CHAPTER 4

Feed-forward Neural Networks

4.1 A BRAIN-INSPIRED METAPHOR

As the name suggests, neural networks were inspired by the brain's computation mechanism, which consists of computation units called neurons. While the connections between artificial neural networks and the brain are in fact rather slim, we repeat the metaphor here for completeness. In the metaphor, a neuron is a computational unit that has scalar inputs and outputs. Each input has an associated weight. The neuron multiplies each input by its weight, and then sums[1] them, applies a nonlinear function to the result, and passes it to its output. Figure 4.1 shows such a neuron.

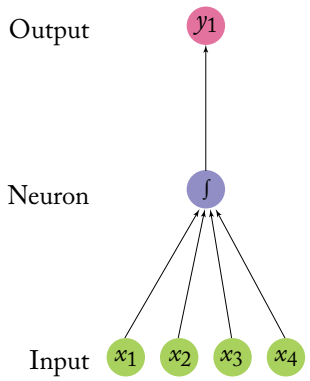

Figure 4.1: A single neuron with four inputs.

The neurons are connected to each other, forming a network: the output of a neuron may feed into the inputs of one or more neurons. Such networks were shown to be very capable computational devices. If the weights are set correctly, a neural network with enough neurons and a nonlinear activation function can approximate a very wide range of mathematical functions (we will be more precise about this later).

A typical feed-forward neural network may be drawn as in Figure 4.2. Each circle is a neuron, with incoming arrows being the neuron's inputs and outgoing arrows being the neuron's outputs. Each arrow carries a weight, reflecting its importance (not shown). Neurons are arranged in layers, reflecting the flow of information. The bottom layer has no incoming arrows, and is

[1]While summing is the most common operation, other functions, such as a max, are also possible.

the input to the network. The top-most layer has no outgoing arrows, and is the output of the network. The other layers are considered "hidden." The sigmoid shape inside the neurons in the middle layers represent a nonlinear function (i.e., the logistic function $1/(1+e^{-x})$) that is applied to the neuron's value before passing it to the output. In the figure, each neuron is connected to all of the neurons in the next layer—this is called a *fully connected layer* or an *affine layer*.

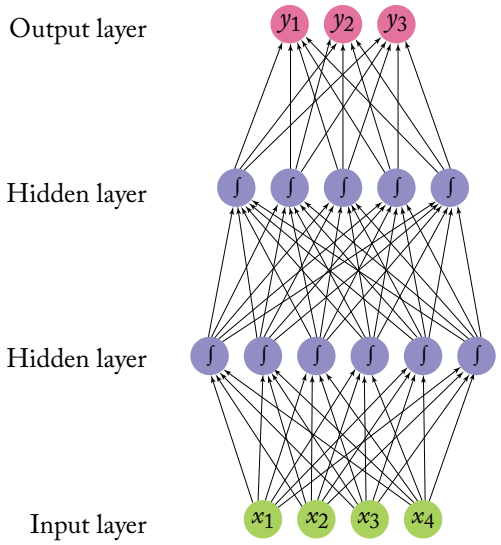

Figure 4.2: Feed-forward neural network with two hidden layers.

While the brain metaphor is sexy and intriguing, it is also distracting and cumbersome to manipulate mathematically. We therefore switch back to using more concise mathematical notation. As will soon become apparent, a feed-forward network as the one in Figure 4.2 is simply a stack of linear models separated by nonlinear functions.

The values of each row of neurons in the network can be thought of as a vector. In Figure 4.2 the input layer is a 4-dimensional vector ($\boldsymbol{x}$), and the layer above it is a 6-dimensional vector ($\boldsymbol{h^1}$). The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions. A fully connected layer implements a vector-matrix multiplication, $\boldsymbol{h} = \boldsymbol{x}\boldsymbol{W}$ where the weight of the connection from the ith neuron in the input row to the jth neuron in the output row is $\boldsymbol{W}_{[i,j]}$.[2] The values of $\boldsymbol{h}$ are then transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer. The whole computation from input to output can be written as: $(g(\boldsymbol{x}\boldsymbol{W^1}))\boldsymbol{W^2}$ where $\boldsymbol{W^1}$ are the weights of the first layer and $\boldsymbol{W^2}$ are the weights of the second one. Taking this view, the single neuron in Figure 4.1 is equivalent to a logistic (log-linear) binary classifier $\sigma(\boldsymbol{x}\boldsymbol{w})$ without a bias term .

