CHAPTER

9 Sequence Processing with Recurrent Networks

Time will explain.
Jane Austin, *Persuasion*

In Chapter 7, we explored feedforward neural networks along with their applications to neural language models and text classification. In the case of language models, we saw that such networks can be trained to make predictions about the next word in a sequence given a limited context of preceding words — an approach that is reminiscent of the Markov approach to language modeling discussed in Chapter 3. These models operated by accepting a small fixed-sized window of tokens as input; longer sequences are processed by sliding this window over the input making incremental predictions, with the end result being a sequence of predictions spanning the input. 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.

Unfortunately, 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 problematic 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 example, in Fig. 9.1 the phrase *the ground* appears twice in different windows: once, as shown, in the first and second positions in the window, and in in the preceding step in the second and third slots, thus forcing the network to learn two separate patterns for a single constituent.

The subject of this chapter is recurrent neural networks, a class of networks designed to address these problems by processing sequences *explicitly as sequences*, allowing us to handle variable length inputs without the use of arbitrary fixed-sized windows.

9.1 Simple Recurrent Networks

A recurrent neural network is any network that contains is a cycle within its network connections. That is, any network where the value of a unit is directly, or indirectly, dependent on its own output as an input. In general, such networks are difficult to reason about, and to train. However, within the general class of recurrent networks

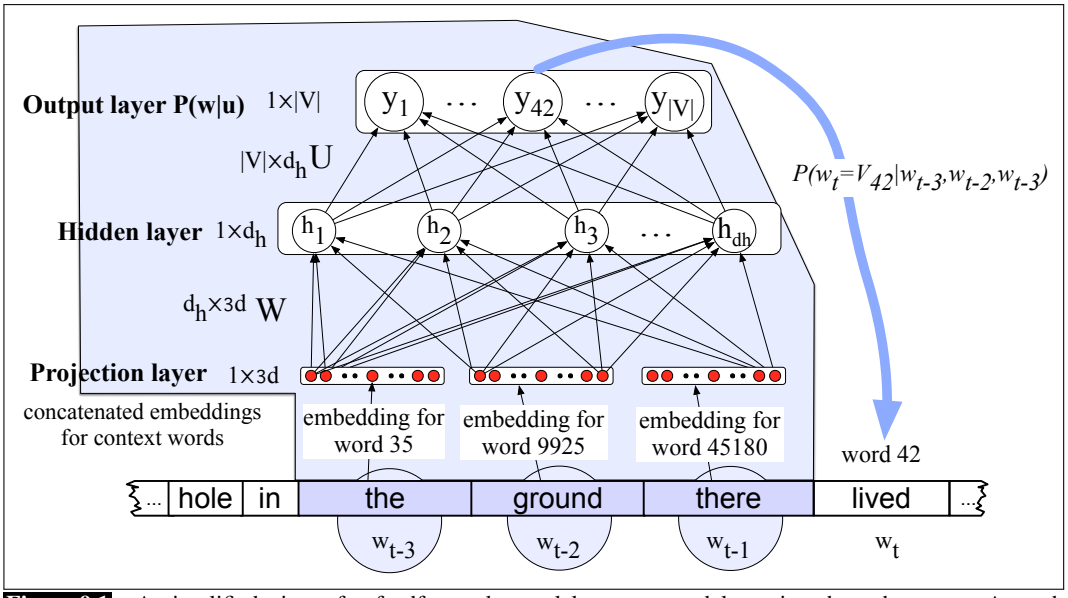

Figure 9.1 A simplified view of a feedforward neural language model moving through a text. At each timestep t the network takes the 3 context words, converts each to a d-dimensional embeddings, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer x for the network.

Simple Recurrent Networks
Elman Networks

there are constrained architectures that have proven to be extremely useful when applied to language problems. In this section, we'll introduce a class of recurrent networks referred to as **Simple Recurrent Networks** (SRNs) or **Elman Networks** (Elman, 1990). These networks are useful in their own right, and will serve as the basis for more complex approaches to be discussed later in this chapter and again in Chapter 22.

Fig. 9.2 abstractly illustrates the recurrent structure of an SRN. As with ordinary feed-forward 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 activation value for a layer of hidden of units. This hidden layer is, in turn, used to calculate a corresponding output, y_t. Sequences are processed by

