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:

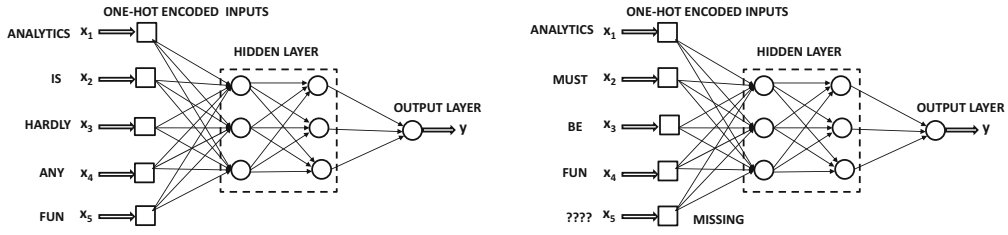
1. In a time-series data set, the values on successive time-stamps are closely related to one another. If one uses the values of these time-stamps as independent features, then key information about the relationships among the values of these time-stamps is lost. For example, the value of a time-series 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

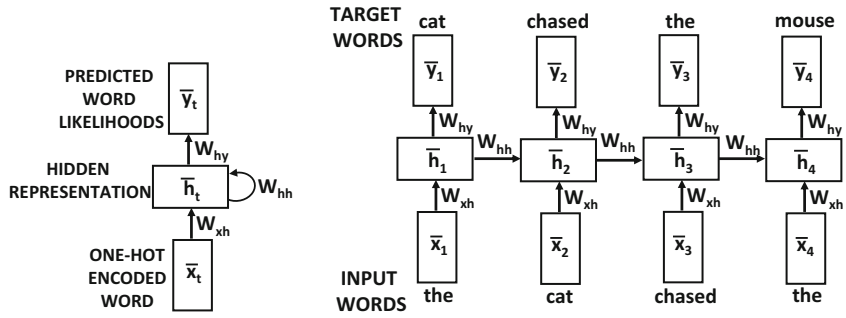
This chapter is organized as follows. The next section will introduce the basic architecture of the recurrent neural network along with the associated training algorithm. The challenges of training recurrent networks are discussed in Section 7.3. Because of these challenges, several variations of the 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

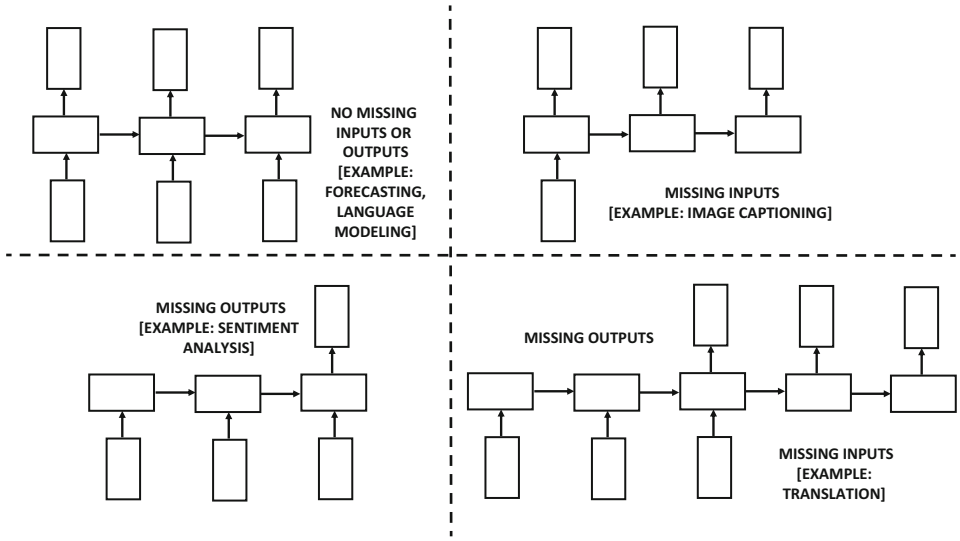


Figure 7.3: The different variations of recurrent networks with missing inputs and outputs

