Chapter 7

Recurrent Neural Networks

"Democracy is the recurrent suspicion that more than half the people are right more than half the time."—*The New Yorker*, July 3, 1944.

7.1 Introduction

All the neural architectures discussed in earlier chapters are inherently designed for multi-dimensional data in which the attributes are largely independent of one another. However, certain data types such as time-series, text, and biological data contain sequential dependencies among the attributes. Examples of such dependencies are as follows:

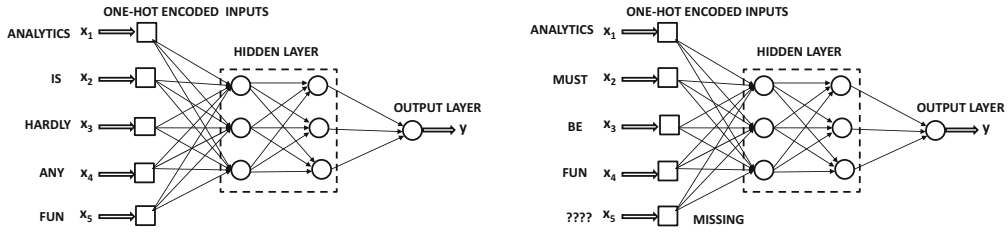
1. In a time-series data set, the values on successive time-stamps are closely related to one another. If one uses the values of these time-stamps as independent features, then key information about the relationships among the values of these time-stamps is lost. For example, the value of a time-series at time t is closely related to its values in the previous window. However, this information is lost when the values at individual time-stamps are treated independently of one another.

2. Although text is often processed as a bag of words, one can obtain better semantic insights when the ordering of the words is used. In such cases, it is important to construct models that take the sequencing information into account. Text data is the most common use case of recurrent neural networks.

3. Biological data often contains sequences, in which the symbols might correspond to amino acids or one of the nucleobases that form the building blocks of DNA.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_7

The individual values in a sequence can be either real-valued or symbolic. Real-valued sequences are also referred to as time-series. Recurrent neural networks can be used for either type of data. In practical applications, the use of symbolic values is more common. Therefore, this chapter will primarily focus on symbolic data in general, and on text data in particular. Throughout this chapter, the default assumption will be that the input to the recurrent network will be a text segment in which the corresponding symbols of the sequence are the word identifiers of the lexicon. However, we will also examine other settings, such as cases in which the individual elements are characters or in which they are real values.



(a) 5-word sentence
"*Analytics is hardly any fun.*"

(b) 4-word sentence
"*Analytics must be fun.*"

Figure 7.1: An attempt to use a conventional neural network for sentiment analysis faces the challenge of variable-length inputs. The network architecture also does not contain any helpful information about sequential dependencies among successive words.

Many sequence-centric applications like text are often processed as bags of words. Such an approach ignores the ordering of words in the document, and works well for documents of reasonable size. However, in applications where the semantic interpretation of the sentence is important, or in which the size of the text segment is relatively small (e.g., a single sentence), such an approach is simply inadequate. In order to understand this point, consider the following pair of sentences:

The cat chased the mouse.
The mouse chased the cat.

The two sentences are clearly very different (and the second one is unusual). However, the bag-of-words representation would deem them identical. Hence, this type of representation works well for simpler applications (such as classification), but a greater degree of linguistic intelligence is required for more sophisticated applications in difficult settings such as *sentiment analysis*, *machine translation*, or *information extraction*.

One possible solution is to avoid the bag-of-words approach and create one input for each position in the sequence. Consider a situation in which one tried to use a conventional neural network in order to perform sentiment analysis on sentences with one input for each position in the sentence. The sentiment can be a binary label depending on whether it is positive or negative. The first problem that one would face is that the length of different sentences is different. Therefore, if we used a neural network with 5 sets of one-hot encoded word inputs (cf. Figure 7.1(a)), it would be impossible to enter a sentence with more than five words. Furthermore, any sentence with less than five words would have missing inputs (cf. Figure 7.1(b)). In some cases, such as Web log sequences, the length of the input sequence might run into the hundreds of thousands. More importantly, small changes in word ordering can lead to semantically different connotations, and *it is important to somehow encode information about the word ordering more directly within the architecture of the*

network. The goal of such an approach would be to reduce the parameter requirements with increasing sequence length; recurrent neural networks provide an excellent example of (parameter-wise) *frugal architectural design* with the help of domain-specific insights. Therefore, the two main desiderata for the processing of sequences include (i) the ability to receive and process inputs in the same order as they are present in the sequence, and (ii) the treatment of inputs at each time-stamp in a similar manner in relation to previous history of inputs. A key challenge is that we somehow need to construct a neural network with a fixed number of parameters, but with the ability to process a variable number of inputs.

These desiderata are naturally satisfied with the use of *recurrent neural networks (RNNs)*. In a recurrent neural network, there is a one-to-one correspondence between the layers in the network and the specific positions in the sequence. The position in the sequence is also referred to as its *time-stamp*. Therefore, instead of a variable number of inputs in a single input layer, the network contains a variable number of layers, and each layer has a single input corresponding to that time-stamp. Therefore, the inputs are allowed to directly interact with down-stream hidden layers depending on their positions in the sequence. Each layer uses the same set of parameters to ensure similar modeling at each time stamp, and therefore the number of parameters is fixed as well. In other words, the same layer-wise architecture is repeated in time, and therefore the network is referred to as *recurrent*. Recurrent neural networks are also feed-forward networks with a specific structure based on the notion of *time layering*, so that they can take a *sequence* of inputs and produce a sequence of outputs. Each temporal layer can take in an input data point (either single attribute or multiple attributes), and optionally produce a multidimensional output. Such models are particularly useful for sequence-to-sequence learning applications like machine translation or for predicting the next element in a sequence. Some examples of applications include the following:

1. The input might be a sequence of words, and the output might be the same sequence shifted by 1, so that we are predicting the next word at any given point. This is a classical *language model* in which we are trying the predict the next word based on the sequential history of words. Language models have a wide variety of applications in text mining and information retrieval [6].

2. In a real-valued time-series, the problem of learning the next element is equivalent to *autoregressive analysis*. However, a recurrent neural network can learn far more complex models than those obtained with traditional time-series modeling.

3. The input might be a sentence in one language, and the output might be a sentence in another language. In this case, one can hook up two recurrent neural networks to learn the translation models between the two languages. One can even hook up a recurrent network with a different type of network (e.g., convolutional neural network) to learn captions of images.

4. The input might be a sequence (e.g., sentence), and the output might be a vector of class probabilities, which is triggered by the end of the sentence. This approach is useful for sentence-centric classification applications like sentiment analysis.

From these four examples, it can be observed that a wide variety of basic architectures have been employed or studied within the broader framework of recurrent neural networks.

There are significant challenges in learning the parameters of a recurrent neural network. One of the key problems in this context is that of the vanishing and the exploding gradient

problem. This problem is particularly prevalent in the context of deep networks like recurrent neural networks. As a result, a number of variants of the recurrent neural network, such as long short-term memory (LSTM) and gated recurrent unit (GRU), have been proposed. Recurrent neural networks and their variants have been used in the context of a variety of applications like sequence-to-sequence learning, image captioning, machine translation, and sentiment analysis. This chapter will also study the use of recurrent neural networks in the context of these different applications.

7.1.1 Expressiveness of Recurrent Networks

Recurrent neural networks are known to be *Turing complete* [444]. Turing completeness means that a recurrent neural network can simulate any algorithm, given enough data and computational resources [444]. This property is, however, not very useful in practice because the amount of data and computational resources required to achieve this goal in arbitrary settings can be unrealistic. Furthermore, there are practical issues in training a recurrent neural network, such as the vanishing and exploding gradient problems. These problems increase with the length of the sequence, and more stable variations such as long short-term memory can address this issue only in a limited way. The neural Turing machine is discussed in Chapter 10, which uses external memory to improve the stability of neural network learning. A neural Turing machine can be shown to be equivalent to a recurrent neural network, and it often uses a more traditional recurrent network, referred to as the *controller*, as an important action-deciding component. Refer to Section 10.3 of Chapter 10 for a detailed discussion.

Chapter Organization

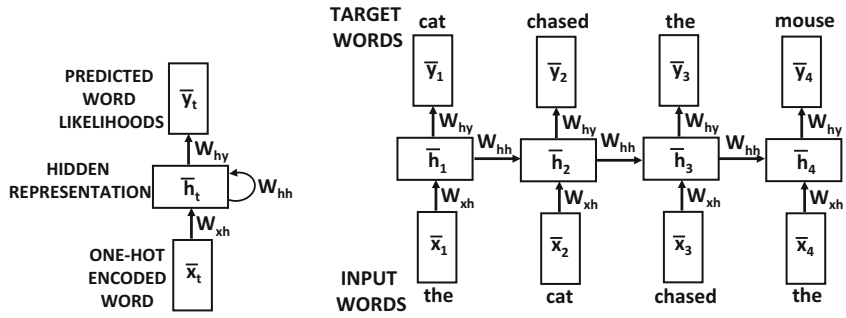
This chapter is organized as follows. The next section will introduce the basic architecture of the recurrent neural network along with the associated training algorithm. The challenges of training recurrent networks are discussed in Section 7.3. Because of these challenges, several variations of the recurrent neural network architecture have been proposed. This chapter will study several such variations. Echo-state networks are introduced in Section 7.4. Long short-term memory networks are discussed in Section 7.5. The gated recurrent unit is discussed in Section 7.6. Applications of recurrent neural networks are discussed in Section 7.7. A summary is given in Section 7.8.

7.2 The Architecture of Recurrent Neural Networks

In the following, the basic architecture of a recurrent network will be described. Although the recurrent neural network can be used in almost any sequential domain, its use in the text domain is both widespread and natural. We will assume the use of the text domain throughout this section in order to enable intuitively simple explanations of various concepts. Therefore, the focus of this chapter will be mostly on discrete RNNs, since that is the most popular use case. Note that exactly the same neural network can be used both for building a word-level RNN and a character-level RNN. The only difference between the two is the set of base symbols used to define the sequence. For consistency, we will stick to the word-level RNN while introducing the notations and definitions. However, variations of this setting are also discussed in this chapter.

The simplest recurrent neural network is shown in Figure 7.2(a). A key point here is the presence of the self-loop in Figure 7.2(a), which will cause the hidden state of the neural

network to change after the input of each word in the sequence. In practice, one only works with sequences of finite length, and it makes sense to unfold the loop into a "time-layered" network that looks more like a feed-forward network. This network is shown in Figure 7.2(b). Note that in this case, we have a different node for the hidden state at each time-stamp and the self-loop has been unfurled into a feed-forward network. This representation is mathematically equivalent to Figure 7.2(a), but is much easier to comprehend because of its similarity to a traditional network. The weight matrices in different temporal layers *are shared* to ensure that the same function is used at each time-stamp. The annotations W_{xh}, W_{hh}, and W_{hy} of the weight matrices in Figure 7.2(b) make the sharing evident.



(a) RNN (b) Time-layered representation of (a)

Figure 7.2: A recurrent neural network and its time-layered representation

It is noteworthy that Figure 7.2 shows a case in which each time-stamp has an input, output, and hidden unit. In practice, it is possible for either the input or the output units to be missing at any particular time-stamp. Examples of cases with missing inputs and outputs are shown in Figure 7.3. The choice of missing inputs and outputs would depend on the specific application at hand. For example, in a time-series forecasting application, we might need outputs at each time-stamp in order to predict the next value in the time-series. On the other hand, in a sequence-classification application, we might only need a single output label at the end of the sequence corresponding to its class. In general, it is possible for any subset of inputs or outputs to be missing in a particular application. The following discussion will assume that all inputs and outputs are present, although it is easy to generalize it to the case where some of them are missing by simply removing the corresponding terms or equations.

The particular architecture shown in Figure 7.2 is suited to language modeling. A language model is a well-known concept in natural language processing that predicts the next word, given the previous history of words. Given a sequence of words, their one-hot encoding is fed one at a time to the neural network in Figure 7.2(a). This temporal process is equivalent to feeding the individual words to the inputs at the relevant time-stamps in Figure 7.2(b). A time-stamp corresponds to the position in the sequence, which starts at 0 (or 1), and increases by 1 by moving forward in the sequence by one unit. In the setting of language modeling, the output is a vector of probabilities predicted for the next word in the sequence. For example, consider the sentence:

The cat chased the mouse.

