Estudo da aplicação de Deep Q Learning no jogo *Snake*

João Lourenço e **Miguel Faria**
Departamento de Engenharia Informática, Universidade de Coimbra
uc2019217948@student.uc.pt, miguelfaria@student.dei.uc.pt

Abstract

A inteligência artificial nos jogos tem vindo a evoluir, contribuindo para uma melhor experiência e maior realismo, ao simular comportamento humano. Nesse sentido, este documento pretende retratar a aplicação, experimentação e análise de um algoritmo de Reinforcement Learning, Deep Q Learning, a que um agente no jogo *Snake* foi sujeito. No final, espera-se que este consiga atingir uma pontuação elevada no jogo de forma consistente e eficaz, no menor número de iterações de aprendizagem possível e que a sua aprendizagem tenha um progresso crescente. Com este objetivo, foram realizadas diversas experiências: as iniciais direcionadas a otimização dos parâmetros do algoritmo; posteriormente com o intuito de testar mecanismos que visam evitar o choque do agente e a otimização do tempo que demora a capturar os objetivos. Os resultados obtidos foram positivos, tendo conseguido aumentar a pontuação obtida pelo agente consideravelmente, no entanto, não foi possível alcançar o objetivo de manter um sentido crescente da evolução do seu desempenho.

1 Introdução

A aprendizagem usando Reinforcement Learning é um dos ramos de Inteligência Artificial com mais relevo. Baseado num método de treino relativo a Machine Learning, consiste em gerir a forma como as ações tomadas por um agente inteligente impactuam a sua aprendizagem, atribuindo *rewards* de acordo com o resultado obtido e o objetivo final. Existem diversas implementações que se enquadram neste tipo de aprendizagem, sendo que neste trabalho vai ser estudada a utilização do algoritmo Deep Q Learning num agente inteligente no jogo *Snake*.

O jogo *Snake* consiste num ambiente no qual estão inseridos uma linha, que simboliza a cobra, e diversos objetivos, que simbolizam os frutos. O agente tem como finalidade capturar o maior número de objetivos sem que embata em si próprio ou em qualquer um dos limites da área de jogo (bordas da aplicação), perdendo caso isso aconteça. Após apanhar um dos objetivos, o tamanho da cobra aumenta uma unidade e um novo objetivo é definido numa posição aleatória da área de jogo que não se encontre ocupada.

Com a utilização do algoritmo referido, iremos variar os diferentes parâmetros que nele intervêm, nomeadamente: *reward*, *learning rate*, *discount rate* e *randomness* (*tradeoff exploration/exploitation*). Após obter os resultados das experiências, irá ser analisada a maneira como esses parâmetros influenciam não só a pontuação máxima que o agente atinge num determinado número de tentativas ou tempo, bem como a rapidez com que ele atinge uma detereminada pontuação. Para além disso, também vai ser observada a evolução das *rewards*, para melhor perceber que parâmetros poderão ser mais benéficos. No final, pretendemos observar se o uso de Reinforcement Learning, mais precisamente do algoritmo Deep Q Learning, permite obter resultados satisfatórios para as diferentes finalidades pretendidas, bem como algumas das suas limitações. Assim, iremos analisar se a sua aplicação em jogos deste gênero poderá ser viável e adequada caso seja necessário um agente inteligente.

2 Metodologia

O Reinforcement Learning inclui diversos algoritmos, usados comummente em processos de controlo, como robótica e condução autónoma, e de tomada de decisão. Isto vai exatamente de encontro ao objetivo do jogo que escolhemos para implementar este método: a cobra tem sempre de decidir o seu próximo passo com o objetivo de apanhar a fruta, tendo sempre em conta o que já "aprendeu". Assim, optámos por usar o algoritmo Deep Q Learning pertencente a esse ramo de aprendizagem, uma vez que nos irá permitir variar diversos parâmetros de forma simples e de modo a tentar obter os melhores resultados possíveis. Por consequência, teremos então dois componentes principais: o ambiente, que será o jogo, e o agente, que será a cobra ou mais corretamente a Deep Neural Network que irá determinar as suas ações.

Com recurso a este algoritmo, foram efetuadas diversas experiências, tendo sempre em consideração a validade e adequação dos resultados.

2.1 Deep Q Learning

O Deep Q Learning é um tipo de algoritmo onde são combinados conceitos de Reinforcement Learning com uma rede neural profunda (Deep Neural Network), de maneira a criar uma função ação-valor ideal, que irá ser usada na determinação da melhor ação para um certo estado. O objetivo do algoritmo é aprender uma política que maximize a soma das recompensas obtidas em cada jogo (*reward* cumulativa, mencionada posteriormente.

