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 human 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 large scale problems 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), **encoder-decoder** models, **attention** and the **Transformer** (Chapter 10).

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**:

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

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 ReLU) but it's pedagogically convenient to start with the

sigmoid

sigmoid function since we saw it in Chapter 5:

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

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 **??** will be handy for learning.

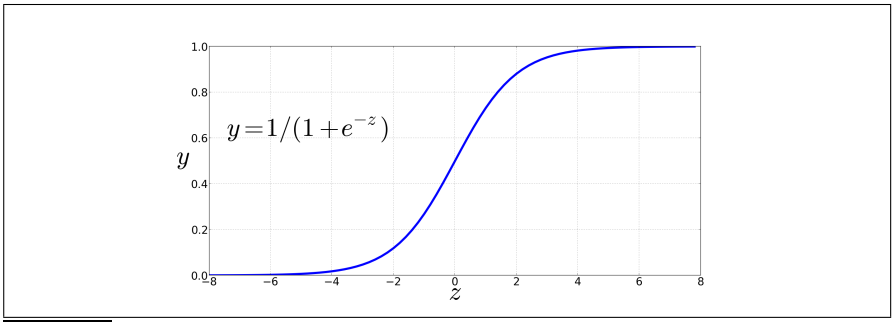

Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

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

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

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.

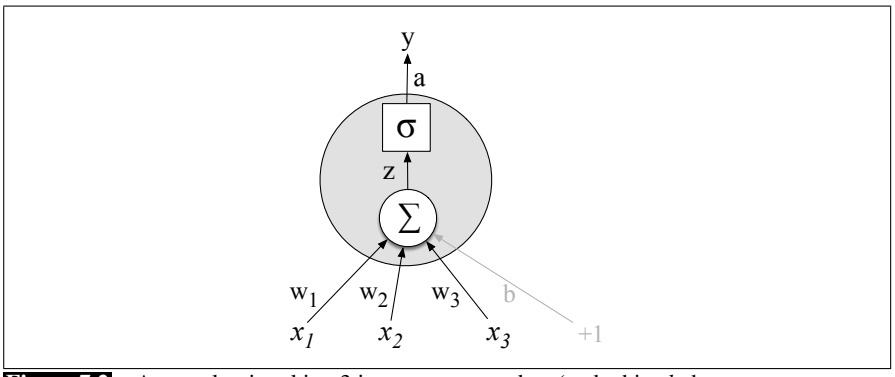

Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of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

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

What would this unit do with the following input vector:

$$x = [0.5, 0.6, 0.1]$$

The resulting output y would be:

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

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:

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

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 x when x is positive, and 0 otherwise:

$$y = max(x, 0) \tag{7.6}$$

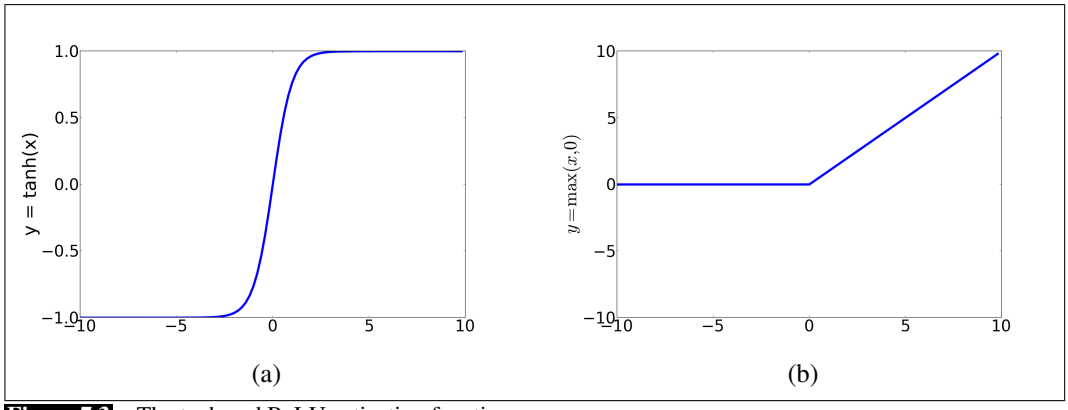

Figure 7.3 The tanh and ReLU activation functions.

saturated

These activation functions have different properties that make them useful for different language applications or network architectures. For example the rectifier function has nice properties that 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, which causes problems for learning. Rectifiers don't have this problem, since the output of values close to 1 also approaches 1 in a nice gentle linear way. By contrast, the tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean.

