Chapter 6

Feedforward Deep Networks

Feedforward deep networks, also known as *multilayer perceptrons (MLPs)*, are the quintessential deep networks. They are parametric functions defined by composing together many parametric functions. Each of these component functions has multiple inputs and multiple outputs. In neural network terminology, we refer to each sub-function as a *layer* of the network, and each scalar output of one of these functions as a *unit* or sometimes as a *feature*. Even though each unit implements a relatively simple mapping or transformation of its input, the function represented by the entire network can become arbitrarily complex.

Not every deep learning algorithm can be understood in terms of defining a single, deterministic function like feedforward deep networks, but all of them share the property of containing many layers of many units. We can think of the number of units in each layer as being the *width* of a machine learning model, and the number of layers as its *depth*. Feedforward deep networks provide a conceptually simple example of an algorithm that captures the many advantages that come from having significant width and depth. Feedforward deep networks are also the key technology underlying most of the contemporary commercial applications of deep learning to large datasets.

In Chapter 5, we encountered several different traditional machine learning algorithms, including linear regression, linear classifiers, logistic regression and kernel machines. All of these algorithms work by applying a linear transformation to a fixed set of features. These algorithms can learn non-linear functions, but the non-linear part is fixed. In other words, the functions are non-linear in the space of inputs $\boldsymbol{x}$, but they are linear in some other pre-defined space.

Neural networks allow us to learn new kinds of non-linearity. Another way to view this idea is that neural networks allow us to learn the features provided to a linear model. From this point of view, neural networks allow us to automate the design of features—a task that until recently was performed gradually and collectively, by the combined efforts of an entire community of researchers.

6.1 MLPs from the 1980's

Feedforward supervised neural networks were among the first and most successful non-linear learning algorithms (Rumelhart *et al.*, 1986e,c). These networks learn at least one function defining the features, as well as a (typically linear) function mapping from features to output. The layers of the network that correspond to features rather than outputs are called *hidden layers*. This is because the correct values of the features are unknown. The features must be created by the training algorithm. The input and output of the network is by contrast *observed* or *visible* in the training data. Fig. 6.1 shows a classical MLP architecture from the 1980's, with a single hidden layer. A deeper version is obtained by simply having more hidden layers. Modern versions include changes in the non-linearities used and training procedure.

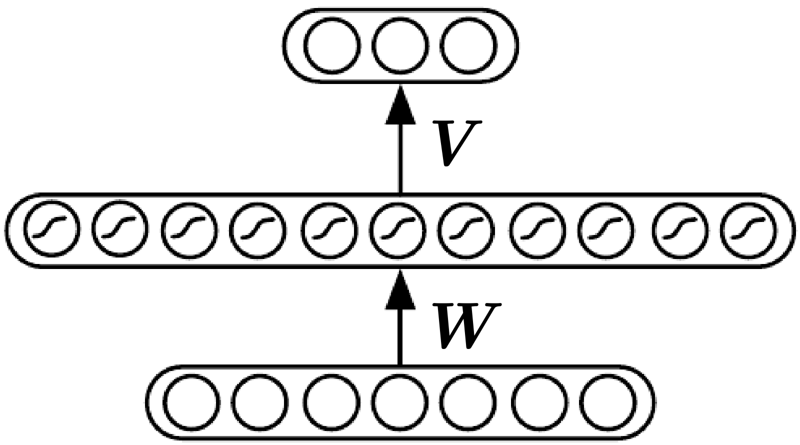

Figure 6.1: Shallow MLP, with one sigmoid hidden layer, computing vector-valued hidden unit vector $\boldsymbol{h} = \text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x})$ with weight matrix $\boldsymbol{W}$ and offset vector $\boldsymbol{c}$. The output vector is obtained via another learned affine transformation $\hat{\boldsymbol{y}} = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}$, with weight matrix $\boldsymbol{V}$ and output offset vector $\boldsymbol{b}$. The vector of hidden unit values $\boldsymbol{h}$ provides a new set of features, i.e., a new representation, derived from the raw input $\boldsymbol{x}$.

Example 6.1.1 introduces the equations for a shallow MLP for regression similar to those introduced in the 1980's, illustrated in Fig. 6.1, which we will generalize below in the next few sections.

Example 6.1.1. Shallow Multi-Layer Neural Network for Regression

Based on the above definitions, we could pick the family of input-output functions to be

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{b} + \boldsymbol{V}\text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x}),$$

illustrated in Fig. 6.1, where $\text{sigmoid}(a) = 1/(1 + e^{-a})$ is applied element-wise, the input is the vector $\boldsymbol{x} \in \mathbb{R}^{n_i}$, the hidden layer outputs are the elements of the vector $\boldsymbol{h} = \text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x})$ with n_h entries, the parameters are $\boldsymbol{\theta} = (\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{V}, \boldsymbol{W})$ (with $\boldsymbol{\theta}$ also viewed as the flattened vectorized version of the tuple) with $\boldsymbol{b} \in \mathbb{R}^{n_o}$ a vector the same dimension as the output (n_o), $\boldsymbol{c} \in \mathbb{R}^{n_h}$ of the same dimension as $\boldsymbol{h}$ (number of hidden units), $\boldsymbol{V} \in \mathbb{R}^{n_o \times n_h}$ and $\boldsymbol{W} \in \mathbb{R}^{n_h \times n_i}$ being weight matrices.
The loss function for this classical example could be the squared error $L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2$ (see Sec. 6.2 discussing how it makes $\hat{\boldsymbol{y}}$ an estimator of $\mathbb{E}[\boldsymbol{Y} \mid \boldsymbol{x}]$). The regularizer could be the ordinary L^2 weight decay $||\boldsymbol{\omega}||^2 = (\sum_{ij} W_{ij}^2 + \sum_{ki} V_{ki}^2)$, where we define the set of weights $\boldsymbol{\omega}$ as the concatenation of the elements of matrices $\boldsymbol{W}$ and $\boldsymbol{V}$. The L^2 weight decay thus penalizes the squared norm of the weights, with λ a scalar that is larger to penalize stronger weights, thus yielding smaller weights. During training, we minimize a cost function obtained by adding together the squared loss and the regularization term:

$$J(\boldsymbol{\theta}) = \lambda||\boldsymbol{\omega}||^2 + \frac{1}{n}\sum_{t=1}^{n} ||\boldsymbol{y}^{(t)} - (\boldsymbol{b} + \boldsymbol{V}\text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x}^{(t)}))||^2 .$$

where $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)})$ is the t-th training example, an (input,target) pair. Finally, the classical training procedure in this example is stochastic gradient descent, which iteratively updates $\boldsymbol{\theta}$ according to

$$\begin{aligned}
\boldsymbol{\omega} &\leftarrow \boldsymbol{\omega} - \epsilon \left(2\lambda\boldsymbol{\omega} + \nabla_{\boldsymbol{\omega}} L(f(\boldsymbol{x}^{(t)};\boldsymbol{\theta}), \boldsymbol{y}^{(t)}) \right) \\
\boldsymbol{\beta} &\leftarrow \boldsymbol{\beta} - \epsilon\nabla_{\boldsymbol{\beta}} L(f(\boldsymbol{x}^{(t)};\boldsymbol{\theta}), \boldsymbol{y}^{(t)}),
\end{aligned}$$

where $\boldsymbol{\beta} = (\boldsymbol{b}, \boldsymbol{c})$ contains the offset[1] parameters, $\boldsymbol{\omega} = (\boldsymbol{W}, \boldsymbol{V})$ the weight matrices, ϵ is a learning rate and t is incremented after each training example, modulo n. Sec. 6.4.3 shows how gradients can be computed efficiently thanks to back-propagation.

MLPs can learn powerful non-linear transformations: in fact, with enough hidden units they can represent arbitrarily complex but smooth functions, they can be universal approximators, as described below in Sec. 6.6. This is achieved by composing simple but non-linear learned transformations. By transforming the data non-linearly into a new space, a classification problem that was not linearly separable (not solvable by a linear classifier) can become separable, as illustrated in Fig. 6.2 and Fig. 6.3.

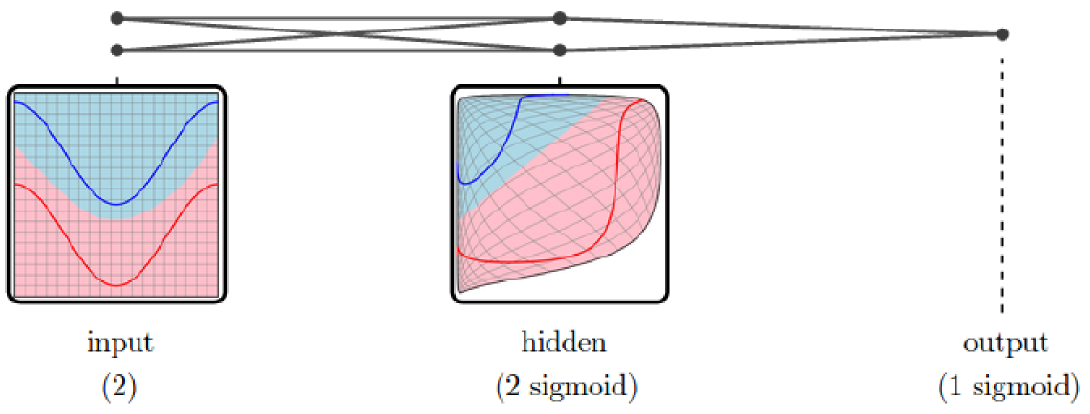

Figure 6.2: Each layer of a trained neural network non-linearly transforms its input, distorting the space so that the task becomes easier to perform, e.g., linear classification in the new feature space, in the above figures. The figure shows how a simple neural network with 2 inputs, 2 hidden units and one output can transform the 2-D input space so that the examples from the two classes become linearly separable. The red and blue solid curves are where the training examples come from (with color indicating class). The paler red (resp. blue) region indicates the region that should be labeled as red (resp. blue), with the blue-to-red interface corresponding to a good decision surface. On the left, we see in a black square the good decision surface in the original input space, and see that it is non-linear (i.e., a linear classifier could not do a good job if applied directly on the raw inputs). The raw input space is also mapped by a regular grid in the left square, and the grid gets transformed non-linearly, i.e., warped differently in different parts of the space (consider how the grid was transformed), to obtain the middle square, i.e., in the space of hidden units: every (x_1, x_2) point in the left block is mapped to a point (h_1, h_2) in the middle block using the parameters of the hidden units. We see that when the data is properly transformed non-linearly by the first hidden layer, the good decision surface becomes a linear one, which can be implemented by the output layer (which is a linear classifier when viewed as a function of the last hidden layer). Reproduced with permission by Chris Olah from `http://colah.github.io/`, where many more insightful visualizations can be found.

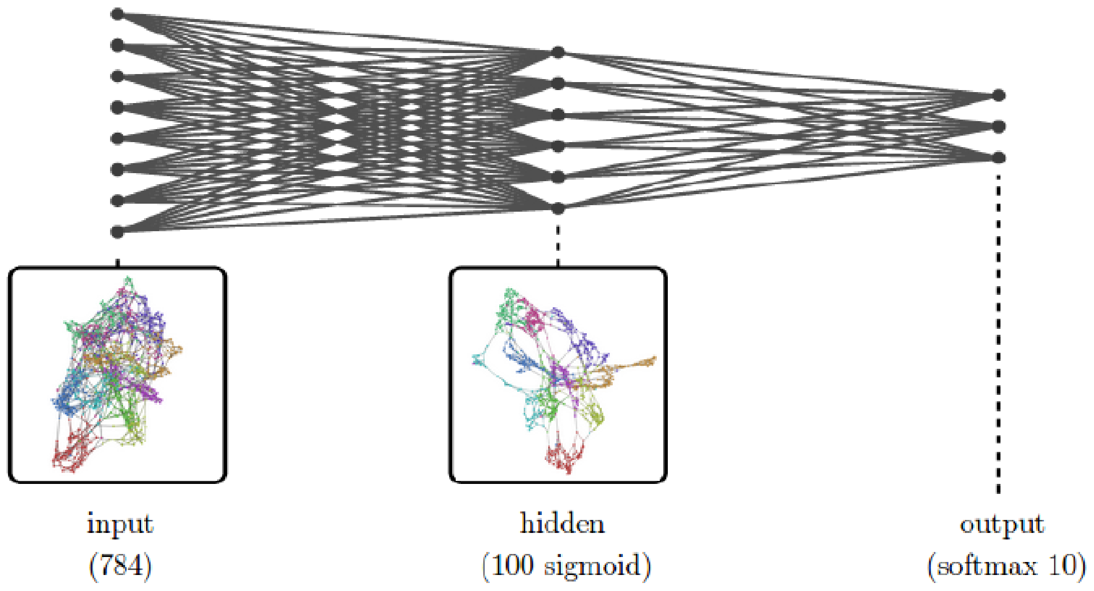

Figure 6.3: Like Fig. 6.2, this figure shows how the hidden layer of an MLP can non-linearly warp the input space so as to make the classification task easier for a linear classifier (the output layer of a shallow MLP). The MLP now has 784 inputs (pixels of a 28×28 image of a digit), 100 hidden units and 10 outputs corresponding to the 10 digit classes. We cannot directly visualize with a 2-D grid the input and hidden layer spaces, but we can visualize a 2-D approximation obtained by dimensionality reduction. The squares below the input and hidden layer show where the training examples are in this reduced space, with one point per example, colored according to the digit class. Again, we see that the digits of different classes can be more easily separated in the feature space of the hidden layer than in the raw pixel space, with digits of the same class tending to form better separated clusters. Reproduced with permission by Chris Olah from `http://colah.github.io/`, where more detail about this experiment can be found.

6.2 Estimating Conditional Statistics

To gently move from linear predictors to non-linear ones, let us consider the squared error loss function studied in the previous chapter, where the learning task is the estimation of the expected value of $\mathbf{y}$ given $\mathbf{x}$. In the context of linear regression, the conditional expectation of $\boldsymbol{y}$is used as the mean of a Gaussian distribution that we fit with maximum likelihood. We can generalize linear regression to regression via any function f by defining the mean squared error of f:

$$\mathbb{E}[||\boldsymbol{y} - f(\boldsymbol{x})||^2]$$

where the expectation is over the training set during training, and over the data generating distribution to obtain generalization error.

