Deep Reinforcement Learning

André Figueiredo, Luís Ferreira, Pedro Machado, and Rafael Lourenço

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal

Perfil de Sistemas Inteligentes

Computação Natural, Trabalho Prático Nº2, Grupo 5

6 de Junho de 2021

e-mail: {a84807, a86265, a83719, a86266}@alunos.uminho.pt

Resumo Neste relatório é apresentado um algoritmo de *Deep Reinforcement Learning* com o intuito de treinar um agente para jogar o famoso jogo da *Atari 2600*, `Breakout`.
Os algoritmos de *Deep Reinforcement Learning*, tal como o nome indica, são uma combinação de *Reinforcement Learning* (`RL`) com *Deep Learning* (`DL`), ou seja, a parte `RL` consiste na aprendizagem da tomada de decisão de um agente, através de tentativa-erro, e a parte `DL` permite tomar essas decisões, a partir de uma grande quantidade de dados não estruturados, neste caso, *pixels* renderizados do jogo.
Assim sendo, neste relatório será descrito o modelo e a arquitetura da solução desenvolvida, efetuada uma análise dos resultados obtidos e discutidas algumas alterações que poderiam trazer resultados positivos.

Keywords: *Deep Reinforcement Learning* · *Deep Q-Learning* · *Atari* · `Breakout` · *OpenAI Gym* · *Soft Computing* · *Machine Learning*

1 Introdução

O `Breakout` é um jogo *arcade* desenvolvido pela *Atari, Inc.* que foi lançado em 1976 e é baseado no também famoso jogo `Pong`. O objetivo deste é destruir as várias camadas de blocos com uma bola, controlando apenas uma plataforma.

Assim, o trabalho proposto consiste na criação de uma rede neuronal, em conjunto com um algoritmo de *Reinforcement Learning* (no caso de estudo, *Q-Learning*), que deve ser aplicada ao jogo `Breakout` - o agente aprende que tipo de resultado cada ação terá, tentando escolher as ações com a melhor perspetiva de recompensa.

No contexto deste trabalho, recorreu-se à biblioteca *baselines*[1] que implementa um *wrapper*, de forma a integrar mais facilmente o jogo. O objetivo proposto é não só a obtenção de uma pontuação alta, como também a manifestação de comportamentos inteligentes, explorando várias estratégias.

Em suma, este relatório encontra-se divido em três partes - introdução dos conceitos abordados, seguida do modelo, algoritmos desenvolvidos e hiper-pârametros utilizados, terminando com a discussão dos resultados, possíveis melhorias e dificuldades enfrentadas durante o desenvolvimento.

[1] Disponível em: `https://github.com/openai/baselines.git`

1.1 *Reinforcement Learning*

A principal característica do *Reinforcement Learning* (ou aprendizagem por reforço) prende-se com a presença de duas entidades: um agente e o ambiente em que este está colocado.

Deste modo, o objetivo do *Reinforcement Learning* é o treino de um agente que interage com um certo ambiente. Este enfrenta cenários distintos, denominados por *states* ou *observations*, em que cada ação provoca uma *reward* positiva, negativa ou até nula.

Assim, a meta do agente passa por maximizar o total de *rewards* obtidas durante um episódio, sendo que este consiste em tudo o que ocorre entre o primeiro estado e o estado terminal do ambiente. Desta forma, incentiva-se o agente a aprender consoante as suas melhores ações da sua experiência.

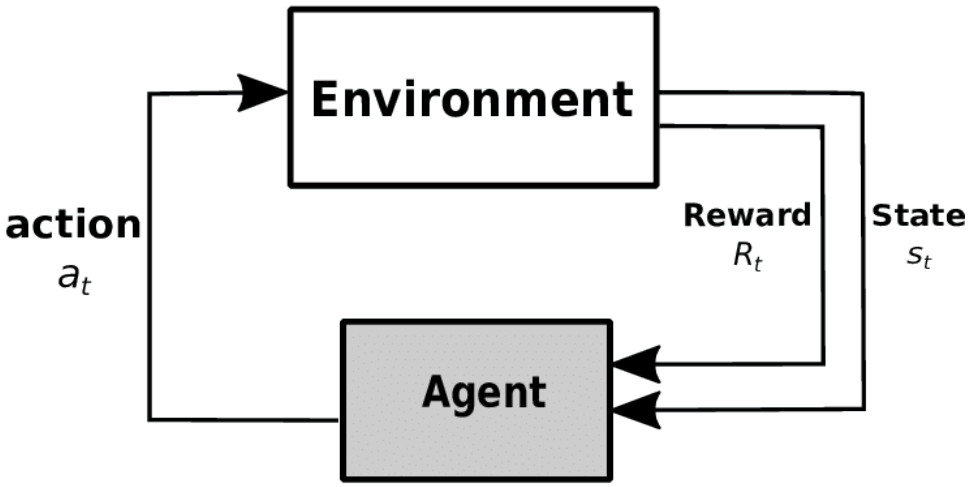


Figura 1: *Reinforcement Learning*

1.2 *Q-Learning*

Nestes contextos, cada estado do ambiente é uma consequência do estado anterior, pelo que se pode assumir que cada estado segue uma propriedade de *Markov*. Esta indica que um estado depende unicamente do seu antecessor, o que faz com que apenas seja necessária informação do penúltimo estado para descobrir a melhor ação a ser tomada.

Desta forma, a *reward* da ação com maior valor denomina-se de `Q-value` e pode ser caracterizada da seguinte forma:

$$Q^*(s,a) = r(s,a) + \gamma \max_a Q(s',a) \tag{1}$$

A equação anterior indica que o `Q-value` no estado s, através da ação a, obtém uma *reward* imediata ($r(s,a)$), para além do maior valor possível do `Q-value` no próximo estado, s'. Assim, o valor de γ poderá diminuir ou aumentar a contribuição dos futuros valores de recompensa, mediante o `Q-value` previsto. Esta equação

é derivada do que foi apresentado por Richard Bellman em [1] e que ficou conhecida, posteriormente, como a *equação de Bellman*. Uma vez que esta função possui recursividade, pode deduzir-se que:

$$Q^*(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2}$$

Nesta equação, o valor de α tanto pode significar o *learning rate* durante o processo de treino, como o número de passos (*step size*). Assim, através desta, é determinado até que ponto a informação mais recente se sobrepõe aos estados antigos.

