CHAPTER

# 6 Neural Networks

"[M]achines of this character can behave in a very complicated manner when the number of units is large."

Alan Turing (1948) "Intelligent Machines", page 6

Neural networks are a fundamental computational tool for language processing, and a very old one. They are called neural because their origins lie in the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the biological neuron as a kind of computing element that could be described in terms of propositional logic. But the modern use in language processing no longer draws on these early biological inspirations.

Instead, a modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value. In this chapter we introduce the neural net applied to classification. The architecture we introduce is called a **feedforward network** because the computation proceeds iteratively from one layer of units to the next. The use of modern neural nets is often called **deep learning**, because modern networks are often **deep** (have many layers).

**feedforward**

**deep learning**

Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single 'hidden layer') can be shown to learn any function.

Neural net classifiers are different from logistic regression in another way. With logistic regression, we applied the regression classifier to many different tasks by developing many rich kinds of feature templates based on domain knowledge. When working with neural networks, it is more common to avoid most uses of rich hand-derived features, instead building neural networks that take raw tokens as inputs and learn to induce features as part of the process of learning to classify. We saw examples of this kind of representation learning for embeddings in Chapter 5, and we'll see lots of examples once we start studying deep transformers networks. Nets that are very deep are particularly good at representation learning. For that reason deep neural nets are the right tool for tasks that offer sufficient data to learn features automatically.

In this chapter we'll introduce feedforward networks as classifiers, first with hand-built features, and then using the embeddings that we studied in Chapter 5. In subsequent chapters we'll introduce many other kinds of neural models, most importantly the **transformer** and **attention**, (Chapter 8), but also **recurrent neural networks** (Chapter 13) and **convolutional neural networks** (Chapter 15). And in the next chapter we'll introduce the paradigm of neural large language models.

## 6.1 Units

The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.

At its heart, a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a **bias term**. Given a set of inputs $x_1...x_n$, a unit has a set of corresponding weights $w_1...w_n$ and a bias $b$, so the weighted sum $z$ can be represented as:

bias term

$$z = b + \sum_i w_i x_i \tag{6.1}$$

Often it's more convenient to express this weighted sum using vector notation; recall from linear algebra that a **vector** is, at heart, just a list or array of numbers. Thus we'll talk about $z$ in terms of a weight vector $w$, a scalar bias $b$, and an input vector $x$, and we'll replace the sum with the convenient **dot product**:

vector

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{6.2}$$

As defined in Eq. 6.2, $z$ is just a real valued number.

Finally, instead of using $z$, a linear function of $x$, as the output, neural units apply a non-linear function $f$ to $z$. We will refer to the output of this function as the **activation** value for the unit, $a$. Since we are just modeling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call $y$. So the value $y$ is defined as:

activation

$$y = a = f(z)$$

We'll discuss three popular non-linear functions $f$ below (the sigmoid, the tanh, and the rectified linear unit or ReLU) but it's pedagogically convenient to start with the **sigmoid** function since we saw it in Chapter 4:

sigmoid

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{6.3}$$

The sigmoid (shown in Fig. 6.1) has a number of advantages; it maps the output into the range $(0,1)$, which is useful in squashing outliers toward 0 or 1. And it's differentiable, which as we saw in Section 4.15 will be handy for learning.

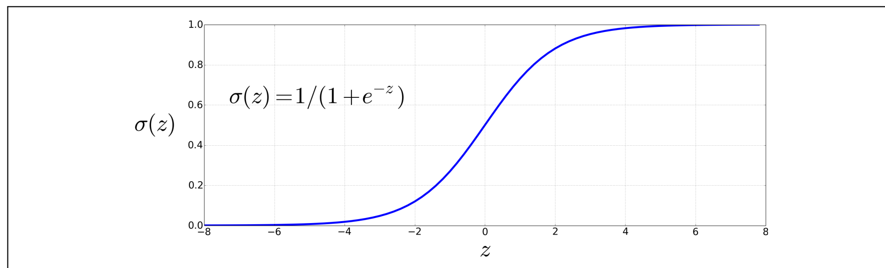


**Figure 6.1** The sigmoid function takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Substituting Eq. 6.2 into Eq. 6.3 gives us the output of a neural unit:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+\exp(-(\mathbf{w} \cdot \mathbf{x} + b))} \tag{6.4}$$

Fig. 6.2 shows a final schematic of a basic neural unit. In this example the unit takes 3 input values $x_1, x_2$, and $x_3$, and computes a weighted sum, multiplying each value by a weight ($w_1$, $w_2$, and $w_3$, respectively), adds them to a bias term $b$, and then passes the resulting sum through a sigmoid function to result in a number between 0 and 1.

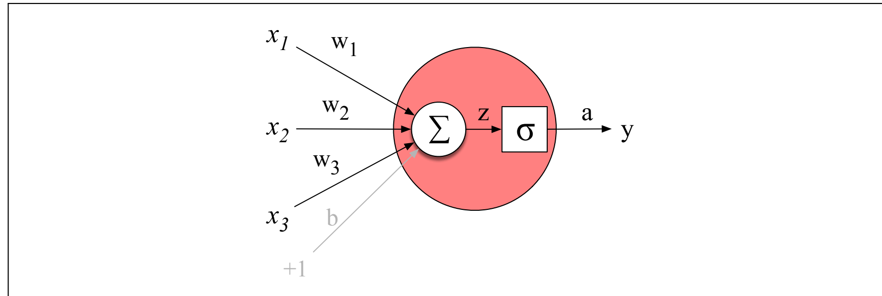


**Figure 6.2** A neural unit, taking 3 inputs $x_1$, $x_2$, and $x_3$ (and a bias $b$ that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, $z$, and the output of the sigmoid, $a$. In this case the output of the unit $y$ is the same as $a$, but in deeper networks we'll reserve $y$ to mean the final output of the entire network, leaving $a$ as the activation of an individual node.

Let's walk through an example just to get an intuition. Let's suppose we have a unit with the following weight vector and bias:

$$\begin{aligned} \mathbf{w} &= [0.2, 0.3, 0.9] \\ b &= 0.5 \end{aligned}$$

What would this unit do with the following input vector:

$$\mathbf{x} = [0.5, 0.6, 0.1]$$

The resulting output $y$ would be:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+e^{-(\mathbf{w} \cdot \mathbf{x}+b)}} = \frac{1}{1+e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1+e^{-0.87}} = .70$$

tanh

In practice, the sigmoid is not commonly used as an activation function. A function that is very similar but almost always better is the **tanh** function shown in Fig. 6.3a; tanh is a variant of the sigmoid that ranges from -1 to +1:

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{6.5}$$

ReLU

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**, shown in Fig. 6.3b. It's just the same as $z$ when $z$ is positive, and 0 otherwise:

$$y = \text{ReLU(z)} = max(z, 0) \tag{6.6}$$

These activation functions have different properties that make them useful for different language applications or network architectures. For example, the tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean. The rectifier function, on the other hand, has nice properties that

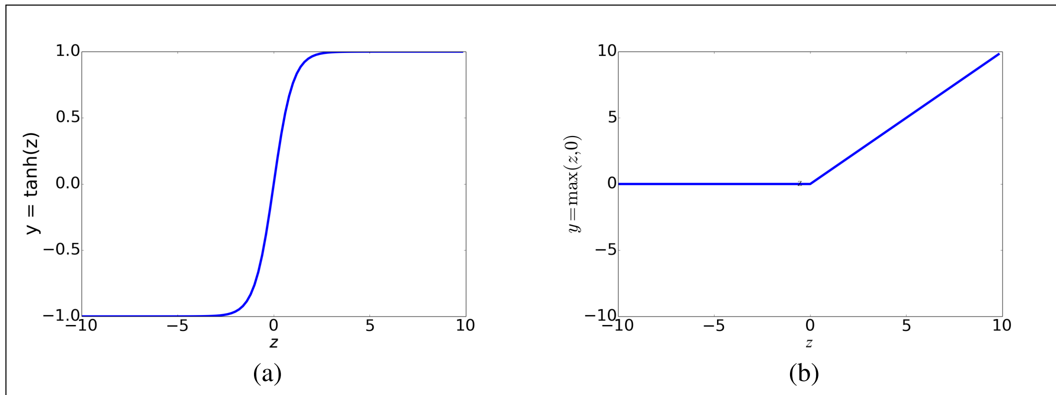


**Figure 6.3** The tanh and ReLU activation functions.

result from it being very close to linear. In the sigmoid or tanh functions, very high values of $z$ result in values of $y$ that are **saturated**, i.e., extremely close to 1, and have derivatives very close to 0. Zero derivatives cause problems for learning, because as we'll see in Section 6.6, we'll train networks by propagating an error signal backwards, multiplying gradients (partial derivatives) from each layer of the network; gradients that are almost 0 cause the error signal to get smaller and smaller until it is too small to be used for training, a problem called the **vanishing gradient** problem. Rectifiers don't have this problem, since the derivative of ReLU for high values of $z$ is 1 rather than very close to 0.

saturated

vanishing gradient

## 6.2 The XOR problem

Early in the history of neural networks it was realized that the power of neural networks, as with the real neurons that inspired them, comes from combining these units into larger networks.

One of the most clever demonstrations of the need for multi-layer networks was the proof by Minsky and Papert (1969) that a single neural unit cannot compute some very simple functions of its input. Consider the task of computing elementary logical functions of two inputs, like AND, OR, and XOR. As a reminder, here are the truth tables for those functions:

| AND | | | OR | | | XOR | | |
|---|---|---|---|---|---|---|---|---|
| x1 | x2 | y | x1 | x2 | y | x1 | x2 | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

perceptron

This example was first shown for the **perceptron**, which is a very simple neural unit that has a binary output and has a very simple step function as its non-linear activation function. The output $y$ of a perceptron is 0 or 1, and is computed as follows (using the same weight $\mathbf{w}$, input $\mathbf{x}$, and bias $b$ as in Eq. 6.2):

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases} \tag{6.7}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs; Fig. 6.4 shows the necessary weights.

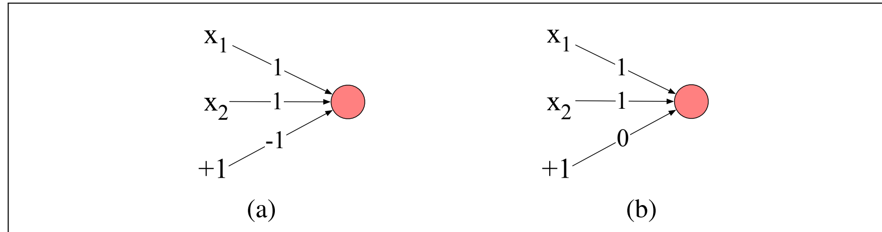


**Figure 6.4** The weights $w$ and bias $b$ for perceptrons for computing logical functions. The inputs are shown as $x_1$ and $x_2$ and the bias as a special node with value $+1$ which is multiplied with the bias weight $b$. (a) logical AND, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

It turns out, however, that it's not possible to build a perceptron to compute logical XOR! (It's worth spending a moment to give it a try!)

The intuition behind this important result relies on understanding that a perceptron is a linear classifier. For a two-dimensional input $x_1$ and $x_2$, the perceptron equation, $w_1x_1 + w_2x_2 + b = 0$ is the equation of a line. (We can see this by putting it in the standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$.) This line acts as a **decision boundary** in two-dimensional space in which the output 0 is assigned to all inputs lying on one side of the line, and the output 1 to all input points lying on the other side of the line. If we had more than 2 inputs, the decision boundary becomes a hyperplane instead of a line, but the idea is the same, separating the space into two categories.

decision boundary

Fig. 6.5 shows the possible logical inputs (`00`, `01`, `10`, and `11`) and the line drawn by one possible set of parameters for an AND and an OR classifier. Notice that there is simply no way to draw a line that separates the positive cases of XOR (01 and 10) from the negative cases (00 and 11). We say that XOR is not a **linearly separable** function. Of course we could draw a boundary with a curve, or some other function, but not a single line.

linearly separable

### 6.2.1 The solution: neural networks

While the XOR function cannot be calculated by a single perceptron, it can be calculated by a layered network of perceptron units. Rather than see this with networks of simple perceptrons, however, let's see how to compute XOR using two layers of ReLU-based units following Goodfellow et al. (2016). Fig. 6.6 shows a figure with the input being processed by two layers of neural units. The middle layer (called $h$) has two units, and the output layer (called $y$) has one unit. A set of weights and biases are shown that allows the network to correctly compute the XOR function.

Let's walk through what happens with the input $\mathbf{x}$ = [0, 0]. If we multiply each input value by the appropriate weight, sum, and then add the bias $b$, we get the vector [0, -1], and we then apply the rectified linear transformation to give the output of the $\mathbf{h}$ layer as [0, 0]. Now we once again multiply by the weights, sum, and add the bias (0 in this case) resulting in the value 0. The reader should work through the computation of the remaining 3 possible input pairs to see that the resulting $y$ values are 1 for the inputs [0, 1] and [1, 0] and 0 for [0, 0] and [1, 1].

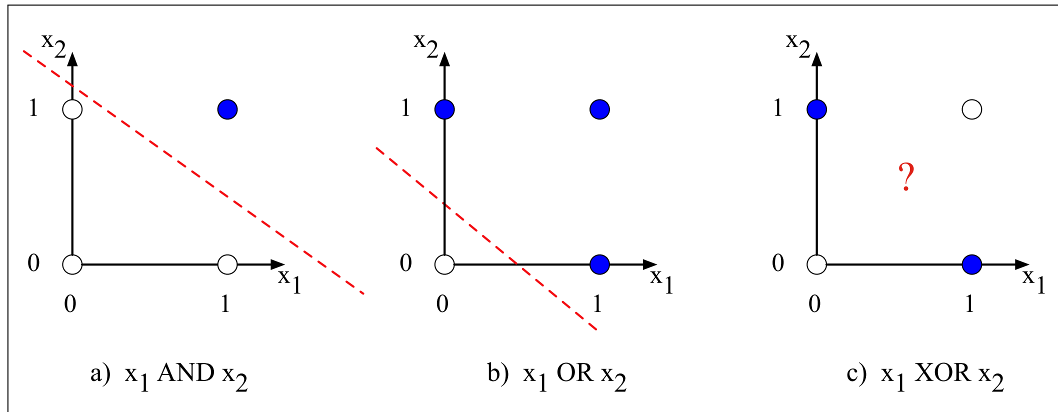


**Figure 6.5** The functions AND, OR, and XOR, represented with input $x_1$ on the x-axis and input $x_2$ on the y-axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

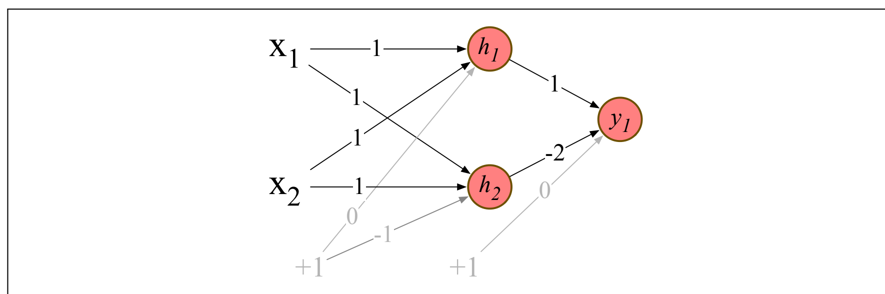


**Figure 6.6** XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them $h_1$, $h_2$ ($h$ for "hidden layer") and $y_1$. As before, the numbers on the arrows represent the weights $w$ for each unit, and we represent the bias $b$ as a weight on a unit clamped to +1, with the bias weights/units in gray.

It's also instructive to look at the intermediate results, the outputs of the two hidden nodes $h_1$ and $h_2$. We showed in the previous paragraph that the $\mathbf{h}$ vector for the inputs $\mathbf{x}$ = [0, 0] was [0, 0]. Fig. 6.7b shows the values of the $\mathbf{h}$ layer for all 4 inputs. Notice that hidden representations of the two input points $\mathbf{x}$ = [0, 1] and $\mathbf{x}$ = [1, 0] (the two cases with XOR output = 1) are merged to the single point $\mathbf{h}$ = [1, 0]. The merger makes it easy to linearly separate the positive and negative cases of XOR. In other words, we can view the hidden layer of the network as forming a representation of the input.

In this example we just stipulated the weights in Fig. 6.6. But for real examples the weights for neural networks are learned automatically using the error backpropagation algorithm to be introduced in Section 6.6. That means the hidden layers will learn to form useful representations. This intuition, that neural networks can automatically learn useful representations of the input, is one of their key advantages, and one that we will return to again and again in later chapters.

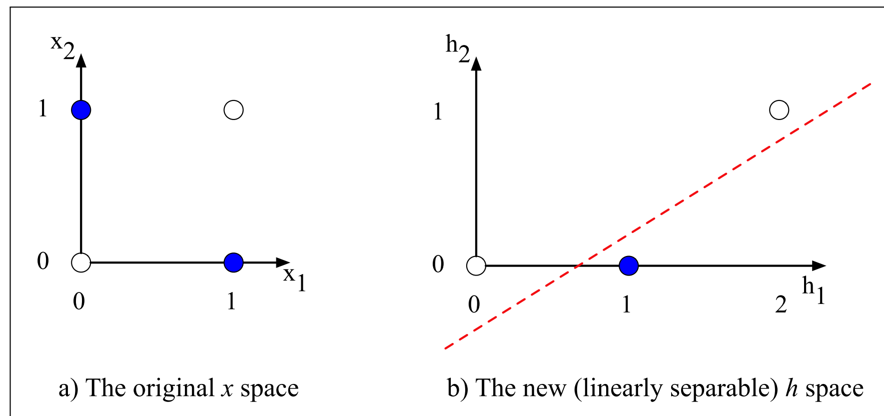


**Figure 6.7** The hidden layer forming a new representation of the input. (b) shows the representation of the hidden layer, **h**, compared to the original input representation **x** in (a). Notice that the input point [0, 1] has been collapsed with the input point [1, 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

## 6.3 Feedforward Neural Networks

**feedforward network**

Let's now walk through a slightly more formal presentation of the simplest kind of neural network, the **feedforward network**. A feedforward network is a multilayer network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. (In Chapter 13 we'll introduce networks with cycles, called **recurrent neural networks**.)

**multi-layer perceptrons**
**MLP**

For historical reasons multilayer networks, especially feedforward networks, are sometimes called **multi-layer perceptrons** (or **MLP**s); this is a technical misnomer, since the units in modern multilayer networks aren't perceptrons (perceptrons have a simple step-function as their activation function, but modern networks are made up of units with many kinds of non-linearities like ReLUs and sigmoids), but at some point the name stuck.

Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units.

Fig. 6.8 shows a picture. The input layer $\mathbf{x}$ is a vector of simple scalar values just as we saw in Fig. 6.2.

