Aula20

DATA SCIENCE IPT

LINKED EDUCATION

V2019

Keras **is a high-level** neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling **fast experimentation**. *Being able to go from idea to result with the least possible delay is key to doing good research.*
Use Keras if you need a deep learning library that:
Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
Supports both convolutional networks and recurrent networks, as well as combinations of the two.
Runs seamlessly on CPU and GPU.

Fonte: https://keras.io/

Basicamente, há duas formas de definir um modelo com Keras:

- **Sequential model**
- **Model Class com Functional API**

Fonte: https://keras.io/

O Sequential é uma pilha linear de camadas

Exemplos:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Fonte: https://keras.io/

Revisão Keras

Functional API : é a maneira mais usual para a definição de modelos complexos, como modelos de múltiplas saídas, por exemplo.

Exemplo:

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
```

Fonte: https://keras.io/

“Compilação”: depois da definição do modelo, na compilação é definida a função de custo (loss), o otimizador e eventuais métricas.

Exemplos:

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')

# For custom metrics
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

Fonte: https://keras.io/

“Treinamento”: de forma análoga ao sklearn, no treinamento fazemos o “fit” entre dados e rótulos para ajustar o modelo.

Exemplo:

```
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
```

Fonte: https://keras.io/

A avaliação do modelo (métrica pré-definida) é feita com **evaluate**

Exemplo:

```
score = model.evaluate(x_test, y_test)
```

Predições são feitas com **predict**

Exemplo:

```
ynew = model.predict_classes(Xnew)
```

Fonte: https://keras.io/

Comparar códigos

CNN-Keras-Digits.ipynb
e
MLP-Keras-Digits.ipynb

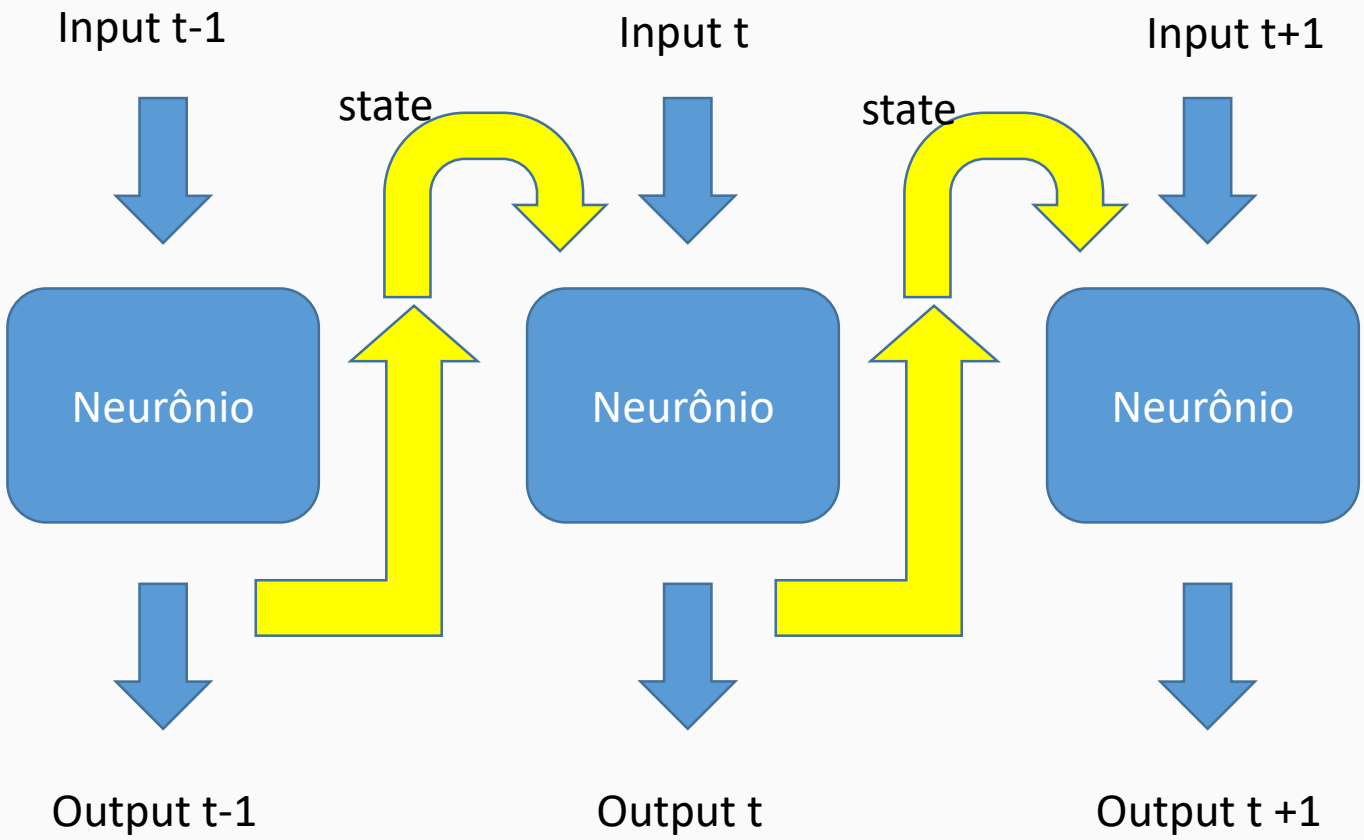
Analisar a formação das camadas com model.summary()
Observar validation_split já em model.fit()

As redes neurais recorrentes têm conexões com loops, adicionando feedback e memória às redes ao longo do tempo. Essa memória permite que esse tipo de rede aprenda e generalize através de seqüências de entradas, em vez de padrões individuais, como acontrece com as MLPs.

Redes neurais recorrentes são tipicamente usadas no processamento de sequências (textos, séries temporais etc.)

Recurrent Neural Networks (RNN)

Exemplo de RNN (visão “unfold”) ao longo do tempo

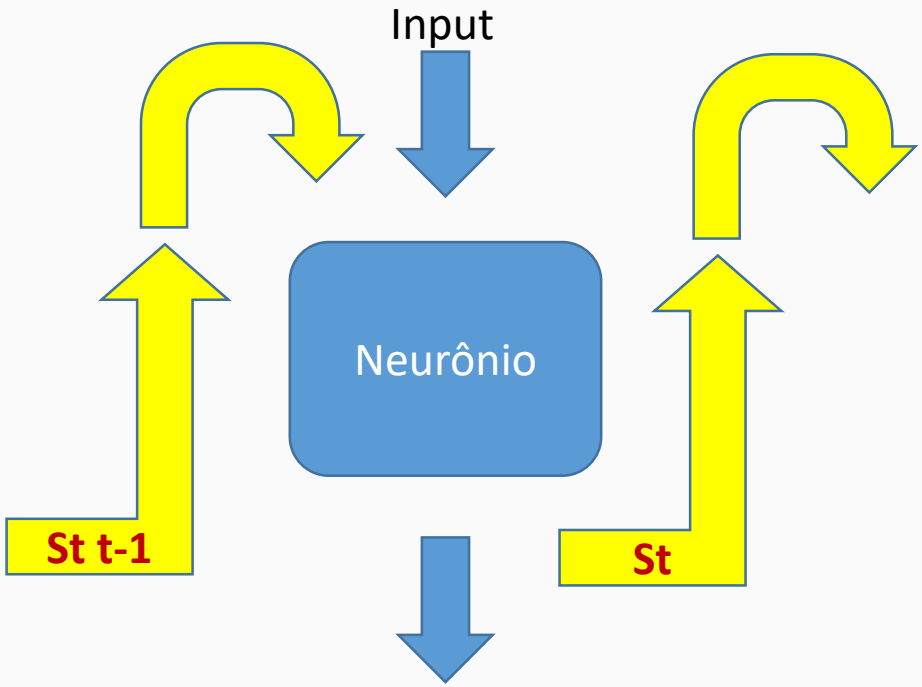


Esse arranjo pode ser implementado com SimpleRNN com Keras

Recurrent Neural Networks (RNN)

Vamos implementar uma RNN que “adivinhará” o próximo número em sequência de inteiros. Por exemplo, para a entrada 5,6,7,8, o valor a ser previsto é 9.

