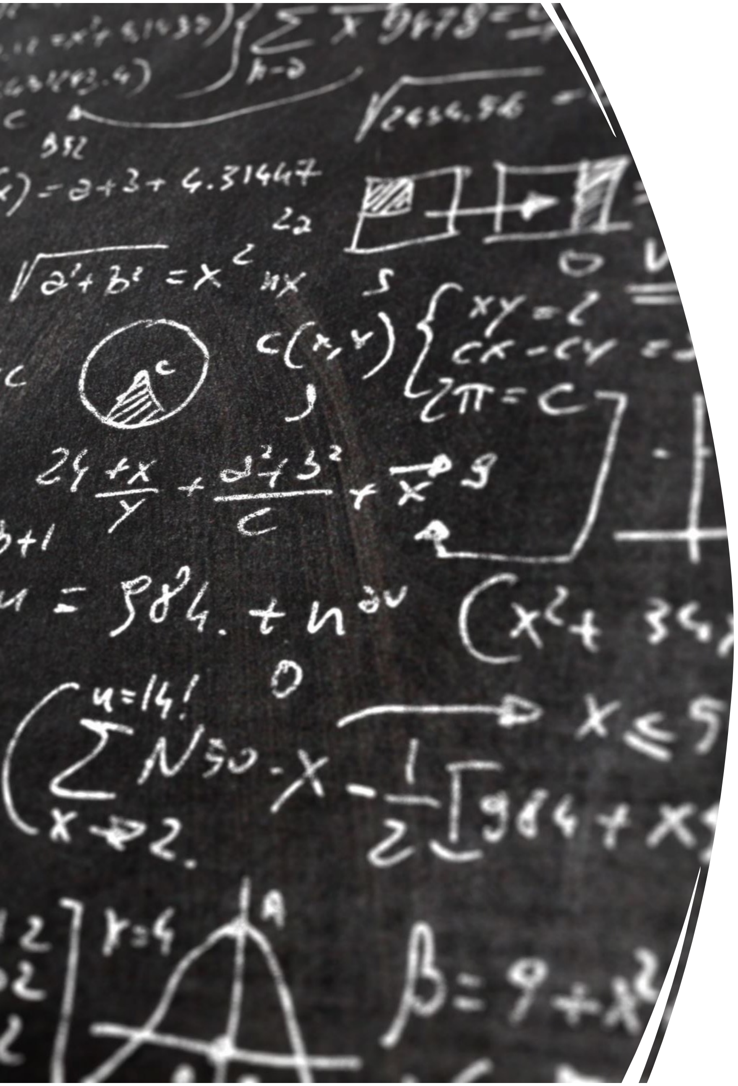


CS 383 – Computational Text Analysis

Lecture 14
LSTMs, Sentence-Representations, Probing, Attention

Adam Poliak

03/13/2023

Slides adapted from Jordan Boyd-Graber, Daniel Khashabi, Yoav Goldberg, Chris Manning

Announcements

- Reading 05
  - CTA/TADA/CSS papers using Word Embeddings
  - Look at piazza for deadline – Wednesday after spring break
  - No programming portion
- Reading 06
  - Will be back to Mondays
- Office hours this week:
  - Normal Thursday slot

Outline

Recap - RNNs

LSTM

Sentence Representations/Probing

Attention

Transformer

Machine Learning in a nutshell

In a ML model, what are we training?

- **Parameters!**

How do we train parameters in supervised learning?

train parameters == figure out values for the parameters

- Update weights by using them to make predictions and seeing **how far off our predictions** are
  - **Loss function!**

Algorithm to learn weights?

- **SGD**
- Others exist but not covering them

RNN - motivation

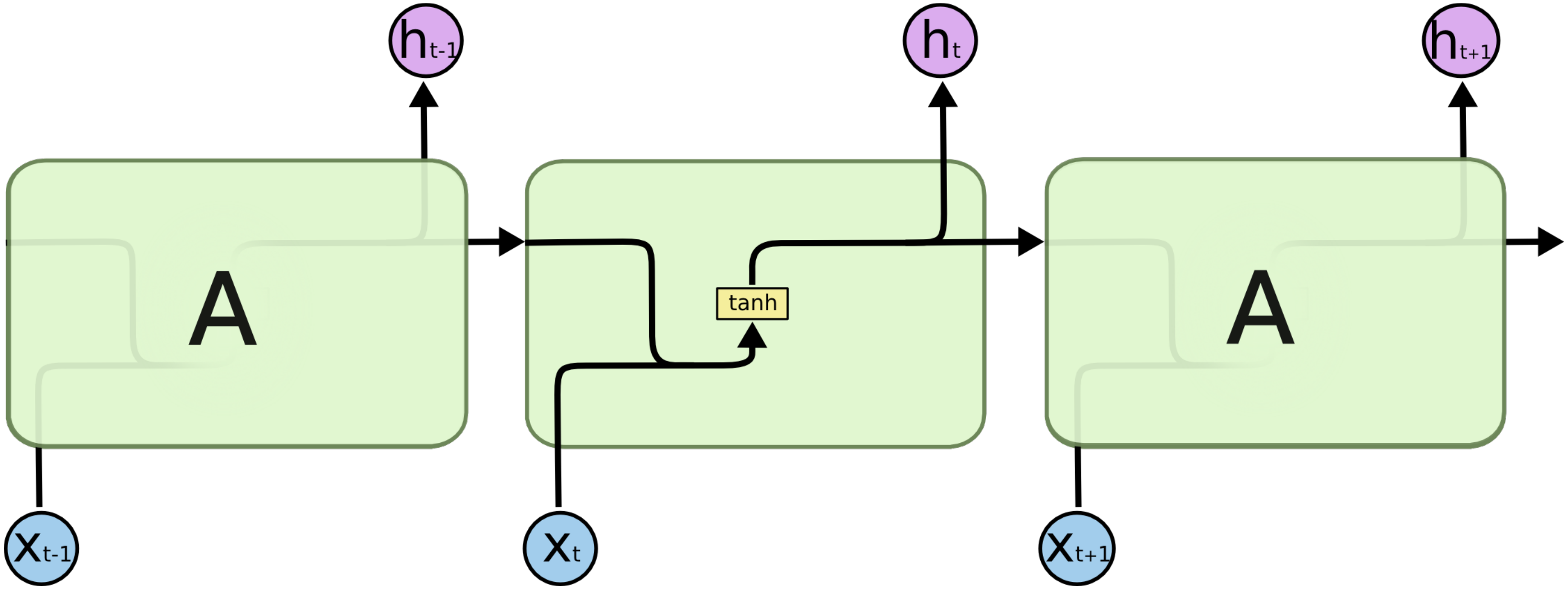
How can we model a **long** (possibly infinite) context using a finite **model?**

Recursion

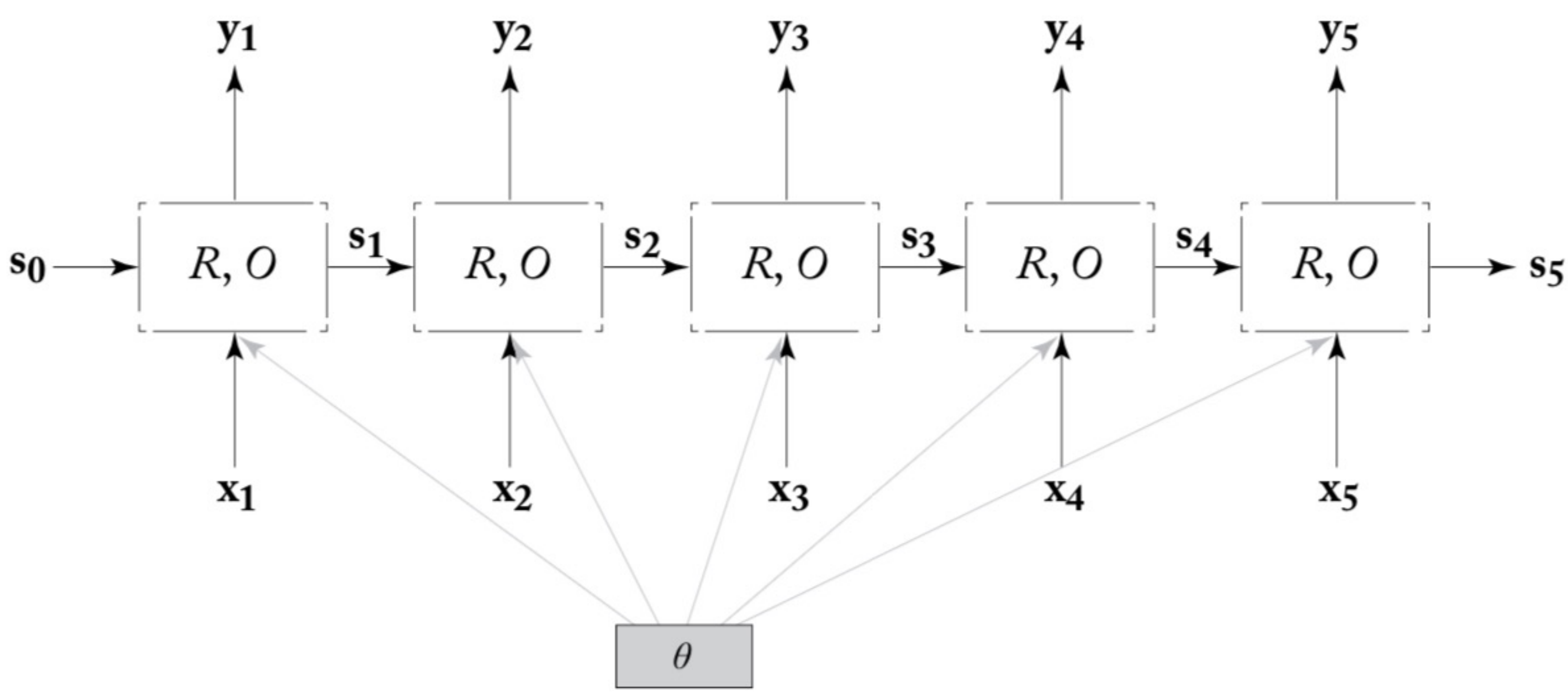
Recurrent Neural Networks are a family of NNs that learn sequential data via **recursive dynamics**

RNN internal

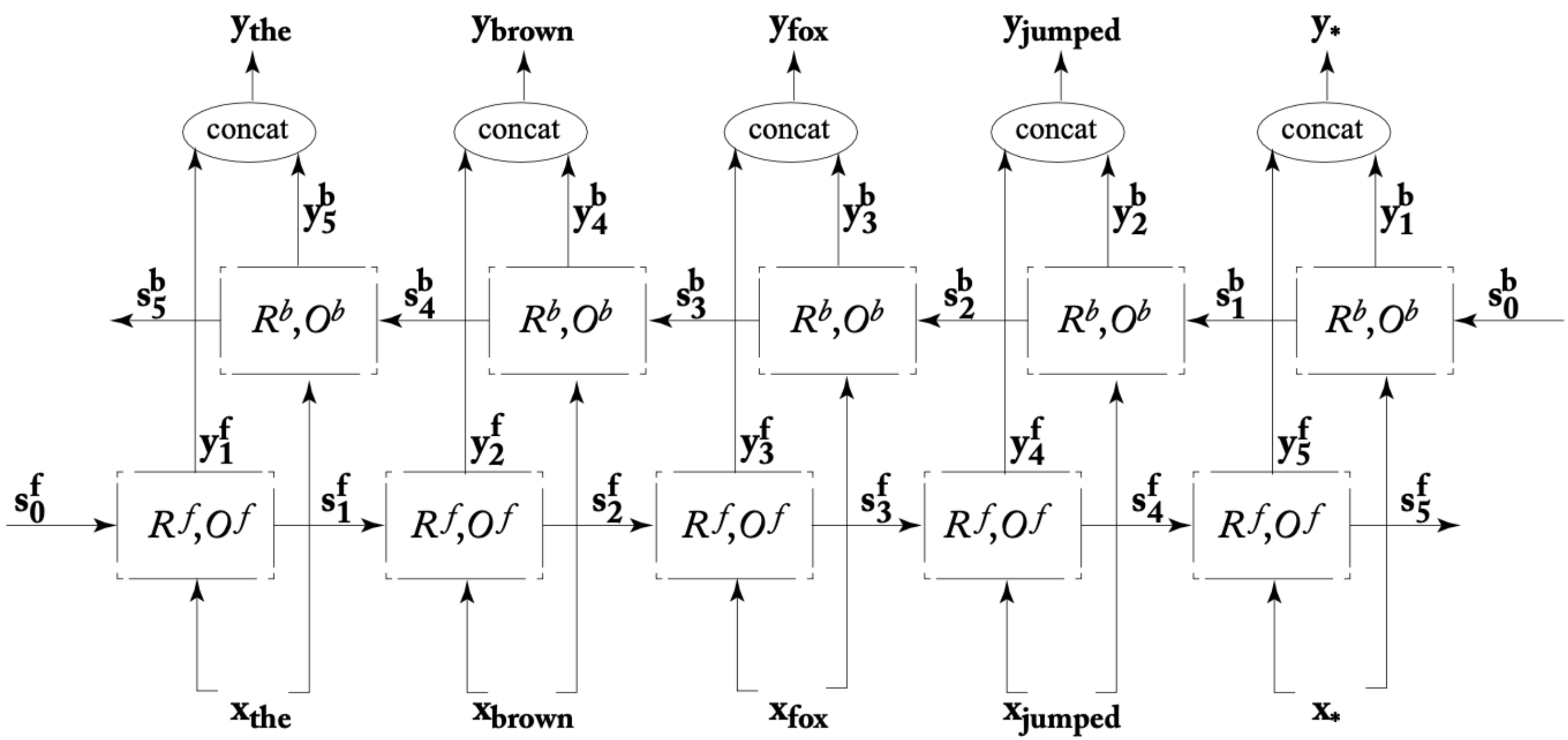
$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$



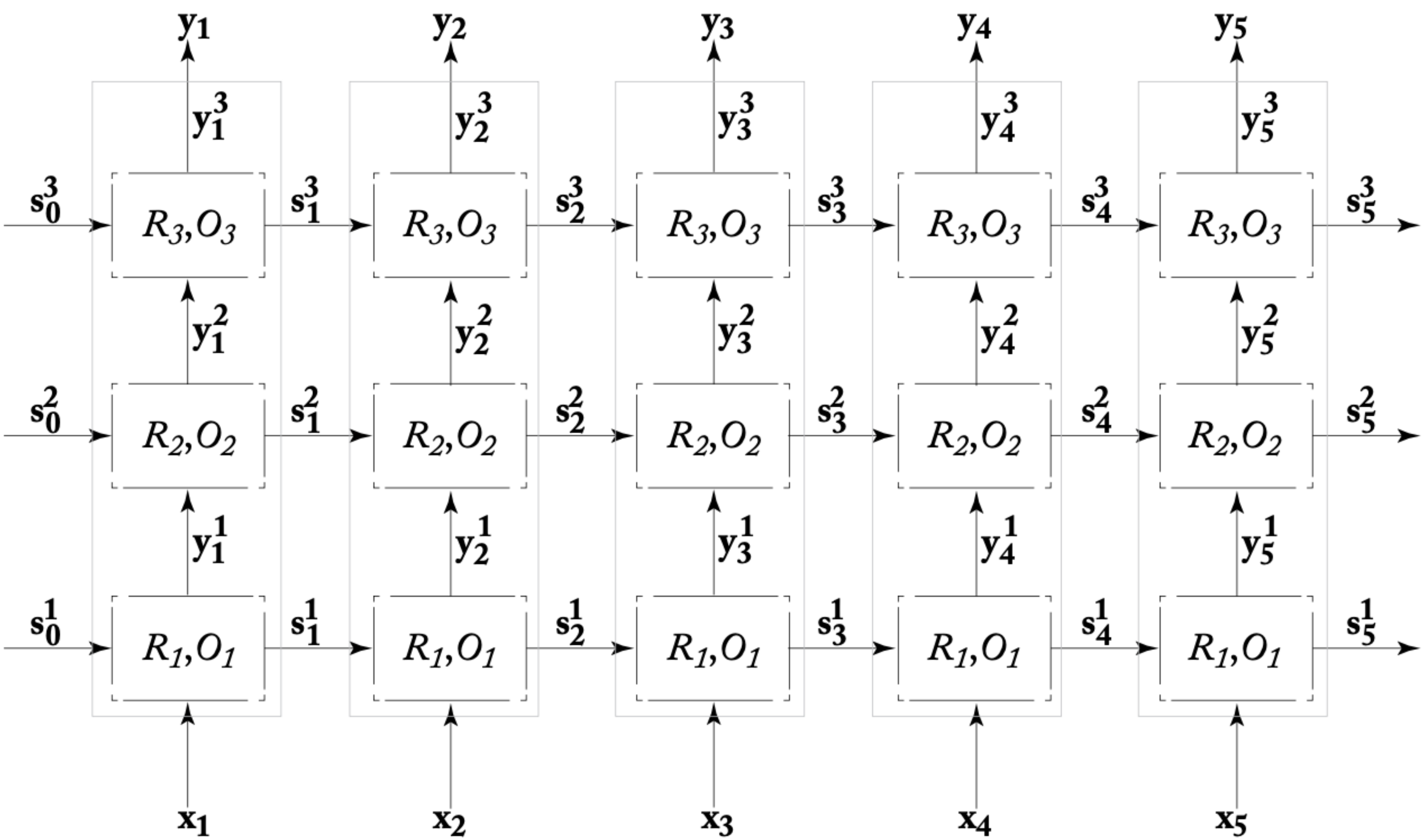
How else can we expand this?



