos o histórico de *rewards* por episódio, assim como o *loss* do treino da rede e o estado do *epsilon* ao longo dos episódios.

3 Resultados

Ao fim de umas horas de treino, verificámos que o modelo aprende, mas ainda longe de ser "perfeito". No início o treino é um pouco instável, como seria de esperar da natureza da técnica usada, sendo minimizada com o uso de *Double Q-Learning*, eventualmente estabilizando num valor de *loss* por muit tempo, não convergindo totalmente. Com a leitura de vários *papers*, assim como a aprendizagem de outras técnicas mais recentes (como o *AC3*, *PPO*) verificámos que fazia sentido os resultados obtidos: dado a natureza do *q-learning* (baseado no valor que cada ação tem para

cada estado) e a aleatoriadade existente nos casos de treino, a rede neuronal responsável por calcular os *q-values* para um determinado estado acaba por aprender e "desaprender", levando a estas oscilações, mas que usando métodos como *Double Q Learning* para diminuir esta variância e aumentar a velocidade de convergência e a implementação de um *Experience Replay Memory* para retirar a correlância dos dados de treino, foi o suficiente para o modelo chegar a aprender minimamente o jogo. Se utilizássemos técnicas *policy based* e *value based* como o *A3C* ou o *A2C*, onde o modelo determina a probabilidade de tomar cada ação para cada estado ajustando estes com base num *feedback* por parte de um segundo modelo que obseva a evolução do ambiente, esta convergência seria mais rápida e, para certos problemas onde temos uma quantidade de ações *infinitas* (ou de grande ordem) este cálculo seria mais rápido e menos custoso comparadamente ao tradicional *Deep Q-Learning*. Já para não falar de que, ao contrário do *Deep Q-Learning*, o outros mencionados têm a possibilidade de executarem vários ambientes ao mesmo tempo, obviamente acelerando, e muito, o treino deste.

3.1 Modelo da rede simples

Treino de 500 epochs para o modelo simples, durante 3 horas:

Figura 3: Loss do treino 1, modelo simples.

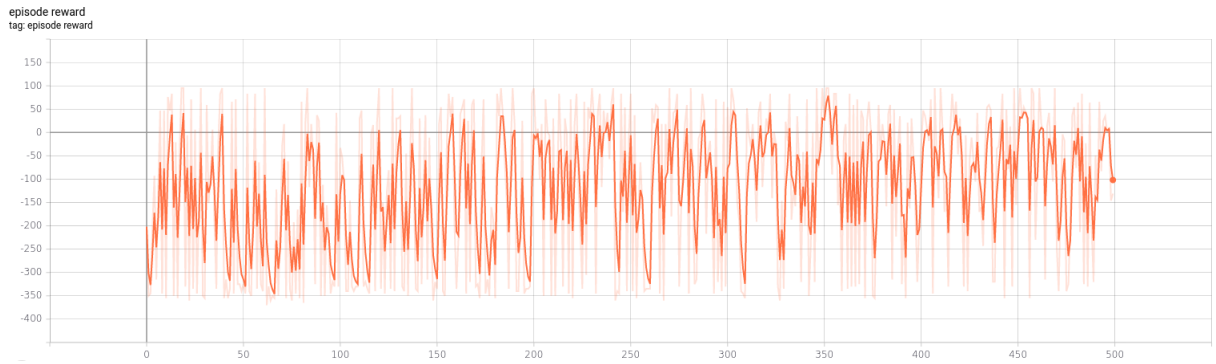

Figura 4: Rewards do treino 1, obtidos durante o treino do modelo simples.

Treino de mais 450 epochs (2,5 horas):

Figura 5: Loss do treino 2, modelo simples.

