Instituto Tecnológico de Aeronáutica

CT-213

Laboratório 12 - Aprendizado por Reforço com Aproximação de Função e Gradiente de Política

Aluno: Pedro Elardenberg Sousa e Souza
Professor: Marcos Ricardo Omena de Albuquerque Maximo

23 de agosto de 2024

Conteúdo

1 Resumo

Neste Laboratório, foi implementado o algoritmos de Aprendizado por Reforço Profundo *Deep Q-Learning / Deep Q-Networks* (DQN), e com ele resolvido o problema de subida de ladeira por um carrinho. Nesse problema clássico de aprendizado por reforço, tem-se um carro em uma montanha como pode ser visto na figura 1. No caso, o carro começa numa numa posição aleatória próxima ao "vale" mostrado na figura. O objetivo do carro é subir a montanha até o seu ponto mais alto na direita.

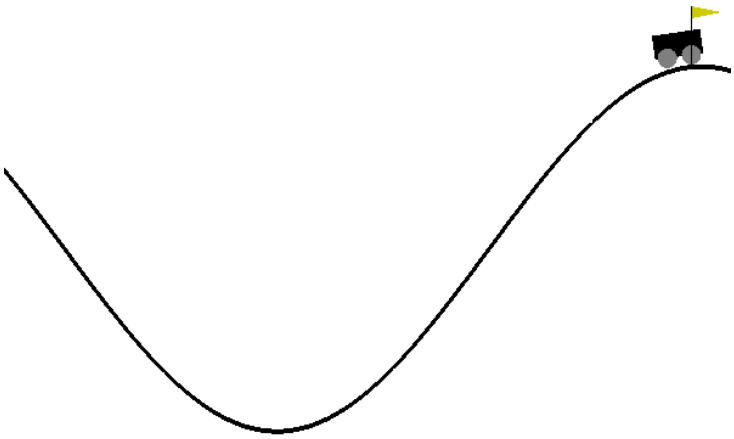

Figura 1: Representação do problema do *"Mountain car"*

Os resultados foram então mostrados e discutidos neste relatório.

2 Introdução

O carro tem o espaço de estados e de ações como mostrado nas tabelas 1 e 2, respectivamente:

Tabela 1: Espaço de estados do *Mountain Car*

Número do estado	**Estado**	**Mínimo**	**Máximo**
0	posição	-1,2	0,6
1	velocidade	-0,07	0,07

Tabela 2: Espaço de ações do *Mountain Car*

Número da ação	**Ação**
0	Empurrar para a esquerda (push left)
1	Sem empurrar (no push)
2	Empurrar para a direita (push right)

A recompensa é -1 por passo de tempo, até que o objetivo de posição 0,5 (na direita) seja atingido. O limite esquerdo da tela funciona como uma parede. O estado inicial é uma posição entre -0,6 e -0,4 com velocidade nula. Finalmente, o episódio termina quando o carro atinge 0,5 ou executa-se 200 passos de tempo no episódio, o que ocorrer primeiro.

Quanto ao algoritmo de DQN, como o nome indica, é uma versão modificada do algoritmo de *Q-Learning* para estabilizar melhor o aprendizado quando se usa uma rede neural como aproximador da função ação-valor. No caso, DQN trouxe duas inovações principais [1]:

- Uso de *experience replay:* em vez de se atualizar o algoritmo através de experiências consecutivas que ocorrem durante a interação do agente com o ambiente, como é natural em algoritmos clássicos de Aprendizado por Reforço, armazena-se as experiências (estado, ação, recompensa, novo estado) em um *replay buffer* (memória). Posteriormente, um *mini-batch* de amostras aleatórias desse *buffer* é utilizado para treinar a rede neural. Isso quebra a correlação entre as amostras usadas para treinamento da rede, o que é bom para o treinamento de redes neurais.

