the *convolutional neural network architecture* (cf. Chapter 8), in which the architecture is carefully designed in order to conform to the typical properties of image data. Such an approach minimizes the risk of *overfitting* by incorporating domain-specific insights (or *bias*). As we will discuss later in this book (cf. Chapter 4), overfitting is a pervasive problem in neural network design, so that the network often performs very well on the training data, but it *generalizes* poorly to unseen test data. This problem occurs when the number of free parameters, (which is typically equal to the number of weight connections), is too large compared to the size of the training data. In such cases, the large number of parameters memorize the specific nuances of the training data, but fail to recognize the statistically significant patterns for classifying unseen test data. Clearly, increasing the number of nodes in the neural network tends to encourage overfitting. Much recent work has been focused both on the architecture of the neural network as well as on the computations performed within each node in order to minimize overfitting. Furthermore, the way in which the neural network is trained also has an impact on the quality of the final solution. Many clever methods, such as *pretraining* (cf. Chapter 4), have been proposed in recent years in order to improve the quality of the learned solution. This book will explore these advanced training methods in detail.

1.2.3 The Multilayer Network as a Computational Graph

It is helpful to view a neural network as a *computational graph*, which is constructed by piecing together many basic parametric models. Neural networks are fundamentally more powerful than their building blocks because the parameters of these models are learned *jointly* to create a highly optimized composition function of these models. The common use of the term "perceptron" to refer to the basic unit of a neural network is somewhat misleading, because there are many variations of this basic unit that are leveraged in different settings. In fact, it is far more common to use logistic units (with sigmoid activation) and piecewise/fully linear units as building blocks of these models.

A multilayer network evaluates compositions of functions computed at individual nodes. A path of length 2 in the neural network in which the function $f(\cdot)$ follows $g(\cdot)$ can be considered a composition function $f(g(\cdot))$. Furthermore, if $g_1(\cdot)$, $g_2(\cdot) \ldots g_k(\cdot)$ are the functions computed in layer m, and a particular layer-$(m+1)$ node computes $f(\cdot)$, then the composition function computed by the layer-$(m+1)$ node in terms of the layer-m inputs is $f(g_1(\cdot), \ldots g_k(\cdot))$. The use of nonlinear activation functions is the key to increasing the power of multiple layers. If all layers use an identity activation function, then a multilayer network can be shown to simplify to linear regression. It has been shown [208] that a network with a single hidden layer of nonlinear units (with a wide ranging choice of squashing functions like the sigmoid unit) and a single (linear) output layer can compute almost any "reasonable" function. As a result, neural networks are often referred to as *universal function approximators*, although this theoretical claim is not always easy to translate into practical usefulness. The main issue is that the number of hidden units required to do so is rather large, which increases the number of parameters to be learned. This results in practical problems in training the network with a limited amount of data. In fact, deeper networks are often preferred because they reduce the number of hidden units in each layer as well as the overall number of parameters.

The "building block" description is particularly appropriate for multilayer neural networks. Very often, off-the-shelf softwares for building neural networks[2] provide analysts

[2]Examples include Torch [572], Theano [573], and TensorFlow [574].

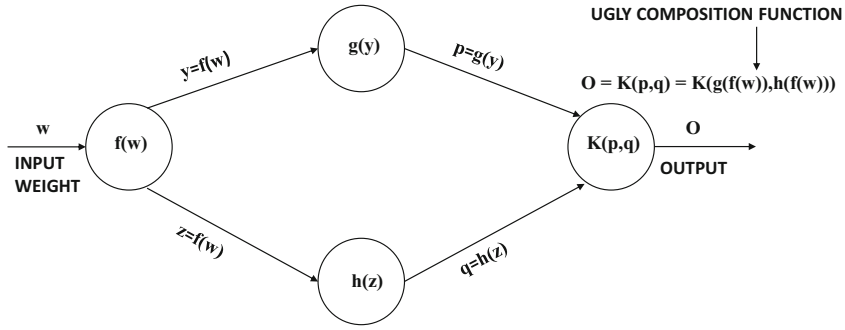
with access to these building blocks. The analyst is able to specify the number and type of units in each layer along with an off-the-shelf or customized loss function. A deep neural network containing tens of layers can often be described in a few hundred lines of code. All the learning of the weights is done automatically by the *backpropagation algorithm* that uses dynamic programming to work out the complicated parameter update steps of the underlying computational graph. The analyst does not have to spend the time and effort to explicitly work out these steps. This makes the process of trying different types of architectures relatively painless for the analyst. Building a neural network with many of the off-the-shelf softwares is often compared to a child constructing a toy from building blocks that appropriately fit with one another. Each block is like a unit (or a layer of units) with a particular type of activation. Much of this ease in training neural networks is attributable to the backpropagation algorithm, which shields the analyst from explicitly working out the parameter update steps of what is actually an extremely complicated optimization problem. Working out these steps is often the most difficult part of most machine learning algorithms, and an important contribution of the neural network paradigm is to bring modular thinking into machine learning. In other words, the modularity in neural network design translates to modularity in learning its parameters; the specific name for the latter type of modularity is "backpropagation." This makes the design of neural networks more of an (experienced) engineer's task rather than a mathematical exercise.

