

XI Encontro de Física e Astronomia da UFSC

Introdução às redes neurais com Python e Keras

Prof. Dr. Robson da Silva Oliboni



Introdução às redes neurais com Python e Keras

Redes neurais recorrentes (RNNs)

Sumário

1. Séries temporais
2. Neurônios recorrentes e RNNs
3. Células LSTM
4. Células GRU
5. RNNs bidirecionais

Séries temporais (*timeseries*) podem ser qualquer dado obtido via medidas em intervalos regulares; nesses dados, é essencial entender a dinâmica do sistema.

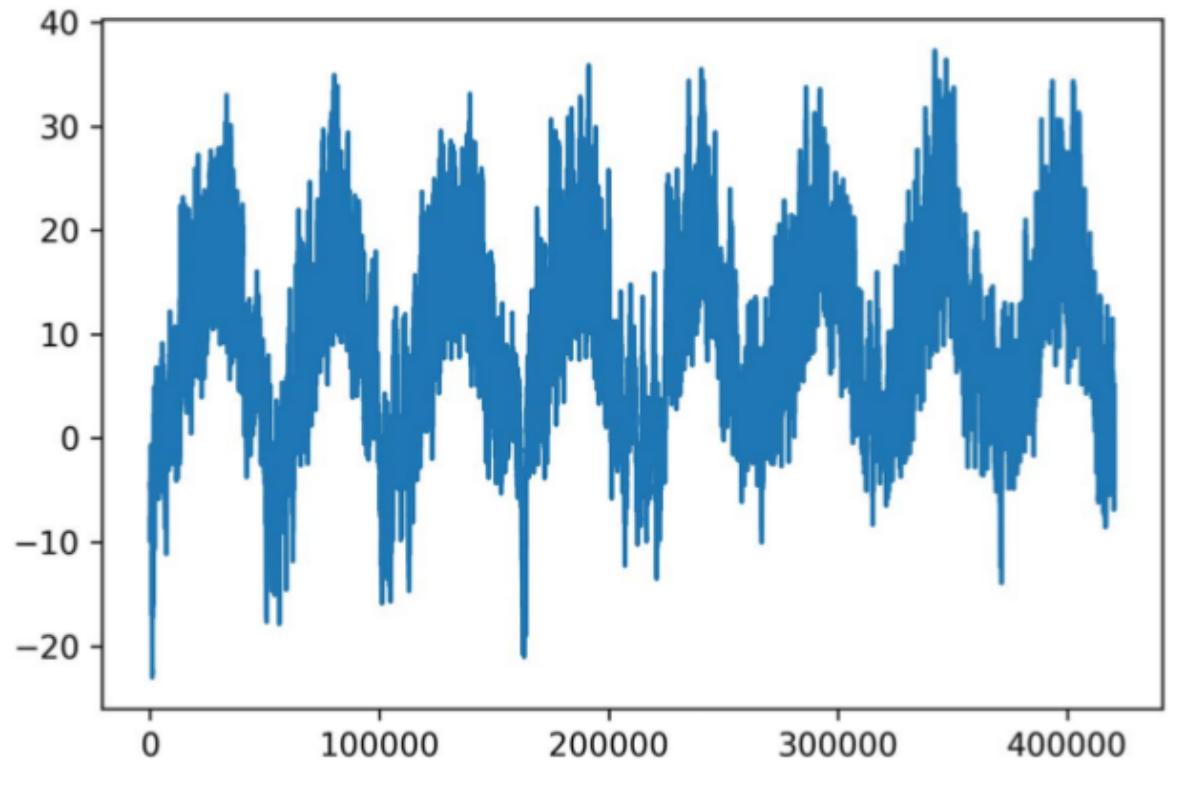


Figure 10.1 Temperature over the full temporal range of the dataset (°C)

Podem ser utilizadas técnicas de classificação, detecção de eventos e anomalias, etc.

Problemas com o uso de redes FF em sequências

Redes neurais *feedforward* (FF): Não possuem memória.

A inteligência biológica processa informação de forma incremental enquanto mantém um modelo interno do que está processando, construído de informação passada e constantemente atualizada à medida que chega nova informação.

Uma **rede neural recorrente** (RNN) adota o mesmo princípio: Ela processa sequências ao iterar através dos elementos da sequência e mantém um estado que contém informação relativa ao que foi processado até então.

Problemas com o uso de redes FF em sequências

Redes neurais recorrentes (RNNs) podem ser usadas em sequências de dados de quaisquer tamanhos, em vez de serem usadas em sequência de dados de tamanho fixo como em todas as outras redes.

Uma rede neural recorrente se parece muito com uma rede neural *feedforward*, exceto que também tem conexões *backward*.

Um único neurônio recorrente recebe entradas, gera uma saída e envia essa saída de volta para si mesmo. Em cada intervalo de tempo ou passo de tempo t (*frame*), este neurônio recorrente recebe as entradas, $\mathbf{X}_{(t)}$, bem como seu próprio intervalo de tempo anterior, $y_{(t-1)}$.

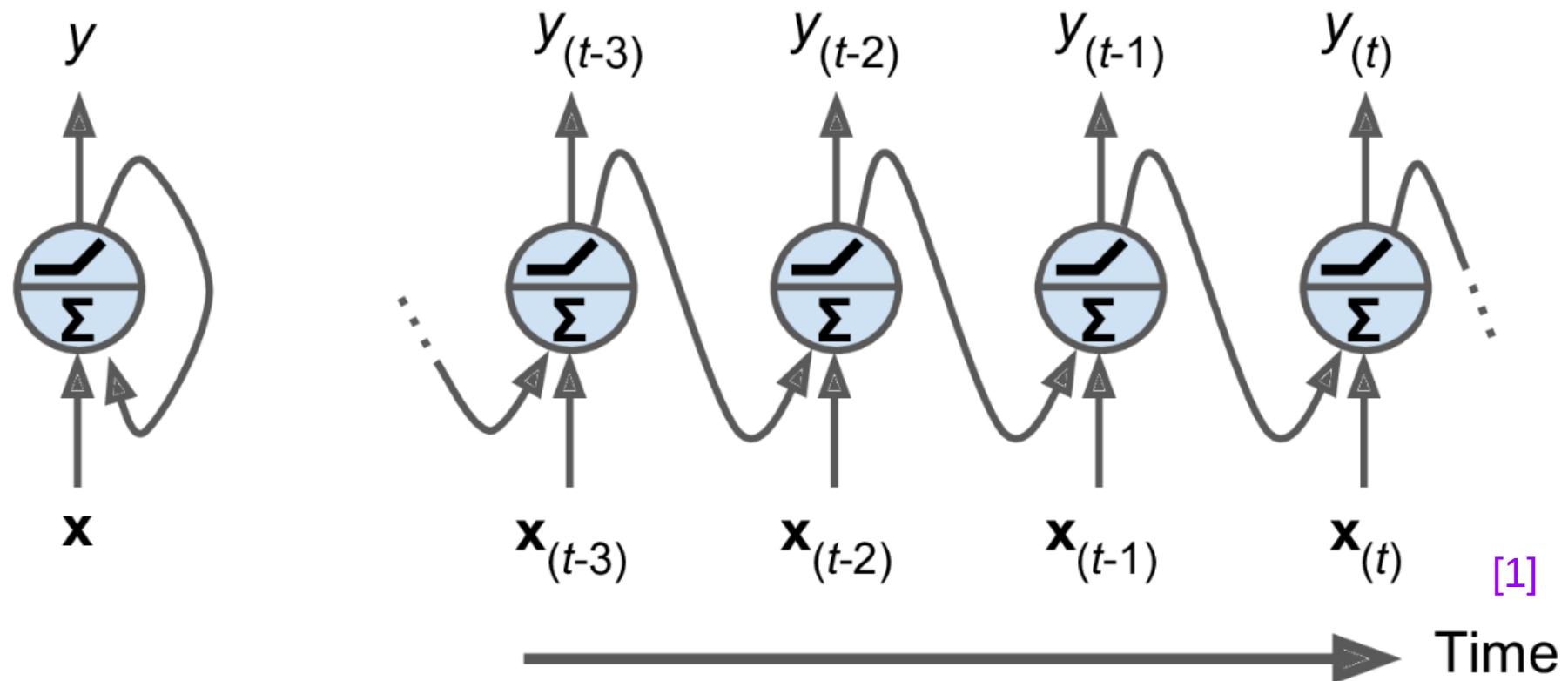


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

Em relação ao eixo do tempo, à direita: *unrolling the network through time*.

Uma camada de neurônios recorrentes. Agora, as saídas também são vetores.

