Deep Reinforcement Learning
Exploration in Atari Game Pitfall

Grupo 4 - Aprendizagem Profunda, Sistemas Inteligentes
Alexandre Soares[pg50166], Gonçalo Braz[a93178], and Rúben Santos[pg50733]

Universidade do Minho, Braga, Portugal

1 Introdução

Para este projeto prático o grupo escolheu a alternativa A do enunciado e o tema que escolhemos abordar é "Deep Reinforcement Learning - Jogos" mais concretamente a resolução de um problema de exploração difícil. O tema que nos tinha sido atribuído inicialmente era "Deep Reinforcement Learning Robotics". O grupo decidiu alterar o tema, porque após alguma deliberação entre os membros do grupo e diálogo com a equipa docente percebemos que seria mais interessante tentarmos resolver um problema de exploração num jogo, que é o ambiente final onde o nosso modelo seria utilizado. No caso da robótica tal não seria possível, porque o grupo não possui nenhum robô, e por isso, o modelo seria treinado e testado em ambiente simulado sem termos a hipótese de testá-lo num ambiente real.

Dito isto, o jogo que foi escolhido para o grupo tentar resolver o problema de exploração foi o jogo Pitfall! feito para a consola Atari 2600. Escolhemos o jogo Pitfall!, porque é considerado um dos jogos de exploração mais difícil da consola Atari, esta classificação devesse ao facto das recompensas do jogo serem muito escassas e difíceis de atingir. Dito isto, este é um problema interessante de abordar, uma vez que, os problemas mais difíceis do mundo real também são problemas de exploração.

2 Metodologia

O métodos escolhido pelo grupo é composto por três fases principais. Estas fases são: estudo do estado da arte, mais especificamente, o estudo dos algoritmos que foram utilizados para resolver este tipo de problemas, escolha dos algoritmos que vamos utilizar para resolver o problema e o treino/teste destes algoritmos.

Na primeira fase, fomos verificar as classificações obtidas a jogar o jogo Pitfall! no site Papers With Code[5]. Ao analisar esta tabela reparámos que havia poucos algoritmos a conseguir obter uma classificação positiva no jogo. Existem apenas três entradas na tabela de algoritmos que foram capazes de o fazer. Estas entradas são do algoritmo Go-Explore(que tem duas versões diferentes submetidas)[6,7] e do algoritmo Agent57[9].

O sucesso do algoritmo Go-Explore[6,7] deve-se ao facto deste algoritmo ser capaz de revisitar estados promissores do jogo e a versão do algoritmo [6] que

teve mais sucesso tinha acesso a conhecimento do domínio(posição do agente e número do quarto atual). No caso do algoritmo Agent57 o seu sucesso deve-se ao desenvolvimento de melhorias para aplicar ao agente NGU(Never Give Up). Este agente junta duas ideias, a exploração motivada por curiosidade(curiosity-driven exploration) e a utilização de agentes distribuídos de DRL, nomeadamente o R2D2.

Para além disso, constatamos que os algoritmos que estavam na tabela de classificação que obtinham pontuações nulas ou negativas faziam parte de estudos mais generalizados que avaliavam a performance dos algoritmos o jogar jogos Atari sem ter como principal foco a resolução de problemas de exploração difícil.

Na segunda fase, escolhemos os algoritmos que íamos usar para jogar o jogo Pitall!. Para tal escolhemos os algoritmos PPO[10] e DQN[11] que são dois algoritmos que já tinham sido estudados e analisados pelo grupo no trabalho de investigação desta unidade curricular. Para além disso, constatamos que existiam estudos onde estes algoritmos[10,12] foram postos a jogar o jogo Pitfall! que não estavam presentes na tabela classificativa referida anteriormente. A escolha destes algoritmos foi apoiada pelo facto de acreditarmos que pela adição de conhecimento do domínio(domain knowledge) à função de recompensa do jogo Pitfall! seriámos capazes de obter pelo menos um dos tesouros disponíveis no mesmo.

Na terceira fase, tivemos de escolher onde iríamos realizar o treino e o teste dos algoritmos. O local escolhido para realizar o treino dos modelos foi a plataforma Kaggle que permite a utilização de uma placa gráfica Tesla P100 que acelerou bastante o processo de treino dos modelos. Dito isto, era necessário escolher o número de timesteps(cada timestep equivale a 4 frames) utilizados para treinar os nossos modelos. Uma iteração de treino de um modelo com 1M(milhão) de timesteps demora cerca de 6 horas a treinar. Após está constatação decidimos treinar todos os nossos modelos com pelo menos 1M de timesteps, porque embora o número de timesteps recomendado seja na ordem 100M ou até mais este número não ia ser alcançável devido ao limite de utilização semanal da placa gráfica P100 e pelos cerca de 30 dias que tínhamos para realizarmos o trabalho. O grupo chegou à conclusão que seria melhor testar vários modelos com diferentes funções de recompensa do que treinar apenas um modelo com o máximo de timesteps possível.

