CS 383: Machine Learning

Prof Adam Poliak

Fall 2024

11/19/2024

Lecture 25

Announcements – Remaining Assignments

HW07: due Wednesday 11/27

HW08: due Friday 12/06 (might extend this too)

Project Presentations – 12/17 10:00am

Outline

Issues when training NNs

Pytorch

FFN’s for variable length input

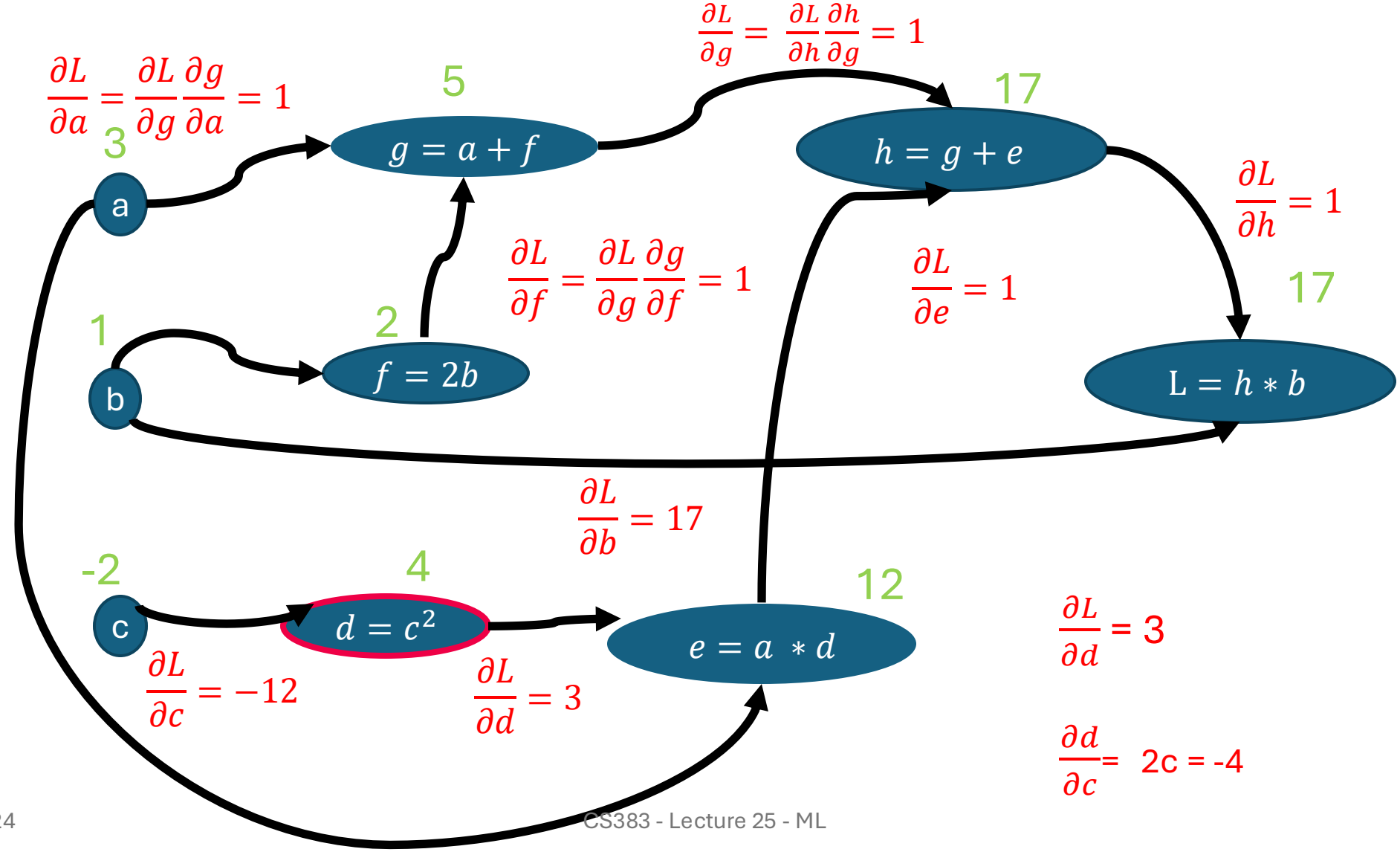
RNN

LSTM

Attention

Self Attention

backward pass



Exploding gradient

The gradient can accumulate, becoming very big

Issues:

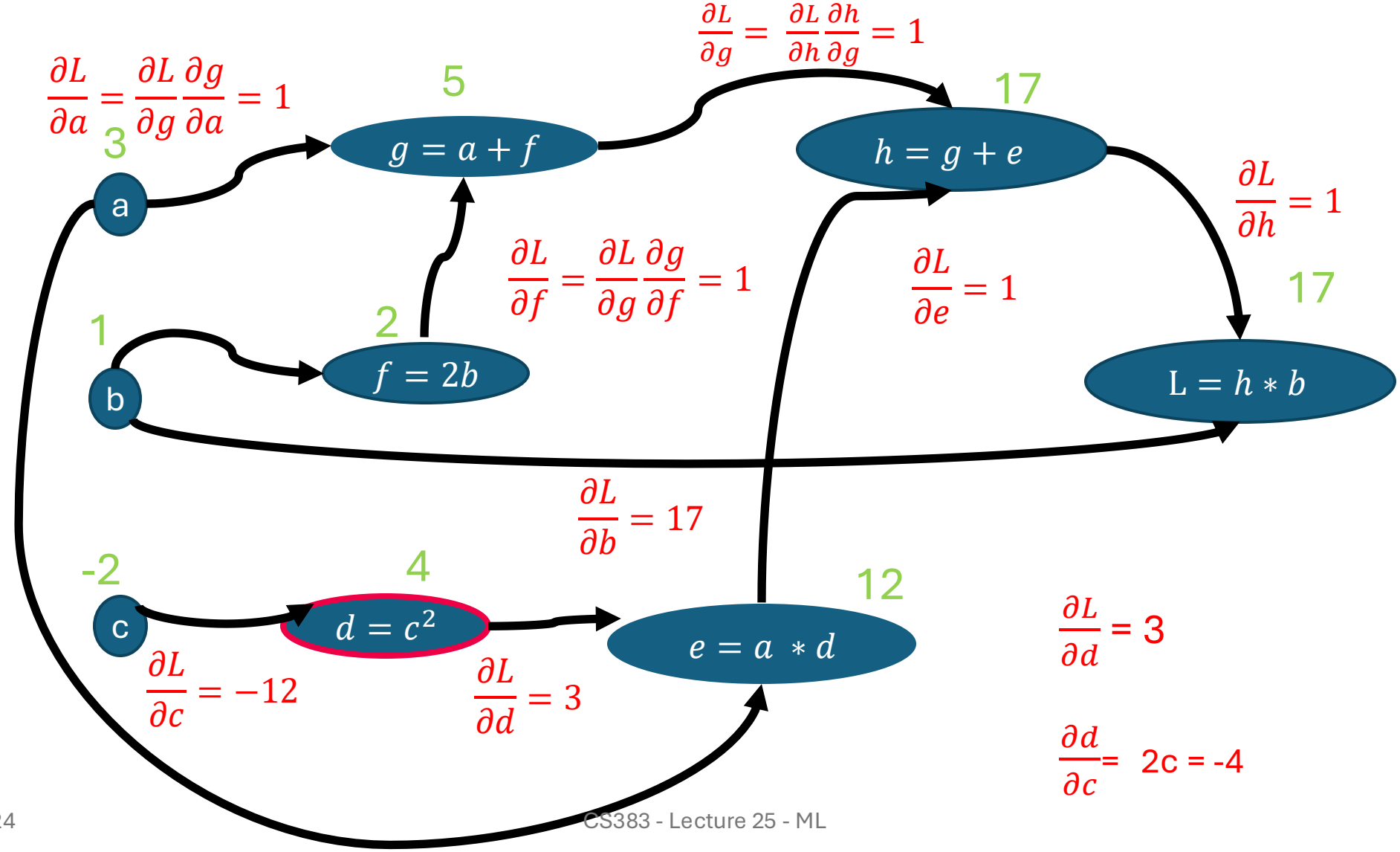
- might move our weights too much
- result in Nan

Solution:

Clipping

- Maximum value for gradients
- Can be dynamic

backward pass



Vanishing gradient

The gradient become 0

Issues:

wont be able to update weights (because 0 gets passed all the way back)

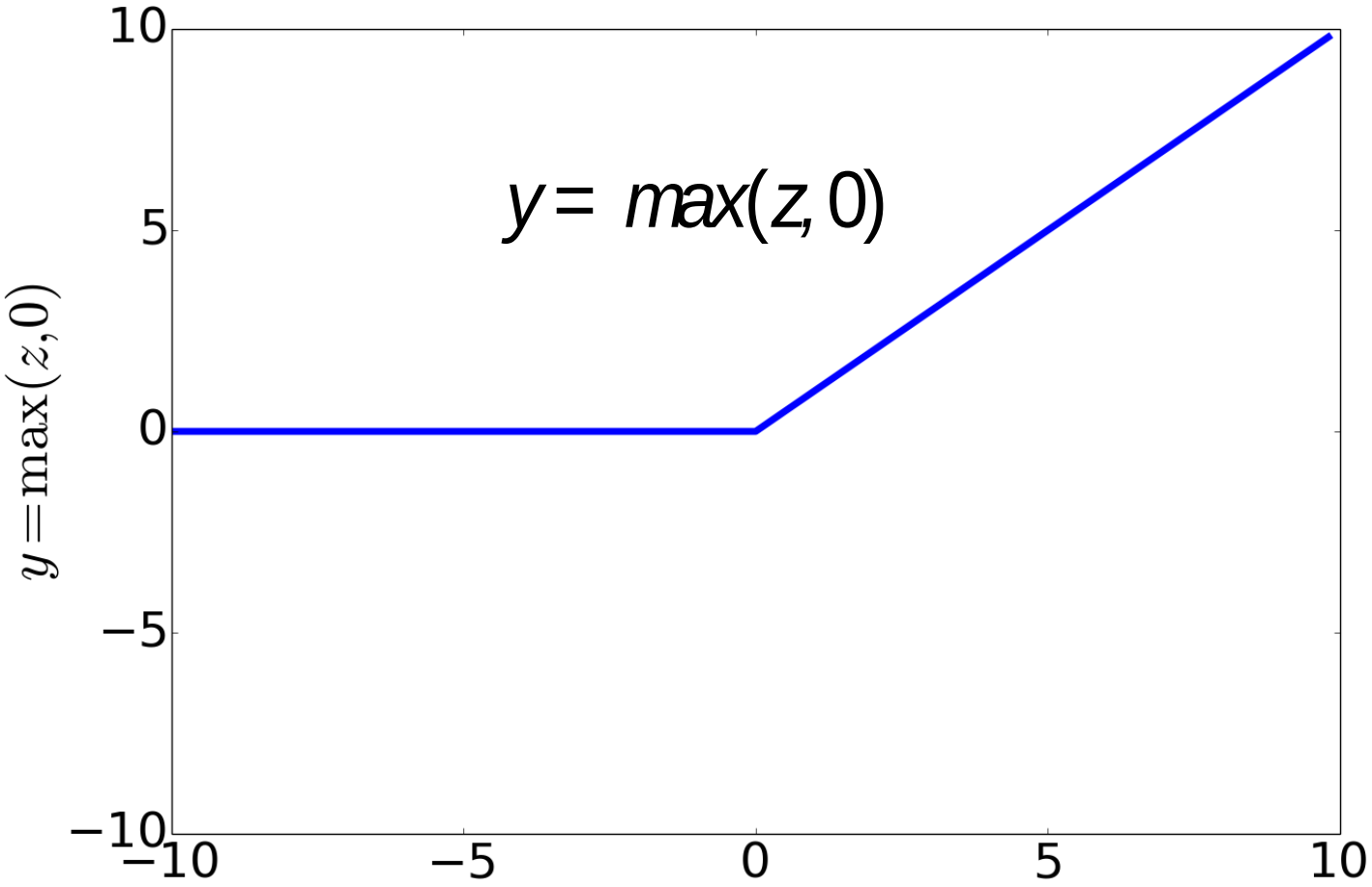
stuck in a local optima

Solution:

ReLU activation function

z = max(0, z)

ReLU



One node view

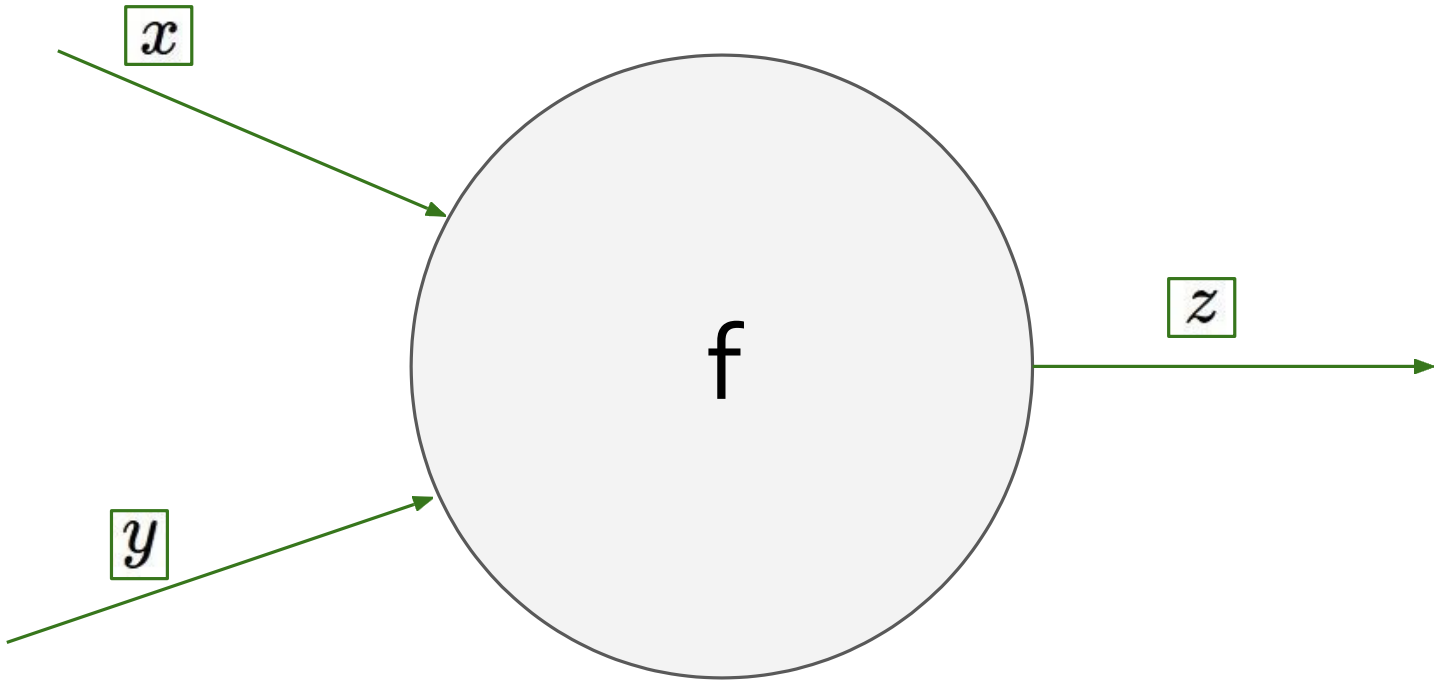


Figure from Andrej Karpathy

Dead neuron

In forward pass, output of a node w/ ReLU activation often will be 0

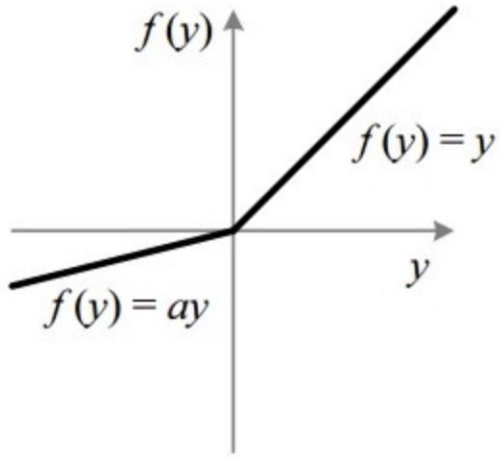
Issues:

wont pass information from one node to the next

lots of useless nodes

Solution:

Leaky ReLU activation function



Outline

Issues when training NNs

Pytorch

FFN's for variable length input

RNN

LSTM

Attention

Self Attention

Pytorch

Torch: Facebook’s deep learning framework

Originally written in Lua (C backend)

Optimized to run computations on GPU

Mature, industry-supported framework

Defining a model

```
import torch
from torch import nn


class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        return out
```

nn.Module

Base class for all neural network modules.

Creates a computation graph

Define the model in __init__

Specify how to make predictions in forward

If only use built-in modules, no need to implement backprop

Defining a model

```
import torch
from torch import nn

class FNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(FNN, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        # no activation and no softmax at the end
        return out
```

Train a model

Define:

- Loss function
- Learning algorithm (e.g. SGD)
- Learning rate
- Number of epochs

```
num_epochs = 100
learning_rate = 0.003
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.CrossEntropyLoss()
```

Train a model

In each iteration:

- Make a prediction
- Compute the loss
- Autograd (Automatic differentiation), backprop
- Update the weights

```
optimizer.zero_grad()
prediction = model(X[i])
loss_val = loss_fn(prediction, labels[0][i])
loss_val.backward()
optimizer.step()
```

Train a model

```
# Training the Model
for epoch in range(num_epochs):
  num_correct = 0
  for i in range(100):
    optimizer.zero_grad()
    prediction = model(X[i])
    loss_val = loss_fn(prediction, labels[0][i])
    loss_val.backward()
    optimizer.step()

  print(f"loss at epoch {epoch}: {loss_val}")
  print(f"accuracy at epoch {epoch}: {num_correct / 100}")
```

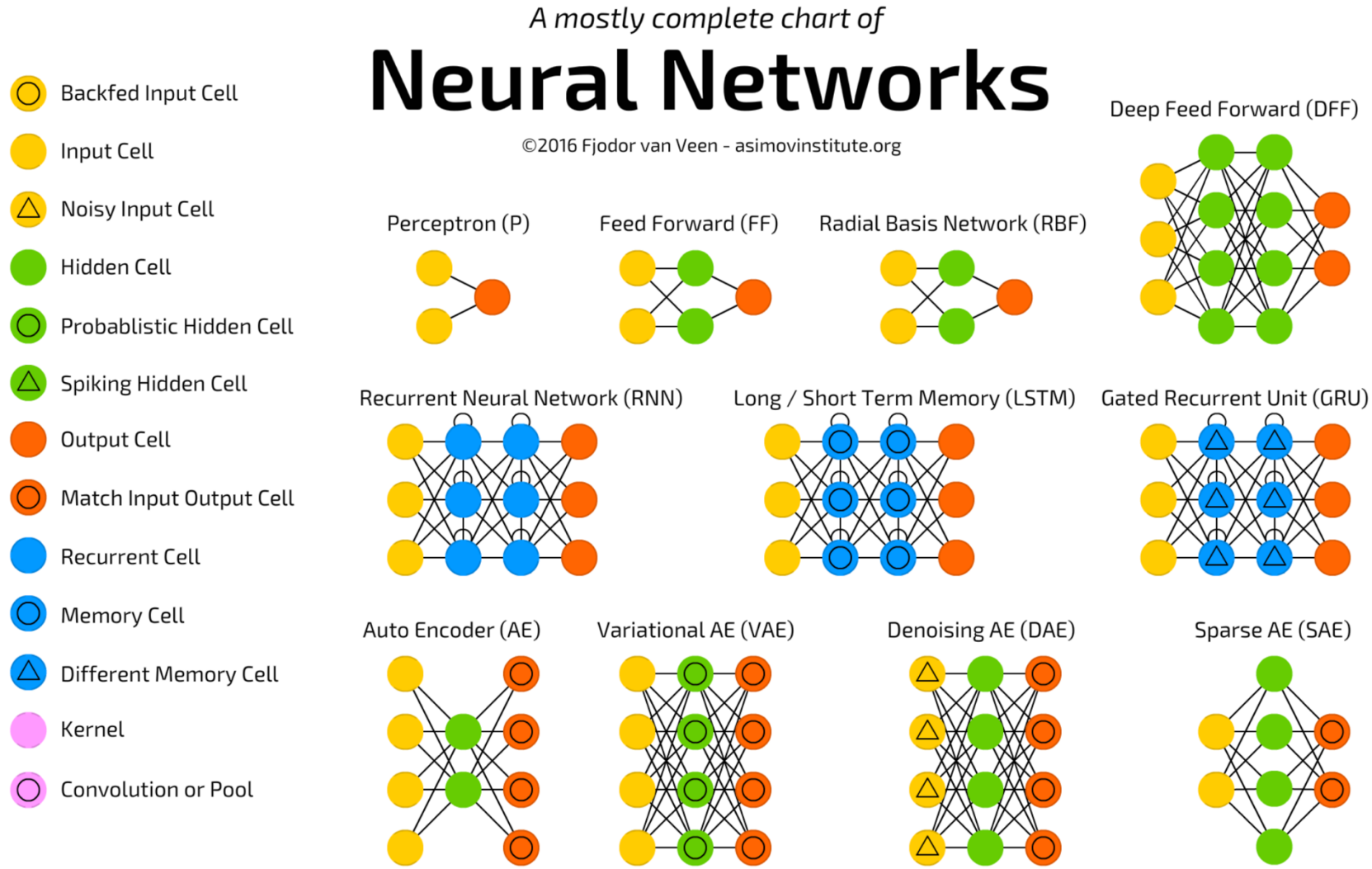


Image from Neural Network Zoo: http://www.asimovinstitute.org/neural-network-zoo/

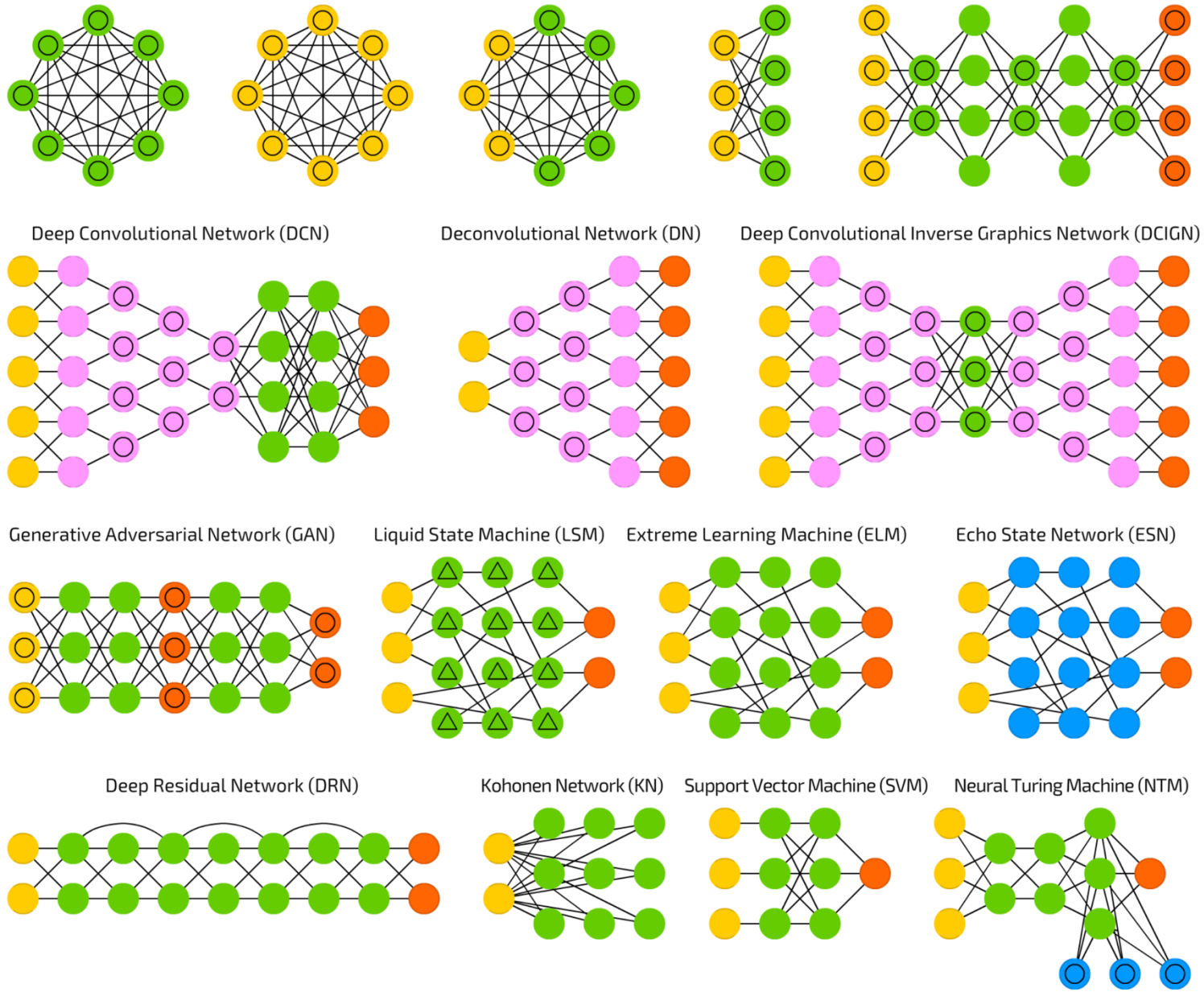


Image from Neural Network Zoo: http://www.asimovinstitute.org/neural-network-zoo/

Outline

Issues when training NNs

Pytorch

FFN’s for variable length input

RNN

LSTM

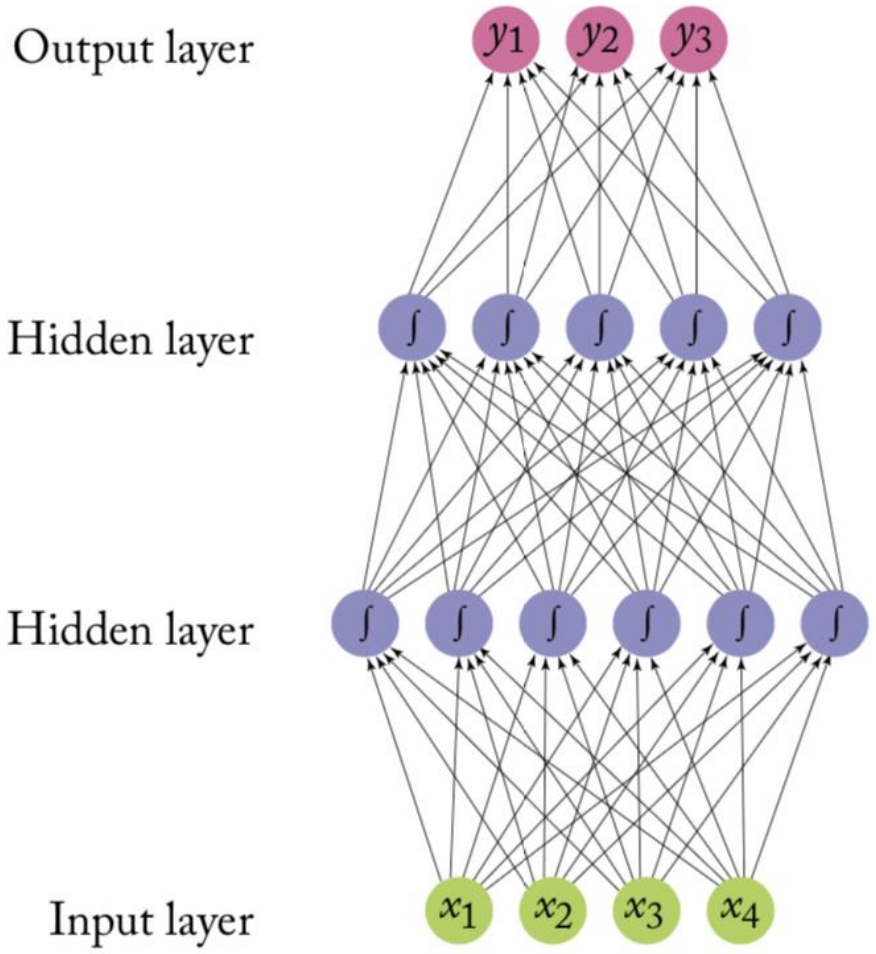
Attention

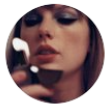
Self Attention

Classify a tweet as viral or not

Classify a tweet as viral or not

Classify a tweet as viral or not

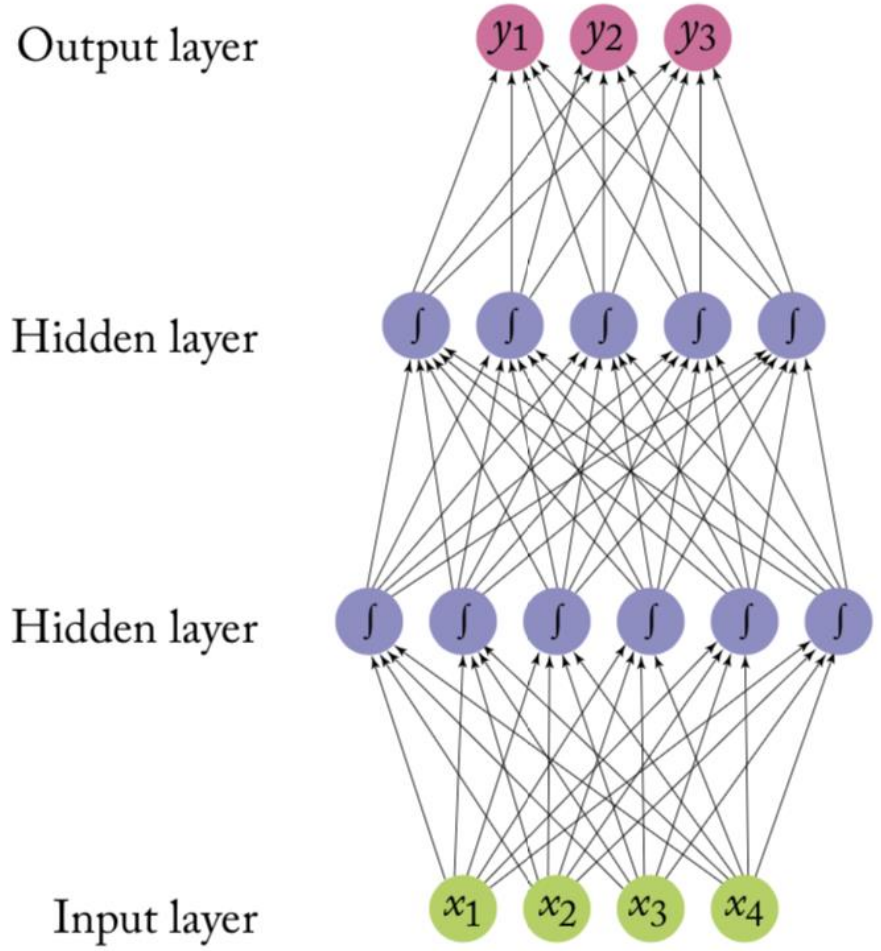


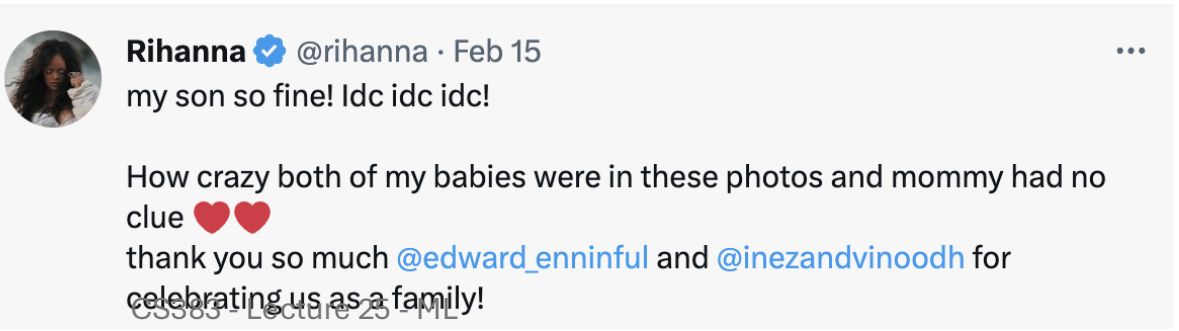


Taylor Swift @taylorswift13 · Jan 27

The Lavender Haze video is out now. There is lots of lavender. There is lots of haze. There is my incredible costar @laith_ashley who I absolutely adored working with.

Classify a tweet as viral or not





Rihanna @rihanna · Feb 15

my son so fine! Idc idc idc!

How crazy both of my babies were in these photos and mommy had no clue ❤️❤️
thank you so much @edward_enninful and @inezandvinoodh for celebrating us as a family!

Approach 1: Feature Engineering

Create a fixed-length vector to represent the input

Examples:

- Length of vector is V: size of vocabulary
- Value at index i represents static of word i
  - How many times word i appears in input
  - Term-frequency inverse document frequency of word i
    - Penalize words that appear in lots of inputs

Approach 1: Feature Engineering

Create a fixed-length vector to represent the input

$x = w_1, w_2, \ldots w_n$

$z_0 = [count(w_1), count(w_2), count(w_3), \ldots, count(w_v)]$

$\hat{y} = MLP(z_o)$

Approach 2: Deep Averaging Network

Represent each document as a continuous bag of words, averaging the word embeddings

$$x = w_1, w_2, \ldots w_n$$

$$z_0 = CBOW(w_1, w_2, \ldots w_n). CBOW = \sum_i E[w_i]$$

$$\hat{y} = MLP(z_o)$$

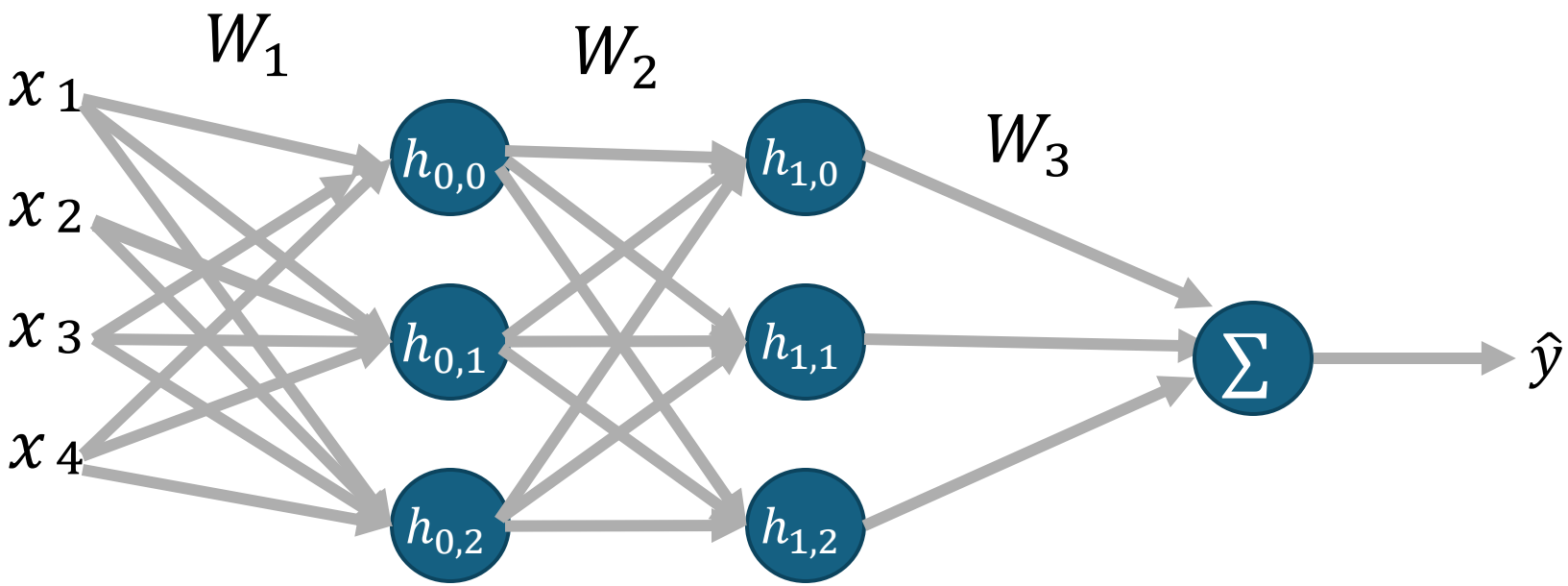
Multilayer Perceptron

Feed-forward NN

$$MLP_1 = g(xW_1 + b_1)W_2 + b_2$$

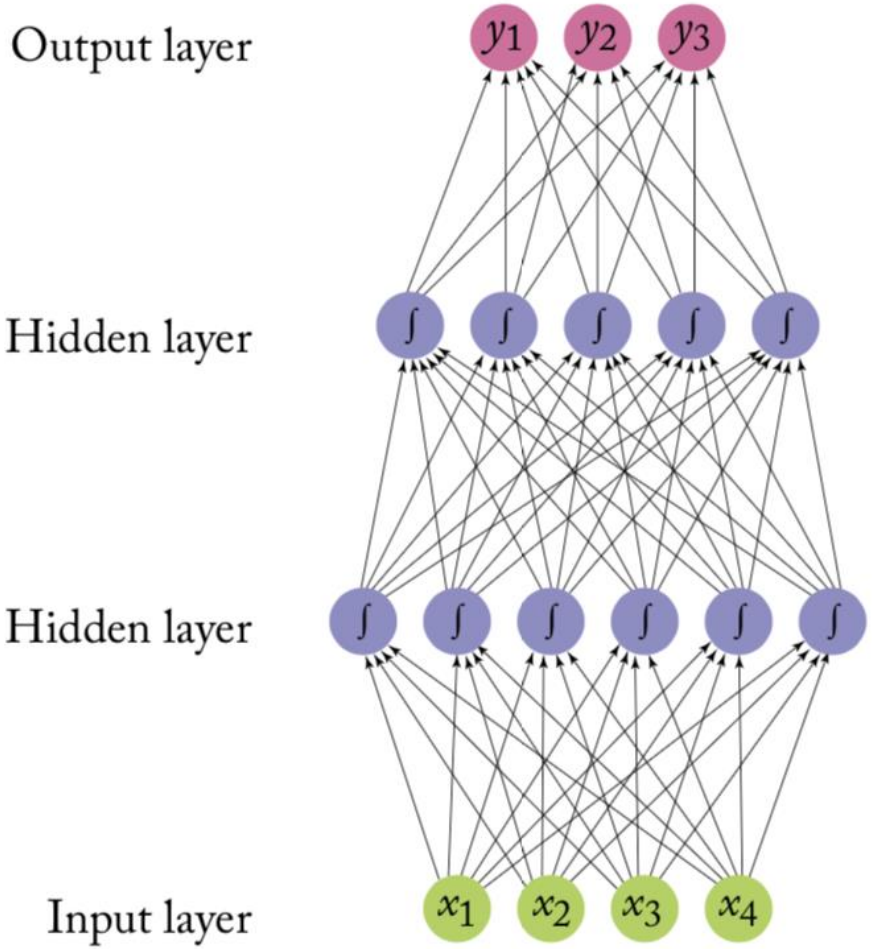
$$MLP_2 = g(g(xW_1 + b_1)W_2 + b_2)\,W_3 + b_3$$

MLP_2



$$\boldsymbol{h}_0 = \sigma(xW_1) \qquad \boldsymbol{h}_1 = \sigma(\sigma(xW_1)W_2$$

MLP_2


Output layer
Hidden layer
Hidden layer
Input layer
y_1
y_2
y_3
x_1
x_2
x_3
x_4

FFN’s issues

Fixed input size

Solutions:

1. Create a fixed length representation
2. Recurrent Neural Networks

Outline

Issues when training NNs

Pytorch

FFN’s for variable length input

RNN

LSTM

Attention

Self Attention

RNN - motivation

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

Recursion

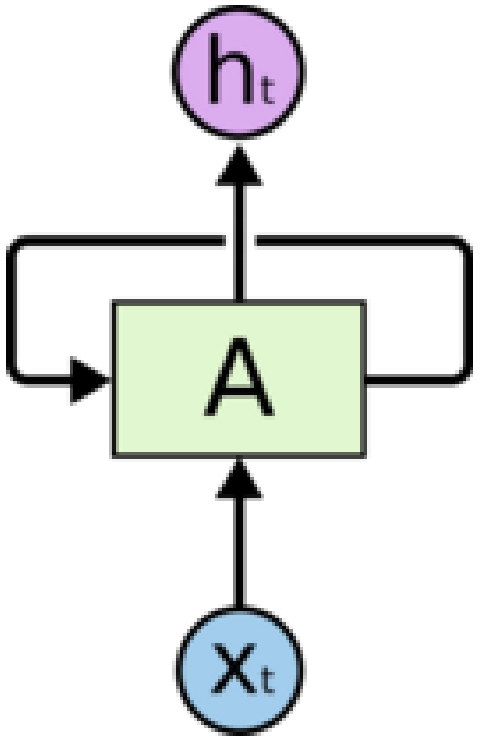
Recurrent Neural Networks are a family of NNs that learn sequential data via **recursive dynamics**

Recurrent Neural Network (RNN)

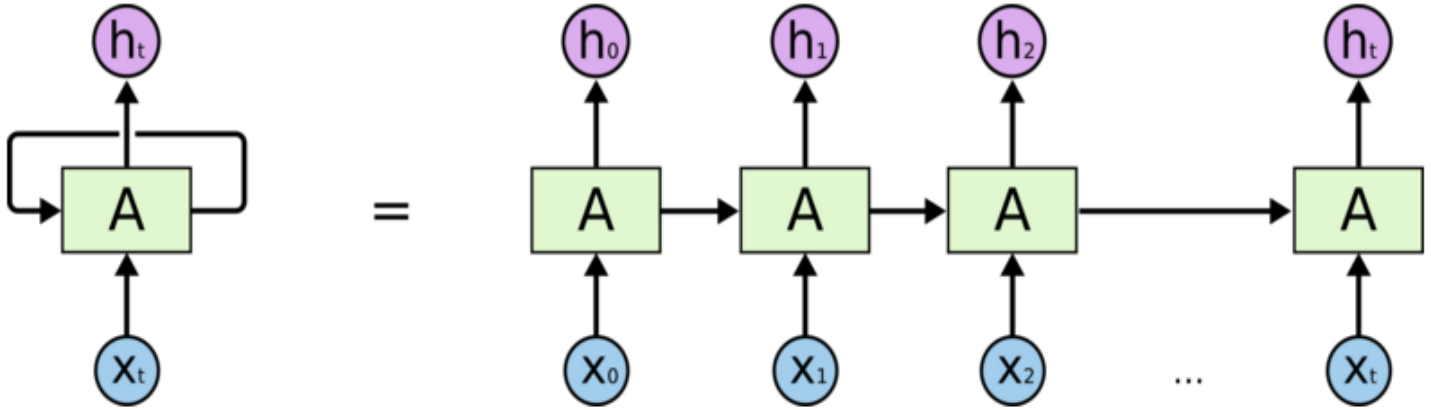
$h_t = f(h_{t-1}, x_t)$

In the diagram, $f(\dots)$ looks at some input x_t and its previous hidden state h_{t-1} and outputs a revised state h_t.

