Estudo de caso de Deep Q-Learning

Vítor Kei Taira Tamada

Curso:
Bacharelado em Ciência da Computação

Orientador:
Prof. Dr. Denis Deratani Mauá

São Paulo, janeiro de 2019

Estudo de caso de Deep Q-Learning

Esta é a versão revisada da monografia elaborada pelo aluno
Vítor Kei Taira Tamada

Resumo

TAMADA, V. K. T. **Estudo de caso de Deep Q-Learning**. Trabalho de Conclusão de Curso - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Deep Q-Learning é uma técnica de aprendizado de máquina que une rede neural convolucional com aprendizado por reforço para ensinar uma inteligência artificial a obter sucesso em um ambiente recebendo apenas imagens dele como entrada. Enquanto a rede é responsável pela visualização da imagem, detecção de características e tomada de decisão, o aprendizado por reforço faz a avaliação da ação executada por meio da recompensa recebida. Essa forma de se aprender se assemelha a como uma pessoa aprende a realizar uma tarefa nova sem receber ajuda ou instruções, como jogar um jogo eletrônico novo.

O objetivo do trabalho foi estudar e avaliar a capacidade de uma inteligência artificial aprender e ter sucesso no ambiente que está inserida, assim como as dificuldades de implementar e obter bons resultados quando o *Deep Q-Learning* é utilizado. Por não ser uma técnica abordada nas disciplinas de inteligência artificial da graduação, também foi um estudo mais aprofundado de aprendizado de máquina, redes neurais e aprendizado profundo.

Os experimentos foram realizados em três ambientes de características e graus de complexidade distintos para verificar a flexibilidade de se ter sucesso com essa técnica e da dificuldade em obtê-lo: *Gridworld*, *Pong* do Atari 2600, e *Asteroids* do Atari 2600, sendo o primeiro o mais simples, com poucos estados, regras, e ações possíveis, e o último o mais complexo. O desempenho foi avaliado pelo sucesso de se alcançar o objetivo, no caso do *Gridworld*, e pela pontuação obtida ao fim do jogo, no caso do *Pong* e do *Asteroids*.

Utilizando diferentes arquiteturas de rede e números de episódios de treinamento para cada ambiente, o agente obteve bons resultados no *Gridworld* e no *Pong*, mas não no *Asteroids*. Trabalhos relacionados indicam que *Asteroids* poderia ter uma pontuação melhor e até obter sucesso se as devidas otimizações, escolhas de hiper-parâmetros e tempo suficiente de treinamento fossem utilizados.

Palavras-chave: inteligência artificial, deep q-learning, estudo de caso

Sumário

Capítulo 1

Introdução

Um tipo muito conhecido de inteligência artificial, ou IA, dos dias atuais é o que controla oponentes em jogos eletrônicos. Na maior parte dos casos, elas seguem um conjunto pré-determinado de regras escritas pelo desenvolvedor com o intuito de criar um desafio para o jogador. Entretanto, por mais que seja possível fazer a IA ter capacidades muito acima de seres humanos para jogar, elas não possuem a capacidade de se adaptar ao como seres humanos fazem para melhorar seu desempenho. Quando pessoas não recebem ajuda externa ou leem um manual, normalmente elas aprendem e se adaptam explorando o jogo, descobrindo o que as ações fazem e suas respectivas consequências. Ao invés de explicitar as regras que a máquina deve seguir, é possível deixá-la aprender as que considerar melhor, similar a pessoas, por meio de **aprendizado por reforço**.

Entretanto, por mais que um computador consiga aprender como um ser humano, ele normalmente não consegue enxergar como um. Uma pessoa consegue inferir o que é inimigo e o que é terreno quando aparece na tela em poucos movimentos ou a partir de experiências passadas com jogos diferentes. Para um computador, um pixel que mude de posição já faz ele não conseguir mais distinguir o que está vendo, tendo que reaprender a cada nova combinação de pixels detectada. Em outras palavras, seres humanos conseguem abstrair as informações que enxergam com facilidade, enquanto computadores não. Se IAs não conseguem mais identificar um objeto na tela por causa de um pixel que esteja diferente, como sistemas de detecção de imagem funcionam? Utilizando uma variante de rede neural profunda chamada de **rede neural convolucional** (*convolutional neural network* (CNN)), é possível fazer uma máquina abstrair essas informações e inferir que um objeto em diferentes lugares da tela, assumindo diferentes tamanhos, são o mesmo - ou seja, visualizar e compreender imagens semelhante a uma pessoa.

Unindo a forma de se aprender de aprendizado por reforço com a capacidade de análise de imagens de redes neurais convolucionais, obtem-se uma técnica chamada ***Deep Q-Learning*** [MKS+13]. Essa forma de aprendizado permite que uma inteligência artificial aprenda a ter sucesso em um ambiente apenas recebendo imagens como entrada, assim como uma pessoa faria para aprender um jogo novo quando sua única fonte de informações é a tela de um monitor.

Motivado pelo interesse nessa técnica de aprendizado de máquina, o objetivo deste trabalho foi fazer um estudo de caso quando uma *Deep Q-Network* (DQN) é utilizada por uma inteligência artificial em três ambientes com características e graus de complexidade distintos: Gridworld, *Pong*, e *Asteroids*. O estudo buscou analisar a capacidade de um agente obter bons resultados utilizando este método, e as dificuldades enfrentadas no processo, assim como aprofundar o conhecimento em aprendizado de máquina, redes neurais e aprendizado profundo.

Os ambientes foram emulados utilizando as ferramentas Gym [1] e Gym-Retro [2], o código foi escrito em

[1] https://gym.openai.com/
[2] https://blog.openai.com/gym-retro/

Python3 [3], e a rede neural foi construída com o arcabouço TensorFlow [4]. Os resultados obtidos pelo agente treinado foram comparados com um aleatório e, no caso do *Pong* e do *Asteroids*, com um ser humano jogando. No *Gridworld*, o agente consegui obter resultados positivos de maneira consistente, mesmo com algumas pequenas alterações nos hiper-parâmetros. No *Pong*, a pontuação final que a inteligência artificial conseguiu ao final de cada partida cresceu lentamente ao longo do treinamento, mostrando ser capaz, ainda que com dificuldade, de ter sucesso. No *Asteroids*, por outro lado, não houve indícios de melhorias ao longo dos treinamentos realizados, com desempenho inferiores a um agente aleatório.

Inicialmente, no capítulo 2, serão explicados os fundamentos teóricos utilizados neste trabalho, como rede neural convolucional, aprendizado profundo e *Deep Q-Learning*, para a construção da inteligência artificial. Em seguida, no capítulo 3, serão detalhados os três ambientes de treinamento, a arquitetura das redes neurais, e, por último, como foram os experimentos. Por fim, no capítulo 4, são apresentados e analisados os resultados obtidos pelos agentes.

[3] https://www.python.org/
[4] https://www.tensorflow.org/

Capítulo 2

Fundamentos

Para se criar e treinar uma inteligência artificial, diversos arcabouçous são necessários. Por um lado, existe a parte teórica e matemática na qual a inteligência se baseia para aprender. Por outro, do lado computacional, existem as bibliotecas que auxiliam no desenvolvimento, efetuando as operações necessárias e, neste trabalho em particular, emulando os ambiente. Este capítulo tem o intuito de familiarizar o leitor com a teoria e técnicas utilizadas na modelagem e treinamento da inteligência artificial deste trabalho.

2.1 Redes neurais

Redes neurais artificiais, mais conhecidas como redes neurais, são uma forma de processamento de informação inspirada no funcionamento do cérebro. Assim como o órgão no qual foram baseadas, elas possuem uma grande quantidade de elementos de processamento de informação conectados entre si, chamados de neurônios, que trabalham em conjunto para resolver problemas. Dado que aprendem com exemplos, é considerada uma técnica de aprendizado supervisionado. Elas são muito utilizadas como aproximadoras de funções desconhecidas, que não são facilmente modeláveis matematicamente.

Com os avanços nos estudos dessa técnica nos últimos anos, diversos tipos diferentes de redes neurais foram desenvolvidos, como redes neurais convolucionais, o tipo utilizado neste trabalho, e redes neurais de memória de curto-longo prazo[1] (*Long/Short Term Memory*, LSTM), que não será abordada. Apesar de cada uma ter sua particularidade, redes neurais clássicas possuem duas características principais: a estrutura dividida em **camadas**, e os **neurônios** que as compõe. Existem redes que não são consideradas clássicas pela falta de estrutura em camadas, como Redes de Hopfield (*Hopfield Network*) e Máquinas de Boltzmann (*Boltzmann Machine*), que também não serão discutidas neste trabalho.

As camadas de uma rede neural são compostas por neurônios e podem ser separadas em três categorias: **entrada**, **oculta**, e **saída**. Quando um dado é suprido à rede, ele vem no formato de um vetor de forma que os neurônios da camada de **entrada** recebem seus elementos para serem processados ao longo da rede; as **ocultas** realizam o processamento do dado de entrada; e a de **saída** devolve um vetor de números que representa o resultado da rede neural para a entrada dada. Enquanto o formato da entrada e da saída da rede neural determinam seus respectivos números de neurônios, a quantidade de nós nas camadas ocultas são arbitrários, normalmente definidos por meio de tentativa e erro.

Neurônios, ou nós, são funções que recebem, como entrada, a saída de cada neurônio da camada anterior e devolvem um número cujo significado e como são usados variam de acordo com o objetivo da rede. Cada neurônio das camadas ocultas representa uma característica detectada ao longo do treina-

[1] Tradução livre feita pelo autor

mento. Se essa característica estiver presente na camada de entrada, então o neurônio correspondente será **ativado**. A ativação de um ou mais neurônios pode levar à ativação de outros neurônios na camada seguinte e assim sucessivamente. A forma mais comum de se utilizá-los em redes neurais é por meio da **combinação linear** dos valores dos neurônios de uma camada seguido por uma **função de ativação**.

