

Aprendizado Profundo 1

LSTMs e GRUs

Professor: Lucas Silveira Kupssinskü

Agenda

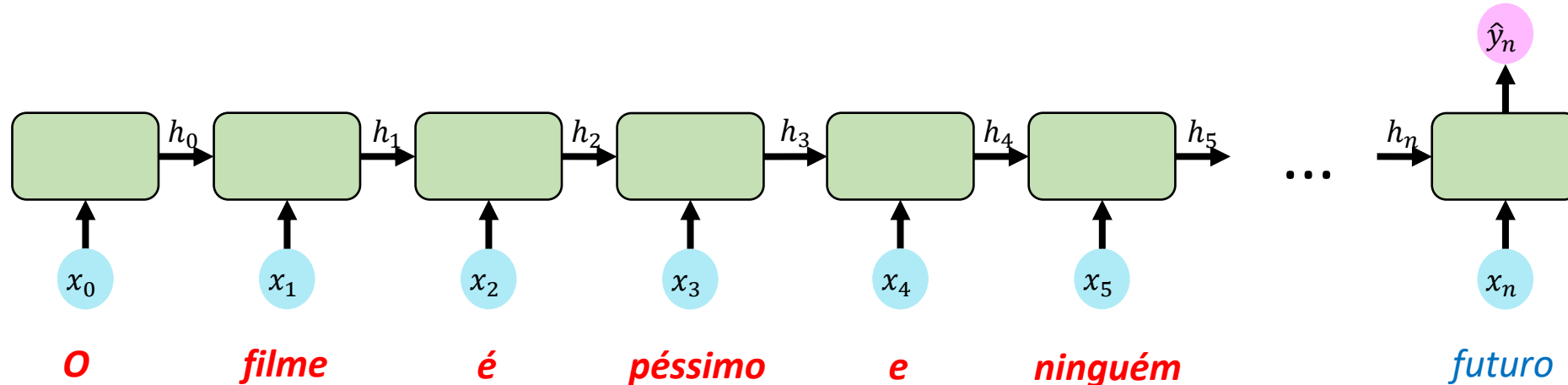
- Problemas em redes recorrentes
- LSTMs
- GRUs

Problemas em redes recorrentes

- *“**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 essas tentativa em outras obras no futuro”*

Problemas em redes recorrentes

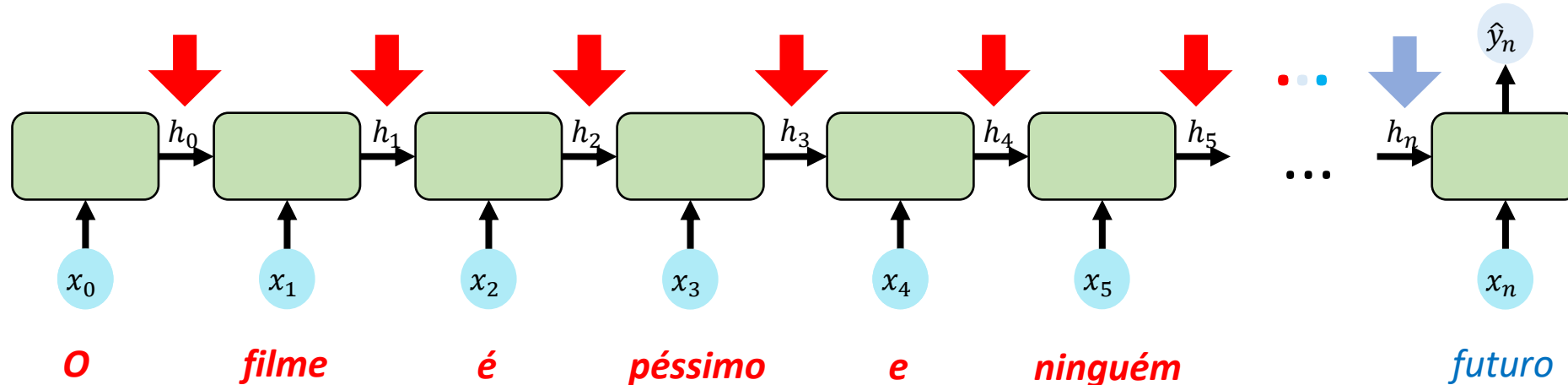
- “**O filme é péssimo e ninguém deveria perder tempo assistindo.** Contudo eu admiro a tentativa de hollywood de aumentar a representatividade em termos de história e atores, certamente será possível colher frutos sobre essas tentativa em outras obras no futuro”



Problemas em redes recorrentes

- “*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 essas tentativa em outras obras no futuro*”

Repare que, em todos os passos da RNN o estado interno passa por atualizações. Isso dificulta a manutenção de informações por longos períodos de tempo.

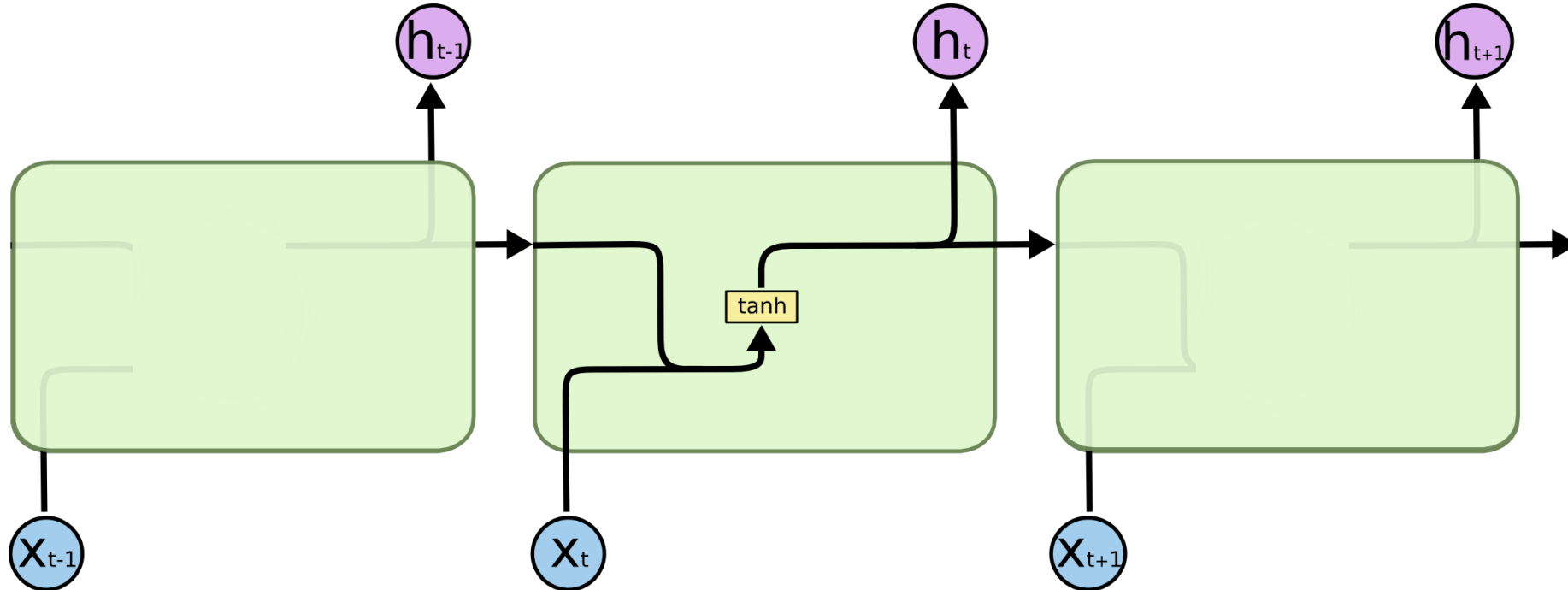


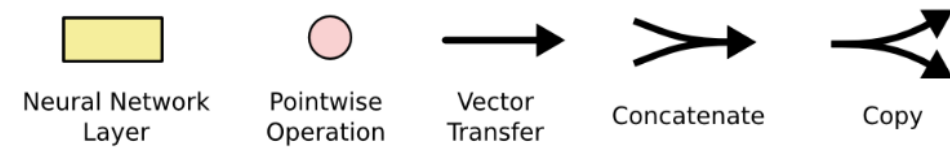
Solução

- Permitir que a unidade aprenda quando deve ser atualizado e quando deve ser esquecido o estado interno
 - *Long Short Term Memory*
- É a arquitetura com mais “partes” dentre todas as vistas até agora
 - Contudo, todas as partes se reduzem a operações já conhecidas: multiplicação de matrizes e funções de ativação