A loop allows information to be passed from one step of the network to the next.



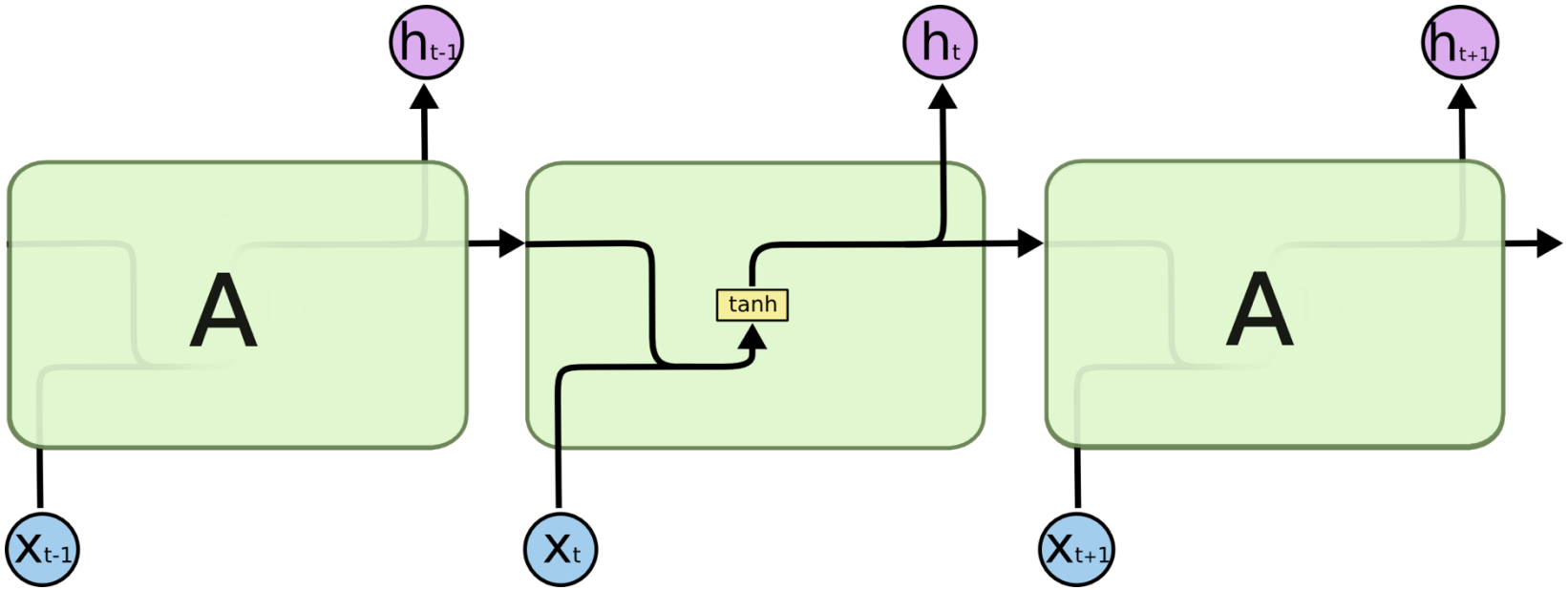
Unrolling an RNN



A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

RNN internal

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



Updating weights in a RNN

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

$$\frac{\partial \tan(x)}{\partial x} = 1 - \tan(x)^2$$

So what variables do we need to update?

Our weights and biases

Updating weights in a RNN

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

$$\frac{\partial \tan(x)}{\partial x} = 1 - \tan(x)^2$$

$$\frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \\ \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \end{bmatrix}$$

$$\frac{\partial \tan(x)}{\partial x} = 1 - \tan(x)^2$$

Updating weights in a RNN

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

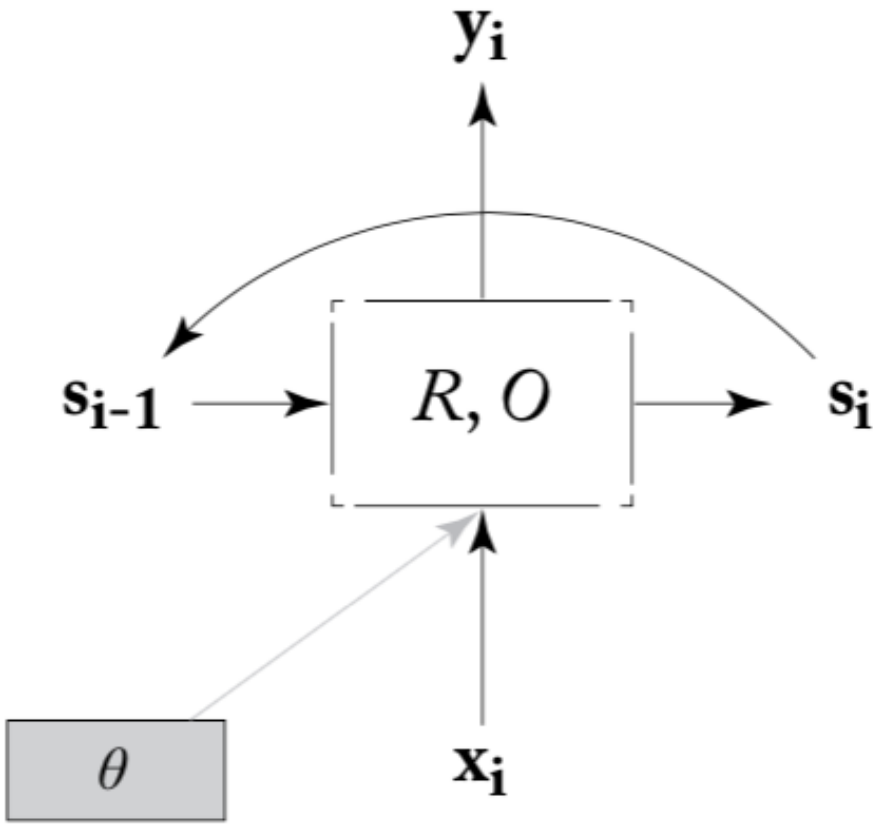
Lets let this be **g**

$$\frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \\ \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \end{bmatrix} = \begin{bmatrix} 1 - \tan(g)\, x_t \\ 1 - \tan(g) \\ 1 - \tan(g)\, h_{t-1} \\ 1 - \tan(g)\, x_t \end{bmatrix}$$

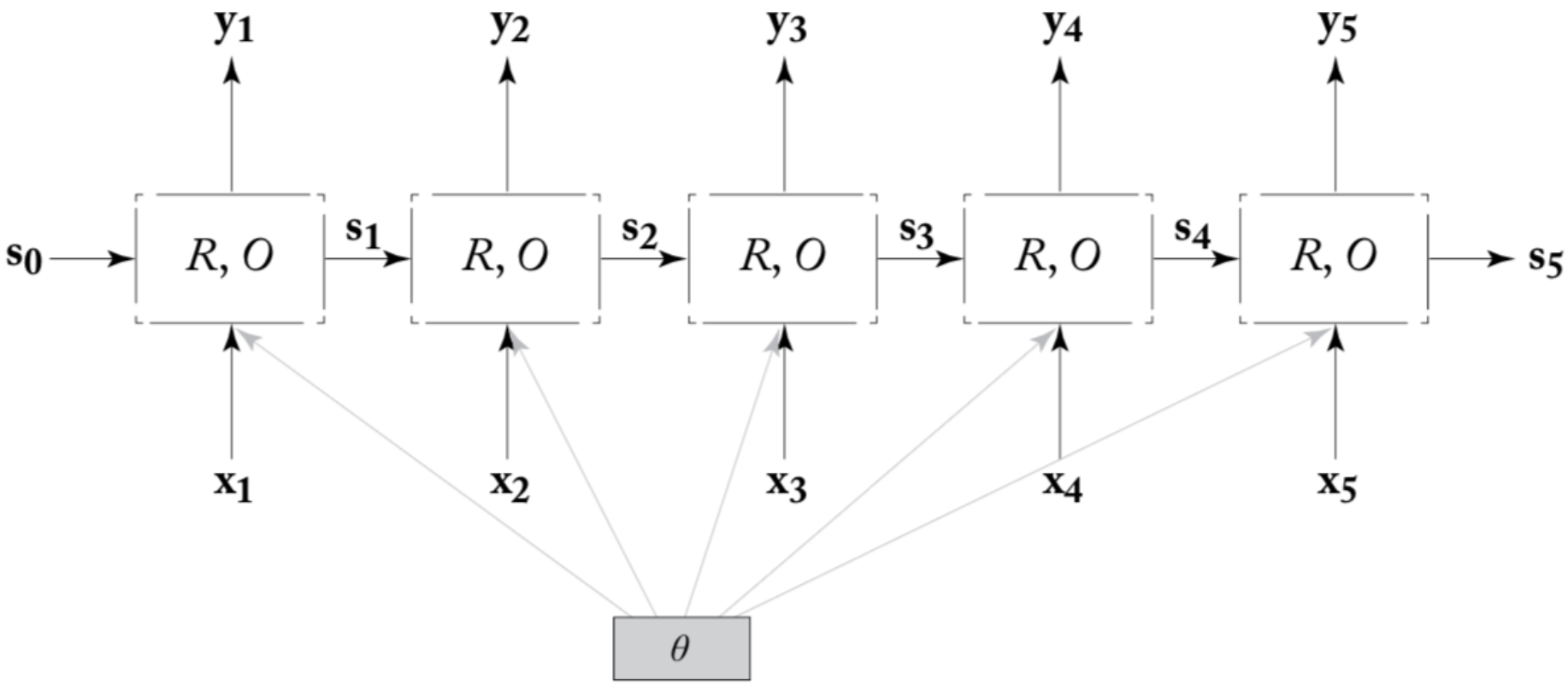
RNN cell

$$s_i = R(x_i, s_{i-1}, \theta)$$

$$\widehat{y_i} = O(s_i)$$

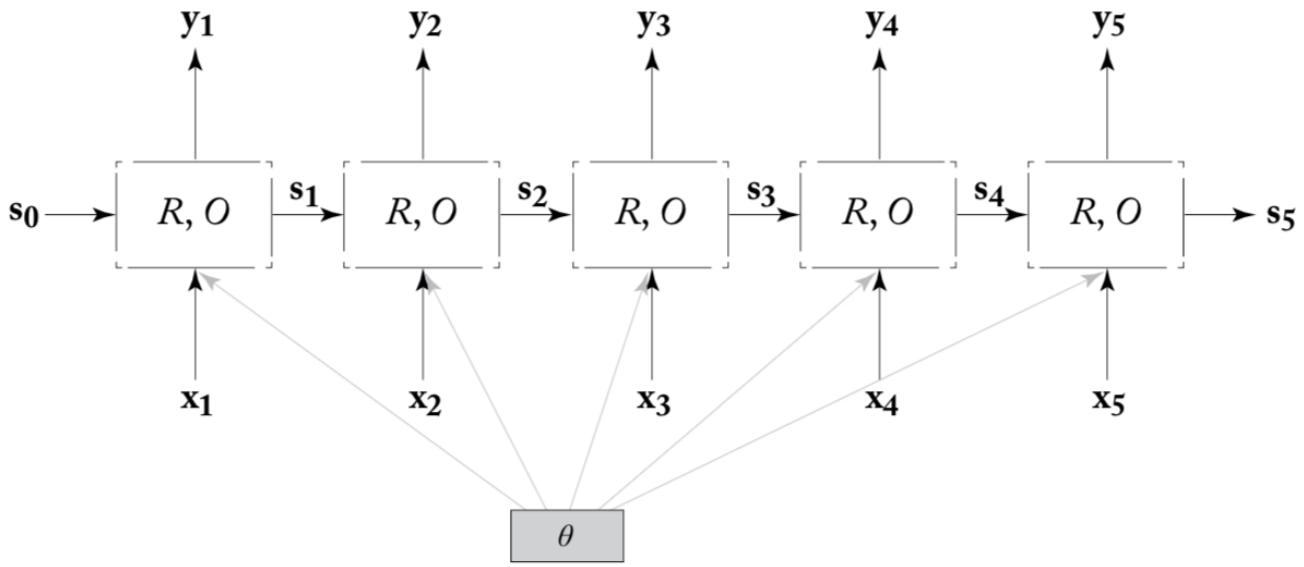


Unrolling RNN



Revisiting LM

$$P(x_t|x_{t-1}, x_{t-2}, \ldots x_1)$$

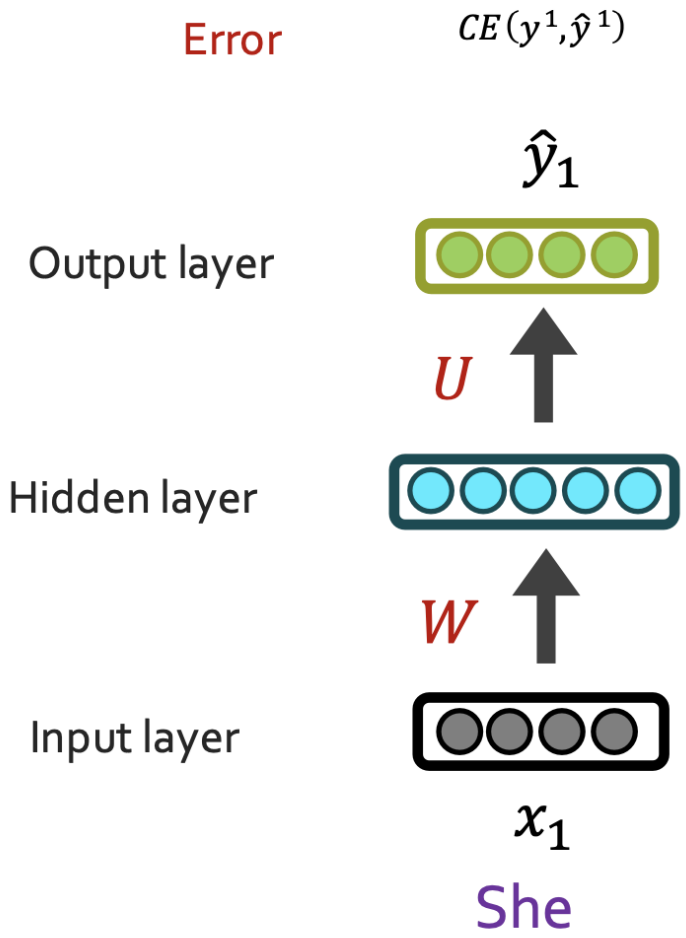


Pass in one word at a time

Compute probability over entire vocab by applying predictive head to last output

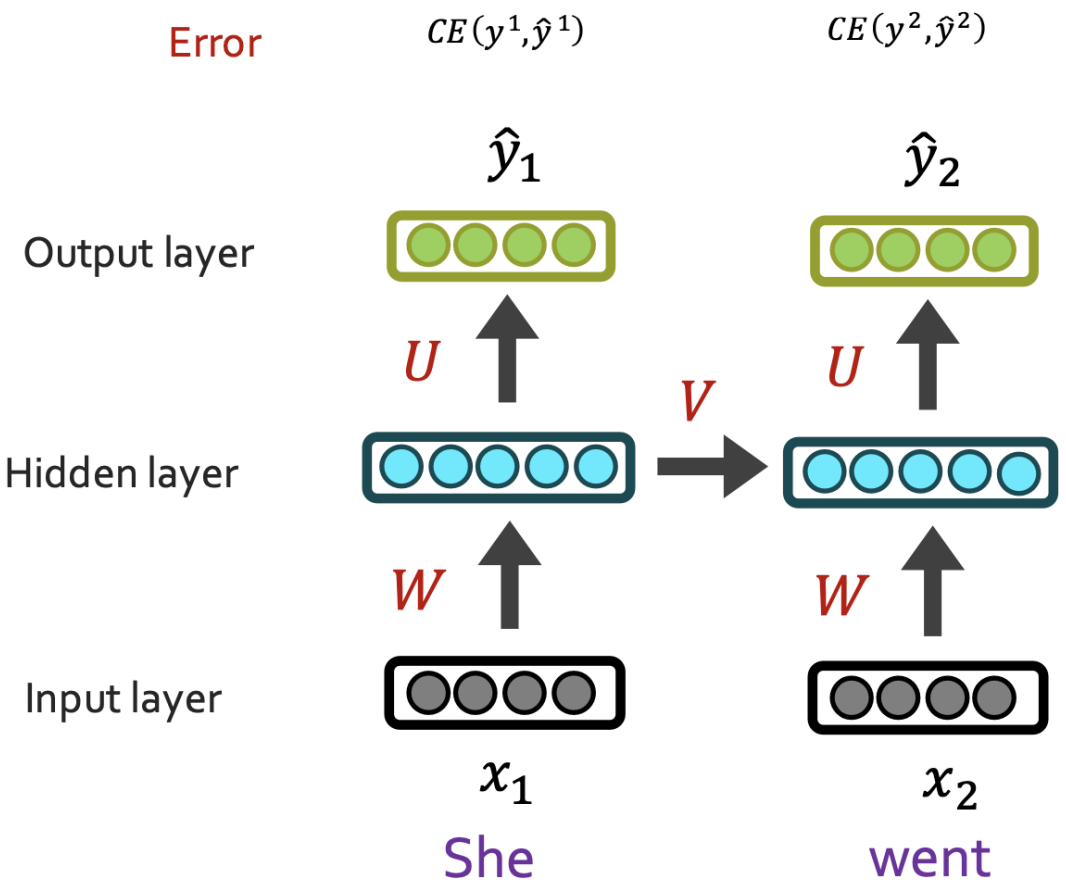
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



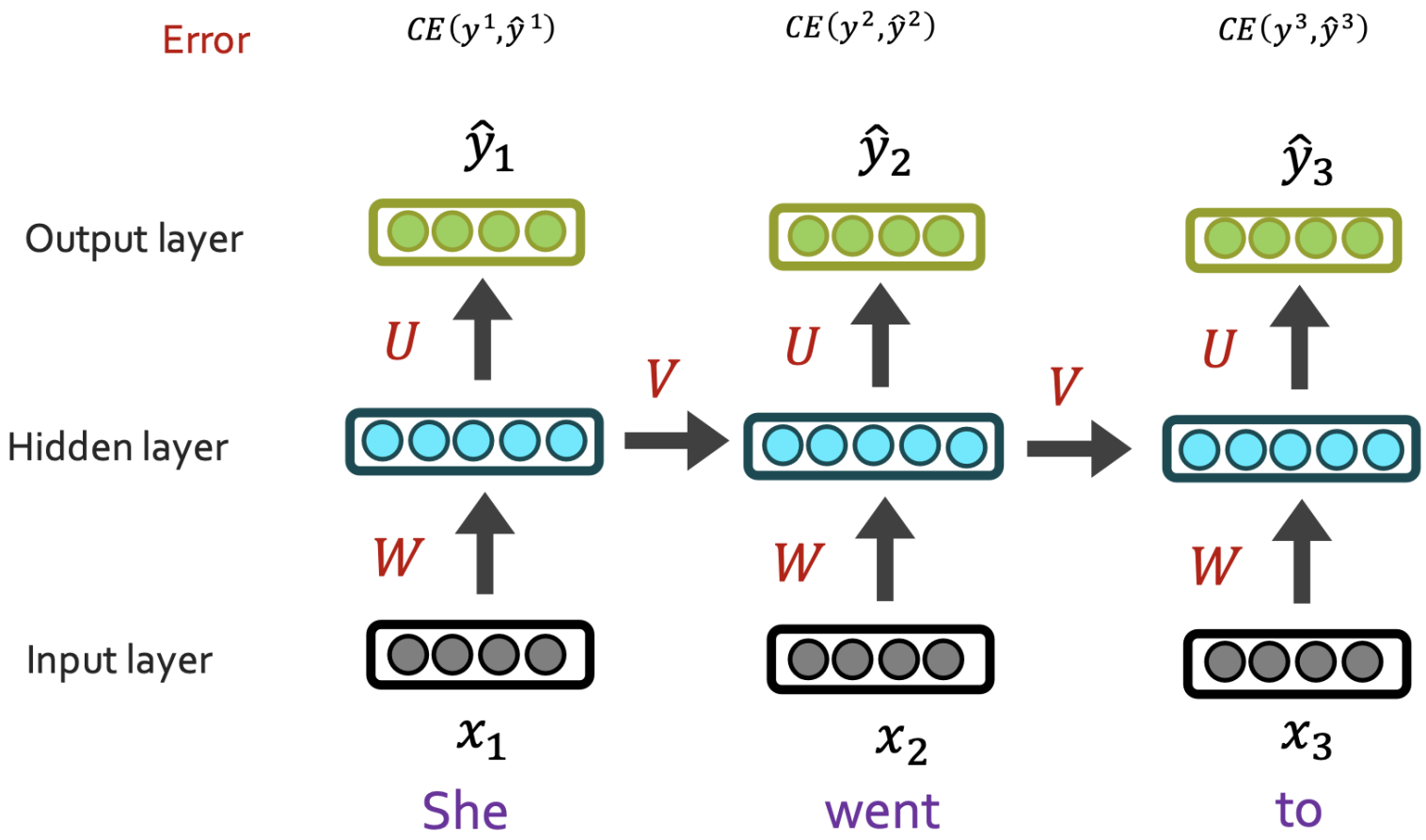
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



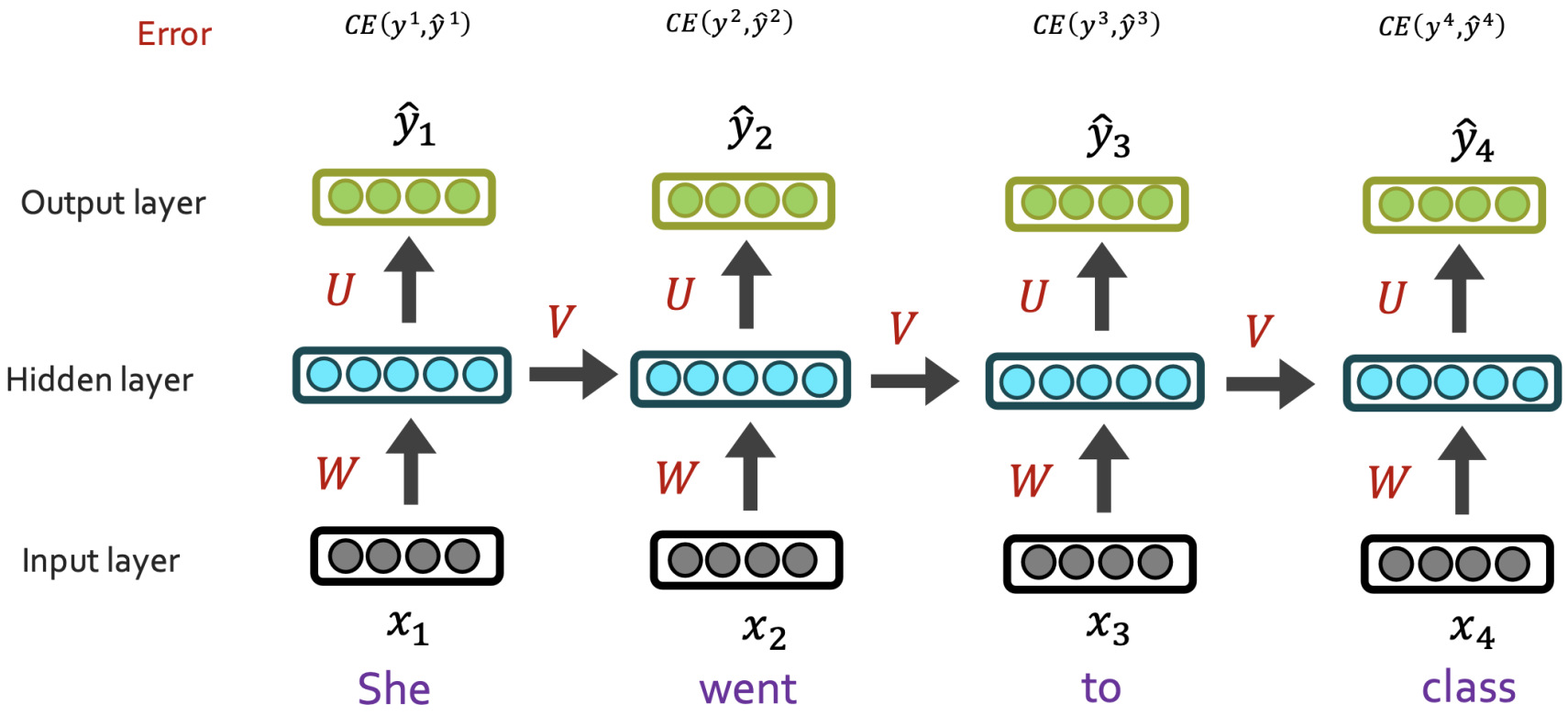
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



RNN: Forward

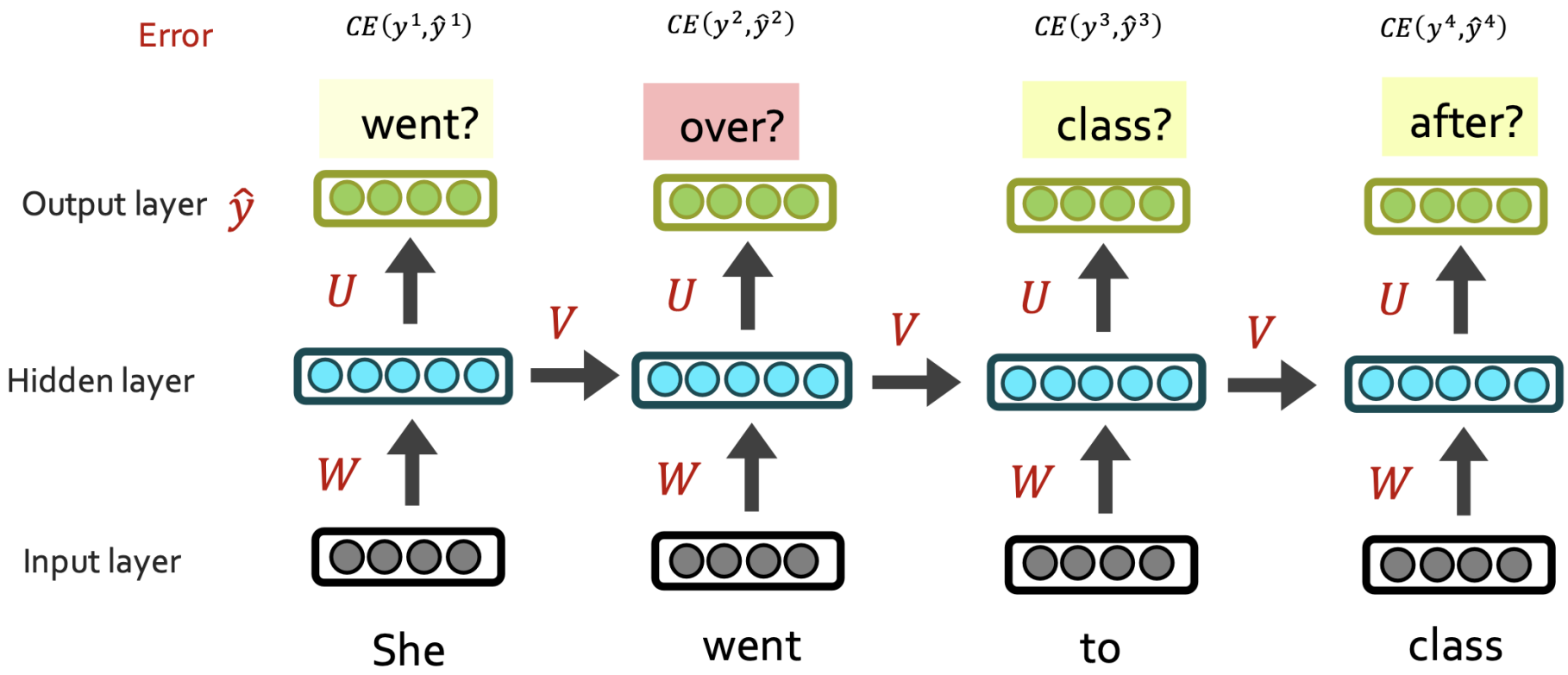
$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$

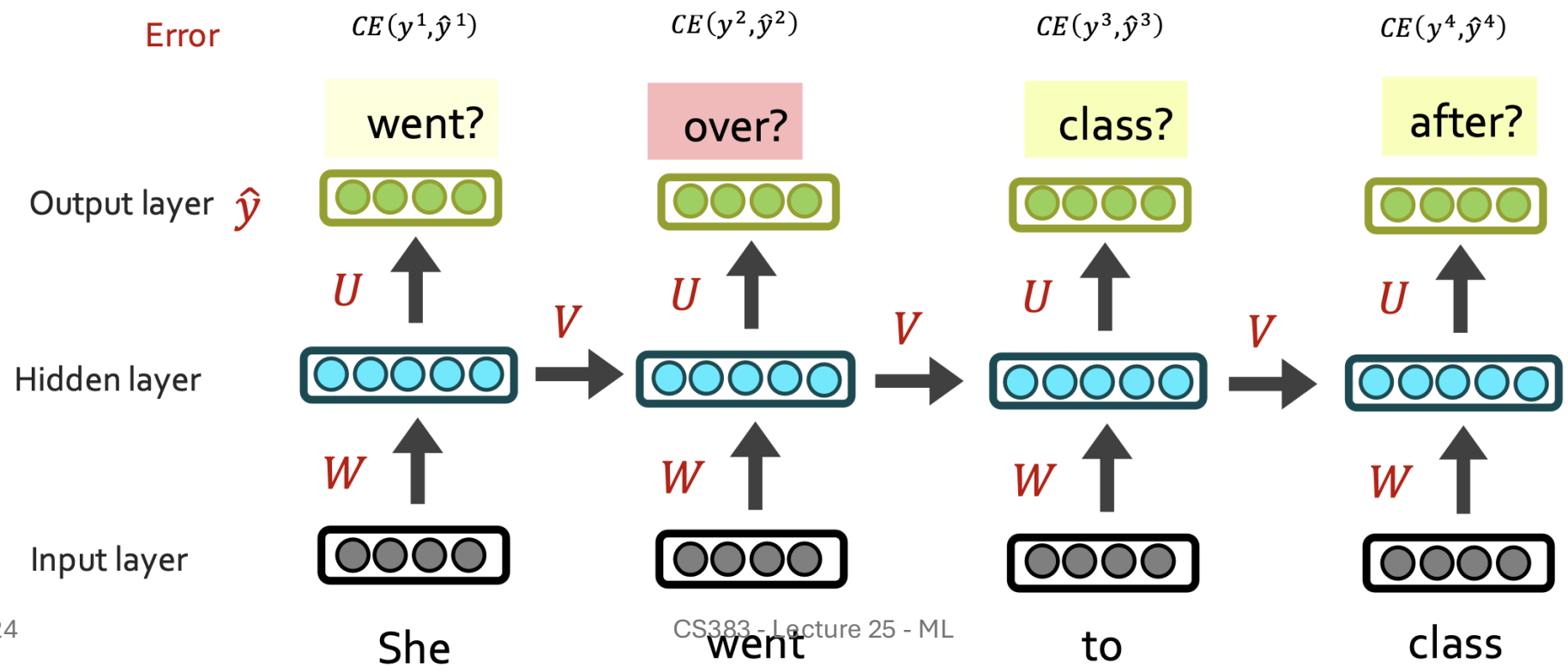
Loss is just averaging Cross-Entropy all predictions



RNN: Backwards

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \hat{y_w}$$

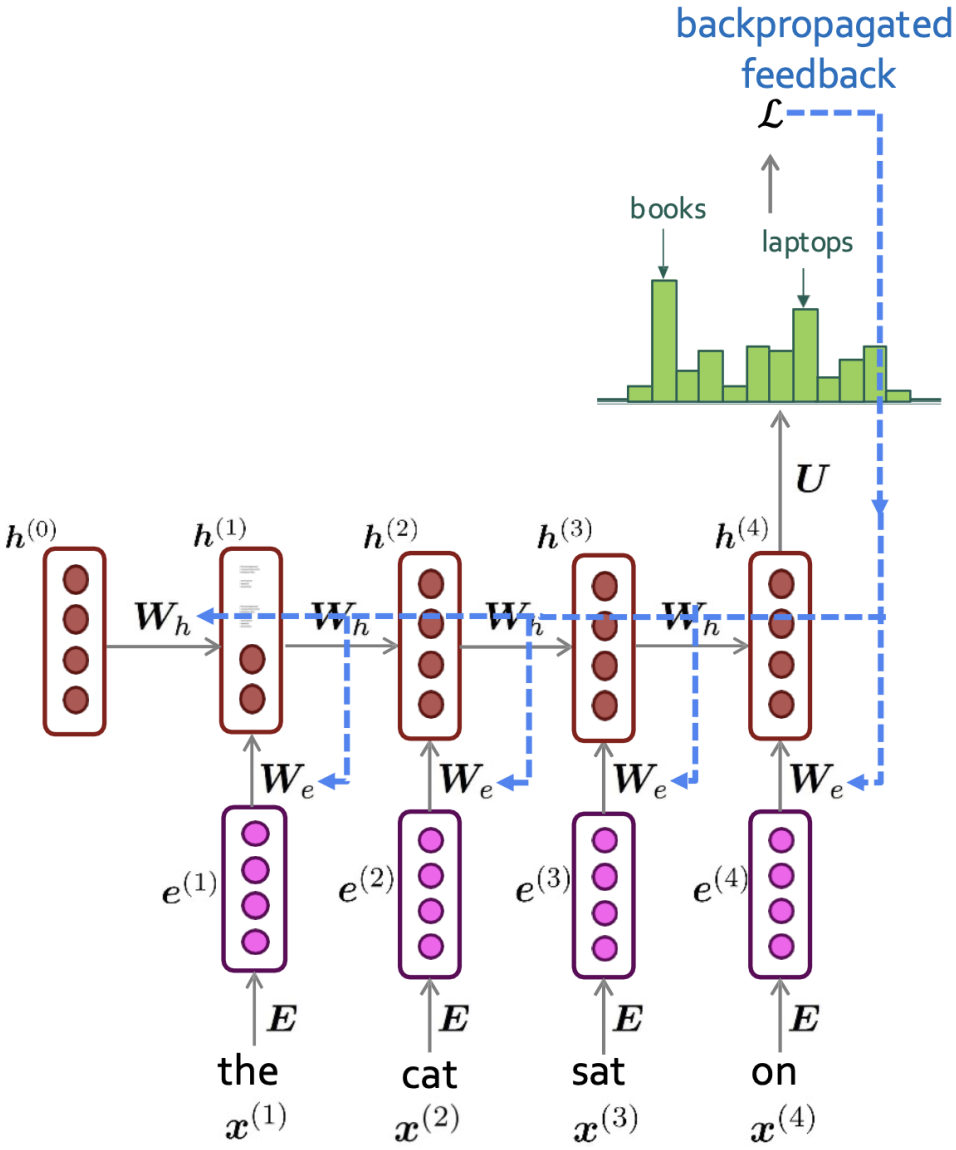
Compute the loss at the end, then propagate derivative of loss back to update the parameters



Training RNNs

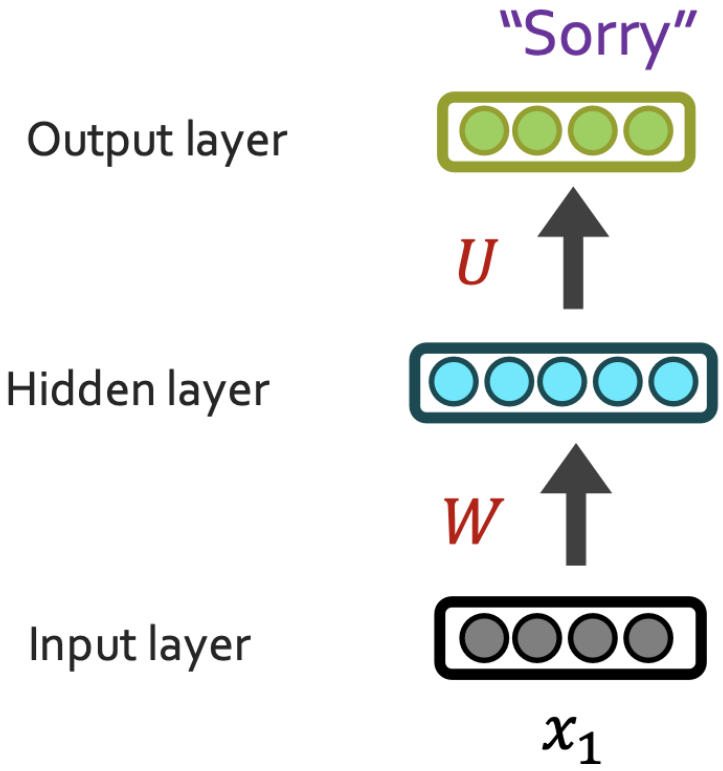
"Backprop over time"

1. Compute $\mathcal{L}$ for a batch of sentences
2. Compute gradients of $\mathcal{L}$ in respect to parameters
3. Repeat



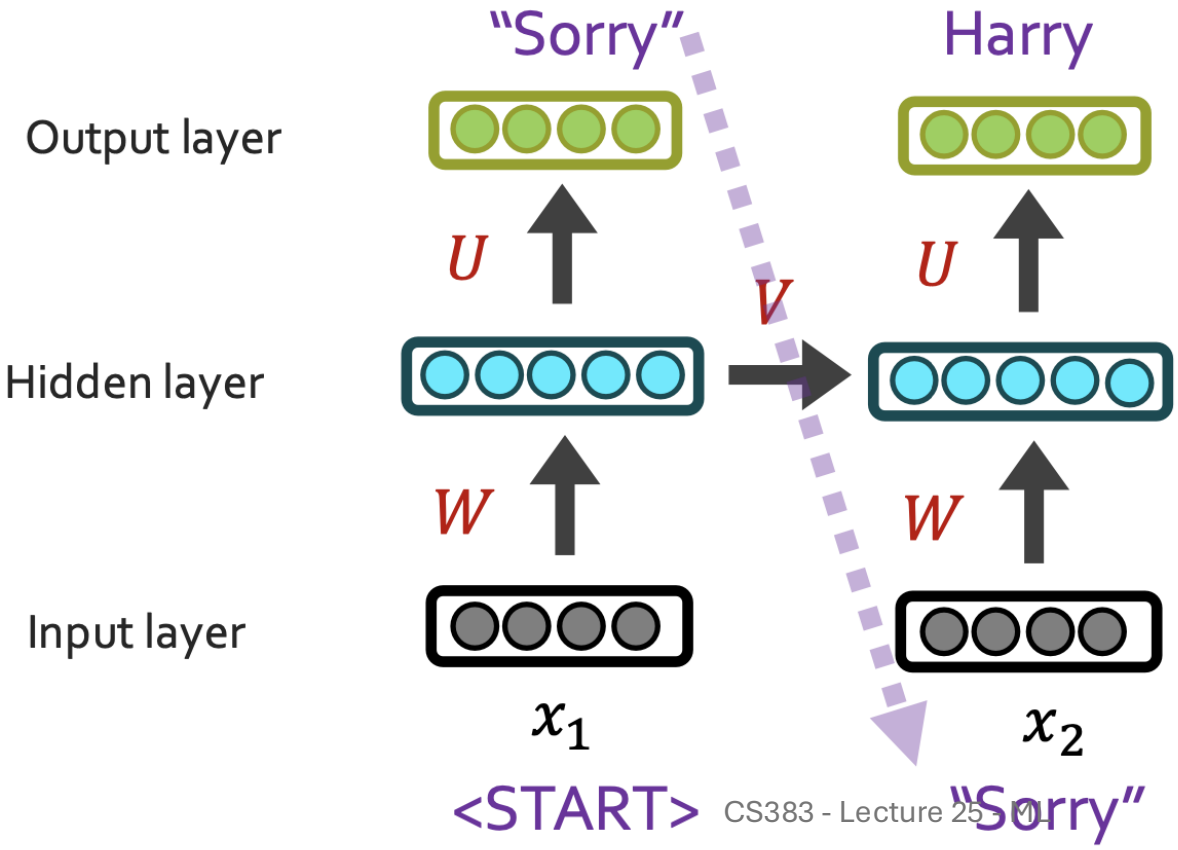
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



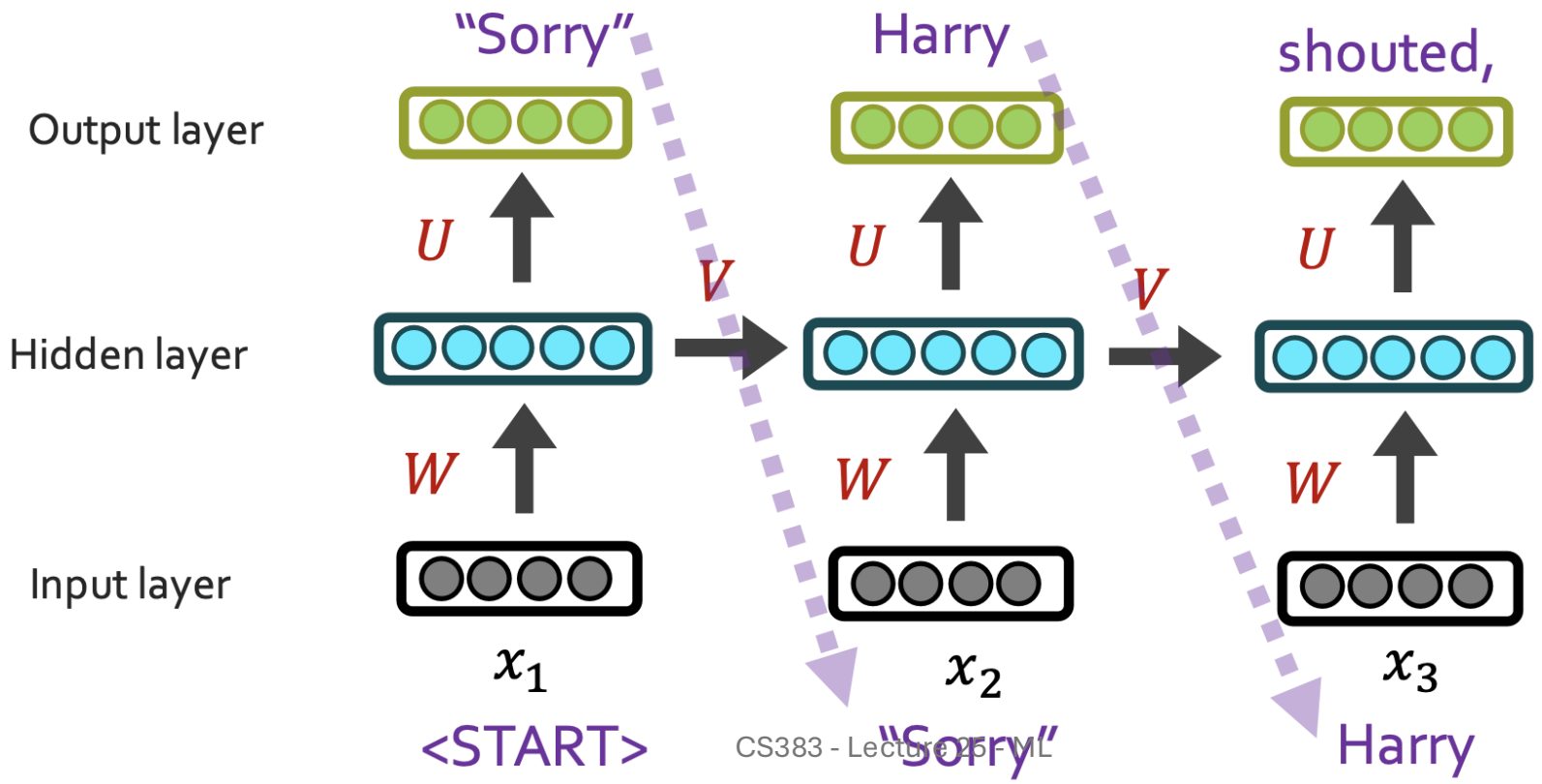
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