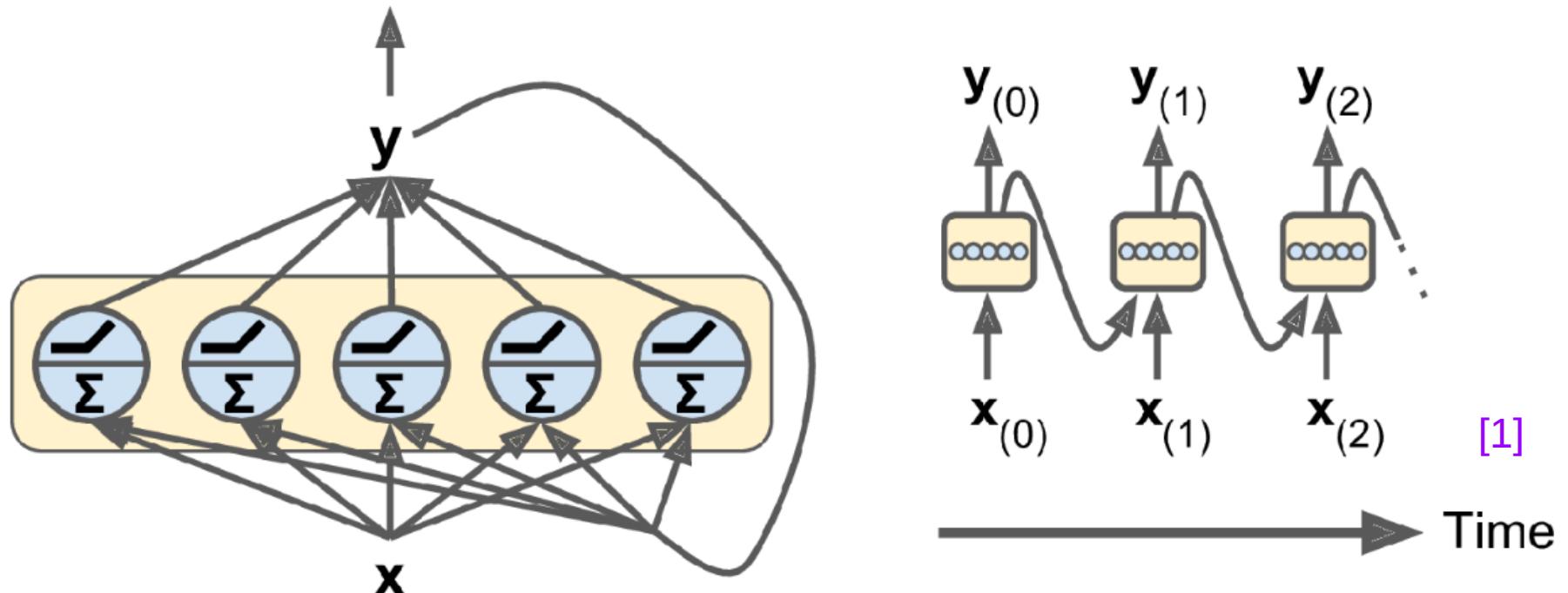


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Cada neurônio recorrente tem dois conjuntos de pesos: um para as entradas $\mathbf{X}_{(t)}$ e outro para as saídas do intervalo anterior, $\mathbf{y}_{(t-1)}$. Se considerarmos toda a camada recorrente em vez de somente um neurônio recorrente, podemos colocar todos os vetores de peso \mathbf{w}_x e \mathbf{w}_y em duas matrizes de peso, \mathbf{W}_x e \mathbf{W}_y .

$$\mathbf{y}_{(t)} = \phi(\mathbf{W}_x^\top \mathbf{x}_{(t)} + \mathbf{W}_y^\top \mathbf{y}_{(t-1)} + \mathbf{b})$$

Assim como as redes neurais *feedforward*, podemos calcular a saída de uma camada recorrente de uma só vez para cada mini-batch inteiro ao inserir todas as entradas no intervalo de tempo t em uma matriz de entrada $\mathbf{X}_{(t)}$:

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi(\mathbf{X}_{(t)} \mathbf{W}_x + \mathbf{Y}_{(t-1)} \mathbf{W}_y + \mathbf{b}) \\ &= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}\end{aligned}$$

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- \mathbf{W}_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- \mathbf{W}_y is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- \mathbf{b} is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices \mathbf{W}_x and \mathbf{W}_y are often concatenated vertically into a single weight matrix \mathbf{W} of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of [Equation 15-2](#)).
- The notation $[\mathbf{X}_{(t)} \mathbf{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$.

A saída $\mathbf{Y}_{(t)}$ é uma função de $\mathbf{X}_{(t)}$ e $\mathbf{Y}_{(t-1)}$, que é uma função de $\mathbf{X}_{(t-1)}$ e $\mathbf{Y}_{(t-2)}$, que por sua vez é uma função de $\mathbf{X}_{(t-2)}$ e $\mathbf{Y}_{(t-3)}$, e assim por diante. Isso faz com que $\mathbf{Y}_{(t)}$ seja uma função de todas as entradas.

Pode-se dizer que um neurônio recorrente tem uma forma de *memória*. A parte de uma rede neural que preserva algum estado em intervalos de tempo se chama células de memória (ou simplesmente *células*).

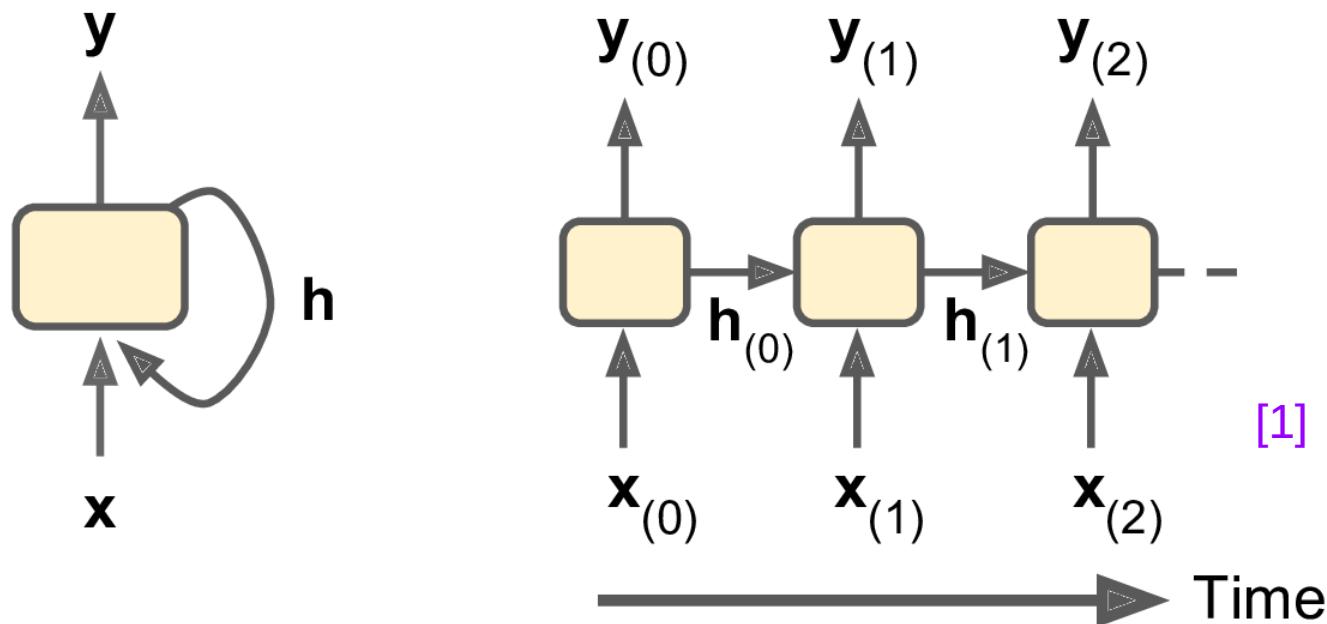
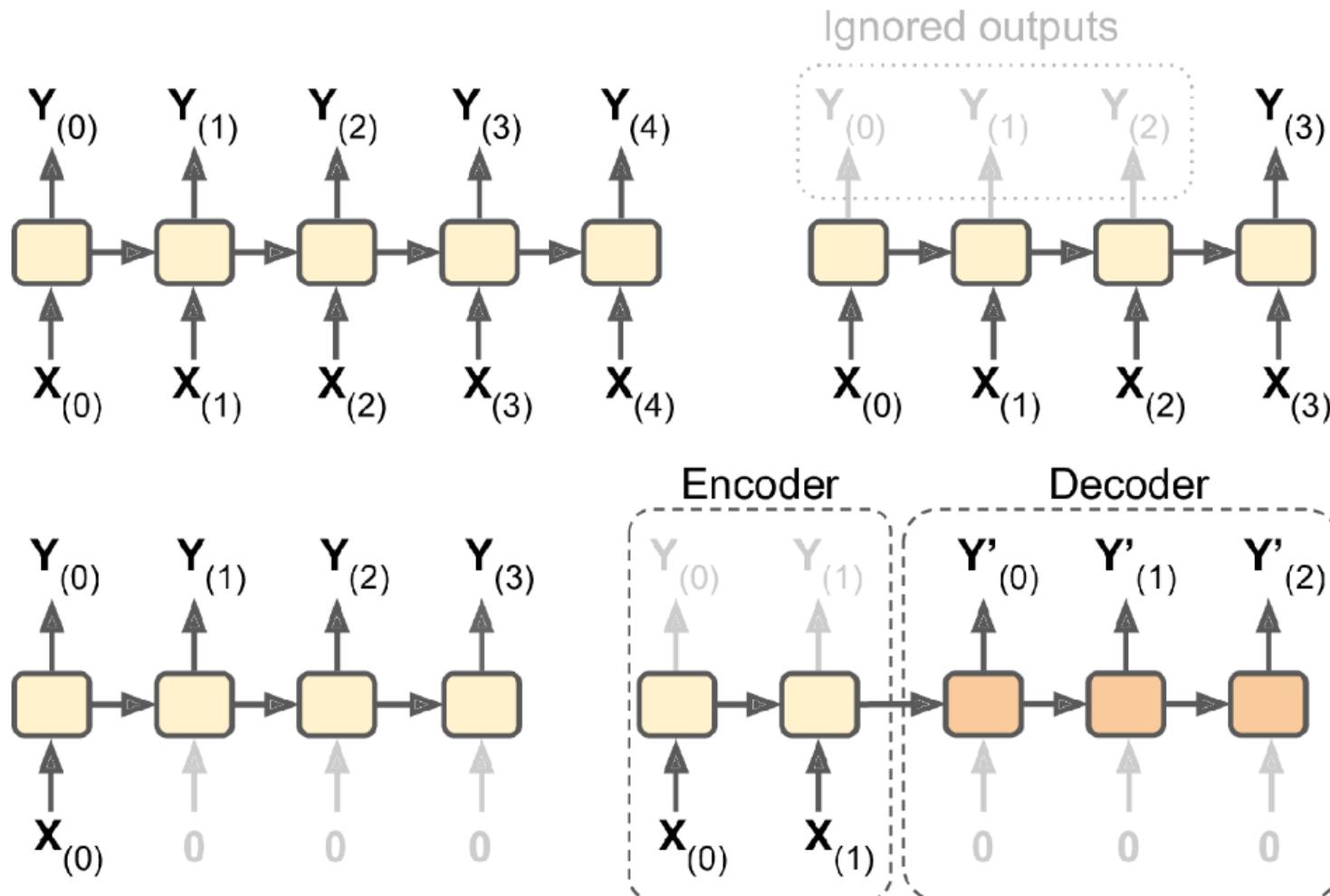


