prevent overfitting by eliminating "irrelevant" features. To see this, imagine pre-computing all possible weak-learners, and defining a feature vector of the form $\boldsymbol{\phi}(\mathbf{x}) = [\phi_1(\mathbf{x}), \ldots, \phi_K(\mathbf{x})]$. We could use ℓ_1 regularization to select a subset of these. Alternatively we can use boosting, where at each step, the weak learner creates a new ϕ_k on the fly. It is possible to combine boosting and ℓ_1 regularization, to get an algorithm known as **L1-Adaboost** (Duchi and Singer 2009). Essentially this method greedily adds the best features (weak learners) using boosting, and then prunes off irrelevant ones using ℓ_1 regularization.

Another explanation has to do with the concept of margin, which we introduced in Section 14.5.2.2. (Schapire et al. 1998; Ratsch et al. 2001) proved that AdaBoost maximizes the margin on the training data. (Rosset et al. 2004) generalized this to other loss functions, such as log-loss.

16.4.9 A Bayesian view

So far, our presentation of boosting has been very frequentist, since it has focussed on greedily minimizing loss functions. A likelihood interpretation of the algorithm was given in (Neal and MacKay 1998; Meek et al. 2002). The idea is to consider a mixture of experts model of the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \sum_{m=1}^{M} \pi_m p(y|\mathbf{x}, \boldsymbol{\gamma}_m) \tag{16.59}$$

where each expert $p(y|\mathbf{x}, \boldsymbol{\gamma}_m)$ is like a weak learner. We usually fit all M experts at once using EM, but we can imagine a sequential scheme, whereby we only update the parameters for one expert at a time. In the E step, the posterior responsibilities will reflect how well the existing experts explain a given data point; if this is a poor fit, these data points will have more influence on the next expert that is fitted. (This view naturally suggest a way to use a boosting-like algorithm for unsupervised learning: we simply sequentially fit mixture models, instead of mixtures of experts.)

Notice that this is a rather "broken" MLE procedure, since it never goes back to update the parameters of an old expert. Similarly, if boosting ever wants to change the weight assigned to a weak learner, the only way to do this is to add the weak learner again with a new weight. This can result in unnecessarily large models. By contrast, the BART model (Chipman et al. 2006, 2010) uses a Bayesian version of backfitting to fit a small sum of weak learners (typically trees).

16.5 Feedforward neural networks (multilayer perceptrons)

A **feedforward neural network**, aka **multi-layer perceptron** (**MLP**), is a series of logistic regression models stacked on top of each other, with the final layer being either another logistic regression or a linear regression model, depending on whether we are solving a classification or regression problem. For example, if we have two layers, and we are solving a regression problem, the model has the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^T \mathbf{z}(\mathbf{x}), \sigma^2) \tag{16.60}$$

$$\mathbf{z}(\mathbf{x}) = g(\mathbf{V}\mathbf{x}) = [g(\mathbf{v}_1^T \mathbf{x}), \ldots, g(\mathbf{v}_H^T \mathbf{x})] \tag{16.61}$$

where g is a non-linear **activation** or **transfer function** (commonly the logistic function), $\mathbf{z}(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x}, \mathbf{V})$ is called the **hidden layer** (a deterministic function of the input), H is the

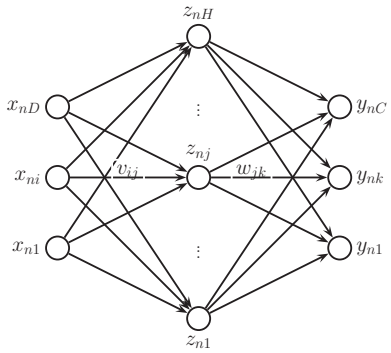

Figure 16.11 A neural network with one hidden layer.

number of **hidden units**, $\mathbf{V}$ is the weight matrix from the inputs to the hidden nodes, and $\mathbf{w}$ is the weight vector from the hidden nodes to the output. It is important that g be non-linear, otherwise the whole model collapses into a large linear regression model of the form $y = \mathbf{w}^T(\mathbf{V}\mathbf{x})$. One can show that an MLP is a **universal approximator**, meaning it can model any suitably smooth function, given enough hidden units, to any desired level of accuracy (Hornik 1991).

