Chapter 3

Nonlinear classification

Linear classification may seem like all we need for natural language processing. The bag-of-words representation is inherently high dimensional, and the number of features is often larger than the number of labeled training instances. This means that it is usually possible to find a linear classifier that perfectly fits the training data, or even to fit any arbitrary labeling of the training instances! Moving to nonlinear classification may therefore only increase the risk of overfitting. Furthermore, for many tasks, **lexical features** (words) are meaningful in isolation, and can offer independent evidence about the instance label — unlike computer vision, where individual pixels are rarely informative, and must be evaluated holistically to make sense of an image. For these reasons, natural language processing has historically focused on linear classification.

But in recent years, nonlinear classifiers have swept through natural language processing, and are now the default approach for many tasks (Manning, 2015). There are at least three reasons for this change.

- There have been rapid advances in **deep learning**, a family of nonlinear methods that learn complex functions of the input through multiple layers of computation (Goodfellow et al., 2016).
- Deep learning facilitates the incorporation of **word embeddings**, which are dense vector representations of words. Word embeddings can be learned from large amounts of unlabeled data, and enable generalization to words that do not appear in the annotated training data (word embeddings are discussed in detail in chapter 14).
- While CPU speeds have plateaued, there have been rapid advances in specialized hardware called graphics processing units (GPUs), which have become faster, cheaper, and easier to program. Many deep learning models can be implemented efficiently on GPUs, offering substantial performance improvements over CPU-based computing.

This chapter focuses on **neural networks**, which are the dominant approach for nonlinear classification in natural language processing today.[1] Historically, a few other nonlinear learning methods have been applied to language data.

- **Kernel methods** are generalizations of the **nearest-neighbor** classification rule, which classifies each instance by the label of the most similar example in the training set. The application of the **kernel support vector machine** to information extraction is described in chapter 17.
- **Decision trees** classify instances by checking a set of conditions. Scaling decision trees to bag-of-words inputs is difficult, but decision trees have been successful in problems such as coreference resolution (chapter 15), where more compact feature sets can be constructed (Soon et al., 2001).
- **Boosting** and related **ensemble methods** work by combining the predictions of several "weak" classifiers, each of which may consider only a small subset of features. Boosting has been successfully applied to text classification (Schapire and Singer, 2000) and syntactic analysis (Abney et al., 1999), and remains one of the most successful methods on machine learning competition sites such as Kaggle (Chen and Guestrin, 2016).

Hastie et al. (2009) provide an excellent overview of these techniques.

3.1 Feedforward neural networks

Consider the problem of building a classifier for movie reviews. The goal is to predict a label $y \in \{\text{GOOD}, \text{BAD}, \text{OKAY}\}$ from a representation of the text of each document, $\boldsymbol{x}$. But what makes a good movie? The story, acting, cinematography, editing, soundtrack, and so on. Now suppose the training set contains labels for each of these additional features, $\boldsymbol{z} = [z_1, z_2, \ldots, z_{K_z}]^\top$. With a training set of such information, we could build a two-step classifier:

1. **Use the text $\boldsymbol{x}$ to predict the features $\boldsymbol{z}$.** Specifically, train a logistic regression classifier to compute $\text{p}(z_k \mid \boldsymbol{x})$, for each $k \in \{1, 2, \ldots, K_z\}$.
2. **Use the features $\boldsymbol{z}$ to predict the label y.** Again, train a logistic regression classifier to compute $\text{p}(y \mid \boldsymbol{z})$. On test data, $\boldsymbol{z}$ is unknown, so we will use the probabilities $\text{p}(\boldsymbol{z} \mid \boldsymbol{x})$ from the first layer as the features.

This setup is shown in Figure 3.1, which describes the proposed classifier in a **computation graph**: the text features $\boldsymbol{x}$ are connected to the middle layer $\boldsymbol{z}$, which is connected to the label y.

[1] I will use "deep learning" and "neural networks" interchangeably.

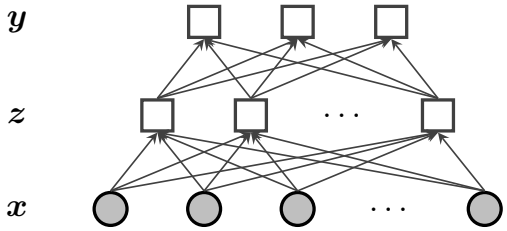

Figure 3.1: A feedforward neural network. Shaded circles indicate observed features, usually words; squares indicate nodes in the computation graph, which are computed from the information carried over the incoming arrows.

If we assume that each z_k is binary, $z_k \in \{0, 1\}$, then the probability $\text{p}(z_k \mid \boldsymbol{x})$ can be modeled using binary logistic regression:

$$\Pr(z_k = 1 \mid \boldsymbol{x}; \boldsymbol{\Theta}^{(x\rightarrow z)}) = \sigma(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x}) = (1 + \exp(-\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x}))^{-1}, \quad [3.1]$$

where σ is the **sigmoid** function (shown in Figure 3.2), and the matrix $\boldsymbol{\Theta}^{(x\rightarrow z)} \in \mathbb{R}^{K_z \times V}$ is constructed by stacking the weight vectors for each z_k,

$$\boldsymbol{\Theta}^{(x\rightarrow z)} = [\boldsymbol{\theta}_1^{(x\rightarrow z)}, \boldsymbol{\theta}_2^{(x\rightarrow z)}, \ldots, \boldsymbol{\theta}_{K_z}^{(x\rightarrow z)}]^\top. \quad [3.2]$$

We will assume that $\boldsymbol{x}$ contains a term with a constant value of 1, so that a corresponding offset parameter is included in each $\boldsymbol{\theta}_k^{(x\rightarrow z)}$.

The output layer is computed by the multi-class logistic regression probability,

$$\Pr(y = j \mid \boldsymbol{z}; \boldsymbol{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) = \frac{\exp(\boldsymbol{\theta}_j^{(z\rightarrow y)} \cdot \boldsymbol{z} + b_j)}{\sum_{j' \in \mathcal{Y}} \exp(\boldsymbol{\theta}_{j'}^{(z\rightarrow y)} \cdot \boldsymbol{z} + b_{j'})}, \quad [3.3]$$

where b_j is an offset for label j, and the output weight matrix $\boldsymbol{\Theta}^{(z\rightarrow y)} \in \mathbb{R}^{K_y \times K_z}$ is again constructed by concatenation,

$$\boldsymbol{\Theta}^{(z\rightarrow y)} = [\boldsymbol{\theta}_1^{(z\rightarrow y)}, \boldsymbol{\theta}_2^{(z\rightarrow y)}, \ldots, \boldsymbol{\theta}_{K_y}^{(z\rightarrow y)}]^\top. \quad [3.4]$$

The vector of probabilities over each possible value of y is denoted,

$$\text{p}(\boldsymbol{y} \mid \boldsymbol{z}; \boldsymbol{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z\rightarrow y)} \boldsymbol{z} + \boldsymbol{b}), \quad [3.5]$$

where element j in the output of the **SoftMax** function is computed as in Equation 3.3.