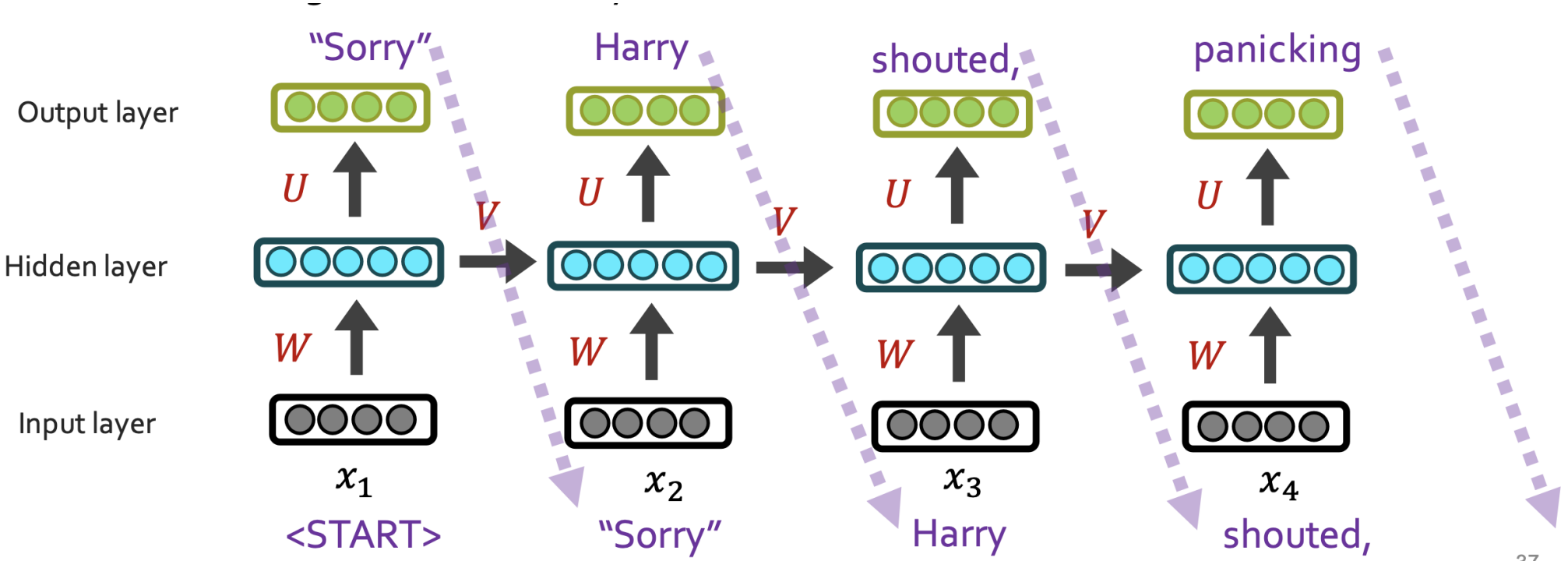
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



RNN's: Pros and Cons

Pros:

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

Cons:

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

Exploding/Vanishing gradient

Backpropagated loss multiplied at each layer

If |loss|> 1,

exponential growth -> ∞

If loss > 0 and <1

exponential decay -> 0

Solution – Gradient Clipping

If the gradient is greater than some threshold, scale it before updating weights

Pascanu et al. 2013
http://proceedings.mlr.press/v28/pascanu13.pdf

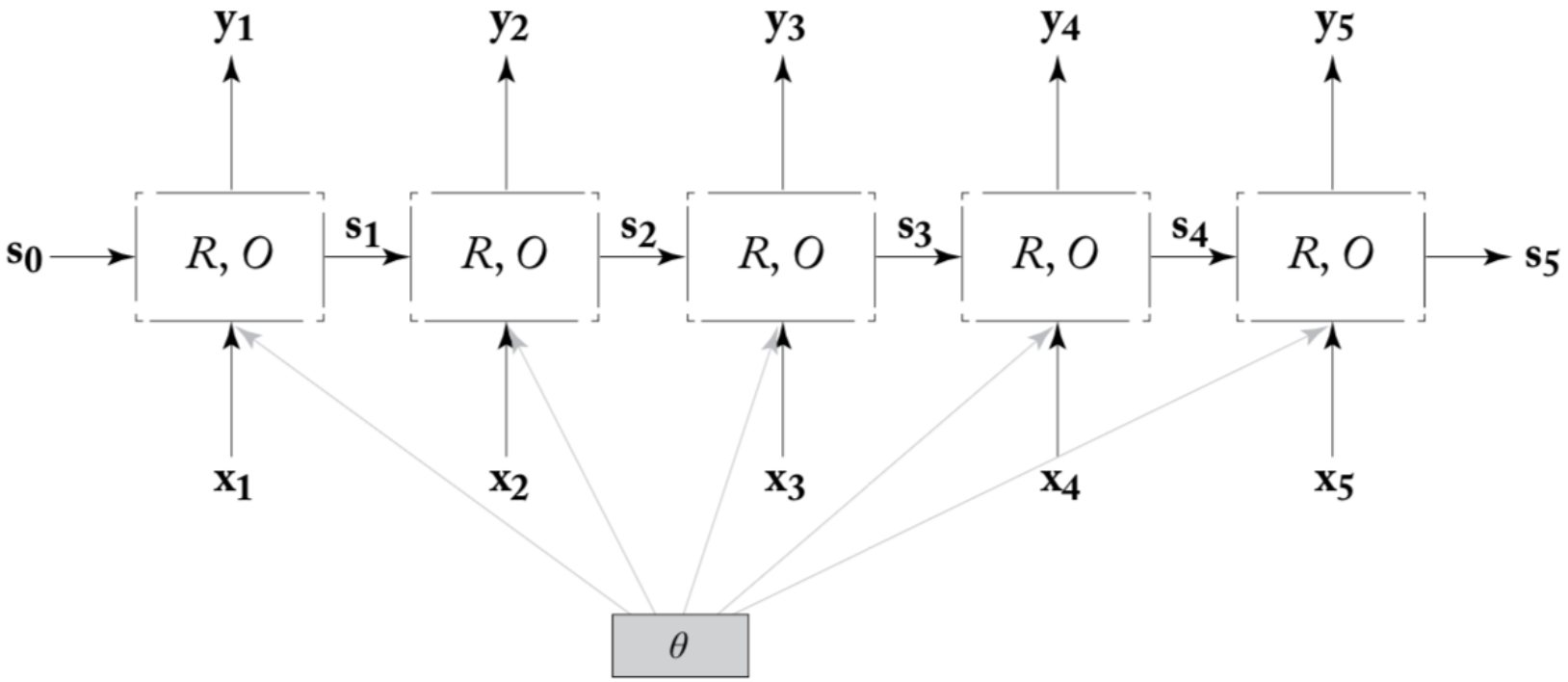
Intuition:

Take a step in the same direction, but smaller

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

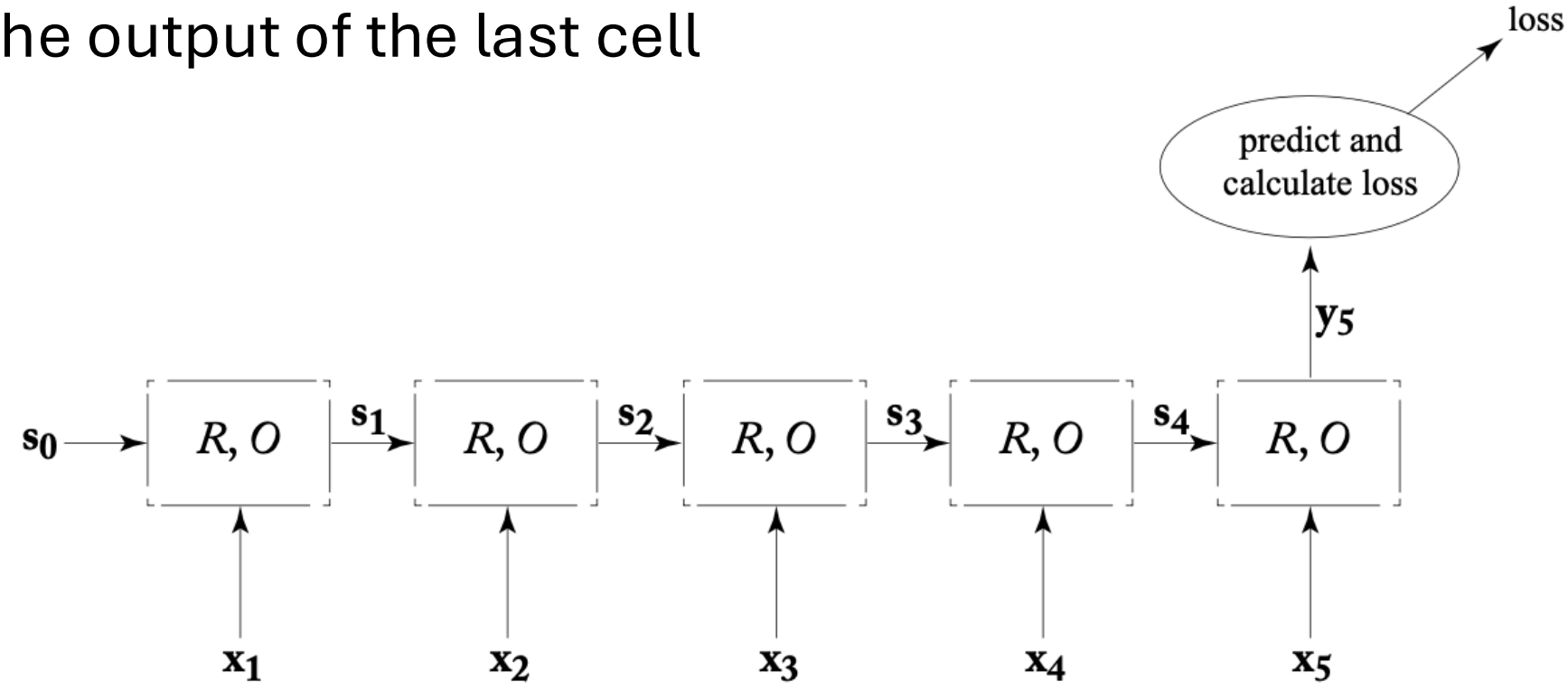
Extracting representation from RNN layer



Extracting representation from RNN layer

Acceptor

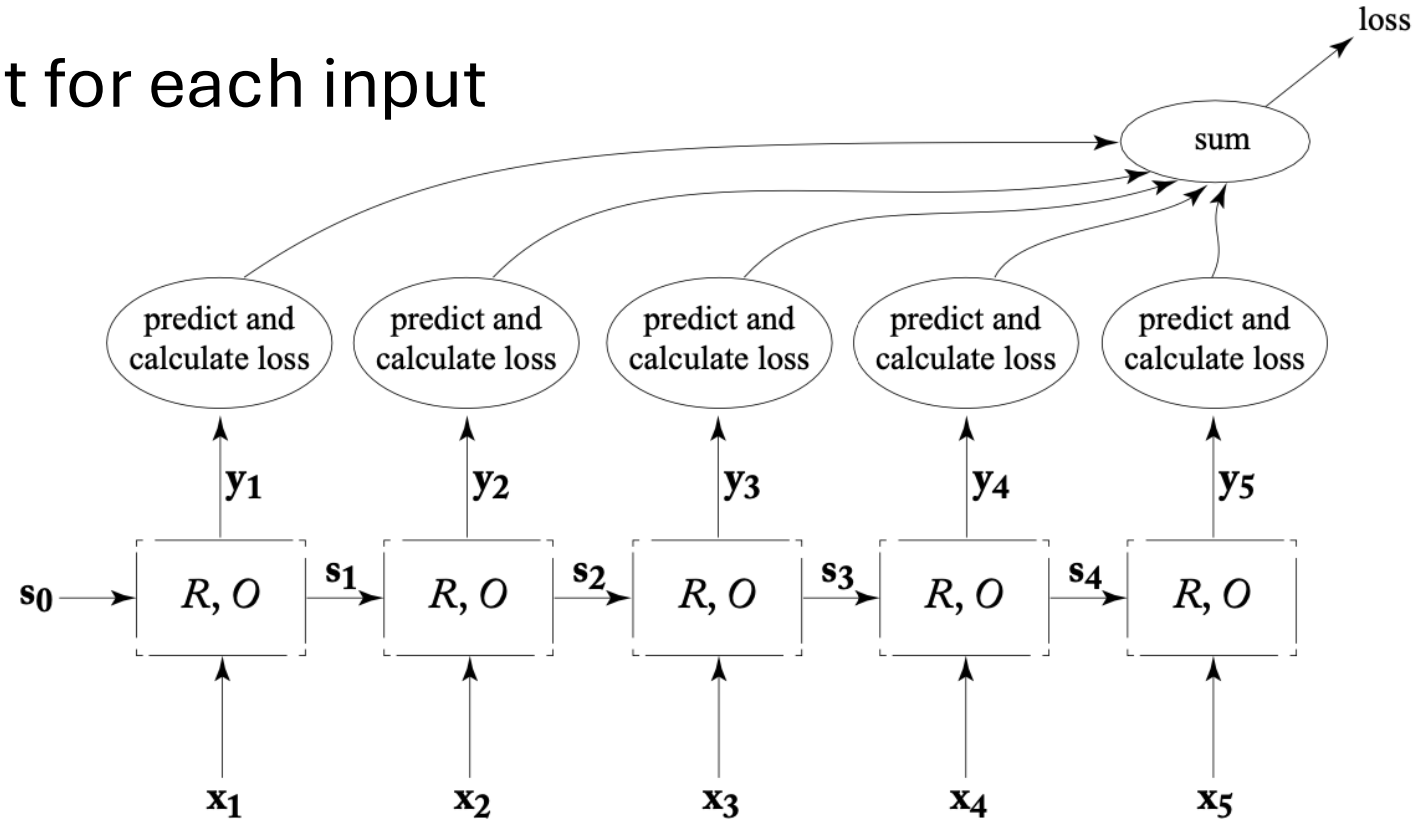
- Take the output of the last cell



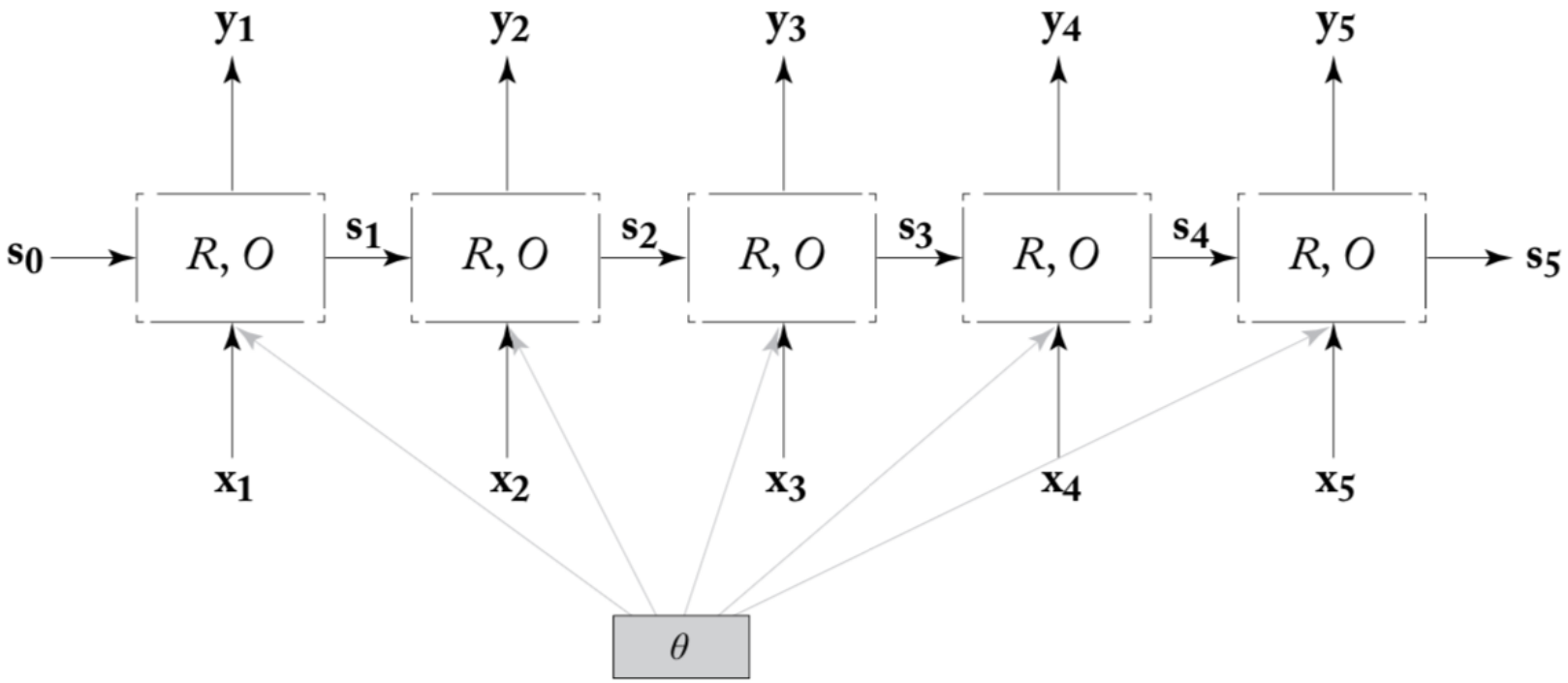
Extracting representation from RNN layer

Transducer

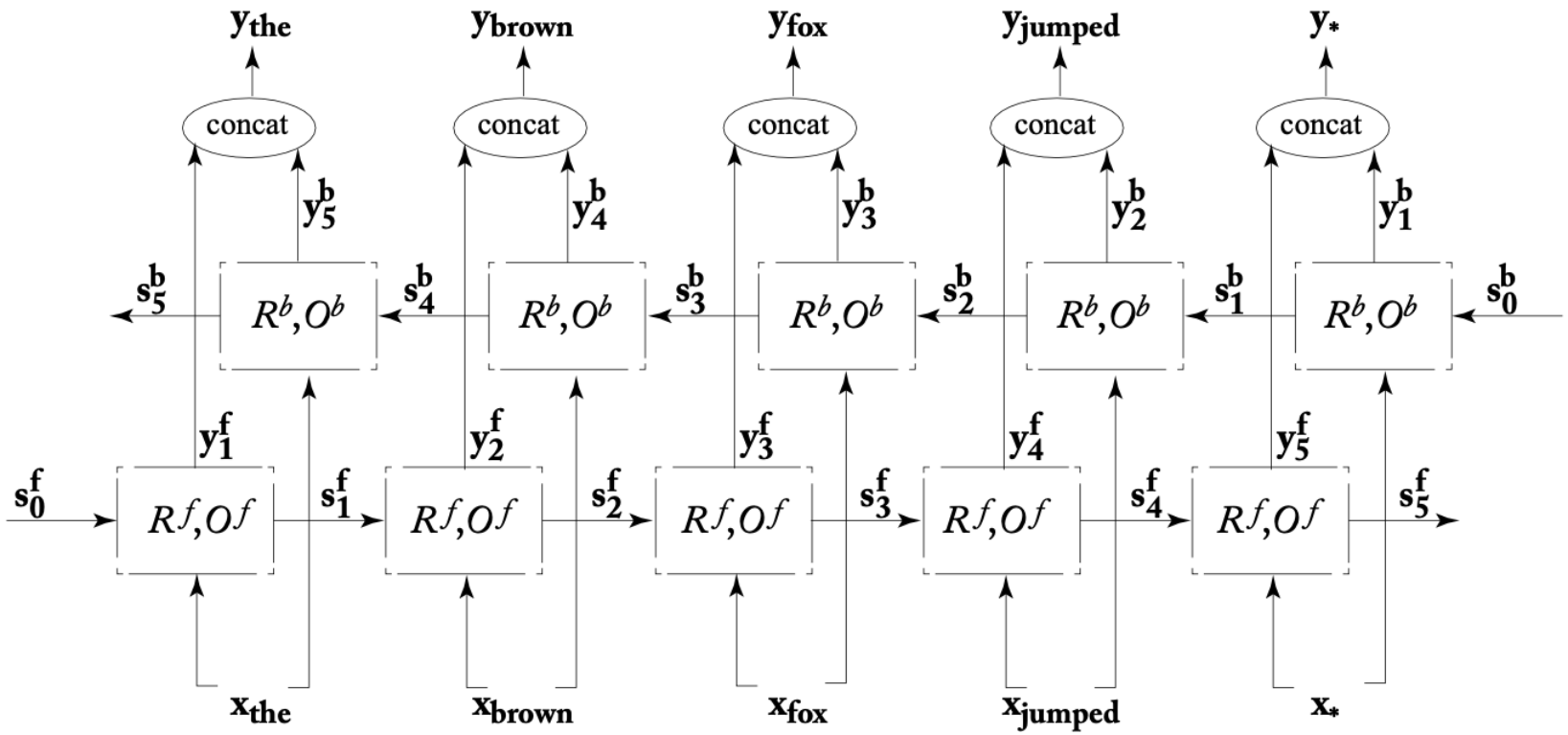
- Create an output for each input



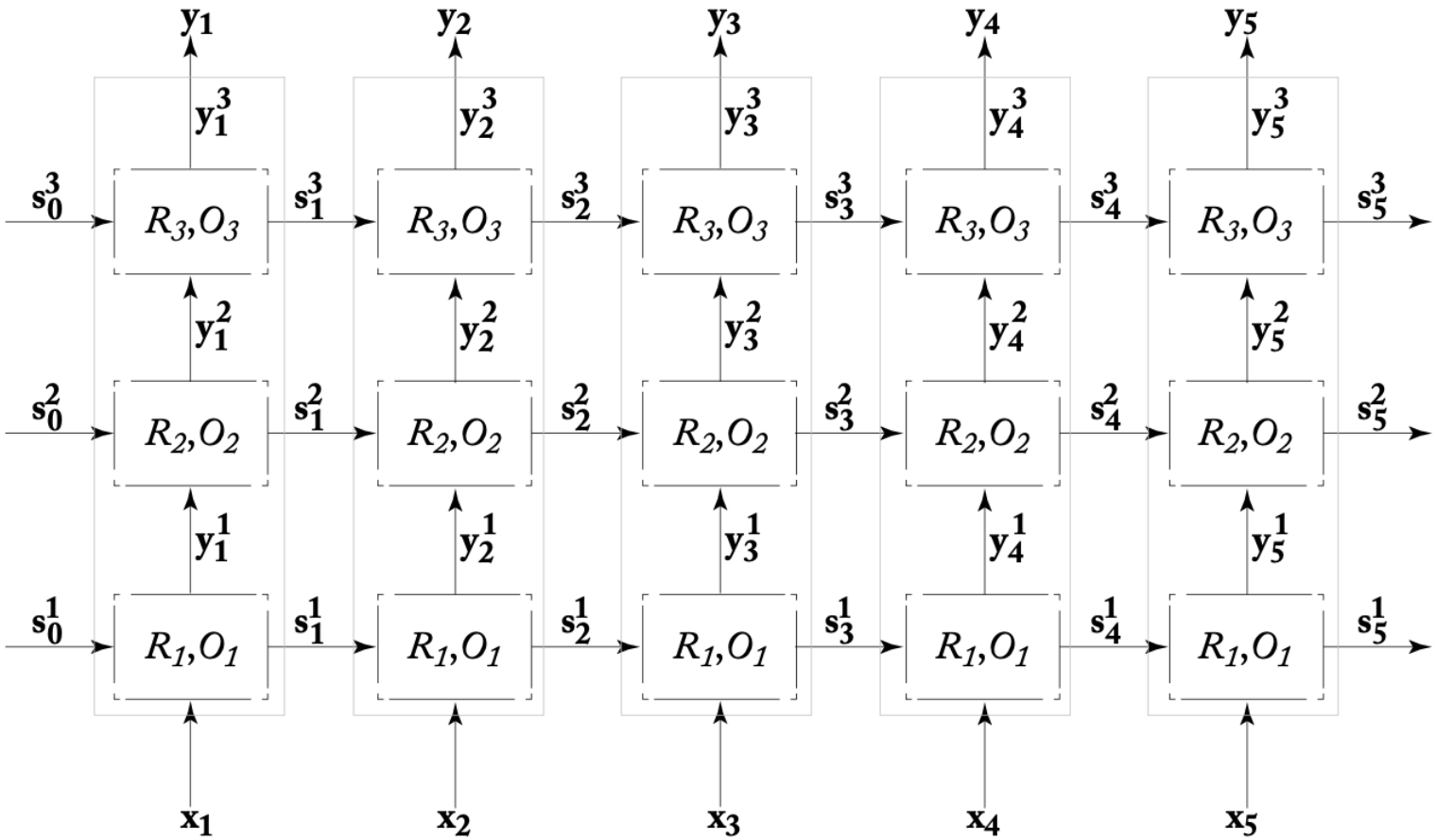
How else can we expand this?



Bi-directional



Stack more layers



Pytorch - nn.RNN

Parameters:

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

RNNs – long input

RNNs can remember anything (in theory)

Sometimes its important to forget

Solution: Long-Short Term Memory (LSTM)

Outline

Issues when training NNs

Pytorch

FFN’s for variable length input

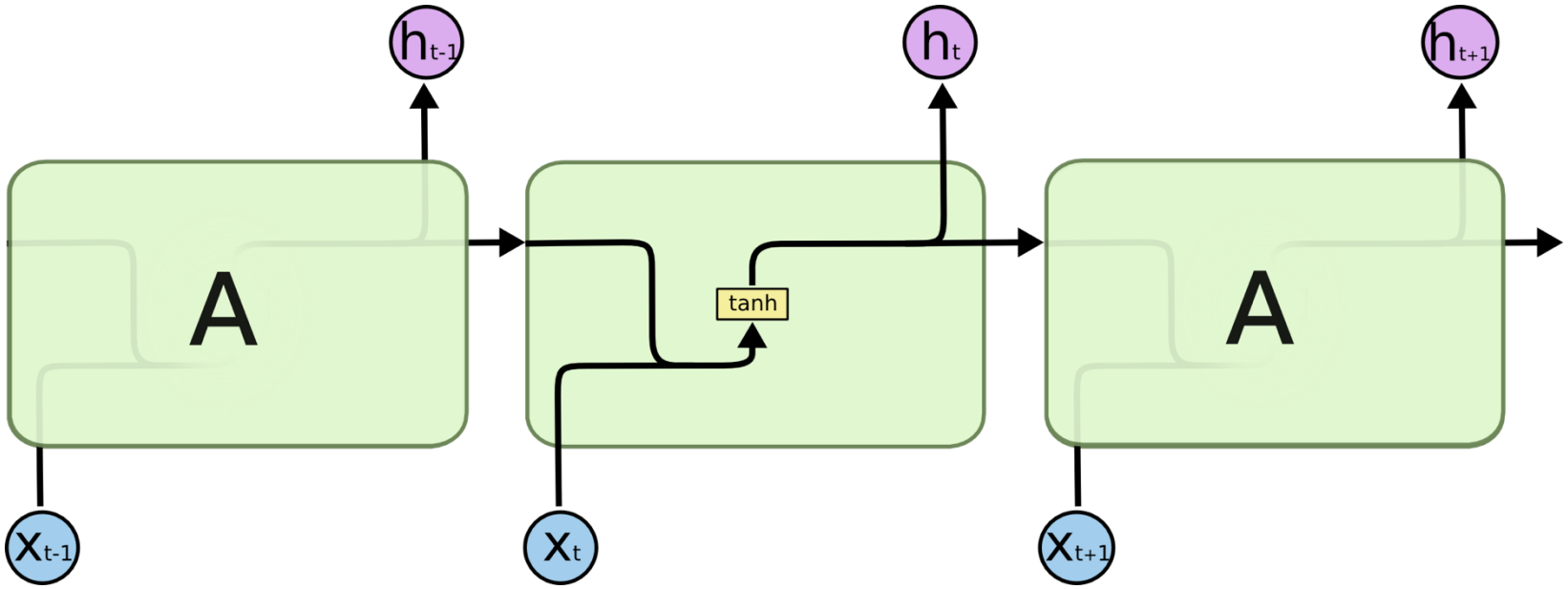
RNN

LSTM

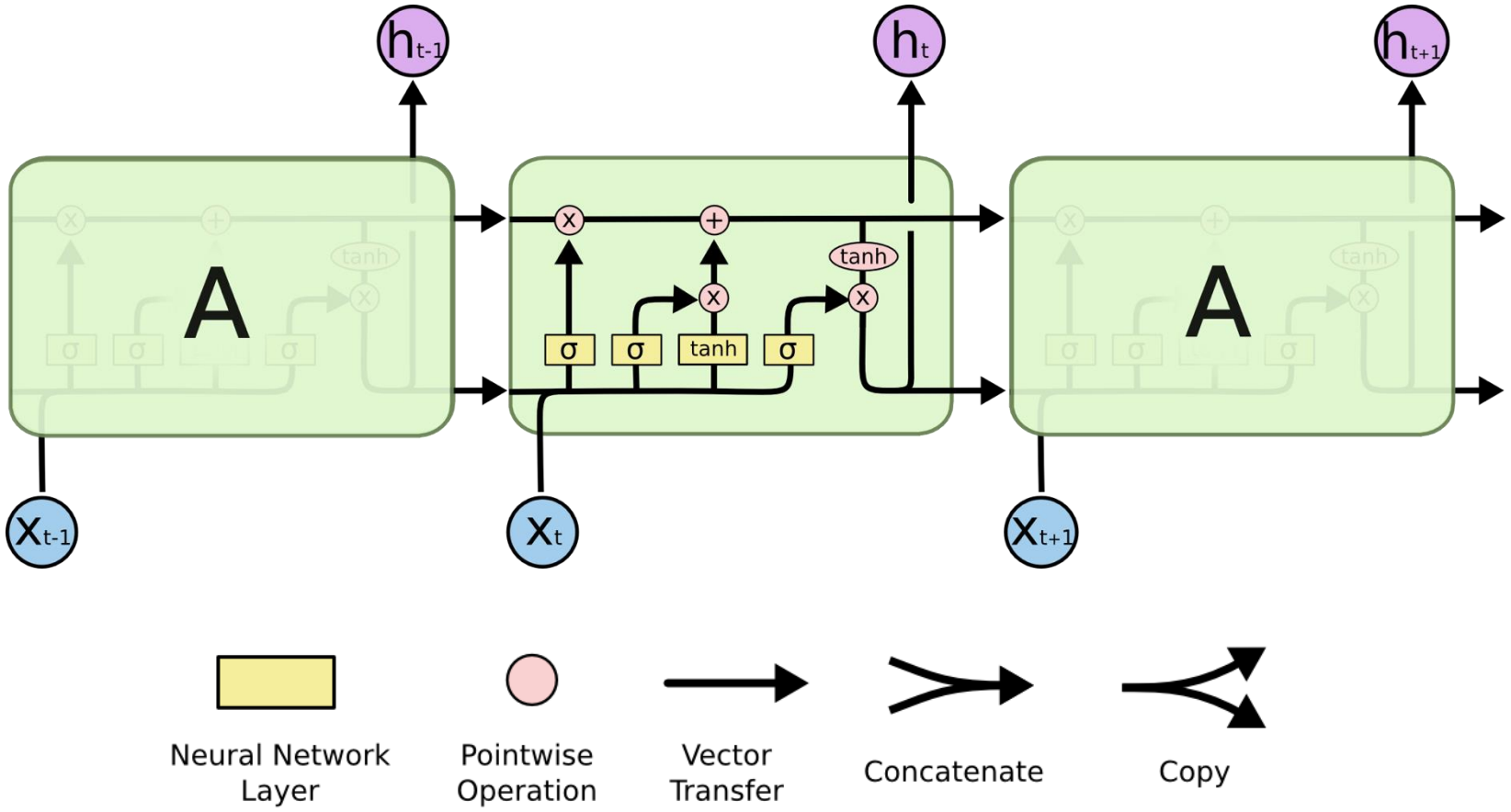
Attention

Self Attention

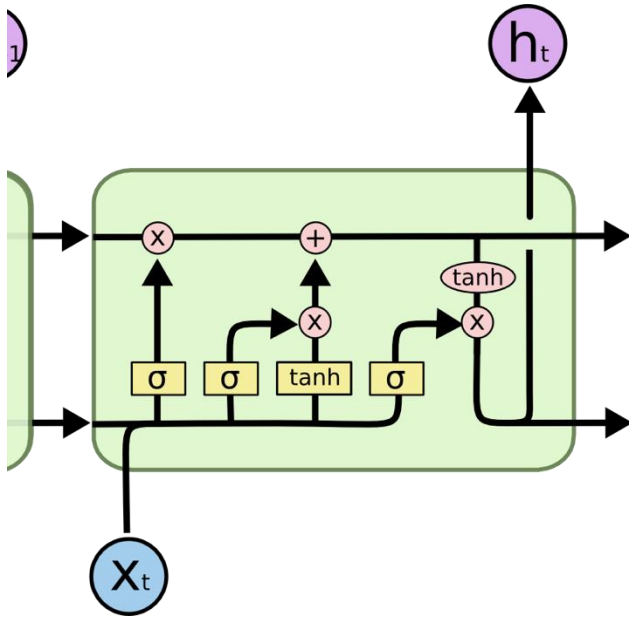
RNN internal



LSTM internal



LSTM internal



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

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

$$h_j = o \odot \tanh(c_j)$$

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

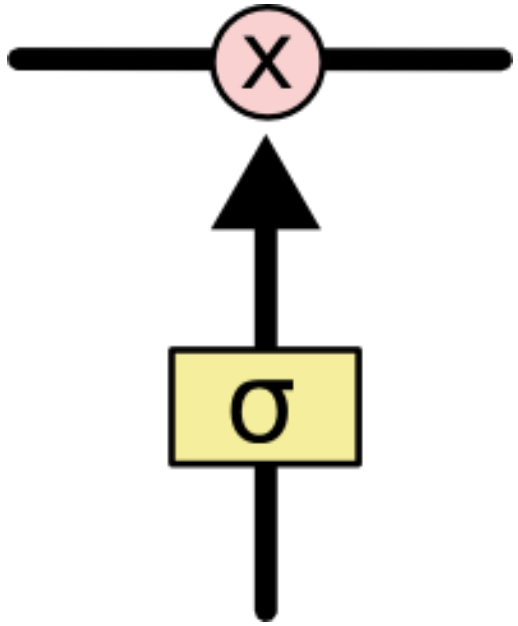
$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

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

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

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

LSTM's rely on gates

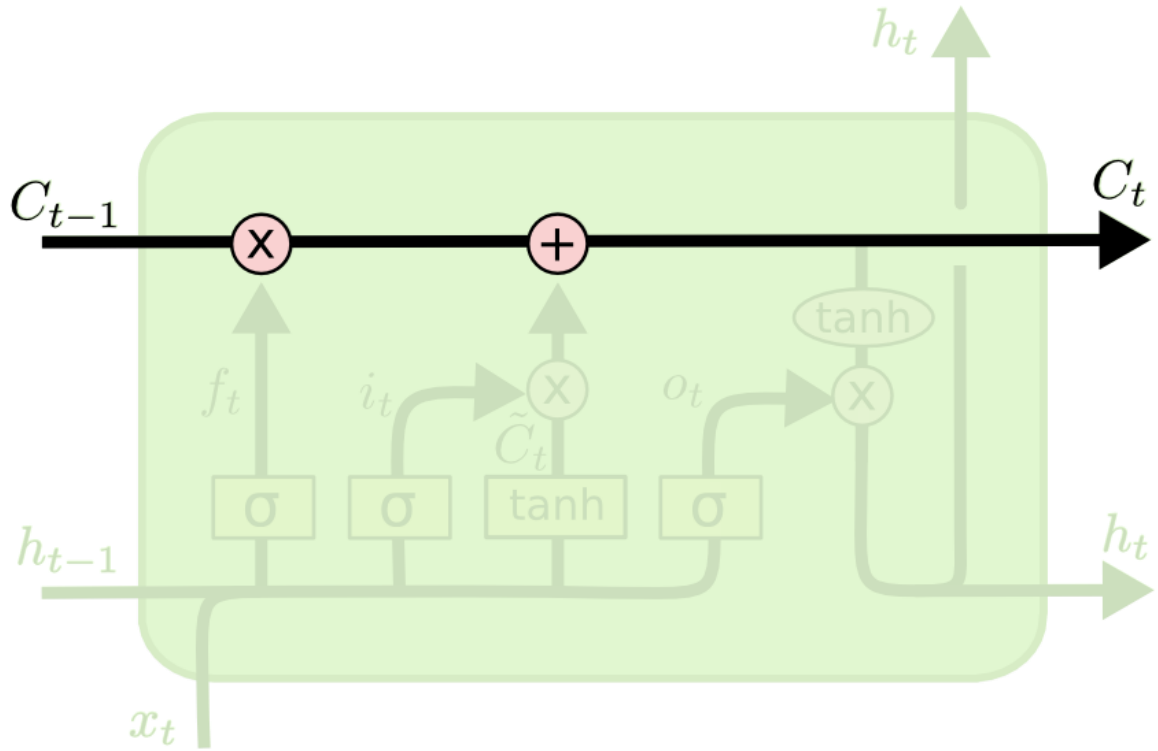


- Multiply input by value in 0,1]
- Zero means forget everything
- 1 means carry everything through (unchanged)

- 4 gates used in LSTM

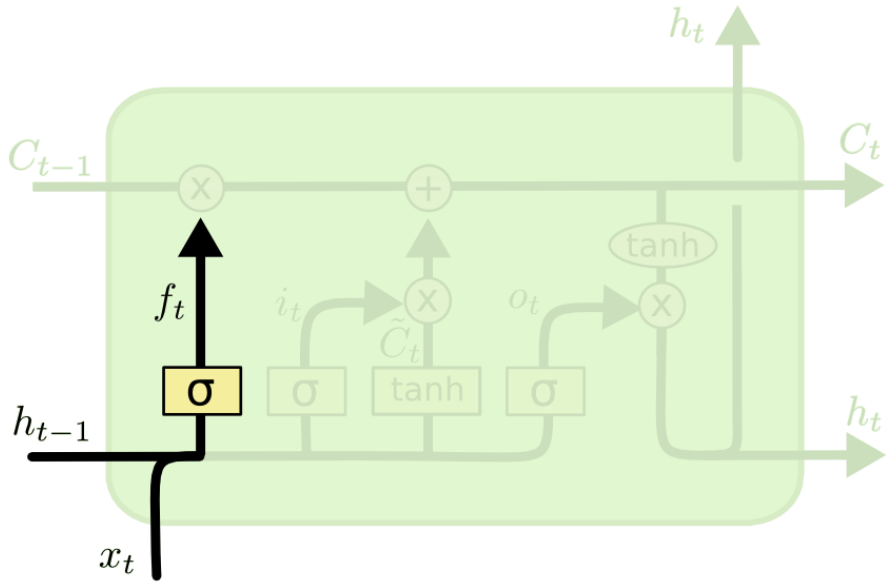
LSTM gates: cell state

- Passes the memory through the cell



LSTM gates: forget

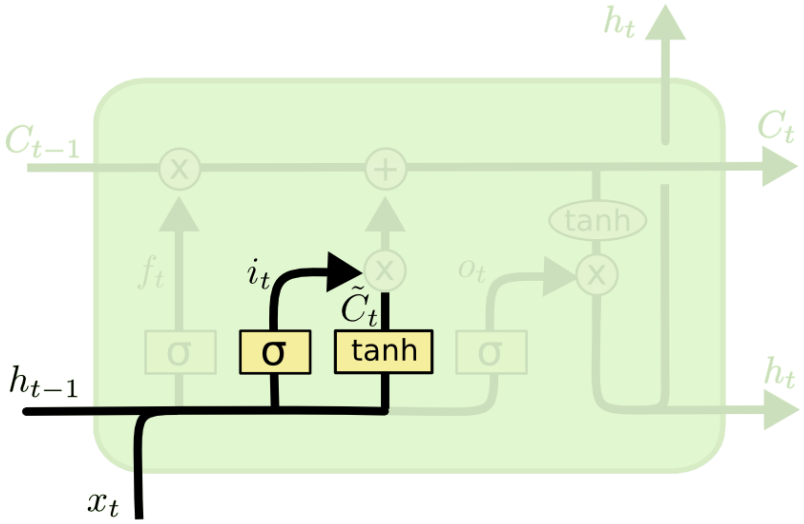
- Can decide to forget the previous state h_{t-1}



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

LSTM gates: update

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

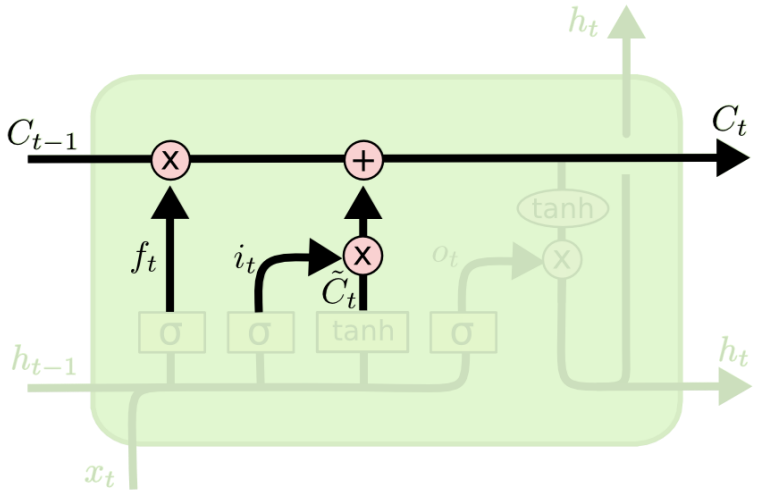


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

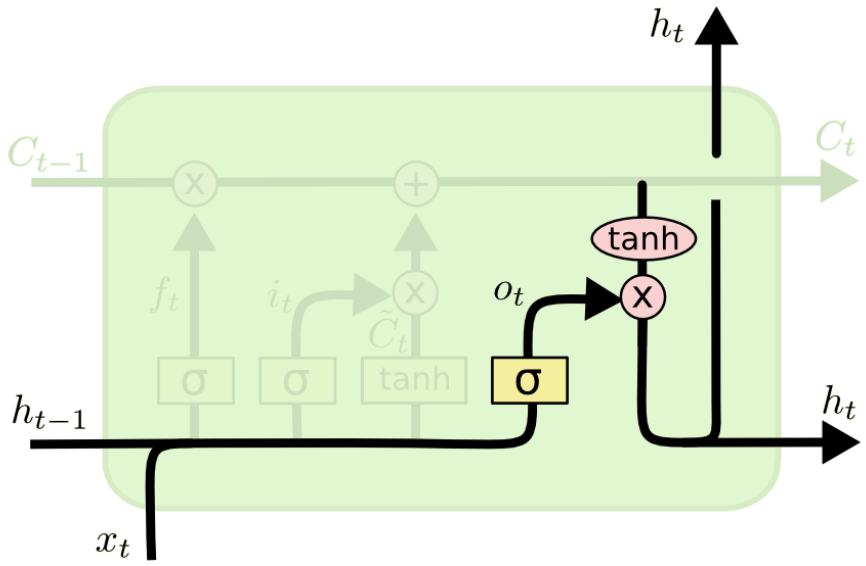
LSTM gates: update (interpolate)