[2]To see why this is the case, denote the weight of the ith input of the jth neuron in $\boldsymbol{h}$ as $\boldsymbol{W}_{[i,j]}$. The value of $\boldsymbol{h}_{[j]}$ is then $\boldsymbol{h}_{[j]} = \sum_{i=1}^{4} \boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,j]}$.

4.2 IN MATHEMATICAL NOTATION

From this point on, we will abandon the brain metaphor and describe networks exclusively in terms of vector-matrix operations.

The simplest neural network is called *a perceptron*. It is simply a linear model:

$$\text{NN}_{\text{Perceptron}}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b} \tag{4.1}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}},$$

where $\boldsymbol{W}$ is the weight matrix and $\boldsymbol{b}$ is a bias term.[3] In order to go beyond linear functions, we introduce a nonlinear hidden layer (the network in Figure 4.2 has two such layers), resulting in the Multi Layer Perceptron with one hidden-layer (MLP1). A feed-forward neural network with one hidden-layer has the form:

$$\text{NN}_{\text{MLP1}}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{1}} + \boldsymbol{b}^{\boldsymbol{1}})\boldsymbol{W}^{\boldsymbol{2}} + \boldsymbol{b}^{\boldsymbol{2}} \tag{4.2}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W}^{\boldsymbol{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \boldsymbol{b}^{\boldsymbol{1}} \in \mathbb{R}^{d_1}, \quad \boldsymbol{W}^{\boldsymbol{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \boldsymbol{b}^{\boldsymbol{2}} \in \mathbb{R}^{d_2}.$$

Here $\boldsymbol{W}^{\boldsymbol{1}}$ and $\boldsymbol{b}^{\boldsymbol{1}}$ are a matrix and a bias term for the first linear transformation of the input, g is a nonlinear function that is applied element-wise (also called a *nonlinearity* or an *activation function*), and $\boldsymbol{W}^{\boldsymbol{2}}$ and $\boldsymbol{b}^{\boldsymbol{2}}$ are the matrix and bias term for a second linear transform.

Breaking it down, $\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{1}} + \boldsymbol{b}^{\boldsymbol{1}}$ is a linear transformation of the input $\boldsymbol{x}$ from d_{in} dimensions to d_1 dimensions. g is then applied to each of the d_1 dimensions, and the matrix $\boldsymbol{W}^{\boldsymbol{2}}$ together with bias vector $\boldsymbol{b}^{2}$ are then used to transform the result into the d_2 dimensional output vector. The nonlinear activation function g has a crucial role in the network's ability to represent complex functions. Without the nonlinearity in g, the neural network can only represent linear transformations of the input.[4] Taking the view in Chapter 3, the first layer transforms the data into a good representation, while the second layer applies a linear classifier to that representation.

We can add additional linear-transformations and nonlinearities, resulting in an MLP with two hidden-layers (the network in Figure 4.2 is of this form):

$$\text{NN}_{\text{MLP2}}(\boldsymbol{x}) = (g^2(g^1(\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{1}} + \boldsymbol{b}^{\boldsymbol{1}})\boldsymbol{W}^{\boldsymbol{2}} + \boldsymbol{b}^{\boldsymbol{2}}))\boldsymbol{W}^{\boldsymbol{3}}. \tag{4.3}$$

It is perhaps clearer to write deeper networks like this using intermediary variables:

$$\begin{aligned} \text{NN}_{\text{MLP2}}(\boldsymbol{x}) &= \boldsymbol{y} \\ \boldsymbol{h}^{\boldsymbol{1}} &= g^1(\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{1}} + \boldsymbol{b}^{\boldsymbol{1}}) \\ \boldsymbol{h}^{\boldsymbol{2}} &= g^2(\boldsymbol{h}^{\boldsymbol{1}}\boldsymbol{W}^{\boldsymbol{2}} + \boldsymbol{b}^{\boldsymbol{2}}) \\ \boldsymbol{y} &= \boldsymbol{h}^{\boldsymbol{2}}\boldsymbol{W}^{\boldsymbol{3}}. \end{aligned} \tag{4.4}$$

[3] The network in Figure 4.2 does not include bias terms. A bias term can be added to a layer by adding to it an additional neuron that does not have any incoming connections, whose value is always 1.

