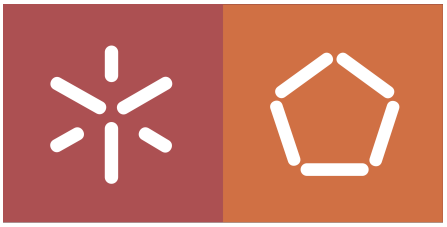

Universidade do Minho
Escola de Engenharia

Deep Reinforcement Learning

Computação Natural
MEI - 4º Ano - 2º Semestre
Grupo 7

PG42818 Carolina Marques
PG42820 Constança Elias
PG42844 Maria Araújo Barbosa
A86271 Renata Ribeiro

6 de junho de 2021

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

Este trabalho foi desenvolvido no âmbito da unidade curricular de Computação Natural e tem como objetivo principal o desenvolvimento de um algoritmo de *Deep Reinforcement Learning (DRL)* para a construção de um *bot* que seja capaz de jogar sozinho o jogo `Breakout` do Atari 2600 (mais concretamente a versão *BreakoutDeterministic-v4*). Para tal, pretende-se criar um ambiente de desenvolvimento *Python* em conjunto com as bibliotecas *Tensorflow* e *OpenAI Gym*.

1.2 Estrutura do Relatório

Este relatório divide-se em oito capítulos:

- O Capítulo 2, corresponde a uma breve análise dos requisitos e especificações deste trabalho face à interpretação do problema;
- No Capítulo 3, apresenta-se uma elaboração do Estado de Arte relativo ao caso de estudo;
- O Capítulo 4 faz uma breve descrição do ambiente do jogo *BreakoutDeterministic*;
- No Capítulo 5, apresentam-se todos os processos realizados no que toca ao algoritmo de DRL: pré-processamento da imagem, componentes de *Q-learning* usadas, a rede neuronal convolucional usada e ainda o processo de treino do algoritmo;
- No Capítulo 6, é explicado o processo de otimização do algoritmo e o que resultou do mesmo;
- Os resultados obtidos ao longo deste projeto são discutidos no Capítulo 7.
- Finalmente, no Capítulo 8, é feita uma breve síntese do trabalho realizado, apresentando as principais conclusões e considerações ao longo do mesmo bem como a proposta de trabalho futuro.

Capítulo 2

Análise e especificação dos requisitos

O presente trabalho tem como objetivos desenvolver um modelo de *Deep Q-Learning*, analisar e validar a *performance* do mesmo. Pretende-se aplicar o conhecimento desenvolvido na unidade curricular de Computação Natural para criar um agente capaz de aprender a jogar o jogo *Breakout* do *Atari 2600*, através de um algoritmo de *Deep Reinforcement Learning*. Para tal, será utilizada a biblioteca *OpenAI Gym* para ter acesso ao ambiente do jogo, cuja versão pretendida é a *BreakoutDeterministic-v4*. É ainda requerido que se otimize o modelo de modo a obter-se a melhor performance possível.

Tendo isso em conta, este trabalho pode ser dividido em duas fases.

- Uma primeira dedicada à formação de um modelo protótipo, onde o agente vai formar uma *Deep QNetwork*. Este modelo receberá como entrada uma sequência de *frames* pré-processadas (por exemplo, 3-5 *frames*) do estado actual do jogo (estado), e vai produzir os valores Q previstos para cada acção;
- Uma segunda fase dedicada ao teste e optimização da *Deep Q-Learning Network*.

Capítulo 3

Estado de arte

O termo "reforço"(*reinforcement*) apareceu pela primeira vez na tradução inglesa de 1927 da monografia sobre reflexos condicionados de Pavlov e traduz-se no poder do comportamento que reage a estímulos (*reinforcers*). A ideia de implementar um sistema de aprendizagem de tentativa-erro num computador apareceu mais tarde, nos inícios da criação de sistemas inteligentes [6].

Na literatura são várias as aplicações de *Reinforcement learning* como método de treino de modelos. Em 1992, Gerald Tesauro [7], da IBM, desenvolveu um programa capaz de jogar sozinho e apreender unicamente por resposta a estímulos. Este programa recebeu o nome de *TD-gammon* e usa um sistema de redes neuronais composto por três camadas e um algoritmo semelhante ao *Q-learning*.

Por sua vez, o algoritmo Q-learning foi proposto por *Watkins* em 1989 e tornou-se uma das melhores opções para aprendizagem baseada em agentes por ser simples e ainda assim conseguir obter bons resultados. Este algoritmo aprende o valor da recompensa que irá receber a longo prazo para cada par de estado-ação *(s,a)*.

Algoritmos de aprendizagem de reforço como o *Q-learning* dependem da exploração total do ambiente para alcançar um comportamento ideal. Assim, com o seu surgimento, foram vários os *papers* a fazerem referência a este algoritmo e a estabelecer uma combinação de técnicas de *deep learning* e *reinforcement learning*, garantindo assim a capacidade de lidar com tarefas mais complexas. Algumas destas tarefas incluíam aprender a partir de dados com diferentes níveis de características e ainda diminuir a necessidade de exploração total do ambiente.

