CHAPTER

9 Deep Learning Architectures for Sequence Processing

Time will explain.
Jane Austen, *Persuasion*

Language is an inherently temporal phenomenon. When we comprehend and produce spoken language, we process continuous input streams of indefinite length. Even when dealing with written text, we normally process it sequentially. The temporal nature of language is reflected in the metaphors we use; we talk of the *flow of conversations*, *news feeds*, and *twitter streams*, all of which call out the notion that language is a sequence that unfolds in time.

This temporal nature is reflected in the algorithms we use to process language. For example, when applied to the problem of part-of-speech tagging, the Viterbi algorithm works its way incrementally through the input a word at a time, carrying forward information gleaned along the way. On the other hand, the machine learning approaches we've studied for sentiment analysis and other text classification tasks don't have this temporal nature – they assume simultaneous access to all aspects of their input. This is especially true of feedforward neural networks, including their application to neural language models. These fully-connected networks use fixed-size inputs, along with associated weights, to capture all the relevant aspects of an example at once. This makes it difficult to deal with sequences of varying length and fails to capture important temporal aspects of language.

A work-around for these problems is the sliding window approach employed with neural language models. These models operate by accepting fixed-sized windows of tokens as input; sequences longer than the window size are processed by walking through the input making predictions along the way, with the end result being a sequence of predictions spanning the input. Importantly, decisions made in one window have no impact on subsequent decisions. Fig. 9.1, reproduced here from Chapter 7, depicts the operation of a neural language model using this approach with a window of size 3. Here, we're predicting which word will come next given the input *for all the*. Subsequent words are predicted by sliding the window forward a word at a time.

This general approach is problematic for a number of reasons. First, it shares the primary weakness of our earlier Markov N-gram approaches in that it limits the context from which information can be extracted; anything outside the context window has no impact on the decision being made. This is an issue since there are many tasks that require access to information that can be arbitrarily distant from the point at which processing is happening. Second, the use of windows makes it difficult for networks to learn systematic patterns arising from phenomena like constituency. For example, in Fig. 9.1 the phrase *all the* appears in two separate windows: first as the second and third positions in the window, and again in the next step where it appears as the first and second positions, thus forcing the network to

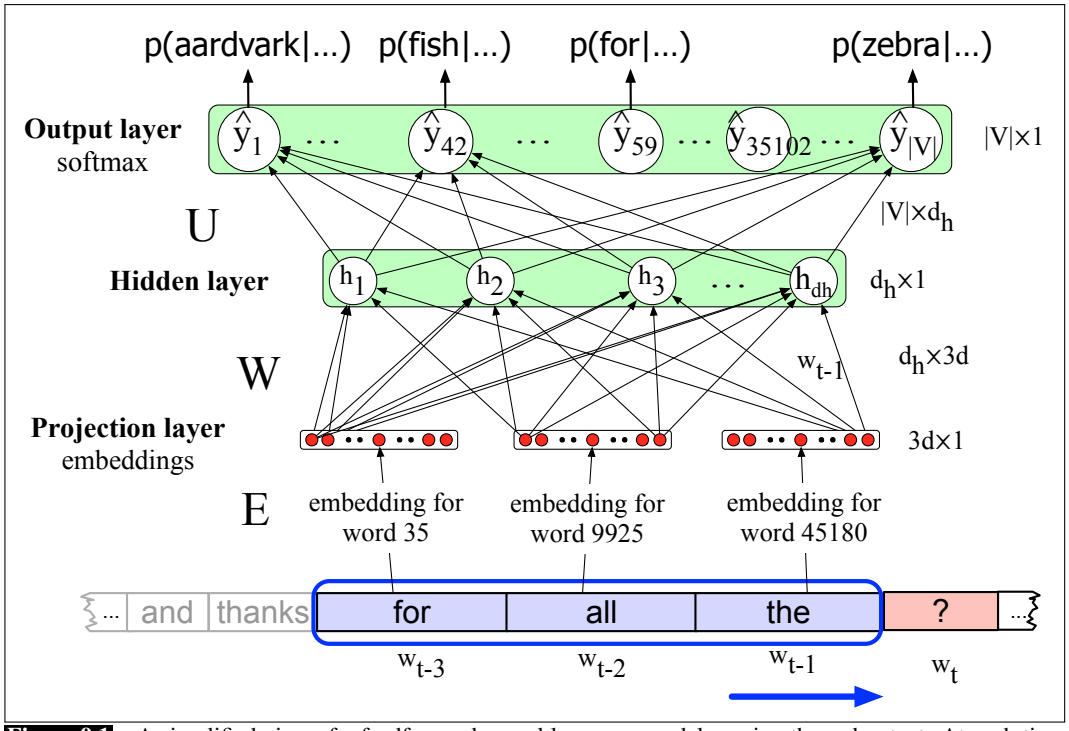

Figure 9.1 A simplified view of a feedforward neural language model moving through a text. At each time step t the network takes the 3 context words, converts each to a d-dimensional embedding, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer x for the network. The output of the network is a probability distribution over the vocabulary representing the models belief with respect to each word being the next possible word.

learn two separate patterns for what should be the same item.

This chapter covers two closely related deep learning architectures designed to address these challenges: recurrent neural networks and transformer networks. Both approaches have mechanisms to deal directly with the sequential nature of language that allow them to handle variable length inputs without the use of arbitrary fixed-sized windows, and to capture and exploit the temporal nature of language.

9.1 Language Models Revisited

In this chapter, we'll explore these two architectures primarily through the lens of probabilistic language models. Recall from Chapter 3 that probabilistic language models predict the next word in a sequence given some preceding context. For example, if the preceding context is *"Thanks for all the"* and we want to know how likely the next word is *"fish"* we would compute:

$$P(fish|Thanks\ for\ all\ the)$$

Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. We can also assign probabilities to entire sequences by using these conditional probabilities in

combination with the chain rule:

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

This formulation gives rise to a wide range of sequence labeling applications, and as we'll see, it provides a clear training objective based on how well a model is predicting the next word in a sequence.

We've already seen two ways to instantiate probabilistic language models with the N-gram models from Chapter 3 and the feedforward neural networks with sliding windows from Chapter 7. Unfortunately, both of these methods are constrained by the Markov assumption embodied in the following equation.

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{(n-N+1):(n-1)})$$

That is, the prediction is based on a fixed preceding context of size N; any input that occurred earlier than that has no bearing on the outcome. The methods we explore in this chapter will relax this assumption, allowing the models to make use of much larger contexts.

We evaluate language models by examining how well they predict unseen data drawn from the same source as the training data. Intuitively, good models are those that assign higher probabilities to unseen data. To make this intuition concrete, we use **perplexity** as a measure of model quality. The perplexity (*PP*) of a model θ with respect to an unseen test set is the probability the model assigns to it, normalized by its length.

perplexity

$$PP_\theta(w_{1:n}) = P(w_{1:n})^{\frac{1}{n}}$$

An alternative way of viewing perplexity, inspired by information theory, is in terms of entropy.

$$\begin{aligned} PP(w_{1:n}) &= 2^{H(w_{1:n})} \\ &= 2^{-\frac{1}{n}\sum_1^n log_2 m(w_n)} \end{aligned}$$

In this formulation, the value in the exponent is the cross-entropy of our current model with respect to the true distribution.

Another way to assess a language model is to use it to generate novel sequences. The extent to which a generated sequence mirrors the training data is an indication of the quality of the model. We saw how to do this in Chapter 3 by adapting a technique suggested contemporaneously by Claude Shannon (Shannon, 1951) and the psychologists George Miller and Selfridge (Miller and Selfridge, 1950). To get started, we randomly sample a word to begin a sequence based on its suitability as the start of a sequence. Having sampled the first word, we sample further words *conditioned on our previous choices* until we reach a pre-determined length, or an end of sequence token is generated. Today, this approach is called **autoregressive generation** and we'll cover its practical application in problems like machine translation and text summarization in this and later chapters.

autoregressive generation

9.2 Recurrent Neural Networks

A recurrent neural network (RNN) is any network that contains a cycle within its network connections. That is, any network where the value of a unit is directly, or indirectly, dependent on its own earlier outputs as an input. While powerful, 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 effective when applied to spoken and written language. In this section, we consider a class of recurrent networks referred to as **Elman Networks** (Elman, 1990) or **simple recurrent networks**. These networks are useful in their own right and serve as the basis for more complex approaches like the Long Short-Term Memory (LSTM) networks discussed later in this chapter. Going forward, when we use the term RNN we'll be referring to these simpler more constrained networks.

Elman Networks

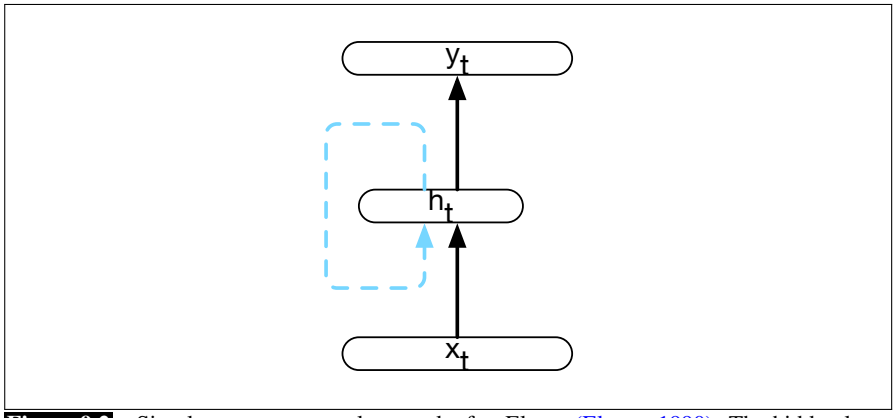

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

Fig. 9.2 illustrates the structure of an RNN. As with ordinary feedforward networks, an input vector representing the current input, x_t, is multiplied by a weight matrix and then passed through a non-linear activation function to compute the values for a layer of hidden units. This hidden layer is then used to calculate a corresponding output, y_t. In a departure from our earlier window-based approach, sequences are processed by presenting one item at a time to the network. The key difference from a feedforward 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 value of the hidden layer *from the preceding point in time*.