Como o próprio nome indica, este método funciona de modo semelhante a Q Learning, diferenciando-se apenas no uso de uma rede neuronal ao invés de uma tabela (*Q-Table*) para o cálculo dos valores Q (*Q-values*) para cada ação num determinado estado. Isto deve-se ao facto do uso uma tabela não ser apropriado para casos com bastantes estados (*game states*) e ações (apesar de no nosso caso o número dessas ações ser relativamente reduzido).

O algoritmo é iterativo, onde em cada iteração, é recolhido o conjunto de estados e ações e determinada a ação ideal usando a função ação-valor. Em seguida, é realizada a ação e observada a recompensa resultante, que ira ser usada para atualizar os pesos da rede neural a fim de maximizar a recompensa esperada. Este processo é repetido até que a recompensa esperada convirja e a política seja aprendida.

Para obter os valores Q, é tida em conta a equação de Bellman, enunciada na *Figura 1*:

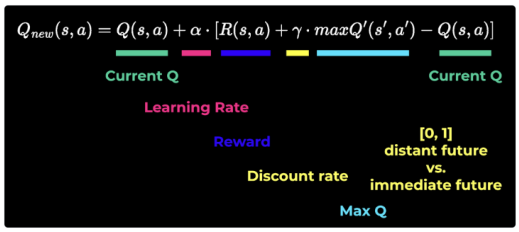


Figura 1 - Equação de Bellman

A manipulação de algumas das variáveis desta equação é o que nos poderá permitir otimizar o agente, obtendo melhores resultados:

- *Learning rate* (LR) – taxa na qual o agente ajusta os seus valores para cada ação que toma; determina o quanto da *reward* o agente atribui à ação e o quanto é usado na atualização dos valores do agente para cada ação; quanto maior a *learning rate*, mais rápida será a aprendizagem e quanto menor, mais lenta será.
- *Discount rate* (*gamma*) – taxa usada para determinar o valor e importância de *rewards* futuras; determina o quanto da *reward* o agente atribui a *rewards* que receberá no futuro; quanto maior a *discount rate*, maior o valor atribuído a *rewards* futuras e quanto menor, menor será o valor atribuído.
- *Randomness (tradeoff exploration/exploitation)* (*epsilon*) – fator que irá permitir ao agente alternar entre *exploration*, onde ele toma novas decisões e procura novas rewards, e *exploitation*, onde ele toma as decisões tendo em conta o que aprendeu e maximiza as *rewards*; deve existir um bom balanceamento entre estas duas abordagens, visto que caso o agente apenas realize *exploration*, irá tomar várias vezes ações desfavoráveis, mas caso apenas realize *exploitation*, irá ficar limitar a sua evolução pois apenas toma decisões que maximizem a *reward* segundo as experiências passadas.
- *Reward* – valor usado para medir o sucesso ou falhanço de uma ação tomada pelo agente; funcionam como um *feedback* de maneira a moldar o comportamento do agente.

2.2 Jogo Snake

No caso do nosso jogo, como visível na *Figura 2*, a cobra tem cor azul, começando sempre com tamanho de 3 unidades. Relativamente aos objetivos, estes têm cor vermelha e encontram-se aleatoriamente na área de jogo representada a preto. Os movimentos possíveis para a cobra são cima, baixo, esquerda e direita, controláveis usando as setas do teclado, caso seja testado o jogo manualmente. No canto superior esquerdo é possível observar a pontuação atual.

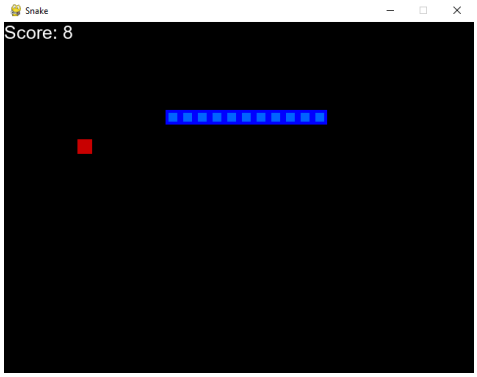


Figura 2 - Exemplo do jogo Snake usado

2.3 Processo de testagem

Ao longo de todas as experiências, procuramos manter os testes coerentes e consistentes, de forma a conseguir mais facilmente analisar os resultados e realizar comparações mais corretas. Deste modo, foram efetuadas, para cada tipo de experiência, pelo menos 3 execuções, permitindo que os resultados de eventuais comportamentos anómalos do agente afetem de forma negativa as conclusões. Assim, para realizar as análises, será usada a média dessas execuções, no caso da pontuação e da *reward*. Para os respetivos valores médios, estes serão calculados com os valores obtidos.

