Deep Learning Architectures for Sequence Processing

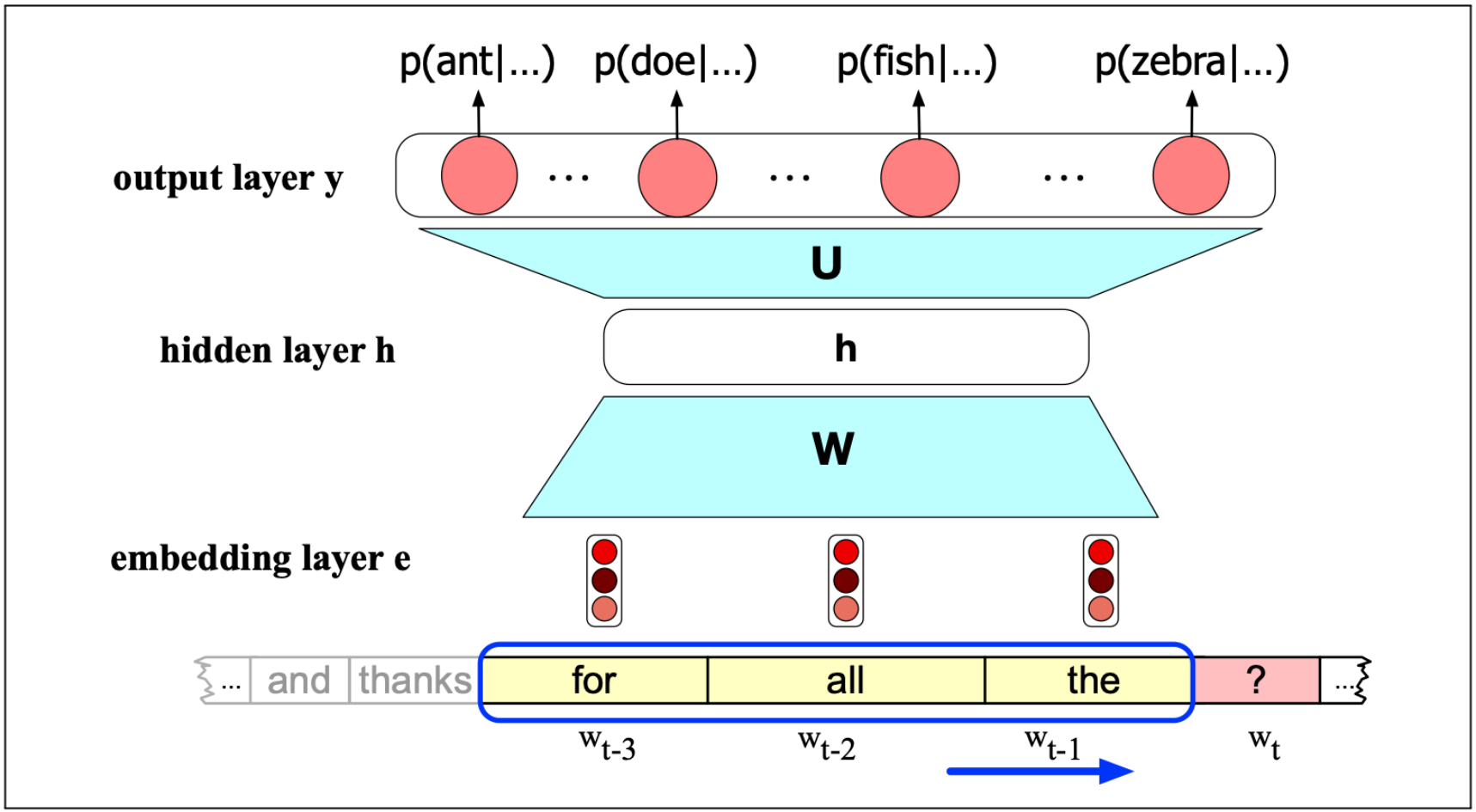
Erhard Hinrichs

Seminar für Sprachwissenschaft
Eberhard-Karls Universität Tübingen

Three Alternatives for Dealing with Sequences of Language Data (1)

- Presentation of an input sequence to a feed-forward network (FFN) as a sliding window of subsequences of fixed length
- Advancing the sliding window by one token at a time and presenting each subsequence as separate inputs
- The tokens in each input subsequence are projected to their corresponding embeddings, and the embeddings are concatenated in the projection layer.
- Disadvantages of this approach
  - Sliding windows provides only limited context.
  - No context information is preserved beyond the fixed context window.

Language Modeling with a FFN



Language Models Revisited (see J&N, ch. 3)

- $P(\textit{fish}|\textit{Thanks for all the})$
- More generally: $P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{<i})$ (Chain rule)

Perplexity

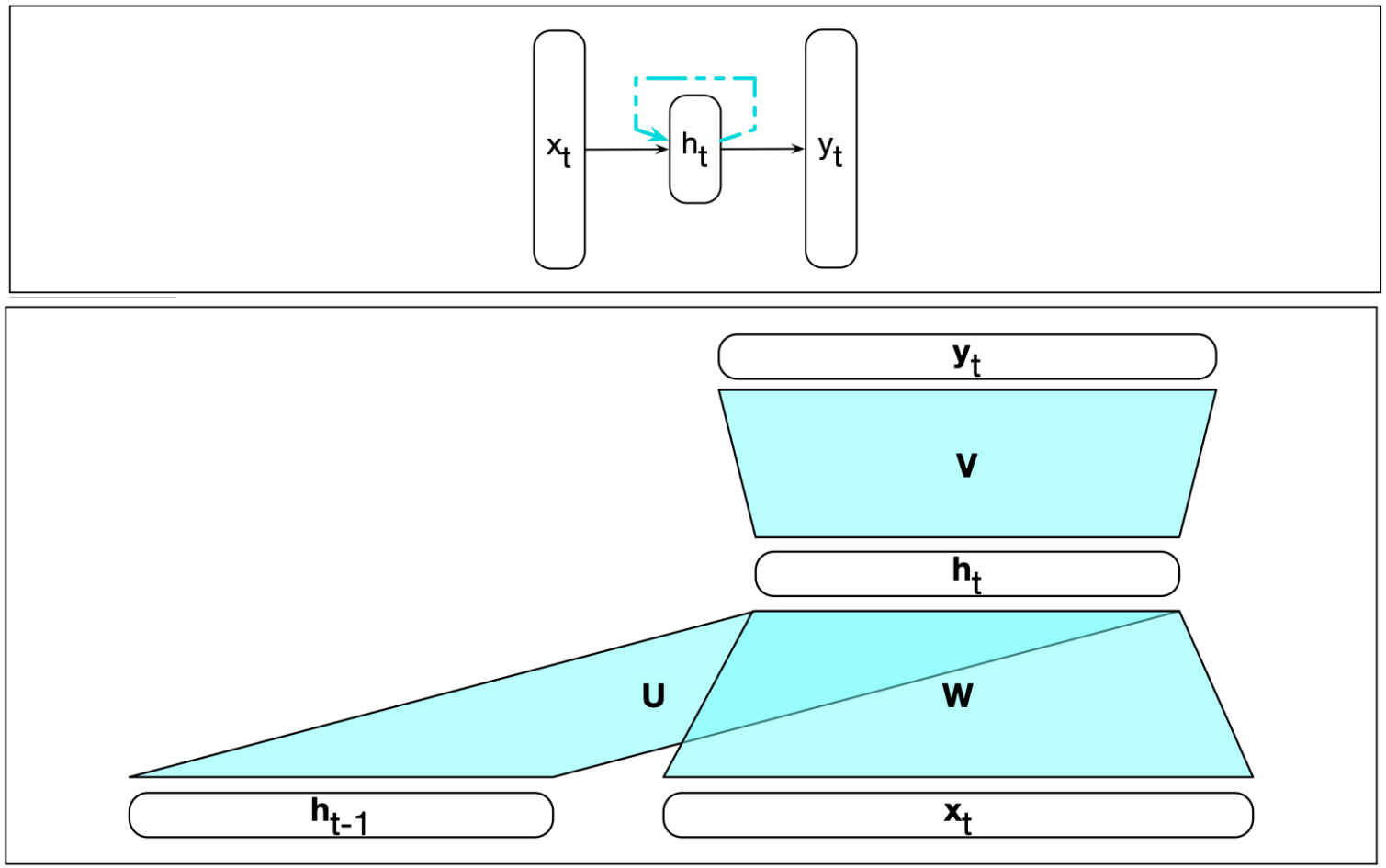
$$PP_\theta(w_{1:n}) = P_\theta(w_{1:n})^{-\frac{1}{n}} = \sqrt[n]{\frac{1}{P_\theta(w_{1:n})}} \tag{1}$$

$$PP_\theta(w_{1:n}) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P_\theta(w_i | w_{1:i-1})}} \tag{2}$$

Three Alternatives for Dealing with Sequences of Language Data (2)

- Presenting individual input tokens to a recurrent neural network (RNN)
  - Enriching the hidden state of each input with output representations of previous tokens in the same sequence.
- Disadvantages of this approach
  - Difficulty of modelling long-distance dependencies among tokens
  - Exploding or vanishing gradients.

Recurrent Neural Networks



Inference in RNNs

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \tag{3}$$

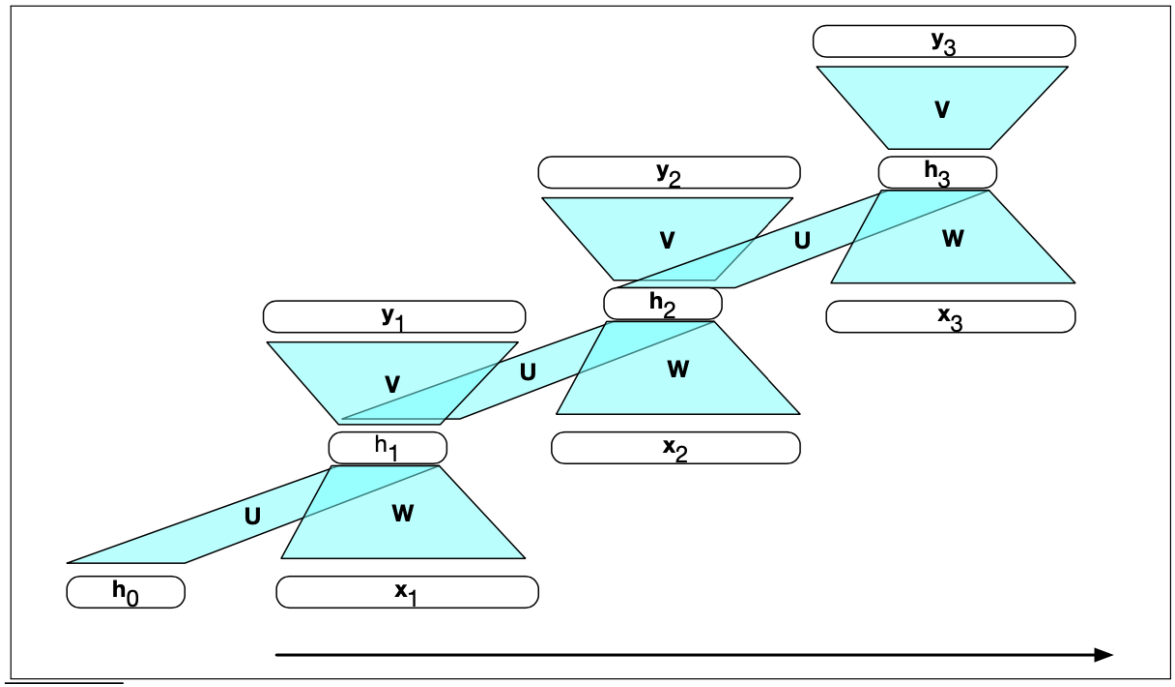
$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t) \tag{4}$$

$$\mathbf{y}_t = \mathrm{softmax}(\mathbf{V}\mathbf{h}_t) \tag{5}$$

Forward Inference in RNNs

function FORWARDRNN(**x**, *network*) **returns** output sequence **y**

$\mathbf{h}^0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(**x**) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y



Comparing FFN and RNN Language Models

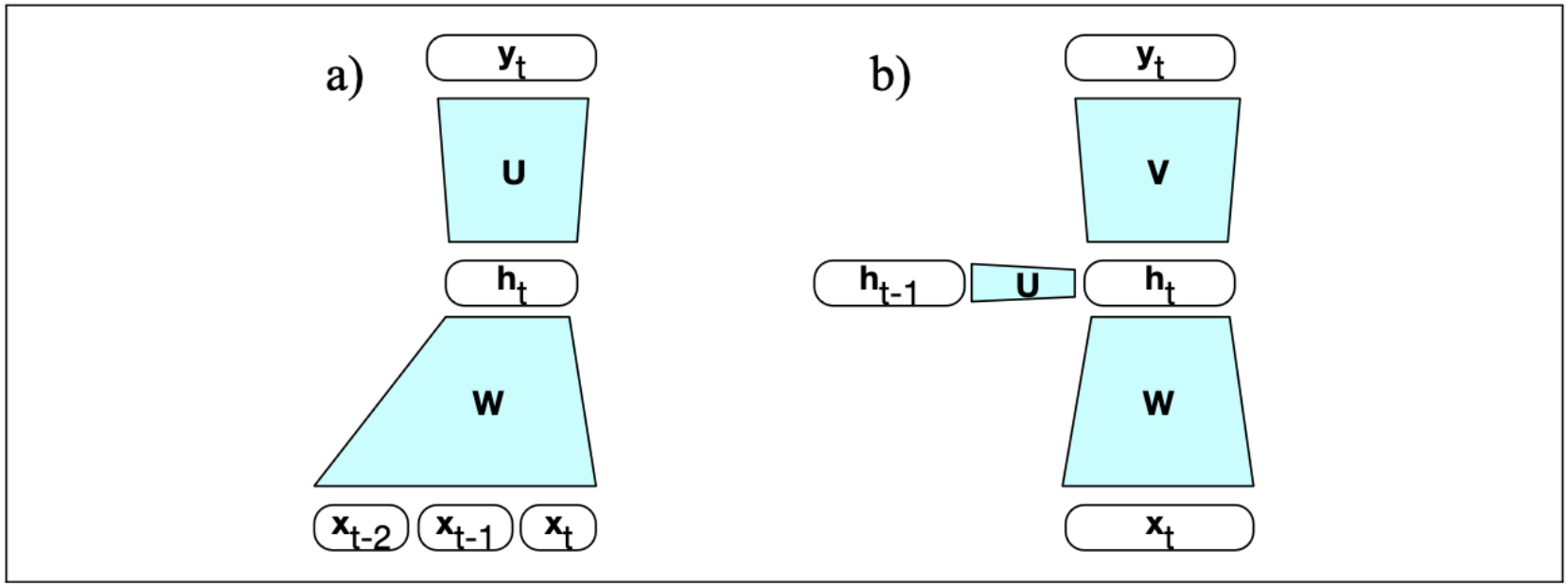


Figure 9.5 Simplified sketch of (a) a feedforward neural language model versus (b) an RNN language model moving through a text.

RNNs as Language Models

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{6}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{7}$$

$$\mathbf{y}_t = \mathrm{softmax}(\mathbf{V}\mathbf{h}_t) \tag{8}$$

RNNs as Language Models

$$P(w_{t+1} = i | w_1, \ldots, w_t) = \mathbf{y}_t[i] \tag{9}$$

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \tag{10}$$

$$= \prod_{i=1}^{n} \mathbf{y}_i[w_i] \tag{11}$$

RNNs as Language Models

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] log\ \hat{\mathbf{y}}_t[w] \tag{12}$$

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -log\ \hat{\mathbf{y}}_t[w_{t+1}] \tag{13}$$

Training a RNN Language Model

- Get a big corpus of text which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ for *every step t*.
  - i.e. predict probability dist of *every word*, given words so far

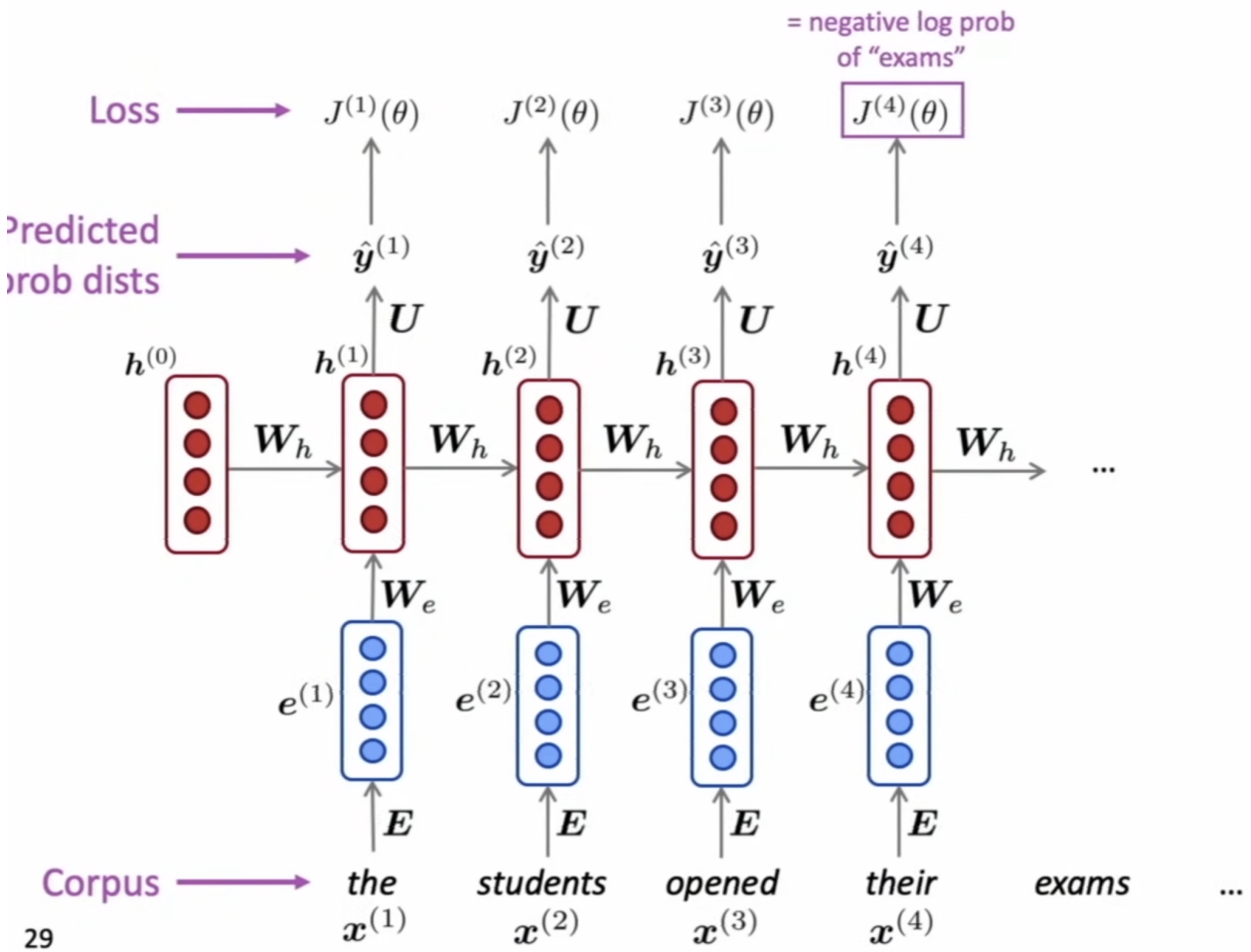
- Loss function on step *t* is cross-entropy between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

