Chapter 11

Optimization in Computational Graphs

"Science is the differential calculus of the mind. Art the integral calculus; they may be beautiful when apart, but are greatest only when combined."– Ronald Ross

11.1 Introduction

A computational graph is a network of connected nodes, in which each node is a unit of computation and stores a variable. Each edge joining two nodes indicates a relationship between the corresponding variables. The graph may be either directed or undirected. In a directed graph, a node computes its associated variable as a function of the variables in the nodes that have edges incoming to it. In an undirected graph, the functional relationship works in both directions. Most practical computational graphs (e.g., conventional neural networks) are *directed acyclic graphs*, although many undirected probabilistic models in machine learning can be implicitly considered computational graphs with cycles. Similarly, the variables at the nodes might be continuous, discrete, or probabilistic, although most real-world computational graphs work with continuous variables.

In many machine learning problems, parameters may be associated with the edges, which are used as additional arguments to the functions computed at nodes connected to these edges. These parameters are learned in a *data-driven* manner so that variables in the nodes mirror relationships among attribute values in data instances. Each data instance contains both *input* and *target* attributes. The variables in a subset of the *input* nodes are fixed to input attribute values in data instances, whereas the variables in all other nodes are *computed* using the node-specific functions. The variables in some of the computed nodes are compared to observed *target* values in data instances, and edge-specific parameters are modified to match the observed and computed values as closely as possible. By learning the parameters along the edges in a data-driven manner, one can learn a function relating the input and target attributes in the data.

In this chapter, we will primarily focus on directed acyclic graphs with continuous, deterministic variables. A *feed-forward neural network* is an important special case of this type

C. C. Aggarwal, *Linear Algebra and Optimization for Machine Learning*,
https://doi.org/10.1007/978-3-030-40344-7_11

of computational graph. The inputs often correspond to the features in each data point, whereas the output nodes might correspond to the target variables (e.g., class variable or regressand). The optimization problem is defined over the edge parameters so that the predicted variables match the observed values in the corresponding nodes as closely as possible. In other words, the *loss function* of a computational graph might penalize differences between predicted and observed values. In computational graphs with continuous variables, one can use gradient descent for optimization. *Almost all machine learning problems that we have seen so far in this book, such as linear regression, logistic regression, SVMs, SVD, PCA, and recommender systems, can be modeled as directed acyclic computational graphs with continuous variables.*

This chapter is organized as follows. The next section will introduce the basics of computational graphs. Section 11.3 discusses optimization in directed acyclic graphs. Applications to neural networks are discussed in Section 11.4. A general view of computational graphs is provided in Section 11.5. A summary is given in Section 11.6.

11.2 The Basics of Computational Graphs

We will define the notion of a directed acyclic computational graph (without cycles).

Definition 11.2.1 (Directed Acyclic Computational Graph) *A directed acyclic computational graph contains nodes, so that each node is associated with a variable. A set of directed edges connect nodes, which indicate functional relationships among nodes. Edges* **might** *be associated with learnable parameters. A variable in a node is either fixed externally (for input nodes with no incoming edges), or it is computed as a function of the variables in the tail ends of edges incoming into the node and the learnable parameters on the incoming edges.*

It is technically possible to define computational graphs with cycles, although we do not consider this rare possibility. The computational graph contains three types of nodes, which are the input, output, and hidden nodes. The input nodes contain the external inputs to the computational graph, and the output node(s) contain(s) the final output(s). The hidden nodes contain intermediate values. Each hidden and output node computes a relatively simple **local** function of its incoming node variables. The cascading effect of the computations over the whole graph implicitly defines a **global** vector-to-vector function from input to output nodes. The variable in each input node is fixed to an externally specified input value. Therefore, no function is computed at an input node. The node-specific functions also use parameters associated with their incoming edges, and the inputs along those edges are scaled with the weights. By choosing weights appropriately, one can control the (global) function defined by the computational graph. This global function is often *learned* by feeding the computational graph input-output pairs (training data) and adjusting the weights so that predicted outputs matched observed outputs.

An example of a computational graph with two weighted edges is provided in Figure 11.1. This graph has three inputs, denoted by x_1, x_2, and x_3. Two of the edges have weights w_2 and w_3. Other than the input nodes, all nodes perform a computation such as addition, multiplication, or evaluating a function like the logarithm. In the case of weighted edges, the values at the tail of the edge are scaled with the weights before computing the node-specific function. The graph has a single output node, and computations are cascaded in

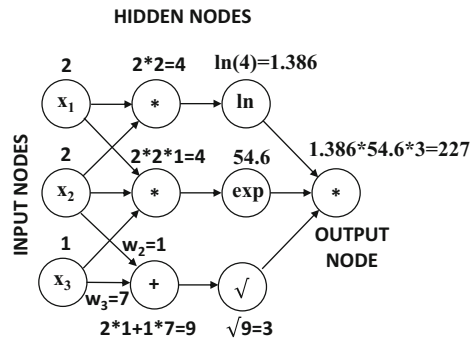


Figure 11.1: Examples of computational graph with two weighted edges

the forward direction from the input to the output. For example, if the weights w_2 and w_3 are chosen to be 1 and 7, respectively, the global function $f(x_1, x_2, x_3)$ is as follows:

$$f(x_1, x_2, x_2) = \ln(x_1 x_2) \cdot \exp(x_1 x_2 x_3) \cdot \sqrt{x_2 + 7x_3}$$

For $[x_1, x_2, x_3] = [2, 2, 1]$, the cascading sequence of computations is shown in the figure with a final output value of approximately 227.1. However, if the *observed* value of the output is only 100, it means that the weights need to be readjusted to change the computed function. In this case, one can observe from inspection of the computational graph that reducing either w_2 or w_3 will help reduce the output value. For example, if we change the weight w_3 to -1, while keeping $w_2 = 1$, the computed function becomes the following:

$$f(x_1, x_2, x_2) = \ln(x_1 x_2) \cdot \exp(x_1 x_2 x_3) \cdot \sqrt{x_2 - x_3}$$

In this case, for the same set of inputs $[x_1, x_2, x_3] = [2, 2, 1]$, the computed output becomes 75.7, which is much closer to the true output value of 100. Therefore, it is clear that one must use the mismatch of predicted values with observed outputs to adjust the computational function, so that there is a better matching between predicted and observed outputs across the data set. Although we adjusted w_3 here by inspection, such an approach will not work in very large computational graphs containing millions of weights.

The goal in machine learning is to learn parameters (like weights) using examples of input-output pairs, while adjusting weights with the help of the observed data. The key point is to convert the problem of adjusting weights into an optimization problem. The computational graph may be associated with a loss function, which typically penalizes the differences in the *predicted* outputs from *observed* outputs, and adjusts weights accordingly. Since the outputs are functions of inputs and edge-specific parameters, the loss function can also be viewed as a complex function of the inputs and edge-specific parameters. The goal of learning the parameters is to minimize the loss, so that the input-output pairs in the computational graph mimic the input-output pairs in the observed data. It should be immediately evident that the problem of learning the weights is likely to challenging, if the underlying computational graph is large with a complex topology.

The choice of loss function depends on the application at hand. For example, one can model least-squares regression by using as many input nodes as the number of input variables (regressors), and a single output node containing the predicted regressand. Directed edges exist from each input node to this output node, and the parameter on each such edge

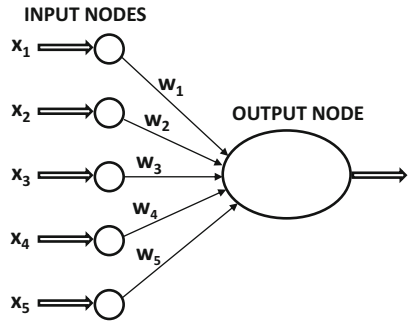


Figure 11.2: A single-layer computational graph that can perform linear regression

corresponds to the weight associated with that input variable (cf. Figure 11.2). The output node computes the following function of the variables $x_1 \ldots x_d$ in the d input nodes:

$$\hat{o} = f(x_1, x_2, \ldots, x_d) = \sum_{i=1}^{d} w_i x_i$$

If the observed regressand is o, then the loss function simply computes $(o - \hat{o})^2$, and adjusts the weights $w_1 \ldots w_d$ so as to reduce this value. Typically, the derivative of the loss is computed with respect to each weight in the computational graph, and the weights are updated by using this derivative. One processes each training point one-by-one and updates the weights. *The resulting algorithm is identical to using stochastic gradient descent in the linear regression problem* (cf. Section 4.7 of Chapter 4). In fact, by changing the nature of the loss function at the output node, it is possible to model both logistic regression and the support vector machine.

Problem 11.2.1 (Logistic Regression with Computational Graph) *Let o be an observed* **binary** *class label drawn from $\{-1, +1\}$ and $\hat{o}$ be the predicted* **real** *value by the neural architecture of Figure 11.2. Show that the loss function $log(1 + exp(-o\hat{o}))$ yields the same loss function for each data instance as the logistic regression model in Equation 4.56 of Chapter 4. Ignore the regularization term in Chapter 4.*

Problem 11.2.2 (SVM with Computational Graph) *Let o be an observed* **binary** *class label drawn from $\{-1, +1\}$ and $\hat{o}$ be the predicted* **real** *value by the neural architecture of Figure 11.2. Show that the loss function $max\{0, 1 - o\hat{o}\}$ yields the same loss function for each data instance as the L_1-loss SVM in Equation 4.51 of Chapter 4.*

In the particular case of Figure 11.2, the choice of a computational graph for model representation does not seem to be useful because a single computational node is rather rudimentary for model representation; indeed, one can directly compute gradients of the loss function with respect to the weights without worrying about computational graphs at all! The main usefulness of computational graphs is realized when the topology of computation is more complex.

The nodes in the directed acyclic graph of Figure 11.2 are arranged in *layers*, because all paths from an input node to any node in the network have the same length. This type of architecture is common in computational graphs. Nodes that are reachable by a path of a particular length i from input nodes are assumed to belong to layer i. At first glance,

Figure 11.2 looks like a two-layer network. However, such networks are considered single-layer networks, because the non-computational input layer is not counted among the number of layers.

