18

Neural Networks and Deep Learning

Something happened in the mid 1980s that shook up the applied statistics community. Neural networks (NNs) were introduced, and they marked a shift of predictive modeling towards computer science and machine learning. A neural network is a highly parametrized model, inspired by the architecture of the human brain, that was widely promoted as a *universal approximator*—a machine that with enough data could learn any smooth predictive relationship.

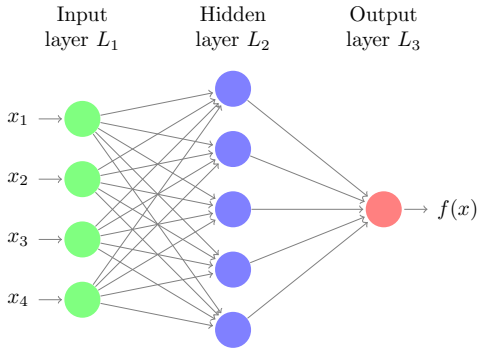

Figure 18.1 Neural network diagram with a single hidden layer. The hidden layer derives transformations of the inputs—nonlinear transformations of linear combinations—which are then used to model the output.

Figure 18.1 shows a simple example of a *feed-forward* neural network diagram. There are four predictors or inputs x_j, five hidden units $a_\ell = g(w_{\ell 0}^{(1)} + \sum_{j=1}^{4} w_{\ell j}^{(1)} x_j)$, and a single output unit $o = h(w_0^{(2)} + \sum_{\ell=1}^{5} w_\ell^{(2)} a_\ell)$. The language associated with NNs is colorful: memory units or *neurons* automatically learn new features from the data through a process called

supervised learning. Each neuron a_l is connected to the input layer via a vector of parameters or *weights* $\{w_{\ell j}^{(1)}\}_1^p$ (the (1) refers to the first layer and ℓj refers to the jth variable and ℓth unit). The intercept terms $w_{\ell 0}^{(1)}$ are called a *bias*, and the function g is a nonlinearity, such as the sigmoid function $g(t) = 1/(1 + e^{-t})$. The idea was that each neuron will learn a simple binary on/off function; the sigmoid function is a smooth and differentiable compromise. The final or output layer also has weights, and an output function h. For quantitative regression h is typically the identity function, and for a binary response it is once again the sigmoid. Note that without the nonlinearity in the hidden layer, the neural network would reduce to a generalized linear model (Chapter 8). Typically neural networks are fit by maximum likelihood, usually with a variety of forms of regularization.

The knee-jerk response from statisticians was "What's the big deal? A neural network is just a nonlinear model, not too different from many other generalizations of linear models."

While this may be true, neural networks brought a new energy to the field. They could be scaled up and generalized in a variety of ways: many hidden units in a layer, multiple hidden layers, weight sharing, a variety of colorful forms of regularization, and innovative learning algorithms for massive data sets. And most importantly, they were able to solve problems on a scale far exceeding what the statistics community was used to. This was part computing scale and expertise, part liberated thinking and creativity on the part of this computer science community. New journals were devoted to the field,[†] and several popular annual conferences (initially at ski resorts) attracted their denizens, and drew in members of the statistics community.

†1

After enjoying considerable popularity for a number of years, neural networks were somewhat sidelined by new inventions in the mid 1990s, such as boosting (Chapter 17) and SVMs (Chapter 19). Neural networks were *passé*. But then they re-emerged with a vengeance after 2010—the reincarnation now being called *deep learning*. This renewed enthusiasm is a result of massive improvements in computer resources, some innovations, and the ideal niche learning tasks such as image and video classification, and speech and text processing.

18.1 Neural Networks and the Handwritten Digit Problem

Neural networks really cut their baby teeth on an optical character recognition (OCR) task: automatic reading of handwritten digits, as in a zipcode. Figure 18.2 shows some examples, taken from the **MNIST** corpus.[†] The †2
idea is to build a classifier $C(x) \in \{0, 1, \ldots, 9\}$ based on the input image $x \in \mathbb{R}^{28\times 28}$, a 28×28 grid of image intensities. In fact, as is often the case, it is more useful to learn the probability function $\Pr(y = j|x)$, $j = 0, 1, 2, \ldots, 9$; this is indeed the target for our neural network. Figure 18.3

Figure 18.2 Examples of handwritten digits from the **MNIST** corpus. Each digit is represented by a 28×28 grayscale image, derived from normalized binary images of different shapes and sizes. The value stored for each pixel in an image is a nonnegative eight-bit representation of the amount of gray present at that location. The 784 pixels for each image are the predictors, and the 0–9 class labels the response. There are 60,000 training images in the full data set, and 10,000 in the test set.

shows a neural network with three hidden layers, a successful configuration for this digit classification problem. In this case the output layer has 10 nodes, one for each of the possible class labels. We use this example to walk the reader through some of the aspects of the configuration of a network, and fitting it to training data. Since all of the layers are functions of their previous layers, and finally functions of the input vector x, the network represents a somewhat complex function $f(x; \mathcal{W})$, where $\mathcal{W}$ represents the entire collection of weights. Armed with a suitable loss function, we could simply barge right in and throw it at our favorite optimizer. In the early days this was not computationally feasible, especially when special

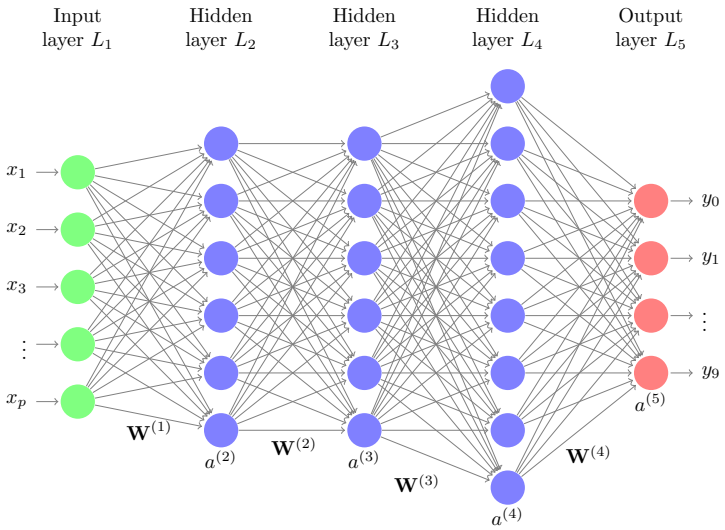

Figure 18.3 Neural network diagram with three hidden layers and multiple outputs, suitable for the MNIST handwritten-digit problem. The input layer has $p = 784$ units. Such a network with hidden layer sizes $(1024, 1024, 2048)$, and particular choices of tuning parameters, achieves the state-of-the art error rate of 0.93% on the "official" test data set. This network has close to four million weights, and hence needs to be heavily regularized.

structure is imposed on the weight vectors. Today there are fairly automatic systems for setting up and fitting neural networks, and this view is not too far from reality. They mostly use some form of gradient descent, and rely on an organization of parameters that leads to a manageable calculation of the gradient.

