6.5 Back-Propagation and Other Differentiation Algorithms

When we use a feedforward neural network to accept an input $\boldsymbol{x}$ and produce an output $\hat{\boldsymbol{y}}$, information flows forward through the network. The inputs $\boldsymbol{x}$ provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$. This is called **forward propagation**. During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The **back-propagation** algorithm (Rumelhart *et al.*, 1986a), often simply called **backprop**, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined). Specifically, we will describe how to compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ is an additional set of variables that are inputs to the function but whose derivatives are not required. In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Many machine learning tasks involve computing other derivatives, either as part of the learning process, or to analyze the learned model. The back-propagation algorithm can be applied to these tasks as well, and is not restricted to computing the gradient of the cost function with respect to the parameters. The idea of computing derivatives by propagating information through a network is very general, and can be used to compute values such as the Jacobian of a function f with multiple outputs. We restrict our description here to the most commonly used case where f has a single output.

6.5.1 Computational Graphs

So far we have discussed neural networks with a relatively informal graph language. To describe the back-propagation algorithm more precisely, it is helpful to have a more precise **computational graph** language.

Many ways of formalizing computation as graphs are possible.

Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.

To formalize our graphs, we also need to introduce the idea of an **operation**. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together.

Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector. Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case in our description because it introduces many extra details that are not important to conceptual understanding.

If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y. We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Examples of computational graphs are shown in figure 6.8.

6.5.2 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$,

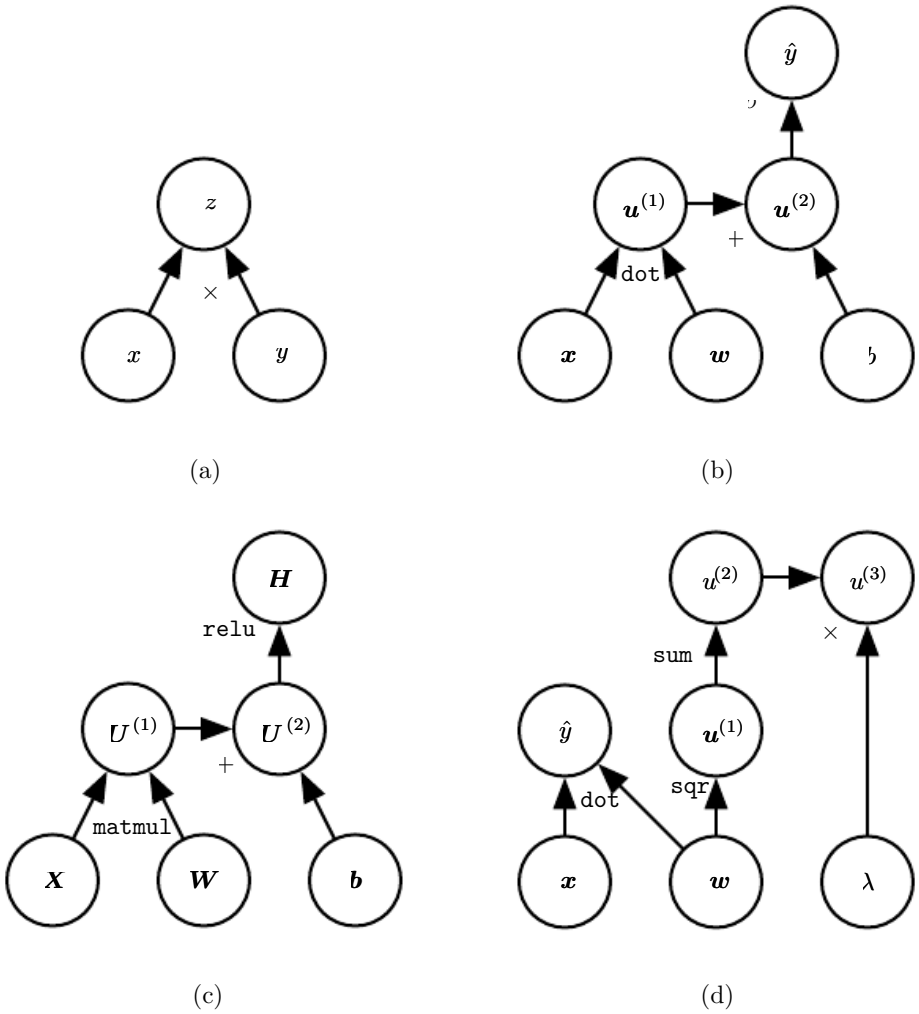

Figure 6.8: Examples of computational graphs. *(a)*The graph using the $\times$ operation to compute $z = xy$. *(b)*The graph for the logistic regression prediction $\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i-th such variable $\boldsymbol{u}^{(i)}$. *(c)*The computational graph for the expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$, which computes a design matrix of rectified linear unit activations $\boldsymbol{H}$ given a design matrix containing a minibatch of inputs $\boldsymbol{X}$. *(d)*Examples a–c applied at most one operation to each variable, but it is possible to apply more than one operation. Here we show a computation graph that applies more than one operation to the weights $\boldsymbol{w}$ of a linear regression model. The weights are used to make both the prediction $\hat{y}$ and the weight decay penalty $\lambda \sum_i w_i^2$.

