Recurrent Neural Networks

7

7.1 Sequences of Unequal Length

Let us take bird's eye view of things. Feedforward neural networks can process vectors, and convolutional neural networks can process matrices (which are translated into vectors). How would we process sequences of unequal length? If we are talking about, e.g. images of different sizes, then we could simply re-scale them to match. If we have a 800 by 600 image and a 1600 by 1200, it is obvious we can simply resize one of the images. We have two options. The first option is to make the bigger picture smaller. We could do this in two ways: either by taking the average of four pixels, or by max-pooling them. On the other hand, we can similarly make the image bigger by interpolating pixels. If the images do not scale nicely, e.g. one is 800 by 600 an the other is 800 by 555, we can simply expand the image in one direction. The deformations made will not affect the image processing since the image will retain most of the shapes. A case where it would affect the neural network would be if we were to build a classifier to discriminate between ellipses and circles and then resize the images, since that would make circles look like ellipses. Note, that if all matrices, we analyse are of the same size they can be represented by long vectors, as we have seen in the section on MNIST. If they vary in size, we cannot encode them as vectors and keep the nice properties since the rows would be of different lengths. If all images are 20 by 20, then we can translate them in a vector of size 400. This means that the second pixel in the third row of the image is the 43 component of the 400-dimensional vector. If we have two images one 20 by 20 and one 30 by 30, then the 43rd component of the ?-dimensional vector (suppose for a second that we can fit a dimensionality here somehow), would be the second pixel in the third row of the first image and the thirteenth pixel of the second row of the second image. But, the real problem is how to fit vectors of different dimensions (400 and 300) in a neural network. Everything we have seen so far, needs a fixed-dimensional vectors.

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics
in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_7

The problem of varying dimensionality can be seen as the problem of learning sequences of unequal length, and audio processing is a nice example of how we might need this, since various audio clips are necessarily of different lengths. We could in theory just take the longest and then make all others of the same length as that one, but this is waste in terms of the space needed. But there is a deeper problem here. Silence is a part of language, and it is often used for communicating meaning, so a sound clip with some content labeled with the label 1 in the training set might be correct, but if add 10 s of silence at the beginning or the end of the clip, the label 1 might not be appropriate anymore, since the clip with the silence may have a different meaning. Think about irony, sarcasm and similar phenomena.

So the question is what we can do? The answer is that we need a different nerural network architecture than we have seen before. Every neural network we have seen so far has connections which push the information forward, and this is why we have called them 'feedforward neural networks'. It will turn out that by having connections that feed the output back into a layer as inputs, we can process sequences of unequal length. This makes the network deep, but it does share weights so it partly avoids the vanishing gradient problem. Networks that have such feedback loops are called *recurrent neural networks*. In the history of recurrent neural networks, there is an interesting twist. As soon as the idea of the perceptron did not seem good, the idea of making a 'multi-layer perceptron' seemed natural. Remember that this idea was theoretical and predated backpropagation (which was widely accepted after 1986), which means that no one was able to make it work back then. Among the theoretical ideas explored was adding a single layer, adding multiple layers and adding feedback loops, which are all natural and simple ideas. This was before 1986.

Since backpropagation was not yet available, J. J. Hopfield introduced the idea of Hopfield networks [1], which can be thought of the first successful recurrent neural networks. We will explore them in detail in Chap. 10. They were specific since they were different from what we consider today to be recurrent neural networks. The most important recurrent neural networks are the *long short-term memory* networks or *LSTM*s which were invented in 1997 by Hochreiter and Schmidhuber [2]. To this day, they remain the most widely used recurrent neural networks and are responsible for many state-of-the-art results in various fields, from speech recognition to machine translation. In this chapter, we will focus on developing the necessary concepts to explain the LSTM in detail.

7.2 The Three Settings of Learning with Recurrent Neural Networks

Let us return a bit to the naive Bayes classifier. As we saw in Chap. 3, the naive Bayes classifier calculates $\mathbb{P}(target|features)$ after we calculate $\mathbb{P}(feature1|target)$, $\mathbb{P}(feature2|target)$, etc., from the dataset. This is how the naive Bayes classifier works, but all classifiers (supervised learning algorithms) try to calculate $\mathbb{P}(target|features)$ or $\mathbb{P}(\mathbf{t}|\mathbf{x})$ in some way. Recall that any predicate $\mathbb{P}$ such that

(i) $\mathbb{P}(A) \geq 0$, (ii) $\mathbb{P}(\Omega) = 1$, where Ω is the possibility space and (iii) for all disjoint A_n, $n \in \mathbb{N}$, $\mathbb{P}(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mathbb{P}(A_n)$ is a probability predicate. Moreover, it is *the* probability predicate (try to work out the why by yourself).

Taking the probabilistic interpretation to analyze the machine learning algorithms from a bird's-eye perspective, we could say that what a *supervised* machine learning algorithm does is calculate $\mathbb{P}(\mathbf{t}|\mathbf{x})$ (where $\mathbf{x}$ denotes an input vector, and $\mathbf{t}$ denotes the target vector[1]). This is the *classic setting*, simple supervised learning with labels.