The network in Figure 18.3 is complex, so it is essential to establish a convenient notation for referencing the different sets of parameters. We continue with the notation established for the single-layer network, but with some additional annotations to distinguish aspects of different layers. From the first to the second layer we have

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j, \tag{18.1}$$

$$a_\ell^{(2)} = g^{(2)}(z_\ell^{(2)}). \tag{18.2}$$

We have separated the linear transformations $z_\ell^{(2)}$ of the x_j from the nonlinear transformation of these, and we allow for layer-specific nonlinear transformations $g^{(k)}$. More generally we have the transition from layer $k-1$ to layer k:

$$z_\ell^{(k)} = w_{\ell 0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{\ell j}^{(k-1)} a_j^{(k-1)}, \tag{18.3}$$

$$a_\ell^{(k)} = g^{(k)}(z_\ell^{(k)}). \tag{18.4}$$

In fact (18.3)–(18.4) can serve for the input layer (18.1)–(18.2) if we adopt the notation that $a_\ell^{(1)} \equiv x_\ell$ and $p_1 = p$, the number of input variables. Hence each of the arrows in Figure 18.3 is associated with a weight parameter.

It is simpler to adopt a vector notation

$$z^{(k)} = \boldsymbol{W}^{(k-1)} a^{(k-1)} \tag{18.5}$$

$$a^{(k)} = g^{(k)}(z^{(k)}), \tag{18.6}$$

where $\boldsymbol{W}^{(k-1)}$ represents the matrix of weights that go from layer L_{k-1} to layer L_k, $a^{(k)}$ is the entire vector of activations at layer L_k, and our notation assumes that $g^{(k)}$ operates elementwise on its vector argument. We have also absorbed the bias parameters $w_{\ell 0}^{(k-1)}$ into the matrix $\boldsymbol{W}^{(k-1)}$, which assumes that we have augmented each of the activation vectors $a^{(k)}$ with a constant element 1.

Sometimes the nonlinearities $g^{(k)}$ at the inner layers are the same function, such as the function σ defined earlier. In Section 18.5 we present a network for natural color image classification, where a number of different activation functions are used.

Depending on the response, the final transformation $g^{(K)}$ is usually special. For M-class classification, such as here with $M = 10$, one typically uses the *softmax* function

$$g^{(K)}(z_m^{(K)}; z^{(K)}) = \frac{e^{z_m^{(K)}}}{\sum_{\ell=1}^{M} e^{z_\ell^{(K)}}}, \tag{18.7}$$

which computes a number (probability) between zero and one, and all M of them sum to one.[1]

[1] This is a symmetric version of the inverse link function used for multiclass logistic regression.

18.2 Fitting a Neural Network

As we have seen, a neural network model is a complex, hierarchical function $f(x; \mathcal{W})$ of the the feature vector x, and the collection of weights $\mathcal{W}$. For typical choices for the $g^{(k)}$, this function will be differentiable. Given a training set $\{x_i, y_i\}_1^n$ and a loss function $L[y, f(x)]$, along familiar lines we might seek to solve

$$\underset{\mathcal{W}}{\text{minimize}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f(x_i; \mathcal{W})] + \lambda J(\mathcal{W}) \right\}, \tag{18.8}$$

where $J(\mathcal{W})$ is a nonnegative regularization term on the elements of $\mathcal{W}$, and $\lambda \geq 0$ is a tuning parameter. (In practice there may be multiple regularization terms, each with their own λ.) For example an early popular penalty is the quadratic

$$J(\mathcal{W}) = \frac{1}{2} \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{\ell=1}^{p_{k+1}} \left\{ w_{\ell j}^{(k)} \right\}^2, \tag{18.9}$$

as in ridge regression (7.41). Also known as the *weight-decay* penalty, it pulls the weights toward zero (typically the biases are not penalized). Lasso penalties (Chapter 16) are also popular, as are mixtures of these (an elastic net).

For binary classification we could take L to be binomial deviance (8.14), in which case the neural network amounts to a penalized logistic regression, Section 8.1, albeit a highly parametrized and penalized one. Loss functions are usually convex in f, but not in the elements of $\mathcal{W}$, so solving (18.8) is difficult, and at best we seek good local optima. Most methods are based on some form of gradient descent, with many associated bells and whistles. We briefly discuss some elements of the current practice in finding good solutions to (18.8).

Computing the Gradient: Backpropagation

The elements of $\mathcal{W}$ occur in layers, since $f(x; \mathcal{W})$ is defined as a series of compositions, starting from the input layer. Computing the gradient is also done most naturally in layers (the chain rule for differentiation; see for example (18.10) in Algorithm 18.1 below), and our notation makes this easier to describe in a recursive fashion. We will consider computing the derivative of $L[y, f(x; \mathcal{W}]$ with respect to any of the elements of $\mathcal{W}$, for a generic input–output pair x, y; since the loss part of the objective is a sum,

the overall gradient will be the sum of these individual gradient elements over the training pairs (x_i, y_i).

The intuition is as follows. Given a training generic pair (x, y), we first make a forward pass through the network, which creates activations at each of the nodes $a_\ell^{(k)}$ in each of the layers, including the final output layer. We would then like to compute an error term $\delta_\ell^{(k)}$ that measures the responsibility of each node for the error in predicting the true output y. For the output activations $a_\ell^{(K)}$ these errors are easy: either residuals or generalized residuals, depending on the loss function. For activations at inner layers, $\delta_\ell^{(k)}$ will be a weighted sum of the errors terms of nodes that use $a_\ell^{(k)}$ as inputs. The *backpropagation* Algorithm 18.1 gives the details for computing the gradient for a single input–output pair x, y. We leave it to the reader to verify that this indeed implements the chain rule for differentiation.