g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Usually we do not apply the back-propagation algorithm merely to vectors, but rather to tensors of arbitrary dimensionality. Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor. We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

To denote the gradient of a value z with respect to a tensor $\mathbf{X}$, we write $\nabla_{\mathbf{X}} z$, just as if $\mathbf{X}$ were a vector. The indices into $\mathbf{X}$ now have multiple coordinates—for example, a 3-D tensor is indexed by three coordinates. We can abstract this away by using a single variable i to represent the complete tuple of indices. For all possible index tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla_{\boldsymbol{x}} z)_i$ gives $\frac{\partial z}{\partial x_i}$. Using this notation, we can write the chain rule as it applies to tensors. If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \tag{6.47}$$

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. However, actually evaluating that expression in a computer introduces some extra considerations.

Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient

will need to choose whether to store these subexpressions or to recompute them several times. An example of how these repeated subexpressions arise is given in figure 6.9. In some cases, computing the same subexpression twice would simply be wasteful. For complicated graphs, there can be exponentially many of these wasted computations, making a naive implementation of the chain rule infeasible. In other cases, computing the same subexpression twice could be a valid way to reduce memory consumption at the cost of higher runtime.

We first begin by a version of the back-propagation algorithm that specifies the actual gradient computation directly (algorithm 6.2 along with algorithm 6.1 for the associated forward computation), in the order it will actually be done and according to the recursive application of chain rule. One could either directly perform these computations or view the description of the algorithm as a symbolic specification of the computational graph for computing the back-propagation. However, this formulation does not make explicit the manipulation and the construction of the symbolic graph that performs the gradient computation. Such a formulation is presented below in section 6.5.6, with algorithm 6.5, where we also generalize to nodes that contain arbitrary tensors.

First consider a computational graph describing how to compute a single scalar $u^{(n)}$ (say the loss on a training example). This scalar is the quantity whose gradient we want to obtain, with respect to the n_i input nodes $u^{(1)}$ to $u^{(n_i)}$. In other words we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1, 2, \ldots, n_i\}$. In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model.

We will assume that the nodes of the graph have been ordered in such a way that we can compute their output one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$. As defined in algorithm 6.1, each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function

$$u^{(i)} = f(\mathbb{A}^{(i)}) \tag{6.48}$$

where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.

That algorithm specifies the forward propagation computation, which we could put in a graph $\mathcal{G}$. In order to perform back-propagation, we can construct a computational graph that depends on $\mathcal{G}$ and adds to it an extra set of nodes. These form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$. Computation in $\mathcal{B}$ proceeds in exactly the reverse of the order of computation in $\mathcal{G}$, and each node of $\mathcal{B}$ computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$. This is done

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

using the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{6.49}$$

as specified by algorithm 6.2. The subgraph $\mathcal{B}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$. The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. In addition, a dot product is performed for each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$. To summarize, the amount of computation required for performing the back-propagation scales linearly with the number of edges in $\mathcal{G}$, where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition. Below, we generalize this analysis to tensor-valued nodes, which is just a way to group multiple scalar values in the same node and enable more efficient implementations.

The back-propagation algorithm is designed to reduce the number of common subexpressions without regard to memory. Specifically, it performs on the order of one Jacobian product per node in the graph. This can be seen from the fact that backprop (algorithm 6.2) visits each edge from node $u^{(j)}$ to node $u^{(i)}$ of the graph exactly once in order to obtain the associated partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$.

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return $\{$`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i\}$

Back-propagation thus avoids the exponential explosion in repeated subexpressions. However, other algorithms may be able to avoid more subexpressions by performing simplifications on the computational graph, or may be able to conserve memory by recomputing rather than storing some subexpressions. We will revisit these ideas after describing the back-propagation algorithm itself.

6.5.4 Back-Propagation Computation in Fully-Connected MLP

To clarify the above definition of the back-propagation computation, let us consider the specific graph associated with a fully-connected multi-layer MLP.

Algorithm 6.3 first shows the forward propagation, which maps parameters to the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input,target) training example $(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$ the output of the neural network when $\boldsymbol{x}$ is provided in input.