This set of equations defines a multilayer classifier, which can be summarized as,

$$\text{p}(\boldsymbol{z} \mid \boldsymbol{x}; \boldsymbol{\Theta}^{(x\rightarrow z)}) = \sigma(\boldsymbol{\Theta}^{(x\rightarrow z)} \boldsymbol{x}) \quad [3.6]$$

$$\text{p}(\boldsymbol{y} \mid \boldsymbol{z}; \boldsymbol{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z\rightarrow y)} \boldsymbol{z} + \boldsymbol{b}), \quad [3.7]$$

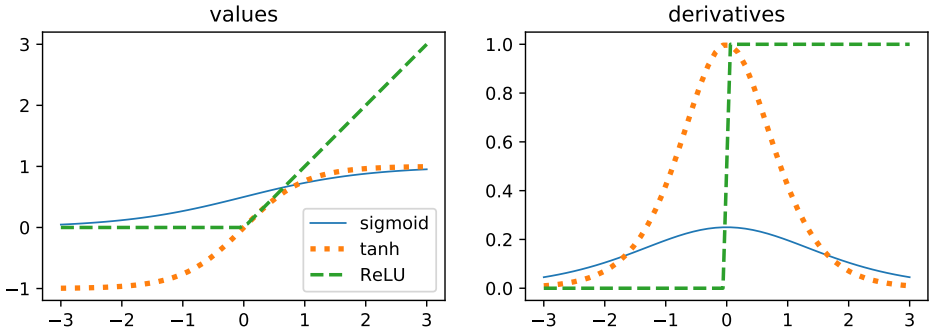

Figure 3.2: The sigmoid, tanh, and ReLU activation functions

where the function σ is now applied **elementwise** to the vector of inner products,

$$\sigma(\mathbf{\Theta}^{(x\rightarrow z)}\boldsymbol{x}) = [\sigma(\boldsymbol{\theta}_1^{(x\rightarrow z)} \cdot \boldsymbol{x}), \sigma(\boldsymbol{\theta}_2^{(x\rightarrow z)} \cdot \boldsymbol{x}), \ldots, \sigma(\boldsymbol{\theta}_{K_z}^{(x\rightarrow z)} \cdot \boldsymbol{x})]^\top. \quad [3.8]$$

Now suppose that the hidden features $\boldsymbol{z}$ are never observed, even in the training data. We can still construct the architecture in Figure 3.1. Instead of predicting y from a discrete vector of predicted values $\boldsymbol{z}$, we use the probabilities $\sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x})$. The resulting classifier is barely changed:

$$\boldsymbol{z} = \sigma(\mathbf{\Theta}^{(x\rightarrow z)}\boldsymbol{x}) \quad [3.9]$$

$$\text{p}(y \mid \boldsymbol{x}; \mathbf{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) = \text{SoftMax}(\mathbf{\Theta}^{(z\rightarrow y)}\boldsymbol{z} + \boldsymbol{b}). \quad [3.10]$$

This defines a classification model that predicts the label $y \in \mathcal{Y}$ from the base features $\boldsymbol{x}$, through a"hidden layer" $\boldsymbol{z}$. This is a **feedforward neural network**.[2]

3.2 Designing neural networks

There several ways to generalize the feedforward neural network.

3.2.1 Activation functions

If the hidden layer is viewed as a set of latent features, then the sigmoid function in Equation 3.9 represents the extent to which each of these features is "activated" by a given input. However, the hidden layer can be regarded more generally as a nonlinear transformation of the input. This opens the door to many other activation functions, some of which are shown in Figure 3.2. At the moment, the choice of activation functions is more art than science, but a few points can be made about the most popular varieties:

[2]The architecture is sometimes called a **multilayer perceptron**, but this is misleading, because each layer is not a perceptron as defined in the previous chapter.

- The range of the sigmoid function is $(0, 1)$. The bounded range ensures that a cascade of sigmoid functions will not "blow up" to a huge output, and this is important for deep networks with several hidden layers. The derivative of the sigmoid is $\frac{\partial}{\partial a}\sigma(a) = \sigma(a)(1 - \sigma(a))$. This derivative becomes small at the extremes, which can make learning slow; this is called the **vanishing gradient** problem.
- The range of the **tanh activation function** is $(-1, 1)$: like the sigmoid, the range is bounded, but unlike the sigmoid, it includes negative values. The derivative is $\frac{\partial}{\partial a}\tanh(a) = 1 - \tanh(a)^2$, which is steeper than the logistic function near the origin (LeCun et al., 1998). The tanh function can also suffer from vanishing gradients at extreme values.
- The **rectified linear unit (ReLU)** is zero for negative inputs, and linear for positive inputs (Glorot et al., 2011),

$$\text{ReLU}(a) = \begin{cases} a, & a \geq 0 \\ 0, & \text{otherwise.} \end{cases} \quad [3.11]$$

  The derivative is a step function, which is 1 if the input is positive, and zero otherwise. Once the activation is zero, the gradient is also zero. This can lead to the problem of "dead neurons", where some ReLU nodes are zero for all inputs, throughout learning. A solution is the **leaky ReLU**, which has a small positive slope for negative inputs (Maas et al., 2013),

$$\text{Leaky-ReLU}(a) = \begin{cases} a, & a \geq 0 \\ .0001a, & \text{otherwise.} \end{cases} \quad [3.12]$$

Sigmoid and tanh are sometimes described as **squashing functions**, because they squash an unbounded input into a bounded range. Glorot and Bengio (2010) recommend against the use of the sigmoid activation in deep networks, because its mean value of $\frac{1}{2}$ can cause the next layer of the network to be saturated, leading to small gradients on its own parameters. Several other activation functions are reviewed in the textbook by Goodfellow et al. (2016), who recommend ReLU as the "default option."

3.2.2 Network structure

Deep networks stack up several hidden layers, with each $\boldsymbol{z}^{(d)}$ acting as the input to the next layer, $\boldsymbol{z}^{(d+1)}$. As the total number of nodes in the network increases, so does its capacity to learn complex functions of the input. Given a fixed number of nodes, one must decide whether to emphasize width (large K_z at each layer) or depth (many layers). At present, this tradeoff is not well understood.[3]

[3] With even a single hidden layer, a neural network can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^N$ to an arbitrarily small non-zero error; see section 6.4.1 of Goodfellow et al. (2016)

It is also possible to "short circuit" a hidden layer, by propagating information directly from the input to the next higher level of the network. This is the idea behind **residual networks**, which propagate information directly from the input to the subsequent layer (He et al., 2016),

$$\boldsymbol{z} = f(\boldsymbol{\Theta}^{(x \to z)}\boldsymbol{x}) + \boldsymbol{x}, \tag{3.13}$$

where f is any nonlinearity, such as sigmoid or ReLU. A more complex architecture is the **highway network** (Srivastava et al., 2015; Kim et al., 2016), in which an addition **gate** controls an interpolation between $f(\boldsymbol{\Theta}^{(x \to z)}\boldsymbol{x})$ and $\boldsymbol{x}$,

$$\boldsymbol{t} = \sigma(\Theta^{(t)}\boldsymbol{x} + \boldsymbol{b}^{(t)}) \tag{3.14}$$

$$\boldsymbol{z} = \boldsymbol{t} \odot f(\boldsymbol{\Theta}^{(x \to z)}\boldsymbol{x}) + (\mathbf{1} - \boldsymbol{t}) \odot \boldsymbol{x}, \tag{3.15}$$

where $\odot$ refers to an elementwise vector product, and $\mathbf{1}$ is a column vector of ones. As before, the sigmoid function is applied elementwise to its input; recall that the output of this function is restricted to the range $(0, 1)$. Gating is also used in the **long short-term memory (LSTM)**, which is discussed in chapter 6. Residual and highway connections address a problem with deep architectures: repeated application of a nonlinear activation function can make it difficult to learn the parameters of the lower levels of the network, which are too distant from the supervision signal.

