CHAPTER

9 Deep Learning Architectures for Sequence Processing

Time will explain.
Jane Austen, *Persuasion*

Language is an inherently temporal phenomenon. Spoken language is a sequence of acoustic events over time, and we comprehend and produce both spoken and written language as a continuous input stream. 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 emphasize that language is a sequence that unfolds in time.

This temporal nature is reflected in some of the algorithms we use to process language. For example, the Viterbi algorithm applied to HMM part-of-speech tagging, proceeds through the input a word at a time, carrying forward information gleaned along the way. Yet other machine learning approaches, like those we've studied for sentiment analysis or other text classification tasks don't have this temporal nature – they assume simultaneous access to all aspects of their input.

The feedforward networks of Chapter 7 also assumed simultaneous access, although they also had a simple model for time. Recall that we applied feedforward networks to language modeling by having them look only at a fixed-size window of words, and then sliding this window over the input, making independent predictions along the way. Fig. 9.1, reproduced from Chapter 7, shows a neural language model with window size 3 predicting what word follows the input *for all the*. Subsequent words are predicted by sliding the window forward a word at a time.

The simple feedforward sliding-window is promising, but isn't a completely satisfactory solution to temporality. By using embeddings as inputs, it does solve the main problem of the simple n-gram models of Chapter 3 (recall that n-grams were based on words rather than embeddings, making them too literal, unable to generalize across contexts of similar words). But feedforward networks still share another weakness of n-gram approaches: limited context. Anything outside the context window has no impact on the decision being made. Yet many language tasks require access to information that can be arbitrarily distant from the current word. Second, 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, in Fig. 9.1 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.

This chapter introduces two important 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 capture and exploit the temporal nature of language. The recurrent network offers a new way to represent the prior context, allowing the model's deci-

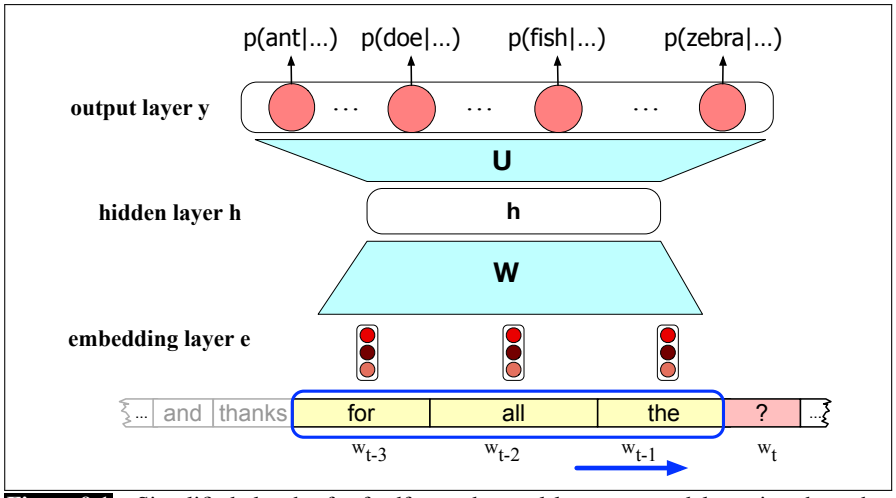

Figure 9.1 Simplified sketch of a feedforward neural language model moving through a text. At each time step t the network converts N context words, each to a d-dimensional embedding, and concatenates the N embeddings together to get the $Nd \times 1$ unit input vector $\mathbf{x}$ for the network. The 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.

sion to depend on information from hundreds of words in the past. The transformer offers new mechanisms (self-attention and positional encodings) that help represent time and help focus on how words relate to each other over long distances. We'll see how to apply both models to the task of language modeling, to sequence modeling tasks like part-of-speech tagging, and to text classification tasks like sentiment analysis.

9.1 Language Models Revisited

In this chapter, we'll begin exploring the RNN and transformer architectures through the lens of probabilistic language models, so let's briefly remind ourselves of the framework for language modeling. 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:

$$P(\textit{fish}|\textit{Thanks for all the})$$

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:

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

Recall that we evaluate language models by examining how well they predict unseen text. Intuitively, good models are those that assign higher probabilities to unseen data (are less surprised when encountering the new words).

perplexity

We instantiate this intuition by using **perplexity** to measure the quality of a language model. Recall from page **??** that the perplexity (PP) of a model θ on an unseen test set is the inverse probability that θ assigns to the test set, normalized by the test set length. For a test set $w_{1:n}$, the perplexity is

$$\begin{aligned} \text{PP}_\theta(w_{1:n}) &= P_\theta(w_{1:n})^{-\frac{1}{n}} \\ &= \sqrt[n]{\frac{1}{P_\theta(w_{1:n})}} \end{aligned} \tag{9.1}$$

To visualize how perplexity can be computed as a function of the probabilities our LM will compute for each new word, we can use the chain rule to expand the computation of probability of the test set:

$$\text{PP}_\theta(w_{1:n}) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P_\theta(w_i|w_{1:i-1})}} \tag{9.2}$$

9.2 Recurrent Neural Networks

A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some 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 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. In this chapter when we use the term RNN we'll be referring to these simpler more constrained networks (although you will often see the term RNN to mean any net with recurrent properties including LSTMs).

Elman Networks

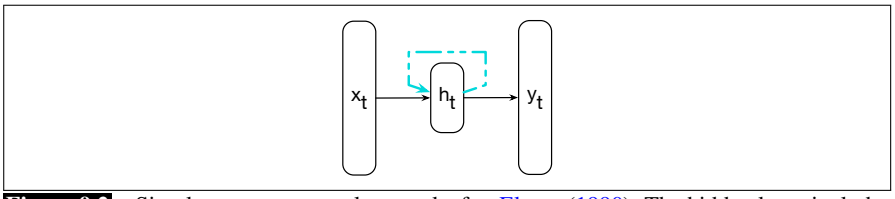

Figure 9.2 Simple recurrent neural network after Elman (1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

Fig. 9.2 illustrates the structure of an RNN. As with ordinary feedforward networks, an input vector representing the current input, $\mathbf{x}_t$, 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, $\mathbf{y}_t$. In a departure from our earlier window-based approach, sequences are processed by presenting one item at a time to the network. We'll use

subscripts to represent time, thus $\mathbf{x}_t$ will mean the input vector $\mathbf{x}$ at time t. 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 can include 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.

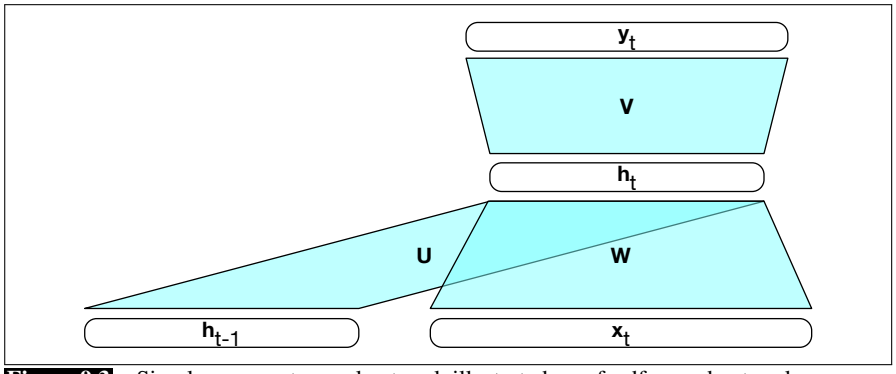

Figure 9.3 Simple recurrent neural network illustrated as a feedforward network.

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 $\mathbf{y}_t$ for an input $\mathbf{x}_t$, we need the activation value for the hidden layer $\mathbf{h}_t$. To calculate this, we multiply the input $\mathbf{x}_t$ with the weight matrix $\mathbf{W}$, and the hidden layer from the previous time step $\mathbf{h}_{t-1}$ with the weight matrix $\mathbf{U}$. 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, $\mathbf{h}_t$. Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \tag{9.3}$$

$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t) \tag{9.4}$$

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: $W \in \mathbb{R}^{d_h \times d_{in}}$, $\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$, and $\mathbf{V} \in \mathbb{R}^{d_{out} \times d_h}$.

