CHAPTER 14
Recurrent Neural Networks: Modeling Sequences and Stacks

When dealing with language data, it is very common to work with sequences, such as words (sequences of letters), sentences (sequences of words), and documents. We saw how feed-forward networks can accommodate arbitrary feature functions over sequences through the use of vector concatenation and vector addition (CBOW). In particular, the CBOW representations allows to encode arbitrary length sequences as fixed sized vectors. However, the CBOW representation is quite limited, and forces one to disregard the order of features. The convolutional networks also allow encoding a sequence into a fixed size vector. While representations derived from convolutional networks are an improvement over the CBOW representation as they offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.[1]

Recurrent neural networks (RNNs) [Elman, 1990] allow representing arbitrarily sized sequential inputs in fixed-size vectors, while paying attention to the structured properties of the inputs. RNNs, particularly ones with gated architectures such as the LSTM and the GRU, are very powerful at capturing statistical regularities in sequential inputs. They are arguably the strongest contribution of deep-learning to the statistical natural-language processing tool-set.

This chapter describes RNNs as an abstraction: an interface for translating a sequence of inputs into a fixed sized output, that can then be plugged as components in larger networks. Various architectures that use RNNs as a component are discussed. In the next chapter, we deal with concrete instantiations of the RNN abstraction, and describe the Elman RNN (also called Simple RNN), the Long-short-term Memory (LSTM), and the Gated Recurrent Unit (GRU). Then, in Chapter 16 we consider examples of modeling NLP problems using with RNNs.

In Chapter 9, we discussed language modeling and the Markov assumption. RNNs allow for language models that do not make the Markov assumption, and condition the next word on the entire sentence history (all the words preceding it). This ability opens the way to *conditioned generation models*, where a language model that is used as a generator is conditioned on some other signal, such as a sentence in another language. Such models are described in more depth in Chapter 17.

[1]However, as discussed in Section 13.3, hierarchical and dilated convolutional architectures do have the potential of capturing relatively long-range dependencies within a sequence.

14.1 THE RNN ABSTRACTION

We use $\boldsymbol{x_{i:j}}$ to denote the sequence of vectors $\boldsymbol{x_i}, \ldots, \boldsymbol{x_j}$. On a high-level, the RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in}-dimensional vectors $\boldsymbol{x_{1:n}} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$, ($\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}$) and returns as output a single d_{out} dimensional vector $\boldsymbol{y_n} \in \mathbb{R}^{d_{out}}$:

$$\boldsymbol{y_n} = \text{RNN}(\boldsymbol{x_{1:n}}) \tag{14.1}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{y_n} \in \mathbb{R}^{d_{out}}.$$

This implicitly defines an output vector $\boldsymbol{y_i}$ for each prefix $\boldsymbol{x_{1:i}}$ of the sequence $\boldsymbol{x_{1:n}}$. We denote by RNN* the function returning this sequence:

$$\begin{aligned} \boldsymbol{y_{1:n}} &= \text{RNN}^\star(\boldsymbol{x_{1:n}}) \\ \boldsymbol{y_i} &= \text{RNN}(\boldsymbol{x}_{1:i}) \end{aligned} \tag{14.2}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}.$$

The output vector $\boldsymbol{y_n}$ is then used for further prediction. For example, a model for predicting the conditional probability of an event e given the sequence $\boldsymbol{x_{1:n}}$ can be defined as $p(e = j|\boldsymbol{x_{1:n}}) = \text{softmax}(\text{RNN}(\boldsymbol{x_{1:n}}) \cdot \boldsymbol{W} + \boldsymbol{b})_{[j]}$, the jth element in the output vector resulting from the softmax operation over a linear transformation of the RNN encoding $\boldsymbol{y_n} = \text{RNN}(\boldsymbol{x_{1:n}})$. The RNN function provides a framework for conditioning on the entire history $\boldsymbol{x_1}, \ldots, \boldsymbol{x_i}$ without resorting to the Markov assumption which is traditionally used for modeling sequences, described in Chapter 9. Indeed, RNN-based language models result in very good perplexity scores when compared to ngram-based models.

Looking in a bit more detail, the RNN is defined recursively, by means of a function R taking as input a state vector $\boldsymbol{s_{i-1}}$ and an input vector $\boldsymbol{x_i}$ and returning a new state vector $\boldsymbol{s_i}$. The state vector $\boldsymbol{s_i}$ is then mapped to an output vector $\boldsymbol{y_i}$ using a simple deterministic function $O(\cdot)$.[2] The base of the recursion is an initial state vector, $\boldsymbol{s_0}$, which is also an input to the RNN. For brevity, we often omit the initial vector $\boldsymbol{s_0}$, or assume it is the zero vector.

When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs $\boldsymbol{x_i}$ as well as the dimensions of the outputs $\boldsymbol{y_i}$. The dimensions of the states $\boldsymbol{s_i}$ are a function of the output dimension.[3]

[2]Using the $\boldsymbol{O}$ function is somewhat non-standard, and is introduced in order to unify the different RNN models to to be presented in the next chapter. For the Simple RNN (Elman RNN) and the GRU architectures, $\boldsymbol{O}$ is the identity mapping, and for the LSTM architecture $\boldsymbol{O}$ selects a fixed subset of the state.