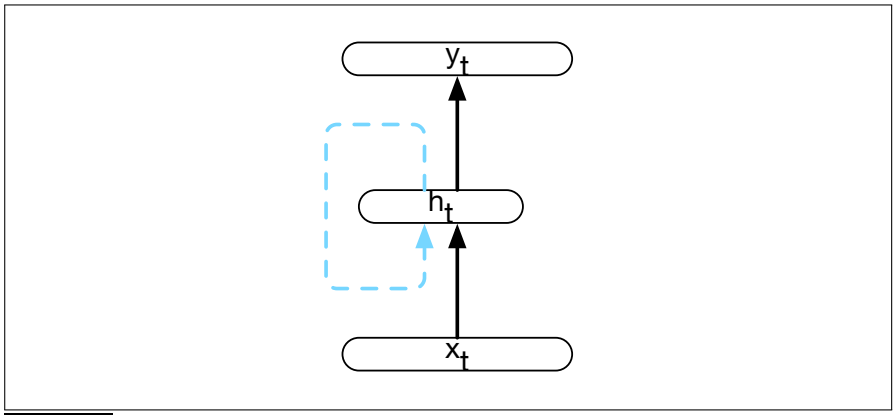

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

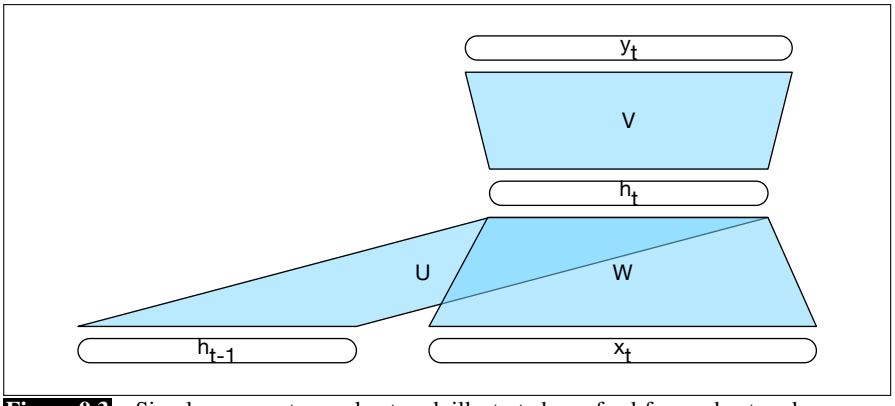

Figure 9.3 Simple recurrent neural network illustrated as a feed-forward network.

presenting one element at a time to the network. The key difference from a feed-forward network lies in the recurrent link shown in the figure with the dashed line. This link augments the input to the 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. Importantly, the architecture does not impose a fixed-length limit on this prior context; the context embodied in the previous hidden layer includes information extending back to the beginning of the sequence.

Adding this temporal dimension makes recurrent networks 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 feed-forward 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 addition 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 will be trained via backpropagation.

9.1.1 Inference in Simple RNNs

Forward inference (mapping a sequence of inputs to a sequence of outputs) in an SRN 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 compute the dot product of the input x_t with the weight matrix W, and the dot product of the hidden layer from the previous time step h_{t-1} with the weight matrix U. We add these values together and pass them through a suitable activation function, g, to arrive at the activation value for the current hidden layer, h_t. Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.

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

In the commonly encountered case of soft classification, finding y_t consists of a softmax computation that provides a normalized probability distribution over the

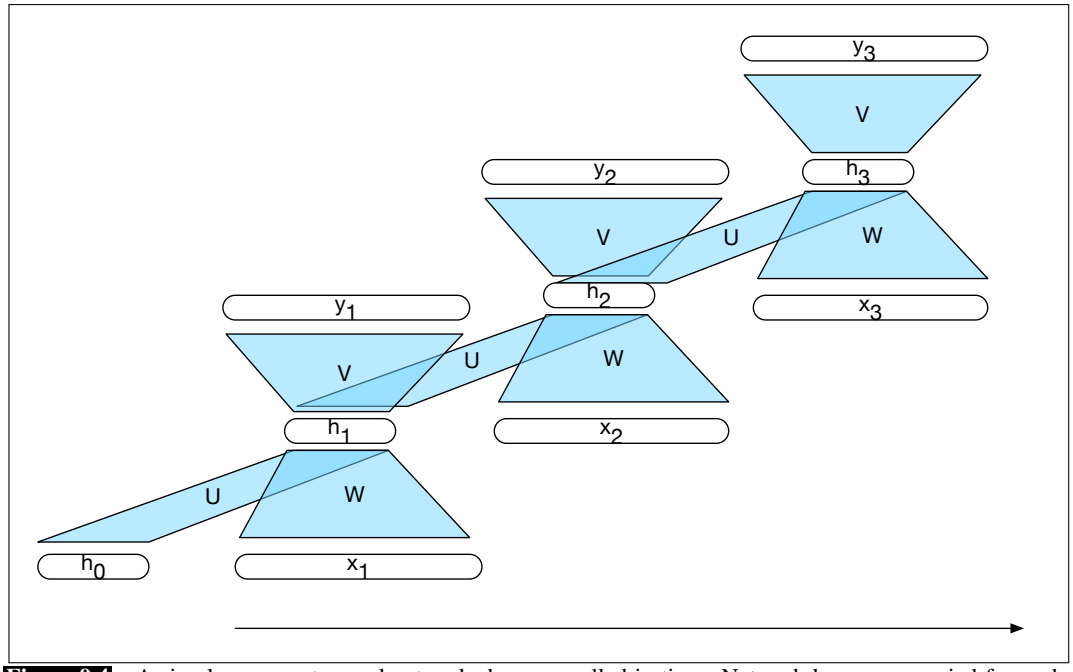

Figure 9.4 A simple recurrent neural network shown unrolled in time. Network layers are copied for each timestep, while the weights U, V and W are shared in common across all timesteps.

possible output classes.

$$y_t = softmax(Vh_t)$$

The sequential nature of simple recurrent networks can be illustrated by *unrolling* the network in time as is shown in Fig. 9.4. In figures such as this, the various layers of units are copied for each time step to illustrate that they will have differing values over time. However the weights themselves are shared across the various timesteps. Finally, 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 shown in Fig. 9.5.

