Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Fábio Henrique Kiyoiti dos Santos Tanaka

Inteligência artificial para o jogo de Hex

São Paulo
Dezembro de 2018

Inteligência artificial para o jogo de Hex

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Denis Maua

São Paulo
Novembro de 2018

Resumo

Hex é um jogo de tabuleiro jogado em uma grade hexagonal no qual cada par de lados opostos tem uma cor, esta grade pode ter quaisquer dimensões ou formas mas geralmente é organizada em um losango de dimensões 11×11 ou 13×13. Uma partida de Hex tem 2 jogadores, cada um com peças de uma cor. Durante o jogo eles se alternam colocando suas peças em espaços vazios do tabuleiro, o objetivo é formar um caminho que conecte os lados opostos marcado com suas cores, o primeiro jogador a fazer isso é considerado o ganhador.

Apesar da simplicidade das regras, o jogo de Hex apresenta uma extensa profundidade estratégica devido ao grande número de jogadas possíveis em cada momento. Este trabalho busca desenvolver uma inteligência artificial para este jogo e assim estudar diferentes conceitos da área de aprendizado de máquina e aprendizado por reforço.

Como o jogo de Hex possui muitos estados possíveis o programa foi desenvolvido utilizando *Deep Q-learning*, uma estratégia que usa redes neurais para avaliar cada jogada possível. A implementação dos algoritmos foi feita por meio da biblioteca *pytorch* [11] da linguagem *python* e para avaliar o progresso foi utilizado um oponente que jogava de acordo com uma heurística forte empregada no *Hexy* [2], o programa campeão da "*5th Computer Olympiad*" [1].

A implementação da inteligência artificial teve sucesso e depois de ser devidamente treinada atingiu um índice de vitórias de 77% quando iniciava o jogo e de 9% quando jogava em segundo. Estes resultados foram obtidos em partidas num tabuleiro 5x5 mas outros testes indicam que com a devida otimização e mais tempo de treinamento esse programa pode ser utilizado mesmo em tabuleiros maiores.

Palavras-chave: Aprendizado por reforço, *Deep Q-learning*, Redes neurais, Hex.

Sumário

Capítulo 1

Introdução

1.1 Motivação

A utilização de jogos para testar uma IA (inteligência artificial) não é uma ideia nova; eles são um bom objeto de estudo para este campo já que apresentam um universo suficientemente complexo, com muitas estratégias e possibilidades, mas ainda assim controlado, já que em cada momento há um número limitado de ações possíveis. Em 1954 já haviam sido criados *softwares* que jogavam damas melhor que um jogador humano médio e em 1997 o computador *Deep Blue* da IBM ganhou fama quando derrotou o campeão mundial de xadrez da época, Garry Kasparov, sendo este considerado um grande marco na história da computação.

Em 2015 outra IA se tornou famosa. Desta vez o programa AlphaGo desenvolvido pela Alphabet Inc.'s se tornou o primeiro a ganhar de um jogador profissional de Go sem nenhum tipo de *handicap*. O que torna esta vitória impressionante é que antes se achava que era impossível ter uma IA suficientemente boa para o jogo de Go já que ele apresenta um número de estados possíveis tão grande que é infactível de um computador calcular todos eles. Para realizar este feito o AlphaGo analisou milhares de partidas jogadas por humanos e treinou jogando contra si mesmo para assim desenvolver uma estratégia.

O resultado apresentado pelo programa se mostrou tão importante que foi inclusive considerado um dos maiores avanços científicos do ano e atraiu atenção de todo o globo. Diante disso, o estudo de aprendizado de máquina e dos algoritmos que permitiram tal feito se mostram de grande importância, não somente pelo o que já conquistaram, mas principalmente pelas possibilidades que ainda podem trazer no futuro.

1.2 O jogo de Hex

Hex é um jogo de tabuleiro abstrato inventado pelo matemático dinamarquês Piet Hein em 1942 e introduzido no instituto Niels Bohr. Em 1948 outro matemático, John Nash, sem ter conhecimento da invenção de Piet re-inventou o jogo e o popularizou quando o introduziu na Universidade de Princeton.

Hex é jogado por 2 jogadores em uma grade hexagonal que teoricamente pode ter tamanho e forma variáveis. Apesar disso o tabuleiro tradicional é construído na forma de um losango com dimensões 11×11 ou 13×13 e cada par de lados opostos recebe uma cor distinta. Durante a partida cada jogador recebe peças de uma dessas cores e se alternam colocando-as em espaços vazios do tabuleiro. O objetivo do jogo é criar um caminho utilizando suas peças de forma a conectar os lados opostos do tabuleiro que estejam marcados com a sua cor.

Não é possível ocorrer empates no jogo. No máximo um participante pode ter um caminho ganhador e em um tabuleiro totalmente preenchido sempre havera tal caminho, tal fato foi provado pela primeira vez por John Nash em 1952 [9]. Outros teoremas sobre o jogo definem que para um tabuleiro $n \times n$ sempre existe uma estratégia vencedora para o primeiro jogador [4] e que determinar se é possível ganhar a partir de uma certa posição é PSPACE-completo [12].

Apesar das regras simples, o jogo de Hex requer tanto planejamento estratégico quanto habilidades táticas tornando-o adequado para ser usado no estudo de algoritmos de aprendizado de máquina.

(a) *Jogo em progresso* **(b)** *Jogo finalizado*

Figura 1.1: *O jogo de Hex.*

1.3 Trabalho relacionado e objetivo

Este projeto é inspirado pelo *paper* "Neurohex: A Deep Q-learning Hex Agent" [7]. Nele é criado um agente para jogar Hex que utiliza *Deep Q-learning* e aplica algoritmos como Redes Neurais Convolucionais (CNNs) e *Experience Replay*.

O objetivo deste trabalho é criar um agente que funcione de forma similar a fim de estudar as técnicas utilizadas e assim aprofundar o conhecimento na área de inteligência artificial, mais especificamente em aprendizado de máquina.

1.4 Implementação e testes

Esta inteligência artificial foi implementada aplicando *Deep Q-learning* ao jogo de Hex. Redes neurais convolucionais (CNNs) foram a arquitetura de rede escolhida para avaliar o tabuleiro e as jogadas possíveis.

O programa foi escrito utilizando a linguagem *python* e em especial as bibliotecas *numpy* [6] e *pytorch* [11], a primeira delas permite o cálculo de operações com matrizes de forma eficiente e a segunda serve para modelar uma rede neural. *Numpy* [6] foi usada principalmente para criar o modelo do jogo e *pytorch* [11] para criar a CNN.

Por fim, o projeto foi testado jogando contra um oponente que utiliza uma heurística forte baseada em circuitos elétricos. Essa mesma heurística foi empregada no *Hexy* [2], o programa campeão da "*5th Computer Olympiad*" [1]. Após realizar o aprendizado, a IA obteve sucesso ao ganhar até 77% das partidas quando ela jogava primeiro e 9% quando jogava em segundo. Dados obtidos durante os testes sugerem que esses resultados podem inclusive melhorar com mais tempo de treinamento ou otimização individual.

1.5 Estrutura do Trabalho

No capítulo 2 será realizada uma explicação dos principais conceitos teóricos utilizados para a construção do programa. No capítulo 3 será explicado como foi realizada a implementação junto com a explicação de algumas decisões de projeto. Por fim no capítulo 4 serão analisados os resultados obtidos.

Capítulo 2

Revisão Teorica

2.1 Aprendizado por Reforço

Aprendizado por reforço é um campo do aprendizado de máquina no qual estuda-se como um agente aprende como interagir com o ambiente por meio de tentativa e erro, isso é, aplicando ações e observando os resultados. Todas as definições e conceitos desta sessão (2.1) foram extraídas do livro *"Artificial Intelligence: a modern approach"*[13].

