



APRENDIZADO POR REFORÇO

Aula 5: Function Approximation and Deep Reinforcement Learning

Lucas Pereira Cotrim
Marcos Menon José

lucas.cotrim@maua.br
marcos.jose@maua.br

Dúvidas Exercício E_3

Exercício E_3

76

- Tarefa A)
Implementar o Agente Q-Learning
- Tarefa B)
Implementar o Loop de Treino
- Tarefa C)
Implementar o Agente Double Q-Learning

Algoritmo: Double Q-Learning (off-policy TD control)

Parâmetro do Algoritmo: Taxa de aprendizado $\alpha \in (0,1]$

Inicializar $Q_1(s, a)$ e $Q_2(s, a)$ arbitrariamente para todo $s \in \mathcal{S}$ e $a \in A(s)$, com $Q_0(s_{termina}) = 0$ para estados terminais.

Repetir para cada episódio:

 Inicializar estado inicial $S_0 \sim \mathbb{P}(S_0 = s), s \in \mathcal{S}$

 Repetir para cada timestep $t = 0, 1, 2, \dots$:

 Escolher ação a de s usando a política derivada de Q (como: ϵ -greedy)

 Executar ação A_t e observar R_{t+1}, S_{t+1}

 Com probabilidade 0.5:

$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_{a \in \mathcal{A}} Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)]$

 Else:

$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_1(S_{t+1}, \arg \max_{a \in \mathcal{A}} Q_2(S_{t+1}, a)) - Q_2(S_t, A_t)]$

$S_t = S_{t+1}$

 Até que S_t seja um estado terminal

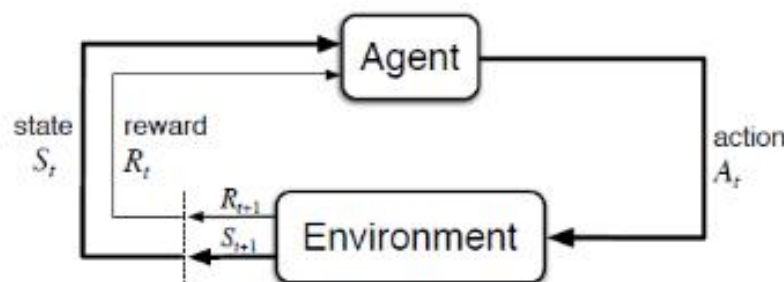
Retorna: Funções Valor e Política ótimas V^* , Q^* e π^*

Recapitulando

APRENDIZADO POR REFORÇO: DEFINIÇÃO

9

- Área de Aprendizado de Máquina associada a como agentes devem escolher ações em determinado ambiente com o objetivo de maximizar recompensas.
- Características do problema:
 - Agente possui sensores que observam o **estado** do **ambiente**.
 - O agente possui um conjunto de possíveis **ações** que pode tomar para alterar esse estado.
 - A cada ação tomada ele recebem uma **recompensa** que indica a qualidade da ação dado o estado.



Recapitulando: *Model Free Control*

Model-Free Control

- Model-Free Prediction:

Estimar Função Valor $V_\pi(s)$, $Q_\pi(s, a)$ dada uma política π atuando em um MDP desconhecido.

- Model-Free Control:

Obter Função Valor Ótima $V^*(s)$, $Q^*(s, a)$ e política ótima π^* de um MDP desconhecido.

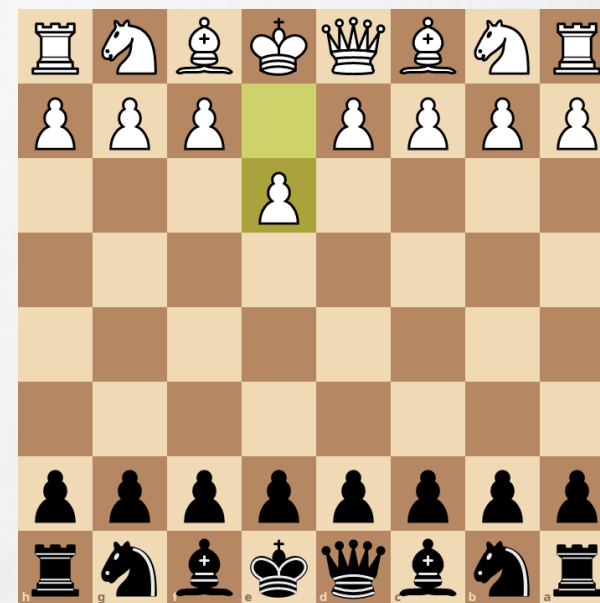
Em Programação Dinâmica vimos como obter $V^*(s)$ quando o modelo $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ é conhecido (Model-Based). Em Model-Free Control os métodos são análogos, mas a experiência deve ser obtida por interação com o ambiente, pois não conhecemos \mathcal{P}, \mathcal{R} .

Function Approximation

Function Approximation

Necessidade de utilizar métodos de aproximadores de função

- Até agora, estudamos métodos tabulares nos quais era possível criar uma tabela para as funções valor $V(s)$ e $Q(s, a)$. Mas como faríamos nos casos:
 - Xadrez: 10^{131} (existem “apenas” 10^{80} átomos no universo observável, ou seja, há 10^{51} vezes mais possibilidades de posições no xadrez)
 - Go: 10^{171}
 - Carro autônomo: espaço das ações contínuo



Fonte: lichess.org

Model-Free Control Métodos Tabulares

- Até agora representamos a Função Valor de forma tabular:
 - Cada estado é um valor na tabela $V(s)$
 - Cada par estado/ação é um valor na tabela $Q(s,a)$
- Como ficaria com um MDP muito grande?
 - Muito difícil de guardar na memória
 - Muito devagar para aprender o valor de estado individualmente

Exemplo de $Q(s,a)$ da aula anterior

	a_1	a_2
0	1,8	-2
1	-1	0
2	0	0
3	20	-20
4	0	0
5	0	0

Aproximação da função valor

- Como lidar com o problema da dimensionalidade?
- Uma solução é criar métodos de aproximação para as funções valor $V(s)$ e $Q(s, a)$.

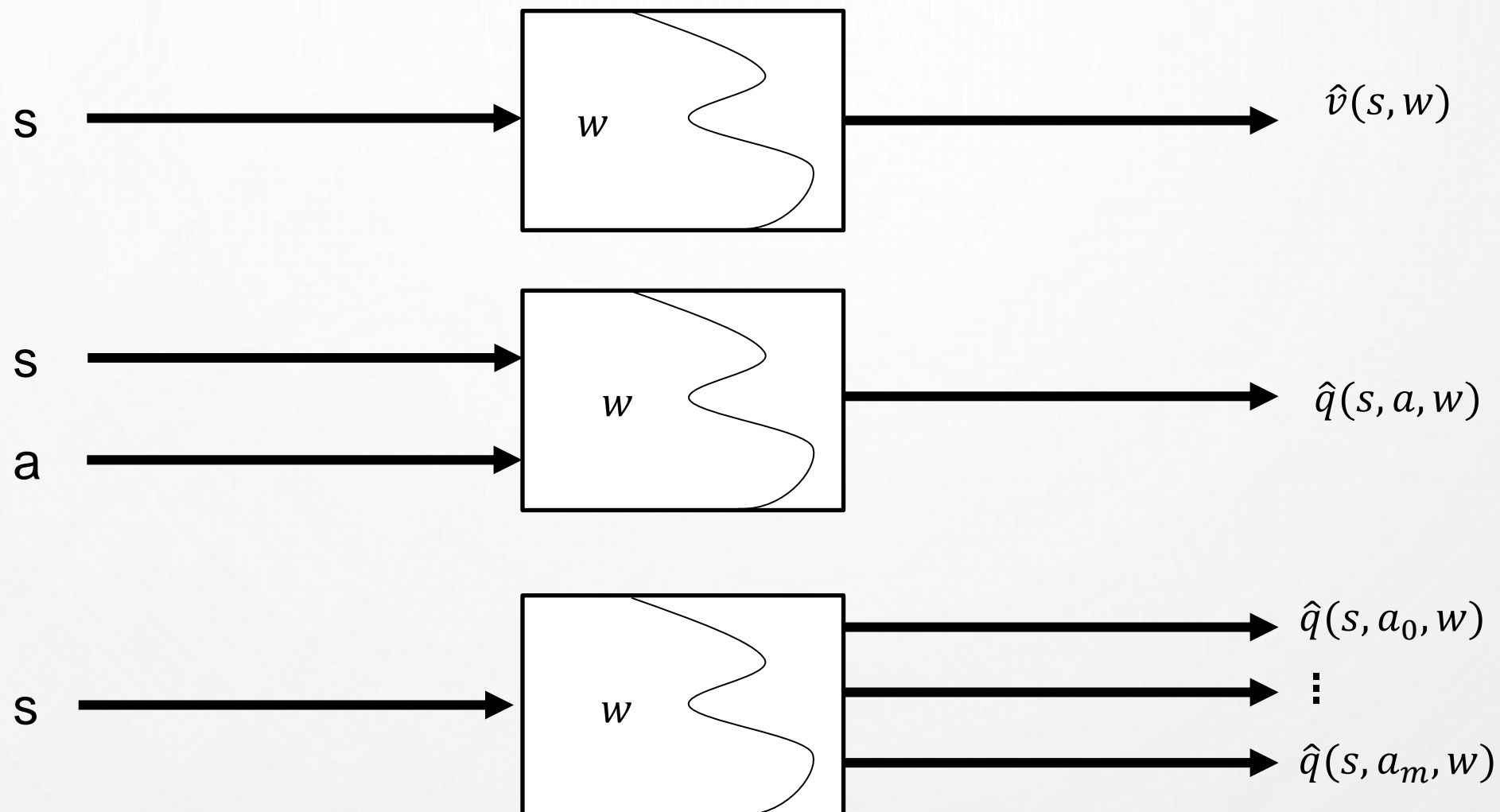
$$\begin{aligned} V_{\pi}(s, \mathbf{w}) &\approx V_{\pi}(s) \\ Q_{\pi}(s, a, \mathbf{w}) &\approx Q_{\pi}(s, a) \end{aligned} \quad \mathbf{w} \rightarrow \text{Pesos/Weights do aproximador de função}$$

- O aproximador da função valor pode ser qualquer método de regressão: regressão linear, lasso, *decision tree* e redes neurais (mais utilizado).
- Em vez de atualizar os valores $V(s)$ e $Q(s, a)$ diretamente nas tabelas, usamos os métodos vistos para atualizar os pesos \mathbf{w} .