Algorithm 18.1 BACKPROPAGATION

1 Given a pair x, y, perform a "feedforward pass," computing the activations $a_\ell^{(k)}$ at each of the layers $L_2, L_3, \ldots, L_K$; i.e. compute $f(x; \mathcal{W})$ at x using the current $\mathcal{W}$, saving each of the intermediary quantities along the way.

2 For each output unit ℓ in layer L_K, compute

$$\begin{aligned}\delta_\ell^{(K)} &= \frac{\partial L[y, f(x, \mathcal{W})]}{\partial z_\ell^{(K)}} \\ &= \frac{\partial L[y, f(x; \mathcal{W})]}{\partial a_\ell^{(K)}} \dot{g}^{(K)}(z_\ell^{(K)}),\end{aligned} \tag{18.10}$$

where $\dot{g}$ denotes the derivative of $g(z)$ wrt z. For example for $L(y, f) = \frac{1}{2}\|y - f\|_2^2$, (18.10) becomes $-(y_\ell - f_\ell) \cdot \dot{g}^{(K)}(z_\ell^{(K)})$.

3 For layers $k = K-1,\ K-2, \ldots, 2$, and for each node ℓ in layer k, set

$$\delta_\ell^{(k)} = \left(\sum_{j=1}^{p_{k+1}} w_{j\ell}^{(k)} \delta_j^{(k+1)} \right) \dot{g}^{(k)}(z_\ell^{(k)}). \tag{18.11}$$

4 The partial derivatives are given by

$$\frac{\partial L[y, f(x; \mathcal{W})]}{\partial w_{\ell j}^{(k)}} = a_j^{(k)} \delta_\ell^{(k+1)}. \tag{18.12}$$

One again matrix–vector notation simplifies these expressions a bit:

(18.10) becomes (for squared-error loss)

$$\delta^{(K)} = -(y - a^{(K)}) \circ \dot{g}^{(K)}(z^{(K)}), \tag{18.13}$$

where $\circ$ denotes the Hadamard (elementwise) product;
(18.11) becomes

$$\delta^{(k)} = \left(\boldsymbol{W}^{(k)\prime}\delta^{(k+1)}\right) \circ \dot{g}^{(k)}(z^{(k)}); \tag{18.14}$$

(18.12) becomes

$$\frac{\partial L[y, f(x; \mathcal{W})]}{\partial \boldsymbol{W}^{(k)}} = \delta^{(k+1)} a^{(k)\prime}. \tag{18.15}$$

Backpropagation was considered a breakthrough in the early days of neural networks, since it made fitting a complex model computationally manageable.

Gradient Descent

Algorithm 18.1 computes the gradient of the loss function at a single generic pair (x, y); with n training pairs the gradient of the first part of (18.8) is given by

$$\Delta \boldsymbol{W}^{(k)} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L[y_i, f(x_i; \mathcal{W}]}{\partial \boldsymbol{W}^{(k)}}. \tag{18.16}$$

With the quadratic form (18.9) for the penalty, a gradient-descent update is

$$\boldsymbol{W}^{(k)} \leftarrow \boldsymbol{W}^{(k)} - \alpha \left(\Delta \boldsymbol{W}^{(k)} + \lambda \boldsymbol{W}^{(k)}\right), \; k = 1, \ldots, K-1, \tag{18.17}$$

where $\alpha \in (0, 1]$ is the *learning rate*.

Gradient descent requires starting values for all the weights $\mathcal{W}$. Zero is not an option, because each layer is symmetric in the weights flowing to the different neurons, hence we rely on starting values to break the symmetries. Typically one would use random starting weights, close to zero; random uniform or Gaussian weights are common.

There are a multitude of "tricks of the trade" in fitting or "learning" a neural network, and many of them are connected with gradient descent. Here we list some of these, without going into great detail.

Stochastic Gradient Descent

Rather than process all the observations before making a gradient step, it can be more efficient to process smaller batches at a time—even batches

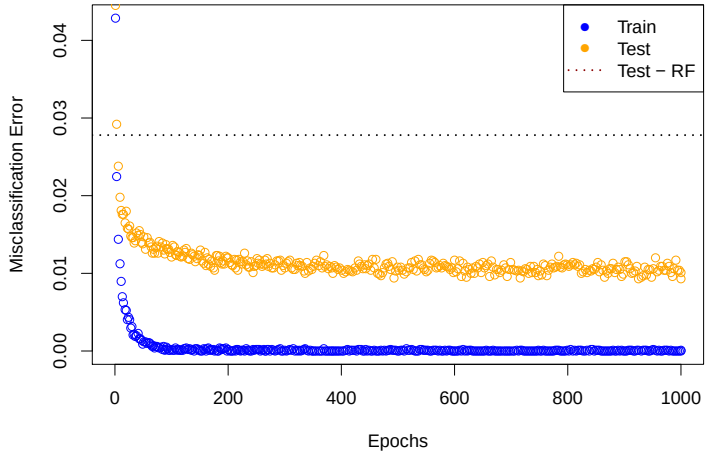

Figure 18.4 Training and test misclassification error as a function of the number of epochs of training, for the `MNIST` digit classification problem. The architecture for the network is shown in Figure 18.3. The network was fit using accelerated gradient descent with adaptive rate control, a rectified linear activation function, and dropout regularization (Section 18.5). The horizontal broken line shows the error rate of a random forest (Section 17.1). A logistic regression model (Section 8.1) achieves only 0.072 (off the scale).

of size one! These batches can be sampled at random, or systematically processed. For large data sets distributed on multiple computer cores, this can be essential for reasons of efficiency. An *epoch* of training means that all n training samples have been used in gradient steps, irrespective of how they have been grouped (and hence how many gradient steps have been made).

Accelerated Gradient Methods

The idea here is to allow previous iterations to build up momentum and influence the current iterations. The iterations have the form

$$\mathcal{V}_{t+1} = \mu\mathcal{V}_t - \alpha(\Delta\mathcal{W}_t + \lambda\mathcal{W}_t), \tag{18.18}$$

$$\mathcal{W}_{t+1} = \mathcal{W}_t + \mathcal{V}_{t+1}, \tag{18.19}$$

using $\mathcal{W}_t$ to represent the entire collection of weights at iteration t. $\mathcal{V}_t$ is a *velocity vector* that accumulates gradient information from previous iterations, and is controlled by an additional momentum parameter μ. When correctly tuned, accelerated gradient descent can achieve much faster convergence rates; however, tuning tends to be a difficult process, and is typically done adaptively.

