Universidade do Minho

Escola de Engenharia

Aprendização Profunda

Mestrado em Engenharia Informática

Grupo 8

Deep Reinforcement Learning em Jogos


Catarina Martins
PG50289


Eduardo Magalhães
PG50352


Laura Rodrigues
PG50542

Junho 2023

Índice

1. Introdução

Neste relatório será analisada a implementação de um modelo de *Deep Reinforcement Learning* para o jogo Super Mario Bros da OpenAI. O objetivo principal foi treinar um agente capaz de aprender a jogar Super Mario de forma autónoma, utilizando técnicas de aprendizagem profunda.

A escolha da alternativa B permitiu-nos aproveitar o ambiente de treino fornecido pela OpenAI Gym, uma biblioteca popular para o desenvolvimento e treino de agentes. O Gym forneceu-nos ambientes simulados que nos permitiram treinar e avaliar o nosso modelo.

Inicialmente, optámos por procurar implementações já existentes de modelos de *Deep Reinforcement Learning* para o Super Mario. Isso permitiu-nos ter uma base de referência para entender o funcionamento dos modelos atuais e seus resultados. Tirámos partido de um tutorial disponível em [1] e explora-mos novas formas de aprimorar o modelo, através de modificações nos algoritmos e no pré processamento dos dados.
A implementação está disponível em: `https://github.com/Laura-Rodrigues/AP`.

1.1 Principais Objetivos

O objetivo geral deste projeto envolve o desenvolvimento de um agente, neste caso o **Mario**, que seja capaz de jogar o jogo *Super Mario Bros* de forma autónoma. O agente deve aprender a tomar decisões com base no estado atual do ambiente, buscando maximizar as recompensas obtidas ao longo do jogo. Para isso foi necessário:

- Implementar um ambiente de simulação do jogo usando a biblioteca *OpenAI Gym.*
- Desenvolver um agente baseado em algoritmos de **Deep Reinforcement Learning**,como o **DDQN** *(Double Deep Q-Network)*, **PPO** *(Proximal Policy Optimization)* e **DQN** (*Deep Q-Network*).
- Treinar o agente no ambiente simulado, permitindo que o mesmo explore o ambiente, aprenda com as interações e redefina as suas estratégias ao longo do tempo.
- Avaliar o desempenho do agente.
- Analisar e interpretar os resultados obtidos, identificando as limitações e possíveis melhorias do sistema.

2. Metodologia

De uma maneira geral a metodologia usada foi inspirada no modelo da metodologia SEMMA (Sample, Explore, Modify, Model, & Assess). Assim, de uma forma mais detalhada a nossa metodologia é constituída pelos seguintes passos:

- **Preparação dos dados**: Nesta fase são carregados os dados do ROM para um ambiente.
- **Visualização de dados**: Esta fase tem como objetivo analisar os dados fornecidos, ou seja, é nesta etapa que se pretende compreender os tipos de dados disponíveis (frames, etc.) para melhor compreender como simplificar o problema.

- **Tratamento de dados**: Após a visualização de dados é necessário efetuar o tratamento dos mesmos. Nesta fase foi importante o uso de Wrappers de modo a obter estruturas de dados mais simples e assim evitar sobrecargas durante os períodos de treino dos modelos.
- **Definição do modelo**: Nesta etapa é construída a rede neuronal a utilizar. Desta forma, são determinadas todas as camadas da rede e outros parâmetros necessários.
- **Treino do modelo**: Tendo a rede neuronal construída é necessário treiná-la com os dados fornecidos. Assim sendo, optámos por treinar os modelos para diferentes números de timesteps, diferentes redes neuronais de modo a perceber qual o melhor para o Mario.
- **Avaliação de performance**: Para verificar o sucesso do treino da rede é usado um método próprio da biblioteca Stable Baselines 3 (*evaluate_policy*).
- **Visualização dos resultados**: Análise de parâmetros importantes como loss, rewards, etc.

3. Descrição e Exploração dos Dados

No contexto do jogo **Super Mario Bros**, o ambiente consiste em tubos, cogumelos e outros componentes. Quando o Mario executa uma ação, o ambiente responde fornecendo o próximo estado, recompensa e outras informações relevantes.[1].

O **treino** do agente **Mario** ocorre através de várias iterações, chamadas episódios. Em cada episódio, o agente executa ações com base na sua política de ação atual. Durante as primeiras iterações, o agente explora diferentes ações de forma aleatória para aprender sobre o ambiente. Conforme o treino progride, o agente começa a utilizar a sua política de ação aprendida para tomar decisões mais otimizadas.