3.2.3 Outputs and loss functions

In the multi-class classification example, a softmax output produces probabilities over each possible label. This aligns with a negative **conditional log-likelihood**,

$$-\mathcal{L} = -\sum_{i=1}^{N} \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \Theta). \tag{3.16}$$

where $\Theta = \{\boldsymbol{\Theta}^{(x \to z)}, \boldsymbol{\Theta}^{(z \to y)}, \boldsymbol{b}\}$ is the entire set of parameters.

This loss can be written alternatively as follows:

$$\tilde{y}_j \triangleq \Pr(y = j \mid \boldsymbol{x}^{(i)}; \Theta) \tag{3.17}$$

$$-\mathcal{L} = -\sum_{i=1}^{N} \boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}} \tag{3.18}$$

for a survey of these theoretical results. However, depending on the function to be approximated, the width of the hidden layer may need to be arbitrarily large. Furthermore, the fact that a network has the *capacity* to approximate any given function does not imply that it is possible to *learn* the function using gradient-based optimization.

where $\boldsymbol{e}_{y^{(i)}}$ is a **one-hot vector** of zeros with a value of 1 at position $y^{(i)}$. The inner product between $\boldsymbol{e}_{y^{(i)}}$ and $\log \tilde{\boldsymbol{y}}$ is also called the multinomial **cross-entropy**, and this terminology is preferred in many neural networks papers and software packages.

It is also possible to train neural networks from other objectives, such as a margin loss. In this case, it is not necessary to use softmax at the output layer: an affine transformation of the hidden layer is enough:

$$\Psi(y; \boldsymbol{x}^{(i)}, \Theta) = \boldsymbol{\theta}_y^{(z \to y)} \cdot \boldsymbol{z} + b_y \quad [3.19]$$

$$\ell_{\text{MARGIN}}(\Theta; \boldsymbol{x}^{(i)}, y^{(i)}) = \max_{y \neq y^{(i)}} \Big(1 + \Psi(y; \boldsymbol{x}^{(i)}, \Theta) - \Psi(y^{(i)}; \boldsymbol{x}^{(i)}, \Theta)\Big)_+ . \quad [3.20]$$

In regression problems, the output is a scalar or vector (see § 4.1.2). For these problems, a typical loss function is the squared error $(y - \hat{y})^2$ or squared norm $||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2^2$.

3.2.4 Inputs and lookup layers

In text classification, the input layer $\boldsymbol{x}$ can refer to a bag-of-words vector, where x_j is the count of word j. The input to the hidden unit z_k is then $\sum_{j=1}^{V} \theta_{j,k}^{(x \to z)} x_j$, and word j is represented by the vector $\boldsymbol{\theta}_j^{(x \to z)}$. This vector is sometimes described as the **embedding** of word j, and can be learned from unlabeled data, using techniques discussed in chapter 14. The columns of $\boldsymbol{\Theta}^{(x \to z)}$ are each K_z-dimensional word embeddings.

Chapter 2 presented an alternative view of text documents, as a sequence of word tokens, $w_1, w_2, \ldots, w_M$. In a neural network, each word token w_m is represented with a one-hot vector, $\boldsymbol{e}_{w_m}$, with dimension V. The matrix-vector product $\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{e}_{w_m}$ returns the embedding of word w_m. The complete document can represented by horizontally concatenating these one-hot vectors, $\mathbf{W} = [\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \ldots, \boldsymbol{e}_{w_M}]$, and the bag-of-words representation can be recovered from the matrix-vector product $\mathbf{W}[1, 1, \ldots, 1]^\top$, which sums each row over the tokens $m = \{1, 2, \ldots, M\}$. The matrix product $\boldsymbol{\Theta}^{(x \to z)} \mathbf{W}$ contains the horizontally concatenated embeddings of each word in the document, which will be useful as the starting point for **convolutional neural networks** (see § 3.4). This is sometimes called a **lookup layer**, because the first step is to lookup the embeddings for each word in the input text.

3.3 Learning neural networks

The feedforward network in Figure 3.1 can now be written as,

$$\boldsymbol{z} \leftarrow f(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x}^{(i)}) \quad [3.21]$$

$$\tilde{\boldsymbol{y}} \leftarrow \text{SoftMax}\Big(\boldsymbol{\Theta}^{(z \to y)} \boldsymbol{z} + \boldsymbol{b}\Big) \quad [3.22]$$

$$\ell^{(i)} \leftarrow - \boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}}, \quad [3.23]$$

where f is an elementwise activation function, such as σ or ReLU, and $\ell^{(i)}$ is the loss at instance i. The parameters $\mathbf{\Theta}^{(x\rightarrow z)}$, $\mathbf{\Theta}^{(z\rightarrow y)}$, and $\boldsymbol{b}$ can be estimated using online gradient-based optimization. The simplest such algorithm is stochastic gradient descent, which was discussed in § 2.6. Each parameter is updated by the gradient of the loss,

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \eta^{(t)} \nabla_{\boldsymbol{b}} \ell^{(i)} \tag{3.24}$$

$$\boldsymbol{\theta}_k^{(z\rightarrow y)} \leftarrow \boldsymbol{\theta}_k^{(z\rightarrow y)} - \eta^{(t)} \nabla_{\boldsymbol{\theta}_k^{(z\rightarrow y)}} \ell^{(i)} \tag{3.25}$$

$$\boldsymbol{\theta}_n^{(x\rightarrow z)} \leftarrow \boldsymbol{\theta}_n^{(x\rightarrow z)} - \eta^{(t)} \nabla_{\boldsymbol{\theta}_n^{(x\rightarrow z)}} \ell^{(i)}, \tag{3.26}$$

where $\eta^{(t)}$ is the learning rate on iteration t, $\ell^{(i)}$ is the loss on instance (or minibatch) i, and $\boldsymbol{\theta}_n^{(x\rightarrow z)}$ is column n of the matrix $\mathbf{\Theta}^{(x\rightarrow z)}$, and $\boldsymbol{\theta}_k^{(z\rightarrow y)}$ is column k of $\mathbf{\Theta}^{(z\rightarrow y)}$.

The gradients of the negative log-likelihood on $\boldsymbol{b}$ and $\boldsymbol{\theta}_k^{(z\rightarrow y)}$ are similar to the gradients in logistic regression. For $\boldsymbol{\theta}^{(z\rightarrow y)}$, the gradient is,

$$\nabla_{\boldsymbol{\theta}_k^{(z\rightarrow y)}} \ell^{(i)} = \left[\frac{\partial \ell^{(i)}}{\partial \theta_{k,1}^{(z\rightarrow y)}}, \frac{\partial \ell^{(i)}}{\partial \theta_{k,2}^{(z\rightarrow y)}}, \ldots, \frac{\partial \ell^{(i)}}{\partial \theta_{k,K_y}^{(z\rightarrow y)}} \right]^\top \tag{3.27}$$

$$\frac{\partial \ell^{(i)}}{\partial \theta_{k,j}^{(z\rightarrow y)}} = -\frac{\partial}{\partial \theta_{k,j}^{(z\rightarrow y)}} \left(\boldsymbol{\theta}_{y^{(i)}}^{(z\rightarrow y)} \cdot \boldsymbol{z} - \log \sum_{y \in \mathcal{Y}} \exp \boldsymbol{\theta}_y^{(z\rightarrow y)} \cdot \boldsymbol{z} \right) \tag{3.28}$$

$$= \left(\Pr(y = j \mid \boldsymbol{z}; \mathbf{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) - \delta\left(j = y^{(i)}\right) \right) z_k, \tag{3.29}$$

where $\delta\left(j = y^{(i)}\right)$ is a function that returns one when $j = y^{(i)}$, and zero otherwise. The gradient $\nabla_{\boldsymbol{b}} \ell^{(i)}$ is similar to Equation 3.29.