[3]While RNN architectures in which the state dimension is independent of the output dimension are possible, the current popular architectures, including the Simple RNN, the LSTM, and the GRU do not follow this flexibility.

$$\begin{aligned}\text{RNN}^{\star}(\boldsymbol{x_{1:n}}; \boldsymbol{s_0}) &= \boldsymbol{y_{1:n}} \\ \boldsymbol{y_i} &= O(\boldsymbol{s_i}) \\ \boldsymbol{s_i} &= R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i})\end{aligned} \tag{14.3}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}, \quad \boldsymbol{s_i} \in \mathbb{R}^{f(d_{out})}.$$

The functions R and O are the same across the sequence positions, but the RNN keeps track of the states of computation through the state vector $\boldsymbol{s_i}$ that is kept and being passed across invocations of R.

Graphically, the RNN has been traditionally presented as in Figure 14.1.

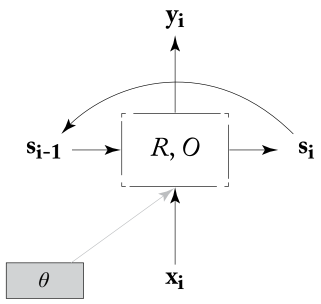

Figure 14.1: Graphical representation of an RNN (recursive).

This presentation follows the recursive definition, and is correct for arbitrarily long sequences. However, for a finite sized input sequence (and all input sequences we deal with are finite) one can *unroll* the recursion, resulting in the structure in Figure 14.2.
While not usually shown in the visualization, we include here the parameters θ in order to highlight the fact that the same parameters are shared across all time steps. Different instantiations of R and O will result in different network structures, and will exhibit different properties in terms of their running times and their ability to be trained effectively using gradient-based methods. However, they all adhere to the same abstract interface. We will provide details of concrete instantiations of R and O—the Simple RNN, the LSTM, and the GRU—in Chapter 15. Before that, let's consider working with the RNN abstraction.

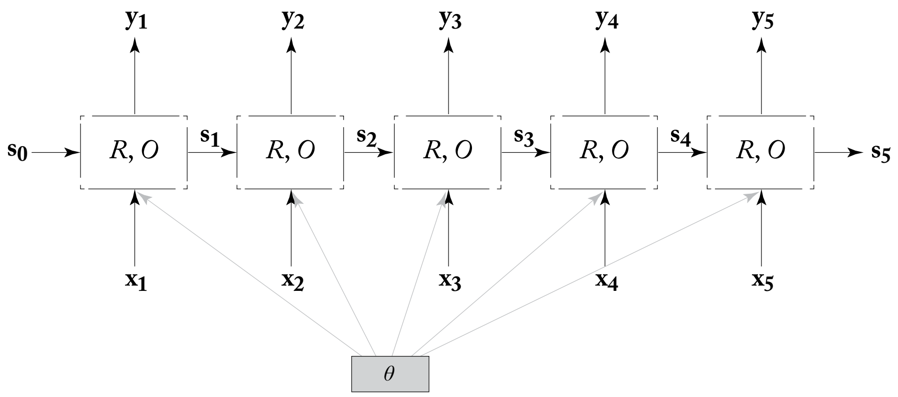

Figure 14.2: Graphical representation of an RNN (unrolled).

First, we note that the value of $\boldsymbol{s_i}$ (and hence $\boldsymbol{y_i}$) is based on the entire input $\boldsymbol{x_1}, \ldots, \boldsymbol{x_i}$. For example, by expanding the recursion for $i = 4$ we get:

$$\begin{aligned}
\boldsymbol{s_4} &= R(\boldsymbol{s_3}, \boldsymbol{x_4}) \\
&= R(\overbrace{R(\boldsymbol{s_2}, \boldsymbol{x_3})}^{\boldsymbol{s_3}}, \boldsymbol{x_4}) \\
&= R(R(\overbrace{R(\boldsymbol{s_1}, \boldsymbol{x_2})}^{\boldsymbol{s_2}}, \boldsymbol{x_3}), \boldsymbol{x_4}) \\
&= R(R(R(\overbrace{R(\boldsymbol{s_0}, \boldsymbol{x_1})}^{\boldsymbol{s_1}}, \boldsymbol{x_2}), \boldsymbol{x_3}), \boldsymbol{x_4}).
\end{aligned} \tag{14.4}$$

Thus, $\boldsymbol{s_n}$ and $\boldsymbol{y_n}$ can be thought of as *encoding* the entire input sequence.[4] Is the encoding useful? This depends on our definition of usefulness. The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are tying to solve.

14.2 RNN TRAINING

Viewed as in Figure 14.2 it is easy to see that an unrolled RNN is just a very deep neural network (or rather, a very large *computation graph* with somewhat complex nodes), in which the same parameters are shared across many parts of the computation, and additional input is added at various layers. To train an RNN network, then, all we need to do is to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph, and then use the backward

[4]Note that, unless $\boldsymbol{R}$ is specifically designed against this, it is likely that the later elements of the input sequence have stronger effect on $\boldsymbol{s_n}$ than earlier ones.

