DSA4213

Lecture 5 - 20240217

Dr Vishal Sharma,
Data Science Lead, H2O.AI

Are We Recording

Sources: Attribution and credit to various internet sourced and copyrighted content with edits applied

Groups
Research Gateway
Project

Pending – tutorials, sample MCQ questions, sample projects

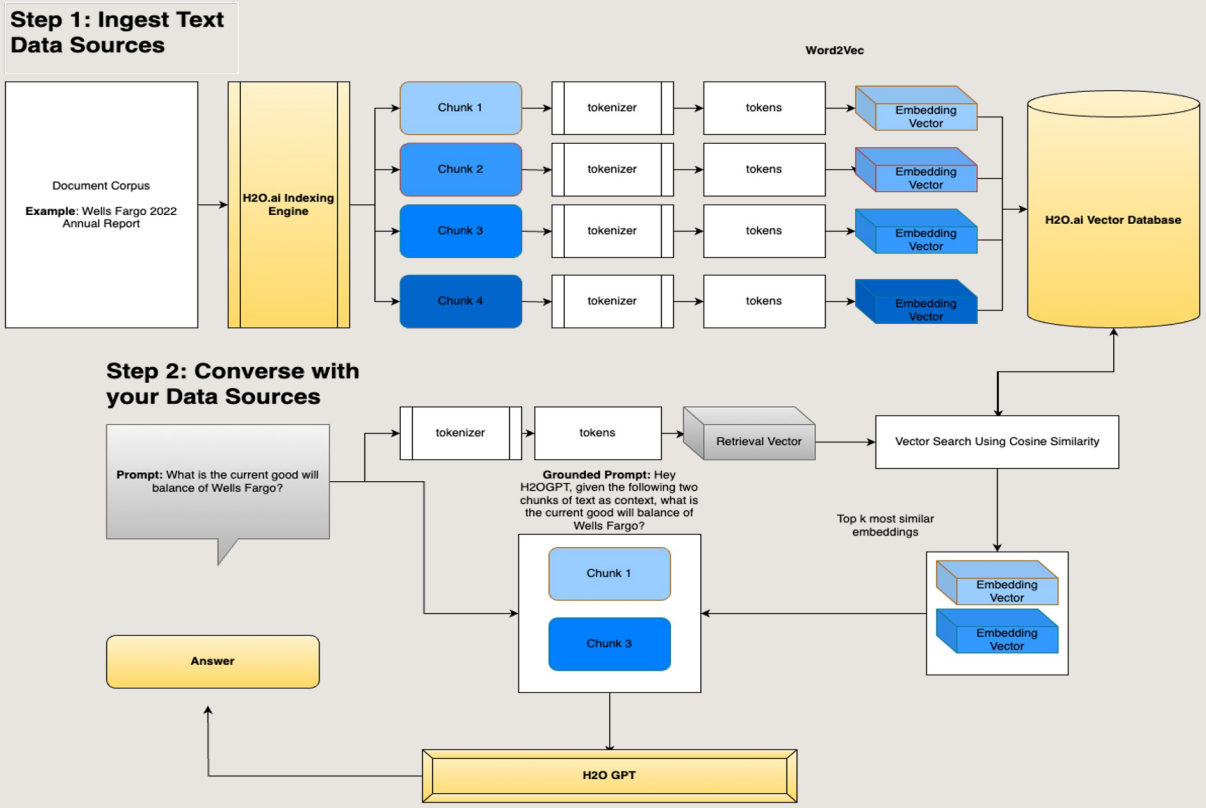
Sources: Attribution and credit to various internet sourced and copyrighted content with edits applied

For Project Use RAG pipeline

RAG as a system is a particularly good use of vector databases.

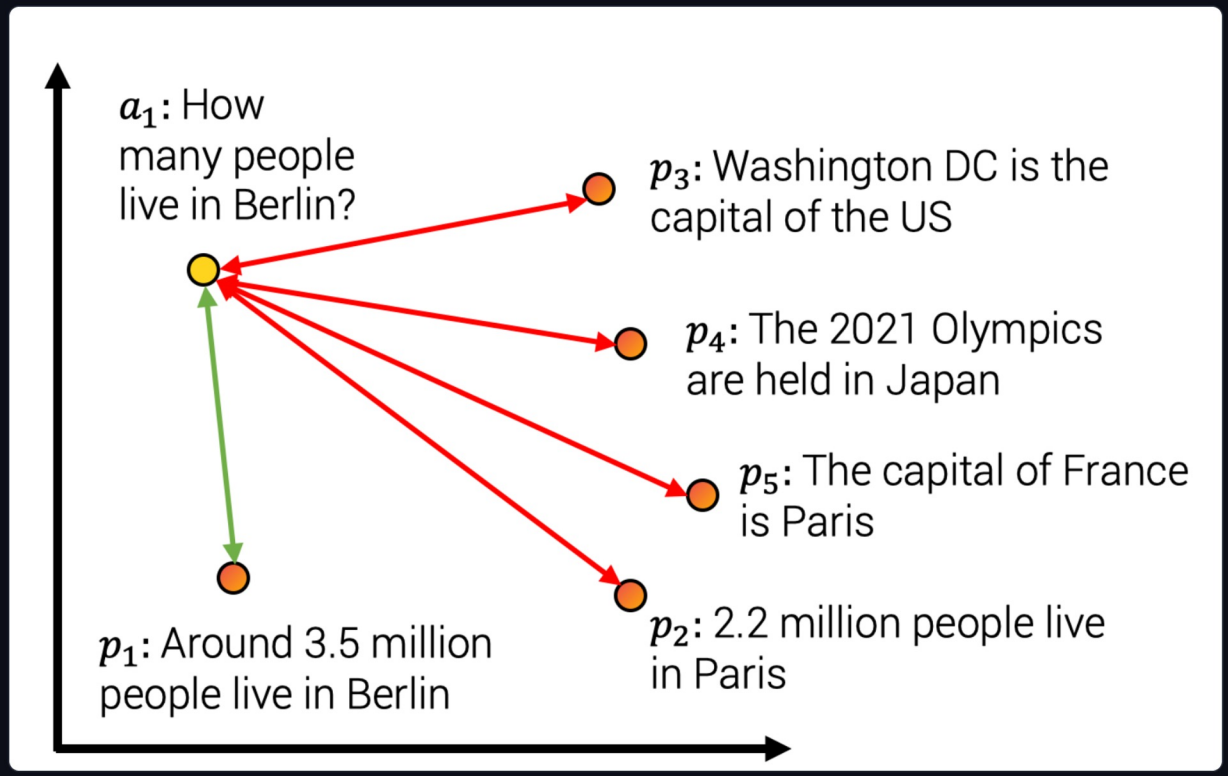
RAG systems take advantage the context window for LLMs, filling it with only the most relevant examples from real data.

This “grounds” the LLM to relevant context and greatly minimizes any hallucination.

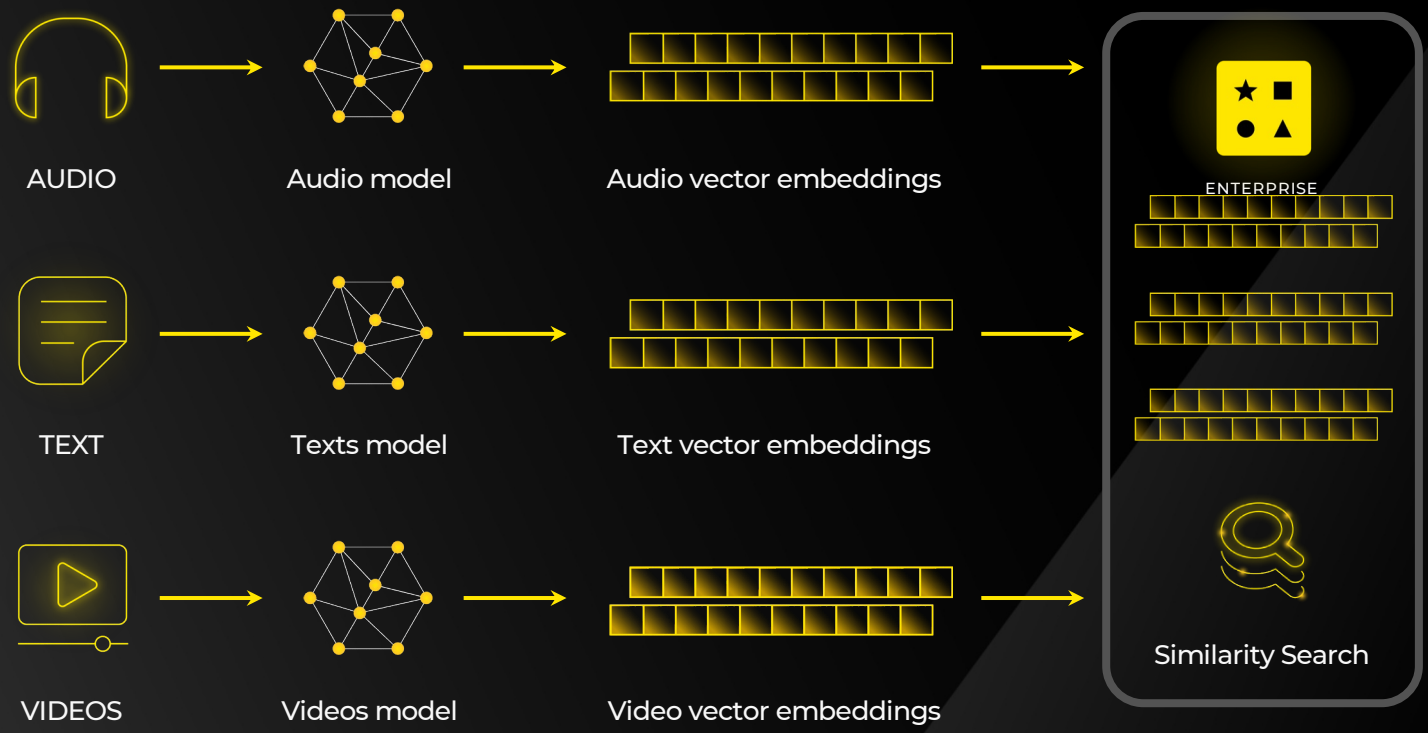


Embedding Models

Source: https://huggingface.co/blog/1b-sentence-embeddings

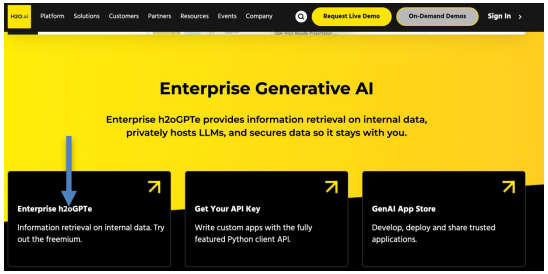


Embeddings - Can go beyond Text



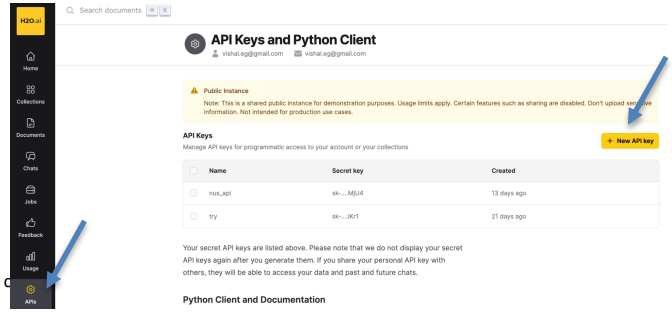
For Project Use RAG pipeline from H2O.ai

1. Visit h2o.ai
2. Scroll down a bit, click on Enterprise h2ogpte



3. Create account and start using
   - Whole group to use one account

4. Go to “APIs” -> “New API key” for pythonic development



Sources: Attribution and c
applied

GenAI Events

1.OpenAI's Sora: text-to-video generative model that can create astonishingly real videos.

2.Google's Gemini 1.5: new-generation multimodal model which boasts an impressive one million tokens context window. mixture of experts architectures.

3.Cohere's Aya: a new multilingual LLM supporting 101 languages.

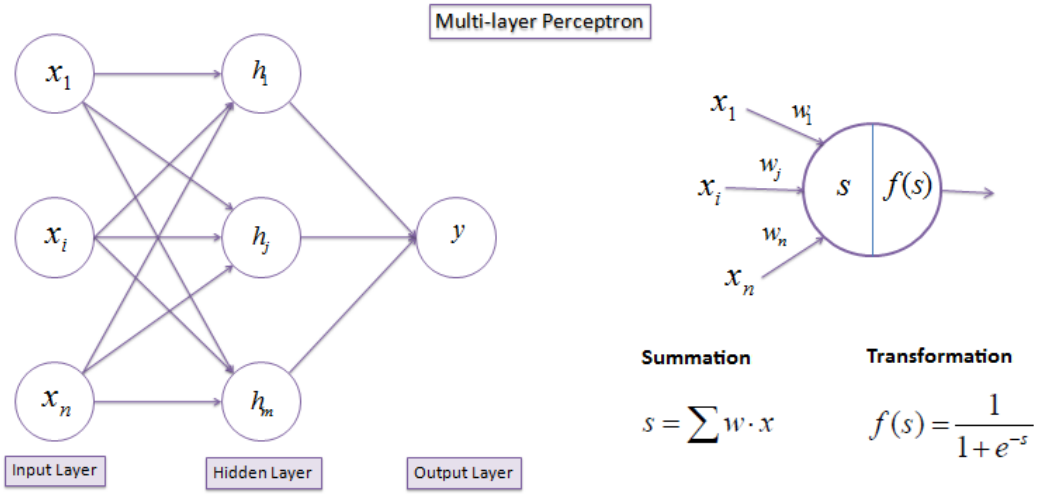
4.Meta's V-JEPA: a non-generative model capable of predicting missing parts of videos using an abstract representation space

5.Stability AI's Stable Cascade: a new text-to-image model. Based on the new Würstchen architecture, which enables efficiency and speed in large-scale image generation models.

Modern Neural Network Architectures

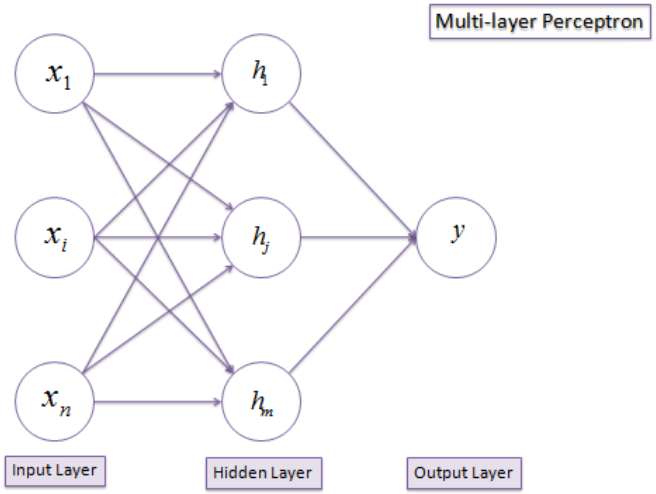
- Recap of Feedforward Neural Networks
- RNNs and LSTM
- Attention Mechanism
- Transformers
- GPTs and LLMs
- Fine tuning and RAG

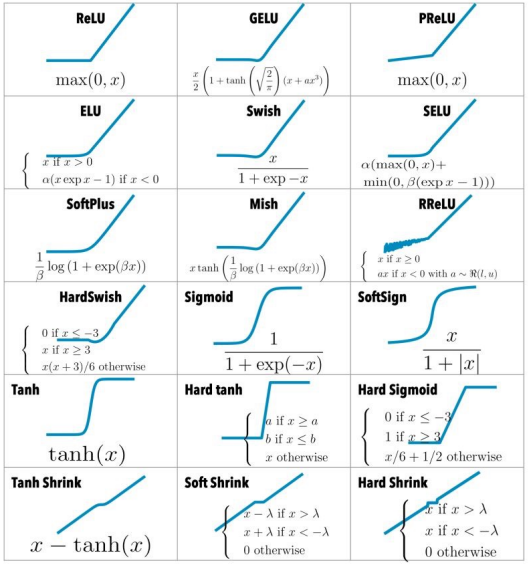
Recap of Feedforward NNs



- Nonlinear Hidden Layer allow nonlinear function approximation / curve fitting
- Training MLPs on real world problems is non-trivial due to complex error surface

Recap of Feedforward NNs





- Nonlinear Hidden Layer allow nonlinear function approximation / curve fitting
  - Possible due to nonlinear activation units
- Training MLPs on real world problems is non-trivial due to complex error surface

Deep learning Simplifies the learning

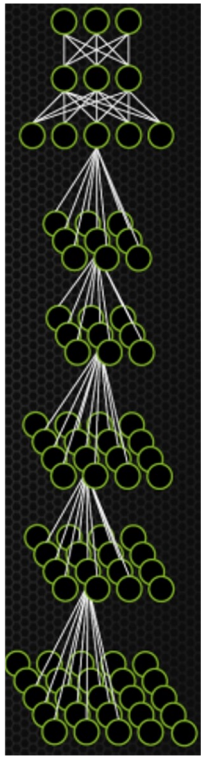
Greedy Layer-Wise Training of Deep Networks

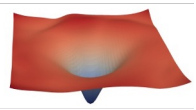
Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle
Université de Montréal
Montréal, Québec
{bengioy,lamblinp,popovicd,larocheh}@iro.umontreal.ca

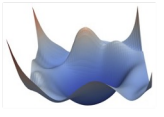
[Bengio et al 2006]

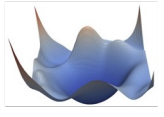
much less expressive than deep ones.

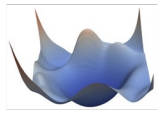
However, until recently, it was believed too difficult to train deep multi-layer neural networks. Empirically, deep networks were generally found to be not better, and often worse, than neural networks with one or two hidden layers (Tesauro, 1992). As this is a negative result, it has not been much reported in the machine learning literature. A reasonable explanation is that gradient-based optimization starting from random initialization may get stuck near poor solutions. An approach that has been explored with some success in the past is based on *constructively* adding layers. This was previously done using a

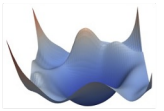












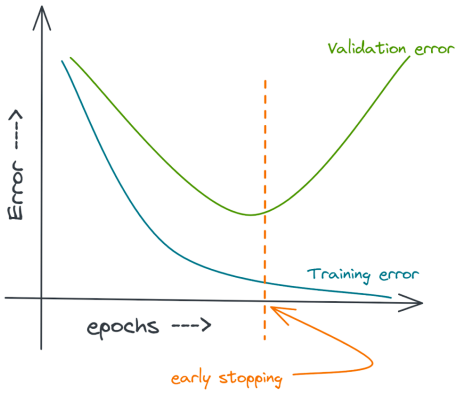
- It took a long time and much work to make deep neural networks practical, thanks to GPUs for making it happen

Deep learning models require Regularization

- A full loss function includes **regularization** over all parameters w, e.g., L2 regularization:

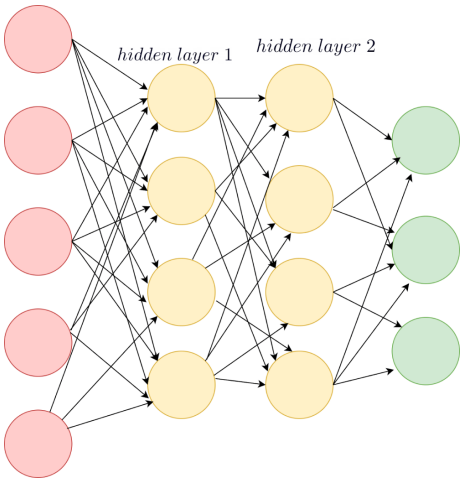
$$Cost\ function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

- Regularization produces models that generalize well when we have a “big” model

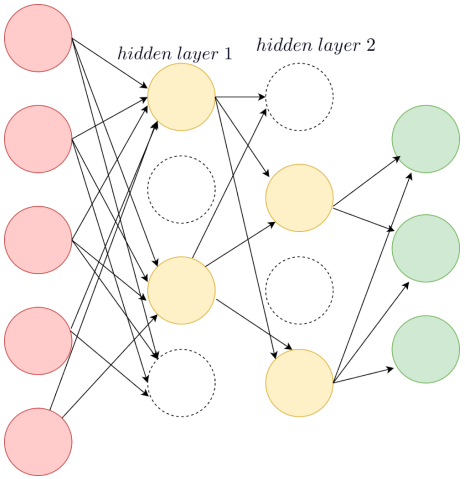


Deep learning models require Regularization

Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)



Dropout involves dropping neurons in the hidden layers. During training, each neuron is assigned a "dropout"probability, like 0.5

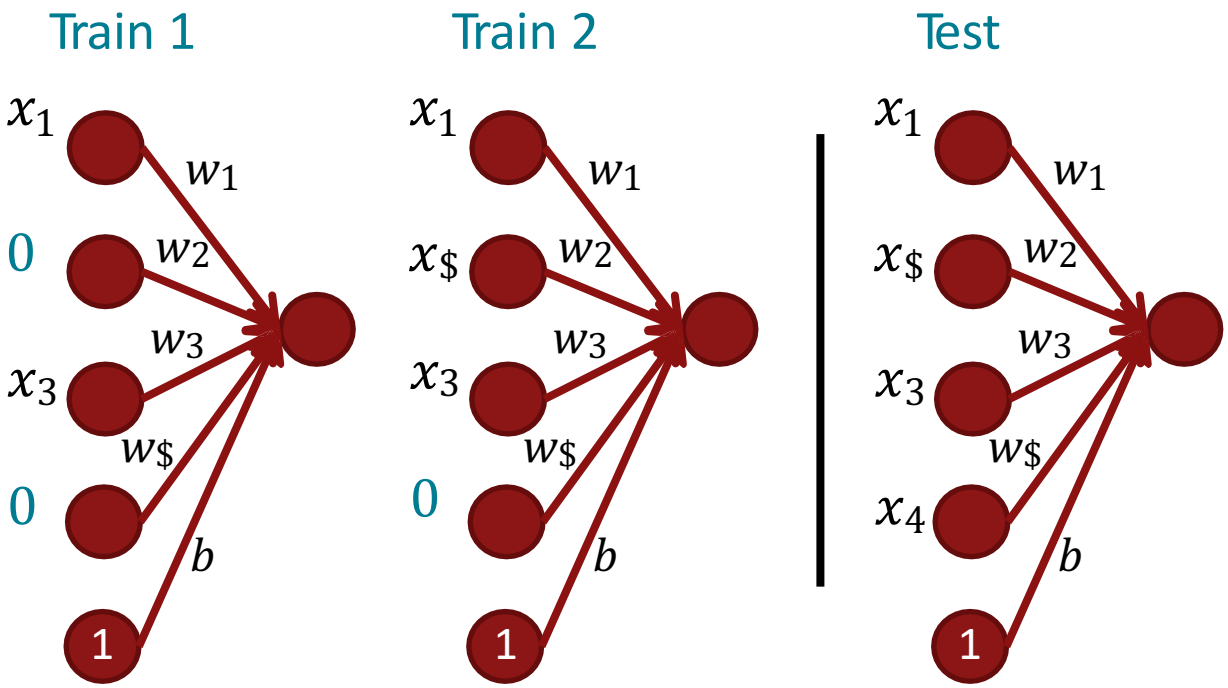


Deep learning models require Regularization

Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

During training
- For each data point each time:
- Randomly set input to 0 with probability p "dropout ratio" (often p = 0.5 except p – 0.15 for input layer) via dropout mask

During testing
- Multiply all weights by 1 – p
- No dropout



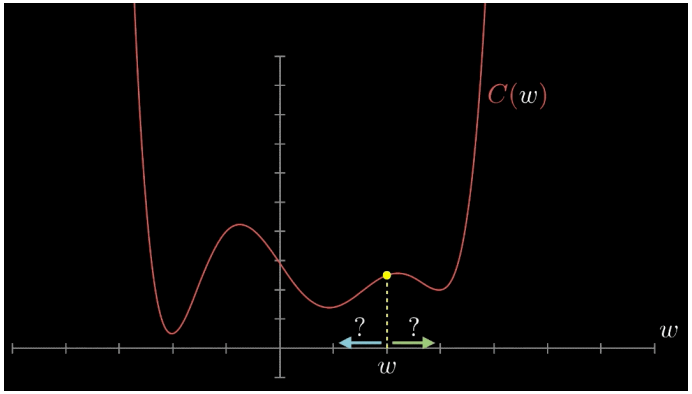
$y = w_1x_1 + w_3x_3 + b$

$y = w_1x_1 + w_2x_2 + w_3x_3$

$y = (1-p)(w_1x_1 + w_2x_2 + w_3x_3 + w_\$x_\$)$

Parameter Initialization is Important

- You normally must initialize weights to small random values (i.e., not zero matrices!)
  To avoid symmetries that prevent learning/specialization



- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize all other weights ~ Uniform(–r, r), with r chosen so numbers get neither too big or too small [later, the need for this is removed with use of layer normalization]

Optimization Algorithms – Variants of Gradient Descent

- Usually, plain **SGD** or mini-batch SGD will works just fine!
  However, getting good results will often require hand-tuning the learning rate
  E.g., start it higher and halve it every k epochs (passes through full data, shuffled or sampled)

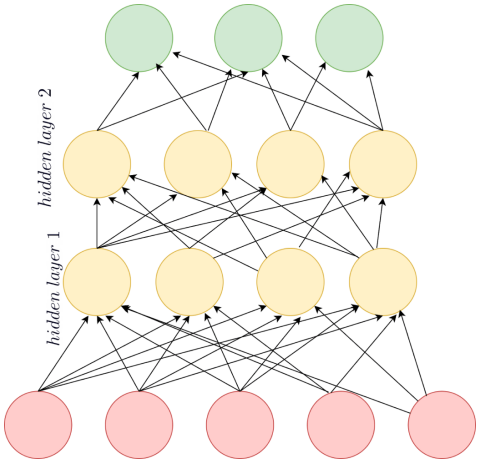
- For more complex nets, or to avoid worry, try more sophisticated "adaptive" optimizers that scale the adjustment to individual parameters by an accumulated gradient
- These models give differential per-parameter learning rates

