Sequence Processing with Recurrent Networks and Transformers

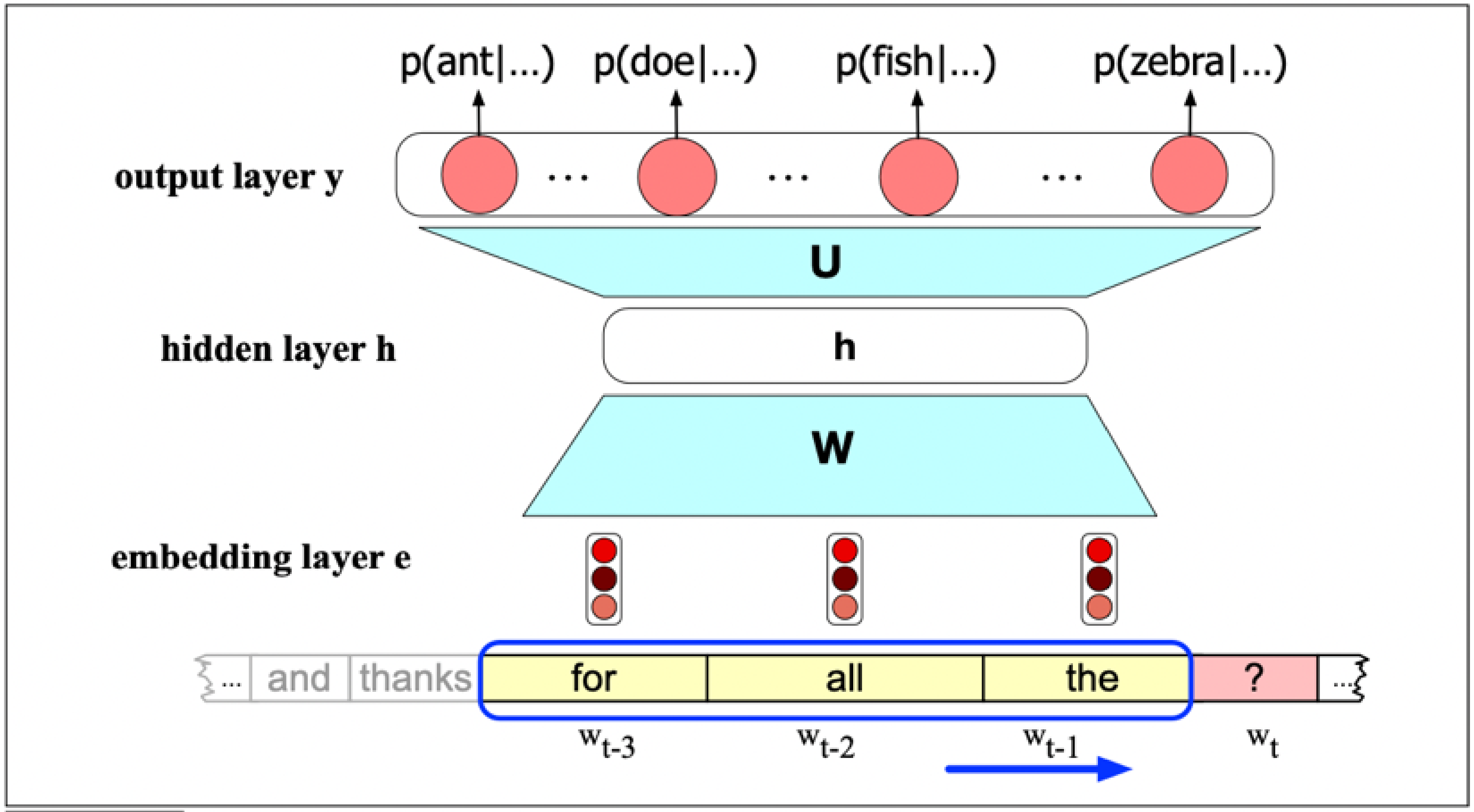
Reference:
- D. Jurafsky and J. Martin, “Speech and Language Processing”

Motivation

- We have seen feedforward neural networks along with their applications to neural language models.
- These models operate by accepting fixed-sized windows of tokens as input.
- Sequences longer than the window are processed by sliding windows over the input making predications as they go, with the end result being a sequence of predictions spanning the input.

Simple Recurrent Networks

Fig. 1



Simplified sketch of a feedforward neural language model moving through a text. At each time step *t* the network converts *N* context words, each to a *d*-dimensional embedding, and concatenates the *N* embeddings together to get the $1 \times Nd$ unit input vector *x* for the network. The output of the network is a probability distribution over the vocabulary representing the model's belief with respect to each word being the next possible word.

Motivation

- Feedforward sliding-window shares the weakness of n-gram approaches: limited context.
  - Anything outside the context window has no impact on the decision being made.
  - Many language tasks require access to information that can be arbitrarily distant from the current word.
- The use of windows makes it difficult for networks to learn systematic patterns arising from phenomena like constituency and compositionality: the way the meaning of words in phrases combine together.
  - For example, the phrase “all the” appears in one window in the second and third positions, and in the next window in the first and second positions, forcing the network to learn two separate patterns for what should be the same item.

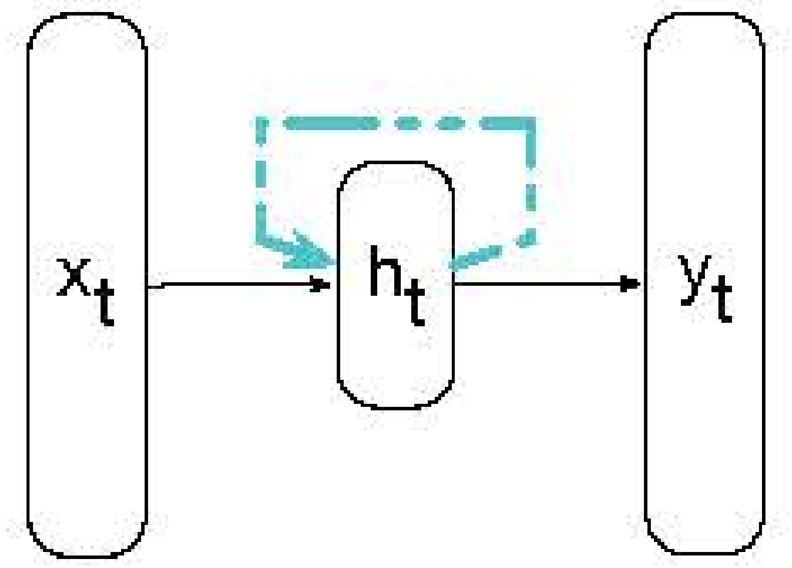
Simple Recurrent Neural Networks

- A recurrent neural network (RNN) is any network that contains a cycle within its network connections.
- Any network where the value of a unit is directly, or indirectly, dependent on earlier outputs as an input.
- In general, such networks are difficult to reason about, and to train.
- However, within the general class of recurrent networks, there are constrained architectures that have proven to be extremely useful when applied to language problems.
- We'll introduce a class of recurrent networks referred to as Simple Recurrent Networks (SRNs) or Elman Networks.
  - These networks are useful in their own right and serve as the basis for more complex approaches to be discussed later.

Simple Recurrent Neural Networks

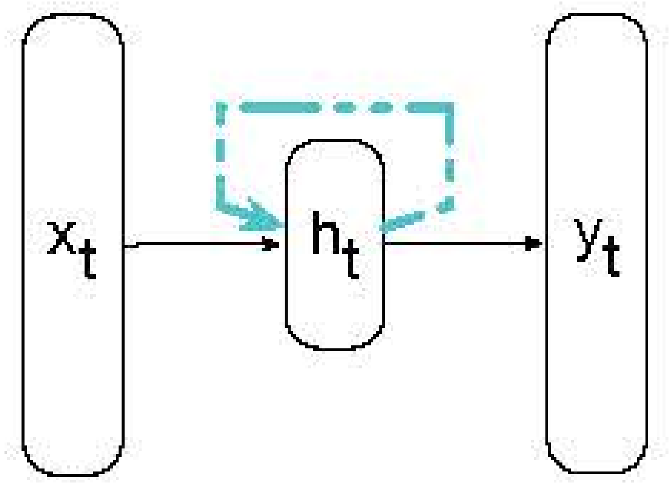
- The right figure abstractly illustrates the recurrent structure of a simple RNN.
- As with ordinary feedforward networks, an input vector representing the current input element, x_t, is multiplied by a weight matrix and then passed through a non-linear activation to compute the values for a layer of hidden units.

Fig. 2



Simple Recurrent Neural Networks

- This hidden layer is then used to calculate a corresponding output, y_t.
- Sequences are processed by presenting one item at a time to the network.
- The key difference from a feed-forward network lies in the recurrent link shown in the figure with the dashed line.
- This link augments the input to the computation at the hidden layer with the activation value of the hidden layer *from the preceding point in time*.



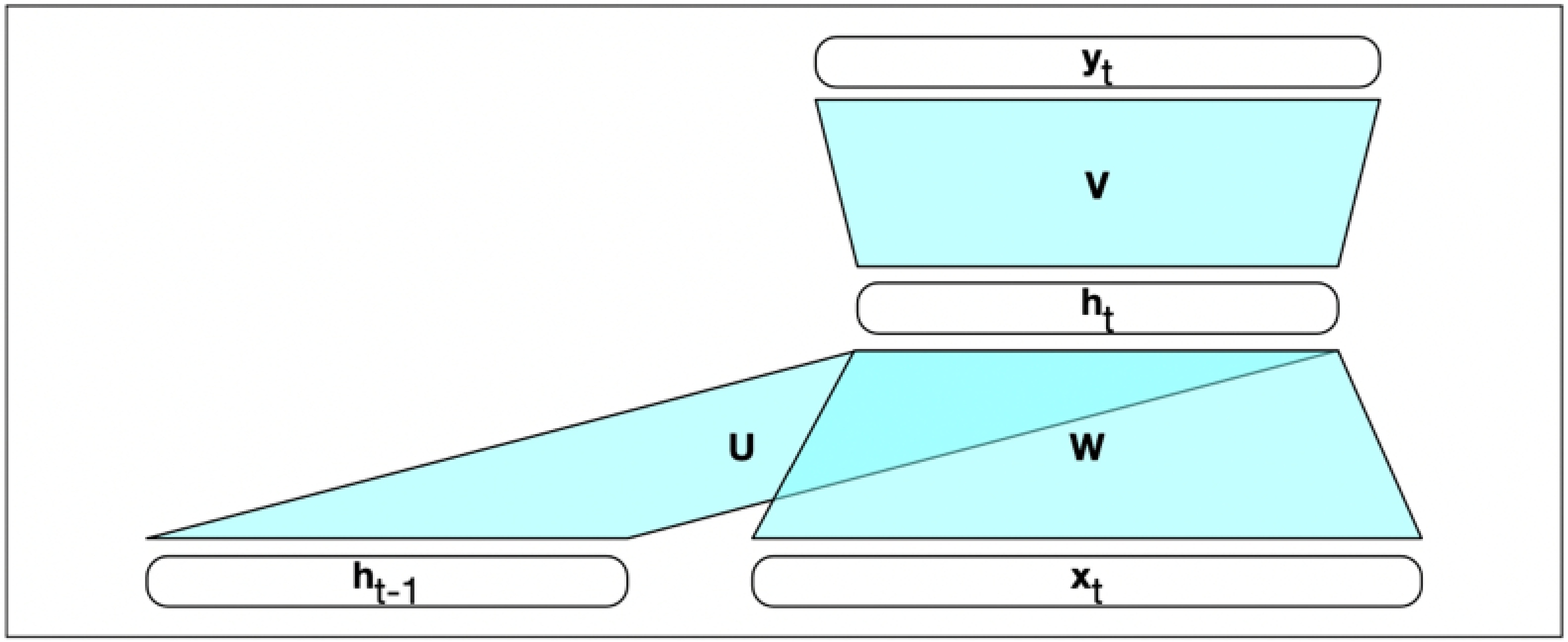
Simple Recurrent Neural Networks

- The hidden layer from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time.
- Importantly, the architecture does not impose a fixed-length limit on this prior context
- The context embodied in the previous hidden layer includes information extending back to the beginning of the sequence.

Simple Recurrent Neural Networks

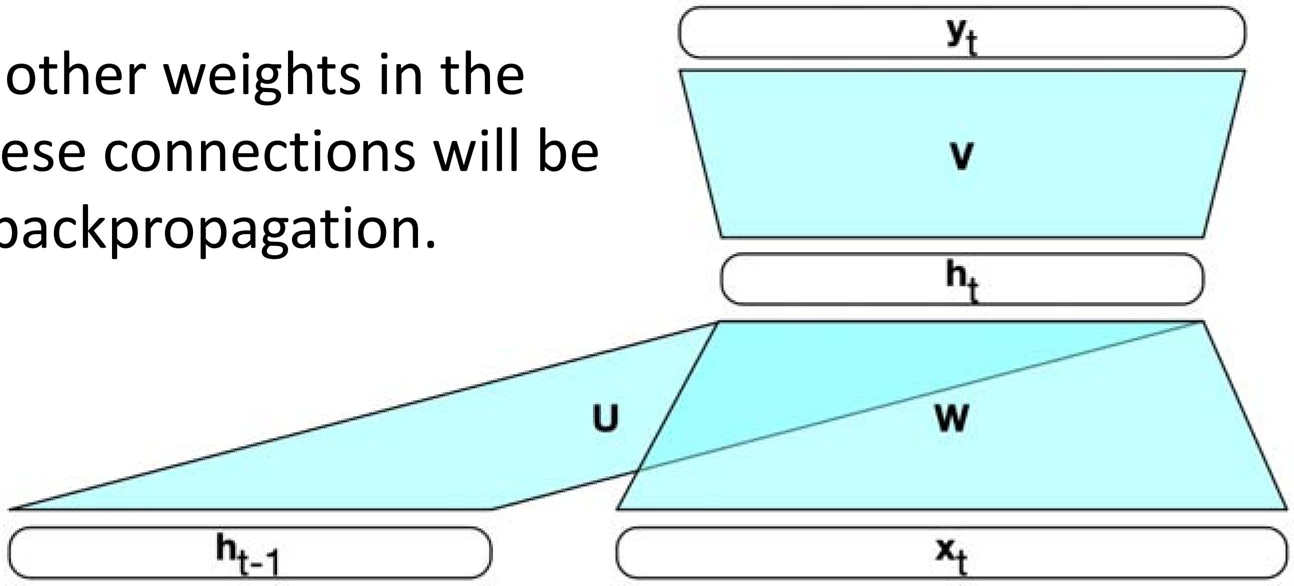
- Given an input vector and the values for the hidden layer from the previous time step, we're still performing the standard feed-forward calculation.
- To see this, consider the following figure (Fig. 3) which clarifies the nature of the recurrence and how it factors into the computation at the hidden layer.

Fig. 3



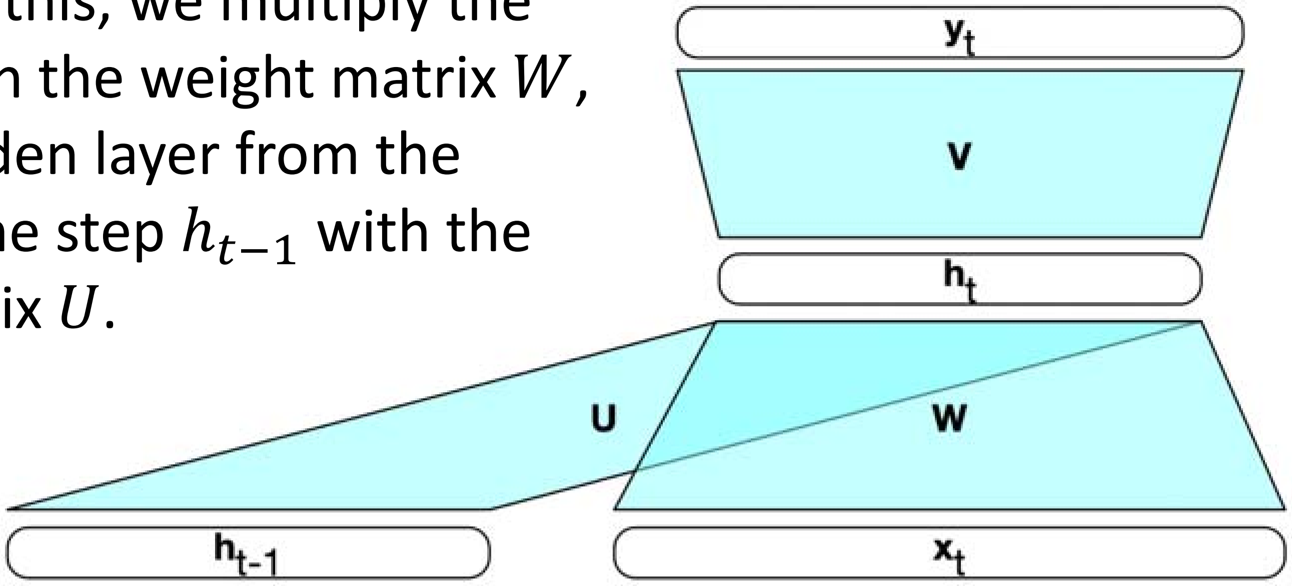
Simple Recurrent Neural Networks

- The most significant change lies in the new set of weights, *U*, that connect the hidden layer from the previous time step to the current hidden layer.
- These weights determine how the network should make use of past context in calculating the output for the current input.
- As with the other weights in the network, these connections will be trained via backpropagation.