Para um neurônio n de uma camada k, $k > 0$, que não seja a de entrada, seja N, $N > 0$, o número de neurônios da camada anterior $k-1$, w_i, $i = 1, ..., N$, o peso do i-ésimo neurônio da camada $k-1$, e a_i, $i = 1, ..., N$, o valor de saída do i-ésimo neurônio da camada $k-1$. A combinação linear dos neurônios é dada por:

$$w_1 a_1 + w_2 a_2 + ... + w_n a_n \tag{2.1}$$

Antes de ser passada para a função de ativação, é comum, mas não obrigatório adicionar à somatória 2.1 um viés [2] b, uma constante que ajuda a melhor aproximar da função desconhecida. A função de ativação realiza uma transformação não-linear no número que recebe como argumento, a soma de 2.1 com o viés neste caso, para mapear seu resultado em um intervalo e determinar se o neurônio n deve ser ativado ou não. Tanto a transformação quanto a forma de mapear e o intervalo são determinados pela função utilizada.

Esse processo, que consiste em várias multiplicações e adições, precisa ser feito em todos os neurônios das camadas da rede neural, o que pode gerar problemas de ponto flutuante e de tempo de processamento. Portanto, é conveniente realizar esse processo utilizando matrizes, uma vez que existem diversas bibliotecas com funções que otimizam operações matriciais. Para um neurônio n de uma camada k, $k > 0$, que não seja a de entrada, seja N, $N > 0$, o número de neurônios da camada $k-1$, W uma matriz tal que $w_{n,j}$, $j = 1, ..., N$, o peso do j-ésimo neurônio da camada $k-1$, a_{k-1} o vetor com os valores de saída dos neurônios da camada $k-1$, e b o viés. Os valores dos neurônios da camada k podem ser representados como resultado da seguinte operação:

$$W a_{k-1} + b \tag{2.2}$$

As funções de ativação mais comuns são a sigmoide (curva logística), ReLU (*Rectified Linear Unit*) e ELU (*Exponential Linear Unit*), sendo a sigmoide a mais antiga e a ELU a mais recente.

O próximo passo é entender como os valores dos neurônios e os respectivos pesos são utilizados para tentar devolver a resposta correta. Como rede neural é um tipo de aprendizado supervisionado, os exemplos inseridos nela possuem rótulos, saídas esperadas. Para que o computador saiba o quão próxima sua resposta estava da correta, é definida uma função de erro, também conhecida como função de custo. Naturalmente, quanto maior for o erro, mais incorreta foi a previsão.

Otimizar os pesos de forma a reduzir os erros obtidos com os exemplos parece ser o melhor caminho para melhorar o modelo, mas isso não é necessariamente verdade. Os exemplos utilizados nessa etapa compõe o **conjunto de treinamento**. Se o modelo tiver zero de erro em relação a esse conjunto, ele estará sofrendo de *overfitting*, ou seja, a rede neural se adequa tanto ao conjunto de treinamento que saberá o que fazer apenas nele, estando possivelmente muito diferente da função que se deseja aproximar. Para determinar o grau de *overfitting*, utiliza-se um **conjunto de validação**, exemplos diferentes do treinamento que são supridos à rede neural ocasionalmente e que não são utilizados para atualização dos pesos. Se o erro do conjunto de treinamento diminuir, mas o do conjunto de validação não, então o modelo estará começando a sofrer de *overfitting* e o treinamento deve ser interrompido. Por fim, utiliza-se um **conjunto de testes**, exemplos escolhidos pela mesma distribuição de probabilidade que o conjunto de treinamento para fazer uma avaliação do modelo obtido após o treinamento. Idealmente, um modelo

[2] *bias*, em inglês

com erro baixo nos três conjuntos não tem problemas de *overfitting* e consegue resolver o problema para o qual foi criado.

De forma resumida, uma rede neural clássica aprende recebendo uma série de números como entrada (exemplo) e devolve uma saída; calcula-se o quão errada essa saída está em relação ao desejado para aquela determinada entrada (rótulo do exemplo), e ajusta os pesos conforme a necessidade para minimizar o erro; se a arquitetura da rede tiver sido contruída adequadamente, a IA deverá aprender a resolver o problema para o qual foi feita após exemplos suficientes serem supridos.

2.2 Aprendizado profundo

Como explicado anteriormente, redes neurais podem ser divididas em três tipos distintos de camadas: entrada, ocultas, e saída. Enquanto existe apenas uma camada de entrada e uma de saída, é possível haver mais de uma camada oculta. Se houver muitas ocultas, a rede neural passa a ser chamada de **rede neural profunda**. Atualmente, não existe uma definição exata de quantas camadas são necessárias para uma rede ser classificada como profunda e, mesmo que houvesse, esse número provavelmente mudaria com o passar do tempo.

Uma rede neural profunda que segue o modelo apresentado na seção anterior é chamada de *feedforward* e é o mais típico de *deep learning*. Ele recebe esse nome pois a informação flui da entrada para a saída sem haver conexões de *feedback* para que a previsão seja feita. Este tipo de rede neural forma a base para **redes neurais convolucionais**.

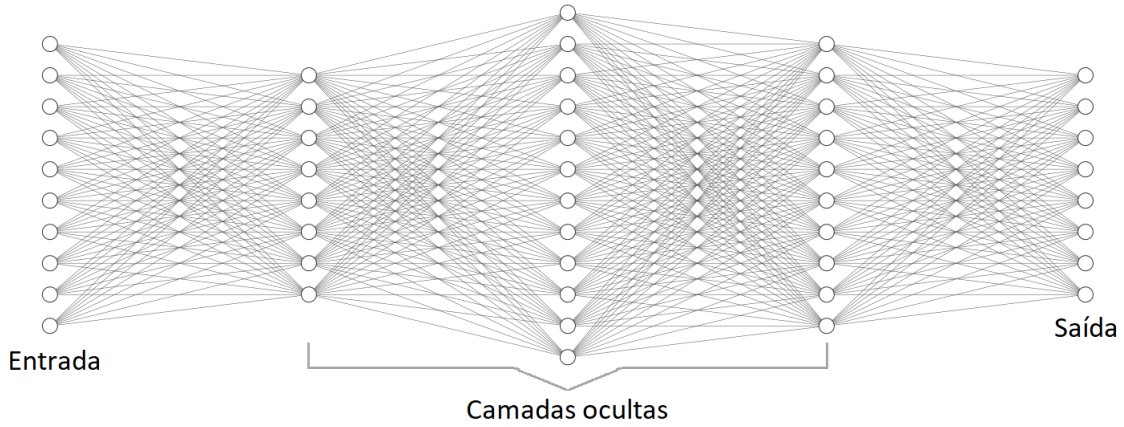

Figura 2.1: Esquema de uma rede neural profunda do tipo *feedforward*. Número de nós e camadas arbitrários para melhor representação. Diagrama feito em http://alexlenail.me/NN-SVG/index.html

2.3 Redes neurais convolucionais

Redes neurais convolucionais (*Convolutional Neural Networks*, CNN) são um tipo de rede neural profunda do tipo *feedforward* especializada em visualização e processamento de imagens. Sendo uma rede neural clássica, ela possui estrutura dividida em camadas que são formadas por neurônios, com apenas as camadas ocultas funcionando de maneira diferente. Elas podem ser de dois tipos diferentes: **convolucionais** ou ***fully-connected***. Enquanto as *fully-connected* funcionam conforme descrito na seção 2.1, com todos seus neurônios conectados a todos da camada anterior, as convolucionais operam de uma forma diferente, tendo duas partes principais: a **etapa convolucional**, que dá o nome ao tipo da rede; e a **função de ativação**.

A detecção de características de uma CNN é feita na **etapa convolucional** pelos filtros convolucionais, também chamados de *kernel*, matrizes muito menores que a de entrada, mas com mesmo número de dimensões, que em muitos casos é 3: altura, largura, e canais de cor. Os filtros convolucionais são os elementos básicos no processamento de imagens, pois são responsáveis por aprender e detectar características nas imagens de entrada. Eles são deslizados pela matriz de entrada, realizando o produto escalar com a área sobre a qual estão e criando uma nova matriz chamada de *feature map*. Essa operação é feita para cada filtro e é chamada de convolução. Após os *feature maps* de cada filtro serem feitos, eles são juntados de forma que a saída da camada tem o formato NxMxD, onde N e M são a altura e largura do *feature map* e D é o número de *kernels*. A imagem 2.2 ilustra como convolução ocorre.

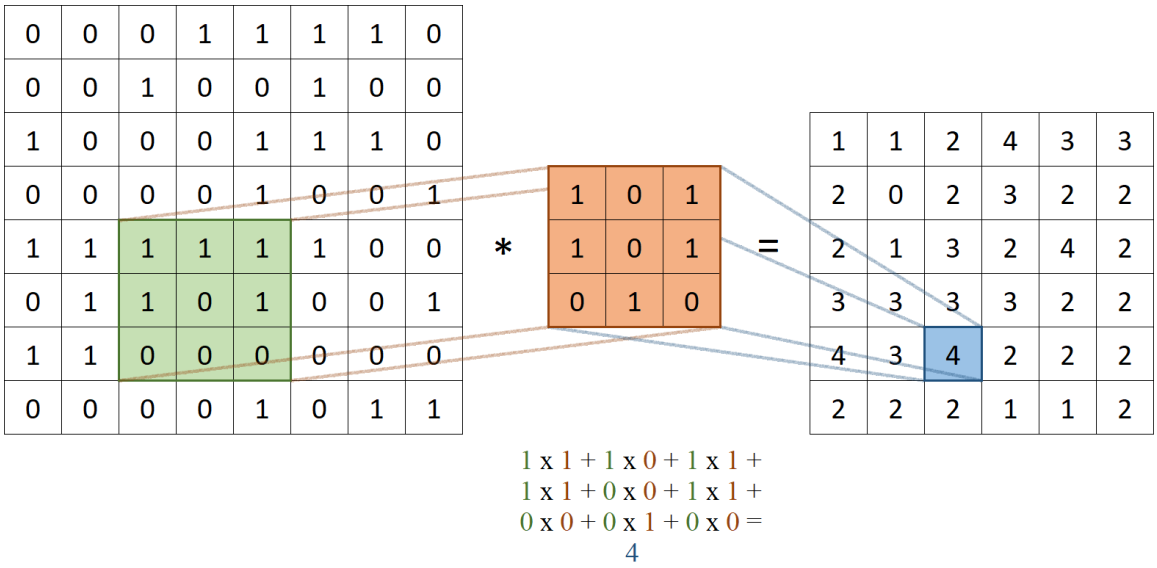

Figura 2.2: Exemplo de uma convolução. À esquerda, a matriz de entrada para a camada de convolução; no meio, o filtro; e, à direita, o *feature map* formado. Neste exemplo, a entrada possui apenas um canal de cor, sendo, na prática, uma matriz de duas dimensões.

