Aprendizado por Reforço usando Aproximação de Funções

Fabrício Olivetti de França

Universidade Federal do ABC

Tópicos

1. Aproximação de Funções

2. Do the evolution

UFABC

Aproximação de Funções

Função Utilidade

Nos algoritmos das aulas anteriores, a política ótima era encontrada por meio de:

- uma função utilidade $U(s)$ que media o interesse em estar no estado atual ou,
- um Q-valor $Q(s, a)$, que media a expectativa de recompensa ao aplicar a ação a no estado s.

UFABC

Função Utilidade

Quando tais funções são conhecidas, a solução torna-se trivial. Mas na prática, raramente temos tal informação e, portanto, temos que estimar esses valores.

Função Utilidade

Na aula passada, tratamos $Q(s, a)$ como uma tabela e, através de amostras de episódios do nosso jogo, preenchemos os valores dela até convergência.

Função Utilidade

Percebemos, porém, que a convergência para jogos com muitos estados possíveis pode demorar além do viável.

Função Utilidade

Uma outra ideia é tentar estimar os valores de $Q(s, a)$ através de uma aproximação de função.

UFABC

Aproximando Q

Assumindo s como um vetor $d-$dimensional e a um vetor binário representando as possíveis ações, podemos tentar aproximar:

$$\hat{Q}(s, a) = w \cdot s + w' \cdot a,$$

com w, w' sendo os coeficientes lineares do vetor representando o estado e do vetor de ações, respectivamente.

UFABC

Aproximando Q

Para esse caso, podemos imaginar a função de erro como:

$(\hat{Q}(s,a) - Q(s,a))^2/2,$

com $Q(s,a)$ sendo o valor observado (por aplicar o Q-Learning após algumas centenas de episódios).

Aproximando Q

Podemos então aplicar o gradiente descendente com as regras:

$$w = w + \alpha(Q(s,a) - \hat{Q}(s,a)) \cdot s,$$

e

$$w' = w' + \alpha(Q(s,a) - \hat{Q}(s,a)) \cdot a,$$

UFABC

Aproximando Q

Basicamente o algoritmo pode ser definido de diversas formas, uma delas sendo:

- Aplique o algoritmo Q-Learning por alguns episódios.
- Aplique o algoritmo de regressão linear.
- Repita até convergência.

Aproximando R

Uma outra forma é substituir $Q(s, a)$ pela recompensa $R(s, a)$ na equação da regressão de tal forma que:

$\hat{R}(s, a) = w \cdot s + w' \cdot a$

e

$erro = (\hat{R}(s, a) - R(s, a))^2/2.$

UFABC

Aproximando R

Dessa forma o algoritmo se torna:

- Inicialize os pesos w, w'
- Aplique a política que maximiza as recompensas dada pela função linear.
- Faça a correção dos valores de recompensa utilizando o gradiente descentente
- Repita até convergir.

Aprendizado como um problema de classificação

Note que nossa regressão linear pode ser convertida para uma regressão logística assumindo que desejamos obter a probabilidade de executar uma ação ou não:

$$\hat{R}(s \mid a) = logit(w \cdot s),$$

para esse caso precisamos gerar um modelo para cada ação. O algoritmo permanece o mesmo!

Deep Q-Learning

O algoritmo Deep Q-Learning aplica as mesmas ideias acima porém substituindo a Regressão Linear/Logística por uma Rede Neural.

Para a modelagem como problema de classificação costuma-se usar a função de ativação *softmax* na camada de saída.

Deep Q-Learning

Não se esqueçam que os atributos de entrada de uma Rede Neural (e das regressões) não precisam ser necessariamente o estado s.

Para o projeto, quais outros atributos podemos criar baseado na informação s fornecida?

Ex.: distância entre o Mario e o inimigo mais próximo.

Do the evolution

Recompensas de longo termo

A abordagem anterior apresenta alguns problemas como:

- dependência da possibilidade de medir uma recompensa instantânea.
- a ação executada idealmente não pode influenciar em recompensas futuras.

Recompensas de longo termo

No nosso projeto percebemos que uma ação a executada em um estado s pode levar a um estado s' em que não existam opções válidas, exceto perder o jogo.

Isso acontece ao pular em direção a um adversário em que o agente recebe a recompensa de seguir em frente por três instantes de tempo para então perder o jogo.

