Midterm Review

COS 484

Max Gupta, Ambri Ma

Topics

1. Word embeddings (5 min)
2. Neural Networks for NLP (feedforward) (5 min)
3. Sequence Models (HMMs) (5 min)
4. RNNs/LSTMs (15 min)
5. Encoder/decoder models + Attention (10 min)
6. Transformers (10 min)
7. Pretraining (Elmo, GPT, BERT) (10 min)

Basics: Probability

$\Pr[A] = P(\text{all outcomes in } A)$
$\Pr[\bar{A}] = 1 - \Pr[A]$

Addition rule:
$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$

Chain rule:
$\Pr[AB] = \Pr[B]\Pr[A \mid B]$

For k events:
$\Pr[A_1A_2\ldots A_k] = \Pr[A_1]\Pr[A_2 \mid A_1]\Pr[A_3 \mid A_1A_2]\cdots\Pr[A_k \mid A_1A_2\ldots A_{k-1}]$

Events A, B are independent if $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$
Independence also implies $\Pr[A \mid B] = \Pr[A]$ and $\Pr[B \mid A] = \Pr[B]$

Bayes rule:

$$Pr[A|B] = \frac{Pr[B|A]Pr[A]}{Pr[B]}$$

Law of total Probability:

$$Pr[B] = \sum_i Pr[B|A_i]Pr[A_i]$$

If $\sum_i Pr[A_i] = 1$

Slides from Midterm Review sp24

Basics: Exponents, Logs and Sums

Exponential Laws	Logarithm Laws
$x^a \cdot x^b = x^{a+b}$	$\log(ab) = \log(a) + \log(b)$
$\frac{x^a}{x^b} = x^{a-b}$	$\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
$(x^a)^b = x^{ab}$	$\log(a^b) = b \cdot \log(a)$
$x^{-a} = \frac{1}{x^a}$	$\log_x\left(\frac{1}{x^a}\right) = -a$
$x^0 = 1$	$\log_x 1 = 0$

$$e^{\log_e x} =$$

$$\sum_i (x_i + y_i) = \sum_i x_i + \sum_i y_i$$

$$\sum_i \sum_j x_{ij} = \sum_j \sum_i x_{ij}$$

$$\sum_{i=1}^{n} x_i = \sum_{i odd} x_i + \sum_{i even}$$

Word Embeddings

Skip-gram

For each position $t = 1,2,\ldots T$, predict context words within context size m, given center word w_t:

all the parameters to be optimized

$$\mathcal{L}(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w_{t+j} \mid w_t; \theta)$$

It is equivalent to minimizing the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log \mathcal{L}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j} \mid w_t; \theta)$$

Skip-gram

How to define $P(w_{t+j} \mid w_t; \theta)$?

- Use two sets of vectors for each word in the vocabulary

$$\mathbf{u}_a \in \mathbb{R}^d \quad \text{: vector for center word } a, \forall a \in V$$

$$\mathbf{v}_b \in \mathbb{R}^d \quad \text{: vector for context word } b, \forall b \in V$$

- Use inner product $\mathbf{u}_a \cdot \mathbf{v}_b$ to measure how likely word a appears with context word b

$$P(w_{t+j} | w_t) = \frac{\exp\left(\mathbf{u}_{w_t} \cdot \mathbf{v}_{w_{t+j}}\right)}{\sum_{k \in V} \exp\left(\mathbf{u}_{w_t} \cdot \mathbf{v}_k\right)}$$

Does this term seem familiar?

word2vec

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m, j\neq 0}\log\frac{\exp(\mathbf{u}_{w_t}\cdot\mathbf{v}_{w_{t+j}})}{\sum_{k\in V}\exp(\mathbf{u}_{w_t}\cdot\mathbf{v}_k)}$$

Q: Why do we need two vectors for each word?

- Because one word is not likely to appear in its own context window, e.g., $P(\text{dog} \mid \text{dog})$ should be low. If we use one set of vectors only, it essentially needs to minimize $\mathbf{u}_{\text{dog}} \cdot \mathbf{u}_{\text{dog}}$

Q: Which set of vectors are used as word embeddings?

- This is an empirical question. Typically just $\mathbf{u}_w$ but you can also concatenate the two vectors..

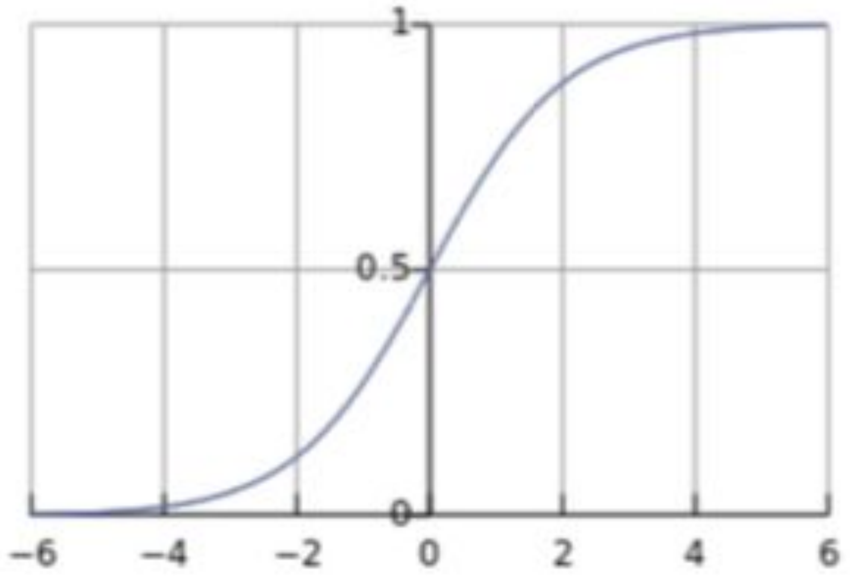
Skip-gram w/ negative sampling (SGNS)

Problem: every time you get one pair of (t, c), you need to update $\mathbf{v}_k$ with all the words in the vocabulary! This is very expensive computationally.

$$\frac{\partial y}{\partial \mathbf{u}_t} = -\mathbf{v}_c + \sum_{k \in V} P(k \mid t)\mathbf{v}_k \quad ; \quad \frac{\partial y}{\partial \mathbf{v}_k} = \begin{cases} (P(k \mid t) - 1)\ \mathbf{u}_t & k = c \\ P(k \mid t)\ \mathbf{u}_t & k \neq c \end{cases}$$

Negative sampling: instead of considering all the words in V, let's randomly sample K (5-20) negative examples.

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



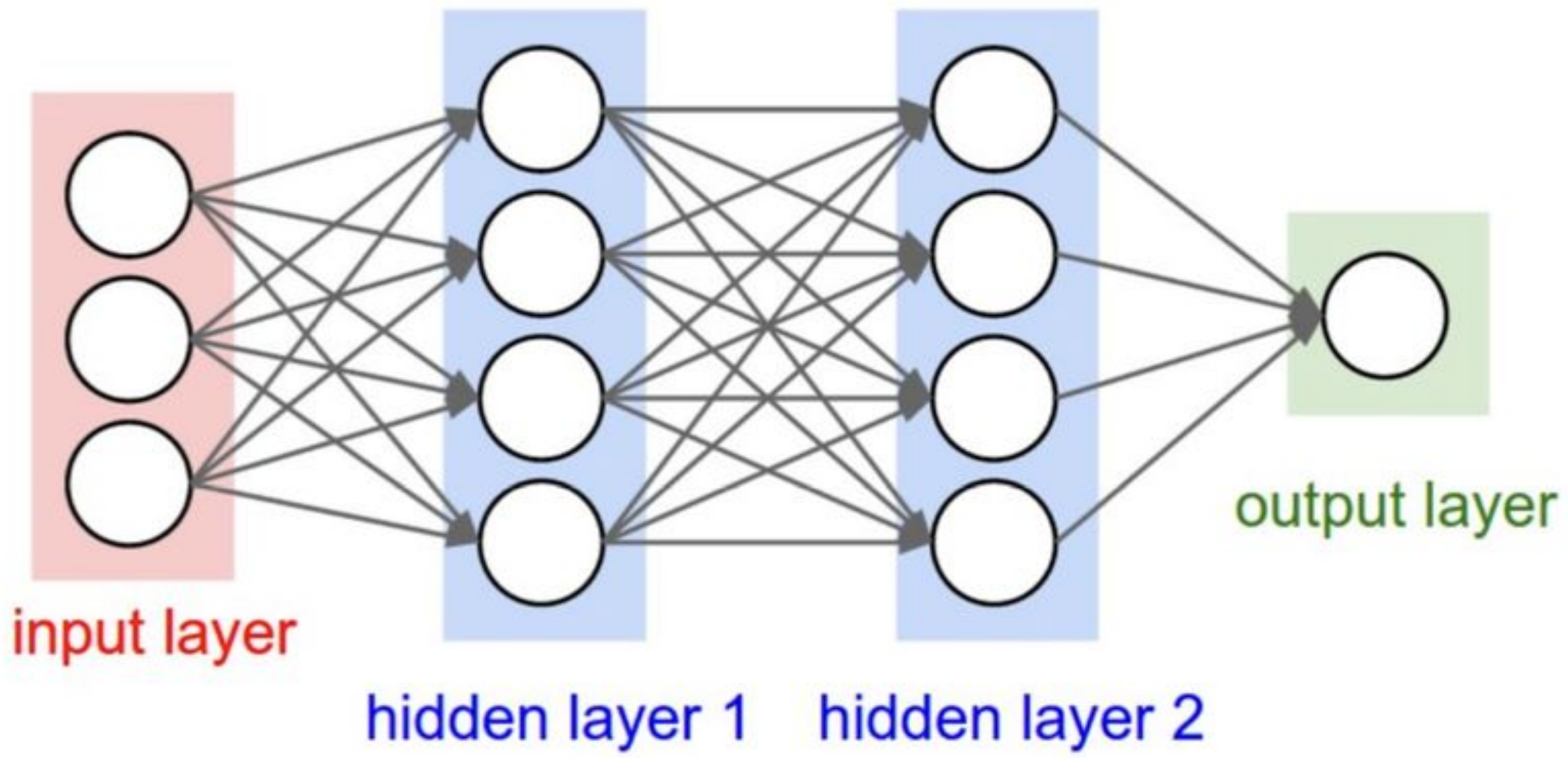
Softmax: $$y = -\log\left(\frac{\exp\left(\mathbf{u}_t \cdot \mathbf{v}_c\right)}{\sum_{k \in V} \exp\left(\mathbf{u}_t \cdot \mathbf{v}_k\right)}\right)$$

Negative sampling: $$y = -\log\left(\sigma(\mathbf{u}_t \cdot \mathbf{v}_c)\right) - \sum_{i=1}^{K} \mathbb{E}_{j \sim P(w)} \log\left(\sigma(-\mathbf{u}_t \cdot \mathbf{v}_j)\right)$$

Neural Networks for NLP (feedforward)

Feed forward neural networks (FFNNs)

- The units are connected with no cycles
- The outputs from units in each layer are passed to units in the next higher layer
- No outputs are passed back to lower layers



Fully-connected (FC) layers:

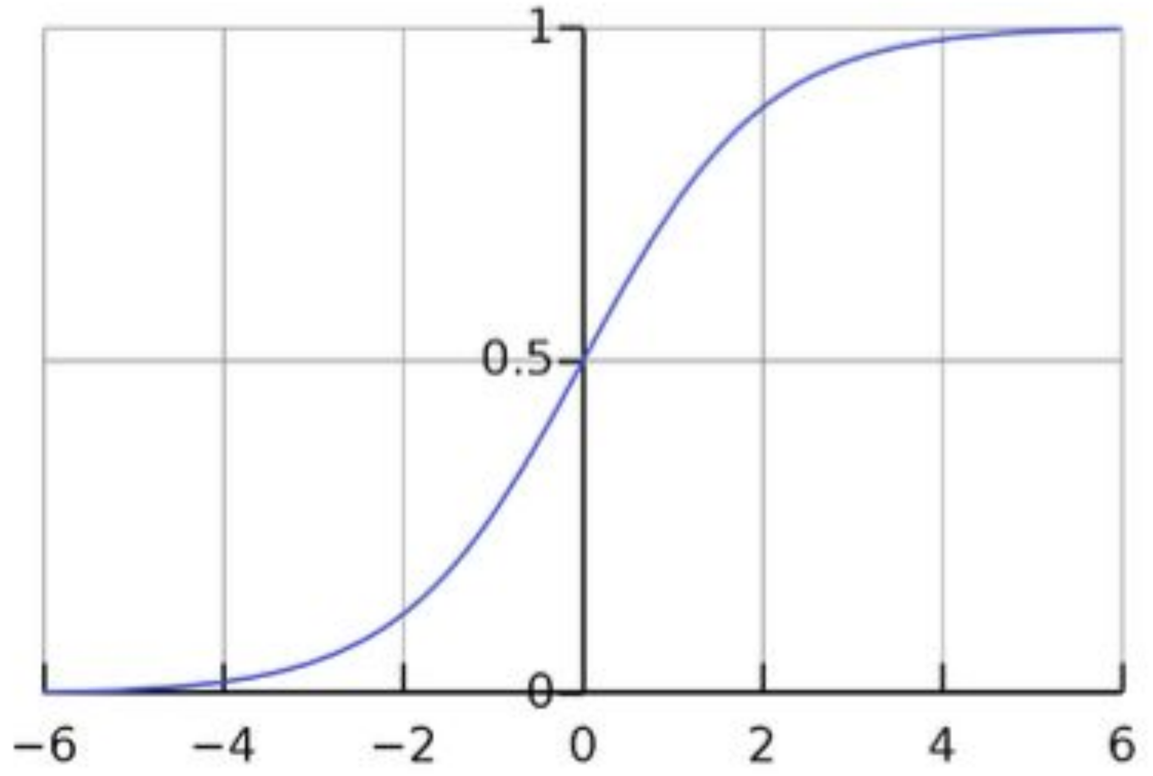
All the units from one layer are fully connected to every unit of the next layer.

Feed forward neural networks (FFNNs)

Activation functions

sigmoid

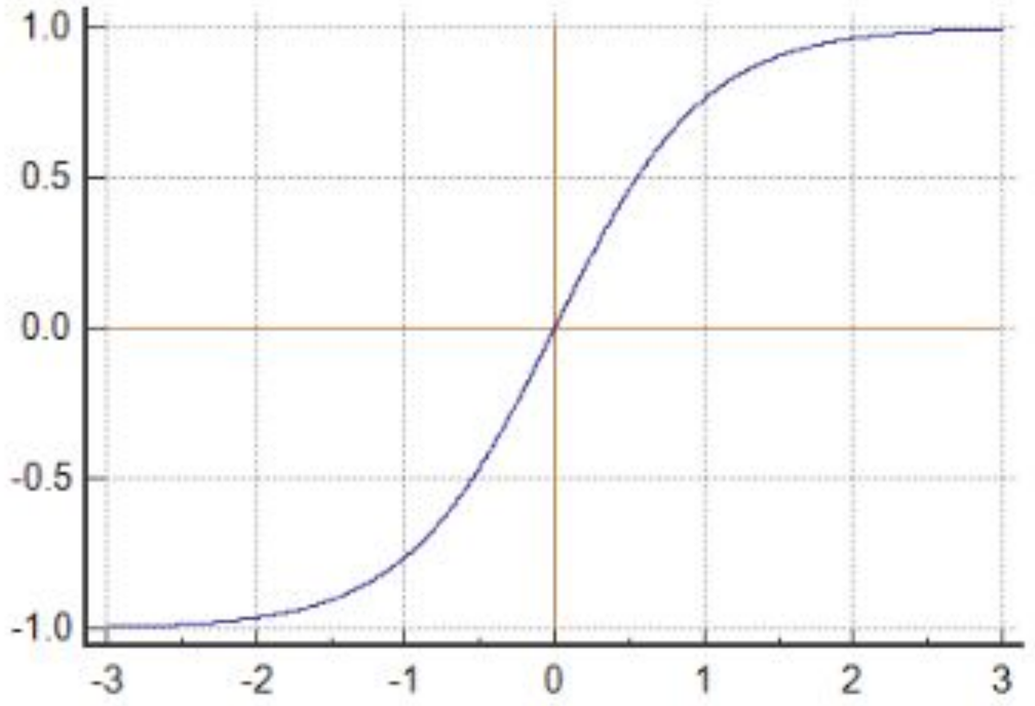
$$f(z) = \frac{1}{1 + e^{-z}}$$



$$f'(z) = f(z) \times (1 - f(z))$$

tanh

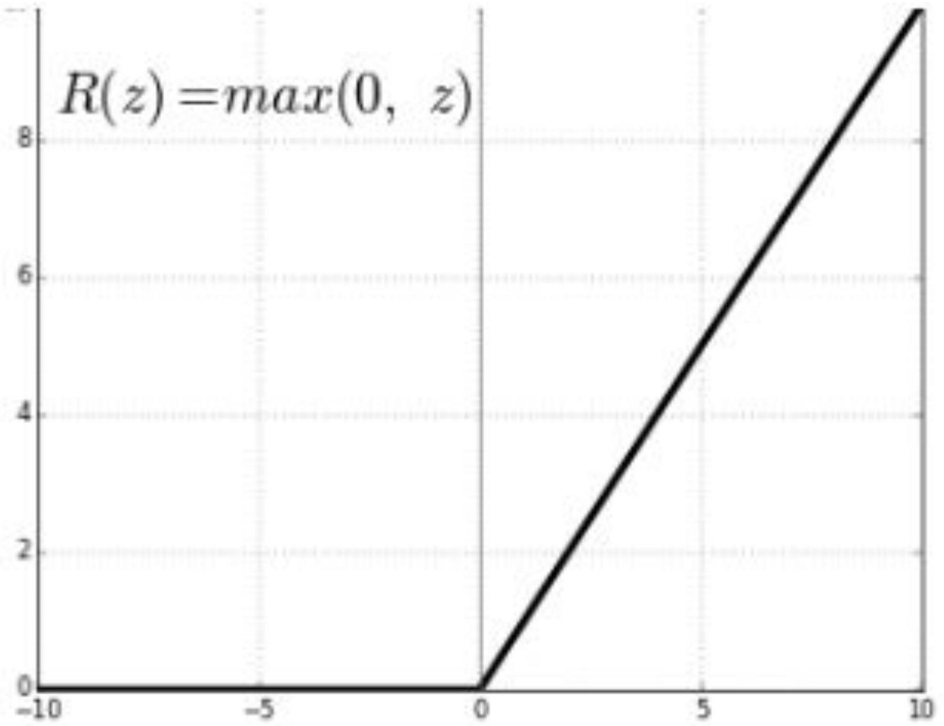
$$f(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$



$$f'(z) = 1 - f(z)^2$$

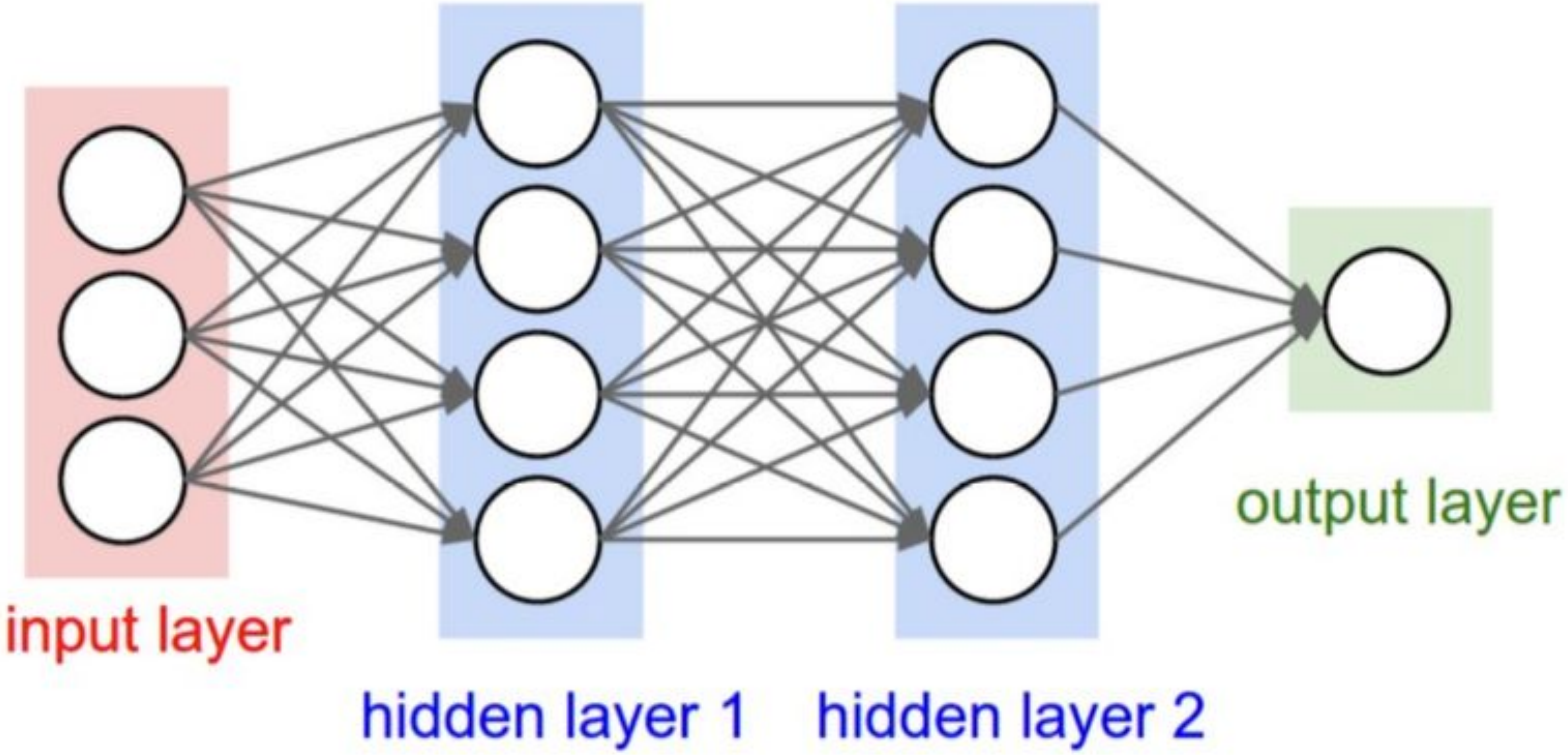
ReLU (rectified linear unit)

$$f(z) = \max(0, z)$$



$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

Feed forward neural networks (FFNNs)