**hidden layer**

The core of the neural network is the **hidden layer h** formed of **hidden units** $\mathbf{h}_i$, each of which is a neural unit as described in Section 6.1, taking a weighted sum of its inputs and then applying a non-linearity. In the standard architecture, each layer

**fully-connected**

is **fully-connected**, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units.

Recall that a single hidden unit has as parameters a weight vector and a bias. We represent the parameters for the entire hidden layer by combining the weight vector and bias for each unit $i$ into a single weight matrix $\mathbf{W}$ and a single bias vector $b$ for the whole layer (see Fig. 6.8). Each element $\mathbf{W}_{ji}$ of the weight matrix $\mathbf{W}$ represents the weight of the connection from the $i$th input unit $x_i$ to the $j$th hidden unit $h_j$.

The advantage of using a single matrix $\mathbf{W}$ for the weights of the entire layer is that now the hidden layer computation for a feedforward network can be done very

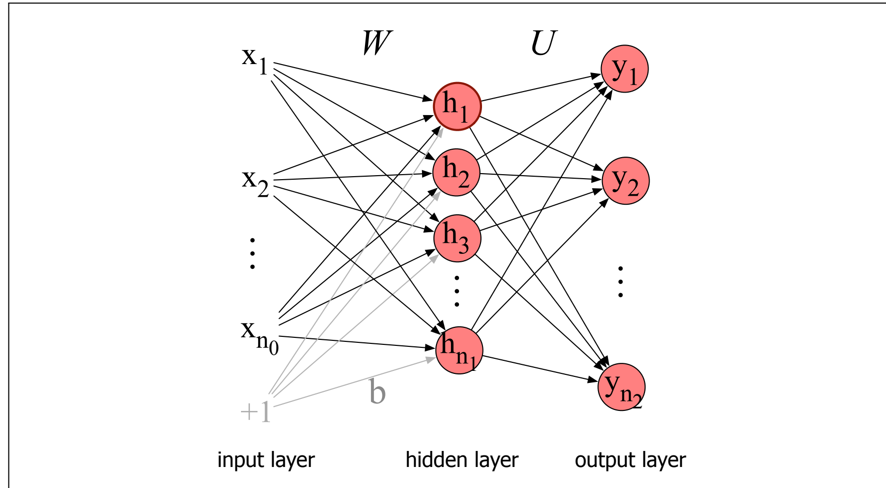


**Figure 6.8** A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

efficiently with simple matrix operations. In fact, the computation only has three steps: multiplying the weight matrix by the input vector $\mathbf{x}$, adding the bias vector $\mathbf{b}$, and applying the activation function $g$ (such as the sigmoid, tanh, or ReLU activation function defined above).

The output of the hidden layer, the vector $\mathbf{h}$, is thus the following (for this example we'll use the sigmoid function $\sigma$ as our activation function):

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}+\mathbf{b}) \tag{6.8}$$

Notice that we're applying the $\sigma$ function here to a vector, while in Eq. 6.3 it was applied to a scalar. We're thus allowing $\sigma(\cdot)$, and indeed any activation function $g(\cdot)$, to apply to a vector element-wise, so $g[z_1,z_2,z_3] = [g(z_1),g(z_2),g(z_3)]$.

Let's introduce some constants to represent the dimensionalities of these vectors and matrices. We'll refer to the input layer as layer 0 of the network, and have $n_0$ represent the number of inputs, so $\mathbf{x}$ is a vector of real numbers of dimension $n_0$, or more formally $\mathbf{x} \in \mathbb{R}^{n_0}$, a column vector of dimensionality $[n_0 \times 1]$. Let's call the hidden layer layer 1 and the output layer layer 2. The hidden layer has dimensionality $n_1$, so $\mathbf{h} \in \mathbb{R}^{n_1}$ and also $\mathbf{b} \in \mathbb{R}^{n_1}$ (since each hidden unit can take a different bias value). And the weight matrix $\mathbf{W}$ has dimensionality $\mathbf{W} \in \mathbb{R}^{n_1 \times n_0}$, i.e. $[n_1 \times n_0]$.

Take a moment to convince yourself that the matrix multiplication in Eq. 6.8 will compute the value of each $\mathbf{h}_j$ as $\sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j\right)$.

As we saw in Section 6.2, the resulting value $\mathbf{h}$ (for *hidden* but also for *hypothesis*) forms a *representation* of the input. The role of the output layer is to take this new representation $\mathbf{h}$ and compute a final output. This output could be a real-valued number, but in many cases the goal of the network is to make some sort of classification decision, and so we will focus on the case of classification.

If we are doing a binary task like sentiment classification, we might have a single output node, and its scalar value $y$ is the probability of positive versus negative sentiment. If we are doing multinomial classification, such as assigning a part-of-speech tag, we might have one output node for each potential part-of-speech, whose output value is the probability of that part-of-speech, and the values of all the output nodes must sum to one. The output layer is thus a vector $\mathbf{y}$ that gives a probability distribution across the output nodes.

Let's see how this happens. Like the hidden layer, the output layer has a weight matrix (let's call it $\mathbf{U}$), but some models don't include a bias vector $\mathbf{b}$ in the output layer, so we'll simplify by eliminating the bias vector in this example. The weight matrix is multiplied by its input vector ($\mathbf{h}$) to produce the intermediate output $\mathbf{z}$:

$$\mathbf{z} = \mathbf{U}\mathbf{h}$$

There are $n_2$ output nodes, so $\mathbf{z} \in \mathbb{R}^{n_2}$, weight matrix $\mathbf{U}$ has dimensionality $\mathbf{U} \in \mathbb{R}^{n_2 \times n_1}$, and element $\mathbf{U}_{ij}$ is the weight from unit $j$ in the hidden layer to unit $i$ in the output layer.

However, $\mathbf{z}$ can't be the output of the classifier, since it's a vector of real-valued numbers, while what we need for classification is a vector of probabilities. There is a convenient function for **normalizing** a vector of real values, by which we mean converting it to a vector that encodes a probability distribution (all the numbers lie between 0 and 1 and sum to 1): the **softmax** function that we saw on page 79 of Chapter 4. More generally for any vector $\mathbf{z}$ of dimensionality $d$, the softmax is defined as:

normalizing

softmax

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d}\exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{6.9}$$

Thus for example given a vector

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1], \tag{6.10}$$

the softmax function will normalize it to a probability distribution (shown rounded):

$$\text{softmax}(\mathbf{z}) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010] \tag{6.11}$$

You may recall that we used softmax to create a probability distribution from a vector of real-valued numbers (computed from summing weights times features) in the multinomial version of logistic regression in Chapter 4.

That means we can think of a neural network classifier with one hidden layer as building a vector $\mathbf{h}$ which is a hidden layer representation of the input, and then running standard multinomial logistic regression on the features that the network develops in $\mathbf{h}$. By contrast, in Chapter 4 the features were mainly designed by hand via feature templates. So a neural network is like multinomial logistic regression, but (a) with many layers, since a deep neural network is like layer after layer of logistic regression classifiers; (b) with those intermediate layers having many possible activation functions (tanh, ReLU, sigmoid) instead of just sigmoid (although we'll continue to use $\sigma$ for convenience to mean any activation function); (c) rather than forming the features by feature templates, the prior layers of the network induce the feature representations themselves.

Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector $\mathbf{x}$, outputs a probability distribution $\mathbf{y}$, and is parameterized by weight matrices $\mathbf{W}$ and $\mathbf{U}$ and a bias vector $\mathbf{b}$:

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})\\ \mathbf{z} &= \mathbf{U}\mathbf{h}\\ \mathbf{y} &= \text{softmax}(\mathbf{z})\end{aligned} \tag{6.12}$$

And just to remember the shapes of all our variables, $\mathbf{x} \in \mathbb{R}^{n_0}$, $\mathbf{h} \in \mathbb{R}^{n_1}$, $\mathbf{b} \in \mathbb{R}^{n_1}$, $\mathbf{W} \in \mathbb{R}^{n_1 \times n_0}$, $\mathbf{U} \in \mathbb{R}^{n_2 \times n_1}$, and the output vector $\mathbf{y} \in \mathbb{R}^{n_2}$. We'll call this network a 2-layer network (we traditionally don't count the input layer when numbering layers, but do count the output layer). So by this terminology logistic regression is a 1-layer network.

### 6.3.1 More details on feedforward networks

Let's now set up some notation to make it easier to talk about deeper networks of depth more than 2. We'll use superscripts in square brackets to mean layer numbers, starting at 0 for the input layer. So $\mathbf{W}^{[1]}$ will mean the weight matrix for the (first) hidden layer, and $\mathbf{b}^{[1]}$ will mean the bias vector for the (first) hidden layer. $n_j$ will mean the number of units at layer $j$. We'll use $g(\cdot)$ to stand for the activation function, which will tend to be ReLU or tanh for intermediate layers and softmax for output layers. We'll use $\mathbf{a}^{[i]}$ to mean the output from layer $i$, and $\mathbf{z}^{[i]}$ to mean the combination of previous layer output, weights and biases $\mathbf{W}^{[i]}\mathbf{a}^{[i-1]}+\mathbf{b}^{[i]}$. The 0th layer is for inputs, so we'll refer to the inputs $\mathbf{x}$ more generally as $\mathbf{a}^{[0]}$.

Thus we can re-represent our 2-layer net from Eq. 6.12 as follows:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{a}^{[0]}+\mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]}+\mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= g^{[2]}(\mathbf{z}^{[2]}) \\
\hat{\mathbf{y}} &= \mathbf{a}^{[2]}
\end{aligned} \tag{6.13}$$

Note that with this notation, the equations for the computation done at each layer are the same. The algorithm for computing the forward step in an n-layer feedforward network, given the input vector $a^{[0]}$ is thus simply:

**for** $i$ **in** 1,...,n
  $\mathbf{z}^{[i]} = \mathbf{W}^{[i]}\,\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$
  $\mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]})$
$\hat{\mathbf{y}} = \mathbf{a}^{[n]}$

It's often useful to have a name for the final set of activations right before the final softmax. So however many layers we have, we'll generally call the unnormalized values in the final vector $\mathbf{z}^{[n]}$, the vector of scores right before the final softmax, the **logits** (see Eq. 4.7).

logits

**The need for non-linear activation functions** One of the reasons we use non-linear activation functions for each layer in a neural network is that if we did not, the resulting network is exactly equivalent to a single-layer network. Let's see why this is true. Imagine the first two layers of such a network of purely linear layers:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x}+\mathbf{b}^{[1]} \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]}+\mathbf{b}^{[2]}
\end{aligned}$$

We can rewrite the function that the network is computing as:

$$\begin{aligned}
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]}+\mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}(\mathbf{W}^{[1]}\mathbf{x}+\mathbf{b}^{[1]})+\mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}\mathbf{W}^{[1]}\mathbf{x}+\mathbf{W}^{[2]}\mathbf{b}^{[1]}+\mathbf{b}^{[2]} \\
&= \mathbf{W}'\mathbf{x}+\mathbf{b}'
\end{aligned} \tag{6.14}$$

This generalizes to any number of layers. So without non-linear activation functions, a multilayer network is just a notational variant of a single layer network with a different set of weights, and we lose all the representational power of multilayer networks.

**Replacing the bias unit** In describing networks, we will sometimes use a slightly simplified notation that represents exactly the same function without referring to an explicit bias node $b$. Instead, we add a dummy node $\mathbf{a}_0$ to each layer whose value will always be 1. Thus layer 0, the input layer, will have a dummy node $\mathbf{a}_0^{[0]} = 1$, layer 1 will have $\mathbf{a}_0^{[1]} = 1$, and so on. This dummy node still has an associated weight, and that weight represents the bias value $b$. For example instead of an equation like

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{6.15}$$

we'll use:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}) \tag{6.16}$$

But now instead of our vector $\mathbf{x}$ having $n_0$ values: $\mathbf{x} = \mathbf{x}_1, \ldots, \mathbf{x}_{n_0}$, it will have $n_0 + 1$ values, with a new 0th dummy value $\mathbf{x}_0 = 1$: $\mathbf{x} = \mathbf{x}_0, \ldots, \mathbf{x}_{n_0}$. And instead of computing each $\mathbf{h}_j$ as follows:

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\,\mathbf{x}_i + \mathbf{b}_j\right), \tag{6.17}$$

we'll instead use:

$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\,\mathbf{x}_i\right), \tag{6.18}$$

where the value $\mathbf{W}_{j0}$ replaces what had been $\mathbf{b}_j$. Fig. 6.9 shows a visualization.

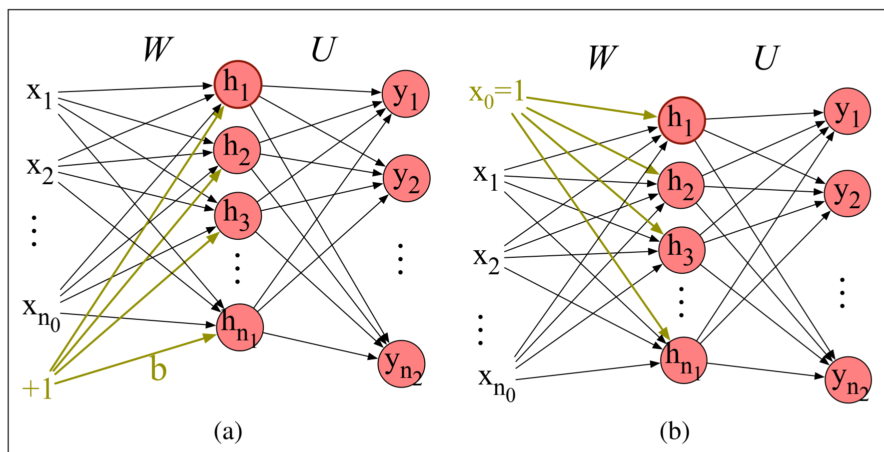


**Figure 6.9** Replacing the bias node (shown in a) with $x_0$ (b).

We'll continue showing the bias as $b$ when we go over the learning algorithm in Section 6.6, but going forward in the book, for most figures and some equations we'll use this simplified notation without explicit bias terms.

# 6.4 Feedforward networks for NLP: Classification

Let's see how to apply feedforward networks to NLP classification tasks. In practice, simple feedforward networks aren't the way we do text classification; for real applications we would use more sophisticated architectures like the BERT transformers

of Chapter 9. Nonetheless seeing a feedforward network text classifier will let us introduce key ideas that will play a role throughout the rest of the book, including the ideas of the **embedding matrix**, representation **pooling**, and **representation learning**.

But before introducing any of these ideas, let's start with a classifier by making only minimal change from the sentiment classifiers we saw in Chapter 4. Like them, we'll take hand-built features, pass them through a classifier, and produce a class probability. The only difference is that we'll use a neural network instead of logistic regression as the classifier.

### 6.4.1 Neural net classifiers with hand-built features

Let's begin with a simple 2-layer sentiment classifier by taking our logistic regression classifier from Chapter 4, which corresponds to a 1-layer network, and just adding a hidden layer. The input element $\mathbf{x}_i$ can be scalar features like those in Fig. 4.2, e.g., $\mathbf{x}_1$ = count(words ∈ doc), $\mathbf{x}_2$ = count(positive lexicon words ∈ doc), $\mathbf{x}_3$ = 1 if "no" ∈ doc, and so on, for a total of $d$ features. And the output layer $\hat{\mathbf{y}}$ could have two nodes (one each for positive and negative), or 3 nodes (positive, negative, neutral), in which case $\hat{\mathbf{y}}_1$ would be the estimated probability of positive sentiment, $\hat{\mathbf{y}}_2$ the probability of negative and $\hat{\mathbf{y}}_3$ the probability of neutral. The resulting equations would be just what we saw above for a 2-layer network (as always, we'll continue to use the $\sigma$ to stand for any non-linearity, whether sigmoid, ReLU or other).

$$
\begin{aligned}
\mathbf{x} &= [\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_d] \quad \text{(each } \mathbf{x}_i \text{ is a hand-designed feature)} \\
\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{U}\mathbf{h} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})
\end{aligned}
\tag{6.19}
$$

Fig. 6.10 shows a sketch of this architecture. As we mentioned earlier, adding this hidden layer to our logistic regression classifier allows the network to represent the non-linear interactions between features. This alone might give us a better sentiment classifier.

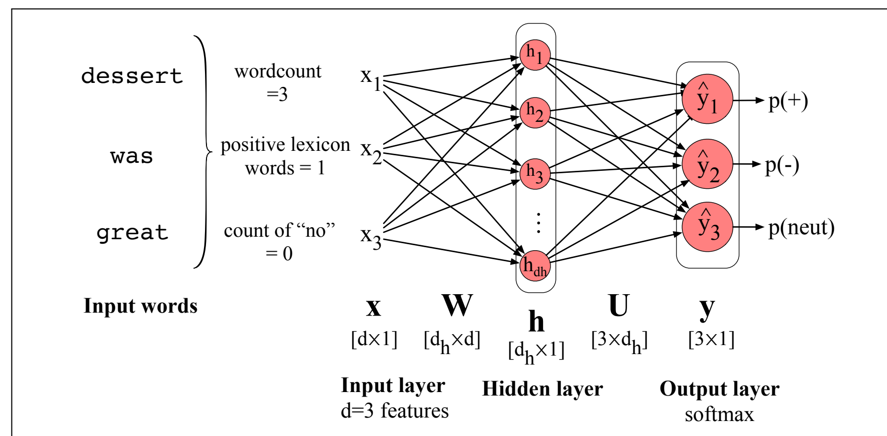


**Figure 6.10** Feedforward network sentiment analysis using traditional hand-built features of the input text.

### 6.4.2 Vectorizing for parallelizing inference

While Eq. 6.19 shows how to classify a single example $x$, in practice we want to efficiently classify an entire test set of $m$ examples. We do this by vectorizing the process, just as we saw with logistic regression; instead of using for-loops to go through each example, we'll use matrix multiplication to do the entire computation of an entire test set at once. First, we pack all the input feature vectors for each input $x$ into a single input matrix $\mathbf{X}$, with each row $i$ a row vector consisting of the features for input example $x^{(i)}$ (i.e., the vector $\mathbf{x}^{(i)}$). If the dimensionality of our input feature vector is $d$, $\mathbf{X}$ will be a matrix of shape $[m \times d]$.