Uma vez que seria necessário grandes quantidades de memória para explorar todos os estados possíveis no ambiente em questão, começou-se a utilizar algoritmos de *Machine Learning*, mais especificamente *Deep Learning*, para a aproximação dos valores da função 2[2]. Assim, é passado um estado para a rede, sendo retornado todos valores de `Q-value` para cada ação possível.

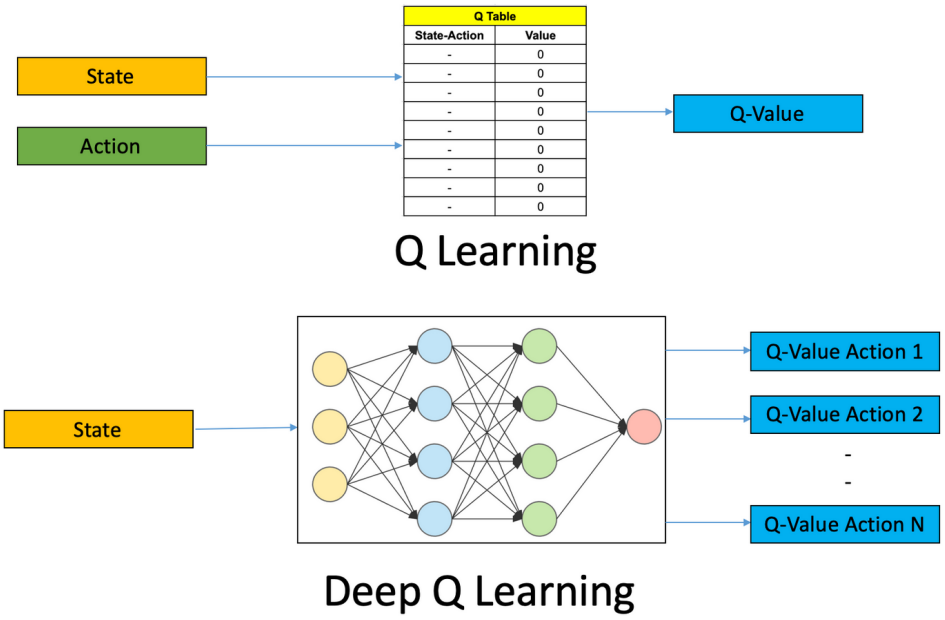


Figura 2: Comparação entre *Q-Learning* e *Deep Q-Learning*

Pode-se, então, definir os passos que um algoritmo *Deep Q-Learning* efetua da seguinte forma:

- Os acontecimentos passados são armazenados pelo agente em memória;
- A próxima ação a ser tomada é calculada através do valor máximo do resultado da rede;
- A *loss function* do modelo é calculada através da comparação entre o valor previsto de `Q-value` e o valor real. Como não se sabe exatamente o valor real (visto tratar-se de um problema de *Reinforcement Learning*), $R_{t+1} + \gamma \max_a Q(S_{t+1})$ é considerado como esse valor real. Embora se esteja a considerar um valor previsto como se tratasse de um real, uma vez que a *reward* é um valor imparcial, acaba por se poder utilizar a *loss* calculada para *backpropagation* dos pesos.

[2]Surgiu, desta forma, o conceito de *Deep Q-Learning*.

1.3 *Double Q-Learning* e outras melhorias

Ainda assim, o algoritmo de *Q-Learning* descrito tem alguns problemas - por exemplo, normalmente, incorre em previsões de `Q-values` altas ao fim de algum tempo de treino, pois as amostras são utilizadas para determinar a ação com a maior esperança de recompensa, assim como os `Q-values` para a mesma ação. Desta forma, é necessário aplicar algumas técnicas para mitigar a superestimação referida, nomeadamente, *Double Q-Learning*. Esta baseia-se na separação da fase de aprendizagem em dois modelos, passando a calcular os `Q-values` para dois estados congruentes e, dada a estocasticidade das amostras, torna-se menos provável que ambas superestimem a mesma ação, diminuindo o *bias* na previsão e, consequentemente, aumentando a qualidade da previsão da recompensa esperada e respetivas ações.

$$Q(s,a) = r(s,a) + \gamma Q'(s', \arg\max_a Q(s',a)) \tag{3}$$

Para além desta, foram ainda aplicadas as técnicas de *Experience Replay Memory* e *Epsilon-Greedy*. A primeira refere-se ao armazenamento de informação de cada *step*, mais especificamente, os estados atual e futuro, a ação tomada e a *reward*, separando assim os processos de aprendizagem do de experiência/exploração. Esta alteração permite ter uma melhor convergência e utilizar, de forma mais eficiente, as experiências passadas, sendo que estas podem ser usadas várias vezes durante o treino, ou seja, não são necessariamente utilizados episódios consecutivos. Caso isto acontecesse, iriam estar a ser utilizadas amostras com uma correlação forte, o que se traduziria numa má aprendizagem, e, portanto, uma seleção aleatória permite eliminar a correlação referida.

No entanto, para que o agente tenha experiências passadas diversificadas, é necessário que tenha um conhecimento maior sobre as consequências das suas ações em variados estados. Para tal, recorre-se à segunda técnica referida, baseada no conceito *Exploration vs. Exploitation*. Para que o agente atinja uma *performance* melhor, necessita de ter conhecimento do ambiente, das ações possíveis e respetivas recompensas. Para tirar o melhor partido da sua experiência, o agente deve explorar situações vantajosas com o intuito de obter recompensas melhores.
Assim sendo, numa fase inicial, o agente deve explorar o ambiente (*Exploration*) tomando várias decisões, apenas com o intuito de o conhecer melhor e, numa fase mais avançada (depois deste já ter treinado e reunido vários estados e ações com as respetivas recompensas), o agente começa a explorar as situações da forma mais vantajosa que "memorizou", tentando aumentar a sua recompensa (*Exploitation*).
Este conceito é muito importante para este tipo de algoritmo, pois aumenta o grau de diversidade dos casos de treino e potencia a descoberta de novas situações favoráveis.