- Input layer: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer 1:

$$\mathbf{h}_1 = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \in \mathbb{R}^{d_1}$$

$$\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}, \mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$$

- Hidden layer 2:

$$\mathbf{h}_2 = f(\mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)}) \in \mathbb{R}^{d_2}$$

$$\mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1}, \mathbf{b}^{(2)} \in \mathbb{R}^{d_2}$$

- Output layer:

$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}$$

*: *f* is applied element-wise

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

C: number of classes

d: input dimension, d_1, d_2: hidden dimensions

Feed forward neural networks (FFNNs)

Forward propagation:
from input to output layer

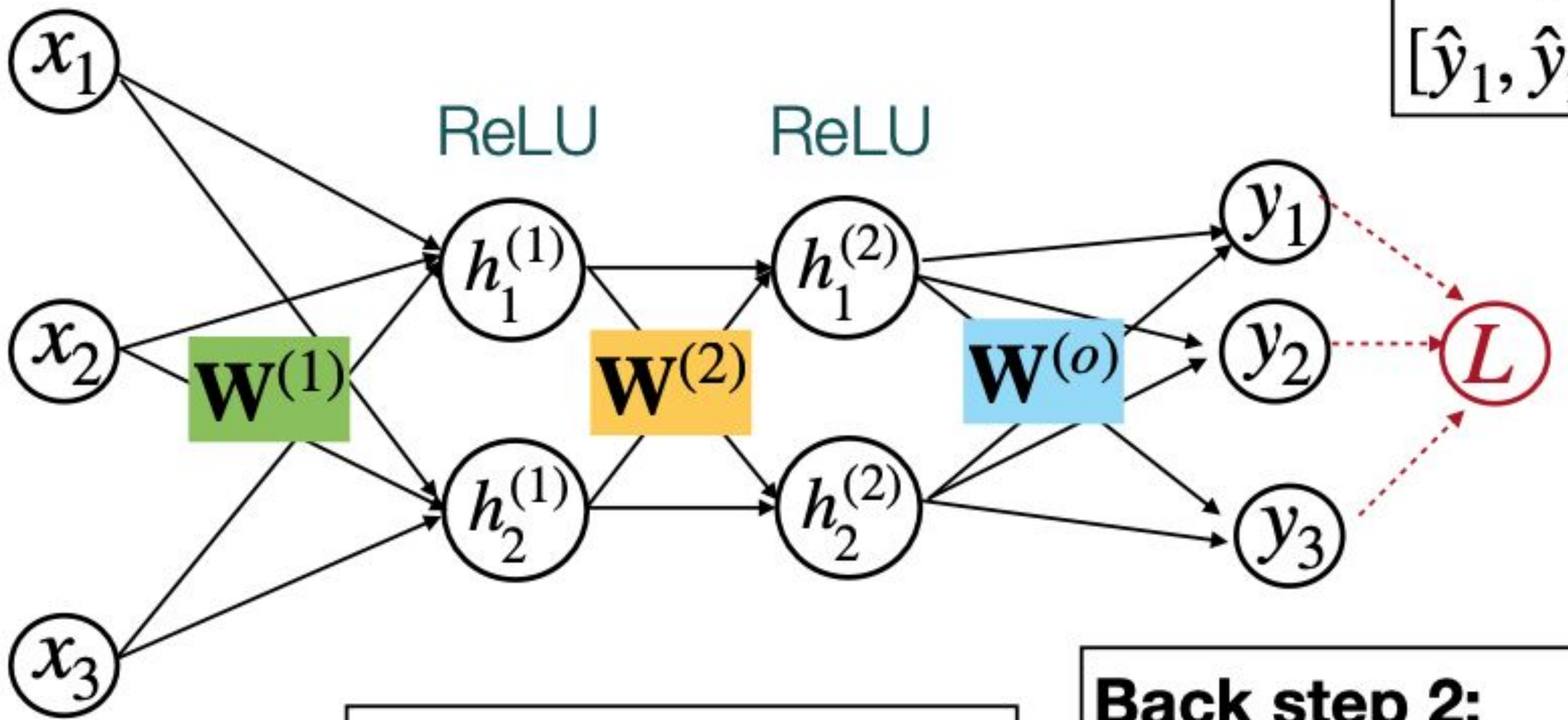
Given: x_1, x_2, x_3 and the class label y
(a single training example)

Forward step 1:
Compute $h_1^{(1)}, h_2^{(1)}$

Forward step 2:
Compute $h_1^{(2)}, h_2^{(2)}$

Forward step 3:
Compute y_1, y_2, y_3 and
$[\hat{y}_1, \hat{y}_2, \hat{y}_3] = \text{softmax}[y_1, y_2, y_3]$

Forward step 4:
Compute loss
$L = -\log \hat{\mathbf{y}}_y$



Back propagation:
from output to input layer

Goal:
$\frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial W^{(o)}}$

Back step 4:
Compute
$\frac{\partial L}{\partial W^{(1)}}$

Back step 3:
Compute
$\frac{\partial L}{\partial h_1^{(1)}}, \frac{\partial L}{\partial h_2^{(1)}}, \frac{\partial L}{\partial W^{(2)}}$

Back step 2:
Compute
$\frac{\partial L}{\partial h_1^{(2)}}, \frac{\partial L}{\partial h_2^{(2)}}, \frac{\partial L}{\partial W^{(o)}}$

Back step 1:
Compute
$\frac{\partial L}{\partial y_1}, \frac{\partial L}{\partial y_2}, \frac{\partial L}{\partial y_3}$

Sequence Models (HMMs)

Named Entity Recognition (NERs)

- Tag each word in a sentence with its part of speech
- Disambiguation task: each word might have different functions in different contexts
  - The/DT man/NN bought/VBD a/DT boat/NN
  - The/DT old/NN man/VBP the/DT boat/NN

Same word, different tags

earnings growth took a **back/JJ** seat
a small building in the **back/NN**
a clear majority of senators **back/VBP** the bill
Dave began to **back/VB** toward the door
enable the country to buy **back/RP** about debt
I was twenty-one **back/RB** then

Some words have many functions!

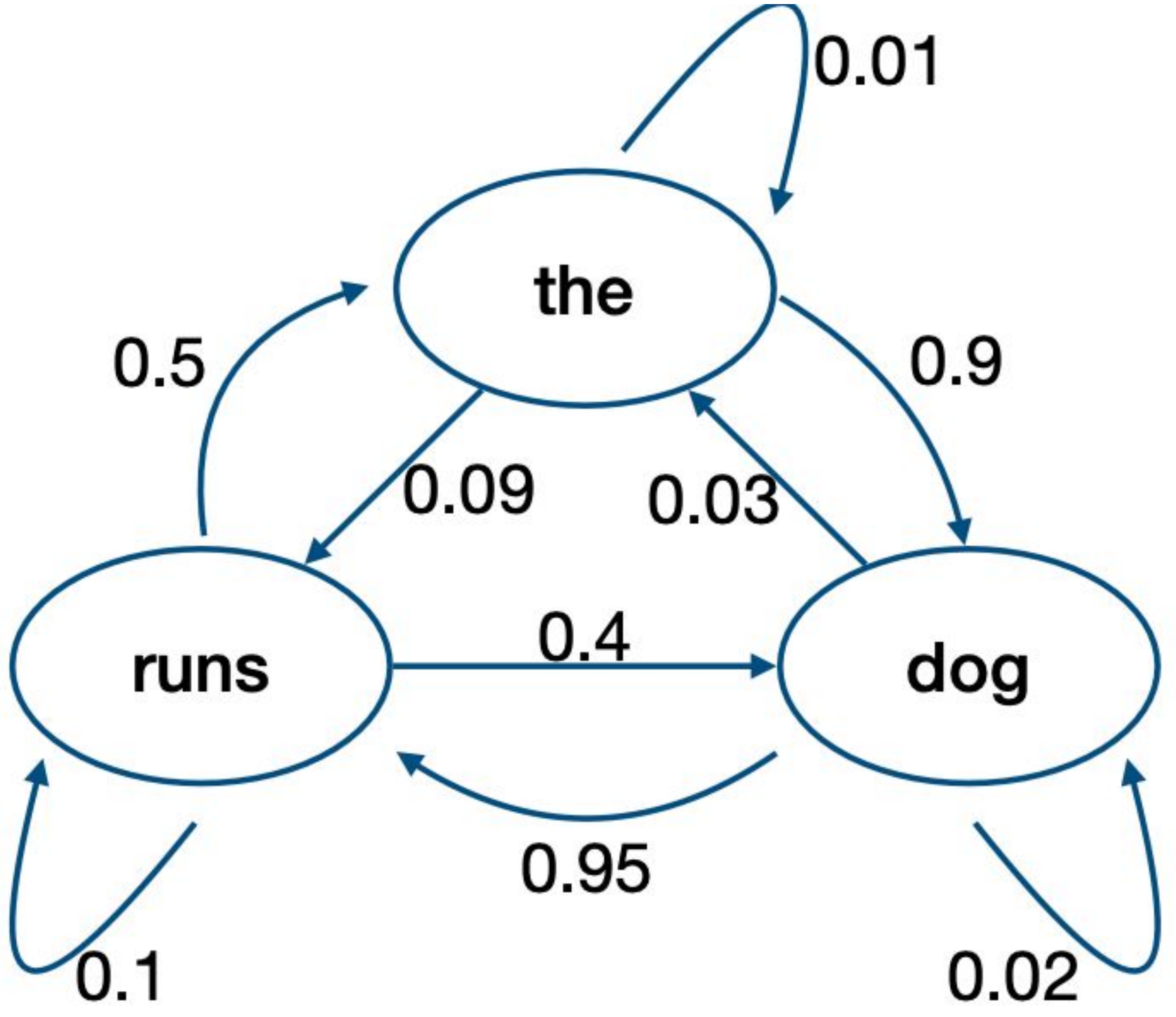
JJ: adjective, NN: single or mass noun, VBP: Verb, non-3rd person singular present
VB: Verb, base form, RP: particle, RB: adverb

Hidden Markov Models (HMMs)

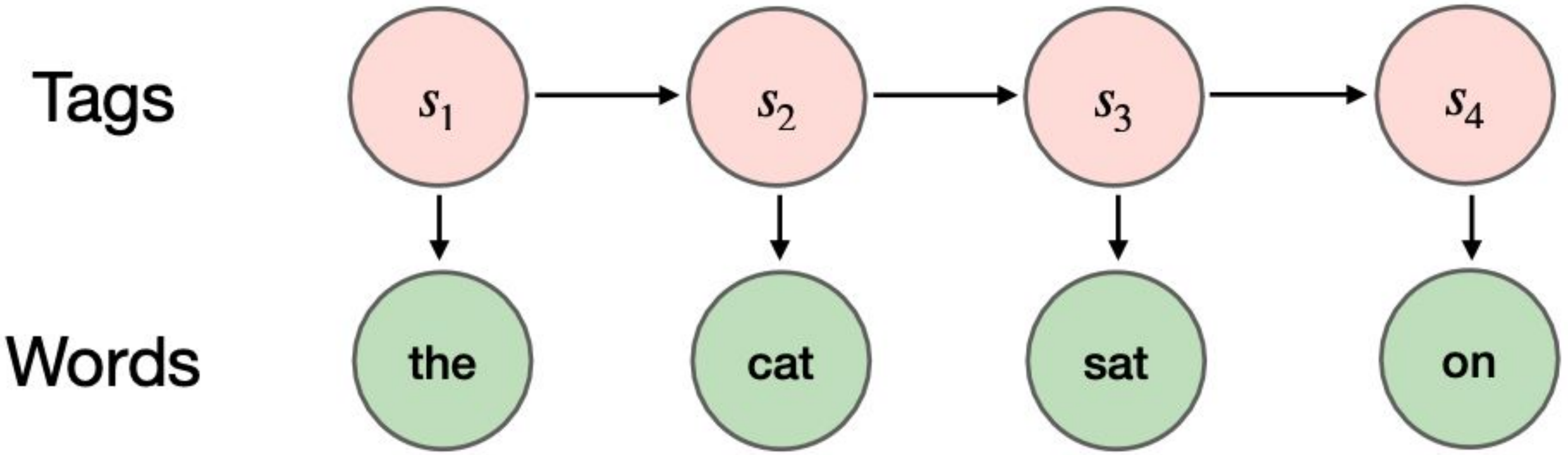
- Each state can take one of K values (can assume {1, 2, ..., K} for simplicity)
- Markov assumption:

  $$P(s_t \mid s_1, s_2, \ldots, s_{t-1}) \approx P(s_t \mid s_{t-1})$$

- A Markov chain is specified by
  - Initial probability distribution $\pi(s), \forall s \in \{1, \ldots, K\}$
  - Transition probability matrix ($K \times K$)



Hidden Markov Models (HMMs)



Markov assumption:

$$P(s_t \mid s_1, \ldots, s_{t-1}) \approx P(s_t \mid s_{t-1})$$

Output independence:

$$P(o_t \mid s_1, \ldots, s_t) \approx P(o_t \mid s_t)$$

1. Set of states S = {1, 2, ..., K} and set of observations $O = \{o_1, \ldots, o_n\}$ $\quad o_i \in V$
2. **Initial state probability distribution** $\pi(s_1)$
3. **Transition probabilities** $P(s_{t+1} \mid s_t)$
4. **Emission probabilities** $P(o_t \mid s_t)$

$$P(S, O) = P(s_1, s_2, \ldots, s_n, o_1, o_2, \ldots, o_n)$$
$$= \pi(s_1)p(o_1 \mid s_1)\prod_{i=2}^{n} P(s_i \mid s_{i-1})P(o_i \mid s_i)$$

transition probability — emission probability

If we add a dummy state $s_0 = \varnothing$ at the beginning,

$$P(S, O) = \prod_{i=1}^{n} P(s_i \mid s_{i-1})P(o_i \mid s_i) \quad [\pi(s_1) = P(s_1 \mid \varnothing)]$$

Hidden Markov Models (HMMs)

Maximum likelihood estimates:

1. The/DT cat/NN sat/VBD on/IN the/DT mat/NN
2. Princeton/NNP is/VBZ in/IN New/NNP Jersey/NNP
3. The/DT old/NN man/VBP the/DT boat/NN

$$P(s_i \mid s_j) = \frac{Count(s_j, s_i)}{Count(s_j)}$$

$$P(o \mid s) = \frac{Count(s, o)}{Count(s)}$$

$\pi(DT) = P(DT \mid \varnothing) =$ 2/3

$P(NN \mid DT) =$ 4/4 $\quad P(DT \mid IN) =$ 1/2

$P(cat \mid NN) =$ 1/4 $\quad P(the \mid DT) =$ 2/4

(assuming we differentiate cased vs uncased words)

Decoding HMMs

Task: Find the most probable sequence of states $S = s_1, s_2, \ldots, s_n$ given the observations $O = o_1, o_2, \ldots, o_n$

$$\hat{S} = \arg\max_{S} P(S \mid O) = \arg\max_{S} \frac{P(O \mid S)P(S)}{P(O)}$$ [Bayes' rule]

$$= \arg\max_{S} P(O \mid S)P(S)$$ [$P(O)$ doesn't depend on S!]

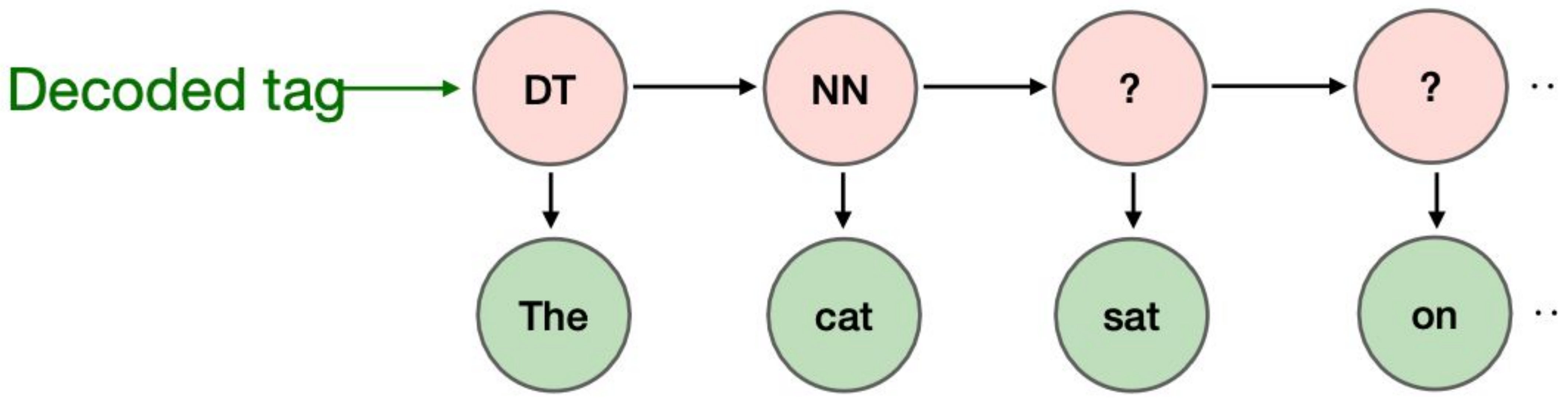
$$= \arg\max_{s_1, s_2, \ldots s_n} \prod_{i=1}^{n} P(o_i \mid s_i) P(s_i \mid s_{i-1})$$ [Markov assumption]

How can we maximize this?
Search over all state sequences?

Decoding

Greedy search

- Idea: Decode one state at at time



- In general, $\hat{s}_t = \arg\max_s p(s \mid \hat{s}_{t-1}) p(o_t \mid s)$
- Very efficient, but not guaranteed to be optimum!

