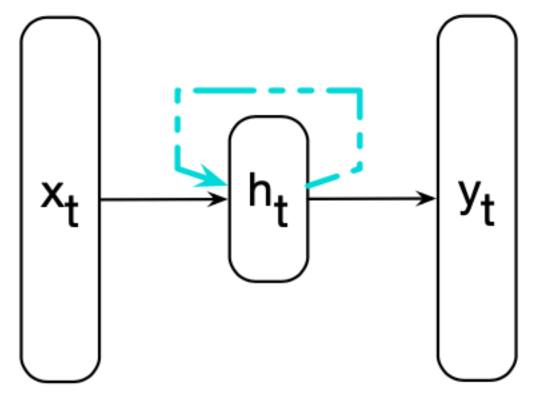
CMPE 297, INSTRUCTOR: JORJETA JETCHEVA

DEEP LEARNING ARCHITECTURES

RNN

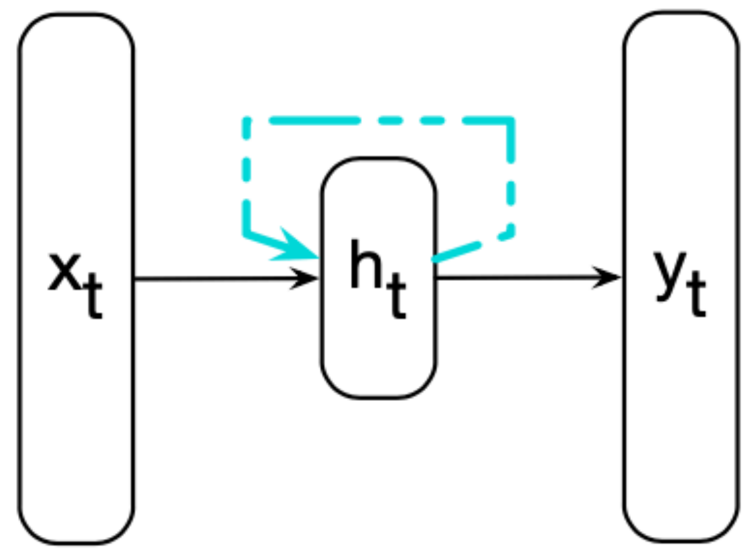
RECURRENT NEURAL NETWORKS (RNNS)

- An RNN is a neural network that has one or more cycles
- But RNNs with arbitrary connectivity are difficult to train and analyze
- So we typically focus on "constrained" RNNs
  - Simple RNN (or Elman Networks) covered here
  - LSTMs, etc.



Activation value depends on input from previous time step.

RNN OVERVIEW



- Network processes 1 input at a time
- Each step can be viewed as associated with a point in time t
- The recurrent connection can be viewed as "memory" because it allows us (in the current time step) to take into account for information learned from previous time steps

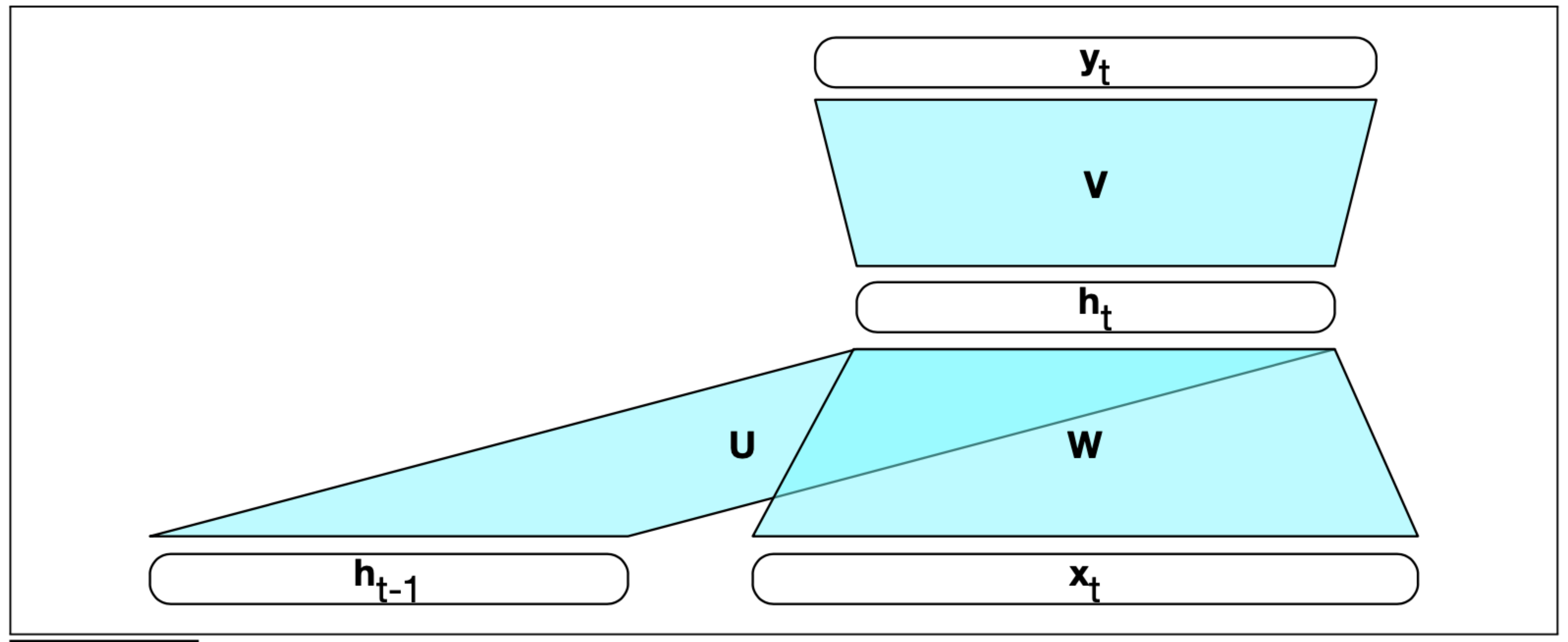


Figure 9.3 Simple recurrent neural network illustrated as a feedforward network.

Difference from Feed Forward network is the extra weight matrix U and the use of the activation from the previous time step h_{t-1}

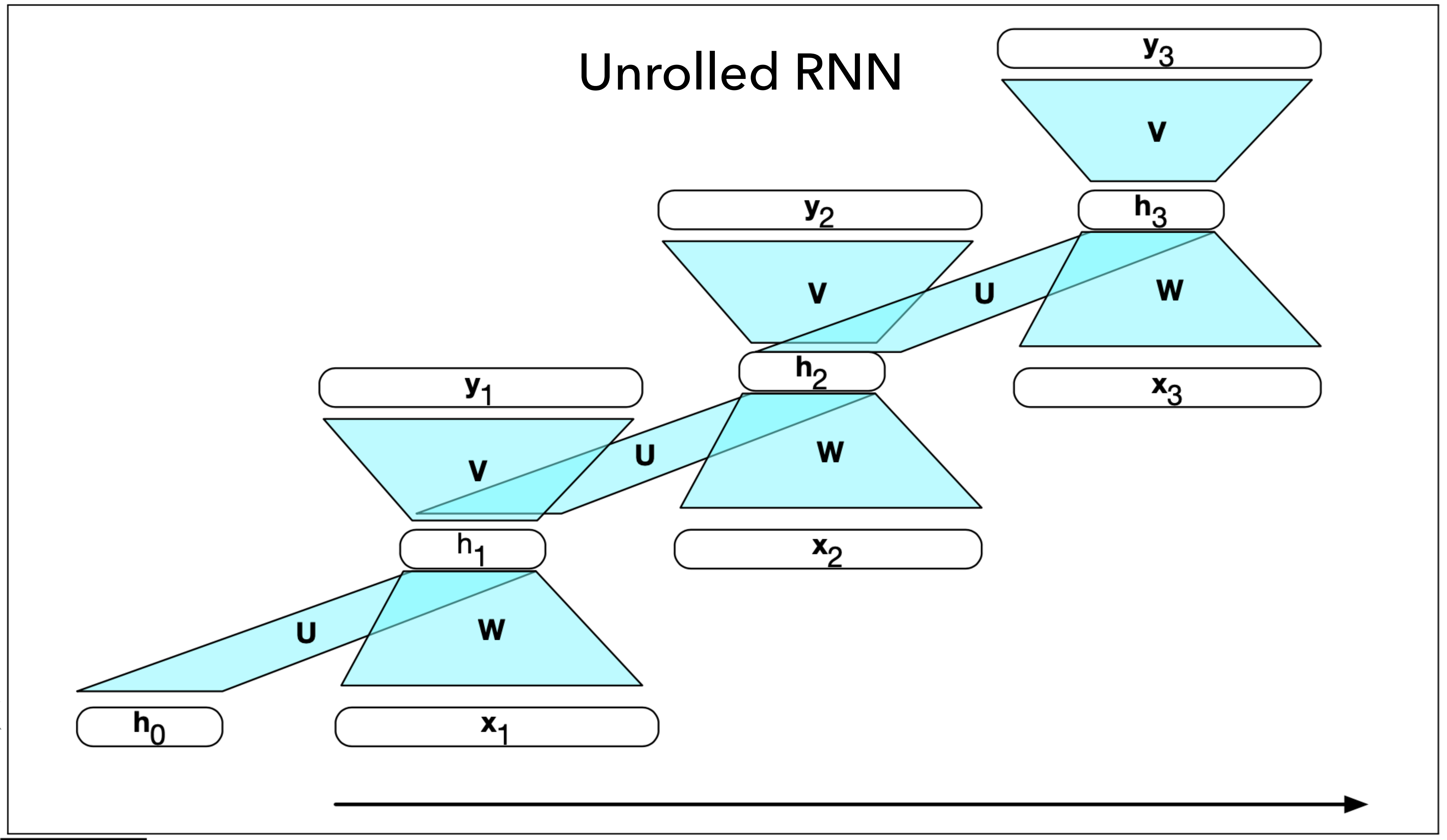


Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared in common across all time steps.

INFERENCE/PREDICTION & TRAINING IN A RNN

- Given an input sequence, we can unroll the network and perform both prediction (inference) and training the same way as we would for a feed forward network
- Or do the unrolling and training on segments (batches) when the input is large

The computation performed in the RNN is as follows:

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

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

Typically, *f* is the softmax function and we have

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

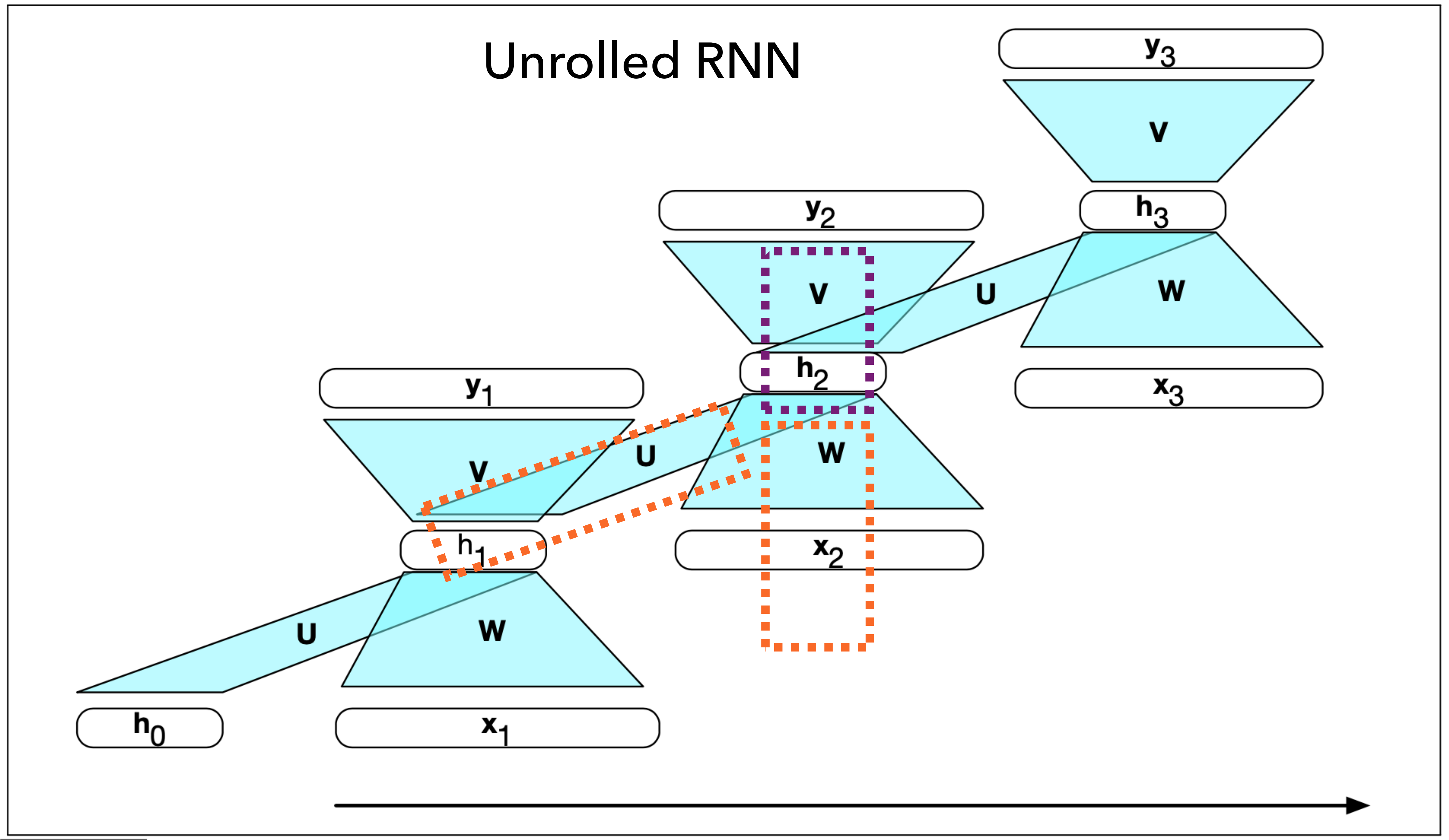


Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared in common across all time steps.