To handle binary classification, we pass the output through a sigmoid, as in a GLM:

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Ber}(y|\text{sigm}(\mathbf{w}^T\mathbf{z}(\mathbf{x}))) \tag{16.62}$$

We can easily extend the MLP to predict multiple outputs. For example, in the regression case, we have

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}|\mathbf{W}\ \boldsymbol{\phi}(\mathbf{x}, \mathbf{V}), \sigma^2\mathbf{I}) \tag{16.63}$$

See Figure 16.11 for an illustration. If we add **mutual inhibition** arcs between the output units, ensuring that only one of them turns on, we can enforce a sum-to-one constraint, which can be used for multi-class classification. The resulting model has the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) \quad = \quad \text{Cat}(y|\mathcal{S}(\mathbf{W}\mathbf{z}(\mathbf{x})) \tag{16.64}$$

16.5.1 Convolutional neural networks

The purpose of the hidden units is to learn non-linear combinations of the original inputs; this is called **feature extraction** or **feature construction**. These hidden features are then passed as input to the final GLM. This approach is particularly useful for problems where the original input features are not very individually informative. For example, each pixel in an image is not very informative; it is the combination of pixels that tells us what objects are present. Conversely, for a task such as document classification using a bag of words representation, each feature (word count) *is* informative on its own, so extracting "higher order" features is less important. Not suprisingly, then, much of the work in neural networks has been motivated by visual pattern

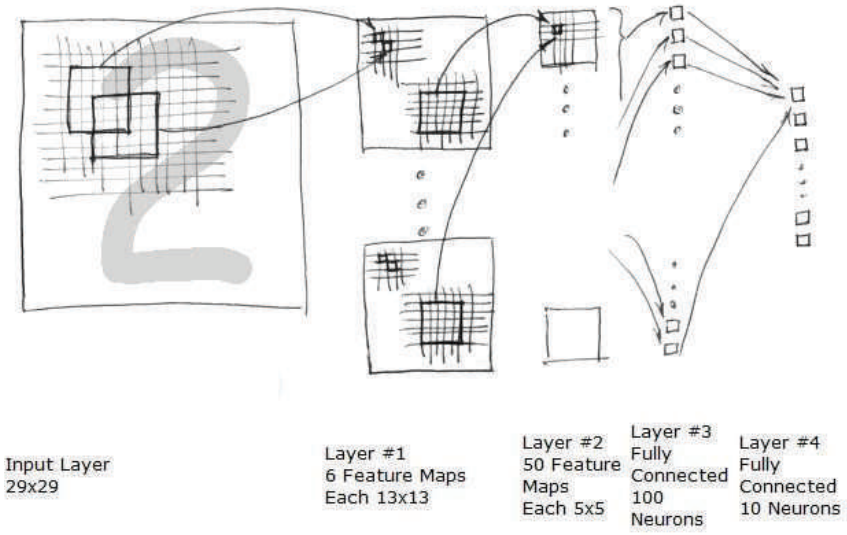

Figure 16.12 The convolutional neural network from (Simard et al. 2003). Source: `http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx` . Used with kind permission of Mike O'Neill.

recognition (e.g., (LeCun et al. 1989)), although they have also been applied to other types of data, including text (e.g., (Collobert and Weston 2008)).

A form of MLP which is particularly well suited to 1d signals like speech or text, or 2d signals like images, is the **convolutional neural network**. This is an MLP in which the hidden units have local **receptive fields** (as in the primary visual cortex), and in which the weights are **tied** or shared across the image, in order to reduce the number of parameters. Intuitively, the effect of such spatial parameter tying is that any useful features that are "discovered" in some portion of the image can be re-used everywhere else without having to be independently learned. The resulting network then exhibits **translation invariance**, meaning it can classify patterns no matter where they occur inside the input image.