- *Fixed Q-targets:* durante um *mini-batch* de treinamento da rede que representa $\hat{q}(s, a)$, usa-se valores fixos de Q para estimar o *target* $R_{t+1} + \gamma Q(S_t, A_t)$. Isso evita que os *targets* mudem durante a atualização

do mini-batch, reduzindo o efeito do *target* ser não-estacionário nesse problema.

A rede usada para aproximação da função ação-valor $\hat{q}(s, a)$ tem a arquitetura apresentada na Tabela 3. Essa rede neural recebe o estado como entrada e retorna o valor da função ação-valor para cada ação. No caso do *Mountain Car*, há 3 ações, então as saídas da rede são: $\hat{q}(s, a_1)$, $\hat{q}(s, a_2)$ e $\hat{q}(s, a_3)$. Apesar de o único problema resolvido ser o do *Mountain Car*, a rede foi implementada de forma genérica em função do número de entradas e saídas para que a implementação de DQN pudesse ser usada em outros problemas.

Tabela 3: Arquitetura da rede neural usada para aproximar a função ação-valor $\hat{q}(s, a)$

Layer	**Neurons**	**Activation Function**
Dense	24	ReLU
Dense	24	ReLU
Dense	action_size	Linear

3 Análise dos Resultados

3.1 Implementação da Definição da Rede Neural

A primeira atividade realizada foi a criação da rede neural do problema, usando Keras. O método *make_model* foi implementado de acordo com a tabela 3, utilizando o método dos mínimos quadrados como *Loss Function* e a função de otimalidade *Adam*.

3.2 Escolha de Ação usando Rede Neural

Assim como no Relatório anterior [2], a escolha da ação tomada pelo agente seguiu uma política ϵ-greedy:

```
def act(self, state):
    rng = np.random.uniform(0, 1)
    possible_states = self.model.predict(state)

    if rng < 1 - self.epsilon:
        return possible_states[0].argmax(axis=0)
    return int(possible_states.shape[1]*rng)
```

3.3 *Reward Engineering*

Para que se obtenha o resultado esperado, foi utilizada uma técnica chamada *reward engineering*. Como o objetivo do carrinho é chegar ao topo da montanha, mas o agente não sabe se está indo no caminho certo, introduz-se recompensas intermediárias que auxiliam o agente a ir na direção do objetivo. Para este caso, recompensou-se o carrinho de acordo com sua proximidade do topo, isto é, mais à direita, e pela sua velocidade.

$$r_{modified} = r_{original} + (position - start)^2 + velocity^2 \tag{1}$$

Além disso, é interessante recompensar muito o agente caso ele consiga ser bem-sucedido na tarefa:

$$r'_{modified} = r_{modified} + 50 \times 1\{next\_position \geq 0.5\} \tag{2}$$

Em que $1\{next\_position \geq 0.5\}$ vale 1 se a condição dentro de for satisfeita e 0 caso contrário.

3.4 Treinamento usando DQN

Após implementadas as funções de ação e de recompensa, foi realizado o treinamento do modelo, por meio do *script train_dqn.py*. O gráfico que mostra o resultado do modelo para cada episódio é mostrado na figura 2. Para facilitar a convergência do algoritmo, usou-se um esquema de *schedule* para o ϵ, da política ϵ-greedy, da sequência forma:

$$\epsilon_e = max(\epsilon_{min}, \epsilon_0 d^{e-1}) \tag{3}$$

Em que ϵ_e é o valor de ϵ no episódio e, ϵ_0 é o valor inicial de ϵ, d é o fator de decaimento e ϵ_{min} é um valor mínimo de ϵ para garantir um mínimo de exploração no final do treinamento.