3. The/DT old/NN man/VBP the/DT boat/NN

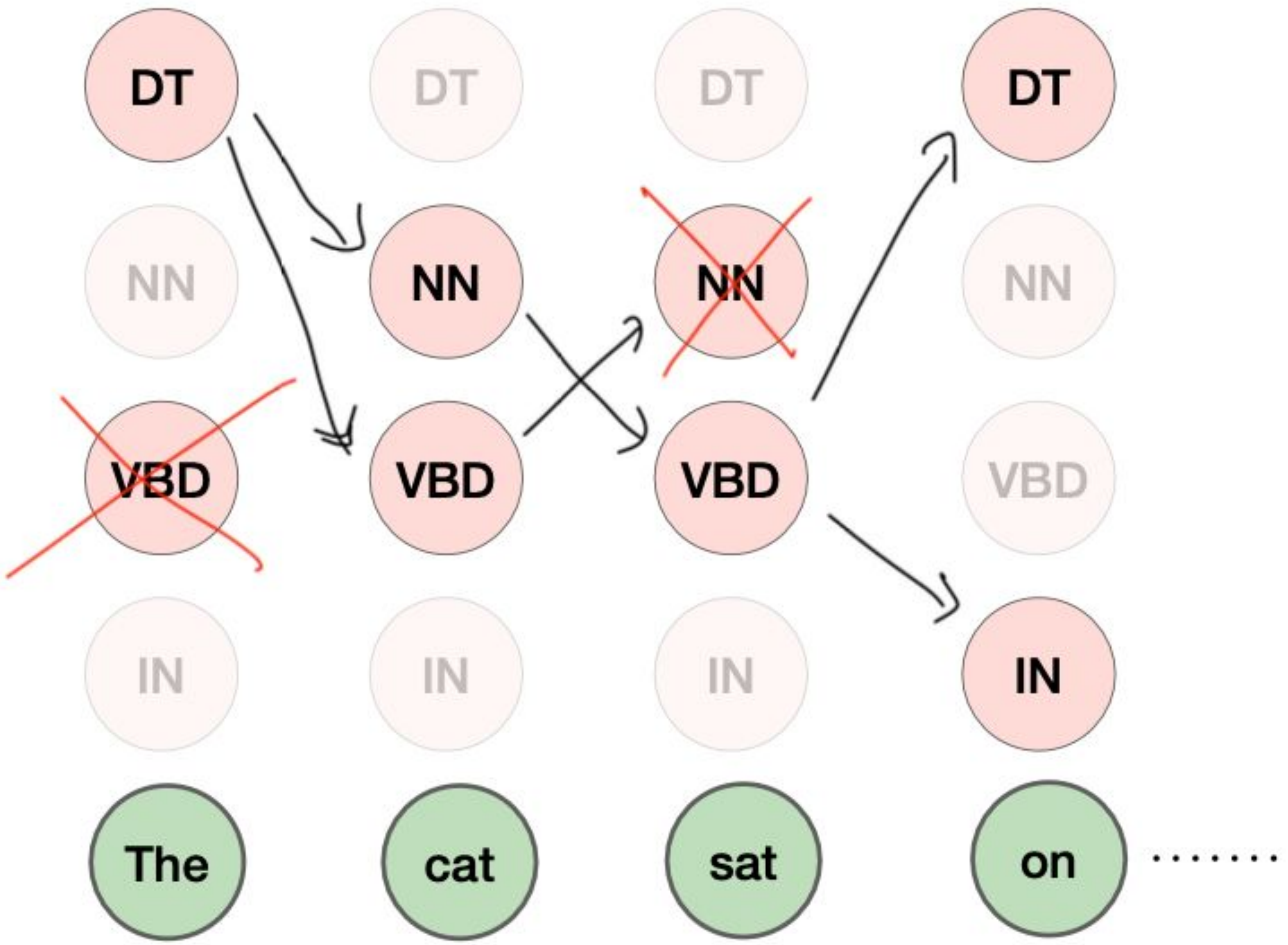
Viterbi decoding

- Use dynamic programming!
- Maintain some extra data structures
- Probability lattice, $M[T, K]$ and backtracking matrix, $B[T, K]$
  - T : Number of time steps
  - K : Number of states
- $M[i, j]$ stores joint probability of most probable sequence of states ending with state j at time i,
- $B[i, j]$ is the tag at time i-1 in the most probable sequence ending with tag j at time i

Refer to Precept 3 slides for a concrete example!

Decoding

- If K (number of possible hidden states) is too large, Viterbi is too expensive!
- **Observation:** Many paths have very low likelihood!
- Keep a fixed number of hypotheses at each point
  - Beam width = β



Pick $\max_k M[n, k]$ from within beam and backtrack

Viterbi Decoding

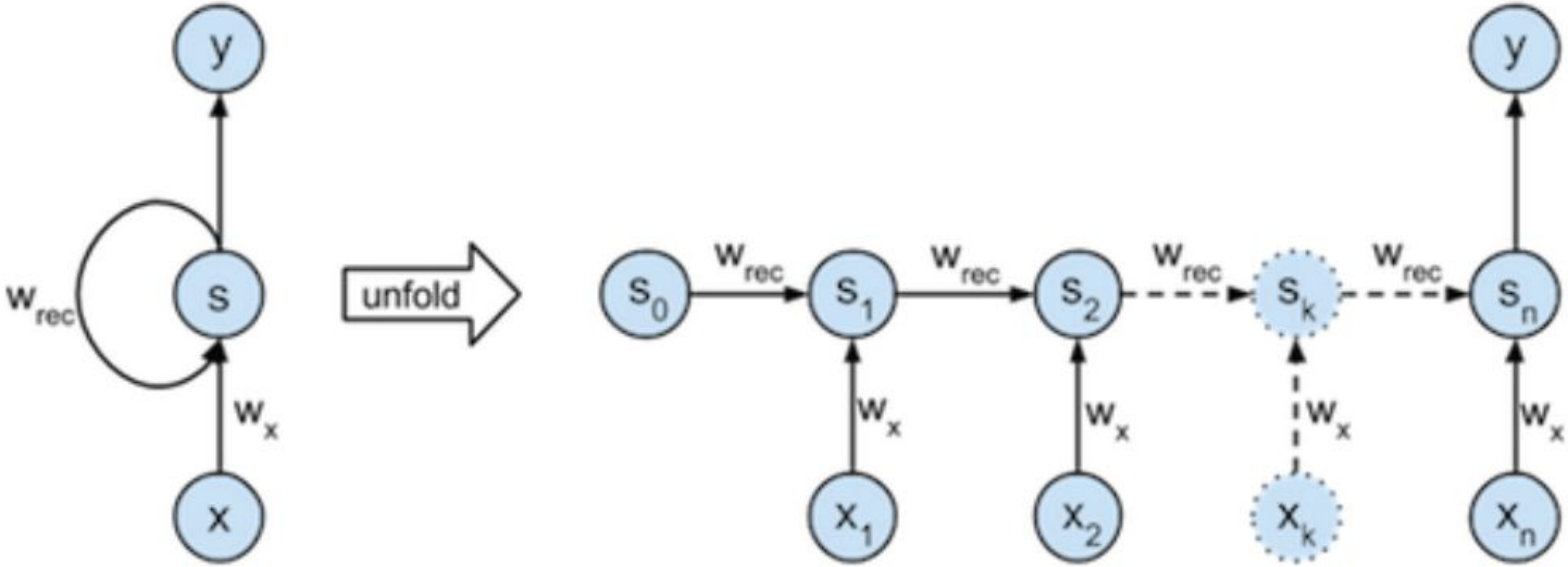
- Complexity: $O(nK^2)$
  - Very expensive if K is large

- Beam search: tradeoff between accuracy and efficiency
  - Set $K = \beta$ fixed (beam width): only keep track a few best sequences so far instead of exploring the entire space
  - Complexity: $O(nK\beta)$

RNNs/LSTMs

Recurrent neural networks (RNNs)

- Handles variable length inputs



A function: $\mathbf{y} = \text{RNN}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \in \mathbb{R}^h$ where $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$

Recurrent neural networks (RNNs)

A function: $\mathbf{y} = \text{RNN}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \in \mathbb{R}^h$ where $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$

$\mathbf{h}_0 \in \mathbb{R}^h$ is an initial state

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h$$

$\mathbf{h}_t$: hidden states which store information from $\mathbf{x}_1$ to $\mathbf{x}_t$

Simple RNNs:

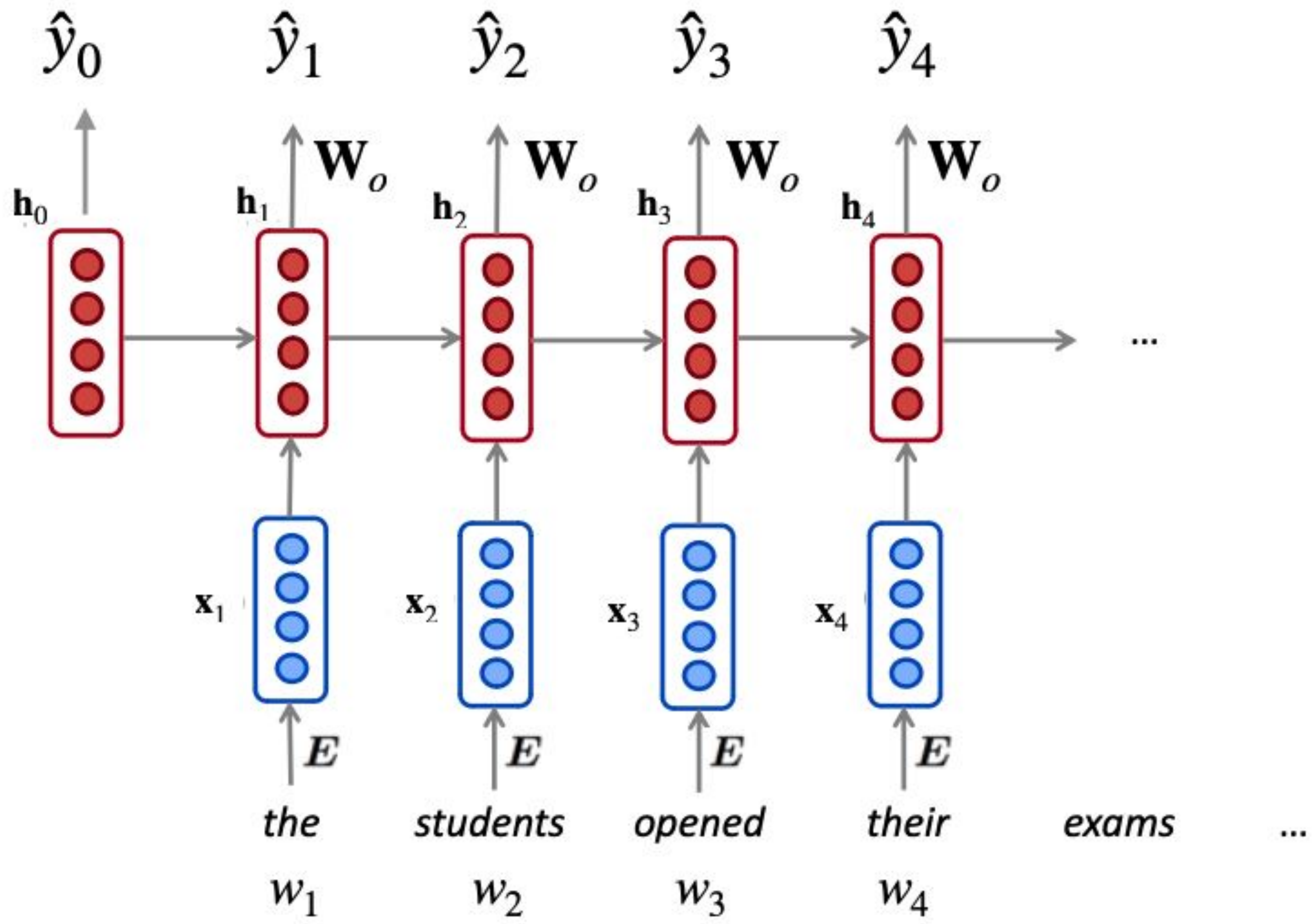
$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \in \mathbb{R}^h$$

g: nonlinearity (e.g. tanh, ReLU),

$$\mathbf{W} \in \mathbb{R}^{h \times h}, \mathbf{U} \in \mathbb{R}^{h \times d}, \mathbf{b} \in \mathbb{R}^h$$

This model contains $h \times (h + d + 1)$ parameters, and optionally *h for* $\mathbf{h}_0$ (a common way is just to set $\mathbf{h}_0$ as $\mathbf{0}$)

Recurrent neural language models (RNNLMs)



$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \in \mathbb{R}^h$$

$$\hat{\mathbf{y}}_t = softmax(\mathbf{W}_o\mathbf{h}_t)$$

Training loss:

$$L(\theta) = -\frac{1}{n}\sum_{t=1}^{n} \log \hat{\mathbf{y}}_{t-1}(w_t)$$

Trainable parameters:

$$\theta = \{\mathbf{W}, \mathbf{U}, \mathbf{b}, \mathbf{W}_o, \mathbf{E}\}$$

Recurrent neural language models (RNNLMs)

- Forward pass + backward pass (compute gradients)

- Forward pass:

$$L = 0 \quad \mathbf{h}_0 = \mathbf{0}$$

For t =1, 2, …, n

$$y = -\log \mathrm{softmax}(\mathbf{W}_o \mathbf{h}_{t-1})(w_t)$$

$$\mathbf{x}_t = e(w_t)$$

$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

$$L = L + \frac{1}{n} y$$

accumulate loss

Backprop through time (BPTT)

$\mathbf{h}_1 = g(\mathbf{W}\mathbf{h}_0 + \mathbf{U}\mathbf{x}_1 + \mathbf{b})$
$\mathbf{h}_2 = g(\mathbf{W}\mathbf{h}_1 + \mathbf{U}\mathbf{x}_2 + \mathbf{b})$
$\mathbf{h}_3 = g(\mathbf{W}\mathbf{h}_2 + \mathbf{U}\mathbf{x}_3 + \mathbf{b}) \qquad \hat{\mathbf{y}}_3 = \text{softmax}(\mathbf{W}_o\mathbf{h}_3)$

$$L_3 = -\log \hat{\mathbf{y}}_3(w_4)$$

First, compute gradient with respect to hidden vector of last time step: $\frac{\partial L_3}{\partial \mathbf{h}_3}$

$$\frac{\partial L_3}{\partial \mathbf{W}} = \frac{\partial L_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}} + \frac{\partial L_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}} + \frac{\partial L_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}}$$

More generally,
$$\boxed{\frac{\partial L}{\partial \mathbf{W}} = -\frac{1}{n}\sum_{t=1}^{n}\sum_{k=1}^{t}\frac{\partial L_t}{\partial \mathbf{h}_t}\left(\prod_{j=k+1}^{t}\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right)\frac{\partial \mathbf{h}_k}{\partial \mathbf{W}}}$$

If *k* and *t* are far away, the gradients can grow/shrink exponentially
(called the gradient exploding or gradient vanishing problem)

Backprop through time (BPTT)

One solution for **gradient exploding** is called **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

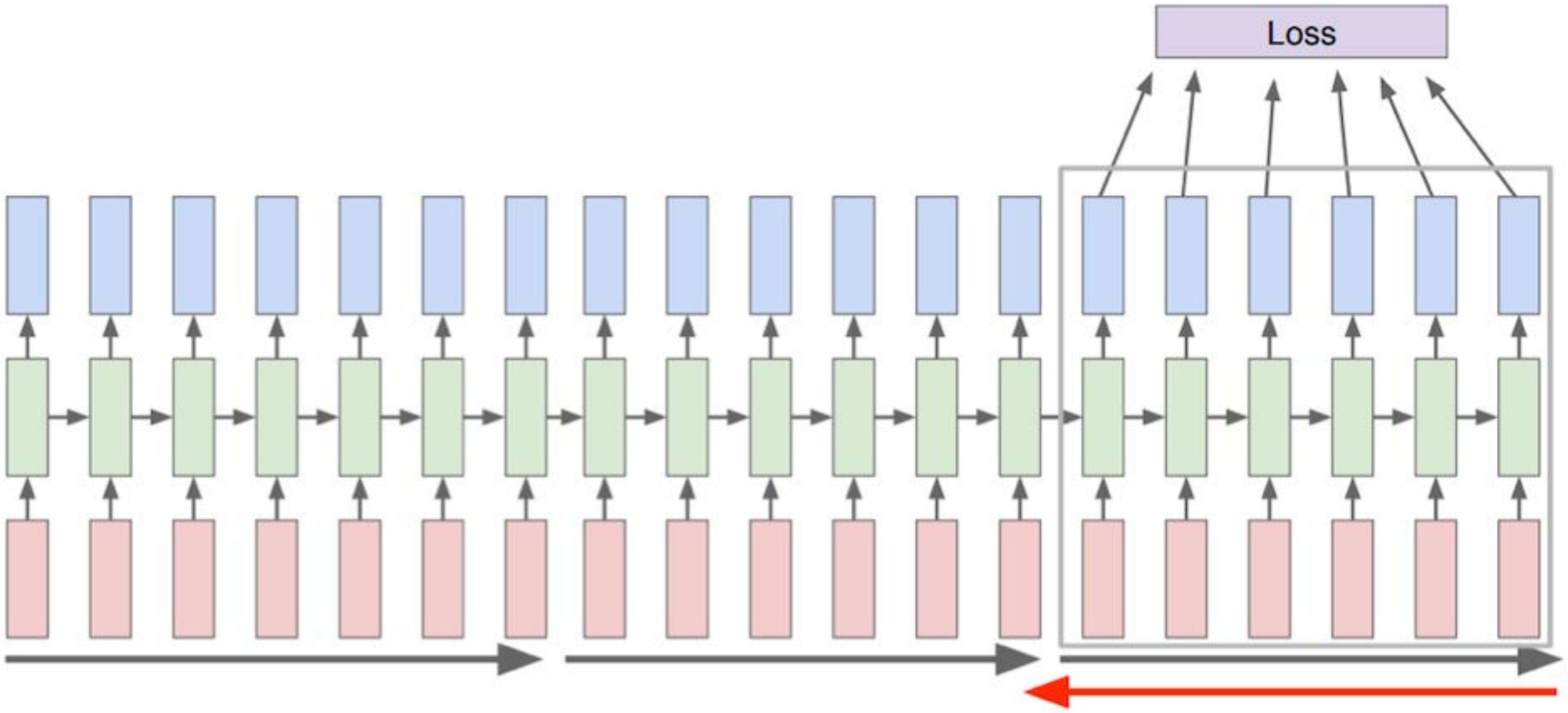
Intuition: take a step in the same direction but a smaller step!

Gradient vanishing is a harder problem to solve:

As the proctor started the clock, the students opened their ______

Backprop through time (BPTT)

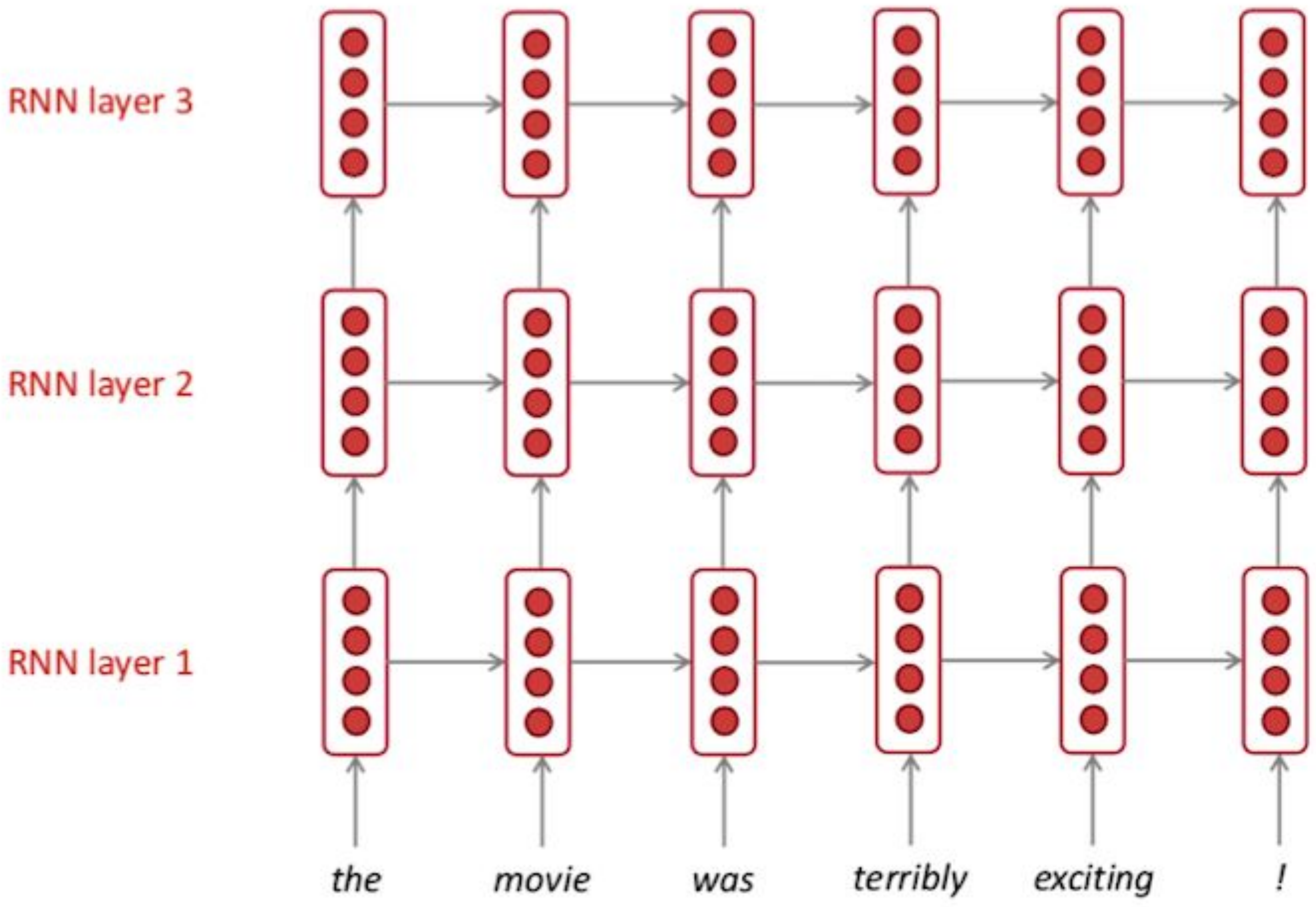
- Backpropagation is very expensive if you handle long sequences



- Run forward and backward through chunks of the sequence instead of whole sequence
- Carry hidden states forward in time forever, but only back-propagate for some smaller number of st

Recurrent neural networks (RNNs)