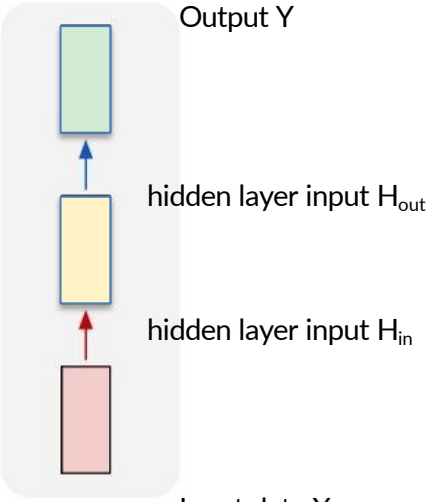
  Adagrad – Simplest member of family, but tends to "stall early"

  RMSprop

  Adam – A fairly good, safe place to begin in many cases

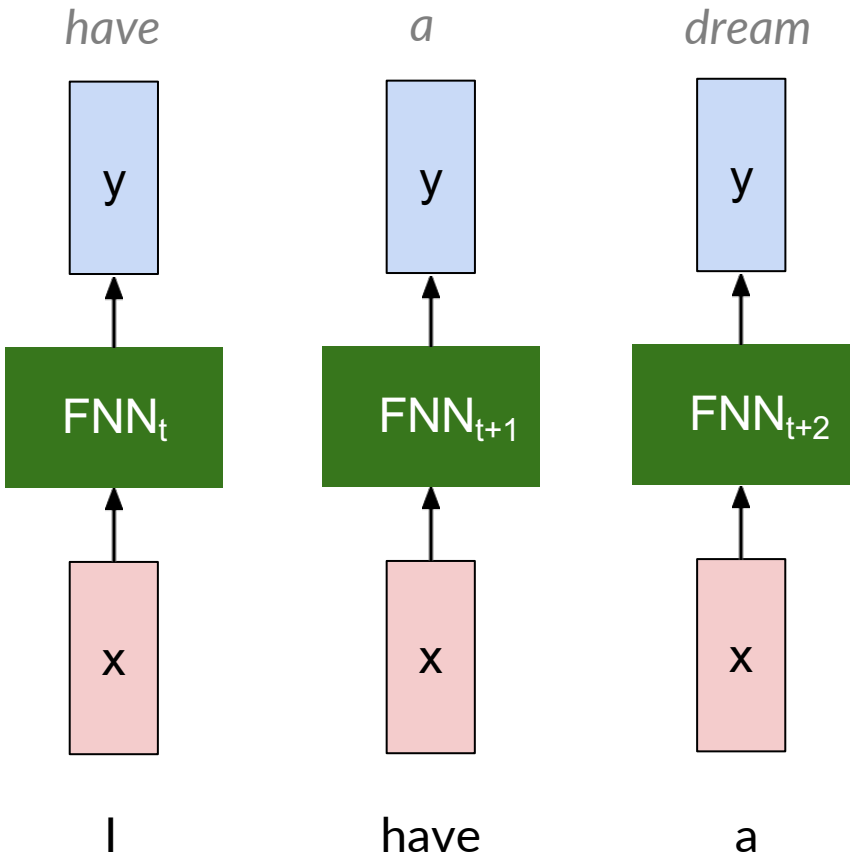
Feedforward NN





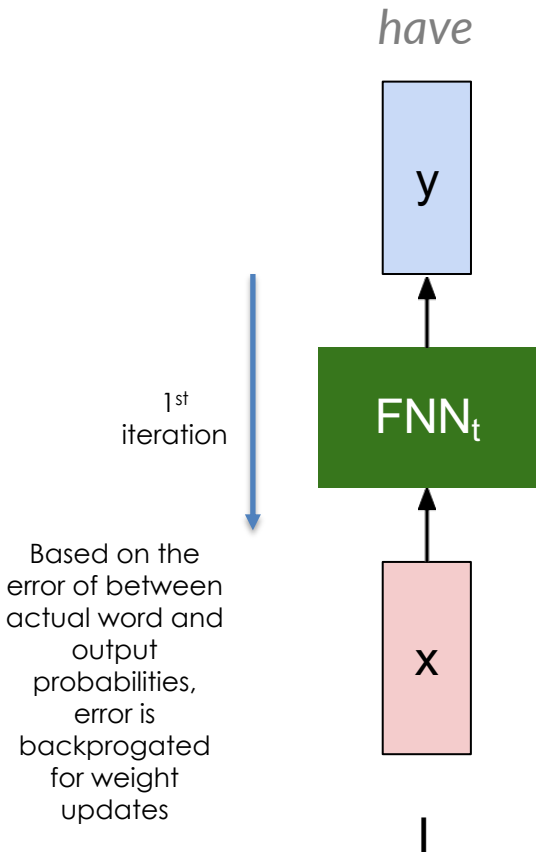
Feedforward NN – can it process sequential data

Eg language modelling – next word prediction
Eg – if you are modelling a sentence “I have a dream”



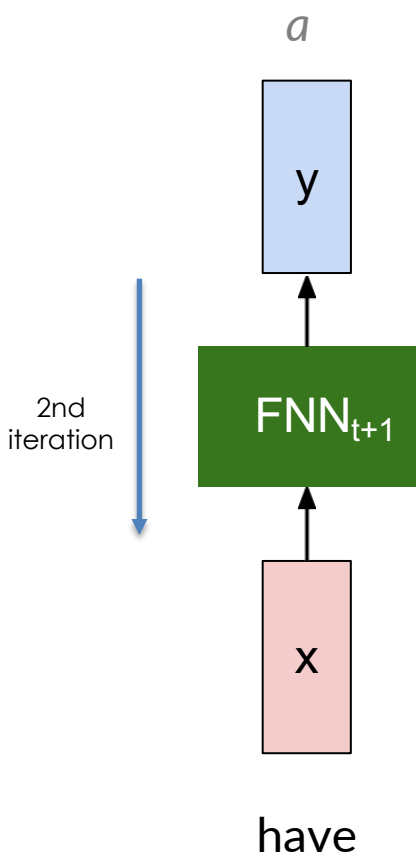
SGD training

Eg language modelling – next word prediction



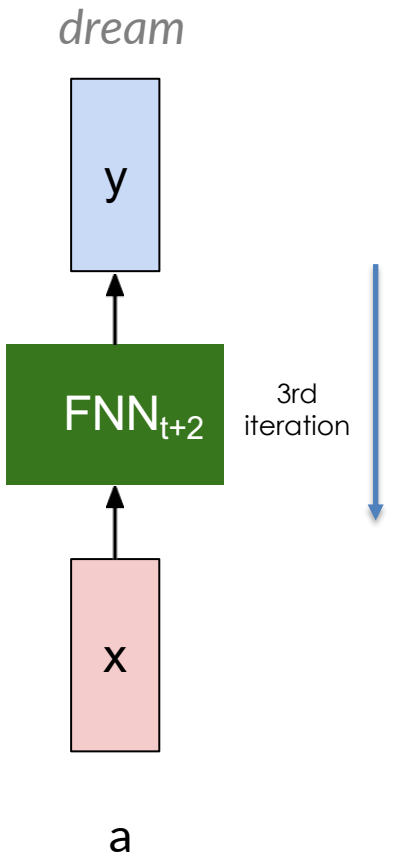
SGD training

Eg language modelling – next word prediction



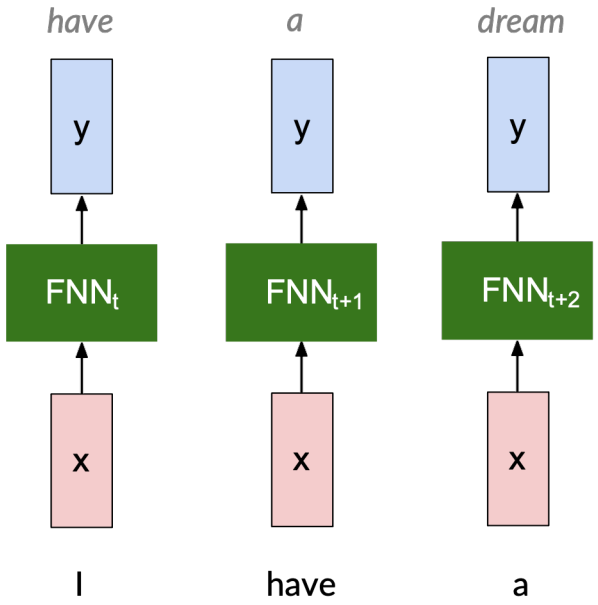
SGD training

Eg language modelling – next word prediction



Sequential information is not captured

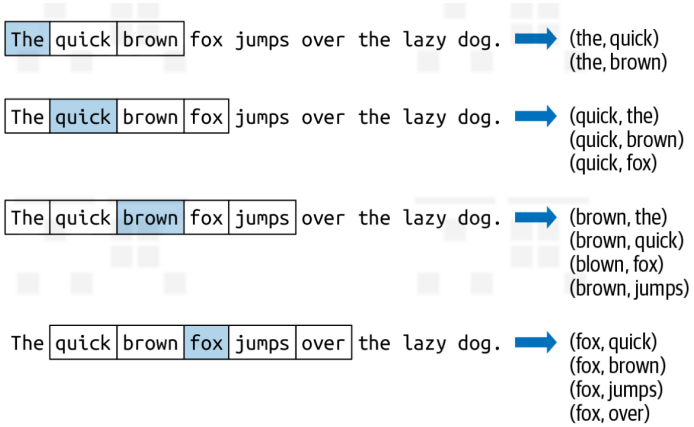
Sequential data information (context of all the words which came before) is not captured for next word prediction

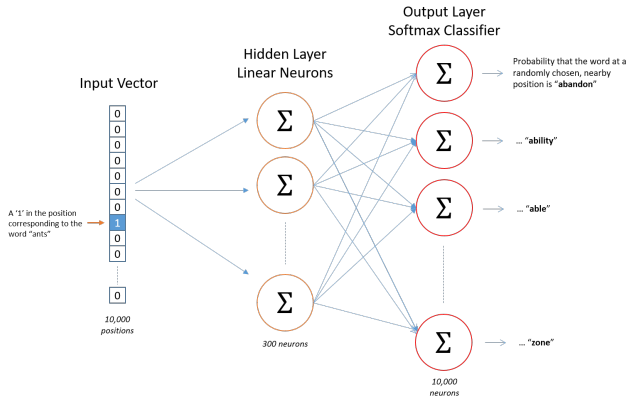


At a particular timestep t, only that word input at time t is used to create hidden state for that particular word at time t

Sequential information is not captured

Sequential data information (context of all the words which came before) is not captured for next word prediction



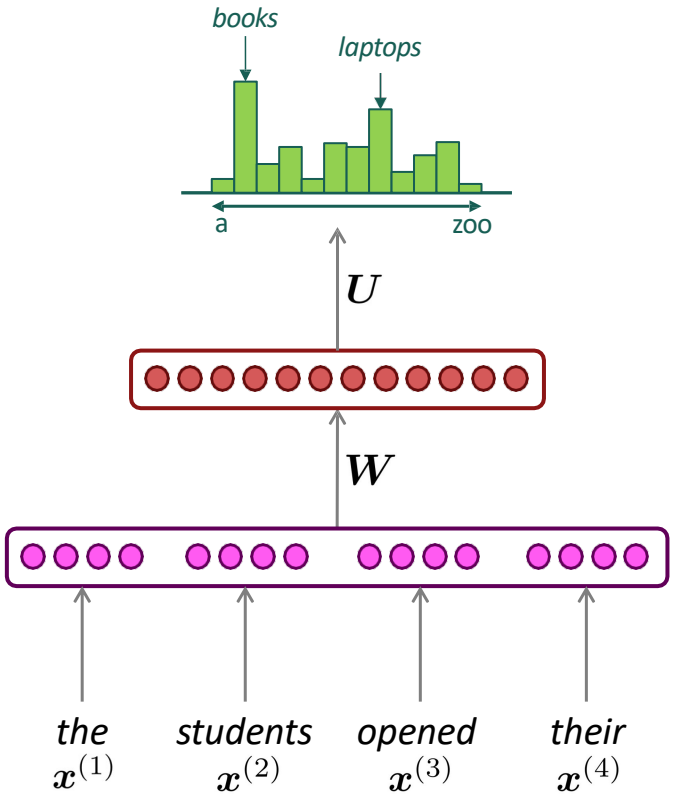


word2vec kind of architectures have limited context length

Plus these models came in 2013

A fixed-window neural Language Model

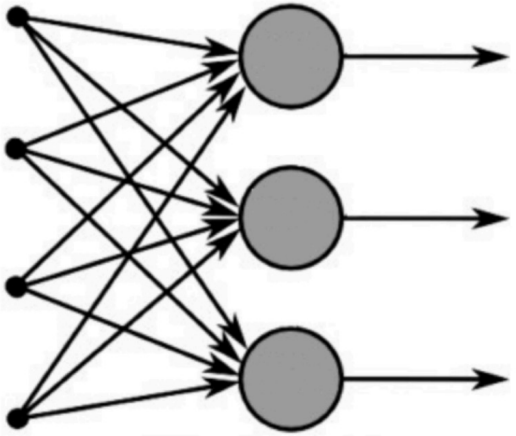
Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model



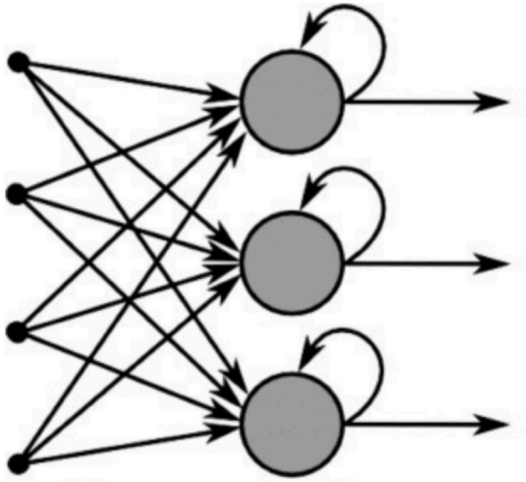
Issues:

- Fixed window of conext
- Enlarging window enlarges W, more weights to train
  - e.g. l x n x h
    - l = word embedding size
    - n = window size
    - h = size of hidden dim
- Window can never be large enough!
- $x(1)$ and $x(2)$ are multiplied by completely different weights in W. No symmetry in how the inputs are processed.

Recurrent Neural Networks

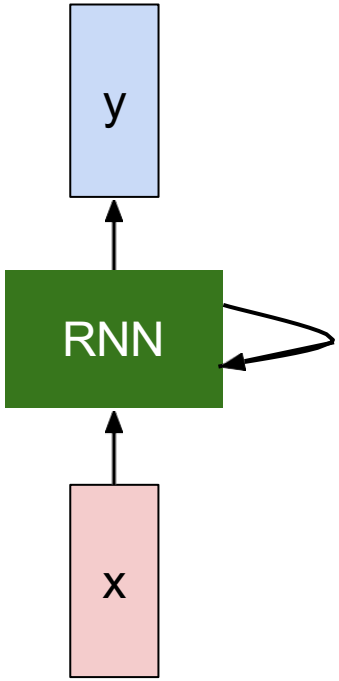


Feed-Forward Neural Network

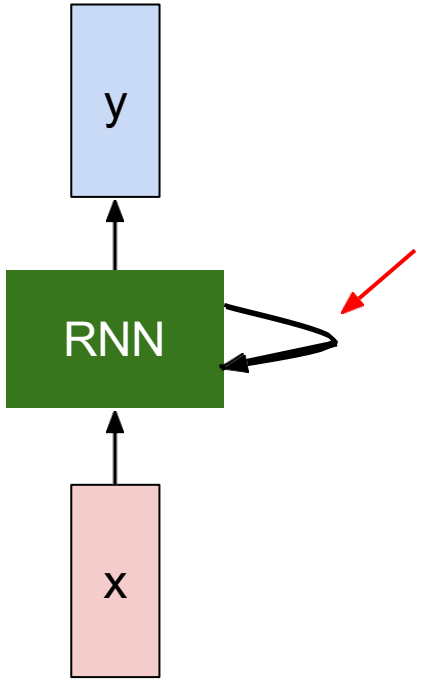


Recurrent Neural Network

Recurrent Neural Networks

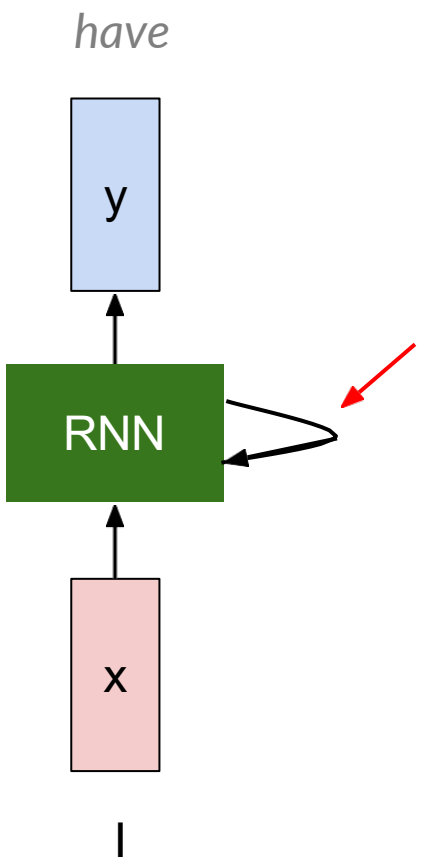


Recurrent Neural Networks



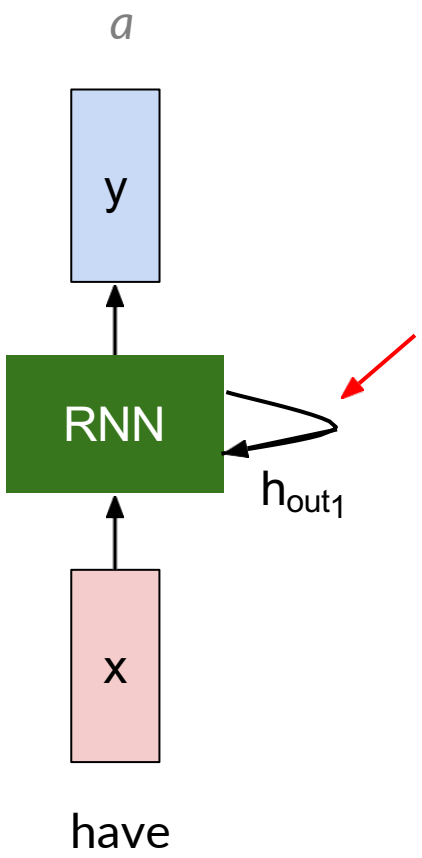
Key idea: RNNs have an "internal state" that is input to the next step

Recurrent Neural Networks



No hidden state yet for first time step from step 0. This can be initilalized to 0 or random hidden state

Recurrent Neural Networks



Output of hidden layer from previous step (hidden state from previous input *"I"*

Recurrent Neural Networks

In equation form

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

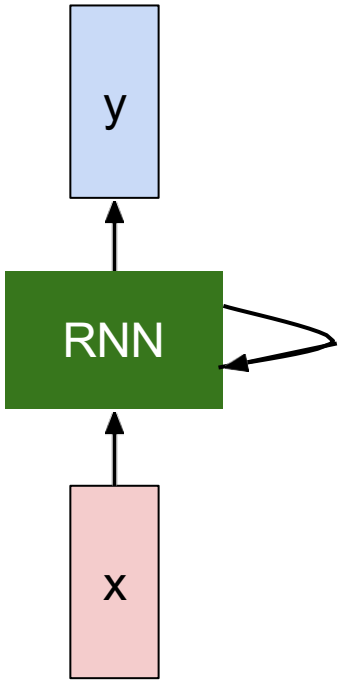
new state — old state — input vector at some time step

some function with parameters W

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

output — new state

another function with parameters W_o



Compared to MLP

In equation form

$$h_t = f_W(x_t)$$

new state

some function with parameters W

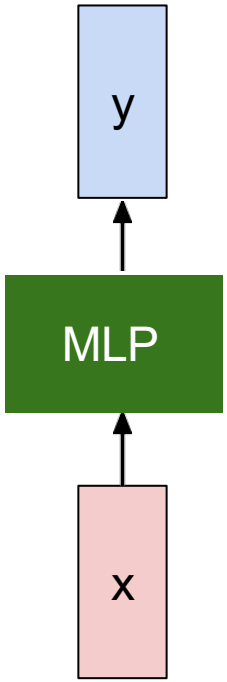
input vector at some time step

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

output

another function with parameters W_o

new state



Recurrent Neural Networks

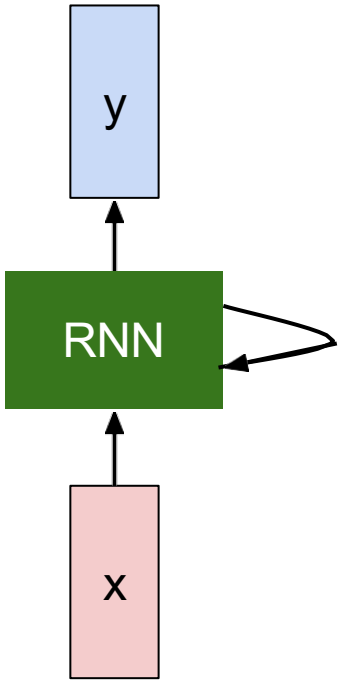
In equation form

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

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

or σ

$$y_t = W_{hy}h_t$$

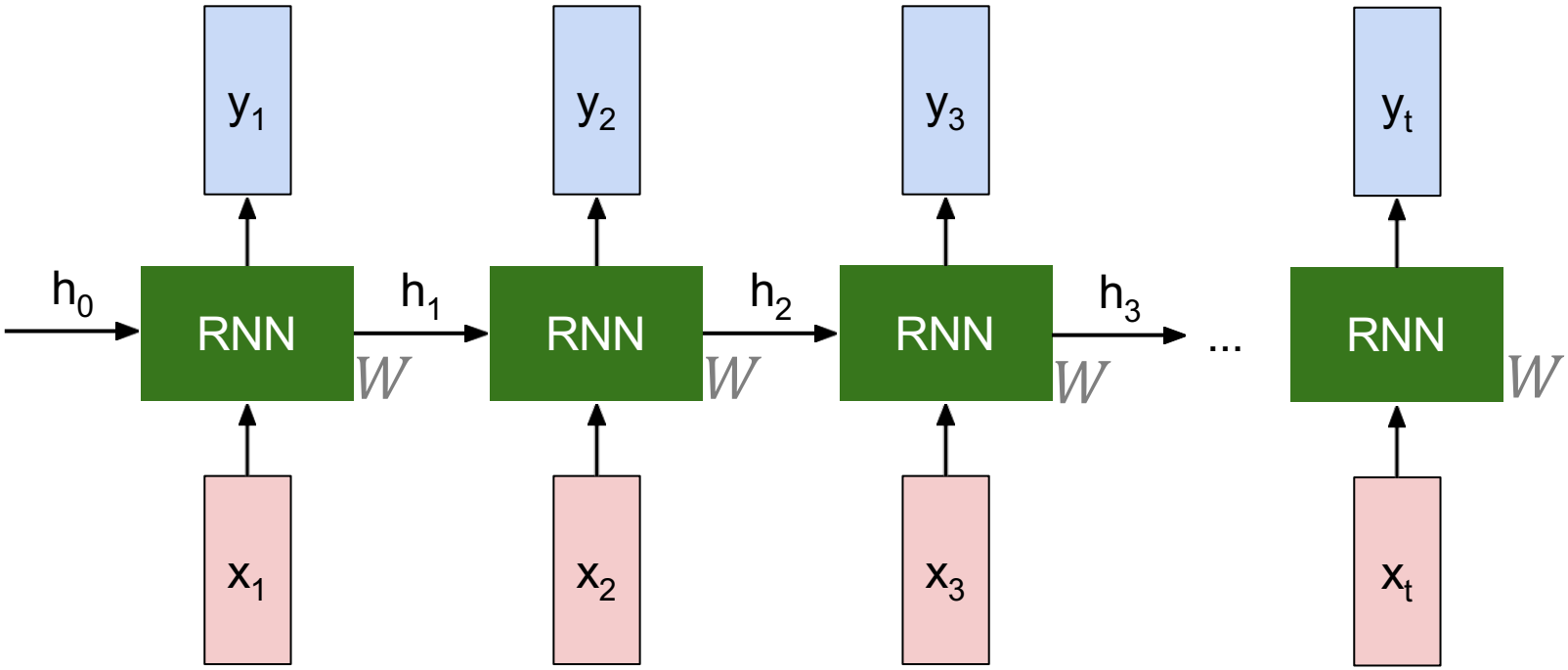


Sometimes called a “Vanilla RNN” or an “Elman RNN” after Prof. Jeffrey Elman

Recurrent Neural Networks

We can unroll RNN over different time steps

Keep the same weights W for every time step



At a particular timestep t, only that word input **as well as** words which came before time t are used to create hidden state for that particular word at time t