Because we are now modeling each input as a row vector rather than a column vector, we also need to slightly modify Eq. 6.19. $\mathbf{X}$ is of shape $[m \times d]$ and $\mathbf{W}$ is of shape $[d_h \times d]$, so we'll reorder how we multiply $\mathbf{X}$ and $\mathbf{W}$ and transpose $\mathbf{W}$ so they correctly multiply to yield a matrix $\mathbf{H}$ of shape $[m \times d_h]$. [1]

The bias vector $\mathbf{b}$ from Eq. 6.19 of shape $[1 \times d_h]$ will now have to be replicated into a matrix of shape $[m \times d_h]$. We'll need to similarly reorder the next step and transpose $\mathbf{U}$. Finally, our output matrix $\hat{\mathbf{Y}}$ will be of shape $[m \times 3]$ (or more generally $[m \times d_o]$, where $d_o$ is the number of output classes), with each row $i$ of our output matrix $\hat{\mathbf{Y}}$ consisting of the output vector $\hat{\mathbf{y}}^{(i)}$. Here are the final equations for computing the output class distribution for an entire test set:

$$
\begin{aligned}
\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^\intercal + \mathbf{b}) \\
\mathbf{Z} &= \mathbf{H}\mathbf{U}^\intercal \\
\hat{\mathbf{Y}} &= \text{softmax}(\mathbf{Z})
\end{aligned}
\tag{6.20}
$$

In this book, we'll sometimes see orderings like $\mathbf{W}\mathbf{X} + \mathbf{b}$ and sometimes $\mathbf{X}\mathbf{W} + \mathbf{b}$. That's why it's always important to be very aware of the shapes of your weight matrices participating in any given equation.

## 6.5 Embeddings as the input to neural net classifiers

While hand-built features are a traditional way to design classifiers, most applications of neural networks for NLP don't use hand-built human-engineered features as inputs. Instead, we draw on deep learning's ability to learn features from the data by representing tokens as embeddings. For this section we'll represent each token by its static word2vec or GloVe embeddings that we saw how to compute in Chapter 5. By static embedding, we mean that each token is represented by a fixed vector that we train once, and then just put into a big dictionary. When we want to refer to that token, we grab its embedding out of the dictionary.

However when we apply neural models to the task of language modeling (as we'll see in Chapter 8) the situation is more complex, and we'll use a more powerful kind of embedding called a *contextual embedding*. Contextual embeddings are different for each time a word occurs in a different context. Furthermore, we'll have the network learn these embeddings as part of the task of word prediction.

So let's explore the text classification domain above, but using static embeddings as features instead of the hand-designed features. Let's focus on the inference stage,

[1] Note that we could have kept the original order of our products if we had instead made our input matrix $\mathbf{X}$ represent each input as a column vector instead of a row vector, making it of shape $[d \times m]$. But representing inputs as row vectors is convenient and common in neural network models.

in which we have already learned embeddings for all the input tokens. An embedding is a vector of dimension $d$ that represents the input token. The dictionary of static embeddings in which we store these embeddings is the **embedding matrix** $\mathbf{E}$. Each row of the embedding matrix represents each token of the vocabulary $V$ as a (row) vector of dimensionality $d$. Since $\mathbf{E}$ has a row for each of the $|V|$ tokens in the vocabulary, $\mathbf{E}$ has shape $[|V| \times d]$. This embedding matrix $\mathbf{E}$ plays a role whenever we are using embeddings as input to neural NLP systems, including in the transformer-based large language models we will introduce over the next chapters.

embedding matrix

Given an input token string like `dessert was great` we first convert the tokens into vocabulary indices (these were created when we first tokenized the input using BPE or SentencePiece). So the representation of `dessert was great` might be $\mathbf{w} = [3, 9824, 226]$. Next we use indexing to select the corresponding rows from $\mathbf{E}$ (row 3, row 9824, row 226).

Another way to think about selecting token embeddings from the embedding matrix is to represent input tokens as one-hot vectors of shape $[1 \times |V|]$, i.e., with one dimension for each word in the vocabulary. Recall that in a **one-hot vector** all the elements are 0 except one, the element whose dimension is the word's index in the vocabulary, which has value 1. So if the word "dessert" has index 3 in the vocabulary, $x_3 = 1$, and $x_i = 0 \ \forall i \neq 3$, as shown here:

one-hot vector

```
[0 0 1 0 0 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...   ... |V|
```

Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant row vector for word $i$, resulting in the embedding for word $i$, as depicted in Fig. 6.11.

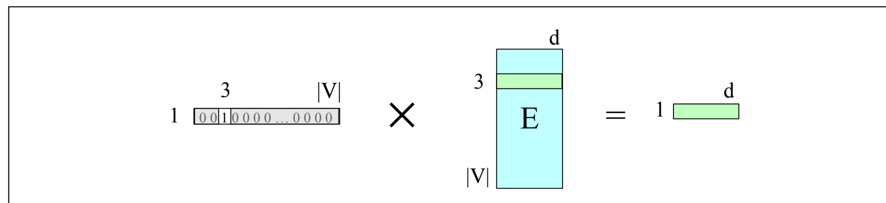


**Figure 6.11** Selecting the embedding vector for word $V_3$ by multiplying the embedding matrix $\mathbf{E}$ with a one-hot vector with a 1 in index 3.

We can extend this idea to represent the entire input token sequence as a matrix of one-hot vectors, one for each of the $N$ input positions as shown in Fig. 6.12.

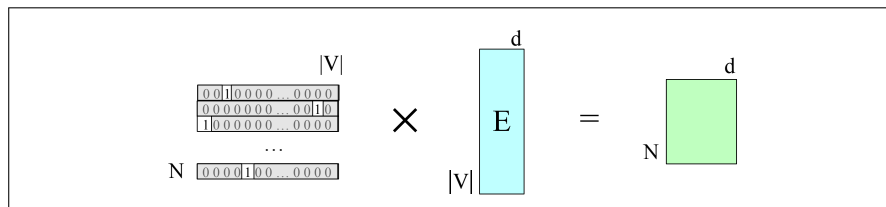


**Figure 6.12** Selecting the embedding matrix for the input sequence of token ids $W$ by multiplying a one-hot matrix corresponding to $W$ by the embedding matrix $\mathbf{E}$.

We now need to classify this input of $N$ $[1 \times d]$ embeddings, representing a window of $N$ tokens, into a single class (like positive or negative).

There are two common ways to pass embeddings to a classifier: **concatenation** and **pooling**. First, we can take this input of shape $[N \times d]$ and reshape it by **concatenating** all the input vectors into one very long vector of shape $[1 \times dN]$. Then

we pass this input to our classifier and let it make its decision. This gives us lots of information, at the cost of using a pretty large network. Second, we can **pool** the $N$ embeddings into a single embedding and then pass that single pooled embedding to the classifier. Pooling gives us less information than would have been present in all the original embeddings, but has the advantage of being small and efficient and is especially useful in tasks for which we don't care as much about the original word order. Let's give an example of each: pooling for the sentiment task, and concatenation for the language modeling task.

pool

**Pooling input embeddings for sentiment** So let's begin with seeing how pooling can work for the sentiment classification task. The intuition of pooling is that for sentiment, the exact position of the input (is some word like `great` the first word? the second word?) is less important than the identity of the word itself.

A pooling function is a way to turn a set of embeddings into a single embedding. For example, for a text with $N$ input words/tokens $w_1,...,w_N$, we want to turn the $N$ row embeddings $\mathbf{e}(w_1),...,\mathbf{e}(w_N)$ (each of dimensionality $d$) into a single embedding also of dimensionality $d$.

mean-pooling

There are various ways to pool. The simplest is **mean-pooling**: taking the mean by summing the embeddings and then dividing by $N$:

$$\mathbf{x}_{mean} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{e}(w_i) \tag{6.21}$$

Here are the equations for this classifier assuming mean pooling:

$$\begin{aligned}
\mathbf{x} &= \text{mean}(\mathbf{e}(w_1),\mathbf{e}(w_2),\ldots,\mathbf{e}(w_n)) \\
\mathbf{h} &= \sigma(\mathbf{xW}+\mathbf{b}) \\
\mathbf{z} &= \mathbf{hU} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})
\end{aligned} \tag{6.22}$$

The architecture is sketched in Fig. 6.13, where we also give the shapes for all the relevant matrices.

max-pooling

There are many other options for pooling, like **max-pooling**, in which case for each dimension we take the element-wise max over all the inputs. The element-wise max of a set of $N$ vectors is a new vector whose $k$th element is the max of the $k$th elements of all the $N$ vectors.

**Concatenating input embeddings for language modeling** For sentiment analysis we saw how to generate an output vector with probabilities over three classes: positive, negative, or neutral, given as input a window of $N$ input tokens, by first pooling those token embeddings into a single embedding vector.

Now let's consider **language modeling**: predicting upcoming words from prior words. In this task we are given the same window of $N$ input tokens, but our task now is to predict the next token that should follow the window. We'll sketch a simple feedforward neural language model, drawing on an algorithm first introduced by Bengio et al. (2003). The feedforward language model introduces many of the important concepts of large language modeling that we will return to in Chapter 7 and Chapter 8.

Neural language models have many advantages over the n-gram language models of Chapter 3. Neural language models can handle much longer histories, can generalize better over contexts of similar words, and are far more accurate at word-

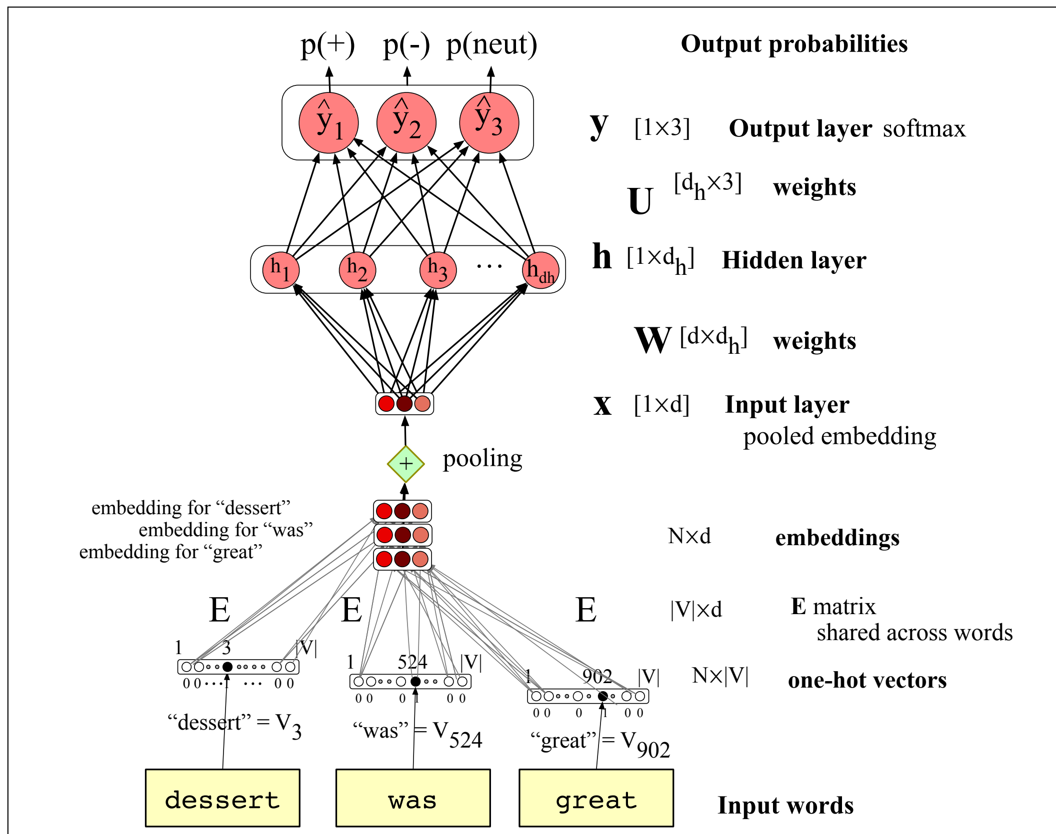


**Figure 6.13** Feedforward network sentiment analysis using a pooled embedding of the input words. At each timestep the network computes a $d$-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix **E**), and pools the resulting $N$ embeddings to get a single embedding that represents the context window as the layer **e**.

prediction. On the other hand, neural net language models are slower, more complex, need vast amounts of energy to train, and are less interpretable than n-gram models, so for some smaller tasks an n-gram language model is still the right tool.

A feedforward neural language model is a feedforward network that takes as input at time $t$ a representation of some number of previous words ($w_{t-1}, w_{t-2}$, etc.) and outputs a probability distribution over possible next words. Thus—like the n-gram LM—the feedforward neural LM approximates the probability of a word given the entire prior context $P(w_t|w_{1:t-1})$ by approximating based on the $N-1$ previous words:

$$P(w_t|w_1,\dots,w_{t-1}) \approx P(w_t|w_{t-N+1},\dots,w_{t-1}) \tag{6.23}$$

In the following examples we'll use a 4-gram example, so we'll show a neural net to estimate the probability $P(w_t = i|w_{t-3}, w_{t-2}, w_{t-1})$.

Neural language models represent words in this prior context by their **embeddings**, rather than just by their word identity as used in n-gram language models. Using embeddings allows neural language models to generalize better to unseen data. For example, suppose we've seen this sentence in training:

I have to make sure that the cat gets fed.

but have never seen the words "gets fed" after the word "dog". Our test set has the

prefix "I forgot to make sure that the dog gets". What's the next word? An n-gram language model will predict "fed" after "that the cat gets", but not after "that the dog gets". But a neural LM, knowing that "cat" and "dog" have similar embeddings, will be able to generalize from the "cat" context to assign a high enough probability to "fed" even after seeing "dog".

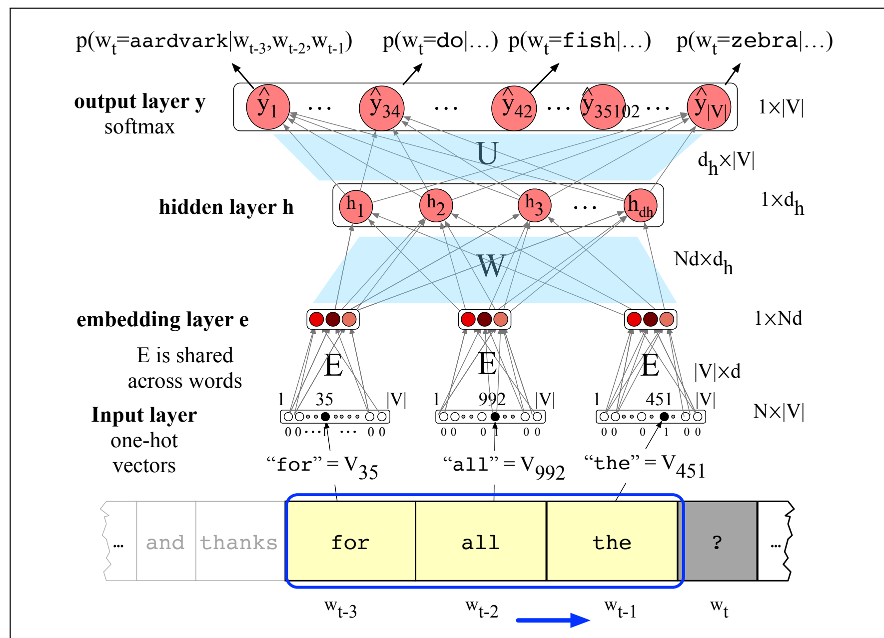


**Figure 6.14** Forward inference in a feedforward neural language model. At each timestep $t$ the network computes a $d$-dimensional embedding for each of the $N = 3$ context tokens (by multiplying a one-hot vector by the embedding matrix $\mathbf{E}$), and concatenates the three to get the embedding $\mathbf{e}$. This embedding $\mathbf{e}$ is multiplied by weight matrix $\mathbf{W}$ and then an activation function is applied element-wise to produce the hidden layer $\mathbf{h}$, which is then multiplied by another weight matrix $\mathbf{U}$. A softmax layer predicts at each output node $i$ the probability that the next word $w_t$ will be vocabulary word $V_i$. We show the context window size $N$ as 3 just to fit on the page, but in practice language modeling requires a much longer context.

This prediction task requires an output vector that expresses $|V|$ probabilities: one probability value for each possible next token. We might have a vocabulary between 60,000 and 300,000 tokens, so the output vector for the task of language modeling is much longer than 3. Another difference for language modeling is that instead of pooling the embeddings of the $N$ input tokens to create a single embedding, we concatenate the inputs into one very long input vector. To predict the next token, it helps to know each of the preceding tokens and what order they were in.

Fig. 6.14 shows the language modeling task, sketched with a very short context window of $N = 3$ just to fit on the page. These 3 embedding vectors are concatenated to produce $\mathbf{e}$, the embedding layer. This is multiplied by a weight matrix $\mathbf{W}$ to produce a hidden layer, and another weight matrix $\mathbf{U}$ to produce an output layer whose softmax gives a probability distribution over words. For example $y_{42}$, the value of output node 42, is the probability of the next word $w_t$ being $V_{42}$, the vocabulary word with index 42 (which is the word 'fish' in our example).

The equations for a simple feedforward neural language model with a window

size of 3, given one-hot input vectors for each input context word, are:

$$\begin{aligned} \mathbf{e} &= [\mathbf{Ex_{t-3}}; \mathbf{Ex_{t-2}}; \mathbf{Ex_{t-1}}] \\ \mathbf{h} &= \sigma(\mathbf{We}+\mathbf{b}) \\ \mathbf{z} &= \mathbf{Uh} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{6.24}$$

Note thatwe we use semicolons to mean concatenation of vectors, so we form the embedding layer $\mathbf{e}$ by concatenating the 3 embeddings for the three context vectors.

We'll return to this idea of using neural networks to do language modeling in Chapter 7 and Chapter 8 when we introduce transformer language models.

## 6.6 Training Neural Nets

A feedforward neural net is an instance of supervised machine learning in which we know the correct output $y$ for each observation $x$. What the system produces, via Eq. 6.13, is $\hat{y}$, the system's estimate of the true $y$. The goal of the training procedure is to learn parameters $\mathbf{W}^{[i]}$ and $\mathbf{b}^{[i]}$ for each layer $i$ that make $\hat{y}$ for each training observation as close as possible to the true $y$.

In general, we do all this by drawing on the methods we introduced in Chapter 4 for logistic regression, so the reader should be comfortable with that chapter before proceeding. We'll explore the algorithm on simple generic networks rather than networks designed for sentiment or language modeling.

First, we'll need a **loss function** that models the distance between the system output and the gold output, and it's common to use the loss function used for logistic regression, the **cross-entropy loss**.

Second, to find the parameters that minimize this loss function, we'll use the **gradient descent** optimization algorithm introduced in Chapter 4.

Third, gradient descent requires knowing the **gradient** of the loss function, the vector that contains the partial derivative of the loss function with respect to each of the parameters. In logistic regression, for each observation we could directly compute the derivative of the loss function with respect to an individual $w$ or $b$. But for neural networks, with millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer. How do we partial out the loss over all those intermediate layers? The answer is the algorithm called **error backpropagation** or **backward differentiation**.