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. Critically, this approach 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.

Adding this temporal dimension makes RNNs appear to be more complex than non-recurrent architectures. But in reality, they're not all that different. Given an input vector and the values for the hidden layer from the previous time step, we're still performing the standard feedforward calculation introduced in Chapter 7. To see this, consider Fig. 9.3 which clarifies the nature of the recurrence and how it

factors into the computation at the hidden layer. 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 makes use of past context in calculating the output for the current input. As with the other weights in the network, these connections are trained via backpropagation.

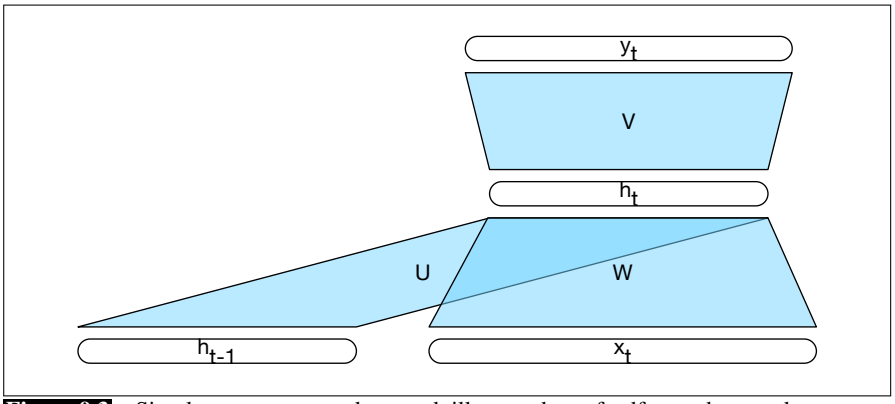

Figure 9.3 Simple recurrent neural network illustrated as a feedforward network.

9.2.1 Inference in RNNs

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. 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.

$$\begin{aligned} h_t &= g(Uh_{t-1} + Wx_t) \\ y_t &= f(Vh_t) \end{aligned}$$

It's worthwhile here to be careful about specifying the dimensions of the input, hidden and output layers, as well as the weight matrices to make sure these calculations are correct. Let's refer to the input, hidden and output layer dimensions as d_{in}, d_h, and d_{out} respectively. Given this, our three parameter matrices are: $W \in \mathbb{R}^{d_h \times d_{in}}$, $U \in \mathbb{R}^{d_h \times d_h}$, and $V \in \mathbb{R}^{d_{out} \times d_h}$.

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

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

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 in Fig. 9.4. The sequential nature of simple recurrent networks can also be seen by *unrolling* the network in time as is shown in Fig. 9.5. In this figure, 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.

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

$h_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(x) **do**
  $h_i \leftarrow g(U\ h_{i-1} + W\ x_i)$
  $y_i \leftarrow f(V\ h_i)$
return y

Figure 9.4 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.

9.2.2 Training

As with feedforward 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. 9.3, 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, and finally V, the weights from the hidden layer to the output layer.

Fig. 9.5 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*.

Tailoring the backpropagation algorithm to this situation leads to a two-pass algorithm for training the weights in RNNs. In the first pass, we perform forward inference, computing h_t, y_t, 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** (Werbos 1974, Rumelhart et al. 1986, Werbos 1990).

Backpropagation Through Time

Fortunately, with modern computational frameworks and adequate computing resources, there is no need for a specialized approach to training RNNs. As illustrated in Fig. 9.5, explicitly unrolling a recurrent network into a feedforward computational graph eliminates any explicit recurrences, allowing the network weights to be trained directly. In such an approach, we provide a template 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.

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

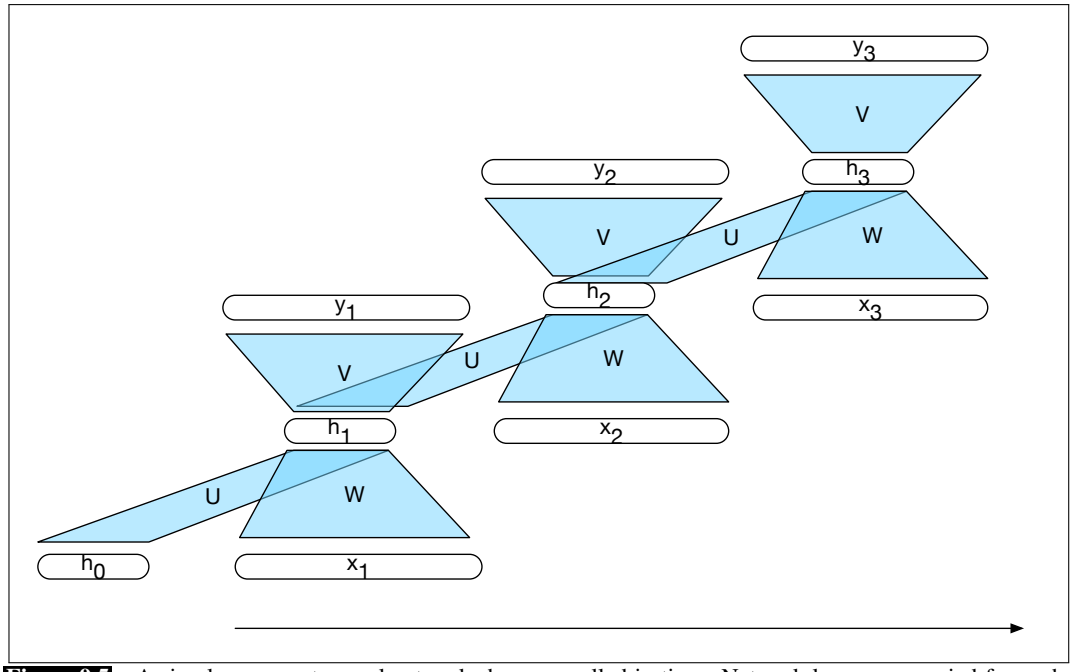

Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are copied for each time step, while the weights U, V and W are shared in common across all time steps.

into manageable fixed-length segments and treat each segment as a distinct training item.

9.2.3 RNNs as Language Models

RNN-based language models process sequences a word at a time, attempting to predict the next word in a sequence by using the current word and the previous hidden state as inputs (Mikolov et al., 2010). The limited context constraint inherent in N-gram models is avoided since the hidden state embodies information about all of the preceding words all the way back to the beginning of the sequence.

Forward inference in a recurrent language model proceeds exactly as described in Section 9.2.1. The input sequence x consists of word embeddings represented as one-hot vectors of size $|V| \times 1$, and the output predictions, y, are represented as vectors representing a probability distribution over the vocabulary. 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. That is, at time t:

$$\begin{aligned} e_t &= E^T x_t \\ h_t &= g(U h_{t-1} + W e_t) \\ y_t &= \text{softmax}(V h_t) \end{aligned}$$

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. Given y, the probability of a particular

word in the vocabulary, i, as the next word is just its corresponding component of y.

$$P(w_{t+1} = i | w_{1:t}) = y_t^i$$

It follows from this that the probability of an entire sequence is just the product of the probabilities of each item in the sequence.

$$\begin{aligned} P(w_{1:n}) &= \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \\ &= \prod_{i=1}^{n} y_{w_i}^i \end{aligned}$$

teacher forcing

To train an RNN as a language a model, we use a corpus of text as training material in combination with a training regimen called **teacher forcing**. The task is to minimize the error in predicting the 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.

$$L_{CE} = -\sum_{w \in V} y_w^t \log \hat{y}_w^t$$

In the case of language modeling, the correct distribution y comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. To be specific, at time t the CE loss is the negative log probability assigned to the next word in the training sequence.

$$L_{CE}(\hat{y}^t, y^t) = -\log \hat{y}_{w_{t+1}}^t \tag{9.1}$$

In practice, the weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent. Fig. 9.6 illustrates this training regimen.

Careful readers may have noticed that the input embedding matrix E and the final layer matrix V, which feeds the output softmax, are quite similar. The rows of E represent the word embeddings for each word in the vocabulary learned during the training process with the goal that words that have similar meaning and function will have similar embeddings. And, since the length of these embeddings corresponds to the size of the hidden layer d_h, the embedding matrix shape E is $|V| \times d_h$.

The final layer matrix V provides a way to score the likelihood of each word in the vocabulary given the evidence present in the final hidden layer of the network through the calculation of Vh. This entails that it also has the dimensionality $|V| \times d_h$. That is, the rows of V provide a *second set* of learned word embeddings that capture relevant aspects of word meaning and function. This leads to an obvious question – is it even necessary to have both? **Weight Tying** is a method that dispenses with this redundancy and uses a single set of embeddings at the input and softmax layers. That is, $E = V$. To do this, we set the dimensionality of the final hidden layer to be the same d_h, (or add an additional projection layer to do the same thing), and simply use the same matrix for both layers. In addition to providing improved perplexity results, this approach significantly reduces the number of parameters required for the model.

