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 e em celulares. Eles normalmente são simples e intuitivos, para qualquer um poder começar a jogar a qualquer momento e rapidamente aprender como se joga. Porém, isso não é algo tão simples para computadores. Utilizando *deep learning* em conjunto com aprendizado por reforço, foi produzida uma inteligência artificial que aprendesse 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 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.

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 as ferramentas utilizadas na modelagem e treinamento da inteligência artificial deste trabalho.

2.1 *Asteroids*

Asteroids é um jogo de fliperama do gênero *shooter* (jogo eletrônico de tiro) lançado em novembro de 1979 pela então desenvolvedora de jogos eletrônicos Atari Inc, atualmente conhecida como Atari. O jogo foi inspirado em *Spacewar!*, *Computer Space*, *Space Invaders*, e *Cosmos*, sendo este último um jogo não finalizado. Diversas versões deste jogo foram criadas ao longo dos anos. As principais diferenças entre as iterações que *Asteroids* teve 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. *Asteroids* é considerado um dos primeiros grandes sucessos da era de ouro dos jogos de fliperama, época em que os jogos eletrônicos começaram a se tornar comuns na cultura popular.

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.

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

Pensando como um humano	**Pensando racionalmente**
O empolgante novo esforço de fazer computadores pensarem, serem máquinas com mentes, no sentido completo e literal da expressão (Haugeland, 1986)	*O estudo das faculdades mentais através de modelos computacionais* (Charniak & McDermott, 1985)
[A automação de] atividades que são associadas ao pensamento humano, como resolução de problemas, tomada de decisão, aprendizado, ... (Hellman, 1978)	*O estudo das computações que tornam possível a percepção, razão, e ação* (Winston, 1992)
Agindo como um humano	**Agindo racionalmente**
A arte de criar máquinas capazes de realizar funções que requerem inteligência quando feitas por pessoas (Kurzweil, 1990)	*Inteligência computacional é o estudo do design de agentes inteligentes* (Poole *et at*, 1998)
O estudo de como fazer os computadores fazerem coisas que, no momento, pessoas fazem melhor (Rich and Knight, 1991)	*IA... está relacionada ao comportamento inteligente em objetos* (Nilsson, 1998)

Tabela 2.1: Definições para inteligência artificial. Apresentação conforme a do livro de RUSSEL et al. (2010). Traduções livres feitas pelo autor.

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.

2.4 Inteligência artificial (IA)

Inteligência artificial (IA) é um dos campos mais recentes de ciência e de engenharia, tendo trabalhos no assunto sendo iniciados em meados do século XX. Atualmente, é composta por diversos campos menores de estudo, podendo ser mais genérico, como aprendizado e percepção, até mais específico, como a capacidade de jogar um jogo, provar teoremas matemáticos, ou dirigir um carro em uma via movimentada. RUSSEL et al. (2010) separaram oito definições de inteligência artificial em uma tabela de duas linhas por duas colunas, com duas definições por espaço. A linha de cima define processo de pensamento (*thought process*) e raciocínio (*reasoning*), enquanto a linha de baixo define comportamento (*behaviour*). Além disso, a coluna da esquerda mede o grau de fidelidade da inteligência quando comparado com performance humana, enquanto a da direita mede a racionalidade da performance - ou seja, se toma a ação "correta" dado o que o sistema sabe.

Essa tabela é representada pela tabela 2.1.

Em linhas gerais, as definições da coluna da esquerda dizem respeito a uma inteligência artificial que se pareça com um humano, enquanto as da direita sobre uma inteligência artificial que toma ações visando estar correta e a atingir o melhor resultado possível. Este trabalho terá um foco maior na categoria "**Agindo racionalmente**", pois as ações tomadas pelo agente terão como objetivo o retorno da maior recompensa possível.

2.4.1 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 que as inspirou, redes neu-

rais 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 nesse campo nos últimos anos, diversos tipos diferentes de redes neurais foram desenvolvidos, como redes neurais convolucionais (*Convolutional Neural Networks*, CNN) e redes neurais de memória de curto-longo prazo[1] (*Long/Short Term Memory*, LSTM). 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 abordadas 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 varia de acordo com o trabalho em questão. No caso deste, os neurônios da primeira camada têm como saída o número que representa cada pixel da imagem, após serem convertidos para uma escala de cinza e em um valor entre 0 e 1. Os quadros (*frames*) de entrada também são redimensionados antes de serem inseridos na rede para acelerar o processamento. Os neurônios da última camada, por outro lado, têm como saída o valor utilidade esperado[2] a ser recebido caso aquela ação seja tomada. Esse valor é utilizado para se tomar a decisão em cada *frame*.