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

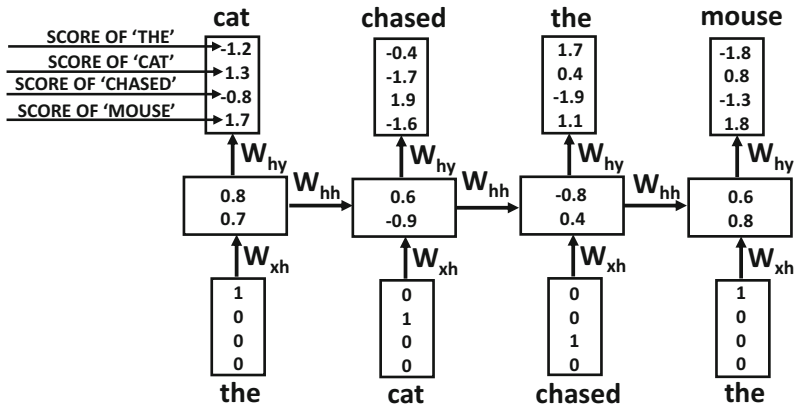


Figure 7.4: Example of language modeling with a recurrent neural network

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.

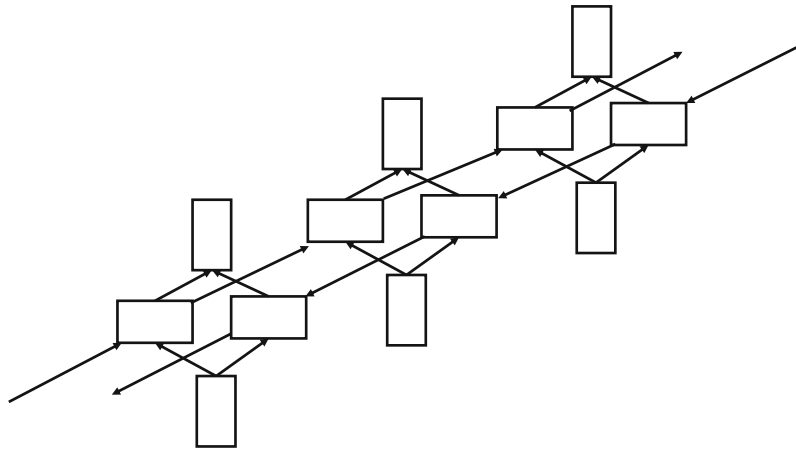


Figure 7.5: Showing three time-layers of a bidirectional recurrent network

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:

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

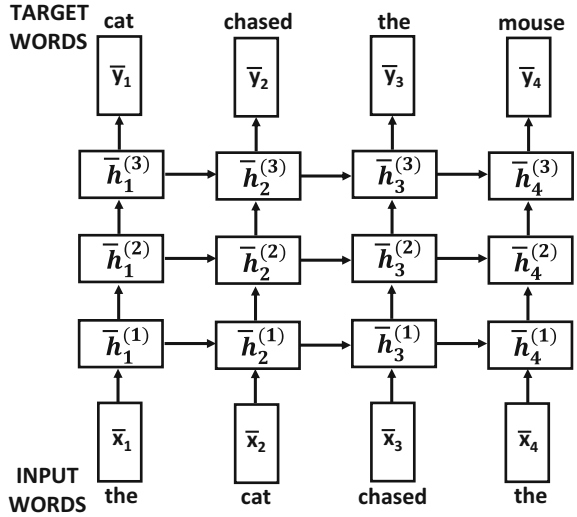


Figure 7.6: Multi-layer recurrent neural networks

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.

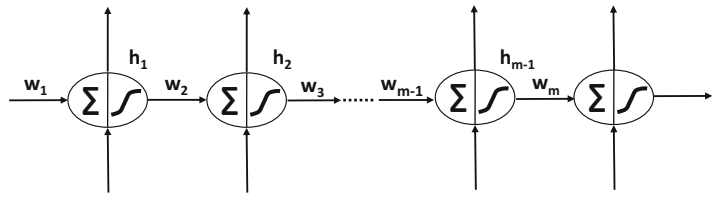


Figure 7.7: The vanishing and exploding gradient problems

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

activation function, for which the derivative $\Phi'(\cdot)$ is almost always less than 1, tends to increase the chances of the vanishing gradient problem.

