CHAPTER

8 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 an essential 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 consider a neural net classifier, built by combining units into a network. As we'll see, this is called a feed-forward 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 hidden layers).

deep learning
deep

Neural networks share some of the same mathematics and learning architectures as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a neural network with one 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 simple and fixed regression classifier to many different asks by developing many rich kinds of feature templates based on domain knowledge. When working with neural networks, it is more common to avoid the use 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. This is especially true with nets that are very deep (have many hidden layers), and for that reason deep neural nets, more than other classifiers, tend to be applied on large scale problems that offer sufficient data to learn features automatically.

In this chapter we'll see feedforward networks as classifiers, and apply them to the simple task of language modeling: assigning probabilities to word sequences and predicting upcoming words.

In later chapters we'll introduce many other aspects of neural models. Chapter 9b will introduce **recurrent neural networks**. Chapter 15 will introduce the use of neural networks to compute the semantic representations for words called **embeddings**. And Chapter 25 and succeeding chapters will introduce the sequence-to-sequence or **seq2seq** model (also called the “encoder-decoder model”) for applications involving language generation: machine translation, conversational agents, and summarization.

8.1 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**. Thus 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:

$$z = b + \sum_i w_i x_i \tag{8.1}$$

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**:

$$z = w \cdot x + b \tag{8.2}$$

As defined in Eq. 8.2, 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:

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

(8.3)

sigmoid

We’ll discuss three popular non-linear functions $f()$ below (the sigmoid, the tanh, and the rectified linear ReLU) but it’s convenient to start with the **sigmoid** function:

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{8.4}$$

The sigmoid 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’ll see in Section 8.4 will be handy for learning. Fig. 8.1 shows a graph.

Substituting the sigmoid equation into Eq. 8.2 gives us the final value for the output of a neural unit:

$$y = \sigma(w \cdot x + b) = \frac{1}{1+exp(-(w \cdot x + b))} \tag{8.5}$$

Fig. 8.2 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

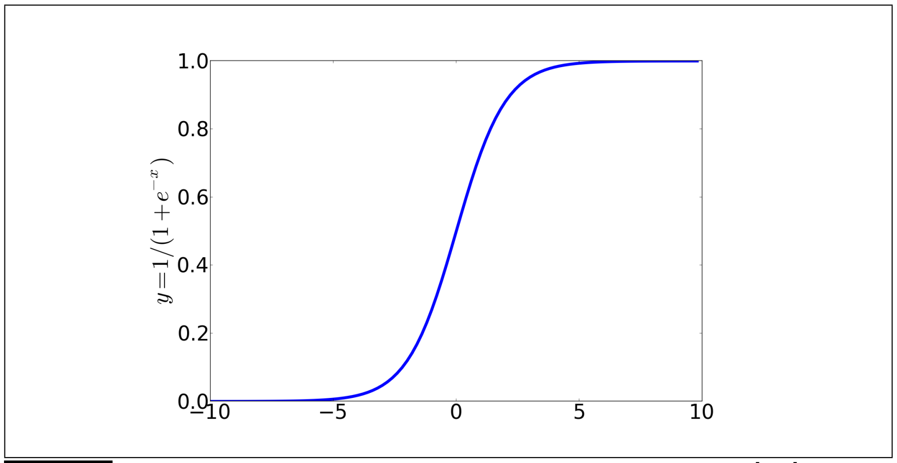

Figure 8.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. Because it is nearly linear around 0 but has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1.

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.

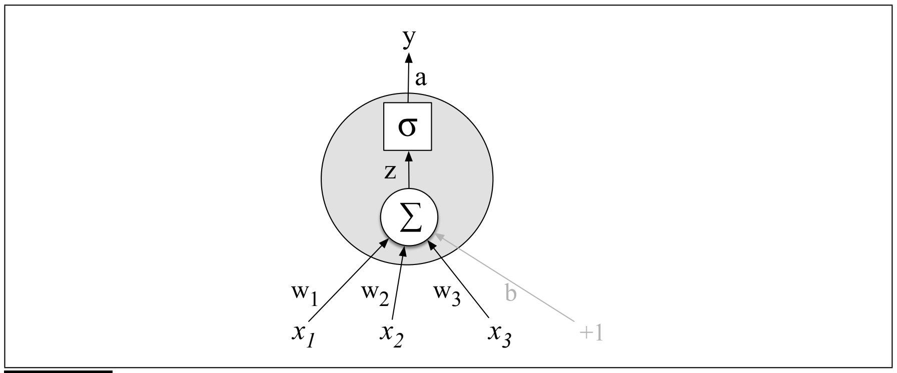

Figure 8.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of 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 vectors 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:

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{1}{1 + e^{-(.5*.2+.3*.6+.8*.1+.5)}} = e^{-0.86} = .42$$