Utilizaremos como ativação da célula, a função linear (identidade), g(x)=x.

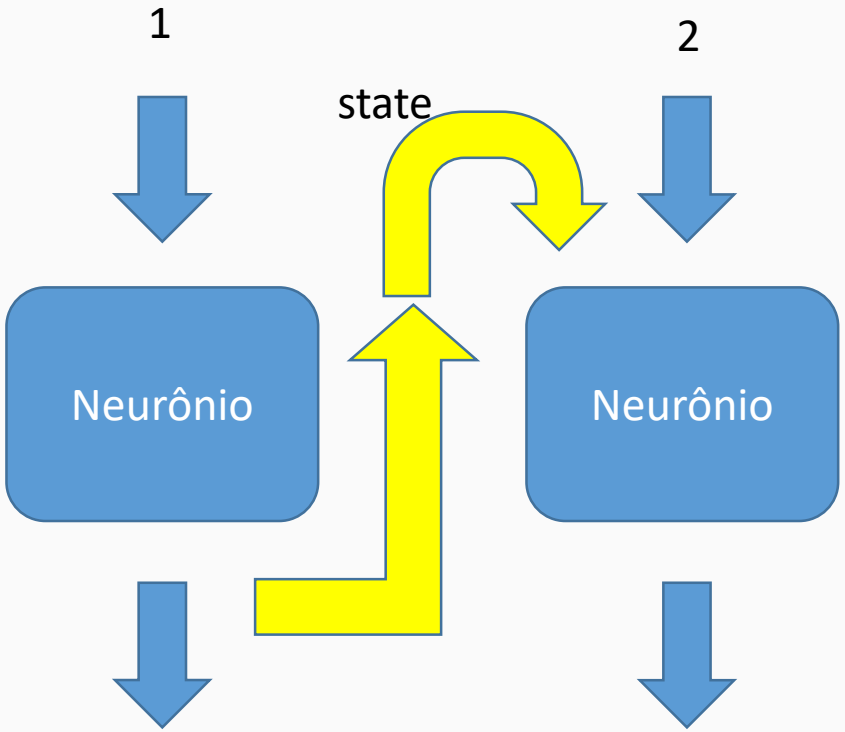


Output= St=g(input*Wi+St_{t-1}*Ws+bias)

Partindo de **SimpleRNN.ipynb :**

1- Analisar código
2-Testar a predição com sequências com tamanho 1 a 10
3- Obtenha os pesos (model.get_weights()
4-Obtenha o output para a entrada 1,2 e o obtenha com os pesos

Wi (input)=0.9054703
Ws (state)=0.10439815
Bias=0.99077886



1*0.9054703+0.99077886
= 1.89624916

1.89624916*0.10439815+2*0.905
4703+0.99077886
= 2.999684364243053 7

Partindo de **SimpleRNN2.ipynb :**

1- Analisar código (amostras com tamanho 10)
2-Testar a predição com sequências com tamanho 2....
Houve melhora em relação às predições de **SimpleRNN.ipynb?**

Partindo de **SimpleRNN3.ipynb :**

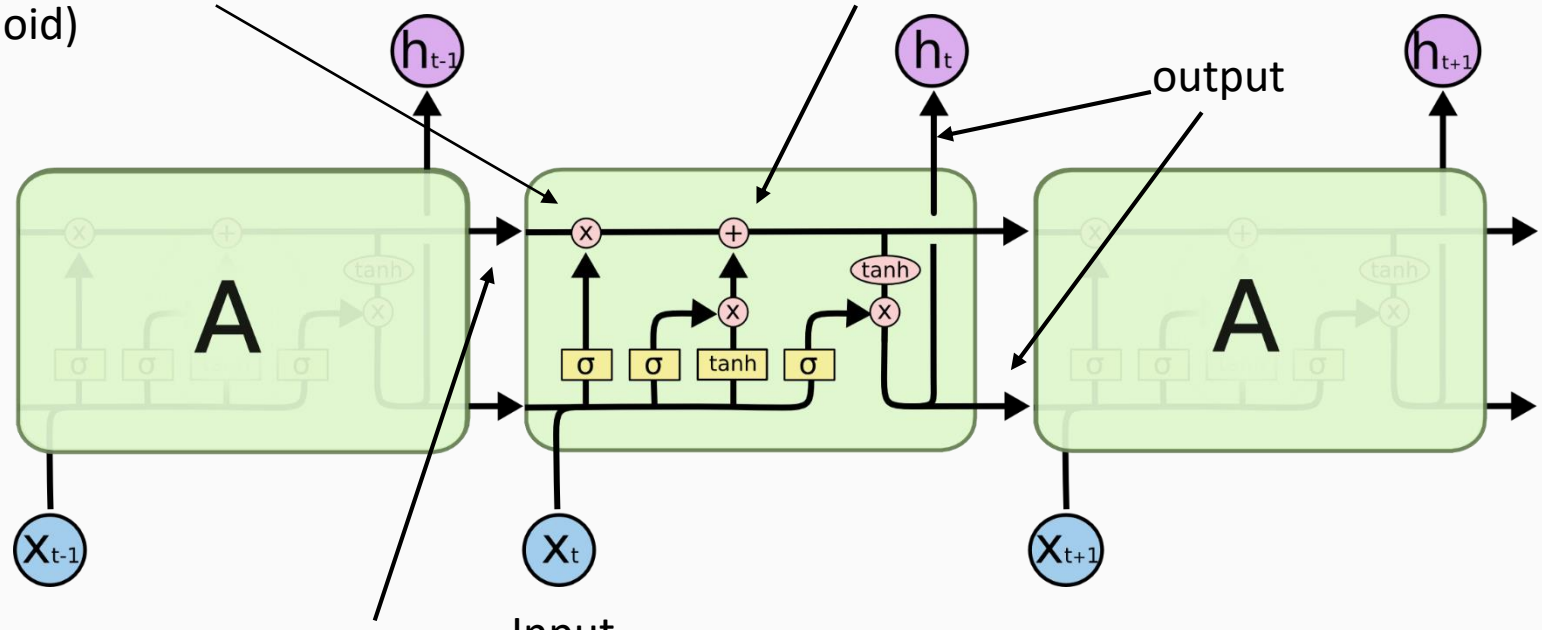
1- Analisar código (amostras com tamanho 10 e 4 células...houve necessidade de uma camada Dense(1). Por que?
2)Explique model.summary() e model.get_weight

Obviamente, para analisar sequências de tamanhos variáveis, devemos treinar a RNN com amostras de tamanho variável.