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



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

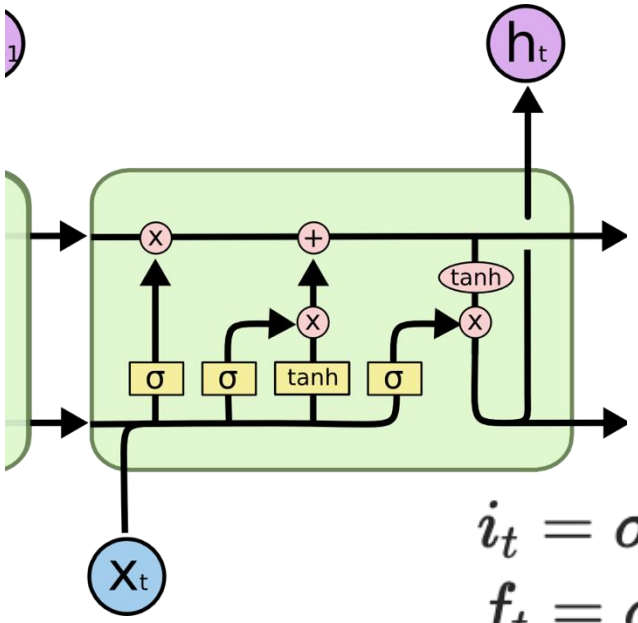
LSTM output (hidden)



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

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

LSTM internal



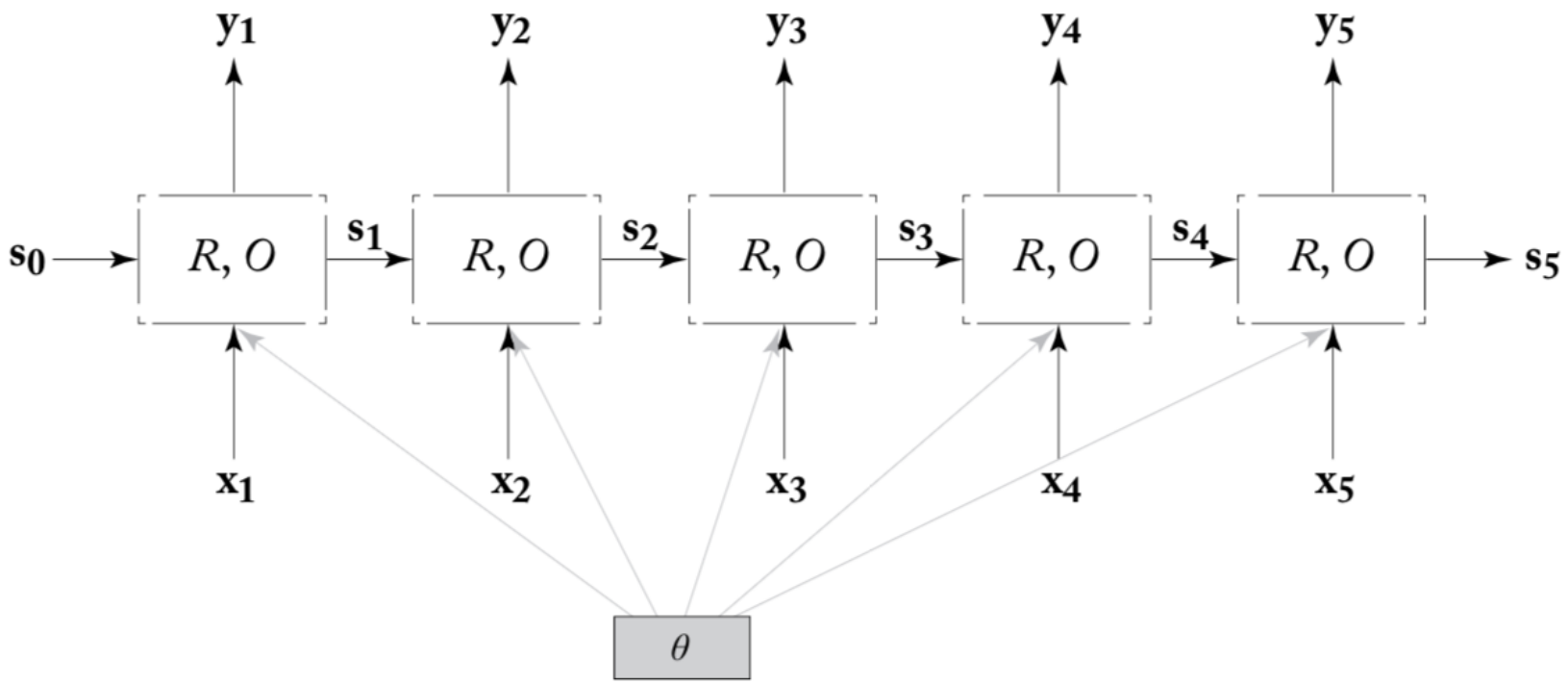
$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot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

Pytorch - nn.LSTM

Parameters:

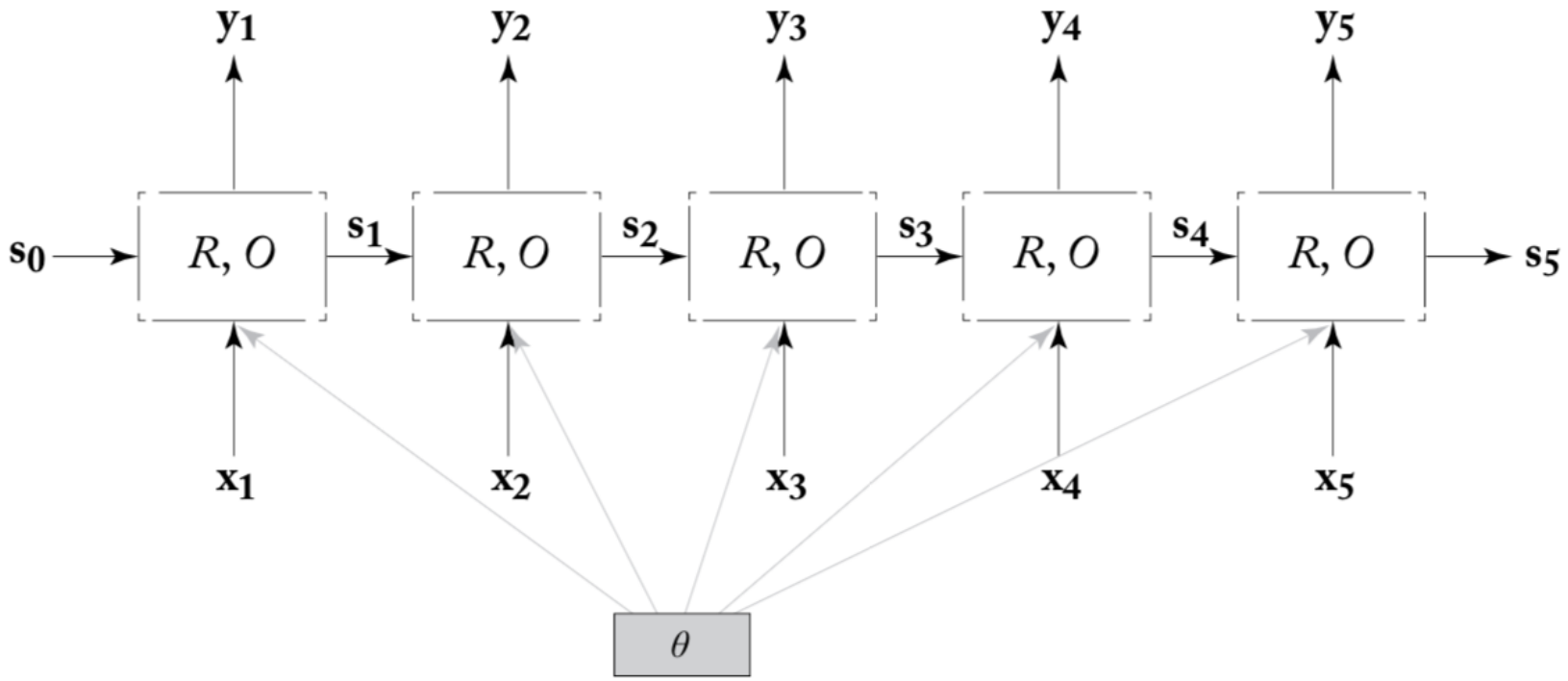
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

Extracting representation from RNN layer



Extracting representation from RNN layer

How might we combine the output layers



Outline

Issues when training NNs

Pytorch

FFN’s for variable length input

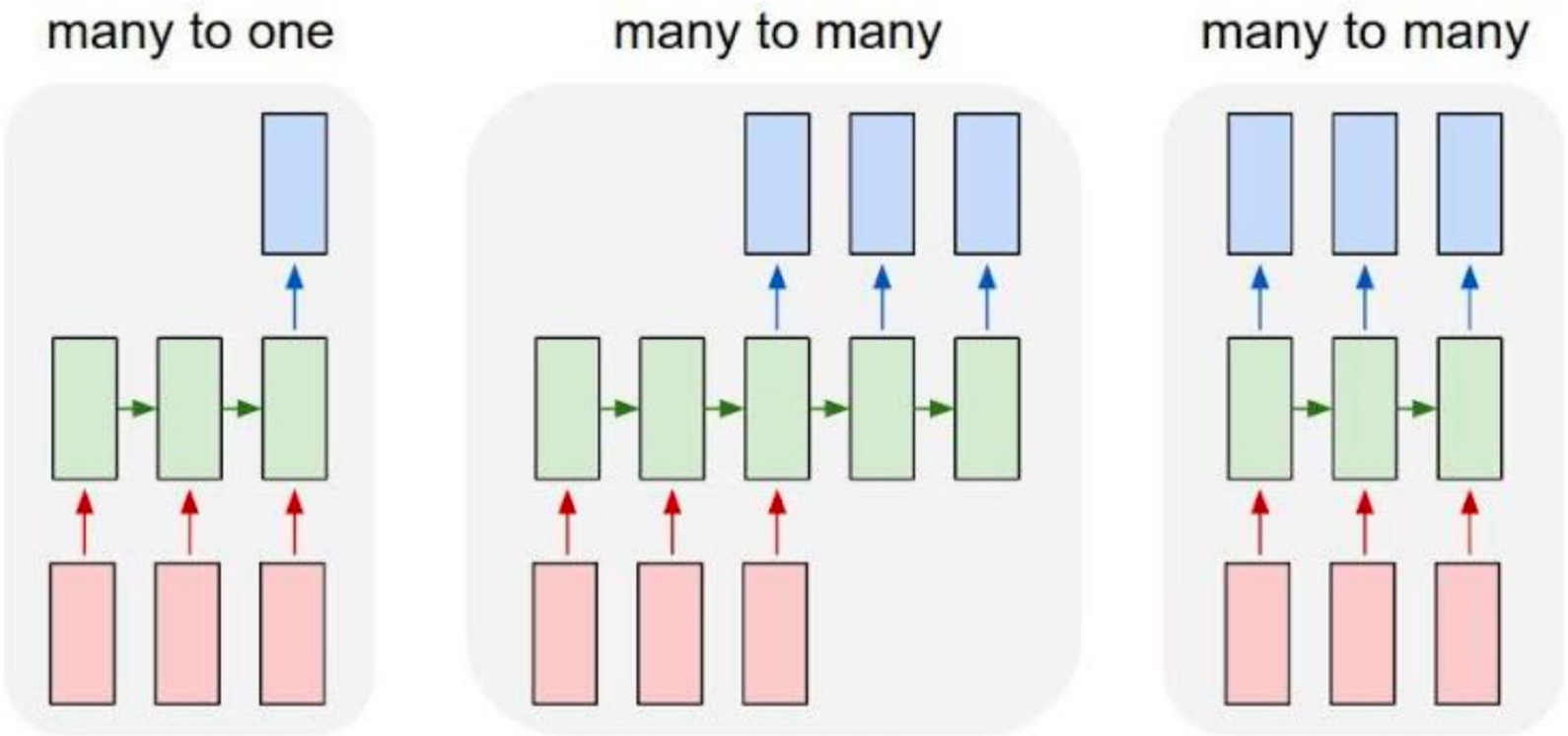
RNN

LSTM

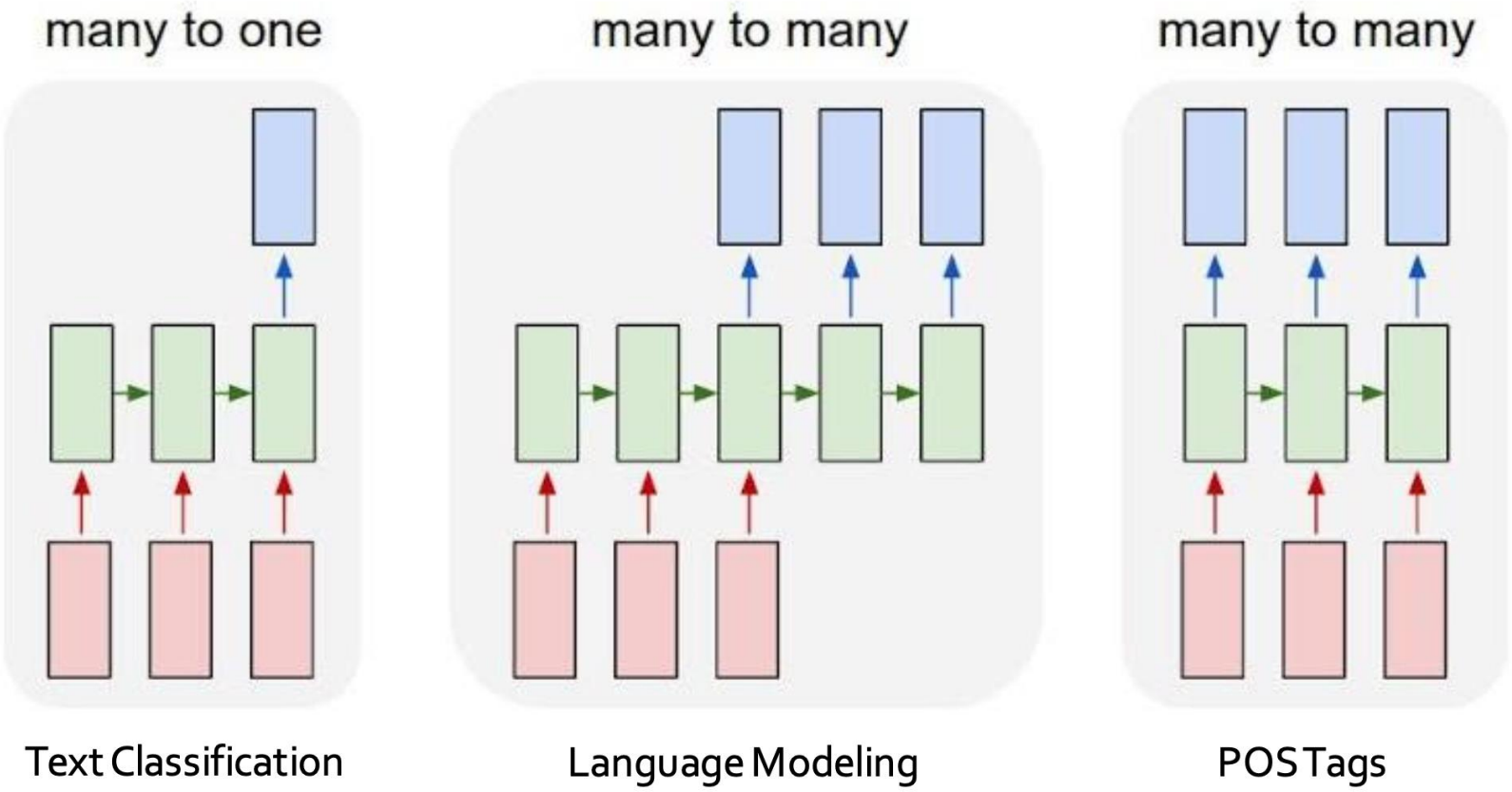
Attention

Self Attention

RNNs applied to NLP tasks

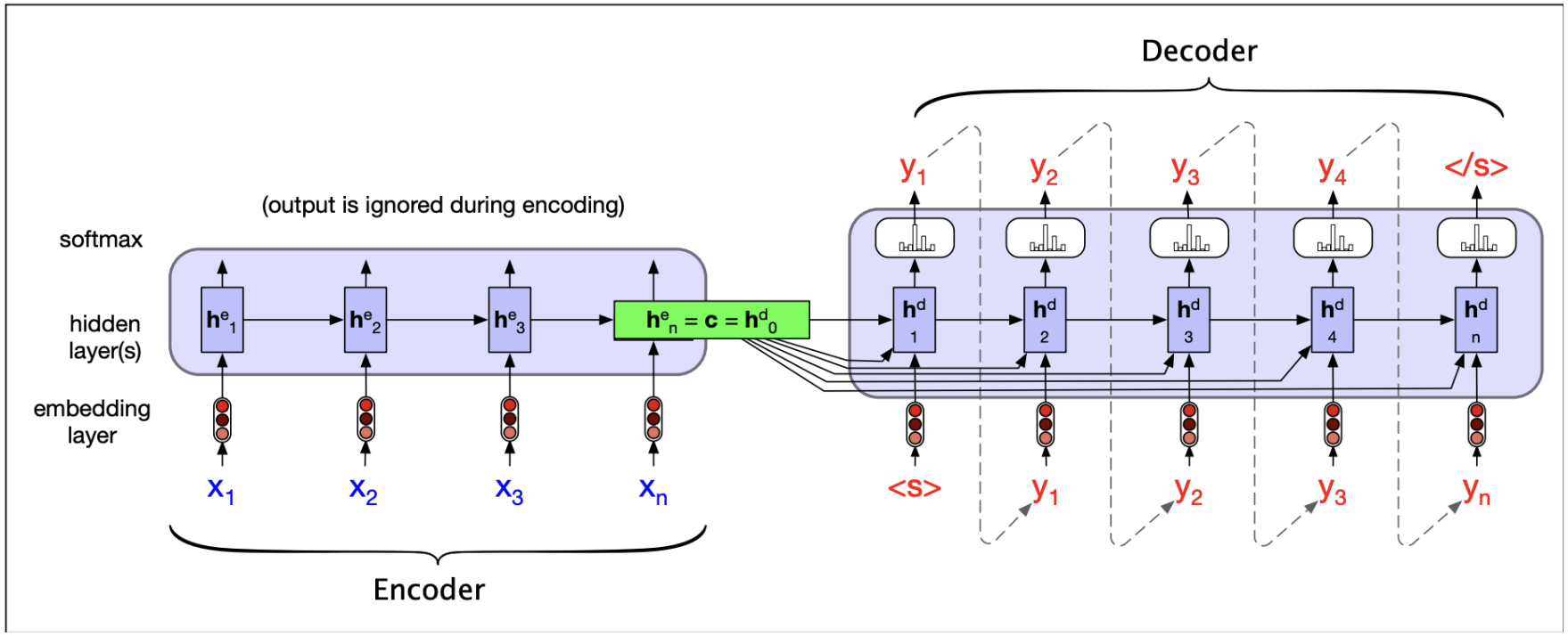


RNNs applied to NLP tasks



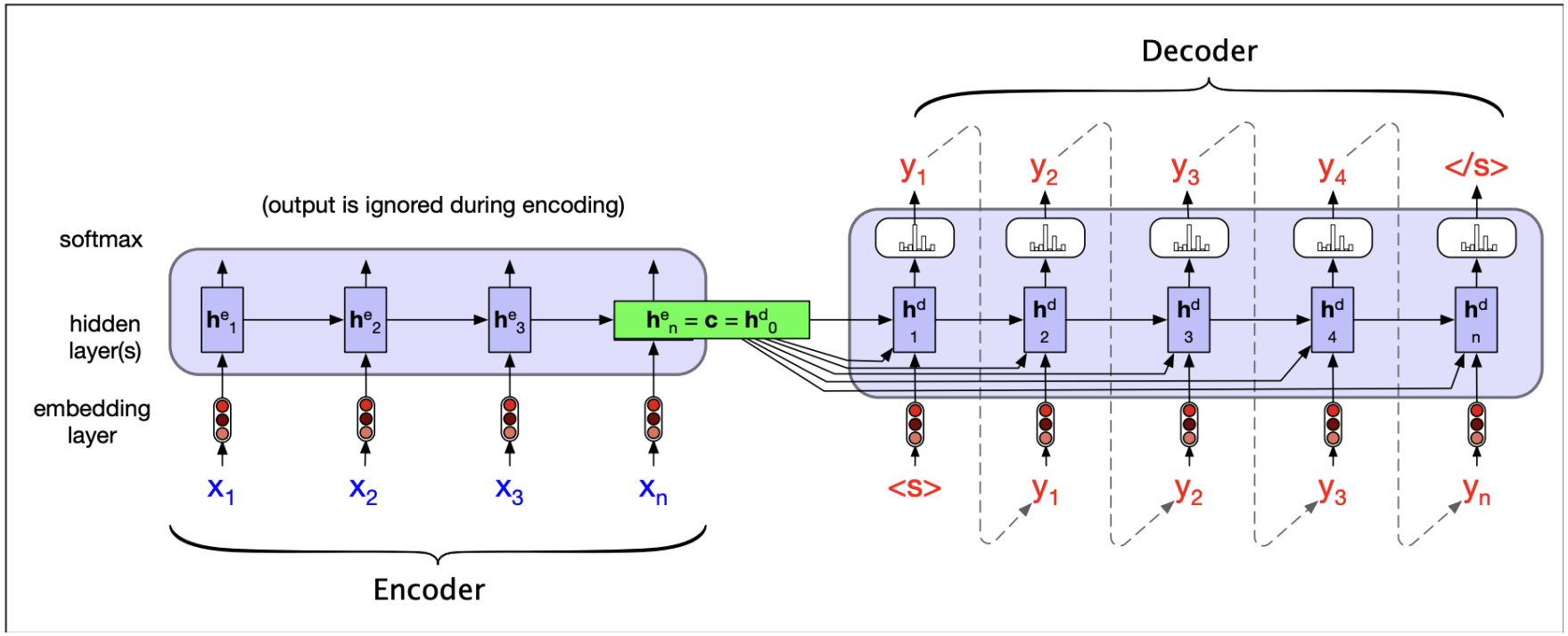
Encoder-decoder

Decoder only uses information from last hidden cell!



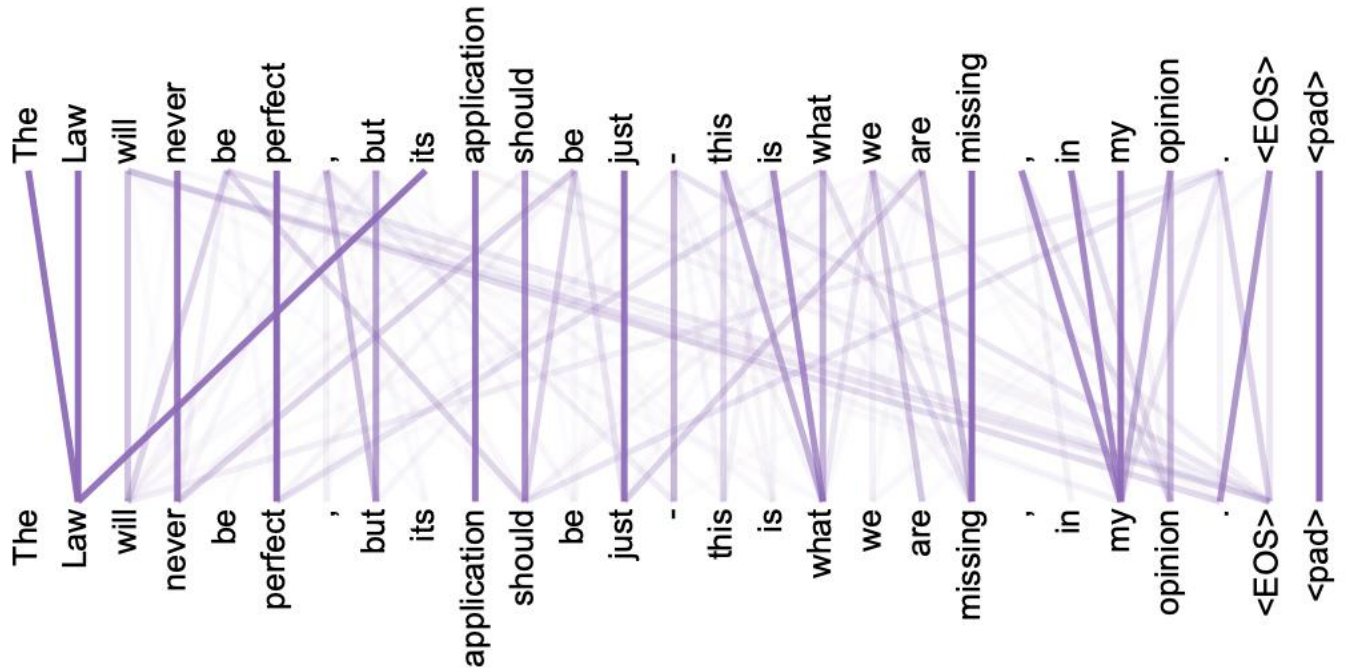
Bottleneck

Last hidden cell is a bottleneck



Solution: Attention!

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

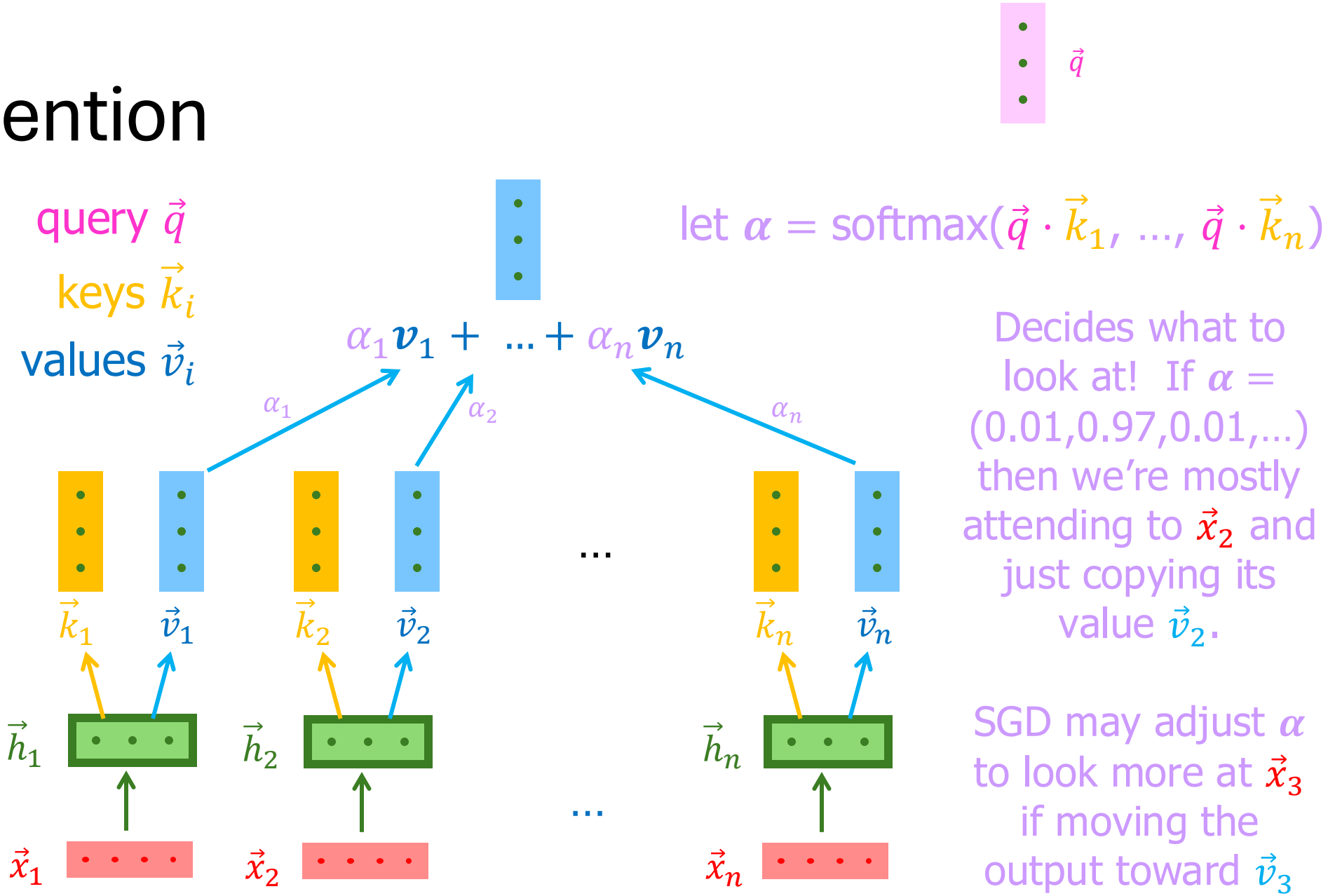


Attention is all you need Vaswani et al 2017

Have we been cramming?

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

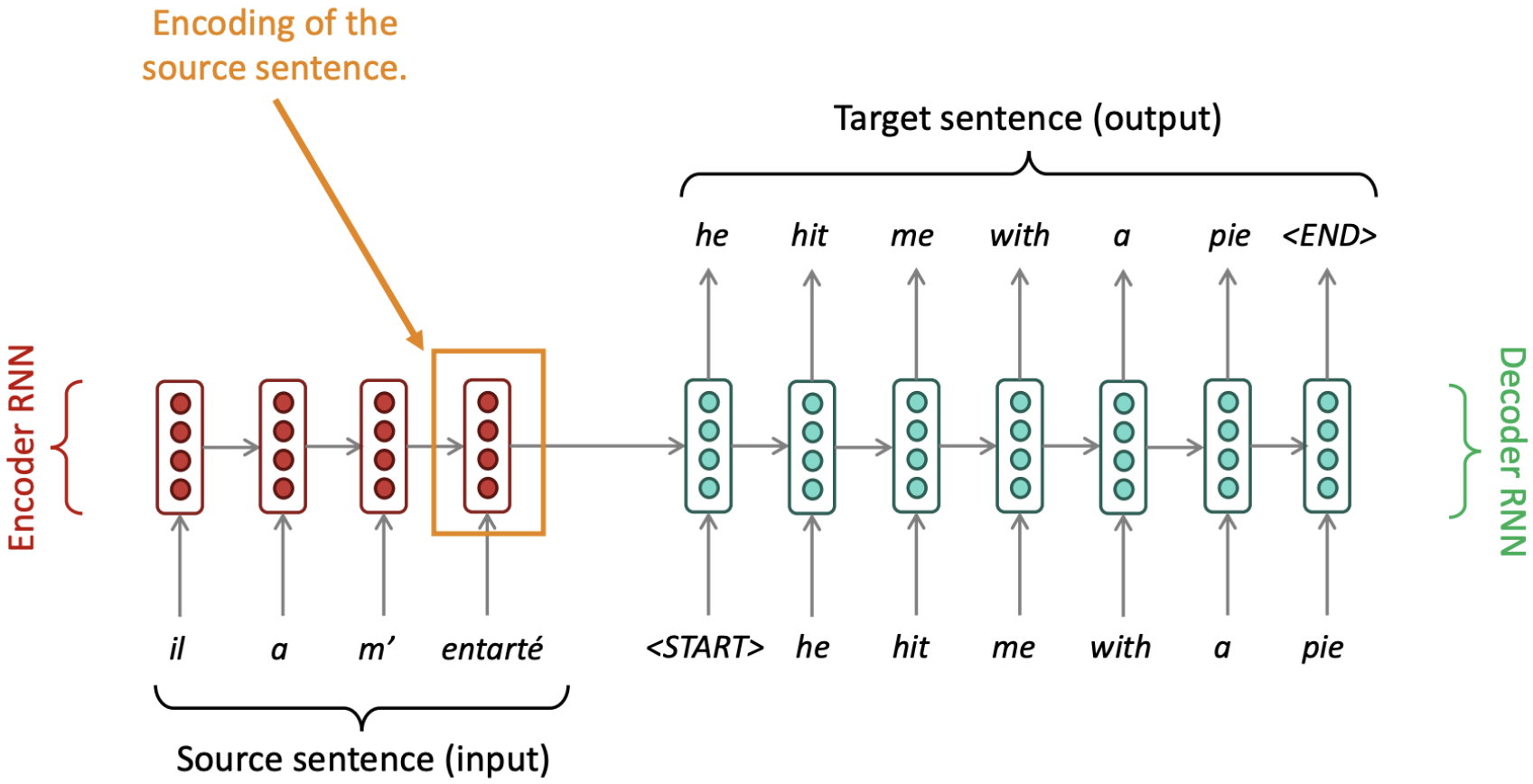
Attention



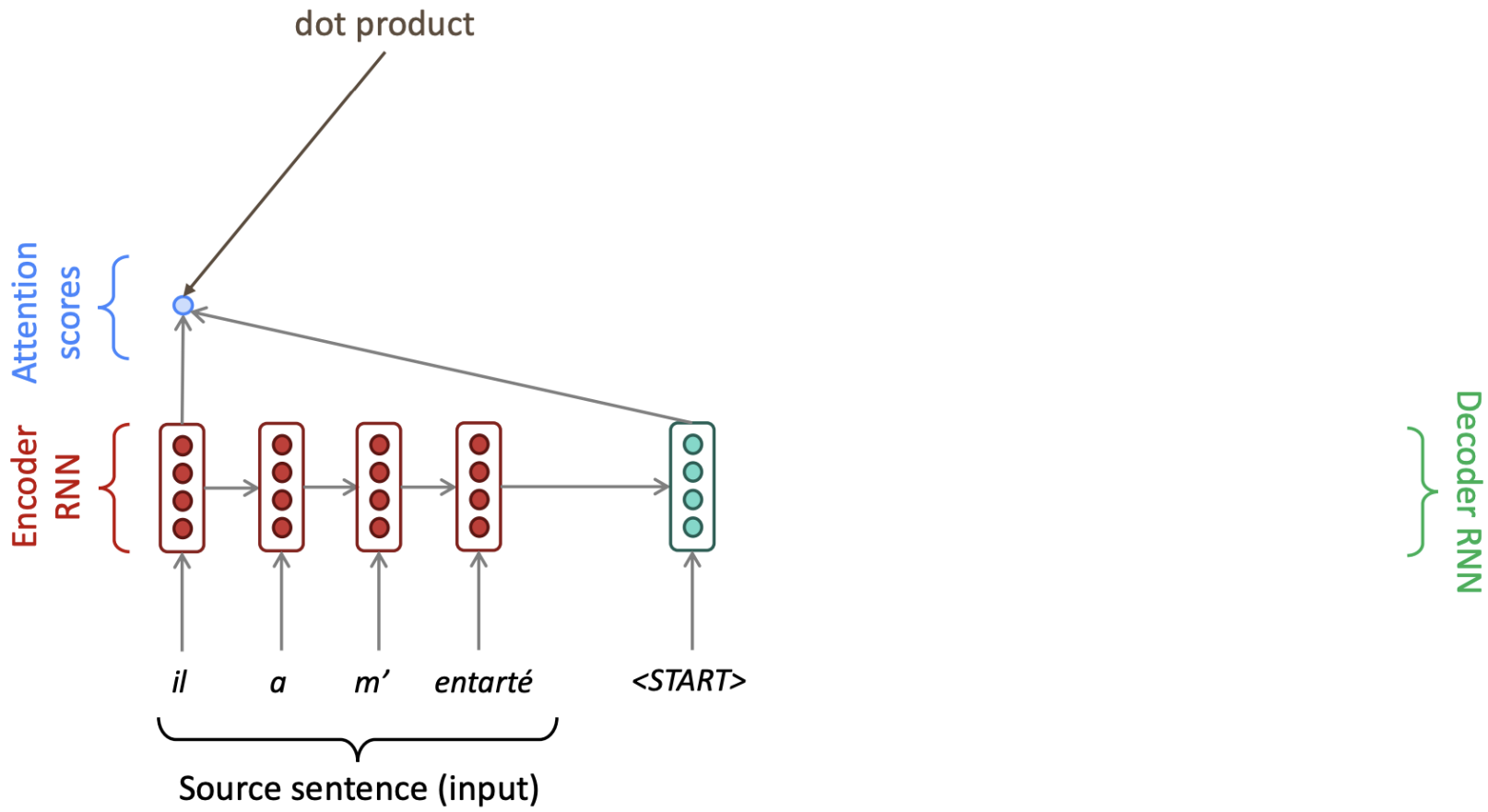
Decides what to look at! If α = (0.01,0.97,0.01,…) then we're mostly attending to $\vec{x}_2$ and just copying its value $\vec{v}_2$.

