CHAPTER

7 Neural Networks and Neural Language Models

> "[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 words 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 6. 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, and also apply them to the simple task of language modeling: assigning probabilities to word sequences and predicting upcoming words. In subsequent chapters we'll introduce many other aspects of neural models, such as **recurrent neural networks** (Chapter 9), the **Transformer** (Chapter 10), and masked language modeling (Chapter 11).

7.1 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.

bias term

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:

$$z = b + \sum_i w_i x_i \tag{7.1}$$

vector

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**:

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{7.2}$$

As defined in Eq. 7.2, z is just a real valued number.

activation

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:

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

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 5:

sigmoid

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{7.3}$$

The sigmoid (shown in Fig. 7.1) 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 5.10 will be handy for learning.

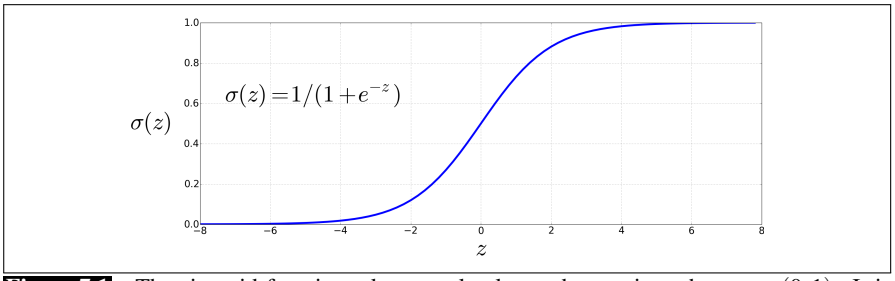

Figure 7.1 The sigmoid function takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Substituting Eq. 7.2 into Eq. 7.3 gives us the output of a neural unit:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+\exp(-(\mathbf{w} \cdot \mathbf{x} + b))} \tag{7.4}$$

Fig. 7.2 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.

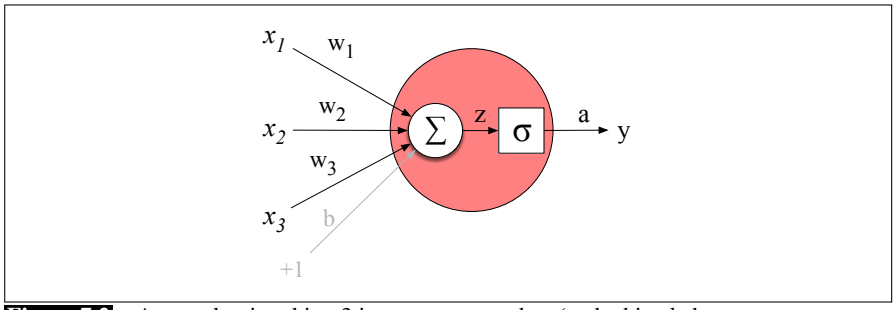

Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of 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. 7.3a; tanh is a variant of the sigmoid that ranges from -1 to +1:

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{7.5}$$

ReLU

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**, shown in Fig. 7.3b. It's just the same as z when z is positive, and 0 otherwise:

$$y = \text{ReLU(z)} = max(z, 0) \tag{7.6}$$

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

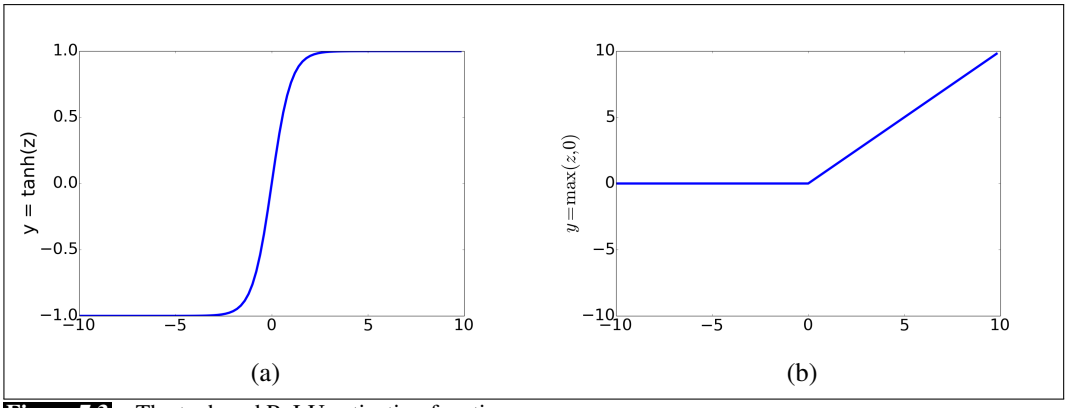

Figure 7.3 The tanh and ReLU activation functions.

saturated

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 7.5, 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.

vanishing gradient

7.2 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 does **not** have a 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. 7.2):

$$y = \begin{cases} 0, & \text{if } \mathbf{w}\cdot\mathbf{x}+b \leq 0 \\ 1, & \text{if } \mathbf{w}\cdot\mathbf{x}+b > 0 \end{cases} \tag{7.7}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs; Fig. 7.4 shows the necessary weights.