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 w, input x, and bias b as in Eq. 7.2):

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

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.

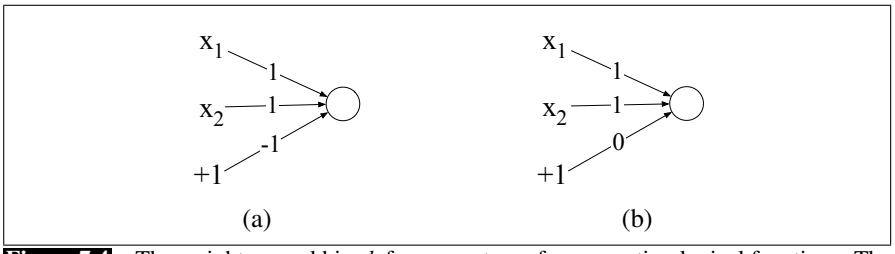

Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, showing 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 perception 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: $x_2 = -(w_1/w_2)x_1 - b$.) 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 units. Let's see an example of how to do this from Goodfellow et al. (2016) that computes XOR using two layers of ReLU-based units. 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 for each ReLU that correctly computes the XOR function.

Let's walk through what happens with the input 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 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].

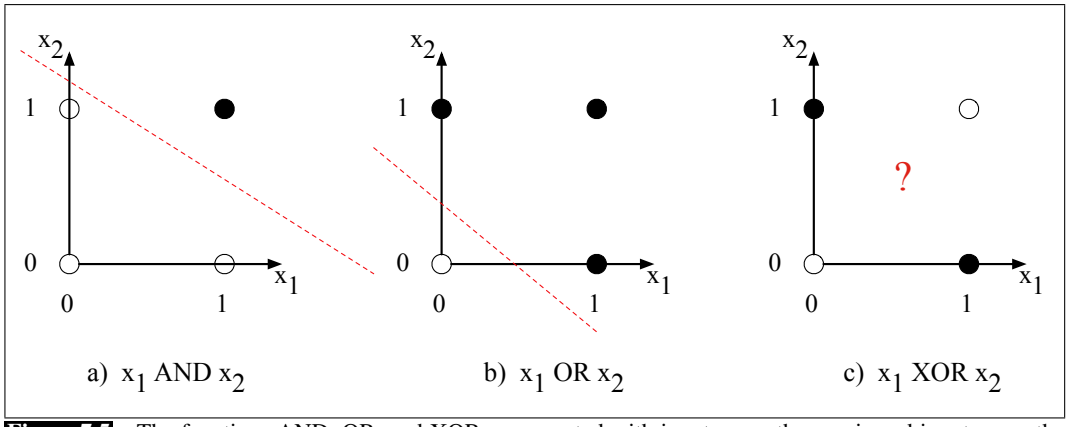

Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled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

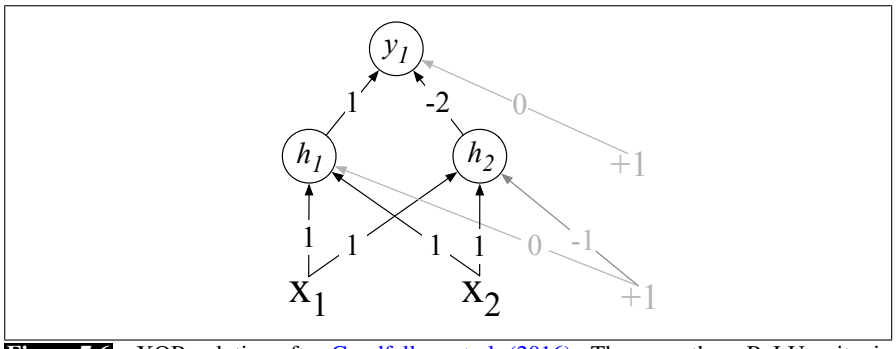

Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

It's also instructive to look at the intermediate results, the outputs of the two hidden nodes h_0 and h_1. We showed in the previous paragraph that the h vector for the inputs x = [0 0] was [0 0]. Fig. 7.7b shows the values of the h layer for all 4 inputs. Notice that hidden representations of the two input points x = [0 1] and x = [1 0] (the two cases with XOR output = 1) are merged to the single point 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 for 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 7.4. 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.

