



# CS60010: Deep Learning

## Spring 2023

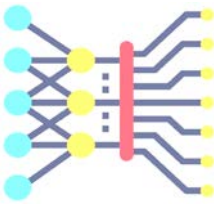
Sudeshna Sarkar

### **RNN Part 2**

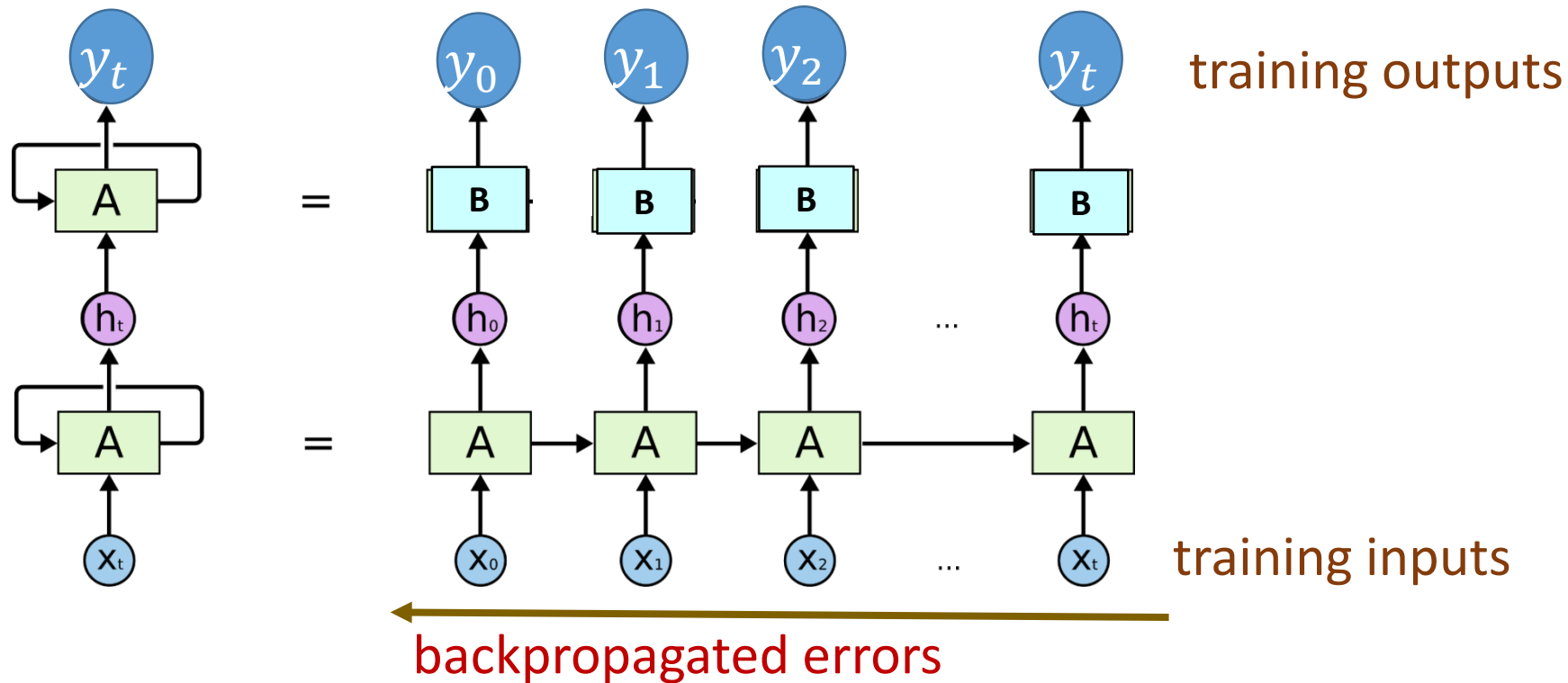
**Sudeshna Sarkar**

2 Mar 2023

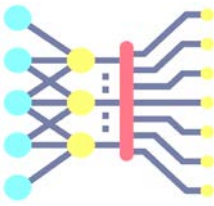
# Training RNN's



- RNNs can be trained using “backpropagation through time.”
  - applying normal backprop to the unrolled network.



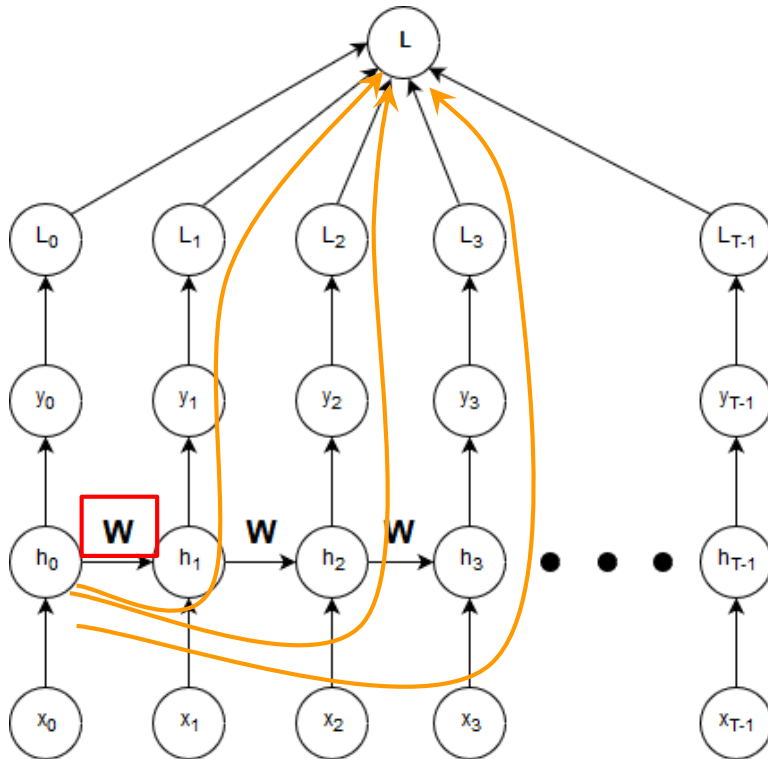
# Backpropagation Through Time (BPTT)



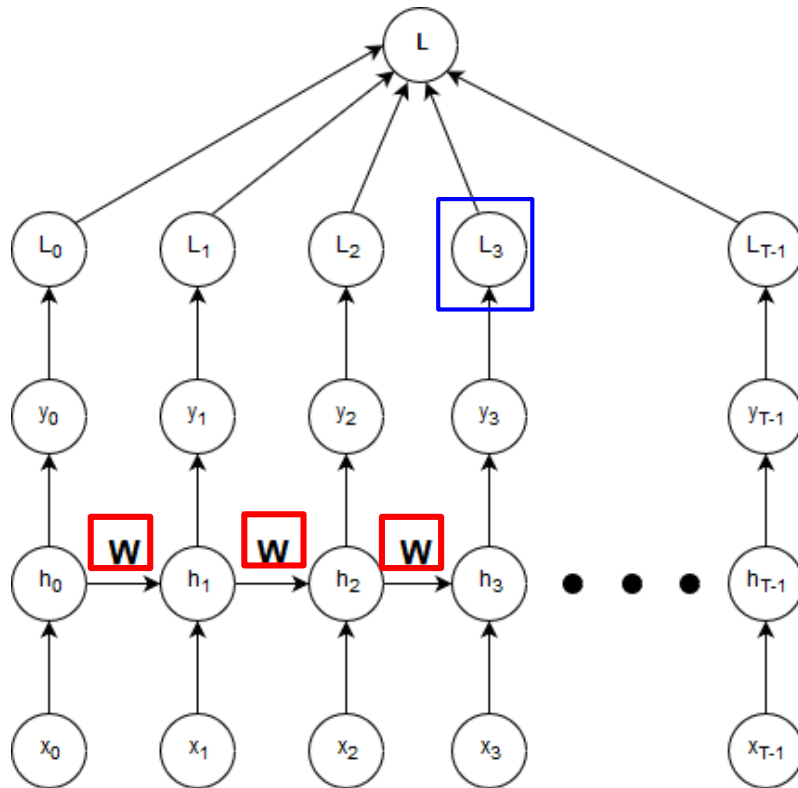
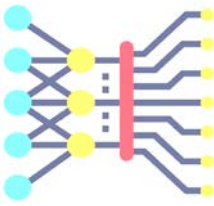
Update the weight matrix:

$$\mathbf{W} \rightarrow \mathbf{W} - \alpha \frac{\partial L}{\partial \mathbf{W}}$$

- Issue:  $\mathbf{W}$  occurs each timestep
- **Every** path from  $\mathbf{W}$  to  $L$  is one dependency
- Find all paths from  $\mathbf{W}$  to  $L$ !



# Backpropagation as two summations



1. First summation over L

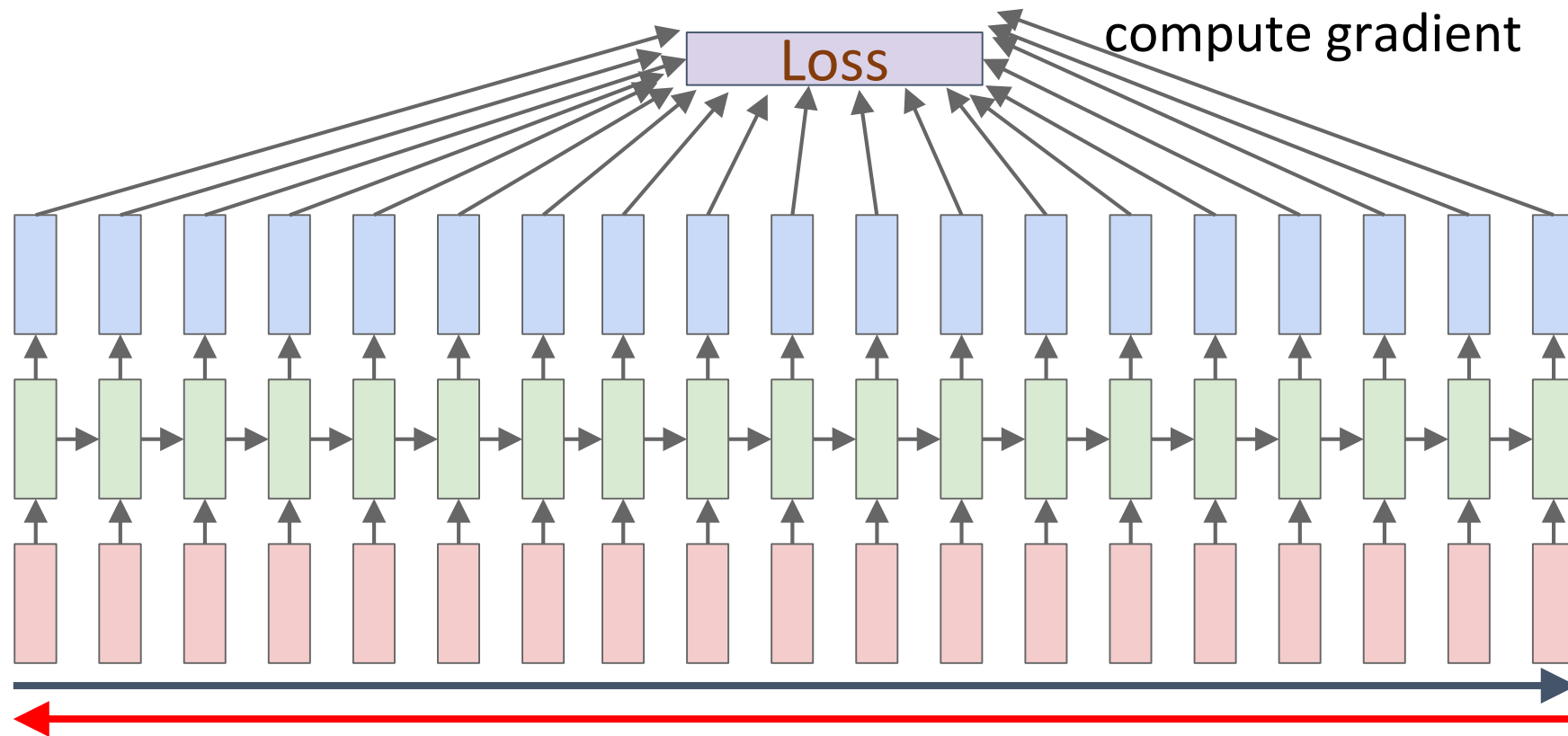
$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial \mathbf{W}}$$

2. Second summation over h:  
Each  $L_j$  depends on the weight matrices *before it*

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^j \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

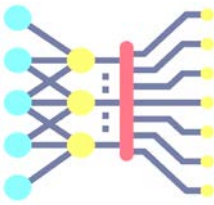
$L_j$  depends on all  $h_k$  before it.