Revisitando RNNs

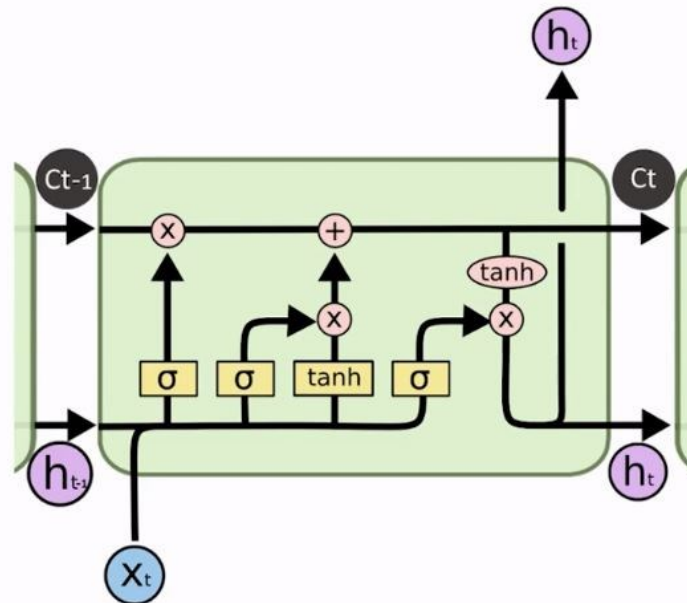
- Em RNN temos que o estado oculto de um passo anterior h_{t-1} é usado como entrada para o computar o próximo estado oculto h_t
- Repare que a RNN tem “uma rede densa” dentro



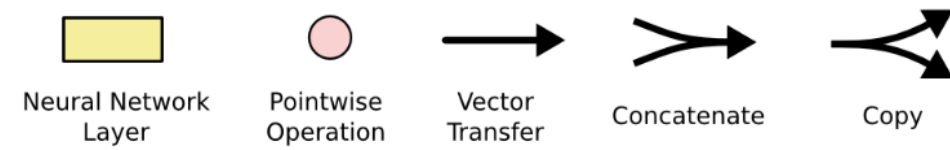


LSTMs

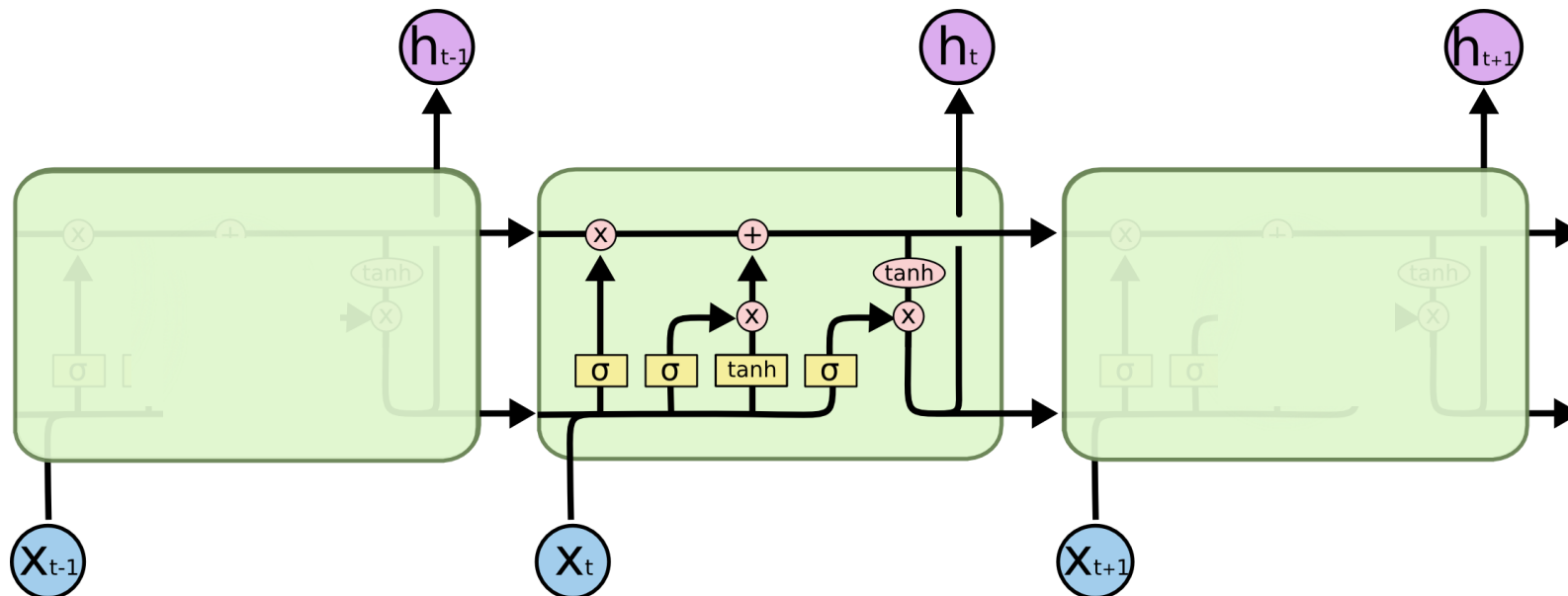
- Mesma ideia da recorrência, mas agora temos um estado oculto H_t e uma memória (*cell state*) C_t
 - Temos *gates* que tem como objetivo controlar o quanto de cada informação deve ser atualizada (usando uma multiplicação com o resultado de uma sigmoid)
 - Temos uma camada densa com tanh para produzir um novo *cell state* e um novo *hidden state*

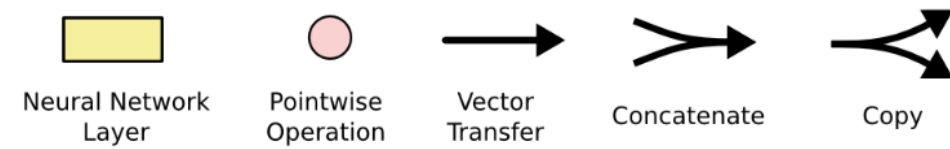


LSTMs



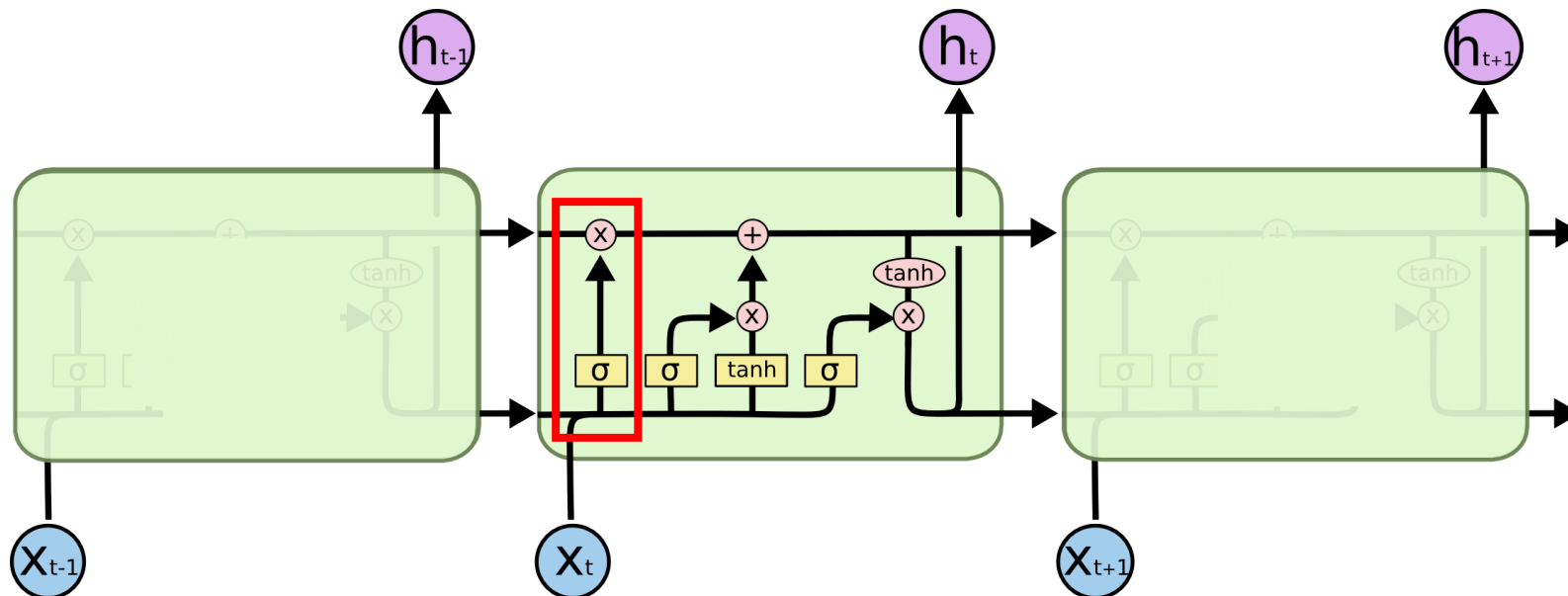
- Mesma ideia da recorrência, mas agora temos quatro camadas densas
 - Três delas fazem função de *gates* usando ativação sigmoid
 - *Forget gate* – esquecer memória (*cell state*)
 - *Input gate* – escolher quais partes da entrada devem atualizar o estado atual
 - *Output gate* – escolher quais partes da saída devem fazer parte do próximo estado oculto