Em cada passo do processo de aprendizado o agente recebe do ambiente o estado em que se encontra e então seleciona uma ação para tomar. Após isso o ambiente retorna dois valores: o novo estado e uma recompensa, que pode ser tanto positiva quanto negativa, que indica o quão boa foi a ação realizada. Este processo então é repetido até que o ambiente devolva um estado terminal e assim finalize o episódio.

O modelo acima é formalizado matematicamente utilizando um MDP (*Markov Decision Process*)

2.1.1 Processos de Decisão Markovianos (MDP)

Um Processo de Decisão Markoviano (*Markov Decision Process* - MDP) é um processo estocástico que obedece à *propriedade de Markov*: **as consequências de uma ação em determinado momento dependem somente do estado atual e não da sequência de eventos que o precederam**. Além de seguir esta propriedade um MDP é composto pelos seguintes elementos:

- Um conjunto de estados S.
- Um conjunto de ações A.
- Uma função $R(s_t, a_t)$ que devolve um valor real representando a recompensa ao realizar a ação a_t enquanto no estado s_t
- Uma função $P(s_t, a_t, s_{t+1})$ que devolve a probabilidade de se passar para um estado s_{t+1} ao se tomar a ação a_t no estado s_t

A imagem a seguir ilustra o funcionamento deste modelo:

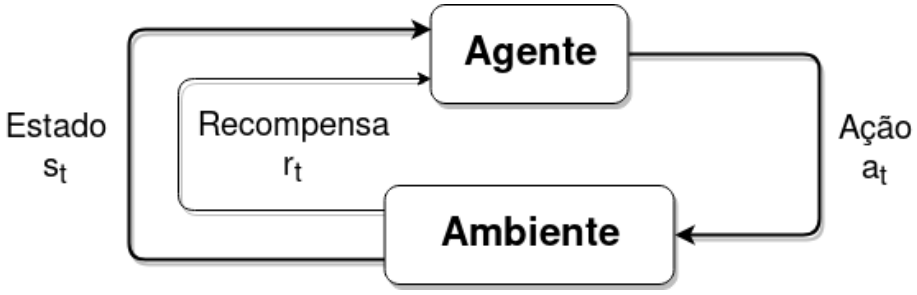


Figura 2.1: *Ilustração do funcionamento de um MDP*

A partir deste conjunto de elementos pode-se também definir uma política π, uma função que ao receber um estado determina uma ação que o agente deve tomar: $\pi(s_t) \rightarrow a_t$.

O objetivo neste modelo é maximizar a recompensa acumulada $\sum_{t=0} \gamma^t r_t$ onde γ é um fator de desconto entre 0 e 1 que indica o quão mais importante são as recompensas imediatas. Para este fim, o valor de uma política π em um estado s é avaliado da seguinte forma:

$$V_\pi(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s)) + \gamma * V(s')] = E[\sum_{t=0} \gamma^t r_t | s_0 = s, \pi]$$

Portanto busca-se uma política π^* que maximize esse valor para um estado inicial.

2.1.2 Grid-World

Um exemplo comum quando se trata de MDPs é o *Grid-world*. Nele o agente se encontra em um tabuleiro e deve se mover com o objetivo de chegar em um estado terminal.

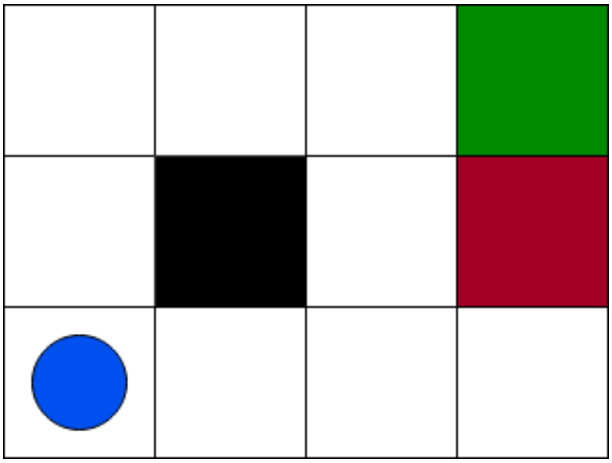


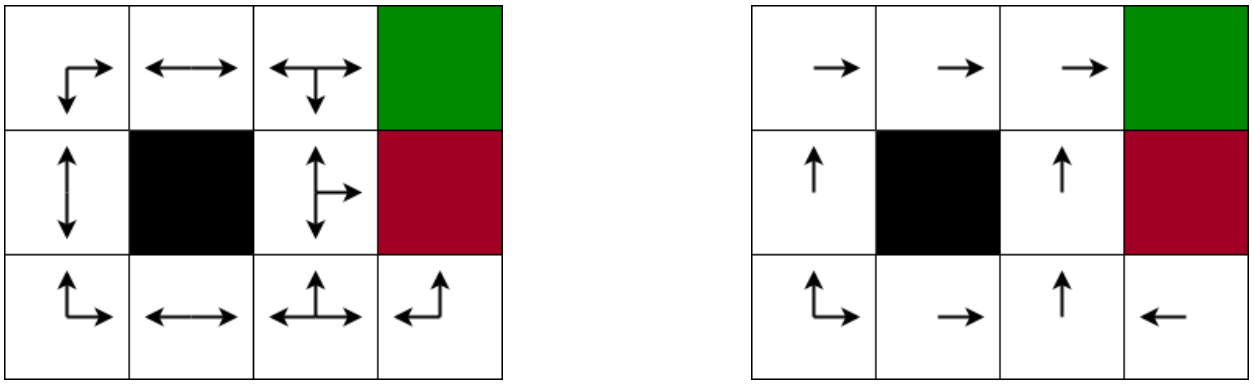
Figura 2.2: *Representação do Grid-world*

No exemplo acima o ambiente é a matriz de dimensões 3×4, o agente é representado pelo círculo azul e o conjunto de estados possíveis são as posições que ele pode ocupar neste tabuleiro. Os quadrados pintados de verde e vermelho representam os estados terminais e o espaço em preto representa um obstáculo.

Neste cenário as ações possíveis para o agente são as direções em que ele pode se locomover. É definido que ele pode se mover 1 espaço por ação desde que ele não saia do tabuleiro e nem ocupe o mesmo espaço que o obstáculo. Quando o agente realiza uma ação que o posiciona em um estado terminal o episódio é finalizado

As recompensas neste exemplo são definidas da seguinte maneira: qualquer ação que mova o agente para um estado não terminal tem recompensa 0, se a ação o mover para o espaço verde a recompensa é de +1 e caso ela o mova para o espaço em vermelho a recompensa é de -1. Desta forma o objetivo seria chegar no espaço verde e evitar o quadrado vermelho.

Por fim, o modelo acima é determinístico, isso é, a ação tomada em qualquer estado tem 100% de chance de determinar o próximo estado. Com essas definições tem-se um MDP e é possível determinar políticas, como mostrado na figura 2.3. O primeiro exemplo apresentado nesta imagem é de uma política aleatória, nele é indicado que em todos os estados o agente pode selecionar qualquer ação legal. O segundo exemplo é de uma política ótima, ele designa o melhor movimento a ser tomado a partir de qualquer posição de forma a se chegar no objetivo com o menor número de movimentos.