Ex Language Model using RNN

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

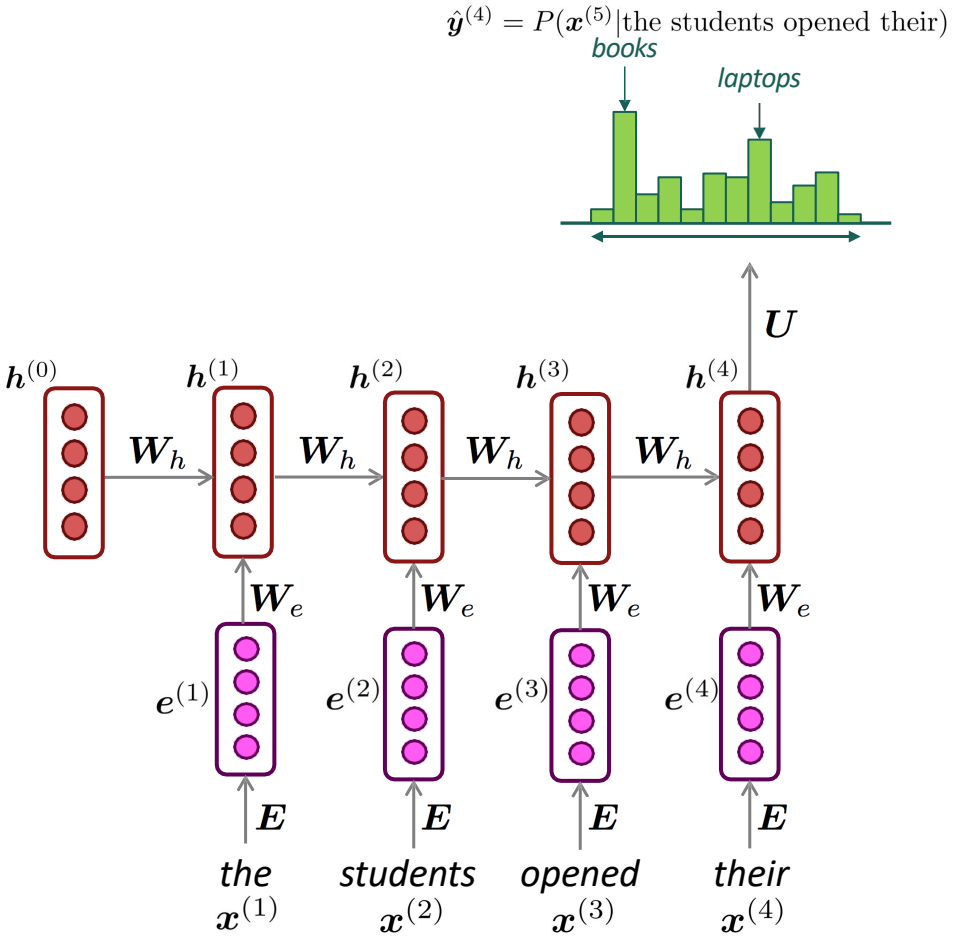
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$

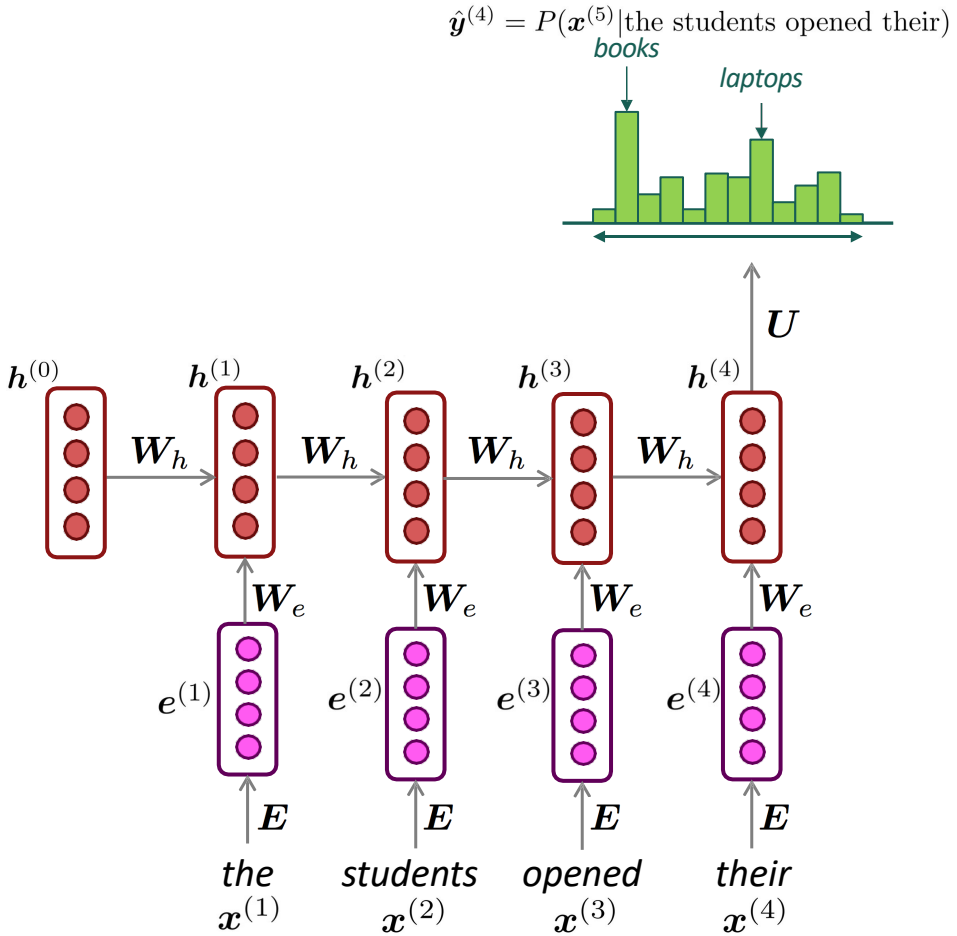


Note: *this input sequence could be arbitrary length*

Ex Language Model using RNN

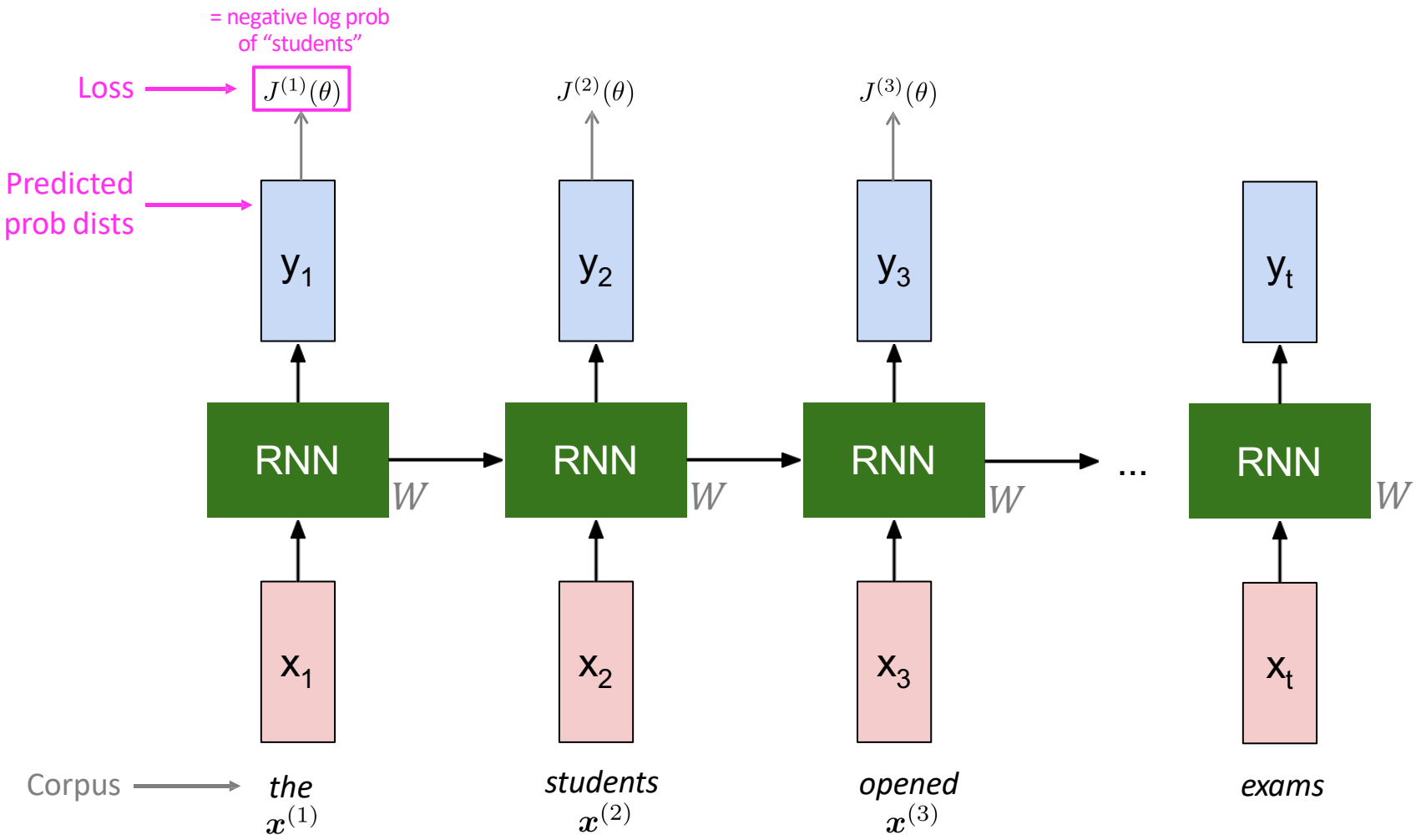
Benefits:

- Incorporate information from all the previous words in the sequence
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.



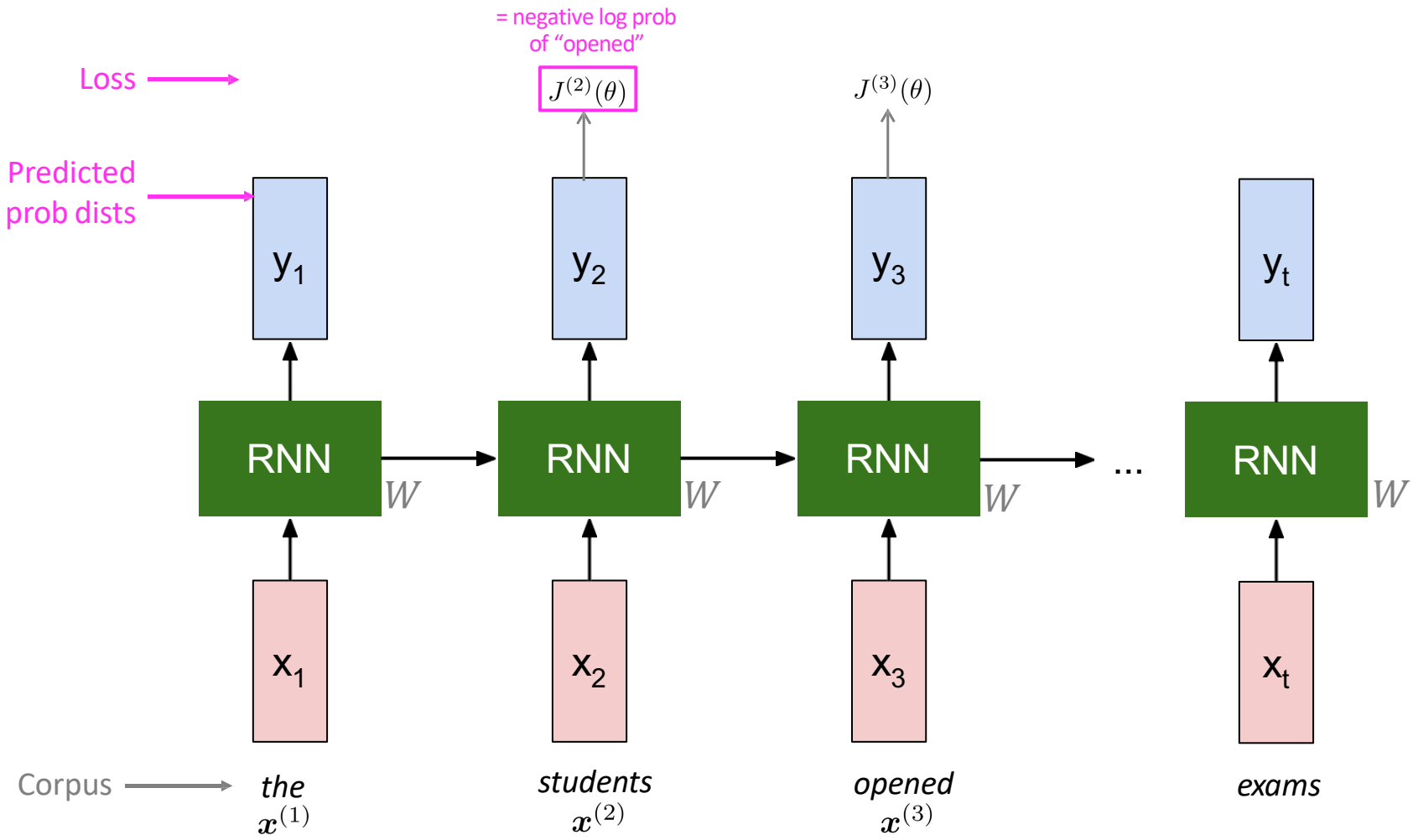
How to Train RNNs

Use unroll RNN



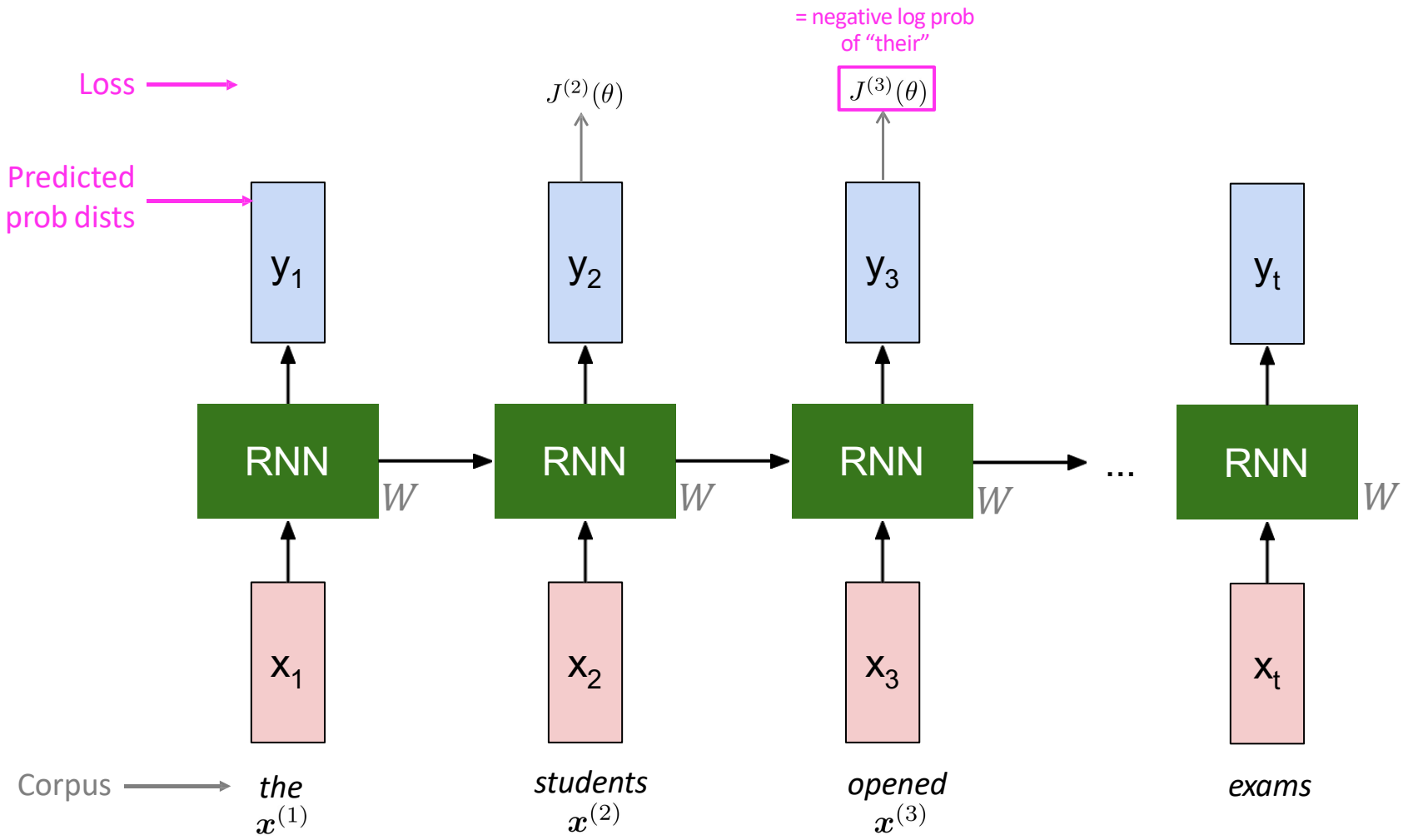
How to Train RNNs

Use unroll RNN



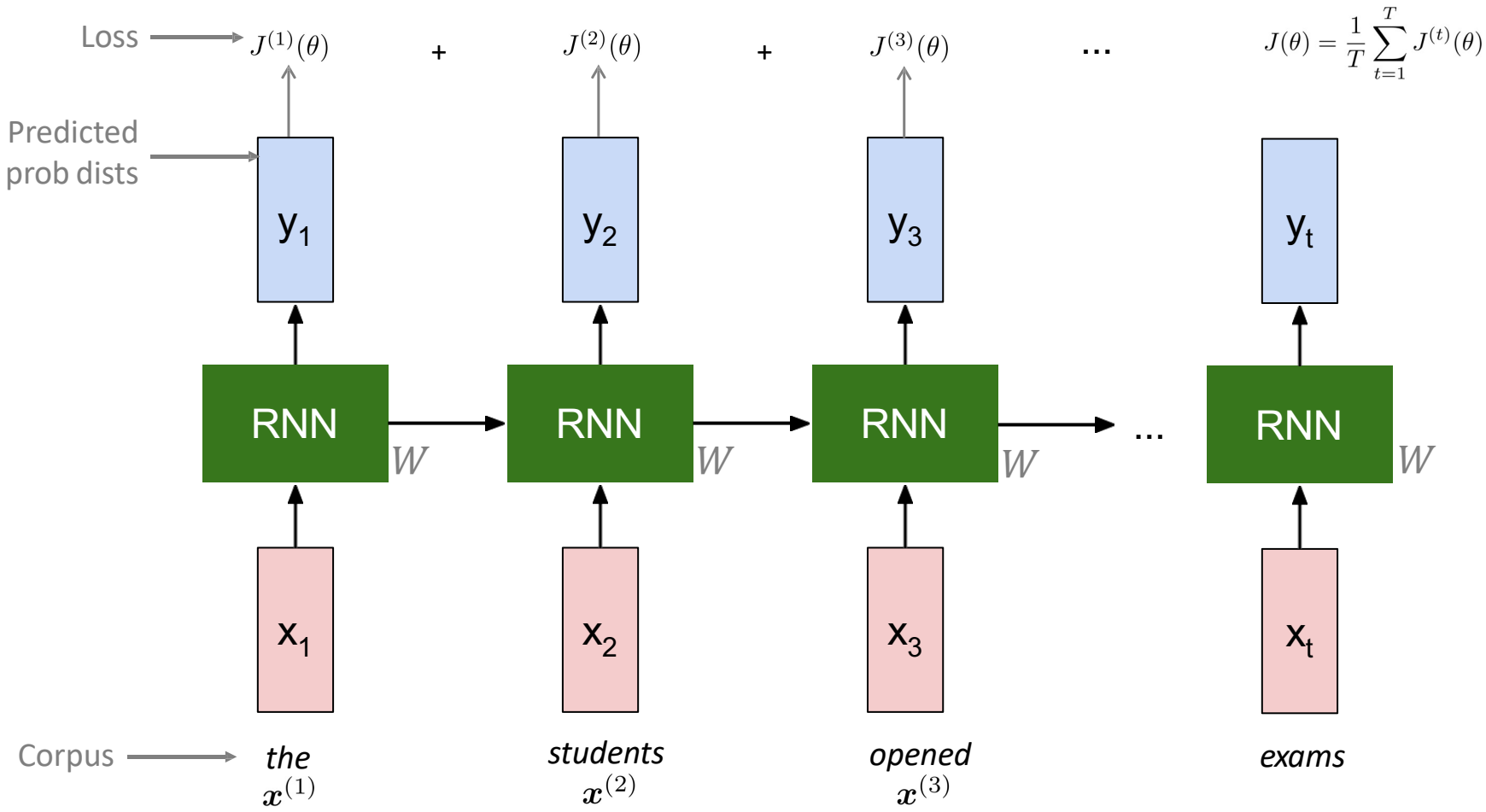
How to Train RNNs

Use unroll RNN

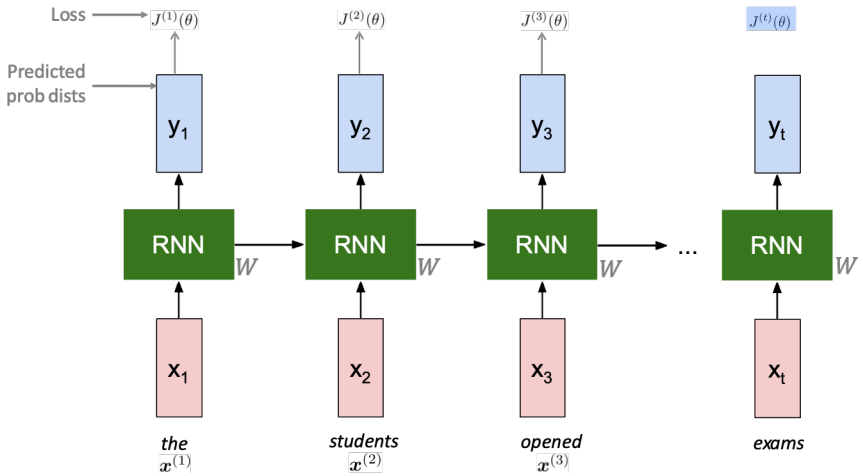


How to Train RNNs

Use unroll RNN



Backpropagation ??

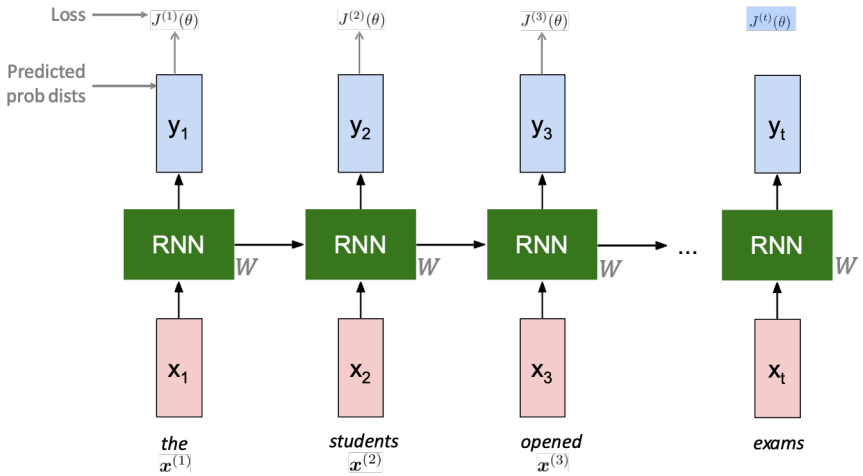


the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W}_h$

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_{\boldsymbol{h}}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_{\boldsymbol{h}}}\bigg|_{(i)}$$

The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears

Backpropagation ??



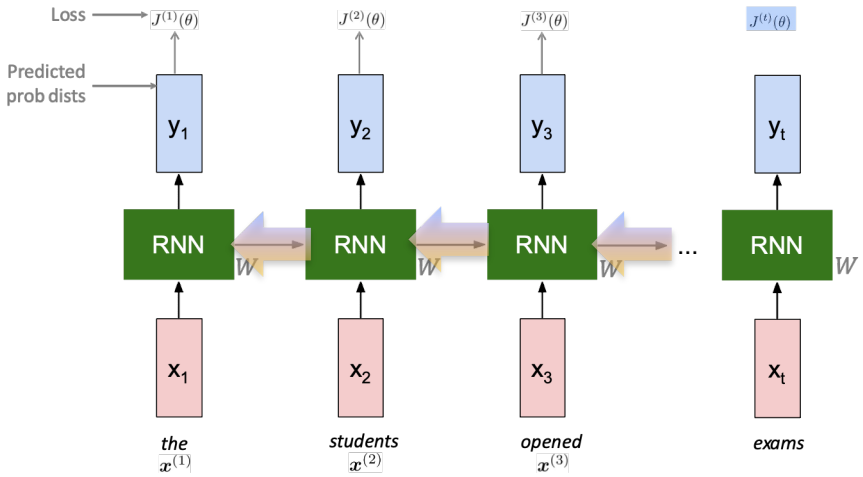
the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W_h}$

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}$$

The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears

Question: How do we calculate this?

Backpropagation through time



the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W_h}$

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}$$

The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears

Question: How do we calculate this?

Answer: Backpropagate over timesteps $i = t, \ldots, 0$, summing gradients as you go.