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

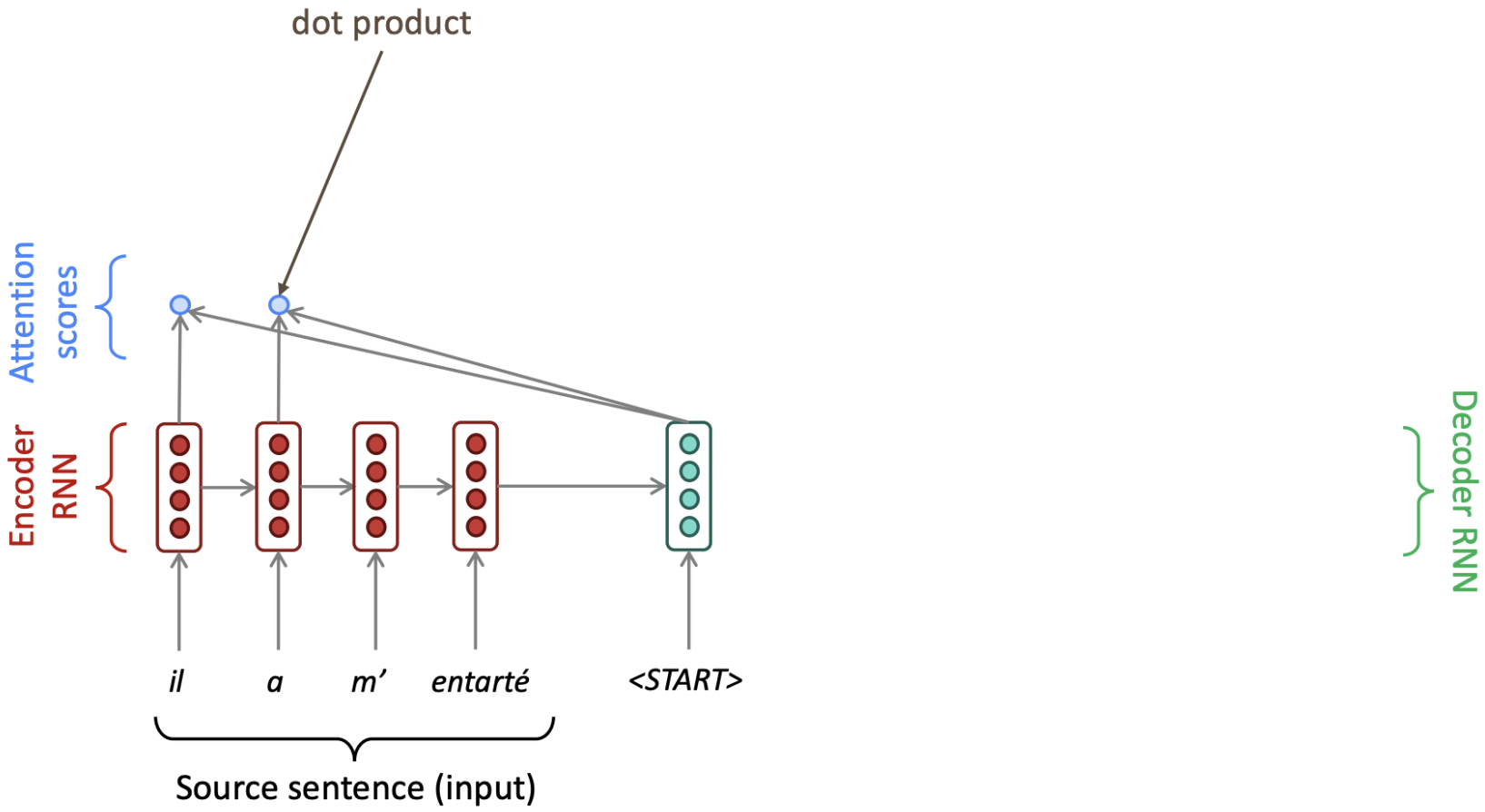
Seq2Seq Model



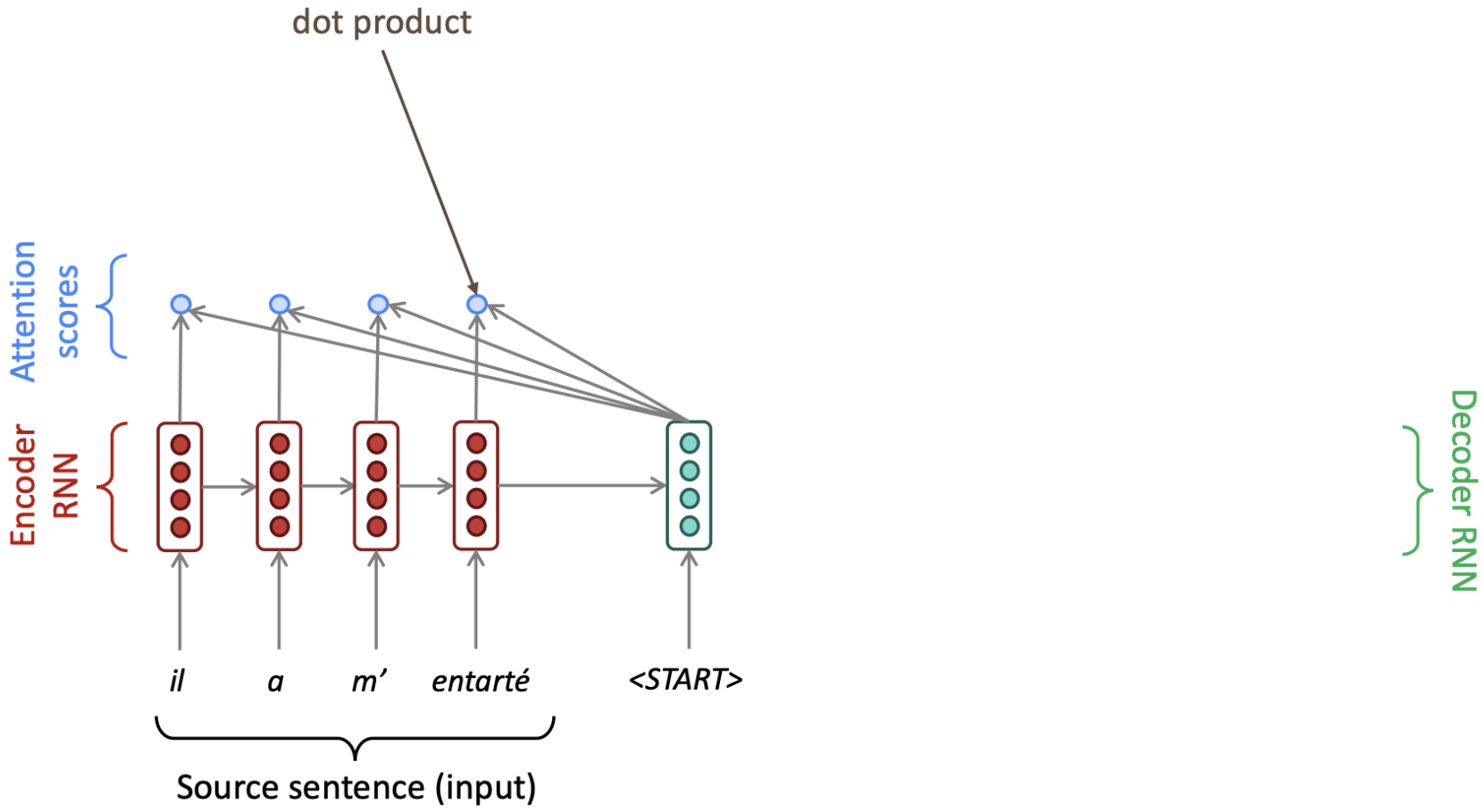
Seq2Seq w/ Attention



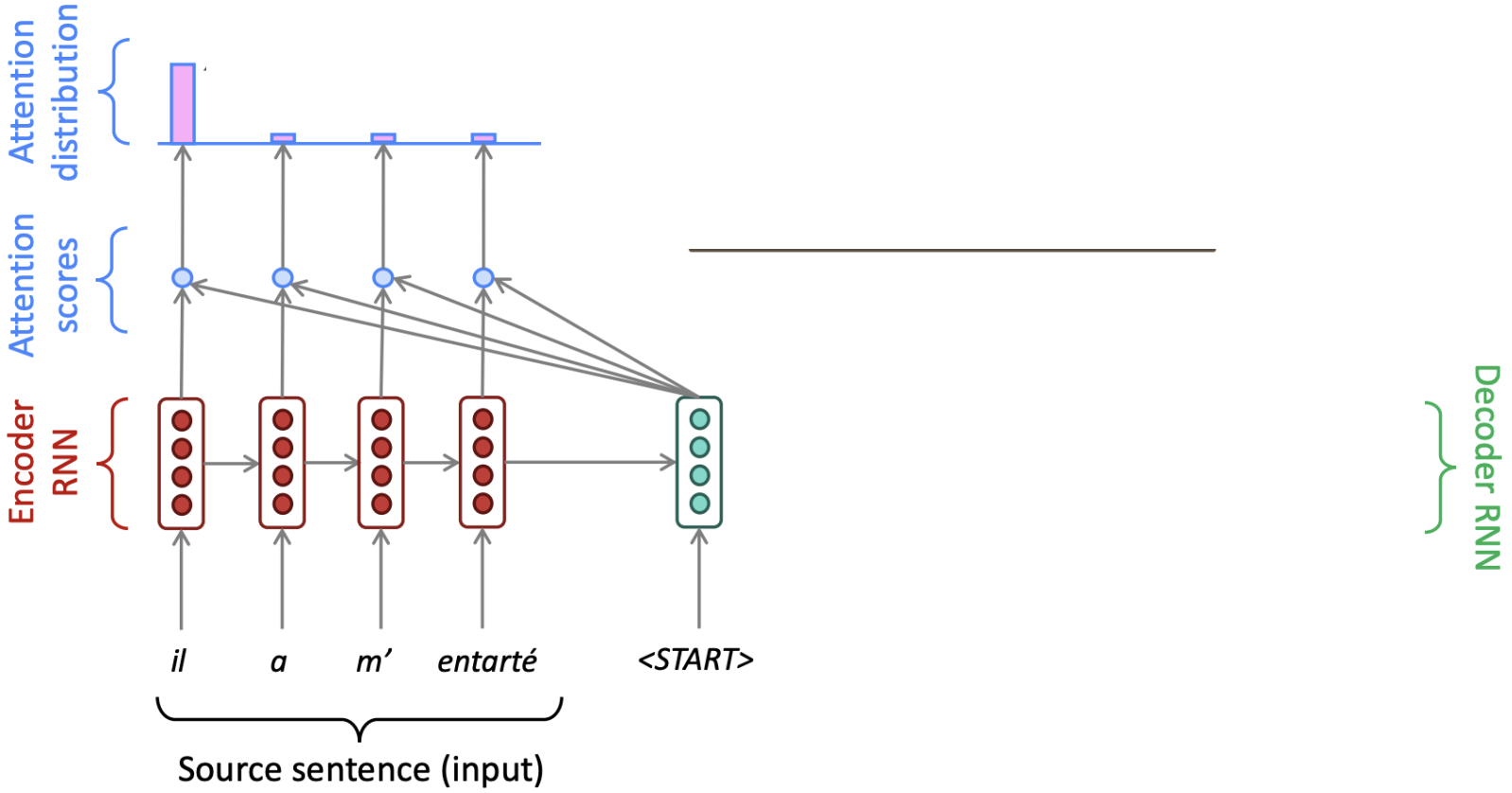
Seq2Seq w/ Attention



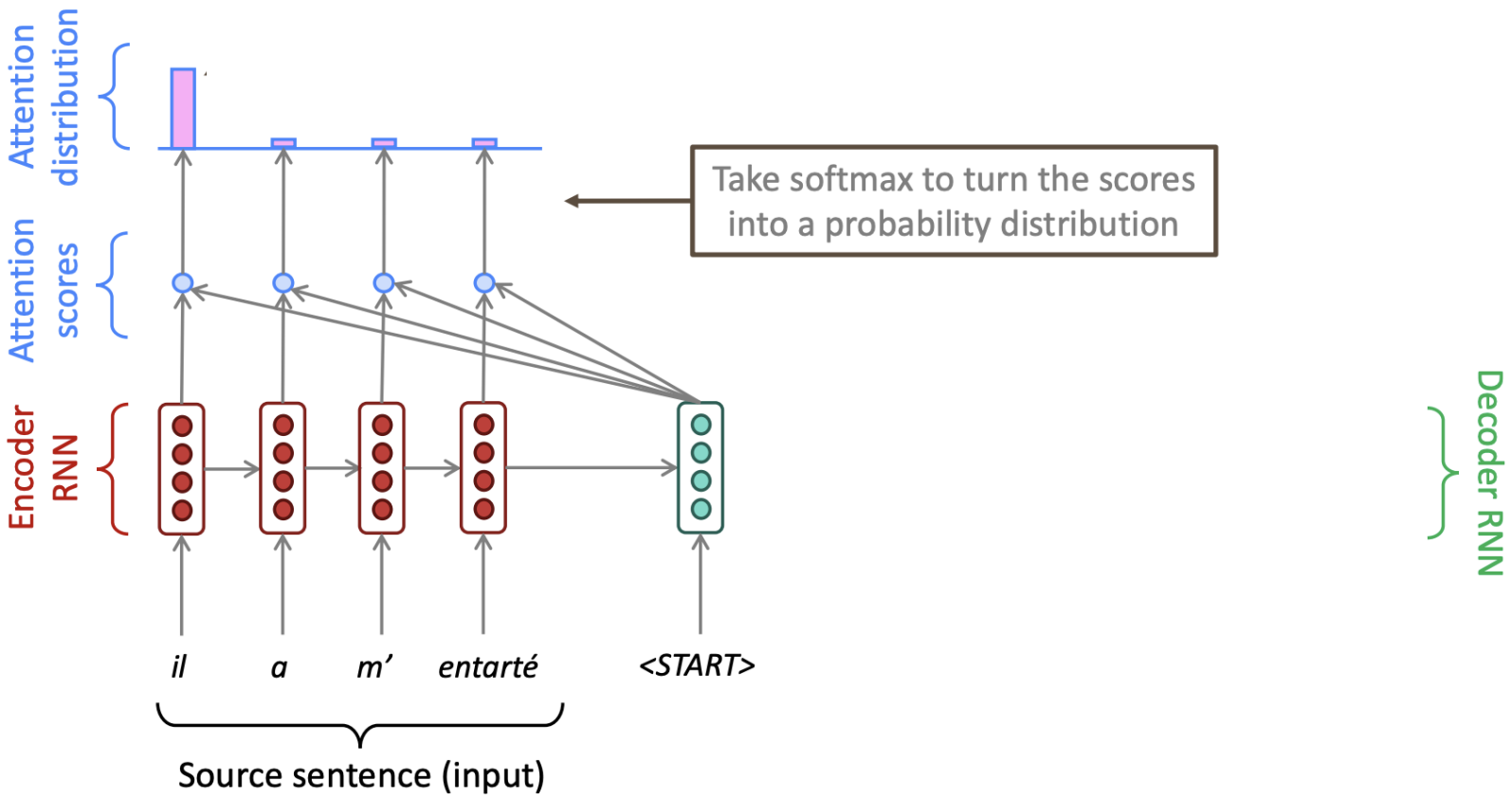
Seq2Seq w/ Attention



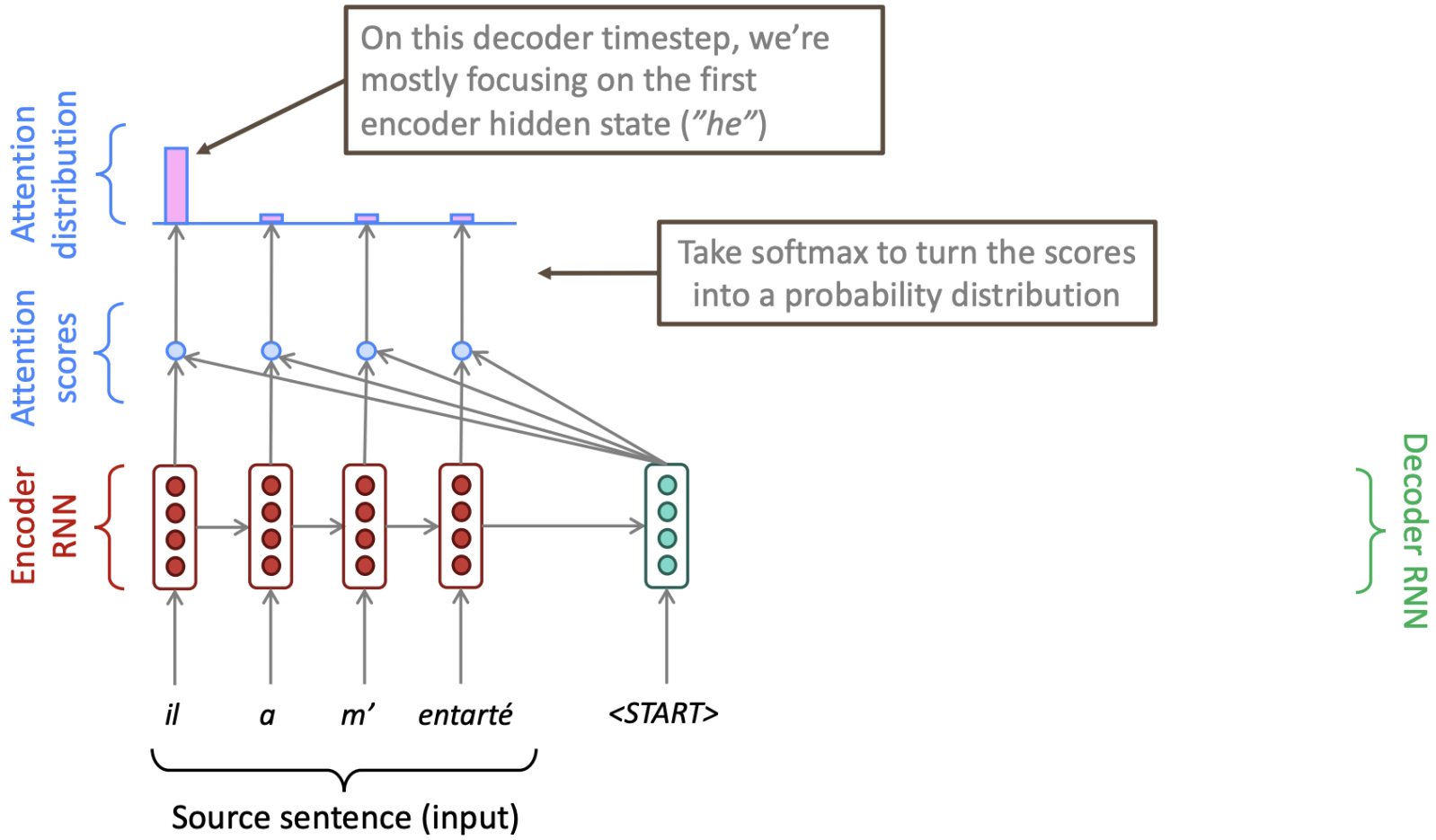
Seq2Seq w/ Attention



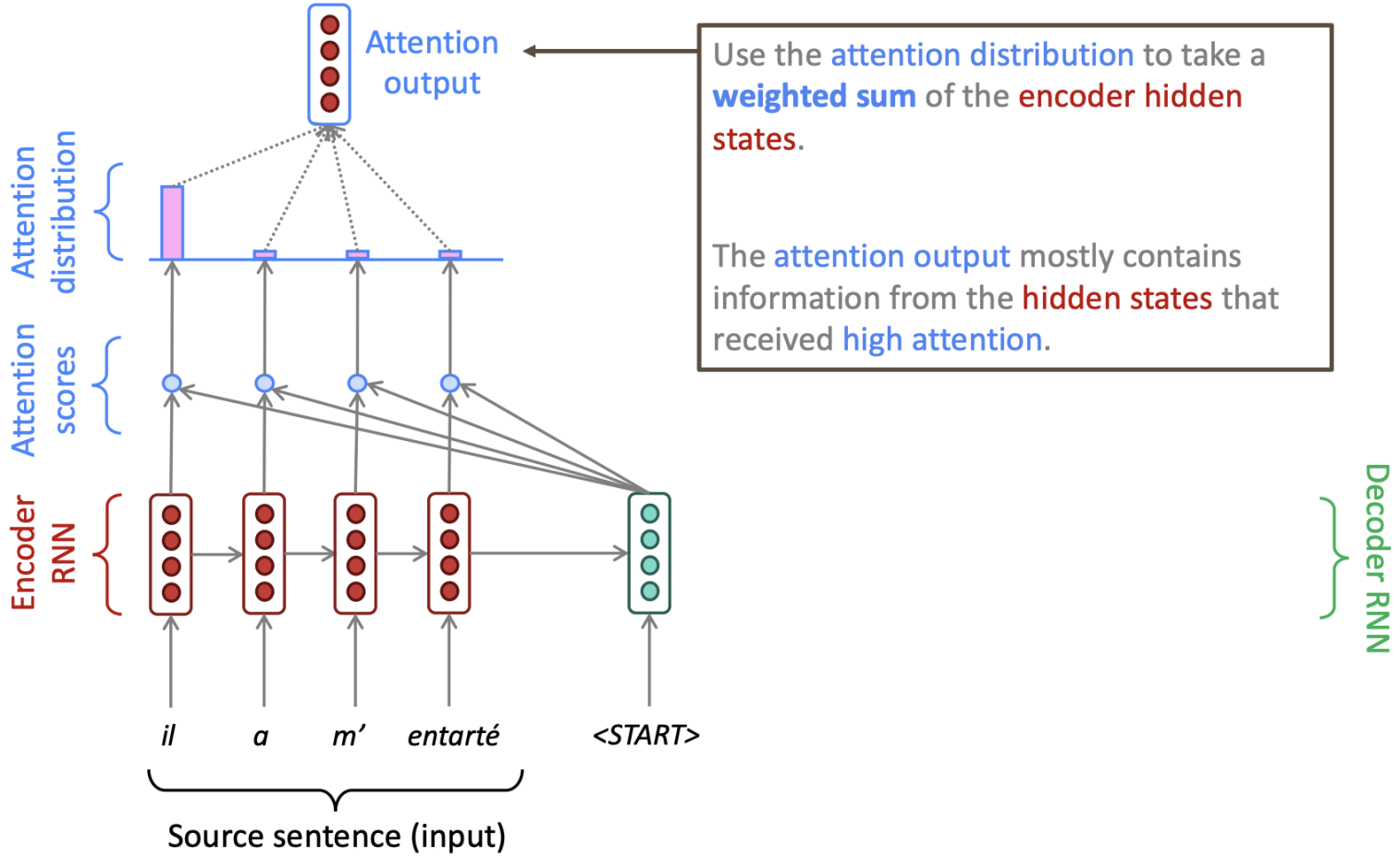
Seq2Seq w/ Attention



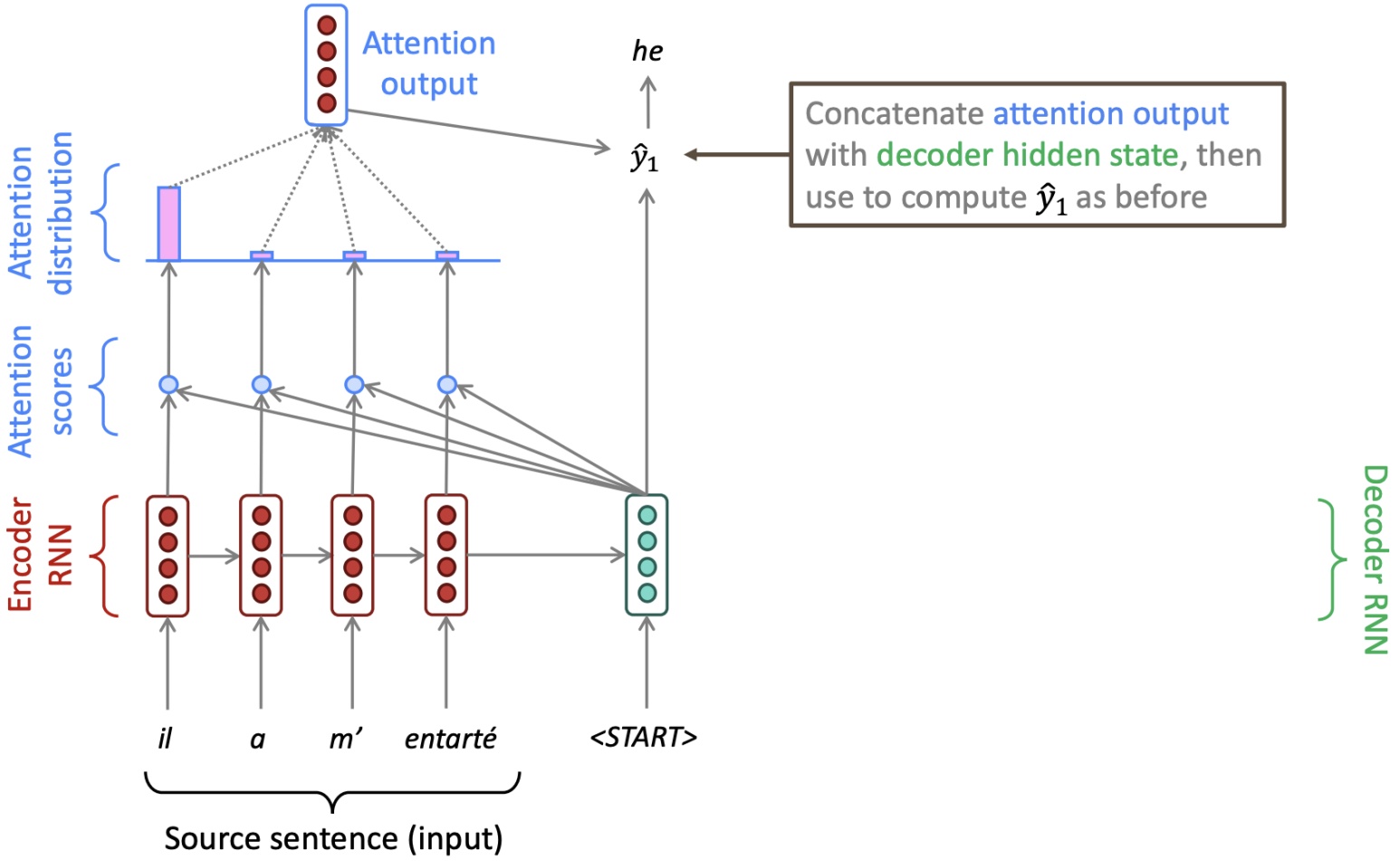
Seq2Seq w/ Attention



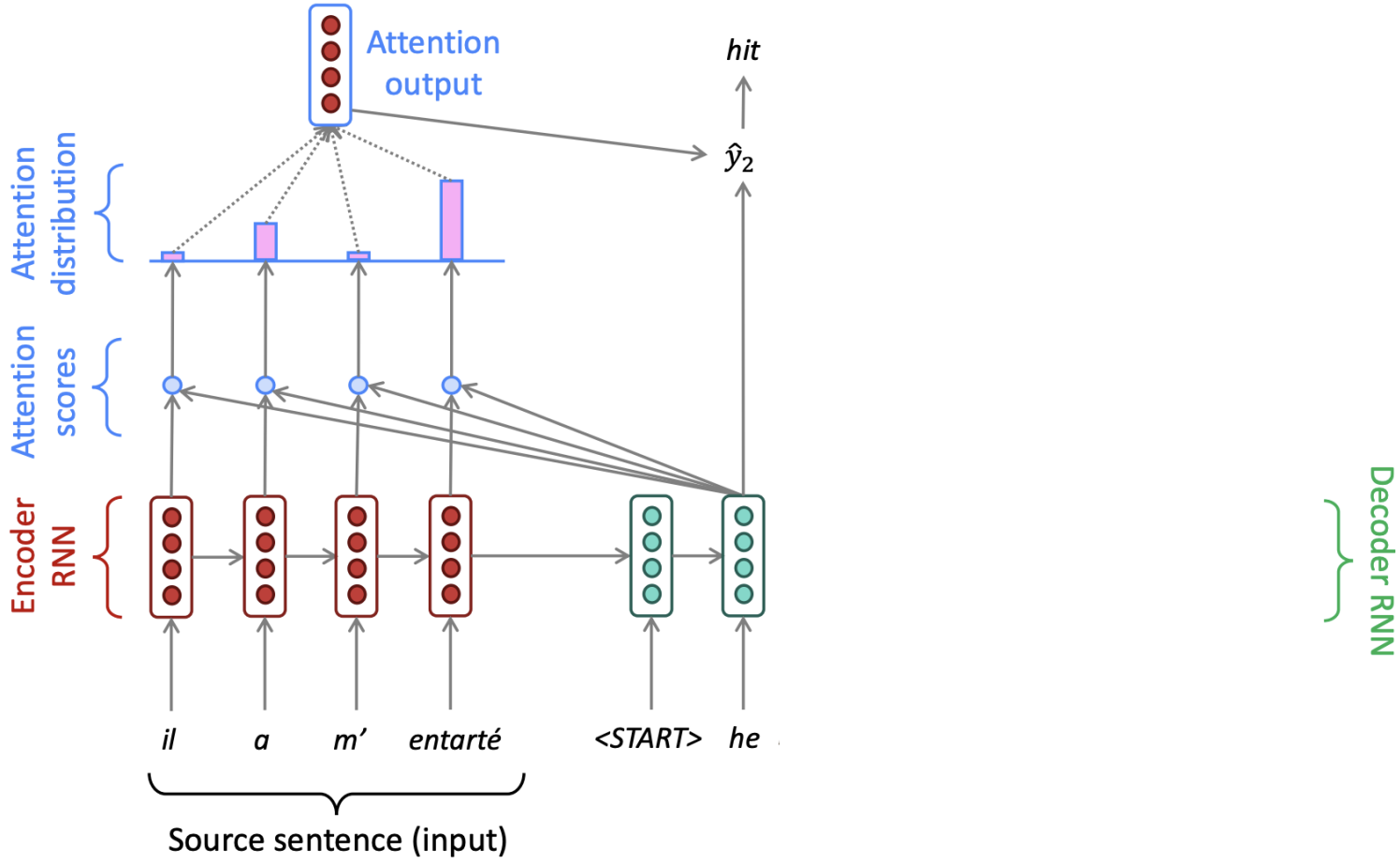
Seq2Seq w/ Attention



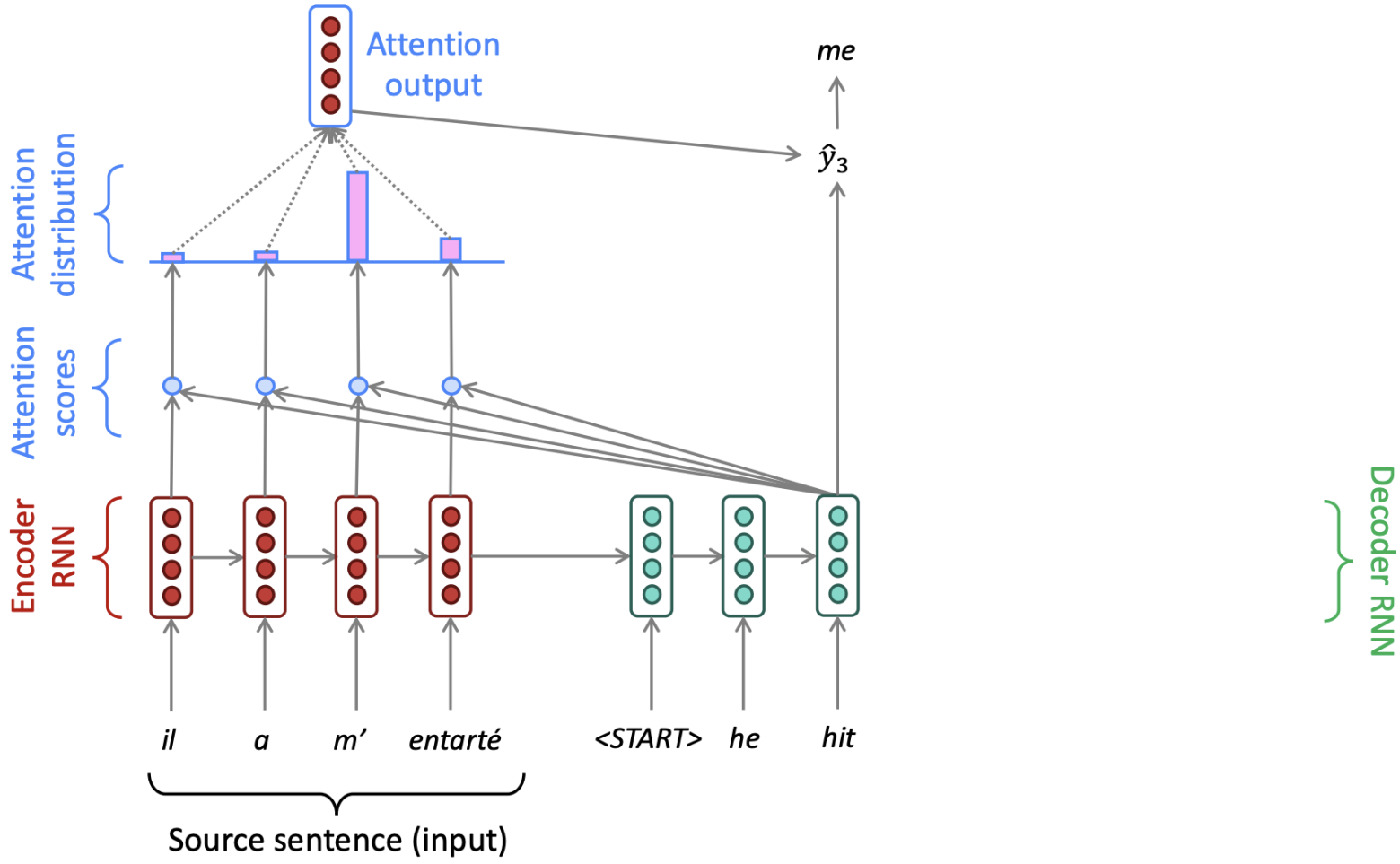
Seq2Seq w/ Attention



Seq2Seq w/ Attention



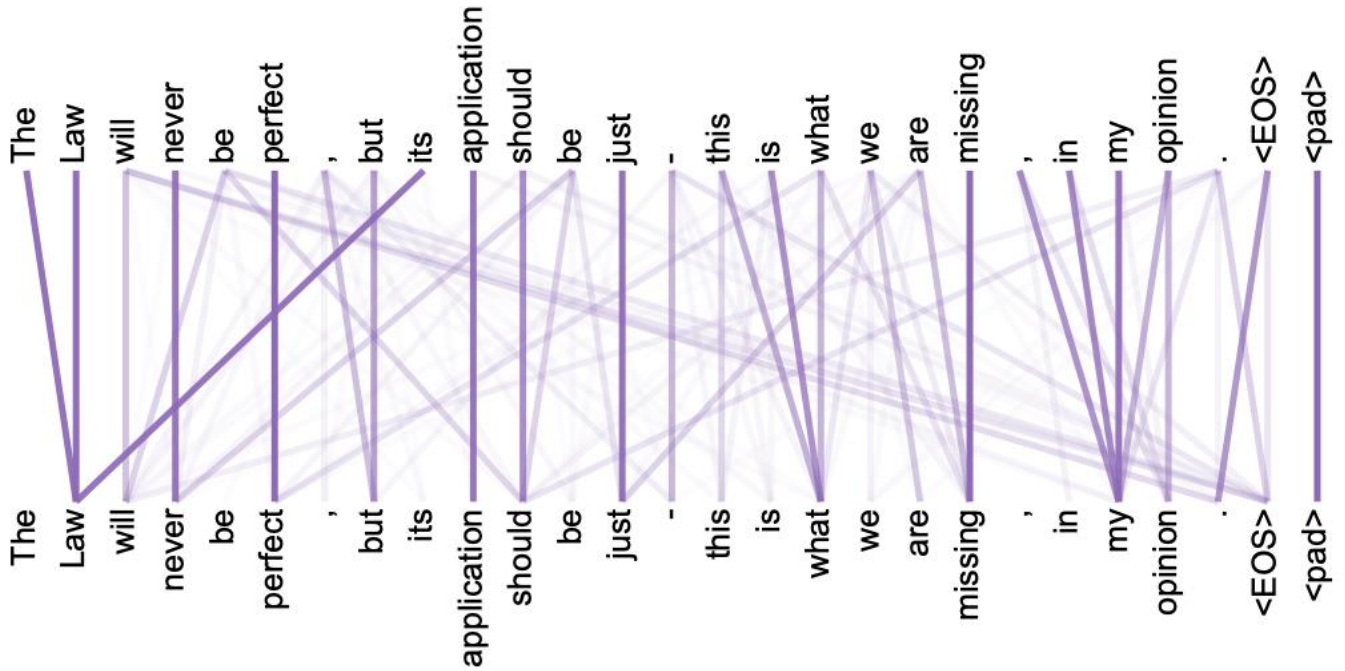
Seq2Seq w/ Attention



Attention pros!

- Significantly improves performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Helps with vanishing gradient problem
  - Provides shortcut to faraway states
- Provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on

Interpretability



Attention pros!

- Significantly improves performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Hwith vanishing gradient problem
  - Provides shortcut to faraway states
- Provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
- Can be applied to any neural model, not just decoder

Attention in a nutshell

For a new item, figure out how relevant each items is in a collection of different items

W/o attention: we are just relying on a naïve summary of the collection

Encoder-decoder setting:

- How relevant are all the words from the input to a single word in the output

Encoder-MLP setting:

- How relevant are all the words from the input to our prediction

Outline

Issues when training NNs

Pytorch

FFN’s for variable length input

RNN

LSTM

Attention

Self Attention

Self-attention in a nutshell

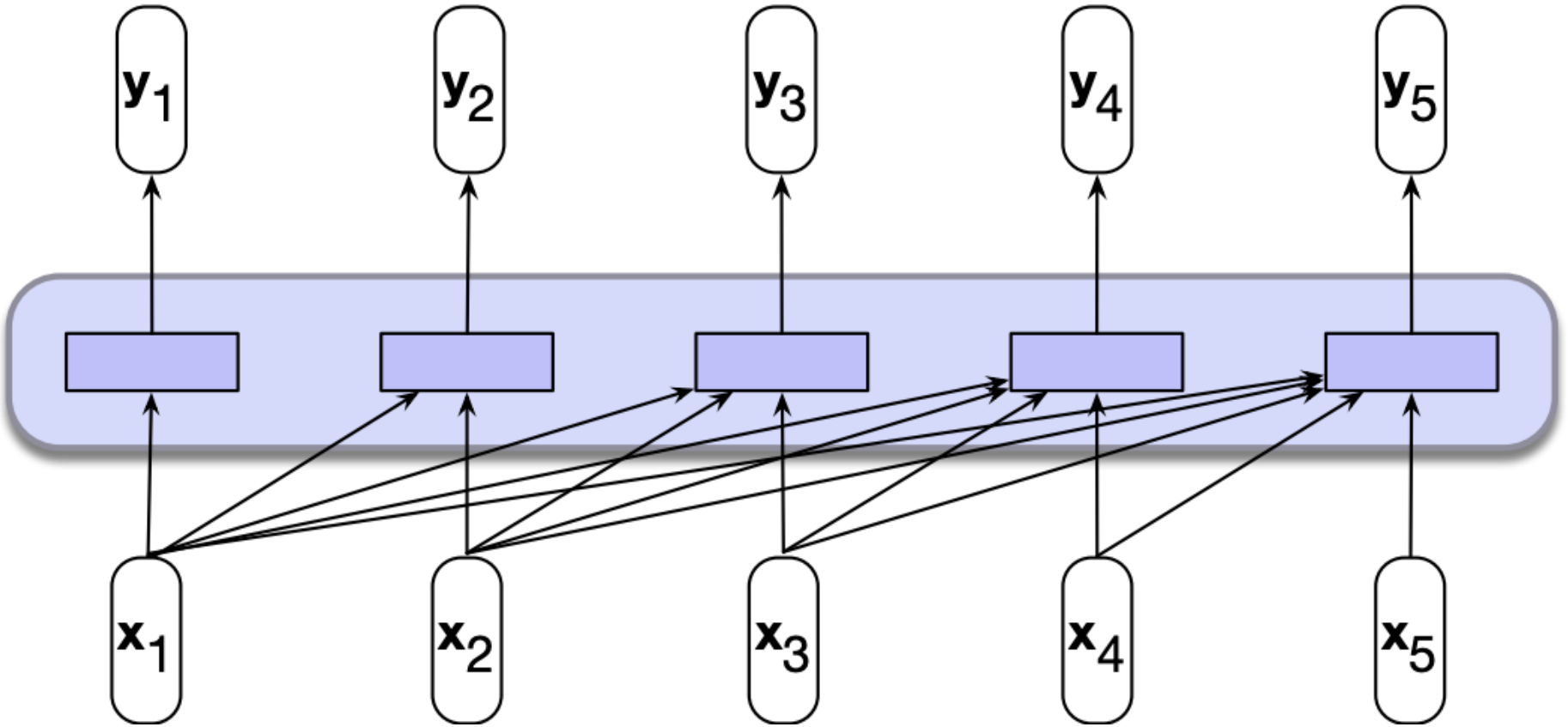
Attention:

- For a new item, figure out how relevant items are in a collection of different items

Self-attention

- How relevant are all the words from the input to a single word in the input

Self-attention



Terminology

Query:

Key:

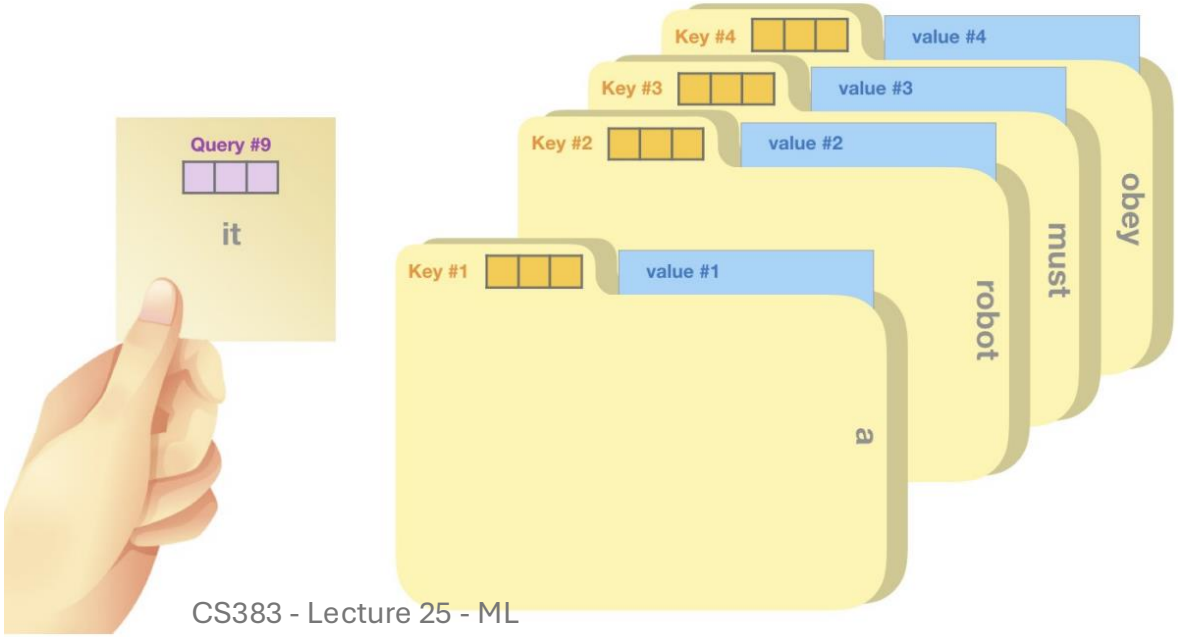
Value:

Terminology

Query: what to match

Key: the thing to match

Value: what to be extracted from the match

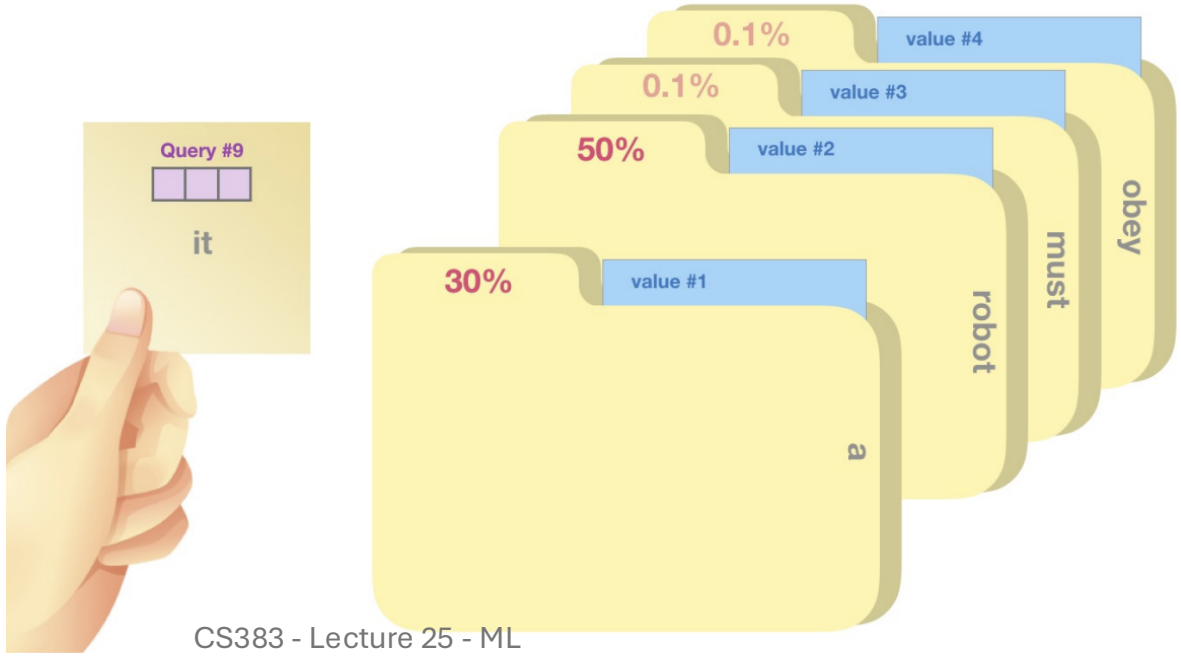


Terminology

Query: what to match

Key: the thing to match

Value: what to be extracted from the match



Terminology

Query: what to match:

$$q_i = W^q x_i$$

Key: the thing to match:

Value: what to be extracted from the match

Terminology

Query: what to match:

$$q_i = W^q x_i$$

Key: the thing to match:

$$k_i = W^k x_i$$

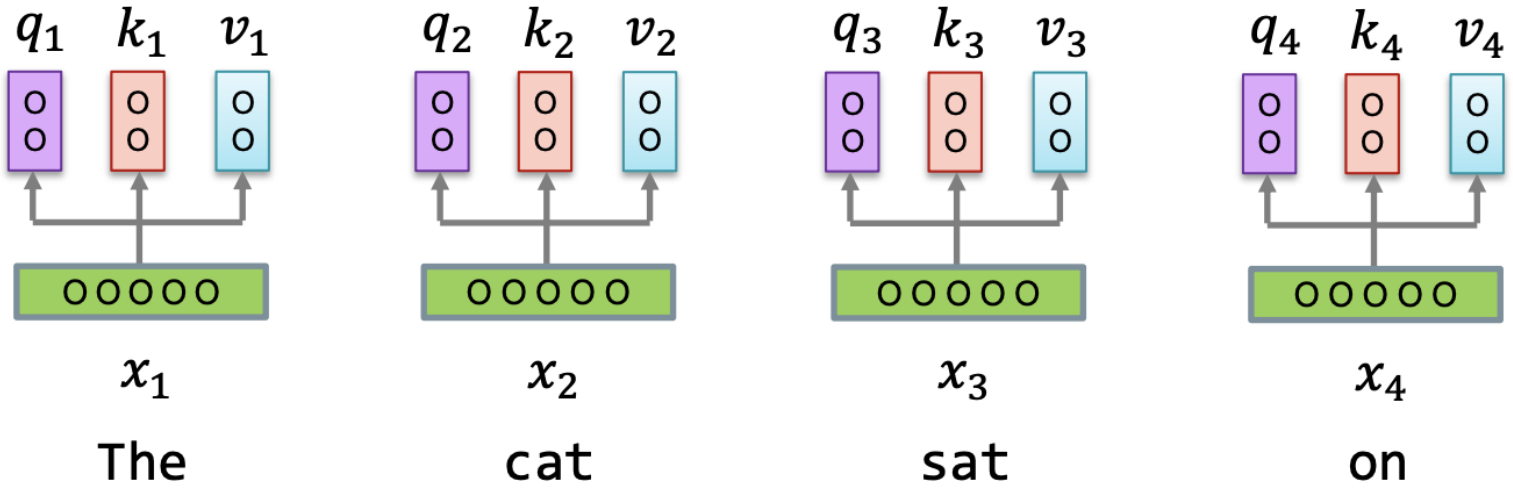
Value: what to be extracted from the match

$$v_i = W^v x_i$$

Output of each input cell

These are three representations of each input

Each representation is created by multiplying the input by a weight matrix



Self-Attention Scores

When creating a representation for x_i, how much weight/focus/attention should we give to x_j

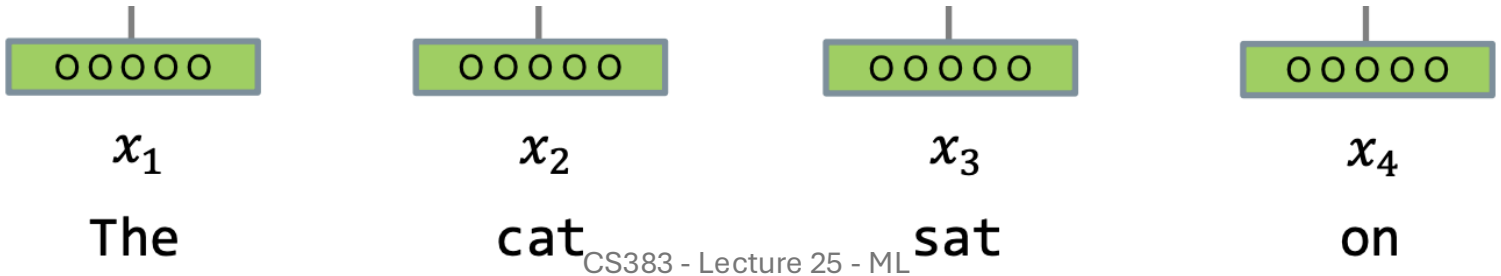
$\forall i, j \in |x|$ we must compute $score(x_i, x_j)$

Self-Attention Scores

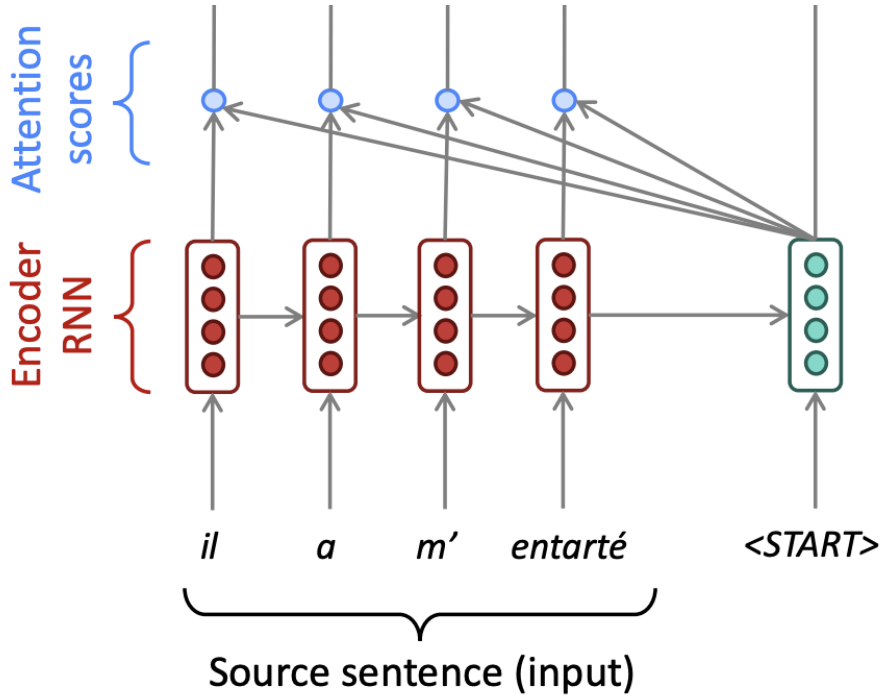
$\forall i, j \in |x|$ we must compute $score(x_i, x_j)$

Question: are these scores distance functions?