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 9.5 Forward inference in a simple recurrent network.

9.1.2 Training

As we did with feed-forward networks, we'll use a training set, a loss function, and backpropagation to adjust the sets of weights in these recurrent networks. As shown in Fig. 9.3, we now have 3 sets of weights to update: W, the weights from the input

layer to the hidden layer, U, the weights from the previous hidden layer to the current hidden layer, and finally V, the weights from the hidden layer to the output layer.

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, we'll use $a^{[i]}$ to refer 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. A simple two-layer feedforward network with W and V as the first and second sets of weights respectively, would be characterized as follows.

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

Fig. 9.4 illustrates the two considerations that we didn't have to worry about with backpropagation in feed-forward 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 next one*.

Consider the situation where we are examining an input/output pair at time 2 as shown in Fig. 9.4. What do we need to compute the gradients needed to update the weights U, V, and W here? Let's start by reviewing how we compute the gradients required to update V (this computation is unchanged from feed-forward 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 is just the derivative of the loss function with respect to the network output, which is just the activation of the output layer, 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 just the activation value of the current hidden layer h_t.

It's useful here to use the first two terms to define δ, 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:

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

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

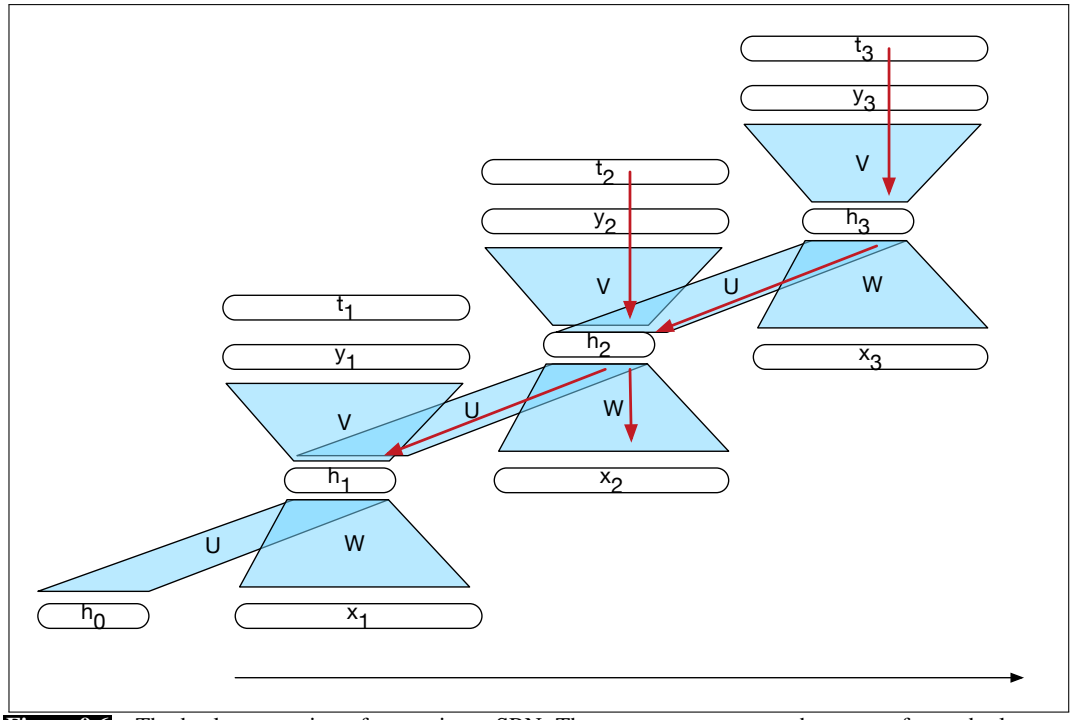

Figure 9.6 The backpropagation of errors in an SRN. The t_i vectors represent the targets for each element of the sequence from the training data. The red arrows illustrate the flow of backpropagated errors required to calculate the updates for U, V and W at time 2. The two incoming arrows converging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

feed-forward 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, δ_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 in the usual way using the chain rule as we did in Chapter 7.

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{da}\frac{da}{dW}$$
$$\frac{dL}{dU} = \frac{dL}{dz}\frac{dz}{da}\frac{da}{dU}$$

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

These gradients provide us with the information needed to update the matrices U and W through ordinary backpropagation.