- Average this to get overall loss for entire training set:

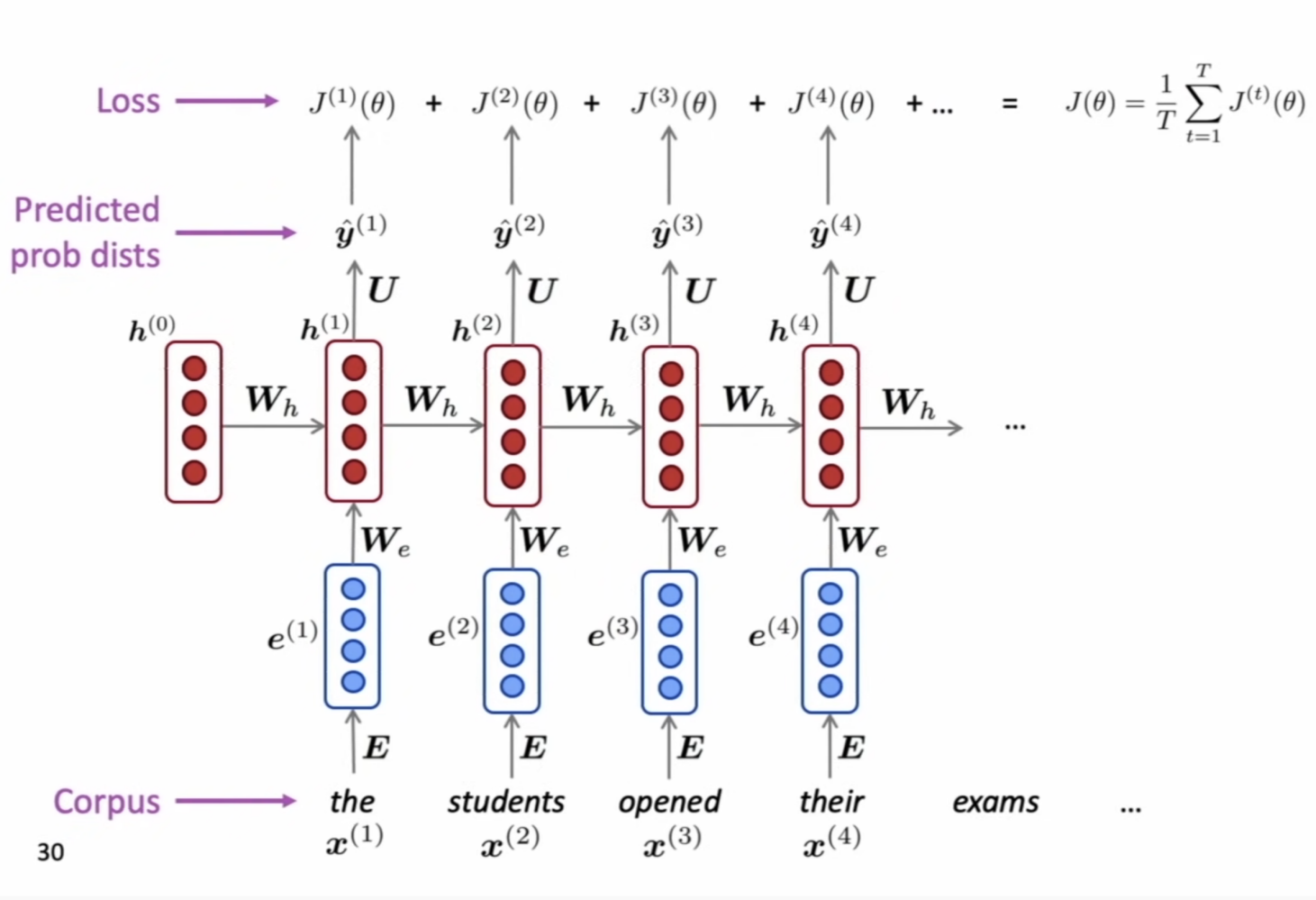
$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

Training a RNN Language Model



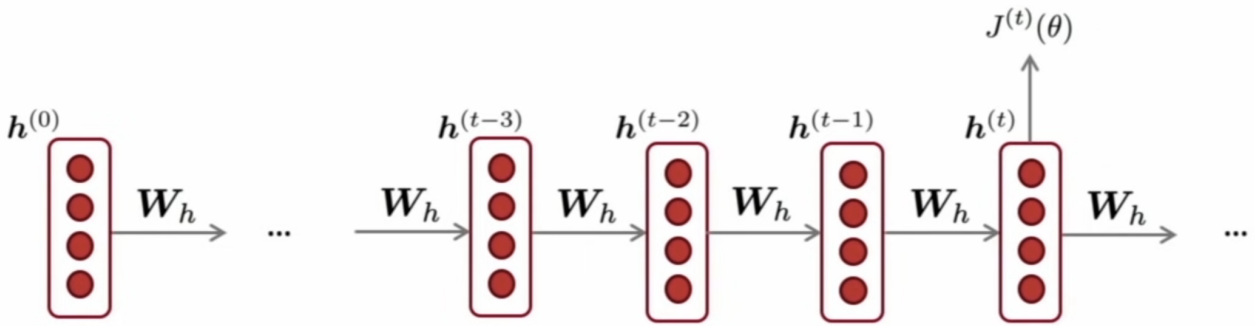
29

Copyright: Stanford CS224N: NLP with Deep Learning, Lecture 6; Winter 2019;
https://www.youtube.com/watch?v=iWea12EAu6U&list=PLoROMvodv4rOhcuXMZkNm7j3fVwBBY42z&index=8



Copyright: Stanford CS224N: NLP with Deep Learning, Lecture 6; Winter 2019;
`https://www.youtube.com/watch?v=iWea12EAu6U&list=PLoROMvodv4rOhcuXMZkNm7j3fVwBBY42z&index=8`

Backpropagation for RNNs

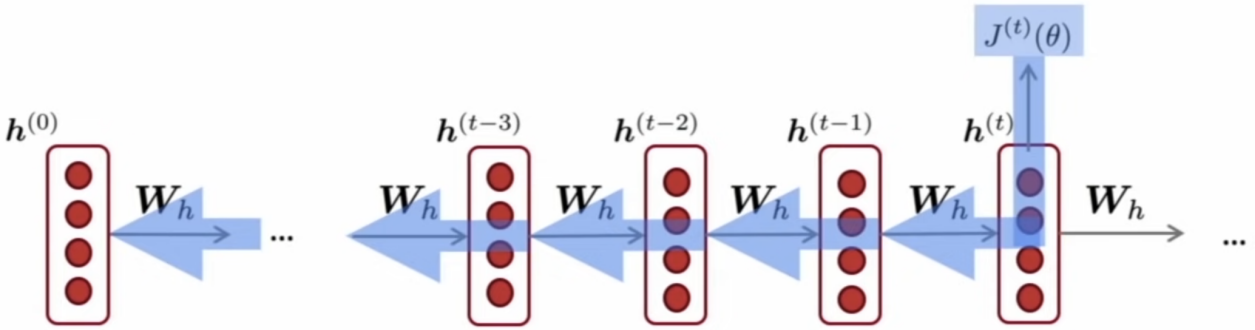


Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W_h}$?

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

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

Backpropagation for RNNs



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

Question: How do we calculate this?

Answer: Backpropagate over timesteps *i*=*t*,...,0, summing gradients as you go.
This algorithm is called **"backpropagation through time"**

35

Copyright: Stanford CS224N: NLP with Deep Learning, Lecture 6; Winter 2019;
`https://www.youtube.com/watch?v=iWea12EAu6U&list=PLoROMvodv4rOhcuXMZkNm7j3fVwBBY42z&index=8`

Backpropagation Through Time (BPT)

$$s_t = tanh(Ux_t + Ws_{t-1}) \tag{14}$$

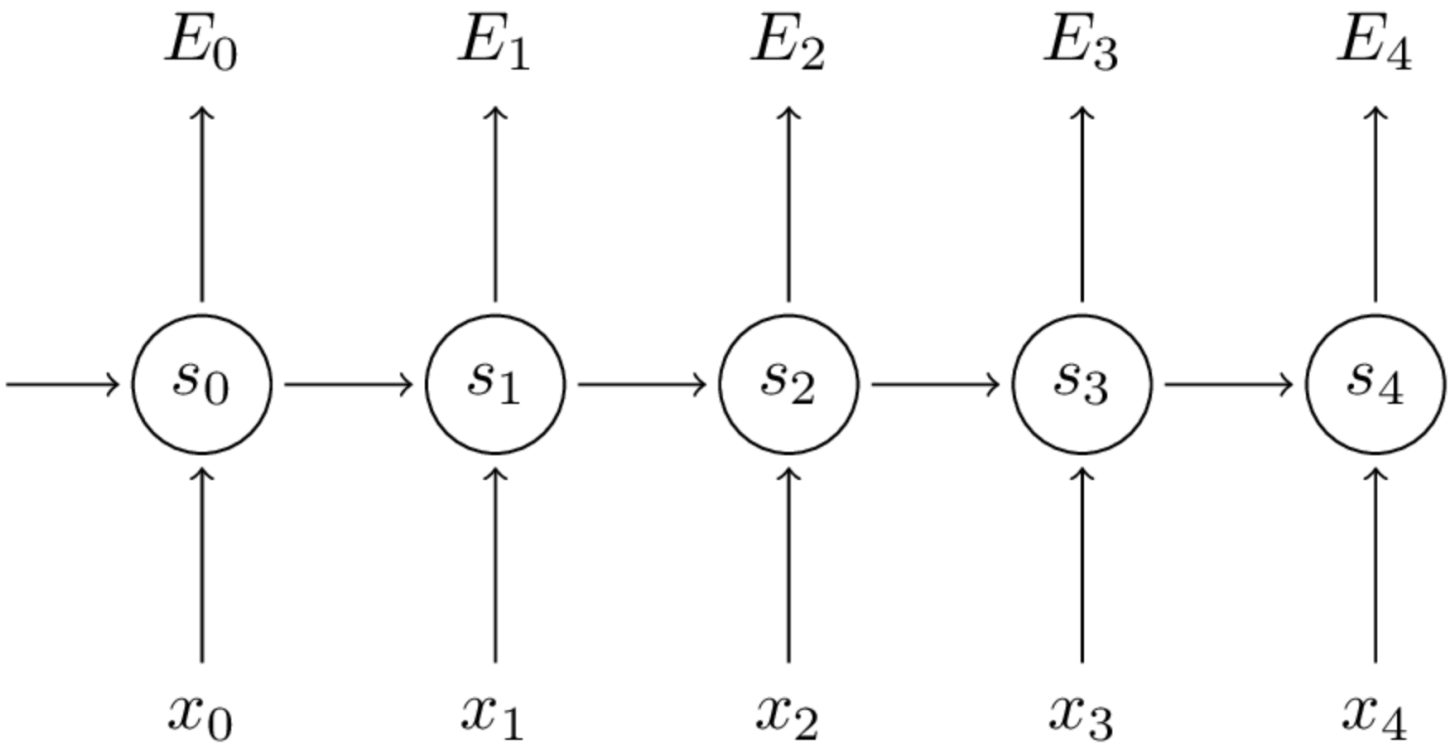
$$\hat{y}_t = \texttt{softmax}(Vs_t) \tag{15}$$

$$E_t(y_t, \hat{y}_t) = -y_t \texttt{log}\hat{y}_t \tag{16}$$

$$\begin{aligned} E_t(y, \hat{y}) &= \sum_t E_t(y_t, \hat{y}_t) \\ &= -\sum_t y_t \texttt{log}\hat{y}_t \end{aligned} \tag{17}$$

Copyright: https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-3/

Example



Copyright: `https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-3/`

Calculate the gradients of the error with respect to our parameters U,V and W

$$\begin{aligned}\frac{\partial E_3}{\partial V} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V} \\ &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V} \\ &= (\hat{y}_3 - y_3) \otimes s_3\end{aligned} \tag{18}$$

Take note

$\frac{\partial E_3}{\partial V}$ depends only on the values at the current time step: $\hat{y}_3, y_3, s_3$

But: $\frac{\partial E_3}{\partial W}$ and $\frac{\partial E_3}{\partial U}$ depend also on previous time steps:

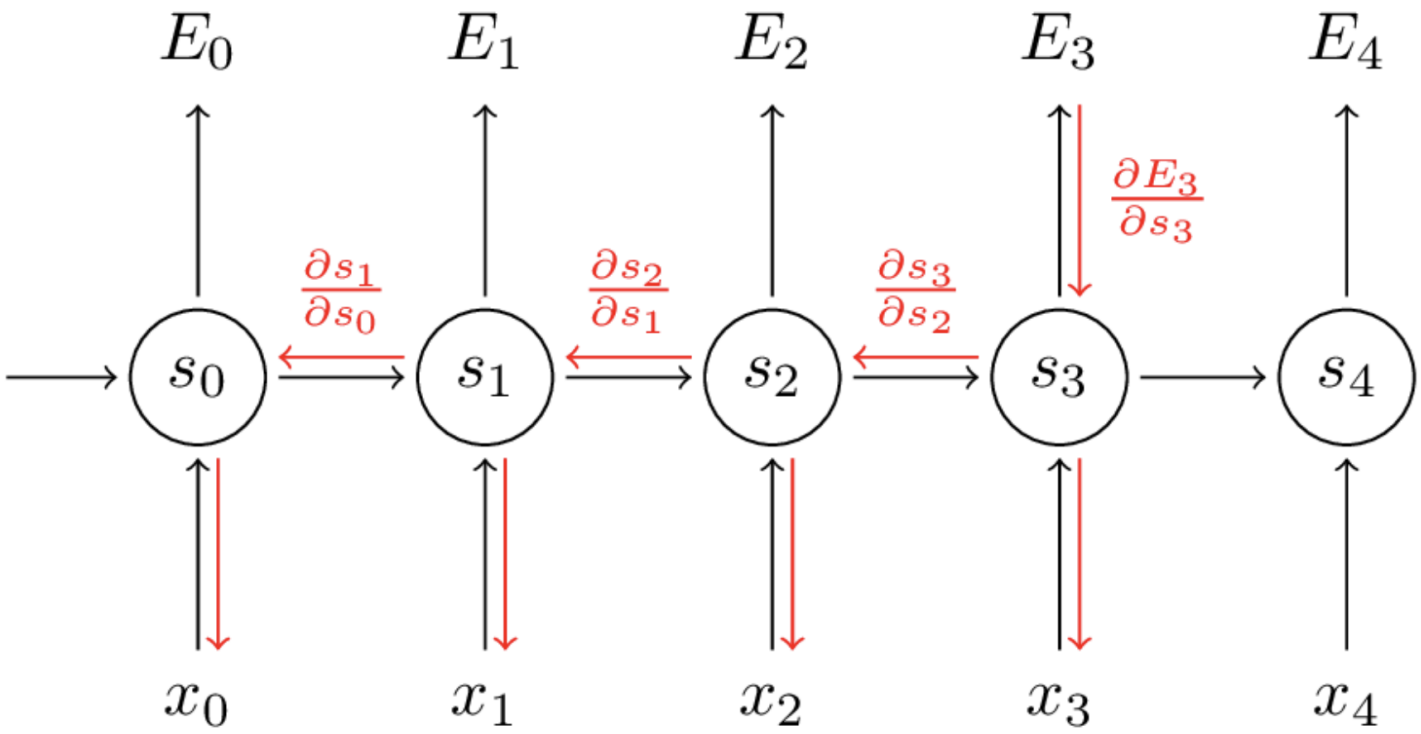
$$s_3 = \tanh(Ux_t + Ws_2) \tag{19}$$

depends on s_2, which depends on W and s_1, and so on.