(backpropagation) algorithm to compute the gradients with respect to that loss. This procedure is referred to in the RNN literature as *backpropagation through time* (BPTT) [Werbos, 1990].[5]

What is the objective of the training? It is important to understand that the RNN does not do much on its own, but serves as a trainable component in a larger network. The final prediction and loss computation are performed by that larger network, and the error is back-propagated through the RNN. This way, the RNN learns to encode properties of the input sequences that are useful for the further prediction task. The supervision signal is not applied to the RNN directly, but through the larger network.

Some common architectures of integrating the RNN within larger networks are given below.

14.3 COMMON RNN USAGE-PATTERNS

14.3.1 ACCEPTOR

One option is to base the supervision signal only at the final output vector, $\boldsymbol{y_n}$. Viewed this way, the RNN is trained as an *acceptor*. We observe the final state, and then decide on an outcome.[6] For example, consider training an RNN to read the characters of a word one by one and then use the final state to predict the part-of-speech of that word (this is inspired by Ling et al. [2015b]), an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment (this is inspired by Wang et al. [2015b]) or an RNN that reads in a sequence of words and decides whether it is a valid noun-phrase. The loss in such cases is defined in terms of a function of $\boldsymbol{y_n} = O(\boldsymbol{s_n})$. Typically, the RNN's output vector $\boldsymbol{y_n}$ is fed into a fully connected layer or an MLP, which produce a prediction. The error gradients are then backpropagated through the rest of the sequence (see Figure 14.3).[7] The loss can take any familiar form: cross entropy, hinge, margin, etc.

14.3.2 ENCODER

Similar to the acceptor case, an encoder supervision uses only the final output vector, $\boldsymbol{y_n}$. However, unlike the acceptor, where a prediction is made solely on the basis of the final vector, here the

[5]Variants of the BPTT algorithm include unrolling the RNN only for a fixed number of input symbols at each time: first unroll the RNN for inputs $\boldsymbol{x_{1:k}}$, resulting in $\boldsymbol{s_{1:k}}$. Compute a loss, and backpropagate the error through the network (k steps back). Then, unroll the inputs $\boldsymbol{x_{k+1:2k}}$, this time using $\boldsymbol{s_k}$ as the initial state, and again backpropagate the error for k steps, and so on. This strategy is based on the observations that for the Simple RNN variant, the gradients after k steps tend to vanish (for large enough k), and so omitting them is negligible. This procedure allows training of arbitrarily long sequences. For RNN variants such as the LSTM or the GRU that are designed specifically to mitigate the vanishing gradients problem, this fixed size unrolling is less motivated, yet it is still being used, for example when doing language modeling over a book without breaking it into sentences. A similar variant unrolls the network for the entire sequence in the forward step, but only propagates the gradients back for k steps from each position.

[6]The terminology is borrowed from Finite-State Acceptors. However, the RNN has a potentially infinite number of states, making it necessary to rely on a function other than a lookup table for mapping states to decisions.

[7]This kind of supervision signal may be hard to train for long sequences, especially so with the Simple RNN, because of the vanishing gradients problem. It is also a generally hard learning task, as we do not tell the process on which parts of the input to focus. Yet, it does work very well in many cases.

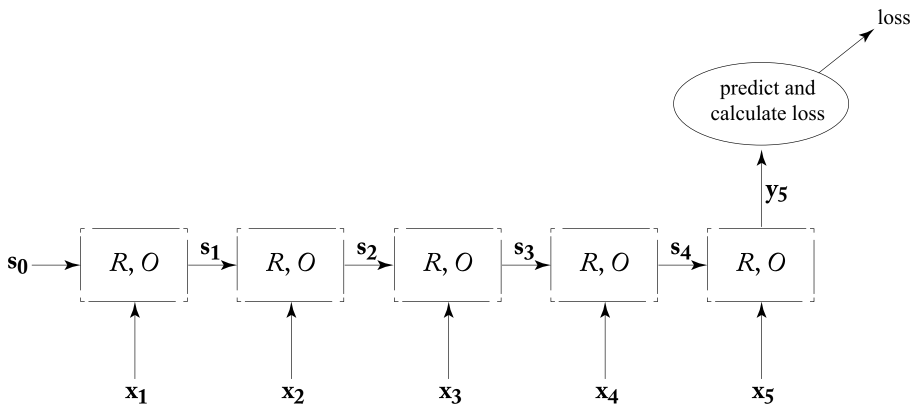

Figure 14.3: Acceptor RNN training graph.

final vector is treated as an encoding of the information in the sequence, and is used as additional information together with other signals. For example, an extractive document summarization system may first run over the document with an RNN, resulting in a vector $\boldsymbol{y_n}$ summarizing the entire document. Then, $\boldsymbol{y_n}$ will be used together with other features in order to select the sentences to be included in the summarization.