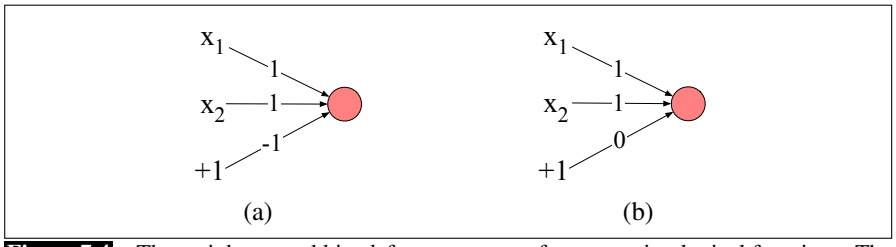

Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical 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. 7.5 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

7.2.1 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. 7.6 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].

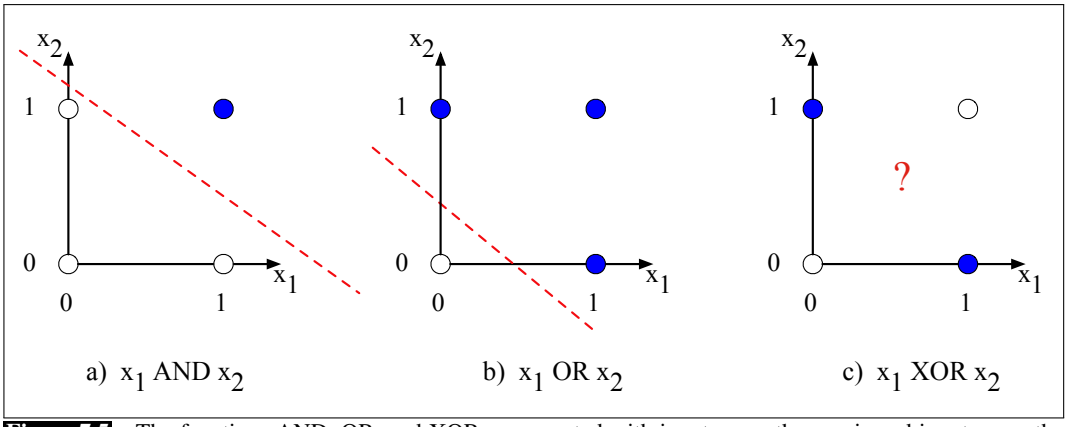

Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y-axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

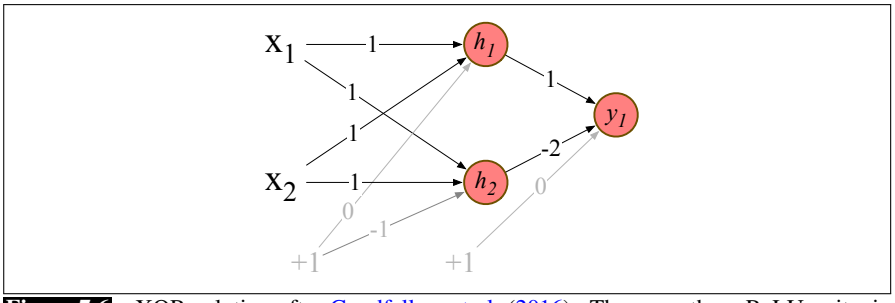

Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with 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. 7.7b 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. 7.6. But for real examples the weights for neural networks are learned automatically using the error backpropagation algorithm to be introduced in Section 7.5. 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.

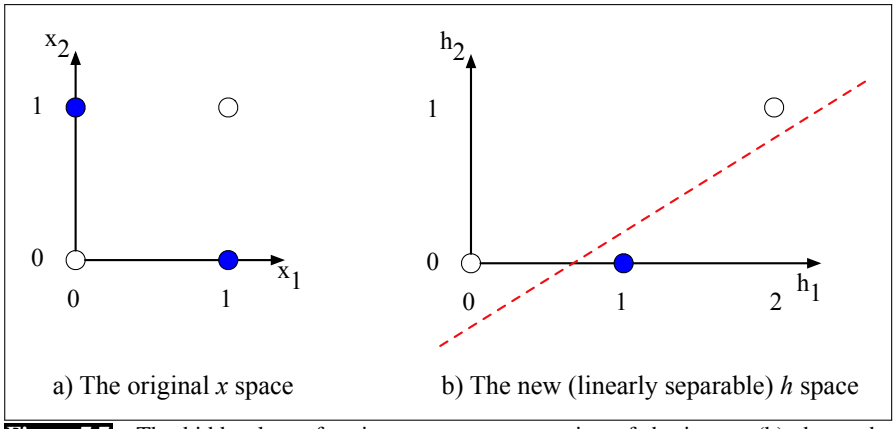

Figure 7.7 The hidden layer forming a new representation of the input. (b) shows the representation of the hidden layer, **h**, compared to the original input representation **x** in (a). Notice that the input point [0, 1] has been collapsed with the input point [1, 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

7.3 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 9 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 are purely linear, but modern networks are made up of units with non-linearities like sigmoids), but at some point the name stuck.

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

Fig. 7.8 shows a picture. The input layer **x** is a vector of simple scalar values just as we saw in Fig. 7.2.