Todos os filtros de uma camada de convolução têm as mesmas dimensões e deslocam-se a mesma quantidade de pixels quando são deslizados para fazer o produto escalar e gerar um ponto do *feature map* correspondente. O número de filtros por camada, seus tamanhos, e a distância que se movem, também chamada de passo, são hiper parâmetros e, portanto, são definidos previamente pelo desenvolvedor da rede. Em seguida, aplica-se a **função de ativação**, em geral a ReLU, em cada elemento de cada *feature map*, resultando na saída da camada convolucional. É comum, mas não obrigatório utilizar *pooling* depois da função de ativação para reduzir o tamanho dos *feature maps* enquanto preserva as informações importantes, mas essa técnica não foi utilizada neste trabalho.

Por fim, a classificação da entrada é feita pela última camada *fully-connected*, que é a de saída, e avaliada como descrito na seção Redes Neurais.

É possível haver mais de uma camada de convolução assim como pode haver mais de uma camada *fully-connected* além da de saída, o que pode ser mais vantajoso, uma vez que aumentar a profundidade da rede, em muitos casos, melhora a predição [GBC16]. Entretanto, não só o custo de tempo e de espaço aumentam, como há um limite para o quão melhor será o desempenho da IA. A partir de um certo ponto, a melhora se torna ínfima em comparação com o tempo desprendido e, portanto, deixa de ser benéfico colocar mais camadas.

2.4 Processo de Decisão de Markov

Antes de falar sobre aprendizado por reforço, é necessário explicar o que é um **Processo de Decisão de Markov** (*Markov Decision Process* - MDP). Um MDP padrão possui as seguintes propriedades: a probabilidade de se chegar em um estado futuro S' dado que a inteligência artificial, também conhecida como agente, se encontra no estado S depende apenas da ação A tomada nesse estado S, o que caracteriza a **propriedade Markoviana**; existe um modelo probabilístico que caracteriza essa transição, dado por $P(S'|S, A)$; todos os estados do ambiente e todas as ações que o agente pode tomar em cada estado são conhecidas; e a recompensa é imediatamente recebida após cada ação ser tomada.

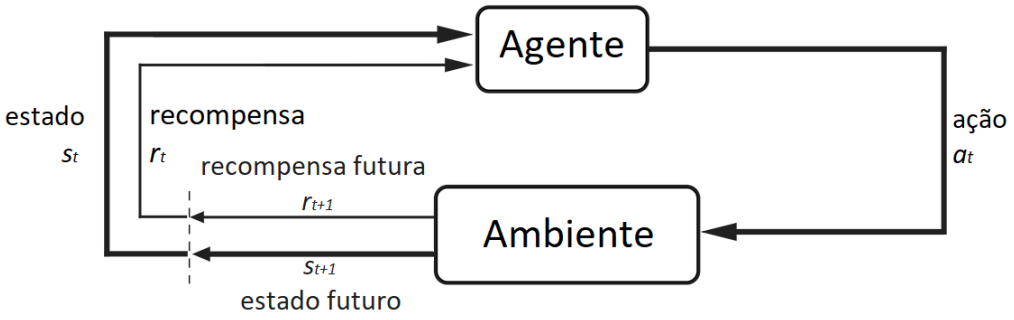

Figura 2.3: Interação agente-ambiente em um processo de decisão de Markov[SB18]. Adaptação e tradução da imagem original feitas pelo autor.

As probabilidades de o agente tomar cada ação em um dado espaço são definidas por uma política π. A qualidade de uma política é medida por sua **utilidade esperada**, e a política ótima é denotada por π^*. Para calcular π^*, utiliza-se um algoritmo de iteração de valor que computa a utilidade esperada do estado atual: começando a partir de um estado arbitrário S, tal que seu valor esperado é $V(S)$, aplica-se a equação de Bellman até haver convergência de $V(S)$, denotado por $V^*(S)$, que é usado para calcular a política ótima $\pi^*(s)$.

Seja i a iteração atual, S o estado atual, S' o estado futuro, A a ação tomada no estado atual, $R(S, A, S')$ a recompensa pela transição do estado S para o estado S' por tomar a ação A, e γ o valor de desconto (valor entre 0 e 1 que determina a importância de recompensas futuras para o agente), temos que:

Equação de Bellman:

$$V^{(i)}(S) = \max_A \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^{(i-1)}(S')] \tag{2.3}$$

$$\lim_{i\to\infty} V^{(i)}(S) = V^*(S) \tag{2.4}$$

Política gulosa para função valor ótima:

$$\pi^*(s) = \operatorname*{argmax}_A \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^*(S')] \tag{2.5}$$

Um processo de decisão de Markov cuja função de probabilidade de transição $P(S'|S, A)$ ou função de recompensa $R(S, A, S')$ é desconhecida se torna um problema de **aprendizado por reforço**.

2.5 Aprendizado por reforço

Aprendizado por reforço, diferente do supervisionado e de redes neurais por consequência, não recebe exemplos rotulados para saber o quão correta ou incorreta sua resposta está para cada entrada. Ao invés disso, o agente interage com o ambiente e recebe recompensas positivas, negativas ou nulas por suas ações. Seu objetivo é explorar o espaço de estados a fim de aprender a recompensa esperada para cada ação tomada em cada um deles. Dessa forma, ele saberá o que fazer em cada situação do ambiente em que se encontra.

As recompensas esperadas de cada ação em cada estado são armazenadas em uma tabela que deve mapear todas as ações para todos os estados. Isso é possível em domínios simples, como um Jogo da Velha, mas se torna impraticável conforme o espaço de estados aumenta. No caso do *Pong* e do *Asteroids*, os *frames* do jogo são os estados. Um pixel que mude de cor já faz ser um estado completamente diferente do ponto de vista do computador. Em uma tela de 210x160 pixels, com cada pixel armazenando três números que vão de 0 à 255 para determinar sua cor, é evidente não ser possível armazenar na memória um mapeamento das ações para cada um desses estados. Mesmo que não houvesse esse obstáculo computacional, existem situações em que não é possível determinar qual ação retornará a maior recompensa, como na figura 2.4.

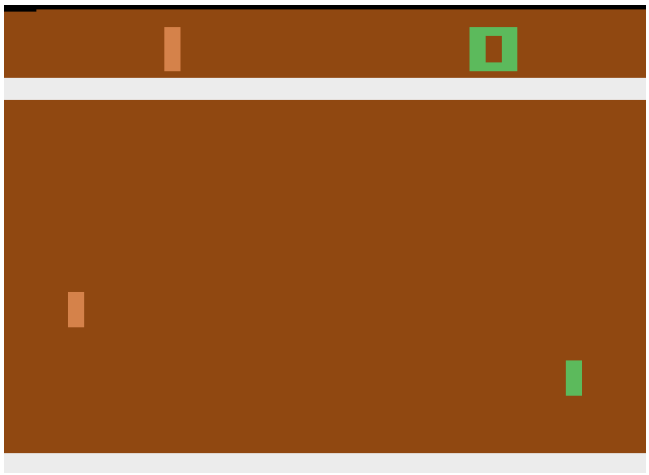

Figura 2.4: Quando a bola no *Pong* fica fora da tela, não é possível dizer qual a melhor ação a ser tomada por falta de informação ou porque todas terão o mesmo resultado. No *Pong*, não é possível ver a bola somente quando algum dos jogadores marca um ponto, pois ela saiu da tela pelo lado esquerdo ou direito; quando isso ocorre, todas as ações tomadas têm o mesmo resultado, pois é preciso esperar alguns instantes para uma nova bola aparecer e o jogo continuar.

Como dito no final da seção anterior, aprendizado por reforço é um MDP que não utiliza as probabilidades de transição ou a função de recompensa para aproximar a política ótima. No contexto deste trabalho, a política ótima será encontrada utilizando uma variante de ***Q-Learning***, uma técnica de aprendizado por reforço.

2.6 *Q-Learning*

Quando não se conhece as probabilidades de transição ou a função de recompensa, informações necessárias para se obter a função valor pela equação de Bellman, é possível estimar $V(S)$ a partir de observações feitas sobre o ambiente. Logo, o problema passa a ser como extrair a política do agente de uma função valor estimada.

Seja $Q^*(S, A)$ a função Q-valor[3] que expressa a recompensa esperada de se começar no estado S, tomar a ação A e continuar de maneira ótima. $Q^*(S, A)$ é uma parte da política gulosa para função valor ótima e é dada por:

$$\begin{aligned} Q^*(S, A) &= \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^*(S')] \\ &= \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma \max_{A'} Q^*(S', A')] \end{aligned} \tag{2.6}$$

Logo, substituindo 2.6 em 2.5, temos que a política gulosa ótima para a função Q-valor ótima é dada por:

$$\pi^*(S) = \operatorname*{argmax}_A Q^*(S, A) \tag{2.7}$$

O próximo passo será entender como atualizar a função Q-valor. Supondo que o agente se encontra no estado S e toma a ação A, que causa uma transição no ambiente para o estado S' e gera uma recompensa $R(S, A, S')$, como computar $Q^{(i+1)}(S, A)$ baseado em $Q^{(i)}(S, A)$ e em $R(S, A, S')$, sendo i o instante atual? Para responder a essa pergunta, duas restrições precisam ser feitas: $Q^{(i+1)}(S, A)$ deve obedecer, pelo menos de forma aproximada, a equação de Bellman, e não deve ser muito diferente de $Q^{(i)}(S, A)$, dado que são médias de recompensas. A seguinte equação responde a essa questão.