Other nonlinear functions besides the sigmoid are also commonly used. The **tanh** function shown in Fig. 8.3a is a variant of the sigmoid that ranges from -1 to +1:

tanh

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (8.6)$$

ReLU

The simplest activation function is the rectified linear unit, also called the **ReLU**, shown in Fig. 8.3b. It's just the same as x when x is positive, and 0 otherwise:

$$y = max(x, 0) \quad (8.7)$$

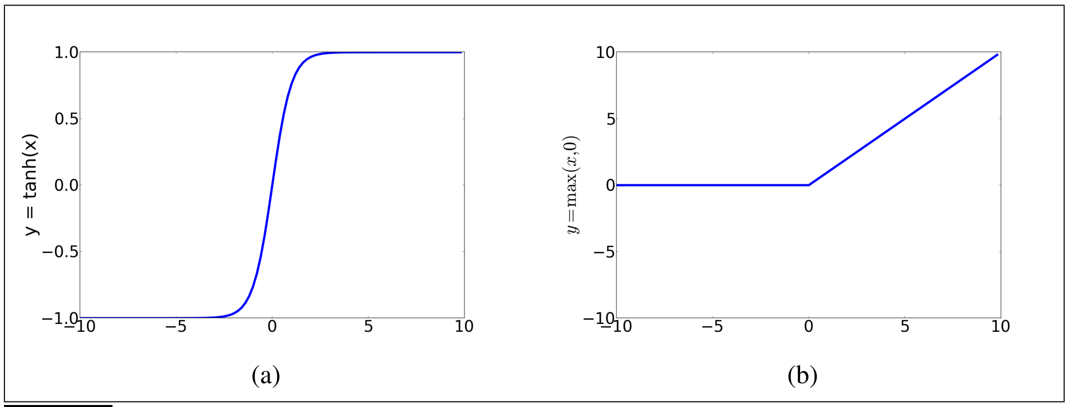

Figure 8.3 The tanh and ReLU activation functions.

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.

saturated

8.2 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. In the next section we take a look at that intuition.

Consider the very simple task of computing simple 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 no non-linear activation function. The output y of a perceptron is 0 or 1, and just computed as follows (using the same weight w, input x, and bias b as in Eq. 8.2):

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases} \tag{8.8}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs; Fig. 8.4 shows the necessary weights.

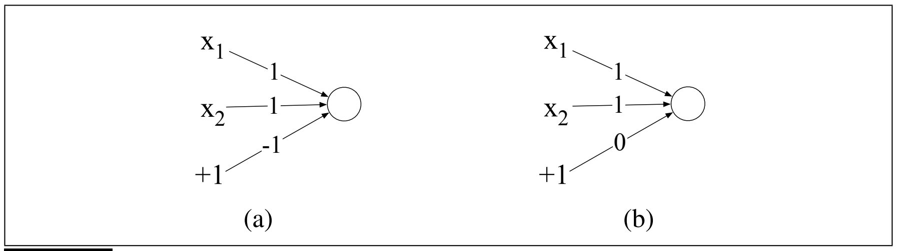

Figure 8.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical 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 x0 and x1, the perception equation, $w1x1 + w2x2 + b = 0$ is the equation of a line (we can see this by putting it in the standard linear format: $x2 = -(w1/w2)x1 - b$.) 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. 8.5 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

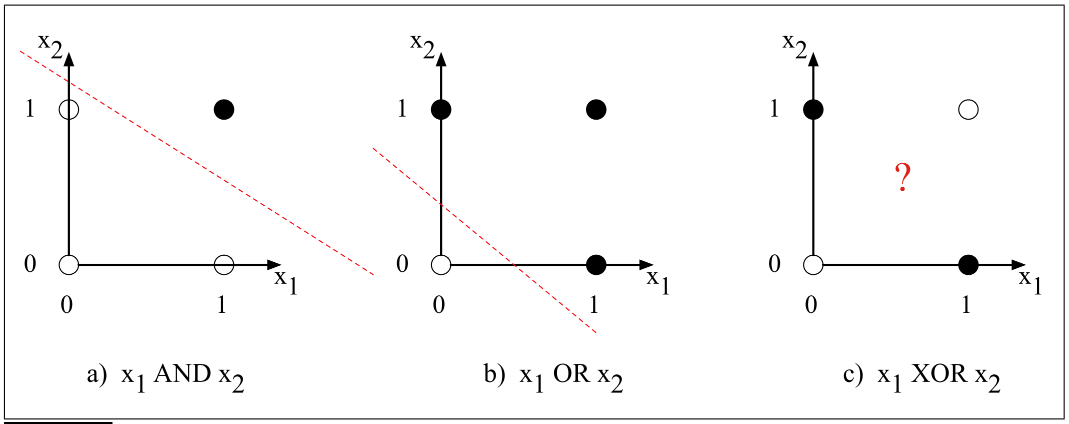

Figure 8.5 The functions AND, OR, and XOR, represented with input x_0 on the x-axis and input x_1 on the y axis, Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