Weight Tying

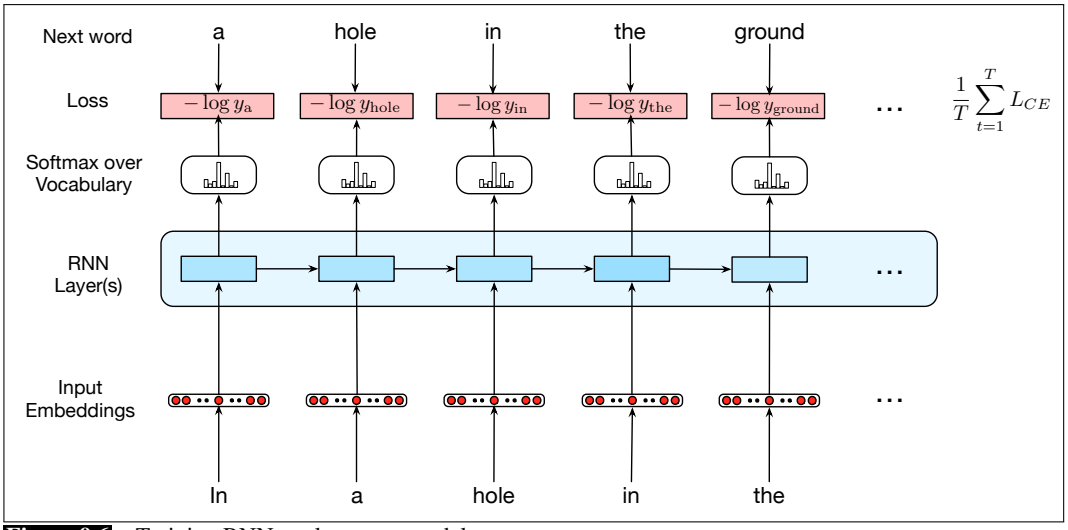

Figure 9.6 Training RNNs as language models.

Generation with RNN-Based Language Models

As with the probabilistic Shakespeare generator from Chapter 3, a useful way to gain insight into a language model is to use a trained model to generate random novel sentences. The procedure is basically the same as that described on **??**.

- To begin, 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

This technique is called **autoregressive generation** since the word generated at each time step is conditioned on the word selected by the network from the previous step. Fig. 9.7 illustrates this approach. In this figure, the details of the RNN's hidden layers and recurrent connections are hidden within the blue block.

While this is an entertaining exercise, this architecture has inspired 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. We'll discuss the application of contextual generation to the problem of summarization in Section **??** in the context of Transformer-based language models.

9.2.4 Other Applications of RNNs

Recurrent neural networks have proven to be an effective approach to language modeling, sequence labeling tasks such as part-of-speech tagging, as well as sequence classification tasks such as sentiment analysis and topic classification. And as we'll see in Chapter 11 and Chapter 11, they form the basis for sequence-to-sequence approaches to summarization, machine translation, and question answering.

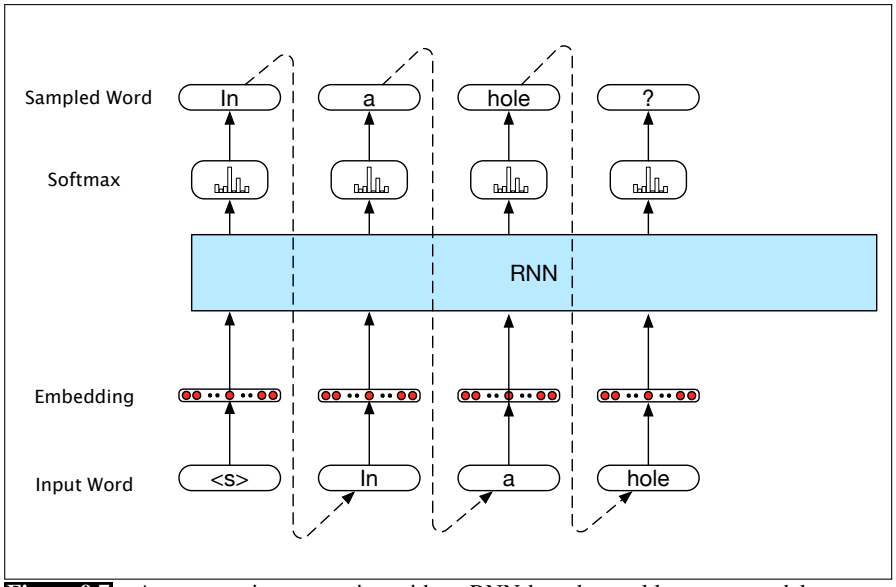

Figure 9.7 Autoregressive generation with an RNN-based neural language model.

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. Canonical examples of sequence labeling include part-of-speech tagging and named entity recognition discussed in detail in Chapter 8. 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. 9.8.

In this figure, the inputs at each time step are pre-trained word embeddings corresponding to the input tokens. The RNN block is an abstraction that represents

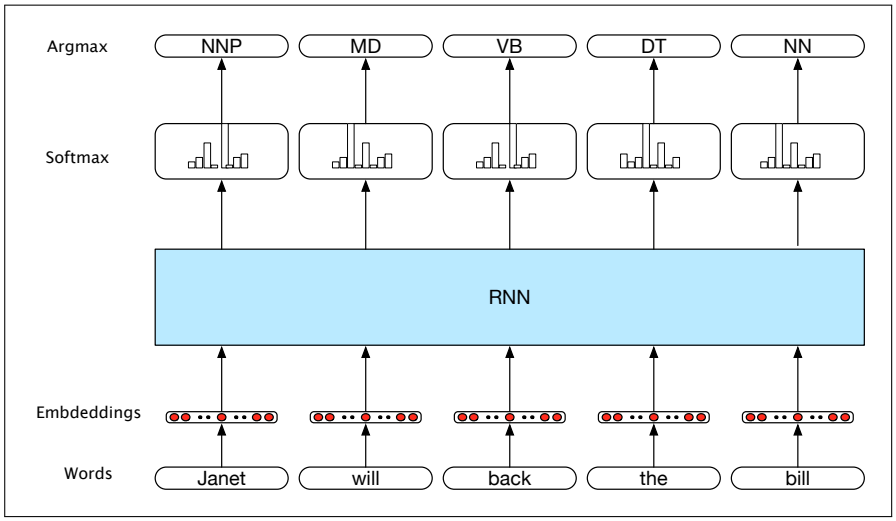

Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

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.

To generate a sequence of tags for a given input, we 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.

9.2.5 RNNs for Sequence Classification

Another use of RNNs is to classify entire sequences rather than the tokens within them. We've already encountered this task in Chapter 4 with our discussion of sentiment analysis. Other examples include document-level topic classification, spam detection, message routing for customer service applications, and deception detection. In all of these applications, sequences of text are classified as belonging to one of a small number of categories.

To apply RNNs in this setting, the text to be classified is passed 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. In the simplest approach to classification, h_n, serves as the input to a subsequent feedforward network that chooses a class via a softmax over the possible classes. Fig. 9.9 illustrates this approach.

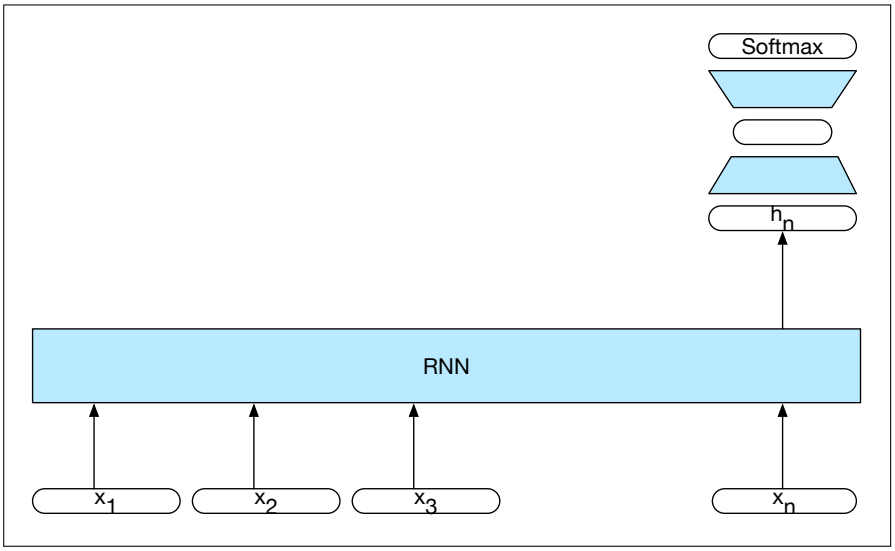

Figure 9.9 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

Note that in this approach there are no intermediate outputs for the words in the sequence preceding the last element. Therefore, there are no loss terms associated with those elements. Instead, the loss function used to train the weights in the network is based entirely on the final text classification task. Specifically, the output from the softmax output from the feedforward classifier together with a cross-entropy loss drives the training. The error signal from the classification is backprop-

agated 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 as described earlier in Section 9.2.2. This combination of a simple recurrent network with a feedforward classifier is our first example of a *deep neural network*. And 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**.

end-to-end training

9.2.6 Stacked and Bidirectional RNNs

As suggested by the sequence classification architecture shown in Fig. 9.9, recurrent networks are quite flexible. By combining the feedforward nature of unrolled computational graphs with vectors as common inputs and outputs, complex networks can be treated as modules that can be combined in creative ways. This section introduces two of the more common network architectures used in language processing with 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. 9.10.