When the word "*The*" is input, the output will be a vector of probabilities of the entire lexicon that includes the word "*cat*," and when the word "*cat*" is input, we will again get a

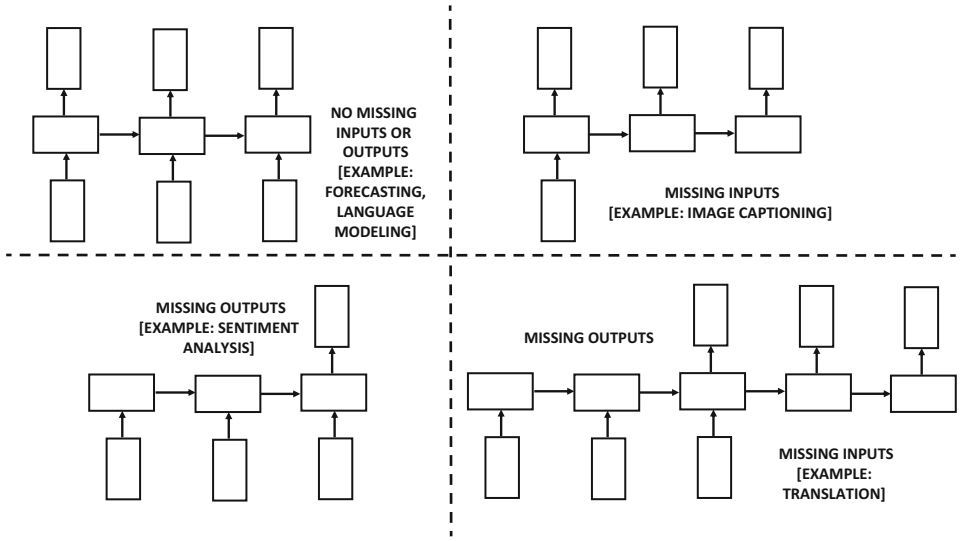


Figure 7.3: The different variations of recurrent networks with missing inputs and outputs

vector of probabilities predicting the next word. This is, of course, the classical definition of a language model in which the probability of a word is estimated based on the immediate history of previous words. In general, the input vector at time t (e.g., one-hot encoded vector of the tth word) is $\overline{x}_t$, the hidden state at time t is $\overline{h}_t$, and the output vector at time t (e.g., predicted probabilities of the $(t+1)$th word) is $\overline{y}_t$. Both $\overline{x}_t$ and $\overline{y}_t$ are d-dimensional for a lexicon of size d. The hidden vector $\overline{h}_t$ is p-dimensional, where p regulates the complexity of the embedding. For the purpose of discussion, we will assume that all these vectors are column vectors. In many applications like classification, the output is not produced at each time unit but is only triggered at the last time-stamp in the end of the sentence. Although output and input units may be present only at a subset of the time-stamps, we examine the simple case in which they are present in all time-stamps. Then, the hidden state at time t is given by a function of the input vector at time t and the hidden vector at time $(t-1)$:

$$\overline{h}_t = f(\overline{h}_{t-1}, \overline{x}_t) \tag{7.1}$$

This function is defined with the use of weight matrices and activation functions (as used by all neural networks for learning), and *the same weights are used at each time-stamp.* Therefore, even though the hidden state evolves over time, the weights and the underlying function $f(\cdot, \cdot)$ remain fixed over all time-stamps (i.e., sequential elements) after the neural network has been trained. A separate function $\overline{y}_t = g(\overline{h}_t)$ is used to learn the output probabilities from the hidden states.

Next, we describe the functions $f(\cdot, \cdot)$ and $g(\cdot)$ more concretely. We define a $p \times d$ input-hidden matrix W_{xh}, a $p \times p$ hidden-hidden matrix W_{hh}, and a $d \times p$ hidden-output matrix W_{hy}. Then, one can expand Equation 7.1 and also write the condition for the outputs as follows:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1})$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

Here, the "tanh" notation is used in a relaxed way, in the sense that the function is applied to the p-dimensional column vector in an element-wise fashion to create a p-dimensional vector with each element in $[-1, 1]$. Throughout this section, this relaxed notation will be used for several activation functions such as tanh and sigmoid. In the very first time-stamp, $\overline{h}_{t-1}$ is assumed to be some default constant vector (such as 0), because there is no input from the hidden layer at the beginning of a sentence. One can also learn this vector, if desired. Although the hidden states change at each time-stamp, the weight matrices stay fixed over the various time-stamps. Note that the output vector $\overline{y}_t$ is a set of continuous values with the same dimensionality as the lexicon. A softmax layer is applied on top of $\overline{y}_t$ so that the results can be interpreted as probabilities. *The p-dimensional output $\overline{h}_t$ of the hidden layer at the end of a text segment of t words yields its embedding, and the p-dimensional columns of W_{xh} yield the embeddings of individual words.* The latter provides an alternative to *word2vec* embeddings (cf. Chapter 2).

Because of the recursive nature of Equation 7.1, the recurrent network has the *ability to compute a function of variable-length inputs.* In other words, one can expand the recurrence of Equation 7.1 to define the function for $\overline{h}_t$ in terms of t inputs. For example, starting at $\overline{h}_0$, which is typically fixed to some constant vector (such as the zero vector), we have $\overline{h}_1 = f(\overline{h}_0, \overline{x}_1)$ and $\overline{h}_2 = f(f(\overline{h}_0, \overline{x}_1), \overline{x}_2)$. Note that $\overline{h}_1$ is a function of only $\overline{x}_1$, whereas $\overline{h}_2$ is a function of both $\overline{x}_1$ and $\overline{x}_2$. In general, $\overline{h}_t$ is a function of $\overline{x}_1 \ldots \overline{x}_t$. Since the output $\overline{y}_t$ is a function of $\overline{h}_t$, these properties are inherited by $\overline{y}_t$ as well. In general, we can write the following:

$$\overline{y}_t = F_t(\overline{x}_1, \overline{x}_2, \ldots \overline{x}_t) \tag{7.2}$$

Note that the function $F_t(\cdot)$ varies with the value of t although its relationship to its immediately previous state is always the same (based on Equation 7.1). Such an approach is particularly useful for variable-length inputs. This setting occurs often in many domains like text in which the sentences are of variable length. For example, in a language modeling application, the function $F_t(\cdot)$ indicates the probability of the next word, taking into account all the previous words in the sentence.

7.2.1 Language Modeling Example of RNN

In order to illustrate the workings of the RNN, we will use a toy example of a single sequence defined on a vocabulary of four words. Consider the sentence:

The cat chased the mouse.

In this case, we have a lexicon of four words, which are {"*the*," "*cat*," "*chased*," "*mouse*"}. In Figure 7.4, we have shown the probabilistic prediction of the next word at each of time-stamps from 1 to 4. Ideally, we would like the probability of the next word to be predicted correctly from the probabilities of the previous words. Each one-hot encoded input vector $\overline{x}_t$ has length four, in which only one bit is 1 and the remaining bits are 0s. The main flexibility here is in the dimensionality p of the hidden representation, which we set to 2 in this case. As a result, the matrix W_{xh} will be a 2×4 matrix, so that it maps a one-hot encoded input vector into a hidden vector $\overline{h}_t$ vector of size 2. As a practical matter, each column of W_{xh} corresponds to one of the four words, and one of these columns is copied by the expression $W_{xh}\overline{x}_t$. Note that this expression is added to $W_{hh}\overline{h}_t$ and then transformed with the tanh function to produce the final expression. The final output $\overline{y}_t$ is defined by $W_{hy}\overline{h}_t$. Note that the matrices W_{hh} and W_{hy} are of sizes 2×2 and 4×2, respectively.

In this case, the outputs are continuous values (not probabilities) in which larger values indicate greater likelihood of presence. These continuous values are eventually converted

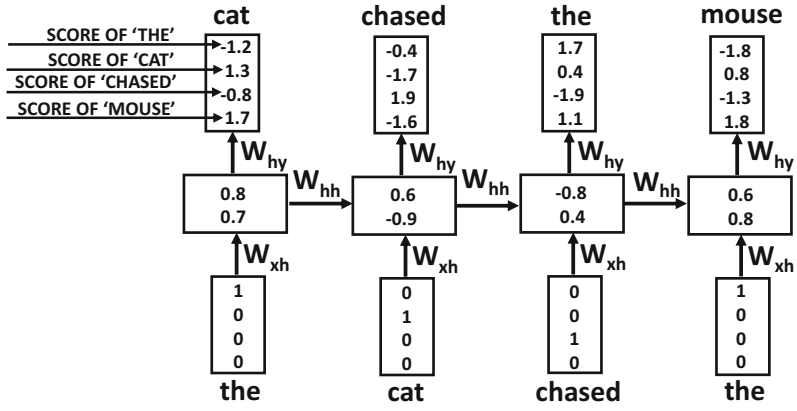


Figure 7.4: Example of language modeling with a recurrent neural network

to probabilities with the softmax function, and therefore one can treat them as substitutes to log probabilities. The word "*cat*" is predicted in the first time-stamp with a value of 1.3, although this value seems to be (incorrectly) outstripped by "*mouse*" for which the corresponding value is 1.7. However, the word "*chased*" seems to be predicted correctly at the next time-stamp. As in all learning algorithms, one cannot hope to predict every value exactly, and such errors are more likely to be made in the early iterations of the backpropagation algorithm. However, as the network is repeatedly trained over multiple iterations, it makes fewer errors over the training data.

7.2.1.1 Generating a Language Sample

Such an approach can also be used to generate an arbitrary sample of a language, once the training has been completed. How does one use such a language model at testing time, since each state requires an input word, and none is available during language generation? The likelihoods of the tokens at the first time-stamp can be generated using the <START> token as input. Since the <START> token is also available in the training data, the model will typically select a word that often starts text segments. Subsequently, the idea is to sample one of the tokens generated at each time-stamp (based on the predicted likelihood), and then use it as an input to the next time-stamp. To improve the accuracy of the sequentially predicted token, one might use beam search to expand on the most likely possibilities by always keeping track of the b best sequence prefixes of any particular length. The value of b is a user-driven parameter. By recursively applying this operation, one can generate an arbitrary sequence of text that reflects the particular training data at hand. If the <END> token is predicted, it indicates the end of that particular segment of text. Although such an approach often results in syntactically correct text, it might be nonsensical in meaning. For example, a character-level RNN[1] authored by Karpathy, Johnson, and Fei Fei [233, 580] was trained on William Shakespeare's plays. A character-level RNN requires the neural network to learn both syntax *and* spelling. After only five iterations of learning across the full data set, the following was a sample of the output:

[1] A long-short term memory network (LSTM) was used, which is a variation on the vanilla RNN discussed here.

KING RICHARD II:
Do cantant,-'for neight here be with hand her,-
Eptar the home that Valy is thee.

NORONCES:
Most ma-wrow, let himself my hispeasures;
An exmorbackion, gault, do we to do you comforr,
Laughter's leave: mire sucintracce shall have theref-Helt.

Note that there are a large number of misspellings in this case, and a lot of the words are gibberish. However, when the training was continued to 50 iterations, the following was generated as a part of the sample:

KING RICHARD II:
Though they good extremit if you damed;
Made it all their fripts and look of love;
Prince of forces to uncertained in conserve
To thou his power kindless. A brives my knees
In penitence and till away with redoom.

GLOUCESTER:
Between I must abide.

This generated piece of text is largely consistent with the syntax and spelling of the archaic English in William Shakespeare's plays, although there are still some obvious errors. Furthermore, the approach also indents and formats the text in a manner similar to the plays by placing new lines at reasonable locations. Continuing to train for more iterations makes the output almost error-free, and some impressive samples are also available at [235].

Of course, the semantic meaning of the text is limited, and one might wonder about the usefulness of generating such nonsensical pieces of text from the perspective of machine learning applications. The key point here is that by providing an additional *contextual* input, such as the neural representation of an image, the neural network can be made to give intelligent outputs such as a grammatically correct description (i.e., caption) of the image. In other words, language models are best used by generating *conditional* outputs.

The primary goal of the language-modeling RNN is not to create arbitrary sequences of the language, but to provide an architectural base that can be modified in various ways to incorporate the effect of the specific context. For example, applications like machine translation and image captioning learn a language model that is *conditioned* on another input such as a sentence in the source language or an image to be captioned. Therefore, the precise design of the application-dependent RNN will use the same principles as the language-modeling RNN, but will make small changes to this basic architecture in order to incorporate the specific context. In all these cases, the key is in choosing the input and output values of the recurrent units in a judicious way, so that one can backpropagate the output errors and learn the weights of the neural network in an application-dependent way.

7.2.2 Backpropagation Through Time

The negative logarithms of the softmax probability of the correct words at the various time-stamps are aggregated to create the loss function. The softmax function is described in Section 3.2.5.1 of Chapter 3, and we directly use those results here. If the output vector $\overline{y}_t$ can be written as $[\hat{y}_t^1 \ldots \hat{y}_t^d]$, it is first converted into a vector of d probabilities using the softmax function:

$$[\hat{p}_t^1 \ldots \hat{p}_t^d] = \text{Softmax}([\hat{y}_t^1 \ldots \hat{y}_t^d])$$

The softmax function above can be found in Equation 3.20 of Chapter 3. If j_t is the index of the ground-truth word at time t in the training data, then the loss function L for all T time-stamps is computed as follows:

$$L = -\sum_{t=1}^{T} \log(\hat{p}_t^{j_t}) \tag{7.3}$$

This loss function is a direct consequence of Equation 3.21 of Chapter 3. The derivative of the loss function with respect to the raw outputs may be computed as follows (cf. Equation 3.22 of Chapter 3):

$$\frac{\partial L}{\partial \hat{y}_t^k} = \hat{p}_t^k - I(k, j_t) \tag{7.4}$$

Here, $I(k, j_t)$ is an indicator function that is 1 when k and j_t are the same, and 0, otherwise. Starting with this partial derivative, one can use the straightforward backpropagation update of Chapter 3 (on the unfurled temporal network) to compute the gradients with respect to the weights in different layers. The main problem is that the weight sharing across different temporal layers will have an effect on the update process. An important assumption in correctly using the chain rule for backpropagation (cf. Chapter 3) is that the weights in different layers are distinct from one another, which allows a relatively straightforward update process. However, as discussed in Section 3.2.9 of Chapter 3, it is not difficult to modify the backpropagation algorithm to handle shared weights.

