Natural Language Processing

From RNN to Neural Machine Translation (NMT)

Prof. Iacopo Masi and Prof. Stefano Faralli

In [1]:
```
import matplotlib.pyplot as plt
import scipy
import random
import numpy as np
import pandas as pd
pd.set_option('display.colheader_justify', 'center')
```

In [2]:

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
#plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
```

```
                [x[3]],
                angles='xy', scale_units='xy', scale=1, color=cols[i],
                alpha=alpha, linestyle=linestyle, linewidth=2)
```

My own latex definitions

Today's lecture

- Recap on RNN

- Stacked, Bidirectional RNN

- Long short-Term Memory Networks (LSTM)

- Introduction to Self-Attention and Transformers

This lecture material is taken from

📘 **Chapter 9 Jurafsky Book**

📘 **Chapter 6.3 Eisenstein Book**

- Stanford Slide LSTM
- Stanford Lecture LSTM
- Stanford Notes on RNN and LSTM
- Andrej Karpathy Lecture on LSTM
- Andrej Karpathy Slides on LSTM

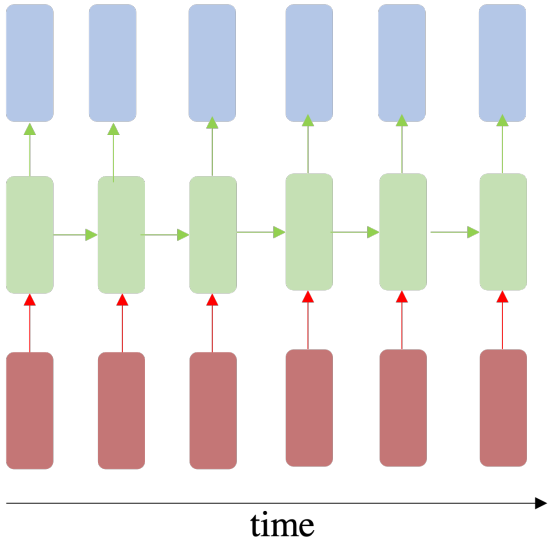
Another resource with code is [d2l.ai] Modern RNN

LSTM: colah.github.io | Illustrated Guide to LSTM

Deep Learning for Sequence Processing

Stacked Bidirectional RNN

RNN so far



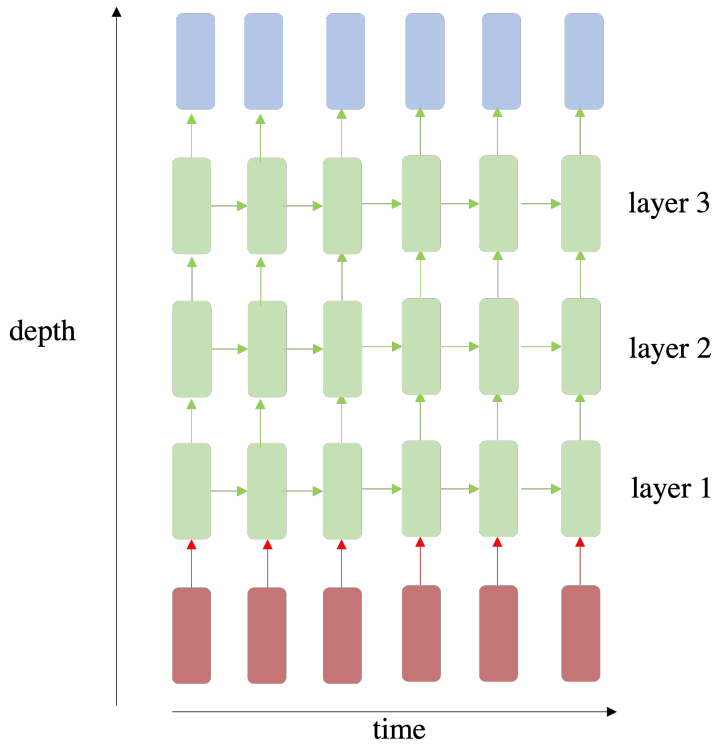
Stacked RNN (or Multi-layer RNN)

- RNNs are already "deep" on one dimension, **the time dimension**---they unroll over many timesteps.
- We can also make them “deep” in **another dimension** (the representation dimension)! We can do so by applying multiple RNNs – this is a multi-layer RNN (stacked RNN).

- This allows the network to compute more complex representations
- The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.

Stacked RNN (or Multi-layer RNN)

The hidden states from RNN layer i are the inputs to RNN layer $i+1$



Stacked RNN (or Multi-layer RNN)

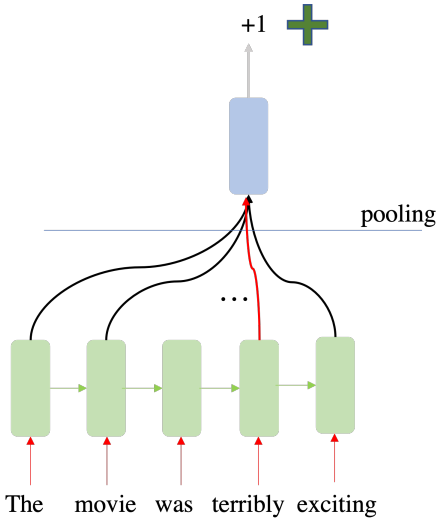
- **Multi-layer or stacked RNNs** allow a network to compute **more complex representations**: they work better than just have one layer of high-dimensional encodings!
- High-performing RNNs are usually multi-layer BUT NOT as deep as convolutional or feed-forward networks

- In a 2017 paper, Britz et al. find that for **Neural Machine Translation**, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN:
  - Often 2 layers is a lot better than 1, and 3 might be a little better than 2
  - Usually, skip-connections/dense-connections are needed to train deeper RNNs (e.g., 8 layers)

Bidirectional RNN: Motivation

Task: Sentiment Classification

We can regard the hidden state $\mathbf{h}_4$ of `terribly` as a representation of the word given prior context this sentence. We call this a *contextual representation*. Note that the sentence denotes `positivity` but `terribly` is usually used as negative word.

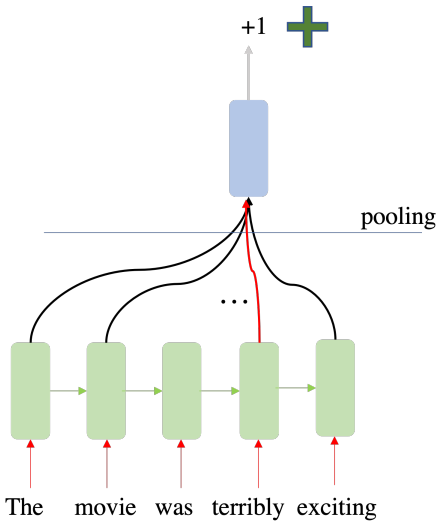


Bidirectional RNN: Motivation

Task: Sentiment Classification

The pooling of all hidden states may cancel then contribution of the hidden state of `exciting` making the model less effective.

It would be nice if we could condition terribly on what comes "in the future" (right)

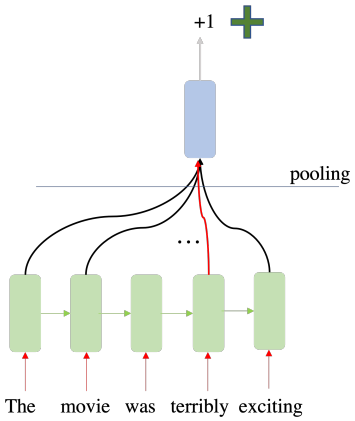


Bidirectional RNN: Motivation

Task: Sentiment Classification

Depending on the task, we have the text at our hand so we can go back at the end of text, we can go right etc.

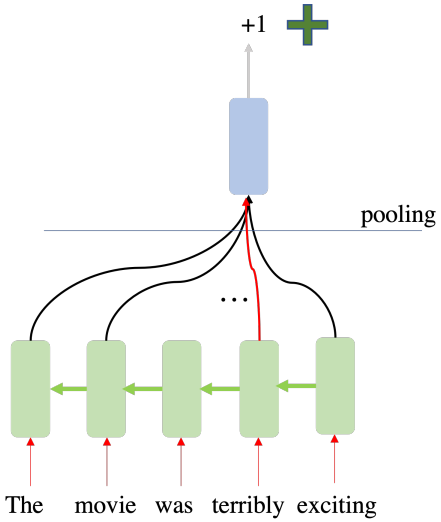
Bottom line: We can move in the text as we want.



Bidirectional RNN: Motivation

Task: Sentiment Classification

Idea: we learn the RNN in reverse order, but what do we do with the previous RNN? 🤔



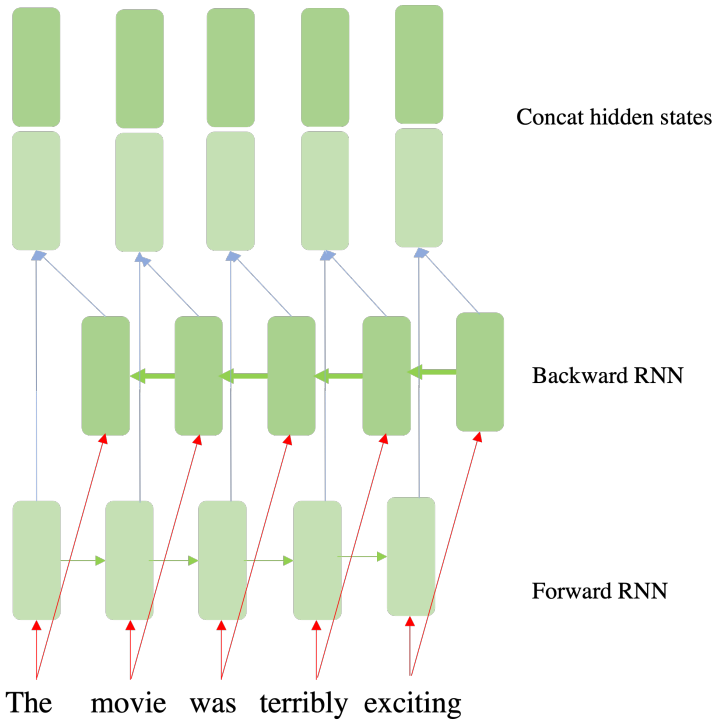
Bidirectional RNN

Task: Sentiment Classification

Idea: We use both of the hidden layers.

1. A set of hidden layers goes from left → right
2. Another set of hidden layers goes left ← right (new)
3. **Fusion:** We somehow "pool" the final representation (usually `concat`).

Bidirectional RNN



Bidirectional RNN

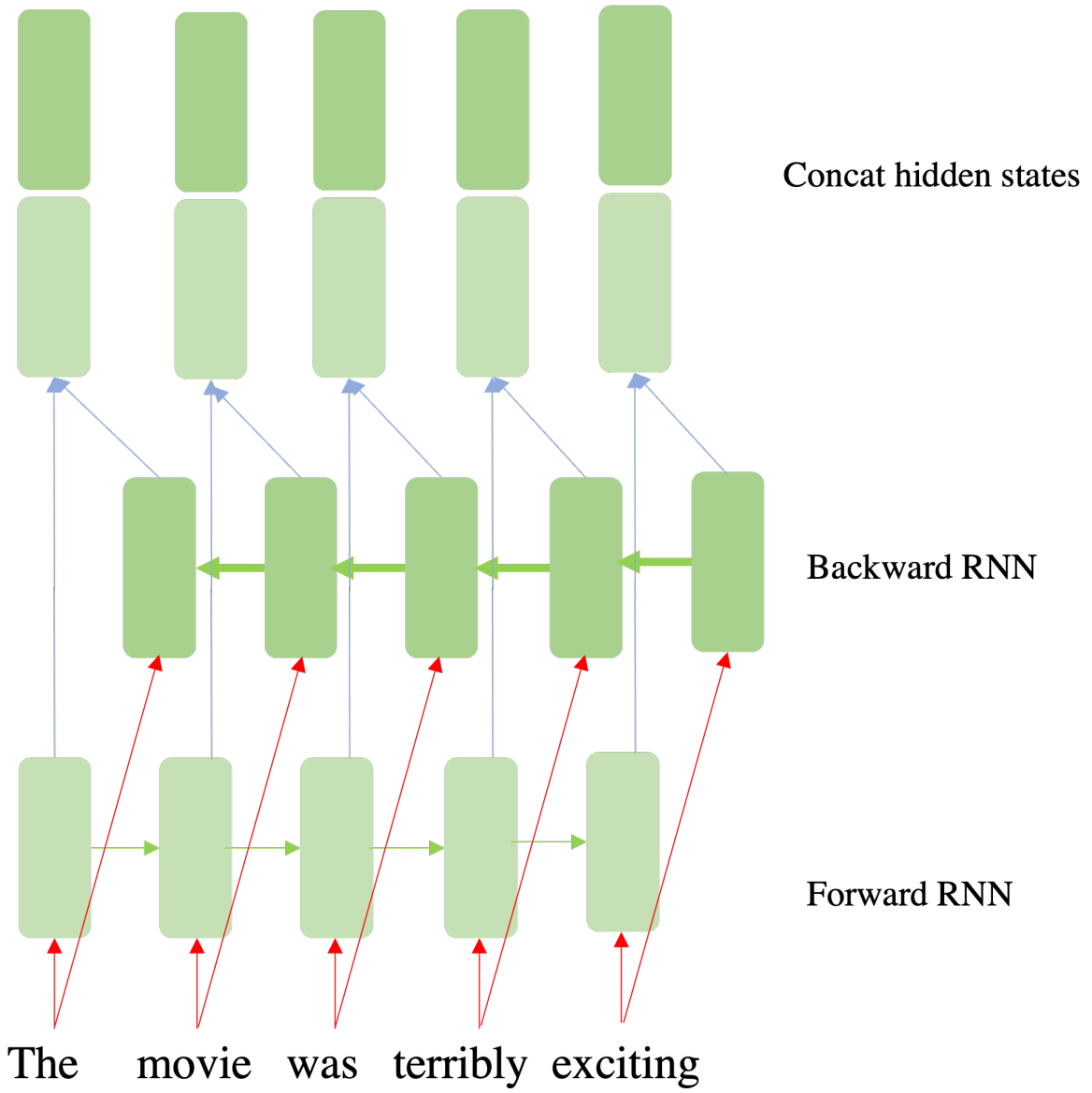
In general for each side of the RNN, you have different weights.

The representation is thus:

$$\overrightarrow{\mathbf{h}}(t) = \mathrm{RNN}_{\mathrm{FW}}\left(\overrightarrow{\mathbf{h}}(t-1), \mathbf{x}(t)\right)$$

$$\overleftarrow{\mathbf{h}}(t) = \mathrm{RNN}_{\mathrm{BW}}\left(\overleftarrow{\mathbf{h}}(t+1), \mathbf{x}(t)\right)$$

$$\mathbf{h}(t) = [\overrightarrow{\mathbf{h}}(t); \overleftarrow{\mathbf{h}}(t)]$$


Concat hidden states
Backward RNN
Forward RNN
The
movie
was
terribly
exciting

Bidirectional RNN

Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**

- They are NOT applicable to Language Modeling, because in LM you only have left context available.
- If you do have entire input sequence (e.g., for any kind of encoding), **bidirectionality is powerful** *(you should use it by default!)*.

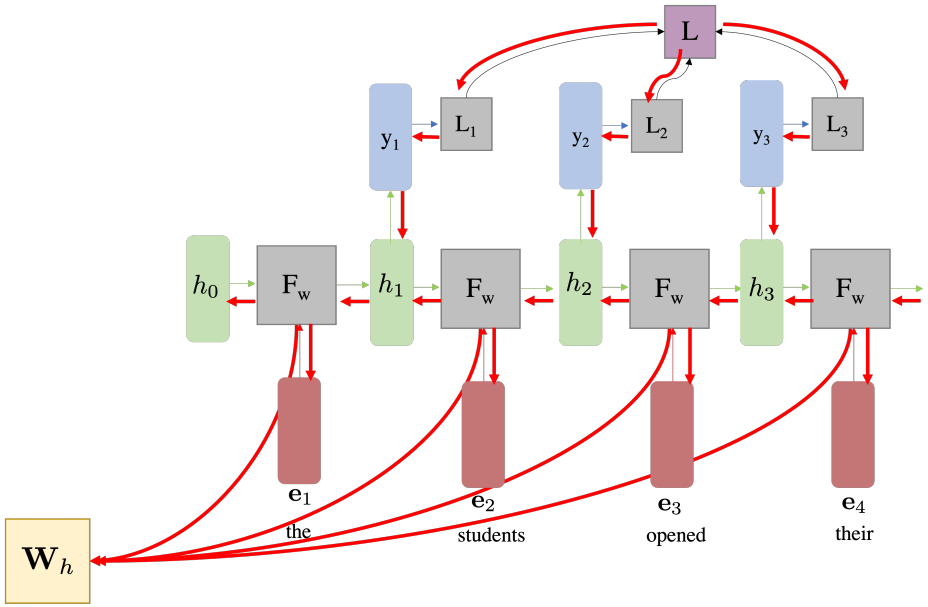
- For example, **BERT (Bidirectional Encoder Representations from Transformers)** is a powerful pretrained contextual representation system built on **bidirectionality**.
- You will learn more about transformers, including BERT, soon.

From simple RNN to LSTM

Remember Back Propagation Through Time (BPTT)?

Back Propagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathcal{L}}}_{1} \underbrace{\frac{\partial \mathcal{L}}{\partial \mathcal{L}_3}}_{1/T=1/3} \underbrace{\frac{\partial \mathcal{L}_3}{\partial \mathbf{y}_3}}_{\mathbb{R}^{1\times|V|}} \underbrace{\frac{\partial \mathbf{y}_3}{\partial \mathbf{h}_3}}_{\mathbb{R}^{|V|\times h}} \underbrace{\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}}_{\mathbb{R}^h \times (\mathbb{R}^h \times \mathbb{R}^h)}$$



Why LSTM? Vanishing Gradient Problem

If the recursive chain is too long, the produce of matrices $\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\dots$ can make the **gradient to vanish** especially if using tanh activation function.

Think as multiply $a \cdot b \cdot b \cdot b \cdot b \cdot b \dots$ where $0 < b < 1$ but for matrices.

At the end the norm of the gradient will be so small that will get to zero numerically!

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h}\Big|_{\text{all time steps}} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_h}$$

RNN to model Sequences

✅ Improvements over window-based LM

- Can process any length input
- Computation for step t can **(in theory)** use information from many steps back
- Bounded model size respect to context size
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

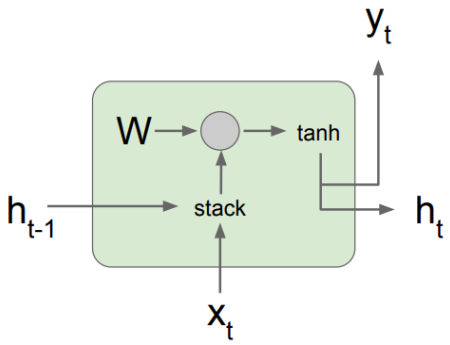
RNN to model Sequences

❌ Remaining problems:

- Recurrent computation is slow (unroll in time)
- In practice, **difficult to access information from many steps back**

RNN Gradient Flow

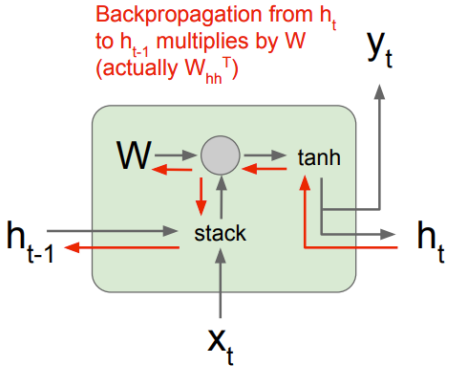
RNN Gradient Flow



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Picture from Stanford

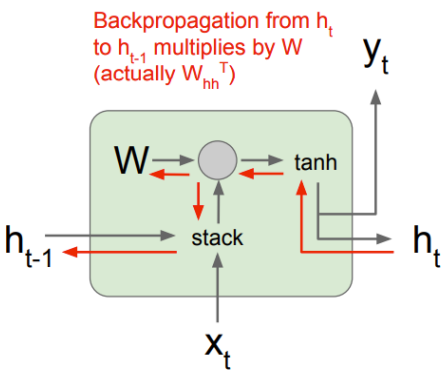
RNN Gradient Flow



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Picture from Stanford

RNN Gradient Flow

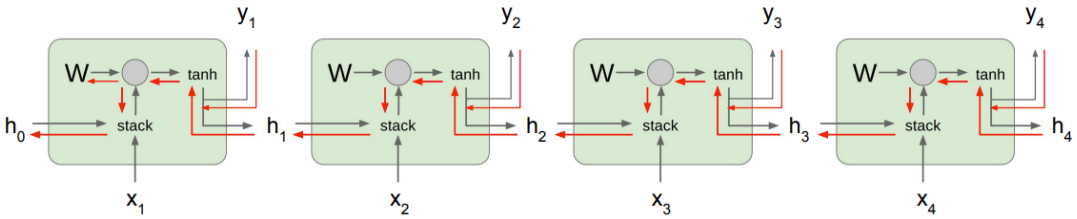


$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

$$
\frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}
$$

Picture from Stanford

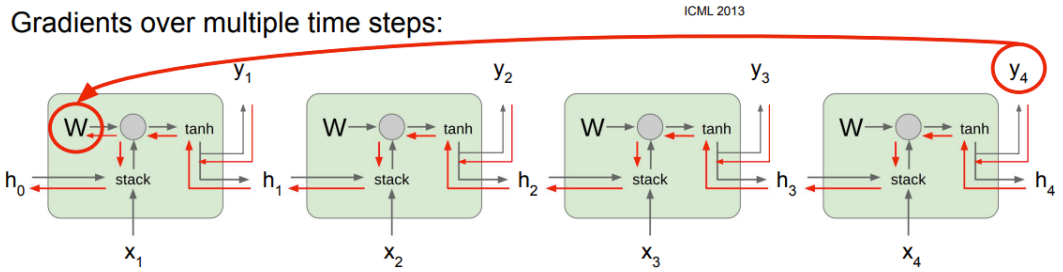
RNN Gradient Flow



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

Picture from Stanford

RNN Gradient Flow

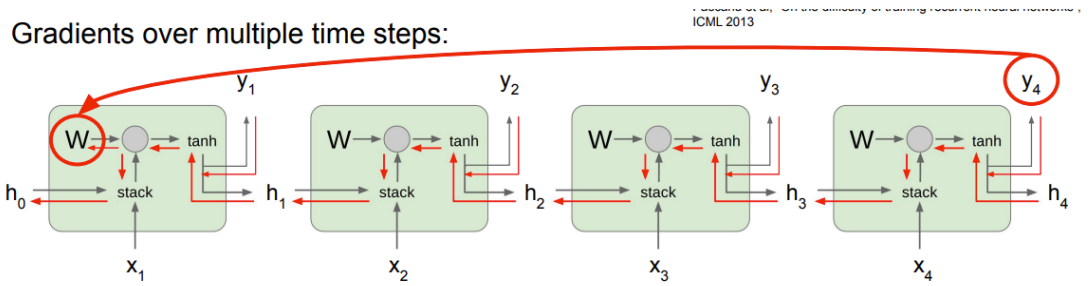


$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W}$$

Picture from Stanford

RNN Gradient Flow

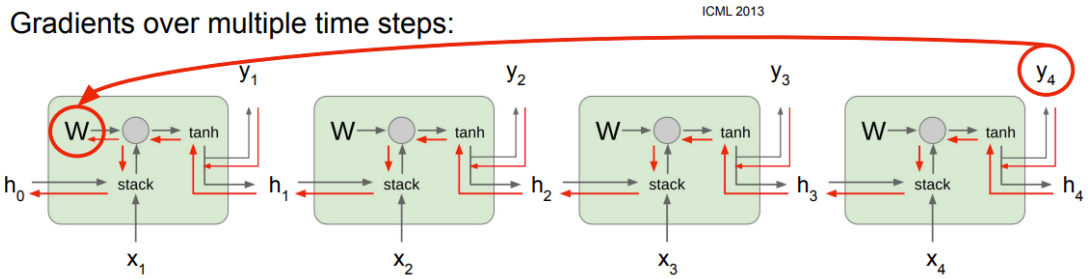


$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

Picture from Stanford

RNN Gradient Flow

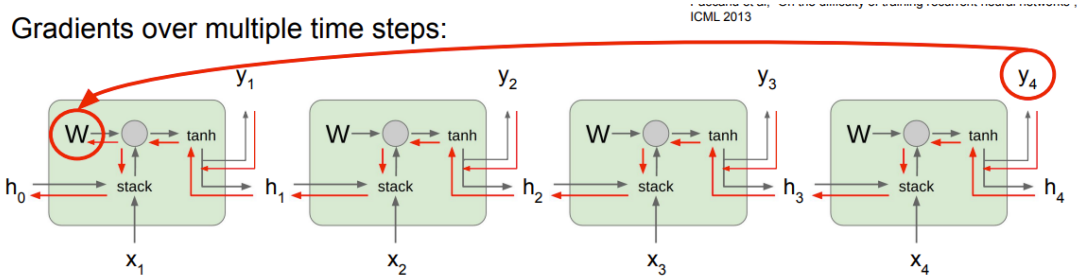


$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} \qquad \frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh} h_{t-1} + W_{xh} x_t) W_{hh}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

Picture from Stanford

RNN Gradient Flow



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

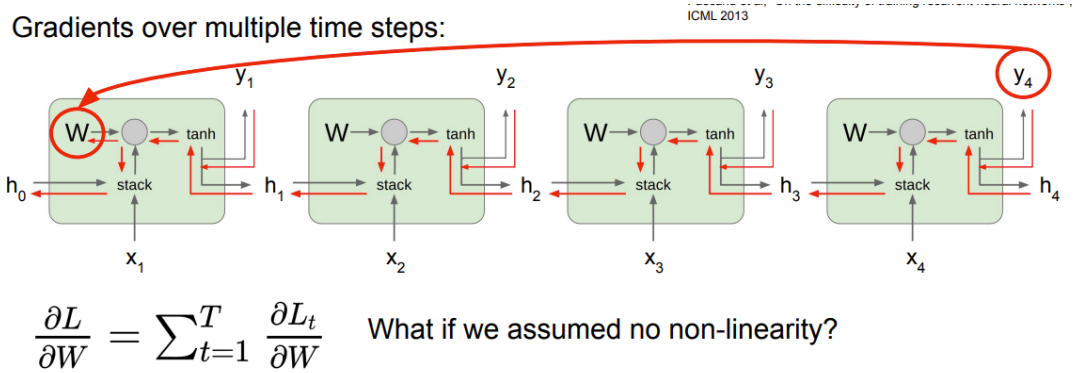
Almost always < 1

Vanishing gradients

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} tanh'\left(W_{hh}h_{t-1} + W_{xh}x_t\right)\right) W_{hh}^{T-1}\frac{\partial h_1}{\partial W}$$

Picture from Stanford

RNN Gradient Flow

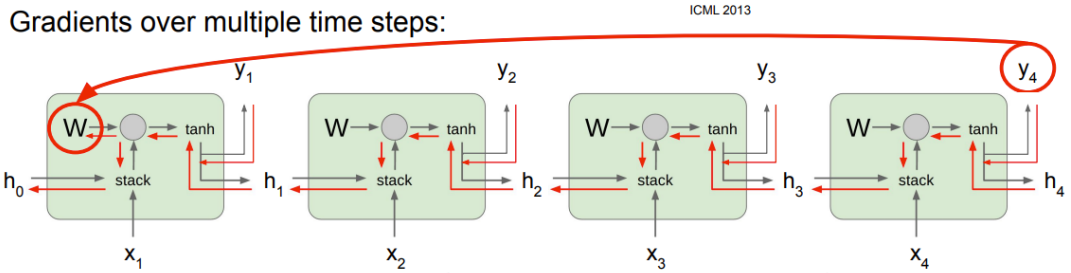


$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

What if we assumed no non-linearity?

Picture from Stanford

RNN Gradient Flow



What if we assumed no non-linearity?

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

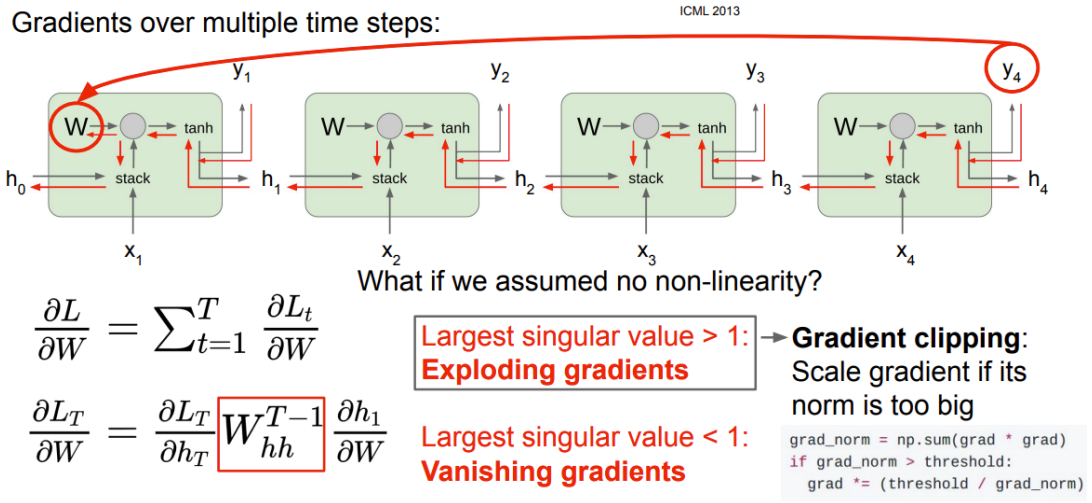
$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

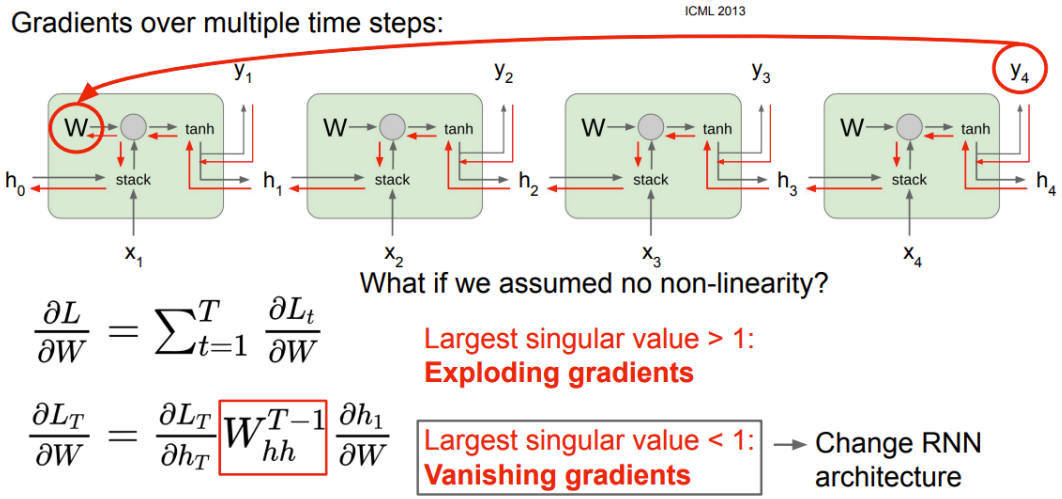
Picture from Stanford

RNN Gradient Flow



Picture from Stanford

RNN Gradient Flow

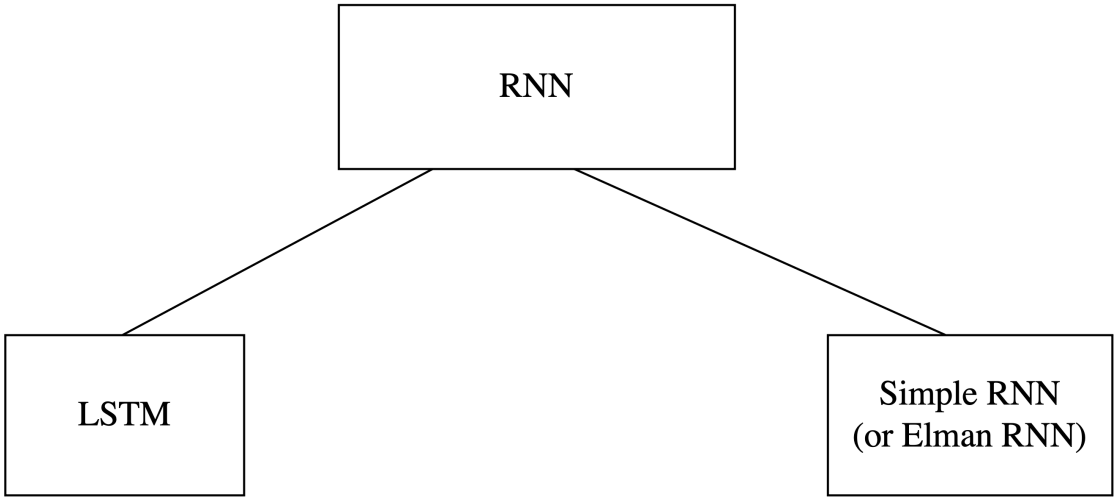


Picture from Stanford

Change RNN architecture!

from simple RNN to Long Short-Term Memory RNN

RNN Terminology

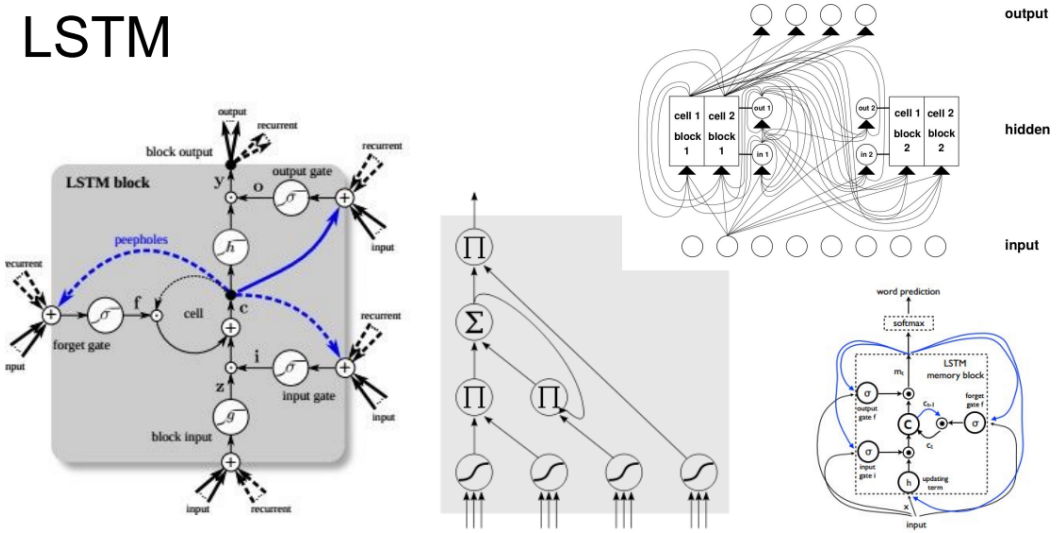


Long Short-Term Memory RNN (LSTM)

We have to modify the **Elman Unit** in something that can mitigate back-propagation vanishing gradients when the recurrence is **"plain"**:

$$\mathbf{h}_t = \tanh\left(\mathbf{W}\begin{pmatrix}\mathbf{h}_{t-1}\\ \mathbf{x}_t\end{pmatrix}\right)$$

LSTM Madness 😱



LSTM Motivation

- While gradient clipping helps with exploding gradients, handling **vanishing gradients appears to require a more elaborate solution.**

- LSTMs: each ordinary recurrent node is replaced by a **memory cell**. Each memory cell contains an **internal state** $\mathbf{c}_t$, i.e., a node with a self-connected recurrent edge of fixed weight 1
- This ensures that the gradient can pass across many time steps without vanishing or exploding (LONG dependency).

Why the name LSTM?

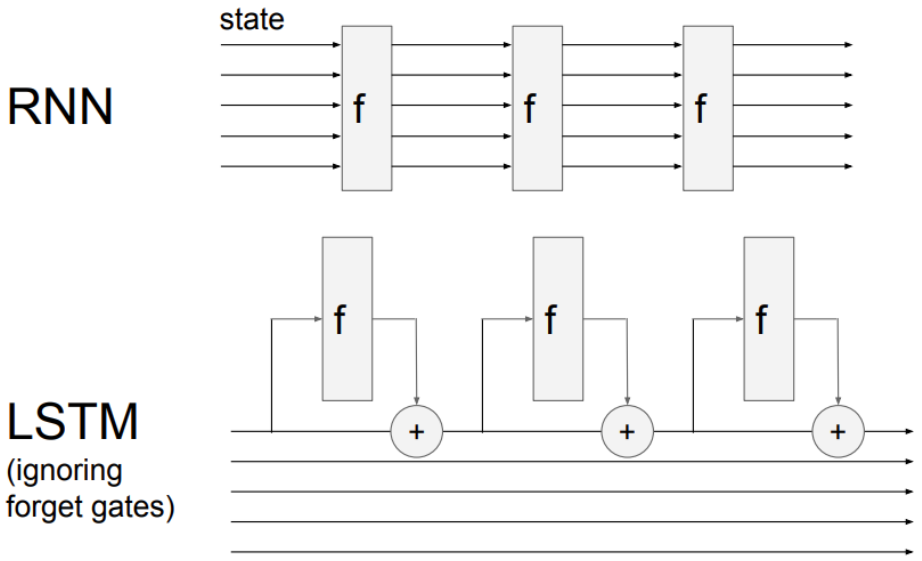
The term "long short-term memory" comes from the following intuition:

- Simple RNN have **long-term memory** in the form of weights. The weights change slowly during training, encoding general knowledge about the data.
- They also have **short-short term memory** in the form of ephemeral activations, which pass from each node to successive nodes.
- The LSTM model introduces an intermediate type of storage via **the memory cell**. **A memory cell is a composite unit, built from simpler nodes in a specific connectivity pattern, with the novel inclusion of multiplicative nodes.**

Best way to understand LSTM: a conveyor

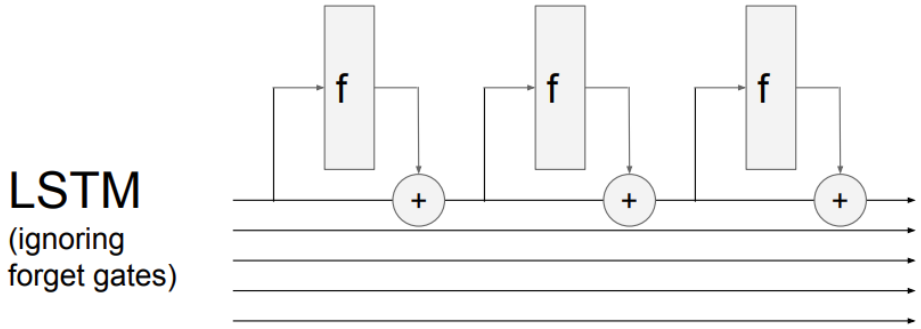
RNN: compute and recurr

LSTM: compute, *let me see what I have to add/forget from the memory*, then recurr

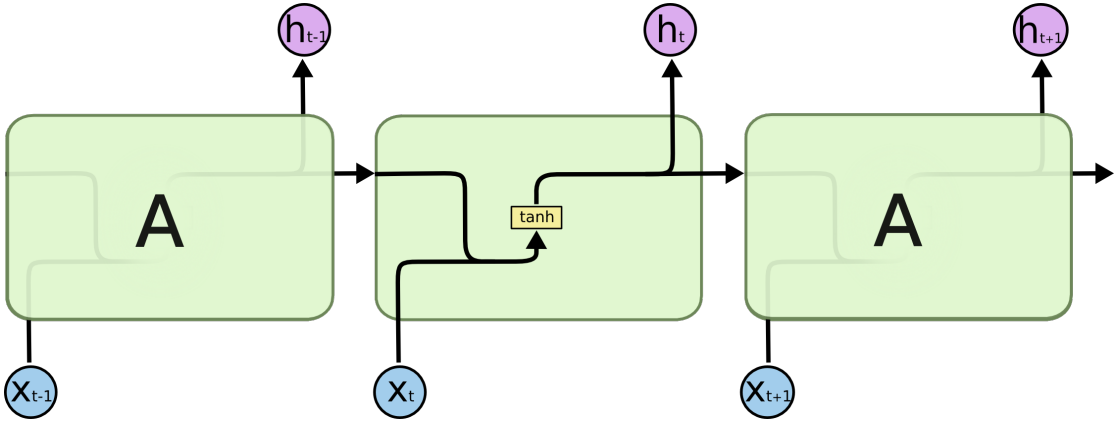


Best way to understand LSTM: a conveyor

LSTM: have these highways (bottom lines) that forward information from the past



From RNN to LSTM Units



Taken from [colah.github.io](https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Breaking down LSTM

The hidden states of the LSTM are now 2:

1. the usual hidden state $\mathbf{h}_t$ like simple RNN
2. another state $\mathbf{c}_t$ which is called **memory cell state vector** $\leftarrow$ this is the highway!

LSTM Unit Input

At each time step t, we have as input:

- $\mathbf{x}_t$ (input from data)
- $\mathbf{c}_{t-1}$ (previous memory state)
- $\mathbf{h}_{t-1}$ (previous hidden state)

LSTM Unit Output

At each time step t, we have to compute:

- $\mathbf{c}_t$ (next memory state)
- $\mathbf{h}_t$ (next hidden state)

Breaking down LSTM

Given some **gates** ("controls"), we want to compute the next state:

$$\mathbf{c}_t = \text{function}\left(\mathbf{c}_{t-1}, \text{function}(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}); \mathbf{gates}\right)$$
$$\mathbf{h}_t = \text{function}\left(\mathbf{c}_t; \mathbf{gates}\right)$$

What are the "controls"? They are Gates!

Gates ("controls") change the inductive bias of the architecture, allowing the network to control the information that flows.

$$\mathbf{gates} = \text{function}\left(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}\right)$$
$$\mathbf{c}_t = \text{function}\left(\mathbf{c}_{t-1}, \text{function}(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}); \mathbf{gates}\right)$$
$$\mathbf{h}_t = \text{function}\left(\mathbf{c}_t; \mathbf{gates}\right)$$

$$\mathbf{gates} = \text{function}\left(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}\right)$$

What are the "controls"? They are Gates!

The LSTM "controls" the information in the memory (as a sort of RAM memory) using various **gates**. Think **gates** as actions performed on the memory:

- **F**orget about the past in the cell memory state $\longrightarrow$ **soft binary gate**
- **I**nput new information in the cell memory state $\longrightarrow$ **soft binary gate**
- **O**utput information to retain in the next hidden state $\longrightarrow$ **soft binary gate**

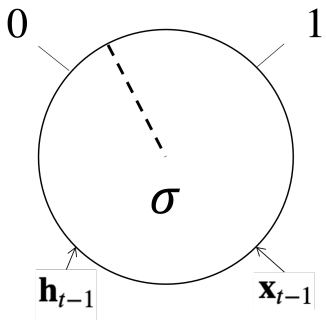
Given that we are in 🇮🇹, to recall just remember **FIO**. 🍊

$$\mathbf{gates} = \text{function}\left(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}\right)$$

Gates

Gates end with **sigmoid** and act as a **soft binary mask** to control how much information should flow.

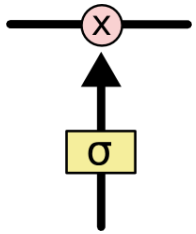
Gates are function of $\left(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}\right)$



Gates

Gates end with sigmoid and act as a **soft binary mask** to control how much information should flow.

Gates are function of $\left(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}\right)$



Gates

A few details and terminology:

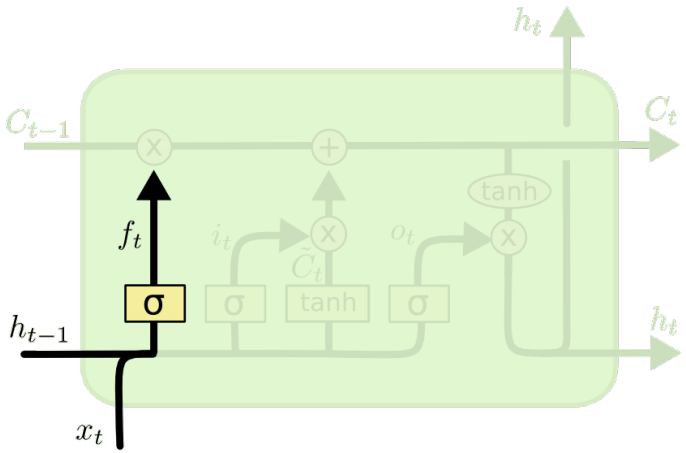
- σ is the sigmoid function---the one you find also in logistic regression LR). Remember that σ gives you an output in $[0...1]$.
  - you can think σ as a probability as in LR or, as in here, as **soft-masking bounded [0,1]**
  - σ tells you how much information you want to make flow.
- $\odot$ is the **Hadamard product** which is element-wise multiplication of tensors.

$$\begin{pmatrix} i \\ f \\ o \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix}$$

Gates

$$\begin{pmatrix} \text{input} \\ \text{forget} \\ \text{output} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix} + \mathbf{b}$$

Forget Gate



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

Gates: alternative equation still same concept

Sometimes you can find it written as below but is more complex. In our case $\mathbf{W} = [\mathbf{W}_{xi}; \mathbf{W}_{hi}; \mathbf{W}_{xf}; \mathbf{W}_{hf}; \mathbf{W}_{xo}; \mathbf{W}_{ho};]$ concat of matrices, same for biases.

$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned}$$

Candidate update for $\mathbf{c}_t$: $\tilde{\mathbf{c}}_t$

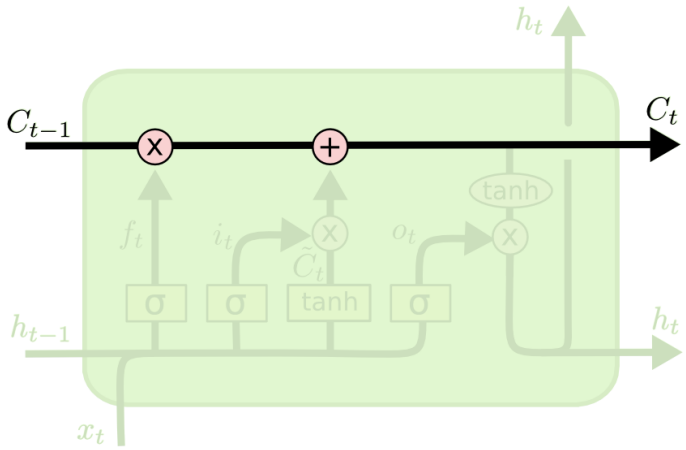
We can compute it in the similar way we do for **gates** but still with its own parameters

$$\mathbf{c}_t = \text{function}\left(\mathbf{c}_{t-1}, \underbrace{\text{function}(\mathbf{h}_{t-1}, \mathbf{x}_{t-1})}_{\text{candidate update } \tilde{\mathbf{c}}_t}; \; \underbrace{\textbf{gates}}_{\text{you know this!}}\right)$$

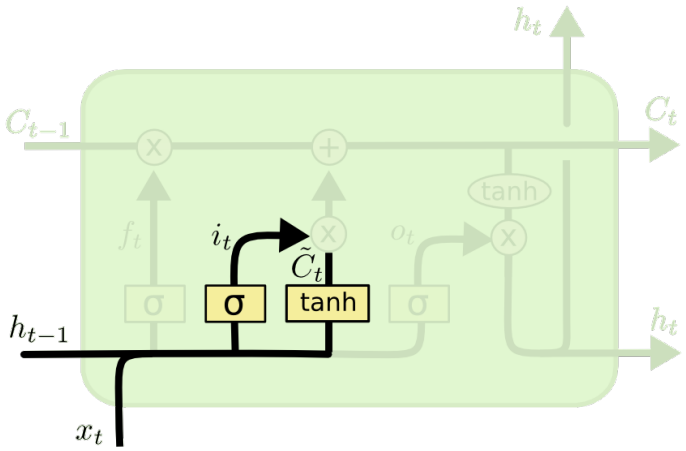
Gates + Candidate update $\tilde{\mathbf{c}}_t$

$$\begin{pmatrix} \text{input} \\ \text{forget} \\ \text{output} \\ \tilde{\mathbf{c}}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix} + \mathbf{b}$$

Update the memory cell state: forget or add input



Update the memory cell state: forget or add input



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

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

Parametric model for LSTM Unit

The parametric model of an LSTM comprises of a matrix $\mathbf{W}$ of dimensionality $4D \times 2D$ where D is the dimension of the hidden state $\mathbf{h}$.

- $2D$ because we take as input both $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$
- $4D$ because we output 4 vector of dimensionality D: forget, input, output and candidate update $\tilde{\mathbf{c}}_t$

The bias is optional and is $4D$.

This assume dim($\mathbf{h}$) = dim($\mathbf{x}$) but they could differ.

$$\mathbf{c}_t = \text{function}\left(\mathbf{c}_{t-1}, \text{function}(\mathbf{h}_{t-1}, \mathbf{x}_{t-1}); \mathbf{controls}\right)$$

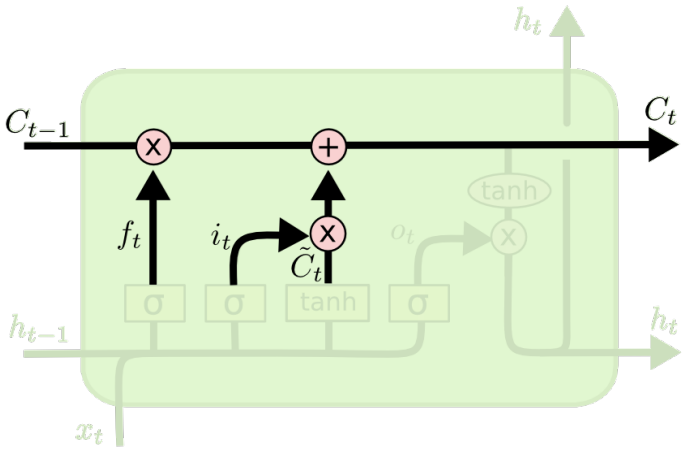
Update the memory cell state

$$\mathbf{c}_t = f \odot \mathbf{c}_{t-1} + i \odot \tilde{\mathbf{c}}_t$$

Update the memory cell state

$$\mathbf{c}_t = \underbrace{f \odot \mathbf{c}_{t-1}}_{\text{forget the past}} + \underbrace{i \odot \tilde{\mathbf{c}}_t}_{\text{input new information}}$$

Update the memory cell state: forget or add input

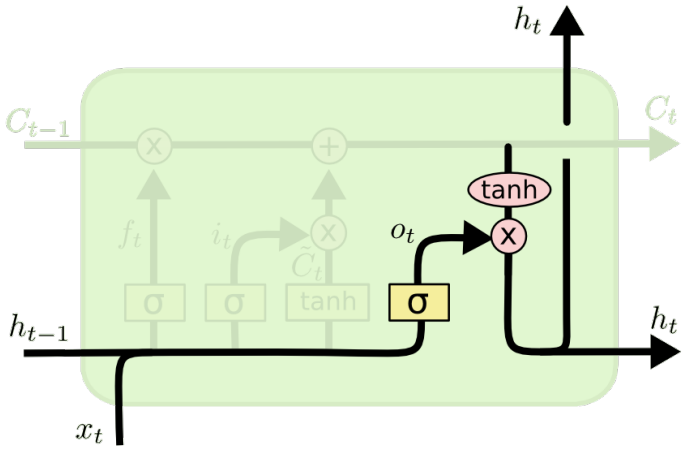


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Update the hidden state

$$\mathbf{c}_t = f \odot \mathbf{c}_{t-1} + i \odot \tilde{\mathbf{c}}_t$$
$$\mathbf{h}_t = o \odot \tanh(\mathbf{c}_t)$$

Update the hidden state

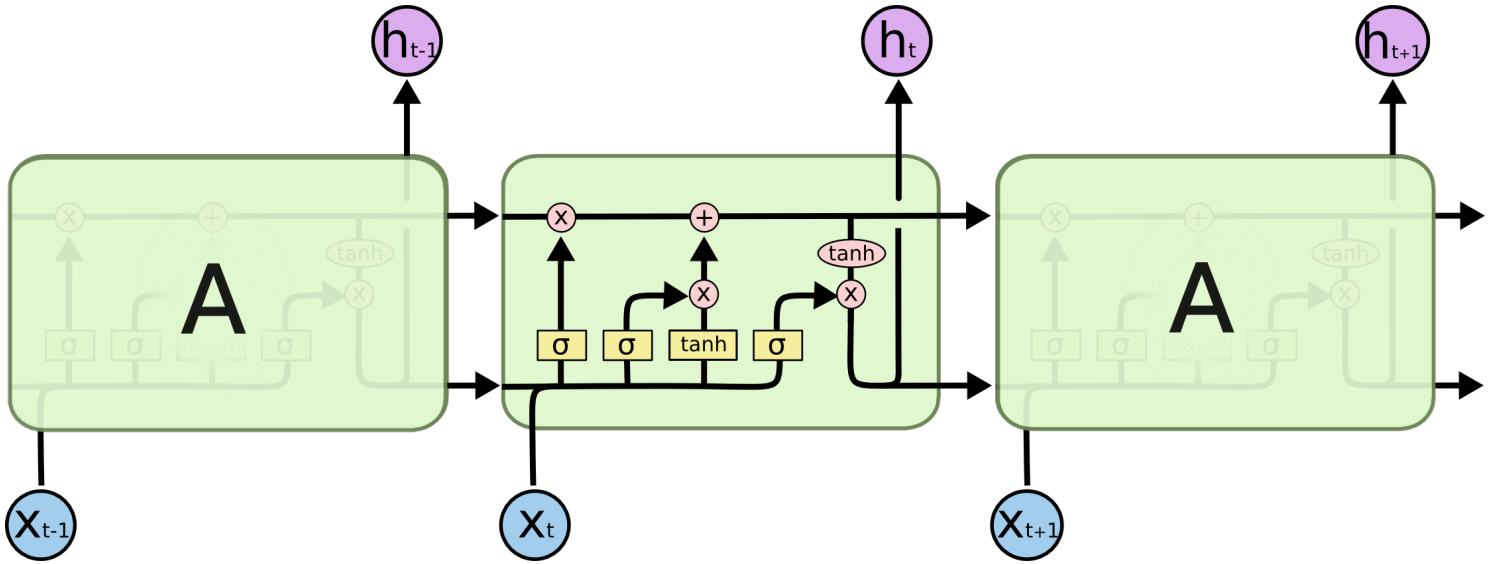


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Putting all together

$$\begin{pmatrix} i \\ f \\ o \\ \tilde{\mathbf{c}}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix} + \mathbf{b}$$
$$\mathbf{c}_t = f \odot \mathbf{c}_{t-1} + i \odot \tilde{\mathbf{c}}_t$$
$$\mathbf{h}_t = o \odot \tanh(\mathbf{c}_t)$$

Putting all together



Simple RNN vs LSTM

$$\mathbf{h}_t = \tanh\left(\mathbf{W}\begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix}\right)$$

$$\begin{pmatrix} i \\ f \\ o \\ \tilde{\mathbf{c}} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix} + \mathbf{b} \tag{1}$$
$$\mathbf{c}_t = f \odot \mathbf{c}_{t-1} + i \odot \tilde{\mathbf{c}}$$
$$\mathbf{h}_t = o \odot \tanh(\mathbf{c}_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Gers, Schmidhuber, and Cummins, 2000. Learning to Forget: Continual Prediction with LSTM

RNN vs LSTM Gradients

```
H = 5  # dim of hidden staate
T = 50 # steps of unrolling the RNN
Whh = np.random.randn(H, H)
hs, ss = {}, {}
hs[-1] = np.random.randn(H)
# forward pass of the RNN ignoring input
for t in range(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0,ss[t])
# backward pass of the RNN
dhs, dss = {}, {}
dhs[T-1] = np.random.randn(H) # we inject random gradients (similar to having a loss)
for t in reversed(range(T)):
    dss[t]=(hs[t-1] > 0)*dhs[t]  #backprop through the non-linearity
    dhs[t-1] = np.dot(Whh.T,dss[t]) #backprop into previous hidden state
```

Code by A. Karpathy

RNN vs LSTM Gradients

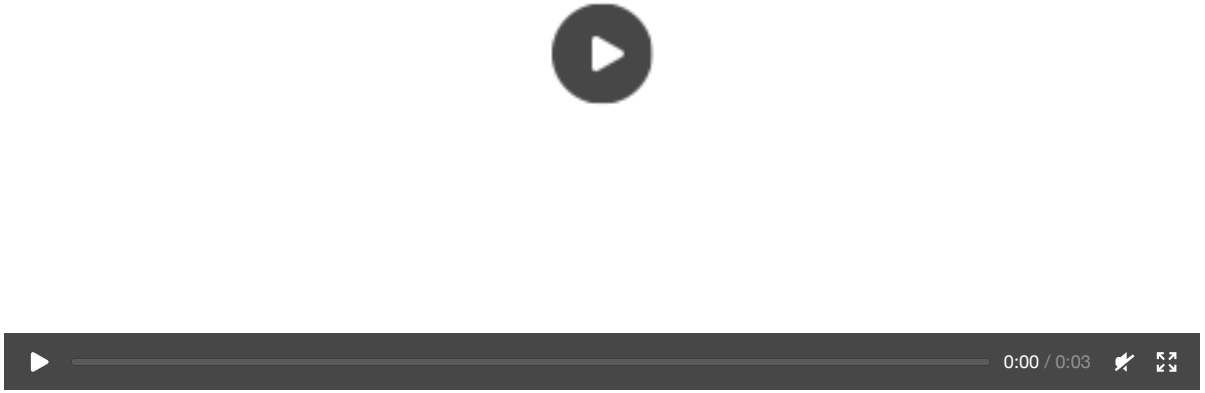
```
dhs[0]= np.dot(Whh.T, dss[1]) # where dss[1]=(hs[0] > 0)*dhs[1]

dhs[0]= np.dot(Whh.T, (hs[0] > 0)*dhs[1]) # where dhs[1]=np.dot(Whh.T, (hs[1] > 0)*dhs[2])

dhs[0]= np.dot(Whh.T, (hs[0] > 0)*np.dot(Whh.T, (hs[1] > 0)*dhs[2]))
```

RNN vs LSTM Gradients

RNN vs LSTM gradients on the input weight matrix

Error is generated at 128th step and propagated back. No error from other steps. At the beginning of training. Weights sampled from Normal Distribution in (-0.1, 0.1).

[Taken from http://imgur.com/gallery/vaNahKE](http://imgur.com/gallery/vaNahKE)



Other RNN Variants

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$
$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

[*LSTM: A Search Space Odyssey*, Greff et al., 2015]

[*An Empirical Exploration of Recurrent Network Architectures*, Jozefowicz et al., 2015]

MUT1:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + b_z) \\ r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + \tanh(x_t) + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

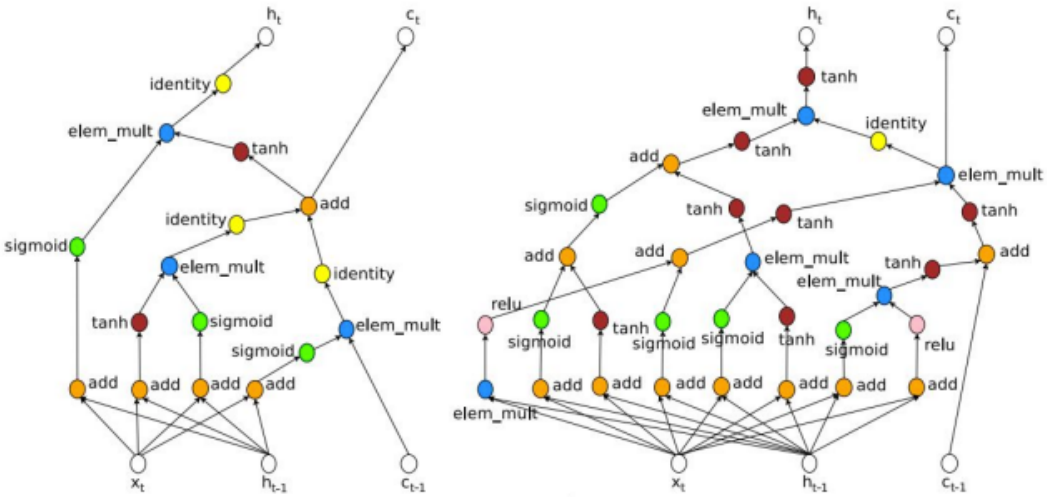
MUT2:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + W_{hz}h_t + b_z) \\ r &= \text{sigm}(x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + W_{hz}\tanh(h_t) + b_z) \\ r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

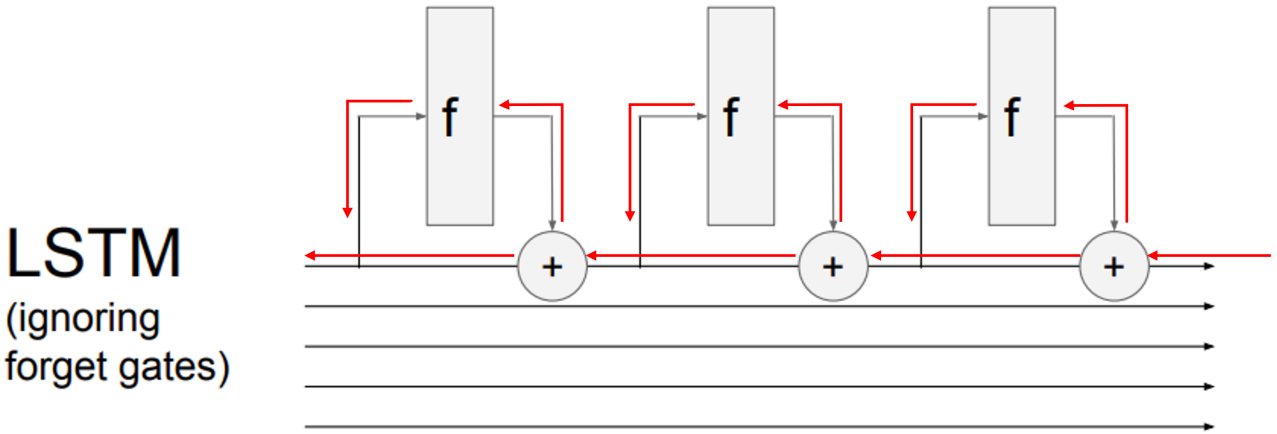
LSTM and Neural Architectural Search



LSTM cell

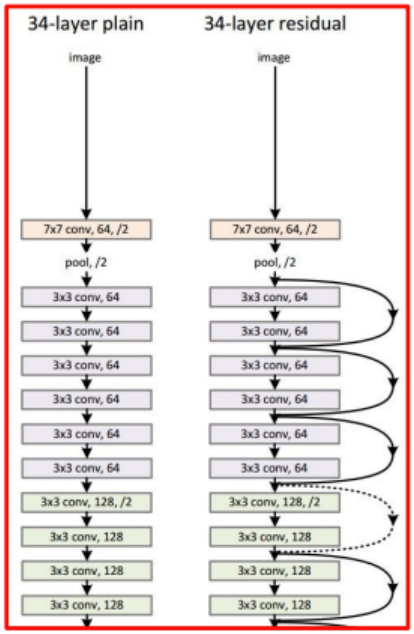
Cell they found

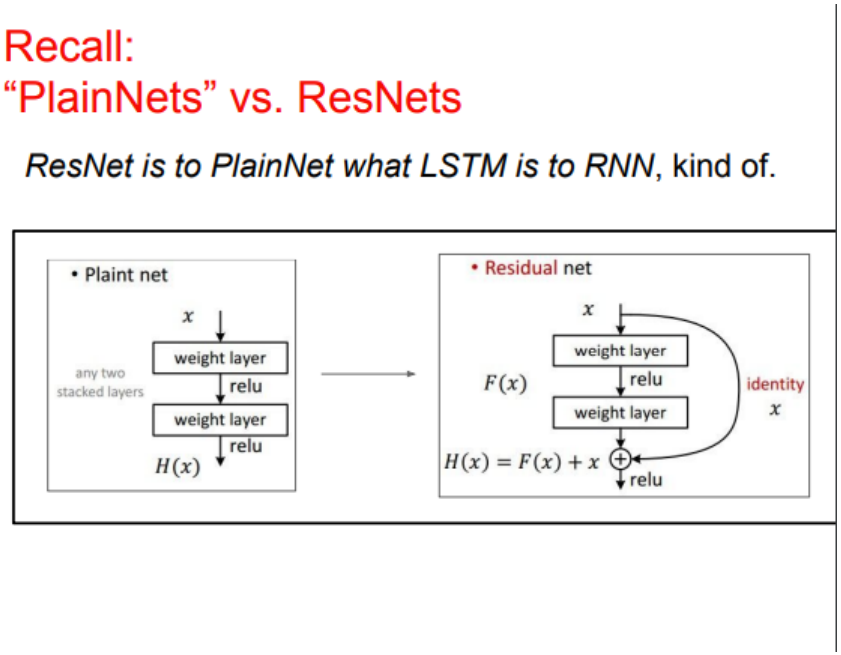
LSTM Grad Flow



Does this configuration recall you of another famous architecture?

LSTM vs ResNet





How does LSTM solve vanishing gradients?

- The LSTM architecture makes it **much easier** for an RNN to **preserve information over many timesteps**
  - e.g., if the **forget gate is set to 1** for a cell dimension and **the input gate set to 0**, then the **information of that cell is preserved indefinitely**.
  - In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix Wh that preserves info in the hidden state
  - In practice, you get about **100 timesteps rather than about 7**
- LSTMs do not guarantee that there is no vanishing/exploding gradients but they do provide an easier way for the model to learn long-distance dependencies.

LSTM: Real-world success

- In 2013–2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include **handwriting recognition, speech recognition, machine translation, parsing, and image captioning**, as well as language models
  - LSTMs became the dominant approach for most NLP tasks

Source: "Findings of the 2016 Conference on Machine Translation (WMT16)", Bojar et al. 2016, http://www.statmt.org/wmt16/pdf/W16-2301.pdf

Source: "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, http://www.statmt.org/wmt18/pdf/WMT028.pdf

Source: "Findings of the 2019 Conference on Machine Translation (WMT19)", Barrault et al. 2019, http://www.statmt.org/wmt18/pdf/WMT028.pdf

Situation now

- Now (2019–2023), **Transformers** have become dominant for all tasks
  - For example, in WMT (a Machine Translation conference + competition):
  - In WMT **2014, there were 0 neural machine translation systems (!)**
  - In WMT **2016**, the summary report contains **"RNN" 44 times** (and these systems won)
  - In WMT **2019 "RNN" 7 times**, **"Transformer" 105 times**

Practical Application with LSTM

Tutorial taken from https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html

Implementing LSTM from scratch with Pytorch

```
class LSTMScratch(nn.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        init_weight = lambda *shape: nn.Parameter(torch.randn(*shape) * sigma)
        triple = lambda: (init_weight(num_inputs, num_hiddens),
                          init_weight(num_hiddens, num_hiddens),
                          nn.Parameter(torch.zeros(num_hiddens)))
        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

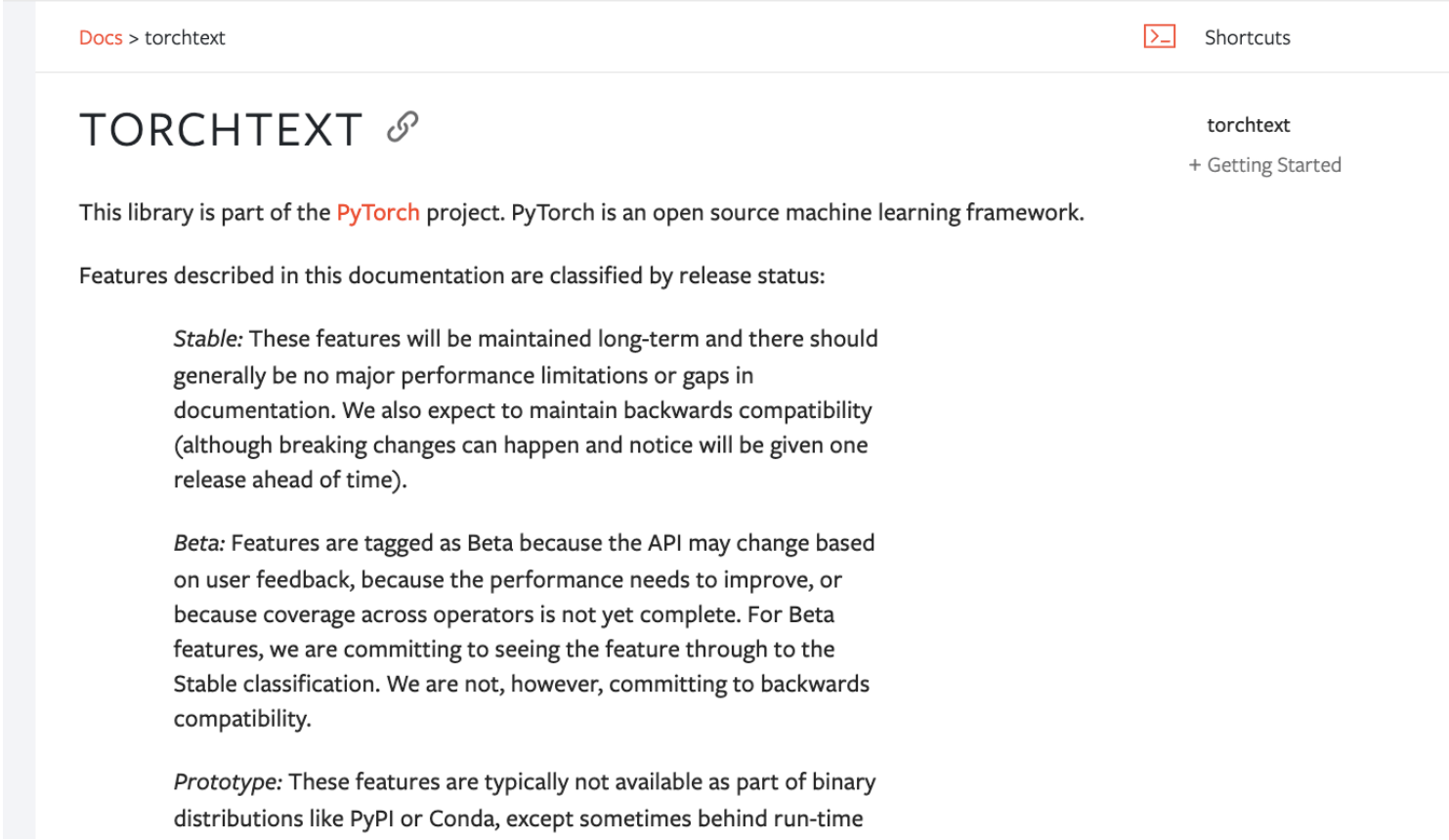
Implementing LSTM from scratch with Pytorch

```
def forward(self, inputs, H_C=None):
    # inputs is #seq_size x dimension
    if H_C is None:
        # Initial state with shape: (batch_size, num_hiddens)
        H = torch.zeros((inputs.shape[1], self.num_hiddens),
                        device=inputs.device)
        C = torch.zeros((inputs.shape[1], self.num_hiddens),
                        device=inputs.device)
    else:
        H, C = H_C
    outputs = []
    for X in inputs:
        I = torch.sigmoid(torch.matmul(X, self.W_xi) +
                          torch.matmul(H, self.W_hi) + self.b_i)
        F = torch.sigmoid(torch.matmul(X, self.W_xf) +
                          torch.matmul(H, self.W_hf) + self.b_f)
        O = torch.sigmoid(torch.matmul(X, self.W_xo) +
                          torch.matmul(H, self.W_ho) + self.b_o)
        C_tilde = torch.tanh(torch.matmul(X, self.W_xc) +
                             torch.matmul(H, self.W_hc) + self.b_c)
        C = F * C + I * C_tilde
        H = O * torch.tanh(C)
        outputs.append(H)
    return outputs, (H, C)
```

Implementing LSTM using Pytorch

- As previously, the hyperparameter `num_hiddens` dictates the number of hidden units.
- We initialize weights following a Gaussian distribution with 0.01 standard deviation, and we set the biases to 0.

Torch Text



Docs > torchtext

Shortcuts

TORCHTEXT

torchtext

+ Getting Started

This library is part of the PyTorch project. PyTorch is an open source machine learning framework.

Features described in this documentation are classified by release status:

Stable: These features will be maintained long-term and there should generally be no major performance limitations or gaps in documentation. We also expect to maintain backwards compatibility (although breaking changes can happen and notice will be given one release ahead of time).

Beta: Features are tagged as Beta because the API may change based on user feedback, because the performance needs to improve, or because coverage across operators is not yet complete. For Beta features, we are committing to seeing the feature through to the Stable classification. We are not, however, committing to backwards compatibility.

Prototype: These features are typically not available as part of binary distributions like PyPI or Conda, except sometimes behind run-time

LSTM Pytorch API

LSTM

CLASS torch.nn.LSTM(**args*, ***kwargs*) [SOURCE]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, h_{t-1} is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

LSTM Pytorch API

Inputs: input, (h_0, c_0)

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

where:

$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{cell} &= \text{hidden\_size} \\
H_{out} &= \text{proj\_size if proj\_size} > 0 \text{ otherwise hidden\_size}
\end{aligned}
$$

LSTM Pytorch API

Outputs: output, (h_n, c_n)

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the LSTM, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence. When `bidirectional=True`, *output* will contain a concatenation of the forward and reverse hidden states at each time step in the sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the sequence. When `bidirectional=True`, *h_n* will contain a concatenation of the final forward and reverse hidden states, respectively.
- **c_n**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the final cell state for each element in the sequence. When `bidirectional=True`, *c_n* will contain a concatenation of the final forward and reverse cell states, respectively.

LSTM Pytorch API

Parameters:

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

In [3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

Out[3]:
```
<torch._C.Generator at 0x7f89f142b390>
```

In [4]:
```
# we define an LSTM model with
# input_size a 10D vector
# hidden_size a 20D vector
# num_layers means 2 stacked LSTM
lstm = nn.LSTM(input_size=10, hidden_size=20, num_layers=2)
```

In [5]:
```
# we define an LSTM model with
# input_size a 3D vector
# hidden_size a 3D vector
# num_layers means 2 stacked LSTM
input_size, hidden_size, batch_size, = 3, 2, 1
lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size, num_layers=1)
```

Let us make up a single training sequence

This is very useful in deep learning to see if your model digests input correctly

Very common to test the network on random data

In [6]:
```
inputs = [torch.randn(1, input_size) for _ in range(5)]  # make a sequence of length 5
```

Now we will do "forward pass" manually

In [7]:
```
# initialize the hidden state.
hidden = (torch.randn(1, batch_size, hidden_size),
          torch.randn(1, batch_size, hidden_size))
for i in inputs: # loop over sequences
    # Step through the sequence one element at a time.
    # after each step, hidden contains the hidden state.
    out, hidden = lstm(i.view(1, 1, -1), hidden) # i becomes 1x1x5
```

We can also let Pytorch handle everything

Batch

Remember that in practice the input is batched. As input to the RNN you feed a 3D tensor $\mathbf{X}$ of size:

$$\mathbf{X} = (\text{seq\_len}, \text{batch}, \text{feature})$$

where:

- `seq_len` is the axis to index the sequence
- `batch` is the axis to index which sequence in the batch
- `feature` is the axis to index the feature

Batch

$$\mathbf{X} = (\text{seq\_len}, \text{batch}, \text{feature})$$

$\mathbf{X}[0][0][:]$ means the feature of the **first element** in the **first sequence**

We can do the entire sequence all at once.

1. first value `out` returned by LSTM is all of the hidden states throughout the sequence.
2. the second `hidden` is just the most recent hidden state

Let's make the training data a batch now

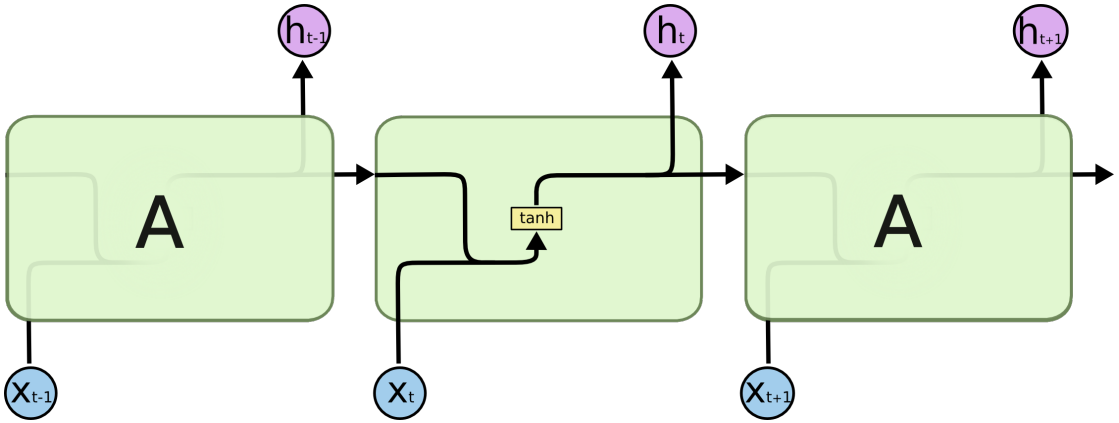
In [8]:
```
# alternatively, we can do the entire sequence all at once.
# the first value returned by LSTM is all of the hidden states throughout
# the sequence. the second is just the most recent hidden state
# (compare the last slice of "out" with "hidden" below, they are the same)
# The reason for this is that:
# "out" will give you access to all hidden states in the sequence
# "hidden" will allow you to continue the sequence and backpropagate,
# by passing it as an argument  to the lstm at a later time

# Add the extra 2nd dimension
inputs_batch = torch.cat(inputs).view(len(inputs), 1, -1)
# torch.cat(inputs) make 5 item lits of 3D vector a 5x3 tensor
# .view(len(inputs), 1, -1) makes the tensor 5x3 a 5x1x3
```

```
inputs_batch = torch.cat(inputs).view(len(inputs), 1, -1)
torch.cat(inputs) # make 5 item lits of 3D vector a 5x3 tensor
.view(len(inputs), 1, -1) #makes the tensor 5x3 a 5x1x3
```

Note we still have a single seq in the batch

Now we forward in the network

LSTM Units



Taken from [colah.github.io](https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

In [9]:
```
hidden = (torch.randn(1, batch_size, hidden_size),
          torch.randn(1, batch_size, hidden_size))  #clean out hidden state
out, hidden = lstm(inputs_batch, hidden)
```

In [10]:
```
print(out)
print(hidden)
```

```
tensor([[[-0.1099, -0.1292]],

        [[ 0.0916, -0.1633]],

        [[ 0.2786, -0.2308]],

        [[ 0.1764, -0.1675]],

        [[ 0.0410, -0.1014]]], grad_fn=<MkldnnRnnLayerBackward0>)
(tensor([[[ 0.0410, -0.1014]]], grad_fn=<StackBackward0>), tensor([[[ 0.0480, -0.5339]]], grad_fn=<StackBackward0>))
```

All hidden states and last hidden state

```
 type(out), type(hidden)
```

{{type(out), type(hidden)}}

All hidden states `out` shape

`out.shape`

{{out.shape}}

5 elements in the sequence, only 1 sequence, of dimensionality 2

`hidden` contains the hidden state $\mathbf{h}$ and the cell state $\mathbf{c}$

```
out[-1], hidden[0] # are the same thing
```

{{out[-1].detach().numpy(), hidden[0].detach().numpy() }}

Let us now batchify the data

Batch size is 10

In [11]:
```
batch_size = 10
inputs_batch = torch.randn(5, batch_size, 3)
hidden = (torch.randn(1, batch_size, hidden_size),
          torch.randn(1, batch_size, hidden_size))  # clean out hidden state
out, (H, C) = lstm(inputs_batch, hidden)
```

Question: what is the size now of `out`, `H` and `C`?

In [12]:
```
out.shape
```

Out[12]:
```
torch.Size([5, 10, 2])
```

In [13]:
```
H.shape, C.shape
```

Out[13]:
```
(torch.Size([1, 10, 2]), torch.Size([1, 10, 2]))
```

POS tagging with LSTM

In this section, we will use an LSTM to get part of speech tags. We will not use Viterbi or Forward-Backward or anything like that.

The model is as follows: let our input sentence be $w_1, \dots, w_M$, where $w_i \in V$, our vocab.

Also, let T be our tag set, and y_i the tag of word w_i. Denote our prediction of the tag of word w_i by $\hat{y}_i$.

This is a structure prediction, model, where our output is a sequence $\hat{y}_1, \dots, \hat{y}_M$, where $\hat{y}_i \in T$.

There is NO POS tagging anymore yet seq2seq modeling

To do the prediction, pass an LSTM over the sentence. Denote the hidden state at timestep i as $\mathbf{h}_i$. Also, assign each tag a unique index. Then our prediction rule for $\hat{y}_i$ is

$$\hat{y}_i = \text{argmax}_j \ (\log \text{Softmax}(A\mathbf{h}_i + \mathbf{b}))_j$$

That is, take the log softmax of the affine map of the hidden state, and the predicted tag is the tag that has the maximum value in this vector. Note this implies immediately that the dimensionality of the target space of A is $|T|$.

Prepare the data

In [14]:
```
training_data = [
    # Tags are: DET - determiner; NN - noun; V - verb
    # For example, the word "The" is a determiner
    ("The dog ate the apple".split(), ["DET", "NN", "V", "DET", "NN"]),
    ("Everybody read that book".split(), ["NN", "V", "DET", "NN"])
]
```

Now to each word we assign a unique ID

In [15]:
```
word_to_ix = {}
for seq_x, seq_y in training_data:
    for word in seq_x:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)
```

```
{'The': 0, 'dog': 1, 'ate': 2, 'the': 3, 'apple': 4, 'Everybody': 5, 'read': 6, 'that': 7, 'book': 8}
```

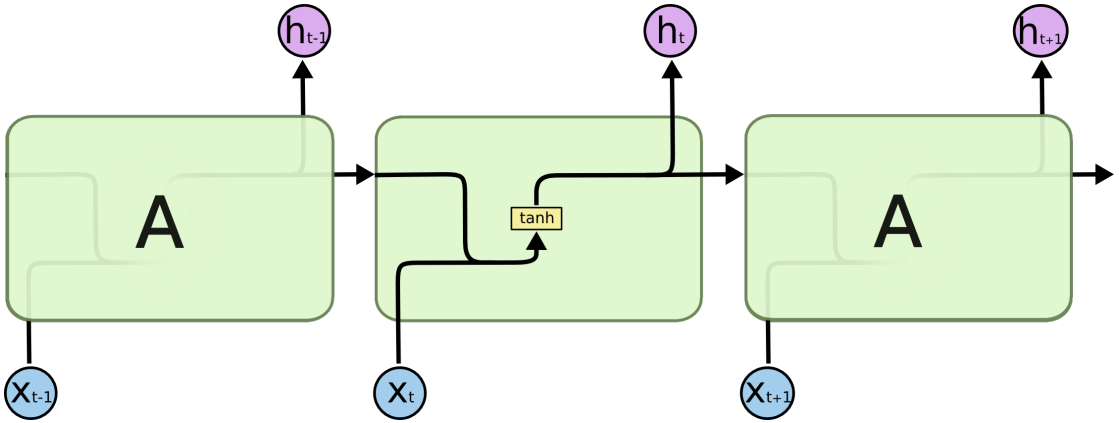
Now to each label we assign a unique ID

In [16]:
```
tag_to_ix = {"DET": 0, "NN": 1, "V": 2}  # Assign each tag with a unique index
ix_to_tag =  { v:k for k,v in tag_to_ix.items()}
```

From text to tensor

In [17]:
```
def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)
```

LSTM + Input embedding + Output projection



Taken from [colah.github.io](https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

In [18]:
```
# These will usually be more like 32 or 64 dimensional.
# We will keep them small, so we can see how the weights change as we train.
EMBEDDING_DIM = 3
HIDDEN_DIM = 2
```

The model

In [19]:
```
class LSTMTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        # NOTE that we could have used LSTM Pytorch API to implement this
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, _ = self.lstm(embeds.view(len(sentence), 1, -1))
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

The training

In [20]:
```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Sanity Check

In [21]:
```
# See what the scores are before training
# Note that element i,j of the output is the score for tag j for word i.
# Here we don't need to train, so the code is wrapped in torch.no_grad()
with torch.no_grad():
    seq_x = training_data[0][0] # 1st training take the input seq
    inputs = prepare_sequence(seq_x, word_to_ix)
    tag_scores = model(inputs)
    print(tag_scores) # num. of words (5) x num. of tags (3)
```

```
tensor([[-1.4133, -0.8714, -1.0838],
        [-1.3909, -0.8884, -1.0793],
        [-1.5004, -0.8467, -1.0552],
        [-1.4326, -0.8832, -1.0560],
        [-1.6684, -0.7974, -1.0189]])
```

Fitting part (training)

```
for epoch in range(300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data: # 5 words --> 3 classes
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is, turn them into
        # Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = prepare_sequence(tags, tag_to_ix)

        # Step 3. Run our forward pass.
        tag_scores = model(sentence_in)

        # Step 4. Compute the loss, gradients, and update the parameters by
        #  calling optimizer.step()
        loss = loss_function(tag_scores, targets)
        loss.backward() # get the gradients over params
        optimizer.step() # update the params
```

In [22]:
```
for epoch in range(300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data: # 5 words --> 3 classes
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is, turn them into
        # Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = prepare_sequence(tags, tag_to_ix)

        # Step 3. Run our forward pass.
        tag_scores = model(sentence_in)

        # Step 4. Compute the loss, gradients, and update the parameters by
        #  calling optimizer.step()
        loss = loss_function(tag_scores, targets)
        loss.backward() # get the gradients over params
        optimizer.step() # update the params
```

Let's see now the score after training (on the training data)

In [23]:
```
# See what the scores are after training
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)

    # The sentence is "the dog ate the apple".  i,j corresponds to score for tag j
    # for word i. The predicted tag is the maximum scoring tag.
    # Here, we can see the predicted sequence below is 0 1 2 0 1
    # since 0 is index of the maximum value of row 1,
    # 1 is the index of maximum value of row 2, etc.
    # Which is DET NOUN VERB DET NOUN, the correct sequence!
    print(tag_scores)
```

```
tensor([[-0.0521, -3.6669, -3.6814],
        [-3.8780, -0.3398, -1.3189],
        [-3.0486, -0.5128, -1.0391],
        [-0.0356, -4.1671, -3.9390],
        [-4.4822, -0.2243, -1.6628]])
```

In [24]:
```
print(training_data[0])
print("predicted ", " -> ".join([ix_to_tag[i.item()] for i in torch.argmax(tag_scores, dim=1)]))
```

```
(['The', 'dog', 'ate', 'the', 'apple'], ['DET', 'NN', 'V', 'DET', 'NN'])
predicted  DET -> NN -> NN -> DET -> NN
```

Remember: for evaluation you have to validate on the valid/test set!

Homework

1. Study and reproduce NLP From Scratch: Classifying Names with a Character-Level RNN
2. Study and reproduce NLP From Scratch: Generating Names with a Character-Level RNN

Natural Language Processing

Neural Machine Translation (NMT)

Prof. Iacopo Masi and Prof. Stefano Faralli

Today's lecture

- What is Machine Translation

- Neural Machine Translation (NMT)

- Beam Search

- How to eval NMT

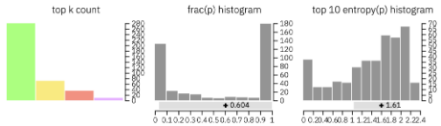
This lecture material is taken from

📘 **Chapter 10 Jurafsky Book**

📘 **Chapter 18 Eisenstein Book**

- Stanford Slide NMT
- Stanford Lecture NMT

The Giant Language Model Test Room - GLTR



Example Analyses

Unicorns?

As a first example, we investigate a now famous generated text, the unicorn sample from an unreleas
OpenAI. The first sentence is the prompt given to the model, and the rest of the text is entirely gener
and it is very hard to detect from reading it whether it was written by an algorithm or a human.

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexpl
ored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the
unicorns spoke perfect English.
The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned,
silver-white unicorns were previously unknown to science.
Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally
solved.
Dr. Jorge PÃ©rez, an evolutionary biologist from the University of La Paz, and several companions,
were exploring the Andes Mountains when they found a small valley, with no other animals or
humans. PÃ©rez noticed that the valley had what appeared to be a natural fountain, surrounded
by two peaks of rock and silver snow.
PÃ©rez and the others then ventured further into the valley. "By the time we reached the top of
one peak, the water looked blue, with some crystals on top," said PÃ©rez.
PÃ©rez and his friends were astonished to see the unicorn herd. These creatures could be seen
from the air without having to move too much to see them âĢĵ they were so close they could
touch their horns.
While examining these bizarre creatures the scientists discovered t lso spoke

try our demo

Machine Translation

Machine Translation (MT) is the task of translating a sentence $\mathbf{x}$ from one language (the source language) to a sentence $\mathbf{y}$ in another language (the target language).

$$\mathbf{x} \quad \text{L'homme est né libre, et partout il est dans les fers}$$

Machine Translation

Machine Translation (MT) is the task of translating a sentence x from one language (the source language) to a sentence y in another language (the target language).

$$\mathbf{x} \quad \text{L'homme est né libre, et partout il est dans les fers}$$

$$\mathbf{y} \quad \text{Man is born free, but everywhere he is in chains}$$

The early history of MT: 1950s

- Machine translation research began in the early 1950s on machines less powerful than high school calculators (before term "A.I." coined!)
- Concurrent with foundational work on automata, formal languages, probabilities, and information theory
- MT heavily funded by military, but basically just simple rule-based systems doing word substitution
- Human language is more complicated than that, and varies more across languages!
- Little understanding of natural language syntax, semantics, pragmatics
- Problem soon appeared **intractable**

1990s-2010s: Statistical Machine Translation

Core idea: Learn a probabilistic model from data

- Suppose we're translating French → English.
- We want to find best English sentence y, given French sentence x

$$\arg\max_y p(y|x)$$

- Use Bayes Rule to break this down into two components to be learned separately:

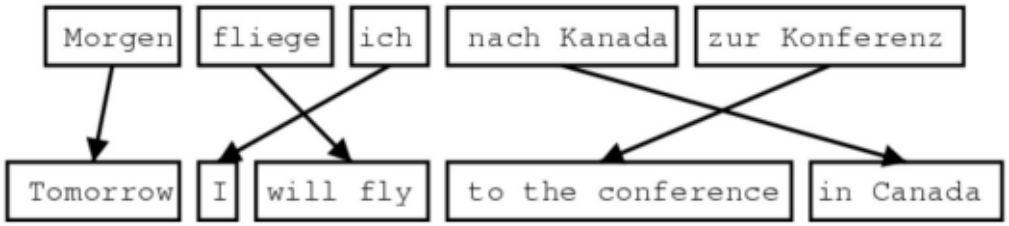
$$= \arg\max_y p(x|y)p(y)$$

$p(y) \longrightarrow$ Models how to write good English **(fluency)**. Learned from monolingual data

$p(x|y) \longrightarrow$ Translation Model: Models how words and phrases should be translated **(fidelity)**. Learned from aligned data.

Translation is not trivial to model!

There is not a clear alignment nor one-to-one mapping



Different symbols

1519年600名西班牙人在墨西哥登陆，去征服几百万人口的阿兹特克帝国，初次交锋他们损兵三分之二。

In 1519, six hundred Spaniards landed in Mexico to conquer the Aztec Empire with a population of a few million. They lost two thirds of their soldiers in the first clash.

Google Translate over time

translate.google.com (2009): 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of soldiers against their loss.

translate.google.com (2013): 1519 600 Spaniards landed in Mexico to conquer the Aztec empire, hundreds of millions of people, the initial confrontation loss of soldiers two-thirds.

translate.google.com (2015): 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of the loss of soldiers they clash.

1990s-2010s: Statistical Machine Translation (SMT)

SMT was a huge research field

- The best systems were extremely complex
- Hundreds of important details
  - Systems had many separately-designed subcomponents
- Lots of feature engineering
  - Need to design features to capture particular language phenomena
  - Required compiling and maintaining extra resources
  - Lots of human effort to maintain
  - Repeated effort for each language pair!

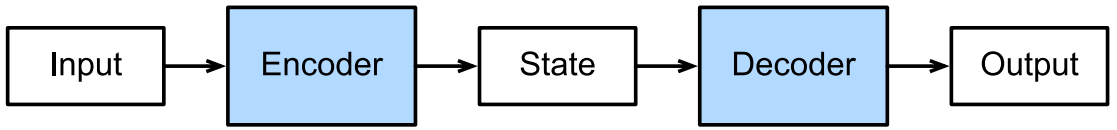
Then came Neural MT [circa 2014]

|

Neural Machine Translation

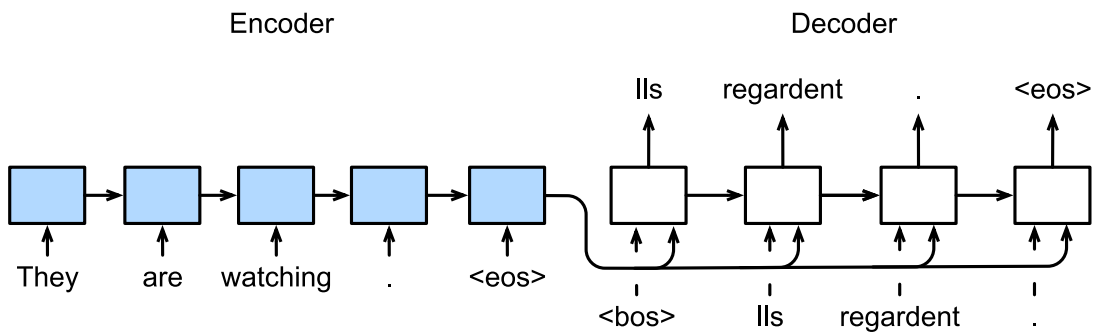
Neural Machine Translation (NMT) is a way to do Machine Translation with **a single end-to-end neural network.**

The neural network architecture is called a **sequence-to-sequence model (aka seq2seq)** and it involves **two RNNs**.



Neural Machine Translation

Here, the encoder RNN will take a **variable-length sequence** as input and transform it into a **fixed-shape hidden state**. Later, we will introduce attention mechanisms, which allow us to access encoded inputs without having to compress the entire input into a single fixed-length representation



Sequence-to-sequence is versatile!

- The general notion here is an **encoder-decoder** model
  - One neural network takes input and produces a neural representation
  - Another network produces output based on that neural representation
  - If the input and output are sequences, we call it a `seq2seq` model
- Sequence-to-sequence is useful for **more than just MT**

NLP tasks ≈ seq2seq:

- Summarization (long text → short text)
- Dialogue (previous utterances → next utterance)
- Parsing (input text → output parsed as sequence)
- Code generation (natural language → Python code)

Python to natural language

Code Translation

Explain a piece of Python code in human understandable language.

Prompt

```
# Python 3
def remove_common_prefix(x, prefix, ws_prefix):
  x["completion"] = x["completion"].str[len(prefix) :]
  if ws_prefix:
    # keep the single whitespace as prefix
    x["completion"] = " " + x["completion"]
return x

# Explanation of what the code does

#
```

Settings

Engine	code-davinci-002
Max tokens	64
Temperature	0
Top p	1.0
Frequency penalty	0.0
Presence penalty	0.0
Stop sequence	

Sample response

The code above is a function that takes a dataframe and a prefix as input and returns a dataframe with the prefix removed from the completion column.

NTM is a Conditional LM

The sequence-to-sequence model is an example of a **Conditional Language Model**

- **Language Model** because the decoder is predicting the next word of the target sentence y
- **Conditional** because its predictions are also conditioned on the source sentence x

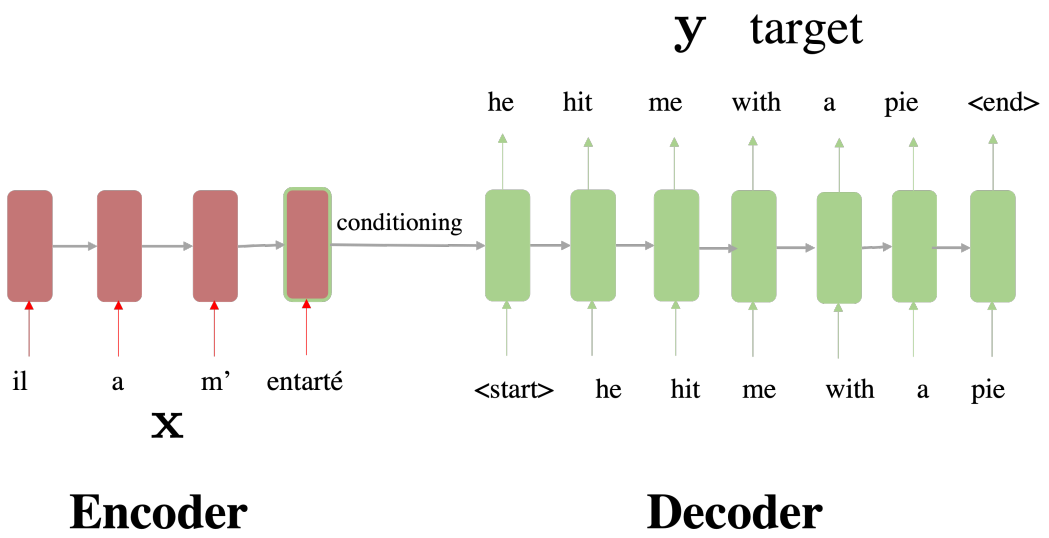
NMT calculates $p(y|x)$ where y is the target sentence and x is the input sentence. Generate likely samples of y given that as input I had x.

$$p(y|x) = p(y_1|x)p(y_2|y_1, x)p(y_3|y_1, y_2, x) \dots p(y_t|y_1, y_2, \dots, y_{t-1}, x)$$

NTM is a Conditional LM

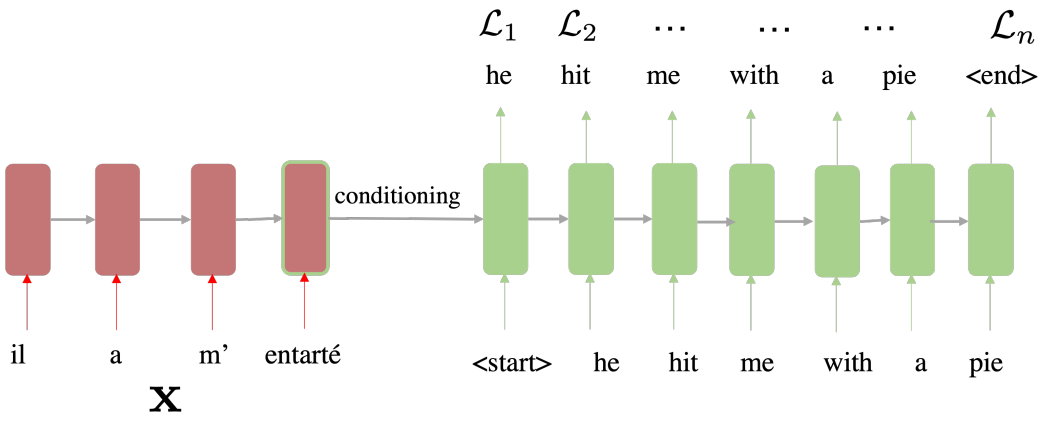
Question: How to train an NMT system?

- (Easy) Answer: Get a big parallel corpus...
- But there is now exciting work on "unsupervised NMT", data augmentation, etc.

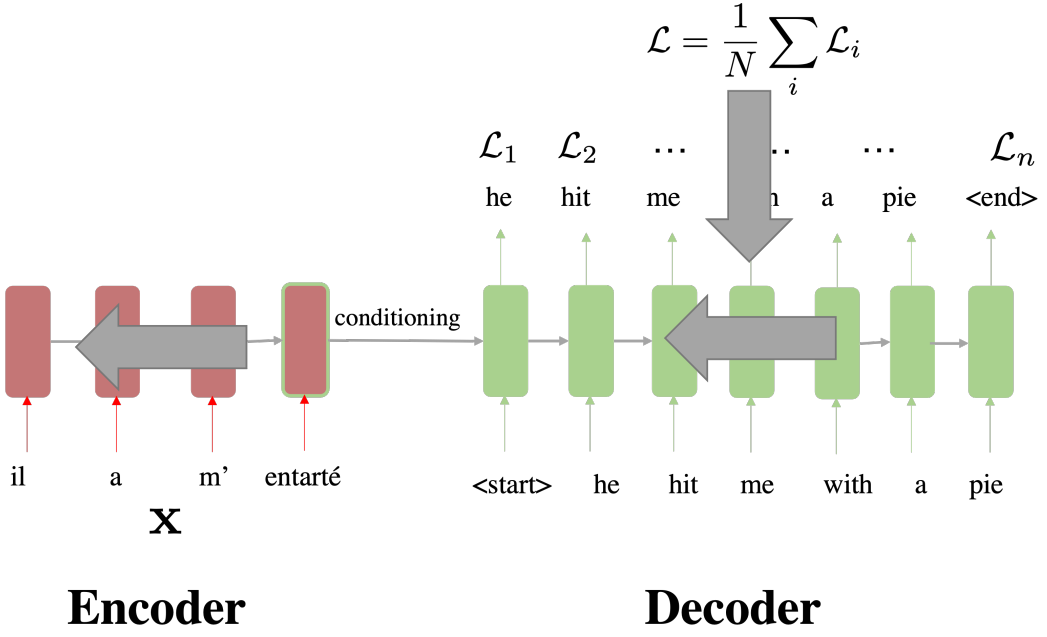


Training NMT

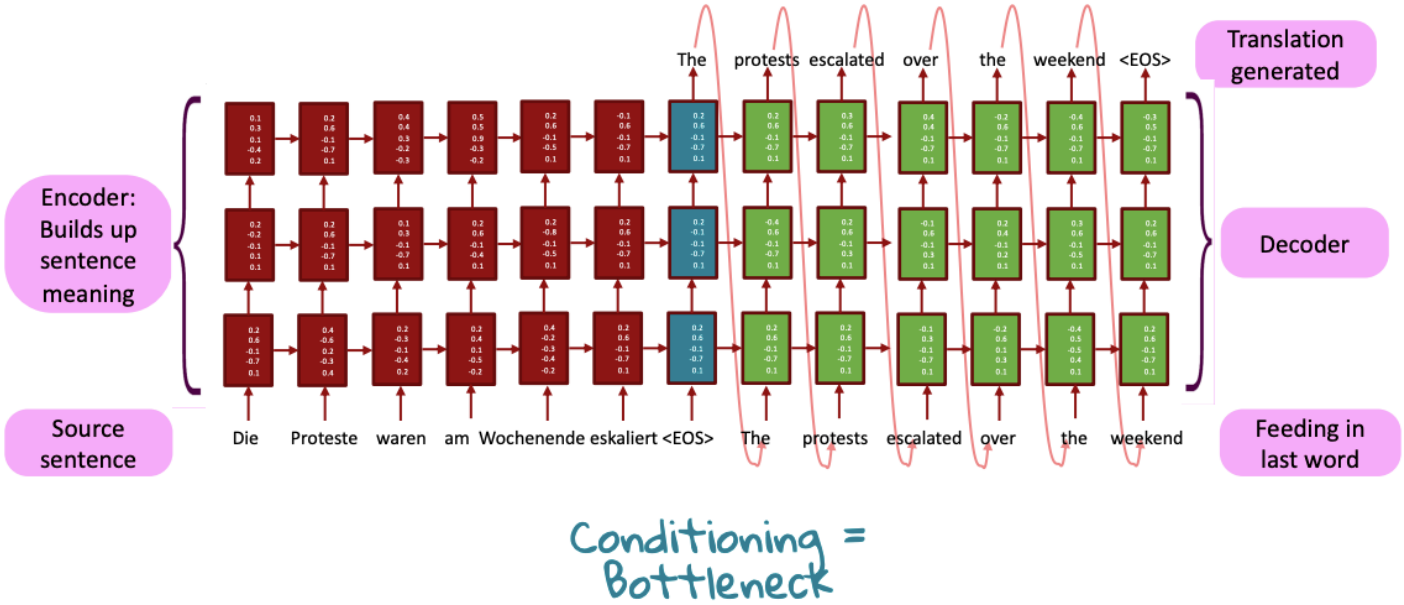
$$\mathcal{L} = \frac{1}{N} \sum_i \mathcal{L}_i$$



Training NMT



Multi-layer Deep encoder-decoder Machine Translation Net



[Sutskever et al. 2014; Luong et al. 2015]
Slide from Stanford

How do we generate a sentence?

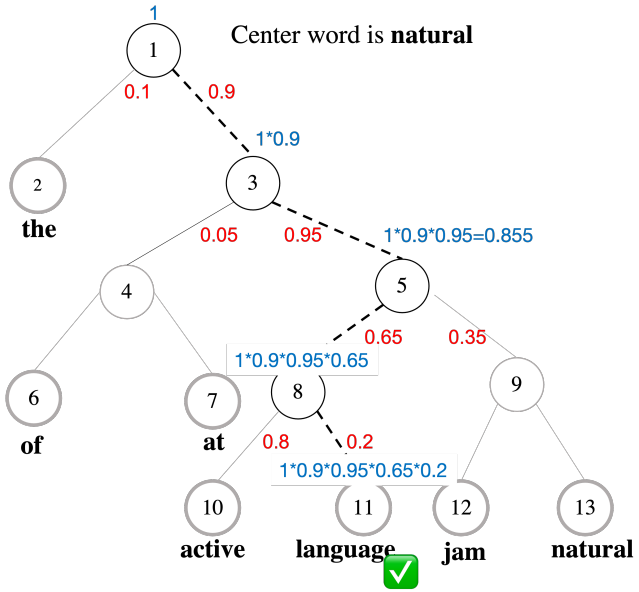
Generating is also called "decoding"

Decoding happens at inference [test] time

Remember something already seen about decoding?

$p(w_{t-1}|w_t = \text{natural}) = 1 \cdot 0.9 \cdot 0.95 \cdot 0.65 \cdot 0.2 = 0.11115$

Loss is $-\log\left(p(w_{t-1}|w_t = \text{natural})\right) = -\log(0.11115)$



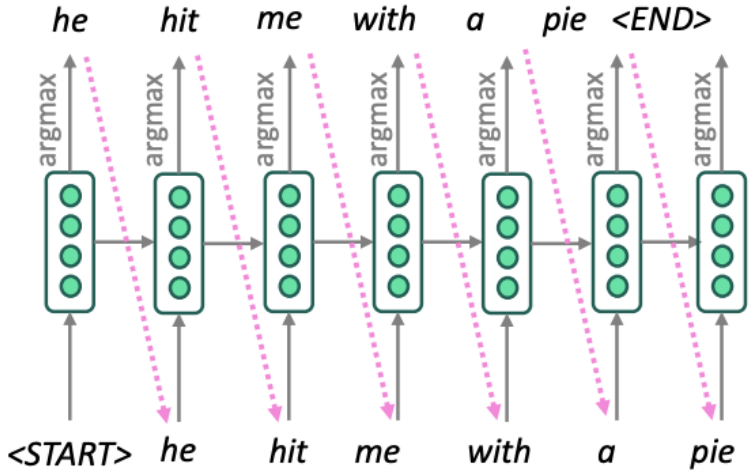
Decoding

Important: In inference with do not have the label!

1. Exhaustive search [too complex]
2. **Greedy search** (at each branch take the branch at maximum probability) [too greedy and deterministic] ⟵
3. Beam search

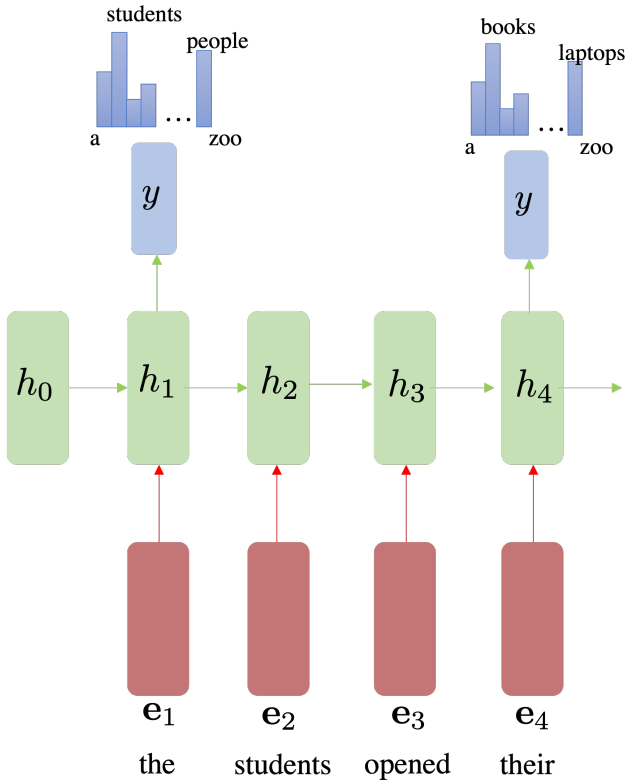
Decoding: Greedy decoding

- We saw how to generate (or "decode") the target sentence by taking `argmax` on each step of the decoder.
- This is greedy decoding (take most probable word on each step)



Decoding: Probabilistic Decoding

- Instead of doing `argmax` we sample from the probability at each layer.
- This makes the algorithm randomized in case we want to generate multiple, similar sentences.



The problem with Greedy Decoding

Greedy decoding has no way to undo decisions! Once a decision is taken, is taken!

- Input: `il a m'entarté` ⟶ `he hit me with a pie`

Step 1: `he _____`

Step 2: `he hit_____`

Step 3: `he hit a_____` *(wrong prediction, no way of going back)*

Greedy is suboptimal: at each local step, you just choose the maximum, without seeing the entire distribution

Decoding

Important: In inference with do not have the label!

1. **Exhaustive search** [too complex] ⟵
2. Greedy search (at each branch take the branch at maximum probability) [too greedy]
3. Beam search (we will cover later on)

Exhaustive Search

Ideally, we want to find a (length T) translation/decoding y that maximizes:

$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_2,y_1,x)\dots(y_T|y_{T-1},\dots,y_1,x) = \prod_{i=1}^{T} p(y_T|y_{T-1},\dots,y_1,x)$$

We could try computing **all possible sequences y** and take globally the most likely.

- This means that on each step t of the decoder, we are tracking V^t possible partial translations, where V is vocab size
- This $\mathcal{O}(|V|^T)$ is **far too expensive.**

Decoding

1. Exhaustive search [too complex]
2. Greedy search (at each branch take the branch at maximum probability) [too greedy]
3. **Beam search** ⟵ ✅

Beam search decoding 🚀

Core idea: take a trade-off approach between local (greedy) and global (exhaustive). On each step of decoder, keep track of the k **most probable partial translations** (which we call **hypotheses**)

A **hypothesis** $\{y_1,\dots,y_t\}$ has a score which is its log probability:

$\text{score}\{y_1,\dots,y_t\} = \log p(y_1,\dots,y_t|x) = \sum_{i=1}^{t} \log p(y_i|y_1,\dots,y_{i-1},x)$

- Scores are all negative (negative means low prob.), and higher score is better (high prob.)
- We search for high-scoring hypotheses, **tracking top-k on each step**

Beam search decoding

Beam search is not guaranteed to find optimal solution but much more efficient than exhaustive search. To better understand how **beam search** works we use a **search tree**.

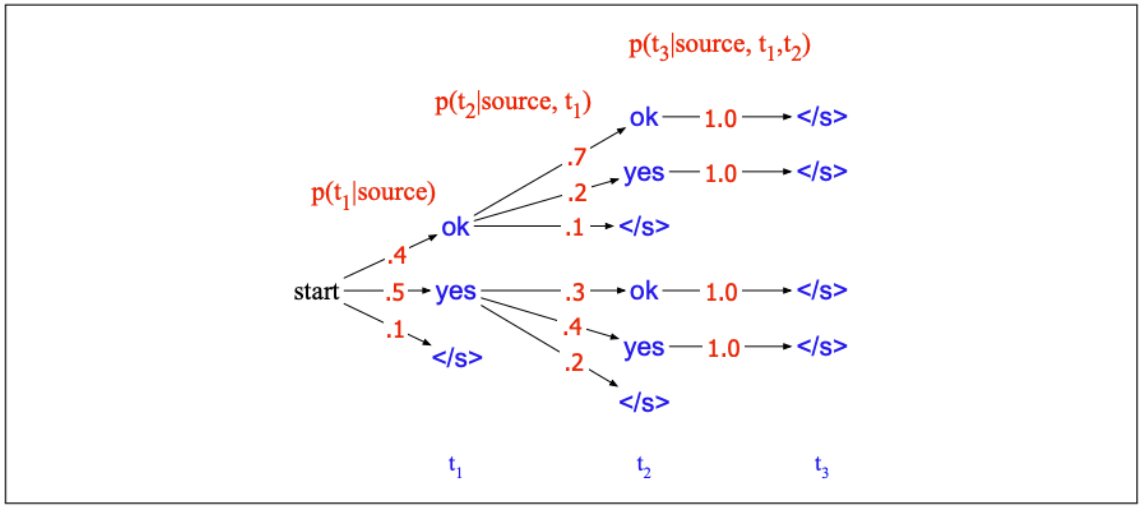


Figure 10.11 A search tree for generating the target string $T = t_1, t_2, \ldots$ from the vocabulary $V = \{\text{yes}, \text{ok}, \text{<s>}\}$, given the source string, showing the probability of generating each token from that state. Greedy search would choose *yes* at the first time step followed by *yes*, instead of the globally most probable sequence *ok ok*.

Beam search decoding

This **fixed-size beam width memory footprint** k is called the **beam width**, on the metaphor of a flashlight beam that can be parameterized to be wider or narrower. In practice k is 5 or 10.

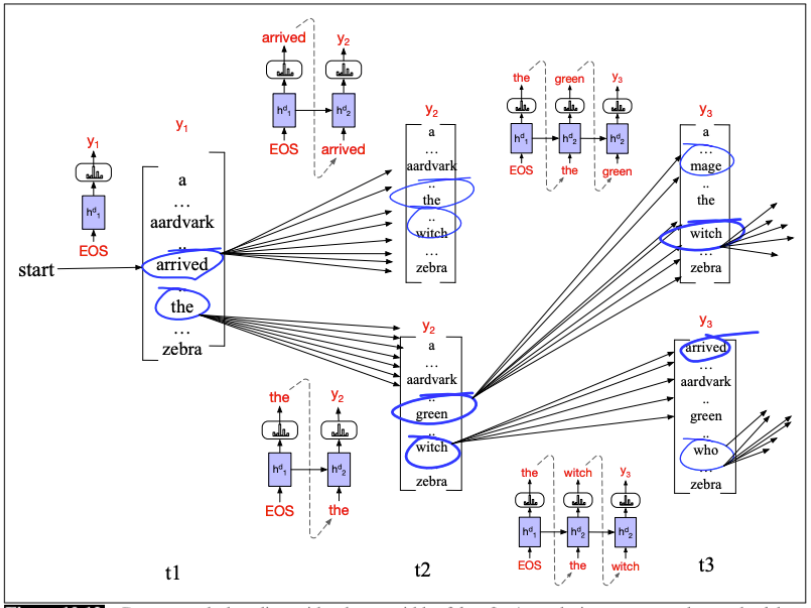


Figure 10.12 Beam search decoding with a beam width of $k = 2$. At each time step, we choose the k best hypotheses, compute the V possible extensions of each hypothesis, score the resulting $k * V$ possible hypotheses and choose the best k to continue. At time 1, the frontier is filled with the best 2 options from the initial state of the decoder: *arrived* and *the*. We then extend each of those, compute the probability of all the hypotheses so far (*arrived the*, *arrived aardvark*, *the green*, *the witch*) and compute the best 2 (in this case *the green* and *the witch*) to be the search frontier to extend on the next step. On the arcs we show the decoders that we run to score the extension words (although for simplicity we haven't shown the context value c_i that is input at each step).

Beam search decoding

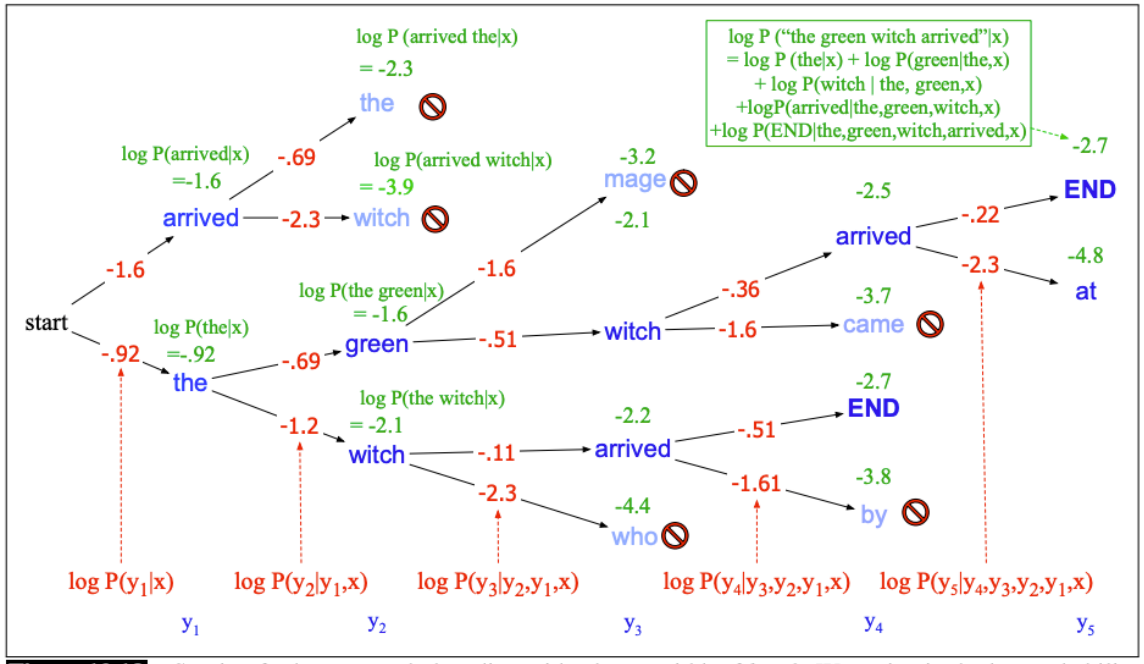


Figure 10.13 Scoring for beam search decoding with a beam width of $k = 2$. We maintain the log probability of each hypothesis in the beam by incrementally adding the logprob of generating each next token. Only the top k paths are extended to the next step.

Beam search decoding: stopping criterion

In greedy decoding, usually we decode until the model produces an `<END>` token

- For example: `<START> he hit me with a pie <END>`

In beam search decoding, different hypotheses may produce `<END>` tokens on different timesteps:

- When a hypothesis produces `<END>`, that hypothesis is complete.
- Place it aside and continue exploring other hypotheses via beam search.

Usually we continue beam search until:

- We reach **timestep T (where T is some pre-defined cutoff)**, OR
- We have at least n completed hypotheses (where n is pre-defined cutoff)

Beam search decoding 🔦: finishing up

We have our list of completed hypotheses.

- How to select top one?
- Each hypothesis $\{y_1, \ldots, y_n\}$ on our list has a score:

$$score(y_1, \ldots, y_n) = \log p(y_1, \ldots, y_n|x) = \sum_{i=i}^{T} \log p(y_i|y_1, \ldots, y_{i-1}, x)$$

Can you spot a problem with this?

Shorter Sentences may have higher scores!

Fix: Normalize by length. Use this to select top one instead

$$\frac{1}{T} \sum_{i=i}^{T} \log p(y_i|y_1, \ldots, y_{i-1}, x)$$

How do we evaluate Machine Translation?

BLEU (Bilingual Evaluation Understudy)

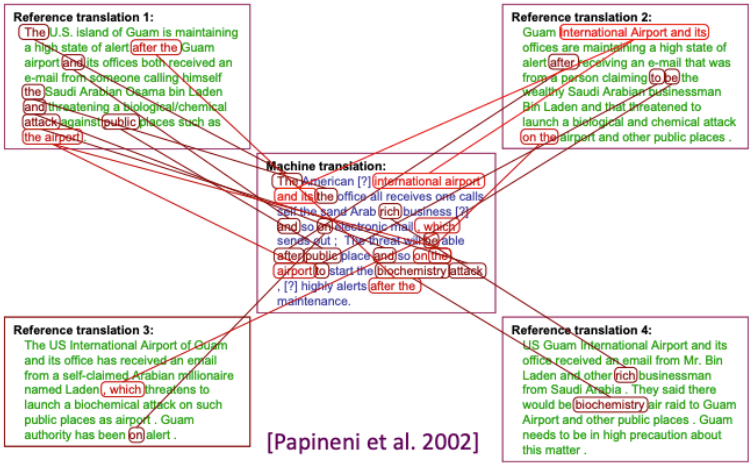
BLEU compares the machine-written translation to one or several human-written translation(s), and **computes a similarity score** based on:

- Geometric mean of **n-gram precision** (usually for 1, 2, 3 and 4-grams)
- Plus a penalty for too-short system translations

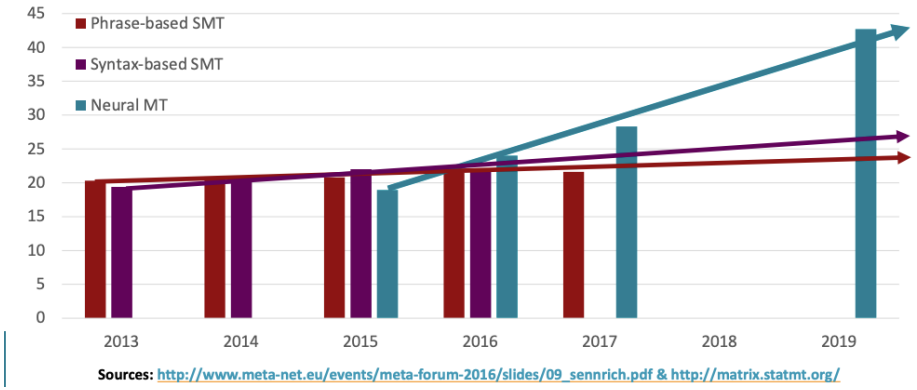
BLEU is useful but imperfect:

- There are many valid ways to translate a sentence
- So a good translation can get a poor BLEU score because it has low n-gram overlap with the human translation

BLEU



NMT Progress



Sources: http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf & http://matrix.statmt.org/

Advantages of NMT ✅

Compared to SMT, NMT has many **advantages**:

- Better performance
  - More fluent
  - Better use of context
  - Better use of phrase similarities
- A single neural network to be optimized end-to-end
  - No subcomponents to be individually optimized
- Requires much less human engineering effort
  - No feature engineering
  - Same method for all language pairs

Disadvantages of NMT ❌

Compared to SMT:

- NMT is **less interpretable**
  - Hard to debug
  - Why this translation came up?
- NMT is **difficult to control**
  - For example, cannot easily specify rules or guidelines for translation
  - Safety concerns!
  - Invention of content not in source
  - Systematic gender biases

NMT: the first big success story of NLP Deep Learning

Neural Machine Translation went from a **fringe research attempt** in 2014 to the leading **standard method** in 2016

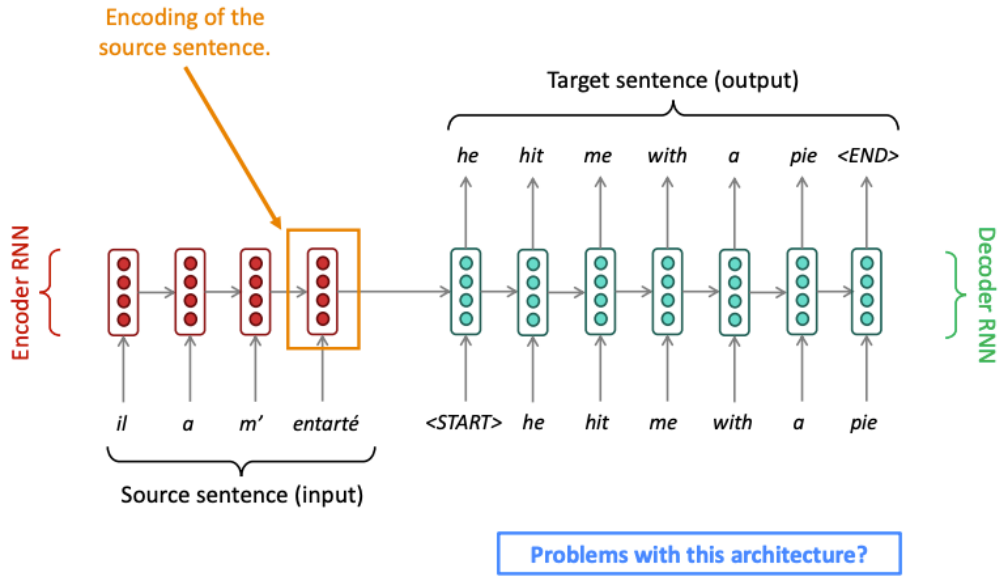
- 2014: First seq2seq paper published [Sutskever et al. 2014]
- 2016: Google Translate switches from SMT to NMT – and by 2018 everyone has.

- SMT systems, built by hundreds of engineers over many years, outperformed by NMT systems trained by small groups of engineers in a few months

Using NMT to introduce Attention 🧐

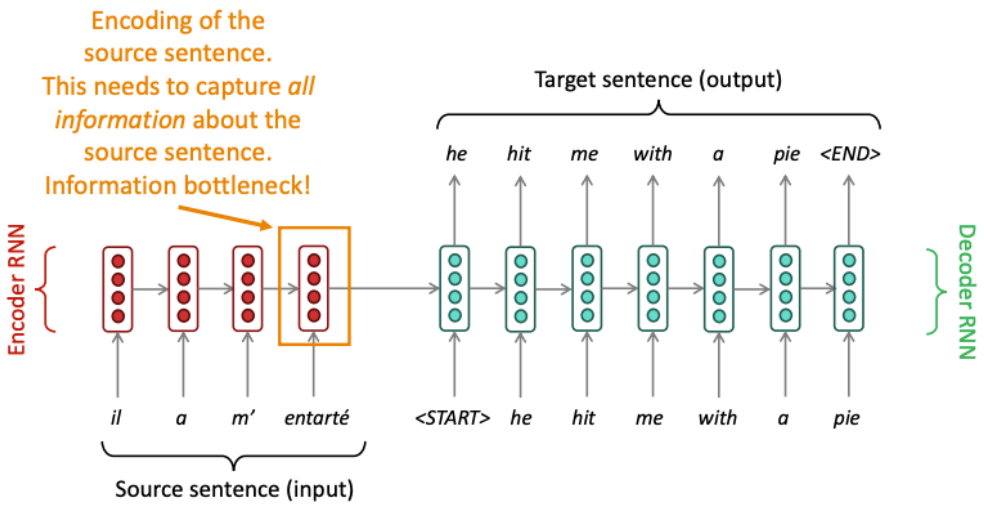
NTM is a Conditional LM

Do you see any problem with this architecture?



Picture from Stanford

Why attention? Sequence-to-sequence: the bottleneck problem



Picture from Stanford

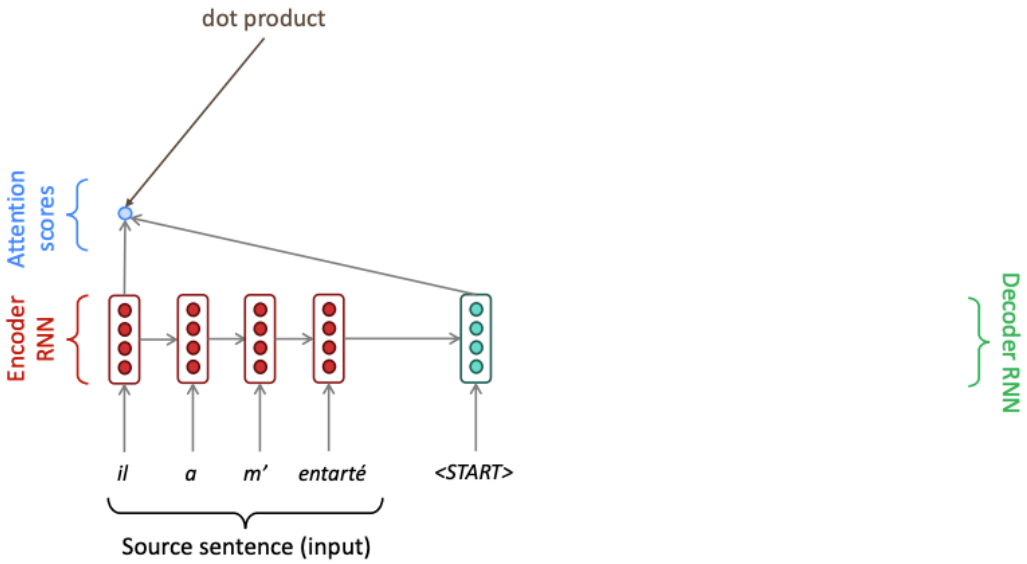
Why attention? Sequence-to-sequence: the bottleneck problem

Attention

Attention provides a **solution to the bottleneck problem.**

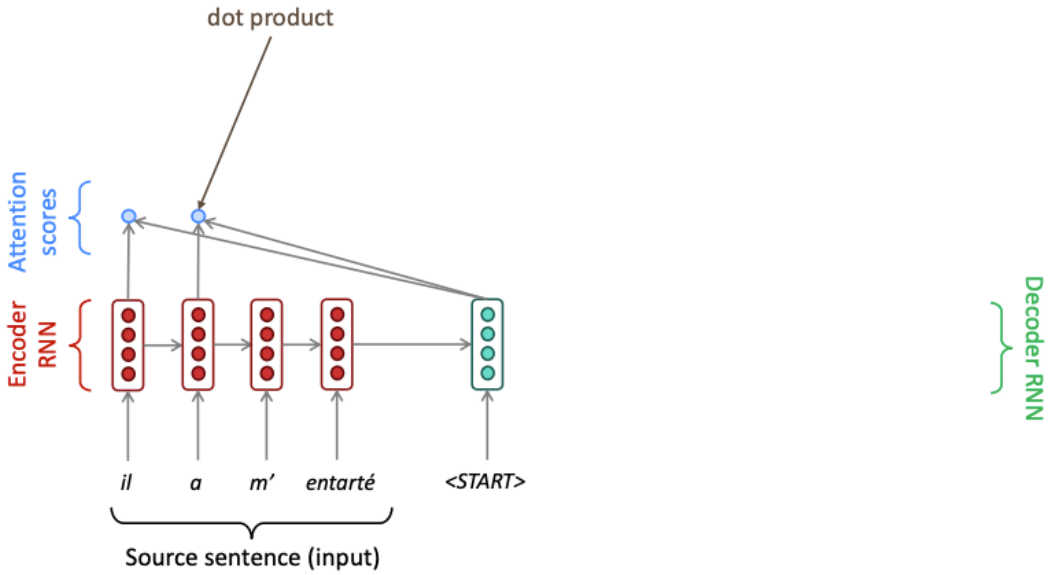
Core idea: on each step of the decoder, **use direct connection to the encoder to focus on a particular part** of the source sequence

Sequence-to-sequence with attention



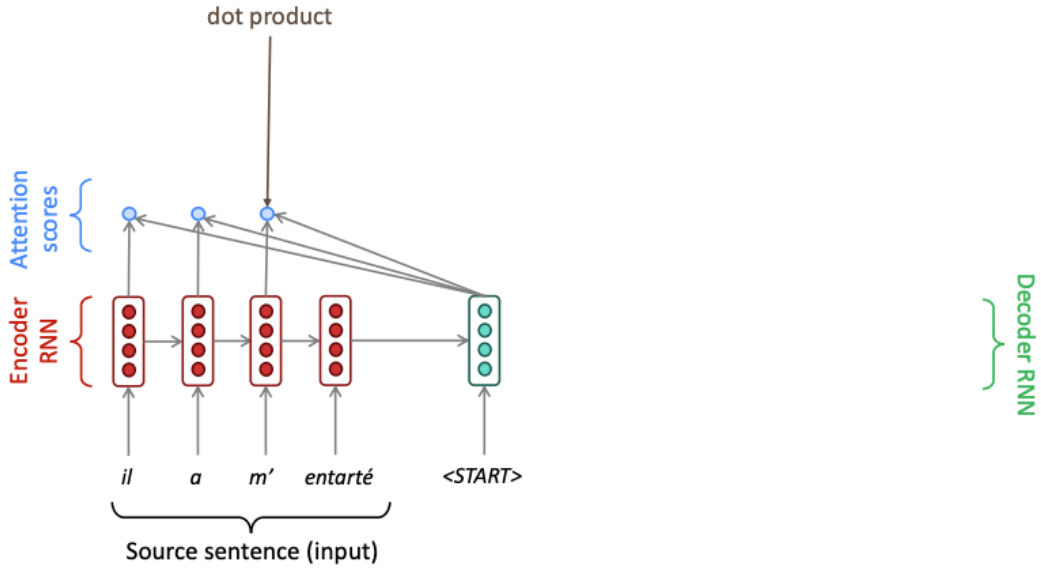
Picture from Stanford

Sequence-to-sequence with attention



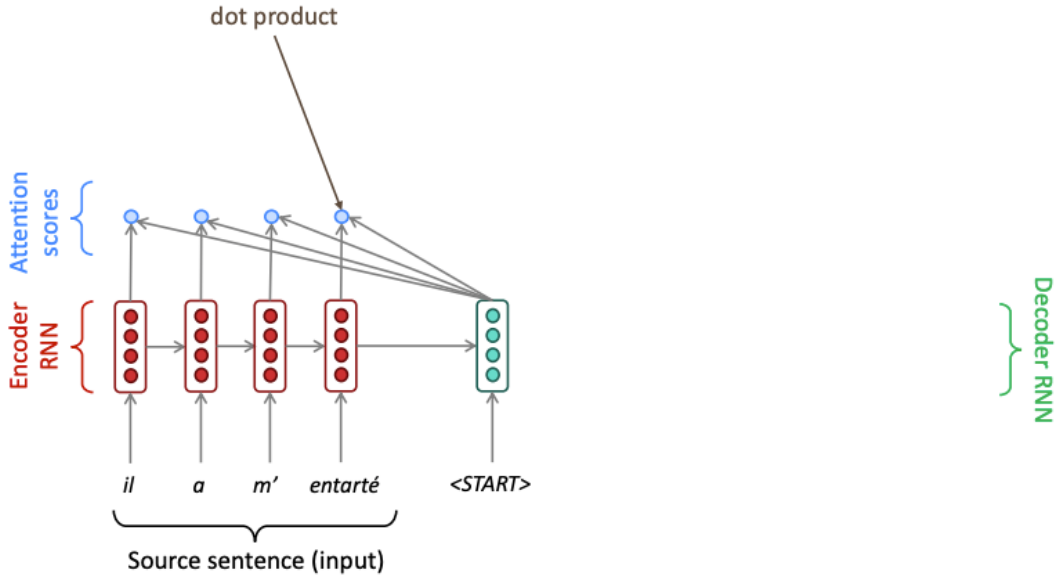
Picture from Stanford

Sequence-to-sequence with attention



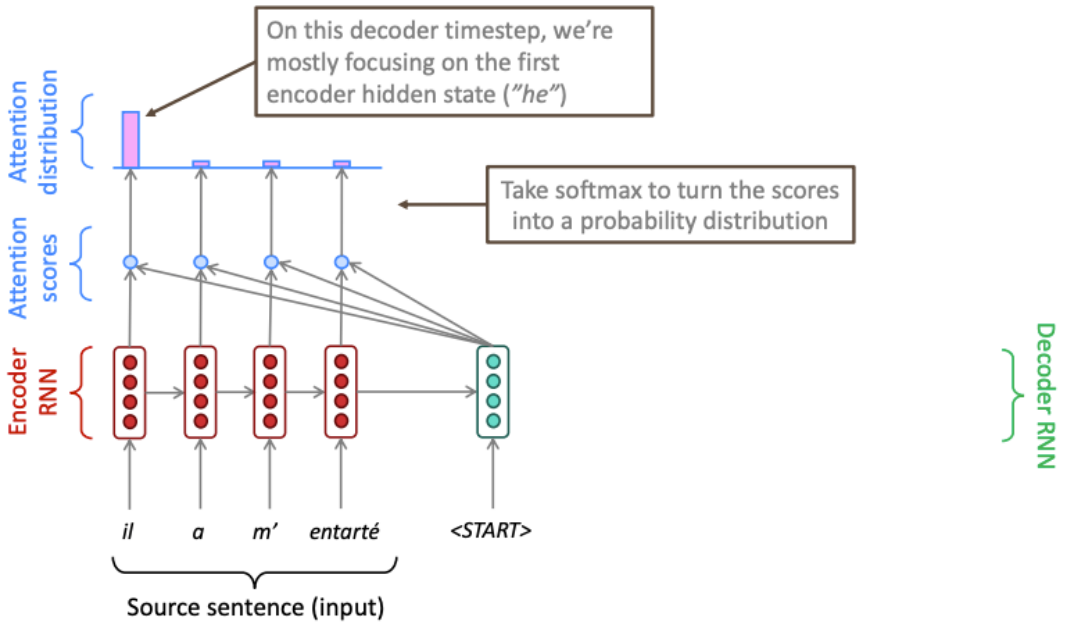
Picture from Stanford

Sequence-to-sequence with attention



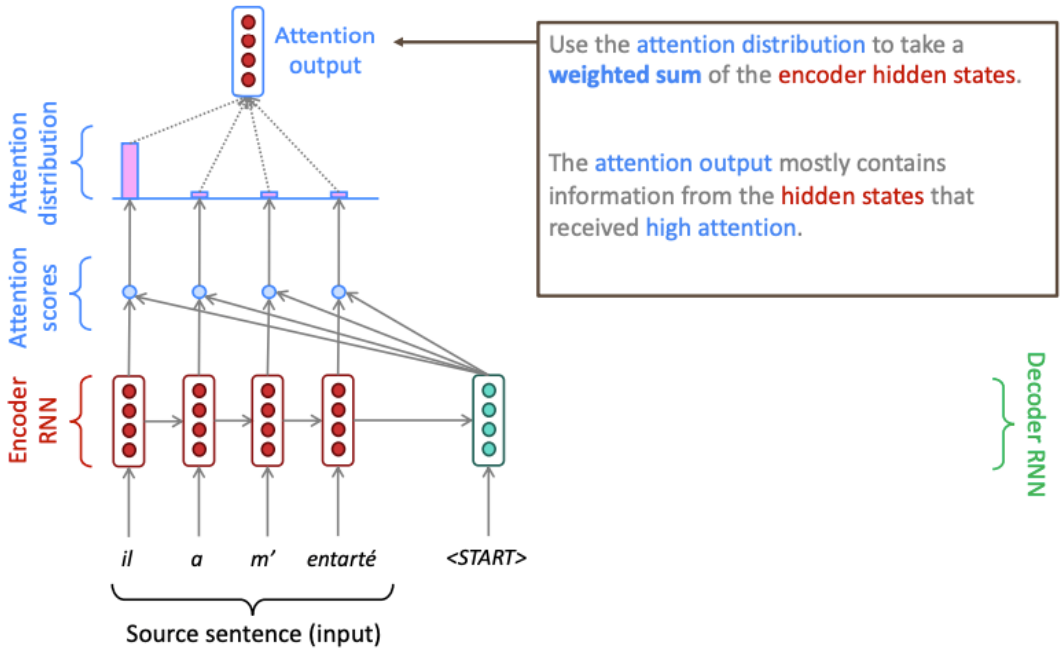
Picture from Stanford

Sequence-to-sequence with attention



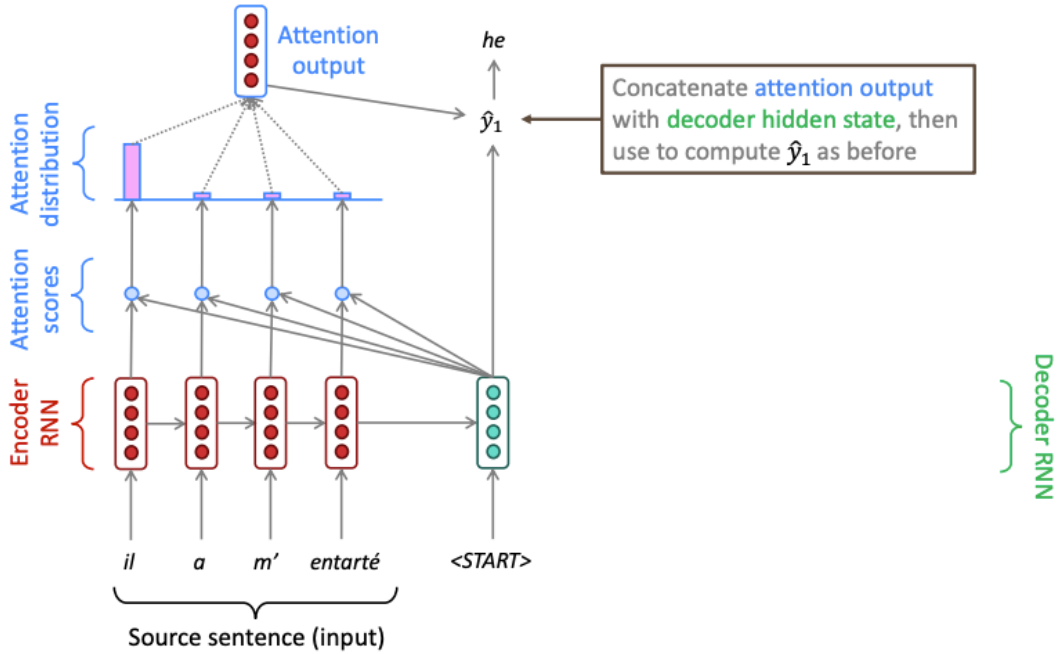
Picture from Stanford

Sequence-to-sequence with attention



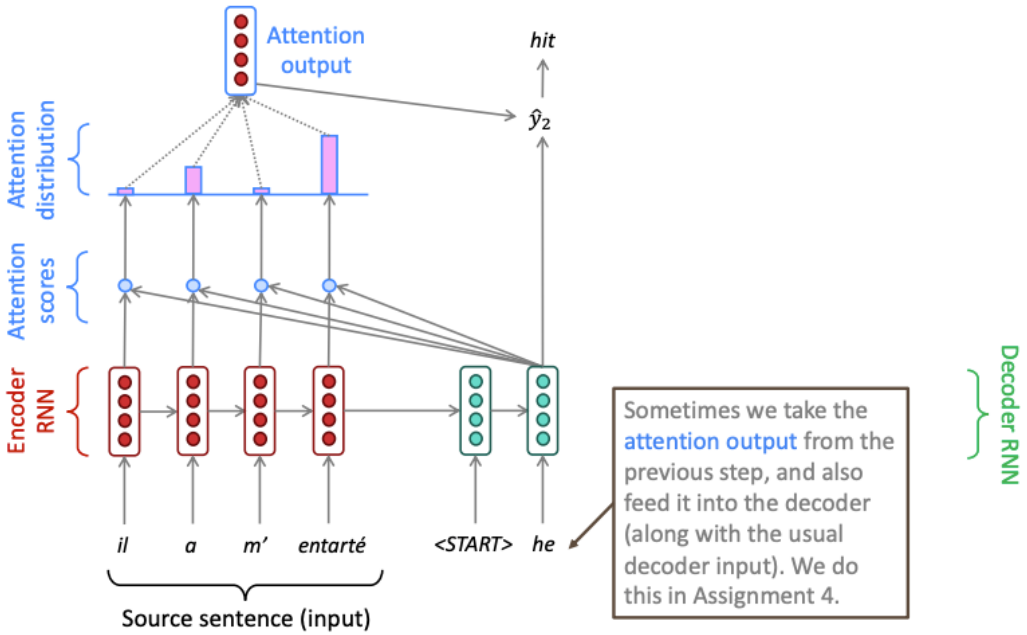
Picture from Stanford

Sequence-to-sequence with attention



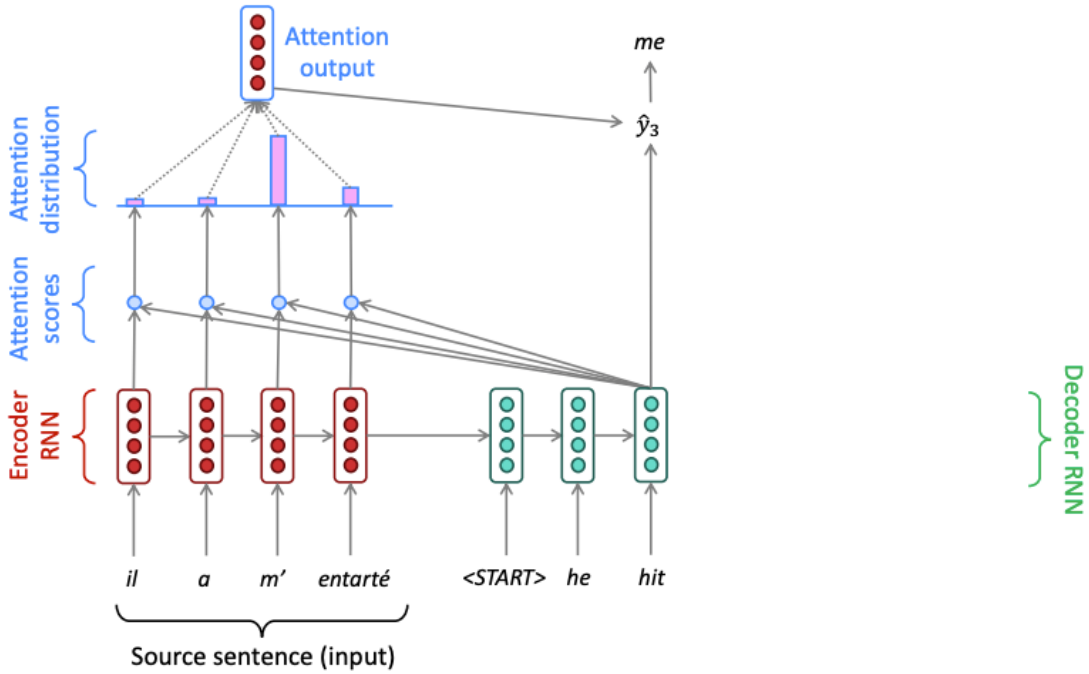
Picture from Stanford

Sequence-to-sequence with attention



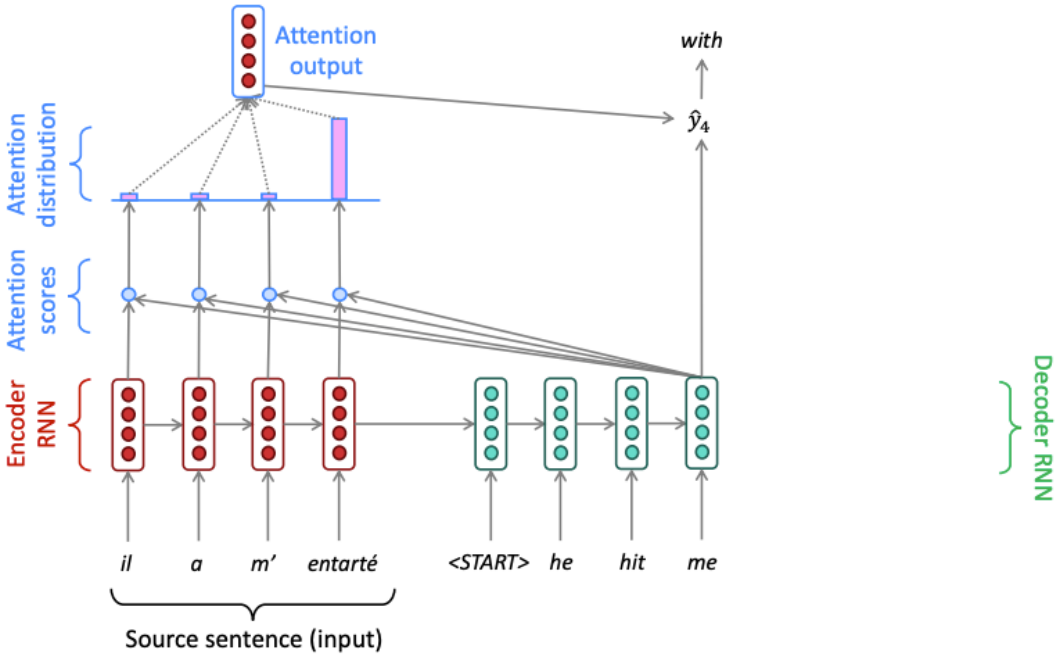
Picture from Stanford

Sequence-to-sequence with attention



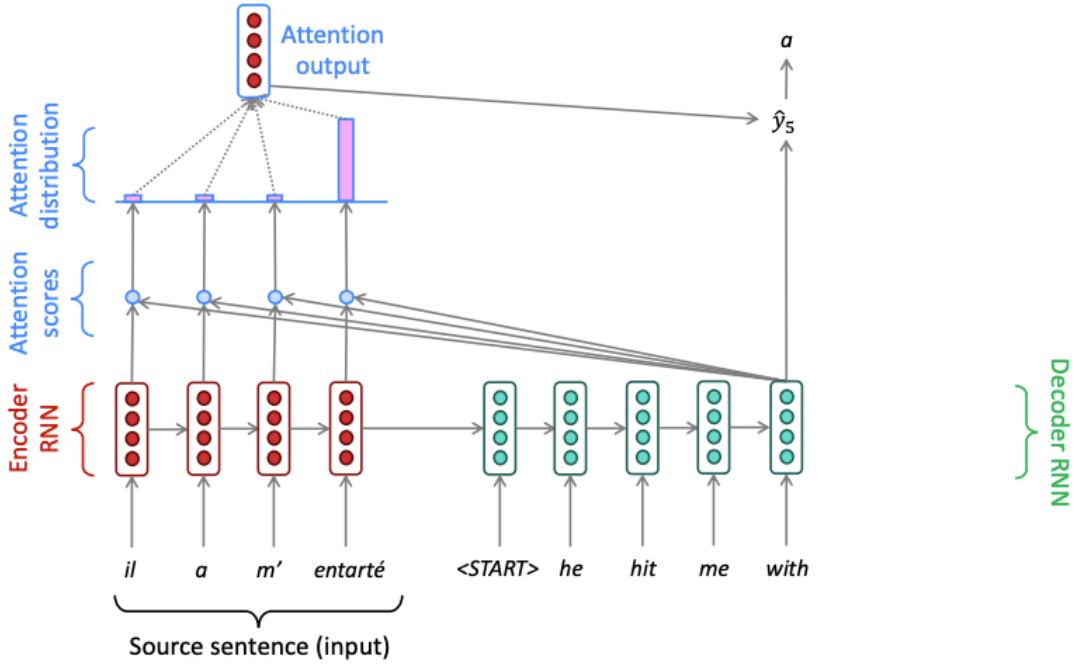
Picture from Stanford

Sequence-to-sequence with attention



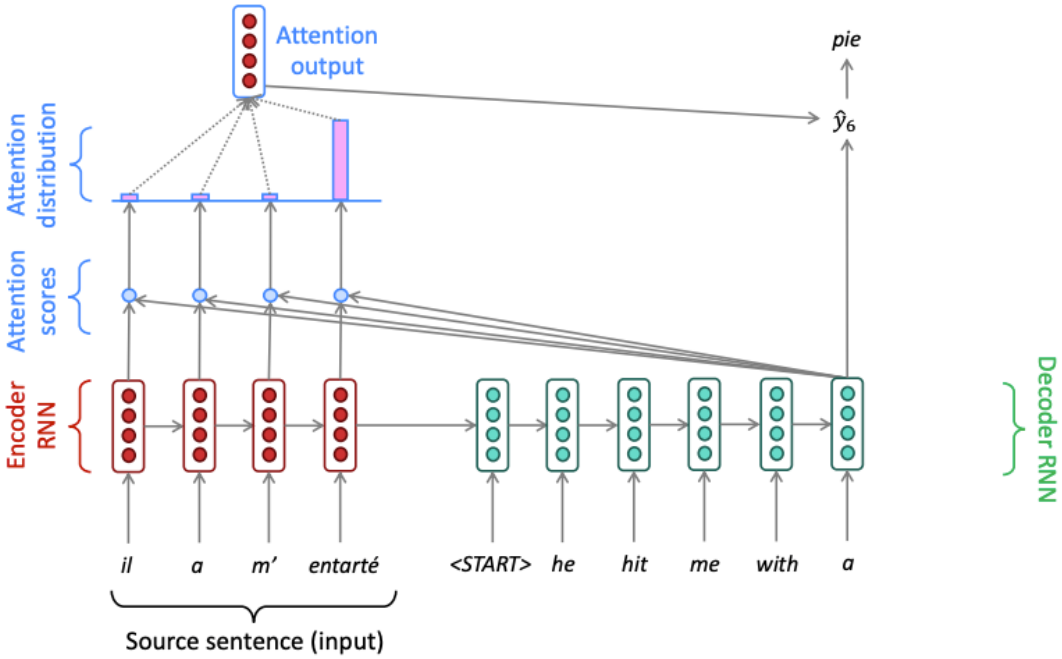
Picture from Stanford

Sequence-to-sequence with attention



Picture from Stanford

Sequence-to-sequence with attention



Picture from Stanford

Attention with Equations

- We have encoder hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_N \in \mathbb{R}^h$
- **On timestep** t, we have decoder hidden state $\mathbf{s}_t \in \mathbb{R}^h$
- We get the attention scores for this step $\mathbf{e}^t$:

$$\mathbf{e}^t = [\mathbf{s}_t^\top \mathbf{h}_1, \ldots, \mathbf{s}_t^\top \mathbf{h}_N] \in \mathbb{R}^N$$

- We take **softmax** to get the attention distribution for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\mathbf{e}^t)$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\mathbf{a}_t$:

$$\mathbf{a}_t = \sum_{i=1}^{N} \alpha_i \mathbf{h}_i \in \mathbb{R}^h$$

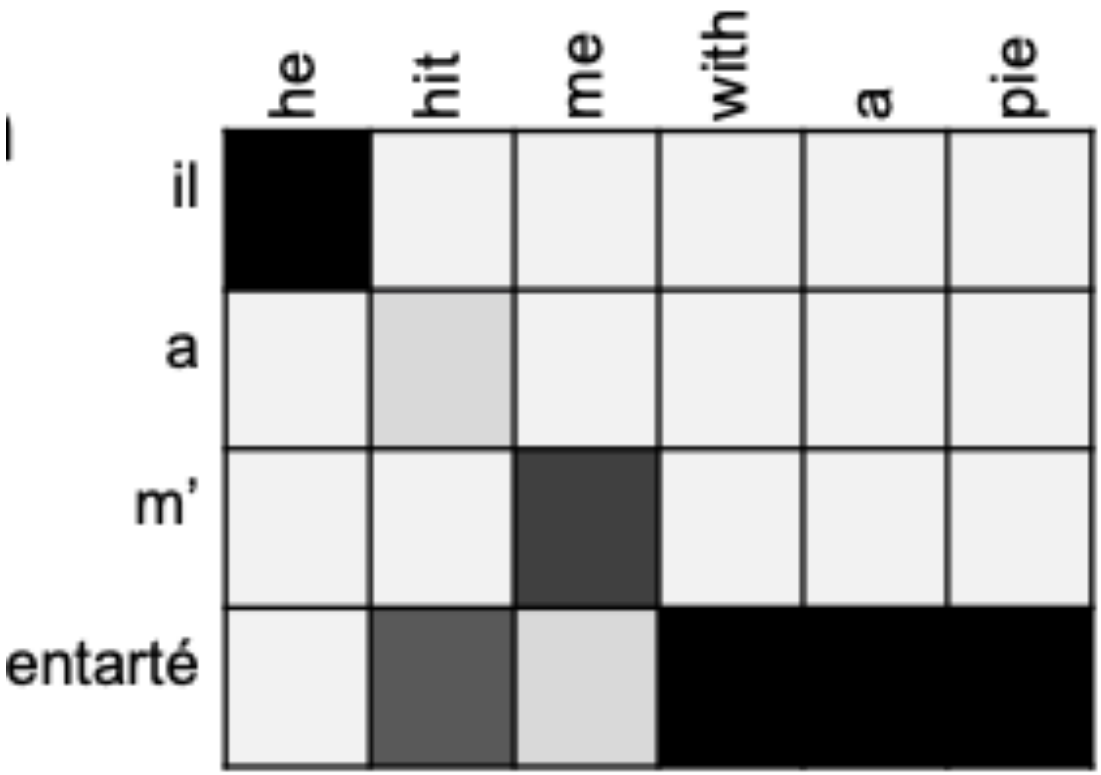
- Finally we concatenate the attention output $\mathbf{a}_t$ with the decoder hidden state $\mathbf{s}_t$ and proceed as in the non-attention seq2seq model:

$$[\mathbf{a}_t, \mathbf{s}_t] \in \mathbb{R}^{2h}$$

Attention 🧐is great! 💪

Attention provides **some interpretability**:

- By inspecting attention distribution, we see what the decoder was focusing on
- We get (soft) alignment for free!
- This is cool because we never explicitly trained an alignment system
- The network just learned alignment by itself



Picture from Stanford

There are *several* attention variants

- We have some **values** $h_1, \ldots, h_n \in \mathbb{R}^{D1}$ and query $\mathbf{s} \in \mathbb{R}^{D2}$
- Attention always involves:
  1. Computing the attention scores $\mathbf{e} \in \mathbb{R}^N$
  2. Taking `softmax` to get attention distribution α:

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^N$$

  3. Using attention distribution to take weighted sum of values:

$$\mathbf{a} = \sum_{i=1}^{N} \alpha_i \mathbf{h}_i \in \mathbb{R}^{D1}$$

  4. thus obtaining the **attention output** $\mathbf{a}$ (sometimes called the **context vector**)

Attention variants

There are several ways you can compute $\mathbf{e} \in \mathbb{R}^N$ from $h_1, \ldots, h_n \in \mathbb{R}^{D1}$ and $\mathbf{s} \in \mathbb{R}^{D2}$.

Dot product attention

Basic dot-product attention: $\mathbf{e}_i = \mathbf{s}^\top \mathbf{h}_i$. Note: this assumes $D1 = D2$ This is the version we saw earlier.

Bilinear attention

Bilinear attention: $\mathbf{e}_i = \mathbf{s}^\top \mathbf{W} \mathbf{h}_i$. $\mathbf{W} \in \mathbb{R}^{D1 \times D2}$ is a weight matrix.

Reduced Rank

As bilinear but low rank: $\mathbf{e}_i = \mathbf{s}^\top (\mathbf{U}^\top \mathbf{V}) \mathbf{h}_i = (\mathbf{U}\mathbf{s})^\top (\mathbf{V}\mathbf{h}_i)$. For low rank matrices $\mathbf{U} \in \mathbb{R}^{k \times D2}$ and $\mathbf{V} \in \mathbb{R}^{k \times D1}$ with $k \ll D1, D2$

Attention is a *general* Deep Learning technique

We have seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.

- However: **You can use attention in many architectures** (not just seq2seq) and **many tasks** (not just MT)

More general definition of attention:

- Given a set of vector **values**, and a vector **query**, attention is a technique to compute a weighted sum of the values, dependent on the query

- We sometimes say that **the query attends to the values**
- For example, in the `seq2seq + attention model`, each decoder hidden state (`query`) attends to all the encoder hidden states (`values`).

Intuition about Attention 🧐

- The weighted sum is a **selective summary** of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a **fixed-size representation** of an arbitrary set of representations (the values), **dependent on some other representation** (the query).