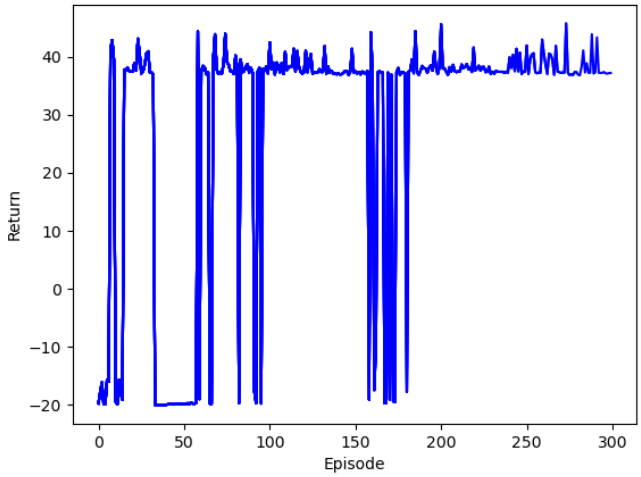

Figura 2: Resultados da função retorno do algoritmo *mountain car DQN* ao longo de 300 episódios

Percebe-se da figura que o modelo consegue realizar a tarefa já nos primeiros 20 episódios e que, após o episódio 60, o modelo consistentemente realiza a tarefa. Os pontos marcados com retorno próximo de 40 indicam que a atividade de subida da montanha foi cumprida com êxito pelo carrinho. Isso está de acordo com as previsões de efetividade do modelo para problemas dessa natureza.

3.5 Avaliação da Política

Realizado o treinamento, foi feita a avaliação de política, a partir do *script* *evaluate_dqn.py*. O resultado da avaliação é mostrado na figura 3

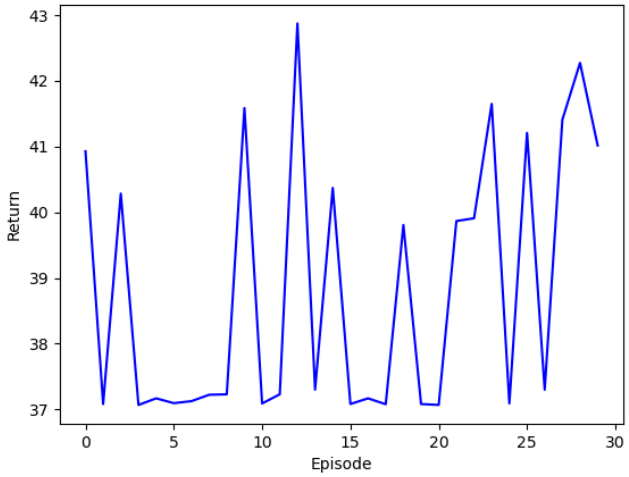

Figura 3: Resultados da avaliação de política do modelo para 30 casos gerados

Considerou-se que a avaliação foi bem-sucedida se os resultados fossem positivos em pelo menos 70% das avaliações. Como, na figura 3, todos os 30 pontos estão próximos a um retorno de 40, em todos os episódios dessa avaliação de política o agente realizou a atividade com sucesso.

4 Conclusão

O presente laboratório mostrou que o algoritmo *Deep Q-Networks* (DQN) é capaz de ser utilizado na resolução de problemas quando não se tem o modelo nem os resultados da função ação-valor desse modelo e para redes neurais profundas.

O algoritmo pôde ser treinado de forma a realizar a tarefa pretendida, de maneira consistente a partir de aproximadamente 60 iterações, chegando a resultados consistentes (recompensa de aproximadamente 40) na maioria das iterações após isso.

Referências

[1] Marcos Ricardo Omena de Albuquerque Maximo. Ct-213 - aula 13 - aprendizado por reforço com aproximação de função e gradiente de política. Apostila, 2020. Curso CT-213 - Inteligência Artificial para Robótica Móvel, Instituto Tecnológico de Aeronáutica.

[2] Pedro Elardenberg Sousa e Souza. Laboratório 11 - aprendizado por reforço livre de modelo. Relatório, 2024. Curso CT-213 - Inteligência Artificial para Robótica Móvel, Instituto Tecnológico de Aeronáutica.