Simple Recurrent Neural Networks

- Forward inference (mapping a sequence of inputs to a sequence of outputs) in an RNN is nearly identical to what we've already seen with feedforward networks.
- To compute an output y_t for an input x_t , we need the activation value for the hidden layer h_t .
- To calculate this, we multiply the input x_t with the weight matrix W, and the hidden layer from the previous time step h_{t-1} with the weight matrix U.

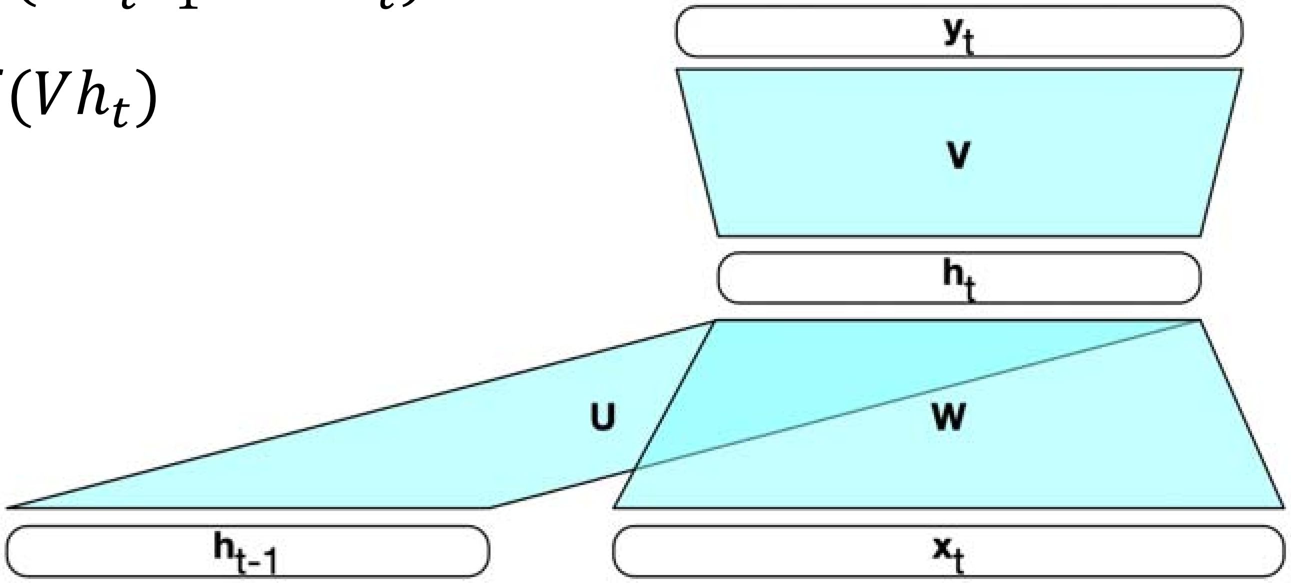


Simple Recurrent Neural Networks

- We add these values together and pass them through a suitable activation function g, to arrive at the activation value for the current hidden layer, h_t .
- Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.

$$h_t = g(Uh_{t-1} + Wx_t)$$

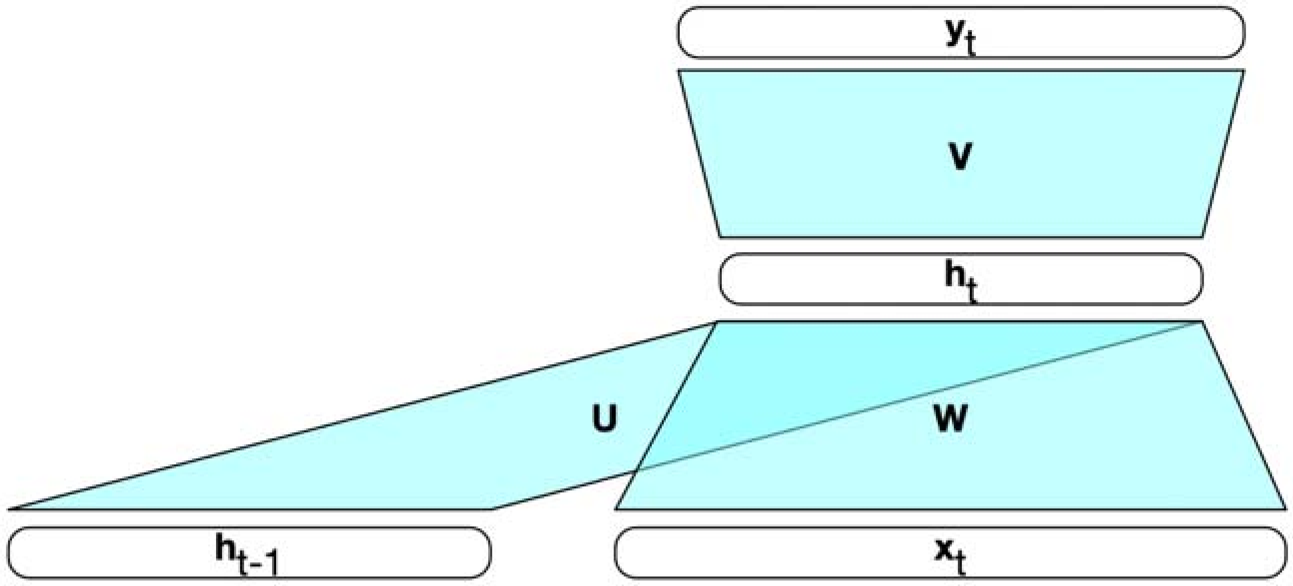
$$y_t = f(Vh_t)$$



Simple Recurrent Neural Networks

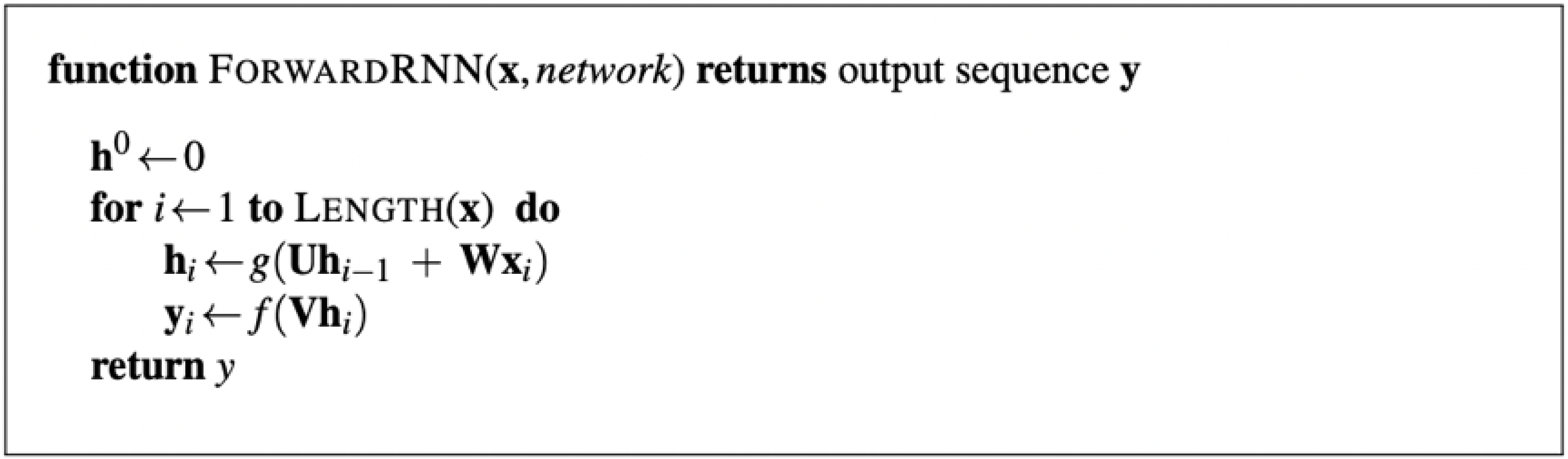
- In the commonly encountered case of soft classification, computing y_t consists of a softmax computation that provides a normalized probability distribution over the possible output classes.

$$y_t = \text{softmax}(Vh_t)$$



Simple Recurrent Neural Networks
Inference in Simple RNNs

- The fact that the computation at time t requires the value of the hidden layer from time $t-1$ mandates an incremental inference algorithm that proceeds from the start of the sequence to the end as illustrated below



```
function FORWARDRNN($\mathbf{x}$, $network$) returns output sequence $\mathbf{y}$

  $\mathbf{h}^0 \leftarrow 0$
  for $i \leftarrow 1$ to LENGTH($\mathbf{x}$) 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 $y$
```

Forward inference in a simple recurrent network. The matrices U, V and W are shared across time, while new values for h and y are calculated with each time step.

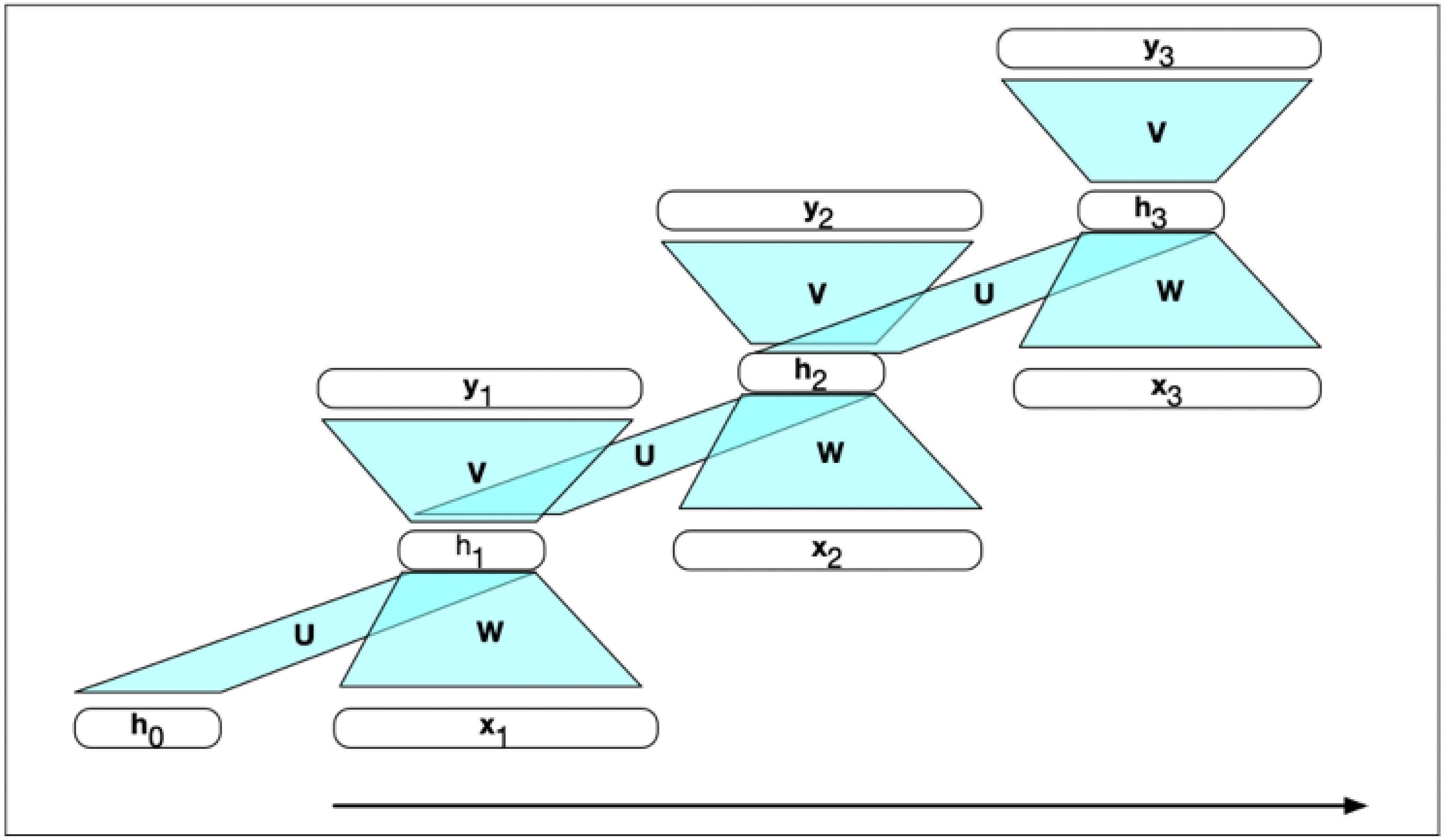
Simple Recurrent Neural Networks
Inference in Simple RNNs

- The sequential nature of simple recurrent networks can be seen by *unrolling* the network in time as is shown in Fig. 4.
- The various layers of units are copied for each time step to illustrate that they will have differing values over time.
- However, the various weight matrices are shared across time.

Simple Recurrent Neural Networks

Inference in Simple RNNs

Fig. 4



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

Simple Recurrent Neural Networks
Training

- As with feed-forward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks.
- As shown in Fig. 2, we now have 3 sets of weights to update:
    - W: the weights from the input layer to the hidden layer
    - U: the weights from the previous hidden layer to the current hidden layer
    - V: the weights from the hidden layer to the output layer.

Simple Recurrent Neural Networks
Training

- Fig. 4 highlights two considerations that we didn't have to worry about with backpropagation in feedforward networks.
- First, to compute the loss function for the output at time t, we need the hidden layer from time $t-1$.
- Second, the hidden layer at time *t* influences both the output at time *t* and the hidden layer at time $t+1$ (and hence the output and loss at $t+1$).
- It follows from this that to assess the error accruing to h_t , we'll need to know its influence on both the current output *as well as the ones that follow*.