Therefore:

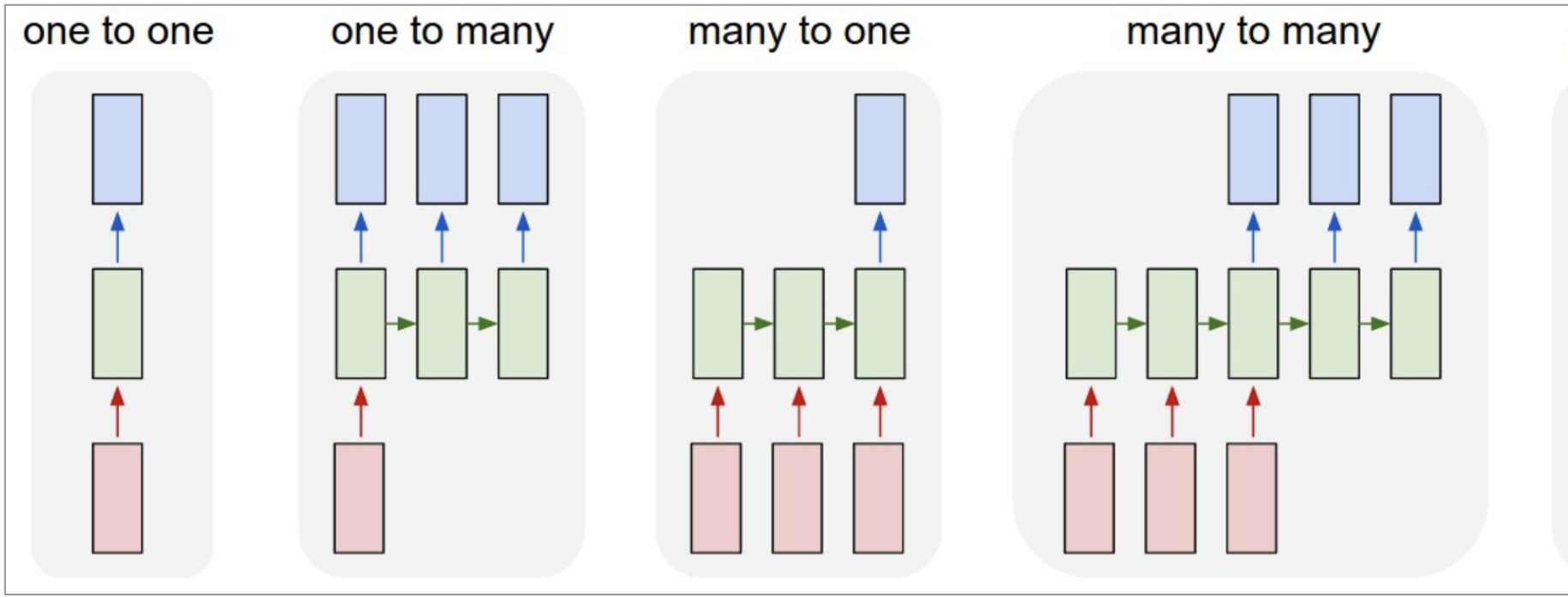
$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W} \tag{20}$$

RNN Unrolled with Gradients



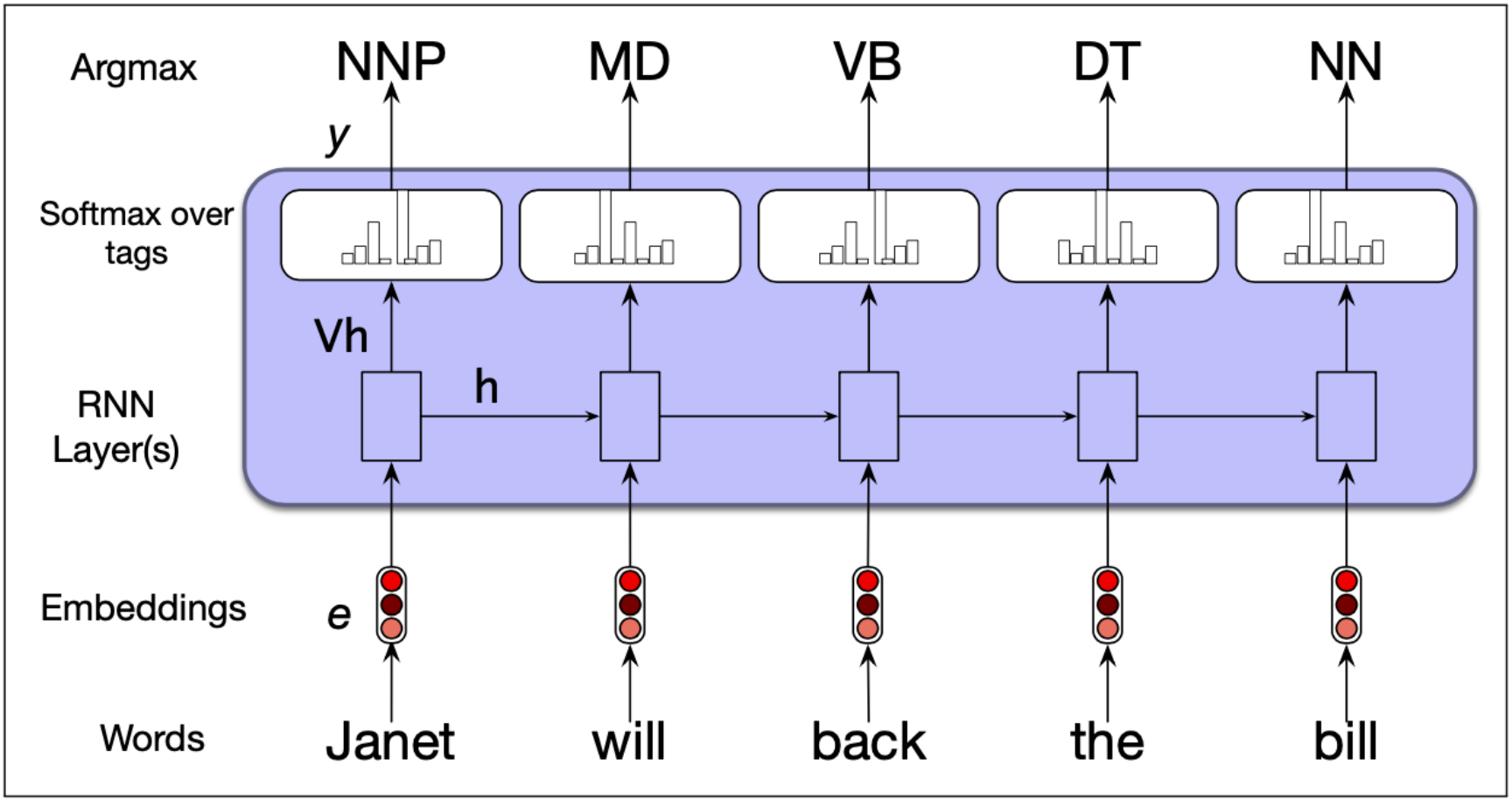
Copyright: https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-3/

RNN Architectures Overview

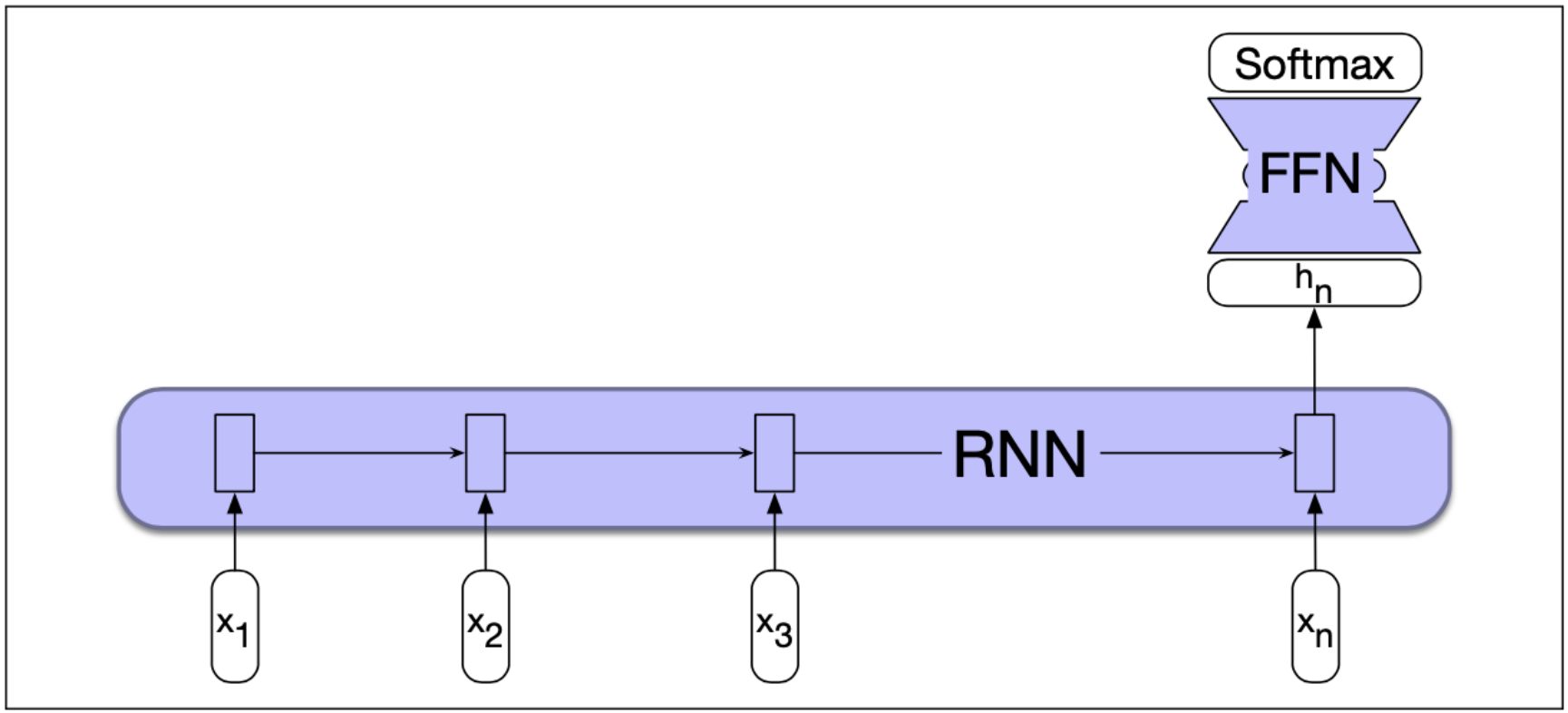


Copyright: Andrej Karpathy blog; https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sequence Labeling



RNNs for Sequence Classification



RNNs for Sequence Classification

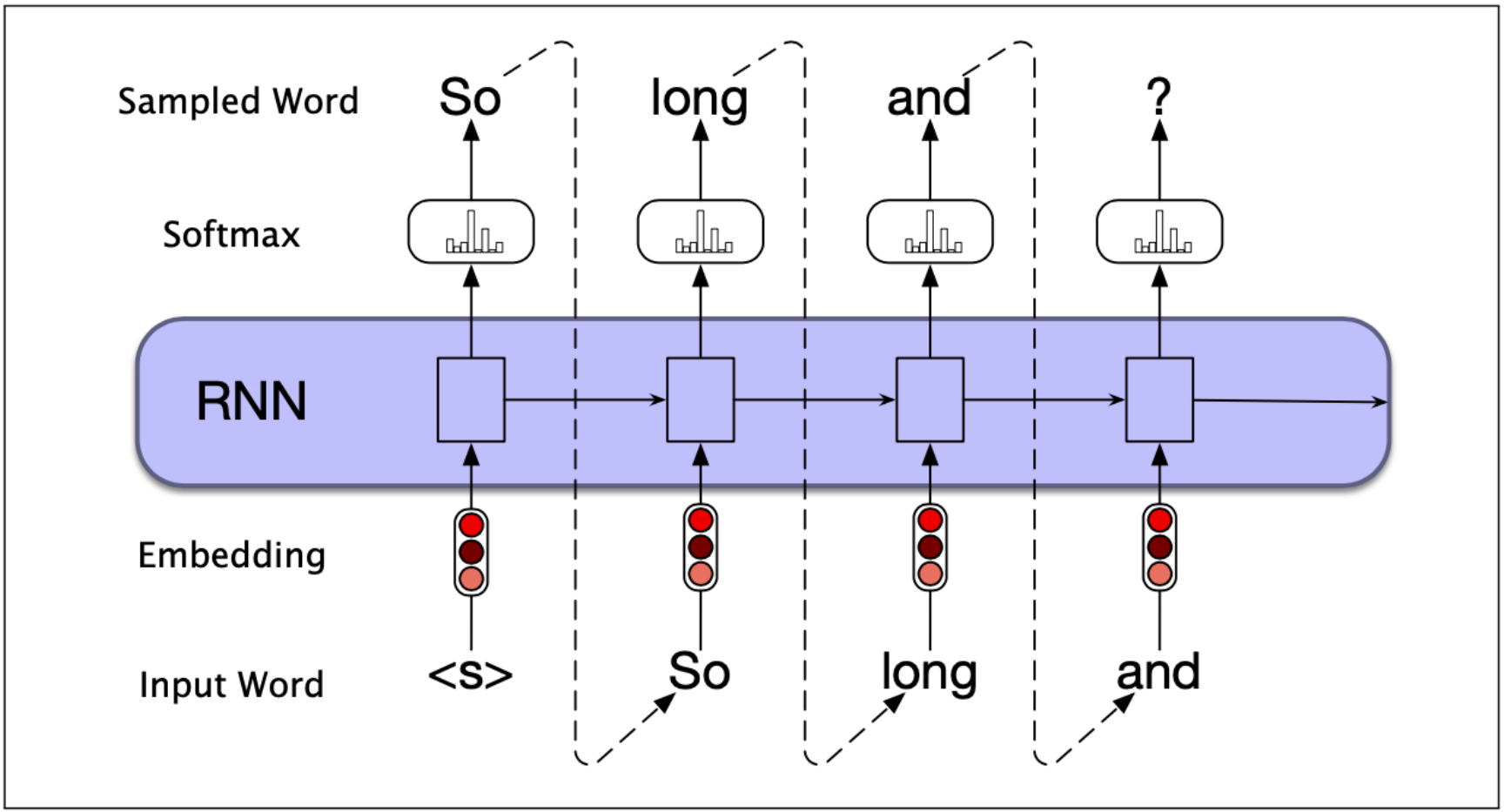
Pooling

$$\mathbf{h}_{mean} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{h}_i \tag{21}$$

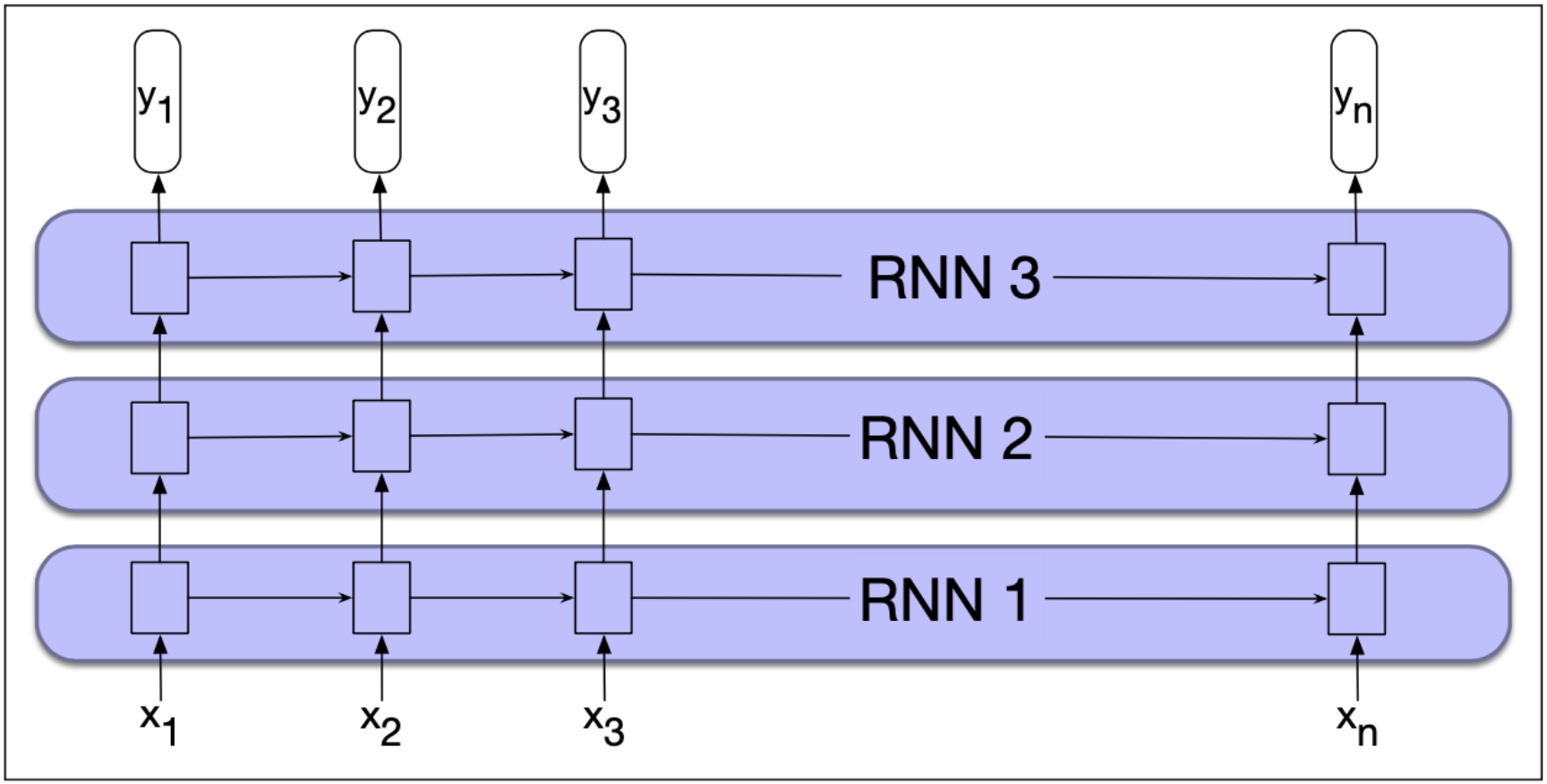
Autoregressive Generation with RNN-Based Language Models

- Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, <s>, as the first input.
- Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
- Continue generating until the end of sentence marker, <s>, is sampled or a fixed length limit is reached.

