Taylor Arnold, Michael Kane, and Bryan Lewis

A Computational Approach to Statistical Learning

Contents

Preface **ix**

1 Introduction **1**
1.1 Computational approach 1
1.2 Statistical learning 2
1.3 Example 3
1.4 Prerequisites 5
1.5 How to read this book 6
1.6 Supplementary materials 7
1.7 Formalisms and terminology 7
1.8 Exercises 9

2 Linear Models **11**
2.1 Introduction 11
2.2 Ordinary least squares 13
2.3 The normal equations 15
2.4 Solving least squares with the singular value decomposition . 17
2.5 Directly solving the linear system 19
2.6 (⋆) Solving linear models using the QR decomposition 22
2.7 (⋆) Sensitivity analysis 24
2.8 (⋆) Relationship between numerical and statistical error . . . 28
2.9 Implementation and notes 31
2.10 Application: Cancer incidence rates 32
2.11 Exercises 40

3 Ridge Regression and Principal Component Analysis **43**
3.1 Variance in OLS 43
3.2 Ridge regression 46
3.3 (⋆) A Bayesian perspective 53
3.4 Principal component analysis 56
3.5 Implementation and notes 63
3.6 Application: NYC taxicab data 65
3.7 Exercises 72

4 Linear Smoothers **75**
4.1 Non-Linearity . . . 75
4.2 Basis expansion . . . 76
4.3 Kernel regression . . . 81
4.4 Local regression . . . 85
4.5 Regression splines . . . 89
4.6 (⋆) Smoothing splines . . . 95
4.7 (⋆) B-splines . . . 100
4.8 Implementation and notes . . . 104
4.9 Application: U.S. census tract data . . . 105
4.10 Exercises . . . 120

5 Generalized Linear Models **123**
5.1 Classification with linear models . . . 123
5.2 Exponential families . . . 128
5.3 Iteratively reweighted GLMs . . . 131
5.4 (⋆) Numerical issues . . . 135
5.5 (⋆) Multi-Class regression . . . 138
5.6 Implementation and notes . . . 139
5.7 Application: Chicago crime prediction . . . 140
5.8 Exercises . . . 148

6 Additive Models **151**
6.1 Multivariate linear smoothers . . . 151
6.2 Curse of dimensionality . . . 155
6.3 Additive models . . . 158
6.4 (⋆) Additive models as linear models . . . 163
6.5 (⋆) Standard errors in additive models . . . 166
6.6 Implementation and notes . . . 170
6.7 Application: NYC flights data . . . 172
6.8 Exercises . . . 178

7 Penalized Regression Models **179**
7.1 Variable selection . . . 179
7.2 Penalized regression with the ℓ_0- and ℓ_1-norms . . . 180
7.3 Orthogonal data matrix . . . 182
7.4 Convex optimization and the elastic net . . . 186
7.5 Coordinate descent . . . 188
7.6 (⋆) Active set screening using the KKT conditions . . . 193
7.7 (⋆) The generalized elastic net model . . . 198
7.8 Implementation and notes . . . 200
7.9 Application: Amazon product reviews . . . 201
7.10 Exercises . . . 206

8 Neural Networks **207**
8.1 Dense neural network architecture 207
8.2 Stochastic gradient descent 211
8.3 Backward propagation of errors 213
8.4 Implementing backpropagation 216
8.5 Recognizing handwritten digits 224
8.6 (⋆) Improving SGD and regularization 226
8.7 (⋆) Classification with neural networks 232
8.8 (⋆) Convolutional neural networks 239
8.9 Implementation and notes 249
8.10 Application: Image classification with EMNIST 249
8.11 Exercises 259

9 Dimensionality Reduction **261**
9.1 Unsupervised learning 261
9.2 Kernel functions 262
9.3 Kernel principal component analysis 266
9.4 Spectral clustering 272
9.5 t-Distributed stochastic neighbor embedding (t-SNE) 277
9.6 Autoencoders 282
9.7 Implementation and notes 283
9.8 Application: Classifying and visualizing fashion MNIST . . . 284
9.9 Exercises 295

10 Computation in Practice **297**
10.1 Reference implementations 297
10.2 Sparse matrices 298
10.3 Sparse generalized linear models 304
10.4 Computation on row chunks 307
10.5 Feature hashing 311
10.6 Data quality issues 318
10.7 Implementation and notes 320
10.8 Application 321
10.9 Exercises 329

A Linear algebra and matrices **331**
A.1 Vector spaces 331
A.2 Matrices 333

B Floating Point Arithmetic and Numerical Computation **337**
B.1 Floating point arithmetic 337
B.2 Computational effort 340

Bibliography **343**

Index **359**

Preface

This book was written to supplement the existing literature in statistical learning and predictive modeling. It provides a novel treatment of the computational details underlying the application of predictive models to modern datasets. It grew out of lecture notes from several courses we have taught at the undergraduate and graduate level on linear models, convex optimization, statistical computing, and supervised learning.

The major distinguishing feature of our text is the inclusion of code snippets that give working implementations of common algorithms for estimating predictive models. These implementations are written in the R programming language using basic vector and matrix algebra routines. The goal is to demystify links between the formal specification of an estimator and its application to a specific set of data. Seeing the exact algorithm used makes it possible to play around with methods in an understandable way and experiment with how algorithms perform on simulated and real-world datasets. This *try and see* approach fits a common paradigm for learning programming concepts. The reference implementations also illustrate the run-time, degree of manual tuning, and memory requirements of each method. These factors are paramount in selecting the best methods in most data analysis applications.

In order to focus on computational aspects of statistical learning, we highlight models that can be understood as extensions of multivariate linear regression. Within this framework, we show how penalized regression, additive models, spectral clustering, and neural networks fit into a cohesive set of methods for the construction of predictive models built on core concepts from linear algebra. The general structure of our text follows that of the two popular texts *An Introduction to Statistical Learning* (ISL) [87] and *The Elements of Statistical Learning* (ESL) [60]. This makes our book a reference for traditional courses that use either of these as a main text. In contrast to both ISL and ESL, our text focuses on giving an in-depth analysis to a significantly smaller set of methods, making it more conducive to self-study as well as appropriate for second courses in linear models or statistical learning.

Each chapter, other than the first, includes a fully worked out application to a real-world dataset. In order to not distract from the main exposition, these are included as a final section to each chapter. There are also many end of chapter exercises, primarily of a computational nature, asking readers to extend the code snippets used within the chapter. Common tasks involve benchmarking performance, adding additional parameters to reference implementations, writing unit tests, and applying techniques to new datasets.

Audience

This book has been written for several audiences: advanced undergraduate and first-year graduate students studying statistical or machine learning from a wide-range of academic backgrounds (i.e., mathematics, statistics, computer science, engineering); students studying statistical computing with a focus on predictive modeling; and researchers looking to understand the algorithms behind common models in statistical learning. We are able to simultaneously write for several backgrounds by focusing primarily on how techniques can be understood within the language of vector calculus and linear algebra, with a minimal discussion of distributional and probabilistic arguments. Calculus and linear algebra are well-studied across the mathematical sciences and benefit from direct links to both the motivation and implementations of many common estimators in statistical learning. While a solid background in calculus-based statistics and probability is certainly helpful, it is not strictly required for following the main aspects of the text.

The text may also be used as a self-study reference for computer scientists and software engineers attempting to pivot towards data science and predictive modeling. The computational angle, in particular our presenting of many techniques as optimization problems, makes it fairly accessible to readers who have taken courses on algorithms or convex programming. Techniques from numerical analysis, algorithms, data structures, and optimization theory required for understanding the methods are covered within the text. This approach allows the text to serve as the primary reference for a course focused on numerical methods in statistics. The computational angle also makes it a good choice for statistical learning courses taught or cross-listed with engineering or computer science schools and departments.

Online references

All of the code and associated datasets included in this text are available for download on our website `https://comp-approach.com`.

Notes to instructors

This text assumes that readers have a strong background in matrix algebra and are familiar with basic concepts from statistics. At a minimum students should be familiar with the concepts of expectation, bias, and variance.

Readers should ideally also have some prior exposure to programming in R. Experience with Python or another scripting language can also suffice for understanding the implementations as pseudocode, but it will be difficult to complete many of the exercises.

It is assumed throughout later chapters that readers are familiar with the introductory material in Chapter 1 and the first four sections of Chapter 2; the amount of time spent covering this material is highly dependent on the prior exposure students have had to predictive modeling. Several chapters should also be read as pairs. That is, we assume that readers are familiar with the first prior to engaging with the second. These are:

- Chapter 2 (Linear Models) and Chapter 3 (Ridge Regression and PCA)
- Chapter 4 (Linear Smoothers) and Chapter 6 (Additive Models)
- Chapter 5 (Generalized Linear Models) and Chapter 7 (Penalized Regression Models)

Within each individual chapter, the material should be covered in the order in which it is presented, though most sections can be introduced quite briefly in the interest of time.

Other than these dependencies, the chapters can be re-arranged to fit the needs of a course. In a one-semester undergraduate course on statistical learning, for example, we cover Chapters 1, 2, 5, 7, and 9 in order and in full. Time permitting, we try to include topics in neural networks (Chapter 8) as final projects. When teaching a semester long course on linear models to a classroom with undergraduates and graduate students in statistics, we move straight through Chapters 1 to 6. For a statistical learning class aimed at graduate students in statistics, we have presented Chapter 1 in the form of review before jumping into Chapter 4 and proceeded to cover all of Chapters 6 through 10.

Completing some of the end-of-chapter exercises is an important part of understanding the material. Many of these are fairly involved, however, and we recommend letting students perfect their answers to a curated set of these rather than having them complete a minimally sufficient answer to a more exhaustive collection.

8

Neural Networks

8.1 Dense neural network architecture

Neural networks are a broad class of predictive models that have enjoyed considerably popularity over the past decade. Neural networks consist of a collection of objects, known as neurons, organized into an ordered set of layers. Directed connections pass signals between neurons in adjacent layers. Prediction proceeds by passing an observation X_i to the first layer; the output of the final layer gives the predicted value $\widehat{y}_i$. Training a neural network involves updating the parameters describing the connections in order to minimize some loss function over the training data. Typically, the number of neurons and their connections are held fixed during this process. The concepts behind neural networks have existed since the early 1940s with the work of Warren McCulloch and Walter Pitts [119]. The motivation for their work, as well as the method's name, drew from the way that neurons use activation potentials to send signals throughout the central nervous system. These ideas were later revised and implemented throughout the 1950s [55, 139]. Neural networks, however, failed to become a general purpose learning technique until the early-2000s, due to the fact that they require large datasets and extensive computing power.

Our goal in this chapter is to de-mystify neural networks by showing that they can be understood as a natural extension of linear models. Specifically, they can be described as a collection of inter-woven linear models. This shows how these techniques fit naturally into the sequence of other techniques we have studied to this point. Neural networks should not be understood as a completely separate approach to predictive modeling. Rather, they are an extension of the linear approaches we have studied applied to the problem of detecting non-linear interactions in high-dimensional data.

A good place to begin is the construction of a very simple neural network. Assume that we have data where the goal is to predict a scalar response y from a scalar input x. Consider applying two independent linear models to this dataset (for now, we will ignore exactly how the slopes and intercepts will be determined). This will yield two sets of predicted values for each input,

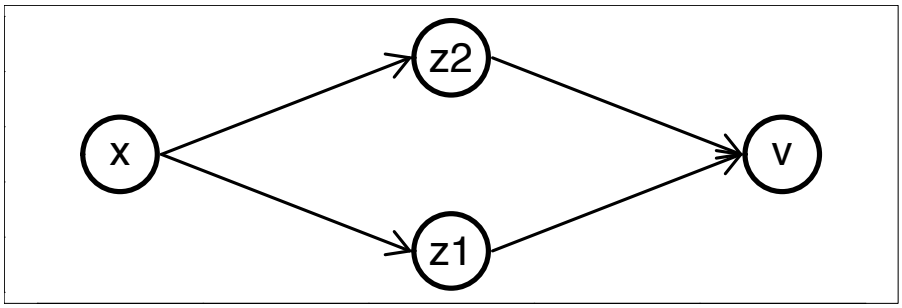

FIGURE 8.1: A diagram describing a simple neural network with one input (x), two hidden nodes (z_1 and z_2), and one output (v). Arrows describe linear relationships between the inputs and the outputs. Each node is also associated with an independent bias term, which is not pictured.

which we will denote by z_1 and z_2

$$z_1 = b_1 + x \cdot w_1 \tag{8.1}$$
$$z_2 = b_2 + x \cdot w_2 \tag{8.2}$$

This requires four parameters: two intercepts and two slopes. Notice that we can consider this as either a scalar equation for a single observation x_i or a vector equation for the entire vector of x.

Now, we will construct another linear model with the outputs z_j as inputs. We will name the output of this next regression model z_3:

$$z_3 = b_3 + z_1 \cdot u_1 + z_2 \cdot u_2. \tag{8.3}$$

Here, we can consider z_3 as being the predicted value $\widehat{y}$. For a visual description of the relationship between these variables, see Figure 8.1. What is the relationship between z_3 and x? In fact, this is nothing but a very complex way of describing a linear relationship between z_3 and x using 7 parameters instead of 2. We can see this by simplifying:

$$\begin{aligned} z_3 &= b_3 + z_1 \cdot u_1 + z_2 \cdot u_2 && (8.4)\\ &= b_3 + (b_1 + x \cdot w_1)u_1 + (b_2 + x \cdot w_2)u_2 && (8.5)\\ &= (b_3 + u_1 \cdot b_1 + u_2 \cdot b_2) + (w_1 \cdot u_1 + w_2 \cdot u_2) \cdot x && (8.6)\\ &= (\text{intercept}) + (\text{slope}) \cdot x && (8.7) \end{aligned}$$

Figure 8.2 shows a visual demonstration of how the first two regression models reduce to form a linear relationship between x and z_3.