Apply the multivariable chain rule:

This algorithm is called **"backpropagation through time"** [Werbos, P.G., 1988, *Neural Networks* **1**, and others]

How to Train RNNs

- Computing loss and gradients across whole data - Batch Gradient Descent

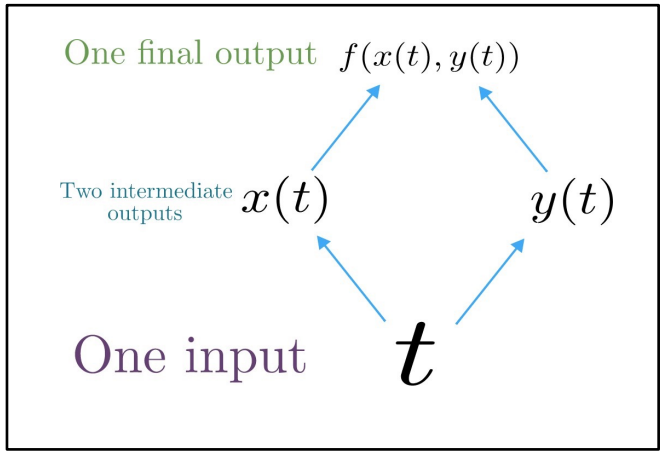
$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

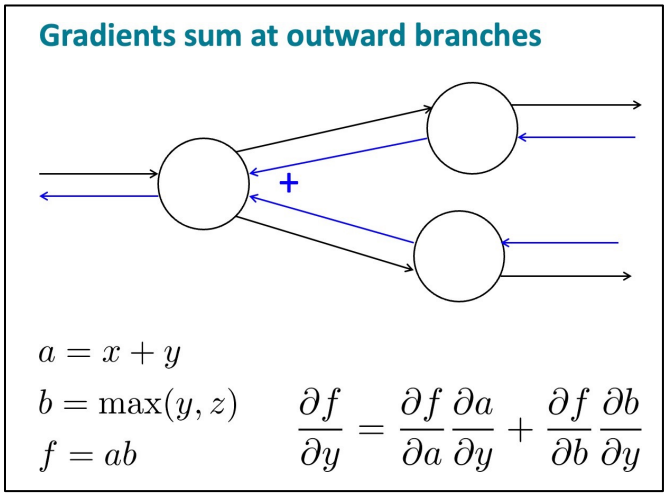
- Stochastic Gradient Descent or mini-batch Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.

Multivariable Chain Rule

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

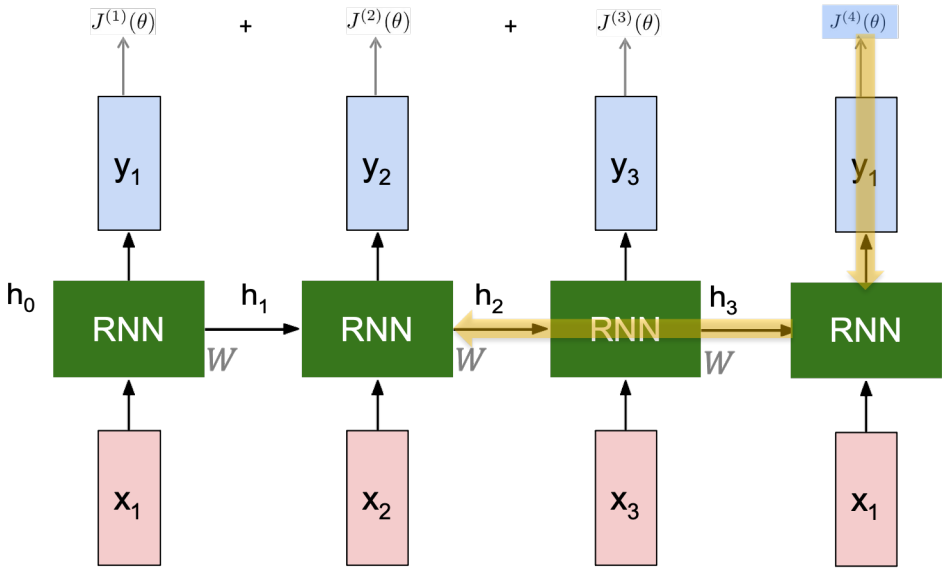
$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$





Source:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

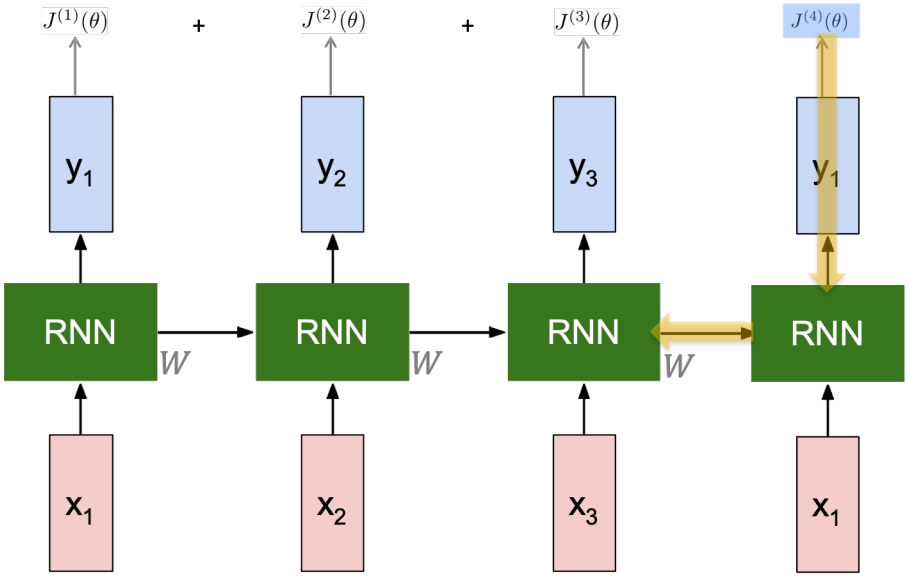
RNNs: BPTT Derivation



chain rule!

Good reference : https://d2l.ai/chapter_recurrent-neural-networks/bptt.html

RNNs: BPTT Derivation

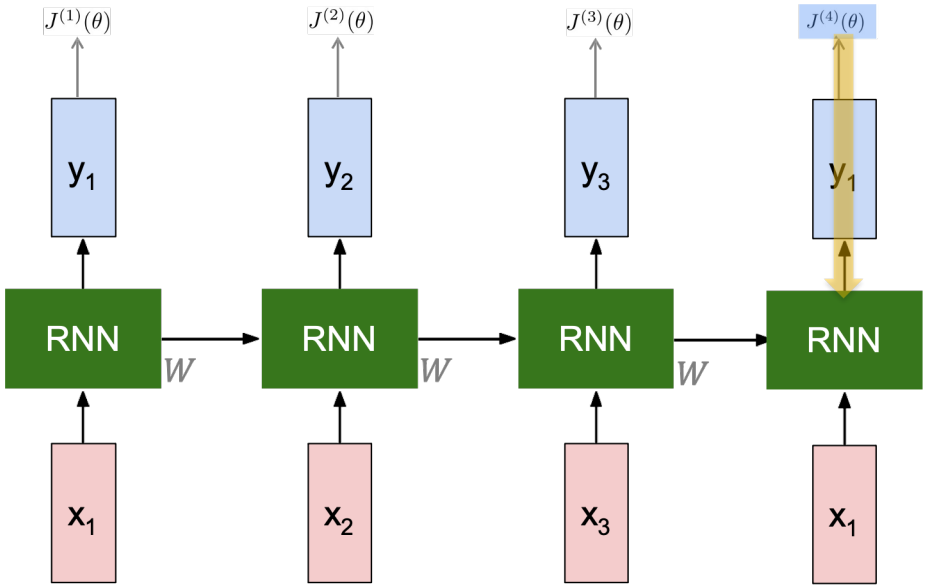


$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(3)}}$$

chain rule!

Good reference : https://d2l.ai/chapter_recurrent-neural-networks/bptt.html

RNNs: BPTT Derivation

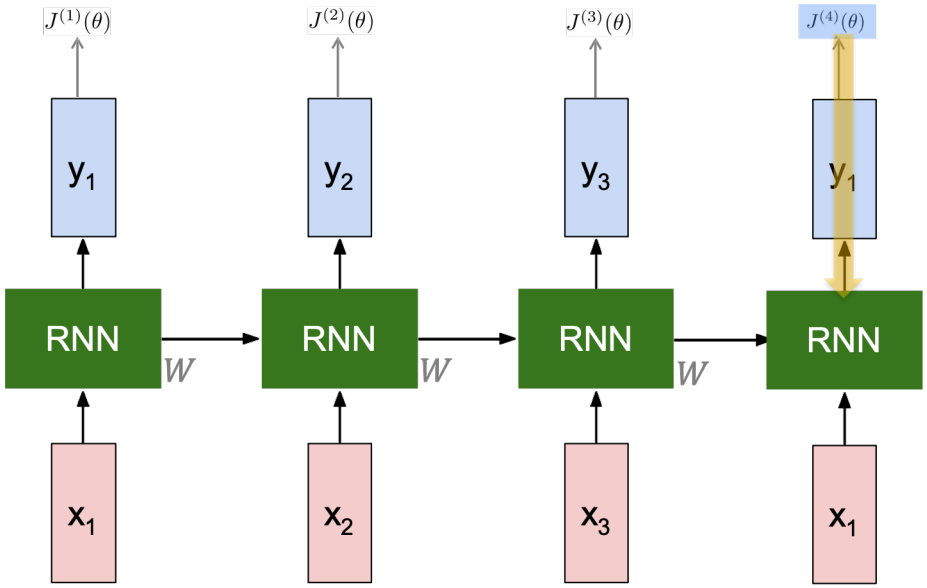


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

chain rule!

Good reference : https://d2l.ai/chapter_recurrent-neural-networks/bptt.html

Problems with RNNs: Vanishing and Exploding Gradients



Vanishing gradient

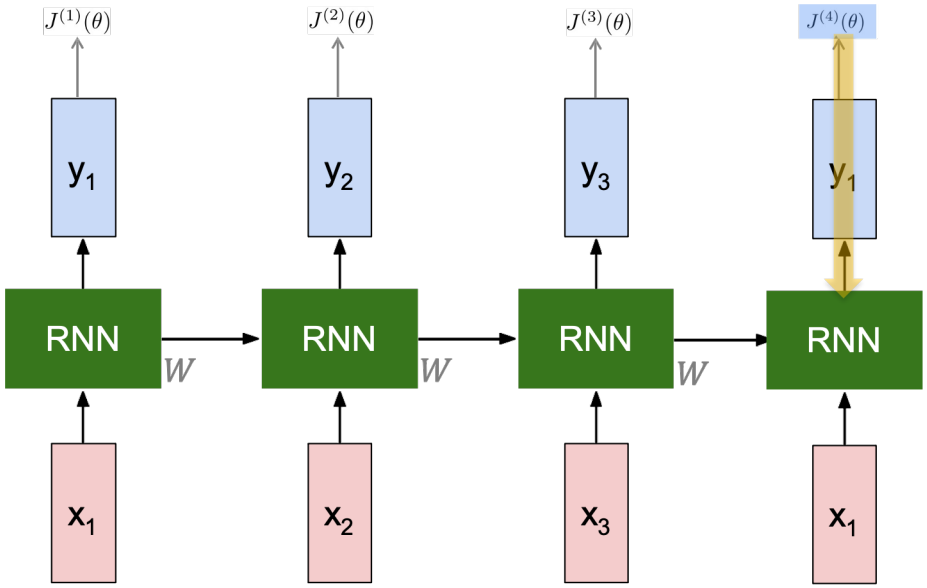
The gradient value gets smaller and smaller as it backpropagates further

$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

chain rule!

If these values are small

Problems with RNNs: Vanishing and Exploding Gradients



Exploding gradient

The gradient value gets larger and larger as it backpropagates further

$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

chain rule!

If these values are large

Problems with RNNs: Vanishing Gradients Proof

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$

- What if σ were the identity function, $\sigma(x) = x$?

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h \qquad \text{(chain rule)}$$

$$= \boldsymbol{I}\, \boldsymbol{W}_h = \boldsymbol{W}_h$$

- Consider the gradient of the loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $\boldsymbol{h}^{(j)}$ on some previous step j. *Let* $\ell = i - j$

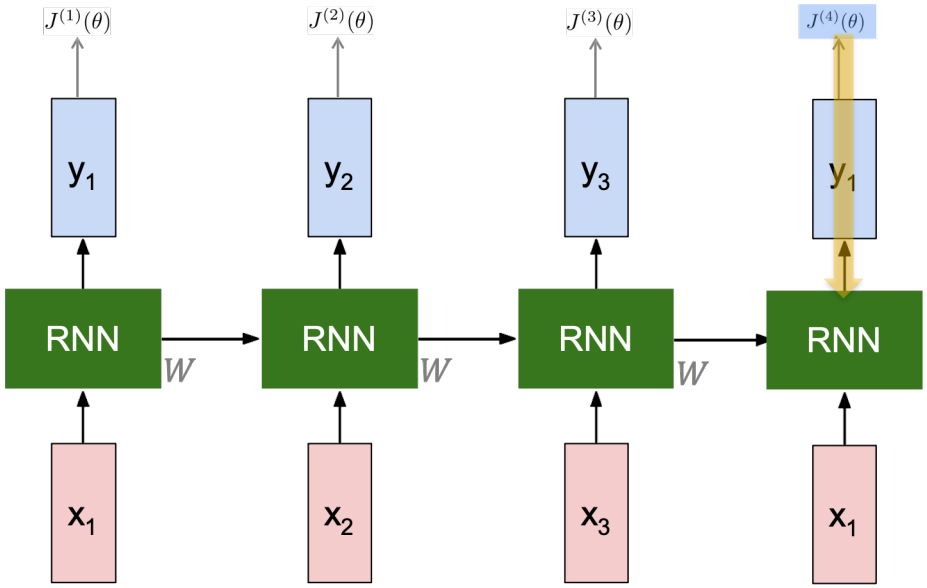
$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \qquad \text{(chain rule)}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \boldsymbol{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell} \qquad \text{(value of } \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\text{)}$$

If W_h is "small", then this term gets exponentially problematic as ℓ becomes large

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf (and supplemental materials), at http://proceedings.mlr.press/v28/pascanu13-supp.pdf

Why is Vanishing Gradients a Problem



The gradient value gets smaller and smaller as it backpropagates further

Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are basically updated only with respect to near effects, not long-term effects.

Worst case – learning is stopped

Vanishing Gradients in Next Word Prediction

- **LM task:** *When I tried to design my final exam, I found out that the students have no clue about LLMs. So I went to the scratchbook and started designing the coursework chapter by chapter. But even I realized that even I did not know about LLMs. Then how to design the* ___

- To learn from this training example, the RNN-LM needs to model the dependency between *"exam"* on the 7th step and the target word *"exam"* at the end.

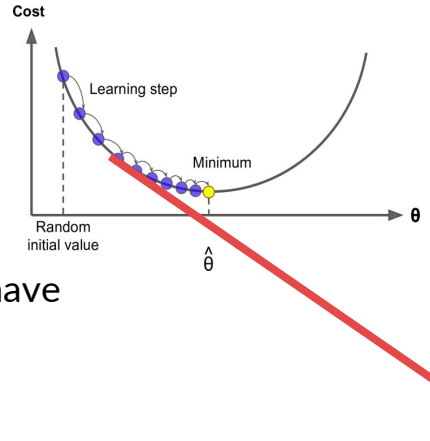
- But if the gradient is small, the model can't learn this dependency
  - So, the model is unable to predict similar long-distance dependencies at test time

Why is Exploding Gradients a Problem

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

- This can cause *bad updates*: we take too large a step and reach a weird and bad parameter configuration (with large loss)
  - You think you've found bukit timah to climb, but suddenly you're in JB



- In the worst case, this will result in *Inf* or *NaN* in your network (then you have to restart training from an earlier checkpoint)

Exploding Gradients can be solved – Gradient Clipping

- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

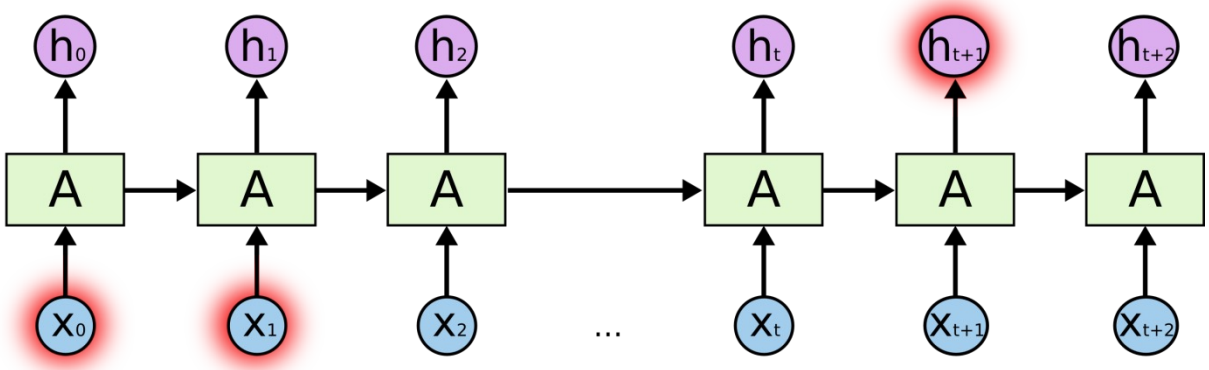
Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

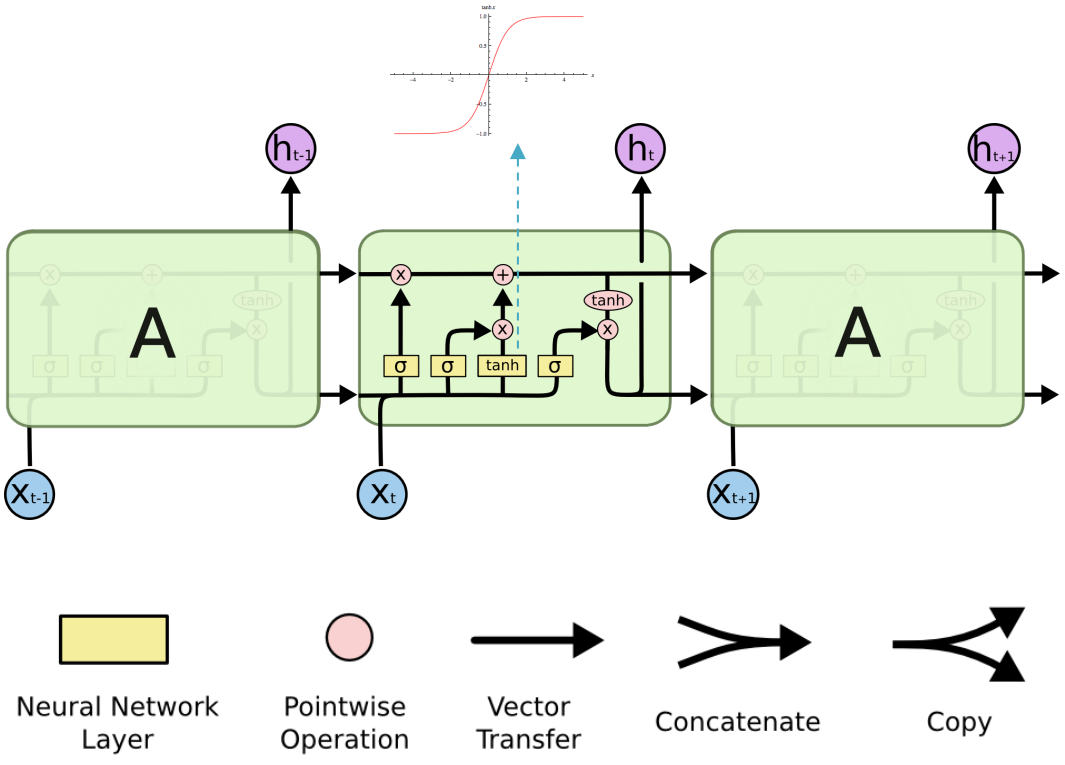
- **Intuition: take a step in the same direction, but a smaller step**
- **In practice, remembering to clip gradients is important, but exploding gradients are an easy problem to solve**

Vanishing Gradient still a problem: Long Range Dependencies

- It is very difficult to train RNNs to retain information over many time steps
- This make is very difficult to learn RNNs that handle long-distance dependencies, which may exist in long sentences.



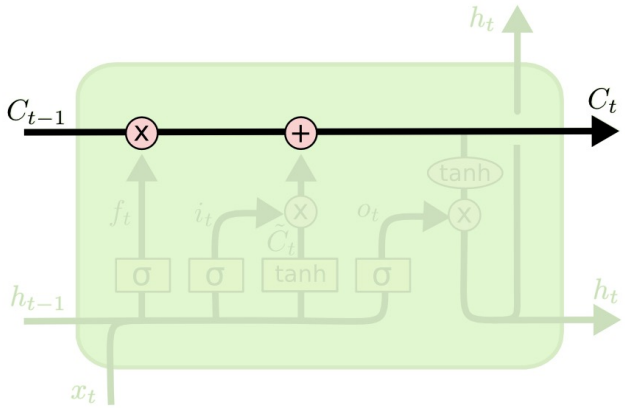
RNN evolution: LSTM (Long Short Term Memory)



- LSTM networks, add additional gating units in each memory cell.
  - Forget gate
  - Input gate
  - Output gate
- Prevents vanishing/exploding gradient problem and allows network to retain state information over longer periods of time.

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

Cell State

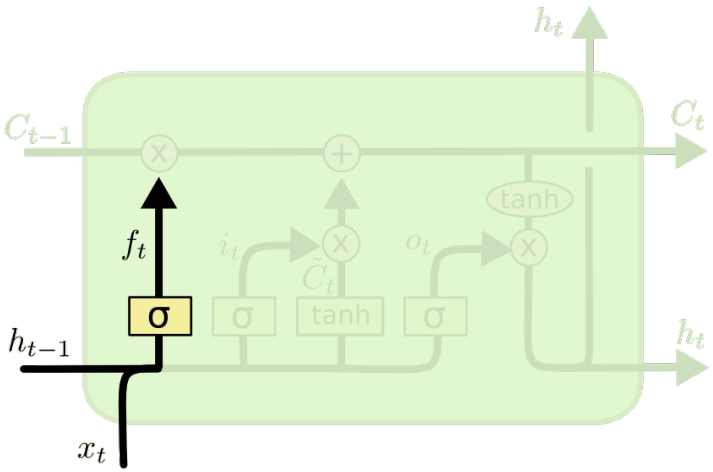


- Additional vector C_t that is the same dimensionality as the hidden state, h_t
- Information can be added or deleted from this state vector via the forget and input gates.

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

Forget Gate

- This gate decides what information should be thrown away or kept.
- Information from the previous hidden state and information from the current input is passed through the sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

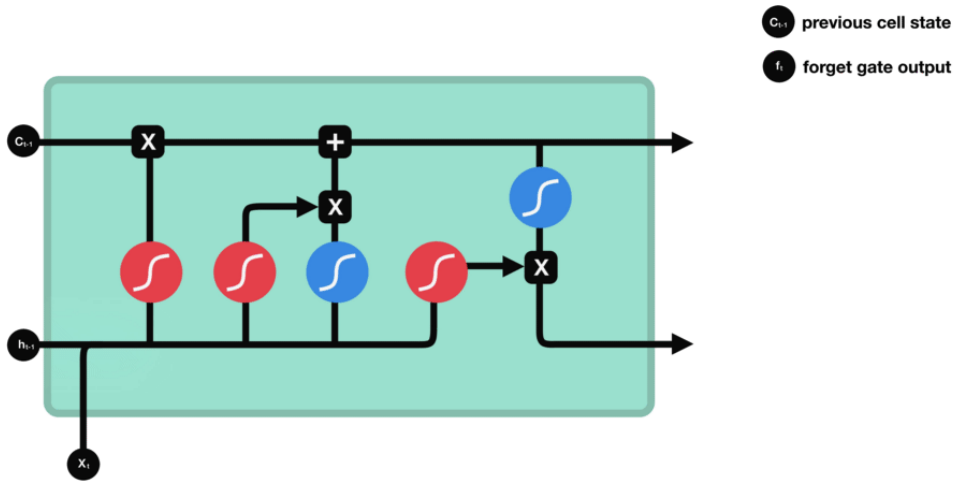


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

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

Forget Gate

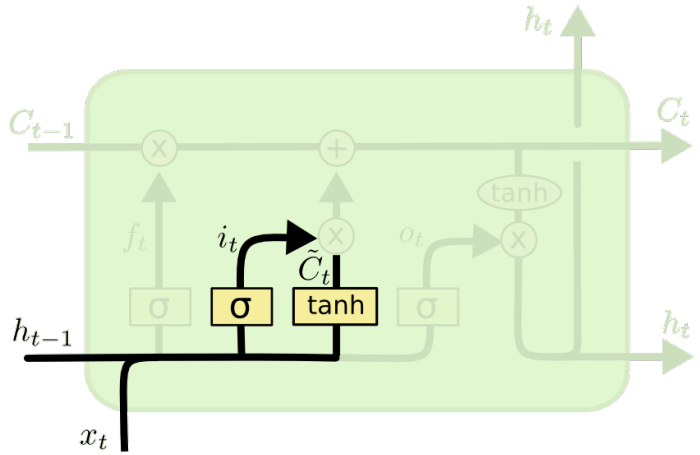
- This gate decides what information should be thrown away or kept.
- Information from the previous hidden state and information from the current input is passed through the sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.



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

Input Gate

- Input Gate is used to update the cell state
- pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network.
- pass the previous hidden state and current input into a sigmoid function to decide which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important.
- sigmoid output decides which information is important to keep from the tanh output.