(a) *Política aleatória* **(b)** *Política ótima*

Figura 2.3: *Políticas para o Grid-world*

2.1.3 Função Q

Uma forma de estimar os valores de um ambiente em uma MDP é por meio da função Q. Intuitivamente, a função Q representa o valor esperado da soma das recompensas quando se toma a ação a em um estado s e depois continua-se agindo de acordo com uma política π. Esta equação é representada da seguinte maneira:

$$Q^{\pi}(s,a) = \sum_{s'} P(s,a,s')[R(s,a) + \gamma * V_{\pi}(s')] = E[\sum_{t=0} \gamma^t r_t | s_0 = s, a_0 = a, \pi]$$

Uma função Q é considerada ótima quando ela utiliza a política que maximiza o seu valor. Neste caso ela é chamada de Q^* e satisfaz a equação de Bellman:

$$Q^*(s_t, a_t) = E[r_t + \gamma \max_a Q^*(s_{t+1}, a) | s_t, a_t]$$

A lógica para esta equação é simples, após receber a recompensa r_t será selecionada e realizada a melhor ação possível a partir do novo estado, então este processo é repetido até se chegar a um estado terminal. É importante notar que essa função ocorre de forma gulosa pois em cada estado é selecionada a melhor ação local.

2.1.4 Exploração vs exploitação

Um tópico que é relevante abordar durante o estudo de aprendizado por reforço é a questão da "Exploração vs exploitação".

Exploitação é tentar se aproveitar das informações obtidas nas experiências anteriores para tomar a ação que maximize a recompensa esperada. Exploração, por outro lado, é selecionar uma ação com o objetivo de descobrir mais sobre o ambiente.

É importante que o sistema realize os dois tipos de ação. Na maior parte do tempo o agente deve tomar a ação que ele considera melhor, mas deve haver uma chance dele tomar uma ação aleatória para explorar o ambiente. Isso permite que o programa descubra novas rotas de ação e possivelmente uma delas pode conter uma melhor recompensa.

2.1.5 Q-learning

Q-learning é um algoritmo usado para aproximar o valor da função Q, para isso normalmente utiliza-se uma tabela que armazena o valor estimado da função para cada par ação, estado. Esta tabela é inicializada com 0 em todas as coordenadas e a cada iteração o valor é atualizado de acordo com a seguinte equação:

$$\underbrace{Q(s_t, a_t)}_{\text{novo valor}} \leftarrow (1-\alpha) \underbrace{Q(s_t, a_t)}_{\text{valor antigo}} + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a))$$

Nesta equação α é chamado de *learning rate* e indica o quanto experiências novas devem sobrescrever experiências antigas. Após execuções o suficiente é esperado que os valores da tabela tenham se aproximado

dos valores de Q^*. Note que não é necessário especificar a função de transição P para este algorítmo, isto é muito util pois esta função pode ser complexa ou mesmo desconhecida.

A figura abaixo mostra como foi aproximado o valor da função Q após 1000 execuções utilizando o *Grid-World* exemplificado na sessão 2.1.2.

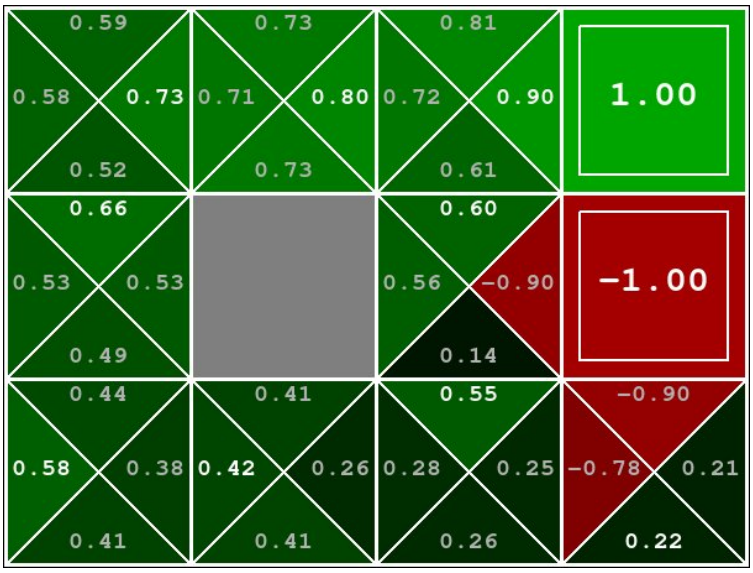


Figura 2.4: *Valores da função Q após 1000 execuções*

Apesar desta abordagem funcionar bem no caso do *Grid-world* ela apresenta 2 grandes problemas. O primeiro é que ela não generaliza situações semelhantes, isso é, estados parecidos são tratados de formas totalmente diferentes e tem suas funções Q calculadas de forma independente. O segundo problema é que é necessário mapear o valor de todo o conjunto de tuplas ação, estado e isso é muito ineficiente quando se trata de espaços muitos grandes. No Hex, por exemplo, mesmo um tabuleiro 5×5 que é considerado pequeno tem 847288609443 estados possíveis já que em cada um dos 25 espaços do tabuleiro pode se ter um de três possíveis valores: vazio, ocupado por peça preta e ocupado por peça branca. Devido a esse problema são utilizadas redes neurais profundas para realizar o aprendizado no processo conhecido por *Deep Q-learning*.

2.2 Aprendizado por Reforço Profundo

Aprendizado profundo é um campo do aprendizado de máquina que tem como objetivo extrair características (*features*) diretamente dos dados. Para isso ele se baseia em abstrações de alto nível que utilizam grafos para representar várias camadas de processamento. Neste projeto serão utilizadas Redes Neurais Convolucionais como a arquitetura para o aprendizado profundo.

Os conceitos de Redes Neurais artificiais (ANNs) e de Redes Neurais Convolucionais (CNNs) foram extraidos de [5] e [10], enquanto os conceitos de *Deep Q-learning* e *Experience Replay* tiveram origem a partir de [3].

2.2.1 Redes Neurais Artificiais (ANNs)

Uma Rede Neural Artificial (*Artificial Neural Networks* - ANNs) é um modelo computacional inspirado pelo funcionamento do cérebro no reino animal. A ideia deste sistema é que a partir de exemplos seja possível aprender e assim realizar tarefas.

Uma ANN é composta por unidades menores chamadas de neurônios que recebem um vetor de valores X, os multiplica por um vetor de pesos W e os soma. Após isso é adicionado um viés b e para que o resultado fique entre 0 e 1 é aplicada uma função de ativação θ. Ou seja, o neurônio realiza a operação $\theta(W^T X + b)$ e devolve o resultado.

Estes neurônios são então organizados na forma de uma rede por camadas tal que a primeira delas recebe

como entrada um vetor real. Cada camada intermediária é chamada de camada oculta e recebe como entrada os valores dos neurônios do nível anterior. Por fim a última delas é chamada de camada de saída e os valores de seus neurônios são organizados em um vetor sendo ele a saída da rede.

(a) *Neurônio* **(b)** *Rede neural artificial*

Figura 2.5: *Exemplos de neurônio e ANN*

Um exemplo clássico no estudo de ANNs é a classificação de números escritos a mão. A entrada da rede são diversas imagens com dígitos e o objetivo é definir quais números cada uma delas representa. Neste exemplo a *output layer* teria 10 neurônios, cada um representando um dígito entre 0 e 9, e seria escolhido como resultado o número cujo respectivo neurônio tivesse o maior valor.

A rede é inicializada com pesos (W) e viés (b) arbitrários e, devido a isso, as previsões que ela faz no início não são precisas. Como de costume, estas predições feitas pela rede neural são avaliadas através de uma função de custo ou de erro. Um exemplo de função comumente utilizada é o erro quadrático médio dado por:

$$loss = \frac{1}{n} \sum_{i=0}^{n-1} \parallel \hat{y}_i - y_i \parallel^2.$$

Nesta função, n é o número de entradas utilizadas, $\hat{y}_i$ é o vetor de saídas esperadas do exemplo i e y_i é o vetor de previsões da rede para a execução i. Os valores de W e de b são então modificados com o intuito de minimizar o valor desta função. Gradiente descendente e *backpropagation* são técnicas utilizadas para realizar esta modificação, as definições destes algoritmos podem ser encontradas em [10].