Although the above discussion only studies the simple case of a hidden layer with one unit, one can generalize the argument to a hidden layer with multiple units [220]. In such a case, it can be shown that the update to the gradient boils down to a repeated multiplication with the same matrix A. One can show the following result:

Lemma 7.3.1 *Let A be a square matrix, the* **magnitude** *of whose largest eigenvalue is λ. Then, the entries of A^t tend to 0 with increasing values of t, when we have $\lambda < 1$. On the other hand, the entries of A^t diverge to large values, when we have $\lambda > 1$.*

The proof of the above result is easy to show by diagonalizing $A = P\Delta P^{-1}$. Then, it can be shown that $A^t = P\Delta^t P^{-1}$, where Δ is a diagonal matrix. The magnitude of the largest diagonal entry of Δ^t either vanishes with increasing t or it grows to an increasingly large value (in absolute magnitude) depending on whether than eigenvalue is less than 1 or larger than 1. In the former case, the matrix A^t tends to 0, and therefore the gradient vanishes. In the latter case, the gradient explodes. Of course, this does not yet include the effect of the activation function, and one can change the threshold on the largest eigenvalue to set up the conditions for the vanishing or exploding gradients. For example, the largest possible value of the sigmoid activation derivative is 0.25, and therefore the vanishing gradient problem will definitely occur when the largest eigenvalue is less that $1/0.25 = 4$. One can, of course, combine the effect of the matrix multiplication and activation function into a single *Jacobian* matrix (cf. Table 3.1 of Chapter 3), whose eigenvalues can be tested.

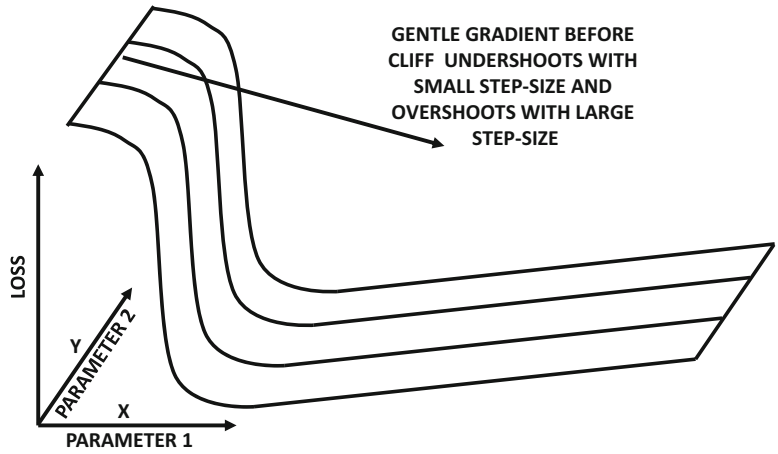


Figure 7.8: Revisiting Figure 3.13 of Chapter 3: An example of a cliff in the loss surface

In the particular case of recurrent neural networks, the *combination of the vanishing/exploding gradient and the parameter tying across different layers causes the recurrent neural network to behave in an unstable way with gradient-descent step size.* In other words, if we choose a step size that is too small, then the effect of some of the layers will cause little progress. On the other hand, if we choose a step size that is too large, then the effect of some of the layers will cause the step to overshoot the optimal point in an unstable way. An important issue here is that the gradient only tells us the best direction of movement for infinitesimally small steps; for finite steps, the behavior of the update could be substantially different from what is predicted by the gradient. The optimal points in recurrent networks

are often hidden near cliffs or other regions of unpredictable change in the topography of the loss function, which causes the best directions of *instantaneous* movement to be extremely poor predictors of the best directions of *finite* movement. Since any practical learning algorithm is required to make finite steps of reasonable sizes to make good progress towards the optimal solution, this makes training rather hard. An example of a cliff is illustrated in Figure 7.8. The challenges associated with cliffs are discussed in Section 3.5.4 of Chapter 3. A detailed discussion of the exploding gradient problem and its geometric interpretation may be found in [369].

There are several solutions to the vanishing and exploding gradient problems, not all of which are equally effective. For example, the simplest solution is to use strong regularization on the parameters, which tends to reduce some of the problematic instability caused by the vanishing and exploding gradient problems. However, very strong levels of regularization can lead to models that do not achieve the full potential of a particular architecture of the neural network. A second solution that is discussed in Section 3.5.5 of Chapter 3 is *gradient clipping*. Gradient clipping is well suited to solving the exploding gradient problem. There are two types of clipping that are commonly used. The first is value-based clipping, and the second is norm-based clipping. In value-based clipping, the largest temporal components of the gradient are clipped before adding them. This was the original form of clipping that was proposed by Mikolov in his Ph.D. thesis [324]. The second type of clipping is norm-based clipping. The idea is that when the entire gradient vector has a norm that increases beyond a particular threshold, it is re-scaled back to the threshold. Both types of clipping perform in a similar way, and an analysis is provided in [368].