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 7.1, taking a weighted sum of its inputs and then applying a non-linearity. In the standard architecture, each layer 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.

fully-connected

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. 7.8). Each element $\mathbf{W}_{ji}$ of the weight matrix $\mathbf{W}$ represents the weight of the connection from the ith input unit x_i to the jth 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 efficiently with simple matrix operations. In fact, the computation only has three

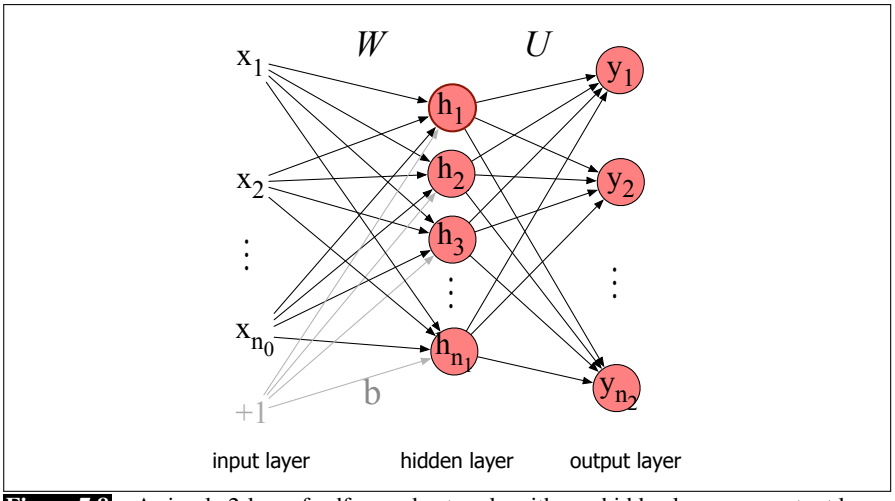

Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

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 σ as our activation function):

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{7.8}$$

Notice that we're applying the σ function here to a vector, while in Eq. 7.3 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 $[n_0, 1]$. 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. $[n_1, n_0]$.

Take a moment to convince yourself that the matrix multiplication in Eq. 7.8 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 7.2, 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 89 of Chapter 5. More generally for any vector $\mathbf{z}$ of dimensionality d, the softmax is defined as:

normalizing

softmax

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d} \exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{7.9}$$

Thus for example given a vector

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1], \tag{7.10}$$

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

$$\text{softmax}(\mathbf{z}) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010] \tag{7.11}$$

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 5.

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 5 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 σ 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 $\mathbf{b}$:

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \mathbf{y} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{7.12}$$

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.

7.3.1 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 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. 7.12 as follows:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= g^{[2]}(\mathbf{z}^{[2]}) \\
\hat{\mathbf{y}} &= \mathbf{a}^{[2]}
\end{aligned} \tag{7.13}$$

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

The activation functions $g(\cdot)$ are generally different at the final layer. Thus $g^{[2]}$ might be softmax for multinomial classification or sigmoid for binary classification, while ReLU or tanh might be the activation function $g(\cdot)$ at the internal layers.

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 (5.7).

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:

$$\begin{aligned}
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}\mathbf{W}^{[1]}\mathbf{x} + \mathbf{W}^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}'\mathbf{x} + \mathbf{b}'
\end{aligned} \tag{7.14}$$

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 often 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

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{7.15}$$

we'll use:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}) \tag{7.16}$$

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:

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j\right), \tag{7.17}$$

we'll instead use:

$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i\right), \tag{7.18}$$

where the value $\mathbf{W}_{j0}$ replaces what had been $\mathbf{b}_j$. Fig. 7.9 shows a visualization.

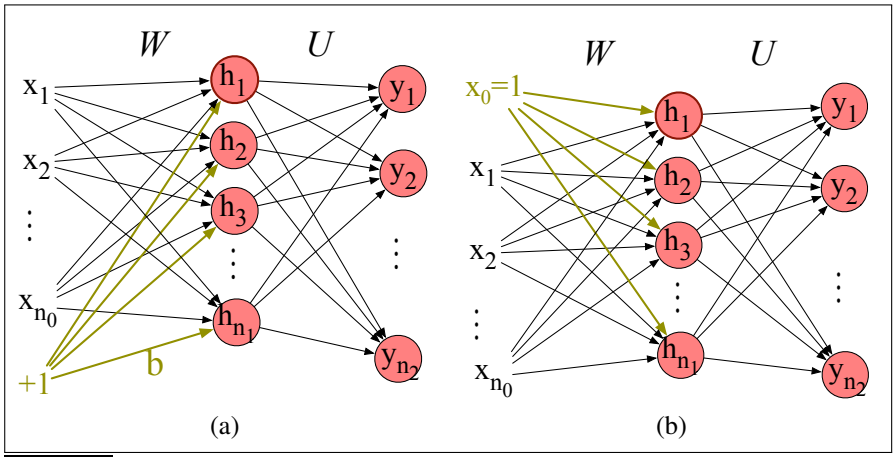

Figure 7.9 Replacing the bias node (shown in a) with x_0 (b).

We'll continue showing the bias as b when we go over the learning algorithm in Section 7.5, but then we'll switch to this simplified notation without explicit bias terms for the rest of the book.

7.4 Feedforward networks for NLP: Classification

Let's see how to apply feedforward networks to NLP tasks! In this section we'll look at classification tasks like sentiment analysis; in the next section we'll introduce neural language modeling.

Let's begin with a simple 2-layer sentiment classifier. You might imagine taking our logistic regression classifier from Chapter 5, which corresponds to a 1-layer network, and just adding a hidden layer. The input element $\mathbf{x}_i$ could be scalar features like those in Fig. 5.2, e.g., $\mathbf{x}_1$ = count(words $\in$ doc), $\mathbf{x}_2$ = count(positive lexicon words $\in$ doc), $\mathbf{x}_3$ = 1 if "no" $\in$ doc, and so on. 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 σ to stand for any non-linearity, whether sigmoid, ReLU or other).

$$\begin{aligned}\mathbf{x} &= [\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_N] \quad \text{(each } \mathbf{x}_i \text{ is a hand-designed feature)} \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{7.19}$$

Fig. 7.10 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.