What we have just described is a very simple neural network with four neurons: x, z_1, z_2, and z_3. As shown in Figure 8.1, these neurons are organized into three layers with each neuron in a given layer connected to every neuron in the following layer. The first layer, containing just x is known as the *input*

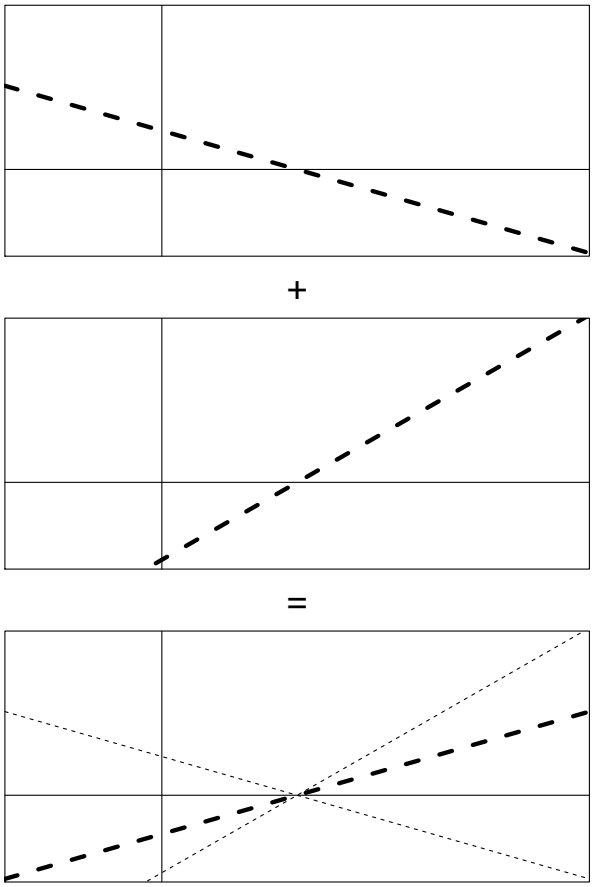

FIGURE 8.2: The top two plots show a linear function mapping the x-axis to the y-axis. Adding these two functions together yields a third linear relationship. This illustrates a simple neural network with two hidden nodes and no activation functions.

layer and the one containing just z_3 known as the *output layer*. Middle layers are called *hidden layers*. The neurons in the hidden layers are known as *hidden nodes*. As written, there is no obvious way of determining the unknown slopes and intercepts. Using the mean squared error as a loss function is reasonable but the model is over-parametrized; there are infinitely many ways to describe the same model. More importantly, there is seemingly no benefit to the neural network architecture here compared to that of a simple linear regression. Both describe the exact same set of relationships between x and y but the latter

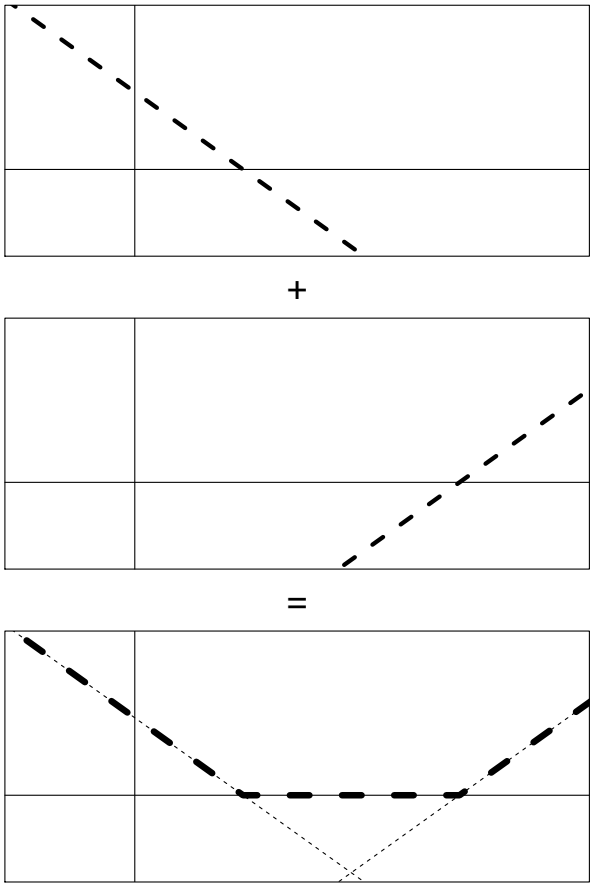

FIGURE 8.3: The top two plots show a linear function mapping the x-axis to the y-axis. The third diagram shows what happens when we add the first two lines together after applying the ReLU activation. In other words, we take the positive part of each function and add it to the other. The resulting function in the third diagram now gives a non-linear relationship between the x-axis and y-axis, illustrating how the use of activation functions is essential to the functioning of neural networks.

has a fast algorithm for computing the unknown parameters and usually has a unique solution.

One small change to our neural network will allow it to capture non-linear relationships between x and y. Instead of writing z_3 as a linear function of z_1

and z_2, we first transform the inputs by a function σ. Namely,

$$z_3 = b_3 + \sigma(z_1) \cdot u_1 + \sigma(z_2) \cdot u_2 \tag{8.8}$$

The function σ is not a learned function. It is just a fixed mapping that takes any real number and returns another real number. In the neural network literature it is called an *activation function*. A popular choice is known as a *rectified linear unit (ReLU)*, which pushes negative values to 0 but returns positive values unmodified:

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{8.9}$$

The addition of this activation function greatly increases the set of possible relationships between x and z_3 that are described by the neural network. Figure 8.3 visually shows how we can now create non-linear relationships by combining two linear functions after applying the ReLU function. The output now looks similar to a quadratic term. By including more hidden units into the model, and more input values into the first layer, we can create very interesting non-linear interactions in the output. In fact, as shown in the work of Barron, nearly any relationship can be approximated by such a model [16].

Allowing the neural network to describe non-linear relationships does come at a cost. We no longer have an analytic formula of finding the intercepts and slopes that minimizes the training loss. Iterative methods, such as those used for the elastic net in Chapter 7, must be used. Sections 8.2 and 8.3 derive the results needed to efficiently estimate the optimal weights in a neural network.

8.2 Stochastic gradient descent

Gradient descent is an iterative algorithm for finding the minimum value of a function f. It updates to a new value by the formula

$$w_{new} = w_{old} - \eta \cdot \nabla_w f(w_{old}), \tag{8.10}$$

for a fixed learning rate η. At each step it moves in the direction the function locally appears to be decreasing the fastest, at a rate proportional to how fast it seems to be decreasing. Gradient descent is a good algorithm choice for neural networks. Faster second-order methods involve the Hessian matrix, which requires the computation of a square matrix with dimensions equal to the number of unknown parameters. Even relatively small neural networks have tens of thousands of parameters making storage and computation of the Hessian matrix infeasible. Conversely, in Section 8.3 we will see that the gradient can be computed relatively quickly.

Neural networks generally need many thousands of iterations to converge to a reasonable minimizer of the loss function. With large datasets and models, while still feasible, gradient descent can become quite slow. Stochastic gradient descent (SGD) is a way of incrementally updating the weights in the model without needing to work with the entire dataset at each step. To understand the derivation of SGD, first consider updates in ordinary gradient descent:

$$\left(w^{(0)} - \eta \cdot \nabla_w f\right) \rightarrow w^{(1)} \tag{8.11}$$

Notice that for squared error loss (it is also true for most other loss functions), the loss can be written as a sum of component losses for each observation. The gradient, therefore, can also be written as a sum of terms over all of the data points.

$$f(w) = \sum_i (\widehat{y}_i(w) - y_i)^2 \tag{8.12}$$

$$= \sum_i f_i(w) \tag{8.13}$$

$$\nabla_w f = \sum_i \nabla_w f_i \tag{8.14}$$

This means that we could write gradient descent as a series of n steps over each of the training observations.

$$\left(w^{(0)} - (\eta/n) \cdot \nabla_{w^{(0)}} f_1\right) \rightarrow w^{(1)} \tag{8.15}$$

$$\left(w^{(1)} - (\eta/n) \cdot \nabla_{w^{(0)}} f_2\right) \rightarrow w^{(2)} \tag{8.16}$$

$$\vdots \tag{8.17}$$

$$\left(w^{(n-1)} - (\eta/n) \cdot \nabla_{w^{(0)}} f_n\right) \rightarrow w^{(n)} \tag{8.18}$$

$$\tag{8.19}$$

The output $w^{(n)}$ here is exactly equivalent to the $w^{(1)}$ from before.

The SGD algorithm actually does the updates in an iterative fashion, but makes one small modification. In each step it updates the gradient with respect to the new set of weights. Writing η' as η divided by the sample size, we can write this as:

$$\left(w^{(0)} - \eta' \cdot \nabla_{w^{(0)}} f_1\right) \rightarrow w^{(1)} \tag{8.20}$$

$$\left(w^{(1)} - \eta' \cdot \nabla_{w^{(1)}} f_2\right) \rightarrow w^{(2)} \tag{8.21}$$

$$\vdots \tag{8.22}$$

$$\left(w^{(n-1)} - \eta' \cdot \nabla_{w^{(n)}} f_n\right) \rightarrow w^{(n)} \tag{8.23}$$

In comparison to the standard gradient descent algorithm, the approach of SGD should seem reasonable. Why work with old weights in each step when we already know what direction the vector w is moving? Notice that SGD does not involve any stochastic features other than being sensitive to the ordering of the dataset. The name is an anachronism stemming from the original paper of Robbins and Monro which suggested randomly selecting the data point in each step instead of cycling through all of the training data in one go [138].

In the language of neural networks, one pass through the entire dataset is called an *epoch*. It results in as many iterations as there are observations in the training set. A common variant of SGD, and the most frequently used in the training of neural networks, modifies the procedure to something between pure gradient descent and pure SGD. Training data are grouped into mini-batches, typically of about 32–64 points, with gradients computed and parameters updated over the entire mini-batch. The benefits of this tweak are two-fold. First, it allows for faster computations as we can vectorize the gradient calculations of the entire mini-batch. Secondly, there is also empirical research suggesting that the mini-batch approach stops the SGD algorithm from getting stuck in saddle points [28, 63]. This latter feature is particularly important because the loss function in a neural network is known to exhibit a dense collection of saddle points [41].

8.3 Backward propagation of errors

In order to apply SGD to neural networks, we need to be able to compute the gradient of the loss function with respect to all of the trainable parameters in the model. For dense neural networks, the relationship between any parameter and the loss is given by the composition of linear functions, the activation function σ, and the chosen loss function. Given that activation and loss functions are generally well-behaved, in theory computing the gradient function should be straightforward for a given network. However, recall that we need to have thousands of iterations in the SGD algorithm and that even small neural networks typically have thousands of parameters. An algorithm for computing gradients as efficiently as possible is essential. We also want an algorithm that can be coded in a generic way that can then be used for models with an arbitrary number of layers and neurons in each layer.

The backwards propagation of errors, or just *backpropagation*, is the standard algorithm for computing gradients in a neural network. It is conceptually based on applying the chain rule to each layer of the network in reverse order. The first step consists in inserting an input x into the first layer and then propagating the outputs of each hidden layer through to the final output. All of the intermediate outputs are stored. Derivatives with respect to parameters in the last layer are calculated directly. Then, derivatives are calculated

showing how changing the parameters in any internal layer affect the output of just that layer. The chain rule is then used to compute the full gradient in terms of these intermediate quantities with one pass backwards through the network. The conceptual idea behind backpropagation can be applied to any hierarchical model described by a directed acyclic graph (DAG). Our discussion here will focus strictly on dense neural networks, with approaches to the more general problem delayed until Section 8.9.

We now proceed to derive the details of the backpropagation algorithm. One of the most important aspects in describing the algorithm is using a good notational system. Here, we will borrow from the language of neural networks in place of our terminology from linear models. Intercept terms become *biases*, slopes are described as *weights*, and we will focus on computing the gradient with respect to a single row of the data matrix X_i. Because we cannot spare any extra indices, X_i will be denoted by the lower case x. To use backpropagation with mini-batch SGD, simply compute the gradients for each sample in the mini-batch and add them together. Throughout, superscripts describe which layer in the neural network a variable refers to. In total, we assume that there are L layers, not including the input layer, which we denote as layer 0 (the input layer has no trainable weights so it is largely ignored in the backpropagation algorithm).

We will use the variable a to denote the outputs of a given layer in a neural network, known as the activations. As the name suggests, we will assume that the activate function as already been applied where applicable. The activations for the 0'th layer is just the input x itself,

$$a^0 = x. \tag{8.24}$$

For each value of l between 1 and L, the matrix $w^l_{j,k}$ gives the weights on the kth neuron in the $(l-1)$-st layer within the jth neuron in the lst layer. Likewise, the values b^l_j give the bias term for the jth neuron in the lst layer. The variable z is used to describe the output of each layer before applying the activation function. Put together, we then have the following relationship describing the a's, w's, b's, and z's:

$$z^l = w^l \circ a^{l-1} + b^l \tag{8.25}$$

$$a^l = \sigma(z^l) \tag{8.26}$$

These equations hold for all l from 1 up to (and including) L. Layer L is the output layer and therefore the activations a^L correspond to the predicted response:

$$a^L = \widehat{y}. \tag{8.27}$$

Finally, we also need a loss function f. As we are dealing with just a single observation at a time, we will write f as a function of only one input. For example, with mean squared error we would have:

$$f(y, a^L) = (y - a^L)^2 \tag{8.28}$$

The global loss function is assumed to be the sum over the losses of each individual observation. There is a great deal of terminology and notation here, but it is all needed in order to derive the backpropagation algorithm. You should make sure that all of the quantities make sense before proceeding.