We can generalize its interpretation beyond the case where f is linear or affine, uncovering an interesting property: minimizing it yields an estimator of the conditional expectation of the output variable $\mathbf{y}$ given the input variable $\mathbf{x}$, i.e.,

$$\arg\min_{f \in \mathbb{H}} \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}[||\boldsymbol{y} - f(\boldsymbol{x})||^2] = \mathbb{E}_{p(\mathbf{x},y)}[\mathbf{y} \mid \boldsymbol{x}]. \tag{6.1}$$

provided that our set of function $\mathbb{H}$ contains $\mathbb{E}_{p(\mathbf{x},\mathbf{y})}[\mathbf{y} \mid \boldsymbol{x}]$. (If you would like to work out the proof yourself, it is easy to do using calculus of variations, which we describe in Chapter 19.4.2).

Similarly, we can generalize conditional maximum likelihood (introduced in Sec. 5.6.1) to other distributions than the Gaussian, as discussed below when defining the objective function for MLPs.

6.3 Parametrizing a Learned Predictor

There are many ways to define the family of input-output functions, cost function (including optional regularizers) and optimization procedure. The most common ones are described below, while more advanced ones are left to later chapters.

6.3.1 Family of Functions

A motivation for the family of functions defined by multi-layer neural networks is to *compose simple transformations in order to obtain highly non-linear ones*. In particular, MLPs compose affine transformations and element-wise non-linearities. As discussed in Sec. 6.6 below, with the appropriate choice of parameters, multi-layer neural networks can in principle approximate any smooth function, with more hidden units allowing one to achieve better approximations.

A multi-layer neural network with more than one hidden layer can be defined by generalizing the above structure. Here we provide one example of how to do so. In our example, we choose to use the hyperbolic tangent activation function[2] instead of the sigmoid activation function:

$$\boldsymbol{h}^{(k)} = \tanh(\boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)})$$

where $\boldsymbol{h}^0 = \boldsymbol{x}$ is the input of the neural net, $\boldsymbol{h}^{(k)}$ (for $k > 0$) is the output of the k-th hidden layer, which has weight matrix $\boldsymbol{W}^{(k)}$ and offset (or bias) vector $\boldsymbol{b}^{(k)}$. If we want the output $f(\boldsymbol{x};\boldsymbol{\theta})$ to lie in some desired range, then we typically define an *output non-linearity* (which we did not have in the above Example 6.1.1). The non-linearity for the output layer is generally different from the tanh, depending on the type of output to be predicted and the associated loss function (see below).

There are several other non-linearities besides the sigmoid and the hyperbolic tangent which have been successfully used with neural networks. In particular, we introduce some piece-wise linear units below such as the the rectified linear unit ($\max(0, b + \boldsymbol{w} \cdot \boldsymbol{x})$) and the maxout unit ($\max_i(b_i + \boldsymbol{W}_{:,i} \cdot \boldsymbol{x})$) which have been particularly successful in the case of deep feedforward or convolutional networks. A longer discussion of these can be found in Sec. 6.8.

These and other non-linear neural network activation functions commonly found in the literature are summarized below. Most of them are typically combined with an affine transformation $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}$ and applied element-wise:

$$\boldsymbol{h} = \phi(\boldsymbol{a}) \Leftrightarrow h_i = \phi(a_i) = \phi(b_i + \boldsymbol{W}_{i,:}\boldsymbol{x}). \tag{6.2}$$

- **Rectifier** or **rectified linear unit (ReLU)** or **positive part**: transformation of the output of the previous layer: $\phi(a) = \max(0, a)$, also written $\phi(a) = (a)^+$. This is by far the most popular hidden unit in current feedforward networks. Some other effective variants are based on using a non-zero slope α_i when $a < 0$: $h_i = \phi(a, \alpha_i) = \max(0, a) + \alpha_i \min(0, a)$. where α_i can be a small fixed value like 0.01 A *leaky ReLU* (Maas *et al.*, 2013) fixes α_i to a small value like 0.01 while a *parametric ReLU* or *PReLU* treats α_i as a learnable parameter (He *et al.*, 2015).

- **Hyperbolic tangent**: $\phi(a) = \tanh(a)$.

- **Sigmoid**: $\phi(a) = 1/(1 + e^{-a})$.

- **Softmax**: This is a vector-to-vector transformation $\phi(\boldsymbol{a}) = \mathrm{softmax}(\boldsymbol{a}) = e^{a_i}/\sum_j e^{a_j}$ such that $\sum_i \phi_i(\boldsymbol{a}) = 1$ and $\phi_i(\boldsymbol{a}) > 0$, i.e., the softmax output can be considered as a probability distribution over a finite set of outcomes. Note that it is not applied element-wise but on a whole vector of "scores". It

[2]The hyperbolic tangent function tanh is related to the sigmoid via $\tanh(\boldsymbol{x}) = 2\times\mathrm{sigmoid}(2\boldsymbol{x}) - 1$ and typically yields easier optimization with stochastic gradient descent (Glorot and Bengio, 2010a).

is mostly used as output non-linearity for predicting discrete probabilities over output categories. See definition and discussion below, around Eq. 6.4.

- **Radial basis function** or **RBF** unit: this one is not applied after a general affine transformation but acts on $\boldsymbol{x}$ using a different form that corresponds to a template matching, i.e., $h_i = \exp\left(-||\boldsymbol{w}_i - \boldsymbol{x}||^2/\sigma_i^2\right)$ (or typically with all the σ_i set to the same value). This is heavily used in kernel SVMs (Boser *et al.*, 1992; Schölkopf *et al.*, 1999) and has the advantage that such units can be easily initialized (Powell, 1987; Niranjan and Fallside, 1990) as a random (or selected) subset of the input examples, i.e., $\boldsymbol{w}_i = \boldsymbol{x}^{(t)}$ for some assignment of examples t to hidden unit templates i.

- **Softplus**: $\phi(a) = \zeta(a) = \log(1+e^a)$. This is a smooth version of the rectifier, introduced by Dugas *et al.* (2001) for function approximation and by Nair and Hinton (2010a) for the conditional distributions of undirected probabilistic models. Glorot *et al.* (2011a) compared the softplus and rectifier and found better results with the latter, in spite of the very similar shape and the differentiability and non-zero derivative of the softplus everywhere, contrary to the rectifier.

- **Hard tanh**: this is shaped similarly to the tanh and the rectifier but unlike the latter, it is bounded, $\phi(a) = \max(-1, \min(1, a))$. It was introduced by Collobert (2004).

- **Absolute value rectification**: $\phi(a) = |a|$ (may be applied on the affine dot product or on the output of a tanh unit). It is also a rectifier and has been used for object recognition from images (Jarrett *et al.*, 2009a), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

- **Maxout**: this is discussed in more detail in Sec. 6.8. It generalizes the rectifier but introduces multiple weight vectors $\boldsymbol{w}_i$ (called filters) for each hidden unit. $h_i = \max_i(b_i + \boldsymbol{w}_i \cdot \boldsymbol{x})$.

This is not an exhaustive list but covers most of the non-linearities and unit computations seen in the deep learning and neural nets literature. Many variants are possible.

As discussed in Sec. 6.4.3, the structure (also called *architecture*) of the family of input-output functions can be varied in many ways, which calls for a generic principle for efficiently computing gradients, described in that section. For example, a common variation is to connect layers that are not adjacent, with skip connections, which are found in the visual cortex (where the word "layer" should be replaced by the word "area"). Other common variations depart from a full connectivity between adjacent layers. For example, each unit at layer k may be connected to only a subset of units at layer $k-1$. A particular case of such form of sparse connectivity is discussed in chapter 9 with *convolutional networks*. The set of connections between units of the whole network needs to form a directed acyclic graph in order to define a meaningful computation (see the computational graph formalism below, Sec. 6.4.3). *Recurrent networks*, treated in 10, are typically depicted using graphs containing cycles. Such graphs are using a different kind of graphical language. The cycles indicate that the value of a unit at time step $t+1$ is a function of the value of the unit at time step t. These cyclical graphs in the recurrent network language can be unrolled into directed acyclic graphs containing multiple time steps in order to obtain a traditional computational graph.

6.3.2 Loss Function and Conditional Log-Likelihood

In the 80's and 90's the most commonly used loss function was the *squared error* $L(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) = ||f(\boldsymbol{x};\boldsymbol{\theta}) - \boldsymbol{y}||^2$. As discussed in Sec. 6.2, if f is unrestricted (non-parametric), minimizing the expected value of the loss function over some data-generating distribution $P(\mathbf{x},\mathbf{y})$ yields $f(\boldsymbol{x}) = \mathbb{E}[\mathbf{y} \mid \mathbf{x} = \boldsymbol{x}]$, the true conditional expectation of $\mathbf{y}$ given $\mathbf{x}$. This tells us what the neural network is trying to learn. Replacing the squared error by an absolute value makes the neural network try to estimate not the conditional expectation but the conditional median[3].

However, we typically want to minimize the cross-entropy [4] between the data distribution and the model distribution. In the case where y is either a hard, binary, target, or a soft target giving the probability of the output being 1, then the cross-entropy for the corresponding Bernoulli distribution is no longer a mean squared error. Instead, it is given by

$$L(f(\boldsymbol{x};\boldsymbol{\theta}),y) = -y\log f(\boldsymbol{x};\boldsymbol{\theta}) - (1-y)\log(1-f(\boldsymbol{x};\boldsymbol{\theta})). \tag{6.3}$$

It can be shown that the optimal (non-parametric) f minimizing this loss function is $f(\boldsymbol{x}) = P(\mathrm{y} = 1 \mid \boldsymbol{x})$. In other words, when maximizing the conditional log-likelihood objective function, we are training the neural net output to estimate conditional probabilities as well as possible in the sense of the KL divergence (see Sec. 3.13, Eq. 3.7). Note that in order for the above expression of the criterion to make sense, $f(\boldsymbol{x};\boldsymbol{\theta})$ must be strictly between 0 and 1 (an undefined or infinite value would otherwise arise). To achieve this, it is common to use the sigmoid as

[3]Showing this is another interesting exercise.

[4] Many authors use the term "cross-entropy" to identify specifically the negative log-likelihood of a Bernoulli or softmax distribution, but that is a misnomer. Any loss consisting of a negative log-likelihood is a cross entropy between the empirical distribution defined by the training set and the model. For example, mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.

non-linearity for the output layer, which matches well with the Bernoulli negative log-likelihood cost function[5]. As explained below (Softmax subsection), the log-likelihood of a Bernoulli variable whose mean is parameterized by a sigmoidal unit allows gradients to pass through the output non-linearity even when the neural network produces a confidently wrong answer, unlike the squared error criterion coupled with a sigmoid or softmax non-linearity.

Learning a Conditional Probability Model

More generally, one can define a loss function as corresponding to a conditional log-likelihood, i.e., the negative log-likelihood (NLL) cost function

$$L_{\text{NLL}}(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) = -\log P(\mathbf{y}=\boldsymbol{y} \mid \mathbf{x}=\boldsymbol{x};\boldsymbol{\theta}).$$

See Sec. 5.6.1 (and the one before) which shows that this criterion corresponds to minimizing the KL divergence between the model P of the conditional probability of $\mathbf{y}$ given $\mathbf{x}$ and the data generating distribution Q, approximated here by the finite training set, i.e., the empirical distribution of pairs $(\boldsymbol{x},\boldsymbol{y})$. Hence, minimizing this objective, as the amount of data increases, yields an estimator of the true conditional probability of $\mathbf{y}$ given $\mathbf{x}$.

For example, if $\mathbf{y}$ is a continuous random variable and we assume that, given $\mathbf{x}$, it has a Gaussian distribution with mean $f(\mathbf{x})$ and variance σ^2, then

$$-\log P(\mathbf{y} \mid \mathbf{x};\boldsymbol{\theta}) = \frac{1}{2}(f(\mathbf{x})-\mathbf{y})^2/\sigma^2 + \log(2\pi\sigma^2).$$

Up to an additive and multiplicative constant (which would give the same choice of $\boldsymbol{\theta}$), minimizing this negative log-likelihood is therefore equivalent to minimizing the squared error loss. Once we understand this principle, we can readily generalize it to other distributions, as appropriate. For example, it is straightforward to generalize the univariate Gaussian to the multivariate case, and under appropriate parametrization consider the variance to be a parameter or even a parametrized function of $\boldsymbol{x}$ (for example with output units that are guaranteed to be positive, or forming a positive definite matrix, as outlined below, Sec. 6.3.2).

Similarly, for binary variables, the Bernoulli negative log-likelihood cost function corresponds to the conditional log-likelihood associated with the Bernoulli distribution with probability $p = f(\boldsymbol{x};\boldsymbol{\theta})$ of generating $\mathrm{y}=1$ given $\mathbf{x}=\boldsymbol{x}$ (and probability $1-p$ of generating $\mathrm{y}=0$):

$$\begin{aligned} L_{\text{NLL}} &= -\log P(\mathrm{y} \mid \boldsymbol{x};\boldsymbol{\theta}) = -\mathbf{1}_{\mathrm{y}=1}\log p - \mathbf{1}_{\mathrm{y}=0}\log(1-p) \\ &= -\mathrm{y}\log f(\boldsymbol{x};\boldsymbol{\theta}) - (1-\mathrm{y})\log(1-f(\boldsymbol{x};\boldsymbol{\theta})). \end{aligned}$$

[5]In reference to statistical models, this "match" between the loss function and the output non-linearity is similar to the choice of a *link function* in generalized linear models (McCullagh and Nelder, 1989).

where $\mathbf{1}_{\mathrm{y}=1}$ is the usual binary indicator.

Softmax

When y is discrete and has a finite domain (say $\{1, \ldots, n\}$) but is not binary, the Bernoulli distribution is extended to the multinoulli distribution (defined in Sec. 3.9.2). This distribution is specified by a vector of $n-1$ probabilities whose sum is less than or equal to 1, each element of which provides the probability $p_i = P(\mathrm{y} = i \mid \boldsymbol{x})$. We can then recover $P(\mathrm{y} = n \mid \boldsymbol{x})$ as $1 - \sum_{i=1}^{n-1} P(\mathrm{y} = i \mid \boldsymbol{x})$. Alternatively, one can specify a vector of n probabilities whose sum is exactly 1. The two options have the same representational power but different learning dynamics.