The main trick for handling shared weights is to first "pretend" that the parameters in the different temporal layers are independent of one another. For this purpose, we introduce the temporal variables $W_{xh}^{(t)}$, $W_{hh}^{(t)}$ and $W_{hy}^{(t)}$ for time-stamp t. Conventional backpropagation is first performed by working under the pretense that these variables are distinct from one another. Then, the contributions of the different temporal avatars of the weight parameters to the gradient are added to create a unified update for each weight parameter. This special type of backpropagation algorithm is referred to as *backpropagation through time (BPTT)*. We summarize the BPTT algorithm as follows:

(i) We run the input sequentially in the forward direction through time and compute the errors (and the negative-log loss of softmax layer) at each time-stamp.

(ii) We compute the gradients of the edge weights in the backwards direction on the unfurled network without any regard for the fact that weights in different time layers are shared. In other words, it is assumed that the weights $W_{xh}^{(t)}$, $W_{hh}^{(t)}$ and $W_{hy}^{(t)}$ in time-stamp t are distinct from other time-stamps. As a result, one can use conventional backpropagation to compute $\frac{\partial L}{\partial W_{xh}^{(t)}}$, $\frac{\partial L}{\partial W_{hh}^{(t)}}$, and $\frac{\partial L}{\partial W_{hy}^{(t)}}$. Note that we have used matrix calculus notations where the derivative with respect to a matrix is defined by a corresponding matrix of element-wise derivatives.

(iii) We add all the (shared) weights corresponding to different instantiations of an edge in time. In other words, we have the following:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{xh}^{(t)}}$$
$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hh}^{(t)}}$$
$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hy}^{(t)}}$$

The above derivations follow from a straightforward application of the multivariate chain rule. As in all backpropagation methods with shared weights (cf. Section 3.2.9 of Chapter 3), we are using the fact that the partial derivative of a temporal copy of each parameter (such as an element of $W_{xh}^{(t)}$) with respect to the original copy of the parameter (such as the corresponding element of W_{xh}) can be set to 1. Here, it is noteworthy that the computation of the partial derivatives with respect to the temporal copies of the weights is not different from traditional backpropagation at all. Therefore, one only needs to wrap the temporal aggregation around conventional backpropagation in order to compute the update equations. The original algorithm for backpropagation through time can be credited to Werbos's seminal work in 1990 [526], long before the use of recurrent neural networks became more popular.

Truncated Backpropagation Through Time

One of the computational problems in training recurrent networks is that the underlying sequences may be very long, as a result of which the number of layers in the network may also be very large. This can result in computational, convergence, and memory-usage problems. This problem is solved by using *truncated backpropagation through time*. This technique may be viewed as the analog of stochastic gradient descent for recurrent neural networks. In the approach, the state values are computed correctly during forward propagation, but the backpropagation updates are done only over segments of the sequence of modest length (such as 100). In other words, only the portion of the loss over the relevant segment is used to compute the gradients and update the weights. The segments are processed in the same order as they occur in the input sequence. The forward propagation does not need to be performed in a single shot, but it can also be done over the relevant segment of the sequence as long as the values in the final time-layer of the segment are used for computing the state values in the next segment of layers. The values in the final layer in the current segment are used to compute the values in the first layer of the next segment. Therefore, forward propagation is always able to accurately maintain state values, although the backpropagation uses only a small portion of the loss. Here, we have described truncated BPTT using non-overlapping segments for simplicity. In practice, one can update using overlapping segments of inputs.

Practical Issues

The entries of each weight matrix are initialized to small values in $[-1/\sqrt{r}, 1/\sqrt{r}]$, where r is the number of columns in that matrix. One can also initialize each of the d columns of the input weight matrix W_{xh} to the *word2vec* embedding of the corresponding word

(cf. Chapter 2). This approach is a form of pretraining. The specific advantage of using this type of pretraining depends on the amount of training data. It can be helpful to use this type of initialization when the amount of available training data is small. After all, pretraining is a form of regularization (see Chapter 4).

Another detail is that the training data often contains a special <START> and an <END> token at the beginning and end of each training segment. These types of tokens help the model to recognize specific text units such as sentences, paragraphs, or the beginning of a particular module of text. The distribution of the words at the beginning of a segment of text is often very different than how it is distributed over the whole training data. Therefore, after the occurrence of <START>, the model is more likely to pick words that begin a particular segment of text.

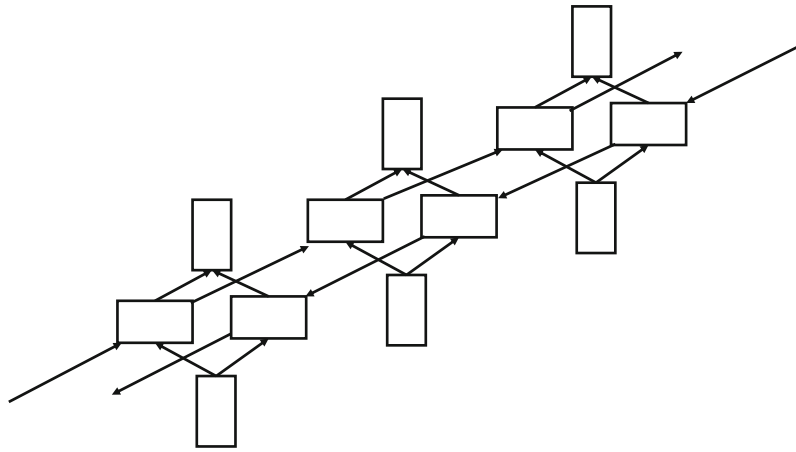


Figure 7.5: Showing three time-layers of a bidirectional recurrent network

There are other approaches that are used for deciding whether to end a segment at a particular point. A specific example is the use of a binary output that decides whether or not the sequence should continue at a particular point. Note that the binary output is in addition to other application-specific outputs. Typically, the sigmoid activation is used to model the prediction of this output, and the cross-entropy loss is used on this output. Such an approach is useful with real-valued sequences. This is because the use of <START> and <END> tokens is inherently designed for symbolic sequences. However, one disadvantage of this approach is that it changes the loss function from its application-specific formulation to one that provides a balance between end-of-sequence prediction and application-specific needs. Therefore, the weights of different components of the loss function would be yet another hyper-parameter that one would have to work with.

There are also several practical challenges in training an RNN, which make the design of various architectural enhancements of the RNN necessary. It is also noteworthy that multiple hidden layers (with long short-term memory enhancements) are used in all practical applications, which will be discussed in Section 7.2.4. However, the application-centric exposition will use the simpler single-layer model for clarity. The generalization of each of these applications to enhanced architectures is straightforward.

7.2.3 Bidirectional Recurrent Networks

One disadvantage of recurrent networks is that the state at a particular time unit only has knowledge about the past inputs up to a certain point in a sentence, but it has no knowledge about future states. In certain applications like language modeling, the results are vastly improved with knowledge about both past and future states. A specific example is handwriting recognition in which there is a clear advantage in using knowledge about both the past and future symbols, because it provides a better idea of the underlying context.

In the bidirectional recurrent network, we have separate hidden states $\overline{h}_t^{(f)}$ and $\overline{h}_t^{(b)}$ for the forward and backward directions. The forward hidden states interact only with each other and the same is true for the backward hidden states. The main difference is that the forward states interact in the forwards direction, while the backwards states interact in the backwards direction. Both $\overline{h}_t^{(f)}$ and $\overline{h}_t^{(b)}$, however, receive input from the same vector $\overline{x}_t$ (e.g., one-hot encoding of word) and they interact with the same output vector $\hat{y}_t$. An example of three time-layers of the bidirectional RNN is shown in Figure 7.5.

There are several applications in which one tries to predict the properties of the current tokens, such as the recognition of the characters in a handwriting sample, a the parts of speech in a sentence, or the classification of each token of the natural language. In general, any property of the *current* word can be predicted more effectively using this approach, because it uses the context on both sides. For example, the ordering of words in several languages is somewhat different depending on grammatical structure. Therefore, a bidirectional recurrent network often models the hidden representations of any specific point in the sentence in a more robust way with the use of backwards and forwards states, irrespective of the specific nuances of language structure. In fact, it has increasingly become more common to use bidirectional recurrent networks in various language-centric applications like speech recognition.

In the case of the bidirectional network, we have separate forward and backward parameter matrices. The forward matrices for the input-hidden, hidden-hidden, and hidden-output interactions are denoted by $W_{xh}^{(f)}$, $W_{hh}^{(f)}$, and $W_{hy}^{(f)}$, respectively. The backward matrices for the input-hidden, hidden-hidden, and hidden-output interactions are denoted by $W_{xh}^{(b)}$, $W_{hh}^{(b)}$, and $W_{hy}^{(b)}$, respectively.

The recurrence conditions can be written as follows:

$$\overline{h}_t^{(f)} = \tanh(W_{xh}^{(f)}\overline{x}_t + W_{hh}^{(f)}\overline{h}_{t-1}^{(f)})$$
$$\overline{h}_t^{(b)} = \tanh(W_{xh}^{(b)}\overline{x}_t + W_{hh}^{(b)}\overline{h}_{t+1}^{(b)})$$
$$\overline{y}_t = W_{hy}^{(f)}\overline{h}_t^{(f)} + W_{hy}^{(b)}\overline{h}_t^{(b)}$$

It is easy to see that the bidirectional equations are simple generalizations of the conditions used in a single direction. It is assumed that there are a total of T time-stamps in the neural network shown above, where T is the length of the sequence. One question is about the forward input at the boundary conditions corresponding to $t = 1$ and the backward input at $t = T$, which are not defined. In such cases, one can use a default constant value of 0.5 in each case, although one can also make the determination of these values as a part of the learning process.

An immediate observation about the hidden states in the forward and backwards direction is that they do not interact with one another at all. Therefore, one could first run the sequence in the forward direction to compute the hidden states in the forward direction, and then run the sequence in the backwards direction to compute the hidden states in the

backwards direction. At this point, the output states are computed from the hidden states in the two directions.

After the outputs have been computed, the backpropagation algorithm is applied to compute the partial derivatives with respect to various parameters. First, the partial derivatives are computed with respect to the output states because both forward and backwards states point to the output nodes. Then, the backpropagation pass is computed only for the forward hidden states starting from $t = T$ down to $t = 1$. The backpropagation pass is finally computed for the backwards hidden states from $t = 1$ to $t = T$. Finally, the partial derivatives with respect to the shared parameters are aggregated. Therefore, the BPTT algorithm can be modified easily to the case of bidirectional networks. One can summarize the steps as follows:

1. Compute forward and backwards hidden states in independent and separate passes.
2. Compute output states from backwards and forward hidden states.
3. Compute partial derivatives of loss with respect to output states and each copy of the output parameters.
4. Compute partial derivatives of loss with respect to forward states and backwards states independently using backpropagation. Use these computations to evaluate partial derivatives with respect to each copy of the forwards and backwards parameters.
5. Aggregate partial derivatives over shared parameters.

Bidirectional recurrent neural networks are appropriate for applications in which the predictions are not causal based on a historical window. A classical example of a causal setting is a stream of symbols in which an event is predicted on the basis of the history of previous symbols. Even though language-modeling applications are formally considered causal applications (i.e., based on immediate history of *previous* words), the reality is that a given word can be predicted with much greater accuracy through the use of the contextual words on each side of it. In general, bidirectional RNNs work well in applications where the predictions are based on bidirectional context. Examples of such applications include handwriting recognition and speech recognition, in which the properties of individual elements in the sequence depend on those on either side of it. For example, if a handwriting is expressed in terms of the strokes, the strokes on either side of a particular position are helpful in recognizing the particular character being synthesized. Furthermore, certain characters are more likely to be adjacent than others.

A bidirectional neural network achieves almost the same quality of results as using an ensemble of two separate recurrent networks, one in which the input is presented in original form and the other in which the input is reversed. The main difference is that the parameters of the forwards and backwards states are trained jointly in this case. However, this integration is quite weak because the two types of states do not interact directly with one another.

7.2.4 Multilayer Recurrent Networks

In all the aforementioned applications, a single-layer RNN architecture is used for ease in understanding. However, in practical applications, a multilayer architecture is used in order to build models of greater complexity. Furthermore, this multilayer architecture can be used in combination with advanced variations of the RNN, such as the LSTM architecture or the gated recurrent unit. These advanced architectures are introduced in later sections.