Mnih conseguiu, com sucesso, treinar um agente de *DRL* a partir de imagens com centenas de *pixels* utilizando redes neuronais artificiais para processar os dados sensoriais. Esta abordagem permitiu alcançar capacidades além das humanas em jogos como *Atari* e *Alpha Go*. Em seguida, Van Hassel em [9], melhorou o algoritmo através da implementação de um *deep Q-Learning* duplo que ajuda a gerar estimativas mais precisas, eliminando a sobrevalorização.

Capítulo 4

BreakoutDeterministic-v4

Neste jogo, o terço superior do ecrã é composto por várias camadas de tijolos e o objectivo é destruí-los a todos, fazendo saltar repetidamente uma bola de uma raquete para bater neles. Cada observação obtida a partir deste ambiente produz uma imagem RGB do ecrã, com a forma de matriz de (210, 160, 3). A figura 4.1 apresenta um exemplo de uma *frame* do jogo. Os números apresentados na parte de cima do ecrã representam, respetivamente: o *score* obtido até ao momento (pontuação obtida pelos tijolos distruídos), o número de vidas que restam (o número inicial é 5) e o nível em que o jogo se encontra. Quanto mais para cima estiver o tijolo que foi destruído, maior é a pontuação obtida.

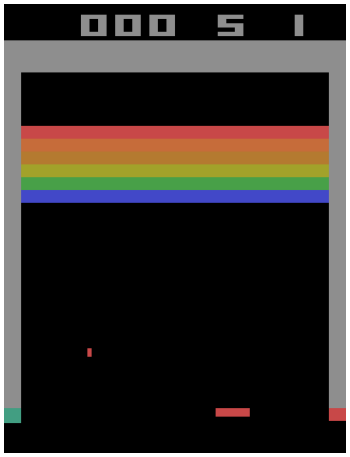

Figura 4.1: Exemplo de uma *frame* do jogo *Breakout*.

As ações possíveis para este jogo são as seguintes:

- 0 - `NOOP`: não fazer nada
- 1 - `FIRE`: disparar a bola no início do jogo
- 2 - `LEFT`: mover o tijolo para a esquerda
- 3 - `RIGHT`: mover o tijolo para a direita

A ação `FIRE`, que é responsável por ativar o lançamento da bola no início do jogo, tem de ser a primeira a ser realizada após ser perdida uma vida. Isto significa que,

num jogo, esta ação tem de ser executada pelo menos 5 vezes (não basta ser apenas no início do jogo). Uma vez que esta ação tem um comportamento semelhante ao *NOOP* (exceto quando se inicia um novo jogo ou uma nova vida), optou-se por descartar a operação *NOOP*, ficando apenas com as ações 1, 2 e 3 no sentido de otimizar, à partida, o processo de aprendizagem. Definiu-se portanto o número de ações como sendo 3 e de cada vez que se executa um passo no ambiente do jogo, basta indicar que a ação real que se pretende executar é `ação + 1`.

Capítulo 5

Algoritmo de *DRL*

5.1 Pré-processamento da Imagem

Como já foi referido anteriormente, as *frames* do jogo possuem um tamanho de 210 x 160 x 3. No entanto, para o modelo aprender o jogo, não precisa deste nível de detalhe pelo que foi necessário fazer *resize* da imagem para 84 x 84, que é o tamanho recomendado no estado de arte. Outro passo consistiu em converter a imagem para a escala de cinzentos, uma vez que esta já permite distinguir os vários elementos que compõem o jogo. Para além disso, a imagem foi também cortada para conter o que realmente importa para o modelo aprender o jogo, excluindo as linhas que apresentam o *score* obtido até ao momento e o número total de vidas. Todos estes passos permitem o obter o conjunto de *frames* que se apresenta na figura 5.1 como exemplo.

Figura 5.1: Exemplo de um conjunto de 4 *frames* pré-processadas

Traduzindo o que foi explicado anteriormente para código, tem-se:

```
def process_frame(frame, shape=(84, 84)):
    frame = frame.astype(np.uint8)

    frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
    frame = frame[34:34+160, :160]
    frame = cv2.resize(frame, shape, interpolation=cv2.INTER_NEAREST)
    frame = frame.reshape((*shape, 1))
    return frame
```

5.2 *Q-Learning*

Será explicado de seguida em que consiste o algoritmo de aprendizagem definido, inspirado no algoritmo que foi fornecido na aula. A figura 5.2 (obtida de [5]) apresenta

um esquema que traduz este algoritmo.