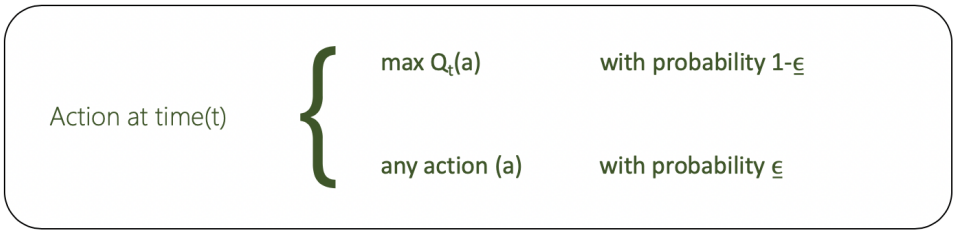
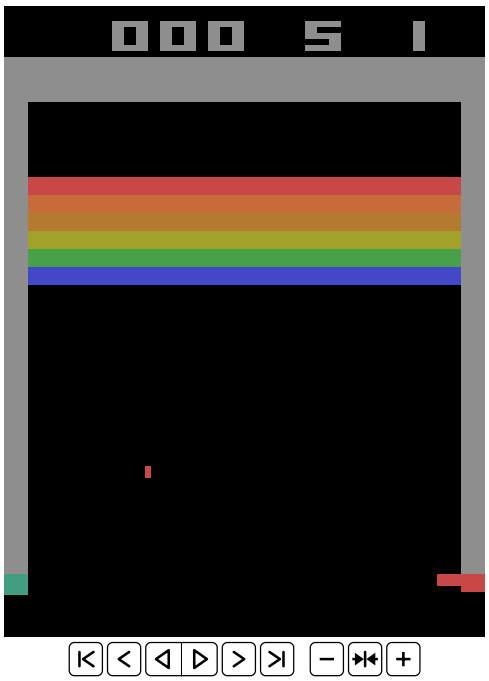


Figura 3: Técnica *Epsilon-Greedy*

1.4 `Breakout`

Como já referido, o ambiente de jogo utilizado para a aprendizagem por reforço, foi o `Breakout`. Este jogo consiste em várias camadas de blocos cujo objetivo é a sua destruição através de uma bola que ressalta numa plataforma (sendo esta o agente).

Exemplo do comportamento do agente[3]



As ações possíveis para este jogo são a deslocação da plataforma/agente para a esquerda (`LEFT`), para a direita (`RIGHT`), não efetuar nenhuma operação, sendo que se mantém na mesma posição (`NOOP`) e, por fim, `FIRE` que serve para colocar a bola em jogo quando se perde uma vida ou o próprio jogo, sendo equivalente a não efetuar nenhuma operação em qualquer outra situação.

A camada de blocos do `Breakout` possui seis cores diferentes que representam *rewards* distintas na sua destruição. Desta forma, a primeira e segunda camadas oferecem uma *reward* de 1 unidade, a terceira e quarta camadas providenciam 4 pontos e, por fim, as últimas duas camadas dão o maior valor de *reward*, 7 unidades.

Outro dado a ter em conta é o crescimento do nível de dificuldade à medida que os blocos superiores são destruídos, resultando em *rewards* maiores. Isto acontece, uma vez que a velocidade da bola aumenta ao bater em blocos de níveis de camadas acima.

Assim, o agente tem como objetivo destruir todos os blocos das camadas, tendo em conta as mudanças de velocidade, *rewards* diferentes e ressaltos nas paredes. Este jogo oferece, assim, diversos estados possíveis que o agente tem de ter em conta para a obtenção de uma boa *performance*.

[3] A figura em causa é, na verdade, um *gif* que pode ser visualizado em anexo no *Adobe Reader*.

2 Solução desenvolvida

2.1 Modelo

O modelo base escolhido para o treino foi a rede definida pelo *paper* da *DeepMind* [2]. Este modelo foi, ainda, uma das sugestões do docente para a execução do treino:

```
inputs = layers.Input(shape=(84, 84, 4,))

layer1 = layers.Conv2D(32, 8, strides=4, activation="relu")(inputs)
layer2 = layers.Conv2D(64, 4, strides=2, activation="relu")(layer1)
layer3 = layers.Conv2D(64, 3, strides=1, activation="relu")(layer2)

layer4 = layers.Flatten()(layer3)

layer5 = layers.Dense(512, activation="relu")(layer4)
action = layers.Dense(num_actions, activation="linear")(layer5)
```

Trata-se de um modelo bastante simples, com três camadas convolucionais para a extração de *features* e duas camadas densas, estando uma delas escondida para melhorar a classificação. A camada de *input* recebe quatro *frames* de tamanho 84×84 e a camada densa de *output* tem quatro neurónios correspondentes ao número de ações possíveis a tomar pelo agente - 0 (`NOOP`), para não ser efetuada nenhuma operação (isto é, parado), 1 (`FIRE`), para voltar a colocar a bola em jogo quando perde uma vida ou perde o jogo, 2 (`RIGHT`), para mover a plataforma para a direita e 3 (`LEFT`), para movimentar a plataforma para a esquerda.

Foram, também, efetuadas algumas modificações à rede, como a mudança do número e tamanho de filtros das camadas convolucionais, várias camadas densas e a alteração do seu número de neurónios. Acrescentou-se, ainda, outras camadas como *Dropout*, *BatchNormalization* e *MaxPooling*, todavia, como estas não geraram resultados mais favoráveis durante a aprendizagem nos primeiros 2000 episódios, decidiu-se manter o modelo base e optar por apenas efetuar alterações no algoritmo/processo de aprendizagem do modelo como, por exemplo, mudanças nos valores de *rewards* e tratamento dos *frames*.