Note that the solution to the XOR problem requires a network of units with nonlinear activation functions. A network made up of simple linear (perceptron) units cannot solve the XOR problem. This is because a network formed by many layers of purely linear units can always be reduced (shown to be computationally identical

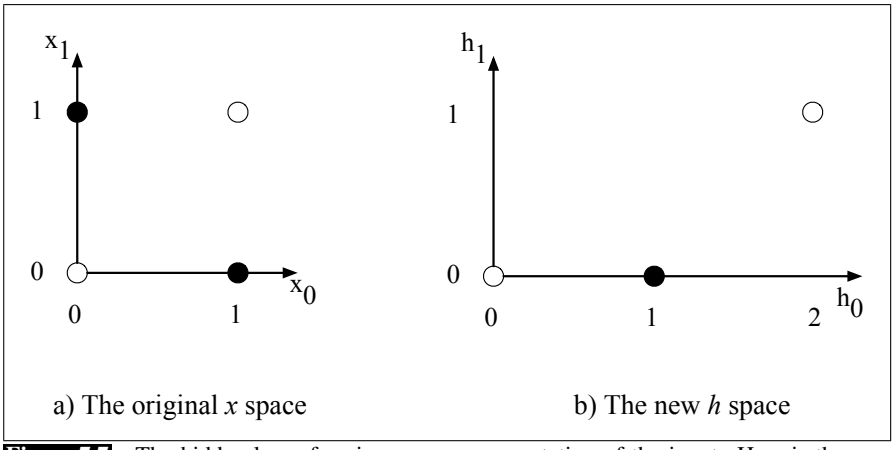

Figure 7.7 The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, h, compared to the original input representation x. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive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

to) a single layer of linear units with appropriate weights, and we've already shown (visually, in Fig. 7.5) that a single unit cannot solve the XOR problem.

7.3 Feed-Forward 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 units are simply scalar values just as we saw in Fig. 7.2.

hidden layer

The core of the neural network is the **hidden layer** formed of **hidden units**, 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

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 parameters w (the weight vector) and b (the bias scalar). We represent the parameters for the entire hidden layer by combining the weight vector $\mathbf{w}_i$ and bias b_i for each unit i into a single weight matrix W and a single bias vector $\mathbf{b}$ for the whole layer (see Fig. 7.8). Each element W_{ij} of the weight matrix W represents the weight of the connection from the ith input unit x_i to

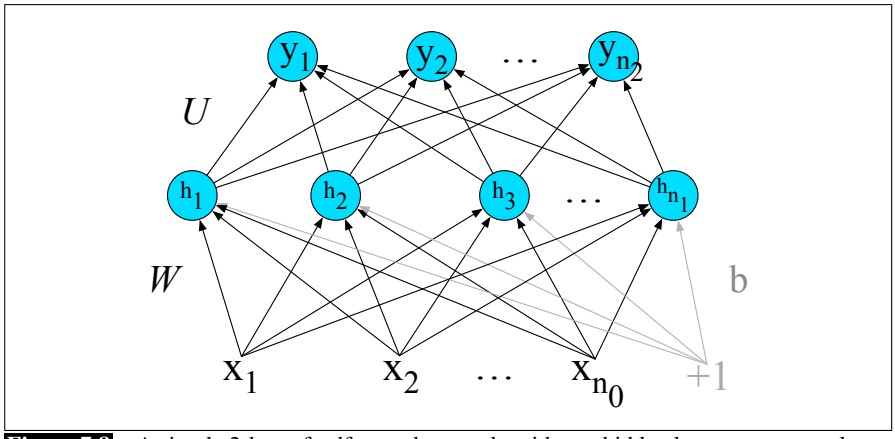

Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

the jth hidden unit h_j.

The advantage of using a single matrix 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 steps: multiplying the weight matrix by the input vector x, adding the bias vector 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 h, is thus the following, using the sigmoid function σ:

$$h = \sigma(Wx + b) \tag{7.8}$$

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 x is a vector of real numbers of dimension n_0, or more formally $x \in \mathbb{R}^{n_0}$. Let's call the hidden layer layer 1 and the output layer layer 2. The hidden layer has dimensionality n_1, so $h \in \mathbb{R}^{n_1}$ and also $b \in \mathbb{R}^{n_1}$ (since each hidden unit can take a different bias value). And the weight matrix W has dimensionality $W \in \mathbb{R}^{n_1 \times n_0}$.

Take a moment to convince yourself that the matrix multiplication in Eq. 7.8 will compute the value of each h_j as $\sigma\left(\sum_{i=1}^{n_x} w_{ij} x_i + b_j\right)$.