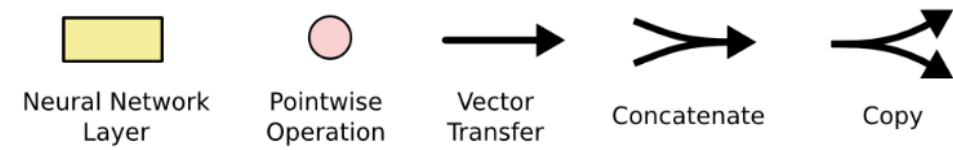




LSTMs

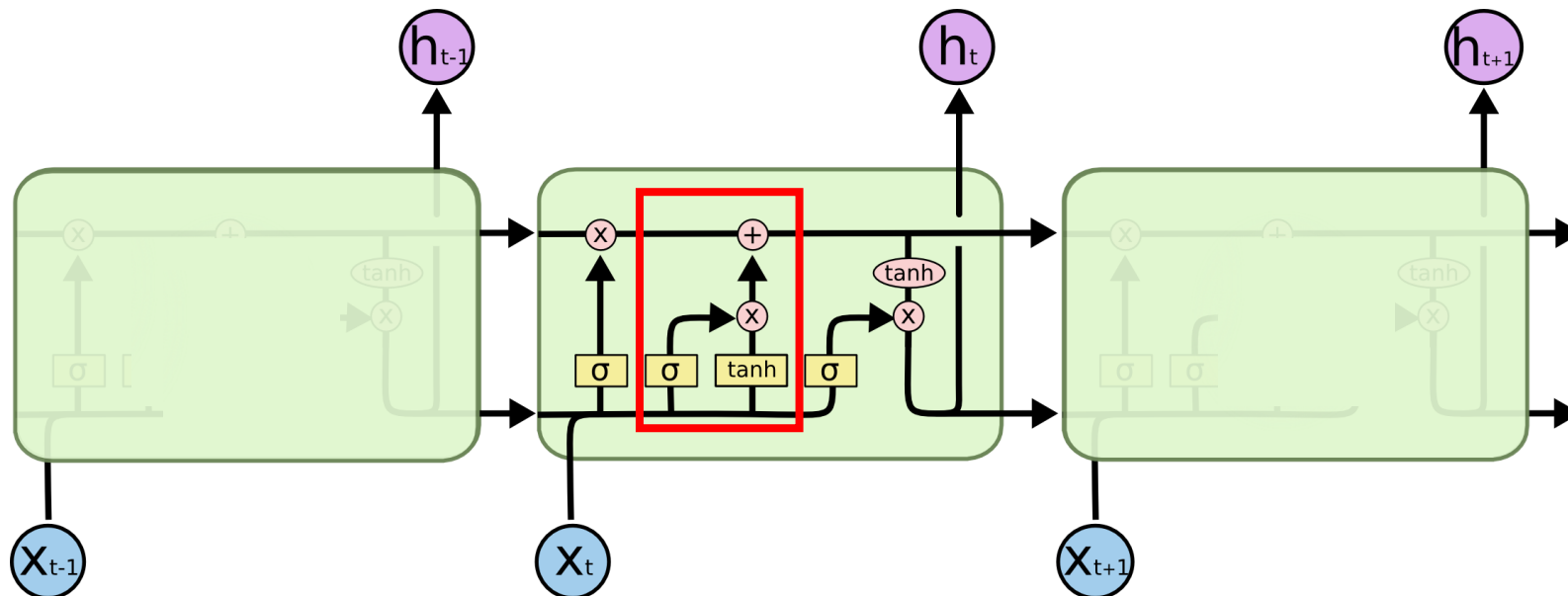
- Mesma ideia da recorrência, mas agora temos quatro camadas densas
 - Três delas fazem função de *gates* usando ativação sigmoid
 - *Forget gate* – esquecer memória (*cell state*)
 - *Input gate* – escolher quais partes da entrada devem atualizar o estado atual
 - *Output gate* – escolher quais partes da saída devem fazer parte do próximo estado oculto

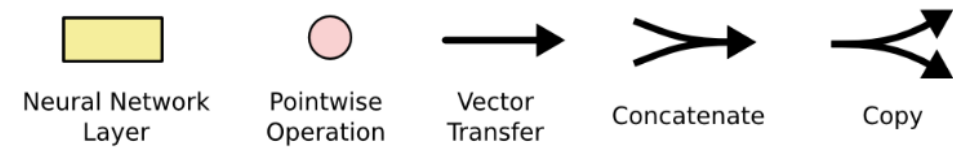




LSTMs

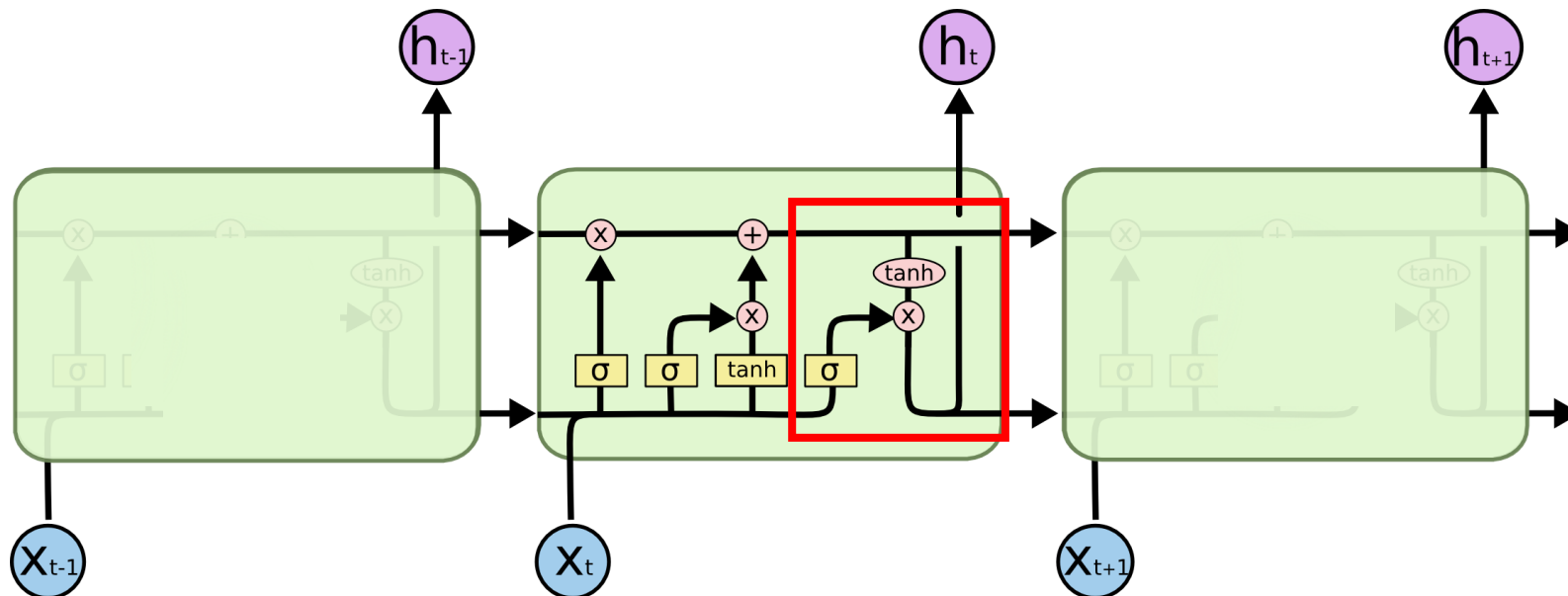
- Mesma ideia da recorrência, mas agora temos quatro camadas densas
 - Três delas fazem função de *gates* usando ativação sigmoid
 - *Forget gate* – esquecer memória (*cell state*)
 - *Input gate* – escolher quais partes da entrada devem atualizar o estado atual
 - *Output gate* – escolher quais partes da saída devem fazer parte do próximo estado oculto