2.2 Algoritmo

Inicialmente, adaptou-se o código fornecido pelos docentes para o treino do jogo `Flappy Bird`, contudo, após várias alterações, tanto a nível de modelo como de algoritmo, o programa não conseguia obter resultados satisfatórios - passadas 72 horas de execução e 2500 episódios, apenas se conseguiu obter uma *reward* máxima de 40 pontos.

Assim, optou-se por procurar por outras soluções, o que levou a uma implementação presente no *website* do `keras` [3].
O algoritmo em questão é baseado na biblioteca *baselines* da `OpenAI`, que serve como *wrapper* ao jogo, alterando alguns parâmetros, como dimensão e cor dos *frames*. No entanto, esta implementação é baseada numa outra variante do `Breakout` de nome "`BreakoutNoFrameskip-v4`", que se comporta de maneira ligeiramente

diferente à pedida neste trabalho ("`BreakoutDeterministic-v4`"). Assim, inicialmente, começou-se por se analisar como este algoritmo se comportava, testando e implementando outros parâmetros. Deste modo, nesta fase, foram já testados vários modelos, vários *epsilons*, *rewards* negativas (quando se perde uma vida), a desnormalização de *reward* e o tratamento do *frame* para remoção do *scoreboard* (estes dois últimos serão explicados de seguida). Com estes testes e modificações conseguiu-se uma *reward* máxima de 394 pontos.

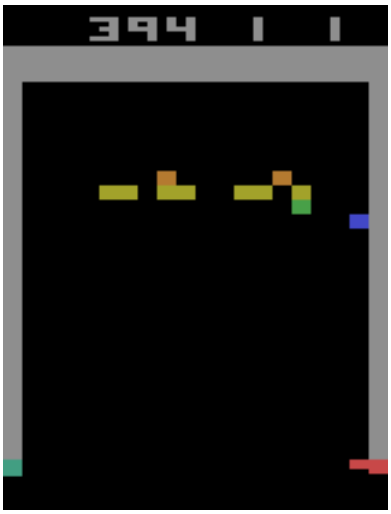


Figura 4: Melhor *reward* no "`BreakoutNoFrameskip-v4`"

Conseguido este valor de *reward*, efetuou-se, então, as alterações necessárias ao código para suportar a variante "`BreakoutDeterministic-v4`", incluindo a alteração no código fonte da biblioteca *baselines* utilizada, de forma a conseguir suportar esta `ROM` que, caso contrário, lança uma exceção.

2.2.1 Algoritmo Base

Nesta implementação, cada episódio é referente a uma vida no jogo, logo cada jogo completo refere-se a cinco episódios.
Um dos parâmetros definidos é o *epsilon* e vai ser este que vai decidir a percentagem de ações aleatórias que o agente vai tomar, tendo um valor máximo, mínimo e, ainda, um decaimento linear a cada *frame* executado.
Outro parâmetro do algoritmo é a *reward* (que foi modificada comparativamente à versão nativa do jogo), sendo que o jogo devolve um valor diferente de *reward* dependendo da camada a que pertence um bloco que foi destruído, como explicado anteriormente, mas, nesta implementação, todo o bloco atingido representa, apenas, 1 unidade de *reward*.
O algoritmo tem, também, um histórico onde armazena a informação relativa a *predicts* passados, sendo este composto por cinco listas:

- `rewards_history` - responsável por guardar as recompensas de cada *step*;
- `action_history` - armazena a informação relativa à ação efetuada num *step*, podendo ser uma das quatro referidas na secção 1.4;
- `state_history` - guarda os valores que dizem respeito aos estados antes do *step*;

- `state_next_history` - é similar ao `state_history`, mas guarda os estados após a execução do *step*;
- `done_history` - guarda o valor *True*, caso tenha perdido uma vida no *step* e *False*, caso contrário.

Por último, temos o processo de treino em si, que é efetuado a cada quatro ações. Utiliza-se um conjunto de várias amostras do histórico com tamanho igual ao hiperparâmetro *batch size*, sendo, assim, usado um algoritmo de *Double Q-Learning* através de um *model* principal e um *target model*. O *target model* vai estar encarregue do cálculo de *rewards* futuras do conjunto escolhido para, posteriormente, serem determinados os valores dos `Q-values` respetivos. Estes valores são, então, passados para o *model*, mais propriamente à função de *loss* `Huber`, de modo a poder ser realizado o treino, que dispõe do `Adam` como otimizador. Apesar do `RMSprop` ser usado por muitos modelos para resolver problemas deste estilo, o `Adam` é considerado uma versão melhorada deste.
É, ainda, de ressalvar que existe um tratamento dos *frames* que é efetuado pelo *wrapper* da biblioteca *baselines*, onde a imagem é passada para a escala de cinzentos, redimensionada e feito o seu *stack* em conjuntos de quatro *frames* para, posteriormente, fornecer ao modelo como *input*.
Os parâmetros desta implementação são os seguintes:

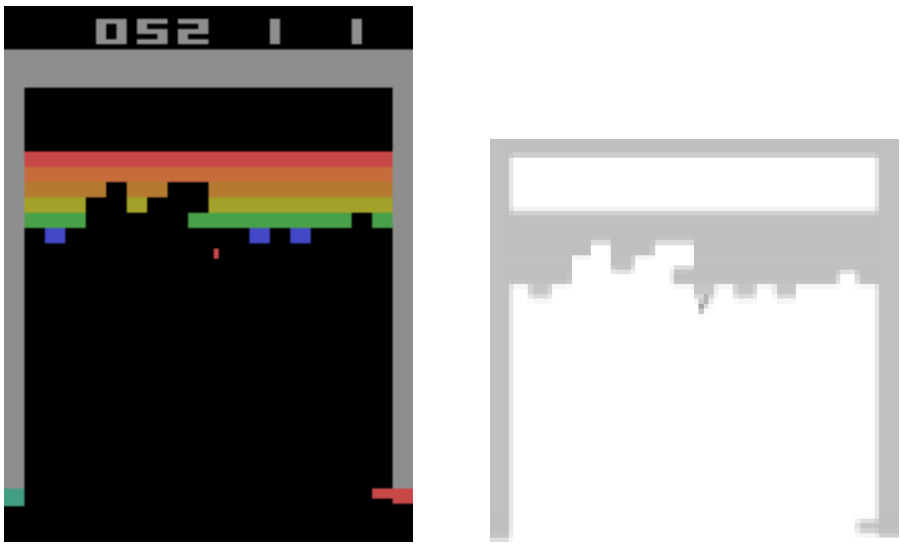
- γ: 0.99
- *Batch Size*: 32
- Tamanho mínimo do *buffer* de experiências: 50000
- Tamanho máximo do *buffer* de experiências: 100000
- Número de *steps* para atualizar o modelo: 4
- Número de *steps* para atualizar o modelo *target*: 10000
- ϵ inicial: 1
- ϵ final: 0.01
- ϵ *decay*: epsilon_max - epsilon_min (Linear)
- *Reward*: Estático, sempre 1
- Penalização por perda de vida: 0