Recurrent neural networks can learn in this standard setting by simply digesting a lot of labelled sequences and then they predict the label of each finished sequence. An example might be classifying audio clips according to emotions. But recurrent neural networks are capable of much more. They can also learn from sequences with multiple labels. Imagine an industrial robotic arm that we wish to train to perform a task. It has a multitude of sensors and it has to learn directions (for simplicity suppose we have only four, North, South, East and West). The training set is then produced with movement sequences, each consisting of a string of directions, e.g. $x_1 N x_2 N x_3 W x_4 E x_5 W x_6 W$ or just $x_1 N x_2 W$. Notice how different this is from what we have seen before. Here we have a sequence of sensor data (x_i) and movements (N, E, S or W, we will denote them by D). Notice that it would be a very bad idea to break up the sequences in $x D$ pieces, since a movement of the form $x N x N$ might happen most often when broken, it might make sense only in the beginning of the sequence (e.g. as a 'get out of the dock' command) and in any other case it would be disastrous. Sequences cannot be broken, and it is not enough to know the previous state to be able to predict the next. The idea that the next state depends only on the current is known as the *Markov assumption*, and one of the greatest strengths of the recurrent neural networks is that they do not need to make the Markov assumption—they can model more complex behaviour. This means that the recurrent network learns from uneven sequences whose parts are labelled and it creates a bunch of labels when it predicts over an unknown vector. This we will call *sequential setting*.

There is a third setting which is an evolved form of the sequential setting and we can call it the *predict-next setting*. This setting does not need labels at all and it is commonly used for natural language processing. Actually, it has labels, but they are implicit. The idea is that for every input sequence (sentence), the recurrent network breaks it down to subsequences and use the next word as the target. We will need special tokens for the start and end of the sentence, which we must put in manually, and we denote them here by \$ ('start') and & ('end'). If we have a sentence 'All I want for Christmas is you', then we first have to transform it into '\$ all I want for Christmas is you &'.[2] Then the sentence is broken into inputs and targets, which we will denote as ('input string','target'):

[1] In machine learning literature, it is common to find the notation $\hat{y}$, which denotes the results from the predictor, and y is kept for denoting target values. We have used a different notation, more common to deep learning, where y denotes the outputs from the predictor, and t is used to denote actual values or targets.

[2] Notice which capital letters we kept and try to conclude why.

- (‘$’,‘all’)
- (‘$ all’,‘I’)
- (‘$ all I’,‘want’)
- (‘$ all I want’, ‘for’)
- (‘$ all I want for’, ‘Christmas’)
- (‘$ all I want for Christmas’, ‘is’)
- (‘$ all I want for Christmas is’, ‘you’)
- (‘$ all I want for Christmas is you’, ‘&’).

Then, the recurrent network will learn how to return the most likely next word after hearing a word sequence. This means that the recurrent network is learning a probability distribution from the inputs, i.e. $\mathbb{P}(\mathbf{x})$, which actually makes this unsupervised learning, since there are no targets. Targets here are synthesized from the inputs.

Note that we will usually want to limit how many words we want to look back (i.e. the word-wise length of the ‘input string’ part). Notice that this is actually quite a big deal since this can be seen as a question answering capability, which is the basis of the Turing test, and this is a step towards not just a useful tool, but also towards general AI. But, we have to make one tiny adjustment here. Notice that if the recurrent network learns which is the most probable word following a sequence, it might become repetitive. Imagine that we have the following five sentences in the training set:

- ‘My name is Cassidy’
- ‘My name is Myron’
- ‘My name is Marcus’
- ‘My name is Marcus’
- ‘My name is Marcus’.

Now, the recurrent neural network would conclude that $\mathbb{P}(Marcus) = 0.6$, $\mathbb{P}(Myron) = 0.2$ and $\mathbb{P}(Cassidy) = 0.2$. So when given a sequence ‘My name is’ it would always pick ‘Marcus’ since it has the highest probability. The trick here is not to let it pick the one with the highest probability, but rather the recurrent neural network should build a probability distribution for every input sequence with the individual probabilities of all outcomes and then randomly sample it. The result will be that in 60% of the time it will give ‘Marcus’ but sometimes it will also produce ‘Myron’ and ‘Cassidy’. Note that this actually solves quite a bit of problems which might arise. If it were not so, we would have always the same response to the same sequences of words. Now that we have given a quick black box view, it is time to dig deep into the mechanics of recurrent neural networks.

7.3 Adding Feedback Loops and Unfolding a Neural Network

Let us now see how recurrent neural networks work. Remember the vanishing gradient problem? There we have seen that adding layers one after the other would severely cripple the ability to learn weights by gradient descent, since the movements would be really small, sometimes even rounded to zero. Convolutional neural networks solved this problem by using a shared set of weights, so learning even little by little is not a problem since each time the same weights get updated. The only problem is that convolutional neural networks have a very specific architecture making them best suited for images and other limited sequences.

Recurrent neural networks work not by adding new layers to a simple feedforward neural network, but by adding recurrent connections on the hidden layer. Figure 7.1a shows a simple feedforward neural netwok and Fig. 7.1b shows how to add recurrent connections to the simple feedforward neural network from Fig. 7.1a. The *outputs* from a given layer are denoted by $\mathbf{I}$, $\mathbf{O}$ and $\mathbf{H}$ for the simple feedforward network, and by $\mathbf{H}_1, \mathbf{H}_2, \mathbf{H}_3, \mathbf{H}_4, \mathbf{H}_5, \ldots$ when we add recurrent connections. The weights in the simple feedforward network are denoted by $\mathbf{w}_x$ (input-to-hidden) and $\mathbf{w}_o$ (hidden-to-output). It is very important not to confuse *multiple outputs from a hidden layer with multiple hidden layers*, since a layer is actually defined in terms of weights, i.e. each layer has its own set of weights, and here all $\mathbf{H}_n$ share the same set of weights, viz. $\mathbf{w}_h$. Figure 7.1c is exactly the same as Fig. 7.1b with the only difference being that we condensed the individual neurons (circles) into vectors (rectangles), which we have been doing since Chap. 3 in our calculations, but now we do it on the visual display as well. Notice that to add the recurrent connection, we had to add a set of weights, $\mathbf{w}_h$, to the calculation and this is all that is needed to add recurrence to the network.