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.

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{9.5}$$

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($\mathbf{x}$, *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

Figure 9.4 Forward inference in a simple recurrent network. The matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ are shared across time, while new values for $\mathbf{h}$ and $\mathbf{y}$ 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: $\mathbf{W}$, the weights from the input layer to the hidden layer, $\mathbf{U}$, the weights from the previous hidden layer to the current hidden layer, and finally $\mathbf{V}$, 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 $\mathbf{h}_t$, 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 $\mathbf{h}_t$, $\mathbf{y}_t$, 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

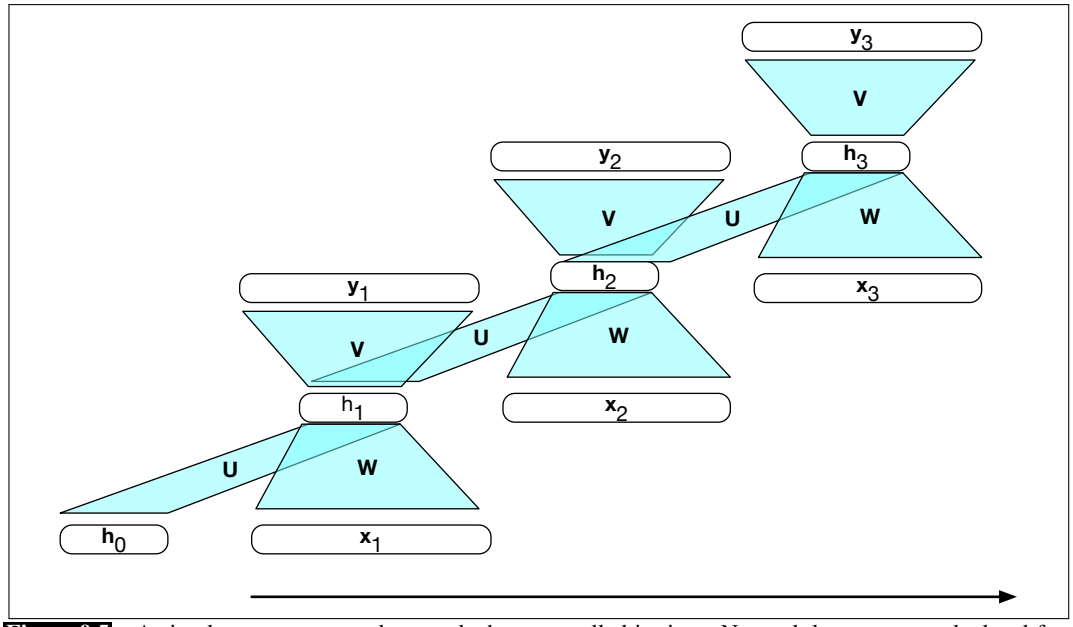

Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared in common across all time steps.

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

9.3 RNNs as Language Models

RNN language models (Mikolov et al., 2010) 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.

Forward inference in a recurrent language model proceeds exactly as described in Section 9.2.1. The input sequence $\mathbf{X} = [\mathbf{x}_1; ...; \mathbf{x}_t; ...; \mathbf{x}_N]$ consists of a series of word embeddings each represented as a one-hot vector of size $|V| \times 1$, and the output

prediction, $\mathbf{y}$, is a vector representing a probability distribution over the vocabulary. At each step, the model uses the word embedding matrix $\mathbf{E}$ 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:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{9.6}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{9.7}$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{9.8}$$

The vector resulting from $\mathbf{Vh}$ can be thought of as a set of scores over the vocabulary given the evidence provided in $\mathbf{h}$. Passing these scores through the softmax normalizes the scores into a probability distribution. The probability that a particular word i in the vocabulary is the next word is represented by $\mathbf{y}_t[i]$, the ith component of $\mathbf{y}_t$:

$$P(w_{t+1} = i | w_1, \ldots, w_t) = \mathbf{y}_t[i] \tag{9.9}$$

The probability of an entire sequence is just the product of the probabilities of each item in the sequence, where we'll use $\mathbf{y}_i[w_i]$ to mean the probability of the true word w_i at time step i.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \tag{9.10}$$

$$= \prod_{i=1}^{n} \mathbf{y}_i[w_i] \tag{9.11}$$

To train an RNN as a language model, we use a corpus of text as training material, having the model predict the next word at each time step t. 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.

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{9.12}$$

In the case of language modeling, the correct distribution $\mathbf{y}_t$ 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. So at time t the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{9.13}$$

Thus at each word position t of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token w_{t+1}. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token w_{t+2}. This idea that we always give the model the correct history sequence to

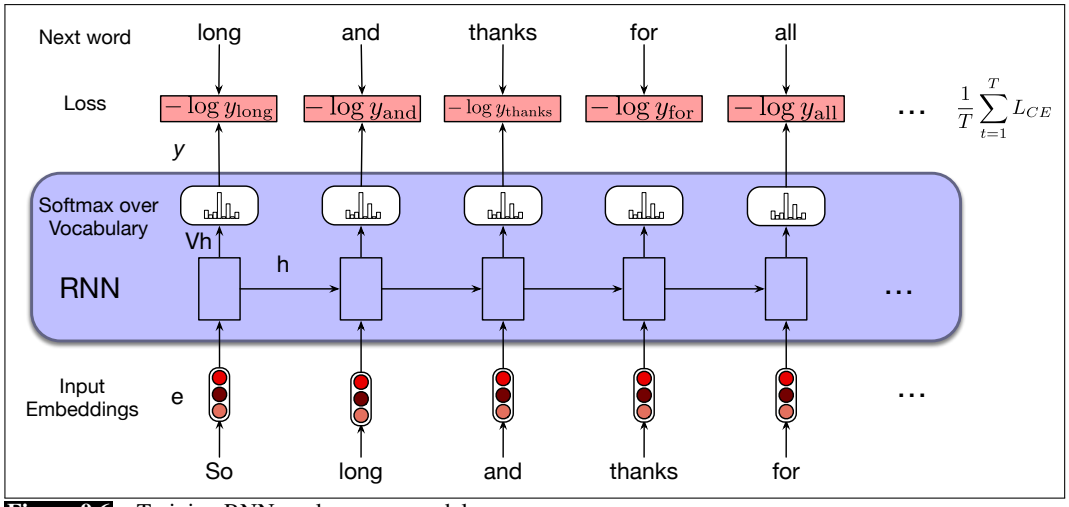

Figure 9.6 Training RNNs as language models.

predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.

teacher forcing

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 $\mathbf{E}$ and the final layer matrix $\mathbf{V}$, which feeds the output softmax, are quite similar. The columns of $\mathbf{E}$ 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 shape of the embedding matrix $\mathbf{E}$ is $d_h \times |V|$.

The final layer matrix $\mathbf{V}$ 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 $\mathbf{Vh}$. This results in a 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 simply uses a single set of embeddings at the input and softmax layers. That is, we dispense with V and use E in both the start and end of the computation.

Weight tying

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{9.14}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{9.15}$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{E}^{intercal}\mathbf{h}_t) \tag{9.16}$$

In addition to providing improved model perplexity, this approach significantly reduces the number of parameters required for the model.

9.4 RNNs for other NLP tasks

Now that we've seen the basic RNN architecture, let's consider how to apply it to three types of NLP tasks: *sequence classification* tasks like sentiment analysis and

topic classification, *sequence labeling* tasks like part-of-speech tagging, and *text generation* tasks. And we'll see in Chapter 10 how to use them for encoder-decoder approaches to summarization, machine translation, and question answering.

9.4.1 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 from 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. 9.7.

