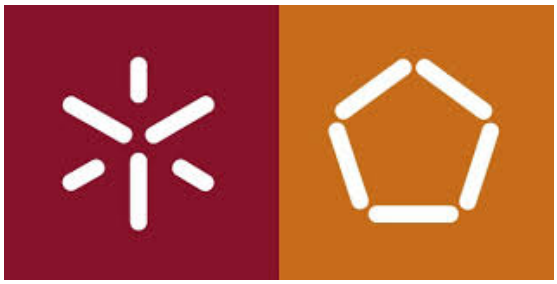

Universidade do Minho - Escola de Engenharia

Mestrado Integrado em Engenharia Informática

Computação Natural

Trabalho prático Reinforcement Learning por Agentes

Carlos José Gomes Campos a74745
José Pedro Ferreira Oliveira a78806
Vitor José Ribeiro Castro a77870

20 de Maio de 2019

Resumo

Este relatório descreve os passos necessários ao desenvolvimento do projeto proposto pela equipa docente, a saber, a aprendizagem por reinforcement learning de um agente. O processo compreende o desenvolvimento de um modelo de reinforcement learning, - no caso, Q-Learning, - bem como a otimização dos parâmetros de treino, - no caso, através de Particle Swarm Optimization. Para tal, o ambiente de desenvolvimento requerido é Python. Este projeto foi desenvolvido no âmbito da unidade curricular de Computação Natural, do perfil de Sistemas Inteligentes, do Mestrado Integrado em Engenharia Informática, na Universidade do Minho.

Conteúdo

1 Introdução

1.1 Contextualização

Recentemente, com o desenvolvimento da área de *machine learning* e a perceção das limitações que os paradigmas de aprendizagem supervisionada e não supervisionada têm, começou a ser dada maior importância e aplicação a *reinforcement learning*.

Atualmente, os sistemas de informação e interação são cada vez mais dinâmicos, com mais interações a surgirem a cada momento. Com isto, os algoritmos de reinforcement learning ganham vantagem perante os outros, uma vez que permitir equilibrar a razão entre exploração de conhecimento e uso do mesmo. Adicionalmente, permite implementar ambientes formulados por um Processo de Decisão de Markov, sem ter a carga computacional necessária para a criação de um modelo grande.

Indubitavelmente, com o crescimento dos jogos virtuais e da necessidade de melhorar as suas ações e realidade, a inteligência artificial tem sido aplicada em cada vez mais títulos, pelo que se torna interessante explorar estes recursos.

1.2 Caso de Estudo

No caso do projeto, pretende-se aplicar a inteligência artificial numa ótica de resolução de problema, ou seja, criar um jogador inteligente. O jogo a tratar é o Pac-Man, título antigo e que, assim, permitirá a máquinas mais modestas criar agentes inteligentes que, por conceção, seriam utilizáveis em situações de maior complexidade.

O agente deverá ser treinado para ter a melhor *performance* possível, para um mapa específico.

1.3 Estrutura do Relatório

O relatório está dividido em cinco partes, correspondentes às etapas de criação do modelo de Q-Learning, Otimização com PSO, Análise de Resultados, Análise em Outros Mapas e Conclusão.

Na secção 2 explica-se o processo de criação do modelo e explica-se cada umas das funções editadas, de acordo com o sugerido. Na secção 3 descreve-se todo o algoritmo de otimização, bem como o porquê do uso desta ao invés de outas opções. Na secção 4, faz-se a análise dos resultados obtidos ao longo das iterações de melhoramento do algoritmo de otimização. Na secção 5, discutem-se os desafios e melhoramentos possíveis para mapas de maior dimensão, comparando-se os resultados obtidos num mapa de tamanho médio com um mapa de tamanho pequeno.

O relatório termina com as conclusões na secção 6 onde é feita uma reflexão acerca do trabalho realizado e as dificuldades sentidas ao longo do mesmo.