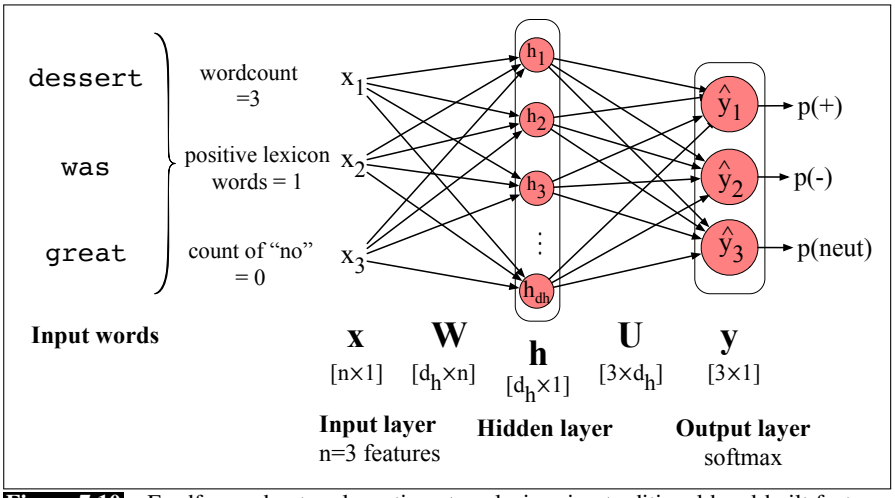

Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

Most applications of neural networks for NLP do something different, however. Instead of using hand-built human-engineered features as the input to our classifier, we draw on deep learning's ability to learn features from the data by representing words as embeddings, like the word2vec or GloVe embeddings we saw in Chapter 6. There are various ways to represent an input for classification. One simple baseline is to apply some sort of **pooling** function to the embeddings of all the words in the

pooling

input. For example, for a text with n input words/tokens $w_1, ..., w_n$, we can turn the n embeddings $\mathbf{e}(w_1), ..., \mathbf{e}(w_n)$ (each of dimensionality d) into a single embedding also of dimensionality d by just summing the embeddings, or by taking their mean (summing and then dividing by n):

$$\mathbf{x}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{e}(w_i) \tag{7.20}$$

There are many other options, like taking the element-wise max. The element-wise max of a set of n vectors is a new vector whose kth element is the max of the kth elements of all the n vectors. Here are the equations for this classifier assuming mean pooling; the architecture is sketched in Fig. 7.11:

$$\begin{aligned}\mathbf{x} &= \text{mean}(\mathbf{e}(w_1), \mathbf{e}(w_2), \ldots, \mathbf{e}(w_n)) \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})\end{aligned} \tag{7.21}$$

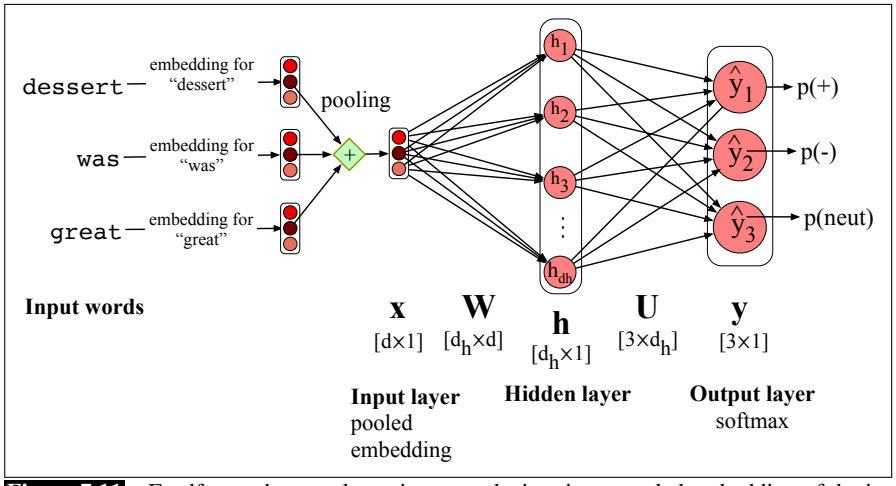

Figure 7.11 Feedforward network sentiment analysis using a pooled embedding of the input words.

While Eq. 7.21 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 vectoring 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 pooled embedding for input example $x^{(i)}$ (i.e., the vector $\mathbf{x}^{(i)}$). If the dimensionality of our pooled input embedding is d, $\mathbf{X}$ will be a matrix of shape $[m \times d]$.

We will then need to slightly modify Eq. 7.21. $\mathbf{X}$ is of shape $[m \times d]$ and $\mathbf{W}$ is of shape $[d_h \times d]$, so we'll have to 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 $[m \times d_h]$. The bias vector $\mathbf{b}$ from Eq. 7.21 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:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{\intercal}+\mathbf{b})\\ \mathbf{Z} &= \mathbf{H}\mathbf{U}^{\intercal}\\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{Z})\end{aligned} \tag{7.22}$$

The idea of using word2vec or GloVe embeddings as our input representation—and more generally the idea of relying on another algorithm to have already learned an embedding representation for our input words—is called **pretraining**. Using pretrained embedding representations, whether simple static word embeddings like word2vec or the much more powerful contextual embeddings we'll introduce in Chapter 11, is one of the central ideas of deep learning. (It's also possible, however, to train the word embeddings as part of an NLP task; we'll talk about how to do this in Section 7.7 in the context of the neural language modeling task.)

pretraining

7.5 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. 7.13, 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 5 for logistic regression, so the reader should be comfortable with that chapter before proceeding.