Seja α a taxa de aprendizado (valor entre 0 e 1 que determina o quão importantes informações novas são em relação ao conhecimento que o agente possui),

$$\begin{aligned} Q^{(i+1)}(S, A) &= (1 - \alpha)Q^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma \max_{A'} Q^{(i)}(S', A')] \\ &= Q^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma \max_{A'} Q^{(i)}(S', A') - Q^{(i)}(S, A)] \end{aligned} \tag{2.8}$$

A convergência de $Q^{(i)}(S, A)$ em $Q^*(S, A)$ é garantida mesmo que o agente aja de forma subótima contanto que o ambiente seja um MDP, a taxa de aprendizado seja manipulada corretamente, e se a exploração não ignorar alguns estados e ações por completo - ou seja, raramente. Mesmo que as condições sejam satisfeitas, a convergência provavelmente será demasiadamente lenta. Entretanto, é interessante analisar os problemas levantados pela segunda e pela terceira condição que garantem a convergência e maneiras de solucioná-los.

Se a **taxa de aprendizado** for muito alta (próxima de 1), a atualização do aprendizado se torna instável. Por outro lado, se for muito baixa (próxima de 0), a convergência se torna lenta. Uma solução possível para essa questão é utilizar valores que mudam de acordo com o estado: mais baixos em estados que já foram visitados muitas vezes, pois o agente já terá uma boa noção da qualidade de cada ação possível, então há pouco que aprender; e mais altos em estados que foram visitados poucas vezes, pois o agente precisa aprender melhor sobre ele.

Uma vez que a política é gulosa em relação ao Q-valor, o agente sempre tomará a ação que retorna a maior recompensa esperada. Ou seja, a ação escolhida depende do valor da taxa de desconto γ: recompensas imediatas serão mais buscadas se for próximo de 0, enquanto recompensas futuras serão mais valorizadas para valores próximos de 1. Isso é bom somente se todas as recompensas possíveis para aquele estados são conhecidas. Porém, se houver ações não exploradas, o agente pode perder uma recompensa maior do que as que ele já conhece apenas porque ignorou a ação que leva a ela. Essa situação caracteriza o dilema ***Exploration versus Exploitation***: é melhor tomar a ação que retorna a maior recompensa

[3] O nome "Q-valor" vem do valor da qualidade da ação

ou buscar uma melhor? Da mesma forma que na taxa de aprendizado, uma forma de contornar esse problema é mudar a probabilidade de decidir explorar o ambiente (*explore*) de acordo com a situação. Conforme o mundo é descoberto, se torna cada vez mais interessante agir de forma gulosa (*exploit*) do que explorar em estados muito visitados, e vice-versa em estados pouco visitados. Esse comportamento pode ser definido por uma função de exploração.

Seja P_{ini} a probabilidade inicial e P_{min} a probabilidade mínima de o agente decidir explorar (*explore*) o ambiente, *decay* a taxa de decaimento e *step* o número de passos dados até o momento. A probabilidade de o agente explorar (*explore*) o ambiente é dada por:

$$P_{explore} = P_{min} + (P_{ini} - P_{min})e^{-step/decay} \tag{2.9}$$

Outro problema enfrentado por *Q-learning* é o de generalização. A política $\pi^*(S)$ determina a melhor ação a se tomar em cada estado. Logo, utiliza-se uma tabela para armazenar todas essas escolhas. Porém, como mencionado anteriormente, isso se torna inviável para espaços de estado muito grandes. Portanto, se não é possível aprender os Q-valores, o melhor que se pode fazer é tentar achar uma aproximação deles.

2.7 *Approximate Q-Learning*

Approximate Q-Learning é o nome dado a um conjunto de métodos de aprendizado por reforço que busca aproximar o Q-valor das ações. Uma técnica comum desse conjunto consiste no uso de **funções lineares** que avaliam características dos estados.

Para lidar com o enorme espaço de estados que alguns ambientes possuem, o agente armazena e aprende apenas algumas propriedades determinadas pelo desenvolvedor, que são funções de valor real, para tomar as decisões. Tais informações são armazenadas em um vetor e cada elemento desse vetor recebe um peso que determina a respectiva importância para que escolhas sejam feitas. Ou seja, a função Q-valor é representada por uma combinação linear das propriedades e é dada da seguinte forma.

Sejam $f_i(S, A)$ e ω_i, i = 1, ..., n, a i-ésima característica do ambientes detectada pelo agente e seu respectivo peso, ambos representados por números reais.

$$Q(S, A) = \omega_1 f_1(S, A) + \omega_2 f_2(S, A) + ... + \omega_n f_n(S, A) \tag{2.10}$$

Como o $V(S')$ é o valor esperado e $Q(S, A)$ é o valor previsto, a atualização pode ser interpretada como ajustar o Q-valor pela diferença desses dois números. Além disso, como o *approximate Q-learning* faz uma avaliação das características do estado em que o agente se encontra ($f_i(S, A)$) para tomar uma decisão, apenas os pesos (ω_i) precisam ser atualizados.

A atualização do k-ésimo peso, ω_k, $k = 1, ..., n$, ocorre conforme a equação 2.11 abaixo.

$$\omega_k^{(i+1)}(S, A) = \omega_k^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma V(S') - Q^{(i)}(S, A)]f_k(S, A), k = 1, 2, ..., n \tag{2.11}$$

Duas grandes vantagens de representar o Q-valor como uma combinação linear são evitar *overfitting*, e ser matematicamente conveniente, ter maneiras convenientes de calcular erro e funções que generalizem as decisões.

Percebe-se neste ponto uma semelhança bem grande com a forma como redes neurais profundas funcionam.

2.8 *Deep Q-Learning*

Agora que as técnicas de aprendizado profundo e de aprendizado por reforço foram apresentadas, falta falar do tipo de aprendizado obtido quando é feita a junção delas, que é o utilizado neste trabalho: ***Deep Q-Learning***. Por se tratar de um tipo de aprendizado por reforço, mais precisamente *approximate Q-learning*, a inteligência artificial também será referida como agente.

Dado que o agente deve aprender enxergando a tela e tudo que ele vê são matrizes, a primeira etapa consiste em processar essas informações para poder aprender. Ou seja, o primeiro passo é passar as imagens da tela por uma **rede neural convolucional**. Essa etapa funciona conforme descrito na seção sobre redes neurais convolucionais: as imagens são a entrada, ocorre o processamento nas camadas ocultas e, na camada de saída, cada neurônio tem um valor que determinam a ação tomada. Contudo, calcular o erro é diferente. Como as imagens passadas não são rotuladas, a IA precisa calcular o erro da saída da rede de outra forma, que será utilizando **aprendizado por reforço**. Conforme explicado anteriormente, aprendizado por reforço aprende sem o uso de exemplos rotulados, mas precisa que as características que a IA deve aprender estejam bem definidas. Enquanto tais características são definidas pela rede, o cálculo do erro para a otimização dos pesos é feito pela parte de aprendizado por reforço.

O **cálculo do erro** é feito por funções que retornam valores altos quando o modelo devolve respostas muito distantes da correta, e próximos de zero caso contrário, como o erro quadrático médio, *cross-entropy* e *Huber loss* [Hub64]. Para minimizar o erro, normalmente utiliza-se o módulo do número retornado pela função de erro, para garantir que será positivo, com alguma variação do **método do maior declive**, também conhecido como **método do gradiente** [Cau47] [4], como o RMSProp [GH14] e o Adam [KB14]. Em matemática, gradiente é uma generalização de derivada para múltiplas variáveis e, portanto, aponta para a direção de maior crescimento da função. O método do gradiente, por outro lado, utiliza o negativo do gradiente para apontar para a direção de maior decrescimento e, por consequência, para um mínimo local.

Seja $F(x)$ uma função de múltiplas variáveis, x_t um ponto no instante t e α um número real que multiplica o gradiente de $F(x)$. Em contexto de aprendizado de máquina, $F(x)$ é a função de erro, x_t é o vetor de pesos na t-ésima iteração e α é a taxa de aprendizado. A redução do erro é feita atualizando os pesos da seguinte forma:

$$x_{t+1} = x_t - \alpha \nabla F(x_t) \tag{2.12}$$

Intuitivamente, começa-se em um ponto x_0 qualquer e utiliza-se esse algoritmo para deslocar-se para um ponto vizinho x_1 que, se α for pequeno o suficiente, $F(x_0) \geq F(x_1)$. Se isso for feito iterativamente, tem-se $F(x_t) \geq F(x_{t+1})$ a cada passo. Importante notar que, se o α for muito alto, a redução do erro é rápida, porém instável ou pode sequer acontecer. Por outro lado, se for muito pequeno, a redução é precisa, mas lenta. A figura 2.5 ilustra essas situações.

2.9 Aprimorando o aprendizado

Deep Q-Learning é a base para o aprendizado da inteligência artificial deste trabalho, mas esse processo se torna muito demorado conforme o problema se torna mais complexo. Nesta seção, serão apresentadas duas técnicas que ajudam a estabilizar e acelerar o aprendizado do programa: ***experience replay*** [Lin92] e **alvo fixo** [VM15].

[4] *Gradient descent*, em inglês

Figura 2.5: Se α for muito alto, a diminuição se torna instável ou pode nem ocorrer (a esquerda); se for muito pequeno, se torna precisa, porém lenta (a direita). Curva representando função de erro desenhada com https://www.desmos.com/calculator

2.9.1 *Experience replay*

Estudado inicialmente como um método para acelerar a convergência em aprendizado por reforço, *experience replay* mostrou-se útil para *Deep Q-Learning* por ser uma técnica desse tipo de aprendizagem. Por conta da forma como *Deep Q-Learning* funciona, se a rede aprendesse com os *frames* conforme o agente os vê, a entrada seria composta por estados sequenciais altamente correlacionados. Isso significa que ela se atualizaria apenas com as experiências passadas mais recentes, esquecendo as mais antigas e sobrescrevedo os pesos obtidos no passado. Para contornar esse problema, utiliza-se um *buffer* de memória que armazena uma quantidade pré-determinada de experiências passadas.

