Trabalho Prático 2
MDPs e Aprendizado por Reforço

Arthur Souto Lima - 2018055113

[1]Departamento de Ciência da Computação – Instituto de Ciências Exatas
Universidade Federal de Minas Gerais (UFMG)
Av. Pres. Antônio Carlos, 6627 - Pampulha, Belo Horizonte - MG, 31270-901

arthursl@ufmg.br

1. Introdução

O jogo Pac-Man é um destacado representantes dos video-games, criado na década de 80 para os fliperamas pela empresa Namco. Nele, o jogador controla o personagem Pac-Man em diversas fases, nas quais ele deve comer todos os pontos do labirinto, enquanto evita os fantasmas, seus inimigos. Há alguns pontos maiores que, se comidos, permitem que o jogador consiga comer os fantasmas para conseguir mais pontos.

Seguindo essa mecânica, é possível derivar tarefas de aprendizado de inteligência artificial e, dados os objetivos do jogo, utilizar heurísticas baseadas em aprendizado por reforço. Portanto, o alvo deste trabalho prático é implementar alguns algoritmos para que o agente seja capaz de jogar o game após um treinamento.

Tal como no trabalho prático anterior, a base de trabalho é a implementação do jogo Pac-Man em Python 2 de um projeto do curso de Introdução à IA da Universidade de Berkley [1]. Este trabalho consiste em finalizar a implementação dos algoritmos Value Iteration, Q-Learning e sua variante aproximada em seus respectivos agentes. Esses então podem ser treinados e, com isso, conseguirão jogar o Pac-Man.

2. Algoritmos

2.1. Value Iteration

Com base no que vimos em aula, o algoritmo Value Iteration se baseia em Processos de Decisão de Markov (ou, em inglês, Markov Decision Process com o acrônimo MDP) e, mais especificamente, na equação de Bellman:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, a, s') V^{\pi}(s')$$

em que $R(s)$ é a recompensa recebida no estado s, γ é o fator de desconto, o qual penaliza recompensas futuras, $T(s, a, s')$ é a probabilidade de, estando no estado s, fazer a ação a e ir para o estado s', segundo o MDP [Chaimowicz 2020b, Russell and Norvig 2020a].

Nesse caso, a partir do MDP, via programação dinâmica, usamos essa equação como princípio de atualização dos pesos das ações para chegarmos na política ótima, normalmente via uma série de iterações. A grande propriedade teórica deste algoritmo é que ele eventualmente converge para um conjunto único de soluções da equação de Bellman [Russell and Norvig 2020a].

[1]http://ai.berkeley.edu/reinforcement.html

2.2. Q-Learning

Um dos grandes revezes do Value Iteration é a necessidade de possuir o modelo de Markov de início [Chaimowicz 2020a], o que normalmente não está disponível. Para contornar isso, no Q-Learning, o agente aprenderá o MDP interagindo com o ambiente. Para tanto, trabalharemos com o os Q-Valores ($Q(s, a)$) para cada estado e ação. Eles serão inicializados como 0 e, a cada etapa do algoritmo, no método *update*, eles serão atualizados segundo a equação:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a) + \gamma V(s') - Q(s, a))$$

em que α é a taxa de aprendizado, $R(s, a)$ é a recompensa do estado s com a ação a, γ é o desconto (discount), $V(s')$ é a função de valor do estado de destino [Chaimowicz 2020a, Russell and Norvig 2020b].

A função de valor do destino pode ser calculada com a fórmula [Chaimowicz 2020a]:

$$V(s) = \max Q(s, a)$$

Assim, o agente explora o ambiente, selecionando ações e observando as recompensas, bem como os estados seguintes, e, através dessa interação, aprende as recompensas a fim de encontrar uma política ótima.

Entretanto, como ressaltado pela especificação, para que o algoritmo de fato encontre políticas ótimas, ele deve, de alguma forma, explorar outros caminhos, que não o ótimo atual disponível. Pode-se usar o método do ϵ-greedy, a qual define que, durante a exploração do ambiente, com uma probabilidade ϵ, o agente escolhe uma ação aleatória e não a melhor disponível.

