# Computação Natural

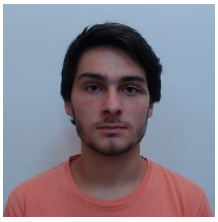
Trabalho Prático 2

Grupo 1

6 de setembro de 2021


Ana Gil
A85266

Beatriz Rocha
A84003



Hugo Matias
A85370

João Abreu
A84802

# Conteúdo

# Lista de Figuras

# 1 Introdução

## 1.1 Contextualização

A implementação de um algoritmo capaz de jogar jogos de modo idêntico a um ser humano experiente ainda é um desafio para a computação atual, principalmente no que toca ao modo como um programa adquire o conhecimento. Neste sentido, a estratégia passa, normalmente, por utilizar métodos de *Machine Learning* que, ao invés de desenvolver um programa com um objetivo específico, muitas vezes incluindo o conhecimento do próprio desenvolvedor (ou de jogadores experientes, neste contexto), o computador é ensinado a aprender sozinho. Assim, existem dois métodos de *Machine Learning* que podem ser úteis para este caso: *Supervised Learning* e *Reinforcement Learning* (RL).

Para a utilização de *Supervised Learning* é necessário ter uma grande quantidade de exemplos de soluções que atinjam o objetivo do problema para, posteriormente, desenvolver um algoritmo que aprenda e imite esse mesmo comportamento. No entanto, ao invés de aprender por imitação, um iniciante, normalmente, aprende sozinho explorando o ambiente, testando como o jogo reage a determinadas ações, *etc.* Esta lógica é obtida através de *Reinforcement Learning* que vai ser o método adotado para a resolução do problema retratado neste relatório. Este, para que a máquina faça o que o programador deseja, irá receber recompensas ou penalidades pelas ações que executa, sendo o seu objetivo maximizar a recompensa total.

## 1.2 Caso de Estudo

Neste projeto, pretende-se aplicar a Inteligência Artificial numa ótica de resolução de problema, ou seja, criar um jogador inteligente. O jogo em questão é o Breakout do Atari 2600, título antigo que, assim, permitirá máquinas mais modestas criar agentes inteligentes que serão utilizáveis em situações de maior complexidade.

Para simular o ambiente de jogo, tal como proposto pelo enunciado, será utilizada a biblioteca **OpenAI Gym** que, por sua vez, permite simular um grande número de ambientes de *Reinforcement Learning*, incluindo jogos Atari.

Neste ambiente, uma camada de tijolos reveste o terço superior da tela e o objetivo consiste em destruí-la com recurso a uma bola e uma raquete sem deixar a primeira cair. Cada observação obtida neste ambiente produz uma imagem **RGB** da tela, com um formato de *array* igual a **(210, 160, 3)**.

# 2 Fundamentação Teórica

## 2.1 *Markov Decision Processes* (MDPs)

Para o nosso propósito, *Reinforcement Learning* é sobre como resolver Processos de Decisão de Markov (do inglês, *Markov Decision Processes*). Um MDP é simplesmente uma maneira formal de descrever um jogo usando os conceitos de **estados**, **ações** e **recompensas**.

### 2.1.1 Estados

O **estado** é a situação atual em que o agente se encontra. A aproximação mais simples de um estado é, essencialmente, o *frame* atual. Infelizmente, isto nem sempre é suficiente: observando a imagem abaixo, não conseguimos dizer se a bola está a subir ou a descer.

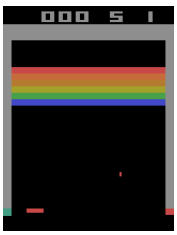


Figura 2.1: Representação do estado

Embora possa parecer inofensivo, na verdade é muito importante sabermos a direção da bola, porque esta representação do estado quebra a propriedade do MDP que afirma que a história de estados e ações passadas é irrelevante, ou seja, não deve haver nenhuma informação útil em estados anteriores para que a propriedade de Markov esteja satisfeita. No caso de usar uma única imagem como nosso estado, estamos a quebrar a mesma, porque os *frames* anteriores poderiam ser usados para inferir a velocidade e a aceleração da bola e da raquete.