Uma rede recorrente básica, como a SimpleRNN (utilizada nos exemplos anteriores) é difícil tratar a dependências de longo prazo. Isso aparece no treinamento (BPTT Backpropagation Through Time), onde séries muito longas podem fazer os gradientes sumirem ou explodirem por problemas de cálculo numérico.
As redes LSTM tratam esse problema “filtrando” a memória que deve ser propagada. Esses filtros são criados no processo de treinamento.

Multiplica componente a componente do vetor original pelo “filtro” (vetor com valores entre 0 e 1 vindos do sigmoid)

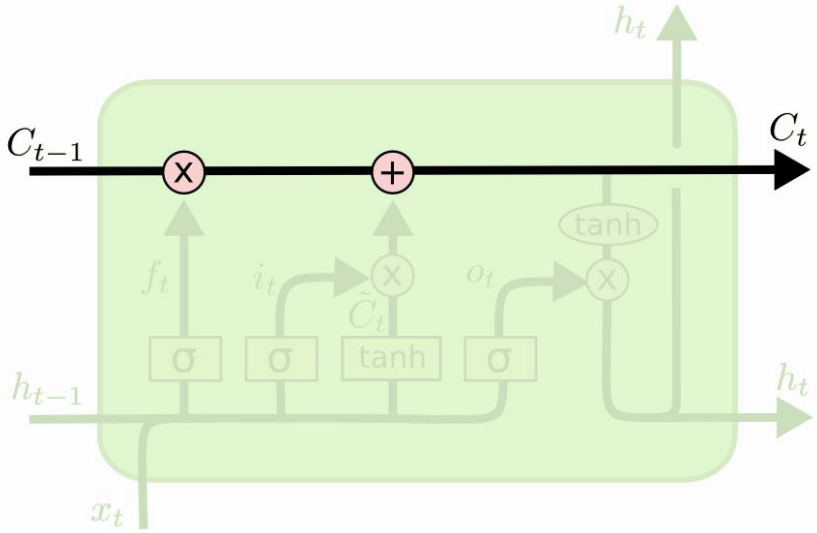
Soma (elemento a elemento dois vetores)



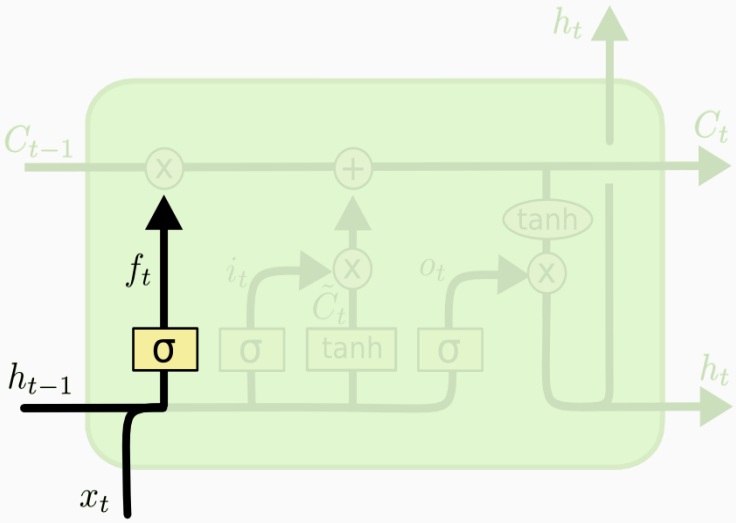
Visão Geral de uma Unidade LSTM

Fonte imagens: [colah's blog]

O Estado mantido C passa de t-1 a t com possíveis alterações...

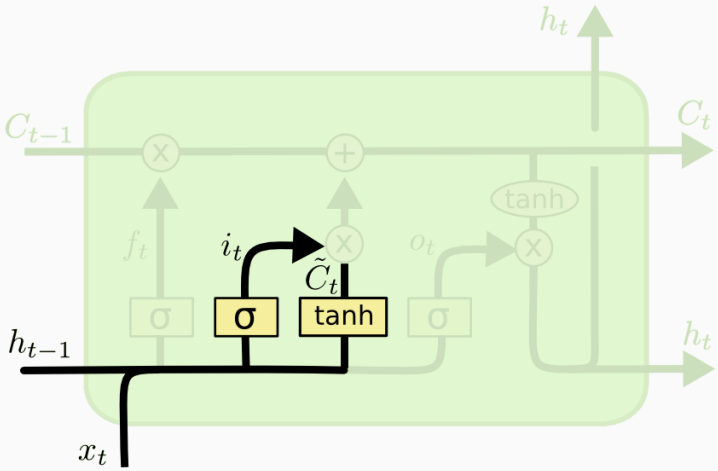


O **“portão forget”** filtra o que vai continuar e o que vai sumir da memória do estado anterior



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