Primeiro, o *buffer* de memória é preenchido com M tuplas t de formato $(S, A, R, S', done)$ antes do treinamento, onde S é o estado atual, A é a ação tomada no estado S, S' é o estado resultante por se tomar a ação A no estado S, R é a recompensa obtida pela transição do estado S para o estado S' como resultado da ação A, e *done* é uma variável que sinaliza se o agente chegou a um estado terminal. Depois, no treinamento, uma tupla é armazenada na memória a cada passo, enquanto um conjunto de tuplas desse mesmo formato é escolhido aleatoriamente do *buffer* e suprido à CNN para a atualização dos pesos. Dessa forma, o aprendizado ocorre sem o esquecimento de experiências passadas, pois elas são utilizadas para atualização dos pesos a todo momento.

2.9.2 Alvo fixo

Para o cálculo do erro da ação escolhida pela rede neural, uma comparação definida pela função de erro escolhida é feita entre o Q-valor de saída da rede com um Q-valor alvo calculado a parte. Sejam θ os parâmetros da rede neural, Q_θ o Q-valor de saída da rede neural e $\overline{Q}_\theta$ o Q-valor alvo. O Q-valor alvo $\overline{Q}_\theta$ e a função de erro são dados, respectivamente, por:

$$\overline{Q}_\theta(S, A) = R(S, A, S') + \gamma max_{A'} Q_\theta(S', A') \tag{2.13}$$

Onde $max_{A'} Q_\theta(S', A')$ é o maior Q-valor do estado S' calculado pela rede neural com os parâmetros θ.

Como os mesmos parâmetros da rede são utilizados para calcular o Q-valor de saída e o Q-valor alvo, tanto o Q_θ quanto o $\overline{Q}_\theta$ mudam. Portanto, a otimização dos pesos tenta aproximar a saída da rede de um alvo que está movimento, causanda uma grande oscilação no treinamento, desestabilizando-o. A solução proposta por Mnih et al.(2015) para amenizar esse problema foi utilizar uma segunda rede neural, inicialmente com os mesmos pesos da rede principal, que é atualizada periodicamente, copiando os parâmetros da principal, ao invés de a cada passo. Seja $\overline{\theta}$ os parâmetros da rede alvo, o cálculo do erro passa a ser feito utilizando a diferença entre $\overline{Q}_{\overline{\theta}}$ e Q_θ.

Com um alvo que se move pouco, a otimização se torna mais estável, melhorando o aprendizado.

Capítulo 3

Implementação

3.1 Ambientes

Antes de apresentar as arquiteturas das redes utilizadas e como os treinamentos foram feitos, serão formalizados os ambientes e os MDPs que os modelam.

3.1.1 *Gridworld*

Os estados S são o mapa em que o agente está inserido com o agente em uma das posições possíveis. Portanto, o número de estados existentes é igual ao número de espaços válidos. As ações possíveis A são mover-se para o espaço acima, abaixo, a direita ou a esquerda contanto que seja válido. É possível o agente decidir mover-se para um espaço inválido, mas isso faz apenas com que ele não saia do lugar. Os valores das recompensas $R(S, A)$ são definidas pelo desenvolvedor do ambiente. O objetivo é um estado terminal de recompensa positiva, a armadilha é um estado terminal de recompensa negativa, e estados não-terminais podem gerar recompensas não-nulas se o desenvolvedor quiser. As probabilidades de transição $P(S, A, S')$ são as probabilidades de o agente estar em um determinado espaço do mapa (estado S) e, a partir do movimento para algum dos espaços adjacentes (ação A), chegar a algum outro espaço do mapa (estado futuro S'). Não existe aleatoriedade no movimento do agente, pois ele sempre se desloca para a direção que escolheu, não sendo possível, por exemplo, mover-se para a direita quando a ação de mover-se para cima for escolhida.

Os experimentos foram feitos no *Frozen Lake* do Gym, pois ele funciona da mesma forma que o *Gridworld* conforme descrito acima, com a exceção de os movimentos não serem determinísticos, ou seja, é possível o agente escolher mover-se para um lado mas deslocar-se para outro. Para que ele funcionasse como o desejado, essa aleatoriedade foi removida. No caso deste trabalho, o mapa é de tamanho 10x10 com 8 armadilhas, 1 objetivo e nenhum espaço inválido, ou seja, há 100 espaços possíveis. O objetivo gera recompensa de valor 1000, as armadilhas de valor -200, e estados não terminais de -1. O mapa em que os experimentos foram feitos está representado pela imagem 3.1.

Gridworld é o ambiente mais simples estudado neste trabalho, pois possui poucas regras, estados possíveis, e ações válidas em cada um deles.

3.1.2 *Pong*

Os estados S são a tela do jogo, representada por matrizes de pixels de três dimensões: altura, largura, e canais de cor. As ações possíveis A são mover a barra que o jogador controla para cima ou para baixo. As recompensas $R(S, A)$ são recebidas quando a bola chega ao fim da tela do lado esquerdo ou direito, gerando

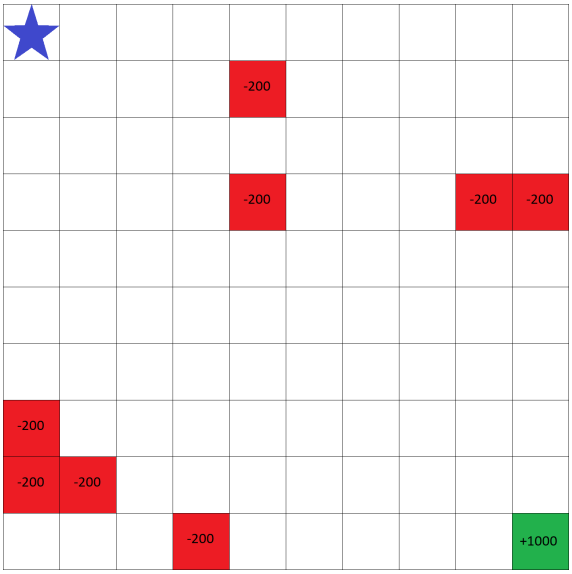

Figura 3.1: Mapa utilizado para os experimentos. A estrela azul indica a posição inicial do agente, o verde com +1000 representa o objetivo, e os vermelhos com -200 as armadilhas. Não há espaços inválidos.

uma positiva se chegar do lado do adversário, e negativa se chegar do lado do jogador. As probabilidades de transição $P(S, A, S')$ são as probabilidades de o jogo estar em um estado S, por exemplo com a bola sendo rebatida pelo jogador, e transitar para algum outro estado futuro S', como marcar um ponto, após tomar uma ação A, como mover a barra para cima. *Pong* é um jogo determinístico no sentido que não existe aleatoriedade na consequência das ações do jogador: se a bola colidir com a barra sempre no mesmo lugar, ela sempre será retornada na mesma direção com a mesma velocidade; marcar ponto sempre aumenta a pontuação do jogador em um. Por outro lado, existe um oponente contra o qual se está jogando e cujo comportamento configura um elemento de aleatoriedade no ambiente. Além disso, o jogador não conseguir rebater a bola como gostaria por falta de precisão também se aproxima de um elemento desse tipo.

Neste trabalho, o *Pong* foi emulado pelo Gym, que utiliza o emulador de Atari 2600, Stella [1]. Nessa versão, os estados são compostos por 210x160 pixels, com cada um podendo assumir 128 cores diferentes por combinações dos valores dos 3 canais de cor. Isso equivale a 33600 pixels e, portanto, 128^{33600} estados, com apenas uma parcela sendo realmente possível. A recompensa por marcar ponto é de +1 e a de o oponente marcar ponto é de -1. Como o jogo termina quando um dos lados alcança 21 pontos, a pontuação final varia de -21 até +21 inclusos.

Pong é o ambiente de complexidade intermediária estudada neste trabalho, pois possui poucas regras, grande espaço de estados, e poucas ações possíveis em cada um deles.

3.1.3 *Asteroids*

Os estados S são a tela do jogo, representada por matrizes de pixels de três dimensões: altura, largura, e canais de cor. As ações possíveis A são mover-se para frente, girar no sentido horário, girar no sentido anti-horário, mover-se no hiper-espaço (se teletransportar para algum lugar aleatório da tela), e atirar para frente. As recompensas $R(S, A)$ são recebidas por destruir os asteróides, com o maior valendo menos pontos e o menor valendo mais pontos, podendo ser obtidas tanto atirando neles quanto colidindo, não havendo recompensa negativa pela perda de vida. As probabilidades de transição $P(S, A, S')$ são as probabilidades de o jogo estar em um estado S, por exemplo o inicial em que o jogador tem zero pontos

[1] https://stella-emu.github.io/

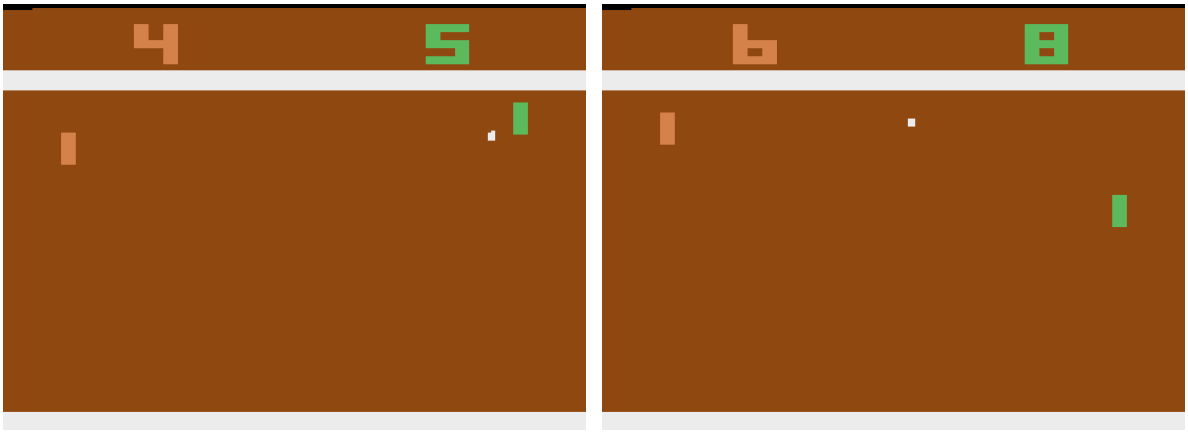

Figura 3.2: Exemplos de tela do jogo no emulador de Atari 2600, Stella. Nestas imagens, o jogador controla a barra verde, do lado direito da tela, enquanto a IA criada pelos desenvolvedores do jogo controla a barra laranja, do lado esquerdo da tela. O ponto branco entre elas é a bola, as barras brancas horizontais que estendem desde o lado equerdo até o lado direito da tela representam os limites superior e inferior do campo, e os números no topo indicam a pontuação de cada jogador.