The gradients on the input layer weights $\mathbf{\Theta}^{(x\rightarrow z)}$ are obtained by the chain rule of differentiation:

$$\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\rightarrow z)}} = \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial z_k}{\partial \theta_{n,k}^{(x\rightarrow z)}} \tag{3.30}$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial f(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x})}{\partial \theta_{n,k}^{(x\rightarrow z)}} \tag{3.31}$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} \times f'(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x}) \times x_n, \tag{3.32}$$

where $f'(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x})$ is the derivative of the activation function f, applied at the input

$\boldsymbol{\theta}_k^{(x\to z)} \cdot \boldsymbol{x}$. For example, if f is the sigmoid function, then the derivative is,

$$\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\to z)}} = \frac{\partial \ell^{(i)}}{\partial z_k} \times \sigma(\boldsymbol{\theta}_k^{(x\to z)} \cdot \boldsymbol{x}) \times (1 - \sigma(\boldsymbol{\theta}_k^{(x\to z)} \cdot \boldsymbol{x})) \times x_n \quad [3.33]$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} \times z_k \times (1 - z_k) \times x_n. \quad [3.34]$$

For intuition, consider each of the terms in the product.

- If the negative log-likelihood $\ell^{(i)}$ does not depend much on z_k, then $\frac{\partial \ell^{(i)}}{\partial z_k} \approx 0$. In this case it doesn't matter how z_k is computed, and so $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\to z)}} \approx 0$.
- If z_k is near 1 or 0, then the curve of the sigmoid function is nearly flat (Figure 3.2), and changing the inputs will make little local difference. The term $z_k \times (1 - z_k)$ is maximized at $z_k = \frac{1}{2}$, where the slope of the sigmoid function is steepest.
- If $x_n = 0$, then it does not matter how we set the weights $\theta_{n,k}^{(x\to z)}$, so $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\to z)}} = 0$.

3.3.1 Backpropagation

The equations above rely on the chain rule to compute derivatives of the loss with respect to each parameter of the model. Furthermore, local derivatives are frequently reused: for example, $\frac{\partial \ell^{(i)}}{\partial z_k}$ is reused in computing the derivatives with respect to each $\theta_{n,k}^{(x\to z)}$. These terms should therefore be computed once, and then cached. Furthermore, we should only compute any derivative once we have already computed all of the necessary "inputs" demanded by the chain rule of differentiation. This combination of sequencing, caching, and differentiation is known as **backpropagation**. It can be generalized to any directed acyclic **computation graph**.

A computation graph is a declarative representation of a computational process. At each node t, compute a value v_t by applying a function f_t to a (possibly empty) list of parent nodes, $\boldsymbol{\pi}_t$. Figure 3.3 shows the computation graph for a feedforward network with one hidden layer. There are nodes for the input $\boldsymbol{x}^{(i)}$, the hidden layer $\boldsymbol{z}$, the predicted output $\hat{\boldsymbol{y}}$, and the parameters Θ. During training, there is also a node for the ground truth label $y^{(i)}$ and the loss $\ell^{(i)}$. The predicted output $\hat{\boldsymbol{y}}$ is one of the parents of the loss (the other is the label $y^{(i)}$); *its* parents include Θ and $\boldsymbol{z}$, and so on.

Computation graphs include three types of nodes:

Variables. In the feedforward network of Figure 3.3, the variables include the inputs $\boldsymbol{x}$, the hidden nodes $\boldsymbol{z}$, the outputs $\boldsymbol{y}$, and the loss function. Inputs are variables that do not have parents. Backpropagation computes the gradients with respect to all

Algorithm 6 General backpropagation algorithm. In the computation graph G, every node contains a function f_t and a set of parent nodes $\boldsymbol{\pi}_t$; the inputs to the graph are $\boldsymbol{x}^{(i)}$.

procedure BACKPROP($G = \{f_t, \boldsymbol{\pi}_t\}_{t=1}^T\}, \boldsymbol{x}^{(i)}$)
  $v_{t(n)} \leftarrow x_n^{(i)}$ for all n and associated computation nodes $t(n)$.
  for $t \in$ TOPOLOGICALSORT(G) **do** ▷ Forward pass: compute value at each node
    if $|\boldsymbol{\pi}_t| > 0$ **then**
      $v_t \leftarrow f_t(v_{\pi_{t,1}}, v_{\pi_{t,2}}, \ldots, v_{\pi_{t,N_t}})$
  $g_{\text{objective}} = 1$ ▷ Backward pass: compute gradients at each node
  for $t \in$ REVERSE(TOPOLOGICALSORT(G)) **do**
    $g_t \leftarrow \sum_{t':t \in \boldsymbol{\pi}_{t'}} g_{t'} \times \nabla_{v_t} v_{t'}$ ▷ Sum over all t' that are children of t, propagating the gradient $g_{t'}$, scaled by the local gradient $\nabla_{v_t} v_{t'}$
  return $\{g_1, g_2, \ldots, g_T\}$

variables except the inputs, and propagates these gradients backwards to the parameters.

Parameters. In a feedforward network, the parameters include the weights and offsets. In Figure 3.3, the parameters are summarized in the node Θ, but we could have separate nodes for $\boldsymbol{\Theta}^{(x \rightarrow z)}$, $\boldsymbol{\Theta}^{(z \rightarrow y)}$, and any offset parameters. Parameter nodes do not have parents; they are not computed from other nodes, but rather, are learned by gradient descent.

Loss. The loss $\ell^{(i)}$ is the quantity that is to be minimized during training. The node representing the loss in the computation graph is not the parent of any other node; its parents are typically the predicted label $\hat{\boldsymbol{y}}$ and the true label $\boldsymbol{y}^{(i)}$. Backpropagation begins by computing the gradient of the loss, and then propagating this gradient backwards to its immediate parents.

If the computation graph is a directed acyclic graph, then it is possible to order the nodes with a topological sort, so that if node t is a parent of node t', then $t < t'$. This means that the values $\{v_t\}_{t=1}^T$ can be computed in a single forward pass. The topological sort is reversed when computing gradients: each gradient g_t is computed from the gradients of the children of t, implementing the chain rule of differentiation. The general backpropagation algorithm for computation graphs is shown in Algorithm 6.

While the gradients with respect to each parameter may be complex, they are composed of products of simple parts. For many networks, all gradients can be computed through **automatic differentiation**. This means that you need only specify the feedforward computation, and the gradients necessary for learning can be obtained automatically. There are many software libraries that perform automatic differentiation on compu-

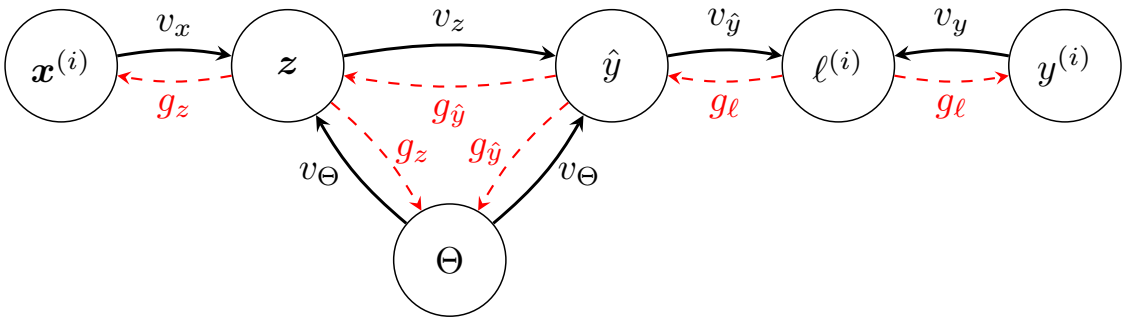

Figure 3.3: A computation graph for the feedforward neural network shown in Figure 3.1.

tation graphs, such as TORCH (Collobert et al., 2011), TENSORFLOW (Abadi et al., 2016), and DYNET (Neubig et al., 2017). One important distinction between these libraries is whether they support **dynamic computation graphs**, in which the structure of the computation graph varies across instances. Static computation graphs are compiled in advance, and can be applied to fixed-dimensional data, such as bag-of-words vectors. In many natural language processing problems, each input has a distinct structure, requiring a unique computation graph. A simple case occurs in recurrent neural network language models (see chapter 6), in which there is one node for each word in a sentence. More complex cases include recursive neural networks (see chapter 14), in which the network is a tree structure matching the syntactic organization of the input.