1.3 Training a Neural Network with Backpropagation

In the single-layer neural network, the training process is relatively straightforward because the error (or loss function) can be computed as a direct function of the weights, which allows easy gradient computation. In the case of multi-layer networks, the problem is that the loss is a complicated composition function of the weights in earlier layers. The gradient of a composition function is computed using the backpropagation algorithm. The backpropagation algorithm leverages the chain rule of differential calculus, which computes the error gradients in terms of summations of local-gradient products over the various paths from a node to the output. Although this summation has an exponential number of components (paths), one can compute it efficiently using *dynamic programming*. The backpropagation algorithm is a direct application of dynamic programming. It contains two main phases, referred to as the *forward* and *backward* phases, respectively. The forward phase is required to compute the output values and the local derivatives at various nodes, and the backward phase is required to accumulate the products of these local values over all paths from the node to the output:

1. *Forward phase:* In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The final predicted output can be compared to that of the training instance and the derivative of the loss function with respect to the output is computed. The derivative of this loss now needs to be computed with respect to the weights in all layers in the backwards phase.

2. *Backward phase:* The main goal of the backward phase is to learn the gradient of the loss function with respect to the different weights by using the chain rule of differential calculus. These gradients are used to update the weights. Since these gradients are learned in the backward direction, starting from the output node, this learning process is referred to as the backward phase. Consider a sequence of hidden units



$$\frac{\partial o}{\partial w} = \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial w} \quad \text{[Multivariable Chain Rule]}$$
$$= \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial w} \quad \text{[Univariate Chain Rule]}$$
$$= \underbrace{\frac{\partial K(p,q)}{\partial p} \cdot g'(y) \cdot f'(w)}_{\text{First path}} + \underbrace{\frac{\partial K(p,q)}{\partial q} \cdot h'(z) \cdot f'(w)}_{\text{Second path}}$$

Figure 1.13: **Illustration of chain rule in computational graphs:** The products of node-specific partial derivatives along paths from weight w to output o are aggregated. The resulting value yields the derivative of output o with respect to weight w. Only two paths between input and output exist in this simplified example.

$h_1, h_2, \ldots, h_k$ followed by output o, with respect to which the loss function L is computed. Furthermore, assume that the weight of the connection from hidden unit h_r to h_{r+1} is $w_{(h_r,h_{r+1})}$. Then, in the case that a single path exists from h_1 to o, one can derive the gradient of the loss function with respect to any of these edge weights using the chain rule:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \quad \forall r \in 1 \ldots k \tag{1.22}$$

The aforementioned expression assumes that only a *single path* from h_1 to o exists in the network, whereas an exponential number of paths might exist in reality. A generalized variant of the chain rule, referred to as the *multivariable chain rule*, computes the gradient in a computational graph, where more than one path might exist. This is achieved by adding the composition along each of the paths from h_1 to o. An example of the chain rule in a computational graph with two paths is shown in Figure 1.13. Therefore, one generalizes the above expression to the case where a set $\mathcal{P}$ of paths exist from h_r to o:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r,h_{r+1},\ldots h_k,o] \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\text{Backpropagation computes } \Delta(h_r,o) = \frac{\partial L}{\partial h_r}} \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \tag{1.23}$$

The computation of $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ on the right-hand side is straightforward and will be discussed below (cf. Equation 1.27). However, the path-aggregated term above [annotated by $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$] is aggregated over an exponentially increasing number of paths (with respect to path length), which seems to be intractable at first sight. A key point is that the computational graph of a neural network does not have cycles, and it is possible to compute such an aggregation in a principled way in the backwards direction by first computing $\Delta(h_k, o)$ for nodes h_k closest to o, and then recursively computing these values for nodes in earlier layers in terms of the nodes in later layers. Furthermore, the value of $\Delta(o, o)$ for each output node is initialized as follows:

$$\Delta(o, o) = \frac{\partial L}{\partial o} \tag{1.24}$$

This type of dynamic programming technique is used frequently to efficiently compute all types of path-centric functions in directed acyclic graphs, which would otherwise require an exponential number of operations. The recursion for $\Delta(h_r, o)$ can be derived using the multivariable chain rule:

$$\Delta(h_r, o) = \frac{\partial L}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \Delta(h, o) \tag{1.25}$$