14.3.3 TRANSDUCER

Another option is to treat the RNN as a transducer, producing an output $\hat{\boldsymbol{t}}_{\boldsymbol{i}}$ for each input it reads in. Modeled this way, we can compute a local loss signal $L_{\text{local}}(\hat{\boldsymbol{t}}_{\boldsymbol{i}}, \boldsymbol{t_i})$ for each of the outputs $\hat{\boldsymbol{t}}_{\boldsymbol{i}}$ based on a true label $\boldsymbol{t_i}$. The loss for unrolled sequence will then be: $L(\hat{\boldsymbol{t}}_{\boldsymbol{1:n}}, \boldsymbol{t_{1:n}}) = \sum_{i=1}^{n} L_{\text{local}}(\hat{\boldsymbol{t}}_{\boldsymbol{i}}, \boldsymbol{t_i})$, or using another combination rather than a sum such as an average or a weighted average (see Figure 14.4). One example for such a transducer is a sequence tagger, in which we take $\boldsymbol{x_{i:n}}$ to be feature representations for the n words of a sentence, and $\boldsymbol{t_i}$ as an input for predicting the tag assignment of word i based on words $1{:}i$. A CCG super-tagger based on such an architecture provides very strong CCG super-tagging results [Xu et al., 2015], although in many cases a transducer based on a bi-directional RNN (biRNN, see Section 14.4 below) is a better fit for such tagging problems.

A very natural use-case of the transduction setup is for language modeling, in which the sequence of words $\boldsymbol{x_{1:i}}$ is used to predict a distribution over the $(i+1)$th word. RNN-based language models are shown to provide vastly better perplexities than traditional language models [Jozefowicz et al., 2016, Mikolov, 2012, Mikolov et al., 2010, Sundermeyer et al., 2012].

Using RNNs as transducers allows us to relax the Markov assumption that is traditionally taken in language models and HMM taggers, and condition on the entire prediction history.

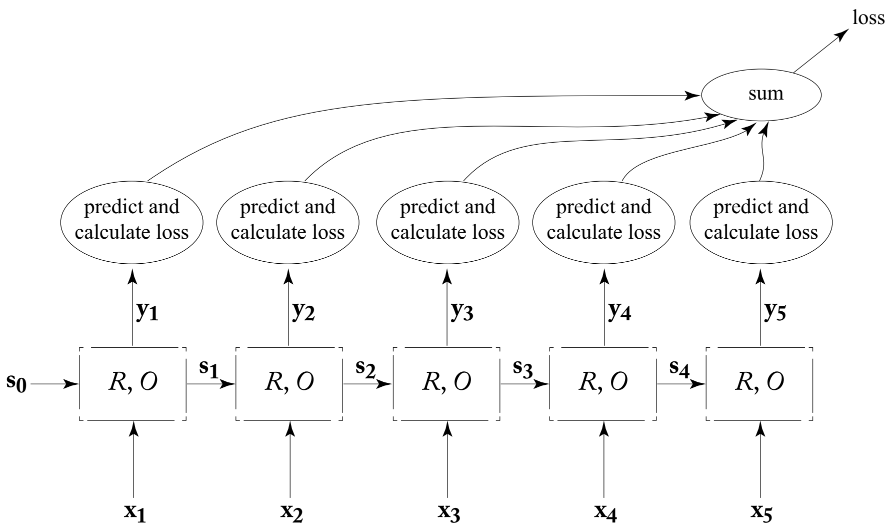

Figure 14.4: Transducer RNN training graph.

Special cases of the RNN transducer is the *RNN generator*, and the related *conditioned-generation* (also called *encoder-decoder*) and the *conditioned-generation with attention* architectures. These will be discussed in Chapter 17.

14.4 BIDIRECTIONAL RNNS (BIRNN)

A useful elaboration of an RNN is a *bidirectional-RNN* (also commonly referred to as biRNN) [Graves, 2008, Schuster and Paliwal, 1997].[8] Consider the task of sequence tagging over a sentence $x_1, \dots, x_n$. An RNN allows us to compute a function of the ith word x_i based on the past—the words $x_{1:i}$ up to and including it. However, the *following* words $x_{i+1:n}$ may also be useful for prediction, as is evident by the common sliding-window approach in which the focus word is categorized based on a window of k words surrounding it. Much like the RNN relaxes the Markov assumption and allows looking arbitrarily back into the past, the biRNN relaxes the fixed window size assumption, allowing to look arbitrarily far at both the past and the future within the sequence.

Consider an input sequence $\boldsymbol{x_{1:n}}$. The biRNN works by maintaining two separate states, $\boldsymbol{s_i^f}$ and $\boldsymbol{s_i^b}$ for each input position i. The *forward state* $\boldsymbol{s_i^f}$ is based on $\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_i}$, while the *backward state* $\boldsymbol{s_i^b}$ is based on $\boldsymbol{x_n}, \boldsymbol{x_{n-1}}, \dots, \boldsymbol{x_i}$. The forward and backward states are generated by two different RNNs. The first RNN (R^f, O^f) is fed the input sequence $\boldsymbol{x_{1:n}}$ as is, while the second RNN (R^b, O^b) is fed the input sequence in reverse. The state representation $\boldsymbol{s_i}$ is

[8]When used with a specific RNN architecture such as an LSTM, the model is called biLSTM.

then composed of both the forward and backward states. The output at position i is based on the concatenation of the two output vectors $\boldsymbol{y}_i = [\boldsymbol{y}_i^f; \boldsymbol{y}_i^b] = [O^f(\boldsymbol{s}_i^f); O^b(\boldsymbol{s}_i^b)]$, taking into account both the past and the future. In other words, $\boldsymbol{y}_i$, the biRNN encoding of the ith word in a sequence is the concatenation of two RNNs, one reading the sequence from the beginning, and the other reading it from the end.