A estrutura das redes neurais clássicas 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, como probabilidades ou valores utilidade esperados. No caso deste trabalho, a entrada é o *frame* atual do jogo e a saída é o valor utilidade esperado de cada ação possível (mover-se, atirar e etc). 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, o número de camadas ocultas e de neurônios em cada uma delas é arbitrário, sendo que essas quantidades são normalmente definidas por meio de experimentos.

Cada neurônio das camadas ocultas representa uma característica (*feature*) detectada pela IA 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, normalmente $[0, 1]$ ou $[-1, +1]$,

[1] Tradução livre feita pelo autor
[2] Mais detalhes sobre valor esperado na seção Aprendizado por reforço

a depender da função, 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, bem como facilita 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. No início, múltiplas saídas serão consideradas como boas respostas (alta probabilidade ou recompensa esperada), pois a IA tem comportamento aleatório. Para que o computador saiba o quão ruim foi sua saída, é definida uma função de erro (também conhecida como função de custo). Uma forma comum de calcular esse erro é o erro quadrático médio, que consiste em subtrair o valor devolvido pela IA pelo valor esperado para aquele neurônio da camada de saída e elevar o resultado ao quadrado. Após essas operações serem realizadas em cada neurônio da camada de saída, os resultados são somados e obtém-se um valor numérico para o quão errada a IA estava em um determinado exemplo. Quanto maior esse valor, mais incorreta a IA está. 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.

Seja M o número de exemplos dados para a rede neural, S o número de neurônios na camada de saída, h_i o vetor que representa a saída da IA para o i-ésimo exemplo, g_i o vetor que representa a saída esperada para o i-ésimo exemplo. O erro quadrático médio é dado por E_{EQM}:

$$\epsilon = (h_i - g_i)^2 \tag{2.3}$$

$$e_i = \sum_{j=1}^{S} \epsilon_j, \qquad \epsilon_j = \text{ j-ésimo elemento do vetor } \epsilon \tag{2.4}$$

$$E_{EQM} = \frac{1}{M} \sum_{i=1}^{M} e_i \tag{2.5}$$

Percebe-se que esse procedimento para avaliar o desempenho do código é uma forma de encapsular a rede neural em uma função que tem como entrada os pesos (W) e viés (b) de cada neurônio e, como saída, o quão bom ou ruim eles são (E_{EQM}). Por ser uma função, é possível calcular "para onde ela deve se mover" de forma que minimize a saída utilizando um método chamado de gradiente descendente (*gradient descent*). Gradiente é uma generalização de derivada para multiplas variáveis que representa a inclinação da tangente da função no ponto dado e aponta para a direção de maior crescimento, direção para onde a função deve "se mover" para chegar no máximo local. Consequentemente, se tomar o valor negativo do gradiente da função, obtém-se a direção de maior decrescimento, direção do mínimo local. Em outras palavras, gradiente descendente é um algoritmo utilizado para encontrar o mínimo local de

uma função, que é quase o que se deseja. O melhor caso para a função de erro seria encontrar o mínimo global, mas como isso nem sempre é possível e dificilmente alcançável, em parte por não haver garantias de que se encontrou o mínimo global, encontrar o melhor mínimo local é o suficiente.

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 faz os ajustes necessários 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.

2.4.2 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, foi escolhido esse tipo de rede para este trabalho.

2.4.3 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 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 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**. É possível dizer que se um filtro for arrastado para o lado, o local onde ele parar será o próximo filtro. 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*.

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.

2.4.4 Aprendizado por reforço

Aprendizado por reforço, diferente do supervisionado, não recebe exemplos com rótulos que dizem qual a resposta esperada como entrada. Ao invés disso, a IA, normalmente chamada de agente, interage com um ambiente e recebe recompensas positivas ou negativas por suas ações. Seu objetivo costuma ser tomar a ação com maior recompensa esperada para cada estado - ou seja, tomar a ação que acredite ser a melhor em cada instante.

Para domínios mais simples, como Jogo da velha, é possível determinar qual a ação com maior recompensa esperada para cada estado, a ação com maior probabilidade de vitória, por meio de uma árvore de decisão. Conforme o domínio se torna mais complexo, fazer esse mapeamento se torna inviável por conta da quantidade de estados que precisam ser armazenado, como é o caso do jogo *Asteroids*. Para se ter uma noção, um pixel que esteja diferente já é considerado um estado novo para o agente. Além disso, é comum haver situações em que não é possível determinar qual ação retornará a maior recompensa. Nesses casos, é mais viável criar e treinar um agente que aprenda a se comportar no ambiente em que está inserido do que informar se cada uma de suas ações em cada um dos estados possíveis é boa ou ruim.