11.2.1 Neural Networks as Directed Computational Graphs

The real power of computational graphs is realized when one uses multiple layers of nodes. Neural networks represent the most common use case of a multi-layer computational graph. The nodes are (typically) arranged in layerwise fashion, so that all nodes in layer-i are connected to nodes in layer-$(i+1)$ (and no other layer). The vector of variables in each layer can be written as a vector-to-vector function of the variables in the previous layer. A pictorial illustration of a multilayer neural network is shown in Figure 11.3(a). In this case, the network contains three computational layers in addition to the input layer. For example, consider the first hidden layer with output values $h_{11} \ldots h_{1r} \ldots h_{1,p_1}$, which can be computed as a function of the input nodes with variables $x_1 \ldots x_d$ in the input layer as follows:

$$h_{1r} = \Phi(\sum_{i=1}^{d} w_{ir} x_i) \quad \forall r \in \{1, \ldots, p_1\}$$

The value p_1 represents the number of nodes in the first hidden layer. Here, the function $\Phi(\cdot)$ is referred to as an *activation* function. The final numerical value of the variable in a particular node (i.e., h_{1r} in this case) for a particular input is also sometimes referred to as its *activation* for that input. In the case of linear regression, the activation function is missing, which is also referred to as using the *identity* activation function or *linear* activation function. However, computational graphs primarily gain better expressive power by using nonlinear activation functions such as the following:

$$\Phi(v) = \frac{1}{1+e^{-v}} \qquad \text{[Sigmoid function]}$$

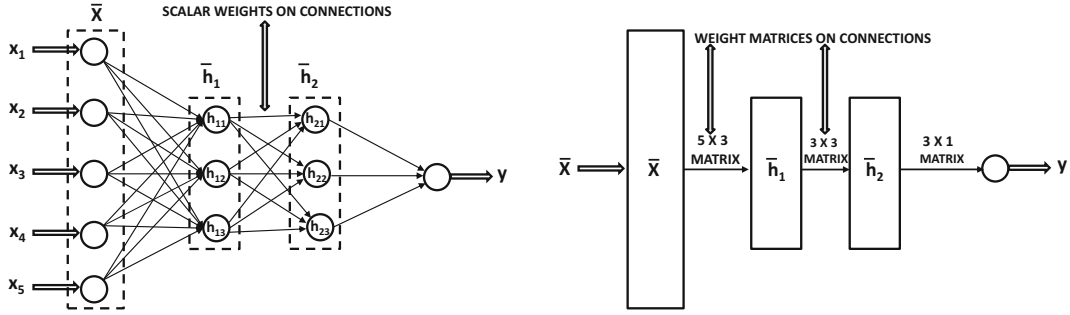
$$\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1} \qquad \text{[Tanh function]}$$

$$\Phi(v) = \max\{v, 0\} \qquad \text{[ReLU: Rectified Linear Unit]}$$

$$\Phi(v) = \max\{\min[v, 1], -1\} \qquad \text{[Hard tanh]}$$

It is noteworthy that these functions are nonlinear, and nonlinearity is essential for greater expressive power of networks with increased depth. Networks containing only linear activation functions are not any more powerful than single-layer networks.

In order to understand this point, consider a two-layer computational graph (not counting the input layer) with 4-dimensional input vector $\overline{x}$, 3-dimensional hidden-layer vector $\overline{h}$, and 2-dimensional output-layer vector $\overline{o}$. Note that we are creating a column vector from the node variables in each layer. Let W_1 and W_2 be two matrices of sizes 3×4 and 2×3 so that $\overline{h} = W_1 \overline{x}$ and $\overline{o} = W_2 \overline{h}$. The matrices W_1 and W_2 contain the weight parameters of each layer. Note that one can express $\overline{o}$ directly in terms of $\overline{x}$ without using $\overline{h}$ as $\overline{o} = W_2 W_1 \overline{x} = (W_2 W_1)\overline{x}$. One can replace the matrix $W_2 W_1$ with a single 2×4 matrix W without any loss of expressive power. In other words, this is a single-layer network! It is not possible to use this type of approach to (easily) eliminate the hidden layer in the case of nonlinear activation functions without creating extremely complex functions at individual nodes (thereby increasing node-specific complexity). This means that increased depth results in increased complexity only when using nonlinear activation functions.



(a) Scalar notation and architecture (b) Vector notation and architecture

Figure 11.3: A feed-forward network with two hidden layers and a single output layer

In the case of Figure 11.3(a), the neural network contains three layers. Note that the input layer is often not counted, because it simply transmits the data and no computation is performed in that layer. If a neural network contains $p_1 \ldots p_k$ units in each of its k layers, then the (column) vector representations of these outputs, denoted by $\overline{h}_1 \ldots \overline{h}_k$ have dimensionalities $p_1 \ldots p_k$. Therefore, the number of units in each layer is referred to as the *dimensionality* of that layer. It is also possible to create a computational graph in which the variables in nodes are vectors, and the connections represent vector-to-vector functions. Figure 11.3(b) creates a computational graph in which the nodes are represented by *rectangles* rather than *circles*. Rectangular representations of nodes correspond to nodes containing vectors. The connections now contain matrices. The sizes of the corresponding *connection* matrices are shown in Figure 11.3(b). For example, if the input layer contains 5 nodes and the first hidden layer contains 3 nodes, the *connection matrix* is of size 5×3. However, as we will see later, the *weight* matrix has size that is the transpose of the connection matrix (i.e., 3×5) in order to facilitate matrix operations. Note that the computational graph in the vector notation has a simpler structure, where the entire network contains only a single path. The weights of the connections between the input layer and the first hidden layer are contained in a *matrix* W_1 with size $p_1 \times d$, whereas the weights between the rth hidden layer and the $(r+1)$th hidden layer are denoted by the $p_{r+1} \times p_r$ matrix denoted by W_r. If the output layer contains s nodes, then the final matrix W_{k+1} is of size $s \times p_k$. Note that the weight matrix has transposed dimensions with respect to the connection matrix. The d-dimensional input vector $\overline{x}$ is transformed into the outputs using the following recursive equations:

$$\overline{h}_1 = \Phi(W_1 \overline{x}) \qquad \text{[Input to Hidden Layer]}$$

$$\overline{h}_{p+1} = \Phi(W_{p+1} \overline{h}_p) \quad \forall p \in \{1 \ldots k-1\} \qquad \text{[Hidden to Hidden Layer]}$$

$$\overline{o} = \Phi(W_{k+1} \overline{h}_k) \qquad \text{[Hidden to Output Layer]}$$

Here, the activation functions are applied in *element-wise* fashion to their vector arguments. Here, it is noteworthy that the final output is a **recursively nested composition function of the inputs**, which is as follows:

$$\overline{o} = \Phi(W_{k+1}(\Phi(W_k \Phi(W_{k-1} \ldots))))$$

This type of neural network is harder to train than single-layer networks because one must compute the derivative of a **nested** composition function with respect to each weight. In

particular, the weights of earlier layers lie inside the recursive nesting, and are harder to learn with gradient descent, because the methodology for computation of the gradient of weights in the inner portions of the nesting (i.e., earlier layers) is not obvious, especially when the computational graph has a complex topology. It is also noticeable that the **global** input-to-output function computed by the neural network is harder to express in closed form neatly. The recursive nesting makes the closed-form representation look extremely cumbersome. A cumbersome closed-form representation causes challenges in derivative computation for parameter learning.

11.3 Optimization in Directed Acyclic Graphs

The optimization of loss functions in computational graphs requires the computation of gradients of the loss functions with respect to the network weights. This computation is done using *dynamic programming* (cf. Section 5.8.4 of Chapter 5). Dynamic programming is a technique from optimization that can be used to compute all types of path-centric functions in *directed acyclic graphs*.

In order to train computational graphs, it is assumed that we have training data corresponding to input-output pairs. The number of input nodes is equal to the number of input attributes and the number of output nodes is equal to the number of output attributes. The computational graph can predict the outputs using the inputs, and compare them to the observed outputs in order to check whether the function computed by the graph is consistent with the training data. If this is not the case, the weights of the computational graph need to be modified.

11.3.1 The Challenge of Computational Graphs

A computational graph naturally evaluates compositions of functions. Consider a variable x at a node in a computational graph with only three nodes containing a path of length 2. The first node applies the function $g(x)$, whereas the second node applies the function $f(\cdot)$ to the result. Such a graph computes the function $f(g(x))$, and it is shown in Figure 11.4. The example shown in Figure 11.4 uses the case when $f(x) = \cos(x)$ and $g(x) = x^2$. Therefore, the overall function is $\cos(x^2)$. Now, consider another setting in which both $f(x)$ and $g(x)$ are set to the same function, which is the sigmoid function:

$$f(x) = g(x) = \frac{1}{1 + \exp(-x)}$$

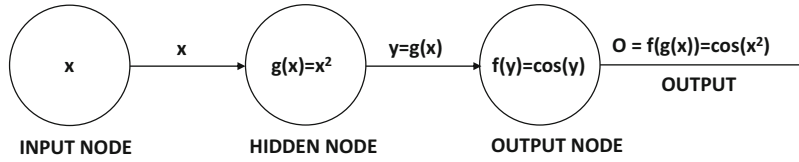


Figure 11.4: A simple computational graph with an input node and two computational nodes

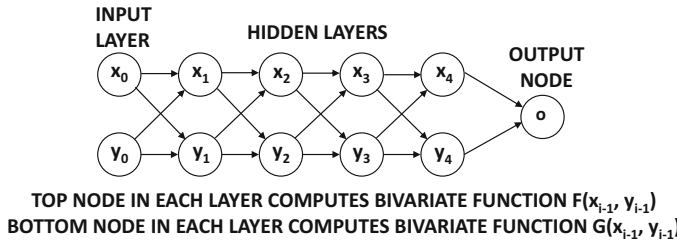


Figure 11.5: The awkwardness of recursive nesting caused by a computational graph

Then, the global function evaluated by the computational graph is as follows:

$$f(g(x)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-x)}\right]} \tag{11.1}$$

This simple graph already computes a rather awkward composition function. Trying the find the derivative of this composition function becomes increasingly tedious with increasing complexity of the graph.

Consider a case in which the functions $g_1(\cdot)$, $g_2(\cdot) \ldots g_k(\cdot)$ are the functions computed in layer m, and they feed into a particular layer-$(m+1)$ node that computes the multivariate function $f(\cdot)$ that uses the values computed in the previous layer as arguments. Therefore, the layer-$(m+1)$ function computes $f(g_1(\cdot), \ldots g_k(\cdot))$. This type of multivariate composition function already appears rather awkward. As we increase the number of layers, a function that is computed several edges downstream will have as many layers of nesting as the length of the path from the source to the final output. For example, if we have a computational graph which has 10 layers, and 2 nodes per layer, the overall composition function would have 2^{10} nested "terms". This makes the handling of closed-form functions of deep networks unwieldy and impractical.

In order to understand this point, consider the function in Figure 11.5. In this case, we have two nodes in each layer other than the output layer. The output layer simply sums its inputs. Each hidden layer contains two nodes. The variables in the ith layer are denoted by x_i and y_i, respectively. The input nodes (variables) use subscript 0, and therefore they are denoted by x_0 and y_0 in Figure 11.5. The two computed functions in the ith layer are $F(x_{i-1}, y_{i-1})$ and $G(x_{i-1}, y_{i-1})$, respectively.