The *softmax* non-linearity (Bridle, 1990): was designed for the purpose of specifying multinoulli distributions:

$$\boldsymbol{p} = \operatorname{softmax}(\boldsymbol{a}) \iff p_i = \frac{e^{a_i}}{\sum_j e^{a_j}}. \tag{6.4}$$

where typically $\boldsymbol{a}$ is a set of activations coming from the lower layers of the network. We often use the overparameterized $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}$, but we may hardcode a_n to 0 in order to specify only $n-1$ of the output probabilities. We can think of a as a vector of scores whose elements a_i are associated with each category i, with larger relative scores yielding exponentially larger probabilities. The corresponding loss function is therefore $L_{\mathrm{NLL}}(\boldsymbol{p}, y) = -\log p_y$. Note how minimizing this loss will push a_y up (increase the score a_y associated with the correct label y) while pushing down a_i for $i \neq y$ (decreasing the score of the other labels, in the context $\boldsymbol{x}$). The first effect comes from the numerator of the softmax while the second effect comes from the normalizing denominator. These forces cancel on a specific example only if $p_y = 1$ and they cancel in average over examples (say sharing the same x) if p_i equals the fraction of times that $\mathrm{y} = i$ for this value $\boldsymbol{x}$. To see this, consider the gradient with respect to the scores $\boldsymbol{a}$:

$$\begin{aligned}
\frac{\partial}{\partial a_k} L_{\mathrm{NLL}}(\boldsymbol{p}, y) &= \frac{\partial}{\partial a_k}(-\log p_y) = \frac{\partial}{\partial a_k}(-a_y + \log \sum_j e^{a_j}) \\
&= -\mathbf{1}_{y=k} + \frac{e^{a_k}}{{}_j\, e^{a_j}} \\
&= p_k - 1_{y=k} \quad \text{or}
\end{aligned}$$

$$\frac{\partial}{\partial \boldsymbol{a}} L_{\mathrm{NLL}}(\boldsymbol{p}, y) = (\boldsymbol{p} - \boldsymbol{e}_y) \ \sum \tag{6.5}$$

where $\boldsymbol{e}_y = [0, \ldots, 0, 1, 0, \ldots, 0]$ is the one-hot vector with a 1 at position y. Examples that share the same $\boldsymbol{x}$ share the same $\boldsymbol{a}$, so the average gradient on $\boldsymbol{a}$

over these examples is 0 when the average of the above expression cancels out, i.e., $\boldsymbol{p} = \mathbb{E}_y[\, \boldsymbol{e}_y \mid \boldsymbol{x}\,]$ where the expectation is over these examples. Thus the optimal p_i for these examples is the average number of times that $y = i$ among those examples. Over an infinite number of examples, we would find that the gradient is 0 when p_i perfectly estimates the true $P(y = i \mid \boldsymbol{x})$. What the above gradient decomposition teaches us as well is the division of the total gradient into (1) a term due to the numerator (the $\boldsymbol{e}_y$) and dependent on the actually observed target y and (2) a term independent of y but which corresponds to the gradient of the softmax denominator. The same principles and the role of the normalization constant (or "partition function") can be seen at play in the training of undirected models, in Chapter 16.

The softmax has other interesting properties. First of all, the gradient of $-logp(y = i \mid \boldsymbol{x})$ with respect to $\boldsymbol{a}$ only saturates in the case when $p(y = i \mid \boldsymbol{x})$ is already nearly maximal, approaching 1. Specifically, let us consider the case where the correct label is i, i.e. $\mathrm{y} = i$. The element of the gradient associated with an erroneous label, say $j \neq i$, is

$$\frac{\partial}{\partial a_j} L_{\mathrm{NLL}}(\boldsymbol{p}, y) = p_j. \tag{6.6}$$

So if the model correctly predicts a low probability that the $\mathrm{y} = j$, i.e. that $p_j \approx 0$, then the gradient is also close to zero. But if the model incorrectly and confidently predicts that j is the correct class, i.e., $p_j \approx 1$, there will be a strong push to reduce a_j. Conversely, if the model incorrectly and confidently predicts that the correct class y should have a low probability, i.e., $p_y \approx 0$, there will be a strong push (a gradient of about -1) to push a_y up. One way to see these is to imagine doing gradient descent on the a_j's themselves (that is what backprop is really based on): the update on a_j would be proportional to the negative gradient on a_j, so a positive gradient on a_j (e.g., incorrectly confident that $p_j \approx 1$) pushes a_j down, while a negative gradient on a_j (e.g., incorrectly confident that $p_y \approx 0$) pushes a_y up. In fact note how a_y is *always pushed up* because $p_y - 1_{y=y} = p_y - 1 < 0$, and the other scores a_j (for $j \neq y$) are always pushed down, because their loss gradient is $p_j > 0$.

There are other loss functions such as the squared error applied to softmax (or sigmoid) outputs (which was popular in the 80's and 90's) which have vanishing gradient when an output unit saturates (when the derivative of the non-linearity is near 0), *even if the output is completely wrong* (Solla *et al.*, 1988). This may be a problem because it means that the parameters will change extremely little, and may fail to change altogether due to numerical rounding, even though the output is wrong.

To see how the squared error interacts with the softmax output, we need to introduce a one-hot encoding of the label, $\boldsymbol{y} = \boldsymbol{e}_i = [0, \ldots, 0, 1, 0, \ldots, 0]$, i.e for the label $\mathrm{y} = i$, we have $\boldsymbol{y}_i = 1$ and $\boldsymbol{y}_j = 0, \forall j \neq i$. We will again consider that we have the output of the network to be $\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{a})$, where, as before, $\boldsymbol{a}$ is the input to the softmax function (e.g. $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}$ with $\boldsymbol{h}$ the output of the last hidden layer).

For the squared error loss $L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y}) = ||\boldsymbol{p}(\boldsymbol{a}) - \boldsymbol{y}||^2$, the gradient of the loss with respect to the input vector to the softmax, $\boldsymbol{a}$, is given by:

$$\begin{aligned}\frac{\partial}{\partial a_i} L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y}) &= \frac{\partial L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y})}{\partial \boldsymbol{p}(\boldsymbol{a})} \frac{\partial \boldsymbol{p}(\boldsymbol{a})}{\partial a_i} \\ &= \sum_j 2(p_j(\boldsymbol{a}) - \boldsymbol{y}_j) p_j (1_{i=j} - p_i). \end{aligned} \tag{6.7}$$

So if the model incorrectly predicts a low probability for the correct class $\mathrm{y} = i$, i.e., if $p_{\mathrm{y}} = p_i \approx 0$, then the score for the correct class, a_{y}, does not get pushed up in spite of a large error, i.e., $\frac{\partial}{\partial a_{\mathrm{y}}} L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y}) \approx 0$. This is one of many reasons that practitioners prefer to use the negative log-likelihood (cross-entropy) cost function, with the softmax non-linearity (as well as with the sigmoid non-linearity), rather than applying the squared error criterion to these probabilities. Negative log-likelihood has the added advantage of a straightforward probabilistic interpretation, as the maximum likelihood criterion for the softmax model.

Another useful property of the softmax is that its output is invariant to adding a scalar to all of its inputs:

$$\mathrm{softmax}(\boldsymbol{a}) = \mathrm{softmax}(\boldsymbol{a} + b).$$

This property is used to implement the numerically stable variant of the softmax, which exploits the fact that $\mathrm{softmax}(\boldsymbol{a}) = \mathrm{softmax}(\boldsymbol{a} - \max_i a_i)$. This allows us to evaluate softmax with only small numerical errors even when a contains extremely large or extremely negative numbers.

Finally, it is interesting to think of the softmax as a way to create a form of *competition* between the units (typically output units, but not necessarily) that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the *lateral inhibition* that is believed to exist between nearby neurons in cortex. At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of *winner-take-all* (one of the outputs is nearly 1 and the others are nearly 0).

Diverse Output Types Trained with Maximum Likelihood

We may wish to use many different types of output layers for our neural networks depending on whether we wish to predict binary, categorical, or real data, or even depending on whether we expect our outputs to cluster around a single point for each input $\boldsymbol{x}$ or whether we expect them to come from a rich multimodal distribution. The task of designing not only multiple kinds of output layers but also appropriate loss functions for each output layer may seem overwhelming. Fortunately, the principle of maximum likelihood provides a straightforward solution. We can view each output layer as defining a conditional probability distribution. The corresponding maximum likelihood loss function is then the negative log-likelihood under that distribution.

In general, for any parametric probability distribution $p(\mathbf{y} \mid \boldsymbol{\omega})$ with parameters $\boldsymbol{\omega}$, we can construct a *conditional distribution* $p(\mathbf{y} \mid \mathbf{x})$ by making $\boldsymbol{\omega}$ a parametrized function of $\mathbf{x}$ and learning that function:

$$p(\mathbf{y} \mid \boldsymbol{\omega} = f(\mathbf{x}))$$

where $f(\mathbf{x})$ is the *output* of a predictor, $\mathbf{x}$ is its input, and $\mathbf{y}$ can be thought of as a *"target"*. The use of the word "target" comes from the common cases of classification and regression, where $f(\mathbf{x})$ is really a prediction associated with random variable $\mathbf{y}$, or with its expected value. However, in general $\omega = f(\mathbf{x})$ may contain parameters of the distribution of $\mathbf{y}$ other than its expected value. For example, it could contain its variance or covariance, in the case where $\mathbf{y}$ is conditionally Gaussian. In the above examples, with the squared error loss, ω is the mean of the Gaussian which captures the conditional distribution of $\mathbf{y}$ (which means that the variance is considered fixed, not a function of $\mathbf{x}$). In the common classification case, ω contains the probabilities associated with the various events of interest.

Viewing the output of a neural network as the parameters of a conditional distribution is a powerful tool that helps us to design loss functions for essentially any neural network without needing to think hard about the appropriate loss function for each new type of output. Once we have defines the conditional probability distribution, we can simply train the model using the principle of maximum likelihood (Sec. 5.6.1). In other words, we simply use the negative log-likelihood of the data under the model distribution as the cost function for training our neural network. The negative log-likelihood is given by

$$L(\mathbf{x}, \mathbf{y}) = -\log p(\mathbf{y} \mid \omega = f(\mathbf{x})). \tag{6.8}$$

This approach automatically tells us to use mean squared error to penalize the parameters of a Gaussian conditional distribution with identity covariance. For other distributions, such as a discrete distribution parameterized by a softmax, we obtain different forms of the negative log-likelihood. We do not need to specially design each of these loss functions for each type of output layer—we just use the cost function that results from subsituting our definition of $p(\mathbf{y} \mid omega)$ into Eq. 6.8.

This approach is even more powerful, in the sense that the neural network can provide more than one kind of output, and the negative log-likelihood tells us how to train all of these outputs. For example, we may wish to learn the variance of a conditional Gaussian for $\mathbf{y}$, given $\mathbf{x}$. In the simple case, where the variance σ^2 is a constant, there is a closed form expression because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations $\mathbf{y}$ and their expected value. A computationally more expensive approach that does not require writing special-case code is to simply include the variance as one of the properties of the distribution $p(\mathbf{y} \mid \mathbf{x})$ that is controlled by $\omega = f(\mathbf{x}; \boldsymbol{\theta})$. Eq. 6.8 will then provide a cost function with the appropriate terms necessary to make our optimization procedure incrementally learn the variance. In the simple case where the variance does not depend on the input, then we can make new parameter in the network that is copied directly into ω. This new parameter might be σ itself or could be a parameter v representing σ^2 or it could be a parameter β representing $\frac{1}{\sigma^2}$, depending on how we choose to parameterize the distribution. We may wish our model to predict a different amount of variance in $\mathbf{y}$ for different values of $\mathbf{x}$. This is called a *heteroscedastic* model. In the heteroscedastic case, we simply make the specification of the variance be one of the values output by $f(\mathbf{x}; \boldsymbol{\theta})$. A typical way to do this is to formulate the Gaussian distribution using precision, rather than variance, as described in Eq. 3.2. In the multivariate case it is most common to use a diagonal precision matrix

$$\text{diag}(\boldsymbol{\beta})$$

. This formulation works well with gradient descent because the formula for the log-likelihood of the Gaussian distribution involves only multiplication by the precision for each element of the output and the addition of the logarithm of the precision for each element of the output. The gradient of multiplication, addition, and logarithm operations is well-behaved. By comparison, if we parameterized the output in terms of variance, we would need to use division. The division function becomes extremely steep near zero, and can cause difficulty for stochastic gradient descent. If we parameterized the output in terms of standard deviation, the log-likelihood would still involve division, and would also involve squaring. The gradient through the squaring operation can vanish near zero, making it difficult to learn parameters that are squared. Regardless of whether we use standard deviation, variance, or precision, we must ensure that the covariance matrix of the Gaussian is positive definite. Because the eigenvalues of the precision matrix are the reciprocals of the eigenvalues of the covariance matrix, this is equivalent to ensuring that the precision matrix is positive definite. If we use a diagonal matrix, or a scalar times the diagonal matrix, then the only condition we need to enforce on the output of the model is positivity. If we suppose that $\boldsymbol{a}$ is the raw activation of the model used to determine the diagonal precision, we can use the softplus function to obtain a positive precision vector: $\boldsymbol{\beta} = \zeta(\boldsymbol{a})$. This same strategy applies equally if using variance or standard deviation rather than precision or if using a scalar times identity rather than diagonal matrix.

It is rare to learn a covariance or precision matrix with richer structure than diagonal. If the covariance is full and conditional, then a parametrization must be chosen that guarantees positive-definiteness of the predicted covariance matrix. This can be achieved by writing $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^\top(\boldsymbol{x})$, where $\boldsymbol{B}$ is an unconstrained square matrix. One practical issue if the matrix is full is that computing the likelihood is expensive, requiring $O(d^3)$ computation for the determinant and inverse of $\boldsymbol{\Sigma}(\boldsymbol{x})$ (or equivalently, and more commonly done, its eigendecomposition or that of $\boldsymbol{B}(\boldsymbol{x})$).

We often want to perform multimodal regression, that is, to predict real values that come from a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that can have several different peaks in $\boldsymbol{y}$ space for the same value of $\boldsymbol{x}$. In this case, a Gaussian mixture is a natural representation for the output Jacobs *et al.* (1991); Bishop (1994). Neural networks with Gaussian mixtures as their output are often called *mixture density networks*. An Gaussian mixture output with n components is defined by the conditional probability distribution

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x}) \mathcal{N}(\boldsymbol{y} \mid \mu_i(\boldsymbol{x}), \boldsymbol{\Sigma}_i(\boldsymbol{x})).$$

The neural network must have three outputs: $p(\mathrm{c} = i \mid \boldsymbol{x})$, $\mu_i(\boldsymbol{x})$ and $\boldsymbol{\Sigma}_i(\boldsymbol{x})$. These outputs must satisfy different constraints:

1. Mixture components $p(\mathrm{c} = i \mid \boldsymbol{x})$: these form a multinoulli distribution over the n different components associated with latent variable[6] c, and can typically be obtained by a softmax over an n-dimensional vector, to guarantee that these outputs are positive and sum to 1.