## Backpropagation through time

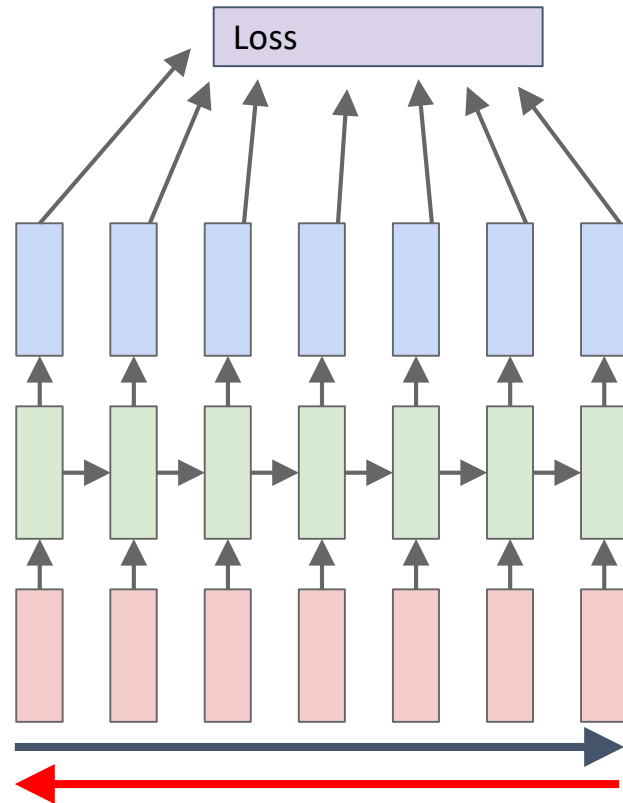


Forward through entire sequence  
to compute loss, then backward  
through entire sequence to  
compute gradient

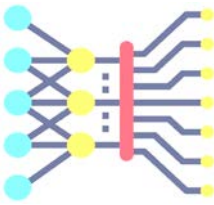




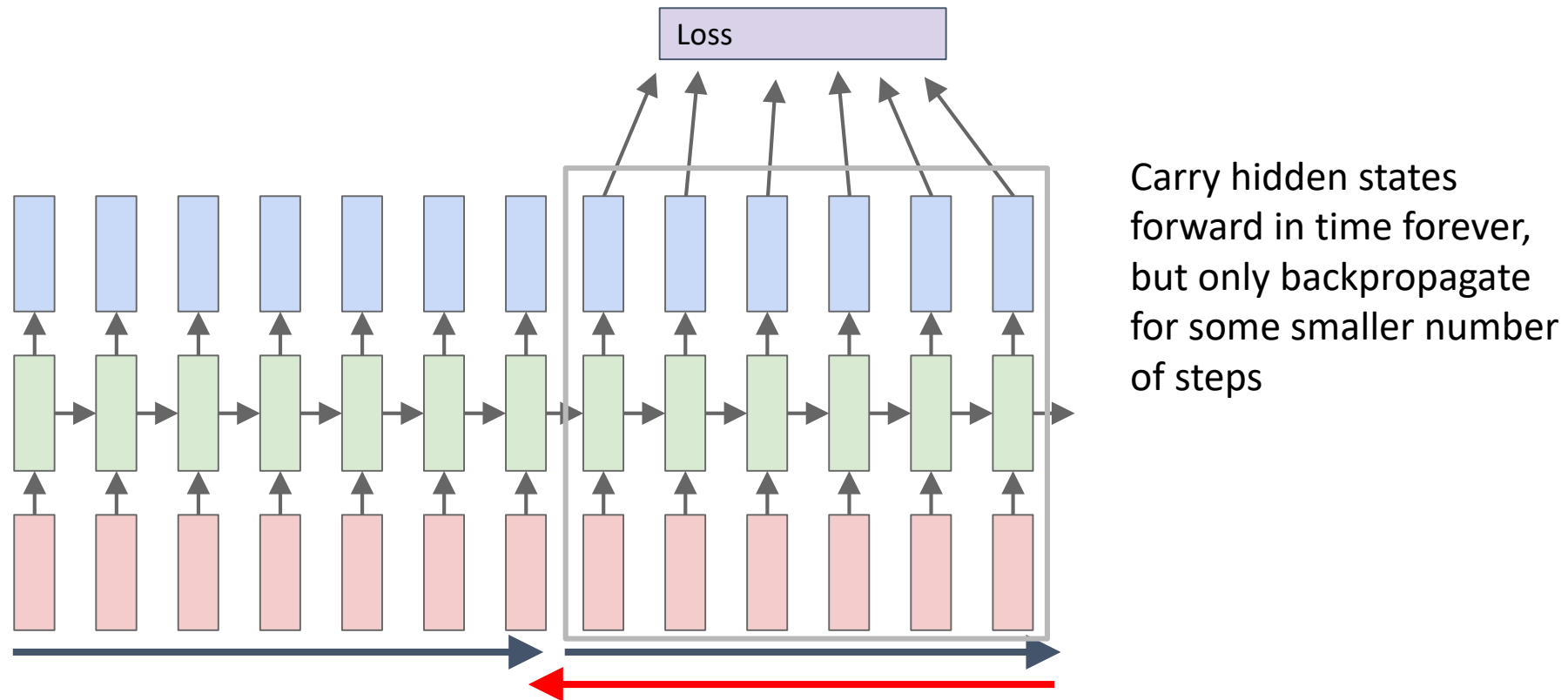
# Truncated Backpropagation through time

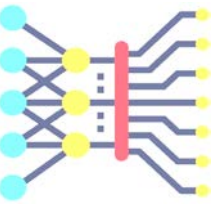


Run forward and backward through chunks of the sequence instead of whole sequence

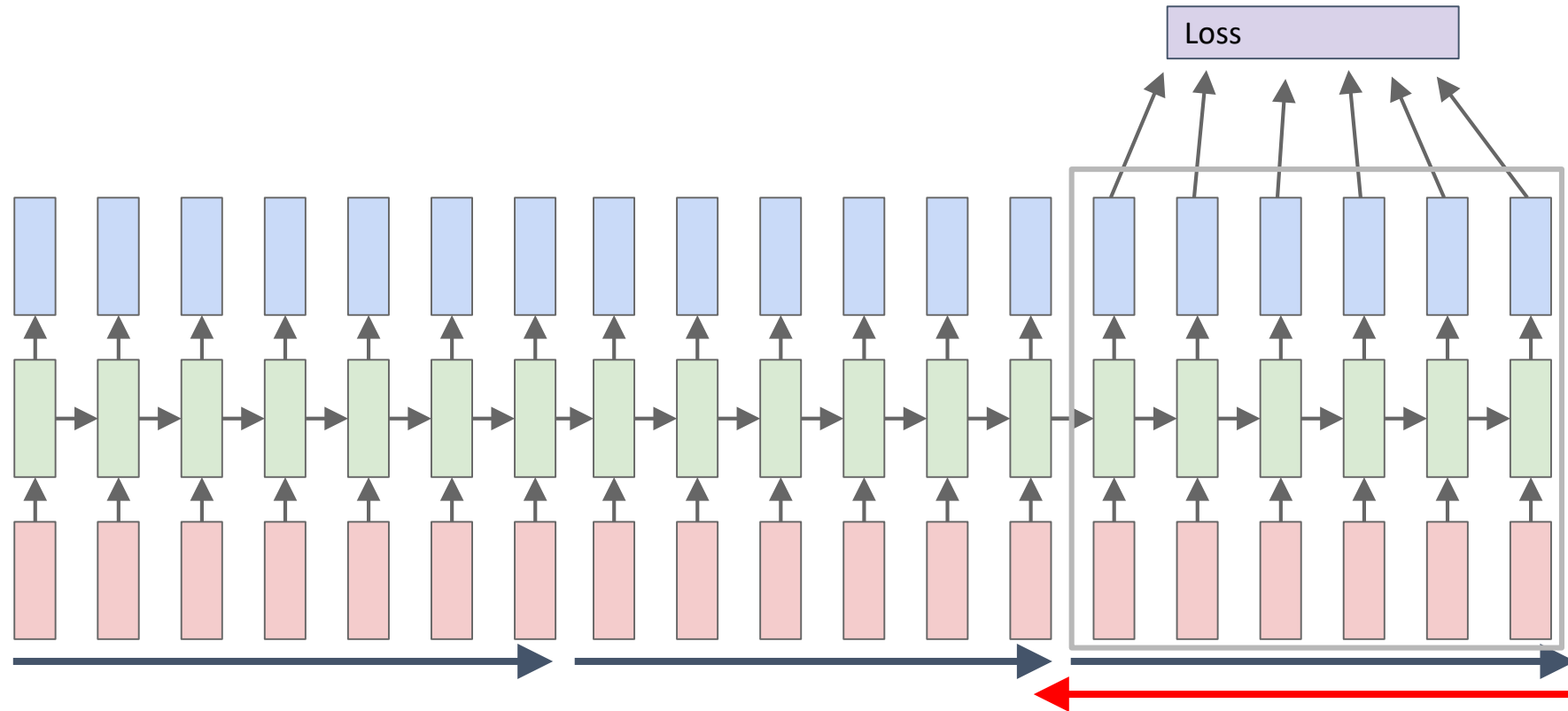


# Truncated Backpropagation through time



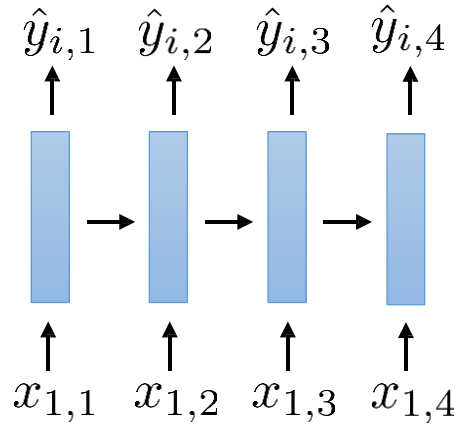
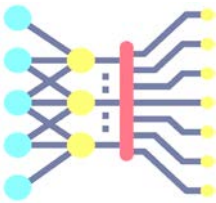


# Truncated Backpropagation through time





# RNNs are extremely deep networks



If sequence length is 100, backpropagate through 100 steps.

vanishing gradients = gradient signal from later steps never reaches the earlier steps

this prevents the RNN from “remembering” things from the beginning!

“vanishing gradients”

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

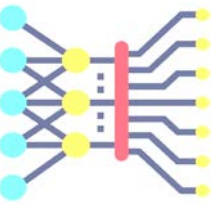
**If we multiply many many numbers together, what will we get?**

If most of the numbers are  $< 1$ , we get 0

If most of the numbers are  $> 1$ , we get infinity

We only get a reasonable answer if the numbers are all close to 1!

“exploding gradients”  
could fix with gradient clipping



# Promoting better gradient flow

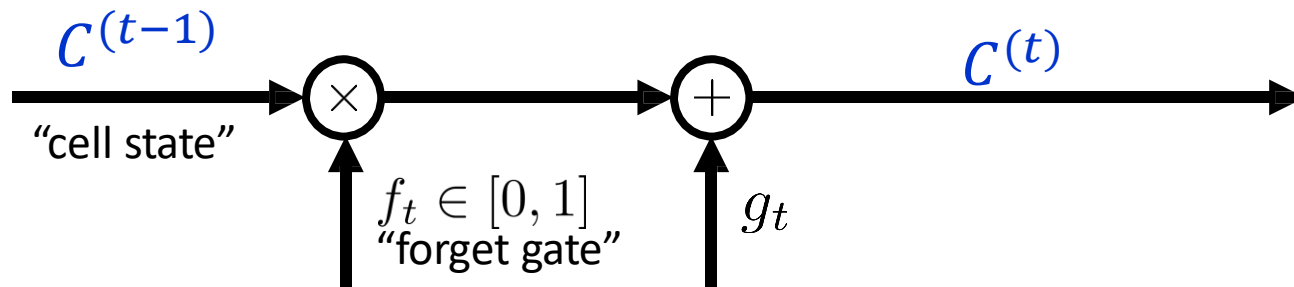
$$C^{(t)} = Q(C^{(t-1)}, x^{(t)})$$

**Basic idea:** we would really like the gradients to be close to 1

for each unit, we have a little “neural circuit” that decides whether to remember or overwrite

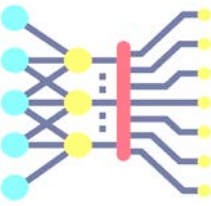
if “remembering,” just copy the previous activation as it is

if “forgetting,” just overwrite it with something based on the current input



$$C^{(t)} = C^{(t-1)} f_t + g_t$$
$$\frac{dQ}{dC^{(t-1)}} = f_t \in [0, 1]$$

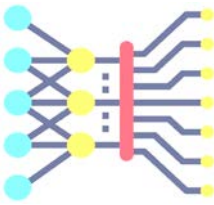
# Gated RNNs



- **Gated RNNs** contain **gates** to control what information is passed through.