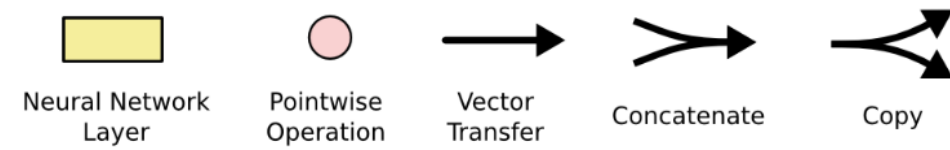


LSTMs

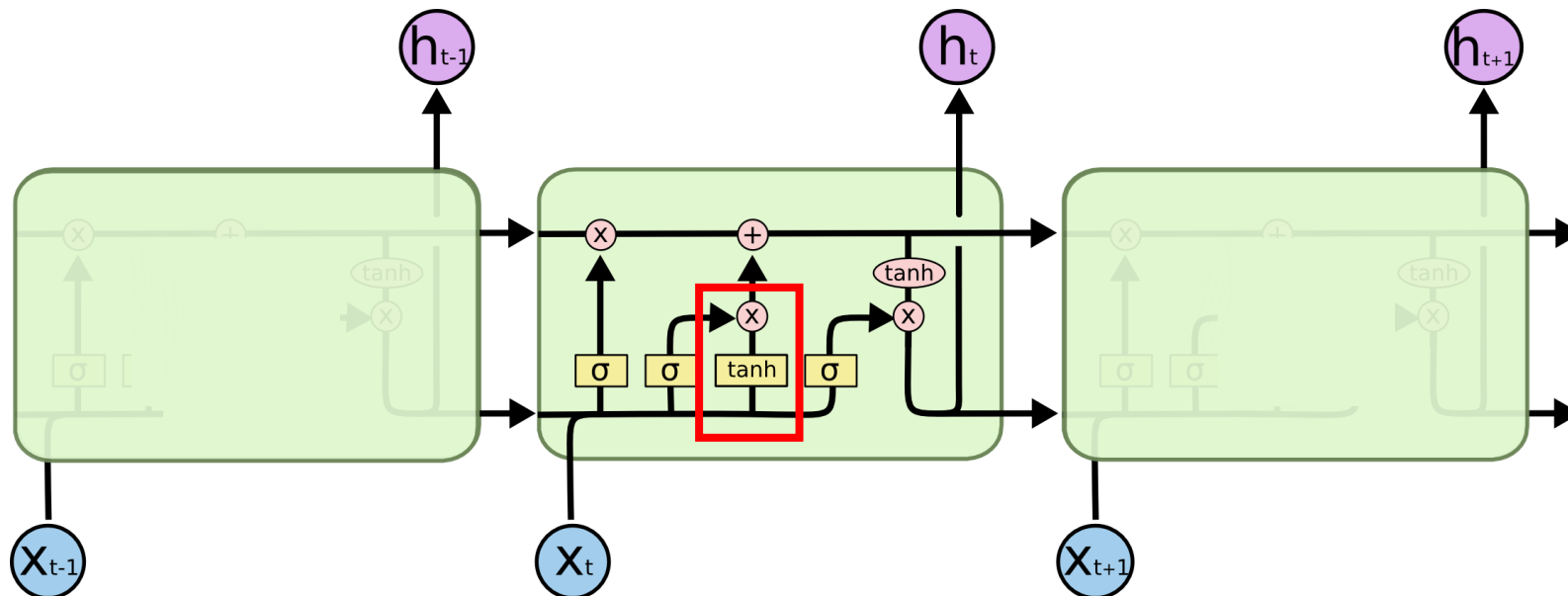
- Mesma ideia da recorrência, mas agora temos quatro camadas densas
 - Três delas fazem função de *gates* usando ativação sigmoid
 - *Forget gate* – esquecer memória (*cell state*)
 - *Input gate* – escolher quais partes da entrada devem atualizar o estado atual
 - *Output gate* – escolher quais partes da saída devem fazer parte do próximo estado oculto



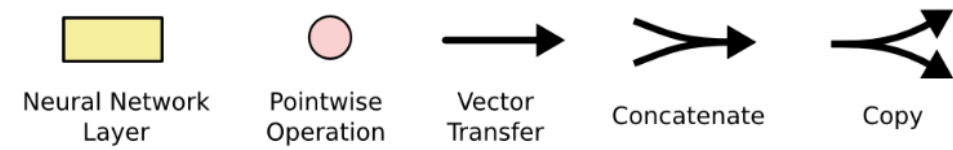
LSTMs



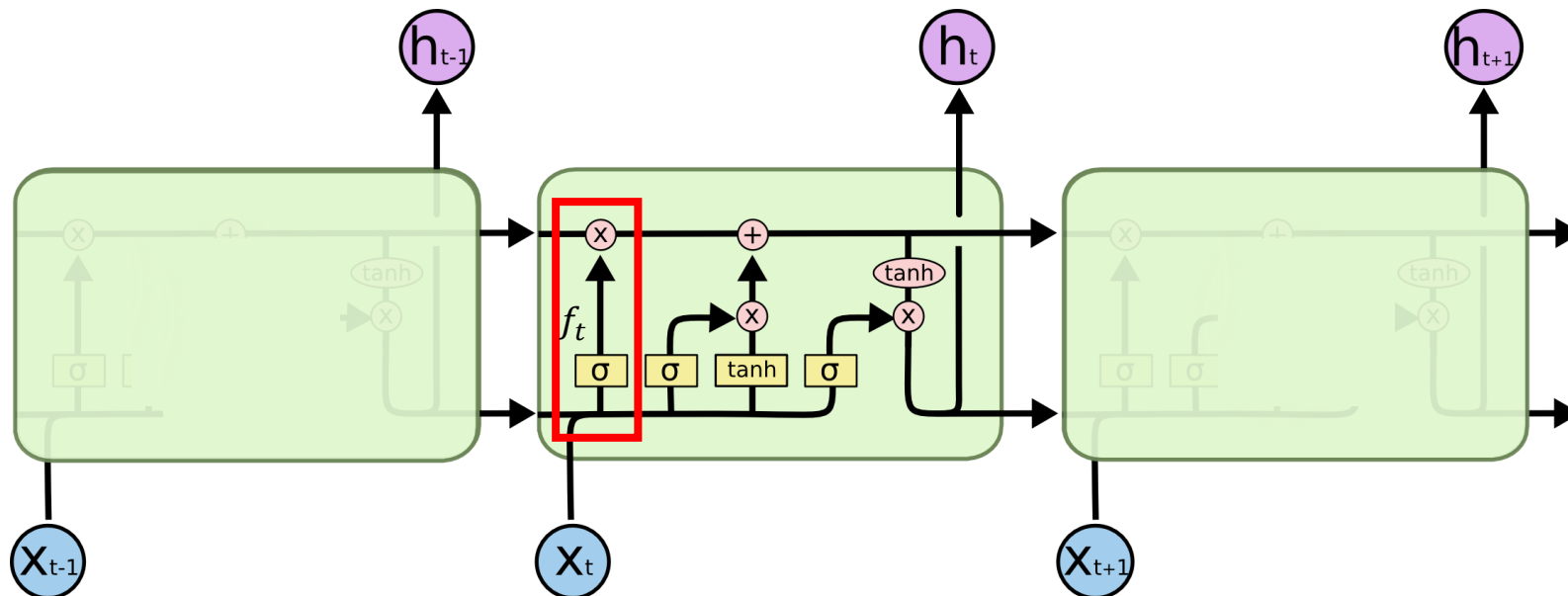
- Mesma ideia da recorrência, mas agora temos quatro camadas densas
 - Uma delas aplica uma transformação não linear na entrada + estado oculto anterior