Figure 16.12 gives an example of a convolutional network, designed by Simard and colleagues (Simard et al. 2003), with 5 layers (4 layers of adjustable parameters) designed to classify 29×29 gray-scale images of handwritten digits from the MNIST dataset (see Section 1.2.1.3). In layer 1, we have 6 **feature maps** each of which has size 13×13. Each hidden node in one of these feature maps is computed by convolving the image with a 5×5 weight matrix (sometimes called a kernel), adding a bias, and then passing the result through some form of nonlinearity. There are therefore $13 \times 13 \times 6 = 1014$ neurons in Layer 1, and $(5 \times 5 + 1) \times 6 = 156$ weights. (The "+1" is for the bias.) If we did not share these parameters, there would be $1014 \times 26 = 26,364$ weights at the first layer. In layer 2, we have 50 feature maps, each of which is obtained by convolving each feature map in layer 1 with a 5×5 weight matrix, adding them up, adding a bias, and passing through a nonlinearity. There are therefore $5 \times 5 \times 50 = 1250$ neurons in Layer 2, $(5 \times 5 + 1) \times 6 \times 50 = 7800$ adjustable weights (one kernel for each pair of feature

maps in layers 1 and 2), and $1250 \times 26 = 32,500$ connections. Layer 3 is fully connected to layer 2, and has 100 neurons and $100 \times (1250 + 1) = 125,100$ weights. Finally, layer 4 is also fully connected, and has 10 neurons, and $10 \times (100 + 1) = 1010$ weights. Adding the above numbers, there are a total of 3,215 neurons, 134,066 adjustable weights, and 184,974 connections.

This model is usually trained using stochastic gradient descent (see Section 16.5.4 for details). A single pass over the data set is called an epoch. When Mike O'Neill did these experiments in 2006, he found that a single epoch took about 40 minutes (recall that there are 60,000 training examples in MNIST). Since it took about 30 epochs for the error rate to converge, the total training time was about 20 hours.[5] Using this technique, he obtained a misclassification rate on the 10,000 test cases of about 1.40%.

To further reduce the error rate, a standard trick is to expand the training set by including **distorted** versions of the original data, to encourage the network to be invariant to small changes that don't affect the identity of the digit. These can be created by applying a random flow field to shift pixels around. See Figure 16.13 for some examples. (If we use online training, such as stochastic gradient descent, we can create these distortions on the fly, rather than having to store them.) Using this technique, Mike O'Neill obtained a misclassification rate on the 10,000 test cases of about 0.74%, which is close to the current state of the art.[6]

Yann Le Cun and colleagues (LeCun et al. 1998) obtained similar performance using a slightly more complicated architecture shown in Figure 16.14. This model is known as **LeNet5**, and historically it came before the model in Figure 16.12. There are two main differences. First, LeNet5 has a **subsampling** layer between each convolutional layer, which either averages or computes the max over each small window in the previous layer, in order to reduce the size, and to obtain a small amount of shift invariance. The convolution and sub-sampling combination was inspired by Hubel and Wiesel's model of simple and complex cells in the visual cortex (Hubel and Wiesel 1962), and it continues to be popular in neurally-inspired models of visual object recognition (Riesenhuber and Poggio 1999). A similar idea first appeared in Fukushima's **neocognitron** (Fukushima 1975), though no globally supervised training algorithm was available at that time.

The second difference between LeNet5 and the Simard architecture is that the final layer is actually an RBF network rather than a more standard sigmoidal or softmax layer. This model gets a test error rate of about 0.95% when trained with no distortions, and 0.8% when trained with distortions. Figure 16.15 shows all 82 errors made by the system. Some are genuinely ambiguous, but several are errors that a person would never make. A web-based demo of the LeNet5 can be found at `http://yann.lecun.com/exdb/lenet/index.html`.

Of course, classifying isolated digits is of limited applicability: in the real world, people usually write strings of digits or other letters. This requires both segmentation and classification. Le Cun and colleagues devised a way to combine convolutional neural networks with a model similar to a conditional random field (described in Section 19.6) to solve this problem. The system was eventually deployed by the US postal service. (See (LeCun et al. 1998) for a more detailed account of the system, which remains one of the best performing systems for this task.)

5. Implementation details: Mike used C++ code and a variety of speedup tricks. He was using standard 2006 era hardware (an Intel Pentium 4 hyperthreaded processor running at 2.8GHz). See `http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx` for details.

6. A list of various methods, along with their misclassification rates on the MNIST test set, is available from `http://yann.lecun.com/exdb/mnist/`. Error rates within 0.1–0.2% of each other are not statistically significantly different.