2.2.2 Modificações ao Algoritmo Base

A primeira modificação realizada foi a de o episódio passar a corresponder a um jogo completo. Quando se perde o jogo, o mapa faz *reset* (contudo, perder uma vida não implica que isso aconteça), desta forma, um episódio proporciona uma melhor representação da reinicialização do mapa.
O *epsilon* sofreu uma pequena alteração, passando o seu *decay* a ser realizado após cada episódio ao invés de a cada *frame* processado, recorrendo a uma função exponencial decrescente em vez de linear. Esta mudança torna cada episódio um pouco mais rápido e resulta numa diminuição mais acentuada do *epsilon*, quando este se encontra em valores altos, e numa menos acentuada, quando se encontra em valores mais baixos, o que é preferencial em problemas deste estilo.
Verificou-se que o agente tinha alguma dificuldade a criar túneis nas camadas, que é uma das melhores estratégias para obter pontos, pois a bola consegue penetrar as camadas, destruindo os blocos das camadas mais difíceis, de forma contínua. Durante este processo, a bola desloca-se bastante rápido, mas o agente não precisa de se preocupar em intercetá-la. Então, para ajudar a que esta estratégia ocorra, foi efetuada a desnormalização da *reward*, passando cada destruição de um bloco a devolver a *reward* nativa do jogo.

Ainda sobre a *reward*, verificou-se que o agente, por vezes, se viciava no lado direito do mapa, já que conseguia assegurar pontos "fáceis". Assim sendo, decidiu-se que nos passos em que o agente perdesse uma vida ou o jogo, este iria receber uma *reward* negativa. Desta forma, o agente é obrigado a se movimentar para não estar constantemente a ser penalizado.

Quanto ao tratamento de *frames*, foi decidido que a parte superior deste continha informação prescindível para o treino do agente, visto que se trata do *scoreboard* com a pontuação e as vidas restantes. Assim sendo, optou-se por excluir do treino a parte referida dos *frames*, utilizando apenas o mapa propriamente dito. Como as imagens são passadas de 210×160 para 84×84 pelo *wrapper*, o nosso programa corta-as, passando para 70×84 e, de seguida, faz o redimensionamento novamente para 84×84.



(a) *Frame* original (b) *Frame* após transformações aplicadas

Figura 5: Resultado das transformações efetuadas aos *frames*

2.2.3 Otimizações dos Hiper-Parâmetros

Tal como referido pelo docente, a solução era significativamente influenciada pelos parâmetros escolhidos, sendo que, deste modo, foram realizados vários testes durante vários estágios do treino.

Inicialmente, o treino foi executado utilizando os seguintes parâmetros:

- γ: 0.99
- *Batch Size*: 32
- Tamanho mínimo do *buffer* de experiências: 50000
- Tamanho máximo do *buffer* de experiências: 100000
- Número de *steps* para atualizar o modelo: 4
- Número de *steps* para atualizar o modelo *target*: 10000
- ϵ inicial: 1
- ϵ final: 0.01
- ϵ *decay*: 0.9999 (Exponencial)

- *Reward*: Estático, sempre 1
- Penalização por perda de vida: -2

Todavia, ao fim de algum tempo de treino, constatou-se que a *reward* deveria ser dada consoante a pontuação e que o *epsilon* final era demasiado baixo. Portanto, optou-se por parar e recomeçar o treino, utilizando como ponto de partida o modelo anteriormente treinado.
Neste segundo treino, foram alterados vários parâmetros e, visto que o agente já era capaz de obter resultados razoáveis, optou-se por alterar o ϵ inicial para 0.2 e diminuir o tamanho mínimo do *buffer* de experiências para 10000, pois considerou-se que o treino poderia mudar para a fase de *Exploiting*. Também se aumentou o ϵ final para 0.05, pois, tal como já referido, este tinha um valor demasiado baixo.

Ainda assim, ao fim de algum tempo, concluiu-se que o agente tinha uma penalização demasiado baixa e esta situação representava um *bottleneck* à *performance* do modelo. Assim, fomos novamente obrigados a efetuar uma alteração, aumentando a penalização para -10. Para além desta, optou-se, ainda, por aumentar o *batch size* para 64, de modo a tentar aumentar a *performance* do modelo, o que resultou nos seguintes valores finais de parâmetros no início do treino:

- γ: 0.99
- *Batch Size*: 64
- Tamanho mínimo do *buffer* de experiências: 50000
- Tamanho máximo do *buffer* de experiências: 100000
- Número de *steps* para atualizar o modelo: 4
- Número de *steps* para atualizar o modelo *target*: 10000
- ϵ inicial: 1
- ϵ final: 0.05
- ϵ *decay*: 0.9999 (Exponencial)
- *Reward*: 1, 4 e 7
- Penalização por perda de vida: -10

Enquanto que, para a continuação do treino, foram utilizados os seguintes parâmetros:

- γ: 0.99
- *Batch Size*: 64
- Tamanho mínimo do *buffer* de experiências: 20000
- Tamanho máximo do *buffer* de experiências: 100000
- Número de *steps* para atualizar o modelo: 4
- Número de *steps* para atualizar o modelo *target*: 10000
- ϵ inicial: 0.1
- ϵ final: 0.05
- ϵ *decay*: 0.9999 (Exponencial)
- *Reward*: 1, 4 e 7
- Penalização por perda de vida: -10

Para a execução do modelo, foram utilizados os seguintes parâmetros:

- *Reward*: 1, 4 e 7
- Penalização por perda de vida: -10

Os restantes parâmetros excluídos da lista anterior foram retirados do código, pois o modelo deixou de treinar e passou a tomar todas as decisões durante o jogo.