3 Sofware Desenvolvido

Os resultados do trabalho encontram-se no repositório [1]. Neste nós incluímos dois playbooks principais: *pitfall_PPO* e *pitfall_DQN* que serviram para desenvolver os modelos mencionados na secção 5.6. Estes modelos, 3 no total, encontram-se na pasta models cada 1 com versões que geraram resultados mais interessantes do conjunto e com os logs respetivos dos seus treinos.

Além disso, na pasta gif incluímos gifs dos resultados de cada um dos modelos a jogar, mas é algo que pode ser reproduzido a partir dos próprios notebooks.

De modo aos notebooks funcionarem, temos a pasta UndergroundCNN que incluí o modelo CNN para previsão da presença do jogador debaixo de terra que será descrito nas secções seguintes.

Por último, a pasta extra contém outro trabalho realizado no processo do desenvolvimento, como um playbook com experiências sobre a RAM do ambiente do gym, assim como o playbook de treino da undergroundCNN e um teste dos algoritmos de DRL num jogo diferente do Pitfall! para comparação de resultados base.

4 Características do Ambiente

O ambiente utilizado para a abordagem do problema foi um ambiente atari construído utilizando a biblioteca *gymnasium* da OpenAi e disponível no repositório 'ALE' [3]. Deste ambiente utilizamos a versão mais recente, i.e. V5.

Ambientes do *gymnasium* são relativamente semelhantes entre eles, não sendo o Pitfall! exceção. Na sua essência, através deste ambiente somos capazes de realizar uma ação no jogo, e esta chamada devolve os resultados da ação, i.e. o novo estado, a reward obtida pela ação e informações sobre o jogo: se já terminou, o número de frames passadas, a quantidade de vidas restantes.

O estado, ou observação, do pode ser devolvido em um de dois formatos, uma matriz representante da frame do jogo, ou o estado da sua memória em formato de Array.

No decorrer do trabalho nós experimentámos a utilização dos dois métodos de observação, tendo começado por notar os resultados da utilização mais simples da frame do jogo. Mais tarde, com o uso do método *render* do ambiente, fomos capazes de utilizar as observações em formato 'ram' sem perder a informação das frames, que são o input usual para as redes de deep reinforcement learning.

Especifiquemos agora melhor este ambiente.

Em termos de ações possíveis, este contém um espaço composto por 18 ações possíveis na Atari que podem ser vistas na tabela Espaço de ações Pitfall. do anexo.

Deste conjunto de ações o grupo identificou 8 ações que são redundantes, ou seja, o conjunto da ações resultante pode ser vista na tabela Espaço de ações não redundantes Pitfall. do anexo.

Em termos de observação, no caso das frames, estas são imagens RGB de 210x160. No caso da RAM esta é um array de bytes com 128 elementos. Para a exploração da RAM, seguimos o trabalho realizado no artigo 'How Pitfall Builds its World' [4].

Por último, as recompensas que o ambiente devolve são completamente proporcionais às alterações do score no jogo, i.e., quando o score no jogo aumenta a recompensa é positiva e equivalente e vice versa para o caso negativo. Isto significa, que o ambiente base não tem qualquer incentivo inerente para o agente à exceção do que o jogo propõe: recompensa positiva na coleta de tesouros e recompensa negativa quando se cai em obstáculos específicos (troncos, cair no subterrâneo sem escada).

5 Desenvolvimento

Nesta secção procuramos descrever de forma simples o processo de desenvolvimento. Fazem parte do notebook principal os seguintes componentes: Modelo base, Dependências do Projeto, Modelo para previsão do agente em posição subterrânea, *Wrapper* do Ambiente de Jogo, Modelo DRL e Treino do Modelo.

Iremos nas próximas subsecções iterar por cada um destes pontos, realçando a sua importância e providenciando pormenores da sua implementação. Os resultados dos testes ao modelo são discutidos na secção Resultados.

5.1 Modelo base

De forma a termos um ponto de partida para percebermos os pontos fracos do modelo, assim como as melhorias passíveis de serem realizadas, começamos por testar uma implementação de uma DRL base, utilizando o algoritmo PPO e sem modificações no ambiente.

Este modelo foi treinado com um learning rate de $10^{-}6$ por um total de cerca de 1 milhão de timesteps, com atualizações no algoritmo a cada 5000 steps.