An example of a deep network containing three layers is shown in Figure 7.6. Note that nodes in higher-level layers receive input from those in lower-level layers. The relationships among the hidden states can be generalized directly from the single-layer network. First, we rewrite the recurrence equation of the hidden layers (for single-layer networks) in a form that can be adapted easily to multilayer networks:

$$\begin{aligned}\overline{h}_t &= \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}) \\ &= \tanh\, W \left[\begin{array}{c} \overline{x}_t \\ \overline{h}_{t-1} \end{array} \right]\end{aligned}$$

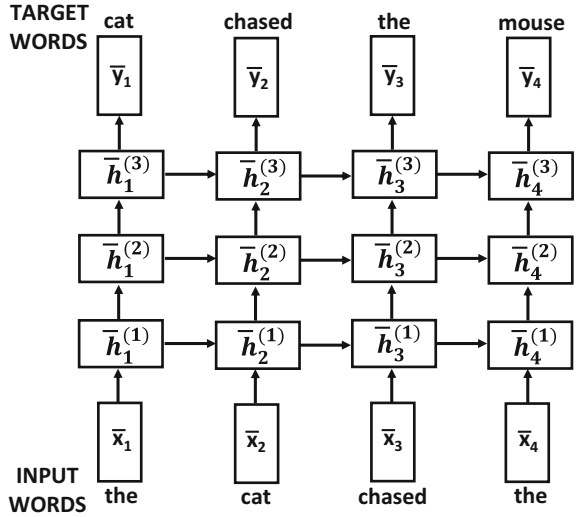


Figure 7.6: Multi-layer recurrent neural networks

Here, we have put together a larger matrix $W = [W_{xh}, W_{hh}]$ that includes the columns of W_{xh} and W_{hh}. Similarly, we have created a larger column vector that stacks up the state vector in the first hidden layer at time $t-1$ and the input vector at time t. In order to distinguish between the hidden nodes for the upper-level layers, let us add an additional superscript to the hidden state and denote the vector for the hidden states at time-stamp t and layer k by $\overline{h}_t^{(k)}$. Similarly, let the weight matrix for the kth hidden layer be denoted by $W^{(k)}$. It is noteworthy that the weights are shared across different time-stamps (as in the single-layer recurrent network), but they are not shared across different layers. Therefore, the weights are superscripted by the layer index k in $W^{(k)}$. The first hidden layer is special because it receives inputs both from the input layer at the current time-stamp and the adjacent hidden state at the previous time-stamp. Therefore, the matrices $W^{(k)}$ will have a size of $p \times (d+p)$ only for the first layer (i.e., $k = 1$), where d is the size of the input vector $\overline{x}_t$ and p is the size of the hidden vector $\overline{h}_t$. Note that d will typically not be the same as p. The recurrence condition for the first layer is already shown above by setting $W^{(1)} = W$. Therefore, let us focus on all the hidden layers k for $k \geq 2$. It turns out that the recurrence condition for the layers with $k \geq 2$ is also in a very similar form as the equation shown above:

$$\overline{h}_t^{(k)} = \tanh\, W^{(k)} \left[\begin{array}{c} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{array} \right]$$

In this case, the size of the matrix $W^{(k)}$ is $p \times (p + p) = p \times 2p$. The transformation from hidden to output layer remains the same as in single-layer networks. It is easy to see that this approach is a straightforward multilayer generalization of the case of single-layer networks. It is common to use two or three layers in practical applications. In order to use a larger number of layers, it is important to have access to more training data in order to avoid overfitting.

7.3 The Challenges of Training Recurrent Networks

Recurrent neural networks are very hard to train because of the fact that the time-layered network is a very deep network, especially if the input sequence is long. In other words, the depth of the temporal layering is input-dependent. As in all deep networks, the loss function has highly varying sensitivities of the loss function (i.e., loss gradients) to different temporal layers. Furthermore, even though the loss function has highly varying gradients to the variables in different layers, the same parameter matrices are shared by different temporal layers. This combination of varying sensitivity and shared parameters in different layers can lead to some unusually unstable effects.

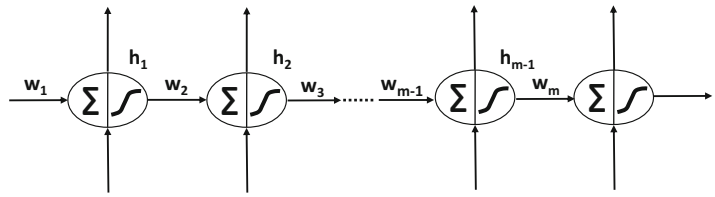


Figure 7.7: The vanishing and exploding gradient problems

The primary challenge associated with a recurrent neural network is that of the *vanishing* and *exploding gradient problems*. This point is explained in detail in Section 3.4 of Chapter 3. In this section, we will revisit this issue in the context of recurrent neural networks. It is easiest to understand the challenges associated with recurrent networks by examining the case of a recurrent network with a single unit in each layer.

Consider a set of T consecutive layers, in which the tanh activation function, $\Phi(\cdot)$, is applied between each pair of layers. The shared weight between a pair of hidden nodes is denoted by w. Let $h_1 \ldots h_T$ be the hidden values in the various layers. Let $\Phi'(h_t)$ be the derivative of the activation function in hidden layer t. Let the copy of the shared weight w in the tth layer be denoted by w_t so that it is possible to examine the effect of the backpropagation update. Let $\frac{\partial L}{\partial h_t}$ be the derivative of the loss function with respect to the hidden activation h_t. The neural architecture is illustrated in Figure 7.7. Then, one derives the following update equations using backpropagation:

$$\frac{\partial L}{\partial h_t} = \Phi'(h_{t+1}) \cdot w_{t+1} \cdot \frac{\partial L}{\partial h_{t+1}} \tag{7.5}$$

Since the shared weights in different temporal layers are the same, the gradient is multiplied with the same quantity $w_t = w$ for each layer. Such a multiplication will have a consistent bias towards vanishing when $w < 1$, and it will have a consistent bias towards exploding when $w > 1$. However, the choice of the activation function will also play a role because the derivative $\Phi'(h_{t+1})$ is included in the product. For example, the presence of the tanh

Note that $\overline{a}_t$ is a vector with as many components as the number of units in the hidden layer (which we have consistently denoted as p in this chapter). We compute the mean μ_t and standard σ_t of the pre-activation values in $\overline{a}_t$:

$$\mu_t = \frac{\sum_{i=1}^{p} a_{ti}}{p}, \qquad \sigma_t = \sqrt{\frac{\sum_{i=1}^{p} a_{ti}^2}{p} - \mu_t^2}$$

Here, a_{ti} denotes the ith component of the vector $\overline{a}_t$.

As in batch normalization, we have additional learning parameters, associated with each unit. Specifically, for the p units in the tth layer, we have a p-dimensional vector of *gain parameters* $\overline{\gamma}_t$, and a p-dimensional vector of *bias parameters* denoted by $\overline{\beta}_t$. These parameters are analogous to the parameters γ_i and β_i in Section 3.6 on batch normalization. The purpose of these parameters is to re-scale the normalized values and add bias in a learnable way. The hidden activations $\overline{h}_t$ of the next layer are therefore computed as follows:

$$\overline{h}_t = \tanh\left(\frac{\overline{\gamma}_t}{\sigma_t} \odot (\overline{a}_t - \overline{\mu}_t) + \overline{\beta}_t\right) \tag{7.6}$$

Here, the notation $\odot$ indicates elementwise multiplication, and the notation $\overline{\mu}_t$ refers to a vector containing p copies of the scalar μ_t. The effect of layer normalization is to ensure that the magnitudes of the activations do not continuously increase or decrease with time-stamp (causing vanishing and exploding gradients), although the learnable parameters allow some flexibility. It has been shown in [14] that layer normalization provides better performance than batch normalization in recurrent neural networks. Some related normalizations can also be used for streaming and online learning [294].

7.4 Echo-State Networks

Echo-state networks represent a simplification of recurrent neural networks. They work well when the dimensionality of the input is small; this is because echo-state networks scale well with the number of temporal units but not with the dimensionality of the input. Therefore, these networks would be a solid option for regression-based modeling of a single or small number of real-valued time series over a relatively long time horizon. However, they would be a poor choice for modeling text in which the input dimensionality (based on one-hot encoding) would be the size of the lexicon in which the documents are represented. Nevertheless, even in this case, echo-state networks are practically useful in the initialization of weights within the network. Echo-state networks are also referred to as *liquid-state machines* [304], except that the latter uses spiking neurons with binary outputs, whereas echo-state networks use conventional activations like the sigmoid and the tanh functions.

Echo-state networks use *random weights* in the hidden-to-hidden layer and even the input-to-hidden layer, although the dimensionality of the hidden states is almost always much larger than the dimensionality of input states. For a single input series, it is not uncommon to use hidden states of dimensionality about 200. Therefore, only the output layer is trained, which is typically done with a linear layer for real-valued outputs. Note that the training of the output layer simply aggregates the errors at different output nodes, although the weights at different output nodes are still shared. Nevertheless, the objective function would still evaluate to a case of linear regression, which can be trained very simply without the need for backpropagation. Therefore, the training of the echo-state network is very fast.

As in traditional recurrent networks, the hidden-to-hidden layers have nonlinear activations such as the logistic sigmoid function, although tanh activations are also possible. A very important caveat in the initialization of the hidden-to-hidden units is that the largest eigenvector of the weight matrix W_{hh} should be set to 1. This can be easily achieved by first sampling the weights of the matrix W_{hh} randomly from a standard normal distribution, and then dividing each entry by the largest absolute eigenvalue $|\lambda_{max}|$ of this matrix.

$$W_{hh} \Leftarrow W_{hh}/|\lambda_{max}| \tag{7.7}$$

After this normalization, the largest eigenvalue of this matrix will be 1, which corresponds to its *spectral radius*. However, using a spectral radius of 1 can be too conservative because the nonlinear activations will have a dampening effect on the values of the states. For example, when using the sigmoid activation, the *largest* possible partial derivative of the sigmoid is always 0.25, and therefore using a spectral radius much larger than 4 (say, 10) is okay. When using the tanh activation function it would make sense to have a spectral radius of about 2 or 3. These choices would often still lead to a certain level of dampening over time, which is actually a useful regularization because very long-term relationships are generally much weaker than short-term relationships in time-series. One can also tune the spectral radius based on performance by trying different values of the scaling factor γ on held-out data to set $W_{hh} = \gamma W_0$. Here, W_0 is a randomly initialized matrix.

It is recommended to use sparse connectivity in the hidden-to-hidden connections, which is not uncommon in settings involving transformations with random projections. In order to achieve this goal, a number of connections in W_{hh} can be sampled to be non-zero and others are set to 0. This number of connections is typically linear in the number of hidden units. Another key trick is to divide the hidden units into groups indexed $1 \ldots K$ and only allow connectivity between hidden states belonging to with the same index. Such an approach can be shown to be equivalent to training an ensemble of echo-state networks (see. Exercise 2).

Another issue is about setting the input-to-hidden matrices W_{xh}. One needs to be careful about the scaling of this matrix as well, or else the effect of the inputs in each time-stamp can seriously damage the information carried in the hidden states from the previous time-stamp. Therefore, the matrix W_{xh} is first chosen randomly to W_1, and then it is scaled with different values of the hyper-parameter β in order to determine the final matrix $W_{xh} = \beta W_1$ that gives the best accuracy on held-out data.

The core of the echo-state network is based on a very old idea that expanding the number of features of a data set with a nonlinear transformation can often increase the expressive power of the input representation. For example, the RBF network (cf. Chapter 5) and the kernel support-vector machine both gain their power from expansion of the underlying feature space according to Cover's theorem on separability of patterns [84]. The only difference is that the echo-state network performs the feature expansion with random projection; such an approach is not without precedent because various types of random transformations are also used in machine learning as fast alternatives to kernel methods [385, 516]. It is noteworthy that feature expansion is primarily effective through nonlinear transformations, and these are provided through the activations in the hidden layers. In a sense, the echo-state method works using a similar principle to the RBF network in the temporal domain, just as the recurrent neural network is the replacement of feed-forward networks in the temporal domain. Just as the RBF network uses very little training for extracting the hidden features, the echo-state network uses little training for extracting the hidden features and instead relies on the randomized expansion of the feature space.

When used on time-series data, the approach provides excellent results on predicting values far out in the future. The key trick is to choose target output values at a time-stamp

t that correspond to the time-series input values at $t+k$, where k is the lookahead required for forecasting. In other words, an echo-state network is an excellent nonlinear autoregressive technique for modeling time-series data. One can even use this approach for forecasting multivariate time-series, although it is inadvisable to use the approach when the number of time series is very large. This is because the dimensionality of hidden states required for modeling would be simply too large. A detailed discussion on the application of the echo-state network for time-series modeling is provided in Section 7.7.5. A comparison with respect to traditional time-series forecasting models is also provided in the same section.

Although the approach cannot be realistically used for very high-dimensional inputs (like text), it is still very useful for initialization [478]. The basic idea is to initialize the recurrent network by using its echo-state variant to train the output layer. Furthermore, a proper scaling of the initialized values W_{hh} and W_{xh} can be set by trying different values of the scaling factors β and γ (as discussed above). Subsequently, traditional backpropagation is used to train the recurrent network. This approach can be viewed as a lightweight pretraining for recurrent networks.

A final issue is about the sparsity of the weight connections. Should the matrix W_{hh} be sparse? This is generally a matter of some controversy and disagreement; while sparse connectivity of echo-state networks has been recommended since the early years [219], the reasons for doing so are not very clear. The original work [219] states that sparse connectivity leads to a decoupling of the individual subnetworks, which encourages the development of individual dynamics. This seems to be an argument for increased diversity of the features learned by the echo-state network. If decoupling is indeed the goal, it would make a lot more sense to do so explicitly, and divide the hidden states into disconnected groups. Such an approach has an ensemble-centric interpretation. It is also often recommended to increase sparsity in methods involving random projections for improved efficiency of the computations. Having dense connections can cause the activations of different states to be embedded in the multiplicative noise of a large number of Gaussian random variables, and therefore more difficult to extract.

7.5 Long Short-Term Memory (LSTM)

As discussed in Section 7.3, recurrent neural networks have problems associated with vanishing and exploding gradients [205, 368, 369]. This is a common problem in neural network updates where successive multiplication by the matrix $W^{(k)}$ is inherently unstable; it either results in the gradient disappearing during backpropagation, or in it blowing up to large values in an unstable way. This type of instability is the direct result of successive multiplication with the (recurrent) weight matrix at various time-stamps. One way of viewing this problem is that a neural network that uses only multiplicative updates is good only at learning over short sequences, and is therefore inherently endowed with good short-term memory but poor long-term memory [205]. To address this problem, a solution is to change the recurrence equation for the hidden vector with the use of the LSTM with the use of long-term memory. The operations of the LSTM are designed to have fine-grained control over the data written into this long-term memory.