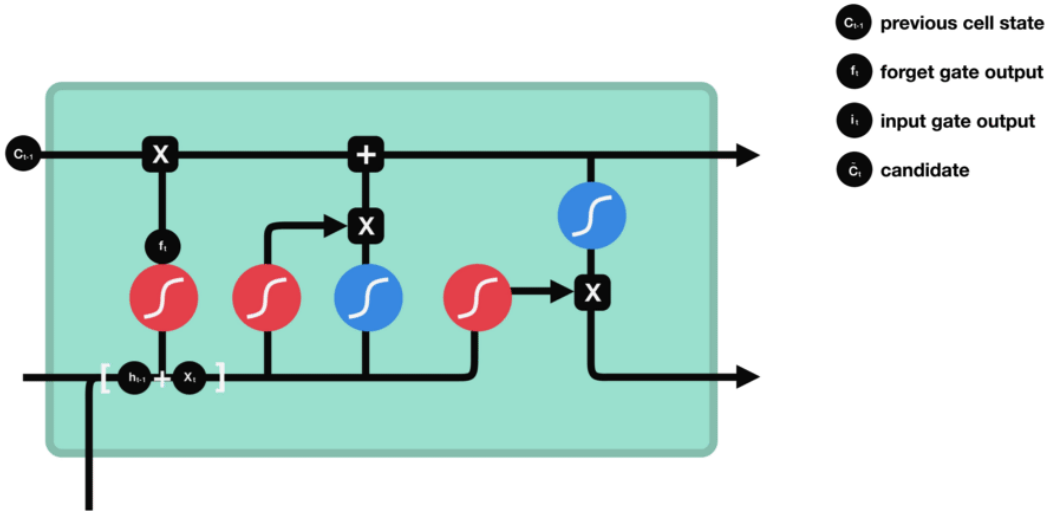
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

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

Input Gate

- Input Gate is used to update the cell state
- pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network.
- pass the previous hidden state and current input into a sigmoid function to decide which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important.
- sigmoid output decides which information is important to keep from the tanh output.

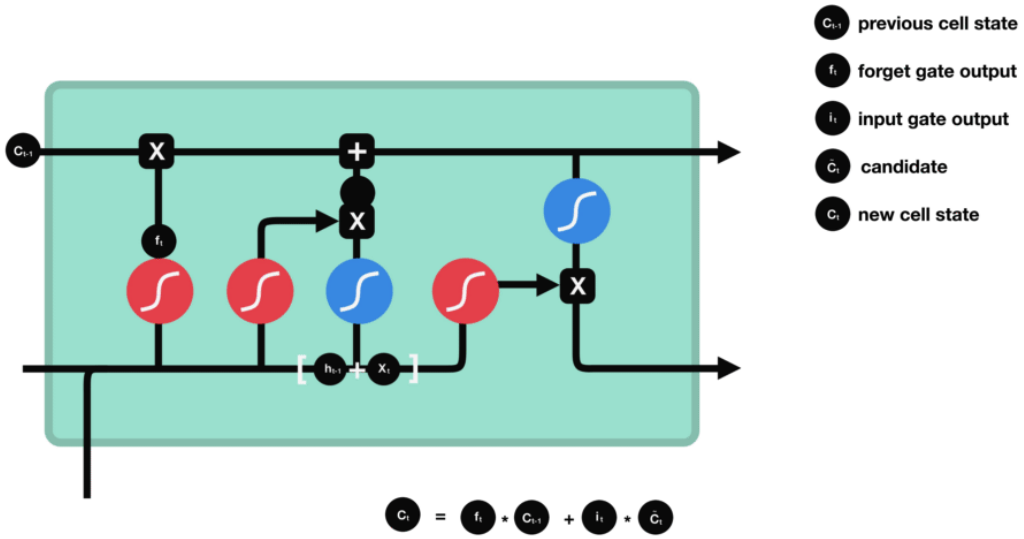


Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Cell State update

- The cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0.

- Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant.

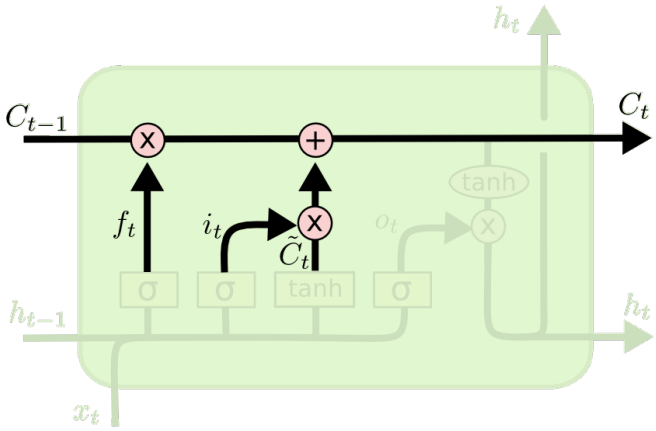


Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Cell State update

- The cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0.

- Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant.

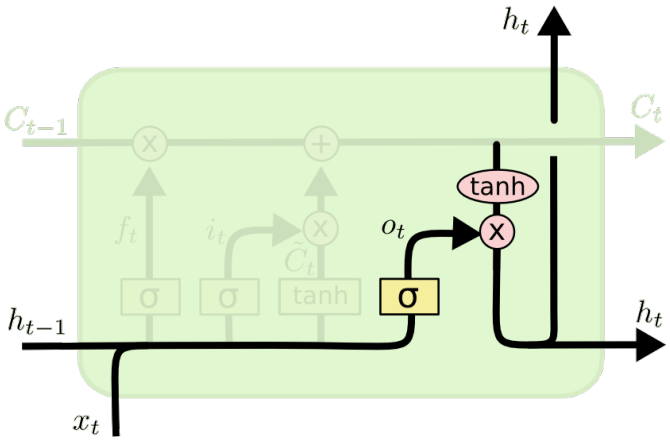


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

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

Output Gate

- The output gate decides what the next hidden state should be.
- First, we pass the previous hidden state and the current input into a sigmoid function.
- Then we pass the newly modified cell state to the tanh function.
- We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry.
- The new cell state and the new hidden is then carried over to the next time step.



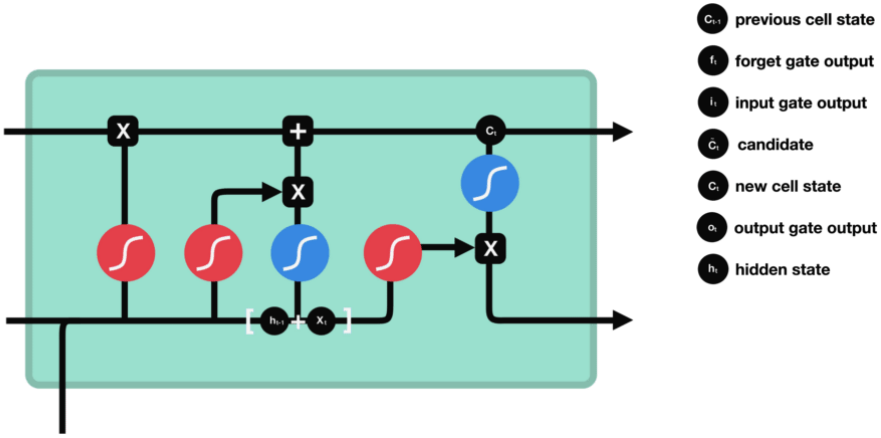
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

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

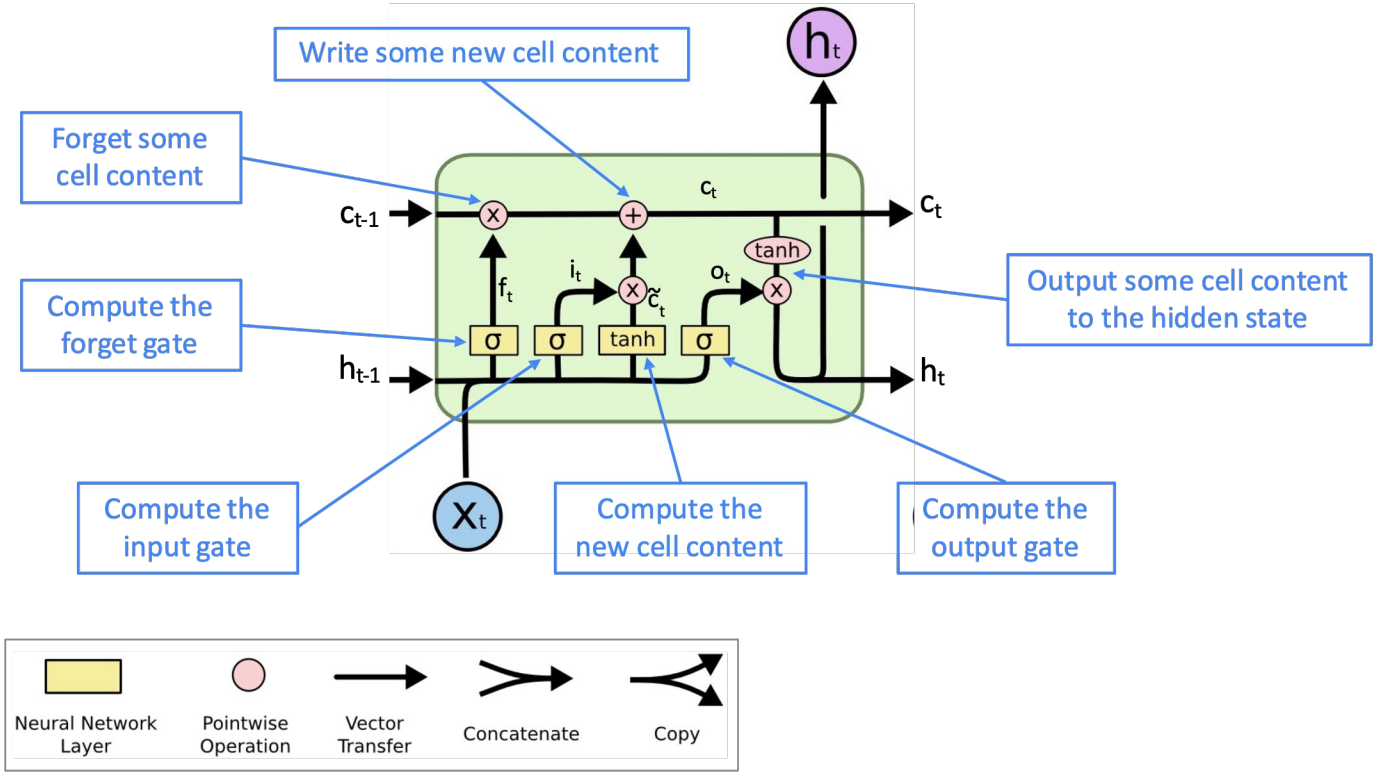
Output Gate

- The output gate decides what the next hidden state should be.
- First, we pass the previous hidden state and the current input into a sigmoid function.
- Then we pass the newly modified cell state to the tanh function.
- We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry.
- The new cell state and the new hidden is then carried over to the next time step.



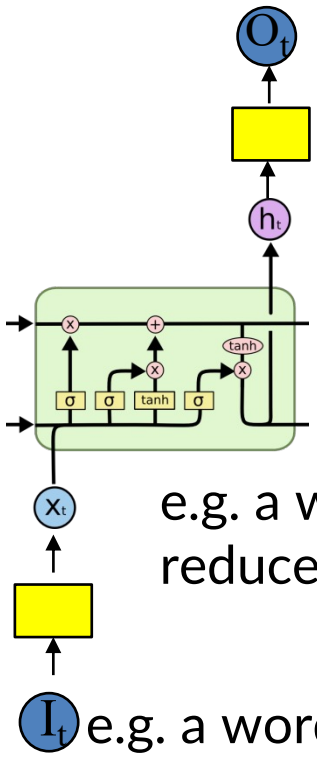
Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Overall Architecture



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

Overall Architecture



e.g. next word

e.g. a word "embedding" with reduced dimensionality

e.g. a word as a "one hot" vector

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

LSTM Training

- Gradient Descent Algorithms:
  - Stochastic gradient descent (randomize order of examples in each epoch) with momentum (bias weight changes to continue in same direction as last update).
  - ADAM optimizer (Kingma & Ma, 2015)

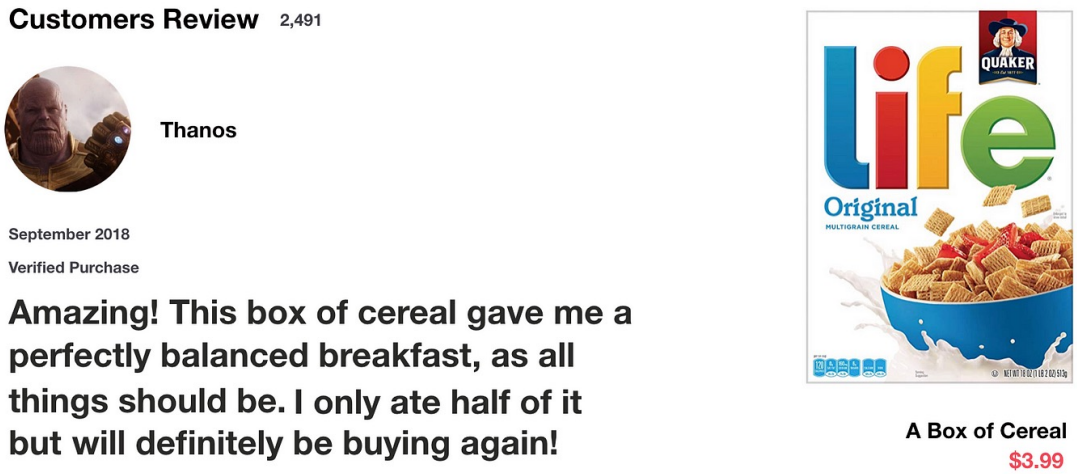
- Each cell has many parameters (W_f, W_i, W_C, W_o)
  - Training LSTMs requires lots of training data.
  - Requires lots of compute time – definitely need GPUs.

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

How LSTMs Help

The gates in LSTMs can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions.

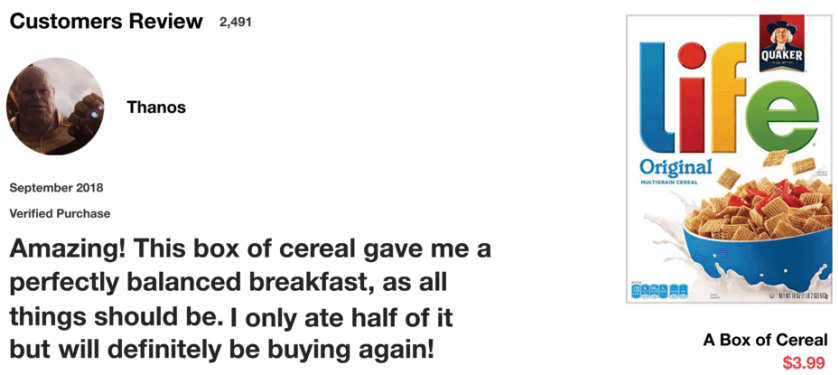
Intuition



How LSTMs Help

The gates in LSTMs can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions.

Intuition



pick up words like “amazing” and “perfectly balanced breakfast”.

Ignore words like “this”, “gave“, “all”, “should”, etc.

How LSTMs Help

- The LSTM architecture makes it much easier for an RNN to preserve information over many timesteps
  - e.g., if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely.
  - In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix W_h that preserves info in the hidden state

- However, there are alternative ways of creating more direct and linear pass-through connections in models for long distance dependencies

Vanishing Gradient is not just RNN problems

- It can be a problem for all neural architectures, especially very deep neural networks with many layers.
  - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
  - Thus, lower layers are learned very slowly (i.e., are hard to train)
- Another solution: lots of new deep feedforward/convolutional architectures add more direct connections (thus allowing the gradient to flow)

- Residual connections aka "ResNet"

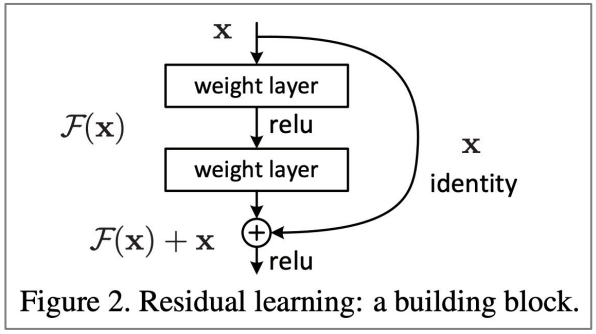


Figure 2. Residual learning: a building block.

- Dense connections aka "DenseNet"
- Directly connect each layer to all future layers!

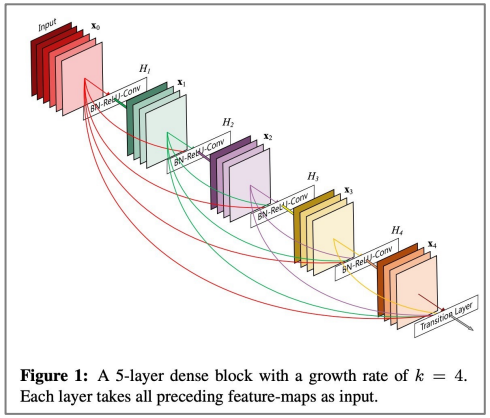
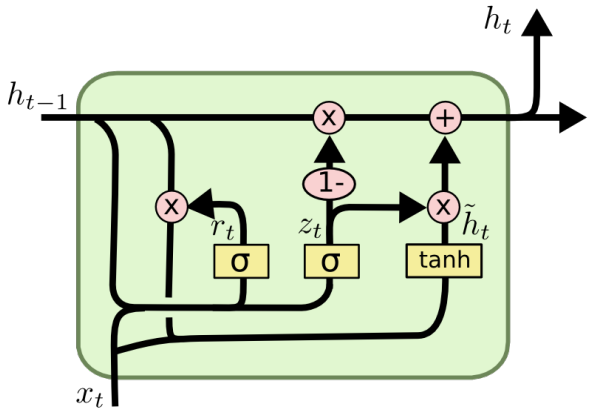


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

"Deep Residual Learning for Image Recognition", He et al, 2015. https://arxiv.org/pdf/1512.03385.pdf
"Densely Connected Convolutional Networks", Huang et al, 2017. https://arxiv.org/pdf/1608.06993.pdf

GRUs – Alternative to LSTMs

- uses fewer gates (Cho, et al., 2014)
  - Combines forget and input gates into “update” gate.
  - Eliminates cell state vector



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTM Possibilities

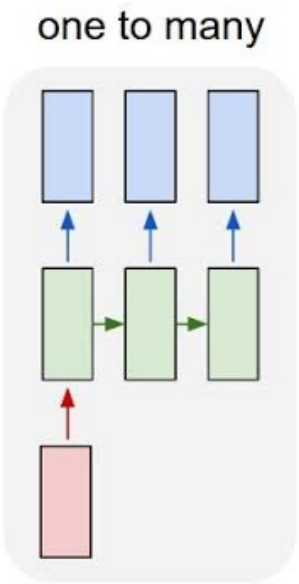
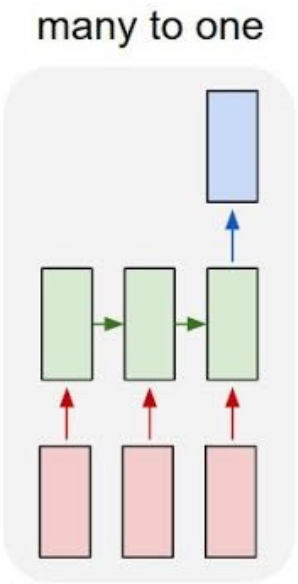
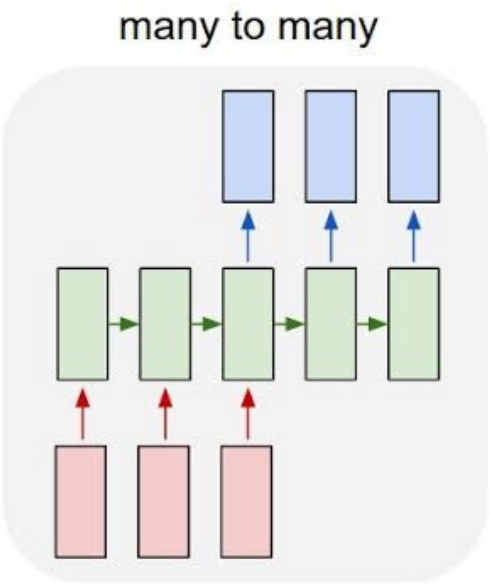


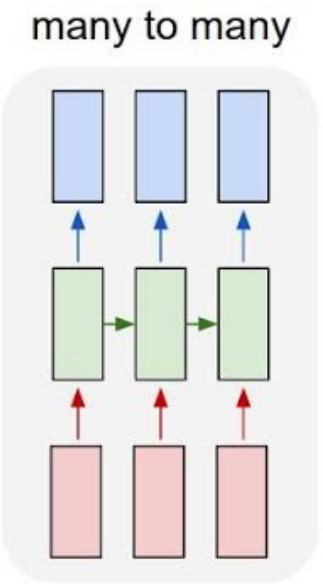
Image Captioning



Video Activity Recog
Text Classification



Video Captioning
Machine Translation

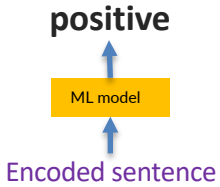


POS Tagging
Language Modeling

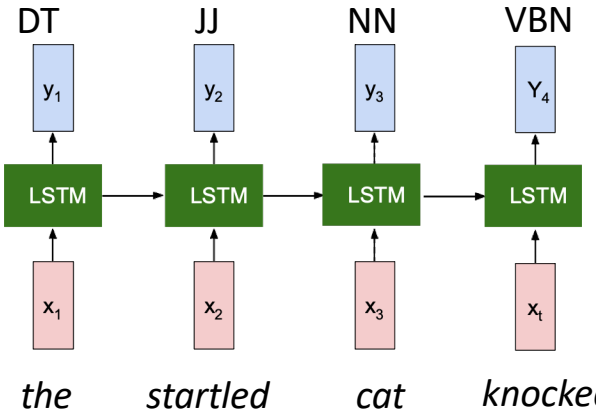
BREAK

Other LSTMs Applications

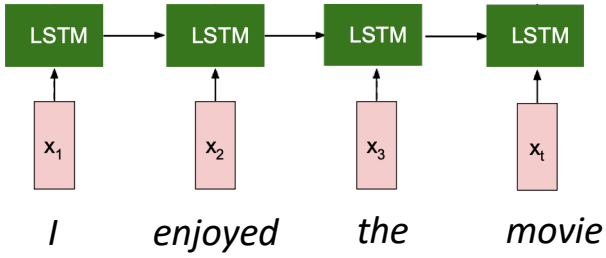
- In 2013–2015, LSTMs started achieving state-of-the-art results and became the dominant approach for most NLP tasks
- Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models



How to compute sentence encoding?

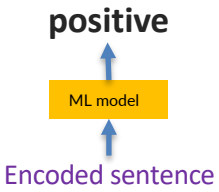


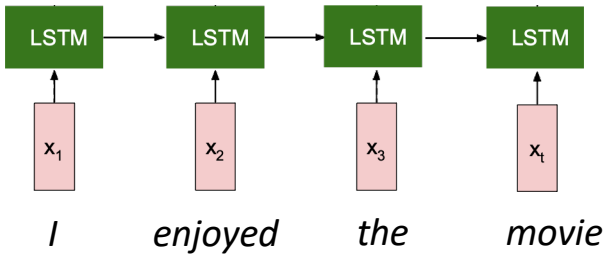
sequence tagging such as POS, NER etc



Sentence encoding such as for sentiment analysis

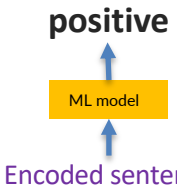
LSTMs for Sentence Encoding



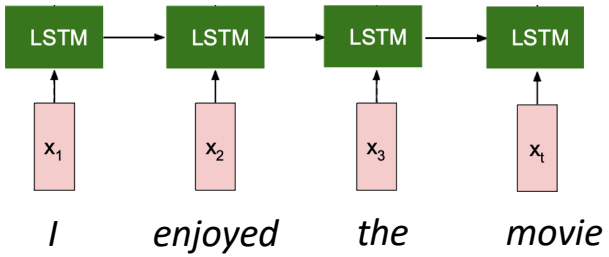


At a particular timestep t, only that word input **as well as** words which came before time t are used to create hidden state for that particular word at time t

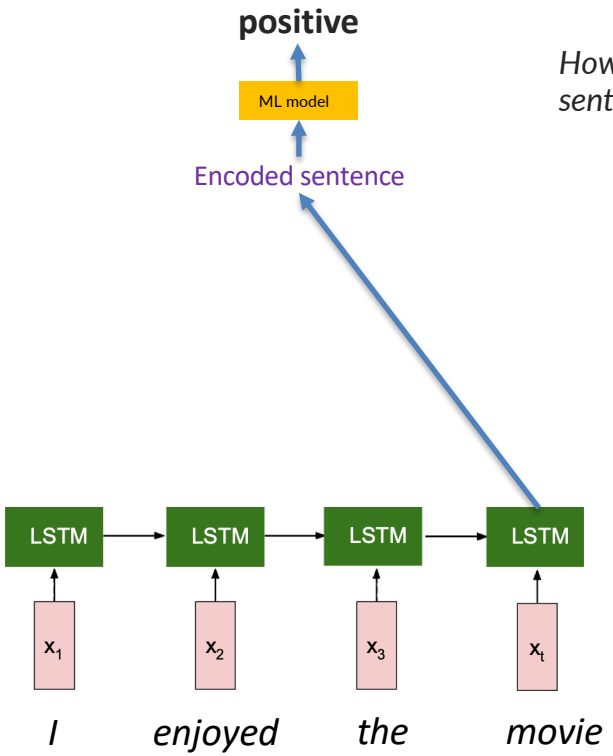
LSTMs for Sentence Encoding



How to compute sentence encoding?