2.4 Avaliação do desempenho

Com o objetivo de avaliar o desempenho do agente para as diversas experiências realizadas, iremos ter em conta diversos fatores:

- Pontuação máxima e pontuação média;
- Evolução da soma de *rewards* de cada jogo;
- Tempo demorado em cada jogo.

Através da análise destes parâmetros poderemos verificar não só se o agente consegue atingir pontuações consideradas boas, mas também se está a ter uma evolução crescente, quer da pontuação quer das *rewards*.

3 Experimentação e Análise de Resultados

O processo de experimentação começou com o teste do algoritmo recorrendo a valores *default* nas diferentes variáveis, que serviu como ponto de partida.

É importante salientar que, dado que não pretendemos que o nosso agente seja influenciado por falta de memória e que não é possível realizar experiências com número de iterações com grande ordem de grandeza, estabelecemos que os valores de memória máxima (MAX_MEMORY) e tamanho do batch (BATCH_SIZE) iriam permanecer inalteráveis, sendo 100_000 e 1000, respetivamente. Para além disso, os testes irão ser realizados para 500 iterações.

3.1 Experiência 1 – Teste Inicial

Para a experiência inicial, foram utilizados os seguintes parâmetros:

- *Learning rate*: 0.005
- *Discount rate*: 0.9
- *Randomness* (*tradeoff exploration/exploitation*): 80 – (número de jogos atual)
- *Reward* por apanhar fruto (CATCH_BONUS): +10
- *Reward* por bater (COLLISION_PENALTY): -10

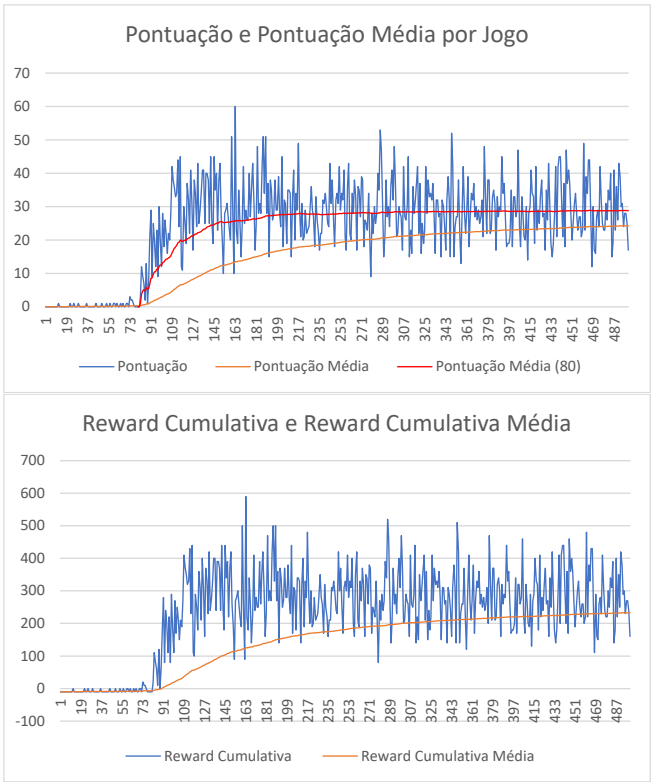


Figura 3 - Resultados obtidos relativos à experiência 1

De acordo com a Figura 3, o agente tem uma tendência positiva de aprendizagem, obtendo uma pontuação máxima perto de 60. Sensivelmente a partir dos 80 jogos, o agente começa a atingir pontuações mais elevadas; no entanto, considerando também a linha de pontuação média, acaba por estabilizar rapidamente, obtendo pontuações pouco variantes entre si a partir das 150 iterações. Estas ocorrências são também comprovadas pelos valores das *rewards* cumulativas de cada jogo, podendo estas ser definidas como a soma das *rewards* que são atribuídas ao agente após cada ação que ele toma.

Esse comportamento de subida repentina da pontuação deve-se à forma como o *tradeoff exploration/exploitation* está implementado nesta primeira experiência. Para permitir que exista uma variação entre o tipo de tomada de decisão que o agente utiliza, é usado um número aleatório até 200; caso esse número seja inferior ao valor de *randomness* definido, o agente toma uma ação aleatória; caso contrário, utiliza o que aprendeu até aí. Deste modo, já que a partir dos 80 jogos é impossível que o número aleatório seja inferior à *randomness*, o agente passa a ter apenas em conta o que já aprendeu nas iterações anteriores, conseguindo então melhores resultados. Apesar disto, esta implementação pode também ser uma das causas pelo que existe uma estabilização dos valores de pontuação e *reward*, uma vez que a partir desse momento o agente nunca toma nenhuma ação nova que poderia permitir a sua evolução.