Algorithm 6.4 then shows the corresponding computation to be done for

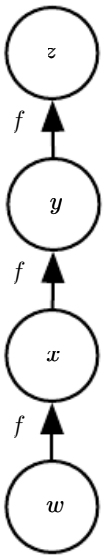

Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient. Let $w \in \mathbb{R}$ be the input to the graph. We use the same function $f : \mathbb{R} \to \mathbb{R}$ as the operation that we apply at every step of a chain: $x = f(w)$, $y = f(x)$, $z = f(y)$. To compute $\frac{\partial z}{\partial w}$, we apply equation 6.44 and obtain:

$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{6.53}$$

Equation 6.52 suggests an implementation in which we compute the value of $f(w)$ only once and store it in the variable x. This is the approach taken by the back-propagation algorithm. An alternative approach is suggested by equation 6.53, where the subexpression $f(w)$ appears more than once. In the alternative approach, $f(w)$ is recomputed each time it is needed. When the memory required to store the value of these expressions is low, the back-propagation approach of equation 6.52 is clearly preferable because of its reduced runtime. However, equation 6.53 is also a valid implementation of the chain rule, and is useful when memory is limited.

applying the back-propagation algorithm to this graph.

Algorithms 6.3 and 6.4 are demonstrations that are chosen to be simple and straightforward to understand. However, they are specialized to one specific problem.

Modern software implementations are based on the generalized form of back-propagation described in section 6.5.6 below, which can accommodate any computational graph by explicitly manipulating a data structure for representing symbolic computation.

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  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
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

6.5.5 Symbol-to-Symbol Derivatives

Algebraic expressions and computational graphs both operate on **symbols**, or variables that do not have specific values. These algebraic and graph-based representations are called **symbolic** representations. When we actually use or train a neural network, we must assign specific values to these symbols. We replace a symbolic input to the network $\boldsymbol{x}$ with a specific **numeric** value, such as $[1.2, 3.765, -1.8]^\top$.

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, 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.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

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):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

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

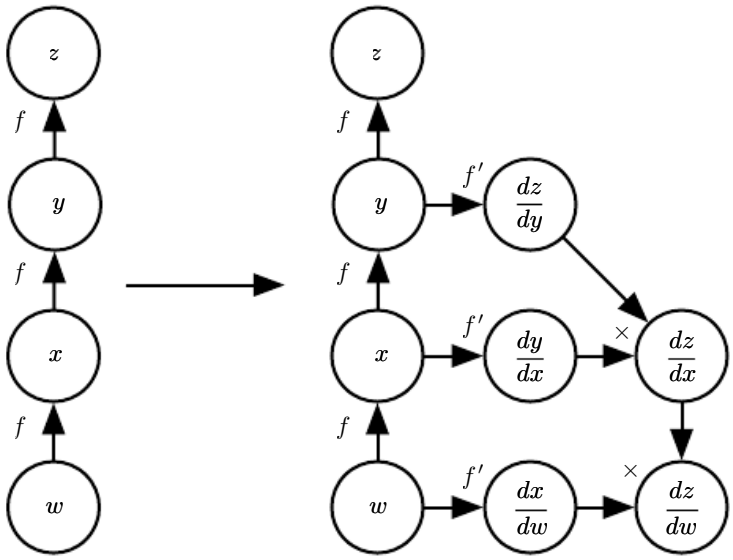

Figure 6.10: An example of the symbol-to-symbol approach to computing derivatives. In this approach, the back-propagation algorithm does not need to ever access any actual specific numeric values. Instead, it adds nodes to a computational graph describing how to compute these derivatives. A generic graph evaluation engine can later compute the derivatives for any specific numeric values. *(Left)*In this example, we begin with a graph representing $z = f(f(f(w)))$. *(Right)*We run the back-propagation algorithm, instructing it to construct the graph for the expression corresponding to $\frac{dz}{dw}$. In this example, we do not explain how the back-propagation algorithm works. The purpose is only to illustrate what the desired result is: a computational graph with a symbolic description of the derivative.

Some approaches to back-propagation take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values. We call this approach "symbol-to-number" differentiation. This is the approach used by libraries such as Torch (Collobert *et al.*, 2011b) and Caffe (Jia, 2013).

Another approach is to take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives. This is the approach taken by Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) and TensorFlow (Abadi *et al.*, 2015). An example of how this approach works is illustrated in figure 6.10. The primary advantage of this approach is that the derivatives are described in the same language as the original expression. Because the derivatives are just another computational graph, it is possible to run back-propagation again, differentiating the derivatives in order to obtain higher derivatives. Computation of higher-order derivatives is described in section 6.5.10.