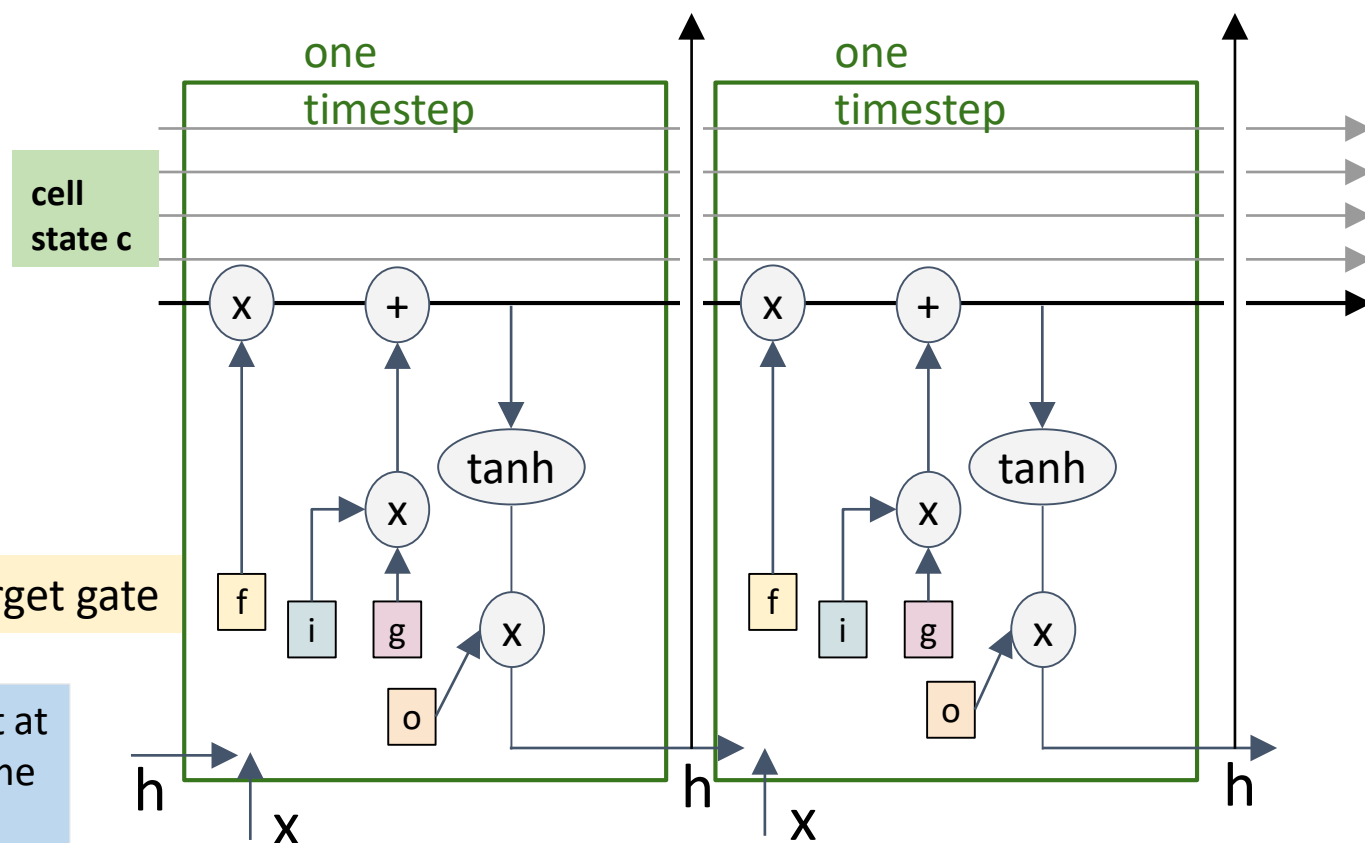
Gated Cells  
LSTM, GRU, etc

# Long short-term memory (LSTM)



1. A special kind of RNN, capable of learning long-term dependencies.
2. Introduced by Hochreiter & Schmidhuber (1997)

# LSTMs: learnable gates



$$g_t = \tanh(U_g h_{t-1} + W_g x_t)$$

$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$

$$f_t = \sigma(U_f h_{t-1} + W_f x_t)$$

$$o_t = \sigma(U_o h_{t-1} + W_o x_t)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

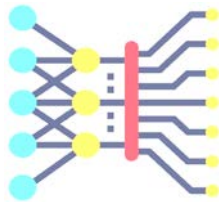
$$h_t = o_t \odot \tanh(c_t)$$

changes very little step to step!

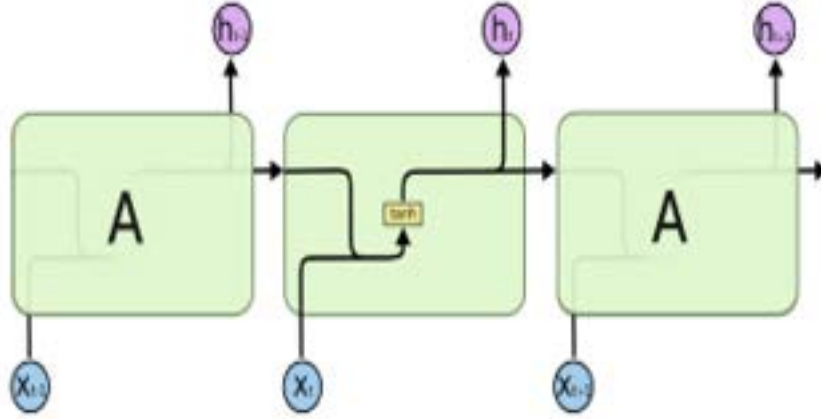
“long term” memory

changes all the time (multiplicative)

“short term” memory

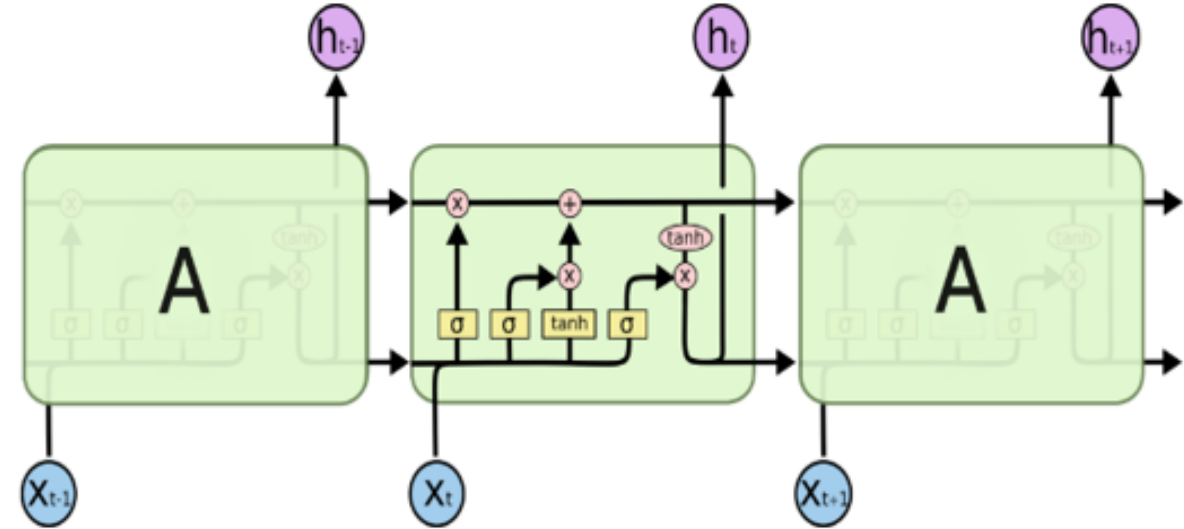


## Basic RNN unit



$$h_t = \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

## LSTM unit

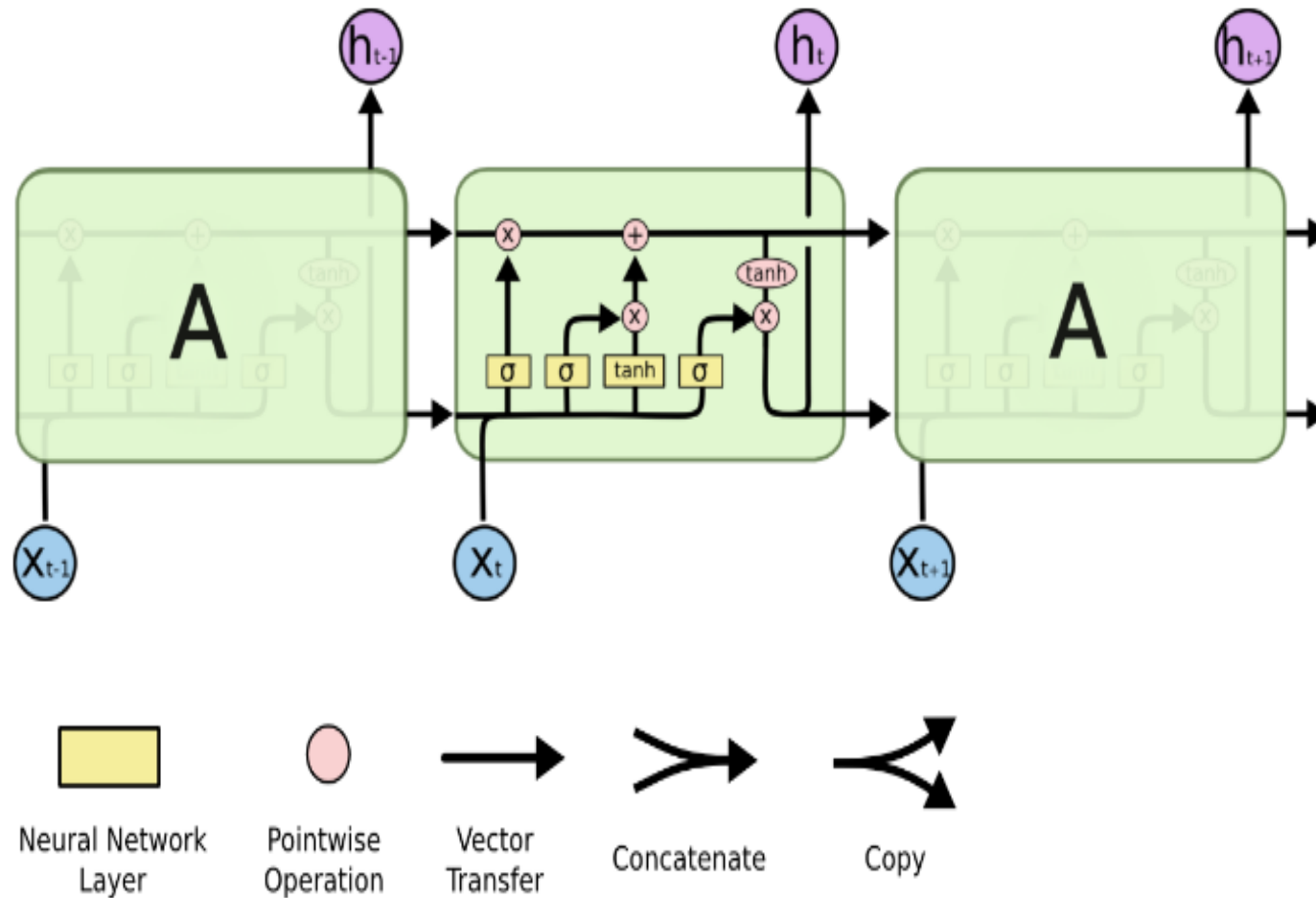
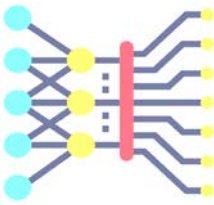


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

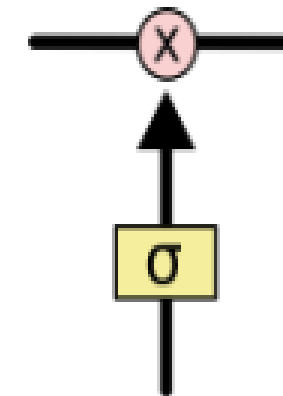
$$h_t = o \odot \tanh(c_t)$$

# Long short-term memory

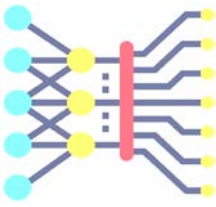


Information added or removed through gates.