Figure 15-3. A cell's hidden state and its output may be different

Em geral, o estado de uma célula em um intervalo de tempo t , representado por $h_{(t)}$, é uma função de algumas entradas nesse intervalo de tempo e estado específicos do intervalo de tempo anterior: $h_{(t)} = f(h_{(t-1)}, x_{(t)})$.

Sequências de entrada e saída



Redes sequência à sequência, sequência a vetor, vetor à sequência e redes codificador-decodificador.

Entendendo as redes neurais recorrentes

Uma RNN simples processa uma sequência ao iterar na sequência de elementos e mantendo um estado que contém informação sobre o que já foi processado.

Uma RNN simples, “desenrolada” no tempo:

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```

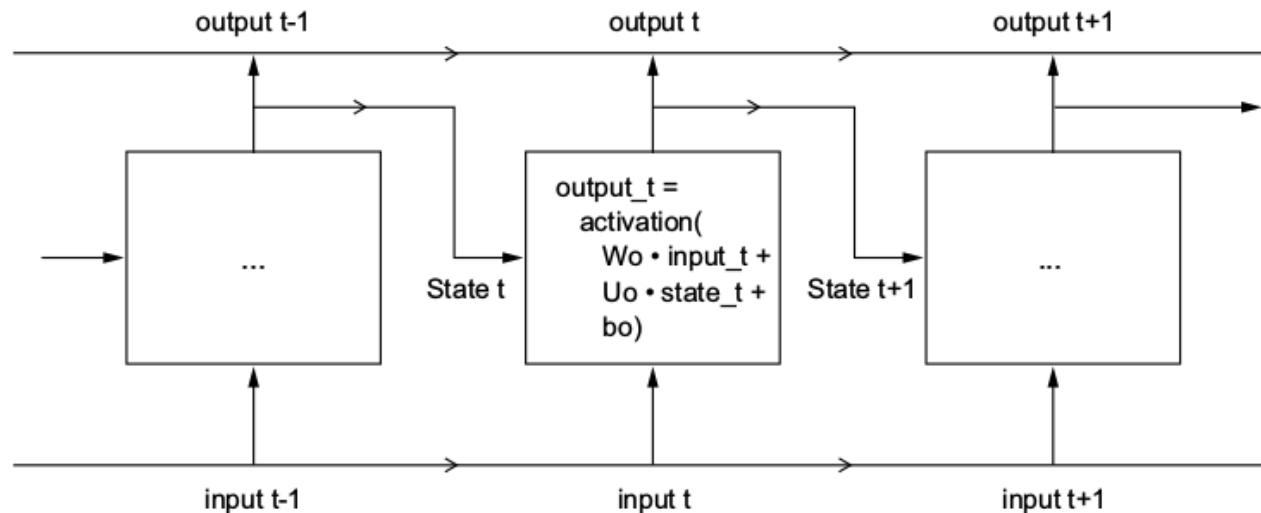


Figure 10.7 A simple RNN, unrolled over time

[2]

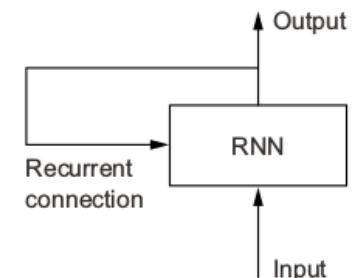


Figure 10.6 A recurrent network: a network with a loop

[2]

RNN simples no Keras: A camada SimpleRNN

No Keras, a camada processa batches de sequências, ou seja, inputs de formato (batch_size, timesteps, input_features).

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

[2]

Timestep não especificado na camada Input, para lidar com sequências de tamanho arbitrário.

As camadas recorrentes no Keras podem rodar em **dois modos diferentes**: Podem retornar sequências inteiras de outputs sucessivos para cada timestep (tensor de rank-3 de formato (batch_size, timesteps, output_features)) ou retornar apenas o último output de cada sequência de input (tensor de rank-2 de formato (batch_size, output_features)).

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
>>> print(outputs.shape)
(None, 16)
```

[2]

Note that
return_sequences=False
is the default.

RNN simples no Keras: A camada SimpleRNN

O exemplo abaixo retorna o estado completo da sequência:

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
>>> print(outputs.shape)
(120, 16) [2]
```

É normal empilhar várias redes recorrentes para aumentar o poder de representação da rede. Assim, deve-retornar a sequência completa nas redes intermediárias:

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x) [2]
```

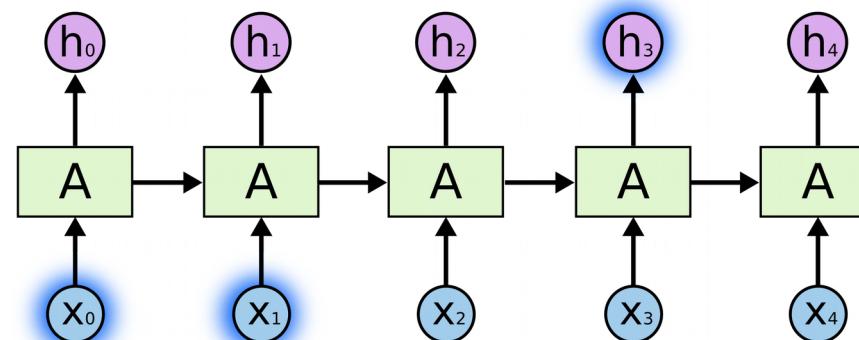
Problemas das RNN simples



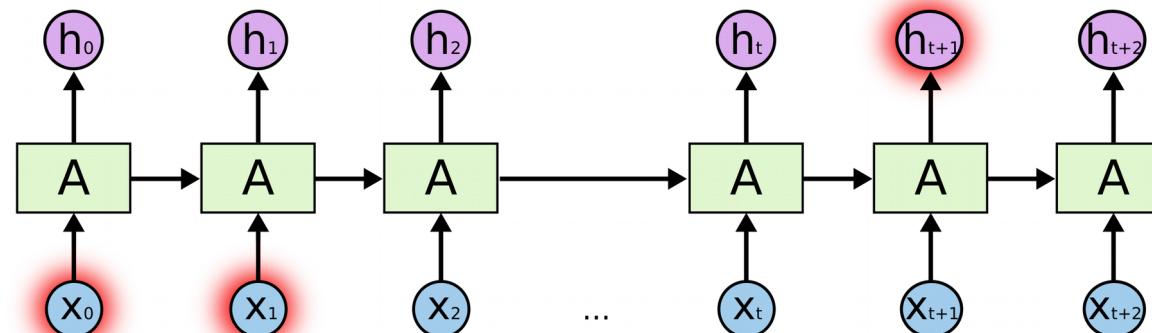
Teoricamente a RNN simples deveria reter informação sobre o passado, porém na prática a rede não aprende dependências de longo prazo.

O problema está relacionado com o vanishing de gradientes das redes FF.

Se o gap de informação é pequeno na sequência, a rede preserva certa memória.



Quando o gap é longo (dependência de longo prazo), há perdas de aprendizado.

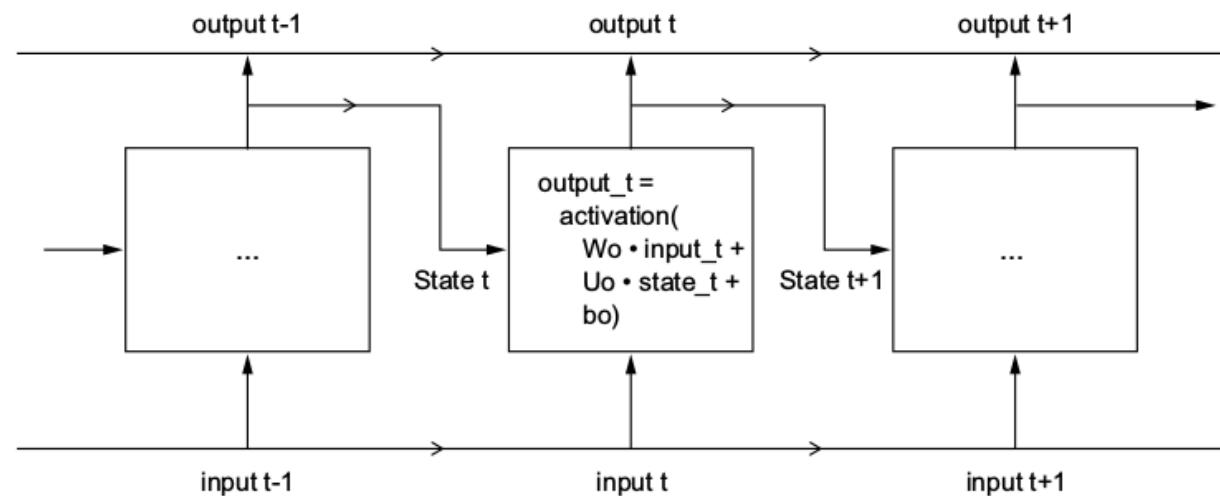


A rede LSTM

A técnica *long short-term memory* (LSTM) foi desenvolvida por Hochreiter e Schmidhuber em 1997. [https://homl.info/93]. Aprimorada por várias pessoas, como Alex Graves [https://homl.info/graves] e Hasim Sak [https://homl.info/94].

Essa rede adiciona uma forma de carregar informação ao longo dos frames, como uma esteira de aeroporto; ela salva informação para depois, evitando que sinais antigos gradualmente se extingam durante o processo.

Começamos representando uma *RNN simple*. Como teremos vários pesos, adicionamos um subscrito para os pesos do output:

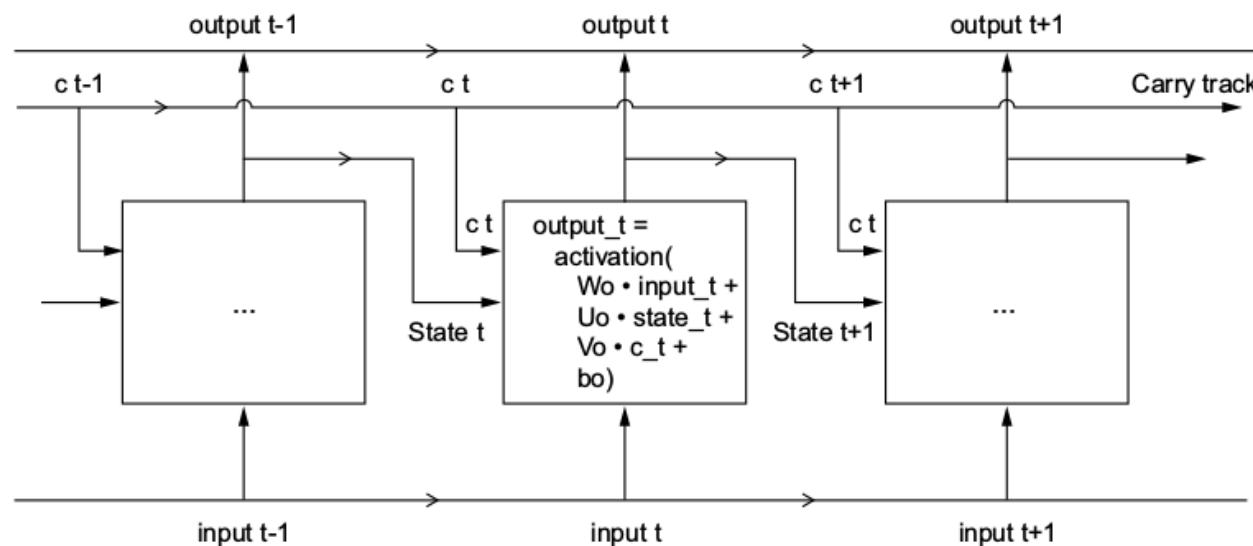


[2]

A rede LSTM

Adicionamos à rede simples um fluxo de dados adicional que carrega informação ao longo dos frames (c_t , C=carry).

Essa informação será combinada com a conexão de input e irá afetar o estado enviado ao próximo frame (via uma função de ativação e uma operação de multiplicação). Conceitualmente, o c_t é uma forma de modular o próximo output e o próximo estado.



[2]

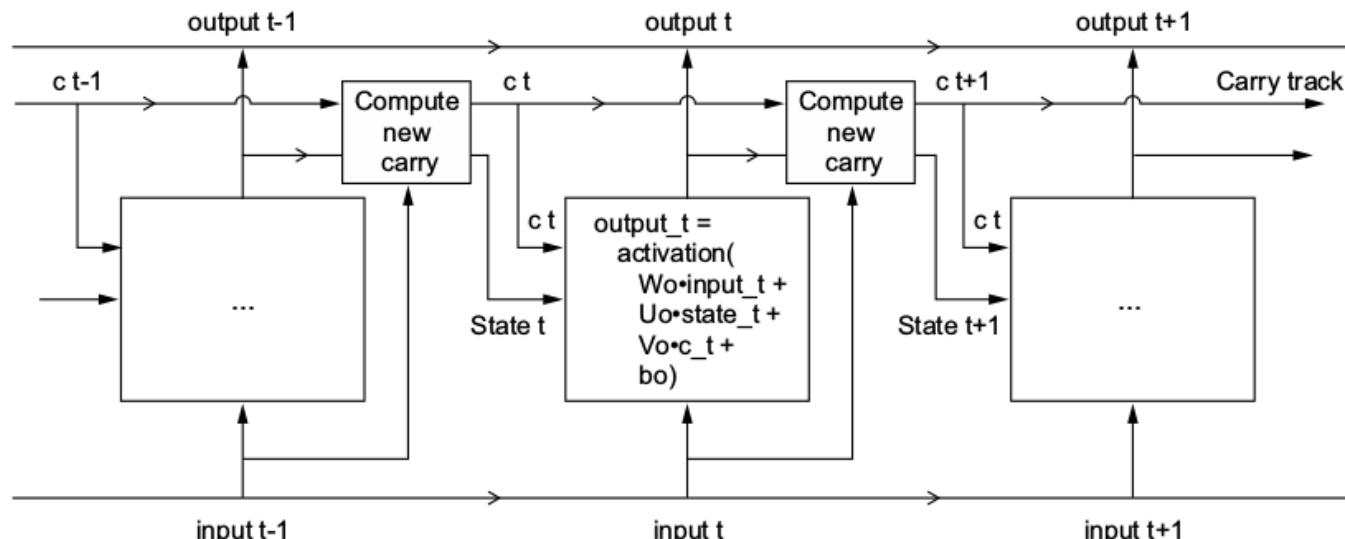
A forma que o próximo valor do *carry* é calculado envolve três transformações distintas, todas na forma de uma célula RNN simples:

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk) [2]
```

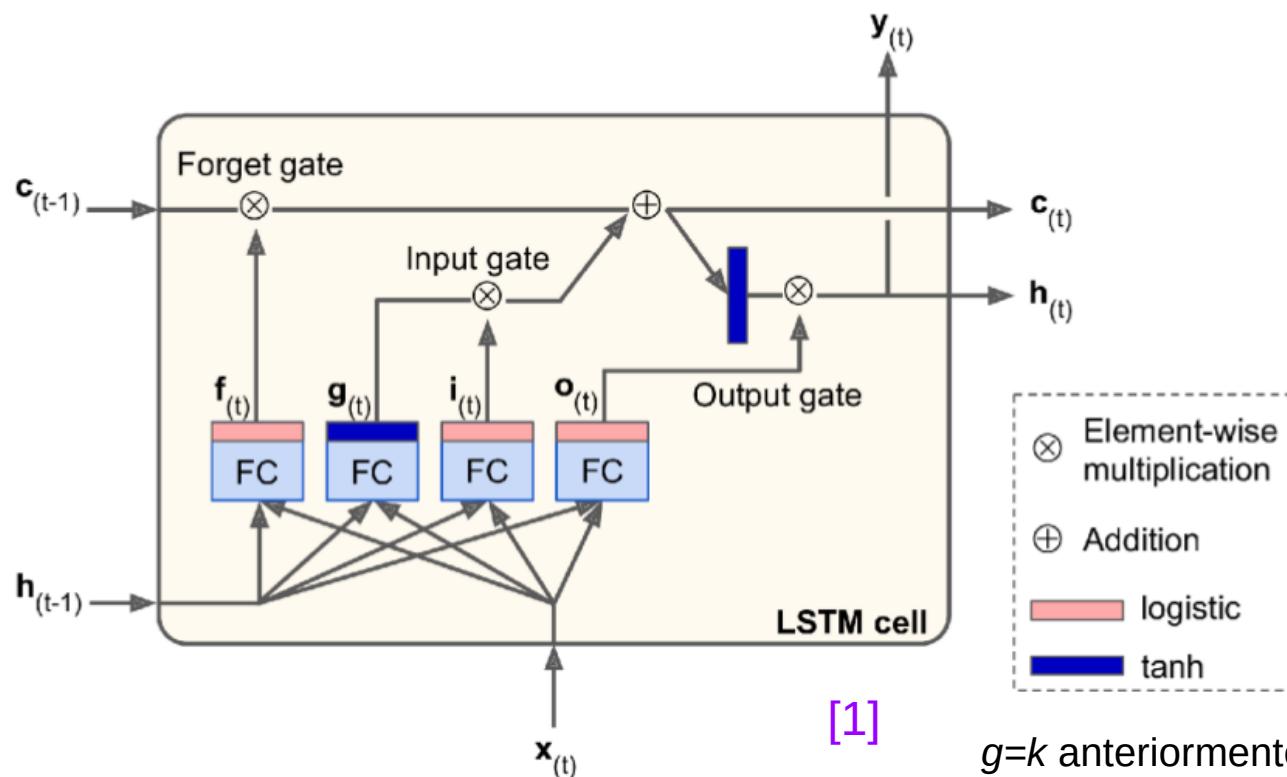
O novo *carry state* é obtido pela combinação dessas equações:

$$c_{t+1} = i_t * k_t + c_t * f_t [2]$$

No **Keras**, uma LSTM é definida pela camada keras.layers.LSTM.



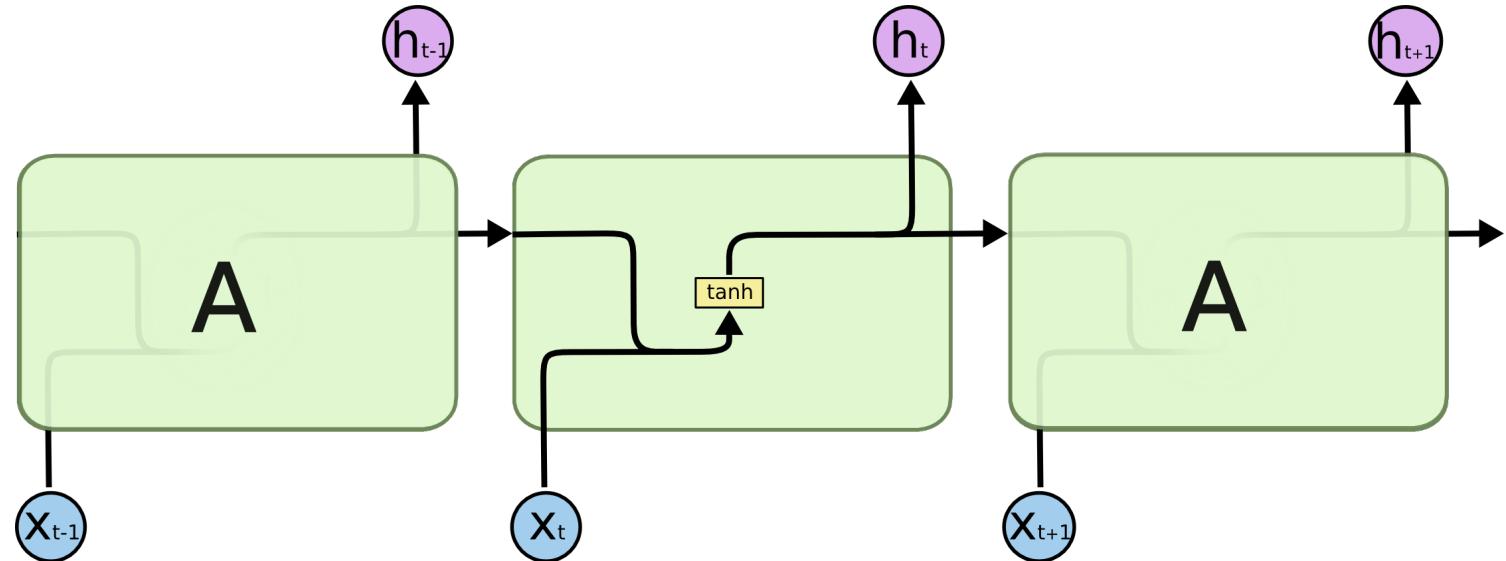
[2]



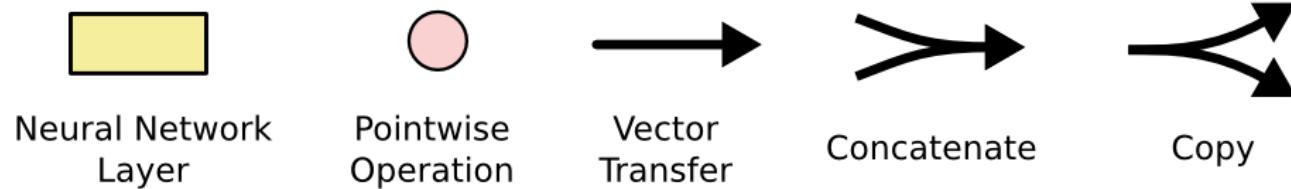
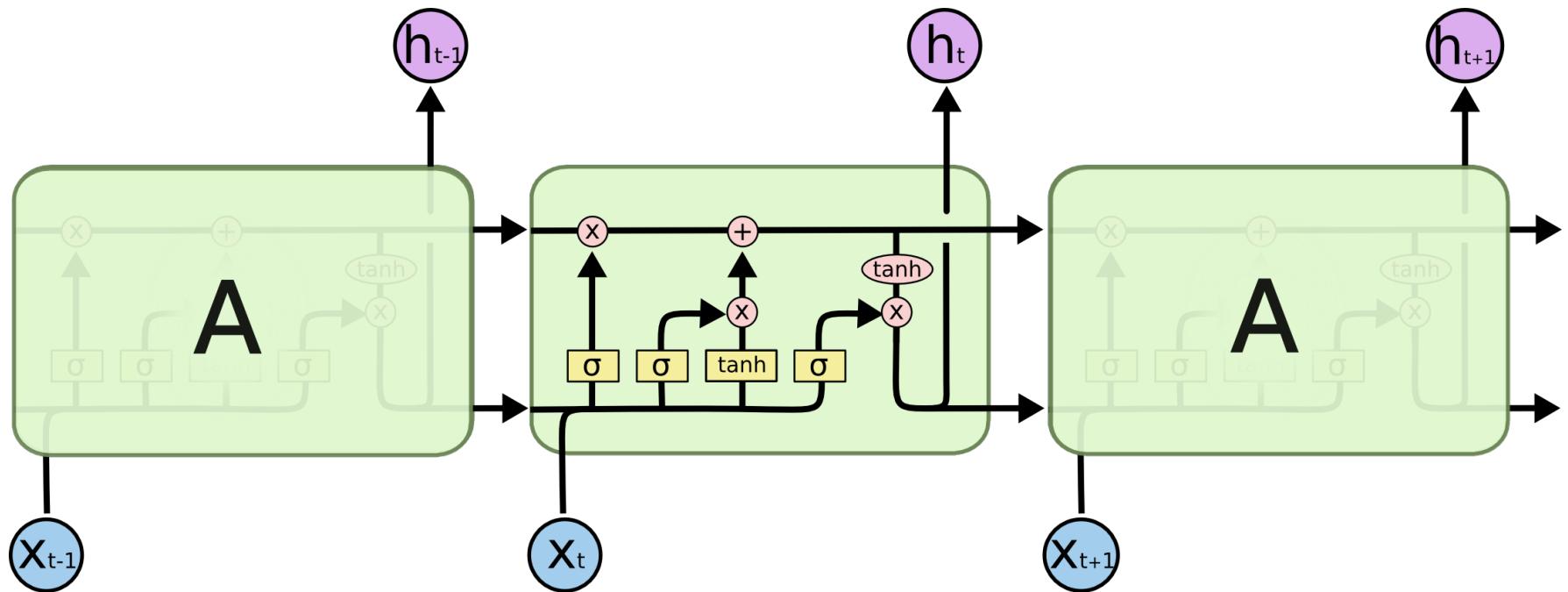
A anatomia de uma camada LSTM

[<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>]