RNN FOR LANGUAGE MODELING

- **Input**: a series of 1-hot vectors for the input sequence of words, indexing into the embedding matrix E
- **Output**: probability of each vocabulary word at each time step
- We use cross-entropy (CE) loss

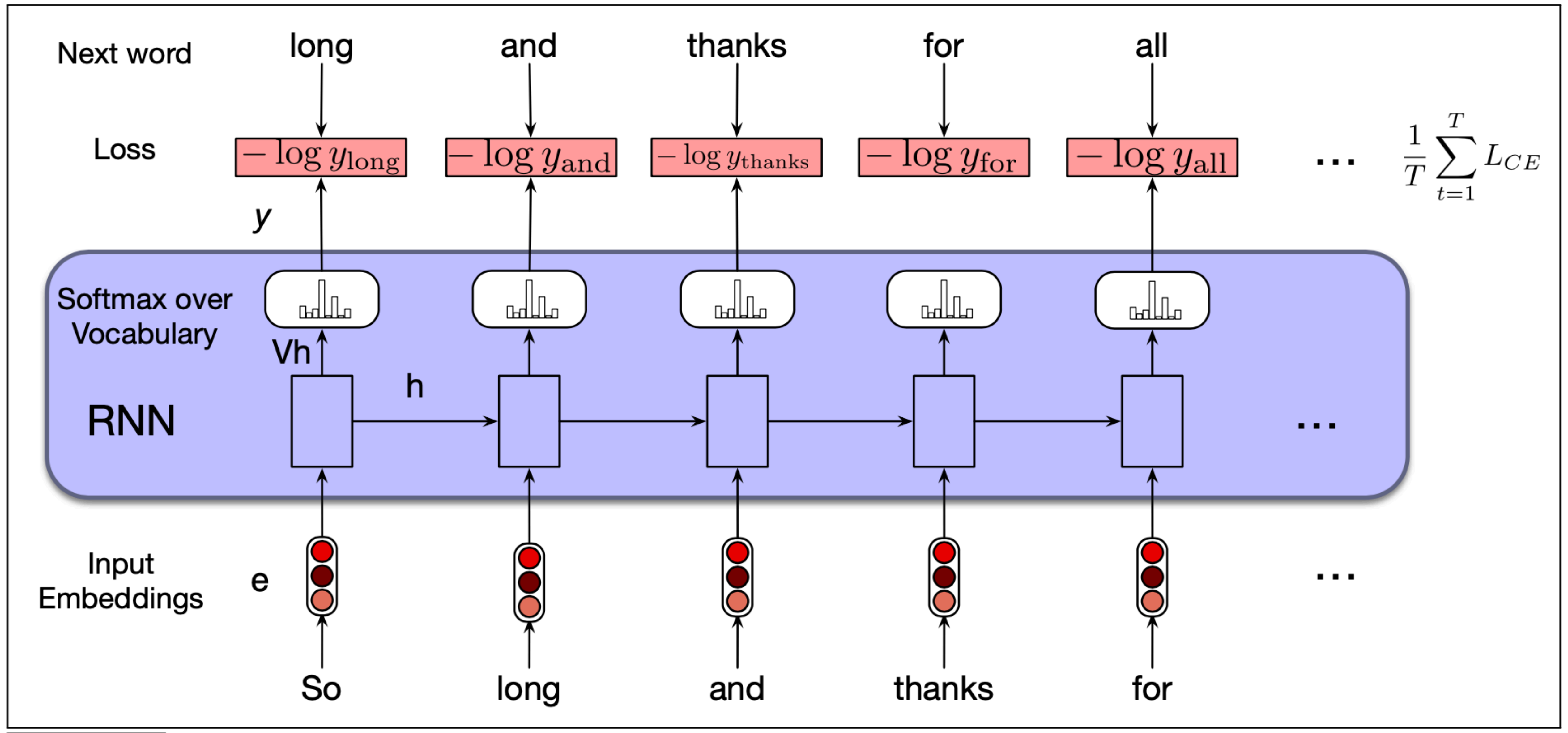


Figure 9.6 Training RNNs as language models.

Teacher forcing-based approach: that during training, we give the model the correct next word at each time step (the correct history sequence), rather than proceeding from what is predicted in the previous time step

RNN FOR SEQUENCE LABELING

- Part-of-Speech (PoS) tagging RNNs are structured the same way as RNNs for language modeling
- **Output**: probability of each PoS tag at each time step

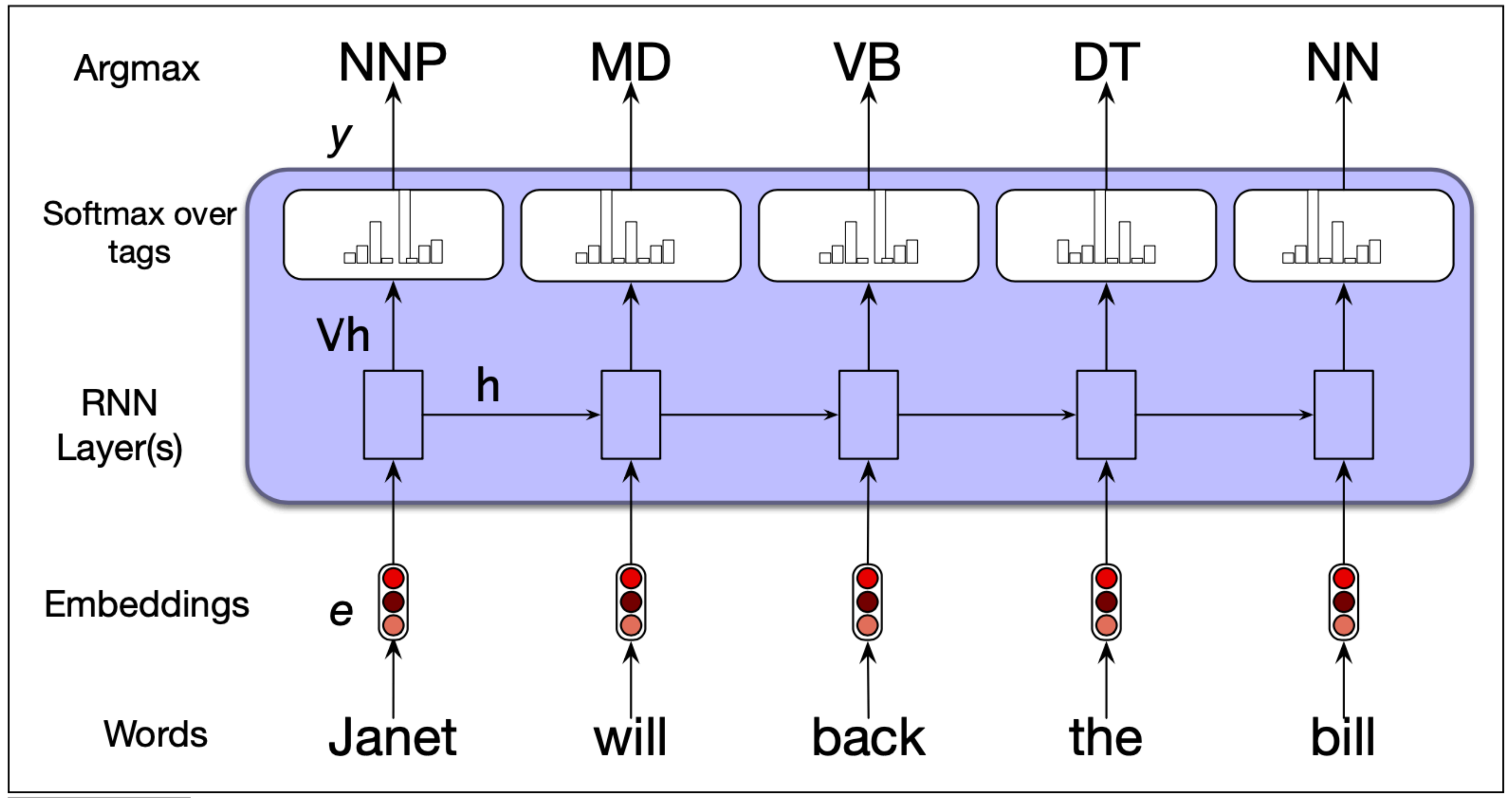


Figure 9.7 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

RNN FOR SEQUENCE CLASSIFICATION

- We pass a sequence of words, which is then classified as a whole, e.g., in sentiment classification, spam, document-level topic classification
- The hidden layer for the last token (h_n) serves as the input layer to a feed forward network (FFN), which then performs the classification
  - h_n is a compressed representation of the input sequence of words
  - Sometimes we might instead use the element-wise mean or max of all the hidden state matrices for each word in the sequence (pooling): $\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i$

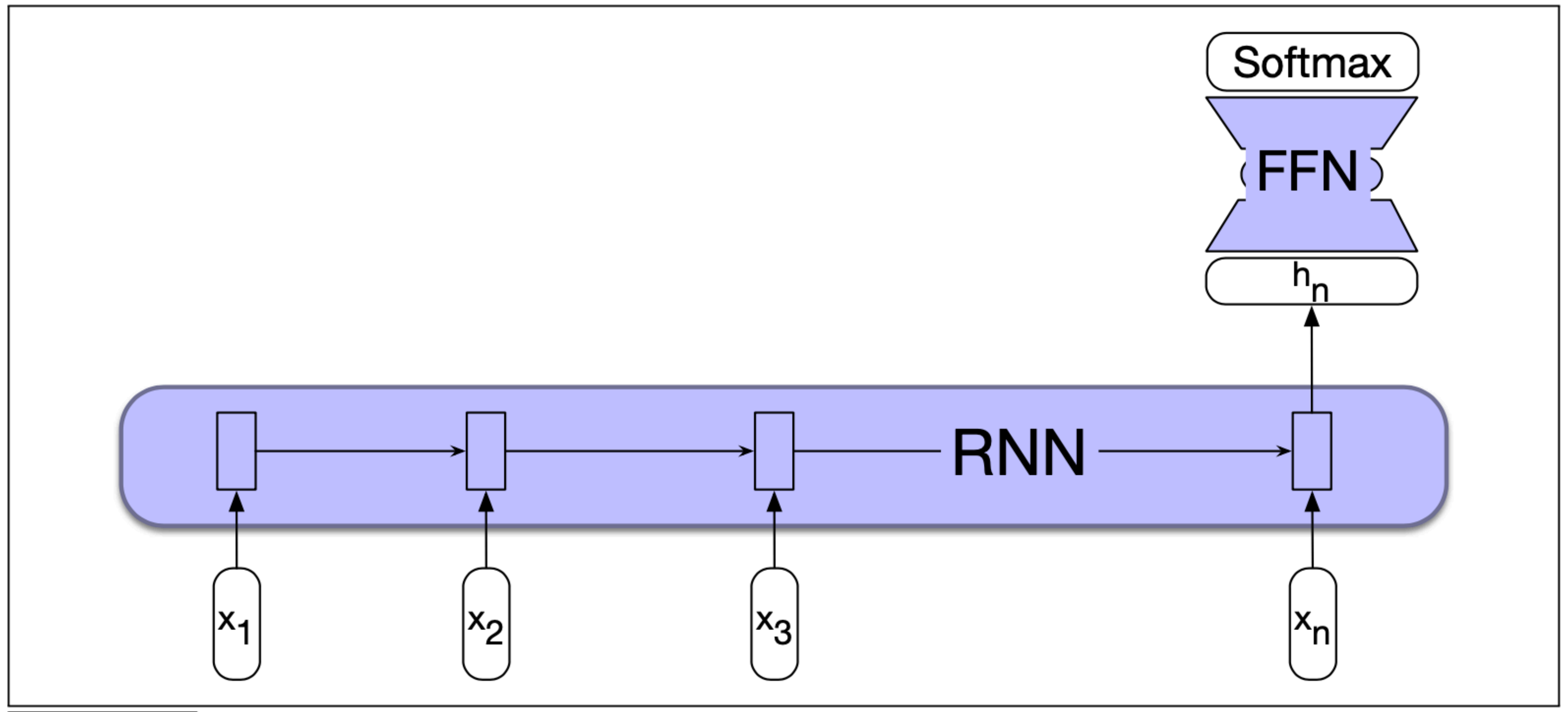


Figure 9.8 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

- Loss is only computed with respect to the output layer of the entire RNN+FFN network - this is called end-to-end training (the loss from the final (downstream application) is used to adjust the weights across the entire network