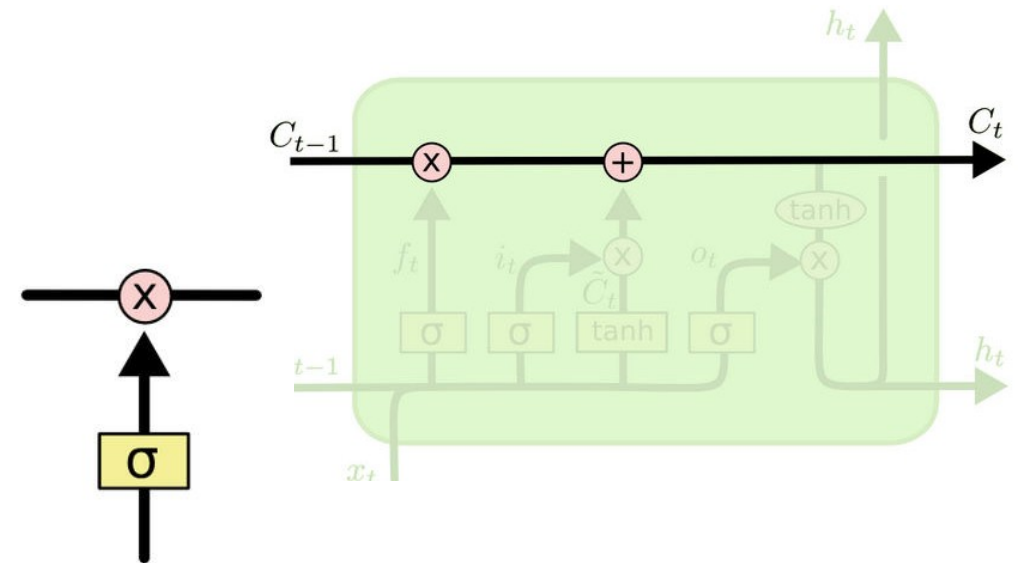
Ex: sigmoid net and pointwise multiplication



# Core idea behind LSTM

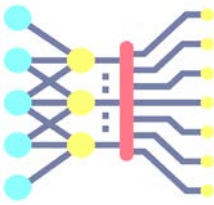


- Gates are an optional way to let information through.
- Three gates protect and control cell state

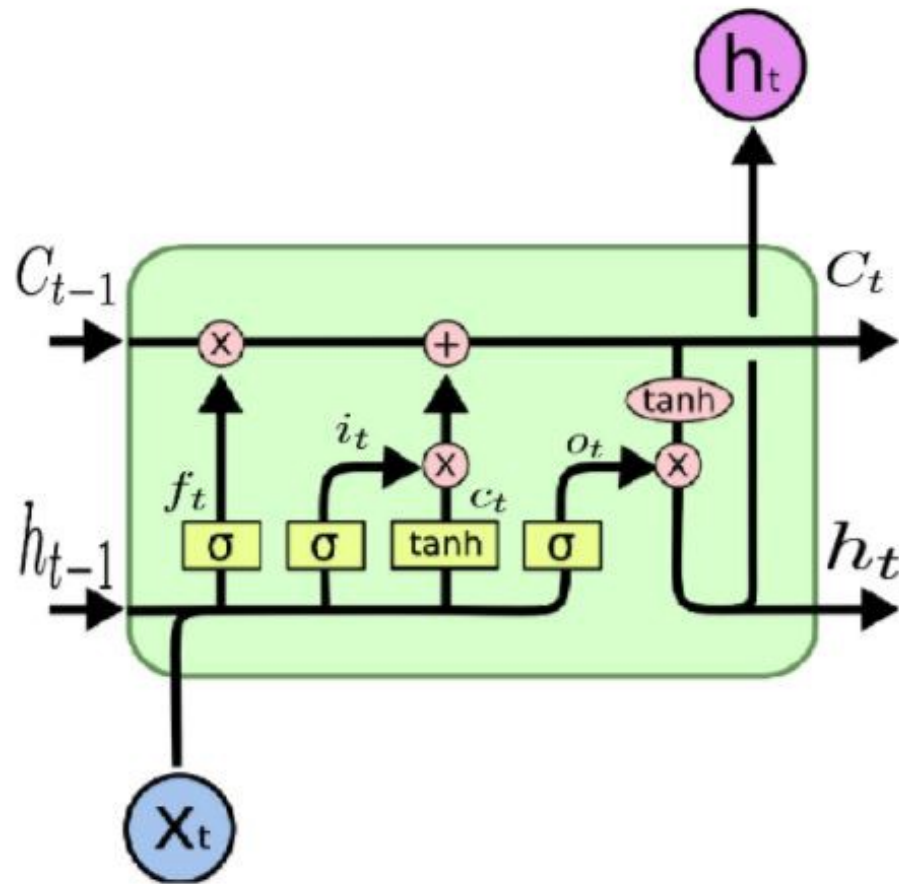




# Long short-term memory

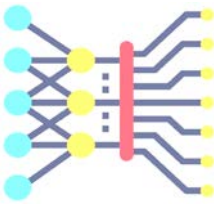


(1) Forget    (2) Input    (3) Update    (4) Output

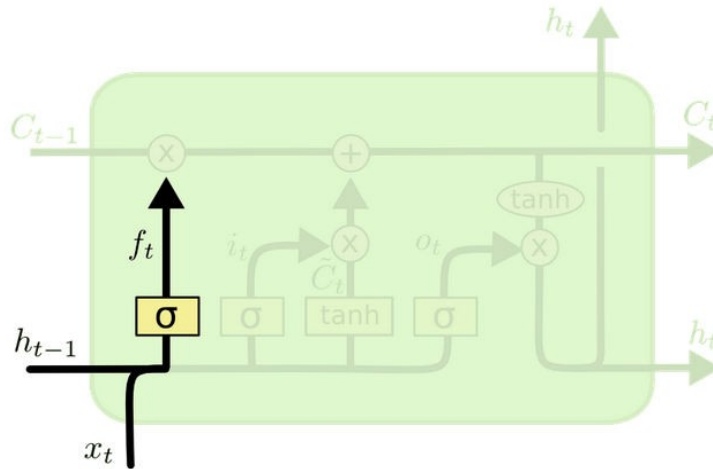


The key to LSTMs is the **cell state**,  
It runs straight down the entire  
chain, with only some minor linear  
interactions.

# LSM walk through: Step 1

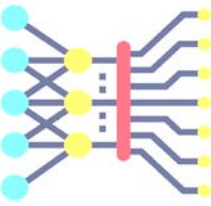


- **Step 1:** decide what information we're going to throw away from the cell state : “forget gate layer.”
- It looks at  $h_{t-1}$  and  $x_t$ , and outputs a number between 0 and 1 for each number in the cell state  $C_{t-1}$  for whether to forget.



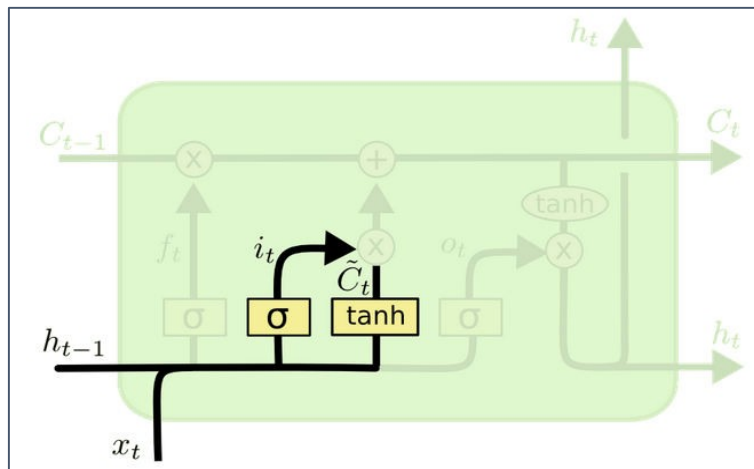
$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

- Language model: predict next word based on previous ones
  - Cell state may include the gender of the present subject so that the proper pronouns can be used
- When we see a new subject we want to forget old subject



# LSTM walk through: Step 2

Decide what new information we're going to store in the cell state



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

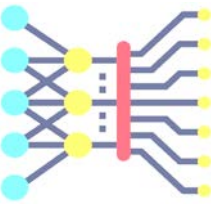
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Two parts:

Input gate layer: decides which values we will update

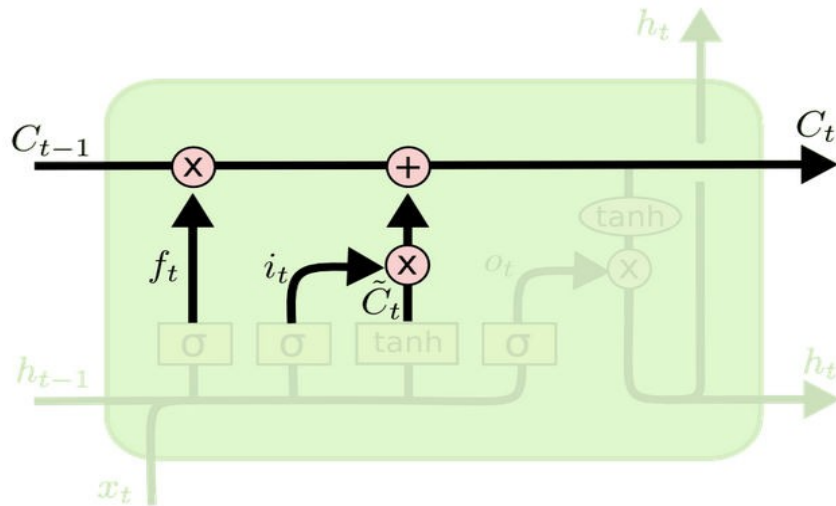
Atanh layer creates a vector of new candidate values ( $\tilde{C}_t$ ) that could be added to the state.

In the Language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we are forgetting



# LSTM walk through: Step 3

- Update old cell state  $C_{t-1}$  into new cell state  $C_t$ .



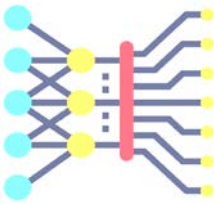
Multiply the old state by  $f_t$ , forgetting the things we decided to forget earlier.

- Then we add  $i_t * \tilde{C}_t$ .

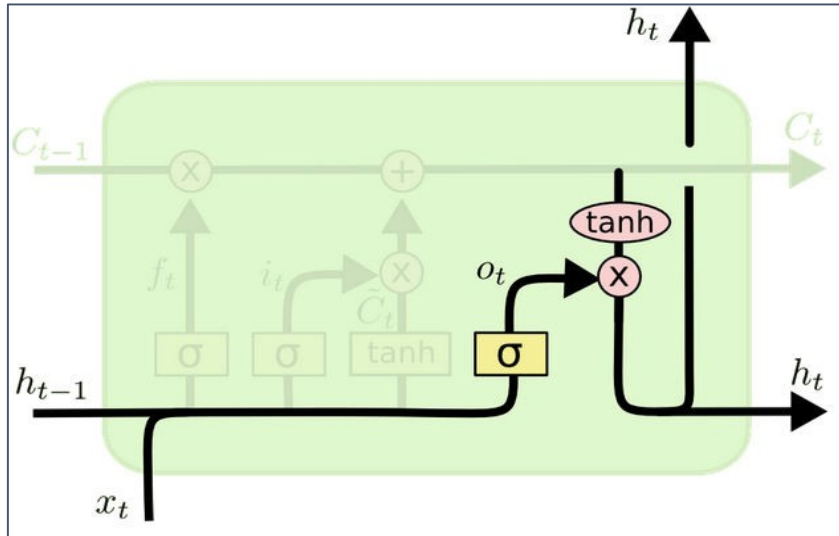
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

In the Language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in previous steps

# LSTM walk through: Step 4



Finally we decide what we are going to output



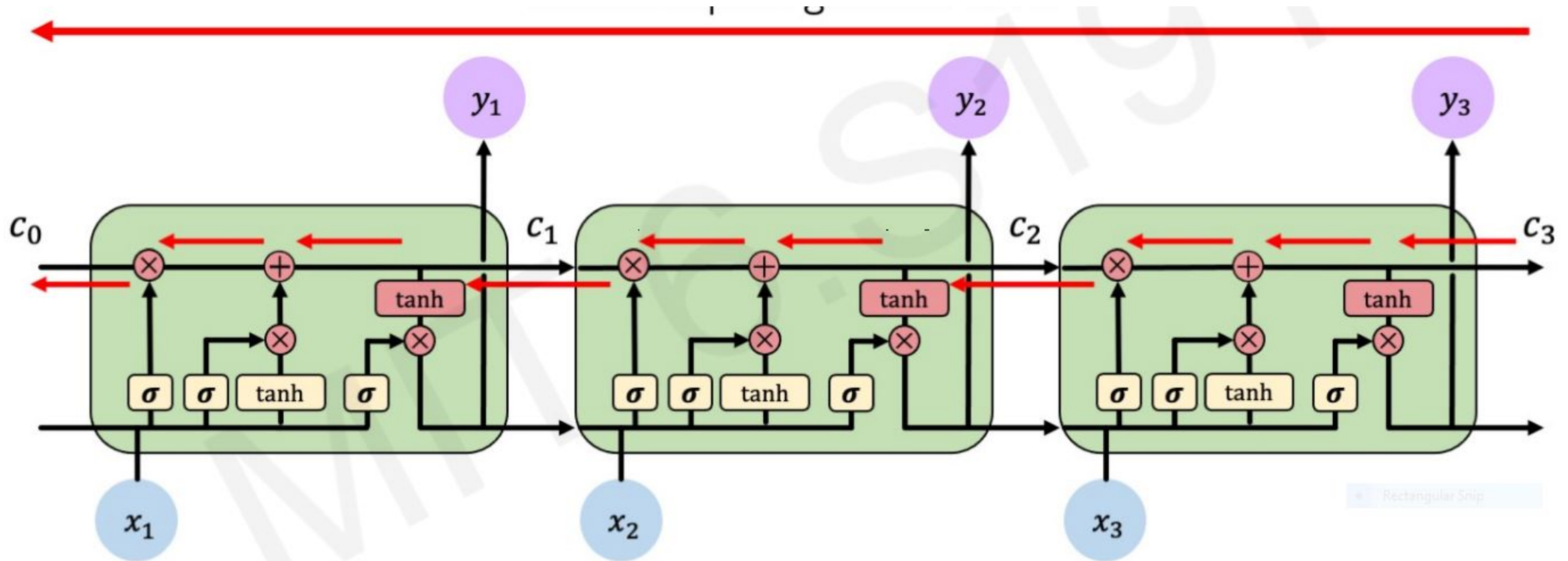
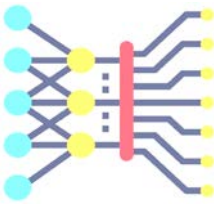
This output will be based on our cell state, but will be a filtered version.