Note that the recurrent neural network can be unfolded so that the recurrent connections are all specified. Figure 7.2a shows the previous network and Fig. 7.2 shows how to unfold the recurrent connections. Figure 7.2c is the same as Fig. 7.2b but with

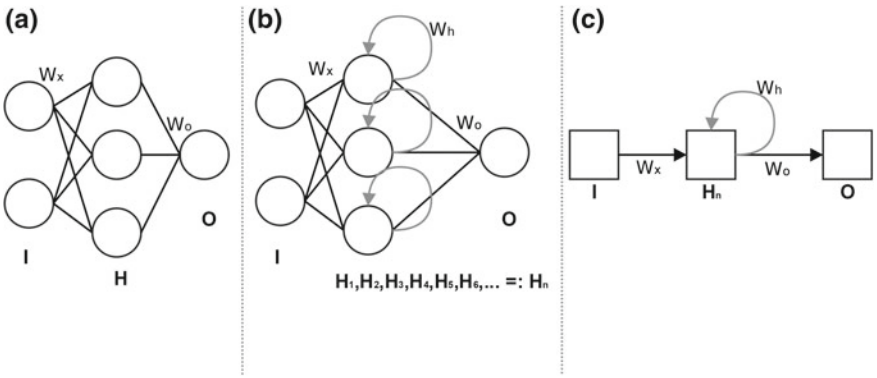

Fig. 7.1 Adding recurrent connections to a simple feedforward neural network

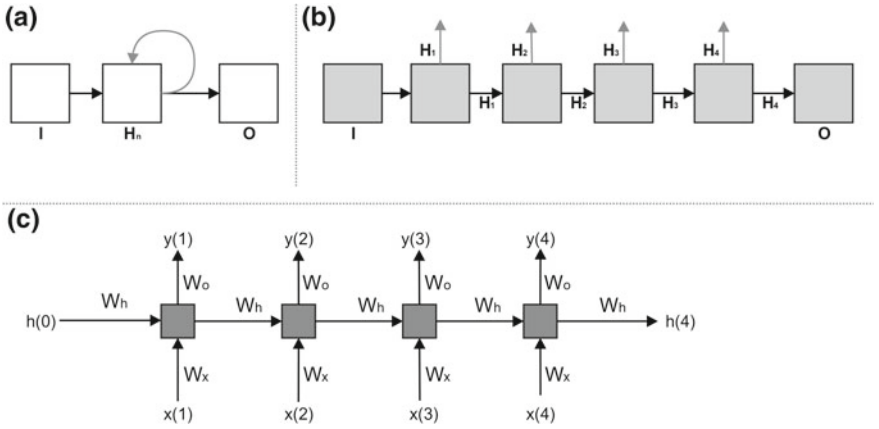

Fig. 7.2 Unfolding a recurrent neural network

the proper and detailed notation used in the recurrent neural network literature, and we will focus on this representation for commenting on the fly how a recurrent neural network works. The next section will use the sub-image C of Fig. 7.2 for reference, and this will be our standard notation for the rest of the chapter.[3]

7.4 Elman Networks

Let us comment on the Fig. 7.2c. $\mathbf{w}_x$ represent input weights, $\mathbf{w}_h$ represent the recurrent connection weights and the $\mathbf{w}_o$ the hidden-to-output weights. The xs are inputs, and the ys are outputs, just like before. But here we have an additional sequential nature, which tries to capture time. So $x(1)$ is the first input, and later it gets $x(2)$ and so on. The same holds of outputs. If we have the classic setting, we would only be using $x(1)$ (to give the input vector) and $y(4)$ to catch the (overall) output. But for the sequential and predict-next settings, we would be using all xs and ys.

Notice that unlike the situation we had in simple feedforward networks, here we also have the h, and they represent the inputs for the recurrent connection. We need something to start with, and we can generate $h(0)$ by simply setting all its entries to 0. We give an example calculation where it can be seen how to calculate all elements and it will be much more insightful than giving a piece by piece calculation. By f, we will be denoting a nonlinearity, and you can think of it as the *logistic function*. A bit later we will see a new nonlinearity called *softmax*, which can be used here and has natural fit with recurrent neural networks. So, the recurrent neural network

[3] We used the shades of grey just to visually denote the gradual transition to the proper notation.

calculates the output y at a final time t. The calculation can be unfolded to the following recursive structure (which makes it clear why we need $h(0)$):

$$y(t) = f(\mathbf{w}_o^\top h(t)) = \tag{7.1}$$
$$= f(\mathbf{w}_o^\top f(\mathbf{w}_h^\top h(t-1) + \mathbf{w}_x^\top x(t))) = \tag{7.2}$$
$$= f(\mathbf{w}_o^\top f(\mathbf{w}_h^\top f(\mathbf{w}_h^\top h(t-2) + \mathbf{w}_x^\top x(t-1)) + \mathbf{w}_x^\top x(t))) = \tag{7.3}$$
$$= f(\mathbf{w}_o^\top f(\mathbf{w}_h^\top f(\mathbf{w}_h^\top f(\mathbf{w}_h^\top h(t-3) + \mathbf{w}_x^\top x(t-2)) + \mathbf{w}_x^\top x(t-1)) + \mathbf{w}_x^\top x(t))). \tag{7.4}$$

We can make this more readable by condensing it to two equations:

$$h(t) = f_h(\mathbf{w}_h^\top h(t-1) + \mathbf{w}_x^\top x(t)) \tag{7.5}$$
$$y(t) = f_o(\mathbf{w}_o^\top h(t)), \tag{7.6}$$

where f_h is the nonlinearity of the hidden layer, and f_o is the nonlinearity of the output layer, which are not necessarily the same function, but they can be the same if we want. This type of recurrent neural network is called *Elman networks* [3], after the linguist and cognitive scientist Jeffrey L. Elman.