Stacked RNNs

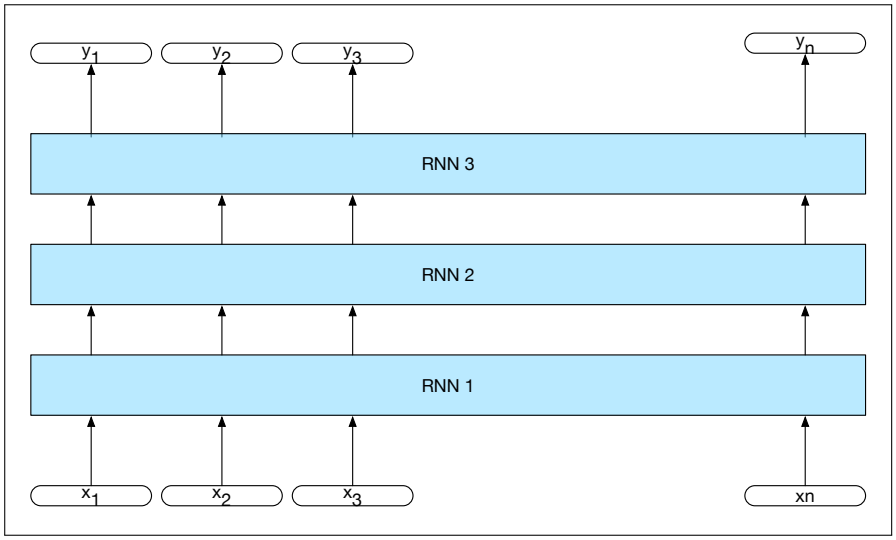

Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

It has been demonstrated across numerous tasks that stacked RNNs can 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 detect 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.

Bidirectional RNNs

In a simple recurrent network, the hidden state at a given time t represents everything the network knows about the sequence up to that point in the sequence. That is, the hidden state at time t is the result of a function of the inputs from the start up through time t. We can think of this as the context of the network to the left of the current time.

$$h_t^f = RNN_{forward}(x_1^t)$$

Where h_t^f corresponds to the normal hidden state at time t, and represents everything the network has gleaned from the sequence to that point.

In many applications we have access to the entire input sequence all at once. We might ask whether it is helpful to take advantage of the context to the right of the current input as well. One way to recover such information is to train an RNN on an input sequence in reverse, using exactly the same kind of networks that we've been discussing. With this approach, the hidden state at time t now represents information about the sequence to the right of the current input.

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

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.

bidirectional RNN

Combining the forward and backward networks results in a **bidirectional RNN** (Schuster and Paliwal, 1997). 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 combine the outputs of the two networks into a single representation that captures both the left and right contexts of an input at each point in time.

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

Fig. 9.11 illustrates a bidirectional network where the outputs of the forward and backward pass are concatenated. 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.

Bidirectional RNNs have also proven to be quite effective for sequence classification. Recall from Fig. 9.10, 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. Bidirectional RNNs provide a simple solution to this problem; as shown in Fig. 9.12, we simply combine the final hidden states from the forward and backward passes and use that as input for follow-on processing. Again, concatenation is a common approach to combining the two outputs but element-wise summation, multiplication or averaging are also used.

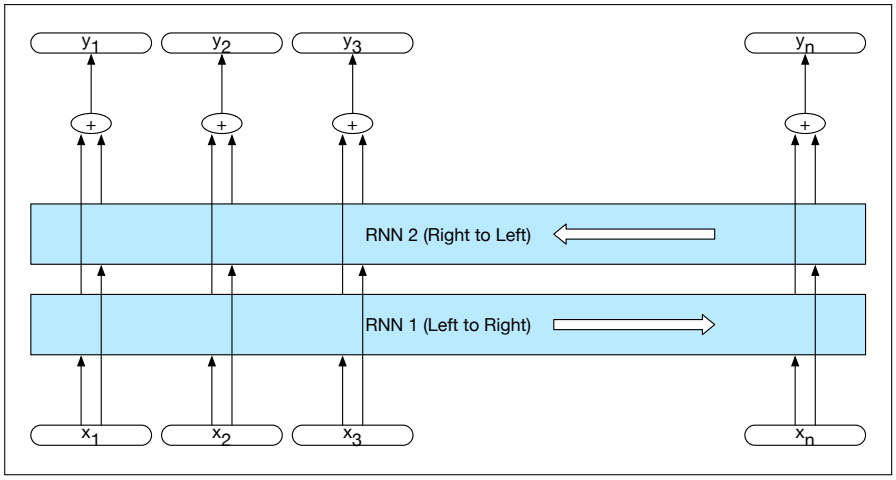

Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time. The box wrapped around the forward and backward network emphasizes the modular nature of this architecture.

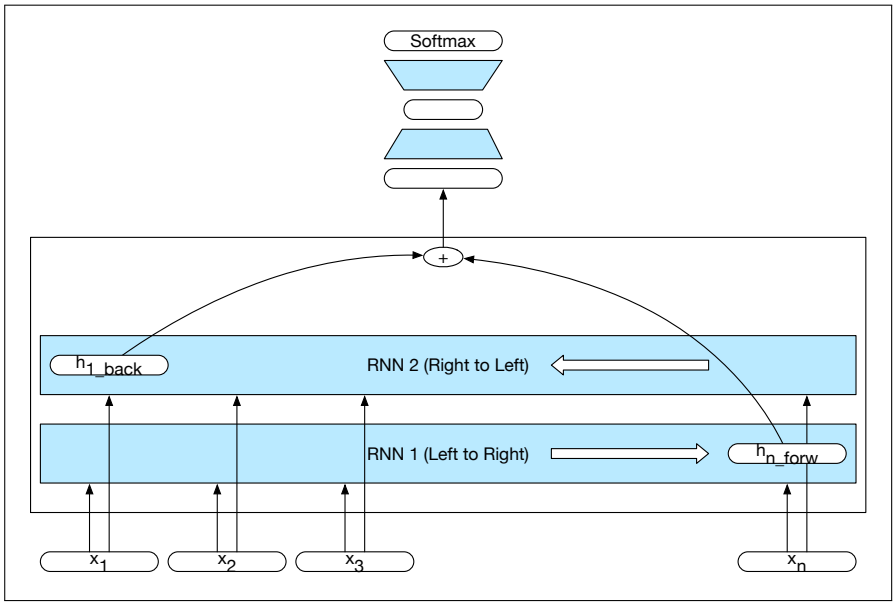

Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

9.3 Managing Context in RNNs: LSTMs and GRUs

In practice, it is quite difficult to train 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. It is often the case, however, that distant information is critical

to many language applications. To see this, consider the following example in the context of language modeling.

(9.2) 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.

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, and updating and carrying forward information required for future decisions.

A second difficulty with training SRNs arises from the need to backpropagate the error signal back through time. Recall from Section 9.2.2 that 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.

vanishing gradients

To address these issues, more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time. More specifically, the network needs to learn to forget information that is no longer needed and to remember information required for decisions still to come.

9.3.1 Long Short-Term Memory

Long short-term memory

Long short-term memory (LSTM) networks (Hochreiter and Schmidhuber, 1997) divide the context management problem into two sub-problems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), 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 that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

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.

forget gate

The first gate we'll consider is the **forget gate**. The purpose of this gate 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.

$$\begin{aligned} f_t &= \sigma(U_f h_{t-1} + W_f x_t) \\ k_t &= c_{t-1} \odot f_t \end{aligned}$$

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.

$$g_t = tanh(U_g h_{t-1} + W_g x_t) \tag{9.3}$$

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

$$i_t = \sigma(U_i h_{t-1} + W_i x_t) \tag{9.4}$$
$$j_t = g_t \odot i_t \tag{9.5}$$

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

$$c_t = j_t + k_t \tag{9.6}$$

output gate The final gate we'll use is the **output gate** which is used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

$$o_t = \sigma(U_o h_{t-1} + W_o x_t) \tag{9.7}$$
$$h_t = o_t \odot tanh(c_t) \tag{9.8}$$
$$\tag{9.9}$$

Fig. 9.13 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.

9.3.2 Gated Recurrent Units

LSTMs introduce a considerable number of additional parameters to our recurrent networks. We now have 8 sets of weights to learn (i.e., the U and W for each of the 4 gates within each unit), whereas with simple recurrent units we only had 2. Training these additional parameters imposes a much significantly higher training cost. Gated Recurrent Units (GRUs)(Cho et al., 2014) ease this burden by dispensing with the use of a separate context vector, and by reducing the number of gates to 2 — a reset gate, r and an update gate, z.

$$r_t = \sigma(U_r h_{t-1} + W_r x_t) \tag{9.10}$$
$$z_t = \sigma(U_z h_{t-1} + W_z x_t) \tag{9.11}$$

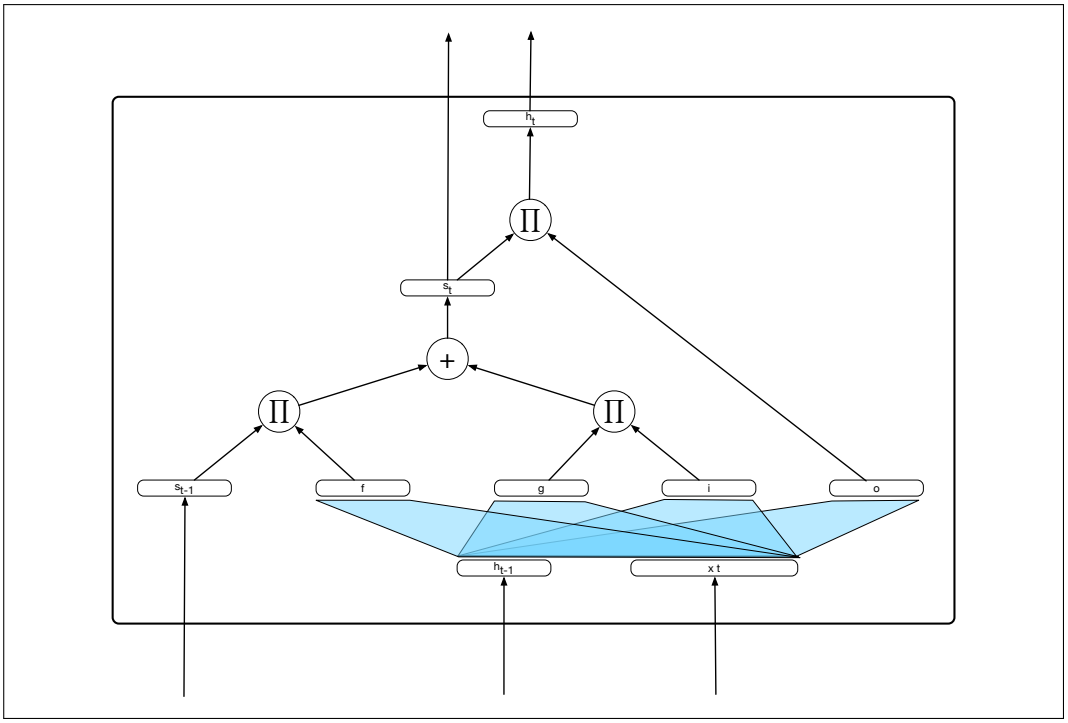

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

As with LSTMs, the use of the sigmoid in the design of these gates results in a binary-like mask that either blocks information with values near zero or allows information to pass through unchanged with values near one. The purpose of the reset gate is to decide which aspects of the previous hidden state are relevant to the current context and what can be ignored. This is accomplished by performing an element-wise multiplication of r with the value of the previous hidden state. We then use this masked value in computing an intermediate representation for the new hidden state at time t.