Um truque simples para lidar com isso é, simplesmente, trazer um pouco da história anterior para o estado atual (o que é perfeitamente aceitável sob a propriedade de Markov). Para isso, será, então, necessário juntar os quatro *frames* anteriores.

### 2.1.2 Ações

Uma **ação** é um comando que se pode dar no jogo na esperança de alcançar um certo estado e recompensa. No caso dos jogos Atari, as ações são todas enviadas por meio de um *joystick*.

Figura 2.2: *Joystick* Atari

Podem ser realizadas um total de 18 ações com o *joystick*: não fazer nada, pressionar o botão de ação, ir numa das 8 direções (para cima, para baixo, esquerda e direita, bem como nas 4 diagonais) e ir em qualquer uma dessas direções enquanto se pressiona o botão. No Breakout, apenas 4 ações se aplicam: não fazer nada (ação 0), "pedir uma bola"no início do jogo pressionando o botão (ação 1) e ir para a direita (ação 2) ou esquerda (ação 3).

É importante referir que o número de estados possíveis é muito maior do que o número de ações possíveis, o que é uma boa notícia, uma vez que lidar com um grande conjunto de estados é mais fácil do que com um grande conjunto de ações.

### 2.1.3 Recompensas

O último componente dos nossos MDPs são as recompensas. As recompensas são dadas após a execução de uma ação e, normalmente, dependem do seu estado inicial, da ação executada e do seu estado final. O objetivo do programa de *Reinforcement Learning* é **maximizar as recompensas a longo prazo.**

No caso do Atari, as recompensas correspondem simplesmente às mudanças na pontuação, ou seja, cada vez que a pontuação aumenta, o jogador recebe uma recompensa positiva e vice-versa se a pontuação diminuir (o que deve ser muito raro).

Na prática, os algoritmos de *Reinforcement Learning* nunca otimizarão para recompensas totais *per se*; em vez disso, otimizarão para recompensas totais com desconto. Noutras palavras, escolheremos um número $\gamma$, onde $0 < \gamma < 1$ e, em cada etapa no futuro, otimizaremos para $r_0 + \gamma * r_1 + \gamma^2 * r_2 + \gamma^3 * r_3...$ (onde $r_0$ é a recompensa imediata, $r_1$ é a recompensa daqui a um passo, *etc.*). Foram fornecidas muitas justificações na literatura de RL (analogias com taxas de juros, o facto de termos uma vida útil finita, *etc.*), mas talvez a maneira mais simples de ver como isto é útil é pensar sobre todas as coisas que podem correr mal sem desconto: com desconto, a soma de recompensas é garantida como finita, enquanto sem desconto pode ser infinita. Recompensas totais infinitas podem criar uma série de questões, por exemplo como é que uma pessoa escolhe entre um algoritmo que obtém +1 a cada etapa e outro que obtém +1 a cada 2 etapas? A resposta pode parecer óbvia, mas, sem desconto, ambos têm uma recompensa total infinita e são, portanto, equivalentes. A taxa de desconto certa costuma ser difícil de escolher: se for muito baixa, o nosso agente colocar-se-á em dificuldades a longo prazo por causa de recompensas imediatas baratas. Se

for muito alto, será difícil para o nosso algoritmo convergir, porque é preciso ter em consideração muito do futuro. Para este jogo e com base em [1], usaremos 0,99 como a nossa taxa de desconto.

## 2.2 *Reinforcement Learning*

Neste trabalho, iremos considerar um algoritmo designado por *Q-Learning*. *Q-Learning* é, provavelmente, o algoritmo de *Reinforcement Learning* mais importante e conhecido. Nesta secção, iremos explicar o mesmo, mas, primeiro, teremos de clarificar o que são políticas.

### 2.2.1 Políticas

As políticas são o resultado de qualquer algoritmo de *Reinforcement Learning*. Estas simplesmente indicam que ação tomar para qualquer estado (*i.e.* uma política pode ser descrita como o conjunto de regras do tipo "se eu estiver no estado A, tomo a ação 1, se eu estiver no estado B, tomo a ação 2, *etc.*").

Uma política é designada determinística, se nunca envolve "atirar uma moeda ao ar"para decidir a ação em cada estado e é designada ótima, se segui-la devolver a maior recompensa com desconto esperada de qualquer política.