We will use the latter approach and describe the back-propagation algorithm in

terms of constructing a computational graph for the derivatives. Any subset of the graph may then be evaluated using specific numerical values at a later time. This allows us to avoid specifying exactly when each operation should be computed. Instead, a generic graph evaluation engine can evaluate every node as soon as its parents' values are available.

The description of the symbol-to-symbol based approach subsumes the symbol-to-number approach. The symbol-to-number approach can be understood as performing exactly the same computations as are done in the graph built by the symbol-to-symbol approach. The key difference is that the symbol-to-number approach does not expose the graph.

6.5.6 General Back-Propagation

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach $\boldsymbol{x}$. For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

More formally, each node in the graph $\mathcal{G}$ corresponds to a variable. To achieve maximum generality, we describe this variable as being a tensor V. Tensor can in general have any number of dimensions. They subsume scalars, vectors, and matrices.

We assume that each variable V is associated with the following subroutines:

- `get_operation`(V): This returns the operation that computes V, represented by the edges coming into V in the computational graph. For example, there may be a Python or C++ class representing the matrix multiplication operation, and the `get_operation` function. Suppose we have a variable that is created by matrix multiplication, $\boldsymbol{C} = \boldsymbol{AB}$. Then `get_operation`($\mathsf{V}$) returns a pointer to an instance of the corresponding C++ class.

- `get_consumers`($\mathsf{V}, \mathcal{G}$): This returns the list of variables that are children of V in the computational graph $\mathcal{G}$.

- `get_inputs`($\mathsf{V}, \mathcal{G}$): This returns the list of variables that are parents of V in the computational graph $\mathcal{G}$.

Each operation `op` is also associated with a `bprop` operation. This `bprop` operation can compute a Jacobian-vector product as described by equation 6.47. This is how the back-propagation algorithm is able to achieve great generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that it participates in. For example, we might use a matrix multiplication operation to create a variable $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$. Suppose that the gradient of a scalar z with respect to $\boldsymbol{C}$ is given by $\boldsymbol{G}$. The matrix multiplication operation is responsible for defining two back-propagation rules, one for each of its input arguments. If we call the `bprop` method to request the gradient with respect to $\boldsymbol{A}$ given that the gradient on the output is $\boldsymbol{G}$, then the `bprop` method of the matrix multiplication operation must state that the gradient with respect to $\boldsymbol{A}$ is given by $\boldsymbol{G}\boldsymbol{B}^\top$. Likewise, if we call the `bprop` method to request the gradient with respect to $\boldsymbol{B}$, then the matrix operation is responsible for implementing the `bprop` method and specifying that the desired gradient is given by $\boldsymbol{A}^\top \boldsymbol{G}$. The back-propagation algorithm itself does not need to know any differentiation rules. It only needs to call each operation's `bprop` rules with the right arguments. Formally, $\texttt{op.bprop}(\texttt{inputs}, \mathsf{X}, \mathsf{G})$ must return

$$\sum_i \left(\nabla_{\mathsf{X}} \texttt{op.f}(\texttt{inputs})_i\right) \mathsf{G}_i, \tag{6.54}$$

which is just an implementation of the chain rule as expressed in equation 6.47. Here, `inputs` is a list of inputs that are supplied to the operation, `op.f` is the mathematical function that the operation implements, X is the input whose gradient we wish to compute, and G is the gradient on the output of the operation.

The `op.bprop` method should always pretend that all of its inputs are distinct from each other, even if they are not. For example, if the `mul` operator is passed two copies of x to compute x^2, the `op.bprop` method should still return x as the derivative with respect to both inputs. The back-propagation algorithm will later add both of these arguments together to obtain $2x$, which is the correct total derivative on x.

Software implementations of back-propagation usually provide both the operations and their `bprop` methods, so that users of deep learning software libraries are able to back-propagate through graphs built using common operations like matrix multiplication, exponents, logarithms, and so on. Software engineers who build a new implementation of back-propagation or advanced users who need to add their own operation to an existing library must usually derive the `op.bprop` method for any new operations manually.

The back-propagation algorithm is formally described in algorithm 6.5.