We define biRNN$(\boldsymbol{x}_{1:n}, i)$ to be the output vector corresponding to the ith sequence position:[9]

$$\text{biRNN}(\boldsymbol{x}_{1:n}, i) = \boldsymbol{y}_i = [\text{RNN}^f(\boldsymbol{x}_{1:i}); \text{RNN}^b(\boldsymbol{x}_{n:i})]. \tag{14.6}$$

The vector $\boldsymbol{y}_i$ can then be used directly for prediction, or fed as part of the input to a more complex network. While the two RNNs are run independently of each other, the error gradients at position i will flow both forward and backward through the two RNNs. Feeding the vector $\boldsymbol{y}_i$ through an MLP prior to prediction will further mix the forward and backward signals. Visual representation of the biRNN architecture is given in Figure 14.5.

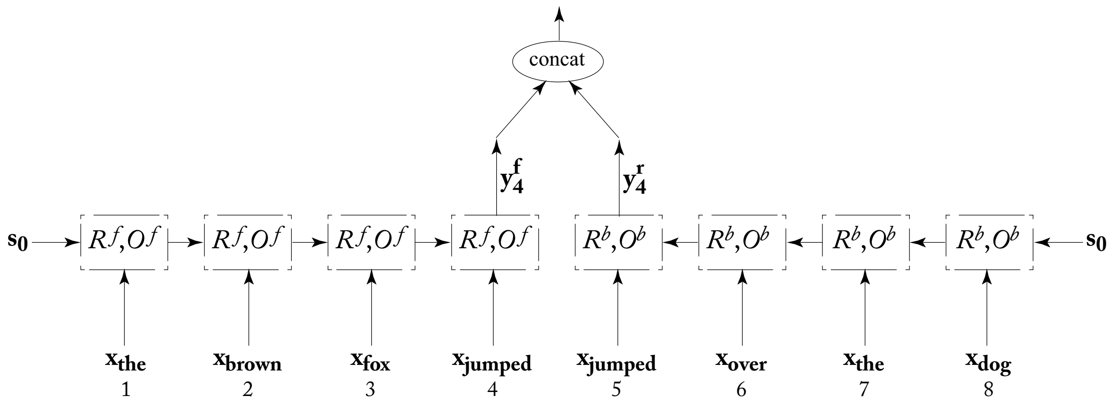

Figure 14.5: Computing the biRNN representation of the word *jumped* in the sentence "the brown fox jumped over the dog."

Note how the vector $\boldsymbol{y}_4$, corresponding to the word *jumped*, encodes an infinite window around (and including) the focus vector $\boldsymbol{x}_{\text{jumped}}$.

Similarly to the RNN case, we also define biRNN$^\star(\boldsymbol{x}_{1:n})$ as the sequence of vectors $\boldsymbol{y}_{1:n}$:

$$\text{biRNN}^\star(\boldsymbol{x}_{1:n}) = \boldsymbol{y}_{i:n} = \text{biRNN}(\boldsymbol{x}_{1:n}, 1), \ldots, \text{biRNN}(\boldsymbol{x}_{1:n}, n). \tag{14.7}$$

[9]The biRNN vector can either a simple concatenation of the two RNN vectors as in Equation (14.6), or followed by another linear-transformation to reduce its dimension, often back to the dimension of the single RNN input:

$$\text{biRNN}(\boldsymbol{x}_{1:n}, i) = \boldsymbol{y}_i = [\text{RNN}^f(\boldsymbol{x}_{1:i}); \text{RNN}^b(\boldsymbol{x}_{n:i})]\boldsymbol{W}. \tag{14.5}$$

This is variant is often used when stacking several biRNNs on top of each other as discussed in Section 14.5.

The n output vectors $\boldsymbol{y}_{i:n}$ can be efficiently computed in linear time by first running the forward and backward RNNs, and then concatenating the relevant outputs. This architecture is depicted in Figure 14.6.

