# CHAPTER 9 Sequence Processing with Recurrent Networks

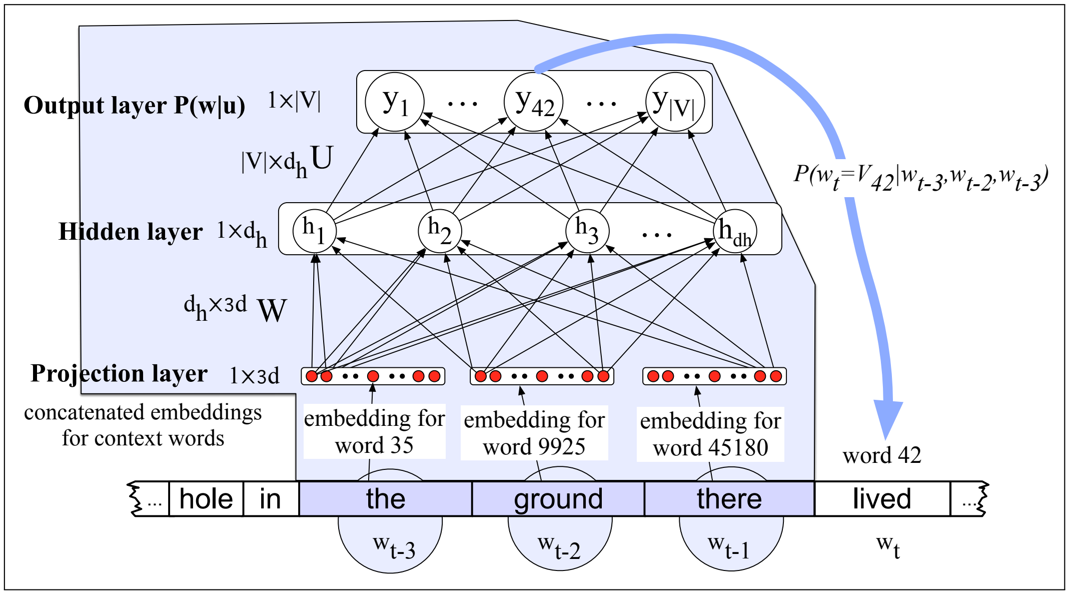
*Time will explain.*
Jane Austin, *Persuasion*

Language is an inherently temporal phenomenon. When we comprehend and produce spoken language, we are processing continuous input streams of indefinite length. And even when dealing with written text we normally process it sequentially, even though we in principle have arbitrary access to all the elements at once. 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 also reflected in the algorithms we use to process language. When applied to the problem of part-of-speech tagging, the Viterbi algorithm works its way incrementally through its input a word at a time, taking into account information gleaned along the way. The syntactic parsing algorithms we cover in Chapters 11, 12, and 13 operate in a similar fashion.

In contrast, the machine learning approaches we've studied for sentiment analysis and other classification tasks do not have this temporal nature. These approaches have simultaneous access to all aspects of their input. This is certainly true of feedforward neural networks, including their application to neural language models. Such networks employ fixed-size input vectors with associated weights to capture all relevant aspects of an example at once. This makes it difficult to deal with sequences of varying length, and they fail to capture important temporal aspects of language.

We saw one work-around for these problems with the case of neural language models. These models operate by accepting fixed-sized windows of tokens as input; sequences longer than the window size are processed by sliding windows over the input making predictions as they go, with the end result being a sequence of predictions spanning the input. Importantly, the decision made for one window has no impact on later decisions. Fig. 9.1, reproduced here from Chapter 7, illustrates this approach with a window of size 3. Here, we're predicting which word will come next given the window *the ground there*. Subsequent words are predicted by sliding the window forward one word at a time.

The sliding window approach is problematic for a number of reasons. First, it shares the primary weakness of Markov 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 language 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



Output layer P(w|u)
1×|V|
$y_1$ … $y_{42}$ … $y_{|V|}$
|V|×$d_h$ U
$P(w_t=V_{42}|w_{t-3},w_{t-2},w_{t-3})$
Hidden layer 1×$d_h$
$h_1$ $h_2$ $h_3$ … $h_{dh}$
$d_h$×3d W
Projection layer 1×3d
concatenated embeddings for context words
embedding for word 35
embedding for word 9925
embedding for word 45180
word 42
… hole in the ground there lived …
$w_{t-3}$ $w_{t-2}$ $w_{t-1}$ $w_t$

**Figure 9.1** A simplified view of a feedforward neural language model moving through a text. At each timestep $t$ the network takes the 3 context words, converts each to a $d$-dimensional embeddings, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer $x$ for the network.

"Dhaka is where I grew up, but now live in Melbourne. I speak fluent ________. "

We need information from the distant past to accurately predict the correct word.
Ref: MIT Lec on Recurrent NN in introtodeeplearning.com

Problem in Bag of Word:
"The food was good, not bad at all."
"The food was bad, not good at all."

Both give same result.!!!

example, in Fig. 9.1 the noun phrase *the ground* appears in two separate windows: once, as shown, in the first and second positions in the window, and in the preceding step where it appears in the second and third positions, thus forcing the network to learn two separate patterns for what should be a constituent.

**recurrent neural networks**

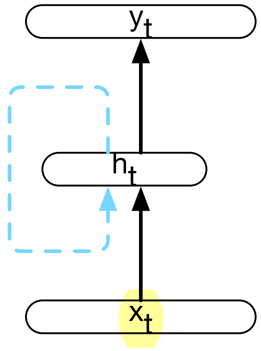
The subject of this chapter is **recurrent neural networks**, a class of networks designed to address these challenges by dealing directly with the temporal aspect of language, allowing us to handle variable length inputs without the use of arbitrary fixed-sized windows, and providing the means to capture and exploit the temporal nature of language.

## 9.1 Simple Recurrent Neural Networks

**Elman 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 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 to be discussed later in this chapter and again in Chapter 10 and Chapter 23. Going forward, when we use the term RNN we'll be referring to these simpler more constrained networks.