Em MDPs, existe sempre uma política determinística ótima, ou seja, é sempre possível encontrar uma política determinística que é melhor do que qualquer outra política (mesmo que o próprio MDP não seja determinístico).

### 2.2.2 Função $Q$

Finalmente, podemos apresentar a função $Q(s, a)$ que devolve o valor total descontado de tomar a ação $a$ no estado $s$. $Q(s, a)$ é simplesmente igual à recompensa que se obtém por tomar a ação $a$ no estado $s$ mais o valor descontado do estado terminal $s'$, sendo que o valor de um estado é simplesmente o valor de tomar a ação ótima nesse estado, *i.e.* $max_a(Q(s, a))$, portanto temos:

$$Q(s, a) = r + \gamma * max'_a(Q(s', a')) \tag{2.1}$$

Na prática, com um ambiente não determinístico, é possível ter uma recompensa e um próximo estado diferentes de cada vez que se tomar a ação $a$ no estado $s$. Contudo, isto não é um problema, uma vez que se pode simplesmente usar a média (isto é, o valor esperado) da equação (2.1) como a função $Q$.

No nosso caso, saber a função $Q$ ótima devolve-nos a política ótima. Em específico, a melhor política consiste em, em cada estado, escolher a ação ótima, ou seja:

$$\pi(s) = argmax_a(Q(s, a)) \tag{2.2}$$

, onde $\pi(s)$ é a política no estado $s$.

# 3 Metodologia adotada

## 3.1 Pré-processamento

Segundo [2], trabalhar diretamente com os *frames* Atari que, como já foi referido anteriormente, são imagens de 210 x 160 pixeis com uma paleta de 128 cores, pode ser computacionalmente exigente.

Assim sendo, começamos por converter os *frames* coloridos para uma escala a preto e branco. De seguida, cortamos uma região de 84 x 84 pixeis da imagem, visto que usamos uma implementação GPU de convoluções 2D que requer imagens quadradas. Finalmente, garantimos que os dados são armazenados no tipo *uint8*, uma vez que armazenar todos estes *frames* em memória é bastante custoso e este é o tipo mais pequeno disponível.

## 3.2 Modelo

### 3.2.1 *Deep Q-Learning*

O nosso algoritmo irá jogar guardando o estado inicial, a ação que foi tomada, a recompensa que obteve e o estado que atingiu a cada passo e estes dados serão usados para treinar a rede neuronal.

A rede neuronal irá receber um estado inicial, ou seja, uma pilha de quatro *frames* pré-processados e irá devolver a sua estimativa de $r+\gamma*max'_a(Q(s',a'))$, onde $r$ é a recompensa que foi obtida durante o jogo, $\gamma$ é a nossa taxa de desconto (0,99) e $Q(s',a')$ resulta de prever a função $Q$ para o próximo estado usando o modelo atual, ou seja, de certa forma, a nossa rede neuronal está a tentar prever o próprio resultado.

Apesar de isto poder parecer estranho, a boa notícia é que se o algoritmo convergir, tenderá a convergir para o valor correto da função $Q$. A má notícia é que não existe uma garantia de que o algoritmo vá convergir de todo.

Outro aspeto que também pode parecer estranho é o facto de a rede neuronal receber um estado, visto que a função $Q(s,a)$ recebe um estado e uma ação. Uma maneira de abordar isto é tendo um resultado por ação, o que, neste caso, é possível, porque temos relativamente poucas ações, ou seja, nesta abordagem, os resultados correspondem aos Q-valores previstos da ação individual para o estado recebido, o que, por sua vez, se traduz num menor tempo de execução, uma vez que é possível calcular Q-valores para todas as ações possíveis num determinado estado com apenas uma única passagem pela rede. [2]

Este método é excelente para prever qual ação tomar a qualquer momento, mas torna a tarefa de treinar a rede um pouco mais complicada: a cada passo, apenas observamos o resultado de uma única ação, portanto, para podermos ter um valor para cada ação, fazemos com que a rede multiplique os seus resultados por uma "máscara"correspondente à ação codificada com o método *one-hot*, o que irá colocar todos os seus resultados a 0 exceto para a ação efetivamente tomada. Assim sendo, quando queremos prever para todas as ações, podemos simplesmente passar uma máscara com todos os valores a 1.