2 Modelo Q-Learning

Tal como proposto no enunciado do trabalho, o grupo desenvolveu uma implementação do modelo Q-Learning para a resolução do problema, que consiste em treinar um agente para jogar Pac-Man. Esta implementação está restringida pelos módulos que dão suporte ao jogo, disponibilizados pela equipa docente. Desta forma, editou-se apenas as funções: *computeValueFromQValues*, *computeActionFromQValues*, *getQValue*, *getAction* e *update*.

2.1 *computeValueFromQValues*

Nesta função o objetivo é devolver o maior Q-Value cuja ação seja permitida. Nesse sentido, cria-se uma estrutura que reserve todos os elementos de Q-Value associados a ações legais, sendo que no fim se retorna o maior desses valores registados. Caso nenhuma ação legal exista é retornado o valor 0.0, graças às propriedades de inicialização desta estrutura.

```
def computeValueFromQValues(self, state):
        """
          Returns max_action Q(state,action)
          where the max is over legal actions.
          Note that if there are no legal actions,
          which is the case at the terminal state,
          you should return a value of 0.0.
        """
        "*** YOUR CODE HERE ***"
        # to store legal values to iterate
        legalValues = util.Counter()
        for action in self.getLegalActions(state):
            legalValues[action] = self.getQValue(state,action)
        # return the biggest value of all
        return legalValues[legalValues.argMax()]
```

2.2 *computeActionFromQValues*

Nesta função o objetivo é devolver a ação cujo *Q-Value* é o maior, em relação ao conjunto de ações permitidas. É verificada a existência de ações legais e, caso não existam, é retornado um *None*. Caso exista, cria-se uma estrutura que reserve todos os elementos de *Q-Value* associados a ações legais, sendo que no fim se retorna a chave correspondente ao maior *Q-Value* desses valores registados.

```
def computeActionFromQValues(self, state):
        """
          Compute the best action to take in a state.
          Note that if there are no legal actions,
          which is the case at the terminal state,
          you should return None.
        """
        "*** YOUR CODE HERE ***"
        # check if there's any legal actions
        # (not in terminal state)
        actions = self.getLegalActions(state)
        if not actions:
            return None

        # to store legal values to iterate
        legalValues = util.Counter()
        for action in actions:
            legalValues[action] = self.getQValue(state,action)
        # return the key which has the biggest value
        return legalValues.argMax()
```

2.3 *getQValue*

Nesta função o objetivo é devolver o *Q-Value* correspondente a um par *State-Action*. Novamente, devido às propriedades da estrutura utilizada, é possível devolver o valor 0.0 caso um par ainda não tenha sido visitado, ou o valor respetivo caso já tenha. Para que este uso seja possível, na função **__init__** é inicializada a estrutura que regista os *Q-Values*.

```
def getQValue(self, state, action):
        """
          Returns Q(state,action)
          Should return 0.0 if we have never seen a state
          or the Q node value otherwise
        """
        "*** YOUR CODE HERE ***"
        return self.Qvalues[(state,action)]
```

2.4 *getAction*

Nesta função o objetivo é devolver a ação a tomar no estado atual. Para isso, e tendo em conta a necessidade de exploração definida em *epsilon*, recorre-se à função **flipCoin** disponibilizada para ter em conta essa aleatoriedade. Num caso, opta-se por escolher de forma aleatória qualquer uma ação legal e, no outro, a ação preferível de acordo com a política existente. É verificada, naturalmente, a existência de ações legais, de modo a prever a localização em estado final.

```
def getAction(self, state):
        """
          Compute the action to take in the current state.  With
          probability self.epsilon, we should take a random action and
          take the best policy action otherwise.  Note that if there are
          no legal actions, which is the case at the terminal state, you
          should choose None as the action.

          HINT: You might want to use util.flipCoin(prob)
          HINT: To pick randomly from a list, use random.choice(list)
        """
        # Pick Action
        legalActions = self.getLegalActions(state)
        action = None
        "*** YOUR CODE HERE ***"
        if legalActions:
            if util.flipCoin(self.epsilon):
                action = random.choice(legalActions)
            else:
                action = self.computeActionFromQValues(state)

        return action
```