Após o treino do agente, avaliamos os resultados do agente **Mario**. É medido o desempenho do agente com base em métricas.

4. Arquiteturas e Implementações

4.1 Ambiente de simulação

No projeto foi utilizado o OpenAI Gym, que é uma ferramenta amplamente utilizada para o desenvolvimento, treino e avaliação de algoritmos de aprendizagem por reforço. Neste trabalho, utilizou-se um ambiente específico disponibilizado pelo Gym, denominado "SuperMarioBros-1-1-v0".

De modo a não sobrecarregar os sistemas durante os processos de treino e a torná-los mais rápidos houve necessidade de aplicar pré-processamento aos dados do ambiente usado antes de os enviar para o agente. Para tal são utilizados alguns *"wrappers"* de **pré-processamento** que serão elaborados adiante.

Ao inicializar o ambiente do jogo Super Mario Bros é permitido que o agente interaja com ele. Após inicializar o ambiente e definir as ações disponíveis, o código reinicia o ambiente e obtém o estado inicial. Em seguida, o agente executa uma ação específica no ambiente e o código captura o próximo estado do ambiente (**'next_state'**), a recompensa obtida com a ação **('reward')**, um sinalizador que indica se o jogo foi concluído **('done')**, um sinalizador que indica se o tempo limite para a execução

dação foi alcançado **('trunc')**, etc. Estas informações podem ser usadas para acompanhar o processo do agente.

4.2 Pré-processamento do ambiente

Ao pré-processar o ambiente são utilizados wrappers para transformar os dados antes de serem enviados para o agente. Cada wrapper pretende simplificar o formato original de cada estado, ou seja, [3, 240, 256] (sendo que o primeiro valor representa os canais de cor e os restantes o tamanho da imagem):

- **GrayScaleObservation**: Converte as imagens RGB do ambiente em imagens em escala de cinzas. Isto reduz o número de canais de côr sem se perder informações úteis.
- **ResizeObservation**: Redimensiona cada observação para uma imagem quadrada menor. Esta redução facilita o processamento.
- **SkipFrame**: Permite saltar um número específico de frames intermédias. Como os frames consecutivos geralmente não diferem muito entre si é possível passar à frente alguns sem perder muita informação.
- **FrameStack**: Agrupar frames consecutivos do ambiente num único ponto de observação. Sobrepondo as suas informações facilita a perceção de alguns movimentos do agente.

Após aplicar o pré-processamento, obtemos o seguinte:

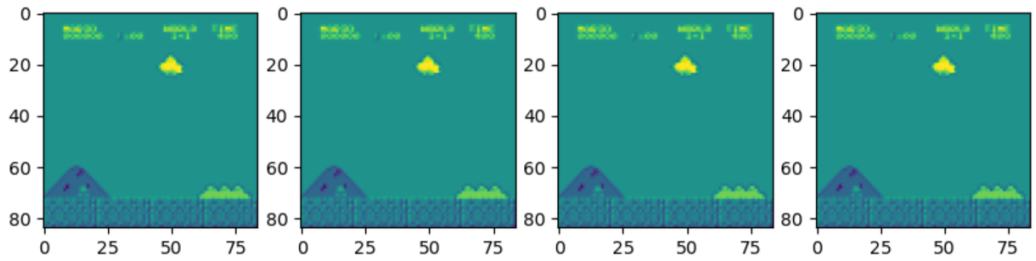

Figura 4.1: Ambiente após pré-processamento

4.3 Implementação modelo - tutorial disponibilizado pelo PyTorch

4.3.1 Agente Mario

Relativamente ao agente **Mario** representado pela classe *Mario* deve ser capaz de aprender a escolher as ações que maximizam as recompensas cumulativas. Esta está também encarregue de inicializar e configurar o agente com base nos hiperparâmetros fornecidos:

- **'state dim'**: Dimensão do espaço de estados do ambiente do jogo.
- **'action_dim'**: Dimensão do espaço de ações disponíveis para o agente.
- **'exploration_rate'**: Representa a taxa de exploração do agente. É inicializada a 1 (exploração máxima) e é diminuida ao longo do tempo.

- **'exploration_rate_decay'**: É o fator de decaimento dessa taxa (parâmetro usado para controlar a redução gradual da taxa de exploração do agente ao longo do treino).
- **'exploration_rate_min'**: É o valor mínimo que a taxa de exploração pode atingir. Isto garante que o agente mantenha uma taxa mínima de exploração para evitar ficar estagnado numa política sub-ótima.