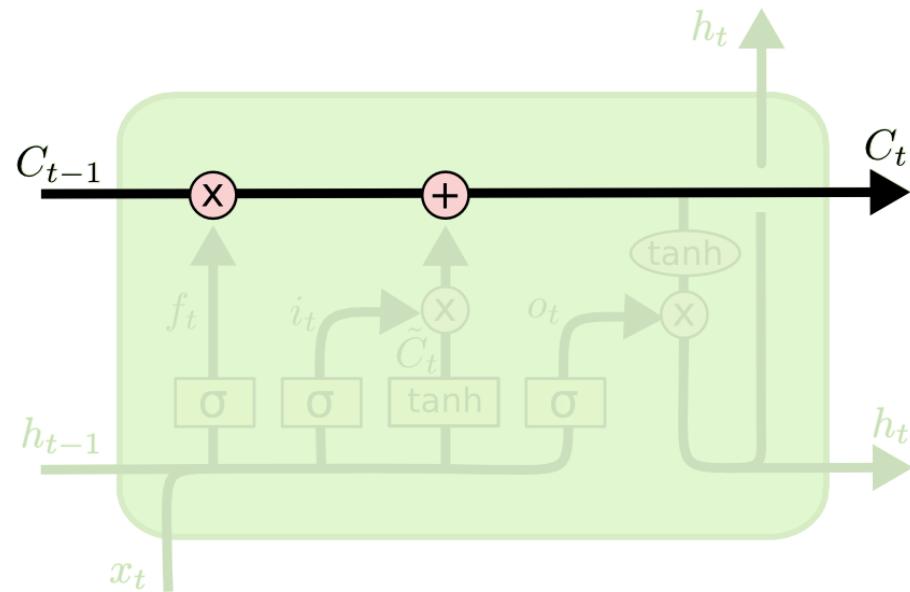
Em uma **RNN simples**, o “módulo” de repetição contém uma única camada:



Em uma **LSTM** simples, o “módulo” de repetição contém quatro camadas interagentes:

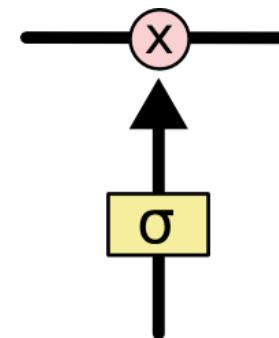


Ideia principal da **LSTM**: O “carry state”, c_t :



A LSTM tem a habilidade de adicionar ou remover informação do estado c_t através de estruturas chamadas *gates* (portas).

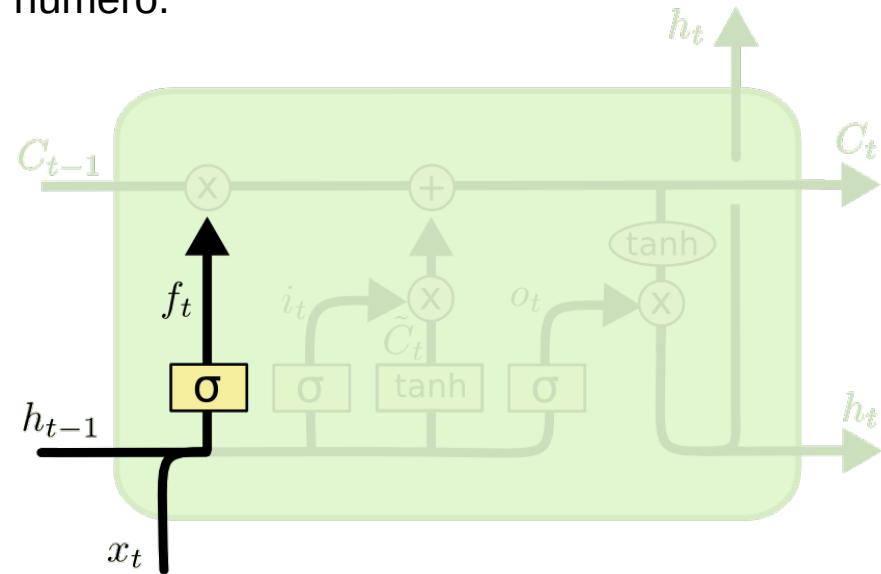
Os *gates* são maneiras de otimizar a transmissão de informação.
Eles são compostos de uma função de ativação (sigmoide) e uma operação de multiplicação.
A saída da sigmoide é entre 0 e 1, que gera o quanto de cada componente pode ser adicionado/removido.
A LSTM possui três *gates*.



Primeiro gate: *Forget*, para decidir qual informação será descarta no estado c_t .

Esse *gate* pega x_t e h_{t-1} e gera um número entre 0 e 1 para cada número no estado c_{t-1} .

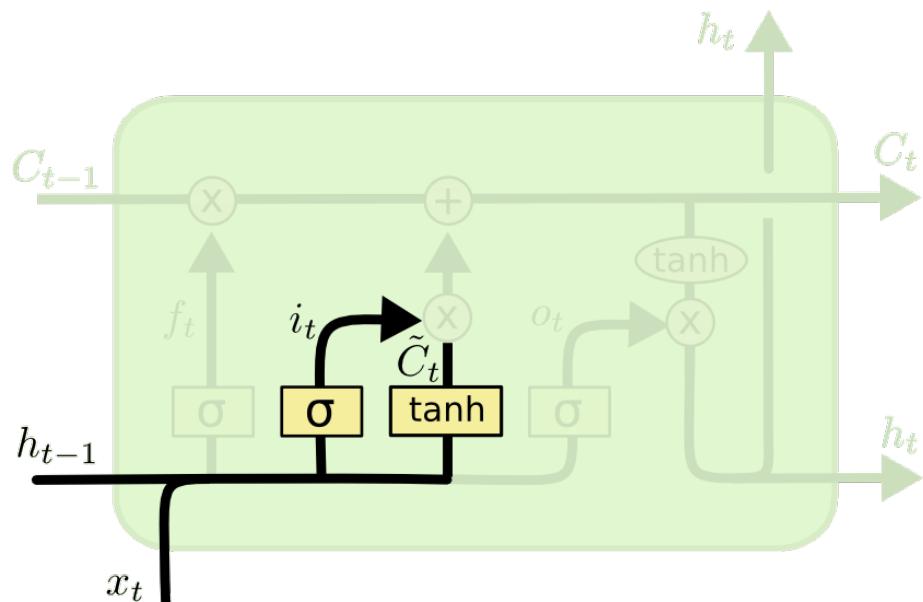
No caso, 1 significaria mantenha esse número totalmente, e 0 esqueça completamente esse número.



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

Em seguida, é decidida qual informação será mantida do estado atual.

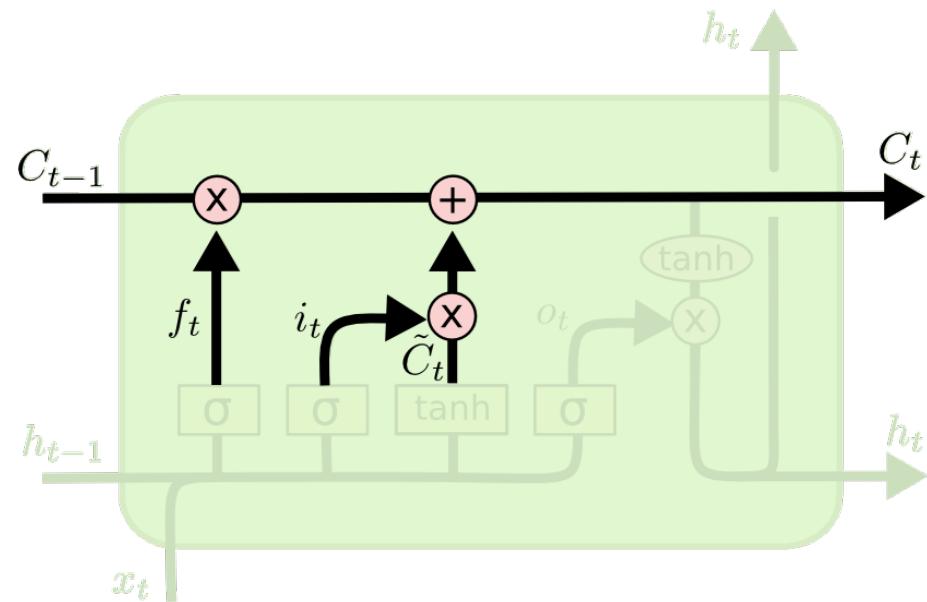
Possui duas partes: Primeiro, uma camada com sigmoide, o *input gate*, decide quais valores serão atualizados. Depois, uma camada tanh crie uma nova candidata c_t que poderá ser adicionada ao estado. Depois, as duas são combinadas.



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

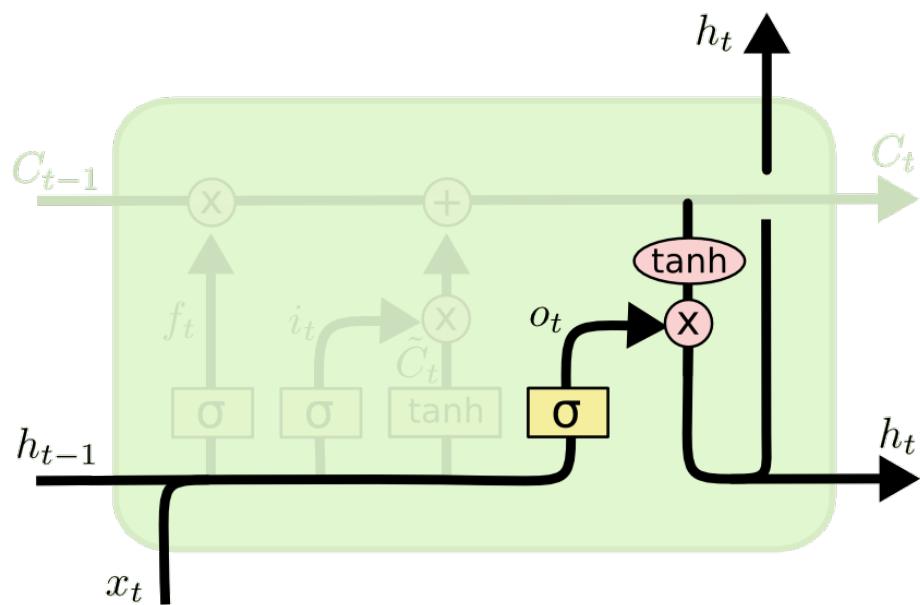
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Agora fazemos a atualização do estado c_t . Multiplicamos o estado anterior por f_t , para “esquecer” certos valores antigos. Então adicionamos $i_t * \tilde{C}_t$, os novos candidatos a valores escalonados por quanto decidimos atualizar cada valor.