O processo conhecido como aprendizado profundo (*Deep Learning*) é o uso de ANNs com muitas camadas ocultas para assim modelar abstrações mais complexas.

2.2.2 Deep Q-learning

No Q-learning tradicional é necessário calcular o valor da função Q para cada par estado, ação e isto é infactível para o jogo de Hex pois existem muitos estados possíveis. Por esse motivo é utilizado o chamado *Deep Q-learning* que aplica redes neurais como forma de aproximar Q de Q^*.

No *Deep Q-learning* uma rede neural é utilizada para avaliar as opções. Para isso cada neurônio da camada de saída é responsável por aproximar o valor da função Q para uma ação em dado estado. O treinamento desta rede é feito utilizando a seguinte função erro para uma rede com pesos W:

$$loss = 0.5[\hat{y} - Q_W(s_t, a)]^2$$

Onde $\hat{y}$ é o resultado esperado e vale:

$$\hat{y} = \begin{cases} r & \text{caso } s_{t+1} \text{ seja terminal} \\ r + \gamma \max_{a_{t+1}} Q_W(s_{t+1}, a_{t+1}) & \text{caso contrário} \end{cases}$$

É importante notar que nesta operação o valor da da função erro utiliza a própia rede em sua definição; isso é incomum em aprendizado de máquina e é uma peculiaridade do *Deep Q-learning*. Tendo esta equação o sistema é iterado da seguinte forma:

Comece por um estado inicial s_0 e com tempo $t = 0$;
enquanto s_t *não for terminal* **faça**
  Com probabilidade e selecione uma ação aleatória a_t;
  Caso contrário selecione a melhor ação a_t de acordo com a rede neural;
  Execute a ação a_t;
  Receba a recompensa r_t e o novo estado s_{t+1};
  Calcule o erro baseado na diferença da recompensa esperada e da recebida;
  Realize a *backpropagation* utilizando este resultado;
  $s_t \leftarrow s_{t+1}$;
  Incremente t;
fim

2.2.3 Experience Replay

Um problema comum quando se utiliza *Deep Q-learning* é que há uma grande correlação entre ações e estados sucessivos e isso pode influenciar os pesos da rede de forma negativa. Quando executamos muitos experimentos em sequência o que pode acontecer é que a rede esqueça experiências mais antigas sobrescrevendo os pesos delas utilizando experiências mais novas.

Para resolver esta questão é utilizado um *batch* de jogadas que armazena diversas tuplas na forma $t = (s_t, a_t, s_{t+1}, r_t)$. Com isso, quando é necessário atualizar a rede, é selecionada somente uma amostra aleatória deste conjunto de tuplas e com isso a rede aprende ao mesmo tempo diversas situações diferentes que não são necessariamente sequenciais.

2.2.4 Redes Neurais Convolucionais

Redes neurais convolucionais (*Convolutional Neural Networks* - CNNs) são uma classe de ANNs do tipo *feed forward* (quando as conexões entre os neurônios não formam ciclos) normalmente utilizadas para a análise de imagens. O intuito de uma CNN é extrair e mapear características da entrada para facilitar o processamento dos dados. Neste projeto o tabuleiro de Hex é tratado como uma imagem e CNNs são utilizadas para auxiliar o processamento.

Na construção de uma CNN são realizadas 4 operações: convolução, aplicação de função não-linear (ReLU), *pooling* e classificação. As 3 primeiras delas formam uma camada oculta (*hidden layer*) e podem ser utilizadas repetidas vezes na construção do modelo.

A convolução é o processo que dá nome à rede e nela a ideia é extrair informação dos dados por meio da aplicação de um filtro (também chamado de *kernel*). A operação é feita deslizando o filtro sobre a matriz de entrada e em cada posição os valores são multiplicados e então somados de forma a criar um novo valor no que é chamado de *feature map*. Este processo pode ser visualizado na imagem abaixo.

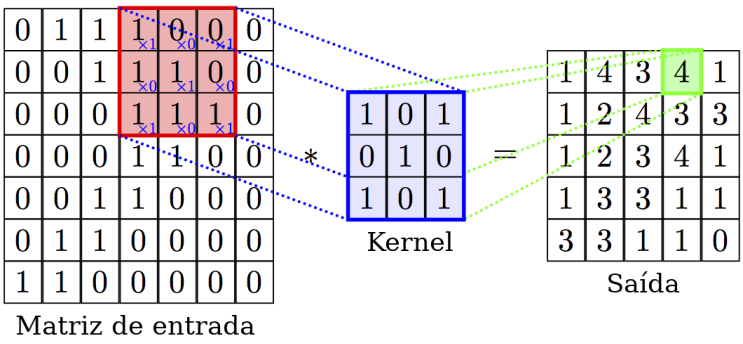


Figura 2.6: *Operação de convolução*

Após a convolução é aplicado em cada elemento uma função chamada de ReLU (*Rectified Linear Unit*) tal que $ReLU(x) = \max(0, x)$. Esta operação troca qualquer valor negativo por 0 e tem como objetivo tornar o dado não-linear pois assim ele fica mais flexível, já que aborda mais situações, e próximo da realidade, já que a maioria dos casos no mundo real é não-linear.

O processo de *pooling* reduz a dimensão do *feature map* tentando manter, ainda assim, as informações relevantes. O tipo mais comum é o *max pooling* que seleciona o maior valor de certo intervalo do dado. O objetivo desta parte é diminuir o tamanho da amostra para agilizar o treinamento e prevenir o *overfitting*.

Por fim é realizada a classificação do dado. Após a entrada ser processada pelas camadas anteriores o resultado é avaliado e classificado por uma ANN.

Capítulo 3

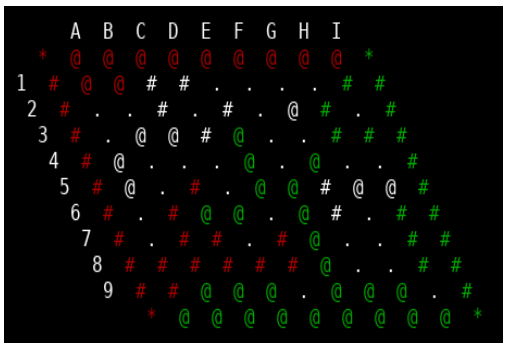
Modelo

3.1 Representação de Estados

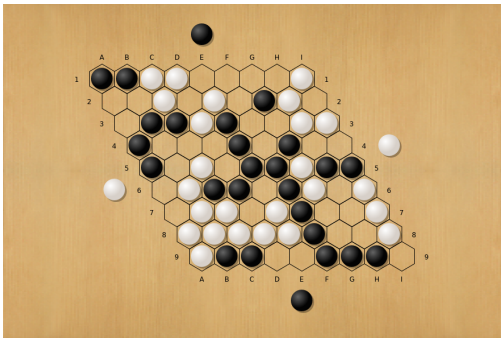
Dado um tabuleiro de tamanho $n \times n$, cada estado s em S é representado como um vetor booleano de tamanho $6 \times n \times n$, representando as seguintes 6 "camadas" de informações do jogo:

- Presença de peça preta
- Presença de peça branca
- Peças brancas conectadas ao lado esquerdo do tabuleiro
- Peças brancas conectadas ao lado direito do tabuleiro
- Peças pretas conectadas ao lado superior do tabuleiro
- Peças pretas conectadas ao lado inferior do tabuleiro

Em cada uma destas matrizes é adicionado um *padding* ao redor do tabuleiro que é usado para simplificar o algoritmo que realiza a busca por conexões nas bordas. Esta arquitetura é a mesma utilizada no Neurohex [7] no qual este projeto foi inspirado.