The equations defining backpropagation center around derivatives with respect to the quantities z^l. As a starting point, notice that these derivatives give the gradient terms of the biases, b_j^l,

$$\frac{\partial f}{\partial b_j^l} = \frac{\partial f}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} \tag{8.29}$$

$$= \frac{\partial f}{\partial z_j^l} \cdot (1) \tag{8.30}$$

$$= \frac{\partial f}{\partial z_j^l}. \tag{8.31}$$

To get the gradient with respect to the weights, we need to weight the derivatives with respect to z by the activations in the prior layer,

$$\frac{\partial f}{\partial w_{jk}^l} = \frac{\partial f}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}^l} \tag{8.32}$$

$$= \frac{\partial f}{\partial z_j^l} \cdot a_k^{l-1}. \tag{8.33}$$

We see already why a forward pass through the data is required as a starting point; this forward pass gives us all the current activations a_k^{l-1}. At this point, we now have reduced the problem of computing the gradient to computing the derivatives of the responses z^l.

The derivatives in the Lth layer are straightforward to compute directly with the chain rule, taking advantage of the fact that a_j^L is a function of only z_j^L and no other activations in the Lth layer:

$$\frac{\partial f}{\partial z_j^L} = \sum_k \frac{\partial f}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L} \tag{8.34}$$

$$= \frac{\partial f}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \tag{8.35}$$

$$= \frac{\partial f}{\partial a_j^L} \cdot \sigma'(z_j^L) \tag{8.36}$$

The derivative of the loss function with respect to the activations in the last layer should be easy to compute since, as shown in Equation 8.28, the loss function is generally written directly in terms of the activations in the last layer.

The last and most involved step is to figure out how the derivatives with

respect to z^l can be written as a function of derivatives with respect to z^{l+1}. Notice that if $l \neq L$, then we can write z^{l+1} in terms of the responses in the prior layer,

$$z_k^{l+1} = \sum_m w_{km}^{l+1} a_m^l + b_k^{l+1} \tag{8.37}$$

$$= \sum_m w_{km}^{l+1} \sigma(z_m^l) + b_k^{l+1}. \tag{8.38}$$

From this, we can take the derivative of z_k^{l+1} with respect to z_j^l,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \tag{8.39}$$

Finally, putting this together in the chain rule gives

$$\frac{\partial f}{\partial z_j^l} = \sum_k \frac{\partial f}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{8.40}$$

$$= \sum_k \frac{\partial f}{\partial z_k^{l+1}} \cdot w_{kj}^{l+1} \sigma'(z_j^l). \tag{8.41}$$

We can simplify this using the Hadamard product $\odot$ (it computes element-wise products between matrices of the same size):

$$\frac{\partial f}{\partial z^l} = \left[(w^{l+1})^T \left(\frac{\partial f}{\partial z_j^{l+1}} \right) \right] \odot \sigma'(z^l) \tag{8.42}$$

And similarly for the last layer of the network,

$$\frac{\partial f}{\partial z^L} = \nabla_a f \odot \sigma'(z^L). \tag{8.43}$$

Putting these all together, backpropagation can be used to compute the gradient of f with respect to the b's and w's using the same order of magnitude number of steps as it takes to conduct forward propagation.

8.4 Implementing backpropagation

We now have all of the pieces required to train the weights in a neural network using SGD. For simplicity we will implement SGD without mini-batches (adding this feature is included as an exercise). We first make one minor change to the algorithm in Section 8.3. Notice that the current setup applies an activation function to the final layer of the network. If this activation is

a ReLU unit, this makes it impossible to predict negative values. Typically, when fitting a neural network to a continuous response we do not use an activation in the final layer. In other words, $z^L = a^L$. Equivalently, we can define the final σ to be the identity function. This just makes the second term on the right-hand side of Equation 8.43, $\sigma'(z^L)$, equal to 1. We will need this activation again when doing classification in Section 8.7.

The code for training and predicting with neural networks is best split into individual functions. Our first step is to define a function that returns the weights and biases of a network in a usable format. We will store these parameters in a list, with one element per layer. Each element is itself a list containing one element `w` (a matrix of weights, w^l) and one element `b` (a vector of biases, b^l). The function `casl_nn_make_weights` creates such a list, filled with randomly generated values from a normal distribution.

```
# Create list of weights to describe a dense neural network.
#
# Args:
#     sizes: A vector giving the size of each layer, including
#            the input and output layers.
#
# Returns:
#     A list containing initialized weights and biases.
casl_nn_make_weights <-
function(sizes)
{
  L <- length(sizes) - 1L
  weights <- vector("list", L)
  for (j in seq_len(L))
  {
    w <- matrix(rnorm(sizes[j] * sizes[j + 1L]),
                ncol = sizes[j],
                nrow = sizes[j + 1L])
    weights[[j]] <- list(w=w,
                         b=rnorm(sizes[j + 1L]))
  }
  weights
}
```

Next, we need to define the ReLU function for the forward pass:

```
# Apply a rectified linear unit (ReLU) to a vector/matrix.
#
# Args:
#     v: A numeric vector or matrix.
#
# Returns:
```

```
#     The original input with negative values truncated to zero.
casl_util_ReLU <-
function(v)
{
  v[v < 0] <- 0
  v
}
```

And the derivative of the ReLU function for the backwards pass:

```
# Apply derivative of the rectified linear unit (ReLU).
#
# Args:
#     v: A numeric vector or matrix.
#
# Returns:
#     Sets positive values to 1 and negative values to zero.
casl_util_ReLU_p <-
function(v)
{
  p <- v * 0
  p[v > 0] <- 1
  p
}
```

We also need to differentiate the loss function for backpropagation. Here we use mean squared error (multiplied by 0.5).

```
# Derivative of the mean squared error (MSE) function.
#
# Args:
#     y: A numeric vector of responses.
#     a: A numeric vector of predicted responses.
#
# Returns:
#     Returned current derivative the MSE function.
casl_util_mse_p <-
function(y, a)
{
  (a - y)
}
```

We will write the code to accept a generic loss function derivative.

With the basic elements in place, we now need to describe how to take an input x and compute all of the responses z and activations a. These will also be stored as lists with one element per layer. Our function here accepts a generic activation function `sigma`.

```
# Apply forward propagation to a set of NN weights and biases.
#
# Args:
#     x: A numeric vector representing one row of the input.
#     weights: A list created by casl_nn_make_weights.
#     sigma: The activation function.
#
# Returns:
#     A list containing the new weighted responses (z) and
#     activations (a).
casl_nn_forward_prop <-
function(x, weights, sigma)
{
  L <- length(weights)
  z <- vector("list", L)
  a <- vector("list", L)
  for (j in seq_len(L))
  {
    a_j1 <- if(j == 1) x else a[[j - 1L]]
    z[[j]] <- weights[[j]]$w %*% a_j1 + weights[[j]]$b
    a[[j]] <- if (j != L) sigma(z[[j]]) else z[[j]]
  }

  list(z=z, a=a)
}
```

With the forward propagation function written, next we need to code a backpropagation function using the results from Equations 8.42 and 8.43. We will have this function accept the output of the forward pass and functions giving the derivatives of the loss and activation functions.

```
# Apply backward propagation algorithm.
#
# Args:
#     x: A numeric vector representing one row of the input.
#     y: A numeric vector representing one row of the response.
#     weights: A list created by casl_nn_make_weights.
#     f_obj: Output of the function casl_nn_forward_prop.
#     sigma_p: Derivative of the activation function.
#     f_p: Derivative of the loss function.
#
# Returns:
#     A list containing the new weighted responses (z) and
#     activations (a).
casl_nn_backward_prop <-
```

```
function(x, y, weights, f_obj, sigma_p, f_p)
{
  z <- f_obj$z; a <- f_obj$a
  L <- length(weights)
  grad_z <- vector("list", L)
  grad_w <- vector("list", L)
  for (j in rev(seq_len(L)))
  {
    if (j == L)
    {
      grad_z[[j]] <- f_p(y, a[[j]])
    } else {
      grad_z[[j]] <- (t(weights[[j + 1]]$w) %*%
                      grad_z[[j + 1]]) * sigma_p(z[[j]])
    }
    a_j1 <- if(j == 1) x else a[[j - 1L]]
    grad_w[[j]] <- grad_z[[j]] %*% t(a_j1)
  }

  list(grad_z=grad_z, grad_w=grad_w)
}
```

The output of the backpropagation function gives a list of gradients with respect to z and w. Recall that the derivatives with respect to z are equivalent to those with respect to the bias terms (Equation 8.31).

Using these building blocks, we can write a function `casl_nn_sgd` that takes input data, runs SGD, and returns the learned weights from the model. As inputs we also include the number of epochs (iterations through the data) and the learning rate `eta`.

```
# Apply stochastic gradient descent (SGD) to estimate NN.
#
# Args:
#     X: A numeric data matrix.
#     y: A numeric vector of responses.
#     sizes: A numeric vector giving the sizes of layers in
#            the neural network.
#     epochs: Integer number of epochs to computer.
#     eta: Positive numeric learning rate.
#     weights: Optional list of starting weights.
#
# Returns:
#     A list containing the trained weights for the network.
casl_nn_sgd <-
function(X, y, sizes, epochs, eta, weights=NULL)
{
```

```
  if (is.null(weights))
  {
    weights <- casl_nn_make_weights(sizes)
  }

  for (epoch in seq_len(epochs))
  {
    for (i in seq_len(nrow(X)))
    {
      f_obj <- casl_nn_forward_prop(X[i,], weights,
                                    casl_util_ReLU)
      b_obj <- casl_nn_backward_prop(X[i,], y[i,], weights,
                                     f_obj, casl_util_ReLU_p,
                                     casl_util_mse_p)

      for (j in seq_along(b_obj))
      {
        weights[[j]]$b <- weights[[j]]$b -
                          eta * b_obj$grad_z[[j]]
        weights[[j]]$w <- weights[[j]]$w -
                          eta * b_obj$grad_w[[j]]
      }
    }
  }

  weights
}
```

We have also written in the ability to include an optional set of starting weights. This allows users to further train a fit model, possibly with a new learning rate or new dataset.

Finally, we need a function that takes the learned weights and a new dataset X and returns the fitted values from the neural network.

```
# Predict values from a training neural network.
#
# Args:
#     weights: List of weights describing the neural network.
#     X_test: A numeric data matrix for the predictions.
#
# Returns:
#     A matrix of predicted values.
casl_nn_predict <-
function(weights, X_test)
{
```

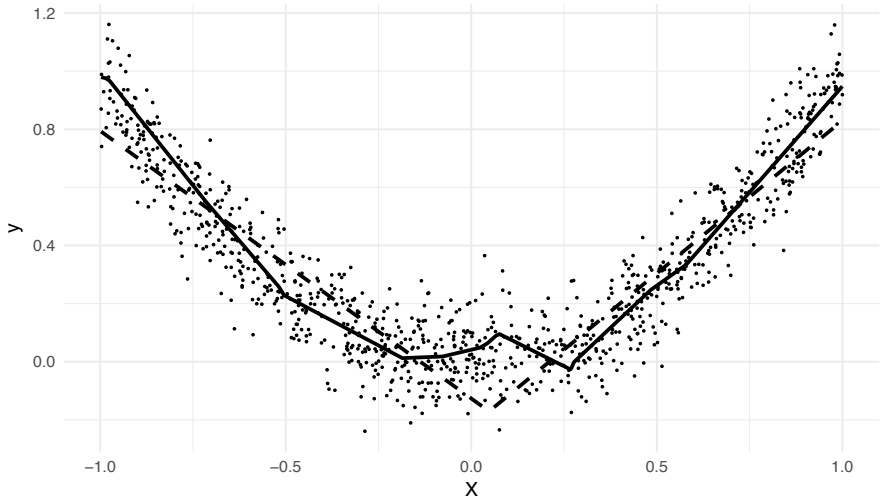

FIGURE 8.4: This scatter plot shows simulated data with a scalar input X and scalar output y (a noisy version of X^2). The dashed line shows the estimated functional relationship from a neural network with one hidden layer containing 5 hidden nodes. The solid line shows the estimated relationship using a model with 25 hidden nodes. Notice that the latter does a better job of fitting the quadratic relationship but slightly overfits near X equal to 0.1. Rectified linear units (ReLUs) were used as activation functions.

```
  p <- length(weights[[length(weights)]]$b)
  y_hat <- matrix(0, ncol = p, nrow = nrow(X_test))
  for (i in seq_len(nrow(X_test)))
  {
    a <- casl_nn_forward_prop(X_test[i,], weights,
                              casl_util_ReLU)$a
    y_hat[i, ] <- a[[length(a)]]
  }

  y_hat
}
```

The implementation here applies the forward propagation function and returns only the last layer of activations.

We can test the code on a small dataset using just a one column input X and one hidden layer with 25 nodes.

```
X <- matrix(runif(1000, min=-1, max=1), ncol=1)
y <- X[,1,drop = FALSE]^2 + rnorm(1000, sd = 0.1)
weights <- casl_nn_sgd(X, y, sizes=c(1, 25, 1),
```

```
                          epochs=25, eta=0.01)
y_pred <- casl_nn_predict(weights, X)
```

The output in Figure 8.4 shows that the neural network does a good job of capturing the non-linear relationship between X and y.

Coding the backpropagation algorithm is notoriously error prone. It is particularly difficult because many errors lead to algorithms that find reasonable, though non-optimal, solutions. A common technique is to numerically estimate the gradient of each parameter for a small model and then check the computed gradients compared to the numerical estimates. Specifically, we can compute

$$\nabla_w f(w_0) \approx \frac{f(w_0 + h) - f(w_0 - h)}{2h} \tag{8.44}$$

For a small vector h with only one non-zero parameter and compare this with the perturbation applied to all trainable parameters in the model.