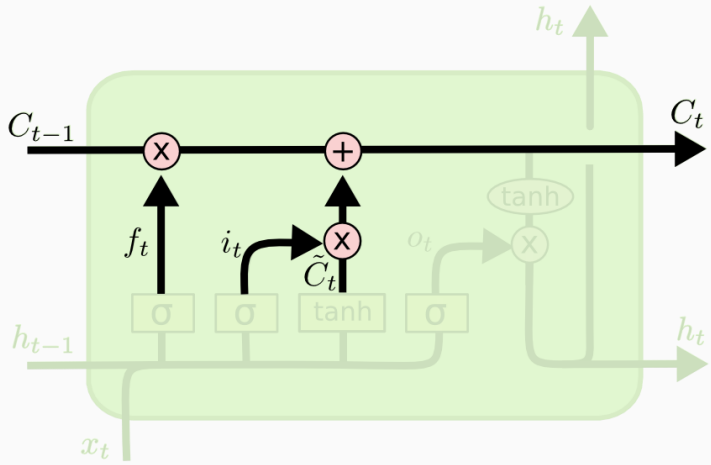
O **“portão input”** filtra o quanto da nova proposta de conteúdo ($\tilde{C}_t$) será combinado com o conteúdo anterior (que já passou pelo filtro “forget”).



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

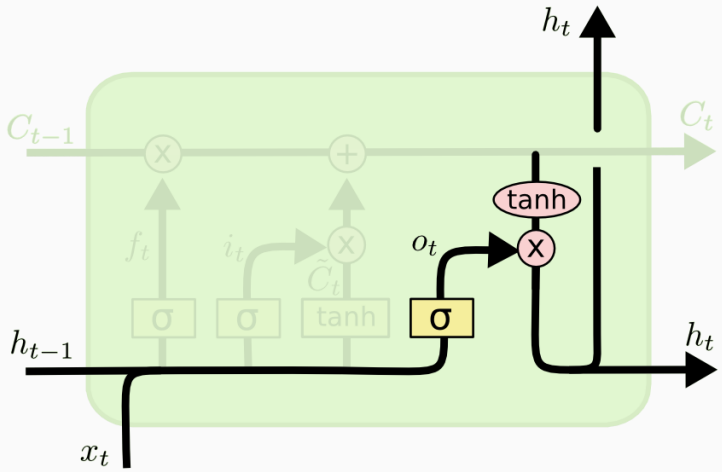
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

O novo conteúdo da memória é a combinação do conteúdo anterior filtrado pelo “forget” e pelo novo, filtrado pelo input.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

O filtro “output”, define quanto da tanh do novo conteúdo se transformará no novo output...



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

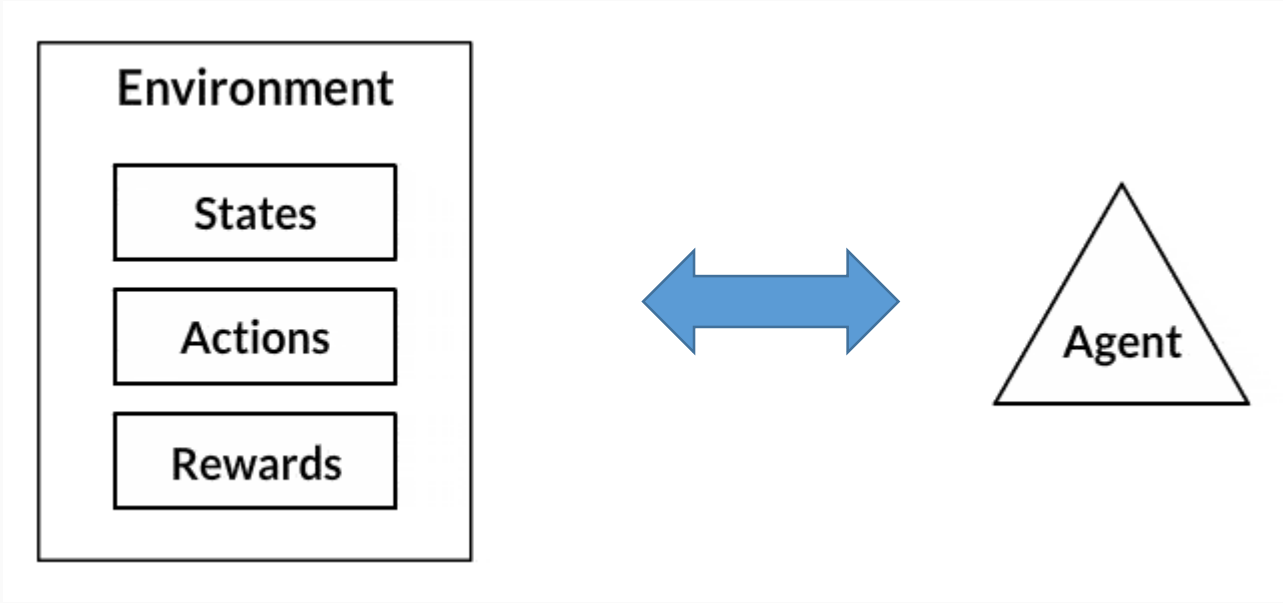
$$h_t = o_t * \tanh\left(C_t\right)$$

Com base em **LSTM-keras.ipynb:**

1- Analisar o código (geração das janelas temporais)
2- Fazer gráfico comparativo (valor real x estimado)

Reinforcement Learning

No aprendizado por reforço, diferentemente de outras técnicas de ML vistas no curso, o aprendizado se dá pela exploração de um ambiente em busca de recompensas.