2.3. Q-Learning Aproximado

Como comentado pela especificação, o Q-Learning padrão não consegue encontrar boas políticas para problemas mais complexos, como é o caso do labirinto clássico do Pac-Man. Para essa tarefa, precisamos de uma versão modificada, o Q-Learning Aproximado, o qual lidará MDPs aproximados [Wiedenbeck 2016]. Nesse algoritmo, temos um conjunto de features (características) cuja combinação linear com um conjunto de pesos nos fornece as recompensas para cada estado. Assim, o algoritmo deve, para cada estado, fazer uma atualização considerando sua representação nas features.

Nesse algoritmo, usaremos para calcular os Q-valores $Q(s, a)$:

$$Q(s, a) = \sum_{i}^{n} f_i(s, a) w_i$$

em que w_i é o peso da característica i e $f_i(s, a)$ é o valor de tal característica para o estado e ação requeridos [Wiedenbeck 2016].

Para atualização dos pesos usaremos a relação:

$$w_i \leftarrow w_i + \alpha(R(s,a) + \gamma V(s')) \cdot f_i(s,a)$$

em que w_i é o peso da característica i, α é a taxa de aprendizado, $R(s,a)$ é a recompensa do estado s com a ação a, γ é o desconto (discount), $V(s')$ é a função de valor do estado de destino e $f_i(s,a)$ é o valor da característica i para o estado e ação atuais [Wiedenbeck 2016].

Com essa estratégia, a tabela de Q-valores reduz seu tamanho significativamente, além de permitir generalização em estados não visitados, o que torna mais robusto tomar decisões parecidas em estados parecidos [Wiedenbeck 2016]. Apesar disso, temos que escolher quais features usar no algoritmo, o que também pode impactar na precisão da função de recompensa [Wiedenbeck 2016].

3. Implementação

3.1. Passo 1

O primeiro passo deste trabalho prático não prevê implementação, apenas processos de leitura e familiarização com o código fornecido.

3.2. Passo 2

Para implementação do algoritmo Value Iteration, seguiu-se o que vimos em aula [Chaimowicz 2020b] e os pseudocódigos presentes no livro-texto [Russell and Norvig 2020a], bem como a equação de Bellman. Felizmente, a especificação já guiava bastante o que deveria ser implementado, elencando as equações de cada passo. A maior dificuldade foi descobrir quais métodos das classes relacionadas estavam disponíveis para auxiliar nessas tarefas.

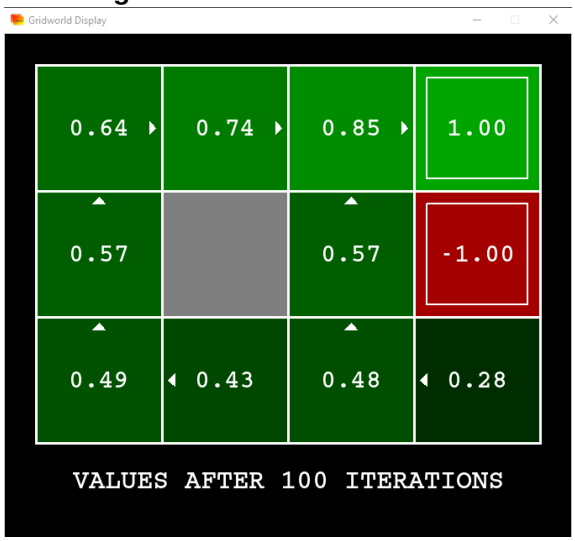
Desse modo, primeiro completou-se o método *computeQValueFromValues* para obtermos o valor de $Q(s,a)$. Em seguida, o método *computeActionFromValues* que utiliza esse resultado para obtermos $\pi(s)$, ou seja, a melhor ação a ser tomada num determinado estado. Apesar de um comportamento parecido, no método construtor dessa classe *ValueIterationAgent* implementamos um procedimento semelhante com *computeQValueFromValues* para inicializar a tabela dos valores.

Os algoritmos não são demasiado grandes e foram corretamente implementados, balizados pelo *autograder* com sucesso. Além disso, na Figura 1, é possível ver a função de valor obtida, além de alguns caminhamentos, seguindo o comando proposto na especificação.

