Deep Q-Learning para ensinar inteligência artificial a jogar Asteroids

Vítor Kei Taira Tamada

Programa:
Bacharelado em Ciência da Computação

Orientador:
Prof. Dr. Denis Deratani Mauá

São Paulo, novembro de 2018

Deep Q-Learning para ensinar inteligência artificial a jogar Asteroids

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

Resumo

TAMADA, V. K. T. **Deep Q-Learning para ensinar inteligência artificial a jogar Asteroids**. Trabalho de Conclusão de Curso - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Jogos eletrônicos se tornaram comuns na vida de muitas pessoas nos últimos anos, seja em consoles de mesa tradicionais, em portáteis, ou em celulares. Eles normalmente são simples e intuitivos, para qualquer um poder começar a qualquer momento e aprender rapidamente como se joga. Porém, isso não é uma tarefa tão fácil para computadores. Utilizando *deep learning* em conjunto com aprendizado por reforço, o objetivo deste trabalho é produzir uma inteligência artificial que aprenda a jogar o jogo de Atari2600 *Asteroids*.

Palavras-chave: inteligência artificial, deep learning, aprendizado por reforço, asteroids

Sumário

Capítulo 1

Introdução

Inteligência artificial, ou IA, é uma área de estudos que pode ser definida de diversas formas, como construir uma máquina que realize com sucesso tarefas tradicionalmente feitas por humanos, ou que aja como um humano. Envolvendo filosofia, matemática, economia, neurociência, psicologia, computação e até mesmo linguística ao longo de sua história, ela abrangeu e ainda abrange diversos campos da ciência, com profissionais de várias formações diferentes podendo contribuir para seus avanços. Existem inúmeros desafios atualmente: alguns resolvidos, como vencer de jogadores profissionais de Xadrez, mas muitos ainda sendo abordados, podendo ser uma busca por alguma solução, ou por uma solução mais eficiente que a já existente.

Um tipo muito conhecido de inteligência artificial dos dias atuais é a que controla oponentes em jogos eletrônicos. Porém, nesses casos, os adversários apenas seguem um conjunto pré-determinado de regras escritas pelo desenvolvedor, não possuindo a capacidade de se adaptar como seres humanos fazem. Por mais que isso seja um tipo de IA e que possa ser mais eficiente em determinadas tarefas, não se assemelha à forma que as pessoas pensam e jogam. De forma geral, seres humanos aprendem interagindo com o ambiente: tocam nos objetos, tentam entender aquilo que os rodeia e qual o resultado de suas ações. Em um jogo, se não passar por um tutorial ou ler um manual, não será muito diferente: o jogador precisará descobrir o que é hostíl, o que cada comando faz e qual o objetivo.

Avanços recentes em inteligência artificial permitiram que máquinas simulem esse tipo de aprendizado por meio de **aprendizado por reforço**. Entretanto, para a maior parte dos jogos eletrônicos, só essa técnica não é o suficiente. Em poucos movimentos, uma pessoa já consegue supor o que é inimigo e o que é terreno quando aparece na tela do jogo. Para um computador, um pixel que mude de posição já faz ele não conseguir mais distinguir o que está vendo, tendo que re-aprender a cada nova combinação de pixels detectada. Em outras palavras, seres humanos conseguem abstrair as informações que enxergam com facilidade, enquanto os computadores não.

Se computadores 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 (*deep neural network*) chamada **rede neural convolucional** (*convolutional neural network* (CNN)), é possível fazer uma inteligência artificial abstrair essas informações e inferir que um objeto em diferentes lugares da tela, assumindo diferentes tamanhos, são o mesmo.

1.1 Motivação e Proposta

Unindo o antigo interesse por jogos com o recente por inteligência artificial, surgiu a proposta deste trabalho de conclusão de curso. Aplicando os conhecimentos adquiridos na faculdade, em particular de

computação e de aprendizado de máquina, o principal objetivo é criar uma inteligência artificial que aprenda a jogar o jogo eletrônico *Asteroids*.

As ferramentas utilizadas, descritas na próxima seção, serão o Gym-Retro como interface, o Stella como emulador e a API do TensorFlow para a computação. As técnicas de aprendizado de máquina utilizadas, descritas no capítulo seguinte, serão **aprendizado por reforço** e **rede neural convolucional**, mais especificamente a união das duas, conhecida como ***deep reinforcement learning*** ou ***deep Q-learning*** [Mni+13].

Espera-se conseguir construir uma arquitetura de aprendizado que permita a inteligência artificial desenvolver um modelo capaz de jogar com um desempenho pelo menos próximo de um ser humano. Em caso negativo, tentar explicar o motivo de o computador ter um desempenho notavelmente pior.