O resultado deste modelo foi de acordo com as nossas expectativas, isto é, o modelo não conseguiu score positivo no final de todo o tempo de treino tendo, no entanto, aprendido a não obter score negativo, para isto ele simplesmente ficava num loop infinito em que não explorava de modo a não encontrar obstáculos.

Isto deve-se ao facto de obter recompensa positiva neste jogo ser algo bastante tardio, sendo necessário percorrer várias salas para isso acontecer.

Além disso, algo que também notámos, foi que caindo na área subterrânea, o agente tendia a ficar num loop infinito em que era incapaz de voltar à superfície.

O mesmo teste foi aplicado no algoritmo DQN, com resultados semelhantes.

5.2 Dependências do Projeto

Este projeto têm um conjunto de dependências que necessitam de ser instaladas para o seu funcionamento. Destas fazem parte os módulos *gymnasium* e o respetivo ambiente atari dado por *gymnasium[atari]*. Para que haja compatibilidade entre o *gymnasium* e o jogo Pitfall é também necessário instalar uma API conhecida como *shimmy*.

Os módulos *torch* e *torchvision* são necessários para definir as redes neuronais convolucionais do projeto, nomeadamente a rede que prevê que o jogador está debaixo de terra e o modelo que dá suporte ao PPO.

O módulo *stable-baselines3* é importante pois define os algoritmos de DRL (PPO, DQN, etc) e fornece ainda uma API para definir uma CNN personalizada que sirva de suporte ao PPO.

O *matplotlib* é utilizado para gráficos e exibição dos resultados do modelo. O *tensorboard* oferece uma interface web para visualização dos resultados do treino em tempo real. Por fim, o *scikit-image* é usado para transformar imagens.

5.3 Modelo para previsão do agente em posição subterrânea

Começando por tentar resolver o problema que observamos do loop em locais subterrâneos, decidimos optar por criar um rede neuronal separada de modo a tentar detetar este atributo (agente debaixo de terra) à partida, não dependendo do modelo DRL, tentando evitar um excesso de tempo de treino para tentar ultrapassar este problema naturalmente.

Para isto, optámos por usar redes convolucionais que recebem como input a frame atual do ambiente, e tentam detetar se o agente está debaixo de terra. Esta rede foi treinada utilizando frames do próprio ambiente com dimensões de (130x130) e com uma distribuição entre labels a rondas os 50%. Em termos de resultados estes rondaram por volta dos 98% de acerto no teste, sendo importante notar que pode haver discrepância com salas mais avançadas, embora o cenário seja sempre similar.

5.4 *Wrapper* do Ambiente de Jogo

A principal variável e foco do trabalho consistiu na modificação do ambiente do jogo, isto de modo a que o ambiente, sem ser demasiado influenciador, proporciona-se mais incentivos ao agente. Ao contrário de outros problemas de redes neuronais, o fine tunning deu-se principalmente no ambiente wrapper criado.

Um ambiente wrapper é uma classe do *gymnasium* que permite dar overwrite num ambiente de forma simples e mantendo as compatibilidades que o original possuía. É neste onde, por exemplo, é aplicada a previsão da CNN de posição subterrânea explicada anteriormente.

O seguinte conjunto representa as diferentes modificações que foram testadas ao longo do desenvolvimento da solução:

- **Limitação do número de vidas e recompensa negativa por perda de vida**:
  Motivação : acoplado com o incentivo de explorar, esperávamos que limitar o número de vidas incentiva-se a aprender a ultrapassar obstáculos.
  Método : primeiramente, utilizávamos a informação das vidas fornecido no resultado do método 'step' do ambiente porém, este valor, devido a animações do próprio jogo, demorava muitos steps entre o momento da morte e a atualização do valor, tornando difícil para o agente compreender que teve score negativo devido à morte. Por este motivo, fazendo comparação entre as últimas frames por diferenças, verificamos que ocorre morte se 5 frames seguidas forem iguais.
- **Repameamento das ações**:
  Motivação : como existem ações redundantes ou sem efeito no jogo, a redução do espaço destas leva o agente a ter um conjunto de escolhas melhor, teoricamente facilitando a aprendizagem.
  Método : simples repameamento de ações utilizando um dicionário.