Fig. 9.2 illustrates the structure of a simple RNN. As with ordinary feedforward networks, an input vector representing the current input element, $x_t$, is multiplied by a weight matrix and then passed through an activation function to compute an



RNN:
ht = g(Uht-1 + WXt)
z=Vht
yt = f(Vht)
g = ReLu or tanh function
f = softmax

output weight vector V in this fig was U in FFN;
In RNN, U is output weight vector of previous
time step coming to current time step.

FFN:
h = sigma(WXt + b)
z = Uh
y = g(z)
g = softmax or ReLu or tanh function

**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 timestep.

activation value for a layer of hidden of units. This hidden layer is, in turn, used to calculate a corresponding output, $y_t$. In a departure from our earlier window-based approach, sequences are processed by presenting one element 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 activation value of the hidden layer *from the preceding point in time*.

The hidden layer from the previous timestep 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 architecture does not impose a fixed-length limit on this prior context; the context embodied in the previous hidden layer includes information extending back to the beginning of the sequence.

Adding this temporal dimension may make RNNs appear to be more exotic 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. 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 timestep to the current hidden layer. These weights determine how the network should make use of past context in calculating the output for the current input. As with the other weights in the network, these connections are trained via backpropagation.

### 9.1.1 Inference in Simple 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

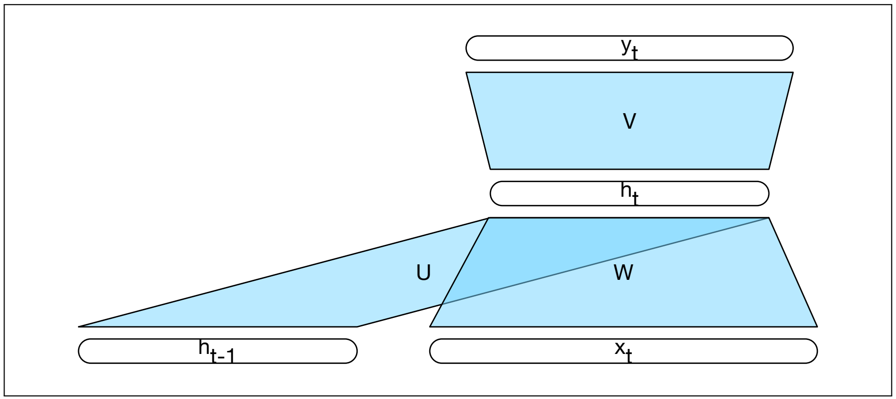


**Figure 9.3** Simple recurrent neural network illustrated as a feedforward network.

output weight vector V in this fig was U in FFN;

In RNN, U is output weight vector of previous time step coming to current time step.

RNN:
ht = g(Uht-1 + WXt)
z=Vht
yt = f(Vht)
g = ReLu or tanh function
f = softmax function

FFN:
h = sigma(WXt + b)
z = Uh
y = g(z)
g = softmax or ReLu or tanh function

output vector.

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

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

$$y_t = softmax(Vh_t)$$

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 timestep.

### 9.1.2 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

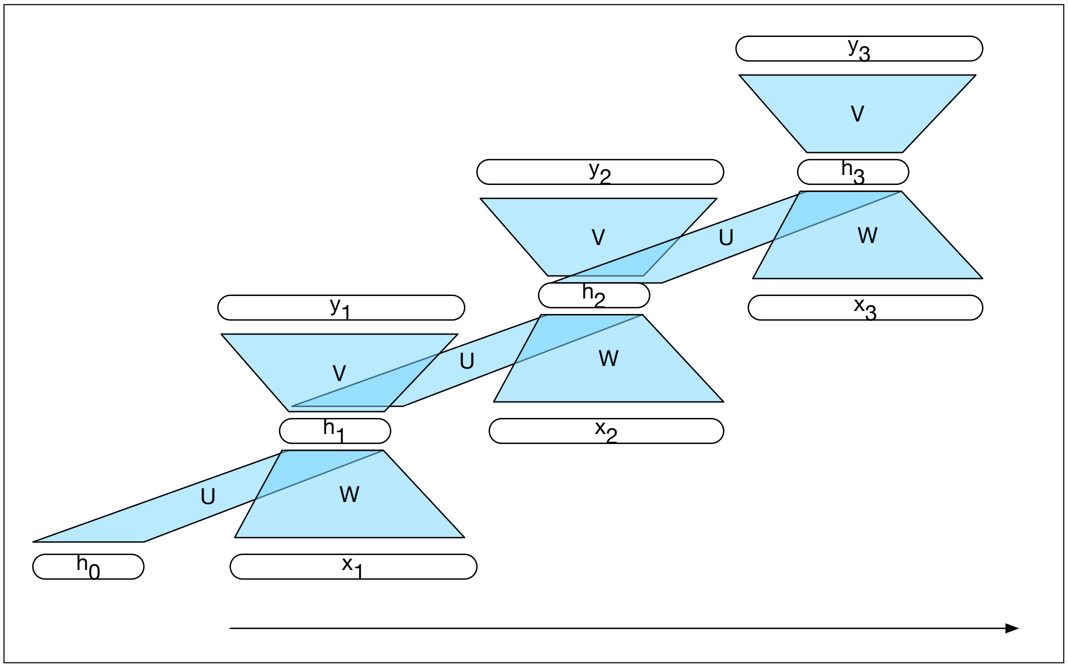


**Figure 9.5** A simple recurrent neural network shown unrolled in time. Network layers are copied for each timestep, while the weights $U$, $V$ and $W$ are shared in common across all timesteps.

$$\begin{aligned} z^{[1]} &= Wx \\ a^{[1]} &= g(z^{[1]}) \\ z^{[2]} &= Ua^{[1]} \\ a^{[2]} &= g(z^{[2]}) \\ y &= a^{[2]} \end{aligned}$$

this are equations for multilayer feed-forward-network
inputs in each layer contains a bias (b); which is omitted here for clarity.
In FFN is was: z = Wx +b;

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.

Before going on, let's first review some of the notation that we introduced in Chapter 7. Assuming a network with an input layer $x$ and a non-linear activation function $g$, $a^{[i]}$ refers to the activation value from a layer $i$, which is the result of applying $g$ to $z^{[i]}$, the weighted sum of the inputs to that layer.

