UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA
DEPARTAMENTO DE MATEMÁTICA APLICADA

MARCELA MEDICINA FERREIRA

Algoritmos de Aprendizado por Reforço Aplicados a Jogos Digitais

Campinas
16/07/2022

MARCELA MEDICINA FERREIRA

Algoritmos de Aprendizado por Reforço Aplicados a Jogos Digitais

Monografia apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos para obtenção de créditos na disciplina Projeto Supervisionado, sob a orientação do(a) Prof. Dr. Paulo José da Silva e Silva.

Resumo

Este trabalho foi desenvolvido para a disciplina *MS777 - Projeto Supervisionado*, e teve como objetivo aplicar algoritmos de aprendizado por reforço ao jogo Pac-Man. Para isso, utilizamos uma técnica de Q-Learning aproximado, com a qual, combinamos as características do ambiente e seus respectivos pesos. Ao fim dos experimentos, pudemos otimizar o agente para obter altos índices de vitórias em diferentes layouts do jogo e pudemos concluir também quais as características mais importantes para sua vitória e os efeitos da normalização das características.

Abstract

This work was developed for the course *MS777 - Projeto Supervisionado*, and its objective was to apply reinforcement learning algorithms to the Pac-Man game. To do this, we used an approximate Q-Learning technique, with which, we combined the environment characteristics and their respective weights. At the end of the experiments, we were able to optimize the agent to obtain high victory rates in different layouts of the game and we could also conclude which features are the most important for its victory and the effects of the normalization of the features.

Conteúdo

1 Introdução

O desenvolvimento tecnológico e hardwares cada vez mais potentes, proporcionaram a popularização da inteligência artificial (IA), que está presente nos principais sistemas computacionais do mundo. Podemos defini-la como uma área de estudo que tem como objetivo emular algum tipo de comportamento inteligente em termos de processos computacionais. A partir dessa ideia, podemos também programar algoritmos que aprendem com dados. Este conceito de aprendizado com dados está associado subárea da inteligência artificial, conhecida como aprendizado de máquinas.

Os algoritmos de aprendizado de máquinas são divididos por alguns autores em três principais categorias: supervisionados, não supervisionados e baseados em reforço (principal foco desse projeto) [1]. Os algoritmos supervisionados, são aqueles que a solução é conhecida, ou seja, alimentamos os algoritmos com os dados de entrada e as respostas desejadas como saída (dados rotulados) [1]. Já no aprendizado não supervisionado, não conhecemos as soluções, não há rótulos, o modelo, por exemplo, pode aprender através de agrupamentos, associações, detecção de anomalias, entre outros [1].

Os algoritmos de aprendizado por reforço por sua vez, são diferentes. Neles, o sistema de aprendizado é chamado de agente. O agente observa o ambiente, seleciona e ações. E, a partir disso, recebe uma recompensa (ou uma penalidade, como é chamada uma recompensa com valor negativo) [1]. O agente deve aprender por si mesmo a tomar as melhores ações e a estratégia é chamada de política. A cada instante no tempo, o agente deve escolher a melhor ação de acordo com a politica e então, receberá uma recompensa. Este tipo de algoritmo é muito comum em aplicações de robótica e jogos, como por exemplo, o AlphaGo da DeepMind, que foi o primeiro modelo capaz de jogar o jogo de tabuleiro Go competindo, e ganhando, dos dois melhores jogadores humanos. Posteriormente ele evoluiu para versões mais poderosas como o AlphaZero, capaz de jogar xadrez e shogi [3] .

Neste projeto, em particular, tivemos como objetivo o estudo e desenvolvimento de algoritmos de aprendizado por reforço para jogar uma variação de Pac-Man. Nesse jogo, o personagem Pac-Man deve percorrer um labirinto 2D (que pode ter diferentes tamanhos e configurações) e comer todas as pílulas amarelas antes que seja pego por um fantasma.