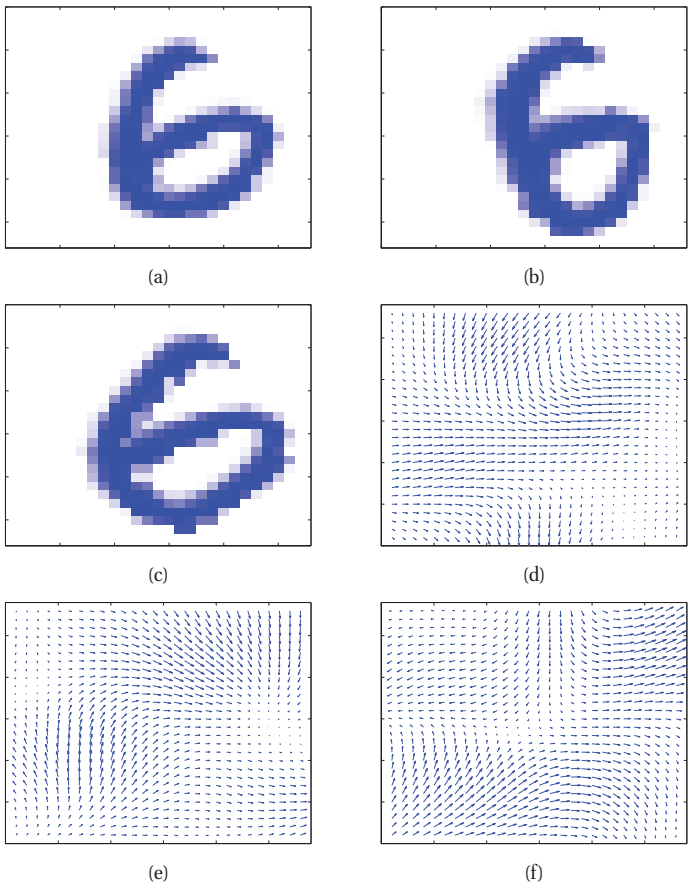

Figure 16.13 Several synthetic warpings of a handwritten digit. Based on Figure 5.14 of (Bishop 2006a). Figure generated by `elasticDistortionsDemo`, written by Kevin Swersky.

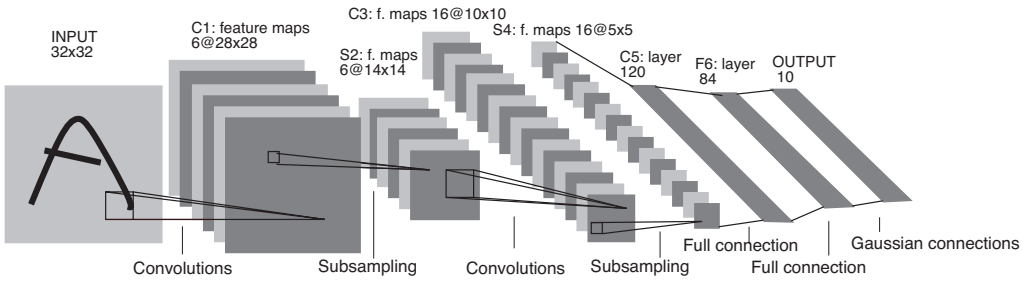

Figure 16.14 LeNet5, a convolutional neural net for classifying handwritten digits. Source: Figure 2 from (LeCun et al. 1998) . Used with kind permission of Yann LeCun.

Figure 16.15 These are the 82 errors made by LeNet5 on the 10,000 test cases of MNIST. Below each image is a label of the form correct-label $\rightarrow$ estimated-label. Source: Figure 8 of (LeCun et al. 1998). Used with kind permission of Yann LeCun. (Compare to Figure 28.4(b) which shows the results of a deep generative model.)

16.5.2 Other kinds of neural networks

Other network topologies are possible besides the ones discussed above. For example, we can have **skip arcs** that go directly from the input to the output, skipping the hidden layer; we can have sparse connections between the layers; etc. However, the MLP always requires that the weights form a directed acyclic graph. If we allow feedback connections, the model is known as a **recurrent neural network**; this defines a nonlinear dynamical system, but does not have a simple probabilistic interpretation. Such RNN models are currently the best approach for language modeling (i.e., performing word prediction in natural language) (Tomas et al. 2011), significantly outperforming the standard n-gram-based methods discussed in Section 17.2.2.

If we allow symmetric connections between the hidden units, the model is known as a **Hopfield network** or **associative memory**; its probabilistic counterpart is known as a Boltzmann machine (see Section 27.7) and can be used for unsupervised learning.