4.3.2 Características do agente

O agente **Mario** deve ser capaz de agir de acordo com a política de ação ótima com base no estado atual do ambiente, lembrar-se das suas experiências, sendo uma experiência definida como (estado atual, ação atual, recompensa, próximo estado) e ainda aprender uma política de ação melhor ao longo do tempo.

Tal como mencionado em [1], o agente possui assim as seguintes funcionalidades principais:

Agir de acordo com a política de ação ótima baseada no estado atual

A função **act** é responsável por selecionar uma ação para o agente Mario com base no estado atual do ambiente (**'state'**). Segue uma estratégia epsilon-greedy. A taxa de exploração é atualizada e diminuída ao longo do tempo. O agente pode escolher entre explorar o ambiente de forma aleatória (**exploration**) ou explorar as ações ótimas com base na política aprendida (**exploitation**). A atuação aleatória permite encontrar um equilíbrio de modo a que o agente explore o ambiente e descubra novas estratégias, enquanto que a atuação baseada na política aprendida visa maximizar as recompensas. O processo funciona da seguinte forma:

- **Explore**: Se um número aleatório entre 0 e 1 for menor que a taxa de exploração atual, uma ação aleatória é escolhida entre as ações possivéis.
- **Exploit**: Caso contrário, o agente utiliza a rede neuronal para estimar os *Q-values*. A ação com o valor mais alto é selecionada.

Recordar experiências - Armazenamento e Recuperação

As funções **cache()** e **recall()** servem como o processo de memória do Mario. A função de armazenamento (cache) guarda experiências passadas, incluindo o estado atual, a ação realizada, a recompensa obtida, o próximo estado e se o jogo foi concluído. Na função **recall()** o agente seleciona aleatoriamente um conjunto de experiências da sua memória e usa-as para aprender o jogo.[1].

Aprender uma política melhor ao longo do tempo

O agente utiliza o algoritmo **DDQN** (*Double Deep Q-Network*) para aprender a política ótima de ações.

- **Algoritmo DDQN**: Este algoritmo utiliza duas redes neuronais convolucionais para aproximar a função de valor de ação ótima. O objetivo é permitir que as redes aprendam diferentes aspetos do ambiente de forma independente.

  A rede ***online*** faz previsões das ações com base no estado atual do ambiente, enquanto a rede ***target*** é uma referência fixa para calcular as metas de aprendizagem. Durante o treino, a rede *online* é atualizada utilizando um algoritmo de otimização e uma função de perda adequados.

4.3.3 Rede Neuronal

O modelo do PyTorch define a rede **MarioNet** que é uma mini CNN (*Convolutional Neural Network*). Esta é responsável por prever a melhor ação. São definidos os hiperparâmetros relacionados à exploração do agente, como referido anteriormente. O atributo **'curr_step'** representa o número atual de passos do agente sendo inicializado a 0.

Esta rede recebe como entrada um estado do ambiente do agente e gera uma saída que representa os valores de ação ótimos. A estrutura da rede é a seguinte:

- A entrada é um tensor com dimensões [canais,altura,largura], onde **canais** representa o número de canais de cor, e altura e largura são as dimensões da imagem.
- A rede possui três camadas de convolução 2D, cada uma seguida por uma função de ativação ReLU. Estas camadas são responsáveis por aprender recursos visuais do estado de entrada.
- A saída da rede é uma camada totalmente conectada sem função de ativação, que determina a reação para ação possível no ambiente.

4.4 Outras Implementações

De modo a conseguir testar outros algoritmos e o seu impacto na performance do agente recorreu-se à biblioteca Stable Baselines 3. Esta permitiu aplicar Deep Reinforcement Learning de maneira mais otimizada ao jogo Super Mario Bros.

4.4.1 Algoritmo PPO - *Proximal Policy Optimization*

É um algoritmo de aprendizagem por reforço que visa otimizar as políticas de ação num ambiente. Foi introduzido pela OpenAI e tornou-se popular devido à sua eficácia e relativa simplicidade em comparação com outros [2]. A implementação geral do algoritmo envolve os seguintes passos:

- **Inicialização da política**: Em seguida, é necessário inicializar a política, que é uma função que mapeia as observações do ambiente para ações. A política utilizada é 'CnnPolicy' (Convolutional Neural Network Policy), que é uma política pré-definida no *stable_baselines3*, esta mostrou-se mais adequada para lidar com observações baseadas em imagens, o que se alinhava bem com o resultado final do pré-processamento realizado. Ele utiliza redes neurais convolucionais para extrair recursos espaciais das imagens de entrada, o que é particularmente útil para ambientes visuais como Super Mario Bros.
- **Configuração dos parâmetros**:O *PPO* possui vários parâmetros que podem ser ajustados para controlar o processo de otimização, como a **learning rate** e número de iterações.
- **Recolha de dados**: O agente interage com o ambiente recolhendo dados do treino.
- **Atualização da política**: Com base nos dados recolhidos, a política é atualizada para melhorar o desempenho do agente.