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.

function BACKPROPTHROUGHTIME(*sequence, network*) **returns** gradients for weight updates
forward pass to gather the loss
backward pass compute error terms and assess blame

Figure 9.7 Backpropagation training through time. The forward pass computes the required loss values at each time step. The backward pass computes the gradients using the values from the forward pass.

This involves backpropagating the error from δ_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. We'll discuss this more in Section 9.5.

Taken together, all of these considerations lead to a two-pass algorithm for training the weights in SRNs. In the first pass, we perform forward inference, computing h_t, y_t, and an 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.

Unfortunately, computing the gradients and updating weights for each item of a sequence individually would be extremely time-consuming. Instead, much as we did with mini-batch training in Chapter 7, we will accumulate gradients for the weights incrementally over the sequence, and then use those accumulated gradients in performing weight updates.

9.1.3 Unrolled Networks as Computational Graphs

We used the unrolled network shown in Fig. 9.4 as a way to understand the dynamic behavior of these networks over time. However, with modern computational frameworks and adequate computing resources, explicitly unrolling a recurrent network into a deep feed-forward 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 SRN, 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 provided with an input sequence such as a training sentence, we can compile a feed-forward graph 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. This approach is called Truncated Backpropagation Through Time (TBTT).

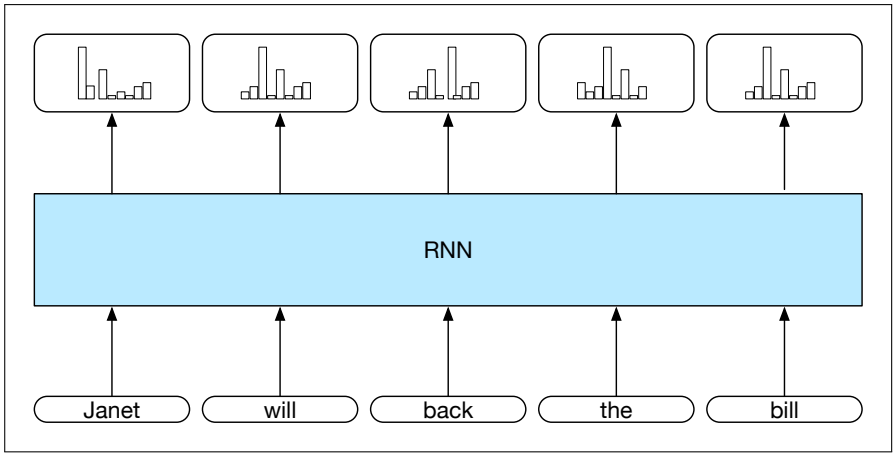

Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

9.2 Applications of RNNs

Simple recurrent 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 22, they form the basic building blocks for sequence to sequence approaches to applications such as summarization and machine translation.

9.2.1 Generation with Neural Language Models

[Coming soon]

9.2.2 Sequence Labeling

In sequence labeling, the network's job is to assign a label to each element of a sequence chosen from a small fixed set of labels. The canonical example of such a task is part-of-speech tagging, discussed in Chapter 8. In a recurrent network-based approach to POS tagging, inputs are words and the outputs are tag probabilities generated by a softmax layer over the POS 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 an actual tag sequence as output, we can run forward inference over the input sequence and select the most likely tag from the softmax at each step. Since we're using a softmax layer to generate the probability distribution over the output tagset at each timestep, we'll rely on the cross entropy loss introduced in Chapter 7 to train the network.

named entity recognition

A closely related, and extremely useful, application of sequence labeling is to find and classify *spans* of text corresponding to items of interest in some task domain. An 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 gory detail in Chapter 17).

To turn a problem like this into a per-word sequence labeling task, we'll use a technique called IOB encoding (Ramshaw and Marcus, 1995). In its simplest form, we'll 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:

(9.5) *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.

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

With such an encoding, the inputs are the usual word embeddings and the output consistes of a sequence of softmax distributions over the tags at each point in the sequence.

9.2.3 Viterbi and Conditional Random Fields (CRFs)

As we saw with applying logistic regression to part-of-speech tagging, choosing the maximum probability label for each element in a sequence does not necessarily result in an optimal (or even very good) tag sequence. 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.

A simple solution to this problem is to use combine the sequence of probability distributions provided by the softmax outputs with a tag-level language model as we did with MEMMs in Chapter 8. Thereby allowing the use of the Viterbi algorithm to select the most likely tag sequence.

[Or a CRF layer... Coming soon]

9.2.4 RNNs for Sequence Classification

Another use of RNNs is to classify entire sequences rather than the tokens within a sequence. 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 hidden layer from the final state of the network is taken to constitute a compressed representation of the entire sequence. This compressed sequence representation can then in turn serve as the input to a feed-forward network trained to select the correct class. Fig. 9.10 illustrates this approach.