In the following, we will write the expression for the variable in each node in order to show the increasing complexity with increasing number of layers:

$$\begin{aligned}
x_1 &= F(x_0, y_0) \\
y_1 &= G(x_0, y_0) \\
x_2 &= F(x_1, y_1) = F(F(x_0, y_0), G(x_0, y_0)) \\
y_2 &= G(x_1, y_1) = G(F(x_0, y_0), G(x_0, y_0))
\end{aligned}$$

We can already see that the expressions have already started looking unwieldy. On computing the values in the next layer, this becomes even more obvious:

$$\begin{aligned}
x_3 &= F(x_2, y_2) = F(F(F(x_0, y_0), G(x_0, y_0)), G(F(x_0, y_0), G(x_0, y_0))) \\
y_3 &= G(x_2, y_2) = G(F(F(x_0, y_0), G(x_0, y_0)), G(F(x_0, y_0), G(x_0, y_0)))
\end{aligned}$$

An immediate observation is that the complexity and length of the closed-form function increases *exponentially* with the path lengths in the computational graphs. This type of complexity further increases in the case when optimization parameters are associated with the edges, and one tries to express the outputs/losses in terms of the inputs and the parameters on the edges. This is obviously a problem, if we try to use the boilerplate approach of first expressing the loss function in closed form in terms of the optimization parameters on the edges (in order to compute the derivative of the closed-form loss function).

11.3.2 The Broad Framework for Gradient Computation

The previous section makes it evident that differentiating closed-form expressions is not practical in the case of computational graphs. Therefore, one must somehow *algorithmically* compute gradients with respect to edges by using the topology of the computational graph. The purpose of this section is to introduce this broad algorithmic framework, and later sections will expand on the specific details of individual steps.

To learn the weights of a computational graph, an input-output pair is selected from the training data and the error of trying to predict the observed output with the observed input with the current values of the weights in the computational graph is quantified. When the errors are large, the weights need to be modified because the current computational graph does not reflect the observed data. Therefore, a loss function is computed as a function of this error, and the weights are updated so as to reduce the loss. This is achieved by computing the gradient of the loss with respect to the weights and performing a gradient-descent update. The overall approach for training a computational graph is as follows:

1. Use the attribute values from the input portion of a training data point to fix the values in the input nodes. Repeatedly select a node for which the values in all incoming nodes have already been computed and apply the node-specific function to also compute its variable. Such a node can be found in a directed acyclic graph by processing the nodes in order of increasing distance from input nodes. Repeat the process until the values in all nodes (including the output nodes) have been computed. If the values on the output nodes do not match the observed values of the output in the training point, compute the loss value. This phase is referred to as the *forward phase*.

2. Compute the gradient of the loss with respect to the weights on the edges. This phase is referred to as the *backwards phase*. The rationale for calling it a "backwards phase" will become clear later, when we introduce an algorithm that works backwards along the topology of the (directed acyclic) computational graph from the outputs to the inputs.

3. Update the weights in the negative direction of the gradient.

As in any stochastic gradient descent procedure, one cycles through the training points repeatedly until convergence is reached. A single cycle through all the training points is referred to as an *epoch*.

The main challenge is in computing the gradient of the loss function with respect to the weights in a computational graph. It turns out that *the derivatives of the node variables with respect to one another can be easily used to compute the derivative of the loss function with respect to the weights on the edges*. Therefore, in this discussion, we will focus on the computation of the derivatives of the variables with respect to one another. Later, we will show how these derivatives can be converted into gradients of loss functions with respect to weights.

11.3.3 Computing Node-to-Node Derivatives Using Brute Force

As discussed in an earlier section, one can express the function in a computational graph in terms of the nodes in early layers using an awkward closed-form expression that uses nested compositions of functions. If one were to indeed compute the derivative of this closed-form expression, it would require the use of the *chain rule of differential calculus* in order to deal with the repeated composition of functions. However, a blind application of the chain rule is rather wasteful in this case because many of the expressions in different portions of the inner nesting are identical, and one would be repeatedly computing the same derivative. The key idea in *automatic differentiation over computational graphs* is to recognize the fact that structure of the computational graph already provides all the information about which terms are repeated. We can avoid repeating the differentiation of these terms by using the structure of the computational graph itself to store intermediate results (by working backwards starting from output nodes to compute derivatives)! This is a well-known idea from dynamic programming, which has been used frequently in control theory [26, 71]. In the neural network community, this same algorithm is referred to as *backpropagation* (cf. Section 11.4). It is noteworthy that the applications of this idea in control theory were well-known to the traditional optimization community in 1960 [26, 71], although they remained unknown to researchers in the field of artificial intelligence for a while (who coined the term "backpropagation" in the 1980s to independently propose and describe this idea in the context of neural networks).

The simplest version of the chain rule is defined for a univariate composition of functions:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x} \tag{11.2}$$

This variant is referred to as the *univariate chain rule*. Note that each term on the right-hand side is a *local gradient* because it computes the derivative of a *local* function with respect to its immediate argument rather than a recursively derived argument. The basic idea is that a composition of functions is applied on the input x to yield the final output, and the gradient of the final output is given by the product of the local gradients along that path. Each local gradient only needs to worry about its specific input and output, which simplifies the computation. An example is shown in Figure 11.4 in which the function $f(y)$ is $\cos(y)$ and $g(x) = x^2$. Therefore, the composition function is $\cos(x^2)$. On using the univariate chain rule, we obtain the following:

$$\frac{\partial f(g(x))}{\partial x} = \underbrace{\frac{\partial f(g(x))}{\partial g(x)}}_{-\sin(g(x))} \cdot \underbrace{\frac{\partial g(x)}{\partial x}}_{2x} = -2x \cdot \sin(x^2)$$

Note that we can annotate each of the above two multiplicative components on the two *connections* in the graph, and simply compute the product of these values. Therefore, *for a computational graph containing a single path, the derivative of one node with respect to another is simply the product of these annotated values on the connections between the two nodes.* The example of Figure 11.4 is a rather simple case in which the computational graph is a single path. In general, a computational graph with good expressive power will not be a single path. Rather, a single node may feed its output to multiple nodes. For example, consider the case in which we have a single input x, and we have k independent computational nodes that compute the functions $g_1(x), g_2(x), \ldots g_k(x)$. If these nodes are connected to a single output node computing the function $f()$ with k arguments, then the

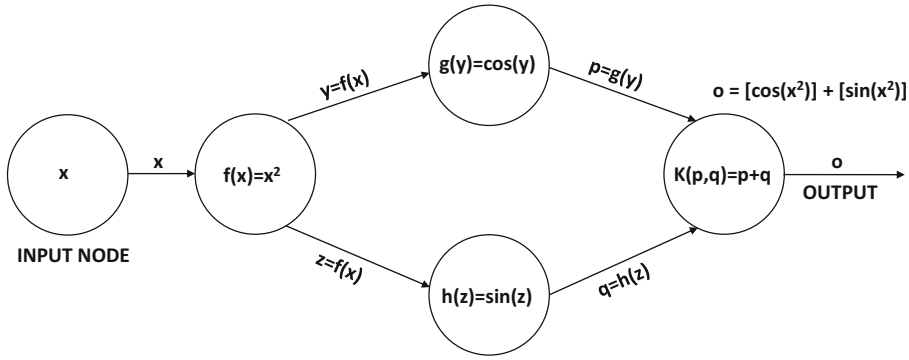


Figure 11.6: A simple computational function that illustrates the chain rule

resulting function that is computed is $f(g_1(x), \ldots g_k(x))$. In such cases, the *multivariate chain rule* needs to be used. The multivariate chain rule is defined as follows:

$$\frac{\partial f(g_1(x), \ldots g_k(x))}{\partial x} = \sum_{i=1}^{k} \frac{\partial f(g_1(x), \ldots g_k(x))}{\partial g_i(x)} \cdot \frac{\partial g_i(x)}{\partial x} \tag{11.3}$$

It is easy to see that the multivariate chain rule of Equation 11.3 is a simple generalization of that in Equation 11.2.

One can also view the multivariate chain rule in a path-centric fashion rather than a node-centric fashion. *For any pair of source-sink nodes, the derivative of the variable in the sink node with respect to the variable in the source node is simply the sum of the expressions arising from the univariate chain rule being applied to all paths existing between that pair of nodes.* This view leads to a direct expression for the derivative between any pair of nodes (rather than the recursive multivariate rule). However, it leads to an excessive computation, because the number of paths between a pair of nodes is exponentially related to the path length. In order to show the repetitive nature of the operations, we work with a very simple closed-form function with a single input x:

$$o = \sin(x^2) + \cos(x^2) \tag{11.4}$$

The resulting computational graph is shown in Figure 11.6. In this case, the multivariate chain rule is applied to compute the derivative of the output o with respect to x. This is achieved by summing the results of the univariate chain rule for each of the two paths from x to o in Figure 11.6:

$$\begin{aligned}
\frac{\partial o}{\partial x} &= \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{-\sin(y)} \cdot \underbrace{f'(x)}_{2x} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(x)}_{2x} \\
&= -2x \cdot \sin(y) + 2x \cdot \cos(z) \\
&= -2x \cdot \sin(x^2) + 2x \cdot \cos(x^2)
\end{aligned}$$

In this simple example, there are two paths, both of which compute the function $f(x) = x^2$. As a result, the function $f(x)$ is differentiated *twice*, once for each path. This type of repetition can have severe effects for large multilayer networks containing many shared nodes, where the same function might be differentiated hundreds of thousands of times as a

portion of the nested recursion. It is this *repeated and wasteful* approach to the computation of the derivative, that it is impractical to express the global function of a computational graph in closed form and explicitly differentiating it.

One can summarize the path-centric view of the multivariate chain rule as follows:

Lemma 11.3.1 (Pathwise Aggregation Lemma) *Consider a directed acyclic computational graph in which the ith node contains variable $y(i)$. The local derivative $z(i, j)$ of the directed edge (i, j) in the graph is defined as $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$. Let a non-null set of paths $\mathcal{P}$ exist from a node s in the graph to node t. Then, the value of $\frac{\partial y(t)}{\partial y(s)}$ is given by computing the product of the local gradients along each path in $\mathcal{P}$, and summing these products over all paths in $\mathcal{P}$.*

$$\frac{\partial y(t)}{\partial y(s)} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j) \tag{11.5}$$

This lemma can be easily shown by applying the multivariate chain rule (Equation 11.3) recursively over the computational graph. Although the use of the pathwise aggregation lemma is a wasteful approach for computing the derivative of $y(t)$ with respect to $y(s)$, it enables a simple and intuitive exponential-time algorithm for derivative computation.

An Exponential-Time Algorithm

The pathwise aggregation lemma provides a natural exponential-time algorithm, which is roughly similar to the steps one would go through by expressing the computational function in closed form with respect to a particular variable and then differentiating it. Specifically, the pathwise aggregation lemma leads to the following exponential-time algorithm to compute the derivative of the output o with respect to a variable x in the graph:

1. Use computational graph to compute the value $y(i)$ of each node i in a forward phase.
2. Compute the local partial derivatives $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge in the computational graph.
3. Let $\mathcal{P}$ be the set of all paths from an input node with value x to the output o. For each path $P \in \mathcal{P}$ compute the product of each local derivative $z(i, j)$ on that path.
4. Add up these values over all paths in $\mathcal{P}$.