Autoregressive Generation with RNN-Based Language Models



Stacked RNNs



Bidirectional RNNs

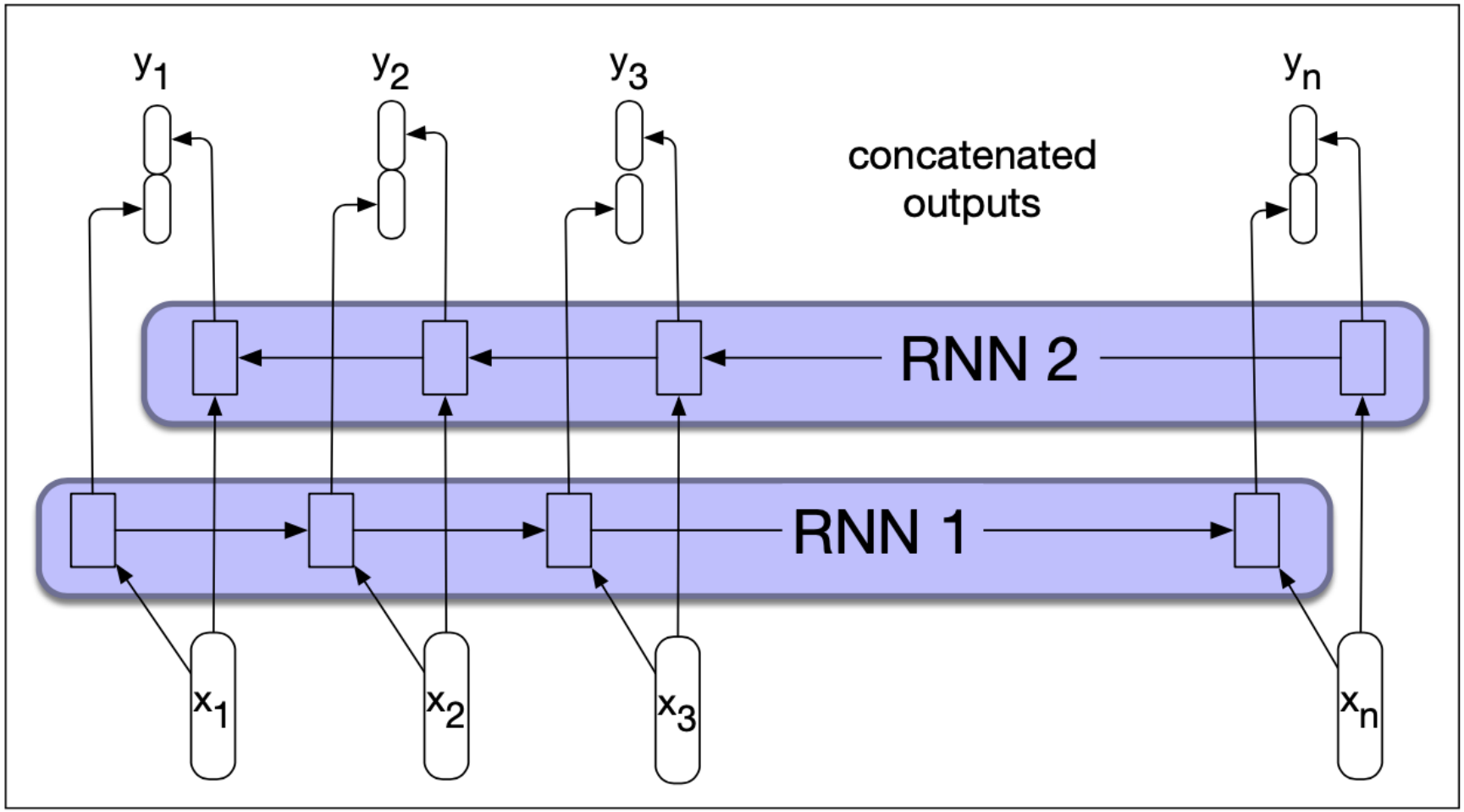
$$\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t) \tag{22}$$

$$\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_t, \ldots, \mathbf{x}_n) \tag{23}$$

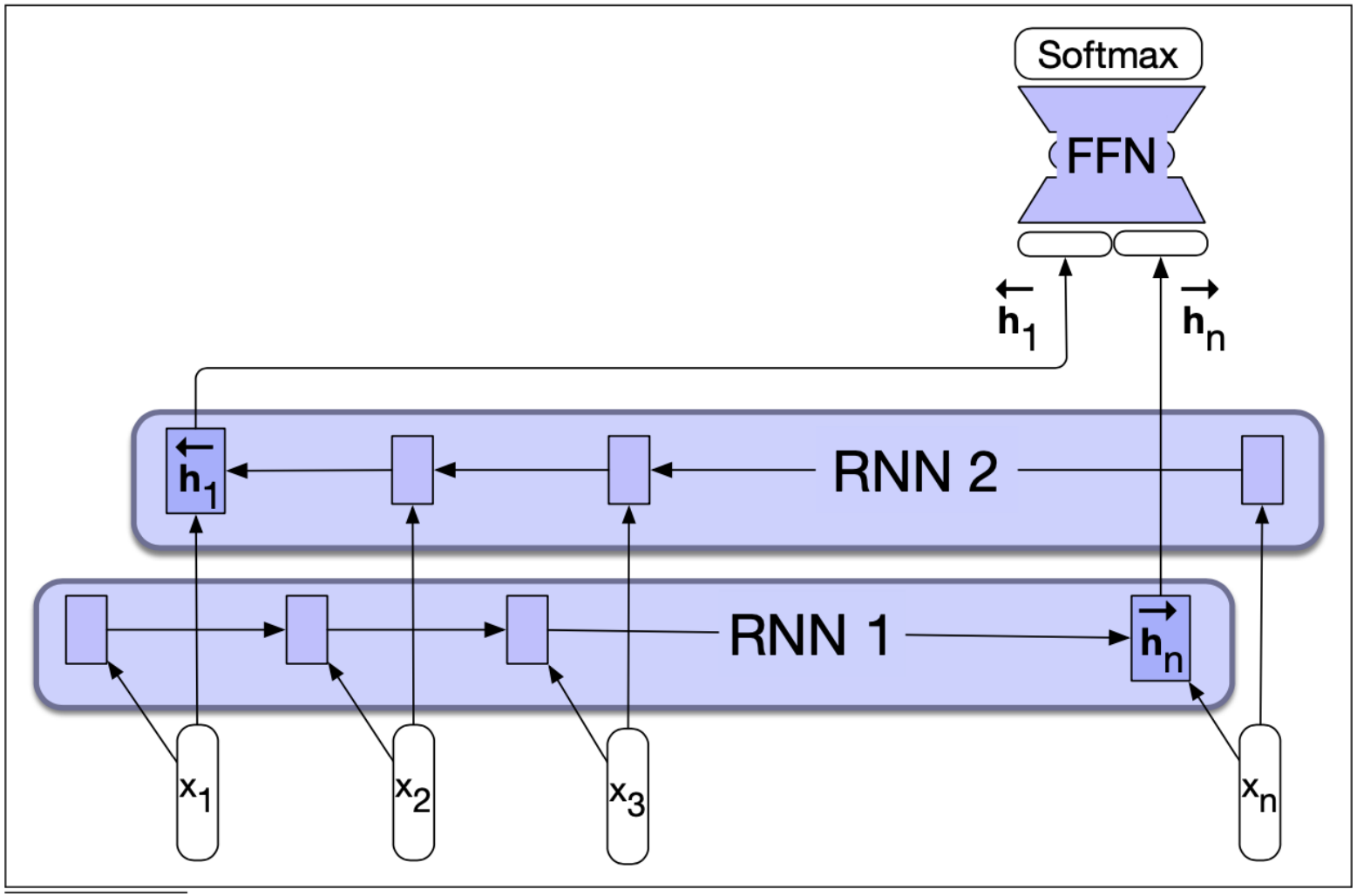
$$\mathbf{h}_t = [\mathbf{h}_t^f; \mathbf{h}_t^b] \tag{24}$$

$$= \mathbf{h}_t^f \oplus \mathbf{h}_t^b \tag{25}$$

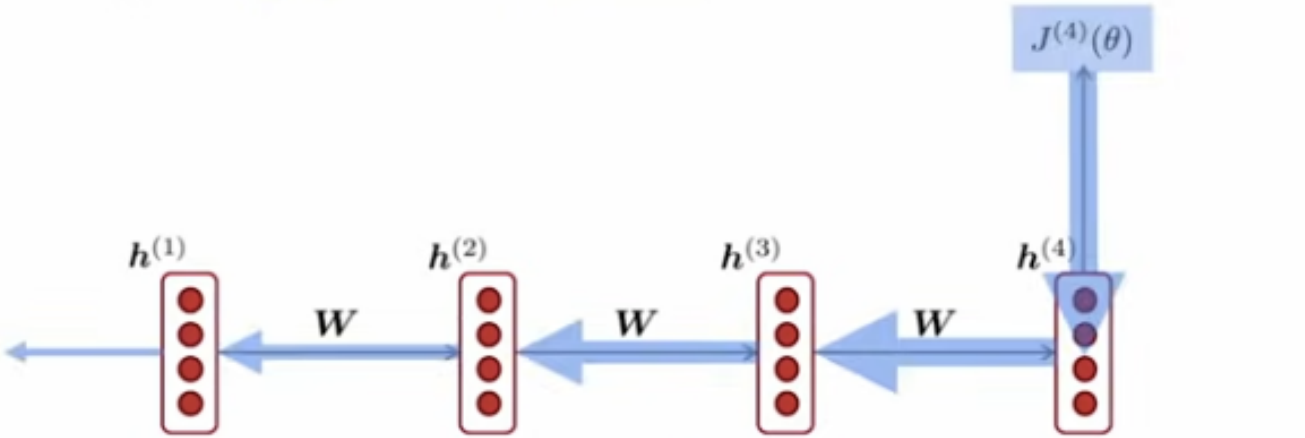
Bi-directional RNN for Sequence Labelling



Bi-directional RNN for Sequence Classification



Vanishing gradient intuition



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

What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

11

Long-distance Dependencies

Example

The flights the airline was cancelling were full.

If trains leave the station near my home early, then they tend to be on time.

Who does Mike want more than anybody to win the race?

Whom does Mike want more than anybody to meet at the party?

Three Alternatives for Dealing with Sequences of Language Data (3)

- Presenting individual input tokens to an LSTM (short for: Long-Short Term Memory; Hochreiter and Schmidhuber 1997) or a GRU (short for: Gated Recurrent Unit; Cho et al. 2014)
- Basic poblem with RNNs: The weights of the hidden layer are responsible for two tasks:
  - supply information for determining the output at the current time step t
  - updating and carrying forward information from previous time steps t-M relevant for future decisions at time steps t+N
- Basic idea underlying GRUs and LSTMs: separate these two tasks and thereby solving the vanishing/exploding gradient problem.

Added Complexity for neural units in GRUs and LSTMs

- Introduction of an additional context vector for input and output to neural units (for LSTMs)
- Encapsulating additional gates within the neural units themselves
- With different internal structures for GRUs and LSTMs

Gated Recurrent Unit

Two most widely used gated recurrent units

Gated Recurrent Unit

[Cho et al., EMNLP2014; Chung, Gulcehre, Cho, Bengio, DLUFL2014]

$$h_t = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$
$$\tilde{h} = \tanh(W\,[x_t] + U(r_t \odot h_{t-1}) + b)$$
$$u_t = \sigma(W_u\,[x_t] + U_u h_{t-1} + b_u)$$
$$r_t = \sigma(W_r\,[x_t] + U_r h_{t-1} + b_r)$$

Long Short-Term Memory

[Hochreiter & Schmidhuber, NC1999; Gers, Thesis2001]

$$h_t = o_t \odot \tanh(c_t)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$\tilde{c}_t = \tanh(W_c\,[x_t] + U_c h_{t-1} + b_c)$$
$$o_t = \sigma(W_o\,[x_t] + U_o h_{t-1} + b_o)$$
$$i_t = \sigma(W_i\,[x_t] + U_i h_{t-1} + b_i)$$
$$f_t = \sigma(W_f\,[x_t] + U_f h_{t-1} + b_f)$$

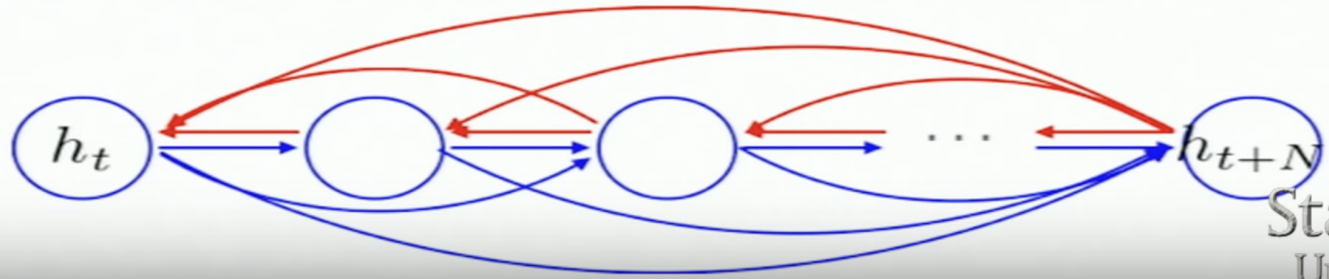
Stanford University

11

Gated Recurrent Unit

- It implies that the error must backpropagate through all the intermediate nodes:

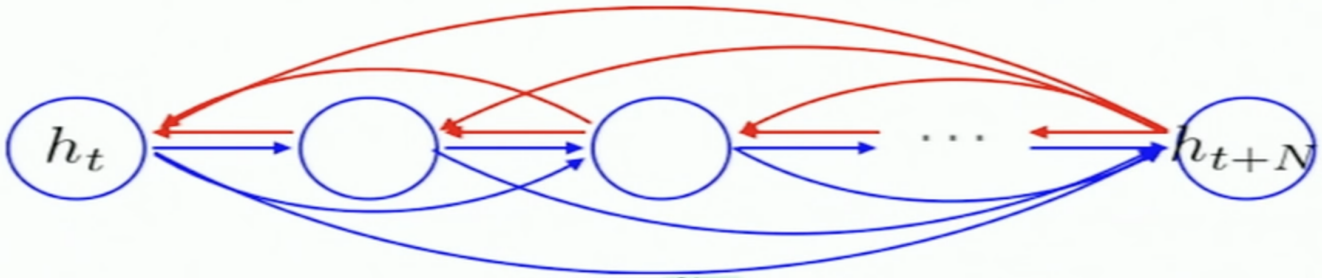
- Perhaps we can create shortcut connections.



Stanford University

Gated Recurrent Unit

- Perhaps we can create *adaptive* shortcut connections.



$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

- Candidate Update $\tilde{h}_t = \tanh(W\,[x_t] + U h_{t-1} + b)$
- Update gate $u_t = \sigma(W_u\,[x_t] + U_u h_{t-1} + b_u)$

Stanford University

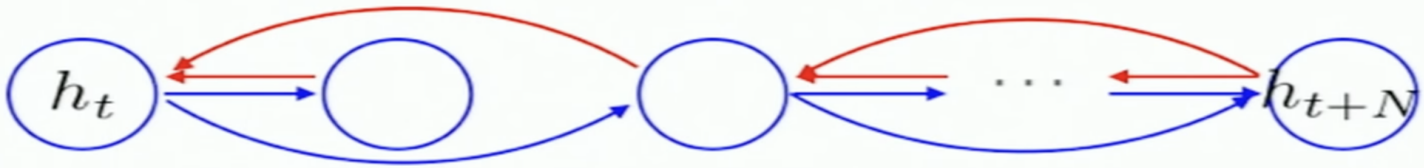
Hadamard Product

The Hadamard product for a 3x3 matrix A with a 3x3 matrix B is:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\, b_{11} & a_{12}\, b_{12} & a_{13}\, b_{13} \\ a_{21}\, b_{21} & a_{22}\, b_{22} & a_{23}\, b_{23} \\ a_{31}\, b_{31} & a_{32}\, b_{32} & a_{33}\, b_{33} \end{bmatrix}$$

Gated Recurrent Unit

- Let the net prune unnecessary connections *adaptively*.