Como se pode perceber, o gráfico das rewards espelha com diferente escala o gráfico das pontuações, pelo que vai ser usado apenas o da pontuação nas experiências para facilitar a análise.

3.2 Experiência 2 – Funções de *Randomness* com Probabilidade de ocorrer *Exploration*

Com intuito de verificar se essa estabilização se deve ao *tradeoff exploration/explotation* usado, foram realizados 3 testes de maneira que, ao longo das iterações, existisse sempre uma probabilidade de ocorrer *exploration*, fazendo com que o agente tomasse uma ação nova que poderia levar a uma melhor *reward*.

Para isso, usou-se então os valores de 10%, 5 % e 1% de probabilidade desse acontecimento, implementado uma lógica semelhante à inicial, no entanto deixando uma probabilidade de a ação escolhida ser a exploração mesmo quando o número de iterações é alto:

```
if self.n_games <= 100-(200*(probability/100)):
    self.epsilon = 100 - self.n_games # random
choosing between exploration and exploitation
else:
    self.epsilon = 200*(probability/100) #
probability% of choosing exploration
```

Segundo a Figura 4, ao contrário do esperado, o agente, para além de continuar com pontuações estacionárias, sem

ocorrência de evolução, também atingiu pontuações máximas e médias menores. Constata-se também que quanto maior foi a probabilidade testada, pior foram os resultados obtidos, razão pela qual o teste para 10% não se encontra sequer representado. Tendo em conta o sucedido, descartámos o uso deste método.

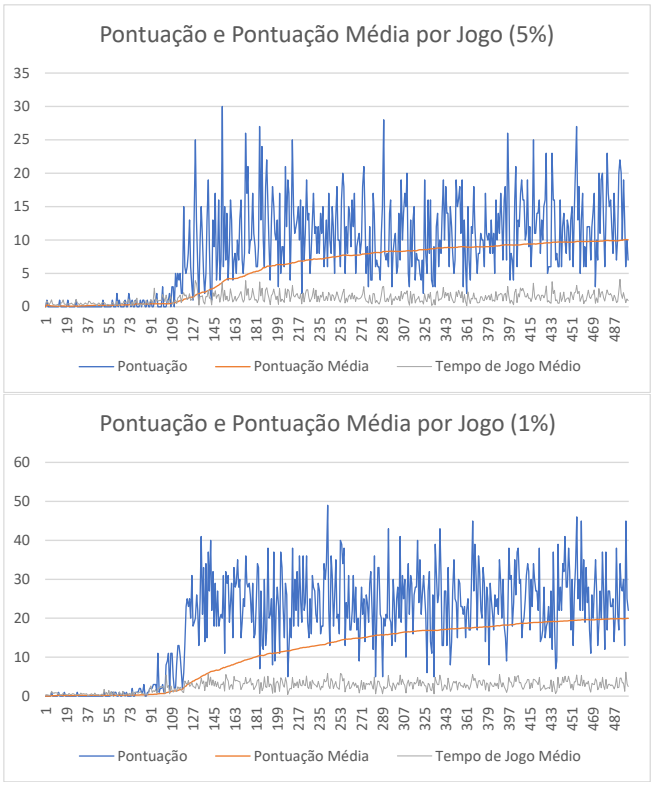


Figura 4 - Resultados obtidos relativos à experiência 2: probabilidade de ocorrer Exploration 5% (cima) e 1% (baixo)

3.3 Experiência 3 – Aumento do Número de Iterações onde pode ocorrer *Exploration*

Visto que os valores da experiência anterior não corresponderam ao comportamento previsto, foi adotada uma outra estratégia para verificar se o agente conseguia atingir pontuações mais elevadas.

Foi usada a transição da *randomness* (*tradeoff*) inicial, mas aumentado o número de iterações onde pode ocorrer *exploration*, para que, apesar de a partir de uma certa altura o agente apenas realizar *exploitation*, até esse instante consegue realizar mais ações de *exploration*, possibilitando que ele tenha em conta mais movimentos aleatórios que poderão ajudá-lo. Deste modo, a lógica aplicada foi a seguinte:

```
self.epsilon = numIterations - self.n_games # from iteration numIterations, only chooses exploitation
```

O número de iterações testados foram 200, 150 e 100, como é visível na Figura 5. Para a realização desta análise importa ter em conta o valor da pontuação média a partir do momento em que o agente apenas opta por *exploitation*, pois os valores anteriores afetam significativamente a visualização do efeito deste método. Esses valores encontram-se representados a vermelho nos gráficos.