Vantagens da Aproximação da função valor

- Contorna problemas de dimensionalidade
 - Reduz memória necessária
 - Reduz o tempo de computação necessário
 - Reduz a experiência necessária para achar a função valor ótima.
- Permite a generalização para estados não vistos a partir do treino de estados similares

Tipos de Aproximação da função valor



$w \rightarrow$ Pesos/*Weights* do aproximador de função

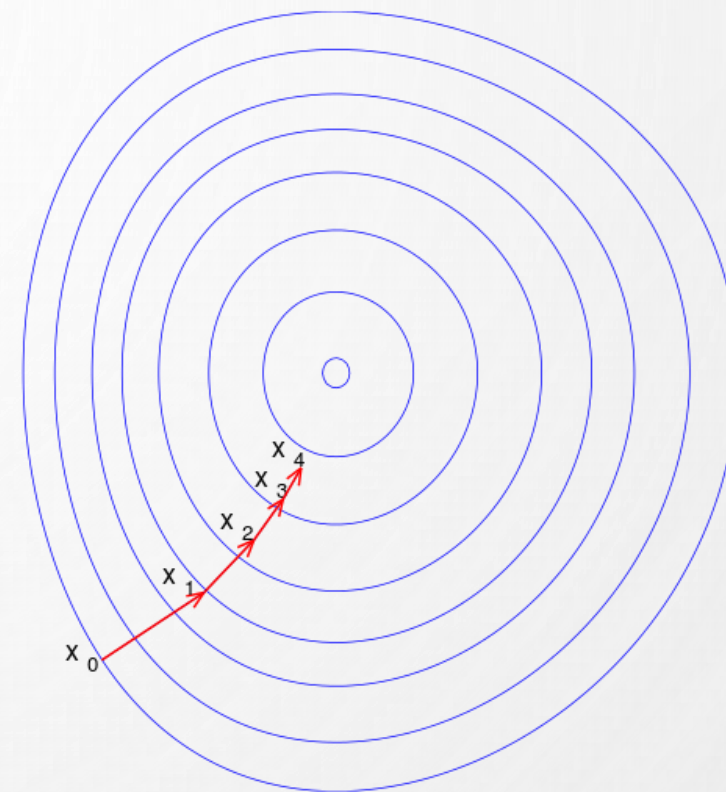
Stochastic Gradient Descent

Stochastic Gradient descent

Método do Gradiente descendente

Stochastic Gradient Descent

- *Stochastic Gradient Descent* é um método numérico de otimização que serve para encontrar o valor mínimo local de uma função iterativamente
- A cada época, o algoritmo toma o passo em direção ao declive negativo máximo do gradiente
- É muito utilizado em *Machine learning* desde algoritmos lineares (como *Support Vector Machines*), regressão logística até em *backpropagation* de redes neurais artificiais.



Fonte:

https://pt.wikipedia.org/wiki/M%C3%A9todo_do_gradiente

Stochastic Gradient Descent

- Seja $J(w)$ uma função diferenciável pelos pesos w (*weights*) que queremos encontrar o mínimo local, temos que seu gradiente é:

$$\nabla_w J(w) = \begin{pmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{pmatrix} \quad \text{Sendo } n \text{ o número de pesos } w \text{ da função } J$$

- A cada época w é ajustado utilizando:

$$w \leftarrow w + \Delta w \quad \Delta w = -\frac{1}{2} \alpha \nabla_w J(w) \quad \text{Sendo } \alpha \text{ a } \textit{Lerning Rate}$$

Stochastic Gradient Descent da função Valor \hat{v}

- Objetivo é encontrar os pesos w que minimizam o erro entre a aproximação da função valor $\hat{v}(s, w)$ com os valores reais de $v_\pi(s)$. Em geral se utiliza o *minimum square error*, mas outras funções poderiam ser utilizadas:

$$J(w) = \mathbb{E}[(v_\pi(S) - \hat{v}(S, w))^2]$$

- Se utiliza o *gradient descent* para encontrar o mínimo local:

$$\Delta_w = -\frac{1}{2}\alpha \nabla_w J(w)$$

- Substituindo o nosso $\nabla_w J(w)$:

$$\Delta_w = \alpha(v_\pi(S) - \hat{v}(S, w))\nabla_w \hat{v}(S, w)$$

Feature Vector

- Com os métodos tabulares, tínhamos um valor $V(s)$ para cada estado s
- Agora temos um aproximador de função e com isso podemos utilizar um *Feature Vector* $x(s)$ para representar um estado s .
- Exemplos:
 - Configuração do tabuleiro de Xadrez
 - Tela do *atari* (ou uma representação da tela)
 - Indicadores, estatísticas e valores passados de uma ação no caso de um robô *trader*

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Vantagens de utilizar *Feature Vector*

- Permite a generalização para estados similares
- Diminui o número de variáveis pois o número de *features* que representa o estado é em geral menor que o número de estados
- Permite usar conhecimento prévio/humano para ajudar o agente a aprender o ambiente (*Hand Crafted Features*):
 - No caso do xadrez, podemos colocar informações como a soma dos valores das peças (dama vale 9, torre 5, etc) .
 - No caso de um carro autônomo, podemos colocar além as informações dos sensores e imagens capturadas como também cálculos de distância

V function approximation

- Métodos que estudamos até agora podem ser implementados com aproximadores de função
- Ainda assim, não são amplamente utilizados porque tem alto custo computacional

Monte Carlo Control com *Value Function Approximation*

- Utilizar o retorno G_t como uma amostra de $v_\pi(S_t)$
- Tira-se a *minimum square error* como *loss*:

$$J(w) = \mathbb{E}[(G_t - \hat{v}(S, w))^2]$$
- Realiza-se o *gradient descent* iterativamente de acordo com cada episódio:

$$\Delta w = \alpha (G_t - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$$

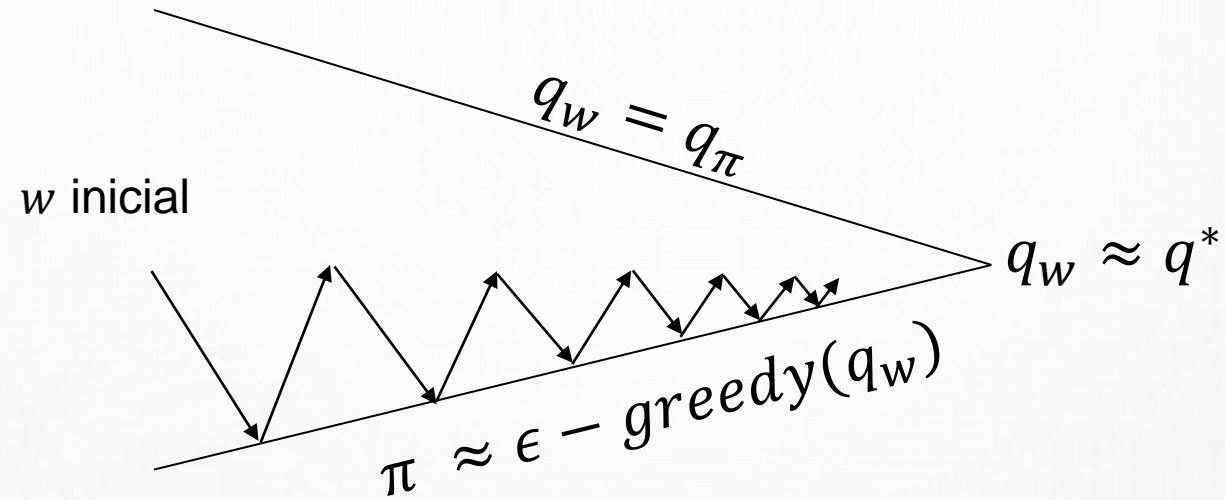
TD Learning com *Value Function Approximation*

- Utilizar $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$ como uma amostra de $v_\pi(S_t)$
- Tira-se a *minimum square error* como *loss*:

$$J(w) = \mathbb{E}[(R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S, w))^2]$$
- Realiza-se o *gradient descent* iterativamente de acordo com cada timestep:

$$\Delta w = \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$$

Model-Free Control with function approximation



A cada timestep:

- *Approximate Policy Evaluation:* $q_w \approx q_\pi$
- *Policy Improvement:* ϵ -greedy Policy Improvement

Model-Free Control with function approximation

- De forma análoga à aproximação da função valor V , podemos aplicar a aproximação da função valor para a função valor Q dos pares estado/ação

$$\hat{q}(s, a, w) \approx q_{\pi}(s, a)$$

- Minimizar o erro aproximado entre $\hat{q}(s, a, w)$ e $q_{\pi}(s, a)$. Esse erro pode ser representado por diversas *loss functions* diferentes. No caso vamos utilizar a mais usada *minimum square error*:

$$J(w) = \mathbb{E}_{\pi} (q_{\pi}(s, a) - \hat{q}(s, a, w))^2$$

- Usar *stochastic gradient descent* para encontrar o mínimo local

$$-\frac{1}{2} \nabla_w J(w) = (q_{\pi}(s, a) - \hat{q}(s, a, w)) \nabla_w \hat{q}(s, a, w)$$

$$\Delta w = \alpha (q_{\pi}(s, a) - \hat{q}(s, a, w)) \nabla_w \hat{q}(s, a, w)$$

Deep Reinforcement Learning

Deep Reinforcement Learning

Vantagens de utilizar uma abordagem neural

Deep Reinforcement Learning

- É um aproximador de função compatível com o *gradient descent* e aprendizado iterativo.
- Consegue aprender dinâmicas complexas do ambiente.
- Pode receber de entrada imagens, vídeo, sons, textos.
- Existem métodos para lidar com diferentes tipos de entradas.
- É possível realizar o *transfer learning* e utilizar redes pré-treinadas. Por exemplo utilizar a VGG16 para imagens ou a rede neural BERT para linguagem natural.

Bibliotecas de Aprendizado por Reforço Profundo

- Existem diversas bibliotecas de aprendizado por reforço e não há uma dominante
- Aprendizado por reforço é muito recente e os algoritmos estão se desenvolvendo muito rápido (as bibliotecas rapidamente ficam atrasadas)
- O *environment*/ambiente tem que ser adaptado para o uso da biblioteca (maioria usa o padrão gym, mas há diferenças)

Bibliotecas de Aprendizado por reforço



Keras RL2

- Versão do Keras RL para tensorflow2
- Uma das mais populares
- Sem suporte atualmente
- Funciona com Keras
- Pouca documentação (possível utilizar a do Keras RL)
- <https://github.com/wau/keras-rl2>



OpenAi Baselines

- Desenvolvida pela OpenAI
- Utilizado para comparar seu algoritmo com outros
- Apenas suporte e sem desenvolvimento
- Não muito customizável
- Funciona com Tensorflow 1
- <https://github.com/openai/baselines>



Stable Baselines 3

- Desenvolvida pela Inria e ParisTech
- Bastante utilizado
- Atualizações constantes
- Boa customização
- Funciona com Tensorflow 1 e 2 e pytorch
- <https://stable-baselines.readthedocs.io/en/master/>

Bibliotecas de Aprendizado por reforço



TF Agents

- Desenvolvida pela equipe do Tensorflow mas não oficial
- Updates Regulares
- Bastante utilizada
- Funciona com Tensorflow 1 e 2
- <https://www.tensorflow.org/agents>



REINFORCE.IO

Tensorforce

- Open Source
- Muitos dos principais algoritmos
- Desenvolvimento regular
- Não muito customizável
- Funciona com Tensorflow 1
- <https://github.com/tensorforce/tensorforce>



RLlib

- Desenvolvida pela equipe do ray
- Uma das mais completas bibliotecas com maior número de opções
- Bem recente
- Não muito customizável
- Funciona com Tensorflow 1 e 2 e pytorch e ray
- <https://docs.ray.io/en/master/rllib.html>

Naive Deep Q-Network

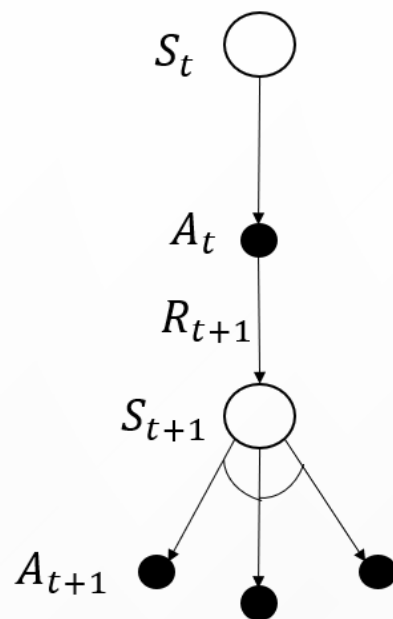
Naive Deep Q-Network

Naive DQN

Relembrando a aula anterior: Q-Learning

Q-Learning

47



Q-Learning target

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. PhD Thesis, University of Cambridge, England.