As we saw in Section 7.2, the resulting value 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 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 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 thus 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 U), but some models don't include a bias vector 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 (h) to produce the intermediate output z.

$$z = Uh$$

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

However, 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 **??** of Chapter 5. For a vector z of dimensionality d, the softmax is defined as:

normalizing

softmax

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \le i \le d \tag{7.9}$$

Thus for example given a vector z=[0.6 1.1 -1.5 1.2 3.2 -1.1], softmax(z) is [0.055 0.090 0.0067 0.10 0.74 0.010].

You may recall that softmax was exactly what is used to create a probability distribution from a vector of real-valued numbers (computed from summing weights times features) in logistic regression in Chapter 5.

That means we can think of a neural network classifier with one hidden layer as building a vector h which is a hidden layer representation of the input, and then running standard logistic regression on the features that the network develops in h. By contrast, in Chapter 5 the features were mainly designed by hand via feature templates. So a neural network is like logistic regression, but (a) with many layers, since a deep neural network is like layer after layer of logistic regression classifiers, and (b) 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 x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:

$$\begin{aligned} h &= \sigma(Wx+b) \\ z &= Uh \\ y &= \text{softmax}(z) \end{aligned} \tag{7.10}$$

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.

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 $W^{[1]}$ will mean the weight matrix for the (first) hidden layer, and $b^{[1]}$ 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 $a^{[i]}$ to mean the output from layer i, and $z^{[i]}$ to mean the

combination of weights and biases $W^{[i]}a^{[i-1]} + b^{[i]}$. The 0th layer is for inputs, so the inputs x we'll refer to more generally as $a^{[0]}$.

Thus we can re-represent our 2-layer net from Eq. 7.10 as follows:

$$\begin{aligned} z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= g^{[1]}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= g^{[2]}(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{7.11}$$

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

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.

7.4 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. 7.11, is $\hat{y}$, the system's estimate of the true y. The goal of the training procedure is to learn parameters $W^{[i]}$ and $b^{[i]}$ 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. Here is one part where learning for neural networks is more complex than for logistic logistic regression. 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 **reverse differentiation**.

7.4.1 Loss function

cross-entropy loss

The **cross-entropy loss** that is used in neural networks is the same one we saw for logistic regression.

In fact, if the neural network is being used as a binary classifier, with the sigmoid at the final layer, the loss function is exactly the same as we saw with logistic regression in Eq. **??**:

$$L_{CE}(\hat{y},y) = -\log p(y|x) = -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{7.12}$$

What about if the neural network is being used as a multinomial classifier? Let y be a vector over the C classes representing the true output probability distribution. The cross-entropy loss here is

$$L_{CE}(\hat{y},y) = -\sum_{i=1}^{C} y_i \log\hat{y}_i \tag{7.13}$$

We can simplify this equation further. Assume this is a **hard classification** task, meaning that only one class is the correct one, and that there is one output unit in y for each class. If the true class is i, then y is a vector where $y_i = 1$ and $y_j = 0 \;\; \forall j \neq i$. A vector like this, with one value=1 and the rest 0, is called a **one-hot vector**. Now let $\hat{y}$ be the vector output from the network. The sum in Eq. 7.13 will be 0 except for the true class. Hence the cross-entropy loss is simply the log probability of the correct class, and we therefore also call this the **negative log likelihood loss**:

negative log likelihood loss

$$L_{CE}(\hat{y},y) = -\log\hat{y}_i \tag{7.14}$$

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

$$L_{CE}(\hat{y},y) = -\log\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{7.15}$$

7.4.2 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. 7.16 (and derived in Section **??**):

$$\begin{aligned}\frac{\partial L_{CE}(w,b)}{\partial w_j} &= (\hat{y}-y)\,x_j \\ &= (\sigma(w\cdot x+b)-y)\,x_j\end{aligned} \tag{7.16}$$

Or for a network with one hidden layer and softmax output, we could use the derivative of the softmax loss from Eq. **??**:

$$\begin{aligned}\frac{\partial L_{CE}}{\partial w_k} &= (1\{y=k\} - p(y=k|x))x_k \\ &= \left(1\{y=k\} - \frac{e^{w_k\cdot x+b_k}}{\sum_{j=1}^{K} e^{w_j\cdot x+b_j}}\right)x_k\end{aligned} \tag{7.17}$$

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 back-propagation

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.

7.4.3 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. 7.9. 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.