1.2 Ferramentas

Nesta seção, serão apresentadas as principais ferramentas utilizadas no desenvolvimento deste trabalho, uma breve descrição sobre elas e o motivo de suas escolhas.

1.2.1 *Asteroids* - Atari2600

Asteroids é um jogo de fliperama do gênero *top down shooter* (jogo eletrônico de tiro visto de cima) lançado em novembro de 1979 pela então desenvolvedora de jogos eletrônicos Atari Inc, atualmente conhecida como Atari. As principais diferenças entre as iterações de *Asteroids* incluem a presença de naves espaciais inimigas que atiram contra o jogador, formatos e tamanhos diferentes dos asteróides e direção que os asteróides se movem.

A versão de *Asteroids* utilizada neste trabalho é a do Atari2600, emulada pelo emulador Stella. Nesta iteração, não existem naves espaciais inimigas, apenas asteróides que assumem três tamanhos distintos, sendo o maior deles o inicial, e três formatos diferentes, mas de aproximadamente mesma altura e largura, e cores diferentes. Quando um asteróide grande (tamanho inicial) é destruído, outros dois de tamanho médio aparecem no lugar; após um asteróide de tamanho médio ser destruído, um de tamanho pequeno aparece em seu lugar. Destruir um asteróide grande gera uma recompensa de 20 pontos, destruir um médio gera uma recompensa de 50, e um pequeno gera uma de 100 pontos. A principal forma de destruir um asteróide e ganhar ponto é atirando neles, mas isso também ocorre quando há colisão entre a nave e um alvo. Isso reduz a quantidade de vidas disponíveis e, portanto, não é um método recomendado, dado que diminui a quantidade total de pontos ganha no final do jogo. Os asteróides também têm uma velocidade horizontal e vertical fixa para cada um. A cada *frame*, se movem 1 pixel na vertical e a cada aproximadamente 12 frames se movem um 1 pixel na horizontal, resultando em seus movimentos serem principalmente verticais.

O jogador possui cinco ações para jogar: mover-se para frente, girar a nave no sentido horário, girar a nave no sentido anti-horário, mover-se no hiper-espaço, e atirar para frente. Mover-se para frente e girar são as principais formas de movimento no jogo, enquanto atirar é a de destruir asteróides e ganhar pontos. Mover-se no hiper-espaço consiste em fazer a nave desaparecer por alguns instantes e reaparecer em um local aleatório da tela, podendo ser inclusive em cima de asteróides. Portanto, é um movimento arriscado, mas útil para fugir de situações complicadas. O jogador tem quatro vidas inicialmente.

A tela do jogo é uma matriz de tamanho 210x160 pixels com cada pixel tendo três números, que variam de 0 a 255 cada, e que determinam sua cor de acordo com a escala RGB, tendo acesso a uma paleta de 128 cores (é necessário que esses três números atinjam um certo valor para mudar a cor do pixel). No topo da tela, há dois números indicando a pontuação total até o momento e quantidade de vidas restantes.

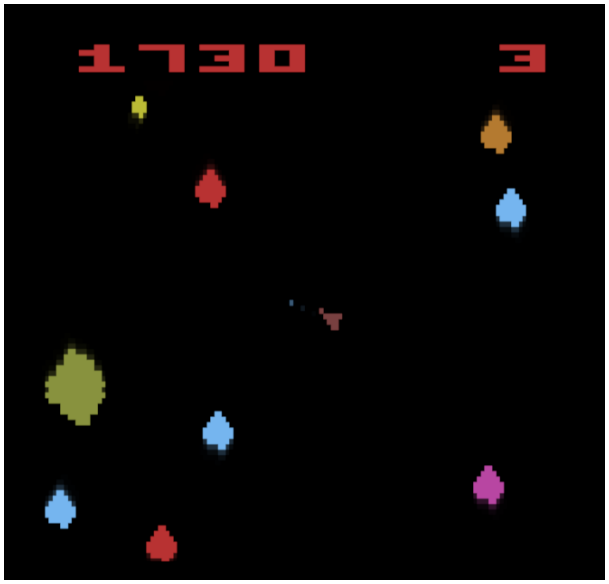

Figura 1.1: Exemplo de tela do jogo. O números no topo da tela são pontuação e quantidade de vidas respectivamente. A nave, no centro, está atirando. Asteróides espalhados pela tela.

Figura 1.2: Exemplo de tela do jogo. A nave acaba de destruir um asteróide de menor tamanho e está recebendo uma recompensa de 100 pontos por isso. Asteróides podem ter diversas cores.

1.2.2 Gym-Retro

Gym-Retro é uma plataforma para pesquisa de aprendizado por reforços e generalização em jogos desenvolvida e mantida pela empresa de pesquisas em inteligência artificial OpenAI. Essa ferramenta auxilia na emulação de diversos consoles de jogos eletrônicos, como Sega Genesis, Nintendo Entertainment System (NES) e Atari2600. Para qualquer jogo que o usuário deseje emular, é necessário que ele tenha a ROM (*Read Only Memory*) do jogo.