### 6.6.1 Loss function

**cross-entropy loss**

The **cross-entropy loss** that is used in neural networks is the same one we saw for logistic regression. If the neural network is being used as a binary classifier, with the sigmoid at the final layer, the loss function is the same logistic regression loss we saw in Eq. 4.19:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{6.25}$$

If we are using the network to classify into 3 or more classes, the loss function is exactly the same as the loss for multinomial regression that we saw in Chapter 4 on

page 82. Let's briefly summarize the explanation here for convenience. First, when we have more than 2 classes we'll need to represent both $\mathbf{y}$ and $\hat{\mathbf{y}}$ as vectors. Let's assume we're doing **hard classification**, where only one class is the correct one. The true label $\mathbf{y}$ is then a vector with $K$ elements, each corresponding to a class, with $\mathbf{y}_c = 1$ if the correct class is $c$, with all other elements of $\mathbf{y}$ being 0. Recall that a vector like this, with one value equal to 1 and the rest 0, is called a **one-hot vector**. And our classifier will produce an estimate vector with $K$ elements $\hat{\mathbf{y}}$, each element $\hat{\mathbf{y}}_k$ of which represents the estimated probability $p(\mathbf{y}_k = 1|\mathbf{x})$.

The loss function for a single example $\mathbf{x}$ is the negative sum of the logs of the $K$ output classes, each weighted by their probability $y_k$:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{6.26}$$

We can simplify this equation further; let's first rewrite the equation using the function $\mathbb{1}\{\}$ which evaluates to 1 if the condition in the brackets is true and to 0 otherwise. This makes it more obvious that the terms in the sum in Eq. 6.26 will be 0 except for the term corresponding to the true class for which $\mathbf{y}_k = 1$:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\sum_{k=1}^{K} \mathbb{1}\{\mathbf{y}_k = 1\} \log \hat{\mathbf{y}}_k$$

In other words, the cross-entropy loss is simply the negative log of the output probability corresponding to the correct class, and we therefore also call this the **negative log likelihood loss**:

negative log likelihood loss

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\log \hat{\mathbf{y}}_c \quad \text{(where } c \text{ is the correct class)} \tag{6.27}$$

Plugging in the softmax formula from Eq. 6.9, and with $K$ the number of classes:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)} \tag{6.28}$$

Let's think about the negative log probability as a loss function. A perfect classifier would assign the correct class $i$ probability 1 and all the incorrect classes probability 0. That means the higher $p(\hat{y}_i)$ (the closer it is to 1), the better the classifier; $p(\hat{y}_i)$ is (the closer it is to 0), the worse the classifier. The negative log of this probability is a beautiful loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss). This loss function also insures that as probability of the correct answer is maximized, the probability of all the incorrect answers is minimized; since they all sum to one, any increase in the probability of the correct answer is coming at the expense of the incorrect answers.

The number $K$ of classes of the output vector $\hat{y}$ can be small or large. Perhaps our task is 3-way sentiment, and then the classes might be positive, negative, and neutral. Or if our task is deciding the part of speech of a word (i.e., whether it is a noun or verb or adjective, etc.), then $K$ is set of possible parts of speech in our tagset (of which there are 17 in the tagset we will define in Chapter 17). And if our task is language modeling, and our classifier is trying to predict which word is next, then our set of classes is the set of words, which might be 50,000 or 100,000.

### 6.6.2 Computing the Gradient

How do we compute the gradient of this loss function? Computing the gradient requires the partial derivative of the loss function with respect to each parameter. For a network with one weight layer and sigmoid output (which is what logistic regression is), we could simply use the derivative of the loss that we used for logistic regression in Eq. 6.29 (and derived in Section 4.15):

$$\begin{aligned} \frac{\partial L_{CE}(\hat{\mathbf{y}},\mathbf{y})}{\partial w_j} &= (\hat{y}-y)\,\mathbf{x}_j \\ &= (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)\,\mathbf{x}_j \end{aligned} \tag{6.29}$$

Or for a network with one weight layer and softmax output (=multinomial logistic regression), we could use the derivative of the softmax loss from Eq. 4.41, shown for a particular weight $\mathbf{w}_k$ and input $\mathbf{x}_i$

$$\begin{aligned} \frac{\partial L_{\mathrm{CE}}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k-\hat{\mathbf{y}}_k)\mathbf{x}_i \\ &= -(\mathbf{y}_k-p(\mathbf{y}_k=1|\mathbf{x}))\mathbf{x}_i \\ &= -\left(\mathbf{y}_k-\frac{\exp(\mathbf{w}_k\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w}_j\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i \end{aligned} \tag{6.30}$$

But these derivatives only give correct updates for one weight layer: the last one! For deep networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network.

error backpropagation

The solution to computing this gradient is an algorithm called **error backpropagation** or **backprop** (Rumelhart et al., 1986). While backprop was invented specially for neural networks, it turns out to be the same as a more general procedure called **backward differentiation**, which depends on the notion of **computation graphs**. Let's see how that works in the next subsection.

### 6.6.3 Computation Graphs

A computation graph is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.

Consider computing the function $L(a,b,c) = c(a+2b)$. If we make each of the component addition and multiplication operations explicit, and add names ($d$ and $e$) for the intermediate outputs, the resulting series of computations is:

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

We can now represent this as a graph, with nodes for each operation, and directed edges showing the outputs from each operation as the inputs to the next, as in Fig. 6.15. The simplest use of computation graphs is to compute the value of the function with some given inputs. In the figure, we've assumed the inputs $a=3$, $b=1$, $c=-2$, and we've shown the result of the **forward pass** to compute the result $L(3,1,-2) = -10$. In the forward pass of a computation graph, we apply each

operation left to right, passing the outputs of each computation as the input to the next node.

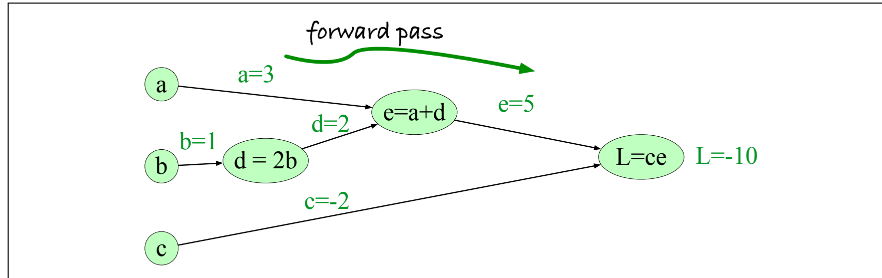


**Figure 6.15** Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of $L$.

### 6.6.4 Backward differentiation on computation graphs

The importance of the computation graph comes from the **backward pass**, which is used to compute the derivatives that we'll need for the weight update. In this example our goal is to compute the derivative of the output function $L$ with respect to each of the input variables, i.e., $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$. The derivative $\frac{\partial L}{\partial a}$ tells us how much a small change in $a$ affects $L$.

**chain rule**

Backwards differentiation makes use of the **chain rule** in calculus, so let's remind ourselves of that. Suppose we are computing the derivative of a composite function $f(x) = u(v(x))$. The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to $x$:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{6.31}$$

The chain rule extends to more than two functions. If computing the derivative of a composite function $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{6.32}$$

The intuition of backward differentiation is to pass gradients back from the final node to all the nodes in the graph. Fig. 6.16 shows part of the backward computation at one node $e$. Each node takes an upstream gradient that is passed in from its parent node to the right, and for each of its inputs computes a local gradient (the gradient of its output with respect to its input), and uses the chain rule to multiply these two to compute a downstream gradient to be passed on to the next earlier node.

Let's now compute the 3 derivatives we need. Since in the computation graph $L = ce$, we can directly compute the derivative $\frac{\partial L}{\partial c}$:

$$\frac{\partial L}{\partial c} = e \tag{6.33}$$

For the other two, we'll need to use the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{6.34}$$

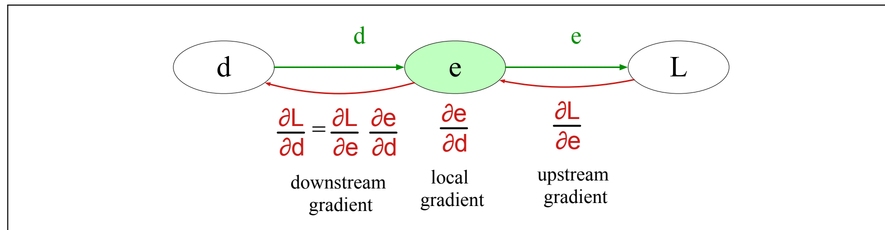


**Figure 6.16** Each node (like $e$ here) takes an upstream gradient, multiplies it by the local gradient (the gradient of its output with respect to its input), and uses the chain rule to compute a downstream gradient to be passed on to a prior node. A node may have multiple local gradients if it has multiple inputs.

Eq. 6.34 and Eq. 6.33 thus require five intermediate derivatives: $\frac{\partial L}{\partial e}$, $\frac{\partial L}{\partial c}$, $\frac{\partial e}{\partial a}$, $\frac{\partial e}{\partial d}$, and $\frac{\partial d}{\partial b}$, which are as follows (making use of the fact that the derivative of a sum is the sum of the derivatives):

$$
\begin{aligned}
L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\
e = a+d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\
d = 2b &: \quad \frac{\partial d}{\partial b} = 2
\end{aligned}
$$

In the backward pass, we compute each of these partials along each edge of the graph from right to left, using the chain rule just as we did above. Thus we begin by computing the downstream gradients from node $L$, which are $\frac{\partial L}{\partial e}$ and $\frac{\partial L}{\partial c}$. For node $e$, we then multiply this upstream gradient $\frac{\partial L}{\partial e}$ by the local gradient (the gradient of the output with respect to the input), $\frac{\partial e}{\partial d}$ to get the output we send back to node d: $\frac{\partial L}{\partial d}$. And so on, until we have annotated the graph all the way to all the input variables. The forward pass conveniently already will have computed the values of the forward intermediate variables we need (like $d$ and $e$) to compute these derivatives. Fig. 6.17 shows the backward pass.

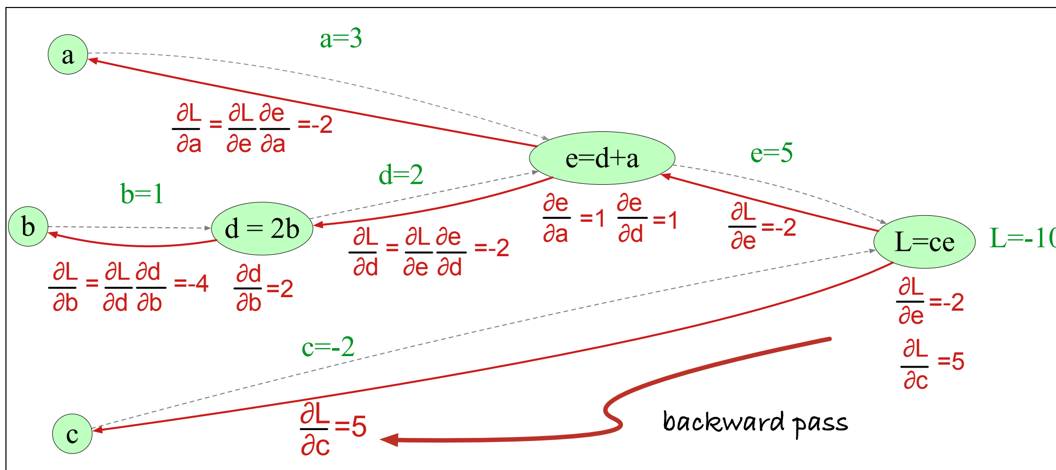


**Figure 6.17** Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

### Backward differentiation for a neural network

Of course computation graphs for real neural networks are much more complex. Fig. 6.18 shows a sample computation graph for a 2-layer neural network with $n_0 = 2$, $n_1 = 2$, and $n_2 = 1$, assuming binary classification and hence using a sigmoid output unit for simplicity. The function that the computation graph is computing is:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x}+\mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= \mathrm{ReLU}(\mathbf{z}^{[1]}) \\
z^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]}+b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned} \tag{6.35}$$

For the backward pass we'll also need to compute the loss $L$. The loss function for binary sigmoid output from Eq. 6.25 is

$$L_{CE}(\hat{y},y) = -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \tag{6.36}$$

Our output $\hat{y} = a^{[2]}$, so we can rephrase this as

$$L_{CE}(a^{[2]},y) = -\left[y\log a^{[2]}+(1-y)\log(1-a^{[2]})\right] \tag{6.37}$$

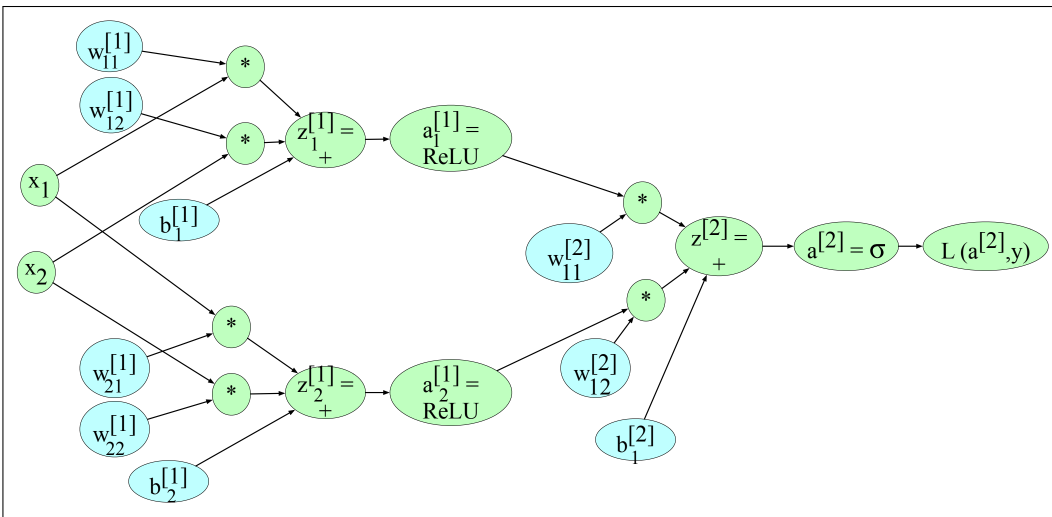


**Figure 6.18** Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input units and 2 hidden units. We've adjusted the notation a bit to avoid long equations in the nodes by just mentioning the function that is being computed, and the resulting variable name. Thus the * to the right of node $w_{11}^{[1]}$ means that $w_{11}^{[1]}$ is to be multiplied by $x_1$, and the node $z^{[1]} = +$ means that the value of $z^{[1]}$ is computed by summing the three nodes that feed into it (the two products, and the bias term $b_i^{[1]}$).

The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in teal. In order to do the backward pass, we'll need to know the derivatives of all the functions in the graph. We already saw in Section 4.15 the derivative of the sigmoid $\sigma$:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{6.38}$$

We'll also need the derivatives of each of the other activation functions. The derivative of tanh is:

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z) \tag{6.39}$$

The derivative of the ReLU is[2]

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; z < 0 \\ 1 & for \;\; z \geq 0 \end{cases} \tag{6.40}$$

We'll give the start of the computation, computing the derivative of the loss function $L$ with respect to $z$, or $\frac{\partial L}{\partial z}$ (and leaving the rest of the computation as an exercise for the reader). By the chain rule:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \tag{6.41}$$

So let's first compute $\frac{\partial L}{\partial a^{[2]}}$, taking the derivative of Eq. 6.37, repeated here:

$$\begin{aligned}
L_{CE}(a^{[2]}, y) &= -\left[y \log a^{[2]} + (1-y)\log(1-a^{[2]})\right] \\
\frac{\partial L}{\partial a^{[2]}} &= -\left(\left(y\frac{\partial \log(a^{[2]})}{\partial a^{[2]}}\right) + (1-y)\frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}}\right) \\
&= -\left(\left(y\frac{1}{a^{[2]}}\right) + (1-y)\frac{1}{1-a^{[2]}}(-1)\right) \\
&= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right)
\end{aligned} \tag{6.42}$$

Next, by the derivative of the sigmoid:

$$\frac{\partial a^{[2]}}{\partial z} = a^{[2]}(1-a^{[2]})$$

Finally, we can use the chain rule:

$$\begin{aligned}
\frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \\
&= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) a^{[2]}(1-a^{[2]}) \\
&= a^{[2]} - y
\end{aligned} \tag{6.43}$$

Continuing the backward computation of the gradients (next by passing the gradients over $b_1^{[2]}$ and the two product nodes, and so on, back to all the teal nodes), is left as an exercise for the reader.

### 6.6.5 More details on learning

Optimization in neural networks is a non-convex optimization problem, more complex than for logistic regression, and for that and other reasons there are many best practices for successful learning.

[2] The derivative is actually undefined at the point $z = 0$, but by convention we treat it as 1.

For logistic regression we can initialize gradient descent with all the weights and biases having the value 0. In neural networks, by contrast, we need to initialize the weights with small random numbers. It's also helpful to normalize the input values to have 0 mean and unit variance.

**dropout**

Various forms of regularization are used to prevent overfitting. One of the most important is **dropout**: randomly dropping some units and their connections from the network during training (Hinton et al. 2012, Srivastava et al. 2014). At each iteration of training (whenever we update parameters, i.e. each mini-batch if we are using mini-batch gradient descent), we repeatedly choose a probability $p$ and for each unit we replace its output with zero with probability $p$ (and renormalize the rest of the outputs from that layer).

**hyperparameter**

Tuning of **hyperparameters** is also important. The parameters of a neural network are the weights $\mathbf{W}$ and biases $\mathbf{b}$; those are learned by gradient descent. The hyperparameters are things that are chosen by the algorithm designer; optimal values are tuned on a devset rather than by gradient descent learning on the training set. Hyperparameters include the learning rate $\eta$, the mini-batch size, the model architecture (the number of layers, the number of hidden nodes per layer, the choice of activation functions), how to regularize, and so on. Gradient descent itself also has many architectural variants such as Adam (Kingma and Ba, 2015).

Finally, most modern neural networks are built using computation graph formalisms that make it easy and natural to do gradient computation and parallelization on vector-based GPUs (Graphic Processing Units). PyTorch (Paszke et al., 2017) and TensorFlow (Abadi et al., 2015) are two of the most popular. The interested reader should consult a neural network textbook for further details; some suggestions are at the end of the chapter.

## 6.7 Summary

- Neural networks are built out of **neural units**. Originally inspired by biological neurons, neural networks are now an abstract computational device rather than a biological model.
- Each neural unit multiplies input values by a weight vector, adds a bias, and then applies a non-linear activation function like sigmoid, tanh, or rectified linear unit.
- In a **fully-connected**, **feedforward** network, each unit in layer $i$ is connected to each unit in layer $i+1$, and there are no cycles.
- The power of neural networks comes from the ability of early layers to learn representations that can be utilized by later layers in the network.
- Neural networks are trained by optimization algorithms like **gradient descent**.
- **Error backpropagation**, backward differentiation on a **computation graph**, is used to compute the gradients of the loss function for a network.
- **Neural language models** use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous $n$ words.
- Neural language models can use pretrained **embeddings**, or can learn embeddings from scratch in the process of language modeling.