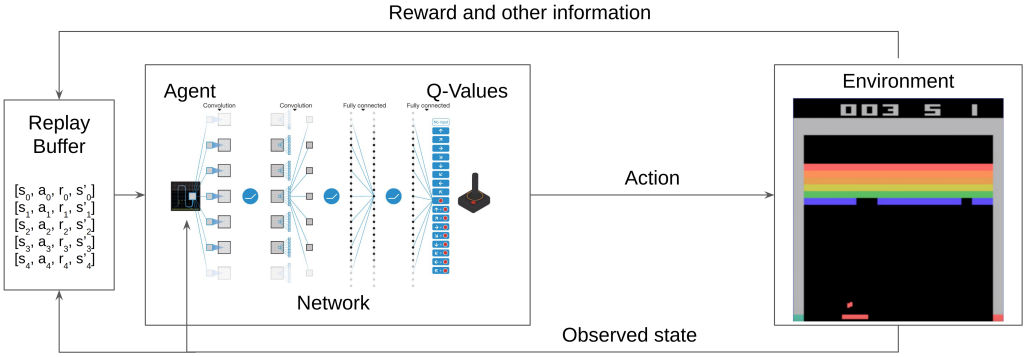

Figura 5.2: Estrutura do algoritmo de aprendizagem

5.2.1 *Epsilon*

De cada vez que é executado mais um passo no jogo, o agente toma uma ação aleatória com probabilidade *epsilon*. Caso contrário, o estado é passado à rede neuronal, e esta toma a acção que prevê que tenha o valor mais alto, tendo em conta o estado. O valor de *epsilon* vai diminuindo linearmente de 1.0 para 0.1 ao longo de um milhão de passos, permanecendo depois em 0.1. Isto significa que no início do processo de treino, o agente procura mais por novas soluções (*exploration*), mas à medida que o treino continua, ele explora as soluções já encontradas (*exploitation*).

5.2.2 *Experience Replay*

Cada passo do jogo é definido pelo tuplo (estado, ação, novo estado, *reward*) num `deque` de tamanho 10000. Isto significa que são guardados os últimos 10000 passos executados anteriormente. Idealmente, este número deveria ser maior. No entanto, ocuparia muita memória e uma vez que os recursos computacionais de que dispomos são limitados, foi necessário optar por um valor menor. A cada iteração a rede treina utilizando um conjunto de 32 *minibatches* independentes.

5.2.3 *Q-values*

Os Q-values são obtidos com base na equação de *Bellman*, que se apresenta na figura 5.3 (obtida de [5]).

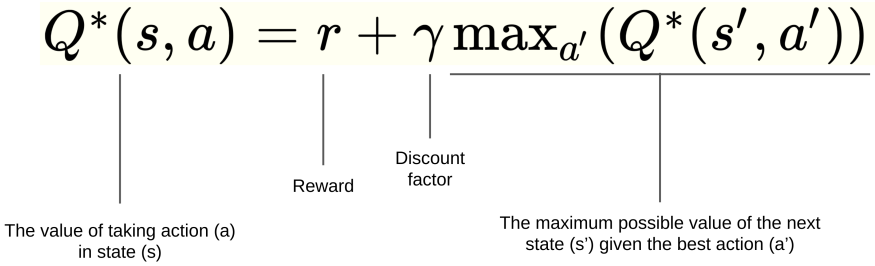

Figura 5.3: Equação de *Bellman*

No presente algoritmo, o `Q-value` apresenta-se da seguinte forma:

```
targets[range(BATCH), action_t] =
reward_t + GAMMA * np.max(Q_sa, axis = 1) * np.invert(terminal)
```

5.2.4 Função de *Loss*

A função de *loss* mais utilizada nos modelos definidos para treinar este jogo no que se refere ao estado de arte é a função *Huber Loss*. Utilizou-se esta função no processo de otimização. Para a definição do modelo inicial, optou-se por usar como função de *loss* a função *MSE*.

5.3 Rede Neuronal Convolucional

Descreve-se agora a arquitetura da CNN definida para aprender o jogo. O *input* da rede neuronal consiste em 4 x 84 x 84 imagens. A primeira *hidden layer* envolve 32 filtros de 8 x 8, com um *stride* de 4 e a função de ativação *ReLU*. A segunda camada envolve 64 filtros de 4 x 4, com um *stride* igual a 2 e mesma função de ativação. A terceira camada envolve 64 filtros de 3 x 3, com *stride* igual a 1 e a mesma função de ativação. A última *hidden layer* é uma *fully-connected layer* e consiste em 512 unidades de retificação. A camada de *output* é uma *fully-connected linear layer* com um único *output* para cada ação válida. A figura 5.4 apresenta a arquitetura desenvolvida para a rede neuroral convolucional sobre a qual se aplicou o algoritmo de *Q-Learning*.