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$.
Initialize `grad_table`, a data structure associating tensors to their gradients
`grad_table`$[z] \leftarrow 1$
for $\mathbf{V}$ in $\mathbb{T}$ **do**
  `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}',$ `grad_table`$)$
end for
Return `grad_table` restricted to $\mathbb{T}$

In section 6.5.2, we explained that back-propagation was developed in order to avoid computing the same subexpression in the chain rule multiple times. The naive algorithm could have exponential runtime due to these repeated subexpressions. Now that we have specified the back-propagation algorithm, we can understand its computational cost. If we assume that each operation evaluation has roughly the same cost, then we may analyze the computational cost in terms of the number of operations executed. Keep in mind here that we refer to an operation as the fundamental unit of our computational graph, which might actually consist of very many arithmetic operations (for example, we might have a graph that treats matrix multiplication as a single operation). Computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations. Here we are counting operations in the computational graph, not individual operations executed by the underlying hardware, so it is important to remember that the runtime of each operation may be highly variable. For example, multiplying two matrices that each contain millions of entries might correspond to a single operation in the graph. We can see that computing the gradient requires as most $O(n^2)$ operations because the forward propagation stage will at worst execute all n nodes in the original graph (depending on which values we want to compute, we may not need to execute the entire graph). The back-propagation algorithm adds one Jacobian-vector product, which should be expressed with $O(1)$ nodes, per edge in the original graph. Because the computational graph is a directed acyclic graph it has at most $O(n^2)$ edges. For the kinds of graphs that are commonly used in practice, the situation is even better. Most neural network cost functions are

Algorithm 6.6 The inner loop subroutine `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$ of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: `grad_table`, a data structure mapping nodes to their gradients
  if $\mathbf{V}$ is in `grad_table` **then**
    Return `grad_table`$[\mathbf{V}]$
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in `get_consumers`$(\mathbf{V}, \mathcal{G}')$ **do**
    `op` $\leftarrow$ `get_operation`$(\mathbf{C})$
    $\mathbf{D} \leftarrow$ `build_grad`$(\mathbf{C}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
    $\mathbf{G}^{(i)} \leftarrow$ `op.bprop`(`get_inputs`$(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  `grad_table`$[\mathbf{V}] = \mathbf{G}$
  Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathbf{G}$

roughly chain-structured, causing back-propagation to have $O(n)$ cost. This is far better than the naive approach, which might need to execute exponentially many nodes. This potentially exponential cost can be seen by expanding and rewriting the recursive chain rule (equation 6.49) non-recursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path } (u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \tag{6.55}$$

Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with the depth of the forward propagation graph. This large cost would be incurred because the same computation for $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ would be redone many times. To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order,

back-propagation avoids repeating many common subexpressions. This table-filling strategy is sometimes called **dynamic programming**.

6.5.7 Example: Back-Propagation for MLP Training

As an example, we walk through the back-propagation algorithm as it is used to train a multilayer perceptron.

Here we develop a very simple multilayer perception with a single hidden layer. To train this model, we will use minibatch stochastic gradient descent. The back-propagation algorithm is used to compute the gradient of the cost on a single minibatch. Specifically, we use a minibatch of examples from the training set formatted as a design matrix $\boldsymbol{X}$ and a vector of associated class labels $\boldsymbol{y}$. The network computes a layer of hidden features $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$. To simplify the presentation we do not use biases in this model. We assume that our graph language includes a `relu` operation that can compute $\max\{0, \boldsymbol{Z}\}$ element-wise. The predictions of the unnormalized log probabilities over classes are then given by $\boldsymbol{H}\boldsymbol{W}^{(2)}$. We assume that our graph language includes a `cross_entropy` operation that computes the cross-entropy between the targets $\boldsymbol{y}$ and the probability distribution defined by these unnormalized log probabilities. The resulting cross-entropy defines the cost J_{MLE}. Minimizing this cross-entropy performs maximum likelihood estimation of the classifier. However, to make this example more realistic, we also include a regularization term. The total cost

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \tag{6.56}$$

consists of the cross-entropy and a weight decay term with coefficient λ. The computational graph is illustrated in figure 6.11.

The computational graph for the gradient of this example is large enough that it would be tedious to draw or to read. This demonstrates one of the benefits of the back-propagation algorithm, which is that it can automatically generate gradients that would be straightforward but tedious for a software engineer to derive manually.

We can roughly trace out the behavior of the back-propagation algorithm by looking at the forward propagation graph in figure 6.11. To train, we wish to compute both $\nabla_{\boldsymbol{W}^{(1)}} J$ and $\nabla_{\boldsymbol{W}^{(2)}} J$. There are two different paths leading backward from J to the weights: one through the cross-entropy cost, and one through the weight decay cost. The weight decay cost is relatively simple; it will always contribute $2\lambda \boldsymbol{W}^{(i)}$ to the gradient on $\boldsymbol{W}^{(i)}$.