$$\tilde{h_t} = tanh(U(r_t \odot h_{t-1}) + Wx_t) \tag{9.12}$$

The job of the update gate z is to determine which aspects of this new state will be used directly in the new hidden state and which aspects of the previous state need to be preserved for future use. This is accomplished by using the values in z to interpolate between the old hidden state and the new one.

$$h_t = (1 - z_t)h_{t-1} + z_t\tilde{h_t} \tag{9.13}$$

9.3.3 Gated Units, Layers and Networks

The neural units used in LSTMs and GRUs 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

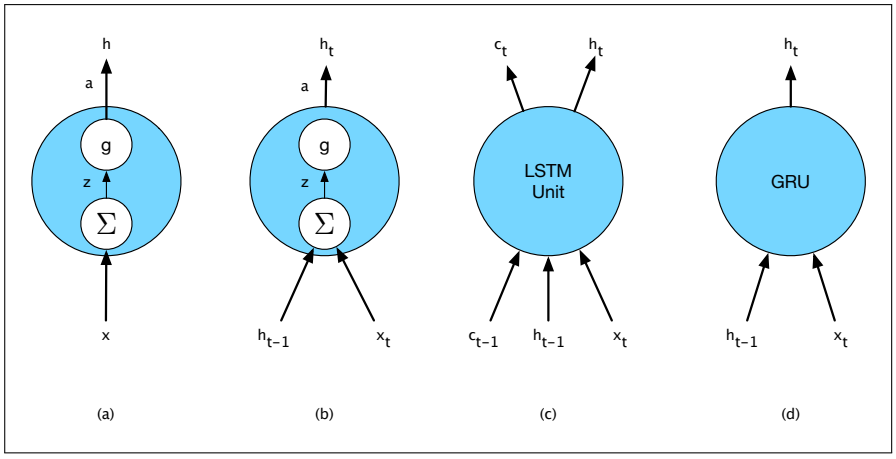

Figure 9.14 Basic neural units used in feedforward, simple recurrent networks (SRN), long short-term memory (LSTM) and gate recurrent units.

easily experiment with different architectures. To see this, consider Fig. 9.14 which illustrates the inputs and outputs associated with each kind of unit.

At the far left, (a) is the basic feedforward unit where 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. Next, (b) represents the unit in a simple recurrent network. Now 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 increased complexity of the LSTM (c) and GRU (d) units on the right is encapsulated within the units themselves. The only additional external complexity for the LSTM over the basic recurrent unit (b) 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.

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 described in Section 9.2.6. And, 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.

9.4 Self-Attention Networks: Transformers

Despite the ability of LSTMs to mitigate the loss of distant information due to the recurrence in RNNs, the underlying problem remains. Passing information forward through an extended series of recurrent connections leads to a loss of relevant information and to difficulties in training. Moreover, the inherently sequential nature of recurrent networks inhibits the use of parallel computational resources. These considerations led to the development of **Transformers** – an approach to sequence processing that eliminates recurrent connections and returns to architectures reminiscent of the fully connected networks described earlier in Chapter 7.

Transformers

Transformers map sequences of input vectors $(x_1, ..., x_n)$ to sequences of output vectors $(y_1, ..., y_n)$ of the same length. Transformers are made up of stacks of network layers consisting of simple linear layers, feedforward networks, and custom

connections around them. In addition to these standard components, the key innovation of transformers is the use of **self-attention** layers. We'll start by describing how self-attention works and then return to how it fits into larger transformer blocks. 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. In this chapter, we'll focus on the application of self-attention to the problems of language modeling and autoregressive generation where the context to be used lies in the past. We'll return to wider applications of self-attention and Transformers in later chapters.

self-attention

Fig. 9.15 illustrates the flow of information in a single causal, or backward looking, self-attention layer. As with the overall Transformer, a self-attention layer maps input sequences $(x_1, ..., x_n)$ to output sequences of the same length $(y_1, ..., y_n)$. When processing each item in the input, the model has access to all of the inputs up to an including the one under consideration, but no access to information about inputs beyond the current one. In addition, the computation performed for each item is independent of all the other computations. The first point ensures that we can use this approach to create language models and use them for autoregressive generation, and the second point means that we can easily parallelize both forward inference and training of such models.

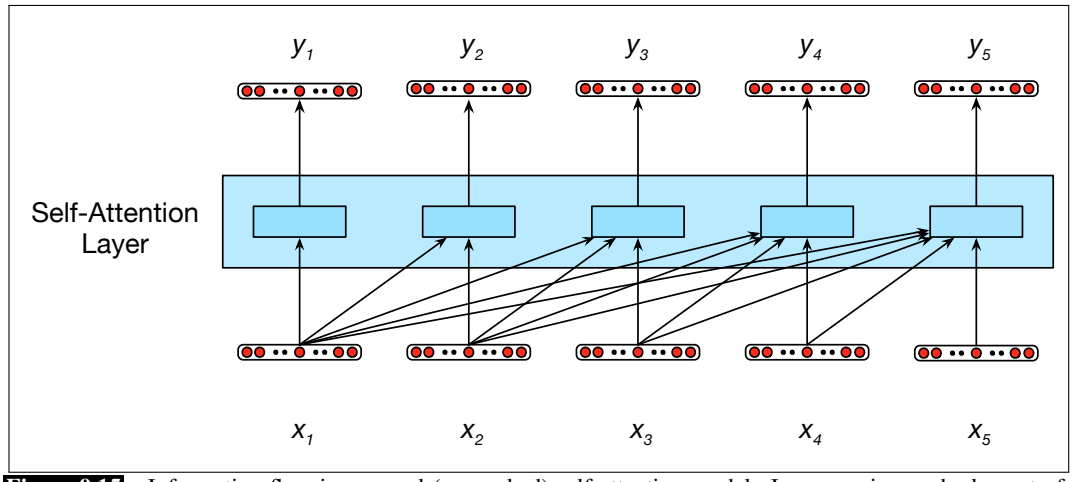

Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be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

At the core of an attention-based approach is the ability to *compare* an item of interest to a collection of other items in way that reveals their relevance in the current context. In the case of 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. For example, returning to Fig. 9.15, 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. The simplest form of comparison between elements in a self-attention layer is a dot product. To allow for other possible comparisons, let's refer to the result of these comparisons as scores.

$$score(x_i, x_j) = x_i \cdot x_j \tag{9.14}$$

The result of a dot product is a scalar value ranging from $-\infty$ to ∞, the larger the value the more similar the vectors that are being compared. Continuing with our

example, the first step in computing y_3 would be to compute three scores: $x_3 \cdot x_1$, $x_3 \cdot x_2$ and $x_3 \cdot x_3$. Then to make effective use of these scores, we'll 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.

$$\alpha_{ij} = \text{softmax}(score(x_i, x_j)) \;\; \forall j \leq i \tag{9.15}$$

$$= \frac{\exp(score(x_i, x_j))}{\sum_{k=1}^{i} \exp(score(x_i, x_k))} \;\; \forall j \leq i \tag{9.16}$$

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

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j \tag{9.17}$$

The steps embodied in Equations 9.14 through 9.17 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. In particular, there are no opportunities to learn the diverse ways that words can contribute to the representation of longer inputs. 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 different roles that each input embedding plays during the course of the attention process.

- As *the current focus of attention* when being compared to all of the other preceding inputs. We'll refer to this role as a *query*.
- In its role as *a preceding input* being compared to the current focus of attention. We'll refer to this role as a *key*.
- And finally, as *a value* used to compute the output for the current focus of attention.

To capture the different roles that input embeddings play in each of these steps, Transformers introduce three sets of weights which we'll call W^Q, W^K, and W^V. These weights will be used to compute linear transformations of each input x with the resulting values being used in their respective roles in subsequent calculations.

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

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.

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. Let's update our previous comparison calculation to reflect this.

$$score(x_i, x_j) = q_i \cdot k_j \tag{9.18}$$

The ensuing softmax calculation resulting in $\alpha_{i,j}$ remains the same, but the output calculation for y_i is now based on a weighted sum over the value vectors v.