The code to do this is relatively straightforward. Note that we need to compute the true gradient over all of the data points, not just for a single input x. The reason for this is that given the ReLU activations, many of the derivatives will be trivially equal to zero for many weights with a given input. Added up over all training points, however, this will not be the case for most parameters.

```
# Perform a gradient check for the dense NN code.
#
# Args:
#     X: A numeric data matrix.
#     y: A numeric vector of responses.
#     weights: List of weights describing the neural network.
#     h: Positive numeric bandwidth to test.
#
# Returns:
#     The largest difference between the empirical and analytic
#     gradients of the weights and biases.
casl_nn_grad_check <-
function(X, y, weights, h=0.0001)
{
  max_diff <- 0
  for (level in seq_along(weights))
  {
    for (id in seq_along(weights[[level]]$w))
    {
      grad <- rep(0, nrow(X))
      for (i in seq_len(nrow(X)))
      {
        f_obj <- casl_nn_forward_prop(X[i, ], weights,
```

```
                                        casl_util_ReLU)
        b_obj <- casl_nn_backward_prop(X[i, ], y[i, ], weights,
                                        f_obj, casl_util_ReLU_p,
                                        casl_util_mse_p)
        grad[i] <- b_obj$grad_w[[level]][id]
      }

      w2 <- weights
      w2[[level]]$w[id] <- w2[[level]]$w[id] + h
      f_h_plus <- 0.5 * (casl_nn_predict(w2, X) - y)^2
      w2[[level]]$w[id] <- w2[[level]]$w[id] - 2 * h
      f_h_minus <- 0.5 * (casl_nn_predict(w2, X) - y)^2

      grad_emp <- sum((f_h_plus - f_h_minus) / (2 * h))

      max_diff <- max(max_diff,
                      abs(sum(grad) - grad_emp))
    }
  }
  max_diff
}
```

We can now check our backpropagation code. Here, we will set the weights with just a single epoch so that the resulting gradients are not too small.

```
weights <- casl_nn_sgd(X, y, sizes=c(1, 5, 5, 1), epochs=1,
                       eta=0.01)
casl_nn_grad_check(X, y, weights)
```

```
[1] 5.911716e-12
```

Our numeric gradients match all of the simulated gradients up to a very small number, indicating that our computation of the gradient is accurate.

8.5 Recognizing handwritten digits

In most chapters we have saved applications with real datasets until the final sections. It is, however, hard to show the real usage of neural networks on simulated data. Here we will apply our neural network implementation to a small set of images in order to illustrate how neural networks behave on real datasets.

The MNIST dataset consists of a collection of 60,000 black and white images of handwritten digits. The images are scaled and rotated so that each

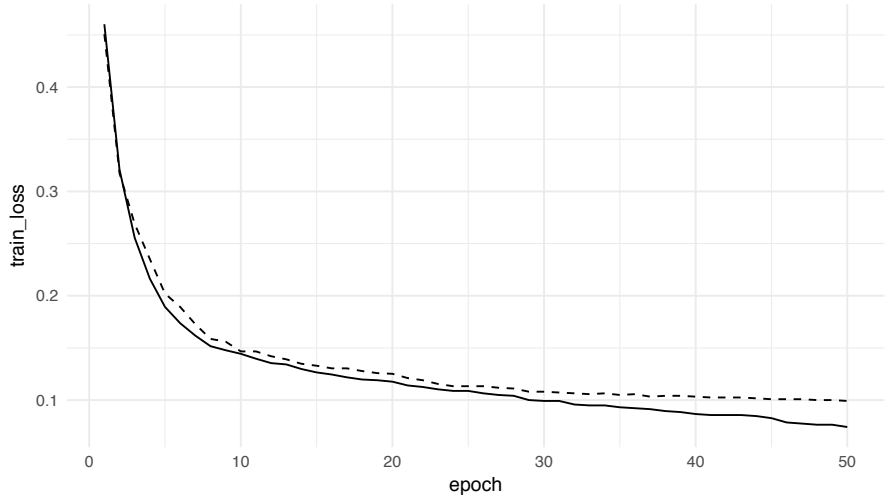

FIGURE 8.5: This plot shows the loss (root mean squared error) during training on both the training and validation sets at the end of each epoch. The solid line is the training error and the dashed line is the validation error.

digit is relatively centered. All images are converted to be 28 pixels by 28 pixels. In Section 8.10 we will see an extension of this dataset that includes handwritten letters. Here we work with a smaller version of MNIST that has been down-sampled to only 7 pixels by 7 pixels and includes only the numbers 0 and 1. This is a binary prediction problem. We can use the neural network from Section 8.4 for a continuous response by predicting the probability that an input is equal to 1 and treating the response as a continuous variable.

The dataset consists of 6000 images for each digits. We split the data in half to form testing and training sets; the response vector is also coded as a matrix as required by our neural network code. The input data, `X_mnist` is given as a three-dimensional array, which we flatten into a matrix by applying the `cbind` function over the rows.

```
X_mnist <- readRDS("data/mnist_x7.rds")
mnist <- read.csv("data/mnist.csv")

X_train <- t(apply(X_mnist[mnist$train_id == "train", , ], 1L,
                   cbind))
X_valid <- t(apply(X_mnist[mnist$train_id == "valid", , ], 1L,
                   cbind))
y_train <- matrix(mnist[mnist$train_id == "train", ]$class,
                  ncol=1L)
y_valid <- matrix(mnist[mnist$train_id == "valid", ]$class,
                  ncol=1L)
```

To learn the output, we will fit a linear model with two hidden layers, each containing 64 neurons. In order to plot its progress through the learning algorithm, the learning algorithm will be wrapped in a loop with the training and validation loss stored after every epoch.

```
results <- matrix(NA_real_, ncol = 2, nrow = 25)
val <- NULL
for (i in seq_len(nrow(results)))
{
  val <- casl_nn_sgd(X_train, y_train,
                     sizes=c(7^2, 64, 64, 1),
                     epochs=1L,
                     eta=0.001,
                     weights=val)

  y_train_pred <- (casl_nn_predict(val, X_train) > 0.5)
  y_train_pred <- as.numeric(y_train_pred)
  y_valid_pred <- (casl_nn_predict(val, X_valid) > 0.5)
  y_valid_pred <- as.numeric(y_valid_pred)
  results[i,1] <- sqrt(mean((y_train - y_train_pred)^2))
  results[i,2] <- sqrt(mean((y_valid - y_valid_pred)^2))
}
```

As we are treating this as a continuous response, the results are stored in terms of mean squared error. The values in `results` are plotted in Figure 8.5. We see that towards the end of the training algorithm the training set error improves but the validation error asymptotes to a value 0.1. A selection of images in the validation set that are misclassified by the algorithm are shown in Figure 8.6.

8.6 ($\star$) Improving SGD and regularization

The straightforward SGD and backpropagation algorithms implemented in Section 8.4 performs reasonably well. While our R implementation will run quite slowly compared to algorithms written in a compiled language, the algorithms themselves could easily scale to problems with several hidden layers and millions of observations. There are, however, several improvements we can make to the model and the training algorithm that reduce overfitting, have a lower tendency to get stuck in local saddle points, or converge in a smaller number of epochs.

One minor change is to update the weight initialization algorithm. We want the starting activations a^l to all be of roughly the same order of magnitude with the initialized weights. Assuming that the activations in layer a^{l-1} are

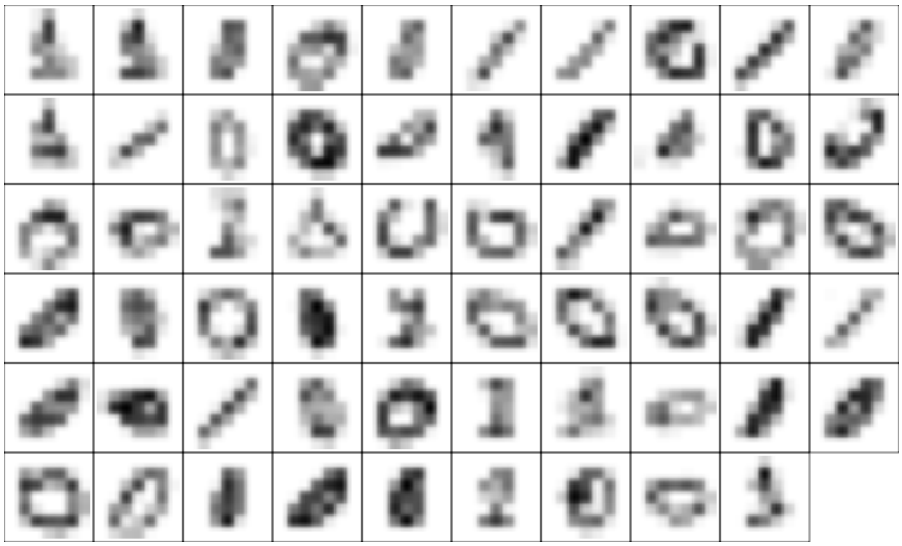

FIGURE 8.6: Misclassified samples from the down-sampled MNIST datasets with only 7-by-7 images.

fixed, we can compute the variance of z^l as a function of the weights in the lth layer,

$$\text{Var}(z_j^l) = \text{Var}\left(\sum_{k=1}^{M^{l-1}} \left[w_{ji}^l a_k^{l-1} + b_k^l\right]\right) \tag{8.45}$$

$$= \sum_{k=1}^{M^{l-1}} Var(w_{ji}^l)\left[a_k^{l-1}\right]^2 \tag{8.46}$$

Where M^l gives the number of neurons in the lth layer of the model. If we sample w_{ji}^{l-1} from the same distribution and assume that the activations in the $(l-1)$st layer are already of approximately the same magnitude, then we see that

$$\text{Var}(z_j^l) \propto M^{l-1} \cdot Var(w_{j1}^l) \tag{8.47}$$

This suggests that we sample the weights such that

$$w_{j1}^l \sim \mathcal{N}(0, \frac{1}{M^{l-1}}). \tag{8.48}$$

This is the recommendation proposed by Yann LeCun and Léon Bottou in a technical report on the subject of neural network initializers [103]. More complex procedures have been suggested by Kaiming He [73] and Xavier Glorot [67]. In our small example with just one hidden layer, the initializations

did not have a large effect. For models with many layers, particularly if they are of vastly different sizes, the choice of initialization scheme can drastically improve the performance of the model learned by SGD.

We now integrate the new initialization scheme into a new version of the `casl_nn_make_weights` function. We keep the bias terms sampled from a standard normal distribution; it is also reasonable to set these equal to zero at the start.

```
# Create list of weights and momentum to describe a NN.
#
# Args:
#     sizes: A vector giving the size of each layer, including
#            the input and output layers.
#
# Returns:
#     A list containing initialized weights, biases, and
#     momentum.
casl_nn_make_weights_mu <-
function(sizes)
{
  L <- length(sizes) - 1L
  weights <- vector("list", L)
  for (j in seq_len(L))
  {
    w <- matrix(rnorm(sizes[j] * sizes[j + 1L],
                      sd = 1/sqrt(sizes[j])),
                ncol = sizes[j],
                nrow = sizes[j + 1L])
    v <- matrix(0,
                ncol = sizes[j],
                nrow = sizes[j + 1L])
    weights[[j]] <- list(w=w,
                         v=v,
                         b=rnorm(sizes[j + 1L]))
  }
  weights
}
```

This is a minor change that can be used as is in the `casl_nn_sgd` with no further modifications.

The large number of trainable parameters in a neural network can obviously lead to overfitting on the training data. Many of the tweaks to the basic neural network structure and SGD are directly related to addressing this concern. One common technique is *early stopping*. In this approach the validation error rate is computed after each epoch of the SGD algorithm. When the validation rate fails to improve after a certain number of epochs, or begins to

degrade, the SGD algorithm is terminated. Early stopping is almost always used when training neural networks. While neural networks are often framed as predictive models described by an optimization problem, this early stopping criterion means that in practice this is not the case. Neural networks are trained with an algorithm motivated by an optimization problem, but are not directly attempting to actually solve the optimization task.

Another approach to address overfitting is to include a penalty term directly into the loss function. We can add an ℓ_2-norm, for example, as was done with ridge regression in Section 3.2,

$$f_\lambda(w, b) = f(w, b) + \frac{\lambda}{2} \cdot ||w||_2^2 \tag{8.49}$$

Notice here that we have penalized just the weights (slopes) but not the biases (intercepts). The gradient of f_λ can be written in terms of the gradient of f:

$$\nabla_w f_\lambda = \nabla_w f(w, b) + \lambda \cdot w \tag{8.50}$$

With this new form, the SGD algorithm is easy to modify. It is helpful to simplify the gradient updates in terms of a weighted version of the old values and a weighted version of the gradient.

$$\begin{aligned} w_{new} &\leftarrow w_{old} - \eta \cdot \nabla_w f_\lambda(w_{old}) && (8.51)\\ &\leftarrow w_{old} - \eta \cdot [\nabla_w(w_{old}) + \lambda \cdot w] && (8.52)\\ &\leftarrow [1 - \eta\lambda] \cdot w_{old} - \eta \cdot \nabla_w f(w_{old}) && (8.53) \end{aligned}$$

Other penalties, such as the ℓ_1-norm or penalties on the bias terms, can also be added with minimal difficulty.

Dropout is another clever technique for reducing over fitting during the training for neural networks. During the forward propagation phase, activations are randomly set to zero with some fixed probability p. During backpropagation, the derivatives of the corresponding activations are also set to zero. The activations are chosen separately for each mini-batch. Together, these modifications "prevents units from co-adapting too much," having much the same effect as an ℓ_2-penalty without the need to choose or adapt the parameter λ [148]. The dropout technique is only used during training; all nodes are turned on during prediction on new datasets (in order for the magnitude to work out, weights need to be scaled by a factor of $\frac{1}{1-p}$). Dropout is a popular technique used in most well-known neural networks and is relatively easy to incorporate into the forward and backpropagation algorithms.

