Reinforcement Learning aplicado a agentes

André Pereira (pg38923)
Carlos Lemos (pg38410)
João Barreira (a73831)
Rafael Costa (a61799)

Maio 2019

Universidade do Minho
Escola de Engenharia

Trabalho Prático 2
Sistemas Inteligentes – Computação Natural
MEI / MIEI – Universidade do Minho

Conteúdo

1 Introdução

Neste trabalho prático, proposto pela Unidade Curricular de Computação Natural, serão abordados diferentes aspetos cujo objetivo é a preparação de um modelo inteligente e respetivo procedimento da sua análise e validação de performance. Após esta abordagem, o último aspeto deste projeto será a realização de uma otimização de métodos de pesquisa de política aplicados no modelo inteligente desenvolvido, que terá sido implementada usando técnicas de *Reinforcement Learning*.

De forma a desenvolver este modelo inteligente, foi criado um agente informático, em *Python*, capaz de aprender a jogar e ganhar o jogo *Pacman* através da implementação de um algoritmo em *Reinforcement Learning*.

O desenvolvimento do algoritmo em *RL* está divido em duas fases. Na primeira fase consta o treino do modelo, sendo aqui que o agente inteligente adquire o conhecimento sobre os valores das posições e das suas ações. A segunda e maior fase é caracterizada pela implementação das técnicas desenvolvidas, e é onde o agente tenta efetivamente derrotar o adversário, explorando a política aprendida sobre este jogo.

O referido modelo foi otimizado através da adoção de outros parâmetros de aprendizagem, usando para isso o algoritmo de *Grid Search* e de *Particle Swarm Optimization*.

Por último, de forma a enriquecer o projeto, foi feita uma análise de treino/teste do algoritmo *RL* desenvolvido para um mapa do jogo de tamanho médio. Assim foi realizada uma análise e comparação dos resultados provenientes da utilização de diferentes mapas/ambientes, listada o conjunto de limitações provenientes de mais estados e acções e, por último, definido um conjunto de medidas necessárias para a otimização do algoritmo de *RL*.

2 Resolução do jogo *Pacman* através de *Reinforcement Learning*

Como principal objetivo do presente projeto, desenvolveu-se um algoritmo de *reinforcement learning* que permitisse a resolução do famoso jogo *Pacman*. Para isso foi recebido um esqueleto de um projeto desenvolvido em *Python* que continha inúmeros ficheiros de código que permitiam a ilustração do jogo em tabuleiros de diferentes dimensões, funções auxiliares para o algoritmo a desenvolver, entre outros aspetos. Dentro desse conjunto de ficheiros, damos destaque aos ficheiros *qlearningAgents.py*, *util.py* e *pacman.py*. O primeiro ficheiro diz respeito ao algoritmo a implementar e já apresentava previamente um conjunto de funções com algumas instruções de código a ser desenvolvido para esse algoritmo. O ficheiro *util.py* contém um conjunto de funções auxiliares ao algoritmo e o último ficheiro diz respeito ao arranque do programa onde se pode efetivamente testar o algoritmo desenvolvido. Antes de se iniciar ao processo de contrução do algoritmo, efetuou-se uma pesquisa na documentação presente em `http://ai.berkeley.edu/reinforcement.html`. Aqui verificou-se que a

classe *QLearningAgente* presente no ficheiro *qlearningAgents.py* recebia, entre outros, três parâmetros essenciais, sendo estes: *alpha*, *gamma* e *epsilon*. Os primeiros dois parâmetros dizem respeito à própria equação para encontrar um novo *Q Value*. O terceiro parâmetro diz respeito ao conceito de exploração do ambiente em *reinforcement learning*. No processo de aprendizagem deste tipo, um agente pode executar uma determinada ação sem testar outras possibilidades, caso essa ação maximize o valor númerico de recompensa. No entanto isto provoca uma controvérsia no processo de aprendizagem em *reinforcement learning*, já que o agente não explora o ambiente em que está inserido à procura de novas ações que possam contribuir para um determinado objetivo. Este terceiro parâmetro indica um valor que corresponde a uma probabilidade de a cada estado, o agente explorar o ambiente com uma nova ação ao invés de verificar quais as melhores ações já efetuadas. Tal técnica é conhecida como *epsilon greedy*. Posto isto, o objetivo foi o de completar um conjunto de funções apresentadas de seguida:

getQValue: recebe como parâmetros um estado e uma ação. Através deste par devolve-se o *Q Value* correspondente ou o valor 0.0 caso esse estado ainda não tenha sido avaliado.