Multi-layer

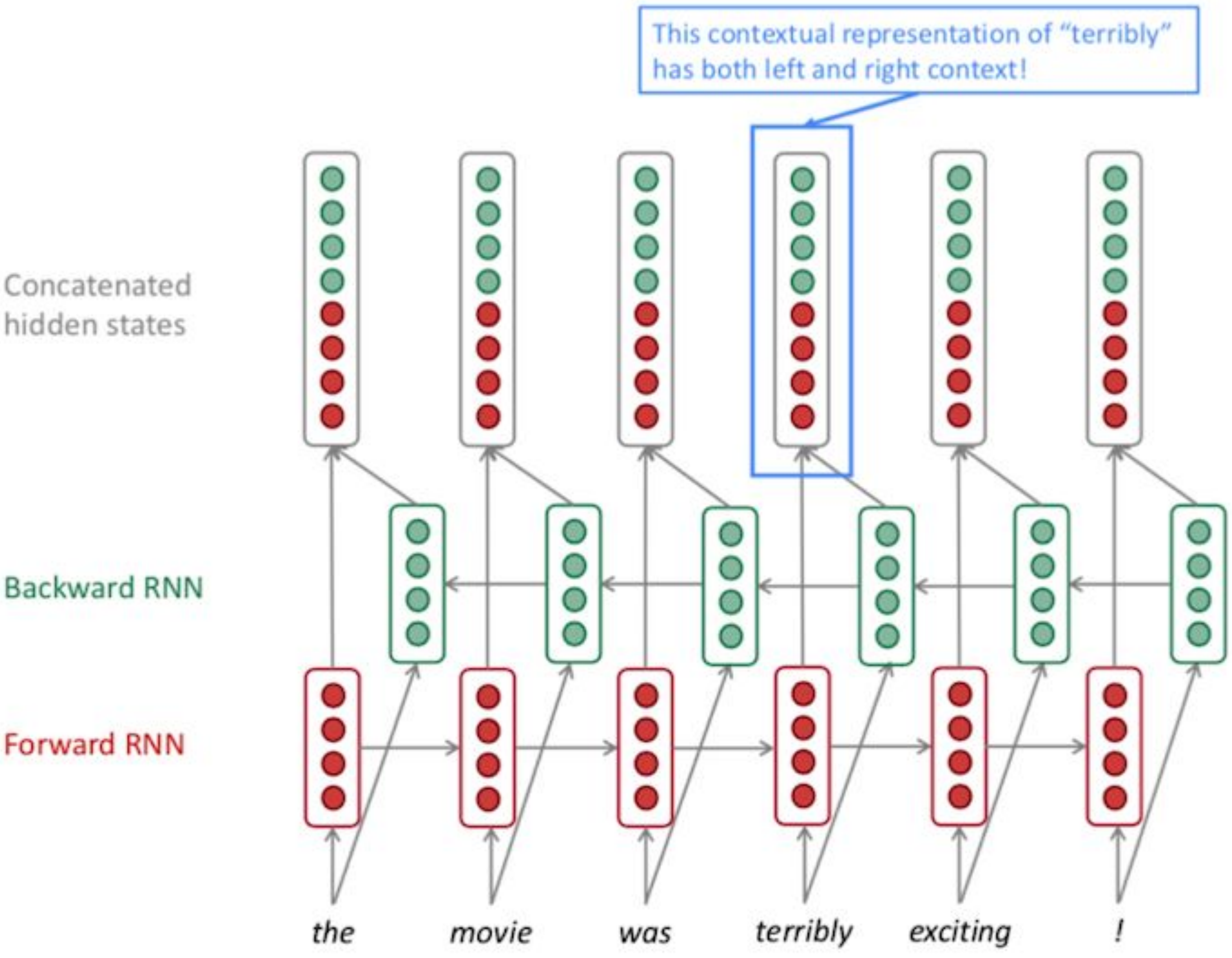


The hidden states from RNN layer i are the inputs to RNN layer $i + 1$

- In practice, using 2 to 4 layers is common (usually better than 1 layer)
- Transformer networks can be up to 24 layers with lots of skip-connections

Recurrent neural networks (RNNs)

Bi-directional



$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h$$

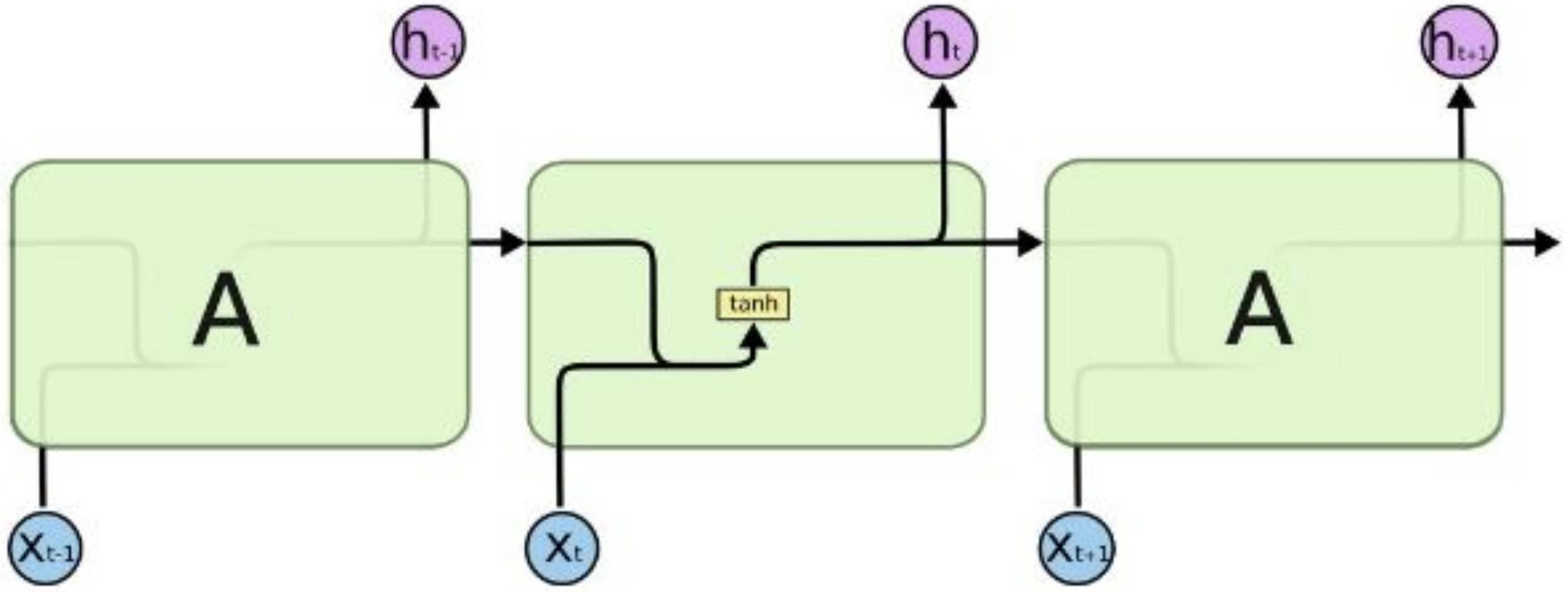
$$\overrightarrow{\mathbf{h}}_t = f_1(\overrightarrow{\mathbf{h}}_{t-1}, \mathbf{x}_t), t = 1,2,\ldots n$$

$$\overleftarrow{\mathbf{h}}_t = f_2(\overleftarrow{\mathbf{h}}_{t+1}, \mathbf{x}_t), t = n, n-1,\ldots 1$$

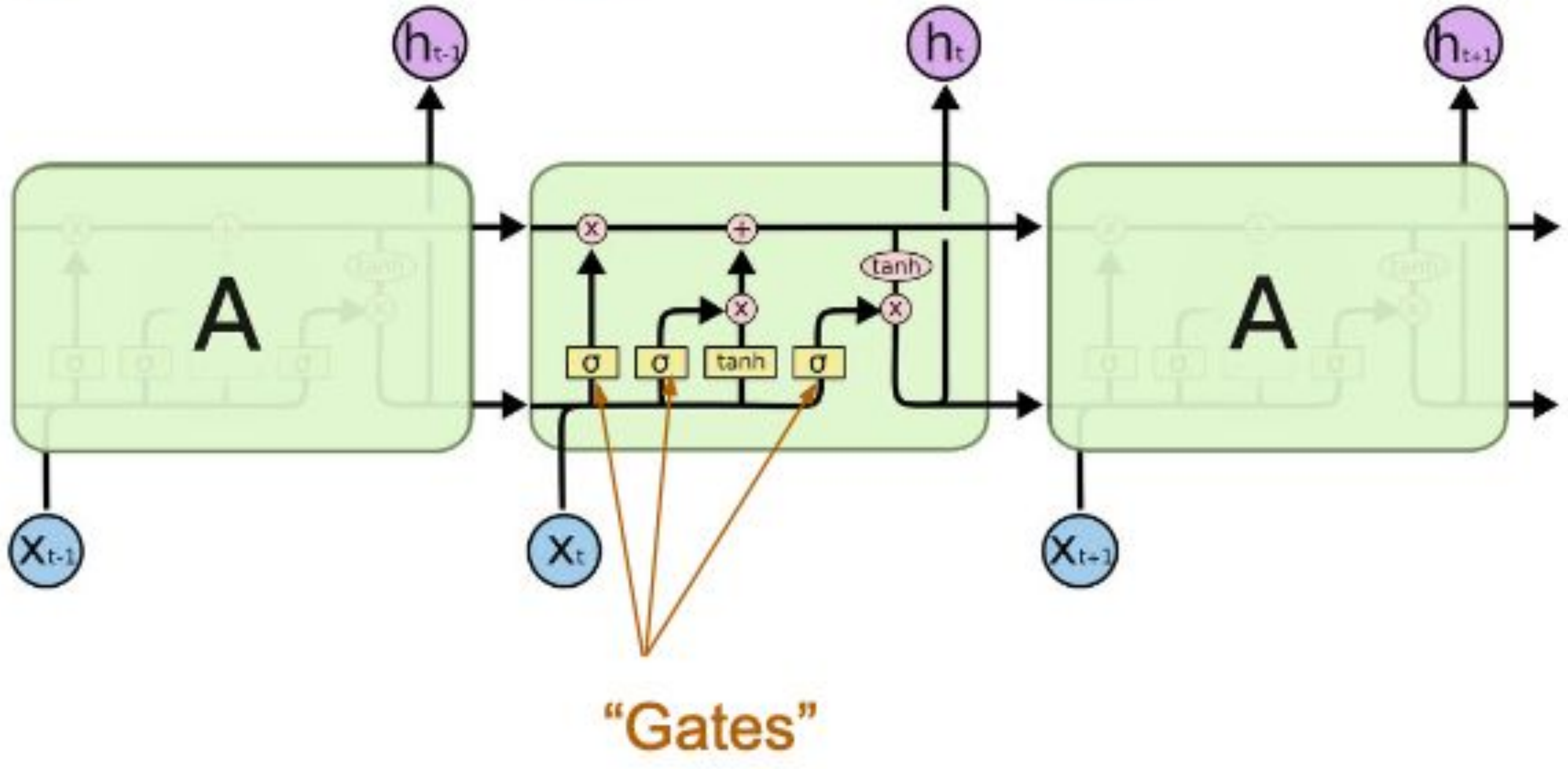
$$\mathbf{h}_t = [\overleftarrow{\mathbf{h}}_t, \overrightarrow{\mathbf{h}}_t] \in \mathbb{R}^{2h}$$

Long short term memory (LSTMs)

Simple RNN



LSTM



Two recurrent values!
(hidden and cell states)

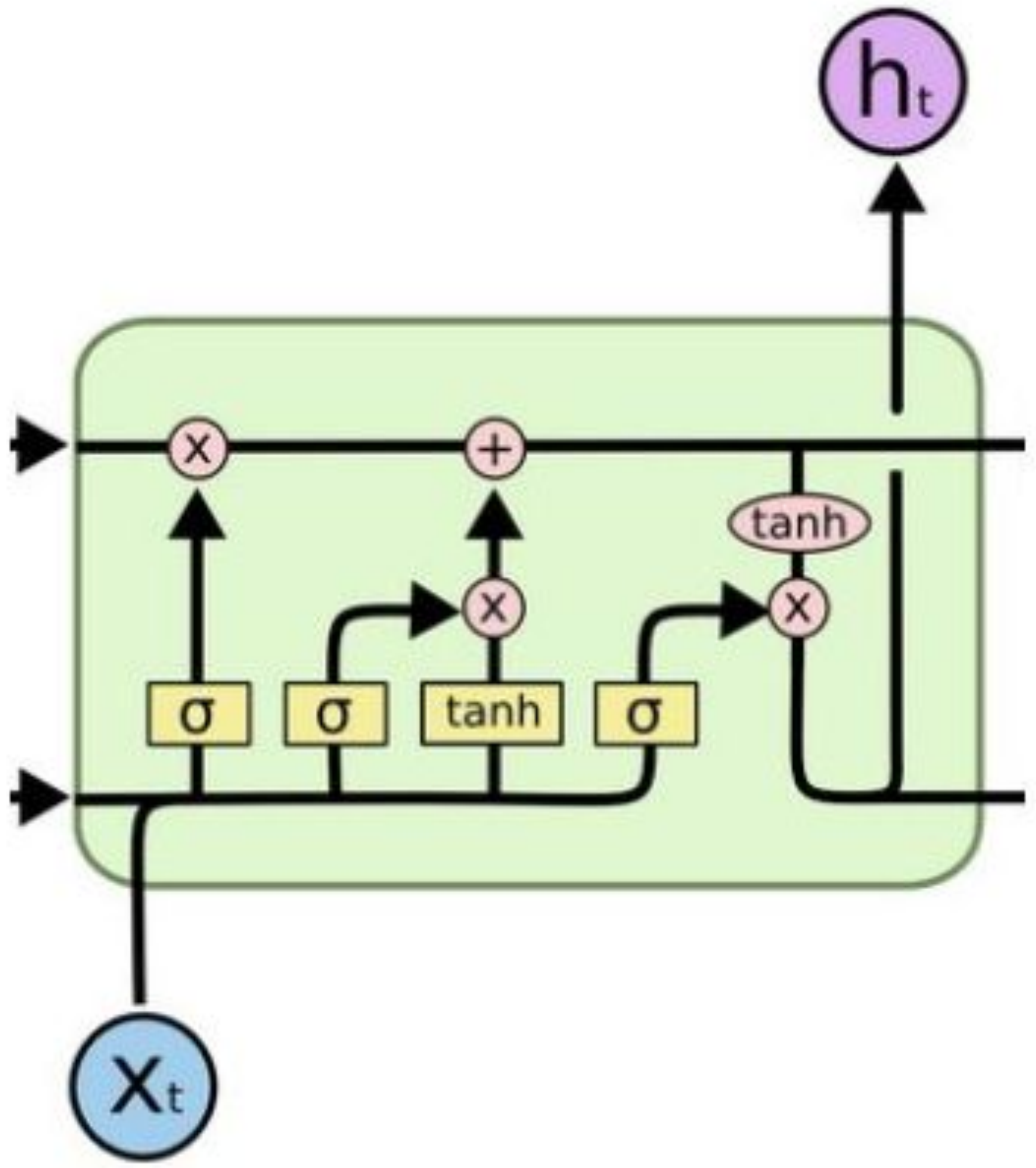
Slides from Precept 6, sp23

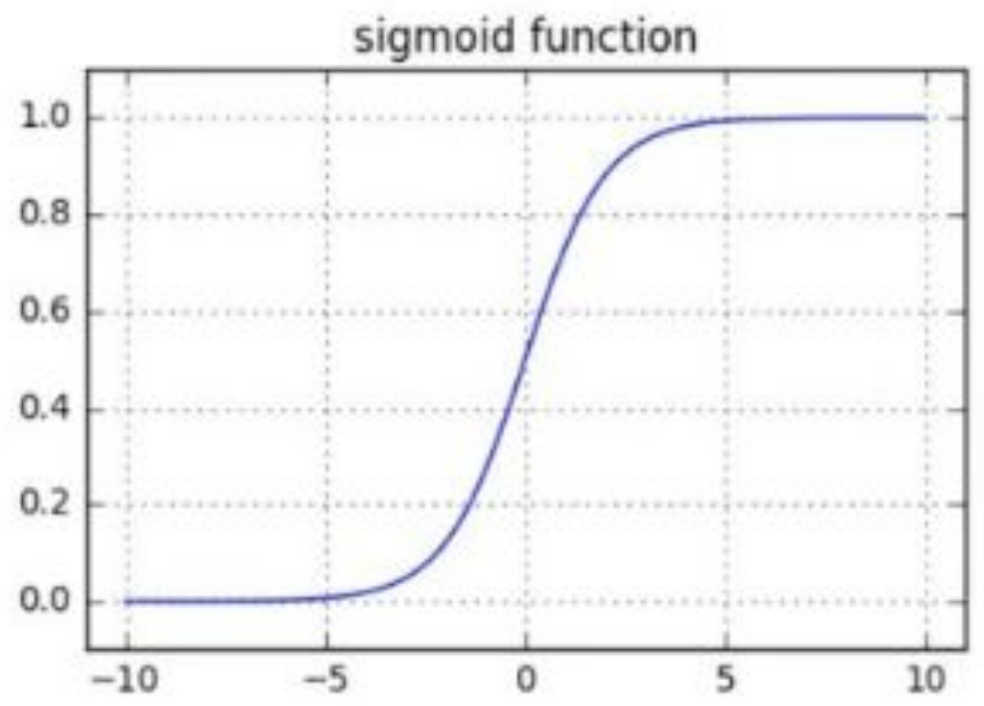
Long short term memory (LSTMs)

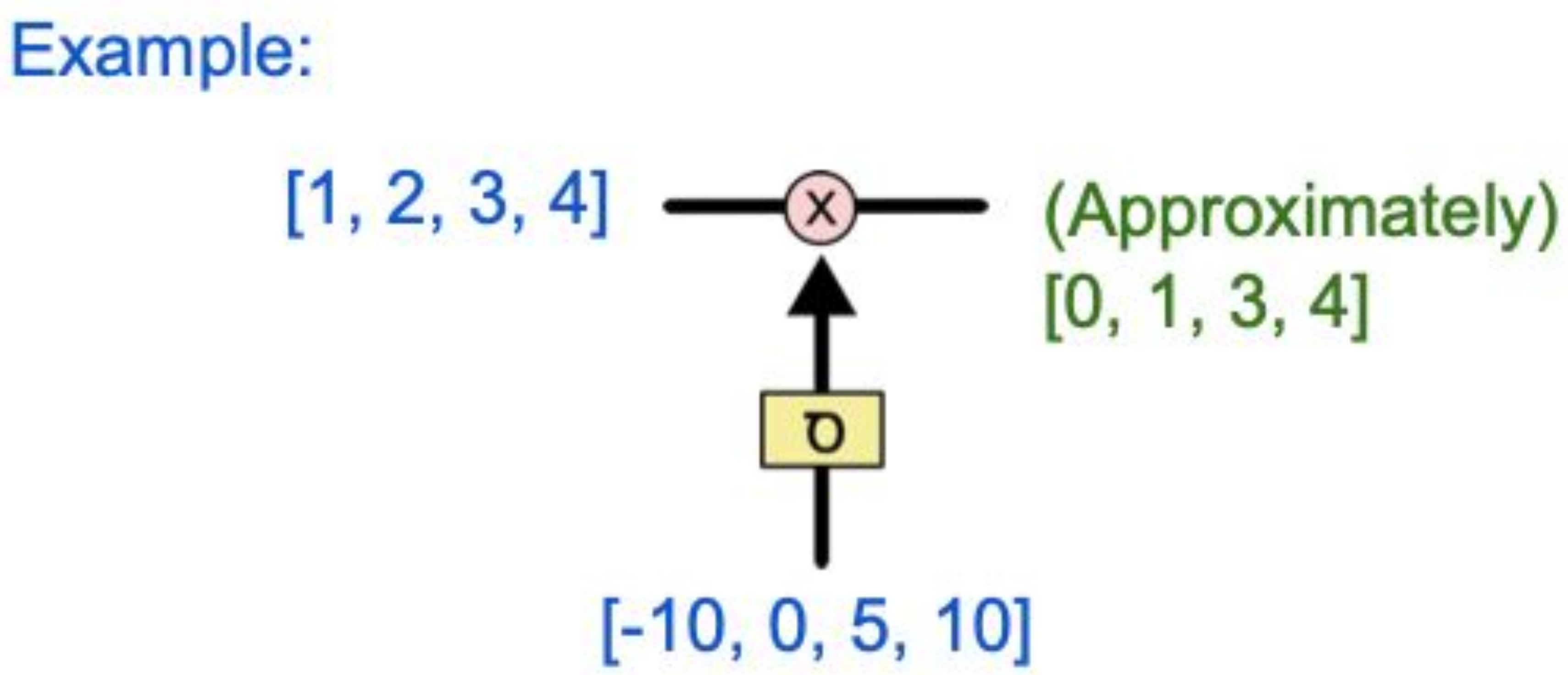
LSTMs Broken Down

Gates (i.e. sigmoid followed by multiplication)

