**Instituto Tecnológico de Aeronáutica – ITA**

**Inteligência Artificial para Robótica Móvel – CT-213**

**Professor: Marcos Ricardo Omena de Albuquerque Maximo**

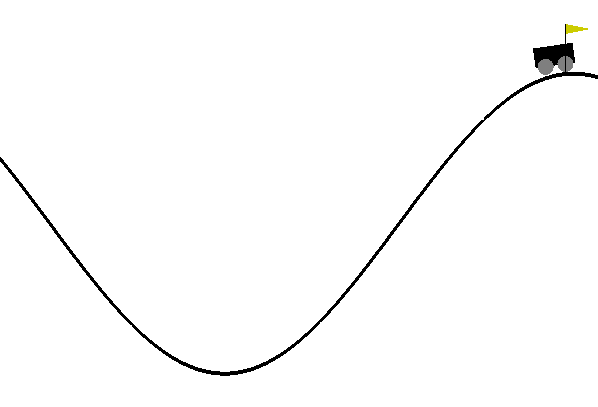
**Laboratório 12 (Aula 13) – Deep Q-Learning**

**Observação:**

* Para realizar esse laboratório, é necessário instalar o OpenAI Gym usando o pip. Abra o terminal do Anaconda e digite: pip install gym.

1. **Introdução**

Nesse laboratório, seu objetivo é resolver o problema de *Mountain Car* usando o algoritmo seminal de Deep Reinforcement Learning: o Deep Q-Learning/Deep Q-Networks (DQN). A Figura 1 mostra o problema do *Mountain Car* no ambiente OpenAI Gym, que também será utilizado nesse laboratório.



**Figura 1:** trajetória do robô seguidor de linha após aprendizado com Q-Learning.

1. **Descrição do Problema**

O problema ser resolvido é o aprendizado da política no problema chamado *Mountain Car*. A seguir, descreve-se formalmente o problema (conforme pode ser visto em <https://github.com/openai/gym/wiki/MountainCar-v0>). 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. O carro tem o espaço de estados e de ações como mostrado nas tabelas 1 e 2, respectivamente.

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

**Tabela 1:** espaço de estados 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*) |

**Tabela 2:** espaço de ações do *Mountain Car*.

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 o algoritmo de DQN, como o nome indica, ele 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:

* Uso de *experience replay*: ao invés 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 , usa-se valores fixos de para estimar o *target* . 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 deve ter a arquitetura apresentada na Tabela 3. Perceba que 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: , e . Apesar de que o único problema resolvido será o *Mountain Car*, implemente a rede de forma genérica em função do número de entradas e saídas para que sua implementação de DQN possa ser usada em outros problemas. Lembre de definir o tamanho da entrada da rede como o tamanho do estado na primeira camada usando o argumento input\_dim.

| **Layer** | **Neurons** | **Activation Function** |
| --- | --- | --- |
| Dense | 24 | ReLU |
| Dense | 24 | ReLU |
| Dense | action\_size | Linear |

**Tabela 3:** arquitetura da rede neural usada para aproximar a função ação-valor .

1. **Código Base**

O código base já implementa a lógica para treinar e avaliar o DQN no problema do *Mountain Car*. Segue uma breve descrição dos arquivos fornecidos:

* dqn\_agent.py: implementação do DQN propriamente dito.
* train\_dqn.py: treinar um agente usando DQN no problema *Mountain Car*.
* evaluate\_dqn.py: treinar o agente baseado em DQN no problema *Mountain Car*.
* utils.py: funções utilitárias.

O foco da sua implementação nesse laboratório são os arquivos dqn\_agent.py e utils.py.

1. **Tarefas**
   1. **Implementação da Definição da Rede Neural**

A primeira implementação é a definição da rede neural para usando Keras. Para isso, no método build\_model() da classe DQNAgent de dqn\_agent.py, implemente o modelo apresentado na Tabela 3 usando Keras. A dimensão do *feature vector* do estado e o número de ações são dados pelas variáveis de classe state\_size e action\_size, respectivamente. Como função de custo da rede neural, use erro quadrático médio (*Mean Squared Error* - MSE). Compare o sumário do modelo implementado com o que é mostrado na Tabela 3. Adicione o sumário no seu relatório.

* 1. **Escolha de Ação usando Rede Neural**

Implemente o método act() da classe DQNAgent. Conforme destacado anteriormente, no caso do *Mountain Car*, para cada estado , a rede fornece como saídas , e . Assim, implemente uma escolha de ação -greedy nesse contexto. Porém, implemente de forma geral para uma quantidade (finita) qualquer de ações. O é a variável de classe epsilon. Perceba que a saída do modelo do Keras no caso é um array de dimensão (1, 3) de NumPy. No caso, state já está no formato esperado pelo método predict(). O retorno do método deve ser o número da ação escolhida.