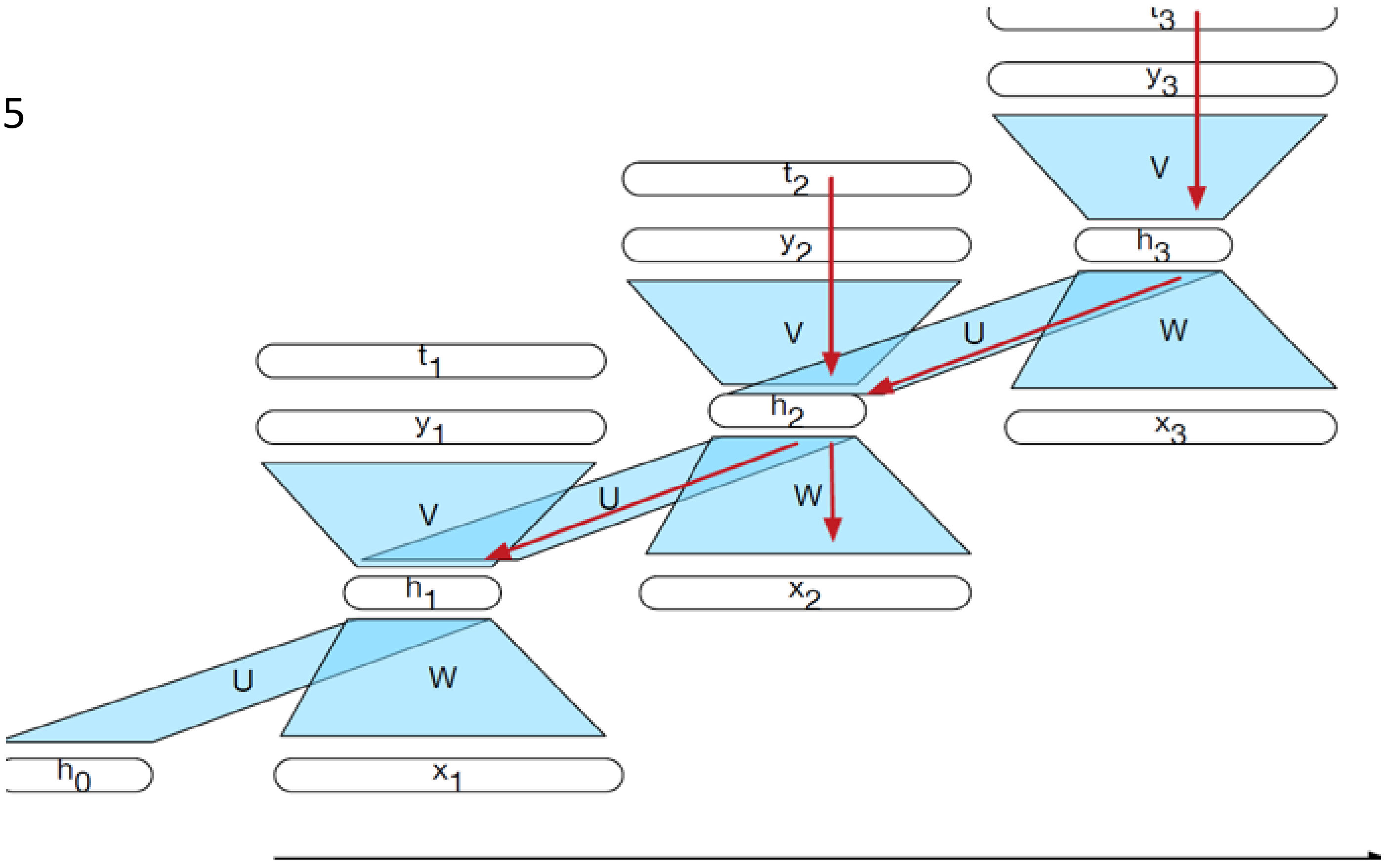
Simple Recurrent Neural Networks
Training

- Consider the situation where we are examining an input/output pair at time 2 as shown in Fig. 5.
- What do we need to compute the gradients required to update the weights U , V , and W here?
- Let's start by reviewing how we compute the gradients required to update V since this computation is unchanged from feedforward networks.
- We need to compute the derivative of the loss function L with respect to the weights V .

Simple Recurrent Neural Networks

Training

Fig. 5



The backpropagation of errors in a simple RNN t_i vectors represent the targets for each element of the sequence from the training data. The red arrows illustrate the flow of backpropagated errors required to calculate the gradients for U, V and W at time 2. The two incoming arrows converging on h_2 signal that these errors need to be summed.

Simple Recurrent Neural Networks
Training

- A two-pass algorithm for training the weights in RNNs.
- In the first pass, we perform forward inference, computing h_t, y_t , and accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step.
- In the second phase, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time.
- This general approach is commonly referred to as **Backpropagation Through Time**.

Simple Recurrent Neural Networks
Training

- We used the unrolled network shown in Fig. 4 as a way to illustrate the temporal nature of RNNs.
- Explicitly unrolling a recurrent network into a feedforward computational graph eliminates any explicit recurrences, allowing the network weights to be trained directly.
- A template is used that specifies the basic structure of the network, including all the necessary parameters for the input, output, and hidden layers, the weight matrices, as well as the activation and output functions to be used.
- Then, when presented with a specific input sequence, we can generate an unrolled feedforward network specific to that input, and use that graph to perform forward inference or training via ordinary backpropagation.

Simple Recurrent Neural Networks
Training

- For applications that involve much longer input sequences, such as speech recognition, character-level processing, or streaming of continuous inputs, unrolling an entire input sequence may not be feasible.
- In these cases, we can unroll the input into manageable fixed-length segments and treat each segment as a distinct training item.

Recurrent Neural Language Models

- Recurrent neural language models process the input sequence one word at a time, attempting to predict the next word from the current word and the previous hidden state.
- RNNs don’t have the limited context problem that n-gram models have, since the hidden state can in principle represent information about all of the preceding words all the way back to the beginning of the sequence.

Recurrent Neural Language Models

- At each step, the model uses the word embedding matrix E to retrieve the embedding for the current word, and then combines it with the hidden layer from the previous step to compute a new hidden layer.
- This hidden layer is then used to generate an output layer which is passed through a softmax layer to generate a probability distribution over the entire vocabulary.

$$e_t = Ex_t$$

$$h_t = g(Uh_{t-1} + \mathrm{W}e_t)$$

$$y_t = \mathrm{softmax}(Vh_t)$$

- The vector resulting from Vh can be thought of as a set of scores over the vocabulary given the evidence provided in h. Passing these scores through the softmax normalizes the scores into a probability distribution.

Recurrent Neural Language Models

- The probability that a particular word *i* in the vocabulary is the next word is represented by $y_t[i]$, the *i*th component of y_t:

$$\mathrm{P}(w_{t+1} = i | w_1, \dots, w_t) = y_t[i]$$

- The probability of an entire sequence is just the product of the probabilities of each item in the sequence.

$$p(w_{1:n}) = \prod_{\mathrm{i}=1}^{n} P(w_i | w_{1:i-1})$$
$$= \prod_{\mathrm{i}=1}^{n} y_i[w_i]$$

Recurrent Neural Language Models

- We use a corpus to train an RNN as a language model. We train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function.
- Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.

$$L_{CE} = -\sum_{w \in V} y_t[w] log \widehat{y_t}[w]$$

- The cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time t the CE loss is the negative log probability the model assigns to the next word in the training sequence.

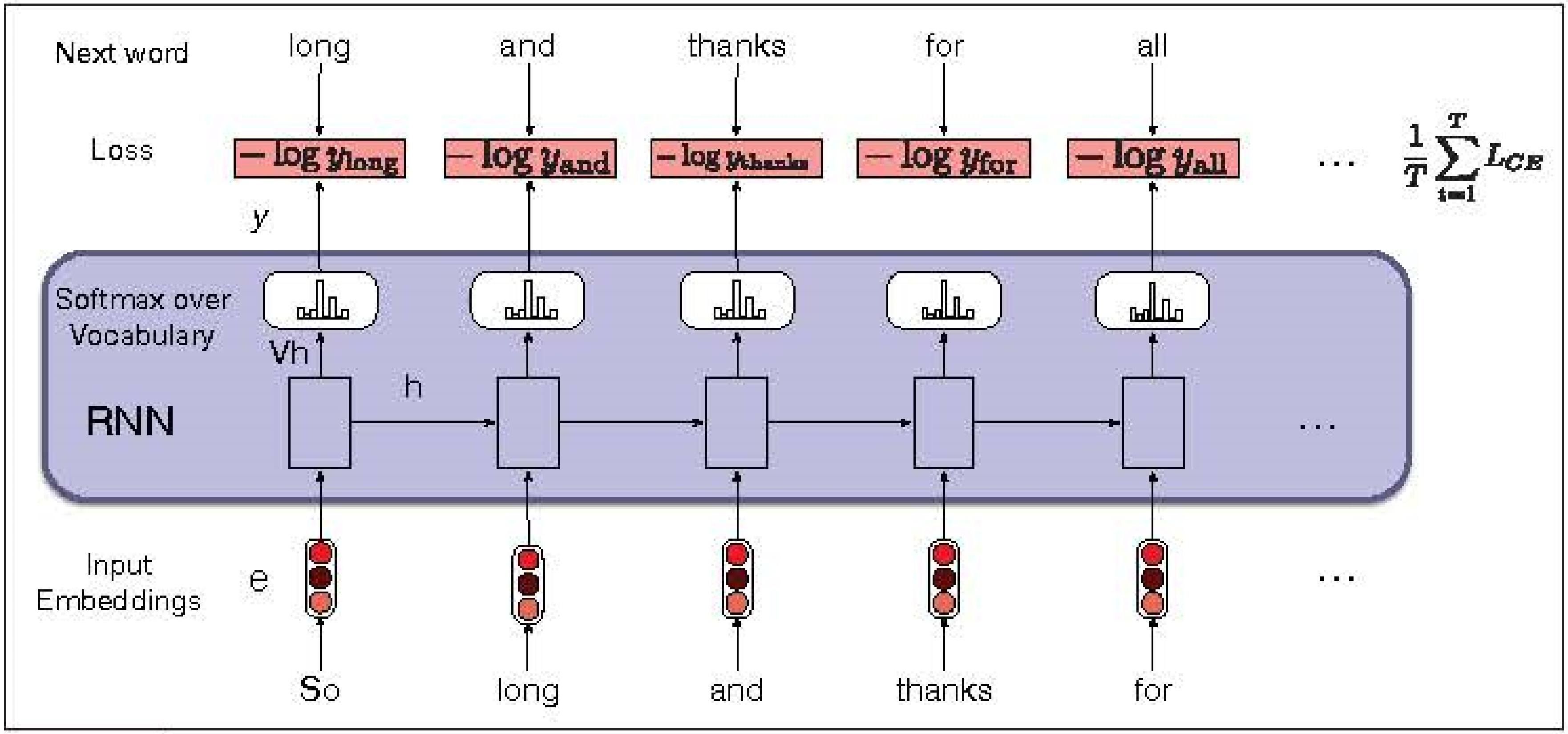
$$L_{CE}(\widehat{y_t}, y_t) = -log \widehat{y_t}[w_{t+1}]$$

Recurrent Neural Language Models

- This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.
- The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

Recurrent Neural Language Models

Fig. 6



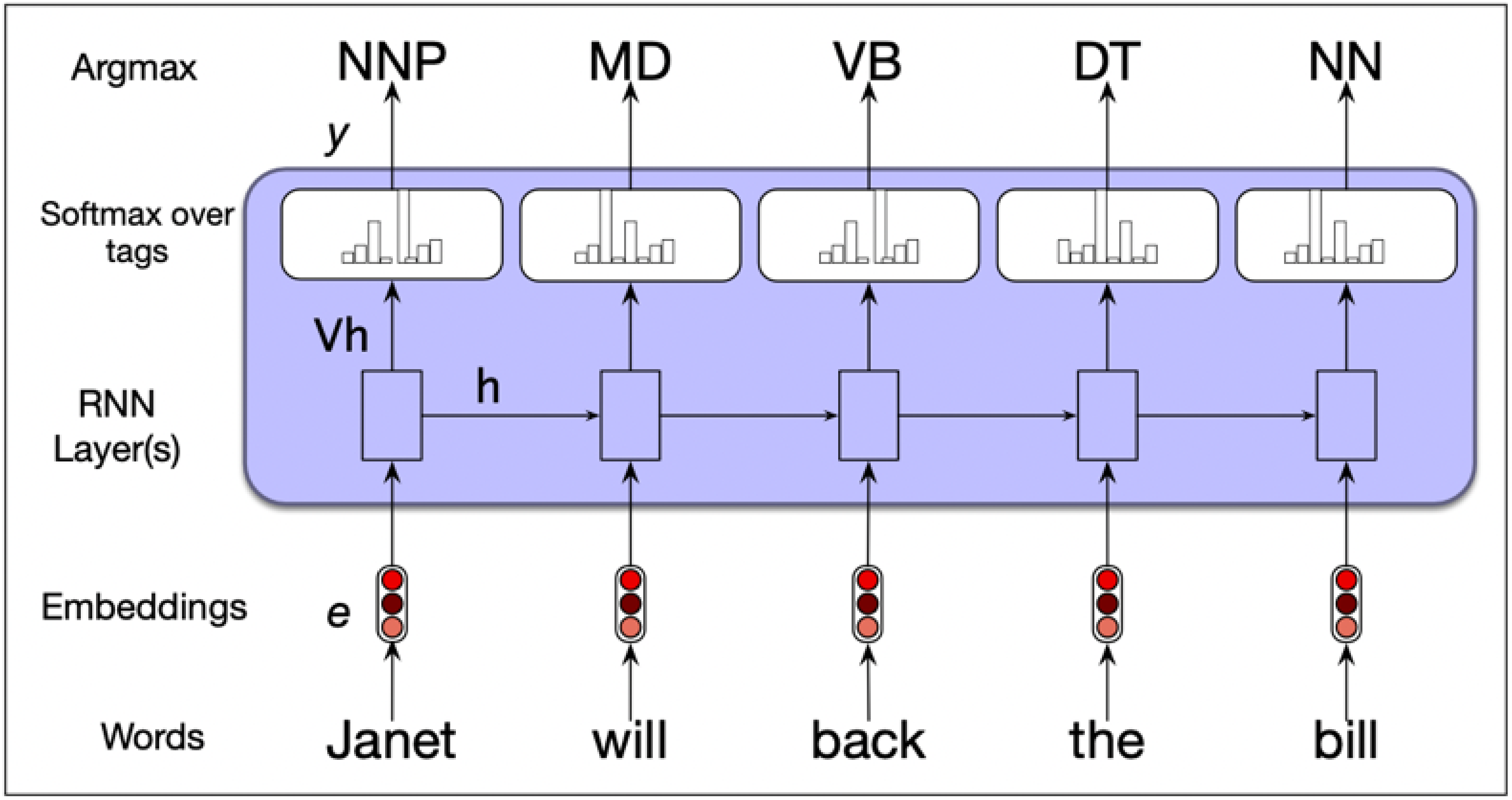
Training RNN as language models

Applications of RNNs
Sequence Labeling

- In sequence labeling, the network's task is to assign a label chosen from a small fixed set of labels to each element of a sequence, like the part-of-speech tagging and named entity recognition tasks
- In an RNN approach to sequence labeling, inputs are word embeddings and the outputs are tag probabilities generated by a softmax layer over the given tagset, as illustrated in Fig. 7.

Applications of RNNs
Sequence Labeling

Fig. 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.

Applications of RNNs
Sequence Labeling

- The inputs at each time step are pre-trained word embeddings corresponding to the input tokens.
- The RNN block is an abstraction that represents an unrolled simple recurrent network consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared U , V and W weight matrices that comprise the network.
- The outputs of the network at each time step represent the distribution over the POS tagset generated by a softmax layer.

Applications of RNNs
Sequence Labeling

- To generate a tag sequence for a given input, we can run forward inference over the input sequence and select the most likely tag from the softmax at each step.
- Since we're using a softmax layer to generate the probability distribution over the output tagset at each time step, we will again employ the cross-entropy loss during training.

Applications of RNNs
RNNs for Sequence Classification

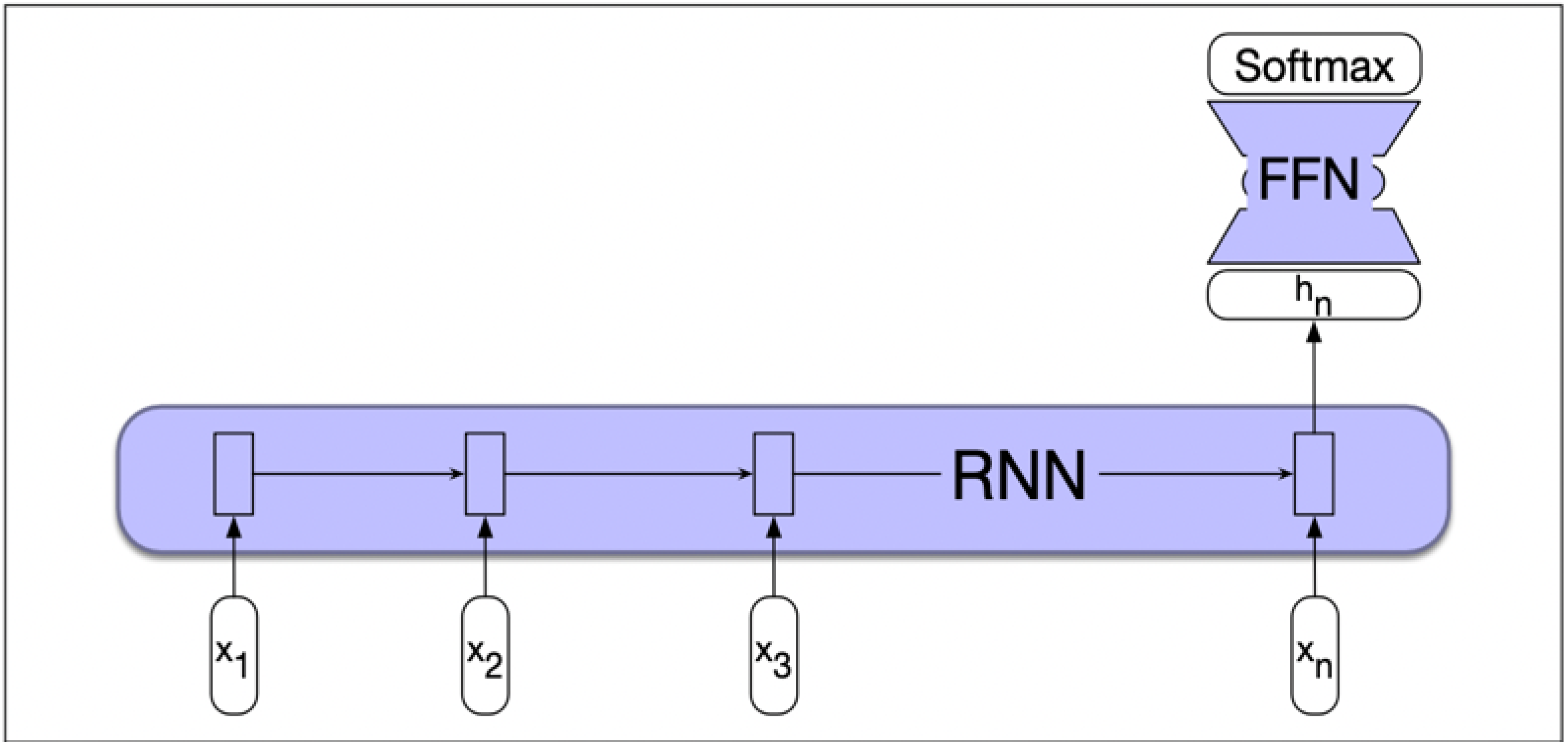
- Another use of RNNs is to classify entire sequences rather than the tokens within them.
- Other sequence classification tasks for mapping sequences of text to one from a small set of categories include document-level topic classification, spam detection, or message routing for customer service applications.
- In all of these applications, sequences of text are classified as belonging to one of a small number of categories.