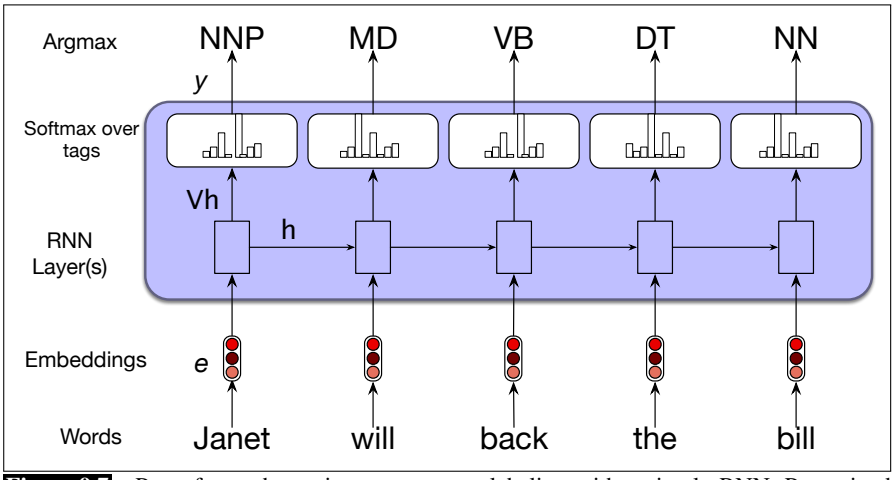

Figure 9.7 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

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 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.

9.4.2 RNNs for Sequence Classification

Another use of RNNs is to classify entire sequences rather than the tokens within them. We've already encountered sentiment analysis in Chapter 4, in which we classify a text as positive or negative. 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.

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. We can then take the hidden layer for the last token of the text, $\mathbf{h}_n$, to constitute a compressed repre-

sentation of the entire sequence. We can pass this representation $\mathbf{h}_n$ to a feedforward network that chooses a class via a softmax over the possible classes. Fig. 9.8 illustrates this approach.

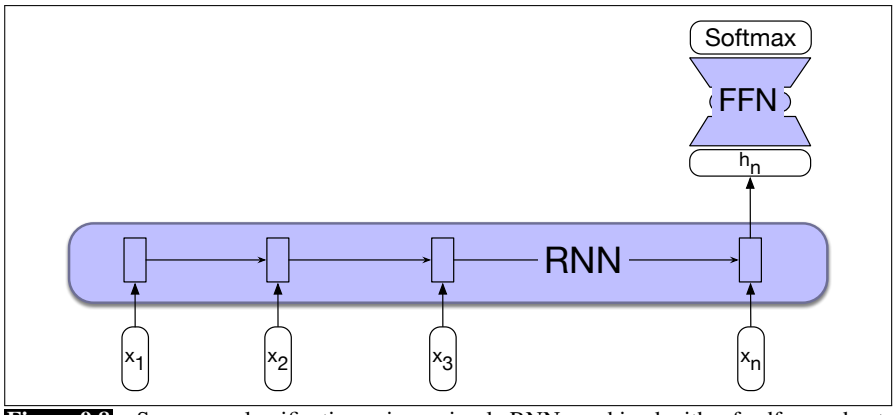

Figure 9.8 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

Note that in this approach there don't need 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. 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 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 as described earlier in Section 9.2.2. 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

pooling

Another option, instead of using just the last token h_n to represent the whole sequence, is to use some sort of **pooling** function of all the hidden states h_i for each word i in the sequence. For example, we can create a representation that pools all the n hidden states by taking their element-wise mean:

$$\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i \tag{9.17}$$

Or we can take the element-wise max; the element-wise max of a set of n vectors is a new vector whose kth element is the max of the kth elements of all the n vectors.

9.4.3 Generation with RNN-Based Language Models

RNN-based language models can also be used to generate text. Text generation is of enormous practical importance, part of tasks like question answering, machine translation, text summarization, and conversational dialogue; any ask where a system needs to produce text, conditioned on some other text.

Recall back in Chapter 3 we saw how to generate text from an n-gram language model by adapting a technique suggested contemporaneously by Claude Shannon (Shannon, 1951) and the psychologists George Miller and Selfridge (Miller and Selfridge, 1950). We first randomly sample a word to begin a sequence based on its

suitability as the start of a sequence. We then continue to sample words *conditioned on our previous choices* until we reach a pre-determined length, or an end of sequence token is generated.

autoregressive generation

Today, 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**. The procedure is basically the same as that described on **??**, in a neural context:

- 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.

Technically an **autoregressive** model is a model that predicts a value at time t based on a linear function of the previous values at times $t-1$, $t-2$, and so on. Although language models are not linear (since they have many layers of non-linearities), we loosely refer to this generation technique as **autoregressive generation** since the word generated at each time step is conditioned on the word selected by the network from the previous step. Fig. 9.9 illustrates this approach. In this figure, the details of the RNN's hidden layers and recurrent connections are hidden within the blue block.

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; for translation the context is the sentence in the source language; for summarization it's the long text we want to summarize. We'll discuss the application of contextual generation to the problem of summarization in Section 9.9 in the context of transformer-based language models, and then again in Chapter 10 when we introduce encoder-decoder models.

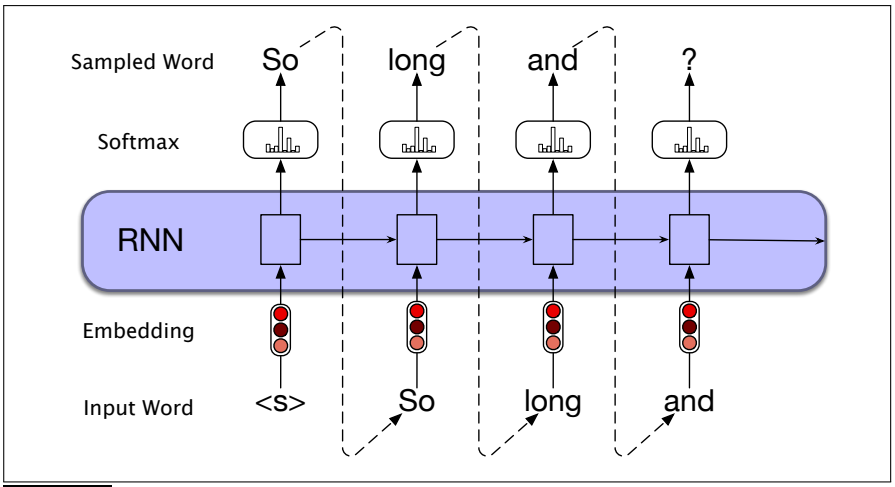

Figure 9.9 Autoregressive generation with an RNN-based neural language model.

9.5 Stacked and Bidirectional RNN architectures

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.

9.5.1 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

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.

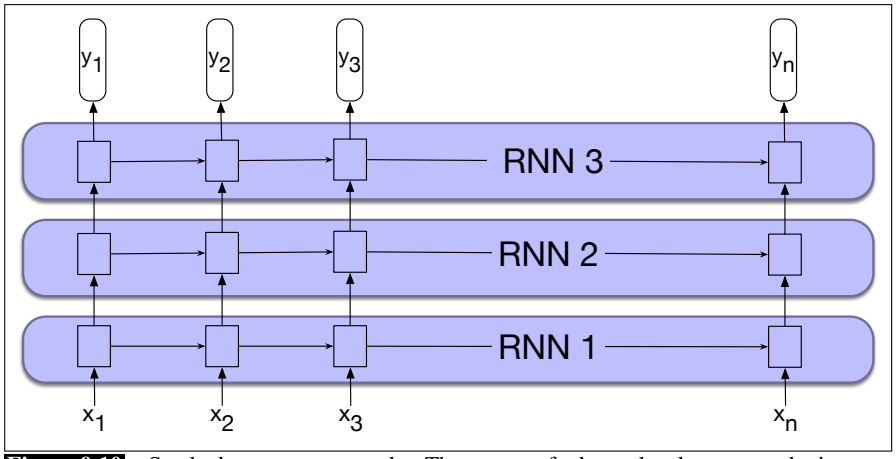

Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

Stacked RNNs generally outperform single-layer networks. One reason for this success seems to be that the network induces 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.