3.3.2 Regularization and dropout

In linear classification, overfitting was addressed by augmenting the objective with a regularization term, $\lambda||\boldsymbol{\theta}||_2^2$. This same approach can be applied to feedforward neural networks, penalizing each matrix of weights:

$$L = \sum_{i=1}^{N} \ell^{(i)} + \lambda_{z\rightarrow y}||\boldsymbol{\Theta}^{(z\rightarrow y)}||_F^2 + \lambda_{x\rightarrow z}||\boldsymbol{\Theta}^{(x\rightarrow z)}||_F^2, \quad [3.35]$$

where $||\boldsymbol{\Theta}||_F^2 = \sum_{i,j} \theta_{i,j}^2$ is the squared **Frobenius norm**, which generalizes the L_2 norm to matrices. The bias parameters $\boldsymbol{b}$ are not regularized, as they do not contribute to the sensitivity of the classifier to the inputs. In gradient-based optimization, the practical effect of Frobenius norm regularization is that the weights "decay" towards zero at each update, motivating the alternative name **weight decay**.

Another approach to controlling model complexity is **dropout**, which involves randomly setting some computation nodes to zero during training (Srivastava et al., 2014). For example, in the feedforward network, on each training instance, with probability ρ we

set each input x_n and each hidden layer node z_k to zero. Srivastava et al. (2014) recommend $\rho = 0.5$ for hidden units, and $\rho = 0.2$ for input units. Dropout is also incorporated in the gradient computation, so if node z_k is dropped, then none of the weights $\boldsymbol{\theta}_k^{(x \rightarrow z)}$ will be updated for this instance. Dropout prevents the network from learning to depend too much on any one feature or hidden node, and prevents **feature co-adaptation**, in which a hidden unit is only useful in combination with one or more other hidden units. Dropout is a special case of **feature noising**, which can also involve adding Gaussian noise to inputs or hidden units (Holmstrom and Koistinen, 1992). Wager et al. (2013) show that dropout is approximately equivalent to "adaptive" L_2 regularization, with a separate regularization penalty for each feature.

3.3.3 *Learning theory

Chapter 2 emphasized the importance of **convexity** for learning: for convex objectives, the global optimum can be found efficiently. The negative log-likelihood and hinge loss are convex functions of the parameters of the output layer. However, the output of a feedforward network is generally not a convex function of the parameters of the input layer, $\boldsymbol{\Theta}^{(x \rightarrow z)}$. Feedforward networks can be viewed as function composition, where each layer is a function that is applied to the output of the previous layer. Convexity is generally not preserved in the composition of two convex functions — and furthermore, "squashing" activation functions like tanh and sigmoid are not convex.

The non-convexity of hidden layer neural networks can also be seen by permuting the elements of the hidden layer, from $\boldsymbol{z} = [z_1, z_2, \ldots, z_{K_z}]$ to $\tilde{\boldsymbol{z}} = [z_{\pi(1)}, z_{\pi(2)}, \ldots, z_{\pi(K_z)}]$. This corresponds to applying π to the rows of $\boldsymbol{\Theta}^{(x \rightarrow z)}$ and the columns of $\boldsymbol{\Theta}^{(z \rightarrow y)}$, resulting in permuted parameter matrices $\boldsymbol{\Theta}_\pi^{(x \rightarrow z)}$ and $\boldsymbol{\Theta}_\pi^{(z \rightarrow y)}$. As long as this permutation is applied consistently, the loss will be identical, $L(\boldsymbol{\Theta}) = L(\boldsymbol{\Theta}_\pi)$: it is *invariant* to this permutation. However, the loss of the linear combination $L(\alpha\boldsymbol{\Theta} + (1-\alpha)\boldsymbol{\Theta}_\pi)$ will generally not be identical to the loss under $\boldsymbol{\Theta}$ or its permutations. If $L(\boldsymbol{\Theta})$ is better than the loss at any points in the immediate vicinity, and if $L(\boldsymbol{\Theta}) = L(\boldsymbol{\Theta}_\pi)$, then the loss function does not satisfy the definition of convexity (see § 2.4). One of the exercises asks you to prove this more rigorously.

In practice, the existence of multiple optima is not necessary problematic, if all such optima are permutations of the sort described in the previous paragraph. In contrast, "bad" local optima are better than their neighbors, but much worse than the global optimum. Fortunately, in large feedforward neural networks, most local optima are nearly as good as the global optimum (Choromanska et al., 2015). More generally, a **critical point** is one at which the gradient is zero. Critical points may be local optima, but they may also be **saddle points**, which are local minima in some directions, but local *maxima* in other directions. For example, the equation $x_1^2 - x_2^2$ has a saddle point at $\boldsymbol{x} = (0, 0)$. In large networks, the overwhelming majority of critical points are saddle points, rather

than local minima or maxima (Dauphin et al., 2014). Saddle points can pose problems for gradient-based optimization, since learning will slow to a crawl as the gradient goes to zero. However, the noise introduced by stochastic gradient descent, and by feature noising techniques such as dropout, can help online optimization to escape saddle points and find high-quality optima (Ge et al., 2015). Other techniques address saddle points directly, using local reconstructions of the Hessian matrix (Dauphin et al., 2014) or higher-order derivatives (Anandkumar and Ge, 2016).

Another theoretical puzzle about neural networks is how they are able to **generalize** to unseen data. Given enough parameters, a two-layer feedforward network can "memorize" its training data, attaining perfect accuracy on any training set. A particularly salient demonstration was provided by Zhang et al. (2017), who showed that neural networks can learn to perfectly classify a training set of images, even when the labels are replaced with random values! Of course, this network attains only chance accuracy when applied to heldout data. The concern is that when such a powerful learner is applied to real training data, it may learn a pathological classification function, which exploits irrelevant details of the training data and fails to generalize. Yet this extreme **overfitting** is rarely encountered in practice, and can usually be prevented by regularization, dropout, and early stopping (see § 3.3.4). Recent papers have derived generalization guarantees for specific classes of neural networks (e.g., Kawaguchi et al., 2017; Brutzkus et al., 2018), but theoretical work in this area is ongoing.

3.3.4 Tricks

Getting neural networks to work sometimes requires heuristic "tricks" (Bottou, 2012; Goodfellow et al., 2016; Goldberg, 2017b). This section presents some tricks that are especially important.

Initialization Initialization is not especially important for linear classifiers, since convexity ensures that the global optimum can usually be found quickly. But for multilayer neural networks, it is helpful to have a good starting point. One reason is that if the magnitude of the initial weights is too large, a sigmoid or tanh nonlinearity will be saturated, leading to a small gradient, and slow learning. Large gradients can cause training to diverge, with the parameters taking increasingly extreme values until reaching the limits of the floating point representation.