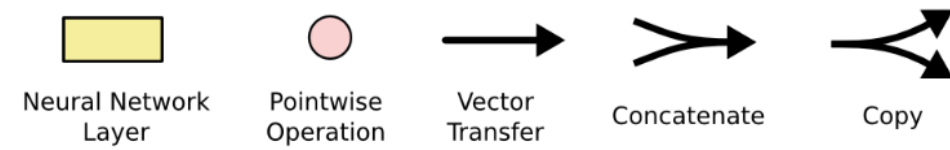
LSTMs



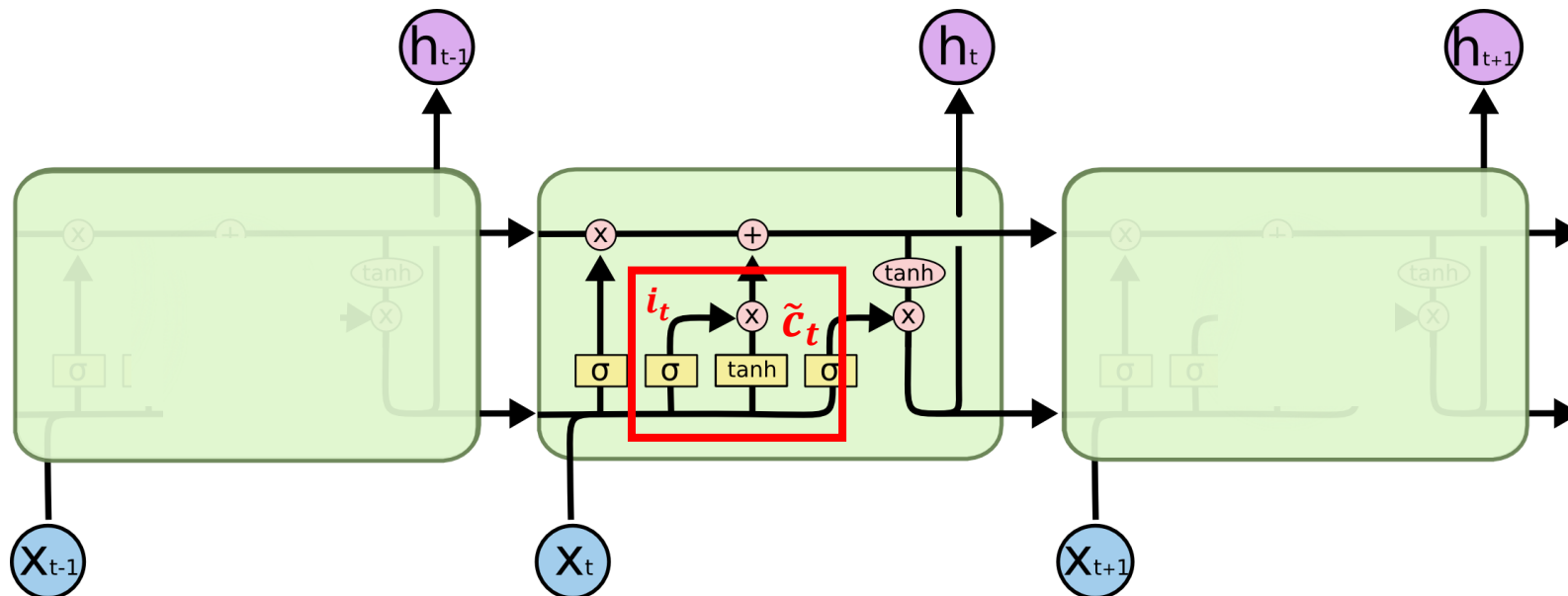
- Vamos passar novamente por cada passo, porém agora vamos detalhar as operações
 - $f_t = \sigma(\theta_f[h_{t-1}, x_t] + b_f)$

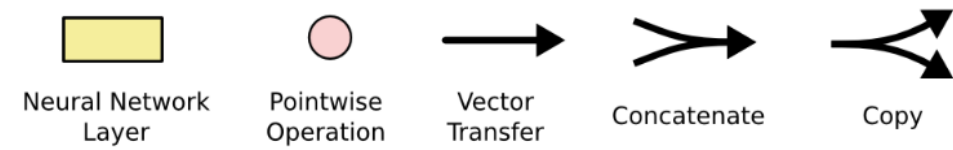


LSTMs



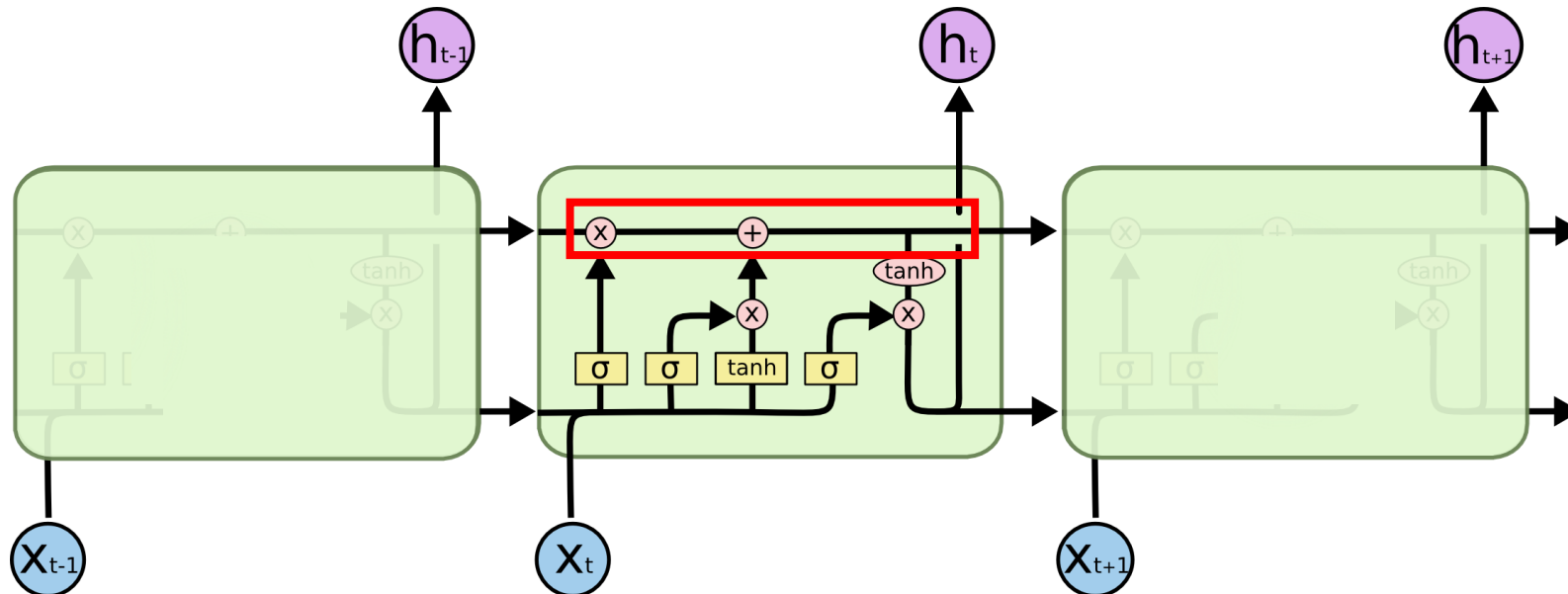
- Vamos passar novamente por cada passo, porém agora vamos detalhar as operações
 - $i_t = \sigma(\theta_i[h_{t-1}, x_t] + b_i)$
 - $\tilde{c}_t = \tanh(\theta_c[h_{t-1}, x_t] + b_c)$



