Deep Learning

LSTMs e GRUs

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Overview

1. Problemas em Redes Recorrentes
2. LSTMs
3. GRUs

Visão Geral

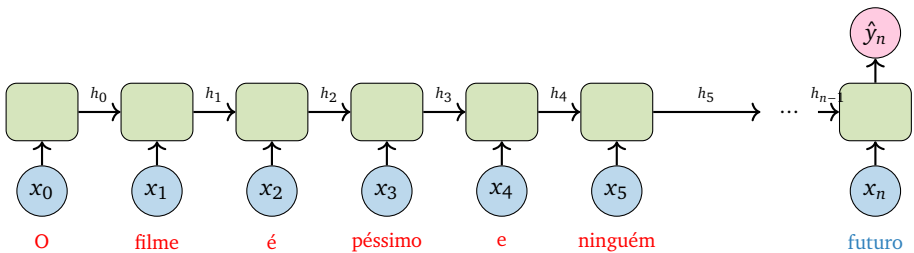
1. **Problemas em Redes Recorrentes**
2. **LSTMs**
3. **GRUs**

Memória de Curto Prazo

- Considere a frase a seguir, em que existe uma quebra de sentimento entre as duas partes:
- *"O filme é péssimo e ninguém deveria perder tempo assistindo. Contudo eu admiro a tentativa de Hollywood de aumentar a representatividade em termos de história e atores, certamente será possível colher frutos sobre essa tentativa em outras obras no futuro."*
- Para classificar o sentimento global, a rede precisa lembrar do início da frase quando chega ao final.
- Com uma RNN simples, esse tipo de **dependência de longo alcance** é difícil de manter.

Estado Interno é Sobrescrito a Cada Passo

- Em uma RNN *vanilla*, o estado interno h_t é totalmente recalculado a cada passo de tempo.



- Em cada célula a entrada atualiza completamente o estado.
- Manter informação por muitos passos de tempo é difícil, esse é o problema da **memória de curto prazo**.

Solução: Aprender Quando Atualizar e Quando Esquecer

- A ideia é deixar a própria rede aprender, a cada passo, quanta informação anterior deve ser mantida e quanta informação nova deve entrar no estado.
- Essa é a motivação da arquitetura *Long Short-Term Memory*[1] (LSTM).
- É a arquitetura com mais "partes" que veremos até agora, mas todas as partes se reduzem a operações já conhecidas:
  - multiplicação de matrizes,
  - funções de ativação (sigmoid, tanh),
  - operações ponto a ponto (soma, multiplicação).

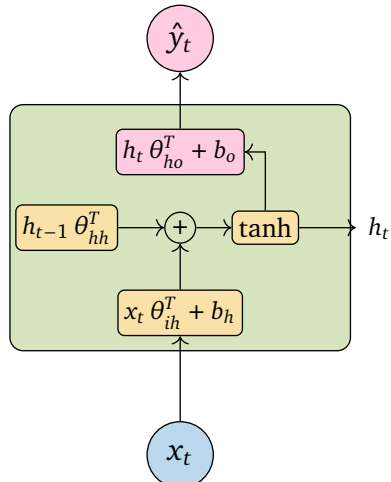
[1] Sepp Hochreiter e Jürgen Schmidhuber. "Long short-term memory". Em: Neural Computation 9.8 (1997), pp. 1735–1780.