TEXT GENERATION

- Autoregressive Generation: Repeatedly sample the next word, conditioned on previous word choices
  - Start with the beginning of sentence marker <s>
  - Generate words until the end of sentence marker </s> is generated
- For translation tasks, we can start with the sentence we want to translate (instead of <s>)
- For summarization tasks, we can start with the text we want to summarize (instead of <s>)

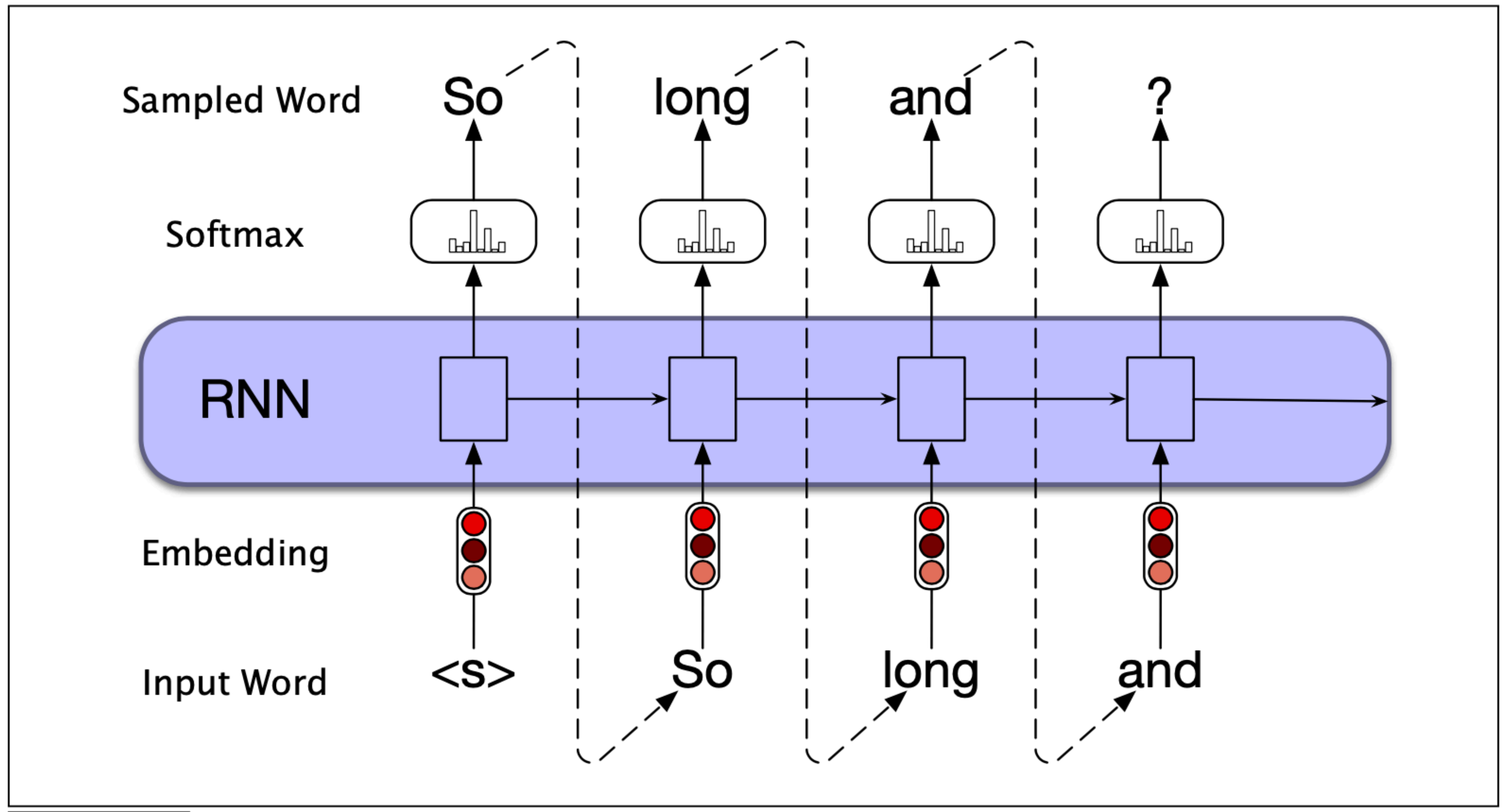


Figure 9.9 Autoregressive generation with an RNN-based neural language model.

EXTENDED RNN ARCHITECTURES

STACKED RNN

- The output layer of one RNN becomes an input layer for another RNN
- Stacked RNNs tend to outperform individual RNNs, though are more computationally intensive to train
- Intuition behind their effectiveness is that each level benefits from the abstraction created of the previous layer

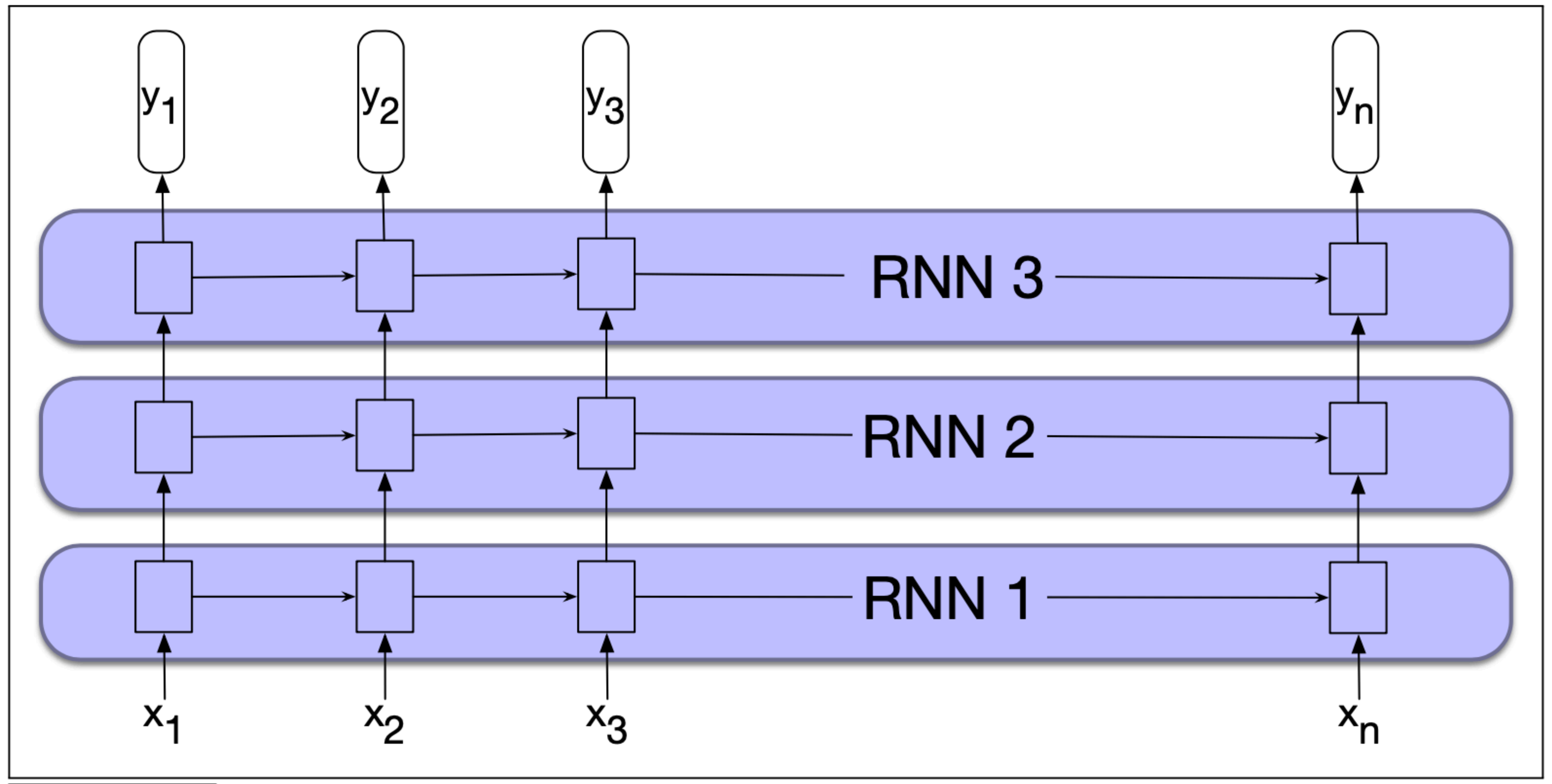


Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

BIDIRECTIONAL RNNS

- A bidirectional RNN concatenates two independent RNNs
  - Forward RNN processes input from start to current time t: $\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t)$
  - Backward RNN processes input from end to current time t: $\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_n, \ldots, \mathbf{x}_t)$
- We concatenate the forward and backward representations: $\mathbf{h}_t = [\mathbf{h}_t^f; \mathbf{h}_t^b]$
- This way for each time point, we capture both the left and right contexts for a particular input sequence