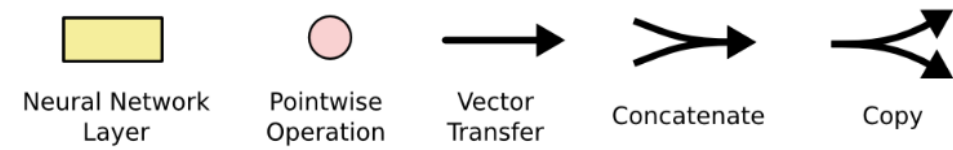


LSTMs

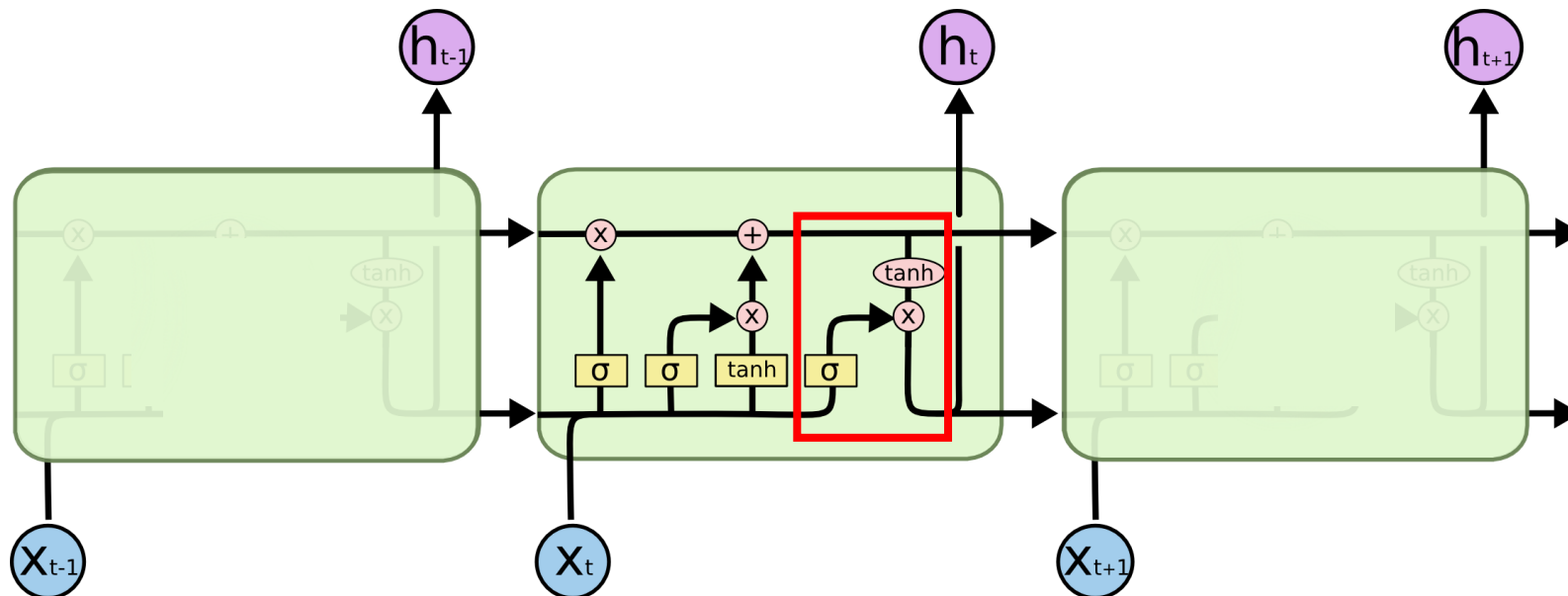
- Vamos passar novamente por cada passo, porém agora vamos detalhar as operações
 - $c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$

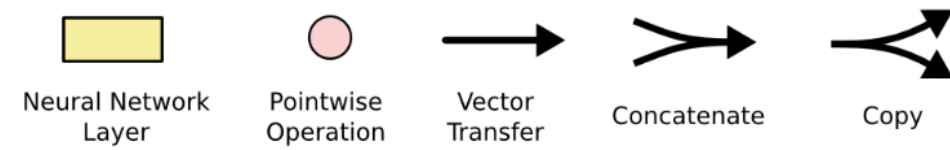


LSTMs




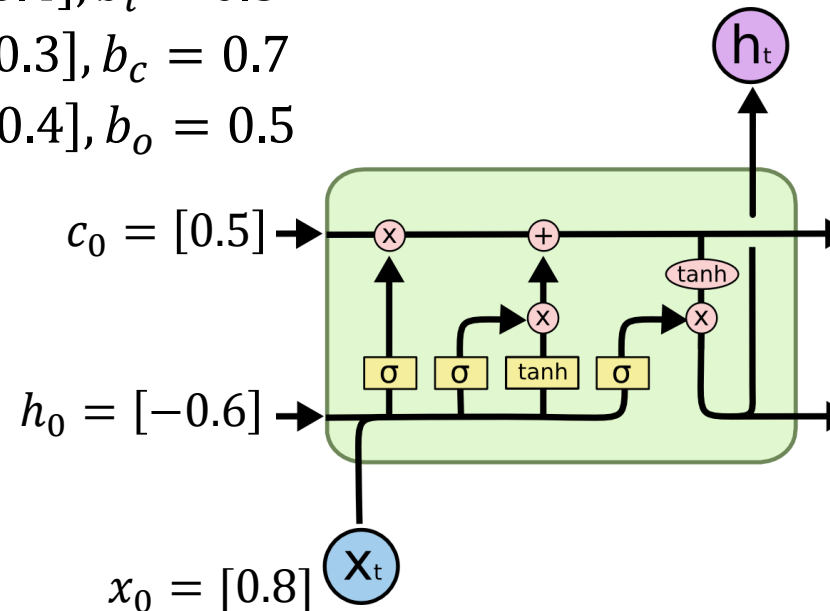
- Vamos passar novamente por cada passo, porém agora vamos detalhar as operações
 - $o_t = \sigma(\theta_o[h_{t-1}, x_t] + b_o)$
 - $h_t = o_t * \tanh(c_t)$





LSTMs

- Agora vamos usar alguns valores para ver as coisas acontecendo
 - $c_0 = [0.5]$
 - $h_0 = [-0.6,]$
 - $x_0 = [0.8]$
 - $\theta_f = [0.5, -0.5], b_f = 0.3$
 - $\theta_i = [0.3, -0.4], b_i = 0.5$
 - $\theta_c = [0.1, -0.3], b_c = 0.7$
 - $\theta_o = [-0.5, 0.4], b_o = 0.5$
- 
- A diagram showing a purple circle containing the text h_t . A black arrow points upwards from below the circle.



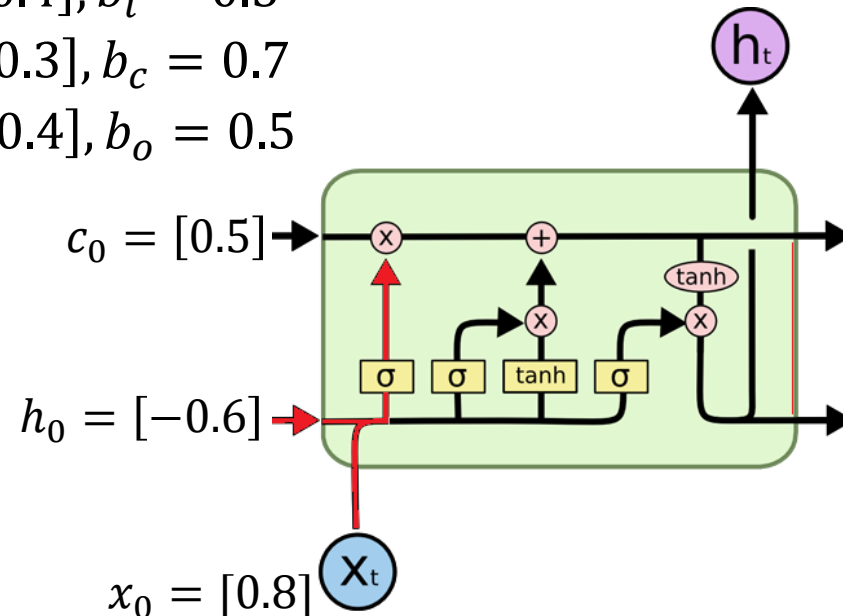
LSTMs

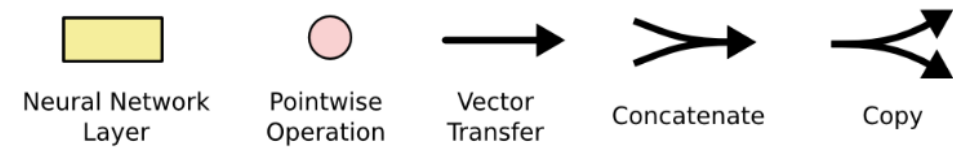
- Agora vamos usar alguns valores para ver as coisas acontecendo