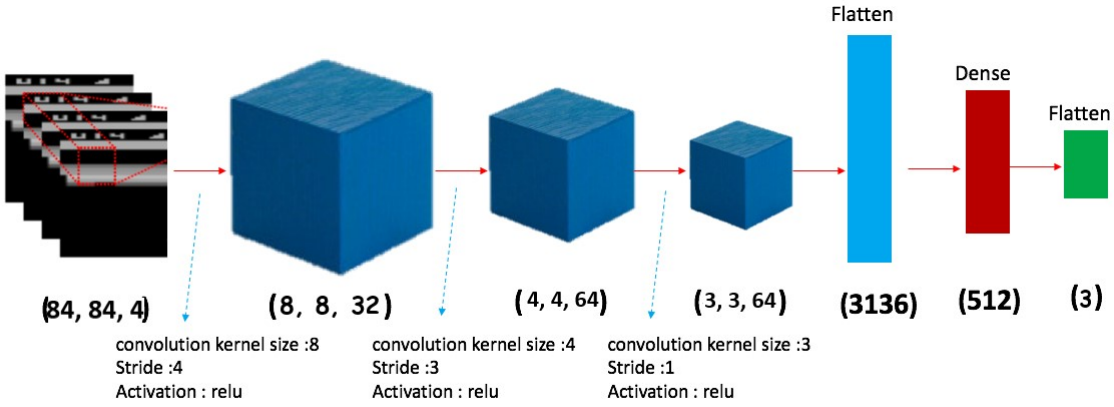

Figura 5.4: Arquitetura da *CNN* definida

5.4 Treino

A definição do modelo para treino do jogo foi inspirado no código fornecido nas aulas para o treino do *Flappy Bird*. O *script* definido pode correr num de três modos (passado como argumento): `Train`, `CTrain` e `Run`. O primeiro modo consiste em treinar a rede neuronal do princípio, o segundo permite continuar o treino da rede já iniciado anteriormente, fazendo o carregamento dos pesos, e o último é utilizado para validar o modelo treinado.

A cada *timestep*, o agente escolhe uma ação, com base no valor de *epsilon*, dá um passo no ambiente, armazena esta transição, escolhe um *batch* de 32 *frames* e utiliza-as para treinar a rede neuronal. Para cada item de treino (s, a, r, s‘) no *minibatch* de 32 *frames*, a rede recebe um estado (s, que é a *stack* de 4 *frames*). Usando o próximo estado (s‘) e a equação de Bellman, obtem-se os alvos para a rede neuronal, e ajusta-se de seguida a sua estimativa para o valor da acção a no estado s, em direcção ao *target*.

Capítulo 6

Otimização do Algoritmo

6.1 Rede Neuronal Convolucional Otimizada

Na segunda fase do desenvolvimento deste trabalho, desenvolveram-se duas redes neuronais com complexidades diferentes da inicialmente apresentada para comparação de resultados.

6.1.1 Rede Otimizada #1

A rede implementada foi inspirada no algoritmo apresentado em [10] e código foi obtido a partir dessa referência. O algoritmo começa por converter a *frame* para uma escala de cinzentos, sendo este suficiente o algoritmo interpretar os objetos da mesma. De seguida, foi recortada a parte superior da *frame*, retirando assim informação que não é necessária para o processamento da mesma e por fim foi efetuado o *resize* das *frames* de tamanho 21x160x3 para 84x84.

A arquitetura do modelo consiste em:

- Convolucional com 32 *kernels* de 8x8 e *stride* 4x4 com ativação relu, inicialização *Variance Scaling* (com *scale* igual a dois);
- Convolucional com 64 *kernels* de 4x4 e *stride* 2x2 com ativação relu, inicialização *Variance Scaling* (com *scale* igual a dois);
- Convolucional com 64 *kernels* de 3x3 e *stride* 1x1 com ativação relu, inicialização *Variance Scaling* (com *scale* igual a dois);
- Convolucional com 1024 *kernels* de 7x7 e *stride* 1x1 com ativação relu, inicialização *Variance Scaling* (com *scale* igual a dois);
- Flatten;
- Dense com 1 *output* e inicialização *Variance Scaling* (com *scale* igual a dois);
- Flatten;
- Dense com 3 *outputs* e inicialização Variance Scaling (com *scale* igual a dois);
- Optimizador Adam com *learning rate* igual a 1e-4 e *Huber Loss function*.

Nesta arquitetura, existem dois fluxos separados para estimar o valor de um estado. Uma *stream* estima o valor de um estado (quão bom é um determinado estado) e a outra *stream* estima a vantagem de realizar uma ação num determinado estado. Ao utilizar uma rede *Dueling Double Deep Q Learning*(DDDQN), esta pode aprender intuitivamente quais os estados mais valiosos, sem ter que aprender o efeito de cada ação naquele estado.

Foi definida a classe **GameWrapper**. Esta classe vai funcionar como uma espécie de cápsula envolvendo o ambiente GYM que tem como objetivo controlar o ambiente em si e o estado que está a ser fornecido ao agente. É através da comparação de *frames* que a rede será capaz de detetar os movimentos no jogo, e para tal, é necessário fornecer à rede um conjunto de 3 a 5 *frames*. Para esta rede, foram escolhidas 4 *frames*.

Uma outra classe definida é a **ReplayBuffer** que será utilizada para armazenar as transições observadas durante o treino e fornece-as para atualizações de parâmetros.

Foi definida ainda a classe *Agent*. Esta classe implementa um agente de DDDQN standard. Os argumentos presentes nesta classe:

- dqn: Um DQN (devolvido pela função DQN) para prever movimentos;
- target_dqn: Um DQN (devolvido pela função DQN) para prever os valores target-q;
- replay_buffer: Um objecto ReplayBuffer para guardar todas as experiências anteriores
- n_actions: Número de acções possíveis para o ambiente em questão;
- input_shape: Tuplo/lista que descreve a forma do ambiente pré-processado;
- batch_size: Número de amostras a retirar da memória de repetição em cada sessão de actualização;
- Entre outros.