[4] To see why, consider that a sequence of linear transformations is still a linear transformation.

The vector resulting from each linear transform is referred to as a *layer*. The outer-most linear transform results in the *output layer* and the other linear transforms result in *hidden layers*. Each hidden layer is followed by a nonlinear activation. In some cases, such as in the last layer of our example, the bias vectors are forced to 0 ("dropped").

Layers resulting from linear transformations are often referred to as *fully connected*, or *affine*. Other types of architectures exist. In particular, image recognition problems benefit from *convolutional* and *pooling* layers. Such layers have uses also in language processing, and will be discussed in Chapter 13. Networks with several hidden layers are said to be *deep* networks, hence the name *deep learning*.

When describing a neural network, one should specify the *dimensions* of the layers and the input. A layer will expect a d_{in} dimensional vector as its input, and transform it into a d_{out} dimensional vector. The dimensionality of the layer is taken to be the dimensionality of its output. For a fully connected layer $l(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$ with input dimensionality d_{in} and output dimensionality d_{out}, the dimensions of $\boldsymbol{x}$ is $1 \times d_{in}$, of $\boldsymbol{W}$ is $d_{in} \times d_{out}$ and of $\boldsymbol{b}$ is $1 \times d_{out}$.

Like the case with linear models, the output of a neural network is a d_{out} dimensional vector. In case $d_{out} = 1$, the network's output is a scalar. Such networks can be used for regression (or scoring) by considering the value of the output, or for binary classification by consulting the sign of the output. Networks with $d_{out} = k > 1$ can be used for k-class classification, by associating each dimension with a class, and looking for the dimension with maximal value. Similarly, if the output vector entries are positive and sum to one, the output can be interpreted as a distribution over class assignments (such output normalization is typically achieved by applying a softmax transformation on the output layer, see Section 2.6).

The matrices and the bias terms that define the linear transformations are the *parameters* of the network. Like in linear models, it is common to refer to the collection of all parameters as Θ. Together with the input, the parameters determine the network's output. The training algorithm is responsible for setting their values such that the network's predictions are correct. Unlike linear models, the loss function of multi-layer neural networks with respect to their parameters is not convex,[5] making search for the optimal parameter values intractable. Still, the gradient-based optimization methods discussed in Section 2.8 can be applied, and perform very well in practice. Training neural networks is discussed in detail in Chapter 5.

4.3 REPRESENTATION POWER

In terms of representation power, it was shown by Hornik et al. [1989] and Cybenko [1989] that MLP1 is a universal approximator—it can approximate with any desired non-zero amount of error a family of functions that includes all continuous functions on a closed and bounded subset of $\mathbb{R}^n$, and any function mapping from any finite dimensional discrete space to another.[6] This

[5] Strictly convex functions have a single optimal solution, making them easy to optimize using gradient-based methods.

[6] Specifically, a feed-forward network with linear output layer and at least one hidden layer with a "squashing" activation function can approximate any Borel measurable function from one finite dimensional space to another. The proof was later extended by Leshno et al. [1993] to a wider range of activation functions, including the ReLU function $g(x) = \max(0, x)$.

may suggest there is no reason to go beyond MLP1 to more complex architectures. However, the theoretical result does not discuss the learnability of the neural network (it states that a representation exists, but does not say how easy or hard it is to set the parameters based on training data and a specific learning algorithm). It also does not guarantee that a training algorithm will find the *correct* function generating our training data. Finally, it does not state how large the hidden layer should be. Indeed, Telgarsky [2016] show that there exist neural networks with many layers of bounded size that cannot be approximated by networks with fewer layers unless these layers are exponentially large.