In general, a computational graph will have an exponentially increasing number of paths with depth and one must add the product of the local derivatives over all paths. An example is shown in Figure 11.7, in which we have five layers, each of which has only two units. Therefore, the number of paths between the input and output is $2^5 = 32$. The jth hidden unit of the ith layer is denoted by $h(i, j)$. Each hidden unit is defined as the product of its inputs:

$$h(i, j) = h(i - 1, 1) \cdot h(i - 1, 2) \quad \forall j \in \{1, 2\} \tag{11.6}$$

In this case, the output is x^{32}, which is expressible in closed form, and can be differentiated easily with respect to x. In other words, we do not really need computational graphs in order to perform the differentiation. However, we will use the exponential-time algorithm to elucidate its workings. The derivatives of each $h(i, j)$ with respect to its two inputs are

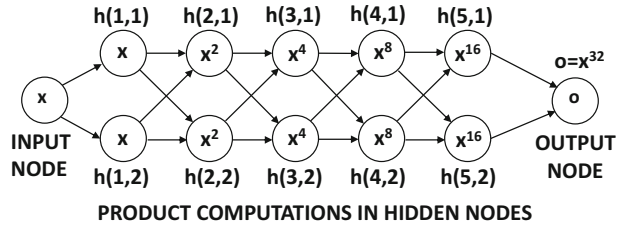


Figure 11.7: The chain rule aggregates the product of local derivatives along $2^5 = 32$ paths

the values of the complementary inputs, because the partial derivative of the multiplication of two variables is the complementary variable:

$$\frac{\partial h(i,j)}{\partial h(i-1,1)} = h(i-1,2), \quad \frac{\partial h(i,j)}{\partial h(i-1,2)} = h(i-1,1)$$

The pathwise aggregation lemma implies that the value of $\frac{\partial o}{\partial x}$ is the product of the local derivatives (which are the complementary input values in this particular case) along all 32 paths from the input to the output:

$$\begin{aligned}\frac{\partial o}{\partial x} &= \sum_{j_1,j_2,j_3,j_4,j_5 \in \{1,2\}^5} \prod \underbrace{h(1,j_1)}_{x} \underbrace{h(2,j_2)}_{x^2} \underbrace{h(3,j_3)}_{x^4} \underbrace{h(4,j_4)}_{x^8} \underbrace{h(5,j_5)}_{x^{16}} \\ &= \sum_{\text{All 32 paths}} x^{31} = 32x^{31}\end{aligned}$$

This result is, of course, consistent with what one would obtain on differentiating x^{32} directly with respect to x. However, an important observation is that it requires 2^5 aggregations to compute the derivative in this way for a relatively simple graph. More importantly, *we repeatedly differentiate the same function computed in a node* for aggregation. For example, the differentiation of the variable $h(3,1)$ is performed 16 times because it appears in 16 paths from x to o.

Obviously, this is an inefficient approach to compute gradients. For a network with 100 nodes in each layer and three layers, we will have a million paths. *Nevertheless, this is exactly what we do in traditional machine learning when our prediction function is a complex composition function.* Manually working out the details of a complex composition function is tedious and impractical beyond a certain level of complexity. It is here that one can apply dynamic programming (which is guided by the structure of the computational graph) in order to store important intermediate results. By using such an approach, one can minimize repeated computations, and achieve polynomial complexity.

11.3.4 Dynamic Programming for Computing Node-to-Node Derivatives

In graph theory, computing all types of path-aggregative values over directed acyclic graphs is done using dynamic programming. Consider a directed acyclic graph in which the value $z(i,j)$ (interpreted as local partial derivative of variable in node j with respect to variable

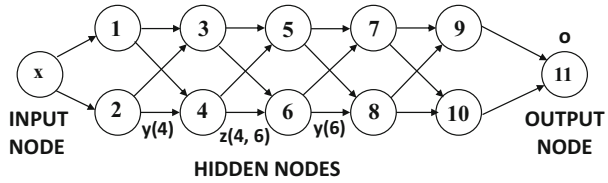


Figure 11.8: Edges are labeled with local partial derivatives such as $z(4,6) = \frac{\partial y(6)}{\partial y(4)}$

in node i) is associated with edge (i, j). In other words, if $y(p)$ is the variable in the node p, we have the following:

$$z(i,j) = \frac{\partial y(j)}{\partial y(i)} \tag{11.7}$$

An example of such a computational graph is shown in Figure 11.8. In this case, we have associated the edge $(2, 4)$ with the corresponding partial derivative. We would like to compute the product of $z(i, j)$ over each path $P \in \mathcal{P}$ from source node s to output node t and then add them in order to obtain the partial derivative $S(s,t) = \frac{\partial y(t)}{\partial y(s)}$:

$$S(s,t) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i,j) \tag{11.8}$$

Let $A(i)$ be the set of nodes at the end points of outgoing edges from node i. We can compute the aggregated value $S(i,t)$ for each intermediate node i (between source node s and output node t) using the following well-known dynamic programming update:

$$S(i,t) \Leftarrow \sum_{j \in A(i)} S(j,t) z(i,j) \tag{11.9}$$

This computation can be performed backwards starting from the nodes directly incident on o, since $S(t,t) = \frac{\partial y(t)}{\partial y(t)}$ is already known to be 1. This is because the partial derivative of a variable with respect to itself is always 1. Therefore one can describe the pseudocode of this algorithm as follows:

```
Initialize S(t, t) = 1;
repeat
  Select an unprocessed node i such that the values of S(j, t) all of its outgoing
      nodes j ∈ A(i) are available;
  Update S(i, t) ⇐ Σ_{j∈A(i)} S(j, t)z(i, j);
until all nodes have been selected;
```

Note that the above algorithm always selects a node i for which the value of $S(j,t)$ is available for all nodes $j \in A(i)$. Such a node is always available in directed acyclic graphs, and the node selection order will always be in the backwards direction starting from node t. Therefore, the above algorithm will work only when the computational graph does not have cycles, and it is referred to as the *backpropagation algorithm*.

The algorithm discussed above is used by the network optimization community for computing all types of path-centric functions between *source-sink* node pairs (s,t) on directed acyclic graphs, which would otherwise require exponential time. For example, one can even use a variation of the above algorithm to find the longest path in a directed acyclic graph [8].

Interestingly, *the aforementioned dynamic programming update is exactly the multivariate chain rule of Equation 11.3, which is repeated in the backwards direction starting at the output node where the local gradient is known.* This is because we derived the path-aggregative form of the loss gradient (Lemma 11.3.1) using this chain rule in the first place. The main difference is that we apply the rule in a particular order in order to minimize computations. We emphasize this important point below:

> Using dynamic programming to efficiently aggregate the product of local gradients along the exponentially many paths in a computational graph results in a dynamic programming update that is identical to the multivariate chain rule of differential calculus. The main point of dynamic programming is to apply this rule in a particular order, so that the derivative computations at different nodes are not repeated.

This approach is the backbone of the backpropagation algorithm used in neural networks. We will discuss more details of neural network-specific enhancements in Section 11.4. In the case where we have multiple output nodes $t_1, \ldots t_p$, one can initialize each $S(t_r, t_r)$ to 1, and then apply the same approach for each t_r.

11.3.4.1 Example of Computing Node-to-Node Derivatives

In order to show how the backpropagation approach works, we will provide an example of computation of node-to-node derivatives in a graph containing 10 nodes (see Figure 11.9). A variety of functions are computed in various nodes, such as the sum function (denoted by '+'), the product function (denoted by '*'), and the trigonometric sine/cosine functions. The variables in the 10 nodes are denoted by $y(1) \ldots y(10)$, where the variable $y(i)$ belongs to the ith node in the figure. Two of the edges incoming into node 6 also have the weights w_2 and w_3 associated with them. Other edges do not have weights associated with them. The functions computed in the various layers are as follows:

Layer 1: $y(4) = y(1) \cdot y(2)$, $y(5) = y(1) \cdot y(2) \cdot y(3)$, $y(6) = w_2 \cdot y(2) + w_3 \cdot y(3)$

Layer 2: $y(7) = \sin(y(4))$, $y(8) = \cos(y(5))$, $y(9) = \sin(y(6))$

Layer 3: $y(10) = y(7) \cdot y(8) \cdot y(9)$

We would like to compute the derivative of $y(10)$ with respect to each of the inputs $y(1)$, $y(2)$, and $y(3)$. One possibility is to simply express the $y(10)$ in closed form in terms of the inputs $y(1)$, $y(2)$ and $y(3)$, and then compute the derivative. By recursively using the above relationships, it is easy to show that $y(10)$ can be expressed in terms of $y(1)$, $y(2)$, and $y(3)$ as follows:

$$y(10) = \sin(y(1) \cdot y(2)) \cdot \cos(y(1) \cdot y(2) \cdot y(3)) \cdot \sin(w_2 \cdot y(2) + w_3 \cdot y(3))$$

As discussed earlier computing the closed-form derivative is not practical for larger networks. Furthermore, since one needs to compute the derivative of the output with respect to each and every node in the network, such an approach would also required closed-form expressions in terms of upstream nodes like $y(4)$, $y(5)$ and $y(6)$. All this tends to increase the amount of repeated computation. Luckily, backpropagation frees us from this repeated computation, since the derivative in $y(10)$ with respect to each and every node is computed by the backwards phase. The algorithm starts, by initializing the derivative of the output $y(10)$ with respect to itself, which is 1:

$$S(10, 10) = \frac{\partial y(10)}{\partial y(10)} = 1$$

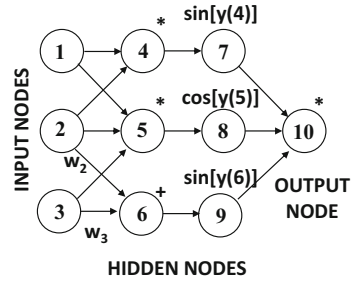


Figure 11.9: Example of node-to-node derivative computation

Subsequently, the derivatives of $y(10)$ with respect to all the variables on its incoming nodes are computed. Since $y(10)$ is expressed in terms of the variables $y(7)$, $y(8)$, and $y(9)$ incoming into it, this is easy to do, and the results are denoted by $z(7, 10)$, $z(8, 10)$, and $z(9, 10)$ (which is consistent with the notations used earlier in this chapter). Therefore, we have the following:

$$z(7,10) = \frac{\partial y(10)}{\partial y(7)} = y(8) \cdot y(9)$$
$$z(8,10) = \frac{\partial y(10)}{\partial y(8)} = y(7) \cdot y(9)$$
$$z(9,10) = \frac{\partial y(10)}{\partial y(9)} = y(7) \cdot y(8)$$

Subsequently, we can use these values in order to compute $S(7, 10)$, $S(8, 10)$, and $S(9, 10)$ using the recursive backpropagation update:

$$S(7,10) = \frac{\partial y(10)}{\partial y(7)} = S(10,10) \cdot z(7,10) = y(8) \cdot y(9)$$
$$S(8,10) = \frac{\partial y(10)}{\partial y(8)} = S(10,10) \cdot z(8,10) = y(7) \cdot y(9)$$
$$S(9,10) = \frac{\partial y(10)}{\partial y(9)} = S(10,10) \cdot z(9,10) = y(7) \cdot y(8)$$