Ainda dentro da classe *Agent*, oito funções foram definidas, sendo que entre estas tem-se:

- calc_epsilon: retorna o valor de epsilon apropriado com um dado número de *frames*;
- get_action: com base no estado e a *frame* retorna um *integer* que representa o movimento previsto;
- learn: colhe amostras de um batch e utiliza-as para melhorar o DQN;
- Entre outros.

Com a classe criada é de seguida criado o ambiente e construida a rede principal e a rede objetivo. O modelo é treinado, é escolhida uma ação e executada, guardando de seguida essa experiência no *replay memory*, e, por fim, o agente e a rede objetivo são atualizados. Este ciclo irá decorrer até que o mesmo seja parado.

6.1.2 Rede Otimizada #2

Esta segunda rede baseia-se no código original fornecido, sendo que as diferenças foram o modelo e os hiper-parâmetros usados. A arquitetura do mesmo apresenta-se de seguida.

- Convolucional com 32 *kernels* de 8x8 e *stride* 2x2 com ativação elu;
- *BatchNormalization*
- Convolucional com 64 *kernels* de 4x4 e *stride* 2x2 com ativação elu;
- *BatchNormalization*
- Convolucional com 128 *kernels* de 4x4 e *stride* 1x1 com ativação elu;
- *BatchNormalization*
- Flatten;
- Dense com 3 *outputs*;
- Optimizador Adam com *learning rate* igual a 1e-3;

Hiper-parâmetros

Ao testar com vários parâmetros em diferentes tipos de redes concordou-se que estes otimizaram a performance do modelo:

- *Gamma*: 0.85;
- *Observation*: 200;
- *Explore*: 30000;
- Final *Epsilon*: 0.1;
- *Initial Epsilon*: 1;
- *Replay Memory*: 50000;
- *Batch*: 32;
- *Frame per Action*: 1;
- *Learning Rate*: 1e-4;
- *Episodes*: 10000;

Para além disso, foram guardados os resultados obtidos durante o treino para posterior análise, nomeadamente os valores de *Q_Max*, *Loss* e *Reward*, que foram escritos num ficheiro de texto.

6.2 Treino

Foi desenvolvido ainda um *script* em Python que permite correr e/ou treinar o modelo de DQL com parâmetros que o utilizador necessitar de forma simples, tornando assim possível a otimização de qualquer hiper-parâmetro presente no algoritmo. Para isso o utilizador deve executar o comando *py qlearns_start.py* e indicar o parâmetro que quer utilizar em cada métrica. No final o script executa o ficheiro escolhido pelo *user* com os valores por ele selecionados.

Importa referir que apesar de ser uma funcionalidade importante para este trabalho, a mesma não se encontra funcional para todos os modelos, uma vez que o grupo utilizou o *Google Colab* para realizar o treino.

Segue-se um excerto do mesmo.

```
import os

print("Please provide the necessary data:")
model = input('Model to execute. \n')
mode = input('Mode of the program: Run , Train, CTrain.\n')
episodes = input('Number of episodes.\n')

(...)

## Invocar o modelo com os parametros definidos
os.system("python3 " + model + " " +  mode + " " +
episodes + " " + batch_size + " " +(...) )
```

Capítulo 7

Resultados

Esta secção apresenta os principais resultados obtidos para os modelos definidos. Tendo em conta os fatores de otimização que foram sendo aplicados ao longo do processo de treino (como a redução do número de ações possíveis para 3, a atribuiução de penalizações quando a bola saía fora do *environment*, o pré- processsamento da imagem) esperar-se-ia ter chegado a uma maior pontuação no jogo, ao fim de alguns dias de treino. No entanto, entende-se que estes valores fazem sentido tendo em conta o fator aleatório que existe no processo de treino da rede, pois ao serem calculados os *q-values* existe oscilações no processo de aprendizagem, podendo haver pequenas regressões no mesmo (como se pode ver na figura 7.5). Um exemplo significativo desta regressão pode ser verificado aquando da alteração da função de *loss* no modelo original, numa tentativa de otimizar o processo.

Apresenta-se de seguida os gráficos obtidos para os valores de *rewards*, *q_max* e *loss* nas figuras 7.5, 7.1 ,7.3, respetivamente, ao longo de 10000 *timesteps* para a versão inicial do algoritmo e as figuras 7.2, 7.6 e 7.4 contêm os valores de *rewards*, *q_max* e *loss* para o algoritmo otimizado.

Como explicado anteriormente, o grupo adotou, inicialmente, a estratégia de penalizar o agente sempre que a bola saía fora do ambiente do jogo (atribuindo um *reward* negativo de -1), pelo que os valores de reward oscilavam entre -1 e 1, normalmente. Nesta versão o *reward* máximo atingindo foi 4.