e todas as vidas, e transitar para algum outro estado futuro S', como destruir algum asteroide e receber pontos por isso, após tomar uma ação A, como atirar para frente. Assim como em *Pong*, *Asteroids* é um jogo determinístico no sentido que não existe aleatoriedade na consequência das ações do jogador: se ele fizer um disparo, o tiro seguirá reto durante um certo tempo até desaparecer ou atingir um asteroide; cada tamanho de asteroide sempre aumenta a pontuação do jogador pela mesma quantidade quando destruído. Os fatores mais próximos de aleatoriedade existentes no jogo são o jogador ignorar, desconhecer, abstrair e/ou não perceber partes do jogo, como a posição dos asteróides.

Neste trabalho, o *Asteroids* foi emulado pelo Gym-Retro, que utiliza o emulador de Atari 2600, Stella. Como esta versão também é a do Atari 2600, o número de estados possíveis é uma parcela de 128^{33600} assim como no *Pong*. A recompensa por destruir um asteróide grande é de 20 pontos, de um médio é de 50 pontos, e de um pequeno é de 100 pontos. O jogador começa com 4 vidas e recebe uma nova a cada 5000 pontos obtidos. O único estado terminal nativo do jogo ocorre quando todas as vidas são perdidas.

Asteroids é o mais complexo dos ambientes estudados neste trabalho, pois possui mais regras que são mais complexas, o maior espaço de estados dentre os três ambientes, e mais ações possíveis.

3.2 Arquiteturas das redes

Com os ambientes e MDPs formalizado, nesta seção serão descritas as arquiteturas utilizadas nos três ambientes abordados neste trabalho e, em seguida, os respectivos treinamentos feitos. Todo o código foi escrito em Python3, com as redes neurais sendo construídas utilizando o arcabouço TensorFlow.

3.2.1 *Gridworld*

Por ter poucos estados e ações possíveis em comparação com o *Pong* e o *Asteroids*, a arquitetura da rede do *Gridworld* foi muito simples. A rede neural convolucional utilizou uma camada de convolução seguida por uma *fully-connected*, a de saída. A convolucional tinha 8 filtros de tamanho 2x2, passo 1, função de ativação ReLU e inicializador de Xavier [GB10], enquanto a *fully-connected* tinha um nó de saída para cada ação possível, ou seja, um vetor de tamanho 4, sem função de ativação e inicializada com zeros. O cálculo de erro foi feito pela função *Huber loss* [Hub64] e a otimização pela função *Root Mean Square Propagation*, mais conhecida como RMSProp [GH14]. A taxa de aprendizado α foi igual a 0.05; e

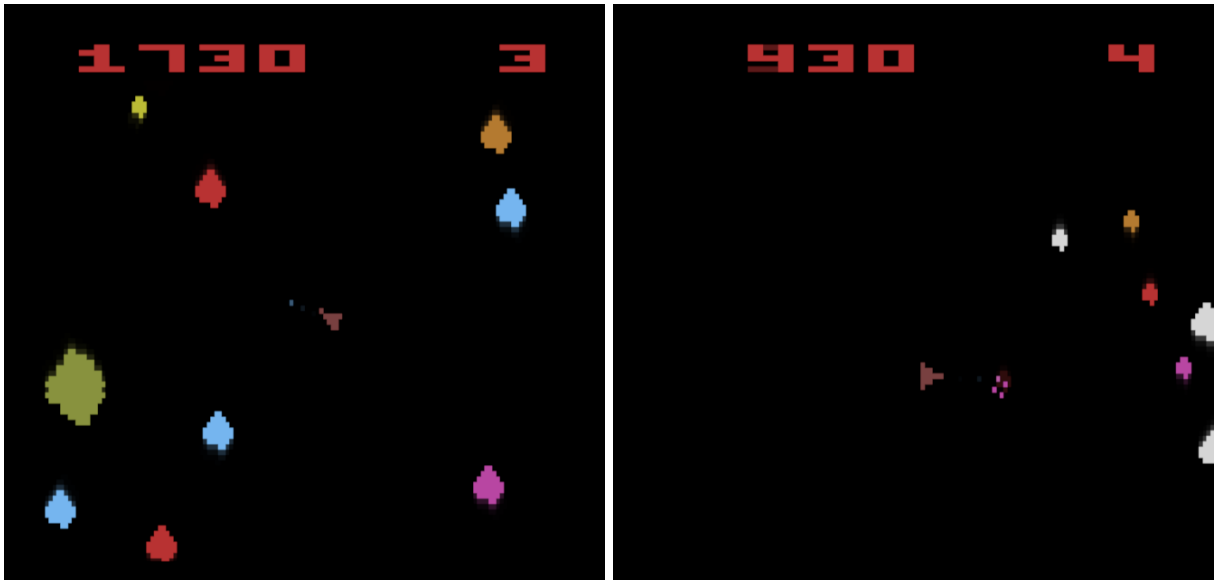

Figura 3.3: Exemplos de tela do jogo no emulador de Atari 2600, Stella. O número no canto superior esquerdo é a pontuação e o no canto superior direito é a quantidade de vidas restante que o jogador tem. A nave é o triângulo no meio da tela, o ponto azul próximo dela na imagem da esquerda é um tiro e o restante é asteroide. Na imagem da direita, os quatro pequenos pontos rosas próximos da nave são um asteroide que acaba de ser destruído.

o *momentum*, variável que indica o quanto gradientes anteriores devem ser considerados para determinar a direção do movimento, foi igual a 0.1.

3.2.2 *Pong*

A arquitetura da rede do *Pong* precisou ser mais elaborada para que o agente pudesse obter sucesso. Foram utilizadas três camadas de convolução e duas camadas *fully-connected*, sendo a segunda delas a de saída. A primeira convolucional tinha 32 filtros de tamanho 8x8 e passo 4; a segunda tinha 64 filtros de tamanho 4x4 e passo 2; e a terceira tinha 64 filtros de tamanho 3x3 e passo 1. A primeira *fully-connected* tinha vetor de saída de tamanho 256; e a segunda tinha um nó de saída para cada ação possível, ou seja, um vetor de tamanho 2. Todas as camadas utilizaram função de ativação ReLU, com exceção da segunda *fully-connected* que não usou nenhuma, e inicializador de He [2] [HZRS15] para os pesos. O cálculo de erro foi feito pela função *Huber loss* e a otimização dos pesos pelo RMSProp. A taxa de aprendizado α foi igual a 0.00025; o *momentum* foi igual a 0.1; e ϵ, variável que impede a divisão por zero no cálculo feito pelo RMSProp, igual a 0.1.

3.2.3 *Asteroids*

O *Asteroids* foi testado com diversas arquiteturas diferentes, mas sem sucessos. Tentou-se utilizar diferentes números de camadas ocultas, número de filtros, tamanhos e passos, funções inicializadoras e ativadoras, unidades de saída nas camadas *fully-connected*, funções de erro, de otimização e de exploração, taxas de aprendizado, de desconto e de atualização da rede alvo, assim como outras configurações externas à rede neural, como tamanho dos *mini-batches*, do *buffer* de memória e número de episódios de treinamento. Por conta da grande gama de hiper-parâmetros a serem ajustados e o tempo consumido para treinar, os testes com este ambiente foram os mais longos.

Para poder demonstrar um exemplo de resultados de treinamento para o capítulo Resultados, foram registrados os resultados de um treinamento cuja rede neural tinha três camadas de convolução e duas

[2] No TensorFlow, esse inicializador é chamado pela função `variance_scaling_initializer()`

fully-connected, sendo a segunda delas a de saída. A primeira convolucional tinha 48 filtros de tamanho 8x8 e passo 4; a segunda tinha 96 filtros de tamanho 4x4 e passo 2; e a terceira tinha 96 filtros de tamanho 3x3 e passo 1. A primeira *fully-connected* tinha vetor de saída de tamaho 512; e a segunda tinha um nó de saída para cada ação possível, ou seja, um vetor de tamanho 5. Assim como na rede do *Pong*, foi utilizada função de ativação ReLU em todas as camadas com exceção da segunda *fully-connected*, e inicializador de He para os pesos. O cálculo de erro foi feito pela função *Huber loss* e a otimização dos pesos pelo RMSProp. A taxa de aprendizado α foi igual a 0.00025; o *momentum* foi igual a 0.95; e ϵ igual a 0.1.

3.3 Experimentos

Os experimentos compõe a maior parte do trabalho por poderem levar alguns minutos, no caso do *Gridworld*, ou várias horas, podendo passar de um dia para o outro, como no caso do *Pong* e do *Asteroids*.

3.3.1 *Gridworld*

Por ser um ambiente com um número baixo de estados e ações possíveis, a arquitetura da rede neural pôde ser simples, com poucas camadas convolucionais e *fully-connected*, com poucos nós cada uma. Como resultado dessa baixa complexidade, foi possível realizar milhares de episódios de treinamento em poucas horas. Além disso, foi mais fácil analisar o aprendizado por ter poucos estados bem definido e por haver soluções evidentes de como chegar ao objetivo. A análise do sucesso do agente foi feita pela sua capacidade de conseguir chegar à recompensa positiva do mapa. O número de vezes que ele conseguiu chegar ao objetivo ao longo dos episódios de treinamento foi considerado como menos importante para avaliar o desempenho do aprendizado, pois há uma alta probabilidade de se tomar uma ação aleatória, de 40%.