Since each h is in a later layer than h_r, $\Delta(h, o)$ has already been computed while evaluating $\Delta(h_r, o)$. However, we still need to evaluate $\frac{\partial h}{\partial h_r}$ in order to compute Equation 1.25. Consider a situation in which the edge joining h_r to h has weight $w_{(h_r,h)}$, and let a_h be the value computed in hidden unit h just *before* applying the activation function $\Phi(\cdot)$. In other words, we have $h = \Phi(a_h)$, where a_h is a linear combination of its inputs from earlier-layer units incident on h. Then, by the univariate chain rule, the following expression for $\frac{\partial h}{\partial h_r}$ can be derived:

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \frac{\partial \Phi(a_h)}{\partial a_h} \cdot w_{(h_r,h)} = \Phi'(a_h) \cdot w_{(h_r,h)}$$

This value of $\frac{\partial h}{\partial h_r}$ is used in Equation 1.25, which is repeated recursively in the backwards direction, starting with the output node. The corresponding updates in the backwards direction are as follows:

$$\Delta(h_r, o) = \sum_{h:h_r \Rightarrow h} \Phi'(a_h) \cdot w_{(h_r,h)} \cdot \Delta(h, o) \tag{1.26}$$

Therefore, gradients are successively accumulated in the backwards direction, and each node is processed exactly once in a backwards pass. Note that the computation of Equation 1.25 (which requires proportional operations to the number of outgoing edges) needs to be repeated for each incoming edge into the node to compute the gradient with respect to all edge weights. Finally, Equation 1.23 requires the computation of $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$, which is easily computed as follows:

$$\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} = h_{r-1} \cdot \Phi'(a_{h_r}) \tag{1.27}$$

Here, the key gradient that is backpropagated is the derivative with respect to *layer activations*, and the gradient with respect to the weights is easy to compute for any incident edge on the corresponding unit.

It is noteworthy that the dynamic programming recursion of Equation 1.26 can be computed in multiple ways, depending on which variables one uses for intermediate chaining. All these recursions are equivalent in terms of the final result of backpropagation. In the following, we give an alternative version of the dynamic programming recursion, which is more commonly seen in textbooks. Note that Equation 1.23 uses the variables in the hidden layers as the "chain" variables for the dynamic programming recursion. One can also use the pre-activation values of the variables for the chain rule. The pre-activation variables in a neuron are obtained after applying the linear transform (but before applying the activation variables) as the intermediate variables. The pre-activation value of the hidden variable $h = \Phi(a_h)$ is a_h. The differences between the pre-activation and post-activation values within a neuron are shown in Figure 1.7. Therefore, instead of Equation 1.23, one can use the following chain rule:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \Phi'(a_o) \cdot \left[\sum_{[h_r,h_{r+1},\ldots h_k,o] \in \mathcal{P}} \frac{\partial a_o}{\partial a_{h_k}} \prod_{i=r}^{k-1} \frac{\partial a_{h_{i+1}}}{\partial a_{h_i}} \right]}_{\text{Backpropagation computes } \delta(h_r,o) = \frac{\partial L}{\partial a_{h_r}}} \underbrace{\frac{\partial a_{h_r}}{\partial w_{(h_{r-1},h_r)}}}_{h_{r-1}} \tag{1.28}$$

Here, we have introduced the notation $\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}}$ instead of $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$ for setting up the recursive equation. The value of $\delta(o, o) = \frac{\partial L}{\partial a_o}$ is initialized as follows:

$$\delta(o, o) = \frac{\partial L}{\partial a_o} = \Phi'(a_o) \cdot \frac{\partial L}{\partial o} \tag{1.29}$$

Then, one can use the multivariable chain rule to set up a similar recursion:

$$\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}} = \sum_{h:h_r \Rightarrow h} \overbrace{\frac{\partial L}{\partial a_h}}^{\delta(h,o)} \underbrace{\frac{\partial a_h}{\partial a_{h_r}}}_{\Phi'(a_{h_r}) w_{(h_r,h)}} = \Phi'(a_{h_r}) \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \cdot \delta(h, o) \tag{1.30}$$

This recursion condition is found more commonly in textbooks discussing backpropagation. The partial derivative of the loss with respect to the weight is then computed using $\delta(h_r, o)$ as follows:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \delta(h_r, o) \cdot h_{r-1} \tag{1.31}$$

As with the single-layer network, the process of updating the nodes is repeated to convergence by repeatedly cycling through the training data in epochs. A neural network may sometimes require thousands of epochs through the training data to learn the weights at the different nodes. A detailed description of the backpropagation algorithm and associated issues is provided in Chapter 3. In this chapter, we provide a brief discussion of these issues.

1.4 Practical Issues in Neural Network Training

In spite of the formidable reputation of neural networks as universal function approximators, considerable challenges remain with respect to actually training neural networks to provide this level of performance. These challenges are primarily related to several practical problems associated with training, the most important one of which is *overfitting*.