Naive DQN

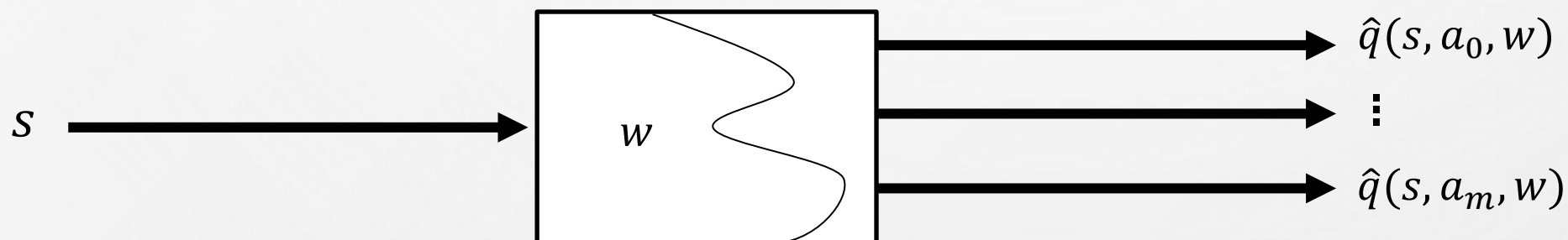
Abordagem Tabular

Tabela Q

	a_0	...	a_m
s_0			
\vdots			
s_n			

Abordagem Neural

Rede Neural



Sendo que o estado s é representado pelo *feature vector* $x(s)$

Naive DQN

Ao invés de realizar o update da tabela Q, é necessário realizar o update dos pesos w que realizam a estimativa dos valores de $q(s, a)$

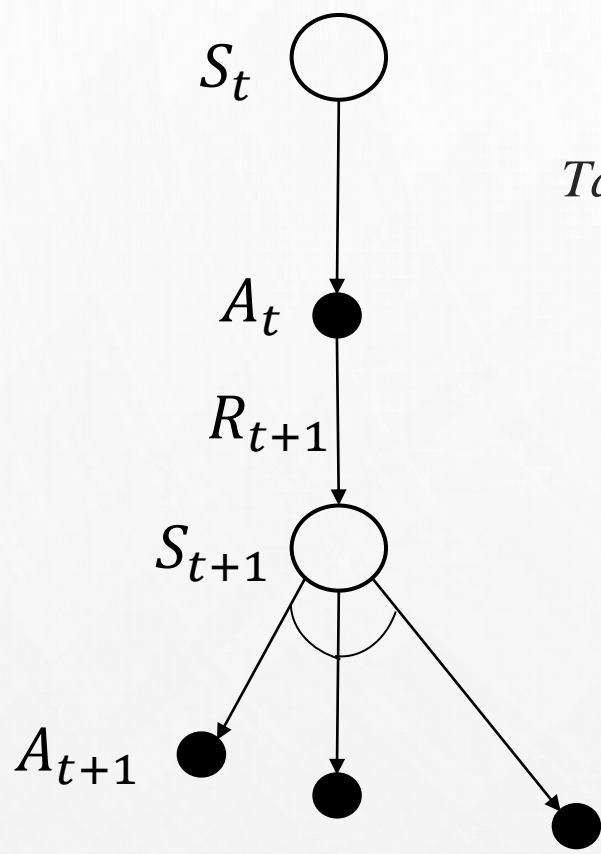
Target:

$$y = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w)$$

Loss function: minimum square error do target com o que obtivemos

$$L(w) = \left(\underbrace{R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w)}_{\text{Target}} - \hat{q}(S_t, A_t, w) \right)^2$$

Target



Update dos pesos da rede:

$$\Delta w = \alpha \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w) \right) \nabla_w \hat{q}(S_t, A_t, w)$$

Naive DQN definição

O algoritmo é definido pela atualização dos pesos da rede neural w a cada *timestep*

$$L(w) = \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w) \right)^2$$

$$\Delta w = \alpha \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w) \right) \nabla_w \hat{q}(S_t, A_t, w)$$

Onde:

- S_t : Estado atual
- A_t : Ação anterior
- α : Taxa de aprendizado (entre 0 e 1)
- R_{t+1} : Recompensa do par estado ação (S_t, A_t)
- Δ_w : Atualização dos pesos w
- $L(w)$: *loss de minimum square error*
- γ : Fator de desconto
- $\max_a \hat{q}(S_{t+1}, a, w)$ maior valor alcançável de \hat{q} para o estado futuro considerando cada uma das possíveis futuras ações

Naive DQN

Algoritmo: Naive DQN

Parâmetro do Algoritmo: Taxa de aprendizado $\alpha \in (0,1]$

Inicializar pesos w arbitrariamente

Repetir para cada episódio:

 Inicializar estado inicial $S_0 \sim \mathbb{P}(S_0 = s), s \in \mathcal{S}$

 Repetir para cada timestep $t = 0, 1, 2, \dots$:

 Escolher ação a de s usando a política derivada da rede neural $\hat{q}(s_t, w)$ (como: ϵ -greedy)

 Executar ação A_t e observar R_{t+1}, S_{t+1}

 Caso S_{t+1} seja estado Terminal:

 Target $y_i = R_{t+1}$

 Else:

 Target $y_i = R_{t+1} + \gamma \max_{a_{t+1}} \hat{q}(s_{t+1}, a_{t+1}, w)$

 Fazer o gradient descent usando a loss $(y_i - \hat{q}(S_t, A_t, w))^2$ fazendo update dos

 valores w : $\Delta w = \alpha \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w) \right) \nabla_w \hat{q}(S_t, A_t, w)$

$S_t = S_{t+1}$

 Até que S_t seja um estado terminal

Retorna: Aproximações da Funções Valor e Política ótimas \hat{q}^* e π^*

Naive DQN -> Colab: Beer_Game_Naive_DQN.ipynb



Beer_Game_Naive_DQN.ipynb ☆

Arquivo Editar Ver Inserir Ambiente de execução Ferramentas Ajuda

Comentário Compartilhar

RAM Disco Edit

Índice

- Exemplo de aula: Beer game environment com DQN
- Explicação do Environment
- Imports
- Environment Class
- Funções Auxiliares
- DQN
 - Network class
 - DQN Agent
- Loop de treino para demanda constante
- Seção

+ Código + Texto

Exemplo de aula: Beer game environment com DQN

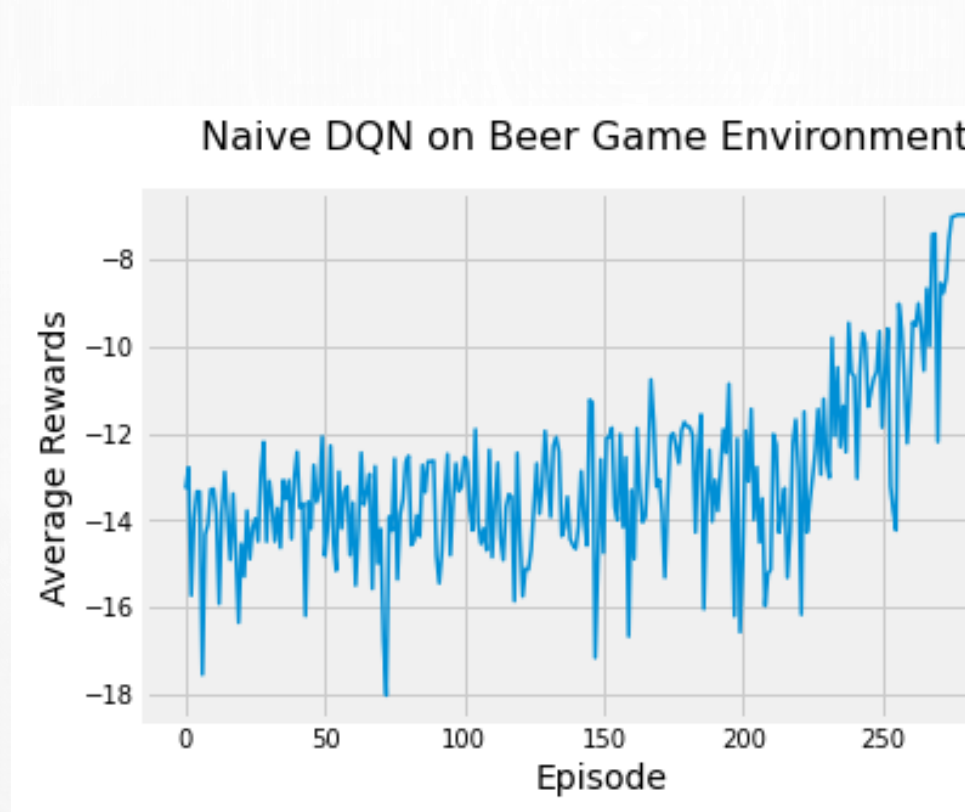
Explicação do Environment

MDP EXAMPLE: SUPPLY CHAIN 58

- Gestão de Cadeia Logística (Supply Chain Management):

Considere o problema de manutenção de inventário de determinado produto, em que cada nível da cadeia mantém uma quantidade do produto e fornece para o nível inferior (Beer Distribution Game)