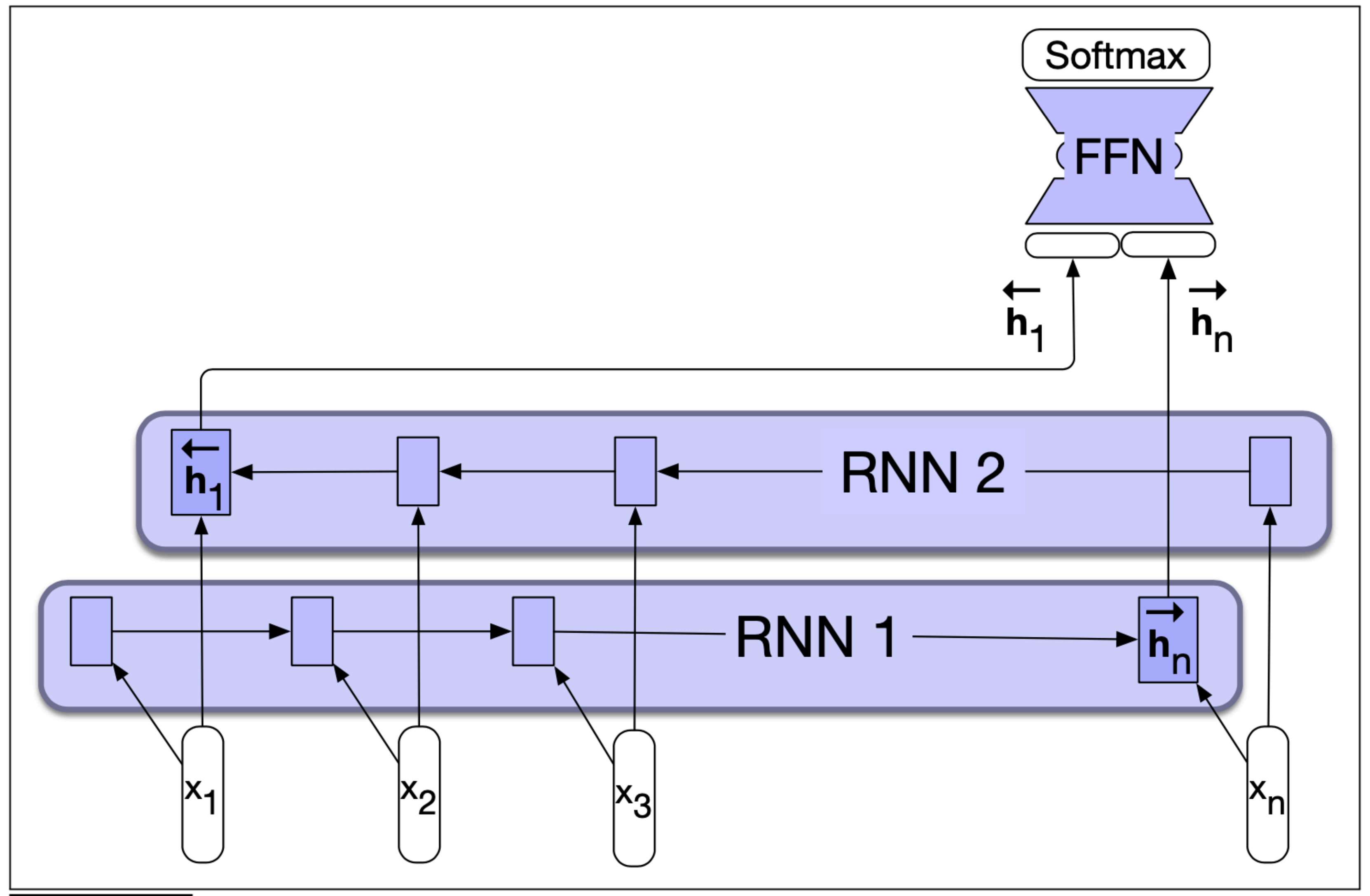


Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

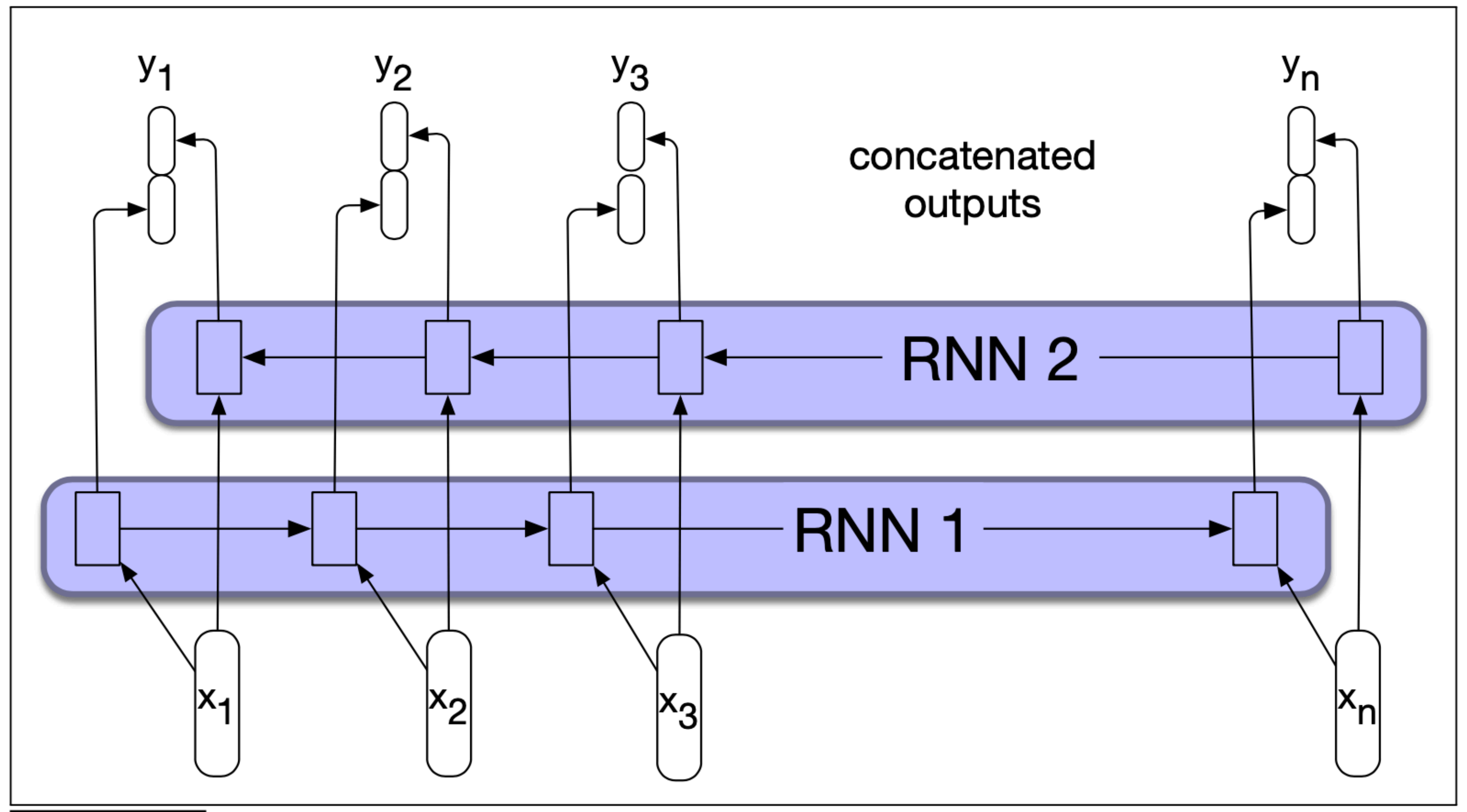


Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

LSTM

SHORTCOMINGS OF RNN

- Information encoded in the hidden state is recent - they tend to forget quickly
  - Example: The flights the airline was cancelling were full.
- Reasons:
  - RNN us trying to do two things at once: provide useful information for current decision/output, and also carry information forward for future decisions
  - Vanishing gradient problem due to many multiplication operations at the hidden layer
- What to do? Create a NN where there is more explicit control over information that needs to be remembered, and information that’s no longer needed

LSTM OVERVIEW

- Long shorter-term memory (LSTM) network
- LSTMs have gates that control the flow of information (context) through the network
- Each gate is a feed forward network with a sigmoid activation function
- The output of the gate is multiplied point-wise with the layer to which the gate is applied
    - Gates end up having the effect of a binary mask (the sigmoid tends to produce values close to 0 and 1)

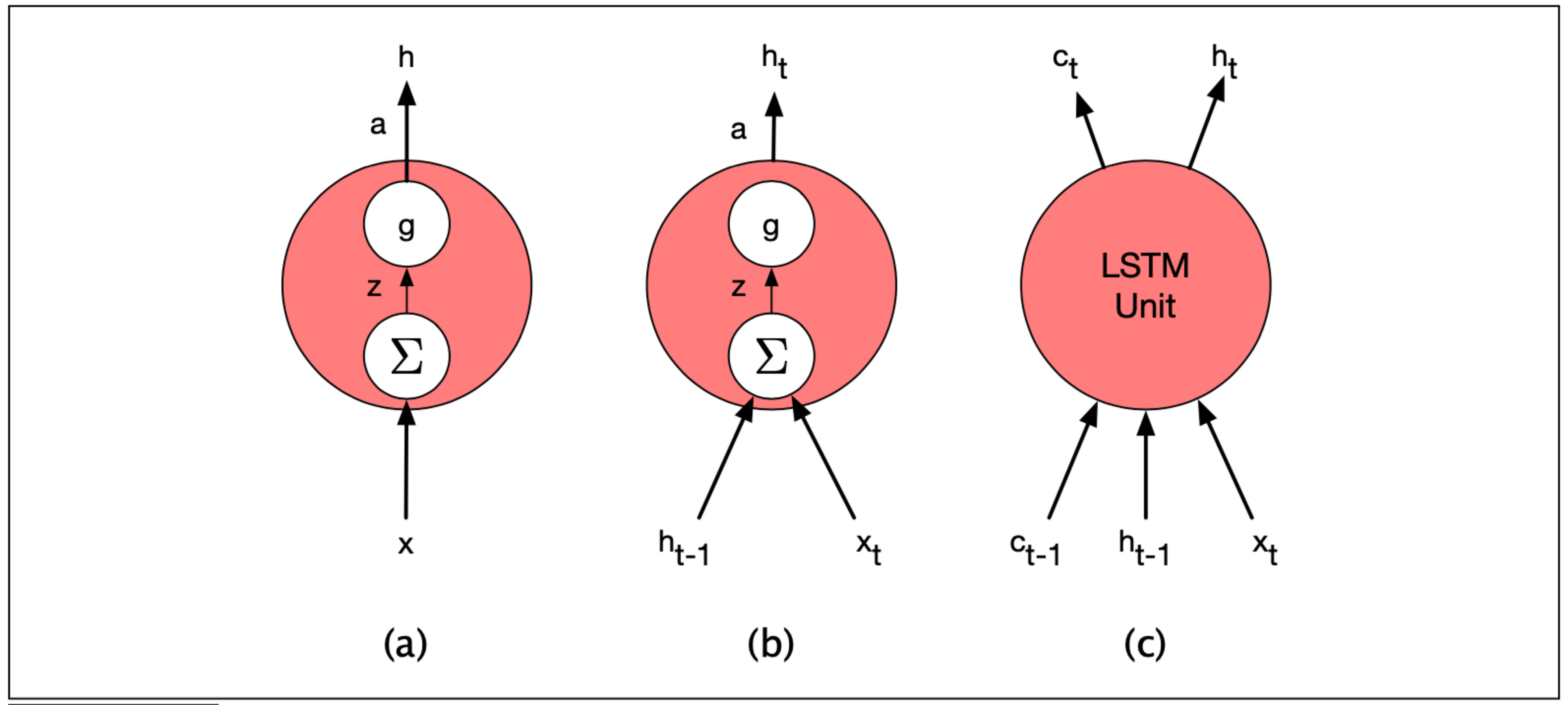


Figure 9.14 Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

GATES

- Forget gate - delete information from the context that is no longer needed

$$\mathbf{f}_t = \sigma(\mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{x}_t)$$
Weighted sum of the previous state's hidden layer ($\mathbf{h}_{t-1}$) and the current input ($\mathbf{x}_t$)

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$$
Element-wise multiplication with the context vector from the previous state, $\mathbf{c}_{t-1}$

- Add gate - select information to add to the current context

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t)$$
Weighted sum of the previous state's hidden layer ($\mathbf{h}_{t-1}$) and the current input ($\mathbf{x}_t$)

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$$
The context modification $\mathbf{j}_t$ is computed as an element-wise multiplication between the current activation (at time t) and the input gate vector

Where $\mathbf{g}_t$ is the activation computed at time t: $\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t)$

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$
We update the context by adding the information we want to add ($\mathbf{k}_t$ is the current context with the information removed by the forget gate already)

GATES (CONT.)

- Output gate - used to decide what information is needed for the current hidden state (rather than preserved for future states)

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

Weighted sum of the previous state's hidden layer ($\mathbf{h}_{t-1}$) and the current input ($\mathbf{x}_t$)

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

Extracting from the context information that is needed for the current hidden state (tanh turns the context into a binary mask)

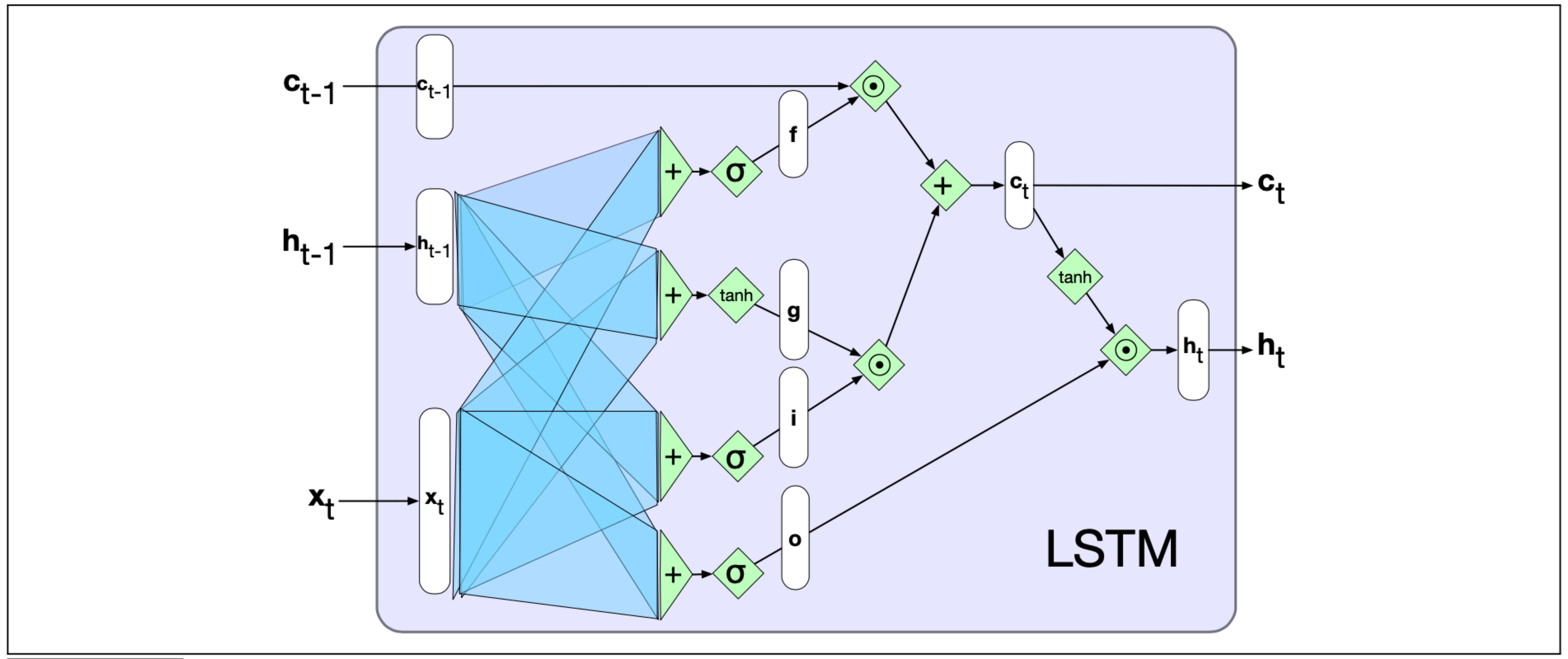


Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

Network can be unrolled and trained same as we discussed for RNNs

TRANSFORMERS

CHALLENGES OF RNNS

- Cannot handle long context - information loss
- Difficulties in training (vanishing gradient along long paths)
- Not possible to parallelize the computation
- These challenges led to the development of Transformers, which do not have recurrent connections