If we change the $h(t-1)$ for $y(t-1)$ in Eq. 7.5, so that it becomes as follows:

$$h(t) = f_h(\mathbf{w}_h^\top y(t-1) + \mathbf{w}_x^\top x(t)). \tag{7.7}$$

We obtain a *Jordan network* [4], which are named after the psychologist, mathematician and cognitive scientist Michael I. Jordan. Both Elman and Jordan networks are known in the literature as *simple recurrent networks* (SRN for short). Simple recurrent networks are seldom used in applications today, but they are the main teaching method for explaining recurrent networks before running in the much more complex LSTMs, which are the main recurrent architecture used today. It is very easy to look down on SRNs today, but when they were first proposed, it became the first model that could operate on words of a text without having to rely on an 'alien' representation such as the bag of words or n-grams. In a sense, those representations seemed to suggest that language processing is something very foreign to a computer, since people do not use anything like the Bag of words for understanding language. The SRN made a decisive move towards the language processing as word sequence processing paradigm we have today, and made the whole process much closer to human intelligence. Consequently, SRNs should be considered a milestone in AI, since they have made that crucial step: what previously seemed impossible was now conceivable. But a couple of years later, a stronger architecture would come and take over all practical applications, but this strength comes with a price: LSTMs are much slower to train than SRNs.

7.5 Long Short-Term Memory

In this section, we will give a graphical illustration of the workings of the *long short-term memory* (LSTM), and the interested reader should have no problem in coding LSTMs from scratch just by following our explanation and the accompanying images. All images in the current section on LSTMs are reproduced from Christopher Olah's blog.[4] We follow the same notation as is used there (except from a couple of minor details), and we omit the weights in Fig. 7.3 to simply exposition, but we will add them when addressing individual components of the LSTM in the later images. Since we know from Eq. 7.5 that $y(t) = f_o(\mathbf{w}_o \cdot h(t))$ (f_o is the nonlinearity of choice for the output layer), in this chapter $y(t)$ is the same as $h(t)$, but we still point to the places, where $h(t)$ is to be multiplied by $\mathbf{w}_o$ to get $y(t)$ by simply noting $y(t) = h(t)$. This is really not that important from a purely formal point of view, but we hope to be more clear by holding a place for $y(t)$.

Figure 7.3 shows a bird's-eye perspective on LSTMs and compares them to SRNs. One thing that can be seen right away is that SRNs have one link from one unit to the next (it is the flow of $h(t)$), whereas the LSTMs have the same $h(t)$ but also $C(t)$. This $C(t)$ is called *the cell state*, an this is the main flow of information through the LSTMs. Figuratively speaking, the cell state is the 'L', the 'T' and the 'M' from 'LSTM', i.e. it is the *long-term memory* of the model. Everything else that happens is just different filters to decide what should be kept or added to the cell state. The

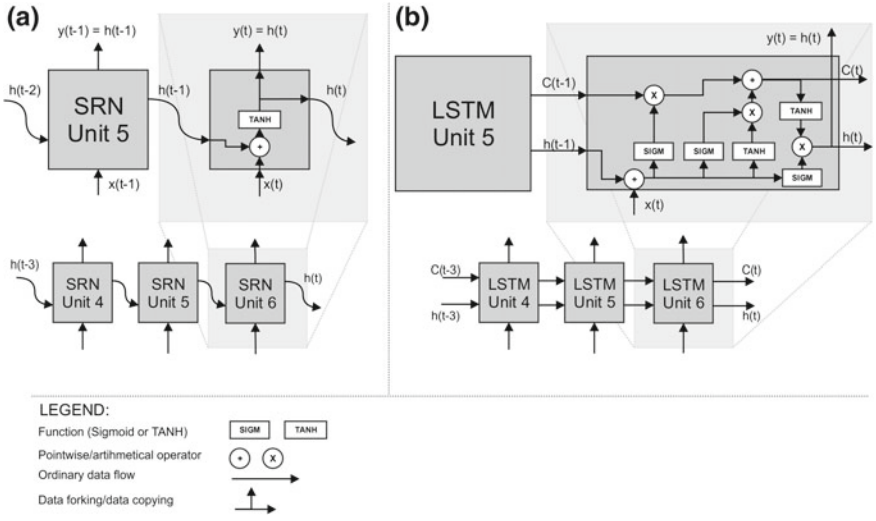

Fig. 7.3 SRN and LSTM units zoomed

[4] http://colah.github.io/posts/2015-08-Understanding-LSTMs/, accessed 2017-03-22.