* 1. ***Reward Engineering***

*Mountain Car* da forma como é definido originalmente é um problema particularmente difícil para Aprendizado por Reforço, pois um agente recebe recompensa relevante apenas quando é bem-sucedido na tarefa (atinge o topo da montanha). Com isso, até que o agente tenha sido capaz de atinigir o topo da montanha, ele permanece “às escuras” enquanto explora, sem saber se está melhorando seu desempenho. Dessa forma, seu agente provavelmente não será capaz de aprender *Mountain Car* apenas com o que você implementou até o momento.

Há diversas tarefas de Aprendizado por Reforço em que efeitos semelhantes a esse acontecem (e.g. treinar um robô para chutar a bola o mais distante possível também sofre de um problema parecido). Nesses casos, em geral é conveniente criar recompensas intermediárias “artificiais” para facilitar o aprendizado do agente. Costuma-se chamar essa heurística de *reward engineering*. No caso do *Mountain Car*, pode-se pensar o seguinte: queremos recompensar o carro por ficar longe do “vale” onde ele começa, mesmo que não consiga subir a montanha. Além disso, parece interessante recompensar ele pode estar se movendo rápido. Dessa forma, verifica-se que a seguinte recompensa modificada ajuda muito o carro a aprender mais rapidamente nesse problema:

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

em que vale 1 se a condição dentro de for satisfeita e 0 caso contrário.

Para implementar essa *reward engineering*, utilize o método reward\_engineering\_mountain\_car() de utils.py. Tem-se o seguinte mapeamento entre conceitos apresentados aqui e as variáveis no código:

* : reward.
* : state[0].
* : state[1].
* : next\_state[0].
* : START\_POSITION\_CAR (variável global no utils.py).
  1. **Treinamento usando DQN**

Basta rodar o script train\_dqn.py. O treinamento demora um tempo considerável. Se sua implementação estiver correta, seu agente deve completar a tarefa pelo menos uma vez até 100 episódios e possivelmente ter um bom desempenho após 300 episódios de treinamento. A cada 20 episódios, um gráfico do retorno da tarefa ao longo dos episódios é exibido para verificar o funcionamento do treinamento. Este gráfico também é salvo em disco. Ademais, perceba que para facilitar a convergência do algoritmo, usou-se um esquema de *schedule* para o da sequência forma:

em que é o valor de no episódio , é o valor inicial de , é o fator de decaimento e é um valor mínimo de para garantir um mínimo de exploração no final do treinamento. Perceba que a rede é salva a cada 20 episódios, de modo que mesmo que o treinamento seja parado prematuramente, ele pode ser reiniciado depois (porém, será resetado). Caso queira esquecer os pesos de um treinamento anterior, basta apagar o arquivo mountain\_car.h5. Finalmente, destaca-se que os hiperparâmetros já foram ajustados para esse problema, então não há necessidade de alterá-los. Inclua no seu relatório o gráfico do treinamento para 300 episódios. Discuta seus resultados.

Se não estiver satisfeito com o desempenho do seu agente, rode o treinamento novamente. Às vezes, a política piora após treinar por mais tempo. Caso prefira, você pode também ficar acompanhando o treinamento e parar quando estiver satisfeito com o desempenho. Por favor, entregue o arquivo mountain\_car.h5 da sua rede para avaliação.

* 1. **Avaliação da Política**

Basta rodar o script evaluate\_dqn.py. Inclua no seu relatório os gráficos gerados por esse *script*. Discuta seus resultados. Como há ruído no estado inicial, nem sempre o agente consegue chegar no objetivo. Assim, considere sucesso se seu agente estiver conseguindo completar a tarefa em pelo menos 70% dos casos (i.e. 21 de 30).

1. **Entrega**

A entrega consiste do código e de um relatório, submetida através do Google Classroom. Modificações nos arquivos do código base são permitidas, desde que o nome e a interface dos scripts “main” não sejam alterados. A princípio, não há limitação de número de páginas para o relatório, mas pede-se que seja sucinto. O relatório deve conter:

* Breve descrição em alto nível da sua implementação.
* Figuras que comprovem o funcionamento do seu código.

Por limitações do Google Classroom (e por motivo de facilitar a automatização da correção), entregue seu laboratório com todos os arquivos num único arquivo **.zip** (**não** utilize outras tecnologias de compactação de arquivos) com o seguinte padrão de nome: “**<login\_email\_google\_education>\_labX.zip**”. Por exemplo, no meu caso, meu login Google Education é **marcos.maximo**, logo eu entregaria o lab 12 como “**marcos.maximo\_lab12.zip**”. **Não** crie subpastas para os arquivos da sua entrega, **deixe todos os arquivos na “raiz” do .zip**. Os relatórios devem ser entregues em formato **.pdf**.

1. **Dicas**

* Caso tenha dúvida em como usar o Keras, relembre nos laboratórios anteriores.
* Para pegar o índice do máximo elemento de certo array **unidimensional** em numpy, faça:

index = np.argmax(array)