(a) *Modelo em modo texto*



(b) *Representação da mesma partida em tabuleiro*

Figura 3.1: *Visualização do jogo de Hex*

Toda esta representação foi feita utilizando a biblioteca *numpy* [6] do python pois ela permite manipulações mais rápidas de operações em matrizes e no cálculo da heurística que será explicada mais para frente.

3.2 Representação de Ações e Recompensa

Neste projeto cada ação possível é representada por um inteiro entre 0 e n^2 onde n é o tamanho do tabuleiro. Dessa forma haverão n^2 neurônios na última camada da rede neural e cada ação a em A será equivalente ao neurônio com maior valor. É importante comentar que todas as ações são avaliadas independente

do estado, caso a de maior resultado seja uma ação ilegal (colocar uma peça em uma posição já ocupada) é considerada a próxima com maior valor.

A função recompensa devolve um valor a partir do estado do jogo após a ação ser realizada. Se o estado é não terminal a recompensa é 0, caso contrário ela vale +100 se o agente ganhar o jogo e -100 caso ele perca.

3.3 Tabuleiro

Como definido anteriormente o jogo de Hex pode ser jogado em tabuleiros de diferentes tamanhos sendo usado como padrão uma grade 11×11 ou 13×13. Quanto maior o tabuleiro maior a complexidade do jogo pois existem mais possibilidades de jogada em cada turno além de mais estados possíveis no total.

Outro fator influenciado pelo tamanho do tabuleiro é o número de jogadas em uma partida. Em um tabuleiro $n \times n$ o menor caminho entre duas bordas opostas tem tamanho n e já que os jogadores se alternam colocando peças é simples perceber que o número mínimo de jogadas para finalizar o jogo é $2n-1$. Por outro lado o número máximo de jogadas que podem ocorrer em um só jogo é n^2, isso acontece quando todos os espaços do tabuleiro são preenchidos e só a última jogada define o ganhador.

Esses dois fatores, número mínimo e máximo de jogadas, são especialmente relevantes quando vai se treinar a IA pois são jogadas centenas de milhares de partidas. Se o jogo ocorre de forma aleatória ou sem muita estratégia por parte dos dois jogadores o número de turnos tende a se aproximar de n^2. Quando um jogador tem uma estratégia considerada fraca e joga contra um oponente forte, o jogo tende a ser mais curto e ter um número de turnos mais próximo de $2n-1$.

Por esses motivos foi optado por treinar a IA em um tabuleiro pequeno de dimensões 5x5. Isso permitiu uma análise mais detalhada durante a criação do programa já que possibilita a realização de mais testes em um tempo de treinamento menor. Apesar da maior parte do desenvolvimento e dos testes terem sido realizados neste tamanho de tabuleiro, também foi testado o funcionamento em grades maiores e apesar do processo de treino demorar significantemente mais ele ainda apresentou resultados positivos. Este tópico será abordado mais a fundo no próximo capítulo.

3.4 Treinamento

Utilizando os algoritmos e técnicas discutidos no capitulo 2 é feito o treinamento da rede. Em uma sessão de treinamento são realizadas M partidas contra o oponente com o objetivo de fazer a CNN fornecer avaliações mais precisas de cada jogada. Em cada partida o agente e o oponente se alternam realizando suas ações.

Quando é o turno do agente, primeiro ele salva o estado atual, isso é, a matriz que representa o tabuleiro no momento. Depois é selecionado uma jogada aleatória com probabilidade e, caso contrário, utilizando a CNN é escolhida a ação com o maior valor esperado. Essa jogada é realizada e recebe-se a recompensa. Ao fim desta sequência estarão salvos três valores: o estado antes de realizar a ação, a jogada realizada e a recompensa recebida. Eles são armazenados nas variáveis s_{antigo}, a e r respectivamente.

Durante o turno do oponente, ele seleciona e realiza uma ação de acordo com sua configuração. Se esta jogada não finalizar a partida, o novo estado que o tabuleiro se encontra é salvo em uma variável s_{novo}, caso contrário, esta variável recebe o valor *null* para indicar que é um estado terminal. Após isso, é adicionado à memória de *Experience Replay* a transição representada pela tupla: s_{antigo}, a, r e s_{novo}.

Após o fim de cada jogo, é selecionado de forma aleatória um *batch* de N transições da memória. É calculado o valor da função erro para cada tupla do *batch* como discutido na sessão 2.2.2 e utilizando estes valores é realizado o *backpropagation*.

É importante notar que quando o agente realiza uma ação ele nunca recebe a recompensa de derrota pois nenhum jogador consegue formar um caminho para o adversário ao jogar uma de suas peças. Devido a isso, é necessário verificar se o oponente ganhou a partida após ele jogar, se este for o caso a recompensa é substituída pelo valor equivalente à derrota.

Existe mais uma verificação que é necessário realizar. Caso o jogador tenha ganho a partida, a transição referente à vitória não foi adicionada na memória pois este processo só ocorre após a jogada do oponente. Para corrigir isso, ao final do jogo, se o agente ganhou é adicionado uma nova tupla à memória com s_{novo} valendo *null* para indicar que este é um estado terminal.

Abaixo é possível encontrar o pseudo-código referente ao treino, note que nele o agente obeserva somente as suas própias jogadas.

inicialize a CNN com todos os pesos valendo 0;
inicialize o oponente;
inicialize a memória de *Experience Replay*;
para *episodio = 0* **até** *M* **faça**
  inicialize um novo jogo;
  turno = 0;
  enquanto *jogo não tiver vencedor* **faça**
    se *é o turno do agente* **então**
      $s_{antigo} \leftarrow$ estado de jogo atual;
      com probabilidade e selecione uma ação aleatória a;
      caso contrário selecione a melhor ação a de acordo com a CNN;
      execute a ação a;
      receba a recompensa r e o atualize o estado de jogo;
    fim
    se *é o turno do oponente* **então**
      oponente seleciona ação $a_{oponente}$;
      execute a ação $a_{oponente}$ e atualize o estado de jogo;
      $s_{novo} \leftarrow$ estado de jogo atual;
      se *o oponente ganhou o jogo* **então**
        $r \leftarrow$ recompensa por derrota;
        $s_{novo} \leftarrow null$;
      fim
      adicione ao *Experience Replay* a transição $(s_{antigo}, a_{agente}, s_{novo}, r)$;
    fim
    incrementa o *turno*;
  fim
  se *o agente ganhou o jogo* **então**
    $s_{novo} \leftarrow null$;
    adicione ao *Experience Replay* a transição $(s_{antigo}, a_{agente}, s_{novo}, r)$;
  fim
  selecione um *batch* aleatório de N transições do *Experience Replay*;
  calcule o *erro* deste *batch*;
  realize o *backpropagation* utilizando o *erro* e atualize a rede;
fim

3.5 Métodos de Treino

O processo de treinamento da IA foi realizado jogando contra 3 diferentes tipos de oponente. Essa pequena variedade permitiu uma melhor análise durante o desenvolvimento do programa já que foi possível observar o comportamento da rede em cada um dos casos. Os oponentes foram os seguintes:

- **Aleatório:** Este agente joga de forma aleatória em qualquer momento do jogo, não há nenhuma estratégia envolvida. Este oponente foi útil principalmente nas fases iniciais do desenvolvimento, já que o oponente não tenta impedir a IA de ganhar ele foi usado para verificar se a rede conseguia descobrir o objetivo do jogo. Durante a fase final da implementação ele também serviu para adicionar variabilidade de estados e verificar se o programa tinha o funcionamento desejado em situações simples mas que não foram vistas antes.
- **Heurítica:** Este oponente joga seguindo uma heurística baseada em circuitos elétricos. Esta estratégia é considerada forte e foi utilizada pelo programa chamado de *Hexy* [2] para ganhar a medalha de ouro

nas "5th Computer Olympiad"em 2000 [1]. O principal objetivo na maior parte do desenvolvimento foi conseguir vencer este oponente.