Naive DQN -> Colab: Beer_Game_Naive_DQN.ipynb



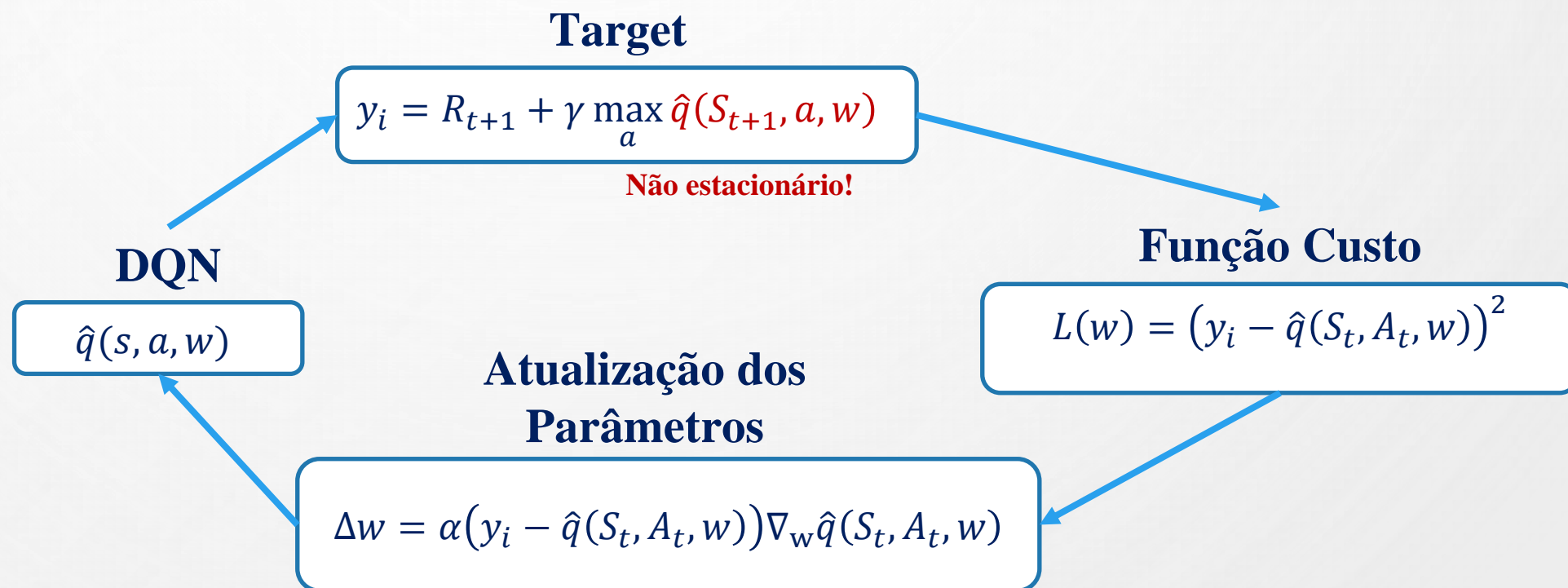
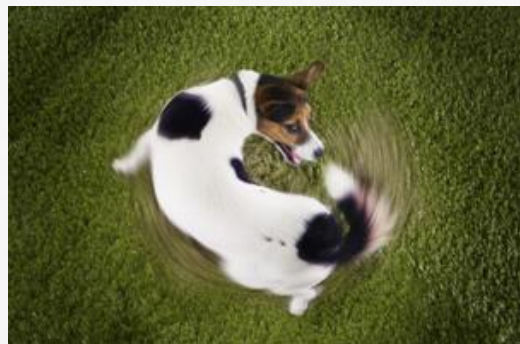
Teste para demanda constante

Problemas do Naive DQN: Correlação Temporal

- Há um grande problema no Naive DQN que é a Correlação Temporal
- Durante um episódio, realiza-se o treino da rede neural diversas vezes alterando todos os seus pesos w . O problema é que um episódio tem seus estados muito correlacionados, ou seja, há o treino seguido desses estados.
- Suponha, por exemplo, um ambiente em que as ações levam a estados muito similares ao anterior (há pouca mudança no sistema), o agente irá treinar diversas vezes seguidas para essa configuração e possivelmente esquecerá de outras.

Problemas do Naive DQN: treinamento circular

- O treino do Naive DQN é circular, pois nosso target depende da própria rede neural que estamos treinando.
- Diferentemente do treinamento tabular, quando fazemos update da rede, alteramos todos os pesos w . Enquanto nas tabelas alteramos apenas o valor $Q(S_t, A_t)$ que não interfere em $Q(S_{t+1}, A_{t+1})$



Deep Q-Network

Deep Q-Network

DQN

Experience Replay

- Uma das formas de resolver o problema da correlação temporal é o Experience Replay
- Ao invés de passar por um estado e já treinar a rede neural, o agente guarda cada uma das experiências $(S_t, A_t, R_{t+1}, S_{t+1})$ em um replay memory \mathcal{D}
- A cada *timestep*, uma amostra aleatória chamada *minibatch* é retirada do buffer \mathcal{D}
- Realiza-se o gradient descent para o minibatch

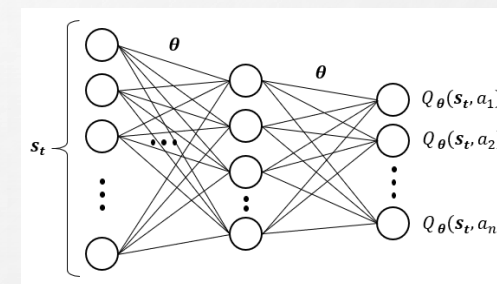
$(S_t, A_t, R_{t+1}, S_{t+1})$

Transições
correlacionadas

Replay
Memory

Amostra

Treino



Stochastic Gradient Descent with Experience Replay

- Dado o replay *memory* \mathcal{D} :

$$\mathcal{D} = \{(S_1, A_1, R_2, S_2), (S_2, A_2, R_3, S_3) \dots (S_T, A_T, R_{T+1}, S_{T+1})\}$$

- Se amostra de forma aleatória a nb transições
- Para cada transição i dentro do batch, encontrar o target y_i

$$y_i = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w)$$

- Aplicar o gradiente descente para cada uma das transições no *minibatch*

$$\Delta w = \alpha (y_i - \hat{q}(S_t, A_t, w)) \nabla_w \hat{q}(S_t, A_t, w)$$

Prioritized Experience Replay

- O Experience Replay retira uma amostra aleatória da replay *memory* \mathcal{D} . Porém há algum jeito melhor de escolher?
- Uma solução encontrada é utilizar o Prioritized Experience Replay que dá prioridade de selecionar as transições que tenham o maior DQN Error. Abaixo a função *priority*:

$$p(i) = | \underbrace{r + \gamma \max_a \hat{q}(S_{t+1}, a, w)}_{\text{Target}} - \hat{q}(S_t, a, w) |$$

- O Prioritized Experience Replay dá chances maiores de selecionar as transições que tenham maior erro em relação ao seu target.

Prioritized Experience Replay

$$p(i) = |r + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, a, w)|$$

- Para a probabilidade de escolher uma ação, tiramos proporção de todos os $p(i)$ (*stochastic prioritization*):

$$\mathbb{P}(i) = \frac{p(i)^\beta}{\sum_k p(k)^\beta}$$

- β pode ser um qualquer valor real positivo. Caso seja zero, temos uma distribuição uniforme (equivalente ao Experience Replay) e, quanto maior, maior a chance de selecionar os elementos com maior erro.

Target Network

- Com o objetivo de resolver o problema de treinamento circular e deixar o processo mais estável, a equipe Deepmind desenvolveu a *Target Network*
- Cria-se uma cópia da rede neural $\hat{q}(s, a, w)$ com pesos fixos w^-
- A *loss function* é alterada para calcular o alvo y_i usando $\hat{q}(s, a, w^-)$:

$$L(w) = \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w^-) - \hat{q}(S_t, A_t, w) \right)^2$$

$$\Delta w = \alpha \left(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w^-) - \hat{q}(S_t, A_t, w) \right) \nabla_w \hat{q}(S_t, A_t, w)$$

- A cada certo número de updates de w , se atualiza w^- com os valores de w

- Melhoria do Naive DQN ao utilizar tanto Experience Replay quanto *Target Network*
- Um dos algoritmos mais utilizados em aprendizado por reforço
- Funciona somente para ambientes com ações discretas
- Desenvolvido pela equipe do DeepMind foi um dos primeiros algoritmos profundos estáveis em aprendizado por reforço

Algoritmo DQN proposto nos dois artigos a seguir:

Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning.

Mnih, V., Kavukcuoglu, K., Silver, D. et al. *Human-level control through deep reinforcement learning*. Nature 518, 529–533 (2015). <https://doi.org/10.1038/nature14236>

Algoritmo: DQN (com target network e experience replay)

Parâmetro do Algoritmo: Taxa de aprendizado $\alpha \in (0,1]$, minibatch size $nb \in \mathbb{N}$ e target network update frequency $C \in \mathbb{N}$

Inicializar pesos w arbitrariamente e $w^- \leftarrow w$

Repetir para cada episódio:

 Inicializar estado inicial $S_0 \sim \mathbb{P}(S_0 = s), s \in \mathcal{S}$

 Repetir para cada timestep $t = 0, 1, 2, \dots$:

 Escolher ação A_t de S_t usando a política derivada da rede neural $\hat{q}(S_t, a, w)$ (como: ϵ -greedy)

 Executar ação A_t e observar R_{t+1}, S_{t+1}

 Gravar transição $(S_t, A_t, R_{t+1}, S_{t+1})$ no replay buffer \mathcal{D}

 Amostrar um minibatch aleatório de \mathcal{D} de tamanho nb

 Para cada transição i dentro do minibatch \mathcal{D}

 Caso seja Estado Terminal:

$$y_i = R_{t+1}$$

 Else:

$$y_i = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w^-)$$

 Fazer o gradient descent usando a loss $(y_i - \hat{q}(S_t, A_t, w))^2$ fazendo update dos

$$\text{valores } w: \Delta w = \alpha \left(r + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w) \right) \nabla_w \hat{q}(S_t, A_t, w)$$

$$S_t \leftarrow S_{t+1}$$

 Caso $\text{mod}(n_{\text{atualizações}}, C) == 0$:

$$w^- \leftarrow w$$

 Até que S_t seja um estado terminal

Retorna: Aproximações da Funções Valor e Política ótimas \hat{q}^* e π^*

DQN em comparação com naive DQN

Extended Data Table 3 | The effects of replay and separating the target Q-network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

DQN agents were trained for 10 million frames using standard hyperparameters for all possible combinations of turning replay on or off, using or not using a separate target Q-network, and three different learning rates. Each agent was evaluated every 250,000 training frames for 135,000 validation frames and the highest average episode score is reported. Note that these evaluation episodes were not truncated at 5 min leading to higher scores on Enduro than the ones reported in Extended Data Table 2. Note also that the number of training frames was shorter (10 million frames) as compared to the main results presented in Extended Data Table 2 (50 million frames).