2. Means $\mu_i(\boldsymbol{x})$: these indicate the center or mean associated with the i-th Gaussian component, and are unconstrained (typically with no non-linearity at all for these output units). If $\mathbf{y}$ is a d-vector, then the network must output an $n \times d$ matrix containing all n of these d-dimensional vectors. Learning these means with maximum likelihood is slightly more complicated than learning the means of a distribution with only one output mode. We only want to update the mean for the component that actually produced the observation. In practice, we do not know which component produced each

[6]We consider c to be latent because we do not observe it in the data: given input $\mathbf{x}$ and target $\mathbf{y}$, it is not possible to know with certainty which Gaussian component was responsible for $\mathbf{y}$, but we can imagine that $\mathbf{y}$ was generated by picking one of them, and make that unobserved choice a random variable.

observation. The expression for the negative log-likelihood naturally weights each example's contribution to the loss for each component by the probability that the component produced the example.

3. Covariances $\boldsymbol{\Sigma}_i(\boldsymbol{x})$: these specify the covariance matrix for each component i. As when learning a single Gaussian component, we typically use a diagonal matrix to avoid needing to compute determinants. As with learning the means of the mixture, maximum likelihood is complicated by needing to assign partial responsibility for each point to each mixture component. Gradient descent will automatically follow the correct process if given the correct specification of the negative log-likelihood under the mixture model.

It has been reported that gradient-based optimization of conditional Gaussian mixtures (on the output of neural networks) can be unreliable, in part because one gets divisions (by the variance) which can be numerically unstable (when some variance gets to be small for a particular example, yielding very large gradients). One solution is to *clip gradients* (see Sec. 10.8.7 and Mikolov (2012); Pascanu and Bengio (2012); Graves (2013); Pascanu *et al.* (2013a)), while another is to scale the gradients heuristically (Murray and Larochelle, 2014).

Multiple Output Variables

When $\mathbf{y}$ is actually a tuple formed by multiple random variables $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k)$, then one has to choose an appropriate form for their joint distribution, conditional on $\mathbf{x} = \boldsymbol{x}$. The simplest and most common choice is to assume that the $\mathbf{y}_i$ are conditionally independent, i.e.,

$$p(\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_k \mid \boldsymbol{x}) = \prod_{i=1}^{k} p(\mathrm{y}_i \mid \boldsymbol{x}).$$

This brings us back to the single variable case, especially since the log-likelihood now decomposes into a sum of terms $\log p(\mathbf{y}_i \mid \boldsymbol{x})$. If each $p(\mathbf{y}_i \mid \boldsymbol{x})$ is separately parametrized (e.g. a different neural network), then we can train these neural networks independently. However, a more common and powerful choice assumes that the different variables $\mathbf{y}_i$ share some common factors, given $\boldsymbol{x}$, that can be represented in some hidden layer of the network (such as the top hidden layer). See Sec.s 6.7 and 7.12 for a deeper treatment of the notion of underlying factors of variation and multi-task training: each $(\mathbf{x}, \mathbf{y}_i)$ pair of random variables can be associated with a different learning task, but it might be possible to exploit what these tasks have in common. See also Fig. 7.6 illustrating these concepts.

If the conditional independence assumption is considered too strong, what can we do? At this point it is useful to step back and consider everything we know about learning a joint probability distribution. Since any probability distribution $p(\mathbf{y};\boldsymbol{\omega})$ parametrized by parameters $\boldsymbol{\omega}$ can be turned into a conditional distribution $p(\mathbf{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ (by making $\boldsymbol{\omega}$ a function $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$ parametrized by $\boldsymbol{\theta}$), we can go beyond the simple parametric distributions we have seen above (Gaussian, Bernoulli, multinoulli), and use more complex joint distributions. If the set of values that $\mathbf{y}$ can take is small enough (e.g., we have 8 binary variables $\mathbf{y}_i$, i.e., a joint distribution involving $2^8 = 256$ possible values), then we can simply model all these joint occurences as separate values, e.g., with a softmax and multinoulli over all these configurations. However, when the set of values that $\mathbf{y}_i$ can take cannot be easily enumerated and the joint distribution is not unimodal or factorized, we need other tools. The third part of this book is about the frontier of research in deep learning, and much of it is devoted to modeling such complex joint distributions, also called *graphical models*: see Chapters 16, 18, 19, 20. In particular, Sec. 12.5 discusses how sophisticated joint probability models with parameters $\boldsymbol{\omega}$ can be coupled with neural networks that compute $\boldsymbol{\omega}$ as a function of inputs $\boldsymbol{x}$, yielding *structured output* models conditioned with deep learning.

6.3.3 Cost Functions for Neural Networks

Typically, the training criteria for neural networks are primarily based on maximum likelihood. In the case of supervised learning, this will be the conditional version of maximum likelihood when we perform supervised learning.

In addition to the negative log-likelihood cost, we also often add some sort of a regularization term. Many of the regularization terms that apply to linear models also apply to neural networks. For example, weight decay applies to neural networks as well as to linear models.

These ideas have all been described for machine learning models in general in Chapter 5. When designing cost functions for neural networks specifically, we can often specialize the regularization terms for neural networks in various ways, such as controlling the properties of the individual hidden units. These strategies are covered in Chapter 7.

In practice, a good choice for the criterion is maximum likelihood regularized with dropout, possibly also with weight decay.

6.3.4 Optimization Procedure

Previously, we have seen simple machine learning models which could sometimes be fit in closed form. Neural networks must essentially always be optimized with iterative procedures. Optimization of neural networks is so difficult that the choice of optimization procedure is often tightly intertwined with the choice of model. In other words, we often design the model to make optimization easier. Chapter 8 is devoted to the iterative optimization procedures used to train neural networks and other deep models, including optimization strategies that involve designing the model to be easier to optimize.

In practice, a good choice for the optimization algorithm for a feedforward network is usually stochastic gradient descent with momentum. Typically, to make the model easier to optimize, it is best to use piecewise linear hidden units.

6.4 Computational Graphs and Back-Propagation

The term *back-propagation* is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually it just means the method for computing gradients in such networks. Furthermore, it is generally understood as something very specific to multi-layer neural networks, but once its derivation is understood, it can easily be generalized to arbitrary families of parametrized functions (for which computing a gradient is meaningful) and their corresponding computational graph. We describe this generalization here, focusing on the case of interest in machine learning where the output of the function to differentiate (e.g., the training criterion J) is a scalar and we are interested in its derivative with respect to a set of parameters (considered to be the elements of a vector $\boldsymbol{\theta}$), or equivalently, a set of inputs[7] . The partial derivative of J with respect to $\boldsymbol{\theta}$ (called the gradient) tells us whether $\boldsymbol{\theta}$ should be increased or decreased in order to decrease J, and is a crucial tool in optimizing the training objective. It can be readily proven that the back-propagation algorithm for computing gradients has optimal computational complexity in the sense that there is no algorithm that can compute the gradient faster (in the $O(\cdot)$ sense, i.e., up to an additive and multiplicative constant).

The basic idea of the back-propagation algorithm is that the partial derivative of the cost J with respect to parameters $\boldsymbol{\theta}$ can be *decomposed recursively* by taking into consideration the composition of functions that relate $\boldsymbol{\theta}$ to J , via intermediate quantities that mediate that influence, e.g., the activations of hidden units in a deep neural network.

6.4.1 Chain Rule

The basic mathematical tool for considering derivatives through compositions of functions is the **chain rule**, illustrated in Fig. 6.4. The partial derivative $\frac{\partial y}{\partial x}$ measures the locally linear influence of a variable x on another one y, while we denote $\nabla_{\boldsymbol{\theta}} J$ for the gradient vector of a scalar J with respect to some vector of

[7]It is useful to know which inputs contributed most to the output or error made, and the sign of the derivative is also interesting in that context.

variables $\boldsymbol{\theta}$. If x influences y which influences z, we are interested in how a tiny change in x propagates into a tiny change in z via a tiny change in y. In our case of interest, the "output" is the cost, or objective function $z = J(g(\boldsymbol{\theta}))$, we want the gradient with respect to some parameters $x = \boldsymbol{\theta}$, and there are intermediate quantities $y = g(\boldsymbol{\theta})$ such as neural net activations. The gradient of interest can then be decomposed, according to the chain rule, into

$$\nabla_{\boldsymbol{\theta}} J(g(\boldsymbol{\theta})) = \nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta})) \frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{6.9}$$

which works also when J, g or $\boldsymbol{\theta}$ are vectors rather than scalars (in which case the corresponding partial derivatives are understood as Jacobian matrices of the appropriate dimensions). In the purely scalar case we can understand the chain rule as follows: a small change in $\boldsymbol{\theta}$ will propagate into a small change in $g(\boldsymbol{\theta})$ by getting multiplied by $\frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$. Similarly, a small change in $g(\boldsymbol{\theta})$ will propagate into a small change in $J(g(\boldsymbol{\theta}))$ by getting multiplied by $\nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta}))$. Hence a small change in $\boldsymbol{\theta}$ first gets multiplied by $\frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ to obtain the change in $g(\boldsymbol{\theta})$ and this then gets multiplied by $\nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta}))$ to obtain the change in $J(g(\boldsymbol{\theta}))$. Hence the ratio of the change in $J(g(\boldsymbol{\theta}))$ to the change in $\boldsymbol{\theta}$ is the product of these partial derivatives.

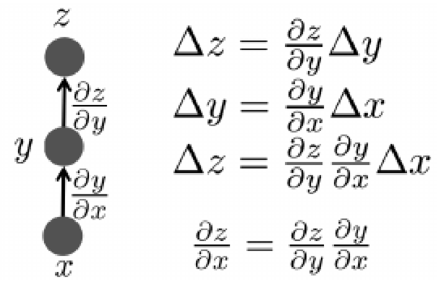
z
$\frac{\partial z}{\partial y}$
y
$\frac{\partial y}{\partial x}$
x
$\Delta z = \frac{\partial z}{\partial y} \Delta y$
$\Delta y = \frac{\partial y}{\partial x} \Delta x$
$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x$
$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$

Figure 6.4: The chain rule, illustrated in the simplest possible case, with z a scalar function of y, which is itself a scalar function of x. A small change Δx in x gets turned into a small change Δy in y through the partial derivative $\frac{\partial y}{\partial x}$, from the first-order Taylor approximation of $y(x)$, and similarly for $z(y)$. Plugging the equation for Δy into the equation for Δz yields the chain rule.

Now, if g is a vector, we can rewrite the above as follows:

$$\nabla_{\boldsymbol{\theta}} J(g(\boldsymbol{\theta})) = \sum_i \frac{\partial J(g(\boldsymbol{\theta}))}{\partial g_i(\boldsymbol{\theta})} \frac{\partial g_i(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

which sums over the influences of $\boldsymbol{\theta}$ on $J(g(\boldsymbol{\theta}))$ through all the intermediate variables $g_i(\boldsymbol{\theta})$. This is illustrated in Fig. 6.5 with $x = \boldsymbol{\theta}$, $y_i = g_i(\boldsymbol{\theta})$, and $z = J(g(\boldsymbol{\theta}))$.

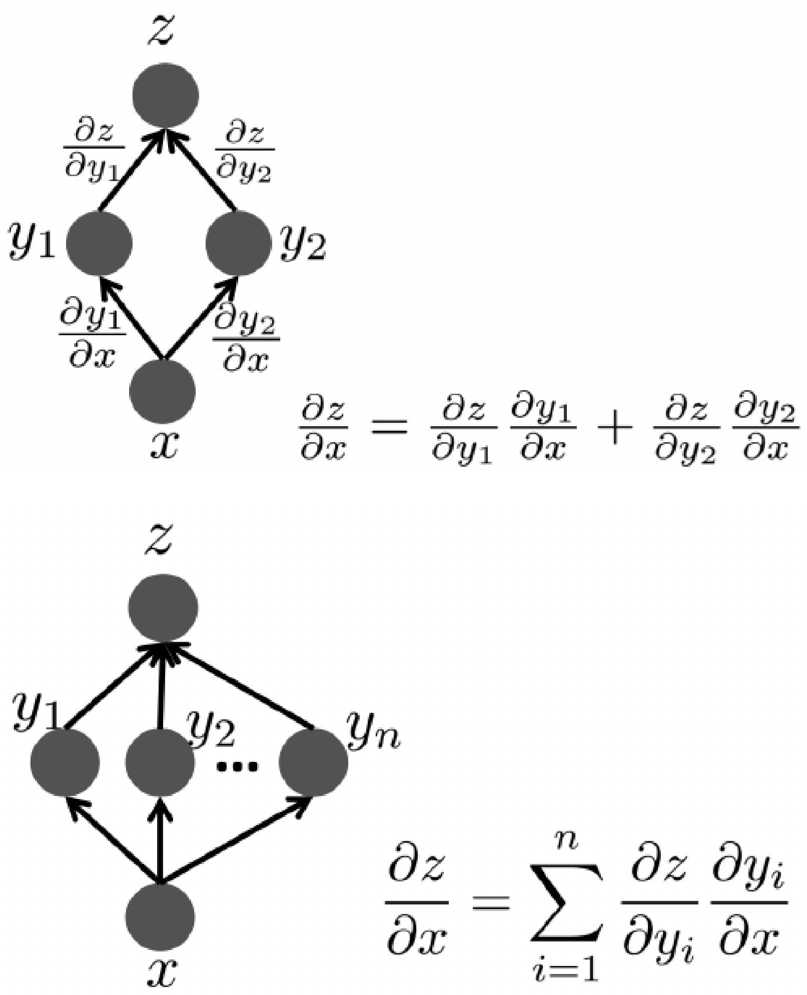
z
$\frac{\partial z}{\partial y_1}$
$\frac{\partial z}{\partial y_2}$
y_1
y_2
$\frac{\partial y_1}{\partial x}$
$\frac{\partial y_2}{\partial x}$
x
$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$
z
y_1
y_2
...
y_n
x
$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$

Figure 6.5: Top: The chain rule, when there are two intermediate variables y_1 and y_2 between x and z, creating two paths for changes in x to propagate and yield changes in z Bottom: more general case, with n intermediate variables y_1 to y_n.