- ***Mixed*:** Por utilizar uma heurística puramente determinística o agente anterior sempre joga da mesma forma quado encontra o mesmo estado, isso pode permitir que haja uma forma de agir que seja 100% ótima já que o oponente sempre jogará da mesma forma. Devido a isso foi criado um oponente que joga de forma aleatória com chance x, sendo que por padrão $x = 0.2$. Este foi o oponente utilizado na maior parte dos testes finais pois mistura boa estratégia com certa imprevisibilidade

Além dos oponentes abordados acima,foram utilizadas duas formas de se observar o jogo para fins de aprendizado:

- ***Simple:*** Esta é a forma utilizada na maior parte da literatura sobre o assunto. A IA observa o jogo da forma comum, ou seja, realiza, observa e avalia somente o valor de suas própias jogadas. O pseudo-código da sessão anterior é referente à este método.
- ***Mirrored:*** Além de avaliar as própias jogadas a IA também examina as jogadas do oponente. Para realizar a análise das jogadas do adversário é necessário que o tabuleiro seja invertido, isso é, a posição de cada peça é transposta e tem as cores invertidas.

O primeiro método permite que uma partida seja jogada mais rapidamente pois são processadas menos jogadas a cada iteração. Por outro lado, o segundo método faz a IA aprender de forma mais veloz pois o tabuleiro é sempre observado por 2 pontos de vista. Uma análise mais detalhada deste fenômeno é discutida no próximo capítulo.

3.6 Heurística para o jogo de Hex

Apesar de ser possível treinar a rede do zero fazendo-a jogar contra ela mesma, esse processo demanda muito tempo e poder de processamento. Isso é especialmente relevante no início da execução quando a rede não tem muitos dados e a maior parte de seus movimentos ocorre de forma aleatória, nesta etapa muitas das posições obtidas são pouco relevantes para o aprendizado já que não seguem nenhuma estratégia. Por esse motivo o treinamento da rede foi realizado utilizando uma heurística comum no jogo de Hex que é baseada em resistências elétricas [2]. Esta heurística é considerada uma estratégia forte e foi utilizada pelo programa *Hexy* para ganhar o campeonato mundial em 2000 [1].

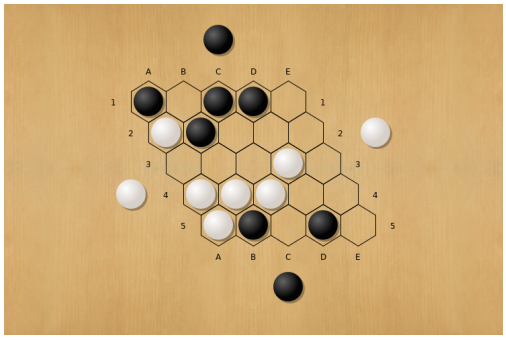
A heurística busca estimar o valor esperado de cada espaço no tabuleiro simulando um circuito elétrico onde dois lados opostos recebem uma carga e cada espaço representa uma resistência. O valor de cada espaço é calculado da mesma forma para os dois jogadores mas o processo é feito de forma separada sendo considerado 2 circuitos diferentes, um para o jogador de peças pretas e outro para o jogador de peças brancas.

No circuito do jogador de peças pretas as bordas norte e sul recebem a carga, cada espaço vazio tem resistência igual a 1, cada espaço com uma peça preta tem resistência 0 e por fim cada espaço com uma peça branca tem resistência $+\infty$.

No circuito do jogador de peças brancas as bordas que recebem a carga são a leste e a oeste. Apesar de os espaços vazios também terem resistência igual a 1, os espaços ocupados por uma peça tem o valor oposto ao circuito anterior, ou seja, peças brancas tem resistência 0 e peças pretas tem resistência $+\infty$.

Então, utilizando as *leis de Kirchoff* é calculada a resistência de cada circuito e como ela diminuiria se uma peça fosse colocada em um dado espaço. Quanto mais um espaço diminuir a resistência do sistema, melhor é a jogada, e quando a resistência vale 0, significa que o circuito foi fechado e o jogador que o finalizou ganhou. De forma semelhante se o circuito fica com resistência $+\infty$ significa que o oponente completou o circuito dele e venceu.

Durante a implementação desta heurística no programa o valor de cada jogada foi convertido para estar entre -1 e +1 de forma que quanto maior o valor melhor, neste caso -1 indica que não é possível colocar uma peça no espaço.



(a) *Exemplo de partida*

	A	B	C	D	E
1	-1.	0.59	-1.	-1.	0.80
2	-1.	-1.	0.68	0.86	1.
3	0.61	0.65	0.65	-1.	1.
4	-1.	-1.	-1.	0.82	0.83
5	-1.	-1.	0.66	-1.	0.66

(b) *Valor de cada jogada*

Figura 3.2: *Exemplo de heurística para o jogador de peças brancas*

3.7 Arquitetura da Rede

A CNN foi construída utilizando a biblioteca *pytorch* [11] do *python* pois ela permite realizar operações na rede neural e atualizá-la de forma simples e eficiente. As configurações finais de como a rede foi implementada estão listadas abaixo:

- 2 camadas convolucionais: A primeira com entrada de 6 canais e saída de 48. A segunda com entrada de 48 e saída de 384. O *kernel* para cada uma dessas camadas tem tamanho 2x2.
- Uma camada inteiramente conectada com saída igual ao número de ações possíveis. No caso do tabuleiro 5x5 a saída é um vetor de 25 valores.
- $\gamma = 0.8$
- *learning rate* $= 0.01$
- *momentum* $= 0.1$
- Chance de jogar aleatório $= 0.1$

3.8 Conversão de *numpy* para *pytorch*

Neste programa o tabuleiro foi implementado em *numpy* [6] e a rede neural em *pytorch* [11]. Devido a isto, toda vez que era preciso avaliar as ações do tabuleiro e o armazenar como um estado era necessário converte-lo para um tensor em *pytorch* para ser possível de o processar na rede. Foi testado implementar todo o sistema em *pytorch* para evitar realizar essa operação e com isso economizar tempo.

Quando o sistema foi inteiramente convertido para *pytorch* ele acabou ficando mais lento que anteriormente, isso ocorreu provavelmente pois a implementação das operações sobre tensores deve ser menos otimizada do que as em *numpy* para realizar operações em matrizes. Devido a este resultado o programa utiliza as duas bibliotecas de forma conjunta.

Capítulo 4

Resultados

4.1 Parâmetros gerais

A não ser que seja dito o contrário durante algum exemplo específico, os valores padrões para os testes foram:

- Número de partidas: 150000
- Tamanho da memória de *replay*: 300
- Tamanho do *batch* utilizado para otimizações: 200
- Atualização dos pesos da rede: 1 vez a cada 5 jogos
- Tabuleiro: 5x5 com 1 espaço de *padding*
- IA joga de peças brancas (começa a partida)
- Camada convolucional com entrada de 6 canais e saída de 48.
- Para testes com duas camadas convolucionais a entrada da segunda tem tamanho 48 e saída 384
- *Kernel* de tamanho 2 para cada camada convolucional
- Todos os pesos da rede foram inicializados com valor 0
- A recompensa por ganhar era de +100
- A recompensa por perder era -100 ponderado pelo número de jogadas até a derrota, isso fazia com que perder com mais jogadas fosse menos pior do que perder rapidamente.
- Qualquer outra jogada recebia recompensa 0.