- $c_0 = [0.5]$
- $h_0 = [-0.6,]$
- $x_0 = [0.8]$
- $\theta_f = [0.5, -0.5], b_f = 0.3$
- $\theta_i = [0.3, -0.4], b_i = 0.5$
- $\theta_c = [0.1, -0.3], b_c = 0.7$
- $\theta_o = [-0.5, 0.4], b_o = 0.5$

$$f_t = \sigma([0.5, -0.5][0.8, -0.6]^T + 0.3)$$

$$f_t = \sigma(1) = 0.7311$$





LSTMs

- Agora vamos usar alguns valores para ver as coisas acontecendo

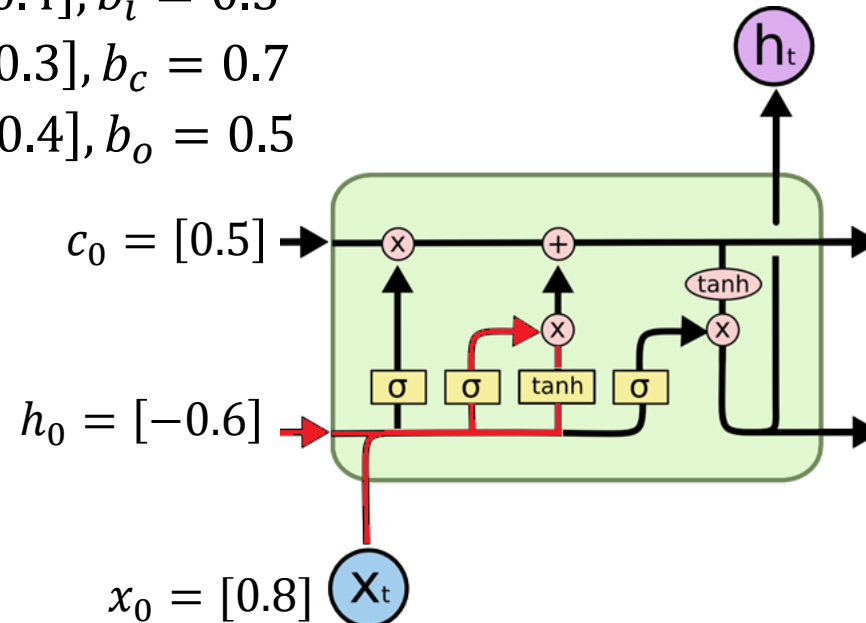
- $c_0 = [0.5]$
- $h_0 = [-0.6]$
- $x_0 = [0.8]$
- $\theta_f = [0.5, -0.5], b_f = 0.3$
- $\theta_i = [0.3, -0.4], b_i = 0.5$
- $\theta_c = [0.1, -0.3], b_c = 0.7$
- $\theta_o = [-0.5, 0.4], b_o = 0.5$

$$i_t = \sigma([0.3, -0.4][0.8, -0.6]^T + 0.5)$$

$$i_t = \sigma(0.98) = 0.7271$$

$$\tilde{c}_t = \tanh([0.1, -0.3][0.8, -0.6]^T + 0.7)$$

$$\tilde{c}_t = \tanh(0.96) = 0.7443$$



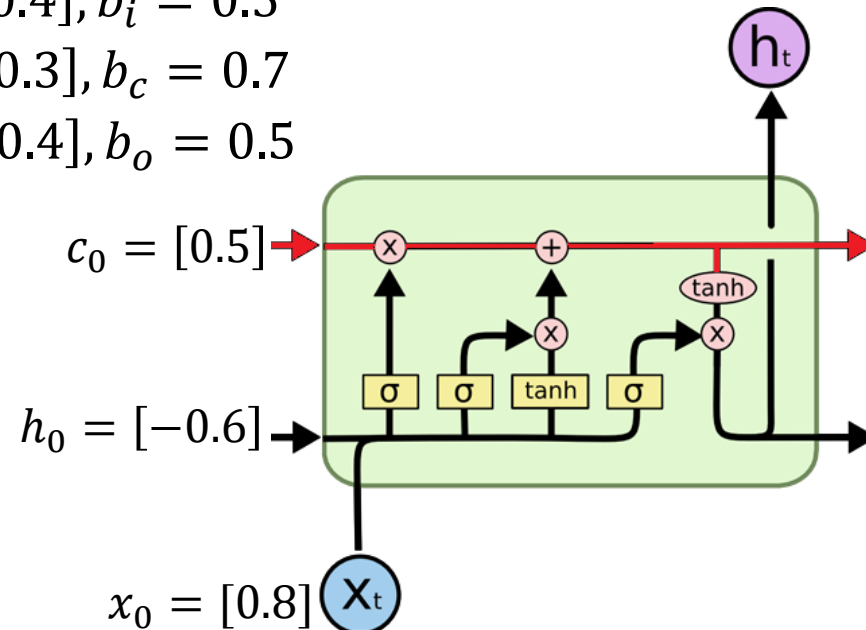
LSTMs

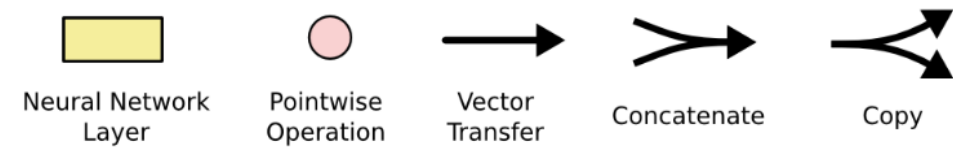
- Agora vamos usar alguns valores para ver as coisas acontecendo

- $c_0 = [0.5]$
- $h_0 = [-0.6,]$
- $x_0 = [0.8]$
- $\theta_f = [0.5, -0.5], b_f = 0.3$
- $\theta_i = [0.3, -0.4], b_i = 0.5$
- $\theta_c = [0.1, -0.3], b_c = 0.7$
- $\theta_o = [-0.5, 0.4], b_o = 0.5$

$$c_t = 0.7311 * 0.5 + 0.7271 * 0.7443$$

$$c_t = 0.9067$$





LSTMs

- Agora vamos usar alguns valores para ver as coisas acontecendo

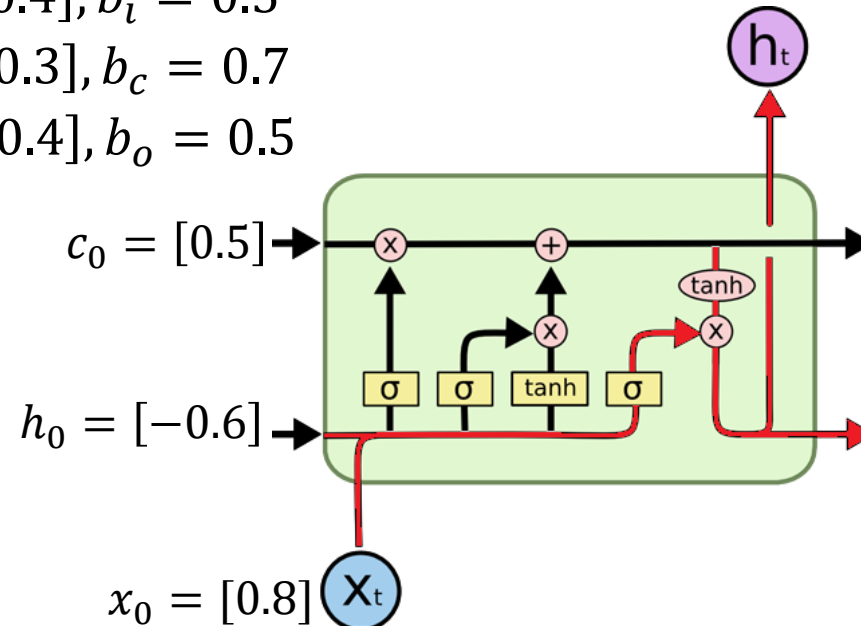
- $c_0 = [0.5]$
- $h_0 = [-0.6,]$
- $x_0 = [0.8]$
- $\theta_f = [0.5, -0.5], b_f = 0.3$
- $\theta_i = [0.3, -0.4], b_i = 0.5$
- $\theta_c = [0.1, -0.3], b_c = 0.7$
- $\theta_o = [-0.5, 0.4], b_o = 0.5$