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

Finalmente calculamos o output. Ele se baseará no estado c_t , mas uma versão filtrada. Primeiro, aplicamos uma camada sigmoide na célula para decidir quais partes entrarão no output. Depois, aplicamos uma camada tanh no estado c_t e multiplicamos pelo output da gate sigmoide, assim geramos apenas os outputs das partes que decidimos.



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

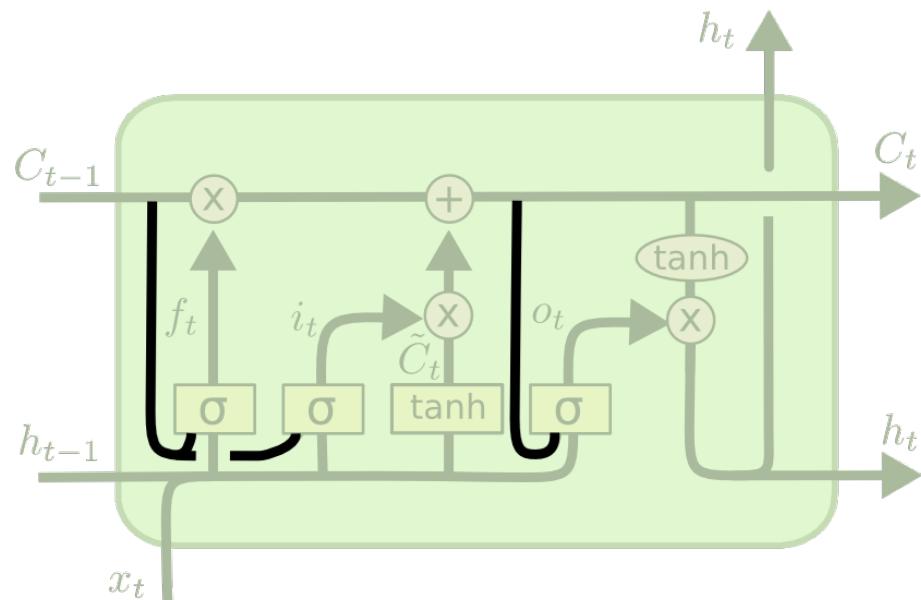
$$h_t = o_t * \tanh (C_t)$$



Variações da LSTM: Peephole connections

Nesta variação, permitimos às camadas sigmoides que interajam com o estado c_t .

[<https://homl.info/96>]



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

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

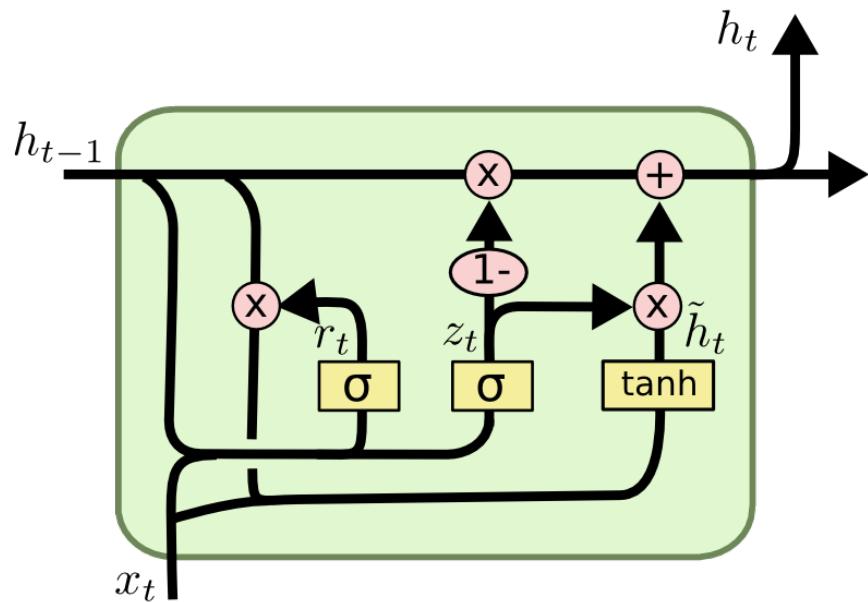
$$o_t = \sigma (W_o \cdot [C_t, h_{t-1}, x_t] + b_o)$$

No **Keras**, há apenas a célula experimental `tf.keras.experimental.PeepholeLSTMCell`.

Variações da LSTM: Gate recurrent unit, GRU

A GRU combina as gates input e forget em um único *gate*, *update gate*. Também combina os estados *hidden* e *cell*, entre outras mudanças.

[<https://homl.info/97>]



$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

O **Keras** fornece a camada `keras.layers.GRU`.

Qual variação é melhor?

[<http://arxiv.org/pdf/1503.04069.pdf>]

[<http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf>]

Uso de RNNs: Gradientes instáveis

As RNNs desenroladas no tempo se assemelham a redes profundas, logo as técnicas para combate de gradientes instáveis é quase a mesma: Boa inicialização de parâmetros, otimizadores mais rápidos, *dropout* ...

Em especial, utilizar funções de ativação saturadoras! Padrão: Troca de ReLU por tanh.

A normalização de batch normalmente não é eficiente! [<https://homl.info/rnnbn>]
Alternativa: Normalização de camada [<https://homl.info/layernorm>]

No **Keras**, há a camada `keras.layers.LayerNormalization`

Dropout: No **Keras**, todas as camadas recorrentes tem o hiperparâmetro `recurrent_dropout`.

Nas camadas recorrentes, o mesmo *dropout mask* (o mesmo padrão de unidades dropadas) deve ser aplicado em cada frame; em GRUs e LSTMs, uma *dropout mask* temporariamente constante deve ser aplicada às ativações recorrentes internas da camadas.
[<https://www.cs.ox.ac.uk/people/yarin.gal/website/thesis/thesis.pdf>]

Exemplo: Previsão do tempo

Utilizaremos um modelo simples e redes neurais para fazer a previsão do tempo.

Quatro modelos:

- Um modelo simples para *baseline*;
- LSTM simples;
- LSTM com *dropout*;
- GRUs com *dropout* (2 camadas).

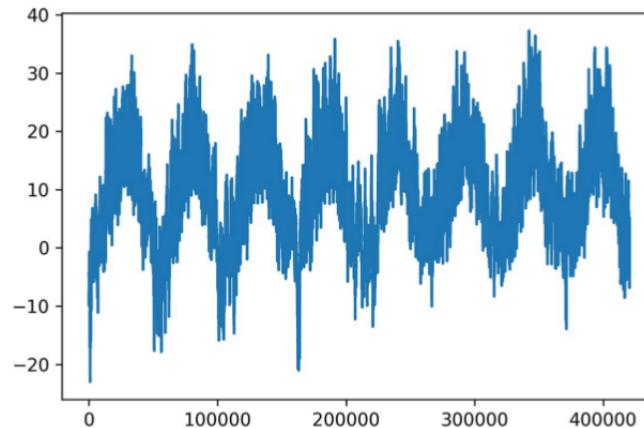


Figure 10.1 Temperature over the full temporal range of the dataset (°C)



RNNs.ipynb

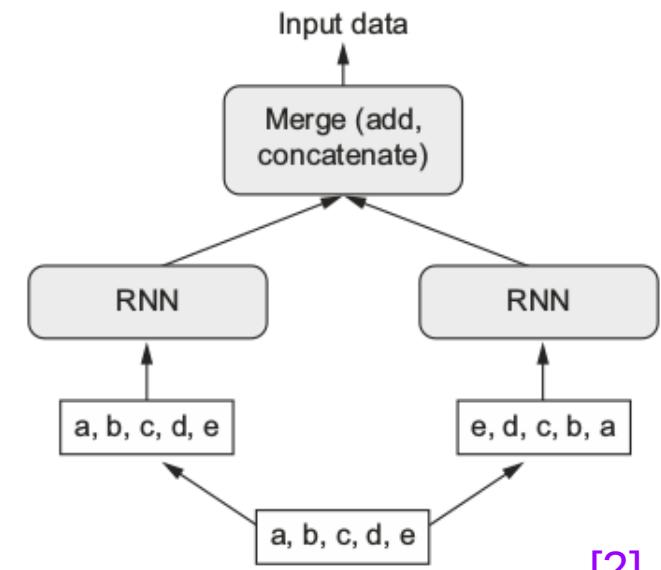
RNN bidirecional

“Swiss army knife of deep learning for natural language processing”

RNNs dependem da ordem da sequência: elas processam os frames de suas sequências de input em ordem, e reverter ou misturar os frames mudará completamente a representação que as RNNs extraem da sequência.