As in the previous sections, the notation $\overline{h}_t^{(k)}$ represents the hidden states of the kth layer of a multi-layer LSTM. For notational convenience, we also assume that the input layer $\overline{x}_t$ can be denoted by $\overline{h}_t^{(0)}$ (although this layer is obviously not hidden). As in the case of the recurrent network, the input vector $\overline{x}_t$ is d-dimensional, whereas the hidden states are p-dimensional. The LSTM is an enhancement of the recurrent neural network architecture

of Figure 7.6 in which we change the recurrence conditions of how the hidden states $\overline{h}_t^{(k)}$ are propagated. In order to achieve this goal, we have an additional hidden vector of p dimensions, which is denoted by $\overline{c}_t^{(k)}$ and referred to as the *cell state*. One can view the cell state as a kind of long-term memory that retains at least a part of the information in earlier states by using a combination of partial "forgetting" and "increment" operations on the previous cell states. It has been shown in [233] that the nature of the memory in $\overline{c}_t^{(k)}$ is occasionally interpretable when it is applied to text data such as literary pieces. For example, one of the p values in $\overline{c}_t^{(k)}$ might change in sign after an opening quotation and then revert back only when that quotation is closed. The upshot of this phenomenon is that the resulting neural network is able to model long-range dependencies in the language or even a specific pattern (like a quotation) extended over a large number of tokens. This is achieved by using a gentle approach to update these cell states over time, so that there is greater persistence in information storage. Persistence in state values avoids the kind of instability that occurs in the case of the vanishing and exploding gradient problems. One way of understanding this intuitively is that if the states in different temporal layers share a greater level of similarity (through long-term memory), it is harder for the gradients with respect to the incoming weights to be drastically different.

As with the multilayer recurrent network, the update matrix is denoted by $W^{(k)}$ and is used to premultiply the column vector $[\overline{h}_t^{(k-1)}, \overline{h}_{t-1}^{(k)}]^T$. However, this matrix is of size[2] $4p \times 2p$, and therefore pre-multiplying a vector of size $2p$ with $W^{(k)}$ results in a vector of size $4p$. In this case, the updates use four intermediate, p-dimensional vector variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$ that correspond to the $4p$-dimensional vector. The intermediate variables $\overline{i}$, $\overline{f}$, and $\overline{o}$ are respectively referred to as *input*, *forget*, and *output* variables, because of the roles they play in updating the cell states and hidden states. The determination of the hidden state vector $\overline{h}_t^{(k)}$ and the cell state vector $\overline{c}_t^{(k)}$ uses a multi-step process of first computing these intermediate variables and then computing the hidden variables from these intermediate variables. Note the difference between intermediate variable vector $\overline{c}$ and primary cell state $\overline{c}_t^{(k)}$, which have completely different roles. The updates are as follows:

$$\begin{array}{l}\text{Input Gate:}\\ \text{Forget Gate:}\\ \text{Output Gate:}\\ \text{New C.-State:}\end{array} \begin{bmatrix} \overline{i} \\ \overline{f} \\ \overline{o} \\ \overline{c} \end{bmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix} \quad \textbf{[Setting up intermediates]}$$

$$\overline{c}_t^{(k)} = \overline{f} \odot \overline{c}_{t-1}^{(k)} + \overline{i} \odot \overline{c} \quad \textbf{[Selectively forget and add to long-term memory]}$$

$$\overline{h}_t^{(k)} = \overline{o} \odot \tanh(\overline{c}_t^{(k)}) \quad \textbf{[Selectively leak long-term memory to hidden state]}$$

[2] In the first layer, the matrix $W^{(1)}$ is of size $4p \times (p + d)$ because it is multiplied with a vector of size $(p + d)$.

Here, the element-wise product of vectors is denoted by "$\odot$," and the notation "sigm" denotes a sigmoid operation. For the very first layer (i.e., $k = 1$), the notation $\overline{h}_t^{(k-1)}$ in the above equation should be replaced with $\overline{x}_t$ and the matrix $W^{(1)}$ is of size $4p \times (p + d)$. In practical implementations, biases are also used[3] in the above updates, although they are omitted here for simplicity. The aforementioned update seems rather cryptic, and therefore it requires further explanation.

The first step in the above sequence of equations is to set up the intermediate variable vectors $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$, of which the first three should *conceptually* be considered binary values, although they are continuous values in $(0, 1)$. Multiplying a pair of binary values is like using an AND gate on a pair of boolean values. We will henceforth refer to this operation as gating. The vectors $\overline{i}$, $\overline{f}$, and $\overline{o}$ are referred to as input, forget, and output gates. In particular, these vectors are conceptually used as boolean gates for deciding (i) whether to add to a cell-state, (ii) whether to forget a cell state, and (iii) whether to allow leakage into a hidden state from a cell state. The use of the binary abstraction for the input, forget, and output variables helps in understanding the types of decisions being made by the updates. In practice, a continuous value in $(0, 1)$ is contained in these variables, which can enforce the effect of the binary gate in a probabilistic way if the output is seen as a probability. In the neural network setting, it is essential to work with continuous functions in order to ensure the differentiability required for gradient updates. The vector $\overline{c}$ contains the newly proposed contents of the cell state, although the input and forget gates regulate how much it is allowed to change the previous cell state (to retain long-term memory).

The four intermediate variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$, are set up using the weight matrices $W^{(k)}$ for the kth layer in the first equation above. Let us now examine the second equation that updates the cell state with the use of some of these intermediate variables:

$$\overline{c}_t^{(k)} = \underbrace{\overline{f} \odot \overline{c}_{t-1}^{(k)}}_{\text{Reset?}} + \underbrace{\overline{i} \odot \overline{c}}_{\text{Increment?}}$$

This equation has two parts. The first part uses the p forget bits in $\overline{f}$ to decide which of the p cell states from the previous time-stamp to reset[4] to 0, and it uses the p input bits in $\overline{i}$ to decide whether to add the corresponding components from $\overline{c}$ to each of the cell states. Note that such updates of the cell states are in additive form, which is helpful in avoiding the vanishing gradient problem caused by multiplicative updates. One can view the cell-state vector as a continuously updated long-term memory, where the forget and input bits respectively decide (i) whether to reset the cell states from the previous time-stamp and forget the past, and (ii) whether to increment the cell states from the previous time-stamp to incorporate new information into long-term memory from the current word. The vector $\overline{c}$ contains the p amounts with which to increment the cell states, and these are values in $[-1, +1]$ because they are all outputs of the tanh function.

[3]The bias associated with the forget gates is particularly important. The bias of the forget gate is generally initialized to values greater than 1 [228] because it seems to avoid the vanishing gradient problem at initialization.

[4]Here, we are treating the forget bits as a vector of binary bits, although it contains continuous values in $(0, 1)$, which can be viewed as probabilities. As discussed earlier, the binary abstraction helps us understand the conceptual nature of the operations.

Finally, the hidden states $\overline{h}_t^{(k)}$ are updated using leakages from the cell state. The hidden state is updated as follows:

$$\overline{h}_t^{(k)} = \underbrace{\overline{o} \odot \tanh(\overline{c}_t^{(k)})}_{\text{Leak } \overline{c}_t^{(k)} \text{ to } \overline{h}_t^{(k)}}$$

Here, we are copying a functional form of each of the p cell states into each of the p hidden states, depending on whether the output gate (defined by $\overline{o}$) is 0 or 1. Of course, in the continuous setting of neural networks, partial gating occurs and only a fraction of the signal is copied from each cell state to the corresponding hidden state. It is noteworthy that the final equation does not always use the tanh activation function. The following alternative update may be used:

$$\overline{h}_t^{(k)} = \overline{o} \odot \overline{c}_t^{(k)}$$

As in the case of all neural networks, the backpropagation algorithm is used for training purposes.

In order to understand why LSTMs provide better gradient flows than vanilla RNNs, let us examine the update for a simple LSTM with a single layer and $p = 1$. In such a case, the cell update can be simplified to the following:

$$c_t = c_{t-1} * f + i * c \tag{7.8}$$

Therefore, the partial derivative c_t with respect to c_{t-1} is f, which means that the backward gradient flows for c_t are multiplied with the value of the forget gate f. Because of elementwise operations, this result generalizes to arbitrary values of the state dimensionality p. The biases of the forget gates are often set to high values initially, so that the gradient flows decay relatively slowly. The forget gate f can also be different at different time-stamps, which reduces the propensity of the vanishing gradient problem. The hidden states can be expressed in terms of the cell states as $h_t = o * \tanh(c_t)$, so that one can compute the partial derivative with respect to h_t with the use of a single tanh derivative. In other words, the long-term cell states function as gradient super-highways, which leak into hidden states.

7.6 Gated Recurrent Units (GRUs)

The Gated Recurrent Unit (GRU) can be viewed as a simplification of the LSTM, which does not use explicit cell states. Another difference is that the LSTM directly controls the amount of information changed in the hidden state using separate forget and output gates. On the other hand, a GRU uses a single reset gate to achieve the same goal. However, the basic idea in the GRU is quite similar to that of an LSTM, in terms of how it partially resets the hidden states. As in the previous sections, the notation $\overline{h}_t^{(k)}$ represents the hidden states of the kth layer for $k \geq 1$. For notational convenience, we also assume that the input layer $\overline{x}_t$ can be denoted by $\overline{h}_t^{(0)}$ (although this layer is obviously not hidden). As in the case of LSTM, we assume that the input vector $\overline{x}_t$ is d-dimensional, whereas the hidden states are p-dimensional. The sizes of the transformation matrices in the first layer are accordingly adjusted to account for this fact.

In the case of the GRU, we use two matrices $W^{(k)}$ and $V^{(k)}$ of sizes[5] $2p \times 2p$ and $p \times 2p$, respectively. Pre-multiplying a vector of size $2p$ with $W^{(k)}$ results in a vector of size $2p$, which will be passed through the sigmoid activation to create two intermediate, p-dimensional vector variables $\overline{z}_t$ and $\overline{r}_t$, respectively. The intermediate variables $\overline{z}_t$ and $\overline{r}_t$ are respectively referred to as update and reset gates. The determination of the hidden state vector $\overline{h}_t^{(k)}$ uses a two-step process of first computing these gates, then using them to decide how much to change the hidden vector with the weight matrix $V^{(k)}$:

$$\begin{array}{l}\text{Update Gate:}\\ \text{Reset Gate:}\end{array} \left[\begin{array}{c}\overline{z}\\ \overline{r}\end{array}\right] = \left(\begin{array}{c}\text{sigm}\\ \text{sigm}\end{array}\right) W^{(k)} \left[\begin{array}{c}\overline{h}_t^{(k-1)}\\ \overline{h}_{t-1}^{(k)}\end{array}\right] \quad \textbf{[Set up gates]}$$

$$\overline{h}_t^{(k)} = \overline{z} \odot \overline{h}_{t-1}^{(k)} + (1 - \overline{z}) \odot \tanh\, V^{(k)} \left[\begin{array}{c}\overline{h}_t^{(k-1)}\\ \overline{r} \odot \overline{h}_{t-1}^{(k)}\end{array}\right] \quad \textbf{[Update hidden state]}$$

Here, the element-wise product of vectors is denoted by "$\odot$," and the notation "sigm" denotes a sigmoid operation. For the very first layer (i.e., $k = 1$), the notation $\overline{h}_t^{(k-1)}$ in the above equation should be replaced with $\overline{x}_t$. Furthermore, the matrices $W^{(1)}$ and $V^{(1)}$ are of sizes $2p \times (p + d)$ and $p \times (p + d)$, respectively. We have also omitted the mention of biases here, but they are usually included in practical implementations. In the following, we provide a further explanation of these updates and contrast them with those of the LSTM.

Just as the LSTM uses input, output, and forget gates to decide how much of the information from the previous time-stamp to carry over to the next step, the GRU uses the update and the reset gates. The GRU does not have a separate internal memory and also requires fewer gates to perform the update from one hidden state to another. Therefore, a natural question arises about the precise role of the update and reset gates. The reset gate $\overline{r}$ decides how much of the hidden state to carry over from the previous time-stamp for a matrix-based update (like a recurrent neural network). The update gate $\overline{z}$ decides the *relative* strength of the contributions of this matrix-based update and a more direct contribution from the hidden vector $\overline{h}_{t-1}^{(k)}$ at the previous time-stamp. By allowing a direct (partial) copy of the hidden states from the previous layer, the gradient flow becomes more stable during backpropagation. The update gate of the GRU simultaneously performs the role of the input and forget gates in the LSTM in the form of $\overline{z}$ and $1 - \overline{z}$, respectively. However, the mapping between the GRU and the LSTM is not precise, because it performs these updates directly on the hidden state (and there is no cell state). Like the input, output, and forget gates in the LSTM, the update and reset gates are intermediate "scratch-pad" variables.

In order to understand why GRUs provide better performance than vanilla RNNs, let us examine a GRU with a single layer and single state dimensionality $p = 1$. In such a case, the update equation of the GRU can be written as follows:

$$h_t = z \cdot h_{t-1} + (1 - z) \cdot \tanh[v_1 \cdot x_t + v_2 \cdot r \cdot h_{t-1}] \tag{7.9}$$

Note that layer superscripts are missing in this single-layer case. Here, v_1 and v_2 are the two elements of the 2×1 matrix V. Then, it is easy to see the following:

$$\frac{\partial h_t}{\partial h_{t-1}} = z + (\text{Additive Terms}) \tag{7.10}$$

[5]In the first layer ($k = 1$), these matrices are of sizes $2p \times (p + d)$ and $p \times (p + d)$.