6.4.2 Back-Propagation in an MLP

Example 6.1.1 illustrated the case of an MLP with a single hidden layer. In this section we extend back-propagation to a deep MLP. This MLP is the same as before, but with multiple hidden layers rather than a single hidden layer. For this purpose, we will recursively apply the chain rule illustrated in Fig. 6.5. The algorithm proceeds by first computing the gradient of the cost J with respect to output units, and these are used to compute the gradient of J with respect to the top hidden layer activations, which directly influence the outputs. We can then continue computing the gradients of lower level hidden units one at a time in the same way. The gradients on hidden and output units can be used to compute the gradient of J with respect to the parameters. In a typical network divided into many layers with each layer parameterized by a weight matrix and a vector of

Algorithm 6.1 *Forward* computation associated with input $\boldsymbol{x}$ for a deep neural network with ordinary affine layers composed with an arbitrary elementwise differentiable (almost everywhere) non-linearity f. There are l such layers, each mapping their vector-valued input $\boldsymbol{h}^{(k)}$ to a pre-activation vector $\boldsymbol{a}^{(k)}$ via a weight matrix $\boldsymbol{W}^{(k)}$ which is then transformed via f into $\boldsymbol{h}^{(k+1)}$. The input vector $\boldsymbol{x}$ corresponds to $\boldsymbol{h}^{(0)}$ and the predicted outputs $\hat{\boldsymbol{y}}$ corresponds to $\boldsymbol{h}^{(l)}$. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on a target $\boldsymbol{y}$ (see Sec. 6.3.2 for examples of loss functions). The loss may be added to a regularizer Ω (see Sec. 6.3.3 and Chapter 7) to obtain the example-wise cost J. Algorithm 6.2 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For computational efficiency on modern computers (especially GPUs), it is important to implement these equations on minibatches. Rather than using a vector $\boldsymbol{h}^{(k)}$ (and similary $\boldsymbol{a}^{(k)}$) to represent the activations on one example, an efficient implementation should use a matrix with the additional dimension representing the index of an example within the minibatch. Accordingly, $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ should have an additional dimension for the example index in the minibatch. The cost function J remains a scalar because it is the average cost across examples in the minibatch.

```
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ do
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega$
```

biases, the gradient on the weights and the biases is determined by the input to the layer and the gradient on the output of the layer.

Algorithm 6.1 describes in matrix-vector form the forward propagation computation for a classical multi-layer network with l layers, where each layer computes an affine transformation (defined by a bias vector $\boldsymbol{b}^{(k)}$ and a weight matrix $\boldsymbol{W}^{(k)}$) followed by a non-linearity f. In general, the non-linearity may be different on different layers. Typically at least the output layer has a different type than the other layers (see Sec. 6.3.1). Each unit at layer k computes an output $\boldsymbol{h}_i^{(k)}$ as follows:

$$\begin{aligned} a_i^{(k)} &= b_i^{(k)} + \sum_j W_{ij}^{(k)} h_j^{(k-1)} \\ h_i^{(k)} &= \phi(a_i^{(k)}) \end{aligned} \tag{6.10}$$

where we separate the affine transformation from the non-linear activation opera-

Algorithm 6.2 *Backward* computation for the deep neural network of Algorithm 6.1, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods. Note that although this algorithm as presented has pedagogical value, it is not a flexible way of implementing back-propagation because it is tied to the particulars of the network architecture and computation. Instead, consider the generalized form of back-propagation described in Sec. 6.4.3 below, which can accommodate any computational graph.

```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, y) + \lambda \nabla_{\hat{\boldsymbol{y}}} \Omega$
(typically $\Omega$ is only a function of parameters not activations, so the last term would be zero)
**for** $k = l, l-1, \ldots, 1$ **do**
    Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
    Compute gradients on weights and biases (including the regularization term, where needed):
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega$
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega$
    Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
**end for**
```

tions ϕ for ease of exposition of the back-propagation computations.

These are described in matrix-vector form by Algorithm 6.2 and proceed from the output layer towards the first hidden layer, as outlined above. Note that this way of implementing back-propagation is not flexible enough to accommodate changes in architecture, and is considered obsolete, replaced by the generic form of back-propagation with automatic differentation, described next, and which is not specific to the machine learning context.

6.4.3 Back-Propagation in a General Computational Graph

In this section we call the intermediate quantities between inputs (parameters $\boldsymbol{\theta}$) and output (cost J) of the graph nodes u_j (indexed by j) and consider the general case in which they form a directed acyclic graph that has J as its final node u_n, that depends of all the other nodes u_j. The back-propagation algorithm exploits the chain rule for derivatives to compute $\frac{\partial J}{\partial u_j}$ when $\frac{\partial J}{\partial u_i}$ has already been computed for successors u_i of u_j in the graph, e.g., the hidden units in the next layer downstream. This recursion can be initialized by noting that $\frac{\partial J}{\partial u_n} = \frac{\partial J}{\partial J} = 1$ and at each step only requires to use the partial derivatives associated with each arc of the graph, $\frac{\partial u_i}{\partial u_j}$, when u_i is a successor of u_j.

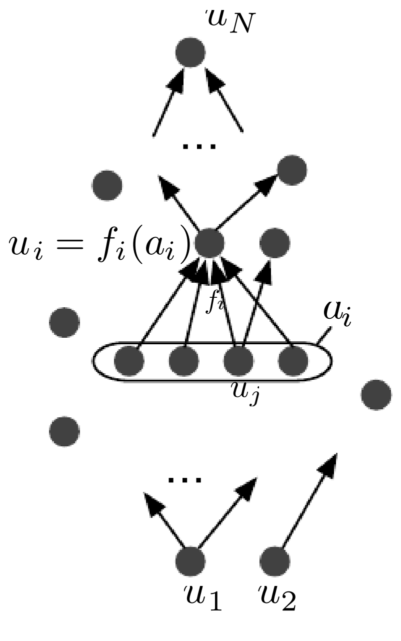

Figure 6.6: Illustration of recursive forward computation, where at each node u_i we compute a value $u_i = f_i(\boldsymbol{a}^{(i)})$, with $\boldsymbol{a}^{(i)}$ being the list of values from parents u_j of node u_i. Following Algorithm 6.3, the overall inputs to the graph are $u_1, \ldots, u_m$ (e.g., the parameters we may want to tune during training), and there is a single scalar output u_n (e.g., the loss which we want to minimize).

Algorithm 6.3 A computational graph describing forward propagation. Each node computes numerical value u_i by applying a function f_i to its argument list $\boldsymbol{a}^{(i)}$ that comprises the values of previous nodes u_j, $j < i$, with $j \in \text{parents}(i)$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first m nodes u_1 to u_m. The output of the computational graph is read off the last (output) node u_n.

```
for $i = 1, \ldots, m$ do
    $u_i \leftarrow x_i$
end for
for $i = m + 1, \ldots, n$ do
    $\boldsymbol{a}^{(i)} \leftarrow (u_j)_{j \in \text{parents}(i)}$
    $u_i \leftarrow f_i(\boldsymbol{a}^{(i)})$
end for
return $u_n$
```

More generally than multi-layered networks, we can think about decomposing a function $J(\boldsymbol{\theta})$ into a more complicated graph of computations. This graph is called a **computational graph**. Each node u_i of the graph denotes a numerical quantity that is obtained by performing a computation requiring the values u_j of other nodes, with $j < i$. The nodes satisfy a partial order which dictates in what order the computation can proceed. In practical implementations of such functions (e.g. with the criterion $J(\boldsymbol{\theta})$ or its value estimated on a minibatch), the final computation is obtained as the *composition of simple functions* taken from a given set (such as the set of numerical operations that the `numpy` library can perform on arrays of numbers).

We will define the back-propagation in a general flow-graph, using the following generic notation: $u_i = f_i(\boldsymbol{a}^{(i)})$, where $\boldsymbol{a}^{(i)}$ is a list of arguments for the application of f_i to the values u_j for the parents of i in the graph: $\boldsymbol{a}^{(i)} = (u_j)_{j \in \text{parents}(i)}$. This is illustrated in Fig. 6.6.

The overall computation of the function represented by the computational graph can thus be summarized by the forward computation algorithm, Algorithm 6.3.

In addition to having some code that tells us how to compute $f_i(\boldsymbol{a}^{(i)})$ for some values in the vector $\boldsymbol{a}^{(i)}$, we also need some code that tells us how to compute its partial derivatives, $\frac{\partial f_i(\boldsymbol{a}^{(i)})}{\partial a_{ik}}$ with respect to any immediate argument a_{ik}. Let $k = \pi(i, j)$ denote the index of u_j in the list $\boldsymbol{a}^{(i)}$. Note that u_j could influence u_i through multiple paths. Whereas $\frac{\partial u_i}{\partial u_j}$ would denote the total gradient adding up all of these influences, $\frac{\partial f_i(\boldsymbol{a}^{(i)})}{\partial a_{ik}}$ only denotes the derivative of f_i with respect to its specific k-th argument, keeping the other arguments fixed, i.e., only considering

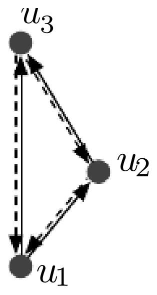

Figure 6.7: Illustration of indirect effect and direct effect of variable u_1 on variable u_3 in a computational graph, which means that the derivative of u_3 with respect to u_1 must include the sum of two terms, one for the direct effect (derivative of u_3 with respect to its first argument) and one for the indirect effect through u_2 (involving the product of the derivative of u_3 with respect to u_2 times the derivative of u_2 with respect to u_1). Forward computation of u_i's (as in Fig. 6.6) is indicated with upward full arrows, while backward computation (of derivatives with respect to u_i's, as in Fig. 6.8) is indicated with downward dashed arrows.

the influence through the arc from u_j to u_i.

In general, when manipulating partial derivatives, it is important to distinguish two kinds of derivatives: those that consider all the other arguments fixed (this is typically what we mean by partial derivative), and those that allow for indirect influences going through other arguments. The latter is typically what we mean by total derivatives, except that in our case there will simultaneously be multiple such arguments (the different parameters, or source nodes of the graph) for which we wish to compute the total derivative, holding the other source nodes fixed. For that reason, we still use the partial derivative notation for these, but we need to be aware of the different kinds of derivatives we are handling, which will depend on what type of node (internal or source) is under consideration.

For example consider $f_3(a_{3,1}, a_{3,2}) = e^{a_{3,1}+a_{3,2}}$ and $f_2(a_{2,1}) = a_{2,1}^2$, while $u_3 = f_3(u_2, u_1)$ and $u_2 = f_2(u_1)$, illustrated in Fig. 6.7. The direct derivative of f_3 with respect to its argument $a_{3,2}$ (keeping $a_{3,1}$ fixed) is $\frac{\partial f_3}{\partial a_{3,2}} = e^{a_{3,1}+a_{3,2}}$. On the other hand, if we consider the variables u_3 and u_1 to which these arguments correspond, there are two paths from u_1 to u_3, and we obtain as the total derivative the sum of partial derivatives over these two paths, $\frac{\partial u_3}{\partial u_1} = e^{u_1+u_2}(1 + 2u_1)$. The results are different because $\frac{\partial u_3}{\partial u_1}$ involves not just the direct dependency of u_3 on u_1 but also the indirect dependency through u_2.

Armed with this understanding, we can define the back-propagation algorithm as follows, in Algorithm 6.4, which would be computed *after* the forward propagation (Algorithm 6.3) has been performed. Note the recursive nature of the application of the chain rule, in Algorithm 6.4: we compute the gradient on node j by re-using the already computed gradient for children nodes i, starting the recurrence from

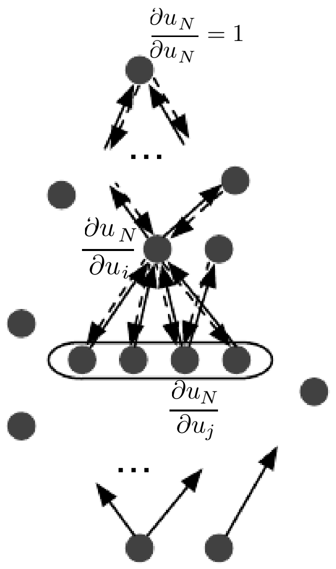
$\frac{\partial u_N}{\partial u_N} = 1$
$\frac{\partial u_N}{\partial u_i}$
$\frac{\partial u_N}{\partial u_j}$

Figure 6.8: Illustration of recursive backward computation, where we associate to each node j not just the values u_j computed in the forward pass (Fig. 6.6, bold upward arrows) but also the gradient $\frac{\partial u_n}{\partial u_j}$ with respect to the output scalar node u_n. These gradients are recursively computed in exactly the opposite order, as described in Algorithm 6.4 by using the already computed $\frac{\partial u_n}{\partial u_i}$ of the children i of j (dashed downward arrows).

the trivial $\frac{\partial u_n}{\partial u_n} = 1$ that sets the gradient for the output node. This is illustrated in Fig. 6.8.

This recursion is a form of efficient factorization of the total gradient, i.e., it is an application of the principles of dynamic programming[8]. Indeed, the derivative of the output node with respect to any node can also be written down in this intractable form:

$$\frac{\partial u_n}{\partial u_i} = \sum_{\text{paths } u_{k_1},\dots,u_{k_l}:\, k_1=i,k_l=n} \prod_{j=2}^{n} \frac{\partial u_{k_j}}{\partial u_{k_{j-1}}}$$

where the paths $u_{k_1},\dots,u_{k_l}$ go from the node $k_1 = i$ to the final node $k_l = n$ in the computational graph and $\frac{\partial u_{k_j}}{\partial u_{k_{j-1}}}$ refers only to the immediate derivative considering $u_{k_{j-1}}$ as the argument number $\pi(k_j, k_{j-1})$ of a_{k_j} into u_{k_j}, i.e.,

$$\frac{\partial u_{k_j}}{\partial u_{k_{j-1}}} = \frac{\partial f_{k_j}(a_{k_j})}{\partial a_{k_j,\pi(k_j,k_{j-1})}}.$$

[8] Here we refer to "dynamic programming" in the sense of table-filling algorithms that avoid re-computing frequently used subexpressions. In the context of machine learning, "dynamic programming" can also refer to iterating Bellman's equations. That is not the kind of dynamic programming we refer to here.