16.5.3 A brief history of the field

Neural networks have been the subject of great interest for many decades, due to the desire to understand the brain, and to build learning machines. It is not possible to review the entire history here. Instead, we just give a few "edited highlights".

The field is generally viewed as starting with McCulloch and Pitts (McCullich and Pitts 1943), who devised a simple mathematical model of the neuron in 1943, in which they approximated the

output as a weighted sum of inputs passed through a threshold function, $y = \mathbb{I}(\sum_i w_i x_i > \theta)$, for some threshold θ. This is similar to a sigmoidal activation function. Frank Rosenblatt invented the perceptron learning algorithm in 1957, which is a way to estimate the parameters of a McCulloch-Pitts neuron (see Section 8.5.4 for details). A very similar model called the **adaline** (for adaptive linear element) was invented in 1960 by Widrow and Hoff.

In 1969, Minsky and Papert (Minsky and Papert 1969) published a famous book called "Perceptrons" in which they showed that such linear models, with no hidden layers, were very limited in their power, since they cannot classify data that is not linearly separable. This considerably reduced interest in the field.

In 1986, Rumelhart, Hinton and Williams (Rumelhart et al. 1986) discovered the backpropagation algorithm (see Section 16.5.4), which allows one to fit models with hidden layers. (The backpropagation algorithm was originally discovered in (Bryson and Ho 1969), and independently in (Werbos 1974); however, it was (Rumelhart et al. 1986) that brought the algorithm to people's attention.) This spawned a decade of intense interest in these models.

In 1987, Sejnowski and Rosenberg (Sejnowski and Rosenberg 1987) created the famous **NETtalk** system, that learned a mapping from English words to phonetic symbols which could be fed into a speech synthesizer. An audio demo of the system as it learns over time can be found at `http://www.cnl.salk.edu/ParallelNetsPronounce/nettalk.mp3`. The systems starts by "babbling" and then gradually learns to pronounce English words. NETtalk learned a **distributed representation** (via its hidden layer) of various sounds, and its success spawned a big debate in psychology between **connectionism**, based on neural networks, and **computationalism**, based on syntactic rules. This debate lives on to some extent in the machine learning community, where there are still arguments about whether learning is best performed using low-level, "neural-like" representations, or using more structured models.

In 1989, Yann Le Cun and others (LeCun et al. 1989) created the famous LeNet system described in Section 16.5.1.

In 1992, the support vector machine (see Section 14.5) was invented (Boser et al. 1992). SVMs provide similar prediction accuracy to neural networks while being considerably easier to train (since they use a convex objective function). This spawned a decade of interest in kernel methods in general.[7] Note, however, that SVMs do not use adaptive basis functions, so they require a fair amount of human expertise to design the right kernel function.

In 2002, Geoff Hinton invented the contrastive divergence training procedure (Hinton 2002), which provided a way, for the first time, to learn deep networks, by training one layer at a time in an unsupervised fashion (see Section 27.7.2.4 for details). This in turn has spawned renewed interest in neural networks over the last few years (see Chapter 28).

16.5.4 The backpropagation algorithm

Unlike a GLM, the NLL of an MLP is a non-convex function of its parameters. Nevertheless, we can find a locally optimal ML or MAP estimate using standard gradient-based optimization methods. Since MLPs have lots of parameters, they are often trained on very large data sets.

7. It became part of the folklore during the 1990s that to get published in the top machine learning conference known as NIPS, which stands for "neural information processing systems", it was important to ensure your paper did not contain the word "neural network"!

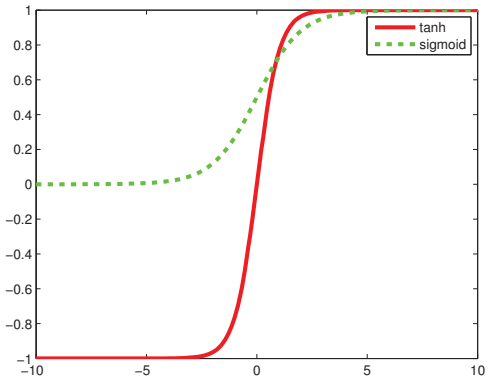

Figure 16.16 Two possible activation functions. $\tanh$ maps $\mathbb{R}$ to $[-1, +1]$ and is the preferred nonlinearity for the hidden nodes. sigm maps $\mathbb{R}$ to $[0, 1]$ and is the preferred nonlinearity for binary nodes at the output layer. Figure generated by `tanhPlot`.