- **Incentivo de exploração de novas salas**:
  Motivação : para descobrir tesouros, o agente terá de percorrer entre as salas, por isso nós decidimos introduzir uma recompensa por entrar em salas nunca antes vistas.
  Método : através da informação da RAM, conseguimos identificar a posição que indica a sala atualmente renderizada. Guardamos assim as salas visitadas numa lista e, caso a sala atual não esteja na lista, uma recompensa positiva é dada ao agente. De forma a que o agente seja mais incentivado a não retroceder no caminho (seguir sempre para a esquerda ou direita), testámos ainda dar um reward negativo quando entra numa sala já visitada.
  Além disso, de modo a não ficar permanentemente numa sala para não ter reward negativo dos obstáculos, testámos a introdução de um reward negativo se ele passar demasiados 'steps' na mesma sala.
- **Normalização das recompensas**:
  Motivação : de modo a diminuir o impacto de rewards de valores muito dispersos dos introduzidos por nós, normalizamos de modo a estarem sempre dentro de um intervalo limitado.
  Método : normalização no intervalo [-1,1].
- **Diminuir tempo de baixo de terra**:
  Motivação : andar debaixo de terra, embora permita ao jogador percorrer o mapa mais depressa, não tem qualquer tesouro a aparecer lá, e tem paredes que impedem o agente de passar entre salas, obrigando a que ele aprende a voltar a subir para a superfície, algo que dificultaria o treino.
  Método : redimensionamento das frames para 130x130, seguido pela passagem destas frames para a CNN de previsão do subterrâneo. Caso seja previsto que o agente está debaixo de terra, é dado reward negativo.
- **Terminação do jogo de acordo com a exploração realizada**:
  Motivação : ao contrário da grande maioria de outros jogos de arcade, o PitFall! apresenta um mapa bastante grande e que não te impede de ficar numa mesma sala durante 20 minutos, sendo que isso poderá ser melhor do que explorar outras salas, acabando com score menor do que o inicial. Deste modo, decidimos, inspirando-nos de certa forma no *Go-Explore*[6], segmentar o jogo de modo a em cada iniciação do ambiente. Assim, esperávamos que o seu treino fosse mais progressivo, assim como a evolução do seu reward (auxiliado com reward por exploração). Quanto melhor o agente se tornar, maior será o objetivo que ele terá disponível (número de salas a explorar) e maior será o rewards máximo possível de obter.
  Método : utilizando uma variável que guarda o número de salas visitadas até ao momento, esta não é redefinida no método *reset*, sendo utilizada para comparar na próxima iteração do ambiente se o número de salas exploradas é maior do que o explorado na iteração anterior, se for, o jogo termina e o número de salas exploradas é atualizado. Caso o jogo termine antes de o jogador alcançar este objetivo, então o objetivo é reduzido.

Nos modelos finais, 1 utilizando o algoritmo PPO e outro utilizando DQN, incluímos as modificações: repameamento das ações, incentivo de exploração de

novas salas, normalização de recompensas e diminuição de tempo de baixo de terra. A limitação do número de vidas, embora lógico à partida, tendeu a impedir o agente de explorar, fazendo com que ele permanece-se nas primeiras salas.

5.5 Modelo DRL

Neste projeto o modelo de Deep Reinforcement Learning (DRL) utiliza uma arquitetura CNN personalizada. A CNN é responsável por extrair características relevantes das observações do jogo, que são usadas pela rede de políticas (*policy*) para tomar decisões.

A CNN consiste em três camadas convolucionais seguidas por funções de ativação ReLU e camadas *max pooling*. A utilização de camadas convolucionais permite capturar informações espaciais nos *frames* do jogo, enquanto que o *max pooling* reduz as dimensões espaciais e ajuda a extrair recursos importantes. As funções de ativação ReLU introduzem não linearidade ao modelo, permitindo que ele aprenda padrões complexos nos dados. Finalmente, uma operação de *Flatten* é realizada para transformar a saída das camadas convolucionais num formato vetorial. A camada linear consiste numa camada totalmente conectada com ativação ReLU. Essa camada processa ainda mais os recursos extraídos pela CNN e reduz a dimensionalidade para a dimensão de recursos desejada, que é definida como 128 neste caso. Em anexo, na tabela Arquitetura Personalizada da CNN utilizada no modelo de DRL., podemos consultar a arquitetura da rede em mais detalhe.

Esta CNN é então integrada em dois algoritmos de aprendizagem por reforço: Proximal Policy Optimization (PPO) e Deep Q-Network (DQN).

O PPO é conhecido pela sua eficiência na otimização das políticas, o que o levou a ser o mais utilizado ao longo do processo de desenvolvimento. O PPO atualiza a sua política de forma equilibrada, tendo em conta um equilíbrio entre *exploration-exploitation*. Com este algoritmo podemos esperar uma política constante que não sofre alterações excessivamente divergentes da política anterior.