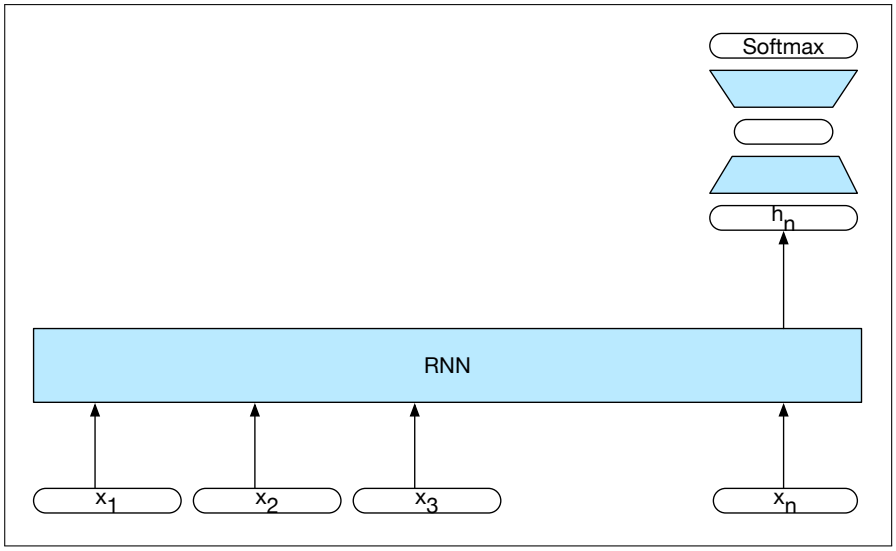

Figure 9.9 Sequence classification using a simple RNN combined with a feedforward network.

Note that in this approach, there are no intermediate outputs for the items in the sequence preceding the last element, and therefore there are no loss terms associated with those individual items. Instead, the loss used to train the network weights is based on the loss from the final classification task. Specifically, we use the output from the softmax layer from the final classifier along with a cross-entropy loss function to drive our network training. The loss 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*.

9.3 Deep Networks: Stacked and Bidirectional RNNs

As suggested by the sequence classification architecture shown in Fig. 9.9, recurrent networks are in fact quite flexible. 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

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.

It has been demonstrated across numerous tasks that stacked RNNs can outper-

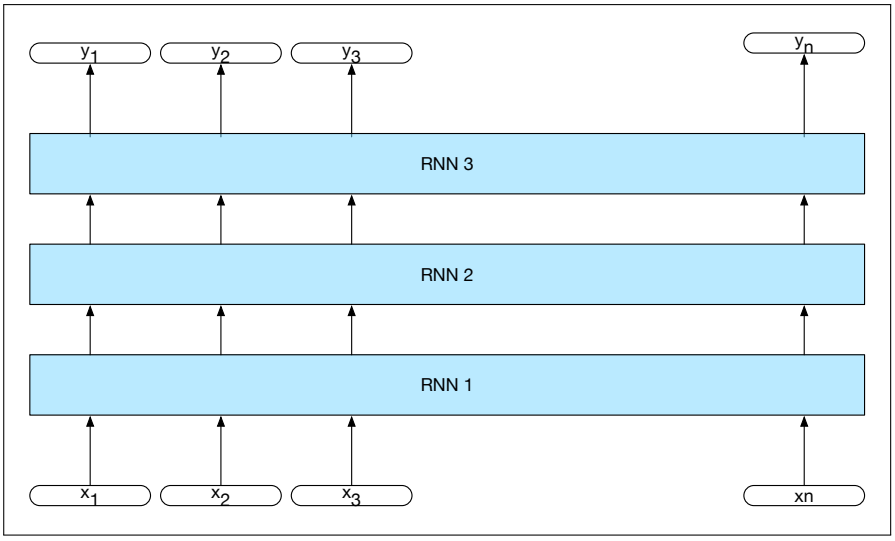

Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

form 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.

9.3.2 Bidirectional RNNs

In an 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.

$$h_t^{forward} = SRN_{forward}(x_1 : x_t)$$

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

Of course, in text-based 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 a recurrent network on an input sequence in reverse, using the same kind of network 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.

$$h_t^{backward} = SRN_{backward}(x_n : x_t)$$

Here, the hidden state $h_t^{backward}$ represents all the information we have discerned about the sequence from t to the end of the sequence.

bidirectional RNN

Putting these networks together results in a **bidirectional RNN**. A Bi-RNN consists of two independent recurrent networks, one where the input is processed from