Next, we compute the derivatives $z(4, 7)$, $z(5, 8)$, and $z(6, 9)$ associated with all the edges incoming into nodes 7, 8, and 9:

$$z(4,7) = \frac{\partial y(7)}{\partial y(4)} = \cos[y(4)]$$
$$z(5,8) = \frac{\partial y(8)}{\partial y(5)} = -\sin[y(5)]$$
$$z(6,9) = \frac{\partial y(9)}{\partial y(6)} = \cos[y(6)]$$

These values can be used to compute $S(4, 10)$, $S(5, 10)$, and $S(6, 10)$:

$$S(4,10) = \frac{\partial y(10)}{\partial y(4)} = S(7,10) \cdot z(4,7) = y(8) \cdot y(9) \cdot \cos[y(4)]$$
$$S(5,10) = \frac{\partial y(10)}{\partial y(5)} = S(8,10) \cdot z(5,8) = -y(7) \cdot y(9) \cdot \sin[y(5)]$$
$$S(6,10) = \frac{\partial y(10)}{\partial y(6)} = S(9,10) \cdot z(6,9) = y(7) \cdot y(8) \cdot \cos[y(6)]$$

In order to compute the derivatives with respect to the input values, one now needs to compute the values of $z(1, 3)$, $z(1, 4)$, $z(2, 4)$, $z(2, 5)$, $z(2, 6)$, $z(3, 5)$, and $z(3, 6)$:

$$z(1,4) = \frac{\partial y(4)}{\partial y(1)} = y(2)$$
$$z(2,4) = \frac{\partial y(4)}{\partial y(2)} = y(1)$$
$$z(1,5) = \frac{\partial y(5)}{\partial y(1)} = y(2) \cdot y(3)$$
$$z(2,5) = \frac{\partial y(5)}{\partial y(2)} = y(1) \cdot y(3)$$
$$z(3,5) = \frac{\partial y(5)}{\partial y(3)} = y(1) \cdot y(2)$$
$$z(2,6) = \frac{\partial y(6)}{\partial y(2)} = w_2$$
$$z(3,6) = \frac{\partial y(6)}{\partial y(3)} = w_3$$

These partial derivatives can be backpropagated to compute $S(1, 10)$, $S(2, 10)$, and $S(3, 10)$:

$$\begin{aligned} S(1,10) &= \frac{\partial y(10)}{\partial y(1)} = S(4,10) \cdot z(1,4) + S(5,10) \cdot z(1,5) \\ &= y(8) \cdot y(9) \cdot \cos[y(4)] \cdot y(2) - y(7) \cdot y(9) \cdot \sin[y(5)] \cdot y(2) \cdot y(3) \\ S(2,10) &= \frac{\partial y(10)}{\partial y(2)} = S(4,10) \cdot z(2,4) + S(5,10) \cdot z(2,5) + S(6,10) \cdot z(2,6) \\ &= y(8) \cdot y(9) \cdot \cos[y(4)] \cdot y(1) - y(7) \cdot y(9) \cdot \sin[y(5)] \cdot y(1) \cdot y(3) + \\ &\quad + y(7) \cdot y(8) \cdot \cos[y(6)] \cdot w_2 \\ S(3,10) &= \frac{\partial y(10)}{\partial y(3)} = S(5,10) \cdot z(3,5) + S(6,10) \cdot z(3,6) \\ &= -y(7) \cdot y(9) \cdot \sin[y(5)] \cdot y(1) \cdot y(2) + y(7) \cdot y(8) \cdot \cos[y(6)] \cdot w_3 \end{aligned}$$

Note that the use of a backward phase has the advantage of computing the derivative of $y(10)$ (output node variable) with respect to all the hidden and input node variables. These different derivatives have many sub-expressions in common, although the derivative computation of these sub-expressions is not repeated. This is the advantage of using the backwards phase for derivative computation as opposed to the use of closed-form expressions.

Because of the tedious nature of the closed-form expressions for outputs, the algebraic expressions for derivatives are also very long and awkward (no matter how we compute

them). One can see that this is true even for the simple, ten-node computational graph of this section. For example, if one examines the derivative of $y(10)$ with respect to each of nodes $y(1)$, $y(2)$ and $y(3)$, the algebraic expression wraps into multiple lines. Furthermore, one cannot avoid the presence of repeated subexpressions within the algebraic derivative. This is counter-productive because our original goal in the backwards algorithm was to avoid the repeated computation endemic to traditional derivative evaluation with closed-form expressions. Therefore, one does not *algebraically* compute these types of expressions in real-world networks. One would first *numerically* compute all the node variables for a *specific* set of numerical inputs from the training data. Subsequently, one would *numerically* carry the derivatives backward, so that one does not have to carry the large algebraic expressions (with many repeated sub-expressions) in the backwards direction. The advantage of carrying numerical expressions is that multiple terms get consolidated into a single numerical value, which is specific to a particular input. By making the numerical choice, one must repeat the backwards computation algorithm *for each training point*, but it is still a better choice than computing the (massive) symbolic derivative in one shot and substituting the values in different training points. This is the reason that such an approach is referred to as *numerical differentiation* rather than *symbolic differentiation*. In much of machine learning, one first computes the algebraic derivative (which is symbolic differentiation) before substituting numerical values of the variables in the expression (for the derivative) to perform gradient-descent updates. This is different from the case of computational graphs, where the backwards algorithm is *numerically* applied to each training point.

11.3.5 Converting Node-to-Node Derivatives into Loss-to-Weight Derivatives

Most computational graphs define loss functions with respect to output node variables. One needs to compute the derivatives with respect to weights on *edges* rather than the node variables (in order to update the weights). In general, the node-to-node derivatives can be converted into loss-to-weight derivatives with a few additional applications of the univariate and multivariate chain rule.

Consider the case in which we have computed the node-to-node derivative of output variables in nodes indexed by $t_1, t_2, \ldots t_p$ with respect to the variable in node i using the dynamic programming approach in the previous section. Therefore, the computational graph has p output nodes in which the corresponding variable values are $y(t_1) \ldots y(t_p)$ (since the indices of the output nodes are $t_1 \ldots t_p$). The loss function is denoted by $L(y(t_1), \ldots y(t_p))$. We would like to compute the derivative of this loss function with respect to *the weights in the incoming edges of* i. For the purpose of this discussion, let w_{ji} be the weight of an edge from node index j to node index i. Therefore, we want to compute the derivative of the loss function with respect to w_{ji}. In the following, we will abbreviate $L(y_{t_1}, \ldots y_{t_p})$ with L for compactness of notation:

$$\frac{\partial L}{\partial w_{ji}} = \left[\frac{\partial L}{\partial y(i)}\right] \frac{\partial y(i)}{\partial w_{ji}} \qquad \text{[Univariate chain rule]}$$

$$= \left[\sum_{k=1}^{p} \frac{\partial L}{\partial y(t_k)} \frac{\partial y(t_k)}{\partial y(i)}\right] \frac{\partial y(i)}{\partial w_{ji}} \qquad \text{[Multivariate chain rule]}$$

Here, it is noteworthy that the loss function is typically a closed-form function of the variables in the node indices $t_1 \ldots t_p$, which is often either is least-squares function or a logarithmic loss function (like the examples in Chapter 4). Therefore, each derivative of the

loss L with respect to $y(t_i)$ is easy to compute. Furthermore, the value of each $\frac{\partial y(t_k)}{\partial y(i)}$ for $k \in \{1 \ldots p\}$ can be computed using the dynamic programming algorithm of the previous section. The value of $\frac{\partial y_i}{\partial w_{ji}}$ is a derivative of the **local** function at each node, which usually has a simple form. Therefore, the loss-to-weight derivatives can be computed relatively easily, once the node-to-node derivatives have been computed using dynamic programming.

Although one can apply the pseudocode of page 460 to compute $\frac{\partial y(t_k)}{\partial y(i)}$ for each $k \in \{1 \ldots p\}$, it is more efficient to collapse all these computations into a single backwards algorithm. In practice, one initializes the derivatives at the output nodes to the loss derivatives $\frac{\partial L}{\partial y(t_k)}$ for each $k \in \{1 \ldots p\}$ rather than the value of 1 (as shown in the pseudocode of page 460). Subsequently, the entire loss derivative $\Delta(i) = \frac{\partial L}{\partial y(i)}$ is propagated backwards. Therefore, the modified algorithm for computing the loss derivative with respect to the *node variables* as well as the *edge variables* is as follows:

Initialize $\Delta(t_r) = \frac{\partial L}{\partial y(t_k)}$ for each $k \in \{1 \ldots p\}$;
repeat
  Select an unprocessed node i such that the values of $\Delta(j)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $\Delta(i) \Leftarrow \sum_{j \in A(i)} \Delta(j) z(i, j)$;
until all nodes have been selected;
for each edge (j, i) with weight w_{ji} **do** compute $\frac{\partial L}{\partial w_{ji}} = \Delta(i) \frac{\partial y(i)}{\partial w_{ji}}$;

In the above algorithm, $y(i)$ denotes the variable at node i. The key difference of this algorithm from the algorithm on page 460 is in the nature of the initialization and the addition of a final step computing the edge-wise derivatives. However, the core algorithm for computing the node-to-node derivatives remains an integral part of this algorithm. In fact, one can convert all the weights on the edges into additional "input" nodes containing weight parameters, and also add computational nodes that multiply the weights with the corresponding variables at the tail nodes of the edges. Furthermore, a computational node can be added that computes the loss from the output node(s). For example, the architecture of Figure 11.9 can be converted to that in Figure 11.10. Therefore, a computational graph with *learnable weights* can be converted into an unweighted graph with *learnable node variables* (on a subset of nodes). Performing only node-to-node derivative computation in Figure 11.10 from the loss node to the weight nodes is equivalent to loss-to-weight derivative computation. In other words, *loss-to-weight derivative computation in a weighted graph is equivalent to node-to-node derivative computation in a modified computational graph.* The derivative of the loss with respect to each weight can be denoted by the vector $\frac{\partial L}{\partial \overline{W}}$ (in matrix calculus notation), where $\overline{W}$ denotes the weight vector. Subsequently, the standard gradient descent update can be performed:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}} \tag{11.10}$$

Here, α is the learning rate. This type of update is performed to convergence by repeating the process with different inputs in order to learn the weights of the computational graph.