First we run a sigmoid layer which decides what parts of cell state we're going to output. Then we put the cell state through tanh (to push values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we output only the parts we decided to

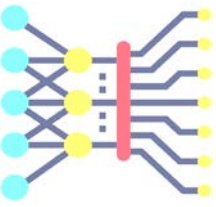
For the Language model,  
Since it just saw a subject it  
might want to output  
information relevant to a verb,  
in case that

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

# LSTM gradient flow is uninterrupted.

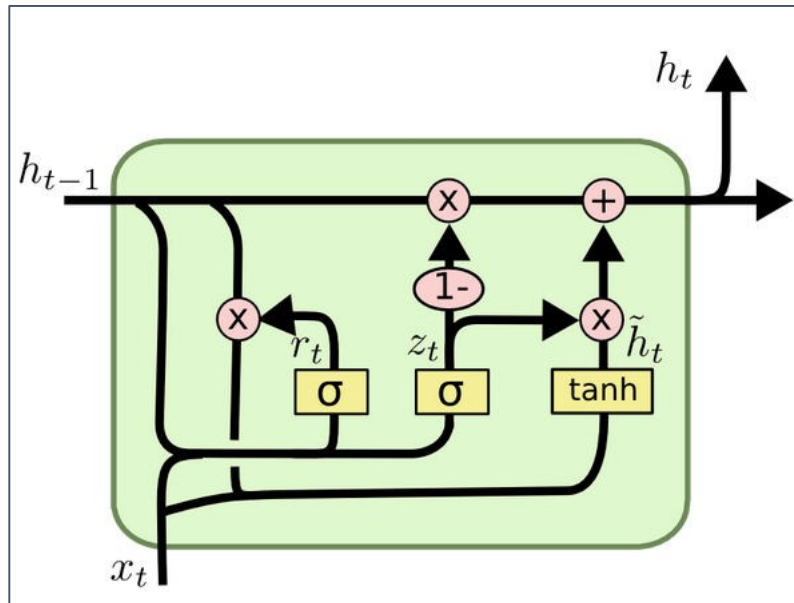


Backpropagation from  $c_t$  to  $c_{t-1}$  only elementwise multiplication by  $f$ , no matrix multiply by  $W$



# Gated Recurrent Unit (GRU)

- A dramatic variant of LSTM
  - It combines the forget and input gates into a single update gate
  - It also merges the cell state and hidden state, and makes some other changes
  - The resulting model is simpler than LSTM models



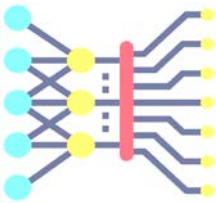
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

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

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

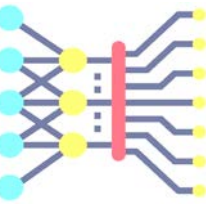
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

# Notes



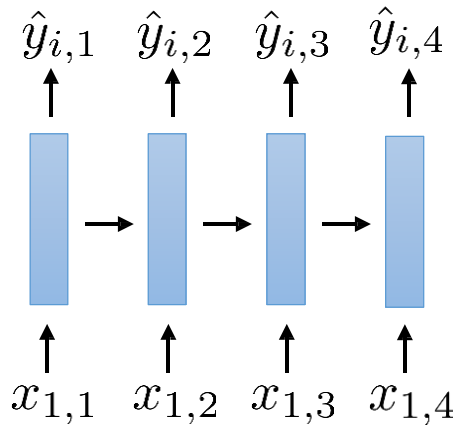
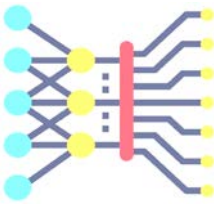
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- LSTM units are OK – they work fine in many cases, and dramatically improve over naïve RNNs
  - Still require lot of hyperparameter tuning
- LSTM cells are complicated, but once implemented, they can be used the same as any other type of layer.





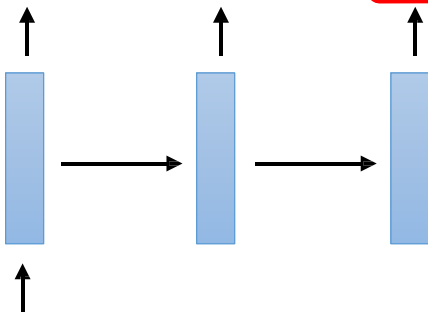
Using RNNs

# Autoregressive models and structured prediction



**Example:** text generation

think: 0.3    therefore: 0.3    I: 0.3  
like: 0.3    machine: 0.3    learning: 0.3  
am: 0.4    not: 0.4    just: 0.4



most problems that require multiple outputs have strong **dependencies** between these outputs

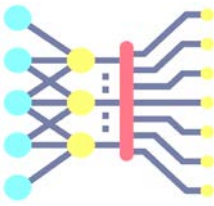
this is sometimes referred to as **structured** prediction

I think therefore I am  
I like machine learning  
I am not just a neural network

we get a nonsense output even though the network had exactly the right probabilities!

**Key idea:** past outputs should influence future outputs!

# Autoregressive models and structured prediction



How do we train it?

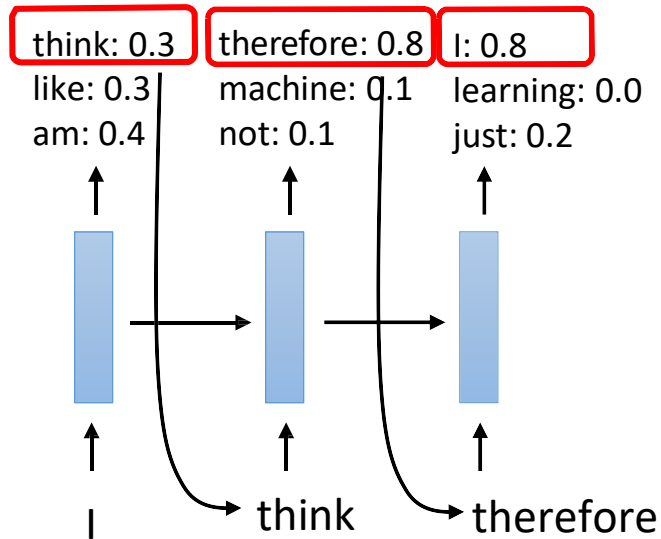
**Basic version:** just set inputs to be entire training sequences, and ground truth outputs to be those same sequences (offset by one step)

$x_{1:5} = (\text{"I"}, \text{"think"}, \text{"therefore"}, \text{"I"}, \text{"am"})$

$y_{1:5} = (\text{"think"}, \text{"therefore"}, \text{"I"}, \text{"am"}, \text{stop\_token})$

This teaches the network to output “am” if it sees “I think therefore I”

**Example:** text generation

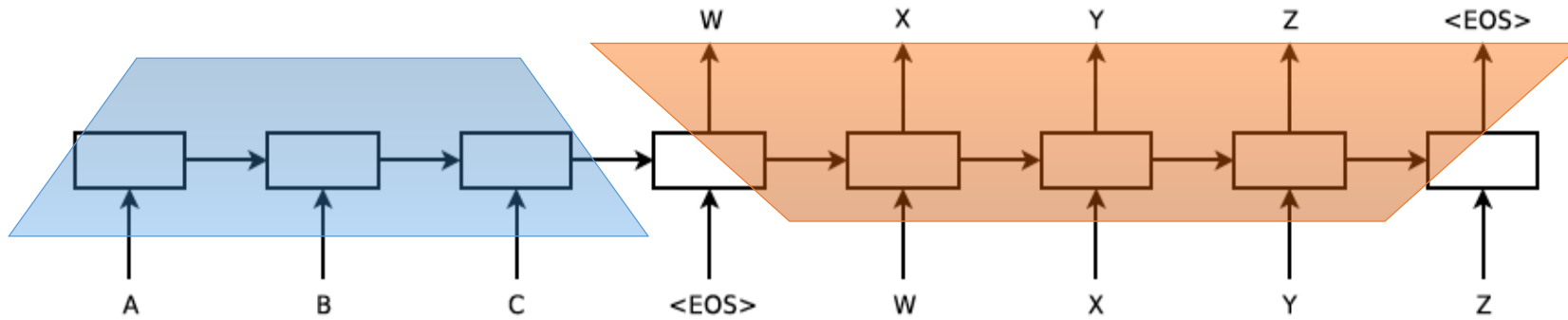
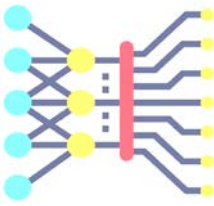


I think therefore I am

I like machine learning

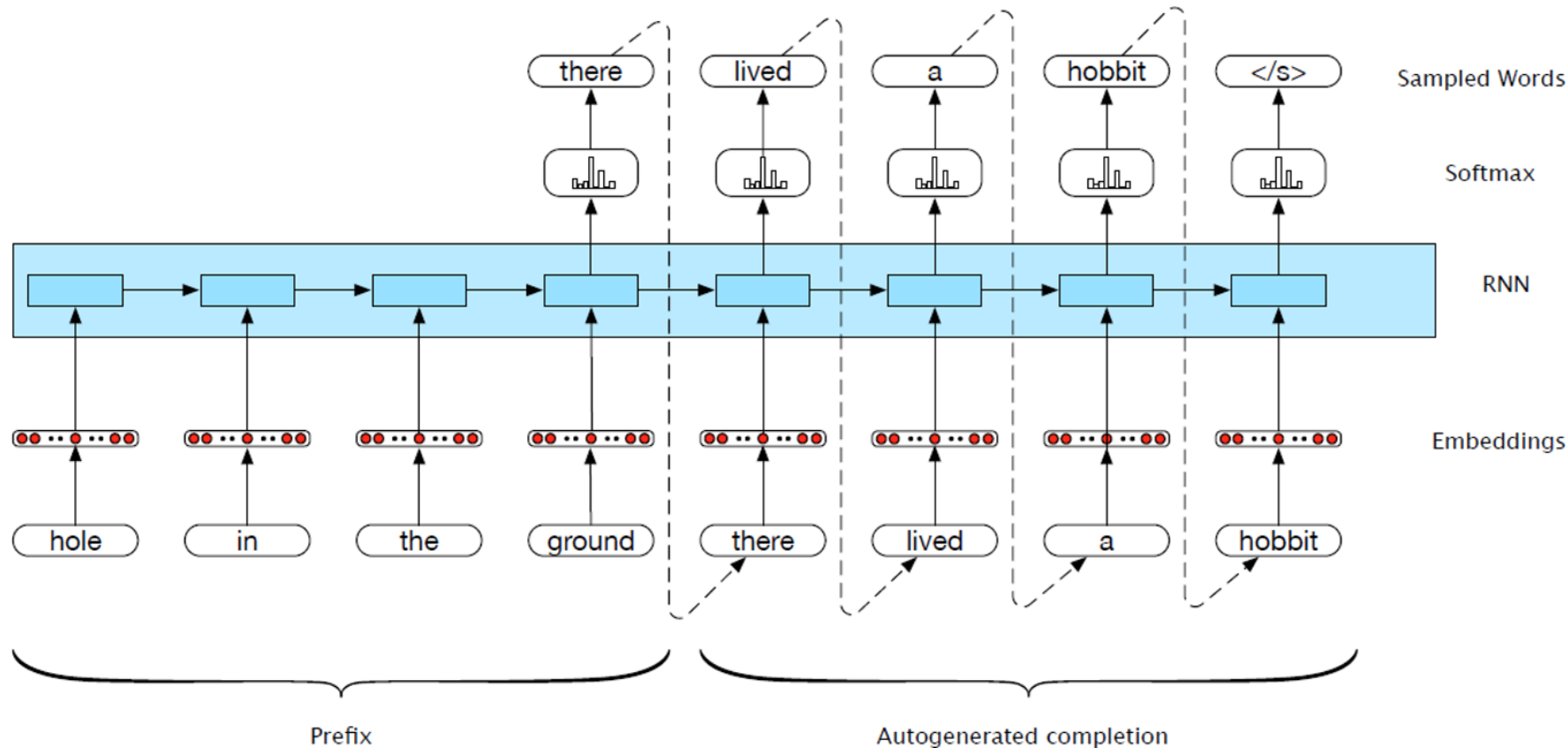
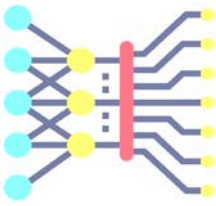
I am not just a neural network