One observation about suddenly changing curvatures (like cliffs) is that first-order gradients are generally inadequate to fully model local error surfaces. Therefore, a natural solution is to use higher-order gradients. The main challenge with higher-order gradients is that they are computationally expensive. For example, the use of second-order methods (cf. Section 3.5.6 of Chapter 3) requires the inversion of a Hessian matrix. For a network with 10^6 parameters, this would require the inversion of a $10^6 \times 10^6$ matrix. As a practical matter, this is impossible to do with the computational power available today. However, some clever tricks for implementing second-order methods with Hessian-free methods have been proposed recently [313, 314]. The basic idea is to never compute the Hessian matrix exactly, but always work with rough approximations. A brief overview of many of these methods is provided in Section 3.5.6 of Chapter 3. These methods have also met with some success in training recurrent neural networks.

The type of instability faced by the optimization process is sensitive to the specific point on the loss surface at which the current solution resides. Therefore, choosing good initialization points is crucial. The work in [140] discusses several types of initialization that can avoid instability in the gradient updates. Using momentum methods (cf. Chapter 3) can also help in addressing some of the instability. A discussion of the power of initialization and momentum in addressing some of these issues is provided in [478]. Often simplified variants of recurrent neural networks, like *echo-state networks*, are used for creating a robust initialization of recurrent neural networks.

Another useful trick that is often used to address the vanishing and exploding gradient problems is that of batch normalization, although the basic approach requires some modifications for recurrent networks [81]. Batch normalization methods are discussed in Section 3.6 of Chapter 3. However, a variant known as *layer normalization* is more effective in recurrent networks. Layer normalization methods have been so successful that they have become a standard option while using a recurrent neural network or its variants.

Finally, a number of variants of recurrent neural networks are used to address the vanishing and exploding gradient problems. The first simplification is the use of echo-state networks in which the hidden-to-hidden matrices are randomly chosen, but only the output layers are trained. In the early years, echo-state networks were used as viable alternatives to recurrent neural networks, when it was considered too hard to train recurrent neural networks. However, these methods are too simplified to used in very complex settings. Nevertheless, these methods can still be used for robust initialization in recurrent neural networks [478]. A more effective approach for dealing with the vanishing and exploding gradient problems is to arm the recurrent network with internal memory, which lends more stability to the states of the network. The use of long short-term memory (LSTM) has become an effective way of handing the vanishing and exploding gradient problems. This approach introduces some additional states, which can be interpreted as a kind of long-term memory. The long-term memory provides states that are more stable over time, and also provide a greater level of stability to the gradient-descent process. This approach is discussed in Section 7.5.

7.3.1 Layer Normalization

The batch normalization technique discussed in Section 3.6 of Chapter 3 is designed to address the vanishing and exploding gradient problems in deep neural networks. In spite of its usefulness in most types of neural networks, the approach faces some challenges in recurrent neural networks. First, the batch statistics vary with the time-layer of the neural network, and therefore different statistics need to be maintained for different time-stamps. Furthermore, the number of layers in a recurrent network is *input-dependent*, depending on the length of the input sequence. Therefore, if a test sequence is longer than any of the training sequences encountered in the data, mini-batch statistics may not be available for some of the time-stamps. In general, the computation of the mini-batch statistics is not equally reliable for different time-layers (irrespective of mini-batch size). Finally, batch normalization cannot be applied to online learning tasks. One of the problematic issues is that batch normalization is a relatively unconventional neural network operation (compared to traditional neural networks) because the activations of the units depend on other training instances in the batch, and not just the current instance. Although batch-normalization can be adapted to recurrent networks [81], a more effective approach is *layer normalization*.

In layer normalization, the normalization is performed only over a single training instance, although the normalization factor is obtained by using all the current activations *in that layer of only the current instance*. This approach is closer to a conventional neural network operation, and we no longer have the problem of maintaining mini-batch statistics. All the information needed to compute the activations for an instance can be obtained from that instance only!

In order to understand how layer-wise normalization works, we repeat the hidden-to-hidden recursion of page 276:

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

This recursion is prone to unstable behavior because of the multiplicative effect across time-layers. We will show how to modify this recurrence with layer-wise normalization. As in the case of conventional batch normalization of Chapter 3, the normalization is applied to *pre-activation* values before applying the tanh activation function. Therefore, the pre-activation value at the tth time-stamp is computed as follows:

$$\overline{a}_t = W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}$$

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.