LSTMs for Sentence Encoding

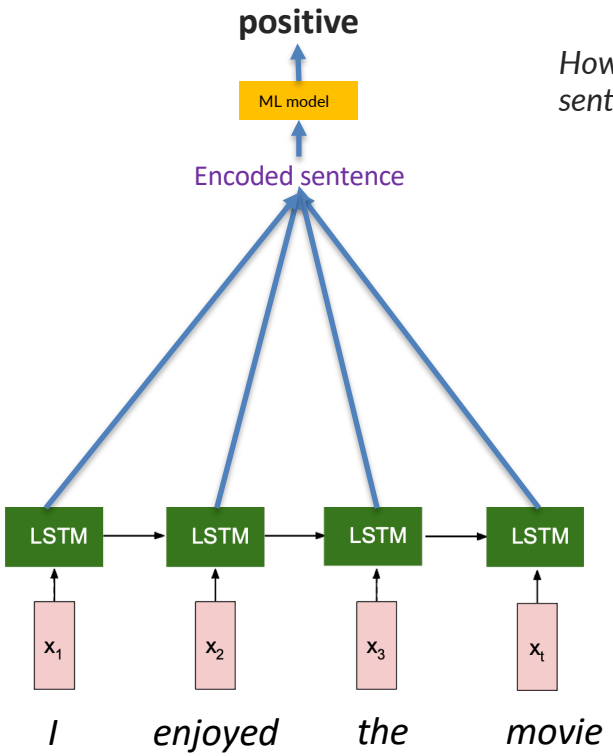


How to compute sentence encoding?

Basic way:

Use final hidden state

LSTMs for Sentence Encoding

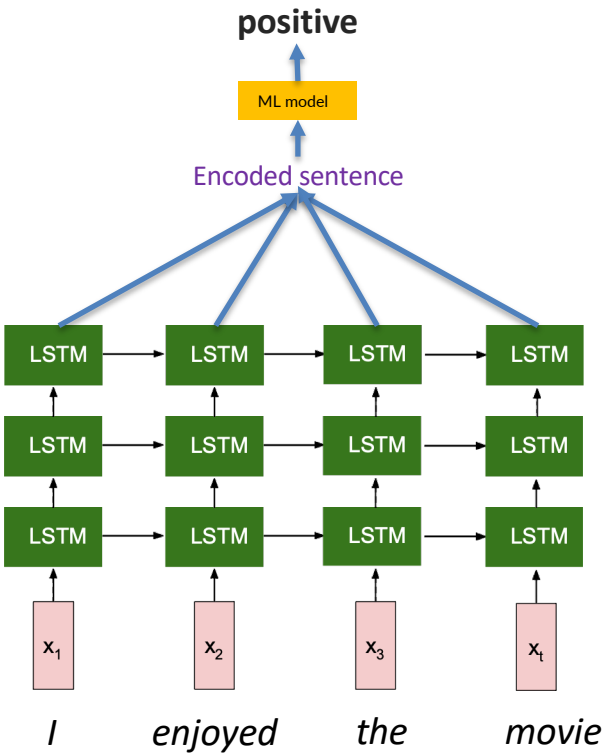


How to compute sentence encoding?

Better way:

Take element-wise max or mean of all hidden states

Deep LSTM models



LSTM for Sentence Encoding AND Decoding Machine Translation

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

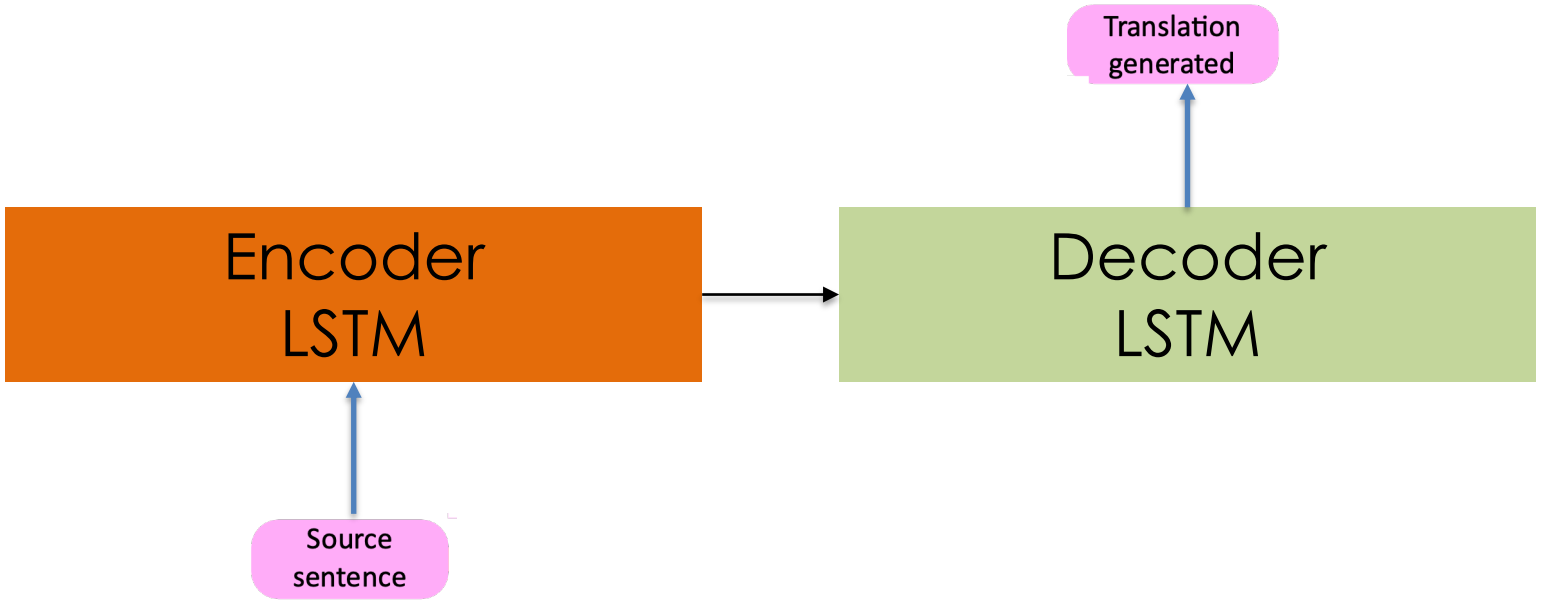
x: *L'homme est né libre, et partout il est dans les fers*

y: *Man is born free, but everywhere he is in chains*

1990s-2010s: Statistical Machine Translation

2014: Neural Machine Translation

LSTM for Sentence Encoding and Decoding - Machine Translation

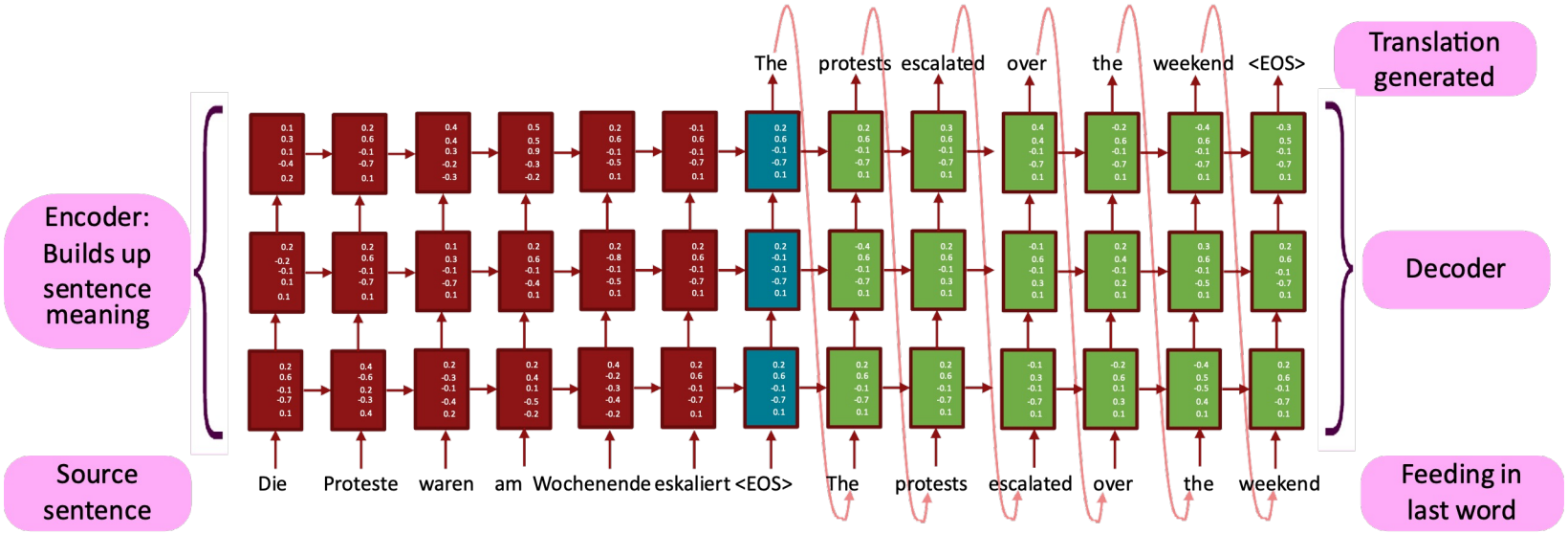


This architecture is also referred to as sequence to sequence, or seq2seq models

Some other examples of seq2seq models

- Summarization
- Question answering

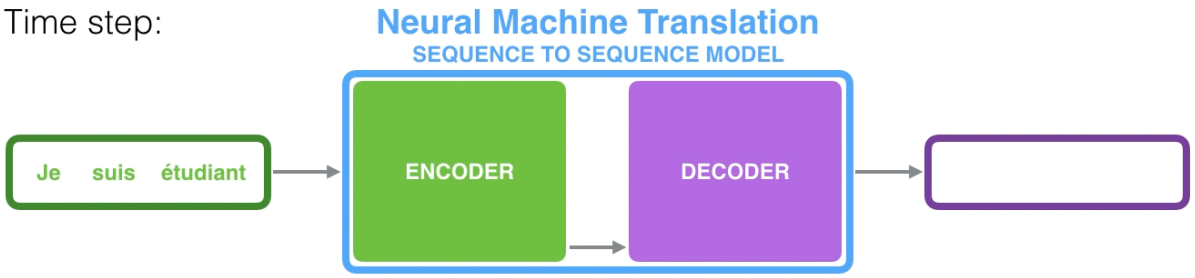
LSTM for Sentence Encoding and Decoding - Machine Translation



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

LSTM for Machine Translation – how it works

The encoder processes each word in the input sentence, it compiles the information it captures into a vector (called the context).
After processing the entire input sequence, the encoder sends the context over to the decoder, which begins producing the output sentence word by word.

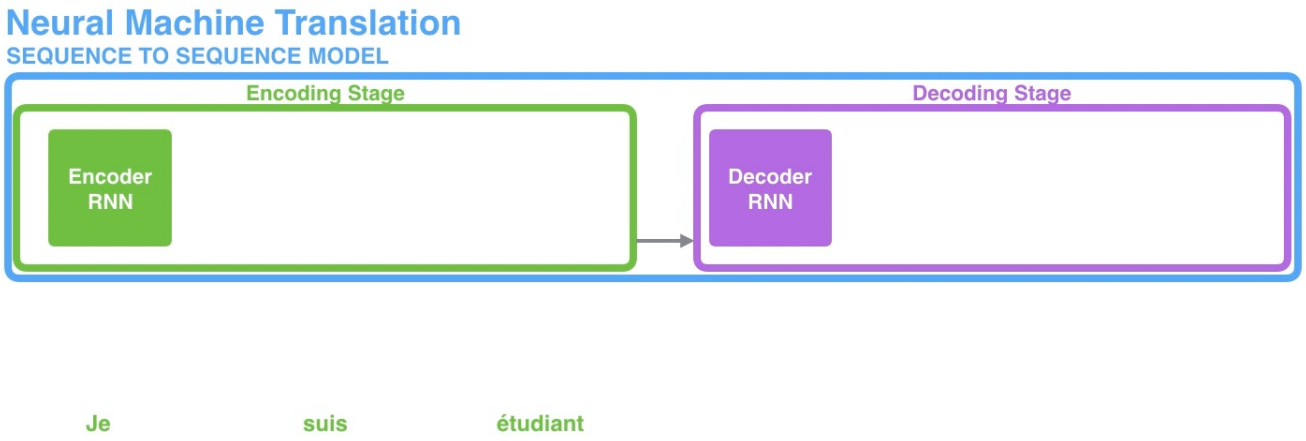


The context is a vector (an array of numbers, basically)

https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

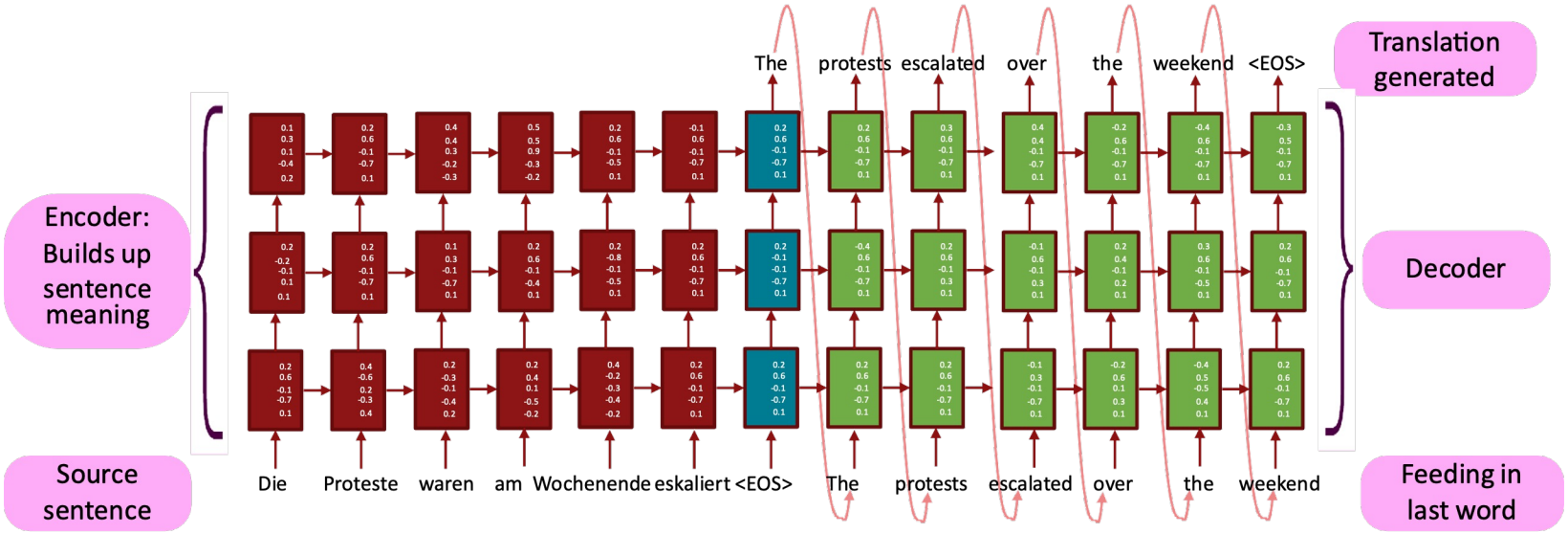
LSTM for Machine Translation – how it works

"unrolled" view



https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

LSTM for Sentence Encoding and Decoding - Machine Translation

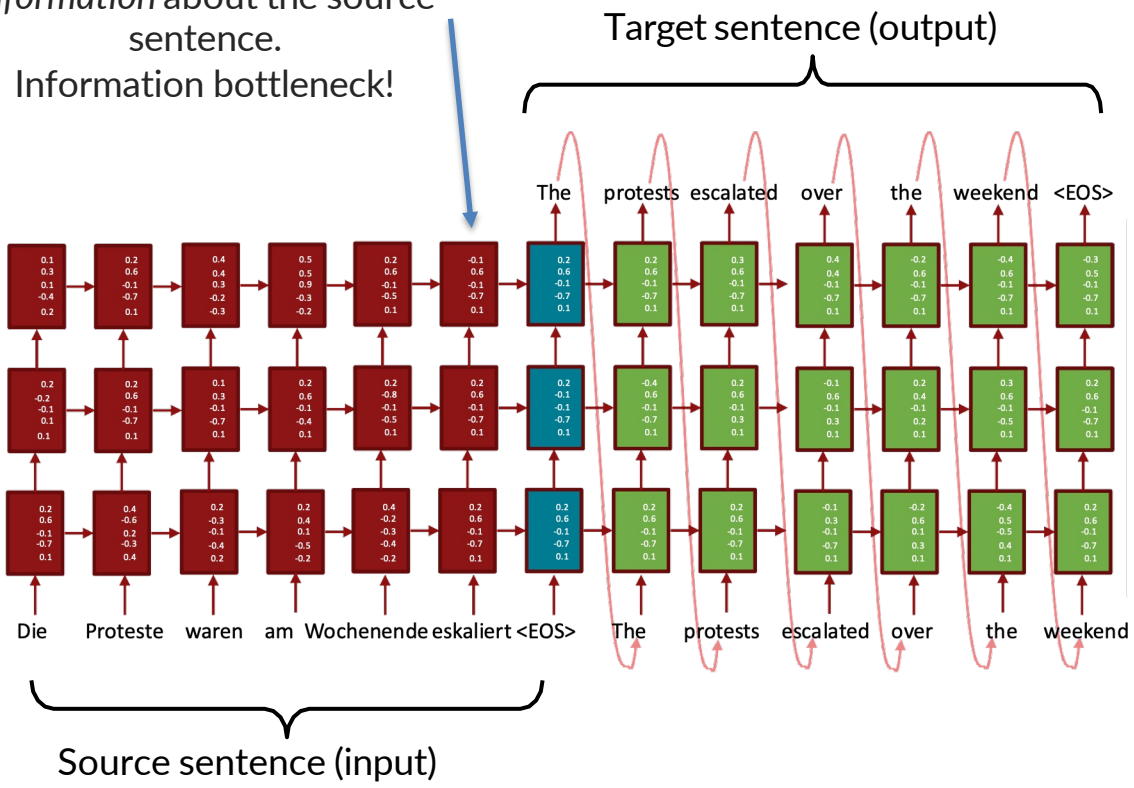


The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1

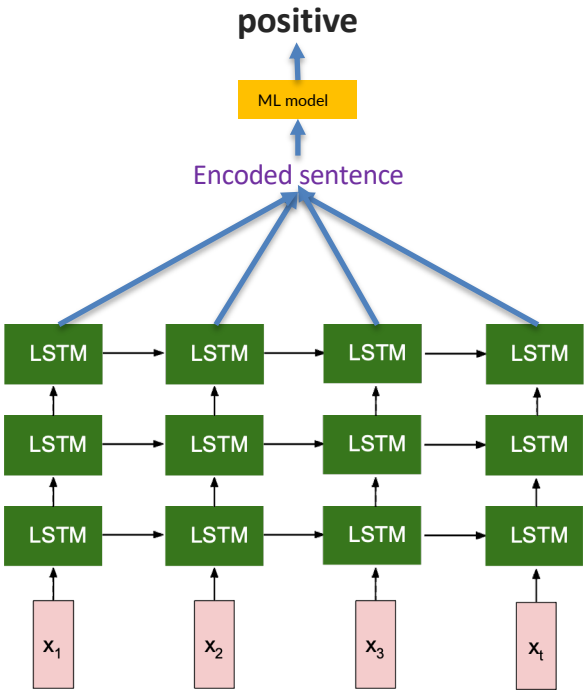
Issue with the Architecture

Encoding of the source sentence.

This needs to capture *all information* about the source sentence.
Information bottleneck!



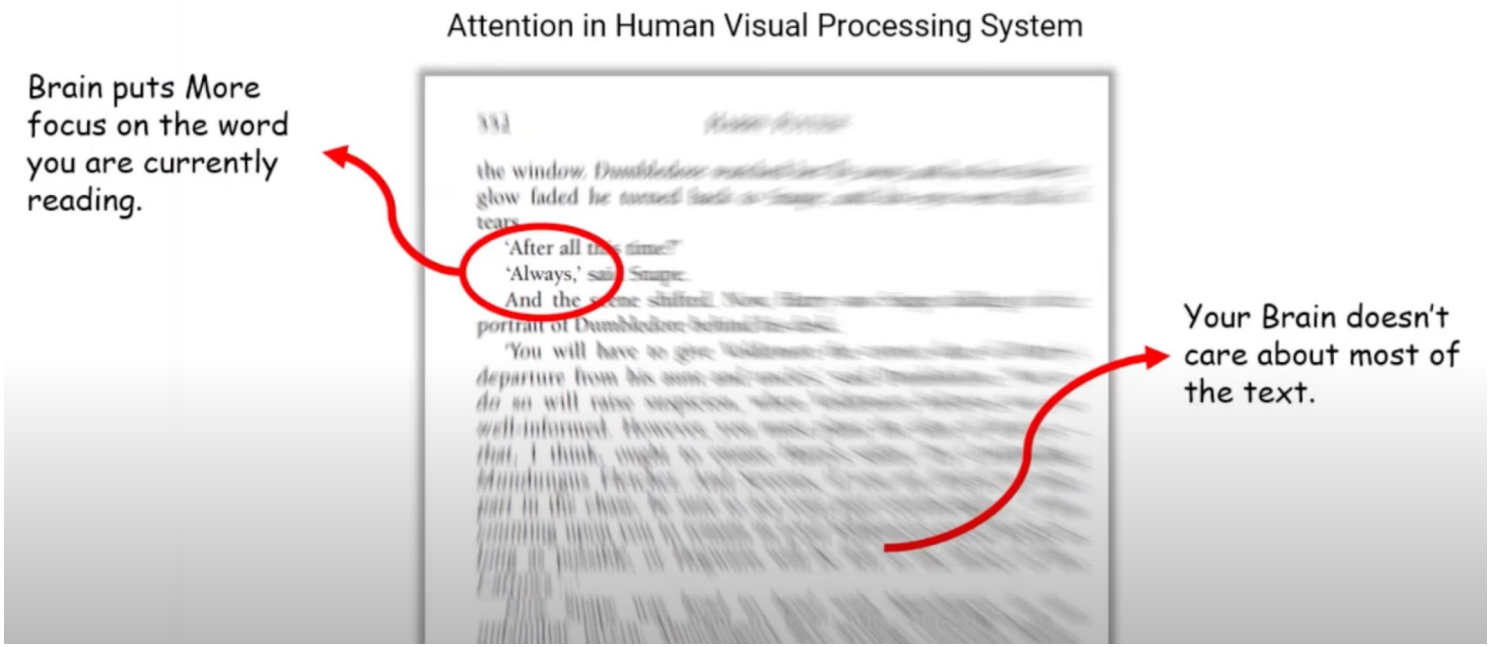
Issue with the Architecture



One option is to use average/max pooling of all hidden states

However we loose of information at individual word level

Solution - Attention



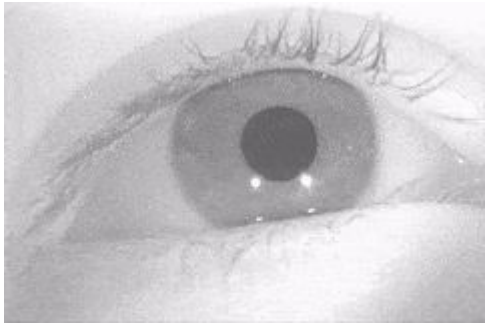
Solution - Attention

A solution was proposed in Bahdanau et al., 2014 and Luong et al., 2015.

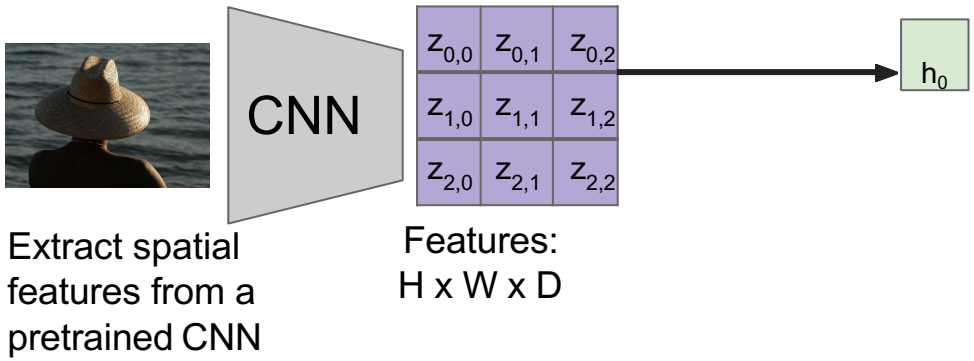
Inspired from computer vision

Attention idea: New context vector at every time step.