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 5.

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**.

7.5.1 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. 5.23:

$$L_{CE}(\hat{y},y) = -\log p(y|x) = -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \tag{7.23}$$

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 5 on page 101. 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:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{7.24}$$

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. 7.24 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

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c \quad \text{(where } c \text{ is the correct class)} \tag{7.25}$$

Plugging in the softmax formula from Eq. 7.9, and with K the number of classes:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)} \tag{7.26}$$

7.5.2 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. 7.27 (and derived in Section 5.10):

$$\begin{aligned} \frac{\partial L_{CE}(\hat{\mathbf{y}}, \mathbf{y})}{\partial w_j} &= (\hat{y} - y)\, \mathbf{x}_j \\ &= (\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y)\, \mathbf{x}_j \end{aligned} \tag{7.27}$$

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. 5.48, shown

for a particular weight $\mathbf{w}_k$ and input $\mathbf{x}_i$

$$\begin{aligned}\frac{\partial L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k - \hat{\mathbf{y}}_k)\mathbf{x}_i \\ &= -(\mathbf{y}_k - p(\mathbf{y}_k = 1|\mathbf{x}))\mathbf{x}_i \\ &= -\left(\mathbf{y}_k - \frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i \end{aligned} \tag{7.28}$$

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.

7.5.3 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. 7.12. 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.

7.5.4 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

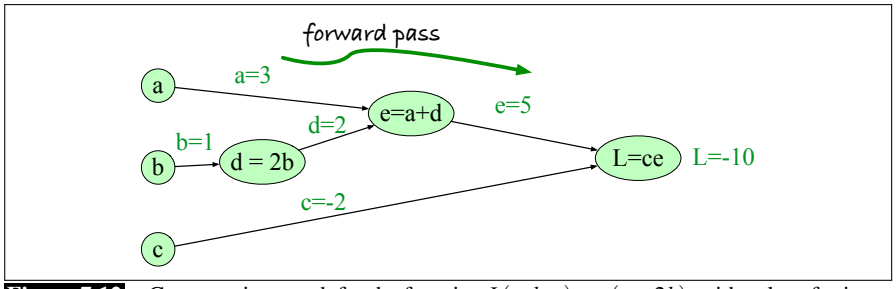

Figure 7.12 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{7.29}$$

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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{7.30}$$

The intuition of backward differentiation is to pass gradients back from the final node to all the nodes in the graph. Fig. 7.13 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.

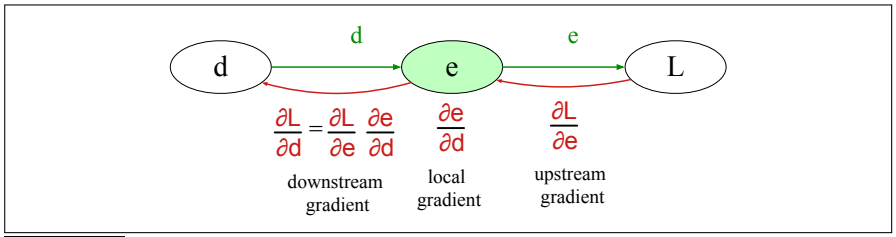

Figure 7.13 Each node (like e here) takes an upstream gradient, multiplies it by the local gradient (the gradient of its output with respect to its input), and uses the chain rule to compute a downstream gradient to be passed on to a prior node. A node may have multiple local gradients if it has multiple inputs.

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

$$\frac{\partial L}{\partial c} = e \tag{7.31}$$

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

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{7.32}$$

Eq. 7.32 and Eq. 7.31 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. 7.14 shows the backward pass.

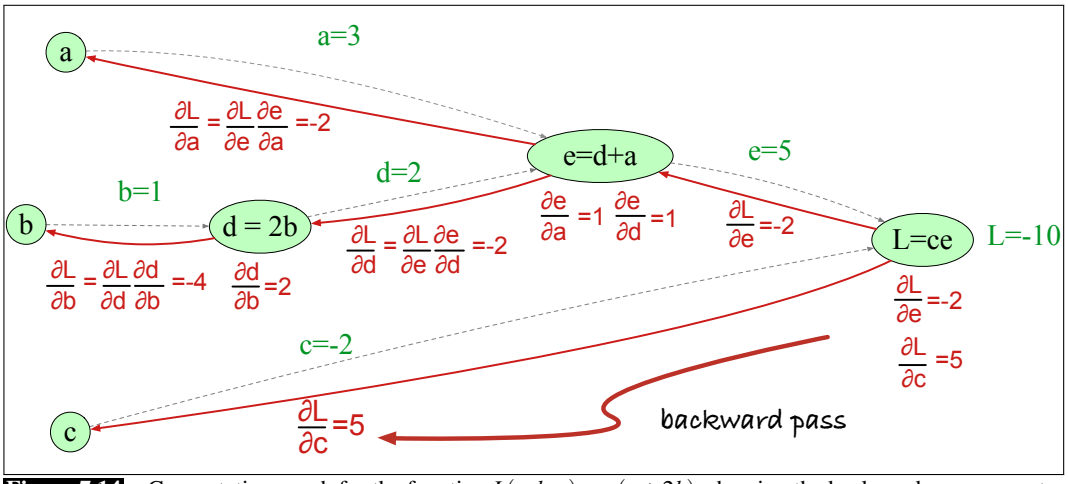

Figure 7.14 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Backward differentiation for a neural network