Fig. 9.5 illustrates 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*.

Consider the situation where we are examining an input/output pair at time 2 as shown in Fig. 9.6. What do we need to compute the gradients required to update the weights $U$, $V$, and $W$ here? Let's start by reviewing how we compute the gradients required to update $V$ since this computation is unchanged from feedforward networks. To review from Chapter 7, we need to compute the derivative of the loss function $L$ with respect to the weights $V$. However, since the loss is not expressed directly in terms of the weights, we apply the chain rule to get there indirectly.

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

The first term on the right is the derivative of the loss function with respect to the network output, $a$. The second term is the derivative of the network output with respect to the intermediate network activation $z$, which is a function of the activation function $g$. The final term in our application of the chain rule is the derivative of the network activation with respect to the weights $V$, which is the activation value of the current hidden layer $h_t$.

It's useful here to use the first two terms to define $\delta$, an error term that represents how much of the scalar loss is attributable to each of the units in the output layer.

$$\delta_{out} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} \tag{9.1}$$

$$\delta_{out} = L'g'(z) \tag{9.2}$$

Therefore, the final gradient we need to update the weight matrix $V$ is just:

$$\frac{\partial L}{\partial V} = \delta_{out} h_t \tag{9.3}$$

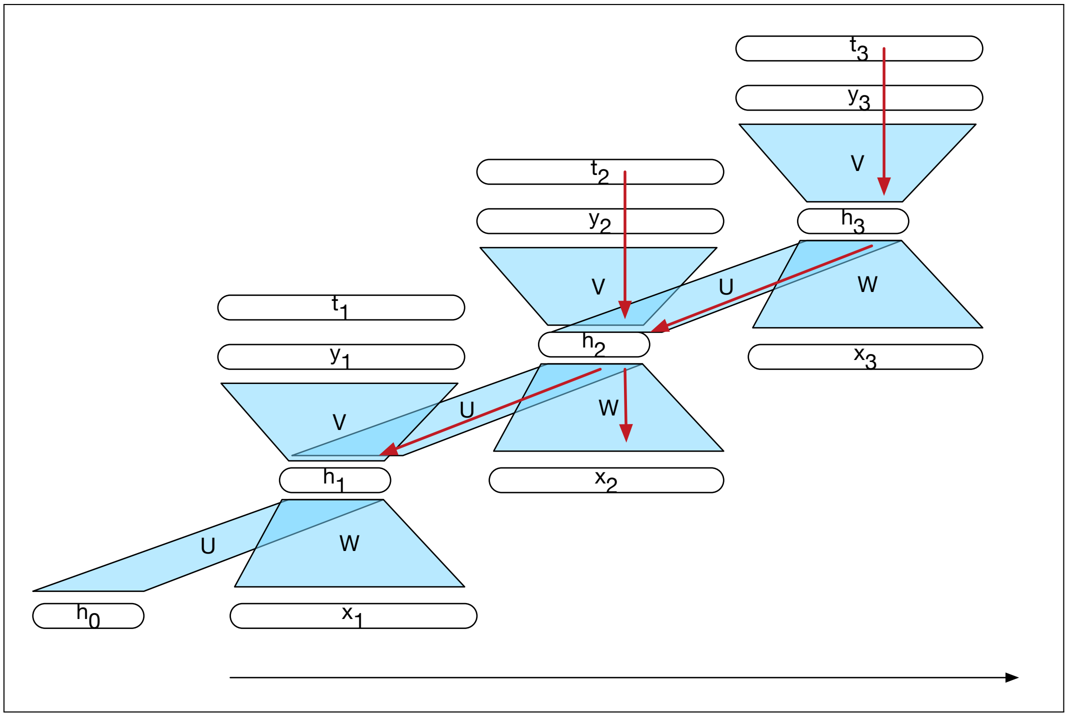


**Figure 9.6** The backpropagation of errors in a simple RNN $t_i$ vectors represent the targets for each element of the sequence from the training data. The red arrows illustrate the flow of backpropagated errors required to calculate the gradients for $U$, $V$ and $W$ at time 2. The two incoming arrows converging on $h_2$ signal that these errors need to be summed.

Moving on, we need to compute the corresponding gradients for the weight matrices $W$ and $U$: $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial U}$. Here we encounter the first substantive change from feedforward networks. The hidden state at time $t$ contributes to the output and associated error at time $t$ and to the output and error at the next timestep, $t+1$. Therefore, the error term, $\delta_h$, for the hidden layer must be the sum of the error term from the current output and its error from the next time step.

$$\delta_h = g'(z)V\delta_{out} + \delta_{next}$$

Given this total error term for the hidden layer, we can compute the gradients for the weights $U$ and $W$ using the chain rule as we did in Chapter 7.

$$\begin{aligned}\frac{dL}{dW} &= \frac{dL}{dz}\frac{dz}{da}\frac{da}{dW}\\ \frac{dL}{dU} &= \frac{dL}{dz}\frac{dz}{da}\frac{da}{dU}\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial W} &= \delta_h x_t\\ \frac{\partial L}{\partial U} &= \delta_h h_{t-1}\end{aligned}$$

These gradients provide us with the information needed to update the matrices $U$ and $W$.

We're not quite done yet, we still need to assign proportional blame (compute the error term) back to the previous hidden layer $h_{t-1}$ for use in further processing. This involves backpropagating the error from $\delta_h$ to $h_{t-1}$ proportionally based on the weights in $U$.

$$\delta_{next} = g'(z)U\delta_h \tag{9.4}$$

At this point we have all the gradients needed to perform weight updates for each of our three sets of weights. Note that in this simple case there is no need to backpropagate the error through $W$ to the input $x$, since the input training data is assumed to be fixed. If we wished to update our input word or character embeddings we would backpropagate the error through to them as well.

Taken together, all of these considerations lead to a two-pass algorithm for training the weights in RNNs. In the first pass, we perform forward inference, computing $h_t$, $y_t$, and accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step. In the second phase, we process the sequence in reverse, computing the required error terms 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. 1986a, Werbos 1990).