2.5 *update*

Esta função é disponibilizada ao motor do jogo e será chamada para atualizar os *Q-Values* dos estados anteriores, após recompensa recebida. É feito o cálculo da recompensa estimada e adicionada ao *Q-Value* correspondente, mediante os valores *alpha* e *gamma* usados na inicialização do processo de treino.

```
def update(self, state, action, nextState, reward):
        """
          The parent class calls this to observe a
          state = action => nextState and reward transition.
          You should do your Q-Value update here

          NOTE: You should never call this function,
          it will be called on your behalf
        """
        "*** YOUR CODE HERE ***"
        estimatedQ =
            reward
            + self.discount * self.computeValueFromQValues(nextState)
        self.Qvalues[(state,action)] =
            (1 - self.alpha) * self.getQValue(state,action)
            + self.alpha * estimatedQ
```

3 Otimização por PSO

Com vista à otimização dos parâmetros a utilizar para a fase de treino do *Q-Learning*, implementou-se um algoritmo de *Particle Swarm Optimization*. O objetivo é que cada partícula procure os melhores valores para *epsilon*, *gamma* e *alpha*, na medida em que um bom treino irá permitir um bom *score* médio no teste do conhecimento adquirido. Cada partícula partilhará, posteriormente, o resultado do uso dos seus valores para estes parâmetros, ocorrendo um ajuste para aquele que é o melhor resultado pessoal e da população.

Para a realização da otimização exigida, criou-se um módulo Python de nome "pso". Este pode ser corrido com recurso ao comando **python pso.py** e implementa o algoritmo de PSO para 3 dimensões, a saber, de *epsilon*, *alpha* e *gamma*. Adicionalmente, é também inicializado um limite para os valores que estes parâmetros a otimizar podem tomar, que é de 0 a 1 para todos.

O número de partículas definidas para a otimização é de 15 e o número de iterações a fazer é 30. Estes valores foram aceites como necessários e suficientes pelo grupo, tendo em conta o tempo e complexidade do problema. Para o caso em questão, o algoritmo de otimização corre em cerca de 6 horas, um valor que permite a sua execução em períodos noturnos numa máquina de 2 cores sem gpu.

3.1 Partícula

No caso em questão, uma partícula não passa de um momento de treino e teste único, que podia ser executado através do comando **python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid -a epsilon=0.1,alpha=0.3,gamma=0.7**, para o caso em que epsilon=0.1, alpha=0.3 e gamma=0.7.

Primeiramente, é necessário inicializar o conjunto de todas as partículas. Na sua inicialização, as partículas são compostas por:

- posição atual (position_i)
- velocidade atual (velocity_i)
- posição que melhor *score* obteve (pos_best_i)
- melhor *score* obtido (score_best_i)
- *score* obtido na posição atual (score_i)

O melhor *score* é inicializado com um valor indicativo, -1, para que se saiba que ainda não foi feita qualquer avaliação até ao momento. A posição é escolhida de forma aleatória, com recurso à função random() do módulo random, que retorna um valor entre 0 e 1. Adicionalmente, a velocidade inicial é definida como sendo 0, por forma a prevenir explosão de partículas e *clamping* posterior.

3.2 Algoritmo

Apesar de o algoritmo *PSO* ser algo já conhecido, interessa explicar a sua aplicação no caso em específico. Assim, os passos executados são os seguintes:

- Avaliar a performance da partícula, com os parâmetros atuais (posições);
- Verificar se a partícula obteve um resultado melhor que todos os outros que obteve anteriormente. Se sim, atualizar melhor *score* e posição pessoal;
- Fazer a mesma verificação em relação aos resultados globais;
- Avaliar as partículas restantes;
- Recalcular as posições e velocidades das partículas;
- Iterar novamente por todas as partículas.