- Outputs value in range (0, 1)
- Intuitive meaning:
  - Close to 0 => “forget this value”
  - Close to 1 => “keep this value”



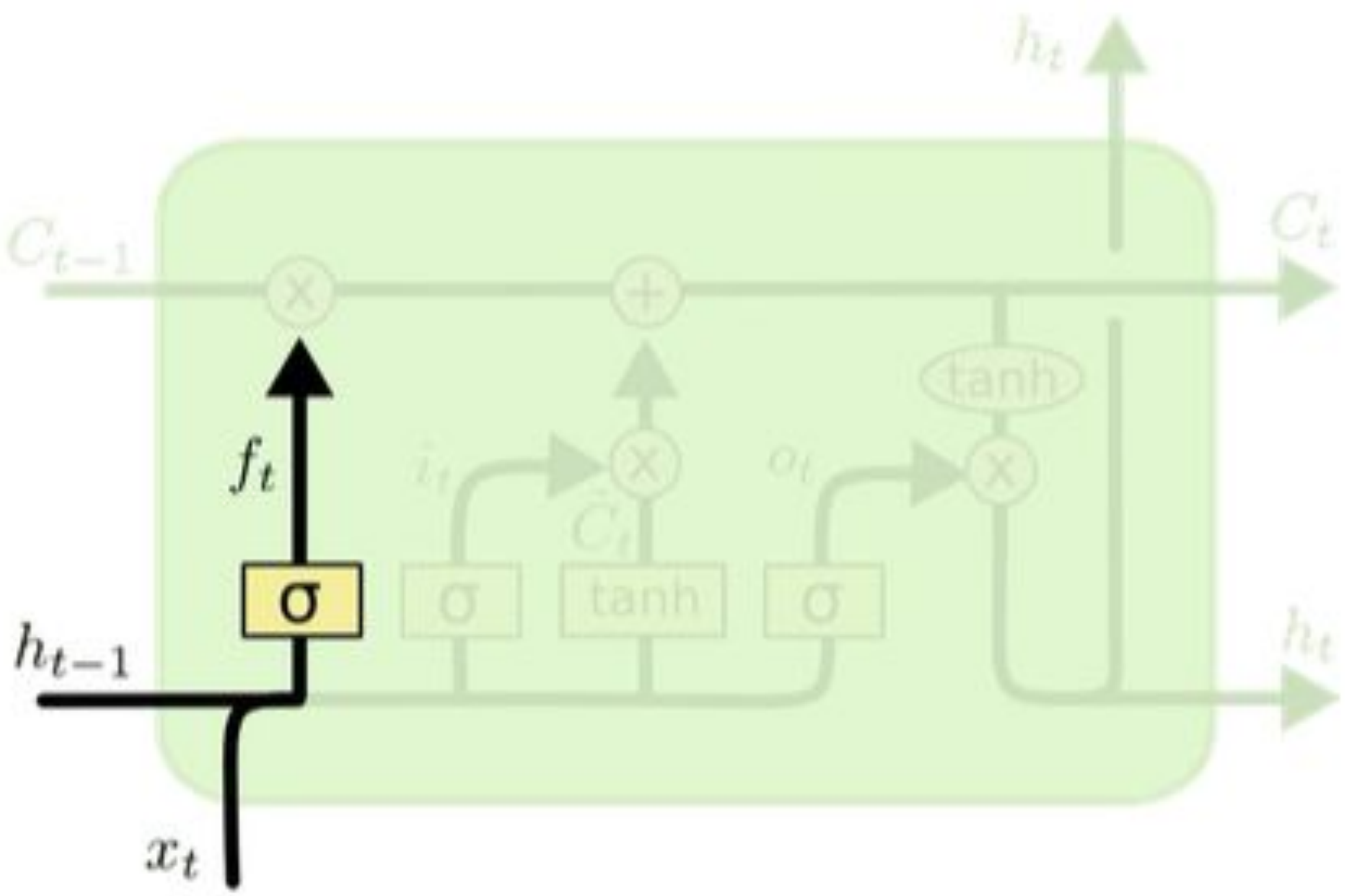




Long short term memory (LSTMs)

Suppose we are predicting the sentence "Jon is a boy. Sally is a girl."

Step 1 (Forget gate): Discard information.
"Given the current input and the previous hidden state, how much should I ***discard from*** *the cell state?*



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

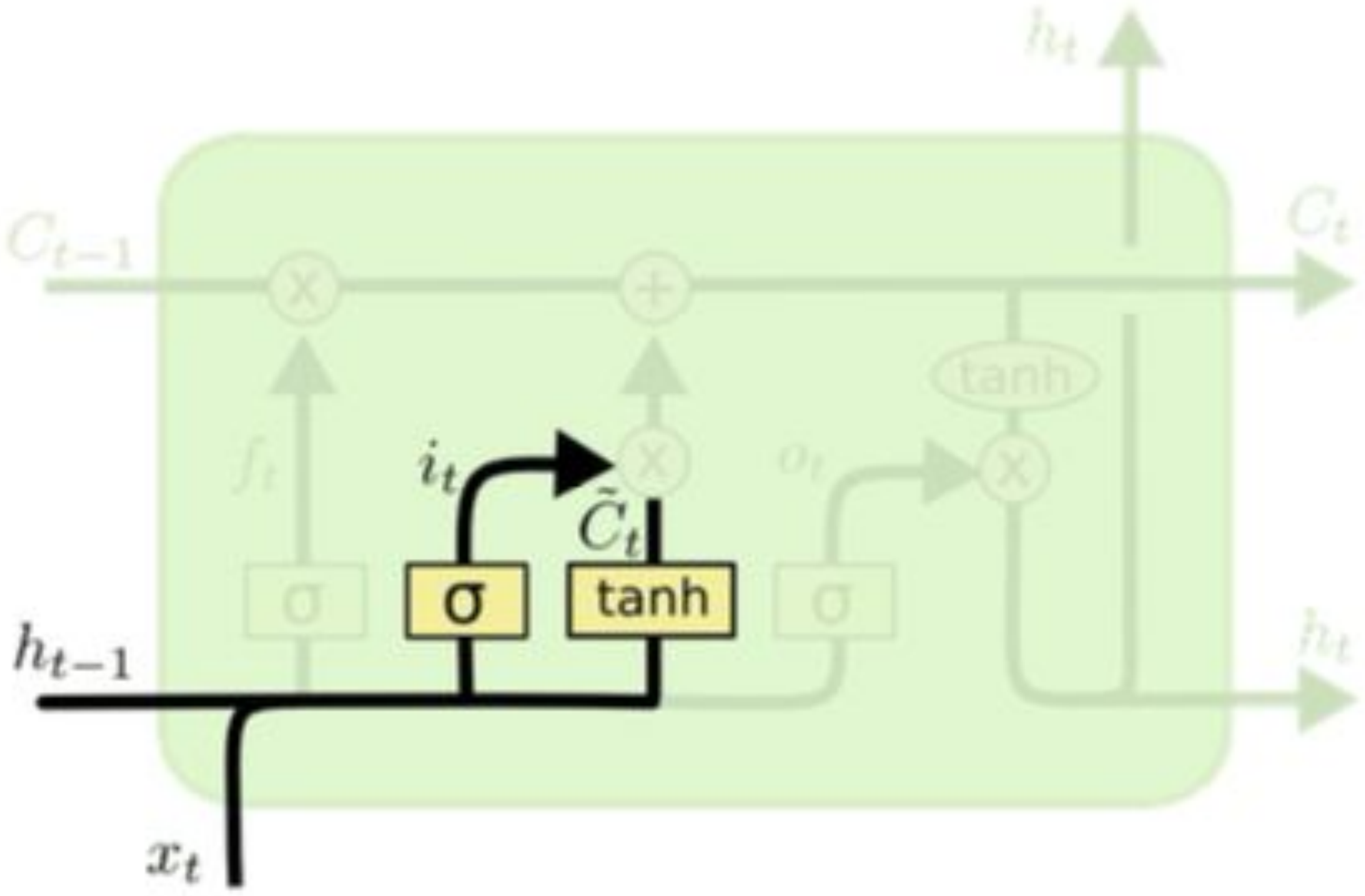
e.g. When I see the word "Sally", I may want to discard existing information associated with the gender of the subject in the cell state (which may be carried over from the first half of the sentence)

Long short term memory (LSTMs)

Suppose we are predicting the sentence "Jon is a boy. Sally is a girl."

Step 1 (Forget gate): Discard information.
"Given the current input and the previous hidden state, how much should I ***discard from*** *the cell state?"*

Step 2 (Input gate): Add new information.
"Given the current input and the previous hidden state, what should I ***add to*** *the cell state?"*



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

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

e.g. When I see the word "Sally", I may want to add information to the cell state indicating that the subject is female

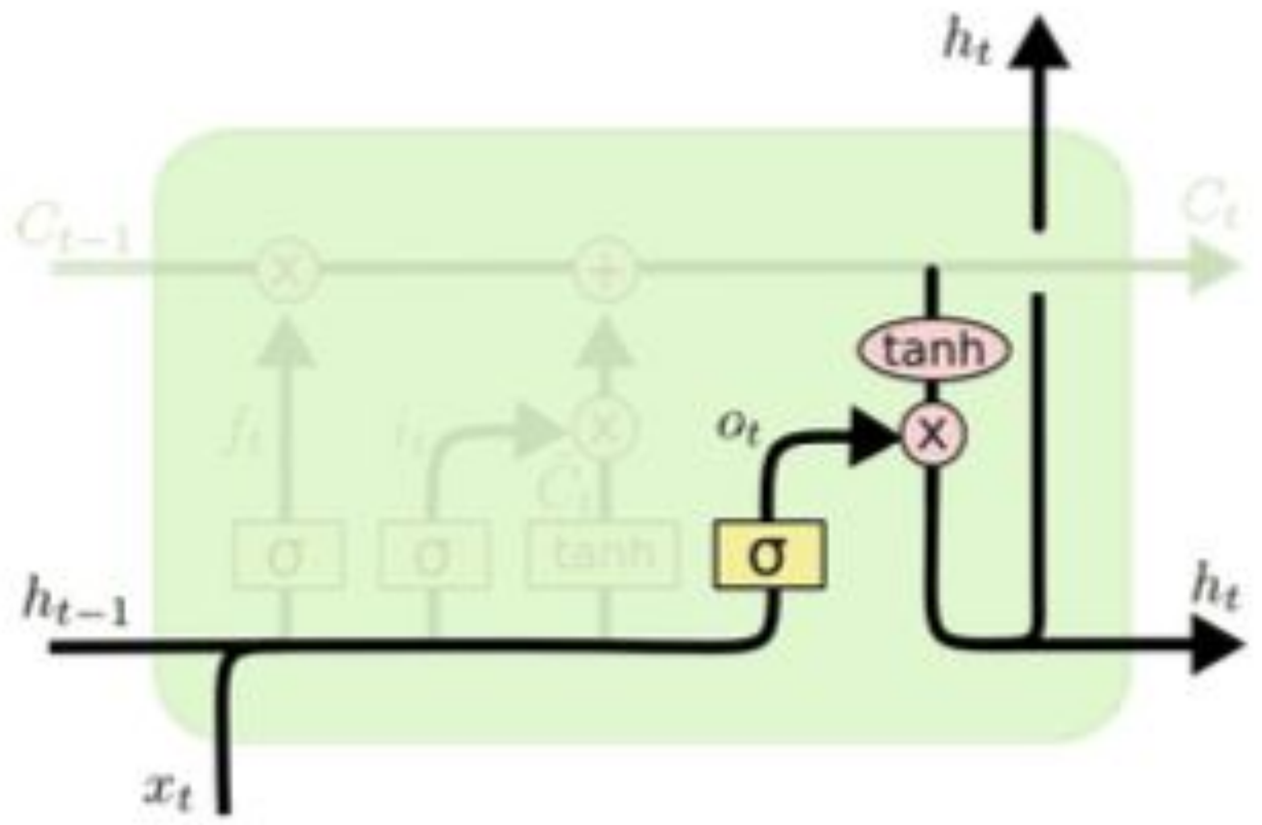
Long short term memory (LSTMs)

Suppose we are predicting the sentence "Jon is a boy. Sally is a girl."

Step 1 (Forget gate): Discard information.
*"Given the current input and the previous hidden state, how much should I **discard from** the cell state?"*

Step 2 (Input gate): Add new information.
*"Given the current input and the previous hidden state, what should I **add to** the cell state?"*

Step 3 (Output gate): Compute the output.
"Given the current input, previous hidden state, and updated cell state, what should I output?"

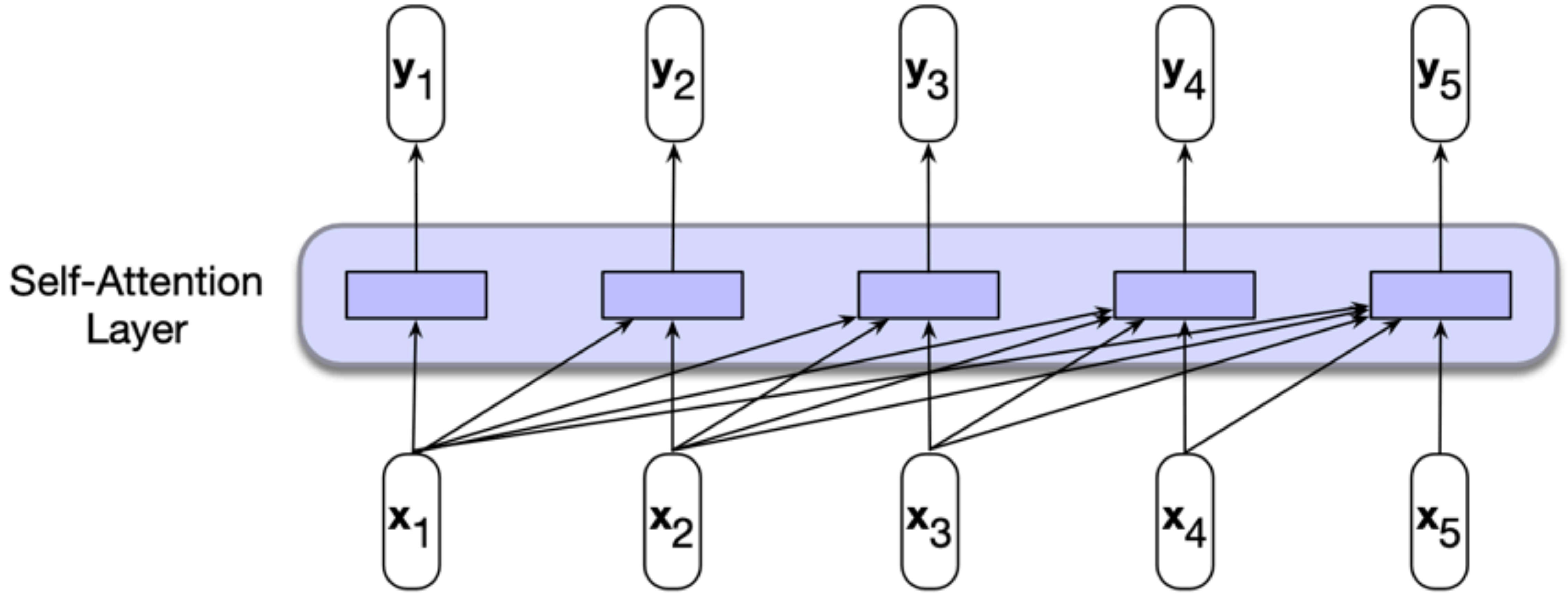


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

e.g. Predicting the word "girl" given that your cell state should contain gender information from when it saw Sally

Bi-Directional LMs

Autoregressive / left-to-right

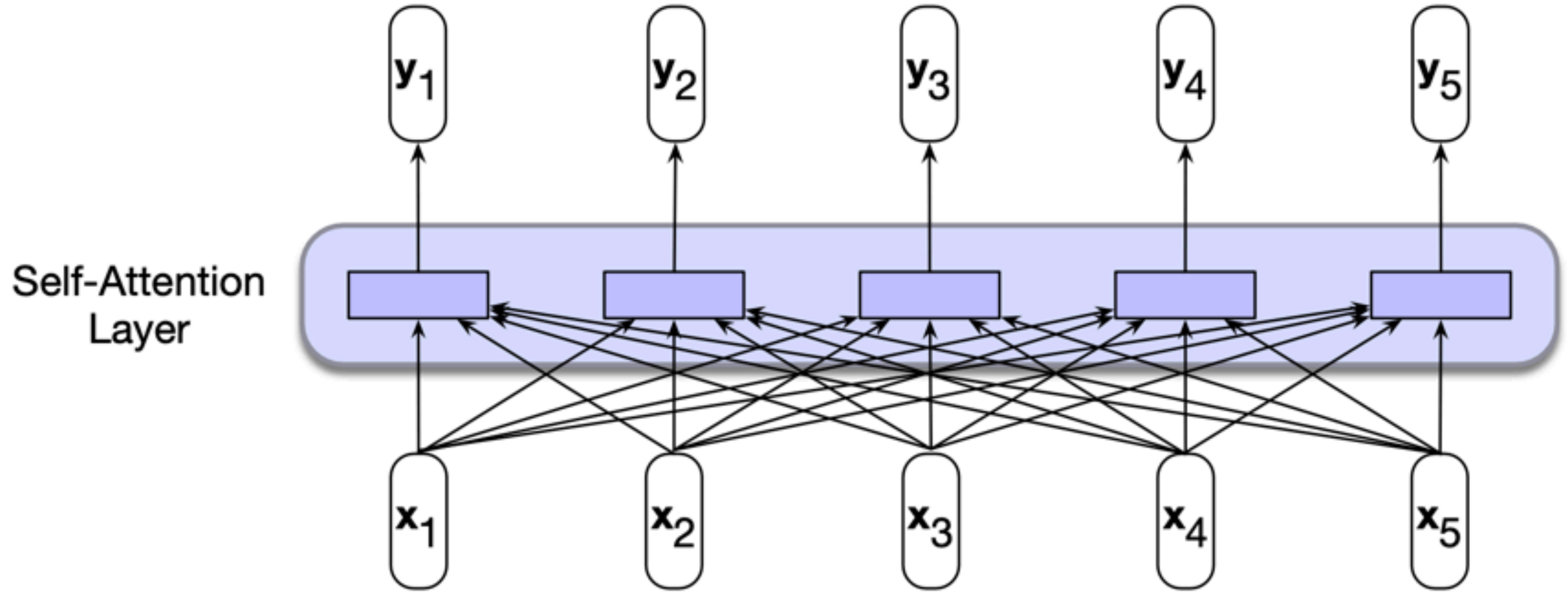


computation for input $\mathbf{x}_1, \ldots, \mathbf{x}_3$ blind to $\mathbf{x}_4$ and $\mathbf{x}_5$

$\mathbf{y}_5$ is embedding for input $\mathbf{x}_1, \ldots, \mathbf{x}_5$

$\mathbf{y}_5$ is a "left-contextual embedding"

Bidirectional

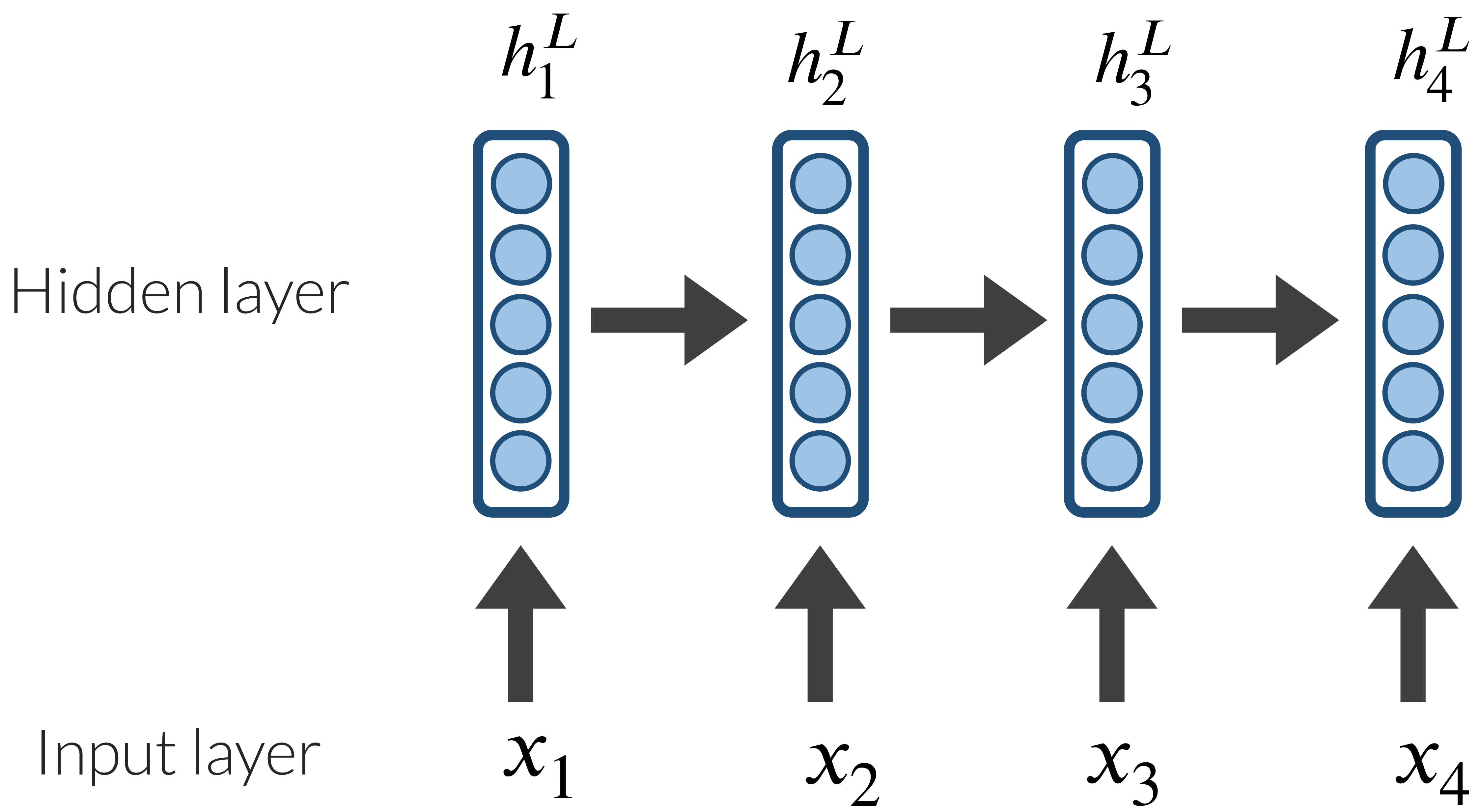


computation for input $\mathbf{x}_1, \ldots, \mathbf{x}_3$ sees $\mathbf{x}_4$ and $\mathbf{x}_5$