Para poder desenvolver este agente de reforço, utilizamos um ambiente para o

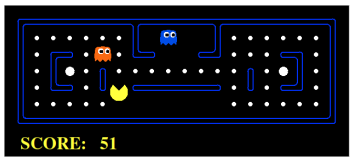
jogo escrito em Python 3 e disponível num repositório GitLab [*]. O agente foi implementado também em Python e utiliza o algoritmo Q-Learning aproximado para aprender. Além disso, no ambiente é possível obter diversas informações sobre o labirinto, fantasmas e pílulas. Mais detalhes sobre o ambiente, as regras do jogo e o algoritmo serão discutidos nas próximas seções.

2 Regras do Jogo e Ambiente Python

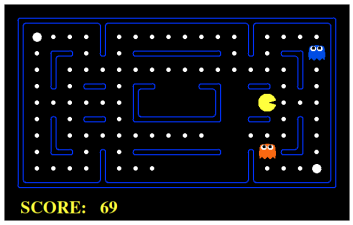
O Pac-Man é um jogo clássico e que possui muitas adaptações, algumas delas com formato totalmente distinto ao que utilizamos neste projeto, por isso, esta seção será dedicada a formalização do funcionamento do ambiente e das regras do jogo.

2.1 Layouts

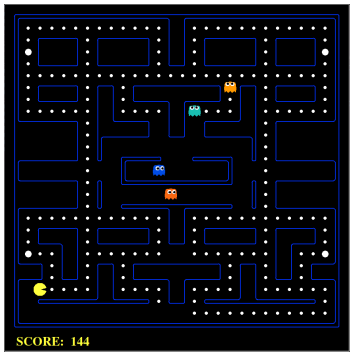
O layout é o ambiente em que os agentes — fantasmas e jogadores — interagem entre si durante a realização do jogo. Esse ambiente consiste em um labirinto 2D, onde os agentes andam de casa em casa, de forma que podem se movimentar para a direita, esquerda, cima, e baixo. Os labirintos são cercados por paredes que os impedem de movimentar livremente.



(a) Layout smallClassic.



(b) Layout mediumClassic.



(c) Layout originalClassic

Figura 1: Na imagem, vemos os três layouts, nos quais, treinamos nosso agente.

[*] https://gitlab.techniek.hva.nl/artificial-intelligence/pacman-contest-3/

2.2 Pílulas

As pílulas são divididas em dois tipos *power pills* e pílulas. As *power pills* são pílulas especiais que deixam todos os fantasmas "fracos", com a cor branca. Quando o Pac-Man se aproxima de um fantasma que está branco ele o mata, fazendo-o reaparecer na casa inicial agora sem ser branco. Comer um fantasma também concede muitos pontos para o Pac-Man.

O objetivo principal do jogo do Pac-Man é "comer" todas as pílulas do mapa. E, assim ganhar.

2.3 Fantasmas

Em um layout de Pac-Man, são encontrados diversos agentes inimigos, os fantasmas. Encostar em um fantasma implica em fim de jogo, exceto quando a *power pill* estiver ativa — nesse caso o Pac-Man "come" o fantasma, matando-o e fazendo-o reaparecer no mapa original, agora sem o efeito da *power pill*.

Nos layouts `smallClassic` e `mediumClassic` existem 2 fantasmas no mapa, enquanto que no `originalClassic` existem 4. Vale ressaltar, que os fantasmas ativos, aqueles que o Pac-Man não pode comer, são coloridos e os fantasmas assustados, aqueles que o Pac-Man pode comer após a *power pill*, são brancos.

2.3.1 Movimento

Os fantasmas se movem aleatoriamente com qualquer um dos possíveis movimentos válidos. Isso inclui a posição atual do Pac-Man, mas impede que fantasmas passem um por cima do outro ou atravessem paredes. Também é possível utilizar outros tipos de fantasma, com movimentação direcionada à atacar o Pac-Man e podendo fugir caso esteja sob o efeito da *power pill*. Esse tipo de fantasma não foi utilizado durante o projeto devido ao considerável aumento de complexidade, mas sua utilização é uma possível sugestão para trabalhos futuros.

