Zhed e Reinforcement Learning

Faculdade de Engenharia da Universidade do Porto - Mestrado Integrado em Engenharia Informatica e Computação - Inteligência Artificial 19/20

Matheus Gonçalves
MIEIC
FEUP
Ponte de Lima, Viana do Castelo
up201405081@fe.up.pt

Miguel Pires
MIEIC
FEUP
Montalegre, Vila Real
up201406989@fe.up.pt

Ricardo Cardoso
MIEIC
FEUP
Esmoriz, Aveiro
up201604686@fe.up.pt

***Abstract*—Este documento vem a propósito de um projeto desenvolvido na unidade curricular de Inteligência Artificial e tem como conteúdos o processo de desenvolvimento de um agente para o jogo de tabuleiro Zhed (Google Play) bem como o seu posterior estudo.**

***Index Terms*—zhed, reinforcement, learning, puzzle, boardgame, solitary, sarsa, q-learning, artificial, intelligence, gym, agent**

I. INTRODUÇÃO

O Zhed é um jogo de tabuleiro do tipo solitário que consiste em resolver um puzzle. O tabuleiro é constituído por quadrículas numeradas que representam as peças do jogo e uma casa de destino. O objectivo é expandir as peças numeradas (em que o número simboliza quantas casas irá ocupar quando expandida) em quatro direções (cima, esquerda, direita, baixo) de forma que a sua expansão cubra a casa destino. Uma mecânica interessante do jogo é o facto de uma peça já expandida poder ser "atravessada" por outra a expandir.

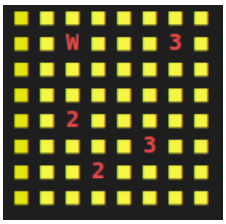

Fig. 1. Exemplo de um tabuleiro no seu estado inicial

II. DESCRIÇÃO DO PROBLEMA

O problema que nos foi proposto é referente à resolução do jogo Zhed com recurso à implementação de dois algoritmos de reinforcement learning, nomeadamente o Q-Learning e Sarsa. O objetivo é criar um agente que com os treinos necessários consiga resolver um nível recorrendo à informação que é guardada durante os treinos.

Para isso criamos um ambiente constituído por um tabuleiro que contém informação sobre o jogo. Esta informação é atualizada a cada iteração realizada por cada um dos algoritmos, sempre com o objetivo de treinar o algoritmo a atingir a solução do problema de uma forma o mais eficaz possível.

III. ABORDAGEM

Começamos por implementar toda a lógica do jogo usando a biblioteca OpenAI Gym no ficheiro `gym_zhed/envs/zhed_env.py`.

Tal lógica foi implementada definindo um espaço de observação bem como um espaço de ação para o nosso agente. O espaço de observação engloba todos os estados que o ambiente poderá assumir, para tal chegamos a uma fórmula que traduz esse mesmo número:

$$n! + 4^n$$

sendo n o número de peças presentes no tabuleiro.

Para o espaço de ação definimos 4 ações diferentes por cada peça ou seja,

$$4 * n$$

De seguida implementamos 2 algoritmos:

A. $Q\_Learning$

Inicialmente é verificado se ja existem alguns dados guardados de possíevis treinos realizados anteriormente dentro da pasta `tables`. Caso nao existam valores guardados é inicializada uma tabela com todos os valores iguais a 0.

De seguida começamos por definir quantos episódios (total de jogos) é que o agente irá concluir. Por cada episódio o agente escolhe uma ação entre duas hipóteses: ou explora o ambiente escolhendo uma ação aleatória do conjunto de ações disponíveis ou então aproveita a ação com o melhor valor de Q para aquele estado da tabela de valores guardados. A escolha de tal decisão (explorar ou aproveitar) é feita através de um valor pré definido que traduz a taxa de exploração (ϵ). Depois de escolhida a ação o agente dá o passo necessário a executá-la retornando um novo estado e uma recompensa por a ter executado. As recompensas são mapeadas da seguinte forma:

- Agente seleciona ação inválida:

$$r = -1$$

- Agente chega ao fim do jogo:

$$r = +1000$$

- Agente sai fora do tabuleiro:

$$r = -(n) * 10$$

,sendo n o número de casas que saiu fora.
- Agente expande uma peça:

$$r = (E - P) * 50$$

Sendo E igual ao número de casas que o agente expandiu numa dada direção e P o número associado à peça, deixando assim o nosso agente com uma reward positiva se expandiu mais casas do que aquelas que eventualmente poderia expandir ou negativa caso não aproveite todo o potencial de uma peça, isto é, caso a sua expansão seja feita para fora do tabuleiro.