Backward gradient flow is multiplied with this factor. Here, the term $z \in (0, 1)$ helps in passing *unimpeded* gradient flow and makes computations more stable. Furthermore, since the additive terms heavily depend on $(1-z)$, the overall multiplicative factor that tends to be closer to 1 even when z is small. Another point is that the value of z and the multiplicative factor $\frac{\partial h_t}{\partial h_{t-1}}$ is *different* for each time stamp, which tends to reduce the propensity for vanishing or exploding gradients.

Although the GRU is a closely related simplification of the LSTM, it should not be seen as a special case of the LSTM. A comparison of the LSTM and the GRU is provided in [71, 228]. The two models are shown to be roughly similar in performance, and the relative performance seems to depend on the task at hand. The GRU is simpler and enjoys the advantage of greater ease of implementation and efficiency. It might generalize slightly better with less data because of a smaller parameter footprint [71], although the LSTM would be preferable with an increased amount of data. The work in [228] also discusses several practical implementation issues associated with the LSTM. The LSTM has been more extensively tested than the GRU, simply because it is an older architecture and enjoys widespread popularity. As a result, it is generally seen as a safer option, particularly when working with longer sequences and larger data sets. The work in [160] also showed that none of the variants of the LSTM can reliably outperform it in a consistent way. This is because of the explicit internal memory and the greater gate-centric control in updating the LSTM.

7.7 Applications of Recurrent Neural Networks

Recurrent neural networks have numerous applications in machine learning applications, which are associated with information retrieval, speech recognition, and handwriting recognition. Text data forms the predominant setting for applications of RNNs, although there are several applications to computational biology as well. Most of the applications of RNNs fall into one of two categories:

1. *Conditional language modeling:* When the output of a recurrent network is a language model, one can enhance it with context in order to provide a relevant output to the context. In most of these cases, the context is the neural output of another neural network. To provide one example, in image captioning the context is the neural representation of an image provided by a convolutional network, and the language model provides a caption for the image. In machine translation, the context is the representation of a sentence in a source language (produced by another RNN), and the language model in the target language provides a translation.

2. *Leveraging token-specific outputs:* The outputs at the different tokens can be used to learn other properties than a language model. For example, the labels output at different time-stamps might correspond to the properties of the tokens (such as their parts of speech). In handwriting recognition, the labels might correspond to the characters. In some cases, all the time-stamps might not have an output, but the end-of-sentence marker might output a label for the entire sentence. This approach is referred to as sentence-level classification, and is often used in sentiment analysis. In some of these applications, bidirectional recurrent networks are used because the context on both sides of a word is helpful.

The following material will provide an overview of the numerous applications of recurrent neural networks. In most of these cases, we will use a single-layer recurrent network for ease in explanation and pictorial illustration. However, in most cases, a multi-layer LSTM is used. In other cases, a bidirectional LSTM is used, because it provides better performance. Replacing a single-layer RNN with a multi-layer/bidirectional LSTM in any of the following applications is straightforward. Out broader goal is to illustrate how this *family* of architectures can be used in these settings.

7.7.1 Application to Automatic Image Captioning

In image captioning, the training data consists of image-caption pairs. For example, the image[6] in the left-hand side of Figure 7.9 is obtained from the National Aeronautics and Space Administration Web site. This image is captioned "*cosmic winter wonderland.*" One might have hundreds of thousands of such image-caption pairs. These pairs are used to train the weights in the neural network. Once the training has been completed, the captions are predicted for unknown test instances. Therefore, one can view this approach as an instance of image-to-sequence learning.

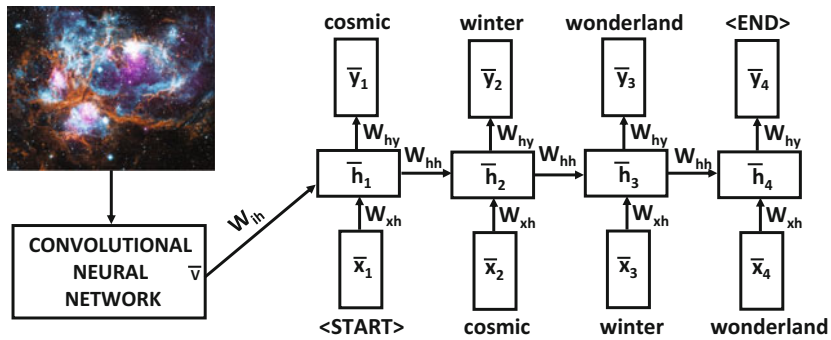


Figure 7.9: Example of image captioning with a recurrent neural network. An additional convolutional neural network is required for representational learning of the images. The image is represented by the vector $\overline{v}$, which is the output of the convolutional neural network. The inset image is by courtesy of the National Aeronautics and Space Administration (NASA).

One issue in the automatic captioning of images is that a separate neural network is required to learn the representation of the images. A common architecture to learn the representation of images is the *convolutional neural network*. A detailed discussion of convolutional neural networks is provide in Chapter 8. Consider a setting in which the convolutional neural network produces the q-dimensional vector $\overline{v}$ as the output representation. This vector is then used as an input to the neural network, but only[7] at the first time-stamp. To account for this additional input, we need another $p \times q$ matrix W_{ih}, which maps the image representation to the hidden layer. Therefore, the update equations for the various layers now need to be modified as follows:

$$\overline{h}_1 = \tanh(W_{xh}\overline{x}_1 + W_{ih}\overline{v})$$

[6] https://www.nasa.gov/mission_pages/chandra/cosmic-winter-wonderland.html

[7] In principle, one can also allow it to be input at all time-stamps, but it only seems to worsen performance.

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}) \quad \forall t \geq 2$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

An important point here is that the convolutional neural network and the recurrent neural network are not trained in isolation. Although one might train them in isolation in order to create an initialization, the final weights are always trained jointly by running each image through the network and matching up the predicted caption with the true caption. In other words, for each image-caption pair, the weights in both networks are updated when errors are made in predicting any particular token of the caption. In practice, the errors are soft because the tokens at each point are predicted probabilistically. Such an approach ensures that the learned representation $\overline{v}$ of the images is sensitive to the specific application of predicting captions.

After all the weights have been trained, a test image is input to the entire system and passed through both the convolutional and recurrent neural network. For the recurrent network, the input at the first time-stamp is the <START> token and the representation of the image. At later time-stamps, the input is the most likely token predicted at the previous time-stamp. One can also use beam search to keep track of the b most likely sequence prefixes to expand on at each point. This approach is not very different from the language generation approach discussed in Section 7.2.1.1, except that it is conditioned on the image representation that is input to the model in the first time-stamp of the recurrent network. This results in the prediction of a relevant caption for the image.

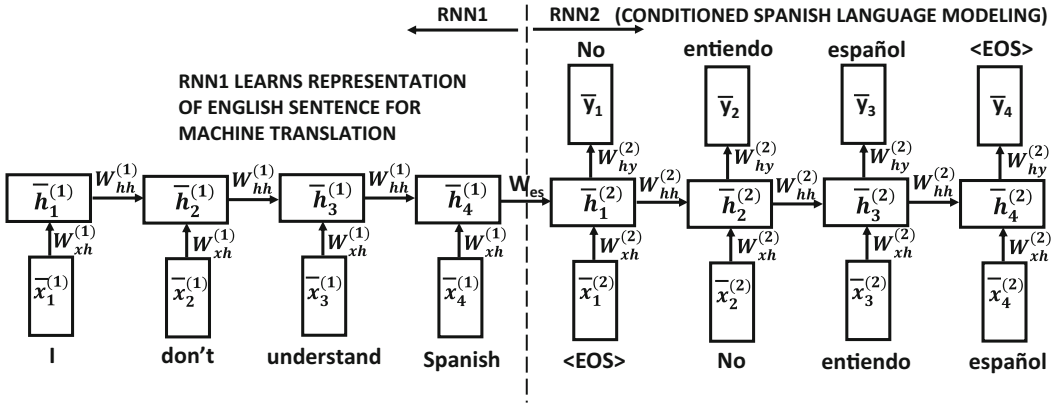


Figure 7.10: Machine translation with recurrent neural networks. Note that there are two separate recurrent networks with their own sets of shared weights. The output of $\overline{h}_4^{(1)}$ is a fixed length encoding of the 4-word English sentence.

7.7.2 Sequence-to-Sequence Learning and Machine Translation

Just as one can put together a convolutional neural network and a recurrent neural network to perform image captioning, one can put together two recurrent networks to translate one language into another. Such methods are also referred to as *sequence-to-sequence* learning because a sequence in one language is mapped to a sequence in another language. In principle, sequence-to-sequence learning can have applications beyond machine translation. For example, even question-answering (QA) systems can be viewed as sequence-to-sequence learning applications.

In the following, we provide a simple solution to machine translation with recurrent neural networks, although such applications are rarely addressed directly with the simple forms of recurrent neural networks. Rather, a variation of the recurrent neural network, referred to as the long short-term memory (LSTM) model is used. Such a model is much better in learning long-term dependencies, and can therefore work well with longer sentences. Since the general approach of using an RNN applies to an LSTM as well, we will provide the discussion of machine translation with the (simple) RNN. A discussion of the LSTM is provided in Section 7.5, and the generalization of the machine translation application to the LSTM is straightforward.

In the machine translation application, two different RNNs are hooked end-to-end, just as a convolutional neural network and a recurrent neural network are hooked together for image captioning. The first recurrent network uses the words from the source language as input. No outputs are produced at these time-stamps and the successive time-stamps accumulate knowledge about the source sentence in the hidden state. Subsequently, the end-of-sentence symbol is encountered, and the second recurrent network starts by outputting the first word of the target language. The next set of states in the second recurrent network output the words of the sentence in the target language one by one. These states also use the words of the target language as input, which is available for the case of the training instances but not for test instances (where predicted values are used instead). This architecture is shown in Figure 7.10.

The architecture of Figure 7.10 is similar to that of an autoencoder, and can even be used with pairs of identical sentences in the same language to create fixed-length representations of sentences. The two recurrent networks are denoted by RNN1 and RNN2, and their weights are not the same. For example, the weight matrix between two hidden nodes at successive time-stamps in RNN1 is denoted by $W_{hh}^{(1)}$, whereas the corresponding weight matrix in RNN2 is denoted by $W_{hh}^{(2)}$. The weight matrix W_{es} of the link joining the two neural networks is special, and can be independent of either of the two networks. This is necessary if the sizes of the hidden vectors in the two RNNs are different because the dimensions of the matrix W_{es} will be different from those of both $W_{hh}^{(1)}$ and $W_{hh}^{(2)}$. As a simplification, one can use[8] the same size of the hidden vector in both networks, and set $W_{es} = W_{hh}^{(1)}$. The weights in RNN1 are devoted to learning an encoding of the input in the source language, and the weights in RNN2 are devoted to using this encoding in order to create an output sentence in the target language. One can view this architecture in a similar way to the image captioning application, except that we are using two recurrent networks instead of a convolutional-recurrent pair. The output of the final hidden node of RNN1 is a fixed-length encoding of the source sentence. Therefore, irrespective of the length of the sentence, the encoding of the source sentence depends on the dimensionality of the hidden representation.

The grammar and length of the sentence in the source and target languages may not be the same. In order to provide a grammatically correct output in the target language, RNN2 needs to learn its language model. It is noteworthy that the units in RNN2 associated with the target language have both inputs and outputs arranged in the same way as a language-modeling RNN. At the same time, the output of RNN2 is conditioned on the input it receives from RNN1, which effectively causes language translation. In order to achieve this goal, training pairs in the source and target languages are used. The approach passes the source-target pairs through the architecture of Figure 7.10 and learns the model parameters

[8]The original work in [478] seems to use this option. In the Google Neural Machine Translation system [579], this weight is removed. This system is now used in Google Translate.

with the use of the backpropagation algorithm. Since only the nodes in RNN2 have outputs, only the errors made in predicting the target language words are backpropagated to train the weights in both neural networks. The two networks are jointly trained, and therefore the weights in both networks are optimized to the errors in the translated outputs of RNN2. As a practical matter, this means that the internal representation of the source language learned by RNN1 is highly optimized to the machine translation application, and is very different from one that would be learned if one had used RNN1 to perform language modeling of the source sentence. After the parameters have been learned, a sentence in the source language is translated by first running it through RNN1 to provide the necessary input to RNN2. Aside from this contextual input, another input to the first unit of RNN2 is the <EOS> tag, which causes RNN2 to output the likelihoods of the first token in the target language. The most likely token using beam search (cf. Section 7.2.1.1) is selected and used as the input to the recurrent network unit in the next time-stamp. This process is recursively applied until the output of a unit in RNN2 is also <EOS>. As in Section 7.2.1.1, we are generating a sentence from the target language using a language-modeling approach, except that the specific output is conditioned on the internal representation of the source sentence.

The use of neural networks for machine translation is relatively recent. Recurrent neural network models have a sophistication that greatly exceeds that of traditional machine translation models. The latter class of methods uses phrase-centric machine learning, which is often not sophisticated enough to learn the subtle differences between the grammars of the two languages. In practice, deep models with multiple layers are used to improve the performance.

One weakness of such translation models is that they tend to work poorly when the sentences are long. Numerous solutions have been proposed to solve the problem. A recent solution is that the sentence in the source language is input in the *opposite order* [478]. This approach brings the first few words of the sentences in the two languages closer in terms of their time-stamps within the recurrent neural network architecture. As a result, the first few words in the target language are more likely to be predicted correctly. The correctness in predicting the first few words is also helpful in predicting the subsequent words, which are also dependent on a neural language model in the target language.