2.4 Utilização do Ambiente

O ambiente é escrito em Python, e composto de um programa driver `pacman.py`, que lê as opções passadas pelos usuários e subsequentemente carrega os agentes, layouts e regras do jogo escolhidas pelo usuário. Para isso o ambiente é escrito de forma modular, com diversos layouts pré definidos. Para os testes utilizamos somente os layouts `smallClassic` (Figura 1a), `mediumClassic` (Figura 1b) e `originalClassic` (Figura 1c). O ambiente também contém outros agentes pré definidos, que não serão utilizados.

Para a aplicação do algoritmo no código implementamos a classe `RLAgents`, com o construtor e as funções `getAction(state)` e `final(state)`, que realizam, respectivamente, o cálculo do próximo movimento, dado o estado do jogo, e o um *checkup* final após o jogo acabar.

3 Aprendizagem por Reforço

O aprendizado por reforço é caracterizado como um sistema de aprendizado (chamado de agente) que aprende comportamentos por meio de tentativa e erro com base em suas interações com um ambiente dinâmico. Existem duas principais estratégias para resolvermos problemas de reforço, a primeira delas com algoritmos genéticos e programação genética, e a segunda utilizando métodos estatísticos e de programação dinâmica, que são nosso foco [2].

3.1 Modelo de Aprendizagem

Em um modelo básico de aprendizado por reforço há quatro elementos fundamentais: ações, estados políticas e recompensas. O agente está conectado ao ambiente através de ações a, e recebe informações através do estado atual s a partir de uma entrada i. Ao obter informações do estado, o agente decide uma ação futura a' e vai para um estado futuro s' de acordo uma determinada *política* [2].

A política é um conceito central para a aprendizagem, pois é através das políticas que os agentes decidem a melhor ação a ser tomada em um certo instante, em alguns casos ela pode ser uma função ou até mesmo uma tabela.

Os sinais de recompensa definem o principal objetivo do problema de aprendizagem. A cada ação, o agente recebe do ambiente um *feedback* chamado de recompensa. O objetivo do agente é maximizar a recompensa total que recebe a longo prazo. Essa recompensa pode ser positiva, que significa que o agente fez algo bom e que contribuiu com sua evolução no aprendizado ou negativa, que significa que das próximas vezes, ele deve evitar tal ação. A Figura 3.1, ilustra os conceitos relacionados a recompensa, ações e política [4].

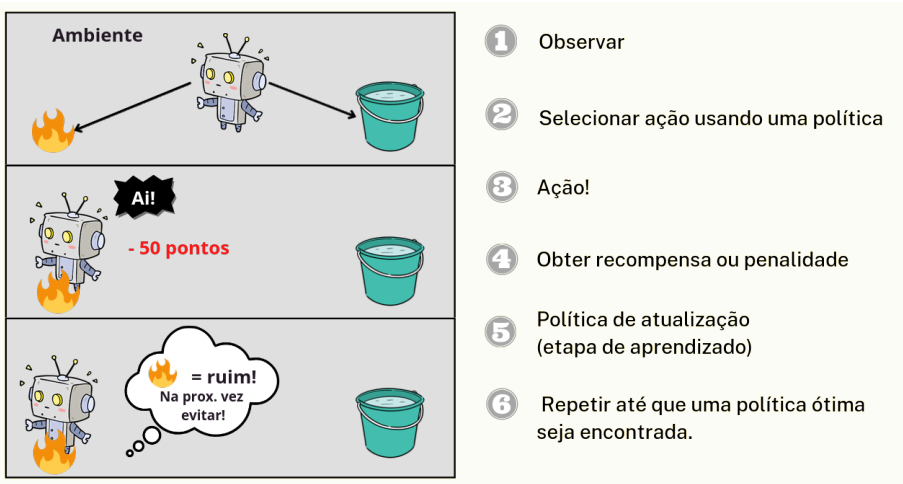


Figura 2: Ilustração do processo de aprendizagem por reforço.