Fonte: Mnih, V., Kavukcuoglu, K., Silver, D. et al. *Human-level control through deep reinforcement learning*. Nature 518, 529–533 (2015). <https://doi.org/10.1038/nature14236>

DQN Beer game > Colab: Beer_Game_DQN.ipynb

Beer_Game_DQN.ipynb ☆

Arquivo Editar Ver Inserir Ambiente de execução Ferramentas Ajuda [Todas as alterações foram salvas](#)

Comentário Compartilhar

Reconectar Edit

Índice

- Exemplo de aula: Beer game environment com DQN
- Explicação do Environment
- Imports
- Environment Class
- Funções Auxiliares
- DQN
 - Network class
 - DQN Agent
- Loop de treino para demanda constante**
- Loop de treino para demanda variável
- Loop de treino para demanda variável e ambiente muito mais complexo (mais estados e ações)
- Seção

+ Código + Texto

Exemplo de aula: Beer game environment com DQN

Explicação do Environment

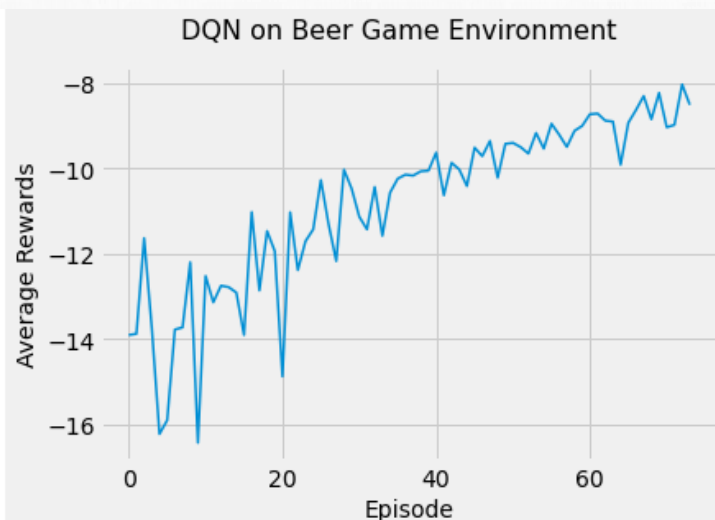
MDP EXAMPLE: SUPPLY CHAIN 58

- Gestão de Cadeia Logística (Supply Chain Management):

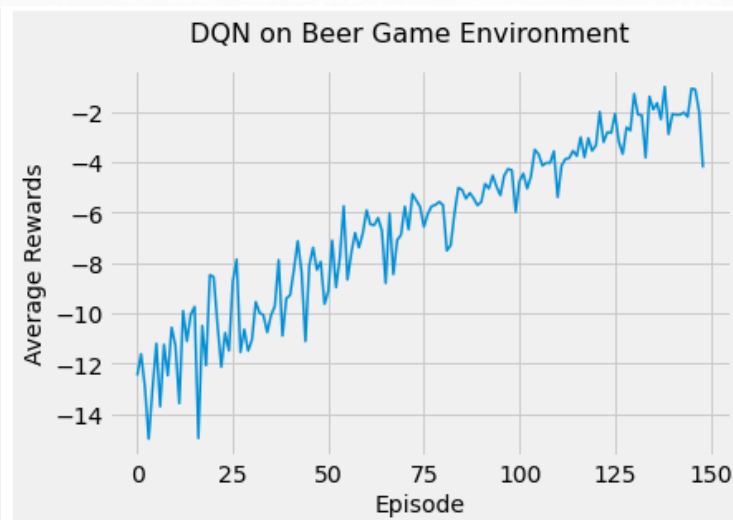
Considere o problema de manutenção de inventário de determinado produto, em que cada nível da cadeia mantém uma quantidade do produto e fornece para o nível inferior (Beer Distribution Game)
- $S_i(t)$: Estoque do nível i no instante de tempo t .
- $O_{ij}(t)$: Tamanho de pedido do nível i para nível j no instante de tempo t .

Beer env comparação de resultados para demanda Constante

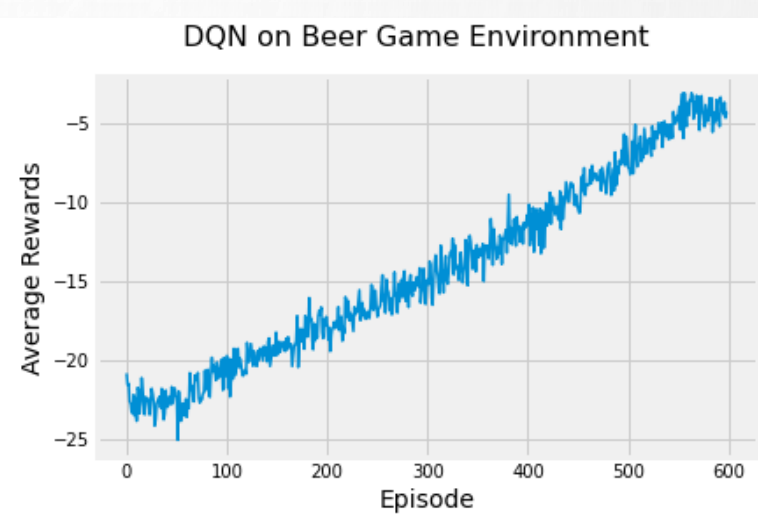
DQN para demanda constante



DQN para demanda variável



DQN para demanda variável e ambiente muito mais complexo



Espaço de Estados
 $|S| = 1296$
 Espaço de Ações:
 $|A| = 256$

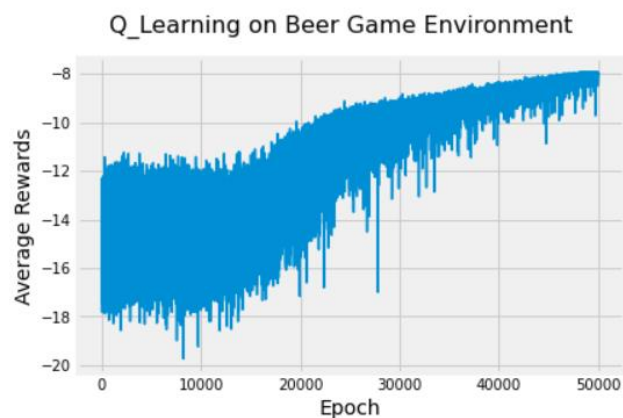
Espaço de Estados
 $|S| = 625$
 Espaço de Ações:
 $|A| = 81$

Espaço de Estados
 $|S| = 65536$
 Espaço de Ações:
 $|A| = 2401$

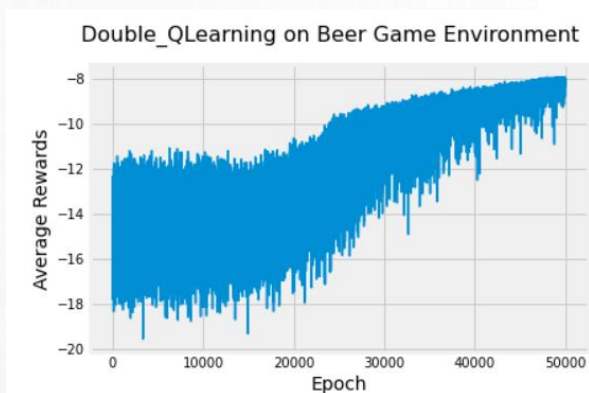
Repare que o DQN é escalável

DQN Beer env

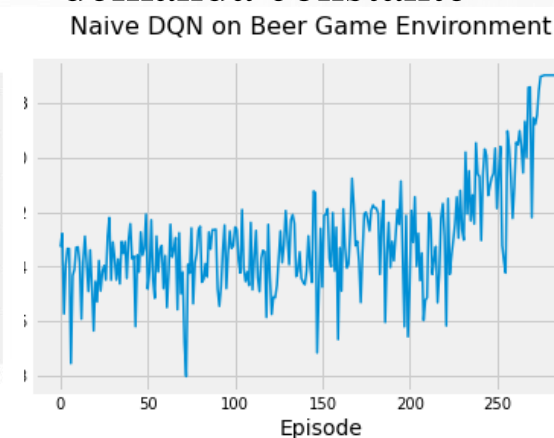
Q-Learning para demanda constante



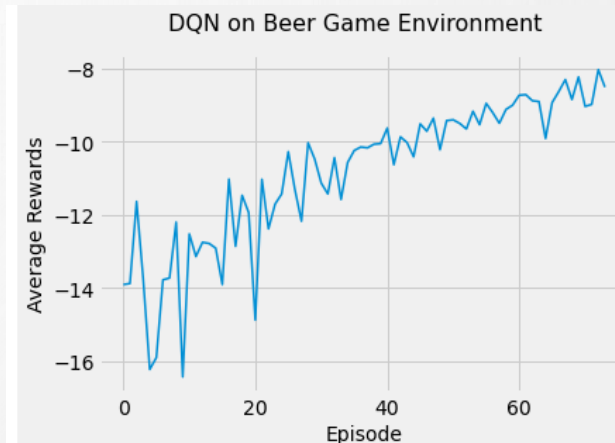
Double Q-Learning para demanda constante



Naive DQN para demanda constante



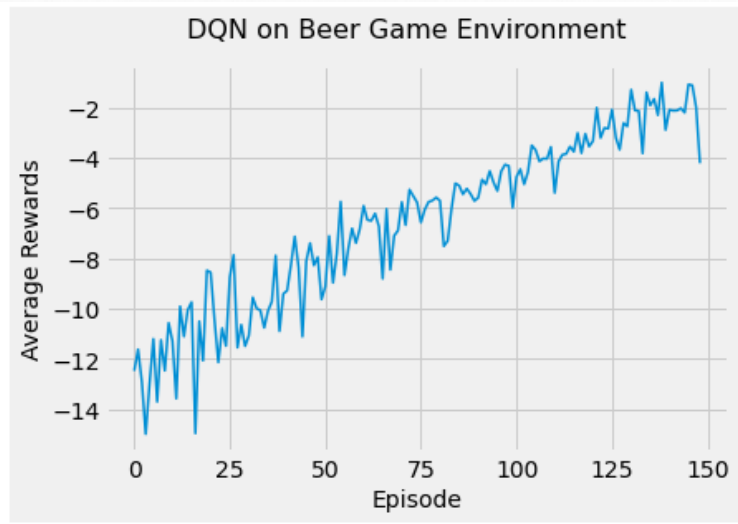
DQN para demanda constante



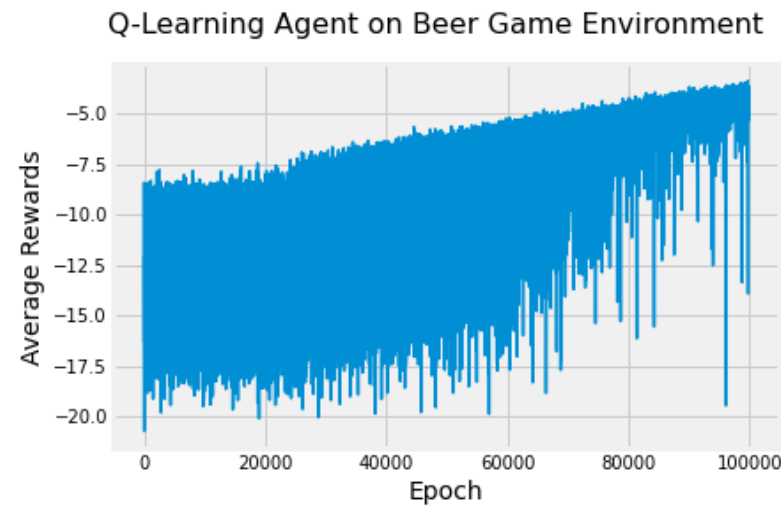
Espaço de Estados
 $|S| = 1296$
 Espaço de Ações:
 $|A| = 256$