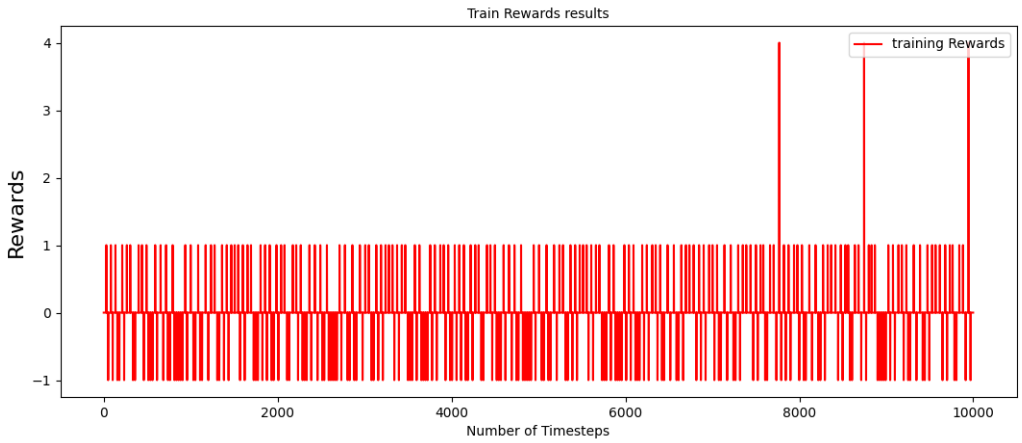

Figura 7.1: Valores de *reward* obtidos

O gráfico 7.2, mostra a variação dos valores de *reward* na rede otimizada 1. Nesta rede o agente não é penalizado e por isso a variação destes valores ocorre sempre num intervalo positivo.

Figura 7.2: Valores de *reward* obtidos na rede optimizada.

Ao contrário da rede anterior, esta apresenta uma elevada variação dos valores de *reward*, atingindo um pico máximo de 80. É possível ainda verificar a existência de um padrão que se repete a cada 4000 episódios, apresentando inicialmente valores de *rewards* baixos que vão aumentando progressivamente até atingir um pico máximo, fazendo o processo reiniciar. Este padrão pode estar relacionado com as competências que vão sendo adquiridas pelo modelo que, à medida que aumentam, tornam o processo de aprendizagem mais complexo.

Relativamente aos valores de *loss*, na primeira versão do modelo, estes começam a diminuir significativamente após 6000 *timesteps*. Tendo treinado durante mais tempo, o grupo prevê que a rede teria tido tempo de convergir e chegar a valores mais baixos.

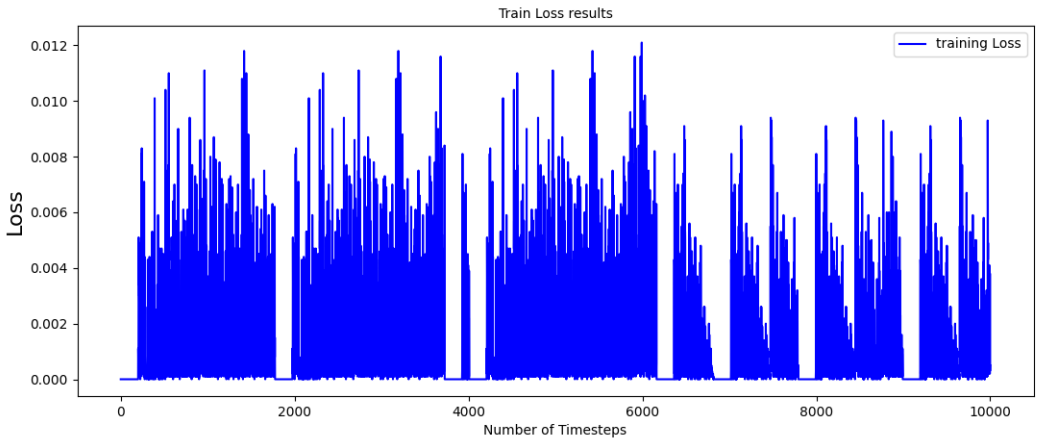

Figura 7.3: Valores de *loss* obtidos

Em relação à versão otimizada, os valores de *loss*, de um modo geral são bastantes superiores.

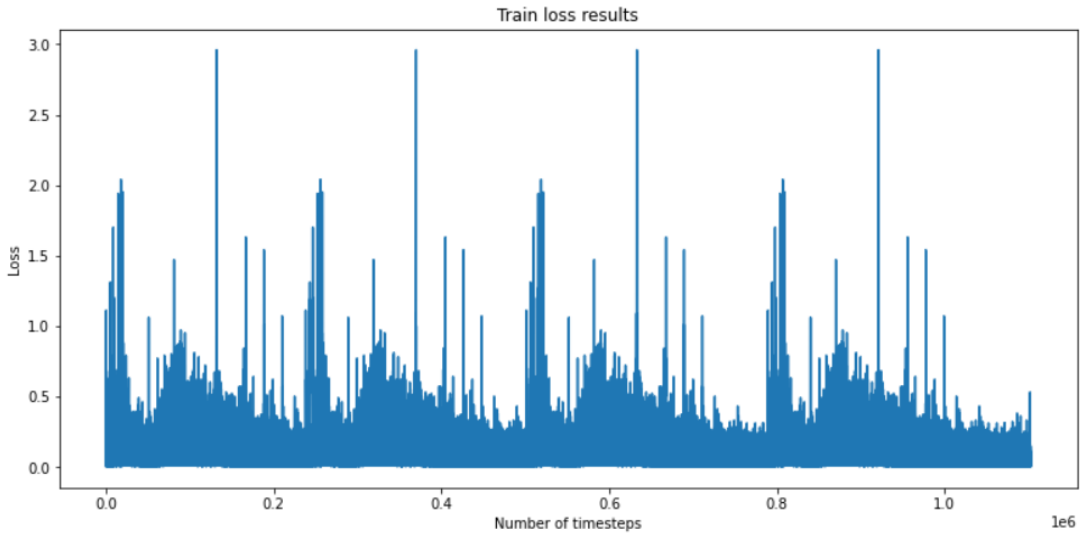

Figura 7.4: Valores de *loss* obtidos na rede optimizada.