3.2 Processos de Decisão Markovianos

O aprendizado por reforço está fortemente relacionado com os Processos de Decisão Markovianos, também chamado de MDP, sigla vinda do inglês *Markov Decision Processes*. Os problemas de reforço são modelados como MDPs, que consistem em um conjunto de estados S, um conjunto de ações A, uma função de recompensas $R : S \times A \longrightarrow \Re$ e uma função de estado de transição $T : S \times A \longrightarrow \Pi(S)$, na qual, todo membro $\Pi(S)$ é uma probabilidade de distribuição sobre o conjunto S, ou seja, a função é um mapeamento entre estados e probabilidades. E nós podemos escrever $T(s, a, s')$ para representar a probabilidade de fazer uma transição do estado s ao s' tomando uma ação a [2].

A função de transição T, especifica o próximo estado do ambiente a partir do estado atual e da ação atual do agente. A função de recompensa informa a recompensa instantânea esperada decorrência da última ação do agente, dizemos que um modelo é Markoviano apenas se as transições de estado forem independentes de qualquer estado

anterior do ambiente das ações anteriores do agente [2]. Vale ressaltar também, que no MDP é possível que existam infinitos estados, mas vamos trabalhar apenas com modelos finitos, até porque, o jogo tem uma quantidade finita de jogadas, e posições.

Note que, para o agente de reforço, é importante maximizar a recompensa. Para isso, busca-se encontrar os valores ótimos para cada estado afim de acumular as recompensas usando uma política ótima para as ações [2]. A política de decisão completa π está descrita na Equação 1, γ representa o desconto de cada recompensa num estado s e $\mathbb{E}$ a esperança da recompensa total de todos os estados sucessivos a partir de um estado atual s

$$V^*(s) = \max_{\pi} \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t r_t) \tag{1}$$

Essa função de valor ótimo é única e é definida como solução para as Equações 2 e 3. A Equação 2 é a Equação de Bellman [2]. A recompensa para s é instantânea, γ é um fator desconto para cada melhor ação futura disponível para o estado futuro s', e $T(s,a,s')$ a função de que mapeia a probabilidade de ir de um estado s para s' a partir de uma ação a. Em seguida, dada a função de valor ótimo, na Equação 3, nos ensina como podemos obter a política ótima. Tanto V^* quanto π^* são valores ótimos e queremos aproximá-los o melhor e mais rapidamente possível [2].

$$V^*(s) = \max_{a}(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')), \forall s \in S \tag{2}$$

$$\pi^*(s) = arg\max_{a}(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')) \tag{3}$$

Por fim, políticas ótimas também compartilham a mesma função de ação-valor ótima, chamada função Q, exibida abaixo [4]. A função Q representa o valor de um estado s e uma ação a a partir de uma política π.

$$Q^* = \max_{\pi}(Q^{\pi}(s,a)) \tag{4}$$

Conforme a maneira que a função Q for aproximada, ela pode ser usada de modo que seja uma combinação linear de características do ambiente (e não uma descrição completa do estado e uma tabela). Com todos esses conceitos em mente, podemos implementar

uma variação de um algoritmo simples para jogar as partidas chamado *Q-Learning*, conforme descrito na Seção de Implementação do Algoritmo.

4 Implementação do Algoritmo

Para implementarmos nosso agente, criamos a classe `RLAgent`. Essa classe tem como atributos a taxa de aprendizado, o épsilon de aleatoriedade, o desconto dado para recompensas futuras, o vetor de pesos, o estado inicial e o número de pílulas amarelas do mapa, também implementamos métodos para cômputo de recompensas, seleção da próxima ação, atualização nos pesos e cômputo do valor de Q. Vale ressaltar que Pac-Man não é um jogo trivial. Nele, temos um ambiente dinâmico, com características e estado do jogo mudando constantemente de acordo com as ações do agente, então pensar em Q como uma tabela (estratégia muito comum em jogos simples e com ambientes não-dinâmicos) não funcionaria. Por isso, é preciso usar uma descrição compacta, mesmo que aproximada, de Q. Em nossa implementação, optamos por uma representação afim baseada em características pré-selecionadas.