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j \tag{9.19}$$

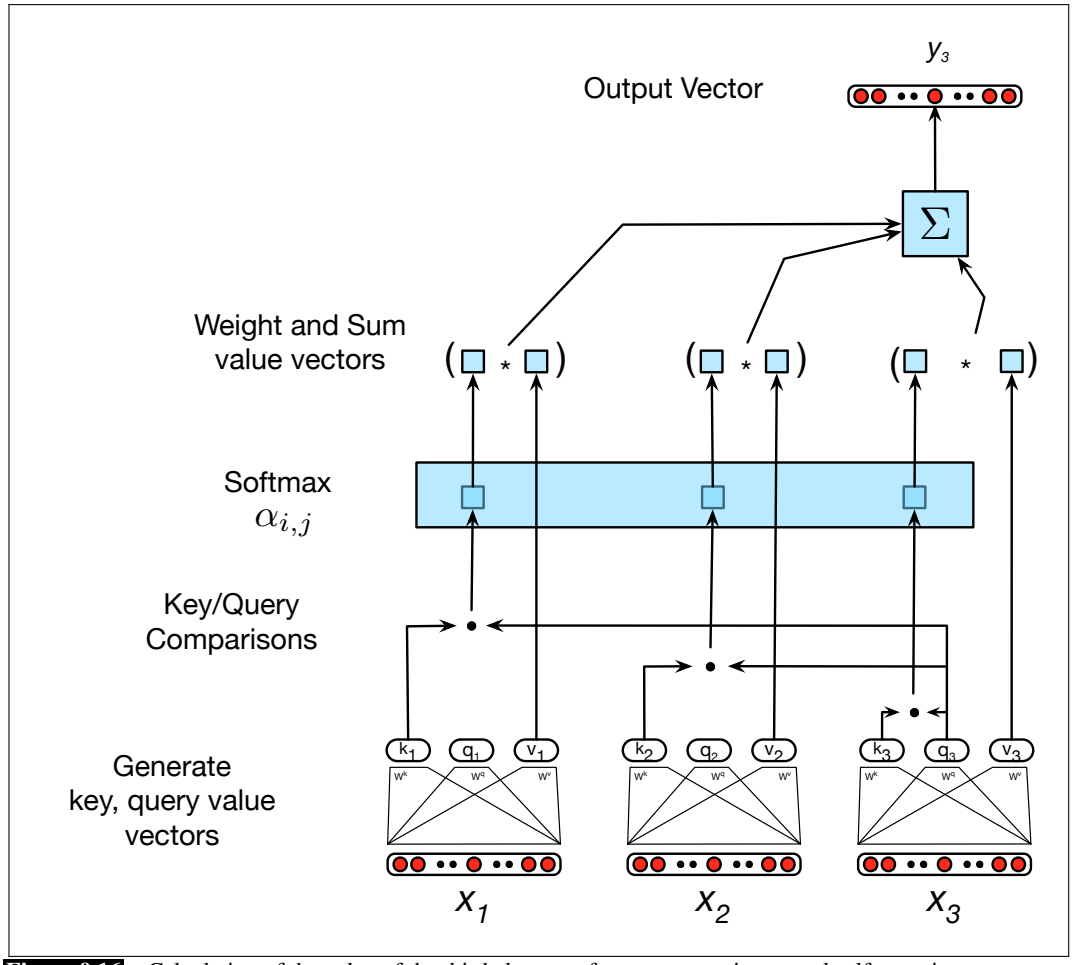

Figure 9.16 Calculation of the value of the third element of a sequence using causal self-attention.

Fig. 9.16 illustrates this calculation in the case of computing the third output y_3 in a sequence.

A practical consideration that arises in computing α_{ij} arises from the use of a dot product as a comparison in combination with the exponential in the softmax. The result of dot product can be an arbitrarily large (positive or negative) value. Exponentiating such large values can lead to numerical issues and to an effective loss of gradients during training. To avoid this, the dot product needs to be scaled in a suitable fashion. 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, leading us to update our scoring function one more time.

$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}} \tag{9.20}$$

This description of the self-attention process has been from the perspective of computing a single output at a particular point in time. However, since each output, y_i, is computed independently this entire process can be parallelized by taking advantage of efficient matrix multiplication routines by packing the input embed-

dings into a single matrix and multiplying it by the key, query and value matrices to produce matrices containing all the key, query and value vectors.

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

Given these matrices we can compute all the requisite query-key comparisons simultaneously by multiplying K and Q in a single matrix multiplication. Taking this one step further, we can scale these scores, take the softmax, and then multiply the result by V, thus reducing the entire self-attention step for an entire sequence to the following computation.

$$SelfAttention(Q,K,V) \;=\; \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{9.21}$$

Unfortunately, this process goes a bit too far since the calculation of the comparisons in QK^T results in a score for each query value to every key value, *including those that follow the query*. This is inappropriate in the setting of language modeling since guessing the next word is pretty simple if you already know it. To fix this, the elements in the upper-triangular portion of the comparisons matrix are zeroed out (set to $-\infty$), thus eliminating any knowledge of words that follow in the sequence.

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. Fig. 9.17 illustrates 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. These blocks can then be stacked just as was the case for stacked RNNs.

Multihead Attention

The different words in a sentence can relate to each other in many different ways simultaneously. For example, distinct syntactic, semantic, and discourse relationships can hold between verbs and their arguments in a sentence. It would 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**. These are 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.

multihead self-attention layers

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, with the rest of the self-attention computation remaining unchanged. The output of a multi-head layer with h heads consists of h vectors of the same length. To make use of these vectors in further processing, they are combined and then reduced down to the original input dimension d_m. This is accomplished by concatenating the outputs from each head and then using yet another linear projection to reduce it to the original output dimension.

$$\begin{aligned} MultiHeadAttn(Q,K,V) &= W^O(head_1 \oplus head_2 ... \oplus head_h) \\ head_i &= SelfAttention(W_i^Q X, W_i^K X, W_i^V X) \end{aligned}$$

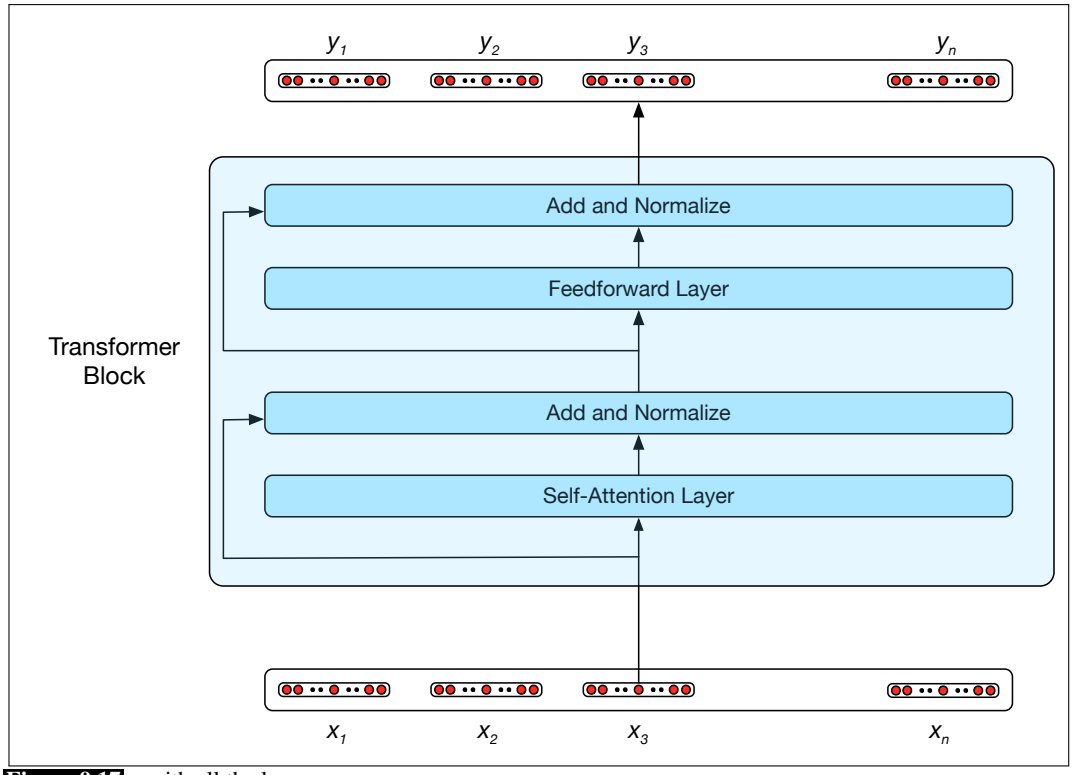

Figure 9.17 with all the layers

Fig. 9.18 illustrates this approach with 4 self-attention heads. This multihead layer replaces the single self-attention layer in the transformer block shown earlier in Fig. 9.17, the rest of the Transformer block with its feedforward layer, residual connections, and layer norms remains the same.

Positional Embeddings

With RNNs information about the order of the inputs was baked into the nature of the models. Unfortunately, the same isn't true for Transformers; there's nothing that would allow such models to make use of information about the relative, or absolute, positions of the elements of an input sequence. This can be seen from the fact that 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.

positional embeddings

Where do we get these positional embeddings? A simple and effective approach is to start with randomly initialized embeddings corresponding to each possible input position up to some maximum length. For example, just as we have an embedding for the word *fish*, we'll have an embedding for the position 3. As with word embeddings, these positional embeddings are learned along with other parameters during training. To produce an input embedding that captures positional information, we just 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 with this approach is that there will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits. These latter embeddings may be poorly trained and may not