Initialization can help avoid these problems by ensuring that the variance over the initial gradients is constant and bounded throughout the network. For networks with tanh activation functions, this can be achieved by sampling the initial weights from the

following uniform distribution (Glorot and Bengio, 2010),

$$\theta_{i,j} \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_{\text{in}}(n)+d_{\text{out}}(n)}}, \frac{\sqrt{6}}{\sqrt{d_{\text{in}}(n)+d_{\text{out}}(n)}}\right], \tag{3.36}$$

$$\tag{3.37}$$

For the weights leading to a ReLU activation function, He et al. (2015) use similar argumentation to justify sampling from a zero-mean Gaussian distribution,

$$\theta_{i,j} \sim N(0, \sqrt{2/d_{\text{in}}(n)}) \tag{3.38}$$

Rather than initializing the weights independently, it can be beneficial to initialize each layer jointly as an **orthonormal matrix**, ensuring that $\boldsymbol{\Theta}^\top\boldsymbol{\Theta} = \mathbb{I}$ (Saxe et al., 2014). Orthonormal matrices preserve the norm of the input, so that $||\boldsymbol{\Theta}\boldsymbol{x}|| = ||\boldsymbol{x}||$, which prevents the gradients from exploding or vanishing. Orthogonality ensures that the hidden units are uncorrelated, so that they correspond to different features of the input. Orthonormal initialization can be performed by applying **singular value decomposition** to a matrix of values sampled from a standard normal distribution:

$$a_{i,j} \sim N(0,1) \tag{3.39}$$

$$\mathbf{A} = \{a_{i,j}\}_{i=1,j=1}^{d_{\text{in}}(j),d_{\text{out}}(j)} \tag{3.40}$$

$$\mathbf{U}, \mathbf{S}, \mathbf{V}^\top = \text{SVD}(\mathbf{A}) \tag{3.41}$$

$$\boldsymbol{\Theta}^{(j)} \leftarrow \mathbf{U}. \tag{3.42}$$

The matrix $\mathbf{U}$ contains the **singular vectors** of $\mathbf{A}$, and is guaranteed to be orthonormal. For more on singular value decomposition, see chapter 14.

Even with careful initialization, there can still be significant variance in the final results. It can be useful to make multiple training runs, and select the one with the best performance on a heldout development set.

Clipping and normalization Learning can be sensitive to the magnitude of the gradient: too large, and learning can diverge, with successive updates thrashing between increasingly extreme values; too small, and learning can grind to a halt. Several heuristics have been proposed to address this issue.

- In **gradient clipping** (Pascanu et al., 2013), an upper limit is placed on the norm of the gradient, and the gradient is rescaled when this limit is exceeded,

$$\text{CLIP}(\tilde{\boldsymbol{g}}) = \begin{cases} \boldsymbol{g} & ||\hat{\boldsymbol{g}}|| < \tau \\ \frac{\tau}{||\boldsymbol{g}||}\boldsymbol{g} & \text{otherwise.} \end{cases} \tag{3.43}$$

- In **batch normalization** (Ioffe and Szegedy, 2015), the inputs to each computation node are recentered by their mean and variance across all of the instances in the minibatch $\mathcal{B}$ (see § 2.6.2). For example, in a feedforward network with one hidden layer, batch normalization would tranform the inputs to the hidden layer as follows:

$$\boldsymbol{\mu}^{(\mathcal{B})} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \boldsymbol{x}^{(i)} \quad [3.44]$$

$$\boldsymbol{s}^{(\mathcal{B})} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(\mathcal{B})})^2 \quad [3.45]$$

$$\overline{\boldsymbol{x}}^{(i)} = (\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(\mathcal{B})}) / \sqrt{\boldsymbol{s}^{(\mathcal{B})}}. \quad [3.46]$$

  Empirically, this speeds convergence of deep architectures. One explanation is that it helps to correct for changes in the distribution of activations during training.

- In **layer normalization** (Ba et al., 2016), the inputs to each nonlinear activation function are recentered across the layer:

$$\boldsymbol{a} = \boldsymbol{\Theta}^{(x \rightarrow z)} \boldsymbol{x} \quad [3.47]$$

$$\mu = \frac{1}{K_z} \sum_{k=1}^{K_z} a_k \quad [3.48]$$

$$s = \frac{1}{K_z} \sum_{k=1}^{K_z} (a_k - \mu)^2 \quad [3.49]$$

$$\boldsymbol{z} = (\boldsymbol{a} - \mu) / \sqrt{s}. \quad [3.50]$$

  Layer normalization has similar motivations to batch normalization, but it can be applied across a wider range of architectures and training conditions.

Online optimization There is a cottage industry of online optimization algorithms that attempt to improve on stochastic gradient descent. **AdaGrad** was reviewed in § 2.6.2; its main innovation is to set adaptive learning rates for each parameter by storing the sum of squared gradients. Rather than using the sum over the entire training history, we can keep a running estimate,

$$v_j^{(t)} = \beta v_j^{(t-1)} + (1 - \beta) g_{t,j}^2, \quad [3.51]$$

where $g_{t,j}$ is the gradient with respect to parameter j at time t, and $\beta \in [0, 1]$. This term places more emphasis on recent gradients, and is employed in the AdaDelta (Zeiler, 2012) and Adam (Kingma and Ba, 2014) optimizers. Online optimization and its theoretical background are reviewed by Bottou et al. (2016). **Early stopping**, mentioned in § 2.3.2, can help to avoid overfitting by terminating training after reaching a plateau in the performance on a heldout validation set.

Practical advice The bag of tricks for training neural networks continues to grow, and it is likely that there will be several new ones by the time you read this. Today, it is standard practice to use gradient clipping, early stopping, and a sensible initialization of parameters to small random values. More bells and whistles can be added as solutions to specific problems — for example, if it is difficult to find a good learning rate for stochastic gradient descent, then it may help to try a fancier optimizer with an adaptive learning rate. Alternatively, if a method such as layer normalization is used by related models in the research literature, you should probably consider it, especially if you are having trouble matching published results. As with linear classifiers, it is important to evaluate these decisions on a held-out development set, and not on the test set that will be used to provide the final measure of the model's performance (see § 2.2.5).

3.4 Convolutional neural networks

A basic weakness of the bag-of-words model is its inability to account for the ways in which words combine to create meaning, including even simple reversals such as *not pleasant*, *hardly a generous offer*, and *I wouldn't mind missing the flight*. Computer vision faces the related challenge of identifying the semantics of images from pixel features that are uninformative in isolation. An earlier generation of computer vision research focused on designing *filters* to aggregate local pixel-level features into more meaningful representations, such as edges and corners (e.g., Canny, 1987). Similarly, earlier NLP research attempted to capture multiword linguistic phenomena by hand-designed lexical patterns (Hobbs et al., 1997). In both cases, the output of the filters and patterns could then act as base features in a linear classifier. But rather than designing these feature extractors by hand, a better approach is to learn them, using the magic of backpropagation. This is the idea behind **convolutional neural networks**.