Depois de ter feito uma ação o agente recalcula o novo valor para ser inserido na tabela seguindo a fórmula de Q-Learning:

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \alpha[\underbrace{R(s,a)}_{\text{Reward}} + \gamma \overbrace{\max Q'(s',a')}^{\text{Maximum predicted reward, given new state and all possible actions}} - Q(s,a)]$$

New Q-Value

Discount rate

- α : Simboliza a taxa de aprendizagem, i.e a importância da recompensa atual (quanto maior for o valor maior será o impacto da recompensa).
- γ : Factor de desconto que traduz a importância de recompensas futuras (valor alto significa mais importância em recompensas futuras).

Por fim o algoritmo verifica se conseguiu alcançar o objectivo que é preencher a casa destino, ou se alcançou o limite de ações num dado nível, dando o episódio por terminado em ambos os casos. Com o episódio terminado o tabuleiro volta ao seu estado inicial e o processo repete-se até atingir o número de treinos pretendido.

Finalmente os dados da tabela com os valores finais de Q são guardados num ficheiro de texto dentro da pasta `tables`.

B. SARSA (State Action Reward State Action)

O procedimento de treino para este algoritmo é exatamente igual ao de Q-Learning excepto na função de atualização de novos valores para a tabela. A única diferença que é visível é o facto de este algoritmo ter em conta no cálculo do novo valor de Q para uma determinada ação o estado futuro bem como a futura ação que irá desempenhar/executar.

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \alpha[\underbrace{R(s,a)}_{\text{Reward}} + \gamma \overbrace{Q'(s',a')}^{\text{Q-Value for next state and next action}} - Q(s,a)]$$

New Q-Value

Discount rate

IV. Avaliação Experimental

Como exemplo prático consideramos o nível 4 (Fig. 2). Um jogador humano facilmente conseguiria resolver este nível sem fazer muito esforço, teríamos então de mover a peça 3 na horizontal para a esquerda, a peça 1 de cima para baixo, e a peça 1 da esquerda executava o movimento para a direita para terminar o nível. No caso do nosso agente ele precisa primeiro explorar o ambiente e tirar notas para depois se tornar eficaz a resolvê-lo.

Fig. 2. Exemplo de nível para o jogo Zhed - Nível 4

No nosso caso para treinar o agente com o algoritmo de Q-Learning é preciso executar o ficheiro `train_q_learning` e especificar o nível (neste caso o 4) bem como o total de episódios a executar por treino.

É mportante deixar o agente experenciar o ambiente durante varias iterações e ir alterando os hiperparâmetros (α, γ, ϵ) de forma a se adequarem a cada nível em específico. No caso deste jogo as recompensas futuras são bastante importantes, ou seja, deve-se priorizar mais as recompensas futuras de forma a atingir o objectivo. Em muitos agentes de aprendizagem por reforço o valor de ϵ (epsilon) tem como valor um número muito próximo de 0 (ex: 0,1) mas como já averiguamos cada nível é um nível e por vezes um valor mais baixo de epsilon implica menos experimentação de caminhos que são à partida piores por parte do agente naquele determinado nível, mesmo que possam ser estes a originar a solução final depois de completamente explorados. Para este exemplo usaremos então valores adequados:

- α = 0.8
- γ = 1
- ϵ = 0.2

Correndo o treino para o nível 4 usando Q-Learning com um total de 5000 episódios obtemos o seguinte resultado:

```
Results after 5000 train sections:
Total timesteps: 24521
Total penalties: 10377
Total solutions found: 2992
Average timesteps per episode: 4.9042
Average penalties per episode: 2.0754
```

Fig. 3. Após 5000 episódios verificamos que por episódio houve a escolha de 2 ações inválidas por parte do agente, bem como uma média de 5 movimentos por episódio encontrando assim 2992 soluções.