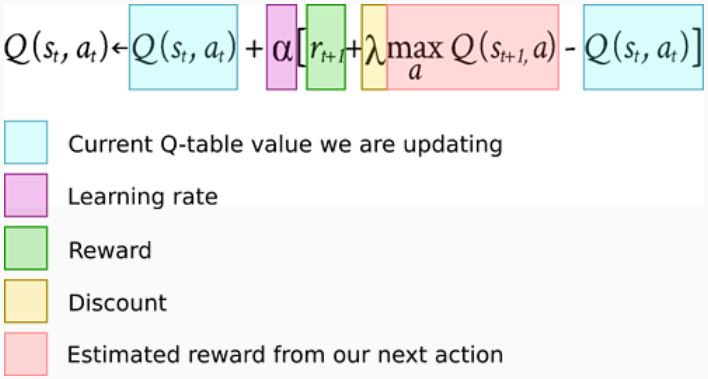
Reinforcement Learning

Conteúdo

Uma técnica popular de RL é o Q-Learning. A ideia é maximizar as recompensas obtidas. Isso acontece por experiências adquiridas na exploração do ambiente.

Estando em um estado St, em função da ação at a ser tomada (dentre as possíveis), sabemos a "qualidade" dessa ação: Q(st,at). Daí podemos escolher entre a que (pelo nosso conhecimento até o momento) é a melhor (exploit) ou arriscar novos caminhos (explore). Há várias políticas para isso, umas mais gulosas que outras.

Quando passamos do estado st para o estado st+1, a qualidade de Q(st,at) é atualizada conforme a fórmula de Bellman.



Discount é um fator que indica o quanto o futuro é importante. Discount menor, mais gulosa é a política (pouco importa o futuro).

Para implementarmos essa técnica, precisamos:

1) Obter a informação de ações x estados possíveis, a partir de um estado.

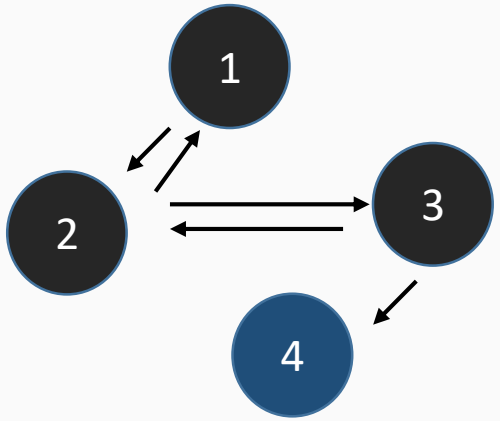