# Neural Machine Translation



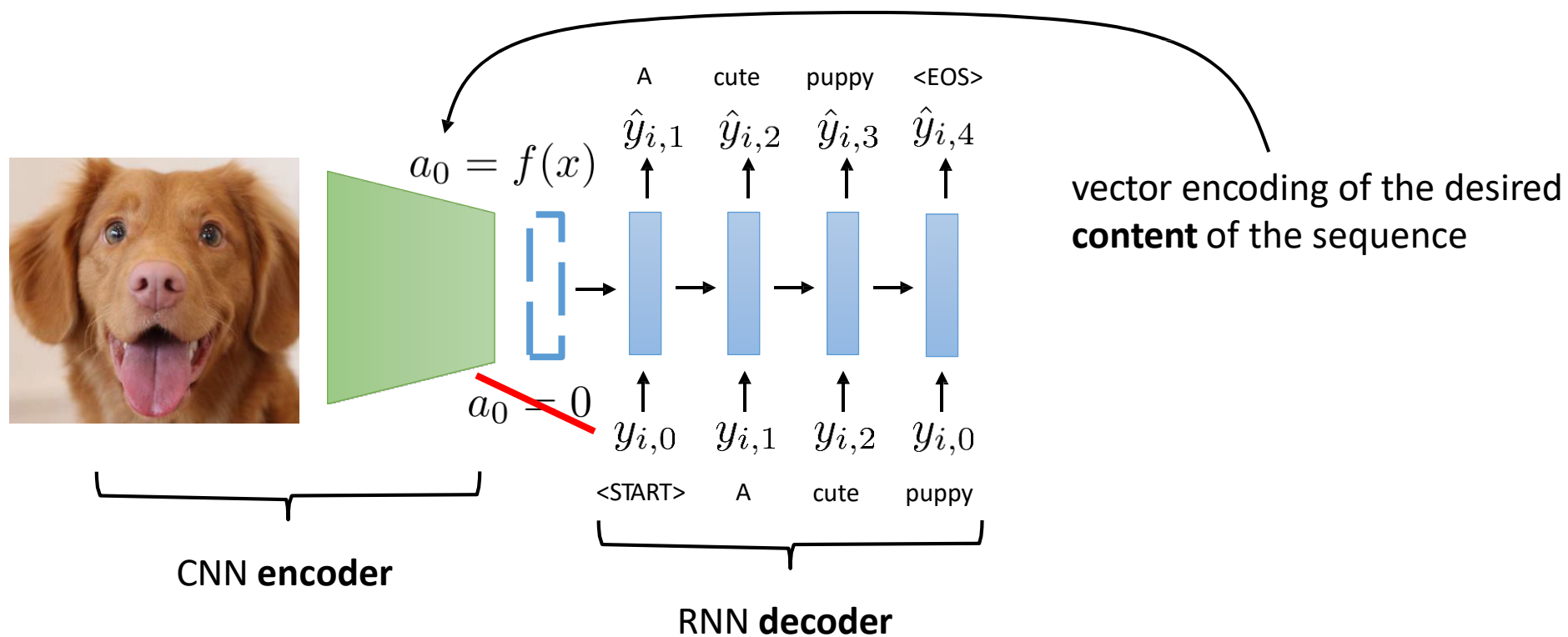
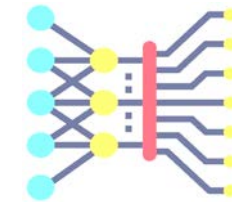
Sutskever et al. 2014

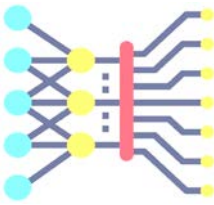
# Generation with prefix: Conditional Language Model



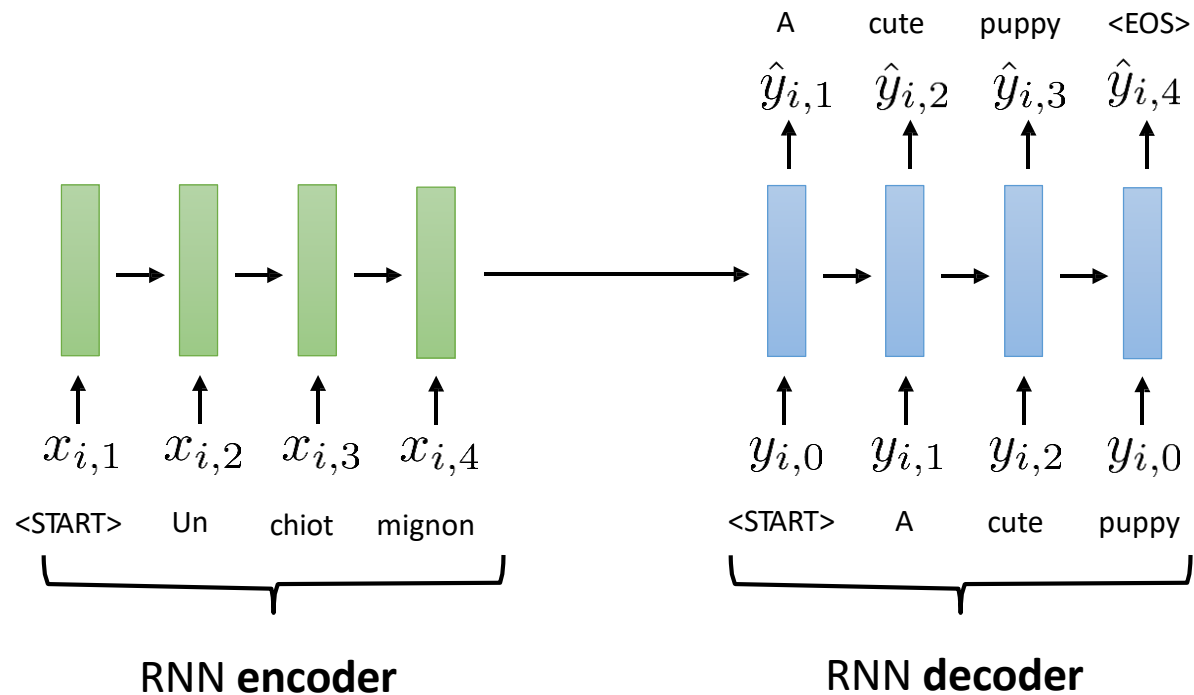
$$h_t = g(h_{t-1}, x_t)$$
$$y_t = f(h_t)$$

# A conditional language model





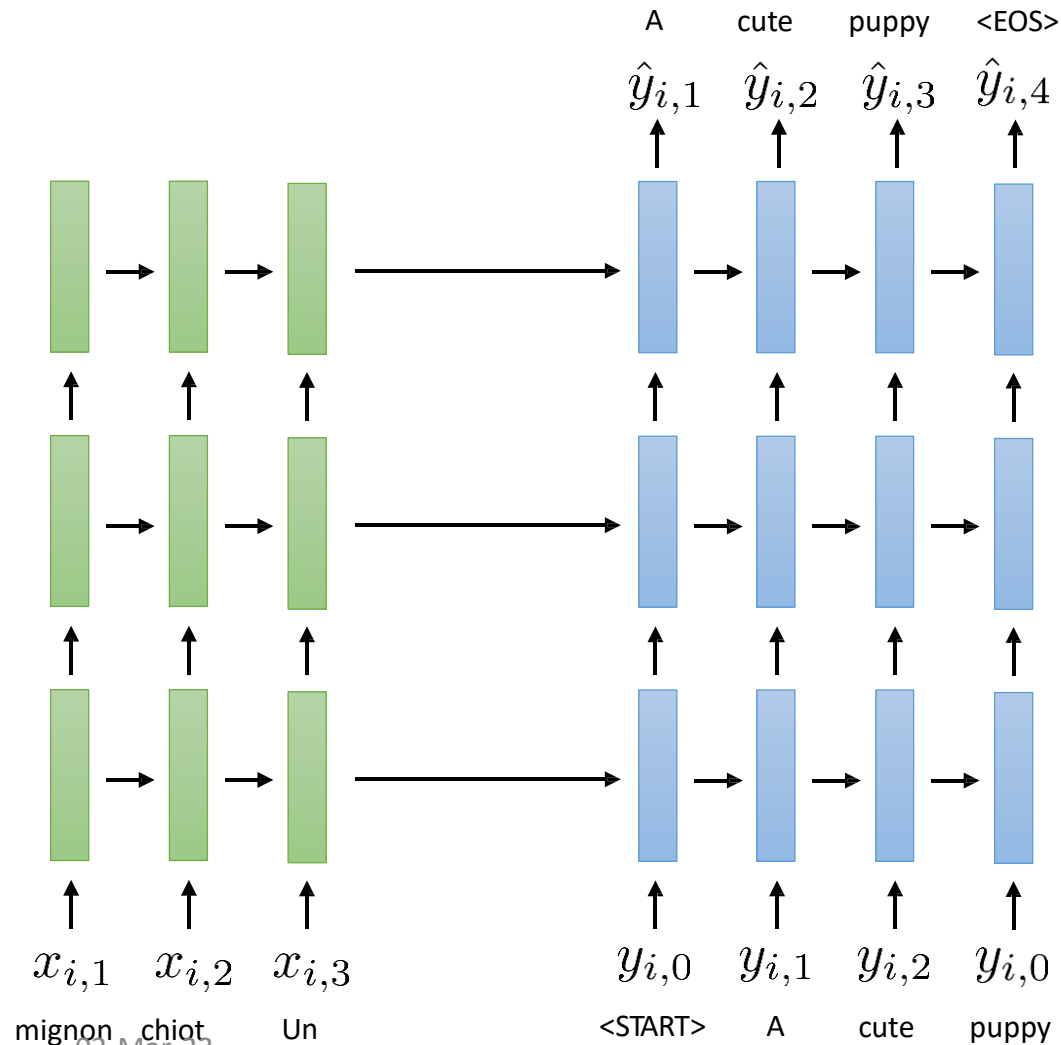
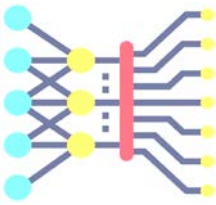
# Sequence to sequence models



typically two **separate** RNNs (with different weights)

trained **end-to-end** on paired data (e.g., pairs of French & English sentences)

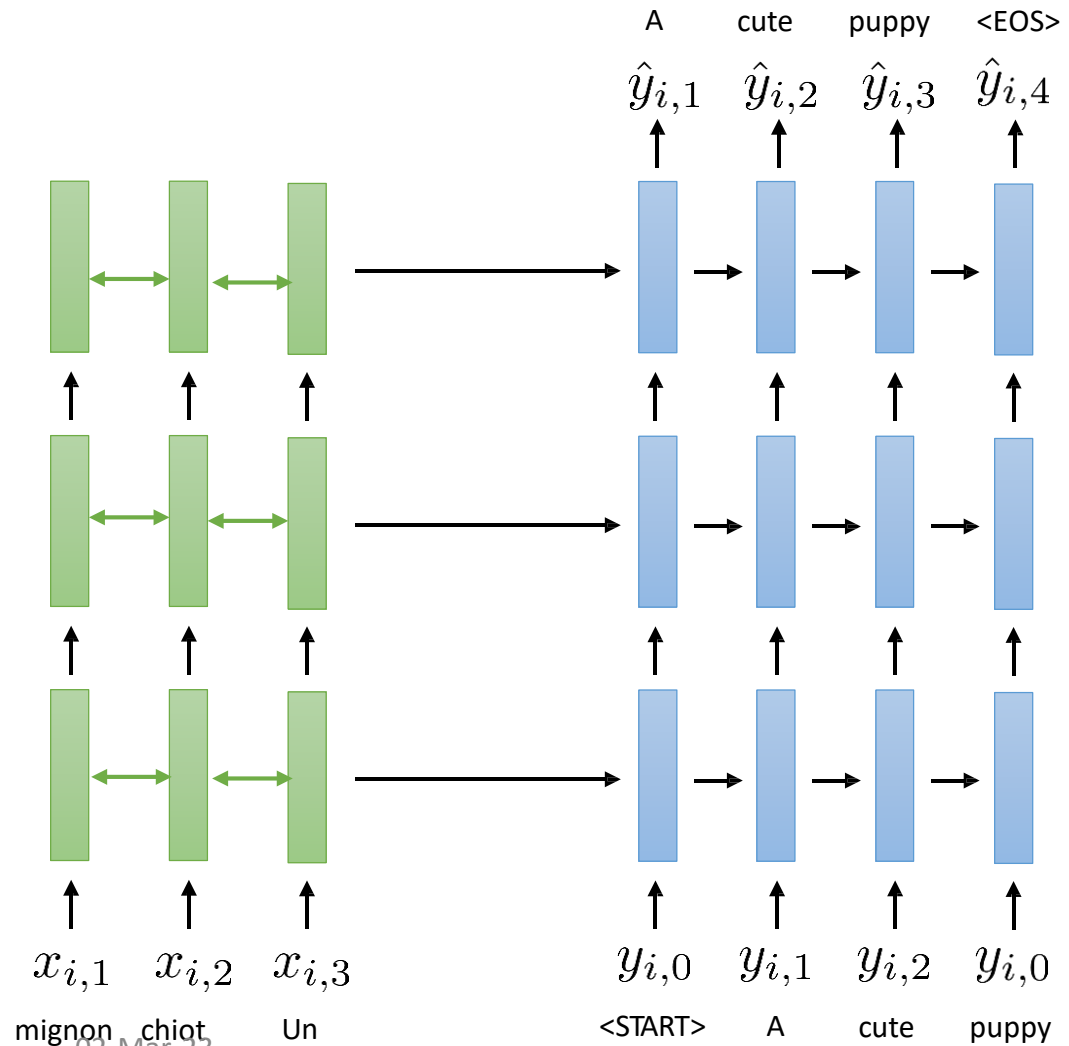
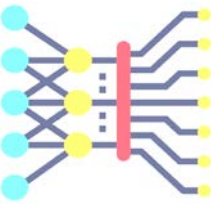
# Deeper models



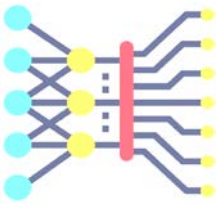
- Multiple RNN layers
  - Each RNN layer uses LSTM cells (or GRU)
  - Trained end-to-end on pairs of sequences
  - Sequences can be different lengths
- 
- Translate **one language** into **another language**
  - Summarize a **long sentence** into a **short sentence**
  - Respond to a **question** with an **answer**
  - Code generation? **text** to **Python code**
- For more, see: Ilya Sutskever, Oriol Vinyals, Quoc V. Le.  
**Sequence to Sequence Learning with Neural Networks.** 2014.