## Historical Notes

The origins of neural networks lie in the 1940s **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the biological neuron as a kind of computing element that could be described in terms of propositional logic. By the late 1950s and early 1960s, a number of labs (including Frank Rosenblatt at Cornell and Bernard Widrow at Stanford) developed research into neural networks; this phase saw the development of the perceptron (Rosenblatt, 1958), and the transformation of the threshold into a bias, a notation we still use (Widrow and Hoff, 1960).

The field of neural networks declined after it was shown that a single perceptron unit was unable to model functions as simple as XOR (Minsky and Papert, 1969). While some small amount of work continued during the next two decades, a major revival for the field didn't come until the 1980s, when practical tools for building deeper networks like error backpropagation became widespread (Rumelhart et al., 1986). During the 1980s a wide variety of neural network and related architectures were developed, particularly for applications in psychology and cognitive science (Rumelhart and McClelland 1986b, McClelland and Elman 1986, Rumelhart and McClelland 1986a, Elman 1990), for which the term **connectionist** or **parallel distributed processing** was often used (Feldman and Ballard 1982, Smolensky 1988). Many of the principles and techniques developed in this period are foundational to modern work, including the ideas of distributed representations (Hinton, 1986), recurrent networks (Elman, 1990), and the use of tensors for compositionality (Smolensky, 1990).

connectionist

By the 1990s larger neural networks began to be applied to many practical language processing tasks as well, like handwriting recognition (LeCun et al. 1989) and speech recognition (Morgan and Bourlard 1990). By the early 2000s, improvements in computer hardware and advances in optimization and training techniques made it possible to train even larger and deeper networks, leading to the modern term **deep learning** (Hinton et al. 2006, Bengio et al. 2007). We cover more related history in Chapter 13 and Chapter 15.

There are a number of excellent books on neural networks, including Goodfellow et al. (2016) and Nielsen (2015).

CHAPTER

# 13 RNNs and LSTMs

*Time will explain.*
Jane Austen, *Persuasion*

Language is an inherently temporal phenomenon. Spoken language is a sequence of acoustic events over time, and we comprehend and produce both spoken and written language as a sequential input stream. The temporal nature of language is reflected in the metaphors we use; we talk of the *flow of conversations*, *news feeds*, and *twitter streams*, all of which emphasize that language is a sequence that unfolds in time.

This chapter introduces a deep learning architecture, the **recurrent neural network (RNN)**, and RNN variants like LSTMs, that offer a different way of representing time than feedforward and transformer networks. RNNs have a mechanism that deals directly with the sequential nature of language, allowing them to handle the temporal nature of language without the use of arbitrary fixed-sized windows. The recurrent network offers a new way to represent the prior context, in its **recurrent connections**, allowing the model's decision to depend on information from hundreds of words in the past. We'll see how to apply the model to the task of language modeling, to text classification tasks like sentiment analysis, and to sequence modeling tasks like part-of-speech tagging.

## 13.1 Recurrent Neural Networks

A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input. While powerful, such networks are difficult to reason about and to train. However, within the general class of recurrent networks there are constrained architectures that have proven to be extremely effective when applied to language. In this section, we consider a class of recurrent networks referred to as **Elman Networks** (Elman, 1990) or **simple recurrent networks**. These networks are useful in their own right and serve as the basis for more complex approaches like the Long Short-Term Memory (LSTM) networks discussed later in this chapter. In this chapter when we use the term RNN we'll be referring to these simpler more constrained networks (although you will often see the term RNN to mean any net with recurrent properties including LSTMs).

**Elman Networks**

Fig. 13.1 illustrates the structure of an RNN. As with ordinary feedforward networks, an input vector representing the current input, $\mathbf{x}_t$, is multiplied by a weight matrix and then passed through a non-linear activation function to compute the values for a layer of hidden units. This hidden layer is then used to calculate a corresponding output, $\mathbf{y}_t$. In a departure from our earlier window-based approach, sequences are processed by presenting one item at a time to the network. We'll use

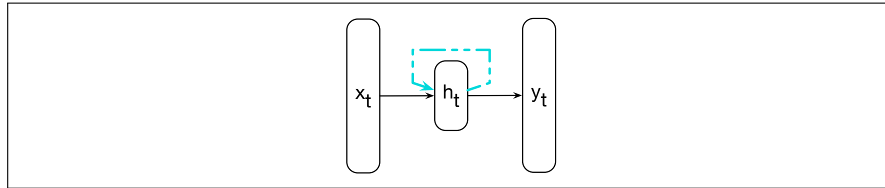


**Figure 13.1** Simple recurrent neural network after Elman (1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

subscripts to represent time, thus $\mathbf{x}_t$ will mean the input vector $\mathbf{x}$ at time $t$. The key difference from a feedforward network lies in the recurrent link shown in the figure with the dashed line. This link augments the input to the computation at the hidden layer with the value of the hidden layer *from the preceding point in time*.

The hidden layer from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time. Critically, this approach does not impose a fixed-length limit on this prior context; the context embodied in the previous hidden layer can include information extending back to the beginning of the sequence.

Adding this temporal dimension makes RNNs appear to be more complex than non-recurrent architectures. But in reality, they're not all that different. Given an input vector and the values for the hidden layer from the previous time step, we're still performing the standard feedforward calculation introduced in Chapter 6. To see this, consider Fig. 13.2 which clarifies the nature of the recurrence and how it factors into the computation at the hidden layer. The most significant change lies in the new set of weights, $\mathbf{U}$, that connect the hidden layer from the previous time step to the current hidden layer. These weights determine how the network makes use of past context in calculating the output for the current input. As with the other weights in the network, these connections are trained via backpropagation.

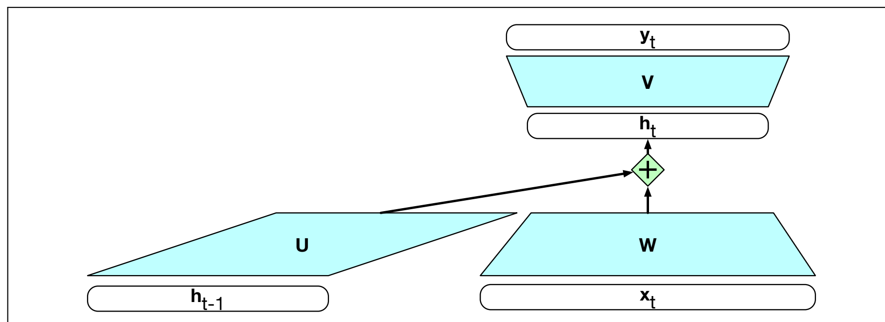


**Figure 13.2** Simple recurrent neural network illustrated as a feedforward network. The hidden layer $\mathbf{h}_{t-1}$ from the prior time step is multiplied by weight matrix $\mathbf{U}$ and then added to the feedforward component from the current time step.

### 13.1.1 Inference in RNNs

Forward inference (mapping a sequence of inputs to a sequence of outputs) in an RNN is nearly identical to what we've already seen with feedforward networks. To compute an output $\mathbf{y}_t$ for an input $\mathbf{x}_t$, we need the activation value for the hidden layer $\mathbf{h}_t$. To calculate this, we multiply the input $\mathbf{x}_t$ with the weight matrix $\mathbf{W}$, and

the hidden layer from the previous time step $\mathbf{h}_{t-1}$ with the weight matrix $\mathbf{U}$. We add these values together and pass them through a suitable activation function, $g$, to arrive at the activation value for the current hidden layer, $\mathbf{h}_t$. Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \quad (13.1)$$

$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t) \quad (13.2)$$

Let's refer to the input, hidden and output layer dimensions as $d_{in}$, $d_h$, and $d_{out}$ respectively. Given this, our three parameter matrices are: $\mathbf{W} \in \mathbb{R}^{d_h \times d_{in}}$, $\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$, and $\mathbf{V} \in \mathbb{R}^{d_{out} \times d_h}$.

We compute $y_t$ via a softmax computation that gives a probability distribution over the possible output classes.

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \quad (13.3)$$

The fact that the computation at time $t$ requires the value of the hidden layer from time $t-1$ mandates an incremental inference algorithm that proceeds from the start of the sequence to the end as illustrated in Fig. 13.3. The sequential nature of simple recurrent networks can also be seen by *unrolling* the network in time as is shown in Fig. 13.4. In this figure, the various layers of units are copied for each time step to illustrate that they will have differing values over time. However, the various weight matrices are shared across time.

```
function FORWARDRNN(x, network) returns output sequence y

  h_0 ← 0
  for i ← 1 to LENGTH(x) do
      h_i ← g(U h_{i-1} + W x_i)
      y_i ← f(V h_i)
  return y
```

**Figure 13.3** Forward inference in a simple recurrent network. The matrices **U**, **V** and **W** are shared across time, while new values for **h** and **y** are calculated with each time step.

### 13.1.2 Training

As with feedforward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks. As shown in Fig. 13.2, we now have 3 sets of weights to update: $\mathbf{W}$, the weights from the input layer to the hidden layer, $\mathbf{U}$, the weights from the previous hidden layer to the current hidden layer, and finally $\mathbf{V}$, the weights from the hidden layer to the output layer.

Fig. 13.4 highlights two considerations that we didn't have to worry about with backpropagation in feedforward networks. First, to compute the loss function for the output at time $t$ we need the hidden layer from time $t-1$. Second, the hidden layer at time $t$ influences both the output at time $t$ and the hidden layer at time $t+1$ (and hence the output and loss at $t+1$). It follows from this that to assess the error accruing to $\mathbf{h}_t$, we'll need to know its influence on both the current output *as well as the ones that follow*.

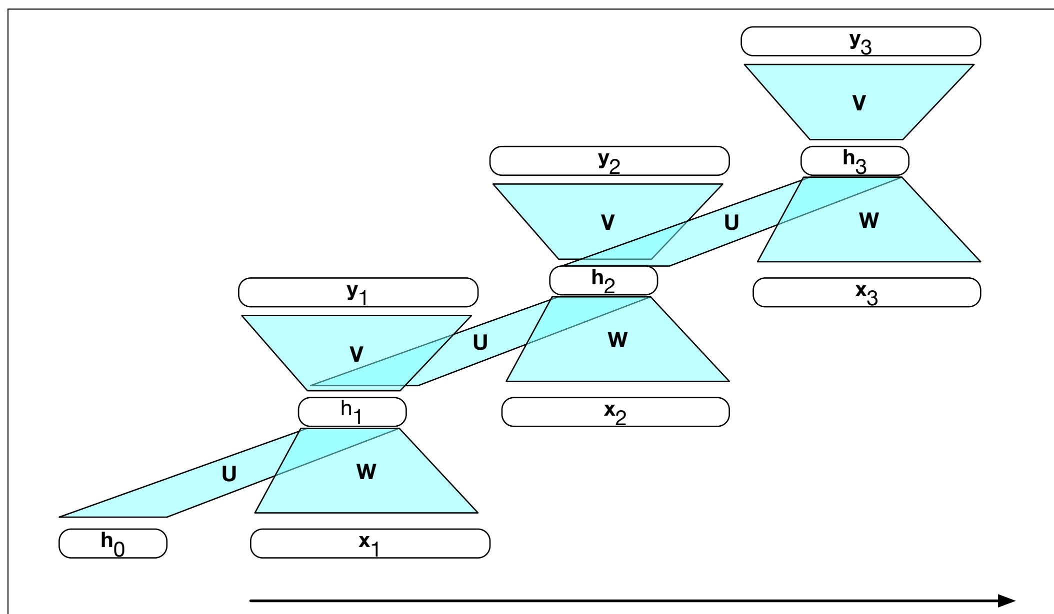


**Figure 13.4** A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared across all time steps.

Tailoring the backpropagation algorithm to this situation leads to a two-pass algorithm for training the weights in RNNs. In the first pass, we perform forward inference, computing $\mathbf{h}_t$, $\mathbf{y}_t$, accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step. In the second pass, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time. This general approach is commonly referred to as **backpropagation through time** (Werbos 1974, Rumelhart et al. 1986, Werbos 1990).

**backpropagation through time**

Fortunately, with modern computational frameworks and adequate computing resources, there is no need for a specialized approach to training RNNs. As illustrated in Fig. 13.4, explicitly unrolling a recurrent network into a feedforward computational graph eliminates any explicit recurrences, allowing the network weights to be trained directly. In such an approach, we provide a template that specifies the basic structure of the network, including all the necessary parameters for the input, output, and hidden layers, the weight matrices, as well as the activation and output functions to be used. Then, when presented with a specific input sequence, we can generate an unrolled feedforward network specific to that input, and use that graph to perform forward inference or training via ordinary backpropagation.

For applications that involve much longer input sequences, such as speech recognition, character-level processing, or streaming continuous inputs, unrolling an entire input sequence may not be feasible. In these cases, we can unroll the input into manageable fixed-length segments and treat each segment as a distinct training item.

## 13.2 RNNs as Language Models

Let's see how to apply RNNs to the language modeling task. Recall from Chapter 3 that language models predict the next word in a sequence given some preceding context. For example, if the preceding context is *"Thanks for all the"* and we want to know how likely the next word is *"fish"* we would compute:

$$P(\textit{fish}|\textit{Thanks for all the})$$

Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. We can also assign probabilities to entire sequences by combining these conditional probabilities with the chain rule:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{<i})$$

The n-gram language models of Chapter 3 compute the probability of a word given counts of its occurrence with the $n-1$ prior words. The context is thus of size $n-1$. For the feedforward language models of Chapter 6, the context is the window size.

RNN language models (Mikolov et al., 2010) process the input sequence one word at a time, attempting to predict the next word from the current word and the previous hidden state. RNNs thus don't have the limited context problem that n-gram models have, or the fixed context that feedforward language models have, since the hidden state can in principle represent information about all of the preceding words all the way back to the beginning of the sequence. Fig. 13.5 sketches this difference between a FFN language model and an RNN language model, showing that the RNN language model uses $h_{t-1}$, the hidden state from the previous time step, as a representation of the past context.

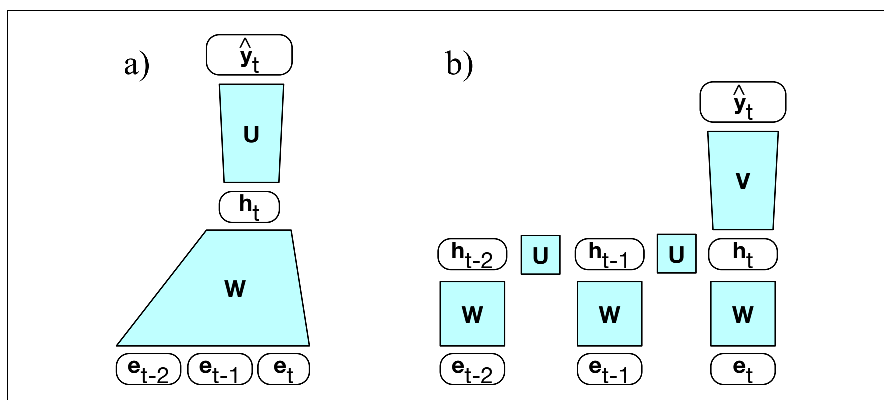


**Figure 13.5** Simplified sketch of two LM architectures moving through a text, showing a schematic context of three tokens: (a) a feedforward neural language model which has a fixed context input to the weight matrix $\mathbf{W}$, (b) an RNN language model, in which the hidden state $\mathbf{h}_{t-1}$ summarizes the prior context.

### 13.2.1 Forward Inference in an RNN language model

Forward inference in a recurrent language model proceeds exactly as described in Section 13.1.1. The input sequence $\mathbf{X} = [\mathbf{x}_1; ...; \mathbf{x}_t; ...; \mathbf{x}_N]$ consists of a series of

words each represented as a one-hot vector of size $|V| \times 1$, and the output prediction, $\hat{\mathbf{y}}$, is a vector representing a probability distribution over the vocabulary. At each step, the model uses the word embedding matrix $\mathbf{E}$ to retrieve the embedding for the current word, multiples it by the weight matrix $\mathbf{W}$, and then adds it to the hidden layer from the previous step (weighted by weight matrix $\mathbf{U}$) to compute a new hidden layer. This hidden layer is then used to generate an output layer which is passed through a softmax layer to generate a probability distribution over the entire vocabulary. That is, at time $t$:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \quad (13.4)$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \quad (13.5)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \quad (13.6)$$

When we do language modeling with RNNs (and we'll see this again in Chapter 8 with transformers), it's convenient to make the assumption that the embedding dimension $d_e$ and the hidden dimension $d_h$ are the same. So we'll just call both of these the **model dimension** $d$. So the embedding matrix $\mathbf{E}$ is of shape $[d \times |V|]$, and $\mathbf{x_t}$ is a one-hot vector of shape $[|V| \times 1]$. The product $\mathbf{e}_t$ is thus of shape $[d \times 1]$. $\mathbf{W}$ and $\mathbf{U}$ are of shape $[d \times d]$, so $\mathbf{h}_t$ is also of shape $[d \times 1]$. $\mathbf{V}$ is of shape $[|V| \times d]$, so the result of $\mathbf{Vh}$ is a vector of shape $[|V| \times 1]$. This vector can be thought of as a set of scores over the vocabulary given the evidence provided in $\mathbf{h}$. Passing these scores through the softmax normalizes the scores into a probability distribution. The probability that a particular word $k$ in the vocabulary is the next word is represented by $\hat{\mathbf{y}}_t[k]$, the $k$th component of $\hat{\mathbf{y}}_t$:

$$P(w_{t+1} = k | w_1, \ldots, w_t) = \hat{\mathbf{y}}_t[k] \quad (13.7)$$

The probability of an entire sequence is just the product of the probabilities of each item in the sequence, where we'll use $\hat{\mathbf{y}}_i[w_i]$ to mean the probability of the true word $w_i$ at time step $i$.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \quad (13.8)$$

$$= \prod_{i=1}^{n} \hat{\mathbf{y}}_i[w_i] \quad (13.9)$$

### 13.2.2 Training an RNN language model

self-supervision

To train an RNN as a language model, we use the same **self-supervision** (or **self-training**) algorithm we saw in Section 7.5: we take a corpus of text as training material and at each time step $t$ ask the model to predict the next word. We call such a model self-supervised because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision! We simply train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function. Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \quad (13.10)$$

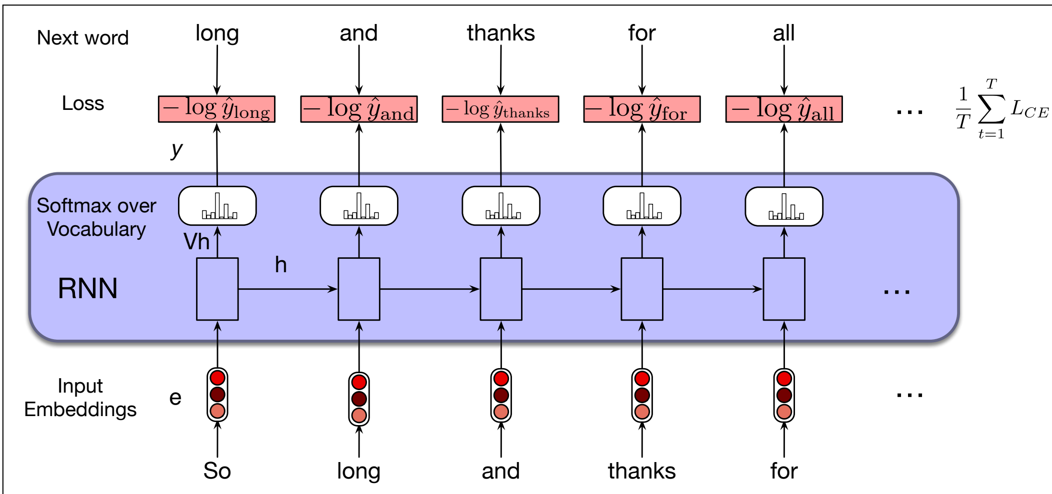


**Figure 13.6** Training RNNs as language models.

In the case of language modeling, the correct distribution $\mathbf{y}_t$ comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time $t$ the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{13.11}$$

Thus at each word position $t$ of the input, the model takes as input the correct word $w_t$ together with $h_{t-1}$, encoding information from the preceding $w_{1:t-1}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token $w_{t+1}$. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct word $w_{t+1}$ along with the prior history encoded to estimate the probability of token $w_{t+2}$. This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.

teacher forcing

The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent. Fig. 13.6 illustrates this training regimen.