computeValueFromQValue: devolve o melhor *Q Value* a partir de todas as *legal actions* de um determinado estado. Caso não existam tais ações é devolvido o valor 0.0.

computeActionFromQValues: devolve a melhor ação a ser tomada num determinado estado de acordo com os seus *Q Values*. Se duas ou mais ações apresentarem o melhor *Q Value*, é usada a função *random.choice* para escolher aleatoriamente uma dessas ações. Caso não haja nenhuma ação até ao momento é devolvido o valor *None*.

getAction: esta função diz respeito ao parâmetro *epsilon*. Este valor é passado como parâmetro à função *flipCoin* implementada previamente no ficheiro *util.py*. Caso a probabilidade seja satisfeita é tomada uma ação aleatória de modo a explorar o ambiente. Caso contrário devolve-se uma ação através da função *computeActionFromQValues*.

update: função de atualização do *Q Value*. Aqui utilizou-se a seguinte fórmula:

$$(1-\alpha) * getQValue(state, action) + \alpha * (reward + \gamma * computeValueFromQValues(next\_state))$$

Tendo todas estas funções desenvolvidas, procedeu-se à verificação do algoritmo através do comando: "*python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid*". Este comando corre o algoritmo implementado num tabuleiro de pequenas dimensões. Além disso corre 100 jogos de teste após ter realizado 2000 jogos de treino. Destes 100 jogos mostra em tempo real a execução de 10 jogos. Após a execução deste comando verificou-se que na maioria das vezes a percentagem de vitórias era de 100%, sendo que em algumas situações era de 90%. De acordo com a documentação, a execução deste comando fornecia os parâmetros *epsilon*, *alpha* e *gamma* já otimizados para o problema [1].

3 Hiperparametrização do algoritmo

Após a construção do algoritmo descrito na secção anterior procedou-se à sua hiperparametrização. Neste caso pretende-se estimar três parâmetros: *alpha* e *gamma* que fazem parte da própria equação para atualizar o *Q value*, e o parâmetro *epsilon* que é responsável por introduzir algum comportamento aleatório ao agente de modo a explorar o ambiente onde está inserido. Após um estudo da documentação do ambiente de desenvolvimento para este problema, verificou-se que os valores por defeito destes três parâmetros correspondem aos melhores valores para a resolução deste problema, sendo o valor de *epsilon* 0.05, o de *alpha* 0.2 e o de *gamma* 0.8. Assim, tomando estes valores como referência o objetivo passou a ser o de encontrar outros valores que permitissem ao agente desenvolvido a resolução de 80 a 90% dos jogos no tabuleiro *smallGrid*. No caso da inexistência de tais valores, seriam efetivamente os valores por defeito mencionados, os melhores valores para o problema em questão. Para testar estes novos valores acrescentou-se, ao comando descrito na secção anterior, a opção *-a* seguida dos valores que se pretendiam testar.

Nesta secção descrevem-se as estratégias utilizadas para a obtenção dos melhores valores do algoritmo desenvolvido. O primeiro processo caracteriza-se como um algoritmo de "força bruta" em que se percorreu uma gama de valores fornecidos até se chegar ao resultado pretendido. O outro processo desenvolvido consiste num algoritmo mais inteligente e muito menos pesado computacionalmente chamado *Particle Swarm Optimization*.