O segundo algoritmo utilizado foi o DQN, que utiliza redes neuronais profundas para aproximar a função de valor da ação. Esta rede do DQN não foi manipulada explicitamente. No jogo Pitfall o planeamento e exploração a longo prazo são cruciais para o sucesso. O DQN implementa um buffer de experiências que permite ao agente armazenar e experimentar experiências passadas. Isto leva a uma melhor exploração e tomada de decisão.

Em suma, ao combinar estes dois algoritmos o modelo demonstrou melhorias significativas face a outras políticas de aprendizagem. A CNN personalizada, face à *standard* oferecida pelo módulo do PPO foi capaz de extrair algumas *features* importantes do ambiente que resultaram, por exemplo, na aprendizagem de evitar obstáculos móveis.

5.6 Treino do Modelo

Nesta secção vamos referir quais foram as funções de recompensa utilizadas para produzir os melhores modelos PPO e o melhor modelo DQN, uma vez que a

configuração da rede nunca foi alterada entre modelos. Vamos utilizar os pontos da secção 5.4 para referirmos os aspetos que foram incluídos, mudados e ou alterados entre os modelos.

Em primeiro lugar vamos referir os componente que foram utilizados para calcular a função de recompensa usada para treinar um dos algoritmos PPO. Esta função tinha os seus valores de recompensa normalizados entre -1 e 1, não limitava o número de vidas, mas atribuía uma recompensa negativa pela morte do agente, o espaço da ação foi reduzido das 18 ações iniciais para 10 ações, o agente era incentivado a explorar novas salas e recebia recompensas negativas sempre que estivesse na zona subterrânea do ecrã. O aspeto de explorar duas salas pode ser subdividido em dois tipos de recompensa: a recompensa positiva recebida por encontrar um novo ecrã e a recompensa negativa dada ao agente quando ficava demasiado tempo no mesmo quarto. Este modelo não termina o jogo com base na exploração realizada. Os seus gráficos de loss e reward vão ser colocados em anexo.

Quanto à função de recompensa do algoritmo de DQN podemos referir que é quase idêntica à função de recompensa do modelo PPO a única diferença está na maneira de como a recompensa de explorar novas salas é calculada. A sua componente positiva de exploração manteve-se inalterada. No modelo PPO a componente negativa da recompensa começava a 0.15 e era atribuída em intervalos regulares de 15 timesteps pelo agente passar muito tempo no mesmo quarto. Para o modelo DQN a recompensa negativa era atribuída em intervalos de 50 timesteps com o valor de 0.10 pelo agente passar muito tempo num dos quartos já visitados. Para este algoritmo apenas conseguimos colecionar os seus valores de reward. O gráfico que representa estes valores também vai ser colocado em anexo.

O último modelo relevante de menção foi uma última versão do nosso ambiente que, embora estivesse a mostrar resultados promissores, ainda precisava de mais treino. Este modelo foi também treinado com o algoritmo PPO de igual modo aos primeiros. No caso deste modelo no entanto, foi onde introduzimos o método de limitação do jogo de acordo com a quantidade de salas visitadas. Além disso, o ambiente estava ainda limitado a apenas 2 vidas, de forma a ter a possibilidade de ignorar pelo menos uma sala mais difícil; foi retirada a normalização das recompensas, de modo que a recompensa negativa dos obstáculos naturais do jogo é muito mais relevante para o agente, incentivando a que ele não ignore o seu impacto. As restantes características do ambiente são semelhantes ao primeiro destes três modelos. Este modelo foi treinado um total de 1.5 milhões de timesteps.

Em termos de hiperparâmetros, estes foram mantidos constantes entre os modelos, de acordo com os seus valores base dos algoritmos, visto que verificamos ao longo de todas as nossas iterações do desenvolvimento que, embora relevantes, as maiores modificações na aprendizagem vinham da capacidade do ambiente de incentivar o agente a melhorar, tendo sido portanto esse o nosso maior foco.

6 Resultados

Em termos de resultados, estes não são tão palpáveis a partir de gráficos como no caso de outros problemas de aprendizagem profunda, ainda para mais porque partíamos de um problema cuja medida de desempenho principal, o reward, não estava apropriadamente definido para o treino de DRL, pelo motivo que, no trabalho discutimos como e porque que este foi modificado.

Assim sendo, iremos analisar estes a partir de uma combinação dos rewards obtidos nos nossos ambientes, com os rewards verdadeiros do ambiente base, com o que pressentimos da aprendizagem dos modelos.