3.3. Passo 3

A implementação desse passo é bastante simples, bastando alterar uma constante na função *passo3* do arquivo *analysis.py*. Nesse exemplo, "cair" significa que o agente desviou de seu caminho. Desse modo, para evitar que ele caia, é necessário diminuir o valor do parâmetro de ruído para próximo de zero para que ele não intente realizar outros movimentos para fora da ponte.

Figura 1. Resultado do Passo 2



3.4. Passo 4

Seguindo as ideias do Passo anterior, temos que modificar os hiperparâmetros nas funções *passo4** também do arquivo *analysis.py*. Após alguns experimentos os resultados para o parâmetro de ruído (noise), desconto (discount, normalmente representado pelo γ) e o living reward são:

(a) Ruído baixo, γ baixo, living reward baixo
(b) Ruído mediano, γ baixo, living reward mediano-alto
(c) Ruído baixo, γ alto. Basicamente a letra (a), mas com γ alto.
(d) Ruído alto, γ alto. Basicamente a letra (b), mas com γ alto.
(e) Living Reward positivo e bastante alto (maior que a maior recompensa disponível)

Apenas para clarificar, um γ alto indica que recompensas futuras são relevantes, ou seja, são pouco descontadas/penalizadas. Na letra (e), os outros dois parâmetros não têm tanta relevância para o objetivo dela.

3.5. Passo 5

Tal como no Value Iteration, a implementação do Q-Learning no trabalho foi bastante guiada pelas orientações da especificação. Não houve necessidade de modificações no construtor da classe *QLearningAgent*. No método *getQValue*, foi necessário apenas um simples acesso à estrutura de dados *qValues*. O grande foco são os três outros métodos pedidos.

Tanto o método *computeValueFromQValues* quanto *computeActionFromQValues* usam estratégias semelhantes para computar o máximo *qValue*, mas um retorna a ação e o outro o valor máximo. Seguimos as ideias do pseudocódigo visto em aula e no livro [Chaimowicz 2020b, Russell and Norvig 2020b].

Por fim, o método *update* usa a equação proposta nas orientações para atualizar o atributo *qValues*. A maior adversidade nessa implementação foi descobrir quais atributos e quais métodos acessar para obter cada parâmetro da equação.

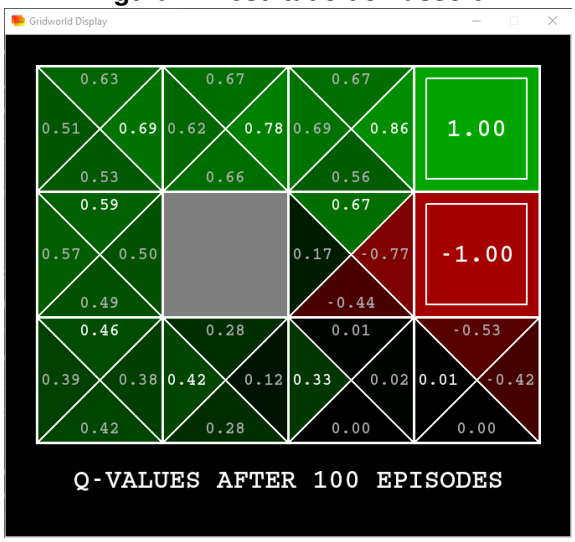
A base do Q-Learning implementada neste passo foi avalizada com sucesso pelo *autograder* fornecido no projeto.

3.6. Passo 6

Este passo se concentra em implementar a heurística ϵ-greedy no algoritmo do Q-Learning feito anteriormente. O foco foi o método *getAction* e todas as dicas, tanto da documentação do código quanto da especificação, foram essenciais para indicar o caminho correto para o algoritmo implementado.

Novamente, a solução projetada foi considerada correta pelo *autograder* e, além disso, podemos ver o desempenho do agente usando esse algoritmo e estratégia com o comando sugerido pela especificação na Figura 2. Nota-se uma semelhança com o resultado do algoritmo Value Iteration, constante na Figura 1.