8.2.1 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. 8.6 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

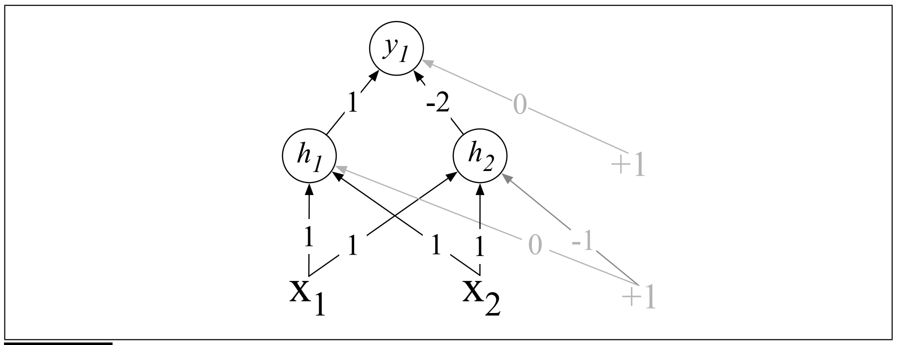

Figure 8.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

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 we 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 correctly are 1 for the inputs [0 1] and [1 0] and 0 for [0 0] and [1 1].

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. 8.7b 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 is forming a representation for the input.

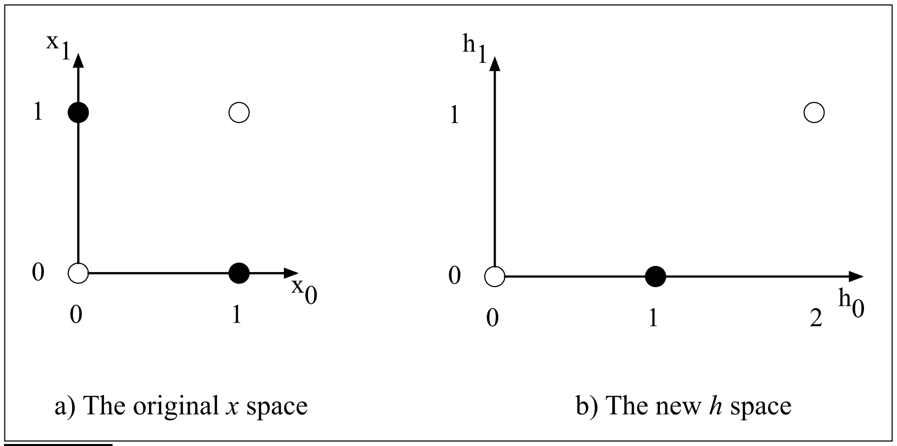

Figure 8.7 The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, h, compared to the original input representation x. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

In this example we just stipulated the weights in Fig. 8.6. But for real examples the weights for neural networks are learned automatically using the error backpropagation algorithm to be introduced in Section 8.4. 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 to) a single layer of linear units with appropriate weights, and we've already shown (visually, in Fig. 8.5) that a single unit cannot solve the XOR problem.

8.3 Feed-Forward Neural Networks

feed-forward network

Let's now walk through a slightly more formal presentation of the simplest kind of neural network, the **feed-forward network**. A feed-forward 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. (Later 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 feed-forward networks have three kinds of nodes: input units, hidden units, and output units. Fig. 8.8 shows a picture.

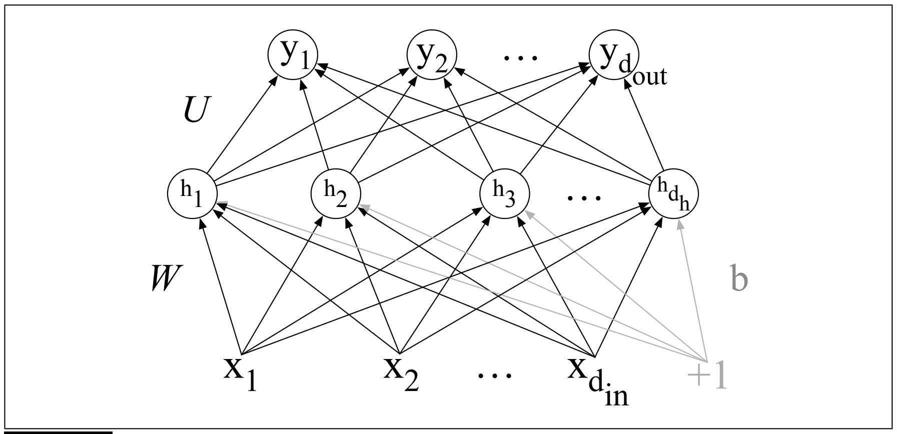

Figure 8.8 Caption here

The input units are simply scalar values just as we saw in Fig. 8.2.