Uma RNN bidirecional explora a sensibilidade da ordem nas RNNs: Ela usa dois RNNs regulares para processar a sequência de input em uma direção e na direção reversa, e então funde as suas representações.

Ao processar uma sequência nas duas ordens, um RNN bidirecional pode obter padrões possivelmente ignorados por uma RNN unidirecional.



Uso da camada Bidirectional no Keras:

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                     epochs=10,
                     validation_data=val_dataset)
```

[2]

No caso de séries temporais, pode não ser tão eficiente → fenômeno no futuro pode depender mais dos dados mais atuais.

Exemplo

pubs.acs.org/jcim

Article

Bidirectional Molecule Generation with Recurrent Neural Networks

Francesca Grisoni,* Michael Moret, Robin Lingwood, and Gisbert Schneider*



Cite This: *J. Chem. Inf. Model.* 2020, 60, 1175–1183



Read Online

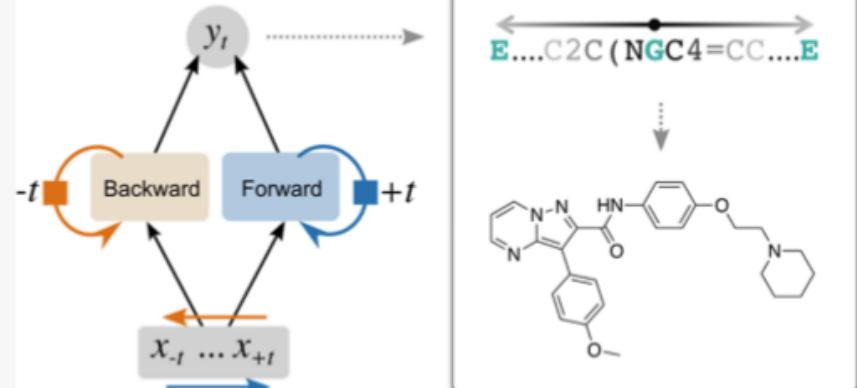
ACCESS |

Metrics & More

Article Recommendations

SI Supporting Information

ABSTRACT: Recurrent neural networks (RNNs) are able to generate de novo molecular designs using simplified molecular input line entry systems (SMILES) string representations of the chemical structure. RNN-based structure generation is usually performed unidirectionally, by growing SMILES strings from left to right. However, there is no natural start or end of a small molecule, and SMILES strings are intrinsically nonunivocal representations of molecular graphs. These properties motivate bidirectional structure generation. Here, bidirectional generative RNNs for SMILES-based molecule design are introduced. To this end, two established bidirectional methods were implemented, and a new method for SMILES string generation and data augmentation is introduced—the bidirectional molecule design by alternate learning (BIMODAL). These three bidirectional strategies were compared to the unidirectional forward RNN approach for SMILES string generation, in terms of the (i) novelty, (ii)



<https://doi.org/10.1021/acs.jcim.9b00943>

Exemplo

- Espaço químico: 10^{60} a 10^{100} moléculas;
- Várias variações de RNNs testadas;
- Sequência: caracteres da estrutura SMILES;
- ~ 272k compostos; 2 camadas de LSTMs, de 128-512 unidades, com dropout;
- Validação cruzada ($k=5, 10$);
- método BIMODAL (RNN bidirecional) mais promissor.

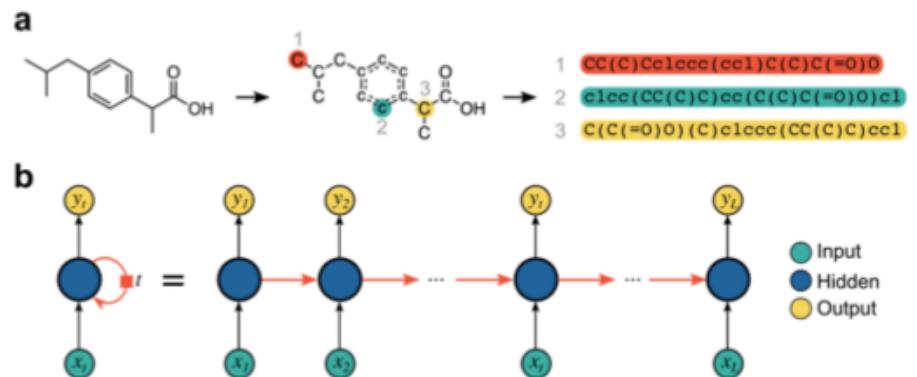


Figure 1. Overview of the basic concepts of this study. (a) SMILES strings, obtained from a molecular graph representation, where each atom is indicated by its element symbol, while branching and connectivity are indicated by symbols or lowercase letters (e.g., “(”), “=” and “c” for branching, double bonds, and aromatic carbons, respectively). Examples of three SMILES strings representing the drug ibuprofen are shown; the start atoms used for SMILES string production are indicated by gray numbers. (b) Simplified scheme of a forward RNN with one recurrent neuron layer. RNNs model a dynamic system, in which the network state at any t -th time point depends both on the current observation (x_t) and on the previous state (at $t - 1$) and is used to predict the output (y_t).

Exemplo

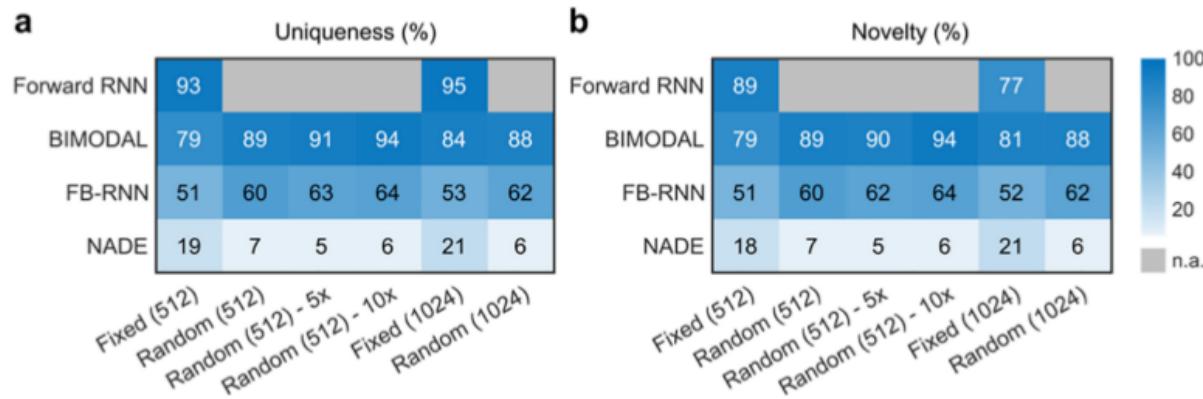


Figure 4. (a) Uniqueness and (b) novelty values (%) obtained by each investigated method (n.a. = not available). The generative models were tested with fixed and random starting points, two levels of augmentation (5- and 10-fold), and two network sizes (512 and 1024 hidden units).

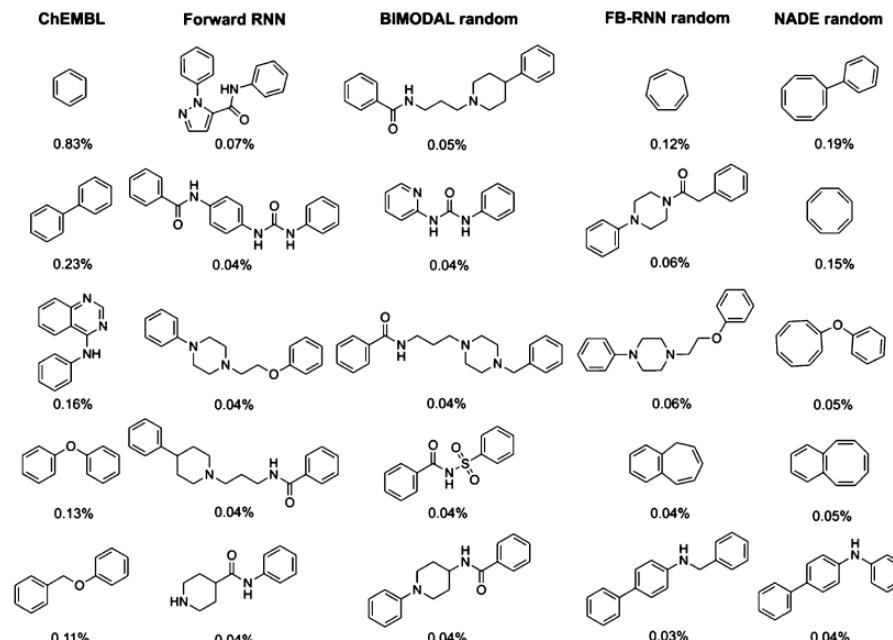


Figure 6. Most frequently generated scaffolds of each investigated model (512 hidden units and no data augmentation), in comparison with

Pelotas, Rio Grande do Sul (RS)



“Candy valley”





Computational chemistry group

Jonathan A. Fetter (undergrad student)

Daniel F. Pietezak (PhD student)



Acknowledgments



CNPq (Process number 409674/2021-4)



FAPERGS (grant number 21/2551-0002137-3)

Obrigado!

rsoliboni@ufpel.edu.br