Consequently it is common to use first-order online methods, such as stochastic gradient descent (Section 8.5.2), whereas GLMs are usually fit with IRLS, which is a second-order offline method.

We now discuss how to compute the gradient vector of the NLL by applying the chain rule of calculus. The resulting algorithm is known as **backpropagation**, for reasons that will become apparent.

For notational simplicity, we shall assume a model with just one hidden layer. It is helpful to distinguish the pre- and post-synaptic values of a neuron, that is, before and after we apply the nonlinearity. Let $\mathbf{x}_n$ be the n'th input, $\mathbf{a}_n = \mathbf{V}\mathbf{x}_n$ be the pre-synaptic hidden layer, and $\mathbf{z}_n = g(\mathbf{a}_n)$ be the post-synaptic hidden layer, where g is some **transfer function**. We typically use $g(a) = \text{sigm}(a)$, but we may also use $g(a) = \tanh(a)$: see Figure 16.16 for a comparison. (When the input to sigm or $\tanh$ is a vector, we assume it is applied component-wise.)

We now convert this hidden layer to the output layer as follows. Let $\mathbf{b}_n = \mathbf{W}\mathbf{z}_n$ be the pre-synaptic output layer, and $\hat{\mathbf{y}}_n = h(\mathbf{b}_n)$ be the post-synaptic output layer, where h is another nonlinearity, corresponding to the canonical link for the GLM. (We reserve the notation $\mathbf{y}_n$, without the hat, for the output corresponding to the n'th training case.) For a regression model, we use $h(\mathbf{b}) = \mathbf{b}$; for binary classifcation, we use $h(\mathbf{b}) = [\text{sigm}(b_1), \ldots, \text{sigm}(b_c)]$; for multi-class classification, we use $h(\mathbf{b}) = \mathcal{S}(\mathbf{b})$.

We can write the overall model as follows:

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n \tag{16.65}$$

The parameters of the model are $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$, the first and second layer weight matrices. Offset or bias terms can be accomodated by clamping an element of $\mathbf{x}_n$ and $\mathbf{z}_n$ to 1.[8]

8. In the regression setting, we can easily estimate the variance of the output noise using the empirical variance of the residual errors, $\hat{\sigma}^2 = \frac{1}{N}||\hat{\mathbf{y}}(\hat{\boldsymbol{\theta}}) - \mathbf{y}||^2$, after training is complete. There will be one value of σ^2 for each output node, if we are performing multi-target regression, as we usually assume.

In the regression case, with K outputs, the NLL is given by the squared error:

$$J(\boldsymbol{\theta}) = -\sum_n \sum_k (\hat{y}_{nk}(\boldsymbol{\theta}) - y_{nk})^2 \tag{16.66}$$

In the classification case, with K classes, the NLL is given by the cross entropy

$$J(\boldsymbol{\theta}) = -\sum_n \sum_k y_{nk} \log \hat{y}_{nk}(\boldsymbol{\theta}) \tag{16.67}$$

Our task is to compute $\nabla_{\boldsymbol{\theta}} J$. We will derive this for each n separately; the overall gradient is obtained by summing over n, although often we just use a mini-batch (see Section 8.5.2).

Let us start by considering the output layer weights. We have

$$\nabla_{\mathbf{w}_k} J_n = \frac{\partial J_n}{\partial b_{nk}} \nabla_{\mathbf{w}_k} b_{nk} = \frac{\partial J_n}{\partial b_{nk}} \mathbf{z}_n \tag{16.68}$$

since $b_{nk} = \mathbf{w}_k^T \mathbf{z}_n$. Assuming h is the canonical link function for the output GLM, then Equation 9.91 tells us that

$$\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^w = (\hat{y}_{nk} - y_{nk}) \tag{16.69}$$

which is the error signal. So the overall gradient is

$$\nabla_{\mathbf{w}_k} J_n = \delta_{nk}^w \mathbf{z}_n \tag{16.70}$$

which is the pre-synaptic input to the output layer, namely $\mathbf{z}_n$, times the error signal, namely δ_{nk}^w.

