# Hands-on Deep Learning - Aula 4

## Redes Neurais Recorrentes

Camila Laranjeira[1], Hugo Oliveira[12], Keiller Nogueira[12]

[1]Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Minas Gerais

[2]Interest Group in Pattern Recognition and Earth Observation (PATREO)
Universidade Federal de Minas Gerais

18 de Agosto, 2018

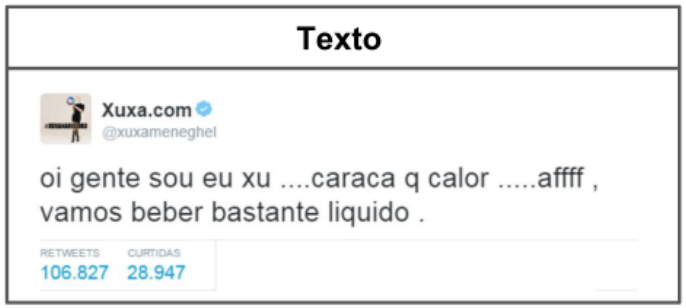
DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

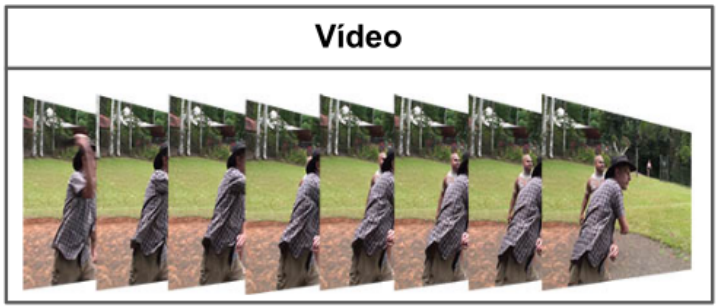
UFMG

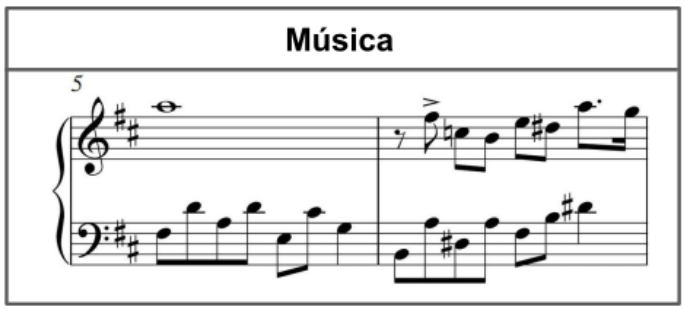
# Agenda

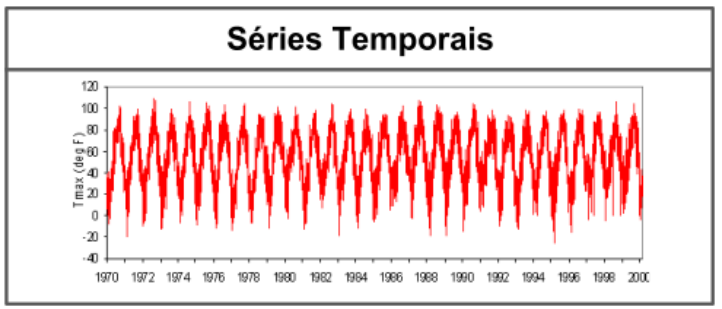
# Agenda

# Sequências




- Eventos relacionados entre si.
- Ordem dos eventos é um fator relevante.

# Memória de sequências


- Experimente listar esse conjunto de eventos fora da ordem (de trás pra frente por exemplo):
  - Alfabeto
  - Letras de música
  - Consegue pensar em outro exemplo?

# Memória de sequências

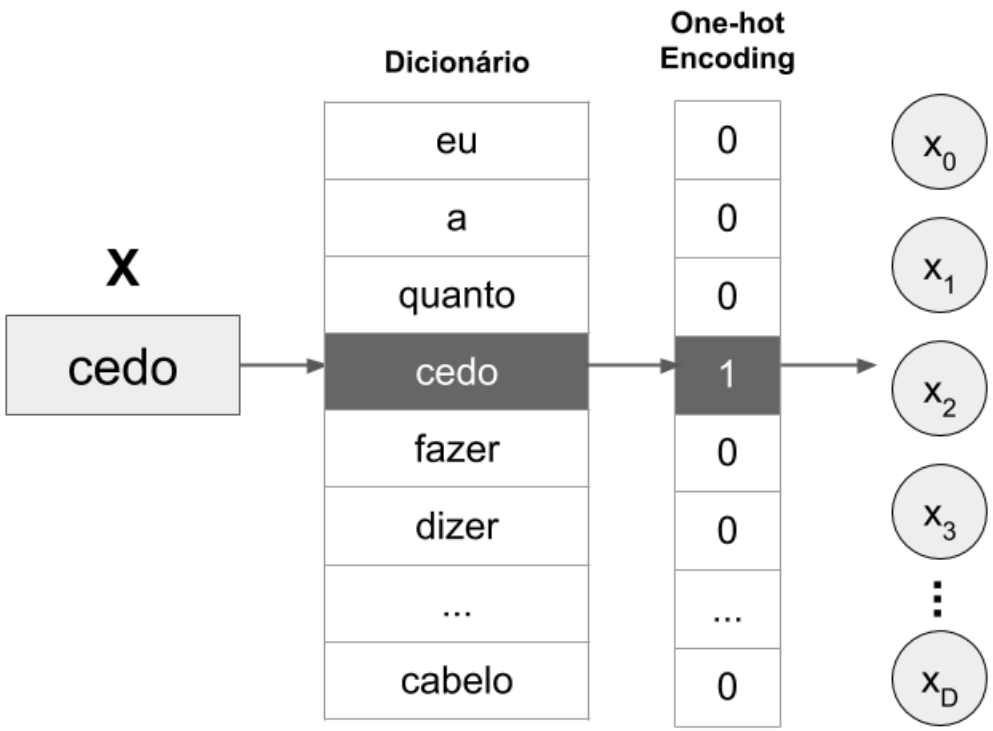

- Experimente listar esse conjunto de eventos fora da ordem (de trás pra frente por exemplo):
  - Alfabeto
  - Letras de música
  - Número de telefone
  - CPF
  - Senhas
  - etc...
- A memória de sequência é condicional. Elementos não são memorizados individualmente, registramos a organização entre eles.

# MLP para Sequências


- Vamos tentar modelar um problema de sequência com uma MLP
- *Sequence tagging* / *Sequence labeling*
  - Dado uma sequência de palavras, rotule cada uma das palavras de acordo com a sua categoria gramatical
  - Exemplo 1:
    $X = \{$ Nós, fizemos, um, **acordo** $\}$,
    $Y = \{$ Pronome, verbo, artigo indefinido, **substantivo** $\}$
  - Exemplo 2:
    $X = \{$ Eu, **acordo**, cedo $\}$,
    $Y = \{$ Pronome, **verbo**, advérbio $\}$

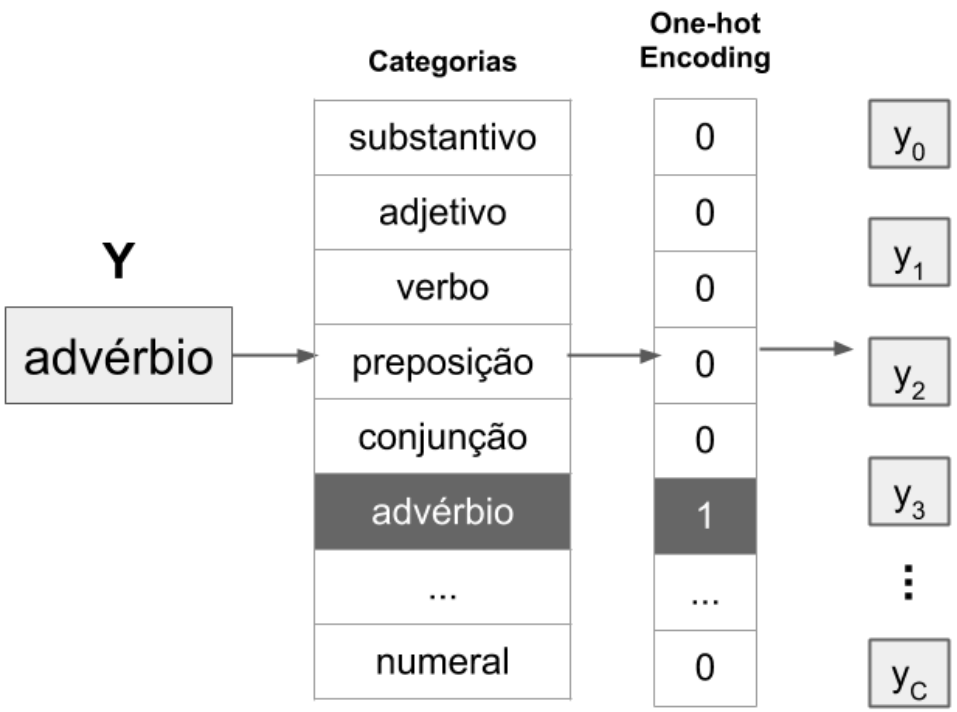
# MLP para sequência

- Modelando a entrada



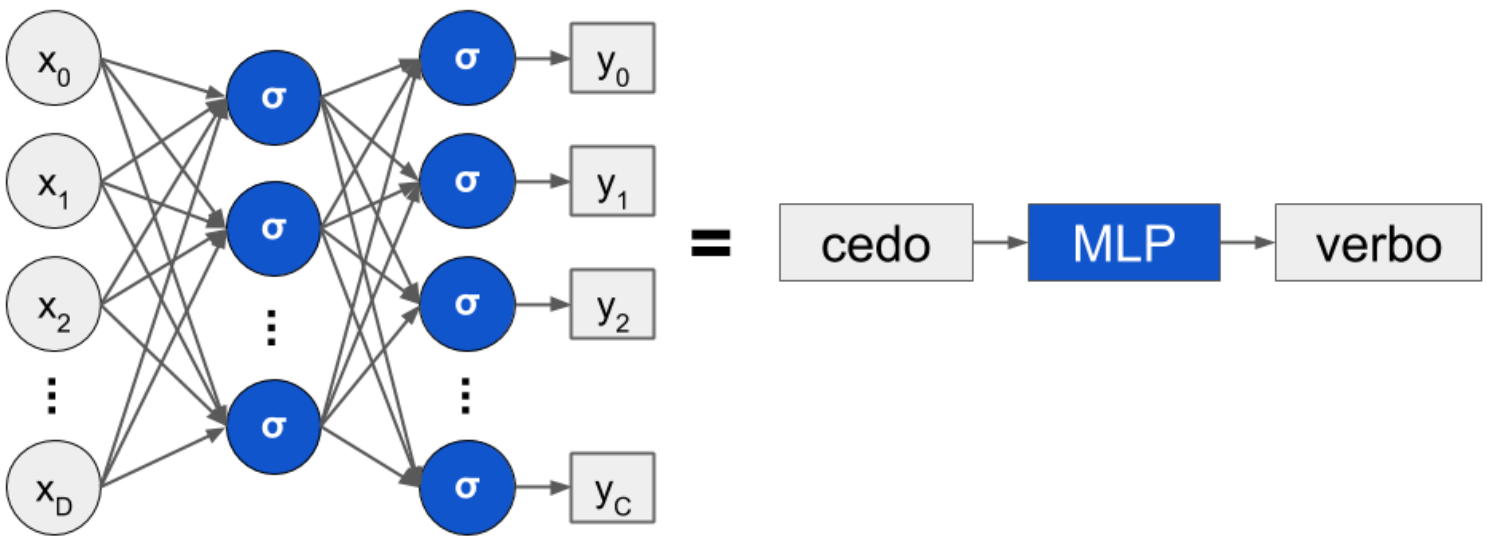
# MLP para sequência

- Modelando a saída



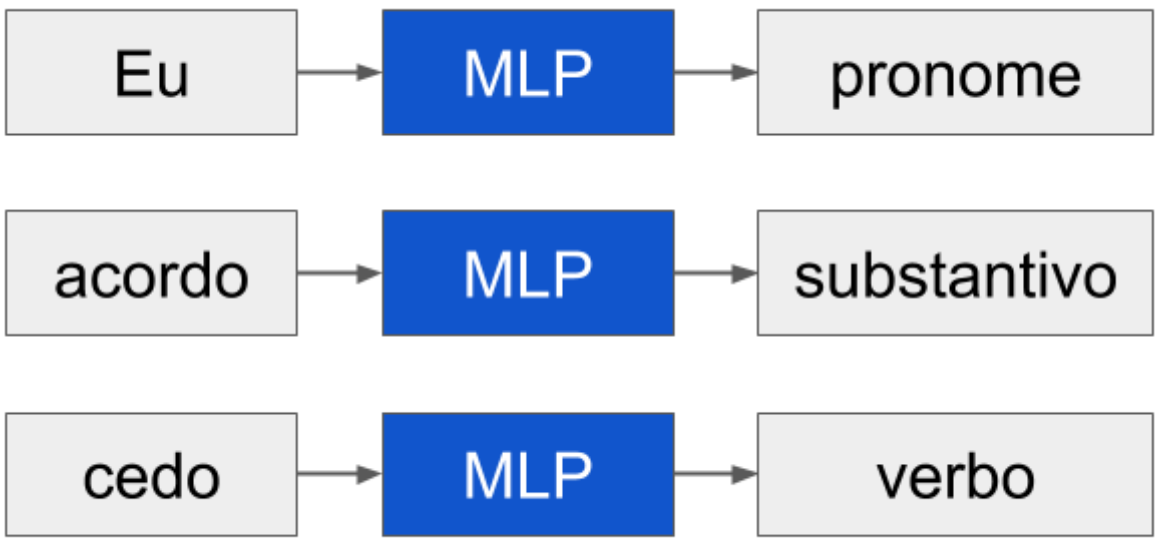
# MLP para sequência

- Modelando o MLP que categoriza palavras gramaticalmente (para uma palavra)



# MLP para sequência


- Modelando o MLP que categoriza palavras gramaticalmente (para uma **sequência** de palavras)



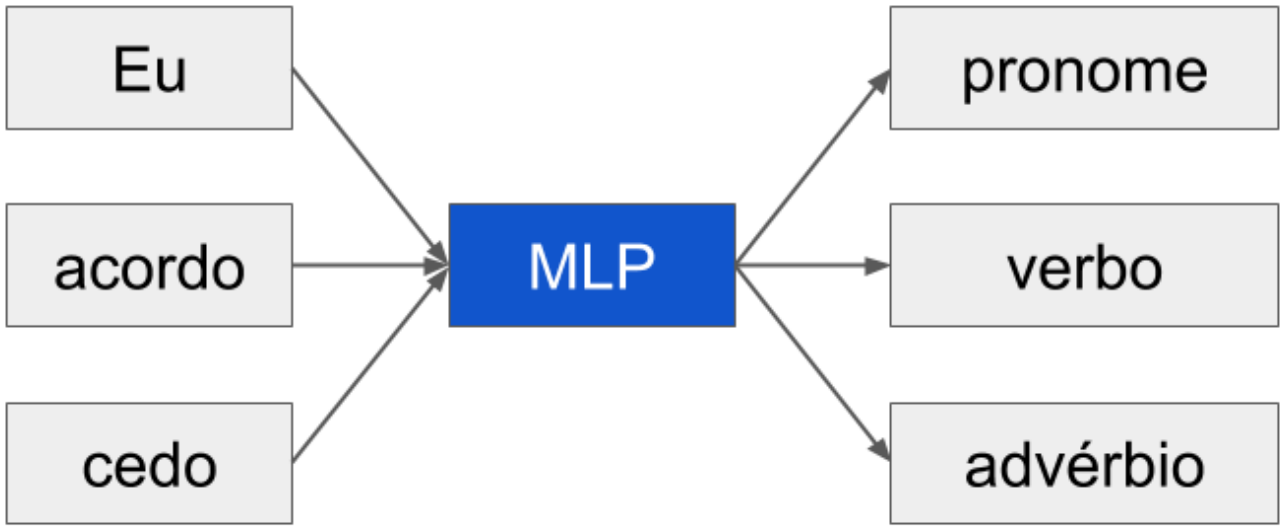
# MLP para Sequências


- Limitações dessa modelagem:
  - O modelo não incorpora a **relação entre palavras**, já que recebe uma entrada de cada vez e atualiza seus pesos de acordo com a informação individual de cada palavra.