### 13.2.3 Weight Tying

Careful readers may have noticed that the input embedding matrix $\mathbf{E}$ and the final layer matrix $\mathbf{V}$, which feeds the output softmax, are quite similar.

The columns of $\mathbf{E}$ represent the word embeddings for each word in the vocabulary learned during the training process with the goal that words that have similar meaning and function will have similar embeddings. And, since when we use RNNs for language modeling we make the assumption that the embedding dimension and the hidden dimension are the same (= the model dimension $d$), the embedding matrix $\mathbf{E}$ has shape $[d \times |V|]$. And the final layer matrix $\mathbf{V}$ provides a way to score the likelihood of each word in the vocabulary given the evidence present in the final hidden layer of the network through the calculation of $\mathbf{Vh}$. $\mathbf{V}$ is of shape $[|V| \times d]$. That is, the rows of $\mathbf{V}$ are shaped like a transpose of $\mathbf{E}$, meaning that $\mathbf{V}$ provides a

*second set* of learned word embeddings.

Instead of having two sets of embedding matrices, language models use a single embedding matrix, which appears at both the input and softmax layers. That is, we dispense with $\mathbf{V}$ and use $\mathbf{E}$ at the start of the computation and $\mathbf{E}^\intercal$ (because the shape of $\mathbf{V}$ is the transpose of $\mathbf{E}$ at the end. Using the same matrix (transposed) in two places is called **weight tying**.[1] The weight-tied equations for an RNN language model then become:

weight tying

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{13.12}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{13.13}$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{E}^\intercal\mathbf{h}_t) \tag{13.14}$$

In addition to providing improved model perplexity, this approach significantly reduces the number of parameters required for the model.

# 13.3 RNNs for other NLP tasks

Now that we've seen the basic RNN architecture, let's consider how to apply it to three types of NLP tasks: *sequence classification* tasks like sentiment analysis and topic classification, *sequence labeling* tasks like part-of-speech tagging, and *text generation* tasks, including with a new architecture called the **encoder-decoder**.

## 13.3.1 Sequence Labeling

In sequence labeling, the network's task is to assign a label chosen from a small fixed set of labels to each element of a sequence. One classic sequence labeling tasks is part-of-speech (POS) tagging (assigning grammatical tags like NOUN and VERB to each word in a sentence). We'll discuss part-of-speech tagging in detail in Chapter 17, but let's give a motivating example here. In an RNN approach to sequence labeling, inputs are word embeddings and the outputs are tag probabilities generated by a softmax layer over the given tagset, as illustrated in Fig. 13.7.

In this figure, the inputs at each time step are pretrained word embeddings corresponding to the input tokens. The RNN block is an abstraction that represents an unrolled simple recurrent network consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ weight matrices that comprise the network. The outputs of the network at each time step represent the distribution over the POS tagset generated by a softmax layer.

To generate a sequence of tags for a given input, we run forward inference over the input sequence and select the most likely tag from the softmax at each step. Since we're using a softmax layer to generate the probability distribution over the output tagset at each time step, we will again employ the cross-entropy loss during training.

## 13.3.2 RNNs for Sequence Classification

Another use of RNNs is to classify entire sequences rather than the tokens within them. This is the set of tasks commonly called **text classification**, like sentiment analysis or spam detection, in which we classify a text into two or three classes (like positive or negative), as well as classification tasks with a large number of

[1] We also do this for transformers (Chapter 8) where it's common to call $\mathbf{E}^\intercal$ the **unembedding matrix**.

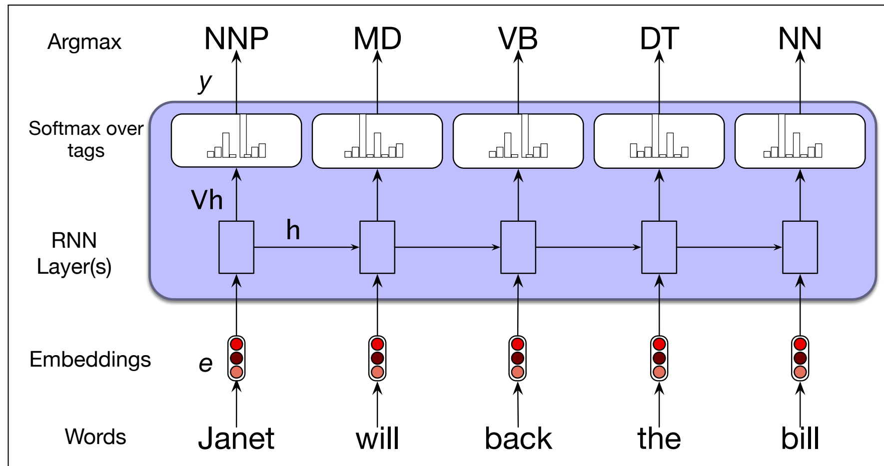


**Figure 13.7** Part-of-speech tagging as sequence labeling with a simple RNN. The goal of part-of-speech (POS) tagging is to assign a grammatical label to each word in a sentence, drawn from a predefined set of tags. (The tags for this sentence include NNP (proper noun), MD (modal verb) and others; we'll give a complete description of the task of part-of-speech tagging in Chapter 17.) Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

categories, like document-level topic classification, or message routing for customer service applications.

To apply RNNs in this setting, we pass the text to be classified through the RNN a word at a time generating a new hidden layer representation at each time step. We can then take the hidden layer for the last token of the text, $\mathbf{h}_n$, to constitute a compressed representation of the entire sequence. We can pass this representation $\mathbf{h}_n$ to a feedforward network that chooses a class via a softmax over the possible classes. Fig. 13.8 illustrates this approach.

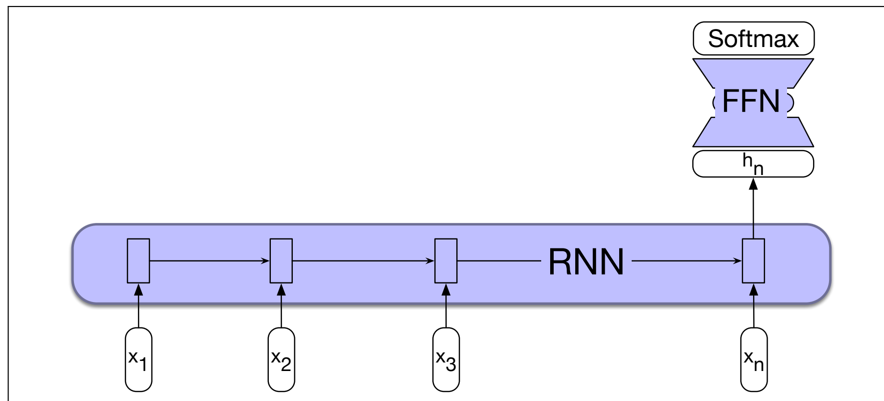


**Figure 13.8** Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

Note that in this approach we don't need intermediate outputs for the words in the sequence preceding the last element. Therefore, there are no loss terms associated with those elements. Instead, the loss function used to train the weights in the network is based entirely on the final text classification task. The output from the softmax output from the feedforward classifier together with a cross-entropy loss

drives the training. The error signal from the classification is backpropagated all the way through the weights in the feedforward classifier through, to its input, and then through to the three sets of weights in the RNN as described earlier in Section 13.1.2. The training regimen that uses the loss from a downstream application to adjust the weights all the way through the network is referred to as **end-to-end training**.

end-to-end training

Another option, instead of using just the hidden state of the last token $h_n$ to represent the whole sequence, is to use some sort of **pooling** function of all the hidden states $h_i$ for each word $i$ in the sequence. For example, we can create a representation that pools all the $n$ hidden states by taking their element-wise mean:

pooling

$$\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i \tag{13.15}$$

Or we can take the element-wise max; the element-wise max of a set of $n$ vectors is a new vector whose $k$th element is the max of the $k$th elements of all the $n$ vectors.

The long contexts of RNNs makes it quite difficult to successfully backpropagate error all the way through the entire input; we'll talk about this problem, and some standard solutions, in Section 13.5.

### 13.3.3 Generation with RNN-Based Language Models

RNN-based language models can also be used to generate text, Text generation, along with image generation and code generation, constitute a new area of AI that is often called **generative AI**. Those of you who have already read Chapter 7 and Chapter 8 will have already seen this, but we reintroduce it here for those who are reading in a different order.

Recall back in Chapter 3 we saw how to generate text from an n-gram language model by adapting a **sampling** technique suggested at about the same time by Claude Shannon (Shannon, 1951) and the psychologists George Miller and Jennifer Selfridge (Miller and Selfridge, 1950). We first randomly sample a word to begin a sequence based on its suitability as the start of a sequence. We then continue to sample words *conditioned on our previous choices* until we reach a pre-determined length, or an end of sequence token is generated.

Today, this approach of using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**. The procedure is basically the same as that described on page 49, but adapted to a neural context:

autoregressive generation

- Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, <s>, as the first input.
- Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
- Continue generating until the end of sentence marker, </s>, is sampled or a fixed length limit is reached.

Technically an **autoregressive** model is a model that predicts a value at time $t$ based on a linear function of the previous values at times $t-1$, $t-2$, and so on. Although language models are not linear (since they have many layers of non-linearities), we loosely refer to this generation technique as **autoregressive generation** since the word generated at each time step is conditioned on the word selected by the network from the previous step. Fig. 13.9 illustrates this approach. In this figure, the details of the RNN's hidden layers and recurrent connections are hidden within the blue block.

This simple architecture underlies state-of-the-art approaches to applications such as machine translation, summarization, and question answering. The key to these approaches is to prime the generation component with an appropriate context. That is, instead of simply using <s> to get things started we can provide a richer task-appropriate context; for translation the context is the sentence in the source language; for summarization it's the long text we want to summarize.

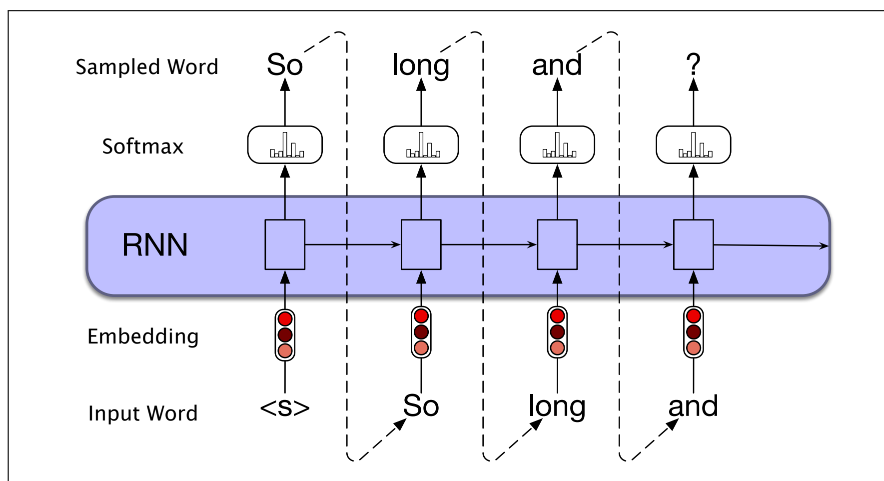


**Figure 13.9** Autoregressive generation with an RNN-based neural language model.

# 13.4 Stacked and Bidirectional RNN architectures

Recurrent networks are quite flexible. By combining the feedforward nature of unrolled computational graphs with vectors as common inputs and outputs, complex networks can be treated as modules that can be combined in creative ways. This section introduces two of the more common network architectures used in language processing with RNNs.

## 13.4.1 Stacked RNNs

In our examples thus far, the inputs to our RNNs have consisted of sequences of word or character embeddings (vectors) and the outputs have been vectors useful for predicting words, tags or sequence labels. However, nothing prevents us from using the entire sequence of outputs from one RNN as an input sequence to another one. **Stacked RNNs** consist of multiple networks where the output of one layer serves as the input to a subsequent layer, as shown in Fig. 13.10.

Stacked RNNs

Stacked RNNs generally outperform single-layer networks. One reason for this success seems to be that the network induces representations at differing levels of abstraction across layers. Just as the early stages of the human visual system detect edges that are then used for finding larger regions and shapes, the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers—representations that might prove difficult to induce in a single RNN. The optimal number of stacked RNNs is specific to each application and to each training set. However, as the number of stacks is increased the training costs rise

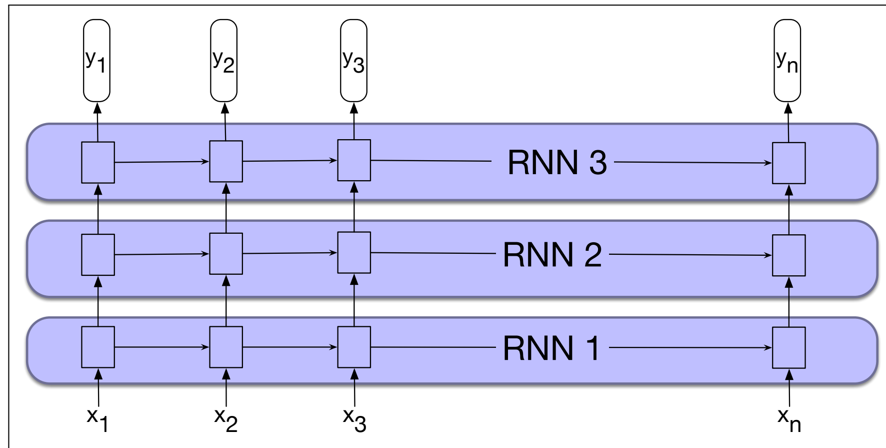


**Figure 13.10** Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

quickly.

## 13.4.2 Bidirectional RNNs

The RNN uses information from the left (prior) context to make its predictions at time $t$. But in many applications we have access to the entire input sequence; in those cases we would like to use words from the context to the right of $t$. One way to do this is to run two separate RNNs, one left-to-right, and one right-to-left, and concatenate their representations.

In the left-to-right RNNs we've discussed so far, the hidden state at a given time $t$ represents everything the network knows about the sequence up to that point. The state is a function of the inputs $x_1, ..., x_t$ and represents the context of the network to the left of the current time.

$$\mathbf{h}^f_t = \text{RNN}_{\text{forward}}(\mathbf{x}_1, \ldots, \mathbf{x}_t) \tag{13.16}$$

This new notation $\mathbf{h}^f_t$ simply corresponds to the normal hidden state at time $t$, representing everything the network has gleaned from the sequence so far.

To take advantage of context to the right of the current input, we can train an RNN on a *reversed* input sequence. With this approach, the hidden state at time $t$ represents information about the sequence to the *right* of the current input:

$$\mathbf{h}^b_t = \text{RNN}_{\text{backward}}(\mathbf{x}_t, \ldots \mathbf{x}_n) \tag{13.17}$$

Here, the hidden state $\mathbf{h}^b_t$ represents all the information we have discerned about the sequence from $t$ to the end of the sequence.

**bidirectional RNN**

A **bidirectional RNN** (Schuster and Paliwal, 1997) combines two independent RNNs, one where the input is processed from the start to the end, and the other from the end to the start. We then concatenate the two representations computed by the networks into a single vector that captures both the left and right contexts of an input at each point in time. Here we use either the semicolon ";" or the equivalent symbol $\oplus$ to mean vector concatenation:

$$\begin{aligned}\mathbf{h}_t &= [\mathbf{h}^f_t\,;\,\mathbf{h}^b_t] \\ &= \mathbf{h}^f_t \oplus \mathbf{h}^b_t \end{aligned} \tag{13.18}$$

Fig. 13.11 illustrates such a bidirectional network that concatenates the outputs of the forward and backward pass. Other simple ways to combine the forward and backward contexts include element-wise addition or multiplication. The output at each step in time thus captures information to the left and to the right of the current input. In sequence labeling applications, these concatenated outputs can serve as the basis for a local labeling decision.

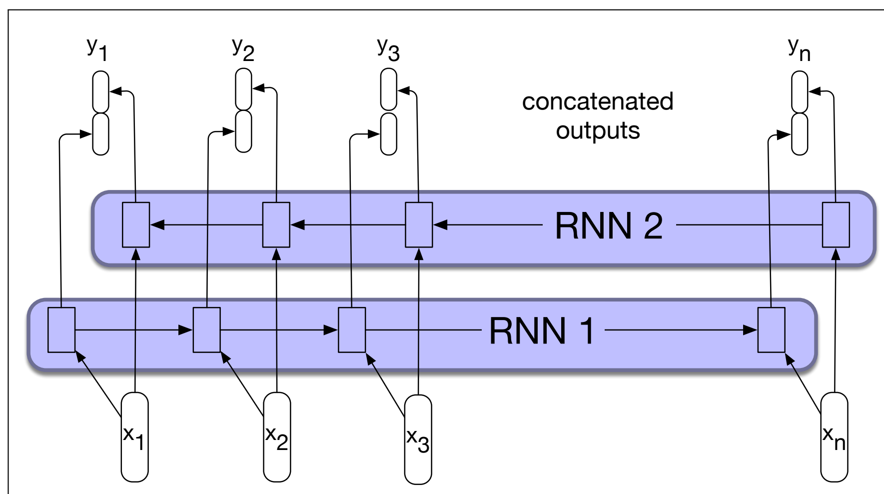


**Figure 13.11** A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Bidirectional RNNs have also proven to be quite effective for sequence classification. Recall from Fig. 13.8 that for sequence classification we used the final hidden state of the RNN as the input to a subsequent feedforward classifier. A difficulty with this approach is that the final state naturally reflects more information about the end of the sentence than its beginning. Bidirectional RNNs provide a simple solution to this problem; as shown in Fig. 13.12, we simply combine the final hidden states from the forward and backward passes (for example by concatenation) and use that as input for follow-on processing.