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

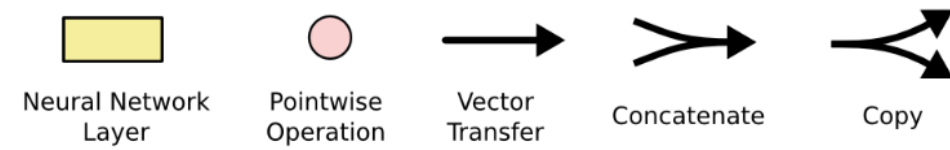
$$o_t = 0.4651$$

$$h_t = 0.4651 * \tanh(0.9067)$$

$$h_t = 0.7195$$



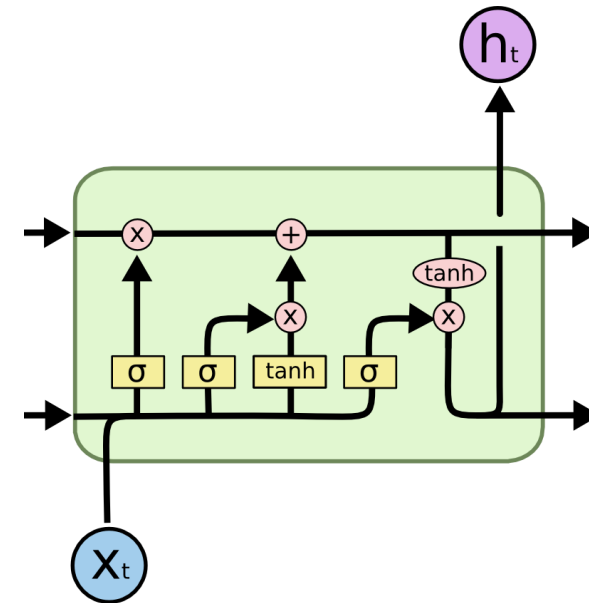
LSTMs



- Podemos otimizar o *Forward Pass* para fazer apenas duas multiplicações de matrizes sem concatenação
- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

- $c_0 = [0.5]$
- $h_0 = [-0.6,]$
- $x_0 = [0.8]$
- $U = [0.5, 0.3, 0.1, -0.5]$
- $V = [-0.5, -0.4, -0.3, 0.4]$
- $b = [0.3, 0.5, 0.7, 0.5]$

$$\begin{aligned}
 A_t &= x_0 W + h_0 U + b \\
 f_t &= \sigma(A_t[:, 0:h_s]) \\
 i_t &= \sigma(A_t[:, h_s:h_s * 2]) \\
 o_t &= \sigma(A_t[:, h_s * 2:h_s * 3]) \\
 \tilde{c}_t &= \tanh(A_t[:, h_s * 3:h_s * 4]) \\
 c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \\
 h_t &= o_t * \tanh(c_t)
 \end{aligned}$$



LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```


LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- is : input size
- hs : hidden size
- $W_{is \times hs \times 4}$
- $U_{hs \times hs \times 4}$
- $b_{hs \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:hs])$$

$$i_t = \sigma(A_t[:, hs:hs * 2])$$

$$o_t = \sigma(A_t[:, hs * 2:hs * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, hs * 3:hs * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

LSTMs

- i_s : input size
- h_s : hidden size
- $W_{i_s \times h_s \times 4}$
- $U_{h_s \times h_s \times 4}$
- $b_{h_s \times 4}$

$$A_t = x_t W + h_{t-1} U + b$$

$$f_t = \sigma(A_t[:, 0:h_s])$$

$$i_t = \sigma(A_t[:, h_s:h_s * 2])$$

$$o_t = \sigma(A_t[:, h_s * 2:h_s * 3])$$

$$\tilde{c}_t = \tanh(A_t[:, h_s * 3:h_s * 4])$$

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

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

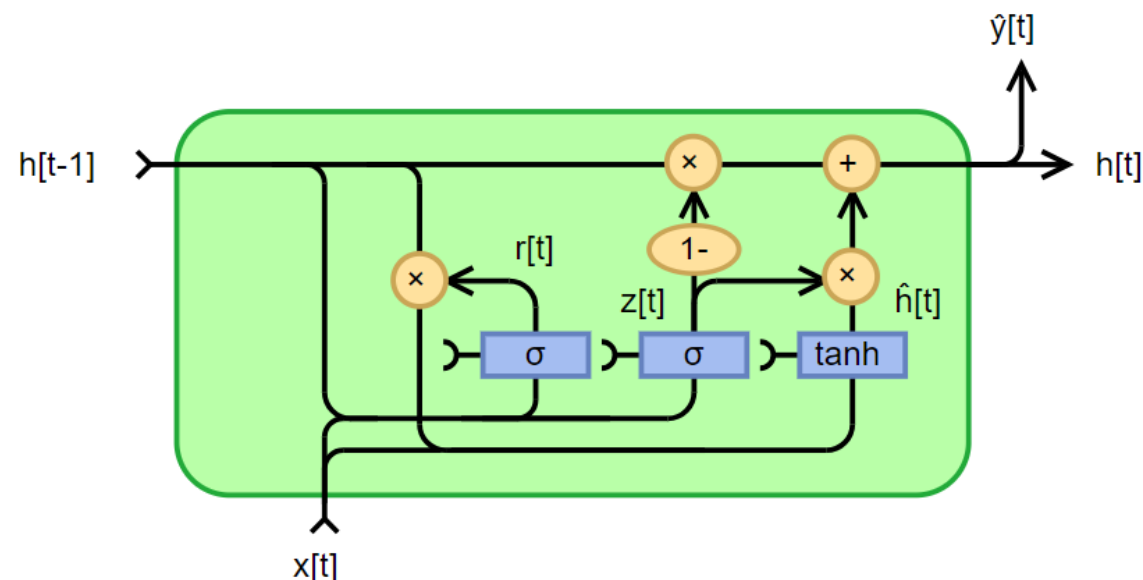
```
class MyLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.W = nn.Parameter(torch.Tensor(input_sz, hidden_sz * 4))
        self.U = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz * 4))
        self.bias = nn.Parameter(torch.Tensor(hidden_sz * 4))
        self.init_weights() # not shown

    def forward(self, x):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                    torch.zeros(bs, self.hidden_size).to(x.device))

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

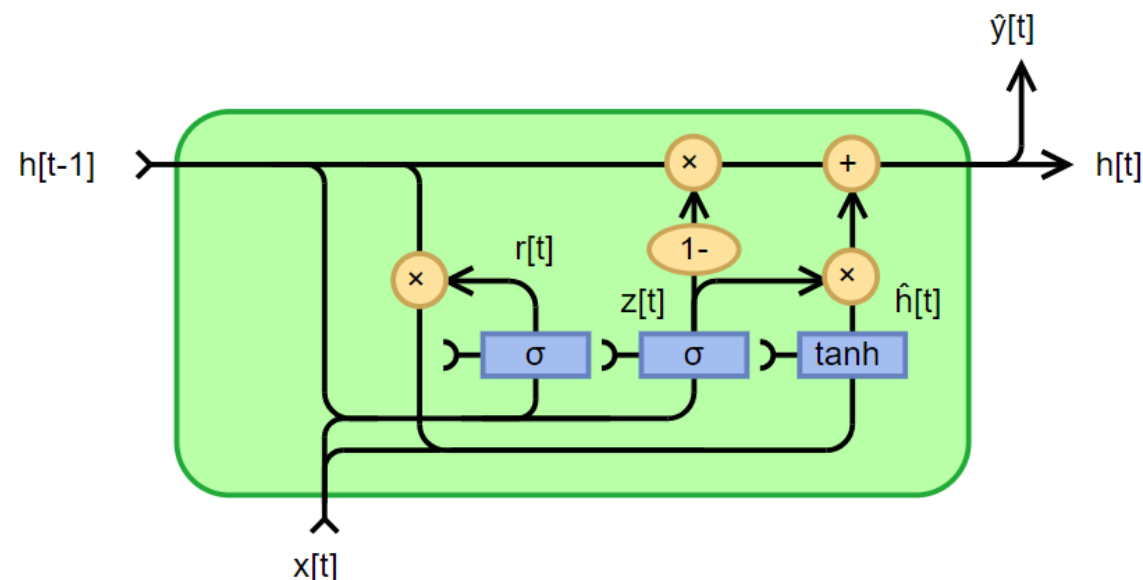

GRU

- *Gated Recurrent Unit* foi introduzida por Cho et al em 2014
- É considerada uma versão simplificada da LSTM
- Não mantemos um c e um h separados
 - Ao invés disso temos uma *reset gate* r e uma *update gate* z



GRU

- *Reset gate*
 - $r_t = \sigma(w_r[x_t, h_{t-1}] + b_r)$
- *Update gate*
 - $z_t = \sigma(w_z[x_t, h_{t-1}] + b_z)$
- *Hidden Candidate*
 - $\tilde{h}_t = \tanh(w_h[x_t, r_t * h_{t-1}] + b_h)$
- $h_t = h_{t-1} * (1 - z_t) + z_t * \tilde{h}_t$



Referências:

- Sugere-se a leitura de:
 - Capítulo 10 - GOODFELLOW, I., BENGIO, Y., COURVILLE, A. Deep Learning. MIT Press, 775p., 2016. (<https://www.deeplearningbook.org/>)
- Material baseado em:
 - MIT Introduction to Deep Learning, Ava Amini. 2023.