Following § 3.2.4, define the base layer of a neural network as,

$$\mathbf{X}^{(0)} = \mathbf{\Theta}^{(x \rightarrow z)}[\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \ldots, \boldsymbol{e}_{w_M}], \tag{3.52}$$

where $\boldsymbol{e}_{w_m}$ is a column vector of zeros, with a 1 at position w_m. The base layer has dimension $\mathbf{X}^{(0)} \in \mathbb{R}^{K_e \times M}$, where K_e is the size of the word embeddings. To merge information across adjacent words, we *convolve* $\mathbf{X}^{(0)}$ with a set of filter matrices $\mathbf{C}^{(k)} \in \mathbb{R}^{K_e \times h}$. Convolution is indicated by the symbol $*$, and is defined,

$$\mathbf{X}^{(1)} = f(\boldsymbol{b} + \mathbf{C} * \mathbf{X}^{(0)}) \quad \Longrightarrow \quad x^{(1)}_{k,m} = f\left(b_k + \sum_{k'=1}^{K_e} \sum_{n=1}^{h} c^{(k)}_{k',n} \times x^{(0)}_{k',m+n-1}\right), \tag{3.53}$$

where f is an activation function such as tanh or ReLU, and $\boldsymbol{b}$ is a vector of offsets. The convolution operation slides the matrix $\mathbf{C}^{(k)}$ across the columns of $\mathbf{X}^{(0)}$. At each position m, we compute the elementwise product $\mathbf{C}^{(k)} \odot \mathbf{X}^{(0)}_{m:m+h-1}$, and take the sum.

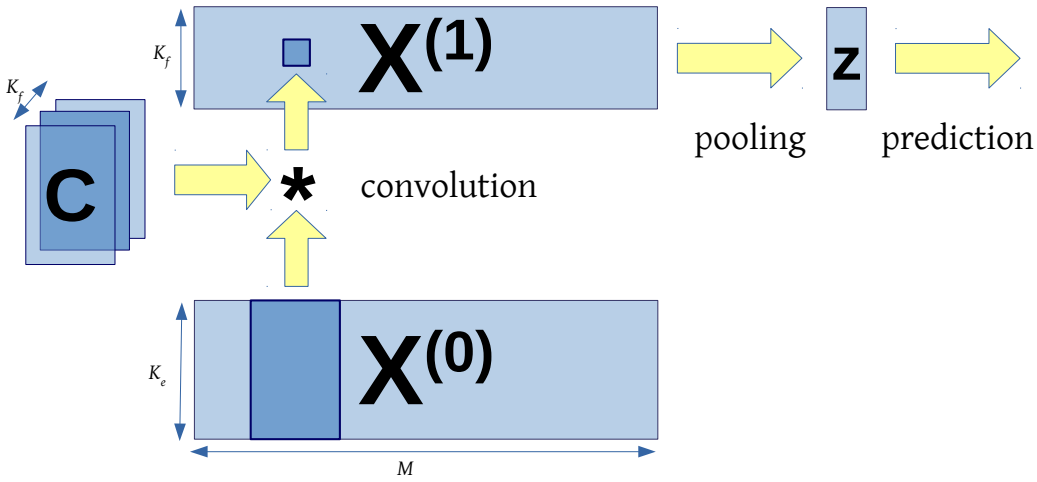

Figure 3.4: A convolutional neural network for text classification

A simple filter might compute a weighted average over nearby words,

$$\mathbf{C}^{(k)} = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.5 & 1 & 0.5 \\ \dots & \dots & \dots \\ 0.5 & 1 & 0.5 \end{bmatrix}, \quad [3.54]$$

thereby representing trigram units like *not so unpleasant*. In **one-dimensional convolution**, each filter matrix $\mathbf{C}^{(k)}$ is constrained to have non-zero values only at row k (Kalchbrenner et al., 2014). This means that each dimension of the word embedding is processed by a separate filter, and it implies that $K_f = K_e$.

To deal with the beginning and end of the input, the base matrix $\mathbf{X}^{(0)}$ may be padded with h column vectors of zeros at the beginning and end; this is known as **wide convolution**. If padding is not applied, then the output from each layer will be $h - 1$ units smaller than the input; this is known as **narrow convolution**. The filter matrices need not have identical filter widths, so more generally we could write h_k to indicate to width of filter $\mathbf{C}^{(k)}$. As suggested by the notation $\mathbf{X}^{(0)}$, multiple layers of convolution may be applied, so that $\mathbf{X}^{(d)}$ is the input to $\mathbf{X}^{(d+1)}$.

After D convolutional layers, we obtain a matrix representation of the document $\mathbf{X}^{(D)} \in \mathbb{R}^{K_z \times M}$. If the instances have variable lengths, it is necessary to aggregate over all M word positions to obtain a fixed-length representation. This can be done by a **pooling** operation,

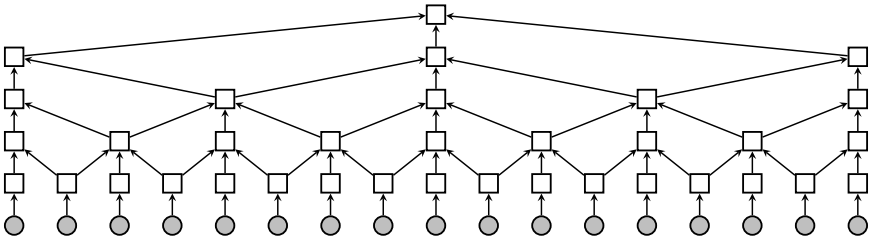

Figure 3.5: A dilated convolutional neural network captures progressively larger context through recursive application of the convolutional operator

such as max-pooling (Collobert et al., 2011) or average-pooling,

$$\boldsymbol{z} = \text{MaxPool}(\mathbf{X}^{(D)}) \quad \Longrightarrow \quad z_k = \max\left(x_{k,1}^{(D)}, x_{k,2}^{(D)}, \ldots x_{k,M}^{(D)}\right) \tag{3.55}$$

$$\boldsymbol{z} = \text{AvgPool}(\mathbf{X}^{(D)}) \quad \Longrightarrow \quad z_k = \frac{1}{M}\sum_{m=1}^{M} x_{k,m}^{(D)}. \tag{3.56}$$

The vector $\boldsymbol{z}$ can now act as a layer in a feedforward network, culminating in a prediction $\hat{y}$ and a loss $\ell^{(i)}$. The setup is shown in Figure 3.4.

Just as in feedforward networks, the parameters $(\mathbf{C}^{(k)}, \boldsymbol{b}, \Theta)$ can be learned by backpropagating from the classification loss. This requires backpropagating through the max-pooling operation, which is a discontinuous function of the input. But because we need only a local gradient, backpropagation flows only through the argmax m:

$$\frac{\partial z_k}{\partial x_{k,m}^{(D)}} = \begin{cases} 1, & x_{k,m}^{(D)} = \max\left(x_{k,1}^{(D)}, x_{k,2}^{(D)}, \ldots x_{k,M}^{(D)}\right) \\ 0, & \text{otherwise.} \end{cases} \tag{3.57}$$

The computer vision literature has produced a huge variety of convolutional architectures, and many of these innovations can be applied to text data. One avenue for improvement is more complex pooling operations, such as k-max pooling (Kalchbrenner et al., 2014), which returns a matrix of the k largest values for each filter. Another innovation is the use of **dilated convolution** to build multiscale representations (Yu and Koltun, 2016). At each layer, the convolutional operator applied in *strides*, skipping ahead by s steps after each feature. As we move up the hierarchy, each layer is s times smaller than the layer below it, effectively summarizing the input (Kalchbrenner et al., 2016; Strubell et al., 2017). This idea is shown in Figure 3.5. Multi-layer convolutional networks can also be augmented with "shortcut" connections, as in the residual network from § 3.2.2 (Johnson and Zhang, 2017).