Rede

Uma vez que este algoritmo apresentou melhor performance optou-se por tentar melhorá-lo ainda mais tornando a rede (previamente igual à do modelo base) mais robusta.

A implementação do PPO no projeto envolveu, assim, a criação de um rede neuronal convolucional personalizada. A implementação define uma classe **'Custom_CNN'** que herda de **'BaseFeaturesExtractor'**. A rede modificada é composta por 3 camadas convolucionais e recebe

como entrada observações do ambiente, que podem ser representadas como tensores com dimensões CxHxW(canais, altura e largura). Após as camadas convolucionais, é aplicada uma camada de *max pooling* (**'nn.MaxPool2d'**) para reduzir a dimensionalidade da saída. Após esta camada, é aplicada uma camada de *Flatten* seguida de 3 camadas lineares totalmente conectadas.

4.4.2 Algoritmo DQN - *Deep Q- Network*

O **DQN** é um algoritmo que combina técnicas de Aprendizagem Profunda (*Deep Learning*) e Aprendizagem por Reforço (*Reinforcement Learning*) e é eficaz quando o espaço de ações e estados são amplos.

Mais uma vez, o objetivo do **DQN** é aprender uma política de ação ótima, ou seja, que maximize a recompensa cumulativa, aproximando e otimizando a função Q. Para isso, é utilizada uma rede neuronal profunda que recebe como entrada o estado atual e estima os valores Q para todas as ações possíveis. O agente seleciona a ação com maior valor Q estimado.

A otimização da rede neuronal tira partido do algoritmo Gradiente Descendente (*Gradient Descent*) para minimizar a diferença entre os valores Q estimados e os valores Q reais.

No treino da **DQN** são definidos vários hiperparâmetros, como a taxa de aprendizagem (**'learning_rate'**), taxa de exploração,etc.

6. Resultados obtidos

Com base nos resultados obtidos, foi possível otimizar o agente. Isto envolveu ajustar os hiperparâmetros do algoritmo, modificar a arquitetura da rede neuronal ou explorar outras técnicas. Assim, como melhores resultados apresentam-se os do algoritmo PPO, após correr o modelo durante 2500000 *timesteps*.

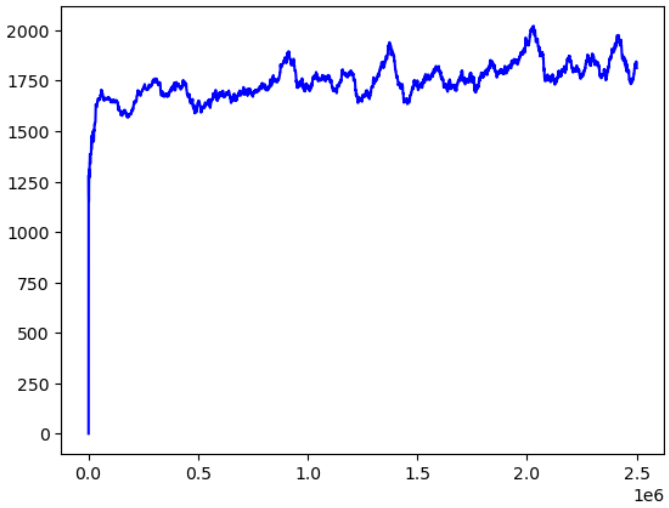

Figura 5.1: Média de Reward/Timesteps

Como podemos reparar na figura acima a *reward* obtida ainda se encontra a aumentar. Logo, podemos concluir que o modelo ainda poderá ser treinado durante uma maior número de *timesteps* com o intuito de melhorar o mesmo e o agente conseguir ultrapassar o nível em questão.