3.3 Alternativas a PSO

Apesar de PSO ser o algoritmo proposto no enunciado, o grupo decidiu pesquisar sobre quais as alternativas disponíveis ao uso deste. A primeira alternativa que surgiu foram Algoritmos Genéticos, devido à experiência anterior dos elementos do grupo. Esta opção foi abandonada pois percebeu-se que os GA são melhores em problemas de otimização discreta. De facto, proceder a mutações para os valores de *epsilon*, *alpha* e *gamma* dificilmente iria levar a uma aproximação de um valor globalmente bom. O problema em questão é, reconhecidamente, de otimização contínua.

Partindo para o *Gradient Descent*, o grupo verificou que o problema em causa não tem uma função de custo expressa e que seja derivável facilmente. De facto, não se procura a aproximação a um conjunto de pontos mas apenas a melhora de um *score* cujo mínimo e máximo é desconhecido e o valor é apenas obtido no fim do cálculo, em cada partícula.

Analisando sob a perspetiva de Evolução Diferencial, pode-se verificar a existência de *crossover* e *mutation*, caindo no mesmo mal de que os Algoritmos Genéticos sofrem, ou seja, que as mutações causem mais dispersão dos resultados e sejam feitos cálculos desnecessários.

Adicionalmente, analisou-se o *Firefly Algorithm* que, apesar de interessante, iria levar à necessidade de muitas partículas. Este algoritmo é diferenciado do PSO por conseguir encontrar vários mínimos locais e, dentro desses, encontrar o mínimo global. Se em alguns problemas isto pode ser relevante, no problema em causa não o é e iria causar mais carga computacional para um problema de relativamente pouca complexidade.

Por fim, tendo-se analisado outros algoritmos de forma menos aprofundada, foi possível verificar que o PSO é um algoritmo de fácil implementação e complexidade computacional baixa face a outros. Assim, faz sentido usá-lo sobre qualquer outro algoritmo analisado.

4 Análise de Resultados

Tendo sido corrido o algoritmo de otimização, numa primeira fase, obtiveram-se os seguintes valores que se pensavam ser ideais:

- *epsilon:* 0
- *alpha:* 0.9954457084902311
- *gamma:* 1

Com estes valores, o *score* médio, obtido em 10 jogos, foi de 503 pontos. Este *score* representa uma taxa de vitória de 100 por cento, numa série de jogos perfeitos cujo *score* máximo é precisamente 503. Com estes valores de treino foram realizados mais testes que resultaram sempre numa taxa de vitória de 100 por cento.

Interessa também verificar o custo do processo de otimização, nesta fase inicial, que durava 7h30m, com 2 *cores* de *CPU*. Este valor podia ser reduzido, quer por menor número de partículas (preferencialmente), quer por menos número de iterações. Todavia, o grupo considerou este ser um tempo de execução satisfatório face à complexidade do problema.

No entanto, depois do estudo que o grupo decidiu fazer em relação ao comportamentodo algoritmo em outros mapas, verificou-se que havia uma importante melhora a fazer. De facto, num momento inicial do treino, acontecia frequentemente de o agente avançar para uma casa e retornar para a anterior, consecutivamente, graças à existência de todos os Q-Value ser de 0. Como é feita uma iteração sobre os Q-Value, o algoritmo selecionava sempre a primeira ação legal, apenas não o fazendo por responsabilidade de *epsilon*. Assim, o grupo implementou aleatoriedade na escolha da próxima ação quando se verificasse que a escolha seria entre ações com Q-Value = 0. Com isto, verificou-se a um melhoramento na fase inicial de exploração por parte do agente, levando a duas situações: primeiramente, porque permitia explorar mais o mapa e atualizar mais rapidamente os Q-Value, tornando o treino mais eficaz; adicionalmente, reduziu o tempo de término de uma iteração de treino, pois o agente não ficaria num determinado local até que um fantasma o encontrasse.