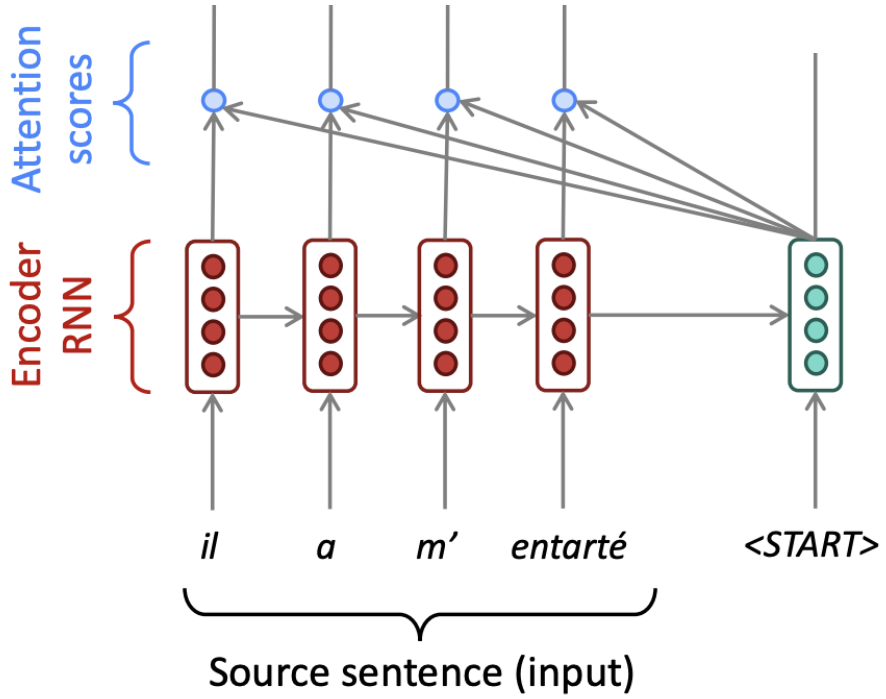
No! $score(x_i, x_j)$ shouldn't be equal to $score(x_i, x_j)$

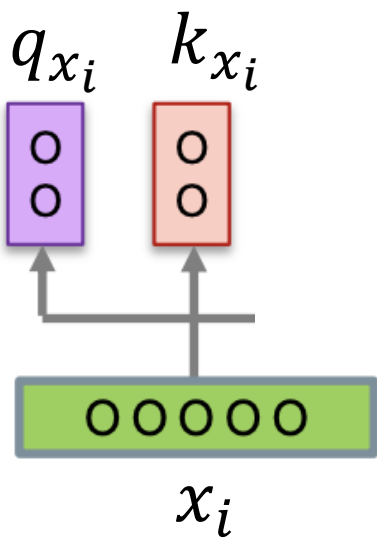


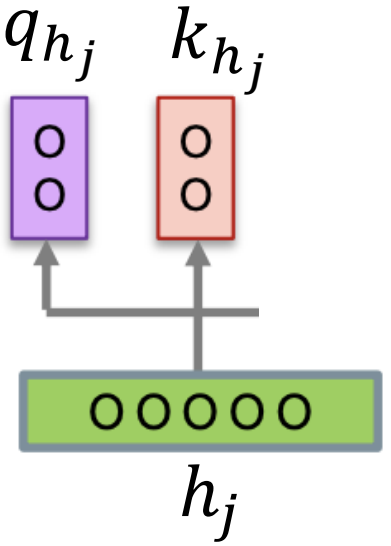
Attention score



Attention score

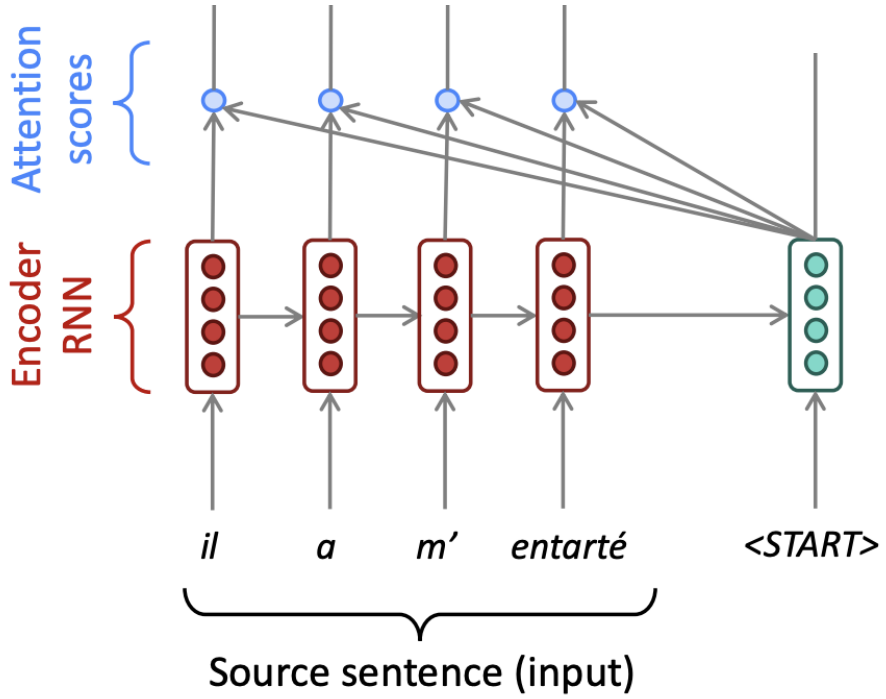


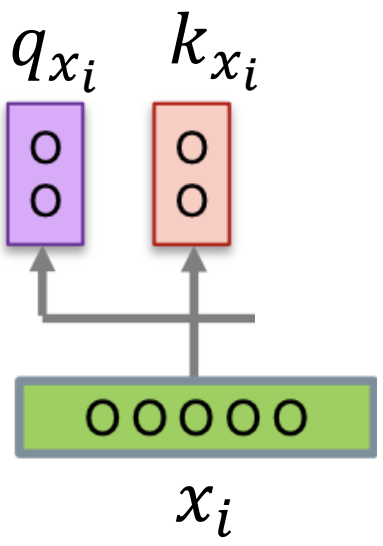


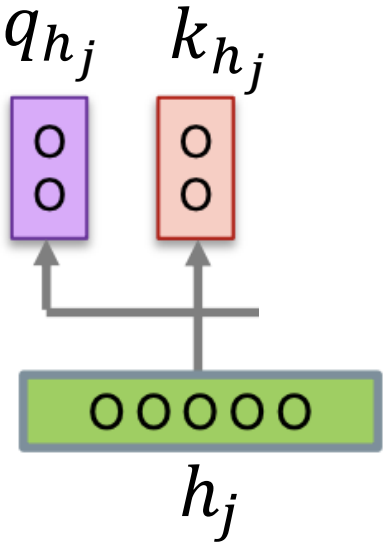


$$score(h_j, x_i) = ?$$

Attention score

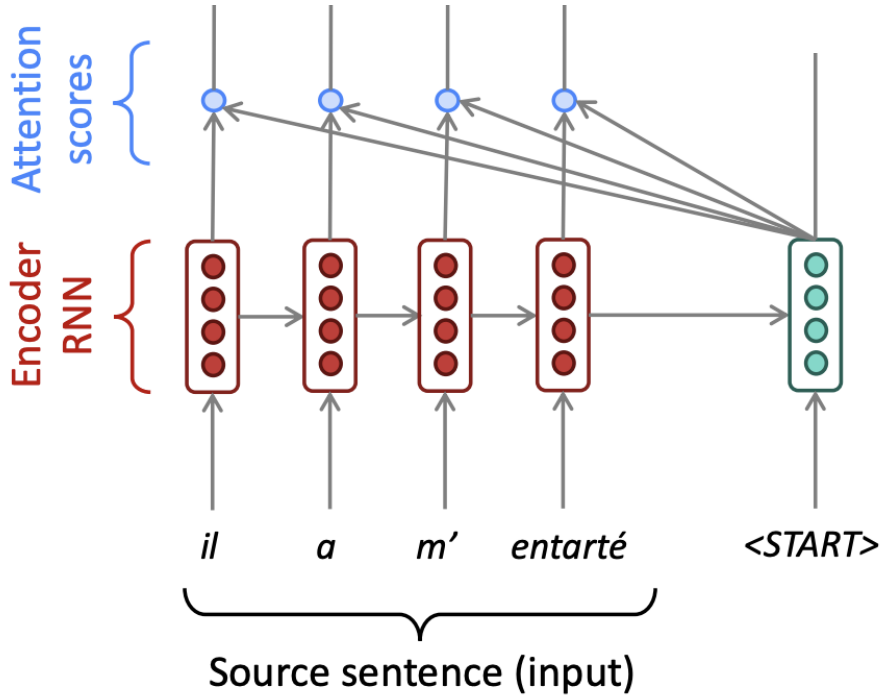


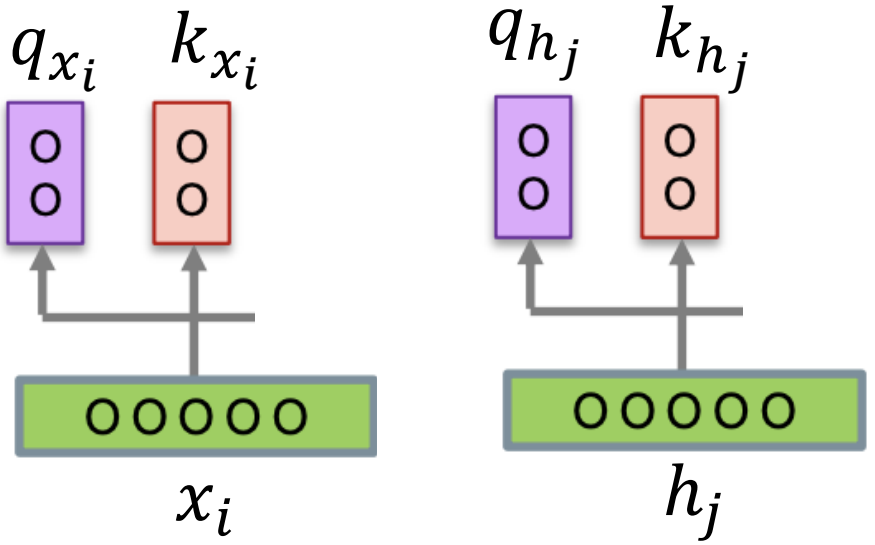




$$score(h_j, x_i) = q_{h_j} \cdot k_{x_i}$$

Attention score





$$score(h_j, x_i) = \frac{q_{h_j} \cdot k_{x_i}}{\sqrt{d_k}}$$

Attention Scores

$$A_{score} = \begin{bmatrix} \alpha_{1,1} & \cdots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \cdots & \alpha_{m,n} \end{bmatrix}$$

$$= \begin{bmatrix} score(h_1, x_1) & \cdots & score(h_1, x_n) \\ \vdots & \ddots & \vdots \\ score(h_m, x_1) & \cdots & score(h_m, x_n) \end{bmatrix}$$

$$= \begin{bmatrix} \frac{q_{h_1} \cdot k_{x_1}}{\sqrt{d_k}} & \cdots & \frac{q_{h_1} \cdot k_{x_n}}{\sqrt{d_k}} \\ \vdots & \ddots & \vdots \\ \frac{q_{h_m} \cdot k_{x_1}}{\sqrt{d_k}} & \cdots & \frac{q_{h_m} \cdot k_{x_n}}{\sqrt{d_k}} \end{bmatrix}$$

We can store all the q's and k's in a matrix as well

Attention Scores

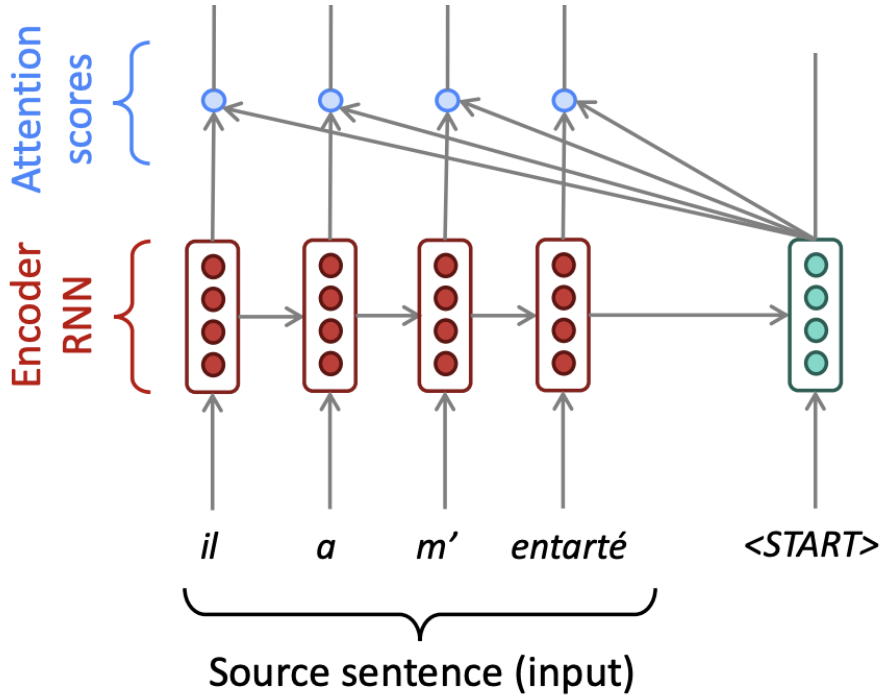
$$A_{score} = \begin{bmatrix} \alpha_{1,1} & \cdots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \cdots & \alpha_{m,n} \end{bmatrix}$$

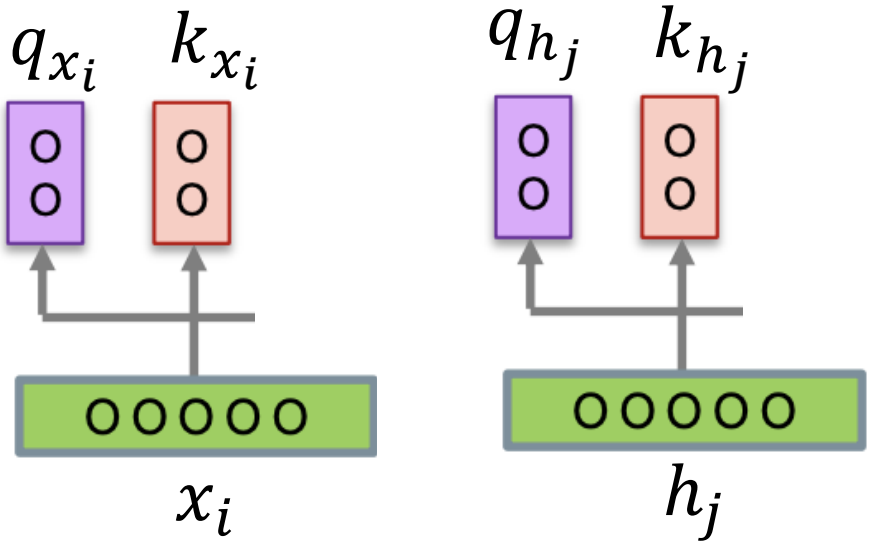
$$= \begin{bmatrix} \frac{q_{h_1} \cdot k_{x_1}}{\sqrt{d_k}} & \cdots & \frac{q_{h_1} \cdot k_{x_n}}{\sqrt{d_k}} \\ \vdots & \ddots & \vdots \\ \frac{q_{h_m} \cdot k_{x_1}}{\sqrt{d_k}} & \cdots & \frac{q_{h_m} \cdot k_{x_n}}{\sqrt{d_k}} \end{bmatrix}$$

$$A_{score} = \frac{QK^T}{\sqrt{d_k}}$$

We can store all the q's and k's in a matrix as well

Attention score

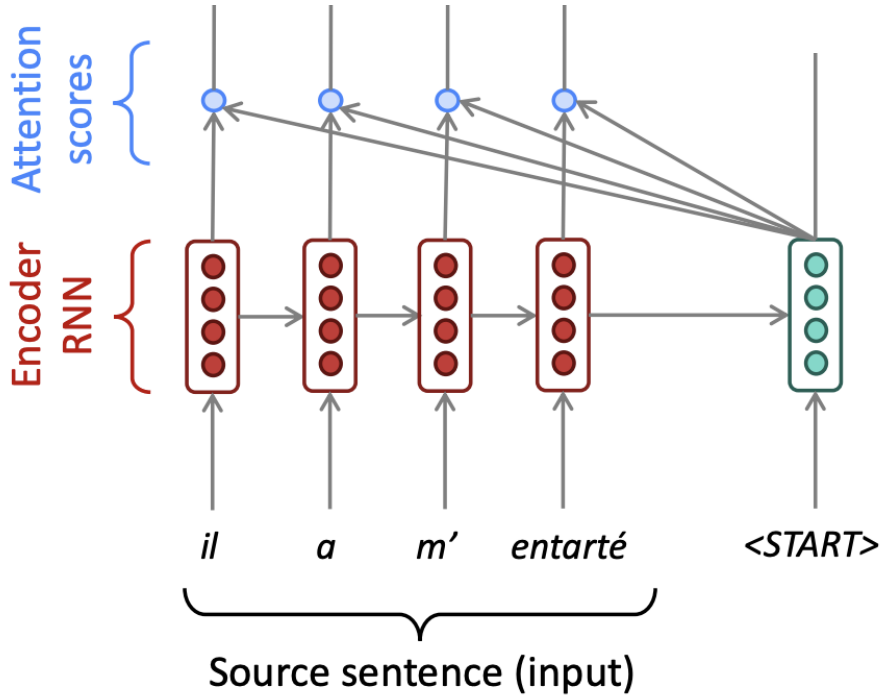


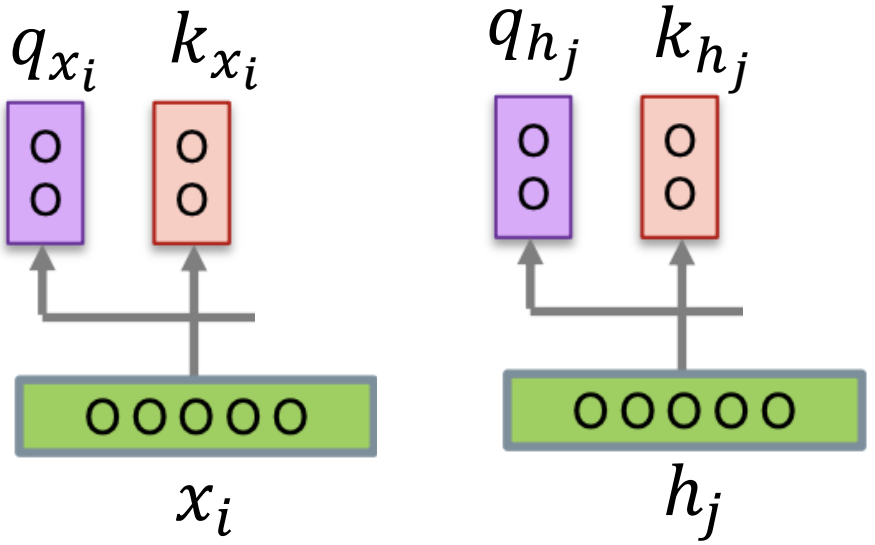


$$score(h_j, x_i) = \frac{q_{h_j} \cdot k_{x_i}}{\sqrt{d_k}}$$

$$A_{score} = \frac{QK^T}{\sqrt{d_k}}$$

Attention

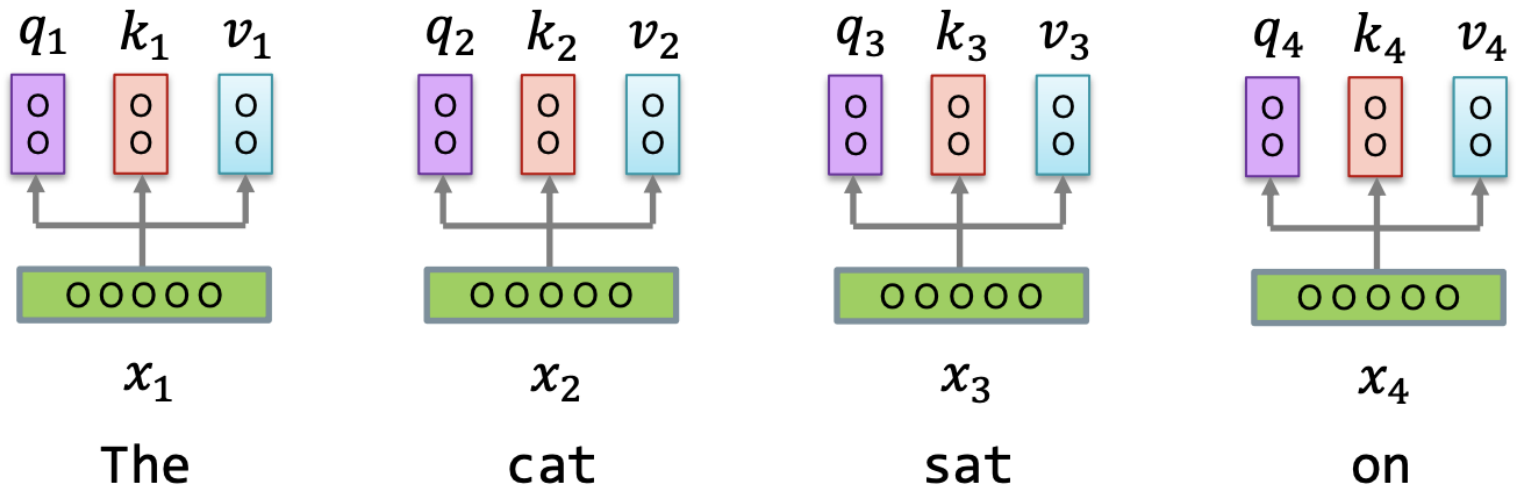




$$A = softmax(\frac{QK^T}{\sqrt{d_k}})$$

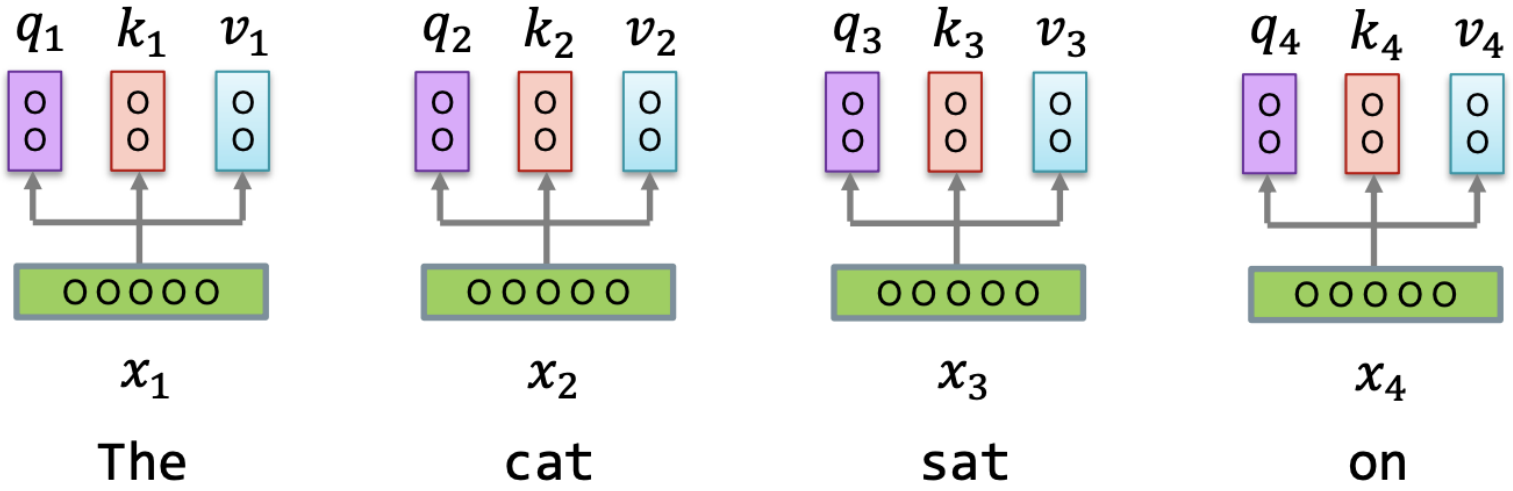
Self-attention

When creating a representation for x_i, how much weight/focus/attention should we give to x_j

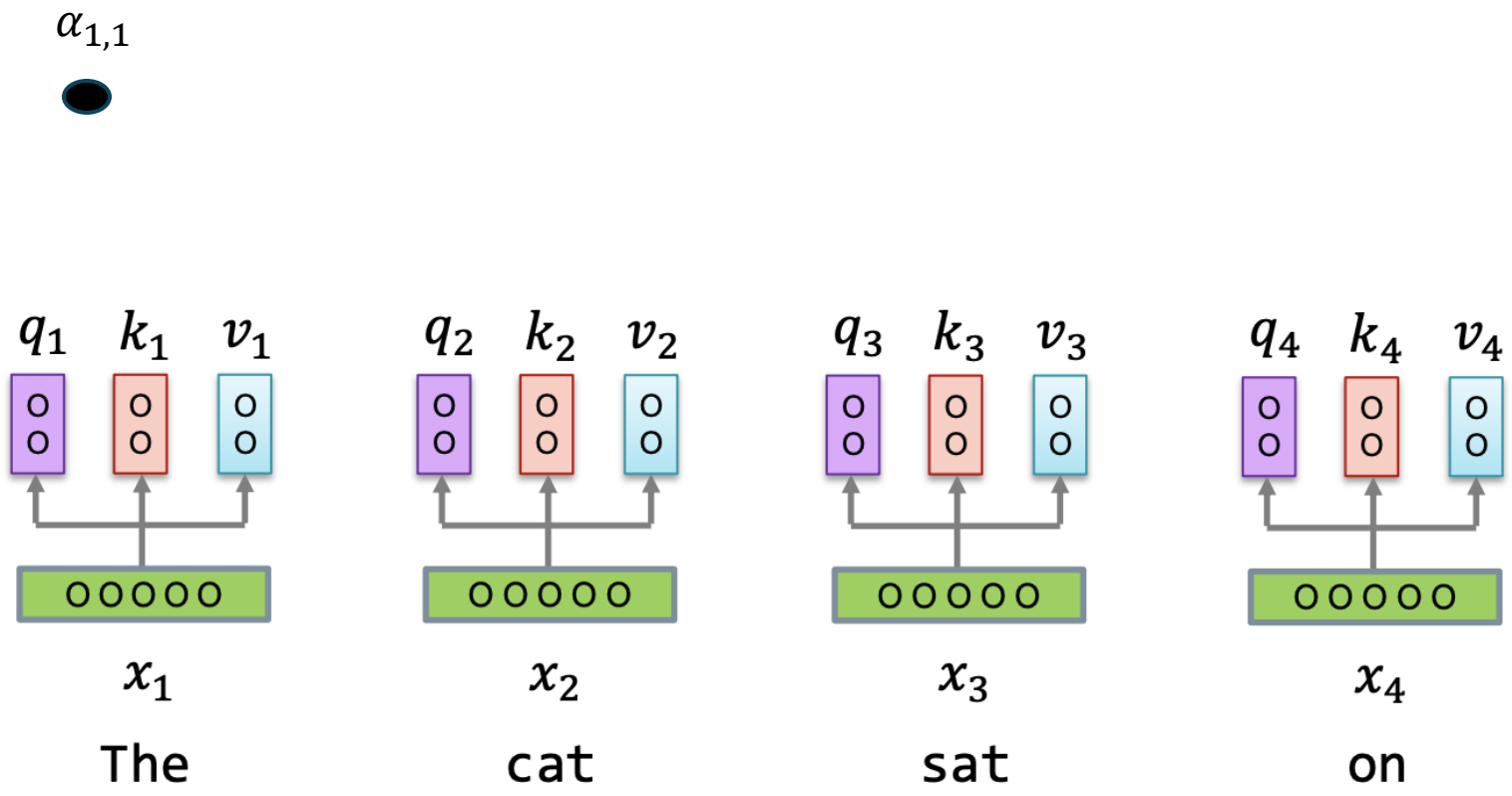


Self-attention

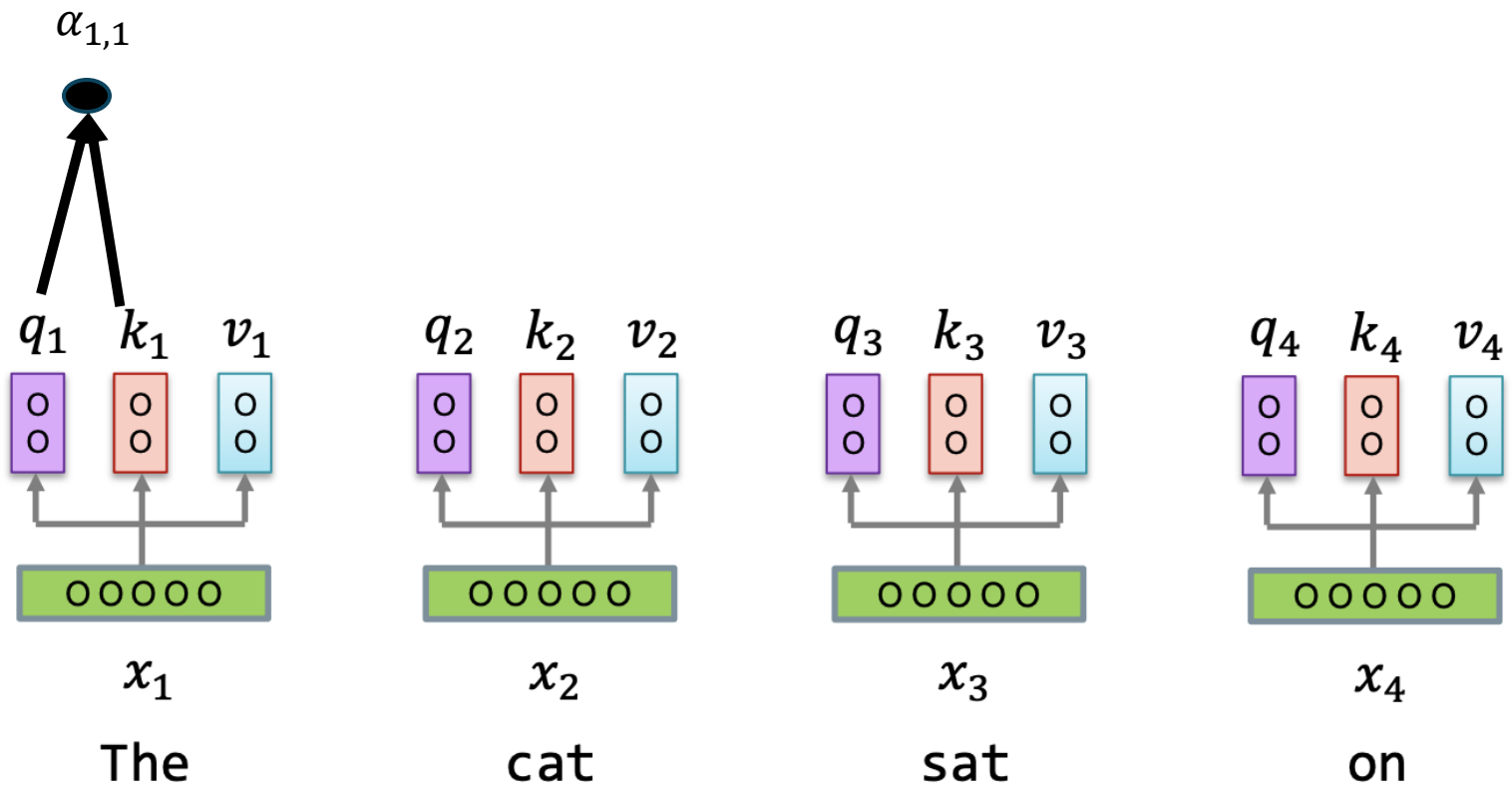
When creating a representation for x_1, how much weight/focus/attention should we give to x_j