Visão Geral

1. Problemas em Redes Recorrentes
2. **LSTMs**
3. GRUs

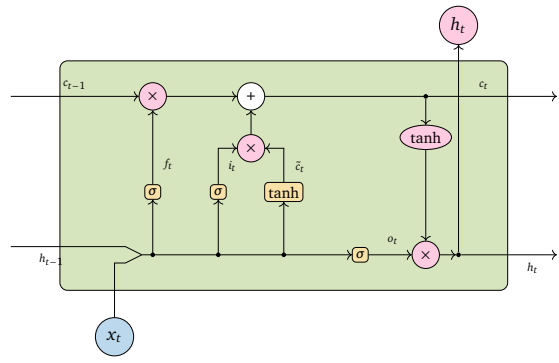
Revisitando RNNs

- Em uma RNN *vanilla* o estado oculto h_{t-1} é combinado com a entrada x_t por uma camada densa com ativação tanh.
- Repare que dentro da célula recorrente existe "uma rede densa".
- A LSTM mantém essa ideia, porém com mais camadas internas e um caminho separado para a memória de longo prazo.



Visão Geral da Célula LSTM

- A célula carrega **dois** sinais ao longo do tempo: o estado oculto h_t (*hidden state*) e a memória de longo prazo c_t (*cell state*).
- No diagrama, a linha **superior** conduz c_t e a **inferior** conduz h_t.
- *Gates* (camadas densas com *sigmoid*, saída em $[0, 1]$) controlam o quanto de cada informação é mantido, atualizado ou exposto, multiplicando vetores ponto a ponto.



σ camada dense

$\times$ operação ponto a ponto

transferência de vetor

concatenação

Quatro Camadas Densas Dentro da Célula

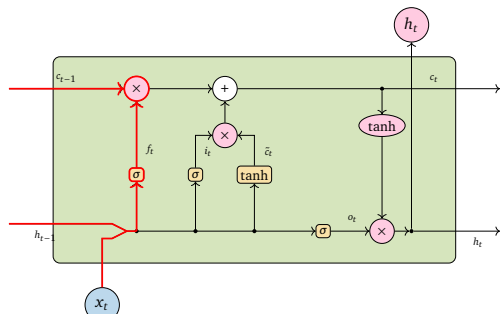
- $[h_{t-1}, x_t]$ é a **concatenação** dos dois vetores num único vetor de dimensão $h_{\text{size}} + x_{\text{size}}$.
- Dentro da célula LSTM existem **quatro** camadas densas que recebem $[h_{t-1}, x_t]$:
  - três delas funcionam como *gates*, com ativação *sigmoid*:
    - *forget gate* (f_t): decide o quanto da memória c_{t-1} deve ser **esquecido**.
    - *input gate* (i_t): decide quais partes da nova informação devem **atualizar** a memória.
    - *output gate* (o_t): decide quais partes da memória devem **compor** o novo estado oculto.
  - a quarta camada usa tanh e produz uma proposta de atualização $\tilde{c}_t$ para a memória.
- As três *gates* aprendem a regular o fluxo de informação ao longo do tempo.

Forget Gate

- Decide quanto da memória c_{t-1} deve ser **esquecida**.
- Recebe $[h_{t-1}, x_t]$, passa por uma camada densa e por uma *sigmoid*:

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

- $f_t \in [0, 1]$, multiplica c_{t-1} ponto a ponto.
  - $f_t \approx 1$: mantém a memória.
  - $f_t \approx 0$: esquece.
- No exemplo do filme, o *forget gate* pode aprender a **manter** o sinal de "péssimo" enquanto a segunda parte da frase é processada.



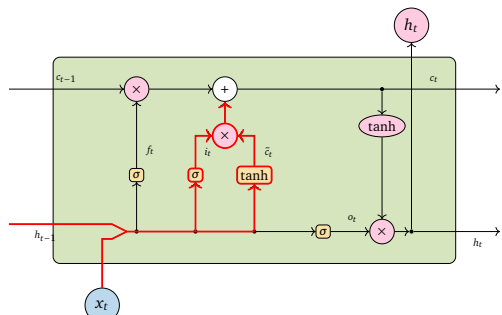
Input Gate e Candidato $\tilde{c}_t$

- Decide quais partes da nova entrada devem **atualizar** a memória.
- Duas camadas trabalham juntas:

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

$$\tilde{c}_t = \tanh\big(\theta_c\,[h_{t-1}, x_t] + b_c\big)$$

- $i_t \in [0, 1]$ filtra a contribuição.
- $\tilde{c}_t \in [-1, 1]$ é a proposta de atualização.