Correndo novamente o mesmo teste com 5000 episódios e como epsilon prioriza as ações já exploradas obtemos o seguinte resultado:

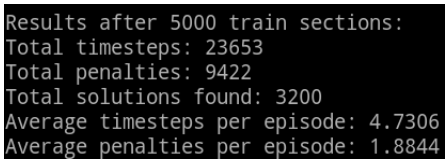

Fig. 4. Após 5000 episódios verificamos que por episódio houve a diminuição de passos por episódio bem como uma redução significativa nas penalizações. É de notar também o aumento de conclusões de nível de 2992 para 3200.

Para o SARSA fizemos o teste com diferentes valores nos hiper parâmetros:

- $\alpha = 0.8$
- $\gamma = 1$
- $\epsilon = 0.1$

Obtendo os seguintes resultados depois de 5000 episódios:

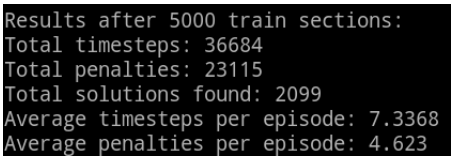

Fig. 5. Após 5000 episódios verificamos que teve mais passos comparado ao primeiro teste com Q-Learning, bem como mais penalizações durante o treino mesmo tendo um valor inferior de epsilon o que significa que o Q-Learning teve uma melhor performance no primeiro treino.

Fazendo outro treino de 5000 episódios obtivemos os seguintes resultados:

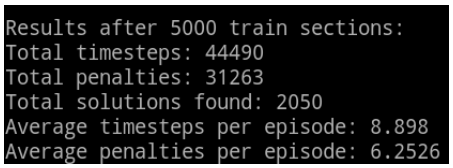

Fig. 6. Após outros 5000 episódios verificamos que não houve uma subida de performance por parte deste algoritmo, muito pelo contrário. O que nos leva a querer que para o contexto do nosso jogo, seria melhor usar q-learning.

Para depois verificar a integridade destes algoritmos bem como as tabelas de valores por eles gerados temos um script que simplesmente escolhe a melhor ação para um determinado estado a partir do maior valor para a ação dessa tabela. Correndo o mesmo verificamos que encontramos facilmente uma solução:

V. Conclusão

Após a realização deste trabalho concluímos que apesar de tanto o *Q-Learning* como o *SARSA* serem algoritmos de *aprendizagem por reforço* existem algumas diferenças entre eles. Primeiro detectamos que para obter resultados significativos usando SARSA teríamos de efetuar cada vez mais testes comparado ao Q-Learning. Também detectamos que SARSA aplicado ao nosso contexto não é assim tão bom quanto esperávamos.

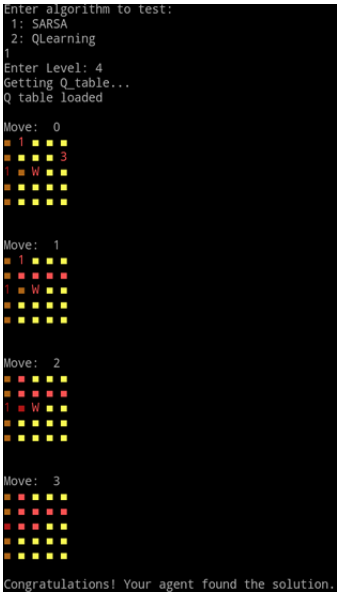

Fig. 7. Um episódio basta para chegar ao objectivo do jogo utilizando o algoritmo SARSA

Referências

[1] ZHED Solver.
[2] SARSA algorithm.
[3] Q-Learning algorithm.
[4] OpenAI Gym.
[5] OpenAI Gym Example.
[6] Russel, S., Norvig, P. (2010) Artificial Intelligence A Modern Approach, New Jersey..