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 8.1, 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 parameters w (the weight vector) and b (the bias scalar). We represent the parameters for the entire hidden layer by combining the weight w_i and bias b_i for each unit i into a single weight matrix W and a single bias vector b for the whole layer (see Fig. 8.8). Each element W_{ij} of the weight matrix W represents the weight of the connection from the ith input unit x_i to the the jth hidden unit h_j.

The advantage of using a single matrix W for the weights of the entire layer is that now that 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 f (such as the sigmoid, tanh, or rectified linear activation function defined above).

The output of the hidden layer, the vector h, is thus the following, assuming the sigmoid function σ:

$$h = \sigma(Wx+b) \tag{8.9}$$

Notice that we're apply the σ function here to a vector, while in Eq. 8.4 it was applied to a scalar. We're thus allowing $\sigma(\cdot)$, and indeed any activation function $f(\cdot)$, to apply to a vector element-wise, so $f[z_1,z_2,z_3] = [f(z_1), f(z_2), f(z_3)]$.

Let's introduce some constants to represent the dimensionalities of these vectors and matrices. We'll have d_{in} represent the number of inputs, so x is a vector of real numbers of dimensionality d_{in}, or more formally $x \in \mathbb{R}^{d_{in}}$. The hidden layer has dimensional d_h, so $h \in \mathbb{R}^{d_h}$ and also $b \in \mathbb{R}^{d_h}$ (since each hidden unit can take a different bias value). And the weight matrix W has dimensionality $W \in \mathbb{R}^{d_h \times d_{in}}$.

Take a moment to convince yourself that the matrix multiplication in Eq. 8.9 will compute the value of each h_{ij} as $\sum_{i=1}^{d_{in}} w_{ij}x_i + b_j$.

As we saw in Section 8.2, 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 it often doesn't have a bias vector b, so we'll eliminate it in our examples here. The weight matrix is multiplied by the input vector (h) to produce the intermediate output z.

$$z = Uh$$

There are d_{out} output nodes, so $z \in \mathbb{R}^{d_{out}}$, weight matrix U has dimensionality $U \in \mathbb{R}^{d_{out} \times d_h}$, 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

normalizing

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

softmax

between 0 and 1 and sum to 1): the **softmax** function.

For a vector z of dimensionality D, the softmax is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le D \tag{8.10}$$

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 7; the equation for computing the probability of y being of class c given x in multinomial logistic regression was (repeated from Eq. 8.11):

$$p(c|x) = \frac{\exp\left(\sum_{i=1}^{N} w_i f_i(c,x)\right)}{\sum_{c' \in C} \exp\left(\sum_{i=1}^{N} w_i f_i(c',x)\right)} \tag{8.11}$$

In other words, 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 7 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 feed-forward 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:

$$h = \sigma(Wx+b) \tag{8.12}$$
$$z = Uh \tag{8.13}$$
$$y = \text{softmax}(z) \tag{8.14}$$
$$\tag{8.15}$$

8.4 Training Neural Nets

To train a neural net, meaning to set the weights and biases W and b for each layer, we use optimization methods like **stochastic gradient descent**, just as with logistic regression in Chapter 7.

Let's use the variable θ to mean all the parameters we need to learn (W and b for each layer). The intuition of gradient descent is to start with some initial guess at θ, for example setting all the weights randomly, and then nudge the weights (i.e. change θ slightly) in a direction that improves our system.

8.4.1 Loss function

If our goal is to move our weights in a way that improves the system, we'll obviously need a metric for whether the system has improved or not.

The neural nets we have been describing are supervised classifiers, which means we know the right answer for each observation in the training set. So our goal is for the output from the network for each training instance to be as close as possible to the correct gold label.

Rather than measure how close the system output for each training instance is to the gold label for that instance we generally instead measure the opposite. We measure the *distance* between the system output and the gold output, and we call this distance the **loss** or the **cost function**.

loss

cost function

So our goal is to define a loss function, and then find a way to minimize this loss.

Imagine a very simple regressor with one output node that computes a real value of some single input node x. The true value is y, and our network estimates a value $\hat{y}$ which it computes via some function $f(x)$. We can express this as:

$$L(\hat{y},y) \;=\; \text{How much } \hat{y} \text{ differs from the true } y \tag{8.16}$$

or equivalently, but with more details, making transparent the fact that $\hat{y}$ is computed by a function f that is parameterized by θ:

$$L(f(x;\theta),y) = \text{How much } f(x) \text{ differs from the true } y \tag{8.17}$$

mean-squared error
MSE

A common loss function for such a network (or for the similar case of *linear regression*) is the **mean-squared error** or **MSE** between the true value $y^{(i)}$ and the system's output $\hat{y}^{(i)}$, the average over the m observations of the square of the error in $\hat{y}$ for each one:

$$L_{\text{MSE}}(\hat{y},y) = \frac{1}{n}\sum_{i=1}^{m}(\hat{y}^{(m)} - y^{(i)})^2 \tag{8.18}$$

cross entropy loss

While mean squared error makes sense for regression tasks, mostly in this chapter we have been considering nets as probabilistic classifiers. For probabilistic classifiers a common loss function—also used in training logistic regression—is the **cross entropy loss**, also called the **negative log likelihood**. Let y be a vector over the C classes representing the true output probability distribution. Assume this is a **hard classification** task, meaning that only one class is the correct one. If the true class is i, then y is a vector where $y_i = 1$ and $y_j = 0 \quad \forall j \neq i$. 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 loss is simply the log probability of the correct class:

$$L(\hat{y},y) = -\log p(\hat{y}_i) \tag{8.19}$$

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

Given a loss function[1] our goal in training is to move the parameters so as to minimize the loss, finding the minimum of the loss function.

8.4.2 Following Gradients

How shall we find the minimum of this loss function? Gradient descent is a method that finds a minimum of a function by figuring out in which direction (in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction.

The intuition is that if you are hiking the Grand Canyon and trying to descend most quickly down to the river you might look around yourself 360 degrees, find the direction where the ground is sloping the steepest, and walk downhill in that direction.

Although the algorithm (and the concept of gradient) are designed for direction *vectors*, let's first consider a visualization of the the case where, θ, the parameter of our system, is just a single scalar, shown in Fig. 8.9.

Given a random initialization of θ at some value θ_1, and assuming the loss function L happened to have the shape in Fig. 8.9, we need the algorithm to tell us

[1] See any machine learning textbook for lots of other potential functions like the useful **hinge loss**.

whether at the next iteration, we should move left (making θ_2 smaller than $theta_1$) or right (making θ_2 bigger than θ_1) to reach the minimum.

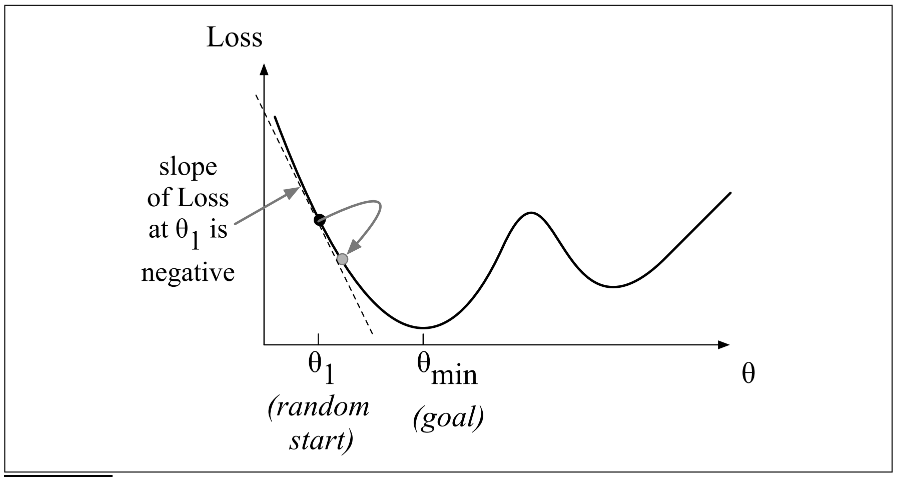

Figure 8.9 The first step in iteratively finding the minimum of this loss function, by moving θ in the reverse direction from the slope of the function. Since the slope is negative, we need to move θ in a positive direction, to the right.

gradient The gradient descent algorithm answers this question by finding the **gradient** of the loss function at the current point and moving in the opposite direction. The gradient of a function of many variables is a vector pointing in the direction of greatest change in a function. The gradient is a multi-variable generalization of the slope, and indeed for a function of one variable like the one in Fig. 8.9, we can informally think of the gradient as the slope. The dotted line in Fig. 8.9 shows the slope of this hypothetical loss function at point $\theta = \theta_0$. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving θ in a positive direction.

learning rate The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{d\theta}f(\theta_0)$ weighted by another variable called the **learning rate** η. A higher (faster) learning rate means that we should move θ more on each step. The change we make in our parameter is the learning rate times the gradient (or the slope, in our single-variable example):

$$\theta_{t+1} = \theta_t - \eta \frac{d}{d\theta} f(\theta_0) \tag{8.20}$$

Now let's extend the intuition from a function of one variable to many variables, because we don't just want to move left or right, we want to know where in the N-dimensional space (of the N parameters that make up θ) we should move. Recall our intuition from standing at the rim of the Grand Canyon. If we are on a mesa and want to know which direction to walk down, we need a vector that tells us in which direction we should move. The **gradient** is just such a vector; it expresses the directional components of the sharpest slope along each of those N dimensions. If we're just imagining the two dimensions of the plane, the gradient might be a vector with two orthogonal components, each of which tells us how much the ground slopes in that direction. Fig. 8.10 shows a visualization:

In an actual network θ, the parameter vector, is much longer than 2; it contains all the weights and biases for the whole network, which can be millions of parameters. For each dimension/variable θ_j that makes up θ, the gradient will have a

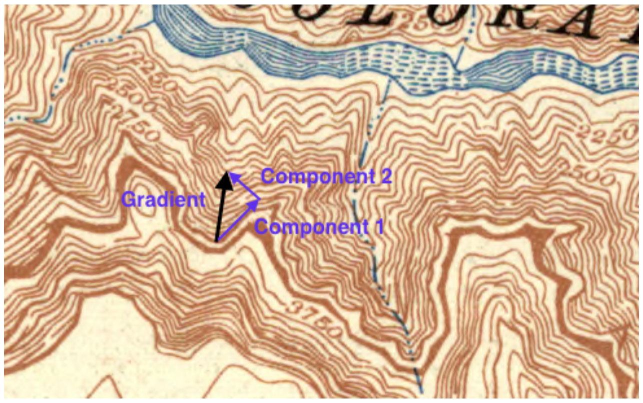

Figure 8.10 Visualization of the gradient vector in two dimensions.

component that tells us the slope with respect to that variable. Essentially we're asking: "How much would a small change in that variable θ_j influence the loss function L?"

In each dimension θ_j, we express the slope as a partial derivative $\frac{\partial}{\partial \theta_j}$ of the loss function. The gradient is then defined as a vector of these partials:

$$\nabla_{\theta} L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial \theta_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial \theta_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial \theta_m} L(f(x;\theta),y) \end{bmatrix} \tag{8.21}$$

The final equation for updating θ based on the gradient is thus

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y) \tag{8.22}$$

8.4.3 Computing the Gradient

Computing the gradient requires the partial derivative of the loss function with respect to each parameter. This can be complex for deep networks, where 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 known as **error backpropagation** or **backprop** (Rumelhart et al., 1986), which turns out to be a special case of backward differentiation. In backprop, the loss function is first modeled as a computation graph, in which each edge is a computation and each node the result of the computation. The chain rule of differentiation is then used to annotate this graph with the partial derivatives of the loss function along each edge of the graph. We give a brief overview of the algorithm in the next subsections; further details can be found in any machine learning or data-intensive computation textbook.

Computation Graphs

TBD

Error Back Propagation

TBD

8.4.4 Stochastic Gradient Descent

Once we have computed the gradient, we can use it to train θ. The stochastic gradient descent algorithm (LeCun et al., 2012) is an online algorithm that computes this gradient after each training example, and nudges θ in the right direction. Fig. 8.11 shows the algorithm.

```
function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, $x$, $y$) returns $\theta$
      # where: L is the loss function
      #       f is a function parameterized by $\theta$
      #       x is the set of training inputs $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$
      #       y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$

$\theta \leftarrow$ small random values
while not done
   Sample a training tuple $(x^{(i)}, y^{(i)})$
   Compute the loss $L(f(x^{(i)};\theta), y^{(i)})$   # How far off is $f(x^{(i)})$ from $y^{(i)}$?
   $g \leftarrow \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$              # How should we move $\theta$ to maximize loss ?
   $\theta \leftarrow \theta - \eta_k\, g$          # go the other way instead
```

Figure 8.11 The stochastic gradient descent algorithm, after (Goldberg, 2017).

Stochastic gradient descent is called stochastic because it chooses a single random example at a time, moving the weights so as to improve performance on that single example. That can result in very choppy movements, so an alternative version of the algorithm, **minibatch** gradient descent, computes the gradient over batches of training instances rather than a single instance.

minibatch

The learning rate η_k is a parameter that must be adjusted. If it's too high, the learner will take steps that are too large, overshooting the minimum of the loss function. If it's too low, the learner will take steps that are too small, and take too long to get to the minimum. It is most common to being the learning rate at a higher value, and then slowly decrease it, so that it is a function of the iteration k of training.

8.5 Neural Language Models

Now that we've introduced neural networks it's time to see an application. The first application we'll consider is language modeling: predicting upcoming words from prior word context.

Although we have already introduced a perfectly useful language modeling paradigm (the smoothed N-grams of Chapter 4), neural net-based language models turn out to have many advantages. 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. Furthermore, neural net language models underlie many of the models we'll introduce for generation, summarization, machine translation, and dialog.

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 traditional language modeling is still the right technology.

In this chapter we'll describe simple feedforward neural language models, first introduced by Bengio et al. (2003). We will turn to the recurrent language model, more commonly used today, in Chapter 9b.