Bi-directional



Stack more layers



Pytorch - nn.RNN

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 RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **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 RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

Recap: RNN’s: Pros and Cons

Pros:

- Model size doesn’t increase for longer inputs.
  - Reusing same parameters
- Computation can use information from many previous steps

Cons:

- Slow computation
- Can forget longer history/context
- Vanishing/exploding gradients

RNNs – long input

RNNs can remember anything (in theory)

Sometimes its important to forget

Solution: Long-Short Term Memory (LSTM)

Outline

Recap - RNNs

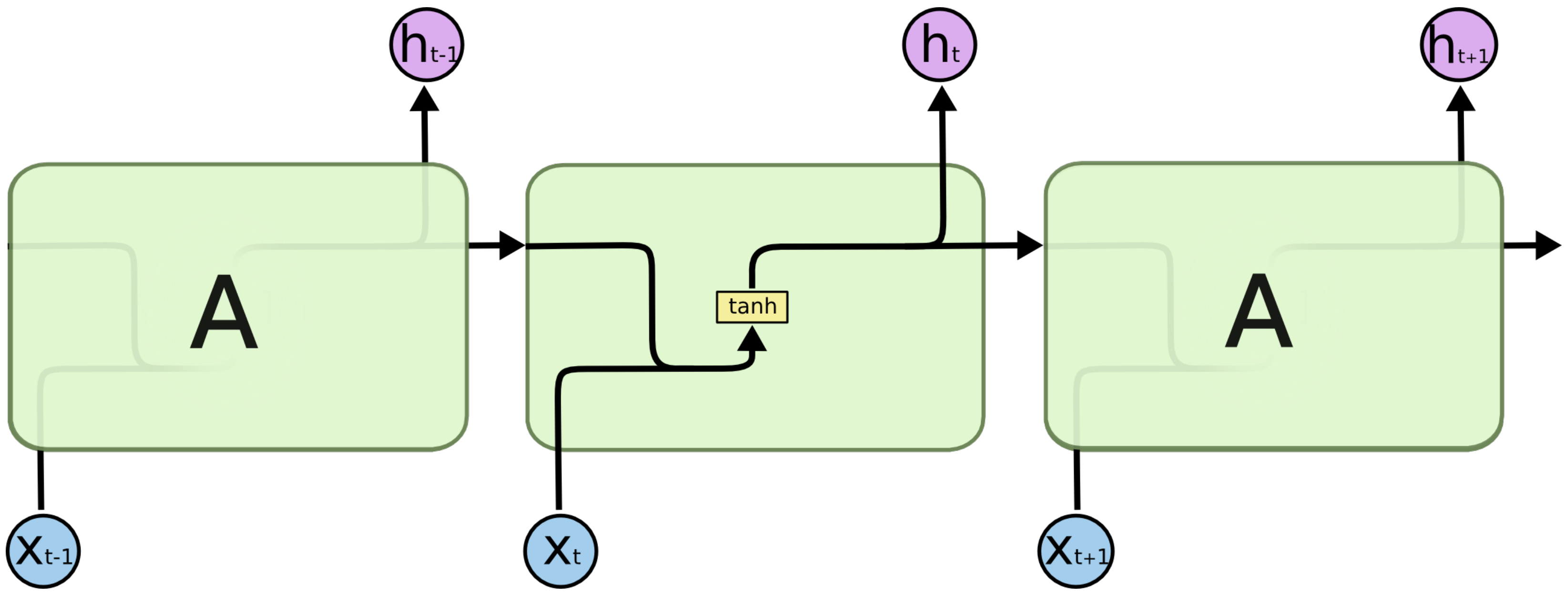
LSTM

Sentence Representations

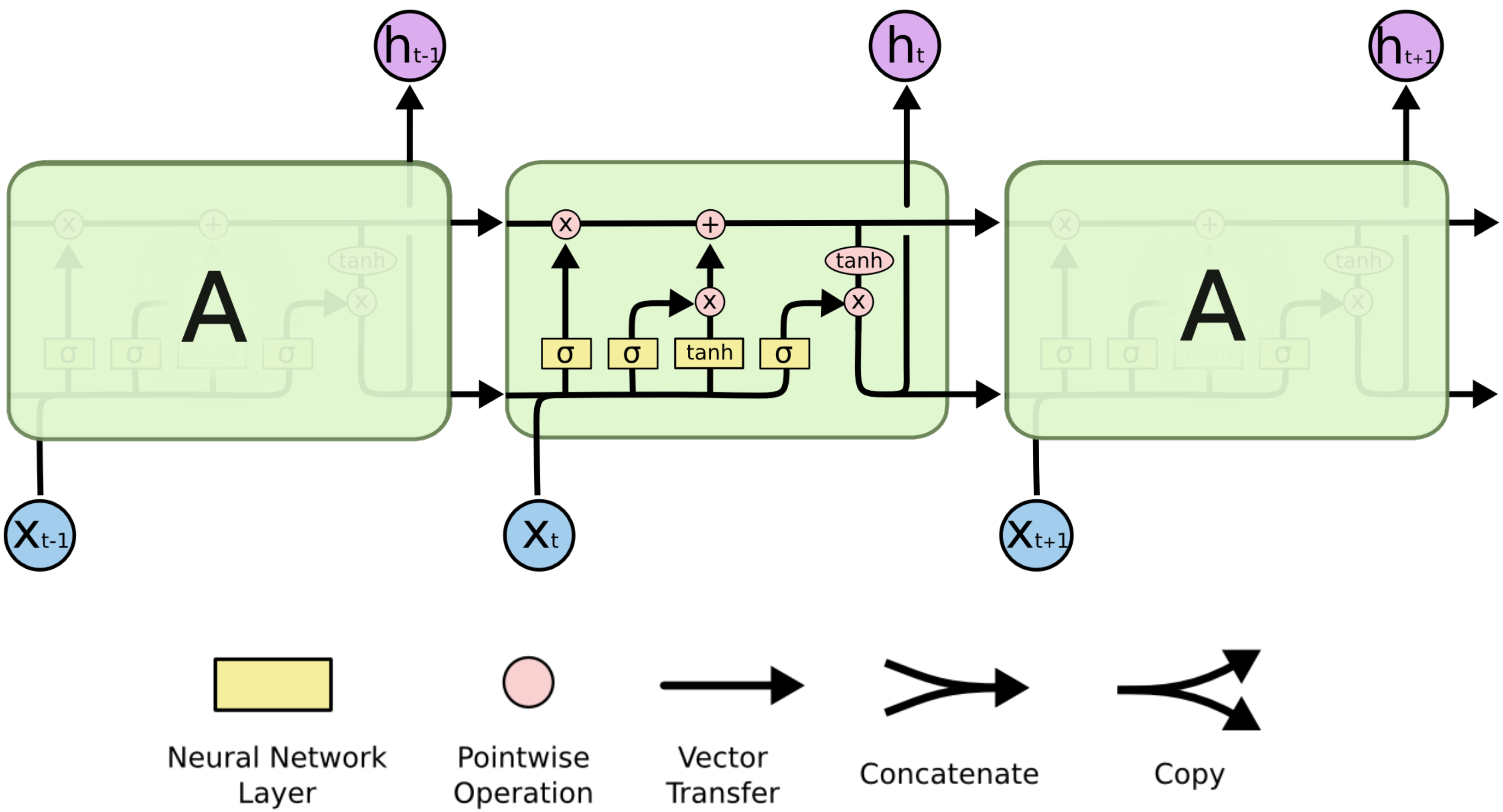
Attention

Transformer

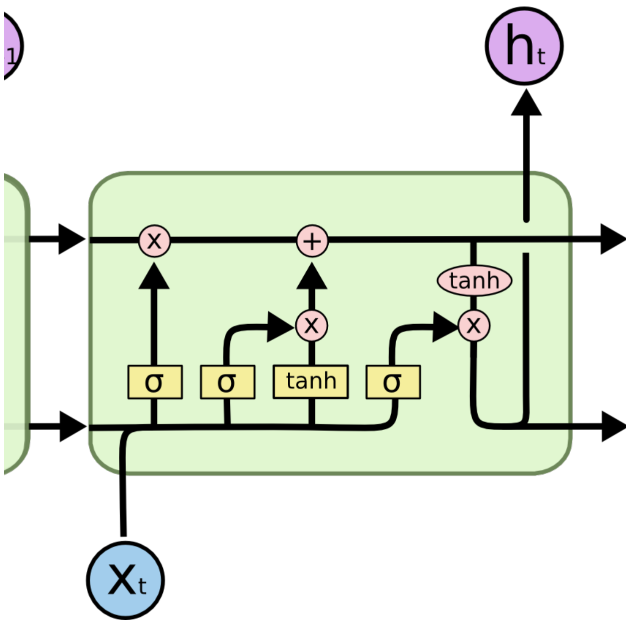
RNN internal



LSTM internal



LSTM internal



$$\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) = [\boldsymbol{c_j}; \boldsymbol{h_j}]$$

$$\boldsymbol{c_j} = \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z}$$

$$\boldsymbol{h_j} = \boldsymbol{o} \odot \tanh(\boldsymbol{c_j})$$

$$\boldsymbol{i} = \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}})$$

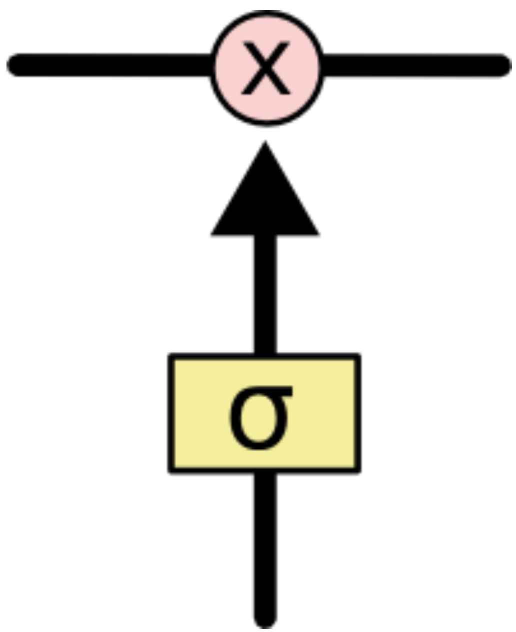
$$\boldsymbol{f} = \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}})$$

$$\boldsymbol{o} = \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}})$$

$$\boldsymbol{z} = \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})$$

$$\boldsymbol{y_j} = O_{\text{LSTM}}(\boldsymbol{s_j}) = \boldsymbol{h_j}$$

LSTM's rely on gates

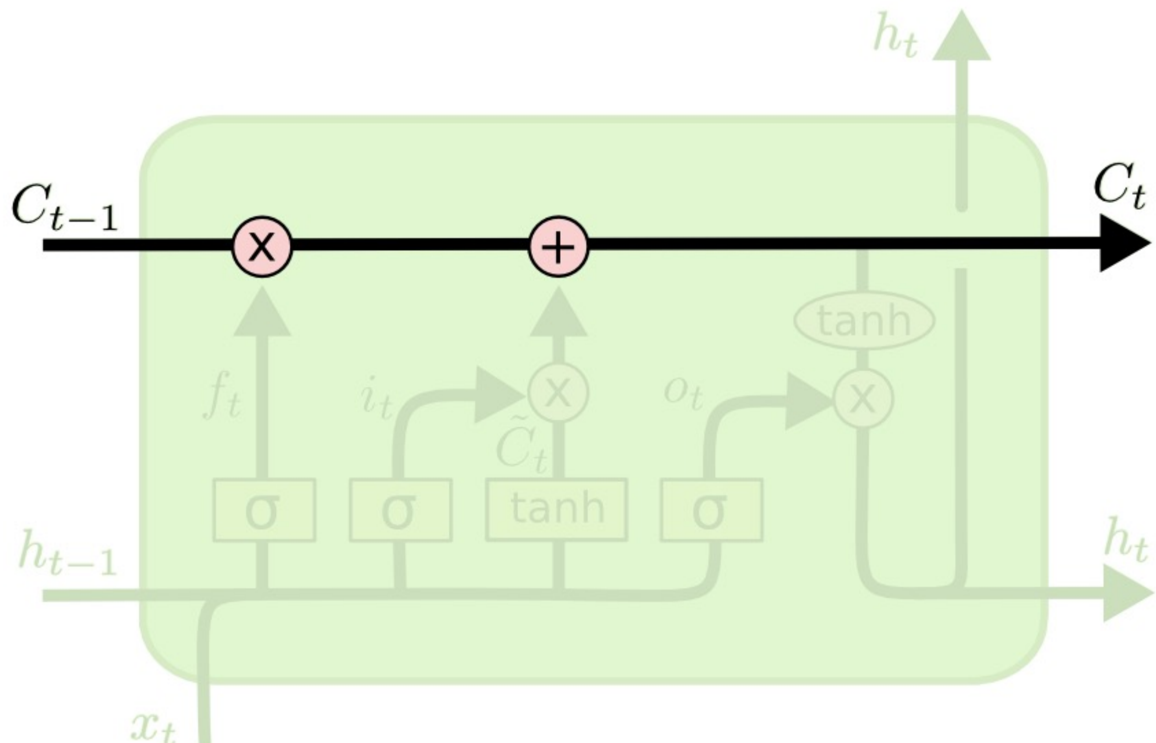


- Multiply input by value in 0,1]
- Zero means forget everything
- 1 means carry everything through (unchanged)

- 4 gates used in LSTM

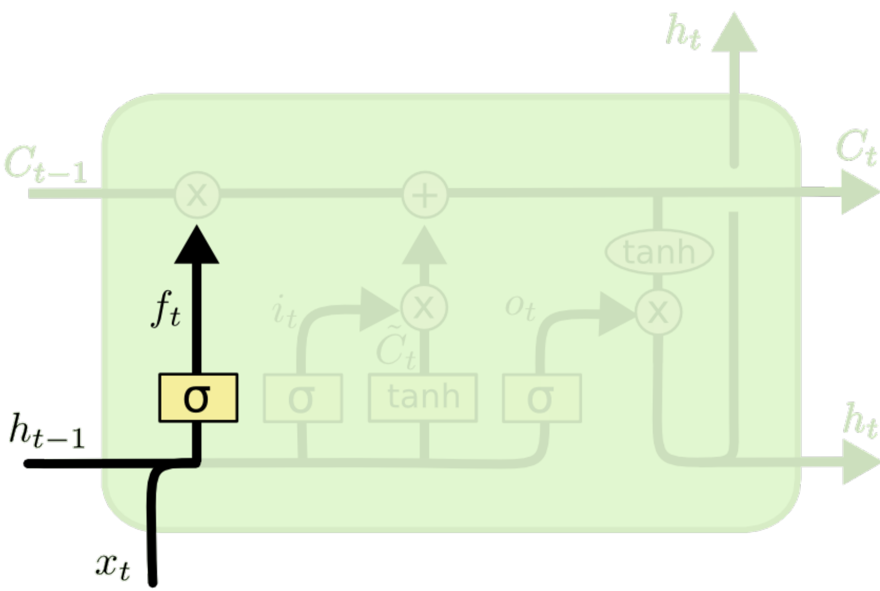
LSTM gates: cell state

- Passes the memory through the cell



LSTM gates: forget

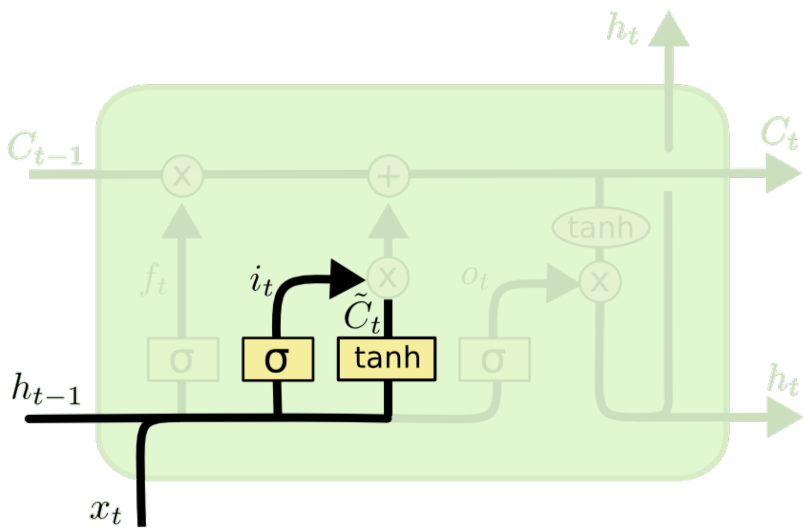
- Can decide to forget the previous state h_{t-1}



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

LSTM gates: update

- Compute new contribution to cell state based on hidden state and input.