3.1 Hiperparametrização com *Grid Search*

A primeira versão de hiperparametrização consistiu num processo denominado *Grid Search*. Este processo é de implementação simples mas bastante pesado computacionalmente, visto que é efetuado um teste para toda a gama de valores que se pretende estimar. Para isso foi efetuado um ciclo para cada parâmetro que se pretendia avaliar (neste caso foram efetuados 3 ciclos). A cada iteração é executada a função *runGames* que devolve uma lista que contém informações dos diversos jogos efetuados. Através dessa lista pode-se calcular o rácio de jogos que o *pacman* venceu. Esse calculo foi efetuado por uma função auxiliar chamada *get_win_ratio* que percorre a lista de jogos e verifica para cada posição dessa lista o *score* obtido. Um *score* com um valor positivo diz respeito a uma vitória. De modo a otimizar-se o processo de obtenção dos parâmetros por este algoritmo de hiperparametrização, não se efetuou uma *Grid Search* convencional. Ao invés de se percorrer toda a gama de valores, desenvolveu-se um critério de paragem caso um determinado conjunto de parâmetros permitisse uma percentagem de vitórias igual ou superior a 90%. Optou-se por esta percentagem mínima e não de 80% de modo a aumentar o grau de exigência da hiperparametrização. O algoritmo desenvolvido foi o seguinte:

```
def evaluate_grid_search(command):
    epsilon = 0
```

```
alpha = 0
gamma = 0
win_ratio = 0

command.append("-a")
command.append(build_options(epsilon, alpha, gamma))

# calcular valores entre 0 e 1 inclusive com steps de
    0.05 para epsilon e de 0.1 para alpha e gamma
for i in range(21):
    if win_ratio > 0.8:
        break
    for j in range(11):
        if win_ratio > 0.8:
            break
        for k in range(11):
            command[-1] = build_options(i * 0.05, j *
                0.1, k * 0.1)
            args = readCommand(command[1:])
            games = runGames(**args)
            aux_win_ratio = get_win_ratio(games)
            if aux_win_ratio >= win_ratio:
                win_ratio = aux_win_ratio
                epsilon = i * 0.05
                alpha = j * 0.1
                gamma = k * 0.1
            if win_ratio > 0.8:
                break

print('Grid Search win ratio=', win_ratio)
print('epsilon:',epsilon, ",alpha:", alpha, ",gamma:"
    , gamma)
```

A gama de valores escolhida foi entre 0 e 1 inclusive. Para o parâmetro *epsilon* efetou-se um incremento de 0.05 unidades e para os parâmetros *alpha* e *gamma* efetou-se um incremento de 0.1 unidades. Após a execução deste algoritmo obteve-se o seguinte resultado:

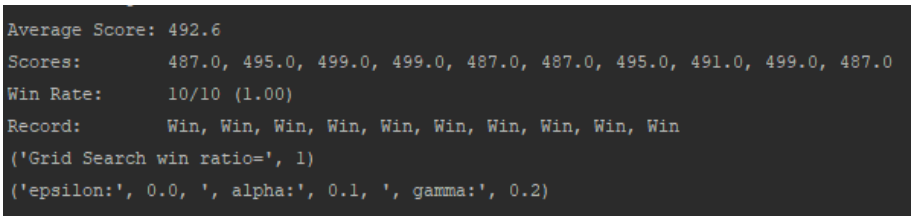

Figura 1: Resultado obtido pelo processo de *Grid Search*

A análise dos resultados revela que ao fim de 12 iterações chegou-se a uma solução que permitiu ao agente resolver todos os jogos. Deste resultado concluímos que os parâmetros por defeito permitiam não só maximizar o número de vitórias como também maximizar o *score* de cada jogo.

3.2 Hiperparametrização com *Particle Swarm Optimization*

Tal como foi dito, o processo de *Grid Search* é de fácil implementação, no entanto é bastante pesado computacionalmente e para este funcionar corretamente é necessário fornecer uma gama de valores a testar (pelo que pode mesmo assim não encontrar a solução mais adequada ao problema). Para a gama de valores testados, foi até necessário desenvolver um critério de paragem para este algoritmo, caso contrário o processo de descoberta dos melhores parâmetros demoraria dias de execução. Posto isto, procedeu-se à implementação de um processo de hiperparametrização que permitisse uma convergência mais rápida e inteligente. O processo implementado demonina-se *Particle Swarm Optimization* que é baseado em padrões encontrados na natureza (movimento de um indivíduo encontrado em cardumes de peixes ou bandos de aves). Este algoritmo gera um conjunto de soluções válidas para um determinado problema. Cada solução é gerada através de uma amostra chamada partícula e a essa solução é atribuído um valor. Cada partícula tenta melhorar este valor de modo a atingir um determinado objetivo. Este conjunto de partículas é denominado de espaço de procura em que é avaliada constantemente a melhor solução de cada uma das partículas. A melhor solução é caracterizada como sendo a melhor experiência global do espaço de procura. A cada iteração as partículas tendem a convergir para a melhor solução global.