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 traditional 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:

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1}) \tag{8.23}$$

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

8.5.1 Embeddings

The insight of neural language models is in how to represent the prior context. Each word is represented as a vector of real numbers of of dimension d; d tends to lie between 50 and 500, depending on the system. These vectors for each words are called **embeddings**, because we represent a word as being embedded in a vector space. By contrast, in many traditional NLP applications, a word is represented as a string of letters, or an index in a vocabulary list.

embeddings

Why represent a word as a vector of 50 numbers? Vectors turn out to be a really powerful representation for words, because a distributed representation allows words that have similar meanings, or similar grammatical properties, to have similar vectors. As we'll see in Chapter 15, embedding that are learned for words like "cat" and "dog"— words with similar meaning and parts of speech—will be similar vectors. That will allow us to generalize our language models in ways that wasn't possible with traditional N-gram models.

For example, suppose we've seen this sentence in training:

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

and then in our test set we are trying to predict what comes after the prefix "I forgot when I got home to feed the".

A traditional N-gram model will predict "cat". But suppose we've never seen the word "dog" after the words "feed the". A traditional LM won't expect "dog". But by representing words as vectors, and assuming the vector for "cat" is similar to the vector for "dog", a neural LM, even if it's never seen "feed the dog", will assign a reasonably high probability to "dog" as well as "cat", merely because they have similar vectors.

Representing words as embeddings vectors is central to modern natural language processing, and is generally referred to as the **vector space model** of meaning. We will go into lots of details on the different kinds of embeddings in Chapter 15 and Chapter 152.

Let's set aside—just for a few pages—the question of how these embeddings are learned. Imagine that we had an embedding dictionary E that gives us, for each word in our vocabulary V, the vector for that word.

Fig. 8.12 shows a sketch of this simplified FFNNLM with N=3; we have a moving window at time t with a one-hot 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.

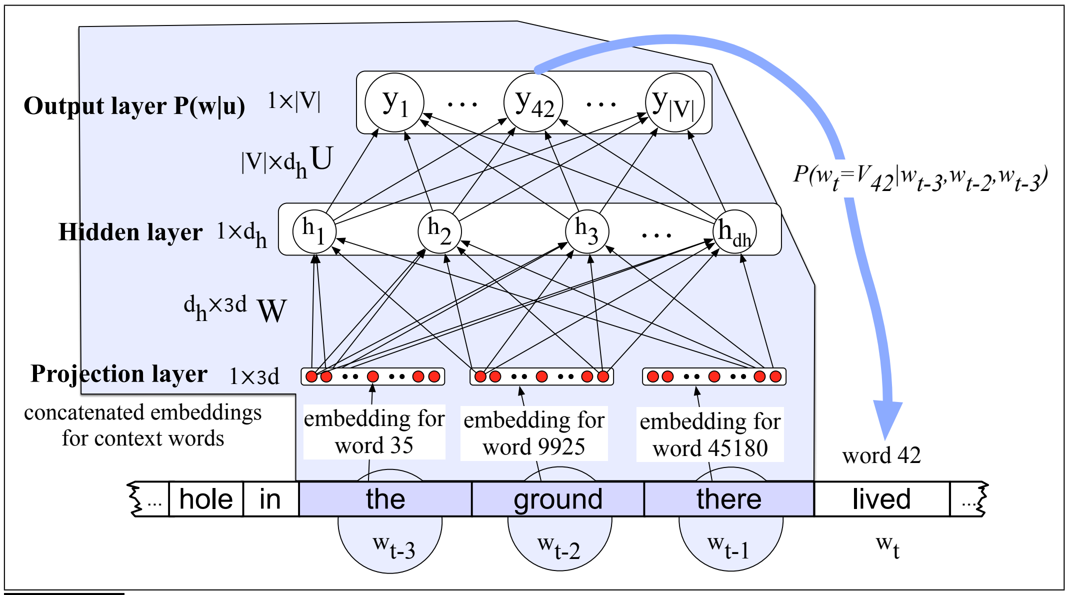

Figure 8.12 A simplified view of a feedforward neural language model moving through a text. At each timestep t the network takes the 3 context words, converts each to a d-dimensional embeddings, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer x for the network. These 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 a dictionary table E the "embedding vector", a d-dimensional vector representing each word, but doesn't yet show us how these embeddings are learned.)