Beer env comparação de resultados para demanda variável

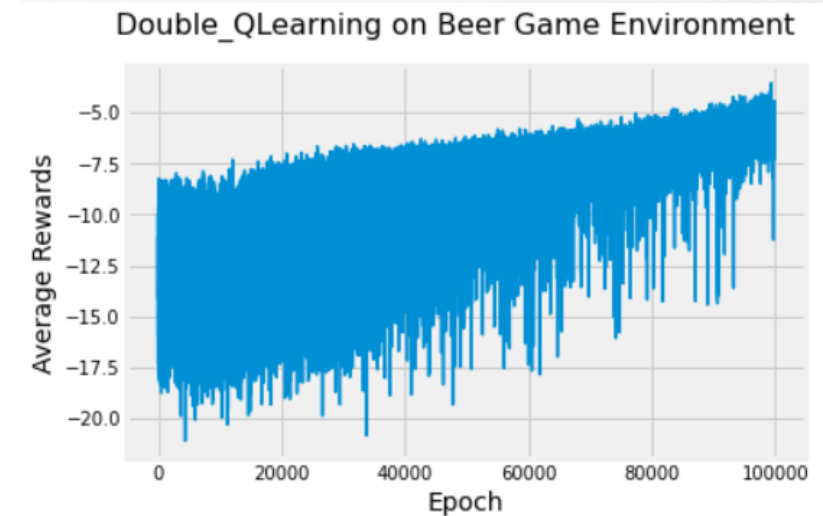
DQN para demanda variável



Q-Learning para demanda variável



Double Q-Learning para demanda variável




Espaço de Estados

$|S| = 625$

Espaço de Ações:

$|A| = 81$

DQN KerasRL-> Colab: keras_rl_cartpole_DQN.ipynb

 keras_rl_cartpole_DQN.ipynb ☆

Arquivo Editar Ver Inserir Ambiente de execução Ferramentas Ajuda [Todas as alterações foram salvas](#)

Comentário

+ Código + Texto

Índice

🔍 DQN para resolver o problema Cartpole usando Keras RL

<> Fazendo instalação do keras-RL2

Imports

📁 Iniciando o ambiente

Criação da Rede neural

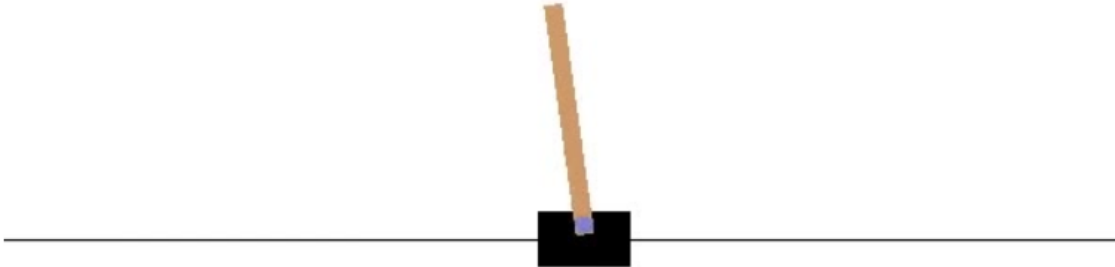
Utilizando o Keras-RL

Testando o Agente e salvando os pesos da rede

Plotando as recompensas do sistema ao longo do treino

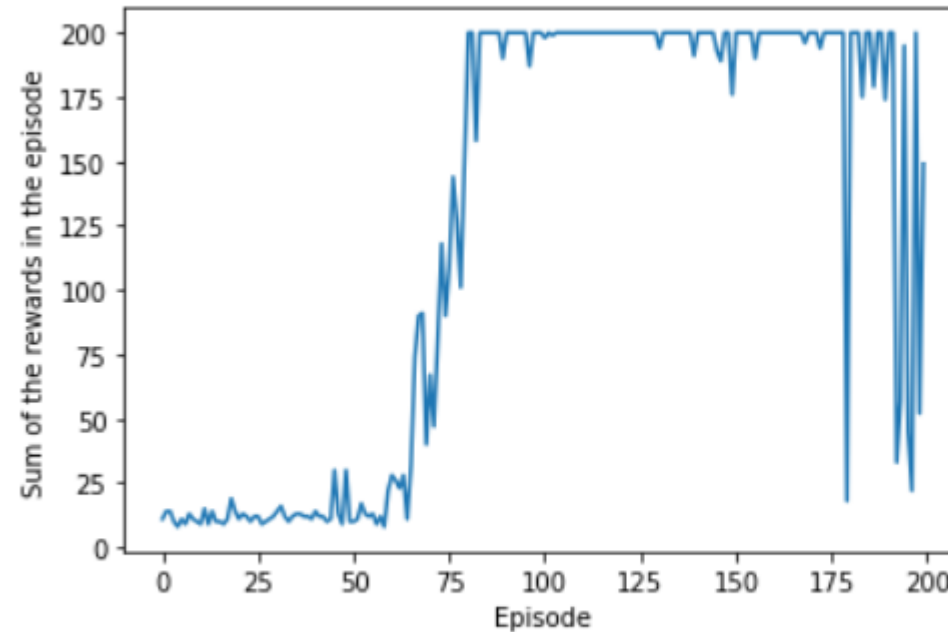
+ Seção

▼ DQN para resolver o problema Cartpole usando Keras RL



DQN KerasRL-> Colab: keras_rl_cartpole_DQN.ipynb

Recompensa média do sistema para cada época usando DQN: 124.88



Double Deep Q-Network

Double Deep Q-Network

Double DQN

Relembrando Tendência de Superestimação do Q-Learning

Tendência de Superestimação do Q-Learning

66

- O Q-Learning costuma superestimar os valores Q.
- De forma intuitiva, imagine: uma aplicação que toda a tabela Q é aproximadamente 0. Dado ao caráter randômico da exploração, alguns valores serão negativos e outros positivos. Como o Q-Learning utiliza o maior valor possível de $Q(s_{t+1}, A)$, ele vai acabar utilizando valores positivos para calcular o $Q(s, a)$ assim superestimando seu valor.
- Chamado em inglês de maximization bias.

Double Deep Q-Network

- Da mesma forma que o Q-Learning tem o problema de *maximization bias*, o DQN também tem pois ambos são análogos
- Uma possibilidade é utilizar o Double DQN (em alguns lugares da literatura é encontrado como DDQN)
- Há duas redes neurais neurais \widehat{q}_1 e \widehat{q}_2 com pesos w_1 e w_2 respectivamente:
 - A rede que será treinada escolhe a ação do target
 - A outra rede avalia o par estado ação do target

Relembrando Double Q-Learning

Double Q-Learning

67

- Um método criado para resolver o problema de *maximization bias* é o Double Q-Learning
- Similar ao Q-Learning, o Double Q-Learning armazena na memória duas tabelas Q_1 e Q_2 e toda vez que vai se chamar o treino roda-se uma moeda. Para cada um dos casos possíveis, se atualiza apenas uma das tabelas Q:

$$\begin{aligned} Q_1(S_t, A_t) &\leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_{a \in \mathcal{A}} Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \\ Q_2(S_t, A_t) &\leftarrow Q_2(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_1(S_{t+1}, \arg \max_{a \in \mathcal{A}} Q_2(S_{t+1}, a)) - Q_2(S_t, A_t)] \end{aligned}$$

- Como é utilizado uma tabela diferente para pegar os valores de $\arg \max_{a \in \mathcal{A}} Q_2(S_{t+1}, a)$ não há superestimação.

Double Deep Q-Network (Double DQN)

- Similar ao DQN porém com duas redes neurais \widehat{q}_1 e \widehat{q}_2 com pesos w_1 e w_2 respectivamente. A cada treino há 50% de chance de treinar cada uma das delas:

Treino da rede \widehat{q}_1

$$L(w_1) = \mathbb{E}_{\pi} \left(R_{t+1} + \gamma \widehat{q}_2(S_{t+1}, \arg \max_{a \in \mathcal{A}} \widehat{q}_1(S_{t+1}, a, w_1), w_2^-) - \widehat{q}_1(S_t, A_t, w_1) \right)^2$$

$$\Delta w_1 = \alpha \left(R_{t+1} + \gamma \widehat{q}_2(S_{t+1}, \arg \max_{a \in \mathcal{A}} \widehat{q}_1(S_{t+1}, a, w_1), w_2^-) - \widehat{q}_1(S_t, A_t, w_1) \right) \nabla_w \widehat{q}_1(S_t, A_t, w_1)$$

De forma análoga, treino da rede \widehat{q}_2

$$L(w_2) = \mathbb{E}_{\pi} \left(R_{t+1} + \gamma \widehat{q}_1(S_{t+1}, \arg \max_{a \in \mathcal{A}} \widehat{q}_2(S_{t+1}, a, w_2), w_1^-) - \widehat{q}_2(S_t, A_t, w_2) \right)^2$$

$$\Delta w_2 = \alpha \left(R_{t+1} + \gamma \widehat{q}_1(S_{t+1}, \arg \max_{a \in \mathcal{A}} \widehat{q}_2(S_{t+1}, a, w_2), w_1^-) - \widehat{q}_2(S_t, A_t, w_2) \right) \nabla_w \widehat{q}_2(S_t, A_t, w_2)$$

Double DQN KerasRL-> Colab: keras_rl_cartpole_Double_DQN.ipynb

Colab interface showing the notebook **keras_rl_cartpole_Double_DQN.ipynb**. The interface includes a top bar with the Colab logo, file name, and navigation links (Arquivo, Editar, Ver, Inserir, Ambiente de execução, Ferramentas, Ajuda). A right sidebar shows "Comentário" and "Recon".

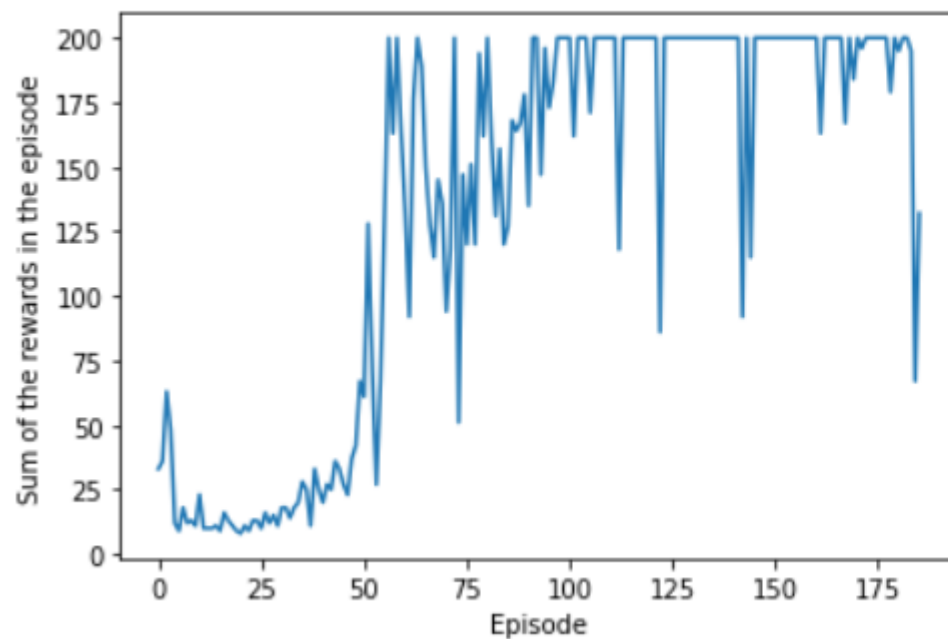
The left sidebar displays the **Índice** (Index) with the following sections:

- DQN para resolver o problema Cartpole usando Keras RL** (selected)
- Fazendo instalação do keras-RL2
- Imports
- Iniciando o ambiente
- Criação da Rede neural
- Utilizando o Keras-RL
- Testando o Agente e salvando os pesos da rede
- Plotando as recompensas do sistema ao longo do treino
- + Seção

The main content area shows the title **DQN para resolver o problema Cartpole usando Keras RL** and a visual representation of the Cartpole environment. The cartpole is shown as a black square on a horizontal track, with a brown pole attached to it, tilted at an angle.