Portanto, com base nas informações obtidas através do ambiente, pudemos criar um vetor de características, conforme descrito na Seção de Seleção de Características. A partir disso, calculamos o produto interno do vetor de pesos com o de características com o objetivo de computar $Q(s,a)$. Para calcular a função Q, passamos como parâmetros o estado atual s, a ação a, e dentro dessa função chamamos um método da classe `RLAgent` que calcula o valor das características com este estado e ação. Utilizamos como base a Equação 5, sendo w_i o vetor com os pesos das características e $f_i(s,a)$ o vetor de características naquele instante de jogo.

$$Q(s,a;w) = \sum_{i=1}^{n} f_i(s,a)\, w_i \tag{5}$$

Tanto a função para calculo do valor de Q quanto a de atualização dos pesos foram baseadas no artigo "Reinforcement Learning in Pacman" de [**?**]. Utilizamos a Equação 6 para atualizar o peso de cada característica do ambiente, isso é feito a todo instante para cada ação que o agente toma, tentando sempre maximizar a recompensa.

$$w_i \leftarrow w_i + \alpha \cdot (r + \gamma \max_{a'}(Q(s',a') - Q(s,a)) \cdot f_i(s,a) \tag{6}$$

A função de recompensa é chamada a cada atualização dos pesos e foi implementada de forma a estimular o agente comer pílulas e explorar o mapa com o objetivo de encontrar as mais próximas, visto que ao comer uma pílula ele ganhará mais ponto e ao acabarem as pílulas do mapa, ele ganhará o jogo. A recompensa total de um estado é a diferença entre o pontuação (score) no estado atual s e no estado sucessor s' subtraindo-se cinco vezes o número de pílulas restantes no estado atual (getNumFood) pelo número total de pílulas existentes no início do mapa (nFood), como mostra na Equação 7.

$$recompensa = s'.score - s.score - 5 * (getNumFood)/nFood \tag{7}$$

Por fim, para determinar a próxima ação, verificamos quais são as possíveis ações que o Pac-Man pode tomar no estado atual. Após esta verificação, utilizamos uma estrutura do tipo *If-Else*, nela, chamamos o método *random* da biblioteca Python de mesmo nome. Este método, retorna um valor aleatório no intervalo de 0 a 1, se esse valor for menor que a taxa de aleatoriedade do atributo épsilon, retornamos uma ação aleatória. Caso contrário, a próxima ação é escolhida de forma ótima, a que resultaria no maior valor para $Q(s', a')$.

Todos os códigos estão disponíveis no github e serão compartilhados. Eles foram uma colaboração entre os autores e os alunos Daniel Gardin Gratti do bacharelado em matemática aplicada que contribuiu com a implementação do agente e computacional da UNICAMP e Frederico Bulhões de Souza Ribeiro do bacharelado em ciência da computação USP-SC que contribuiu na correção das falhas do ambiente.

5 Seleção de Características

O conjunto de dados e as características do jogo são obtidas durante as partidas. A partir da instância `GameState` podemos coletar informações do jogo como, por exemplo, posição dos fantasmas e do Pac-Man naquele estado, assim como todas as possíveis jogadas a partir daquele estado, e com isso, pudemos implementar funções em Python para as seguintes características:

- $bias$

  - Constante 1.

- $nearest\_food/maxDist$
  - Distância mínima a uma pílula, normalizada pela distância máxima a alguma posição no mapa;

- $5 \cdot next\_state\_num\_food/total\_food\_eaten$
  - Razão entre a quantidade de *pills* no mapa após realizar a ação e a quantidade de *pills* já comidas. Esse valor começa alto e depois diminui de acordo com a progressão do jogo, normalizado com o valor 5.

- $num\_food - next\_num\_food$
  - É avaliado como 1 caso haja uma *pill* ainda não comida na próxima casa, e 0 caso contrário

- $1 - num\_ghosts\_1\_step$ e $1 - num\_ghosts\_2\_step$ Conta o número de fantasmas a 1 e 2 movimentos de distância, respectivamente. Normalizado no 1.

- $(next\_score - cur\_score)/500 + 1$
  - A diferença entre o próxima pontuação e a atual, normalizados pela divisão por 500 e com soma do valor 1.

6 Resultados e Contribuições

6.1 Ambiente

Durante a implementação do algoritmo podemos perceber que o ambiente tinha alguma forma de vazamento de memória. Esse problema foi observado quando o programa era executado por muitas gerações (5000+), e foi possível observar que o gasto de memória crescia linearmente com o número de iterações, chegando a vários GiB.