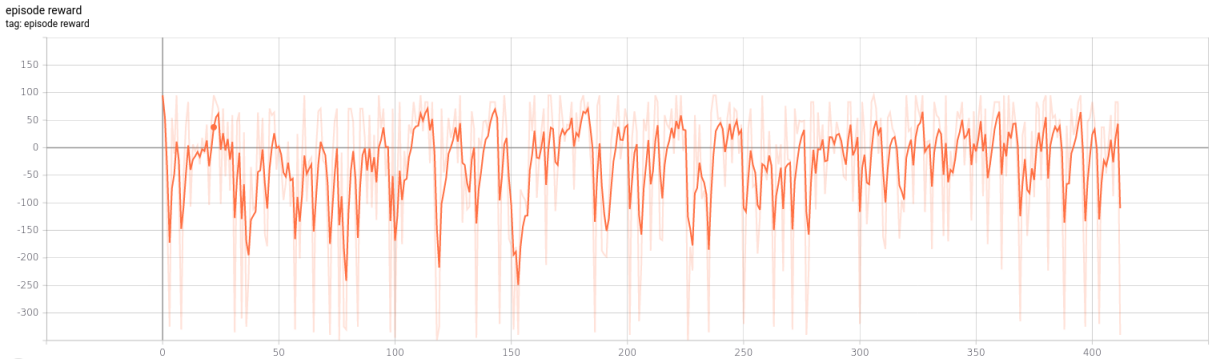

Figura 6: Rewards do treino 2, obtidos durante o treino do modelo simples.

Treino final de mais 1000 epochs (2,5 horas) - total de 2000 epochs:

Figura 7: Loss do treino final, modelo simples.

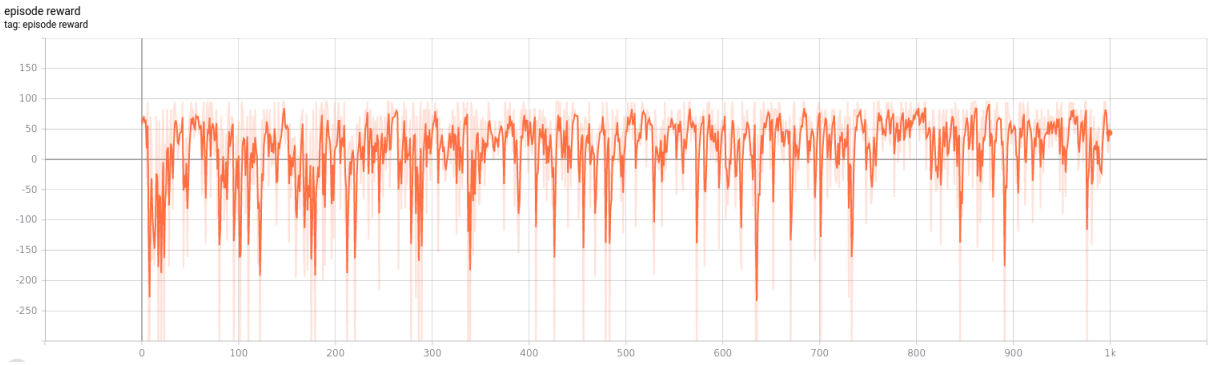

Figura 8: Rewards do treino final obtidos durante o treino do modelo simples.

Se tivêssemos corrido durante mais tempo (11 horas) com mais epochs, provávelmente chegaríamos a este caso:

Figura 9: Loss de um treino exaustivo de um modelo com lapso no número de *outputs*, modelo simples.

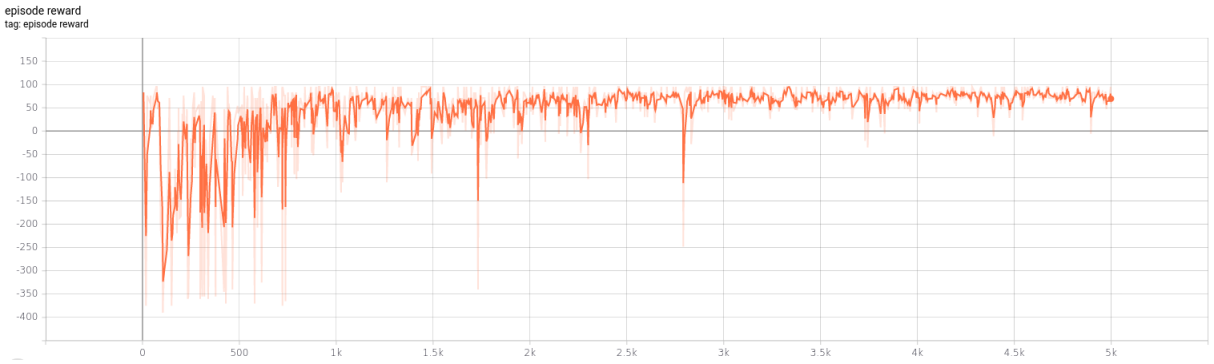

Figura 10: Rewards obtidos durante esse treino exaustivo.