O principal motivo de esta ferramenta ter sido escolhida é o suporte ao jogo *Asteroids* e pela facilidade de seu uso.

1.2.3 TensorFlow

TensorFlow é um arcabouço de código aberto para computações numéricas de alta performance, desenvolvido e mantido pela Google. Seu núcleo de computação numérica flexível permite o uso da biblioteca em diversos campos cienctíficos. Oferece, em particular, grande suporte a aprendizado de máquina e aprendizado profundo, ou, como é mais conhecido, *deep learning*. Esta ferramenta foi escolhida por oferecer uma API em Python estável, ter grande suporte, comunidade ativa, e ser de código aberto.

1.3 Proposta

A proposta do trabalho é criar uma arquitetura para que uma inteligência artificial seja capaz de aprender a jogar *Asteroids* do Atari2600 tendo como entrada de dados apenas a tela do jogo, representada por uma matriz de pixels de 210x160 e 3 canais de cores. O emulador Stella e o Gym-Retro serão utilizados para emular o jogo e servir de interface com o jogo respectivamente. As técnicas de aprendizado de máquina utilizadas serão **aprendizado por reforço** e **rede neural convolucional**, em particular o ***deep Q-learning***, que é a junção dessas duas.

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 contas necessárias e, neste trabalho em particular, emulando o jogo que serve de ambiente para o aprendizado. Este capítulo tem o intuito de familiarizar o leitor com a teoria e fundamentos técnicas utilizados 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 foi baseada, 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, similar a pessoas, é considerada uma técnica de aprendizado supervisionado.

Com os avanços nos estudos dessa técnica nos últimos anos, diversos tipos diferentes de redes neurais foram desenvolvidos, como redes neurais convolucionais (*Convolutional Neural Networks*, CNN), a utilizada 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: os **neurônios**, e a estrutura dividida em **camadas**. 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 não serão discutidas neste trabalho.

Neurônios são funções que recebem como entrada a saída de cada neurônio da camada anterior, e devolvem um número, em geral entre 0 e 1 inclusos, cujo significado e como são usados variam de acordo com o trabalho em questão.

A estrutura de uma rede neural clássica é dividida em **camadas** que podem ser classificadas de três formas distintas: **entrada**, **oculta**, ou **saída**. A **entrada** é o que a IA recebe inicialmente e precisa processar; as camadas **ocultas** (*hidden layers*) são o processamento; e a **saída** é uma série de números utilizados pela IA para tomar uma decisão ou fazer uma predição. Pode-se dizer que uma rede neural é um aproximador de uma função que mapeia entrada e saída. Enquanto o número de neurônios na entrada e na saída são definidos pelo trabalho em questão, como número de pixels da tela e número de ações possíveis neste trabalho, o número de camadas ocultas e de neurônios em cada uma delas são arbitrários, sendo normalmente definidos por meio de tentativa e erro.

[1] Tradução livre feita pelo autor

Cada neurônio das camadas ocultas representa uma característica (*feature*) detectada ao longo do treinamento. Se essa característica estiver presente na camada de entrada, então o neurônio correspondente a essa característica será **ativado**. A ativação de um ou mais neurônios pode levar a ativação de outros neurônios na camada seguinte e assim sucessivamente. Esse é um comportamento inspirado na forma como neurônios do cérebro enviam sinais de um para o outro. Em redes neurais artificiais, um neurônio é ativado quando a soma dos números de entrada passa de um certo valor e por uma função de ativação.

Porém, nem todos os valores de entrada devem ser igualmente importantes, então cada um desses números recebe um peso que determina sua importância para a ativação da característica. Matematicamente, isso é representado da seguinte forma: seja n o número de neurônios na camada anterior, w_i, $i = 1, ..., n$, os pesos das saídas de cada neurônio da camada anterior, e a_i, $i = 1, ..., n$, o valor de saída de cada neurônio da camada anterior e, por consequência, cada valor de entrada do neurônio atual, e b o viés (*bias*) da função, que será explicado nos próximos parágrafos.

$$w_1a_1 + w_2a_2 + ... + w_na_n - b \tag{2.1}$$

Como essa soma pode ter qualquer valor no intervalo $(-\infty, +\infty)$, o neurônio precisa saber a partir de que ponto ele estará ativado. Para isso, utiliza-se uma **função de ativação**. Funções de ativação recebem a soma 2.1 como entrada, limitam seu valor a um certo intervalo e determinam se o neurônio deve ser ativado ou não.