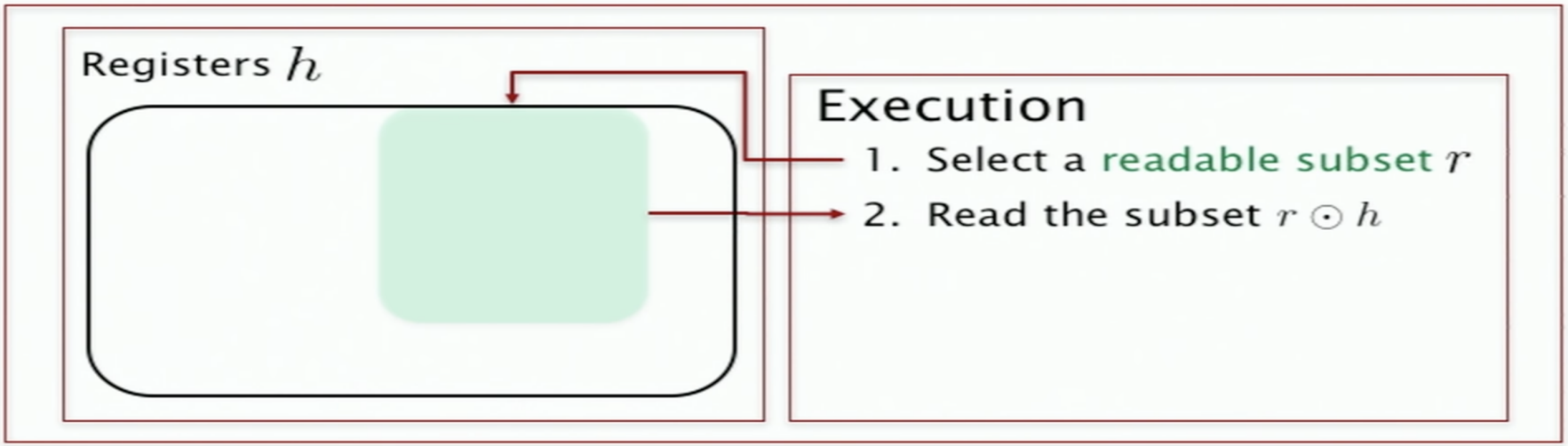


$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

- Candidate Update $\tilde{h}_t = \tanh(W\,[x_t] + U(r_t \odot h_{t-1}) + b)$
- Reset gate $r_t = \sigma(W_r\,[x_t] + U_r h_{t-1} + b_r)$
- Update gate $u_t = \sigma(W_u\,[x_t] + U_u h_{t-1} + b_u)$

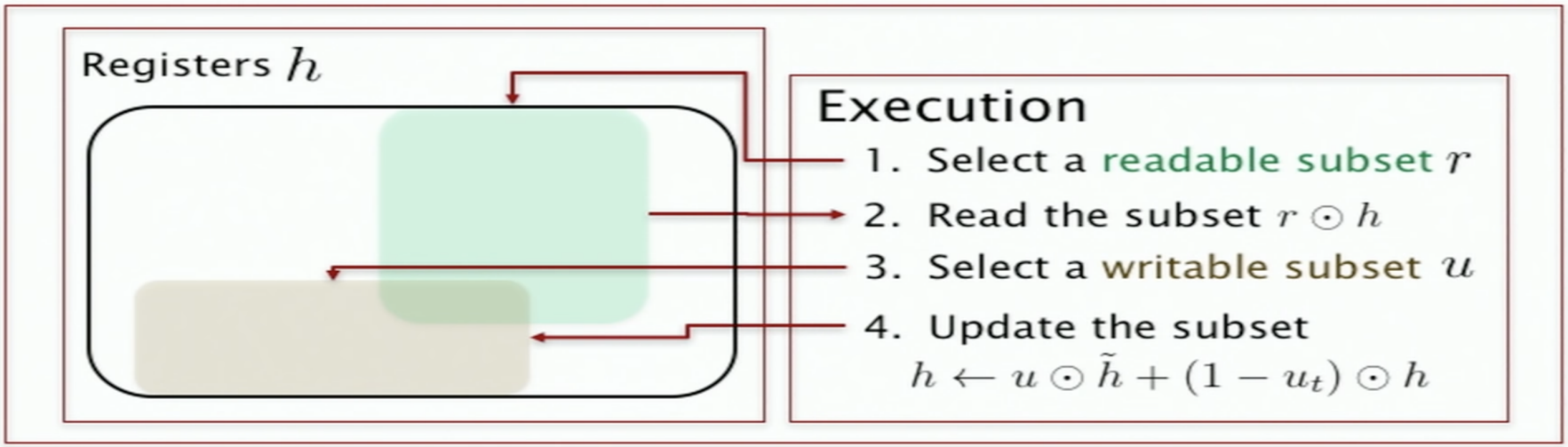
Gated Recurrent Unit

GRU ...

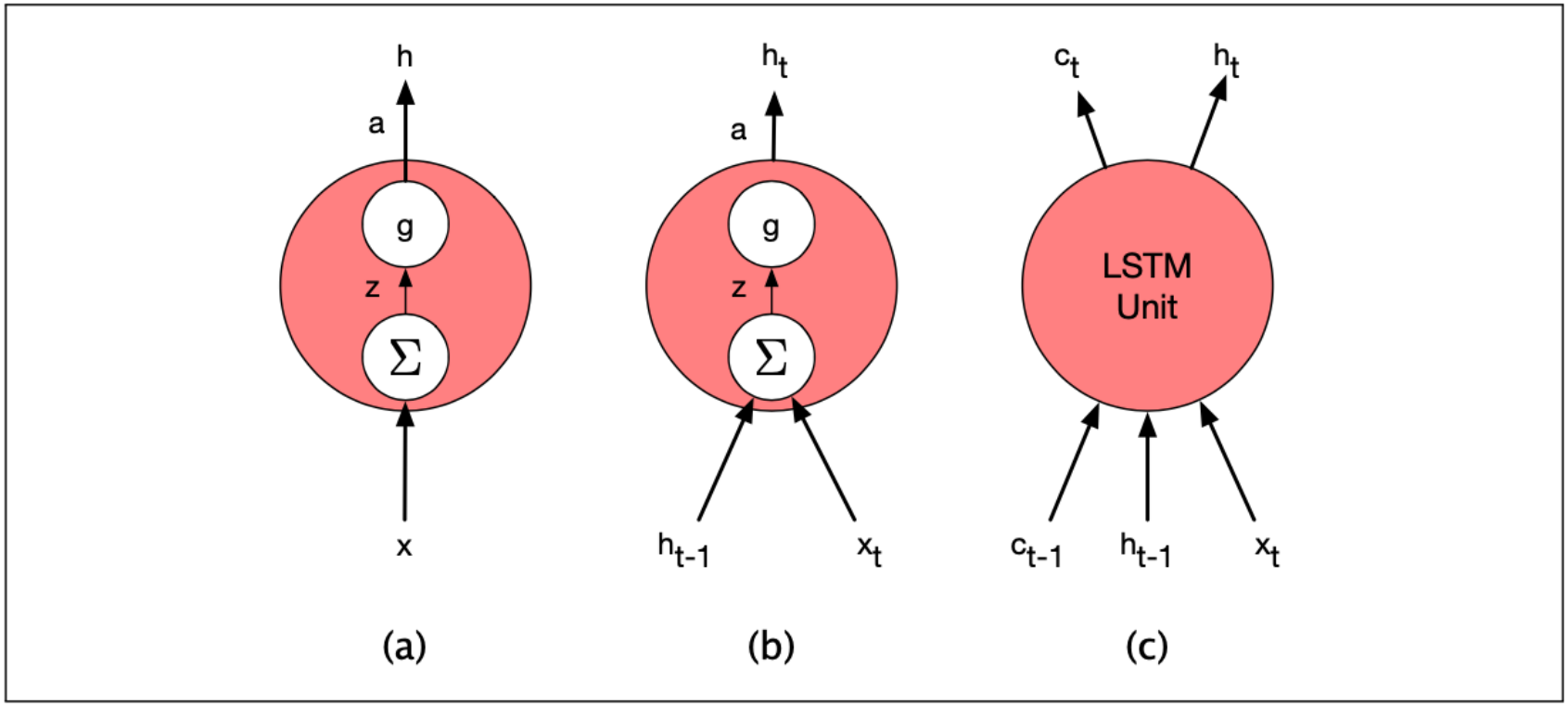


Gated Recurrent Unit

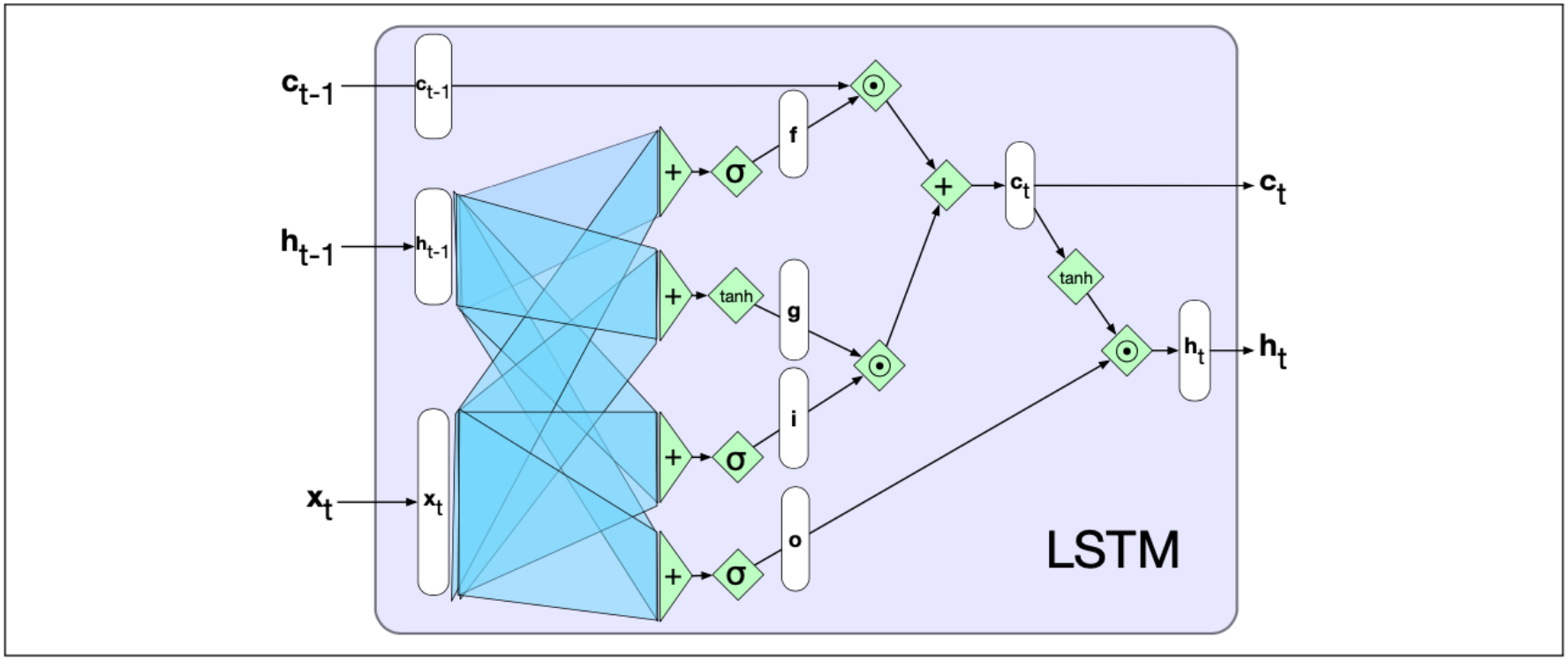
GRU ...



Gated Units, Layers and Networks



The LSTM



The LSTM

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t) \tag{26}$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \tag{27}$$

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t) \tag{28}$$

The LSTM

$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t) \tag{29}$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \tag{30}$$

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \tag{31}$$

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t) \tag{32}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{33}$$

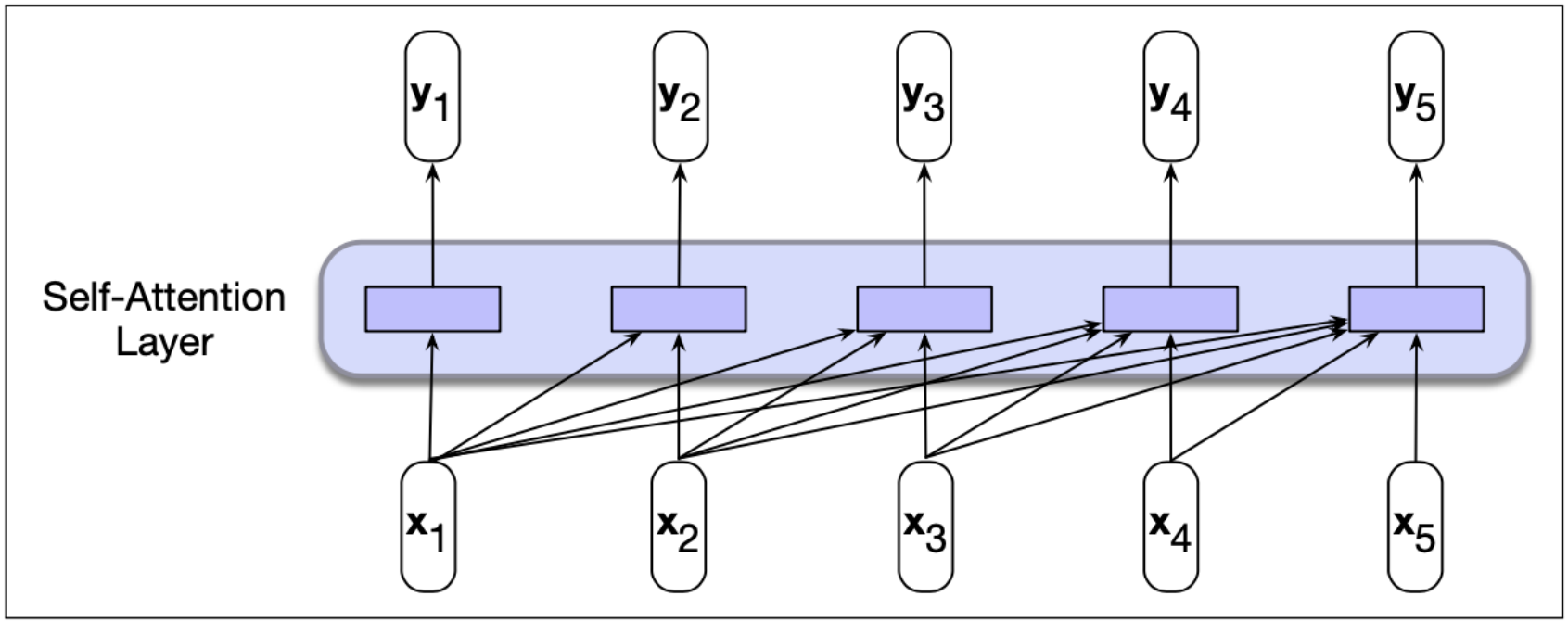
Transformers

- Transformers are non-recurrent networks based on (self-)attention.
- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass them through intermediate recurrent connections as in RNNs.
- A self-attention layer maps input sequences to output sequences of the same length, using attention heads.
- Attention heads model how the surrounding words are relevant for the processing of the current word.

Flow of Information in a Self-Attention Layer

- When processing each item in the input of a self-attention layer, the model has access to all inputs up to and including the one under consideration, but no access to information about inputs beyond the current one.
- The computation performed for each item is independent of all the other computations. Hence, forward inference and training can proceed in parallel.

Self-Attention Layer



Self-Attention Networks: Transformers

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{34}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \; \forall j \leq i \tag{35}$$

$$= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \; \forall j \leq i \tag{36}$$

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{37}$$

Self-Attention Networks: Different Roles

An input embedding can play three different roles:

- **query**: as the *current focus of attention* that is compared to all of the other preceding inputs.
- **key**: as *a preceding input* that is compared to the current focus of attention.
- **value** that is used to compute the output for the current focus of attention.

Self-Attention Networks: Transformers

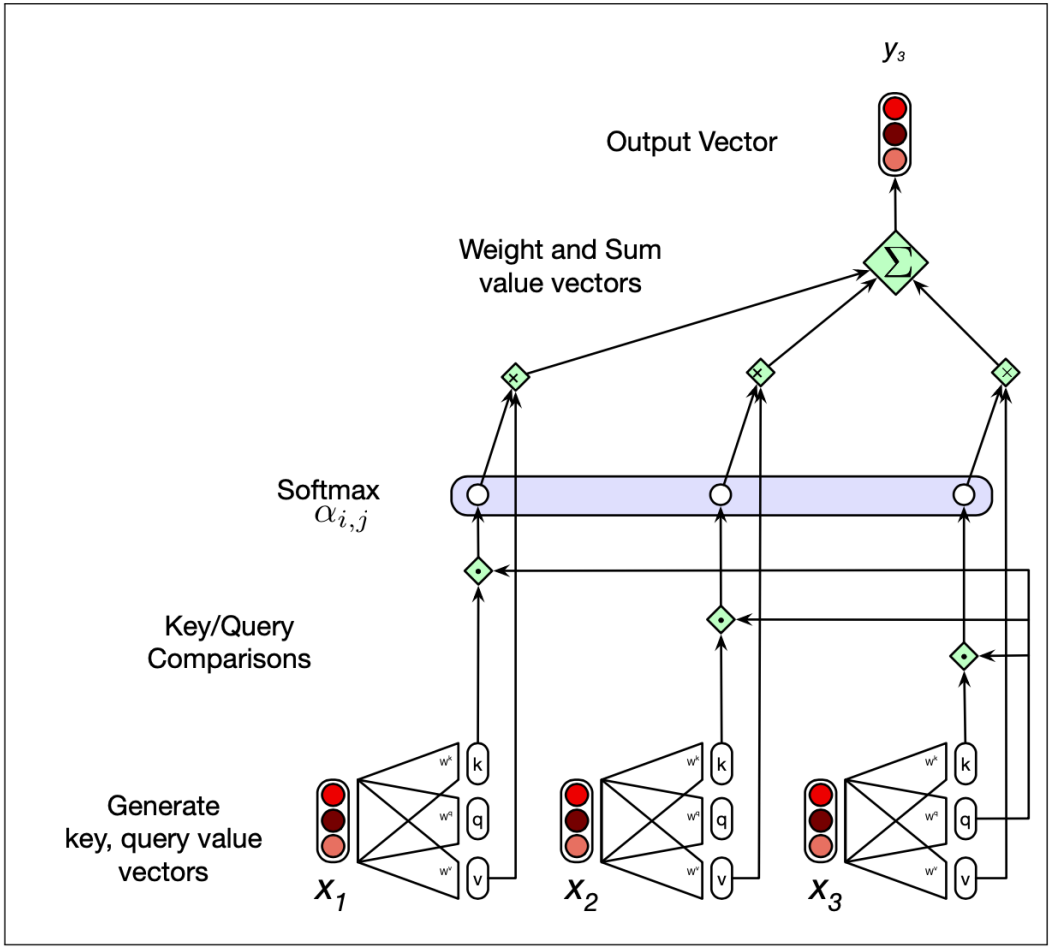
- The roles of query, key, and value are differentiated by three different weight matrices: $\mathbf{W}^{\mathbf{Q}} \in \mathbb{R}^{d \times d'}$, $\mathbf{W}^{\mathbf{K}} \in \mathbb{R}^{d \times d'}$, and $\mathbf{W}^{\mathbf{V}} \in \mathbb{R}^{d \times d''}$.
- The inputs and outputs of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality $1 \times d$.

$$\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i;\ \mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i;\ \mathbf{v}_i = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i \tag{38}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j \tag{39}$$

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij}\mathbf{v}_j \tag{40}$$

Calculation of an Output Embedding in a Self-Attention Layer



Self-Attention Networks: Further Adjustments

Scaling the dot product by the square root of the dimensionality

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{41}$$

Parallelizing the Cmputation: packing the input embeddings of the N tokens into a single matrix