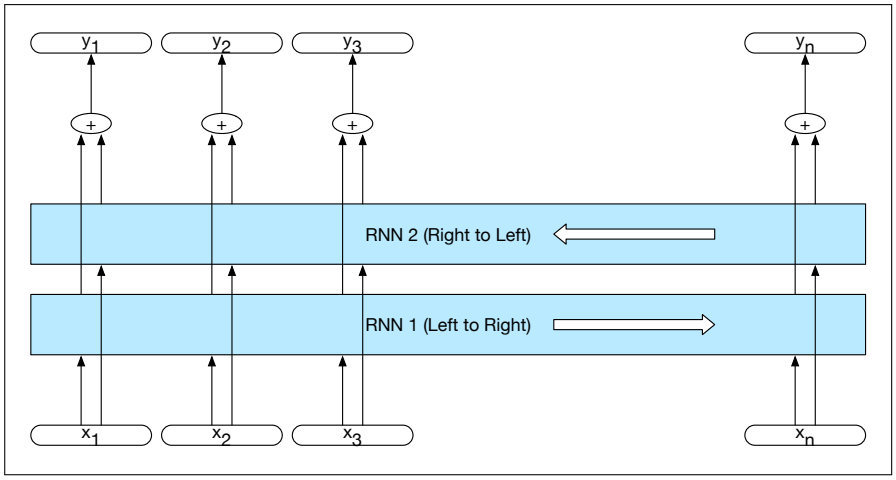

Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time. The box wrapped around the forward and backward network emphasizes the modular nature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

the start to the end, and the other from the end to the start. We can 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.

$$h_t = h_t^{forward} \oplus h_t^{backward} \tag{9.7}$$

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.

9.4 Managing Context in RNNs: LSTMs and GRUs

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

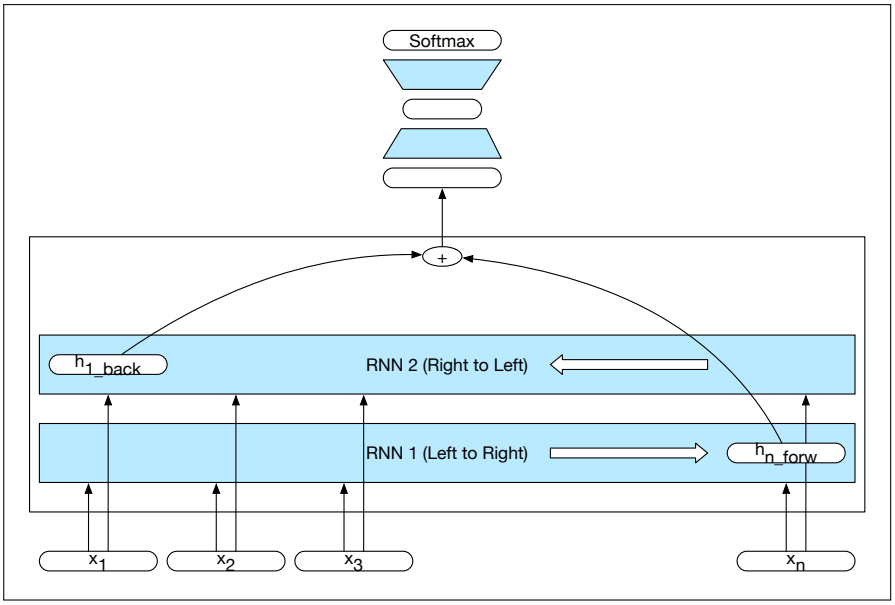

Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

Consider the following example in the context of language models.

(9.8) The flights the airline was cancelling were full.

Assigning a high probability to *was* following *airline* is straightforward since *was* 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 more recent context contains singular constituents. Ideally, a network should be able to retain the distant information about plural *flights* until it is needed, all the while processing intermediate parts of the sequence correctly.

One reason for the inability of SRNs to carry forward critical information is that the hidden layer in SRNs, and, by extension, the weights that determine the values in the hidden layer, are being asked to perform two tasks simultaneously: provide information useful to the decision being made in the current context, and updating and carrying forward information useful for future decisions.

A second difficulty to successfully training simple recurrent networks arises from the need to backpropagate training error back in time through the hidden layers. 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 dot products, as determined by the length of the sequence. A frequent result of this process is that the gradients are either driven to zero or saturate. Situations that are referred to as vanishing gradients or exploding gradients, respectively.

To address these issues more complex network architectures have been designed to explicitly manage the task of maintaining contextual information over time. These approaches treat context as a kind of memory unit that needs to be managed explicitly. More specifically, the network needs to forget information that is no longer needed and to remember information as needed for later decisions.