Esse procedimento é feito em cada neurônio de cada camada da rede neural, o que pode ser muito custoso se não executado com cuidado. Como existem diversas bibliotecas que otimizam operações matriciais, é mais rápido e conveniente utilizar matrizes, além de facilitar a leitura do código: seja W a matriz tal que cada linha contém os pesos de cada neurônio da camada anterior para um determinado neurônio da camada atual, $a^{(i)}$ o vetor tal que cada elemento é o valor de saída de cada neurônio da camada anterior, e b o viés, é possível efetuar a soma 2.1 para todos os neurônios de uma camada da seguinte forma:

$$Wa^{(i)} + b, \qquad i = 1, ..., n \tag{2.2}$$

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 a inteligência conseguir soltar a resposta correta. Como mencionado anteriormente, conforme as características se mostram presentes na camada de entrada, os neurônios referentes a esses atributos são ativados, até que o neurônio com a resposta dada pela inteligência artificial seja ativado. Como rede neural é um tipo de aprendizado supervisionado, os exemplos inseridos nela possuem rótulos, saídas esperadas (qual valor que cada neurônio de saída deve ter). Para que o computador saiba o quão ruim foi sua saída, é definida uma função de erro, também conhecida como função de custo. Em muitos casos, utiliza-se o erro quadrático médio, que dá mais peso para erros maiores uma vez que a diferença é elevada ao quadrado, mas existem diversas funções diferentes para isso. Naturalmente, quanto maior for o erro, mais incorreta foi a previsão. Depois de esse procedimento ser feito com milhares de exemplos, calcula-se a média dos erros obtidos e, com isso, avalia-se o desempenho da inteligência.

Otimizar os pesos de forma que se reduz a média dos erros obtidos com os exemplos parece ser o melhor caminho para melhorar o modelo, mas isso não é necessariamente verdade. Os milhares de exemplos utilizados nessa etapa compõe o conjunto de treinamento. Se o modelo tiver erro zero em relação a esse conjunto, ele estará sofrendo de *overfitting*: a inteligência se adequa tanto ao conjunto

de treinamento que saberá o que fazer apenas nele. O mais desejável é minimizar o erro sobre todos os dados possíveis ou, no mínimo, os esperados em cenários reais, e o conjunto de treinamento dificilmente conseguirá abranger todos eles.

De forma resumida, uma rede neural clássica aprende recebendo uma série de números como entrada e devolve uma saída; calcula-se o quão errada essa saída está em relação ao desejado para aquela determinada entrada, e ajusta os pesos conforme a necessidade para minimizar o erro; após repetir esses passos milhares de vezes, espera-se que a IA tenha aprendido o suficiente a resolver o problema em mãos.

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 uma ou mais camadas ocultas. Se houver muitas camadas ocultas, a rede neural passa a ser chamada de rede neural profunda (*deep neural network*). Atualmente, não existe uma definição exata de quantas camadas a rede neural precisa ter para começar a 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**, técnica muito utilizada em reconhecimento de imagens. Como a ideia é treinar uma inteligência artificial que aprende vendo a tela do jogo, esse foi o tipo escohido para este trabalho.

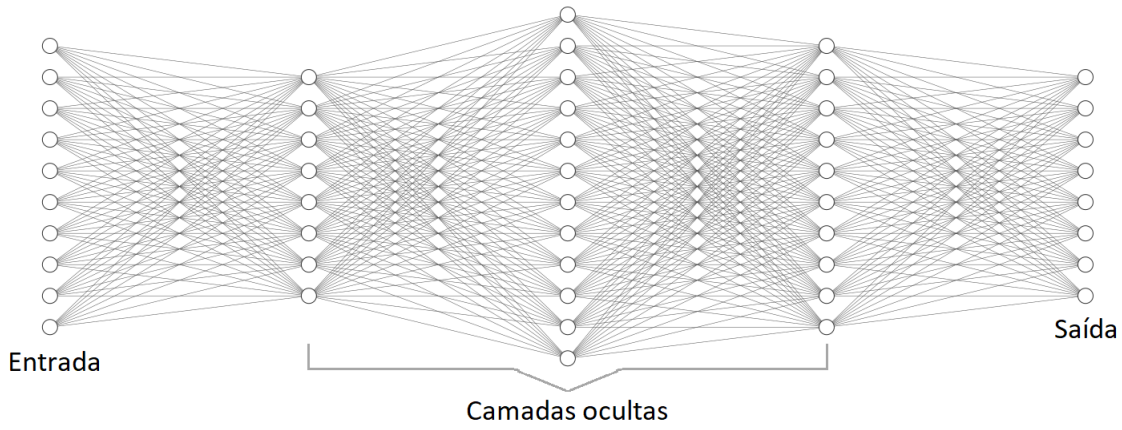

Figura 2.1: Esquema de uma rede neural profunda do tipo *feedforward*. Número de nós e camadas arbitrários para melhor representação. Diagrama feito em http://alexlenail.me/NN-SVG/index.html

2.3 Rede neural convolucional

Como neste trabalho a IA precisa aprender o que é um asteróide apenas enxergando a tela e interagindo com o ambiente, utilizar apenas redes neurais profundas sofre de um problema: o computador não consegue reconhecer um mesmo objeto em diferentes locais da tela e de diferentes tamanhos como o mesmo. Para cada local muito diferente que ele aparecer, como direita e esquerda da tela, a IA teria que re-aprender a identificá-lo.