Posto isto, temos tudo para poder construir a nossa função de *fit*.

### 3.2.2 Definição do Modelo

Inicialmente, são criadas duas *input layers*: uma alusiva aos *frames* cujo formato, tal como seria de esperar, é 84 x 84 x 4 e uma alusiva às ações cujo formato é 3 (número de ações possíveis que o agente pode tomar). A primeira é normalizada para facilitar o processo e, de seguida, o resultado é passado a duas *convolutional layers*: a primeira possui 16 *filters*, *kernel_size* igual a (8, 8), *strides* igual a (4,4) e *activation* igual a ReLU e a segunda é semelhante, à exceção dos *filters*, *kernel_size* e *strides* que passam para 32, (4,4) e (2,2), respetivamente. Depois, o resultado da convolução é achatado com recurso a uma *flatten layer* e passado a uma última *hidden layer* que, por sua vez, é *fully-connected* e possui 256 *units* com *activation* igual a ReLU. Por fim, chegamos à *output layer* que também é *fully connected* e possui 3 *units*. Esta última camada vai ser filtrada, sendo este processo caracterizado pela multiplicação da mesma pela *input layer* alusiva às ações, o que garante que apenas é alterado o valor de tomar uma determinada ação num determinado estado e não há interferência com os valores das outras ações.

## 3.3 *Experience Replay*

*Experience replay* é o principal método que evita que a nossa rede divirja e a estratégia por detrás é bastante simples: em cada iteração, joga-se um passo no jogo, mas, em vez de atualizar o modelo com base nesse último passo, adiciona-se toda a informação relevante do passo acabado de tomar (estado atual, próximo estado, ação tomada, recompensa e se o próximo estado é o final) a uma memória de tamanho finito (*replay memory*) e, depois, executa-se a função de *fit* numa amostra dessa memória. É importante mencionar que, antes de realizar qualquer iteração na rede neuronal, preenchemos a memória com uma política aleatória até um certo número de elementos.

A razão pela qual *experience replay* é útil tem a ver com o facto de, em *Reinforcement Learning*, estados sucessivos serem altamente semelhantes, o que significa que existe um alto risco de que a rede se esqueça completamente sobre como é estar num estado que já não vê há algum tempo, logo este método previne isso, mostrando *frames* antigos à rede.

## 3.4 *Exploration vs Exploitation*

Um problema típico de *Reinforcement learning* consiste em saber quanto tempo o agente deve passar a explorar a sua política conhecida (*exploitation*) e quanto tempo deve passar a explorar novas e possivelmente melhores ações (*exploration*).

É claro que, enquanto a nossa *Deep Q-Network* (DQN) não for muito boa, queremos explorar ações que não seriam tomadas, caso contrário esta seguirá o que quer que a nossa inicialização de pesos aleatória lhe tenha dito. Por outro lado, especialmente quando a nossa rede se tiver tornado melhor, não queremos continuar a seguir uma política aleatória eternamente, porque caso contrário a nossa rede não aprenderá a comportar-se de maneira ótima nas etapas mais avançadas do jogo, algo que pode nunca chegar a atingir com uma política aleatória.

Este problema tem uma solução aparentemente óbvia: às vezes devemos escolher ações aleatórias, para que o nosso modelo possa aprender mais acerca da ação que (ainda) não acha ser ótima e outras vezes devemos escolher ações de acordo com o nosso modelo, para que possamos avançar no jogo e aprender sobre estados mais avançados.

Esta simples estratégia é conhecida como estratégia $\epsilon$-guloso e funciona da seguinte forma:

$$Ação\ a\ tomar = \begin{cases} melhor\ ação\ atual, & \text{com probabilidade 1 - } \epsilon \\ qualquer\ ação, & \text{com probabilidade } \epsilon \end{cases} \tag{3.1}$$

, ou seja, com probabilidade $\epsilon$ escolhemos uma ação aleatória, caso contrário escolhemos a ação gulosa (aquela que tem o Q-valor máximo para aquele estado).