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}};\ \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}};\ \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}; \tag{42}$$

Final Result: Reducing the Self-Attention Step for an Entire Sequence of N Tokens

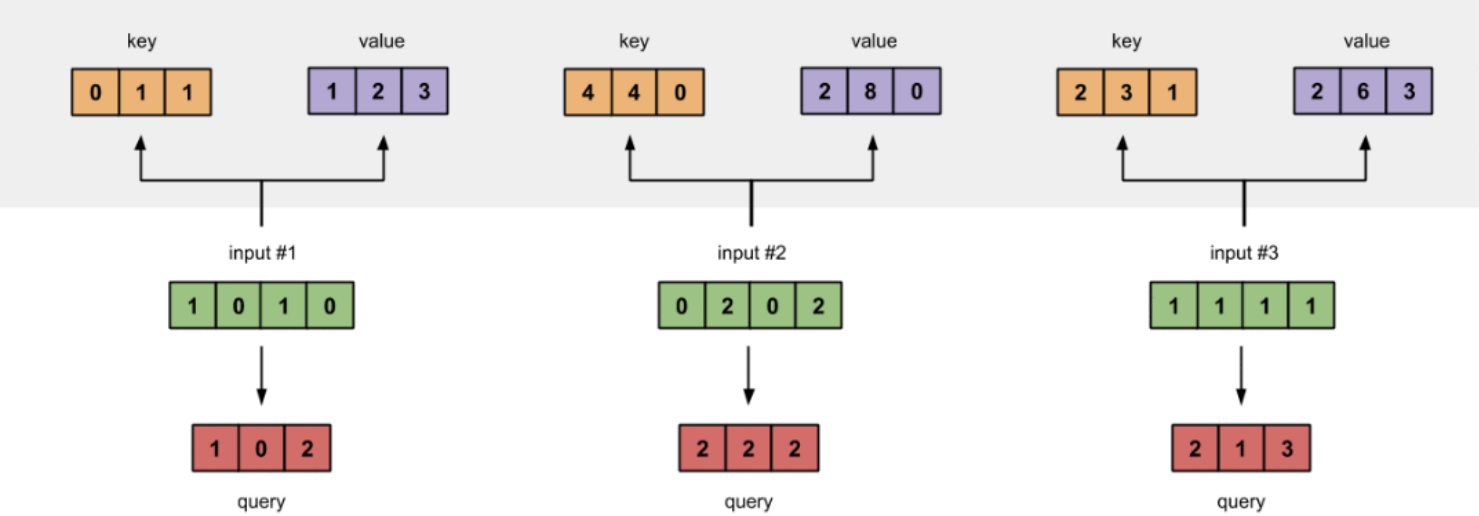
$$\textit{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{43}$$

Working through an Example with 3 Input Vectors

1. Prepare inputs
2. Initialise weights
3. Derive key, query and value
4. Calculate attention scores for Input 1
5. Calculate softmax
6. Multiply scores with values
7. Sum weighted values to get Output 1
8. Repeat steps 4–7 for Input 2 and Input 3

This workflow and the illustrations in the next three slides are due to `https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a`

Step 1-3


key
value
0 1 1
1 2 3
input #1
1 0 1 0
1 0 2
query
key
value
4 4 0
2 8 0
input #2
0 2 0 2
2 2 2
query
key
value
2 3 1
2 6 3
input #3
1 1 1 1
2 1 3
query

Initialize Weights for Query, Key, and Value

Weights for **key**:

```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

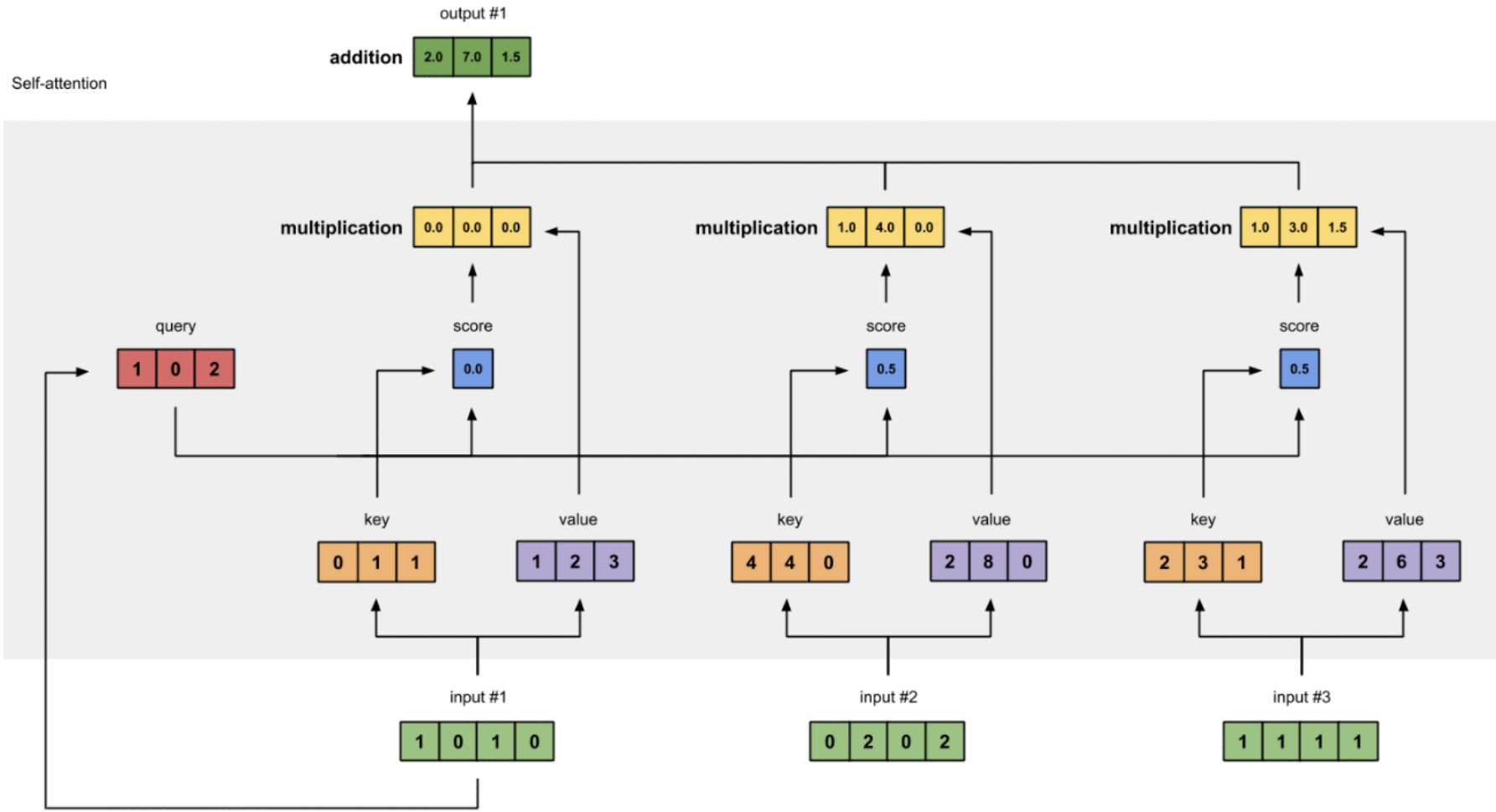
Weights for **query**:

```
[[1, 0, 1],
 [1, 0, 0],
 [0, 0, 1],
 [0, 1, 1]]
```

Weights for **value**:

```
[[0, 2, 0],
 [0, 3, 0],
 [1, 0, 3],
 [1, 1, 0]]
```

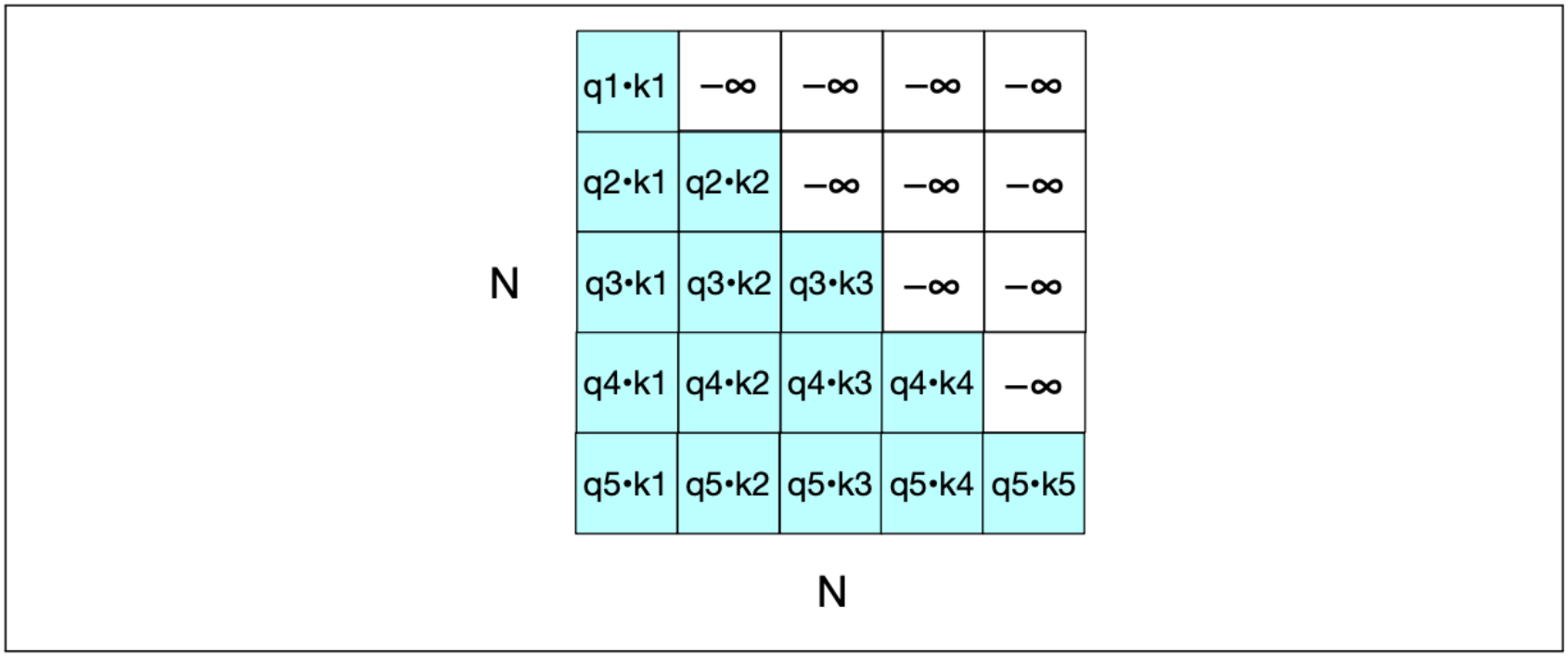
Calculate Attention Scores



Calculate Softmax

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{k} exp(z_j)} \quad 1 \leq i \leq k \tag{44}$$

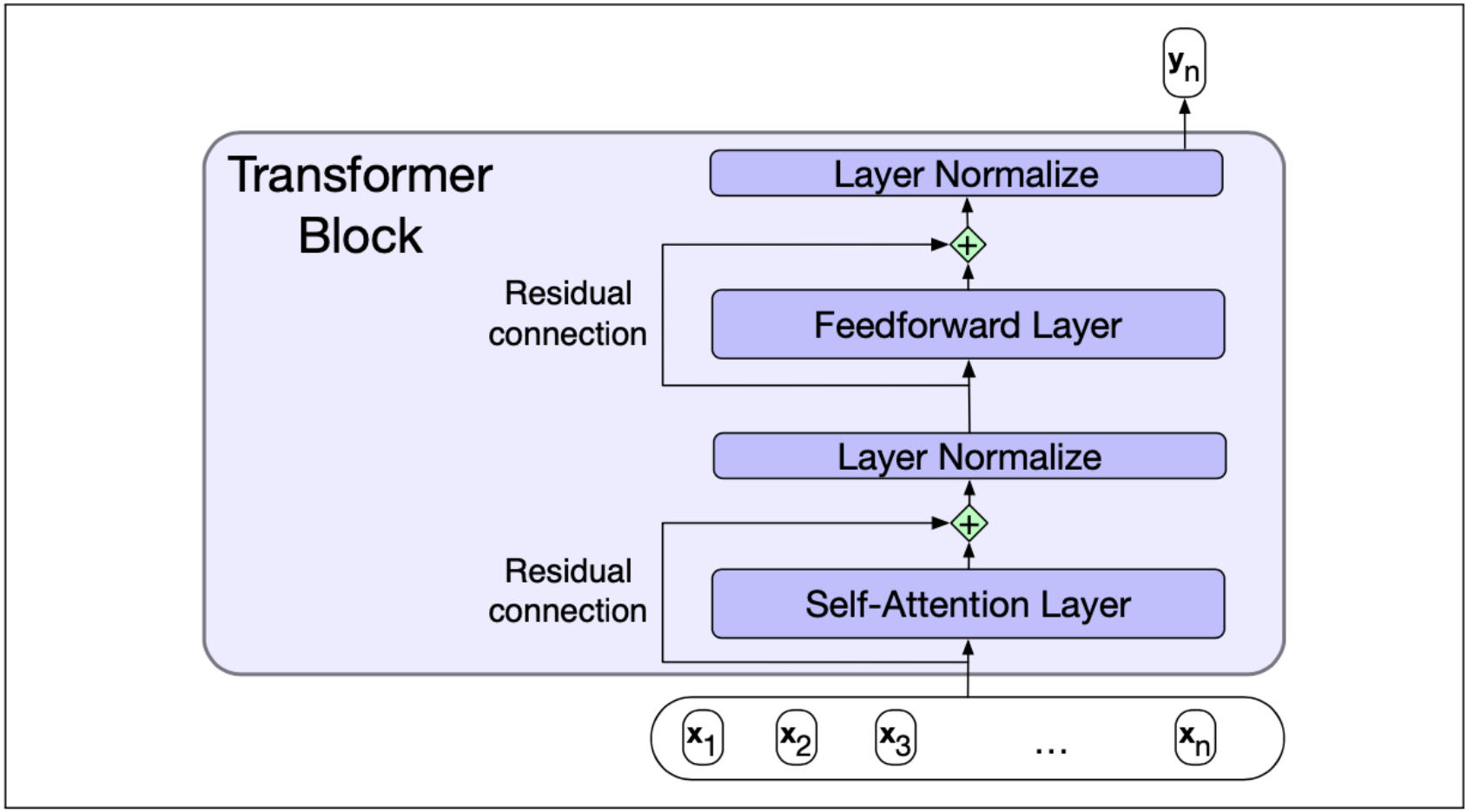
Zeroing Out the Upper Triangular Portion of a QT^T Matrix



Transformer Blocks

- A transformer block consists of a single attention layer followed by a feed-forward layer with residual connections and layer normalizations following each.
- Transformer blocks can be stacked to make deeper and more powerful networks.

Transformer Blocks



Transformer Blocks

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{SelfAttn}(\mathbf{x})) \tag{45}$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFNN}(\mathbf{z})) \tag{46}$$

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{47}$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2} \tag{48}$$

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma} \tag{49}$$

$$\textit{LayerNorm} = \gamma\,\hat{\mathbf{x}} + \beta \tag{50}$$

Multihead Attention Layers

- are sets of self-attention layers, each with its own set of key, query, and value matrices:
  - $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$
  - $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$
  - $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$
- Each member of such a set of self-attention layers is called a **head**
- Each head gets multiplied by the inputs packed into $\mathbf{X}$ to produce
  - $\mathbf{W}_i^Q \in \mathbb{R}^{N \times d_k}$
  - $\mathbf{W}_i^K \in \mathbb{R}^{N \times d_k}$
  - $\mathbf{W}_i^V \in \mathbb{R}^{N \times d_v}$

Multihead Attention Layers

- The output of a multi-head layer with h heads consists of h vectors of shape $N \times d_v$
- These outputs are concatenated from each head and then, using a linear projection with weight matrix $\mathbf{W}_i^O \in \mathbb{R}^{hd_k \times d}$, reduced to $N \times d$ output.

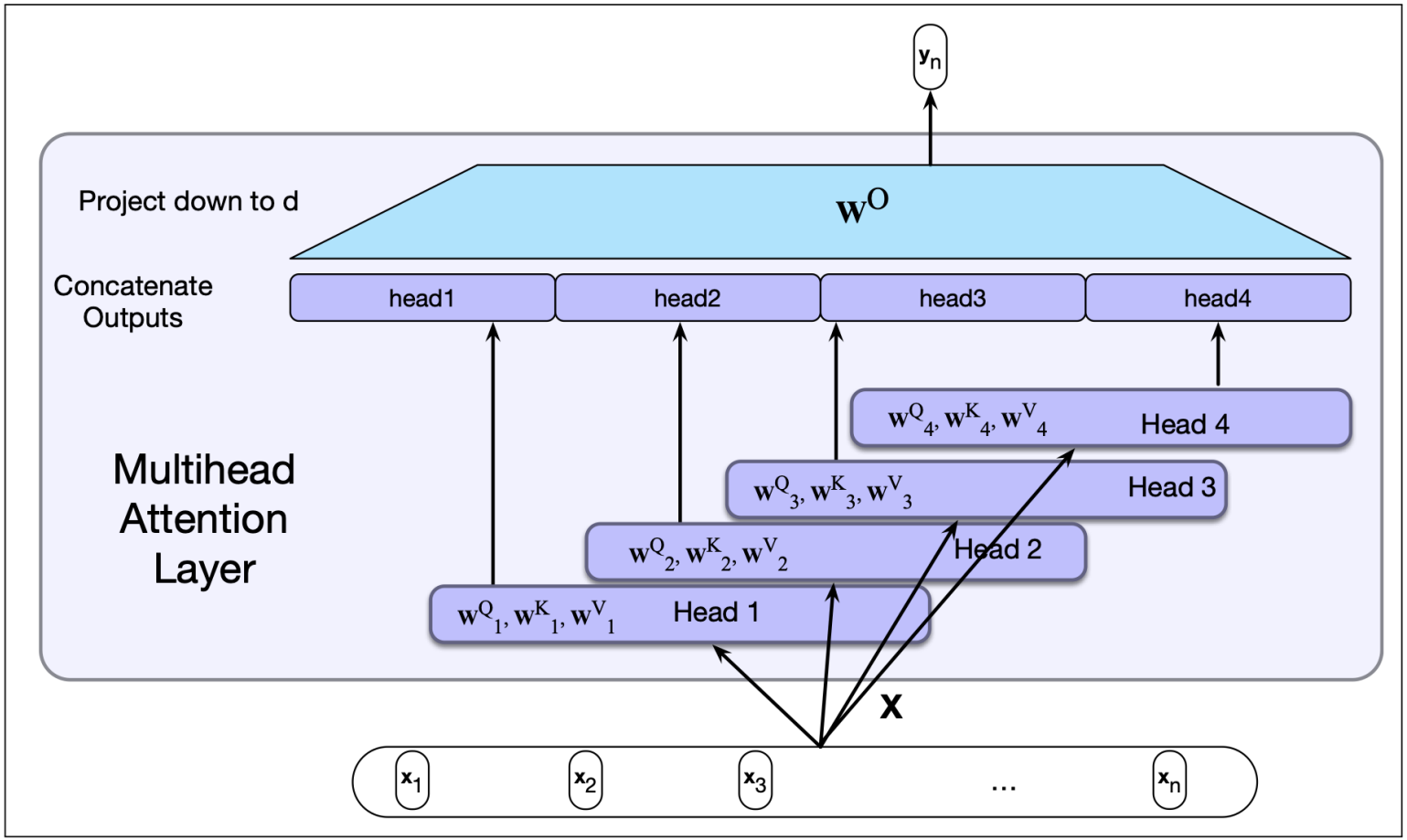
Multihead Attention

$$\mathit{MultiHeadAttn}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \cdots \oplus \mathbf{head}_h)\mathbf{W}^O \tag{51}$$

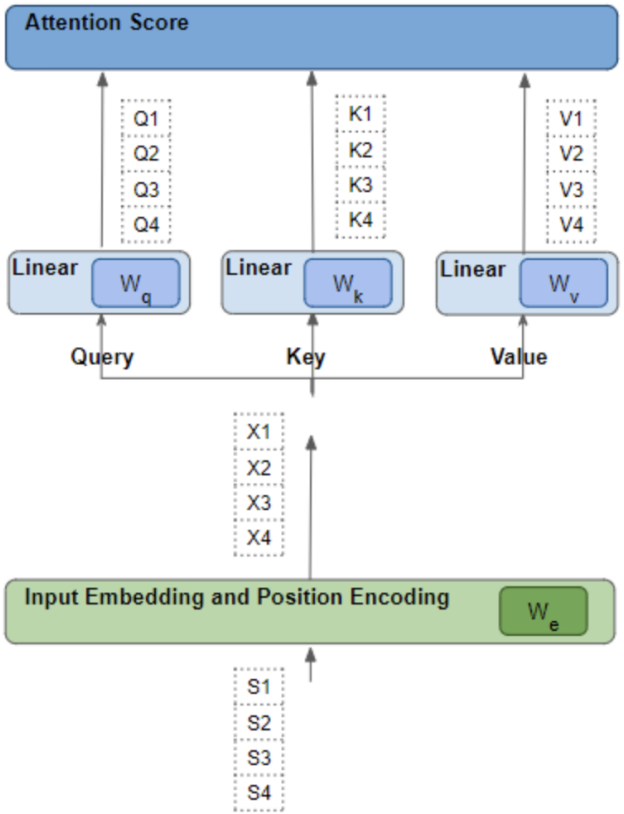
$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q;\ \mathbf{K} = \mathbf{X}\mathbf{W}_i^K;\ \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \tag{52}$$

$$\mathbf{head}_i = \mathit{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \tag{53}$$

Multihead Attention

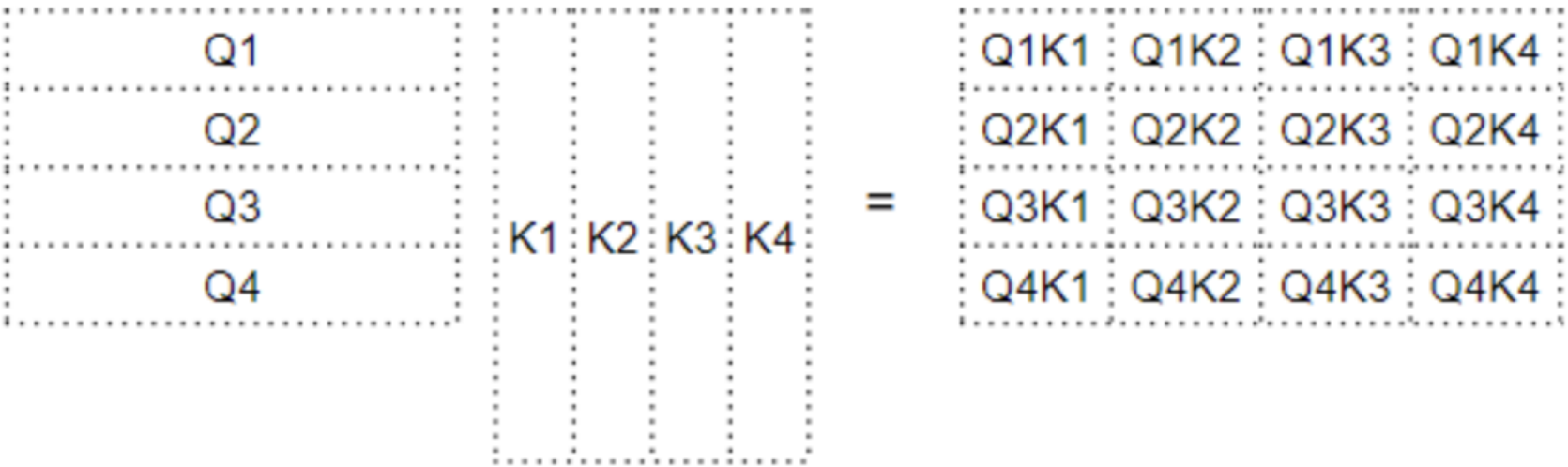


Multihead Attention: Visual Summary



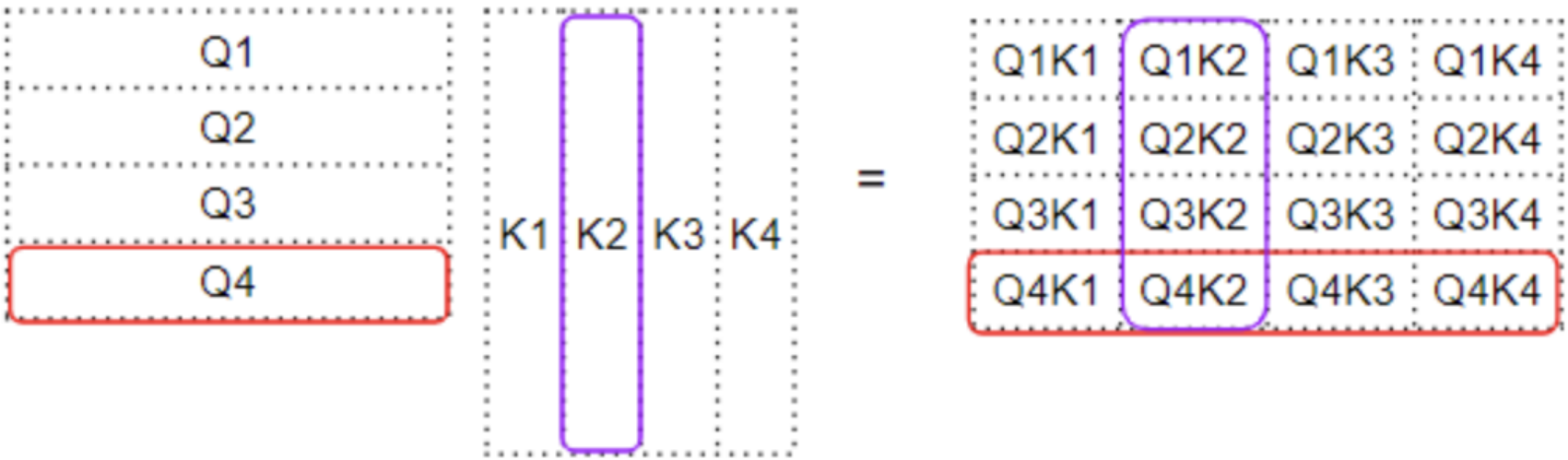
Slide due to Keitan Doshi, https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3

Multihead Attention: Dot Product between Query and Key matrices



Slide due to Keitan Doshi
https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3

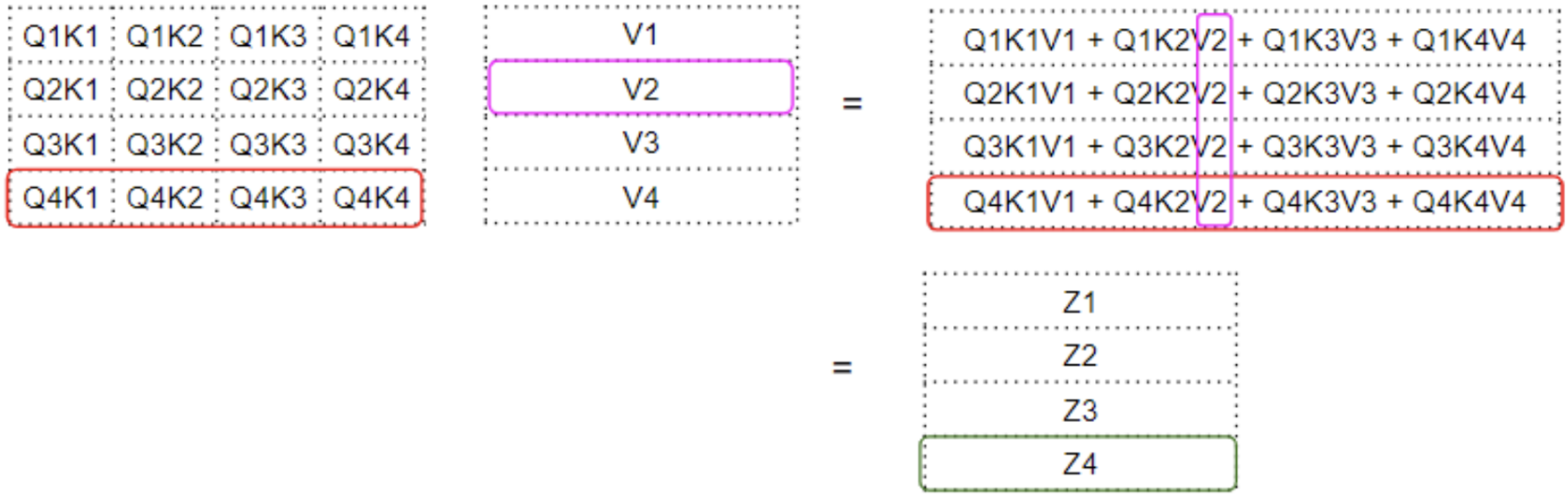
Multihead Attention: Dot Product between Query and Key matrices



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

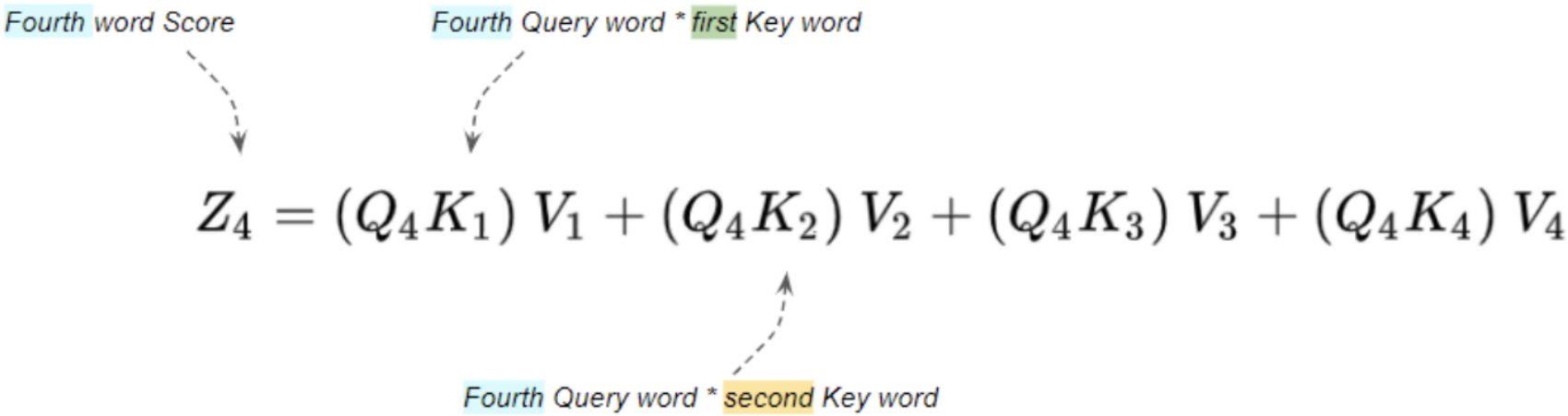
Dot Product between Query-Key and Value Matrices



Slide due to Keitan Doshi

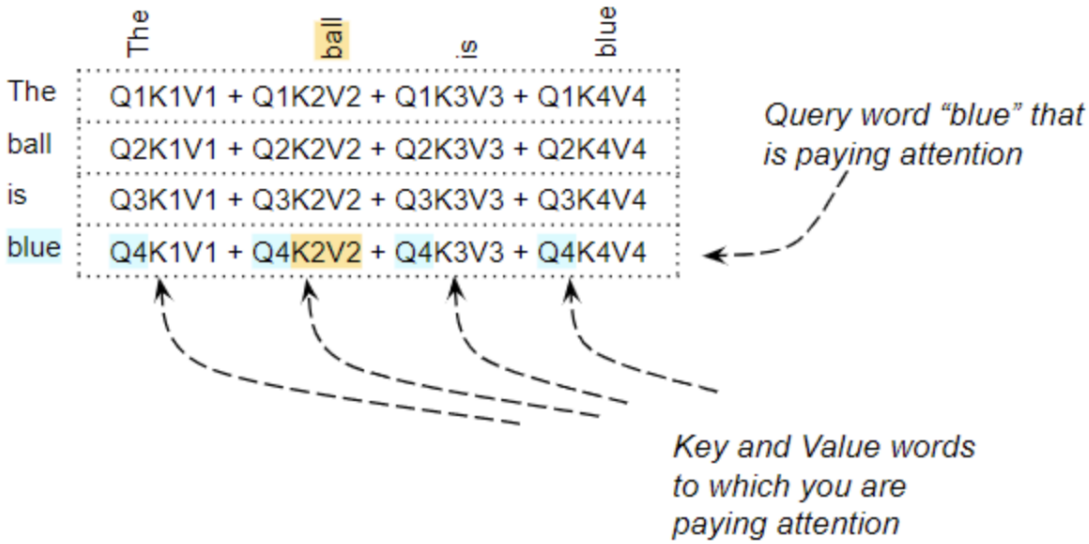
`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

Attention Score for the word blue pays attention to every other word



Slide due to Keitan Doshi
`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

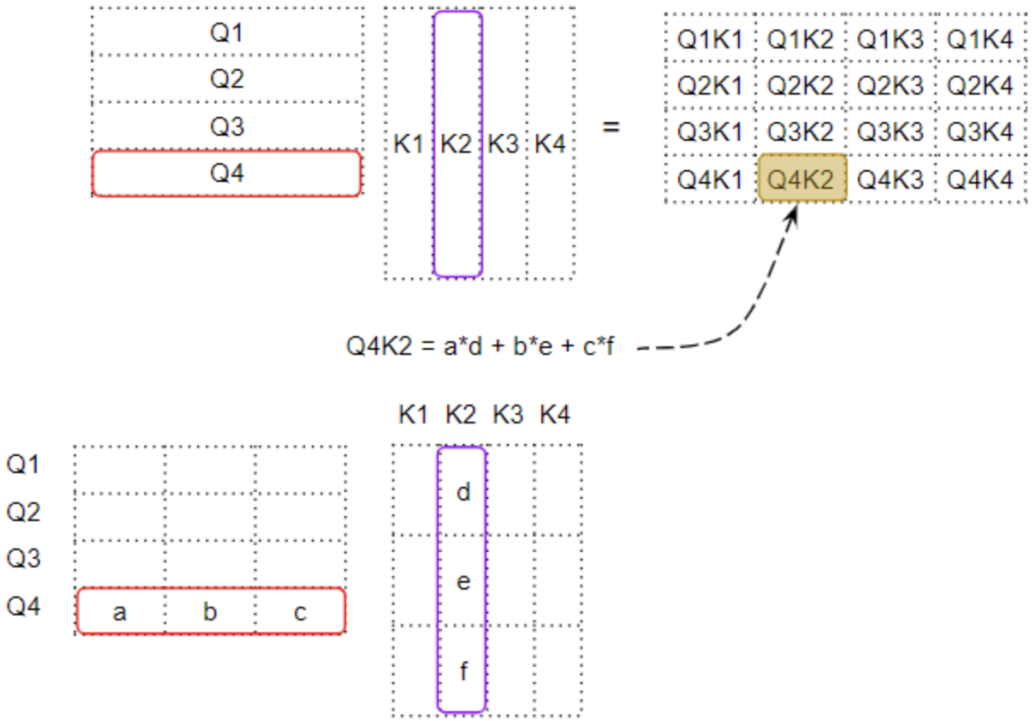
Dot Product between Query-Key and Value Matrices



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

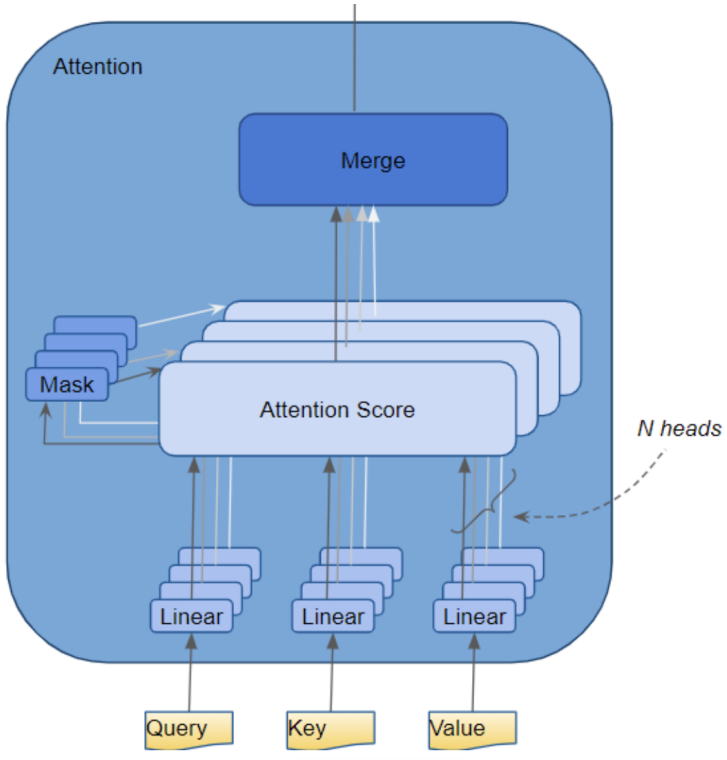
Each cell is a dot product between two word vectors



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-`

Multihead Attention: Visual Summary



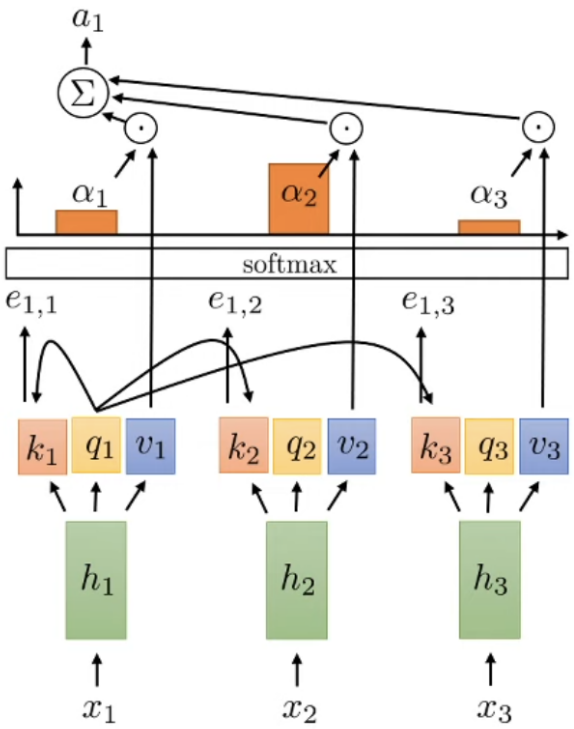
Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

Modeling word order: positional embedding

- Transformers models do not come with built-in information about sequence order such as time-step information in an RNN