Para não precisar fornecer à rede neural imagens inteiras para que ela aprenda que um objeto continua

sendo o mesmo não importa onde da tela apareça, foi utilizada convolução, mais precisamente a 2D, já que a entrada é uma imagem, uma matriz de pixels. Uma rede neural convolucional (*convolutional neural network* - CNN) continua sendo um tipo de rede neural profunda e, portanto, mantém o formato de três tipos de camadas (entrada, ocultas, e saída). Porém, para facilitar o entendimento de convolução, esta explicação dividirá a rede em duas partes: **convolução** e **previsão**. Na etapa de **convolucão**, a imagem de entrada é dividida em várias imagens menores; elas podem ser adjacentes ou parcialmente sobrepostas, sendo o segundo caso mais comum. Cada uma dessas imagens menores é chamada de **filtro convolucional**. É possível dizer que se um filtro convolucional for arrastado para o lado, o local onde ele parar será o próximo filtro convolucional. Esse arrasto é chamado de passo (*stride*) e a distância que o filtro é arrastado é chamada de tamanho do passo. Em seguida, cada uma dessas imagens menores é passada por uma rede neural menor, sendo processada normalmente. As saídas dessas redes neurais menores são então passadas como entrada para a próxima etapa. Na etapa de **previsão**, a informação passa por uma ou mais redes neurais maiores que farão a previsão. Para diferenciá-la das redes da etapa de convolução, as desta fase são chamadas de *fully-connected*.

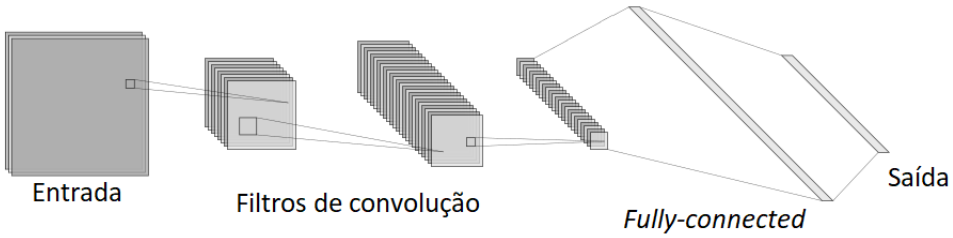

Figura 2.2: Esquema de uma rede neural convoluciona. Número de filtros e camadas arbitrários para melhor representação. Diagrama feito em http://alexlenail.me/NN-SVG/LeNet.html

É possível haver mais de uma camada de convolução assim como pode haver mais de uma camada *fully-connected*, e isso pode ser mais vantajoso. Quanto mais camadas houver, mais precisa será a predição. Entretanto, não só o custo de tempo e espaço aumenta, 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.

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 $V(S)$, que será denotado por $V^*(S)$. Esse $V^*(S)$ é usado para calcular a política ótima $\pi^*(s)$.

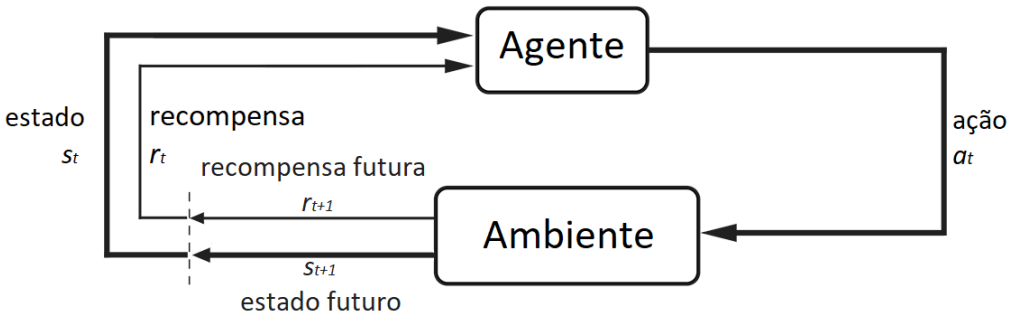

Figura 2.3: Interação agente-ambiente em um processo de decisão de Markov[SB18]. Adaptação e tradução da imagem original feitas pelo autor.