## 13.5 The LSTM

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

(13.19) The flights the airline was canceling were full.

Assigning a high probability to *was* following *airline* is straightforward since *airline* provides a strong local context for the singular agreement. However, assigning an appropriate probability to *were* is quite difficult, not only because the plural *flights* is quite distant, but also because the singular noun *airline* is closer in the intervening

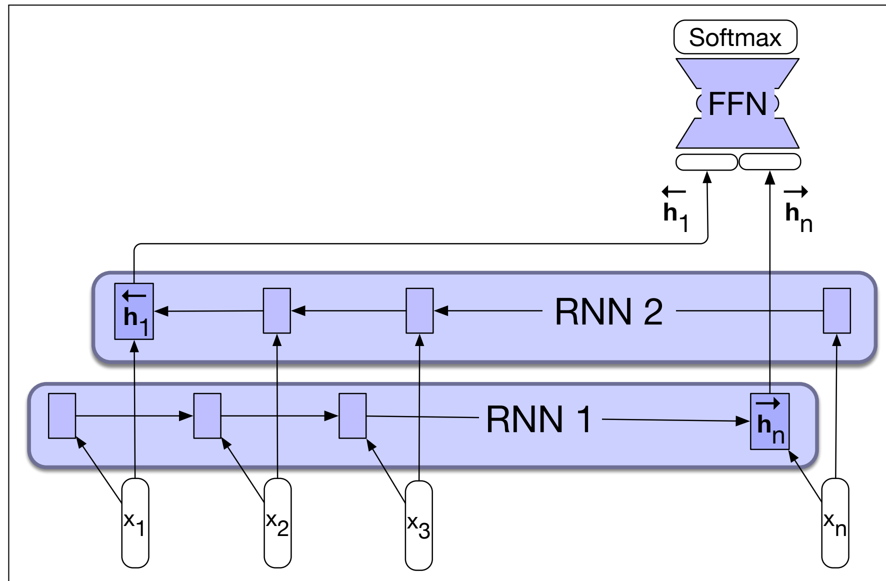


**Figure 13.12** A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

context. Ideally, a network should be able to retain the distant information about plural *flights* until it is needed, while still processing the intermediate parts of the sequence correctly.

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

A second difficulty with training RNNs arises from the need to backpropagate the error signal back through time. Recall from Section 13.1.2 that the hidden layer at time $t$ contributes to the loss at the next time step since it takes part in that calculation. As a result, during the backward pass of training, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence. A frequent result of this process is that the gradients are eventually driven to zero, a situation called the **vanishing gradients** problem.

vanishing gradients

To address these issues, more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time, by enabling the network to learn to forget information that is no longer needed and to remember information required for decisions still to come.

The most commonly used such extension to RNNs is the **long short-term memory** (LSTM) network (Hochreiter and Schmidhuber, 1997). LSTMs divide the context management problem into two subproblems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of *gates* to control the flow of information into and out of the units that

long short-term memory

comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

The gates in an LSTM share a common design pattern; each consists of a feed-forward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated. The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1. Combining this with a pointwise multiplication has an effect similar to that of a binary mask. Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

**forget gate**

The first gate we'll consider is the **forget gate**. The purpose of this gate is to delete information from the context that is no longer needed. The forget gate computes a weighted sum of the previous state's hidden layer and the current input and passes that through a sigmoid. This mask is then multiplied element-wise by the context vector to remove the information from context that is no longer required. Element-wise multiplication of two vectors (represented by the operator $\odot$, and sometimes called the **Hadamard product**) is the vector of the same dimension as the two input vectors, where each element $i$ is the product of element $i$ in the two input vectors:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t) \quad (13.20)$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \quad (13.21)$$

The next task is to compute the actual information we need to extract from the previous hidden state and current inputs—the same basic computation we've been using for all our recurrent networks.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t) \quad (13.22)$$

**add gate**

Next, we generate the mask for the **add gate** to select the information to add to the current context.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t) \quad (13.23)$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \quad (13.24)$$

Next, we add this to the modified context vector to get our new context vector.

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \quad (13.25)$$

**output gate**

The final gate we'll use is the **output gate** which is used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t) \quad (13.26)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (13.27)$$

Fig. 13.13 illustrates the complete computation for a single LSTM unit. Given the appropriate weights for the various gates, an LSTM accepts as input the context layer, and hidden layer from the previous time step, along with the current input vector. It then generates updated context and hidden vectors as output.

It is the hidden state, $h_t$, that provides the output for the LSTM at each time step. This output can be used as the input to subsequent layers in a stacked RNN, or at the final layer of a network $h_t$ can be used to provide the final output of the LSTM.

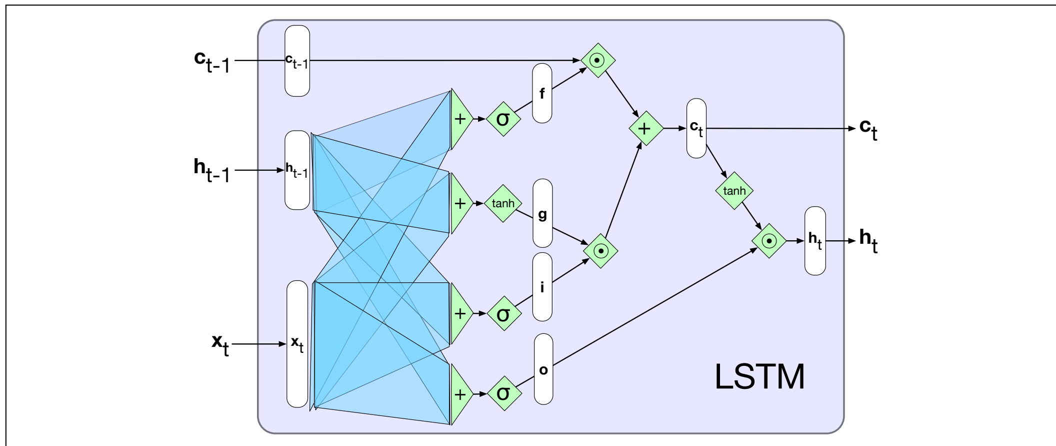


**Figure 13.13** A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$ and an updated context, $c_t$.

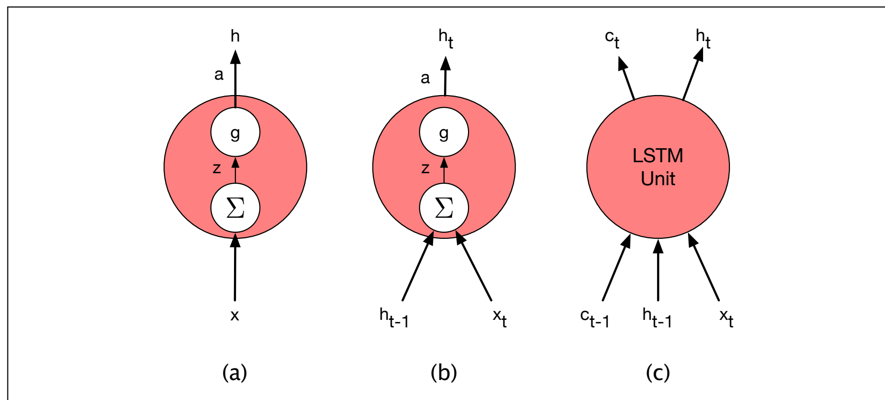


**Figure 13.14** Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

### 13.5.1 Gated Units, Layers and Networks

The neural units used in LSTMs are obviously much more complex than those used in basic feedforward networks. Fortunately, this complexity is encapsulated within the basic processing units, allowing us to maintain modularity and to easily experiment with different architectures. To see this, consider Fig. 13.14 which illustrates the inputs and outputs associated with each kind of unit.

At the far left, (a) is the basic feedforward unit where a single set of weights and a single activation function determine its output, and when arranged in a layer there are no connections among the units in the layer. Next, (b) represents the unit in a simple recurrent network. Now there are two inputs and an additional set of weights to go with it. However, there is still a single activation function and output.

The increased complexity of the LSTM units is encapsulated within the unit itself. The only additional external complexity for the LSTM over the basic recurrent unit (b) is the presence of the additional context vector as an input and output.

This modularity is key to the power and widespread applicability of LSTM units. LSTM units (or other varieties, like GRUs) can be substituted into any of the network architectures described in Section 13.4. And, as with simple RNNs, multi-layered

networks making use of gated units can be unrolled into deep feedforward networks and trained in the usual fashion with backpropagation. In practice, therefore, LSTMs rather than RNNs have become the standard unit for any modern system that makes use of recurrent networks.

## 13.6 Summary: Common RNN NLP Architectures

We've now introduced the RNN, seen advanced components like stacking multiple layers and using the LSTM version, and seen how the RNN can be applied to various tasks. Let's take a moment to summarize the architectures for these applications.

Fig. 13.15 shows the three architectures we've discussed so far: sequence labeling, sequence classification, and language modeling. In sequence labeling (for example for part of speech tagging), we train a model to produce a label for each input word or token. In sequence classification, for example for sentiment analysis, we ignore the output for each token, and only take the value from the end of the sequence (and similarly the model's training signal comes from backpropagation from that last token). In language modeling, we train the model to predict the next word at each token step. In the next section we'll introduce a fourth architecture, the **encoder-decoder**.

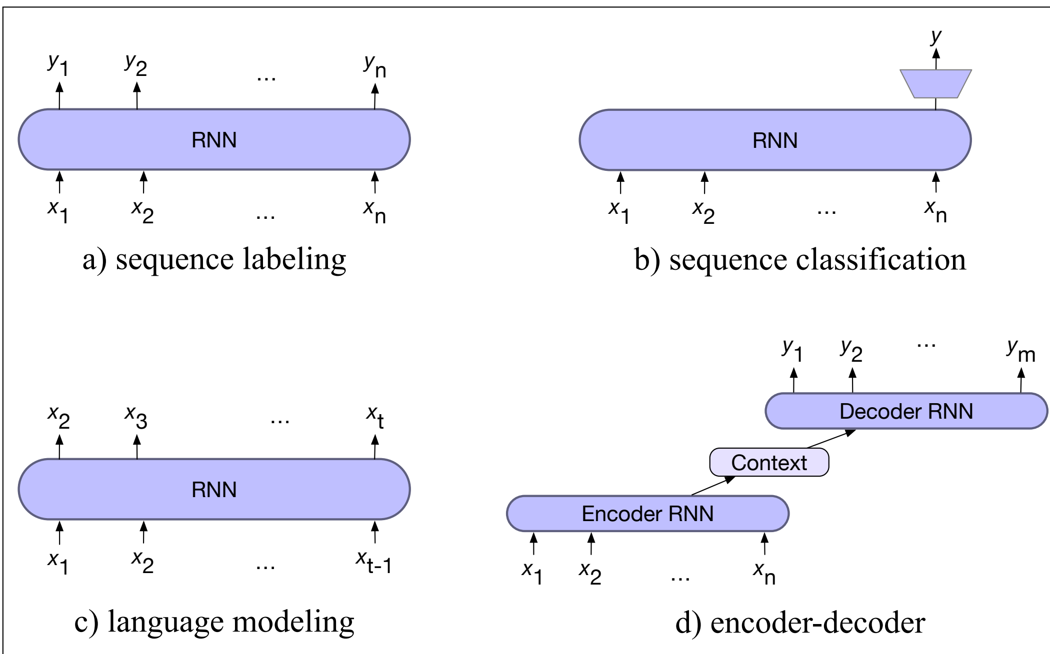


**Figure 13.15** Four architectures for NLP tasks. In sequence labeling (POS or named entity tagging) we map each input token $x_i$ to an output token $y_i$. In sequence classification we map the entire input sequence to a single class. In language modeling we output the next token conditioned on previous tokens. In the encoder model we have two separate RNN models, one of which maps from an input sequence **x** to an intermediate representation we call the **context**, and a second of which maps from the context to an output sequence **y**.

## 13.7 The Encoder-Decoder Model with RNNs

In this section we introduce the **encoder-decoder** model, which is used when we are taking an input sequence and translating it to an output sequence that is of a different length than the input, and doesn't align with it in a word-to-word way.

Those of you who already read Chapter 12 will have already seen this model in the transformer architecture, and its application to machine translation, but we introduce this architecture again here for those who come to the concepts in a different order and are reading about RNNs before transformers.

Recall that in the sequence labeling task, we have two sequences, but they are the same length (for example in part-of-speech tagging each token gets an associated tag), each input is associated with a specific output, and the labeling for that output takes mostly local information. Thus deciding whether a word is a verb or a noun, we look mostly at the word and the neighboring words.

By contrast, encoder-decoder models are used especially for tasks like machine translation, where the input sequence and output sequence can have different lengths and the mapping between a token in the input and a token in the output can be very indirect (in some languages the verb appears at the beginning of the sentence; in other languages at the end). We introduced machine translation in Chapter 12, but for now we'll just point out that the mapping for a sentence in English to a sentence in Tagalog or Yoruba can have very different numbers of words, and the words can be in a very different order.

**encoder-decoder**

**Encoder-decoder** networks, sometimes called **sequence-to-sequence** networks, are models capable of generating contextually appropriate, arbitrary length, output sequences given an input sequence. Encoder-decoder networks have been applied to a very wide range of applications including summarization, question answering, and dialogue, but they are particularly popular for machine translation.

The key idea underlying these networks is the use of an **encoder** network that takes an input sequence and creates a contextualized representation of it, often called the **context**. This representation is then passed to a **decoder** which generates a task-specific output sequence. Fig. 13.16 illustrates the architecture.

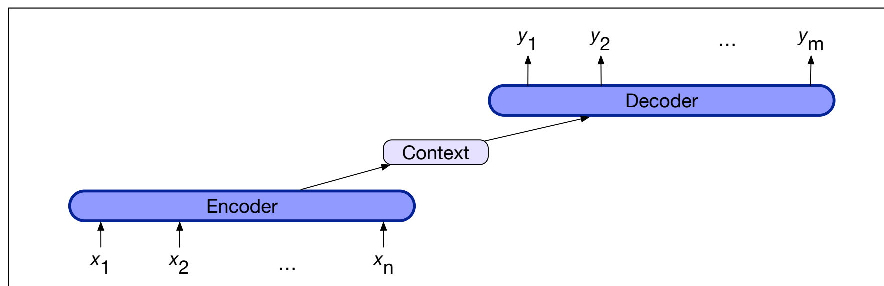


**Figure 13.16** The encoder-decoder architecture. The context is a function of the hidden representations of the input, and may be used by the decoder in a variety of ways.

Encoder-decoder networks consist of three conceptual components:

1. An **encoder** that accepts an input sequence, $x_{1:n}$, and generates a corresponding sequence of contextualized representations, $h_{1:n}$. LSTMs, convolutional networks, and transformers can all be employed as encoders.
2. A **context vector**, $c$, which is a function of $h_{1:n}$, and conveys the essence of the input to the decoder.

3. A **decoder**, which accepts $c$ as input and generates an arbitrary length sequence of hidden states $h_{1:m}$, from which a corresponding sequence of output states $y_{1:m}$, can be obtained. Just as with encoders, decoders can be realized by any kind of sequence architecture.

In this section we'll describe an encoder-decoder network based on a pair of RNNs, but we'll see in Chapter 12 how to apply them to transformers as well. We'll build up the equations for encoder-decoder models by starting with the conditional RNN language model $p(y)$, the probability of a sequence $y$.

Recall that in any language model, we can break down the probability as follows:

$$p(y) = p(y_1)p(y_2|y_1)p(y_3|y_1,y_2)\dots p(y_m|y_1,...,y_{m-1}) \quad (13.28)$$

In RNN language modeling, at a particular time $t$, we pass the prefix of $t-1$ tokens through the language model, using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix. We then use the final hidden state of the prefix as our starting point to generate the next token.

More formally, if $g$ is an activation function like *tanh* or ReLU, a function of the input at time $t$ and the hidden state at time $t-1$, and the softmax is over the set of possible vocabulary items, then at time $t$ the output $\mathbf{y}_t$ and hidden state $\mathbf{h}_t$ are computed as:

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t) \quad (13.29)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{h}_t) \quad (13.30)$$

We only have to make one slight change to turn this language model with autoregressive generation into an encoder-decoder model that is a translation model that can translate from a **source text** in one language to a **target text** in a second: add a **sentence separation** marker at the end of the source text, and then simply concatenate the target text.

**sentence separation**

Let's use <s> for our sentence separator token, and let's think about translating an English source text ("the green witch arrived"), to a Spanish sentence ("*llegó la bruja verde*" (which can be glossed word-by-word as 'arrived the witch green'). We could also illustrate encoder-decoder models with a question-answer pair, or a text-summarization pair.

Let's use $x$ to refer to the source text (in this case in English) plus the separator token <s>, and $y$ to refer to the target text $y$ (in this case in Spanish). Then an encoder-decoder model computes the probability $p(y|x)$ as follows:

$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)\dots p(y_m|y_1,...,y_{m-1},x) \quad (13.31)$$

Fig. 13.17 shows the setup for a simplified version of the encoder-decoder model (we'll see the full model, which requires the new concept of **attention**, in the next section).