Applications of RNNs
RNNs for Sequence Classification

- To apply RNNs in this setting, we pass the text to be classified through the RNN a word at a time generating a new hidden layer at each time step.
- The hidden layer for the final element of the text, h_n, is taken to constitute a compressed representation of the entire sequence.
- We can pass this representation h_n to a feedforward network that chooses a class via a softmax over the possible classes. Fig.8 illustrates this approach.

Applications of RNNs
RNNs for Sequence Classification

Fig. 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.

Applications of RNNs
RNNs for Sequence Classification

- There are no intermediate outputs for the words in the sequence preceding the last element, and therefore there are no loss terms associated with those elements.
- The loss function used to train the network weights is based entirely on the final text classification task. The output from the softmax output from the feedforward classifier together with a cross-entropy loss drives the training.

Applications of RNNs
RNNs for Sequence Classification

- The error signal from the classification is backpropagated all the way through the weights in the feedforward classifier through to its input, and then through to the three sets of weights in the RNN.
- The training regimen that uses the loss from a downstream application to adjust the weights all the way through the network is referred to as **end-to-end training**.

Applications of RNNs
Generation with Neural Language Models

- RNN-based language models can also be used to generate text.
  - 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.

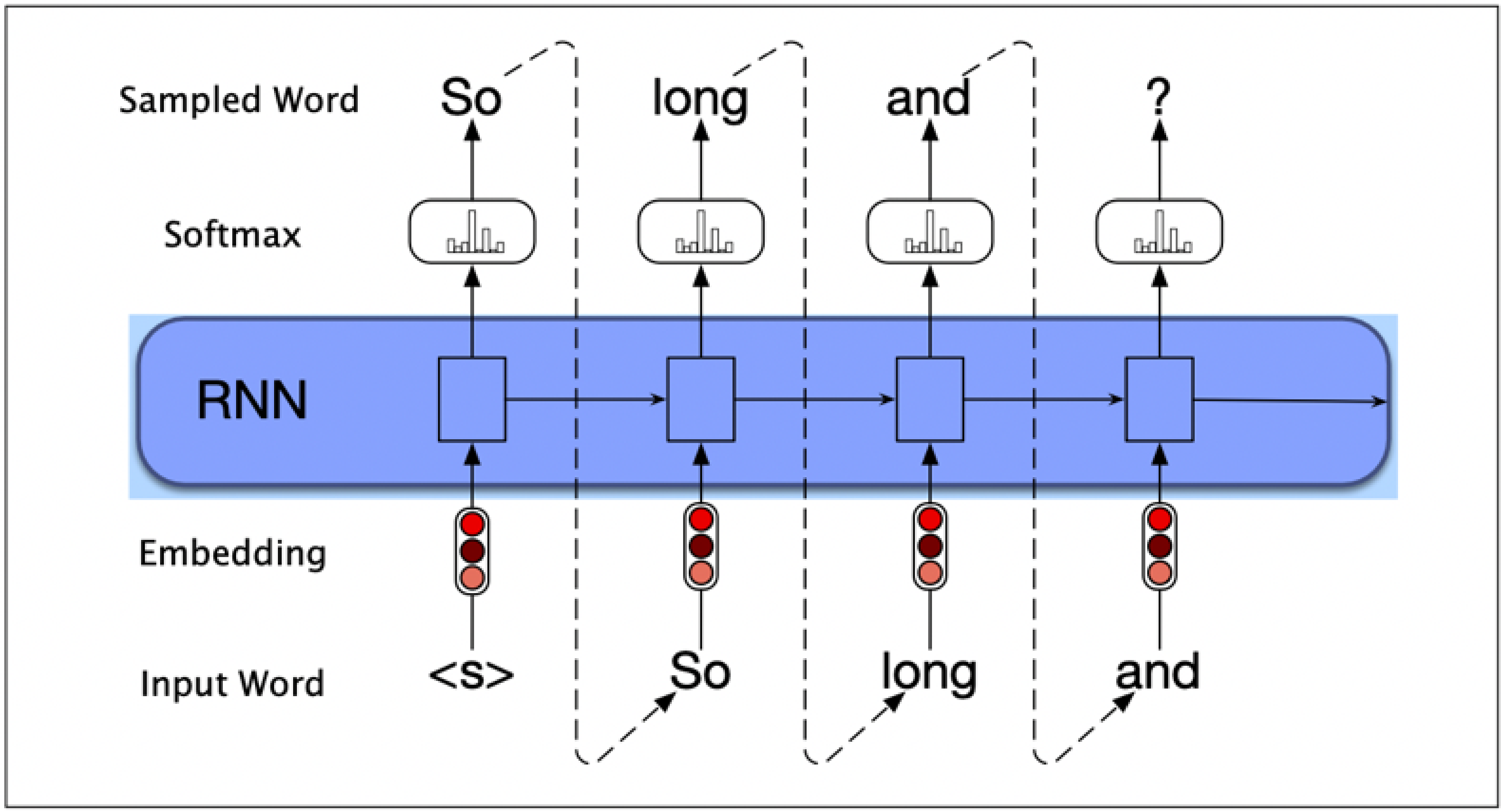
Applications of RNNs

Generation with Neural Language Models

- This approach of using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation**.
- Fig.9 illustrates this approach. In this figure, the details of the RNN’s hidden layers and recurrent connections are hidden within the blue block.

Applications of RNNs
Generation with Neural Language Models

Fig. 9



Autoregressive generation with an RNN-based neural language model.

Applications of RNNs
Generation with Neural Language Models

- This simple architecture underlies state-of-the-art approaches to applications such as machine translation, summarization, and question answering.
- The key to these approaches is to prime the generation component with an appropriate context.
- That is, instead of simply using <s> to get things started we can provide a richer task-appropriate context.

Deep Networks: Stacked and Bidirectional RNNs

- As suggested by the sequence classification architecture, recurrent networks are quite flexible.
- By combining the feedforward nature of un-rolled computational graphs with vectors as common inputs and outputs, complex networks can be treated as modules that can be combined in creative ways.
- We will introduce two of the more common network architectures used in language processing with RNNs.

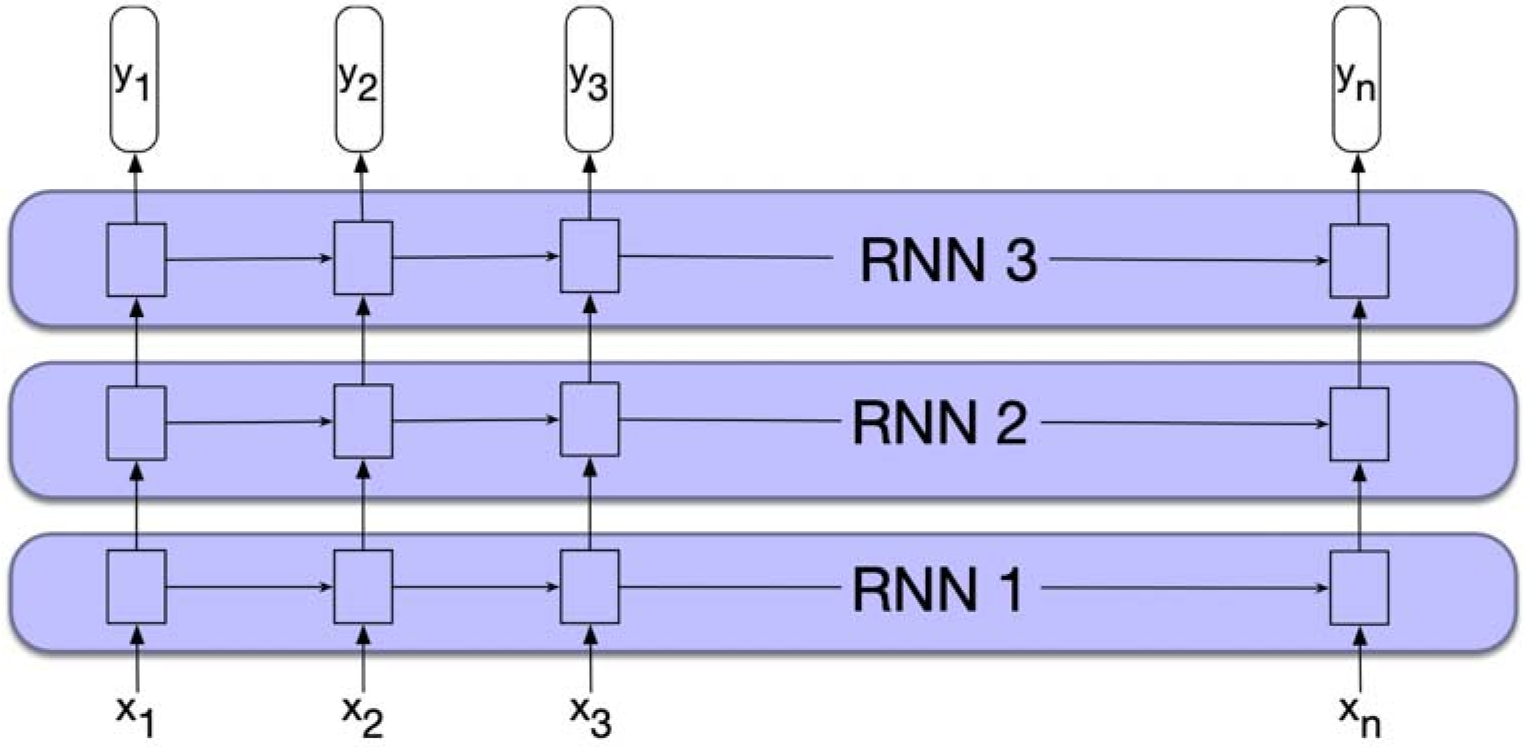
Deep Networks: Stacked and Bidirectional RNNs
Stacked RNNs

- In our examples thus far, the inputs to our RNNs have consisted of sequences of word or character embeddings (vectors) and the outputs have been vectors useful for predicting words, tags or sequence labels.
- However, nothing prevents us from using the entire sequence of outputs from one RNN as an input sequence to another one.
- Stacked RNNs consist of multiple networks where the output of one layer serves as the input to a subsequent layer, as shown in Fig. 10.

Deep Networks: Stacked and Bidirectional RNNs

Stacked RNNs

Fig. 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.

Deep Networks: Stacked and Bidirectional RNNs
Stacked RNNs

- Stacked RNNs generally outperform single-layer networks.
- One reason for this success has to do with the network’s ability to induce representations at differing levels of abstraction across layers.
- Just as the early stages of the human visual system detects edges that are then used for finding larger regions and shapes， the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers — representations that might prove difficult to induce in a single RNN.
- The optimal number of stacked RNNs is specific to each application and to each training set. However, as the number of stacks is increased the training costs rise quickly.

Deep Networks: Stacked and Bidirectional RNNs
Bidirectional RNNs

- In the left-to-right RNNs we've discussed so far, the hidden state at a given time t represents everything the network knows about the sequence up to that point.
- The state is a function of the inputs x_1,..., x_t and represents the context of the network to the left of the current time.

$$h_t^f = RNN_{forward}(x_1, \ldots, x_t)$$

- This new notation h_t^f simply corresponds to the normal hidden state at time t, representing everything the network has gleaned from the sequence so far.

Deep Networks: Stacked and Bidirectional RNNs

Bidirectional RNNs

- To take advantage of context to the right of the current input, we can train an RNN on a *reversed* input sequence.
- With this approach, the hidden state at time *t* now represents information about the sequence to the right of the current input.

$$h_t^b = RNN_{backward}(x_t, x_n)$$

- Here, the hidden state h_t^b represents all the information we have discerned about the sequence from t to the end of the sequence.

Deep Networks: Stacked and Bidirectional RNNs

Bidirectional RNNs

- Combining the forward and backward networks results in a **bidirectional RNN**.
- A Bi-RNN consists of two independent RNNs, one where the input is processed from the start to the end, and the other from the end to the start.
- We then concatenate the two representations computed by the networks into a single vector that captures both the left and right contexts of an input at each point in time.

$$h_t = [h_t^f ; h_t^b]$$

$$= h_t^f \oplus h_t^b$$

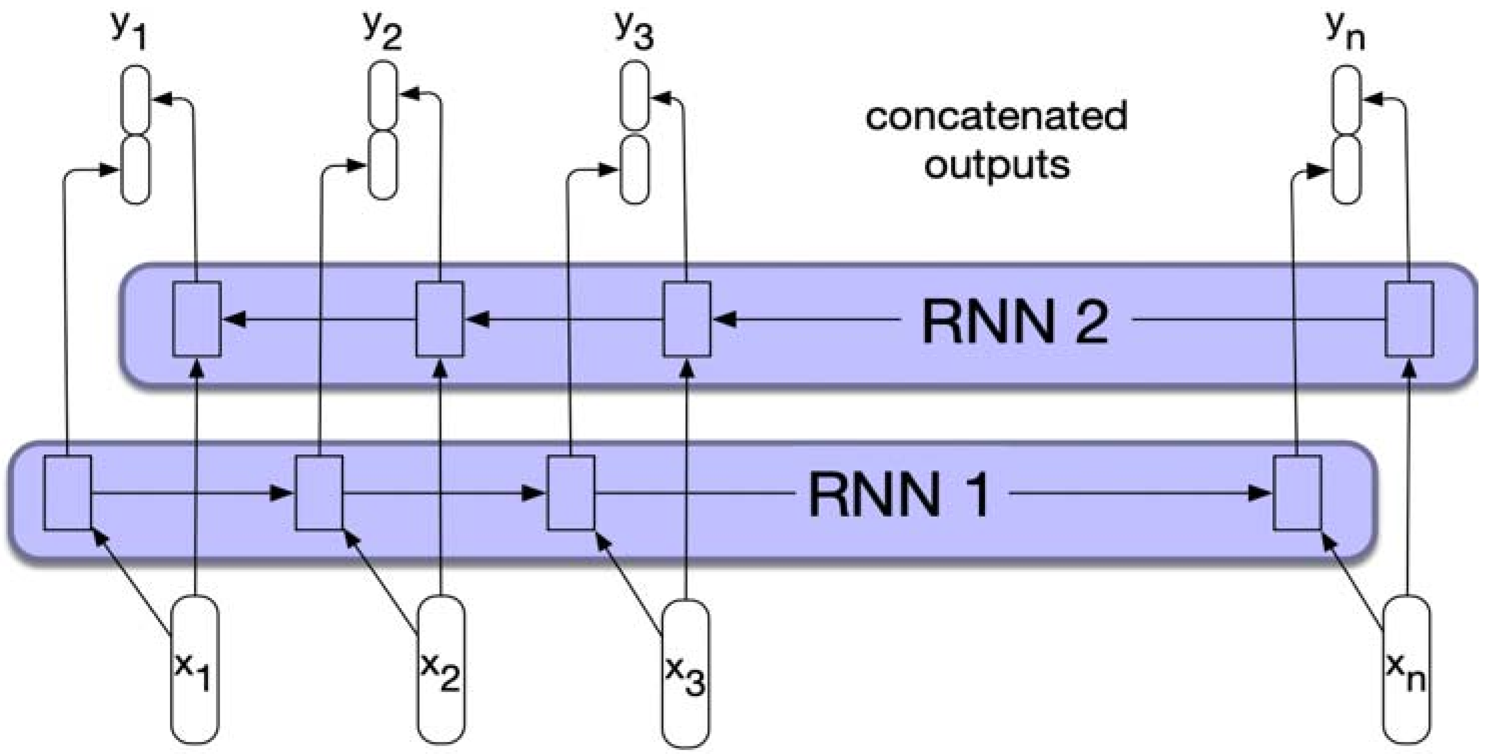
Deep Networks: Stacked and Bidirectional RNNs
Bidirectional RNNs

- Fig. 11 illustrates such a bidirectional network that concatenates the outputs of the forward and backward pass.
- Other simple ways to combine the forward and backward contexts include element-wise addition or multiplication.
- The output at each step in time thus captures information to the left and to the right of the current input.
- In sequence labeling applications, these concatenated outputs can serve as the basis for a local labeling decision.