Para resolver esse problema foram utilizados diversas ferramentas, como `pdb`, um debugger para Python; também foi utilizada a biblioteca `objgraph`, uma ferramenta

para observar o grafo dos objetos de Python. Com isso conseguimos encontrar a fonte do vazamento de memória.

Dentro do classe `GameState`, implementada no arquivo `pacman.py`, existe a variável estática `explored`, que supostamente é um *Set* que armazena todos os estados do jogo que já tiveram a função `getLegalActions` chamada. Na realidade, após o *debugging*, podemos verificar que na realidade esse *Set* armazena todos os estados que já tiveram a função `generateSuccessor` chamada, e que além disso, esse `Set` não é utilizado em parte alguma do código, apenas armazenando todos os estados sem nenhum uso.

Observando isso, verificamos que a função `generateSuccessor` é chamada diversas vezes em cada iteração do treinamento, e dessa forma acaba armazenando uma quantidade enorme de dados após um certo número de execuções, já que acaba guardando todos os estados de cada iteração do jogo. Removendo essa variável e todas as suas menções na classe `GameState` conseguimos acabar com o vazamento de memória e assim conseguimos executar o treinamento com um número maior de iterações (10000+).

Acreditamos que essa parte do código foi originalmente colocada para a implementação de algum tipo de *caching*, que acabou não sendo implementado.

6.2 Agente de Reforço

Após consolidar toda a implementação da classe `RLAgents`, fizemos testes com o agente no ambiente pequeno, médio e original. Realizamos 100, 1000 e 5000 jogos de treino e 500 de teste. Além disso, variamos também a taxa de aprendizado, como pode ser visto na Tabelas 1, 2 e 3: Ao total, tivemos cerca de 18 horas de computação, somando o tempo de todos os layouts. O layout $originalClassic$, foi particularmente demorado. Os testes foram realizados em computador com o processador AMD Ryzen 5700U de 8 cores e 16 threads com 8 GiB de memória RAM.

Tabela 1: Resultados para o layout smallClassic.

Taxa de aprendizado	Número de testes	Iterações de treino	Pontuação média de testes	Pontuação média de treino	Razão de vitórias	Tempo de execução (s)
0.001	500	100	571.0	269.0	0.63	199
0.01	500	100	700.4	568.0	0.75	133
0.1	500	100	-505.6	-500.0	0.00	74
0.001	500	1000	772.2	724.0	0.81	330
0.01	500	1000	680.9	647.0	0.74	327
0.1	500	1000	-476.3	-491.0	0.00	211
0.001	500	5000	595.0	716.0	0.65	1271
0.01	500	5000	617.2	639.0	0.68	1394
0.1	500	5000	547.1	245.0	0.61	1321

Tabela 2: Resultados para o layout mediumClassic.

Taxa de aprendizado	Número de testes	Iterações de treino	Pontuação média de testes	Pontuação média de treino	Razão de vitórias	Tempo de execução (s)
0.001	500	100	1460.0	1148.0	0.92	323
0.01	500	100	1484.9	1365.0	0.93	324
0.1	500	100	-475.1	-415.0	0.00	172
0.001	500	1000	1490.2	1337.0	0.93	828
0.01	500	1000	755.9	959.0	0.64	1979
0.1	500	1000	-541.1	-668.0	0.00	1230
0.001	500	5000	1434.0	1440.0	0.90	2399
0.01	500	5000	262.4	590.0	0.25	3853
0.1	500	5000	-541.0	-568.0	0.00	1908

Tabela 3: Resultados para o layout originalClassic.