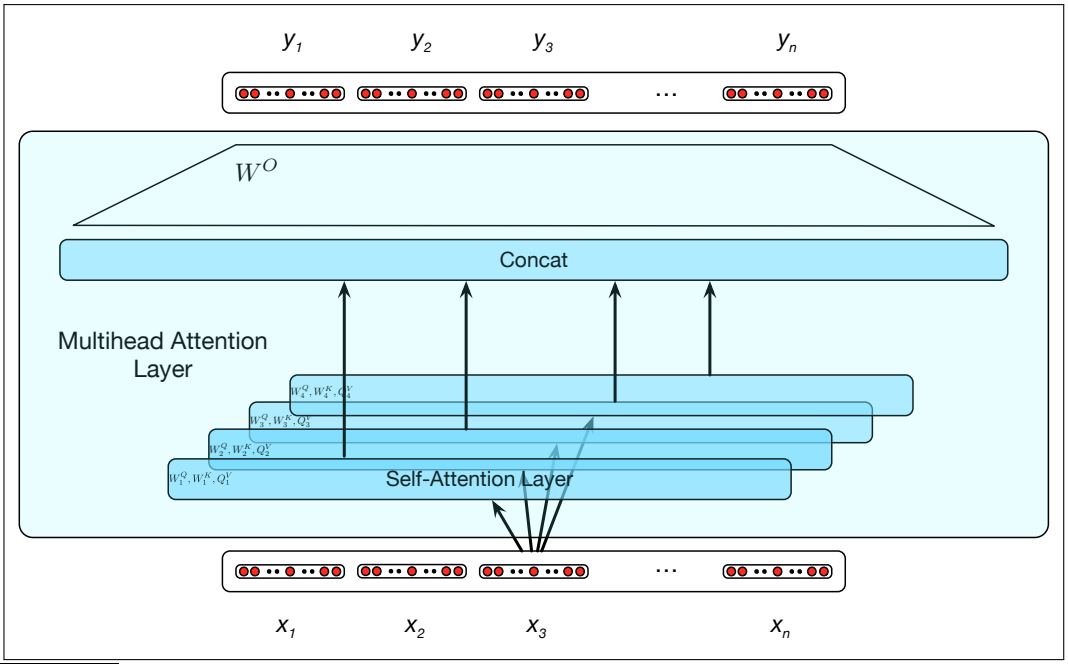

Figure 9.18 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d_{model}, thus producing an output of the right size.

generalize well during testing. An alternative approach to positional embeddings is to choose a static function that maps an integer inputs to real-valued vectors in a way that captures the inherent relationships among the positions. That is, it captures 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.

9.4.1 Transformers as Autoregressive Language Models

Now that we've seen all the major components of Transformers, let's examine how to deploy them as language models via semi-supervised learning. To do this, we'll 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. Fig. 9.19 illustrates the general approach. 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.

Note the key difference between this figure and the earlier RNN-based version for shown in Fig. 9.6. There the calculation of the outputs and the losses at each step was inherently serial given the recurrence in the calculation of the hidden states. 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 the perplexity of the resulting model, or autoregressively generate novel text just as with RNN-based models.

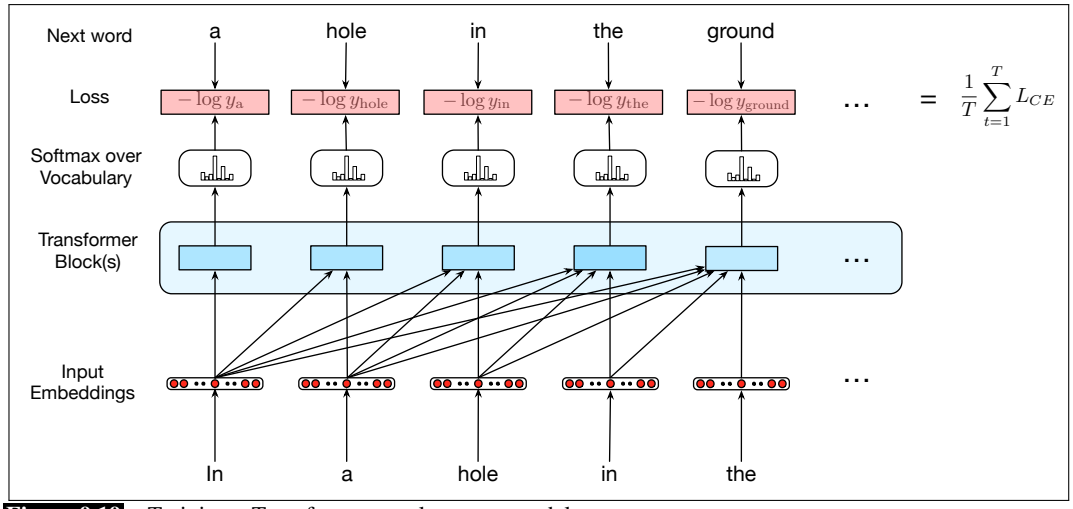

Figure 9.19 Training a Transformer as a language model.

Contextual Generation and Summarization

A simple variation on autoregressive generation that underlies a number of practical applications uses a prior context to prime the autoregressive generation process. Fig. 9.20 illustrates this with the task of text completion. Here a standard language model is given the prefix to some text and is asked to generate a possible completion to it. Note that 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.

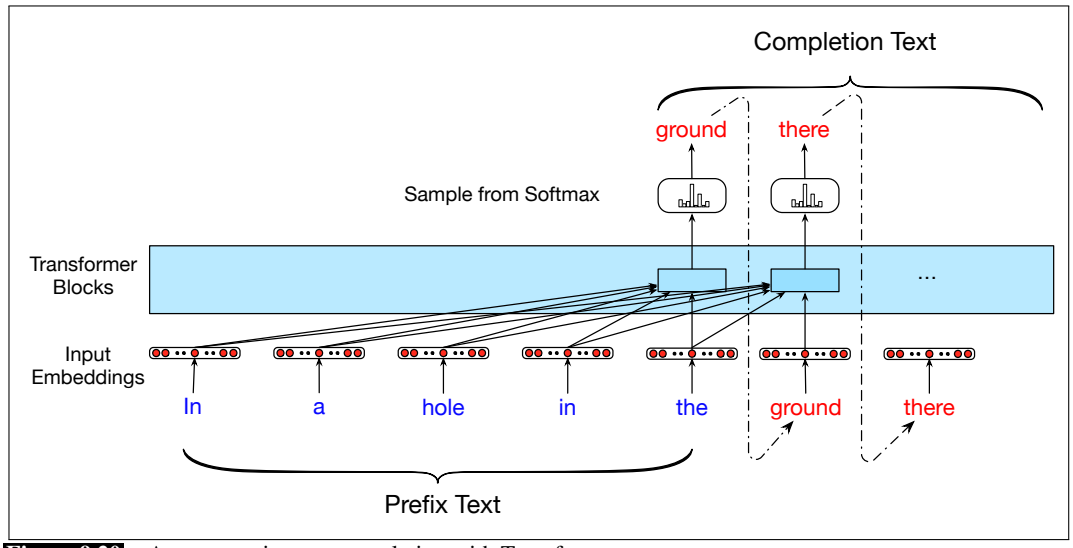

Figure 9.20 Autoregressive text completion with Transformers.

Text summarization

Text summarization is a practical application of context-based autoregressive generation. Here, 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, we start with a corpus consisting of full-length articles accompanied by their

corresponding summaries. Fig. 9.21 shows an example of this kind of data from a widely used summarization corpus consisting of CNN and Daily Mirror news articles.

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.

Figure 9.21 Examples of articles and summaries from the CNN/Daily Mail corpus (Hermann et al., 2015), (Nallapati et al., 2016).

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 $(x_1,...,x_m)$, $(y_1,...,y_n)$ in a training corpus is converted into a single training instance $(x_1,...,x_m,\delta,y_1,...y_n)$ 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 as illustrated in Fig. 9.22. Note that, in contrast to RNNs, the model has access to the original article as well as to the newly generated text throughout the process.

As we'll see in later chapters, variations on this simple scheme are the basis for successful text-to-text applications including machine translation, summarization and question answering.

9.5 Potential Harms from Language Models

Large neural language models exhibit many of the potential harms discussed in Chapter 4 and Chapter 6. Problems may occur whenever language models are used for text generation, such as in assistive technologies like web search query completion or predictive typing for email (Olteanu et al., 2020).

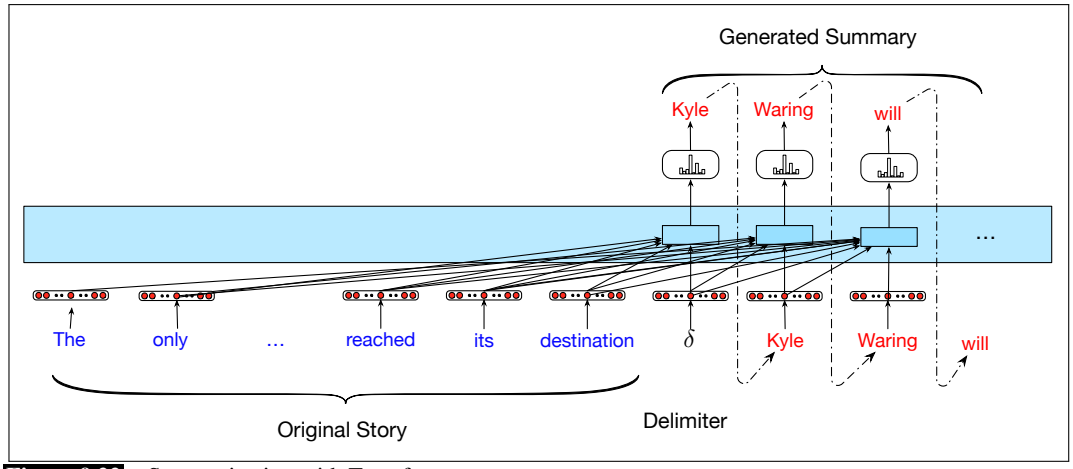

Figure 9.22 Summarization with Transformers.

For example, language models can generate toxic language. Gehman et al. (2020) show that many kinds of completely non-toxic prompts can nonetheless lead large language models to output hate speech and abuse. Brown et al. (2020) and Sheng et al. (2019) showed that large language models generate sentences displaying negative attitudes toward minority identities such as being Black or gay.

Indeed, language models are biased in a number of ways by the distributions of their training data. Gehman et al. (2020) shows that large language model training datasets include toxic text scraped from banned sites. In addition to problems of toxicity, internet data is disproportionally generated by authors from developed countries, and many large language models train on data from Reddit, whose authors skew male and young. Such biased population samples likely skew the resulting generation away from the perspectives or topics of underrepresented populations. Furthermore, language models can amplify demographic and other biases in training data, just as we saw for embedding models in Chapter 6.