Deep Networks: Stacked and Bidirectional RNNs

Bidirectional RNNs

Fig.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.

Deep Networks: Stacked and Bidirectional RNNs
Bidirectional RNNs

- Bidirectional RNNs have also proven to be quite effective for sequence classification.
- Recall from Fig. 8, that for sequence classification we used the final hidden state of the RNN as the input to a subsequent feedforward classifier.
- A difficulty with this approach is that the final state naturally reflects more information about the end of the sentence than its beginning.

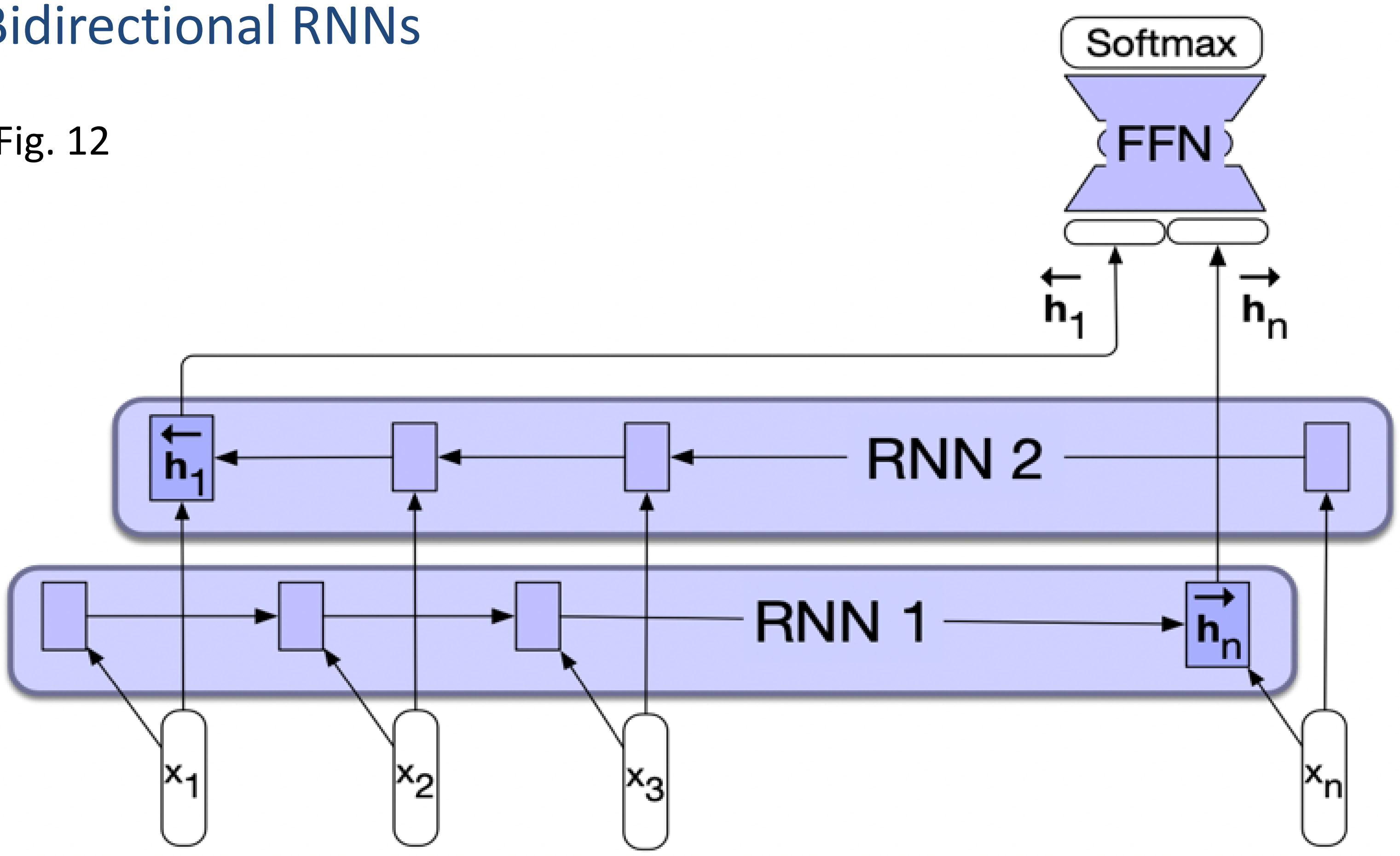
Deep Networks: Stacked and Bidirectional RNNs
Bidirectional RNNs

- Bidirectional RNNs provide a simple solution to this problem, as shown in Fig. 12, we simply combine the final hidden states from the forward and backward passes and use that as input for follow-on processing.
- Concatenation is a common approach to combining the two outputs but element-wise summation, multiplication or averaging are also used.

Deep Networks: Stacked and Bidirectional RNNs

Bidirectional RNNs

Fig. 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.

Managing Context in RNNs: LSTM

- In practice, it is quite difficult to train simple RNNs for tasks that require a network to make use of information distant from the current point of processing.
- Despite having access to the entire preceding sequence, the information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence and recent decisions.
- Yet distant information is critical to many language applications.

Managing Context in RNNs: LSTM

- Consider the following example in the context of language modeling.

  The flights the airline was cancelling were full.

- Assigning a high probability to *was* following *airline* is straightforward since *airline* provides a strong local context for the singular agreement.
- However, assigning an appropriate probability to *were* is quite difficult, not only because the plural *flights* is quite distant, but also because the intervening context involves singular constituents.
- Ideally, a network should be able to retain the distant information about plural *flights* until it is needed, while still processing the intermediate parts of the sequence correctly.

Managing Context in RNNs: LSTM

- One reason for the inability of RNNs to carry forward critical information is that the hidden layers, and by extension, the weights that determine the values in the hidden layer, are being asked to perform two tasks simultaneously:
  - provide information useful for the current decision
  - update and carry forward information required for future decisions
- A second difficulty with training RNNs arises from the need to backpropagate the error signal back through time.

Managing Context in RNNs: LSTM

- The hidden layer at time t contributes to the loss at the next time step since it takes part in that calculation.
- As a result, during the backward pass of training, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence.
- A frequent result of this process is that the gradients are eventually driven to zero — the so-called **vanishing gradients** problem.
- To address these issues, more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time.
  - The network needs to learn to forget information that is no longer needed and to remember information required for decisions still to come.

Managing Context in RNNs: LSTM

- Long short-term memory (LSTM) networks, divide the context management problem into two sub-problems:
  - removing information no longer needed from the context
  - adding information likely to be needed for later decision making.
- LSTMs accomplish this by adding an explicit context layer to the architecture and through the use of specialized neural units that make use of *gates* to control the flow of information into and out of the units
- These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

Managing Context in RNNs: LSTM

- The gates in an LSTM share a common design pattern; each consists of a feedforward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated.
- The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1.
- Combining this with a pointwise multiplication has an effect similar to that of a binary mask.
- Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

Managing Context in RNNs: LSTM

- The first gate we’ll consider is the **forget gate**, whose purpose is to delete information from the context that is no longer needed.
- The forget gate computes a weighted sum of the previous state’s hidden layer and the current input and passes that through a sigmoid.
- This mask is then multiplied by the context vector to remove the information from context that is no longer required.

$$f_t = \sigma\left(U_f h_{t-1} + W_f x_t\right)$$
$$k_t = c_{t-1} \odot f_t$$

Managing Context in RNNs: LSTM

- The next task is compute the actual information we need to extract from the previous hidden state and current inputs — the same basic computation we've been using for all our recurrent networks.

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

- Next, we generate the mask for the **add gate** to select the information to add to the current context.

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

$$j_t = g_t \odot i_t$$

Managing Context in RNNs: LSTM

- Next, we add this to the modified context vector to get our new context vector.

$$c_t = j_t + k_t$$

- The final gate we'll use is the **output gate** which is used to decide what information is required for the current hidden state.

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

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

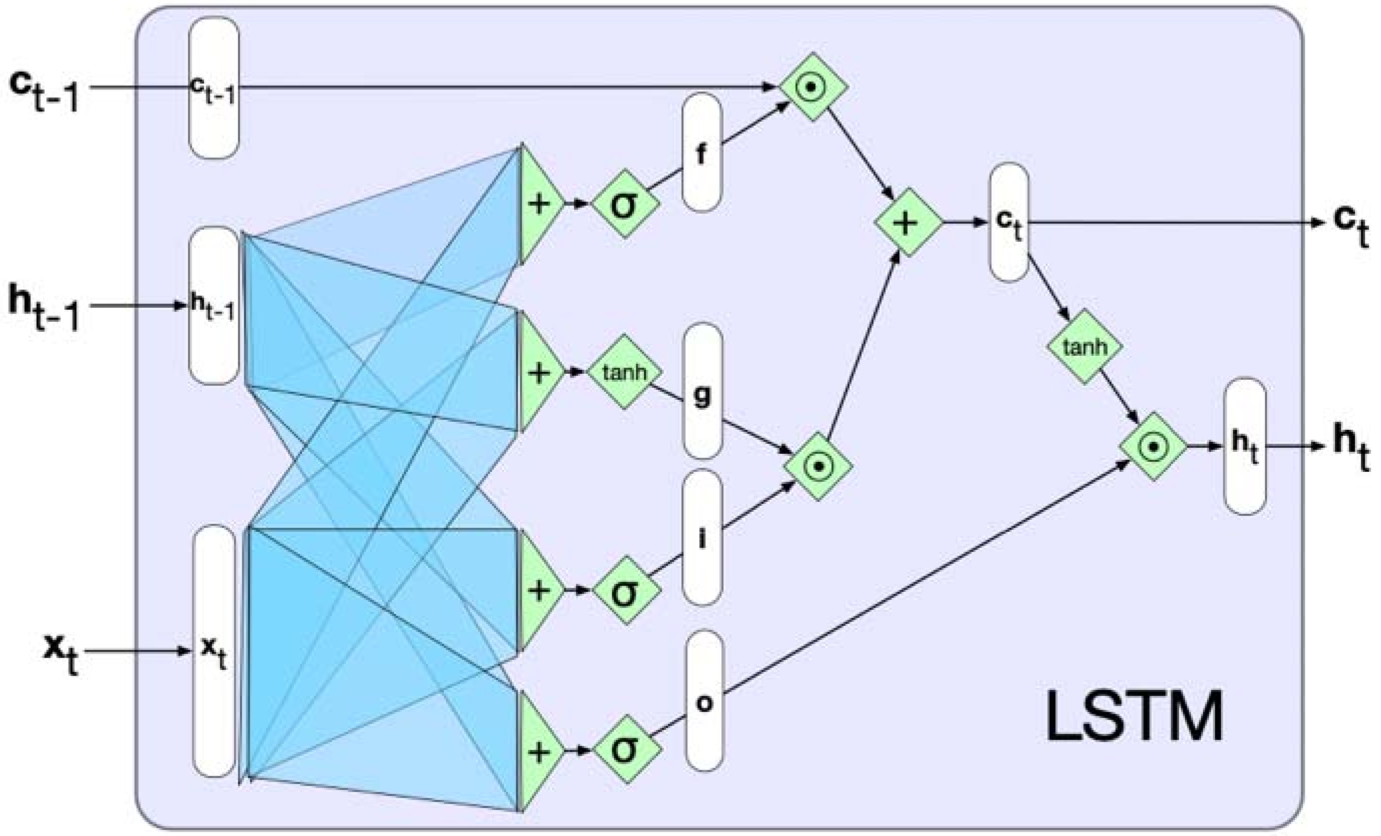
Managing Context in RNNs: LSTM

- Fig. 13 illustrates the complete computation for a single LSTM unit.
- Given the appropriate weights for the various gates, an LSTM accepts as input the context layer, and hidden layer from the previous time step, along with the current input vector. It then generates updated context and hidden vectors as output.
- The hidden layer, h_t, can be used as input to subsequent layers in a stacked RNN, or to generate an output for the final layer of a network.

Managing Context in RNNs: LSTMs and GRU

Long Short-Term Memory

Fig. 13



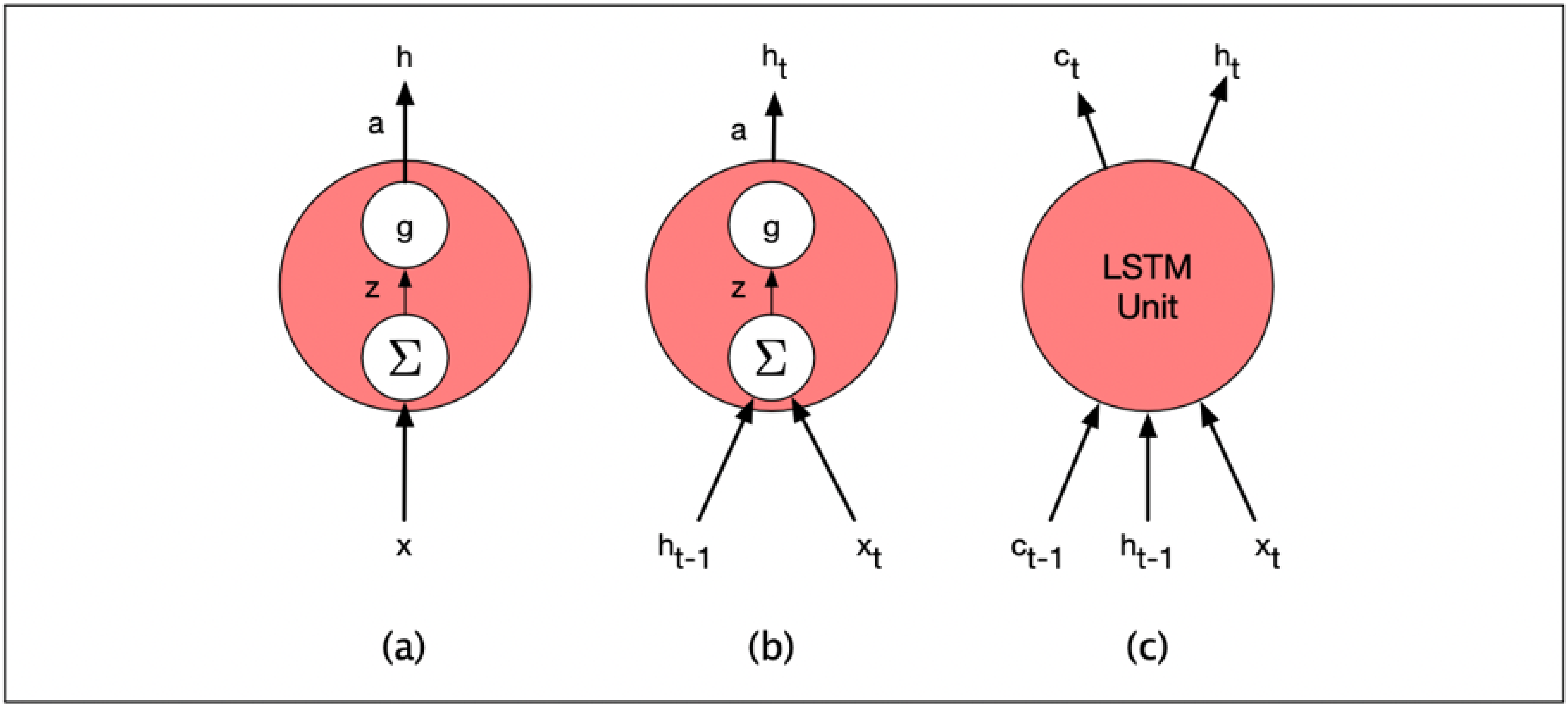
A single LSTM memory unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

Managing Context in RNNs: LSTM

- The neural units used in LSTMs are obviously much more complex than those used in basic feedforward networks.
- Fortunately, this complexity is encapsulated within the basic processing units, allowing us to maintain modularity and to easily experiment with different architectures.
- To see this, consider Fig. 14 which illustrates the inputs and outputs associated with each kind of unit.