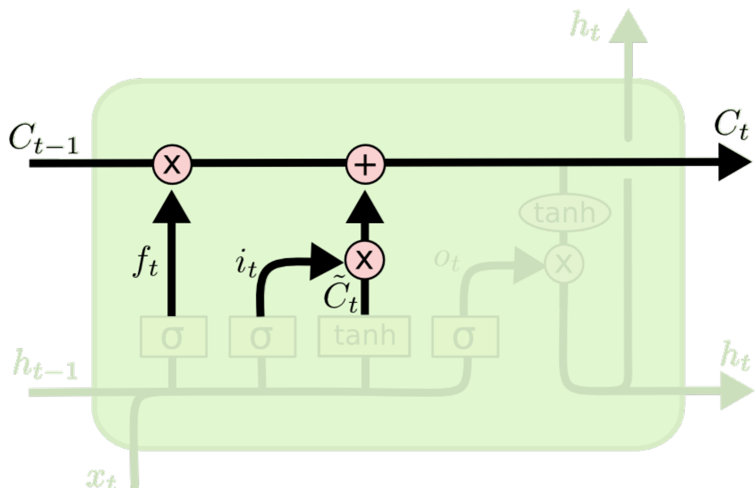


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \;+\; b_i\right)$$

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

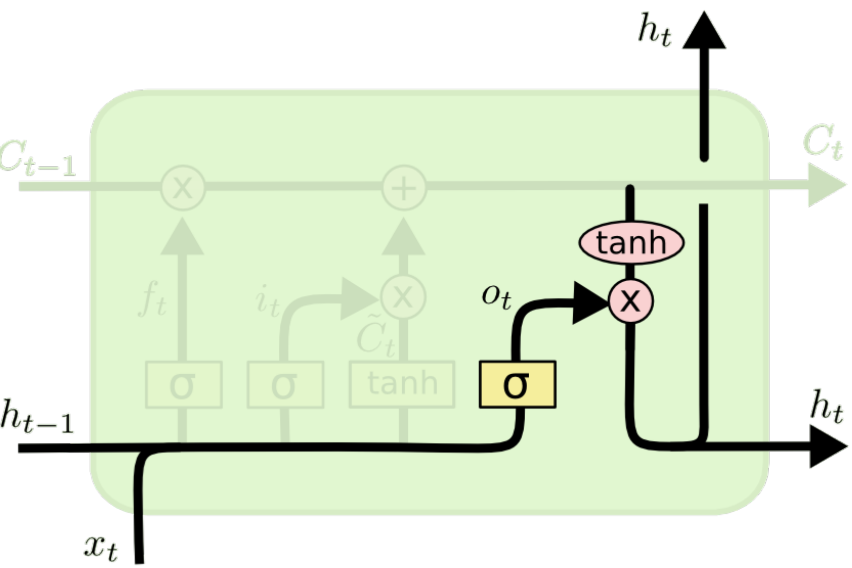
LSTM gates: update (interpolate)

- Can decide to forget the previous state h_{t-1}



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

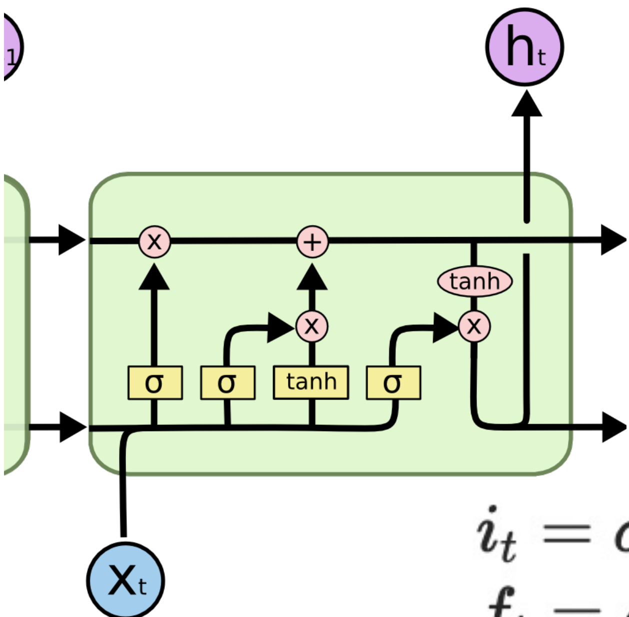
LSTM output (hidden)



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

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

LSTM internal



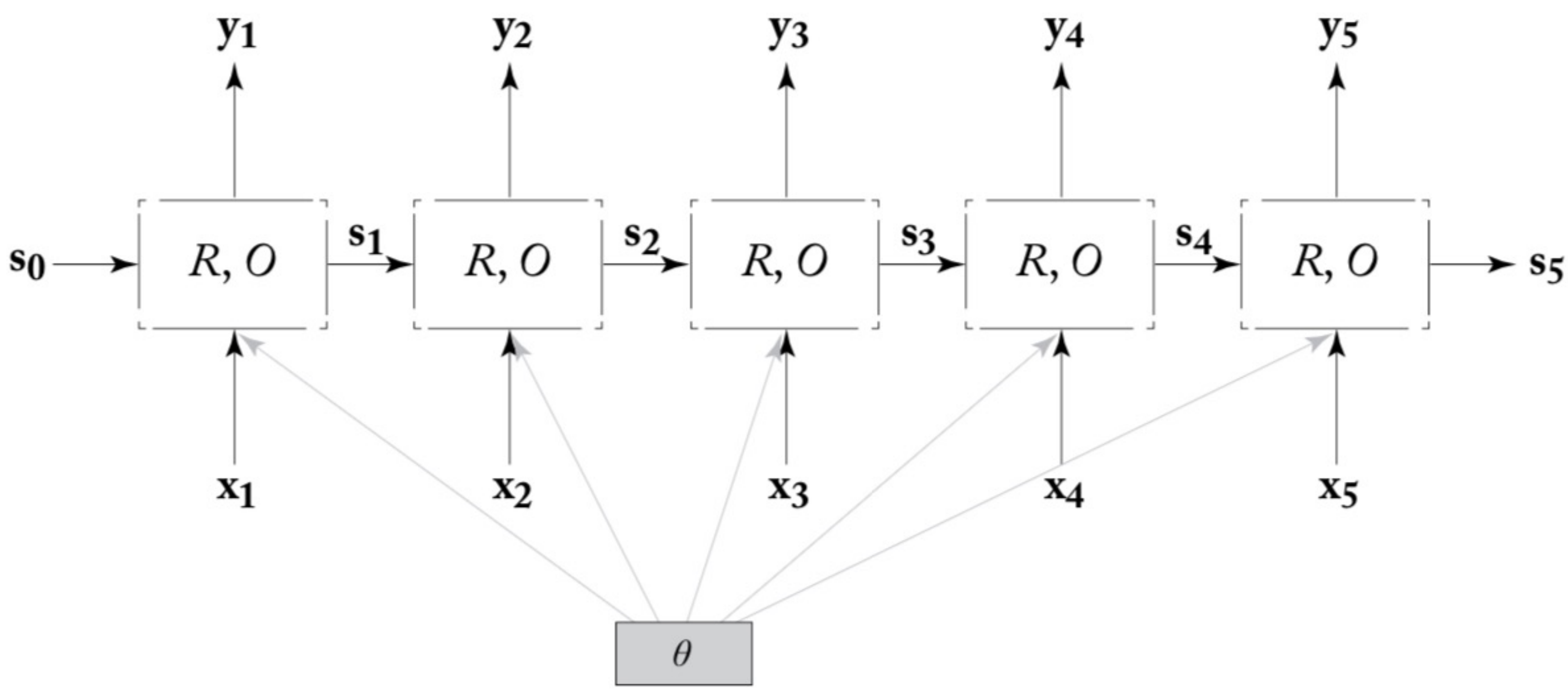
$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$
$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t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

Pytorch - nn.LSTM

Parameters:

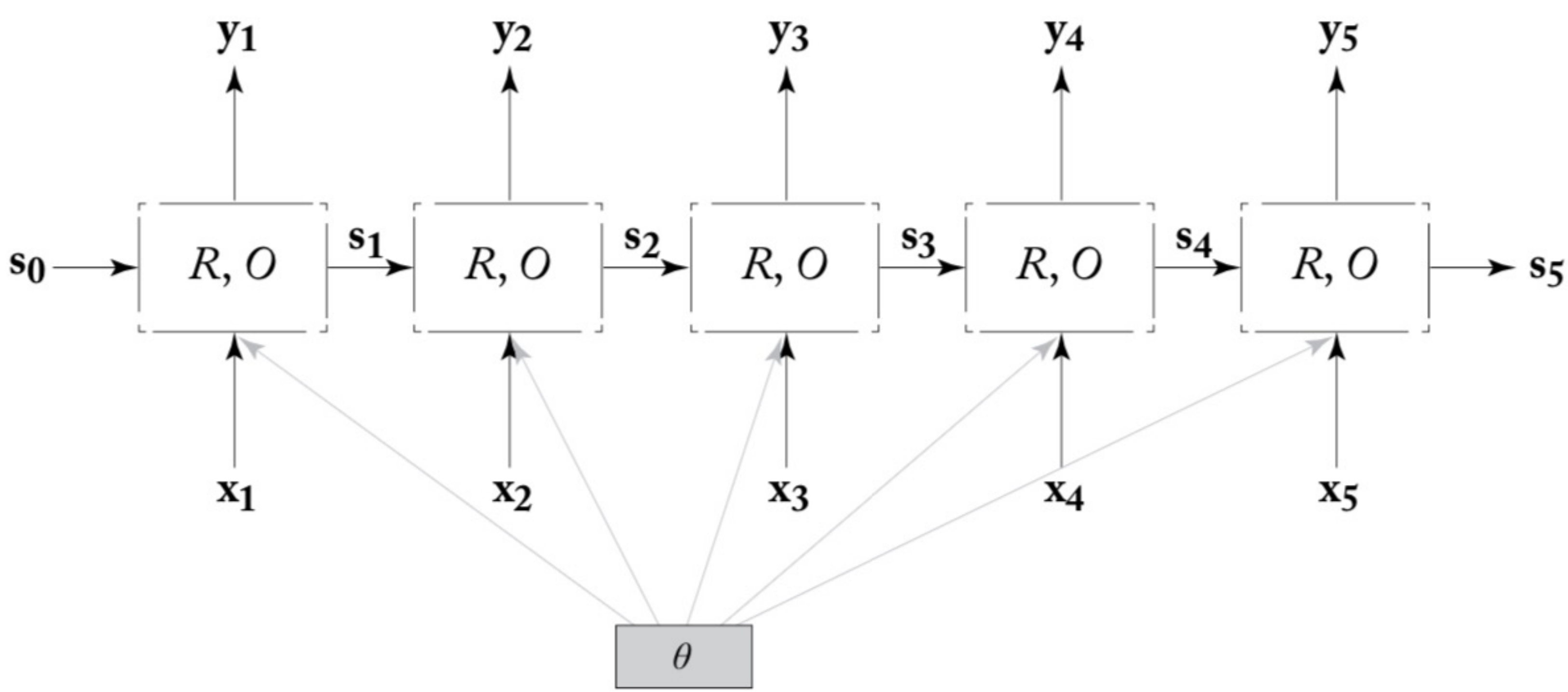
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking 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

Extracting representation from RNN layer



Extracting representation from RNN layer

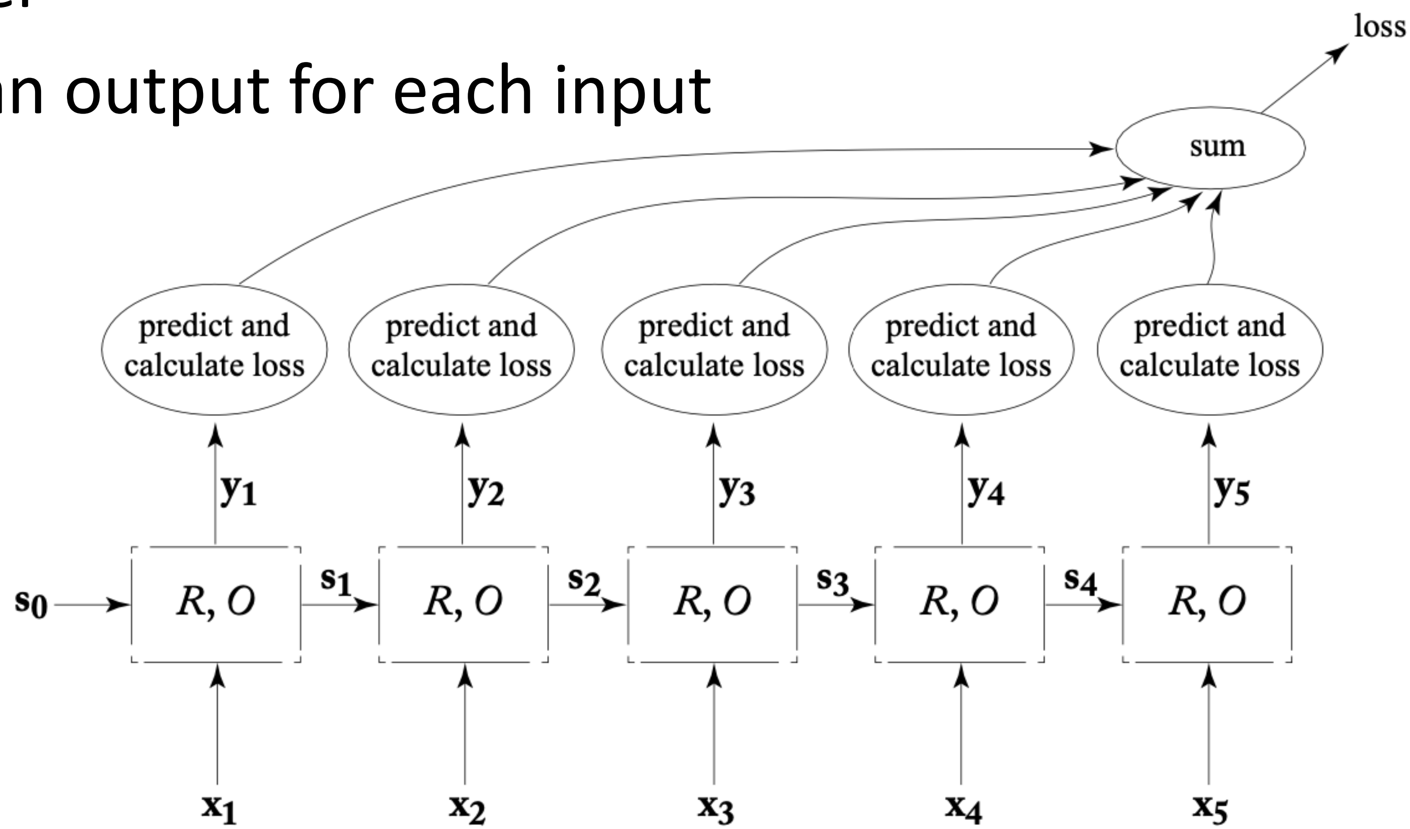
How might we combine the output layers



Extracting representation from RNN layer

Transducer

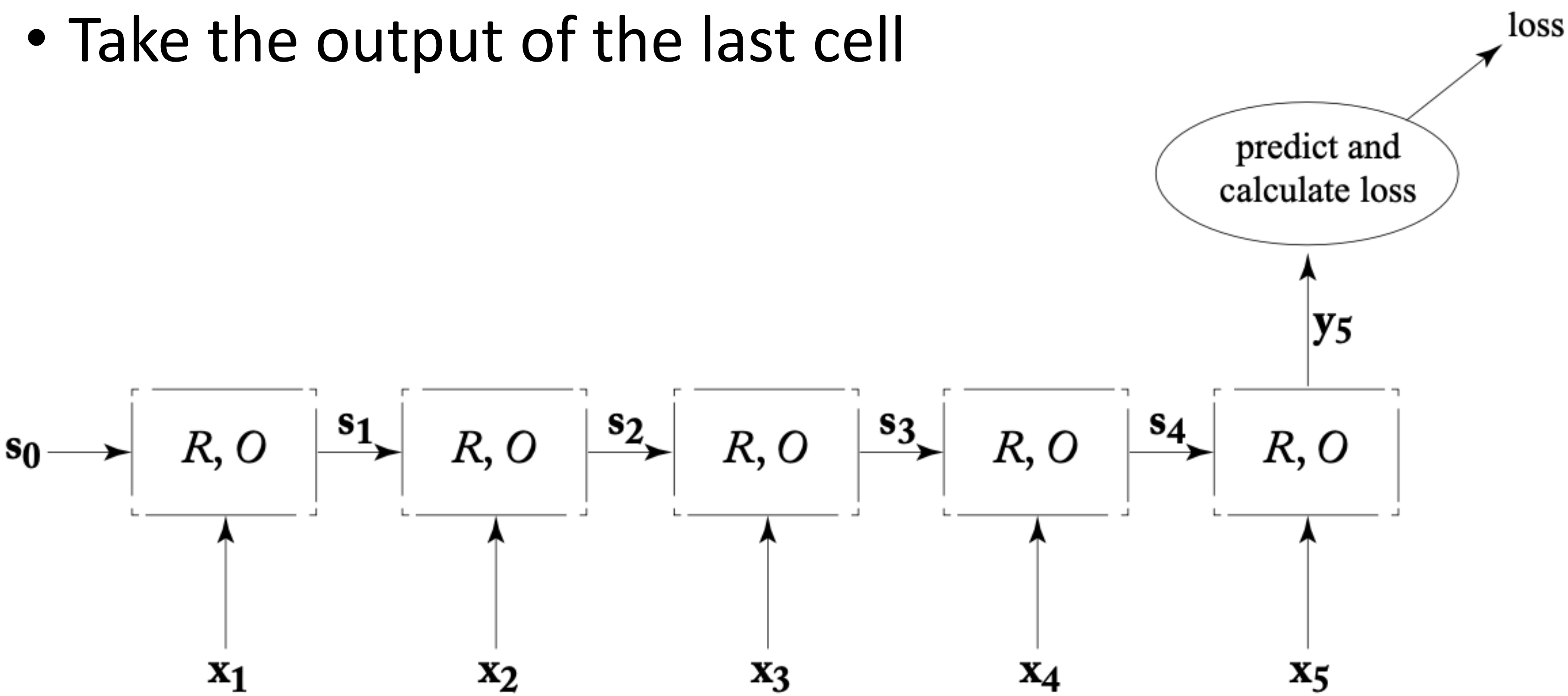
- Create an output for each input



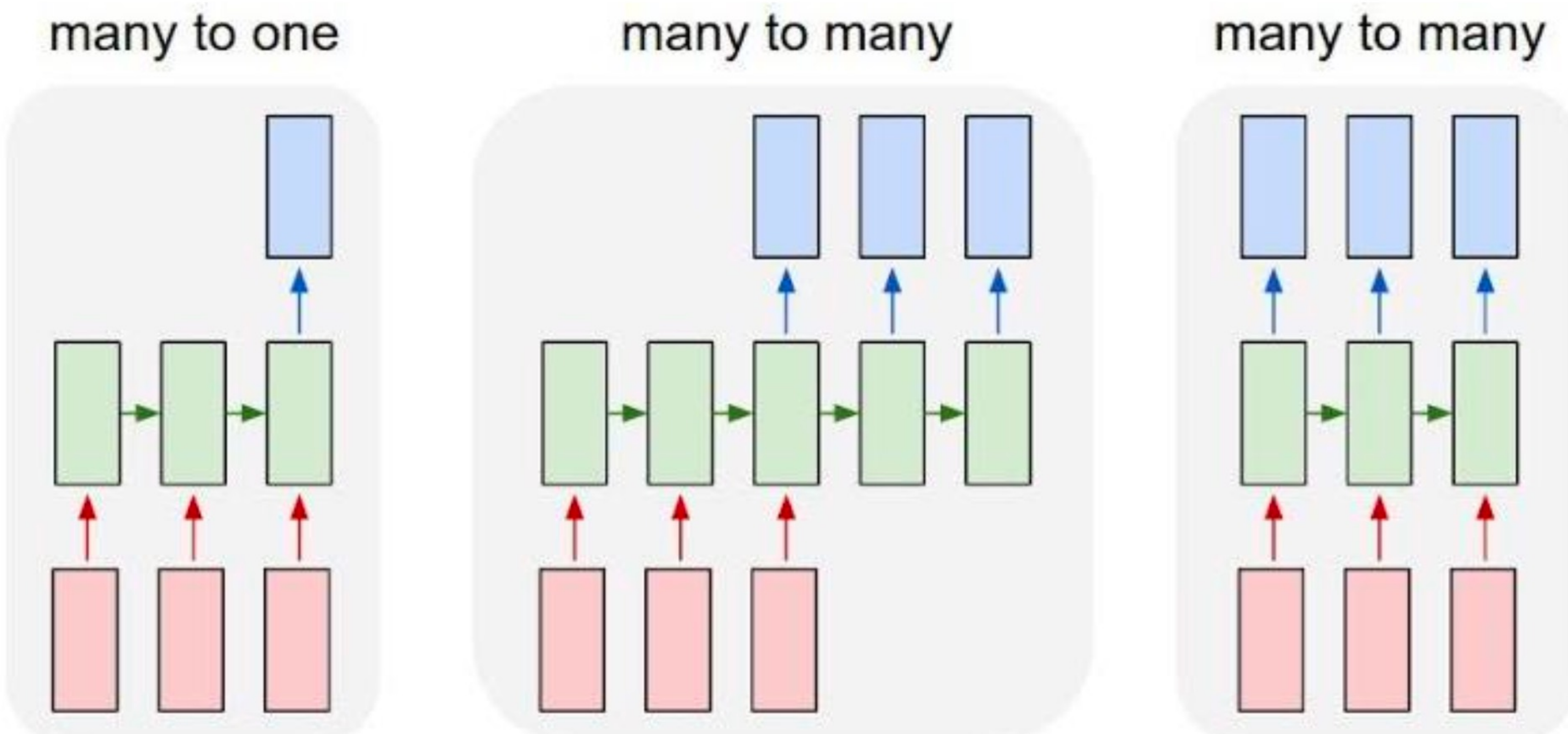
Extracting representation from RNN layer

Acceptor/encoder

- Take the output of the last cell



RNNs applied to NLP tasks



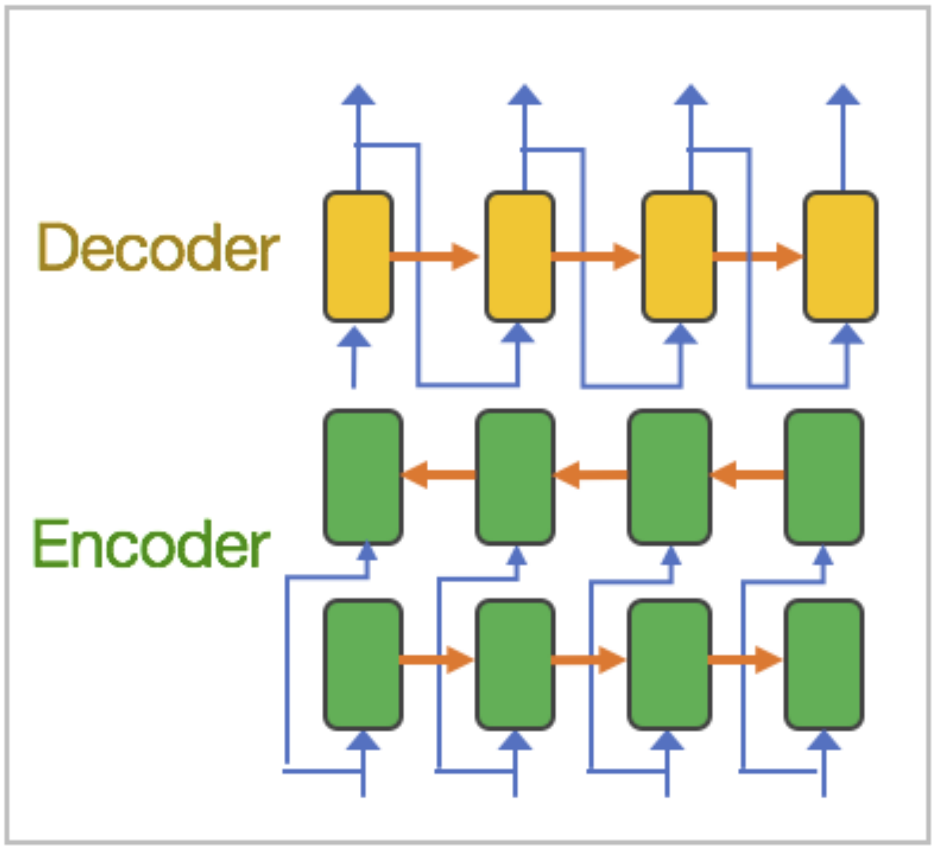
Text Classification

Language Modeling

POS Tags

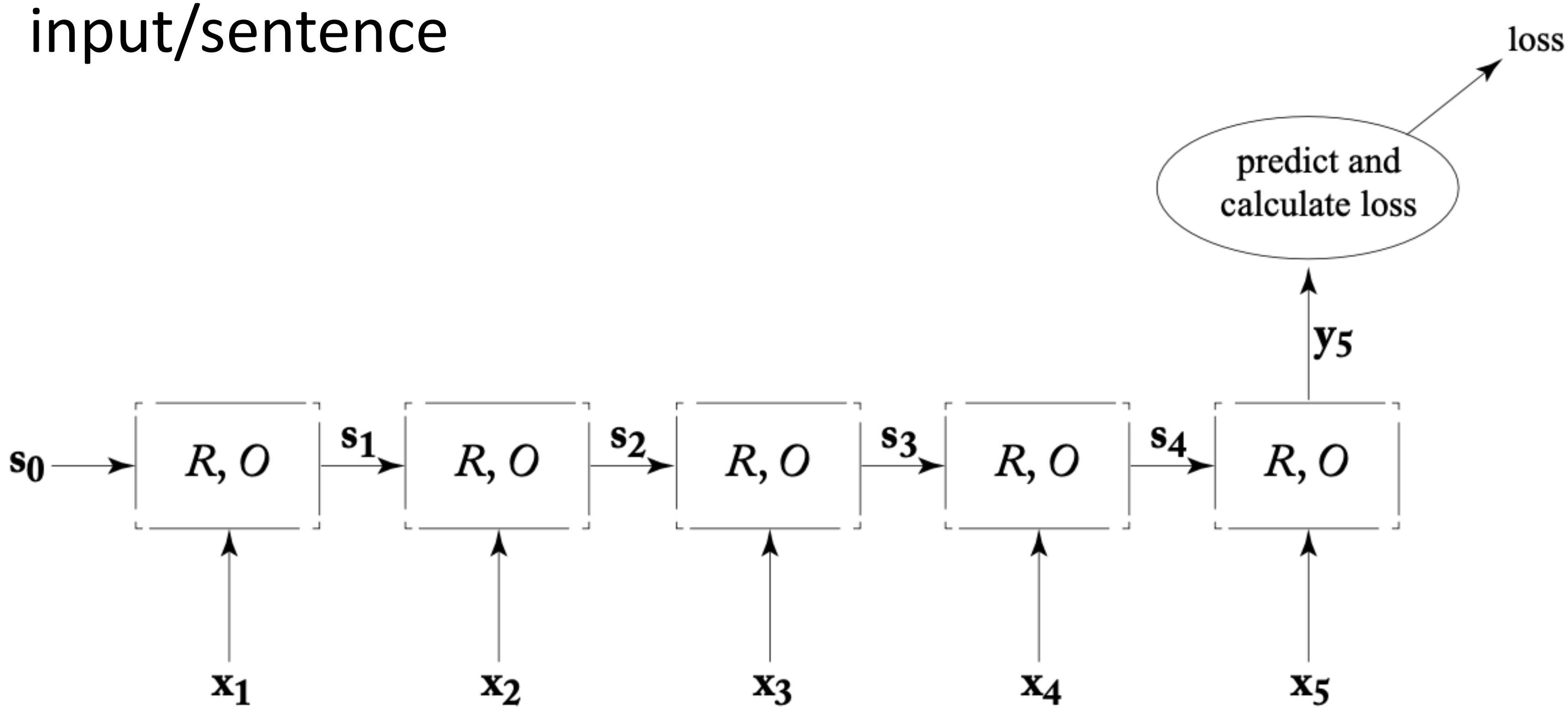
Encoder-Decoder model

Called seq2seq model when encoder sequence and decode a sequence



Encoder-Decoder model

We can view $y5$ as the vector representation of our input/sentence



Outline

Recap - RNNs

LSTM

Sentence Representations/Probing

Attention

Transformer

Fine-grained Analysis of Sentence Embeddings Using Auxiliary Prediction Tasks

Yossi Adi[1,2], Einat Kermany[2], Yonatan Belinkov[3], Ofer Lavi[2], Yoav Goldberg[1]

ABSTRACT

There is a lot of research interest in encoding variable length sentences into fixed length vectors, in a way that preserves the sentence meanings. Two common methods include representations based on averaging word vectors, and representations based on the hidden states of recurrent neural networks such as LSTMs. The sentence vectors are used as features for subsequent machine learning tasks or for pre-training in the context of deep learning. However, not much is known about the properties that are encoded in these sentence representations and about the language information they capture.

We propose a framework that facilitates better understanding of the encoded representations. We define prediction tasks around isolated aspects of sentence structure (namely sentence length, word content, and word order), and score representations by the ability to train a classifier to solve each prediction task when using the representation as input. We demonstrate the potential contribution of the approach by analyzing different sentence representation mechanisms. The analysis sheds light on the relative strengths of different sentence embedding methods with respect to these low level prediction tasks, and on the effect of the encoded vector's dimensionality on the resulting representations.

What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties

Alexis Conneau
Facebook AI Research
Université Le Mans
aconneau@fb.com

German Kruszewski
Facebook AI Research
germank@fb.com

Guillaume Lample
Facebook AI Research
Sorbonne Universités
glample@fb.com

Loïc Barrault
Université Le Mans
loic.barrault@univ-lemans.fr

Marco Baroni
Facebook AI Research
mbaroni@fb.com

Abstract

Although much effort has recently been devoted to training high-quality sentence embeddings, we still have a poor understanding of what they are capturing. “Downstream” tasks, often based on sentence classification, are commonly used to evaluate the quality of sentence representations. The complexity of the tasks makes it however difficult to infer what kind of information is present in the representations. We introduce here 10 probing tasks designed to capture simple linguistic features of sentences, and we use them to study embeddings generated by three different encoders trained in eight distinct ways, uncovering intriguing properties of both encoders and training methods.

Conneaue et al 2018

Trained NN's to:

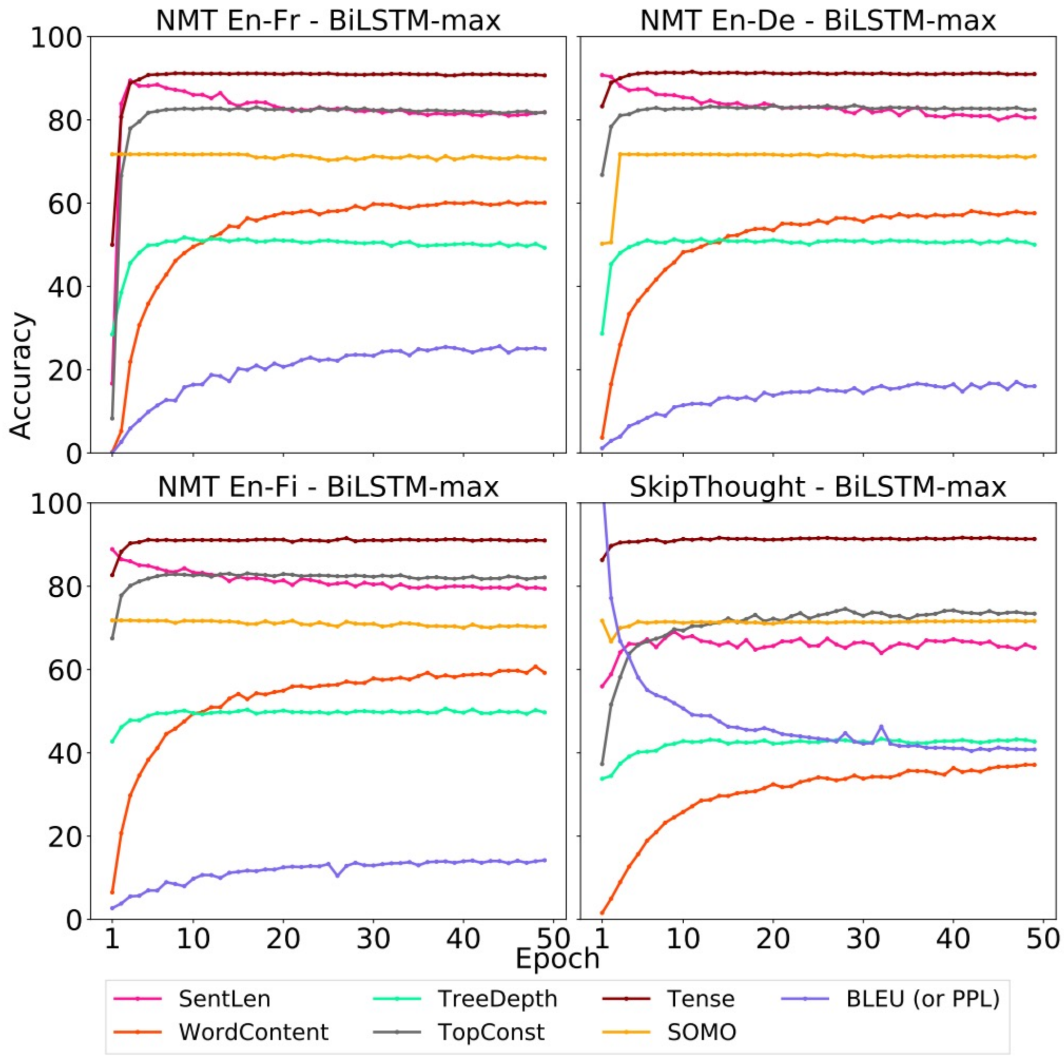
- Translate text
- Predict the next sentence
- Determine if one sentence can be inferred from another (Natural Language Inference)
- Random encoder as baseline

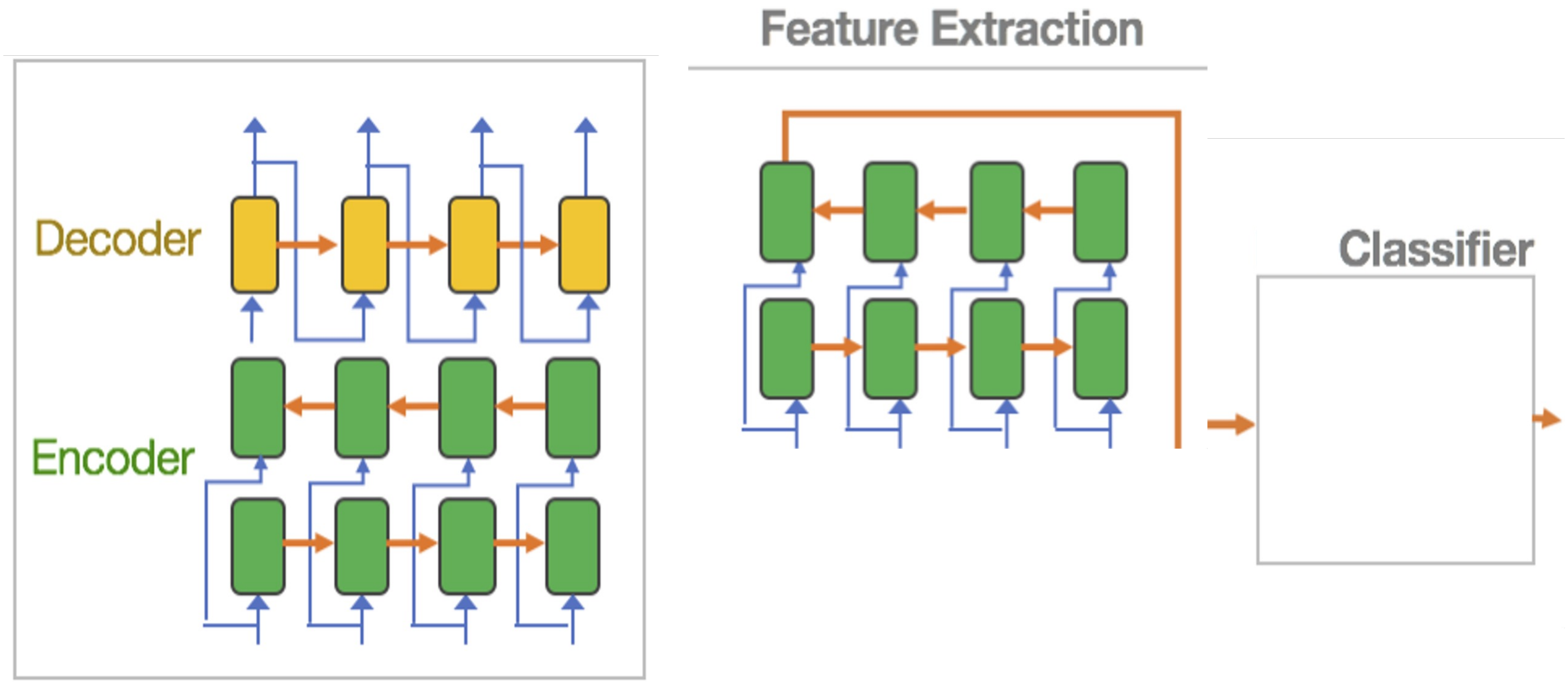
Conneaue et al 2018

Used representations from these encoder to predict

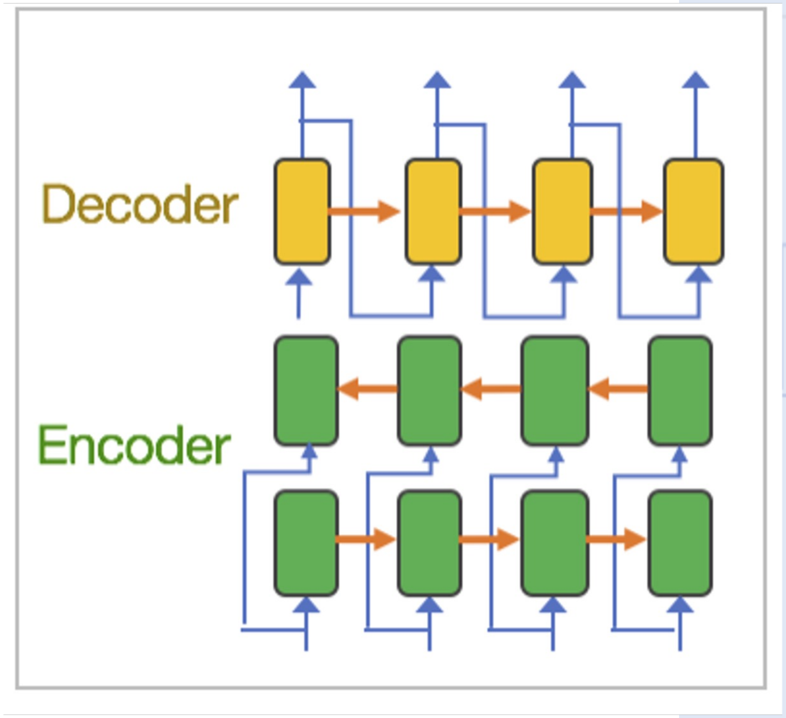
- Surface form information:
  - The length of the sentence
- Syntactic information:
  - If two words in a sentence have been swapped
    - "What you are doing out there?" (Bshift)
- Semantic information:
  - Tense
  - Semantic Odd Man Out
    - Replaced random verb or noun in a sentence

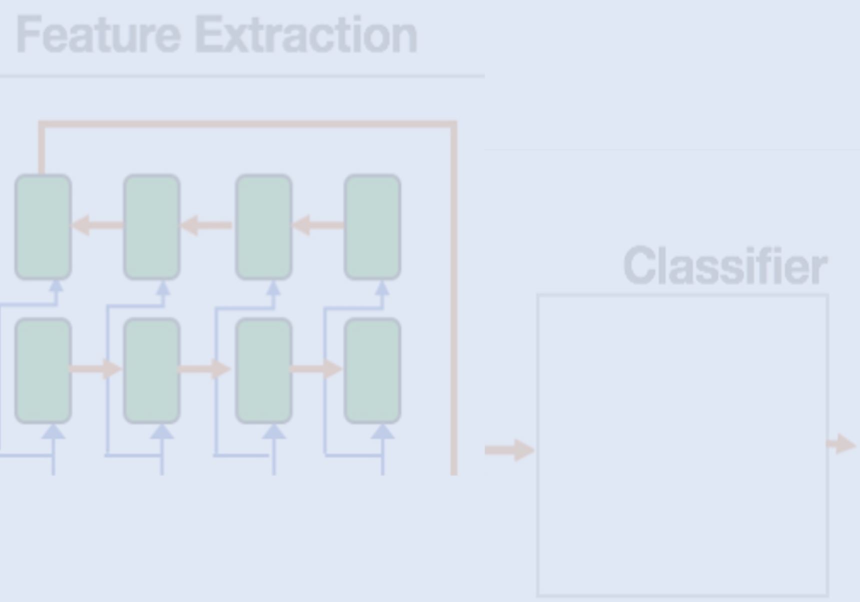
Conneaue et al 2018

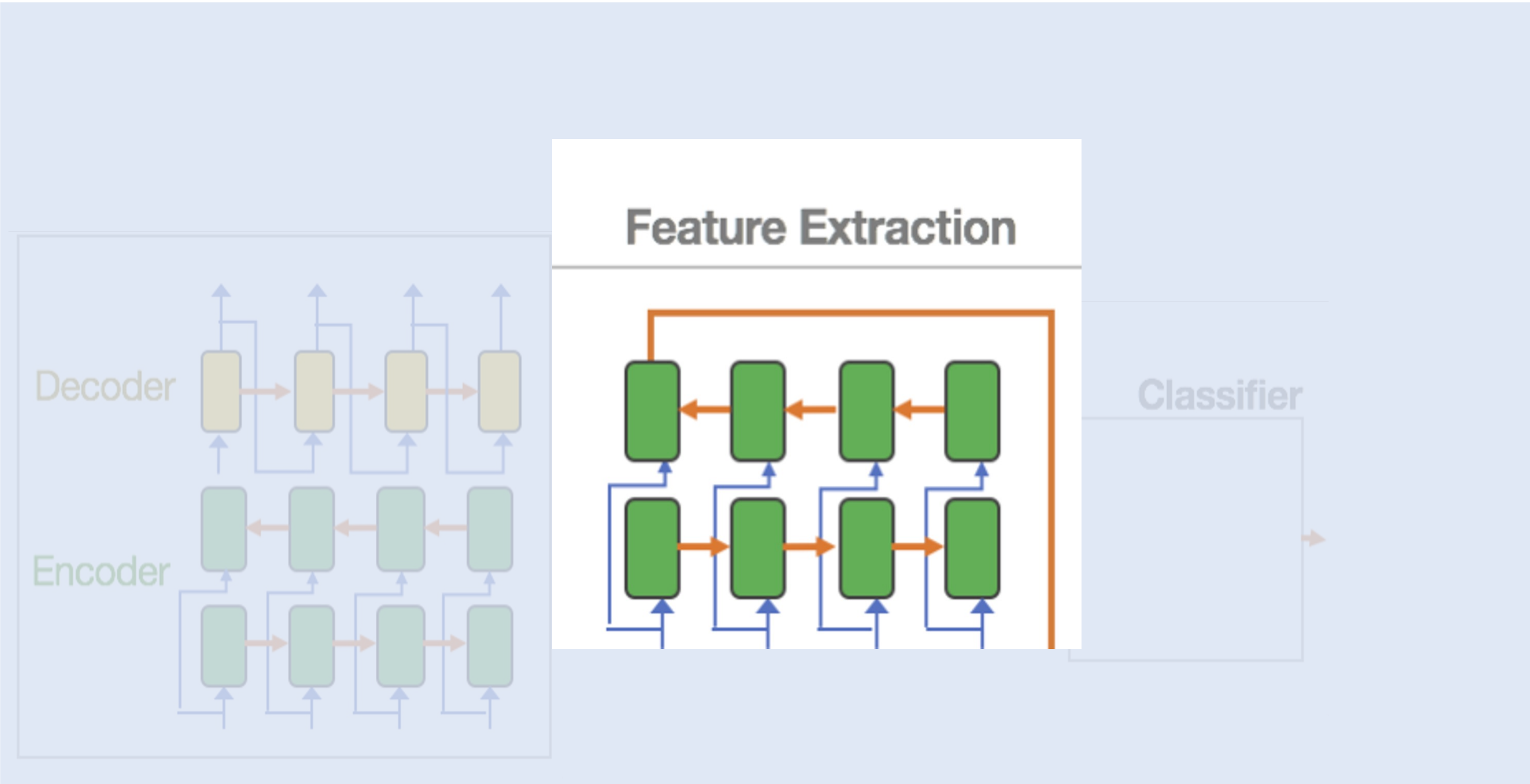



Decoder
Encoder
Feature Extraction
Classifier

1. Train a neural model

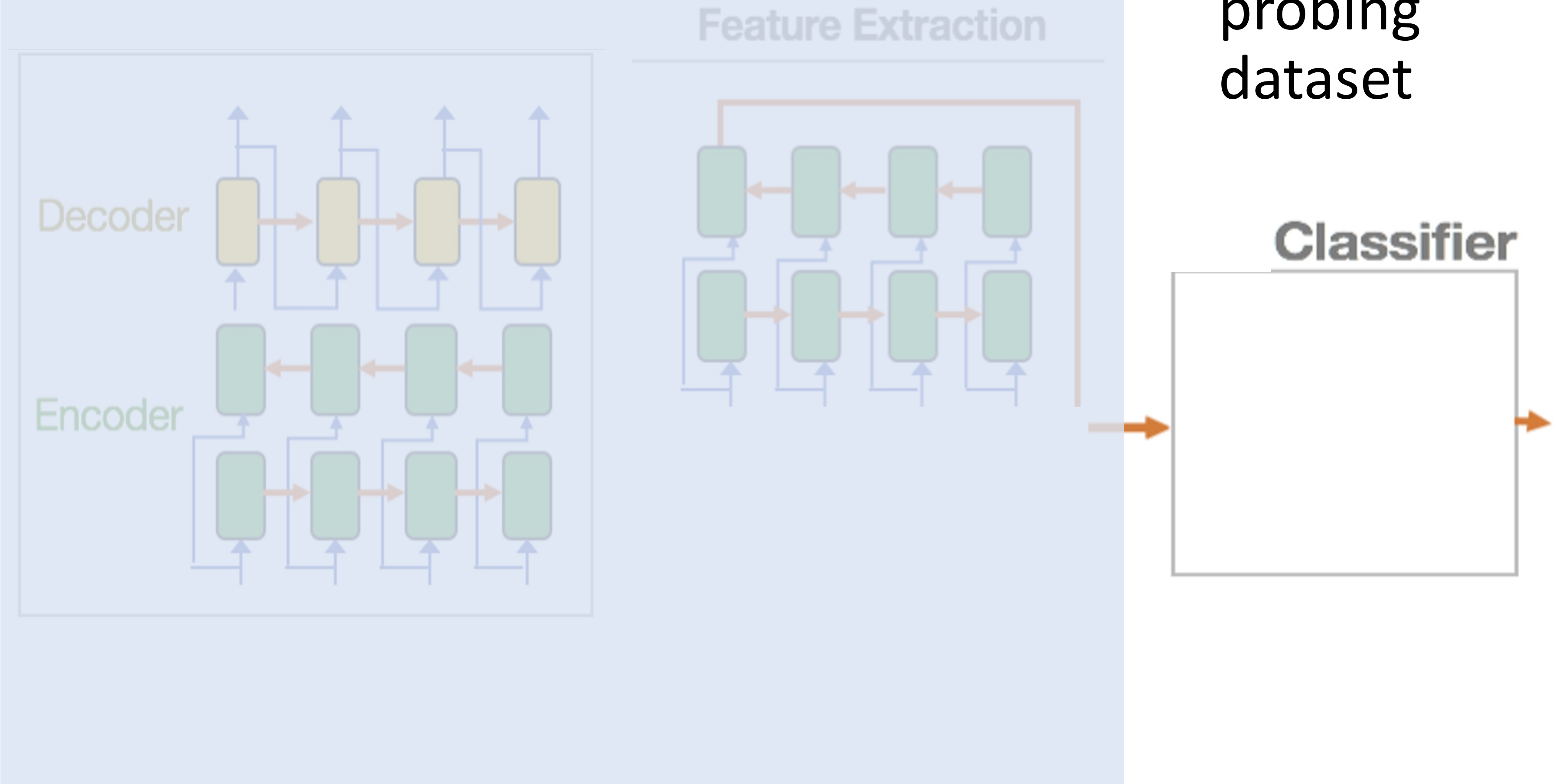




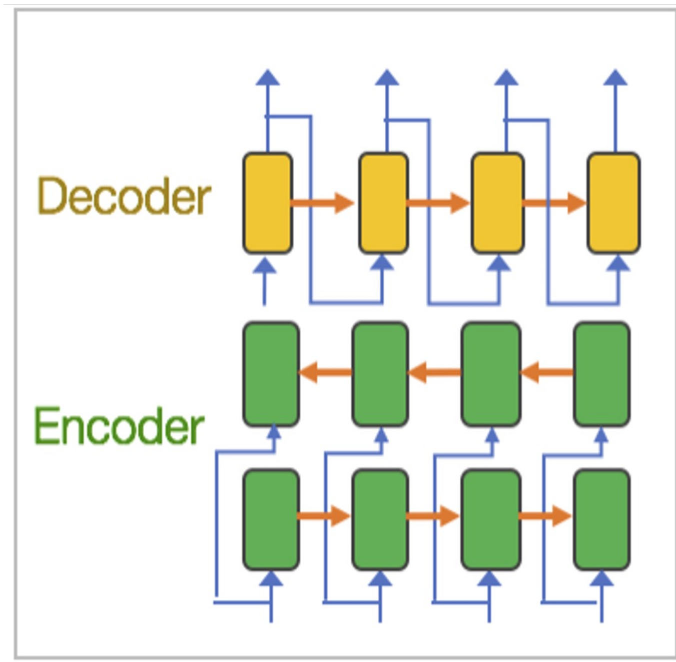


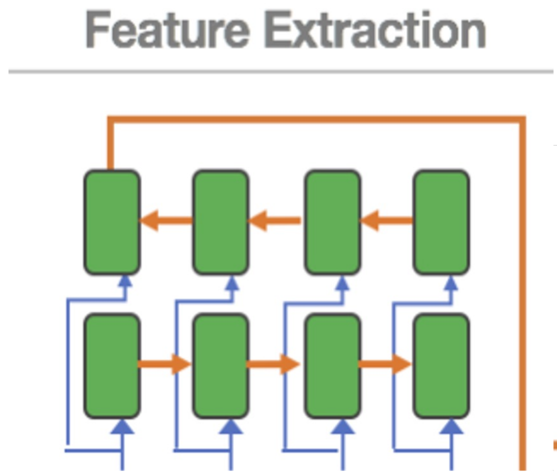
2. Extract sentence representations from trained neural encoder

3. Train and test classifier on probing dataset

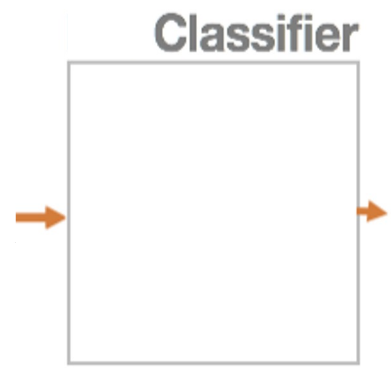


1. Train a neural model





3. Train and test classifier on probing dataset



2. Extract sentence representations from trained neural encoder

Probing Classifiers: Promises, Shortcomings, and Advances

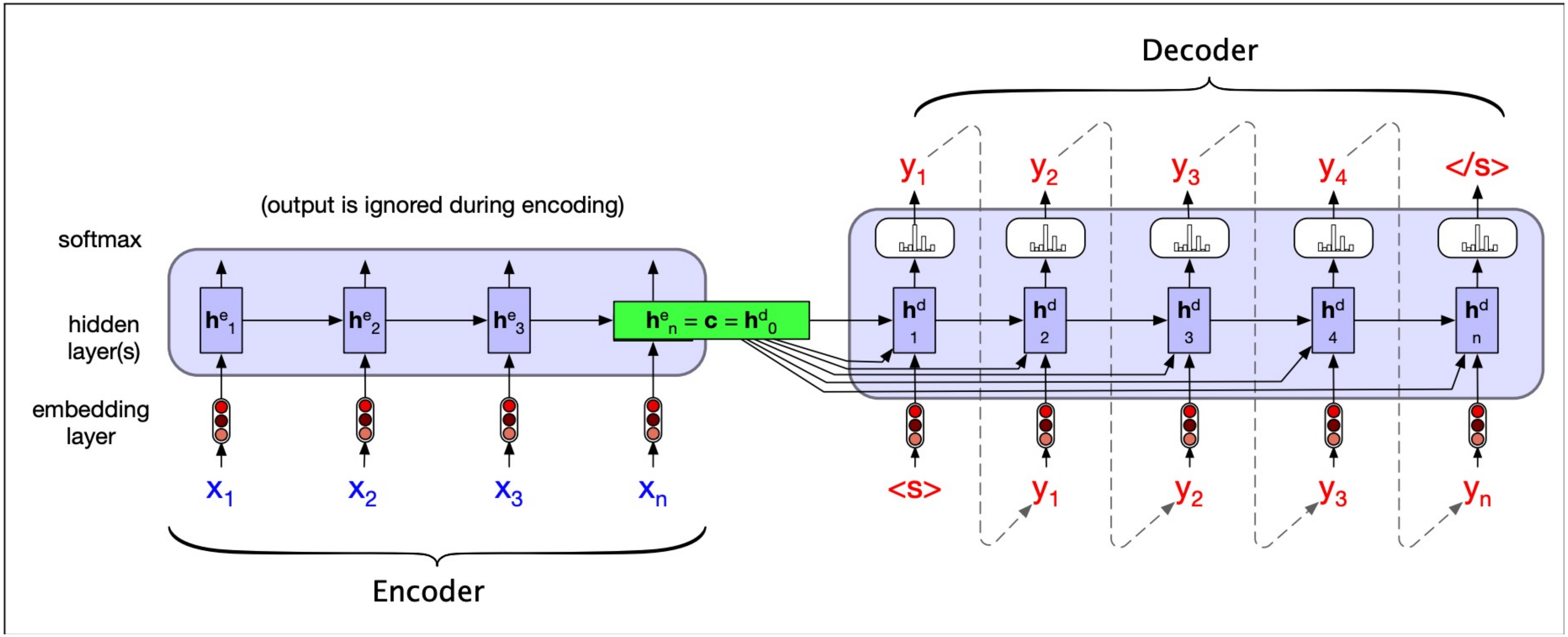
Yonatan Belinkov*
Technion – Israel Institute of Technology
belinkov@technion.ac.il

Probing classifiers have emerged as one of the prominent methodologies for interpreting and analyzing deep neural network models of natural language processing. The basic idea is simple—a classifier is trained to predict some linguistic property from a model's representations—and has been used to examine a wide variety of models and properties. However, recent studies have demonstrated various methodological limitations of this approach. This squib critically reviews the probing classifiers framework, highlighting their promises, shortcomings, and advances.

https://direct.mit.edu/coli/article-abstract/48/1/207/107571

Encoder-decoder

Decoder only uses information from last hidden cell!



Outline

Recap - RNNs

LSTM

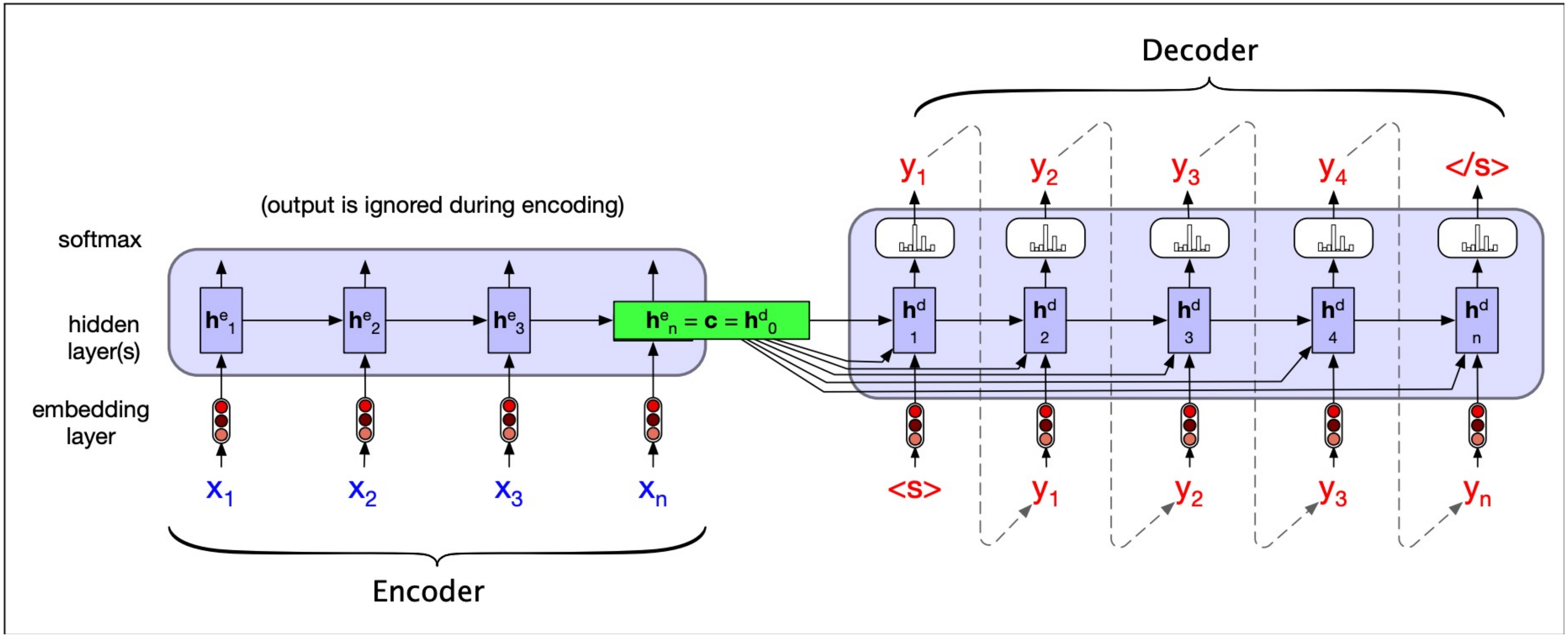
Sentence Representations/Probing

Attention

Transformer

Bottleneck

Last hidden cell is a bottleneck



Solution: Attention!

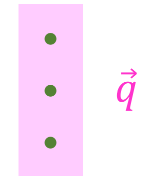
- solution to the bottleneck problem

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

Have we been cramming?

- RNNs etc. – decide early what to keep
  - Encode context into $\mathbb{R}^d$
  - Hope it supports later decoding needs
    - E.g., any reading comprehension question
- Attention – keep it all, decide later what to look at
  - At each decoding step, get to look back at all of the n encoded context objects, each in $\mathbb{R}^d$
  - Take a weighted average of them, where the weights depend on a query created at decoding time
  - This average "completes the encoding" into $\mathbb{R}^d$

Attention

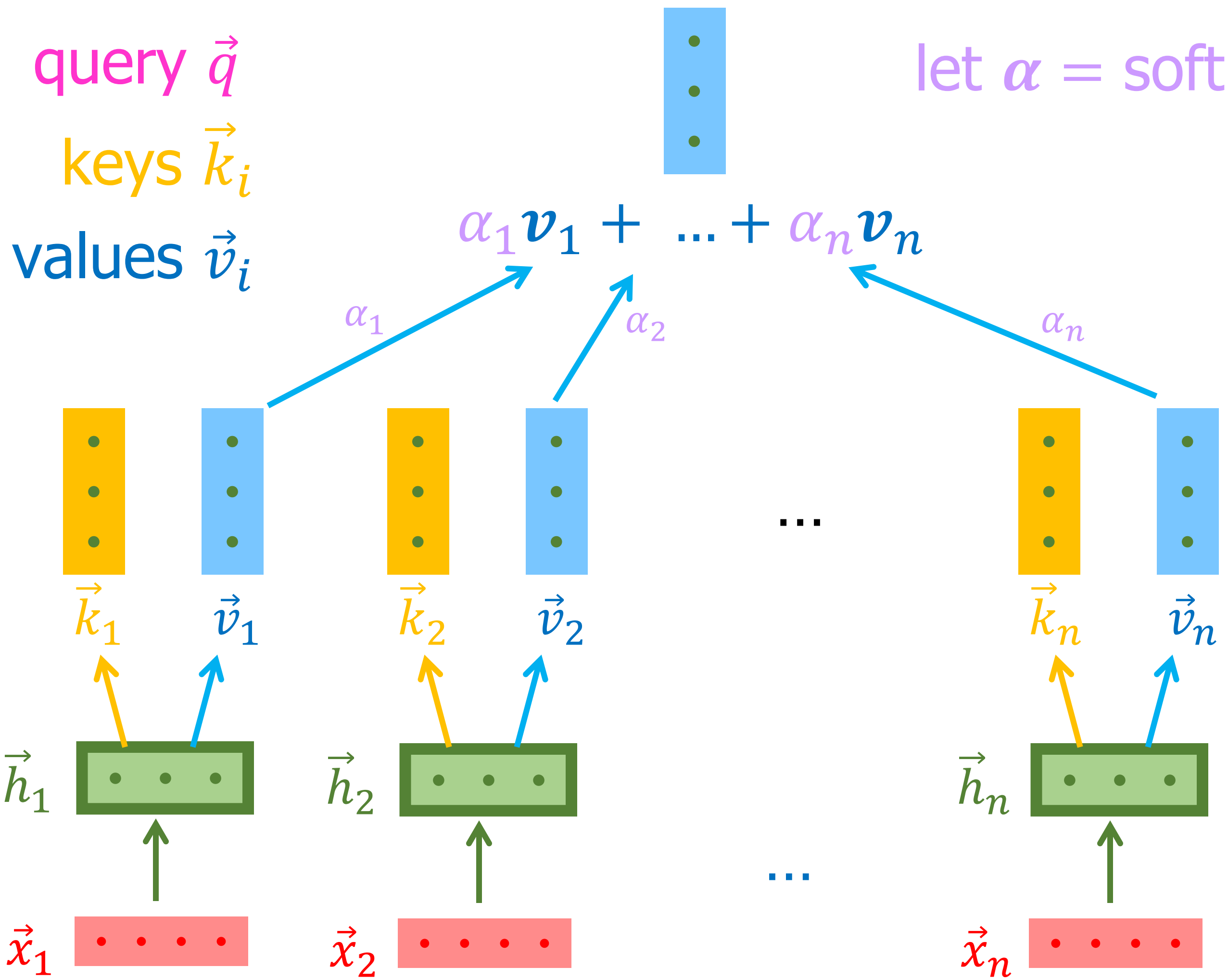


query $\vec{q}$

keys $\vec{k}_i$

values $\vec{v}_i$

let $\boldsymbol{\alpha} = \text{softmax}(\vec{q} \cdot \vec{k}_1, \ldots, \vec{q} \cdot \vec{k}_n)$



Decides what to look at! If $\boldsymbol{\alpha} = (0.01, 0.97, 0.01, \ldots)$ then we're mostly attending to $\vec{x}_2$ and just copying its value $\vec{v}_2$.

SGD may adjust $\boldsymbol{\alpha}$ to look more at $\vec{x}_3$ if moving the output toward $\vec{v}_3$ would help loss.

Uses of Attention

- Which input word to translate next?
- Which input word to copy next?
- Which database record to look at?
- Which part of the image to look at?
- Which document from the corpus to look at?
  - (fast data structures for Maximum Inner Product Search)

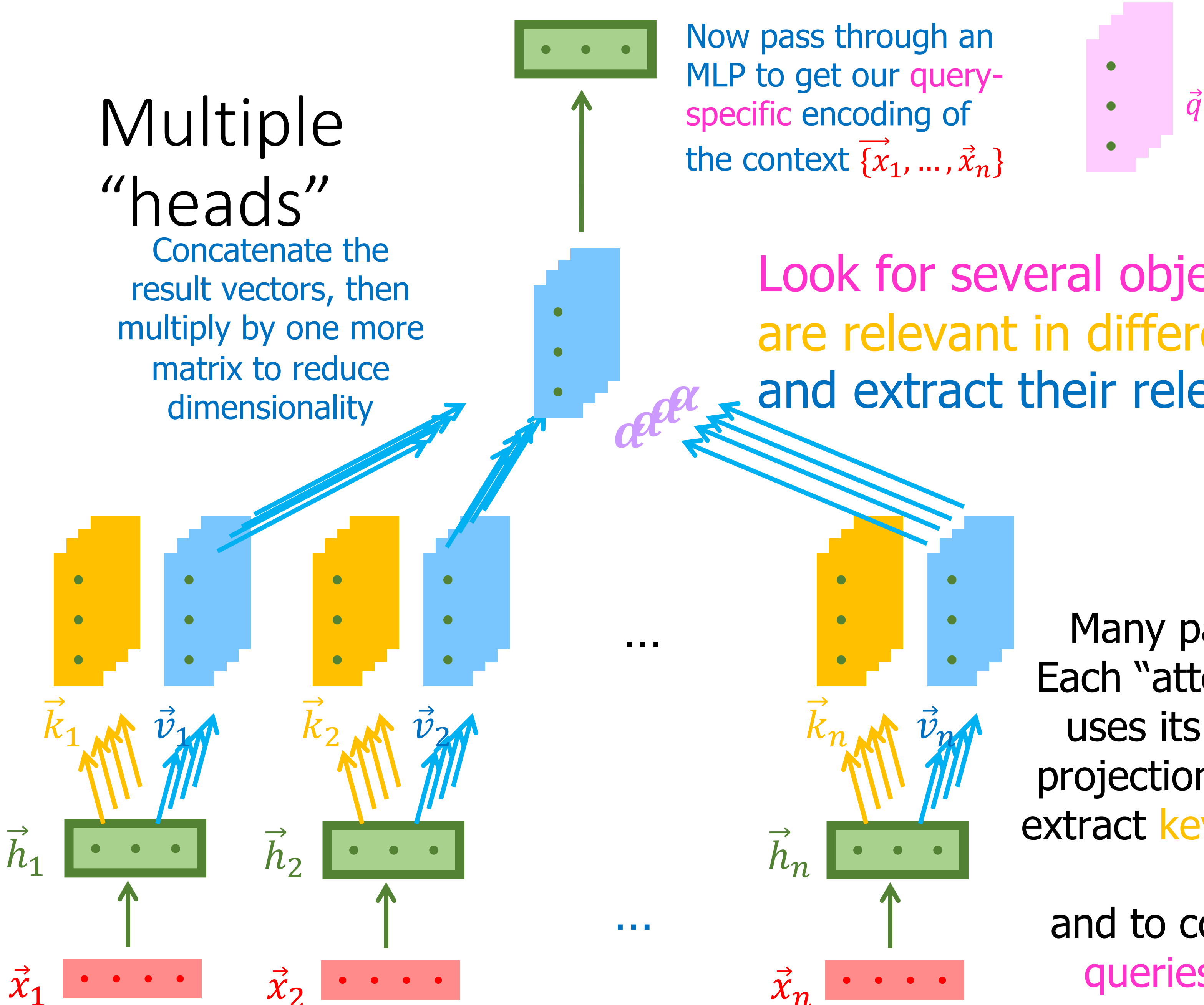
- Can encode an unordered bag of objects, of any size, in a way that's determined by a query
  - Use this contextual encoding within a larger model

Multiple “heads”

Now pass through an MLP to get our query-specific encoding of the context $\overrightarrow{\{\vec{x}_1, \ldots, \vec{x}_n\}}$

Concatenate the result vectors, then multiply by one more matrix to reduce dimensionality

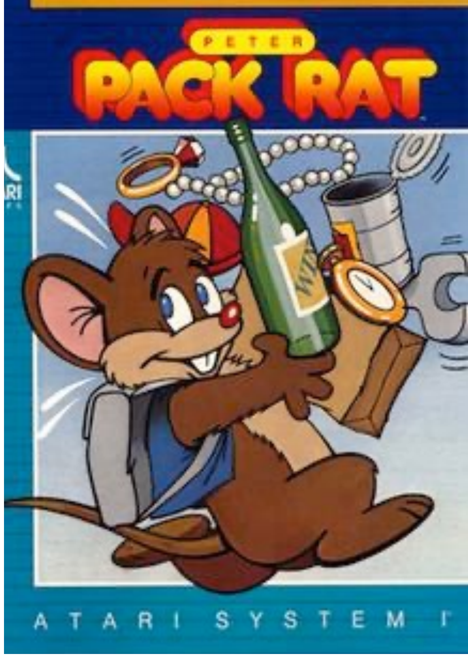
Look for several objects $\vec{x}_i$ that are relevant in different ways and extract their relevant info



Many parameters: Each “attention head” uses its own linear projection matrices to extract keys and values

and to construct the queries from the current prediction task

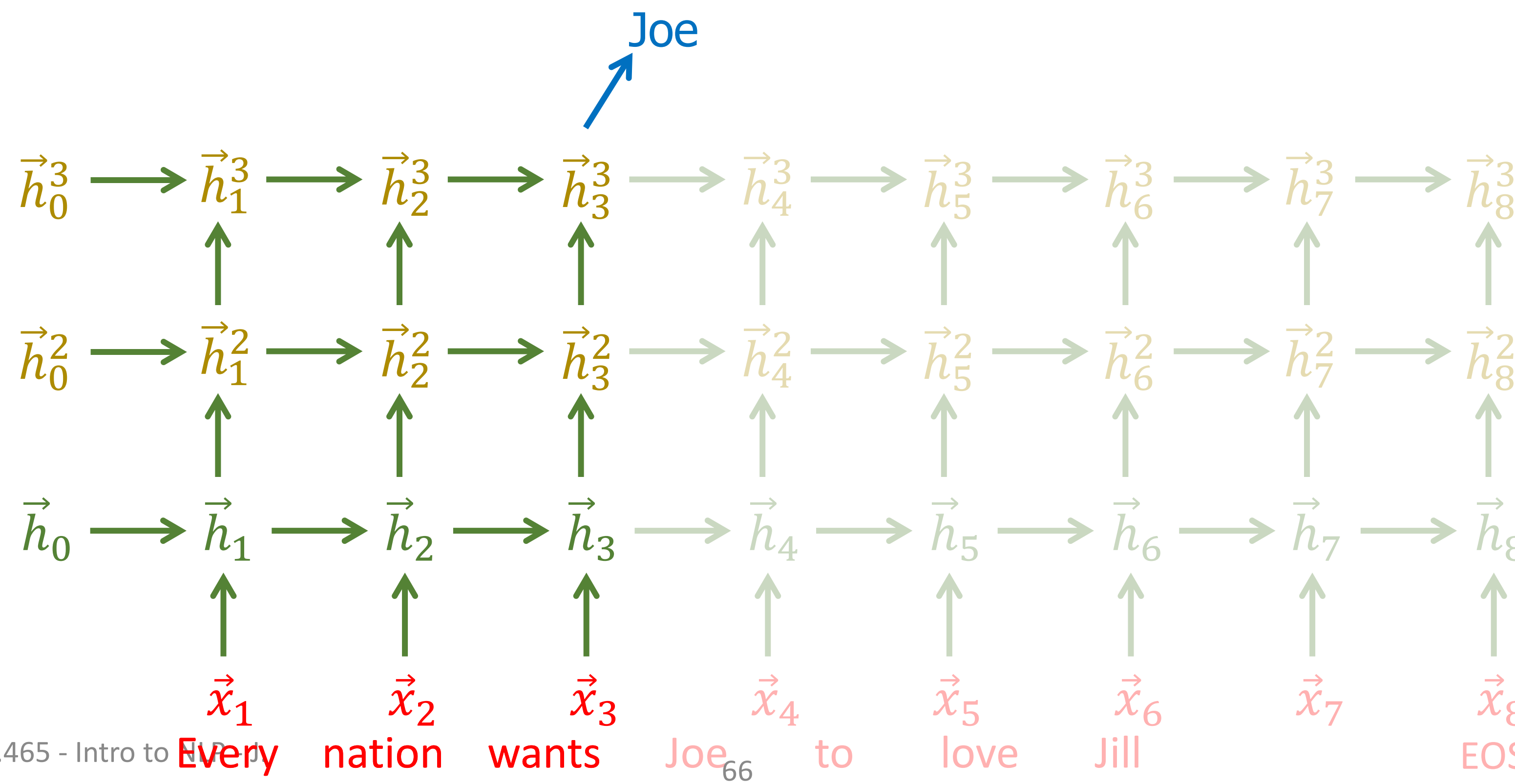
Uses of Attention



- Which input word to translate next?
- Which input word to copy next?
- Which constituent / named entity to look at?
- Which database record to look at?
- Which part of the image to look at next?
- Which document from the corpus to look at?

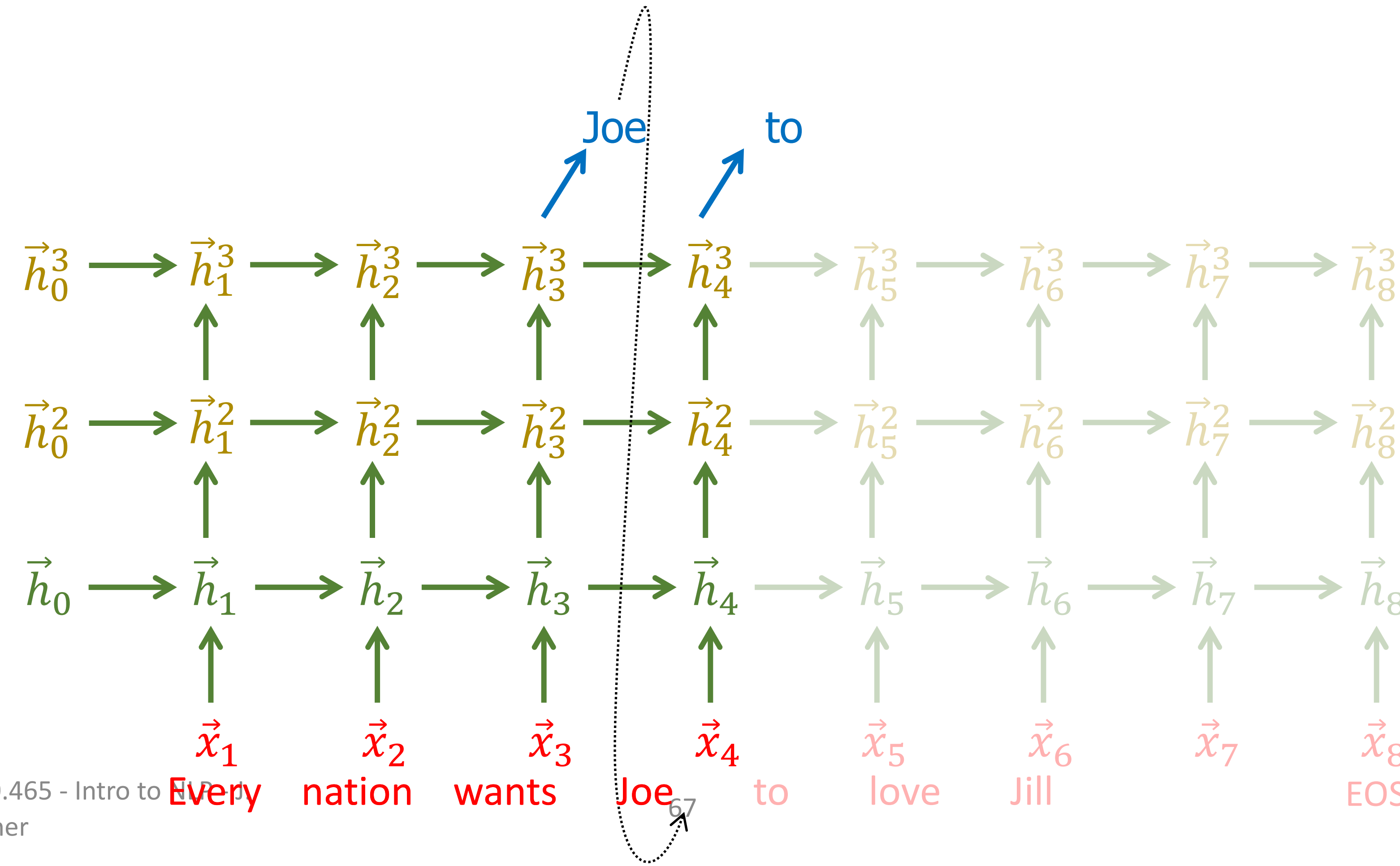
- Can encode an unordered bag of objects, of any size, in a way that's determined by a query
  - Use this contextual encoding within a larger model
  - Transformer architecture: "Attention is all you need"

RNN LM

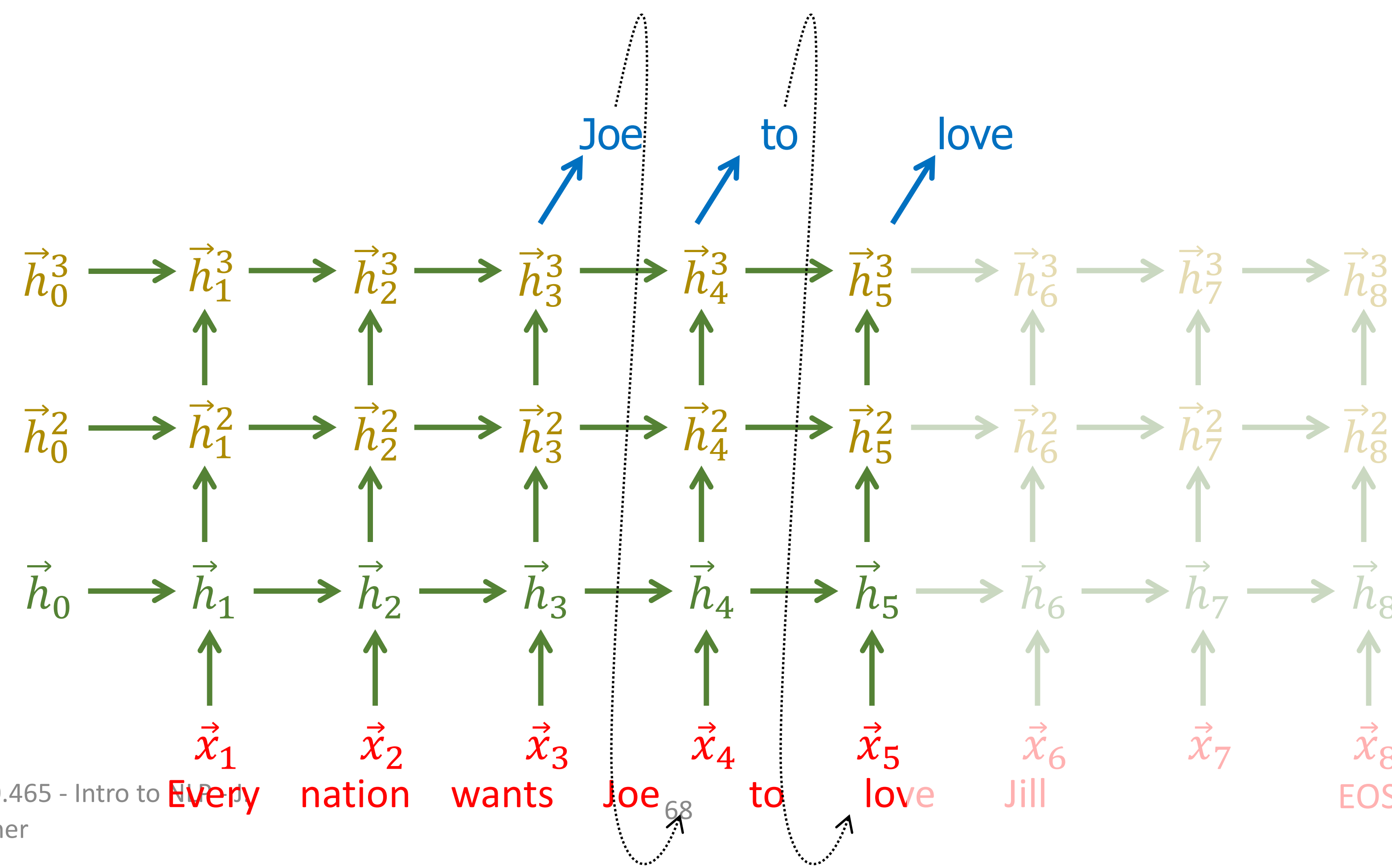


Joe
$\vec{h}_0^3$ $\vec{h}_1^3$ $\vec{h}_2^3$ $\vec{h}_3^3$ $\vec{h}_4^3$ $\vec{h}_5^3$ $\vec{h}_6^3$ $\vec{h}_7^3$ $\vec{h}_8^3$
$\vec{h}_0^2$ $\vec{h}_1^2$ $\vec{h}_2^2$ $\vec{h}_3^2$ $\vec{h}_4^2$ $\vec{h}_5^2$ $\vec{h}_6^2$ $\vec{h}_7^2$ $\vec{h}_8^2$
$\vec{h}_0$ $\vec{h}_1$ $\vec{h}_2$ $\vec{h}_3$ $\vec{h}_4$ $\vec{h}_5$ $\vec{h}_6$ $\vec{h}_7$ $\vec{h}_8$
$\vec{x}_1$ $\vec{x}_2$ $\vec{x}_3$ $\vec{x}_4$ $\vec{x}_5$ $\vec{x}_6$ $\vec{x}_7$ $\vec{x}_8$
Every nation wants Joe to love Jill EOS

RNN LM


Joe
to
$\vec{h}_0^3$ $\vec{h}_1^3$ $\vec{h}_2^3$ $\vec{h}_3^3$ $\vec{h}_4^3$ $\vec{h}_5^3$ $\vec{h}_6^3$ $\vec{h}_7^3$ $\vec{h}_8^3$
$\vec{h}_0^2$ $\vec{h}_1^2$ $\vec{h}_2^2$ $\vec{h}_3^2$ $\vec{h}_4^2$ $\vec{h}_5^2$ $\vec{h}_6^2$ $\vec{h}_7^2$ $\vec{h}_8^2$
$\vec{h}_0$ $\vec{h}_1$ $\vec{h}_2$ $\vec{h}_3$ $\vec{h}_4$ $\vec{h}_5$ $\vec{h}_6$ $\vec{h}_7$ $\vec{h}_8$
$\vec{x}_1$ $\vec{x}_2$ $\vec{x}_3$ $\vec{x}_4$ $\vec{x}_5$ $\vec{x}_6$ $\vec{x}_7$ $\vec{x}_8$
Every nation wants Joe to love Jill EOS

RNN LM



Joe
to
love
$\vec{h}_0^3$ $\vec{h}_1^3$ $\vec{h}_2^3$ $\vec{h}_3^3$ $\vec{h}_4^3$ $\vec{h}_5^3$ $\vec{h}_6^3$ $\vec{h}_7^3$ $\vec{h}_8^3$
$\vec{h}_0^2$ $\vec{h}_1^2$ $\vec{h}_2^2$ $\vec{h}_3^2$ $\vec{h}_4^2$ $\vec{h}_5^2$ $\vec{h}_6^2$ $\vec{h}_7^2$ $\vec{h}_8^2$
$\vec{h}_0$ $\vec{h}_1$ $\vec{h}_2$ $\vec{h}_3$ $\vec{h}_4$ $\vec{h}_5$ $\vec{h}_6$ $\vec{h}_7$ $\vec{h}_8$
$\vec{x}_1$ $\vec{x}_2$ $\vec{x}_3$ $\vec{x}_4$ $\vec{x}_5$ $\vec{x}_6$ $\vec{x}_7$ $\vec{x}_8$
Every nation wants Joe to love Jill EOS

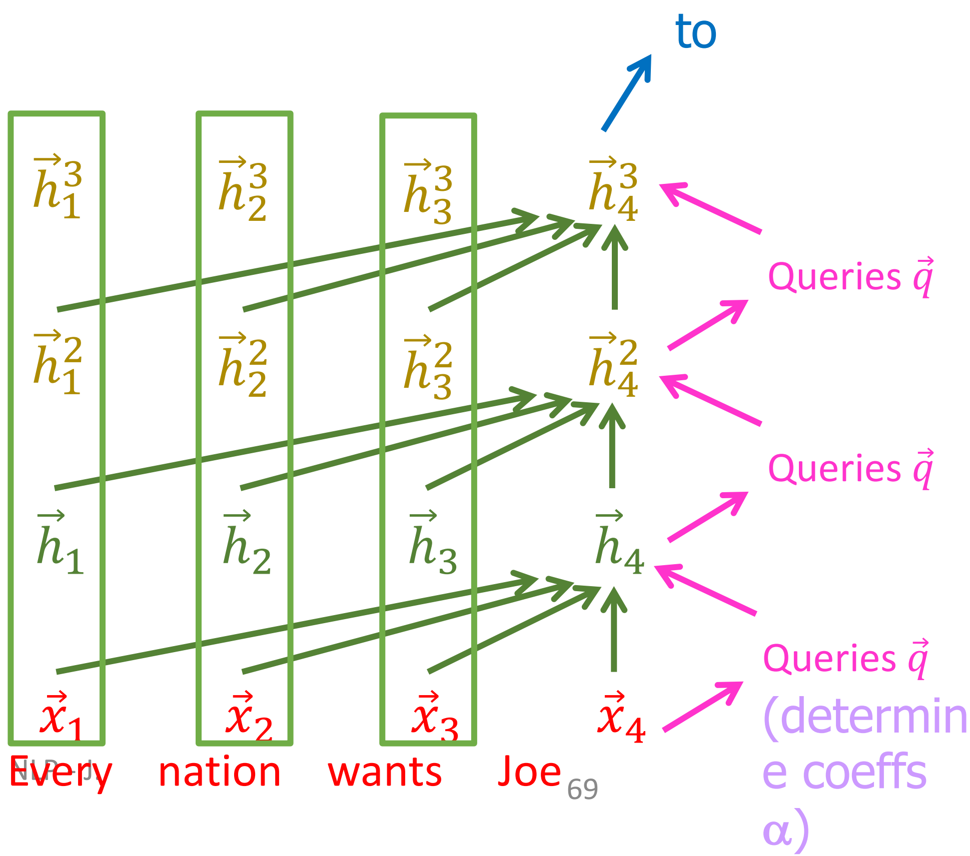
For some other missing details: Check out "[The Illustrated Transformer]" by Jay Alammar

Transformer LM (e.g., GPT-3)

Joe is repeatedly transformed to consider more and more context.

Joe's current representation tells the heads how to query *all* the words in the current layer. Each head's query returns an averaged value. Those answers are concatenated and go thru MLP to get Joe's transformed representation. Repeat!

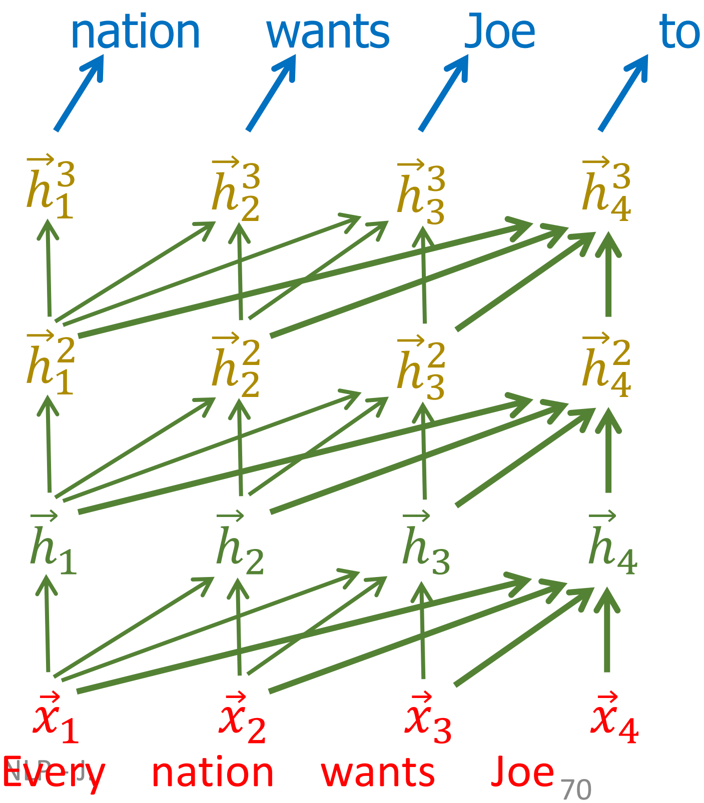
Attention looks at these representations of the previous words, which were obtained in same way while making previous predictions. Their keys/vals are unchanged.



(Actually, the transformed representation is the old representation *plus* the MLP output. Like a residual RNN.)

Training can be parallelized

At training time, the whole sentence is known.
Layer-L representations can be computed in parallel, with each word attending to the layer-(L-1) representations of itself and previous words



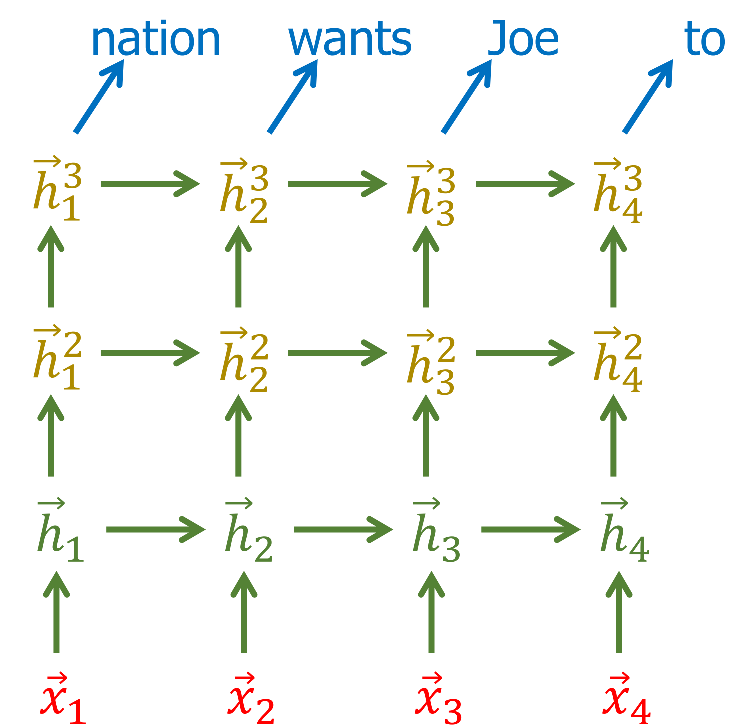
(oops, to predict the very first word, we needed $\vec{x}_0 =< s >$! It's missing from our diagrams.)

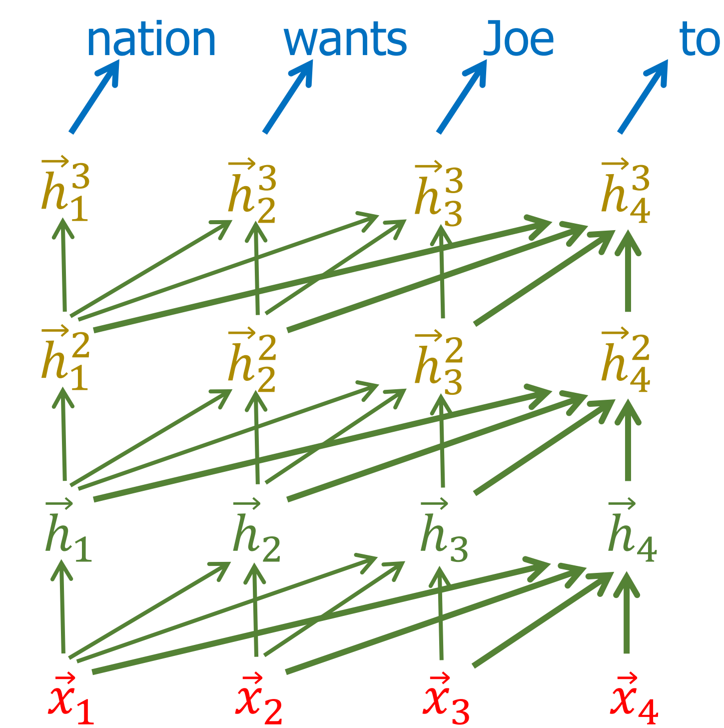
Training, on GPU, per layer ↓

RNN vs. Transformer

Computations: ☺ O(n) ☹ $O(n^2)$

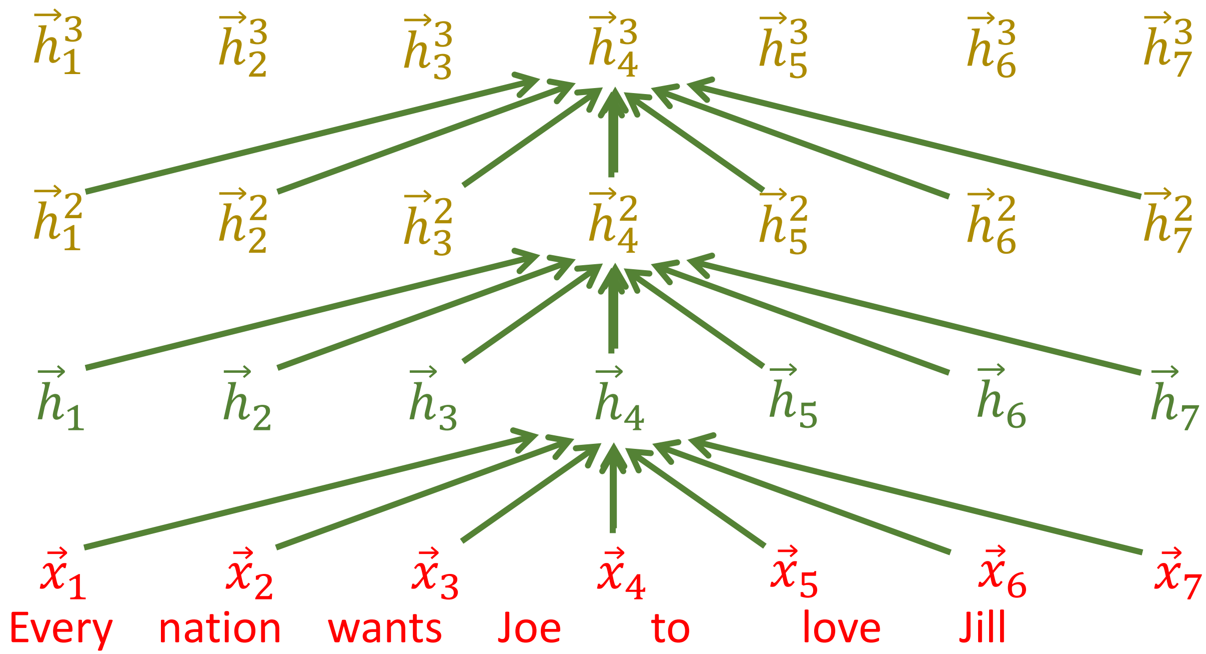
\# serial steps: ☹ O(n) due to → ☺ O(1): all ↗↑ in parallel + O(log n) to sum n inputs





Transformer *encoder*

- A decoder (LM) mustn't peek ahead at words it's trying to predict
- But an encoder is given all the words at the start
- So a word in one layer can look at *all* words in previous layers
- Now we get highly contextual top-level encodings of all input words

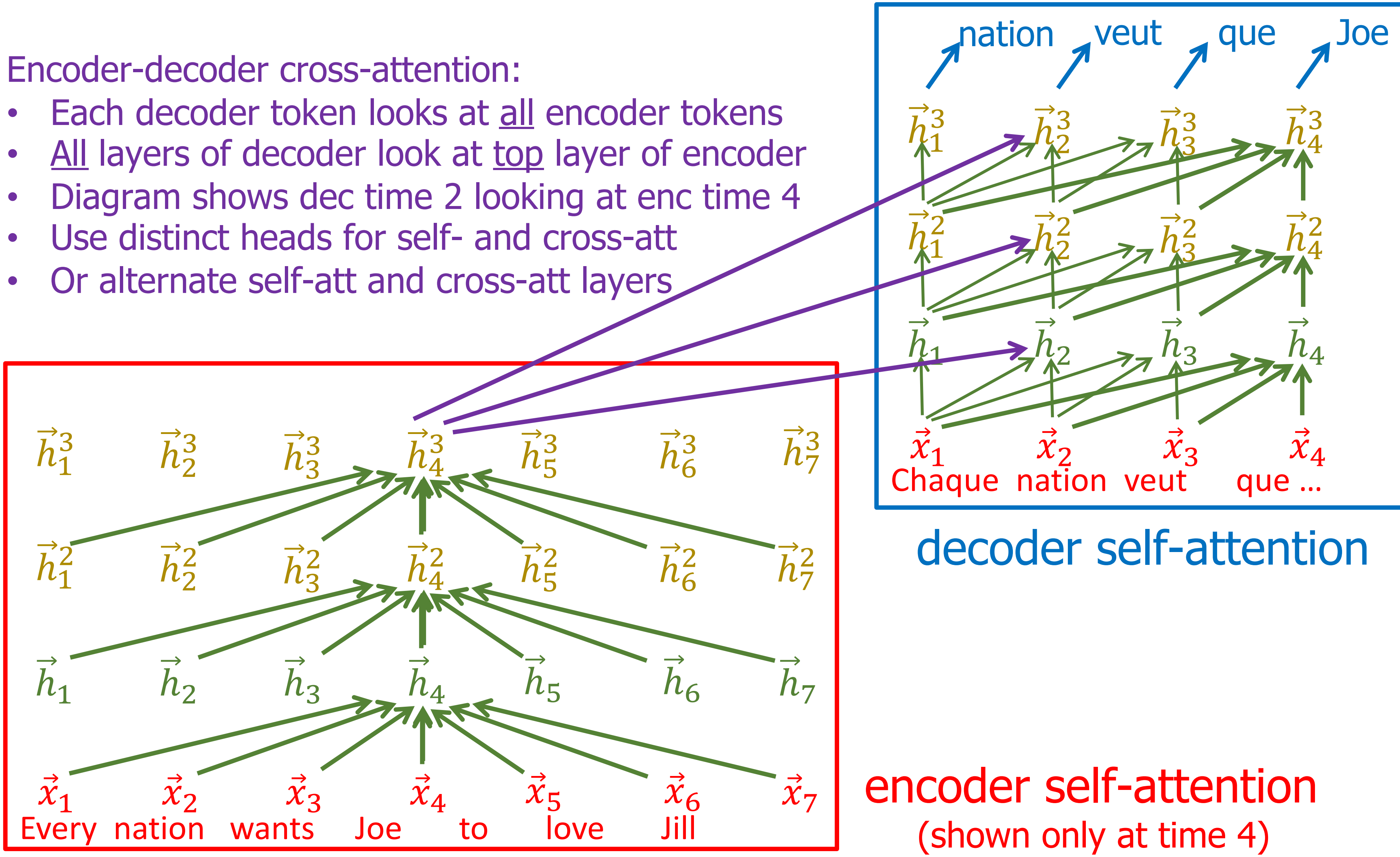


Transformer seq2seq

For some missing details: Check out "[The Illustrated Transformer]" by Jay Alammar

Encoder-decoder cross-attention:

- Each decoder token looks at all encoder tokens
- All layers of decoder look at top layer of encoder
- Diagram shows dec time 2 looking at enc time 4
- Use distinct heads for self- and cross-att
- Or alternate self-att and cross-att layers



Positional embeddings

- Attention doesn't see the order of the words.
- One standard solution:
  - Replace input $\vec{x}_4$ with $\vec{x}_4 + \vec{p}_4$
  - Vector $\vec{p}_4$ encodes "position 4"
  - Vector $\vec{x}_4$ encodes the word at that position
  - Attention sees both: e.g., the $\vec{q} \cdot \vec{k}_4$ logit (attention on input 4) will be a sum of logits from $\vec{x}_4$ and $\vec{p}_4$
- There's a standard sinusoidal scheme for constructing the vectors $\vec{p}_i$ so we don't have to learn them – they're fixed