# Bidirectional models

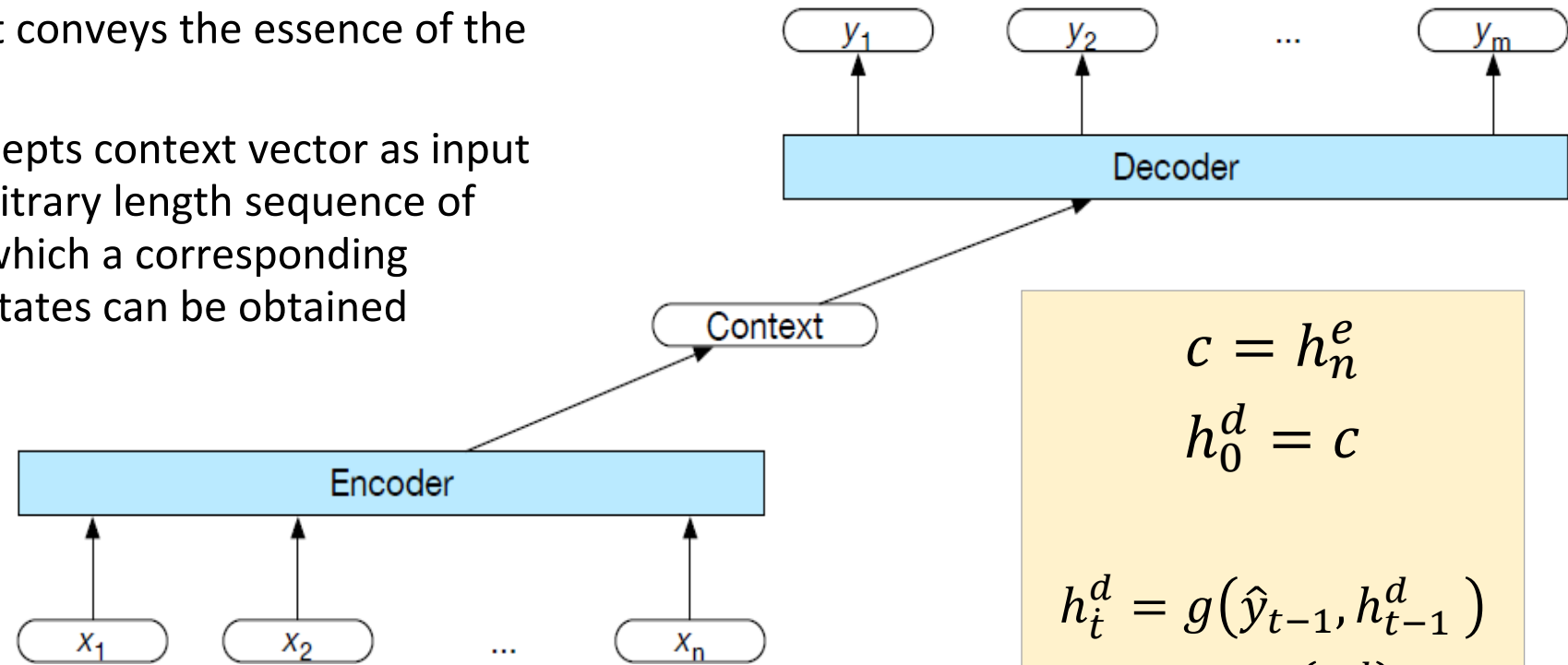


# Encoder-decoder networks



- An **encoder** that accepts an input sequence and generates a corresponding sequence of contextualized representations
- A **context vector** that conveys the essence of the input to the decoder
- A **decoder**, which accepts context vector as input and generates an arbitrary length sequence of hidden states, from which a corresponding sequence of output states can be obtained

simple RNNs, LSTMs, GRUs,  
stacked Bi-LSTMs widely used  
CNN



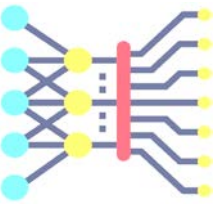
$$c = h_n^e$$

$$h_0^d = c$$

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d)$$

$$z_t = f(h_t^d)$$

$$y_t = \text{soft max}(z_t)$$



# Decoder Weaknesses

The context vector  $c$  *only* available at the beginning of the generation process.

- Its influence became less-and-less important as the output sequence was generated.
- Solution: Make  $c$  available at each step in the decoding process,
  1. when generating the hidden states in the deocoder
  2. while producing the generated output.

$$c = h_n^e$$

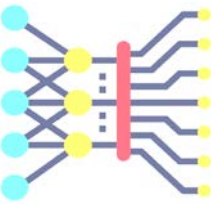
$$h_0^d = c$$


$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$

$$z_t = f(h_t^d)$$

$$y_t = \text{soft max}(\hat{y}_{t-1}, z_t, c)$$

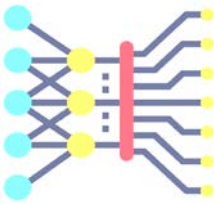
# Choosing the best output



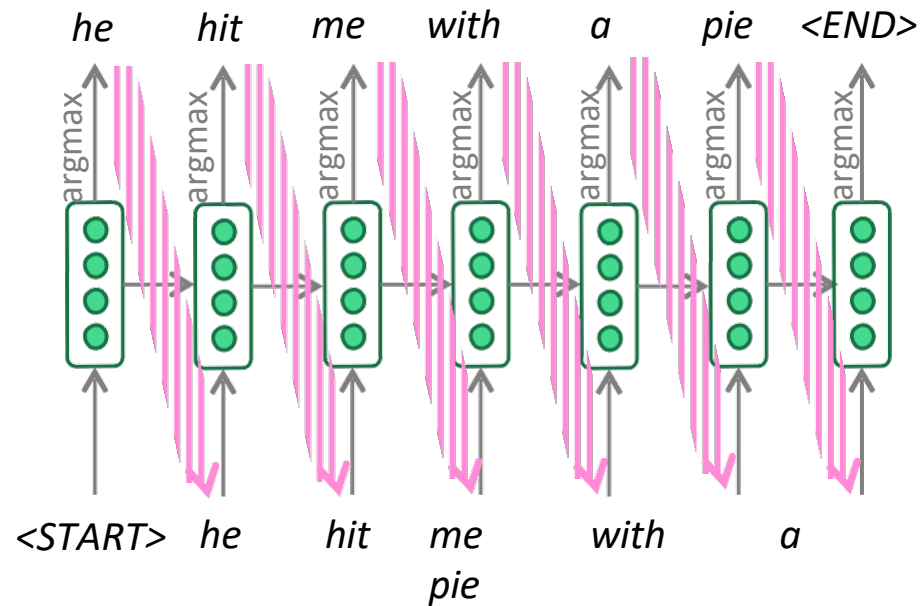
- For neural generation, we can sample from the softmax distribution.
- In  MT where we're looking for a specific output sequence, sampling isn't useful.
- Greedy Decoding: we choose the most likely output at each time step by taking the argmax over the softmax output

$$\hat{y} = \operatorname{argmax} P(y_i / y_{<i})$$

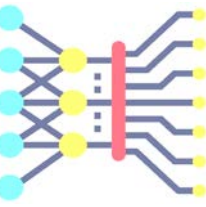
# Greedy decoding



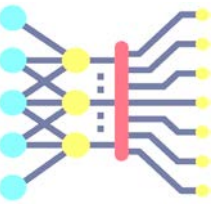
•



greedy decoding : take most probable word on each step

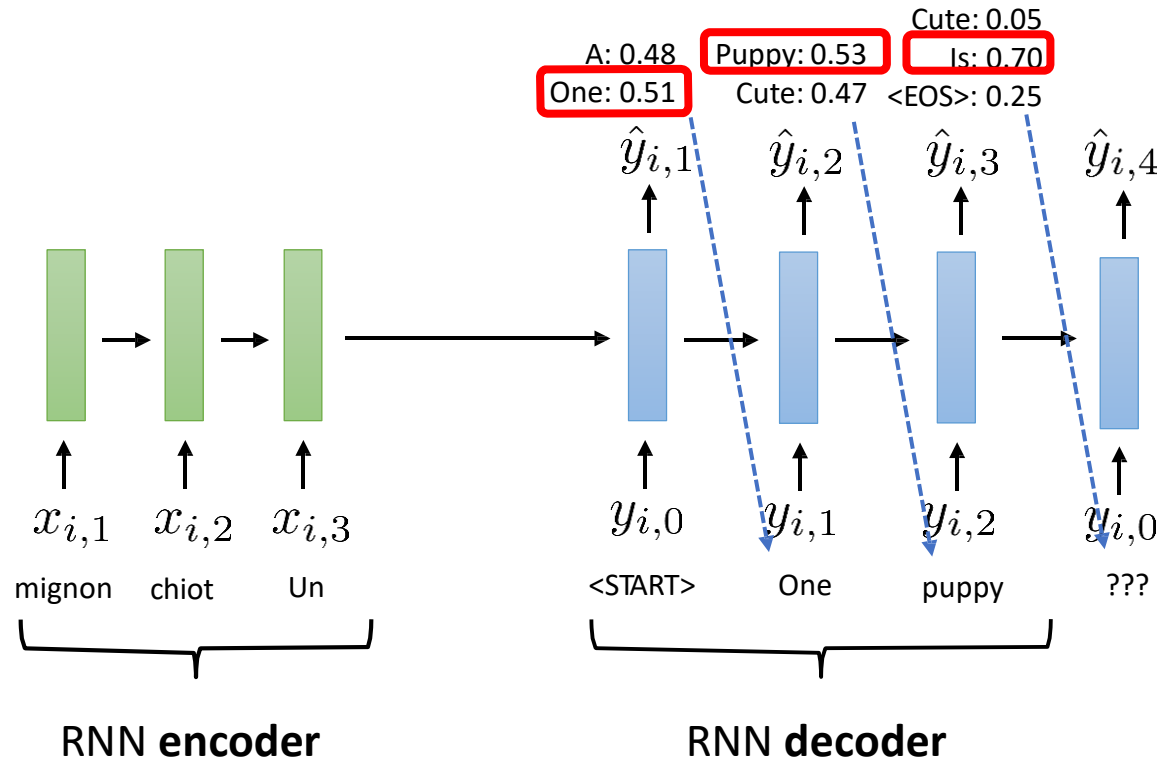


# Decoding with beam search

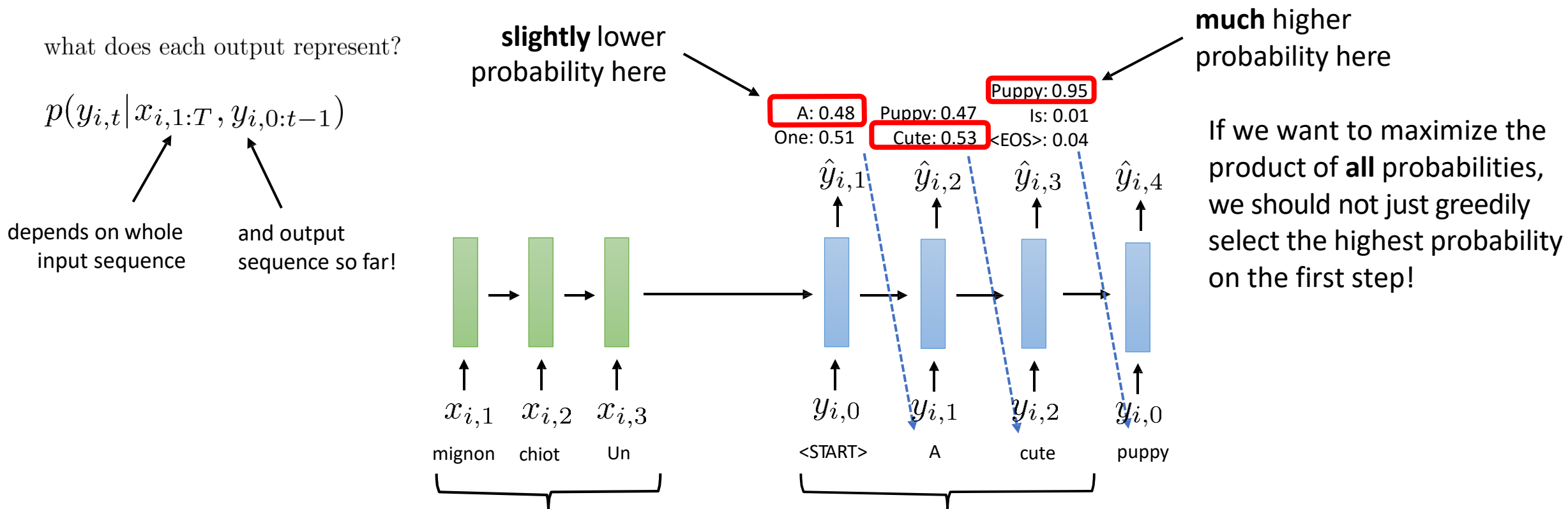
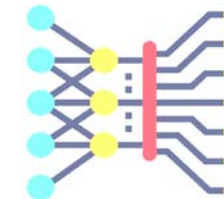