In practice, we train neural networks on relatively small amounts of data using local search methods such as variants of stochastic gradient descent, and use hidden layers of relatively modest sizes (up to several thousands). As the universal approximation theorem does not give any guarantees under these non-ideal, real-world conditions, there is definitely benefit to be had in trying out more complex architectures than MLP1. In many cases, however, MLP1 does indeed provide strong results. For further discussion on the representation power of feed-forward neural networks, see Bengio et al. [2016, Section 6.5].

4.4 COMMON NONLINEARITIES

The nonlinearity g can take many forms. There is currently no good theory as to which nonlinearity to apply in which conditions, and choosing the correct nonlinearity for a given task is for the most part an empirical question. I will now go over the common nonlinearities from the literature: the sigmoid, tanh, hard tanh and the rectified linear unit (ReLU). Some NLP researchers also experimented with other forms of nonlinearities such as cube and tanh-cube.

Sigmoid The sigmoid activation function $\sigma(x) = 1/(1 + e^{-x})$, also called the logistic function, is an S-shaped function, transforming each value x into the range $[0, 1]$. The sigmoid was the canonical nonlinearity for neural networks since their inception, but is currently considered to be deprecated for use in internal layers of neural networks, as the choices listed below prove to work much better empirically.

Hyperbolic tangent (tanh) The hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range $[-1, 1]$.

Hard tanh The hard-tanh activation function is an approximation of the tanh function which is faster to compute and to find derivatives thereof:

$$\text{hardtanh}(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise.} \end{cases} \tag{4.5}$$

Rectifier (ReLU) The rectifier activation function [Glorot et al., 2011], also known as the rectified linear unit is a very simple activation function that is easy to work with and was shown many

times to produce excellent results.[7] The ReLU unit clips each value $x < 0$ at 0. Despite its simplicity, it performs well for many tasks, especially when combined with the dropout regularization technique (see Section 4.6):

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise.} \end{cases} \tag{4.6}$$

As a rule of thumb, both ReLU and tanh units work well, and significantly outperform the sigmoid. You may want to experiment with both tanh and ReLU activations, as each one may perform better in different settings.

Figure 4.3 shows the shapes of the different activations functions, together with the shapes of their derivatives.

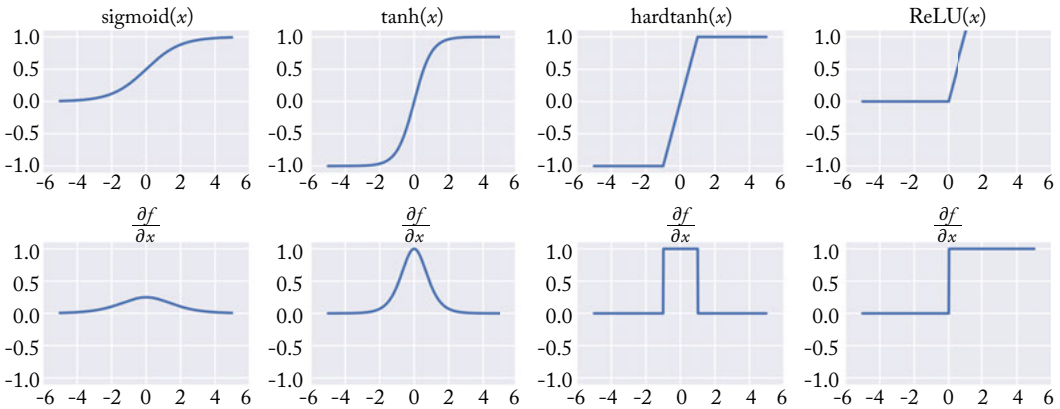

Figure 4.3: Activation functions (top) and their derivatives (bottom).