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 dado das fórmulas acima comum de se faltar é a probabilidade de transição $P(S'|S, A)$ e isso não é diferente no jogo *Asteroids*. Portanto, utiliza-se **aprendizado por reforço** para contornar esse problema.

2.5 Aprendizado por reforço

Aprendizado por reforço, diferente do supervisionado e, por consequência, de redes neurais, não recebe exemplos rotulados para saber o quão 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 ou um *Gridworld*, mas se torna impraticável conforme o espaço de estados aumenta. No caso do jogo *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, há muitas situações em que não é possível determinar qual ação retornará a maior recompensa.

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

2.6 *Q-learning*

Quando não se conhece as probabilidades de transição, informação necessária 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 deixa de ser tentar encontrar P e passa a ser como extrair a política do agente de uma função valor estimada.

Seja $Q^*(S, A)$ a função Q-valor[2] 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 momento 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: utilizar valores mais baixos em estados que já foram visitados muitas vezes, pois o agente já terá uma boa noção da qualidade

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

de cada ação possível, então há pouco que aprender; e utilizar valores mais altos em estados que foram visitados poucas vezes, pois o agente precisa aprender melhor sobre o estado.

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 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 visitado, e vice-versa em estados pouco visitados. Esse comportamento pode ser definido por uma função de exploração (*exploration function*).

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:

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

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 é o 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, 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:

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

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 número. Além disso, como o *approximate Q-learning* avalia características, apenas os pesos precisam ser atualizados:

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

Duas grandes vantagens de representar o Q-valor como uma combinação linear são evitar *overfitting* (a IA aprender tanto com o conjunto de treinamento que não consegue tomar decisões que diferem

demais dele), 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 aprendizado profundo e 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.

Como a ideia é o agente 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 os *frames* da tela do jogo por uma **rede neural convolucional**. Essa etapa funciona conforme descrito na seção sobre redes neurais convolucionais: os *frames* são a entrada, ocorre o processamento nas camadas ocultas e, na camada de saída, cada neurônio tem um valor. Contudo, calcular o erro é diferente. Como as imagens passadas não são rotuladas e não é possível fazer isso manualmente, a IA precisa calcular o erro da saída da rede de outra forma. É nessa etapa que entra o **aprendizado por reforço**. Como 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 e otimização dos pesos é feito pela parte de aprendizado por reforço.

O **cálculo do erro** é feito por uma função que, como toda função, possui um mínimo. Esse mínimo é normalmente calculado por alguma variação do **método do maior declive** ou **método do gradiente** [3]. Em matématica, gradiente é uma generalização de derivada para múltiplas variáveis e, portanto, aponta para a direção de maior crescimento da função sobre a qual foi aplicado no ponto dado. Método do gradiente, por sua vez, utiliza o **negativo** do gradiente para apontar para o mínimo local e, com isso, minimizar o valor da função.

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

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

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 e pode sequer chegar perto do mínimo do erro. Por outro lado, se for muito pequeno, a redução fica precisa, mas lenta. A figura 2.4 representa essas situações.

Existem diversos algoritmos de redução do erro e otimização dos pesos, como o *RMSProp*, *Adam*, *Adadelta*, dentre outros.

[3] *Gradient descent* em inglês

Figura 2.4: 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 Aprimorando o aprendizado

Deep Q-Learning é a base para o aprendizado da inteligência artificial deste trabalho. Porém, ao longo do tempo, foram descobertas outras técnicas que melhoram a aprendizagem do agente, seja acelerando ou evitando decisões nocivas. Nesta seção, serão apresentadas duas técnicas que ajudam a estabilizar e acelerar o aprendizado do programa: ***experience replay*** [Lin92] e **alvo fixo**.

2.9.1 *Experience Replay*

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 de estados sequenciais. Isso significa que ela se atualizaria apenas com as experiências passadas mais próximas, "esquecendo" as mais antigas. Para contornar esse problema, utiliza-se um *buffer* que armazena experiências anteriores e, a cada ação feita, ao invés de passar a transição de estados resultado dessa ação, passa-se uma amostra de experiências escolhidas aleatoriamente desse *buffer* para a rede ser atualizada. Utilizar *experience replay* também ajuda o agente a tomar ações diferentes ao invés de se prender a algumas que tiveram sucesso no passado próximo e que, possivelmente, não serão tão úteis no futuro.

2.9.2 Alvo fixo

A inteligência artificial utiliza os Q-valores tanto para decidir quais ações tomar quanto para atualizá-los na tentativa de melhor aproximar o Q-valor real. Entretanto, se o mesmo for utilizado para essas duas etapas, o aprendizado se torna instável, pois o agente não conseguirá "alcançar" o Q-valor alvo já que ele está em constante mudança. Para resolver esse problema, utiliza-se uma segunda rede neural alvo para a atualização dos Q-valores e que é atualizada de tempos em tempos, enquanto a primeira, a principal, continua sendo utilizada para escolher as ações.