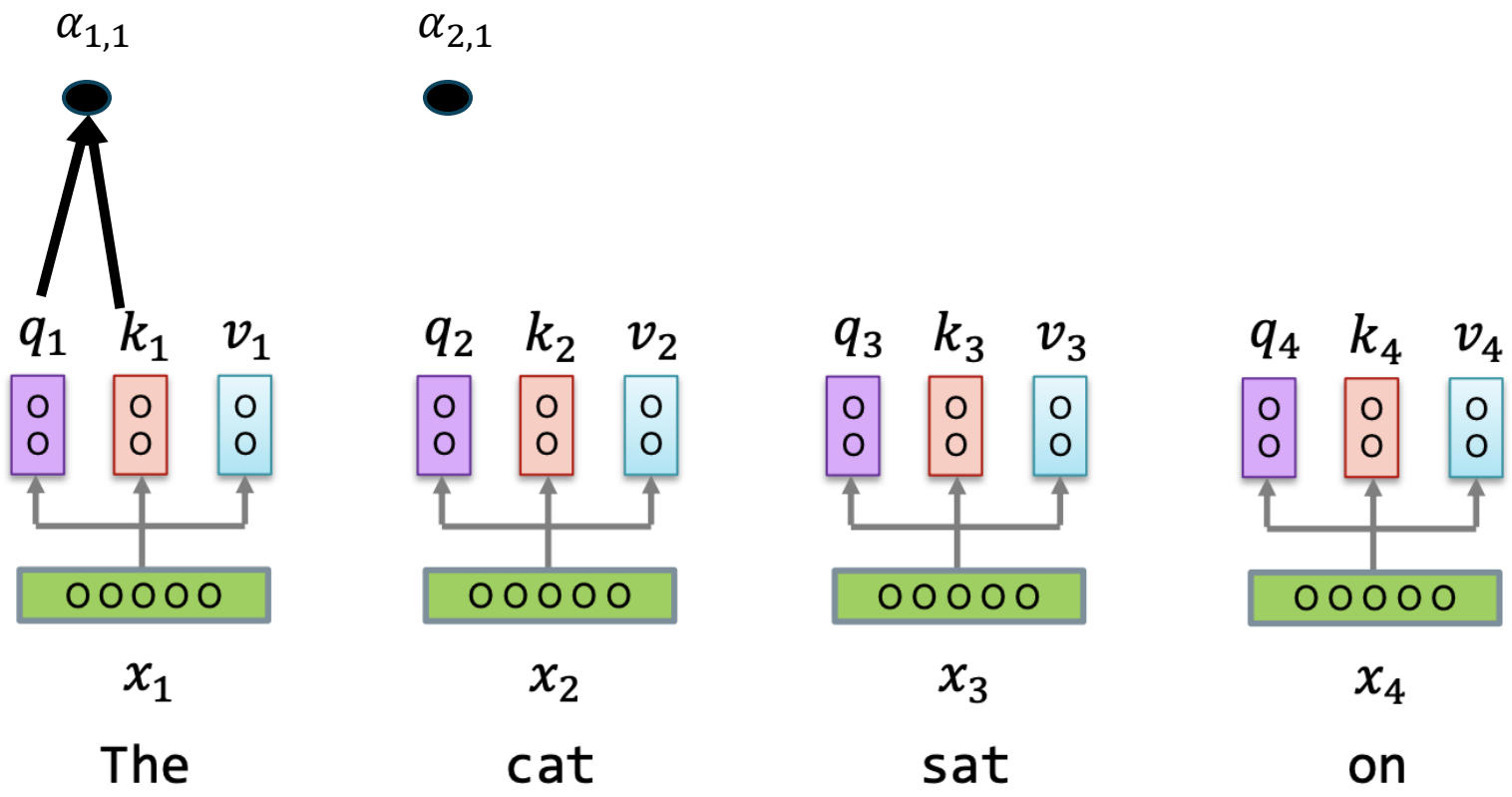
Self-attention



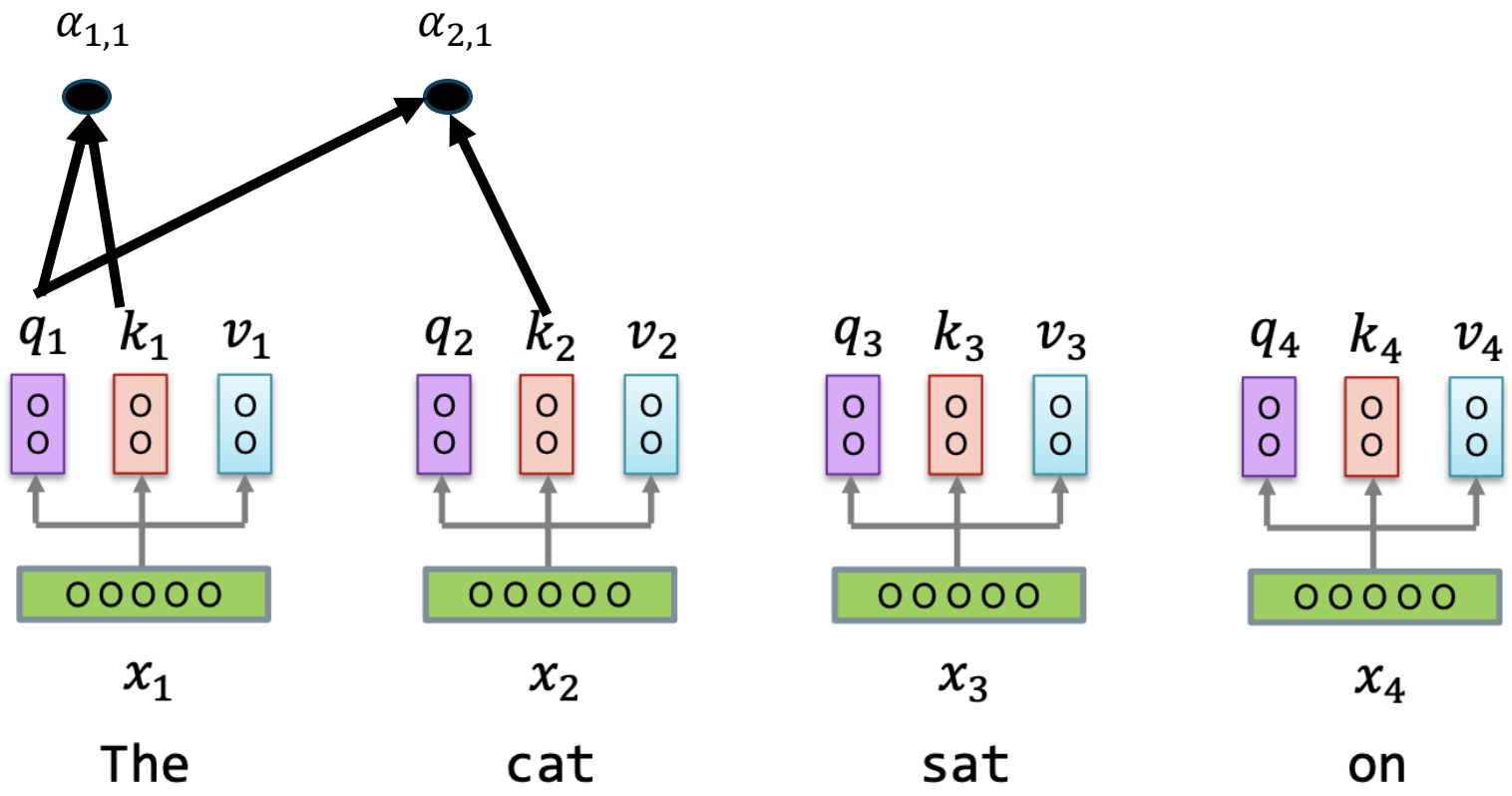
Self-attention



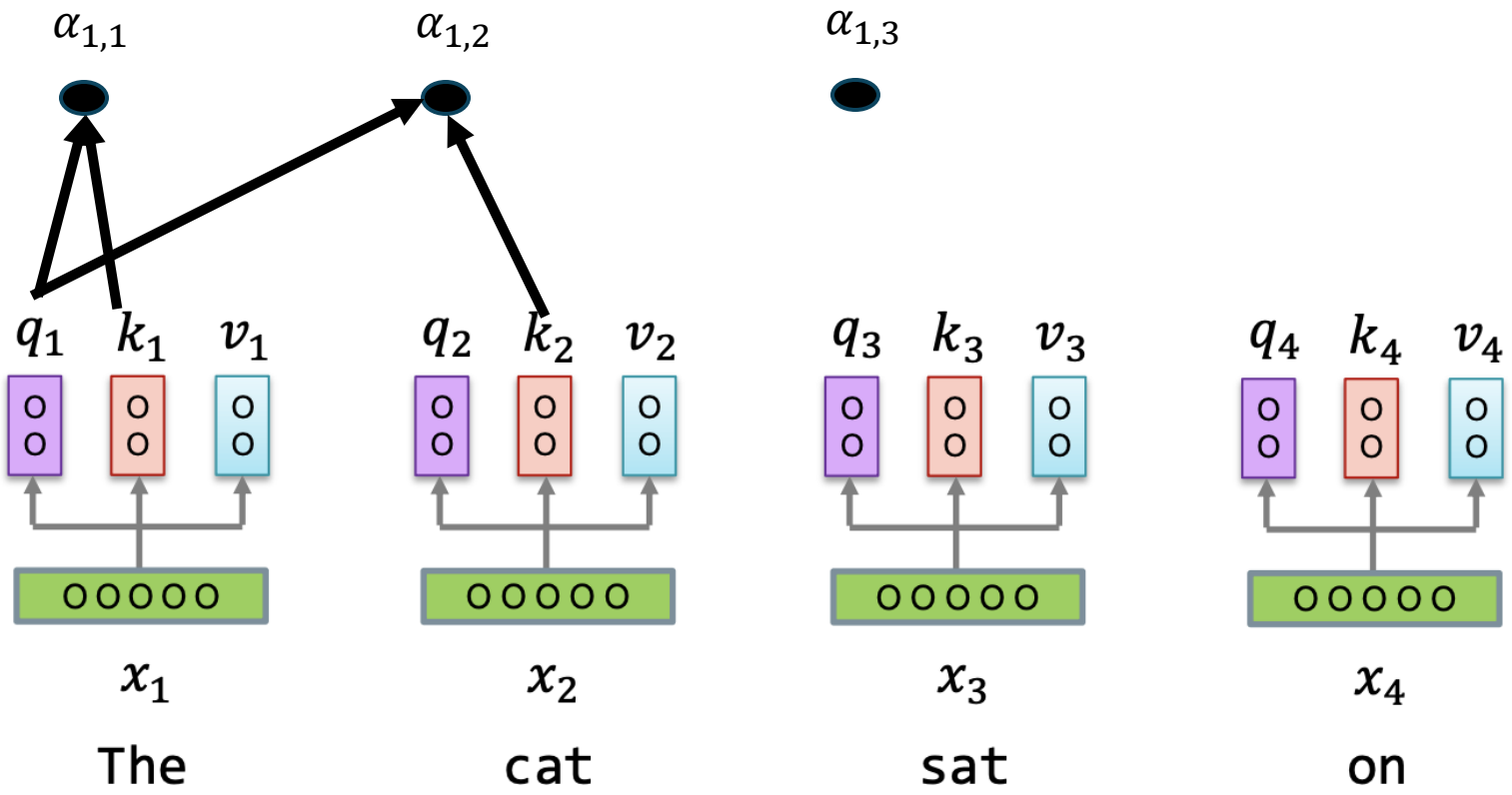
Self-attention



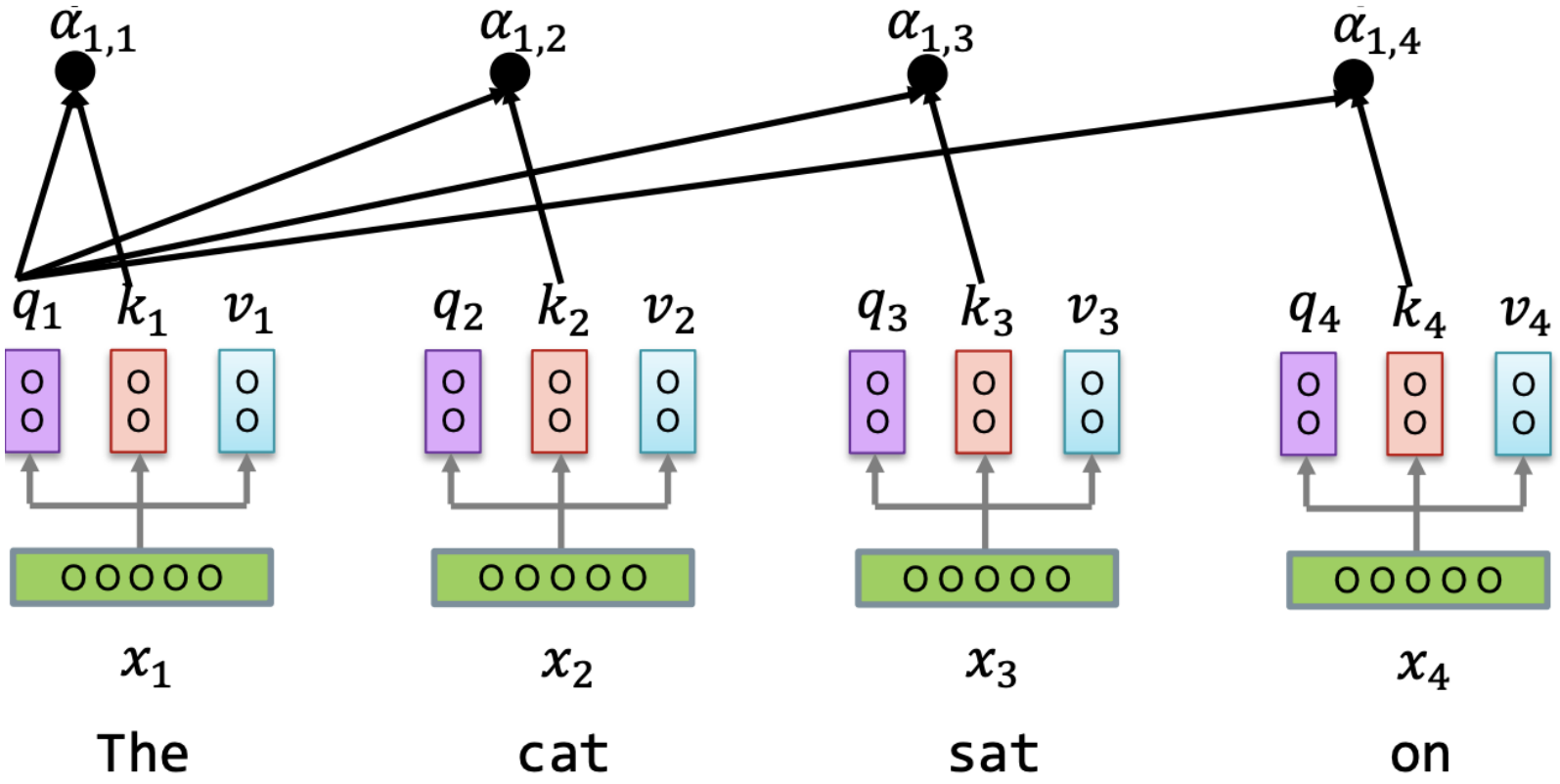
Self-attention



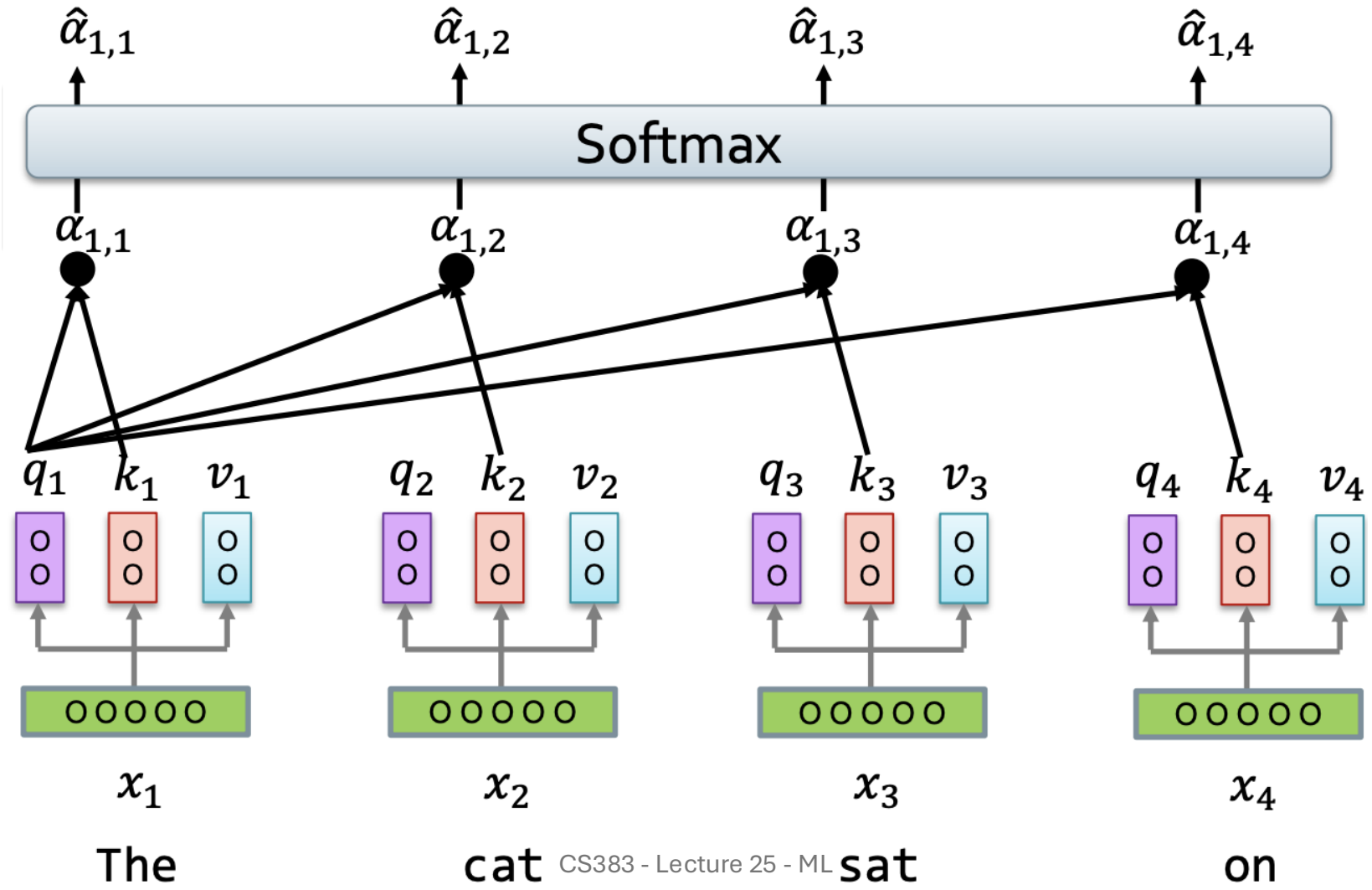
Self-attention



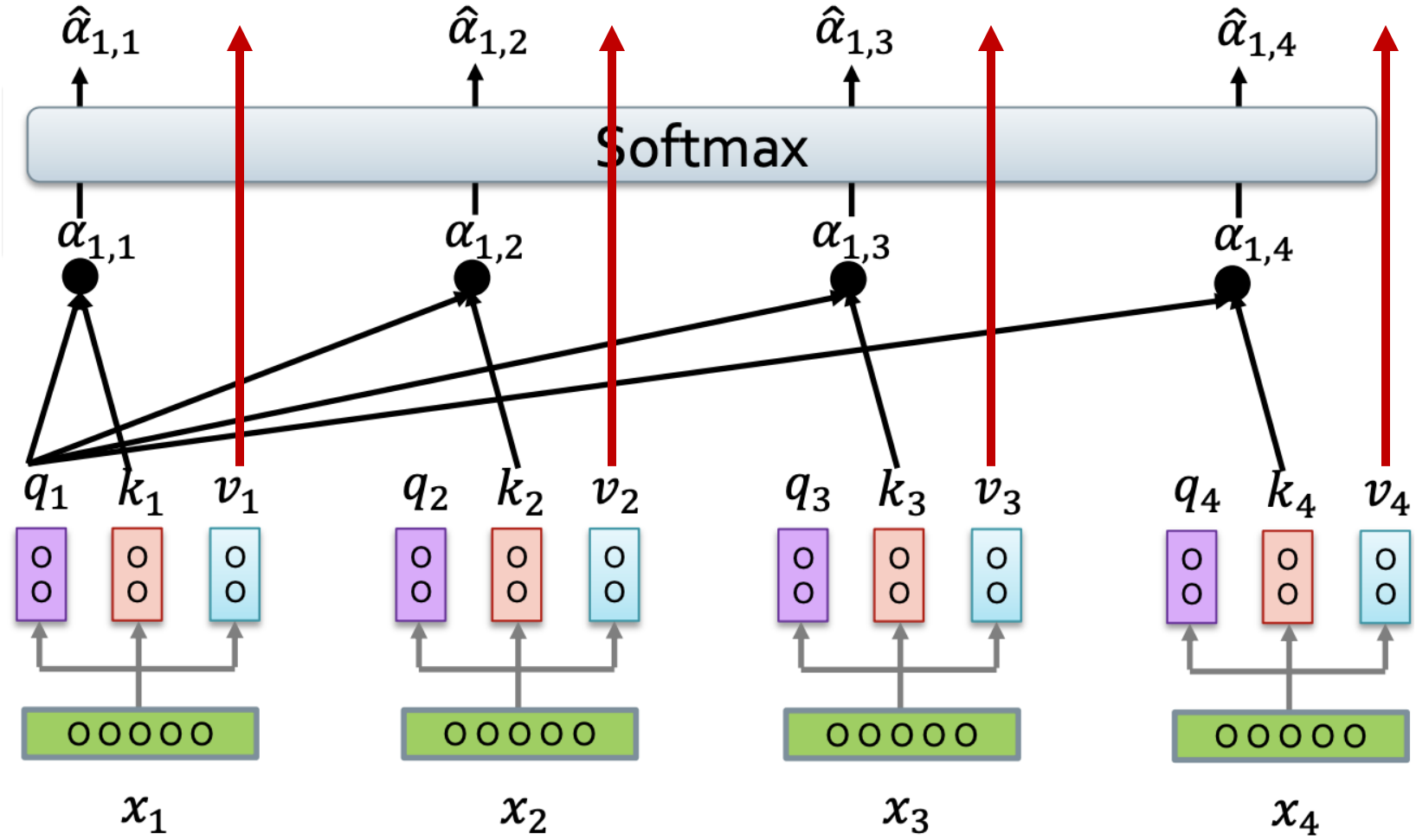
Self-attention



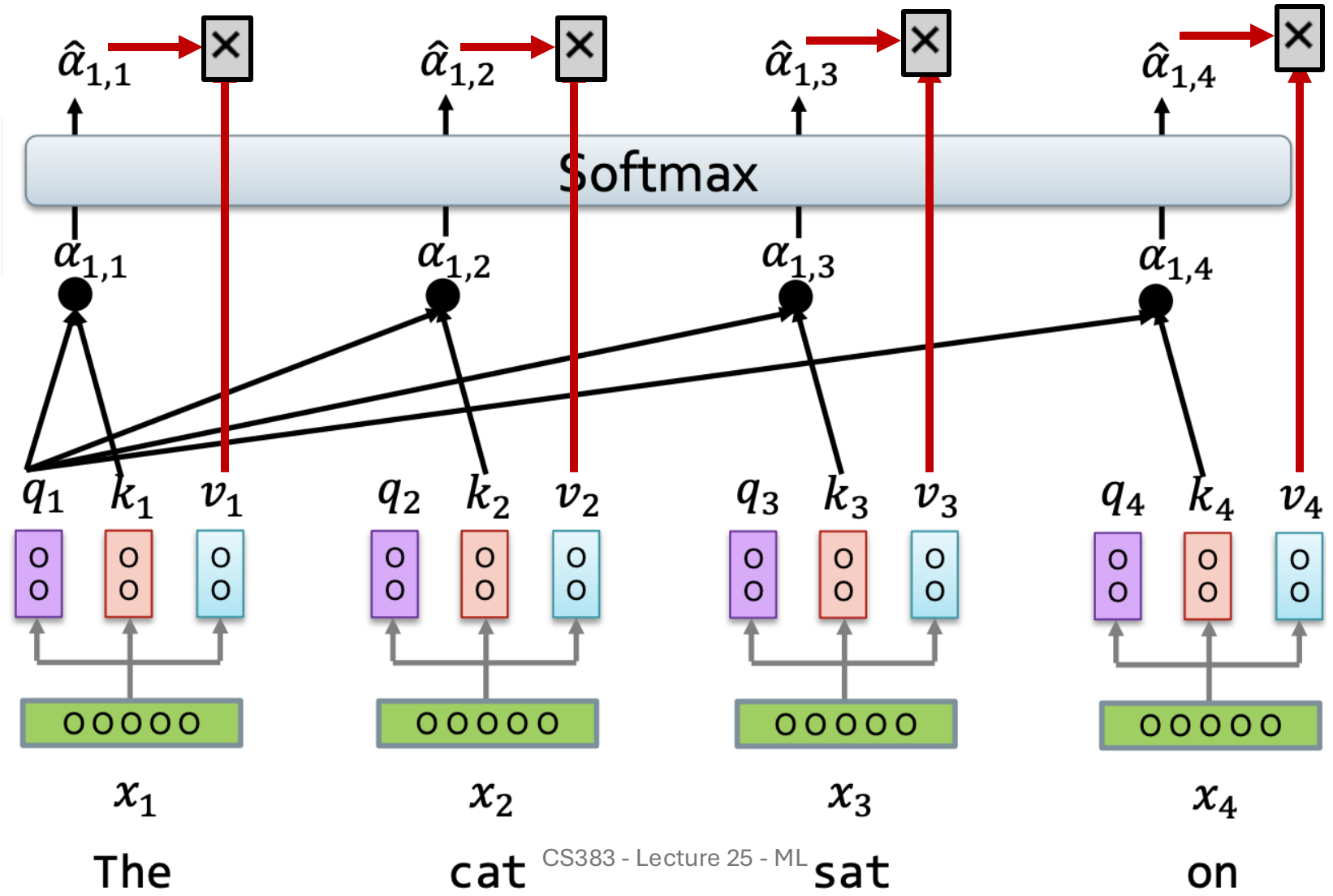
Self-attention



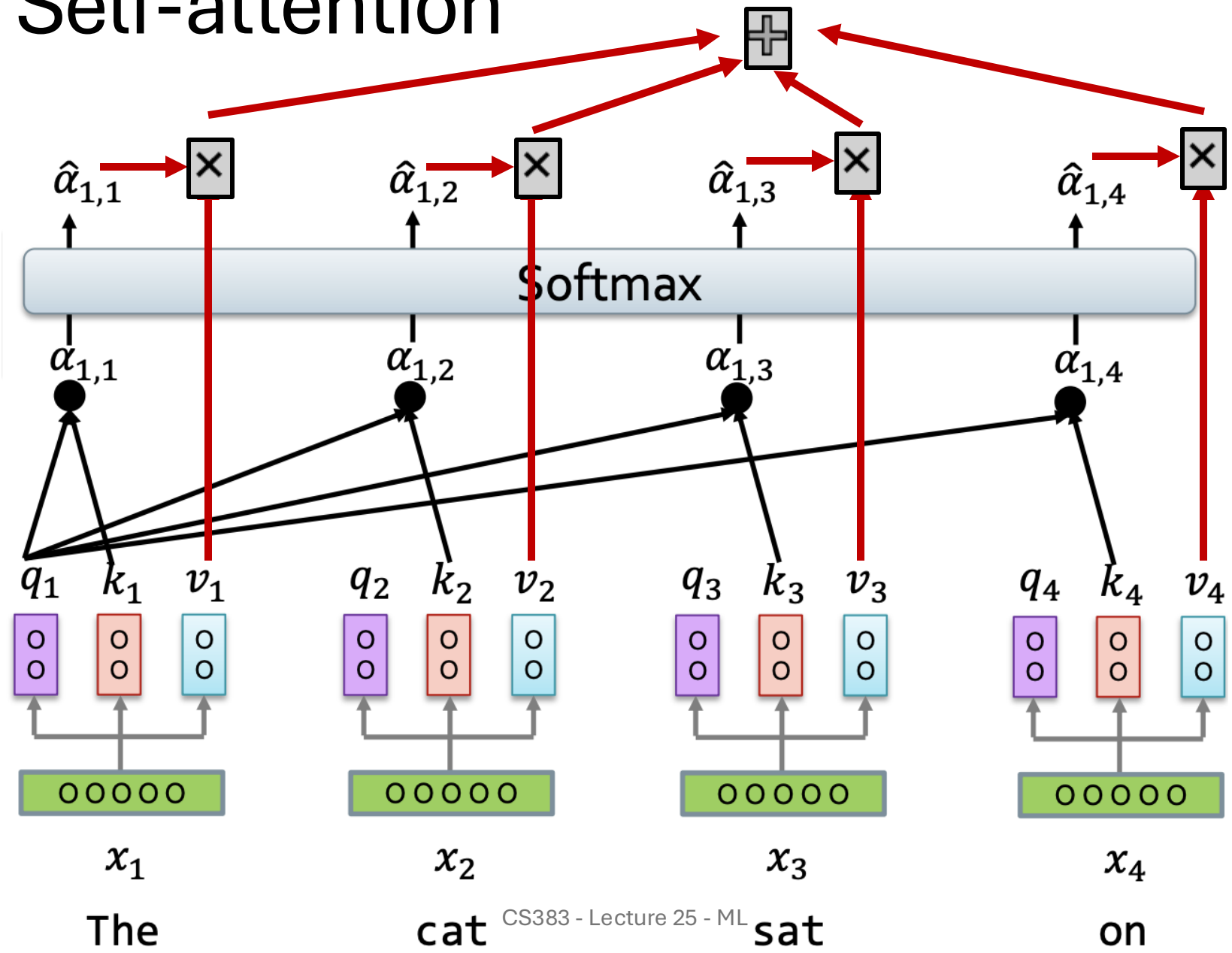
Self-attention



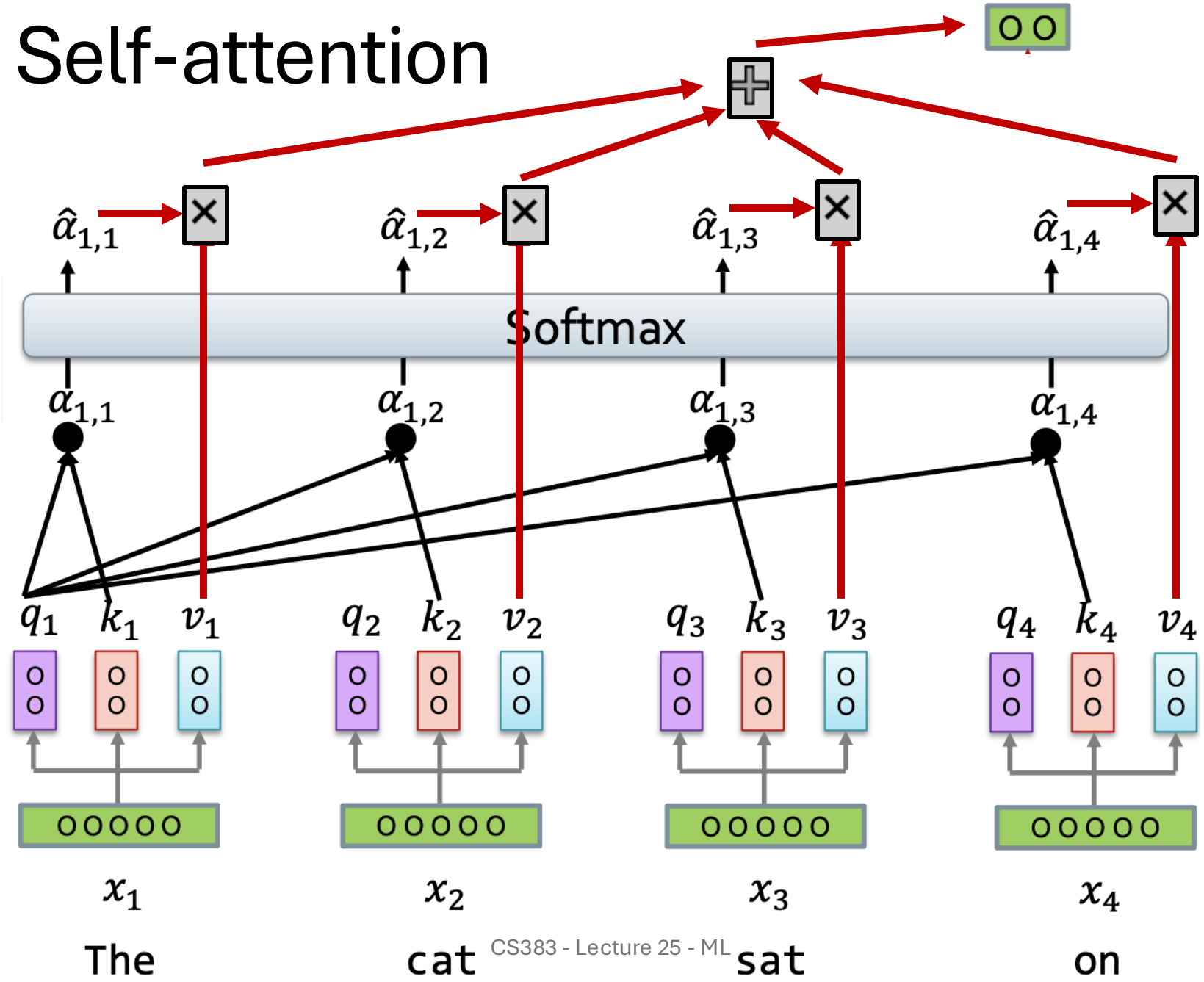
Self-attention



Self-attention

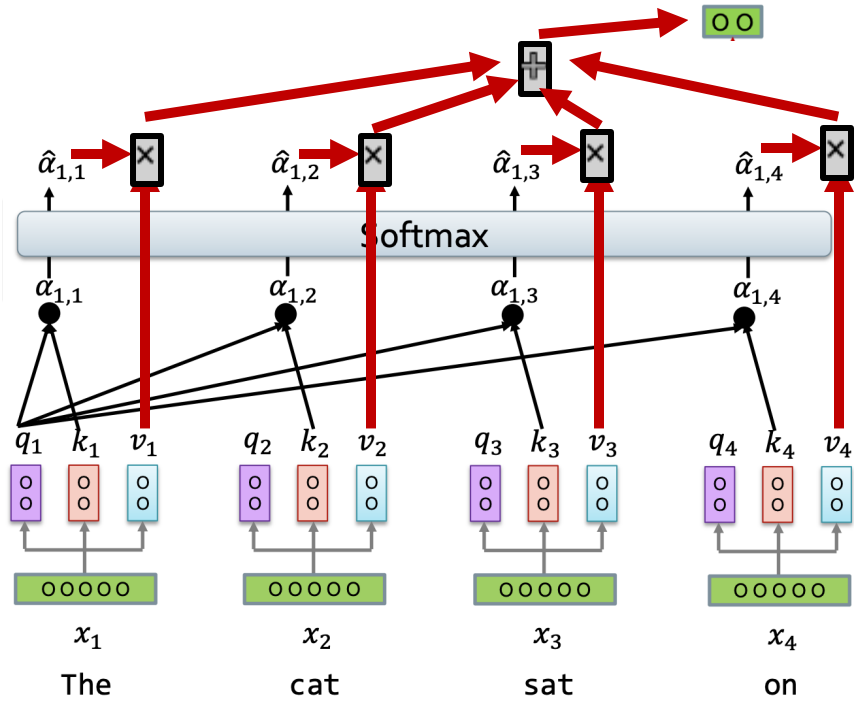


Self-attention



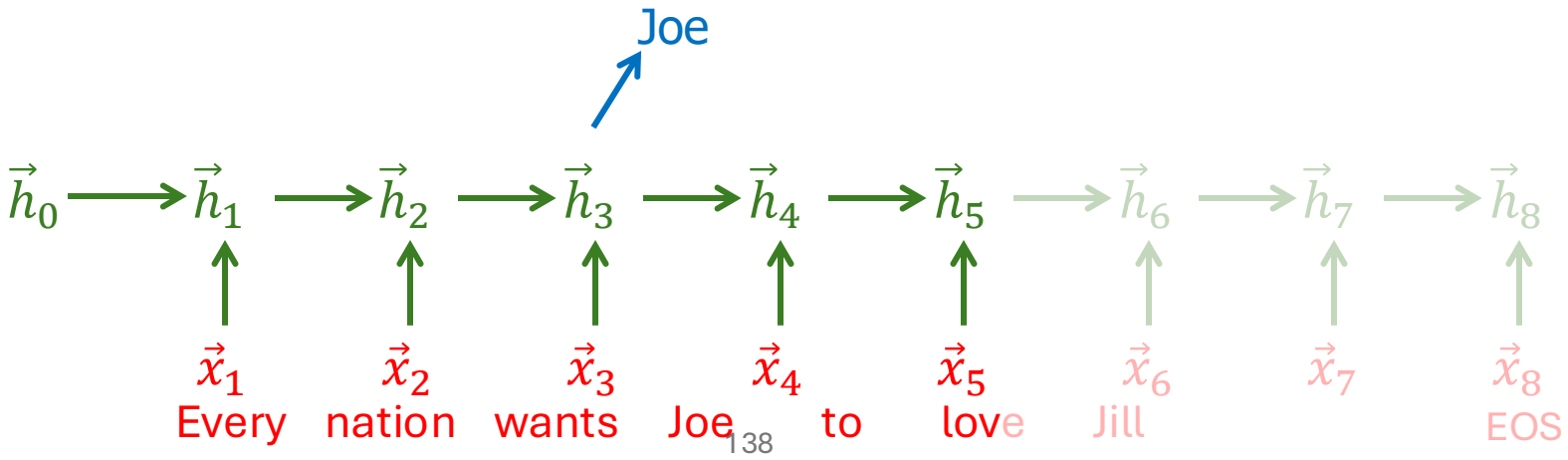
Self-attention

$$A = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

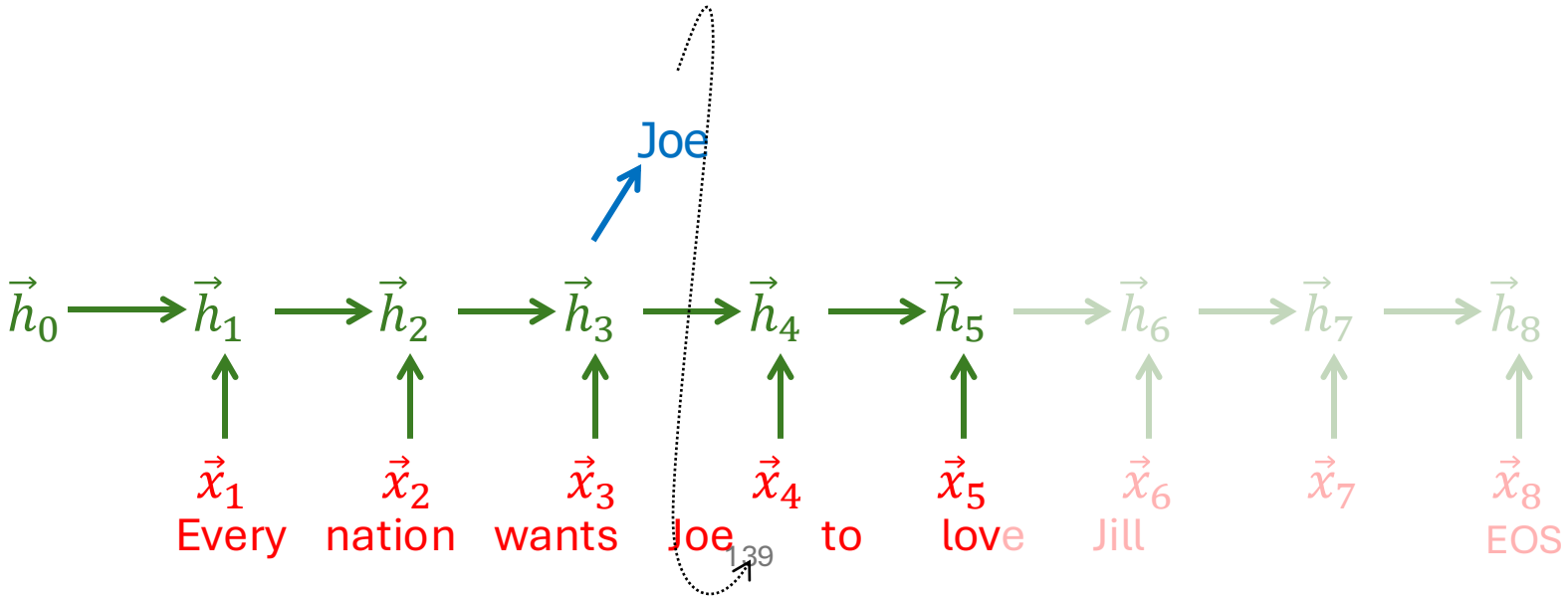


This is the main idea behind a **transformer**

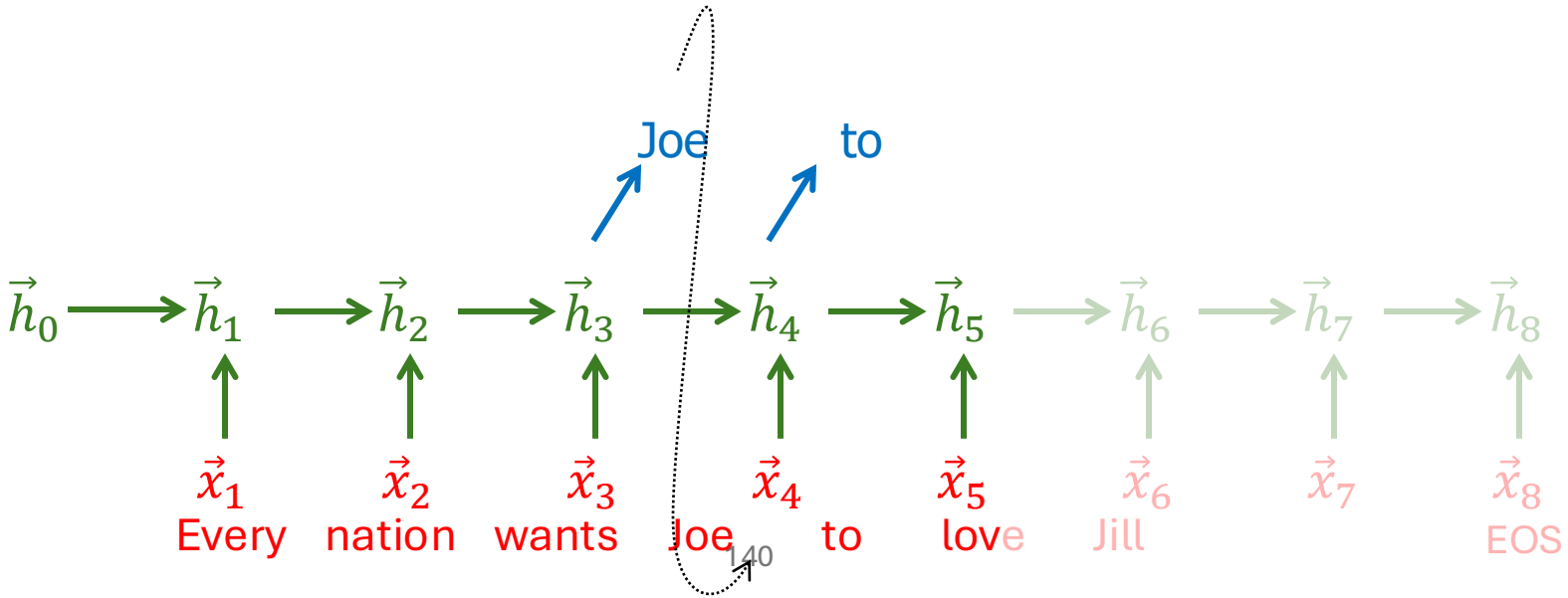
RNN LM


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

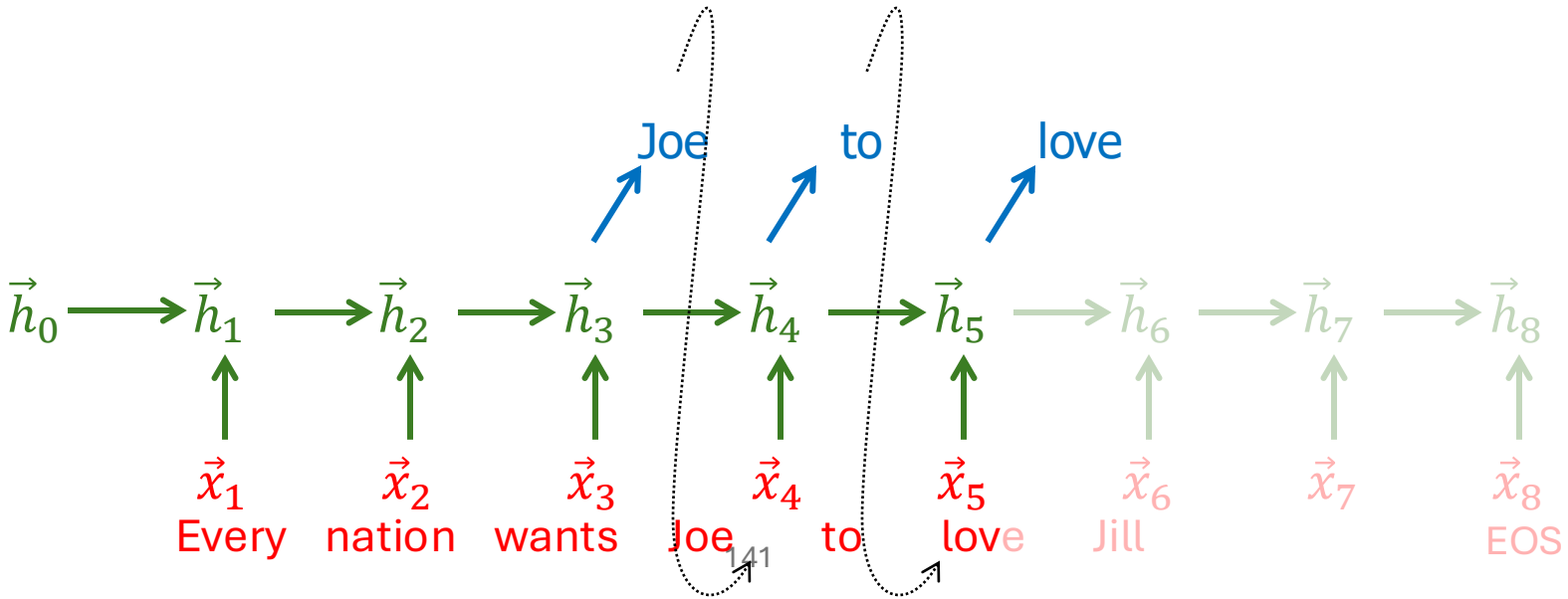
RNN LM


Joe
$\vec{h}_0$ $\vec{h}_1$ $\vec{h}_2$ $\vec{h}_3$ $\vec{h}_4$ $\vec{h}_5$ $\vec{h}_6$ $\vec{h}_7$ $\vec{h}_8$
$\vec{x}_1$ $\vec{x}_2$ $\vec{x}_3$ $\vec{x}_4$ $\vec{x}_5$ $\vec{x}_6$ $\vec{x}_7$ $\vec{x}_8$
Every nation wants Joe to love Jill EOS
139

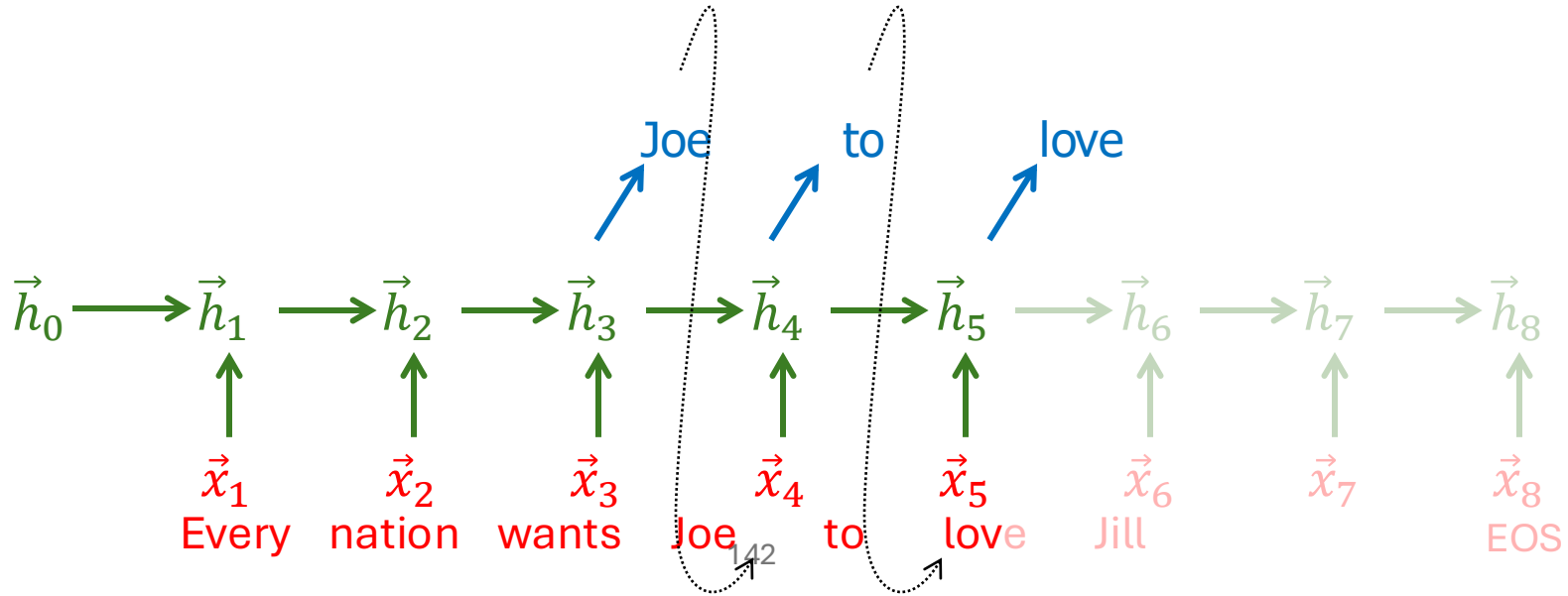
RNN LM


Joe
to
$\vec{h}_0$
$\vec{h}_1$
$\vec{h}_2$
$\vec{h}_3$
$\vec{h}_4$
$\vec{h}_5$
$\vec{h}_6$
$\vec{h}_7$
$\vec{h}_8$
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_3$
$\vec{x}_4$
$\vec{x}_5$
$\vec{x}_6$
$\vec{x}_7$
$\vec{x}_8$
Every
nation
wants
Joe
to
love
Jill
EOS
140