Double DQN -> Colab: keras_rl_cartpole_Double_DQN.ipynb

Recompensa média do sistema para cada época usando Double DQN: 133.88709677419354

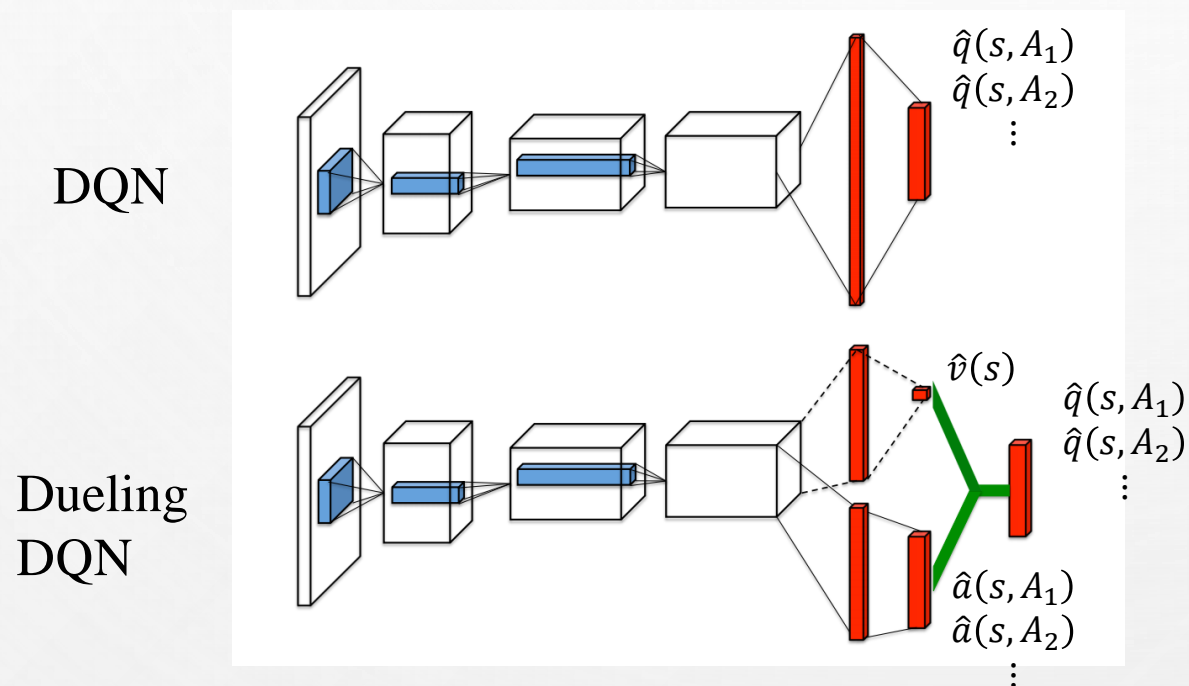


Dueling Deep Q-Network (Dueling DQN)

Dueling Deep Q-Network
Dueling DQN

Dueling Deep Q-Network (Dueling DQN)

- Uma expansão do DQN, o Dueling DQN mexe um pouco na saída da rede neural
- Ao invés da rede neural tentar prever apenas o valor $\hat{q}(s, a, w)$, a rede neural tem duas saídas: função valor do estado $\hat{v}(s)$ e a função *advantage*/vantagem $\hat{a}(s, a)$.



Fonte: Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., Freitas, N. *Dueling Network Architectures for Deep Reinforcement Learning*, ICML 2016

Dueling Deep Q-Network (Dueling DQN)

- A função *advantage* é definida pela seguinte fórmula:

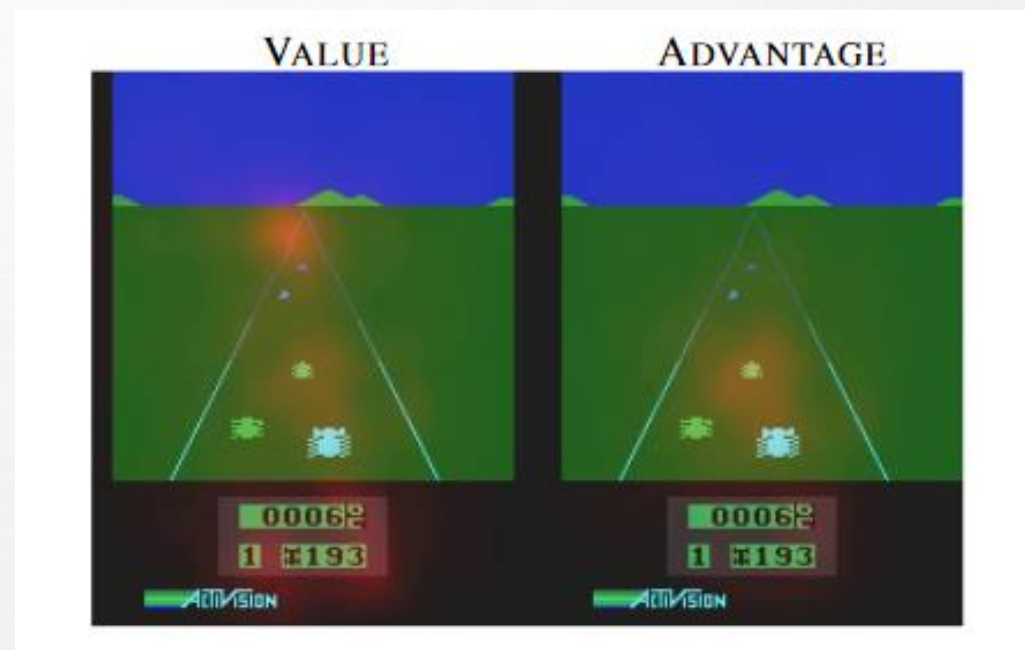
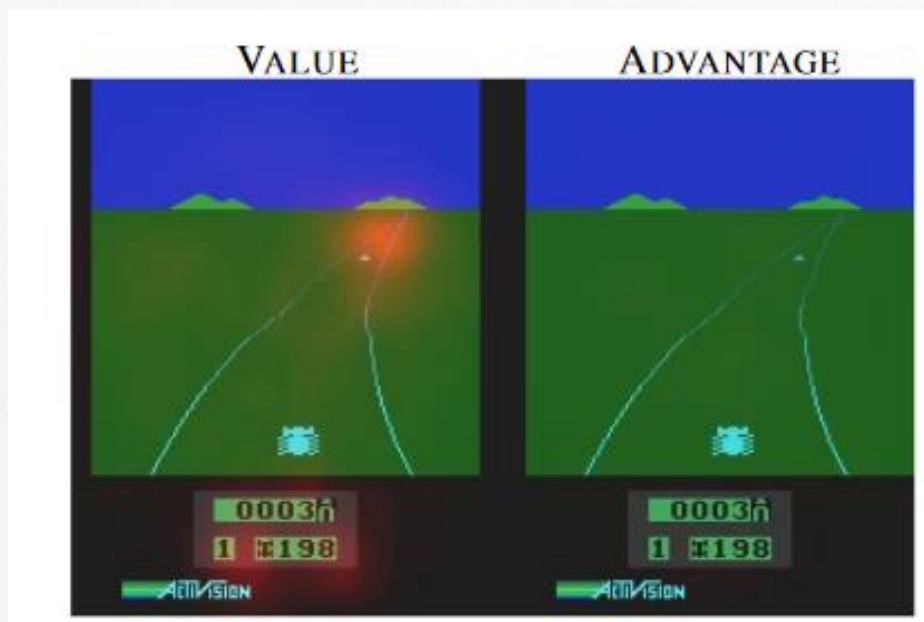
$$A(s, a) = Q(s, a) - V(s)$$

- A partir dos valores $\hat{a}(s, a, w)$ e $\hat{v}(s, w)$ calculados pela rede neural, é possível reconstruir o valor $\hat{q}(s, a, w)$ com mais precisão:

$$\hat{q}(s, a, w) = \hat{v}(s, w) + (\hat{a}(s, a, w) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} \hat{a}(s, a', w))$$

Vantagens Deep Q-Network (Dueling DQN)

- Ao forçar a rede a aprender a calcular $\hat{v}(s, w)$ facilita o treino em casos em que as ações não interferem muito, pois há a diferenciação entre o que é uma ação boa e o que é um estado bom.