For the input layer weights, we have

$$\nabla_{\mathbf{v}_j} J_n = \frac{\partial J_n}{\partial a_{nj}} \nabla_{\mathbf{v}_j} a_{nj} \triangleq \delta_{nj}^v \mathbf{x}_n \tag{16.71}$$

where we exploited the fact that $a_{nj} = \mathbf{v}_j^T \mathbf{x}_n$. All that remains is to compute the first level error signal δ_{nj}^v. We have

$$\delta_{nj}^v = \frac{\partial J_n}{\partial a_{nj}} = \sum_{k=1}^K \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}} = \sum_{k=1}^K \delta_{nk}^w \frac{\partial b_{nk}}{\partial a_{nj}} \tag{16.72}$$

Now

$$b_{nk} = \sum_j w_{kj} g(a_{nj}) \tag{16.73}$$

so

$$\frac{\partial b_{nk}}{\partial a_{nj}} = w_{kj} g'(a_{nj}) \tag{16.74}$$

where $g'(a) = \frac{d}{da} g(a)$. For tanh units, $g'(a) = \frac{d}{da} \tanh(a) = 1 - \tanh^2(a) = \text{sech}^2(a)$, and for sigmoid units, $g'(a) = \frac{d}{da} \sigma(a) = \sigma(a)(1 - \sigma(a))$. Hence

$$\delta_{nj}^v = \sum_{k=1}^K \delta_{nk}^w w_{kj} g'(a_{nj}) \tag{16.75}$$

Thus the layer 1 errors can be computed by passing the layer 2 errors back through the $\mathbf{W}$ matrix; hence the term "backpropagation". The key property is that we can compute the gradients locally: each node only needs to know about its immediate neighbors. This is supposed to make the algorithm "neurally plausible", although this interpretation is somewhat controversial.

Putting it all together, we can compute all the gradients as follows: we first perform a forwards pass to compute $\mathbf{a}_n$, $\mathbf{z}_n$, $\mathbf{b}_n$ and $\hat{\mathbf{y}}_n$. We then compute the error for the output layer, $\boldsymbol{\delta}_n^{(2)} = \hat{\mathbf{y}}_n - \mathbf{y}_n$, which we pass backwards through $\mathbf{W}$ using Equation 16.75 to compute the error for the hidden layer, $\boldsymbol{\delta}_n^{(1)}$. We then compute the overall gradient as follows:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_n [\boldsymbol{\delta}_n^v \mathbf{x}_n,\ \boldsymbol{\delta}_n^w \mathbf{z}_n] \tag{16.76}$$

16.5.5 Identifiability

It is easy to see that the parameters of a neural network are not identifiable. For example, we can change the sign of the weights going into one of the hidden units, so long as we change the sign of all the weights going out of it; these effects cancel, since tanh is an odd function, so $\tanh(-a) = -\tanh(a)$. There will be H such sign flip symmetries, leading to 2^H equivalent settings of the parameters. Similarly, we can change the identity of the hidden units without affecting the likelihood. There are $H!$ such permutations. The total number of equivalent parameter settings (with the same likelihood) is therefore $H!2^H$.

In addition, there may be local minima due to the non-convexity of the NLL. This can be a more serious problem, although with enough data, these local optima are often quite "shallow", and simple stochastic optimization methods can avoid them. In addition, it is common to perform multiple restarts, and to pick the best solution, or to average over the resulting predictions. (It does not make sense to average the parameters themselves, since they are not identifiable.)

16.5.6 Regularization

As usual, the MLE can overfit, especially if the number of nodes is large. A simple way to prevent this is called **early stopping**, which means stopping the training procedure when the error on the validation set first starts to increase. This method works because we usually initialize from small random weights, so the model is initially simple (since the $\tanh$ and sigm functions are nearly linear near the origin). As training progresses, the weights become larger, and the model becomes nonlinear. Eventually it will overfit.

Another way to prevent overfitting, that is more in keeping with the approaches used elsewhere in this book, is to impose a prior on the parameters, and then use MAP estimation. It is standard to use a $\mathcal{N}(0, \alpha^{-1}\mathbf{I})$ prior (equivalent to ℓ_2 regularization), where α is the precision (strength) of the prior. In the neural networks literature, this is called **weight decay**, since it encourages small weights, and hence simpler models. The penalized NLL objective becomes

$$J(\boldsymbol{\theta}) = -\sum_{n=1}^{N} \log p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) + \frac{\alpha}{2}[\sum_{ij} v_{ij}^2 + \sum_{jk} w_{jk}^2] \tag{16.77}$$