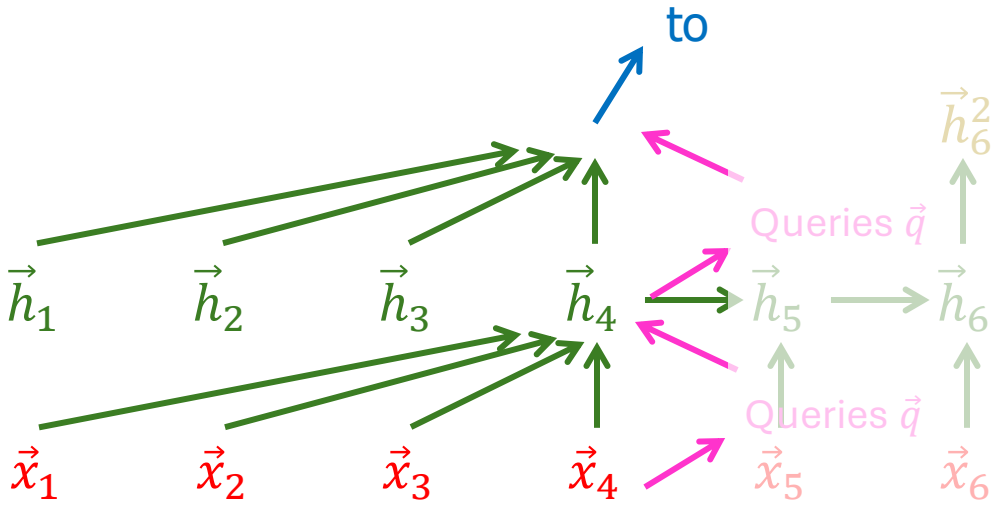
RNN LM


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

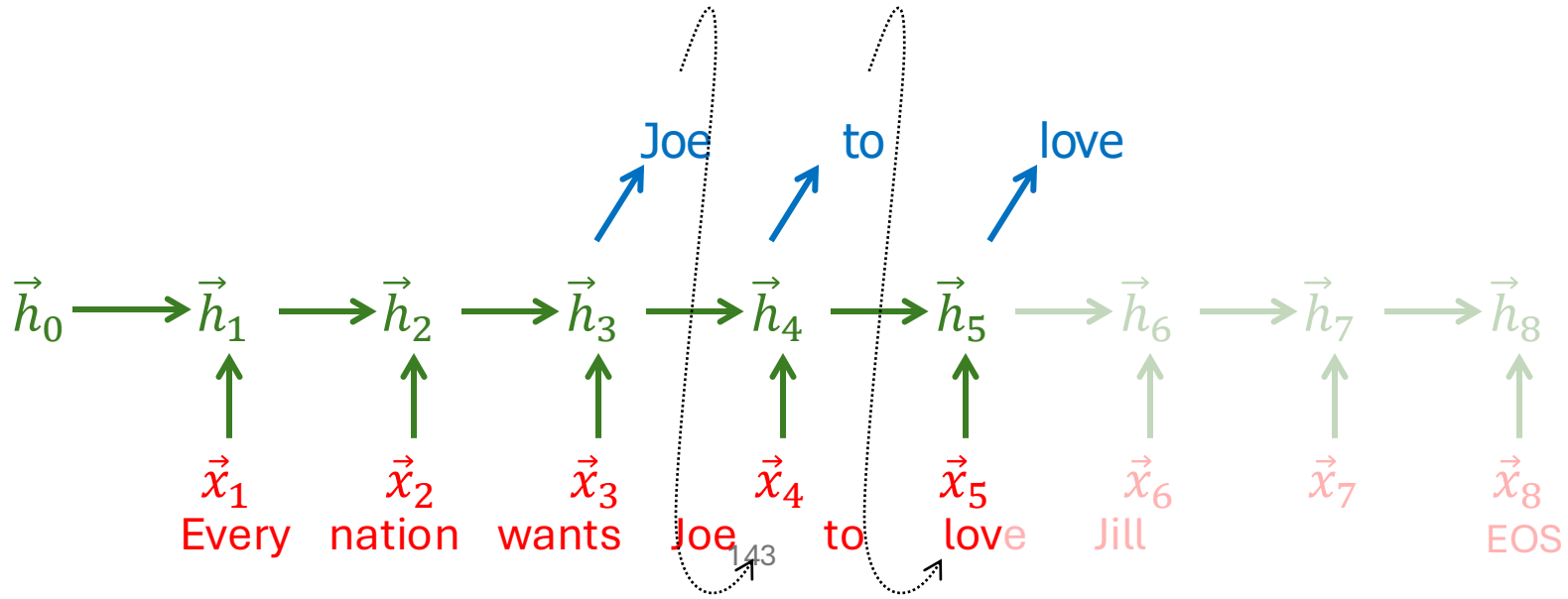
RNN LM



Transformer (self-attention) LM

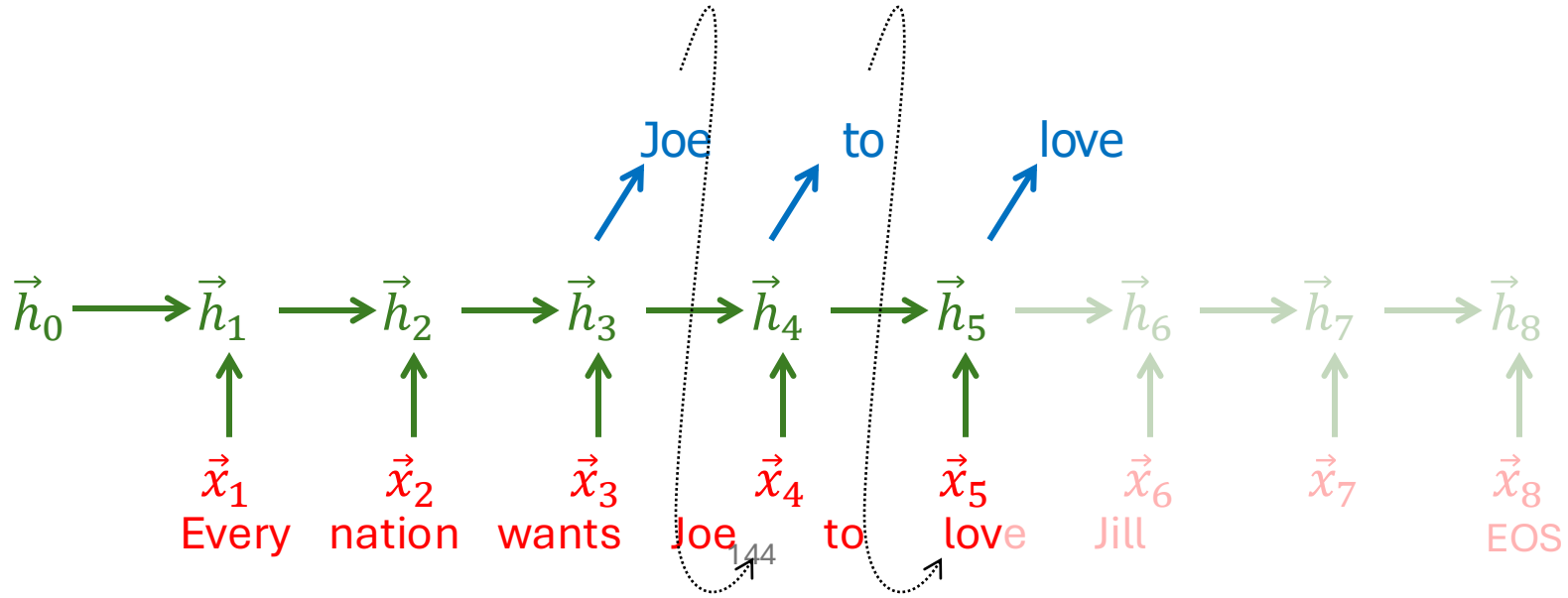


RNN LM

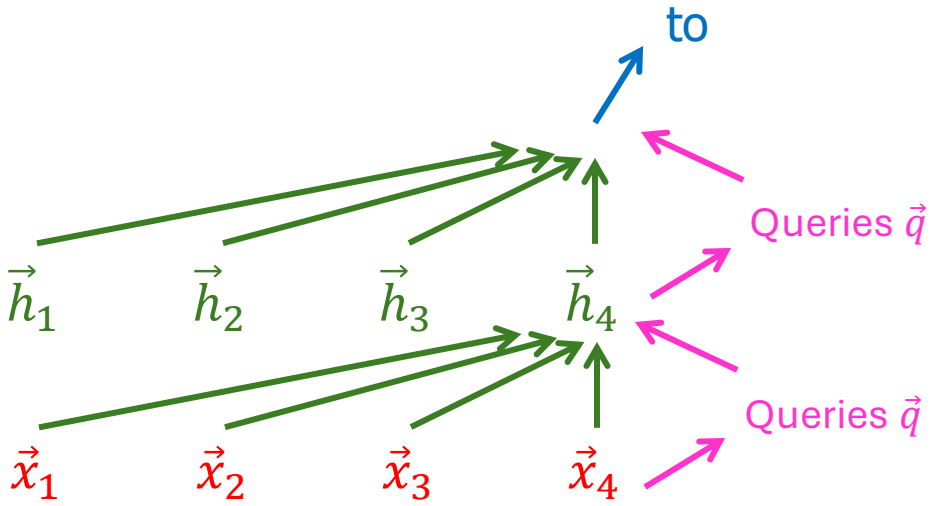


Transformer (self-attention) LM

RNN LM

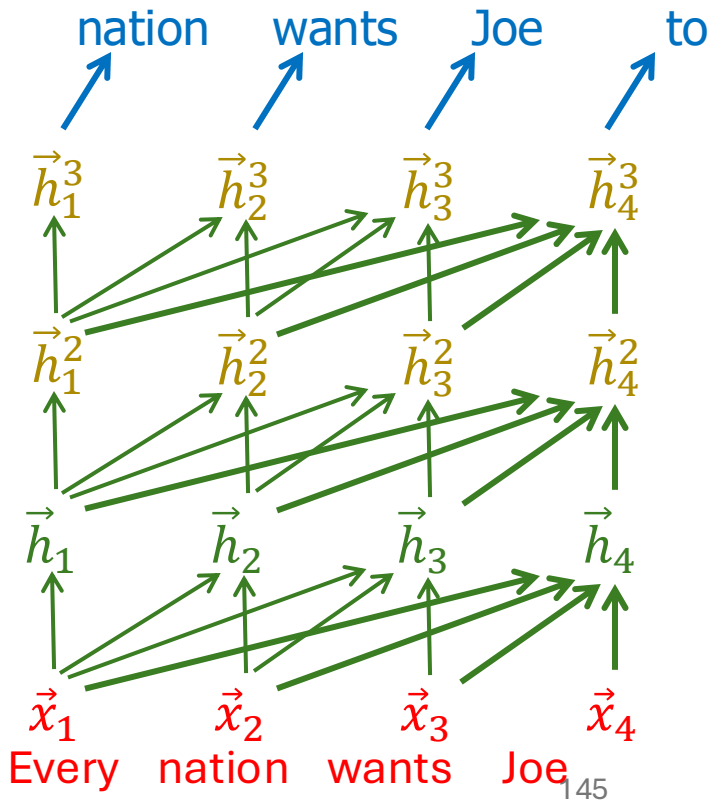


Transformer (self-attention) LM



Training can be parallelized

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



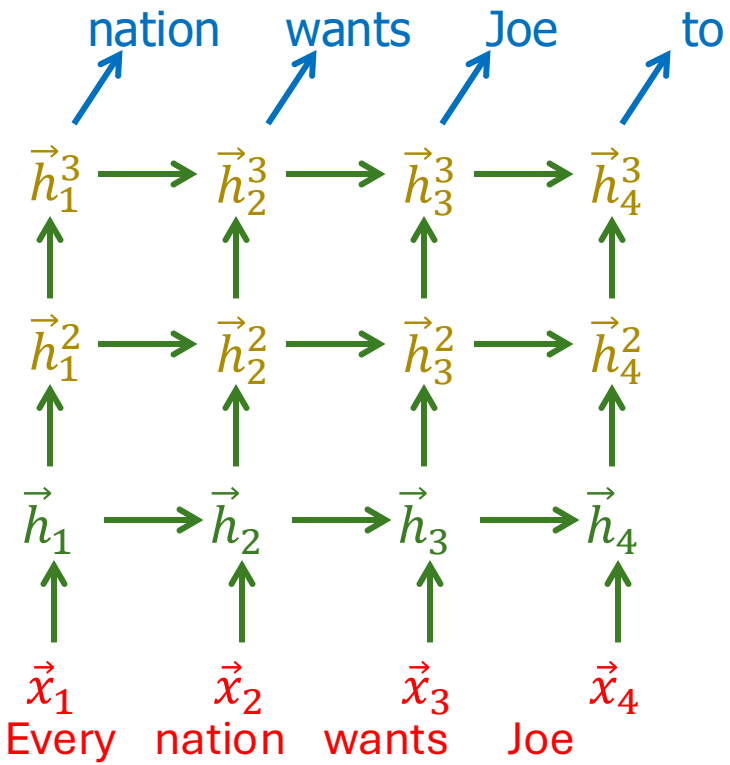
(oops, to predict the very first word, we needed $\vec{x}_0 =< s >$!
It's missing from our diagrams.)

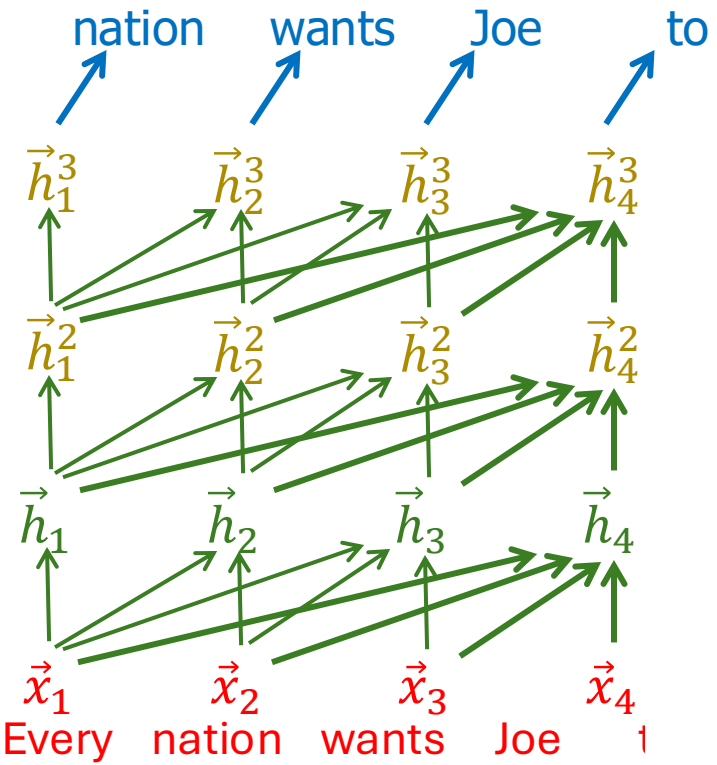
RNN vs. Transformer

Training, on GPU, per layer ↓

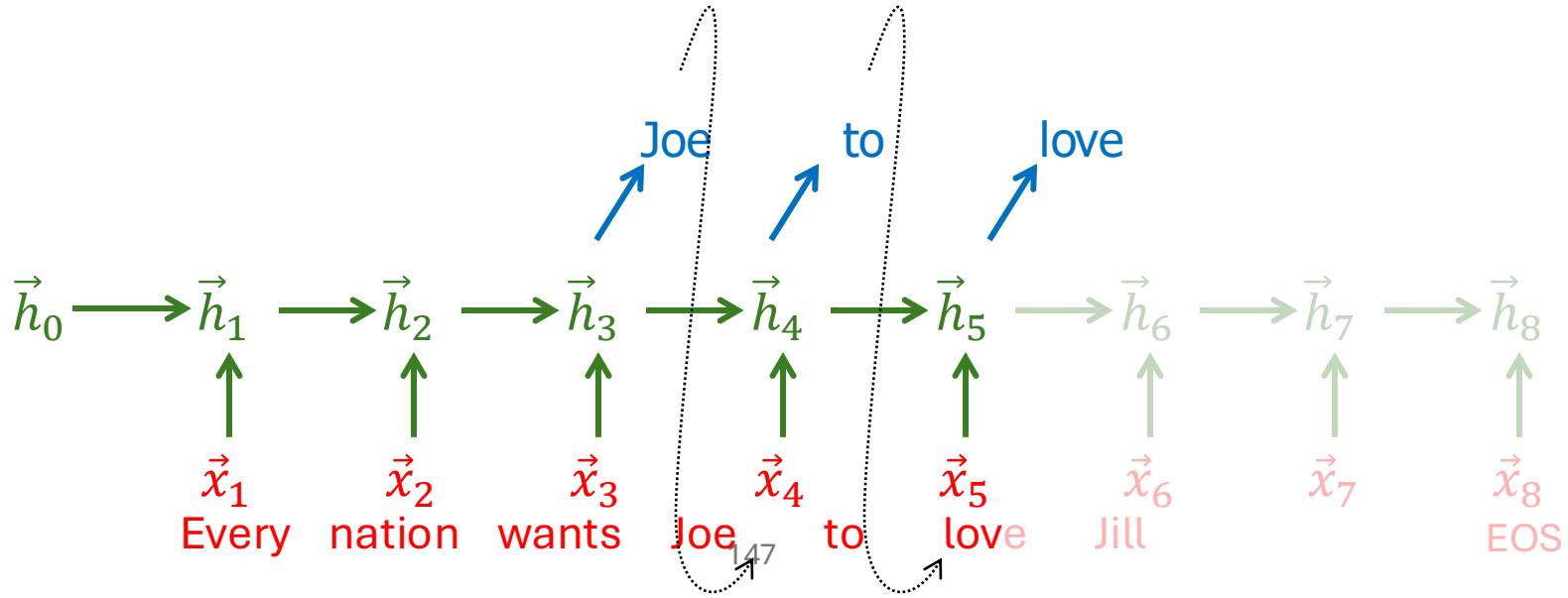
Computations: ☺ O(n)
serial steps: ☹ O(n) due to →

☹ $O(n^2)$
☺ O(1): all ↗↑ in parallel
+ O(log n) to sum n inputs

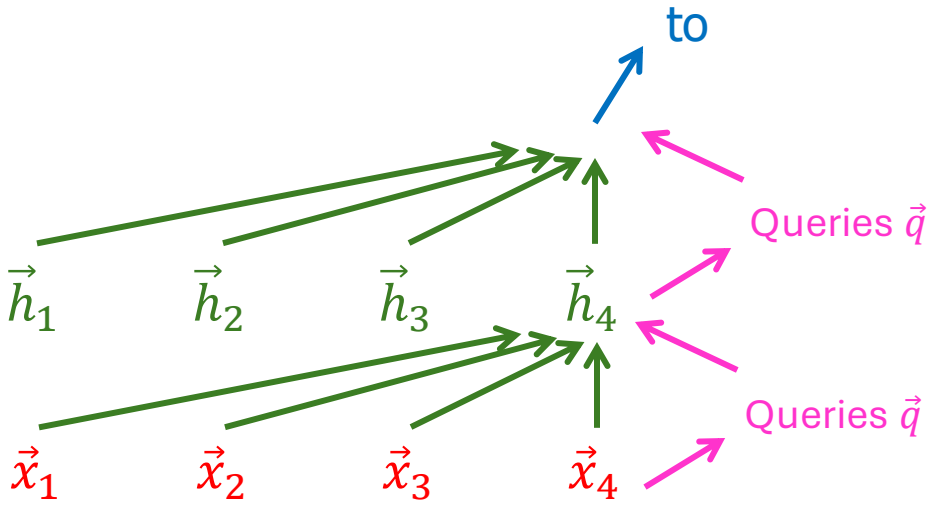




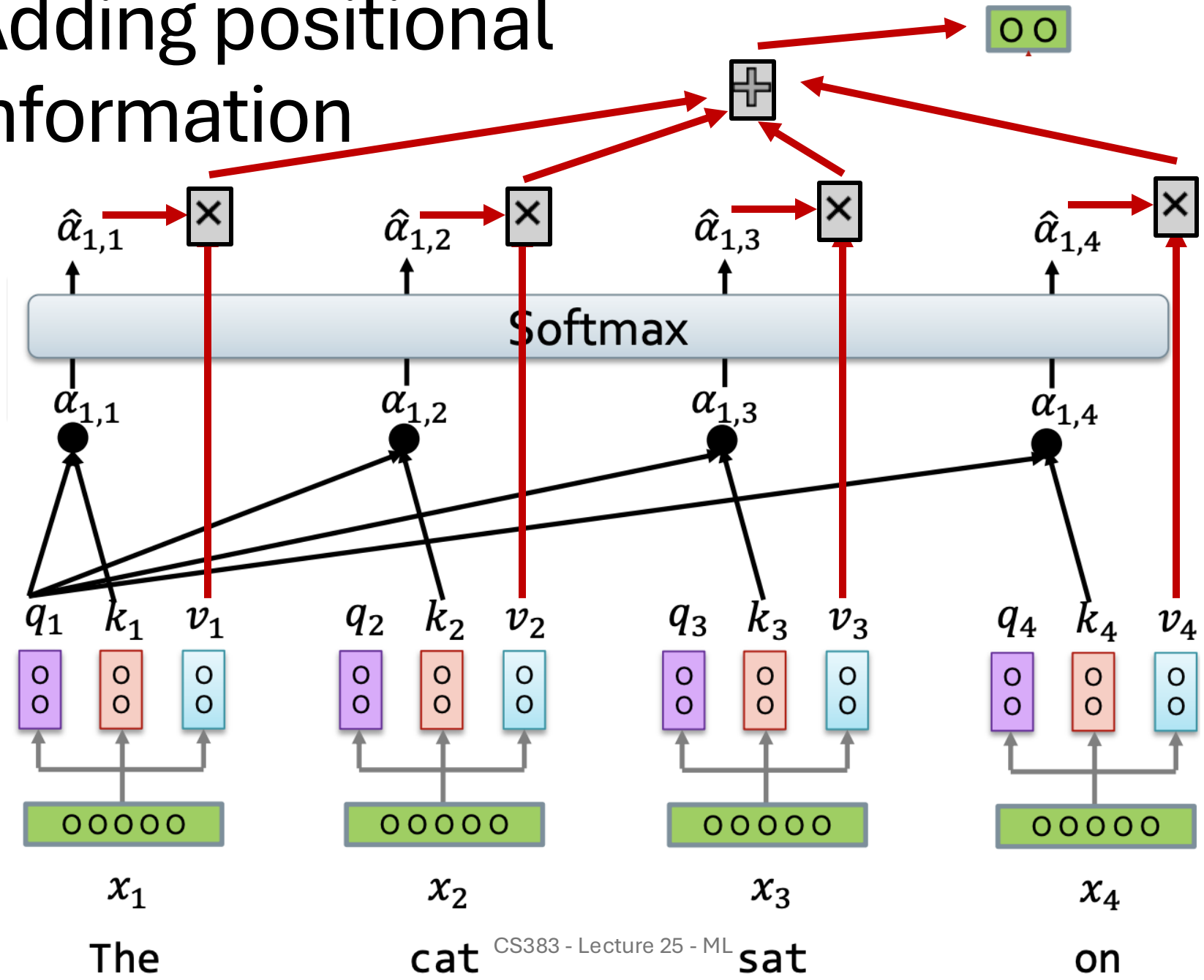
RNN LM



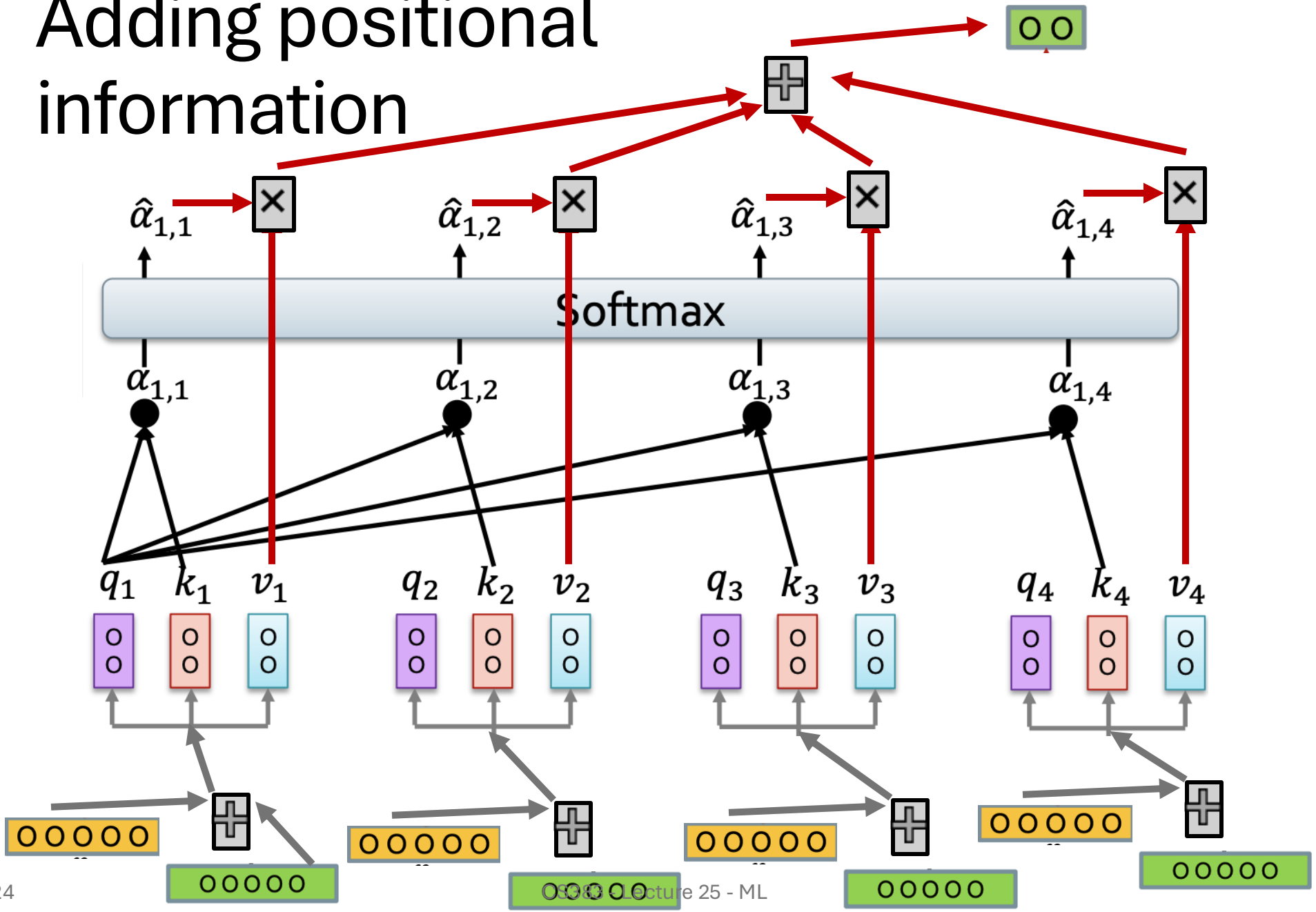
Transformer (self-attention) LM



Adding positional information

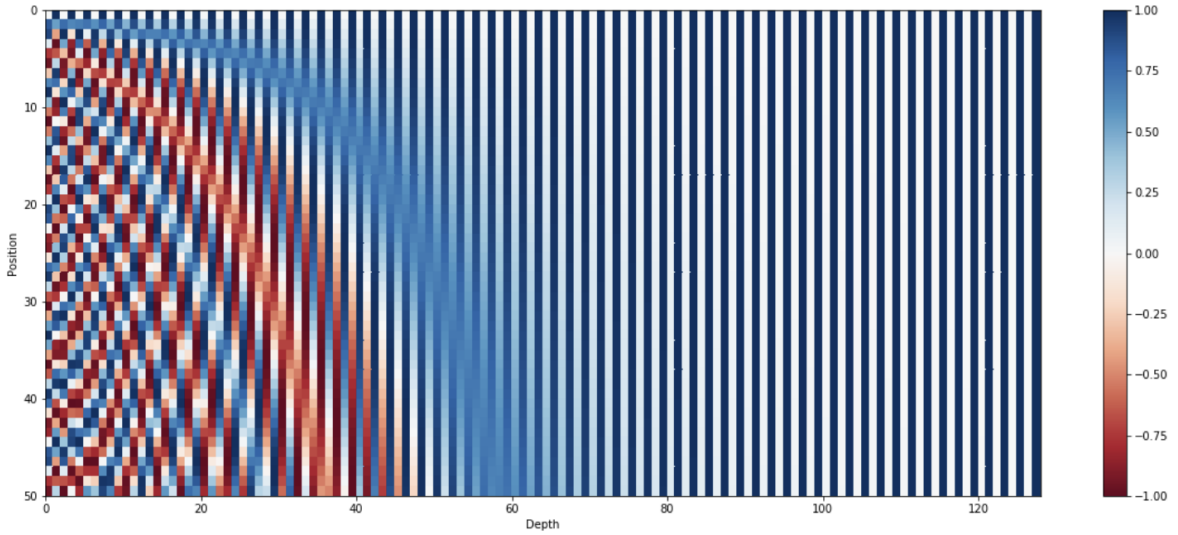


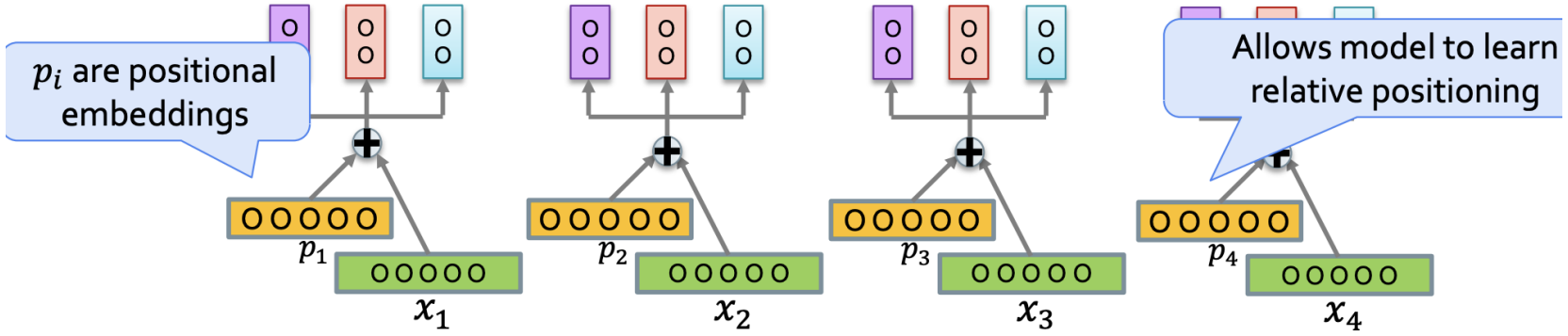
Adding positional information



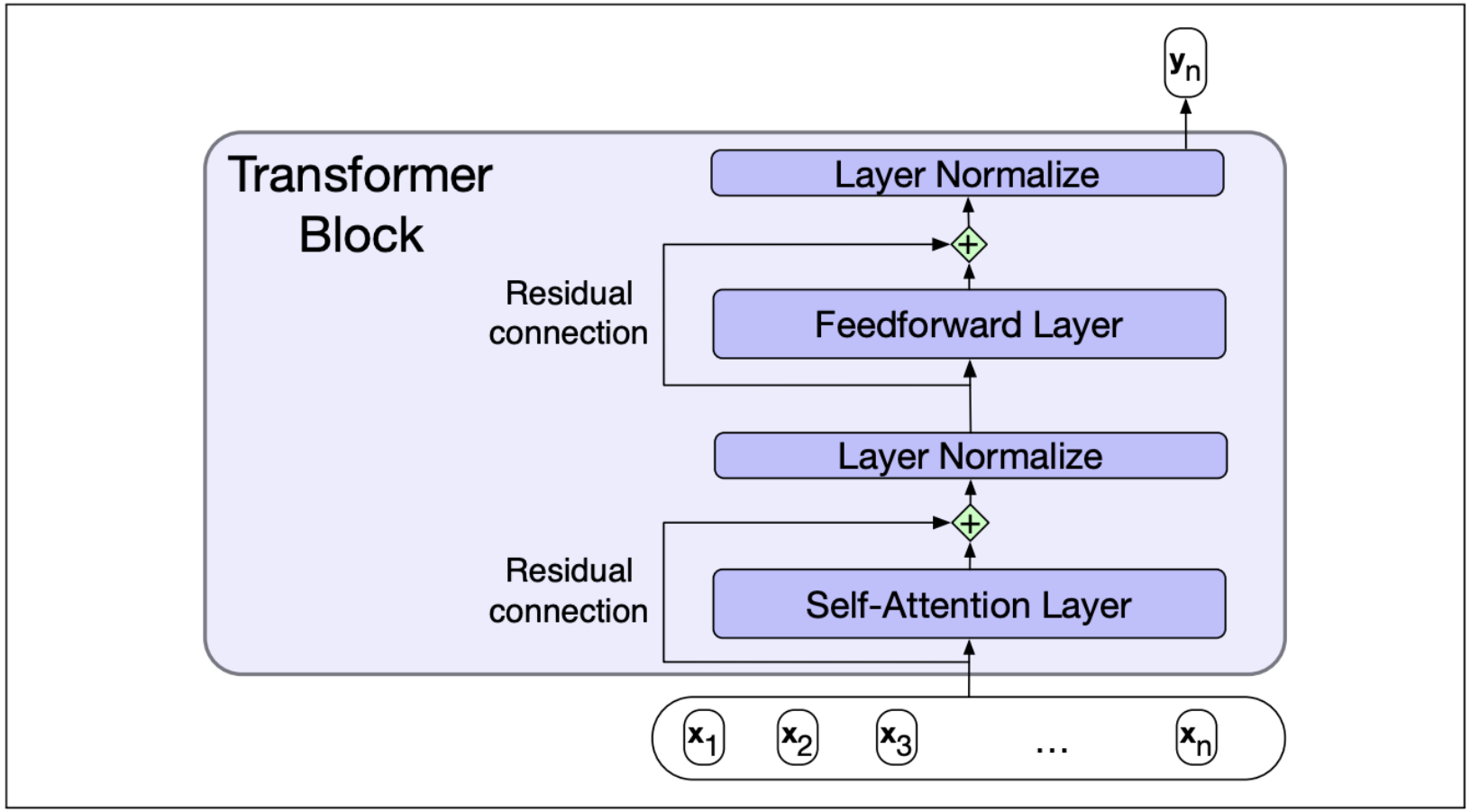
Adding positional information

An approach:
Sine/Cosine encoding





Transformer block



Transformer block

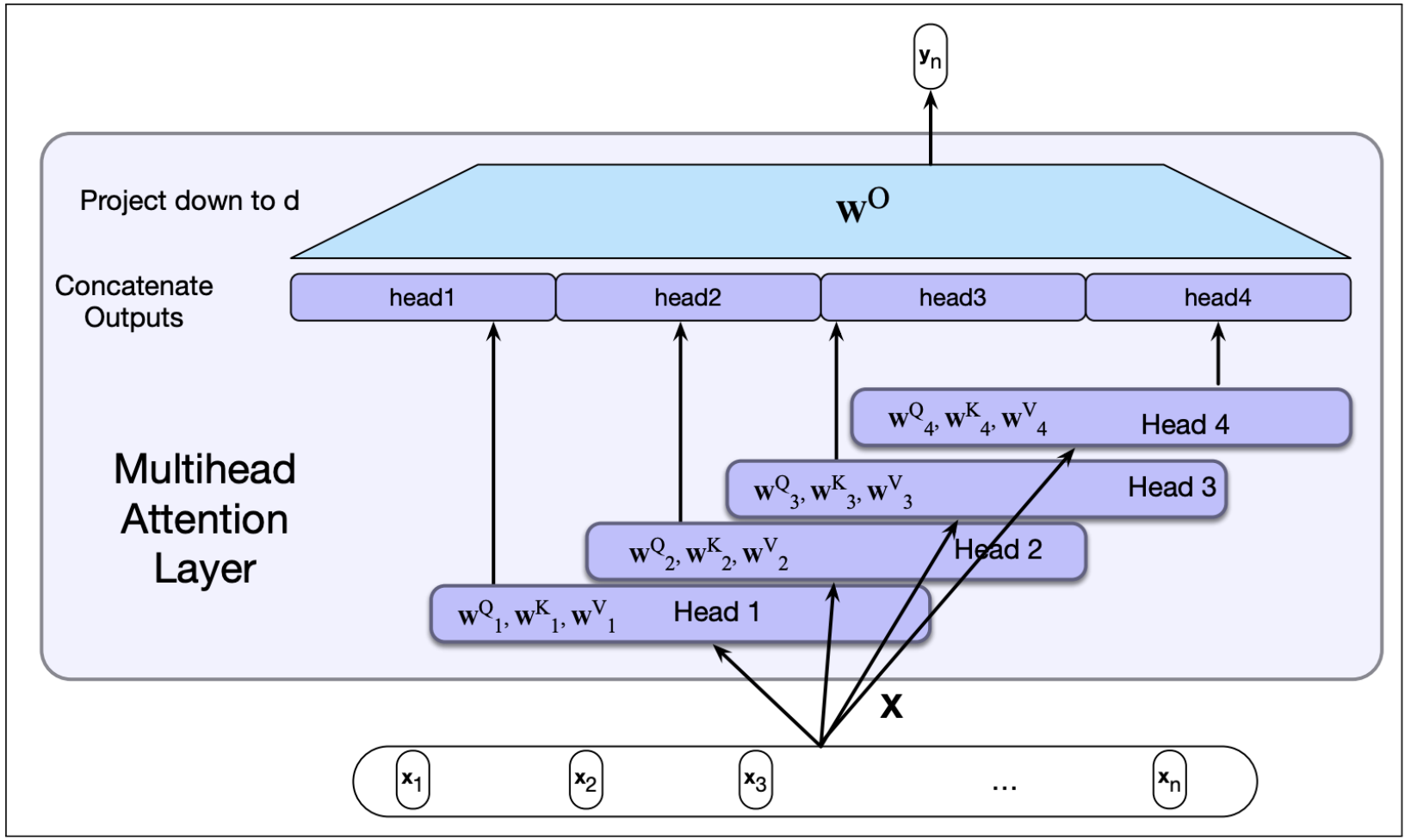
Residual connection:

- Passes information from a lower layer to a higher layer directly (w/out going through intermediate layers)

Layer normalization

- Ensures the values in a layer are in an appropriate range
- Based on normalization/z-scores in statistics (we'll cover normalization later this semester)

Multi-head attention



Transformers as LM

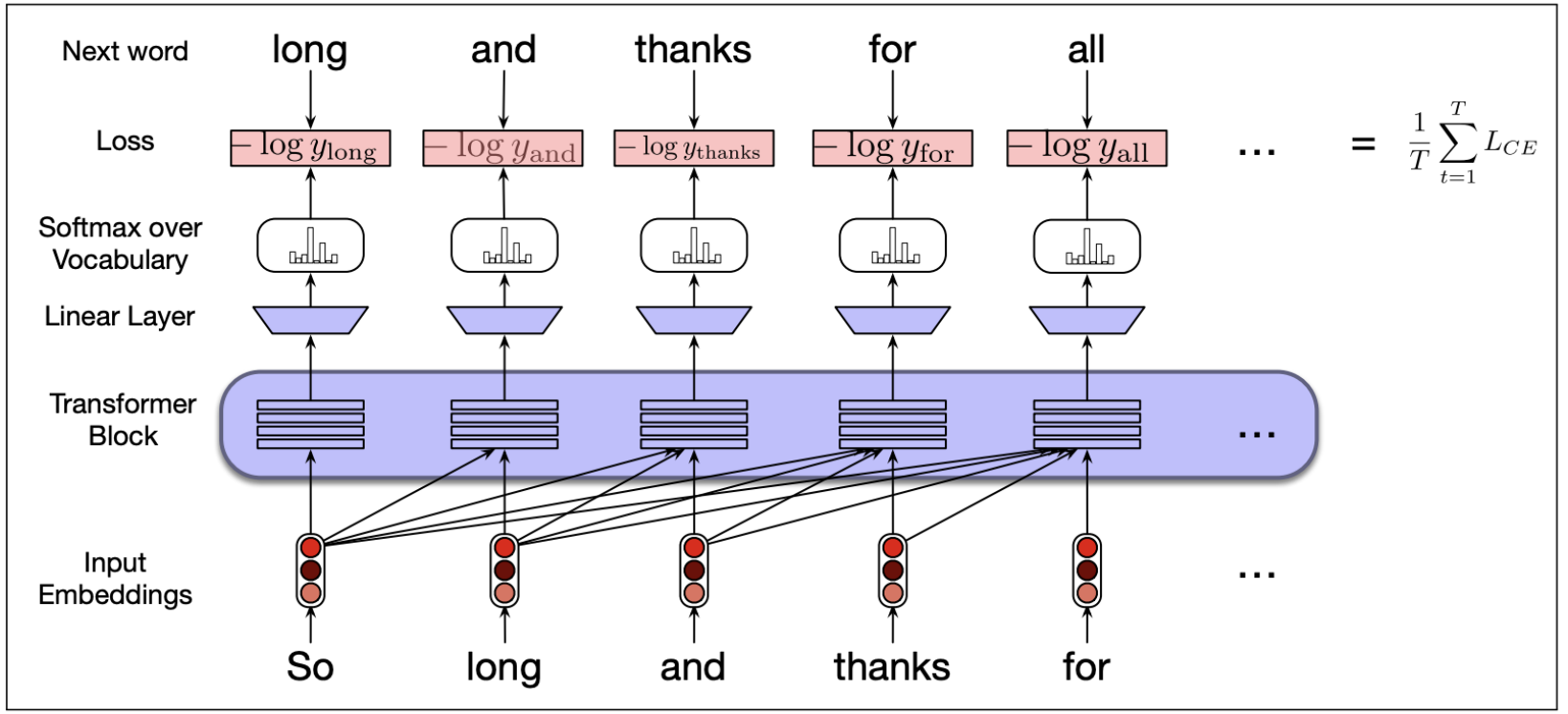


Figure 10.7 Training a transformer as a language model.

Training tansformers

parameters in transformer >> # parameters in LSTM

So, training requires a lot of data

We can pre-train a transformer, and then use it as a sentence-representation/feature extracter

Led to State-of-the-art models