# MLP para Sequências


- Vamos tentar mais uma abordagem...

# MLP para Sequências




- Tratamos a sequência de entrada como um único dado
- A sequência de saída também é um único vetor

# MLP para Sequências

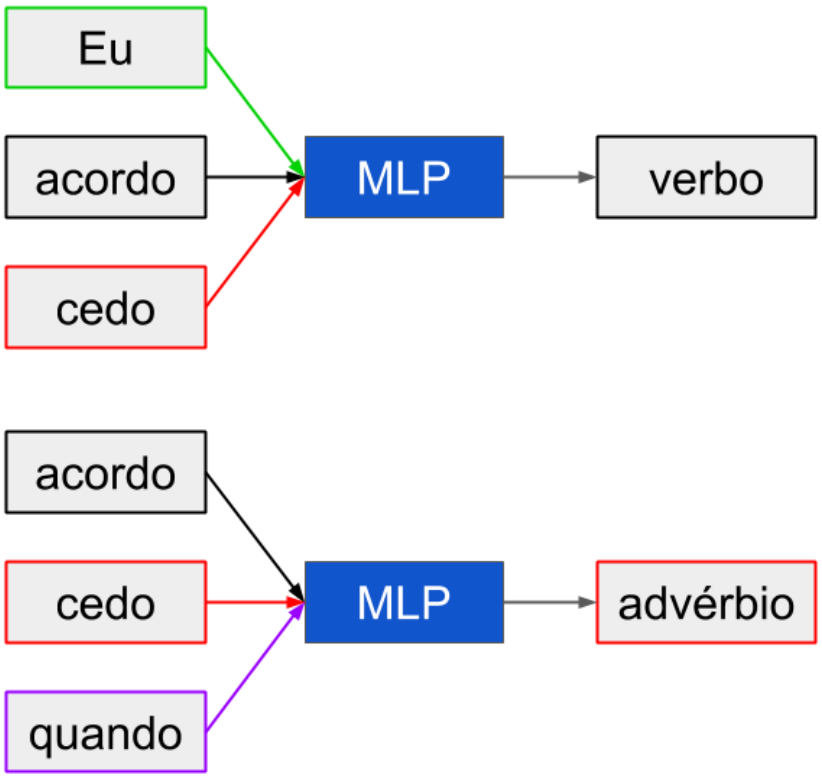

- Limitações dessa modelagem:
  - Input e output precisam ter tamanho fixo. Problemas do mundo real envolvem sequências de tamanho variável.
  - Saídas complexas como essa caem no problema intitulado "Predição estruturada" [1].
    - Trata-se de um problema muito difícil de otimizar.

[1] Linguistic Structure Prediction by Noah A. Smith http://www.cs.cmu.edu/ nasmith/LSP/

# MLP para Sequências

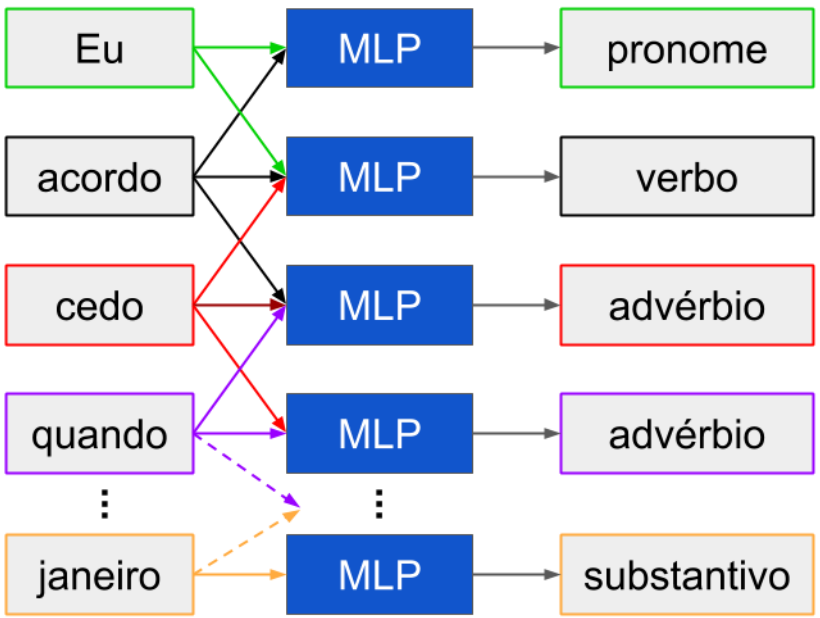

- Última tentativa!

# MLP para Sequências




- Define-se uma janela de tamanho $w$
- Saídas são geradas individualmente em função de uma parte da sequência.

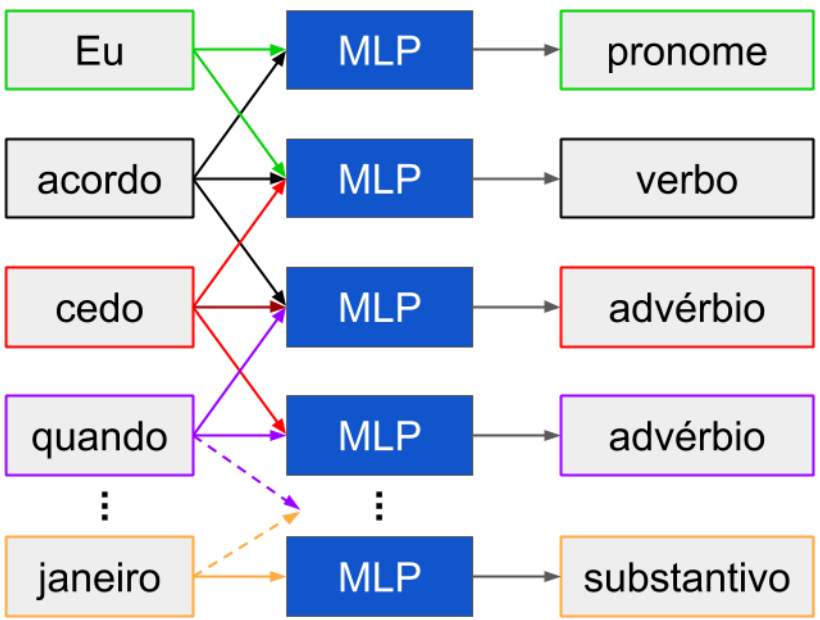
# MLP para Sequências

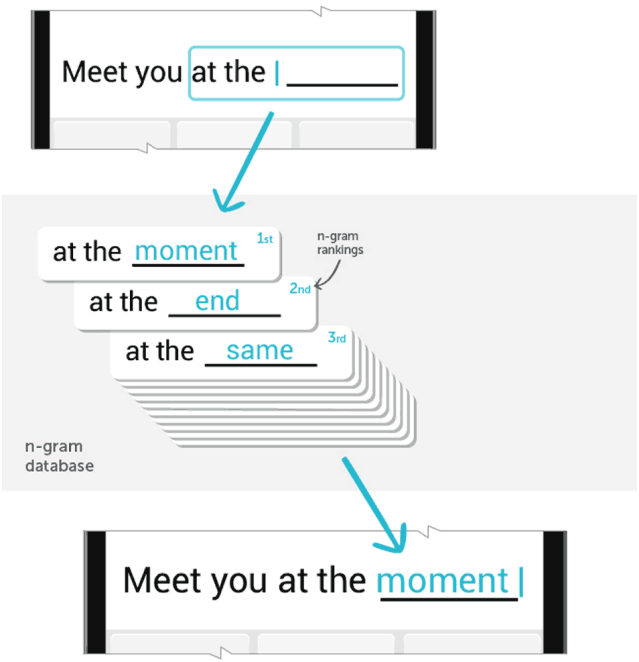



- Define-se uma janela de tamanho $w$
- Saídas são geradas individualmente em função de uma parte da sequência.

# MLP para Sequências

- Essa modelagem é utilizado para modelos de linguagem *n-gram*
- *n-gram* - Sequência contínua de *n* items dado um texto (Palavras, sílabas, letras, etc.)

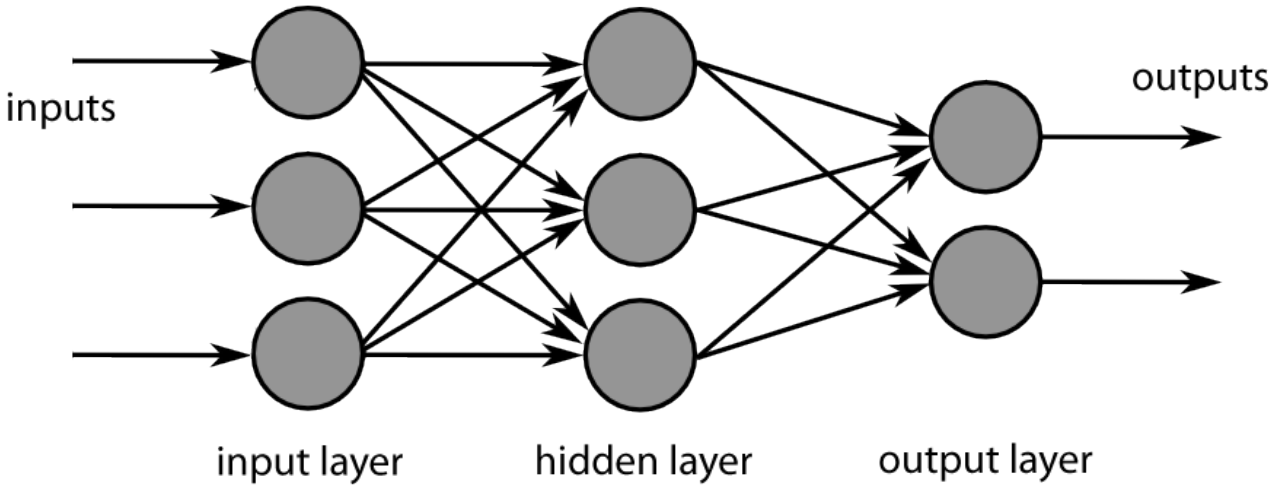


# MLP para Sequências

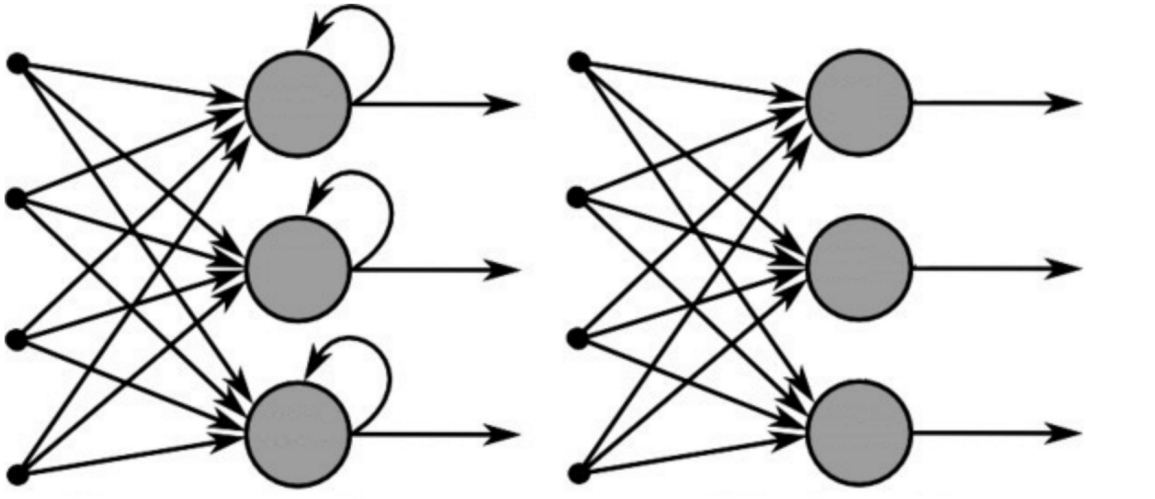

- Limitações dessa modelagem:
  - Uma janela fixa pode não atender bem todos os casos (algumas palavras podem exigir um contexto maior)
    - Dependências de longo prazo podem ser perdidas

# Limitações de Feed-Forward Networks




- Em resumo, as redes "feed-forward"apresentam limitações para lidar com sequências
- Em grande parte essas limitações estão associadas com a incapacidade de guardar memória das instâncias anteriores

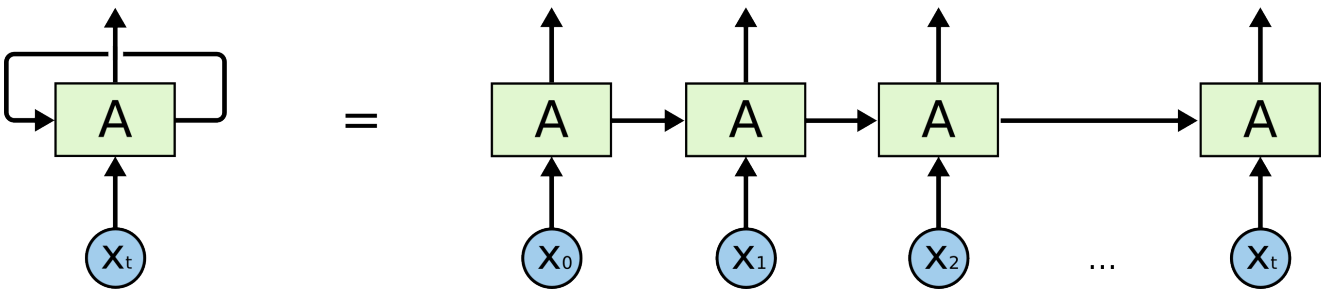
# Redes Neurais Recorrentes



Recurrent Neural Network  Feed-Forward Neural Network

- Diferente de unidades da Feed-Forward Network, unidades recorrentes guardam seu próprio estado, compondo uma memória interna.

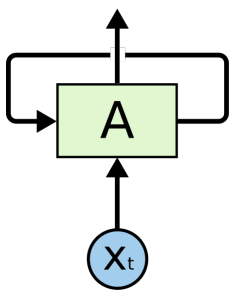
# Redes Neurais Recorrentes




- Cada entrada $x_t$ gera um estado que é retroalimentado para a unidade recorrente, permitindo que a informação do passado persista.
- Essa arquitetura permite lidar com sequências de tamanhos variáveis.

# Redes Neurais Recorrentes

- Cada entrada $x_t$ gera um estado que é retroalimentado para a unidade recorrente, permitindo que a informação do passado persista.

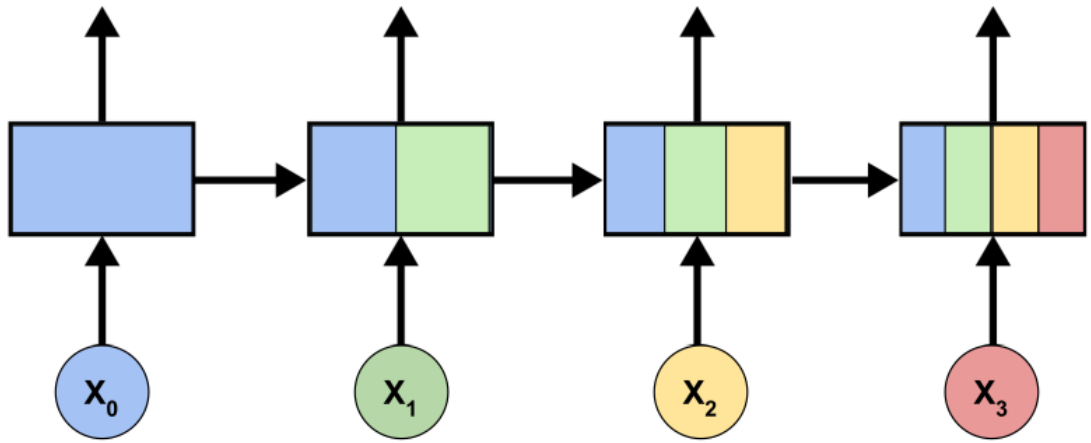



$$= \quad h_t = f(h_{t-1}, x_t)$$

# Redes Neurais Recorrentes

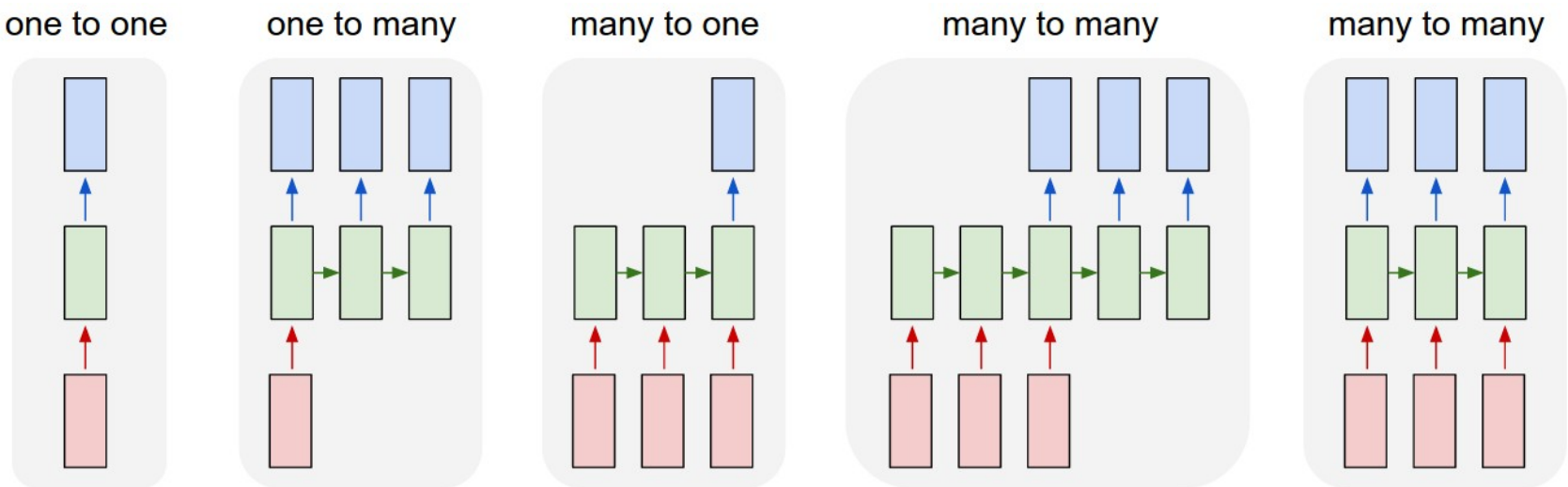

- Essa arquitetura permite lidar com sequências de tamanhos variáveis.
  - Ponto para o Pytorch! Outros frameworks (de grafos estáticos) exigem que você fixe o tamanho da sequência quando instancia a rede.

# Redes Neurais Recorrentes

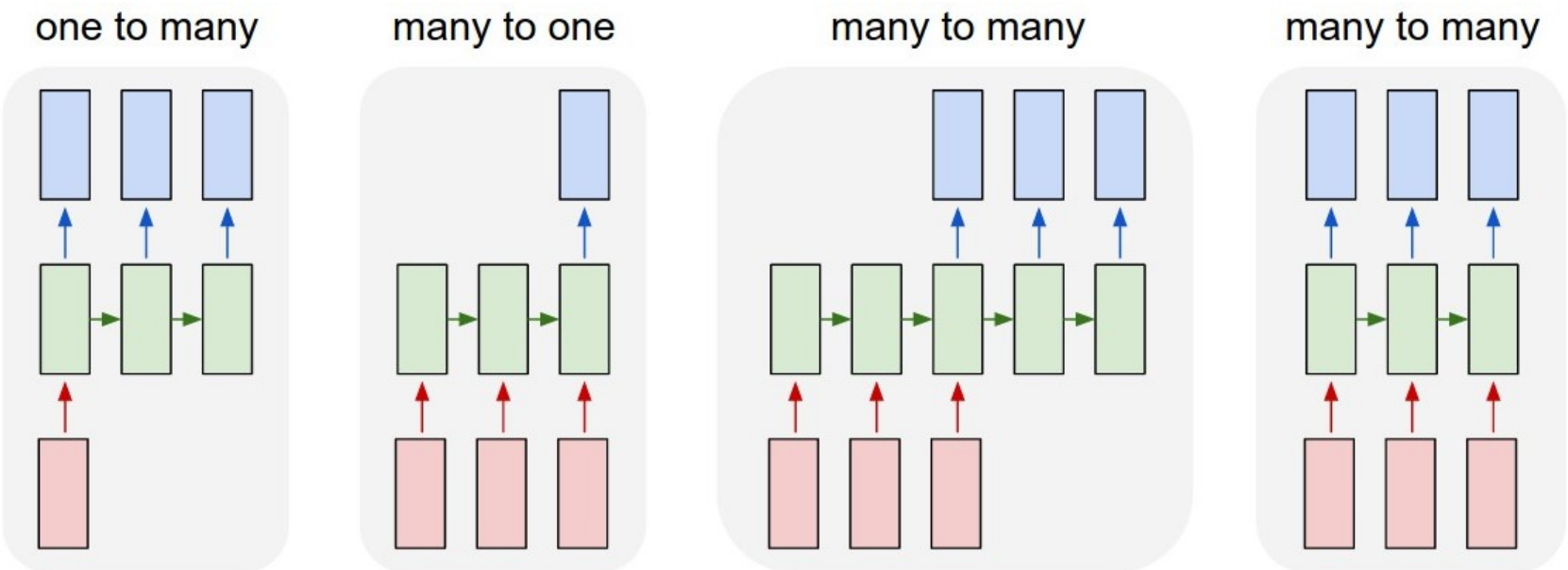



- A memória da rede recorrente persiste ao longo do tempo, ou seja, o estado $h_t$ acumula conhecimento dos seus predecessores $\{t-1, t-2, ..., 1, 0\}$.
- Certamente há um limite de persistência da memória. Discutiremos isso mais a frente.

# Taxonomia dos Problemas



# Taxonomia dos Problemas



# Taxonomia dos Problemas

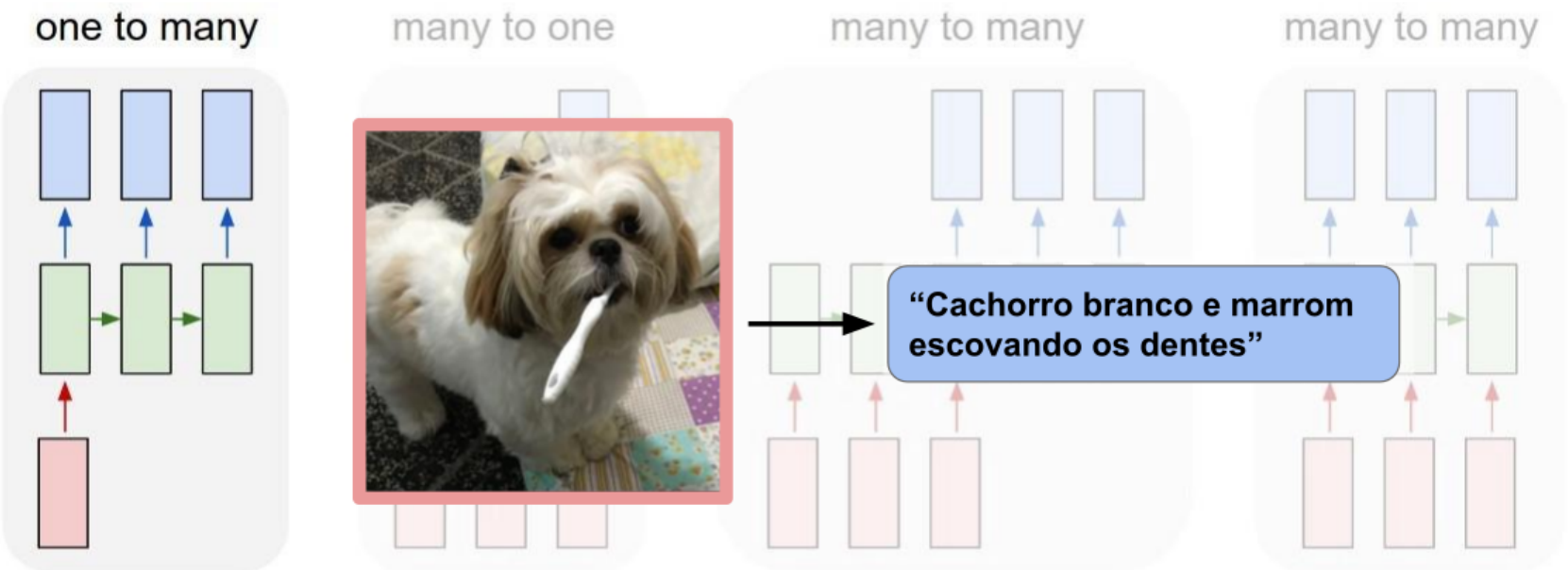



**Figura:** Exemplo de problema Um para Muitos: *Image Captioning*.

# Taxonomia dos Problemas

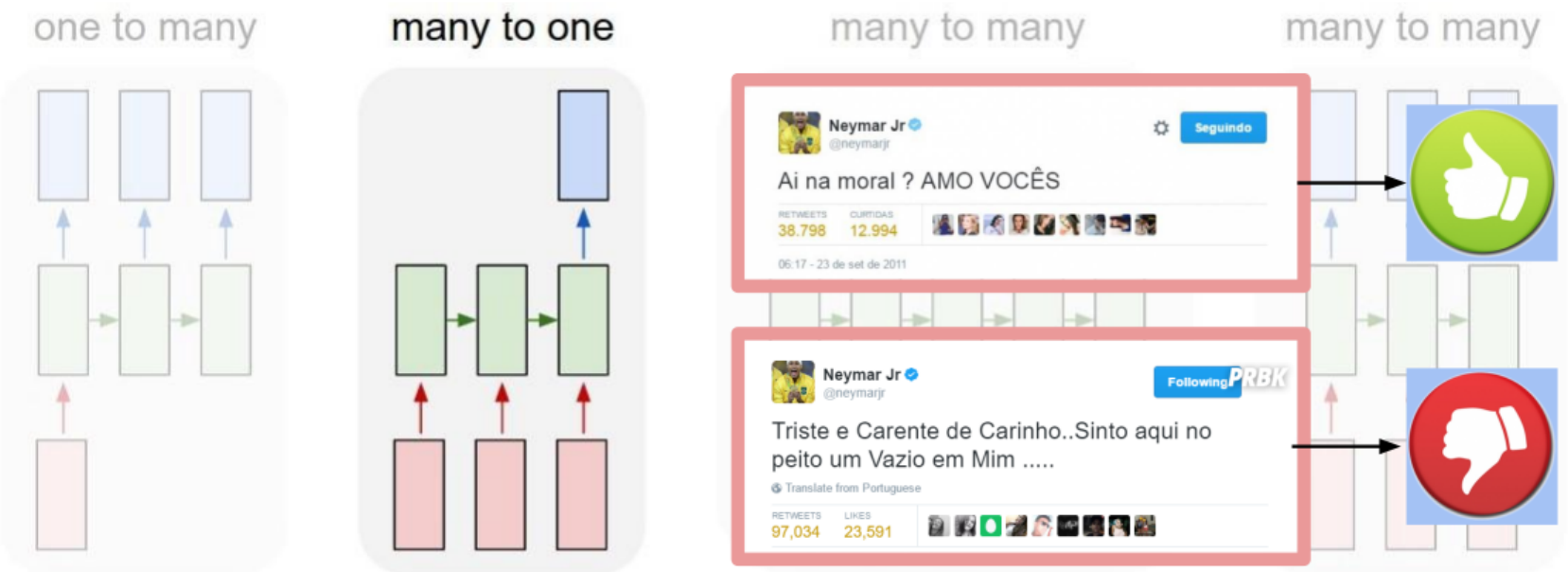



**Figura:** Exemplo de problema Muitos para Um: Análise de Sentimentos

# Taxonomia dos Problemas

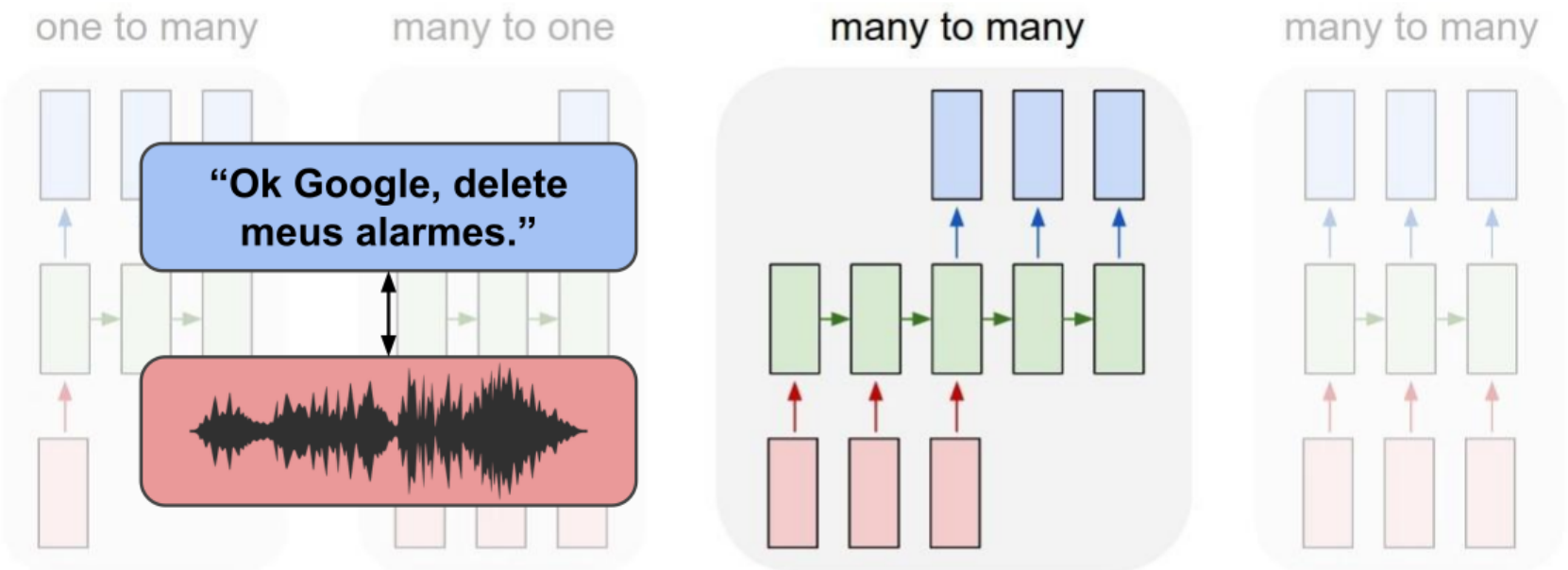



**Figura:** Exemplo de problema Muitos para Muitos: Voz para Texto

# Taxonomia dos Problemas

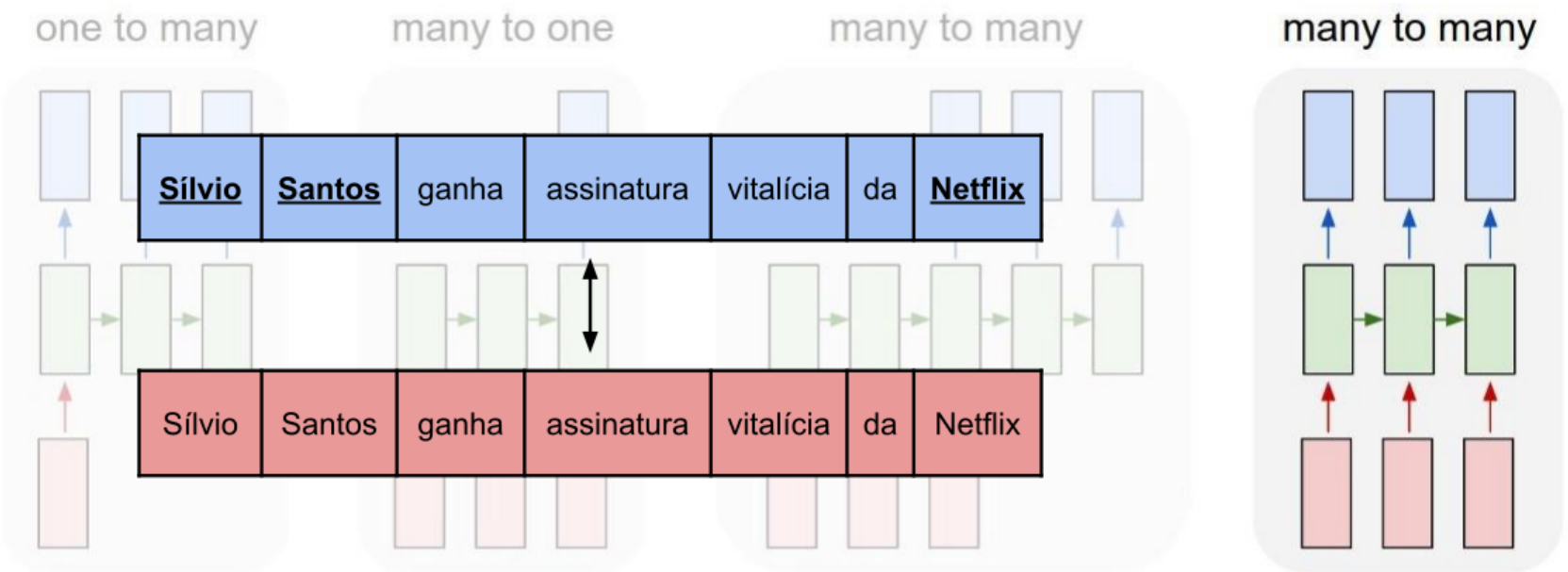



**Figura:** Exemplo de problema Muitos para Muitos sincronizado: Reconhecimento de entidade nomeada

# Outros problemas

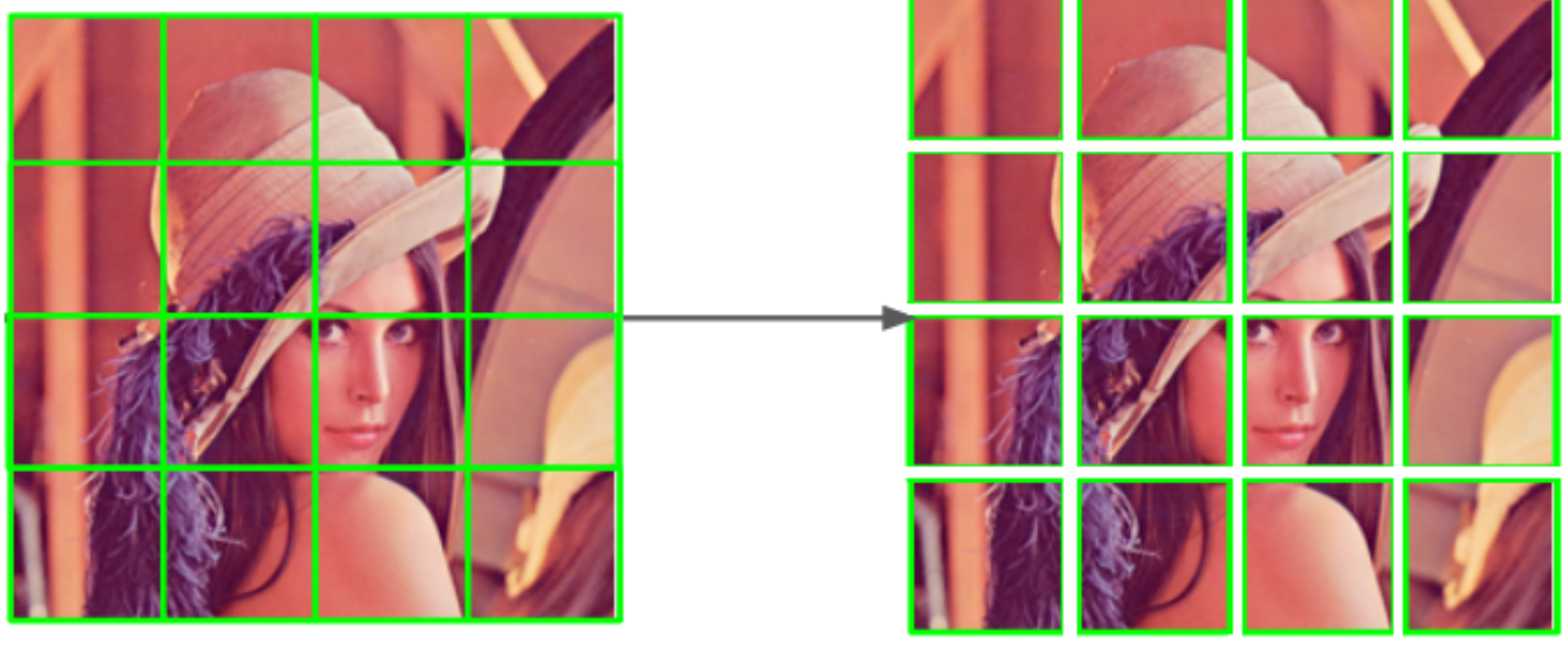

- Reconhecimento de ações em vídeo
- Detecção de anomalias em séries temporais
- Análise de sequências de DNA
- Composição de música
- Geração de texto
- etc...

# E imagens?


## Será que é possível aplicar redes recorrentes a uma única imagem?

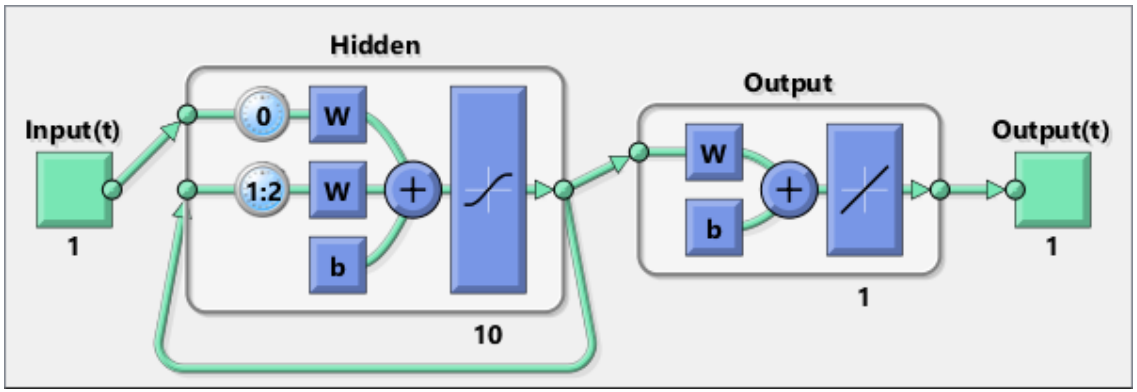
# E imagens?

## Será que é possível aplicar redes recorrentes a uma única imagem?



# Agenda

# Unidade Recorrente (*Vanilla*)




$$\begin{cases} h_t = g(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \\ y_t = g(W_{hy}h_t + b_y) \end{cases}$$

```
self.h = np.tanh(np.dot(self.W_hh, self.h) +
                 np.dot(self.W_xh, x) + b_h)

y = sigm(np.dot(self.W_hy, self.h) + b_y)
```

# Unidade Recorrente (*Vanilla*)

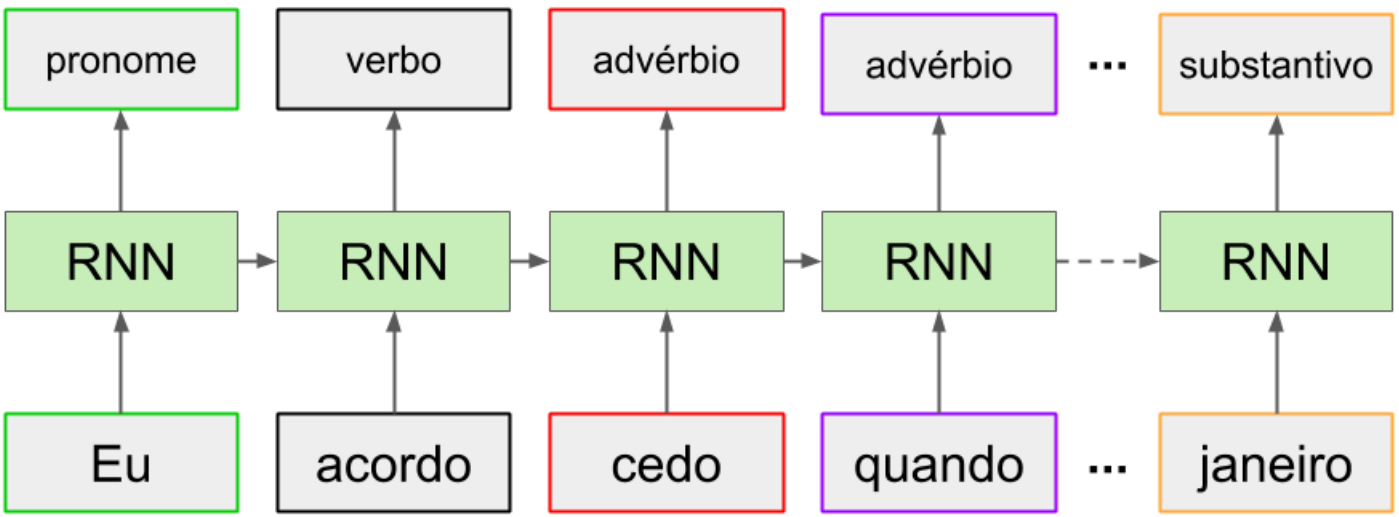
- Parâmetros Otimizáveis
  - Pesos
    - $W_{hh}$ - *hidden to hidden*
    - $W_{xh}$ - *input to hidden*
    - $W_{hy}$ - *hidden to output*
  - Bias
    - $b_h$ - *bias hidden*
    - $b_y$ - *bias output*

$$\begin{cases} h_t = g(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \\ y_t = g(W_{hy}h_t + b_y) \end{cases}$$

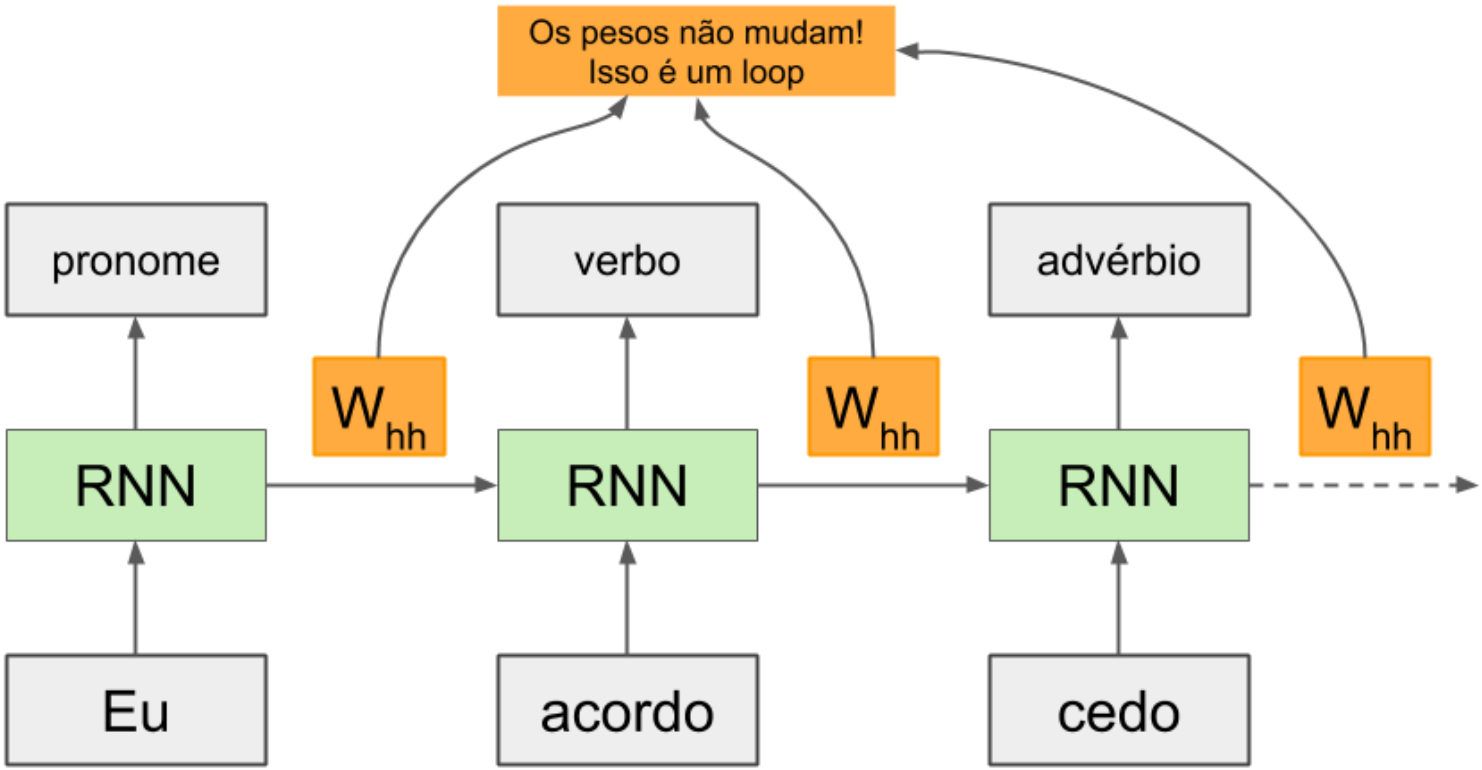
```
self.h = np.tanh(np.dot(self.W_hh, self.h) +
                 np.dot(self.W_xh, x) + b_h)

y = sigm(np.dot(self.W_hy, self.h) + b_y)
```

# Feed Forward - Many-to-Many


- Como fica a solução para o nosso problema de *sequence tagging* usando uma RNN?

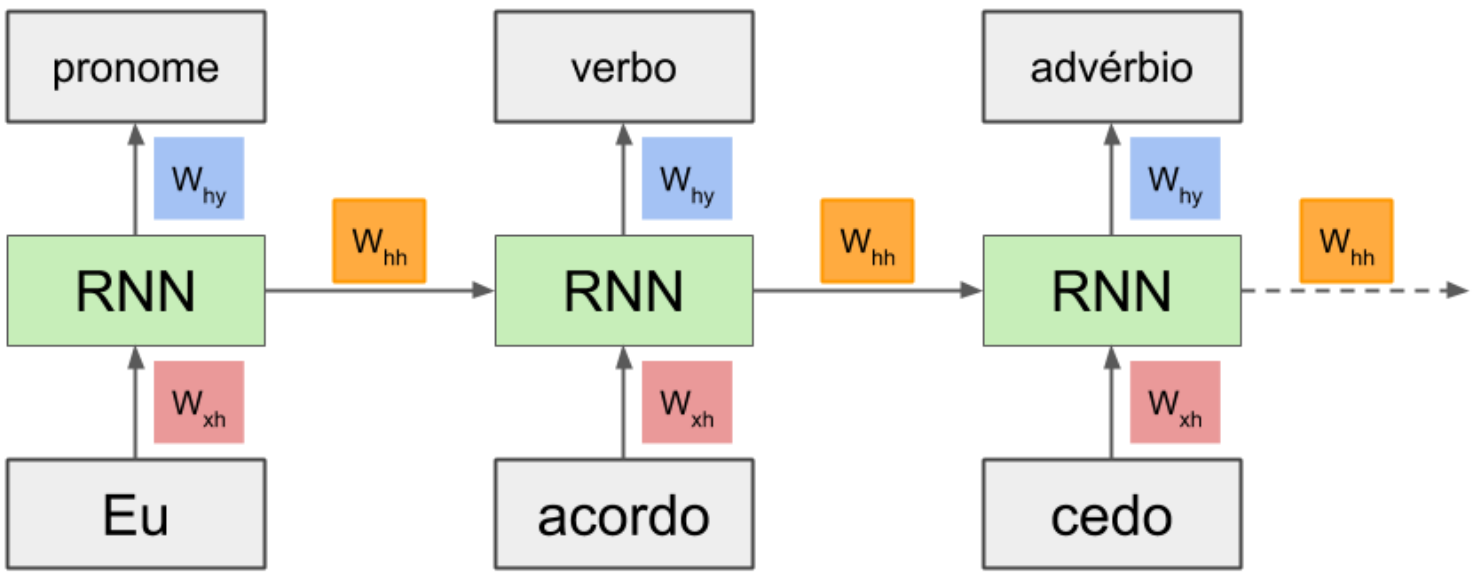


# Feed Forward - Many-to-Many



# Feed Forward - Many-to-Many

- Os pesos e os bias são os mesmos para todos os *timesteps* (iterações) ao longo da entrada



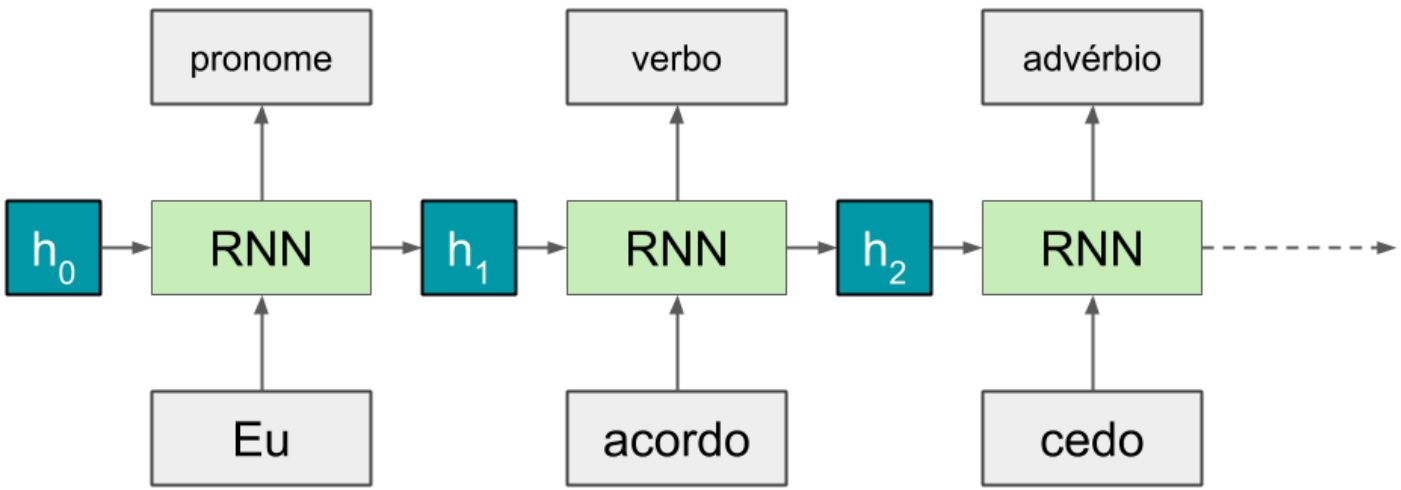
# Feed Forward - Many-to-Many


- Os pesos e os bias são os mesmos para todos os *timesteps* (iterações) ao longo da entrada

In [ ]:
```
class RNN():
    ...

    def step(x):
        self.hidden = np.tanh( np.dot(self.W_hh, self.hidden) +
                               np.dot(self.W_xh, x) + b_h)
        y = sigm( np.dot(self.W_hy, self.hidden) + b_y )

        return y

    def forward(input_data):

        output = []
        for x in input_data:
            output.append(self.step(x))
```

# Feed Forward - Many-to-Many


- O *hidden state* precisa ser inicializado a cada novo batch de sequências.



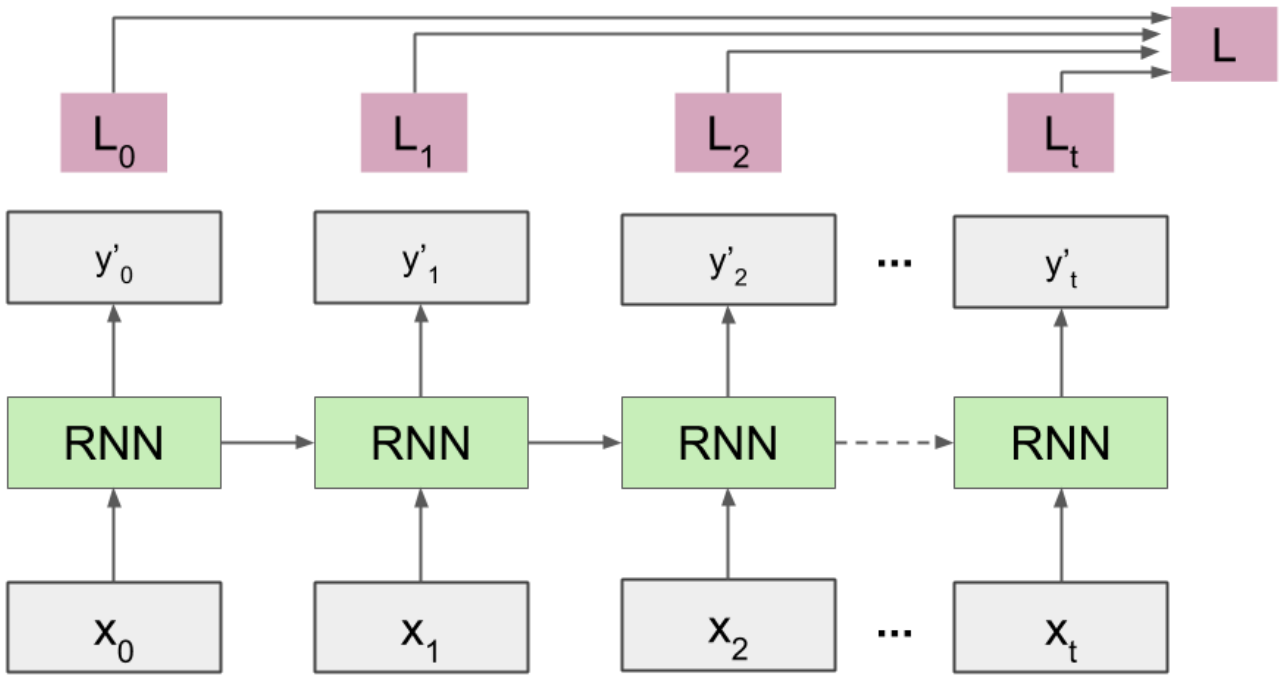
# Feed Forward - Many-to-Many


- O *hidden state* precisa ser inicializado a cada novo batch de sequências.

In [ ]:
```
class RNN():
    ...

    def step(x):
        self.hidden = np.tanh( np.dot(self.W_hh, self.hidden) +
                               np.dot(self.W_xh, x) + b_h)
        y = sigm( np.dot(self.W_hy, self.hidden) + b_y )

        return y

    def forward(input_data):

        self.hidden = np.zeros(self.batch_size, self.hidden_size)

        # input_data.size() = (seq_len, batch_size, input_size)
        output = []
        for x in input_data:
            output.append(self.step(x))
```

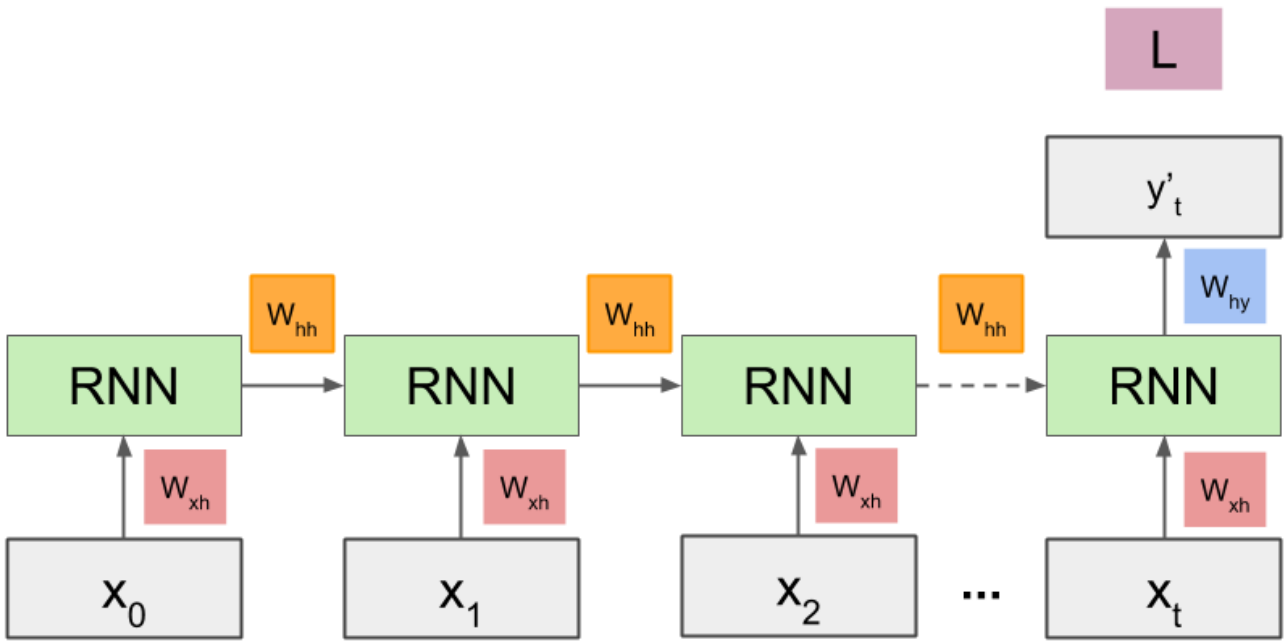
# Feed Forward: Many-to-Many


- A função de perda é dada pelo acúmulo das perdas ao longo dos *timesteps*



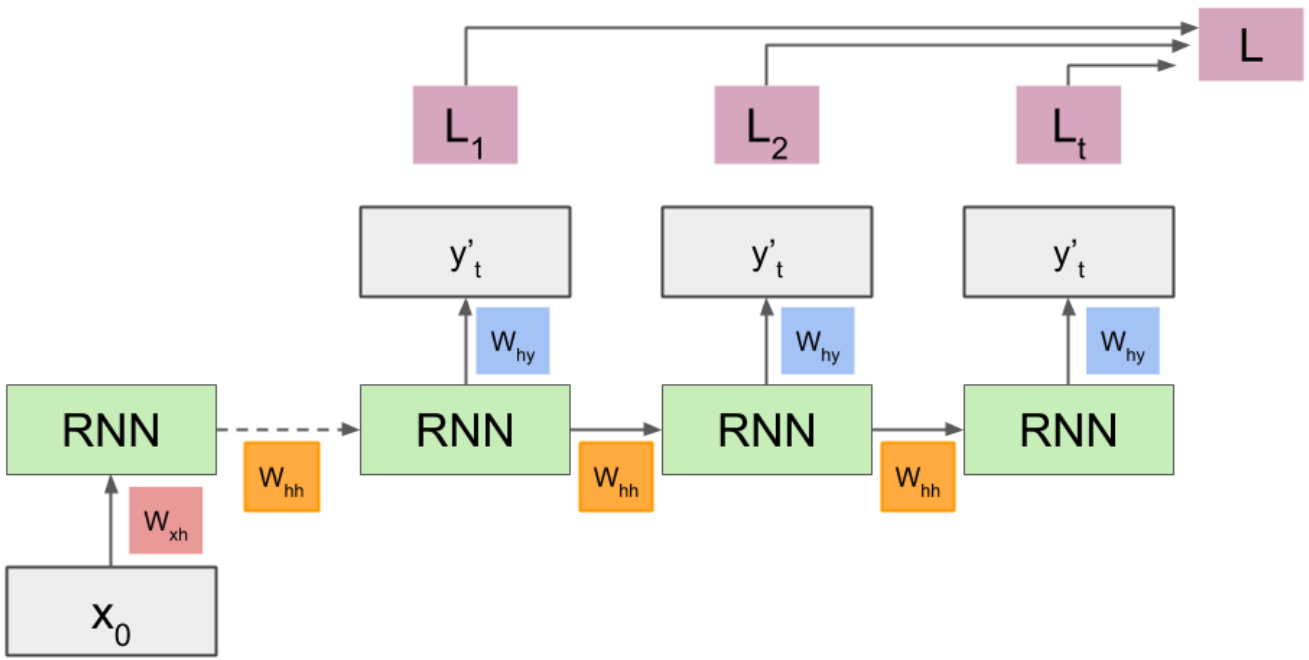
# Feed Forward: Many-to-One

- Função de perda é dada apenas pelo último *timestep*
- Função de output apenas no último *hidden state*



# Feed Forward: One-to-Many

- A função de perda é dada pelo acúmulo das perdas ao longo dos *timesteps*
- Apenas uma operação em função do input



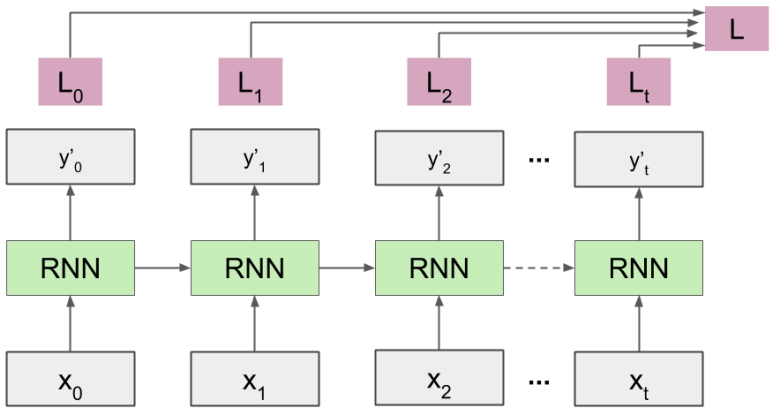
# Backpropagation Through Time


- A *backpropagation* nesse caso consiste em:
  - Desenrolar a rede para calcular o gradiente
  - Enrolar novamente para propagar
- Vamos considerar novamente o nosso problema de *sequence tagging*

# Backpropagation Through Time

- Na etapa de feed-forward computamos a perda em função do acúmulo das perdas intermediárias

$$\mathcal{L} = \sum_{t=0}^{T} \mathcal{L}_t(y'_t, y_t)$$

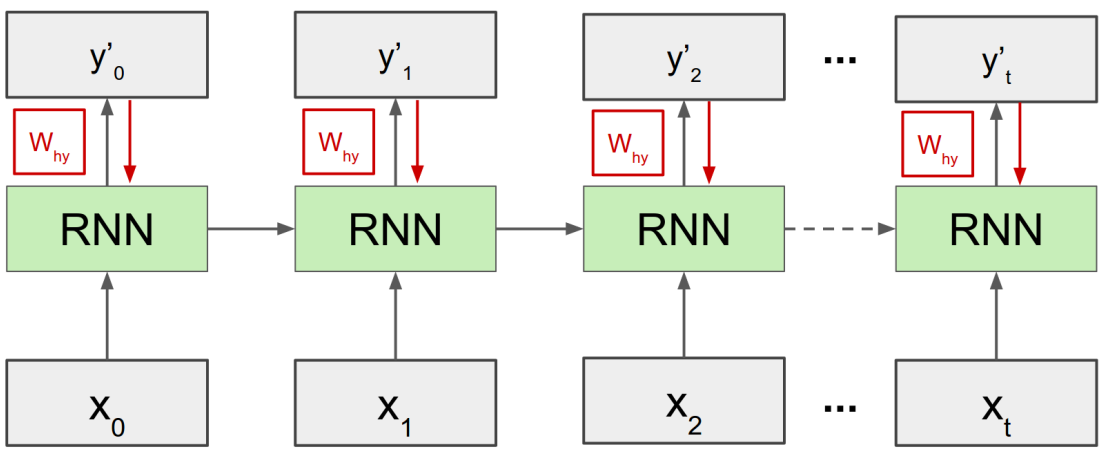



- Mas como fica o gradiente nessa confusão?

# Backpropagation Through Time


- O gradiente final também é dado pelo acúmulo dos gradientes.
  - Em relação a $W_{hy}$

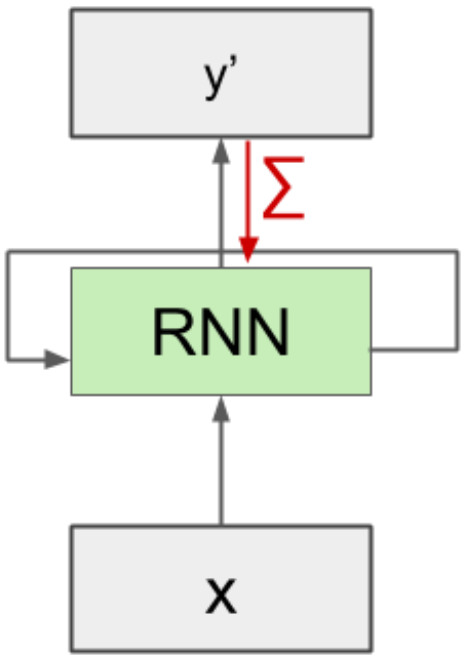
$$\sum_{t=0}^{T} \frac{\partial \mathcal{L}_t}{\partial W_{hy}}$$




[1] Denny Britz. Recurrent Neural Network Tutorial, Part 4. 2018. http://www.wildml.com/2015/10/

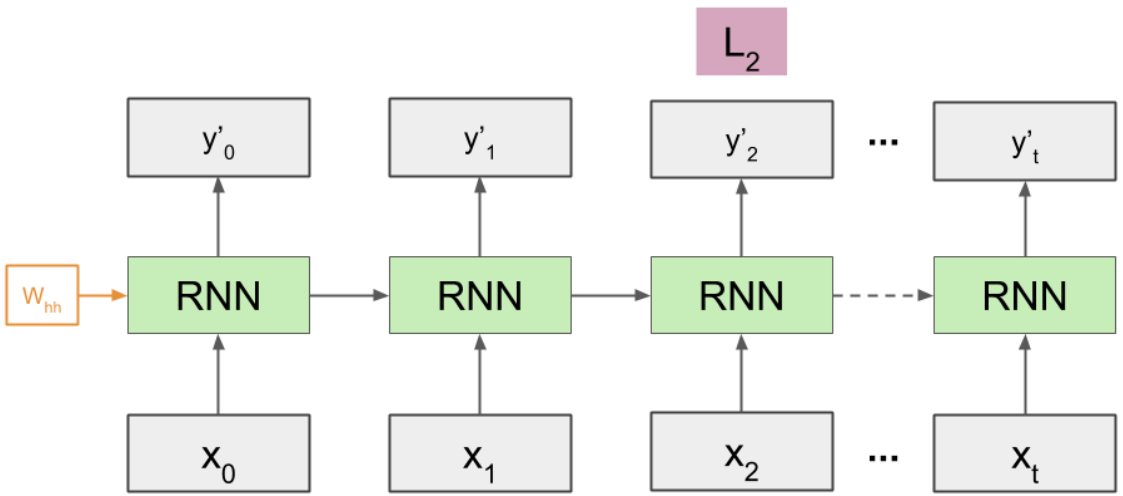
# Backpropagation Through Time


- Depois é só atualizar $W_{hy}$ propagando o gradiente acumulado!
- Mas como calcular $\frac{\partial \mathcal{L}}{\partial W_{hh}}$ e $\frac{\partial \mathcal{L}}{\partial W_{xh}}$ ?



# Backpropagation Through Time


- Como calcular $\frac{\partial \mathcal{L}_t}{\partial W_{hh}}$ **para um timestep**? [1]
- Precisamos aplicar a regra da cadeia desde o *timestep* atual até $t = 0$




[1] Denny Britz. Recurrent Neural Network Tutorial, Part 4. 2018. http://www.wildml.com/2015/10/

# Backpropagation Through Time


- Como calcular $\frac{\partial \mathcal{L}_t}{\partial W_{hh}}$ ? [1]
- Precisamos aplicar a regra da cadeia desde o *timestep* atual até $t = 0$

$$\frac{\partial \mathcal{L}_t}{\partial W_{hh}} = \sum_{k=0}^{t} \frac{\partial \mathcal{L}_t}{\partial y'_t} \frac{\partial y'_t}{\partial h_t} \left( \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_{hh}}$$

- Repetimos o mesmo processo para todos os timesteps, acumulando os resultados

$$\sum_{t=0}^{T} \frac{\partial \mathcal{L}_t}{\partial W_{hh}}$$

[1] Denny Britz. Recurrent Neural Network Tutorial, Part 4. 2018. http://www.wildml.com/2015/10/

# Backpropagation Through Time


- O mesmo vale para $W_{xh}$ [1]
- Precisamos aplicar a regra da cadeia desde o *timestep* atual até $t = 0$

$$\frac{\partial \mathcal{L}_t}{\partial W_{xh}} = \sum_{k=0}^{t} \frac{\partial \mathcal{L}_t}{\partial y'_t} \frac{\partial y'_t}{\partial h_t} \left( \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_{xh}}$$

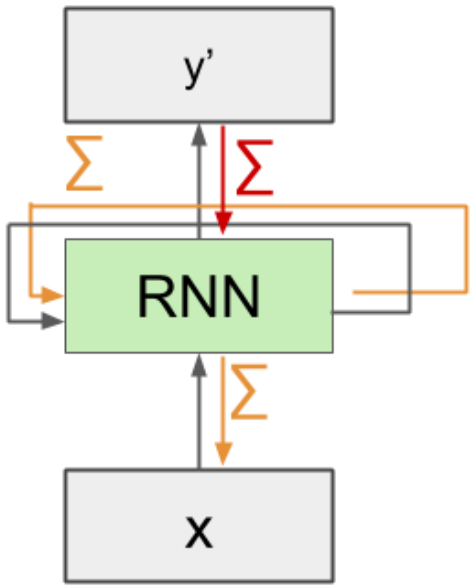
- Repetimos o mesmo processo para todos os timesteps, acumulando os resultados

$$\sum_{t=0}^{T} \frac{\partial \mathcal{L}_t}{\partial W_{xh}}$$

[1] Denny Britz. Recurrent Neural Network Tutorial, Part 4. 2018. http://www.wildml.com/2015/10/

# Backpropagation Through Time

- Depois é só atualizar os pesos propagando os gradientes acumulados!



# Vanishing / Exploding Gradient


- Relembrando:
  - **vanishing gradient**: gradiente tende a 0
  - **exploding gradient**: gradiente tende a infinito
- É um problema que se intensifica nas redes recorrentes
- Cada iteração da célula recorrente impacta a propagação como adicionar profundidade em uma MLP
- Dependências de longo prazo exigem longas sequências
  - Como consequência, o estado das iterações mais antigas não contribuem para o aprendizado (em caso de vanishing)

# RNNCell no Pytorch


- RNNCell (torch.nn.RNNCell)
  - input_size: Número de features da entrada
  - hidden_size: Número de features no *hidden state*

In [ ]:
```
class RNN():
    def init(self, input_size, hidden_size):
        self.rnn = torch.nn.RNNCell(input_size, hidden_size)

    def forward(input_data):
        # Set initial hidden and cell states
        self.hidden = Variable(torch.zeros(batch_size, hidden_size))

        for x in input_data:
            self.hidden = self.rnn(x, self.hidden)
```

# RNNCell no Pytorch


- Detalhes de implementação (atividade prática)
  - RNNCell (torch.nn.RNNCell)
    - input_size: Número de features da entrada
    - hidden_size: Número de features no *hidden state*
  - Linear (torch.nn.Linear)
    - in_features: Tamanho da entrada
    - out_features: Tamanho da saída
    - bias: [True, False]
  - Ativação LogSoftmax (torch.nn.LogSoftmax)
    - escolha arbitrária para o problema da atividade

# Procedimento de Treinamento de RNNs


## Atividade Prática

- Arquitetura
  - Camada RNNCell (input_size, hidden_size)
  - Camada Fully Connected (hidden_size, output_size)
  - Camada LogSoftmax
- Forward (Many-to-One)
  - # Inicialize o estado interno da RNN
  - # Loop ao longo da sequência para alimentar a RNNCell
  - # Alimente as camadas Fully Connected e LogSoftmax

# Atividade Prática

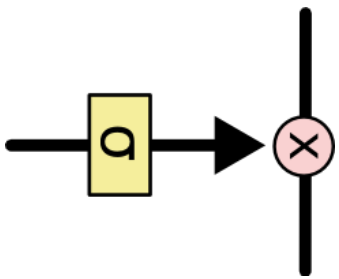

## Classificando nomes próprios

**rnn_classification.py**

# Agenda

# Unidades Avançadas


- Existem duas variações de redes recorrentes que se tornaram muito populares na literatura:
  - GRU - Gated Recurrent Unit
  - LSTM - Long Short-Term Memory
- São capazes de aprender dependências de longo prazo
- Evitam o problema de *vanishing* / *exploding gradient*
- Seu estado interno é reguladado por um conjunto de "*gates*"

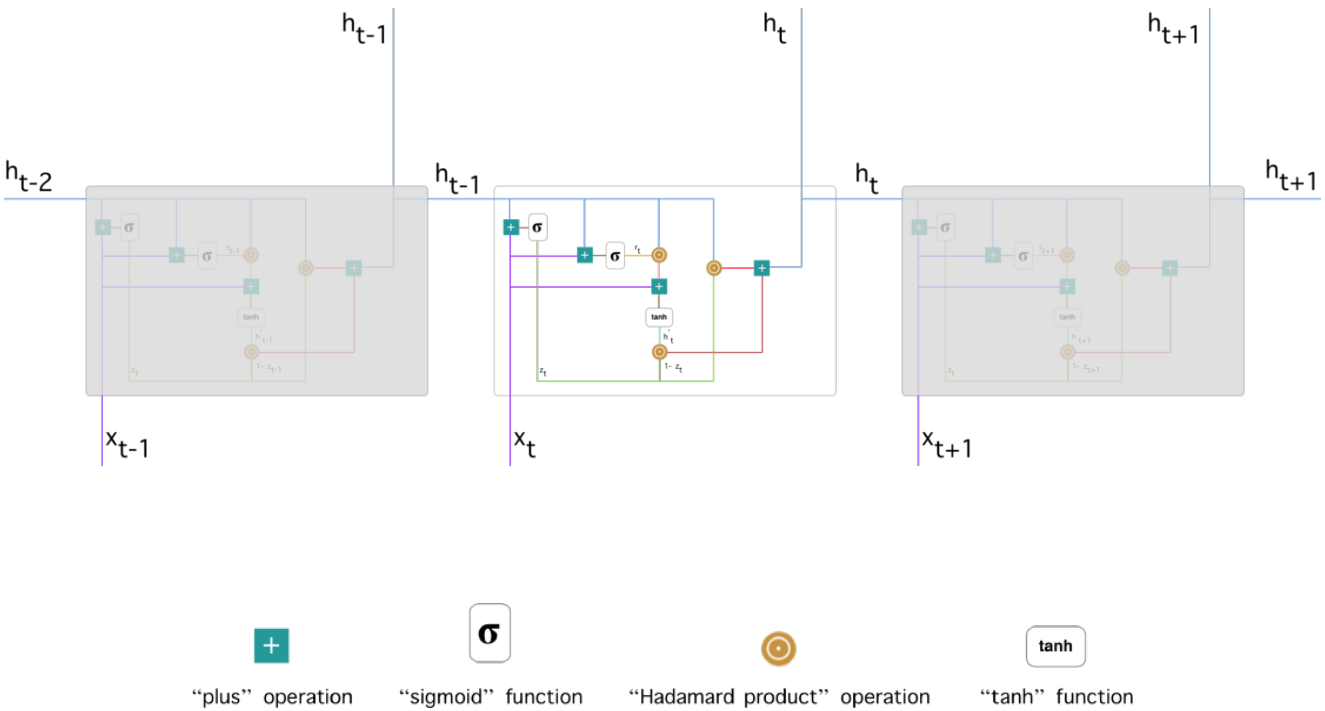



  - Literalmente um portão que decide quanto da informação pode passar.

# GRU - Gated Recurrent Unit


- É composta por dois *gates*
  - Update Gate $z_t$
  - Reset Gate $r_t$
- Possui uma memória interna $h'_t$ além da já existente $h_t$

# GRU - Gated Recurrent Unit [2]

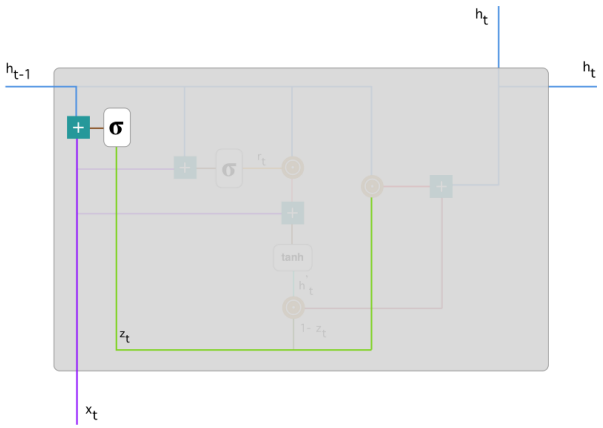



[2] https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

# GRU Gates


**Update Gate** ($z_t$)

- Combina o novo input $x_t$ e o estado anterior $h_{t-1}$
- Gate resultante define quanto da informação do passado deve ser **lembrada**

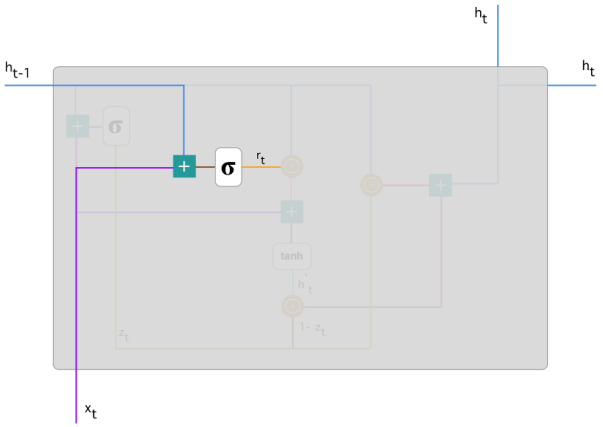



$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

# GRU Gates

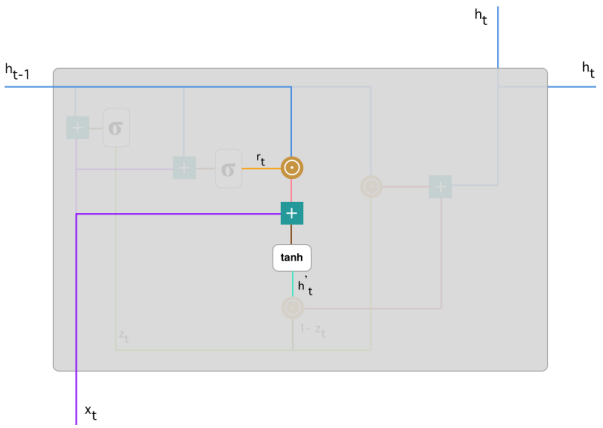

**Reset Gate** ($r_t$)

- Combina o novo input $x_t$ e o estado anterior $h_{t-1}$
- Gate resultante define quanto da informação do passado deve ser **esquecida**




$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

# GRU Gates




**Memória interna** ($h'_t$)

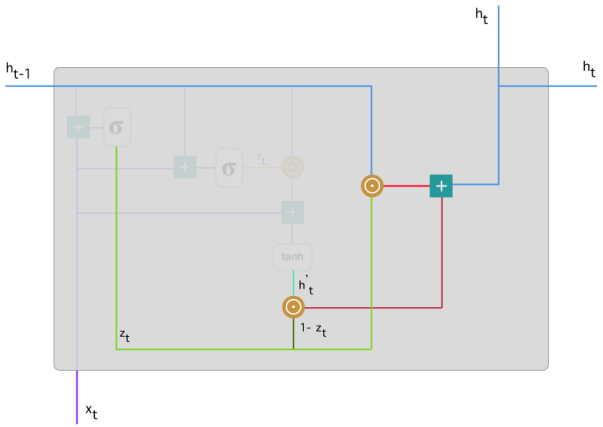
- Aplica o **reset gate** sobre a informação do passado
- Operação *element-wise*

$$h_t^{'} = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

# GRU Gates

**Hidden state** ($h_t$)

- Aplica o **update gate** sobre a informação do passado e a memória interna
- Operação *element-wise*




$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t^{'}$$

# GRUCell no Pytorch


- GRUCell (torch.nn.RNNCell)
  - input_size
  - hidden_size
  - bias
- A utilização é essencialmente a mesma que a RNNCell

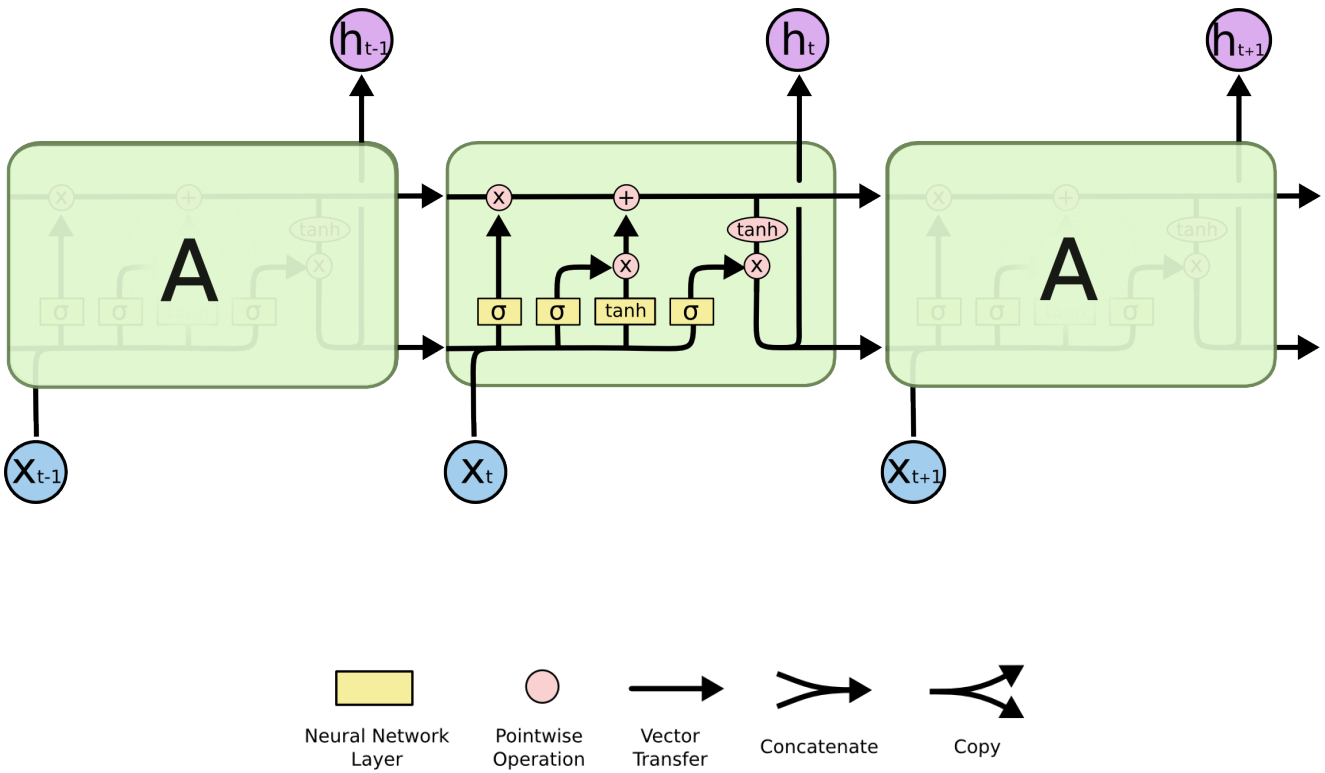
In [ ]:
```
class GRU():
    def init(self, input_size, hidden_size):
        self.rnn = torch.nn.GRUCell(input_size, hidden_size)

    def forward(input_data):
        # Set initial hidden and cell states
        self.hidden = Variable(torch.zeros(batch_size, hidden_size))

        for x in input_data:
            self.hidden = self.rnn(x, self.hidden)
```

- Com a diferença que o resultado costuma ser melhor!

# LSTM - Long Short-Term Memory


- É composta por três *gates*
  - Forget Gate $f_t$
  - Input Gate $i_t$
  - Output Gate $o_t$
- Possui um estado interno (*cell state* $C_t$) atualizado de maneira mais estável
- Rede recorrente mais popular atualmente na literatura

# LSTM - Long Short-Term Memory [3]

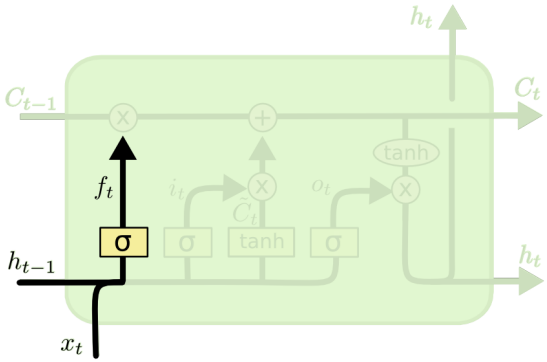



[3] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

# LSTM Gates


**Forget Gate** ($f_t$)

- Decide quanto da informação **anterior** será esquecida

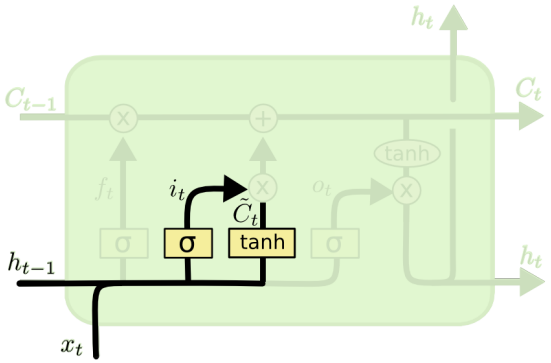



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

# LSTM Gates

## Input Gate

- ($i_t$): Decide quanto da informação **nova** vai ser incorporada no estado da célula
- ($C'_t$): Valores candidatos a estado da célula (*Cell State*)



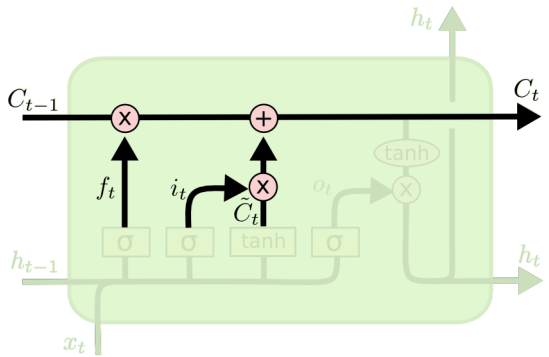

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

# LSTM Gates


**Cell State** ($C_t$)

- Aplica o **input gate** no vetor de valores candidatos
- Aplica o **forget gate** no estado da célula do passado
- Essa combinação atualiza o *cell state* através de uma **soma**.

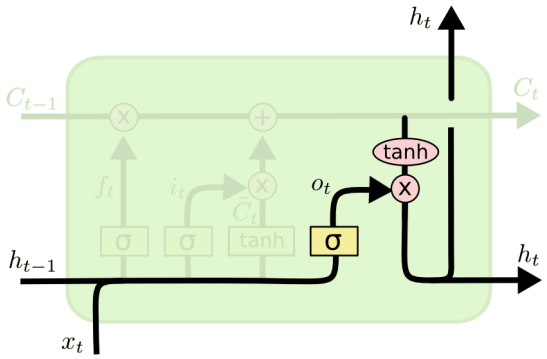



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

# LSTM Gates

**Output Gate** ($o_t$)

- Decide quanto do estado da célula será passado para a próxima iteração.
- Output gate é aplicado no estado da célula para gerar o estado interno ($h_t$)




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

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

# LSTMCell no Pytorch


- Detalhes de implementação (atividade prática)
  - LSTMCell (torch.nn.LSTMCell)
    - input_size
    - hidden_size
    - bias
  - Um parâmetro a mais para controlar: Cell State

In [ ]:
```
class LSTM():
    def init(input_size, hidden_size):
        self.rnn = torch.nn.LSTMCell(input_size, hidden_size)

    def forward(input_data):
        # Set initial hidden and cell states
        self.hidden     = Variable(torch.zeros(batch_size, hidden_size))
        self.cell_state = Variable(torch.zeros(batch_size, hidden_size))

        for x in input_data:
            self.hidden, self.cell_state = self.lstm(x, (self.hidden, self.cell_state))
            output = self.linear(self.hidden)
```

# Procedimento de Treinamento de RNNs


## Atividade Prática

- Arquitetura
  - Camada LSTMCell (input_size, hidden_size)
  - Camada LSTMCell (hidden_size, hidden_size)
  - Camada Fully Connected (hidden_size, output_size)
- Forward (Many-to-Many)
  - # Inicialize o estado interno da LSTM ($h_t$ e $C_t$)
  - # Loop ao longo da sequência para alimentar as camadas LSTMCell e Fully Connected
  - # Em tempo de teste, loop para prever instâncias futuras

# Procedimento de Treinamento de RNNs


- Em tempo de teste, loop para prever instâncias futuras

In [ ]:
```
class LSTM():
    ...

    def forward(input_data, future=0):
        ...

        for x in input_data:
        ...

        for i in range(future):
            self.hidden, self.cell_state = self.lstm(output, (self.hidden, self.cell_state))
            output = self.linear(self.hidden)
```

# Atividade Prática


## Prevendo instâncias futuras

**forecast.py**

# Multilayers em Pytorch


- Diferente das unidades **Cell*, o Pytorch oferece outro tipo de camada
  - RNN (torch.nn.RNN)
  - GRU (torch.nn.GRU)
  - LSTM (torch.nn.LSTM)

# Multilayers em Pytorch


- O laço de repetição que implementamos até então é realizado internamente na camada
- É mais rápido que iterar nas unidades tipo **Cell*
- Retorna o estado interno para $t = seq\_len$ ($hn$, $cn$)

In [ ]:
```
def init(self, input_size, hidden_size):
    self.rnn = torch.nn.LSTMCell(input_size, hidden_size)

def forward(self, input_data):
        h, c = init_hidden()
        output = []
        for x in input_data:
            h, c = self.rnn(x, (h,c))
            output.append(h)
```

In [ ]:
```
def init(self, input_size, hidden_size):
    self.rnn = torch.nn.LSTM(input_size, hidden_size)

def forward(self, input_data):
        h0, c0 = init_hidden()
        output, (hn, cn) = self.rnn(input_data, (h0,c0))
```

# Multilayers em Pytorch


- RNN (torch.nn.RNN)
- GRU (torch.nn.GRU)
- LSTM (torch.nn.LSTM)

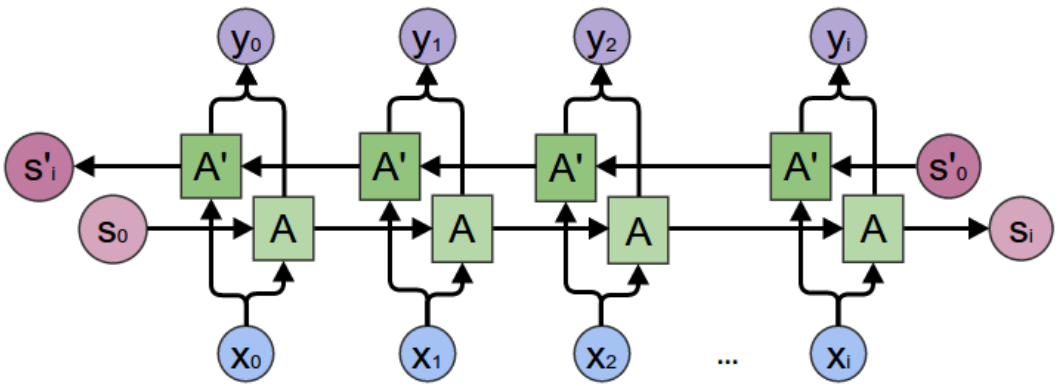
- Parâmetros
  - Input Size
  - Hidden Size
  - Bias
  - **Num Layers**
  - **Batch First**
  - **Dropout**
  - **Bidirectional**

# Multilayers em Pytorch


- Num Layers: Número de camadas recorrentes
  - A segunda camada recebe inputs da primeira, a terceira recebe da segunda, e assim por diante.
- Batch First: marcador booleano
  - batch_first = **False**: input.size() = (seq_len, **batch_size**, input_size)
  - batch_first = **True**: input.size() = (**batch_size**, seq_len, input_size)

# Multilayers em Pytorch

- Dropout: float
  - Introduz uma camada de Dropout depois de todas as camadas recorrentes, exceto a última
- Bidirectional: marcador booleano
  - Bidirectional = **True**



# Multilayers em Pytorch

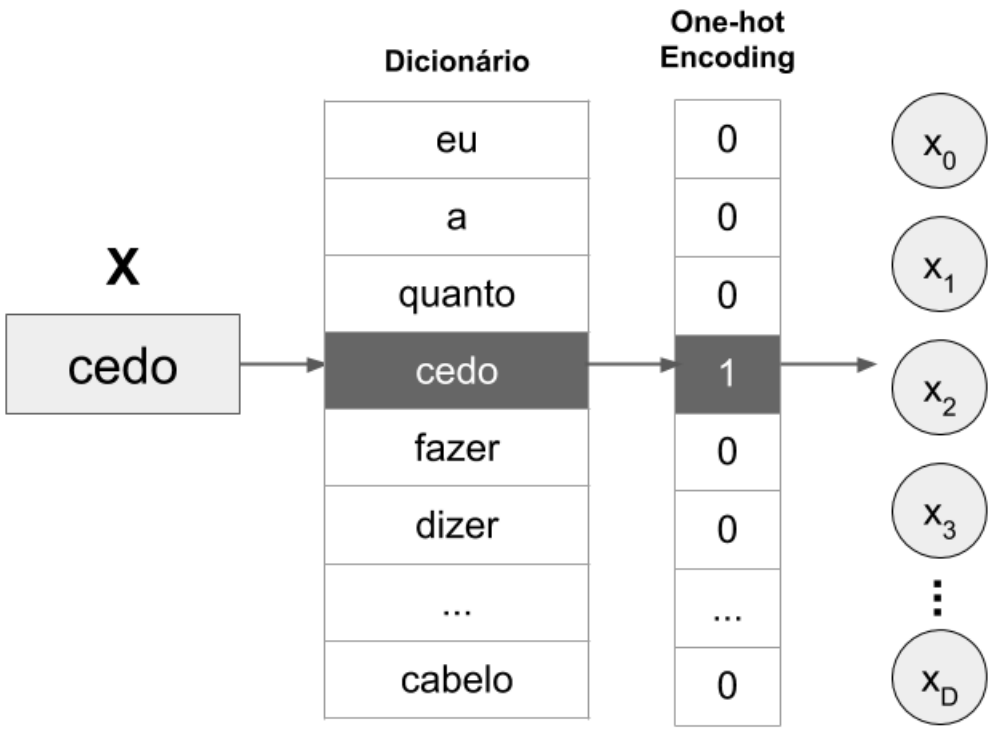

- Detalhes de implementação (atividade prática)
  - Embedding (torch.nn.Embedding)
    - input_size
    - embbeding_size
  - GRU (torch.nn.GRU)
    - input_size
    - hidden_size
    - num_layers
    - batch_first
    - dropout
  - Linear (torch.nn.Linear)
    - in_features: Tamanho da entrada
    - out_features: Tamanho da saída

# Embedding Layer


- Transforma valores inteiros em vetores densos
  - Por que?

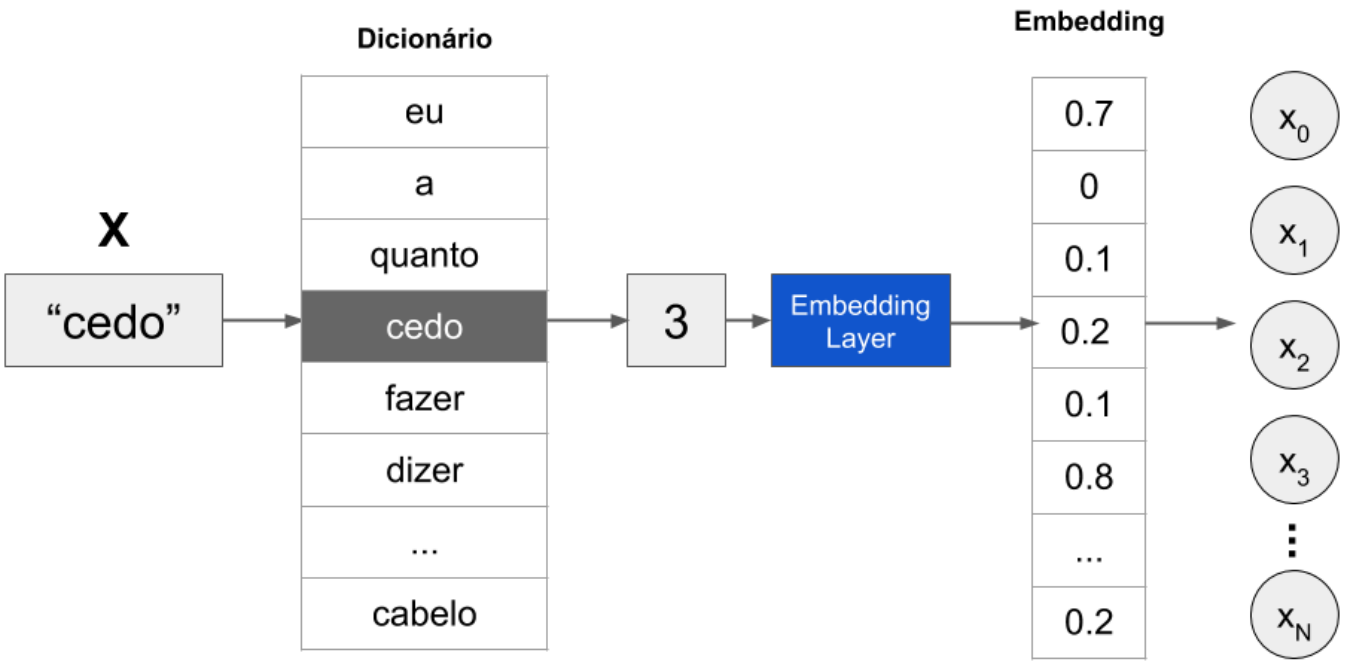
# Embedding Layer

- Lembram da representação one-hot?



# Embedding Layer

- Que tal aprender um vetor latente de tamanho controlável?



# Procedimento de Treinamento de RNNs

## Atividade Prática

- Arquitetura
  - Embedding (input_size, embedding_size)
  - Camada RNN (input_size, hidden_size, batch_first=True)
    - num_layers e dropout a seu critério
  - Camada Fully Connected (hidden_size, output_size)

- Forward (Many-to-Many)
  - # Dessa vez não inicializar os pesos (Por que?)
    - Dica: eles estão sendo atualizados em outras partes do código
  - Embedding input: (batch_size, seq_len)
  - GRU input: (batch_size, seq_len, *)
  - Linear: (seq_len, *)
  - * tamanho da saída da camada anterior

# Multilayers em Pytorch


## Gerando letras de Música

**Music.py**