Fig. 13.17 shows an English source text ("the green witch arrived"), a sentence separator token (<s>, and a Spanish target text ("*llegó la bruja verde*"). To translate a source text, we run it through the network performing forward inference to generate hidden states until we get to the end of the source. Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker. Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

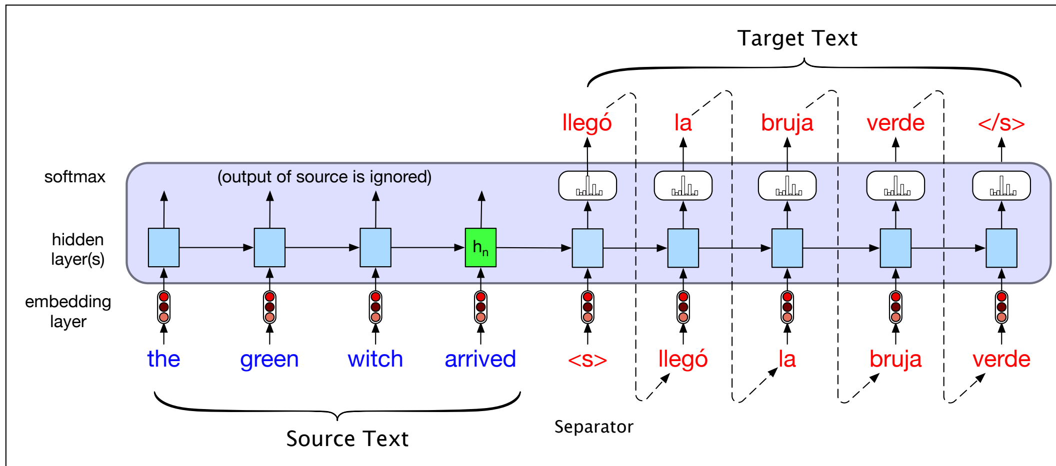


**Figure 13.17** Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

Let's formalize and generalize this model a bit in Fig. 13.18. (To help keep things straight, we'll use the superscripts $e$ and $d$ where needed to distinguish the hidden states of the encoder and the decoder.) The elements of the network on the left process the input sequence $x$ and comprise the **encoder**. While our simplified figure shows only a single network layer for the encoder, stacked architectures are the norm, where the output states from the top layer of the stack are taken as the final representation, and the encoder consists of stacked biLSTMs where the hidden states from top layers from the forward and backward passes are concatenated to provide the contextualized representations for each time step.

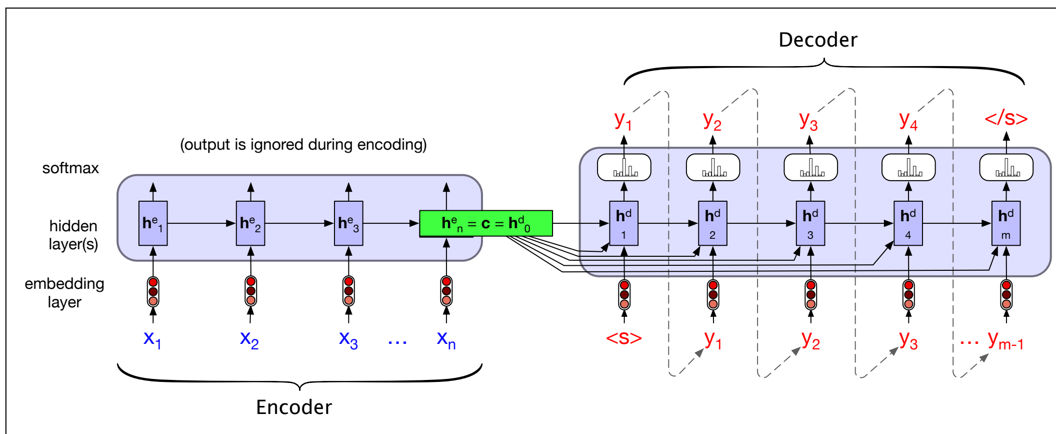


**Figure 13.18** A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN, $h_n^e$, serves as the context for the decoder in its role as $h_0^d$ in the decoder RNN, and is also made available to each decoder hidden state.

The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder, $\mathbf{h}_n^e$. This representation, also called **c** for **context**, is then passed to the decoder.

The simplest version of the **decoder** network would take this state and use it just to initialize the first hidden state of the decoder; the first decoder RNN cell would

use $c$ as its prior hidden state $\mathbf{h}_0^d$. The decoder would then autoregressively generate a sequence of outputs, an element at a time, until an end-of-sequence marker is generated. Each hidden state is conditioned on the previous hidden state and the output generated in the previous state.

As Fig. 13.18 shows, we do something more complex: we make the context vector **c** available to more than just the first decoder hidden state, to ensure that the influence of the context vector, **c**, doesn't wane as the output sequence is generated. We do this by adding **c** as a parameter to the computation of the current hidden state. using the following equation:

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \tag{13.32}$$

Now we're ready to see the full equations for this version of the decoder in the basic encoder-decoder model, with context available at each decoding timestep. Recall that $g$ is a stand-in for some flavor of RNN and $\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step:

$$\begin{aligned}
\mathbf{c} &= \mathbf{h}_n^e \\
\mathbf{h}_0^d &= \mathbf{c} \\
\mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\
\hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{h}_t^d)
\end{aligned} \tag{13.33}$$

Thus $\hat{\mathbf{y}}_t$ is a vector of probabilities over the vocabulary, representing the probability of each word occurring at time $t$. To generate text, we sample from this distribution $\hat{\mathbf{y}}_t$. For example, the greedy choice is simply to choose the most probable word to generate at each timestep. We discussed other sampling methods in Section 7.4.

### 13.7.1 Training the Encoder-Decoder Model

Encoder-decoder architectures are trained end-to-end. Each training example is a tuple of paired strings, a source and a target. Concatenated with a separator token, these source-target pairs can now serve as training data.

For MT, the training data typically consists of sets of sentences and their translations. These can be drawn from standard datasets of aligned sentence pairs, as we'll discuss in Section 12.2.2. Once we have a training set, the training itself proceeds as with any RNN-based language model. The network is given the source text and then starting with the separator token is trained autoregressively to predict the next word, as shown in Fig. 13.19.

Note the differences between training (Fig. 13.19) and inference (Fig. 13.17) with respect to the outputs at each time step. The decoder during inference uses its own estimated output $\hat{y}_t$ as the input for the next time step $x_{t+1}$. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating more tokens. In training, therefore, it is more common to use **teacher forcing** in the decoder. Teacher forcing means that we force the system to use the gold target token from training as the next input $x_{t+1}$, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$. This speeds up training.

teacher forcing

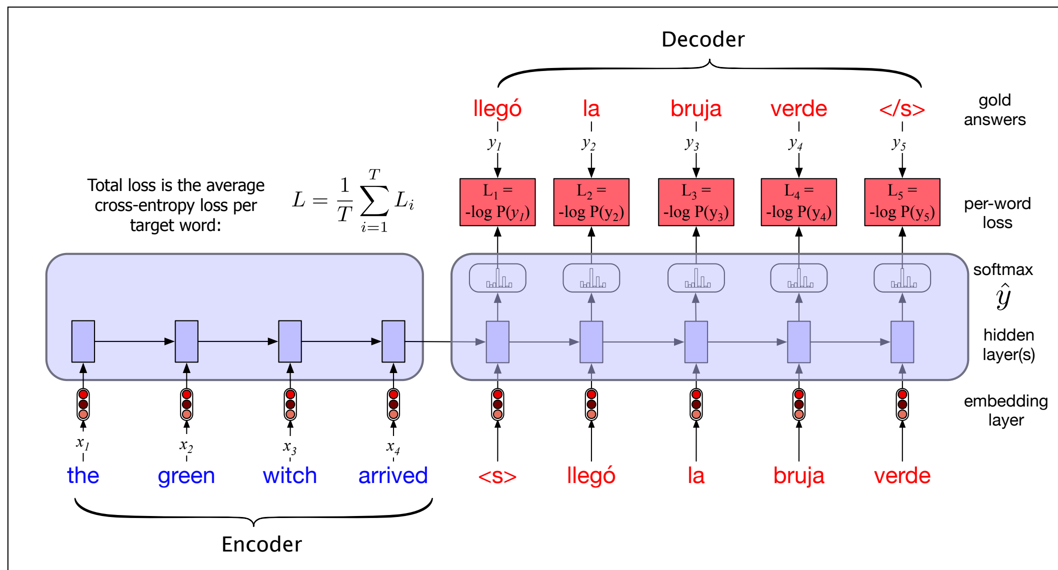


**Figure 13.19** Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution over $\hat{y}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence. This loss is then propagated through the decoder parameters and the encoder parameters.

## 13.8 Attention

The simplicity of the encoder-decoder model is its clean separation of the encoder—which builds a representation of the source text—from the decoder, which uses this context to generate a target text. In the model as we've described it so far, this context vector is $h_n$, the hidden state of the last ($n^{\text{th}}$) time step of the source text. This final hidden state is thus acting as a **bottleneck**: it must represent absolutely everything about the meaning of the source text, since the only thing the decoder knows about the source text is what's in this context vector (Fig. 13.20). Information at the beginning of the sentence, especially for long sentences, may not be equally well represented in the context vector.

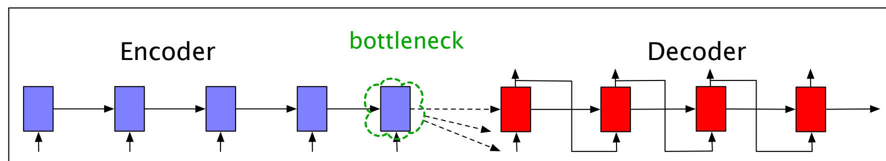


**Figure 13.20** Requiring the context $c$ to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

**attention mechanism**

The **attention mechanism** is a solution to the bottleneck problem, a way of allowing the decoder to get information from *all* the hidden states of the encoder, not just the last hidden state.

In the attention mechanism, as in the vanilla encoder-decoder model, the context vector **c** is a single vector that is a function of the hidden states of the encoder. But instead of being taken from the last hidden state, it's a weighted average of **all** the

hidden states of the encoder. And this weighted average is also informed by part of the decoder state as well, the state of the decoder right before the current token $i$. That is, $\mathbf{c}_i = f(\mathbf{h}^e_1 \ldots \mathbf{h}^e_n, \mathbf{h}^d_{i-1})$. The weights focus on ('attend to') a particular part of the source text that is relevant for the token $i$ that the decoder is currently producing. Attention thus replaces the static context vector with one that is dynamically derived from the encoder hidden states, but also informed by and hence different for each token in decoding.

This context vector, $\mathbf{c}_i$, is generated anew with each decoding step $i$ and takes all of the encoder hidden states into account in its derivation. We then make this context available during decoding by conditioning the computation of the current decoder hidden state on it (along with the prior hidden state and the previous output generated by the decoder), as we see in this equation (and Fig. 13.21):

$$\mathbf{h}^d_i \quad = \quad g(\hat{y}_{i-1}, \mathbf{h}^d_{i-1}, \mathbf{c}_i) \tag{13.34}$$

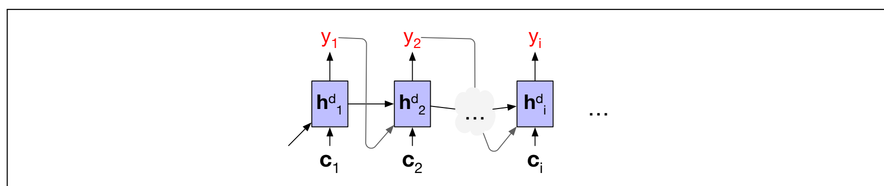


**Figure 13.21** The attention mechanism allows each hidden state of the decoder to see a different, dynamic, context, which is a function of all the encoder hidden states.

The first step in computing $\mathbf{c}_i$ is to compute how much to focus on each encoder state, how *relevant* each encoder state is to the decoder state captured in $\mathbf{h}^d_{i-1}$. We capture relevance by computing— at each state $i$ during decoding—a $score(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)$ for each encoder state $j$.

**dot-product attention**

The simplest such score, called **dot-product attention**, implements relevance as similarity: measuring how similar the decoder hidden state is to an encoder hidden state, by computing the dot product between them:

$$\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) \quad = \quad \mathbf{h}^d_{i-1} \cdot \mathbf{h}^e_j \tag{13.35}$$

The score that results from this dot product is a scalar that reflects the degree of similarity between the two vectors. The vector of these scores across all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

To make use of these scores, we'll normalize them with a softmax to create a vector of weights, $\alpha_{ij}$, that tells us the proportional relevance of each encoder hidden state $j$ to the prior hidden decoder state, $h^d_{i-1}$.

$$\begin{aligned}\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)) \\ &= \frac{\exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)}{\sum_k \exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_k))}\end{aligned} \tag{13.36}$$

Finally, given the distribution in $\alpha$, we can compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

$$\mathbf{c}_i \quad = \quad \sum_j \alpha_{ij}\, \mathbf{h}^e_j \tag{13.37}$$

With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically updated to reflect the needs of the decoder at each step of decoding. Fig. 13.22 illustrates an encoder-decoder network with attention, focusing on the computation of one context vector $\mathbf{c}_i$.

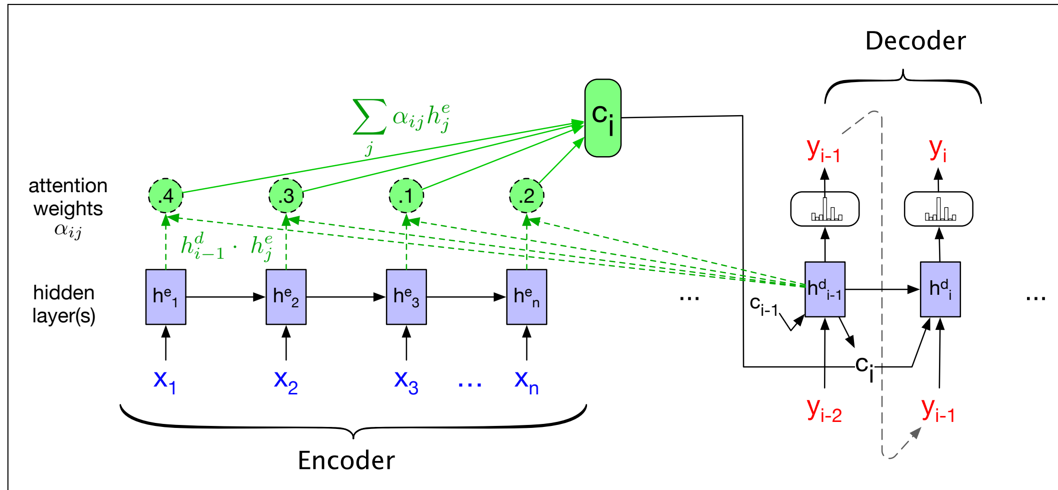


**Figure 13.22** A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$. The context value $\mathbf{c}_i$ is one of the inputs to the computation of $\mathbf{h}^d_i$. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state $\mathbf{h}^d_{i-1}$.

It's also possible to create more sophisticated scoring functions for attention models. Instead of simple dot product attention, we can get a more powerful function that computes the relevance of each encoder hidden state to the decoder hidden state by parameterizing the score with its own set of weights, $\mathbf{W}_s$.

$$\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) = \mathbf{h}^d_{i-1}\mathbf{W}_s\mathbf{h}^e_j \tag{13.38}$$

The weights $W_s$, which are then trained during normal end-to-end training, give the network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current application. This bilinear model also allows the encoder and decoder to use different dimensional vectors, whereas the simple dot-product attention requires that the encoder and decoder hidden states have the same dimensionality.

We'll return to the concept of attention when we define the transformer architecture in Chapter 8, which is based on a slight modification of attention called **self-attention**.

## 13.9 Summary

This chapter has introduced the concepts of recurrent neural networks and how they can be applied to language problems. Here's a summary of the main points that we covered:

- In simple Recurrent Neural Networks sequences are processed one element at a time, with the output of each neural unit at time $t$ based both on the current input at $t$ and the hidden layer from time $t-1$.

- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as **backpropagation through time** (BPTT).
- Simple recurrent networks fail on long inputs because of problems like **vanishing gradients**; instead modern systems use more complex gated architectures such as **LSTMs** that explicitly decide what to remember and forget in their hidden and context layers.
- Common language-based applications for RNNs include:
  - Probabilistic language modeling: assigning a probability to a sequence, or to the next element of a sequence given the preceding words.
  - Auto-regressive generation using a trained language model.
  - Sequence labeling like part-of-speech tagging, where each element of a sequence is assigned a label.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.
  - Encoder-decoder architectures, where an input is mapped to an output of different length and alignment.

# Historical Notes

Influential investigations of RNNs were conducted in the context of the Parallel Distributed Processing (PDP) group at UC San Diego in the 1980's. Much of this work was directed at human cognitive modeling rather than practical NLP applications (Rumelhart and McClelland 1986c, McClelland and Rumelhart 1986). Models using recurrence at the hidden layer in a feedforward network (Elman networks) were introduced by Elman (1990). Similar architectures were investigated by Jordan (1986) with a recurrence from the output layer, and Mathis and Mozer (1995) with the addition of a recurrent context layer prior to the hidden layer. The possibility of unrolling a recurrent network into an equivalent feedforward network is discussed in (Rumelhart and McClelland, 1986c).

In parallel with work in cognitive modeling, RNNs were investigated extensively in the continuous domain in the signal processing and speech communities (Giles et al. 1994, Robinson et al. 1996). Schuster and Paliwal (1997) introduced bidirectional RNNs and described results on the TIMIT phoneme transcription task.

While theoretically interesting, the difficulty with training RNNs and managing context over long sequences impeded progress on practical applications. This situation changed with the introduction of LSTMs in Hochreiter and Schmidhuber (1997) and Gers et al. (2000). Impressive performance gains were demonstrated on tasks at the boundary of signal processing and language processing including phoneme recognition (Graves and Schmidhuber, 2005), handwriting recognition (Graves et al., 2007) and most significantly speech recognition (Graves et al., 2013).

Interest in applying neural networks to practical NLP problems surged with the work of Collobert and Weston (2008) and Collobert et al. (2011). These efforts made use of learned word embeddings, convolutional networks, and end-to-end training. They demonstrated near state-of-the-art performance on a number of standard shared tasks including part-of-speech tagging, chunking, named entity recognition and semantic role labeling without the use of hand-engineered features.

Approaches that married LSTMs with pretrained collections of word-embeddings based on word2vec (Mikolov et al., 2013a) and GloVe (Pennington et al., 2014)

quickly came to dominate many common tasks: part-of-speech tagging (Ling et al., 2015), syntactic chunking (Søgaard and Goldberg, 2016), named entity recognition (Chiu and Nichols, 2016; Ma and Hovy, 2016), opinion mining (Irsoy and Cardie, 2014), semantic role labeling (Zhou and Xu, 2015a) and AMR parsing (Foland and Martin, 2016). As with the earlier surge of progress involving statistical machine learning, these advances were made possible by the availability of training data provided by CONLL, SemEval, and other shared tasks, as well as shared resources such as Ontonotes (Pradhan et al., 2007b), and PropBank (Palmer et al., 2005).

The modern neural encoder-decoder approach was pioneered by Kalchbrenner and Blunsom (2013), who used a CNN encoder and an RNN decoder. Cho et al. (2014) (who coined the name "encoder-decoder") and Sutskever et al. (2014) then showed how to use extended RNNs for both encoder and decoder. The idea that a generative decoder should take as input a soft weighting of the inputs, the central idea of attention, was first developed by Graves (2013) in the context of handwriting recognition. Bahdanau et al. (2015) extended the idea, named it "attention" and applied it to MT.