Apesar de ter bons resultados, este treino foi feito numa rede com um lapso no número de outputs, logo não poderia ser utilizado para modelo final desta rede. E, apesar de 5000 *epochs*, verificámos que a *loss* estabilizou, mas os *rewards* ainda não estão no seu melhor, logo assumimos que iria começar a haver uma depressão episódios mais tarde.

3.2 Modelo da rede complexa

Treino de 450 *epochs* para o modelo complexo, durante 3 horas também:

Figura 11: *Loss* do treino 1, modelo complexo.

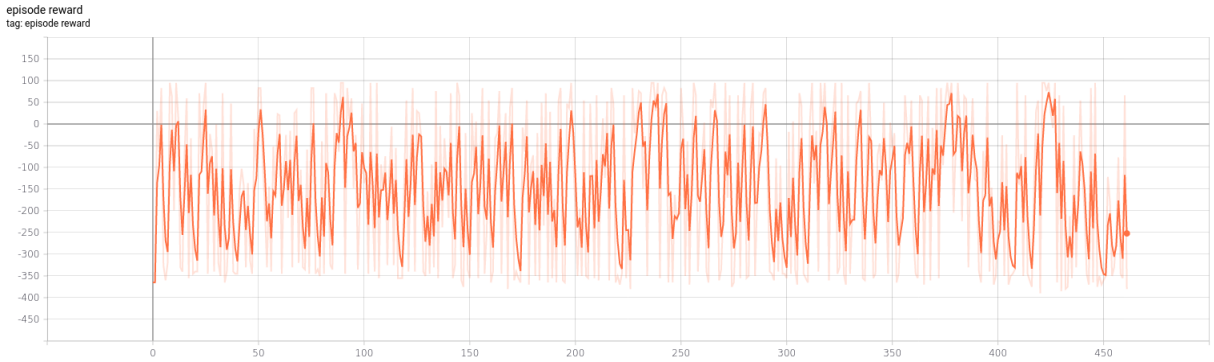

Figura 12: *Rewards* do treino 1, obtidos durante o treino do modelo complexo.

Treino de mais 300 *epochs* para o modelo complexo, durante 3 horas também:

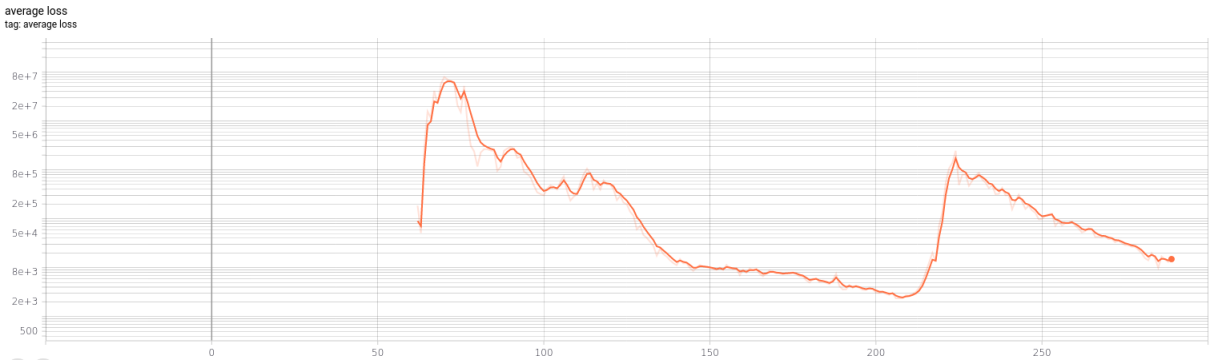

Figura 13: *Loss* do treino final, modelo complexo (resultados em base logarítmica para facilidade da leitura destes valore

.