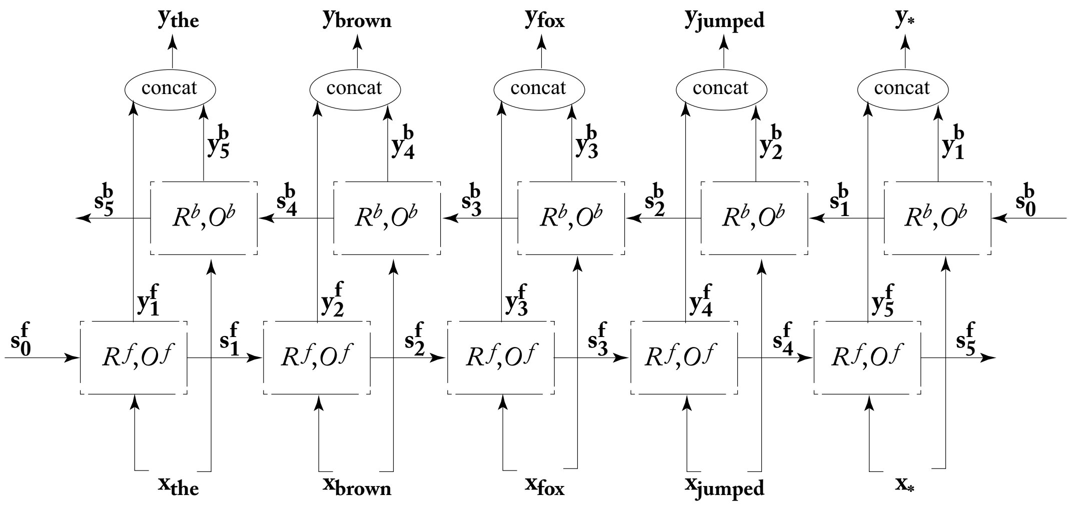
$\mathbf{y}_{\text{the}}$
$\mathbf{y}_{\text{brown}}$
$\mathbf{y}_{\text{fox}}$
$\mathbf{y}_{\text{jumped}}$
$\mathbf{y}_{*}$
concat
$\mathbf{y}_5^{\mathbf{b}}$
$\mathbf{y}_4^{\mathbf{b}}$
$\mathbf{y}_3^{\mathbf{b}}$
$\mathbf{y}_2^{\mathbf{b}}$
$\mathbf{y}_1^{\mathbf{b}}$
$\mathbf{s}_5^{\mathbf{b}}$
$\mathbf{s}_4^{\mathbf{b}}$
$\mathbf{s}_3^{\mathbf{b}}$
$\mathbf{s}_2^{\mathbf{b}}$
$\mathbf{s}_1^{\mathbf{b}}$
$\mathbf{s}_0^{\mathbf{b}}$
R^b,O^b
$\mathbf{y}_1^{\mathbf{f}}$
$\mathbf{y}_2^{\mathbf{f}}$
$\mathbf{y}_3^{\mathbf{f}}$
$\mathbf{y}_4^{\mathbf{f}}$
$\mathbf{y}_5^{\mathbf{f}}$
$\mathbf{s}_0^{\mathbf{f}}$
$\mathbf{s}_1^{\mathbf{f}}$
$\mathbf{s}_2^{\mathbf{f}}$
$\mathbf{s}_3^{\mathbf{f}}$
$\mathbf{s}_4^{\mathbf{f}}$
$\mathbf{s}_5^{\mathbf{f}}$
R^f,O^f
$\mathbf{x}_{\text{the}}$
$\mathbf{x}_{\text{brown}}$
$\mathbf{x}_{\text{fox}}$
$\mathbf{x}_{\text{jumped}}$
$\mathbf{x}_{*}$

Figure 14.6: Computing the biRNN* for the sentence "the brown fox jumped."

The biRNN is very effective for tagging tasks, in which each input vector corresponds to one output vector. It is also useful as a general-purpose trainable feature-extracting component, that can be used whenever a window around a given word is required. Concrete usage examples are given in Chapter 16.

The use of biRNNs for sequence tagging was introduced to the NLP community by Irsoy and Cardie [2014].

14.5 MULTI-LAYER (STACKED) RNNS

RNNs can be stacked in layers, forming a grid [Hihi and Bengio, 1996]. Consider k RNNs, $\text{RNN}_1, \ldots, \text{RNN}_k$, where the jth RNN has states $\boldsymbol{s}^j_{1:n}$ and outputs $\boldsymbol{y}^j_{1:n}$. The input for the first RNN are $\boldsymbol{x}_{1:n}$, while the input of the jth RNN ($j \geq 2$) are the outputs of the RNN below it, $\boldsymbol{y}^{j-1}_{1:n}$. The output of the entire formation is the output of the last RNN, $\boldsymbol{y}^k_{1:n}$. Such layered architectures are often called *deep RNNs*. A visual representation of a three-layer RNN is given in Figure 14.7. biRNNs can be stacked in a similar fashion.[10]

[10]The term *deep-biRNN* is used in the literature to describe to different architecture: in the first, the biRNN state is a concatenation of two deep RNNs. In the second, the output sequence of on biRNN is fed as input to another. My research group found the second variant to often performs better.