**Backpropagation Through Time**

## 9.1.3 Unrolled Networks as Computation Graphs

We used the unrolled network shown in Fig. 9.5 as a way to illustrate the temporal nature of RNNs. However, with modern computational frameworks and adequate computing resources, explicitly unrolling a recurrent network into a deep feedforward computational graph is quite practical for word-by-word approaches to sentence-level processing. 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 particular 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-by-character sentence 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.

# 9.2 Applications of Recurrent Neural Networks

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 10 and Chapter 23, they form the basis for sequence-to-sequence approaches to summarization, machine translation, and question answering.

## 9.2.1 Recurrent Neural Language Models

We've already seen two ways to create probabilistic language models: $N$-gram models and feedforward networks with sliding windows. Given a fixed preceding context, both attempt to predict the next word in a sequence. More formally, they compute the conditional probability of the next word in a sequence given the preceding words, $P(w_n|w_1^{n-1})$.

In both approaches, the quality of a model is largely dependent on the size of the context and how effectively the model makes use of it. Thus, both $N$-gram and sliding-window neural networks are constrained by the Markov assumption embodied in the following equation.

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}) \tag{9.5}$$

That is, anything outside the preceding context of size $N$ has no bearing on the computation.

Recurrent neural 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 input (Mikolov et al., 2010). Thus, the limited context constraint inherent in both $N$-gram models and sliding window approaches 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 as described in Section 9.1.1. At each step the network retrieves a word embedding for the current word as input and 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.

$$\begin{aligned} P(w_n|w_1^{n-1}) &= y_n & (9.6)\\ &= softmax(Vh_n) & (9.7) \end{aligned}$$

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

$$\begin{aligned} P(w_1^n) &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) & (9.8)\\ &= \prod_{k=1}^{n} y_k & (9.9) \end{aligned}$$

As with the approach introduced in Chapter 7, to train such a model we use a corpus of representative text as training material. The task is to predict the next word in a sequence given the previous words, using cross-entropy as the loss function. Recall that the cross-entropy loss for a single example is the negative log probability assigned to the correct class, which is the result of applying a softmax to the final output layer.

$$L_{CE}(\hat{y},y) = -\log \hat{y}_i \tag{9.10}$$

$$= -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{9.11}$$

Here, the correct class $i$ is the word that actually comes next in the data and $y_i$ is the probability assigned to that word, and the softmax is over the entire vocabulary, which has size $K$. The weights in the network are adjusted to minimize the cross-entropy loss over the training set via gradient descent.

### Generation with Neural Language Models

As we saw with the probabilistic Shakespeare generator in Chapter 3, a fun way to gain insight into a language model is to use Shannon's method (Shannon, 1951) to generate random sentences. The procedure is basically the same as that described on **??**.

- To begin, sample the first 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 the each time step is conditioned on the word generated by the network at 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 return to these more advanced applications in Chapter 10, where we discuss encoder-decoder networks.

Finally, as we did with Shakespeare, we can move beyond informally assessing the quality of generated output by using perplexity to objectively compare the output to a held-out sample of the training corpus.

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \tag{9.12}$$

The lower the perplexity, the better the model.

## 9.2.2 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. The canonical example of such a

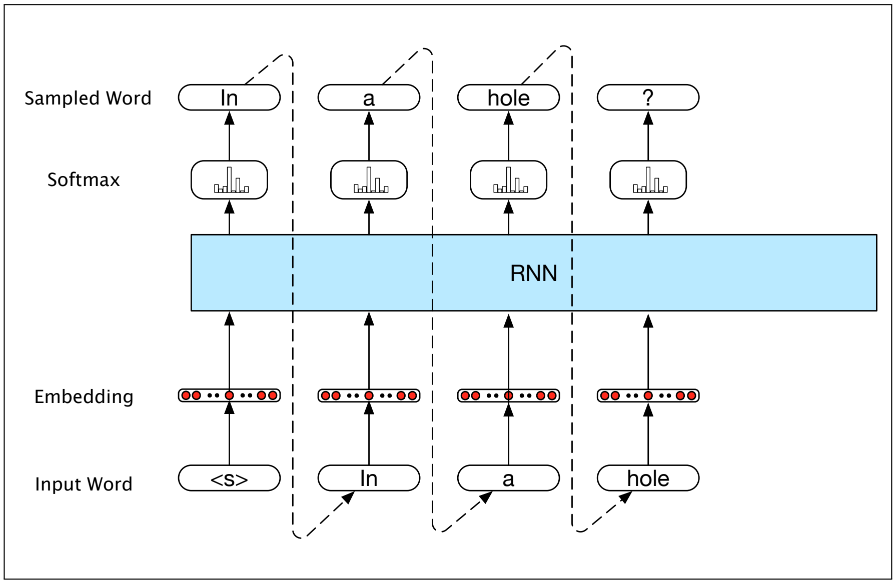


**Figure 9.7** Autoregressive generation with an RNN-based neural language model.

task is part-of-speech tagging, discussed in detail in Chapter 8. In an RNN approach to POS tagging, inputs are word embeddings and the outputs are tag probabilities generated by a softmax layer over the 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 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 tag sequence for a given input, we can run forward inference over

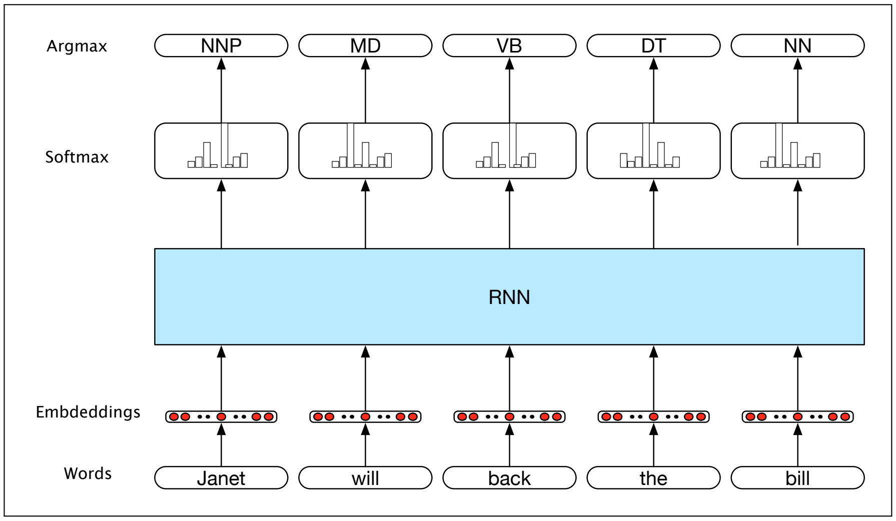


**Figure 9.8** Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

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 timestep, we will again employ the cross-entropy loss during training.