9.5.2 Bidirectional RNNs

The RNN uses information from the left (prior) context to make its predictions at time t. But in many applications we have access to the entire input sequence; in those cases we would like to use words from the context to the right of t. One way to do this is to run two separate RNNs, one left-to-right, and one right-to-left, and concatenate their representations.

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.

$$\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t) \tag{9.18}$$

This new notation $\mathbf{h}_t^f$ simply corresponds to the normal hidden state at time t, representing everything the network has gleaned from the sequence so far.

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 represents information about the sequence to the *right* of the current input:

$$\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_t, \ldots \mathbf{x}_n) \tag{9.19}$$

Here, the hidden state $\mathbf{h}_t^b$ represents all the information we have discerned about the sequence from t to the end of the sequence.

bidirectional RNN

A **bidirectional RNN** (Schuster and Paliwal, 1997) combines 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. Here we use either the semicolon ";" or the equivalent symbol $\oplus$ to mean vector concatenation:

$$\begin{aligned} \mathbf{h}_t &= [\mathbf{h}_t^f ; \mathbf{h}_t^b] \\ &= \mathbf{h}_t^f \oplus \mathbf{h}_t^b \end{aligned} \tag{9.20}$$

Fig. 9.11 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.

Bidirectional RNNs have also proven to be quite effective for sequence classification. Recall from Fig. 9.8 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. 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 (for example by concatenation) and use that as input for follow-on processing.

9.6 The LSTM

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. Yet distant information is critical to many language applications. Consider the following example in the context of language modeling.

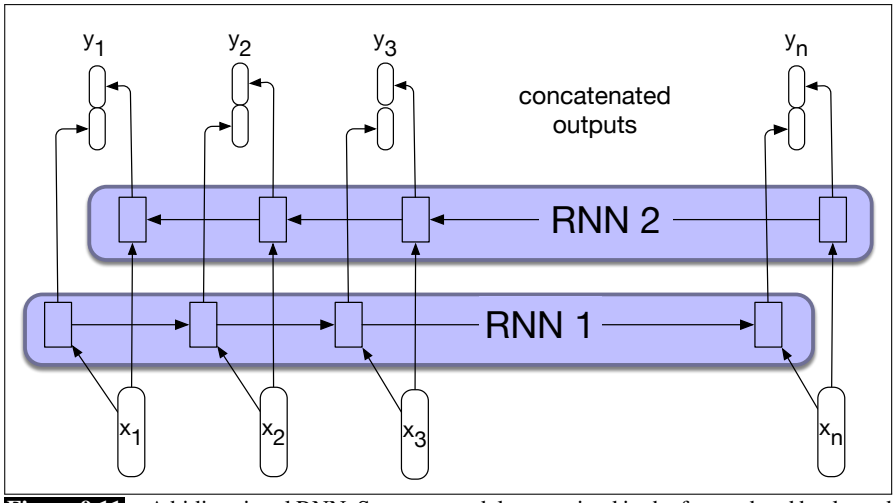

Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

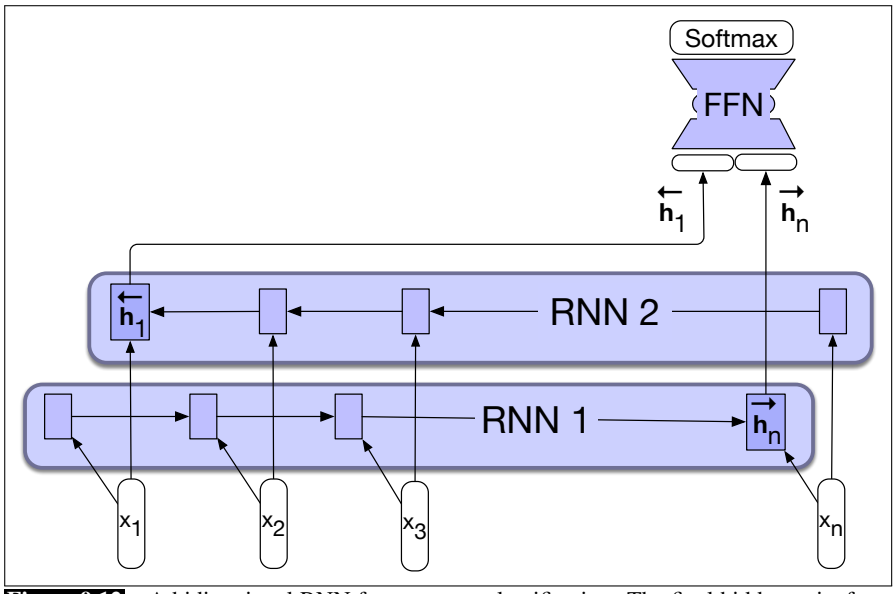

Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

(9.21) 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 hid-

den 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 RNNs 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, a situation called the **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, by enabling the network to learn to forget information that is no longer needed and to remember information required for decisions still to come.

Long short-term memory

The most commonly used such extension to RNNs is the **Long short-term memory** (LSTM) network (Hochreiter and Schmidhuber, 1997). LSTMs 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 feed-forward 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 element-wise by the context vector to remove the information from context that is no longer required. Element-wise multiplication of two vectors (represented by the operator $\odot$, and sometimes called the **Hadamard product**) is the vector of the same dimension as the two input vectors, where each element i is the product of element i in the two input vectors:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{x}_t) \tag{9.22}$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \tag{9.23}$$

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.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t) \tag{9.24}$$

add gate

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

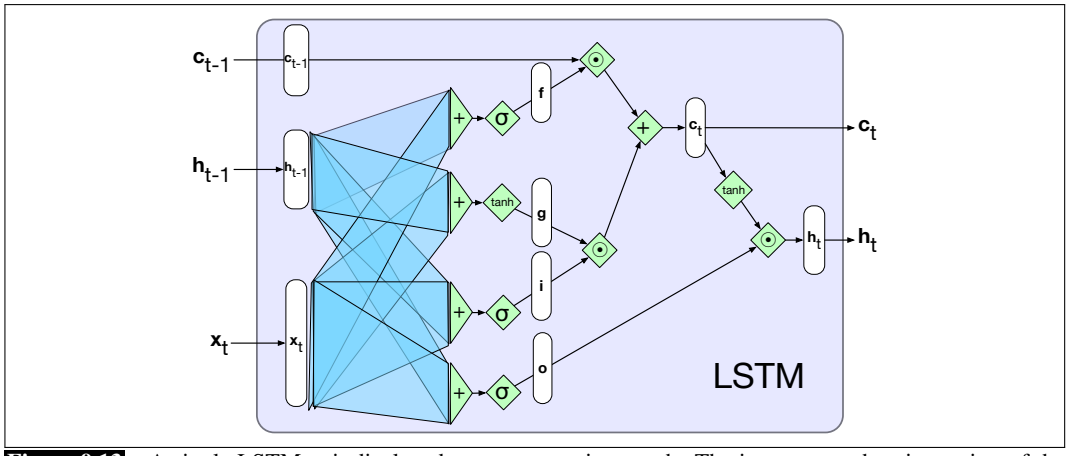

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

current context.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t) \tag{9.25}$$
$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \tag{9.26}$$

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

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \tag{9.27}$$

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).

$$\mathbf{o}_t = \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t) \tag{9.28}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{9.29}$$

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.

9.6.1 Gated Units, Layers and Networks

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. 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.