No que toca ao primeiro modelo PPO que referimos na secção Treino do Modelo conseguimos através da observação de algumas sessões de jogo realizadas pelo algoritmo conseguimos perceber que os modelos gerados desta aprendizagem eram os melhores até ao momento (resultados em 1). O modelo era capaz de explorar cerca de 3 a 4 salas para além da primeira o que era uma melhoria em relação aos modelos desenvolvidos por funções de recompensa ligeiramente diferentes. Isto não quer dizer a função de recompensa não podia ser melhorada, porque ainda estava longe de estar ótima. A otimização que reparamos que tinha de ser feita era uma alteração da atribuição da recompensa negativa por passar muito tempo no quarto atual para penalizarmos a passagem de muito tempo em qualquer quarto visitado. Esta alteração foi necessária, porque o algoritmo percebia que para não perder recompensa podia estar constantemente a alternar entre dois quartos adjacentes evitando assim receber uma recompensa negativa.

O segundo modelo que utiliza o algoritmo DQN já teve essa alteração da função de recompensa referida no paragrafo correspondente ao primeiro modelo PPO(resultados em 2). Mas embora a mudança da função da recompensa fosse necessária acabou por causar outro problema que foi o 'medo' de explorar por parte do algoritmo. O grupo identificou duas possíveis razões para este 'medo' se ter revelado no modelo. A primeira pode ser a mudança do algoritmo PPO para o algoritmo DQN. A segunda é o facto da recompensa negativa por chocar contra obstáculos ficou mais alta que a recompensa negativa por não explorar. Em termos de desempenho os modelos inicias acabam por conseguir explorar mais devido às decisões iniciais terem uma componente mais aleatória/exploratória e conseguiram explorar nos melhores casos 3 salas. Dito isto, percebemos que era preciso aumentar a penalização atribuída ao agente por ficar muito tempo na mesma sala.

O último modelo, com a pseudo segmentação do jogo, obteve os resultados observados em 3. Estes resultados estão partidos em vários gráficos devido ao facto de que o modelo foi treinado em diferentes sessões não contínuas.

Como podemos observar pela evolução dos valores da reward, durante os primeiros 40 mil steps, o modelo frequentemente tomava ações que levavam a rewards negativos e, observando o valor destes, os seus picos são usualmente provocados por rewards negativos do próprio jogo, como cair no subterrâneo ou bater num tronco, visto que estes têm valor muito inferior aos rewards negativos inseridos por nós. Ao longo do treino observamos no entanto uma enorme redução

destes picos, notando uma mais frequente presença de reward positiva, de valor baixo, ou seja, o valor de recompensa por nós inserido por explorar novas salas.

Com isto, é possível concluir que, embora o modelo continue muito aquém do esperado de um jogador normal, nunca tendo chegado a conseguir rewards de forma natural no jogo, coleção de tesouros, este conseguiu no entanto aprender a evitar recompensas negativas do próprio jogo, sendo capaz de evitar com ou mais intenção (perdendo vidas por vezes) estes obstáculos de modo a manter um score final de jogo neutro, mas este sendo conseguido sem uso de um loop em que o agente não explora novas salas. Embora um resultado melhor do que os modelos base explorados nos papers de DRL [5], cujos resultados se podem observar na figura 4, ficamos ainda com uma grande margem de evolução possível, principalmente quando comparado com algoritmos como o *Go-Explore*.

É importante realçar que, embora algoritmos como o *Go-Explore* e o *Agent-57* tenham conseguido resultados extraordinários no jogo, os resultados por estes fornecidos tiraram proveito de um tempo de treino ordens de magnitude (milhares de milhões de steps contra 1 ou 2 milhões no nosso caso) acima do nosso além de claro, um algoritmo DRL mais focado para este tipo de problemas com recompensas muito dispersas e que requerem muita exploração.

7 Trabalho Futuro

Neste trabalho foi tomada a decisão de não imitar o trabalho desenvolvido por outros autores, nomeadamente em [6], [7] e [9]. Tentamos desenvolver a nossa solução própria para o problema e, como seria de esperar, introduzir alguns conceitos propostos pelos autores dos artigos referidos.

Uma forma de melhorar o desempenho do algoritmo é introduzir mais métricas de avaliação do comportamento do agente. No algoritmo Go-Explore o próprio cenário é subdividido em porções mais pequenas e são dadas recompensas em função de, por exemplo, esperar que um buraco se feche ou que uma corda esteja ao alcance do jogador. Este *fine-graining* é uma opção óbvia para melhorar a performance do agente para esta tarefa em específico.

É também possível dizer ao agente onde estão os tesouros e que cenários contém que obstáculos, dado que esta informação é previamente conhecida. Estas opções, assim como a análise da própria informação que está em RAM, permitem com que o agente obtenha informação com um elevado grau de certeza, relativamente mais complicado de alcançar com a utilização de CNNs.