**named entity recognition**

A closely related, and extremely useful, application of sequence labeling is to find and classify *spans* of text that correspond to concepts of interest in some task domain. One example of such a task is **named entity recognition** — the problem of finding all the spans in a text that correspond to names of people, places or organizations (a problem we'll study in more detail in Chapter 17).

To use sequence labeling for a span-recognition problem, we'll use a technique called IOB encoding (Ramshaw and Marcus, 1995). In its simplest form, we label any token that *begins* a span of interest with the label B, tokens that occur *inside* a span are tagged with an I, and any tokens outside of any span of interest are labeled O. Consider the following example:

(9.13) *United cancelled the flight from Denver to San Francisco.*
B O O O O B O B I

Here, the spans of interest are *United*, *Denver* and *San Francisco*.

In applications where we are interested in more than one class of entity (e.g., finding and distinguishing names of people, locations, or organizations), we can specialize the B and I tags to represent each of the more specific classes, thus expanding the tagset from 3 tags to $2*N+1$, where $N$ is the number of classes we're interested in. Applying this approach to the previous example results in the following encoding.

(9.14) *United cancelled the flight from Denver to San Francisco.*
B-ORG O O O O B-LOC O B-LOC I-LOC

Given such an encoding, we've reduced the span recognition task to a per-word labeling task where the inputs are our usual word embeddings and the output consists of a sequence of softmax distributions over the tags at each point in the sequence.

**structure prediction**

Yet another application of sequence labeling is to the problem of **structure prediction**. Here the task is to take an input sequence and produce some kind of structured output, such as a parse tree or meaning representation. One way to model problems like this is to learn a sequence of actions, or operators, which when executed would produce the desired structure. Therefore, instead of predicting a label for each element of an input sequence, the network is trained to select a sequence of actions, which when executed in sequence produce the desired output. The clearest example of this approach is transition-based parsing which borrows the shift-reduce paradigm from compiler construction. We'll return to this application in Chapter 13 when we take up dependency parsing.

### Viterbi and Conditional Random Fields (CRFs)

As we saw when we applied logistic regression to part-of-speech tagging, choosing the maximum probability label for each element in a sequence independently does not necessarily result in an optimal (or even very good) sequence of tags. In the case of IOB tagging, it doesn't even guarantee that the resulting sequence will be well-formed. For example, nothing in approach described in the last section prevents an output sequence from containing an I following an O, even though such a transition is illegal. Similarly, when dealing with multiple classes nothing would prevent an I-LOC tag from following a B-PER tag.

One solution to this problem is to combine the sequence of outputs from a recurrent network with an output-level language model as discussed in Chapter 8. We can then use a variant of the Viterbi algorithm to select the most likely tag sequence. This approach is usually implemented by adding a CRF (Lample et al., 2016) layer as the final layer of recurrent network.

### 9.2.3 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.

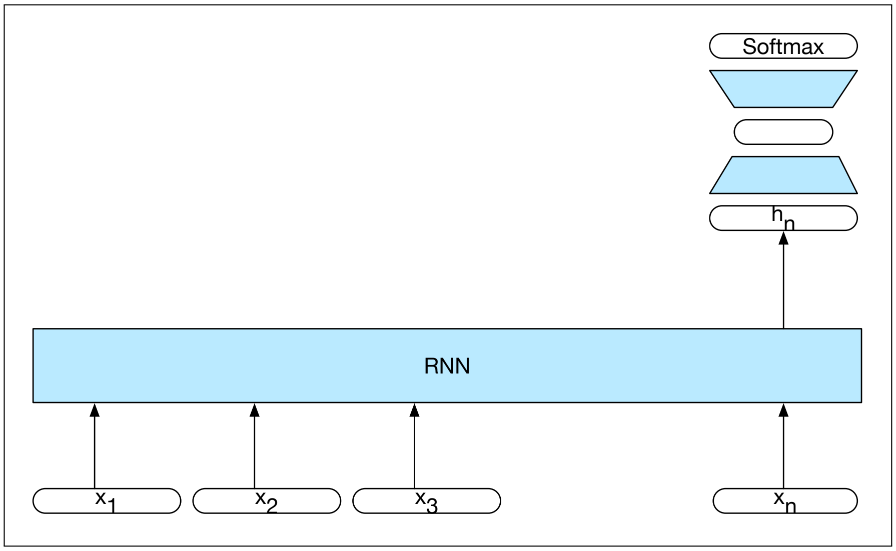


Figure 9.9 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

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 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 9.1.2. 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

## 9.3 Deep Networks: 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.

### 9.3.1 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** 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

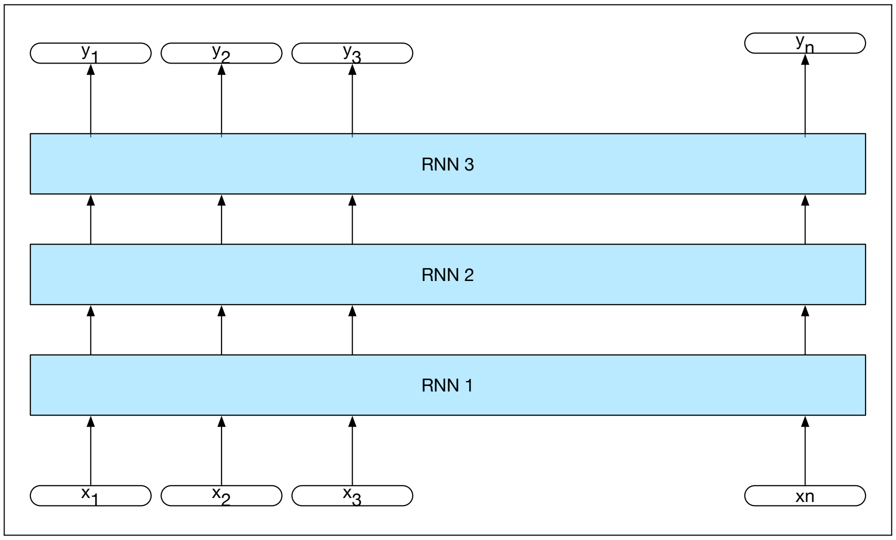


**Figure 9.10** Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

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 networks ability to induce representations at differing levels of abstraction across layers. Just as the early stages of the human visual system detects edges that are then used for finding larger regions and shapes, the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers — representations that might prove difficult to induce in a single RNN.