Finally, we can also modify the SGD algorithm itself. It is not feasible to store second derivative information directly due to the large number of parameters in a neural network. However, relying only on the gradient leads to models getting stuck at saddle points and being very sensitive to the learning rate. To alleviate this problem, we can incorporate a term known as the *momentum* into the algorithm. The gradient computed on each mini-batch is

used to update the momentum term, and the momentum term is then used to update the current weights. This setup gives the SGD algorithm three useful properties: if the gradient remains relatively unchanged over several steps it will 'pick-up speed' (momentum) and effectively use a larger learning rate; if the gradient is changing rapidly, the step-sizes will shrink accordingly; when passing through a saddle point, the built up momentum from prior steps helps propel the algorithm past the point. Expressing the momentum as the vector v, we then have update rules,

$$v_{new} \leftarrow v_{old} \cdot \mu - \eta \cdot \nabla_w f(w_{old}) \tag{8.54}$$

$$w_{new} \leftarrow [1 - \eta\lambda] \cdot w_{old} + v_{new}, \tag{8.55}$$

with μ some quantity between 0 and 1 (typically set between 0.7 and 0.9). Notice that this scheme requires only storing twice as much information as required for the gradient. Here, and in our implementation below, we apply momentum only to the weights. It is also possible, and generally advisable, to apply momentum terms to the biases as well.

Implementing an ℓ_2-penalty term and momentum only requires changes to the `casl_nn_sgd` function. Early stopping requires no direct changes and can be applied as is by running the SGD function for a single epoch, checking the validation rate, and then successively re-running the SGD for another epoch starting with the current weights. Dropout requires minor modifications to both `casl_nn_forward_prop` and `casl_nn_backward_prop`, which we leave as an exercise.

```
# Apply stochastic gradient descent (SGD) to estimate NN.
#
# Args:
#     X: A numeric data matrix.
#     y: A numeric vector of responses.
#     sizes: A numeric vector giving the sizes of layers in
#            the neural network.
#     epochs: Integer number of epochs to computer.
#     eta: Positive numeric learning rate.
#     mu: Non-negative momentum term.
#     l2: Non-negative penalty term for l2-norm.
#     weights: Optional list of starting weights.
#
# Returns:
#     A list containing the trained weights for the network.
casl_nn_sgd_mu <-
function(X, y, sizes, epochs, eta, mu=0, l2=0, weights=NULL) {

  if (is.null(weights))
  {
    weights <- casl_nn_make_weights_mu(sizes)
```

```
  }

  for (epoch in seq_len(epochs))
  {
    for (i in seq_len(nrow(X)))
    {
      f_obj <- casl_nn_forward_prop(X[i, ], weights,
                                    casl_util_ReLU)
      b_obj <- casl_nn_backward_prop(X[i, ], y[i, ], weights,
                                     f_obj, casl_util_ReLU_p,
                                     casl_util_mse_p)

      for (j in seq_along(b_obj))
      {
        weights[[j]]$b <- weights[[j]]$b -
                            eta * b_obj$grad_z[[j]]
        weights[[j]]$v <- mu * weights[[j]]$v -
                          eta * b_obj$grad_w[[j]]
        weights[[j]]$w <- (1 - eta * l2) *
                          weights[[j]]$w +
                               weights[[j]]$v
      }
    }
  }

  weights
}
```

The function accepts two new parameters, `mu` and `l2`, to define the momentum and penalty terms, respectively.

Figure 8.7 shows the loss function during training both with and without momentum using the data from our simulation in Section 8.4. We see that the momentum-based algorithm trains significantly faster and seems to avoid a saddle point that the non-momentum based SGD gets trapped in. To illustrate the use of the ℓ_2-norm, we will apply the SGD algorithm using successively larger values of λ (`l2`).

```
l2_norm <- rep(NA_real_, 3)
l2_vals <- c(0, 0.01, 0.04, 0.05)
weights_start <- casl_nn_make_weights(sizes=c(1, 10, 1))
for (i in seq_along(l2_vals))
{
  weights <- casl_nn_sgd_mu(X, y, weights=weights_start,
                            epochs=10, eta=0.1,
                            l2=l2_vals[i])
  l2_norm[i] <- sum((weights[[1]]$w)^2)
```

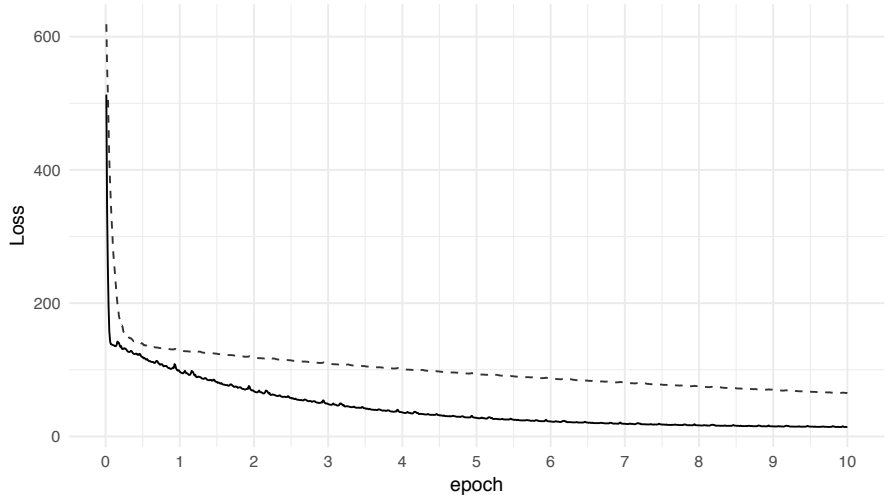

FIGURE 8.7: This plot shows the loss (mean squared error) during training on the training set as a function of the epoch number. The dashed line uses stochastic gradient descent whereas the solid line uses stochastic gradient descent plus momentum.

```
}
l2_norm
```

```
[1] 6.942948e+00 1.674410e+00 3.672386e-02 7.117368e-07
```

We see that, as expected, the size of the weights decreases as the penalty term increases.

8.7 ($\star$) Classification with neural networks

The structure of neural networks is well-suited to classification tasks with many possible classes. In fact, we mentioned in Section 5.6 that the original multinomial regression function in R (`multinom`) is contained in the **nnet** package. Here, we will show how to best use neural networks for classification and integrate these changes to our implementation.

With neural networks there is nothing special about the structure of the output layer. As with hidden layers, it is easy to have a multi-valued output layer. For classification tasks, we can convert a vector y of integer coded classes

into a matrix Y containing indicator variables for each class,

$$\begin{pmatrix} 2 \\ 4 \\ \vdots \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{8.56}$$

In neural network literature this is known as a *one-hot encoding*. It is equivalent to the indicator variables used in Section 2.1 applied to categorical variables in the data matrix X. Now, values from the neural network in the output layer can be regarded as probabilities over the possible categories.

Treating the output layer as probabilities raises the concern that these values may not sum to 1 and could produce negative values or values greater than one depending on the inputs. As we did with the logistic link function in Section 5.1, we need to apply a function to the output layer to make it behave as a proper set of probabilities. This will become the activation function σ in the final layer that we have in Equation 8.43. The activation we use is called the *softmax function*, defined as:

$$a_j^L = \text{softmax}(z_j^L) \tag{8.57}$$

$$= \frac{e^{z_j}}{\sum_k e^{z_k}}. \tag{8.58}$$

It should be obvious from the definition that the returned values are all non-negative and sum to 1 (and therefore can never be greater than 1). While we could use squared error loss to train categorization models, it is not an ideal choice. We instead use a quantity known as categorical cross-entropy:

$$f(a^L, y) = -\sum_k y_k \cdot log(a_k^L) \tag{8.59}$$

If the form of this seems surprising, notice that in two-class prediction this reduces to the log-likelihood for logistic regression. In fact, the multinomial distribution can be written as a multidimensional exponential family with the softmax function as an inverse link function and categorical cross-entropy as the log-likelihood function (see Section 5.2 for a description of exponential families and [125] for more details).

The derivative of the softmax function can be written compactly as a function of the Dirac delta operator δ_{ij}. The Dirac delta function is equal to 1 if the indices i and j match, and is equal to 0 otherwise. The softmax derivatives then become

$$\frac{\partial a_j^L}{\partial z_i^L} = \frac{\delta_{ij} e^{z_j} \cdot (\sum_k e^{z_k}) - e^{z_j} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} \tag{8.60}$$

$$= a_j^L(\delta_{ij} - a_i^L). \tag{8.61}$$

With categorical cross-entropy, Equation 8.43 becomes a simple linear function of the activations a^L and the values of Y. Denoting y_i as the ith column of a particular row of Y, we have

$$\frac{\partial f}{\partial z_i^L} = -\sum_k y_k \cdot \frac{\partial}{\partial z_i^L}\left(log(a_k^L)\right) \tag{8.62}$$

$$= -\sum_k y_k \cdot \frac{1}{a_k^L} \cdot \frac{\partial a_k^L}{\partial z_i^L} \tag{8.63}$$

$$= -\sum_k y_k \cdot \frac{1}{a_k^L} \cdot a_k^L(\delta_{ik} - a_i^L) \tag{8.64}$$

$$= -\sum_k y_k \cdot (\delta_{ik} - a_i^L) \tag{8.65}$$

$$= a_i^L \cdot \left(\sum_k y_k\right) - \sum_k y_k \cdot \delta_{ik} \tag{8.66}$$

$$= a_i^L - y_i. \tag{8.67}$$

A change in the weighted response z^L has a linear effect on the loss function, a feature that stops the model from becoming too saturated with very small or very large predicted probabilities.

In order to implement a categorical neural network, we first need a softmax function to apply during forward propagation.

```
# Apply the softmax function to a vector.
#
# Args:
#     z: A numeric vector of inputs.
#
# Returns:
#     Output after applying the softmax function.
casl_util_softmax <-
function(z)
{
  exp(z) / sum(exp(z))
}
```

We then create a modified forward propagation function that takes advantage of the softmax function in the final layer.

```
# Apply forward propagation to for a multinomial NN.
#
# Args:
#     x: A numeric vector representing one row of the input.
#     weights: A list created by casl_nn_make_weights.
#     sigma: The activation function.
```

```
#
# Returns:
#     A list containing the new weighted responses (z) and
#     activations (a).
casl_nnmulti_forward_prop <-
function(x, weights, sigma)
{
  L <- length(weights)
  z <- vector("list", L)
  a <- vector("list", L)
  for (j in seq_len(L))
  {
    a_j1 <- if(j == 1) x else a[[j - 1L]]
    z[[j]] <- weights[[j]]$w %*% a_j1 + weights[[j]]$b
    if (j != L) {
      a[[j]] <- sigma(z[[j]])
    } else {
      a[[j]] <- casl_util_softmax(z[[j]])
    }
  }

  list(z=z, a=a)
}
```

Similarly, we need a new backpropagation function that applies the correct updates to the gradient of terms z^L from the final layer of the model.

```
# Apply backward propagation algorithm for a multinomial NN.
#
# Args:
#     x: A numeric vector representing one row of the input.
#     y: A numeric vector representing one row of the response.
#     weights: A list created by casl_nn_make_weights.
#     f_obj: Output of the function casl_nn_forward_prop.
#     sigma_p: Derivative of the activation function.
#
# Returns:
#     A list containing the new weighted responses (z) and
#     activations (a).
casl_nnmulti_backward_prop <-
function(x, y, weights, f_obj, sigma_p)
{
  z <- f_obj$z; a <- f_obj$a
  L <- length(weights)
  grad_z <- vector("list", L)
  grad_w <- vector("list", L)
```

```
  for (j in rev(seq_len(L)))
  {
    if (j == L)
    {
      grad_z[[j]] <- a[[j]] - y
    } else {
      grad_z[[j]] <- (t(weights[[j + 1L]]$w) %*%
                      grad_z[[j + 1L]]) * sigma_p(z[[j]])
    }
    a_j1 <- if(j == 1) x else a[[j - 1L]]
    grad_w[[j]] <- grad_z[[j]] %*% t(a_j1)
  }

  list(grad_z=grad_z, grad_w=grad_w)
}
```

Next, we construct an updated SGD function that calls these new forward and backward steps. The momentum and ℓ_2-norm terms remain unchanged.

```
# Apply stochastic gradient descent (SGD) for multinomial NN.
#
# Args:
#     X: A numeric data matrix.
#     y: A numeric vector of responses.
#     sizes: A numeric vector giving the sizes of layers in
#            the neural network.
#     epochs: Integer number of epochs to computer.
#     eta: Positive numeric learning rate.
#     mu: Non-negative momentum term.
#     l2: Non-negative penalty term for l2-norm.
#     weights: Optional list of starting weights.
#
# Returns:
#     A list containing the trained weights for the network.
casl_nnmulti_sgd <-
function(X, y, sizes, epochs, eta, mu=0, l2=0, weights=NULL) {

  if (is.null(weights))
  {
    weights <- casl_nn_make_weights_mu(sizes)
  }

  for (epoch in seq_len(epochs))
  {
    for (i in seq_len(nrow(X)))
    {
```

```
      f_obj <- casl_nnmulti_forward_prop(X[i, ], weights,
                                         casl_util_ReLU)
      b_obj <- casl_nnmulti_backward_prop(X[i, ], y[i, ],
                                          weights, f_obj,
                                          casl_util_ReLU_p)

      for (j in seq_along(b_obj))
      {
        weights[[j]]$b <- weights[[j]]$b -
                            eta * b_obj$grad_z[[j]]
        weights[[j]]$v <- mu * weights[[j]]$v -
                          eta * b_obj$grad_w[[j]]
        weights[[j]]$w <- (1 - eta * l2) *
                          weights[[j]]$w +
                               weights[[j]]$v
      }
    }
  }

  weights
}
```

Finally, we also produce a new prediction function that uses the correct forward propagation function for classification.

```
# Predict values from training a multinomial neural network.
#
# Args:
#     weights: List of weights describing the neural network.
#     X_test: A numeric data matrix for the predictions.
#
# Returns:
#     A matrix of predicted values.
casl_nnmulti_predict <-
function(weights, X_test)
{

  p <- length(weights[[length(weights)]]$b)
  y_hat <- matrix(0, ncol=p, nrow=nrow(X_test))
  for (i in seq_len(nrow(X_test)))
  {
    a <- casl_nnmulti_forward_prop(X_test[i, ], weights,
                                   casl_util_ReLU)$a
    y_hat[i,] <- a[[length(a)]]
  }
```