Para a construção deste algoritmo optou-se por desenvolver uma arquitetura orientada a objetos com recurso a duas classes distintas em *Python* chamadas *Particle* e *Space*. A classe *Particle* é definida à custa de dois vetores de três posições (já que se pretende estimar três parâmetros) sendo estes o vetor da posição da partícula, ou seja, a solução do problema, e o vetor velocidade. Este vetor consiste no vetor que é somado ao vetor da posição da partícula de modo a esta convergir para a partícula que se encontra na melhor posição do espaço de procura. Este vetor é inicializado a zero. O vetor da posição da partícula

é incializado com um valor aleatório entre 0 e 1 inclusive. Além disso, uma partícula guarda também a melhor posição alcançada e o melhor valor dado por essa posição. Inicialmente este valor é inicializado com um valor negativo, visto o objetivo máximo ser 1 (ou seja 100% de vitórias). Foram também implementados um método de movimento da partícula e um método que permite ilustrar a partícula na consola de texto. A classe é a seguinte:

```
class Particle:
    def __init__(self):
        self.position = np.array([random_position(),
            random_position(), random_position()])
        self.best_position = self.position
        self.best_value = -0.1
        self.velocity = np.array([0.0, 0.0, 0.0])

    def __str__(self):
        print("Position: ", self.position, " Best
            position: ", self.best_position)

    def move(self):
        self.position = self.position + self.velocity
```

A classe *Space* contém todas as partículas e contém a melhor posição e o melhor valor obtido pelas partículas. Além disso, contém o valor objetivo que neste caso é 1 (100% de vitórias), assim como um valor mínimo de erro (neste caso considerou-se 0.1, ou seja, 10% de derrotas). Recebe como parâmetros o número de partículas e parâmetros associados ao processo do cálculo da velocidade a ser aplicada às partículas (*w*, *c1* e *c2*). Inicialmente, todas as partículas são inicializadas através do construtor da classe *Particle*. A cada iteração do algoritmo é invocado o método *set_best*, que percorre cada partícula e calcula o valor da solução de acordo com a posição atual desta através de uma função auxiliar chamada *fitness*. Caso o valor obtido seja uma melhor solução que a sua solução anterior, então é atualizada a melhor posição e a melhor solução dessa partícula. Caso esta solução seja melhor que a solução global do espaço de procura, então essa passa a ser a melhor solução do espaço. Além disto, esta classe contém um método para mover as partículas após cada iteração do algoritmo e um método para ilustrar todas as partículas na consola de texto. A classe é a seguinte:

```
class Space:
    def __init__(self, target, target_error,
        num_particles, w, c1, c2):
        self.target = target
        self.target_error = target_error
        self.num_particles = num_particles
        self.particles = []
        self.best_value = -0.1
```

```
        self.best_position = np.array([random_position(),
            random_position(), random_position()])
        self.w = w
        self.c1 = c1
        self.c2 = c2

    def print_particles(self):
        for p in self.particles:
            p.__str__()

    @staticmethod
    def fitness(particle):
        return run_game(particle)

    def set_best(self):
        for p in self.particles:
            fitness_value = self.fitness(p)
            if p.best_value < fitness_value:
                p.best_value = fitness_value
                p.best_position = p.position
            if self.best_value < fitness_value:
                self.best_value = fitness_value
                self.best_position = p.position
            if abs(self.best_value - self.target) <= self
                .target_error:
                break

    def move_particles(self):
        for p in self.particles:
            new_velocity = (self.w * p.velocity) + (self.
                c1 * random.random()) * (p.best_position -
                p.position) + \
                           (random.random() * self.c2) *
                               (self.best_position - p.
                               position)
            p.velocity = new_velocity
            p.move()
```

De notar que no método *set_best* foi imposta uma otimização em que não se testam todas as partículas caso uma destas já tenha atingido uma solução admissível. Este critério foi acrescentado já que, no contexto deste problema, não é necessário que todas as partículas cheguem à melhor solução.