Recompensas de longo termo

Uma forma de aliviar esse problema nos algoritmos anteriores é atualizar os valores das recompensas obtidas utilizando a informação de quão longe o agente foi naquele determinado episódio.

Outra estratégia é levar em conta apenas o desempenho final do agente e não a cada instante t.

Aprendizado Supervisionado sem supervisão

Mas, para ajustar os pesos nossos algoritmos dependem da informação de gradiente.

Essa informação de gradiente requer o valor da recompensa a cada instante de tempo e não apenas um valor único para o episódio inteiro.

Métodos de Otimização sem gradiente

Uma solução são os algoritmos de otimização que não utilizam a informação de gradientes.

Dentre esses algoritmos, destacam-se os baseados em Computação Evolutiva.

Computação Evolutiva

Inspirados na evolução das espécies e conhecidas como Meta-heurísticas de busca.

Meta-heurística: que define heurísticas.

Ou seja, algoritmos que definem algoritmos.

Meta-heurística básica:

- Cria um conjunto de soluções candidatas P
- Avalia essas soluções P
- Aplica um algoritmo que altera essas soluções *um pouquinho* (mutação), armazena em P'
- Avalia P'
- Seleciona $|P|$ soluções com probabilidade proporcional a qualidade delas para substituir P e repete o processo

A ideia básica é que o procedimento de mutação em uma solução p_i busque por soluções vizinhas melhores.

O processo de seleção direciona a busca para os ótimos locais próximos aos pontos selecionados.

Neuroevolução

Retomando nosso problema de ajustar os pesos de uma rede neural para determinar as melhores ações de um agente.

Podemos pensar como um problema de otimização e busca em que queremos o melhor conjunto de valores w^* que maximiza a recompensa final do agente.

Neuroevolução

Então dada uma estrutura pré-determinada da rede neural, faça:

- Cria um conjunto de n vetores w candidatos W
- Execute o agente com a rede neural para cada peso de W e determine a recompensa final
- Aplique a função de mutação em cada w criando W'
- Avalie W'
- Selecione n soluções de $W + W'$ através de um torneio aleatório
- Repita por t iterações

Mutação I

Um tipo tradicional de mutação para esse tipo de problema é:

$w' = w + \mathcal{N}(0, 1)$

Ou seja, soma cada valor do vetor w a um valor aleatório com distribuição gaussiana.

Mutação II

Podemos também sortear uma intensidade para esses valores:

$w' = w + U(0,1) \cdot \mathcal{N}(0,1)$

Sendo $U(0,1)$ um valor aleatório entre 0 e 1.

UFABC

Mutação III

Uma outra forma é escolher um segundo vetor de pesos w^{ref} e fazer:

$w' = \alpha w + (1 - \alpha) w^{ref}$

Sendo $\alpha = U(0, 1)$ um valor aleatório entre 0 e 1. Nesse caso temos a combinação linear entre duas soluções. Essa operação também é conhecida como *cruzamento*.

Mutação III

A seleção de w^{ref} pode ser feita da mesma forma que a seleção por torneio, como será visto adiante.

UFABC

Mutação IV

Podemos também fazer uma aproximação do gradiente:

- Gere n vetores $w' = w + \alpha \cdot \mathcal{N}(0, 1)$
- Avalie esses vetores, sendo $f_i = R(w_i')$ a recompensa final de w_i'
- Faça $w = w + \alpha E(f \cdot w)$, com $\alpha \in [0, 1]$ e $E(.)$ a média dos vetores.

UFABC

Seleção I

Uma possível seleção é a Roleta Aleatória:

- Crie uma roleta em que cada solução i tem uma fatia de tamanho $f_i / \sum_j f_j$
- Gire a roleta n vezes para selecionar n soluções para a próxima iteração

Seleção II

Uma seleção com propriedades mais interessantes é o torneio:

- Defina um número k de participantes
- Repita n vezes:
  - Sorteie k soluções
  - Seleciona a solução com maior f dentre essas k e inclua no novo conjunto

Outros algoritmos

Existem diversos outros algoritmos que podem ser interessantes para o projeto:

- Particle Swarm Optimization (PSO)
- Differential Evolution (DE)
- Ant Colony Optimization (ACO)
- NeAT (evolução dos pesos e topologia de uma rede neural

Próxima aula

Não teremos mais aula :(

Mas estarei aqui para tirar dúvidas e bater um papo referentes ao projeto e do curso.

UFABC