7.7.2.1 Question-Answering Systems

A natural application of sequence-to-sequence learning is that of question answering (QA). Question-answering systems are designed with different types of training data. In particular, two types of question-answering systems are common:

1. In the first type, the answers are directly inferred based on the phrases and clue words in the question.

2. In the second type, the question is first transformed into a database query, and is used to query a structured knowledge base of facts.

Sequence-to-sequence learning can be helpful in both settings. Consider the first setting, in which we have training data containing question-answer pairs like the following:

What is the capital of China? <EOQ> The capital is Beijing. <EOA>

These types of training pairs are not very different from those available in the case of machine translation, and the same techniques can be used in these cases. However, note that one key difference between machine translation and question-answering systems is that

there is a greater level of reasoning in the latter, which typically requires an understanding of the relationships between various entities (e.g., people, places, and organizations). This problem is related to the quintessential problem of *information extraction*. Since questions are often crafted around various types of named entities and relationships among them, information extraction methods are used in various ways. The utility of entities and information extraction is well known in answering "what/who/where/when" types of questions (e.g., entity-oriented search), because *named entities* are used to represent persons, locations, organizations, dates, and events, and *relationship extraction* provides information about the interactions among them. One can incorporate the meta-attributes about tokens, such as entity types, as additional inputs to the learning process. Specific examples of such input units are shown in Figure 7.12 of Section 7.7.4, although the figure is designed for the different application of token-level classification.

An important difference between question-answering and machine translation systems is that the latter is seeded with a large corpus of documents (e.g., a large knowledge base like Wikipedia). The query resolution process can be viewed as a kind of entity-oriented search. From the perspective of deep learning, an important challenge of QA systems is that a much larger capacity to store the knowledge is required than is typically available in recurrent neural networks. A deep learning architecture that works well in these settings is that of *memory networks* [528]. Question-answering systems pose many different settings in which the training data may be presented, and the ways in which various types of questions may be answered and evaluated. In this context, the work in [527] discusses a number of template tasks that can be useful for evaluating question-answering systems.

A somewhat different approach is to convert natural language questions into queries that are properly posed in terms of entity-oriented search. Unlike machine translation systems, question answering is often considered a multi-stage process in which understanding what is being asked (in terms of a properly represented query) is sometimes more difficult than answering the query itself. In such cases, the training pairs will correspond to the informal and formal representations of questions. For example, one might have a pair as follows:

$$\underbrace{\text{What is the capital of China? <EOQ1>}}_{\text{Natural language question}} \quad \underbrace{\textbf{CapitalOf}(\ \textit{China},\ ?)\ \text{<EOQ2>}}_{\text{Formal Representation}}$$

The expression on the right-hand side is a structured question, which queries for entities of different types such as persons, places, and organizations. The first step would be to convert the question into an internal representation like the one above, which is more prone to query answering. This conversion can be done using training pairs of questions and their internal representations in conjunction with an recurrent network. Once the question is understood as an entity-oriented search query, it can be posed to the indexed corpus, from which relevant relationships might already have been extracted up front. Therefore, the knowledge base is also preprocessed in such cases, and the question resolution boils down to matching the query with the extracted relations. It is noteworthy that this approach is limited by the complexity of the syntax in which questions are expressed, and the answers might also be simple one-word responses. Therefore, this type of approach is often used for more restricted domains. In some cases, one learns how to paraphrase questions by rewording a more complex question as a simpler question before creating the query representation [115, 118]:

$$\underbrace{\text{How can you tell if you have the flu? <EOQ1>}}_{\text{Complex question}} \quad \underbrace{\text{What are the signs of the flu? <EOQ2>}}_{\text{Paraphrased}}$$

The paraphrased question can be learned with sequence-to-sequence learning, although the work in [118] does not seem to use this approach. Subsequently, it is easier to convert the paraphrased question into a structured query. Another option is to provide the question in structured form to begin with. An example of a recurrent neural network that supports factoid question answering from QA training pairs is provided in [216]. However, unlike pure sequence-to-sequence learning, it uses the dependency parse trees of questions as the input representation. Therefore, a part of the formal understanding of the question is already encoded into the input.

7.7.3 Application to Sentence-Level Classification

In this problem, each sentence is treated as a training (or test) instance for classification purposes. Sentence-level classification is generally a more difficult problem than document-level classification because sentences are short, and there is often not enough evidence in the vector space representation to perform the classification accurately. However, the sequence-centric view is more powerful and can often be used to perform more accurate classification. The RNN architecture for sentence-level classification is shown in Figure 7.11. Note that the only difference from Figure 7.11(b) is that we no longer care about the outputs at each node but defer the class output to the end of the sentence. In other words, a single class label is predicted at the very last time-stamp of the sentence, and it is used to backpropagate the class prediction errors.

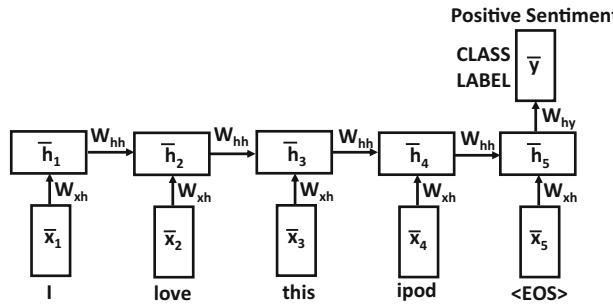


Figure 7.11: Example of sentence-level classification in a sentiment analysis application with the two classes "*positive sentiment*" and "*negative sentiment*."

Sentence-level classification is often leveraged in *sentiment analysis*. This problem attempts to discover how positive or negative users are about specific topics by analyzing the content of a sentence [6]. For example, one can use sentence-level classification to determine whether or not a sentence expresses a positive sentiment by treating the sentiment polarity as the class label. In the example shown in Figure 7.11, the sentence clearly indicates a positive sentiment. Note, however, that one cannot simply use a vector space representation containing the word "*love*" to infer the positive sentiment. For example, if words such as "*don't*" or "*hardly*" occur before "*love*", the sentiment would change from positive to negative. Such words are referred to as *contextual valence shifters* [377], and their effect can be modeled only in a sequence-centric setting. Recurrent neural networks can handle such settings because they use the accumulated evidence over the specific sequence of words in order to predict the class label. One can also combine this approach with linguistic features. In the next section, we show how to use linguistic features for token-level classification; similar ideas also apply to the case of sentence-level classification.

7.7.4 Token-Level Classification with Linguistic Features

The numerous applications of token-level classification include information extraction and text segmentation. In information extraction, specific words or combinations of words are identified that correspond to persons, places, or organizations. The linguistic features of the word (capitalization, part-of-speech, orthography) are more important in these applications than in typical language modeling or machine translation applications. Nevertheless, the methods discussed in this section for incorporating linguistic features can be used for any of the applications discussed in earlier sections. For the purpose of discussion, consider a *named-entity recognition application* in which every entity is to be classified as one of the categories corresponding to person (P), location (L), and other (O). In such cases, each token in the training data has one of these labels. An example of a possible training sentence is as follows:

$$\underbrace{\text{William}}_{P}\ \underbrace{\text{Jefferson}}_{P}\ \underbrace{\text{Clinton}}_{P}\ \underbrace{\text{lives}}_{O}\ \underbrace{\text{in}}_{O}\ \underbrace{\text{New}}_{L}\ \underbrace{\text{York}}_{L}.$$

In practice, the tagging scheme is often more complex because it encodes information about the beginning and end of a set of contiguous tokens with the same label. For test instances, the tagging information about the tokens is not available.

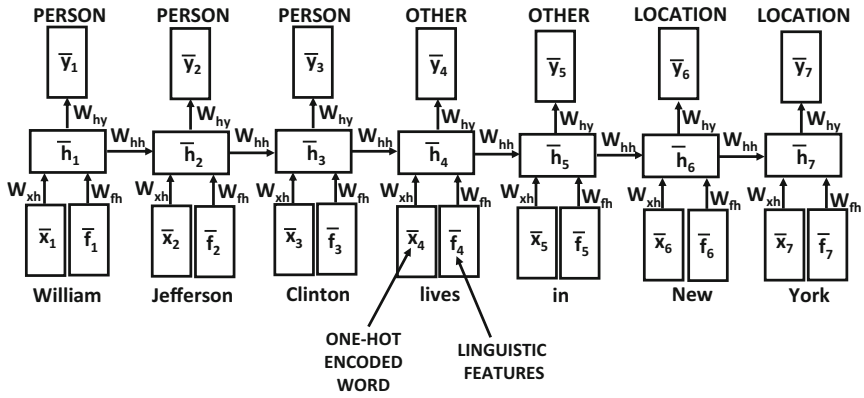


Figure 7.12: Token-wise classification with linguistic features

The recurrent neural network can be defined in a similar way as in the case of language modeling applications, except that the outputs are defined by the tags rather than the next set of words. The input at each time-stamp t is the one-hot encoding $\overline{x}_t$ of the token, and the output $\overline{y}_t$ is the tag. Furthermore, we have an additional set of q-dimensional linguistic features $\overline{f}_t$ associated with the tokens at time-stamp t. These linguistic features might encode information about the capitalization, orthography, capitalization, and so on. The hidden layer, therefore, receives two separate inputs from the tokens and from the linguistic features. The corresponding architecture is illustrated in Figure 7.12. We have an additional $p \times q$ matrix W_{fh} that maps the features $\overline{f}_t$ to the hidden layer. Then, the recurrence condition at each time-stamp t is as follows:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{fh}\overline{f}_t + W_{hh}\overline{h}_{t-1})$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

The main innovation here is in the use of an additional weight matrix for the linguistic features. The change in the type of output tag does not affect the overall model significantly.

In some variations, it might also be helpful to *concatenate* the linguistic and token-wise features into as separate *embedding layer*, rather than adding them. The work in [565] provides an example in the case of recommender systems, although the principle can also be applied here. The overall learning process is also not significantly different. In token-level classification applications, it is sometimes helpful to use bidirectional recurrent networks in which recurrence occurs in both temporal directions [434].

7.7.5 Time-Series Forecasting and Prediction

Recurrent neural networks present a natural choice for time-series forecasting and prediction. The main difference from text is that the input units are real-valued vectors rather than (discrete) one-hot encoded vectors. For real-valued prediction, the output layer always uses linear activations, rather than the softmax function. In the event that the output is a discrete value (e.g., identifier of a specific event), it is also possible to use discrete outputs with softmax activation. Although any of the variants of the recurrent neural network (e.g., LSTM or GRU) can be used, one of the common problems in time-series analysis is that such sequences can be extremely long. Even though the LSTM and the GRU provide a certain level of protection with increased time-series length, there are limitations to the performance. This is because LSTMs and GRUs do degrade for series beyond certain lengths. Many time-series can have a very large number of time-stamps with various types of short- and long-term dependencies. The prediction and forecasting problems present unique challenges in these cases.

However, a number of useful solutions exist, at least in cases where the number of time-series to be forecasted is not too large. The most effective method is the use of the echo-state network (cf. Section 7.4), in which it is possible to effectively forecast and predict both real-valued and discrete observations with a *small* number of time-series. The caveat that the number of inputs is small is an important one, because echo-state networks rely on randomized expansion of the feature space via the hidden units (see Section 7.4). If the number of original time series is too large, then it may not turn out to be practical to expand the dimensionality of the hidden space sufficiently to capture this type of feature engineering. It is noteworthy that the vast majority of forecasting models in the time-series literature are, in fact, univariate models. A classical example is the *autoregressive model* (AR), which uses the immediate window of history in order to perform forecasting.

The use of an echo-state network in order to perform time-series regression and forecasting is straightforward. At each time-stamp, the input is a vector of d values corresponding to the d different time series that are being modeled. It is assumed that the d time series are synchronized, and this is often accomplished by preprocessing and interpolation. The output at each time-stamp is the predicted value. In forecasting, the predicted value is simply the value(s) of the different time-series at k units ahead. One can view this approach as the time-series analog of language models with discrete sequences. It is also possible to choose an output corresponding to a time-series not present in the data (e.g., predicting one stock price from another) or to choose an output corresponding to a discrete event (e.g., equipment failure). The main differences among all these cases lie in the specific choice of the loss function for the output at hand. In the specific case of time-series forecasting, a neat relationship can be shown between autoregressive models and echo-state networks.

Relationship with Autoregressive Models

An *autoregressive model* models the values of a time-series as a linear function of its immediate history of length p. The p coefficients of this model are learned with linear regression. Echo-state networks can be shown to be closely related to autoregressive models, in which the connections of the hidden-to-hidden matrix are sampled in a particular way. The additional power of the echo-state network over an autoregressive model arises from the nonlinearity used in the hidden-to-hidden layer. In order to understand this point, we will consider the special case of an echo-state network in which its input corresponds to a single time series and the hidden-to-hidden layers have linear activations. Now imagine that we could somehow choose the hidden-to-hidden connections in such a way that the values of the hidden state in each time-stamp is exactly equal to the values of the time-series in the last p ticks. What kind of sampled weight matrix would achieve this goal?

First, the hidden state needs to have p units, and therefore the size of W_{hh} is $p \times p$. It is easy to show that a weight matrix W_{hh} that shifts the hidden state by one unit and copies the input value to the vacated state caused by the shifting will result in a hidden state, which is exactly the same as the last window of p points. In other words, the matrix W_{hh} will have exactly $(p-1)$ non-zero entries of the form $(i, i+1)$ for each $i \in \{1 \ldots p-1\}$. As a result, pre-multiplying any p-dimensional column vector $\overline{h}_t$ with W_{hh} will shift the entries of $\overline{h}_t$ by one unit. For a 1-dimensional time-series, the element x_t is a 1-dimensional input into the tth hidden state of the echo state network, and W_{xh} is therefore of size $p \times 1$. Setting only the entry $(p, 0)$ of W_{xh} to 1 and all other entries to 0 will result in copying x_t into the first element of $\overline{h}_t$. The matrix W_{hy} is a $1 \times p$ matrix of *learned weights*, so that $W_{hy}\overline{h}_t$ yields the prediction $\hat{y}_t$ of the observed value y_t. In autoregressive modeling, the value of y_t is simply set to x_{t+k} for some lookahead k, and the value of k is often set to 1. It is noteworthy that the matrices W_{hh} and W_{xh} are fixed, and only W_{hy} needs to learned. This process leads to the development of a model that is identical to the time-series autoregressive model [3].