11.3.5.1 Example of Computing Loss-to-Weight Derivatives

Consider the case of Figure 11.9 in which the loss function is defined by $L = \log[y(10)^2]$, and we wish to compute the derivative of the loss with respect to the weights w_2 and w_3. In such a case, the derivative of the loss with respect to the weights is given by the following:

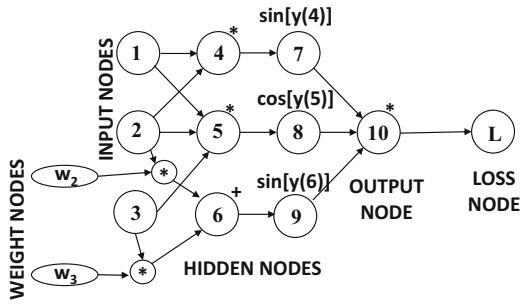


Figure 11.10: Converting loss-to-weight derivatives into node-to-node derivative computation based on Figure 11.9. Note the extra weight nodes and an extra loss node

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y(10)} \frac{\partial y(10)}{\partial y(6)} \frac{\partial y(6)}{\partial w_2} = \left[\frac{2}{y(10)}\right] [y(7) \cdot y(8) \cdot \cos[y(6)]] \, y(2)$$
$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial y(10)} \frac{\partial y(10)}{\partial y(6)} \frac{\partial y(6)}{\partial w_3} = \left[\frac{2}{y(10)}\right] [y(7) \cdot y(8) \cdot \cos[y(6)]] \, y(3)$$

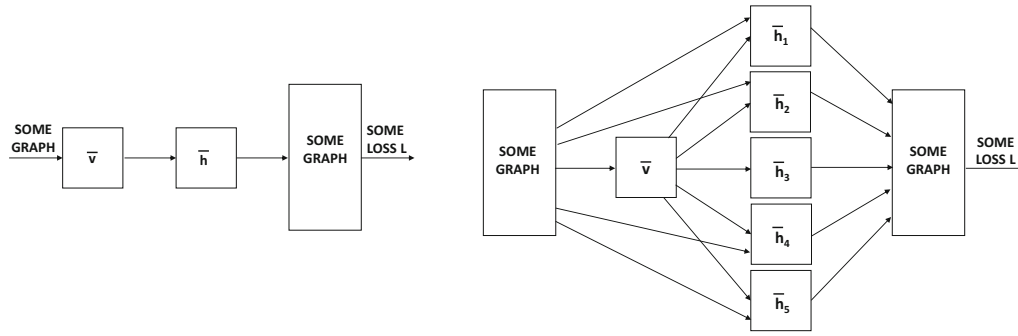
Note that the quantity $\frac{\partial y(10)}{\partial y(6)}$ has been obtained using the example in the previous section on node-to-node derivatives. In practice, these quantities are not computed algebraically. This is because the aforementioned algebraic expressions can be extremely awkward for large networks. Rather, for each numerical input set $\{y(1), y(2), y(3)\}$, one computes the different values of $y(i)$ in a forward phase. Subsequently, the derivatives of the loss with respect to each node variable (and incoming weights) are computed in a backwards phase. Again, these values are computed numerically for a specific input set $\{y(1), y(2), y(3)\}$. The numerical gradients can be used in order to update the weights for learning purposes.

11.3.6 Computational Graphs with Vector Variables

The previous section discuss the simple case in which each node of a computational graph contains a single scalar variable, whereas this section allows vector variables. In other words, the ith node contains the vector variable $\overline{y}_i$. Therefore, the **local** functions applied at the computational nodes are also vector-to-vector functions. For any node i, its local function uses an argument which corresponds to all the vector components of all its incoming nodes. From the input perspective of this local function, this situation is not too different from the previous case, where the argument is a vector corresponding to all the scalar inputs. However, the main difference is that the *output* of this function is a vector rather than a scalar. One example of such a vector-to-vector function is the *softmax* function (cf. Equation 4.64 of Chapter 4), which takes k real values as inputs and outputs k probabilities. In general, the number of inputs of the function need not be the same as the number of outputs in vector-to-vector functions. An important observation is the following:

> One can compute vector-to-vector derivatives in a computational graph using the vector-centric chain rule (cf. Section 4.6.3 of Chapter 4) rather than the scalar chain rule.

As discussed in Equation 4.19 of Chapter 4, a vector-to-vector derivative is a *matrix*. Consider two vectors, $\overline{v} = [v_1 \ldots v_d]^T$ and $\overline{h} = [h_1 \ldots h_m]^T$, which occur somewhere in the computational graph shown in Figure 11.11(a). There might be nodes incoming into $\overline{v}$ as



(a) Vector-centric graph with single path (b) Vector-centric graph with multiple paths

Figure 11.11: Examples of vector-centric computational graphs

well as a loss L computed in a later layer. Then, using the denominator layout of matrix calculus, the vector-to-vector derivative is the transpose of the *Jacobian* matrix, which was introduced in Chapter 4:

$$\frac{\partial \overline{h}}{\partial \overline{v}} = \text{Jacobian}(\overline{h}, \overline{v})^T$$

The (i, j)th entry of the above vector-to-vector derivative is simply $\frac{\partial h_j}{\partial v_i}$. Since $\overline{h}$ is an m-dimensional vector and $\overline{v}$ is a d-dimensional vector, the vector derivative is a $d \times m$ matrix. As discussed in Section 4.6.3 of Chapter 4, the chain rule over a single vector-centric path looks almost identical to the univariate chain rule over scalars, when one substitutes local partial derivatives with Jacobians. In other words, we can derive the following vector-valued chain rule for the single path of Figure 11.11(a):

$$\frac{\partial L}{\partial \overline{v}} = \underbrace{\frac{\partial \overline{h}}{\partial \overline{v}}}_{d \times m} \underbrace{\frac{\partial L}{\partial \overline{h}}}_{m \times 1} = \text{Jacobian}(\overline{h}, \overline{v})^T \frac{\partial L}{\partial \overline{h}}$$

Therefore, once the gradient of the loss is available with respect to a layer, it can be backpropagated by multiplying it with the transpose of a Jacobian! Here the *ordering of the matrices is important*, since matrix multiplication is not commutative.

The above provides the chain rule only for the case where the computational graph is a single path. What happens when the computational graph has an arbitrary structure? In such a case, we might have a situation where we have multiple nodes $\overline{h}_1 \ldots \overline{h}_s$ between node $\overline{v}$ and a network in later layers, as shown in Figure 11.11(b). Furthermore, there are connections between alternate layers, which are referred to as *skip connections*. Assume that the vector $\overline{h}_i$ has dimensionality m_i. In such a case, the partial derivative turns out to be a simple generalization of the previous case:

$$\frac{\partial L}{\partial \overline{v}} = \sum_{i=1}^{s} \underbrace{\frac{\partial \overline{h}_i}{\partial \overline{v}}}_{d \times m_i} \underbrace{\frac{\partial L}{\partial \overline{h}_i}}_{m_i \times 1} = \sum_{i=1}^{s} \text{Jacobian}(\overline{h}_i, \overline{v})^T \frac{\partial L}{\partial \overline{h}_i}$$

In most layered neural networks, we only have a single path and we rarely have to deal with the case of branches. Such branches might, however, arise in the case of neural networks with

skip connections [see Figures 11.11(b) and 11.13(b)]. However, even in complicated network architectures like Figures 11.11(b) and 11.13(b), *each node only has to worry about its local outgoing edges during backpropagation*. Therefore, we provide a very general vector-based algorithm below that can work even in the presence of skip connections.

Consider the case where we have p output nodes containing vector-valued variables, which have indices denoted by $t_1 \ldots t_p$, and the variables in it are $\overline{y}(t_1) \ldots \overline{y}(t_p)$. In such a case, the loss function L might be function of all the components in these vectors. Assume that the ith node contains a column vector of variables denoted by $\overline{y}(i)$. Furthermore, in the denominator layout of matrix calculus, each $\overline{\Delta}(i) = \frac{\partial L}{\partial \overline{y}(i)}$ is a column vector with dimensionality equal to that of $\overline{y}(i)$. It is this *vector* of loss derivatives that will be propagated backwards. The vector-centric algorithm for computing derivatives is as follows:

Initialize $\overline{\Delta}(t_k) = \frac{\partial L}{\partial \overline{y}(t_k)}$ for each output node t_k for $k \in \{1 \ldots p\}$;
repeat
  Select an unprocessed node i such that the values of $\overline{\Delta}(j)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $\overline{\Delta}(i) \Leftarrow \sum_{j \in A(i)} \text{Jacobian}(\overline{y}(j), \overline{y}(i))^T \overline{\Delta}(j)$;
until all nodes have been selected;
for the vector $\overline{w}_i$ of edges incoming to each node i **do** compute $\frac{\partial L}{\partial \overline{w}_i} = \frac{\partial \overline{y}(i)}{\partial \overline{w}_i} \overline{\Delta}(i)$;

In the final step of the above pseudocode, the derivative of vector $\overline{y}(i)$ with respect to the vector $\overline{w}_i$ is computed, which is itself the transpose of a Jacobian matrix. This final step converts a vector of partial derivatives with respect to node variables into a vector of partial derivatives with respect to weights incoming at a node.

11.4 Application: Backpropagation in Neural Networks

In this section, we will describe how the generic algorithm based on computational graphs can be used in order to perform the backpropagation algorithm in neural networks. The key idea is that specific variables in the neural networks need to be defined as nodes of the computational-graph abstraction. The same neural network can be represented by different types of computational graphs, depending on which variables in the neural network are used to create computational graph nodes. The precise methodology for performing the backpropagation updates depends heavily on this design choice.

Consider the case of a neural network that first applies a linear function with weights w_{ij} on its inputs to create the pre-activation value $a(i)$, and then applies the activation function $\Phi(\cdot)$ in order to create the output $h(i)$:

$$h(i) = \Phi(a(i))$$

The variables $h(i)$ and $a(i)$ are shown in Figure 11.12. In this case, it is noteworthy that there are several ways in which the computational graph can be created. For example, one might create a computational graph in which each node contains the post-activation value $h(i)$, and therefore we are implicitly setting $y(i) = h(i)$. A second choice is to create a computational graph in which each node contains the pre-activation variable $a(i)$ and therefore we are setting $y(i) = a(i)$. It is even possible to create a decoupled computational graph containing both $a(i)$ and $h(i)$; in the last case, the computational graph will have twice as many nodes as the neural network. In all these cases, a relatively straightforward special-case/simplification of the pseudocodes in the previous section can be used for learning the gradient:

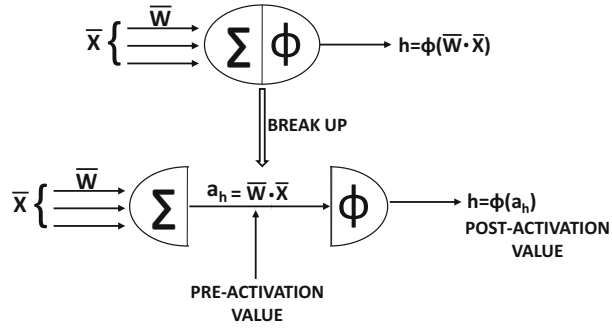


Figure 11.12: Pre- and post-activation values within a neuron

1. The post-activation value $y(i) = h(i)$ could represent the variable in the ith computational node in the graph. Therefore, each computational node in such a graph *first* applies the linear function, *and then* applies the activation function. The post-activation value is shown in Figure 11.12. In such a case, the value of $z(i, j) = \frac{\partial y(j)}{\partial y(i)} = \frac{\partial h(j)}{\partial h(i)}$ in the pseudocode of page 465 is $w_{ij}\Phi'_j$. Here, w_{ij} is the weight of the edge from i to j and $\Phi'_j = \frac{\partial \Phi(a(j))}{\partial a(j)}$ is the local derivative of the activation function at node j with respect to its argument. The value of each $\Delta(t_r)$ at output node t_r is simply the derivative of the loss function with respect to $h(t_r)$. The final derivative with respect to the weight w_{ji} (in the final line of the pseudocode on page 465) is equal to $\Delta(i)\frac{\partial h(i)}{\partial w_{ji}} = \Delta(i)h(j)\Phi'_i$.