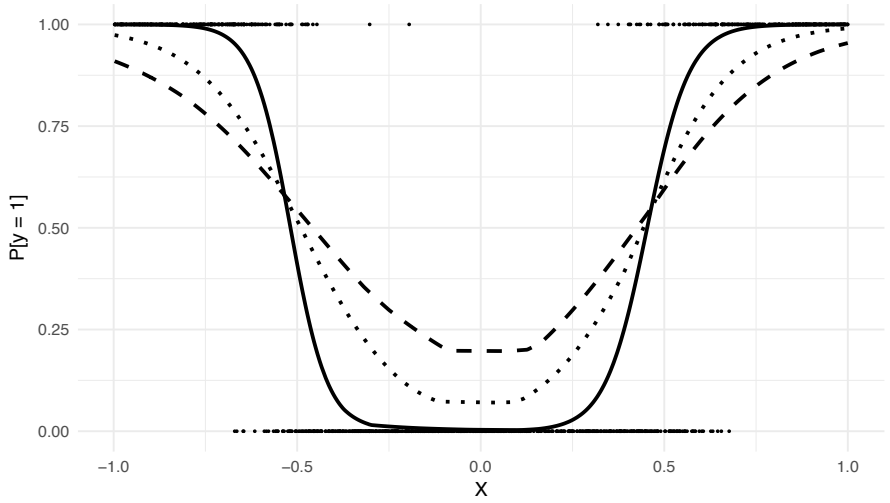

FIGURE 8.8: A scatter plot showing simulated data from a scalar quantity X and a binary variable y. The lines show fitted probabilities from a neural network with one hidden layer containing 25 neurons. Rectified linear units (ReLUs) were used on the hidden layer and softmax activations were used on the output layer to produce valid probabilities. The dashed line shows the fit at the end of the first epoch, the dotted line at the end of 2 epochs, and the solid line the solid line after 25 epochs. Stochastic gradient descent using momentum and categorical cross-entropy was used to the train the model.

```
  y_hat
}
```

The results of the prediction are a matrix with one row for each row in `X_text` and one column for each class in the classification task. To get the predicted class, can simply find whichever column contains the largest probability.

To illustrate this approach, we will simulate a small dataset with just two classes. We again restrict ourselves to a scalar input `X` in order to be able to plot the output in x-y space.

```
X <- matrix(runif(1000, min=-1, max=1), ncol=1)
y <- X[, 1, drop=FALSE]^2 + rnorm(1000, sd=0.1)
y <- cbind(as.numeric(y > 0.3), as.numeric(y <= 0.3))
weights <- casl_nnmulti_sgd(X, y, sizes=c(1, 25, 2),
                            epochs=25L, eta=0.01)
y_pred <- casl_nnmulti_predict(weights, X)
```

The predicted probabilities for various number of epochs are shown in Figure 8.8. During training, the predicted probabilities become more extreme as the algorithm gains confidence that certain inputs always lead to a particular

category. On the boundary regions, the neural network correctly predicts a smooth continuum of probabilities. It also has no difficulty detecting the non-linear relationship between the probabilities and the input X (class 1 is most common when X is both relatively small or relatively large).

8.8 (⋆) Convolutional neural networks

As we have mentioned, prediction tasks with images as inputs constitute some of most popular applications of neural networks. When images are relatively small, it is possible to learn a neural network with individual weights placed on each input pixel. For larger images this becomes impractical. A model learned in this way does not take into account the spatial structure of the image, thus throwing away useful information. The solution to this problem is to include convolutional layers into the neural network. Convolutions apply a small set of weights to subsections of the image; the same weights are used across the image.

In the context of neural networks, a convolution can be described by a kernel matrix. Assume that we have the following kernel matrix, to be applied over a black and white image

$$K = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{8.68}$$

The convolution described by this kernel takes every pixel value and subtracts is from the pixel value to its immediate lower right. Notice that this cannot be applied directly to pixels in the last row or column as there is no corresponding pixel to subtract from. The result of the kernel, then, is a new image with one fewer row and column. As shown in Figure 8.9, the convolution created by the kernel is able to capture edges in the original image.

A convolutional layer in a neural network applies several kernels to the input image; the weights, rather than being fixed, are themselves learned as part of the training algorithm. Subsequent layers of the network *flatten* out the multidimensional array and fit dense hidden layers as we have done in previous sections. The idea is that different convolutions will pick up different features. We have seen that one choice of a kernel matrix detects edges; learned convolutions can identify features such as oriented edges, descriptions of texture, and basic object types. If the input image is in color, the kernel matrix K must be a three-dimensional array with weights applied to each color channel. Similarly, we can apply multiple layers of convolutions to the image. The third dimension of kernels in the second layer has to match the size of the number of kernels in the second layer. Stacking convolutions allows neural networks to extract increasingly complex features from the input data.

For simplicity, we will implement a convolutional neural network with a

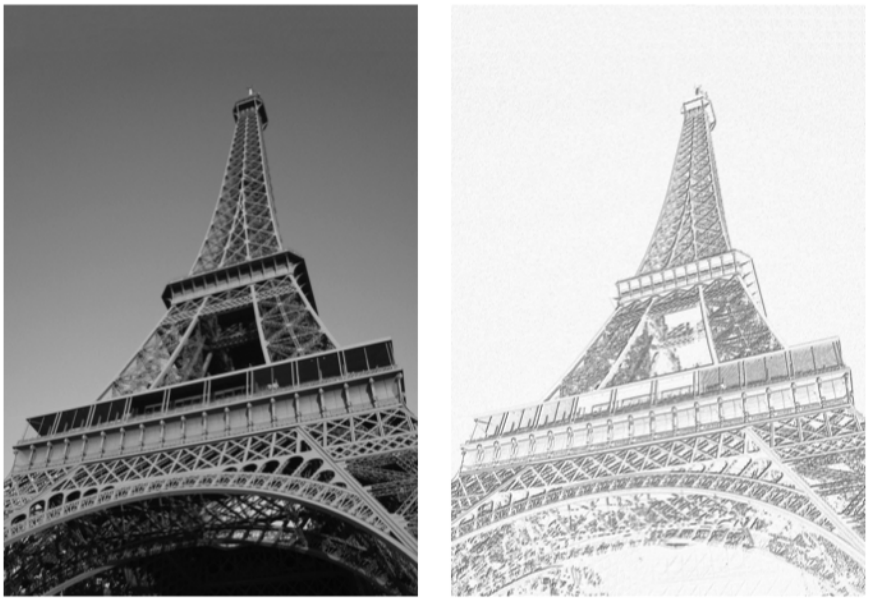

FIGURE 8.9: The left image is a black and white image of the Eiffel Tower (photo from WikiMedia by Arnaud Ligny). The right image shows the filter obtained by applying the kernel defined in Equation 8.68 to the original image.

single convolutional layer applied to a black and white image. We will also hard code the kernels to be of size 3-by-3 (the most common choice in image processing). As most image prediction tasks involve classification, we will build off of the implementation for multiclass classification from Section 8.7.

A mathematical definition of a convolution is relatively straightforward. Using the notation of Section 8.3, we will describe a neural network that replaces the first layer with a convolution. First, we will need to describe the input using two indices to represent the two spatial dimensions of the data,

$$a^0_{i,j} = x_{i,j}. \tag{8.69}$$

Then, assuming we want to use kernels of size k_1-by-k_2, the output of the first hidden layer is given by

$$z^1_{i,j,k} = \sum_{m=0}^{k_1} \sum_{n=0}^{k_2} w^1_{m,n} \cdot a^0_{i,j} + b^1_k. \tag{8.70}$$

The indices i and j represent the height and width of an input pixel and k indicates the kernel number. To define the rest of the neural network, we

re-parametrize z^1 as:

$$z^1_q = z^1_{i,j,k}, \quad q = (i-1) \cdot W \cdot K + (j-1) \cdot K + j \tag{8.71}$$

with K the total number of kernels and W the width of the input image. Using z^1_q, all of our previous equations for dense neural networks hold, including those for backpropagation.

By applying the definition in Equation 8.70, it is relatively easy to compute the forward pass step in training neural networks with a single convolutional layer. How does the backpropagation step work? Notice that equations for $\frac{\partial f}{\partial z^1_q}$ still hold with no changes. The only steps required are computing the gradient of the parameters in layer 1 in terms of the derivatives with respect to z^1_q. Once again, this is achieved by applying the chain rule, first for the intercepts

$$\frac{\partial f}{\partial b^1_k} = \sum_i \sum_j \frac{\partial f}{\partial z^1_{i,j}} \cdot \frac{\partial z^1_{i,j}}{\partial b^1_k} \tag{8.72}$$

$$= \sum_i \sum_j \frac{\partial f}{\partial z^1_{i,j}} \tag{8.73}$$

and then for the weights

$$\frac{\partial f}{\partial w^1_{m,n,k}} = \sum_i \sum_j \frac{\partial f}{\partial z^1_{i,j}} \cdot \frac{\partial z^1_{i,j}}{\partial w^1_{m,n,k}} \tag{8.74}$$

$$= \sum_i \sum_j \frac{\partial f}{\partial z^1_{i,j}} \cdot a^0_{i+m,j+n}. \tag{8.75}$$

Given that a weight or bias term in the convolutional layers affects all of the outputs in a given filter, it should seem reasonable that our derivatives now involve a sum over many terms. The only slight complication involves making sure that we compute the derivatives of z indexed as vector, but then apply them to the next layer as an array with three dimensions. This is conceptually simple but some care must be taken in the implementation.

The weights for the first layer of our convolutional neural network need an array of dimension 3-by-3-by-K, where K is the number of kernels. The second layer of weights accepts $(W-2) \cdot (H-2) \cdot K$ inputs, where W and H are the width and height of the input image. We will implement `casl_nn_make_weights` such that the first size gives the size of the outputs in the first convolution and the second size gives the number of filters in the convolution.

```
# Create list of weights and momentum to describe a CNN.
#
# Args:
#     sizes: A vector giving the size of each layer, including
```

```
#            the input and output layers.
#
# Returns:
#     A list with initialized weights, biases, and momentum.
casl_cnn_make_weights <-
function(sizes)
{
  L <- length(sizes) - 1L
  weights <- vector("list", L)
  for (j in seq_len(L))
  {
    if (j == 1)
    {
      w <- array(rnorm(3 * 3 * sizes[j + 1]),
                 dim=c(3, 3, sizes[j + 1]))
      v <- array(0,
                 dim=c(3, 3, sizes[j + 1]))
    } else {
      if (j == 2) sizes[j] <- sizes[2] * sizes[1]
      w <- matrix(rnorm(sizes[j] * sizes[j + 1],
                        sd=1/sqrt(sizes[j])),
                  ncol=sizes[j],
                  nrow=sizes[j + 1])
      v <- matrix(0,
                  ncol=sizes[j],
                  nrow=sizes[j + 1])
    }

    weights[[j]] <- list(w=w,
                         v=v,
                         b=rnorm(sizes[j + 1]))
  }
  weights
}
```

The output is, as before, a list containing the weights, velocities, and bias terms.

The forward propagation step now needs to convolve the weights in the first layer with the input image.

```
# Apply forward propagation to a set of CNN weights and biases.
#
# Args:
#     x: A numeric vector representing one row of the input.
#     weights: A list created by casl_nn_make_weights.
#     sigma: The activation function.
```

```
#
# Returns:
#     A list containing the new weighted responses (z) and
#     activations (a).
casl_cnn_forward_prop <-
function(x, weights, sigma)
{
  L <- length(weights)
  z <- vector("list", L)
  a <- vector("list", L)
  for (j in seq_len(L))
  {
    if (j == 1)
    {
      a_j1 <- x
      z[[j]] <- casl_util_conv(x, weights[[j]])
    } else {
      a_j1 <- a[[j - 1L]]
      z[[j]] <- weights[[j]]$w %*% a_j1 + weights[[j]]$b
    }
    if (j != L)
    {
      a[[j]] <- sigma(z[[j]])
    } else {
      a[[j]] <- casl_util_softmax(z[[j]])
    }
  }

  list(z=z, a=a)
}
```

The `casl_util_conv` function called by the implementation of forward propagation function is defined as follows.

```
# Apply the convolution operator.
#
# Args:
#     x: The input image as a matrix.
#     w: Matrix of the kernel weight.
#
# Returns:
#     Vector of the output convolution.
casl_util_conv <-
function(x, w) {
  d1 <- nrow(x) - 2L
  d2 <- ncol(x) - 2L
```

```
  d3 <- dim(w$w)[3]
  z <- rep(0, d1 * d2 * d3)
  for (i in seq_len(d1))
  {
    for (j in seq_len(d2))
    {
      for (k in seq_len(d3))
      {
        val <- x[i + (0:2), j + (0:2)] * w$w[,,k]
        q <- (i - 1) * d2 * d3 + (j - 1) * d3 + k
        z[q] <- sum(val) + w$b[k]
      }
    }
  }

  z
}
```

Notice that this code involves a triple loop, so it will be relatively slow in native R code. Custom libraries, which we discuss in Section 8.9, provide fast implementations of the convolution operations.