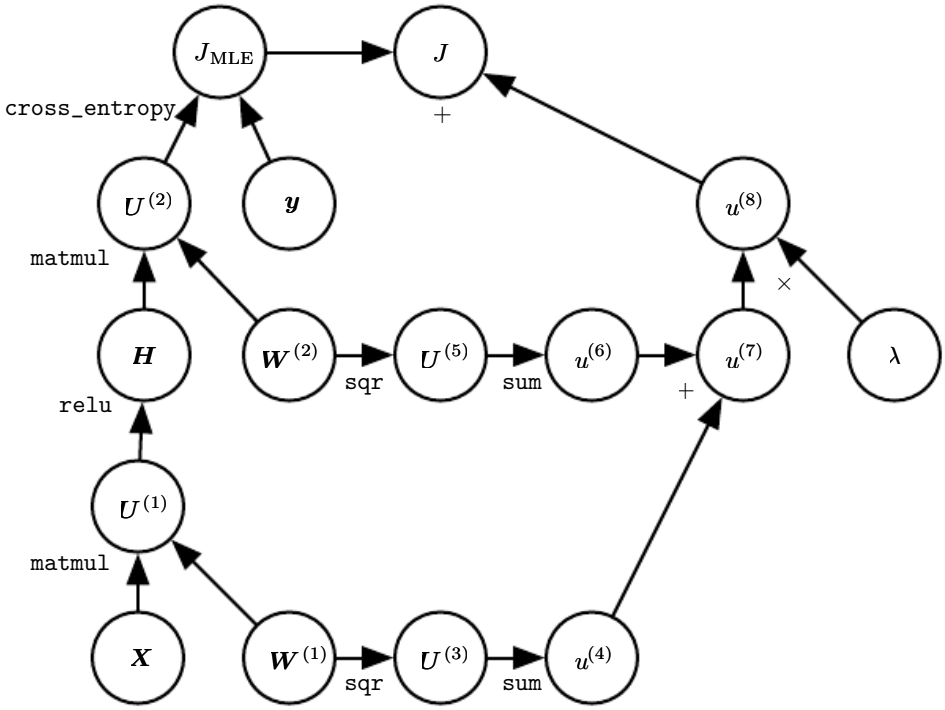

Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

The other path through the cross-entropy cost is slightly more complicated. Let $\boldsymbol{G}$ be the gradient on the unnormalized log probabilities $\boldsymbol{U}^{(2)}$ provided by the `cross_entropy` operation. The back-propagation algorithm now needs to explore two different branches. On the shorter branch, it adds $\boldsymbol{H}^\top \boldsymbol{G}$ to the gradient on $\boldsymbol{W}^{(2)}$, using the back-propagation rule for the second argument to the matrix multiplication operation. The other branch corresponds to the longer chain descending further along the network. First, the back-propagation algorithm computes $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$ using the back-propagation rule for the first argument to the matrix multiplication operation. Next, the `relu` operation uses its back-propagation rule to zero out components of the gradient corresponding to entries of $\boldsymbol{U}^{(1)}$ that were less than 0. Let the result be called $\boldsymbol{G}'$. The last step of the back-propagation algorithm is to use the back-propagation rule for the second argument of the `matmul` operation to add $\boldsymbol{X}^\top \boldsymbol{G}'$ to the gradient on $\boldsymbol{W}^{(1)}$.

After these gradients have been computed, it is the responsibility of the gradient descent algorithm, or another optimization algorithm, to use these gradients to update the parameters.

For the MLP, the computational cost is dominated by the cost of matrix multiplication. During the forward propagation stage, we multiply by each weight

matrix, resulting in $O(w)$ multiply-adds, where w is the number of weights. During the backward propagation stage, we multiply by the transpose of each weight matrix, which has the same computational cost. The main memory cost of the algorithm is that we need to store the input to the nonlinearity of the hidden layer. This value is stored from the time it is computed until the backward pass has returned to the same point. The memory cost is thus $O(mn_h)$, where m is the number of examples in the minibatch and n_h is the number of hidden units.

6.5.8 Complications

Our description of the back-propagation algorithm here is simpler than the implementations actually used in practice.

As noted above, we have restricted the definition of an operation to be a function that returns a single tensor. Most software implementations need to support operations that can return more than one tensor. For example, if we wish to compute both the maximum value in a tensor and the index of that value, it is best to compute both in a single pass through memory, so it is most efficient to implement this procedure as a single operation with two outputs.

We have not described how to control the memory consumption of back-propagation. Back-propagation often involves summation of many tensors together. In the naive approach, each of these tensors would be computed separately, then all of them would be added in a second step. The naive approach has an overly high memory bottleneck that can be avoided by maintaining a single buffer and adding each value to that buffer as it is computed.