4.5 LOSS FUNCTIONS

When training a neural network (more on training in Chapter 5), much like when training a linear classifier, one defines a loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$, stating the loss of predicting $\hat{\boldsymbol{y}}$ when the true output is $\boldsymbol{y}$. The training objective is then to minimize the loss across the different training examples. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ assigns a numerical score (a scalar) to the network's output $\hat{\boldsymbol{y}}$ given the true expected output $\boldsymbol{y}$. The loss functions discussed for linear models in Section 2.7.1 are relevant and widely used also for neural networks. For further discussion on loss functions in the

[7]The technical advantages of the ReLU over the sigmoid and tanh activation functions is that it does not involve expensive-to-compute functions, and more importantly that it does not saturate. The sigmoid and tanh activation are capped at 1, and the gradients at this region of the functions are near zero, driving the entire gradient near zero. The ReLU activation does not have this problem, making it especially suitable for networks with multiple layers, which are susceptible to the vanishing gradients problem when trained with the saturating units.

context of neural networks, see LeCun and Huang [2005], LeCun et al. [2006] and Bengio et al. [2016].

4.6 REGULARIZATION AND DROPOUT

Multi-layer networks can be large and have many parameters, making them especially prone to overfitting. Model regularization is just as important in deep neural networks as it is in linear models, and perhaps even more so. The regularizers discussed in Section 2.7.2, namely L_2, L_1 and the elastic-net, are also relevant for neural networks. In particular, L_2 regularization, also called *weight decay* is effective for achieving good generalization performance in many cases, and tuning the regularization strength λ is advisable.

Another effective technique for preventing neural networks from overfitting the training data is *dropout training* [Hinton et al., 2012, Srivastava et al., 2014]. The dropout method is designed to prevent the network from learning to rely on specific weights. It works by randomly dropping (setting to 0) half of the neurons in the network (or in a specific layer) in each training example in the stochastic-gradient training. For example, consider the multi-layer perceptron with two hidden layers (MLP2):

$$\begin{aligned}
\text{NN}_{\text{MLP2}}(\boldsymbol{x}) &= \boldsymbol{y} \\
\boldsymbol{h^1} &= g^1(\boldsymbol{x}\boldsymbol{W^1} + \boldsymbol{b^1}) \\
\boldsymbol{h^2} &= g^2(\boldsymbol{h^1}\boldsymbol{W^2} + \boldsymbol{b^2}) \\
\boldsymbol{y} &= \boldsymbol{h^2}\boldsymbol{W^3}.
\end{aligned}$$

When applying dropout training to MLP2, we randomly set some of the values of $\boldsymbol{h^1}$ and $\boldsymbol{h^2}$ to 0 at each training round:

$$\begin{aligned}
\text{NN}_{\text{MLP2}}(\boldsymbol{x}) &= \boldsymbol{y} \\
\boldsymbol{h^1} &= g^1(\boldsymbol{x}\boldsymbol{W^1} + \boldsymbol{b^1}) \\
\boldsymbol{m^1} &\sim \text{Bernouli}(r^1) \\
\tilde{\boldsymbol{h}}^{\boldsymbol{1}} &= \boldsymbol{m^1} \odot \boldsymbol{h^1} \\
\boldsymbol{h^2} &= g^2(\tilde{\boldsymbol{h}}^{\boldsymbol{1}}\boldsymbol{W^2} + \boldsymbol{b^2}) \\
\boldsymbol{m^2} &\sim \text{Bernouli}(r^2) \\
\tilde{\boldsymbol{h}}^{\boldsymbol{2}} &= \boldsymbol{m^2} \odot \boldsymbol{h^2} \\
\boldsymbol{y} &= \tilde{\boldsymbol{h}}^{\boldsymbol{2}}\boldsymbol{W^3}.
\end{aligned} \tag{4.7}$$

Here, $\boldsymbol{m^1}$ and $\boldsymbol{m^2}$ are random *masking vectors* with the dimensions of $\boldsymbol{h^1}$ and $\boldsymbol{h^2}$, respectively, and $\odot$ is the element-wise multiplication operation. The values of the elements in the masking

vectors are either 0 or 1, and are drawn from a Bernouli distribution with parameter r (usually $r = 0.5$). The values corresponding to zeros in the masking vectors are then zeroed out, replacing the hidden layers $\boldsymbol{h}$ with $\tilde{\boldsymbol{h}}$ before passing them on to the next layer.

Work by Wager et al. [2013] establishes a strong connection between the dropout method and L_2 regularization. Another view links dropout to model averaging and ensemble techniques [Srivastava et al., 2014].