Finalmente, os valores de *q_max* obtidos na versão inicial e apresentados na figura 7.5, oscilam entre 0 e 1 mas ao fim de 7000 *steps*, estes aumentam para valores de quatro. Estes valores idealmente deveriam manter-se o mais próximos de zero e não tender a aumentar.

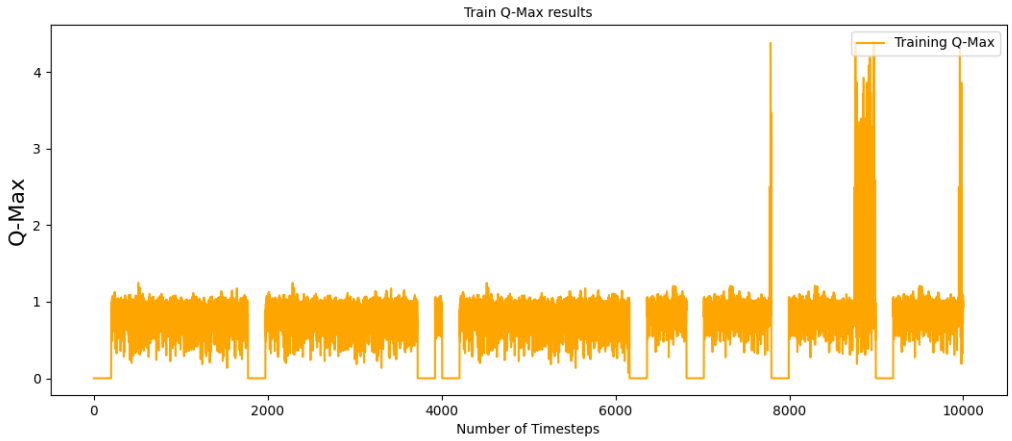

Figura 7.5: Valores de *q_max* obtidos

Já na rede otimizada (figura 7.6), os valores de q_max são mais elevados e concentrados entre cinco e seis com tendência a diminuir. Prevê-se que com um treino mais extensivo estes valores iriam diminuir à medida que o modelo convergisse.

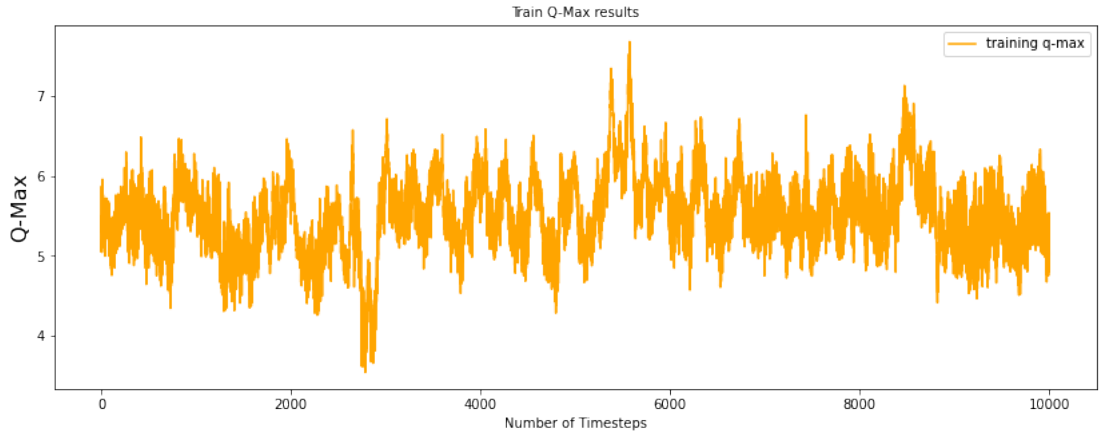

Figura 7.6: Valores de *q_max* obtidos na rede otimizada

Para colmatar os problemas referidos, tentou-se começar a utilizar outras estratégias recomendadas, nomeadamente o método de *Double Q-Learning*. Esta estratégia permite aumentar a velocidade de convergência. Neste caso de estudo, não se chegou a ver os efeitos da aplicação deste método devido ao esgotamento dos recursos de que dispunhamos. Outras técnicas recomendadas que poderiam ser utilizadas seriam por exemplo o A2C ou o A3C.