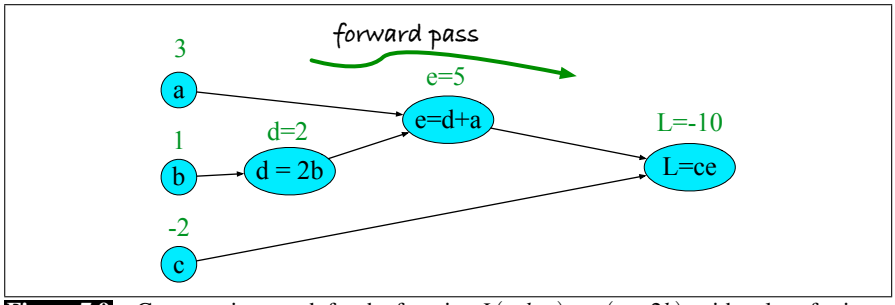

Figure 7.9 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

7.4.4 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. 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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{7.18}$$

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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{7.19}$$

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

$$\frac{\partial L}{\partial c} = e \tag{7.20}$$

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

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{7.21}$$

Eq. 7.21 thus requires 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, multiplying the necessary partials to result in the final derivative we need. Thus we begin by annotating the final node with $\frac{\partial L}{\partial L} = 1$. Moving to the left, we then compute $\frac{\partial L}{\partial c}$ and $\frac{\partial L}{\partial e}$, and so on, until we have annotated the graph all the way to 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. 7.10 shows the backward pass. At each node we need to compute the local partial derivative with respect to the parent, multiply it by the partial derivative that is being passed down from the parent, and then pass it to the child.

Backward differentiation for a neural network

Of course computation graphs for real neural networks are much more complex. Fig. 7.11 shows a sample computation graph for a 2-layer neural network with $n_0 =$

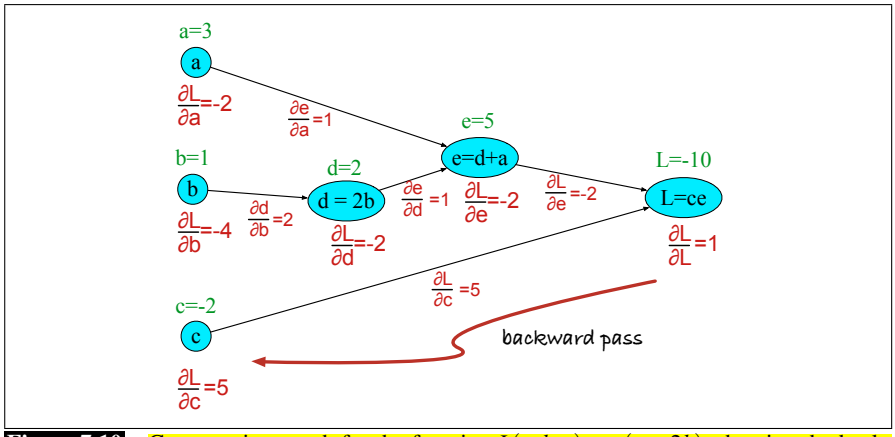

Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

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:

$$\begin{aligned} z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\ a^{[1]} &= \text{ReLU}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{7.22}$$

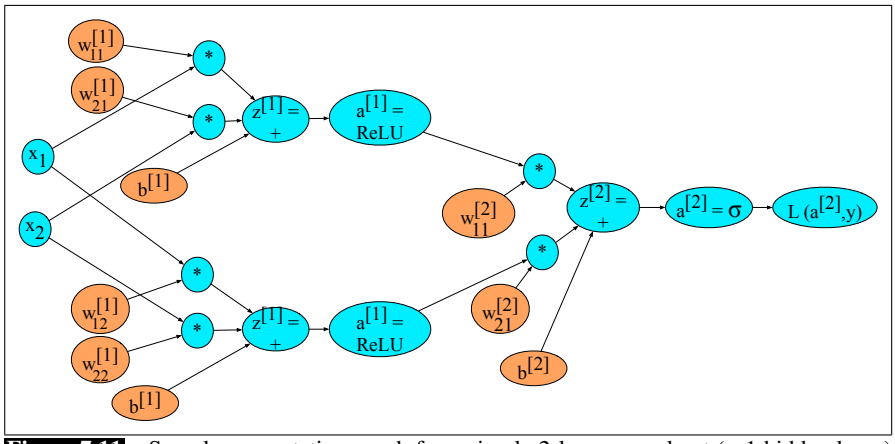

Figure 7.11 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.