The optimal number of stacked RNNs is specific to each application and to each training set. However, as the number of stacks is increased the training costs rise quickly.

### 9.3.2 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 its 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 the 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.

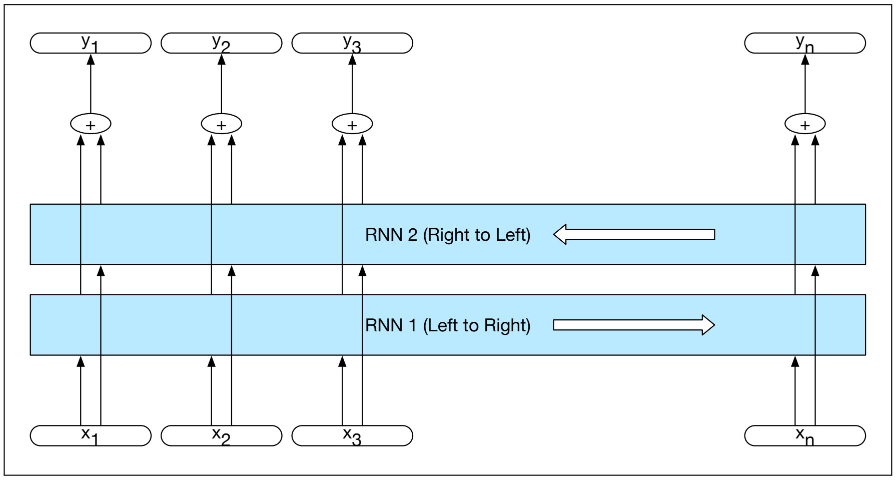


**Figure 9.11** A bidirectional RNN. Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time. The box wrapped around the forward and backward network emphasizes the modular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

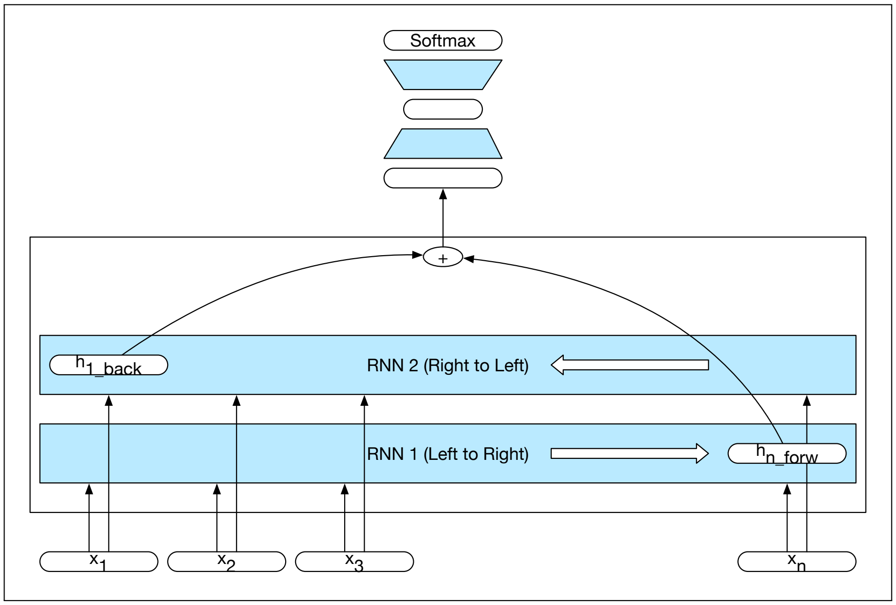


**Figure 9.12** A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

## 9.4 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.

(9.15) 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 9.1.2 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.

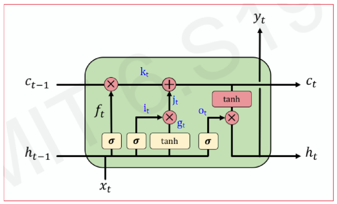
**vanishing gradients**

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.

## 9.4.1 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 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 by the context vector to remove the information from context that is no longer required.

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

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.

$$\boxed{g_t = tanh(U_g h_{t-1} + W_g x_t)} \tag{9.16}$$

**add gate**

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

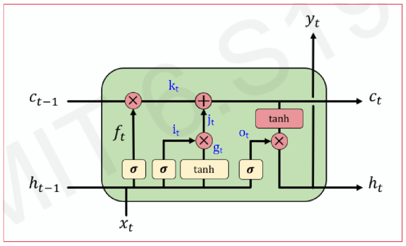
$$\boxed{\begin{aligned} i_t &= \sigma(U_i h_{t-1} + W_i x_t) \\ j_t &= g_t \odot i_t \end{aligned}} \qquad \begin{aligned} &(9.17) \\ &(9.18) \end{aligned}$$

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

$$\boxed{c_t = j_t + k_t} \tag{9.19}$$

**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).

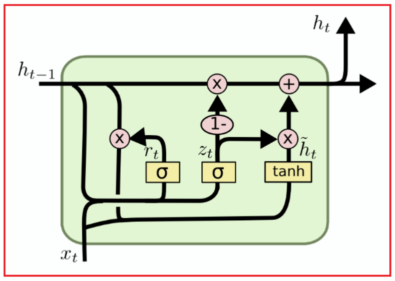


$$\boxed{\begin{aligned} o_t &= \sigma(U_o h_{t-1} + W_o x_t) \\ h_t &= o_t \odot tanh(c_t) \end{aligned}} \qquad \begin{aligned} &(9.20) \\ &(9.21) \\ &(9.22) \end{aligned}$$

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 timestep, 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.