Algorithm 6.4 *Back-propagation* computation of a computational graph (full, upward arrows, Figs.6.8 and 6.6), which itself produces an additional computational graph (dashed, backward arrows). See the forward propagation in a flow-graph (Algorithm 6.3, to be performed first) and the required data structure. In addition, a quantity $\frac{\partial u_n}{\partial u_i}$ needs to be stored (and computed) at each node, for the purpose of gradient back-propagation. Below, the notation $\pi(i, j)$ is the index of u_j as an argument to f_i. The back-propagation algorithm efficiently computes $\frac{\partial u_n}{\partial u_i}$ for all i's (traversing the graph backwards this time), and in particular we are interested in the derivatives of the output node u_n with respect to the "inputs" $u_1, \dots, u_m$ (which could be the parameters, in a learning setup). The cost of the overall computation is proportional to the number of arcs in the graph, assuming that the partial derivative associated with each arc requires a constant time. This is of the same order as the number of computations for the forward propagation.

```
$\frac{\partial u_n}{\partial u_n} \leftarrow 1$
**for** $j = n - 1$ down to 1 **do**
    $\frac{\partial u_n}{\partial u_j} \leftarrow \sum_{i:j \in \text{parents}(i)} \frac{\partial u_n}{\partial u_i} \frac{\partial f_i(a_i)}{\partial a_{i,\pi(i,j)}}$
**end for**
**return** $\left(\frac{\partial u_n}{\partial u_i}\right)_{i=1}^{m}$
```

Computing the sum as above would be intractable because the number of possible paths can be exponential in the depth of the graph. The back-propagation algorithm is efficient because it employs a dynamic programming strategy to reuse rather than re-compute partial sums associated with the gradients on intermediate nodes.

Although the above was stated as if the u_i's were scalars, exactly the same procedure can be run with u_i's being tuples of numbers (more easily represented by vectors). The same equations remain valid. Earlier, we wrote these equations using scalar multiplication of scalar partial derivatives. Using vectors, these multiplications turn into matrix-vector products. We multiply the row vector of gradients $\frac{\partial u_n}{\partial u_i}$ by a Jacobian matrix of partial derivatives associated with the $j \rightarrow i$ arc of the graph, $\frac{\partial f_i(a_i)}{\partial a_{i,\pi(i,j)}}$.

The quintessential neural network training scenario is the case where each $\mathbf{U}^{(i)}$ is a matrix containing m examples in a minibatch and n hidden unit activation values. In this case, both forward propagation and back-propagation can be expressed as a product between a matrix of activations or gradients and a matrix of weights. From a computational point of view, this is much more efficient than training on a single example at a time. When we do not use the minibatch version of the algorithm, $\mathbf{U}^{(i)}$ is only a vector. The main operation used in forward and back-propagation is then a matrix-vector product, between the weight matrix and the vector of activations or gradients. Matrix-matrix products are typically implemented with a high degree of parallel computation (e.g. in BLAS library implementations) which is essential for obtaining good performance on modern multicore CPUs and GPUs. Matrix-matrix products for minibatch forward and back-propagation allow parallelization across both examples and units, while matrix-vector products for the processing of a single example allow only parallelization across units.

6.4.4 Symbolic Back-propagation and Automatic Differentiation

The algorithm for generalized back-propagation (Alg. 6.4) was presented with the interpretation that actual computations take place at each step of the algorithm. This generalized form of back-propagation is just a particular way to perform *automatic differentiation* (Rall, 1981) in computational graphs defined by Algorithm 6.3. Automatic differentiation automatically obtains derivatives of a given expression and has numerous uses in machine learning (Baydin *et al.*, 2015). As an alternative (and often as a debugging tool) derivatives could be obtained by numerical methods based on measuring the effects of small changes, called *numerical differentiation* (Lyness and Moler, 1967). For example, a finite difference approximation of the gradient follows from the definition of derivative as a ratio of the change in output that results in a change in input, divided by the change in input. Methods based on random perturbations also exist which randomly jiggle all the input variables (e.g. parameters) and associate these random input changes with the resulting overall change in the output variable in order to estimate the gradient (Spall, 1992).

However, for obtaining a gradient (i.e., with respect to many variables, e.g., parameters of a neural network), back-propagation has two advantages over numerical differentiation: (1) it performs exact computation (up to machine precision), and (2) it is computationally much more efficient, obtaining all the required derivatives in one go. Instead, numerical differentiation methods either require to redo the forward propagation separately for each parameter (keeping the other ones fixed) or they yield stochastic estimators (from a random perturbation of all parameters) whose variances grows linearly with the number of parameters. Automatic differentiation of a function with d inputs and m outputs can be done either by carrying derivatives forward, known as forward mode computation, or carrying them backwards, known as backward mode computation. The former is more efficient when $d < m$ and the latter is more efficient when $d > m$. In our use case, the output is a scalar (the cost), and the backward approach, also called reverse accumulation. Hence for machine learning applications where we want to compute gradients (partial derivatives with respect to a scalar cost), back-propagation is much more efficient than the approach of propagating derivatives forward in the graph.

Although Algorithm 6.4 can thus be seen as a particular form of automatic differentiation, its implementation can be done in different ways, the most general one involving *symbolic differentiation*. Symbolic differentation exploits our knowledge of how to compute derivatives of elementary operations, such as arithmetic operations, or any computation performed in the computer and specified by a symbolic expression. Symbolic differentiation takes a symbolic expression (for computing a function) and returns another symbolic expression (for computing the required derivatives). Automatic differentiation techniques such as the forward or backward mode allow one to perform these computations efficiently, avoiding a potentially exponential blow-up in the size of the computational graph for computing the derivatives.

The popular Torch library (Collobert *et al.*, 2011b) for deep learning, like most other open source deep learning libraries, provides a limited form of automatic differentiation restricted to the "programs" obtained by composing a predefined set of operations, each corresponding to a "module". The set of these modules is designed such that many neural network architectures and computations can be performed by composing the building blocks represented by each of these modules. Each module is defined by two main functions:

1. One that computes the outputs $\boldsymbol{y}$ of the module given its inputs $\boldsymbol{x}$, e.g., with an "`fprop`" function
$$\boldsymbol{y} = \text{module.}\texttt{fprop}(\boldsymbol{x}),$$
and the input ($\boldsymbol{x}$), output ($\boldsymbol{y}$) and potentially intermediate results of this computation are *stored in the module.*

2. One that computes the gradient $\frac{\partial J}{\partial \boldsymbol{x}}$ of a scalar (typically the minibatch cost J) with respect to the inputs $\boldsymbol{x}$, given the gradient $\frac{\partial J}{\partial \boldsymbol{y}}$ with respect to the outputs, e.g., with a "`bprop`" function
$$\nabla_{\boldsymbol{x}} J = \text{module.}\texttt{bprop}\left(\nabla_{\boldsymbol{y}} J\right).$$

The `bprop` function thus implicitly knows the Jacobian of the $\boldsymbol{x}$ to $\boldsymbol{y}$ mapping, $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$, at $\boldsymbol{x}$. Specifically, the `bprop` function specifies how to multiply this Jacobian by a vector passed to the function as an argument. It also needs to have access to the value of $\boldsymbol{x}$ that was previously fed as input to `fprop`, so that the Jacobian is computed at that value. For avoiding recomputations, $\boldsymbol{y}$ and other values that were computed in `fprop` are also typically stored where the module has access to these values.

During execution of the back-propagation algorithm, `bprop` will be called with $\nabla_{\boldsymbol{y}} J$ given as argument. When called in this manner, `bprop` computes

$$\nabla_{\boldsymbol{x}} J = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}^{\top} \nabla_{\boldsymbol{y}} J.$$

In practice, implementations work in parallel over a whole minibatch (transforming matrix-vector operations into matrix-matrix operations) and may operate on objects which are not vectors (maybe higher-order tensors like those involved with images or sequences of vectors). Furthermore, the `bprop` function does not have to explicitly compute the Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ and perform an actual matrix multiplication: it can do that matrix multiplication implicitly, which is often more efficient. For example, if the true Jacobian is diagonal, then the actual number of computations required is much less than the size of the Jacobian matrix.

To keep computations efficient and avoid the overhead of the glue required to compose modules together, neural net packages such as Torch define modules that perform coarse-grained operations such as the cross-entropy loss, a convolution, the affine operation associated with a fully-connected neural network layer, or a softmax. It means that if one wants to write differentiable code for some computation that is not covered by the existing set of modules, one has to write their own code for a new module, providing both the code for `fprop` and the code for `bprop`. This is in contrast with standard automatic differentiation systems, which know how to compute derivatives through all the operations in a general-purpose programming language such as C.

This is how software libraries including as Theano (Bergstra *et al.*, 2010b; Bastien *et al.*, 2012) and TensorFlow (Abadi *et al.*, 2015) handle derivatives. The user provides a symbolic representation of the function to the library, and the library returns a symbolic representation of that function's gradient. Crucially, the expression for the gradient is in the same language as the expression for the original function. Because the language is closed under differentiation, it is possible to compute higher-order derivatives by repeated calls to the differentation operator. Libraries such as Torch that express the gradients in a different way than they express the original function do not allow the computation of such higher order derivatives. The differentation machinery provided by Theano and TensorFlow is unambiguously a form of symbolic differentiation. If Theano and TensorFlow can be considered programming languages, this process may also be considered automatic differentiation of the function to be differentiated.

Theano and TensorFlow graphs are described by using a set of primitive operations to build more complicated expressions. The set of basic operations includes most of the linear algebra operations and other operations on tensors and linear algebra defined by standard numerical computation libraries such as `numpy`. Both libraries also include many other primitive operations that are commonly used for neural networks, such as convolution. It is thus very rare that a user would need to write a new primitive operation, except if they want to provide an alternative implementation (say, more efficient or numerically stable in some cases). Another immediate advantage of libraries that build explicit computational graphs is that they can take advantage of the other tools of *symbolic computation* (Buchberger *et al.*, 1983), such as simplification (to make computation faster and more memory-efficient) and transformations that make the computation more numerically stable (Bergstra *et al.*, 2010b). Such libraries also provide a *compiler* that converts computational graphs into executable code. Like most compilers, compilation to different execution environments (for example, CPU or GPU) can yield different executable code. This frees the user from specifying implementation details as part of the high-level computational expression.

6.4.5 Higher-order Derivatives

Some software frameworks support the use of higher-order derivatives. Among the deep learning software frameworks, this includes at least Theano and TensorFlow. These libraries use the same kind of data structure to describe the expressions for derivatives as they use to describe the original function being differentiated. This means that the symbolic differentiation machine can be applied to derivatives.

In the context of deep learning, it is rare to compute a single second derivative of a scalar function. Instead, we are usually interested in properties of the Hessian matrix. If we have a function $f : \mathbb{R}^n \to \mathbb{R}$, then the Hessian matrix is of size $n \times n$. In typical deep learning applications, n will be the number of parameters in the model, which could easily number in the billions. The entire Hessian matrix is thus infeasible to even represent.

Instead of explicitly computing the Hessian, the typical deep learning approach is to use *Krylov methods*. Krylov methods are a set of iterative techniques for performing various operations like approximately inverting a matrix or finding approximations to its eigenvectors or eigenvalues, without using any operation other than matrix-vector products.

In order to use Krylov methods on the Hessian, we only need to be able to compute the product between the Hessian matrix $\boldsymbol{H}$ and an arbitrary vector $\boldsymbol{v}$. A straightforward technique(Christianson, 1992) for doing so is to compute

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[\left(\nabla_{\boldsymbol{x}} f(x)\right)^{\top} \boldsymbol{v} \right] .$$

Both of the gradient computations in this expression may be computed automatically by the appropriate software library. Note that the outer gradient expression takes the gradient of a function of the inner gradient expression.

If $\boldsymbol{v}$ is itself a vector produced by a computational graph, it is important to specify that the automatic differentiation software should not differentiate through the graph that produced $\boldsymbol{v}$.

While computing the Hessian is usually not advisable, it is possible to do with Hessian vector products. One simply computes $\boldsymbol{H}\boldsymbol{e}^{(i)}$ for all $i = 1, \ldots, n$, where $\boldsymbol{e}^{(i)}$ is the one-hot vector with $e_i^{(i)} = 1$ and all other entries equal to 0.

6.5 Back-propagation through Random Operations and Graphical Models

Traditional neural networks implement a deterministic transformation of some input variables $\boldsymbol{x}$. We can extend neural networks to implement stochastic transformations of $\boldsymbol{x}$. One straightforward way to do this is to augment the neural network with randomly sampled inputs $\boldsymbol{z}$. The neural network can then continue to perform deterministic computation internally, but the function $f(\boldsymbol{x}, \boldsymbol{z})$ will appear stochastic to an observer who does not have access to $\boldsymbol{z}$. Provided that f is continuous and differentiable, we can then generally apply back-propagation as usual.

As an example, let us consider the operation consisting of drawing samples y from a Gaussian distribution with mean μ and variance σ^2:

$$\mathrm{y} \sim \mathcal{N}(\mu, \sigma^2).$$

Because an individual sample of y is not produced by a function, but rather by a sampling process whose output changes every time we query it, it may seem counterintuitive to take the derivatives of y with respect to the parameters of its distribution, μ and σ^2. However, we can rewrite the sampling process as transforming an underlying random value $\mathrm{z} \sim \mathcal{N}(0, 1)$ to obtain a sample from the desired distribution:

$$y = \mu + \sigma\eta \tag{6.11}$$

We are now able to back-propagate through the sampling operation, by regarding it as a deterministic operation with an extra input z. Crucially, the extra input is a random variable whose distribution is not a function of any of the variables whose derivatives we want to calculate. The result tells us how an infinitesimal change in μ or σ would change the output if we could repeat the sampling operation again with the same value of z.

Being able to back-propagate through this sampling operation allows us to incorporate it into a larger graph; e.g. we can compute the derivatives of some loss function $J(y)$. Moreover, we can introduce functions that shape the distribution, e.g. $\mu = f(\boldsymbol{x}; \boldsymbol{\theta})$ and $\sigma = g(\boldsymbol{x}; \boldsymbol{\theta})$ and use back-propagation through this functions to derive $\nabla_{\boldsymbol{\theta}} J(y)$.