3 Resultados, Dificuldades e Trabalho futuro

Nesta secção serão discutidos os resultados obtidos durante o desenvolvimento do projeto e será efetuada uma comparação com os resultados esperados. Para além disto, serão também apresentadas várias decisões tomadas (sempre que possível, acompanhadas do raciocínio por detrás delas), as dificuldades sentidas, as melhorias possíveis e o trabalho futuro.

3.1 Análise de Resultados

Relativamente aos resultados obtidos, o algoritmo foi executado em vários cenários, obtendo resultados bastante distintos e, portanto, foi possível chegar a várias conclusões.

Primeiramente, observou-se que o comportamento do agente era significativamente influenciado pelo valor das *rewards* e penalizações. Para valores de *reward* estáticos, este tornava-se conservador e tentava minimizar os riscos de jogo, alvejando, primeiramente, os blocos inferiores (que mantinham a velocidade da bola baixa). Todavia, por dar primazia a estes blocos, quando deparado com uma velocidade maior, rapidamente, o agente perdia o resto das suas vidas.
Por outro lado, para valores de penalização baixos (menores que a *reward* máxima), o agente demonstrava um comportamento impaciente, tentando acertar nos blocos superiores sem a intenção de se manter em jogo. Nestas situações, por vezes, o agente também desenvolvia o comportamento de se encostar a uma das laterais, garantindo pontos automaticamente, mas perdendo o jogo ao fim de quatro/cinco pontos obtidos.
Já para valores de penalização altos, o agente atingia certas situações de jogo em que, devido ao "receio", entrava em ciclos infinitos, tentando apenas sobreviver, sem a intenção de destruir blocos.
Tendo em conta estes três pontos, considerou-se que, para potencializar a *performance* do agente, deviam ser atribuídas *rewards* diferentes consoante a pontuação dada por cada bloco e as penalizações deviam ser ligeiramente superiores à *reward* máxima, por exemplo, 150% a 200% desta.

Tal como já referido, foram ainda testados vários tipos de modelos, diferentes camadas, parâmetros das camadas da rede e um otimizador. No entanto, os primeiros dois pontos enumerados não surtiram melhorias na *performance*, enquanto que os restantes permitiram melhorar a solução. O ponto dos parâmetros (terceiro) foi benéfico numa fase inicial do jogo, ao passo que a utilização de um otimizador (quarto), nomeadamente, `RMSprop`, permitiu que uma solução capaz de atingir 422 pontos passasse a atingir 428.
Para além disto, foram ainda testados diferentes valores de *epsilon* (inicial e final), tamanho do *replay buffer* e *batch size*, sendo que a mudança do *epsilon* foi bastante favorável para a continuação do treino de modelos pré-treinados (espécie de *Transfer Learning*), visto que apenas a partir dos 20% é que o algoritmo começou a atingir pontuações altas com bastante frequência, tal como se pode ver na figura 6. Quanto ao tamanho do *replay buffer* não se observaram grandes diferenças no resultado, possivelmente devido ao número mais baixo testado ser 40000, não muito diferente do *standard*, 100000. Por último, o *batch size* igual a 64 alcançou os melhores resultados, considerando o tempo de treino.

Tentou-se, ainda, alterar o modo de treino, pois constatou-se que, caso o mesmo modelo fizesse um jogo no início e no fim de um episódio, este tomaria decisões ligeiramente diferentes e poderia criar uma flutuação na pontuação obtida. Assim sendo, testou-se efetuar apenas um treino por episódio, utilizando, para isso, um *batch* maior, cerca de 1024. Desta forma, o resultado do modelo durante a execução do jogo seria exatamente a pontuação que este obteria e, assim, podiam ser guardados os seus pesos, sem o risco da sua *performance* diminuir. Este tipo de treino tornou-se bastante útil durante os treinos finais, com *epsilon* igual a 0, pois, caso fosse obtida uma nova pontuação máxima, era possível guardar os pesos da rede e, posteriormente, replicar o mesmo resultado. Para além desta vantagem, o treino tornou-se bastante mais rápido, mesmo com um *batch* muito maior, porém o algoritmo parecia aprender mais devagar e, no geral, tinha um "teto" mais baixo.

No que toca aos diferentes `ROMS` analisados, pudemos constatar que o modelo tinha comportamentos ligeiramente diferentes quando treinado nestes ambientes diferentes. No modo `NoFrameskip`, o agente era mais fluído, reagindo à bola com mais antecedência. Já na versão `Deterministic`, o agente era obrigado a tomar decisões tendo por base um menor número de *frames*, visto que tomava as decisões tardiamente. Ainda assim, estas, quase na totalidade, eram as acertadas.
Esta característica, muito provavelmente, deve-se ao *skip* efetuado pelo *wrapper* utilizado, implicando que cada ação prevista seja executada quatro vezes, problema este agravado devido à natureza do jogo onde já é realizado um *skip* de quatro *frames*, sendo, então, o equivalente a executar um *skip* de dezasseis *frames*.

Por último, um ponto que permitiu atingir uma *performance* melhor, foi a utilização de modelos previamente treinados, por exemplo, quando a *reward* foi modificada de 1 para os valores originais, ao invés de começar o treino do zero, decidiu-se utilizar um modelo treinado até então. Ao fim de algumas horas de treino, o modelo passou a recorrer a novas estratégias, estratégias estas que se tornaram mais benéficas após a modificação referida.
Este permitiu, ainda, treinar o modelo com os pesos das iterações que continham os resultados mais elevados, tentando, assim, tirar o máximo partido do modelo desenvolvido.