The dropout technique is one of the key factors contributing to very strong results of neural-network methods on image classification tasks [Krizhevsky et al., 2012], especially when combined with ReLU activation units [Dahl et al., 2013]. The dropout technique is effective also in NLP applications of neural networks.

4.7 SIMILARITY AND DISTANCE LAYERS

We sometimes wish to calculate a scalar value based on two vectors, such that the value reflects the *similarity*, *compatibility* or *distance* between the two vectors. For example, vectors $\boldsymbol{v_1} \in \mathbb{R}^d$ and $\boldsymbol{v_2} \in \mathbb{R}^d$ may be the output layers of two MLPs, and we would like to train the network to produce similar vectors for some training examples, and dissimilar vectors for others.

In what follows we describe common functions that take two vectors $\boldsymbol{u} \in \mathbb{R}^d$ and $\boldsymbol{v} \in \mathbb{R}^d$, and return a scalar. These functions can (and often are) integrated in feed-forward neural networks.

Dot Product A very common options is to use the dot-product:

$$\text{sim}_{\text{dot}}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u} \cdot \boldsymbol{v} = \sum_{i=1}^{d} \boldsymbol{u}_{[i]} \boldsymbol{v}_{[i]} \tag{4.8}$$

Euclidean Distance Another popular options is the Euclidean Distance:

$$\text{dist}_{\text{euclidean}}(\boldsymbol{u}, \boldsymbol{v}) = \sqrt{\sum_{i=1}^{d} (\boldsymbol{u}_{[i]} - \boldsymbol{v}_{[i]})^2} = \sqrt{(\boldsymbol{u} - \boldsymbol{v}) \cdot (\boldsymbol{u} - \boldsymbol{v})} = ||\boldsymbol{u} - \boldsymbol{v}||_2 \tag{4.9}$$

Note that this is a distance metric and not a similarity: here, small (near zero) values indicate similar vectors and large values dissimilar ones. The square-root is often omitted.

Trainable Forms The dot-product and the euclidean distance above are fixed functions. We sometimes want to use a parameterized function, that can be trained to produce desired similarity (or dissimilarity) values by focusing on specific dimensions of the vectors. A common trainable similarity function is the *bilinear form*:

$$\text{sim}_{\text{bilinear}}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}\boldsymbol{M}\boldsymbol{v} \tag{4.10}$$

$$\boldsymbol{M} \in \mathbb{R}^{d \times d}$$

where the matrix $\boldsymbol{M}$ is a parameter that needs to be trained.

Similarly, for a trainable distance function we can use:

$$\text{dist}(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u} - \boldsymbol{v})\boldsymbol{M}(\boldsymbol{u} - \boldsymbol{v}) \tag{4.11}$$

Finally, a multi-layer perceptron with a single output neuron can also be used for producing a scalar from two vectors, by feeding it the concatenation of the two vectors.

4.8 EMBEDDING LAYERS

As will be further discussed in Chapter 8, when the input to the neural network contains symbolic categorical features (e.g., features that take one of k distinct symbols, such as words from a closed vocabulary), it is common to associate each possible feature value (i.e., each word in the vocabulary) with a d-dimensional vector for some d. These vectors are then considered parameters of the model, and are trained jointly with the other parameters. The mapping from a symbolic feature values such as "word number 1249" to d-dimensional vectors is performed by an *embedding layer* (also called a *lookup layer*). The parameters in an embedding layer are simply a matrix $\boldsymbol{E} \in \mathbb{R}^{|\text{vocab}| \times d}$ where each row corresponds to a different word in the vocabulary. The lookup operation is then simply indexing: $v_{1249} = \boldsymbol{E}_{[1249,:]}$. If the symbolic feature is encoded as a one-hot vector $\boldsymbol{x}$, the lookup operation can be implemented as the multiplication $\boldsymbol{x}\boldsymbol{E}$.

The word vectors are often concatenated to each other before being passed on to the next layer. Embeddings are discussed in more depth in Chapter 8 when discussing dense representations of categorical features, and in Chapter 10 when discussing pre-trained word representations.