2) Manter atualizada a informação Q(st,at). Isso pode ser feito com uma matriz Q, com estados nas linhas, ações nas colunas.

3) Ter uma política para a escolha da próxima ação (seguir a orientação de Q(st,at) (exploit) ou arriscar caminhos novos (explore))? Podemos gerar um número randômico para decidir isso. Quanto mais optarmos pela orientação de Q(st,at), mais seremos “exploit”..isso pode não nos levar a maximizar as recompensas.

4) Definir recompensas para algumas situações (estados atingidos).

Exemplo

Há quatro estados possíveis: 1,2,3,4



Recompensas x estados

Matriz Q (estados x ações)

Ação> Estado	Direita	Esquerda
1	0.3	0.0
2	0.1	0.7
3	0.0	0.3
4	0.0	0.0

Política: 90% exploit, 10% explore

Estado Inicial : 1, Terminal: 4

Partindo de **Reinfrocement-simple.ipynb**

Atividades:

1) Analisar código
2) Mostrar a cada nova ação a matriz Q atualizada
3) Deixar a estratégia mais exploradora
4) Alterar learning rate e observar resultados
5) Reduzir gamma e verificar resultados

Código original : https://github.com/MorvanZhou


That's all Folks!

Cursos com Alta Performance de Aprendizado