Of course computation graphs for real neural networks are much more complex. Fig. 7.15 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:

$$\begin{aligned} \mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\ \mathbf{a}^{[1]} &= \text{ReLU}(\mathbf{z}^{[1]}) \\ z^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{7.33}$$

For the backward pass we'll also need to compute the loss L. The loss function for binary sigmoid output from Eq. 7.23 is

$$L_{CE}(\hat{y},y) = -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \tag{7.34}$$

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

$$L_{CE}(a^{[2]},y) = -\left[y\log a^{[2]}+(1-y)\log(1-a^{[2]})\right] \tag{7.35}$$

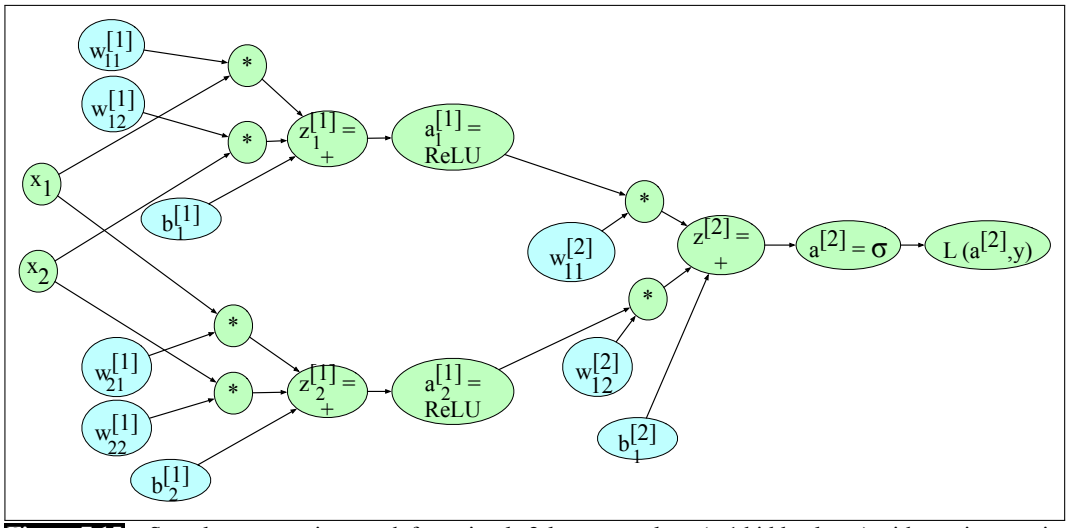

Figure 7.15 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input units and 2 hidden units. We've adjusted the notation a bit to avoid long equations in the nodes by just mentioning the function that is being computed, and the resulting variable name. Thus the * to 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 5.10 the derivative of the sigmoid σ:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{7.36}$$

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

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z) \tag{7.37}$$

The derivative of the ReLU is

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; z<0 \\ 1 & for \;\; z\geq 0 \end{cases} \tag{7.38}$$

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:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z} \tag{7.39}$$

So let's first compute $\frac{\partial L}{\partial a^{[2]}}$, taking the derivative of Eq. 7.35, repeated here:

$$\begin{aligned} L_{CE}(a^{[2]},y) &= -\left[y\log a^{[2]}+(1-y)\log(1-a^{[2]})\right] \\ \frac{\partial L}{\partial a^{[2]}} &= -\left(\left(y\frac{\partial \log(a^{[2]})}{\partial a^{[2]}}\right)+(1-y)\frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}}\right) \\ &= -\left(\left(y\frac{1}{a^{[2]}}\right)+(1-y)\frac{1}{1-a^{[2]}}(-1)\right) \\ &= -\left(\frac{y}{a^{[2]}}+\frac{y-1}{1-a^{[2]}}\right) \end{aligned} \tag{7.40}$$

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:

$$\begin{aligned} \frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z} \\ &= -\left(\frac{y}{a^{[2]}}+\frac{y-1}{1-a^{[2]}}\right)a^{[2]}(1-a^{[2]}) \\ &= a^{[2]}-y \end{aligned} \tag{7.41}$$

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.

7.5.5 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.

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). Tuning of **hyperparameters** is also important. The parameters of a neural network are the weights W and biases 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 η, 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).

hyperparameter

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.

7.6 Feedforward Neural Language Modeling

As our second application of feedforward networks, let's consider **language modeling**: predicting upcoming words from prior words. Neural language modeling—based on the transformer architecture that we will see in Chapter 10—is the algorithm the underlies all of modern NLP. In this section and the next we'll introduce a simpler version of neural language models for feedforward networks, an algorithm first introduced by Bengio et al. (2003). The feedforward language model introduces many of the important concepts of neural language modeling, concepts we'll return to as we describe more powerful models inChapter 9 and Chapter 10.

Neural language models have many advantages over the n-gram language models of Chapter 3. Compared to n-gram models, neural language models can handle much longer histories, can generalize better over contexts of similar words, and are more accurate at word-prediction. On the other hand, neural net language models are much more complex, are slower and need more 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 (LM) 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:

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1}) \tag{7.42}$$

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".