The model shown in Fig. 8.12 is quite sufficient, assuming we learn the embeddings separately by a method like the **word2vec** methods of Chapter 152. 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 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 $x_i = 0\ \forall i \neq 5$, as shown here:

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

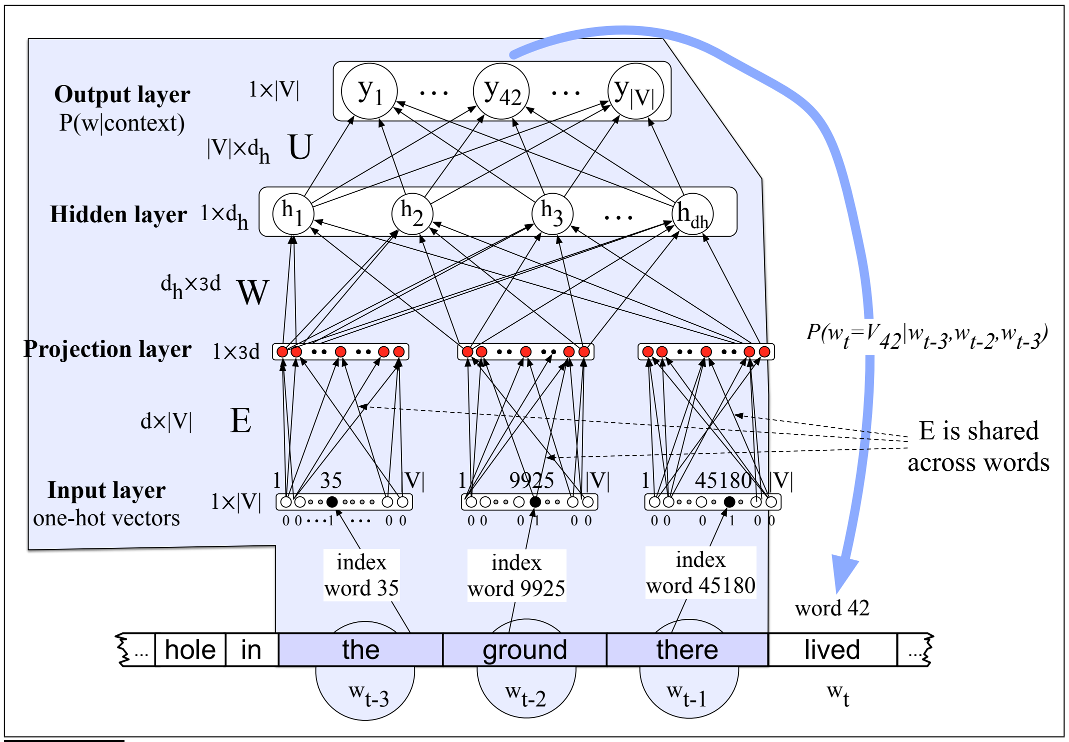

Figure 8.13 learning all the way back to embeddings. notice that the embedding matrix E is shared among the 3 context words.

Fig. 8.13 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 E embedding matrix. 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 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. 8.13.

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' is (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 columnvector 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 embedding for the three context vectors, the equations for a neural language model become:

$$e = (Ex_1, Ex_2, ..., Ex) \tag{8.24}$$
$$h = \sigma(We + b) \tag{8.25}$$
$$z = Uh \tag{8.26}$$
$$y = \text{softmax}(z) \tag{8.27}$$

8.5.2 Training the neural language model

To train the model, i.e. to set all the parameters $\theta = E, W, U, b$, we use the SGD algorithm of Fig. 8.11, with error back propagation 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 proceedings by taking as input a very long text, concatenating all the sentences, start with random weights, and then iteratively moving through the text predicting each word w_t. At each word w_t, the categorial cross-entropy (negative log likelihood) loss is:

$$L = -\log p(w_t|w_{t-1}, ..., w_{t-n+1}) \tag{8.28}$$

The gradient is computed for this loss by differentiation:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \log p(w_t|w_{t-1}, ..., w_{t-n+1})}{\partial \theta} \tag{8.29}$$

And then backpropagated through U, W, b, E.

8.6 Summary

- Neural networks are built out of **neural units**, originally inspired by human neurons but now simple 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 **stochastic gradient descent**.

- **Error back propagation** is used to compute the gradients of the loss function for a network.
- **Neural language modeling** uses a network as a probabilistic classifier, to compute the probability of the next word given the previous N word.
- Neural language models make use of **embeddings**, dense vectors of between 50 and 500 dimensions that represent words in the input vocabulary.

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 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 back propagation 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 idea of distributed representations (Hinton, 1986), of 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 9b.

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).

The description in this chapter has been quite high-level, and there are many details of neural network training and architecture that are necessary to successfully train models. For example various forms of regularization are used to prevent overfitting, including **dropout**: randomly dropping some units and their conntections from the network during training (Hinton et al. 2012, Srivastava et al. 2014). Faster optimization methods than vanilla stochastic gradient descent are often used, such as Adam (Kingma and Ba, 2015).

dropout

Since neural networks training and decoding require significant numbers of vector operations, modern systems are often trained using vector-based GPUs (Graphic Processing Units). A number of software engineering tools are widely available including TensorFlow (Abadi et al., 2015) and others.

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*, pp. 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*, pp. 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*, pp. 396–404.

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–48. Springer.

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*, pp. 630–637.

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

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

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, pp. 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, pp. 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. (2014). Dropout: a simple way to prevent neural networks from overfitting.. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

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