Tendo estas duas classes chave construídas, desenvolveu-se um ciclo que a cada iteração testa a melhor solução gerada por cada uma das partículas contidas no espaço de procura e verifica se alguma dessas soluções já atingiu o objetivo dentro do erro mínimo pretendido. Caso esse objetivo ainda não tenha

sido cumprido, movem-se as partículas em direção à melhor solução atingida até ao momento e repete-se o ciclo. O código desenvolvido foi o seguinte:

```
def evaluate_pso(w, c1, c2):
    space = Space(1.0, 0.1, 10, w, c1, c2)
    particles = [Particle() for _ in range(space.
        num_particles)]
    space.particles = particles
    space.print_particles()
    num_iterations = 20
    i = 1

    while i < num_iterations:
        space.set_best()
        print("Particles set")
        if abs(space.best_value - space.target) <= space.
            target_error:
            break
        space.move_particles()
        i = i + 1

    print("PSO best solution is: epsilon=", space.
        best_position[0], ",alpha=", space.best_position
        [1], ",gamma=",
           space.best_position[2])
    print("PSO value: ", space.best_value)
    print("Iterations: ", i)
    return i, space.best_value
```

A função recebe como parâmetros três componentes pertencentes à atualização das velocidades das partículas e devolve o número de iterações necessárias até ter atingido o objetivo pretendido, assim como o melhor valor atingido. De notar que não é garantido que este algoritmo devolva a melhor solução, sendo que se estipulou um número máximo de 20 iterações. Inicialmente esta função foi testada com valores de *w*, *c1* e *c2* arbitrários, sendo que se conseguiu atingir uma percentagem de vitórias entre 90 a 100% em apenas uma iteração. Isto mostra que o algoritmo assemelhou-se muito a um algoritmo de pesquisa aleatória já que não teve tempo para as partículas convergirem para uma melhor solução global. Também evidencia o facto que o algoritmo de *reinforcement learning* produzido permite um conjunto de soluções que garantem esta percentagem de vitórias. Os valores escolhidos foram: *w=0.5*, *c1 = 0.8* e *c2 = 0.9*.

De modo a avaliar os parâmetros necessários para o cálculo da velocidade das partículas do algoritmo de *paricle swarm optimization*, procedeu-se à implementação de uma nova *grid search* através de um conjunto de ciclos que avaliou uma gama entre 0 a 1, com um incremento de 0.1 unidades. A cada iteração, é invocada a função descrita acima (*evaluate_pso*) e testa-se o número de iterações obtidas com os parâmetros enviados. O ciclo termina caso o algoritmo tenha

convergido para uma solução ótima em apenas uma iteração. Optou-se por esta estratégia de modo a não ter que se avaliar todas as combinações, o que seria um processo bastante moroso. Além disso, uma iteração indica que os parâmetros recebidos são ótimos. O resultado obtido foi o seguinte:

```
('PSO best solution is: epsilon=', 0.16784792153877567, ',alpha=', 1.0, ',gamma=', 0.8315537193885034)
('PSO value: ', 1.0)
('Iterations: ', 1)
('PSO best values: w=', 0.0, ',c1=', 0.0, ',c2=', 0.0)
('Iterations: ', 1)
```

Figura 2: Resultado obtido pelo processo de *Particle Swarm Optimization*

O resultado indica que na primeira iteração, os valores fornecidos como parâmetro ao algoritmo de hiperparametrização, são suficientes para o algoritmo atingir a solução ótima na primeira iteração [2].