# Decoding the most likely sequence

notice we feed  
this in reverse



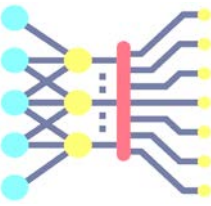
# What we *should* have done



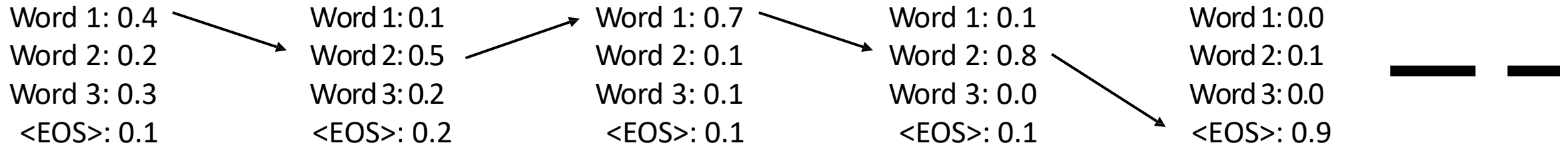
$$p(y_{i,1:T_y} | x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

probabilities at each time step





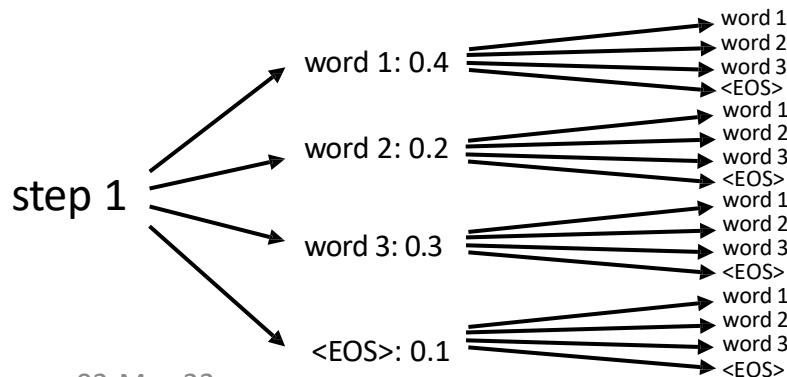
# How many possible decodings are there?



for  $M$  words, in general there are  $M^T$  sequences  
of length  $T$

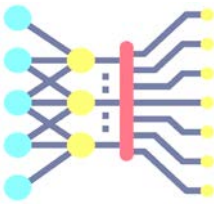
*any* one of these might be the optimal one!

Decoding is a **search** problem

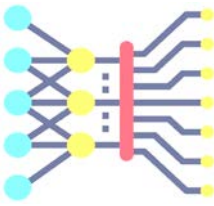


We could use *any* tree search  
algorithm But exact search in this  
case is **very** expensive  
The **structure** of this problem makes  
some simple **approximate search**  
methods work **very well**

# Beam search decoding



- Core idea: On each step of decoder, keep track of the ***k* most probable** partial translations (which we call ***hypotheses***)
  - *k* is the **beam size** (in practice around 5 to 10)
- A hypothesis  $y_1, \dots, y_t$  has a score which is its log probability:
$$\text{score}(y_1, \dots, y_t) = \log P_{\text{LM}}(y_1, \dots, y_t | x) = \sum_{i=1}^t \log P_{\text{LM}}(y_i | y_1, \dots, y_{i-1}, x)$$
  - Scores are negative, and higher score is better
  - We search for high-scoring hypotheses, tracking top *k* on each step
- Beam search is **not guaranteed** to find optimal solution
- But **much more efficient** than exhaustive search!



# Decoding with approximate search

**Basic intuition:** while choosing the **highest-probability** word on the first step may not be optimal, choosing a **very low-probability** word is very unlikely to lead to a good result

**Equivalently:** we can't be greedy, but we can be *somewhat greedy*.

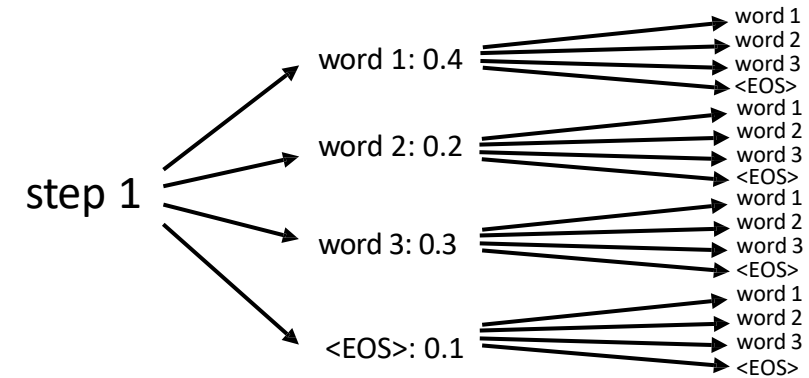
**Beam search** intuition: store the **k** best sequences **so far**, and update each of them.

special case of **k** = 1 is just

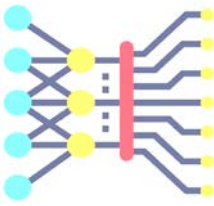
greedy decoding often use **k**

around 5-10

Decoding is a **search** problem



# Beam search example



$$p(y_{i,1:T_y} | x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

$$\log p(y_{i,1:T_y} | x_{i,1:T}) = \sum_{t=1}^{T_y} \log p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

in practice, we **sum up** the log probabilities as we go (to avoid underflow)

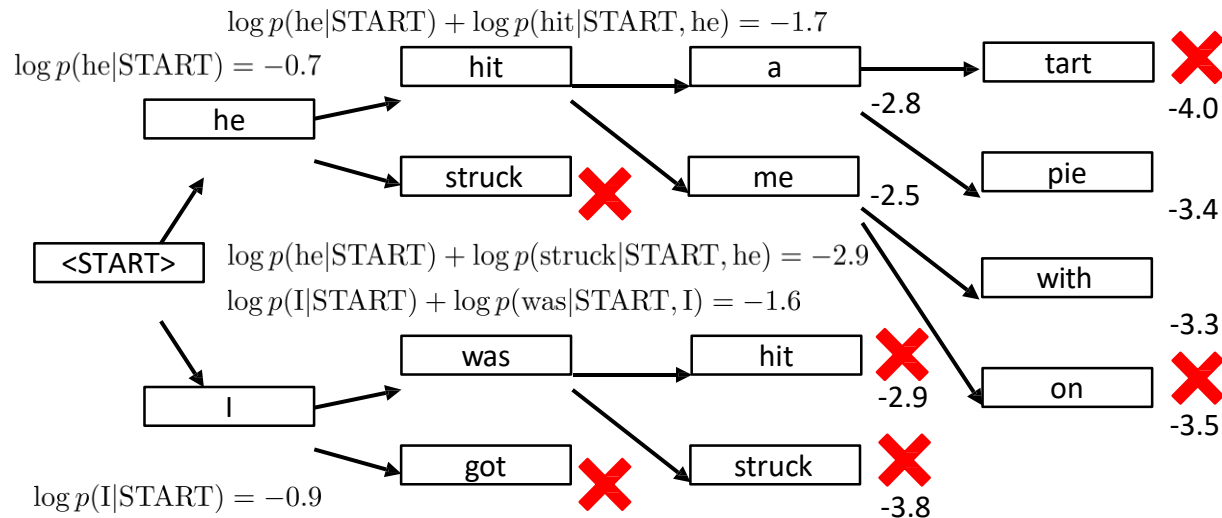
## Example

**k** = 2 (track the 2 most likely hypotheses)

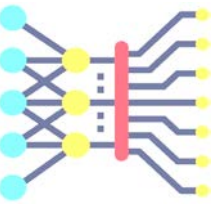
translate (Fr->En): il a m'entarté

(he hit me with a pie)

no perfectly equivalent English word, makes this hard



...and many other choices with lower log-prob



# Beam search summary

$$\log p(y_{i,1:T_y} | x_{i,1:T}) = \sum_{t=1}^{T_y} \log p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

there are  $k$  of these

at each time step  $t$ :

1. for each hypothesis  $y_{1:t-1,i}$  that we are tracking:

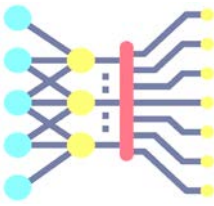
find the top  $k$  tokens  $y_{t,i,1}, \dots, y_{t,i,k}$

very easy, we get this from the softmax log-probs

2. sort the resulting  $k^2$  length  $t$  sequences by their *total* log-probability

3. keep the top  $k$

4. advance each hypothesis to time  $t + 1$

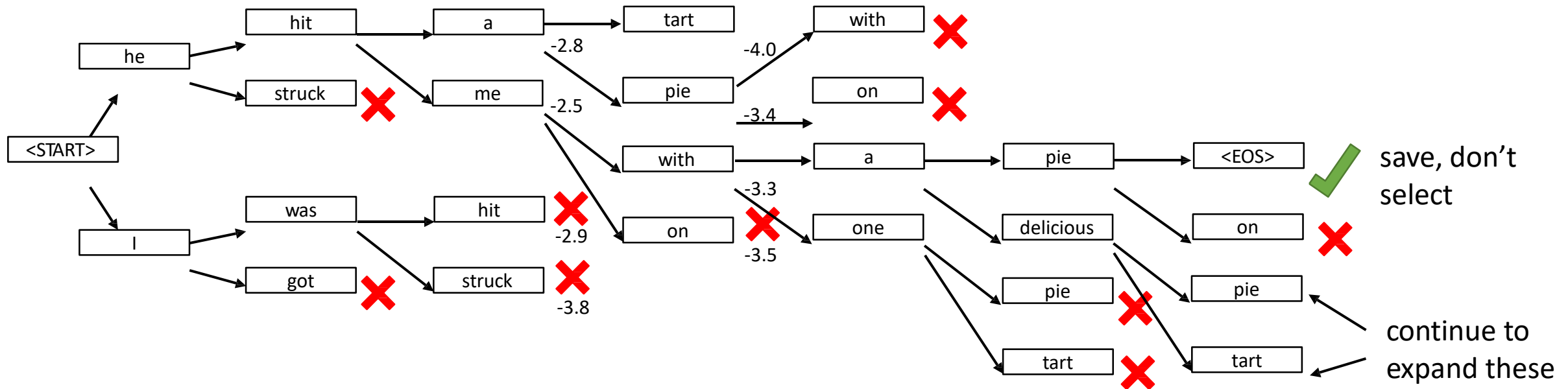


# When do we stop decoding?

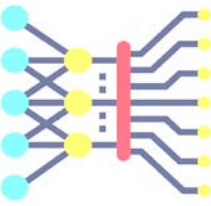
Suppose one of the highest-scoring hypotheses ends in <END>

Save it, along with its score, but do **not** pick it to expand further (there is nothing to expand)

Keep expanding the **k** remaining best hypotheses



Continue until either some cutoff length **T** or until we have **N** hypotheses that end in <EOS>



# Which sequence do we pick?

At the end we might have something like this:

he hit me with a pie he  $\log p = -4.5$

threw a pie  $\log p = -3.2$

I was hit with a pie that he threw  $\log p = -7.2$

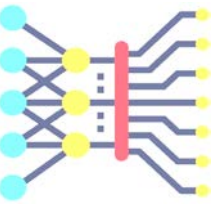
$$\log p(y_{i,1:T}|x_{i,1:T}) = \sum_{t=1}^T \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

**Problem:**  $p < 1$  **always**, hence  $\log p < 0$  **always**

The **longer** the sequence the **lower** its total score (more negative numbers added together)

Simple “fix”: just divide by sequence length

$$\text{score}(y_{i,1:T}|x_{i,1:T}) = \frac{1}{T} \sum_{t=1}^T \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$



# Beam search summary

$$\text{score}(y_{i,1:T}|x_{i,1:T}) = \frac{1}{T} \sum_{t=1}^T \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

at each time step  $t$ :

1. for each hypothesis  $y_{1:t-1,i}$  that we are tracking:  
find the top  $k$  tokens  $y_{t,i,1}, \dots, y_{t,i,k}$
2. sort the resulting  $k^2$  length  $t$  sequences by their *total* log-probability
3. save any sequences that end in EOS
4. keep the top  $k$
5. advance each hypothesis to time  $t + 1$  if  $t < H$

return saved sequence with highest score