Com os melhoramentos referidos, foi possível reduzir o tempo de execução da otimização. Adicionalmente, o grupo removeu a opção gráfica para os jogos de teste, o que levou a uma nova redução de tempo de execução do algoritmo, o que permitiu aumentar o número de jogos de teste, de 10 para 100. Este aumento permite remover um fator de sorte que possa ocorrer, fornecendo mais fiabilidade à escolha da melhor partícula. O número de partículas, 15, foi considerado ajustado ao problema pois reflete a norma de que o número de partículas deve ser 3 a 4 vezes superior à complexidade do problema, ou seja, de 3, sendo no caso usado um fator 5. O número de jogos de treino, 2000, não foi alterado, pois considera-se que se era suficiente para o treino anteriormente, neste momento também será. O número de iterações, 30, manteve-se após se verificar que as partículas convergem para aquela que fornece o melhor resultado. Quer os

valores a otimizar, quer o estado das partículas, são apresentados de seguida, provando a sua convergência e valores finais de otimização.

Parâmetros finais de otimização:

- *epsilon:* 0
- *alpha:* 0.8528980101257206
- *gamma:* 1

Posições das partículas, atestando convergência:

- *epsilon:* 0, *alpha:* 0.7745934340590099, *gamma:* 1
- *epsilon:* 0, *alpha:* 0.534612646535806, *gamma:* 1
- *epsilon:* 0.0016595236545817545, *alpha:* 0.8836159246652415, *gamma:* 1
- *epsilon:* 0, *alpha:* 0.2627160232887614, *gamma:* 1
- *epsilon:* 0.0434282134806337, *alpha:* 0.009128751802415236, *gamma:* 1
- *epsilon:* 0, *alpha:* 0.9097169552417116, *gamma:* 0.9794275617910148
- *epsilon:* 0, *alpha:* 0.7313054982529023, *gamma:* 0.9946388022651523
- *epsilon:* 0, *alpha:* 0.8403468352268277, *gamma:* 0.9980439869090084
- *epsilon:* 0, *alpha:* 0, *gamma:* 1
- *epsilon:* 0, *alpha:* 0.5274948929730368, *gamma:* 0.9998735237834728
- *epsilon:* 0, *alpha:* 0.8458936899803827, *gamma:* 1
- *epsilon:* 0, *alpha:* 0.7967045785631481, *gamma:* 1
- *epsilon:* 0, *alpha:* 0.8428498053783146, *gamma:* 0.9923536972082919
- *epsilon:* 0.007927716406400893, *alpha:* 1, *gamma:* 1
- *epsilon:* 0.00017285003848879657, *alpha:* 0.28801790569619623, *gamma:* 0.9952562189914202

5 Análise em Outros Mapas

Analisando a execução do algoritmo para vários mapas, foi possível tirar algumas conclusões face à eficácia e funcionamento do método. Foi possível identificar, primeiramente, que o tempo que demorava a fase de treino numa situação inicial, em que os Q-Value eram todos 0, era demasiado alta. Para corrigir isso, introduziu-se a aleatoriedade referida na secção anterior. Assim, na tabela que se mostra em baixo, é possível verificar a evolução do *winrate*, em percentagem, após essas modificações.

Adicionalmente, e como representado pela tabela seguinte, que é relativa ao mapa *mediumgrid*, desde logo é possível verificar que o número jogos de treino necessários tem que aumentar consideravelmente. Como referência, no caso da *smallgrid*, que serve de base para o relatório, 2000 jogos de treino são suficientes para atingir *winrates* superiores a 90 por cento dos jogos. Por oposição, 2000 jogos são indubitavelmente insuficientes quando comparamos com mapas maiores, como o *mediumgrid*. De facto, só acima de 15000 jogos é que se atingem margens de vitória próximas do anterior. Isto resulta da necessidade que existe de viver todos os casos possíveis por parte do agente, por forma a registar os Q-Value de forma correta na representação do mapa. Se o tamanho do mapa aumenta de, por exemplo, 4x4, para algo como 8x8, a necessidade de iterações não duplica, pois o número de combinações possíveis de Estado-Ação cresce de forma quase exponencial.