Real-world implementations of back-propagation also need to handle various data types, such as 32-bit floating point, 64-bit floating point, and integer values. The policy for handling each of these types takes special care to design.

Some operations have undefined gradients, and it is important to track these cases and determine whether the gradient requested by the user is undefined.

Various other technicalities make real-world differentiation more complicated. These technicalities are not insurmountable, and this chapter has described the key intellectual tools needed to compute derivatives, but it is important to be aware that many more subtleties exist.

6.5.9 Differentiation outside the Deep Learning Community

The deep learning community has been somewhat isolated from the broader computer science community and has largely developed its own cultural attitudes

concerning how to perform differentiation. More generally, the field of **automatic differentiation** is concerned with how to compute derivatives algorithmically. The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called **reverse mode accumulation**. Other approaches evaluate the subexpressions of the chain rule in different orders. In general, determining the order of evaluation that results in the lowest computational cost is a difficult problem. Finding the optimal sequence of operations to compute the gradient is NP-complete (Naumann, 2008), in the sense that it may require simplifying algebraic expressions into their least expensive form.

For example, suppose we have variables $p_1, p_2, \ldots, p_n$ representing probabilities and variables $z_1, z_2, \ldots, z_n$ representing unnormalized log probabilities. Suppose we define

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \tag{6.57}$$

where we build the softmax function out of exponentiation, summation and division operations, and construct a cross-entropy loss $J = -\sum_i p_i \log q_i$. A human mathematician can observe that the derivative of J with respect to z_i takes a very simple form: $q_i - p_i$. The back-propagation algorithm is not capable of simplifying the gradient this way, and will instead explicitly propagate gradients through all of the logarithm and exponentiation operations in the original graph. Some software libraries such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) are able to perform some kinds of algebraic substitution to improve over the graph proposed by the pure back-propagation algorithm.

When the forward graph $\mathcal{G}$ has a single output node and each partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ can be computed with a constant amount of computation, back-propagation guarantees that the number of computations for the gradient computation is of the same order as the number of computations for the forward computation: this can be seen in algorithm 6.2 because each local partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ needs to be computed only once along with an associated multiplication and addition for the recursive chain-rule formulation (equation 6.49). The overall computation is therefore $O(\#\text{ edges})$. However, it can potentially be reduced by simplifying the computational graph constructed by back-propagation, and this is an NP-complete task. Implementations such as Theano and TensorFlow use heuristics based on matching known simplification patterns in order to iteratively attempt to simplify the graph. We defined back-propagation only for the computation of a gradient of a scalar output but back-propagation can be extended to compute a Jacobian (either of k different scalar nodes in the graph, or of a tensor-valued node containing k values). A naive implementation may then need k times more computation: for

each scalar internal node in the original forward graph, the naive implementation computes k gradients instead of a single gradient. When the number of outputs of the graph is larger than the number of inputs, it is sometimes preferable to use another form of automatic differentiation called **forward mode accumulation**. Forward mode computation has been proposed for obtaining real-time computation of gradients in recurrent networks, for example (Williams and Zipser, 1989). This also avoids the need to store the values and gradients for the whole graph, trading off computational efficiency for memory. The relationship between forward mode and backward mode is analogous to the relationship between left-multiplying versus right-multiplying a sequence of matrices, such as

$$\boldsymbol{ABCD}, \tag{6.58}$$

where the matrices can be thought of as Jacobian matrices. For example, if $\boldsymbol{D}$ is a column vector while $\boldsymbol{A}$ has many rows, this corresponds to a graph with a single output and many inputs, and starting the multiplications from the end and going backwards only requires matrix-vector products. This corresponds to the backward mode. Instead, starting to multiply from the left would involve a series of matrix-matrix products, which makes the whole computation much more expensive. However, if $\boldsymbol{A}$ has fewer rows than $\boldsymbol{D}$ has columns, it is cheaper to run the multiplications left-to-right, corresponding to the forward mode.

In many communities outside of machine learning, it is more common to implement differentiation software that acts directly on traditional programming language code, such as Python or C code, and automatically generates programs that differentiate functions written in these languages. In the deep learning community, computational graphs are usually represented by explicit data structures created by specialized libraries. The specialized approach has the drawback of requiring the library developer to define the `bprop` methods for every operation and limiting the user of the library to only those operations that have been defined. However, the specialized approach also has the benefit of allowing customized back-propagation rules to be developed for each operation, allowing the developer to improve speed or stability in non-obvious ways that an automatic procedure would presumably be unable to replicate.

Back-propagation is therefore not the only way or the optimal way of computing the gradient, but it is a very practical method that continues to serve the deep learning community very well. In the future, differentiation technology for deep networks may improve as deep learning practitioners become more aware of advances in the broader field of automatic differentiation.