O mapa utilizado para os experimentos tinha tamanho de 10x10, oito armadilhas espalhadas, o agente começava no canto superior esquerdo e o objetivo encontrava-se no canto inferior direito.

Foram 2000 episódios de treinamento, cada um com limite de 200 ações tomadas antes de o episódio ser terminado automaticamente; *mini-batches* de tamanho 200; taxa de desconto γ igual a 0.9; *buffer* de memória de tamanho 200 preenchido previamente com 200 ações tomadas aleatoriamente; e atualização da rede alvo feita a cada 200 passos. O dilema *exploration versus exploitation* utilizou $P_{ini} = 0.9$, $P_{min} = 0.4$ e $decay = 200$.

3.3.2 *Pong*

No caso do *Pong*, os treinamentos foram mais demorados, com episódios levando alguns minutos e passando várias horas para se perceber alguma melhoria no aprendizado uma vez que o espaço de estados é muito maior. Além disso, existem diversos momentos em que não há uma ação ótima bem definida a se tomar, como nos momentos em que a bola está se deslocando na direção do lado do adversário ou quando ela sai da tela, marcando ponto para um dos jogadores. Como os episódios só terminam quando um dos lados consegue 21 pontos (ou o número máximo de passos é excedido, o que não aconteceu neste trabalho), a avaliação foi feita pela pontuação obtida pelo agente ao final de cada episódio e da partida realizada após o treinamento, em que foram tomadas apenas ações ótimas segundo o modelo construído: o mínimo possível é de -21 pontos, com o adversário marcando 21 pontos e o agente nenhum, e o máximo é de 21 pontos, sendo a situação oposta.

Como este ambiente é muito mais complexo que o *Gridworld*, os *frames* da tela do jogo passam por uma etapa de pré-processamento para reduzir o tempo de processamento pela rede neural. Primeiro, eles são convertidos para escalas de cinza. Em seguida, partes que não agregam informação para a IA conseguir

jogar, como a área da pontuação, são removidas. Por fim, o que restou do *frame* é redimensionado para o tamanho 84x84 pixels. Para que o agente possa perceber o movimento dos objetos na tela, os últimos quatro *frames* vistos são inseridos em uma fila e supridos à rede de forma que a entrada tem formato 84x84x4.

O agente foi treinado por 298 episódios, o que equivale a aproximadamente um milhão de *frames*, cada um com limite de 18000 ações antes de o episódio ser terminado automaticamente; *mini-batches* de tamanho 32; taxa de desconto γ igual a 0.99; *buffer* de memória de tamanho 1000000 preenchido previamente com 50000 ações tomadas aleatoriamente; e atualização da rede alvo feita a cada 10000 ações. O dilema *exploration versus exploitation* utilizou $P_{ini} = 1.0$, $P_{min} = 0.1$ e $decay = 20000$.

Para acelerar o treinamento, foi utilizada uma técnica de pular *frames* [BNVB12] para economizar tempo. Ela consiste em o agente escolher uma ação a cada 4 *frames* e repeti-la nos três seguintes, ao invés de calcular qual a melhor em cada um deles, uma vez que é mais rápido avançar o emulador em um *frame* do que o agente selecionar uma ação. Esse comportamento é próprio do ambiente [3] utilizado, não havendo partes do código destinada a isso.

3.3.3 *Asteroids*

Por fim, *Asteroids* foi o ambiente em que mais experimentos foram feitos por conta do maior número de alterações feitas nos hiper-parâmetros e nas etapas do aprendizado. Assim como no *Pong*, os treinamentos levaram várias horas, chegando a passar de um dia para o outro em algumas ocasiões. O espaço de estados é mais complexo também, com mais informações na tela em cada instante e mais ações disponíveis. A análise de sucesso foi feita de forma semelhante ao *Pong*: pela pontuação obtida ao longo do treinamento e pelo modelo construído no final. Neste ambiente, a pontuação poderia assumir qualquer valor maior ou igual a zero dentro das restrições de tempo que cada episódio tinha. Uma vez que perder vida não gera recompensa negativa e colisão com asteróides gera pontos por sua destruição, a pontuação mínima que o agente poderia obter, sem exceder o número de passos definido, é de 80 pontos, que corresponde a destruição de quatro asteróides grandes por colisão com a nave.

Assim como no *Pong*, houve uma etapa de pré-processamento dos *frames* da tela do jogo que consistiu em convertê-los para escalas de cinza, remover partes sem informações relevantes da tela, redimensionar para um tamanho menor, e enfileirair para que o agente percebesse movimento, resultando em uma entrada de formato 84x84x4 para a rede neural.

O agente foi treinado por 100 episódios, o que corresponde a aproximadamente um milhão de *frames*, assim como no *Pong*, cada um com limite de 100000 ações tomadas por episódio antes de ser terminado automaticamente; *mini-batches* de tamanho 64; taxa de desconto γ igual a 0.99; *buffer* de memória de tamanho 1000000 preenchido previamente com 500 ações tomadas aleatoriamente; e atualização da rede alvo feita a cada 10000 ações. O dilema *exploration versus exploitation* utilizou $P_{ini} = 1.0$, $P_{min} = 0.1$ e $decay = 20000$.

Assim como no *Pong*, a técnica de pular *frames* foi utilizada, mas a escolha de ação foi feita a cada 3 *frames* ao invés de 4, pois a nave e os tiros aparecem em *frames* intercalados com os dos asteroides. Fazer a decisão de qual ação tomar a cada 3 *frames* permite que tanto a nave e os tiros quanto os asteroides sejam vistos pelo agente. Não existe um ambiente de *Asteroids* com esse comportamento no Gym-Retro, então essa técnica foi implementado no código.

[3] PongDeterministic-v4 do Gym

3.3.4 Pseudocódigo

Apesar de cada ambiente ter suas particularidades nos treinamentos, como número de episódio e tamanho dos *mini-batches*, todos foram treinados seguindo o mesmo formato. As linhas marcadas com um asterisco não são executadas no *Gridworld*, uma vez que as entradas para a rede neste ambiente não necessitam de pré-processamento. As linhas marcadas com dois asteriscos, são executadas apenas pelo *Asteroids*, que compõe a técnica de pular *frames* descrita anteriormente. O pseudocódigo do treinamento encontra-se a seguir.

Deep Q-Learning com *Experience Replay* e alvo fixo

inicializa DQN e TN com pesos inicializados pelo inicializador de He;
inicializa o *buffer* de memória B;
inicia o ambiente e recebe estado inicial S_0;

para $n = 0$ até N **faça**
    executa uma ação aleatória A_n;
    recebe a recompensa R_n, estado seguinte S_{n+1} e variável *done*;
    adiciona tupla (S_n, A_n, R_n, S_{n+1}, *done*) a B;
    se for estado terminal **então**
        reinicia o ambiente e recebe estado inicial em S_{n+1};
    senão
        $S_n \leftarrow S_{n+1}$;

para m $= 0$ até M **faça**
    reinicia o ambiente e recebe estado inicial S_0
    enquanto número máximo de passos não for excedido **faça**
        executa uma ação A_{passo} aleatoriamente com probabilidade ϵ;
        caso contrário, executa a melhor ação A_{passo} segundo a rede neural;
        recebe a recompensa R_{passo}, estado seguinte $S_{passo+1}$ e variável *done*;
        adiciona tupla (S_{passo}, A_{passo}, R_{passo}, $S_{passo+1}$, *done*) a B;
        se for estado terminal **então**
            reinicia o ambiente e recebe estado inicial em $S_{passo+1}$;
        senão
            $S_{passo} \leftarrow S_{passo+1}$;
        seleciona um *mini-batch* do *buffer* de memória;
        calcula o valor alvo Q_T;
        calcula o erro do *mini-batch*;
        otimiza os pesos e atualiza a rede neural;
        se τ passos tiverem passados desde a última atualização de TN **então**
            atualiza os pesos de TN;

Capítulo 4

Resultados

Os resultados foram parcialmente como o esperado. Neste capítulo, serão descritos os resultados e porque estiveram ou não dentro das expectativas. Por conta da natureza de cada ambiente e de seus resultados, formas diferentes de avaliação foram aplicadas.

4.1 *Gridworld*

O agente se saiu bem no ***Gridworld*** conforme as expectativas, consistentemente encontrando um caminho para o objetivo com diferentes arquiteturas que não fossem drasticamente diferentes, mesmo que elas não tenham sido ótimas. A tabela abaixo apresenta a proporção média de vezes que o agente treinado por *Deep Q-Learning* e que um aleatório chegaram ao objetivo. Para o agente treinado, o número foi obtido após ele passar por cinco treinamentos de 2000 episódios cada no mapa de tamanho 10x10 apresentado anteriormente. O modelo final conseguiu fazer o agente chegar ao objetivo quatro vezes e exceder o número máximo de ações uma vez, pois chegou a uma célula da qual não conseguiu sair já que as melhores ações tomadas eram ir em direção às paredes adjacentes. Para o agente aleatório, o número foi obtido após 10000 tentativas de se chegar ao objetivo, equivalente ao número de episódios que o agente treinado teve para chegar ao objetivo.

Jogador	% de chegadas ao objetivo
Aleatório	1.74%
DQL	66.15%

Alguns experimentos diferentes, mas menos aprofundados, com menos treinamentos indicaram que número e posicionamento de armadilhas, e tamanho da *grid* têm considerável influência no modelo construído. Por um lado, o agente encontrou caminhos até o objetivo com mais velocidade e consistência, uma vez que há menos estados a serem explorados e calculados. Por outro, a quantidade de armadilhas e locais em que são colocadas tiveram maior impacto no aprendizado; quando há muitas armadilhas ou ficam muito próximas do objetivo, tornou-se mais comum o agente não conseguir alcançá-lo, pois ele tenta evitar as recompensas negativas a ponto de ficar preso em *loops* entre alguns estados ou tentar mover-se contra uma parede. Esses problemas poderiam ser resolvidos com a escolha de hiper-parâmetros melhores.