2. The pre-activation value (after applying the linear function), which is denoted by $a(i)$, could represent the variable in each computational node i in the graph. Note the subtle distinction between the work performed in computational nodes and neural network nodes. Each *computational* node *first* applies the activation function to each of its inputs before applying a linear function, whereas these operations are performed in the reverse order in a neural network. The structure of the computational graph is roughly similar to the neural network, except that the first layer of computational nodes do not contain an activation. In such a case, the value of $z(i, j) = \frac{\partial y(j)}{\partial y(i)} = \frac{\partial a(j)}{\partial a(i)}$ in the pseudocode of page 465 is $\Phi'_i w_{ij}$. Note that $\Phi(a(i))$ is being differentiated with respect to its argument in this case, rather than $\Phi(a(j))$ as in the case of the post-activation variables. The value of the loss derivative with respect to the pre-activation variable $a(t_r)$ in the rth output node t_r needs to account for the fact that it is a pre-activation value, and therefore, we cannot directly use the loss derivative with respect to post-activation values. Rather the post-activation loss derivative needs to be *multiplied with the derivative* Φ'_{t_r} *of the activation function at that node*. The final derivative with respect to the weight w_{ji} (final line of pseudocode on page 465) is equal to $\Delta(i)\frac{\partial a(i)}{\partial w_{ji}} = \Delta(i)h(j)$.

The use of pre-activation variables for backpropagation is more common than the use of post-activation variables. Therefore, we present the backpropagation algorithm in a crisp pseudocode with the use of pre-activation variables. Let t_r be the index of the rth output node. Then, the backpropagation algorithm with pre-activation variables may be presented as follows:

Initialize $\Delta(t_r) = \frac{\partial L}{\partial y(t_r)} = \Phi'(a(t_r)) \frac{\partial L}{\partial h(t_r)}$ for each output node t_r with $r \in \{1 \ldots k\}$;
repeat
  Select an unprocessed node i such that the values of $\Delta(j)$ all of its outgoing nodes $j \in A(i)$ are available;
  Update $\Delta(i) \Leftarrow \Phi_i' \sum_{j \in A(i)} w_{ij} \Delta(j)$;
until all nodes have been selected;
for each edge (j, i) with weight w_{ji} **do** compute $\frac{\partial L}{\partial w_{ji}} = \Delta(i) h(j)$;

It is also possible to use both pre-activation and post-activation variables as separate nodes of the computational graph. In the next section, we will combine this approach with a vector-centric representation.

11.4.1 Derivatives of Common Activation Functions

It is evident from the discussion in the previous section that backpropagation requires the computation of derivatives of activation functions. Therefore, we discuss the computation of the derivatives of common activation functions in this section:

1. *Sigmoid activation:* The derivative of sigmoid activation is particularly simple, when it is expressed in terms of the *output* of the sigmoid, rather than the input. Let o be the output of the sigmoid function with argument v:

$$o = \frac{1}{1 + \exp(-v)} \tag{11.11}$$

   Then, one can write the derivative of the activation as follows:

$$\frac{\partial o}{\partial v} = \frac{\exp(-v)}{(1 + \exp(-v))^2} \tag{11.12}$$

   The key point is that this sigmoid can be written more conveniently in terms of the outputs:

$$\frac{\partial o}{\partial v} = o(1 - o) \tag{11.13}$$

   The derivative of the sigmoid is often used as a function of the output rather than the input.

2. *Tanh activation:* As in the case of the sigmoid activation, the tanh activation is often used as a function of the output o rather than the input v:

$$o = \frac{\exp(2v) - 1}{\exp(2v) + 1} \tag{11.14}$$

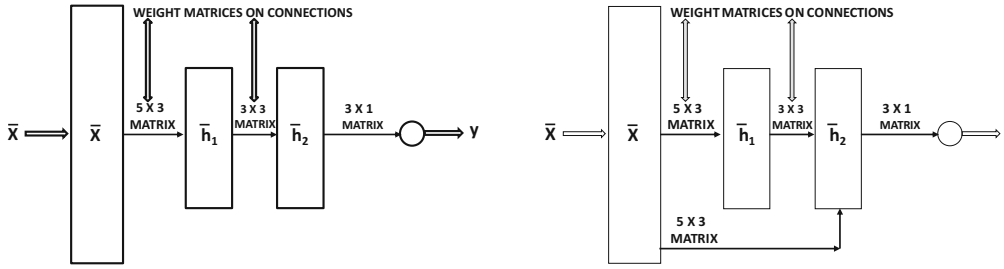
   One can then compute the derivative as follows:

$$\frac{\partial o}{\partial v} = \frac{4 \cdot \exp(2v)}{(\exp(2v) + 1)^2} \tag{11.15}$$

   One can also write this derivative in terms of the output o:

$$\frac{\partial o}{\partial v} = 1 - o^2 \tag{11.16}$$

3. *ReLU and hard tanh activations:* The ReLU takes on a partial derivative value of 1 for non-negative values of its argument, and 0, otherwise. The hard tanh function takes on a partial derivative value of 1 for values of the argument in $[-1, +1]$ and 0, otherwise.



(a) Vector-centric architecture with single path (Common)

(b) Arbitrary topology (Not common)

Figure 11.13: Most neural networks have a layer-wise architecture, and therefore the vector-centric architecture has a single path. However, if there are shortcuts across layers, it is possible for the topology of the vector-centric architecture to be arbitrary

11.4.2 Vector-Centric Backpropagation

As illustrated in Figure 11.3, any *layer-wise* neural architecture can be represented as a computational graph of vector variables *with a single path*. We repeat the vector-centric illustration of Figure 11.3(b) in Figure 11.13(a). Note that the architecture corresponds to a single path of vector variables, which can be further decoupled into linear layers and activation layers. Although it is possible for the neural network to have an arbitrary architecture (with paths of varying length), this situation is not so common. Some variations of this idea have been explored recently in the context of a specialized[1] neural network for image data, referred to as *ResNet* [6, 58]. We illustrate this situation in Figure 11.13(b), where there is a shortcut between alternate layers.

Since the layerwise situation of Figure 11.13(a) is more common, we discuss the approach used for performing backpropagation in this case. As discussed earlier, a node in a neural network performs a combination of a linear operation and a nonlinear activation function. In order to simplify the gradient evaluations, the linear computations and the activation computations are decoupled as separate "layers," and one separately backpropagates through the two layers. Therefore, one can create a neural network in which activation layers are alternately arranged with linear layers, as shown in Figure 11.14. Activation layers (usually) perform one-to-one, elementwise computations on the vector components with the activation function $\Phi(\cdot)$, whereas linear layers perform all-to-all computations by multiplying with the coefficient matrix W. Then, if $\overline{g}_i$ and $\overline{g}_{i+1}$ be the loss gradients in the ith and $(i+1)$th layers, and J_i be the Jacobian matrix between the ith and $(i+1)$th layers, the update is as follows: Let J be the matrix whose elements are J_{kr}. Then, it is easy to see that the backpropagation update from layer to layer can be written as follows:

$$\overline{g}_i = J_i^T \overline{g}_{i+1} \tag{11.17}$$

Writing backpropagation equations as matrix multiplications is often beneficial from an implementation-centric point of view, such as acceleration with Graphics Processor Units, which work particularly well with vector and matrix operations.

[1] *ResNet* is a convolutional neural network in which the structure of the layer is spatial, and the operations correspond to convolutions.

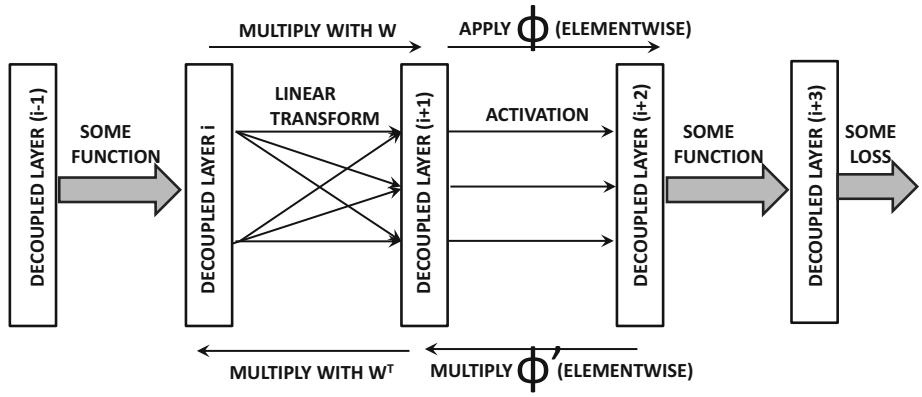


Figure 11.14: A decoupled view of backpropagation

First, the forward phase is performed on the inputs in order to compute the activations in each layer. Subsequently, the gradients are computed in the backwards phase. For each pair of matrix multiplication and activation function layers, the following forward and backward steps need to be performed:

1. Let $\overline{z}_i$ and $\overline{z}_{i+1}$ be the column vectors of activations in the forward direction when the matrix of linear transformations from the ith to the $(i+1)$th layer is denoted by W. Each element of the gradient $\overline{g}_i$ is the partial derivative of the loss function with respect to a hidden variable in the ith layer. Then, we have the following:

$$\overline{z}_{i+1} = W\overline{z}_i \quad \text{[Forward Propagation]}$$
$$\overline{g}_i = W^T\overline{g}_{i+1} \quad \text{[Backward Propagation]}$$

2. Now consider a situation where the activation function $\Phi(\cdot)$ is applied to each node in layer $(i+1)$ to obtain the activations in layer $(i+2)$. Then, we have the following:

$$\overline{z}_{i+2} = \Phi(\overline{z}_{i+1}) \quad \text{[Forward Propagation]}$$
$$\overline{g}_{i+1} = \overline{g}_{i+2} \odot \Phi'(\overline{z}_{i+1}) \quad \text{[Backward Propagation]}$$

   Here, $\Phi(\cdot)$ and its derivative $\Phi'(\cdot)$ are applied in element-wise fashion to vector arguments. The symbol $\odot$ indicates elementwise multiplication.

Note the extraordinary simplicity once the activation is decoupled from the matrix multiplication in a layer. The forward and backward computations are shown in Figure 11.14. Examples of different types of backpropagation updates for various forward functions are shown in Table 11.1. Therefore, the backward propagation operation is just like forward propagation. Given the vector of gradients in a layer, one only has to apply the operations shown in the final column of Table 11.1 to obtain the gradients of the loss with respect to the previous layer. In the table, the vector indicator function $I(\overline{x} > 0)$ is an *element-wise* indicator function that returns a binary vector of the sam size as $\overline{x}$; the ith output component is set to 1 when the ith component of $\overline{x}$ is larger than 0. The notation $\overline{1}$ denotes a column vector of 1s.