Em suma, existem bastantes melhorias que podem ser feitas. A própria exploração entre diferentes políticas de aprendizagem por reforço, quer sejam PPO, DQN, ou outras, pode introduzir melhorias no comportamento do agente que não foram ainda vistas ao longo do desenvolvimento. A divisão da estratégia em duas fases, uma de exploração e outra que parte de ambientes promissores para dar *exploit* de recompensas, pode ser uma estratégia a seguir, dado que está na base do algoritmo Go-Explore. Existe espaço para melhorar os resultados, sendo que o mais fácil é, à partida, aumentar o número de iterações de treino e a capacidade da rede CNN utilizada pelo modelo.

References

1. Repositorio da Implementação Desenvolvida
2. Atari - Gymnasium Documentation
3. Pitfall - Gymnasium Documentation
4. Pitfall - How Pitfall Builds its World. Uma exploração do uso da RAM no jogo Pitfall!.
5. Papers With Code : Pitfall
6. Ecoffet, A., Huizinga, J., Lehman, J. et al. Go-Explore: a New Approach for Hard-Exploration Problems. Uber AI Labs, 2019. https://arxiv.org/abs/1901.10995
7. Ecoffet, A., Huizinga, J., Lehman, J. et al. First return, then explore. Nature 590, 580–586 (2021). https://doi.org/10.1038/s41586-020-03157-9
8. On Pitfall, the best rollout of Go-Explore scores over 100,000 points (YouTube)
9. Badia, Adrià Puigdomènech, et al. "Agent57: Outperforming the atari human benchmark." International conference on machine learning. PMLR, 2020. https://arxiv.org/pdf/2003.13350v1.pdf
10. John Schulman, Filip Wolski, Prafulla Dhariwal, et al. "Proximal Policy Optimization Algorithms" https://arxiv.org/abs/1707.06347v2
11. 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
12. Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, et al. "Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents" https://arxiv.org/abs/1709.06009v2

A Anexos

NOOP	FIRE	UP
RIGHT	LEFT	DOWN
UPRIGHT	UPLEFT	DOWNRIGHT
DOWNLEFT	UPFIRE	RIGHTFIRE
LEFTFIRE	DOWNFIRE	UPRIGHTFIRE
UPLEFTFIRE	DOWNRIGHTFIRE	DOWNLEFTFIRE

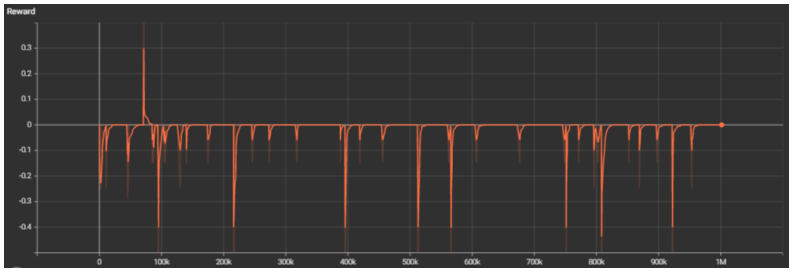
Table 1: Espaço de ações Pitfall.

NOOP	FIRE	UP	
RIGHT	LEFT	DOWN	
DOWNRIGHT	DOWNLEFT	RIGHTFIRE	LEFTFIRE

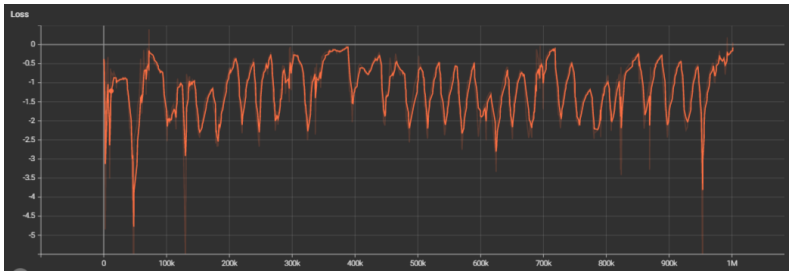
Table 2: Espaço de ações não redundantes Pitfall.

Layer	**Configuration**
Conv2d	Input Channels: n Output Channels: 32 Kernel Size: 3 Stride: 1 Padding: 1
ReLU	-
MaxPool2d	Kernel Size: 2 Stride: 2 Padding: 1
Conv2d	Input Channels: 32 Output Channels: 64 Kernel Size: 3 Stride: 1 Padding: 1
ReLU	-
MaxPool2d	Kernel Size: 2 Stride: 2
Conv2d	Input Channels: 64 Output Channels: 128 Kernel Size: 3 Stride: 1 Padding: 1
ReLU	-
MaxPool2d	Kernel Size: 2 Stride: 2 Padding: 1
Flatten	-

Table 3: Arquitetura Personalizada da CNN utilizada no modelo de DRL.