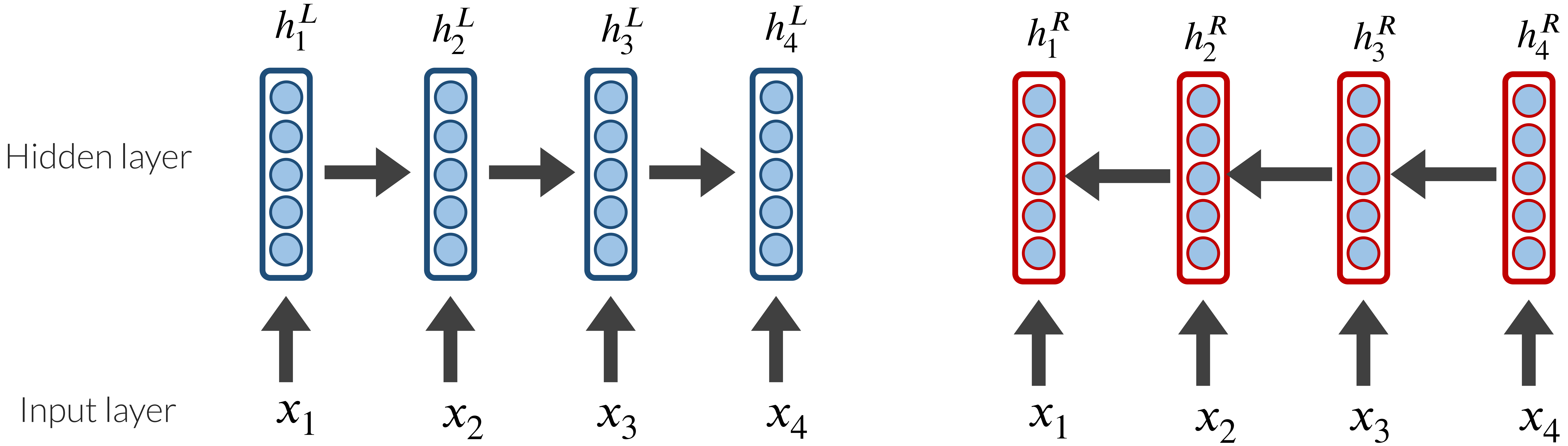
$\mathbf{y}_1, \ldots, \mathbf{y}_5$ is embedding for input $\mathbf{x}_1, \ldots, \mathbf{x}_5$

$\mathbf{y}_i$ are bidirectional "contextual embeddings"

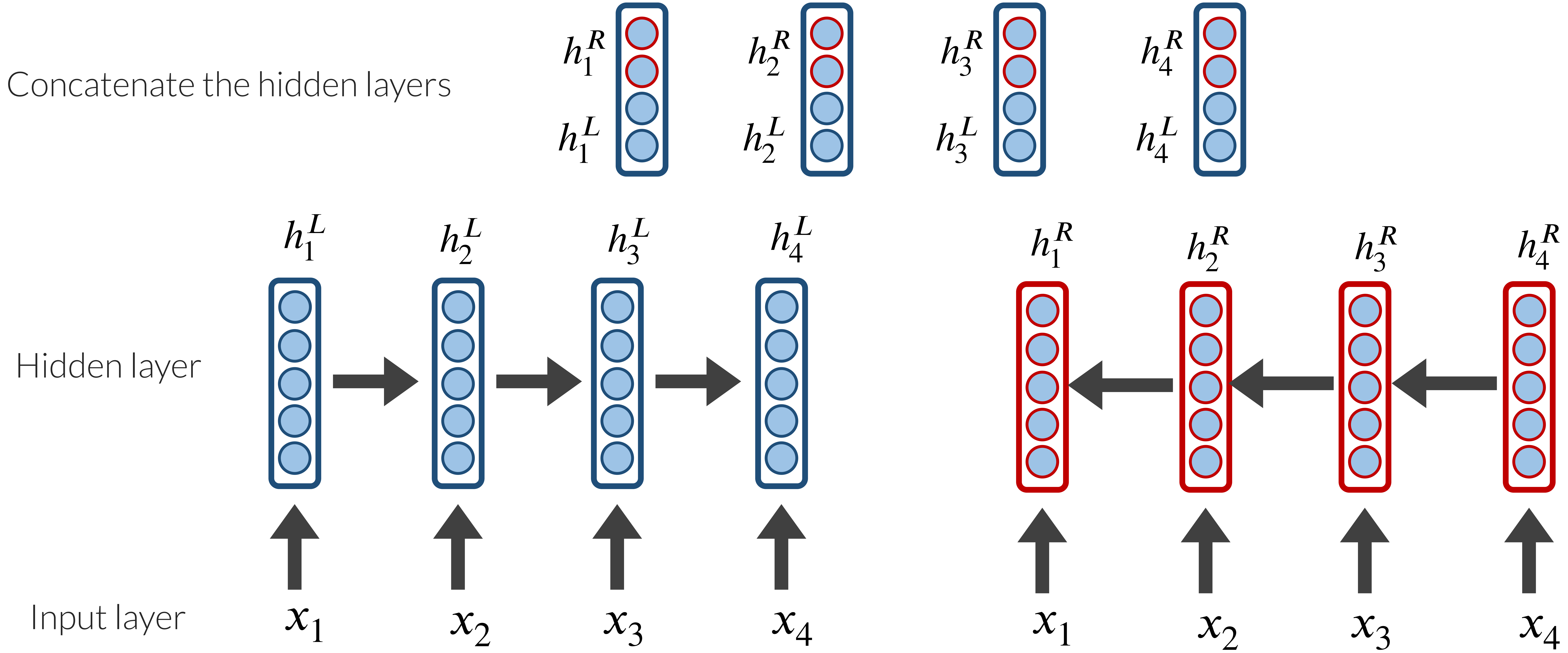
Reading Bidirectionally



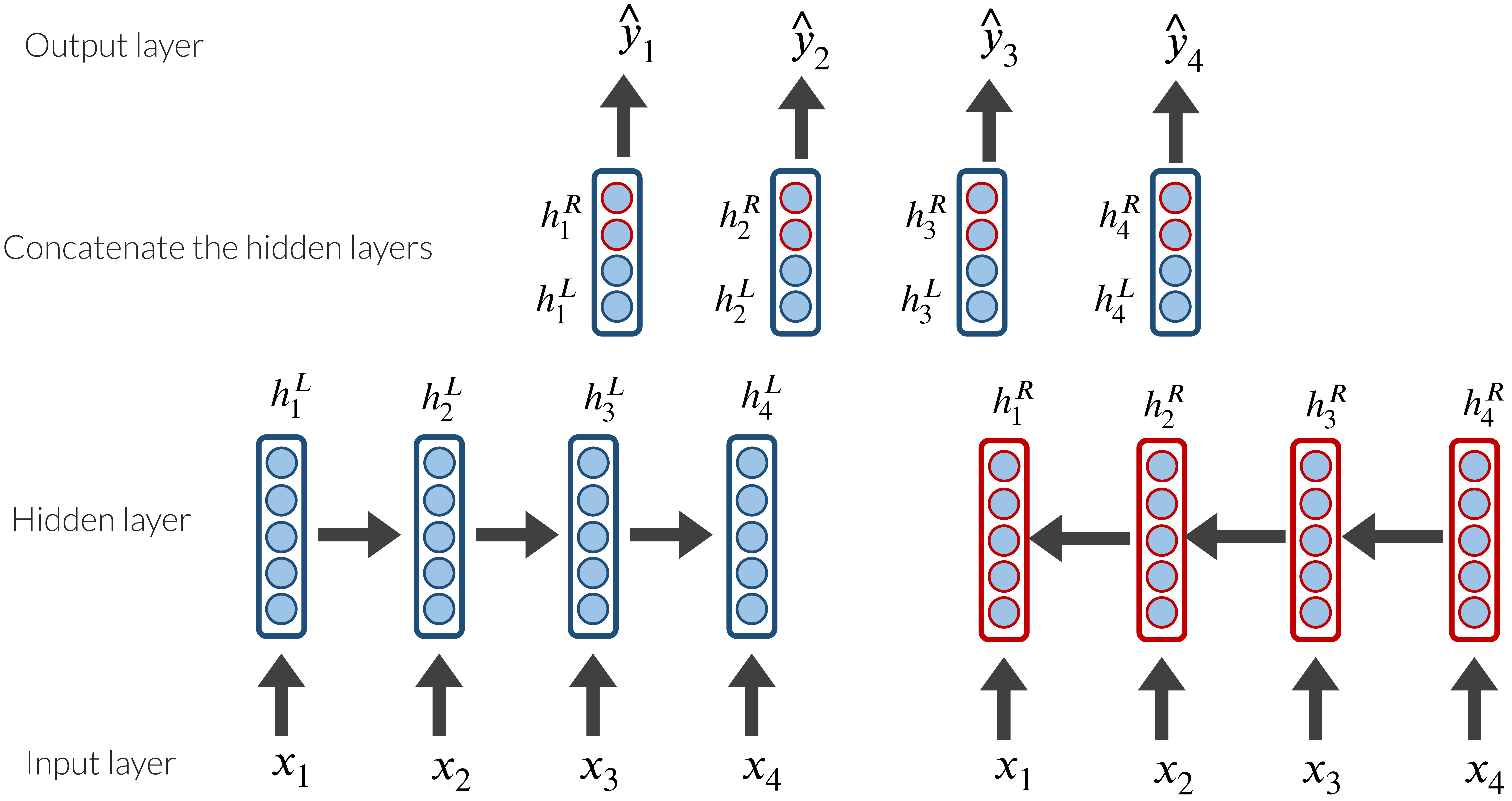
Reading Bidirectionally



Reading Bidirectionally



Reading Bidirectionally



Bi-directional LMs

Strengths

- Usually performs at least as well as uni-directional RNNs/LSTMs
- More encompassing (left & right) contextualized embeddings

Weaknesses

- Slower to train
- Only possible if access to full data is allowed
- Less suitable for (autoregressive) language generation

Bi-LSTM: ELMO

ELMo

General Idea:

- Goal is to obtain highly rich embeddings for each word (unique type)
- Use both directions of context (bi-directional), with increasing abstractions (stacked)
- Linearly combine all abstract representations (hidden layers) and optimize w.r.t. a particular task (e.g., sentiment classification)

ELMo Slides: https://www.slideshare.net/shuntaroy/a-review-of-deep-contextualized-word-representations-peters-2018

ELMo

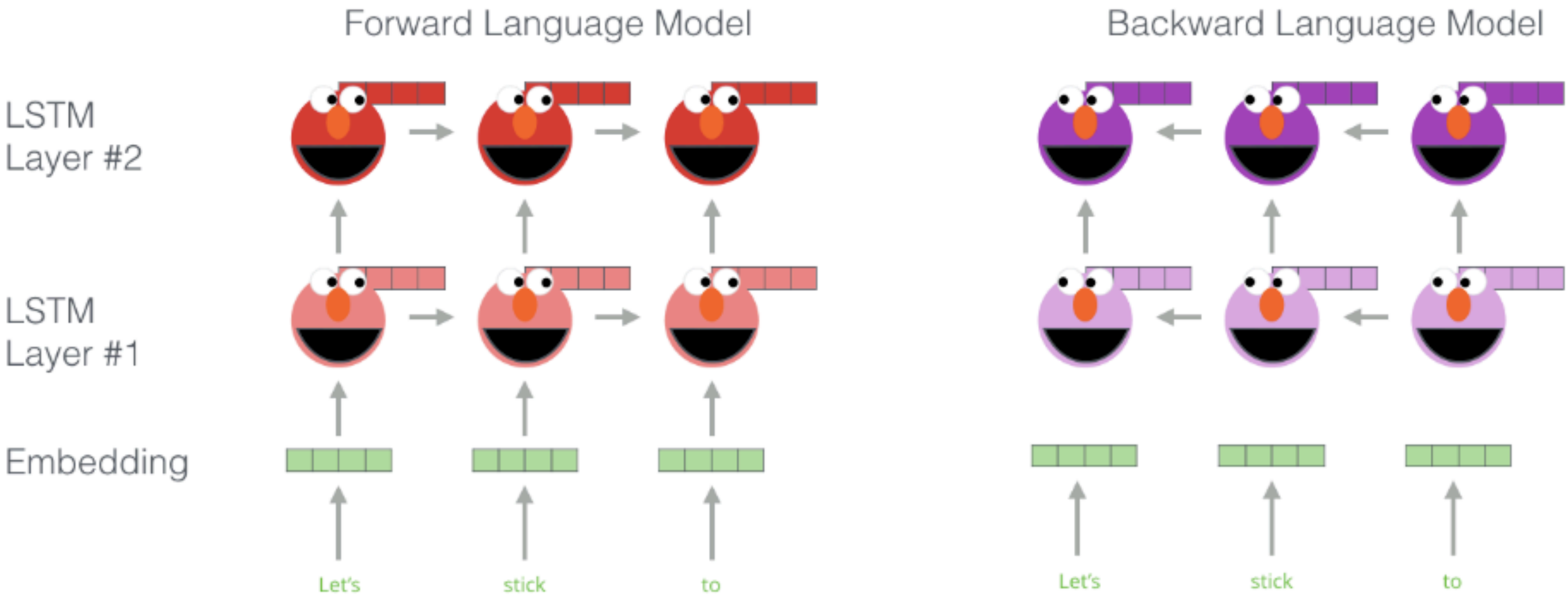


Illustration: http://jalammar.github.io/illustrated-bert/

Embedding of “stick” in “Let’s stick to” - Step #2

1- Concatenate hidden layers

2- Multiply each vector by a weight based on the task

3- Sum the (now weighted) vectors

ELMo embedding of “stick” for this task in this context

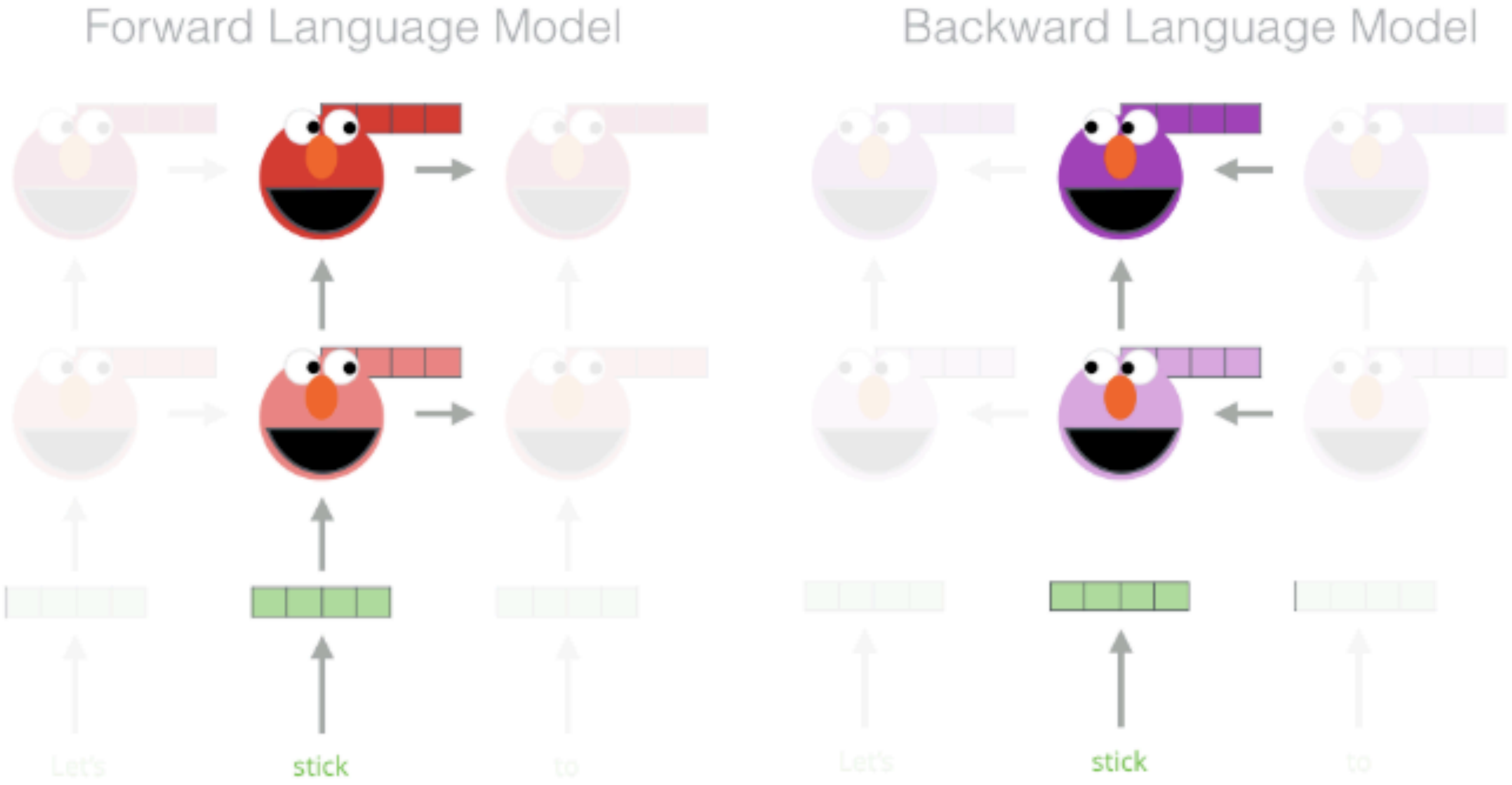
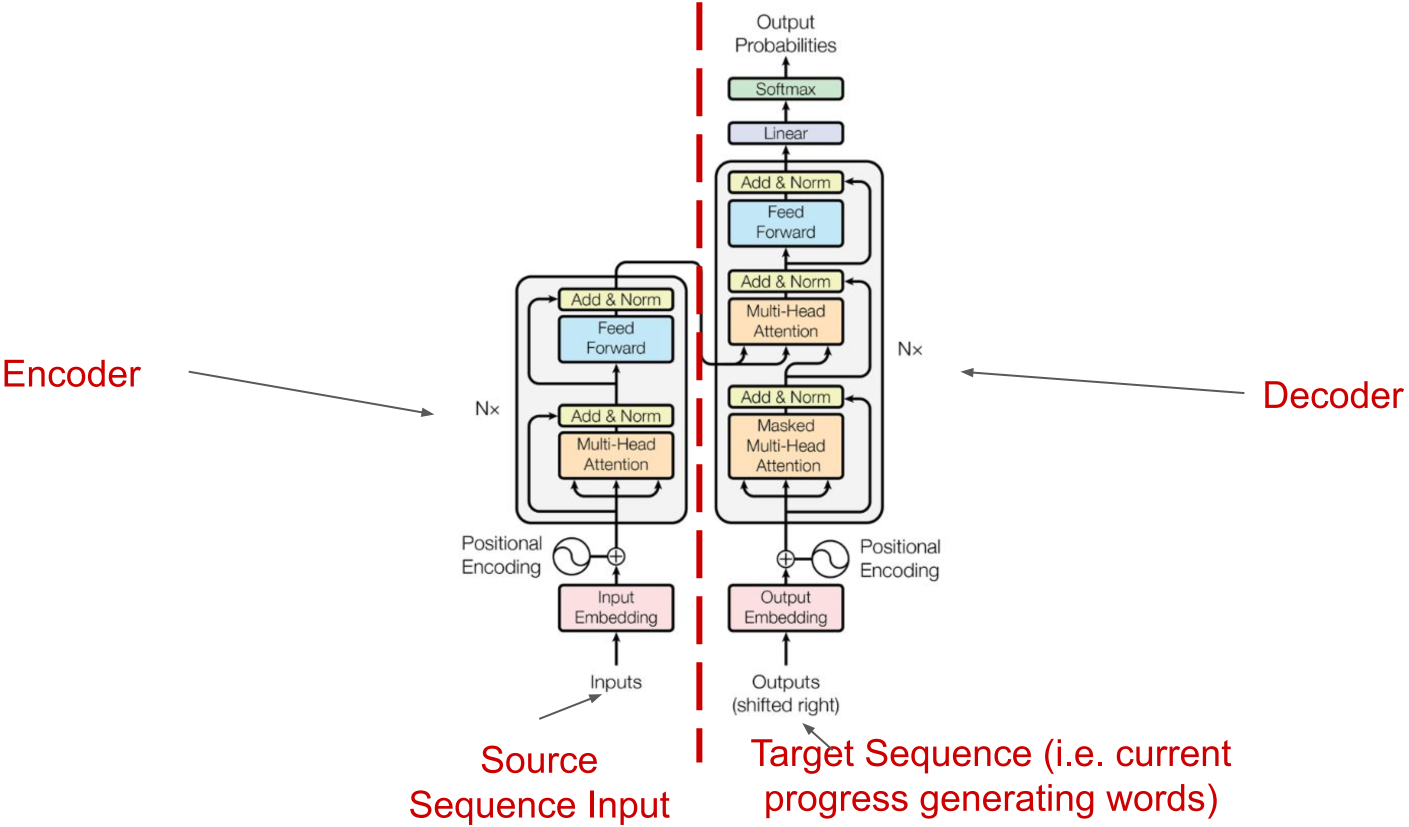
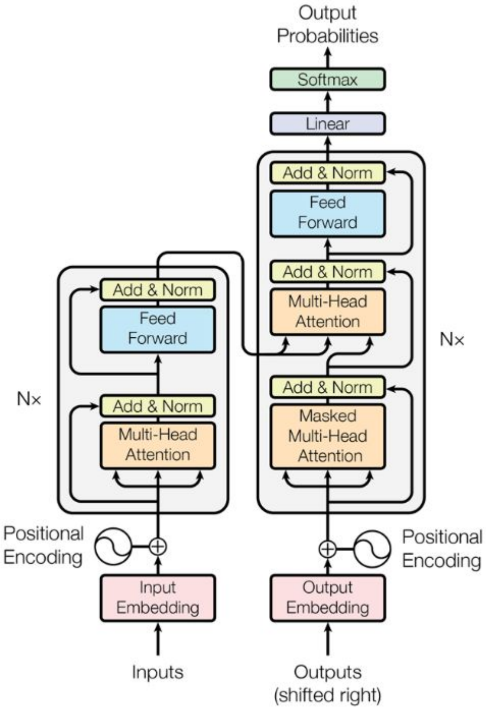