The principle used in this Gaussian sampling example is true in general. We can express any probability distribution of the form $p(\mathrm{y}; \boldsymbol{\theta})$ or $p(\mathrm{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as $p(\mathrm{y} \mid \boldsymbol{\omega})$, where $\boldsymbol{\omega}$ is a variable containing both parameters $\boldsymbol{\theta}$, and if applicable, the inputs $\boldsymbol{x}$. Given a value y sampled from distribution $p(\mathrm{y} \mid \boldsymbol{\omega})$, where $\boldsymbol{\omega}$ may in turn be a function of other variables, , we can rewrite

$$\mathbf{y} \sim p(\mathbf{y} \mid \boldsymbol{\omega})$$

as

$$\boldsymbol{y} = f(\boldsymbol{z}, \boldsymbol{\omega}) \tag{6.12}$$

where $\boldsymbol{z}$ is a source of randomness. Crucially, $\boldsymbol{\omega}$ must not be a function of $\boldsymbol{z}$, and $\boldsymbol{z}$ must not be a function of $\boldsymbol{\omega}$. This is often called the *reparametrization trick*.

Gradient-based optimization can then be applied when f is continuous and differentiable almost everywhere. This of course requires $\boldsymbol{y}$ to be continuous. If we wish to back-propagate through a sampling process that produces discrete-valued samples, it may still be possible to estimate a gradient on $\boldsymbol{\omega}$, using reinforcement learning algorithms such as variants of the REINFORCE algorithm (Williams, 1992), discussed below.

In neural network applications, we typically choose $\boldsymbol{z}$ to be drawn from some simple distribution, such as a unit uniform or unit Gaussian distribution, and achieve more complex distributions by allowing the deterministic portion of the network to reshape its input. This is actually how the random generators for parametric distributions are implemented in software, by performing operations on approximately independent sources of noise (such as random bits).

6.5.1 Back-propagating through Discrete Stochastic Operations

The idea of propagating gradients or optimizing through stochastic operations is old (Price, 1958; Bonnet, 1964), first used for machine learning in the context of reinforcement learning (Williams, 1992), variational approximations (Opper and Archambeau, 2009), and more recently, stochastic or generative neural networks (Bengio *et al.*, 2013b; Kingma, 2013; Kingma and Welling, 2014b,a; Rezende *et al.*, 2014; Goodfellow *et al.*, 2014c). Many networks, such as denoising autoencoders or networks regularized with dropout, are also naturally designed to take noise as an input without requiring any special reparameterization to make the noise independent from the model.

When $\boldsymbol{y}$ is discrete, for a given sampled value of the injected noise $\boldsymbol{z}$, a small change of $\boldsymbol{\omega}$ would generally not change $\boldsymbol{z}$, and thus would not change our loss function J, so straightforward back-propagation is not applicable. Since the early days of research on propagating gradients through stochastic operations, that case was studied, mostly using reinforcement learning ideas. A simple but powerful set of solutions is the REINFORCE algorithm (Williams, 1992). The idea is that J generally becomes a continuous function of $\boldsymbol{\omega}$ when we average over the possible samples of the injected noise $\boldsymbol{z}$. Although that sum is typically not tractable when $\boldsymbol{y}$ is high-dimensional (or many discrete stochastic decisions are taken), it can be estimated unbiasedly using a Monte Carlo average, and that gives us a stochastic gradient which can be used with SGD or other stochastic gradient-based optimization techniques.

The simplest version of REINFORCE can be derived by simply differentiating the integral of interest:

$$\begin{aligned} E[J(\boldsymbol{y})] &= \sum_{\boldsymbol{y}} J(\boldsymbol{y})p(\boldsymbol{y}) \\ \frac{\partial E[J(\boldsymbol{y})]}{\partial \boldsymbol{\omega}} &= \sum_{z} J(\boldsymbol{y})\frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \\ &= \sum_{z} J(\boldsymbol{y})p(vy)\frac{\partial \log p(vy)}{\partial \boldsymbol{\omega}} \\ &\approx \frac{1}{n} \sum_{\mathbf{y}^{(i)} \sim p(\boldsymbol{y}),\, i=1}^{n} J(\boldsymbol{y}^{(i)}) \frac{\partial \log p(\boldsymbol{y}^{(i)})}{\partial \boldsymbol{\omega}} \end{aligned} \tag{6.13}$$

where the second line is true because we have assumed that $\boldsymbol{y}$ is discrete, which means that the derivative of f is zero, the third line exploits the derivative rule for the logarithm, $\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} = \frac{1}{p(\boldsymbol{y})} \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}$, and the last line gives as an unbiased Monte Carlo estimator of the gradient. For example, if $\boldsymbol{y}$ consists of a set of Bernoulli random variables z_i which are independent of each other given $\boldsymbol{\omega}$, with $p_i = p(y_i = 1 \mid \boldsymbol{\omega}) = \text{sigmoid}(\omega_i)$, then $\frac{\log p(\boldsymbol{y})}{\partial \omega_i}$ is the Bernoulli cross-entropy gradient, $z_i(1-p_i) + (1-z_i)p_i$. [9]

One issue with the above simple REINFORCE estimator is that it has a very high variance, so that many samples of $\boldsymbol{y}$ need to be drawn to obtain a good estimator of the gradient, or equivalently, if only one sample is drawn, SGD will converge very slowly and will require a smaller learning rate. It is possible to considerably reduce the variance of that estimator by using *variance reduction* methods (Wilson, 1984; L'Ecuyer, 1994). The idea is to modify the estimator so that its expected value remains unchanged but its varianced get reduced. In the context of REINFORCE, the proposed variance reduction methods involve the computation of a *baseline* that is used to offset $J(y)$. Note that any offset $b(\boldsymbol{\omega})$ that does not depend on y would not change the expectation of the estimated

[9]Note that $p(\boldsymbol{z})$ depends on $\boldsymbol{\omega}$ and $\boldsymbol{\omega}$ is typically encapsulates some input $\boldsymbol{x}$, so that $p(\boldsymbol{y})$ is different for each input $\boldsymbol{x}$. Everywhere we write $p(\boldsymbol{y})$ in this section, we might as well write $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$.

gradient because

$$\begin{aligned} E_{p(y)}\left[\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}}\right] &= \sum_y p(y)\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}} \\ &= \sum_y \frac{p(y)}{\partial \boldsymbol{\omega}} \\ &= \frac{\partial}{\partial \boldsymbol{\omega}}\sum_y p(y) = \frac{\partial}{\partial \boldsymbol{\omega}} 1 = 0, \end{aligned} \tag{6.14}$$

which means that

$$\begin{aligned} E_{p(y)}\left[(J(y) - b(\boldsymbol{\omega}))\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}}\right] &= E_{p(y)}\left[J(y)\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}}\right] - b(\boldsymbol{\omega})E_{p(y)}\left[\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}}\right] \\ &= E_{p(y)}\left[J(y)\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}}\right]. \end{aligned} \tag{6.15}$$

Furthermore, we can obtain the optimal $b(\boldsymbol{\omega})$ by computing the variance of $(J(y) - b(\boldsymbol{\omega}))\frac{\log p(y)}{\partial \boldsymbol{\omega}}$ under $p(y)$ and minimizing with respect to $b(\boldsymbol{\omega})$. What we find[10] is that this optimal baseline $b_i^*(\boldsymbol{\omega})$ is different for each element ω_i of the vector $\boldsymbol{\omega}$:

$$b_i^*(\boldsymbol{\omega}) = \frac{E_{p(y)}\left[J(y)\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}_i}^2\right]}{E_{p(y)}\left[\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}_i}^2\right]}. \tag{6.16}$$

The gradient estimator with respect to ω_i then becomes

$$(J(y) - b_i(\boldsymbol{\omega}))\frac{\partial \log p(y)}{\partial \omega_i}$$

where $b_i(\boldsymbol{\omega})$ estimates the above $b_i^*(\boldsymbol{\omega})$. This can be done by using additional outputs for the neural network that computes $\boldsymbol{\omega}$. In addition, it outputs an estimator of $E_{p(y)}[J(y)\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}_i}^2]$ and an estimator of $E_{p(y)}\left[\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}_i}^2\right]$ for each element of $\boldsymbol{\omega}$. These extra outputs can be trained with the mean squared error objective, using respectively $\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}_i}^2$ and $\frac{\partial \log p(y)}{\partial \boldsymbol{\omega}_i}^2$ as targets when y is sampled from $p(y)$, for a given $\boldsymbol{\omega}$. Mnih and Gregor (2014) preferred to use a single shared output (across all elements i of $\boldsymbol{\omega}$) trained with the target $J(y)$, using as baseline $b(\boldsymbol{\omega}) \approx E_{p(y)}[J(y)]$.

Variance reduction methods have been introduced in the reinforcement learning context (Sutton *et al.*, 2000; Weaver and Tao, 2001), generalizing previous work

[10] This is left as an exercise

on the case of binary reward by Dayan (1990). See Bengio *et al.* (2013b); Mnih and Gregor (2014); Ba *et al.* (2014); Mnih *et al.* (2014); Xu *et al.* (2015a) for examples of modern uses of the REINFORCE algorithm with reduced variance in the context of deep learning. In addition to the use of an input-dependent baseline $b(\boldsymbol{\omega})$, Mnih and Gregor (2014) found that the scale of $(J(y) - b(\boldsymbol{\omega}))$ could be adjusted during training by dividing it by its standard deviation estimated by a moving average during training, as a kind of adaptive learning rate, to counter the effect of important variations that occur during the course of training in the magnitude of this quantity. Mnih and Gregor (2014) called this heuristic *variance normalization*.

Once we have estimated the gradient of the expected loss with respect to $\boldsymbol{\omega}$, we can back-propagate it as usual in the upstream parts of the computational graph that lead to $\boldsymbol{\omega}$, in order to obtain an estimated gradient over the variables of interest (typically parameters of the model). However, REINFORCE-based estimators remain fairly noisy and can be understood as estimating the gradient by correlating choices of y with corresponding values of $J(y)$. If a good value of y is unlikely under the current parametrization, it might take a long time to obtain it by chance, and get the required signal that this configuration should be reinforced.

6.6 Universal Approximation Properties and Depth

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in a convex optimization problem when applied to linear models. Unfortunately, we often want to learn non-linear functions.

At first glance, we might presume that learning a non-linear function requires designing a specialized model family for the kind of non-linearity we want to learn. However, it turns out that feedforward networks with hidden layers provide a universal approximation framework. Specifically, the *universal approximation theorem* (Hornik *et al.*, 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et al.*, 1990). The concept of Borel measurability is beyond the scope of this book; for our purposes it suffices to say that any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space to another. Interestingly, universal approximation theorems have also been proven for a wider class of non-linearities which includes the now commonly used rectified linear unit (Leshno *et al.*, 1993).

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function. However, we are not guaranteed that the training algorithm will be able to *learn* that function. Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function due to overfitting. Recall from Chapter 5.3.1 that the "no free lunch" theorem shows that there is no universal machine learning algorithm. Feedforward networks provide a universal system for representing functions, in the sense that, given a function, we can find a feedforward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

Another but related problem facing our universal approximation scheme is the size of the model needed to represent a given function. The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but it does not say how large this network will be. Barron (1993) provides some bounds on the size of a single-layer network needed to approximate a broad class of functions. Unfortunately, in the worse case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $\boldsymbol{v} \in \{0, 1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

In summary, a feedforward network with a single layer is sufficient to represent any function, but it may be infeasibly large and may fail to learn and generalize correctly. Both of these failure modes suggest that we may want to use deeper models.

First, we may want to choose a model with more than one hidden layer in order to avoid needing to make the model infeasibly large. There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value d, but require a much larger model if depth is restricted to be less than or equal to d. In many cases, the number of hidden units required by the shallow model is exponential in n. Such results have been first proven for circuits of logic gates (Håstad, 1986), then for linear threshold units with non-negative weights (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), which are more like the actual neural networks we care about. This result was extended by Maass (1992); Maass *et al.* (1994), showing advantages of using continuous-valued activations over binary ones. Results about the existence of functions that are difficult to represent with a shallow polynomial circuit (Delalleau and Bengio, 2011) were also shown, which may be interesting because of the proposal to use such circuits to represent distributions, called sum-product networks or SPNs (Poon and Domingos, 2011). Note that in order to obtain tractable computation in the probabilistic interpretation of SPNs, constraints are introduced (the decomposability and completeness conditions) that may limit their expressive power (Martens and Medabalimi, 2014). This last result also proves the existence of a depth hierarchy for SPNs, discriminating between all finite depths, whereas (Delalleau and Bengio, 2011) only discriminated between a shallow and sufficiently deep circuit. More recently, following on results demonstrating the universal approximation properties of shallow networks of rectifier units (Leshno *et al.*, 1993), results were obtained about the expressive power of deep networks of rectifier units (Pascanu *et al.*, 2013b; Montufar *et al.*, 2014). These showed that functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network.

Of course, there is no guarantee that the kinds of functions we want to learn in applications of machine learning (and in particular for AI) share such a property.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation, but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks (Bengio *et al.*, 2007b; Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Goodfellow *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012b; Sermanet *et al.*, 2013; Farabet *et al.*, 2013a; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a). See Fig. 6.9 and Fig. 6.10 for examples of some of these empirical results. This suggests that using deep architectures does indeed express a useful prior over the space of functions the model learns.