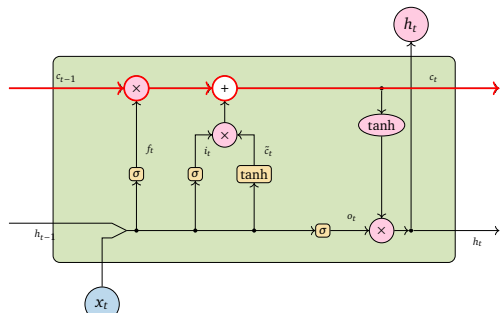


Atualização da Memória c_t

- A nova memória combina o que foi mantido da anterior com a nova proposta filtrada:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

- O símbolo $\odot$ indica multiplicação ponto a ponto (*Hadamard*).
- Repare que c_t tem uma rota direta no tempo, sem ativação não linear no caminho principal.
  - Isso ajuda a mitigar o *vanishing gradient*.



Por que a LSTM Preserva o Gradiente

- Na RNN *vanilla*, o gradiente entre passos distantes acumula um **produto** de fatores que saturam:

$$\frac{\partial h_t}{\partial h_{t-1}} \propto \theta_{hh}\left(1 - \tanh^2(z_t)\right) \qquad \Rightarrow \qquad \prod_k \theta_{hh}\left(1 - \tanh^2(z_k)\right)$$

- Como $1 - \tanh^2 \leq 1$, esse produto tende a **desaparecer** em sequências longas.
- Na LSTM, a memória tem uma rota direta no tempo, $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$. Pelo **caminho direto**:

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t \qquad \Rightarrow \qquad \prod_k f_k$$

- Nesse caminho não há $\tanh'$ repetido nem multiplicação por matrizes, apenas a porta f_k e uma **soma**.
- Quando o *forget gate* aprende $f_k \approx 1$, o gradiente é preservado por muitos passos (*constant error carousel*). As demais rotas, pelas *gates*, contribuem com termos menores.

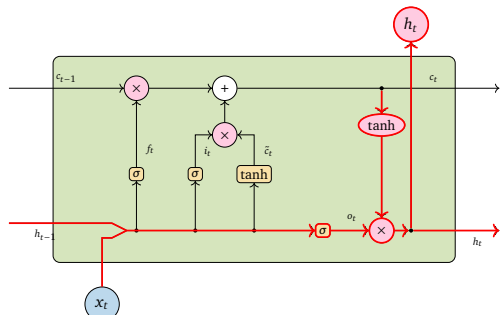
Output Gate e Estado Oculto h_t

- Decide quais partes da memória devem ser **expostas** no estado oculto:

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

$$h_t = o_t \odot \tanh(c_t)$$

- h_t é o que sai da célula a cada passo, e também alimenta a próxima célula.
- c_t continua circulando como memória de longo prazo, em paralelo a h_t.



Resumo das Equações da LSTM

Forget gate	$f_t = \sigma(\theta_f\,[h_{t-1}, x_t] + b_f)$
Input gate	$i_t = \sigma(\theta_i\,[h_{t-1}, x_t] + b_i)$
Candidato $\tilde{c}_t$	$\tilde{c}_t = \tanh(\theta_c\,[h_{t-1}, x_t] + b_c)$
Memória c_t	$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$
Output gate	$o_t = \sigma(\theta_o\,[h_{t-1}, x_t] + b_o)$
Estado oculto h_t	$h_t = o_t \odot \tanh(c_t)$