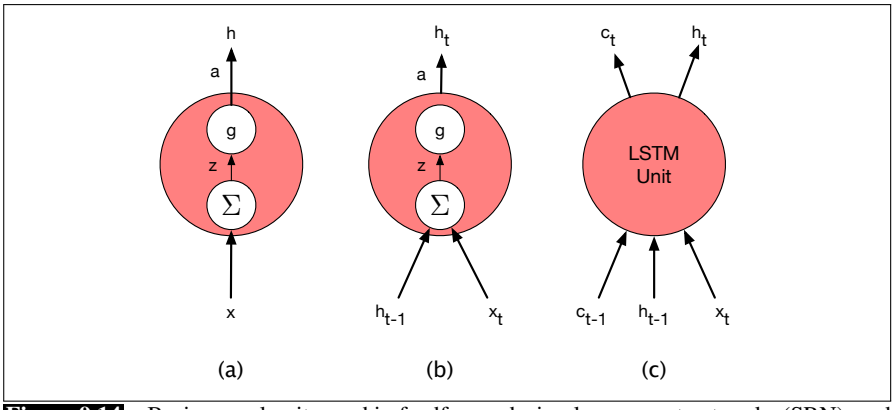

Figure 9.14 Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

The increased complexity of the LSTM units is encapsulated within the unit itself. 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.

This modularity is key to the power and widespread applicability of LSTM units. LSTM units (or other varieties, like GRUs) can be substituted into any of the network architectures described in Section 9.5. 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.

9.7 Self-Attention Networks: Transformers

While the addition of gates allows LSTMs to handle more distant information than RNNs, they don't completely solve the underlying problem: passing information through an extended series of recurrent connections leads to information loss and difficulties in training. Moreover, the inherently sequential nature of recurrent networks makes it hard to do computation in parallel. 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 $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{y}_1, ..., \mathbf{y}_n)$ of the same length. Transformers are made up of stacks of transformer blocks, which are multilayer networks made by combining simple linear layers, feedforward networks, and **self-attention** layers, the 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. We'll start by describing how self-attention works and then return to how it fits into larger transformer blocks.

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 $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to output sequences of the same length $(\mathbf{y}_1, ..., \mathbf{y}_n)$. When processing each item in the input, the model has 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. 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.

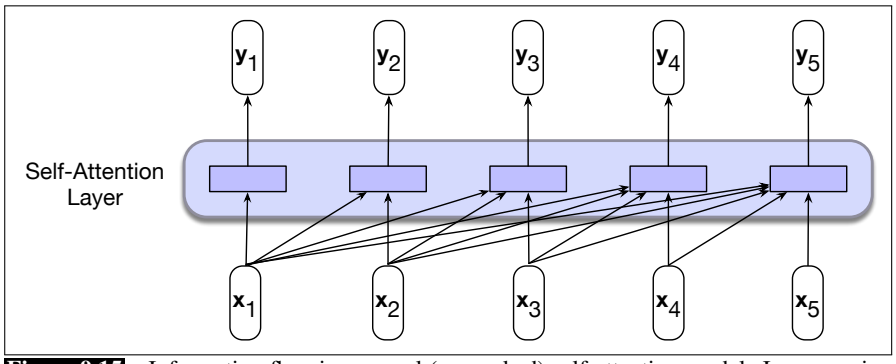

Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be 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 a 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 $\mathbf{y}_3$ is based on a set of comparisons between the input $\mathbf{x}_3$ and its preceding elements $\mathbf{x}_1$ and $\mathbf{x}_2$, and to $\mathbf{x}_3$ itself. The simplest form of comparison between elements in a self-attention layer is a dot product. Let's refer to the result of this comparison as a score (we'll be updating this equation to add attention to the computation of this score):

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{9.30}$$

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: $\mathbf{x}_3 \cdot \mathbf{x}_1$, $\mathbf{x}_3 \cdot \mathbf{x}_2$ and $\mathbf{x}_3 \cdot \mathbf{x}_3$. 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.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \tag{9.31}$$

$$= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \;\; \forall j \leq i \tag{9.32}$$

Given the proportional scores in α, we then generate an output value $\mathbf{y}_i$ by taking the sum of the inputs seen so far, weighted by their respective α value.

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{9.33}$$

The steps embodied in Equations 9.30 through 9.33 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 $\mathbf{y}$ is the result of this straightforward computation over the inputs.

This kind of simple attention can be useful, and indeed we'll see in Chapter 10 how to use this simple idea of attention for LSTM-based encoder-decoder models for machine translation.

But transformers allow us to create a more sophisticated way of representing how words can contribute to the representation of longer inputs. Consider the three different roles that each input embedding plays during the course of the attention process.

query
- 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**.

key
- In its role as *a preceding input* being compared to the current focus of attention. We'll refer to this role as a **key**.

value
- And finally, as a **value** used to compute the output for the current focus of attention.

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

$$\mathbf{q}_i = \mathbf{W^Q}\mathbf{x}_i; \;\; \mathbf{k}_i = \mathbf{W^K}\mathbf{x}_i; \;\; \mathbf{v}_i = \mathbf{W^V}\mathbf{x}_i \tag{9.34}$$

The inputs $\mathbf{x}$ and outputs $\mathbf{y}$ of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality $1 \times d$. For now let's assume the dimensionalities of the transform matrices are $\mathbf{W}^Q \in \mathbb{R}^{d\times d}$, $\mathbf{W}^K \in \mathbb{R}^{d\times d}$, and $\mathbf{W}^V \in \mathbb{R}^{d\times d}$. Later we'll need separate dimensions for these matrices when we introduce multi-headed attention, so let's just make a note that we'll have a dimension d_k for the key and query vectors, and a dimension d_v for the value vectors, both of which for now we'll set to d. In the original transformer work (Vaswani et al., 2017), d was 1024.

Given these projections, the score between a current focus of attention, $\mathbf{x}_i$ and an element in the preceding context, $\mathbf{x}_j$ consists of a dot product between its query vector $\mathbf{q}_i$ and the preceding element's key vectors $\mathbf{k}_j$. This dot product has the right shape since both the query and the key are of dimensionality $1 \times d$. Let's update our previous comparison calculation to reflect this, replacing Eq. 9.30 with Eq. 9.35:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) \;\; = \;\; \mathbf{q}_i \cdot \mathbf{k}_j \tag{9.35}$$

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

$$\mathbf{y}_i \;\; = \;\; \sum_{j\leq i} \alpha_{ij}\mathbf{v}_j \tag{9.36}$$

Fig. 9.16 illustrates this calculation in the case of computing the third output $\mathbf{y}_3$ in a sequence.

The result of a 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

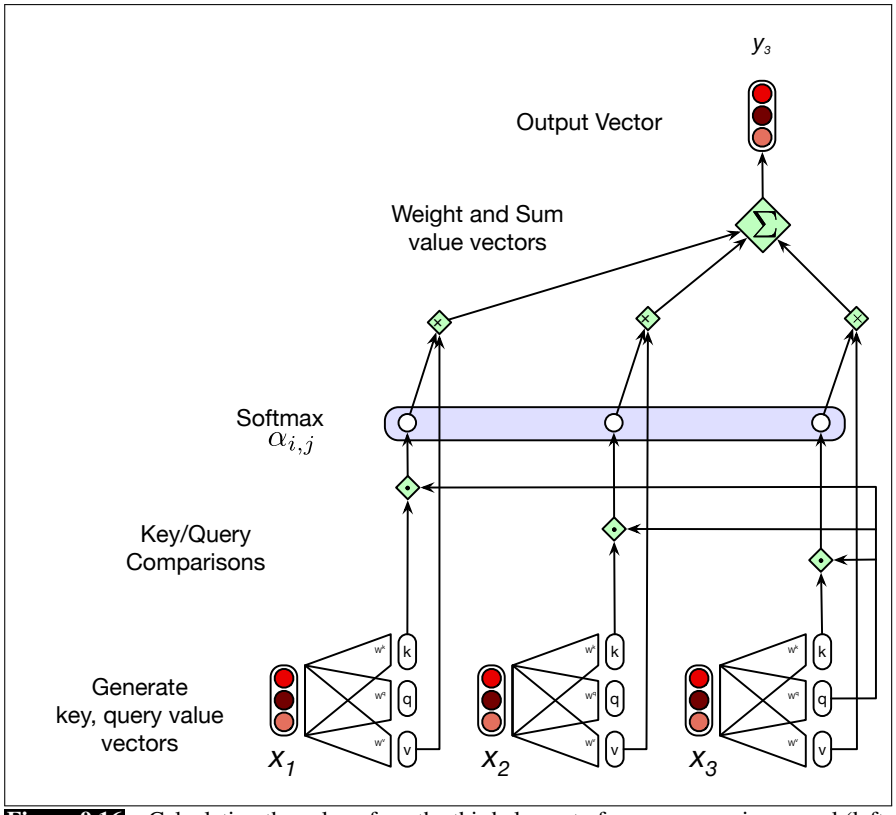

Figure 9.16 Calculating the value of $\mathbf{y}_3$, the third element of a sequence using causal (left-to-right) self-attention.