É comum começar com um $\epsilon$ alto e reduzir o seu valor à medida que a DQN sofre mais iterações. No nosso caso, o valor irá reduzir de forma linear, ou seja, o valor de $\epsilon$ irá decair de 1 até 0,1 durante 1000000 de iterações, sendo que, depois, se irá manter constante, tal como se pode ver na Figura 3.1.

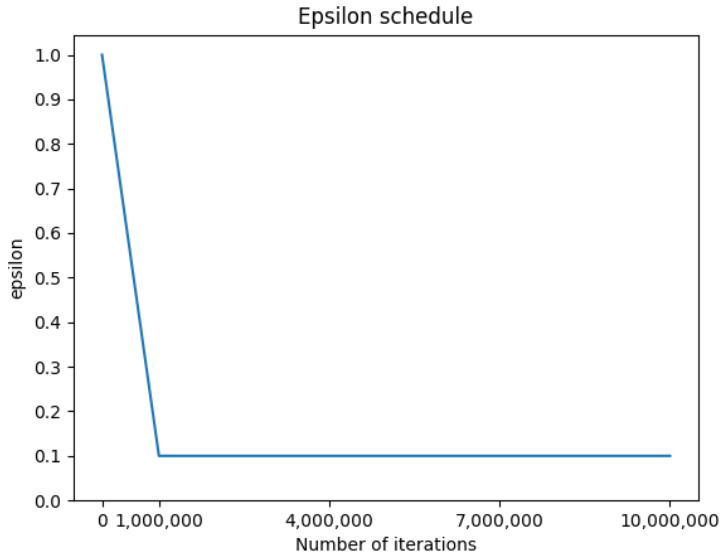



Figura 3.1: Decaimento do $\epsilon$

## 3.5 Algoritmo

**Algorithm 1** deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **for** t=1, T **do**
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition ($\phi_t$, $a_t$, $r_t$, $\phi_{t+1}$) in D
        Sample random minibatch of transitions ($\phi_j$, $a_j$, $r_j$, $\phi_{j+1}$) from D
        Set $y_i = \begin{cases} \mathrm{r}_j, & \text{if episode terminates at step j+1} \\ \mathrm{r}_j + \gamma * max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
        Every C steps reset $\hat{Q} = Q$

Finalmente, podemos juntar todos estes conceitos para construir o nosso algoritmo. De notar que, mais uma vez, este foi inspirado em [1].

Em primeiro lugar, inicializamos a nossa *replay memory* com capacidade N, para podermos armazenar tuplos de experiência (estado atual, ação tomada, recompensa recebida e estado seguinte). De seguida, definimos o nosso modelo e o nosso modelo alvo que, por sua vez, é usado para criar o alvo da nossa rede. Antes de inicializarmos a nossa sequência, tomamos um valor aleatório entre 1 e 30 e não fazemos nada durante esses primeiros passos, o que melhora o modelo, na medida em que o seu estado inicial será sempre diferente. Depois, esse estado inicial é pré-processado e empilhado quatro vezes. De acordo com a estratégia $\epsilon$-guloso, escolhemos a ação a tomar que, por sua vez, é passada ao ambiente. Depois, pré-processamos a observação que o ambiente nos devolve e voltamos a empilhá-la no nosso histórico. A seguir, pegamos nessa transação e armazenamo-la na nossa *replay memory*. De seguida, verificamos se já existem mais de 50000 transações na nossa *replay memory*, se sim podemos chamar a nossa função de *fit* que pega num conjunto aleatório de 32 transações e o modelo prevê o valor de tomar a melhor ação no próximo estado. Se o próximo estado for o final, devolvemos uma recompensa negativa ao nosso alvo por perder (uma vez que o OpenAI Gym não devolve uma recompensa negativa por perder, apenas devolve uma recompensa igual a 0 e nós queremos garantir que o agente sabe que perder é mau), caso contrário o nosso alvo irá receber a recompensa mais $\gamma$ mutiplicado pela previsão do modelo. Depois, usando esse alvo, iremos treinar o nosso modelo. Por fim, depois de chamarmos a nossa função de *fit*, a cada 10000 passos iremos atualizar os pesos do nosso modelo alvo e igualá-los aos pesos do nosso modelo.