Rate Annealing

A variety of creative methods have been proposed to adapt the learning rate to avoid jumping across good local minima. These tend to be a mixture of principled approaches combined with ad-hoc adaptations that tend to work well in practice.[†] Figure 18.4 shows the performance of our neural net on the `MNIST` digit data. This achieves state-of-the art misclassification error rates on these data (just under 0.093% errors), and outperforms random forests (2.8%) and a generalized linear model (7.2%). Figure 18.5 shows the 93 misclassified digits.

†3

Figure 18.5 All 93 misclassified digits in the `MNIST` test set. The true digit class is labeled in blue, the predicted in red.

Other Tuning Parameters

Apart from the many details associated with gradient descent, there are several other important structural and operational aspects of neural networks that have to be specified.

Number of Hidden Layers, and Their Sizes

With a single hidden layer, the number of hidden units determines the number of parameters. In principle, one could treat this number as a tuning parameter, which could be adjusted to avoid overfitting. The current collective wisdom suggests it is better to have an abundant number of hidden units, and control the model complexity instead by weight regularization. Having deeper networks (more hidden layers) increases the complexity as well. The correct number tends to be task specific; having two hidden layers with the digit recognition problem leads to competitive performance.

Choice of Nonlinearities

There are a number of activation functions $g^{(k)}$ in current use. Apart from the sigmoid function, which transforms its input to a values in $(0, 1)$, other popular choices are

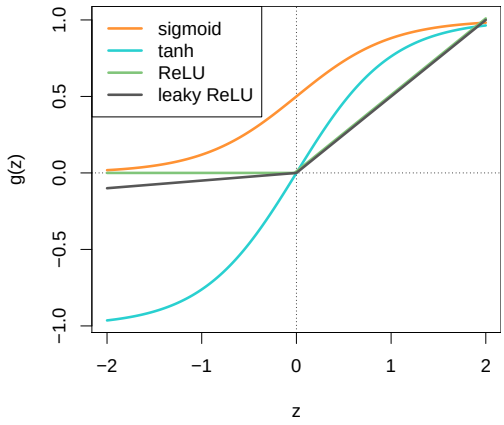

Figure 18.6 Activation functions. ReLU is a rectified linear (unit).

$$\text{tanh:} \quad g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

which delivers values in $(-1, 1)$.

$$\text{rectified linear:} \quad g(z) = z_+,$$

or the positive-part function. This has the advantage of making the gradient computations cheaper to compute.

$$\text{leaky rectified linear:} \quad g_\alpha(z) = z_+ - \alpha z_-,$$

for α nonnegative and close to zero. The rectified linear tends to have flat spots, because of the many zero activations; this is an attempt to avoid these and the accompanying zero gradients.

Choice of Regularization

Typically this is a mixture of ℓ_2 and ℓ_1 regularization, each of which requires a tuning parameter. As in lasso and regression applications, the bias terms (intercepts) are usually not regularized. The weight regularization is typically light, and serves several roles. The ℓ_2 reduces problems with collinearity, the ℓ_1 can ignore irrelevant features, and both slow the rate of overfitting, especially with deep (over-parametrized) networks.

Early Stopping

Neural nets are typically over-parametrized, and hence are prone to overfitting. Originally early stopping was set up as the primary tuning parameter, and the stopping time was determined using a held-out set of validation data. In modern networks the regularization is tuned adaptively to avoid overfitting, and hence it is less of a problem. For example, in Figure 18.4 we see that the test misclassification error has flattened out, and does not rise again with increasing number of epochs.

18.3 Autoencoders

An autoencoder is a special neural network for computing a type of nonlinear principal-component decomposition.

The linear principal component decomposition is a popular and effective linear method for reducing a large set of correlated variables to a typically smaller number of linear combinations that capture most of the variance in the original set. Hence, given a collection of n vectors $x_i \in \mathbb{R}^p$ (assumed to have mean zero), we produce a derived set of uncorrelated features $z_i \in \mathbb{R}^q$

($q \leq p$, and typically smaller) via $z_i = \boldsymbol{V}' x_i$. The columns of $\boldsymbol{V}$ are orthonormal, and are derived such that the first component of z_i has maximal variance, the second has the next largest variance and is uncorrelated with the first, and so on. It is easy to show that the columns of $\boldsymbol{V}$ are the leading q eigenvectors of the sample covariance matrix $\boldsymbol{S} = \frac{1}{n}\boldsymbol{X}'\boldsymbol{X}$.

Principal components can also be derived in terms of a best-approximating linear subspace, and it is this version that leads to the nonlinear generalization presented here. Consider the optimization problem

$$\underset{\boldsymbol{A}\in\mathbb{R}^{p\times q},\, \{\gamma_i\}_1^n\in\mathbb{R}^{q\times n}}{\text{minimize}} \sum_{i=1}^{n} \|x_i - \boldsymbol{A}\gamma_i\|_2^2, \tag{18.20}$$

for $q < p$. The subspace is defined by the column space of $\boldsymbol{A}$, and for each point x_i we wish to locate its best approximation in the subspace (in terms of Euclidean distance). Without loss of generality, we can assume $\boldsymbol{A}$ has orthonormal columns, in which case $\hat{\gamma}_i = \boldsymbol{A}' x_i$ for each i (n separate linear regressions). Plugging in, (18.20) reduces to

$$\underset{\boldsymbol{A}\in\mathbb{R}^{p\times q},\, \boldsymbol{A}'\boldsymbol{A}=\boldsymbol{I}_q}{\text{minimize}} \sum_{i=1}^{n} \|x_i - \boldsymbol{A}\boldsymbol{A}' x_i\|_2^2. \tag{18.21}$$

A solution is given by $\hat{\boldsymbol{A}} = \boldsymbol{V}$, the matrix above of the first q principal-component direction vectors computed from the x_i. By analogy, a single-layer autoencoder solves a nonlinear version of this problem:

$$\underset{\boldsymbol{W}\in\mathbb{R}^{q\times p}}{\text{minimize}} \sum_{i=1}^{n} \|x_i - \boldsymbol{W}' g(\boldsymbol{W} x_i)\|_2^2, \tag{18.22}$$

for some nonlinear activation function g; see Figure 18.7 (left panel). If g is the identity function, these solutions coincide (with $\boldsymbol{W} = \boldsymbol{V}'$).