The main difference of the time-series autoregressive model from the echo-state network is that the latter fixes W_{hh} and W_{xh} randomly, and uses much larger dimensionalities of the hidden states. Furthermore, nonlinear activations are used in the hidden units. As long as the spectral radius of W_{hh} is (slightly) less than 1, a random choice of the matrices W_{hh} and W_{xh} with linear activations can be viewed as a decay-based variant of the autoregressive model. This is because the matrix W_{hh} only performs a random (but slightly decaying) transformation of the previous hidden state. Using a decaying random projection of the previous hidden state intuitively achieves similar goals as a sliding window-shifted copy of the previous state. The precise spectral radius of W_{hh} governs the rate of decay. With a sufficient number of hidden states, the matrix W_{hy} provides enough degrees of freedom to model any decay-based function of recent history. Furthermore, the proper scaling of the W_{xh} ensures that the most recent entry is not given too much or too little weight. Note that echo-state networks do test different scalings of the matrix W_{xh} to ensure that the effect of this input does not wipe out the contributions from the hidden states. The nonlinear activations in the echo-state network give greater power to this approach over a time-series autoregressive model. In a sense, echo-state networks can model complex nonlinear dynamics of the time-series, unlike an off-the-shelf autoregressive model.

7.7.6 Temporal Recommender Systems

Several solutions [465, 534, 565] have been proposed in recent years for temporal modeling of recommender systems. Some of these methods use temporal aspects of users, whereas others use temporal aspects of users and items. One observation is that the properties of items tend to be more strongly fixed in time than the properties of users. Therefore, solutions that use the temporal modeling only at the user level are often sufficient. However, some methods [534] perform the temporal modeling both at the user level and at the item level.

In the following, we discuss a simplification of the model discussed in [465]. In temporal recommender systems, the time-stamps associated with user ratings are leveraged for the recommendation process. Consider a case in which the observed rating of user i for item j at time-stamp t is denoted by r_{ijt}. For simplicity, we assume that the time-stamp t is simply the index of the rating in the sequential order it was received (although many models use the wall-clock time). Therefore, the sequence being modeled by the RNN is a sequence of rating values associated with the content-centric representations of the users and items to which the rating belongs. Therefore, we want to model the value of the rating as a function of content-centric inputs at each time-stamp.

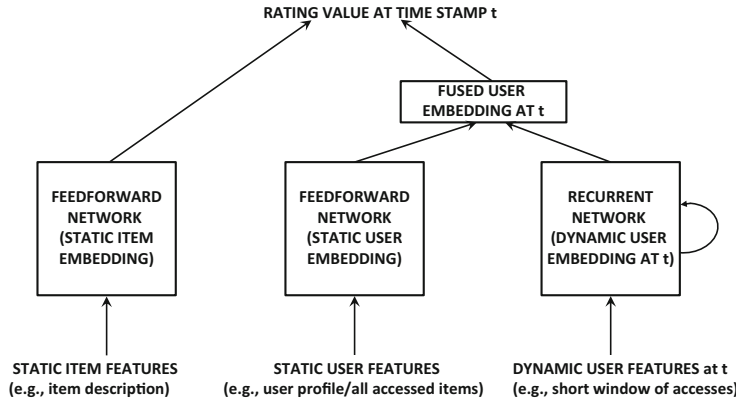


Figure 7.13: Recommendations with recurrent neural networks. At each time-stamp, the static/dynamic user features and static item features are input, and a rating value is output for that user-item combination.

We describe these content-centric representations below. The prediction of the rating r_{ijt} is assumed to be depend on (i) static features associated with the item, (ii) static features associated with the user, and (iii) the dynamic features associated with the user. The static features associated with the item might be item titles or descriptions, and one can create a bag-of-words representation of the item. The static features associated with the user might be a user-specific profile or a fixed history of accesses of this user, which does not change over the data set. The static features associated with the users are also typically represented as a bag of words, and one can even consider item-rating pairs as pseudo-keywords in order to combine user-specified keywords with ratings activity. In the case where ratings activity is used, a fixed history of accesses of the user is always leveraged for designing static features. The dynamic user features are more interesting because they are based on the dynamically changing user access history. In this case, a short history of item-rating pairs can be used as pseudo-keywords, and a bag-of-words representation can be created at time-stamp t.

In several cases, explicit ratings are not available, but implicit feedback data is available corresponding to a user clicking on an item. In the event that implicit feedback is used, negative sampling becomes necessary in which user-item pairs for which activity has not occurred are included in the sequence at random. This approach can be viewed as a hybrid between a content-based and collaborative recommendation approach. While it does use the user-item-rating triplets like a traditional recommender model, the content-centric representations of the users and items are input at each time-stamp. However, the inputs at different time-stamps correspond to different user-item pairs, and therefore the collaborative power of the patterns of ratings among different users and items is used as well.

The overall architecture of this recommender system is illustrated in Figure 7.13. It is evident that this architecture contains three different subnetworks to create feature embeddings out of static item features, static user features, and dynamic user features. The first two of these three are feed-forward networks, whereas the last of them is a recurrent neural network. First, the embeddings from the two user-centric networks are fused using either concatenation or element-wise multiplication. In the latter case, it is necessary to create embeddings of the same dimensionality for static and dynamic user features. Then, this fused user embedding at time-stamp t and the static item embedding is used to predict the rating at time-stamp t. For implicit feedback data, one can predict probabilities of positive activity for a particular user-item pair. The chosen loss function depends on the nature of the rating being predicted. The training algorithm needs to work with a consecutive sequence of training triplets (of some fixed mini-batch size) and backpropagate to the static and dynamic portions of the network simultaneously.

The aforementioned presentation has simplified several aspects of the training procedure presented in [465]. For example, it is assumed that a single rating is received at each time-stamp t, and that a fixed time-horizon is sufficient for temporal modeling. In reality, different settings might required different levels of granularity at which temporal aspects are handled. Therefore, the work in [465] proposes methods to address varying levels of granularity in the modeling process. It is also possible to perform the recommendation under a pure collaborative filtering regime without using content-centric features in any way. For example, it is possible[9] to adapt the recommender system discussed in Section 2.5.7 of Chapter 2 by using a recurrent neural network (cf. Exercise 3).

Another recent work [565] treats the problem as that of working with product-action-time triplets at an e-commerce site. The idea is that a site logs sequential actions performed by each user to various products, such as visiting a product page from a homepage, category page, or sales page, and that of actually buying the product. Each action has a *dwell time*, which indicates the amount of time that the user spends in performing that action. The dwell time is discretized into a set of intervals, which would be uniform or geometric, depending on the application at hand. It makes sense to discretize the time into geometrically increasing intervals.

One sequence is collected for each user, corresponding to the actions performed by the user. One can represent the rth element of the sequence as $(\overline{p}_r, \overline{a}_r, \overline{t}_r)$, where $\overline{p}_r$ is the one-hot encoded product, $\overline{a}_r$ is the one-hot encoded action, and $\overline{t}_r$ is the one-hot encoded discretized value of the time interval. Each of $\overline{p}_r$, $\overline{a}_r$, and $\overline{t}_r$ is a one-hot encoded vector. An embedding layer with weight matrices W_p, W_a, and W_t is used to create the representation $\overline{e}_r = (W_p\overline{p}_r, W_a\overline{a}_r, W_t\overline{t}_r)$. These matrices were pretrained with *word2vec* training applied to sequences extracted from the e-commerce site. Subsequently, the input to the recurrent

[9]Even though the adaptation from Section 2.5.7 is the most natural and obvious one, we have not seen it elsewhere in the literature. Therefore, it might be an interesting exercise for the reader to implement the adaptation of Exercise 3.

neural network is $\overline{e}_1 \ldots \overline{e}_T$, which was used to predict the outputs $\overline{o}_1 \ldots \overline{o}_T$. The output at the time-stamp t corresponds to the next action of the user at that time-stamp. Note that the embedding layer is also attached to the recurrent network, and it is fine-tuned during backpropagation (beyond its *word2vec* initialization). The original work [565] also adds an attention layer, although good results can be obtained even without this layer.

7.7.7 Secondary Protein Structure Prediction

In protein structure prediction, the elements of the sequence are the symbols representing one of the 20 amino acids. The 20 possible amino acids are akin to the vocabulary used in the text setting. Therefore, a one-hot encoding of the input is effective in these cases. Each position is associated with a class label corresponding to the secondary protein structure. This secondary structure can be either the alpha-helix, beta-sheet, or coil. Therefore, this problem can be reduced to token-level classification. A three-way softmax is used in the output layer. The work in [20] used a bidirectional recurrent neural network for prediction. This is because protein structure prediction is a problem that benefits from the context on both sides of a particular position. In general, the choice between using a uni-directional network and a bidirectional network is highly regulated by whether or not the prediction is causal to a historical segment or whether it depends on the context on both sides.

7.7.8 End-to-End Speech Recognition

In end-to-end speech recognition, one attempts to transcribe the raw audio files into character sequences while going through as few intermediate steps as possible. A small amount of preprocessing is still needed in order to make the data presentable as an input sequence. For example, the work in [157] presents the data as *spectrograms* derived from raw audio files using the *specgram* function of the *matplotlib* python toolkit. The width used was 254 Fourier windows with an overlap of 127 frames and 128 inputs per frame. The output is a character in the transcription sequence, which could include a character, a punctuation mark, a space, or even a null character. The label could be different depending on the application at hand. For example, the labels could be characters, phonemes, or musical notes. A bidirectional recurrent neural network is most appropriate to this setting, because the context on both sides of a character helps in improving accuracy.

One challenge associated with this type of setting is that we need the alignment between the frame representation of the audios and the transcription sequence. This type of alignment is not available a priori, and is in fact one of the outputs of the system. This leads to the problem of circular dependency between segmentation and recognition, which is also referred to as *Sayre's paradox*. This problem is solved with the use of *connectionist temporal classification*. In this approach, a dynamic programming algorithm [153] is combined with the (softmax) probabilistic outputs of the recurrent network in order to determine the alignment that maximizes the overall probability of generation. The reader is referred to [153, 157] for details.

7.7.9 Handwriting Recognition

A closely related application to speech recognition is that of handwriting recognition [154, 156]. In handwriting recognition, the input consists of a sequence of (x, y) coordinates, which represents the position of the tip of the pen at each time-stamp. The output corresponds to a sequence of characters written by the pen. These coordinates are then used to extract

further features such as a feature indicating whether the pen is touching the writing surface, the angles between nearby line segments, the velocity of the writing, and normalized values of the coordinates. The work in [154] extracts a total of 25 features. It is evident that multiple coordinates will create a character. However, it is hard to know exactly how many coordinates will create each character because it may vary significantly over the handwriting and style of different writers. Much like speech recognition, the issue of proper segmentation creates numerous challenges. This is the same Sayre's paradox that is encountered in speech recognition.

In unconstrained handwriting recognition, the handwriting contains a set of *strokes*, and by putting them together one can obtain characters. One possibility is to identify the strokes up front, and then use them to build characters. However, such an approach leads to inaccurate results, because the identification of stroke boundaries is an error-prone task. Since the errors tend to be additive over different phases, breaking up the task into separate stages is generally not a good idea. At a basic level, the task of handwriting recognition is no different from speech recognition. The only difference is in terms of the specific way in which the inputs and outputs are represented. As in the case of speech recognition, connectionist temporal classification is used in which a dynamic programming approach is combined with the softmax outputs of a recurrent neural network. Therefore, the alignment and the label-wise classification is performed simultaneously with dynamic programming in order to maximize the probability that a particular output sequence is generated for a particular input sequence. Readers are referred to [154, 156].

7.8 Summary

Recurrent neural networks are a class of neural networks that are used for sequence modeling. They can be expressed as time-layered networks in which the weights are shared between different layers. Recurrent neural networks can be hard to train, because they are prone to the vanishing and the exploding gradient problems. Some of these problems can be addressed with the use of enhanced training methods as discussed in Chapter 3. However, there are other ways of training more robust recurrent networks. A particular example that has found favor is the use of long short-term memory network. This network uses a gentler update process of the hidden states in order to avoid the vanishing and exploding gradient problems. Recurrent neural networks and their variants have found use in many applications such as image captioning, token-level classification, sentence classification, sentiment analysis, speech recognition, machine translation, and computational biology.

7.9 Bibliographic Notes

One of the earliest forms of the recurrent network was the Elman network [111]. This network was a precursor to modern recurrent networks. Werbos proposed the original version of backpropagation through time [526]. Another early algorithm for backpropagation in recurrent neural networks is provided in [375]. The vast majority of work on recurrent networks has been on symbolic data, although there is also some work on real-valued time series [80, 101, 559]. The regularization of recurrent neural networks is discussed in [552].

The effect of the spectral radius of the hidden-hidden matrix on the vanishing/exploding gradient problem is discussed in [220]. A detailed discussion of the exploding gradient problem and other problems associated with recurrent neural networks may be found