TRANSFORMER OVERVIEW

- Transformers map a sequence of inputs $x_1, \ldots, x_n$ to a sequence of outputs $y_1, \ldots, y_n$
- Transformers are stacks of transformer blocks
- Each transformer block has simple linear layers, feed forward network layers, and self-attention layers

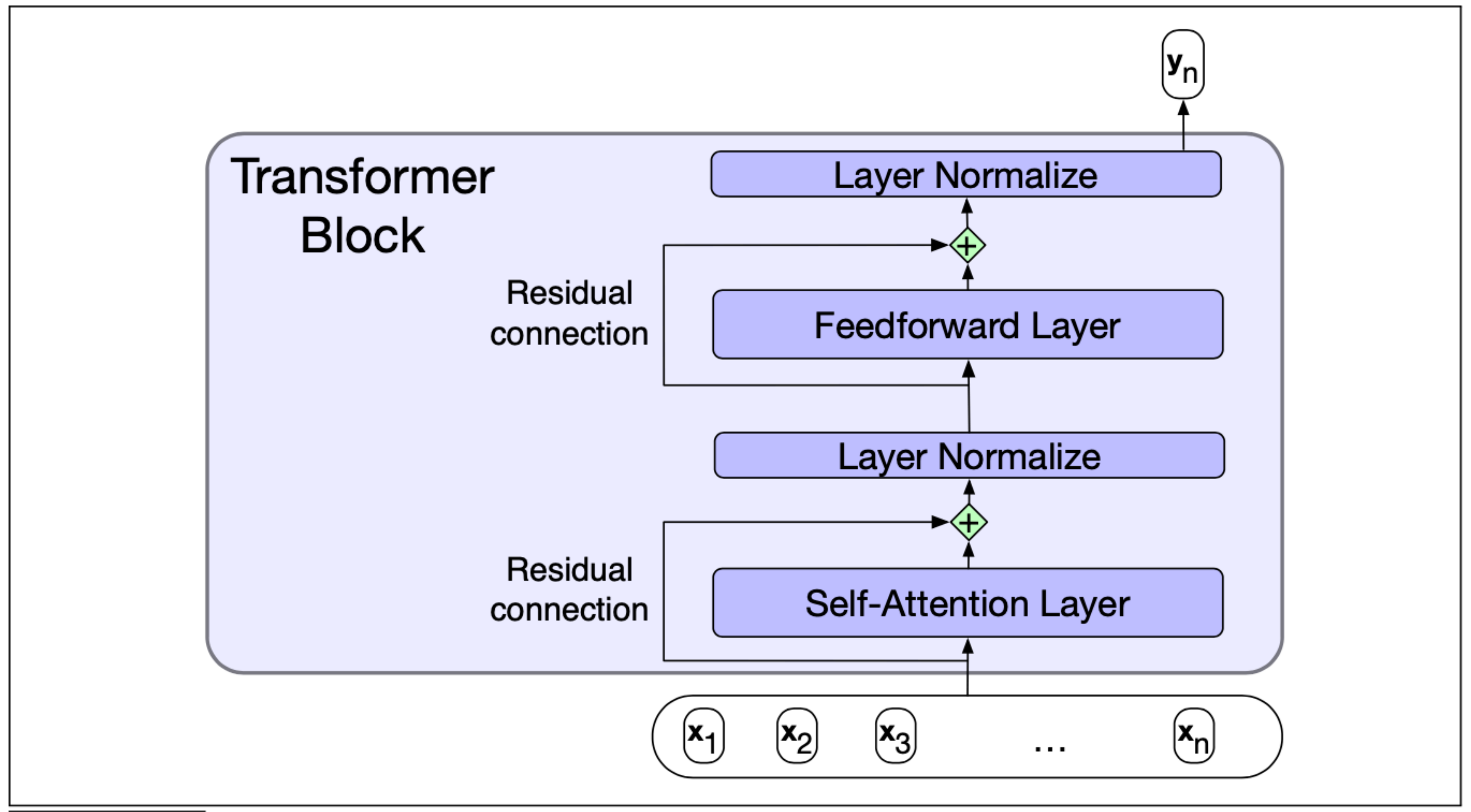


Figure 9.18 A transformer block showing all the layers.

BASIC SELF-ATTENTION

- Self-attention is the major innovation of transformers
- It allows extracting information from arbitrarily large contexts, without needing to pass it along recurrent connections
- The approach is based on comparing an item to a collection of other items to determine their relevance to the current item
  - E.g., in the figure, the computation of $\mathbf{y}_3$ is based on comparing $\mathbf{x}_3$ to its *preceding* two inputs, $\mathbf{x}_2$ and $\mathbf{x}_1$
- Comparison can be done through computing a similarity score using a dot product (e.g., as in Cosine Similarity), e.g., $score(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$
- The higher the score, the more similar the items

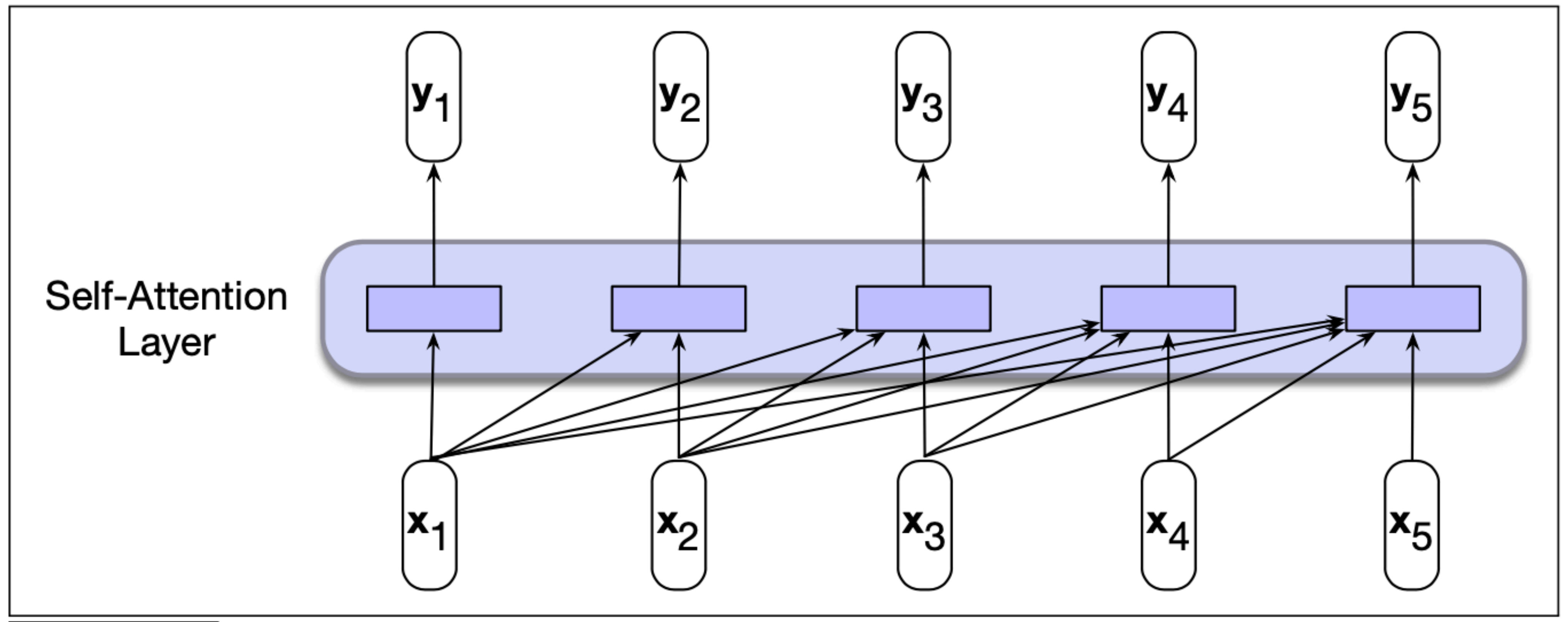


Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

BASIC SELF-ATTENTION (CONT.)

- To convert the similarity output to probability, we can pass the score through a softmax function:

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

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

- The outputs can be generated by summing the inputs weighted by their respective α values:

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} x_j$$

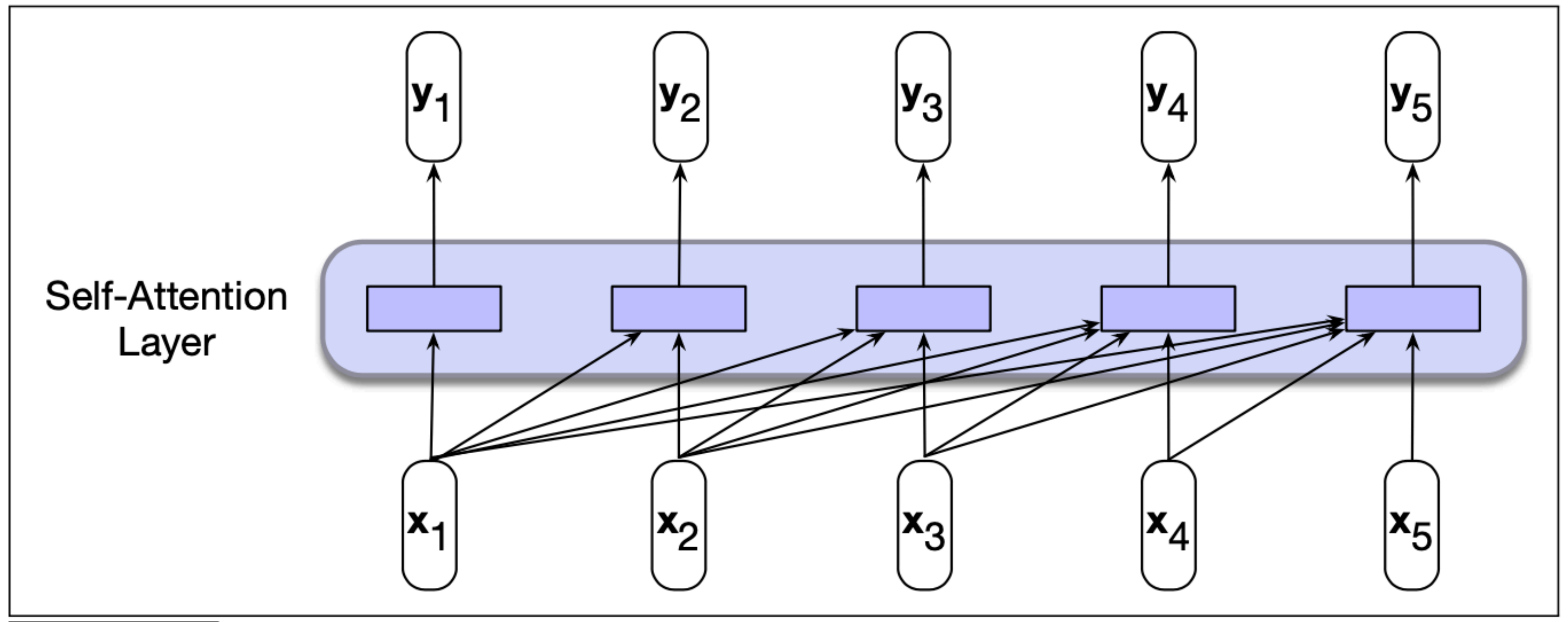


Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

FULL COMPUTATION FOR SELF-ATTENTION FOR A SINGLE OUTPUT

- Each input embedding plays 3 roles:
  - Query (Q): when it is the current focus of attention (and being compared to prior inputs)
  - Key (K): as a preceding input to the current focus of attention
  - Value (V): used to compute the output for the current focus of attention
- Transformers have a weight matrix for each of these roles: $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$
- We use these matrices to project the input into vectors that represent its 3 roles:

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

(x & y have dimensionality $1 \times d$, and the weight matrices have dimensions $d \times d$)

- Therefore, we have

$$score(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j \text{ and } \mathbf{y}_i = \sum_{j \le i} \alpha_{ij} v_j$$