The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in orange. 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 **??** the derivative of the sigmoid σ:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{7.23}$$

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

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z) \tag{7.24}$$

The derivative of the ReLU is

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad x < 0 \\ 1 & for \quad x \geq 0 \end{cases} \tag{7.25}$$

7.4.5 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 onto 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.

7.5 Neural Language Models

As our first application of neural networks, let's consider **language modeling**: predicting upcoming words from prior word context.

Neural net-based language models turn out to have many advantages over the n-gram language models of Chapter 3. Among these are that neural language models don't need smoothing, they can handle much longer histories, and they can generalize over contexts of similar words. For a training set of a given size, a neural language model has much higher predictive accuracy than an n-gram language model. Furthermore, neural language models underlie many of the models we'll introduce for tasks like machine translation, dialog, and language generation.

On the other hand, there is a cost for this improved performance: neural net language models are strikingly slower to train than traditional language models, and so for many tasks an n-gram language model is still the right tool.

In this chapter we'll describe simple feedforward neural language models, first introduced by Bengio et al. (2003). Modern neural language models are generally not feedforward but recurrent, using the technology that we will introduce in Chapter 9.

A feedforward neural LM is a standard 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 previous words:

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1}) \tag{7.26}$$

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

7.5.1 Embeddings

In neural language models, the prior context is represented by embeddings of the previous words. Representing the prior context as embeddings, rather than by exact words as used in n-gram language models, allows neural language models to generalize to unseen data much better than n-gram language models. For example, suppose we've seen this sentence in training:

I have to make sure when I get home to feed the cat.

but we've never seen the word "dog" after the words "feed the". In our test set we are trying to predict what comes after the prefix "I forgot when I got home to feed the".

An n-gram language model will predict "cat", but not "dog". But a neural LM, which can make use of the fact that "cat" and "dog" have similar embeddings, will be able to assign a reasonably high probability to "dog" as well as "cat", merely because they have similar vectors.

Let's see how this works in practice. Let's assume we have an embedding dictionary E that gives us, for each word in our vocabulary V, the embedding for that word, perhaps precomputed by an algorithm like word2vec from Chapter 6.

Fig. 7.12 shows a sketch of this simplified feedforward neural language model with N=3; we have a moving window at time t with an embedding vector representing each of the 3 previous words (words w_{t-1}, w_{t-2}, and w_{t-3}). These 3 vectors are concatenated together to produce x, the input layer of a neural network whose output is a softmax with a probability distribution over words. Thus 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.

The model shown in Fig. 7.12 is quite sufficient, assuming we learn the embeddings separately by a method like the **word2vec** methods of Chapter 6. The method of using another algorithm to learn the embedding representations we use for input words is called **pretraining**. If those pretrained embeddings are sufficient for your purposes, then this is all you need.

pretraining

However, often we'd like to learn the embeddings simultaneously with training the network. This is true when whatever task the network is designed for (sentiment

