Aprendizagem por Reforço: Pivit em Python

(Tema 2D / Grupo XX)

Fellipe Ranheri de Souza
MIEIC
FEUP
Porto, Portugal
up201700127@fe.up.pt

Pedro Leite Galvão
MIEIC
FEUP
Porto, Portugal
up201700488@fe.up.pt

Vasco Marques Lopes Teixeira
MIEIC
FEUP
Porto, Portugal
up201802112@fe.up.pt

O seguinte artigo faz parte na Unidade Curricular de Inteligência Artificial e aborda o tema Pivit(2D).

SARSA, Q-learning, AlphaGo, Jogo para dois jogadores, Phyton

I. Introdução

Neste artigo pretendemos detalhar a nossa abordagem na realização de um projecto da cadeira de inteligência artificial. Neste projecto pretendemos desenvolver um agente que seja capaz de aprender a jogar um jogo de tabuleiro, no nosso caso o "Pivit", com base em aprendizagem por reforço. Para esse efeito utilizamos um algoritmo de Q-Learning.

II. Formulação do Problema

O problema que nos propusemos a solucionar procura ensinar um agente, a partir dum algoritmo de deep Q-Learning, a jogar um jogo de tabuleiro chamado "Pivit".

I. Descrição do Jogo:

As peças básicas, chamadas de *minions*, devem movimentar-se para um quadrado de uma cor diferente. Ou seja, se eles começam num quadrado preto, devem movimentar-se para um quadrado branco. Peças promovidas, chamadas *masters*, podem pousar em qualquer lugar ao longo da sua linha.

As peças não podem saltar por cima de outras e capturam ao aterrar numa casa ocupada pelo inimigo. A promoção ocorre quando um *minion* se movimenta para um espaço do canto do tabuleiro.

O jogo termina quando todos os *minions* de ambos os jogadores são capturados. O vencedor é o jogador com mais *masters* no tabuleiro.

II. Operadores:

As operações a realizar a partir de cada estado são determinadas conforme as regras do jogo. No entanto, é de notar que para cada peça em cada estado existe um número significante de possíveis operações a realizar (aproximadamente 900 acções possíveis), o que fez com que se tornasse exaustivo em termos de processamento.

III. Abordagem

Nossa abordagem para solucionar o problema consiste na utilização de um algoritmo de Deep Q Learning. A motivação para a escolha deste algoritmo consiste no fato de que o número de estados e ações permitidos no jogo é grande demais para ser tratado por um algoritmo de Q Learning com matrizes de estados e ações.

Implementamos o algoritmo em python utilizando as bibliotecas open AI gym e Tensorflow. Utilizamos a interface Env disponibilizada na biblioteca gym para a construção de um ambiente customizado onde se dá o jogo. Utilizamos o Tensorflow, e em particular o Keras, que é inclui do no Tensorflow, para a implementação da rede neural utilizada.

Neste algoritmo faz-se uso de uma rede neural para mapear o estado do jogo a uma ação. Se implementado corretamente espera-se que a rede neural seja capaz de identificar quais características nos inputs são importantes e que tipo de ação deve-se tomar em função destas características.

O agente é posto em um ambiente a efetuar jogadas ou a observar jogadas de um outro agente e guarda a sequência de jogadas e as respectivas recompensas obtidas. Após o fim de cada episódio o agente efetua o treinamento da rede neural utilizando como inputs os estados e como target as ações multiplicadas pelos valores estimados que correspondem a elas. Estes valores são calculados utilizando as recompensas obtidas após esta ação e nas ações seguintes, aplicando-se um fator de desconto conforme a equação abaixo:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Decidimos criar um modo de converter o formato do tabuleiro de modo a obter um input que pudesse ser interpretado pela rede neural. O tabuleiro que antes era uma lista de listas com espaços vazios e objetos que representam as peças passa a ser representado por um conjunto de 5

matrizes com as mesmas dimensões que o tabuleiro com valores booleanos indicando as posições das peças. Estas matrizes representam o posicionamento das peças vermelhas verticais, vermelhas horizontais, azuis verticais, azuis horizontais e a última matriz serve para indicar quais das peças presentes nas outras 4 matrizes são "masters".