- To avoid large values, we scale the computation:

$$score(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \text{ where } d_k \text{ is the size of the key vector}$$

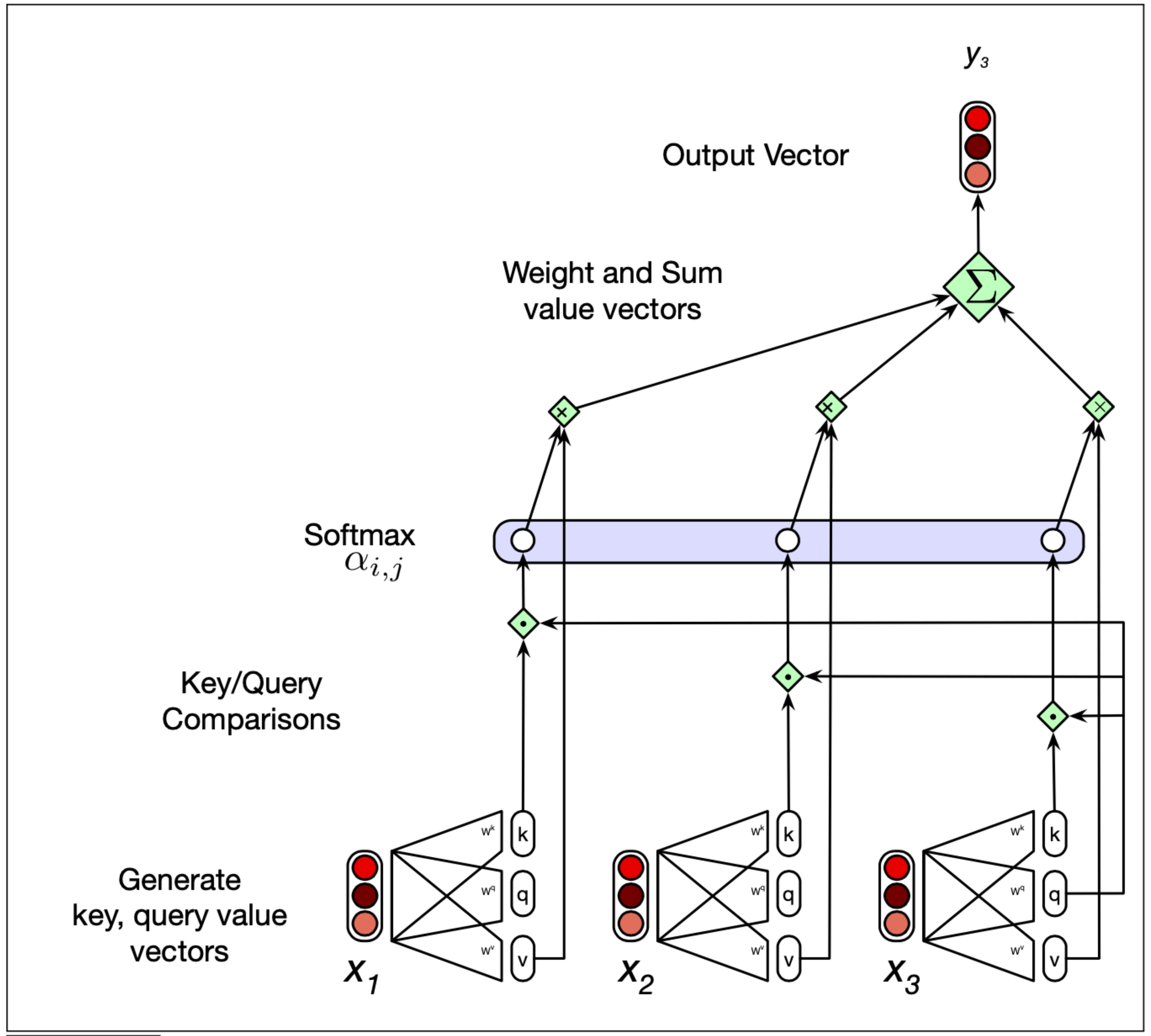


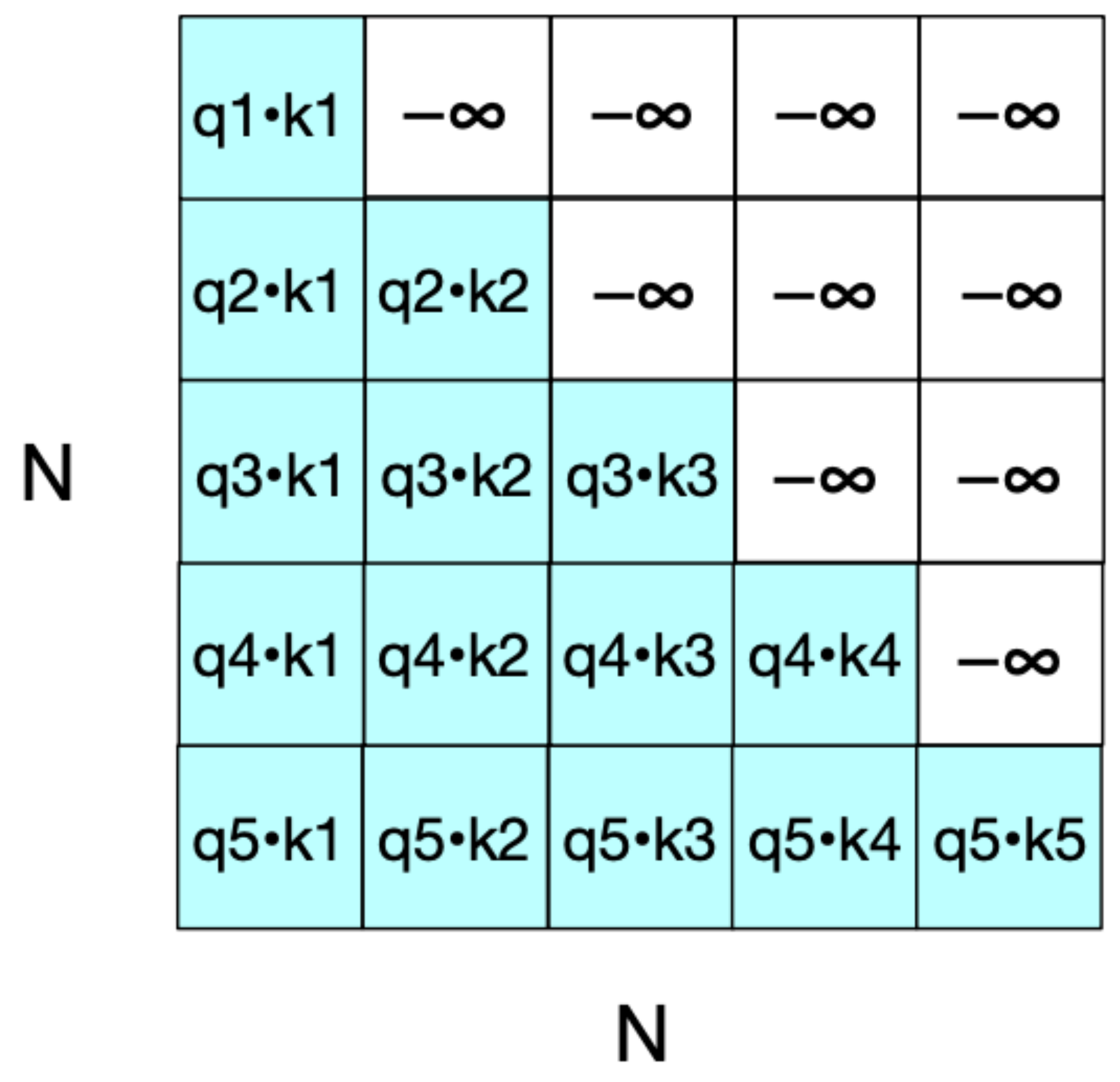
Figure 9.16 Calculating the value of $\mathbf{y}_3$, the third element of a sequence using causal (left-to-right) self-attention.

FULL COMPUTATION FOR SELF-ATTENTION FOR ALL OUTPUTS

- The computation for the whole data is:
  $\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$
- And the self-attention α for a sequence of N tokens can be computed with the following matrix operation:

$$SelfAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}$$

- We need to remove the values in the upper part of the matrix because they correspond to comparisons of inputs that follow the query
- Most applications limit the length of the input (number of tokens) to a paragraph or a page (rather than whole document), because this N by N computation of dot products gets very expensive



$$\begin{array}{c|ccccc} & & & & & \\ & q1\cdot k1 & -\infty & -\infty & -\infty & -\infty \\ & q2\cdot k1 & q2\cdot k2 & -\infty & -\infty & -\infty \\ N & q3\cdot k1 & q3\cdot k2 & q3\cdot k3 & -\infty & -\infty \\ & q4\cdot k1 & q4\cdot k2 & q4\cdot k3 & q4\cdot k4 & -\infty \\ & q5\cdot k1 & q5\cdot k2 & q5\cdot k3 & q5\cdot k4 & q5\cdot k5 \\ & & & N & & \end{array}$$

TRANSFORMER BLOCKS: BASIC ILLUSTRATION

- Residual connections are another element of Transformers and have been found to improve learning
  - Enable both activations and gradients to skip layers
  - Implementation involves adding a layer's input vector to its output vector before passing the result as an input to the next layer
  - The vectors are then normalized (by the Layer Normalize layer (layer norm)) - normalizing internal layer outputs is a common way of improving gradient-based training
- The overall computation can be expressed as follows:

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

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

Normalization uses the standardization formula $\hat{\mathbf{x}} = \frac{(x - \mu)}{\sigma}$, which results in a new vector with 0 mean, and standard deviation equal to 1.

Layer normalization typically includes two learnable parameters γ and β:
$LayerNorm = \gamma \hat{\mathbf{x}} + \beta$

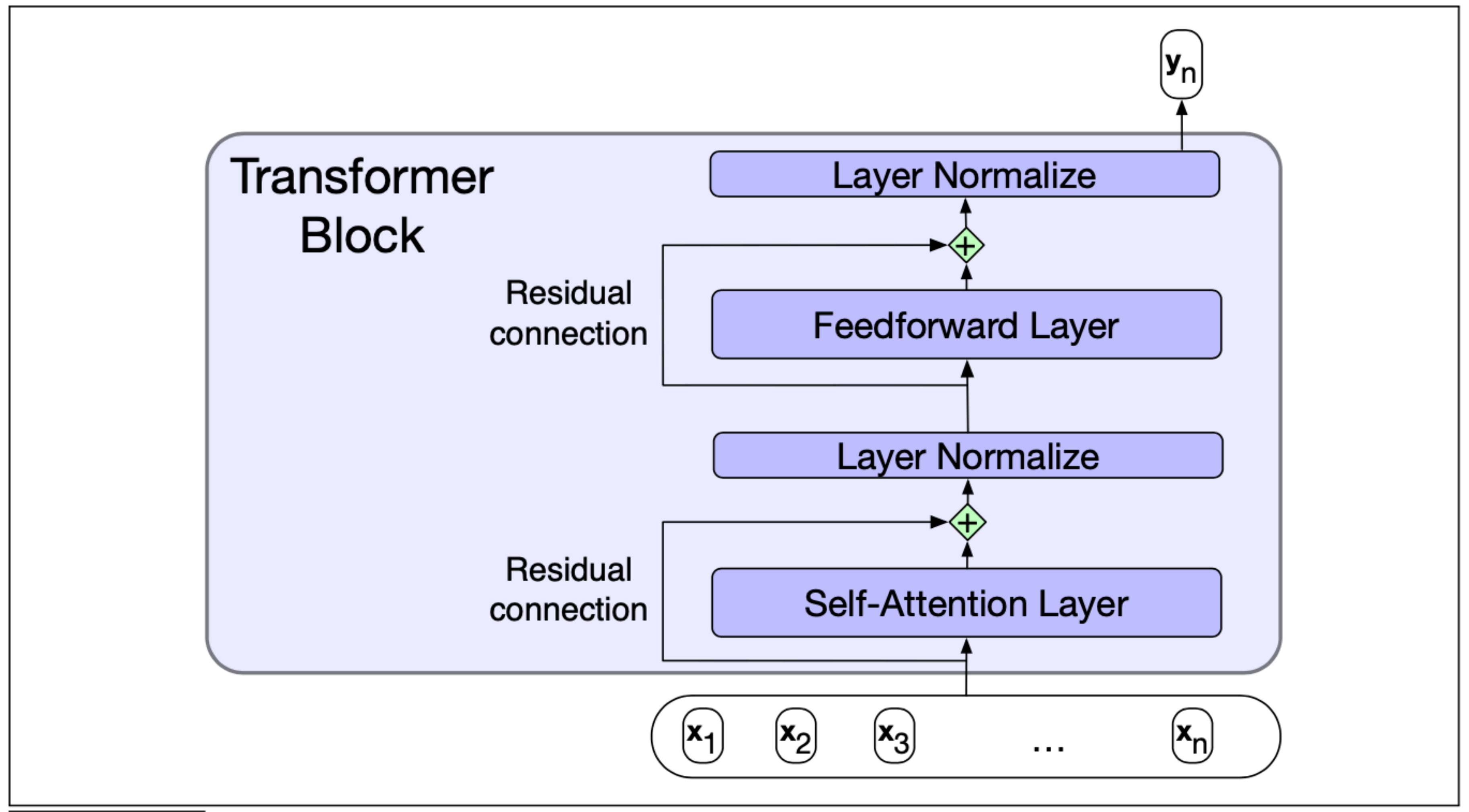


Figure 9.18 A transformer block showing all the layers.