Each context vector will attend to different image regions

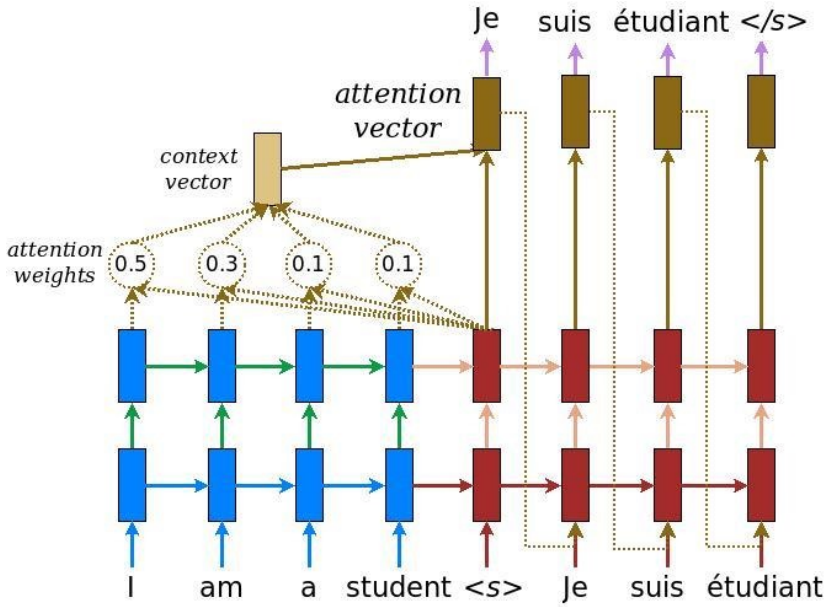


Attention Saccades in humans



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Attention is weighted averaging of encoded sentence



- compare target and source states to generate scores for each state in encoders.
- use softmax to normalize all scores
- Attention weight parameters are introduced to train context vector

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

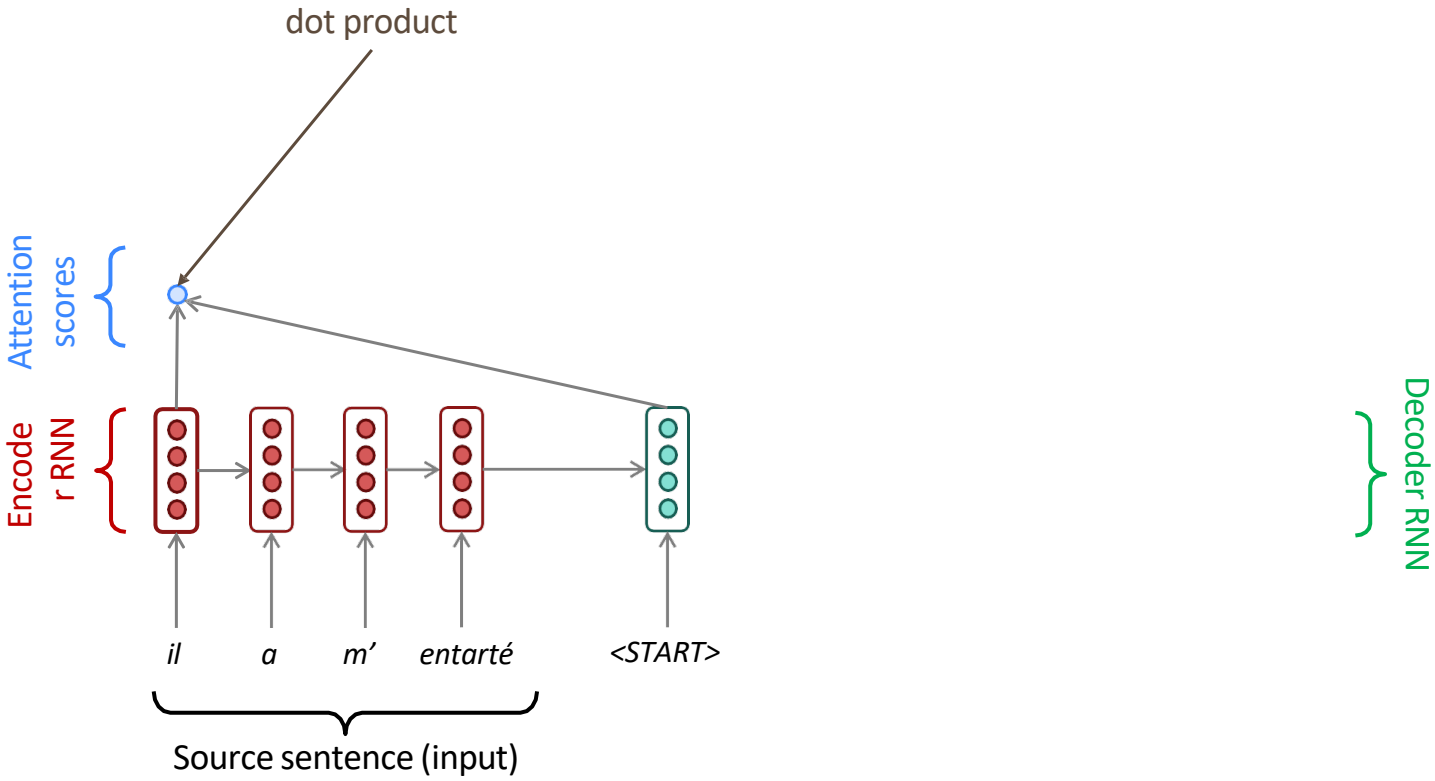
Attention is weighted averaging of encoded sentence

Attention at time step 4

https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

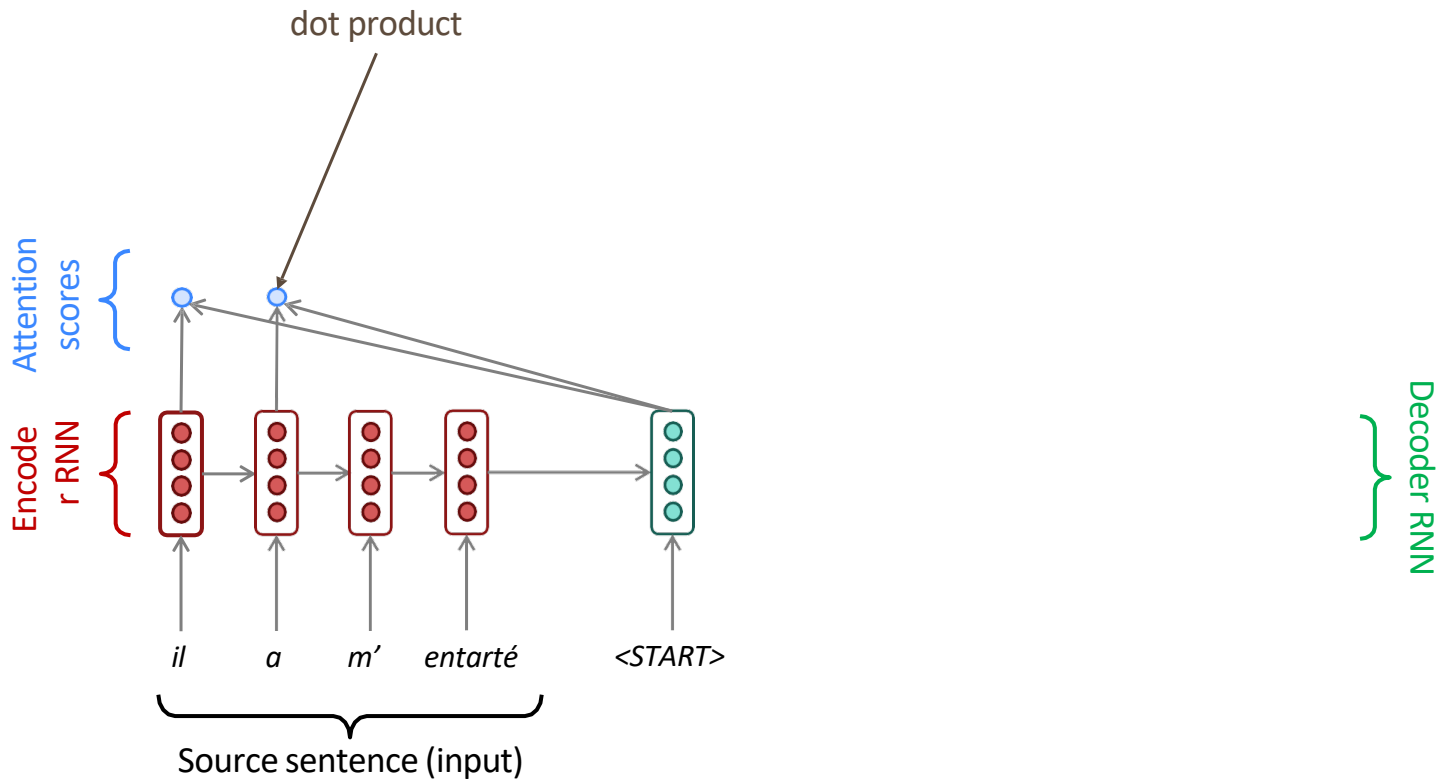
Attention for every time step

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



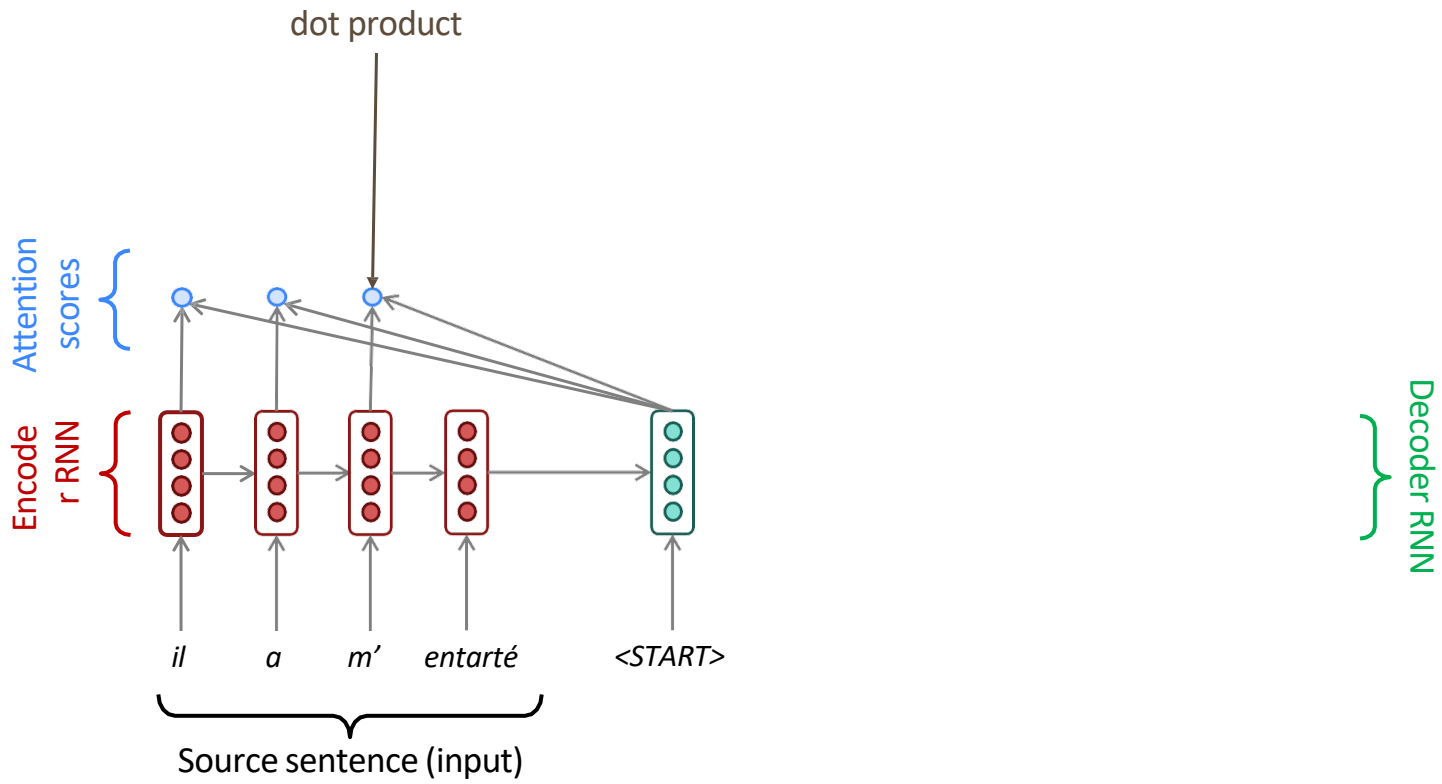
Attention for every time step

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



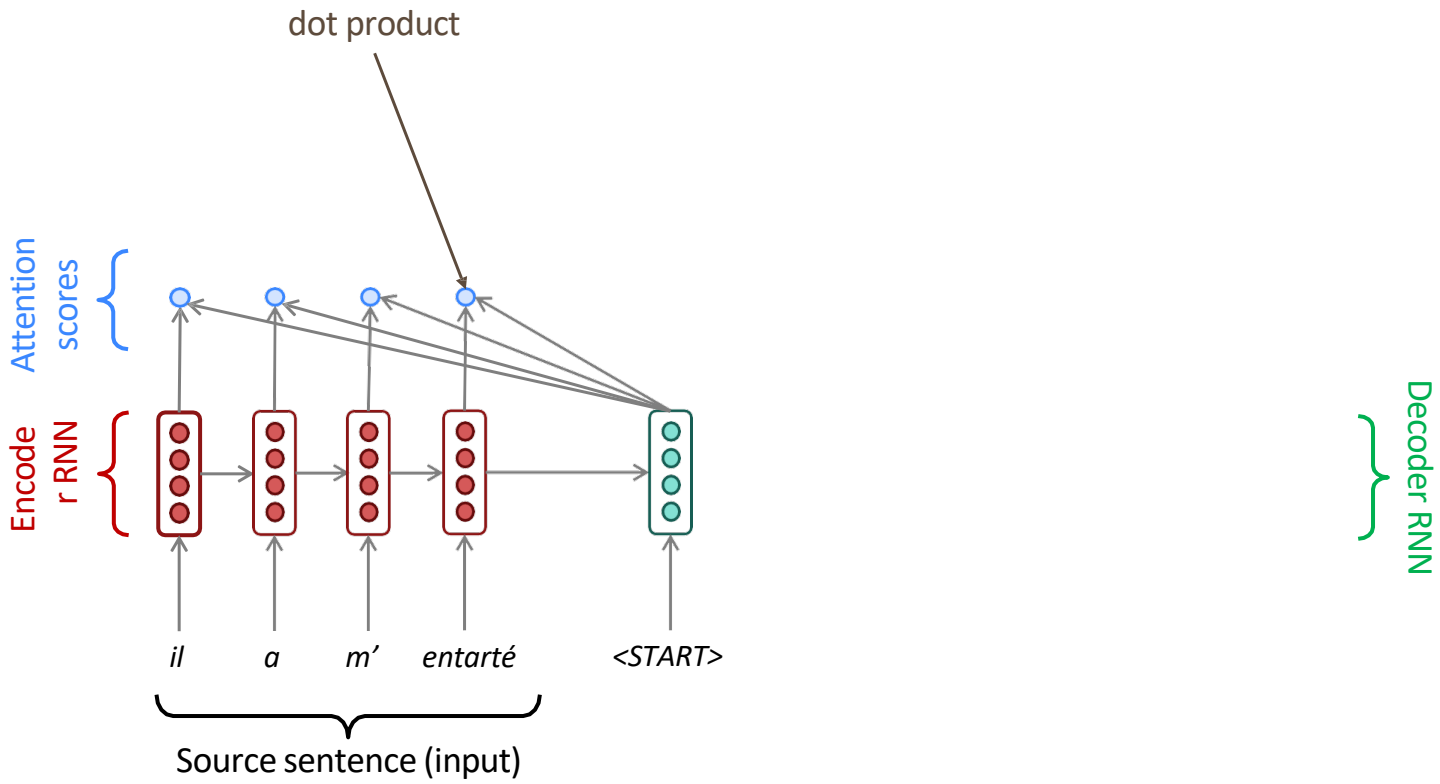
Attention for every time step

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



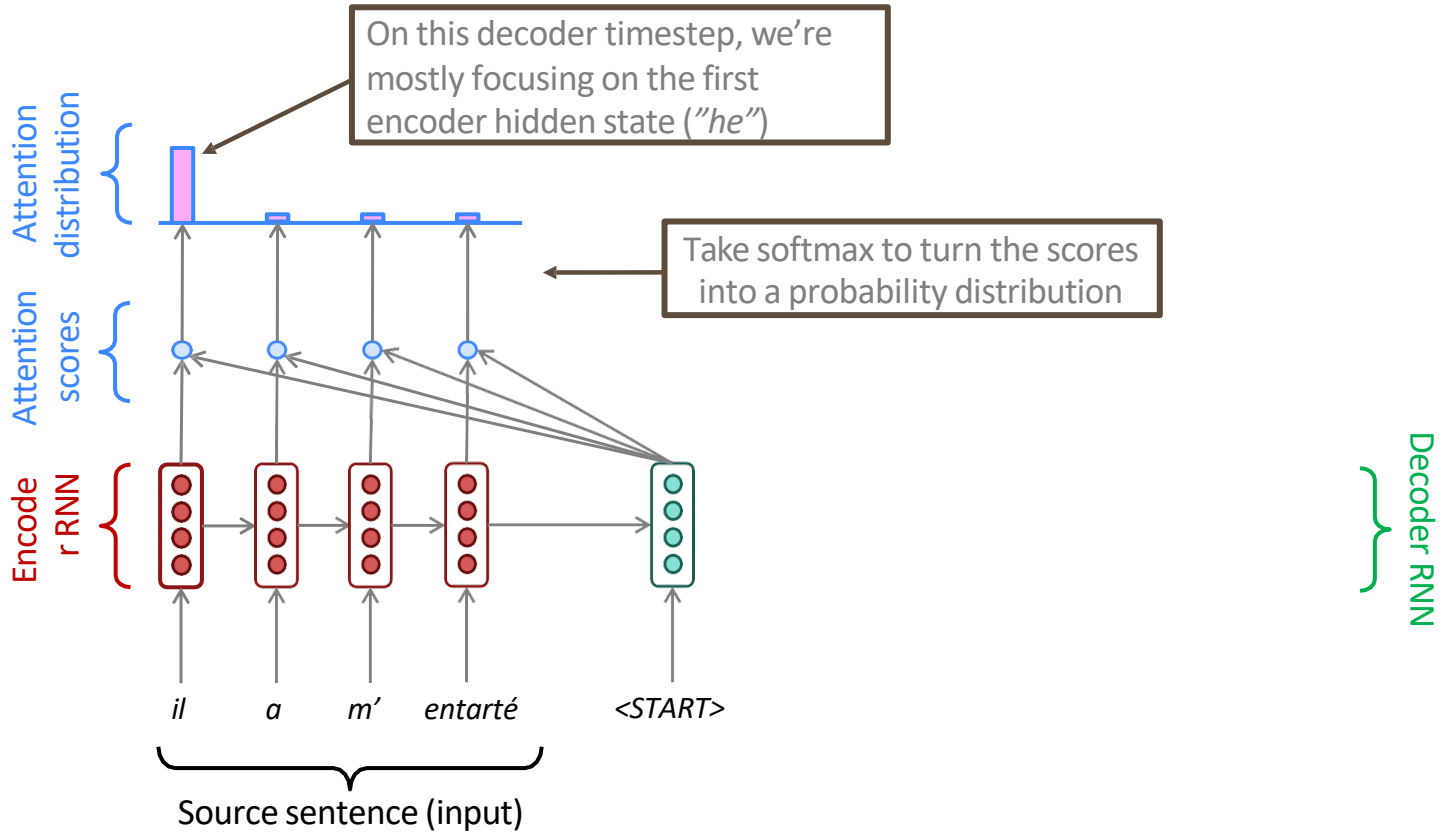
Attention for every time step

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



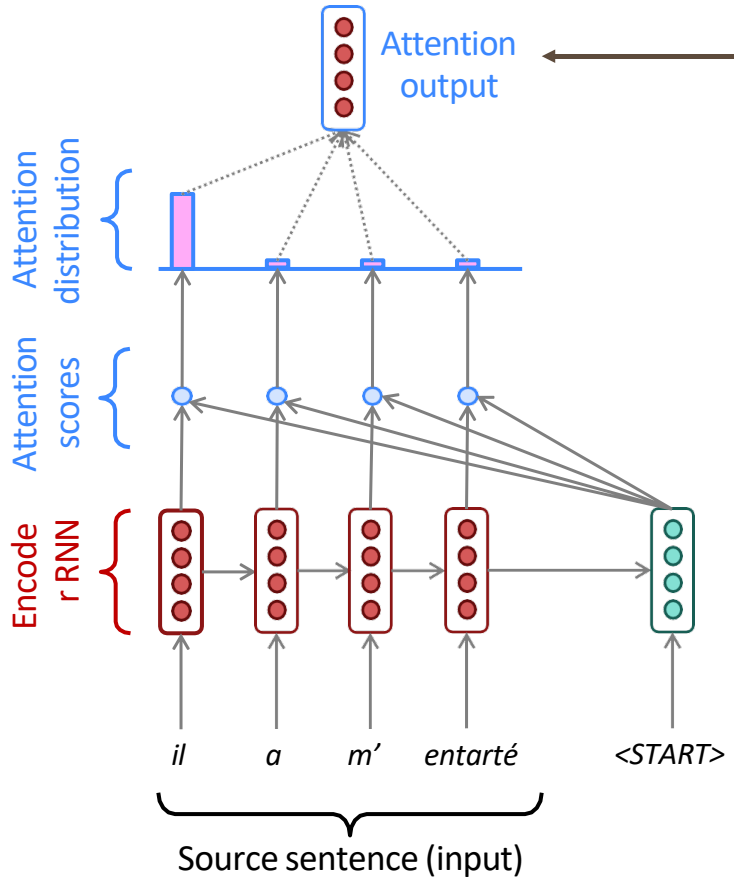
Attention for every time step

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



Attention for every time step

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



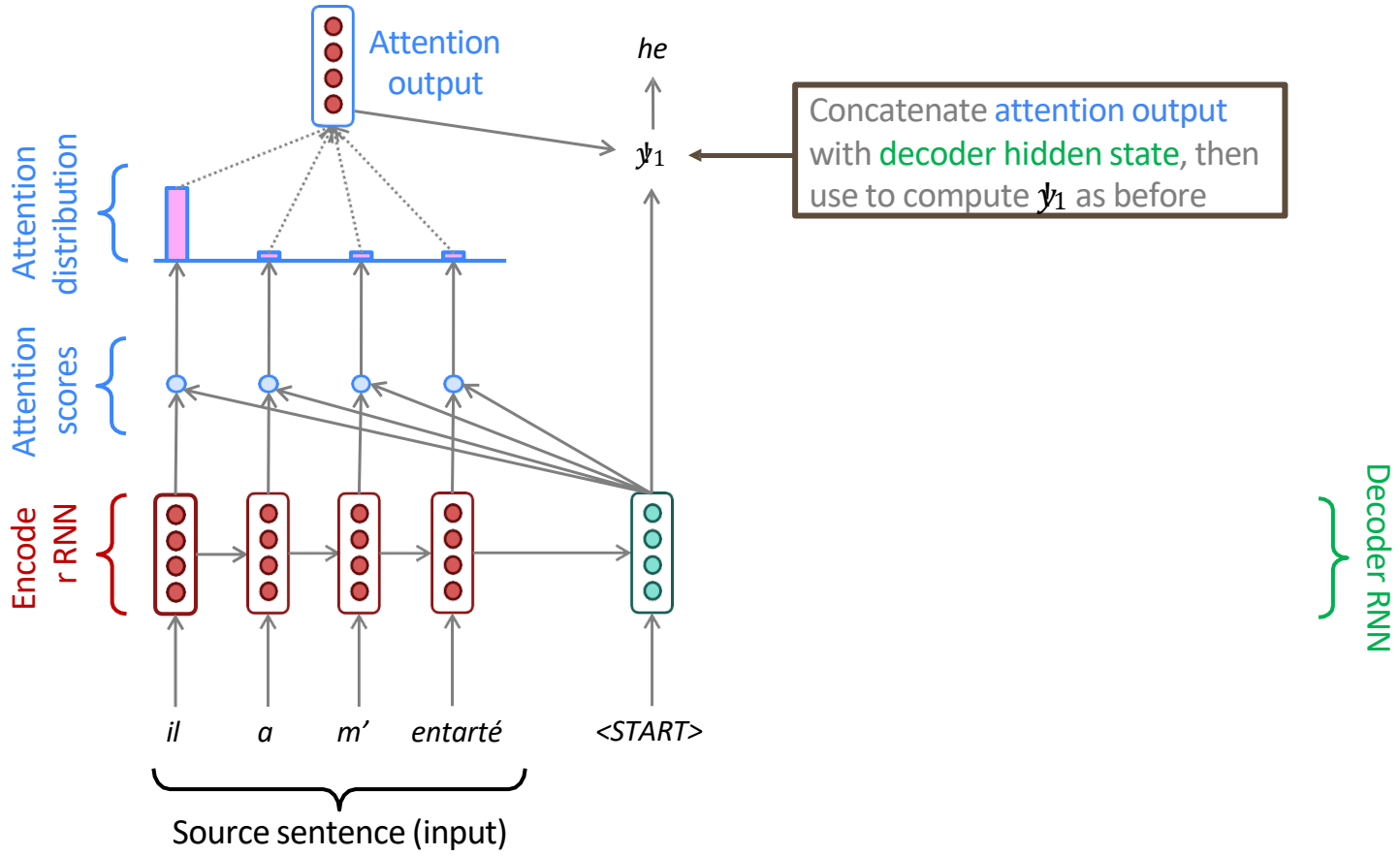
Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.