Figure 18.7 (right panel) represents the learned row of $\boldsymbol{W}$ as images, when the autoencoder is fit to the **MNIST** digit database. Since autoencoders do not require a response (the class labels in this case), this decomposition is unsupervised. It is often expensive to label images, for example, while unlabeled images are abundant. Autoencoders provide a means for extracting potentially useful features from such data, which can then be used with labeled data to train a classifier. In fact, they are often used as *warm starts* for the weights when fitting a supervised neural network.

Once again there are a number of bells and whistles that make autoencoders more effective.

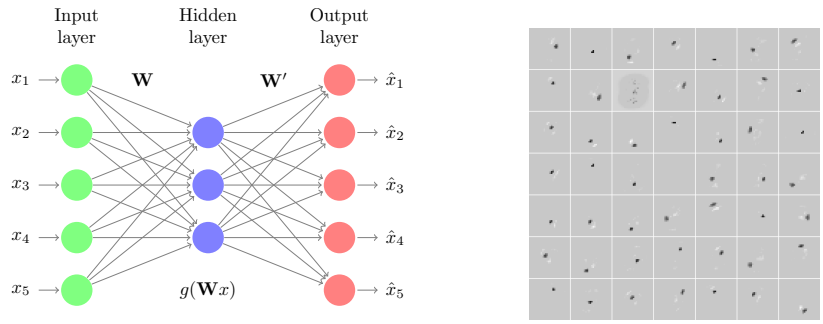

Figure 18.7 Left: Network representation of an autoencoder used for unsupervised learning of nonlinear principal components. The middle layer of hidden units creates a bottleneck, and learns nonlinear representations of the inputs. The output layer is the transpose of the input layer, so the network tries to reproduce the input data using this restrictive representation. Right: Images representing the estimated rows of $\boldsymbol{W}$ using the `MNIST` database; the images can be seen as filters that detect local gradients in the image pixels. In each image, most of the weights are zero, and the nonzero weights are localized in the two-dimensional image space.

- ℓ_1 regularization applied to the rows of $\boldsymbol{W}$ lead to sparse weight vectors, and hence local features, as was the case in our example.
- Denoising is a process where noise is added to the input layer (but not the output), resulting in features that do not focus on isolated values, such as pixels, but instead have some *volume*. We discuss denoising further in Section 18.5.
- With regularization, the bottleneck is not necessary, as in the figure or in principal components. In fact we can learn many more than p components.
- Autoencoders can also have multiple layers, which are typically learned sequentially. The activations learned in the first layer are treated as the input (and output) features, and a model like (18.22) is fit to them.

18.4 Deep Learning

Neural networks were reincarnated around 2010 with "deep learning" as a flashier name, largely a result of much faster and larger computing systems, plus a few new ideas. They have been shown to be particularly successful

in the difficult task of classifying natural images, using what is known as a convolutional architecture. Initially autoencoders were considered a crucial aspect of deep learning, since unlabeled images are abundant. However, as labeled corpora become more available, the word on the street is that supervised learning is sufficient.

Figure 18.8 shows examples of natural images, each with a class label such as `beaver`, `sunflower`, `trout` etc. There are 100 class labels in

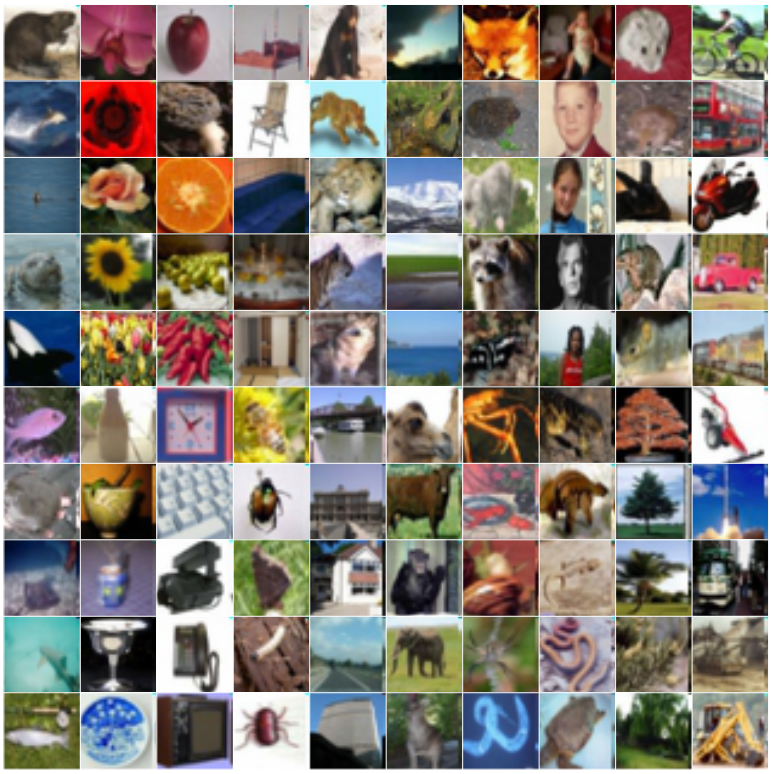

Figure 18.8 Examples of natural images. The `CIFAR-100` database consists of 100 color image classes, with 600 examples in each class (500 train, 100 test). Each image is $32 \times 32 \times 3$ (red, green, blue). Here we display a randomly chosen image from each class. The classes are organized by hierarchical structure, with 20 coarse levels and five subclasses within each. So, for example, the first five images in the first column are `aquatic mammals`, namely `beaver`, `dolphin`, `otter`, `seal` and `whale`.

all, and 500 training images and 100 test images per class. The goal is to build a classifier to assign a label to an image. We present the essential details of a deep-learning network for this task—one that achieves a respectable classification performance of 35% errors on the designated test set.[2] Figure 18.9 shows a typical deep-learning architecture, with many