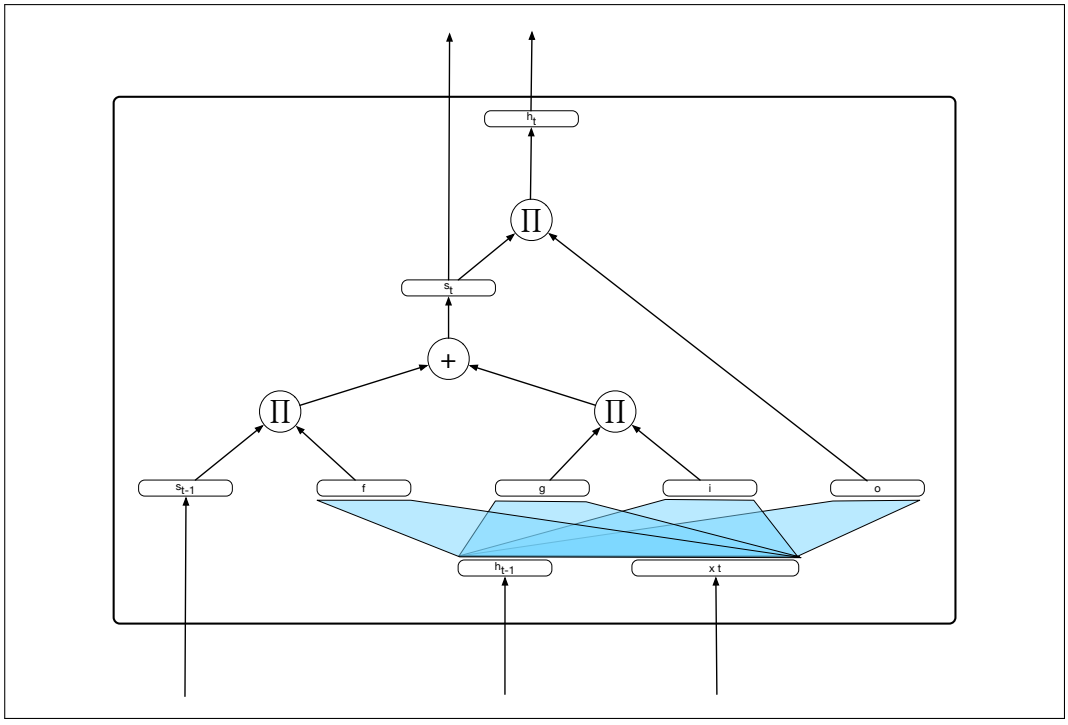

Figure 9.13 A single LSTM memory unit displayed as a computation graph.

9.4.1 Long Short-Term Memory

Long short-term memory (LSTM) networks, 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 the approach is to learn how to manage this context rather than hard-coding a strategy into the architecture.

LSTMs accomplish this through the use of specialized neural units that make use of *gates* that 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 sets of weights that operate sequentially on the context layer.

$$
\begin{aligned}
g_t &= tanh(U_g h_{t-1} + W_g x_t) \\
i_t &= \sigma(U_i h_{t-1} + W_i x_t) \\
f_t &= \sigma(U_f h_{t-1} + W_f x_t) \\
o_t &= \sigma(U_o h_{t-1} + W_o x_t) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
$$

[More on this]

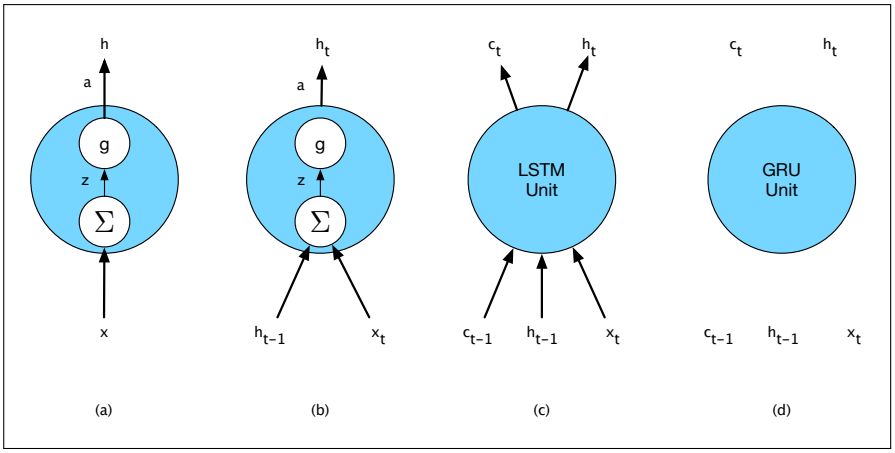

Figure 9.14 Basic neural units used in feed-forward, simple recurrent networks (SRN), long short-term memory (LSTM) and gate recurrent units.

9.4.2 Gated Recurrent Units

While relatively easy to deploy, LSTMs introduce a considerable number of parameters to our networks, and hence carry a much larger training burden. Gated Recurrent Units (GRUs) try to ease this burden by collapsing the forget and add gates of LSTMs into a single update gate with a single set of weights.

[coming soon]

9.4.3 Gated Units, Layers and Networks

The neural units used in LSTMs and GRUs are obviously much more complex than basic feed-forward networks. Fortunately, this complexity is largely 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/outputs and weights associated with each kind of unit.

At the far left, (a) is the basic feed-forward unit $h = g(Wx+b)$. A single set of weights and a single activation function determine its output, and when arranged in a layer there is no connection between the units in the layer. Next, (b) represents the unit in an SRN. Now there are two inputs and additional set of weights to go with it. However, there is still a single activation function and output. When arranged as a layer the hidden layer from each unit feeds in as an input to the next.