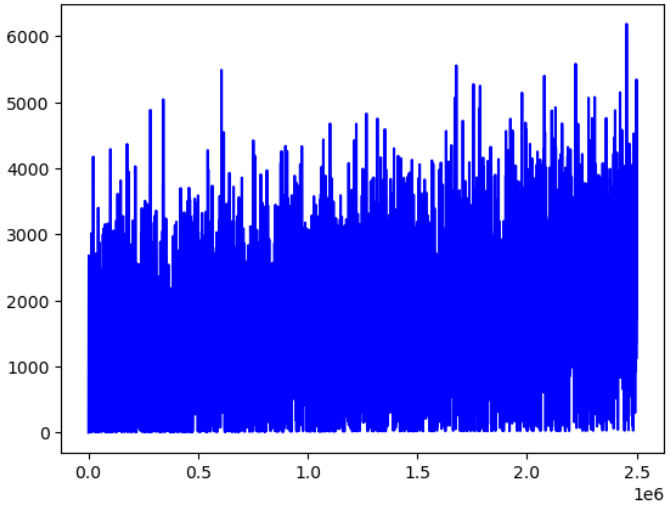

Figura 5.2: Loss/Timesteps

O gráfico da loss apresentou valores que não estariam de acordo com as expectativas. Enquanto se esperava que esta variasse entre 0 e 1, a loss atingiu valores bastante elevados que o grupo teve dificuldade em explicar. Do mesmo modo se estranhou a diminuição da distância percorrida.

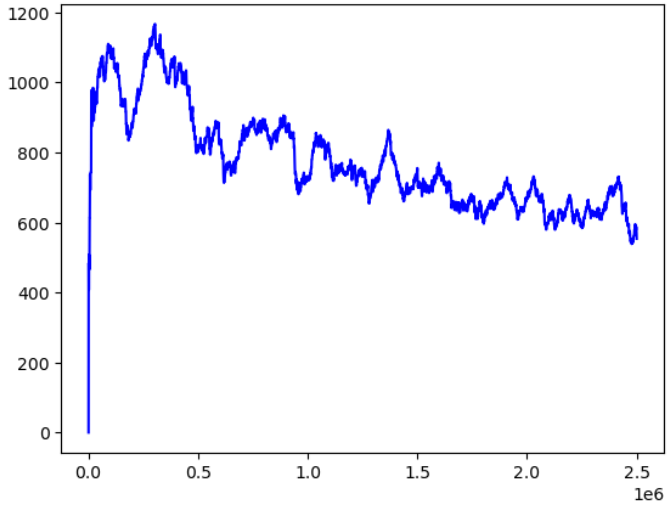

Figura 5.3: Distância média percorrida/Timesteps

Concluímos que o modelo deveria ter sido treinado durante um maior número de timesteps de modo a que o agente consiga concluir o nível em questão.

7. Sugestões e Recomendações

No contexto deste projeto foi explorado o uso de algoritmos de **Deep Reinforcement Learning** como o **DDQN** *(Double Deep Q-Network)*, **PPO** *(Proximal Policy Optimization)* e **DQN** (*Deep Q-Network*), no entanto é importante ressaltar que existem outras abordagens que poderiam ter sido exploradas para enriquecer ainda mais esta pesquisa. Por exemplo:

- **Considerar outros algoritmos de Reinforcement Learning:** Além dos algoritmos utilizados, poderíamos ter considerado outros algoritmos e realizar uma análise comparativa entre eles, por exemplo, o algoritmo do *'stable_baselines3'*, A2C, conhecido também por ter boas performances.
- **Considerar outras arquiteturas das redes neuronais:** Explorar outras arquiteturas mais profundas, por exemplo, redes mais complexas.
- **Ajustar os hiperparâmetros:** Os hiperparâmetros, como **learning rate** e taxa de exploração têm um impacto significativo no desempenho e na velocidade de convergência dos algoritmos.
- **Analisar a influência dos wrappers:** Os wrappers desempenham um papel crucial na preparação dos dados de entrada para a rede neuronal. Realizar uma análise comparativa dos diferentes wrappers seria interessante neste contexto.

8. Conclusões

Concluindo, a elevada complexidade destes ambientes e algoritmos requereu uma grande capacidade computacional à qual houve dificuldade em dar resposta. Após mais de 24h de treino apenas conseguimos uma performance medíocre no nosso melhor algoritmo PPO, na qual o agente não conseguiu completar um nível inteiro.

Apesar deste aspeto, considera-se que este trabalho trouxe uma maior compreensão sobre o funcionamente de redes neuronais e Deep Reinforcement Learning.

Bibliografia

[1] Feng, Y., Subramanian, S., Wang, H., Guo, S. Train a Mario-playing RL Agent. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html

[2] Proximal Policy Optimization. https://spinningup.openai.com/en/latest/algorithms/ppo.html