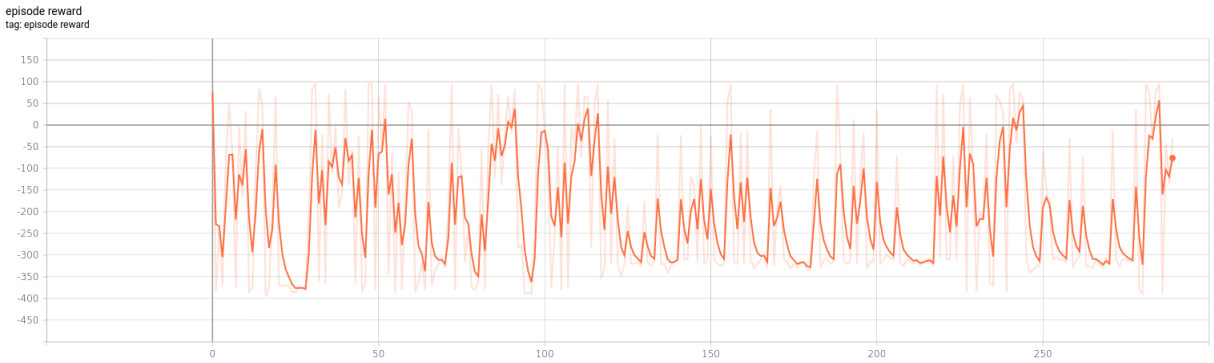

Figura 14: *Rewards* do treino final, obtidos durante o treino do modelo complexo.

3.3 Comparação e avaliação da *performance* de cada modelo

	Jogo 1	Jogo 2	Jogo 3	Jogo 4	Jogo 5
simples	Mean: 14.6 Wins: 8/10	Mean: 77.2 Wins: 10/10	Mean: 60.1 Wins: 9/10	Mean: 59.3 Wins: 9/10	Mean: 55.7 Wins: 9/10
complexo	Mean: -207.3 Wins: 4/10	Mean: -255.7 Wins: 2/10	Mean: -297.1 Wins: 2/10	Mean: -202.6 Wins: 4/10	Mean: -205.0 Wins: 4/10

Tabela 1: Tabela de resultados para cada *bot*, em 5 conjuntos de 10 jogos cada um, apresentando a média de *score* e a quantidade de vitórias.

Analisando cada *bot* a jogar, percebemos que o modelo simples percebeu o funcionamento do jogo: disparar quando vê quando está à frente do inimigo e caso não esteja, andar lateralmente até que fique. E, apesar de falhar às vezes e ficar confuso por momentos, acaba sempre por terminar o jogo com sucesso. Em contra-partida, o outro não aprendeu nada, apenas disparando constantemente podendo acertar ou não (ganhou vários jogos assim, pura sorte). Em comparação com o outro, este precisaria de muito mais treino para chegar a melhores aprendizagens.

4 Conclusão

Investigando esta técnica pioneira de *reinforcement learning* e analisando os novos algoritmos *state of the art* atualmente usados, verificámos uma grande evolução no que toca a aprendizagem de máquinas para realizar específicas tarefas. As claras diferenças de funcionamento entre estas no que toca a *performance* é enorme, assim como os resultados obtidos. Mas que é importante perceber o ambiente em que estas são boas.

Apesar de termos tido bons resultados com o nosso código, o tempo disposto no treino ainda foi longo comparadamente com o necessário para outras técnicas mais recentes. E foi importante perceber que apesar de ter funcionado para este caso em específico, o mesmo poderia não acontecer para outros ambientes (mapas mais complexos, quantidades de ações infinitas, etc.). Exemplos recolhidos *online* de outros indivíduos provam que para este específico mapa era possível criar um *bot* com relativa facilidade, mas para mapas mais complexos seriam semanas para treinar o nosso modelo ao ponto de ter uma ótima *performance*. O próximo passo seria então desenvolvermos técnicas mais recentes para podermos comparar realmente as grandes diferenças entre elas.

Referências

[1] Double Deep Q Network,
https://towardsdatascience.com/double-deep-q-networks-905dd8325412

[2] Introduction to Deep Q-Learning,
https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/

[3] Hado van Hasselt, Arthur Guez and David Silver. *Deep Reinforcement Learning with Double Q-learning*. Google DeepMind, 2016.