7.6.1 Forward inference in the neural language model

forward inference

Let's walk through **forward inference** or **decoding** for neural language models. Forward inference is the task, given an input, of running a forward pass on the

network to produce a probability distribution over possible outputs, in this case next words.

one-hot vector

We first represent each of the N previous words as a one-hot vector of length $|V|$, i.e., with one dimension for each word in the vocabulary. A **one-hot vector** is a vector that has one element equal to 1—in the dimension corresponding to that word's index in the vocabulary— while all the other elements are set to zero. Thus in a one-hot representation for the word "toothpaste", supposing it is V_5, i.e., index 5 in the vocabulary, $x_5 = 1$, and $x_i = 0 \ \forall i \neq 5$, as shown here:

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

The feedforward neural language model (sketched in Fig. 7.17) has a moving window that can see N words into the past. We'll let N equal 3, so the 3 words w_{t-1}, w_{t-2}, and w_{t-3} are each represented as a one-hot vector. We then multiply these one-hot vectors by the embedding matrix **E**. The embedding weight matrix **E** has a column for each word, each a column vector of d dimensions, and hence has dimensionality $d \times |V|$. Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant column vector for word i, resulting in the embedding for word i, as shown in Fig. 7.16.

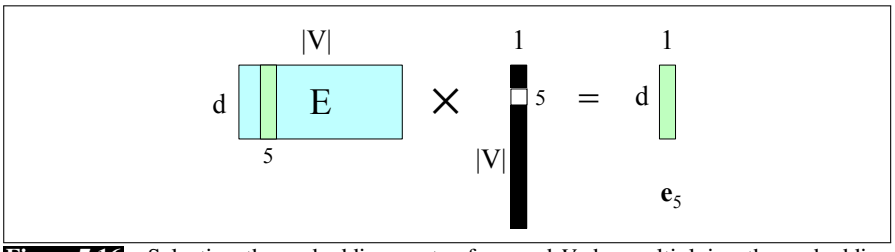

Figure 7.16 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

The 3 resulting embedding vectors are concatenated to produce **e**, the embedding layer. This is followed by a hidden layer and an output layer whose softmax produces 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).

Here's the algorithm in detail for our mini example:

1. **Select three embeddings from E**: Given the three previous words, we look up their indices, create 3 one-hot vectors, and then multiply each by the embedding matrix **E**. Consider w_{t-3}. The one-hot vector for 'for' (index 35) is multiplied by the embedding matrix **E**, to give the first part of the first hidden layer, the **embedding layer**. Since each column of the input matrix **E** is an embedding for a word, and the input is a one-hot column vector $\mathbf{x_i}$ for word V_i, the embedding layer for input w will be $\mathbf{Ex_i} = \mathbf{e_i}$, the embedding for word i. We now concatenate the three embeddings for the three context words to produce the embedding layer **e**.

embedding layer

2. **Multiply by W**: We multiply by W (and add b) and pass through the ReLU (or other) activation function to get the hidden layer h.
3. **Multiply by U**: h is now multiplied by U
4. **Apply softmax**: After the softmax, each node i in the output layer estimates the probability $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3})$

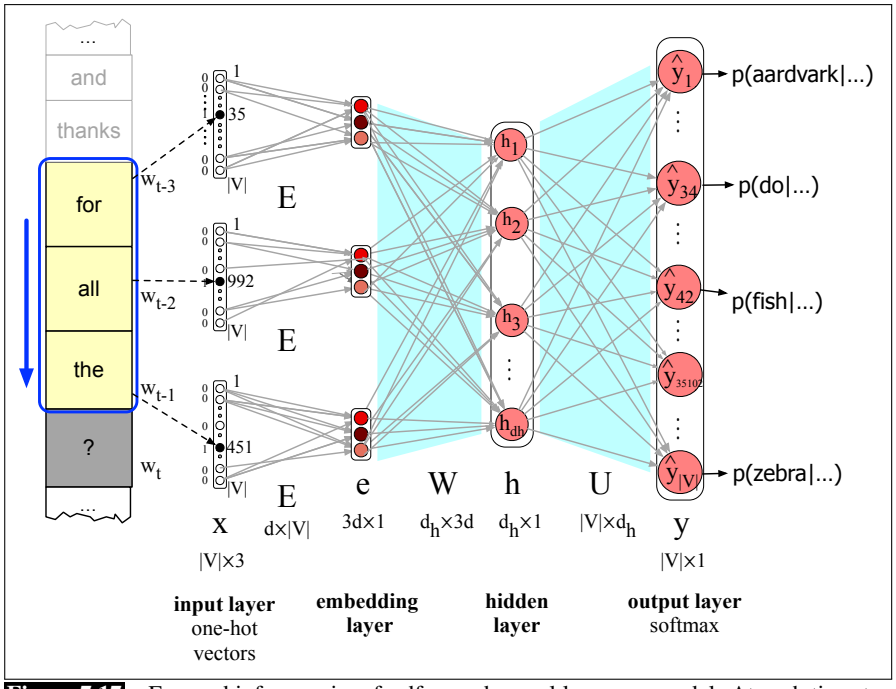

Figure 7.17 Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix $\mathbf{E}$), and concatenates the 3 resulting embeddings to get the embedding layer $\mathbf{e}$. The embedding vector $\mathbf{e}$ is multiplied by a 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 $\mathbf{U}$. Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.

In summary, the equations for a neural language model with a window size of 3, given one-hot input vectors for each input context word, are:

$$\begin{aligned}\mathbf{e} &= [\mathbf{E}\mathbf{x}_{\mathbf{t}-3};\mathbf{E}\mathbf{x}_{\mathbf{t}-2};\mathbf{E}\mathbf{x}_{\mathbf{t}-1}]\\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{e}+\mathbf{b})\\ \mathbf{z} &= \mathbf{U}\mathbf{h}\\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})\end{aligned} \tag{7.43}$$

Note that we formed the embedding layer $\mathbf{e}$ by concatenating the 3 embeddings for the three context vectors; we'll often use semicolons to mean concatenation of vectors.