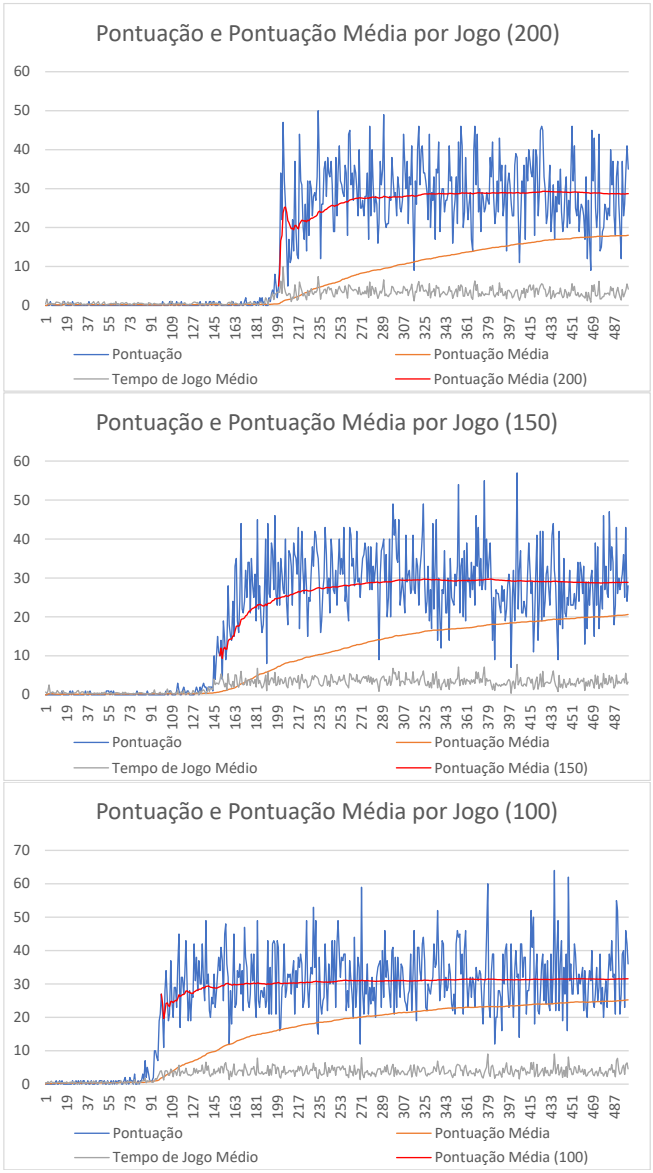


Figura 5 - Resultados obtidos relativos à experiência 3: Possibilidade de ocorrência de Exploration nas 200 (cima), 150 (meio) ou 100 (baixo) primeiras iterações

Observa-se que, nos dois primeiros casos, os valores da pontuação permanecem numa gama semelhante aos do caso inicial, embora salientando que no caso de 200 o valor de pontuação média aparenta estabilizar num menor número de iterações, por volta dos 28 pontos, podendo significar que contribui para uma aprendizagem mais rápida na fase de *exploitation*.

No entanto, para o caso das 100 iterações onde pode ocorrer *exploration*, verifica-se que tanto atinge pontuações máximas mais elevadas comparativamente a todos os outros casos, como também possui uma pontuação média superior, por volta dos 32 pontos. Assim, consideramos que este parâmetro para o *tradeoff* é benéfico no desempenho do agente.

3.4 Experiência 4 – Diferentes valores para *Learning Rate* e *Discount Rate*

Para além de verificar qual a melhor função para o parâmetro *randomness*, é também importante aperfeiçoar os valores de *learning* e *discount rate*. Foram então efetuadas diversas tentativas, a partir da configuração inicial, variando cada um destes dois parâmetros independentemente.

Relativamente à *learning rate*, obteve-se os seguintes resultados:

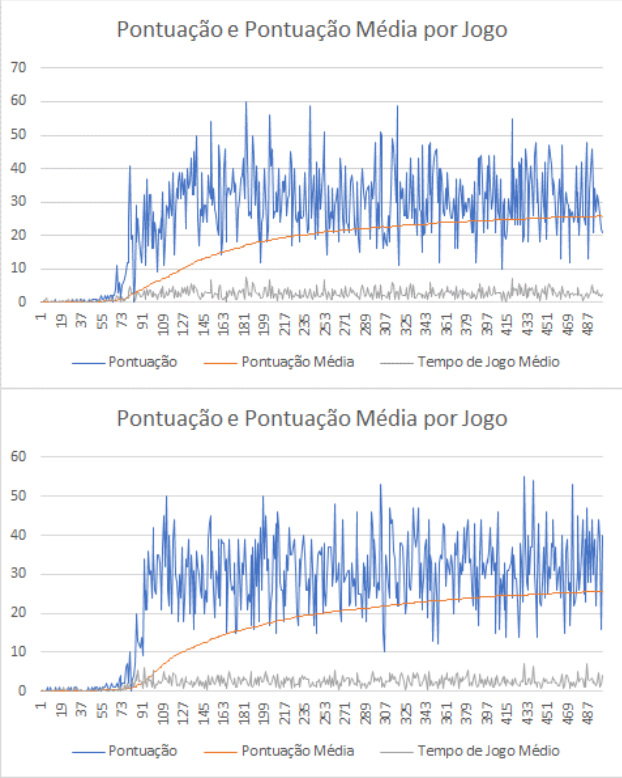


Figura 6 - Resultados obtidos relativos à experiência 4: valor de learning rate 0.003 (cima), 0.02 (baixo)

Os testes foram realizados entre valores 0.001 e 0.1, apresentando os mais significativos na Figura 6. Pode-se concluir que valores de *learning rate* entre 0.001 e 0.005 obtêm resultados bastante semelhantes, no entanto, consegue-se destacar as pontuações obtidas com o valor 0.002, com pontução média a rondar os 27 pontos, que é ligeiramente superior às restantes e aparenta ter as varições a ocorrer num intervalo menor.

Testando agora valores de *discount rate*, mantendo também os valores iniciais e agora o valor de *learning rate* 0.002, obteve-se os resultados da *Figura 7*.

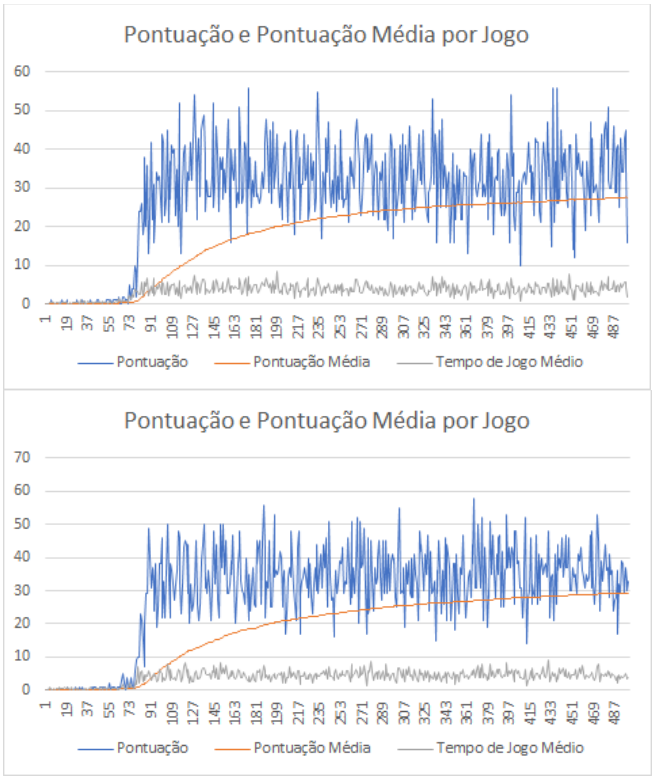


Figura 7 - Resultados obtidos relativosà experiência 4: valor de discount rate 0.8 (cima) e 0.7 (baixo)

Foram utilizados nestes testes valores entre 0.5 e 0.9 encontrando-se representados acima os testes mais relevantes. Tendo em conta o obtido, conclui-se que o valor de 0.7 obtém uma média de pontuação superior ao fim do mesmo número de iterações. Podemos também observar que o gráfico de baixo apresenta uma amplitude de valores menor comparativamente ao gráfico de cima, o que implica uma maior consistência por parte do agente.

3.5 Experiência 5 – Penalização quando o agente se dirige em direção a si próprio

Ao observar a cobra a realizar os movimentos, constatou-se que muita das vezes em que perdia, derivava de se deslocar em direção a uma parte do seu corpo. De maneira a contrariar este acontecimento, foi arranjada uma estratégia que dá uma penalização cada vez que a cobra se dirige em direção a si, sendo ela cada vez maior à medida que a distância entre essas duas partes do seu corpo diminui.