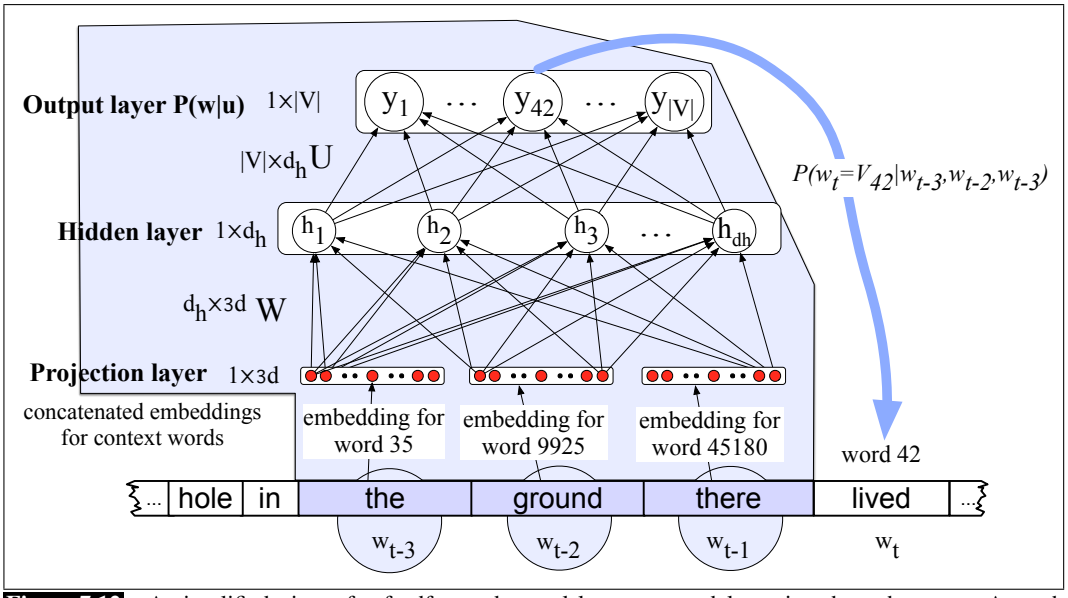

Figure 7.12 A simplified view of a feedforward neural language model moving through a text. At each timestep t the network takes the 3 context words, converts each to a d-dimensional embedding, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer x for the network. These units are multiplied by a weight matrix W and bias vector b and then an activation function to produce a hidden layer h, which is then multiplied by another weight matrix U. (For graphic simplicity we don't show b in this and future pictures.) Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i. (This picture is simplified because it assumes we just look up in an embedding dictionary E the d-dimensional embedding vector for each word, precomputed by an algorithm like word2vec.)

classification, or translation, or parsing) places strong constraints on what makes a good representation.

Let's therefore show an architecture that allows the embeddings to be learned. To do this, we'll add an extra layer to the network, and propagate the error all the way back to the embedding vectors, starting with embeddings with random values and slowly moving toward sensible representations.

For this to work at the input layer, instead of pre-trained embeddings, we're going to 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.