Essa abordagem é conhecida como **Processo de Decisão de Markov** (*Markov Decision Process* (*MDP*)) para ambientes desconhecidos.

2.4.5 Processo de Decisão de Markov

Em um MDP padrão, a probabilidade de se chegar em um estado S' dado que o 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 (*value iteration*), 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 (*Bellman update* ou *Bellman equation*) até haver convergência de $V(S)$, que será denotado por $V^*(S)$. Esse $V^*(S)$ 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:

$$V^{(i)}(S) = \max_A \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^{(i-1)}(S')] \tag{2.6}$$

$$\lim_{i \to \infty} V^{(i)}(S) = V^*(S) \tag{2.7}$$

Política gulosa para função valor ótima:

$$\pi^*(s) = \operatorname*{argmax}_A \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^*(S')] \tag{2.8}$$

Entretanto, essas fórmulas são aplicáveis somente quando as funções de reforço R e de probabilidade de transição P são conhecidas, que não é o caso do jogo *Asteroids*. Para lidar com esse problema, foi adotado o uso de ***Q-learning***.

2.4.6 *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 que expressa a recompensa esperada, a qualidade em 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:

$$\begin{aligned} Q^*(S, A) &= \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^*(S')] \\ &= \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma \max_{A'} Q^*(S', A')] \end{aligned} \tag{2.9}$$

Logo, substituindo 2.9 em 2.8, temos que a política gulosa ótima para a função Q-valor ótima é dada por:

$$\pi^*(S) = \operatorname*{argmax}_A Q^*(S, A) \tag{2.10}$$

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 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),

$$\begin{aligned} Q^{(i+1)}(S, A) &= (1 - \alpha)Q^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma \max_{A'} Q^{(i)}(S', A')] \\ &= Q^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma \max_{A'} Q^{(i)}(S', A') - Q^{(i)}(S, A)] \end{aligned} \tag{2.11}$$

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 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, o agente sempre tomará a ação que retorne a maior recompensa imediata. 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 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*).

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, a solução é generalizar o aprendizado de um estado para o outro: se o agente sabe se comportar em um pequeno conjunto de estados, o ideal é ele saber o que fazer em um estado desconhecido contanto que seja suficientemente parecido com um já aprendido. Em outras palavras, o agente aprende propriedades (*features*) dos estados ao invés dos estados propriamente ditos, e toma decisões a partir dessas informações. Essa forma de fazer escolhar é chamada de ***approximate Q-learning***.

2.4.7 *Approximate Q-learning*

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.

$$Q(S, A) = \omega_1 f_1(S, A) + \omega_2 f_2(S, A) + ... + \omega_n f_n(S, A) \tag{2.12}$$

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:

$$\omega_k^{(i+1)}(S, A) = \omega_k^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma V(S') - Q^{(i)}(S, A)] f_k(S, A), k = 1, 2, ..., n \tag{2.13}$$

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.

Capítulo 3

Implementação

No capítulo anterior, Fundamentos, foram apresentados os dois principais conceitos para o desenvolvimento deste trabalho: **redes neurais convolucionais** e ***approximate Q-learning***. Para a construção da inteligência artificial que jogará o jogo *Asteroids*, cada uma dessas técnicas de aprendizado possui vantagens e desvantagens. **CNNs** são capazes de aprender características dos exemplos dados, mas eles precisam estar rotulados para isso. Uma vez que os exemplos fornecidos para o aprendizado seriam os *frames* do jogo, fornecer exemplos rotulados é uma tarefa impossível, pois não há uma ação correta ou uma melhor ação na maior parte dos estados. ***Approximate Q-learning***, por outro lado, aprende sem a necessidade de exemplos rotulados, mas precisa que as características que a IA deve aprender estejam bem definidas. Apesar de ser mais factível do que resolver o problema que a CNN enfrenta, não há garantias de que todas as características relevantes tenham sido codificadas e nem que foram bem definidas.

Portanto, a solução adotada foi unir as duas técnicas para o sucesso da IA no jogo: enquanto a parte de CNN se encarrega de identificar e aprender as características relevantes, a parte de aprendizado por reforço se preocupa em aprender quais as melhores ações para se tomar em cada momento. O resultado dessa união é uma técnica chamada de ***Deep Q-Learning*** ou ***Deep reinforecement learning***.