7.1 Validação

Para o processo de validação, na rede inicial, correu-se o *script* definido em modo `run` e definiram-se algumas funções para poder obter o vídeo da execução do jogo no *Colab*. Ao fim de cerca de 1 dia (com algumas horas de pausa a meio devido às limitações do ambiente de execução), o modelo conseguia obter um *score* de 6. Após 2 dias, chegou aos *11*. Ao fim de 4 dias, atingiu o máximo *score* obtido, 27, valor que se manteve até à data de entrega.

A rede otimizada 1, foi a que apresentou melhores resultados, ao fim de aproximadamente 24 horas de treino atingiu o score máximo de 64.5. Nas restantes horas em que o treino prosseguiu verificou-se uma regressão na aprendizagem e o modelo passou a apresentar valores de score a oscilar no intervalo [40,50].

Capítulo 8

Conclusão

Apesar dos resultados obtidos, importa estabelecer algumas considerações, nomeadamente, o ambiente de execução, *Breakout Deterministic v4*, e a rede otimizada número dois.

Em relação aos resultados obtidos pode-se afirmar que não foram os ideais devido à limitação de recursos computacionais. Uma vez que os elementos do grupo não possuíam computadores com um bom GPU, recorreu-se ao *Google Colab*.

Para executar o *script* definido e explicado anteriormente, recorreu-se ao *Google Colab* de modo a tirar partido do GPU. Alguns problemas advieram da utilização deste ambiente, nomeadamente a limitação de alocação de recursos a cada utilizador. Optou-se por recorrer a algumas soluções disponíveis na *internet*, de modo a tentar colmatar o problema de o *notebook* do *Colab* ficar *"idle"* por não ter atividade (o que acontece após 90 minutos sem interação por parte do utilizador), mas não resultaram. No entanto, apesar dos resultados obtidos, pensou-se que o modelo definido teria potencial para obter bons resultados no jogo se se dispusesse de melhores recursos computacionais, uma vez que acompanham o estado de arte. O grupo considera também que teria sido benéfico aproveitar melhor as duas semanas seguintes ao lançamento do projeto, uma vez que seria mais tempo útil para treino.

Breakout Deterministic tem como característica principal uma *frame skip* de quatro *frames* a cada *step*. Esta característica faz com que os que os modelos não observem tudo o que acontece e, por isso, tenham tendência a aprendem menos. Um solução para este problema seria usar *NoFrameskip*, uma vez que este não avança *frames* de todo.

Para a rede otimizada número dois, não foi possível obter resultados, uma vez que o modelo guardado no ambiente do *Google Colab* ficou corrompido após um dia de treino e, em posteriores tentativas, o mesmo continuava a acontecer e por isto, o grupo não conseguiu efetivamente retirar conclusões sobre o mesmo.

A rede otimizada um, no pouco tempo que o algoritmo correu, apresentou resultados promissores. Durante o decorrer do treino houve uma regressão dos resultados mas ,se existissem os meios computacionais para suportar o algoritmo, este poderia atingir valores bem mais elevadas. No entanto, pelo estudo de vários estados de arte, este tipo de modelo mostrou-se benéfico em problemas como o atual e o grupo considera que este teria potencial como solução ao problema.

A nível de trabalho futuro, seria interessante fazer uma maior exploração dos parâmetros a otimizar, bem como usar outros algoritmos de *Deep Reinforcement Learning*, de modo a avaliar a sua *performance* neste tipo de jogos, como por exemplo Actor Critic. O algoritmo A3C (Asynchronous Advantage Actor-Critic) é um dos algo-

ritmos mais reconhecidos e que apresenta uma melhor performance particularmente em domínios como a *Atari*. Um outro algoritmo mencionado em alguns estados de arte é o *Proximal Policy Optimization* (PPO), cuja ideia fundamental é atualizar a política diretamente para tornar mais provável a probabilidade de tomar ações que proporcionem uma recompensa futura maior.

Em jeito de conclusão, o grupo considera ter cumprido os requisitos do projeto apesar dos resultados obtidos e que o trabalho desenvolvido reflete não só o trabalho desenvolvido no decorrer da disciplina, mas também os objetivos delineados para o mesmo de uma forma fidedigna.

Bibliografia

[1] https://cs.stanford.edu/ rpryzant/data/rl/paper.pdf

[2] https://github.com/dani-amirtharaj/Deep-Q-Learning

[3] http://www.pinchofintelligence.com/openai-gym-part-3-playing-space-invaders-deep-reinforcement-learning/

[4] https://becominghuman.ai/playing-atari-using-reinforcement-learning-9fe52fd4f262

[5] https://medium.com/analytics-vidhya/building-a-powerful-dqn-in-tensorflow-2-0-explanation-tutorial-d48ea8f3177a

[6] Sutton, R., Barto, A. (2018) Reinforcement Learning: An Introduction, 2nd ed., Cambridge, MA : The MIT Press

[7] Gerald Tesauro. Temporal difference learning and td-gammon. Communications of the ACM, 38(3):58–68, 1995.

[8] Human-level control through deep reinforcement learning. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Nature. 2015 Feb 26; 518(7540): 529–533. doi: 10.1038/nature14236.

[9] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double Q-Learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press, 2094–2100

[10] Building a Powerful DQN in TensorFlow2.0, Sebastian Theller