Figura 2. Resultado do Passo 6



3.7. Passo 7

O passo 7 é o último relacionado ao arquivo *analysis.py*, em que temos que modificar hiperparâmetros do agente visando atingir um objetivo. Neste passo, a meta é atravessar a mesma ponte proposta no passo 3, contudo, agora, com o Q-Learning.

Após diversos experimentos, decidiu-se por classificar esse objetivo como impossível, o que foi corretamente corrigido pelo *autograder*. Em linhas gerais, a ideia é que precisamos aumentar o ϵ para que o agente explore o restante da ponte, caso contrário, ele ficará apenas voltando para o início. Nos experimentos, com $\epsilon < 0.5$ isso ocorria. Alterações na taxa de aprendizado não apresentaram grandes impactos visando esse objetivo.

A grande conclusão deste passo é que, de fato, o agente consegue atravessar a ponte. Contudo, isso fica a cargo do acaso, ou seja, não se pode garantir que ele assim o faça em todas as execuções, apenas com base nesses parâmetros. Por exemplo, durante os experimentos, algumas vezes o agente atravessou, mas numa execução seguinte, com exatamente os mesmos parâmetros, ele não o fez. Por isso essa tarefa deve ser classificada como impossível.

3.8. Passo 8

O passo 8 não inclui implementação, apenas testes dos algoritmos feitos. Os resultados foram satisfatórios, corrigidos pelo *autograder* e atenderam aos 80% de precisão referidos pela especificação.

3.9. Passo 9

Para conclusão do passo 9, foi necessário completar a implementação da classe *ApproximateQAgent* com os dois métodos: *getQValue* e *update*. Seguindo as equações e conceitos tanto na especificação quanto em [Wiedenbeck 2016]. O *getQValue* basicamente faz um produto escalar entre as features e os pesos. O *update* usa a fórmula constante no enunciado com uma leve modificação apresentada em [Wiedenbeck 2016].

O resultado foi avalizado pelo *autograder* e, visualmente, pôde-se ver o agente completando o nível clássico do Pac-Man com sucesso, após treinamento.

4. Discussão

Os resultados obtidos, tendo em vista os algoritmos e suas propriedades, condizem com o apresentado, bem como com o visto em aula e na literatura. Nenhuma situação inusitada ou fora do esperado foi observada.

No Passo 2, usamos a dinâmica do MDP para obter, a priori, sem uma exploração do ambiente pelo agente, a função de valor de cada estado. O agente não interage com o ambiente, diferentemente do Q-Learning dos Passos 5 e 6. Nesse caso, o Q-Learning de fato não se vale do MDP completo como foi o caso do Value Iteration no Passo 2 e, durante o treinamento, o agente interage com o ambiente e atualiza os pesos e recompensas esperadas de cada estado.

No Passo 9, o Q-Leaning Aproximado de fato conseguiu ter sucesso no labirinto clássico, feito que não era possível com o Q-Learning padrão implementado em passos anteriores, devido a sua capacidade de lidar com problemas maiores e/ou mais complexos.

4.1. Hiperparâmetros

Os algoritmos implementados possuem uma série de hiperparâmetros que os balizam nas execuções. Em diversas etapas do trabalho nos foi requisitado experimentar com esses valores a fim de atingir objetivos específicos. Com isso, foi possível aferir os impactos práticos de cada um deles, à luz, naturalmente, de suas propriedades teóricas previstas.

4.1.1. Value Iteration

No Value Iteration, temos três parâmetros importantes: ruído, desconto e living reward. O ruído (noise) define o quanto o agente ”erra” ou desvia do caminho habitual, favorecendo exploração de outras políticas. Por outro lado, o desconto (discount, usualmente referido como γ) define o quanto recompensas futuras são relevantes. Quanto maior o γ mais importantes serão essas recompensas, ou seja, elas terão um menor desconto ou uma menor penalização. Com γ alto, favorece-se exploração de caminhos mais longos.

Finalmente, o parâmetro living reward indica a disposição do agente de procurar caminhos mais longos. Com um valor negativo, o agente tende a preferir caminhos mais curtos para minimizar as perdas de caminhos longos. Com valor positivo, abre-se uma possibilidade para que caminhos mais longos sejam visitados. Contudo, se ele for muito grande, pode-se criar um caso extremo em que o agente prefira continuar caminhando a ir para um estado terminal.