The backpropagation code is where the real work of the convolutional neural network happens. The top layers proceed as before, but on the first layer we need to add up the contributions from each output to the weights $w^1_{m,n}$. We also need to store the gradient of the bias terms as this is no longer trivially equal to the gradient with respect to the terms z.

```
# Apply backward propagation algorithm for a CNN.
#
# Args:
#     x: A numeric vector representing one row of the input.
#     y: A numeric vector representing one row of the response.
#     weights: A list created by casl_nn_make_weights.
#     f_obj: Output of the function casl_nn_forward_prop.
#     sigma_p: Derivative of the activation function.
#
# Returns:
#     A list containing the new weighted responses (z) and
#     activations (a).
casl_cnn_backward_prop <-
function(x, y, weights, f_obj, sigma_p)
{
  z <- f_obj$z; a <- f_obj$a
  L <- length(weights)
  grad_z <- vector("list", L)
  grad_w <- vector("list", L)
```

```
  for (j in rev(seq_len(L)))
  {
    if (j == L)
    {
      grad_z[[j]] <- a[[j]] - y
    } else {
      grad_z[[j]] <- (t(weights[[j + 1]]$w) %*%
                      grad_z[[j + 1]]) * sigma_p(z[[j]])
    }
    if (j == 1)
    {
      a_j1 <- x

      d1 <- nrow(a_j1) - 2L
      d2 <- ncol(a_j1) - 2L
      d3 <- dim(weights[[j]]$w)[3]
      grad_z_arr <- array(grad_z[[j]],
                          dim=c(d1, d2, d3))
      grad_b <- apply(grad_z_arr, 3, sum)
      grad_w[[j]] <- array(0, dim=c(3, 3, d3))

      for (n in 0:2)
      {
        for (m in 0:2)
        {
          for (k in seq_len(d3))
          {
            val <- grad_z_arr[, , k] * x[seq_len(d1) + n,
                                         seq_len(d2) + m]
            grad_w[[j]][n + 1L, m + 1L, k] <- sum(val)
          }
        }
      }

    } else {
      a_j1 <- a[[j - 1L]]
      grad_w[[j]] <- grad_z[[j]] %*% t(a_j1)
    }

  }

  list(grad_z=grad_z, grad_w=grad_w, grad_b=grad_b)
}
```

Notice that the code is simplified by constructing an array version of the gradient, `grad_z_arr`, while working on the first layer parameters.

Finally, we put these parts together in the stochastic gradient descent function. Care needs to be taken to correctly update the bias terms in the first layer. Thankfully, R's vector notation allows us to write one block of code can be used to update the weights regardless of whether they are stored as an array (the convolutional layer) or a matrix (the dense layers).

```
# Apply stochastic gradient descent (SGD) to a CNN model.
#
# Args:
#     X: A numeric data matrix.
#     y: A numeric vector of responses.
#     sizes: A numeric vector giving the sizes of layers in
#            the neural network.
#     epochs: Integer number of epochs to computer.
#     eta: Positive numeric learning rate.
#     mu: Non-negative momentum term.
#     l2: Non-negative penalty term for l2-norm.
#     weights: Optional list of starting weights.
#
# Returns:
#     A list containing the trained weights for the network.
casl_cnn_sgd <-
function(X, y, sizes, epochs, rho, mu=0, l2=0, weights=NULL) {

  if (is.null(weights))
  {
    weights <- casl_cnn_make_weights(sizes)
  }

  for (epoch in seq_len(epochs))
  {
    for (i in seq_len(nrow(X)))
    {
      f_obj <- casl_cnn_forward_prop(X[i,,], weights,
                                     casl_util_ReLU)
      b_obj <- casl_cnn_backward_prop(X[i,,], y[i,], weights,
                                      f_obj, casl_util_ReLU_p)

      for (j in seq_along(b_obj))
      {
        grad_b <- if(j == 1) b_obj$grad_b else b_obj$grad_z[[j]]
        weights[[j]]$b <- weights[[j]]$b -
                          rho * grad_b
        weights[[j]]$v <- mu * weights[[j]]$v -
```

```
                            rho * b_obj$grad_w[[j]]
        weights[[j]]$w <- (1 - rho * l2) *
                            weights[[j]]$w +
                                 weights[[j]]$v
      }
    }
  }

  weights
}
```

Note too that we have to be careful to now index the input `X` as a three-dimensional array (one dimension for the samples and two for the spatial dimensions). We will also write a prediction function for the convolutional network.

```
# Predict values from training a CNN.
#
# Args:
#     weights: List of weights describing the neural network.
#     X_test: A numeric data matrix for the predictions.
#
# Returns:
#     A matrix of predicted values.
casl_cnn_predict <-
function(weights, X_test)
{

  p <- length(weights[[length(weights)]]$b)
  y_hat <- matrix(0, ncol=p, nrow=nrow(X_test))
  for (i in seq_len(nrow(X_test)))
  {
    a <- casl_cnn_forward_prop(X_test[i, , ], weights,
                               casl_util_ReLU)$a
    y_hat[i, ] <- a[[length(a)]]
  }

  y_hat
}
```

The only differences here are the indices on `X` and the particular variant of the forward propagation code applied.

To verify that our convolutional neural network implementation is reasonable, we will apply it to our small MNIST classification task. We can now do proper multiclass classification, so the first step is to construct a response matrix `y_mnist` with two columns.

```
y_mnist <- mnist$class[mnist$class %in% c(0, 1)]
y_mnist <- cbind(1 - y_mnist, y_mnist)
y_train <- y_mnist[1:6000,]
y_valid <- y_mnist[6001:12000,]
```

In the convolutional neural network, we now need to keep the dataset `X_train` as a three-dimensional array.

```
X_train <- X_mnist[seq_len(6000), , ]
X_valid <- X_mnist[seq(6001, 12000), , ]
```

We then pass the training data directly to the neural network training function `casl_cnn_sgd`. Our network includes 5 kernels and one hidden dense layer with 10 neurons. The output layer has two neurons to match the number of columns in `y_train`.

```
out <- casl_cnn_sgd(X_train, y_train,
                    c(5 * 5, 5, 10, 2),
                    epochs=5L, rho=0.003)
pred <- casl_cnn_predict(out, X_valid)
table(pred[, 2] > 0.5, y_valid[, 2])
```

```
           0    1
  FALSE 2962    3
  TRUE    13 3022
```

The results of the model on the validation set show only 16 misclassified points out of a total 6000, an impressive result that greatly improves on the dense neural network from Section 8.5. The predictive power is particularly impressive given that we are working only with images containing 49 total pixels.

The output of the first convolutional layer will be significantly larger, by a factor of K, than the input image. In our small test case this is a not a concern. As we consider larger input images and a corresponding increase in the number of kernels, this can quickly become an issue. The solution is to include *pooling* layers into the neural network. These pooling operators reduce the resolution of the image after applying a convolution. Most typically, they result in a halving of the width and height of the data. Pooling can be performed by grouping the pixels into 2-by-2 squares and either taking the average of the activations (*mean-pooling*) or the maximum of the activations (*max-pooling*). Backpropagation can be applied to these pooling layers by summing the derivatives over the pool for mean-pooling or assigning the derivative to the pixel with the largest intensity for max-pooling. A common pattern in convolutional neural networks involves applying a convolutional layer following by a pooling layer several times, producing many filters that capture high-level features with a relatively low resolution. We will see an example of this in our application in Section 8.10.

8.9 Implementation and notes

In this chapter we have seen that it is relatively easy to implement neural networks using just basic matrix operations in R. If we want to work with larger datasets, this approach will only carry us so far. The code will run relatively slowly as it executes many nested loops. It also cannot as written take advantage of faster GPU implementations, which are highly optimized for computing tensor products and utilized in almost all research papers published with neural networks. Perhaps most critically, the code as written needs to be completely re-factored whenever we make any minor change to the architecture other than the number of hidden layers and neurons. Fortunately, there are several libraries purpose-built for building neural networks.

The R package **tensorflow** [7] provides access to the TensorFlow library [1]. This is achieved by calling the corresponding Python library by the same name. TensorFlow provides the low-level functionality for working efficiently with multidimensional arrays and a generic form of backpropagation. Models written in TensorFlow are compiled directly to machine code, and can be optimized with CPU or GPU processors. Keras is a higher-level library built on top of TensorFlow. It is available in R through the **keras** package. This library, which we will make use of in the following applications, allows for building neural networks out of layer objects. The corresponding backpropagation algorithm is computing automatically and compiled into fast machine code through TensorFlow. It provides support for many common tweaks to the basic neural network framework, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

8.10 Application: Image classification with EMNIST

8.10.1 Data and setup

Here we will work with the EMNIST dataset. This is a recent addition to the MNIST classification data. In place of hand-written digits the EMNIST data consists of handwritten examples of the 26 letters in the Latin alphabet. Both upper and lower case letters are included, but these are combined together into 26 classes. The goal is to use the pixel intensities (a number between 0 and 1) over a 28-by-28 grid to classify the letter.

The data comes in two different parts. This first simply indicates the identity of the letter and whether it is in the training set or the testing set.

```
emnist <- read.csv("data/emnist.csv", as.is=TRUE)
head(emnist)
```

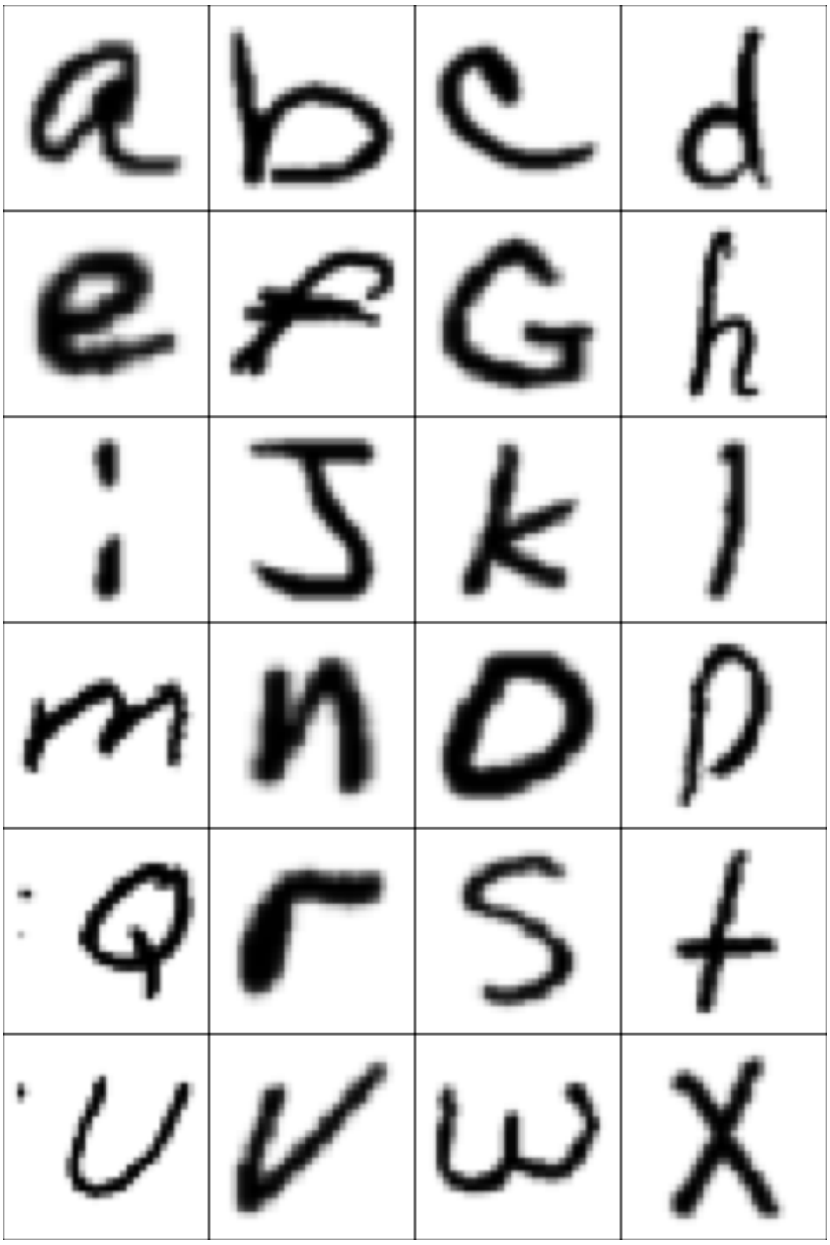

FIGURE 8.10: Example images from the EMNIST dataset, in alphabetical order (y and z not shown). The images are usually coded as white on a black background; we have flipped them to look best in print.

```
     obs_id train_id class class_name
1 id_000001     test     6          g
2 id_000002    train     9          j
3 id_000003    valid    14          o
4 id_000004    train    23          x
5 id_000005    train     5          f
6 id_000006    valid    23          x
```

The actual pixel data is contained in a four-dimensional array.

```
x28 <- readRDS("data/emnist_x28.rds")
dim(x28)
```

```
[1] 124800     28     28      1
```

The first dimension indicates the samples; there are over 120 thousand observations in the dataset. The next two dimensions indicate the height and width of the image. In the final dimension, we simply indicate that this image is black and white rather than color (a color image would have three channels in the fourth position). A plot showing some example letters is given in Figure 8.10.

The first step in setting up the data is to convert the categorical variable `class` into a matrix with 26 columns. We will make use of the **keras** function `to_categorical`; notice that it expects the first category to be zero.

```
library(keras)
Y <- to_categorical(emnist$class, num_classes=26L)
emnist$class[seq_len(12)]
```

```
 [1]  6  9 14 23  5 23 24 17 16  1  5  4
```

The first few rows and columns of resulting matrix should be as expected given the first 12 categories.

```
Y[seq_len(12), seq_len(10)]
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    0    0    0    0    0    1    0    0     0
 [2,]    0    0    0    0    0    0    0    0    0     1
 [3,]    0    0    0    0    0    0    0    0    0     0
 [4,]    0    0    0    0    0    0    0    0    0     0
 [5,]    0    0    0    0    0    1    0    0    0     0
 [6,]    0    0    0    0    0    0    0    0    0     0
 [7,]    0    0    0    0    0    0    0    0    0     0
 [8,]    0    0    0    0    0    0    0    0    0     0
 [9,]    0    0    0    0    0    0    0    0    0     0
```

```
[10,]    0    1    0    0    0    0    0    0    0    0
[11,]    0    0    0    0    0    1    0    0    0    0
[12,]    0    0    0    0    1    0    0    0    0    0
```

Next, we need to flatten the pixel data `x28` into a matrix. This is achieved by applying the `cbind` function of the rows to the array. We then split the data into training and testing sets.

```
X <- t(apply(x28, 1, cbind))

X_train <- X[emnist$train_id == "train",]
X_valid <- X[emnist$train_id != "train",]
Y_train <- Y[emnist$train_id == "train",]
Y_valid <- Y[emnist$train_id != "train",]
```

The **keras** library will allow us to give a validation set in addition to the training set. This allows users to observe how well the model is doing during training, which can take a while. This aids in the process of early stopping as we can kill the training if the model appears to start overfitting.