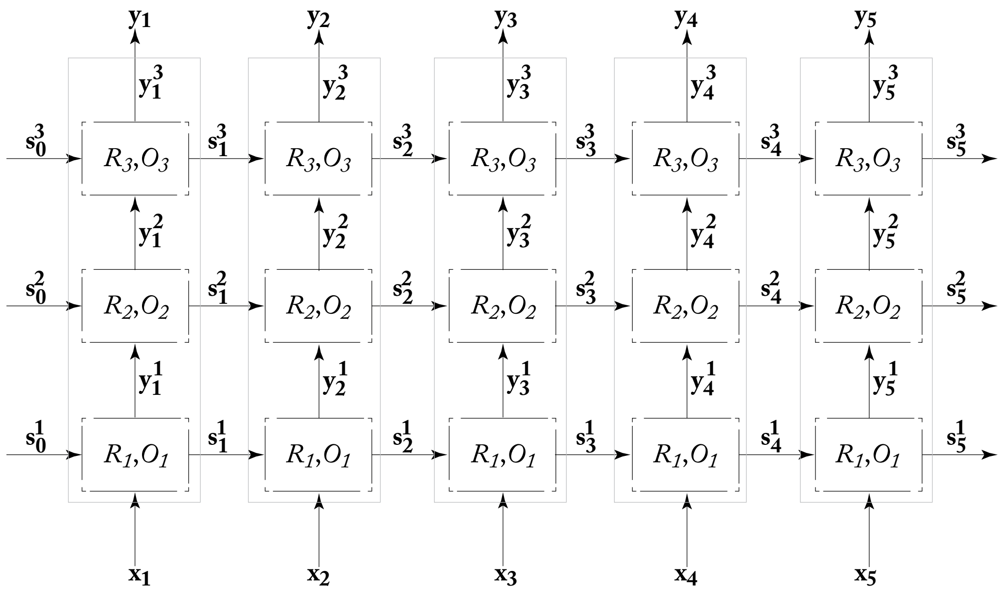
y_1 y_2 y_3 y_4 y_5
y_1^3 y_2^3 y_3^3 y_4^3 y_5^3
s_0^3 s_1^3 s_2^3 s_3^3 s_4^3 s_5^3
R_3,O_3
y_1^2 y_2^2 y_3^2 y_4^2 y_5^2
s_0^2 s_1^2 s_2^2 s_3^2 s_4^2 s_5^2
R_2,O_2
y_1^1 y_2^1 y_3^1 y_4^1 y_5^1
s_0^1 s_1^1 s_2^1 s_3^1 s_4^1 s_5^1
R_1,O_1
x_1 x_2 x_3 x_4 x_5

Figure 14.7: A three-layer ("deep") RNN architecture.

While it is not theoretically clear what is the additional power gained by the deeper architecture, it was observed empirically that deep RNNs work better than shallower ones on some tasks. In particular, Sutskever et al. [2014] report that a four-layers deep architecture was crucial in achieving good machine-translation performance in an encoder-decoder framework. Irsoy and Cardie [2014] also report improved results from moving from a one-layer biRNN to an architecture with several layers. Many other works report result using layered RNN architectures, but do not explicitly compare to one-layer RNNs. In the experiment of my research group, using two or more layers indeed often improves over using a single one.

14.6 RNNS FOR REPRESENTING STACKS

Some algorithms in language processing, including those for transition-based parsing [Nivre, 2008], require performing feature extraction over a stack. Instead of being confined to looking at the k top-most elements of the stack, the RNN framework can be used to provide a fixed-sized vector encoding of the entire stack.

The main intuition is that a stack is essentially a sequence, and so the stack state can be represented by taking the stack elements and feeding them in order into an RNN, resulting in a final encoding of the entire stack. In order to do this computation efficiently (without performing an $O(n)$ stack encoding operation each time the stack changes), the RNN state is maintained together with the stack state. If the stack was push-only, this would be trivial: whenever a new element x is pushed into the stack, the corresponding vector $\boldsymbol{x}$ will be used together with the RNN state $\boldsymbol{s}_i$ in order to obtain a new state $\boldsymbol{s}_{i+1}$. Dealing with pop operation is more challenging, but can be solved by using the persistent-stack data-structure [Goldberg et al., 2013, Okasaki, 1999]. Persistent, or immutable, data-structures keep old versions of themselves intact when modified. The persistent stack construction represents a stack as a pointer to the head of a linked list. An empty stack is the empty list. The push operation appends an element to the list, returning the new head. The pop operation then returns the parent of the head, but keeping the original list intact. From the point of view of someone who held a pointer to the previous head, the stack did not change. A subsequent push operation will add a new child to the same node. Applying this procedure throughout the lifetime of the stack results in a tree, where the root is an empty stack and each path from a node to the root represents an intermediary stack state. Figure 14.8 provides an example of such a tree. The same process can be applied in the computation graph construction, creating an RNN with a tree structure instead of a chain structure. Backpropagating the error from a given node will then affect all the elements that participated in the stack when the node was created, in order. Figure 14.9 shows the computation graph for the stack-RNN corresponding to the last state in Figure 14.8. This modeling approach was proposed independently by Dyer et al. [2015] and Watanabe and Sumita [2015] for transition-based dependency parsing.