Managing Context in RNNs: LSTM

Fig.14



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

Managing Context in RNNs: LSTM

- Basic feedforward unit: Contains a single set of weights and a single activation function determine its output, and when arranged in a layer there are no connections among the units in the layer.
- Simple recurrent network: There are two inputs and an additional set of weights to go with it. However, there is still a single activation function and output.
- The only additional external complexity for the LSTM over the Simple Recurrent unit is the presence of the additional context vector as an input and output.
- The GRU units have the same input and output architecture as the simple recurrent unit.

Managing Context in RNNs: LSTM

- This modularity is key to the power and widespread applicability of LSTM and GRU units.
- LSTM and GRU units can be substituted into any of the network architectures.
- Besides, as with simple RNNs, multi-layered networks making use of gated units can be unrolled into deep feedforward networks and trained in the usual fashion with backpropagation.

Self-Attention Networks: Transformers Motivation

- For RNNs such as LSTMs, passing information forward through an extended series of recurrent connections leads to a loss of relevant information and to difficulties in training
- Inherently sequential nature of recurrent networks *inhibits* the use of parallel computational resources
- Transformers – an approach to sequence processing that eliminates recurrent connections and returns to architectures reminiscent of the fully connected networks.

Self-Attention Networks: Transformers
Basic Architecture

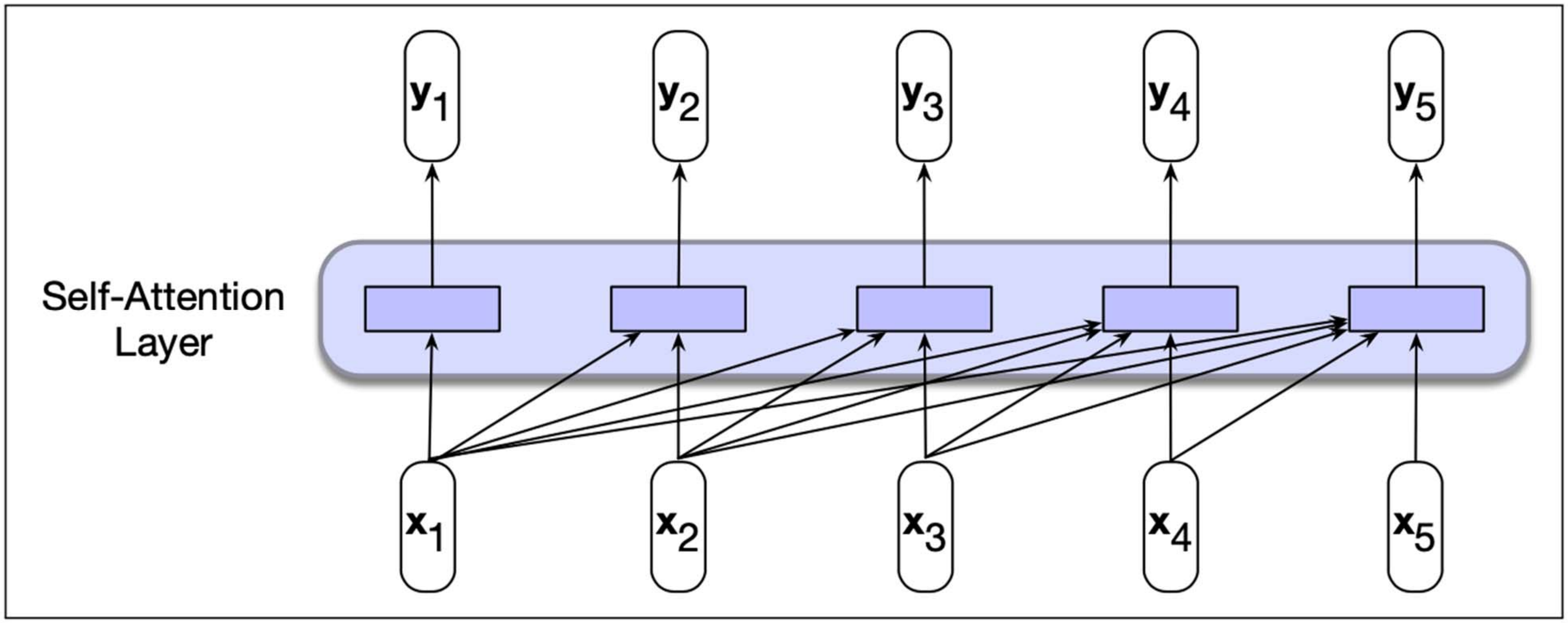
- Map sequence of input vectors $(x_1, \cdots, x_n)$ to sequences of output vectors $(y_1, \cdots, y_n)$ of the same length
- Made up of stacks of network layers consisting of simple linear layers, feedforward networks, and custom connections around them
- Use of self-attention layers – key innovation of Transformers
- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs

Self-Attention Networks: Transformers
Self-Attention Layers

- Map input sequences $(x_1, \cdots, x_n)$ to output sequences of the same length $(y_1, \cdots, y_n)$, as with the overall Transformer
- When processing each item in the input, have access to all of the inputs up to and including the one under consideration, but no access to information about inputs beyond the current one
  - Ensures we can use this approach to create language models and use them for autoregressive generation
- Computation performed for each item is independent of all the other computations
  - Means that we can easily parallelize both forward inference and training of such models

Self-Attention Networks: Transformers
Self-Attention Layers

Fig. 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.

Self-Attention Networks: Transformers
Self-Attention is an Attention-Based Approach

- An attention-based approach – to compare an item of interest to a collection of other items in way that reveals their relevance in the current context
- Self-attention – the set of comparisons are to other elements within a given sequence
- The result of these comparisons is then used to compute an output for the current input
- Returning to Fig. 15, the computation of y_3 is based on a set of comparisons between the input x_3 and its preceding elements x_1 and x_2, and to x_3 itself

Self-Attention Networks: Transformers
Self-Attention – the Simplest Form

- The simplest form of comparison between elements in a self-attention layer is a dot product
- Referring to the result of these comparisons as scores

$$score(x_i, x_j) = x_i \cdot x_j$$

  - Ranging from $-\infty$ to ∞, the larger the value the more similar the vectors that are being compared
- The first step in computing y_3 would be to compute three scores: $x_3 \cdot x_1$, $x_3 \cdot x_2$ and $x_3 \cdot x_3$

Self-Attention Networks: Transformers
Self-Attention – the Simplest Form

- To make effective use of these scores, normalize them with a softmax to create a vector of weights, α_{ij}, that indicates the proportional relevance of each input to the input element i that is the current focus of attention

$$\alpha_{ij} = \mathrm{softmax}\left(score\left(x_i, x_j\right)\right)\ \forall j \leq i$$

$$= \frac{\exp\left(score\left(x_i, x_j\right)\right)}{\sum_{k=1}^{i} \exp\left(score\left(x_i, x_k\right)\right)}\ \forall j \leq i$$

- Given the proportional scores in α, we may generate an output value y_i by taking the sum of the inputs seen so far, weighted by their respective α value

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$

Self-Attention Networks: Transformers
Self-Attention – the Simplest Form

- Steps embodied above represent the core of an attention-based approach:
  - A set of comparisons to relevant items in some context,
  - A normalization of those scores to provide a probability distribution, followed by a weighted sum using this distribution.
  - The output y is the result of this straightforward computation over the inputs.
- Unfortunately, this simple mechanism provides no opportunity for learning, everything is directly based on the original input values x.

Self-Attention Networks: Transformers
Self-Attention with Additional Parameters

- To allow for this kind of learning, Transformers include additional parameters in the form of a set of weight matrices that operate over the input embeddings.
- To motivate these new parameters, consider the three different roles that each input embedding plays during the course of the attention process.
  - As the *current attention focus* when being compared to all of the other preceding inputs – a **query**
  - In its role as a *preceding input* being compared to the current focus of attention – a **key**
  - And finally, used to compute the *output* for the current focus of attention – a **value**

Self-Attention Networks: Transformers
Self-Attention with Additional Parameters

- To capture these three different roles, transformers introduce weight matrices W^Q, W^K, and W^V
- These weights will be used to project each input vector x_i into a representation of its role as a key, query, or value.

$$q_i = W^Q x_i;\ k_i = W^K x_i;\ v_i = W^V x_i$$

  - Given input embeddings of size d_m, the dimensionality of these matrices are $d_q \times d_m$, $d_k \times d_m$ and $d_v \times d_m$ respectively
  - In the original Transformer work (Vaswani et al., 2017), d_m was 1024 and 64 for d_k, d_q and d_v

Self-Attention Networks: Transformers
Self-Attention with Additional Parameters

- Given these projections, the score between a current focus of attention, x_i and an element in the preceding context, x_j consists of a dot product between its query vector q_i and the preceding elements key vectors k_j

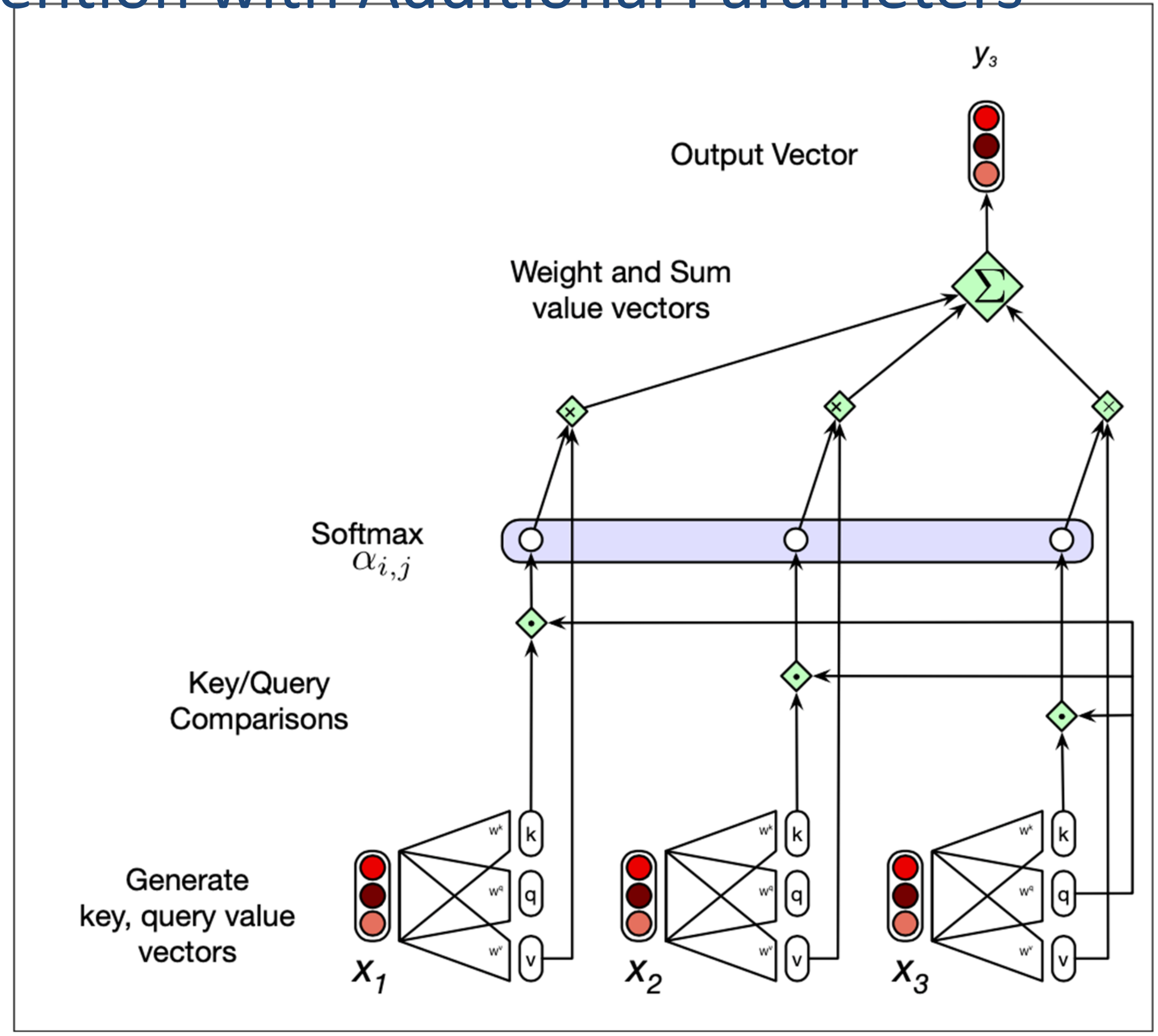
$$score(x_i, x_j) = q_i \cdot k_j$$

- The ensuing softmax calculation resulting in α_{ij} remains the same
- The output calculation for y_i is now based on a weighted sum over the value vectors v

$$y_i = \sum_{j \le i} \alpha_{ij} v_j$$

Self-Attention Networks: Transformers
Self-Attention with Additional Parameters

Fig. 16



Calculation of the value of y_3 , the third element of a sequence using causal (left-to-right) self-attention.

Self-Attention Networks: Transformers
Self-Attention – Scaled Dot-Product

- The result of dot product can be an arbitrarily large (positive or negative) value.
- In computing α_{ij}, exponentiating such large values can lead to numerical issues and to an effective loss of gradients during training.
- A scaled dot-product approach divides the result of the dot product by a factor related to the size of the embeddings before passing them through the softmax
- A typical approach is to divide the dot product by the square root of the dimensionality of the query and key vectors

$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

Self-Attention Networks: Transformers
Self-Attention – Parallelism

- By packing the input embeddings into a single matrix, the entire process can be parallelized by taking advantage of efficient matrix multiplication routines

$$Q = W^Q X;\ K = W^K X;\ V = W^V X$$

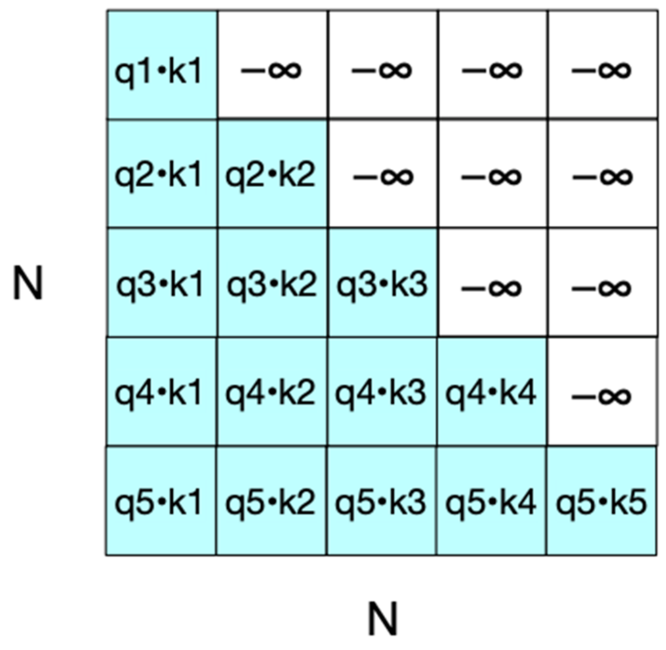
- The entire self-attention step for an entire sequence

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

  - The calculation of the comparisons in QK^T results in a score for each query value to every key value, including even those that follow the query
  - To fix this, the elements in the upper-triangular portion of the comparisons matrix are zeroed out (set to $-\infty$)