## 9.4.2 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, $z$.



$$\boxed{\begin{aligned} r_t &= \sigma(U_r h_{t-1} + W_r x_t) \\ z_t &= \sigma(U_z h_{t-1} + W_z x_t) \end{aligned}} \qquad \begin{aligned} &(9.23) \\ &(9.24) \end{aligned}$$

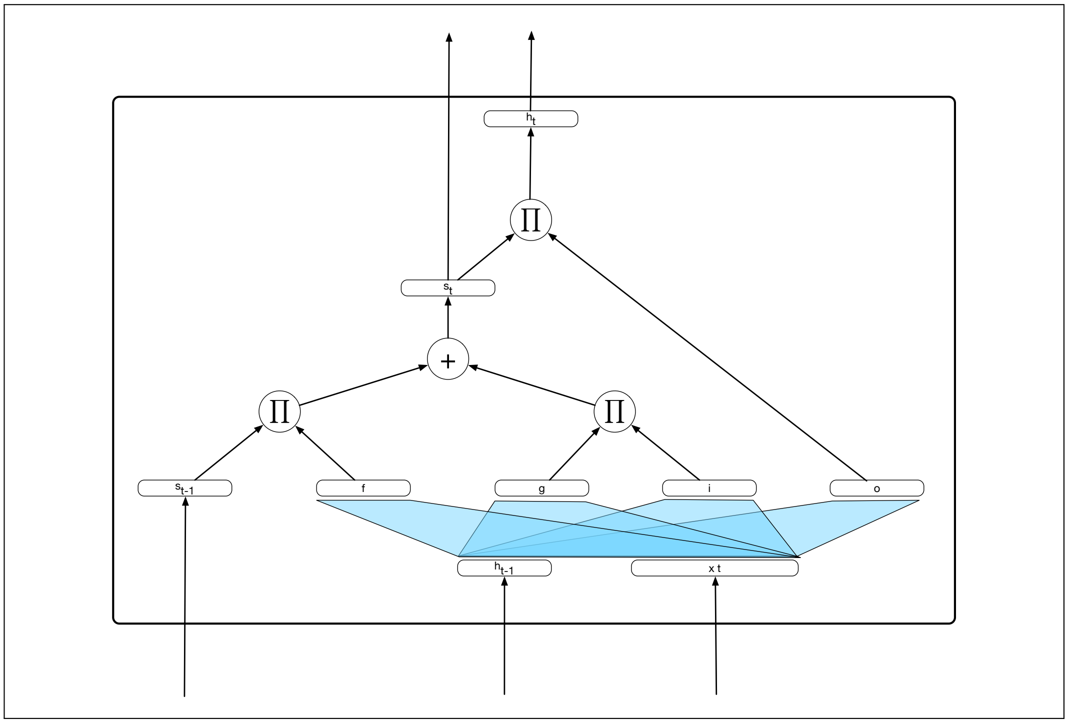
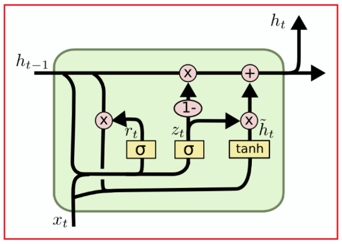


**Figure 9.13** A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$ and an updated context, $c_t$.

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 $t$.



$$\boxed{\tilde{h}_t \;=\; tanh(U(r_t \odot h_{t-1}) + Wx_t)} \tag{9.25}$$

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.

$$\boxed{h_t \;=\; (1-z_t)h_{t-1} + z_t\tilde{h}_t} \tag{9.26}$$

### 9.4.3 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

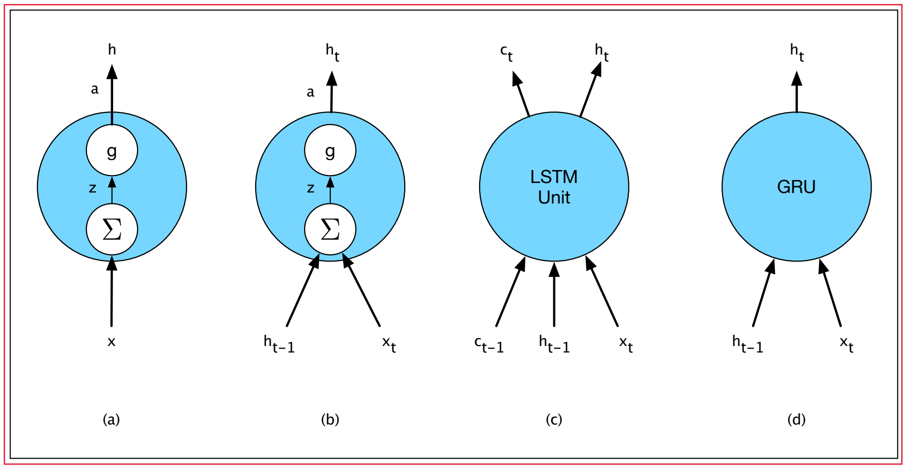


**Figure 9.14** Basic neural units used in feedforward, simple recurrent networks (SRN), long short-term memory (LSTM) and gate recurrent units.

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 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 9.3. 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.