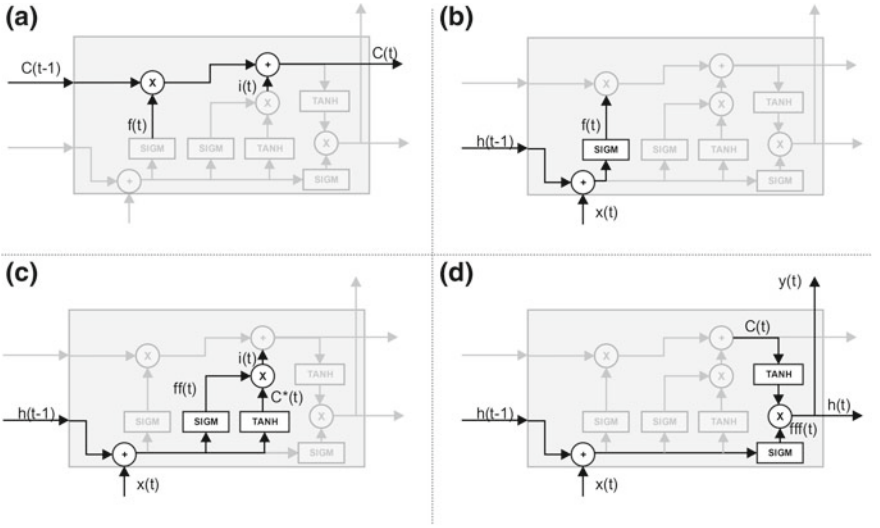

Fig. 7.4 Cell state (**a**), forget gate (**b**), input gate (**c**) and output gate (**d**)

cell state is emphasized on Fig. 7.4a (for now you should ignore the $f(t)$ and $i(t)$ on the image, you will see how they are calculated in a couple of paragraphs).

The LSTM adds or removes information from the cell with so-called *gates*, and these make up the rest of the unit in an LSTM. The gates are actually very simple. They are a combination of addition, multiplication and nonlinearities. The nonlinearities are used simply to 'squash' information. The logistic or sigmoid function (denoted as SIGM in the images) is used to 'squash' information to values between 0 and 1, and the hyperbolic tangent (denoted as TANH in the images) is used to 'squash' the information to values between -1 and 1. You can think of it in the following way: SIGM makes a fuzzy 'yes'/'no' decision, while TANH makes a fuzzy 'negative'/'neutral'/'positive' decision. They do nothing else except this.

The first gate is the *forget gate*, which is emphasized in Fig. 7.4b. The name 'gate' comes from analogies with the logic gates. The forget gate at unit t is denoted by $f(t)$, and is simply $f(t) := \sigma(\mathbf{w}_f(x(t)+h(t-1)))$. Intuitively, it controls how much of the weighted raw input and weighted previous hidden state is to be *remembered*. Note that the σ is the symbol for the logistic function.

Regarding weights, there are different approaches, but we consider the most intuitive to be the one which breaks up $\mathbf{w}_h$ into several different weights, $\mathbf{w}_f$, $\mathbf{w}_{ff}$, $\mathbf{w}_C$ and $\mathbf{w}_{fff}$.[5] The point to remember is that there are different ways to look at the weights and some of them try to keep the same names as they had in simpler models, but the most natural approach for deep learning is to think of an architecture as composed

[5]Notice that we are not quite precise here and that the $\mathbf{w}_f$ in the LSTMs is actually the same as $\mathbf{w}_x$ in the SRN and not a component of the old $\mathbf{w}_h$.

of basic 'building blocks' to be assembled together like LEGO® bricks, and then each block should have its own set of weights. All of the weight in a complete neural network are trained together with backpropagation and the joint training actually makes a neural network a connected whole (like each LEGO brick normally has its own studs to connect to other bricks to make a structure).

The next gate (emphasized in Fig. 7.4c), called the *input gate*, is a bit more complex. It basically decides on what to put in the cell state. It is composed of another forget gate (which we unimaginatively denote with $ff(t)$) but with different weights, but it also has an additional module which creates candidates to be added to the cell state. The $ff(t)$ can be thought of as a saving mechanism, which controls how much of the input we will save to the cell state. In symbols:

$$ff(t) := \sigma(\mathbf{w}_{ff}(x(t) + h(t-1))), \tag{7.8}$$

$$i(t) := ff(t) \cdot C^*(t). \tag{7.9}$$

What we are missing is a calculation for the candidates (denoted by $C^*(t)$). Calculating the candidates is pretty easy: $C^*(t) := \tau(\mathbf{w}_C \cdot (x(t) + h(t-1)))$, where τ is the symbol for the hyperbolic tangent or $tanh$. We are using the hyperbolic tangent here to squash the results to values which range between -1 and 1. Intuitively, the negative part of the range (-1 to 0) can be seen as a way to get quick 'negations', so that even opposites would be considered to get, for example a quick processing of linguistic antonyms.

As we have seen before, an LSTM unit has three outputs: $C(t)$, $y(t)$ and $h(t)$. We have all we need to compute the current cell state $C(t)$ (this calculation is shown in Fig. 7.4a):

$$C(t) := f(t) \cdot C(t-1) + i(t). \tag{7.10}$$

Since $y(t) = g_o(\mathbf{w}_o \cdot h(t))$ (where g_o is a nonlinearity of choice), all that is left is to compute $h(t)$. To compute $h(t)$, we will need a third copy of the forget gate ($fff(t)$), which will have the task of deciding which parts of the inputs and how much of it to include in $h(t)$:

$$fff(t) := \sigma(\mathbf{w}_{fff}(x(t) + h(t-1))). \tag{7.11}$$

Now, the only thing left for a complete *output gate* (whose result is actually not $o(t)$ but $h(t)$), we need to multiply the $fff(t)$ by the current cell state squashed between -1 and 1:

$$h(t) := fff(t) \cdot \tau(C(t)). \tag{7.12}$$

And now finally, we have the complete LSTM. Just a quick final remark: the $fff(t)$ can be thought of as a 'focus' mechanism which tries to say what is the most important part of the cell state. You might think of $f(t)$, $ff(t)$ and $fff(t)$, but the idea is that they all participate in different parts and as such, we hope they will take on the mechanism we want ('remember from last unit', 'save input' and 'focus on this part of the cell state' respectively). Remember that this is only our wild hope, we

have no way to 'force' this interpretation on the LSTM other than with the sequence of calculations or flow of information we have chosen to use. This means that these interpretations are metaphorical, and only if we have made a one-in-a-million lucky guesstimate will these mechanisms actually coincide with the mechanisms in the human brain.