Fortunately, the increased complexity of the LSTM and GRU units is encapsulated within the units themselves. The only additional external complexity over the basic recurrent unit (b) is the presence of the additional context vector input and output. This modularity is key to the power and widespread applicability of LSTM and GRU units. Specifically, LSTM and GRU units can be substituted into any of the network architectures described in Section 9.3. And, as with SRNs, multi-layered networks making use of gated units can be unrolled into deep feed-forward networks and trained in the usual fashion with backpropagation.

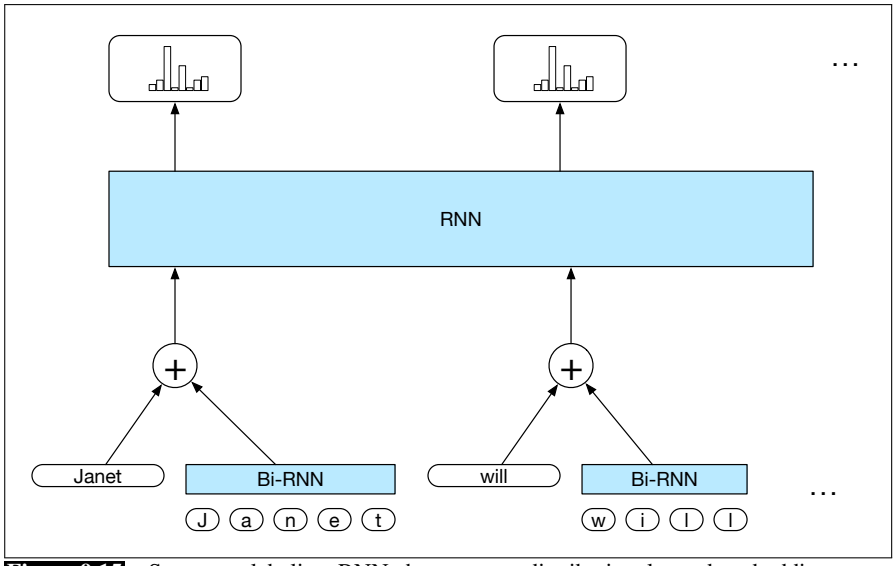

Figure 9.15 Sequence labeling RNN that accepts distributional word embeddings augmented with character-level word embeddings.

9.5 Words, Characters and Byte-Pairs

To this point, we've assumed that the inputs to our networks would be either pre-trained or trained word embeddings. As we've seen, word-based embeddings are great at finding distributional (syntactic and semantic) similarity between words. However, there are significant issues with any solely 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 clearly an important source of information for many applications. Word-based methods are blind to such regularities.

We can overcome some of these issues by augmenting our input word representations 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 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-

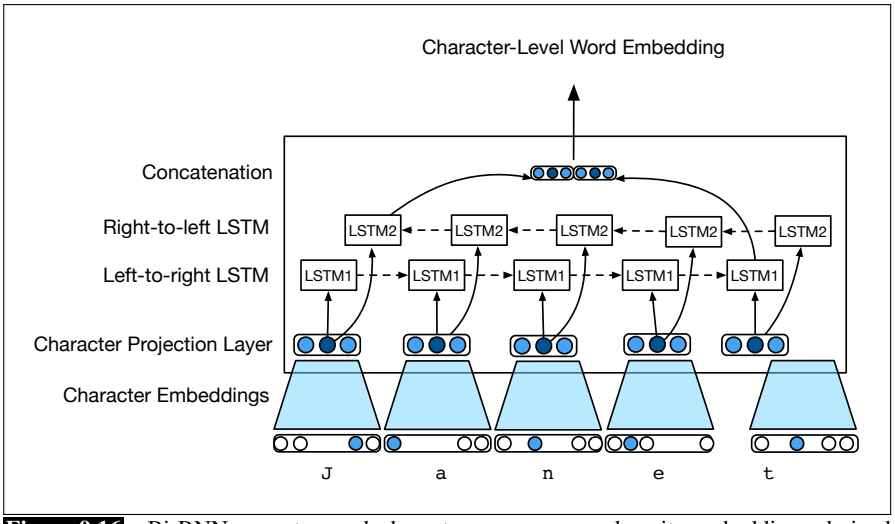

Figure 9.16 Bi-RNN accepts word character sequences and emits embeddings derived from a forward and backward pass over the sequence. The network itself is trained in the context of a larger end-application where the loss is propagated all the way through to the character vector embeddings.

to-right and the other right-to-left. As discussed in Section **??**, 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.

[more on byte-pair encoding approach]

9.6 Summary

- Simple recurrent networks
- Inference and training in SRNs.
- Common use cases for RNNs
  - language modeling
  - sequence labeling
  - sequence classification
- LSTMs and GRUs
- Characters as inputs

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

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.