Decoder RNN

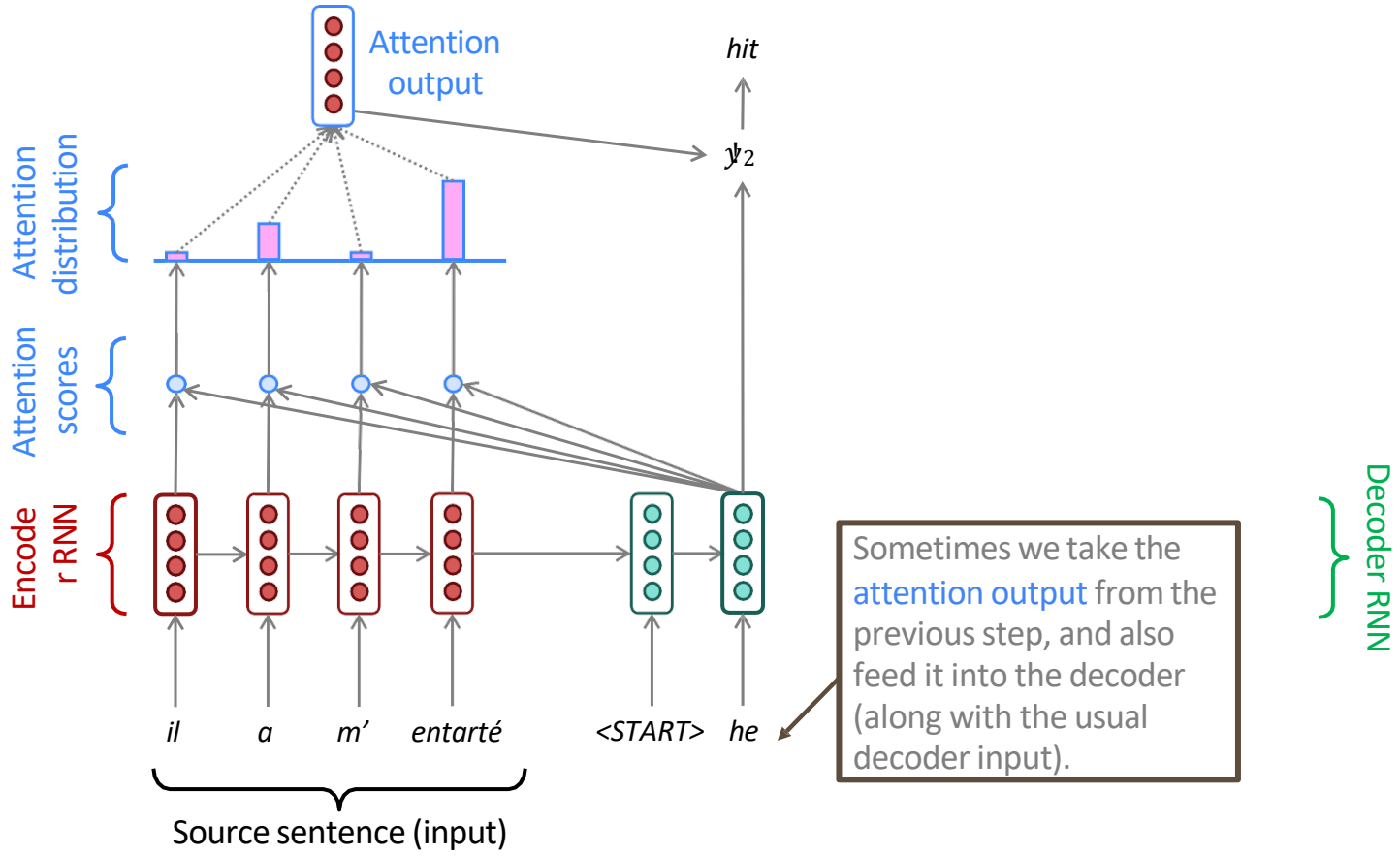
Attention for every time step

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



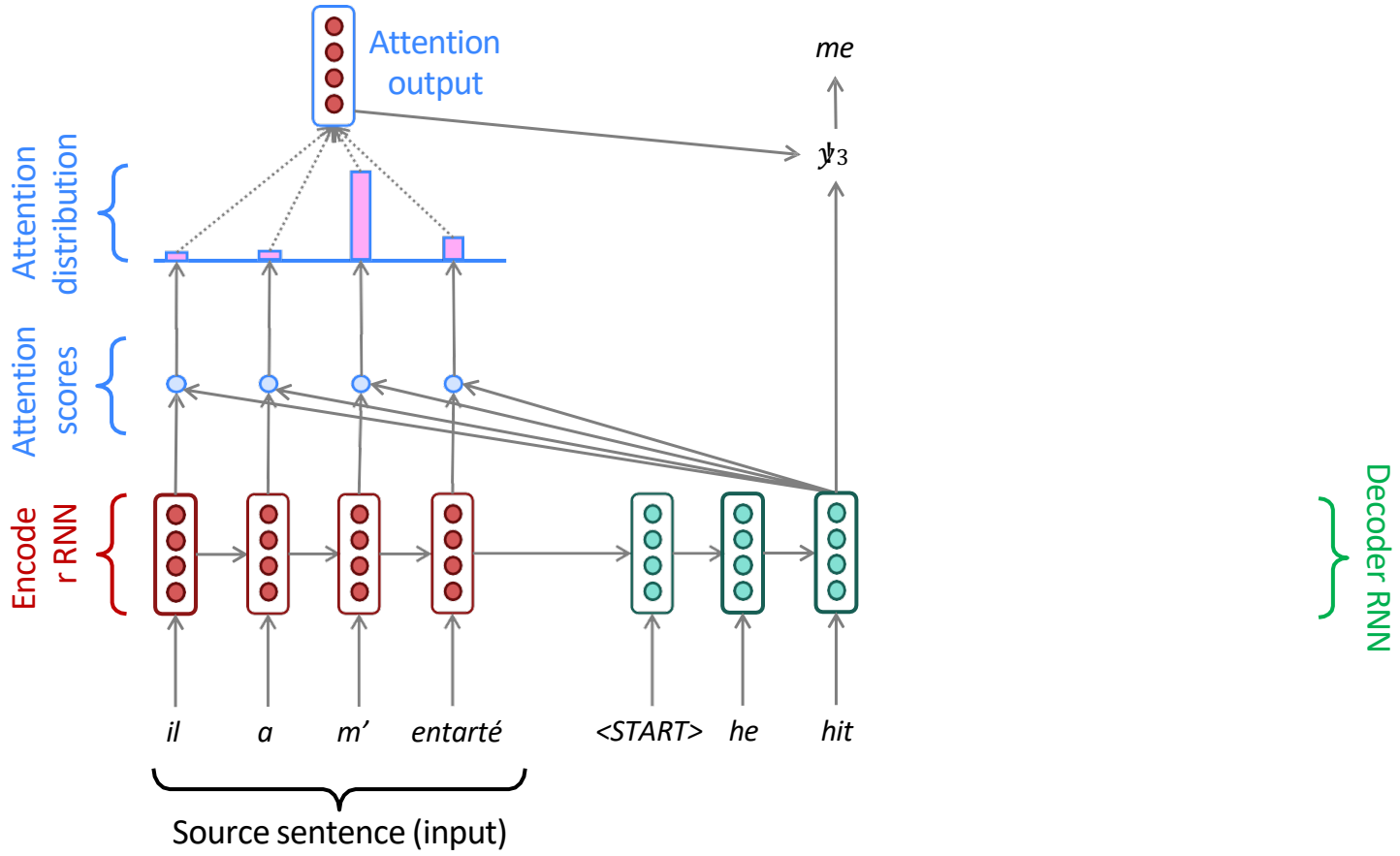
Attention for every time step

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



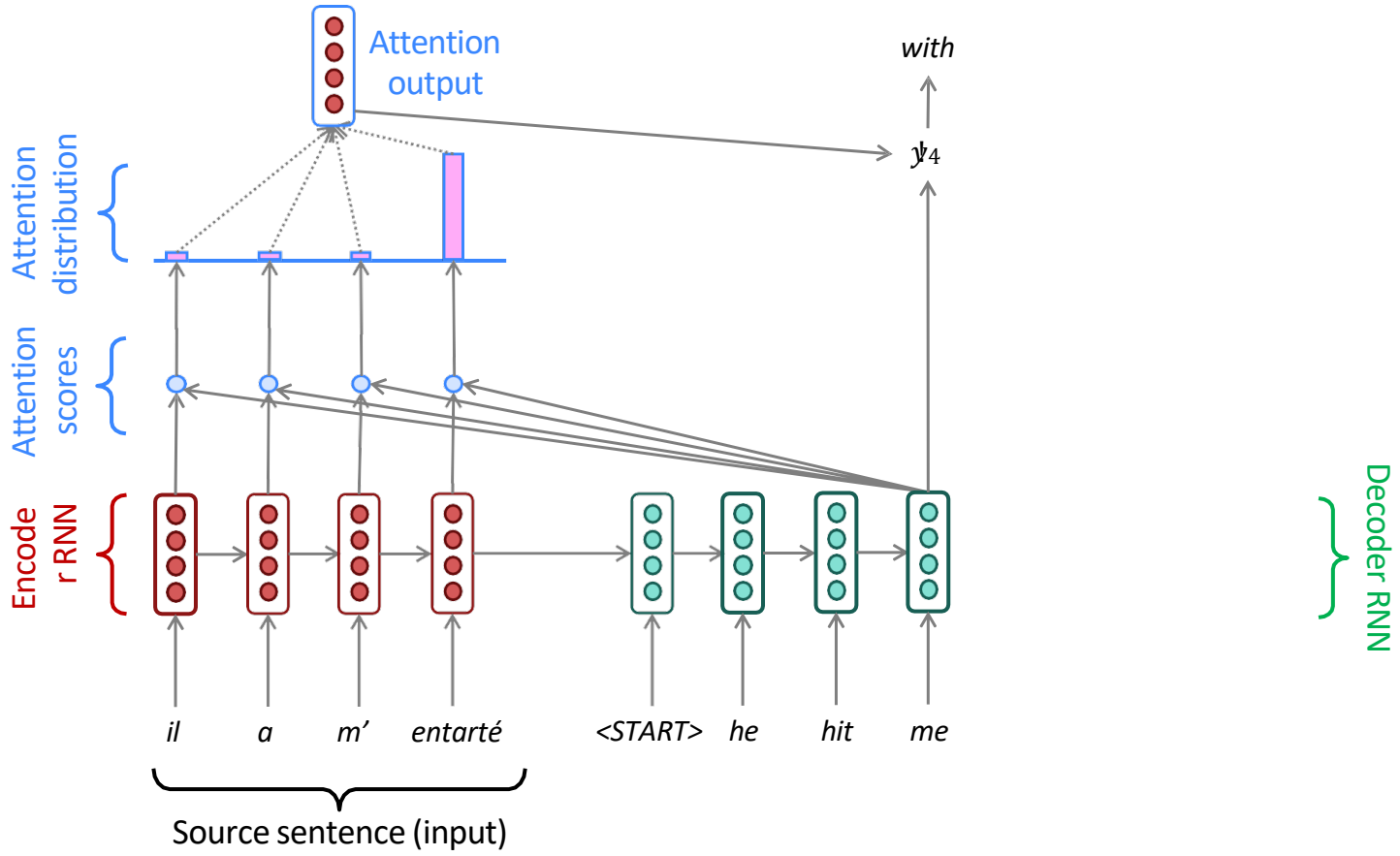
Attention for every time step

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



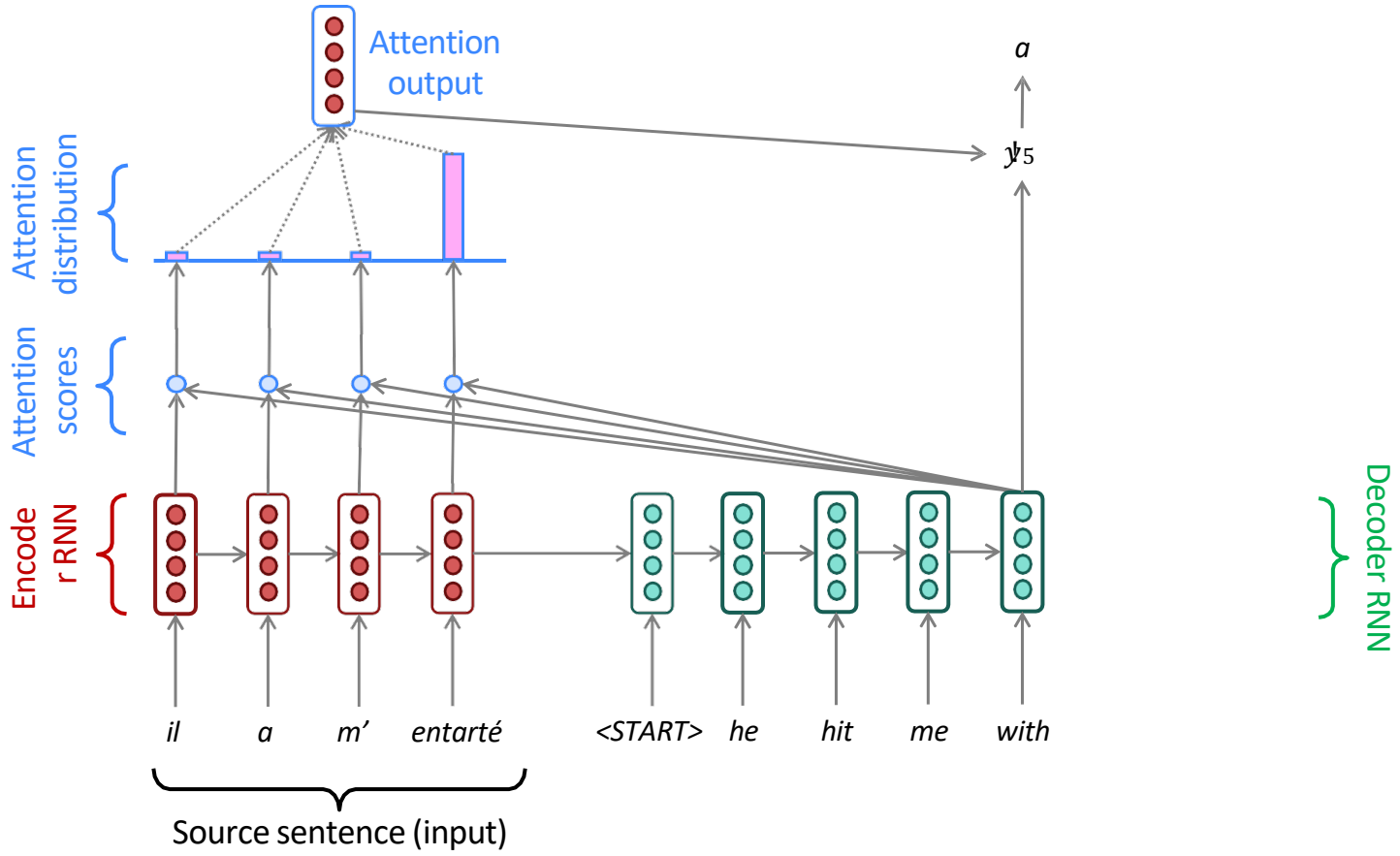
Attention for every time step

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



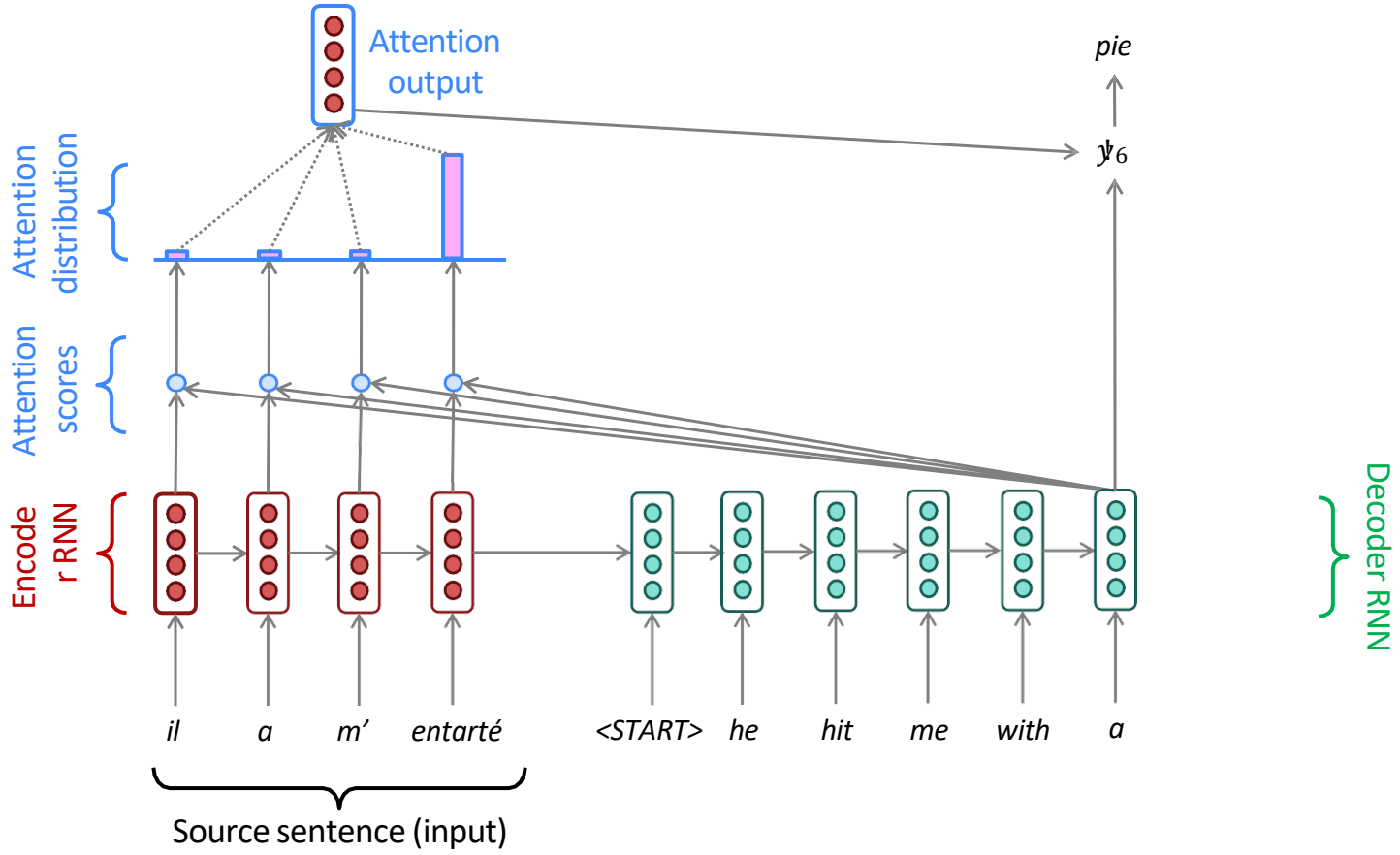
Attention for every time step

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



Attention for every time step

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



Attention Equations

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

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

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

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

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

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

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

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Attention Equations

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

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

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

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

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

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

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

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Attention Variants

- We have some *values* $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and a *query* $\boldsymbol{s} \in \mathbb{R}^{d_2}$

- Attention always involves:
  1. Computing the *attention scores* $\boldsymbol{e} \in \mathbb{R}^N$ ← There are multiple ways to do this
  2. Taking softmax to get *attention distribution* a:

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

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

$$\boldsymbol{a} = \sum_{i=1}^{N} \alpha_i \boldsymbol{h}_i \in \mathbb{R}^{d_1}$$

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

Attention Variants

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

- Basic dot-product attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
  - Note: this assumes $d_1 = d_2$. This is the version we saw earlier.
- Multiplicative attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$ [Luong, Pham, and Manning 2015]
  - Where $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix. Perhaps better called "bilinear attention"
- Reduced-rank multiplicative attention: $e_i = s^T (\boldsymbol{U}^T \boldsymbol{V}) h_i = (\boldsymbol{U}s)^T (\boldsymbol{V}h_i)$ ← Remember this when we look at Transformers next week!
  - For low rank matrices $\boldsymbol{U} \in \mathbb{R}^{k \times d_1}$, $\boldsymbol{V} \in \mathbb{R}^{k \times d_2}$, $k \ll d_1, d_2$
- Additive attention: $\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$ [Bahdanau, Cho, and Bengio 2014]
  - Where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}, \boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector.
  - d_3 (the attention dimensionality) is a hyperparameter
  - "Additive" is a weird/bad name. It's really using a feed-forward neural net layer.

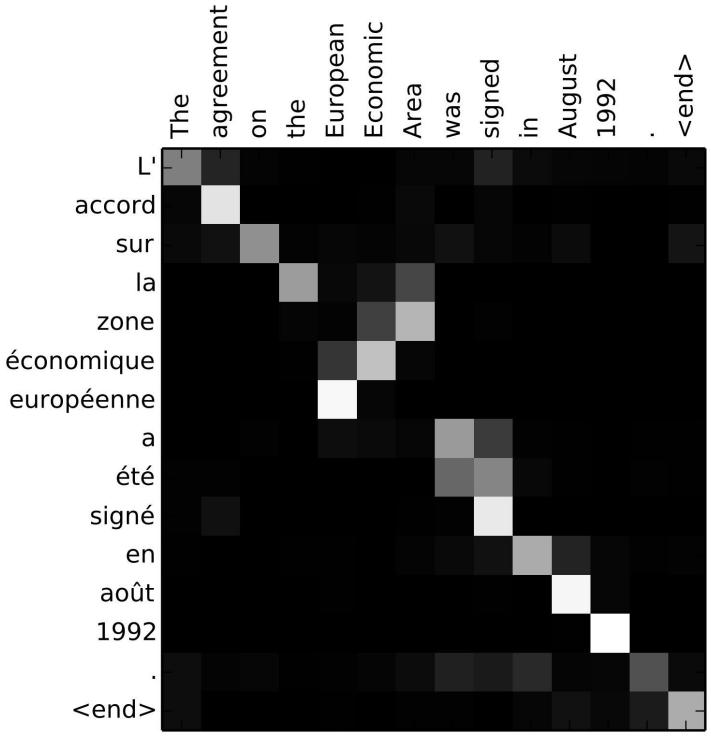
Attention Provides Explainability of Output

Visualize attention weights $a_{t,i}$

Example: English to French translation

Input: "The agreement on the European Economic Area was signed in August 1992."

Output: "L'accord sur la zone économique européenne a été signé en août 1992."



Attention Provides Explainability of Output

Example: English to French translation

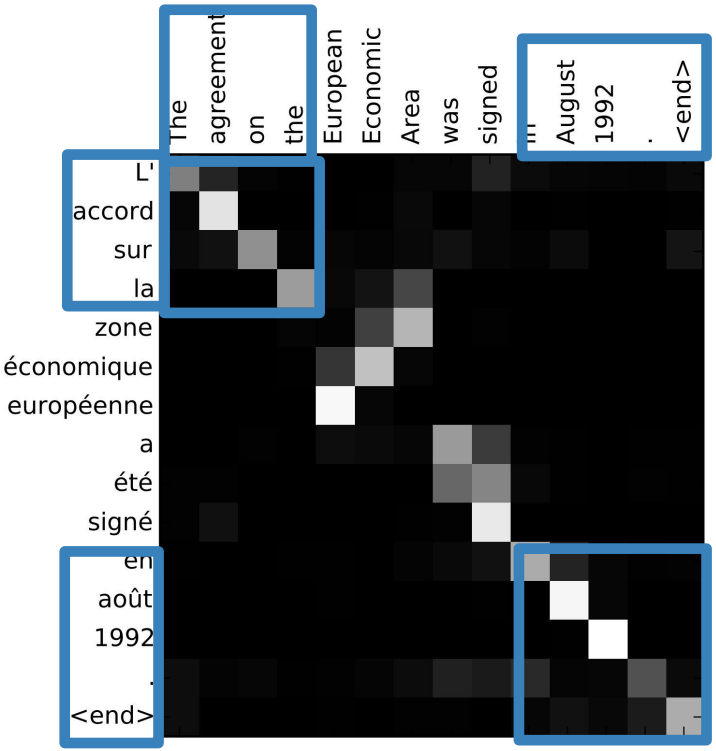
Input: "The agreement on the European Economic Area was signed in August 1992."

Output: "L'accord sur la zone économique européenne a été signé en août 1992."

Visualize attention weights $a_{t,i}$

Diagonal attention means words correspond in order

Diagonal attention means words correspond in order



Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Attention Provides Explainability of Output

Example: English to French translation

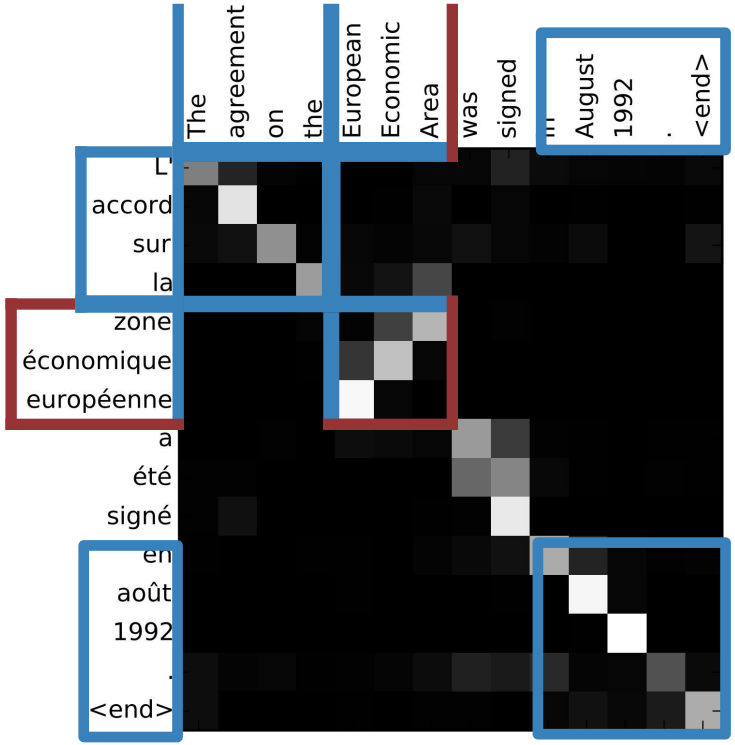
Input: "The agreement on the European Economic Area was signed in August 1992."

Output: "L'accord sur la zone économique européenne a été signé en août 1992."

Diagonal attention means words correspond in order

Attention figures out different word orders

Diagonal attention means words correspond in order



Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

LSTM for Sentence Encoding and Decoding - Machine Translation

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

NUS National University of Singapore

THANK YOU

Next Week - Transformers