softmax. A typical approach is to divide the dot product by the square root of the dimensionality of the query and key vectors (d_k), leading us to update our scoring function one more time, replacing Eq. 9.30 and Eq. 9.35 with Eq. 9.37:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{9.37}$$

This description of the self-attention process has been from the perspective of computing a single output at a single time step i. However, since each output, $\mathbf{y}_i$, is computed independently this entire process can be parallelized by taking advantage of efficient matrix multiplication routines by packing the input embeddings of the N tokens of the input sequence into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$. That is, each row of $\mathbf{X}$ is the embedding of one token of the input. We then multiply $\mathbf{X}$ by the key, query, and value matrices (all of dimensionality $d \times d$) to produce matrices $\mathbf{Q} \in \mathbb{R}^{N \times d}$, $\mathbf{K} \in \mathbb{R}^{N \times d}$, and $\mathbf{V} \in \mathbb{R}^{N \times d}$, containing all the key, query, and value vectors:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \quad \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}} \tag{9.38}$$

Given these matrices we can compute all the requisite query-key comparisons simultaneously by multiplying $\mathbf{Q}$ and $\mathbf{K}^\intercal$ in a single matrix multiplication (the product is of shape $N \times N$; Fig. 9.17 shows a visualization). Taking this one step further, we can scale these scores, take the softmax, and then multiply the result by $\mathbf{V}$ resulting in a matrix of shape $N \times d$: a vector embedding representation for each token in the input. We've reduced the entire self-attention step for an entire sequence of N tokens

to the following computation:

$$\textit{SelfAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \text{softmax}\left(\frac{\mathbf{QK}^{\intercal}}{\sqrt{d_k}}\right)\mathbf{V} \tag{9.39}$$

Unfortunately, this process goes a bit too far since the calculation of the comparisons in $\mathbf{QK}^{\intercal}$ 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 matrix are zeroed out (set to $-\infty$), thus eliminating any knowledge of words that follow in the sequence. Fig. 9.17 depicts the $\mathbf{QK}^{\intercal}$ matrix. (we'll see in Chapter 11 how to make use of words in the future for tasks that need it).

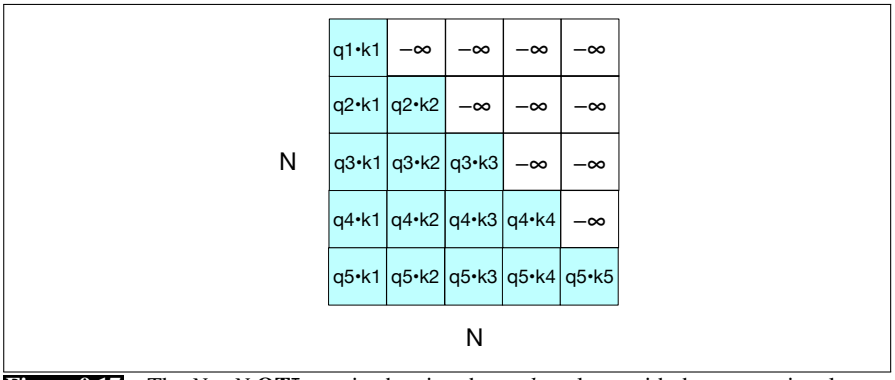

Figure 9.17 The $N \times N$ $\mathbf{QT}^{\intercal}$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

Fig. 9.17 also 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.

9.7.1 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.

Fig. 9.18 illustrates a standard transformer block consisting of a single attention layer followed by a fully-connected feedforward layer with residual connections and layer normalizations following each. We've already seen feedforward layers in Chapter 7, but what are residual connections and layer norm? In deep networks, 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 (He et al., 2016). Residual connections in transformers are implemented

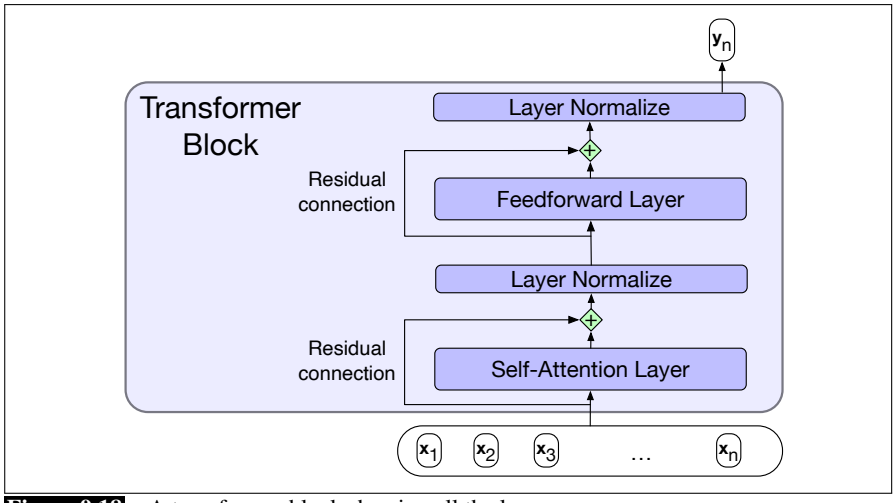

Figure 9.18 A transformer block showing all the layers.

by added a layer's input vector to its output vector before passing it forward. In the transformer block shown in Fig. 9.18, residual connections are used with both the attention and feedforward sublayers. These summed vectors are then normalized using layer normalization (Ba et al., 2016). If we think of a layer as one long vector of units, the resulting function computed in a transformer block can be expressed as:

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{SelfAttn}(\mathbf{x})) \tag{9.40}$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFNN}(\mathbf{z})) \tag{9.41}$$

layer norm

Layer normalization (or **layer norm**) is one of many forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training. Layer norm is a variation of the standard score, or z-score, from statistics applied to a single hidden layer. The first step in layer normalization is to calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized. Given a hidden layer with dimensionality d_h, these values are calculated as follows.

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{9.42}$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i-\mu)^2} \tag{9.43}$$

Given these values, the vector components are normalized by subtracting the mean from each and dividing by the standard deviation. The result of this computation is a new vector with zero mean and a standard deviation of one.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x}-\mu)}{\sigma} \tag{9.44}$$

Finally, in the standard implementation of layer normalization, two learnable parameters, γ and β, representing gain and offset values, are introduced.

$$\mathit{LayerNorm} = \gamma\hat{\mathbf{x}} + \beta \tag{9.45}$$

9.7.2 Multihead Attention

multihead self-attention layers

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.

To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query and value matrices: $\mathbf{W}_i^K$, $\mathbf{W}_i^Q$ and $\mathbf{W}_i^V$. These are used to project the inputs into separate key, value, and query embeddings separately for each head, with the rest of the self-attention computation remaining unchanged. In multi-head attention, instead of using the model dimension d that's used for the input and output from the model, the key and query embeddings have dimensionality d_k, and the value embeddings are dimensionality d_v (in the original transformer paper $d_k = d_v = 64$). Thus for each head i, we have weight layers $\mathbf{W}_i^Q \in \mathbb{R}^{d\times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d\times d_k}$, and $\mathbf{W}_i^V \in \mathbb{R}^{d\times d_v}$, and these get multiplied by the inputs packed into $\mathbf{X}$ to produce $\mathbf{Q} \in \mathbb{R}^{N\times d_k}$, $\mathbf{K} \in \mathbb{R}^{N\times d_k}$, and $\mathbf{V} \in \mathbb{R}^{N\times d_v}$. The output of each of the h heads is of shape $N \times d_v$, and so the output of the multi-head layer with h heads consists of h vectors of shape $N \times d_v$. To make use of these vectors in further processing, they are combined and then reduced down to the original input dimension d. This is accomplished by concatenating the outputs from each head and then using yet another linear projection, $\mathbf{W}^O \in \mathbb{R}^{hd_v\times d}$, to reduce it to the original output dimension for each token, or a total $N \times d$ output.

$$\mathit{MultiHeadAttn}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^O \tag{9.46}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q \; ; \; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \tag{9.47}$$

$$\mathbf{head}_i = \mathit{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \tag{9.48}$$

Fig. 9.19 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. 9.18, the rest of the transformer block with its feedforward layer, residual connections, and layer norms remains the same.