one-hot vector

Thus in a one-hot representation for the word "toothpaste", supposing it happens to have index 5 in the vocabulary, x_5 is one and 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|
```

Fig. 7.13 shows the additional layers needed to learn the embeddings during LM training. Here the N=3 context words are represented as 3 one-hot vectors, fully connected to the embedding layer via 3 instantiations of the embedding matrix E. Note that 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. The embedding weight matrix E thus has

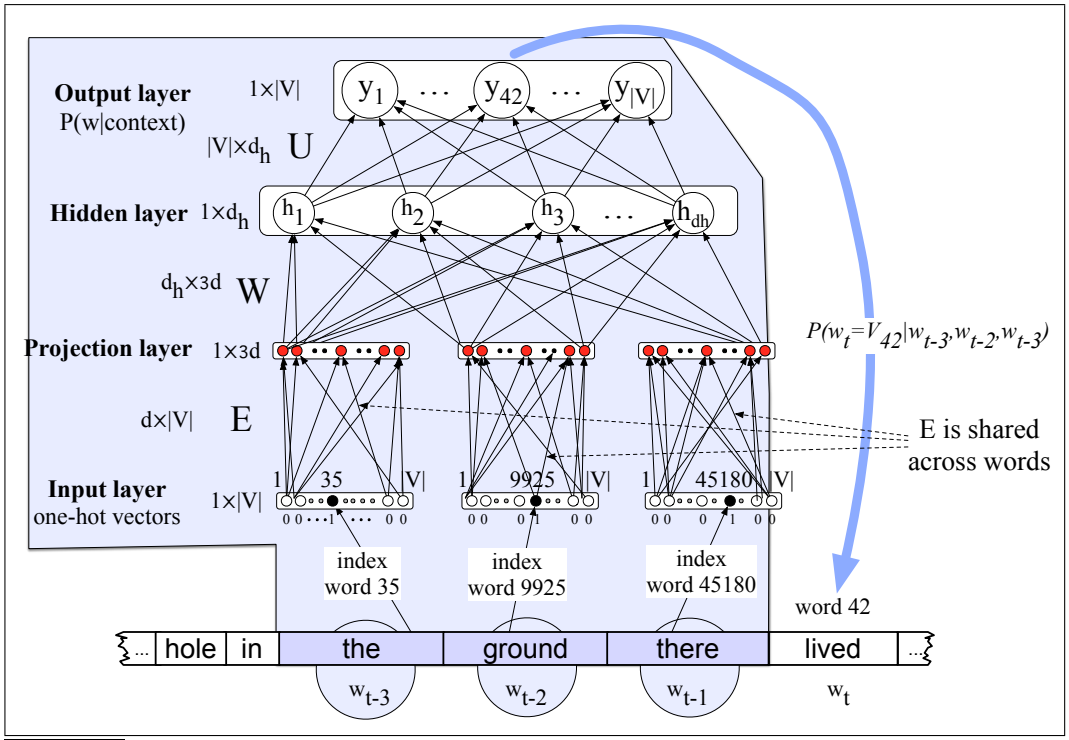

Figure 7.13 Learning all the way back to embeddings. Notice that the embedding matrix E is shared among the 3 context words.

a row for each word, each a vector of d dimensions, and hence has dimensionality $V \times d$.

Let's walk through the forward pass of Fig. 7.13.

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 'the' (index 35) is multiplied by the embedding matrix E, to give the first part of the first hidden layer, called the **projection layer**. Since each row of the input matrix E is just an embedding for a word, and the input is a one-hot column vector x_i for word V_i, the projection layer for input w will be $Ex_i = e_i$, the embedding for word i. We now concatenate the three embeddings for the context words.

projection layer

2. **Multiply by W**: We now multiply by W (and add b) and pass through the rectified linear (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 $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3})$

In summary, if we use e to represent the projection layer, formed by concatenating the 3 embeddings for the three context vectors, the equations for a neural language model become:

$$e = (Ex_1, Ex_2, ..., Ex) \tag{7.27}$$
$$h = \sigma(We + b) \tag{7.28}$$
$$z = Uh \tag{7.29}$$
$$y = \text{softmax}(z) \tag{7.30}$$

7.5.2 Training the neural language model

To train the model, i.e. to set all the parameters $\theta = E, W, U, b$, we do gradient descent (Fig. **??**), using error backpropagation on the computation graph to compute the gradient. Training thus not only sets the weights W and U of the network, but also as we're predicting upcoming words, we're learning the embeddings E for each words that best predict upcoming words.

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, the cross-entropy (negative log likelihood) loss is:

$$L = -\log p(w_t|w_{t-1}, ..., w_{t-n+1}) \tag{7.31}$$

The gradient for this loss is then:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial -\log p(w_t|w_{t-1}, ..., w_{t-n+1})}{\partial \theta} \tag{7.32}$$

This gradient can be computed in any standard neural network framework which will then backpropagate through U, W, b, E.

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 E that can be used as word representations for other tasks.

7.6 Summary

- Neural networks are built out of **neural units**, originally inspired by human 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.
- 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 human neuron as a kind of com-

puting 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, LeCun et al. 1990) and speech recognition (Morgan and Bourlard 1989, 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.

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

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.. Software available from tensorflow.org.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, *3*(Feb), 1137–1155.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *NIPS 2007*, 153–160.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179–211.

Feldman, J. A. and Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science*, *6*, 205–254.

Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing*, Vol. 10 of *Synthesis Lectures on Human Language Technologies*. Morgan & Claypool.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *COGSCI-86*, 1–12.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, *18*(7), 1527–1554.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580.

Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR 2015*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *NIPS 1990*, 396–404.

McClelland, J. L. and Elman, J. L. (1986). The TRACE model of speech perception. *Cognitive Psychology*, *18*, 1–86.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133. Reprinted in *Neurocomputing: Foundations of Research, ed. by J. A. Anderson and E Rosenfeld. MIT Press 1988.*

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press.

Morgan, N. and Bourlard, H. (1989). Generalization and parameter estimation in feedforward nets: Some experiments. In *Advances in neural information processing systems*, 630–637.

Morgan, N. and Bourlard, H. (1990). Continuous speech recognition using multilayer perceptrons with hidden markov models. In *ICASSP-90*, 413–416.

Nielsen, M. A. (2015). *Neural networks and Deep learning*. Determination Press USA.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. In *NIPS-W*.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain.. *Psychological review*, *65*(6), 386–408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L. (Eds.), *Parallel Distributed Processing*, Vol. 2, 318–362. MIT Press.

Rumelhart, D. E. and McClelland, J. L. (1986a). On learning the past tense of English verbs. In Rumelhart, D. E. and McClelland, J. L. (Eds.), *Parallel Distributed Processing*, Vol. 2, 216–271. MIT Press.

Rumelhart, D. E. and McClelland, J. L. (Eds.). (1986b). *Parallel Distributed Processing*. MIT Press.

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach* (2nd Ed.). Prentice Hall.

Smolensky, P. (1988). On the proper treatment of connectionism. *Behavioral and brain sciences*, *11*(1), 1–23.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, *46*(1-2), 159–216.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2014). Dropout: a simple way to prevent neural networks from overfitting.. *JMLR*, *15*(1), 1929–1958.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, Vol. 4, 96–104.