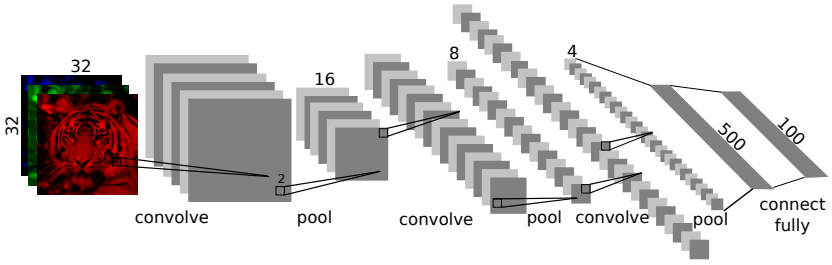

Figure 18.9 Architecture of a deep-learning network for the `CIFAR-100` image classification task. The input layer and hidden layers are all represented as images, except for the last hidden layer, which is "flattened" (vectorized). The input layer consists of the $p_1 = 3$ color (red, green, and blue) versions of an input image (unlike earlier, here we use the p_k to refer to the number of images rather than the totality of pixels). Each of these color panes is 32×32 pixels in dimension. The first hidden layer computes a convolution using a bank of p_2 distinct $q \times q \times p_1$ learned filters, producing an array of images of dimension $p_2 \times 32 \times 32$. The next *pool* layer reduces each non-overlapping block of $\ell \times \ell$ numbers in each pane of the first hidden layer to a single number using a "max" operation. Both q and ℓ are typically small; each was 2 for us. These convolve and pool layers are repeated here three times, with changing dimensions (in our actual implementation, there are 13 layers in total). Finally the 500 derived features are flattened, and a fully connected layer maps them to the 100 classes via a "softmax" activation.

hidden layers. These consist of two special types of layers: "convolve" and "pool." We describe each in turn.

Convolve Layer

Figure 18.10 illustrates a convolution layer, and some details are given in

[2] Classification becomes increasingly difficult as the number of classes grows. With equal representation in each class, the NULL or random error rate for K classes is $(K-1)/K$; 50% for two classes, 99% for 100.

the caption. If an image x is represented by a $k \times k$ matrix, and a filter f

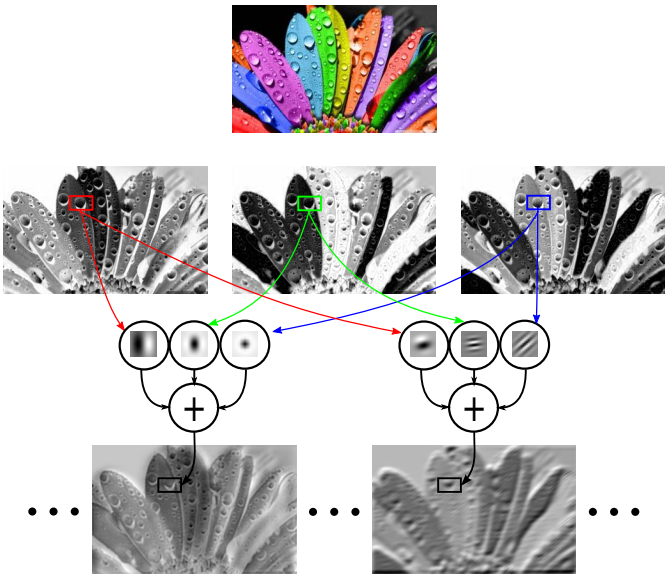

Figure 18.10 Convolution layer for the input images. The input image is split into its three color components. A single filter is a $q \times q \times p_1$ array (here one $q \times q$ for each of the $p_1 = 3$ color panes), and is used to compute an inner product with a correspondingly sized subimage in each pane, and summed across the p_1 panes. We used $q = 2$, and small values are typical. This is repeated over all (overlapping) $q \times q$ subimages (with boundary padding), and hence produces an image of the same dimension as one of the input panes. This is the convolution operation. There are p_2 different versions of this filter, and hence p_2 new panes are produced. Each of the p_2 filters has $p_1 q^2$ weights, which are learned via backpropagation.

is a $q \times q$ matrix with $q \ll k$, the convolved image is another $k \times k$ matrix $\tilde{x}$ with elements $\tilde{x}_{i,j} = \sum_{\ell=1}^{q} \sum_{\ell'=1}^{q} x_{i+\ell, j+\ell'} f_{\ell,\ell'}$ (with edge padding to achieve a full-sized $k \times k$ output image). In our application we used 2×2, but other sizes such as 3×3 are popular. It is most natural to represent the structure in terms of these images as in Figure 18.9, but they could all be vectorized into a massive network diagram as in Figures 18.1 and 18.3. However, the weights would have special sparse structure, with most being zero, and the nonzero values repeated ("weight sharing").

Pool Layer

The pool layer corresponds to a kind of nonlinear activation. It reduces each nonoverlapping block of $r \times r$ pixels ($r = 2$ for us) to a single number by computing their maximum. Why maximum? The convolution filters are themselves small image patches, and are looking to identify similar patches in the target image (in which case the inner product will be high). The max operation introduces an element of local translation invariance. The pool operation reduces the size of each image by a factor r in each dimension. To compensate, the number of tiles in the next convolution layer is typically increased accordingly. Also, as these tiles get smaller, the effective weights resulting from the convolution operator become denser. Eventually the tiles are the same size as the convolution filter, and the layer becomes fully connected.

18.5 Learning a Deep Network

Despite the additional structure imposed by the convolution layers, deep networks are learned by gradient descent. The gradients are computed by backpropagation as before, but with special care taken to accommodate the tied weights in the convolution filters. However, a number of additional tricks have been introduced that appear to improve the performance of modern deep learning networks. These are mostly aimed at regularization; indeed, our 100-class image network has around 50 million parameters, so regularization is essential to avoid overfitting. We briefly discuss some of these.

Dropout

This is a form of regularization that is performed when learning a network, typically at different rates at the different layers. It applies to all networks, not just convolutional; in fact, it appears to work better when applied at the deeper, denser layers. Consider computing the activation $z_\ell^{(k)}$ in layer k as in (18.3) for a single observation during the feed-forward stage. The idea is to randomly set each of the p_{k-1} nodes $a_j^{(k-1)}$ to zero with probability ϕ, and inflate the remaining ones by a factor $1/(1-\phi)$. Hence, for this observation, those nodes that survive have to *stand in* for those omitted. This can be shown to be a form of ridge regularization, and when done correctly †4 improves performance.[†] The fraction ϕ omitted is a tuning parameter, and for convolutional networks it appears to be better to use different values at

different layers. In particular, as the layers become denser, ϕ is increased: from 0 in the input layer to 0.5 in the final, fully connected layer.