Capítulo 3

Implementação

É conveniente formalizar o MDP que modela o problema antes de apresentar a arquitetura utilizada. Os estados S são os *frames* do jogo que tem 210x160 pixels, cada um com três canais que determinam sua cor e intensidade; 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 de 20 pontos por destruir um asteróide grande, 50 pontos por destruir um médio e 100 pontos por destruir um pequeno, podendo ser obtidas tanto atirando neles quanto colidindo, não havendo recompensa negativa (penalidade) por perda de vidas. E 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 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. *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 atpe desaparecer ou atingir um asteroide; cada tamanho de asteroide sempre aumenta a pontuação do jogador pela mesma quantidade quando destruído; e etc. 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.

3.1 Arquitetura

Antes de os *frames* serem passados para a CNN, eles passam por uma etapa de pré-processamento para reduzir o tempo de processamento. Primeiro, são convertidos para escala de cinza, removendo a necessidade de entender o que cores diferentes significam. Em seguida, a moldura da tela do jogo, partes que não agregam informação para a IA conseguir jogar, como pontuação, quantidade de vida e espaços sem nada, foram removidos. Depois, o que sobrou da tela foi redimensionad para 60x60 pixels. Por último, os últimos quatro *frames* vistos são inseridos em uma fila de forma que o agente consiga captar o movimento dos objetos na tela do jogo.

Esses quatro *frames* enfileirados são enviados juntos para a rede neural, de forma que a entrada tem formato 60x60x4. A primeira camada de convolução tem 32 filtros de tamanho 8x8 e passo 4, a segunda tem 64 filtros de tamanho 4x4 e passo 2, a terceira tem 64 filtros de tamanho 3x3 e passo 1. Todas elas são seguidas da função de ativação ReLU (*Rectified Linear Unit*). Depois disso, há uma camada *fully-connected* com 256 unidades e função de ativação ReLU e, por fim, outra camada *fully-connected* com 5 nós de saída, um para cada ação possível, sem função de ativação. Todas as camadas utilizam o inicializador de He [1] [He+15] para os pesos.

[1] No TensorFlow, esse inicializador é chamado pelo `variance_scaling_initializer()`

Após a CNN soltar a ação escolhida, o cálculo de erro é feito pela função *Huber loss* [Hub64] e a otimização dos pesos é feita pelo *Root Mean Square Propagation*, mais conhecido como *RMSProp* [HSS14].

Função de cálculo de erro *Huber loss* sendo a a diferença entre o valor previsto e o observado:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{para } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{c.c.} \end{cases} \tag{3.1}$$

Utilizando as mesmas variáveis da equação 2.12, $g = \nabla F(a_t)$, *decay* uma taxa de decaímento específica para o algoritmo e ϵ uma constante para evitar que a divisão seja feita por zero ou evitar que o gradiente diverja, o algoritmo *RMSProp* atualiza os pesos da seguinte forma:

$$s = decay * s + (1 - decay)g^T g \tag{3.2}$$

$$a_{t+1} = a_t - \alpha * g/\sqrt{s + \epsilon} \tag{3.3}$$

O *RMSProp* utilizou taxa de aprendizado $\alpha = 0.00025$, momentum $= 0.95$ (variável que faz considerar gradientes anteriores para determinar a direção do movimento) e $\epsilon = 0.01$. Foram 500 episódios de treinamento, cada um com limite 18000 ações antes de o episódio ser terminado automaticamente pelo código, *mini-batches* de tamanho 32, taxa de desconto $\gamma = 0.99$, *buffer* de memória de tamanho 1000000 que foi preenchido com 50000 ações aleatórias antes do treinamento começar. O dilema *exploration versus exploitation* utilizou $P_{ini} = 1.0$, $P_{min} = 0.1$ e $decay = 20000$. A rede alvo foi atualizada a cada 10000 ações tomadas.

3.2 Experimentos

Inicialmente, os testes foram feitos no *Asteroids*, conforme a proposta deste trabalho. Entretanto, pela falta de resultados positivos, tentativas de fazer o agente aprender em ambientes mais simples começaram a ser feitos: *Pong* do Atari2600, e *Gridworld*. Esse último, por ser um ambiente muito simples, serviu principalmente para validar o código e garantir que não há erros de implementação, apenas de hiper-parâmetros, já que o uso de CNN não traz melhorias significativas.

Depois dos experimentos no *Gridworld*, os testes moveram-se para um ambiente mais complexo, o *Pong* do Atari2600. Ele é mais parecido com o *Asteroids* por conta do espaço de estados e das entrada possíveis, exigindo o pré-processamento descrito acima. Os momentos que as recompensas são recebidas também são bem diferentes, ocorrendo vários passos depois da ação ser feita. Assim que os testes no *Pong* foram concluídos, voltou-se para o *Asteroids* com o que foi aprendido, como noção de quais números são bons, quais funções inicializadoras ou quais funções ativadoras são boas em cada camada.