- Cada peso $\theta_\bullet$ é uma matriz que opera sobre o vetor concatenado $[h_{t-1}, x_t]$.
- Os *bias* $b_\bullet$ são vetores com a mesma dimensão da saída da *gate*.
- Sendo h_{size} a dimensão do estado oculto e x_{size} a dimensão da entrada, os *shapes* esperados são:

$$\theta_\bullet \in \mathbb{R}^{h_{\text{size}} \times (h_{\text{size}} + x_{\text{size}})}, \qquad b_\bullet \in \mathbb{R}^{h_{\text{size}}}.$$

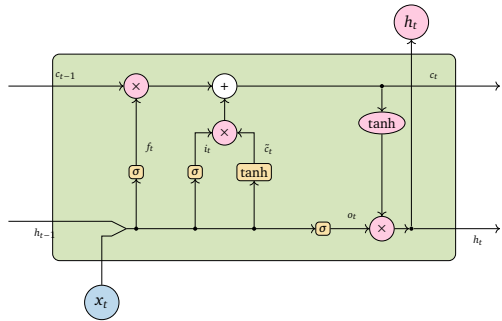
Exemplo Numérico: Configuração

- Vamos rodar um passo da LSTM com valores concretos.
- Estado inicial e entrada (escalares para simplificar):

$$c_0 = 0.5 \qquad h_0 = -0.6 \qquad x_0 = 0.8$$

- Pesos das quatro camadas densas, cada $\theta_\bullet = [\theta_h, \theta_x]$ multiplica $[h_{t-1}, x_t]^T$:

$$\theta_f = [0.5, -0.5], \quad b_f = 0.3$$
$$\theta_i = [0.3, -0.4], \quad b_i = 0.5$$
$$\theta_c = [0.1, -0.3], \quad b_c = 0.7$$
$$\theta_o = [-0.5, 0.4], \quad b_o = 0.5$$

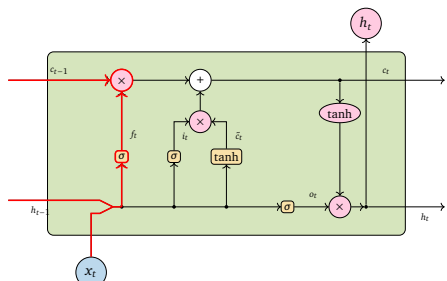


Exemplo: *Forget Gate*

- Usando $[h_0, x_0]^T = [-0.6,\ 0.8]^T$:

$$
\begin{aligned}
f_t &= \sigma\big([0.5,\ -0.5] \cdot [-0.6,\ 0.8]^T + 0.3\big) \\
&= \sigma(0.5 \cdot (-0.6) + (-0.5) \cdot 0.8 + 0.3) \\
&= \sigma(-0.4) \approx 0.4013
\end{aligned}
$$

- Esse valor multiplicará $c_0 = 0.5$, mantendo cerca de 40% da memória anterior.



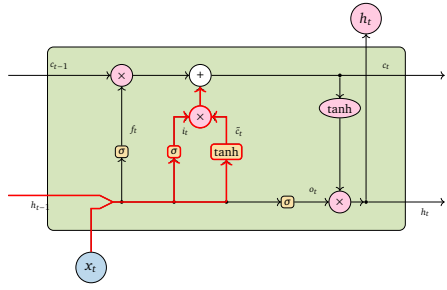
Exemplo: *Input Gate* e Candidato

- *Input gate*:

$$\begin{aligned} i_t &= \sigma([0.3,\ -0.4] \cdot [-0.6,\ 0.8]^T + 0.5) \\ &= \sigma(0) \approx 0.5000 \end{aligned}$$

- Candidato $\tilde{c}_t$:

$$\begin{aligned} \tilde{c}_t &= \tanh([0.1,\ -0.3] \cdot [-0.6,\ 0.8]^T + 0.7) \\ &= \tanh(0.4) \approx 0.3799 \end{aligned}$$

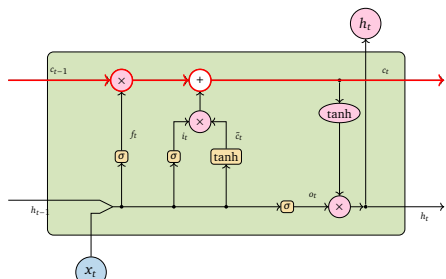


- i_t filtra cerca da metade do candidato, e $\tilde{c}_t$ é a contribuição em si.