6.5.10 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 machinery 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

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[\left(\nabla_{\boldsymbol{x}} f(x) \right)^\top \boldsymbol{v} \right]. \tag{6.59}$$

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.

6.6 Historical Notes

Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation.

From this point of view, the modern feedforward network is the culmination of centuries of progress on the general function approximation task.

The chain rule that underlies the back-propagation algorithm was invented in the 17th century (Leibniz, 1676; L'Hôpital, 1696). Calculus and algebra have long been used to solve optimization problems in closed form, but gradient descent was not introduced as a technique for iteratively approximating the solution to optimization problems until the 19th century (Cauchy, 1847).

Beginning in the 1940s, these function approximation techniques were used to motivate machine learning models such as the perceptron. However, the earliest models were based on linear models. Critics including Marvin Minsky pointed out several of the flaws of the linear model family, such as its inability to learn the XOR function, which led to a backlash against the entire neural network approach.

Learning nonlinear functions required the development of a multilayer perceptron and a means of computing the gradient through such a model. Efficient applications of the chain rule based on dynamic programming began to appear in the 1960s and 1970s, 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). Werbos (1981) proposed applying these techniques to training artificial neural networks. The idea was finally developed in practice after being independently rediscovered in different ways (LeCun, 1985; Parker, 1985; Rumelhart *et al.*, 1986a). The book **Parallel Distributed Processing** presented the results of some of the first successful experiments with back-propagation 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 this school of thought places on the connections between neurons as the locus of learning and memory. In particular, these ideas include the notion of distributed representation (Hinton *et al.*, 1986).

Following the success of back-propagation, neural network research gained popularity and reached a peak in the early 1990s. Afterwards, other machine learning techniques became more popular until the modern deep learning renaissance that began in 2006.

The core ideas behind modern feedforward networks have not changed substantially since the 1980s. The same back-propagation algorithm and the same

approaches to gradient descent are still in use. Most of the improvement in neural network performance from 1986 to 2015 can be attributed to two factors. First, larger datasets have reduced the degree to which statistical generalization is a challenge for neural networks. Second, neural networks have become much larger, due to more powerful computers, and better software infrastructure. However, a small number of algorithmic changes have improved the performance of neural networks noticeably.

One of these algorithmic changes was the replacement of mean squared error with the cross-entropy family of loss functions. Mean squared error was popular in the 1980s and 1990s, but was gradually replaced by cross-entropy losses and the principle of maximum likelihood as ideas spread between the statistics community and the machine learning community. The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using the mean squared error loss.

The other major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piecewise linear hidden units, such as rectified linear units. Rectification using the $\max\{0, z\}$ function was introduced in early neural network models and dates back at least as far as the Cognitron and Neocognitron (Fukushima, 1975, 1980). These early models did not use rectified linear units, but instead applied rectification to nonlinear functions. Despite the early popularity of rectification, rectification was largely replaced by sigmoids in the 1980s, perhaps because sigmoids perform better when neural networks are very small. As of the early 2000s, rectified linear units were avoided due to a somewhat superstitious belief that activation functions with non-differentiable points must be avoided. This began to change in about 2009. Jarrett *et al.* (2009) observed that "using a rectifying nonlinearity 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.* (2009) observed that using rectifying nonlinearities 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.* (2011a) 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.

Rectified linear units are also of historical interest because they show that neuroscience has continued to have an influence on the development of deep learning algorithms. Glorot *et al.* (2011a) motivate rectified linear units from biological considerations. The half-rectifying nonlinearity 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 (i.e., they should have **sparse activations**).

When the modern resurgence of deep learning began in 2006, feedforward networks continued to have a bad reputation. From about 2006-2012, it was widely believed that feedforward networks would not perform well unless they were assisted by other models, such as probabilistic models. Today, it is now known that with the right resources and engineering practices, feedforward networks perform very well. Today, gradient-based learning in feedforward networks is used as a tool to develop probabilistic models, such as the variational autoencoder and generative adversarial networks, described in chapter 20. Rather than being viewed as an unreliable technology that must be supported by other techniques, gradient-based learning in feedforward networks has been viewed since 2012 as a powerful technology that may be applied to many other machine learning tasks. In 2006, the community used unsupervised learning to support supervised learning, and now, ironically, it is more common to use supervised learning to support unsupervised learning.

Feedforward networks continue to have unfulfilled potential. In the future, we expect they will be applied to many more tasks, and that advances in optimization algorithms and model design will improve their performance even further. This chapter has primarily described the neural network family of models. In the subsequent chapters, we turn to how to use these models—how to regularize and train them.