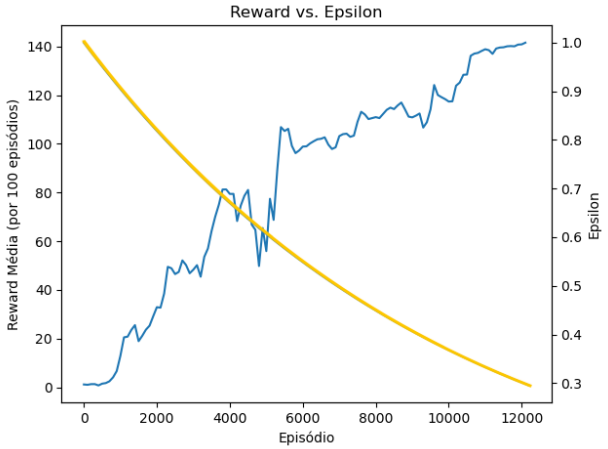


Figura 6: *Epsilon vs. Reward* média[4]

[4]Para esta análise, foram apenas considerados os 20 melhores resultados, a cada 100 episódios.

Relativamente aos resultados esperados, considerámos que o resultado final, 428, ultrapassou qualquer expectativa. Para além do mais, a qualidade de jogo demonstrada pelo agente merece, também, ser alvo de elogios, tendo em conta o curto espaço de tempo de treino do modelo.

3.2 Dificuldades

Durante o desenvolvimento deste projeto, surgiram várias limitações relativas às suas componentes, como, por exemplo, o tempo necessário para o treino e o difícil acompanhamento da evolução do modelo. Assim, nesta secção, serão conjeturadas possíveis soluções para estas barreiras, apresentando as razões pelas quais estas mitigações foram consideradas.

Para tornar a solução mais simples, foi utilizada a biblioteca *baselines*, que permite usar um *wrapper* do *jogo* `Breakout`, fazendo com que seja desnecessário lidar com o *stack* dos *frames* ou, por exemplo, com o recomeço do jogo ao fim de perder uma vida. No entanto, surgiram vários problemas devido ao uso desta biblioteca, nomeadamente, o facto do processamento ser feito automaticamente, todavia o "nosso" processamento da imagem era efetuado sem ter em conta o redimensionamento feito pelo *wrapper*, cortando 30 *pixels* a partir de cima e 8 a partir de baixo, o que resulta num corte demasiado grande, fazendo com que o agente não tivesse acesso à sua posição relativa à bola nem à camada de blocos superior. Para mitigar esta questão, passou-se a retirar 10 e 4 *pixels*, respetivamente.

Para além deste problema, foi também necessário efetuar algumas alterações ao código da biblioteca, pois esta não permitia o uso da versão `Deterministic` e a *reward* tinha como valor máximo 1, independentemente do bloco que era destruído. Tal facto, tornou o agente conservador, tendo como estratégia destruir os blocos mais inferiores, visto que estes são um alvo mais fácil, e, quando atingia um bloco superior, era incapaz de acompanhar a bola devido à sua velocidade. Para mitigar esta questão, foi necessário voltar a utilizar o *reward* "original"/nativo.

Ao fim de algum tempo, constatou-se que, tal como já referido, o agente tinha dificuldades em lidar com uma velocidade alta da bola e, ao fim de alguma discussão, encontrou-se uma razão plausível - a penalização dada quando o agente perdia uma vida era de -2, contudo, nas camadas superiores, cada bloco valia 7, fazendo com que fosse vantajoso arriscar, mesmo sendo quase certo que fosse perder a vida. Assim sendo, foi decidido aumentar esta penalização para -10.

Por último, um entrave que pode ter sido responsável por abrandar o treino, foi o facto do *wrapper* utilizado efetuar um *skip* de quatro *frames*, ou seja, o modelo estava a prever apenas uma ação (sendo que esta é efetuada quatro vezes seguidas) de quatro em quatro *frames*. Assim, caso este problema tivesse sido descoberto numa fase mais inicial, poderia ter-se realizado as alterações necessárias para que o agente treinasse com a nova taxa de *refresh*, dando, assim, maior controlo na tomada de decisão, visto permitir uma diversidade maior nas decisões (sem que a mesma ação fosse tomada quatro vezes seguidas).

3.3 Trabalho Futuro

Tal como já evidenciado, durante o desenvolvimento deste projeto, surgiram várias limitações/problemas que abrandaram o seu desenvolvimento e consequente obten-

ção da solução final.
Assim, nesta secção serão apresentadas algumas melhorias e funcionalidades que poderiam permitir obter resultados ainda melhores e mais breves.

De modo a mitigar as consequências dos problemas e desafios descritos neste relatório, surgiram várias ideias de modo a potenciar a *performance* do agente durante o treino.

Foi considerado penalizar o agente por se viciar num dos lados e tentar forçar este movimento, isto é, o agente já está encostado à parede e continua a tentar deslocar-se para esse mesmo lado. Desta forma, este comportamento poderia ser mitigado até certo ponto, fazendo com que o agente não tomasse decisões inúteis, já que este tipo de ações não alteram o estado do jogo. Assim, o agente poderia explorar outro estilo de estratégias para potenciar a sua *performance*. No entanto, uma vez que pode ser tomada a decisão de não efetuar nenhuma operação e que nos treinos seguintes este problema não se verificou, decidiu-se descartar esta solução.

Foi, ainda, considerado penalizar o agente consoante a distância a que ficou da bola quando perde uma vida, isto é, caso o agente estivesse perto da bola no momento que esta passa por ele, ele teria uma penalização menor do que quando perdesse e a bola passasse mais longe. Deste modo, o agente era incentivado a tentar aproximar-se da bola e, por consequência, aumentar a probabilidade de acertar. No entanto, esta ideia foi abandonada, pois considerou-se que o processamento necessário para fazer esta análise seria muito elevado, tendo em conta o tempo necessário para o treino do agente. Ainda assim, era algo que poderia ser testado numa fase de desenvolvimento posterior, caso os resultados não fossem excelentes, numa tentativa de obter uma solução mais otimizada.