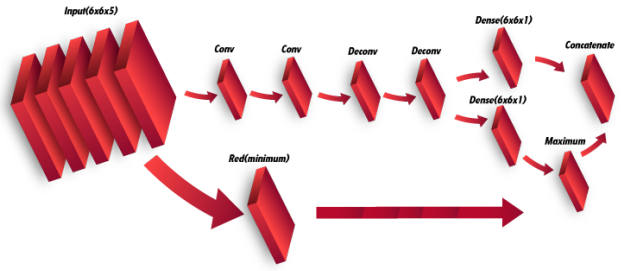
Decidimos utilizar camadas convolucionais pelo fato de que estas lidam bem com simetrias translacionais como as que estão presentes no tabuleiro do jogo. Após as camadas convolucionais adicionamos uma camada totalmente conectada (Dense) e dividimos o output em duas partes. Estas duas partes servirão a seguir para indicar no output a posição de origem e a posição de destino da peça a ser movida.

A primeira parte, que corresponde à origem, passa antes por uma camada que filtra no output apenas os valores correspondentes às posições no tabuleiro nas quais haja peças do jogador ativo. Por meio disto pretende-se impedir que a rede selecione como origem uma casa do tabuleiro que esteja vazia ou ocupada pelo adversário.

Em seguida os outputs das partes da rede que haviam sido divididas passam por camadas softmax. O objetivo disto é conseguir que o somatório dos valores para as possíveis posições de origem somem 1, assim como para as posições de destino, e deste modo o valor do output pode ser interpretado como uma probabilidade.

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

Ao fim disto concatenamos as partes da rede que haviam sido divididas e obtemos o output final. Este output é um array com dimensões (x,x,2), onde x é o tamanho do tabuleiro. Vemos na imagem abaixo uma representação do esquema da rede:



Para o treinamento da rede tentamos utilizar sequências de jogadas efetuadas pelo próprio agente jogando contra si mesmo ou contra um agente que utiliza algoritmo minimax.

IV. CONCLUSÕES

Infelizmente não conseguimos avançar tanto quanto pretendemos na implementação do agente e por isso este conta com algumas falhas que não fomos capazes de solucionar a tempo. Nomeadamente, o mecanismo pelo qual o agente deveria filtrar as possíveis posições de origem das peças movidas não funciona como pretendido, ocasionando que o agente selecione com muita frequência jogadas inválidas partindo de casas vazias ou ocupadas pelo adversário.

Tentamos também fornecer ao agente recompensas positivas por quaisquer jogadas válidas e negativas para jogadas inválidas, mas esta abordagem também não trouxe bons resultados - ao menos dentro do tempo que tivemos para treinar a rede.

DESENVOLVIMENTO FUTURO

Supondo que conseguíssemos solucionar o problema com a seleção da origem da jogada, poderíamos prosseguir em pensar em um modo de garantir que o destino selecionado pelo agente também seria válido, ou seja, estaria na mesma coluna ou fileira do tabuleiro que a posição de origem.

Uma vez que obtivéssemos assim um agente capaz de efetuar somente jogadas válidas, poderíamos ponderar melhor a forma da rede neural utilizada e realizar testes com diferentes arquiteturas para verificar o que funciona melhor.

Por exemplo, no modelo apresentado utilizamos uma sequência de duas camadas convolucionais e duas camadas de deconvolução, o que é uma escolha arbitrária. Possivelmente seria útil a associação de camadas de max pooling e camadas densas alternadas com as camadas de convolução, pois esta é uma técnica muito utilizada em redes convolucionais e com frequência trás bons resultados. Outra possibilidade seria a utilização de ResNet, um tipo de rede que demonstrou excelentes resultados com o programa Alpha Go Zero.

REFERENCES

[1] http://neuralnetworksanddeeplearning.com/
[2] https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf