

Deep Learning for Applications - II

심스리얼리티 임직원을 위한 딥러닝 교육 과정

Dr. Kyong-Ha Lee (kyongha@kisti.re.kr)





Contents

Class2 : 뉴럴 네트워크에 대한 기초를 토대로 좀더 복잡한 모델
(RNN, Attention, Transformer, LLM)들을 이해합니다.

Contents 1

Recurrent Neural Network

Contents 2

Attention Mechanism

Contents 3

Transformers

Contents 4

Language Models

Contents 5

Large Language Models and applications



A Few Quotes about Machine Learning

- Motivation :
 - Why do we need Sequential Modeling?

Examples of Sequence data

Speech Recognition

Machine Translation

Language Modeling

Named Entity Recognition

Sentiment Classification

Video Activity Analysis



Input Data



Hello, I am Pankaj.

Recurrent neural based model

Pankaj lives in Munich

There is nothing to like in this movie.



Output

This is RNN

Haloo, ich bin Pankaj.
हैलो, मैं पंकज हूँ।

network

language

Pankaj lives in Munich
person location



Punching



Motivation: Need for Sequential Modeling

- **Inputs, outputs can be in different lengths in different examples**

Example:

Sentence1: Pankaj lives in Munich

Sentence2: Pankaj Gupta lives in Munich DE





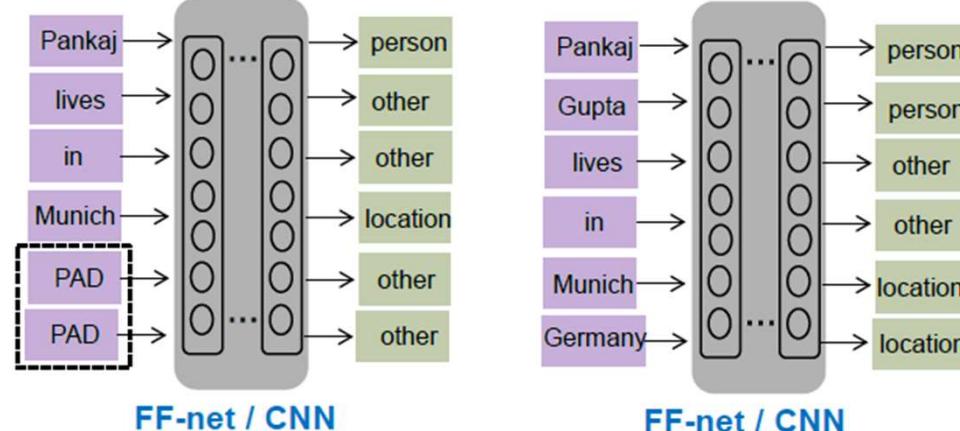
Motivation: Need for Sequential Modeling

- Inputs, outputs can be in different lengths in different examples

Example:

Sentence1: Pankaj lives in Munich

Sentence2: Pankaj Gupta lives in Munich DE



Additional word
'PAD' i.e., padding



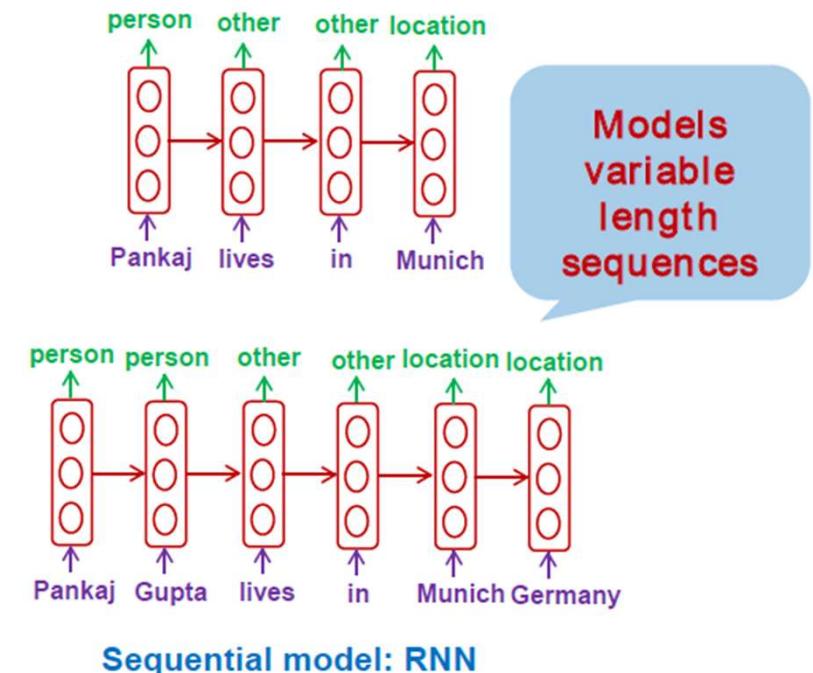
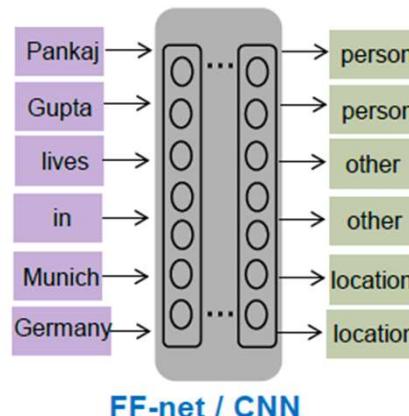
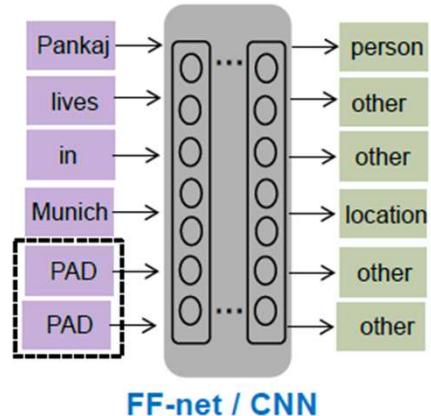
Motivation: Need for Sequential Modeling

- Inputs, outputs can be in different lengths in different examples

Example:

Sentence1: Pankaj lives in Munich

Sentence2: Pankaj Gupta lives in Munich DE





Motivation: Need for Sequential Modeling

- **Shared features learned across different positions or time steps**

Example:

Sentence1: *Market falls into bear territory* → *Trading/Marketing*

Sentence2: *Bear falls into market territory* → *UNK*

Same uni-gram statistics



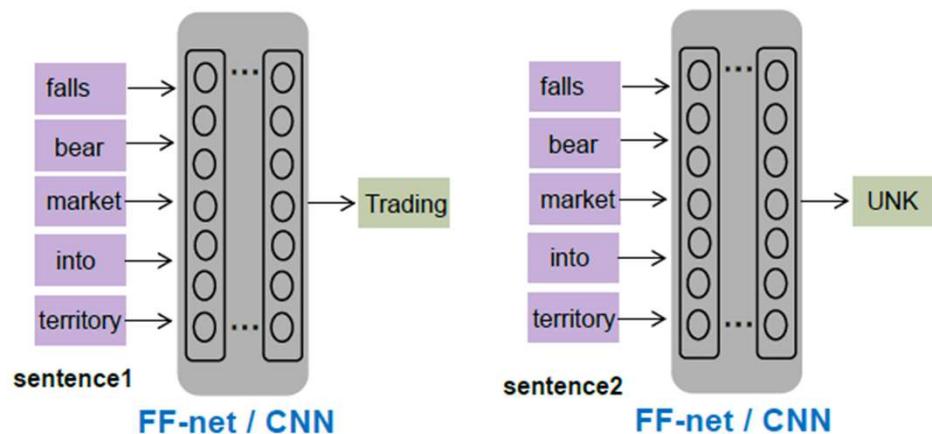
Motivation: Need for Sequential Modeling

- Shared features learned across different positions or time steps

Example:

Sentence1: *Market falls into bear territory* → *Trading/Marketing*

Sentence2: *Bear falls into market territory* → *UNK*



No sequential or temporal modeling, i.e., order-less

Treats the two sentences the same



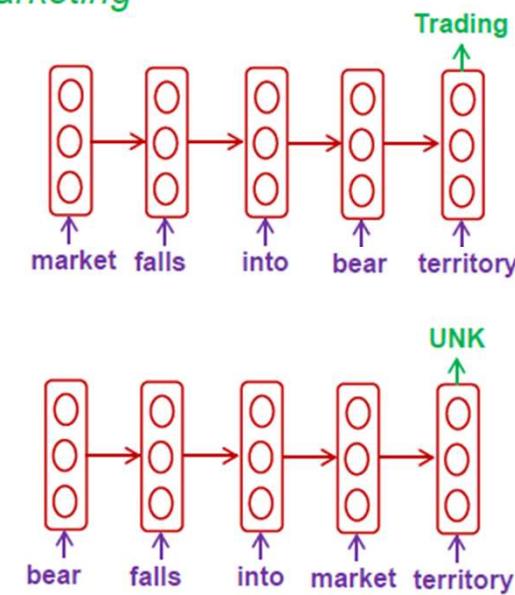
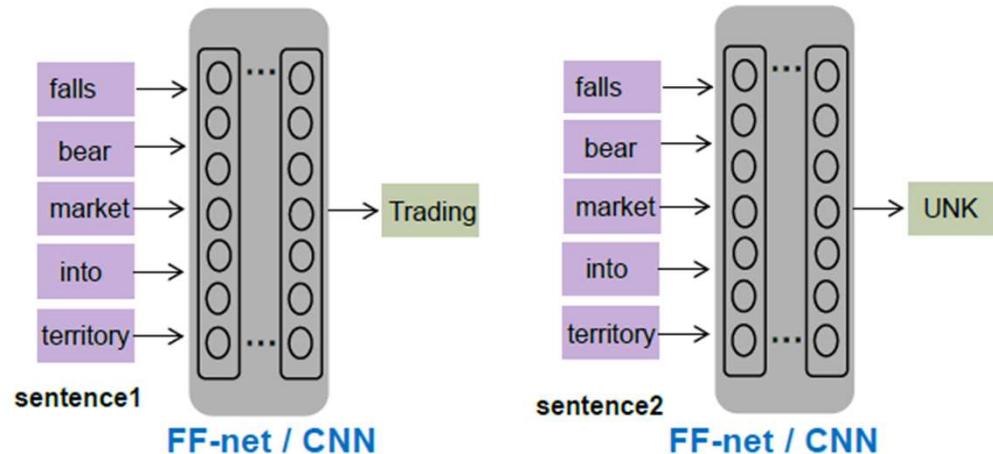
Motivation: Need for Sequential Modeling

- Shared features learned across different positions or time steps

Example:

Sentence1: *Market falls into bear territory* → *Trading/Marketing*

Sentence2: *Bear falls into market territory* → *UNK*



Sequential model: RNN

Language concepts,
Word ordering,
Syntactic & semantic information



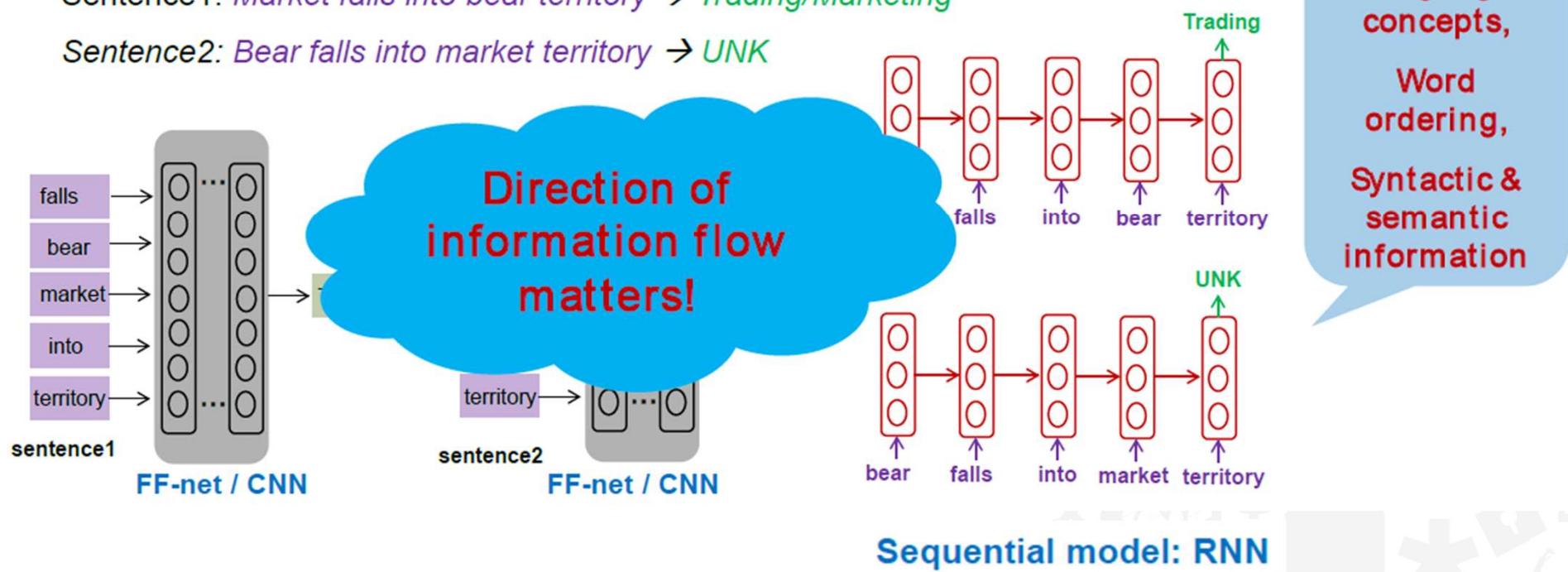
Motivation: Need for Sequential Modeling

- Shared features learned across different positions or time steps

Example:

Sentence1: Market falls into bear territory → Trading/Marketing

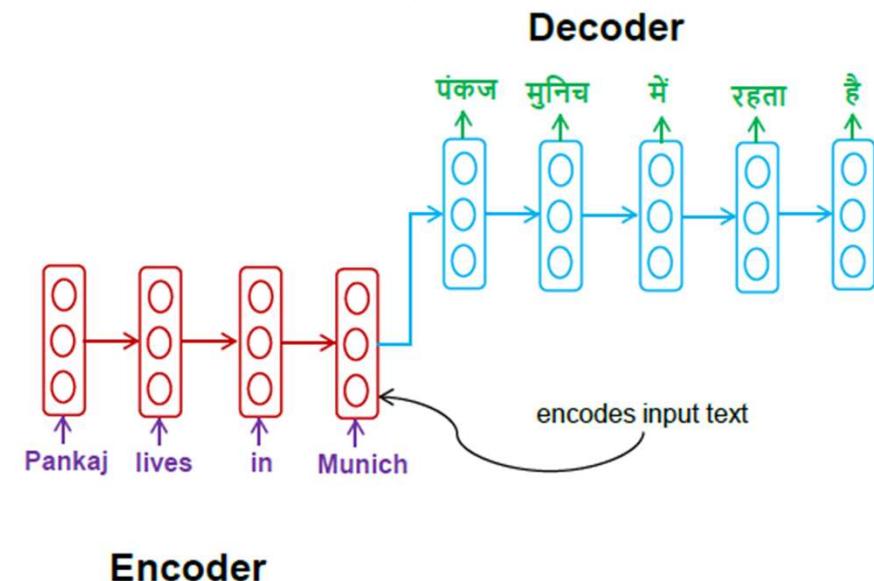
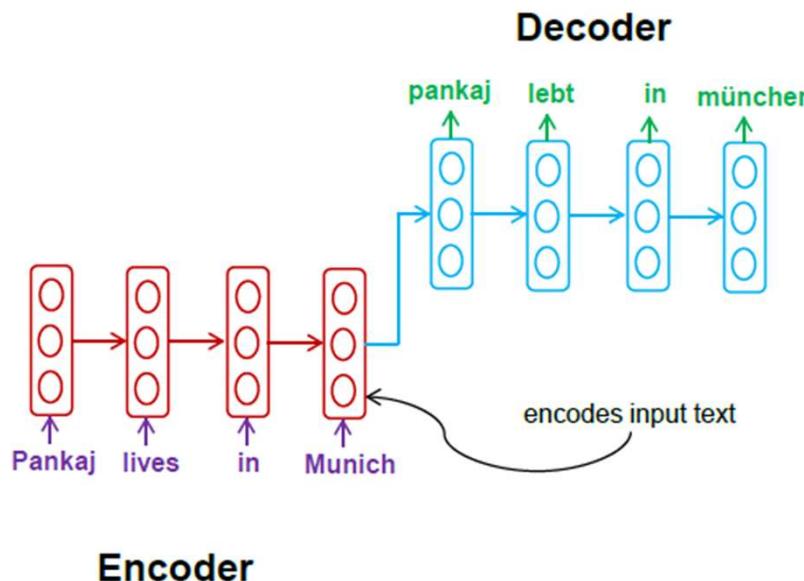
Sentence2: Bear falls into market territory → UNK





Motivation: Need for Sequential Modeling

- **Machine Translation:** different input & output sizes, incurring sequential patterns





Motivation: Need for Sequential Modeling

- **Convolutional vs. Recurrent Neural Networks**

RNN

- perform well when the input data is interdependent in a sequential pattern
- correlation between previous input to the next input
- introduce bias based on your previous output

CNN/FF-Nets

- all the outputs are self dependent
- *Feed-forward nets don't remember historic input data at test time unlike recurrent networks.*

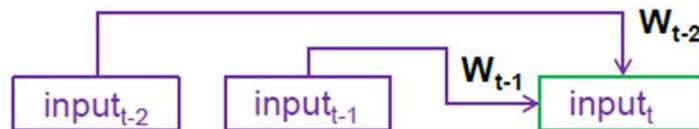


Motivation: Need for Sequential Modeling

Memory-less Models

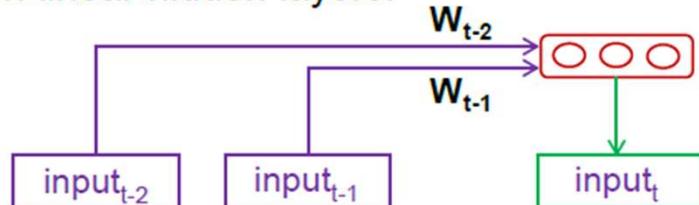
Autoregressive models:

Predict the next input in a sequence from a fixed number of previous inputs using “delay taps”.



Feed-forward neural networks:

Generalize autoregressive models by using non-linear hidden layers.



Memory Networks

-possess a dynamic hidden state that can store long term information, e.g., RNNs.

Recurrent Neural Networks:

RNNs are very powerful, because they combine the following properties-

Distributed hidden state: can efficiently store a lot of information about the past.

Non-linear dynamics: can update their hidden state in complicated ways

Temporal and accumulative: can build semantics, e.g., word-by-word in sequence over time



Term dependencies

Short Term Dependencies

→ need recent information to perform the present task.

For example in a language model, predict the next word based on the previous ones.

"the clouds are in the ?" → 'sky'

"the clouds are in the sky"

→ Easier to predict 'sky' given the context, i.e., *short term dependency*

Long Term Dependencies

→ Consider longer word sequence "I grew up in France..... I speak fluent **French.**"

→ Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back.



Foundation of Recurrent Neural Networks

Goal

- model long term dependencies
- connect previous information to the present task
- model sequence of events with loops, allowing information to persist



punching



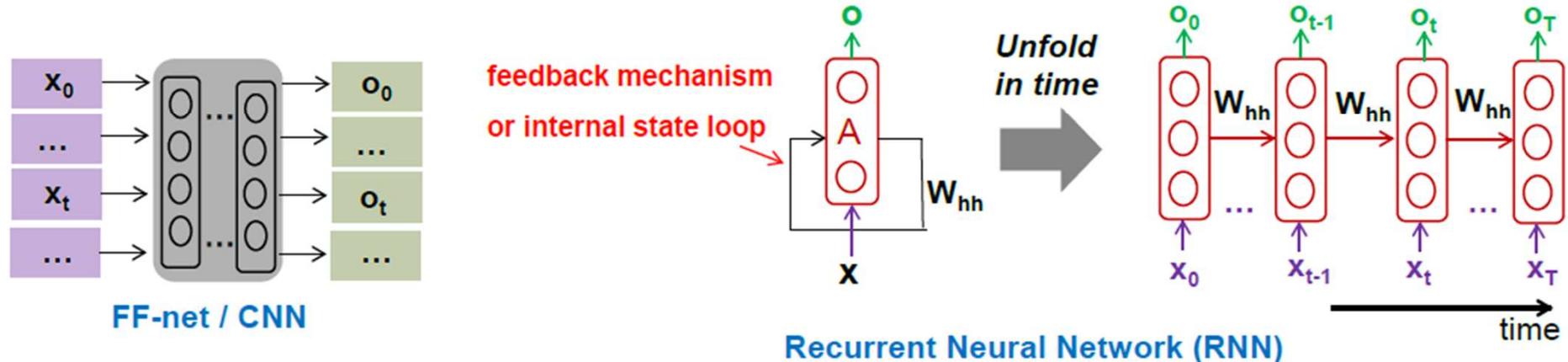
Foundation of Recurrent Neural Networks

Goal

- model long term dependencies
- connect previous information to the present task
- model sequence of events with loops, allowing information to persist

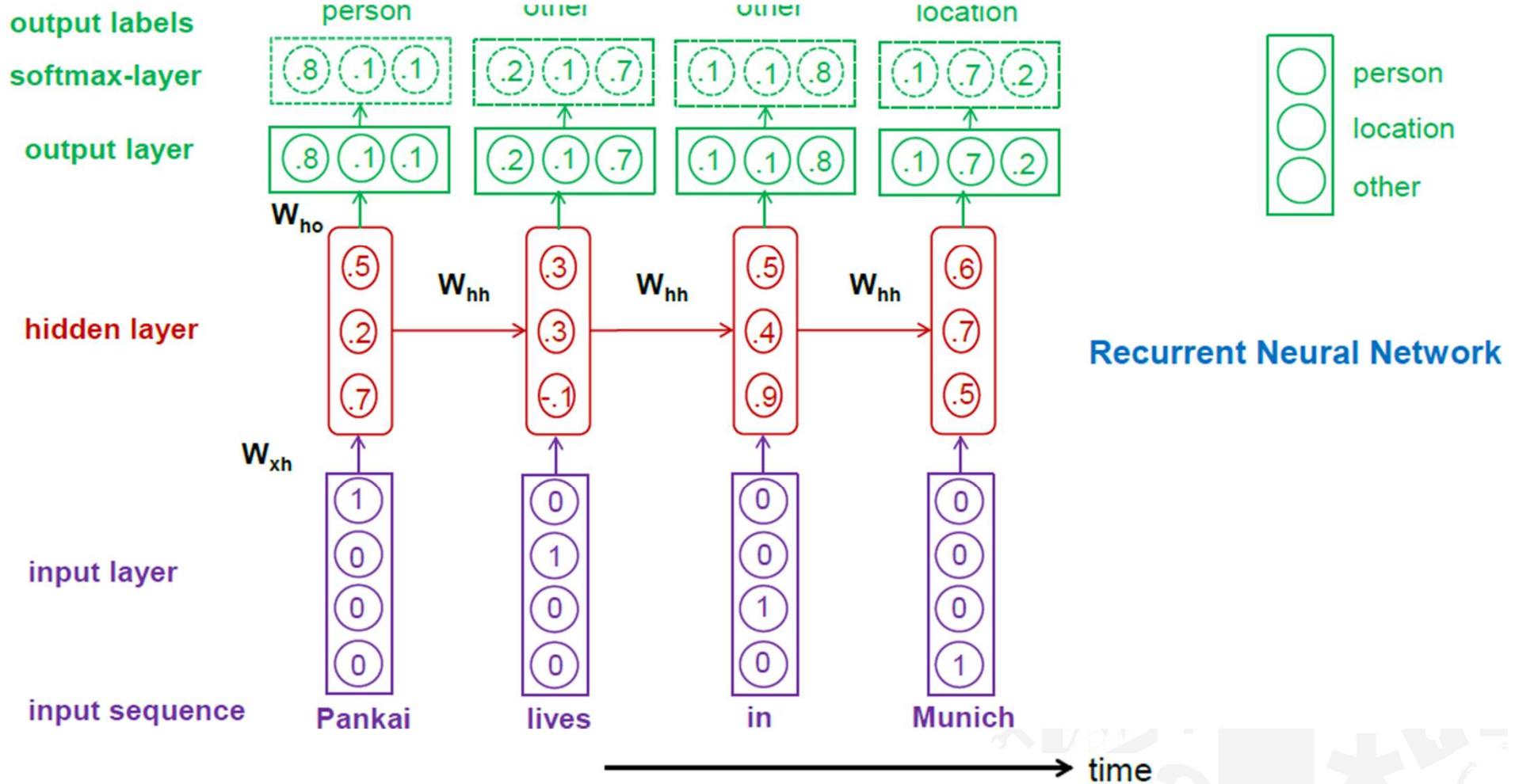
Feed Forward NNets can **not** take time dependencies into account.

Sequential data needs a **Feedback Mechanism**.





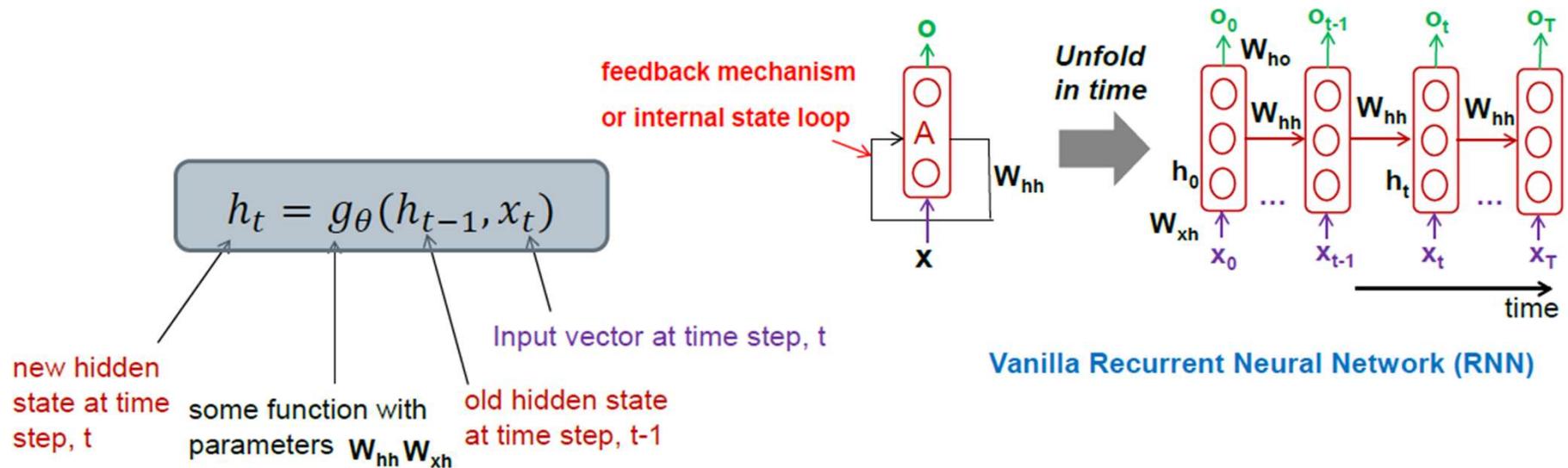
Foundation of Recurrent Neural Networks





(Vanilla) Recurrent Neural Network

Process a sequence of vectors \mathbf{x} by applying a recurrence at every time step:

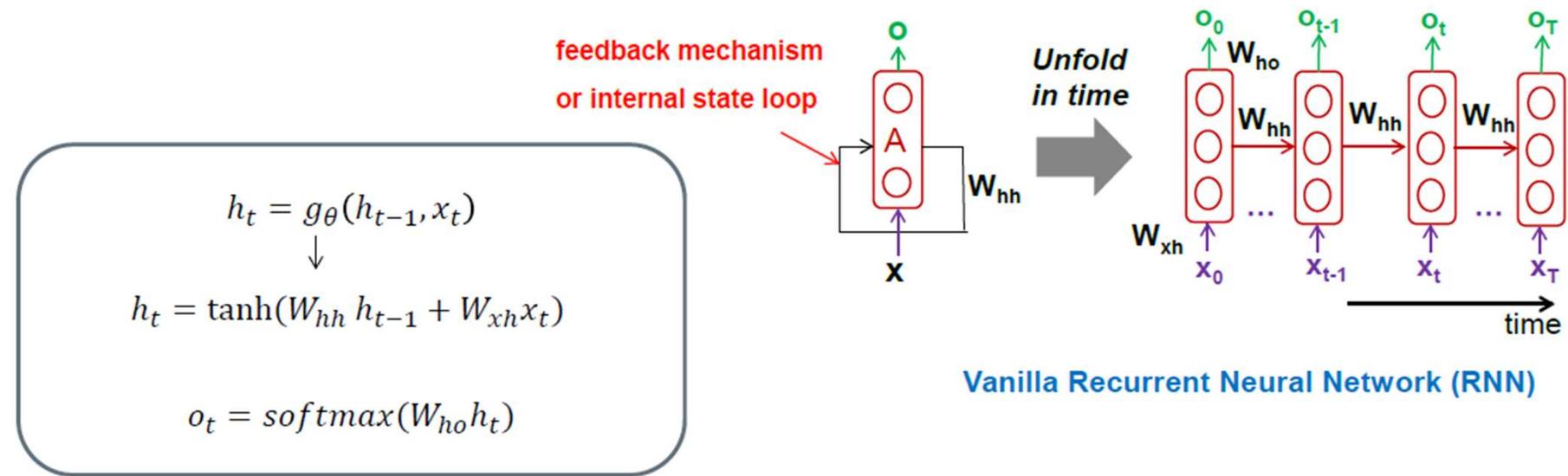


Remark: The same function g and same set of parameters W are used at every time step



(Vanilla) Recurrent Neural Network

Process a sequence of vectors \mathbf{x} by applying a recurrence at every time step:



Remark: RNN's can be seen as **selective summarization** of input sequence in a fixed-size state/hidden vector via a recursive update.

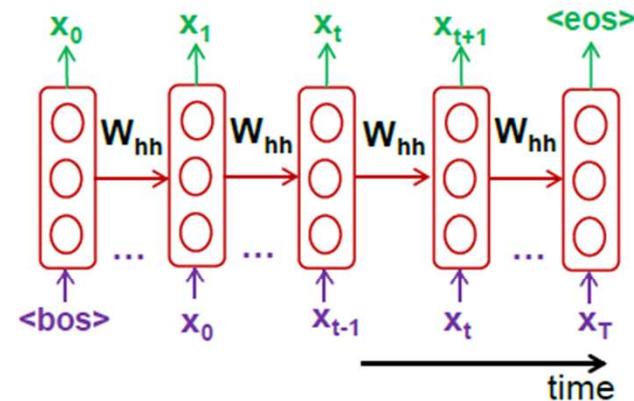


RNN : Probabilistic Interpretation

RNN as a generative model

- induces a set of procedures to model the conditional distribution of x_{t+1} given $x \leq t$ for all $t = 1, \dots, T$

$$P(x) = P(x_1, \dots, x_T) = \sum_{t=1}^T P(x_t | x_{t-1}, x_{t-2}, \dots, x_1)$$



- Think of the output as the probability distribution of the x_t given the previous ones in the sequence
- Training: Computing probability of the sequence and Maximum likelihood training

Generative Recurrent Neural Network (RNN)

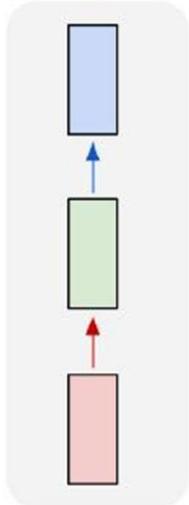
$$L_t = -\log P(x_t | x_{t-1}, x_{t-2}, \dots, x_1)$$

Details: <https://www.cs.cmu.edu/~epxing/Class/10708-17/project-reports/project10.pdf>



RNN : Computational Graphs

one to one

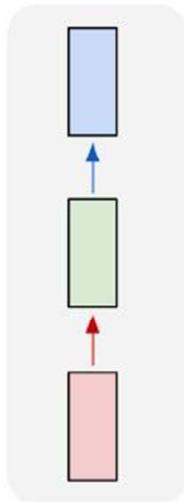


Vanilla Neural Networks

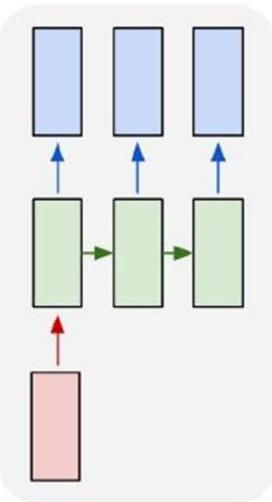


RNN : Computational Graphs

one to one



one to many

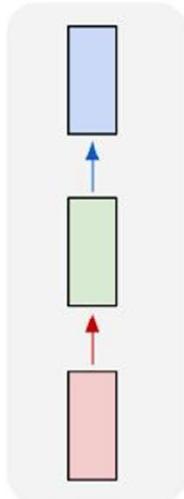


e.g. **Image Captioning**
image -> sequence of words

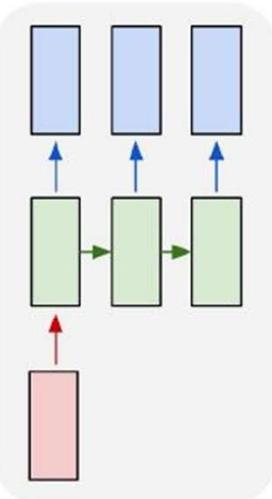


RNN : Computational Graphs

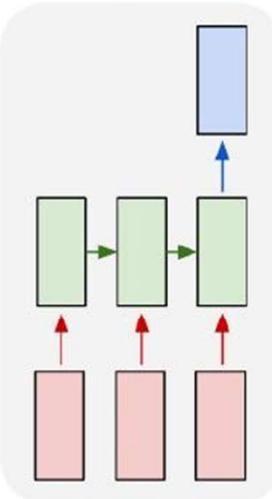
one to one



one to many



many to one

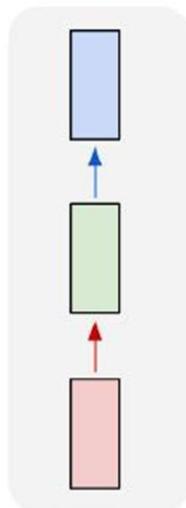


e.g. **action prediction**
sequence of video frames -> action class

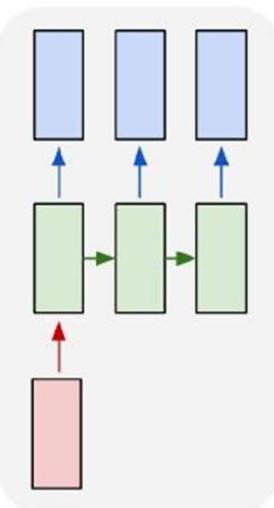


RNN : Computational Graphs

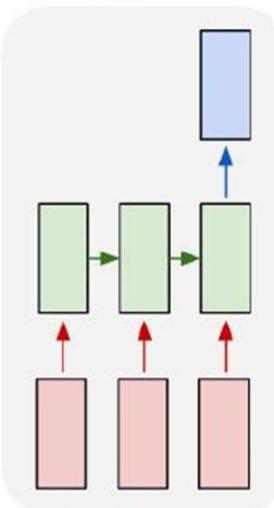
one to one



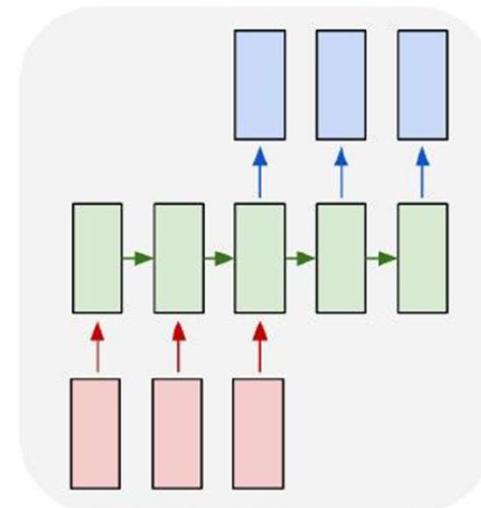
one to many



many to one



many to many



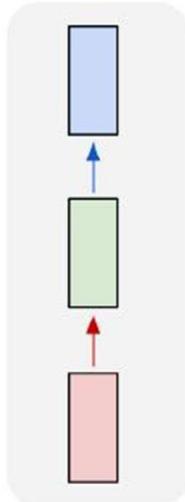
E.g. Video Captioning

Sequence of video frames -> caption

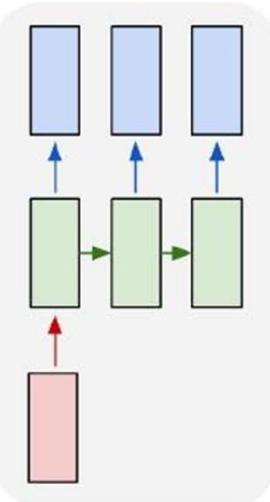


RNN : Computational Graphs

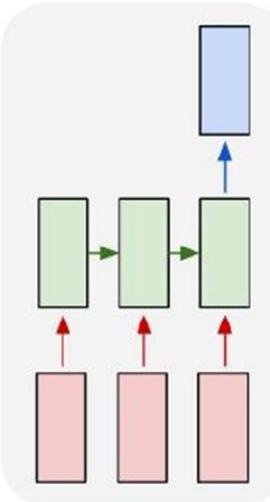
one to one



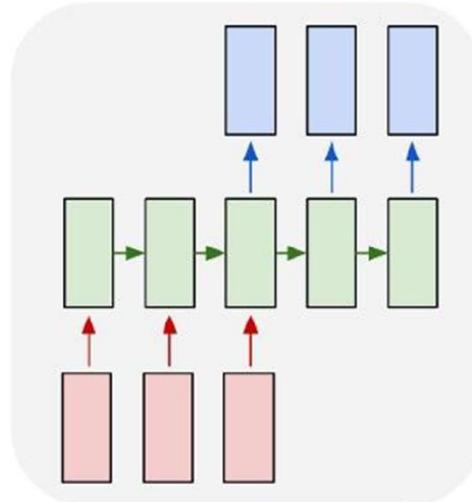
one to many



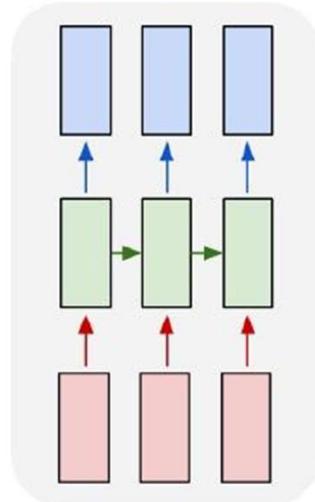
many to one



many to many



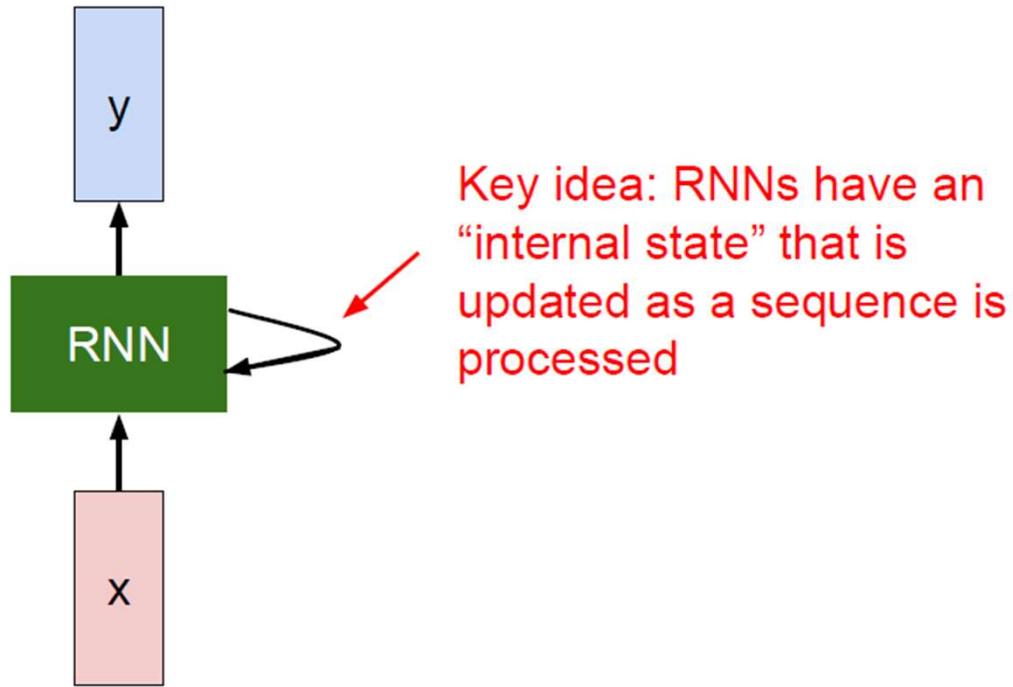
many to many



e.g. Video classification on frame level

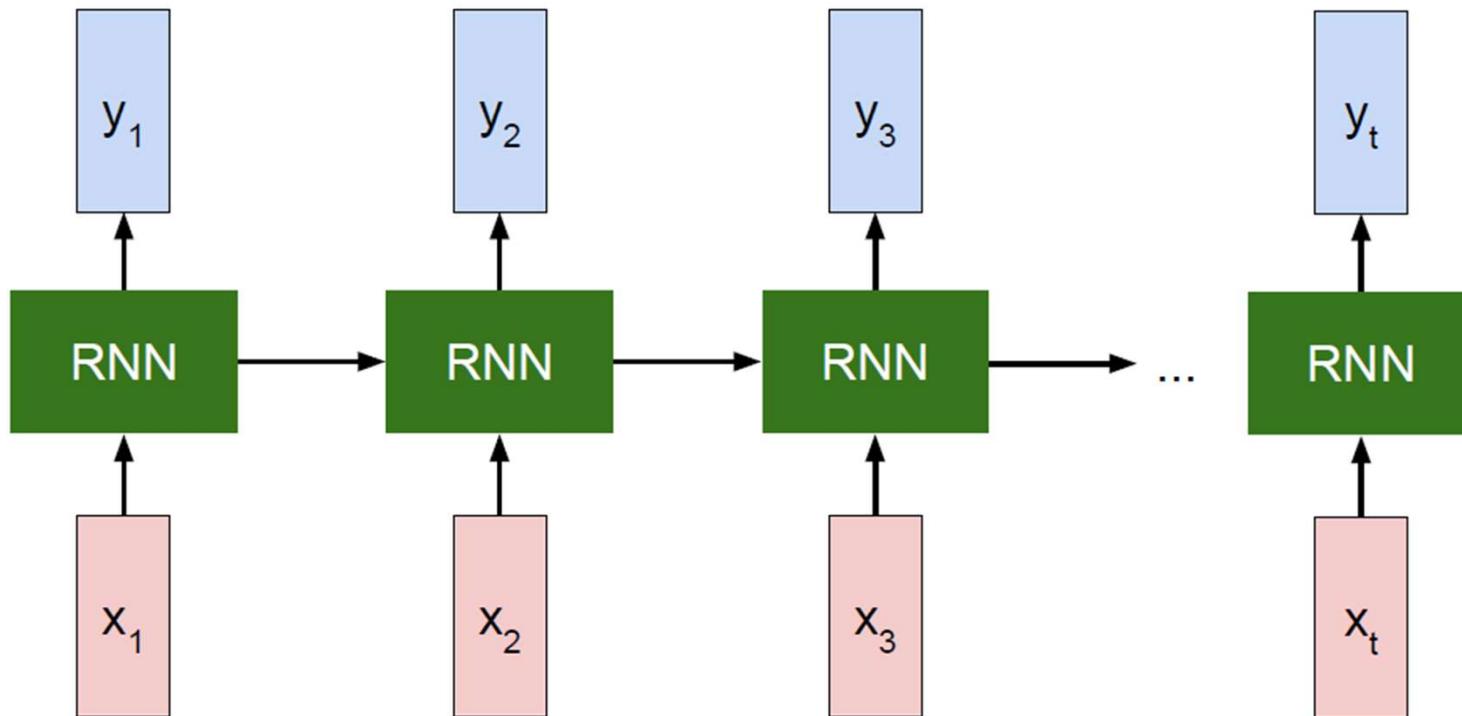


Recurrent Neural Network





RNN Unrolled



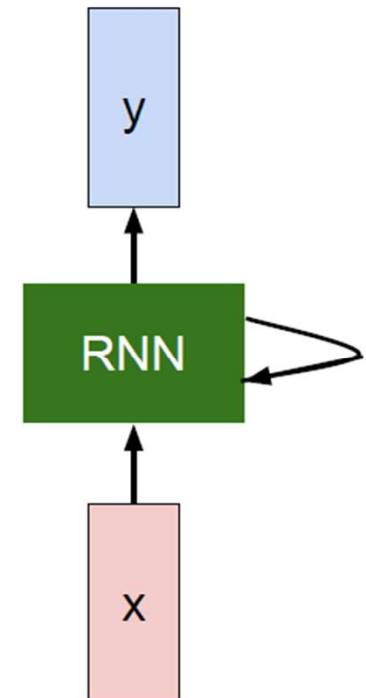


RNN hidden state update

We can process a sequence of vectors \mathbf{x} by applying a **recurrence formula** at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state \old state input vector at
some function some time step
with parameters W



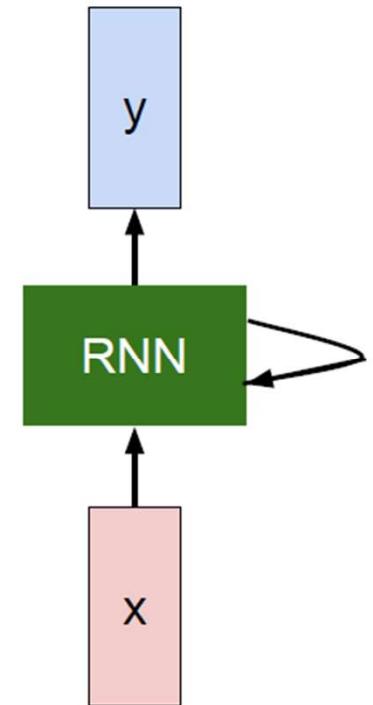


RNN output generation

We can process a sequence of vectors \mathbf{x} by applying a **recurrence formula** at every time step:

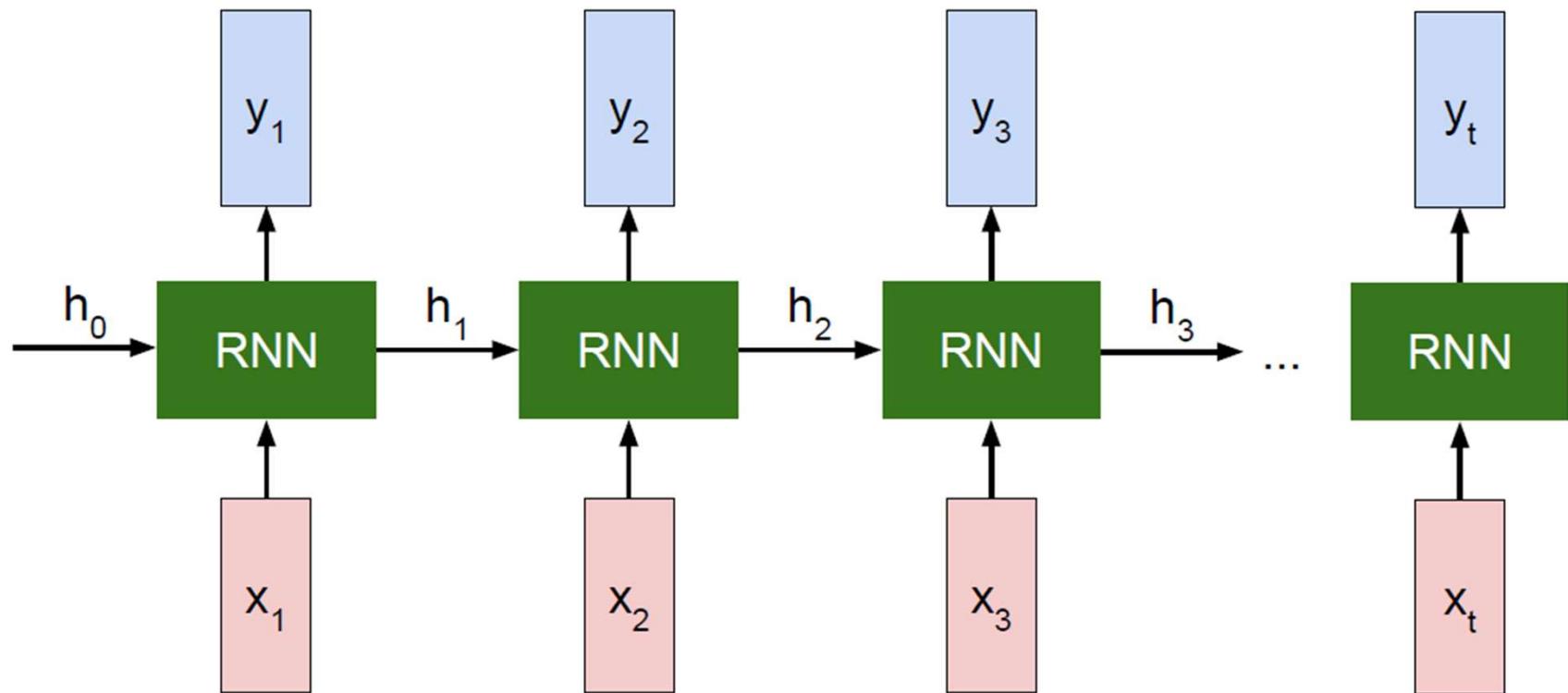
$$y_t = f_{W_{hy}}(h_t)$$

output new state
another function
with parameters W_o





Recurrent Neural Network



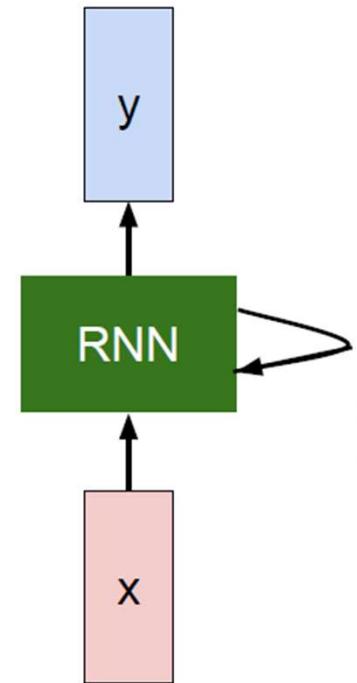


Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a **recurrence formula** at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

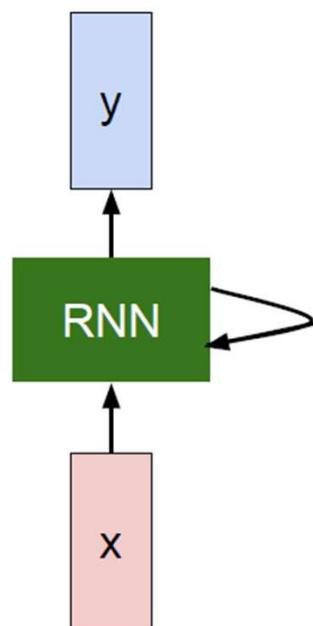
Notice: the same function and the same set of parameters are used at every time step.





Vanilla RNN

The state consists of a single “*hidden*” vector \mathbf{h} :



$$h_t = f_W(h_{t-1}, x_t)$$



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

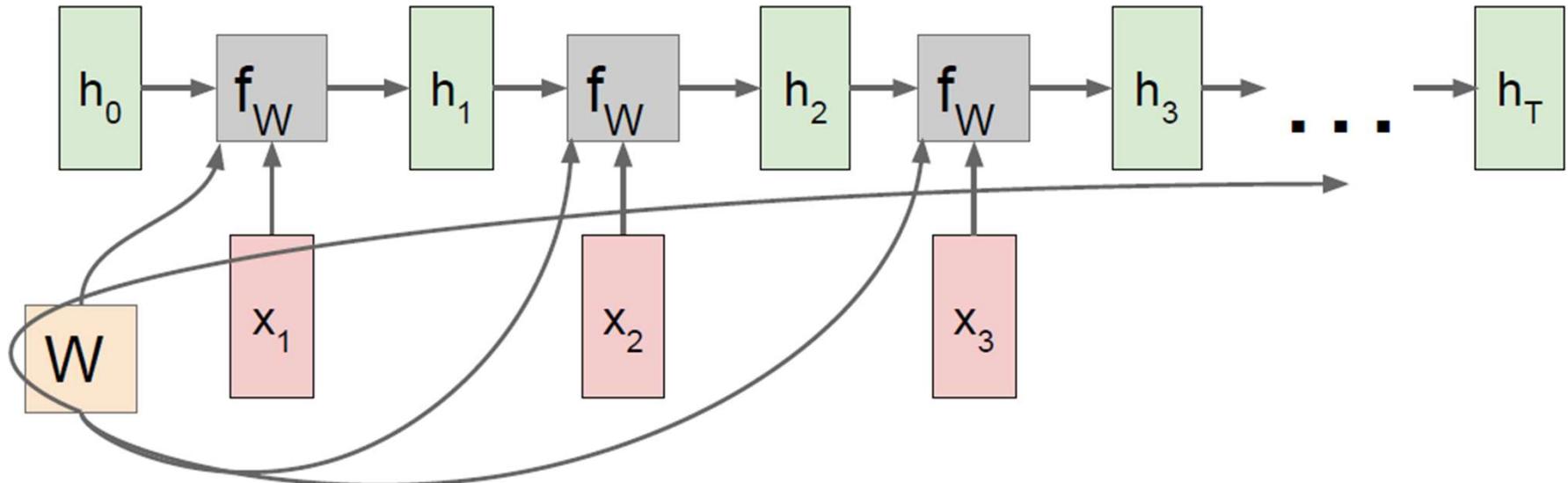
$$y_t = W_{hy}h_t$$

Sometimes called a “Vanilla RNN” or an “Elman RNN” after Prof. Jeffrey Elman



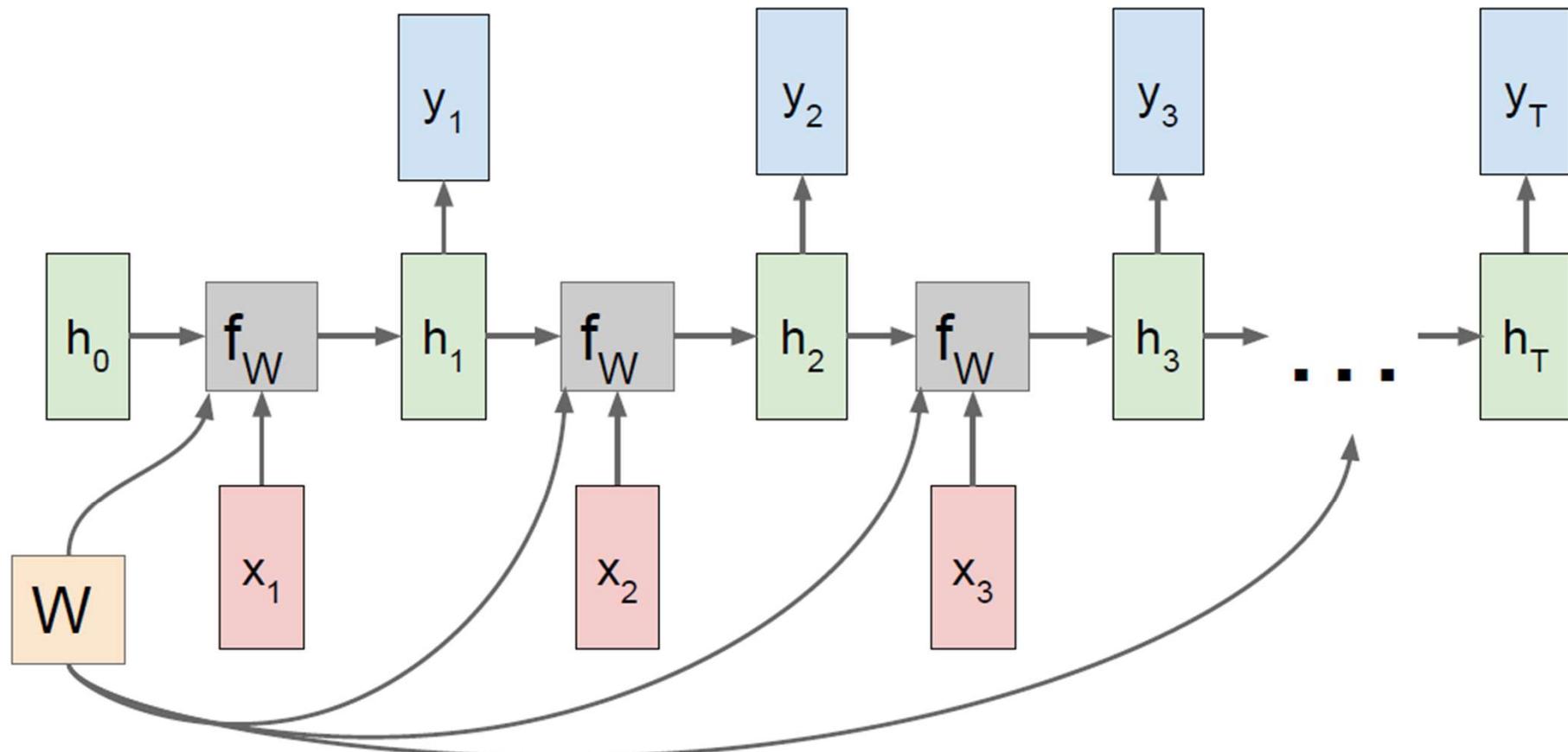
RNN: Computational Graphs

Re-use the same weight matrix at every time-step



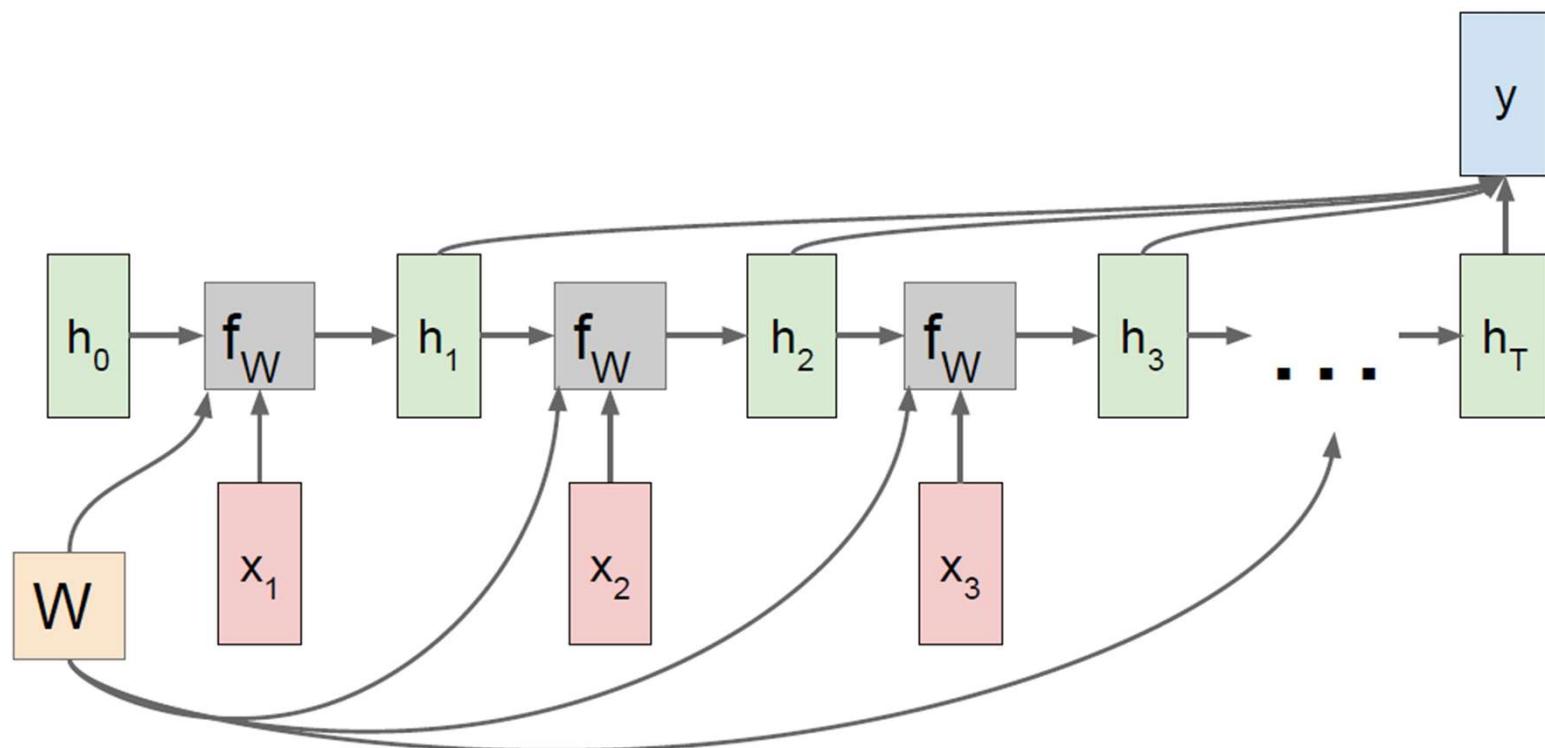


RNN: Computational Graphs: Many to Many



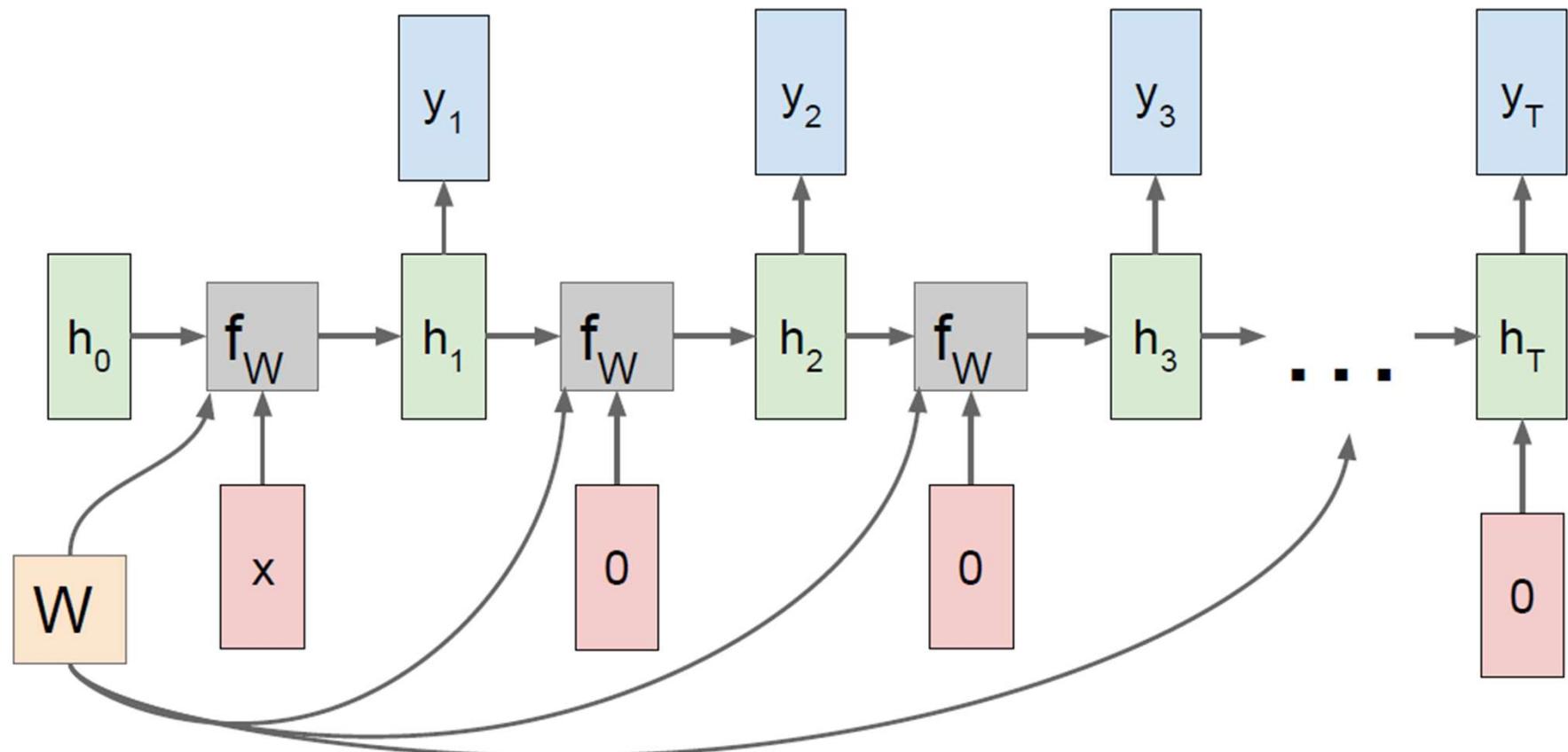


RNN: Computational Graphs: Many to One





RNN: Computational Graphs: One to Many

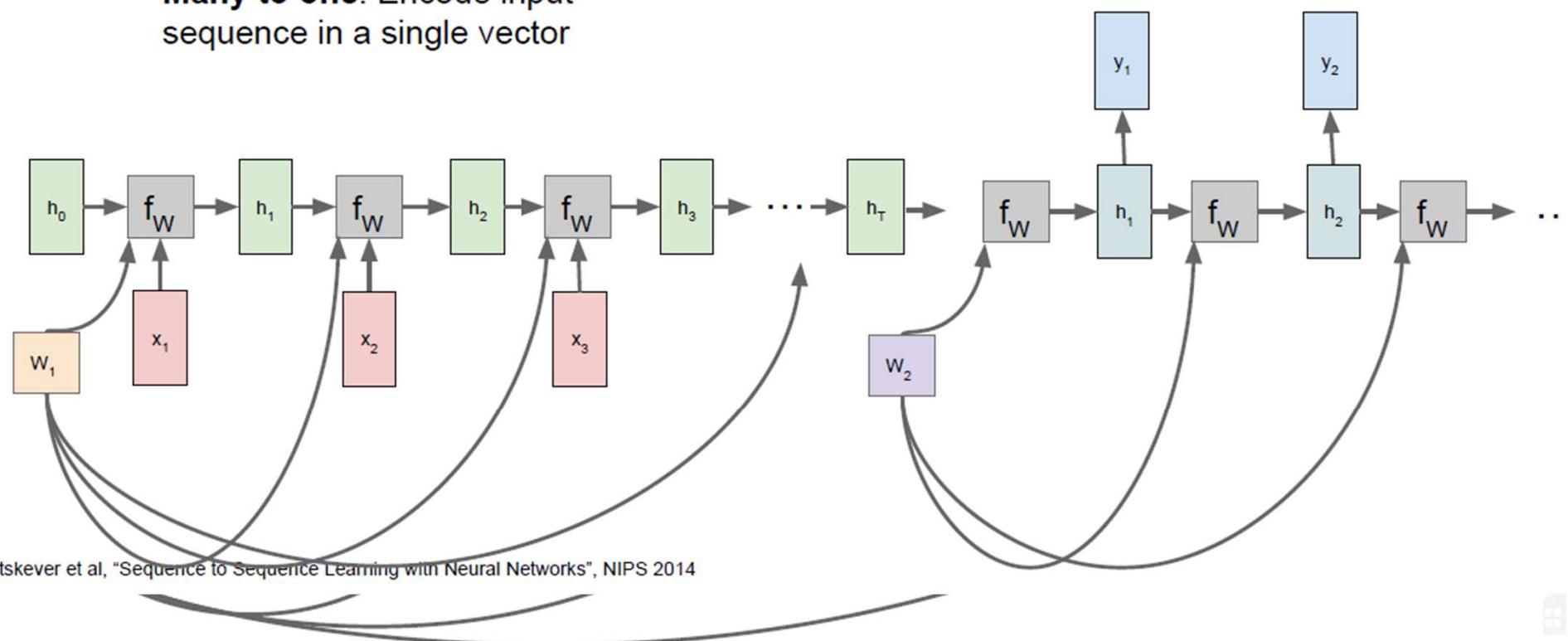




RNN: Computational Graphs: One to Many

Many to one: Encode input sequence in a single vector

One to many: Produce output sequence from single input vector

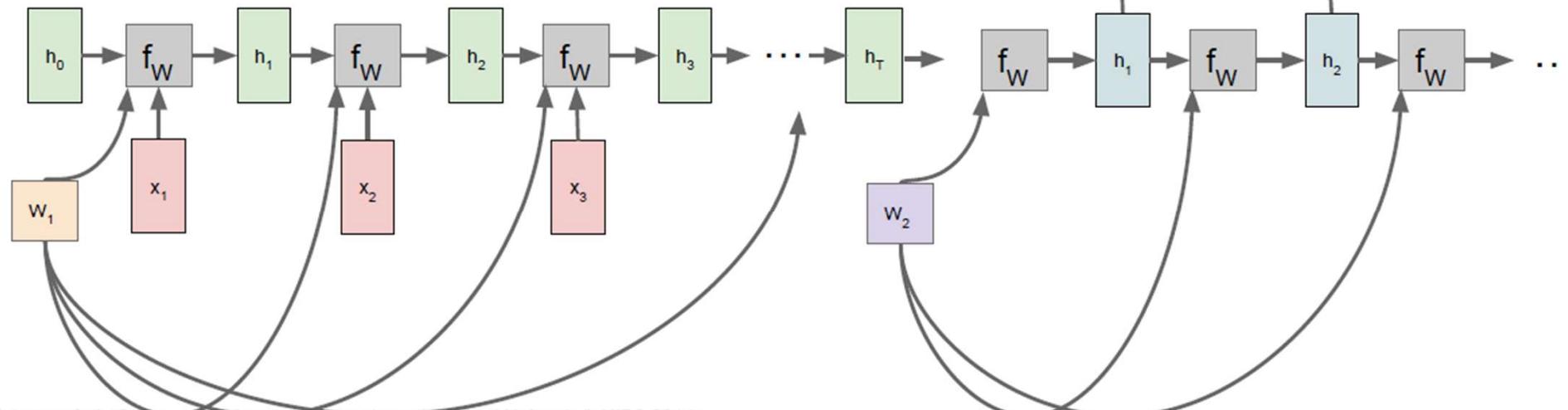


Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014



Seq. to Seq. : Many-to-one + one-to-many

Many to one: Encode input sequence in a single vector



Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

One to many: Produce output sequence from single input vector

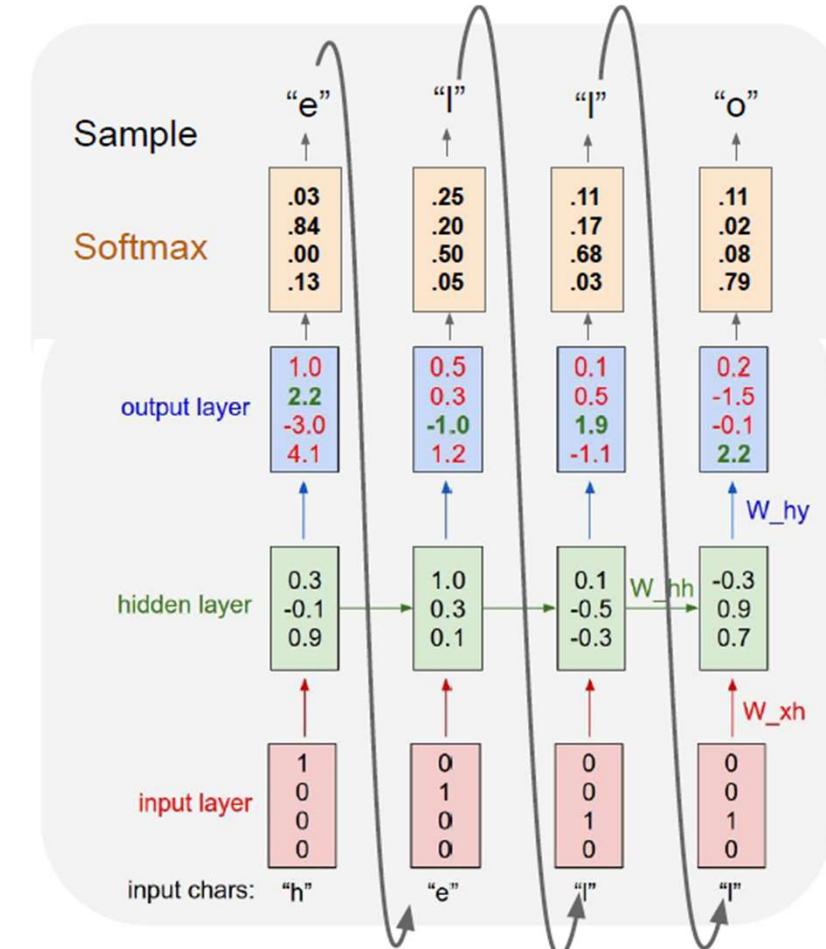


RNN Example

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample
characters one at a time, feed
back to model



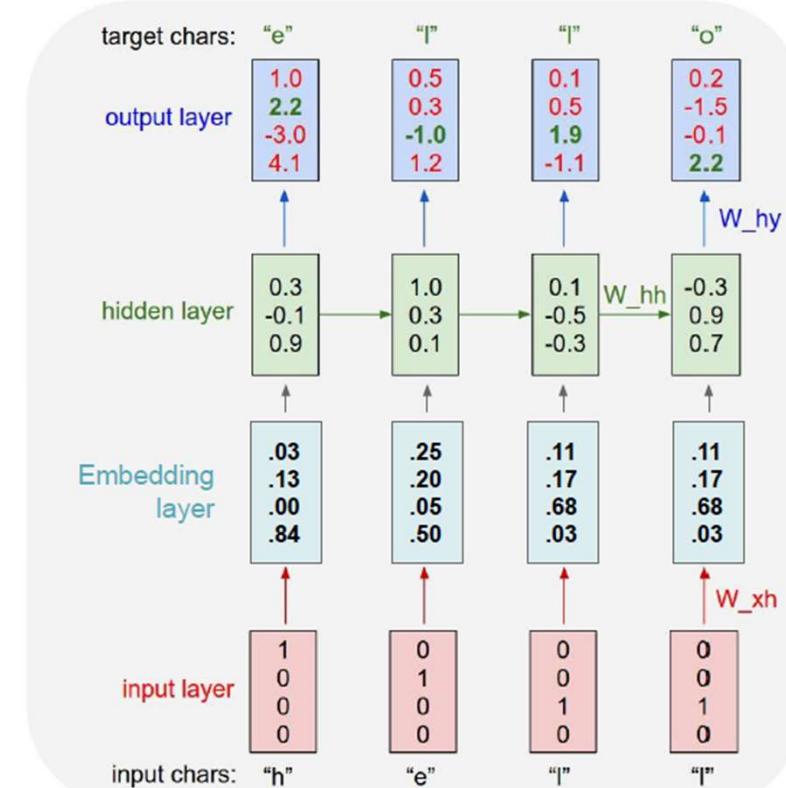


RNN Example

Example: Character-level Language Model Sampling

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \end{bmatrix} [1] \quad [w_{11}] \\ \begin{bmatrix} w_{21} & w_{22} & w_{23} & w_{14} \end{bmatrix} [0] = [w_{21}] \\ \begin{bmatrix} w_{31} & w_{32} & w_{33} & w_{14} \end{bmatrix} [0] \quad [w_{31}] \\ [0] \end{math>$$

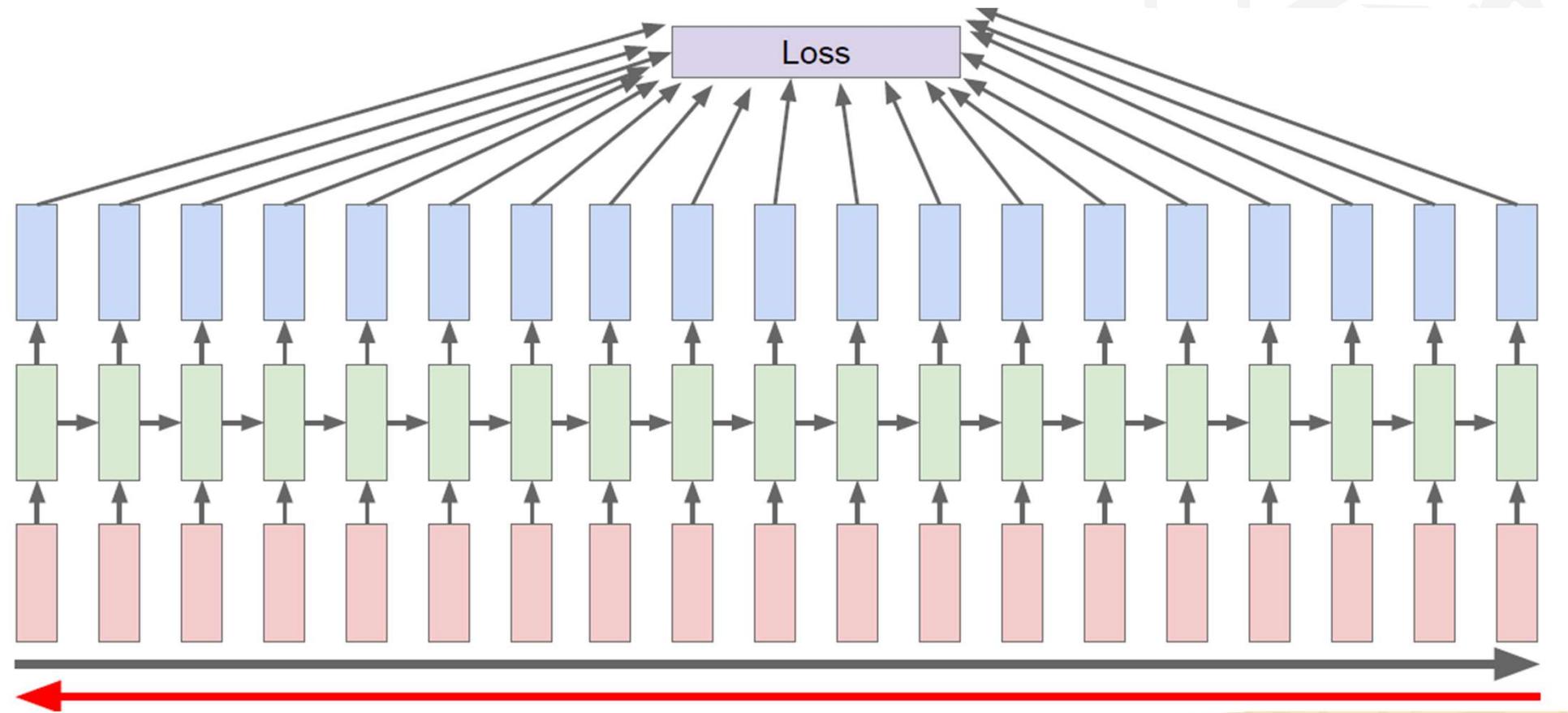
Matrix multiply with a one-hot vector just extracts a column from the weight matrix. We often put a separate **embedding layer** between input and hidden layers.





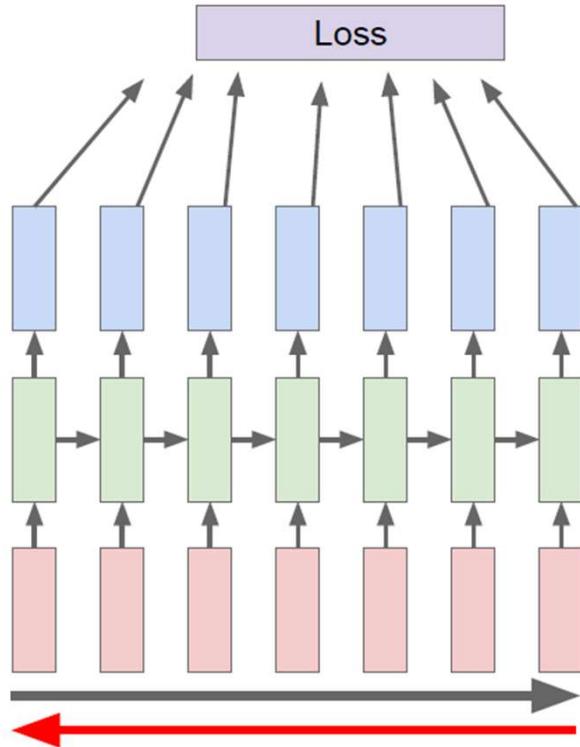
Backpropagation through time

Forward through entire sequence to compute loss,
then backward through entire sequence to compute gradient

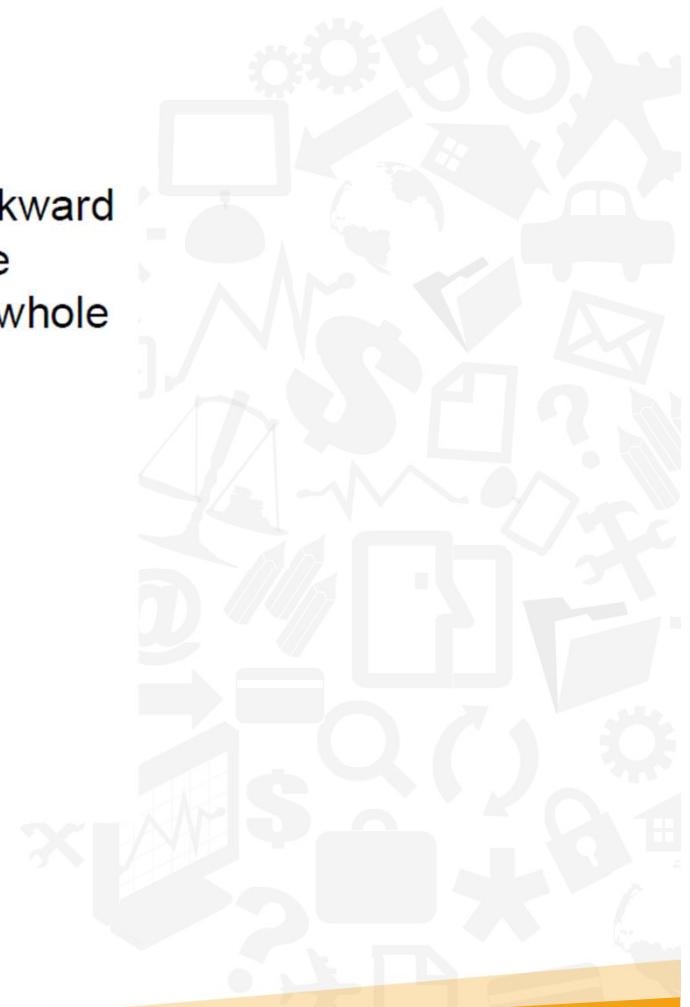




Truncated Backpropagation through time

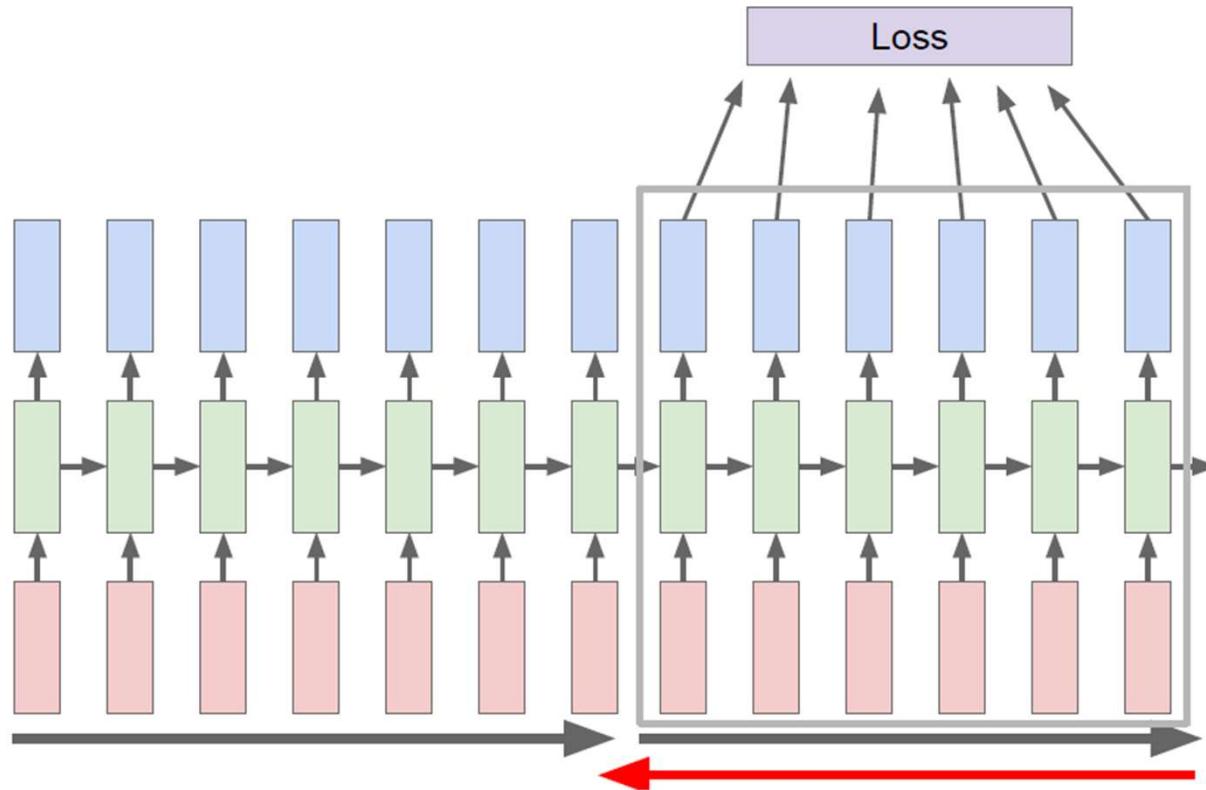


Run forward and backward
through chunks of the
sequence instead of whole
sequence





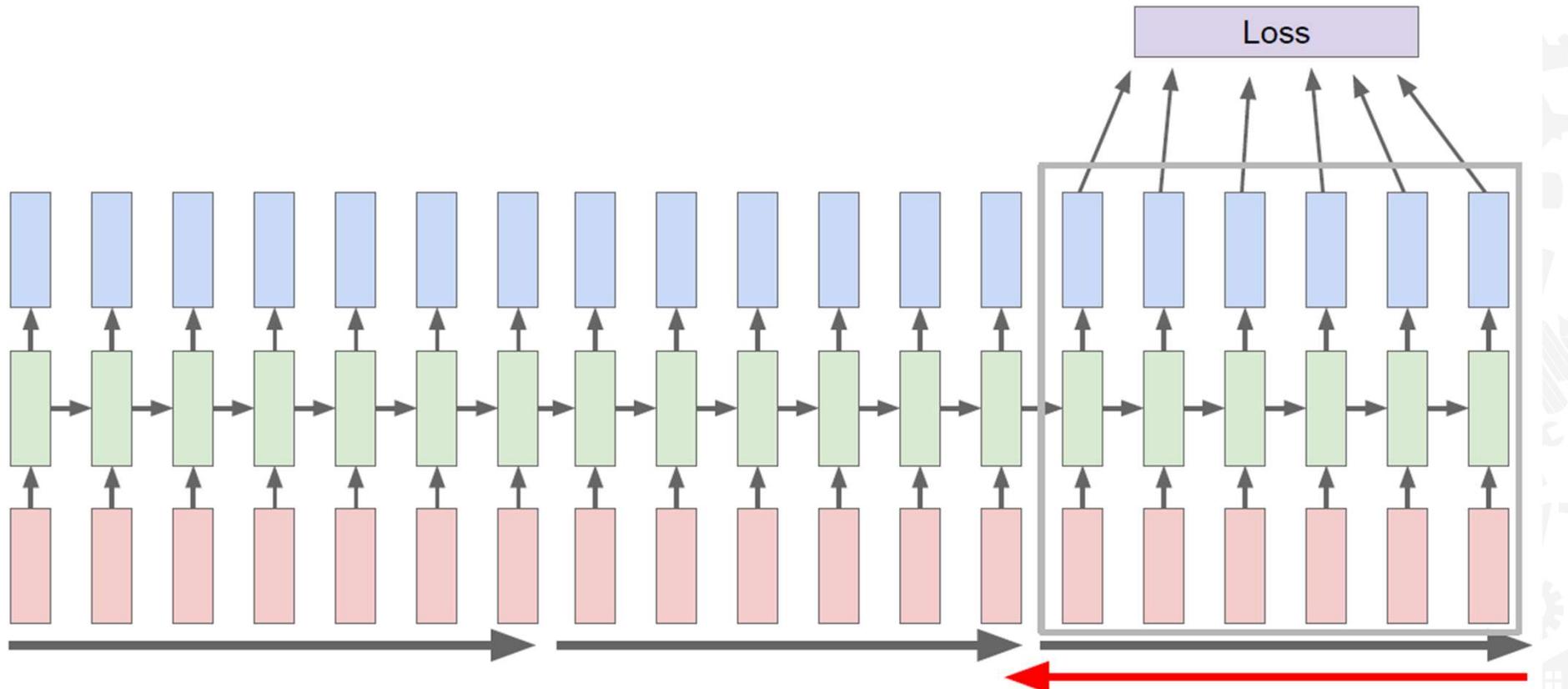
Truncated Backpropagation through time



Carry hidden states
forward in time forever,
but only backpropagate
for some smaller
number of steps



Truncated Backpropagation through time





RNN trade-offs



RNN Advantages:

- Can process any length input
- Computation for step t can (in theory) use information from many steps back
- Model size doesn't increase for longer input
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

RNN Disadvantages:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back



Example: Image Captioning

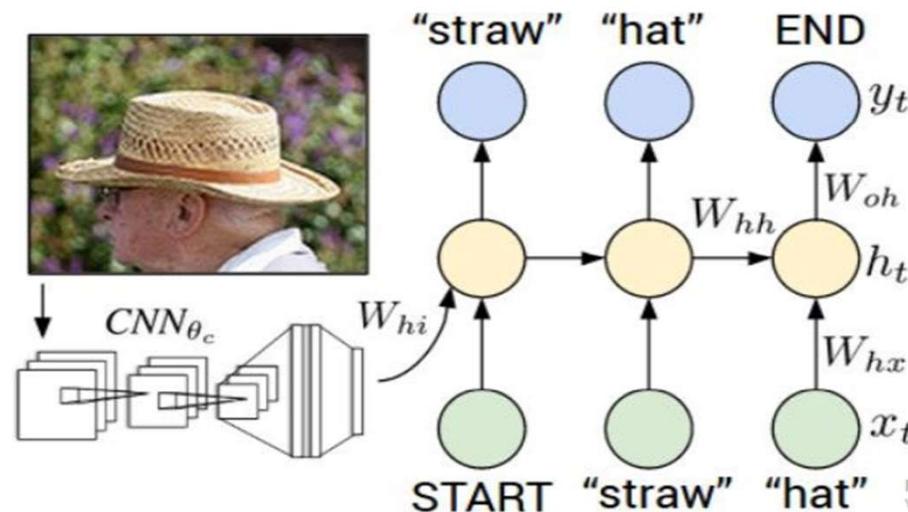


Figure from Karpathy et al, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.
Reproduced for educational purposes.

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.

Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

Show and Tell: A Neural Image Caption Generator, Vinyals et al.

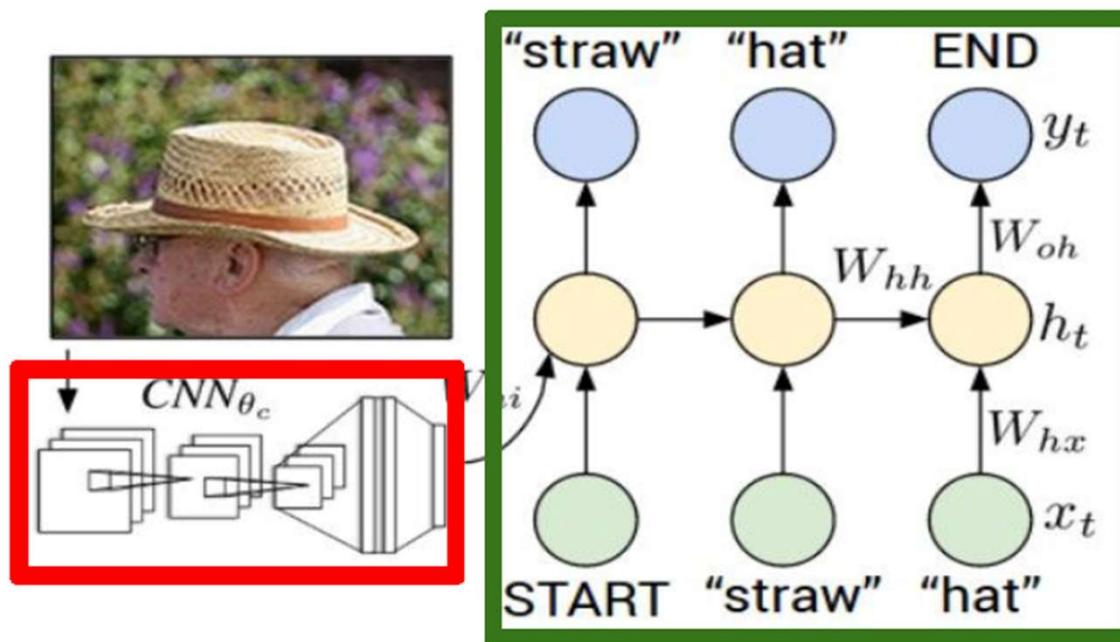
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.

Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick



Example: Image Captioning

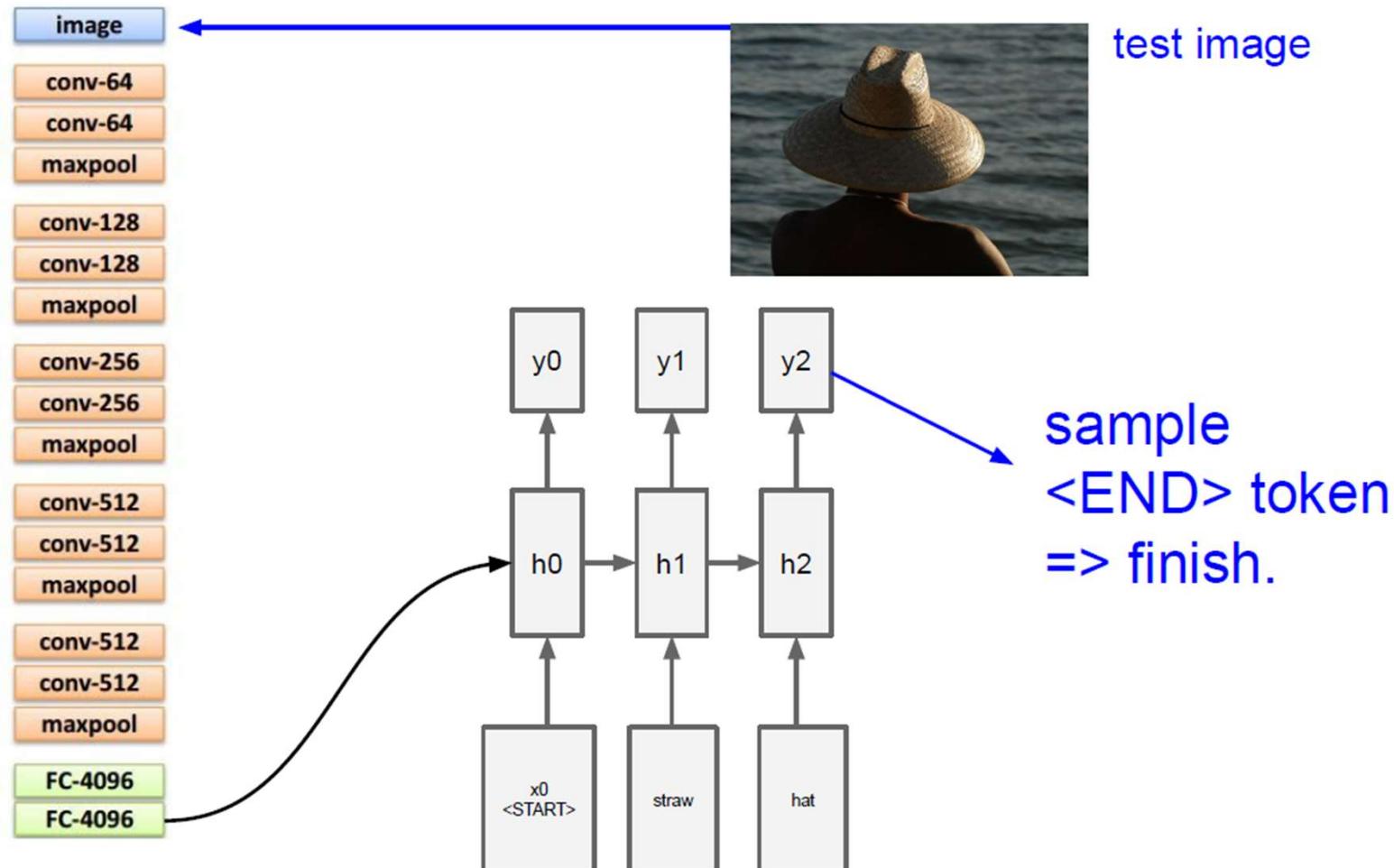
Recurrent Neural Network



Convolutional Neural Network



Example: Image Captioning





Example: Image Captioning



A cat sitting on a suitcase on the floor



A cat is sitting on a tree branch



A dog is running in the grass with a frisbee



A white teddy bear sitting in the grass



Two people walking on the beach with surfboards



A tennis player in action on the court



Two giraffes standing in a grassy field



A man riding a dirt bike on a dirt track



Example: Visual Question Answering(VQA)



Q: What endangered animal is featured on the truck?

- A: A bald eagle.
- A: A sparrow.
- A: A humming bird.
- A: A raven.



Q: Where will the driver go if turning right?

- A: Onto 24 1/4 Rd.
- A: Onto 25 1/4 Rd.
- A: Onto 23 1/4 Rd.
- A: Onto Main Street.



Q: When was the picture taken?

- A: During a wedding.
- A: During a bar mitzvah.
- A: During a funeral.
- A: During a Sunday church service



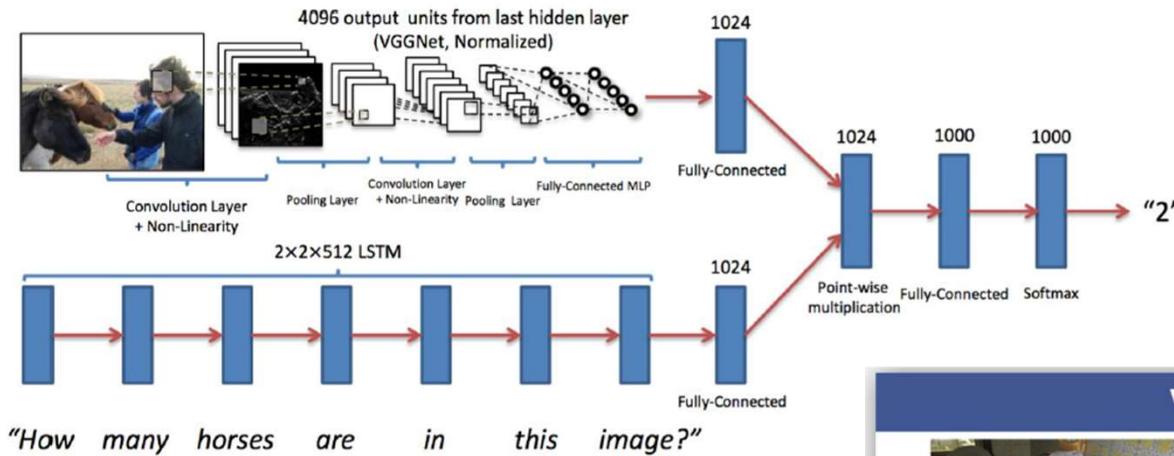
Q: Who is under the umbrella?

- A: Two women.
- A: A child.
- A: An old man.
- A: A husband and a wife.

Agrawal et al, "VQA: Visual Question Answering", ICCV 2015
Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016
Figure from Zhu et al, copyright IEEE 2016. Reproduced for educational purposes.



Example: Visual Question Answering(VQA)

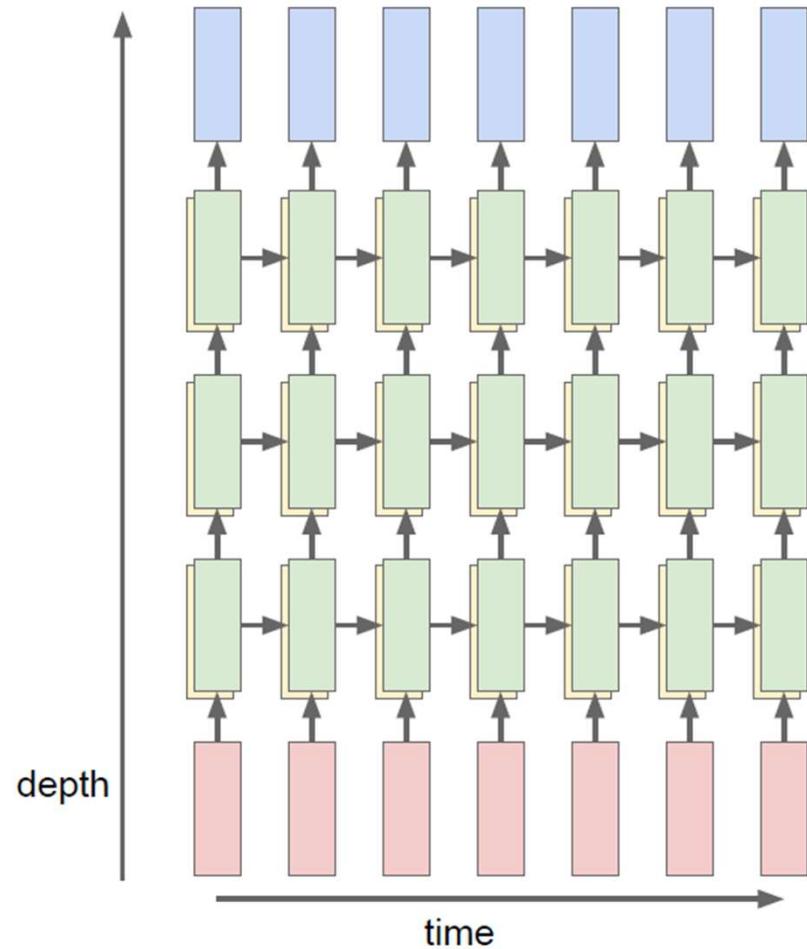


Agrawal et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2015
Figures from Agrawal et al, copyright IEEE 2015. Reproduced for educational purposes.

The screenshot shows a "Visual Dialog" interface. At the top, it says "Visual Dialog". Below that is a image of a cat drinking water from a mug. To the right, the text "A cat drinking water out of a coffee mug." is displayed. A series of messages follow:
User: What color is the mug?
Bot: White and red
User: Are there any pictures on it?
Bot: No, something is there can't tell what it is
User: Is the mug and cat on a table?
Bot: Yes, they are
User: Are there other items on the table?
Bot: Yes, magazines, books, toaster and basket, and a plate



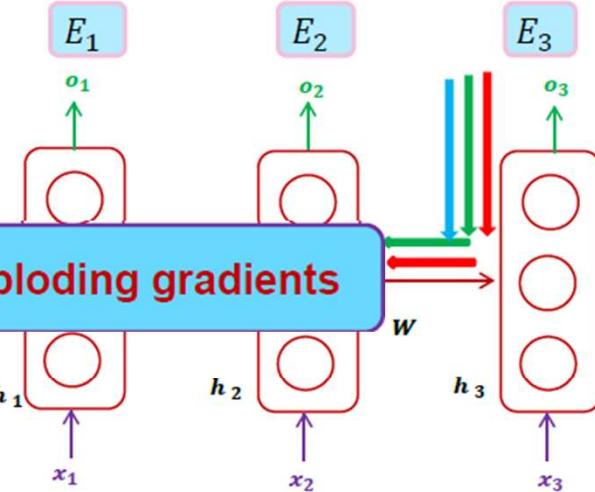
Multilayer RNNs



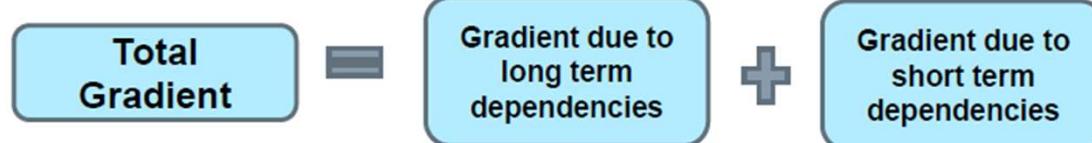


Vanishing vs Exploding Gradient

$$\begin{aligned}\frac{\partial E_3}{\partial W} &= \frac{\partial E_3}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W} + \frac{\partial E_3}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial W} + \frac{\partial E_3}{\partial h_3} \frac{\partial h_3}{\partial h_3} \frac{\partial h_3}{\partial W} \\ &= \lll 1 \quad + \quad \ll 1 \quad + \quad < 1\end{aligned}$$



Repeated matrix multiplications leads to vanishing and exploding gradients



Remark: The gradients due to short term dependencies (just previous dependencies) dominates the gradients due to long-term dependencies.

This means network will tend to focus on short term dependencies which is often not desired

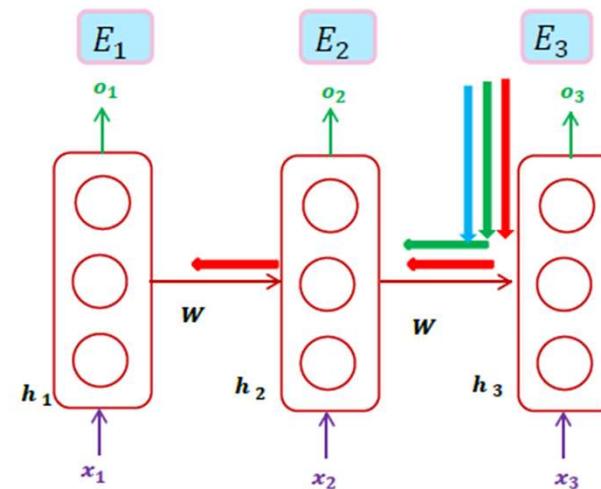
Problem of Vanishing Gradient



Vanishing vs Exploding Gradient

$$\begin{aligned}\frac{\partial E_3}{\partial W} &= \frac{\partial E_3}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W} + \frac{\partial E_3}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial W} + \frac{\partial E_3}{\partial h_3} \frac{\partial h_3}{\partial h_3} \frac{\partial h_3}{\partial W} \\ &= \ggg 1 + \gggg 1 + \gggg 1 \\ &= \text{Very large number, i.e., NaN}\end{aligned}$$

Problem of Exploding Gradient





Vanishing vs Exploding Gradient

$$\left\| \frac{\partial h_3}{\partial h_k} \right\| \leq (\gamma_w \gamma_g)^{t-k}$$

For tanh or linear activation

$\gamma_w \gamma_g$ greater than 1
Gradient Explodes !!!

$\gamma_w \gamma_g$ less than 1
Gradient Vanishes !!!

Remark: This problem of exploding/vanishing gradient occurs because the same number is multiplied in the gradient repeatedly.

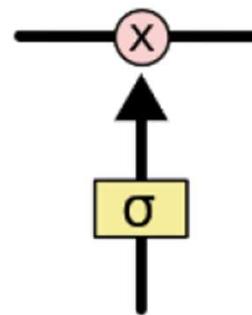


Long Short Term Memory (LSTM) : Gating

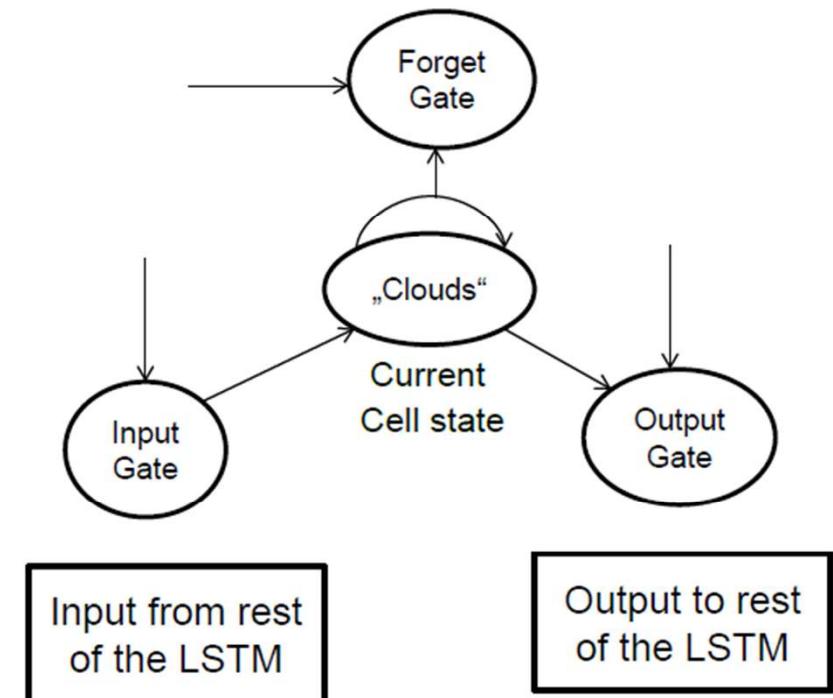
Gates :

- way to optionally let information through.
- composed out of a sigmoid neural net layer and a pointwise multiplication operation.
- remove or add information to the cell state

→ 3 gates in LSTM



→ gates to protect and control the cell state.

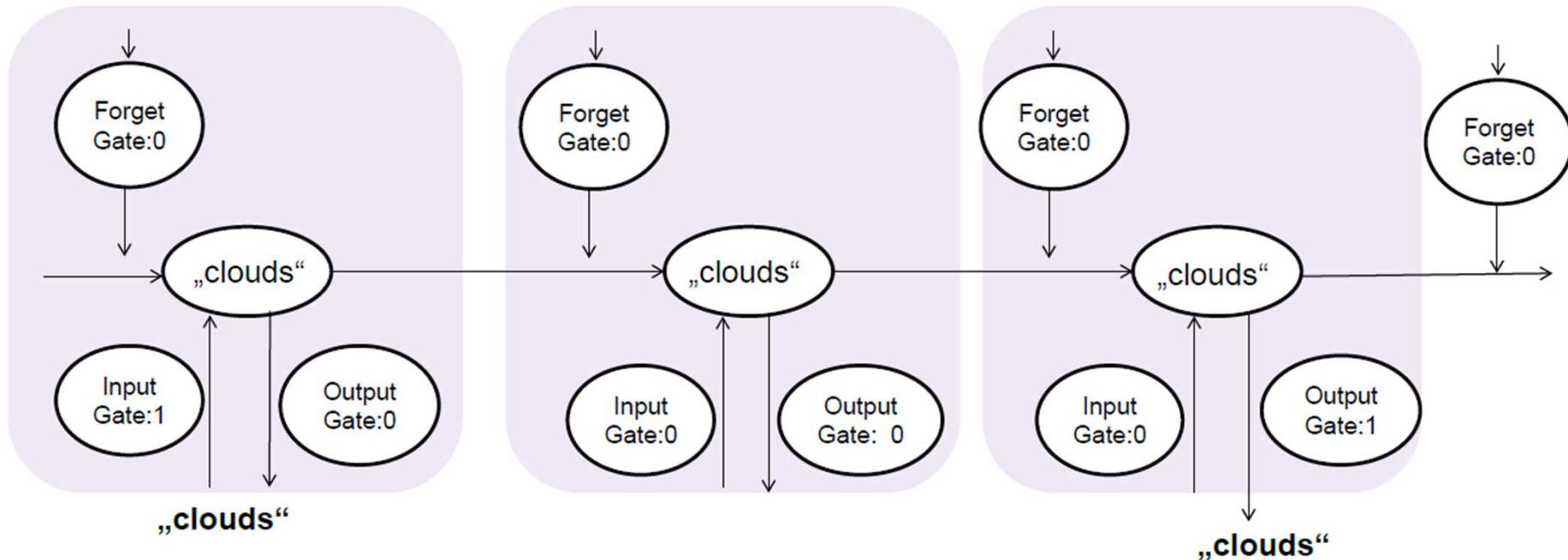


<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>



Long Short Term Memory (LSTM) : Gating

Remember the word „clouds“ over time....



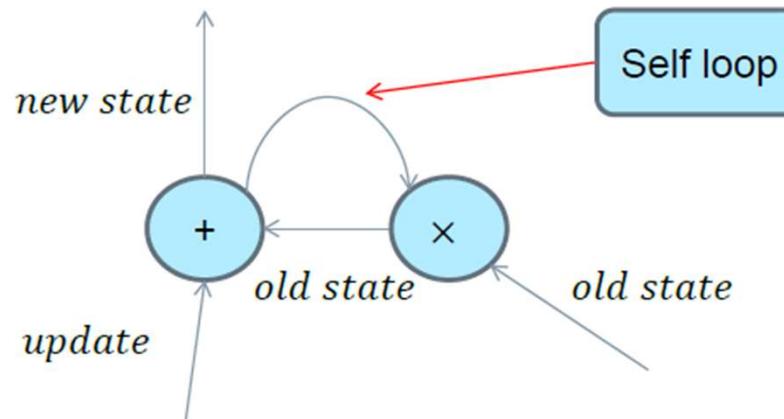
Lecture from the course Neural Networks for Machine Learning by Geoff Hinton



Long Short Term Memory (LSTM) : Gating

Motivation:

- Create a self loop path from where gradient can flow
- self loop corresponds to an eigenvalue of Jacobian to be slightly less than 1



$$\text{new state} = \text{old state} + \text{update}$$

$$\frac{\partial \text{new state}}{\partial \text{old state}} \cong \text{Identity}$$

LONG SHORT-TERM MEMORY, Sepp Hochreiter and Jürgen Schmidhuber

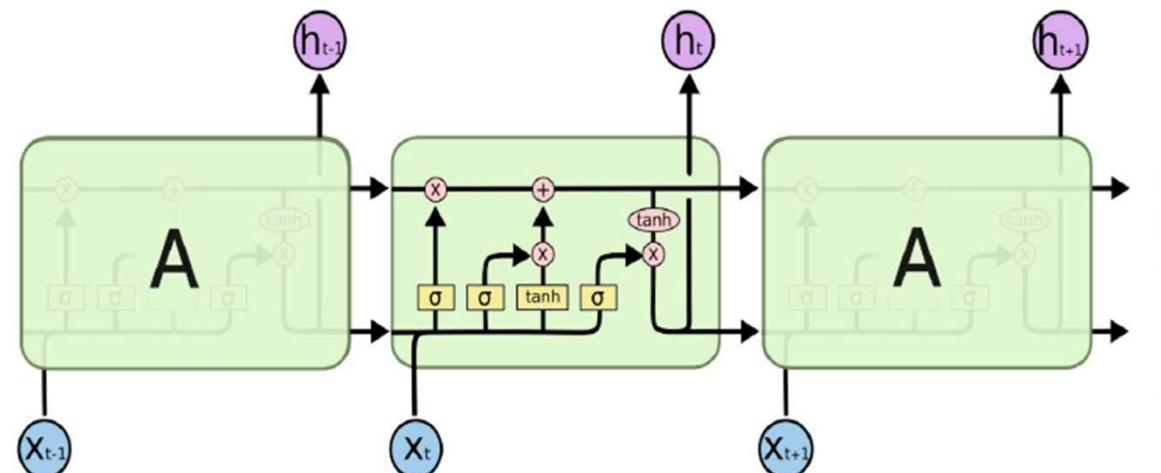
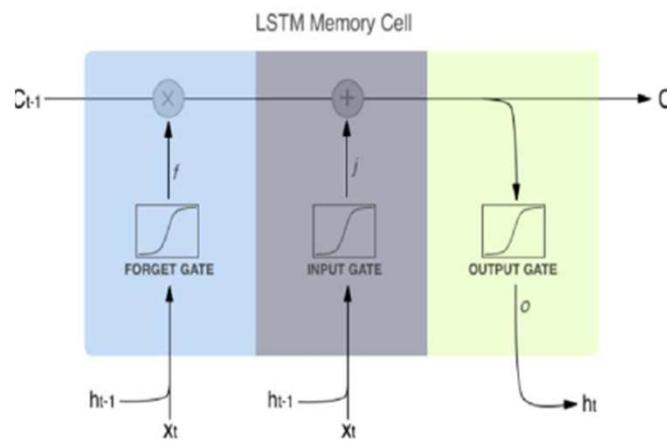
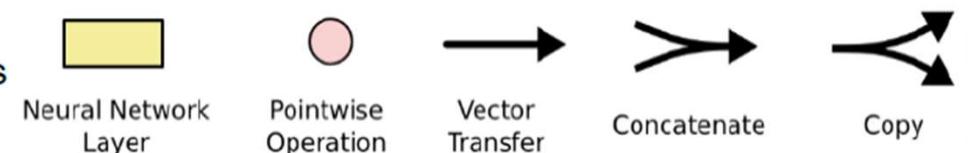


Long Short Term Memory (LSTM) : Step by Step

Key Ingredients

Cell state - transport the information through the units

Gates – optionally allow information passage



<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

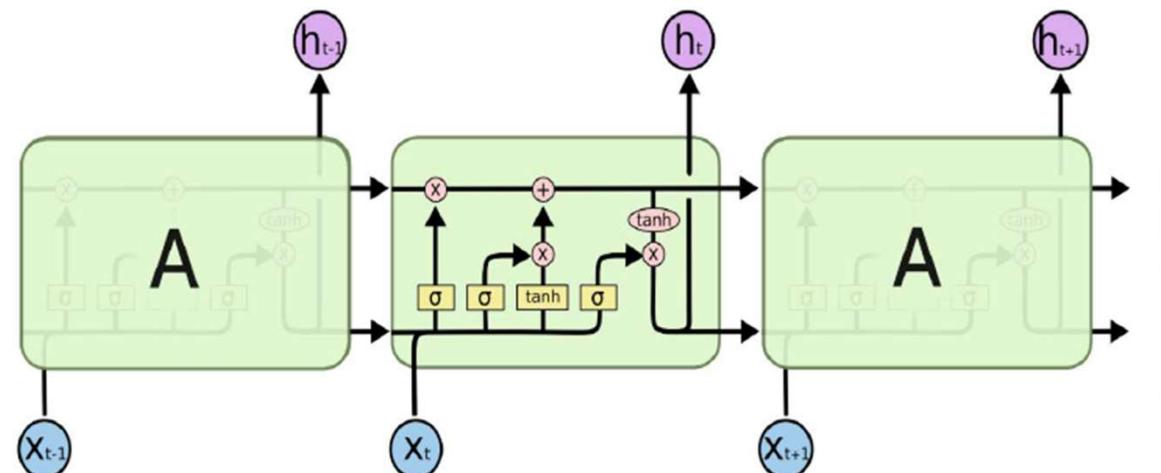
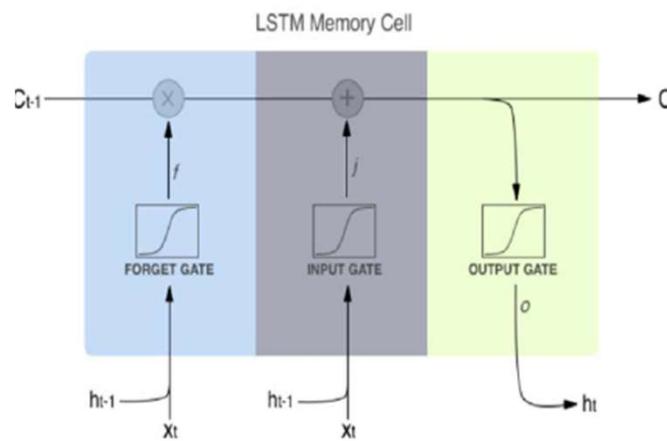
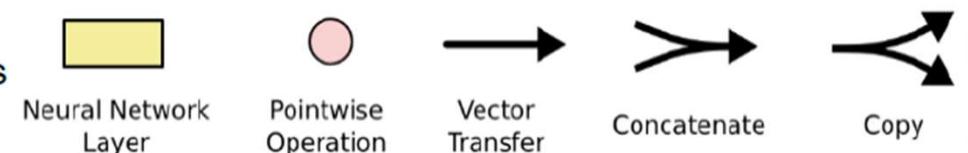


Long Short Term Memory (LSTM) : Step by Step

Key Ingredients

Cell state - transport the information through the units

Gates – optionally allow information passage



<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

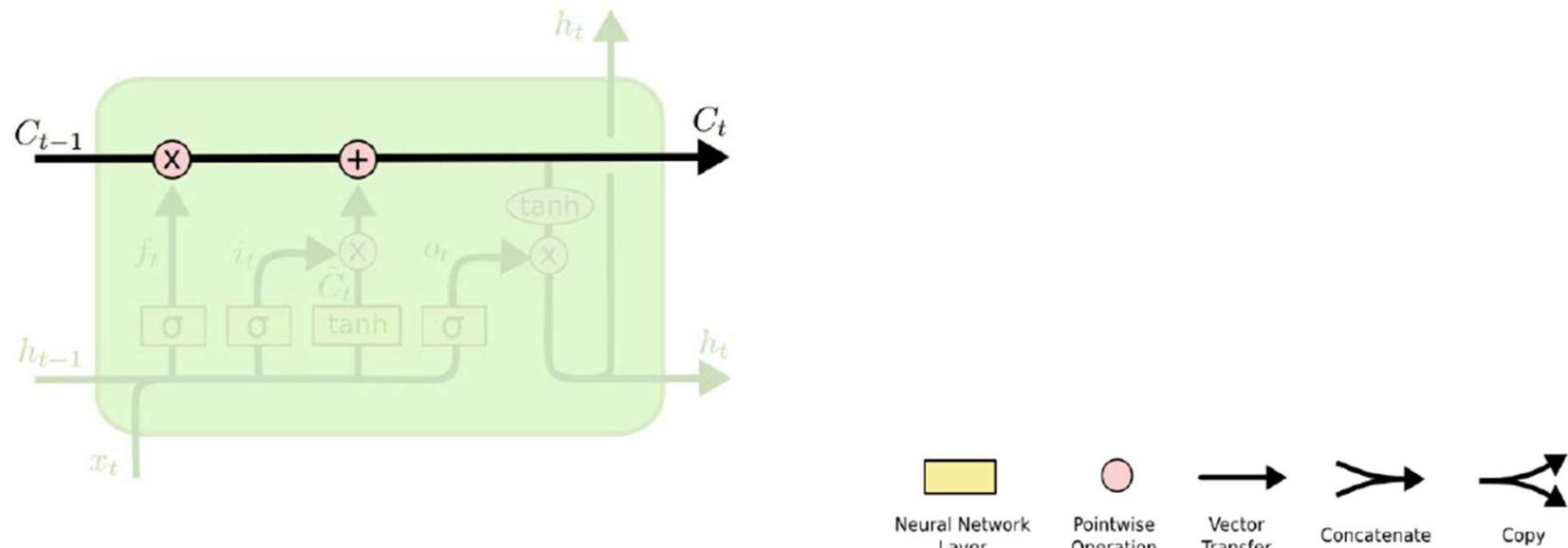


Long Short Term Memory (LSTM) : Step by Step

Cell: Transports information through the units (key idea)

→ the horizontal line running through the top

LSTM removes or adds information to the cell state using gates.





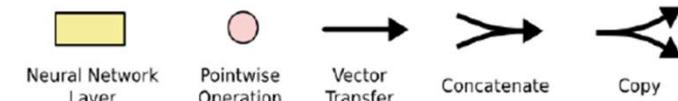
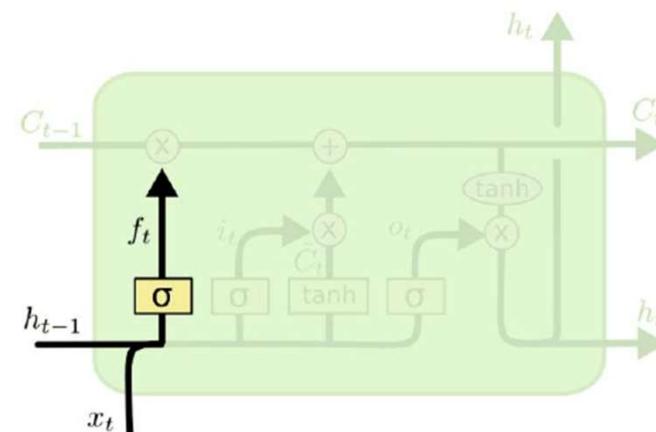
Long Short Term Memory (LSTM) : Step by Step

Forget Gate:

- decides what information to throw away or remember from the previous cell state
 - decision maker: sigmoid layer (*forget gate layer*)
- The output of the sigmoid lies between 0 to 1,
→ 0 being forget, 1 being keep.

$$f_t = \text{sigmoid}(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$$

- looks at h_{t-1} and x_t , and outputs a number between 0 and 1 for each number in the cell state C_{t-1}

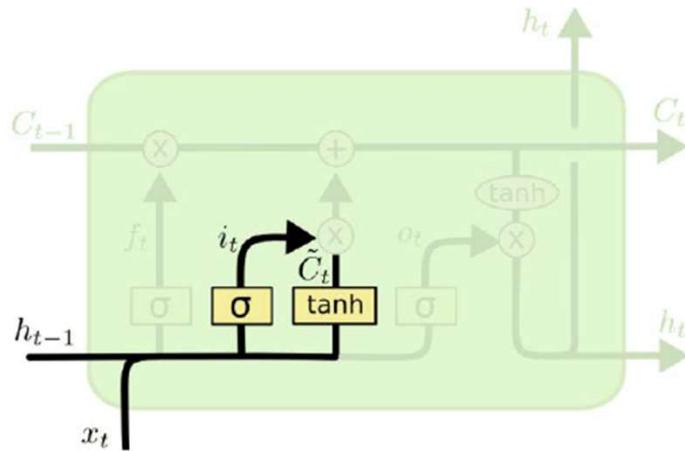




Long Short Term Memory (LSTM) : Step by Step

Input Gate: Selectively updates the cell state based on the new input.

A multiplicative input gate unit to protect the memory contents stored in j from perturbation by irrelevant inputs



$$i_t = \text{sigmoid}(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i)$$

$$\tilde{C}_t = \text{Tanh}(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g)$$

The next step is to decide what new information we're going to store in the cell state. This has two parts:

1. A sigmoid layer called the "input gate layer" decides which values we'll update.
2. A tanh layer creates a vector of new candidate values, \tilde{C}_t , that could be added to the state.

In the next step, we'll combine these two to create an update to the state.

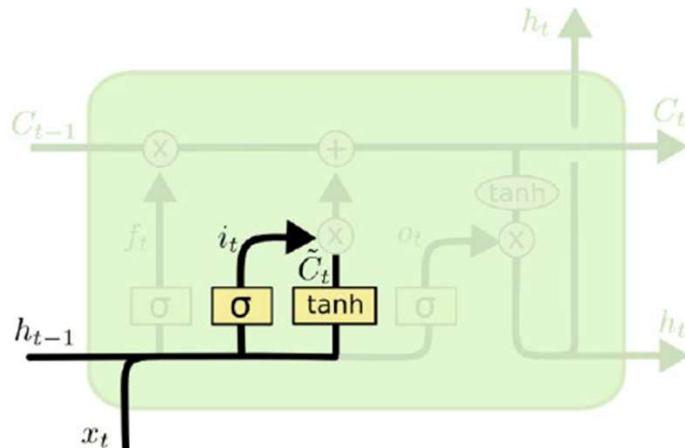




Long Short Term Memory (LSTM) : Step by Step

Input Gate: Selectively updates the cell state based on the new input.

A multiplicative input gate unit to protect the memory contents stored in j from perturbation by irrelevant inputs



$$i_t = \text{sigmoid}(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i)$$

$$\tilde{C}_t = \text{Tanh}(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g)$$

The next step is to decide what new information we're going to store in the cell state. This has two parts:

1. A sigmoid layer called the "input gate layer" decides which values we'll update.
2. A tanh layer creates a vector of new candidate values, \tilde{C}_t , that could be added to the state.

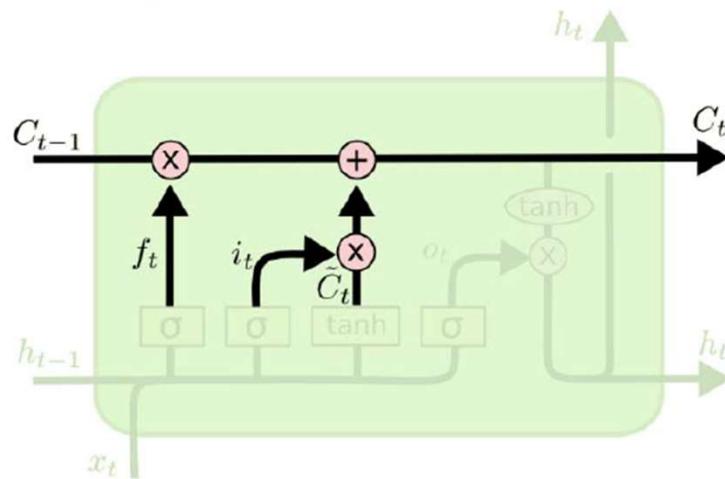
In the next step, we'll combine these two to create an update to the state.





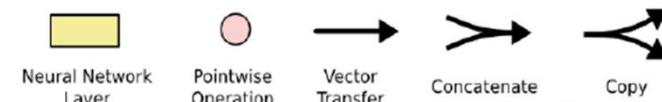
Long Short Term Memory (LSTM) : Step by Step

Cell Update



- update the old cell state, C_{t-1} , into the new cell state C_t
- multiply the old state by f_t , forgetting the things we decided to forget earlier
- add $i_t * \tilde{C}_t$ to get the new candidate values, scaled by how much we decided to update each state value.

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

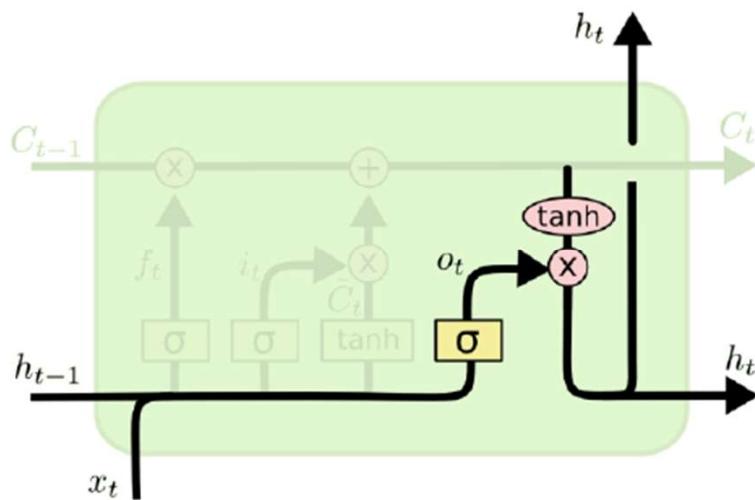




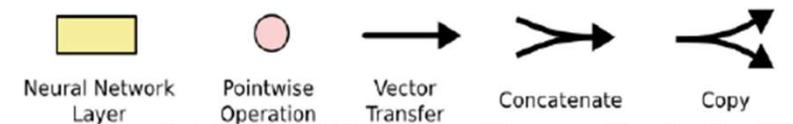
Long Short Term Memory (LSTM) : Step by Step

Output Gate: Output is the filtered version of the cell state

- Decides the part of the cell we want as our output in the form of new hidden state
 - multiplicative output gate to protect other units from perturbation by currently irrelevant memory contents
 - a sigmoid layer decides what parts of the cell state goes to output. Apply tanh to the cell state and multiply it by the output of the sigmoid gate → only output the parts decided



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





Long Short Term Memory (LSTM) : Step by Step

As seen, the gradient vanishes due to the recurrent part of the RNN equations

$$h_t = W_{hh} h_{t-1} + \text{some other terms}$$

?

How LSTM tackled vanishing gradient?

?

Answer: forget gate

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

- The forget gate parameters takes care of the vanishing gradient problem
- Activation function becomes identity and therefore, the problem of vanishing gradient is addressed.
- The derivative of the identity function is, conveniently, always one. **So if $f = 1$** , information from the previous cell state can pass through this step unchanged



LSTMs: 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
- Now (2019–2024), 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

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>

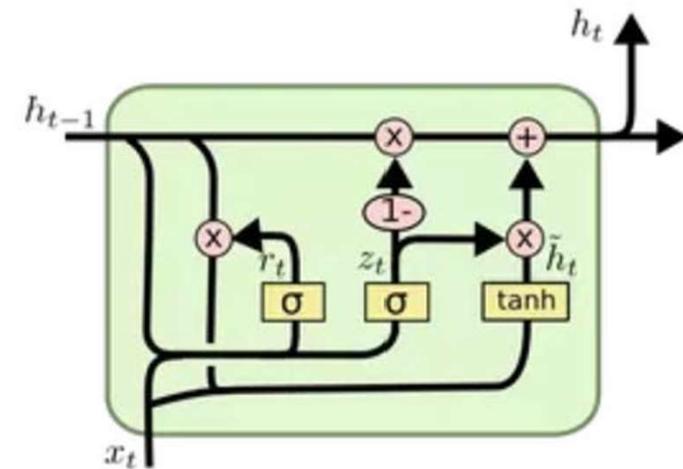
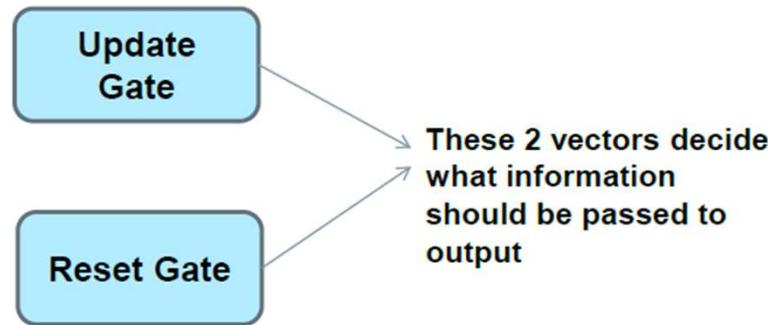
Source: "Findings of the 2019 Conference on Machine Translation (WMT19)", Barrault et al. 2019, <http://www.statmt.org/wmt18/pdf/WMT028.pdf>



Gated Recurrent Unit(GRU)

- GRU like LSTMs, attempts to solve the Vanishing gradient problem in RNN

Gates:



- Units with short-term dependencies will have active reset gates r
- Units with long term dependencies have active update gates z



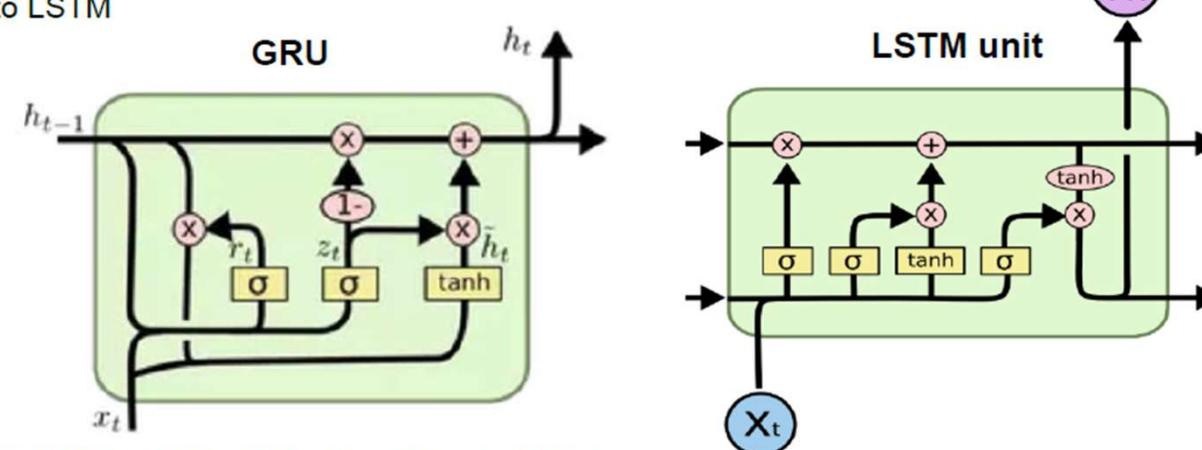
Gated Recurrent Unit(GRU)

LSTM over GRU

One feature of the LSTM has: **controlled exposure of the memory content**, not in GRU.

In the LSTM unit, the amount of the memory content that is seen, or used by other units in the network is controlled by the output gate. On the other hand the **GRU exposes its full content without any control**.

- GRU performs comparably to LSTM



Chung et al, 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling



Bi-directional RNNs

Bidirectional Recurrent Neural Networks (BRNN)

- connects two hidden layers of opposite directions to the same output
- output layer can get information from past (backwards) and future (forward) states simultaneously
- learn representations from future time steps to better understand the context and eliminate ambiguity

Example sentences:

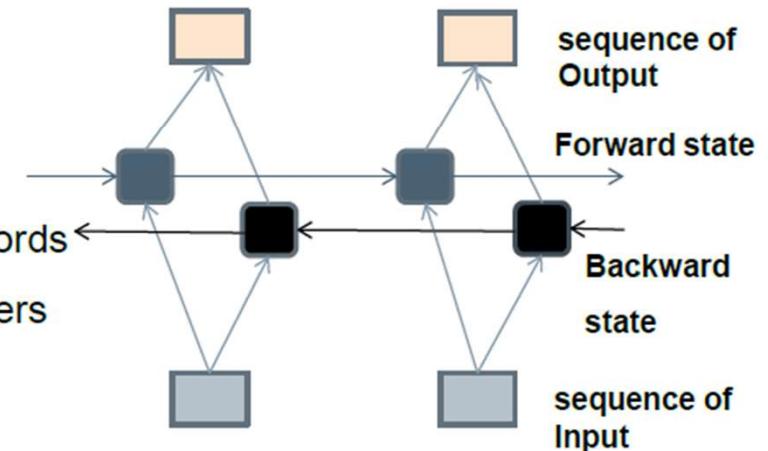
Sentence1: "He said, **Teddy** bears are on sale"

Sentnce2: "He said, **Teddy** Roosevelt was a great President".

when we are looking at the word "Teddy" and the previous two words "He said", we might not be able to understand if the sentence refers to the President or Teddy bears.

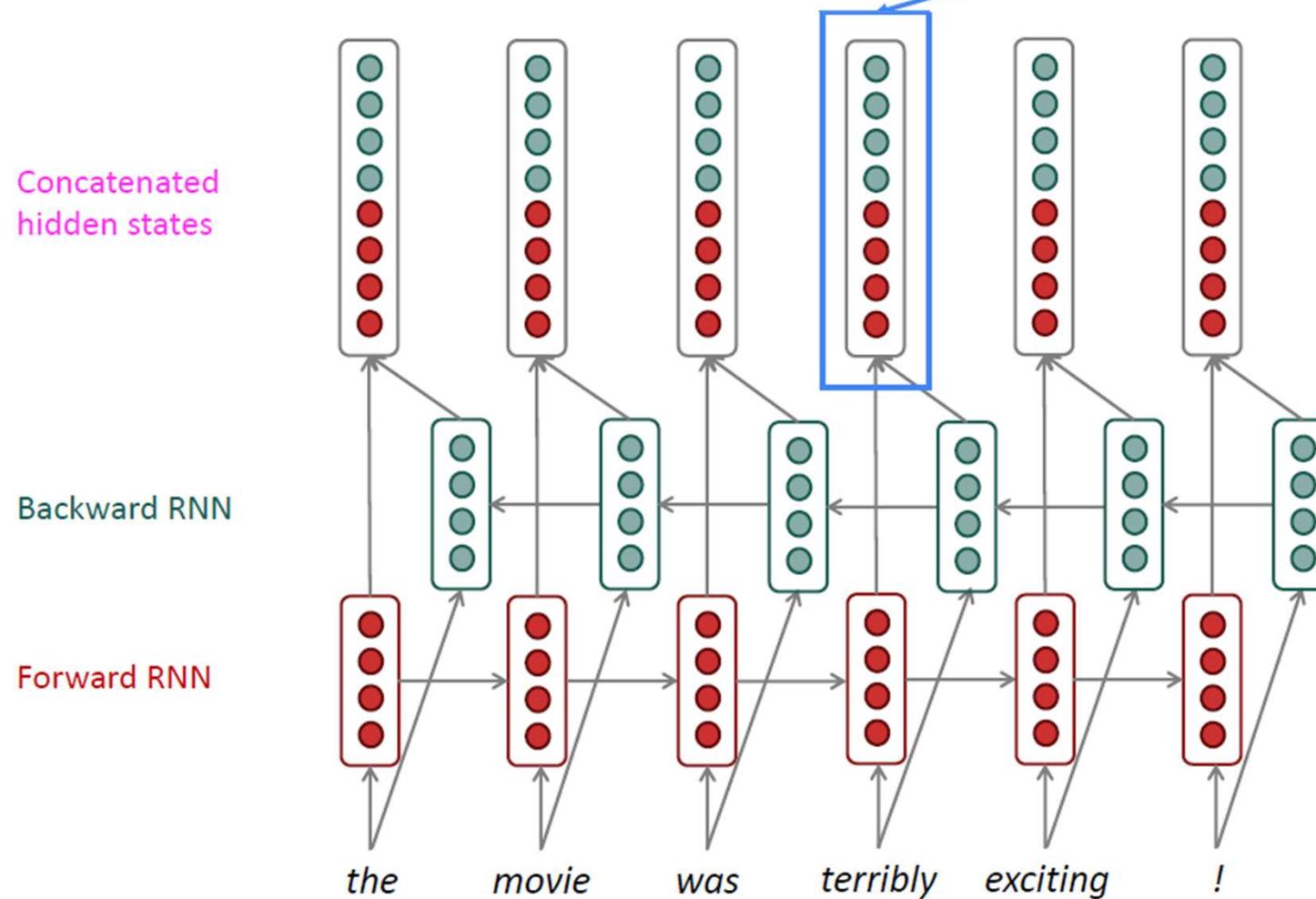
Therefore, to resolve this ambiguity, we need to look ahead.

<https://towardsdatascience.com/introduction-to-sequence-models-rnn-bidirectional-rnn-lstm-gru-73927ec9df15>





Bi-directional RNNs





Multi-layer RNN

This contextual representation of “terribly” has both left and right context!

- RNNs are already “deep” on one dimension (they unroll over many timesteps)
- We can also make them “deep” in another dimension by applying multiple RNNs – this is a multi-layer 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.
- Multi-layer RNNs are also called *stacked RNNs*.

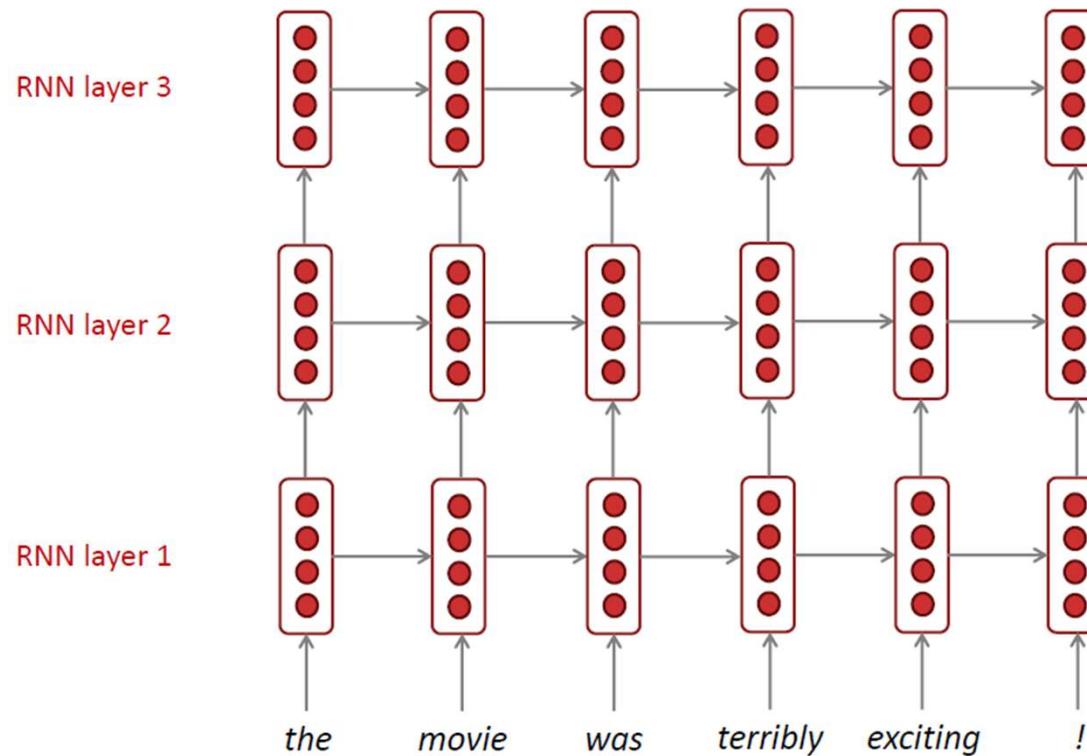




Multi-layer RNN

Multi-layer RNNs

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

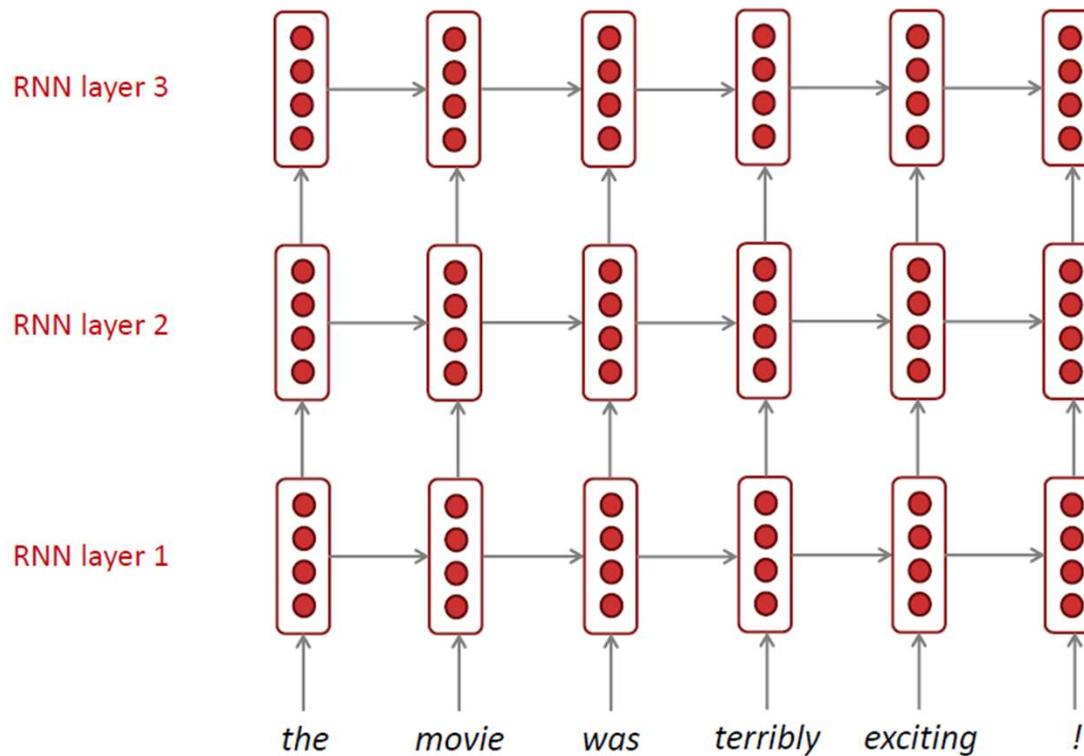




Attention is All you need

Multi-layer RNNs

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

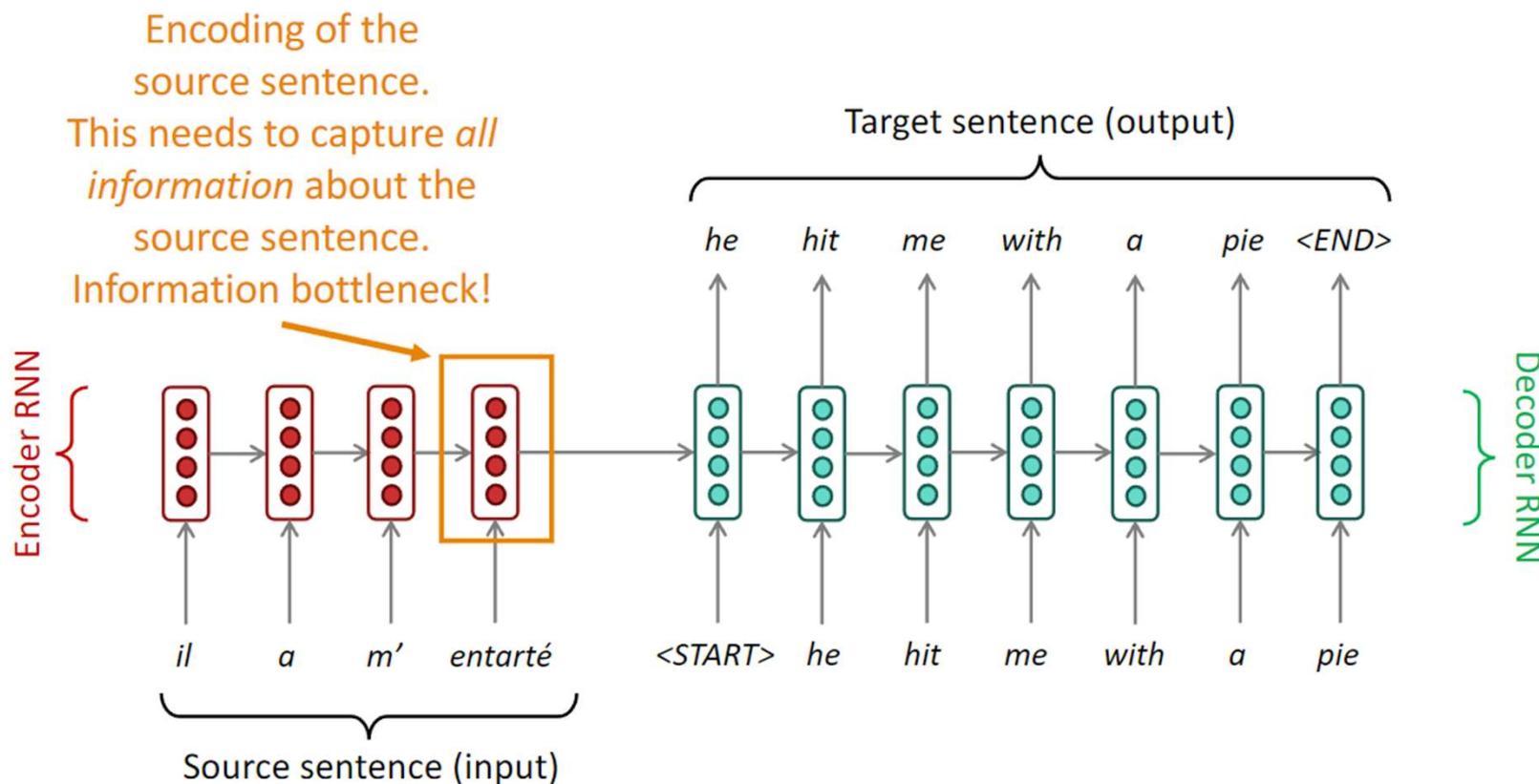


Attention & Transformers





Seq.-to-seq. model: the bottleneck problem





Seq.-to-seq. model: the bottleneck problem

- **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



- First, we will show via diagram (no equations), then we will show with equations



Seq.-to-seq. model: the bottleneck problem

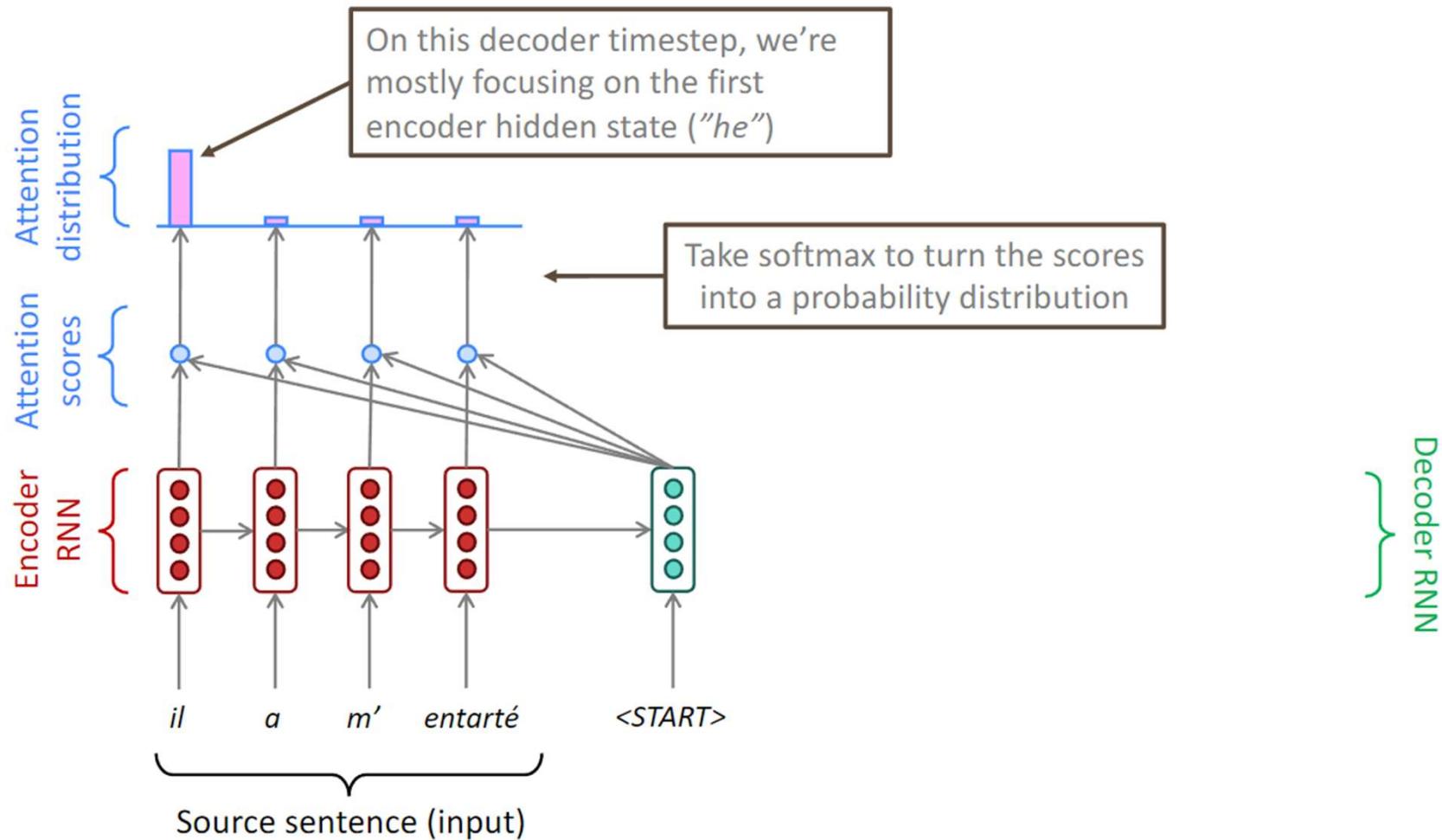
- **Attention** provides a solution to the bottleneck problem.
- **Core idea:** on each step of the decoder, *use direct connection to the encoder to focus on a particular part* of the source sequence



- First, we will show via diagram (no equations), then we will show with equations

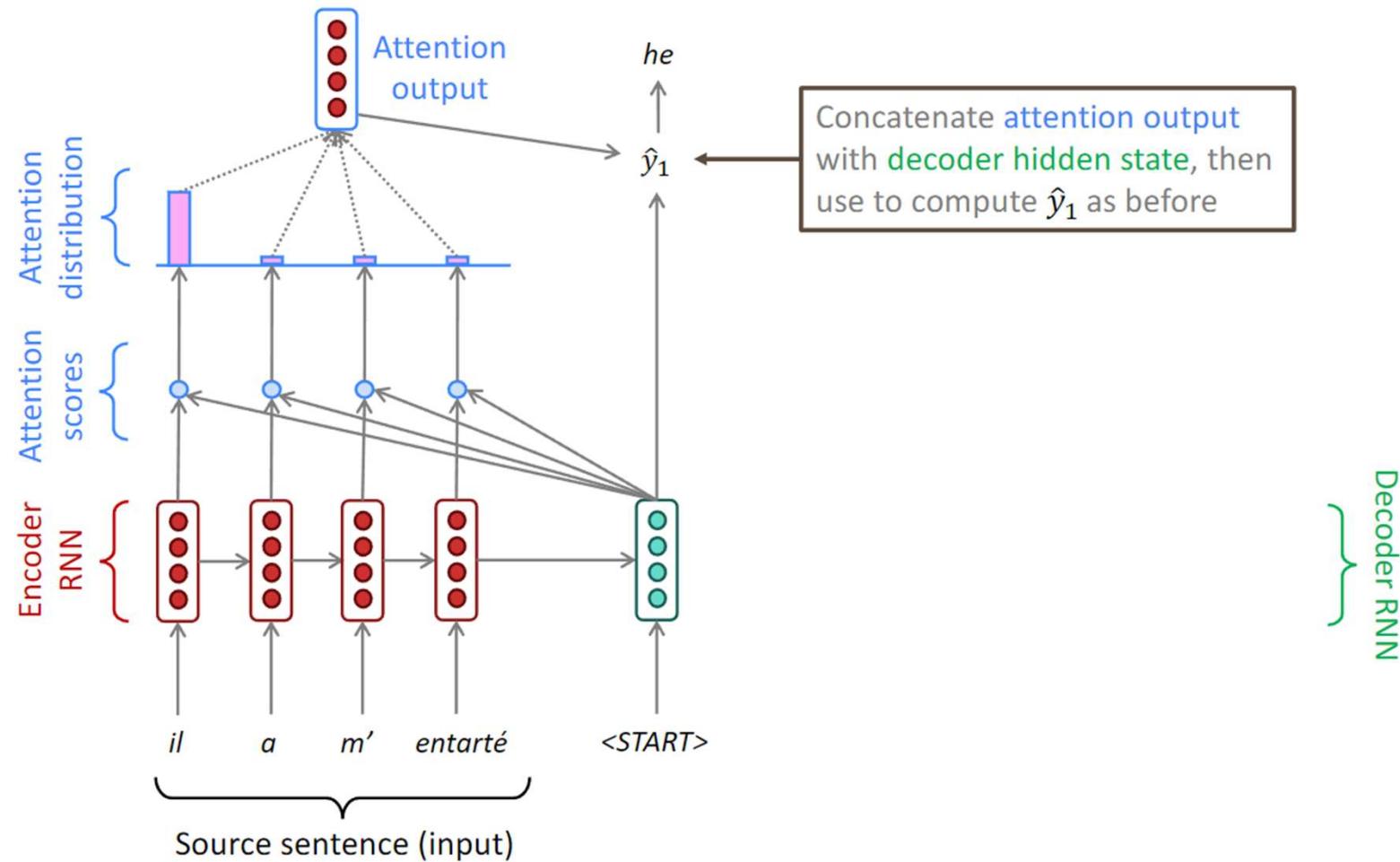


Seq.-to-seq. with attention



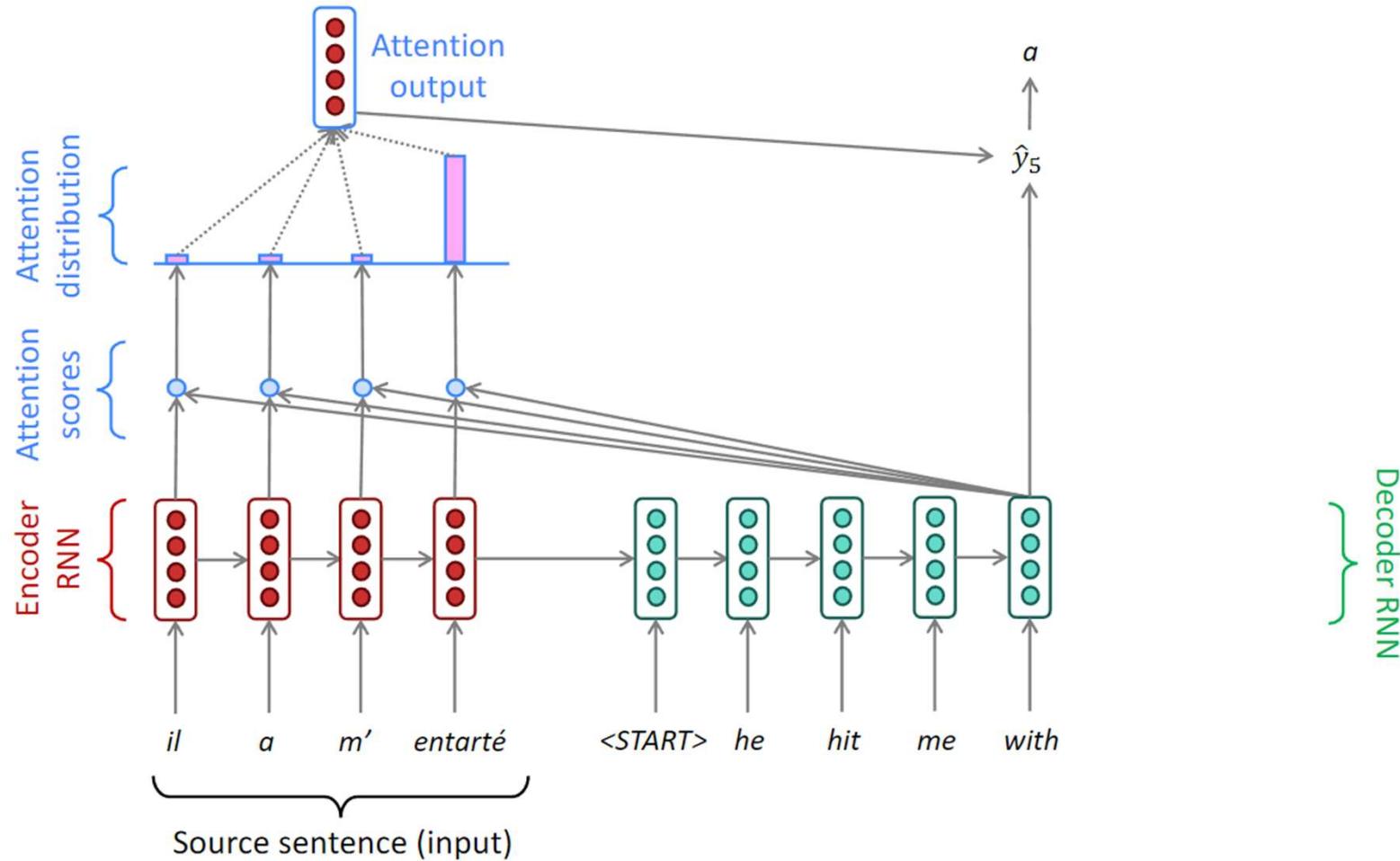


Seq.-to-seq. with attention





Seq.-to-seq. with attention





Seq.-to-seq. with attention

- We have encoder hidden states $h_1, \dots, h_N \in \mathbb{R}^h$
- On timestep t , we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores e^t for this step:

$$e^t = [s_t^T h_1, \dots, s_t^T h_N] \in \mathbb{R}^N$$

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

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

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output a_t

$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output a_t with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[a_t; s_t] \in \mathbb{R}^{2h}$$



Attention is great!

- Attention significantly improves NMT performance
 - It's very useful to allow decoder to focus on certain parts of the source
- Attention provides a more “human-like” model of the MT process
 - You can look back at the source sentence while translating, rather than needing to remember it all
- Attention solves the bottleneck problem
 - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
 - Provides shortcut to faraway states
- 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

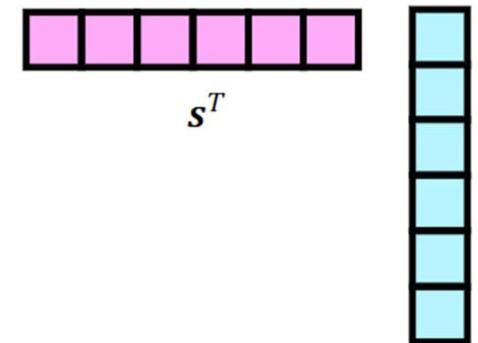
	he	hit	me	with	a	pie
il	█					
a		█				
m'			█			
entarté		█	█	█	█	█



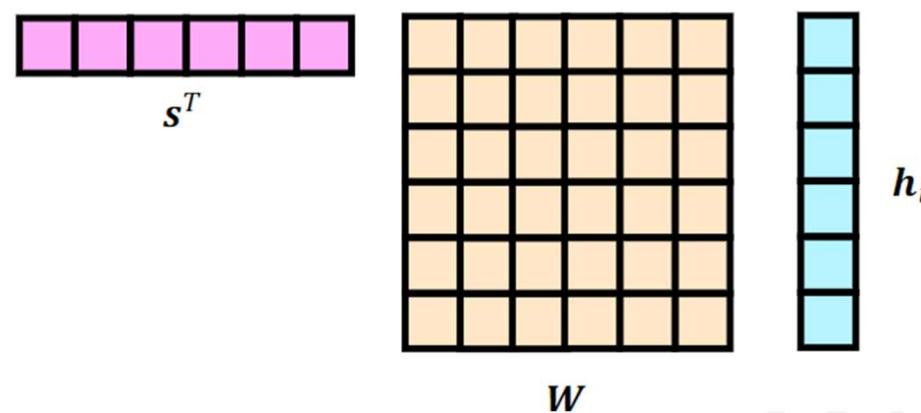
Attention Variants

There are **several ways** you can compute $e \in \mathbb{R}^N$ from $\mathbf{h}_1, \dots, \mathbf{h}_N \in \mathbb{R}^{d_1}$ and $\mathbf{s} \in \mathbb{R}^{d_2}$:

- Basic dot-product attention: $e_i = \mathbf{s}^T \mathbf{h}_i \in \mathbb{R}$
 - This assumes $d_1 = d_2$. This is the version we saw earlier.



- Multiplicative attention: $e_i = \mathbf{s}^T \mathbf{W} \mathbf{h}_i \in \mathbb{R}$ [Luong, Pham, and Manning 2015]
 - Where $\mathbf{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix. Perhaps better called “bilinear attention”





Attention is a general Deep Learning technique

- We've 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).



Attention is a general Deep Learning technique

- 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.

Intuition:

- 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).



Attention is a general Deep Learning technique

- 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.

Intuition:

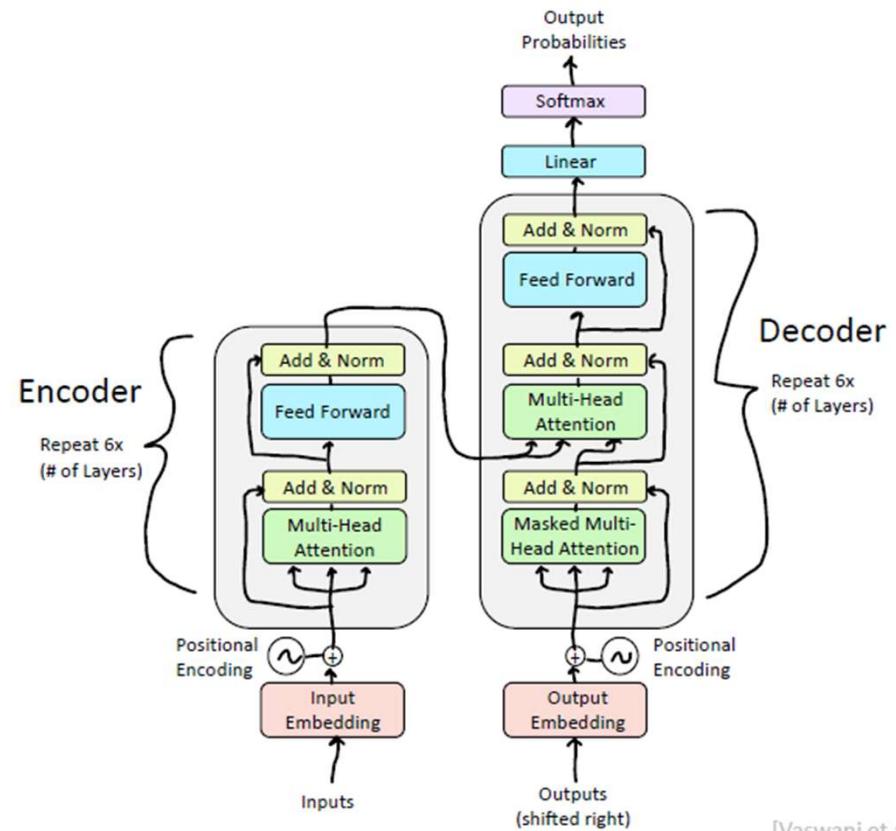
- The weighted sum is a *selective summary* of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a *fixed-size representation of an arbitrary set of representations* (the values), dependent on some other representation (the query).



Transformers have revolutionized AI fields



Courtesy of Paramount Pictures



[Vaswani et al., 2017]



Great results with Transformers: SuperGLUE

SuperGLUE is a suite of challenging NLP tasks, including question-answering, word sense disambiguation, coreference resolution, and natural language inference.

Rank	Name	Model	URL	Score	BoolQ	CB	COPA	MultiRC	ReCoRD	RTE	WiC	WSC	AX-b	AX-g	
1	JDExplore d-team	Vega v2		91.3	90.5	98.6/99.2	99.4	88.2/62.4	94.4/93.9	96.0	77.4	98.6	-0.4	100.0/50.0	
+	2	Liam Fedus	ST-MoE-32B		91.2	92.4	96.9/98.0	99.2	89.6/65.8	95.1/94.4	93.5	77.7	96.6	72.3	96.1/94.1
3	Microsoft Alexander v-team	Turing NLR v5		90.9	92.0	95.9/97.6	98.2	88.4/63.0	96.4/95.9	94.1	77.1	97.3	67.8	93.3/95.5	
4	ERNIE Team - Baidu	ERNIE 3.0		90.6	91.0	98.6/99.2	97.4	88.6/63.2	94.7/94.2	92.6	77.4	97.3	68.6	92.7/94.7	
5	Yi Tay	PaLM 540B		90.4	91.9	94.4/96.0	99.0	88.7/63.6	94.2/93.3	94.1	77.4	95.9	72.9	95.5/90.4	
+	6	Zirui Wang	T5 + UDG, Single Model (Google Brain)		90.4	91.4	95.8/97.6	98.0	88.3/63.0	94.2/93.5	93.0	77.9	96.6	69.1	92.7/91.9
+	7	DeBERTa Team - Microsoft	DeBERTa / TuringNLVR4		90.3	90.4	95.7/97.6	98.4	88.2/63.7	94.5/94.1	93.2	77.5	95.9	66.7	93.3/93.8
8	SuperGLUE Human Baselines				89.8	89.0	95.8/98.9	100.0	81.8/51.9	91.7/91.3	93.6	80.0	100.0	76.6	99.3/99.7
+	9	T5 Team - Google	T5		89.3	91.2	93.9/96.8	94.8	88.1/63.3	94.1/93.4	92.5	76.9	93.8	65.6	92.7/91.9
10	SPoT Team - Google	Frozen T5 1.1 + SPoT		89.2	91.1	95.8/97.6	95.6	87.9/61.9	93.3/92.4	92.9	75.8	93.8	66.9	83.1/82.6	

[Test sets: SuperGLUE Leaderboard Version: 2.0]

[Wang et al., 2019]



Great results with Transformers: SuperGLUE

Today, Transformer-based models dominate LMSYS Chatbot Arena Leaderboard!

Rank	Model	Arena Elo	95% CI	Votes	Organization	License	Knowledge Cutoff
1	GPT-4-Turbo-2024-04-09	1258	+4/-4	26444	OpenAI	Proprietary	2023/12
1	GPT-4-1106-preview	1253	+3/-3	68353	OpenAI	Proprietary	2023/4
1	Claude_3_Opus	1251	+3/-3	71500	Anthropic	Proprietary	2023/8
2	Gemini_1.5_Pro_API-0409-Preview	1249	+4/-5	22211	Google	Proprietary	2023/11
3	GPT-4-0125-preview	1248	+2/-3	58959	OpenAI	Proprietary	2023/12
6	Meta_Llama_3_70b_Instruct	1213	+4/-6	15809	Meta	Llama 3 Community	2023/12
6	Bard_(Gemini_Pro)	1208	+7/-6	12435	Google	Proprietary	Online
7	Claude_3_Sonnet	1201	+4/-2	73414	Anthropic	Proprietary	2023/8



Gemini/Bard
(Google)



ChatGPT/GPT-4
(OpenAI)



Claude3
(Anthropic AI)



Llama 3
(Meta)

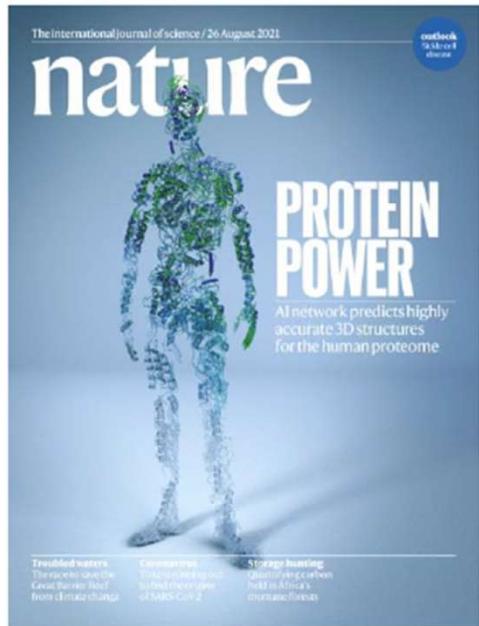


Qwen 2.5
(Alibaba Cloud)



Transformers are even promising outside of NLP

Protein Folding



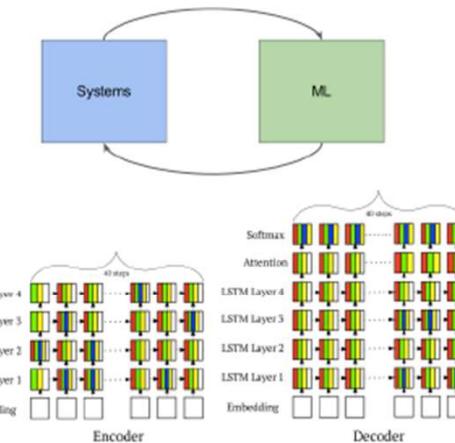
[Jumper et al. 2021] aka AlphaFold2!



Image Classification

[Dosovitskiy et al. 2020]: Vision Transformer (ViT) outperforms ResNet-based baselines with substantially less compute.

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	88.55 ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4 / 88.5*
ImageNet ReaL	90.72 ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	99.50 ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	—
CIFAR-100	94.55 ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	—
Oxford-IIIT Pets	97.56 ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	—
Oxford Flowers-102	99.68 ± 0.02	99.74 ± 0.00	99.61 ± 0.02	99.63 ± 0.03	—
VTAB (19 tasks)	77.63 ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	—
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k



ML for Systems

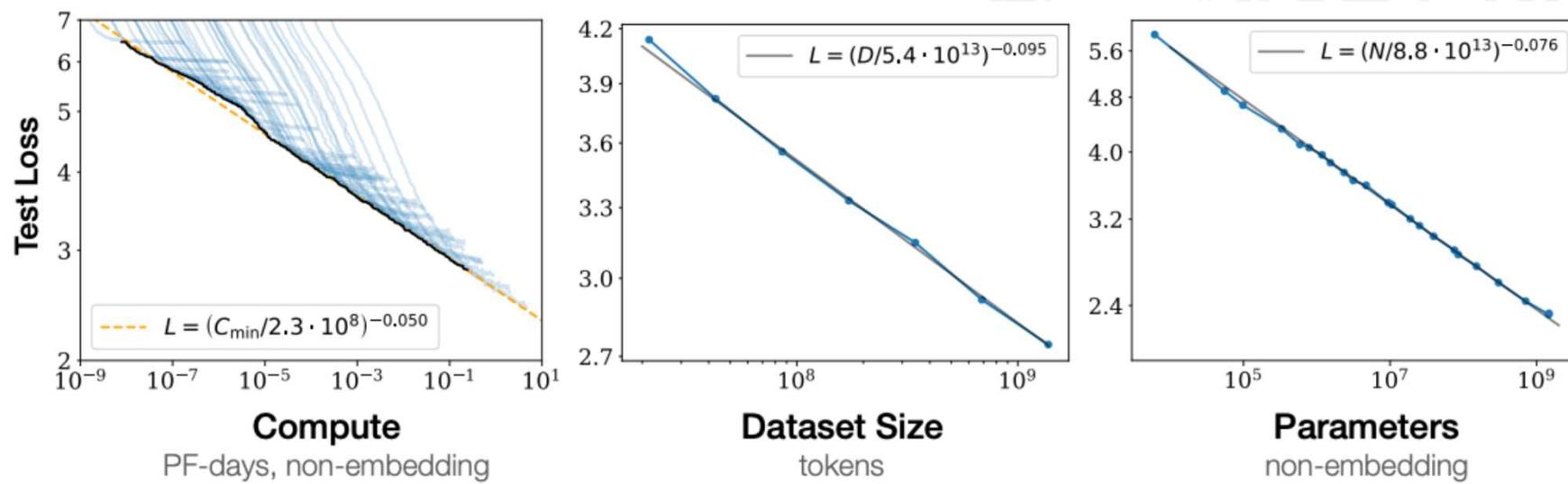
[Zhou et al. 2020]: A Transformer-based compiler model (GO-one) speeds up a Transformer model!

Model (Devices)	GO-one (s)	HP (s)	METIS (s)	HDP (s)	Run time speed up over HP / HDP	Search speed up over HDP
2-layer RNNLM (2)	0.173	0.192	0.355	0.191	9.9% / 9.4%	2.95x
4-layer RNNLM (4)	0.210	0.239	0.503	0.251	13.8% / 16.3%	1.35x
8-layer RNNLM (8)	0.320	0.332	0.607	0.764	3.8% / 58.1%	27.8x
2-layer GNNM (2)	0.191	0.241	0.344	0.27	10.7% / 10.5%	30.5x
4-layer GNNM (4)	0.350	0.469	0.66	0.432	34% / 35.4%	58.8x
8-layer GNNM (8)	0.440	0.662	0.693	0.693	21.7% / 36.5%	73.5x
2-layer Transformer-XL (2)	0.223	0.268	0.37	0.263	20.1% / 17.4%	40x
4-layer Transformer-XL (4)	0.230	0.27	0.406	0.259	17.4% / 12.6%	26.7x
8-layer Transformer-XL (8)	0.350	0.46	0.606	0.425	23.9% / 16.7%	16.7x
Inception-v4	0.229	0.312	0.406	0.301	26.6% / 23.9%	13.5x
Inception-2D (2)	0.423	0.731	0.606	0.498	42.1% / 29.3%	21.0x
ResNeXt-Net (4)	0.394	0.44	0.426	0.418	24.1% / 6.1%	58.8x
2-stack 18-layer WaveNet (2)	0.317	0.376	0.406	0.354	18.6% / 11.7%	6.67x
4-stack 36-layer WaveNet (4)	0.659	0.988	0.606	0.721	50% / 9.4%	20x
GEO-MEAN	-	-	-	-	26.5% / 18.2%	15x



Scaling laws: Are Transformers All we need?

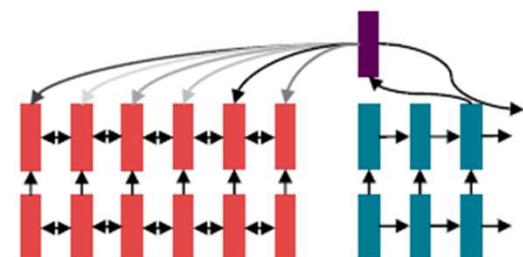
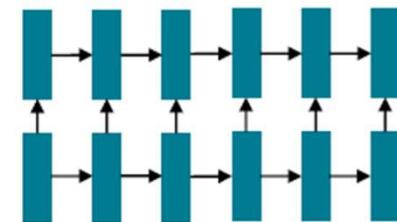
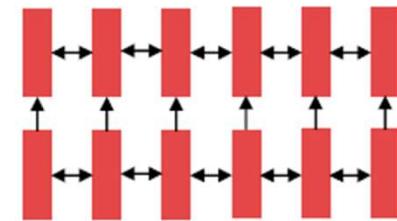
- With Transformers, language modeling performance improves smoothly as we increase model size, training data, and compute resources in tandem.
- This power-law relationship has been observed over multiple orders of magnitude with no sign of slowing!
- If we keep scaling up these models (with no change to the architecture), could they eventually match or exceed human-level performance?





RNN for (most) NLP

- Circa 2016, the de facto strategy in NLP is to **encode** sentences with a bidirectional LSTM:
(for example, the source sentence in a translation)
- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.
- Use attention to allow flexible access to memory





Why Move Beyond Recurrence? Motivation for Transformer Architecture

The Transformers authors had 3 desirata when designing this architecture:

1. Minimize (or at least not increase) computational complexity per layer.
2. Minimize path length between any pair of words to facilitate learning of long-range dependencies.
3. Maximize the amount of computation that can be parallelized.



1. Transformer Motivation: Computational Complexity Per Layer

When sequence length (n) \ll representation dimension (d), complexity per layer is lower for a Transformer compared to the recurrent models we've learned about so far.

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

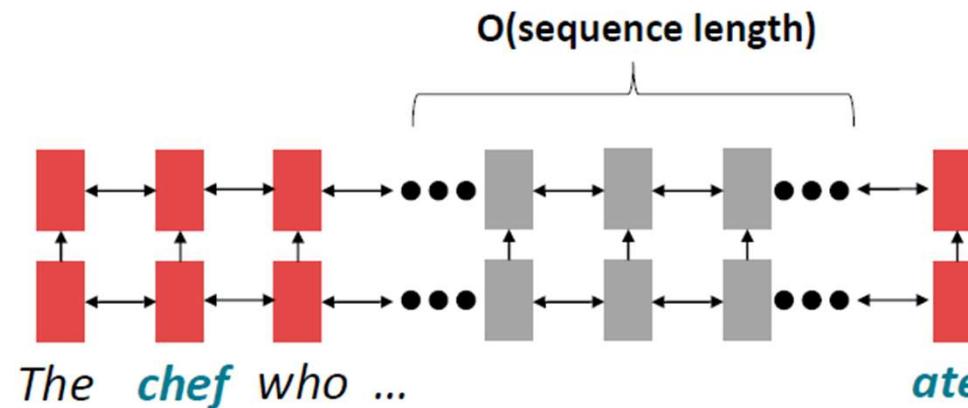
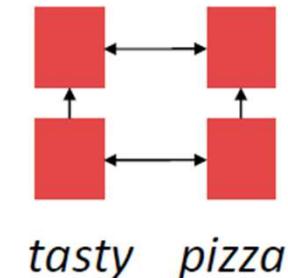
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1 of the Transformer paper.



2. Transformer Motivation: Minimize Linear Interaction Distance

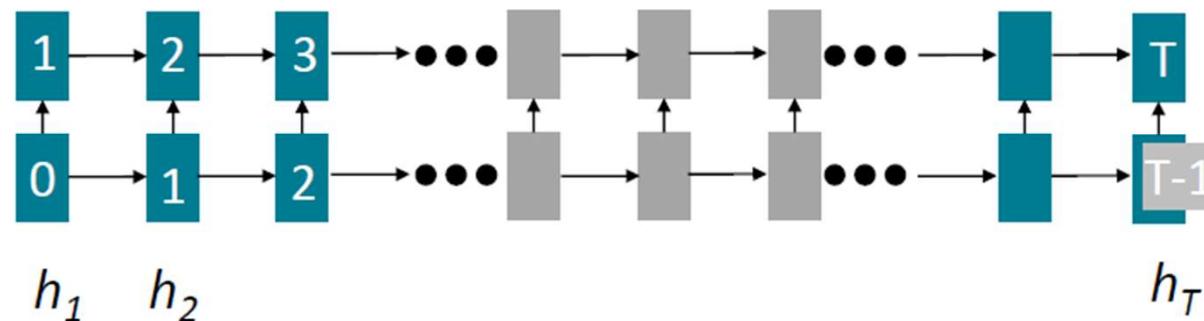
- RNNs are unrolled “left-to-right”.
- It encodes linear locality: a useful heuristic!
 - Nearby words often affect each other’s meanings
- **Problem:** RNNs take **O(sequence length)** steps for distant word pairs to interact.





3. Transformer Motivation: Maximize Parallelizability

- Forward and backward passes have **O(seq length)** unparallelizable operations
 - GPUs (and TPUs) can perform many independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
 - Inhibits training on very large datasets!
 - Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations

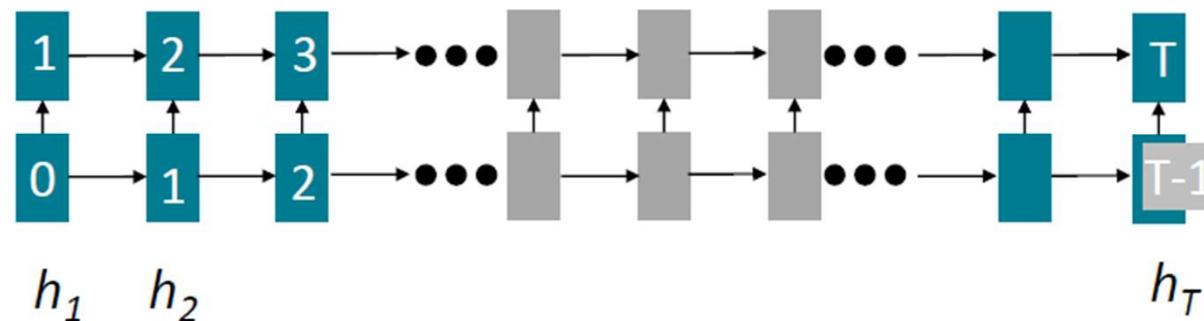


Numbers indicate min # of steps before a state can be computed



3. Transformer Motivation: Maximize Parallelizability

- Forward and backward passes have **O(seq length)** unparallelizable operations
 - GPUs (and TPUs) can perform many independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
 - Inhibits training on very large datasets!
 - Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations

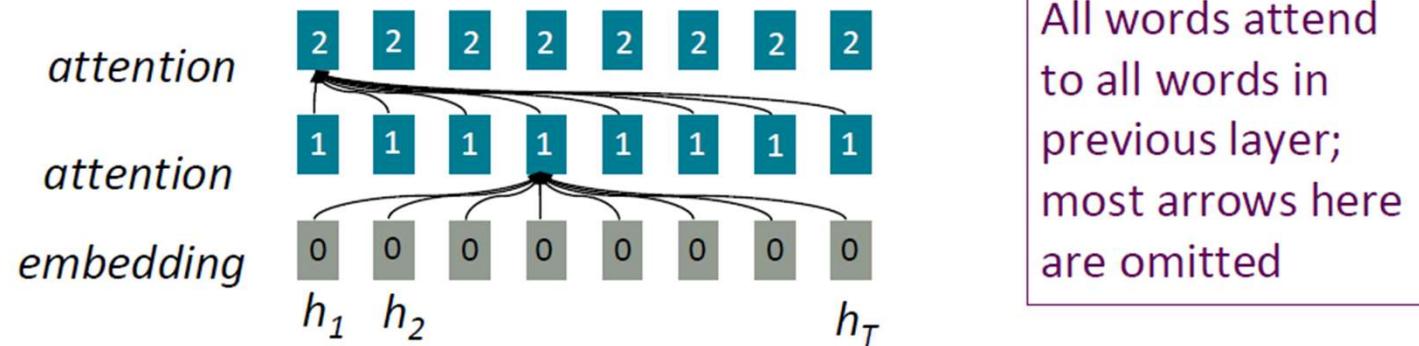


Numbers indicate min # of steps before a state can be computed



High-level Architecture: transformer is all about (self) attention

- To recap, **attention** treats each word's representation as a **query** to access and incorporate information from **a set of values**.
 - Last lecture, we saw attention from the **decoder** to the **encoder** in a recurrent sequence-to-sequence model
 - Self-attention** is **encoder-encoder** (or **decoder-decoder**) attention where each word attends to each other word **within the input (or output)**.

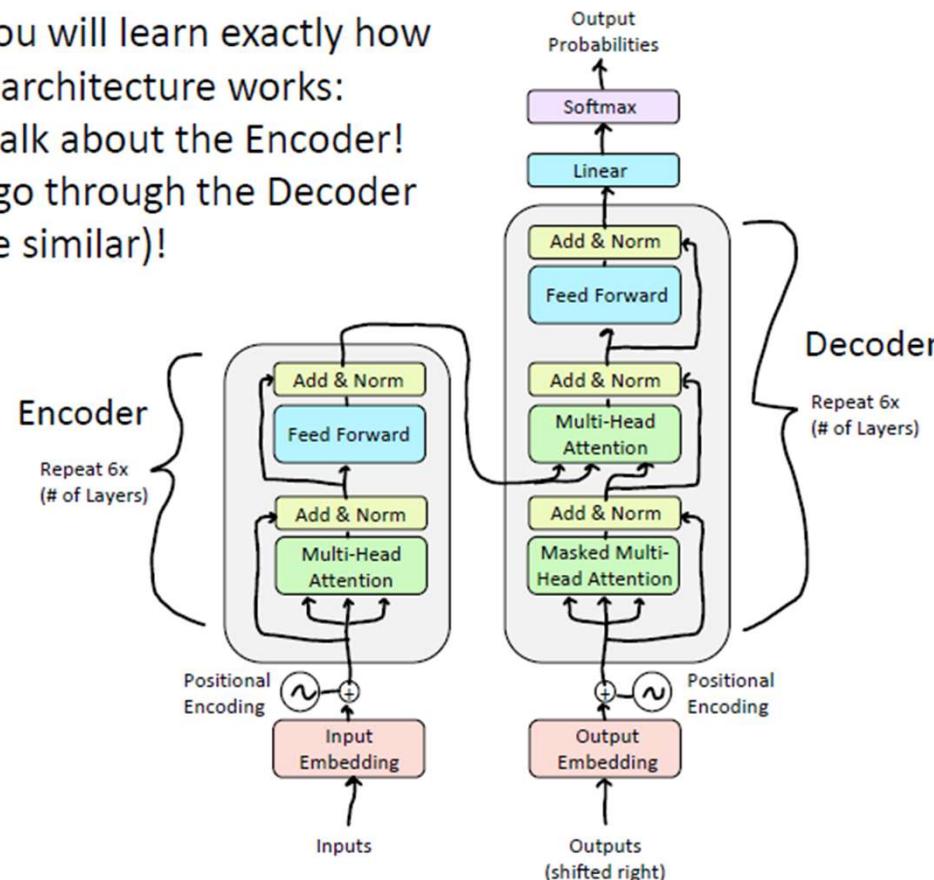




The Transformer Encoder-Decoder [VASWANI et al., 2017]

In this section, you will learn exactly how the Transformer architecture works:

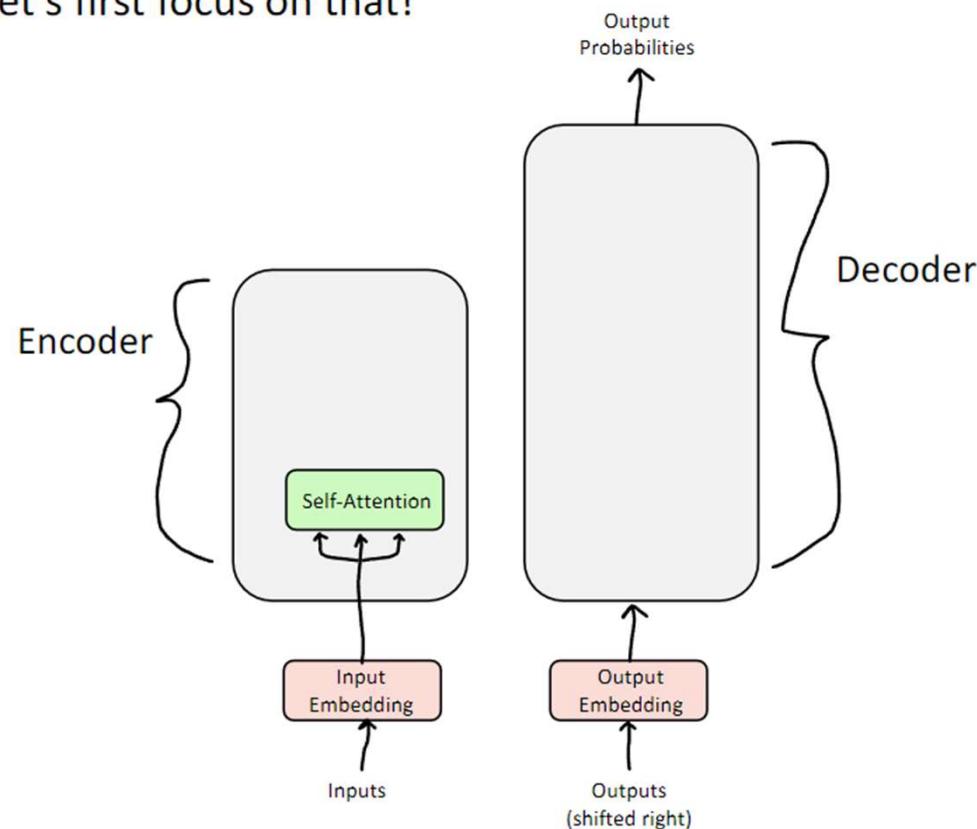
- First, we will talk about the Encoder!
- Next, we will go through the Decoder (which is quite similar)!





Encoder: Self-Attention

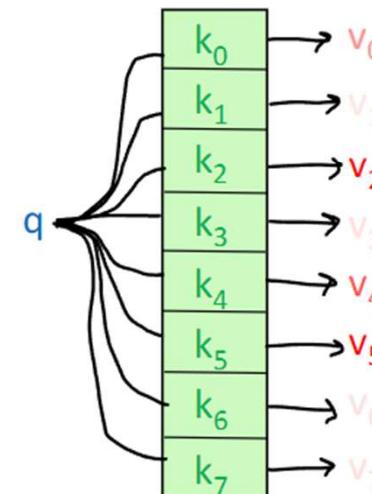
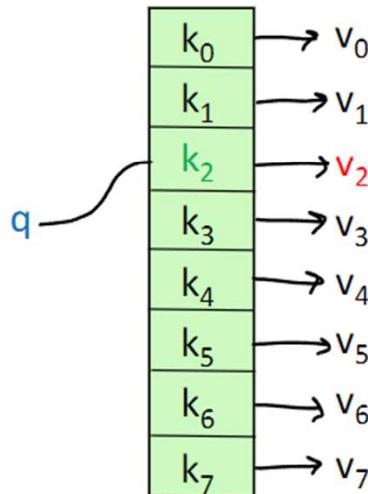
Self-Attention is the core building block of Transformer, so let's first focus on that!





Encoder: Self-Attention

- Let's think of attention as a "fuzzy" or approximate hashtable:
 - To look up a **value**, we compare a **query** against **keys** in a table.
 - In a hashtable (shown on the bottom left):
 - Each **query** (hash) maps to exactly one **key-value** pair.
 - In (self-)attention (shown on the bottom right):
 - Each **query** matches each **key** to varying degrees.
 - We return a sum of **values** weighted by the **query-key** match.





Recipe for Self-Attention in the Transformer Encoder

- Step 1: For each word x_i , calculate its **query**, **key**, and **value**.

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query** and **keys**.

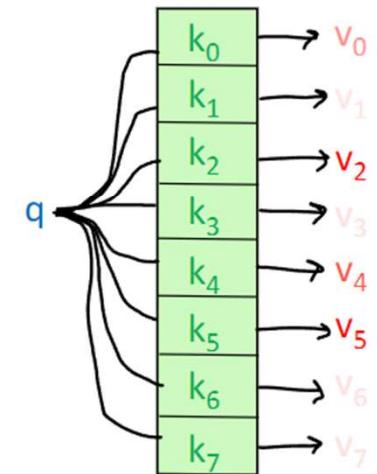
$$e_{ij} = q_i \cdot k_j$$

- Step 3: Take the softmax to normalize attention scores.

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**.

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$





Recipe for Self-Attention in the Transformer Encoder

- Step 1: With embeddings stacked in X , calculate **queries**, **keys**, and **values**.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

- Step 2: Calculate attention scores between **query** and **keys**.

$$E = QK^T$$

- Step 3: Take the softmax to normalize attention scores.

$$A = \text{softmax}(E)$$

- Step 4: Take a weighted sum of **values**.

$$\text{Output} = AV$$

$$\text{Output} = \text{softmax}(QK^T)V$$

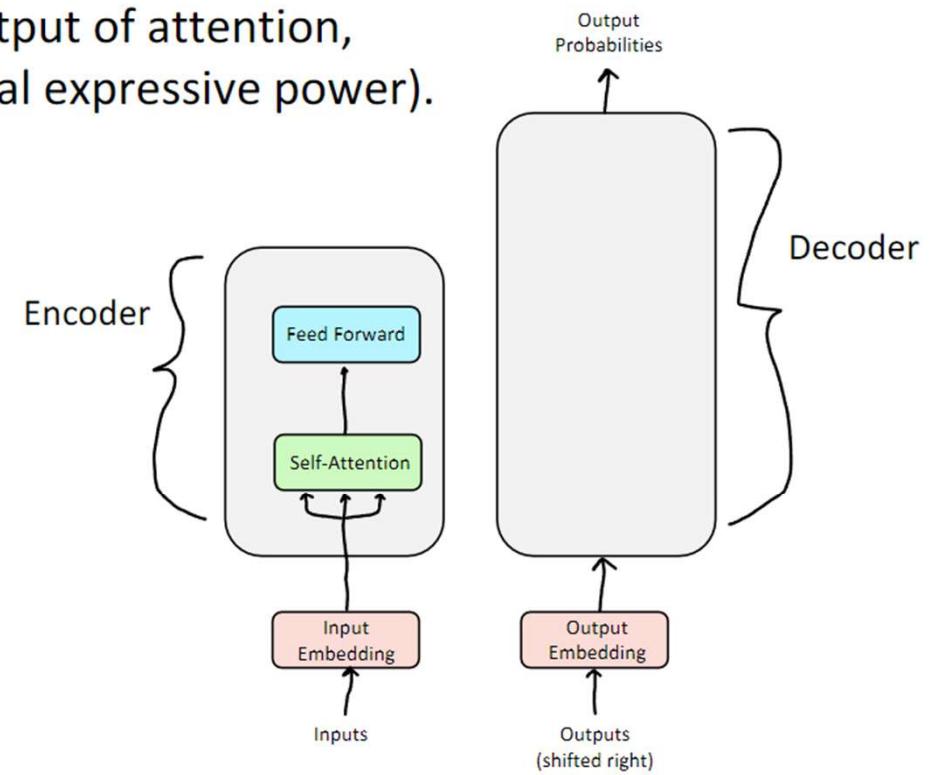
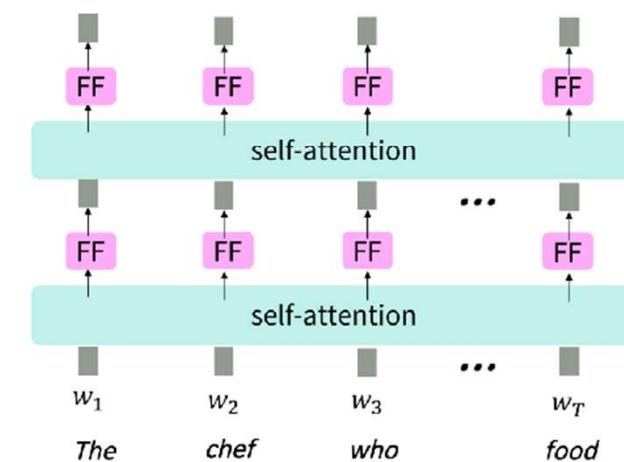


But attention isn't quite all you need!

- **Problem:** Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.
- **Easy fix:** Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

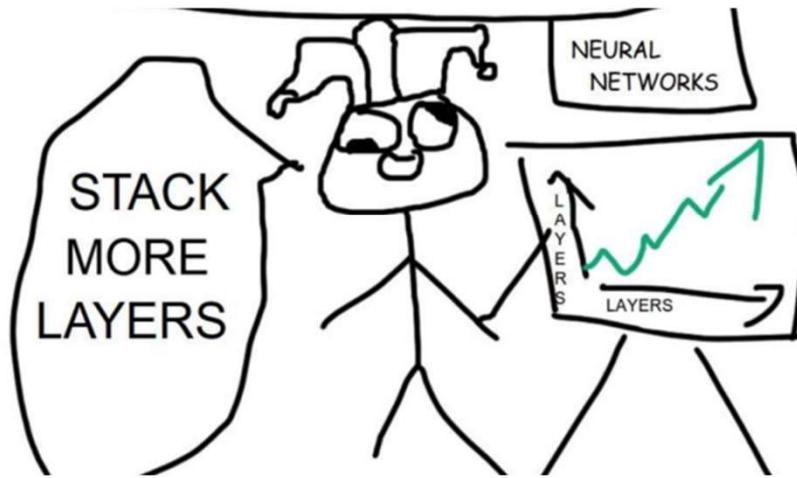
Equation for Feed Forward Layer

$$\begin{aligned} m_i &= \text{MLP}(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2 \end{aligned}$$





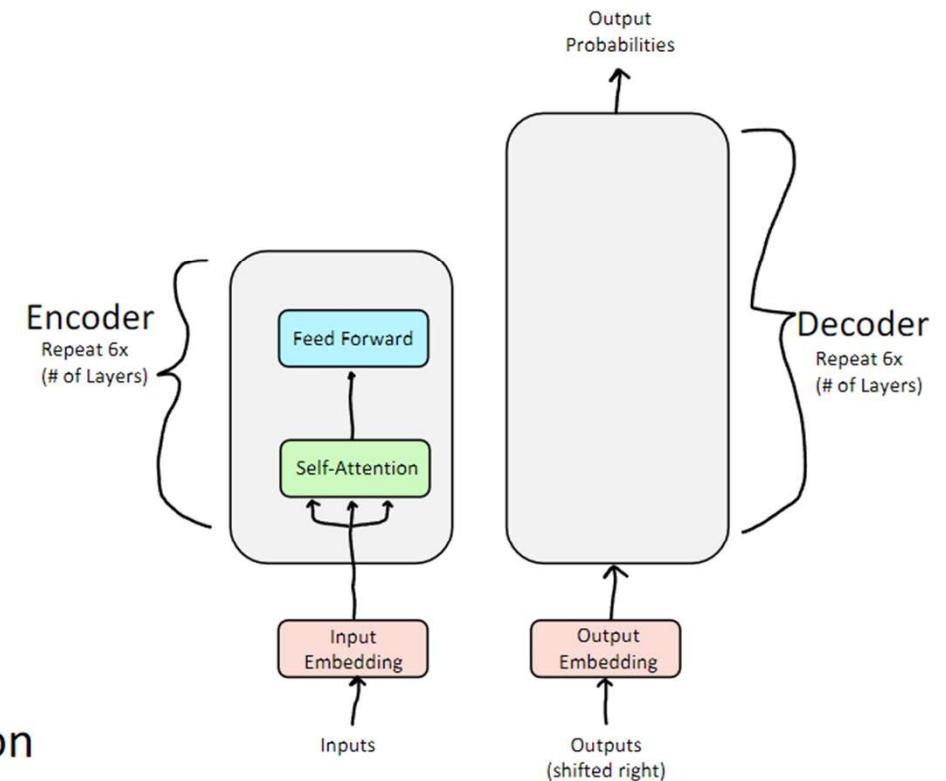
But how do we make this work for deep networks?



Training Trick #1: Residual Connections

Training Trick #2: LayerNorm

Training Trick #3: Scaled Dot Product Attention

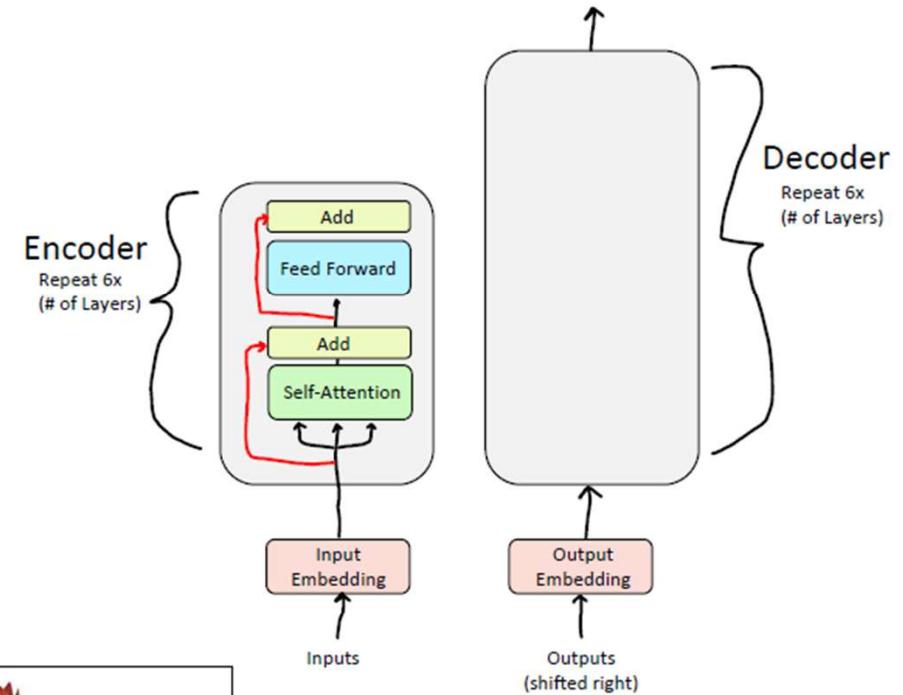
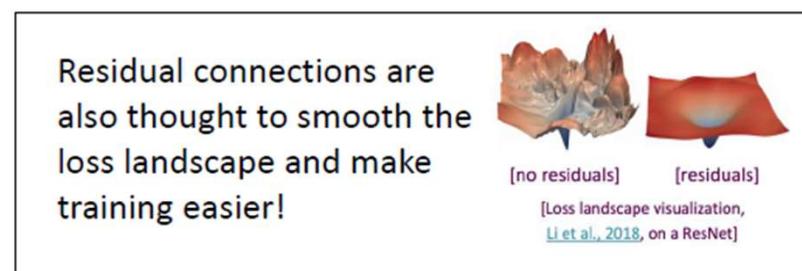




Training Trick #1: Residual Connections

[He et al., 2016]

- Residual connections are a simple but powerful technique from computer vision.
- Deep networks are surprisingly bad at learning the identity function!
- Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!
$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$
- This prevents the network from "forgetting" or distorting important information as it is processed by many layers.

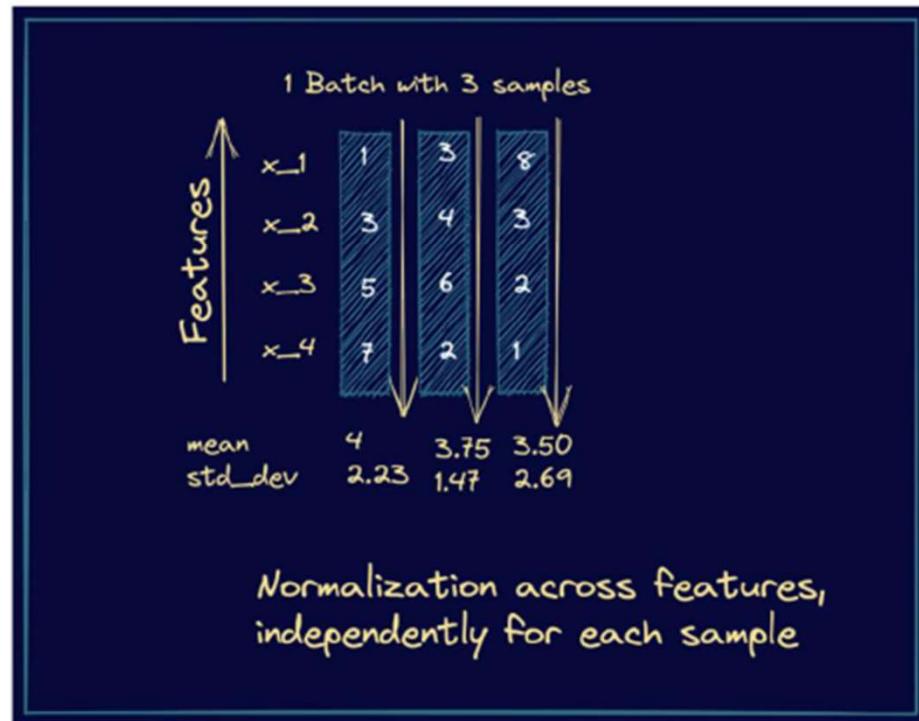




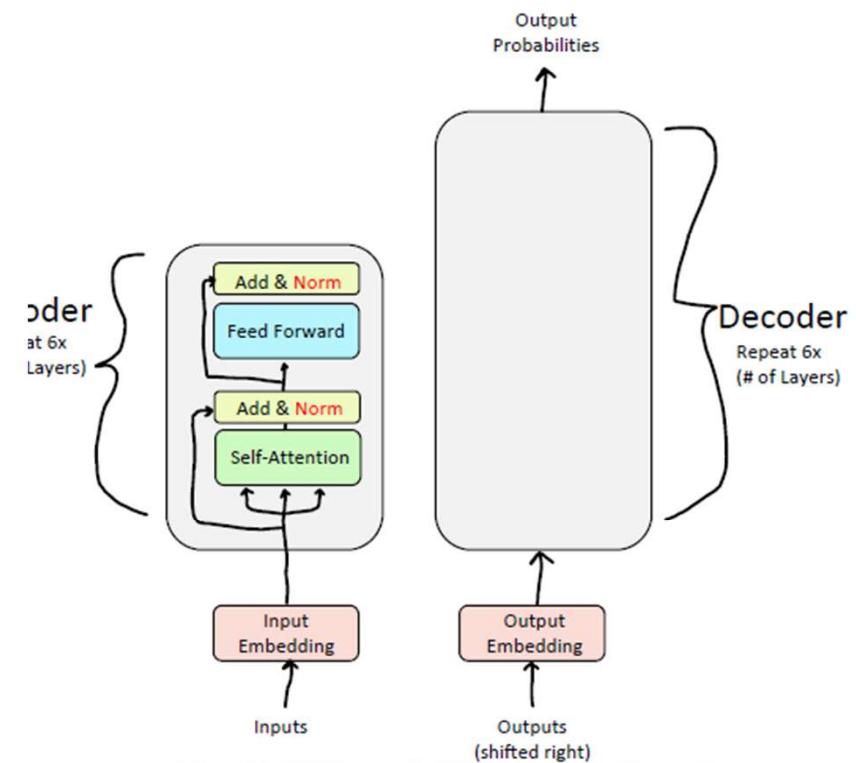
Training Trick #2: Layer Normalization

[Ba et al., 2016]

-
-



An Example of How LayerNorm Works (Image by Bala Priya C, Pinecone)





Training Trick #3: Scaled Dot Product Attention

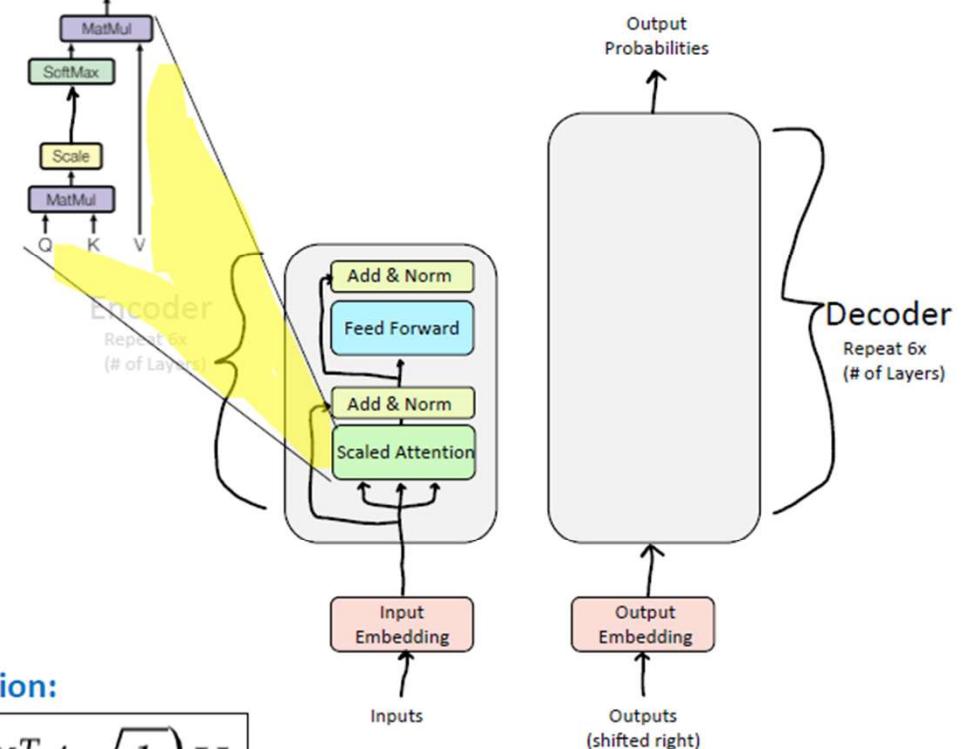
- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k

Quick Statistics Review:

- Mean of sum = sum of means = $d_k * 0 = 0$
- Variance of sum = sum of variances = $d_k * 1 = d_k$
- To set the variance to 1, simply divide by $\sqrt{d_k}$!

Updated Self-Attention Equation:

$$\text{Output} = \text{softmax}\left(QK^T / \sqrt{d_k}\right)V$$

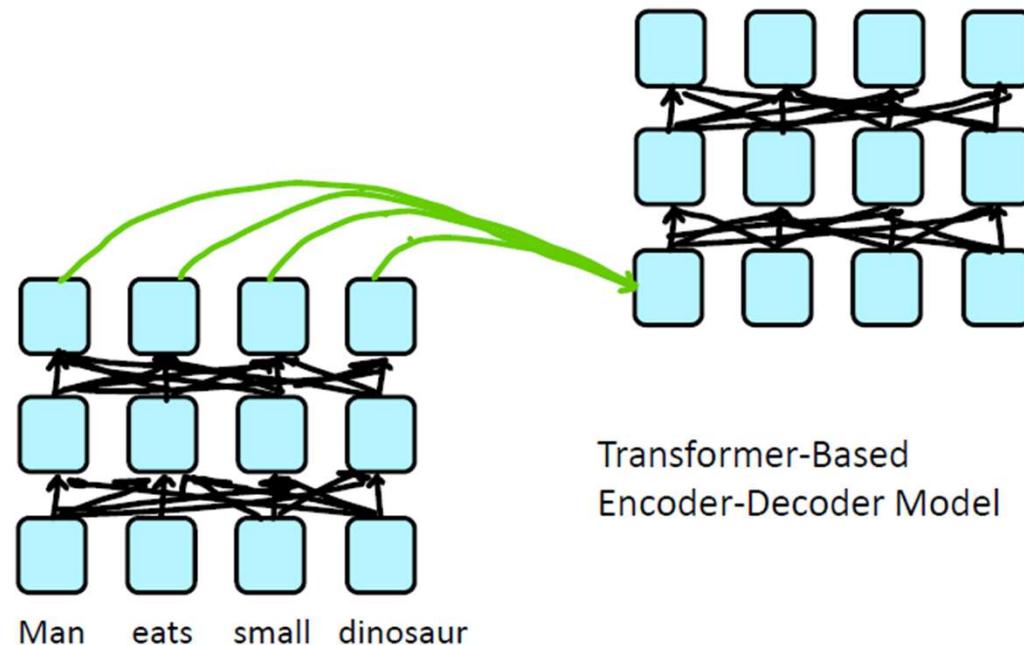




Major Issue!

- We're almost done with the Encoder, but we have a major problem! Has anyone spotted it?
- Consider this sentence:
 - "Man eats small dinosaur."
- Wait a minute, order doesn't impact the network at all!
- This seems wrong given that word order does have meaning in many languages, including English!

$$\text{Output} = \text{softmax}\left(QK^T / \sqrt{d_k}\right)V$$





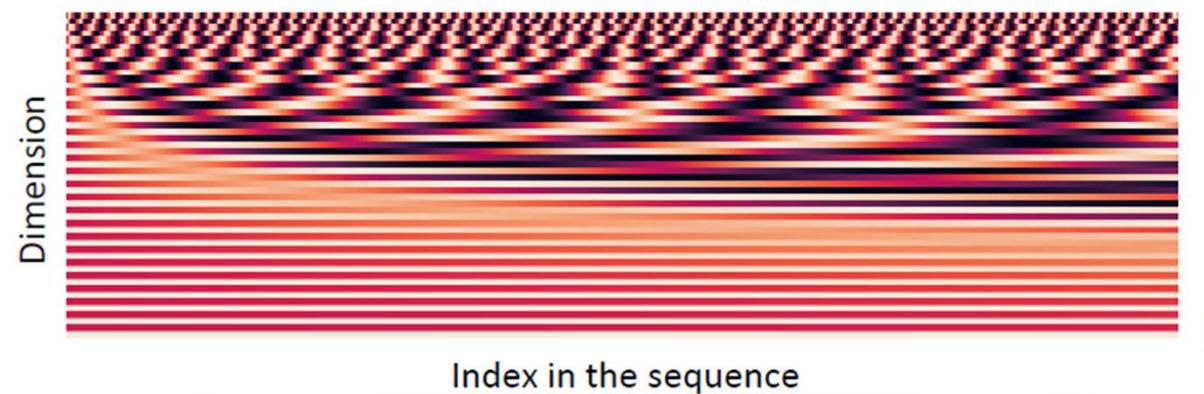
Fixing the first self-attention problem: sequence order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$p_i \in \mathbb{R}^d$, for $i \in \{1, 2, \dots, T\}$ are position vectors

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Let $\tilde{v}_i, \tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.
- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

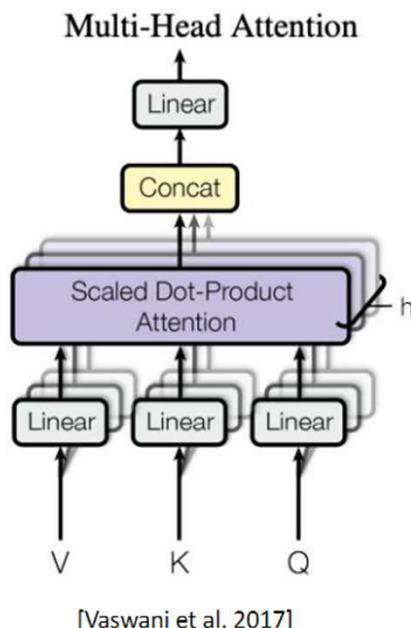
$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$





Multi-head self-attention: k heads are better than 1!

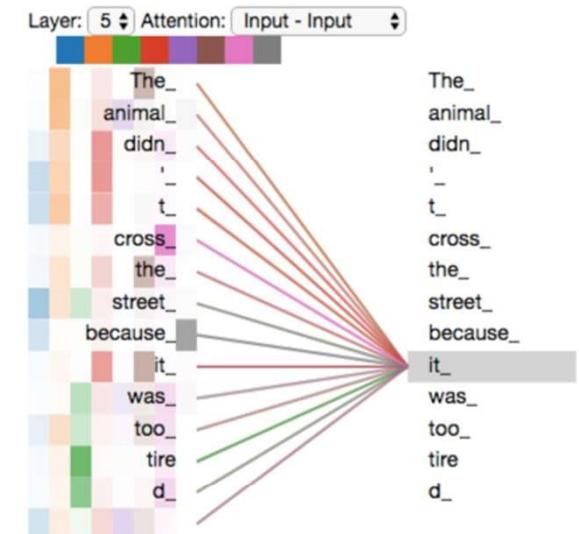
- **High-Level Idea:** Let's perform self-attention multiple times in parallel and combine the results.





Multi-head self-attention: k heads are better than 1!

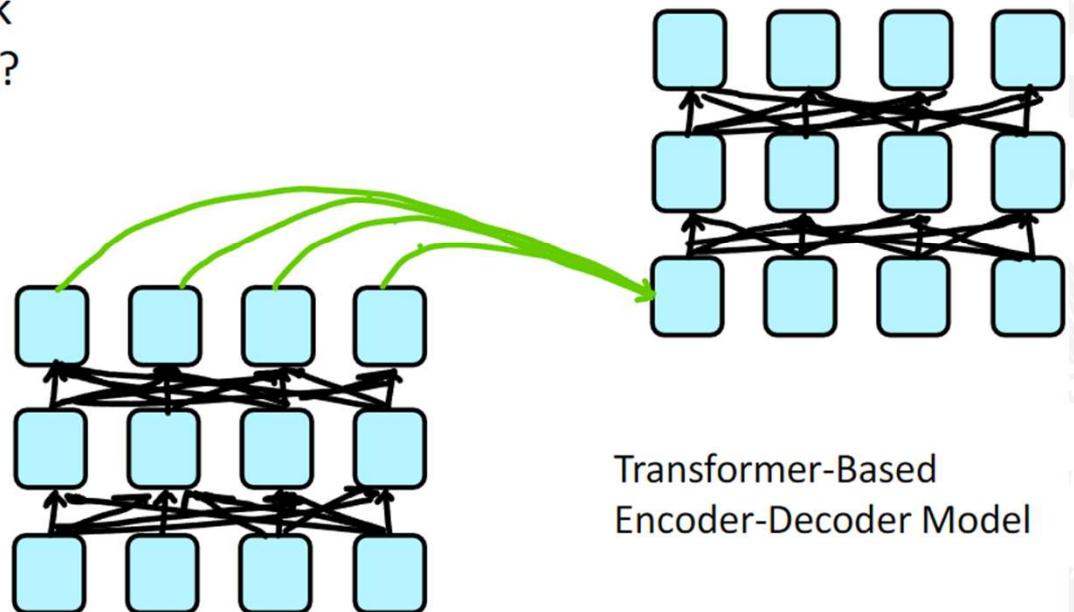
- What if we want to look in multiple places in the sentence at once?
 - For word i , self-attention “looks” where $x_i^T Q^T K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We’ll define **multiple attention “heads”** through multiple Q,K,V matrices
- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h .
- Each attention head performs attention independently:
 - $\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^T X^T) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
 - $\text{output} = Y[\text{output}_1; \dots; \text{output}_h]$, where $Y \in \mathbb{R}^{d \times d}$
- Each head gets to “look” at different things, and construct value vectors differently.





Decoder: Masked Multi-Head Self-Attention

- **Problem:** How do we keep the decoder from “cheating”? If we have a language modeling objective, can't the network just look ahead and “see” the answer?
- **Solution:** Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.



Transformer-Based
Encoder-Decoder Model

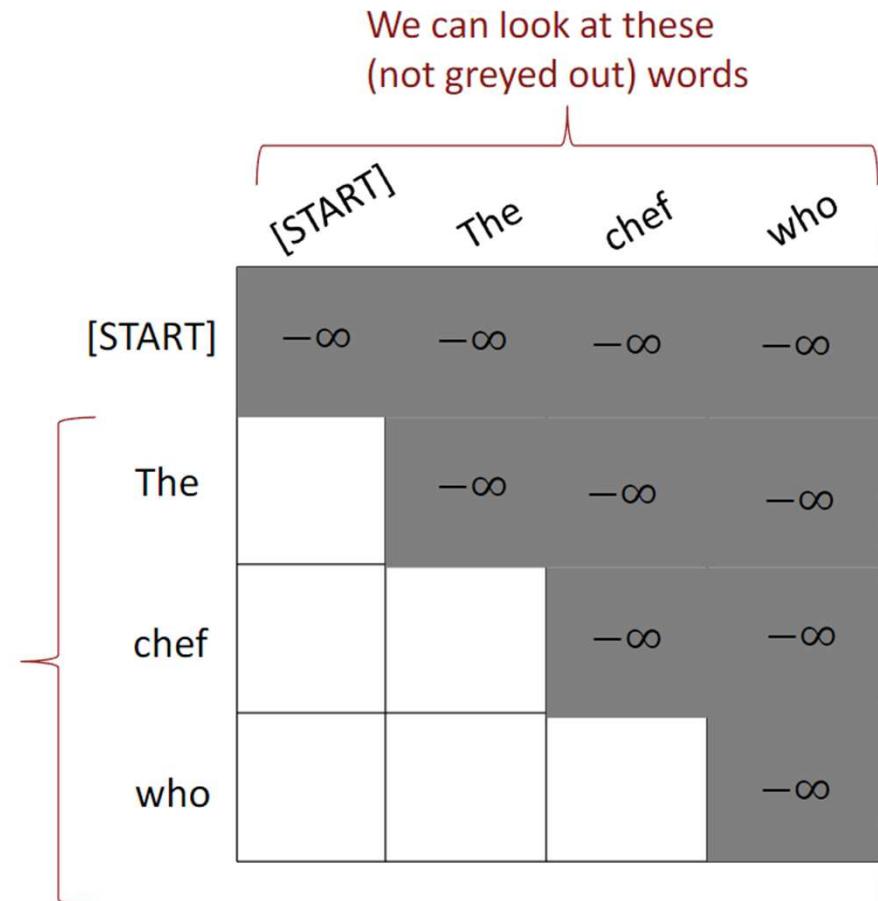


Decoder: Masked Multi-Head Self-Attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^T k_j, & j < i \\ -\infty, & j \geq i \end{cases}$$

For encoding
these words



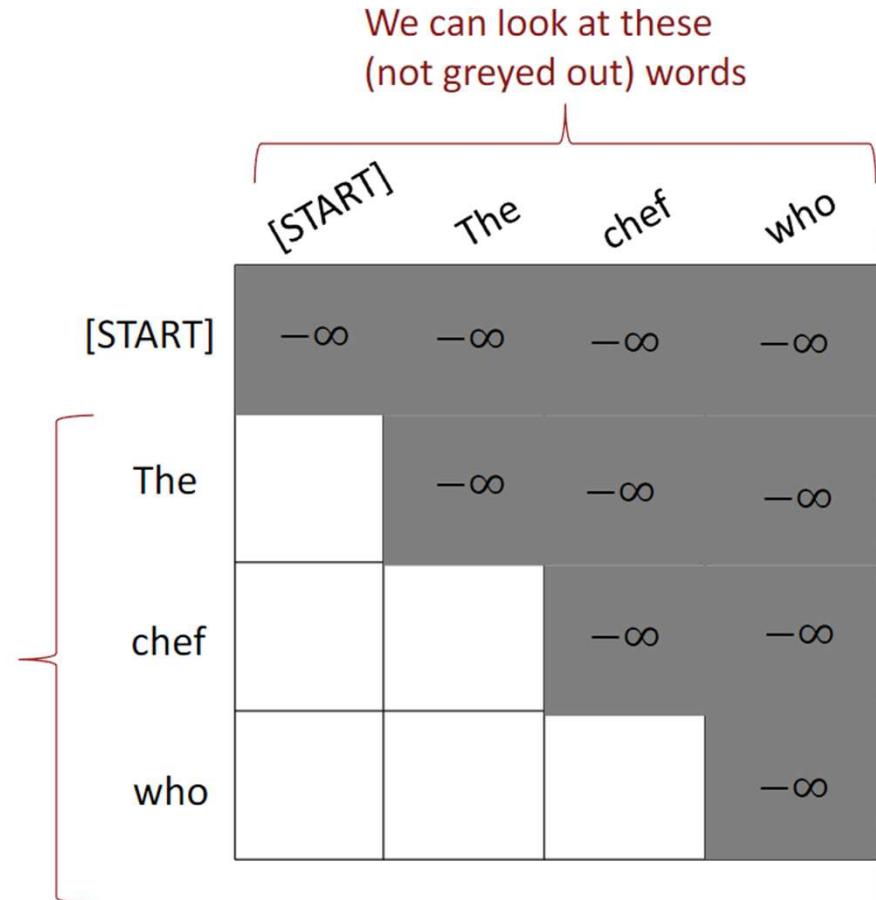


Decoder: Masked Multi-Head Self-Attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^T k_j, & j < i \\ -\infty, & j \geq i \end{cases}$$

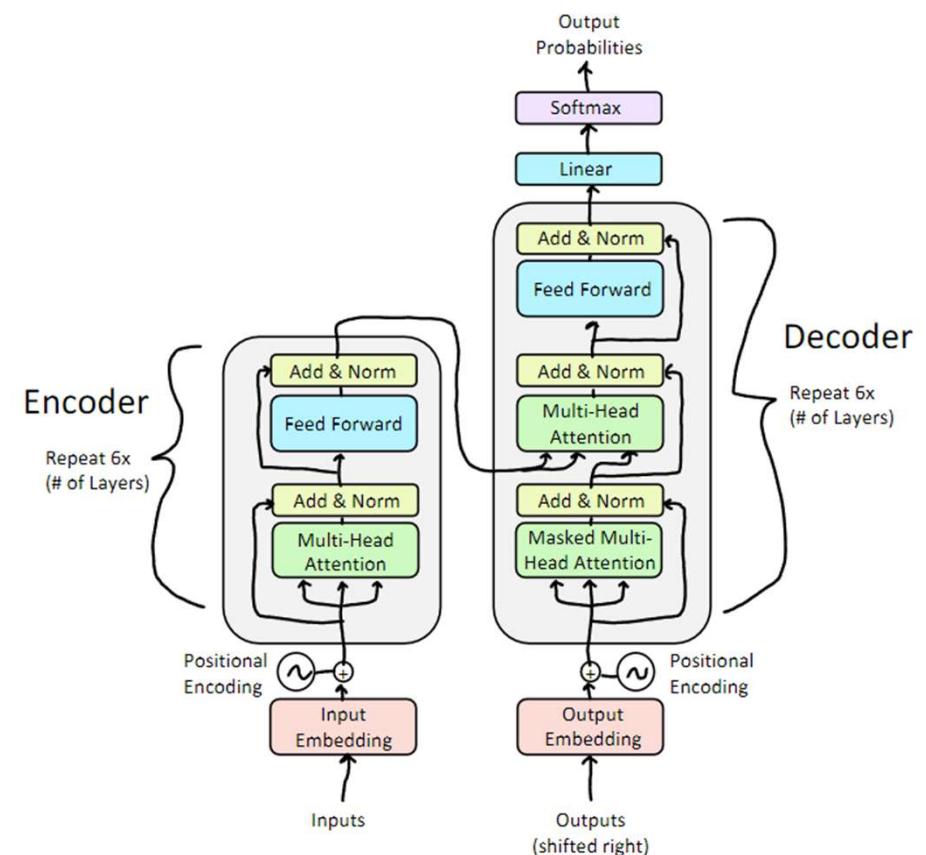
For encoding
these words





Decoder

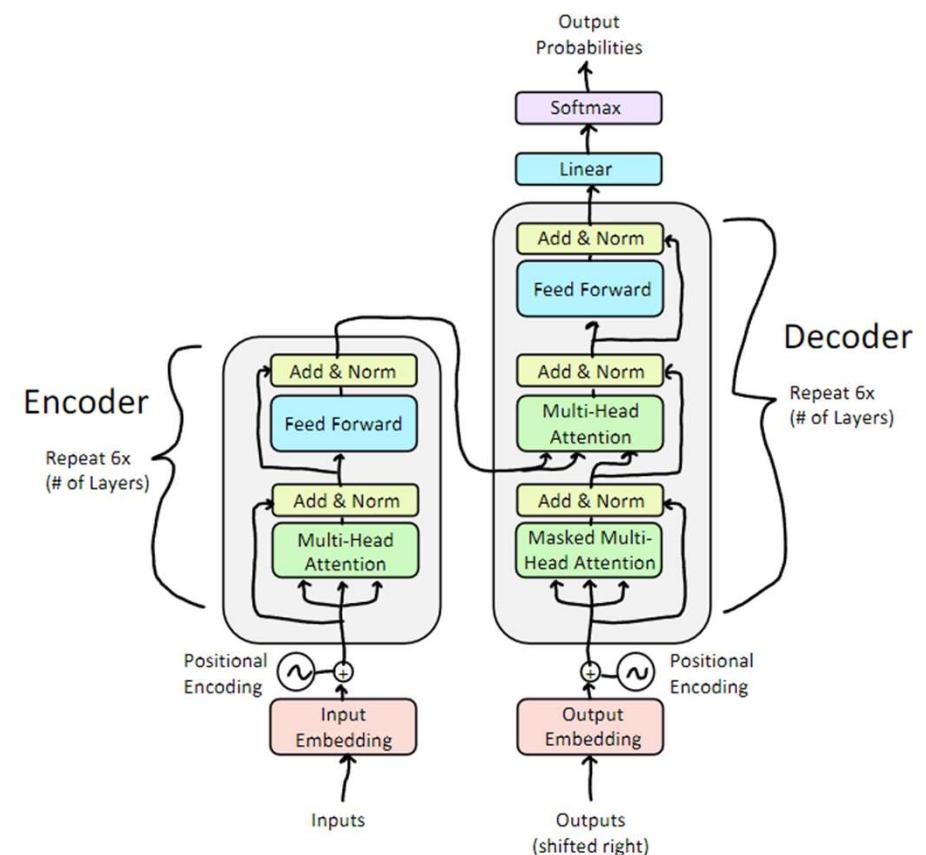
- Add a feed forward layer (with residual connections and layer norm)
- Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)
- Add a final softmax to generate a probability distribution of possible next words!





Decoder

- Add a feed forward layer (with residual connections and layer norm)
- Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)
- Add a final softmax to generate a probability distribution of possible next words!



Thank You!