(a) gráfico de reward



(b) gráfico de loss

Fig. 1: Resultados do treino do 1º modelo

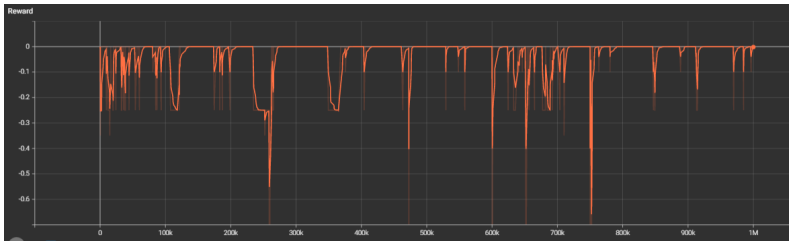
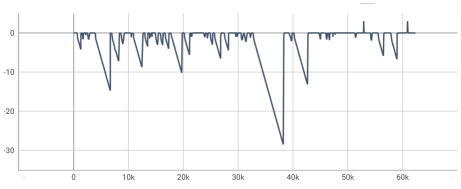
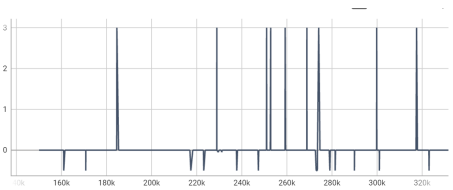


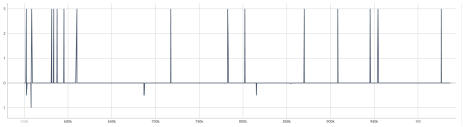
Fig. 2: Resultados do treino do 2º modelo



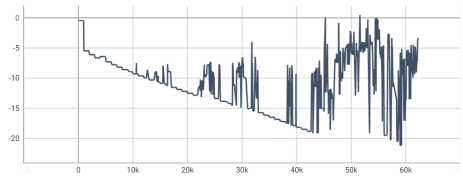
(a) gráfico de reward 1/3



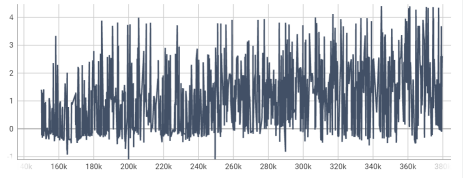
(b) gráfico de reward 2/3



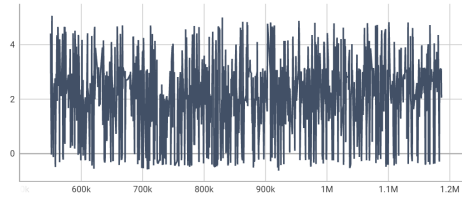
(c) gráfico de reward 3/3



(d) gráfico de loss 1/3



(e) gráfico de loss 2/3



(f) gráfico de loss 3/3

Fig. 3: Resultados do treino do 3º modelo

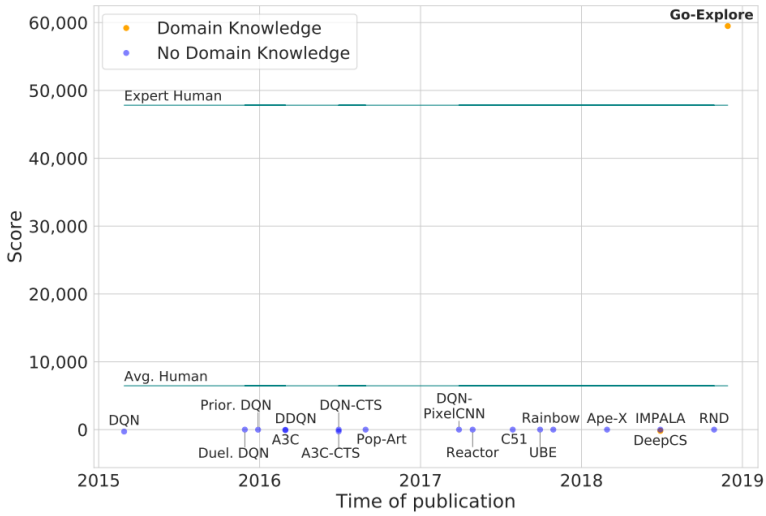


Fig. 4: Progresso da pontuação obtida no jogo Pitfall.