Para aplicar a penalização, foi utilizada a lógica seguinte:

```
if self.direction == Direction.UP and pt.x ==
self.head.x and pt.y < self.head.y:
```

```
    if (self.head.y - pt.y) <= (HITS_ITSELF_GAP
* self.h):
        if (self.head.y - pt.y) < auxH:
            auxH = self.head.y - pt.y
            reward = -HITS_ITSELF_PENALTY *
(HITS_ITSELF_GAP * self.h - (self.head.y -
pt.y)) # reward = -HITS_ITSELF_PENALTY * (per-
centage_of_res * (height or width) - differ-
ence_between_body_parts)
```

Sendo assim, utilizaram-se os melhores parâmetros encontrados até agora, e acrescentaram-se novos que permitem realizar este procedimento:

- *Learning rate*: 0.002
- *Discount rate*: 0.7
- *Randomness* (*tradeoff exploration/exploitation*): 100 – (número de jogos atual)
- *Reward* por apanhar fruto (CATCH_BONUS): +10
- *Reward* por bater (COLLISION_PENALTY): -10
- Penalidade dada quando a cobra se dirige em direção a si própria (HITS_ITSELF_PENALTY): 0.01
- Percentagem da resolução que fornece o espaço mínimo entre as partes da cobra onde começa a penalização (HITS_ITSELF_GAP): 0.4

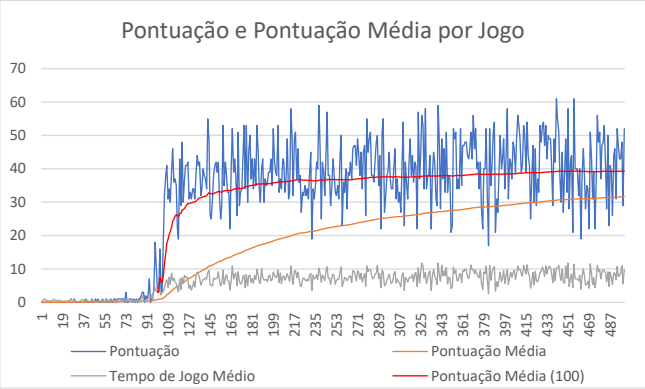


Figura 8 - Resultados relativos relativamente à experiência 5: inclusão de penalização quando o agente se dirige em direção a si próprio

Embora a pontuação média tenha apenas sofrido um pequeno aumento (~33), observa-se que a pontuação média a partir da fase de *explotation* obteve uma subida bastante considerável (~40), de cerca de 10 pontos. As pontuações máximas também aumentaram, encontrando-se por volta dos 60 pontos. No entanto, o agente aparenta continuar a estagnar.

3.7 Experiência 6 – Penalização pelo tempo demorado entre captura de frutos/embates

Com o intuito de reduzir o tempo dos jogos, foi implementada uma função de *reward* que prejudica as colisões ou capturas de frutos mais demoradas, usando-se o método seguinte:

```
reward -= (self.frame_iteration -
self.last_frame) * TIME_PENALTY # give penalty
for time between collisions/catches
```

Para além dos valores usados na Experiência 5 que se revelaram benéficos, foi acrescentado o seguinte valor:

- Penalização por tempo (TIME_PENALTY): 0.01

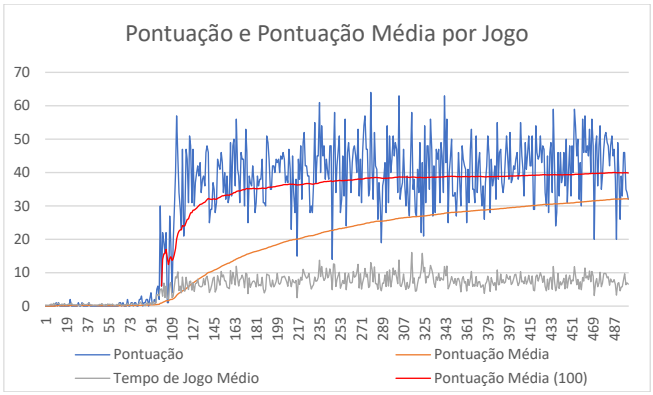


Figura 9 - Resultados relativos à experiência 6: inclusão de penalização pelo tempo demorado entre captura de frutos/embates

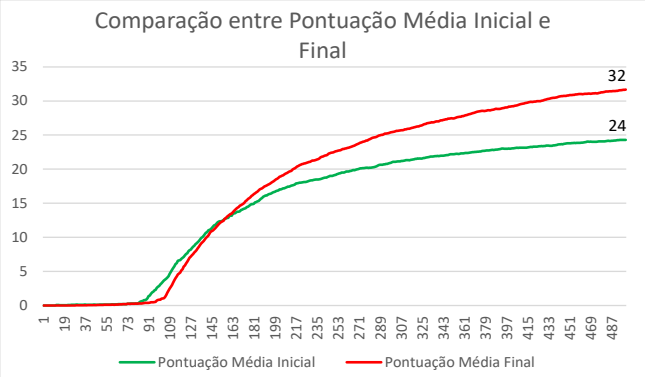
Comparando a *Figura 9* com o resto das experiências desenvolvidas, não é possível inferir se esta implementação afeta positiva ou negativamente o desempenho do agente. Nas 500 épocas realizadas pelo agente não foi possível detetar nenhuma mudança significativa de comportamento que pudesse indicar impacto por parte do algoritmo implementado. Talvez em agentes com maior número de épocas de treino fosse possível detetar alguma mudança e alcançar resultados favoráveis.