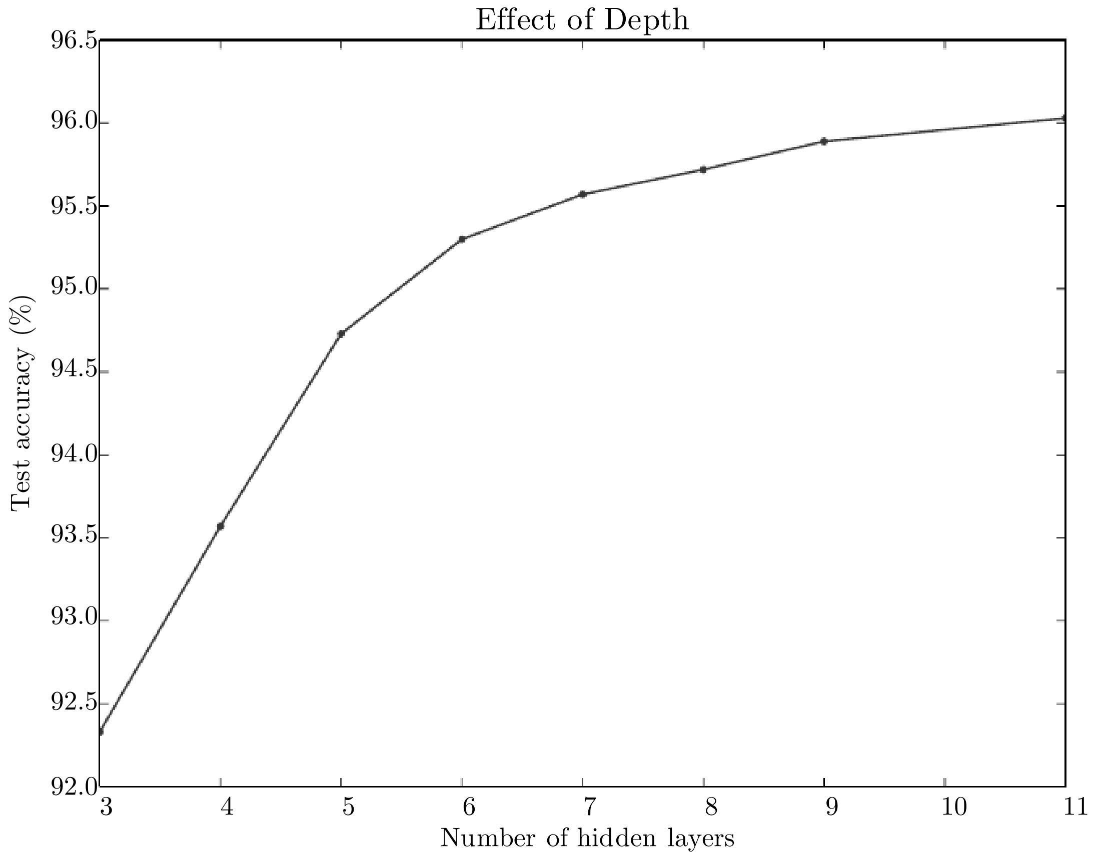

Figure 6.9: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See Fig. 6.10 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

6.7 Feature / Representation Learning

Let us consider again the single layer networks such as the perceptron, linear regression and logistic regression: such linear models are appealing because training them involves a convex optimization problem[11]. Convex optimization comes with some convergence guarantees towards a global optimum, irrespective of initial conditions. Simple and well-understood optimization algorithms are available in this case. However, this limits the representational capacity too much: many

[11] In some cases, shallow linear models can even be fit in closed form. Linear regression and some Gaussian process regression models have this property.

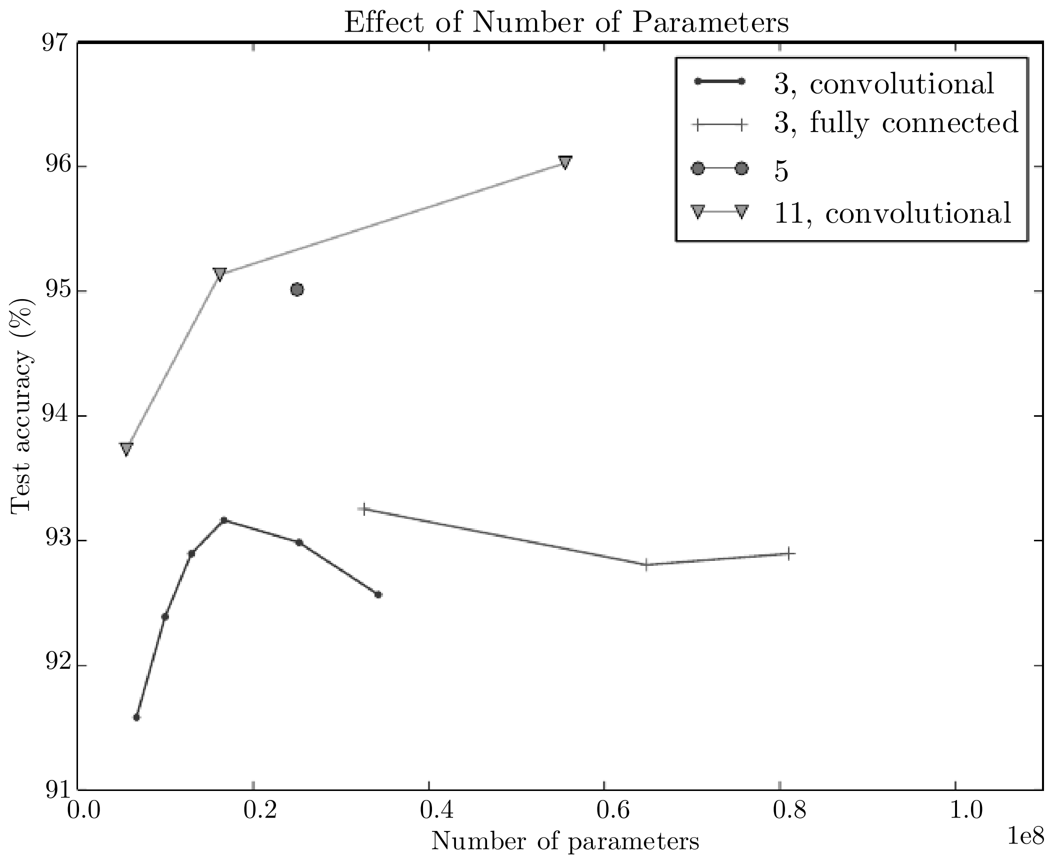

Figure 6.10: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 2 million parameters while deep ones can benefit from having over 6 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

tasks, for a given choice of input representation $\boldsymbol{x}$ (the raw input features), cannot be solved by using only a linear predictor. What are our options to avoid that limitation?

1. One option is to use a kernel machine as described in Sec. 5.8.2. Kernel machines achieve non-linearity by using a fixed non-linear mapping from $\boldsymbol{x}$ to $\phi(\boldsymbol{x})$, where $\phi(\boldsymbol{x})$ is typically of much higher dimension. If $\phi(\boldsymbol{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization is not at all guaranteed: it will depend on the appropriateness of the choice of ϕ as a feature space for our task. Kernel machine theory clearly identifies the choice of ϕ to the choice of a prior. This leads to kernel engineering, which is equivalent to feature engineering, discussed next. When the kernel is not explicitly engineered for a specific task, the kernel is chosen to impose a very broad prior, such as smoothness of the decision function. The most commonly used example is the Gaussian (or RBF) kernel $k(\boldsymbol{u}, \boldsymbol{v}) = \exp\left(-||\boldsymbol{u} - \boldsymbol{v}||/\sigma^2\right)$. Unfortunately, this prior may be insufficient to regularize the classifier sufficiently to overcome the curse of dimensionality, as introduced in Sec. 5.12.1 and developed in more detail in Chapter 14.

2. Another option is to *manually engineer the representation or features* $\phi(\boldsymbol{x})$. Most industrial applications of machine learning rely on hand-crafted features and most of the research and development effort (as well as a very large fraction of the scientific literature in machine learning and its applications) goes into designing new features that are most appropriate to the task at hand. Clearly, faced with a problem to solve and some prior knowledge in the form of representations that are believed to be relevant, the prior knowledge can be very useful. This approach is therefore common in practice, but is not completely satisfying because it involves a very task-specific engineering work and a laborious never-ending effort to improve systems by designing better features. If there were some more general feature learning approaches that could be applied to a large set of related tasks (such as those involved in AI), we would certainly like to take advantage of them. Since humans seem to be able to learn a lot of new tasks (for which they were not programmed by evolution), it seems that such broad priors do exist. This whole question is discussed in more detail in Bengio and LeCun (2007a), and motivates the third option.

3. The third option is to *learn the features*, or *learn the representation*. In a sense, it allows one to interpolate between the almost agnostic approach of a kernel machine with a general-purpose smoothness kernel (such as RBF SVMs and other non-parametric statistical models) and full designer-provided

knowledge in the form of a fixed representation that is perfectly tailored to the task. This is equivalent to the idea of *learning the kernel*, except that whereas most kernel learning methods only allow very few degrees of freedom in the learned kernel, representation learning methods such as those discussed in this book (including multi-layer neural networks) allow the feature function $\phi(\cdot)$ to be very rich (with a number of parameters that can be in the millions or more, depending on the amount of data available). This is equivalent to *learning the hidden layers*, in the case of a multi-layer neural network. Besides smoothness (which comes for example from regularizers such as weight decay), other priors can be incorporated in this feature learning. The most celebrated of these priors is *depth*, discussed above (Sec. 6.6). Other priors are discussed in Chapter 14.

This whole discussion is clearly not specific to neural networks and supervised learning, and is one of the central motivations for this book.

6.8 Piecewise Linear Hidden Units

Most of the recent improvement in the performance of deep neural networks can be attributed to increases in computational power and the size of datasets. The machine learning algorithms involved in recent state-of-the-art systems have mostly existed since the 1980s, with a few recent conceptual advances contributing significantly to increased performance.

One of the main algorithmic improvements that has had a significant impact is the use of piecewise linear units, such as absolute value rectifiers and rectified linear units. Such units consist of two linear pieces and their behavior is driven by a single weight vector. Jarrett *et al.* (2009b) observed that "using a rectifying non-linearity is the single most important factor in improving the performance of a recognition system" among several different factors of neural network architecture design.

For small datasets, Jarrett *et al.* (2009b) observed that using rectifying non-linearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, allowing the classifier layer at the top to learn how to map different feature vectors to class identities.

When more data is available, learning begins to extract enough useful knowledge to exceed the performance of randomly chosen parameters. Glorot *et al.* (2011b) showed that learning is far easier in deep rectified linear networks than in deep networks that have curvature or two-sided saturation in their activation functions. Because the behavior of the unit is linear over half of its domain, it is easy for an optimization algorithm to tell how to improve the behavior of a unit, even when the unit's activations are far from optimal. Just as piecewise linear networks are good at propagating information forward, back-propagation in such a network is also piecewise linear and propagates information about the error derivatives to all of the gradients in the network. Each piecewise linear function can be decomposed into different regions corresponding to different linear pieces. When we change a parameter of the network, the resulting change in the network's activity is linear until the point that it causes some unit to go from one linear piece to another. Traditional units such as sigmoids are more prone to discarding information due to saturation both in forward propagation and in back-propagation. The response of such a network to a change in a single parameter may be highly nonlinear even in a small neighborhood.

Glorot *et al.* (2011b) motivate rectified linear units from biological considerations. The half-rectifying non-linearity was intended to capture these properties of biological neurons: 1) For some inputs, biological neurons are completely inactive. 2) For some inputs, a biological neuron's output is proportional to its input. 3) Most of the time, biological neurons operate in the regime where they are inactive (e.g., they should have *sparse activations)*.

One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. This problem can be mitigated by initializing the biases to a small positive number, but it is still possible for a rectified linear unit to learn to de-activate and then never be activated again. Goodfellow *et al.* (2013a) introduced maxout units and showed that maxout units can successfully learn in conditions where rectified linear units become stuck. Maxout units are also piecewise linear, but unlike rectified linear units, each piece of the linear function has its own weight vector, so whichever piece is active can always learn. Due to the greater number of weight vectors, maxout units typically need extra regularization such as dropout, though they can work satisfactorily if the training set is large and the number of pieces per unit is kept low (Cai *et al.*, 2013). Maxout units have a few other benefits. In some cases, one can gain some statistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights. Because each unit is driven by multiple filters, maxout units have some redundancy that helps them to resist forgetting how to perform tasks that they were trained on in the past. Neural networks trained with stochastic gradient descent are generally believed to suffer from a phenomenon called *catastrophic forgetting* but maxout units tend to exhibit only mild forgetting (Goodfellow *et al.*, 2014a). Maxout units can also be seen as *learning the activation function* itself rather than just the relationship between units. With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, maxout with two pieces can learn to implement the rectified linear activation function or the absolute value rectification function.

Another way to avoid the zero-gradient problem of rectifiers is with the leaky (Maas *et al.*, 2013) or or parametric ReLU (He *et al.*, 2015), introduced above. By having a small slope rather than a zero slope when the argument of the rectifier is negative, gradients pass all the time. PReLUs helped to yield a top-5 test error (4.94%) lower than humans (5.1%) for the first time on the ImageNet benchmark (He *et al.*, 2015).

This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation–a particular straightforward kind of such linear activation. This is discussed further in Sec. 10.8.4.

In addition to helping to propagate information and making optimization easier, piecewise linear units also have some nice properties that can make them easier to regularize. This is discussed further in Sec. 7.11.

Sigmoidal non-linearities still perform well in some contexts and are a popular choice when a hidden unit must compute a number guaranteed to be in a bounded interval (like in the (0,1) interval), but piecewise linear units are now by far the most popular kind of hidden units.

6.9 Historical Notes

Sec. 1.2 already gave an overview of the history of neural networks and deep learning. Here we focus on historical notes regarding back-propagation and the connectionist ideas that are still at the heart of today's research in deep learning.

The chain rule was invented in the 17th century (Leibniz, 1676; L'Hôpital, 1696) and gradient descent in the 19th centry (Cauchy, 1847). Efficient applications of the chain rule which exploit the dynamic programming structure described in this chapter are found already in the 1960's and 1970's, mostly for control applications (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Dreyfus, 1973) but also for sensitivity analysis (Linnainmaa, 1976). Bringing these ideas to the optimization of weights of artificial neural networks with continuous-valued outputs was introduced by Werbos (1981) and rediscovered independently in different ways as well as actually simulated successfully by LeCun (1985); Parker (1985); Rumelhart *et al.* (1986a). The book *Parallel Distributed Processing* (Rumelhart *et al.*, 1986d) presented these findings in a chapter (Rumelhart *et al.*, 1986b) that contributed greatly to the popularization of back-propagation and initiated a very active period of research in multi-layer neural networks. However, the ideas put forward by the authors of that book and in particular by Rumelhart and Hinton go much beyond back-propagation. They include crucial ideas about the possible computational implementation of several central aspects of cognition and learning, which came under the name of "connectionism" because of the importance given the connections between neurons as the locus of learning and memory. In particular, these ideas include the notion of distributed representation, introduced in Chapter 1 and developed a lot more in part III of this book, with Chapter 14, which is at the heart of the generalization ability of neural networks. As discussed with the historical survey in Sec. 1.2, the boom of AI and machine learning research which followed on the connectionist ideas reached a peak in the early 1990's, as far as neural networks are concerned, while other machine learning techniques become more popular in the late 1990's and remained so for the first decade of this century. Neural networks research in the AI and machine learning community almost vanished then, only to be reborn ten years later (starting in 2006) with a novel focus on the depth of representation and the current wave of research on deep learning. In addition to back-propagation and distributed representations, the connectionists brought the idea of iterative inference (they used different words), viewing neural computation in the brain as a way to look for a configuration of neurons that best satisfy all the relevant pieces of knowledge implicitly captured in the weights of the neural network. This view turns out to be central in the topics covered in part III of this book regarding probabilistic models and inference.