The LSTMs have been first proposed by Hochreiter and Schmidhuber in 1997 [2], and they have become one of the most important deep architectures for natural language processing, time series analysis and many other sequential tasks. Today one of the best reference books on recurrent neural networks is [5], and we highly recommend it for any reader that wishes to specialize in these amazing architectures.

7.6 Using a Recurrent Neural Network for Predicting Following Words

In this section, we give a practical example of a simple recurrent neural network used for predicting next words from a text. This sort of task is highly flexible, since it allows not just predictions but also question answering—the (single word) answer is simply the next word in the sequence. The example we use is a modification of an example from [6], with ample comments and explanations. Some portions of the original code have been modified to make the code easier to understand. As we explained in the previous section, this is a working Python 3 code, but you will need to install all dependencies. You should also be able to follow the ideas from the code on chapter, but to see the subtleties, one needs to have the actual code on the computer.[6] We start by importing the Python libraries and we will be needing:

```
from keras.layers import Dense, Activation
from keras.layers.recurrent import SimpleRNN
from keras.models import Sequential
import numpy as np
```

The next thing is to define hyperparameters:

```
hidden_neurons = 50
my_optimizer ="sgd"
batch_size = 60
error_function = "mean_squared_error"
output_nonlinearity = "softmax"
cycles = 5
epochs_per_cycle = 3
context = 3
```

[6]Which you can get either from the book's GitHub repository, or by typing in all the code in this section in one simple file (.txt) and rename it to change its extension to .py.

Let us take a minute and see what we are using. The variable `hidden_neurons` simply states how many hidden units are we going to use. We use Elman units here, so this is the same as the number of feedback loops on the hidden layer. The variable `optimizer` defines which Keras optimizer we are going to use, and in this case it is the stochastic gradient descent, but there are others,[7] and we recommend to experiment with several optimizers just to get a feel. Note that `"sgd"` is a Keras name for it, so you must type it exactly like this, not `"SGD"`, nor `"stochastic_GD"`, nor anything similar. The `batch_size` simply says how many examples we will use for a single iteration of the stochastic gradient descent. The variable `error_function = "mean_squared_error"` tells Keras to use the MSE we have been using before.

But now we come to the activation function `output_nonlinearity`, and we see something we have not seen before, the *softmax* activation function or nonlinearity, with its Keras name `"softmax"`. The softmax function is defined as

$$\zeta(z_j) := \frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_k}}, j = 1, \ldots, N. \tag{7.13}$$

The softmax is quite a useful function: it basically transforms a vector $\mathbf{z}$ with arbitrary real values to a vector with values ranging from 0 to 1, and they are such that they all add up to 1. This is why the softmax is very often used in the final layer of a deep neural network used for multiclass classification[8] to get the output which can be a probability proxy for the classes. It can be shown that if the vector $\mathbf{z}$ has only two components, z_0 and z_1 (which would simulate binary classification) would reduce *exactly* to the logistic function classification, only with the weight being $\mathbf{w}_\sigma = \mathbf{w}_{\zeta 1} - \mathbf{w}_{\zeta 0}$. We can now continue to the next part of the SRN code, bearing in mind that the rest of the parameters we will comment when they become active in the code:

```
def create_tesla_text_from_file(textfile="tesla.txt"):
    clean_text_chunks = []
    with open(textfile, 'r', encoding='utf-8') as text:
        for line in text:
            clean_text_chunks.append(line)
    clean_text = ("".join(clean_text_chunks)).lower()
    text_as_list = clean_text.split()
    return text_as_list
text_as_list = create_tesla_text_from_file()
```

This part of the code opens a plain text file `tesla.txt`, which will be used for training and predicting. This file should be encoded in utf-8 or the `utf-8` in the

[7] There is a full list on https://keras.io/optimizers/.

[8] Where we have more than two classes. Note that in binary classification were we have two classes, say A and B, we actually do a classification (with, for e.g. the logistic function in the output layer) in only one of them and get a probability score p_A. The probability score of B is then calculated as $1 - p_A$.

code should be changed to reflect the appropriate file encoding. Note that most text editors today distinguish 'file encoding' (actual encoding of the file) from 'encoding' (the encoding used to display text for that file in the editor). This approach will work for files that are about 70% the size of the available RAM on the computer you are using. Since we are talking about plain text files, having an 16GB machine and a 10GB file will work out well, and 10GB is a lot of plain text (just for comparison, the whole English Wikipedia with metadata and page history in plain text has a size of 14GB). For larger datasets, we would take a different approach, namely to separate the big file into chunks and consider them batches, and feed them one by one, but the details of such big data processing are beyond the scope of this book.

Notice that when Python opens and reads a file, it returns it line by line, so we are actually accumulating these lines in a list called `clean_text_chunks`. We then glue all of these together in one big string called `clean_text`, and then cut them into individual words and store it in the list called `text_as_list`, and this is what the whole function `create_tesla_text_from_file(textfile="tesla.txt")` returns. The part `(textfile="tesla.txt")` means that the function `create_tesla_text_from_file()` expects an argument (which is refered to as `textfile`) but we have provided a default value `"tesla.txt"`. This means that if we give a file name, it will use that, otherwise it will se `"tesla.txt"`. The final line `text_as_list = create_tesla_text_from_file()` calls the function (with the default file name), and stores what the function has returned in the variable `text_as_list`. Now, we have all of our text in a list, where each individual element is a word. Notice that there may be repetitions of words here, and that is perfectly fine, as this will be handled by the next part of the code:

```
distinct_words = set(text_as_list)
number_of_words = len(distinct_words)
word2index = dict((w, i) for i, w in enumerate(distinct_words))
index2word = dict((i, w) for i, w in enumerate(distinct_words))
```