9.7.3 Modeling word order: positional embeddings

How does a transformer model the position of each token in the input sequence? With RNNs, information about the order of the inputs was built into the structure of the model. Unfortunately, the same isn't true for transformers; the models as we've described them so far don't have any notion of the relative, or absolute, positions of the tokens in the input. This can be seen from the fact that if you scramble the order of the inputs in the attention computation in Fig. 9.16 you get exactly the same answer.

positional embeddings

One simple solution is to modify the input embeddings by combining them with **positional embeddings** specific to each position in an input sequence.

Where do we get these positional embeddings? The simplest method 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

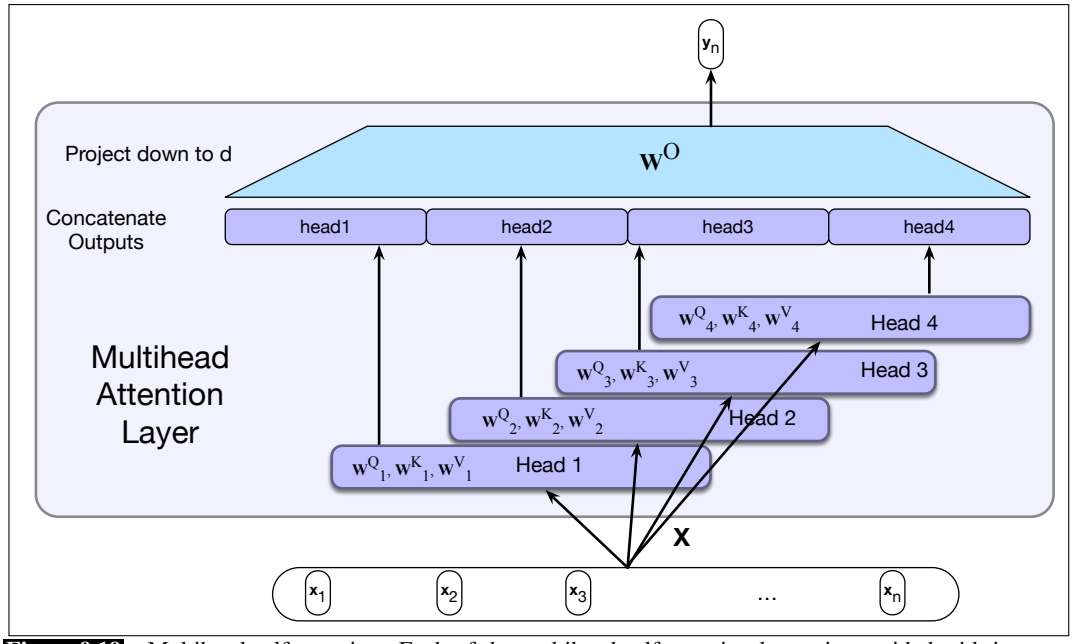

Figure 9.19 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can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

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. Fig. 9.20 shows the idea.

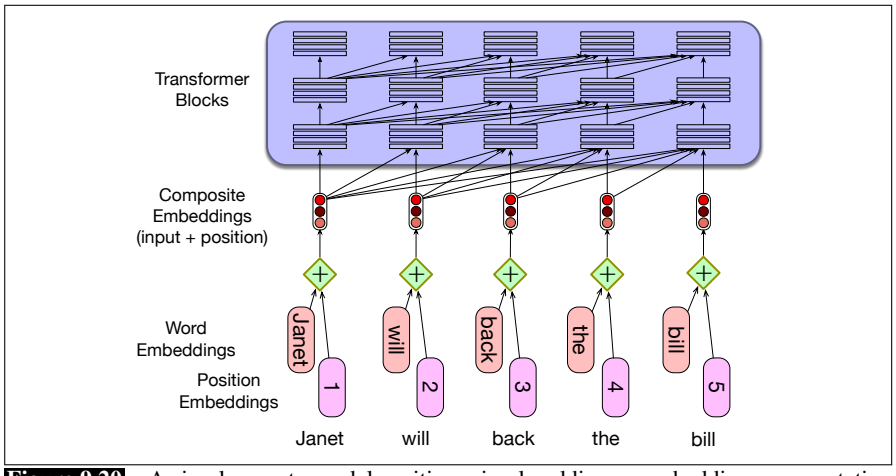

Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

A potential problem with the simple absolute position embedding 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 generalize well during testing. An alternative approach to

positional embeddings is to choose a static function that maps 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. Developing better position representations is an ongoing research topic.

9.8 Transformers as 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. 9.21 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.

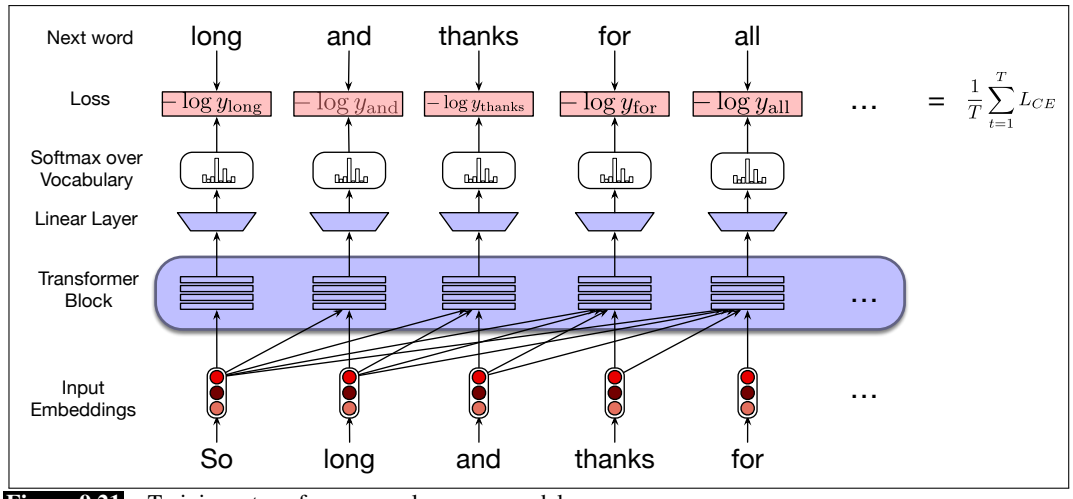

Figure 9.21 Training a transformer as a language model.

Note the key difference between this figure and the earlier RNN-based version 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.

9.9 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. 9.22 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.