Exemplo: Atualização da Memória

$$
\begin{aligned}
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \\
&= 0.4013 \cdot 0.5 + 0.5000 \cdot 0.3799 \\
&\approx 0.2007 + 0.1900 \approx 0.3907
\end{aligned}
$$

- A nova memória c_t combina parte da memória antiga (filtrada por f_t) com parte da nova proposta (filtrada por i_t).
- No caminho de c_t não há tanh nem multiplicação por matrizes, apenas operações ponto a ponto.



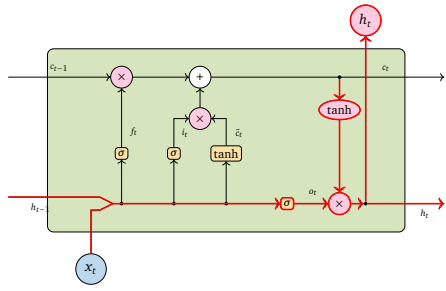
Exemplo: *Output Gate* e h_t

- *Output gate*:

$$o_t = \sigma([-0.5,\ 0.4] \cdot [-0.6,\ 0.8]^T + 0.5)$$
$$= \sigma(1.12) \approx 0.7540$$

- Estado oculto:

$$h_t = o_t \cdot \tanh(c_t)$$
$$= 0.7540 \cdot \tanh(0.3907)$$
$$\approx 0.7540 \cdot 0.3719 \approx 0.2804$$



- O sinal de h_t inverteu em relação a $h_0 = -0.6$, refletindo a entrada positiva $x_0 = 0.8$ filtrada pelas *gates*.

Forward Pass Otimizado

- As quatro camadas densas internas operam sobre o mesmo vetor $[h_{t-1}, x_t]$.
- Em vez de quatro multiplicações de matrizes, agrupamos os pesos das quatro camadas $(\theta_i, \theta_f, \theta_c, \theta_o)$ e separamos o resultado depois:
  - is: dimensão da entrada x_t, hs: dimensão do estado oculto h_t.
  - Cada $\theta_\bullet\,[h_{t-1}, x_t]$ se separa em $x_t\,W_\bullet + h_{t-1}\,U_\bullet$: W reúne as colunas que multiplicam x_t e U as que multiplicam h_{t-1}.

$$W \in \mathbb{R}^{is \times 4hs}, \quad U \in \mathbb{R}^{hs \times 4hs}, \quad b \in \mathbb{R}^{4hs}$$

$$\begin{aligned}
A_t &= x_t W + h_{t-1} U + b \\
i_t &= \sigma(A_t[:, 0 : hs]) \\
f_t &= \sigma(A_t[:, hs : 2hs]) \\
\tilde{c}_t &= \tanh(A_t[:, 2hs : 3hs]) \\
o_t &= \sigma(A_t[:, 3hs : 4hs])
\end{aligned}$$

- Essa é a convenção do PyTorch (`nn.LSTM`): `weight_ih` (W) e `weight_hh` (U) concatenam as *gates* na ordem i, f, g, o (o candidato $\tilde{c}_t$ é o g_t), com dois *bias* $b_{ih} + b_{hh}$. **Atenção**: essa ordem coloca a *input gate* antes da *forget gate*, invertendo a ordem das equações anteriores.