7.7 Training the neural language model

self-training

The high-level intuition of training neural language models, whether the simple feedforward language models we describe here or the more powerful transformer language models of Chapter 10, is the idea of **self-training** or **self-supervision** that we saw in Chapter 6 for learning word representations. In self-training for language modeling, we take a corpus of text as training material and at each time step t ask the model to predict the next word. At first it will do poorly at this task, but since

in each case we know the correct answer (it's the next word in the corpus!) we can easily train it to be better at predicting the correct 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.

freeze

In practice, training the model means setting the parameters $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$. For some tasks, it's ok to **freeze** the embedding layer $\mathbf{E}$ with initial word2vec values. Freezing means we use word2vec or some other pretraining algorithm to compute the initial embedding matrix $\mathbf{E}$, and then hold it constant while we only modify $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{b}$, i.e., we don't update $\mathbf{E}$ during language model training. However, often we'd like to learn the embeddings simultaneously with training the network. This is useful when the task the network is designed for (like sentiment classification, translation, or parsing) places strong constraints on what makes a good representation for words.

Let's see how to train the entire model including $\mathbf{E}$, i.e. to set all the parameters $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$. We'll do this via gradient descent (Fig. 5.6), using error backpropagation on the computation graph to compute the gradient. Training thus not only sets the weights $\mathbf{W}$ and $\mathbf{U}$ of the network, but also as we're predicting upcoming words, we're learning the embeddings $\mathbf{E}$ for each word that best predict upcoming words.

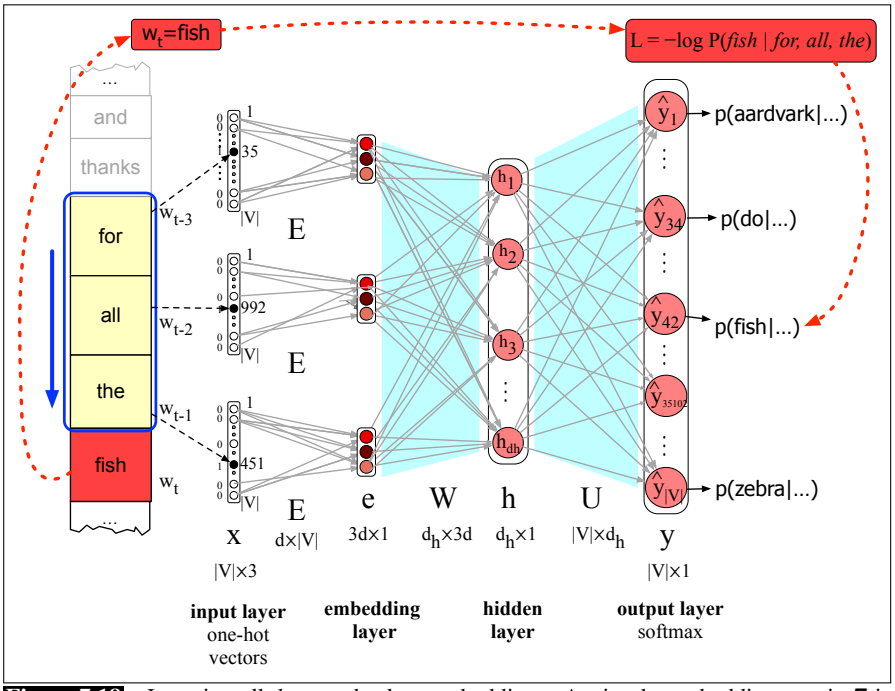

Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix $\mathbf{E}$ is shared among the 3 context words.

Fig. 7.18 shows the set up for a window size of N=3 context words. The input $\mathbf{x}$ consists of 3 one-hot vectors, fully connected to the embedding layer via 3 instantiations of the embedding matrix E. We don't want to learn separate weight matrices for mapping each of the 3 previous words to the projection layer. We want one single embedding dictionary E that's shared among these three. That's because over time, many different words will appear as w_{t-2} or w_{t-1}, and we'd like to just represent

each word with one vector, whichever context position it appears in. Recall that the embedding weight matrix E has a column for each word, each a column vector of d dimensions, and hence has dimensionality $d \times |V|$.

Generally training proceeds by taking as input a very long text, concatenating all the sentences, starting with random weights, and then iteratively moving through the text predicting each word w_t. At each word w_t, we use the cross-entropy (negative log likelihood) loss. Recall that the general form for this (repeated from Eq. 7.25) is:

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)} \tag{7.44}$$

For language modeling, the classes are the words in the vocabulary, so $\hat{y}_i$ here means the probability that the model assigns to the correct next word w_t:

$$L_{\text{CE}} = -\log p(w_t|w_{t-1}, ..., w_{t-n+1}) \tag{7.45}$$

The parameter update for stochastic gradient descent for this loss from step s to $s+1$ is then:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t|w_{t-1}, ..., w_{t-n+1})\right]}{\partial \theta} \tag{7.46}$$

This gradient can be computed in any standard neural network framework which will then backpropagate through $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$.

Training the parameters to minimize loss will result both in an algorithm for language modeling (a word predictor) but also a new set of embeddings $\mathbf{E}$ that can be used as word representations for other tasks.

7.8 Summary

- Neural networks are built out of **neural units**, originally inspired by biological neurons but now simply an abstract computational device.
- 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.

Bibliographical and 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 9 and Chapter 16.

There are a number of excellent books on the subject. Goldberg (2017) has superb coverage of neural networks for natural language processing. For neural networks in general see Goodfellow et al. (2016) and Nielsen (2015).