The `number_of_words` simply counts the number of words in the text. The `word2index` creates a dictionary with unique words as keys and their position in the text as values, and `index2word` does the exact opposite, creates a dictionary where positions are keys and words are values. Next, we have the following:

```
def create_word_indices_for_text(text_as_list):
    input_words = []
    label_word = []
    for i in range(0,len(text_as_list) - context):
        input_words.append((text_as_list[i:i+context]))
        label_word.append((text_as_list[i+context]))
    return input_words, label_word
input_words,label_word = create_word_indices_for_text(text_as_list)
```

Now, it gets interesting. This is a function which creates a list of input words and a list of label words from the original text, which has to be in the form of a list of

individual words. Let us explain a bit of the idea. Suppose we have a tiny text '*why would anyone ever eat anything besides breakfast food?*'. Then we want to make an 'input'/'label' structure for predicting the next word, and we do this by decomposing this sentence into an array:

Input word 1	Input word 2	Input word 3	Label word
why	would	anyone	ever
would	anyone	ever	eat
anyone	ever	eat	anything
ever	eat	anything	besides
eat	anything	besides	breakfast
anything	besides	breakfast	food?

Note that we have used three input words and declared the next one the label, and then shifted for one word and repeated the process. How many input words we use is actually defined by the hyperparameter `context`, and can be changed. The function `create_word_indices_for_text(text_as_list)` takes a text in the form of the list, creates the input words list and the label word list and returns them both. The next part of the code is

```
input_vectors = np.zeros((len(input_words), context, number_of_words), dtype=np.int16)
vectorized_labels = np.zeros((len(input_words), number_of_words), dtype=np.int16)
```

This code produces 'blank' tensors, populated by zeros. Note that the term 'matrix' and 'tensor' come from mathematics, where they are objects that work with certain operations, and are distinct. Computer science treats them both as multidimensional *arrays*. The difference is that computer science places the focus on their structure: if we iterate along one dimension, all elements along that dimension (properly called 'axis') have the same shape. The type of entries in the tensors will be `int16`, but you can change this as you wish.

Let us discuss tensor dimensions a bit. The tensor `input_vectors` is technically called a *third-order tensor*, but in reality this is just a 'matrix' with three dimensions, or simply a 3D array. To understand the dimensionality of the `input_vectors` tensor note that first we have three words (i.e. a number of words defined by `context`) to make a one-hot encoding of. Notice that we are technically using a one-hot encoding and not a bag of words, since we have only kept distinct words from the text. Since we have a one-hot encoding, this would expand a second dimension. This takes care of the `context` and `number_of_words` dimensions of the tensor, and third one (in the code it is the first one, `len(input_words)`) is actually here just to bundle all inputs together, like we had a matrix holding all input vectors in the previous chapters. The `vectorized_labels` is the same, only here we do not have three or n words specified by the variable `context`, but only a single one, the label word, so we need one less dimension in the tensor. Since we have initialized two blank tensors, we need something to put the 1s in the appropriate places, and the next part of the code does just that which is as follows:

```
for i, input_w in enumerate(input_words):
    for j, w in enumerate(input_w):
        input_vectors[i, j, word2index[w]] = 1
        vectorized_labels[i, word2index[label_word[i]]] = 1
```

It is a bit hard, but try to figure out for yourself how this code 'crawls' the tensors and puts the 1s where they should be.[9] Now, we have cleared all the messy parts, and the next part of the code actually specifies the complete simple recurrent neural network with Keras functions.

```
model = Sequential()
model.add(SimpleRNN(hidden_neurons, return_sequences=False,
input_shape=(context,number_of_words), unroll=True))
model.add(Dense(number_of_words))
model.add(Activation(output_nonlinearity))
model.compile(loss=error_function, optimizer=my_optimizer)
```

Most of the things that can be tweaked here are actually placed in the hyperparameters. No change should be done in this part, except perhaps add a number of new layers, which is done by duplicating the line or lines specifying the layer, in particular the second line, *or* the third *and* fourth lines. The only thing left to do is to see how well does the model work, and what does it produce as output. This is done by the final part of the code which is as follows:

```
   for cycle in range(cycles):
    print(">-<" * 50)
    print("Cycle: %d" % (cycle+1))
    model.fit(input_vectors, vectorized_labels, batch_size = batch_size,
epochs = epochs_per_cycle)
    test_index = np.random.randint(len(input_words))
    test_words = input_words[test_index]
    print("Generating test from test index %s with words %s:" % (test_index,
test_words))
    input_for_test = np.zeros((1, context, number_of_words))
    for i, w in enumerate(test_words):
        input_for_test[0, i, word2index[w]] = 1
    predictions_all_matrix = model.predict(input_for_test, verbose = 0)[0]
    predicted_word = index2word[np.argmax(predictions_all_matrix)]
    print("THE COMPLETE RESULTING SENTENCE IS: %s %s" % ("".join(test_words),
predicted_word))
    print()
```

This part of the code trains and tests the complete SRN. Testing would usually be predicting a part of data we held out (test set) and then measuring accuracy. But here

[9] This is perhaps the single most challenging task in this book, but do not skip it since it will be extremely useful for a good understanding, and it is just four lines of code.

we have the predict-next setting, which does not have labels, so we have to adopt a different approach. The idea is to train and test in a *cycle*. A cycle is composed of a training session (with a number of epochs) and then we generate a test sentence from the text and see whether the word which the network gives makes sense when placed after the words from the text. This completes one cycle. These cycles are cumulative, and sentences will become more and more meaningful after each successive cycle. In the hyperparameters we have specified that we will train for 5 cycles, each having 3 epochs.