Self-Attention Networks: Transformers
Self-Attention – Parallelism

Fig. 17



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

The $N \times N$ QK^T matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

Self-Attention Networks: Transformers
Self-Attention – Parallelism

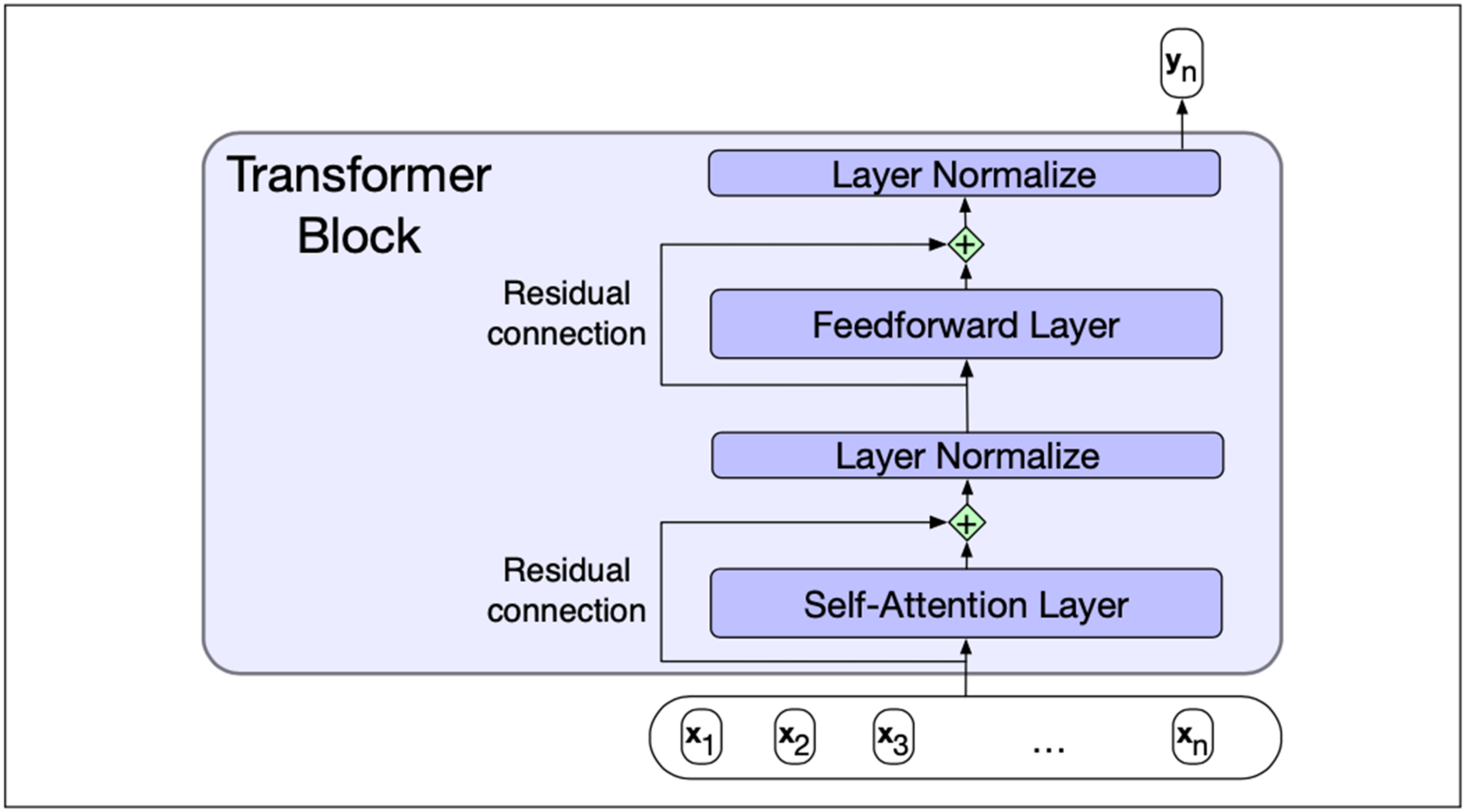
- Fig. 17 makes it clear that attention is quadratic in the length of the input, since at each layer we need to compute dot products between each pair of tokens in the input.
- This makes it extremely expensive for the input to a transformer to consist of long documents (like entire Wikipedia pages, or novels), and so most applications have to limit the input length, for example to at most a page or a paragraph of text at a time. Finding more efficient attention mechanisms is an ongoing research direction.

Self-Attention Networks: Transformers
Transformer Blocks

- The self-attention calculation lies at the core of what’s called a transformer block, which, in addition to the self-attention layer, includes additional feedforward layers, residual connections, and normalizing layers.
- The input and output dimensions of these blocks are matched so they can be stacked just as was the case for stacked RNNs.

Self-Attention Networks: Transformers
Transformer Blocks

Fig. 18



A typical transformer block consisting of a single attention layer followed by a fully-connected feedforward layer with residual connections and layer normalizations following each.

Self-Attention Networks: Transformers
Transformer Blocks

- Residual connections are connections that pass information from a lower layer to a higher layer without going through the intermediate layer.
- Allowing information from the activation going forward and the gradient going backwards to skip a layer improves learning and gives higher level layers direct access to information from lower layers

$$z = \mathrm{LayerNorm}(x + \mathrm{SelfAttn}(x))$$
$$y = \mathrm{LayerNorm}(z + \mathrm{FFNN}(z))$$

- Layer normalization uses a variation of z-score from statistics

$$LayerNorm = \gamma\left(\frac{(x-\mu)}{\sigma}\right) + \beta$$

where γ and β are learnable parameters,

μ is mean, σ is standard deviation

Self-Attention Networks: Transformers
Multihead Attention

- The different words in a sentence can relate to each other in many different ways simultaneously, e.g. distinct syntactic, semantic, and discourse relationships between verbs and their arguments in a sentence.
    - It’d be difficult for a single transformer block to learn to capture all of the different kinds of parallel relations among its inputs.
- Transformers address this issue with multihead self-attention layers
    - Sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters.
- Given these distinct sets of parameters, each head can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

Self-Attention Networks: Transformers Multihead Attention

- To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query and value matrices: W_i^K, W_i^Q and W_i^V. These are used to project the inputs to the layer, x_i, separately for each head.
- The output of a multi-head layer with h heads consists of h vectors of the same length.

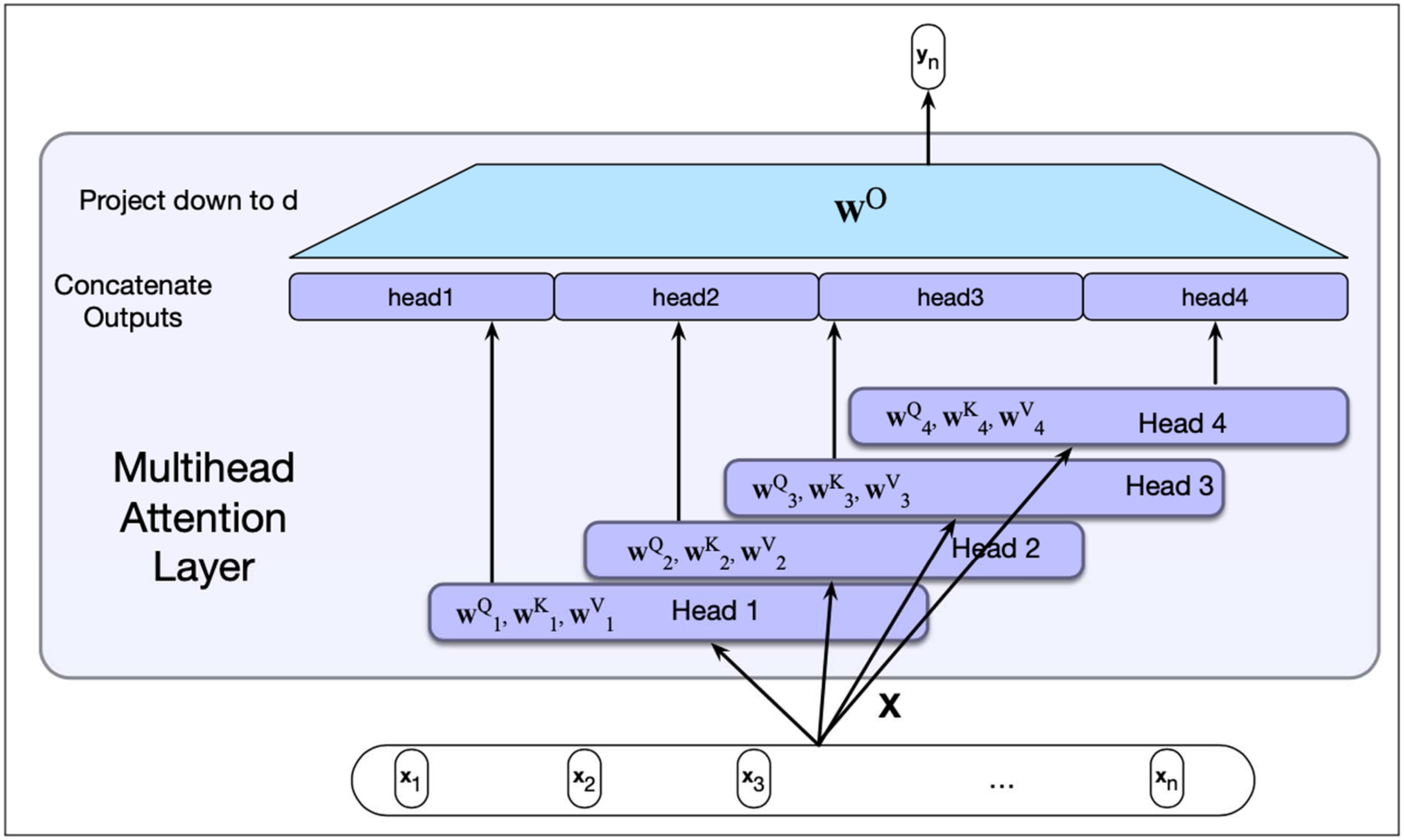
$$head_i = SelfAttention\left(W_i^Q X, W_i^K X, W_i^V X\right)$$

- They are combined and then reduced down to the original input dimension d_m by concatenating the outputs from each head and then using yet another linear projection to reduce it to the original output dimension.

$$MultiHeadAttn(Q,K,V) = W^O(head_1 \oplus head_2 \oplus \cdots \oplus head_h)$$

Self-Attention Networks: Transformers
Multihead Attention

Fig. 19



y_n
Project down to d
W^O
Concatenate Outputs
head1
head2
head3
head4
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
Multihead Attention Layer
X
x_1
x_2
x_3
...
x_n

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_{model}, thus producing an output of the same size as the input so layers can be stacked.

Self-Attention Networks: Transformers
Positional Embeddings

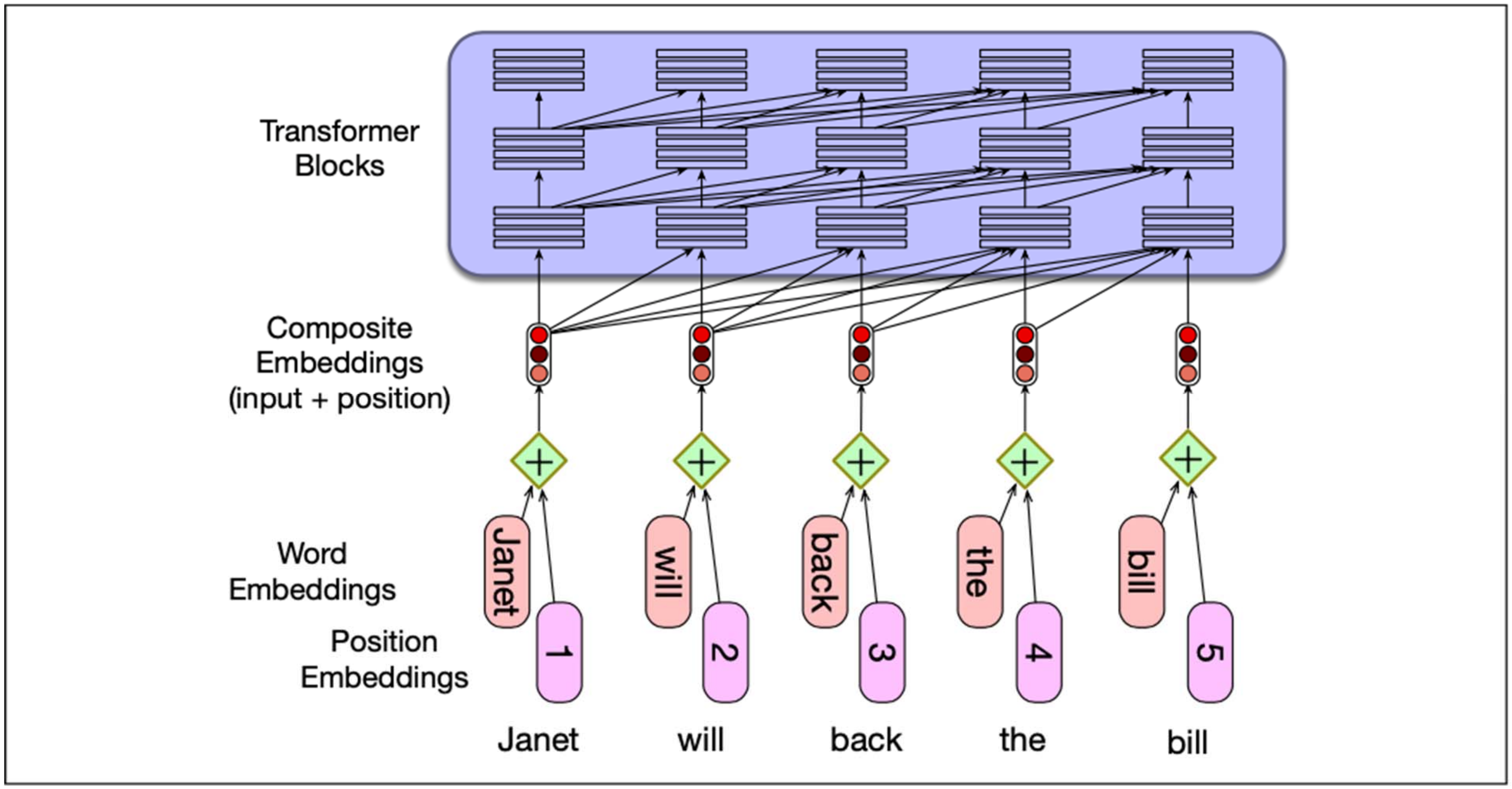
- Unlike RNNs, there’s nothing that would allow Transformers to make use of information about the relative, or absolute, positions of the elements of an input sequence.
- If you scramble the order of inputs in the attention computation illustrated earlier, you get exactly the same answer.
- To address this issue, Transformer inputs are combined with positional embeddings specific to each position in an input sequence.