- Transformers models do not have any notion of relative or absolute positions of the tokens in a sequence.
- In order to model word order, positional embeddings can be combined with input embeddings.
- Positional embeddings that are specific to each position in an input sequence.

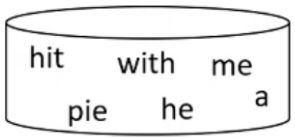
Positional encoding: what is the order?



what we see:

he hit me with a pie

what naïve self-attention sees:



a pie hit me with he

a hit with me he pie

he pie me with a hit

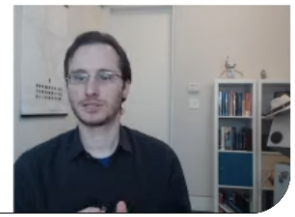
most alternative orderings are nonsense, but some change the meaning

in general the position of words in a sentence carries information!

Idea: add some information to the representation at the beginning that indicates where it is in the sequence!

$$h_t = f(x_t, t)$$

some function



Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

Positional encoding: sin/cos

Naïve positional encoding: just append t to the input $\quad \bar{x}_t = \begin{bmatrix} x_t \\ t \end{bmatrix}$

This is not a great idea, because **absolute** position is less important than **relative** position

I walk my dog every day

every single day I walk my dog

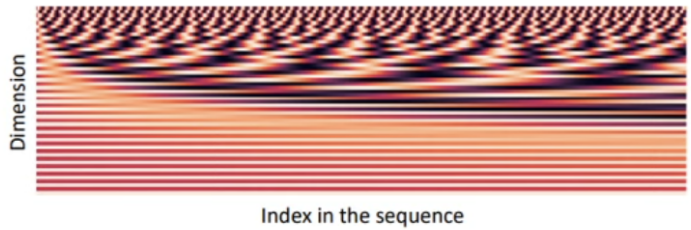
The fact that "my dog" is right after "I walk" is the important part, not its absolute position

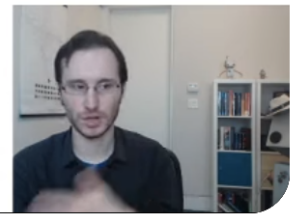
we want to represent **position** in a way that tokens with similar **relative** position have similar **positional encoding**

Idea: what if we use **frequency-based** representations?

$$p_t = \begin{bmatrix} \sin(t/10000^{2*1/d}) \\ \cos(t/10000^{2*1/d}) \\ \sin(t/10000^{2*2/d}) \\ \cos(t/10000^{2*2/d}) \\ \dots \\ \sin(t/10000^{2*\frac{d}{2}/d}) \\ \cos(t/10000^{2*\frac{d}{2}/d}) \end{bmatrix}$$

dimensionality of positional encoding

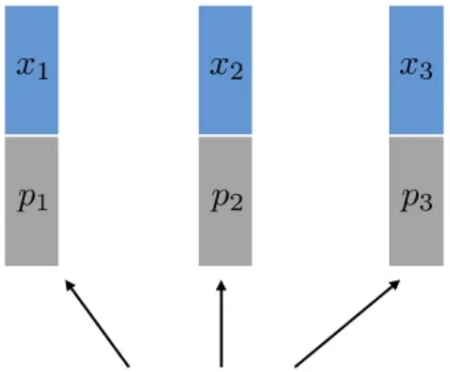




Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

Positional encoding: learned

Another idea: just learn a positional encoding



Different for every input sequence

The same learned values for every sequence

but different for different time steps

dimensionality

max sequence length

How many values do we need to learn?

$$P = [p_1, p_2, \ldots, p_T] \in R^{d \times T}$$

+ more flexible (and perhaps more optimal) than sin/cos encoding

+ a bit more complex, need to pick a max sequence length (and can't generalize beyond it)

Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

How to incorporate positional encoding?

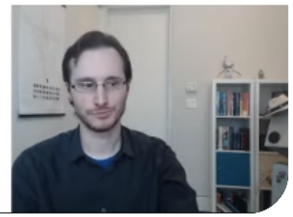
At each step, we have x_t and p_t

Simple choice: just concatenate them $\bar{x}_t = \begin{bmatrix} x_t \\ p_t \end{bmatrix}$

More often: just add after **embedding** the input

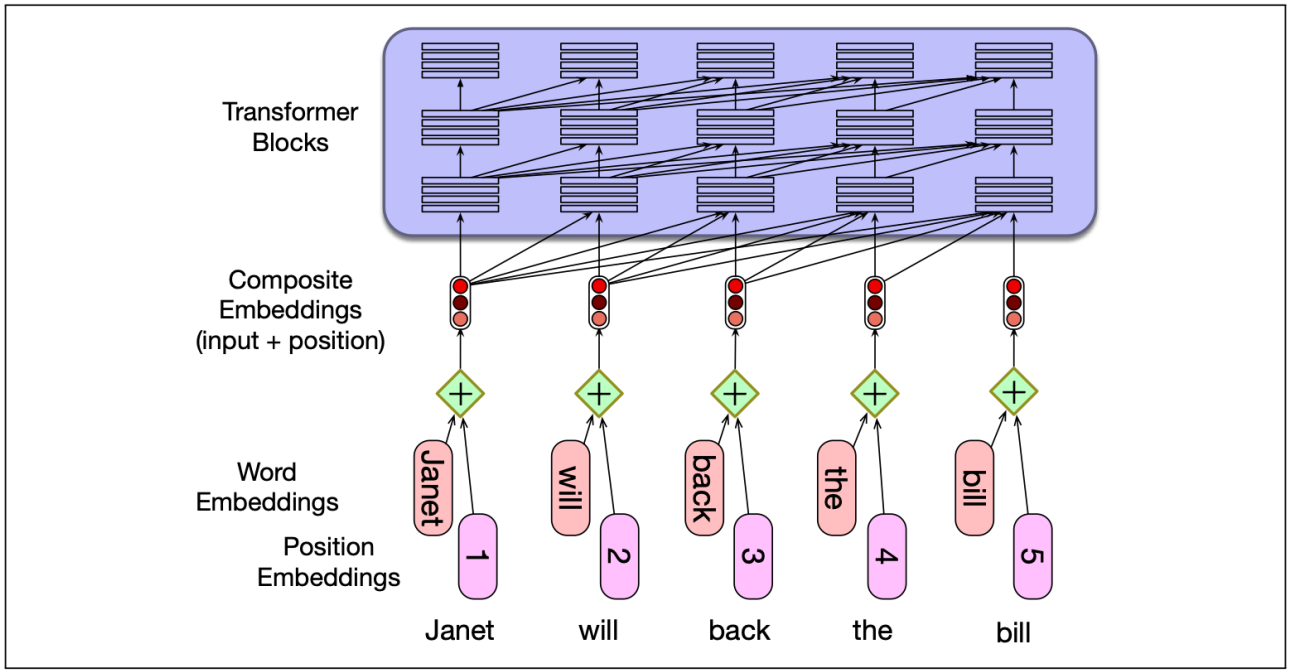
input to self-attention is $\text{emb}(x_t) + p_t$

some learned function (e.g., some fully connected layers with linear layers + nonlinearities)

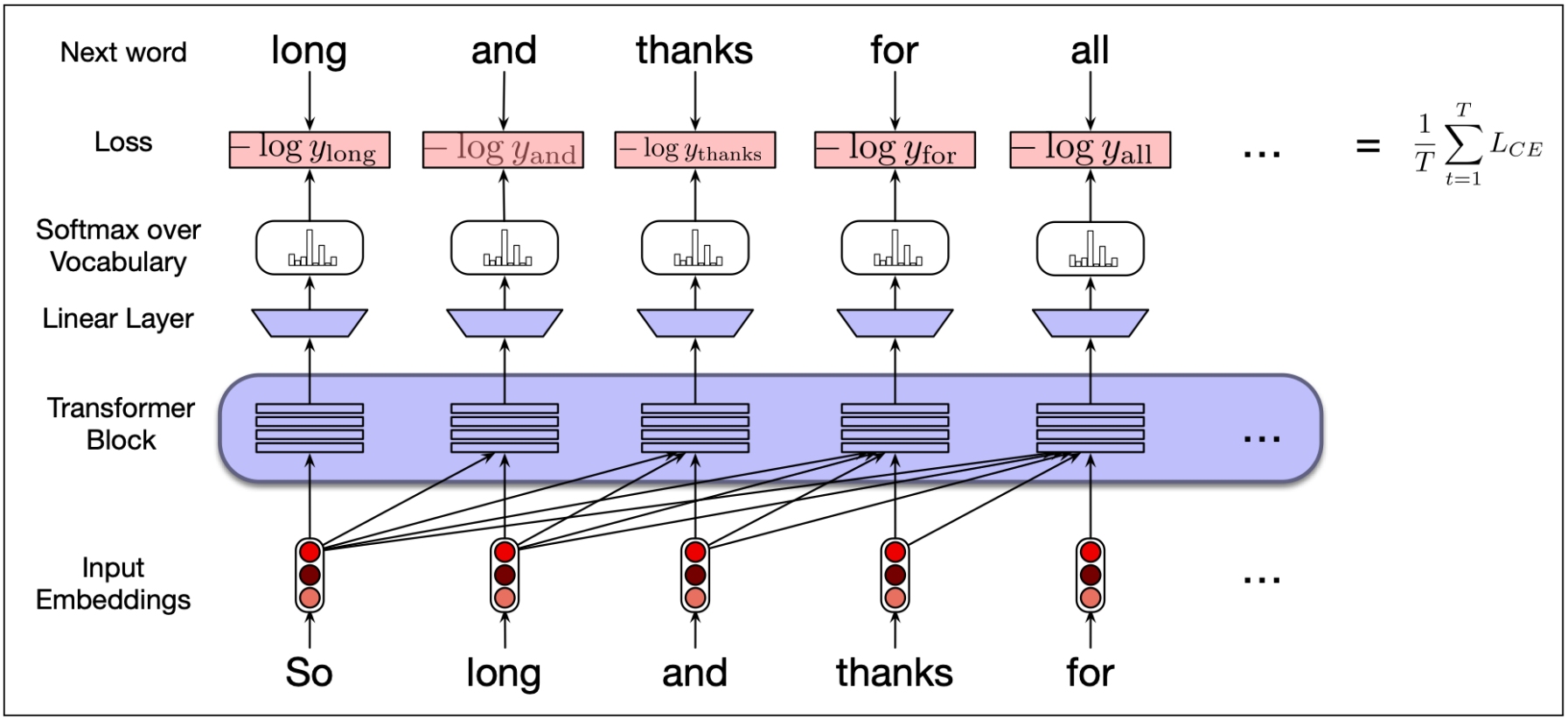


Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

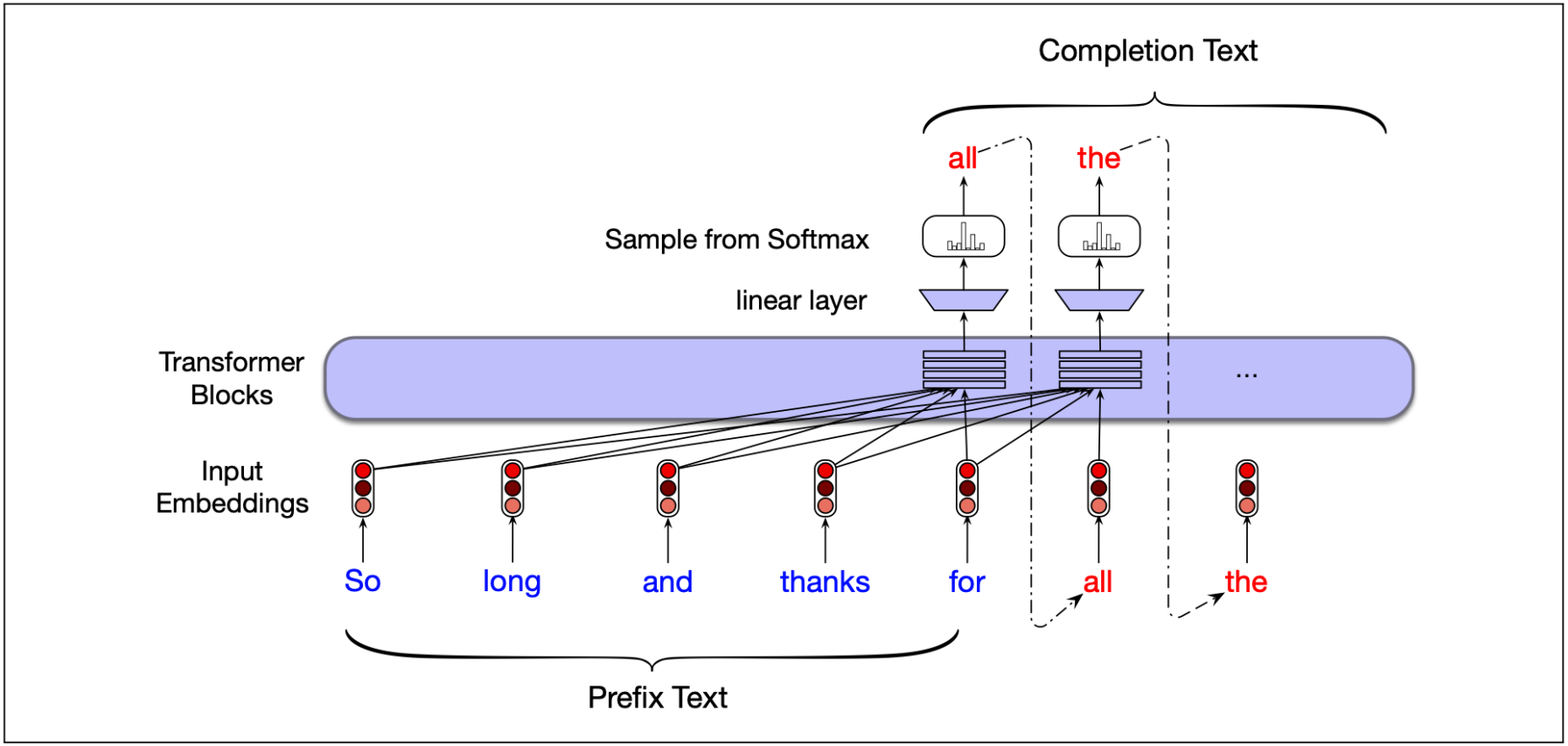
Multihead Attention



Transformers as Language Models



Contextual Generation and Summarization



Contextual Generation and Summarization

Original Article

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states. “We will not ship snow to any states in the northeast!” says Waring’s website, ShipSnowYo.com. “We’re in the business of expunging snow!”

Contextual Generation and Summarization

Continued

His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

Contextual Generation and Summarization

Continued

According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. His business slogan: "Our nightmare is your dream!" At first, ShipSnowYo sold snow packed into empty 16.9-ounce water bottles for $19.99, but the snow usually melted before it reached its destination...

Contextual Generation and Summarization

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

Contextual Generation and Summarization