Dueling DQN KerasRL-> Colab: keras_rl_cartpole_Dueling_DQN.ipynb

 keras_rl_cartpole_Dueling_DQN.ipynb ☆

Arquivo Editar Ver Inserir Ambiente de execução Ferramentas Ajuda

Comentário

Reco

Índice

Q

DQN para resolver o problema Cartpole usando Keras RL

<>

Fazendo instalação do keras-RL2

Imports

□

Iniciando o ambiente

Criação da Rede neural

Utilizando o Keras-RL

Testando o Agente e salvando os pesos da rede

Plotando as recompensas do sistema ao longo do treino

+ Seção

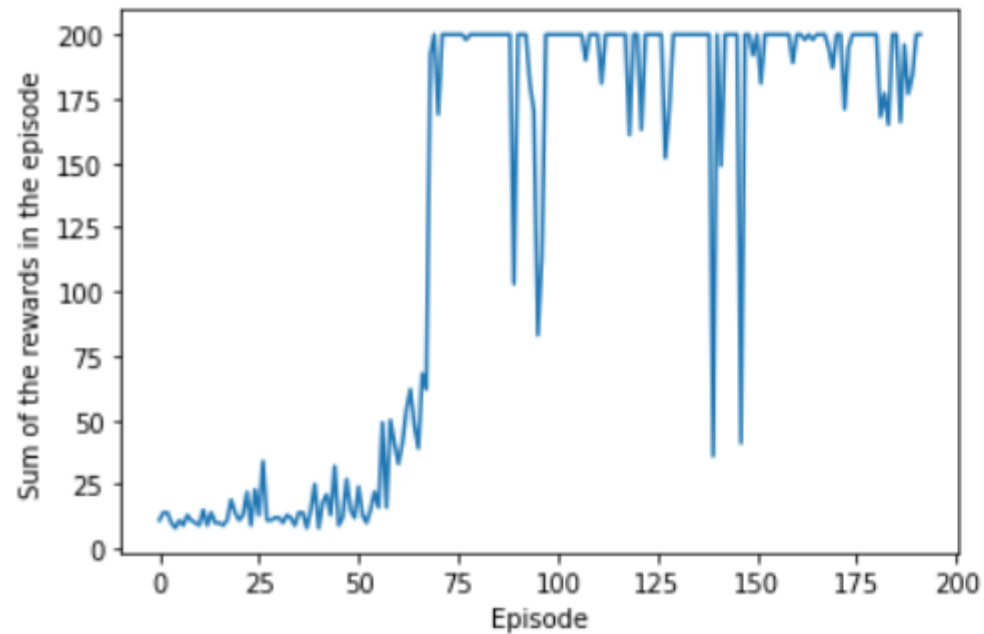
+ Código + Texto

▼ DQN para resolver o problema Cartpole usando Keras RL



Dueling DQN KerasRL-> Colab: keras_rl_cartpole_Dueling_DQN.ipynb

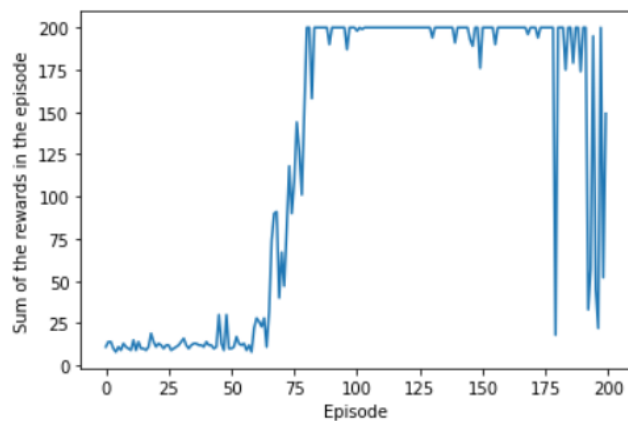
Recompensa média do sistema para cada época usando Dueling DQN: 129.91666666666666



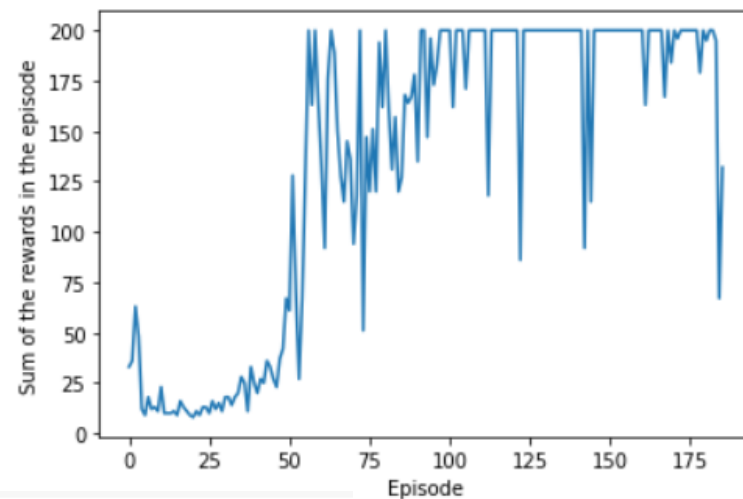
Comparação DQN vs Dueling DQN vs Double DQN

DQN

Recompensa média do sistema para cada época usando DQN: 124.88



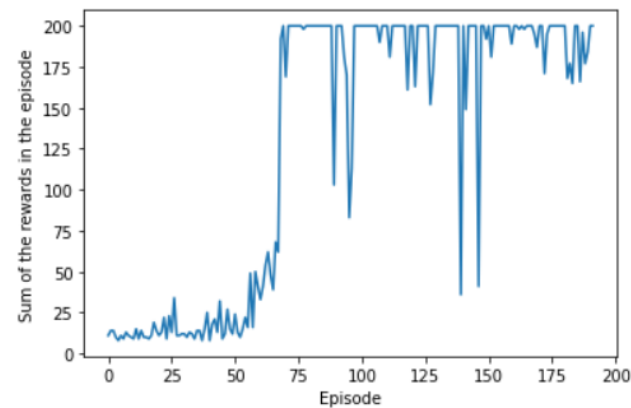
Recompensa média do sistema para cada época usando Double DQN: 133.88709677419354



Double DQN

Dueling DQN

Recompensa média do sistema para cada época usando Dueling DQN: 129.91666666666666

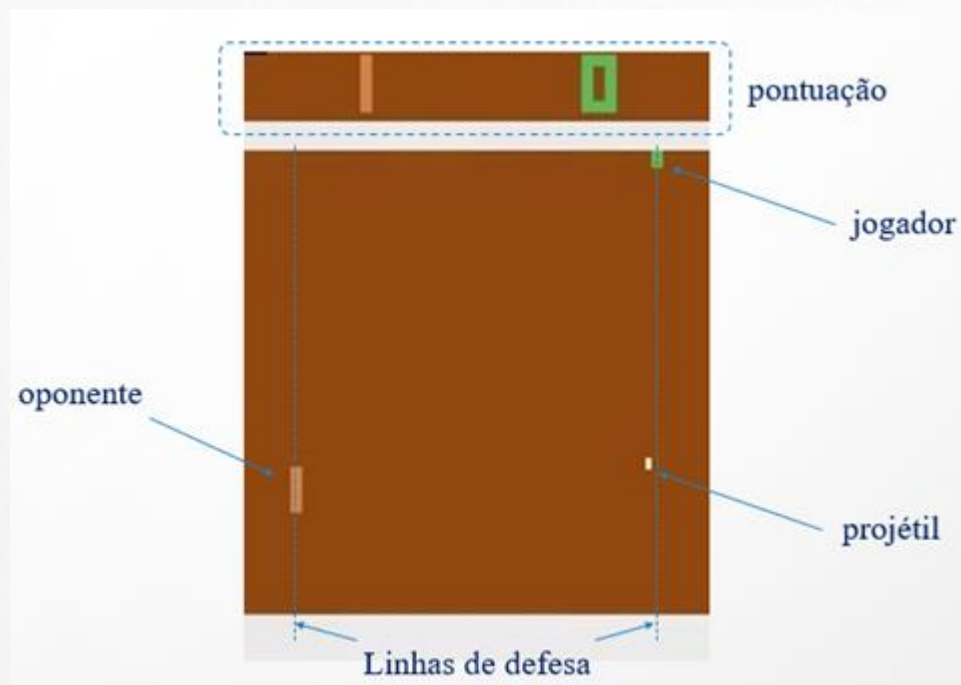


Deep Q-Network DQN

Trabalho T_1 - Jogo Pong

- Tarefa A)

Determinar dimensões dos espaços de observações e ações.



Trabalho T_1 - Jogo Pong

- Tarefa A)

Determinar dimensões dos espaços de observações e ações.

- Tarefa B)

Implementar um agente aleatório e verificar quantos gols o agente consegue fazer em certo número de episódios

```
Número total de gols feitos por agente aleatório = 3
```

Trabalho T_1 - Jogo Pong

- Tarefa A)

Determinar dimensões dos espaços de observações e ações.

- Tarefa B)

Implementar um agente aleatório e verificar sua performance

- Tarefa C)

Implementar o agente DQN para o PONG

DQN Pseudocode

```

1: Input  $C, \alpha, D = \{\}$ , Initialize  $\mathbf{w}, \mathbf{w}^- = \mathbf{w}, t = 0$ 
2: Get initial state  $s_0$ 
3: loop
4:   Sample action  $a_t$  given  $\epsilon$ -greedy policy for current  $\hat{Q}(s_t, a; \mathbf{w})$ 
5:   Observe reward  $r_t$  and next state  $s_{t+1}$ 
6:   Store transition  $(s_t, a_t, r_t, s_{t+1})$  in replay buffer  $D$ 
7:   Sample random minibatch of tuples  $(s_i, a_i, r_i, s_{i+1})$  from  $D$ 
8:   for  $j$  in minibatch do
9:     if episode terminated at step  $i + 1$  then
10:       $y_i = r_i$ 
11:     else
12:       $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$ 
13:     end if
14:     Do gradient descent step on  $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$  for parameters  $\mathbf{w}$ :  $\Delta \mathbf{w} = \alpha (y_i - \hat{Q}(s_i, a_i; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$ 
15:   end for
16:    $t = t + 1$ 
17:   if  $\text{mod}(t, C) == 0$  then
18:      $\mathbf{w}^- \leftarrow \mathbf{w}$ 
19:   end if
20: end loop

```

Trabalho T_1 - Jogo Pong

- Tarefa A)

Determinar dimensões dos espaços de observações e ações.

- Tarefa B)

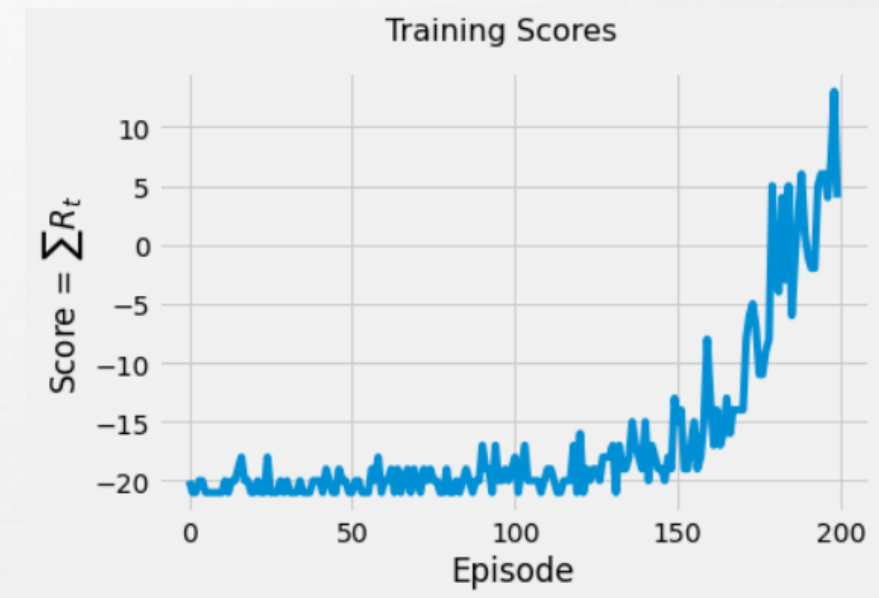
Implementar um agente aleatório e verificar sua performance

- Tarefa C)

Implementar o agente DQN para o PONG

- Tarefa D)

Implementar o Loop de Treino para o DQN



Referências Bibliográficas

- [1] Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction*, The MIT Press (2020).
- [2] Geever, K. *Deep Reinforcement Learning in Inventory Management*, 2020
- [3] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning.
- [4] Mnih, V., Kavukcuoglu, K., Silver, D. et al. *Human-level control through deep reinforcement learning*. Nature 518, 529–533 (2015). <https://doi.org/10.1038/nature14236>
- [5] UCL Course on RL by David Silver (<https://www.davidsilver.uk/teaching/>)
- [6] Schaul, T., Quan, T., Antonoglou, T., Silver D. *Prioritized Experience Replay*, ICLR2016
- [7] Stanford Class by Emma Brunskill (<https://web.stanford.edu/class/cs234/modules.html>)
- [8] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., Freitas, N. *Dueling Network Architectures for Deep Reinforcement Learning*, ICML 2016

Muito obrigado a todos!

Dúvidas