4.2 Testes variando os parâmetros

Após ser verificado que o programa estava de fato funcionando da maneira desejada, foram realizados testes com diversas configurações com o objetivo de estimar os melhores valores para diferentes atributos. Foram avaliadas as variáveis com os valores a seguir:

- *learning rate* (lr): 0.01 e 0.1
- gamma: 0.8, 0.9 e 0.99
- Chance do agente jogar aleatório: 0.1, 0.25 e 0.4
- Número de camadas convolucionais: 1 e 2

Todas as combinações entre esses atributos foram testadas e abaixo é possível encontrar os 12 melhores resultados ordenados pela porcentagem de jogos ganhos. Estes experimentos foram realizados utilizando o treinamento *Mirrored* jogando contra um oponente *Mixed*.

classificação	lr	gamma	aleatoriedade	# de camadas	% de vitórias
1	0.01	0.8	0.1	2	69%
2	0.01	0.9	0.1	1	26%
3	0.1	0.9	0.1	1	14%
4	0.1	0.8	0.1	1	9%
5	0.1	0.8	0.25	1	9%
6	0.01	0.8	0.4	2	9%
7	0.1	0.9	0.25	1	9%
8	0.01	0.9	0.25	1	7%
9	0.01	0.8	0.25	1	4%
10	0.1	0.8	0.25	2	4%
11	0.01	0.8	0.1	1	4%
12	0.1	0.8	0.4	1	3%

Tabela 4.1: *Testes variando diferentes atributos*

Mesmo com uma grande gama de resultados é difícil explicar com precisão como cada variável influenciou o produto final, mesmo grandes projetos de *Machine Learning* geralmente precisam recorrer à tentativa e erro para estimar bons parâmetros. Apesar disto alguns fatores são dignos de nota.

Nenhum dos melhores resultados utilizou-se de gamma igual a 0.99, esse é o único atributo que nesta configuração deve com certeza ser evitado. O programa também parece funcionar melhor quando há uma chance menor dele jogar de forma aleatória, isso pode ser explicado pelo fato que quando se encontra uma estratégia boa, há menos chance de se desviar dela, ainda assim é importante que essa chance exista para poder-se explorar estratégias novas de vez em quando.

É esperado que mesmo quando rodado duas vezes com os mesmos parâmetros, o programa tenha um índice de vitórias diferente. Isso ocorre de forma natural devido a aleatoriedade com que o agente ou o oponente podem jogar, causando experiências distintas em cada execução. Por esse motivo, testes com resultados finais próximos poderiam trocar de colocação facilmente é ainda mais difícil avaliar quais variáveis fizeram a diferença nestes casos.

Mesmo levando em conta a variabilidade que pode ocorrer nos experimentos, é notável a diferença entre o índice de vitórias do primeiro e do segundo melhor teste. Por causa deste resultado, testes utilizando os atributos do melhor colocado foram estudados mais a fundo na próxima seção.

4.3 Análise de erro e número de vitórias

Para uma análise mais precisa foi realizado um novo experimento com a melhor combinação de atributos do teste anterior. Todas as configurações foram mantidas, somente o número de partidas foi aumentado para poder verificar com mais confiabilidade o resultado. Para este teste a IA jogou 300000 partidas, o dobro do teste anterior.

lr	gamma	aleatoriedade	# de camadas	% de vitórias
0.01	0.8	0.1	2	77%

Tabela 4.2: *Resultado do teste*

Figura 4.1: *Resultados após 300000 partidas.*

O resultado em relação à porcentagem de vitórias foi exatamente como se esperava, com mais simulações a IA pode treinar melhor e isso resultou em uma maior porcentagem de vitórias.

O primeiro gráfico da figura 4.1 mostra a quantidade de vitórias em cada 100 jogos. No início esse valor começa próximo de 0 mas ao decorrer da execução ele vai aumentado até estabilizar acima de 90. Esse resultado comprova que a IA tem um aumento de desempenho com o decorrer dos testes e que a maior parte de suas derrotas ocorreu no início do treinamento.

O segundo gráfico mostra o valor da função perda ao passar das execuções. No início ele é razoavelmente baixo pois todos os pesos foram inicializados com 0, apesar de ele prever os valores errados ele não erra por muito já que sempre devolve o mesmo resultado. Logo após esse período a rede começa a tentar "aprender", isso é, tenta avaliar as jogadas ao invés de devolver sempre o mesmo valor e isso faz o valor da função erro aumentar. Com um pouco mais de tempo a IA começa a prever com mais precisão o valor de suas jogadas e isso faz o erro cair. É possível notar que após o período inicial, em geral, o valor da função perda é inversamente proporcional ao número de partidas ganhas, isso significa que quanto menor o valor da *loss funciton*, mais precisa é a previsão e mais jogos são ganhos. Da mesma forma que o gráfico anterior, este também demonstra que a rede teve o comportamento esperado e um desempenho crescente com o passar do tempo.

4.4 Treinamento *Mirrored* x *Simple*

Um resultado interessante obtido durante os testes do programa é relacionado ao tipo de treinamento. Como explicado no capítulo anterior, é possível treinar a IA de duas formas, tentando aprender só as suas própias jogadas ou observando também as jogadas do oponente. A primeira delas é chamada de *Simple* e a outra de *Mirrored*.

A estratégia *Simple* é a utilizada nos artigos que foram lidos para o estudo de *deep Q-learning*. Apesar dela funcionar ela demorava muito até convergir para uma estratégia, isso era especialmente relevante quando se jogava contra a heurística. A estratégia *Mirrored* foi desenvolvida para tentar solucionar este problema; a ideia é que ao observar como o oponente ganhava, a IA iria aprender a jogar de forma similar. Apesar da *Mirrored* causar um pequeno impacto na velocidade do programa já que era necessário processar mais estados, a ideia teve sucesso e cumpriu o objetivo. Abaixo é possível ver resultados dos dois métodos jogando contra a heurística utilizando os atributos que foram discutidos na sessão 4.2.

número de partidas	método de treinamento	% de vitórias
150000	*Simple*	42%
150000	*Mirrored*	72%
300000	*Simple*	55%
300000	*Mirrored*	80%

Tabela 4.3: *Treinamento Simple e Mirrored*

4.5 Experimentos em tabuleiros maiores ou de outra cor

O programa também foi testado em tabuleiros maiores para comprovar que ele é escalável. Apesar de terem sido utilizados os melhores valores dos testes anteriores, não há como provar que eles são os mais adequados para estes novos cenários. Seria preciso rodar uma nova bateria de testes alterando os atributos se o objetivo fosse otimizar a IA nestes casos mas como o objetivo é só comprovar o funcionamento isto não foi realizado.

Este aumento no tamanho do modelo também causou grande impacto na performance, principalmente em tabuleiros 8x8 que apresentaram uma redução de velocidade de aproximadamente 6 vezes e por esse motivo os testes realizados com menos experimentos, apesar de isso também influenciar na precisão do resultado ainda era esperado que houvessem algumas vitórias.

tamanho do tabuleiro	% de vitórias
6x6	45%
7x7	2%
8x8	0.027%

Tabela 4.4: *Resultado em diferentes tabuleiros*

Estes testes evidenciam que mesmo sem nenhuma modificação para se adaptar aos novos tabuleiros a IA ainda obtém resultados. Quanto maior o tabuleiro maior a complexidade do espaço de estados e devido a isto o fato do programa conseguir um índice de vitória não 0 contra uma heurística forte demonstra que sua implementação é escalável e que com uma otimização individual para cada caso, melhores resultados podem ser obtidos.