Implementação em PyTorch (1/2)

```
1  class MyLSTM(nn.Module):
2      def __init__(self, input_sz, hidden_sz):
3          super().__init__()
4          self.input_sz   = input_sz
5          self.hidden_sz  = hidden_sz
6          self.W = nn.Parameter(torch.Tensor(input_sz,  hidden_sz * 4))
7          self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
8          self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
9          self.init_weights()  # nao mostrado
```

- As matrizes W e U contêm os pesos das quatro camadas internas concatenadas.
- Uma única chamada matricial substitui as quatro multiplicações originais.

Implementação em PyTorch (2/2)

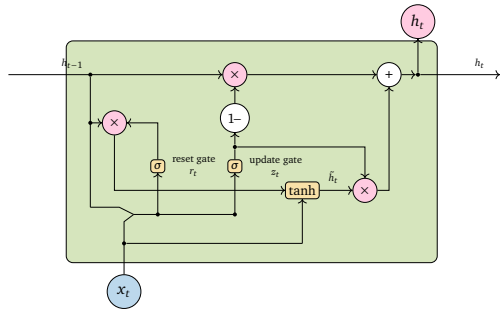
```
1  def forward(self, x):
2      bs, seq_sz, _ = x.size()
3      HS = self.hidden_sz
4      h_t = torch.zeros(bs, HS, device=x.device)
5      c_t = torch.zeros(bs, HS, device=x.device)
6      hidden_seq = []
7      for t in range(seq_sz):
8          x_t   = x[:, t, :]
9          gates = x_t @ self.W + h_t @ self.U + self.bias
10         i_t = torch.sigmoid(gates[:,    :HS])
11         f_t = torch.sigmoid(gates[:, HS:HS*2])
12         g_t = torch.tanh(   gates[:, HS*2:HS*3])
13         o_t = torch.sigmoid(gates[:, HS*3:    ])
14         c_t = f_t * c_t + i_t * g_t
15         h_t = o_t * torch.tanh(c_t)
16         hidden_seq.append(h_t.unsqueeze(0))
17     hidden_seq = torch.cat(hidden_seq, dim=0).transpose(0, 1).contiguous()
18     return hidden_seq, (h_t, c_t)
```

Visão Geral

1. Problemas em Redes Recorrentes
2. LSTMs
3. **GRUs**

Gated Recurrent Unit

- Introduzida por Cho et al. em 2014[a].
- É considerada uma versão **simplificada** da LSTM.
- Não mantém um estado de memória c_t separado do estado oculto h_t, ambos viram um único vetor.
- Em vez das três *gates* da LSTM, a GRU tem apenas duas:
  - *reset gate* r_t,
  - *update gate* z_t.
- Menos parâmetros, treinamento mais rápido, com desempenho competitivo em muitas tarefas.



[a] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". Em: arXiv preprint arXiv:1406.1078 (2014).

Equações da GRU

- *Reset gate*: decide quanto do estado anterior é usado para computar o candidato.

$$r_t = \sigma\big(\theta_r \, [h_{t-1}, x_t] + b_r\big)$$

- *Update gate*: decide a mistura entre o estado anterior e o candidato.

$$z_t = \sigma\big(\theta_z \, [h_{t-1}, x_t] + b_z\big)$$

- Candidato a estado oculto:

$$\tilde{h}_t = \tanh\big(\theta_h \, [r_t \odot h_{t-1}, x_t] + b_h\big)$$

- Novo estado oculto:

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

Exemplo de GRU: *Reset* e *Update Gates*

- Reusando $h_0 = -0.6$ e $x_0 = 0.8$, logo $[h_0, x_0] = [-0.6,\ 0.8]$, com pesos:

$\theta_r = [0.5,\ -0.4],\ b_r = 0.2 \qquad \theta_z = [-0.3,\ 0.6],\ b_z = 0.1 \qquad \theta_h = [0.4,\ 0.5],\ b_h = 0$

- *Reset gate*:

$$r_t = \sigma\big(0.5 \cdot (-0.6) + (-0.4) \cdot 0.8 + 0.2\big) = \sigma(-0.42) \approx 0.3965$$