# 4 Otimizações

## 4.1 *Target Networks*

Tal como explicado anteriormente, no algoritmo *Deep Q-Learning*, a função $Q$ é uma definição recursiva, ou seja, o valor de $Q(s, a)$ é obtido através do valor de $Q(s', a')$. Posto isto, estamos constantemente a atualizar os valores de $Q(s, a)$ e $Q(s', a')$, até que se aproximem do alvo da função $Q$. No entanto, à medida que atualizamos estes valores também o valor do alvo tem tendência a atualizar, fazendo com que seja difícil para a nossa rede convergir.

Para tornar o processo de treino mais estável, decidimos implementar um modelo alvo. Isto implica termos uma cópia do valor da função $Q$, sendo esta constante de modo a obter um alvo estável para conseguir aprender com um número fixo de passos. No entanto, é de notar que, através do treino, a função $Q$ alvo também tem tendência a melhorar, por isso, em vez de usarmos apenas uma cópia durante toda a fase de treino, optou-se por, a cada 10000 passos, fazer uma cópia do nosso modelo e, posteriormente, calcular o alvo da função $Q$, ao invés de calcular com o modelo atual.

A estratégia adotada para fazer a cópia do modelo foi simplesmente utilizar a função `clone_model` disponibilizada pelo Keras.

## 4.2 *Huber Loss*

A segunda otimização implementada no nosso modelo foi a *huber loss*. Esta consiste num híbrido que utiliza propriedades de *Mean Squared Error* (MSE) e de *Mean Absolute Error* (MAE).

O MSE tem a propriedade de atribuir um valor de importância alto a erros grandes relativamente a erros pequenos. Intuitivamente, isto parece ser uma propriedade positiva, porém tal característica pode ter um impacto perverso na rede, pois esta vai mudar radicalmente para tentar corrigir este erro e, consequentemente, o alvo também vai mudar radicalmente.

No caso do MAE não há uma divergência na importância entre os erros grandes e pequenos, ou seja, ambos têm igual importância. Assim, MAE não tem a desvantagem de provocar uma mudança radical na rede, porém tem a desvantagem de ignorar a intuição de que erros maiores deveriam ter uma maior importância do que erros pequenos.

A *huber loss* implementa, então, as vantagens de MAE e MSE, criando um híbrido onde MSE é usado para valores baixos e MAE é usado para valores altos, o que permite eliminar as desvantagens referidas acima e usufruir das vantagens.

# 5 Análise de Resultados

Nesta secção, são apresentados os resultados obtidos pelo nosso modelo final. A partir dos mesmos, podemos observar que, apesar de bastantes episódios de treino, o modelo aprende, mas está ainda muito longe de ser perfeito. Após alguma pesquisa, concluímos que as oscilações que vemos nestes gráficos fazem sentido, uma vez que, dada a natureza do *Q-Learning* e a aleatoriedade nos casos de treino, a rede neuronal responsável por calcular os Q-valores para um determinado estado acaba por aprender e desaprender. Ainda assim, com a utilização de certas metodologias, como *experience replay*, conseguimos que o agente aprendesse a jogar razoavelmente bem, ou seja, este tenta fazer com que a raquete vá atrás da bola não só para evitar que a última caia, mas também para que salte e faça pontos, sendo que, com este modelo final, o nosso agente consegue perfazer um total de 52 pontos.

Contudo e olhando principalmente para a Figura 5.1, é importante referir que mais alguns episódios de treino poderiam levar à melhoria do agente, uma vez que vemos uma ligeira subida consecutiva no *score* (ao mesmo tempo que a *loss* está a diminuir cada vez mais). De notar que, no início, os valores são muito baixos, uma vez que o agente ainda se encontra numa fase de observação.