Illustration: http://jalammar.github.io/illustrated-bert/

Transformer Architecture

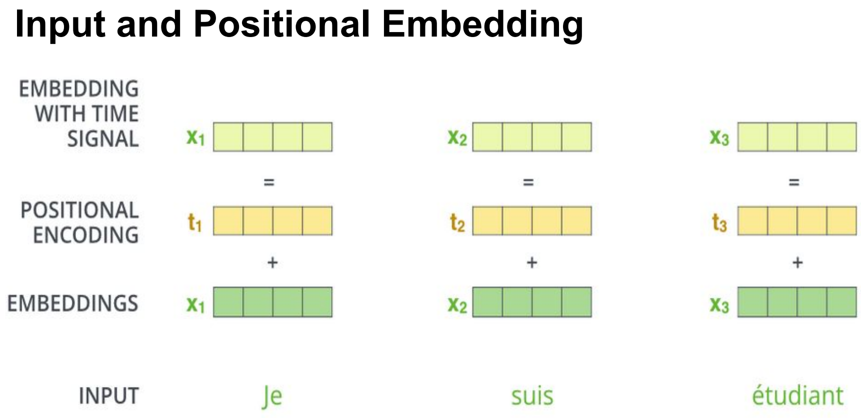


Transformer Encoder

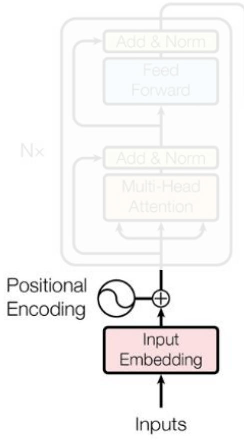


Source sequence $(x_1, \ldots, x_n)$

Transformer Encoder: Positional + Word Embedding



Embedded source sequence
$\mathbb{R}^{n \times d_1}$



Source sequence $(x_1, \ldots, x_n)$

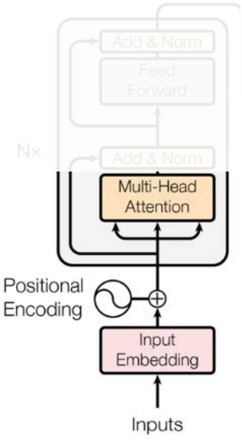
Transformer Encoder: Multi-Head Self Attention

After Multi-Head Attention

$\mathbb{R}^{n \times d_2}$

Embedded source sequence

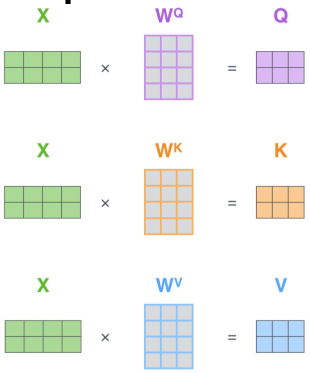
$\mathbb{R}^{n \times d_1}$



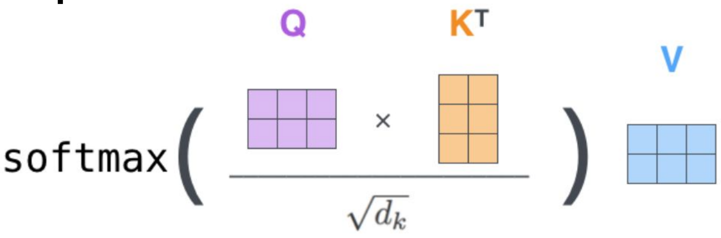
Source sequence $(x_1, \ldots, x_n)$

Self-Attention: $W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$

Step 1:



Step 2:



MultiHead Attention: $W^O \in \mathbb{R}^{d \times d_2}$

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_\mathrm{h})W^O$$

$$\mathrm{head}_i = \mathrm{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

Transformer Encoder: Multi-Head Self Attention

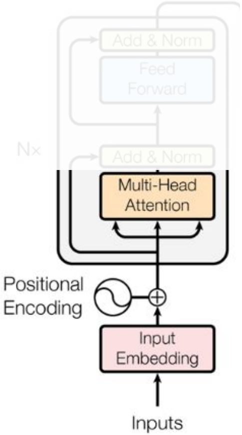
In practice, $d_1 = d_2$

After Multi-Head Attention

$\mathbb{R}^{n \times d_1}$

Embedded source sequence

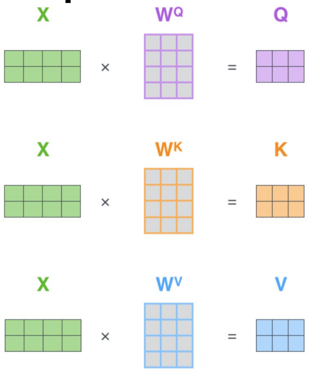
$\mathbb{R}^{n \times d_1}$



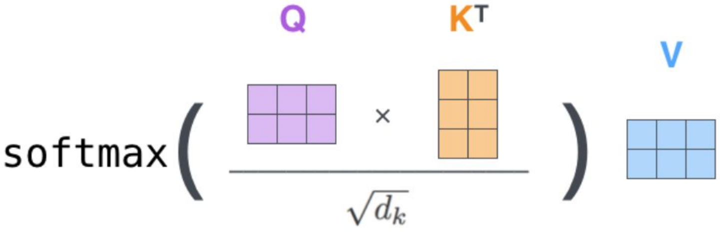
Source sequence $(x_1, \dots, x_n)$

Self-Attention: $W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$

Step 1:



Step 2:



"Self" attention means Q, K, V are all computed from a single sequence

MultiHead Attention: $W^O \in \mathbb{R}^{d \times d_2}$

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, \dots, \mathrm{head_h})W^O$$

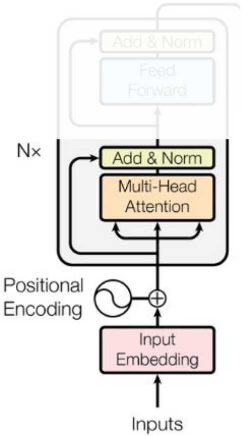
$$\mathrm{head_i} = \mathrm{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

Transformer Encoder: Add & Norm

After Add & Norm
$\mathbb{R}^{n \times d_1}$

After Multi-Head Attention
$\mathbb{R}^{n \times d_1}$

Embedded source sequence
$\mathbb{R}^{n \times d_1}$



Source sequence $(x_1, \ldots, x_n)$

Add & Norm:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

LayerNorm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Layer normalization

- The *layer-normalization function* $\mathrm{LayerNorm}\colon \mathbb{R}^{d_{\mathcal{M}}} \to \mathbb{R}^{d_{\mathcal{M}}}$ is included for training efficiency and defined as:

$$\mathrm{LayerNorm}(\mathbf{x})^{(\ell,k)} = \gamma^{(\ell,k)} \odot \frac{\mathbf{x} - \mu(\mathbf{x})}{\sigma(\mathbf{x})} + \beta^{(\ell,k)},$$

where:
  - $\mu(\mathbf{x}) = \frac{1}{d}\sum_{i=1}^{d_{\mathcal{M}}} x_i$ is the mean of the values in the embedding,
  - $\sigma(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d_{\mathcal{M}}}(x_i - \mu(\mathbf{x}))^2 + \epsilon}$ is the standard deviation with a small constant $\epsilon > 0$ for numerical stability,
  - $\gamma^{(\ell,k)}, \beta^{(\ell,k)} \in \mathbb{R}^{d_{\mathcal{M}}}$ are learned parameters (scale and shift), one for each layer ℓ and occurrence k of the layer-normalization operation in the transformer block, and
  - $\odot$ denotes elementwise multiplication.

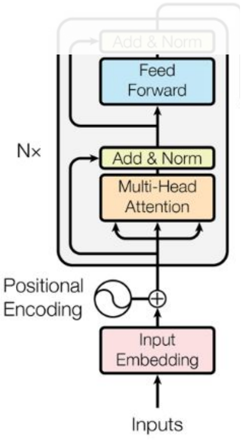
Transformer Encoder: Feed Forward

After Feed Forward
$\mathbb{R}^{n \times d_1}$

After Add & Norm
$\mathbb{R}^{n \times d_1}$

After Multi-Head Attention
$\mathbb{R}^{n \times d_1}$

Embedded source sequence
$\mathbb{R}^{n \times d_1}$



Source sequence $(x_1, \ldots, x_n)$

Feed Forward

$$\mathrm{FFN}(\mathbf{x}_i) = \mathrm{ReLU}(\mathbf{x}_i \mathbf{W}_1 + \mathbf{b}_1)\mathbf{W_2} + \mathbf{b}_2$$

$$\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$$

$$\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}, \mathbf{b}_2 \in \mathbb{R}^{d}$$

Compute transformation over each value in the sequence **independently**

Transformer Encoder: Final Add & Norm

After Final Add & Norm

$\mathbb{R}^{n \times d_1}$

After Feed Forward

$\mathbb{R}^{n \times d_1}$

After Add & Norm

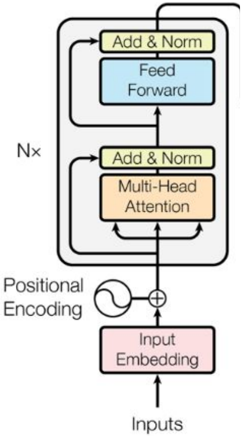
$\mathbb{R}^{n \times d_1}$

After Multi-Head Attention

$\mathbb{R}^{n \times d_1}$

Embedded source sequence

$\mathbb{R}^{n \times d_1}$



Source sequence $(x_1, \ldots, x_n)$

Add & Norm:

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

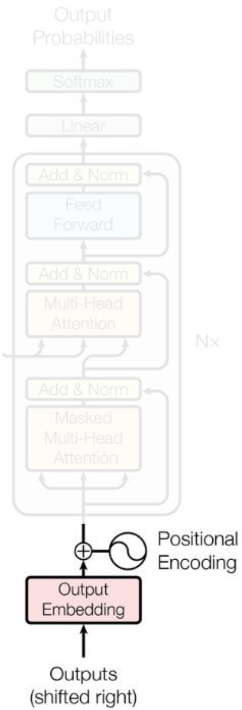
LayerNorm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Transformer Decoder:

Embedded target sequence

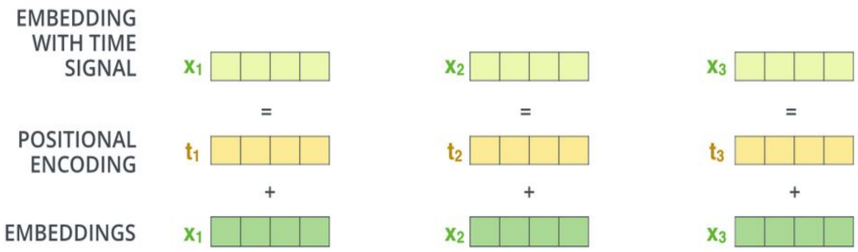
$\mathbb{R}^{m \times d_1}$



Target sequence

(<bos>, x_1, …, x_m)

Output and Positional Embedding



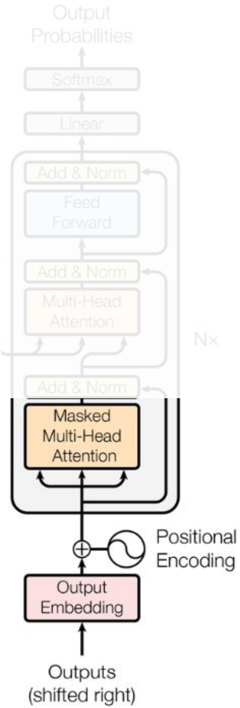
Transformer Decoder: Masked Multi-Head Attention

Masked Multi-Head Attention

$\mathbb{R}^{m \times d_1}$

Embedded target sequence

$\mathbb{R}^{m \times d_1}$

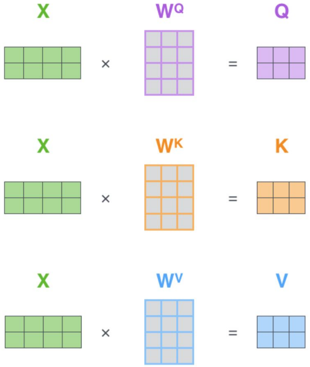


Target sequence

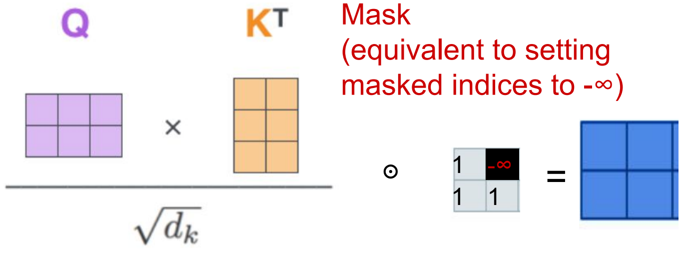
(<bos>, x_1, …, x_m)

Masked Self-Attention: $W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$

Step 1:

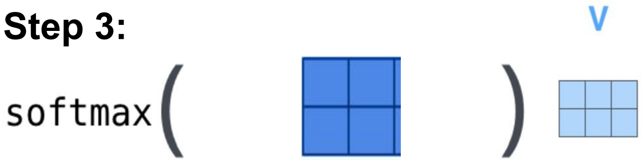


Step 2:



Elementwise Multiply by Mask
(equivalent to setting masked indices to -∞)

Step 3:

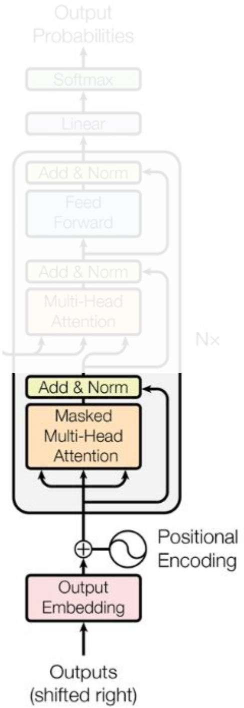


MultiHead Attention: $W^O \in \mathbb{R}^{d \times d_2}$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O$$

$$\text{head}_i = \text{Attention}(X W_i^Q, X W_i^K, X W_i^V)$$

Transformer Decoder:



After Add & Norm
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

Embedded target sequence
$\mathbb{R}^{m \times d_1}$

Target sequence
(<bos>, x_1, …, x_m)

Add & Norm:

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

LayerNorm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

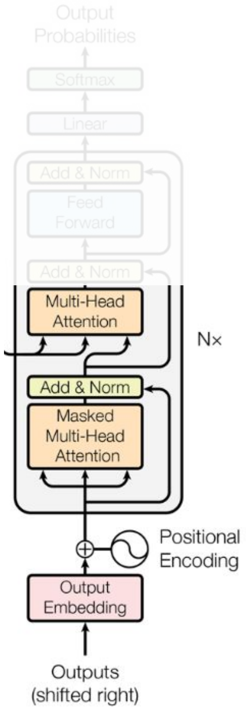
Transformer Decoder: Multi-Head (Cross) Attention

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

After Add & Norm
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

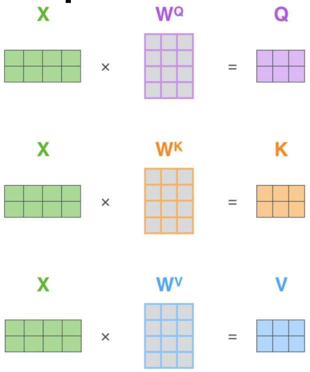
Embedded target sequence
$\mathbb{R}^{m \times d_1}$



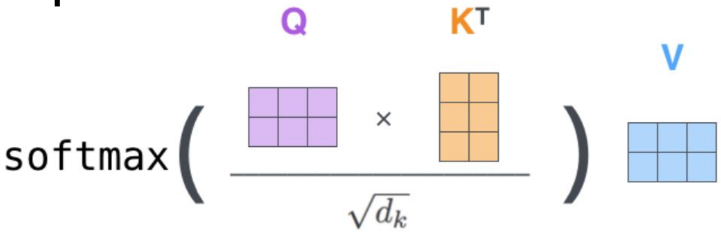
Target sequence
(<bos>, x_1, …, x_m)

Cross-Attention: $W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$

Step 1:



Step 2:



"Cross" attention means Q, K, V are computed from **separate** sequences

MultiHead Attention: $W^O \in \mathbb{R}^{d \times d_2}$

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, \ldots, \mathrm{head_h})W^O$$

$$\mathrm{head}_i = \mathrm{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

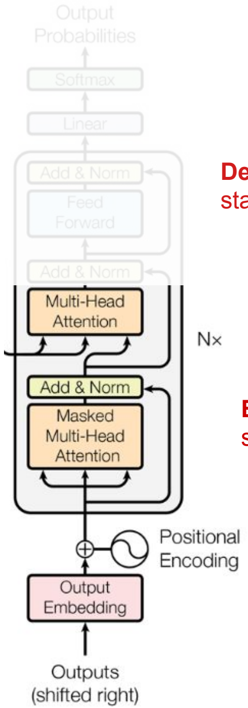
Transformer Decoder: Multi-Head (Cross) Attention

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

After Add & Norm
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

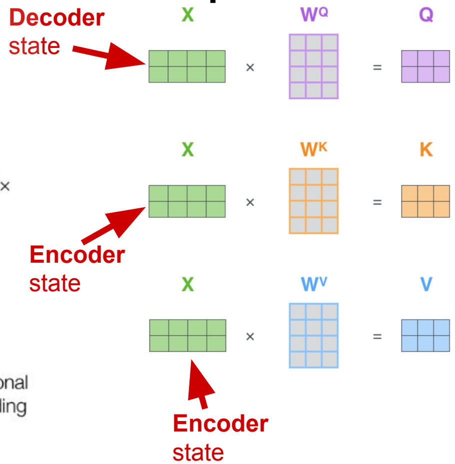
Embedded target sequence
$\mathbb{R}^{m \times d_1}$