- *Update gate*:

$$z_t = \sigma\big((-0.3) \cdot (-0.6) + 0.6 \cdot 0.8 + 0.1\big) = \sigma(0.76) \approx 0.6814$$

Exemplo de GRU: Candidato e Novo Estado

- O *reset gate* filtra o estado anterior **antes** de entrar no candidato:

$$r_t \odot h_0 = 0.3965 \cdot (-0.6) \approx -0.2379$$

- Candidato a estado oculto, usando $[r_t \odot h_0,\ x_0] = [-0.2379,\ 0.8]$:

$$\tilde{h}_t = \tanh\big(0.4 \cdot (-0.2379) + 0.5 \cdot 0.8 + 0\big) = \tanh(0.3048) \approx 0.2957$$

- Novo estado oculto, mistura controlada por z_t:

$$h_t = (1 - z_t)\, h_0 + z_t\, \tilde{h}_t = 0.3186 \cdot (-0.6) + 0.6814 \cdot 0.2957 \approx 0.0103$$

- Diferente da LSTM, não há memória c_t separada: a **mesma** porta z_t decide quanto manter de h_0 e quanto usar do candidato.

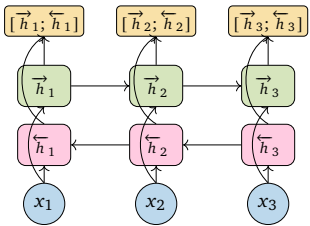
LSTM vs GRU

	LSTM	GRU
Estados que circulam	h_t e c_t	apenas h_t
Quantidade de *gates*	3 (f, i, o)	2 (r, z)
Camadas densas internas	4	3
Parâmetros (relativo)	$4 \cdot d$	$3 \cdot d$

- A linha de parâmetros conta apenas as matrizes densas internas (ignora *bias*). Em prática, uma GRU acaba tendo aproximadamente 75% dos parâmetros de uma LSTM com a mesma dimensão oculta.
- Em sequências longas a LSTM costuma se comportar um pouco melhor por ter o caminho dedicado para c_t.
- A GRU é mais leve e converge mais rápido, sendo uma boa escolha quando há restrição de memória ou de *compute*.

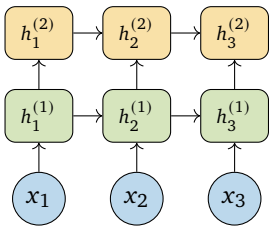
Variantes Comuns: *Bidirectional* e *Stacked*

Bidirectional RNN



- Combina *forward pass* e *backward pass*. Saída é a concatenação dos dois estados ocultos.
- Útil quando toda a sequência está disponível (e.g., *encoders*). Não se aplica a geração autoregressiva.

Stacked RNN



- Várias camadas recorrentes empilhadas. A saída da camada ℓ vira a entrada da camada $\ell + 1$.
- Usado em *encoders* profundos para enriquecer a representação. Cada camada pode ter sua própria LSTM ou GRU.

- Ambas as variantes voltam a aparecer no contexto de *encoder-decoder* no próximo deck.

Leituras Recomendadas

- Capítulo 10 de Ian Goodfellow, Yoshua Bengio e Aaron Courville. Deep Learning. MIT Press, 2016. URL: https://www.deeplearningbook.org/.
- *Understanding LSTM Networks*, Christopher Olah, 2015.
- *MIT Introduction to Deep Learning*, Ava Amini, 2023.

Referências I

[1] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. Em: arXiv preprint arXiv:1406.1078 (2014).

[2] Ian Goodfellow, Yoshua Bengio e Aaron Courville. Deep Learning. MIT Press, 2016. URL: https://www.deeplearningbook.org/.

[3] Sepp Hochreiter e Jürgen Schmidhuber. “Long short-term memory”. Em: Neural Computation 9.8 (1997), pp. 1735–1780.