4.1.2. Q-Learning

Já no caso do algoritmo do Q-Learning, temos dois hiperparâmetros importantes: a taxa de aprendizado e o ϵ. A primeira controla o quão fortes serão as mudanças feitas durante a atualização dos Q-valores, no caso, no método *update*. O segundo, ϵ, define com qual probabilidade o algoritmo investigará uma ação não-ótima aleatória durante cada passo da exploração. Esse é consideravelmente relevante para que o método do Q-Learning consiga, de fato, obter uma melhor política e não fique preso à política ótima atual.

5. Conclusão

Neste trabalho, a tarefa foi implementar o Value Iteration e o Q-Learning para treinar um agente que conseguisse jogar, com sucesso, o labirinto do Pac-Man. No contexto de aprendizado por reforço visto em aula, aprendeu-se o funcionamento dos algoritmos, bem como suas diferenças. Os algoritmos foram implementados em várias etapas, incluindo algumas que exploravam os impactos de seus hiperparâmetros. Ao final, atingidos esses objetivos, obteve-se um agente que, de fato, era capaz de lidar inclusive com o labirinto clássico do jogo.

Esse projeto foi de notável relevância para a consolidação dos algoritmos vistos em aula, no tocante, principalmente, ao aprendizado por reforço. O passo-a-passo indicado forneceu bases para enfoque e compreensão das diversas particularidades dos algoritmos, bem como de sua implementação.

Tal como no primeiro trabalho prático, houve alguns percalços ao longo da realização desta atividade. De forma semelhante à primeira atividade, no que diz respeito à primeira adaptação ao sistema, com suas interfaces e estruturas, houve uma dificuldade inicial. Felizmente, dada a experiência com o trabalho anterior, nem tudo era completamente novo. Ademais, tanto o bem documentado código quanto a clareza e as orientações da especificação ampararam para superar esse obstáculo prontamente. Outra dificuldade encontrada neste projeto foi um estranhamento inicial com a versão da linguagem, Python 2, ao invés da sua versão 3, mais comumente utilizada em outros contextos. Apesar disso, com o decorrer da implementação, foi possivel se acostumar com as poucas diferenças entre essas duas variedades.

Novamente, também como no primeiro trabalho prático, uma vantagem do sistema fornecido foi o *autograder*. Primeiramente, esse módulo permite um fluxo de incrementos graduais centrados em verificação automática, típico do TDD (Test-Driven Development). Além disso, há uma maior segurança na implementação feita, já que temos um feedback instantâneo, ainda que não necessariamente definitivo, dos algoritmos em diversos casos de teste, antes da entrega final do trabalho.

Finalmente, convém ressaltar que esse trabalho foi bastante agradável de se realizar: havia um contexto, um problema claro, uma excelente especificação guiando e delimitando os objetivos, um sistema estável e consistente, pronto para receber e avaliar o desenvolvimento da tarefa. Ademais, como um motivador adicional, trabalha-se num sistema semelhante ao do primeiro trabalho prático, quase que numa continuação do projeto. Além disso, ao final, vê-se o admirável resultado: um agente capaz de jogar eficientemente o labirinto do Pac-Man, tarefa essa que nem sei se, de fato, tenho as habilidades para ser tão eficaz.

References

Chaimowicz, L. (2020a). Aprendizado por reforço. Acesso em: 04 fev 2022.

Chaimowicz, L. (2020b). Mdps: Markov decision processes. Acesso em: 04 fev 2022.

Russell, S. J. and Norvig, P. (2020a). Making complex decisions. In *Artificial Intelligence: A Modern Approach*, chapter 16, pages 552–588. Pearson.

Russell, S. J. and Norvig, P. (2020b). Reinforcement learning. In *Artificial Intelligence: A Modern Approach*, chapter 23, pages 840–873. Pearson.

Wiedenbeck, B. (2016). Approximate q-learning. [2]. Acesso em: 04 fev 2022.

[2] `https://www.cs.swarthmore.edu/~bryce/cs63/s16/slides/3-25_approximate_Q-learning.pdf/`