Input Distortion

This is another form of regularization that is particularly suitable for tasks like image classification. The idea is to augment the training set with many distorted copies of an input image (but of course the same label). These distortions can be location shifts and other small affine transformations, but also color and shading shifts that might appear in natural images. We show

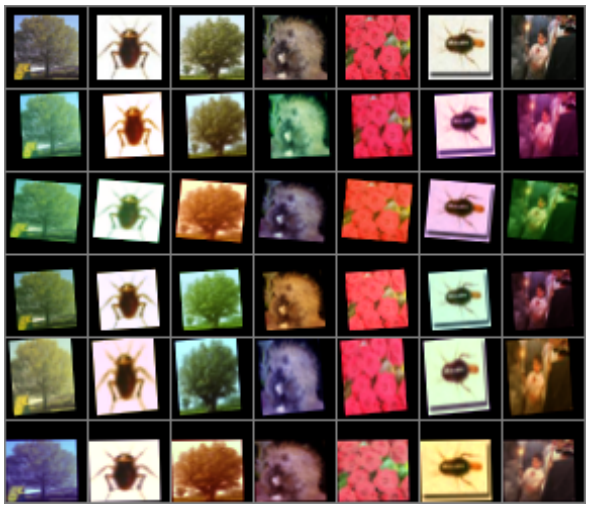

Figure 18.11 Each column represents distorted versions of an input image, including affine and color distortions. The input images are padded on the boundary to increase the size, and hence allow space for some of the distortions.

some distorted versions of input images in Figure 18.11. The distortions are such that a human would have no trouble identifying any of the distorted images if they could identify the original.[†] This both enriches the training data with *hints*, and also prevents overfitting to the original image. One could also apply distortions to a test image, and then "poll" the results to produce a final classification. †5

Configuration

Designing the correct architecture for a deep-learning network, along with the various choices at each layer, appears to require experience and trial

and error. We summarize the third and final architecture which we built †6 for classifying the **CIFAR-100** data set in Algorithm 18.2.† In addition to these size parameters for each layer, we must select the activation functions and additional regularization. In this case we used the leaky rectified linear functions $g_\alpha(z)$ (Section 18.2), with α increasing from 0.05 in layer 5 up to 0.5 in layer 13. In addition a type of ℓ_2 regularization was imposed on the weights, restricting all incoming weight vectors to a node to have ℓ_2 norm bounded by one. Figure 18.12 shows both the progress of the optimization objective (red) and the test misclassification error (blue) as the gradient-descent algorithm proceeds. The accelerated gradient method maintains a memory, which we can see was restarted twice to get out of local minima. Our network achieved a test error rate of 35% on the 10,000 test images (100 images per class). The best reported error rate we have seen is 25%, so apparently we have some way to go!

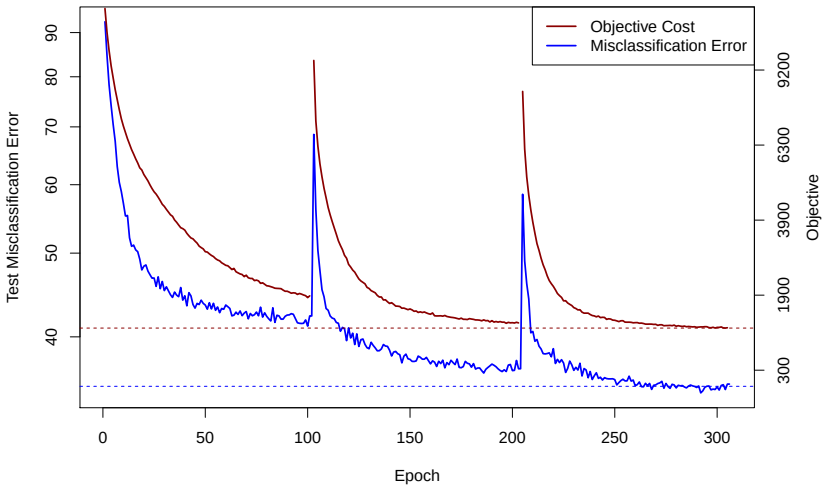

Figure 18.12 Progress of the algorithm as a function of the number of epochs. The accelerated gradient algorithm is "restarted" every 100 epochs, meaning the long-term memory is forgotten, and a new trail is begun, starting at the current solution. The red curve shows the objective (negative penalized log-likelihood on the training data). The blue curve shows test-set misclassification error. The vertical axis is on the log scale, so zero cannot be included.

Algorithm 18.2 CONFIGURATION PARAMETERS FOR DEEP-LEARNING NETWORK USED ON THE `CIFAR-100` DATA.

Layer 1: 100 convolution maps each with $2 \times 2 \times 3$ kernel (the 3 for three colors). The input image is padded from 32×32 to 40×40 to accommodate input distortions.

Layers 2 and 3: 100 convolution maps each $2 \times 2 \times 100$. Compositions of convolutions are roughly equivalent to convolutions with a bigger bandwidth, and the smaller ones have fewer parameters.

Layer 4: Max pool 2×2 layer, pooling nonoverlapping 2×2 blocks of pixels, and hence reducing the images to size 20×20.

Layer 5: 300 convolution maps each $2 \times 2 \times 100$, with dropout learning with rate $\phi_5 = 0.05$.

Layer 6: Repeat of Layer 5.

Layer 7: Max pool 2×2 layer (down to 10×10 images).

Layer 8: 600 convolution maps each $2 \times 2 \times 300$, with dropout rate $\phi_8 = 0.10$.

Layer 9: 800 convolution maps each $2 \times 2 \times 600$, with dropout rate $\phi_9 = 0.10$.

Layer 10: Max pool 2×2 layer (down to 5×5 images).

Layer 11: 1600 convolution maps, each $1 \times 1 \times 800$. This is a pixelwise weighted sum across the 800 images from the previous layer.

Layer 12: 2000 fully connected units, with dropout rate $\phi_{12} = 0.25$.

Layer 13: Final 100 output units, with softmax activation, and dropout rate $\phi_{13} = 0.5$.