Self-Attention Networks: Transformers
Positional Embeddings

- Randomly initialized embeddings corresponding to each possible input position up to some maximum length, e.g. an embedding for the position 3 just like an embedding for the word fish.
- As with word embeddings, learn these positional embeddings along with other parameters during training.
- To produce an input embedding that captures positional information, add the word embedding for each input to its corresponding positional embedding. This new embedding serves as the input for further processing.
- A potential problem: too few training examples at the outer length limits and poorly trained latter embeddings

Self-Attention Networks: Transformers
Positional Embeddings

Fig. 20



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

Self-Attention Networks: Transformers
Positional Embeddings

- Choose a static function that maps an integer inputs to real-valued vectors in a way that captures the inherent relationships among the positions, e.g. the fact that position 4 in an input is more closely related to position 5 than it is to position 17
- A combination of sine and cosine functions with differing frequencies was used in the original Transformer work

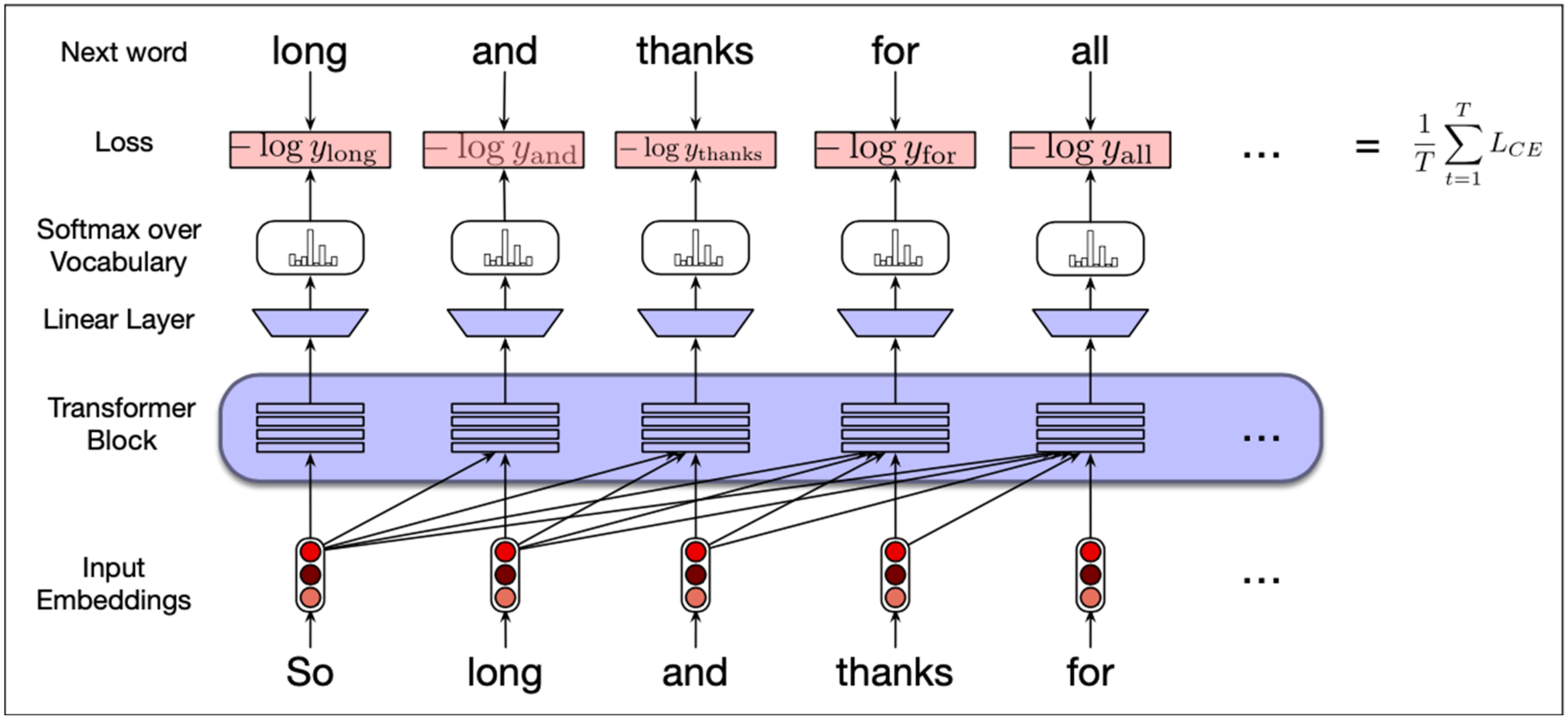
Self-Attention Networks: Transformers
Transformers as Autoregressive Language Models

Let’s examine how to deploy them as language models via semi-supervised learning.

- Proceed just as we did with the RNN-based approach: given a training corpus of plain text we’ll train a model to predict the next word in a sequence using teacher forcing.
- At each step, given all the preceding words, the final Transformer layer produces an output distribution over the entire vocabulary.
- During training, the probability assigned to the correct word is used to calculate the cross-entropy loss for each item in the sequence.
- As with RNNs, the loss for a training sequence is the average cross-entropy loss over the entire sequence.

Self-Attention Networks: Transformers
Transformers as Autoregressive Language Models

Fig. 21



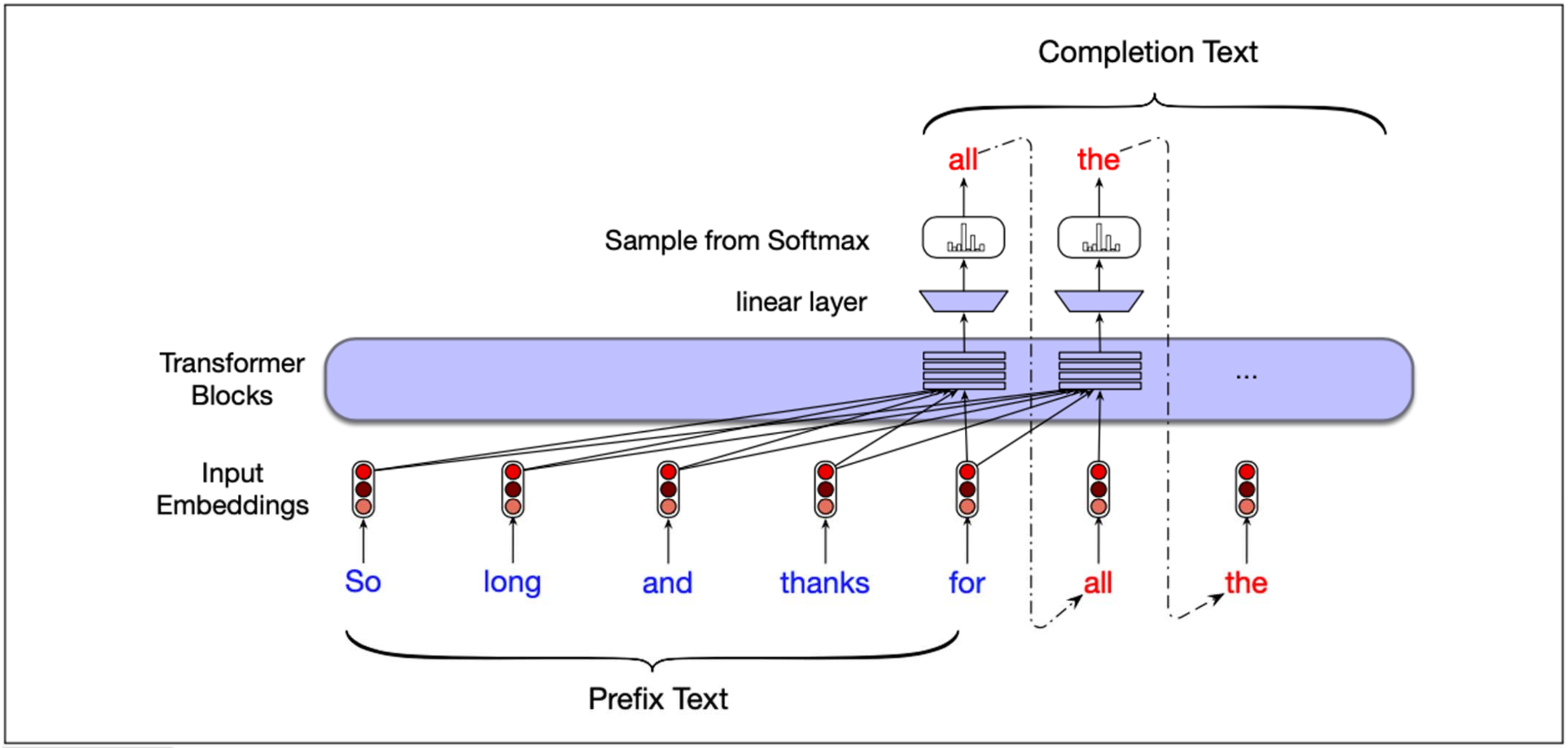
- With Transformers, each training item can be processed in parallel since the output for each element in the sequence is computed separately.
- Once trained, we can compute perplexity of the resulting model, or autoregressively generate novel text just as with RNN-based models.

Self-Attention Networks: Transformers Contextual Generation

- A simple variation on autoregressive generation that underlies a number of practical applications uses a prior context to prime the autoregressive generation process.
- A standard language model is given the prefix to some text and is asked to generate a possible completion to it.
- As the generation process proceeds, the model has direct access to the priming context as well as to all of its own subsequently generated outputs.
- This ability to incorporate the entirety of the earlier context and generated outputs at each time step is the key to the power of these models.

Self-Attention Networks: Transformers Contextual Generation

Fig. 22



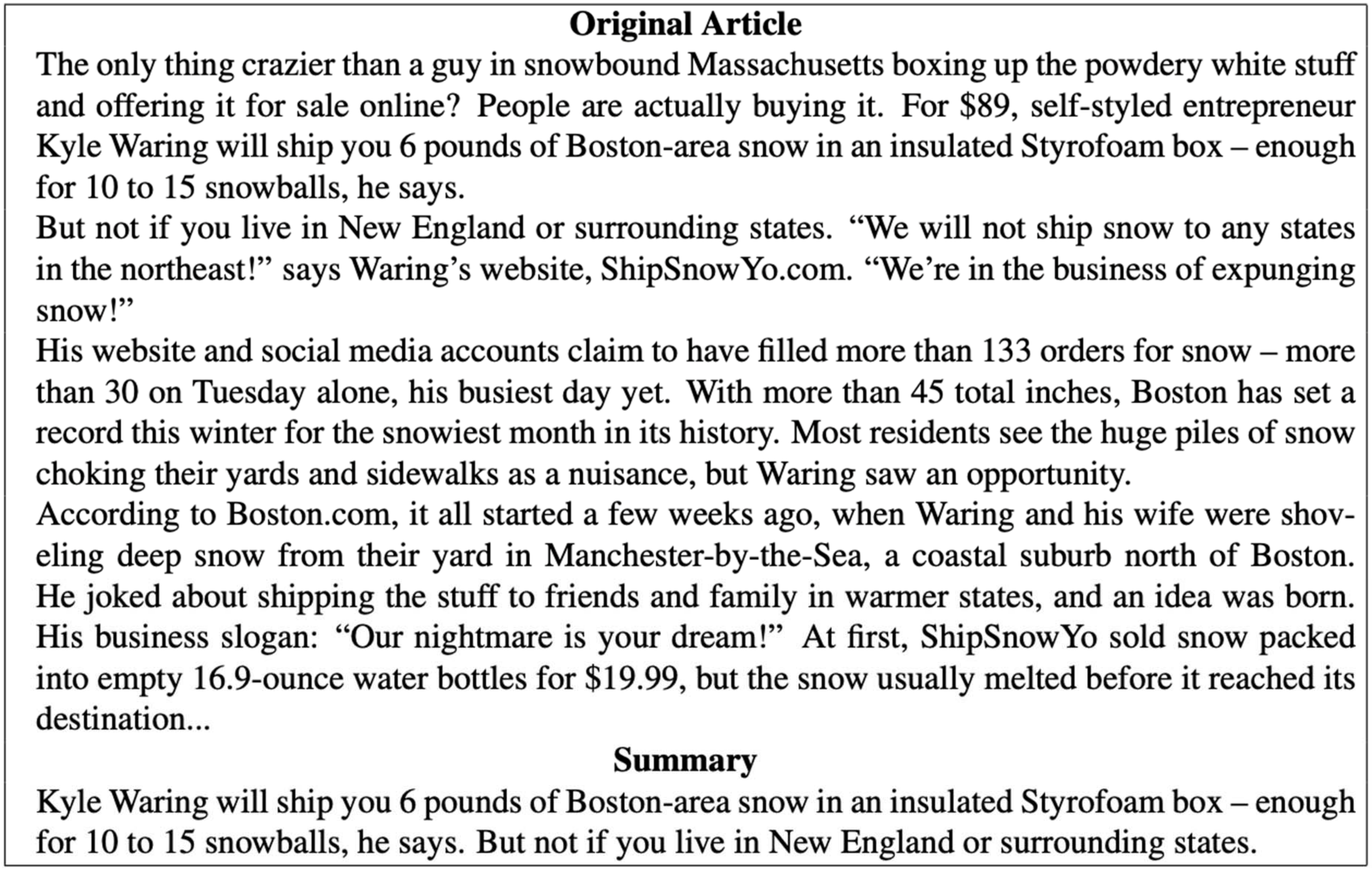
Autoregressive text completion with Transformers.

Self-Attention Networks: Transformers
Text Summarization

- Text summarization is a practical application of context-based autoregressive generation.
- The task is to take a full-length article and produce an effective summary of it.
- To train a Transformer-based autoregressive model to perform this task, start with a corpus consisting of full-length articles accompanied by their corresponding summaries.

Self-Attention Networks: Transformers
Text Summarization

Fig. 23



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!”

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.

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...

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.

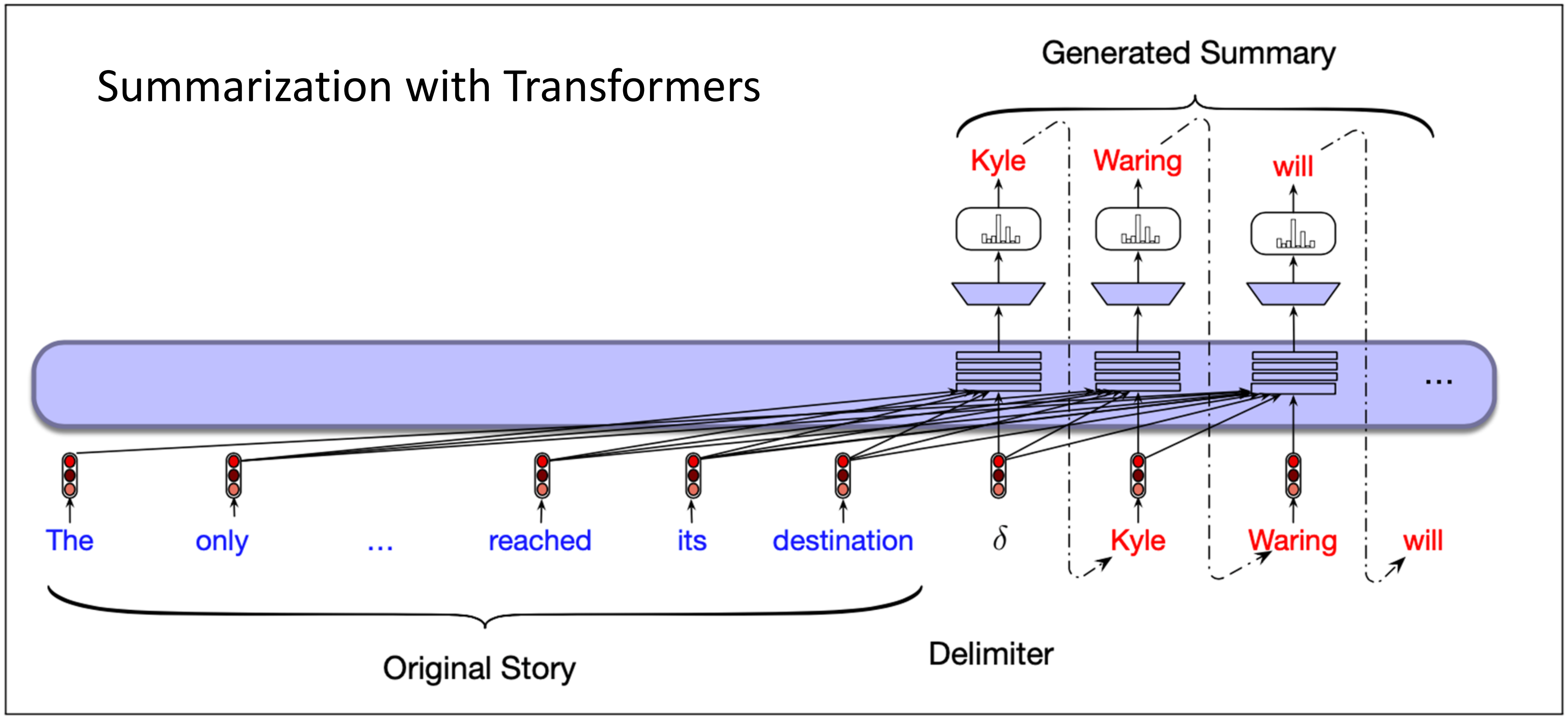
Examples of articles and summaries from the CNN/Daily Mail corpus (Hermann et al., 2015), (Nallapati et al., 2016).

Self-Attention Networks: Transformers
Text Summarization

- A surprisingly effective approach to applying Transformers to summarization is to append a summary to each full-length article in a corpus, with a unique marker separating the two.
- More formally, each article-summary pair $(x_1, \dots, x_m)$, $(y_1, \dots, y_n)$ in a training corpus is converted into a single training instance $(x_1, \dots, x_m, \delta, y_1, \dots y_n)$ with an overall length of $n + m + 1$.
- These training instances are treated as long sentences and then used to train an autoregressive language model using teacher forcing, exactly as we did earlier.
- Once trained, full articles ending with the special marker are used as the context to prime the generation process to produce a summary.

Self-Attention Networks: Transformers
Text Summarization

Fig. 24



- The model has access to the original article as well as to the newly generated text throughout the process.
- Variations on this simple scheme are the basis for successful text-to-text applications including machine translation, summarization and question answering.