No geral, o *Gridworld* foi rápido por conta de suas baixas dimensões e resultados rápidos. *Pong*, por outro lado, já levou várias horas, passando de um dia para o outro, por conta de sua maior entrada e espaço de estados.

Capítulo 4

Resultados

Ainda que os resultados não sejam muito expressivos, o agente obteve sucesso considerável no Gridworld. Os hiper-parâmetros escolhidos não foram os melhores, mas foram o suficiente para cumprir o objetivo dos testes.

No *Pong*, a pontuação final é igual a quantidade de pontos do jogador menos a quantidade de pontos do adversário. Como o jogo acaba quando algum dos lados fizer 21 pontos, a pontuação máxima e mínima são 21 e -21 respectivamente. Os resultados foram conforme o esperado considerando os artigos lidos sobre o assunto: lento, mas razoavelmente consistente. O gráfico abaixo mostra esse crescimento.

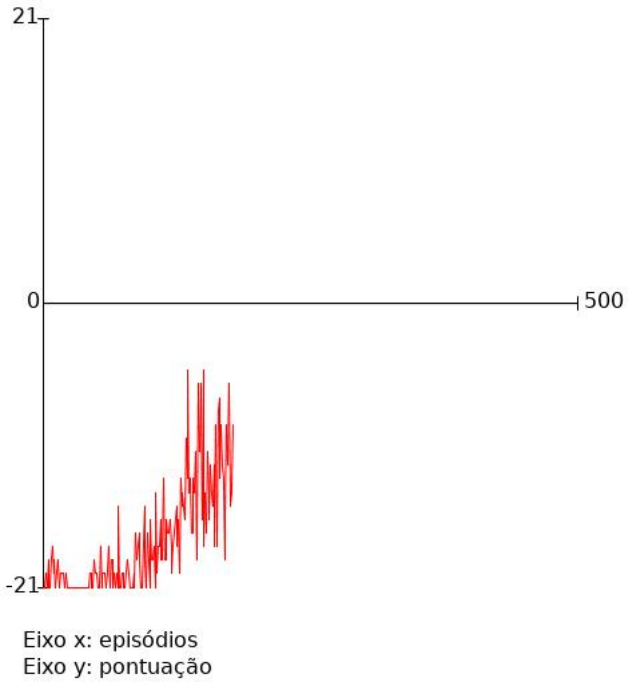

Figura 4.1: Pontuação do agente no *Pong* ao longo de 500 episódios de treinamento.

Capítulo 5

Conclusão - Parte Subjetiva

Este trabalho permitiu explorar uma técnica de aprendizagem de máquina que não é discutida nas disciplinas da graduação, ainda que seja uma junção de duas que são abordadas. A complexidade dos problemas que ela é capaz de resolver foi balanceada pela dificuldade de se utilizar com sucesso no ambiente almejado por este projeto. Há muitos hiper-parâmetros para se ajustar e o treinamento leva horas, até mesmo dias, para terminar. Além disso, não existem regras e teorias bem definidas para quais valores devem ser adotados, apenas relatos de casos bem sucedidos e algumas direções baseadas neles de quais podem ser bons. Mesmo assim, seu estudo e desenvolvimento foram interessantes, ainda que sucesso com o *Asteroids* não tenha sido obtido no final por conta de sua complexidade.

Referências Bibliográficas

[SB18] Sutton, R. S., Barto, A. G. *Reinforcement learning: an introduction.* 2nd ed. The MIT Press, 2018. ISBN: 9780262039246 9

[HSS14] Hinton, G., Srivastava N., Swersky K. Aula 6.5 - Divide the gradient by running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2): 26–31. 16

[Mni+13] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. A. *Playing Atari with Deep Reinforcement Learning.* Em: *CoRR* (dez. de 2013). ISSN: 1312.5602. 2

[Hub64] Huber, P. J. *Robust Estimation of a Location Parameter.* Em: *Ann. Math. Statist.* 35 (mar. de 1964), no. 1, pp. 73–101. DOI: 10.1214/aoms/1177703732. URL: https://doi.org/10.1214/aoms/1177703732 16

[He+15] He, K., Zhang, X., Ren, S., Sun, J. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.*. Em: *Corr* (fev. de 2015). ISSN: 1502.01852. 15

[Lin92] Lin, LJ. *Self-improving reactive agents based on reinforcement learning, planning and teaching.*, Em: *Machine Learning* 8(3–4) (mai. de 1992), pp. 293–321. ISSN: 1573-0565. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699 13