8.10.2 A shallow network

Our first task will be to fit a shallow neural network without any hidden nodes. This will help to explain the basic functionality of the **keras** package and allow us to visualize the learned weights. The first step in building a neural network is to call the function `keras_model_sequential`. This constructs an empty model that we can then add layers to.

```
model <- keras_model_sequential()
```

Next, we add layers to the model using the `%>%` function. Here we just add one dense layer with one neuron per column in the output Y. We also need to specify that the input matrix has 28^2 columns and that we want to apply the softmax activation to the top layer.

```
model %>%
  layer_dense(units=26, input_shape=c(28^2)) %>%
  layer_activation(activation="softmax")
```

Notice that **keras** has an un-R like object-oriented calling structure. We do not need to save the model with the `<-` sign; the object `model` is mutable and changes directly when adding layers. The same calling mechanism works for training. If we manually terminate the SGD algorithm during training, the weights up to that point are not lost.

Next, we need to compile the model using the `compile` function. This is where we specify the loss function (`categorical_crossentropy`), the optimization function (SGD, with an η of 0.01 and momentum term of 0.8), and what metrics we want printed with the result.

```
model %>% compile(loss='categorical_crossentropy',
              optimizer=optimizer_sgd(lr=0.01,
                                      momentum=0.80),
              metrics=c('accuracy'))
```

The model has now been compiled to machine code. Options exist within the R package to compile for GPU architectures if these are available.

Finally, we run the function `fit` on the model to train the weights on the training data. We specify the number of epochs and also pass the the validation data.

```
history <- model %>%
  fit(X_train, Y_train, epochs=10,
      validation_data=list(X_valid, Y_valid))
```

After fitting the model, we can then run the `predict_classes` to get the predicted classes on the entire dataset `X`.

```
emnist$predict <- predict_classes(model, X)
tapply(emnist$predict == emnist$class, emnist$train_id,
       mean)
```

```
    train     valid
0.7259135 0.7122115
```

The accuracy rate here is over 70%, which is actually quite good given the lack of any hidden layers in the network. We can visualize the learned weights for each letter, as shown in Figure 8.11. Notice that the positive weights often seem to trace features of each letter.

8.10.3 A deeper network

To achieve better results we need to use a larger and deeper neural network. The flexibility of the **keras** library makes this easy to code, though of course the algorithm takes significantly longer to run. Here we build a neural network with 4 hidden layers all having 128 neurons. Each layer is followed by a rectified linear unit and dropout with a probability of 25%.

```
model <- keras_model_sequential()
model %>%
  layer_dense(128, input_shape=c(28^2)) %>%
  layer_activation(activation="relu") %>%
  layer_dropout(rate=0.25) %>%

  layer_dense(128, input_shape=c(28^2)) %>%
  layer_activation(activation="relu") %>%
```

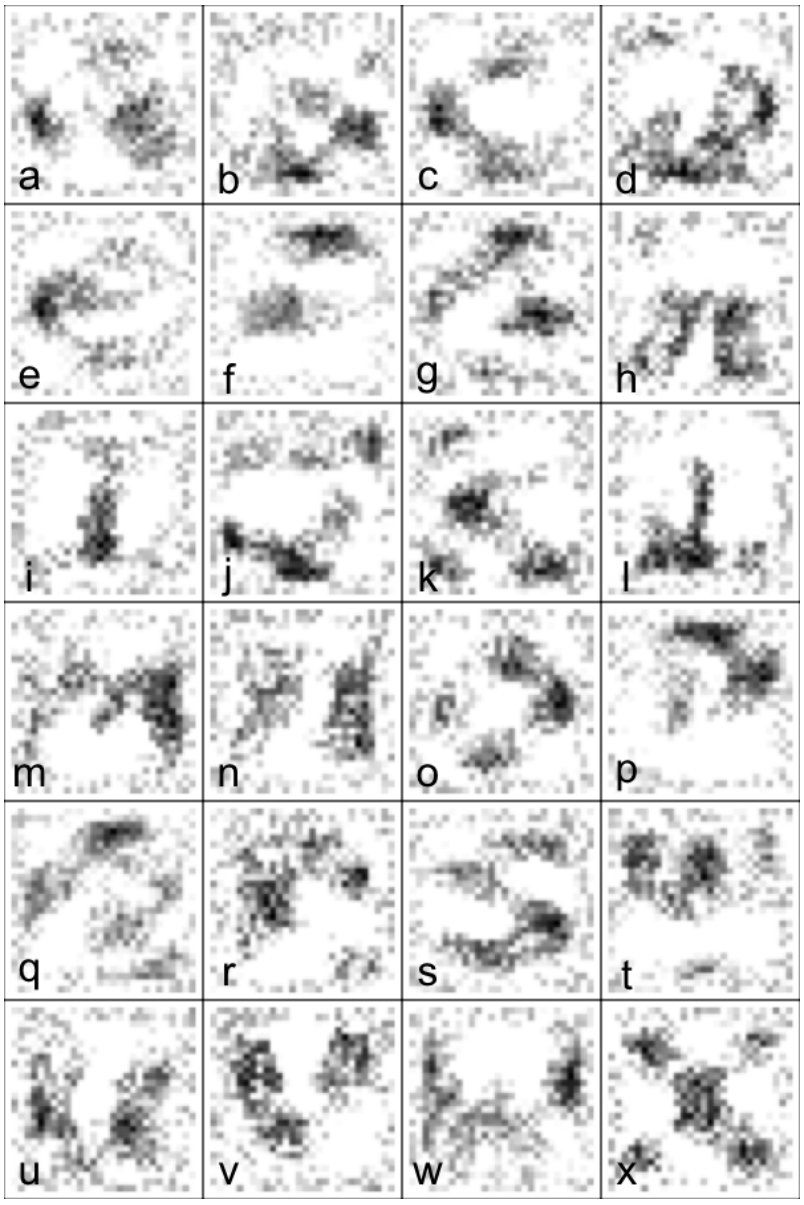

FIGURE 8.11: Visualization of positive weights in a neural network with no hidden layers. The dark pixels indicate strong positive weights and white pixels indicate zero or negative weights. Notice that the positive weights seem to sketch out critical features of the letters, with some (such as M, S, U, and X) sketching the entire letter shape.

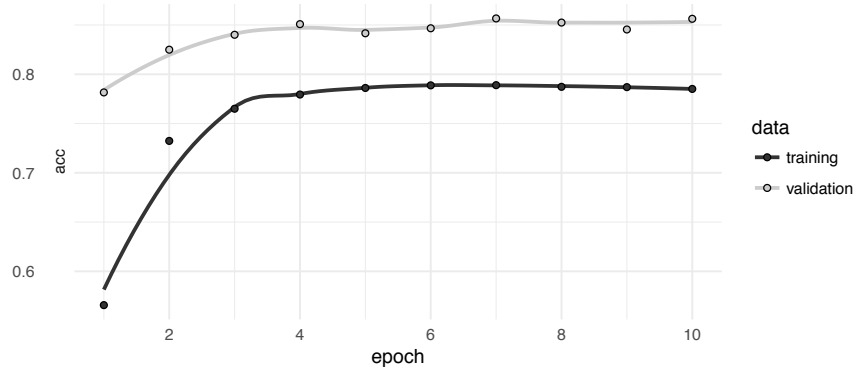

FIGURE 8.12: Accuracy of the dense neural network on the training and validation sets at the end of each epoch in the SGD algorithm. The training accuracy is lower than the validation accuracy due to dropout being turned on in the former but removed in the latter.

```
  layer_dropout(rate=0.25) %>%

  layer_dense(128, input_shape=c(28^2)) %>%
  layer_activation(activation="relu") %>%
  layer_dropout(rate=0.25) %>%

  layer_dense(128, input_shape=c(28^2)) %>%
  layer_activation(activation="relu") %>%
  layer_dropout(rate=0.25) %>%

  layer_dense(units=26) %>%
  layer_activation(activation="softmax")

model %>% compile(loss='categorical_crossentropy',
                  optimizer=optimizer_rmsprop(lr=0.001),
                  metrics=c('accuracy'))

history <- model %>%
  fit(X_train, Y_train, epochs=10,
      validation_data=list(X_valid, Y_valid))
```

Figure 8.12 shows the training and validation accuracy during training. The validation accuracy is better due to the fact that dropout is turned off for the validation set, but not for the training set during training.

The final model improves on the shallow network, achieving an accuracy of around 85% on the testing set.

```
emnist$predict <- predict_classes(model, X)
tapply(emnist$predict == emnist$class, emnist$train_id,
       mean)
```

```
    train     valid
0.8754968 0.8570192
```

Looking at the most confused classes, we see that just a few pairs of letters are causing most of the problems.

```
emnist$predict <- letters[emnist$predict + 1]
tab <- dplyr::count(emnist, train_id, predict, class_name,
                    sort = TRUE)
tab <- tab[tab$train_id == "valid" &
           tab$predict != tab$class_name,]
tab
```

```
   train_id predict class_name    n
 1 valid    i       l          566
 2 valid    q       g          276
 3 valid    l       i           96
 4 valid    e       c           69
 5 valid    o       d           69
 6 valid    u       v           67
 7 valid    g       q           59
 8 valid    o       a           59
 9 valid    v       y           59
10 valid    o       q           57
```

Distinguishing between 'i' and 'l' (probably the upper case version of the first for the lower case version of the second) and 'q' and 'g' seem to be particularly difficult.

8.10.4 A convolutional neural network

In order to improve our predictive model further, we will need to employ convolutional neural networks. Thankfully, this is relatively easy to do in **keras**. To start, we will now need to have the data X un-flattened:

```
X <- array(x28, dim = c(dim(x28), 1L))
X_train <- X[emnist$train_id == "train", , , , drop=FALSE]
X_valid <- X[emnist$train_id != "train", , , , drop=FALSE]
```

Then, we build a keras model as usual, but use the convolutional layers `layer_conv_2d` and `layer_max_pooling_2d`. Options for these determine the number of filters, the kernel size, and options for padding the input.

FIGURE 8.13: Examples of mis-classified test EMNIST observations by the convolutional neural network.

```
model <- keras_model_sequential()
model %>%
  layer_conv_2d(filters = 32, kernel_size = c(2,2),
                input_shape = c(28, 28, 1),
                padding = "same") %>%
  layer_activation(activation = "relu") %>%
  layer_conv_2d(filters = 32, kernel_size = c(2,2),
              padding = "same") %>%
  layer_activation(activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.5) %>%

  layer_conv_2d(filters = 32, kernel_size = c(2,2),
              padding = "same") %>%
  layer_activation(activation = "relu") %>%
  layer_conv_2d(filters = 32, kernel_size = c(2,2),
              padding = "same") %>%
  layer_activation(activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.5) %>%

  layer_flatten() %>%
  layer_dense(units = 128) %>%
  layer_activation(activation = "relu") %>%
  layer_dense(units = 128) %>%
  layer_activation(activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 26) %>%
  layer_activation(activation = "softmax")
```

Before passing the convolutional input into the dense layers at the top of the network, the layer `layer_flatten` is used to convert the multidimensional input into a two-dimensional output.

Compiling and fitting a convolutional neural network is done exactly the same way that dense neural networks are used in **keras**.

```
model %>% compile(loss = 'categorical_crossentropy',
                  optimizer = optimizer_rmsprop(),
                  metrics = c('accuracy'))

history <- model %>%
  fit(X_train, Y_train, epochs = 10,
      validation_data = list(X_valid, Y_valid))
```

The prediction accuracy is now significantly improved, with a testing accuracy of over 90%.

```
emnist$predict <- predict_classes(model, X)
tapply(emnist$predict == emnist$class, emnist$train_id,
       mean)
```

```
    train     valid
0.9241667 0.9207372
```

Figure 8.13 shows a selection of those images which are still incorrectly classified (known as *negative examples*). While many of these are recognizable at first glance by human readers, many of them are quite unclear with at least two feasible options for the letter.

8.11 Exercises

1. Using the **keras** package functions, use a neural network to predict the tip percentage from the NYC Taxicab dataset in Section 3.6.1. How does this compare to the ridge regression approach?

2. Write a function to check the derivatives of the CNN backpropagation routine from Section 8.8. Does it match the analytic derivatives?

3. The **keras** package contains the function `application_vgg16` that loads the VGG16 model for image classification. Load this model into R and print out the model. In each layer, the number of trainable weights is listed. What proportion of trainable weights is in the convolutional layers? Why is this such a small portion of the entire model? In other words, why do dense layers have many more weights than convolutional layers?

4. Adjust the kernel size, and any other parameters you think are useful, in the convolutional neural network for EMNIST in Section 8.10.4. Can you improve on the classification rate?

5. Implement dropout in the dense neural network implementation from Section 8.4.

6. Change the implementation of backpropagation from Section 8.4 to include a mini-batch of size 16. You can assume that the data size is a multiple of 16.

7. Add an ℓ_1-penalty term in addition to the ℓ_2-penalty term in the code from Section 8.6. You will need to first work out analytically how the updates should be performed.

8. Write a function that uses mean absolute deviation as a loss function, instead of mean squared error. Test the use of this function with a simulation containing several outliers. How well do neural networks and SGD perform when using robust techniques?

9. Rewrite the functions from Section 8.8 to allow for a user supplied kernel size (you may assume that it is square).

10. Implement zero padding in the convolutional neural network implementation from Section 8.8.