Visivelmente, o número de Estado-Ação cresce de forma exponencial, levando a um consumo cada vez maior de memória e da necessidade de controlar e calcular os Q-Values novos, trazendo limitações à aplicação deste algoritmo para mapas maiores. Uma medida que pode ser útil para prevenir esta situação é reduzir o tempo inicial de aprendizagem, sendo que a forma mais fácil de o fazer é, de forma simplista, fazer manualmente alguns jogos. Esta ação permitiria mostrar ao agente uma rota de sucesso possível e que poderia ser seguida, dando claro oportunidade a alguma variância e nova aprendizagem por causa de *epsilon*. Todavia, esta aproximação pode, em alguma situações, introduzir um *bias* negativo à aprendizagem de soluções novas por parte do algoritmo. Outra das ações, que foi tomada pela equipa, é introduzir alguma aleatoriedade aquando da existência de valores de Q-Value iguais a 0, que podia ser extendida para quando estes valores são baixos, ou seja, próximos de 0. A título de exemplo, os primeiros jogos num mapa pequeno, *smallgrid*, podiam demorar cerca de 3 segundos, ao passo que no *mediumgrid* podiam demorar 7 segundos. Para um mapa maior, seria perfeitamente possível durar até minutos, pois a única maneira de o jogo acabar quando o agente não toma ações aleatórias é perder por ser atingido por um fantasma. Por fim, uma das outras possíveis atualizações a fazer ao algoritmo seria o de promover uma atualização dos Q-Value em série, ou seja, que não implicasse apenas o estado atual e o anterior, mas sim um conjunto de estados anteriores, em proporção.

Apresenta-se, então, a tabela de resultados, que expõe a *winrate* do algoritmo para diferentes valores de *epsilon*, *gamma* e *alpha*, que foram sendo introduzidos manualmente e numa perspetiva de exploração por parte do grupo.

	winrate	treino	epsilon	gamma	alpha		winrate	treino	epsilon	gamma	alpha
SEM random pick quando q-value = 0	0	2000	0.05	0.8	0.2	COM random pick quando q-value = 0	4	2000	0.05	0.8	0.2
	0	5000	0.05	0.8	0.2		8	5000	0.05	0.8	0.2
	0	5000	0.15	0.8	0.2		9	5000	0.15	0.8	0.2
	0	10000	0.05	0.8	0.2		48	10000	0.05	0.8	0.2
	77	15000	0.00	0.8	0.2		86	15000	0.00	0.8	0.2
	75	15000	0.03	0.8	0.2		75	15000	0.03	0.8	0.2
	61	15000	0.05	0.8	0.2		76	15000	0.05	0.8	0.2
	54	15000	0.10	0.8	0.2		68	15000	0.10	0.8	0.2
	54	15000	0.15	0.8	0.2		59	15000	0.15	0.8	0.2
	23	15000	0.15	0.7	1.0		33	15000	0.15	0.7	1.0
	31	15000	0.00	1.0	1.0		43	15000	0.00	1.0	1.0
	82	20000	0.05	0.8	0.2		89	20000	0.05	0.8	0.2
	77	20000	0.15	0.8	0.2		75	20000	0.15	0.8	0.2

Figura 1: *winrate* por algoritmo e parâmetros, em *mediumgrid*

6 Conclusão

No decurso deste trabalho, o grupo encontrou algumas dificuldades e conseguiu resolver as mesmas. Se, inicialmente, a implementação do algoritmo de *Q-Learning* era algo que nunca havíamos feito, depois de análise tornou-se óbvio que a dificuldade maior seria compreender os módulos fornecidos pela equipa docente.

Resolvida essa questão, a equipa voltou-se para a otimização dos parâmetros que já tinham um valor *default*, conforme sugerido pelos criadores do exercício. O desenvolvimento e implementação do algoritmo de *PSO* permitiu à equipa compreender melhor o funcionamento do mesmo e quais as vantagens face a outros algoritmos de otimização já utilizados. Foi necessário adaptar os módulos fornecidos para correr o algoritmo mas o grupo obteve os resultados que pretendia. Após a análise feita para mapas maiores, foram encontrados alguns pontos a melhorar, sendo que o grupo se sente confiante com os resultados obtidos na conclusão do trabalho.