Additional resources

The deep learning textbook by Goodfellow et al. (2016) covers many of the topics in this chapter in more detail. For a comprehensive review of neural networks in natural language processing, see Goldberg (2017b). A seminal work on deep learning in natural language processing is the aggressively titled "Natural Language Processing (Almost) from Scratch", which uses convolutional neural networks to perform a range of language processing tasks (Collobert et al., 2011), although there is earlier work (e.g., Henderson, 2004). This chapter focuses on feedforward and convolutional neural networks, but recurrent neural networks are one of the most important deep learning architectures for natural language processing. They are covered extensively in chapters 6 and 7.

The role of deep learning in natural language processing research has caused angst in some parts of the natural language processing research community (e.g., Goldberg, 2017a), especially as some of the more zealous deep learning advocates have argued that end-to-end learning from "raw" text can eliminate the need for linguistic constructs such as sentences, phrases, and even words (Zhang et al., 2015, originally titled "Text understanding from scratch"). These developments were surveyed by Manning (2015). While reports of the demise of linguistics in natural language processing remain controversial at best, deep learning and backpropagation have become ubiquitous in both research and applications.

Exercises

1. Figure 3.3 shows the computation graph for a feedforward neural network with one layer.

   a) Update the computation graph to include a residual connection between $\boldsymbol{x}$ and $\boldsymbol{z}$.

   b) Update the computation graph to include a highway connection between $\boldsymbol{x}$ and $\boldsymbol{z}$.

2. Prove that the softmax and sigmoid functions are equivalent when the number of possible labels is two. Specifically, for any $\boldsymbol{\Theta}^{(z \rightarrow y)}$ (omitting the offset $\boldsymbol{b}$ for simplicity), show how to construct a vector of weights $\boldsymbol{\theta}$ such that,

$$\text{SoftMax}(\boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z})[0] = \sigma(\boldsymbol{\theta} \cdot \boldsymbol{z}). \quad [3.58]$$

3. Convolutional neural networks often aggregate across words by using max-**pooling** (Equation 3.55 in § 3.4). A potential concern is that there is zero gradient with respect to the parts of the input that are not included in the maximum. The following

questions consider the gradient with respect to an element of the input, $x_{m,k}^{(0)}$, and they assume that all parameters are independently distributed.

a) First consider a minimal network, with $\boldsymbol{z} = \text{MaxPool}(\mathbf{X}^{(0)})$. What is the probability that the gradient $\frac{\partial \ell}{\partial x_{m,k}^{(0)}}$ is non-zero?

b) Now consider a two-level network, with $\mathbf{X}^{(1)} = f(\boldsymbol{b} + \mathbf{C} * \mathbf{X}^{(0)})$. Express the probability that the gradient $\frac{\partial \ell}{\partial x_{m,k}^{(0)}}$ is non-zero, in terms of the input length M, the filter size n, and the number of filters K_f.

c) Using a calculator, work out the probability for the case $M = 128, n = 4, K_f = 32$.

d) Now consider a three-level network, $\mathbf{X}^{(2)} = f(\boldsymbol{b} + \mathbf{C} * \mathbf{X}^{(1)})$. Give the general equation for the probability that $\frac{\partial \ell}{\partial x_{m,k}^{(0)}}$ is non-zero, and compute the numerical probability for the scenario in the previous part, assuming $K_f = 32$ and $n = 4$ at both levels.

4. Design a feedforward network to compute the XOR function:

$$f(x_1, x_2) = \begin{cases} -1, & x_1 = 1, x_2 = 1 \\ 1, & x_1 = 1, x_2 = 0 \\ 1, & x_1 = 0, x_2 = 1 \\ -1, & x_1 = 0, x_2 = 0 \end{cases}. \quad [3.59]$$

Your network should have a single output node which uses the Sign activation function, $f(x) = \begin{cases} 1, & x > 0 \\ -1, & x \leq 0. \end{cases}$. Use a single hidden layer, with ReLU activation functions. Describe all weights and offsets.

5. Consider the same network as above (with ReLU activations for the hidden layer), with an arbitrary differentiable loss function $\ell(y^{(i)}, \tilde{y})$, where $\tilde{y}$ is the activation of the output node. Suppose all weights and offsets are initialized to zero. Show that gradient descent will not learn the desired function from this initialization.

6. The simplest solution to the previous problem relies on the use of the ReLU activation function at the hidden layer. Now consider a network with arbitrary activations on the hidden layer. Show that if the initial weights are any uniform constant, then gradient descent will not learn the desired function from this initialization.

7. Consider a network in which: the base features are all binary, $\boldsymbol{x} \in \{0, 1\}^M$; the hidden layer activation function is sigmoid, $z_k = \sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x})$; and the initial weights are sampled independently from a standard normal distribution, $\theta_{j,k} \sim N(0, 1)$.

- Show how the probability of a small initial gradient on any weight, $\frac{\partial z_k}{\partial \boldsymbol{\theta}_{j,k}} < \alpha$, depends on the size of the input M. **Hint**: use the lower bound,

$$\Pr(\sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x}) \times (1 - \sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x})) < \alpha) \quad \geq \quad 2\Pr(\sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x}) < \alpha), \tag{3.60}$$

  and relate this probability to the variance $V[\boldsymbol{\theta}_k \cdot \boldsymbol{x}]$.
- Design an alternative initialization that removes this dependence.

8. The ReLU activation function can lead to "dead neurons", which can never be activated on any input. Consider the following two-layer feedforward network with a scalar output y:

$$z_i = \text{ReLU}(\boldsymbol{\theta}_i^{(x \to z)} \cdot \boldsymbol{x} + b_i) \tag{3.61}$$

$$y = \boldsymbol{\theta}^{(z \to y)} \cdot \boldsymbol{z}. \tag{3.62}$$

   Suppose that the input is a binary vector of observations, $\boldsymbol{x} \in \{0, 1\}^D$.

   a) Under what condition is node z_i "dead"? Your answer should be expressed in terms of the parameters $\boldsymbol{\theta}_i^{(x \to z)}$ and b_i.

   b) Suppose that the gradient of the loss on a given instance is $\frac{\partial \ell}{\partial y} = 1$. Derive the gradients $\frac{\partial \ell}{\partial b_i}$ and $\frac{\partial \ell}{\partial \theta_{j,i}^{(x \to z)}}$ for such an instance.

   c) Using your answers to the previous two parts, explain why a dead neuron can never be brought back to life during gradient-based learning.

9. Suppose that the parameters $\Theta = \{\Theta^{(x \to z)}, \Theta(z \to y), \boldsymbol{b}\}$ are a local optimum of a feedforward network in the following sense: there exists some $\epsilon > 0$ such that,

$$\Big(||\tilde{\Theta}^{(x \to z)} - \Theta^{(x \to z)}||_F^2 + ||\tilde{\Theta}^{(z \to y)} - \Theta^{(z \to y)}||_F^2 + ||\tilde{\boldsymbol{b}} - \boldsymbol{b}||_2^2 < \epsilon\Big)$$
$$\Rightarrow \Big(L(\tilde{\Theta}) > L(\Theta)\Big) \tag{3.63}$$

   Define the function π as a permutation on the hidden units, as described in § 3.3.3, so that for any Θ, $L(\Theta) = L(\Theta_\pi)$. Prove that if a feedforward network has a local optimum in the sense of Equation 3.63, then its loss is not a convex function of the parameters Θ, using the definition of convexity from § 2.4

10. Consider a network with a single hidden layer, and a single output,

$$y = \boldsymbol{\theta}^{(z \to y)} \cdot g(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x}). \tag{3.64}$$

    Assume that g is the ReLU function. Show that for any matrix of weights $\boldsymbol{\Theta}^{(x \to z)}$, it is permissible to rescale each row to have a norm of one, because an identical output can be obtained by finding a corresponding rescaling of $\boldsymbol{\theta}^{(z \to y)}$.