## 9.5 Words, Subwords and Characters

To this point, we've been assuming that the inputs to our networks would be word embeddings. As we've seen, word-based embeddings are great at capturing distributional (syntactic and semantic) similarity between words. However, there are drawbacks to an exclusively word-based approach:

- For some languages and applications, the lexicon is simply too large to practically represent every possible word as an embedding. Some means of composing words from smaller bits is needed.
- No matter how large the lexicon, we will always encounter unknown words due to new words entering the language, misspellings and borrowings from other languages.
- Morphological information, below the word level, is a critical source of infor-

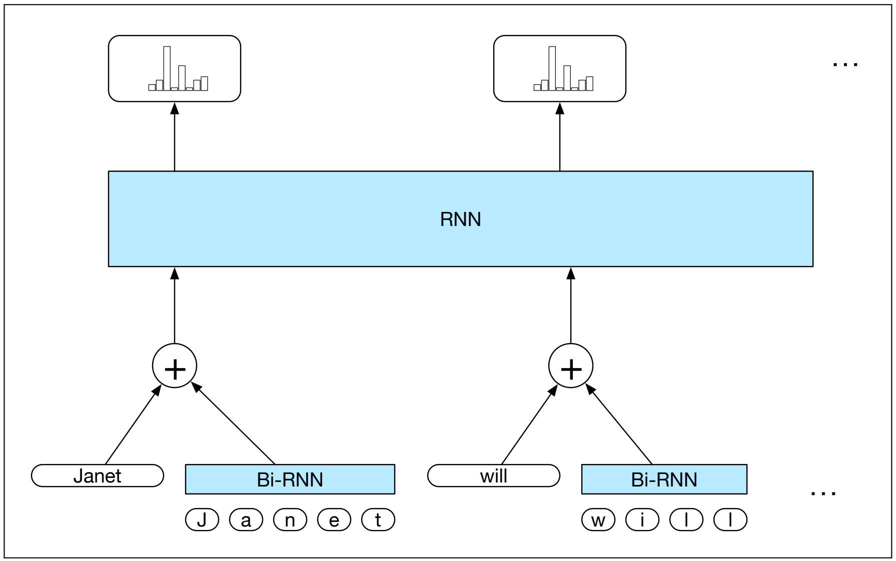


**Figure 9.15** Sequence labeling RNN that accepts distributional word embeddings augmented with character-level word embeddings.

mation for many languages and many applications. Word-based methods are blind to such regularities.

A wide variety of alternatives to the word-based approach have been explored over the past few years. The following are among the primary approaches that have been tried.

- Ignore words altogether and simply use character sequences as the input to RNNs.
- Use subword units such as those derived from byte-pair encoding or phonetic analysis as inputs.
- Use full-blown morphological analysis to derive a linguistically motivated input sequence.

Perhaps not surprisingly there is no clear one-best approach for all applications for all languages.

One particularly successful approach combines word embeddings with embeddings derived from the characters that make up the words. Fig. 9.15 illustrates an approach in the context of part-of-speech tagging. The upper part of the diagram consists of an RNN that accepts an input sequence and outputs a softmax distribution over the tags for each element of the input. Note that this RNN can be arbitrarily complex, consisting of stacked and/or bidirectional network layers.

The inputs to this network consist of ordinary word embeddings enriched with character-level information. Specifically, each input consists of the concatenation of the normal word embedding with embeddings derived from a bidirectional RNN that accepts the character sequences for each word as input, as shown in the lower part of the figure.

The character sequence for each word in the input is run through a bidirectional RNN consisting of two independent RNNs — one that processes the sequence left-to-right and the other right-to-left. As discussed in Section 9.3.2, the final hidden states of the left-to-right and right-to-left networks are concatenated to represent the composite character-level representation of each word. Critically, these character embeddings are trained in the context of the overall task; the loss from the part-of-speech softmax layer is propagated all the way back to the character embeddings.

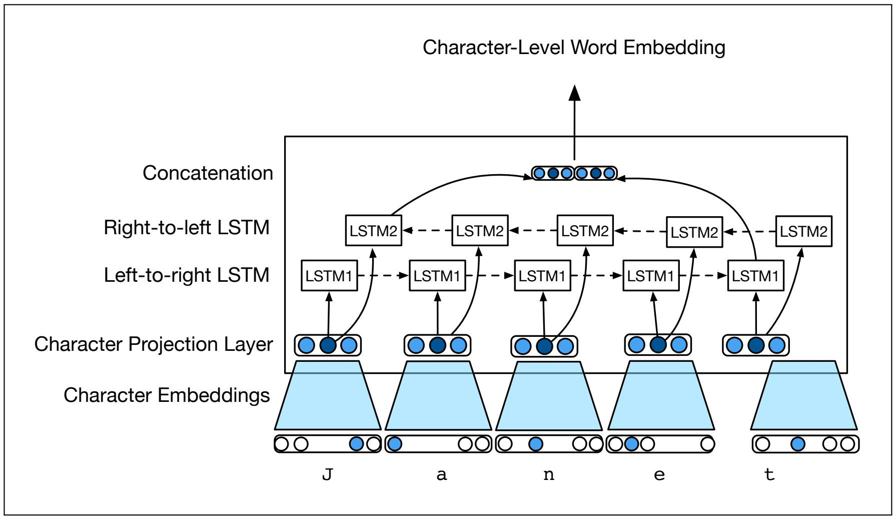


**Figure 9.16** Bi-RNN accepts word character sequences and emits embeddings derived from a forward and backward pass over the sequence. The network itself is trained in the context of a larger end-application where the loss is propagated all the way through to the character vector embeddings.

## 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 timestep.
- 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 et al. 1986b McClelland et al. 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 et al., 1986b).

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).

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. In *EMNLP 2014*, Doha, Qatar, pp. 1724–1734. Association for Computational Linguistics.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, pp. 160–167.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, *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. In *Proceedings of the 10th International Workshop on Semantic Evaluation*, pp. 1197–1201.

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. In *NIPS 2007*, pp. 577–584.

Graves, A., Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*, pp. 6645–6649.

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

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. In *EMNLP 2014*, pp. 720–728.

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

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., and Dyer, C. (2016). Neural architectures for named entity recognition. In *NAACL HLT 2016*.

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. In *EMNLP 2015*, pp. 1520–1530.

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

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

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

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

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pp. 1045–1048.

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. In *EMNLP 2014*, pp. 1532–1543.

Pradhan, S. 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.

Ramshaw, L. A. and Marcus, M. P. (1995). Text chunking using transformation-based learning. In *Proceedings of the 3rd Annual Workshop on Very Large Corpora*, pp. 82–94.

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

Rumelhart, D. E., McClelland, J. L., and The PDP Research Group (1986b). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: *Foundations*. Bradford Books (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.

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

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. In *ACL 2015*, pp. 1127–1137.