4 Análise do algoritmo desenvolvido para mapas de maiores dimensões

Tendo-se implementado com sucesso os dois mecanismos que permitiam uma hiperparametrização do algoritmo desenvolvido, descritos na secção anterior, verificou-se como esse algoritmo seria capaz de se comportar aumentando a complexidade do tabuleiro do jogo. Para isso executou-se o comando "*python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l mediumGrid*" para a execução de todo o processo de treino e de teste do algoritmo com um tabuleiro de dimensões superiores. Inicialmente executou-se este comando com os parâmetros por defeito que segundo a documentação são os que apresentam melhores resultados para este problema. Após múltiplas execuções desse comando, verificou-se que através desses parâmetros por defeito a percentagem de vitórias variou entre os 0 e os 20%. Após essa tentativa executou-se novamente uma *grid search* com o algoritmo *particle swarm optimization* desenvolvido. Após cerca de 8 horas de execução desse processo verificou-se que, até esse momento, a melhor solução variava entre 30 a 40% de vitórias. Isto indicou que o algoritmo de *reinforcement learning* desenvolvido não é o mais adequado para a resolução de tabuleiros mais complexos.

Analisando o comportamento do *pacman* na execução dos jogos em tabuleiros com dimensões superiores e o algoritmo desenvolvido, verifica-se que a estratégia para encontrar a melhor política não é a mais adequada. Isto porque a cada estado apenas se considera a melhor ação a tomar que pode apenas ser uma das quatro hipóteses: mover para norte, este, oeste ou sul. Isto faz com que o *pacman* tome comportamentos muito aleatórios já que não consegue prever o melhor conjunto de soluções a tomar que maximizam uma recompensa tardia. Uma possível melhoria neste aspeto seria considerar não só a melhor ação a

tomar num determinado estado, mas sim o conjunto de N ações a tomar. Ou seja, ao invés de se mover para uma determinada direção, determinar caminhos que o *pacman* podesse seguir de modo a reunir todas as peças necessárias para a vitória ou escapar ou seu inimigo (ou inimigos, dependendo da complexidade do tabuleiro). Evidentemente que neste processo, o custo computacional seria muito mais pesado já que existiria um número muito mais elevado de ações a considerar. Outro aspeto que tem vindo a ser implementado para resolver jogos deste género, muito testado para jogos clássicos da *Atari* é o de complementar este processo de *reinforcement learning* com uma rede neuronal artificial, por exemplo. Este complemento permite introduzir um mecanismo de memória ao algoritmo, alimentando sucessivamente os dados de treino da rede com o conjunto de ações tomadas, produzindo modelos mais independentes de certos aspetos como dimensões e complexidade dos tabuleiros. Obviamente, todo este processo acarreta um elevado custo computacional [3, 4].

5 Conclusão

Este trabalho foi dividido em duas partes diferentes. Uma primeira parte onde foi desenvolvido o treino do modelo utilizando *reinforcement learning* e uma segunda fase onde foram utilizadas duas técnicas de hiperparametrização para se obterem os melhores valores a serem usados na primeira parte.

Relativamente à primeira fase, como se recebeu previamente um esqueleto do projeto a desenvolver com as instruções necessárias para a implementação do algoritmo, adquiriram-se conhecimentos acerca de *reinforcement learning*, nomeadamente o processo da política de *q learning*.

Na segunda fase, com a implementação do algoritmo de *Particle Swarm Optimization*, foi possível comparar esse mecanismo de hiperparametrização com outro de fácil implementação mas com um custo computacional bastante mais elevado, o *Grid Search*.

Finalmente, tentou-se avaliar como o algoritmo desenvolvido se comportaria face a tabuleiros mais complexos, tendo-se verificado que a implementação desenvolvida não era adequada para tal efeito. Para melhorar este aspeto propõe-se uma alteração da quantidade de ações a verificar em cada estado, ou complementar o algoritmo desenvolvido com outra estrutura de dados, como por exemplo uma *RNN* ou uma *LSTM*.

Referências

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

[2] https://pyswarms.readthedocs.io/en/latest/examples.tutorials.html

[3] https://towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814

[4] https://towardsdatascience.com/advanced-reinforcement-learning-6d769f529eb3