4 Conclusões

Tendo em conta os procedimentos realizados, o agente final possui as características mencionadas de seguida:

- *Learning rate*: 0.002
- *Discount rate*: 0.7
- *Randomness* (*tradeoff exploration/exploitation*): 100 – (número de jogos atual)
- *Reward* por apanhar fruto (CATCH_BONUS): +10
- *Reward* por bater (COLLISION_PENALTY): -10
- Penalidade dada quando a cobra se dirige em direção a si própria (HITS_ITSELF_PENALTY): 0.01
- Percentagem da resolução que fornece o espaço mínimo entre as partes da cobra onde começa a penalização (HITS_ITSELF_GAP): 0.4

Como se pode constatar, ao longo de todas as experiências, não foi possível fazer com que o agente conseguisse manter uma tendência crescente ao longo de todas as iterações de jogo, verificando-se que ele acaba quase por estagnar. Apesar disto, conseguiu-se fazer com que o ele obtivesse algumas melhorias, pelo que se conclui que é benéfico o uso deste algoritmo de aprendizagem.



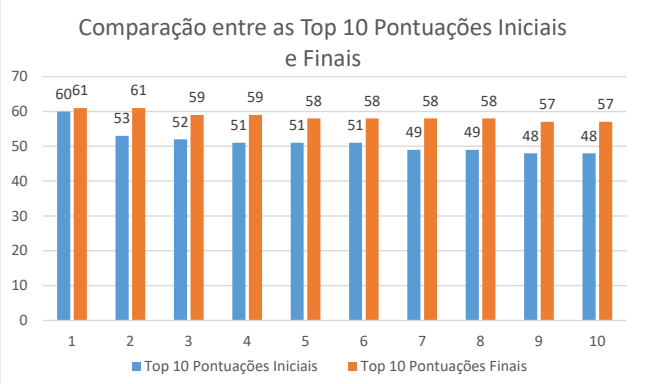


Figura 10 - Evolução do desempenho do agente

Ao observar a *Figura 10*, verifica-se qua as pontuações médias não melhoraram tanto como gostaríamos, apresentando um aumento de apenas 8 pontos. No entanto, relativamente às 10 pontuações mais elevadas, pode-se observar que, apesar da mais elevada ser semelhante entre os dois momentos inicial e final (60 e 61, respetivamente), o agente final consegue manter valores muito mais estáveis, com uma variação de apenas 4 pontos, ao invés do agente inicial, que possui uma variação de 7 pontos entre a primeira e segunda melhor pontuação e de 12 entre os 10 melhores. Com isto podemos concluir que o agente evoluiu positivamente, confirmando que o uso de Deep Q Learning permite que um agente do jogo *Snake* alcance resultados benéficos.

5 Trabalho Futuro

De modo a continuar a evolução do agente, seria interessante testar ainda num número de iterações maior, que poderia levar a melhores resultados em algumas das experiências, bem como realizar mais iterações por experiência, que não foi possível devido às limitações de tempo. Também poderia ser útil aprimorar não só o algoritmo de penalização por tempo demorado, mas principalmente o algoritmo de penalização da cobra por se movimentar num sentido em que irá embater em si própria, que permite ainda modificação de alguns valores.

Nota

Os ficheiros relativos a este projeto, nomeadamente documentação e código, encontram-se no repositório de GitHub "Snake_AI_RL" partilhado com o professor.

Referências

LOEBER, Patrick; SEGUIN, Michael. Snake AI PyTorch: Teach AI To Play Snake! Reinforcement Learning With PyTorch and Pygame. [*S. l.*], 3 dez. 2021. Disponível em: https://github.com/patrickloeber/snake-ai-pytorch. Acesso em: 2 dez. 2022.

LOEBER, Patrick. Reinforcement Learning With (Deep) Q-Learning Explained. [S. l.], 2 fev. 2022. Disponível em: https://www.assemblyai.com/blog/reinforcement-learning-with-deep-q-learning-explained/. Acesso em: 2 dez. 2022.

COMI, Mauro. How to teach AI to play Games: Deep Reinforcement Learning. [*S. l.*], 15 nov. 2018. Disponível em: https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a. Acesso em: 2 dez. 2022.