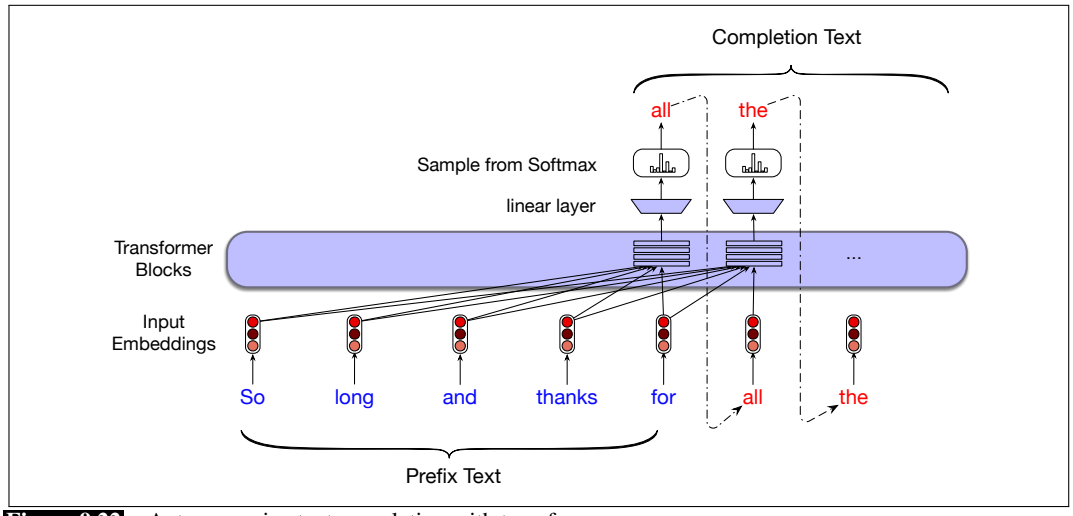

Figure 9.22 Autoregressive text completion with transformers.

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, we start with a corpus consisting of full-length articles accompanied by their corresponding summaries. Fig. 9.23 shows an example of this kind of data from a widely used summarization corpus consisting of CNN and Daily Mirror news articles.

A simple but 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. 9.24. 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.

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

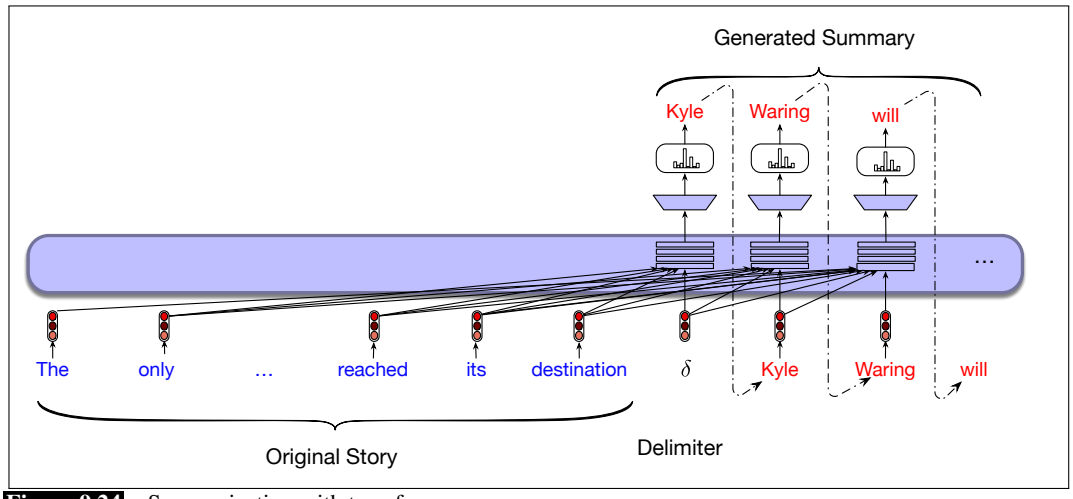

Figure 9.24 Summarization with transformers.

9.9.1 Applying Transformers to other NLP tasks

Transformers can also be used for sequence labeling tasks (like part-of-speech tagging or named entity tagging) and sequence classification tasks (like sentiment classification), as we'll see in detail in Chapter 11. Just to give a preview, however, we don't directly train a raw transformer on these tasks. Instead, we use a technique called **pretraining**, in which we first train a transformer language model on a large corpus of text, in a normal self-supervised way, and only afterwards add a linear or feedforward layer on top that we **finetune** on a smaller dataset hand-labeled with part-of-speech or sentiment labels. Pretraining on large amounts of data via the

pretraining

finetune

self-supervised language model objective turns out to be a very useful way of incorporating rich information about language, and the resulting representations make it much easier to learn from the generally smaller supervised datasets for tagging or sentiment.

9.10 Summary

This chapter has introduced the concepts of recurrent neural networks and transformers 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 one element at a time, with the output of each neural unit at time t based both on the current input at t and the hidden layer from time $t-1$.
- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as **backpropagation through time** (BPTT).
- Simple recurrent networks fail on long inputs because of problems like **vanishing gradients**; instead modern systems use more complex gated architectures such as **LSTMs** that explicitly decide what to remember and forget in their hidden and context layers.
- Transformers are non-recurrent networks based on **self-attention**. A self-attention layer maps input sequences to output sequences of the same length, using attention heads that model how the surrounding words are relevant for the processing of the current word.
- A transformer block consists of a single attention layer followed by a feedforward layer with residual connections and layer normalizations following each. Transformer blocks can be stacked to make deeper and more powerful networks.
- Common language-based applications for RNNs and transformers include:
  - Probabilistic language modeling: assigning 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 like part-of-speech tagging, where each element of a sequence is assigned a label.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.

Bibliographical and Historical Notes

Influential investigations of RNNs 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, Robinson et al. 1996). 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) and Gers et al. (2000). 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), named entity recognition (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).

The transformer (Vaswani et al., 2017) was developed drawing on two lines of prior research: **self-attention** and **memory networks**. Encoder-decoder attention, the idea of using a soft weighting over the encodings of input words to inform a generative decoder (see Chapter 10) was developed by Graves (2013) in the context of handwriting generation, and Bahdanau et al. (2015) for MT. This idea was extended to self-attention by dropping the need for separate encoding and decoding sequences and instead seeing attention a way of weighting the tokens in collecting information passed from lower layers to higher layers (Ling et al., 2015; Cheng et al., 2016; Liu et al., 2016). Other aspects of the transformer, including the terminology of key, query, and value, came from **memory networks**, a mechanism for adding an external read-write memory to networks, by using an embedding of a query to match keys representing content in an associative memory (Sukhbaatar et al., 2015; Weston et al., 2015; Graves et al., 2014).

Ba, J. L., J. R. Kiros, and G. E. Hinton. 2016. Layer normalization. *NeurIPS workshop*.

Bahdanau, D., K. H. Cho, and Y. Bengio. 2015. Neural machine translation by jointly learning to align and translate. *ICLR 2015*.

Cheng, J., L. Dong, and M. Lapata. 2016. Long short-term memory-networks for machine reading. *EMNLP*.

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

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

Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. 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 J. H. Martin. 2016. CU-NLP at SemEval-2016 task 8: AMR parsing using LSTM-based recurrent neural networks. *SemEval-2016*.

Gers, F. A., J. Schmidhuber, and F. Cummins. 2000. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471.

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

Graves, A. 2013. Generating sequences with recurrent neural networks. ArXiv.

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

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

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

Graves, A., G. Wayne, and I. Danihelka. 2014. Neural Turing machines. ArXiv.

He, K., X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. *CVPR*.

Hermann, K. M., T. Kočiský, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom. 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 J. Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

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

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

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

Liu, Y., C. Sun, L. Lin, and X. Wang. 2016. Learning natural language inference using bidirectional LSTM model and inner-attention. ArXiv.

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

Mathis, D. A. and M. C. Mozer. 1995. On the computational utility of consciousness. *Advances in Neural Information Processing Systems VII*. MIT Press.

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

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

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

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

Nallapati, R., B. Zhou, C. dos Santos, Ç. Gulçehre, and B. Xiang. 2016. Abstractive text summarization using sequence-to-sequence RNNs and beyond. *CoNLL*.

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

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

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

Robinson, T., M. Hochberg, and S. Renals. 1996. The use of recurrent neural networks in continuous speech recognition. In C.-H. Lee, F. K. Soong, and K. K. Paliwal, editors, *Automatic speech and speaker recognition*, pages 233–258. Springer.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 2, pages 318–362. MIT Press.

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

Schuster, M. and K. K. Paliwal. 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.

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

Sukhbaatar, S., A. Szlam, J. Weston, and R. Fergus. 2015. End-to-end memory networks. *NeurIPS*.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 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.

Weston, J., S. Chopra, and A. Bordes. 2015. Memory networks. *ICLR 2015*.

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