MULTIHEAD ATTENTION

- A single attention layer only captures one type of relationship between the inputs
- However, there may be many levels at which the inputs interact, e.g., different syntactic relationships, or semantic relationships
- To capture multiple relationship types, we can use multihead attention - these are stacks of multiple attention layers
  - Each attention layer is referred to as a "head"
  - Each head learns a different aspect of the relationship between the inputs

MULTIHEAD ATTENTION (CONT.)

- Each attention layer i has its own matrices for key, query, and value: $\mathbf{W}_i^K, \mathbf{W}_i^Q, \mathbf{W}_i^V$
- These result in different projections of the input for each of the heads
- The outputs of all the layers are then concatenated
- We need to do a final transformation to bring this concatenated output of the multi-headed layer to the same dimension as the input
  - The dimensions of the heads and the concatenated output of the heads are typically different from the input dimension
  - Overall computation for a single output:

$$\begin{aligned} MultiHeadAttn(\mathbf{X}) &= (\mathbf{head}_1 \oplus \mathbf{head}_2 ... \oplus \mathbf{head}_h)\mathbf{W}^O \\ \mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q \; ; \; \mathbf{K} &= \mathbf{X}\mathbf{W}_i^K \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \\ \mathbf{head}_i &= SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) \end{aligned}$$

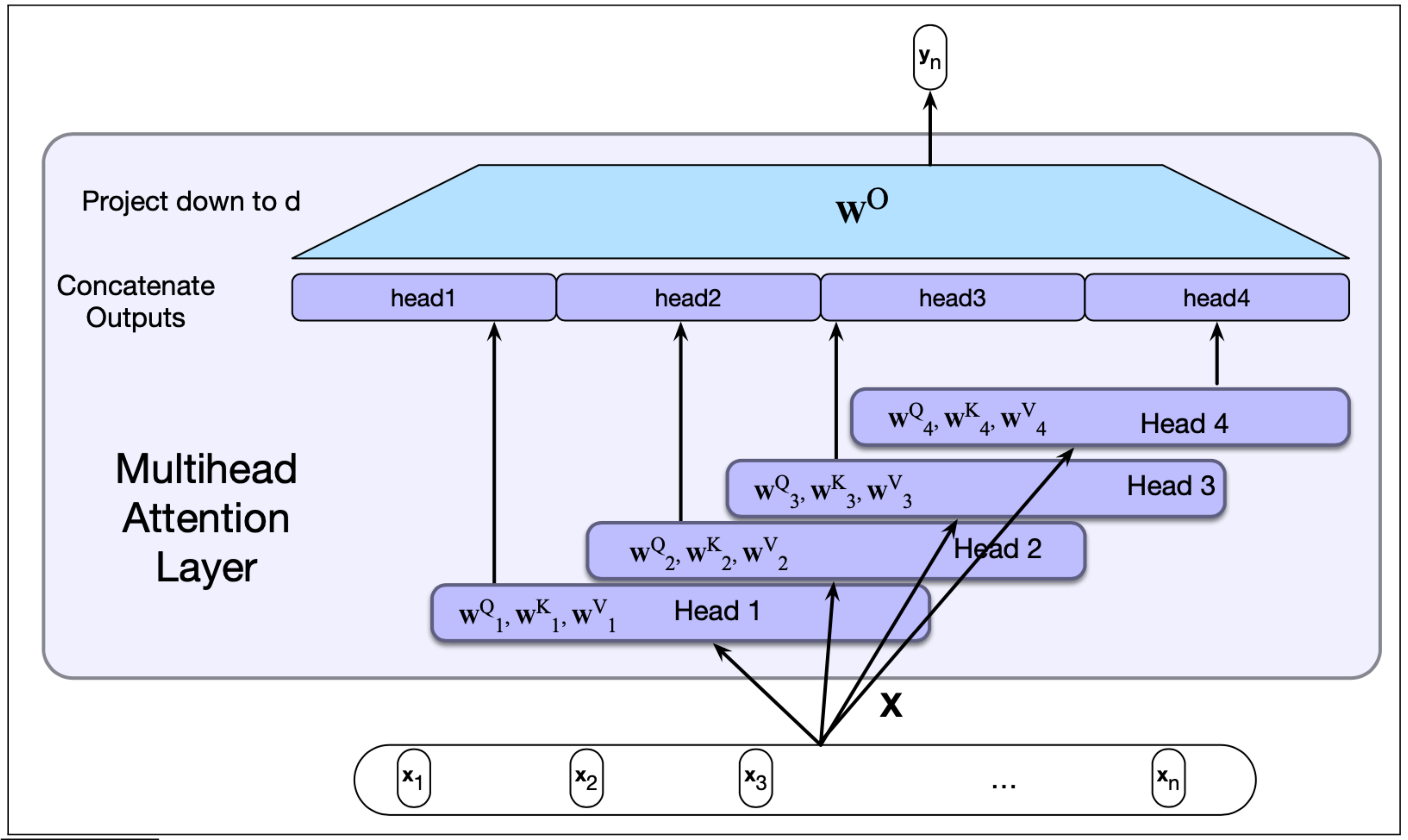

y_n
Project down to d
W^O
Concatenate Outputs
head1
head2
head3
head4
Multihead Attention Layer
W^Q_4, W^K_4, W^V_4 Head 4
W^Q_3, W^K_3, W^V_3 Head 3
W^Q_2, W^K_2, W^V_2 Head 2
W^Q_1, W^K_1, W^V_1 Head 1
X
x_1 x_2 x_3 ... x_n

Figure 9.19 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can be stacked.

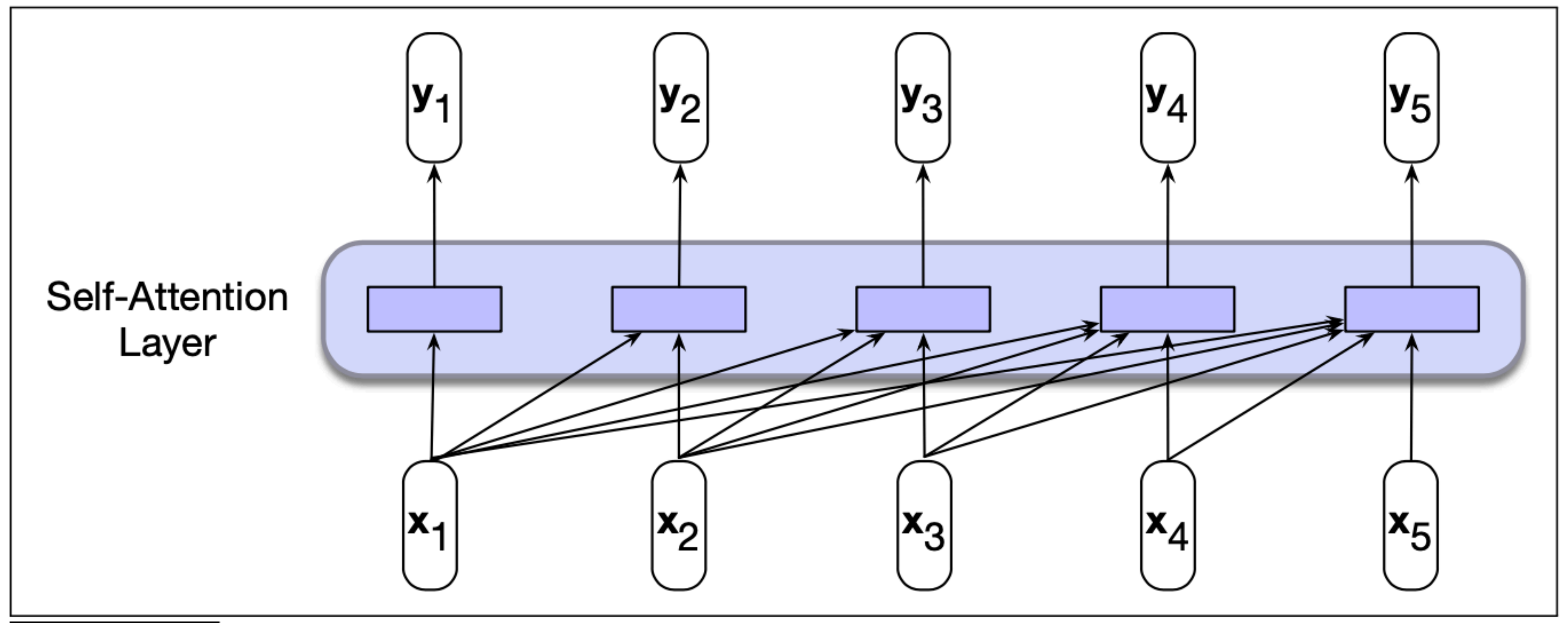


Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

TRANSFORMERS & WORD ORDER: POSITIONAL EMBEDDINGS

- Transformers (as discussed so far) are not “aware” of the relative or absolute positions of the input tokens - computation is the same regardless of the order of the inputs
- A basic way to incorporate position information is to create embeddings for each position
  - E.g., position 1 has its own vector embedding, position 2 has its own vector embedding, etc.
- Then we combine (e.g., add) the input word embedding with the respective position embedding to create a composite embedding for each input
- Problem with this approach – there is typically less data for higher positions
- Creating good positional embeddings is still an open problem

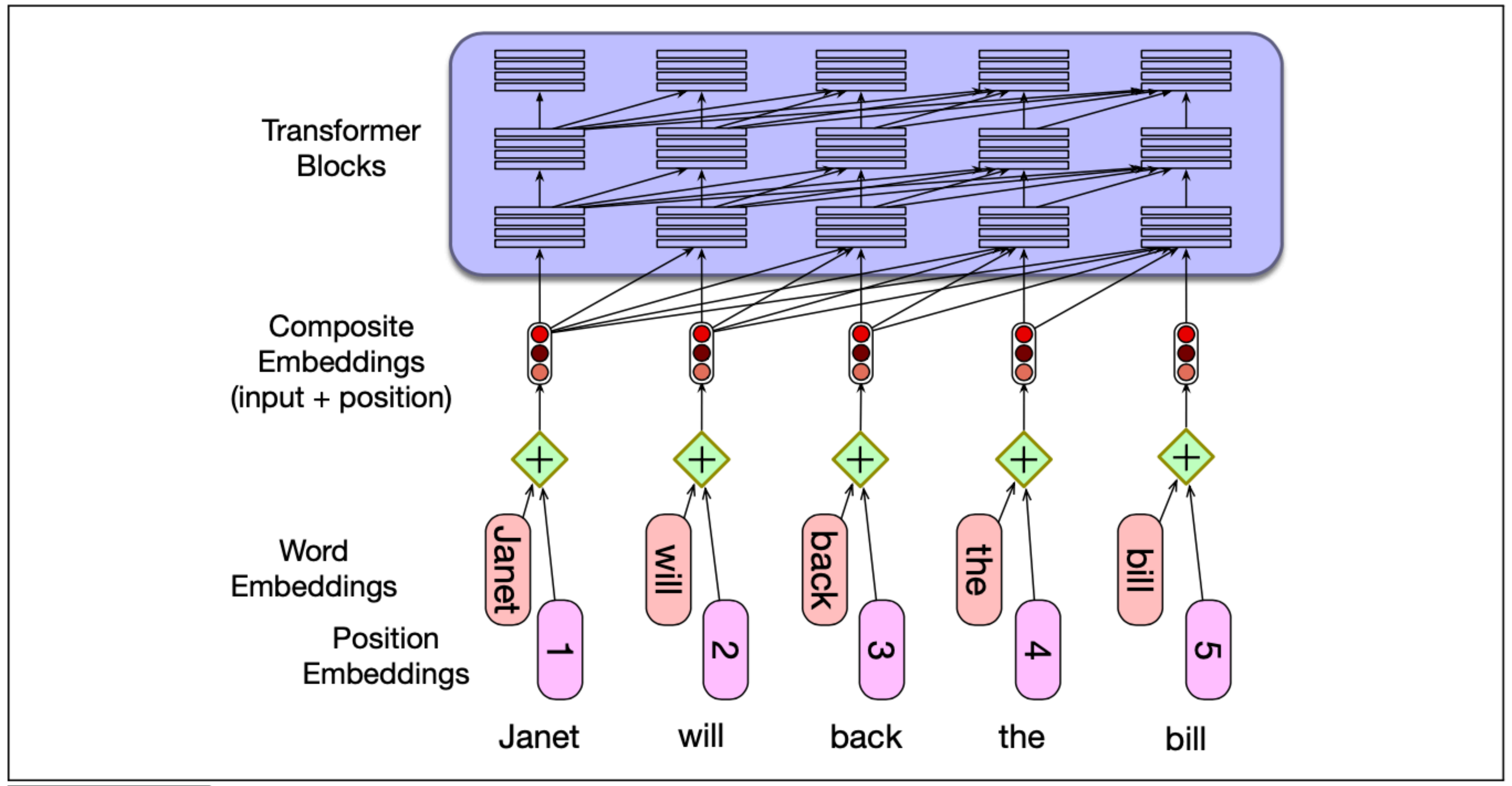


Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

TRANSFORMER EXAMPLES

- Transformers can be applied to various NLP tasks similarly to an RNN, e.g., Language Modeling, Summarization
  - For text generation, the transformer can be primed with some “Prefix Text”
  - Unlike an RNN, the priming context (the inputs) and the outputs generated by the transformer at each step are incorporated into the computation
- Note that training for each input can be done in parallel (unlike in an RNN)