Language models can also be a tool for generating text for misinformation, phishing, radicalization, and other socially harmful activities (Brown et al., 2020). (McGuffie and Newhouse, 2020) show how large language models generate text that emulates online extremists, with the risk of amplifying extremist movements and their attempt to radicalize and recruit.

Finally, there are important privacy issues. Language models, like other machine learning models, can **leak** information about their training data. It is thus possible for an adversary to extract individual training-data phrases from a language model such as an individual person's name, phone number, and address (Carlini et al. 2020, using the techniques introduced by Henderson et al. 2017). This is a problem if large language models are trained on private datasets such has electronic health records (EHRs).

Mitigating all these harms is an important but unsolved research question in NLP. Extra pre-training (Gururangan et al., 2020) on non-toxic subcorpora seems to reduce a language model's tendency to generate toxic language somewhat (Gehman et al., 2020). And analyzing the data used to pretrain large language models is important to understand toxicity and bias in generation, as well as privacy, making it extremely important that language models include **datasheets** (page **??**) or **model cards** (page **??**) giving full replicable information on the corpora used to train them.

9.6 Summary

This chapter has introduced the concept of recurrent neural networks and how they can be applied to language problems. Here's a summary of the main points that we covered:

- In simple Recurrent Neural Networks sequences are processed naturally as an element at a time.
- The output of a neural unit at a particular point in time is based both on the current input and value of the hidden layer from the previous time step.
- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as backpropagation through time (BPTT).
- Common language-based applications for RNNs include:
  - Probabilistic language modeling, where the model assigns a probability to a sequence, or to the next element of a sequence given the preceding words.
  - Auto-regressive generation using a trained language model.
  - Sequence labeling, where each element of a sequence is assigned a label, as with part-of-speech tagging.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.
- Simple recurrent networks often fail since it is extremely difficult to successfully train them do to problems maintaining useful gradients over time.
- More complex gated architectures such as LSTMs and GRUs are designed to overcome these issues by explicitly managing the task of deciding what to remember and forget in their hidden and context layers.

Bibliographical and Historical Notes

Influential investigations of the kind of simple RNNs discussed here were conducted in the context of the Parallel Distributed Processing (PDP) group at UC San Diego in the 1980's. Much of this work was directed at human cognitive modeling rather than practical NLP applications Rumelhart and McClelland 1986 McClelland and Rumelhart 1986. Models using recurrence at the hidden layer in a feedforward network (Elman networks) were introduced by Elman (1990). Similar architectures were investigated by Jordan (1986) with a recurrence from the output layer, and Mathis and Mozer (1995) with the addition of a recurrent context layer prior to the hidden layer. The possibility of unrolling a recurrent network into an equivalent feedforward network is discussed in (Rumelhart and McClelland, 1986).

In parallel with work in cognitive modeling, RNNs were investigated extensively in the continuous domain in the signal processing and speech communities (Giles et al., 1994). Schuster and Paliwal (1997) introduced bidirectional RNNs and described results on the TIMIT phoneme transcription task.

While theoretically interesting, the difficulty with training RNNs and managing context over long sequences impeded progress on practical applications. This situation changed with the introduction of LSTMs in Hochreiter and Schmidhuber

(1997). Impressive performance gains were demonstrated on tasks at the boundary of signal processing and language processing including phoneme recognition (Graves and Schmidhuber, 2005), handwriting recognition (Graves et al., 2007) and most significantly speech recognition (Graves et al., 2013).

Interest in applying neural networks to practical NLP problems surged with the work of Collobert and Weston (2008) and Collobert et al. (2011). These efforts made use of learned word embeddings, convolutional networks, and end-to-end training. They demonstrated near state-of-the-art performance on a number of standard shared tasks including part-of-speech tagging, chunking, named entity recognition and semantic role labeling without the use of hand-engineered features.

Approaches that married LSTMs with pre-trained collections of word-embeddings based on word2vec (Mikolov et al., 2013) and GLOVE (Pennington et al., 2014), quickly came to dominate many common tasks: part-of-speech tagging (Ling et al., 2015), syntactic chunking (Søgaard and Goldberg, 2016), and named entity recognition via IOB tagging Chiu and Nichols 2016, Ma and Hovy 2016, opinion mining (Irsoy and Cardie, 2014), semantic role labeling (Zhou and Xu, 2015) and AMR parsing (Foland and Martin, 2016). As with the earlier surge of progress involving statistical machine learning, these advances were made possible by the availability of training data provided by CONLL, SemEval, and other shared tasks, as well as shared resources such as Ontonotes (Pradhan et al., 2007), and PropBank (Palmer et al., 2005).

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.

Carlini, N., Tramer, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, U., Oprea, A., and Raffel1, C. (2020). Extracting training data from large language models. arXiv preprint arXiv:2012.07805.

Chiu, J. P. C. and Nichols, E. (2016). Named entity recognition with bidirectional LSTM-CNNs. *TACL 4*, 357–370.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. *EMNLP*.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. *ICML*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *JMLR 12*, 2493–2537.

Elman, J. L. (1990). Finding structure in time. *Cognitive science 14*(2), 179–211.

Foland, W. and Martin, J. H. (2016). CU-NLP at semeval-2016 task 8: AMR parsing using lstm-based recurrent neural networks. *Proceedings of the 10th International Workshop on Semantic Evaluation*.

Gehman, S., Gururangan, S., Sap, M., Choi, Y., and Smith, N. A. (2020). RealToxicityPrompts: Evaluating neural toxic degeneration in language models. *Findings of EMNLP*.

Giles, C. L., Kuhn, G. M., and Williams, R. J. (1994). Dynamic recurrent neural networks: Theory and applications. *IEEE Trans. Neural Netw. Learning Syst. 5*(2), 153–156.

Graves, A., Fernández, S., Liwicki, M., Bunke, H., and Schmidhuber, J. (2007). Unconstrained on-line handwriting recognition with recurrent neural networks. *NeurIPS*.

Graves, A., Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks 18*(5-6), 602–610.

Gururangan, S., Marasović, A., Swayamdipta, S., Lo, K., Beltagy, I., Downey, D., and Smith, N. A. (2020). Don't stop pretraining: Adapt language models to domains and tasks. *ACL*.

Henderson, P., Sinha, K., Angelard-Gontier, N., Ke, N. R., Fried, G., Lowe, R., and Pineau, J. (2017). Ethical challenges in data-driven dialogue systems. *AAAI/ACM AI Ethics and Society Conference*.

Hermann, K. M., Kočiský, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. (2015). Teaching machines to read and comprehend. *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. MIT Press.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation 9*(8), 1735–1780.

Irsoy, O. and Cardie, C. (2014). Opinion mining with deep recurrent neural networks. *EMNLP*.

Jordan, M. (1986). Serial order: A parallel distributed processing approach. Tech. rep. ICS Report 8604, University of California, San Diego.

Ling, W., Dyer, C., Black, A. W., Trancoso, I., Fermandez, R., Amir, S., Marujo, L., and Luís, T. (2015). Finding function in form: Compositional character models for open vocabulary word representation. *EMNLP*.

Ma, X. and Hovy, E. H. (2016). End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. *ACL*.

Mathis, D. A. and Mozer, M. C. (1995). On the computational utility of consciousness. Tesauro, G., Touretzky, D. S., and Alspector, J. (Eds.), *Advances in Neural Information Processing Systems VII*. MIT Press.

McClelland, J. L. and Rumelhart, D. E. (Eds.). (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 2: *Psychological and Biological Models*. MIT Press.

McGuffie, K. and Newhouse, A. (2020). The radicalization risks of GPT-3 and advanced neural language models. arXiv preprint arXiv:2009.06807.

Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. (2013). Efficient estimation of word representations in vector space. *ICLR 2013*.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. *INTERSPEECH 2010*.

Miller, G. A. and Selfridge, J. A. (1950). Verbal context and the recall of meaningful material. *American Journal of Psychology 63*, 176–185.

Nallapati, R., Zhou, B., dos Santos, C., GuÌ‡lçehre, Ç., and Xiang, B. (2016). Abstractive text summarization using sequence-to-sequence RNNs and beyond. *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. Association for Computational Linguistics.

Olteanu, A., Diaz, F., and Kazai, G. (2020). When are search completion suggestions problematic?. *CSCW*.

Palmer, M., Kingsbury, P., and Gildea, D. (2005). The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics 31*(1), 71–106.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. *EMNLP*.

Pradhan, S., Hovy, E. H., Marcus, M. P., Palmer, M., Ramshaw, L. A., and Weischedel, R. M. (2007). Ontonotes: a unified relational semantic representation. *Int. J. Semantic Computing 1*(4), 405–419.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. Rumelhart, D. E. and McClelland, J. L. (Eds.), *Parallel Distributed Processing*, Vol. 2, 318–362. MIT Press.

Rumelhart, D. E. and McClelland, J. L. (Eds.). (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: *Foundations*. MIT Press.

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing 45*, 2673–2681.

Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell System Technical Journal 30*, 50–64.

Sheng, E., Chang, K.-W., Natarajan, P., and Peng, N. (2019). The woman worked as a babysitter: On biases in language generation. *EMNLP*.

Søgaard, A. and Goldberg, Y. (2016). Deep multi-task learning with low level tasks supervised at lower layers. *ACL*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *NeurIPS*.

Werbos, P. (1974). *Beyond regression: new tools for prediction and analysis in the behavioral sciences*. Ph.D. thesis, Harvard University.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE 78*(10), 1550–1560.

Zhou, J. and Xu, W. (2015). End-to-end learning of semantic role labeling using recurrent neural networks. *ACL*.