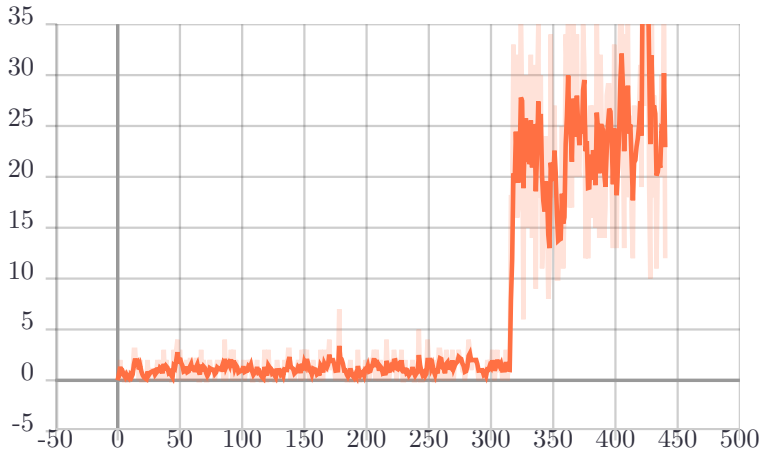



Figura 5.1: *Score*

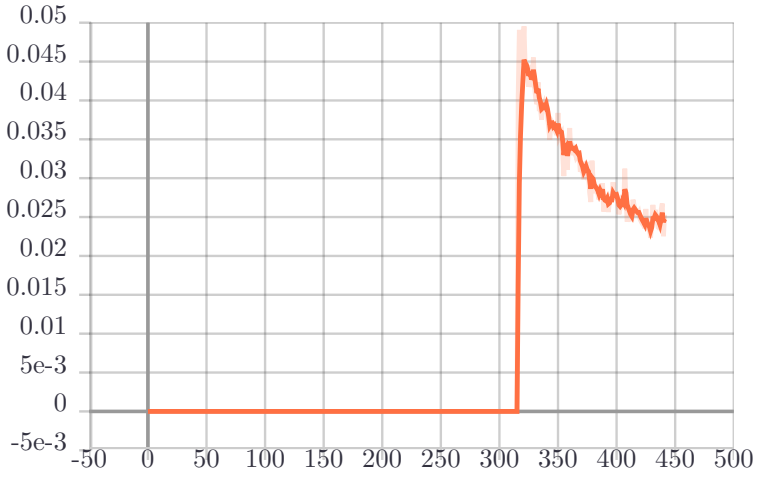



Figura 5.2: *Loss*

# 6 Conclusão

Este trabalho prático consistiu em duas partes distintas: a primeira dedicada ao treino de um modelo, onde o agente treinou uma *Deep Q-Network* e a segunda dedicada ao teste e otimização da rede de *Deep Q-Learning*. Com a realização do mesmo foi, então, possível desenvolver um agente capaz de jogar o famoso jogo Breakout do Atari 2600, obtendo uma pontuação igual a 52.

A principal dificuldade encontrada consistiu nas restrições em termos de memória RAM e GPU. Apesar de terem sido testadas diferentes ferramentas, como o Google Colab, estas apresentaram limites relativamente a estes dois recursos, o que dificultou bastante o desenvolvimento do projeto. Caso isso não se verificasse, eventualmente o desempenho do agente poderia melhorar, porque um maior número de episódios de treino levaria o mesmo a dominar melhor o jogo.

Uma outra dificuldade surgida durante o desenvolvimento do projeto tem que ver com restrições de tempo. Visto que a conceção do algoritmo é um processo iterativo, a fase de treino (que por si só já é um processo demorado) ainda se atrasa mais, não dando tempo suficiente para o modelo treinar um número de episódios razoável.

Embora o nosso agente apresente resultados relativamente satisfatórios, existem alguns aspetos que poderiam ser melhorados, de forma a aperfeiçoar a resolução do trabalho prático proposta neste documento, como a utilização de técnicas baseadas em políticas e valores, por exemplo o algoritmo *Assynchronous Actor-Critic Agent* (A3C), onde o modelo determina a possibilidade de tomar cada ação para cada estado ajustando os mesmos com base num *feedback* devolvido por parte de um segundo modelo que observa a evolução do ambiente. Esta melhoria poderia levar a uma convergência mais rápida, algo que é bastante importante nesta área.

# Bibliografia

[1] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[2] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Playing Atari with Deep Reinforcement Learning (2013).