4.2 *Pong*

Pong mostrou uma arquitetura bem mais sensível, com pequenas alterações nos hiper-parâmetros fazendo o aprendizado se tornar muito mais lento ou sequer acontecer. Mesmo assim, os resultados se

mostraram promissores dado tempo suficiente para o agente treinar e aprender.

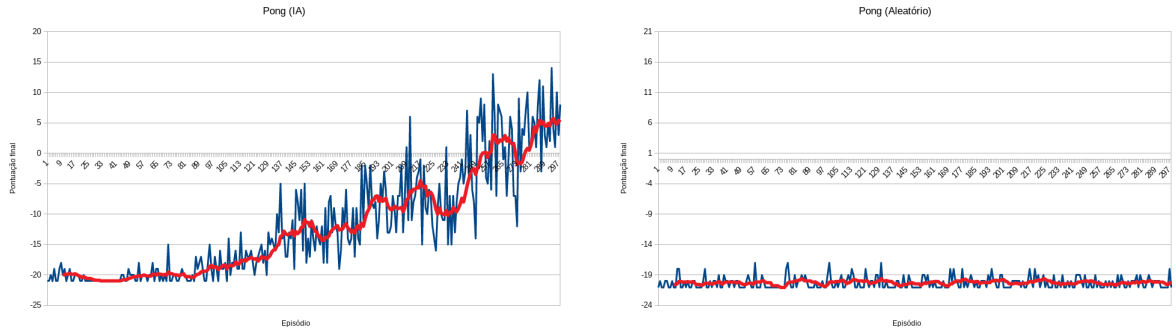

Figura 4.1: A imagem da esquerda mostra a pontuação ao longo do aprendizado e a da direita de um agente aleatório, ambos ao longo de 298 episódios, o que corresponde a aproximadamente um milhão de *frames*. Cada episódio corresponde a uma partida. A linha azul é a pontuação por episódio enquanto a vermelha é a média dos últimos 10 episódios.

O crescimento lento, mas estável da pontuação total obtida por episódio reflete a capacidade do agente de aprender, ainda que com dificuldade, a se comportar neste ambiente de maneira positiva.

Nos experimentos, foram coletadas a pontuação final média obtida por uma pessoa experiente jogando com o mouse no emulador Stella, por um agente jogando aleatoriamente, e pela inteligência artificial treinada, cada um jogando cinco partidas contra a inteligência artificial padrão do jogo.

Jogador	Pontuação média
Humano	7
Aleatório	-20.27
DQL	21

4.3 *Asteroids*

O agente não obteve resultados positivos em ***Asteroids*** ao longo dos vários testes feitos. Apesar de ser um ambiente propício para o aprendizado por *Deep Q-Learning*, tendo todas as informações claras na tela, ações bem definidas e recebimento de recompensas simples e consistente, o agente teve grande dificuldade em conseguir aprender. Por conta dessas características, esperava-se que ele conseguisse aprender a se comportar nesse domínio, ainda que com dificuldade.

O gráfico abaixo foi obtido ao utilizar a arquitetura de rede e treinamento descritos no capítulo Implementação e exemplifica a pontuação obtida pelo agente nos treinamentos pelos quais passou, comparando com a pontuação obtida por um agente aleatório.

Percebe-se que o agente não conseguiu aprender ou sequer mostrar indícios de melhoria mesmo com um tempo de aprendizado (em *frames*) próximo do *Pong*. Nos experimentos, foram coletadas a pontuação final média de uma pessoa sem experiência, de um agente jogando aleatoriamente e da inteligência artificial treinada, cada um jogando cinco partidas. A pontuação de jogadores experientes não foi considerada por estar muito acima, passando com facilidade dos 30000 pontos, não sendo um bom parâmetro de comparação.

Jogador	Pontuação média
Humano	1943.3
Aleatório	1458.1
DQL	607.8

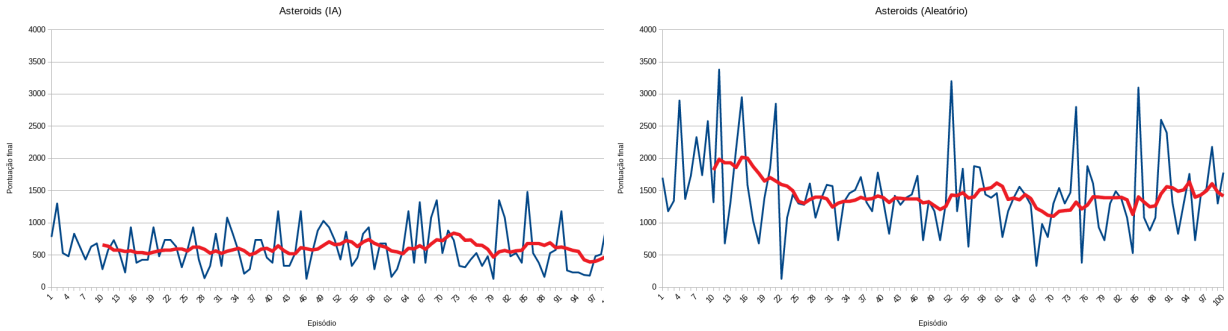

Figura 4.2: A imagem da esquerda mostra a pontuação ao longo do aprendizado e a da direita de um agente aleatório. A linha azul indica a pontuação final do agente nos episódios enquanto a vermelha indica a média da pontuação final dos 10 episódios anteriores.

Capítulo 5

Conclusão

Motivado pelo interesse em uma técnica de aprendizado de máquina não visto nas disciplinas de inteligência artificial da graduação, este trabalho buscou conhecer, estudar e explorar uma das formas utilizadas para ensinar um agente a se comportar em um domínio utilizando apenas imagens como entrada. Os ambientes de características e graus de complexidade distintos permitiram avaliar as capacidades e dificuldades que essa técnica apresenta, com resultados que refletiram as expectativas ainda que apenas em parte. Mesmo a falta de sucesso serviu como elemento de análise para este estudo.

O ***Gridworld*** apresentou sucesso consistente em encontrar um caminho até o objetivo. Isto estava dentro do esperado por conta da baixa dimensionalidade do problema: existem poucos estados, regras e ações disponíveis, e não há aleatoriedade. Há poucas informações para o agente aprender, podendo inclusive ser resolvido por técnicas mais simples sem grandes problemas.

O ***Pong*** já se mostrou mais complicado. Ainda que seu aprendizado tenha sido promissor para os hiper-parâmetros utilizados, ele foi lento e a arquitetura mostrou-se bem sensível, podendo não conseguir aprender por causa de pequenas alterações. Por possuir um espaço de estados bem maior que o *Gridworld*, mas bem menos situações diferentes e ações disponíveis que o *Asteroids*, esse resultado refletiu o grau de complexidade médio do ambiente neste trabalho: possível, mas precisa ser feito com cuidado. Um fato interessante é que, apesar de ainda haver espaço para o agente aprender, como é possível perceber no gráfico de pontuação final do agente, ele obteve 21 pontos em média contra a IA nativa do jogo quando o modelo foi colocado a prova, o que significa que não deixou a bola passar nenhuma vez.

Por fim, o ***Asteroids*** não apresentou resultados promissores. Sendo o ambiente mais complexo dos três, com mais regras e ações para se aprender e uma variedade maior de estados possíveis, o agente não conseguiu criar um modelo que conseguisse uma pontuação boa com as arquiteturas testadas. Existem diversas técnicas que aceleram o aprendizado de *Deep Q-Learning* e que poderiam ser aplicadas neste ambiente para tentar obter resultados positivos, como *Double Deep Q-Learning* [vHGS15], *Dueling Deep Q-Learning* [WdFL15] e *Rainbow* [HMvH+17]. Entretanto, isso seria uma garantia somente se um conjunto de hiper-parâmetros e funções boas fosse utilizado, o que já é um obstáculo por si só considerando a quantidade que existe para serem ajustados e o tempo consumido pelos treinamentos.

Deep Q-Learning apresentou resultados positivos em comparação com o que se esperava. Sua dificuldade de uso e tempo consumido são compensados pela capacidade de resolver problemas complexos sem que informações específicas do ambiente sejam utilizadas, como velocidade e direção de movimento de objetos da tela como da bola no *Pong* ou dos asteroides no *Asteroids*.

Referências Bibliográficas

[BNVB12] Marc G. Bellemare, Yavar Naddaf, Joel Veness e Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012. 18

[Cau47] Augustin Cauchy. Méthode générale pour la résolution de systèmes d'équations simultanées. *Compte rendu des séances de l'académie des sciences*, 1847. 11

[GB10] Xavier Glorot e Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. Em Yee Whye Teh e Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, páginas 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. 15

[GBC16] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. 6

[GH14] Kevin Swersky Geoffrey Hinton, Nitish Srivastava. Overview of mini-batch gradient descent, 2014. https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture$_s$*lides*$_l$*ec*6.*pdf*. 11, 15

[HMvH+17] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar e David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. 25

[Hub64] Peter J. Huber. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 35, 1964. 11, 15

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren e Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. 16

[KB14] Diederik P. Kingma e Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 11

[Lin92] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, May 1992. 11

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra e Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. 1

[SB18] Richard S. Sutton e Andrew G. Barto. *Reinforcement learning: an introduction*. MIT Press, 2018. 7

[vHGS15] Hado van Hasselt, Arthur Guez e David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. 25

[VM15] David Silver Andrei A. Rusu Joel Veness Marc G. Bellemare Alex Graves Martin Riedmiller Andreas K. Fidjeland Georg Ostrovski Stig Petersen Charles Beattie Amir Sadik Ioannis Antonoglou Helen King Dharshan Kumaran Daan Wierstra Shane Legg Demis Hassabis Volodymyr Mnih, Koray Kavukcuoglu. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 02 2015. 11

[WdFL15] Ziyu Wang, Nando de Freitas e Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. 25