Outro teste realizado foi o uso do programa usando as peças pretas ao invés das brancas. Isso implica que a IA jogará em segundo, uma grande desvantagem no jogo de Hex. Para este teste foram realizadas 300000 partidas utilizando o treinamento *Mirrored* jogando contra um oponente *Mixed*.

lr	gamma	aleatoriedade	# de camadas	% de vitórias
0.01	0.8	0.1	2	9%

Tabela 4.5: *Resultado de 300000 partidas utilizando as peças pretas*

Pode-se perceber que jogar com peças pretas piorou os resultados da IA porém ela ainda consegue ganhar algumas partidas. O gráfico abaixo demonstra que mesmo ela não ganhando frequentemente houve uma melhora da sua predição devido à diminuição do valor da função perda.

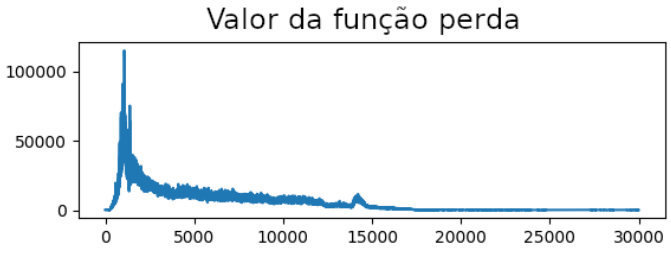


Figura 4.2: *Função perda quando se joga de peças pretas.*

Capítulo 5

Conclusão

No presente trabalho o objetivo principal foi criar um agente que utilizasse inteligência artificial para jogar Hex. Este jogo apesar de ter regras simples apresenta um grande número de estados e ações possíveis de forma que é infactível tentar prever todos eles por força bruta.

O primeiro passo durante o desenvolvimento foi criar um cliente para o jogo. Apesar das regras já terem sido definidas e a implementação ter sido baseada no Neurohex[7], o código foi escrito do zero de forma a se adequar melhor aos métodos propostos. Também foi necessário implementar a heurística para este cliente, sendo preciso uma pesquisa adicional para entender o funcionamento de circuitos elétricos e como se calcula a resistência neles.

Após o modelo do jogo ter sido criado, foi iniciada a implementação do *deep Q-learning*. Primeiramente foi construída uma CNN de classificação de imagens, depois sua função de erro foi alterada de forma a se adaptar para o contexto de Hex e buscar aproximar o valor da função Q. Também foi preciso criar um programa para realizar o treinamento da rede, ele foi responsável por controlar o método de treino utilizado, *Mirrored* e *Simple*, e o oponente contra a qual a IA jogava.

Um dos desafios enfrentados neste projeto foi que o agente só recebe a recompensa no final da execução. Isso significa que a maior parte das ações que ele realiza não recebe nenhum tipo de *feedback* pois a recompensa só é obtida quando é realizada uma ação vencedora para um dos jogadores. Um problema que pode acontecer é que a falta de ações com consequências imediatas impeça a IA de diferenciar ações boas de ruins principalmente no início de jogo. No fim, isso não se mostrou um obstáculo pois com execuções o suficientes a recompensa final conseguiu ser transmitida mesmo para o inicio do jogo. Outro fator que contribuiu para amenizar este problema foi o fato que, em geral, uma jogada no jogo de Hex ou traz muitas consequências boas ou muitas consequências ruins, não há muitas jogadas que tem um valor ambíguo.

Para poder avaliar a eficiência do programa foram testadas diversas configurações sendo que a melhor delas obteve mais de 70% de vitórias. Apesar deste índice depender de certas condições como um tabuleiro pequeno e a IA começando a partida, os resultados indicam que o programa é suficientemente genérico de forma que com mais testes e otimizações individuais, seria possível jogar em tabuleiros maiores ou com o oponente começando. Estes resultados cumprem o objetivo inicial de que é possível criar um agente razoavelmente forte para o jogo de Hex aplicando técnicas de *Deep Q-learning*.

Existem muitos algoritmos e estratégias que não foram implementados neste programa, *Dueling Double DQN* [15], *Prioritized Experience Replay* [14] e *Actor Critic Methods* [8] são alguns deles e seu uso seria interessante de ser estudado. Aprendizado por reforço e redes neurais são duas áreas populares atualmente e diversos estudos estão sendo publicados, espero que este projeto possa ser futuramente expandido e que tenha estimulado o interesse nestes campos da computação.

Referências Bibliográficas

[1] **Anshelevich(2000)** V. V. Anshelevich. Hexy Wins Hex Tournament. *The ICGA Journal*, 23(3): 181–184. Citado na pág. i, 2, 14

[2] **Anshelevich(2000)** Vadim V. Anshelevich. The game of hex: An automatic theorem proving approach to game programming. http://vanshel.com/Hexy/Publications/VAnshelevich-ARTINT.pdf, 2000. Citado na pág. i, 2, 13, 14

[3] **Fei-Fei Li(2017)** Serena Yeung Fei-Fei Li, Justin Johnson. Convolutional neural networks for visual recognition, lecture 14: Deep reinforcement learning, August 2017. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf. Citado na pág. 6

[4] **Gardner(1957)** Martin Gardner. Mathematical games. *Scientific American*, 197(1):145–150. Citado na pág. 1

[5] **Goodfellow** *et al.***(2016)** Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. MIT Press. http://www.deeplearningbook.org. Citado na pág. 6

[6] **Jones** *et al.***(2001–)** Eric Jones, Travis Oliphant, Pearu Peterson *et al.* SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed <today>]. Citado na pág. 2, 11, 15

[7] **Kenny Young(2016)** Gautham Vasan Kenny Young, Ryan Hayward. Neurohex: A deep q-learning hex agent. https://arXiv:1604.07097v2, 2016. Citado na pág. 2, 11, 21

[8] **Konda e Tsitsiklis(2000)** Vijay R. Konda e John N. Tsitsiklis. Actor-critic algorithms. Em S. A. Solla, T. K. Leen e K. Müller, editors, *Advances in Neural Information Processing Systems 12*, páginas 1008–1014. MIT Press. URL http://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf. Citado na pág. 21

[9] **Nash(1952)** John Nash. Some games and machines for playing them. https://www.rand.org/content/dam/rand/pubs/documents/2015/D1164.pdf, 1952. Citado na pág. 1

[10] **Nielsen(2015)** Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/. Citado na pág. 6, 7

[11] **Paszke** *et al.***(2017)** Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga e Adam Lerer. Automatic differentiation in pytorch. Em *NIPS-W*. Citado na pág. i, 2, 15

[12] **Reisch(1957)** Stefan Reisch. Hex is PSPACE-complete. *Acta Informatica*, 15(2):167–191. Citado na pág. 1

[13] **Russell(2010)** Stuart J. (Stuart Jonathan) Russell. *Artificial Intelligence: a modern approach*. Prentice Hall, 3nd edição. Citado na pág. 3

[14] **Tom Schaul(2016)** Ioannis Antonoglou David Silver Tom Schaul, John Quan. Prioritized experience replay. https://arxiv.org/abs/1511.05952, 2016. Citado na pág. 21

[15] **Ziyu Wang(2016)** Matteo Hessel Hado van Hasselt Marc Lanctot Nando de Freitas Ziyu Wang, Tom Schaul. Dueling network architectures for deep reinforcement learning. http://proceedings.mlr.press/v48/wangf16.pdf, 2016. Citado na pág. 21