18.6 Notes and Details

The reader will notice that probability models have disappeared from the development here. Neural nets are elaborate regression methods aimed solely at prediction—not estimation or explanation in the language of Section 8.4. In place of parametric optimality criteria, the machine learning community has focused on a set of specific prediction data sets, like the digits `MNIST` corpus and `CIFAR-100`, as benchmarks for measuring performance.

There is a vast literature on neural networks, with hundreds of books and thousands of papers. With the resurgence of deep learning, this literature is again growing. Two early statistical references on neural networks are Ripley (1996) and Bishop (1995), as well as Hastie *et al.* (2009) devote one chapter to the topic. Part of our description of backpropagation

in Section 18.2 was guided by Andrew Ng's online Stanford lecture notes (Ng, 2015). Bengio *et al.* (2013) provide a useful review of autoencoders. LeCun *et al.* (2015) give a brief overview of deep learning, written by three pioneers of this field: Yann LeCun, Yoshua Bengio and Geoffrey Hinton; we also benefited from reading Ngiam *et al.* (2010). Dropout learning (Srivastava *et al.*, 2014) is a relatively new idea, and its connections with ridge regression were most usefully described in Wager *et al.* (2013). The most popular version of accelerated gradient descent is due to Nesterov (2013). Learning with hints is due to Abu-Mostafa (1995). The material in Sections 18.4 and 18.5 benefited greatly from discussions with Rakesh Achanta (Achanta and Hastie, 2015), who produced some of the color images and diagrams, and designed and fit the deep-learning network to the `CIFAR-100` data.

†1 [p. 352] The Neural Information Processing Systems (NIPS) conferences started in late Fall 1987 in Denver, Colorado, and post-conference workshops were held at the nearby ski resort at Vail. These are still very popular today, although the venue has changed over the years. The NIPS proceedings are refereed, and NIPS papers count as publications in most fields, especially Computer Science and Engineering. Although neural networks were initially the main topic of the conferences, a modern NIPS conference covers all the latest ideas in machine learning.

†2 [p. 353] `MNIST` is a curated database of images of handwritten digits (LeCun and Cortes, 2010). There are 60,000 training images, and 10,000 test images, each a 28×28 grayscale image. These data have been used as a testbed for many different learning algorithms, so the reported best error rates might be optimistic.

†3 [p. 360] *Tuning parameters.* Typical neural network implementations have dozens of tuning parameters, and many of these are associated with the fine tuning of the descent algorithm. We used the `h2o.deepLearning` function in the `R` package `h2o` to fit our model for the `MNIST` data set. It has around 20 such parameters, although most default to factory-tuned constants that have been found to work well on many examples. Arno Candel was very helpful in assisting us with the software.

†4 [p. 368] *Dropout and ridge regression.* Dropout was originally proposed in Srivastava *et al.* (2014), and reinterpreted in Wager *et al.* (2013). Dropout was inspired by the random selection of variables at each tree split in a random forest (Section 17.1). Consider a simple version of dropout for the linear regression problem with squared-error loss. We have an $n \times p$ regression matrix $\mathbf{X}$, and a response n-vector $\mathbf{y}$. For simplicity we assume all variables have mean zero, so we can ignore intercepts. Consider the

following random least-squares criterion:

$$L_I(\beta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij} I_{ij} \beta_j\right)^2.$$

Here the I_{ij} are i.i.d variables $\forall i, j$ with

$$I_{ij} = \begin{cases} 0 & \text{with probability } \phi, \\ 1/(1-\phi) & \text{with probability } 1-\phi, \end{cases}$$

(this particular form is used so that $E[I_{ij}] = 1$). Using simple probability it can be shown that the expected score equations can be written

$$\mathrm{E}\left[\frac{\partial L_I(\beta)}{\partial \beta}\right] = -\boldsymbol{X}'\boldsymbol{y} + \boldsymbol{X}'\boldsymbol{X}\beta + \frac{\phi}{1-\phi}\boldsymbol{D}\beta = 0, \tag{18.23}$$

with $\boldsymbol{D} = \mathrm{diag}\{\|\boldsymbol{x}_1\|^2, \|\boldsymbol{x}_2\|^2, \ldots, \|\boldsymbol{x}_p\|^2\}$. Hence the solution is given by

$$\hat{\beta} = \left(\boldsymbol{X}'\boldsymbol{X} + \frac{\phi}{1-\phi}\boldsymbol{D}\right)^{-1}\boldsymbol{X}'\boldsymbol{y}, \tag{18.24}$$

a generalized ridge regression. If the variables are standardized, the term $\boldsymbol{D}$ becomes a scalar, and the solution is identical to ridge regression. With a nonlinear activation function, the interpretation changes slightly; see Wager *et al.* (2013) for details.

†5 [p. 369] *Distortion and ridge regression.* We again show in a simple example that input distortion is similar to ridge regression. Assume the same setup as in the previous example, except a different randomized version of the criterion:

$$L_N(\beta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p}(x_{ij} + n_{ij})\beta_j\right)^2.$$

Here we have added random noise to the prediction variables, and we assume this noise is i.i.d $(0, \lambda)$. Once again the expected score equations can be written

$$\mathrm{E}\left[\frac{\partial L_N(\beta)}{\partial \beta}\right] = -\boldsymbol{X}'\boldsymbol{y} + \boldsymbol{X}'\boldsymbol{X}\beta + \lambda\beta = 0, \tag{18.25}$$

because of the independence of all the n_{ij} and $\mathrm{E}(n_{ij}^2) = \lambda$. Once again this leads to a ridge regression. So replacing each observation pair x_i, y_i by the collection $\{x_i^{*b}, y_i\}_{b=1}^{B}$, where each x_i^{*b} is a noisy version of x_i, is approximately equivalent to a ridge regression on the original data.

†6 [p. 370] *Software for deep learning.* Our deep learning convolutional network for the `CIFAR-100` data was constructed and run by Rakesh Achanta in `Theano`, a Python-based system (Bastien *et al.*, 2012; Bergstra *et al.*, 2010). `Theano` has a user-friendly language for specifying the host of parameters for a deep-learning network, and uses symbolic differentiation for computing the gradients needed in stochastic gradient descent. In 2015 Google announced an open-source version of their `TensorFlow` software for fitting deep networks.