Let us make a brief remark on what we have done. For computational efficiency, most tools used for the predict-next make use of the *Markov assumption*. Informally, the Markov assumption means that we simplify a probability which would have to consider all steps from the beginning of time, $\mathbb{P}(s_n|s_{n-1}, s_{n-2}, s_{n-3}, \ldots)$, to a probability which just considers the previous step $\mathbb{P}(s_n|s_{n-1})$. If a system takes this computational detour it is said to 'use the Markov assumption'. If a process turns out to be such that it really does not matter anything but the preceding state in time, it is said to be a *Markov process*. Language production is not a Markov process. Suppose you are a classifier and you have a 'training' sentence: 'We need to remember what is important in life: friends, waffles, work. Or waffles, friends, work. Does not matter, but work is third'. If it were a Markov process, and you could make the Markov assumption without a big loss in functionality, you would be needing just one word and you could tell which one follows. If you have 'Does', you can tell that in you training set, after this it always comes 'not', and you would be right. But if you were given 'work', you would have more trouble, but you could get away with a probability distribution. But what if you did not have a predict-next setting, but your task was to identify when the speaker got confused (i.e. when you try to dig into meaning). Then, you would need all of the previous words for comparison. At many times you can cut corners a bit and make the Markov assumption for non-Markov processes and get away with it, but the point is that unlike many other machine learning algorithms, recurrent neural networks do no *have to make* the Markov assumption, since they are fully capable of handling many time steps, not just the last one.

There is one last thing we need to comment before leaving recurrent neural networks, an this is how backpropagation works. Backpropagation in recurrent neural networks is called *backpropagation through time* (BPTT). In our code, we did not have to worry about backpropagation since TensorFlow, which is the default backend for Keras calculated the gradients for us automatically, but let us see what is happening under the hood. Remember that the goal in backpropagation is to calculate the gradients of the error E with respect to $\mathbf{w}_x$, $\mathbf{w}_h$ and $\mathbf{w}_o$.

When we we were talking of the MSE and SSE error functions, we have seen that we resort to summing up the errors, and that this is good enough for machine learning. We can also just sum up the gradients for each training sample at a given point in time:

$$\frac{\partial E}{\partial w_i} = \sum_t \frac{\partial E_t}{\partial w_i}. \tag{7.14}$$

Let us see how this works in a whole example. Say, we want to calculate the gradient for E_2:

$$\frac{\partial E_2}{\partial \mathbf{w}_o} = \frac{\partial E_2}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{w}_o}. \tag{7.15}$$

This means that for $\mathbf{w}_o$ the time component plays no part. As expected, for $\mathbf{w}_h$ ($\mathbf{w}_x$ is similar) it is a bit different which is as follows:

$$\frac{\partial E_2}{\partial \mathbf{w}_h} = \frac{\partial E_2}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{w}_h}. \tag{7.16}$$

But remember that $h_2 = f_h(\mathbf{w}_h\mathbf{h}_1 + \mathbf{w}_x\mathbf{x}_2)$ which means the whole expression depends on $\mathbf{h}_1$, so if we want the derivative with respect to $\mathbf{w}_h$ we cannot treat it as a constant. The proper way to do it is to split the last term into a sum as follows:

$$\frac{\partial \mathbf{h}_2}{\partial \mathbf{w}_h} = \sum_{i=0}^{2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_i}\frac{\partial \mathbf{h}_i}{\partial \mathbf{w}_h}. \tag{7.17}$$

So, except for the summation, backpropagation through time is exactly the same as standard backpropagation. This simplicity of calculation is actually the reason why SRNs are more resistant to the vanishing gradient than a feedforward network with the same number of hidden layers. Let us address a final issue. The error function we have previously used was MSE, and this is a valid choice for regression and binary classification. A better choice for multi-class classification is the *cross-entropy error function*, which is defined as

$$CE = -\frac{1}{n}\sum_{i \in curr\,Batch} (t_i \ln y_i + (1 - y_i)\ln(1 - y_i)). \tag{7.18}$$

Where t is the target, y is the classifier outcome, i is the dummy variable which iterates over the current batch targets and outputs, and n is the number of all samples in the batch. The cross-entropy error function is derived from the log-likelihood, but this derivation is rather tedious and beyond our needs so we skip it. The cross-entropy is a more natural choice of error functions, but it is less straightforward to understand conceptually, so we used the MSE throughout this book, but you will want to use the CE for all multiclass classification tasks. The Keras code is `loss=categorical_crossentropy`, but feel free to browse all loss functions https://keras.io/losses/, you might be surprised to find some functions

which we will discuss in a different context can also be used as a loss or error function in neural network training. In fact, finding or defining a good loss function is often a very important part of getting a good accuracy with a deep learning model.

References

1. J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities. Proc. Nat. Acad. Sci. U.S.A **79**(8), 2554–2558 (1982)
2. S. Hochreiter, J. Schmidhuber, Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
3. J.L. Elman, Finding structure in time. Cogn. Sci. **14**, 179–211 (1990)
4. M.I. Jordan, Attractor dynamics and parallelism in a connectionist sequential machine, in *Proceedings of the 26th Annual International Conference on Machine Learning, Erlbaum, NJ, USA* (Cognitive Science Society, 1986), pp. 531–546
5. A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks* (Springer, New York, 2012)
6. A. Gulli, S. Pal, *Deep Learning with Keras* (Packt publishing, Birmingham, 2017)