3.1 *Asteroids* ver. Atari2600

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. 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. 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 hiperespaço, 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 hiperespaço consiste em fazer a nave desaparecer por alguns instantes e reaparecer em um

Hiper-parâmetro	**Valor**
Dimensões dos *frames* passados para a rede	150x150 pixels
Número de *frames* enfileirados	4 *frames*
Primeira camada de convolução	24 filtros de 12x12 pixels
Segunda camada de convolução	48 filtros de 6x6 pixels
Número de episódios	30 episódios
Número máximo de ações por episódio	100 000 ações
Tamanho do *mini-batch*	32
Probabilidade de exploração inicial	1.0
Probabilidade mínima de exploração	0.1
Taxa de aprendizado α	0.000025
Taxa de desconto γ	0.7
Tamanho da memória	1 000 000 experiências

Tabela 3.1: Resumo da arquitetura da inteligência artificial

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. No topo da tela, há dois números indicando a pontuação total até o momento e quantidade de vidas restantes. Desconsiderando a moldura da tela, o espaço em que o jogo ocorre é de 177x152 pixels.

3.2 Arquitetura

Agora que as técnicas utilizadas para o treinamento e detalhes do ambiente foram apresentados, falta descrever a arquitetura da rede e detalhes do treinamento.

Antes de um *frame* ser analisado pela IA, suas cores são convertidas para escalas de cinza. Em seguida, a moldura da tela do jogo é removida, ficando visível apenas a área de jogo por onde a nave e os asteroides transitam. Os *frames* são então redimensionados para o tamanho 150x150 pixels.

Em seguida, os *frames* são inseridos em filas de tamanho quatro, com o mais antigo ficando no começo e o mais novo no final. Isso serve para que a IA tenha noção de movimento. Por exemplo, se ela vir uma imagem estática, não saberá para onde a nave e os asteróides estão se movendo e com que velocidade. Se vir quatro *frames* em seguida, conseguirá inferir essas informações. A cada vez que a IA vê um novo *frame*, o mais antigo é descartado e o novo é inserido no final.

Após esse pré processamento, os *frames* são passados para a rede neural convolucional. Ela possui duas camadas de convolução, com a primeira tendo 24 filtros de tamanho 12x12 e passo de tamanho 6, e a segunda tendo 48 filtros de tamanho 6x6 e passo de tamanho 3. Por fim, a rede possui duas camadas *fully-connected*. Em todas as camadas, é utilizada a função de ativação ELU. Em seguida, calcula-se o erro quadrático médio das saídas para avaliar o desempenho da IA para o exemplo dado. Como explicitado anteriormente, não é possível dizer qual a melhor ação a se tomar em cada *frame* do jogo e, portanto, não seria possível calcular o erro da rede. Para contornar esse problema, a diferença é feita entre o Q-valor desejado (parte $R(S, A, S') + \gamma \max_{A'} Q^*(S', A)$ da fórmula 2.9) e o Q-valor previsto para a saída da rede neural.

O treinamento teve 30 episódios, *mini-batches* de tamanho 32, probabilidade de exploração inicial 1.0 que decai até um mínimo de 0.1, taxa de aprendizado α igual a 0.000025, e taxa de desconto γ igual a 0.7.

Por fim, a IA utiliza uma técnica chamada de *experience replay* para melhorar o aprendizado. Ela

consiste em ter uma memória que guarda experiências passadas (tuplas (S, A, S' e R)) e utilizá-las aleatóriamente ao longo do treinamento. Primeiro, o agente jogará o jogo uma determinada quantidade de vezes tomando apenas ações aleatórias e armazenando as experiências na memória. Depois, quando estiver aprendendo, ao invés de passar o Q-valor desejado para o cálculo do erro da ação, a IA passa uma experiência passada. Como o ambiente é um processo de decisão Markoviano, cada ação afeta o próximo estado. Portanto, sequências de experiências estão altamente correlacionadas. Se o agente não ajustar os pesos da rede com experiências passadas, ele passará a agir apenas de acordo com o que acabou de fazer, esquecendo o que aprendeu no passado. A principal vantagem de utilizar *experience replay* é evitar esse esquecimento.

A tabela 3.1 resume a arquitetura da IA:

Por fim, a inteligência artificial utiliza o modelo construído durante o treinamento para tentar jogar sozinha. Nesta etapa, não existe aleatoriedade: se a IA jogar sempre as mesmas fases nas mesmas ordens, as ações, e por consequência, a pontuação serão sempre as mesmas.

3.3 Resultados

Após inúmeras tentativas e erros para se chegar na arquitetura acima

Capítulo 4

Conclusões - Parte Subjetiva