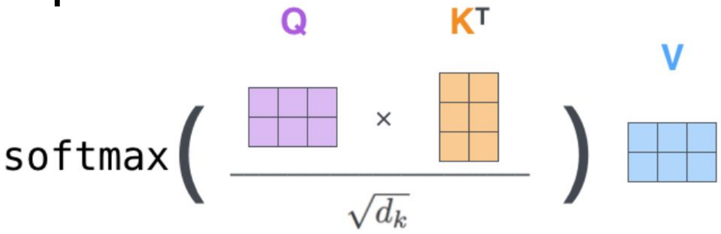
Target sequence
(<bos>, x_1, …, x_m)

Cross-Attention: $W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$

Step 1:



Step 2:



"Cross" attention means Q, K, V are computed from **separate** sequences

MultiHead Attention: $W^O \in \mathbb{R}^{d \times d_2}$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

Transformer Decoder: Add & Norm

Add & Norm

$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention

$\mathbb{R}^{m \times d_1}$

After Add & Norm

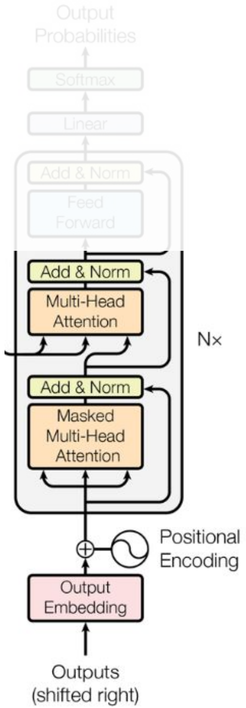
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention

$\mathbb{R}^{m \times d_1}$

Embedded target sequence

$\mathbb{R}^{m \times d_1}$



Target sequence
(<bos>, x_1, …, x_m)

Add & Norm:

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

LayerNorm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Transformer Decoder: Feed Forward

Feed Forward

$\mathbb{R}^{m \times d_1}$

Add & Norm

$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention

$\mathbb{R}^{m \times d_1}$

After Add & Norm

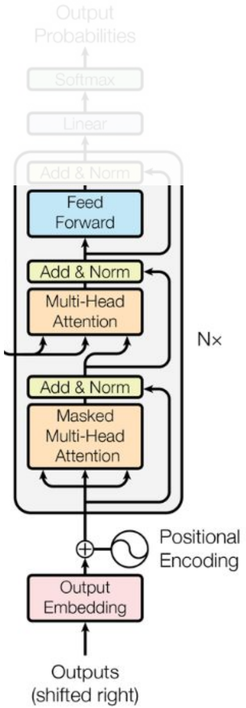
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention

$\mathbb{R}^{m \times d_1}$

Embedded target sequence

$\mathbb{R}^{m \times d_1}$



Target sequence
(<bos>, x_1, …, x_m)

Feed Forward

$$\mathrm{FFN}(\mathbf{x}_i) = \mathrm{ReLU}(\mathbf{x}_i \mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

$$\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$$

$$\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}, \mathbf{b}_2 \in \mathbb{R}^{d}$$

Transformer Decoder: Add & Norm

Add & Norm
$\mathbb{R}^{m \times d_1}$

Feed Forward
$\mathbb{R}^{m \times d_1}$

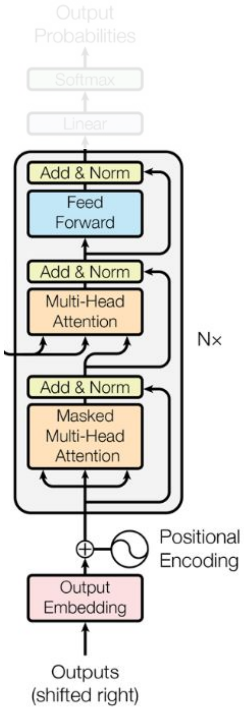
Add & Norm
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

After Add & Norm
$\mathbb{R}^{m \times d_1}$

Masked Multi-Head Attention
$\mathbb{R}^{m \times d_1}$

Embedded target sequence
$\mathbb{R}^{m \times d_1}$



Target sequence
(<bos>, x_1, …, x_m)

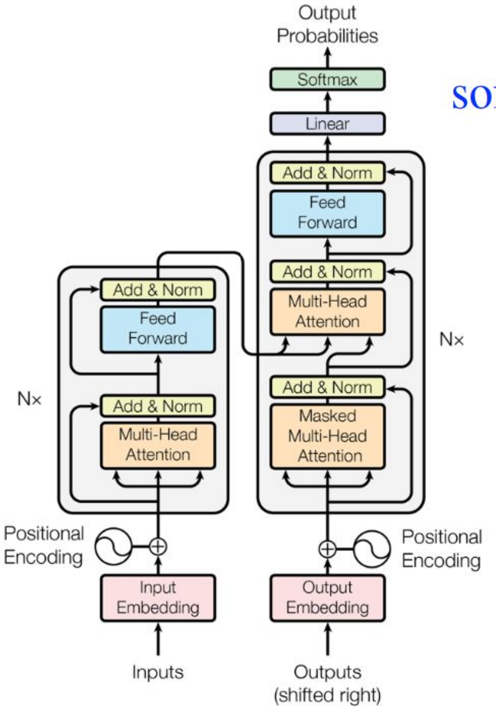
Add & Norm:

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

LayerNorm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Transformer: Final output



$$\text{softmax}(\mathbf{W}_o \mathbf{h}_i)$$

Compute transformation over **concatenated states**

Unembedding and token predictions

- Model predictions for position i are derived by applying the model's *unembedding matrix* $U \in \mathbb{R}^{n_\mathcal{V}, d_\mathcal{M}}$ to $\mathbf{x}_i^{(n_\mathcal{L})}$ (final layer embedding at position i) to obtain a vector of *output logits* $\text{logits}_i(\mathbf{t}) \in \mathbb{R}^{n_\mathcal{V}}$:

$$\text{logits}_i(\mathbf{t}) = U\,\mathbf{x}_i^{(n_\mathcal{L})}$$

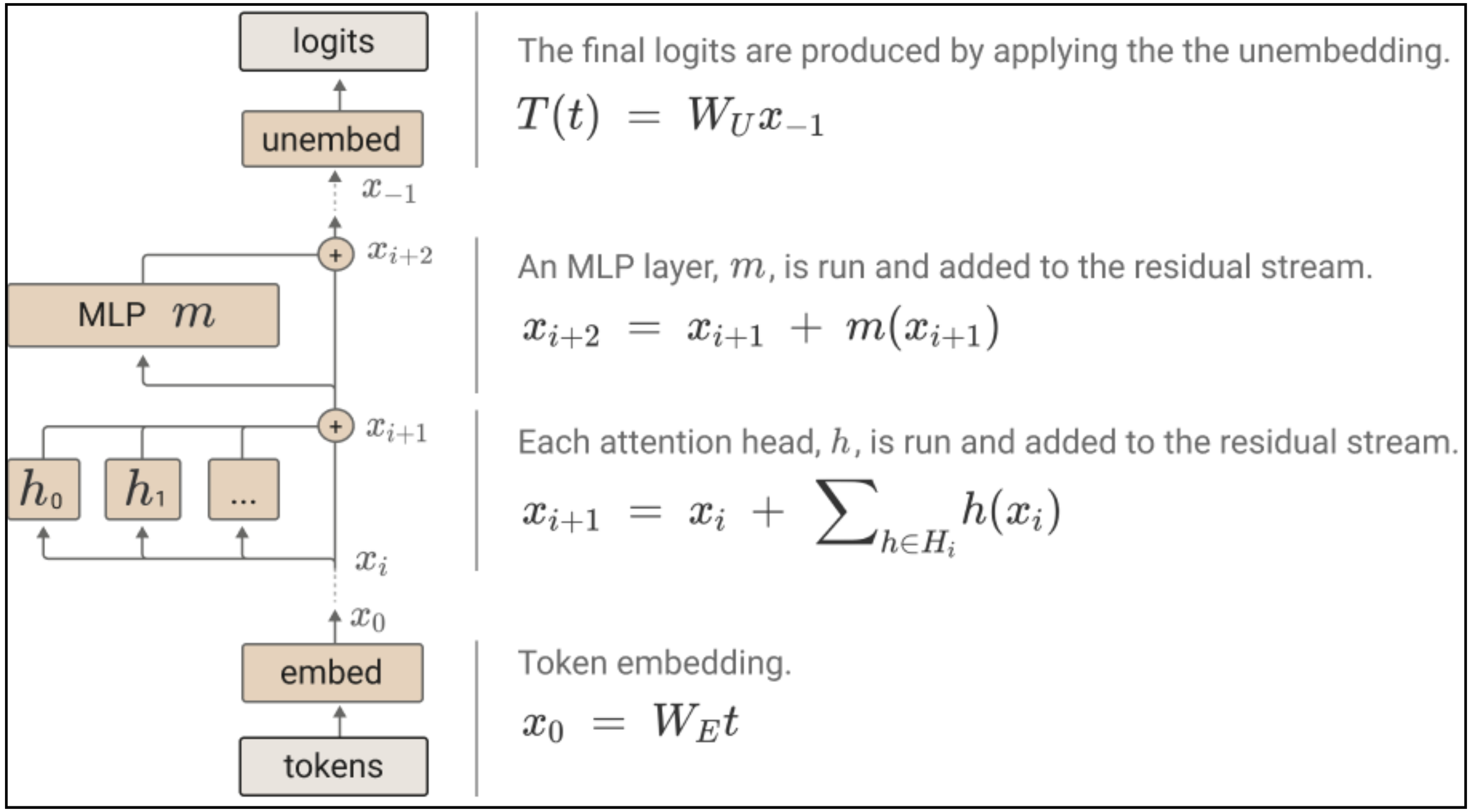
- Logits are transformed into probabilities using softmax:

$$P_\mathcal{M}(t \mid \mathbf{t}, i) = \frac{\exp\left(\text{logits}_i(\mathbf{t})_{\text{Ind}(t)}\right)}{\sum_{j=1}^{n_\mathcal{V}} \exp\left(\text{logits}_i(\mathbf{t})_j\right)}$$

  - (autoregressive) next-token prediction models predict the *next* token at $i+1$
  - (masked) missing-token prediction models (may) predict the the *current* token at i

Transformers as Updated Residual Streams

- sequence of additive updates
  - old info often more important than new info
- same function for each token at each layer
  - only attention module assesses info “laterally”
- final unembedding applicable to each intermediate processing stage
  - early-decoding / logit lense
- ~ “internal memory during processing”
  - a bit like human working memory
  - but “lateral access” to WM @ previous stages
    - “memory of working memory”
- processing stages similar to human processing in time?

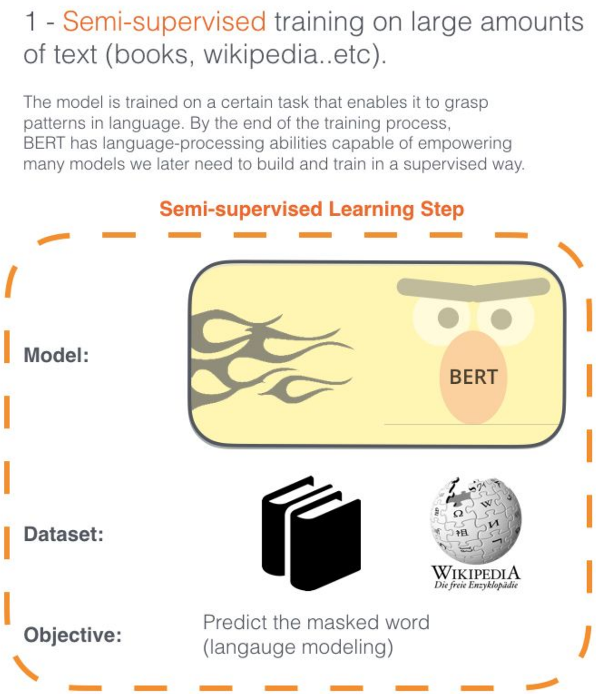


Elhage et al. (2021)

BERT (Devlin et al., 2019)

- Bidirectional Encoder Representations from Transformers
- Key idea: Use a transformer to leverage **bidirectional context**
- Two objectives/loss optimization
  - Masked language modeling (MLM)
  - Next sentence prediction (NSP)
- Impact: one of the first works in NLP showing strong performance using a pre-trained transformer

BERT Pre-training



We first pre-train the model on a lot of data to learn basic language abilities.

[from Alammar, The Illustrated BERT, http://jalammar.github.io/illustrated-bert/]

BERT Pre-training

Key design choice: use a **bidirectional Transformer encoder** instead of a left-to-right decoder.

> **Why can't we just train a bidirectional LM?**
> A standard LM predicts the next token, so it can't look right. BERT's solution: predict **randomly masked** tokens using *both* left and right context.

⇒ **Masked LM (MLM):** mask 15% of tokens; predict them from bidirectional context

80% → [MASK], 10% → random token, 10% → unchanged

⇒ **Next Sentence Prediction (NSP):** predict if sentence B follows sentence A

Later ablations (Liu et al., 2019) show NSP often doesn't help — removed in RoBERTa

BERT Model Sizes

	BERT-Base	BERT-Large
Layers	12	24
Hidden size	768	1024
Attn heads	12	16
Parameters	110M	340M

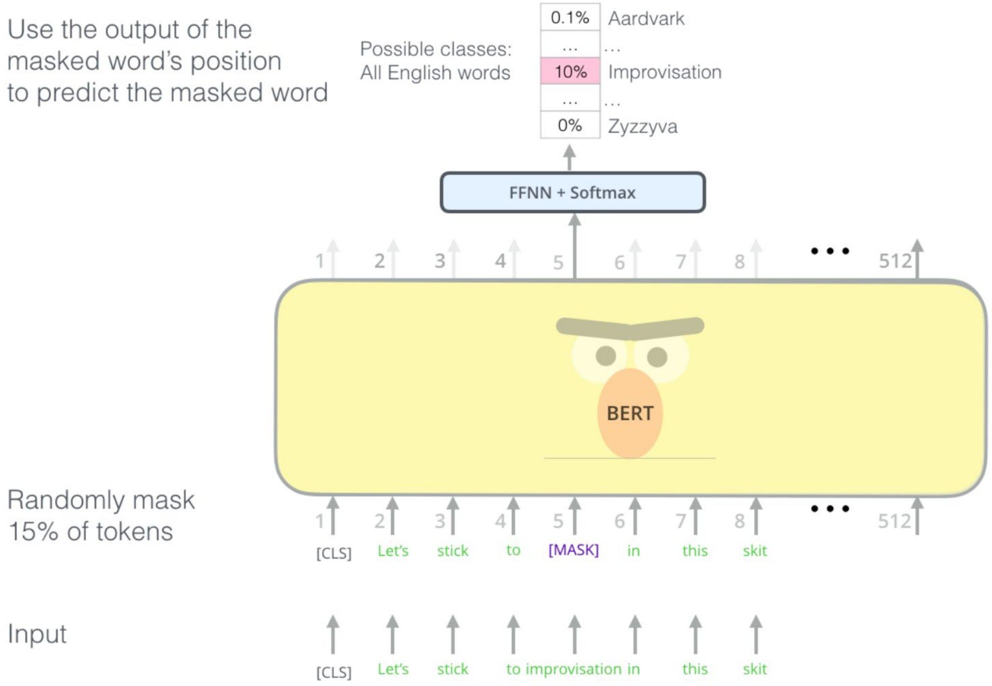
Training Setup

Corpus	Wikipedia (2.5B) + BooksCorpus (0.8B)
Max seq len	512 wordpiece tokens
Vocab size	30,000 wordpieces
Training	1M steps, batch 128K

Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers," NAACL 2019 | Released Oct 2018

Masked Language Modeling (MLM)

Learn to recover a masked word using the context



[from Alammar, The Illustrated BERT, http://jalammar.github.io/illustrated-bert/]

MLM- The 80-10-10 Strategy

Of the 15% selected tokens, the corruption applied varies:

%	Action	Why?
80%	Replace with `[MASK]`	Forces model to infer from context
10%	Replace with random token	Prevents over-reliance on [MASK]; model must check all positions
10%	Keep unchanged	Biases representation toward the true token

Train/inference mismatch: [MASK] tokens never appear at fine-tuning time. The 10/10 mix closes this gap — the model learns that *any* position may need correction, not just [MASK] positions.

Original sentence:

`[CLS]` `the` `man` `went` `to` `the` `store` `[SEP]`

After 80/10/10 corruption:

`[CLS]` `the` `man` `[MASK]` `to` `the` `running` `[SEP]`

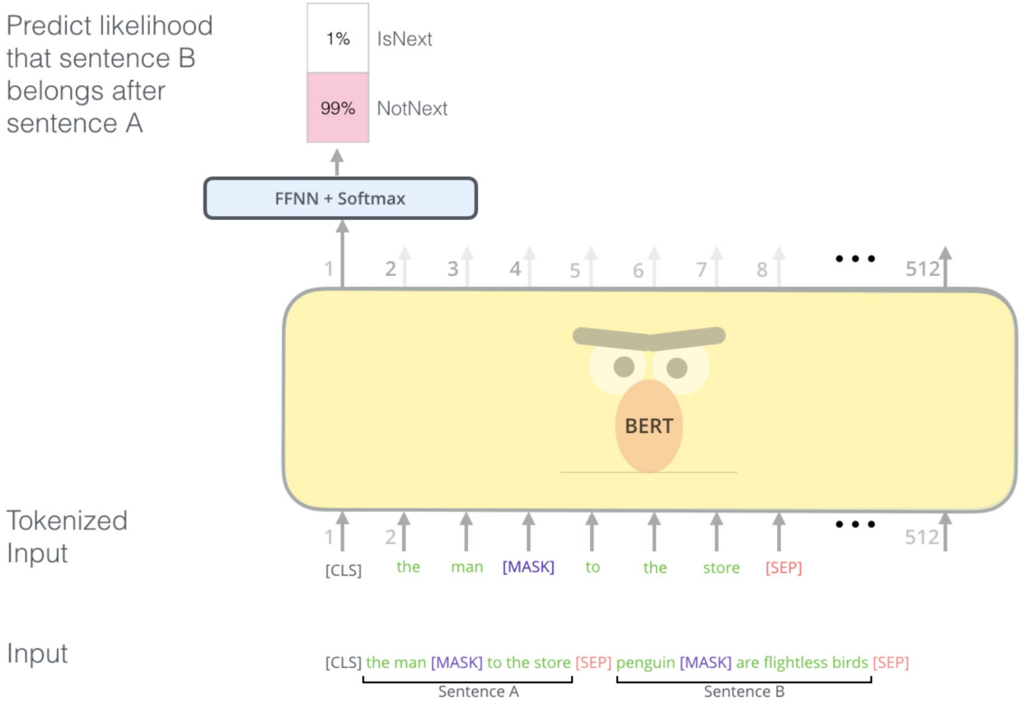
"went" → [MASK] (80%)

"store" → "running" (10%, random word)

Loss computed only at masked positions.

predict: `"went"` and `"store"`

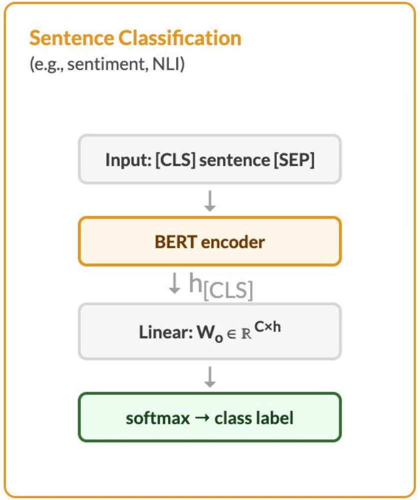
Next Sentence Prediction (NSP)

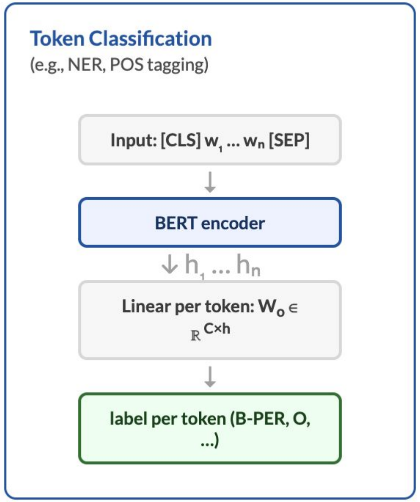


Later works showed that this doesn't always help (Liu et al., 2019)!

[from Alammar, The Illustrated BERT, http://jalammar.github.io/illustrated-bert/]

How to use BERT? Fine-tuning





All parameters updated. Both the original BERT weights and the new task head are trained jointly on labeled data:

$P(y=k) = \text{softmax}_k(W_o \cdot h_{[CLS]})$

Why does this work? BERT's pre-training builds rich contextual representations. Fine-tuning just teaches the model *which aspects* of those representations are relevant for the task — requires very little labeled data.

"Pretrain once, fine-tune many times." — A single task head is added on top of the frozen-then-updated BERT encoder.

Fini