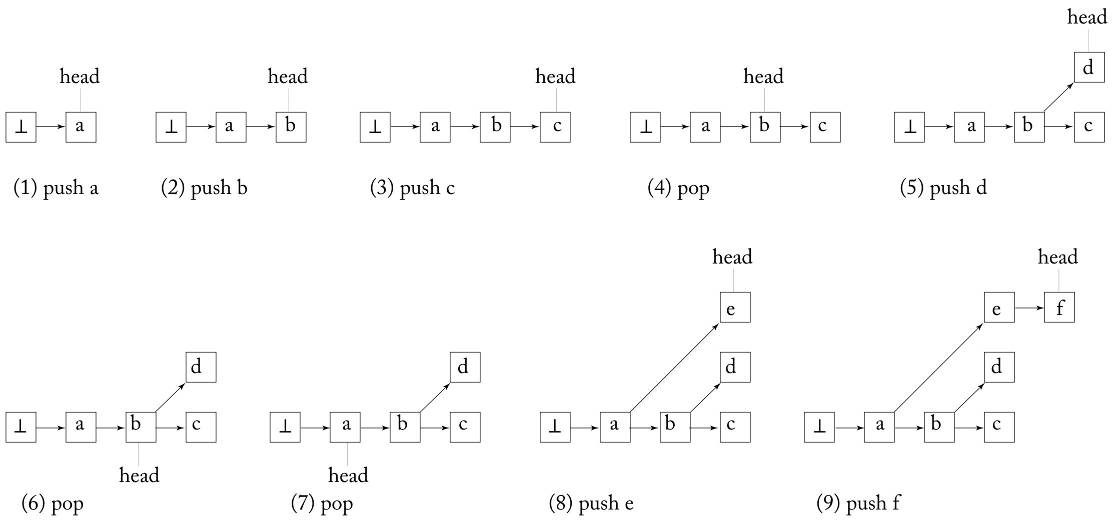

Figure 14.8: An immutable stack construction for the sequence of operations *push a; push b; push c; pop; push d; pop; pop; push e; push f.*

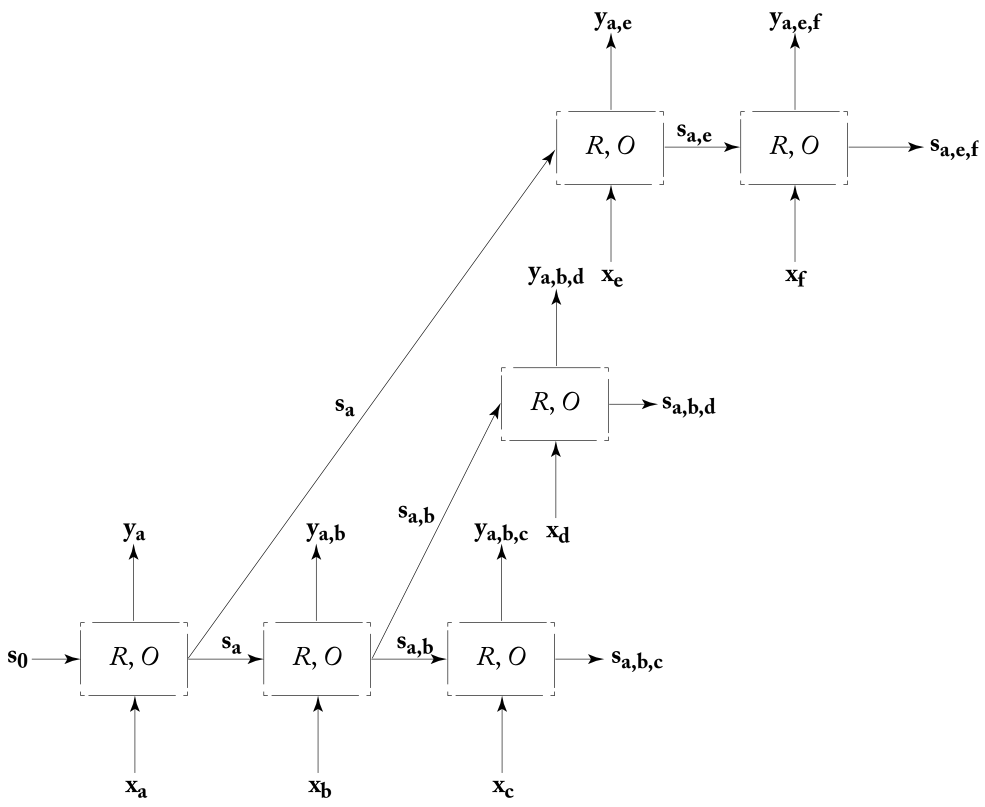

Figure 14.9: The stack-RNN corresponding to the final state in Figure 14.8.

14.7 A NOTE ON READING THE LITERATURE

Unfortunately, it is often the case that inferring the exact model form from reading its description in a research paper can be quite challenging. Many aspects of the models are not yet standardized, and different researchers use the same terms to refer to slightly different things. To list a few examples, the inputs to the RNN can be either one-hot vectors (in which case the embedding matrix is internal to the RNN) or embedded representations; the input sequence can be padded with start-of-sequence and/or end-of-sequence symbols, or not; while the output of an RNN is usually assumed to be a vector which is expected to be fed to additional layers followed by a softmax for prediction (as is the case in the presentation in this tutorial), some papers assume the softmax to be part of the RNN itself; in multi-layer RNN, the "state vector" can be either the output of the top-most layer, or a concatenation of the outputs from all layers; when using the encoder-decoder framework, conditioning on the output of the encoder can be interpreted in various different ways; and so on. On top of that, the LSTM architecture described in the next section has many small variants, which are all referred to under the common name LSTM. Some of these choices are made explicit in the papers, other require careful reading, and others still are not even mentioned, or are hidden behind ambiguous figures or phrasing.

As a reader, be aware of these issues when reading and interpret model descriptions. As a writer, be aware of these issues as well: either fully specify your model in mathematical notation, or refer to a different source in which the model is fully specified, if such a source is available. If using the default implementation from a software package without knowing the details, be explicit of that fact and specify the software package you use. In any case, don't rely solely on figures or natural language text when describing your model, as these are often ambiguous.