Taxa de aprendizado	Número de testes	Iterações de treino	Pontuação média de testes	Pontuação média de treino	Razão de vitórias	Tempo de execução (s)
0.001	500	100	2642.6	2521.0	0.86	1557
0.01	500	100	2000.3	2406.0	0.60	2449
0.1	500	100	1086.0	-214.0	0.18	1154
0.001	500	1000	2515.1	2458.0	0.80	3362
0.01	500	1000	850.9	1446.0	0.04	3351
0.1	500	1000	-510.7	-313.0	0.00	4323
0.001	500	5000	1936.9	2252.0	0.58	10739
0.01	500	5000	762.8	1046.0	0.01	8273
0.1	500	5000	-616.4	-385.0	0.00	10655

Nós optamos por variar a taxa de aprendizado com o objetivo de encontrar experimentalmente a taxa mais adequada para nosso agente. Nas Tabelas 1, 2 e 3, podemos notar que a taxa de aprendizado $\alpha = 0.1$ traz de longe os piores resultados.

Conforme ao que esperávamos, se aumentarmos demais a taxa de aprendizado, o agente não consegue convergir corretamente e seus valores convergem para infinito. Notamos dois comportamentos comuns nesse caso: o Pac-Man se choca rapidamente com os fantasmas, já que começa a fazer movimentos circulares, indo e voltando para a mesma posição, ou então, acaba perdendo a partida porque atingiu o tempo limite de jogo.

Nos casos com a taxa de aprendizado maior, $\alpha = 0.01$ e $\alpha = 0.001$, conseguimos resultados significativamente melhores, essa diferença pode ser percebida principalmente no mapa $originalClassic$.

No que diz respeito ao número de partidas, podemos notar uma espécie de sobreajuste (do inglês, 'overfitting'). Isso pode ser observado porque a partir de uma certa quantidade de treinos, utilizando os mesmos parâmetros, o agente começa a piorar seu desempenho, o que está bem ilustrado na Tabela 3.

Ainda falando em 'overfitting', podemos ver comparando as Tabelas 1, 2 e 3, que a que apresenta os melhores resultados é a do mapa $mediumClassic$, isso se deve ao fato de

que os parâmetros foram ajustados nesse mapa e depois tentamos generalizar o agente para mapas de outros tamanhos. Possivelmente, as contantes que utilizamos nas características descritas na Seção de Seleção de Características poderiam ter sido diferentes se tivéssemos usados outros mapas como padrão.

Finalmente, a porcentagem de vitórias e a pontuação média das partidas de treino e testes foram importantes durante as etapas de escrita dos códigos Python, já que foram com base nesses resultados que atualizávamos nossas funções de obtenção de características e também de cálculo de recompensas.

7 Conclusão

Neste trabalho apresentamos uma implementação do algoritmo de $Q-Learning$ aproximado para o jogo Pac-Man, testamos o nosso agente em diversos layouts e alteramos as taxas de aprendizado e as características utilizadas para o aprendizado do agente. Para as variações na taxa de aprendizado, observamos que o conforme aumentamos o valor, o agente começa a perder mais jogos, isso porque a função Q não converge corretamente e seus valores convergem para infinito, a partir daí, o agente perde todas as partidas. Com um valor menor para taxa de aprendizado, em geral, o aprendizado é mais demorado, mas em compensação temos uma taxa de vitórias muito superior.

Em relação as características, pudemos observar que a normalização dos valores afeta fortemente os resultados do agente. Se não normalizamos os valores da características, os pesos crescem muito rápido e também convergem para infinito. Notamos também, que é de fundamental importância recompensar o agente para cada pílula comida, dessa forma, o agente ativamente procura por elas e evitamos assim o problema dele ficar parado no mapa ou seguir alguma outra direção que não contribui para sua vitória. Para o agente seguir a direção da pílula mais próxima, notamos que função de distância impacta diretamente em seus resultados, em nossas análises, pudemos observar que não basta apenas calcular qual é a pílula geometricamente mais próxima, mas sim temos que levar em conta outros fatores como se existe ou não algum tipo de parede no mapa que impede a passagem do Pac-Man.

Por fim, concluímos que as características mais importantes são a distância da pílula mais próxima normalizada pela distância máxima e a diferença do número de pílulas

no estado atual para o estado futuro. E encerramos nosso projeto com bons resultados médios para os três mapas explorados, com os melhores resultados para o mapa $mediumClassic$, conforme discutido na Seção de Resultados e contribuições.

Referências

[1] A. Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc., 1st edition, 2017.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[4] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA, 2018.