À imagem da possível melhoria anterior, foi discutido recompensar o agente com uma *reward* por cada vez que acertasse na bola, todavia, tal como referido na melhoria anterior, o processamento necessário para analisar estes eventos era demasiado elevado. A razão por detrás desta ideia, deve-se ao facto de a recompensa ser dada aquando da destruição dos blocos, contudo este evento apenas acontece devido à plataforma acertar na bola. Assim, o agente seria recompensado por se manter em jogo, no entanto esta implementação teve de ser adiada para um possível desenvolvimento futuro.

Para além das melhorias anteriores, foi avaliada a possibilidade de penalizar o agente caso este ficasse muito tempo sem obter uma *reward* positiva proveniente do *environment* (excluindo, assim, as nossas *rewards* e penalizações), permitindo mitigar possíveis ciclos infinitos e contribuir para uma exploração maior nestas situações. Inicialmente, esta ideia surgiu para prevenir que o agente ficasse muitos *steps* sem executar a ação `FIRE`, mantendo o jogo parado, evitando penalizações, e, mais tarde, que alguns modelos começassem a entrar em ciclos infinitos, apresentando comportamentos de jogo para sobrevivência, sem tentar destruir blocos. Considerou-se que este comportamento era algo perigoso para a evolução do modelo, todavia, visto que esta estratégia apenas apareceu nos treinos finais em que os modelos obtinham uma *performance* superior, concluiu-se que não era uma tarefa prioritária.

Por último e com vista a tentar alcançar resultados ainda melhores, teria sido útil a utilização de um outro algoritmo, *Actor Critic*. A parcela *Critic* estima os valores de `Q-value` enquanto que o *Actor* atualiza a política de distribuição do gradiente indicado pelo *Critic*. Este algoritmo permitiria, na teoria, alcançar *performances* superiores para problemas com grandes quantidades de ações possíveis e possibilitaria uma melhor exploração do ambiente.

4 Conclusão

A realização deste projeto permitiu aplicar os conhecimentos apreendidos nas aulas teóricas e práticas da unidade curricular de Computação Natural, bem como incentivar a exploração das técnicas de *Deep Reinforcement Learning* e o estado de arte desta área de *Machine Learning*.

Os principais desafios durante o desenvolvimento deste projeto prenderam-se com as dificuldades que os ambientes de treino utilizados criaram, assim como a dificuldade em acompanhar a *performance* do agente durante o seu treino. Outro desafio encontrado foi o facto da biblioteca *baselines* não permitir uma configuração fácil dos parâmetros de treino e, deste modo, vários parâmetros estavam definidos com valores menos bons para o ambiente em causa, resultando num treino mais moroso e com resultados, possivelmente, piores.

Em relação ao comportamento do agente, pudemos observar vários comportamentos inteligentes nos mais variados estágios de treino.
No início do treino, o agente demonstrou uma tendência em se encostar a ambos os lados do mapa, pois garantia que alguns blocos eram obrigatoriamente eliminados. A partir deste ponto, o agente pareceu ficar "preso" a esse comportamento inicial, obrigando a alterar o treino e recomeçar do zero. Neste segundo treino, à imagem do que aconteceu anteriormente, o comportamento de se encostar apareceu novamente, no entanto, ao fim de várias horas de treino, começou a demonstrar um comportamento mais inteligente, tentando executar outras ações para além dessa.
A partir deste momento, o agente começou a obter resultados mais aceitáveis e aprendeu a destruir primeiro os blocos inferiores, tentando eliminar camada a camada, contudo, para as camadas mais avançadas, já começava a demonstrar dificuldade em acompanhar a velocidade de jogo.
Depois de se valorizar mais os blocos superiores, o agente aprendeu o comportamento de tentar criar um túnel no lado esquerdo, podendo assim eliminar as camadas superiores com maior facilidade e velocidade, resultando num risco menor. Ainda assim, o agente tinha dificuldade em acompanhar a velocidade da bola quando esta terminava o túnel, perdendo rapidamente as suas vidas após este ponto.
Por fim, o agente começou a desenvolver o comportamento de explorar a estratégia do túnel, todavia, ao contrário da tática anterior, tenta fazê-lo pelo centro do mapa, demorando um pouco mais, mas eliminando mais blocos das camadas inferiores para tal. No entanto, o agente agora é capaz de acompanhar o ritmo de forma mais eficiente, não perdendo as vidas logo após o fim do túnel, resultando numa pontuação mais alta e num comportamento mais inteligente e dinâmico. Neste ponto, considera-se que foram alcançados resultados excelentes, visto que se atingiu a pontuação de 428 (apenas a 4 pontos da pontuação máxima possível), eliminado praticamente todos os blocos (restando quatro), recorrendo a algumas estratégias identificadas no estado de arte.

Em suma, concluí-se que o trabalho desenvolvido se encontra bastante favorável e cumpre todos requisitos pretendidos pela equipa docente no contexto da problemática em questão e que foi desenvolvido, com sucesso, um agente capaz de jogar e obter um *score* alto. Outro ponto favorável foi a utilização de um conceito introduzido no primeiro trabalho prático desta cadeira, *Transfer Learning*, na medida em que foram utilizados modelos treinados anteriormente, para continuar o treino de novos modelos.

Referências

1. BELLMAN, R. (1954). **The Theory of Dynamic Programming**. *RAND Corporation*. Disponível em: `https://www.rand.org/pubs/papers/P550.html`

2. MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., & RIEDMILLER, M. (2013). **Playing Atari with Deep Reinforcement Learning**. Disponível em: `https://arxiv.org/pdf/1312.5602v1.pdf`

3. CHAPMAN, J., & LECHNER, M. (2020, May 23). **Deep Q-Learning for Atari Breakout**. *Keras*. Disponível em: `https://keras.io/examples/rl/deep_q_network_breakout/`