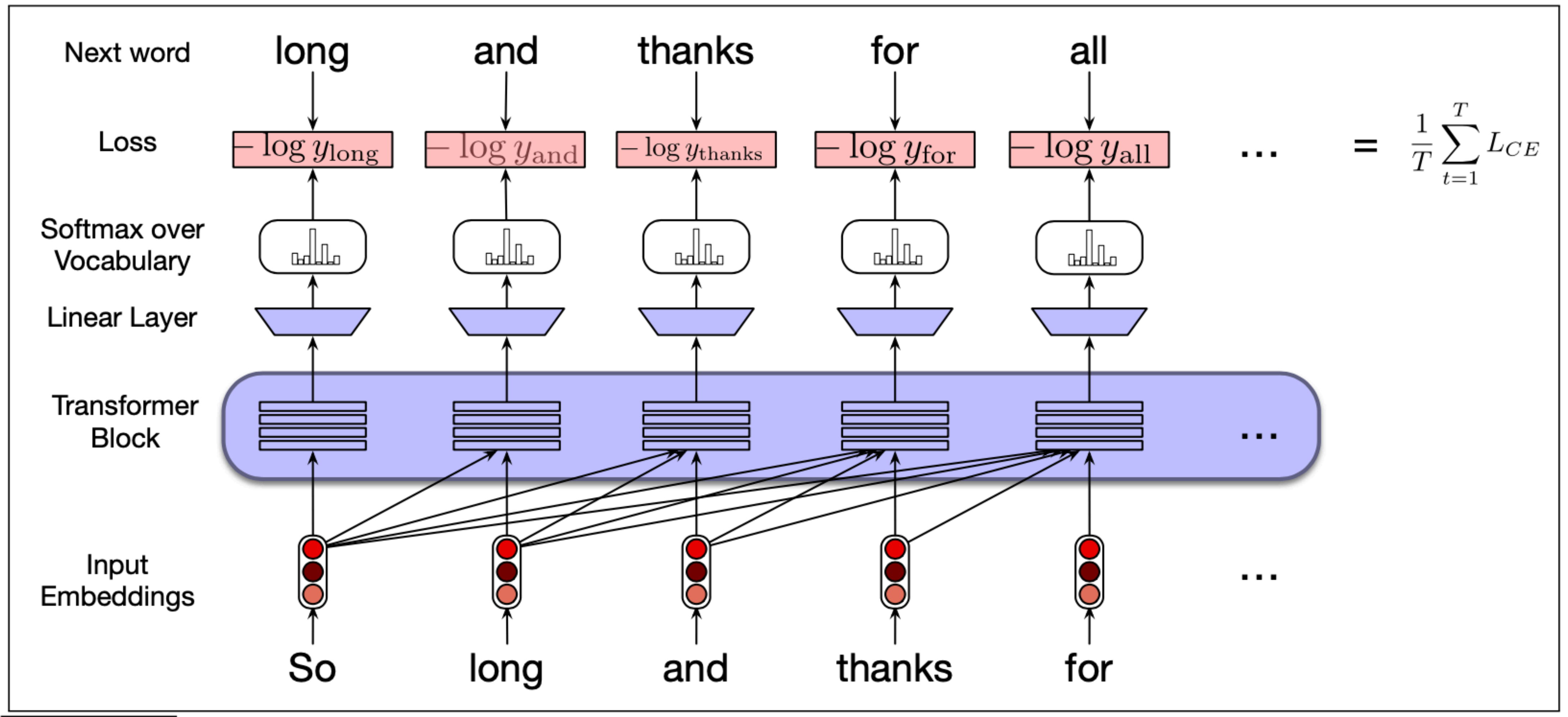


Figure 9.21 Training a transformer as a language model.

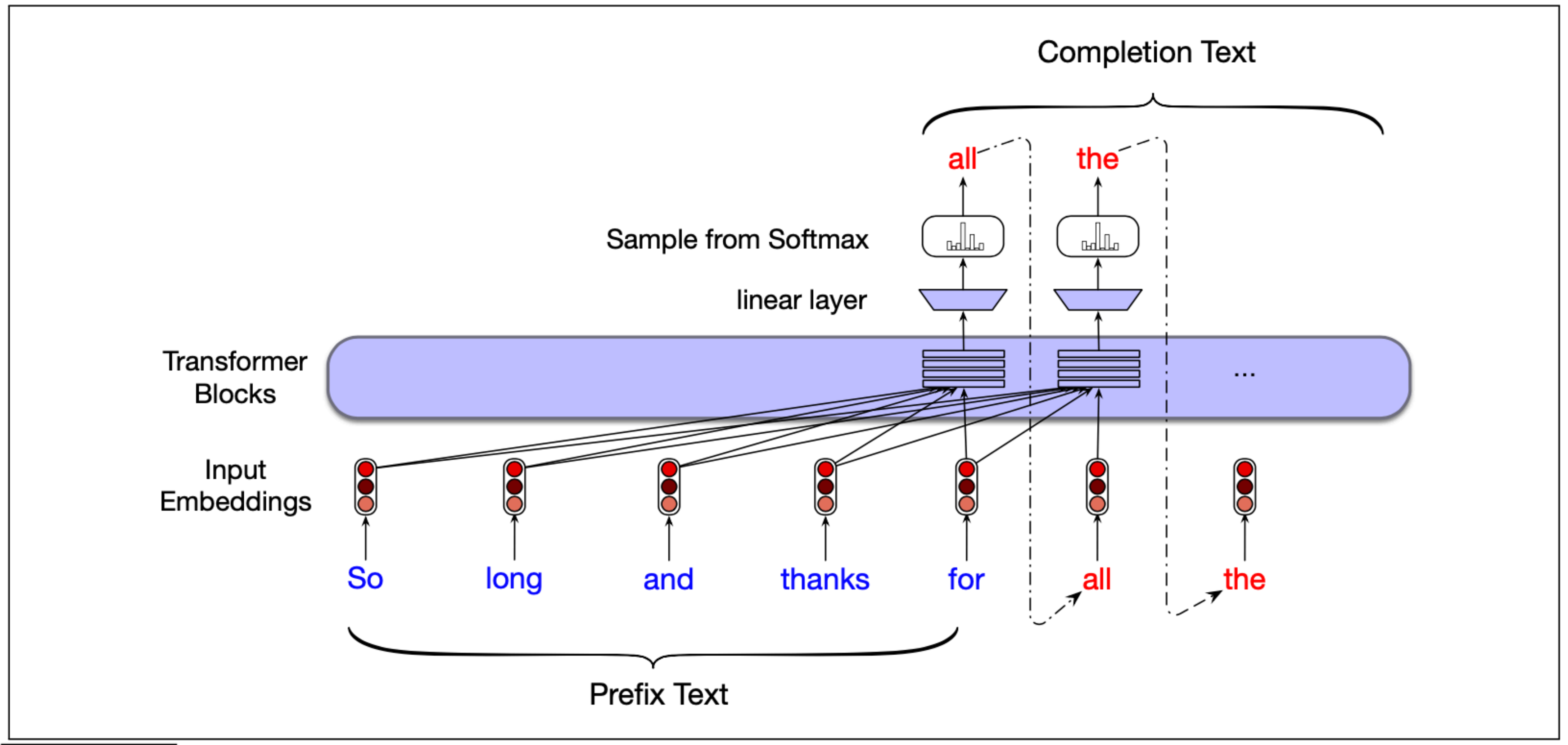


Figure 9.22 Autoregressive text completion with transformers.

SUMMARIZATION WITH TRANSFORMERS

Training the model:

- Given an article with tokens $(x_1, x_2, \ldots, x_m)$ and a summary with tokens $(y_1, y_2, \ldots, y_n)$, convert the training corpus to the concatenation of each article and its summary, separated with the delimiter δ:

  $$(x_1, x_2, \ldots, x_m, \delta, y_1, y_2, \ldots, y_n)$$

- Do autoregressive training with teacher forcing (LM model example from earlier)

Using the model to generate a summary:

- Feed model an article followed by the delimiter δ, which would prompt the model to generate a summary
- The model has access to the full article and the newly generated text at all times

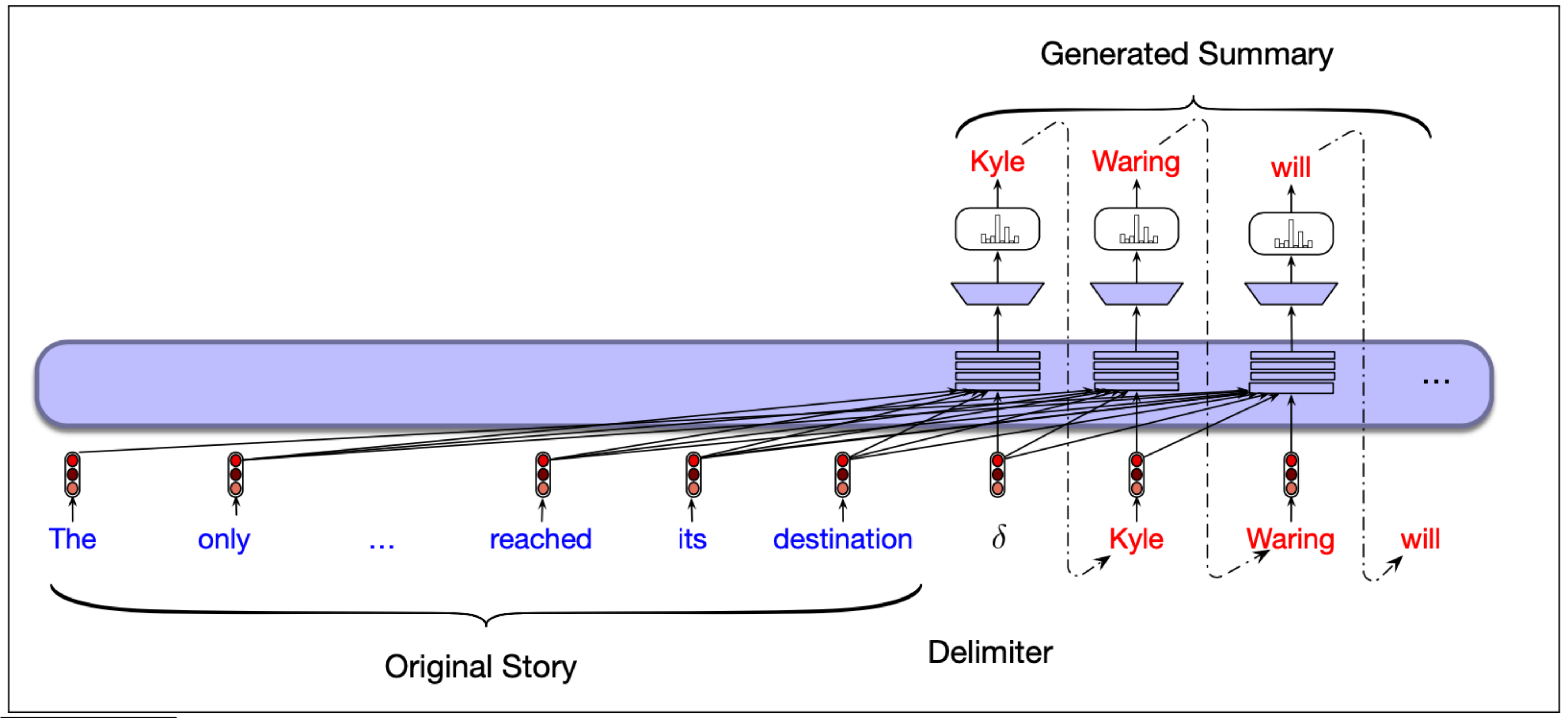


Figure 9.24 Summarization with transformers.