Table 11.1: Examples of different functions and their backpropagation updates between layers i and $(i+1)$. The hidden values and gradients in layer i are denoted by $\overline{z}_i$ and $\overline{g}_i$. Some of these computations use $I(\cdot)$ as the binary indicator function

Function	Type	Forward	Backward
Linear	Many-Many	$\overline{z}_{i+1} = W\overline{z}_i$	$\overline{g}_i = W^T\overline{g}_{i+1}$
Sigmoid	One-One	$\overline{z}_{i+1}$ =sigmoid$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (\overline{1} - \overline{z}_{i+1})$
Tanh	One-One	$\overline{z}_{i+1}$ =tanh$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (\overline{1} - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	One-One	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	One-One	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Many-One	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	Anything	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J_i^T\overline{g}_{i+1}$ J_i is Jacobian$(\overline{z}_{i+1}, \overline{z}_i)$

Initialization and Final Steps

The gradient of the final (output) layer is initialized to the vector of derivatives of the loss with respect to the various outputs in the output layer. This is generally a simple matter, since the loss in a neural network is generally a closed-form function of the outputs. Upon performing the backpropagation, one only obtains the loss-to-node derivatives but not the loss-to-weight derivatives. Note that the elements in $\overline{g}_i$ represent gradients of the loss with respect to the *activations* in the ith layer, and therefore an additional step is needed to compute gradients with respect to the *weights*. The gradient of the loss with respect to a weight between the pth unit of the $(i-1)$th layer and the qth unit of ith layer is obtained by multiplying the pth element of $\overline{z}_{i-1}$ with the qth element of $\overline{g}_i$. One can also achieve this goal using a vector-centric approach, by simply computing the *outer product* of $\overline{g}_i$ and $\overline{z}_{i-1}$. In other words, the entire matrix M of derivatives of the loss with respect to the weights in the $(i-1)$th layer and the ith layer is given by the following:

$$M = \overline{g}_i \overline{z}_{i-1}^T$$

Since M is given by the product of a column vector and a row vector of sizes equal to two successive layers, it is a matrix of exactly the same size as the weight matrix between the two layers. The (q, p)th element of M yields the derivative of the loss with respect to the weight between the pth element of $\overline{z}_{i-1}$ and qth element of $\overline{z}_i$.

11.4.3 Example of Vector-Centric Backpropagation

In order to explain vector-specific backpropagation, we will use an example in which the linear layers and activation layers have been decoupled. Figure 11.15 shows an example of a neural network with two computational layers, but they appear as four layers, since the activation layers have been decoupled as separated layers from the linear layers. The vector for the input layer is denoted by the 3-dimensional column vector $\overline{x}$, and the vectors for the computational layers are $\overline{h}_1$ (3-dimensional), $\overline{h}_2$ (3-dimensional), h_3 (1-dimensional), and output layer o (1-dimensional). The loss function is $L = -\log(o)$. These notations are annotated in Figure 11.15. The input vector $\overline{x}$ is $[2, 1, 2]^T$, and the weights of the edges in the two linear layers are annotated in Figure 11.15. Missing edges between $\overline{x}$ and $\overline{h}_1$

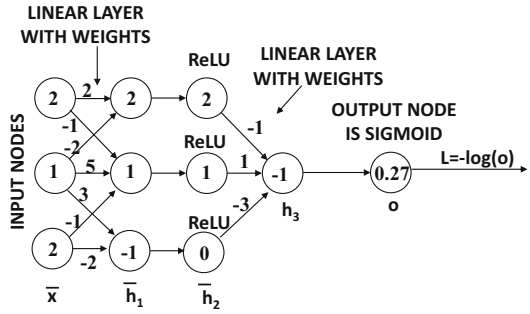


Figure 11.15: Example of decoupled neural network with vector layers $\overline{x}$, $\overline{h}_1$, $\overline{h}_2$, h_3, and o: variable values are shown within the nodes

are assumed to have zero weight. In the following, we will provide the details of both the forward and the backwards phase.

Forward phase: The first hidden layer $\overline{h}_1$ is related to the input vector $\overline{x}$ with the weight matrix W as $\overline{h}_1 = W\overline{x}$. We can reconstruct the weights matrix W and then compute $\overline{h}_1$ for forward propagation as follows:

$$W = \begin{bmatrix} 2 & -2 & 0 \\ -1 & 5 & -1 \\ 0 & 3 & -2 \end{bmatrix}; \quad \overline{h}_1 = W\overline{x} = \begin{bmatrix} 2 & -2 & 0 \\ -1 & 5 & -1 \\ 0 & 3 & -2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

The hidden layer $\overline{h}_2$ is obtained by applying the ReLU function in element-wise fashion to $\overline{h}_1$ during the forward phase. Therefore, we obtain the following:

$$\overline{h}_2 = \text{ReLU}(\overline{h}_1) = \text{ReLU} \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$$

Subsequently, the 1×3 weight matrix $W_2 = [-1, 1, -3]$ is used to transform the 3-dimensional vector $\overline{h}_2$ to the 1-dimensional "vector" h_3 as follows:

$$h_3 = W_2\overline{h}_2 = [-1, 1, -3] \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} = -1$$

The output o is obtained by applying the sigmoid function to h_3. In other words, we have the following:

$$o = \frac{1}{1 + \exp(-h_3)} = \frac{1}{1 + e} \approx 0.27$$

The point-specific loss is $L = -\log_e(0.27) \approx 1.3$.

Backwards phase: In the backward phase, we first start by initializing $\frac{\partial L}{\partial o}$ to $-1/o$, which is $-1/0.27$. Then, the 1-dimensional "gradient" g_3 of the hidden layer h_3 is obtained by using the backpropagation formula for the sigmoid function in Table 11.1:

$$g_3 = o(1 - o) \underbrace{\frac{\partial L}{\partial o}}_{-1/o} = o - 1 = 0.27 - 1 = -0.73 \tag{11.18}$$

The gradient $\overline{g}_2$ of the hidden layer $\overline{h}_2$ is obtained by multiplying g_3 with the transpose of the weight matrix $W_2 = [-1, 1, -3]$:

$$\overline{g}_2 = W_2^T g_3 = \begin{bmatrix} -1 \\ 1 \\ -3 \end{bmatrix} (-0.73) = \begin{bmatrix} 0.73 \\ -0.73 \\ 2.19 \end{bmatrix}$$

Based on the entry in Table 11.1 for the ReLU layer, the gradient $\overline{g}_2$ can be propagated backwards to $\overline{g}_1 = \frac{\partial L}{\partial \overline{h}_1}$ by copying the components of $\overline{g}_2$ to $\overline{g}_1$, when the corresponding components in $\overline{h}_1$ are positive; otherwise, the components of $\overline{g}_1$ are set to zero. Therefore, the gradient $\overline{g}_1 = \frac{\partial L}{\partial \overline{h}_1}$ can be obtained by simply copying the first and second components of $\overline{g}_2$ to the first and second components of $\overline{g}_1$, and setting the third component of $\overline{g}_1$ to 0. In other words, we have the following:

$$\overline{g}_1 = \begin{bmatrix} 0.73 \\ -0.73 \\ 0 \end{bmatrix}$$

Note that we can also compute the gradient $\overline{g}_0 = \frac{\partial L}{\partial \overline{x}}$ of the loss with respect to the input layer $\overline{x}$ by simply computing $\overline{g}_0 = W^T \overline{g}_1$. However, this is not really needed for computing loss-to-weight derivatives.

Computing loss-to-weight derivatives: So far, we have only shown how to compute loss-to-node derivatives in this particular example. These need to be converted to loss-to-weight derivatives with the additional step of multiplying with a hidden layer. Let M be the loss-to-weight derivatives for the weight matrix W between the two layers. Note that there is a one-to-one correspondence between the positions of the elements of M and W. Then, the matrix M is defined as follows:

$$M = \overline{g}_1 \overline{x}^T = \begin{bmatrix} 0.73 \\ -0.73 \\ 0 \end{bmatrix} [2, 1, 2] = \begin{bmatrix} 1.46 & 0.73 & 1.46 \\ -1.46 & -0.73 & -1.46 \\ 0 & 0 & 0 \end{bmatrix}$$

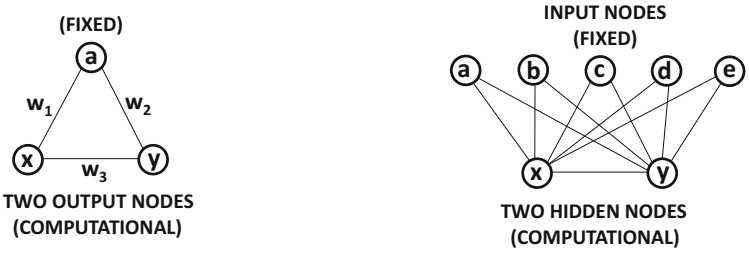
Similarly, one can compute the loss-to-weight derivative matrix M_2 for the 1×3 matrix W_2 between $\overline{h}_2$ and h_3:

$$M_2 = g_3 \overline{h}_2^T = (-0.73)[2, 1, 0] = [-1.46, -0.73, 0]$$

Note that the size of the matrix M_2 is identical to that of W_2, although the weights of the missing edges should not be updated.

11.5 A General View of Computational Graphs

Although the use of directed acyclic graphs on continuous-valued data is extremely common in machine learning (with neural networks being a prominent use case), other variations of such graphs exist. For example, it is possible for a computational graph to define probabilistic functions on edges, to have discrete-valued variables, and also to have cycles in the graph. In fact, the entire field of probabilistic graphical models is devoted to these types of computational graphs. Although the use of cycles in computational graphs is not common in feed-forward neural networks, they are extremely common in many advanced variations



(a) An undirected computational graph (b) More input states than hidden states

Figure 11.16: Examples of undirected computational graphs

of neural networks like Kohonen self-organizing maps, Hopfield networks, and Boltzmann machines. Furthermore, these neural networks use discrete and probabilistic data types as variables within their nodes (implicitly or explicitly).

Another important variation is the use of undirected computational graphs. In undirected computational graphs, each node computes a function of the variables in nodes incident on it, and there is no direction to the links. This is the only difference between an undirected computational graph and a directed computational graph. As in the case of directed computational graphs, one can define a loss function on the observed variables in the nodes. Examples of undirected computational graphs are shown in Figure 11.16. Some nodes are fixed (for observed data) whereas others are computational nodes. The computation can occur in both directions of an edge as long as the value in the node is not fixed externally.

It is harder to learn the parameters in undirected computational graphs, because the presence of cycles creates additional constraints on the values of variables in the nodes. In fact, it is not even necessary for there to be a set of variable values in nodes that satisfy all the functional constraints implied by the computational graph. For example, consider a computational graph with two nodes in which the variable of each node is obtained by adding 1 to the variable on the other node. It is impossible to find a pair of values in the two nodes that can satisfy both constraints (because both variable values cannot be larger than the other by 1). Therefore, one would have to be satisfied with a *best-fit* solution in many cases. This situation is different from a directed acyclic graph, where appropriate variables values can always be defined over all values of the inputs and parameters (as long as the function in each node is computable over its inputs).

Undirected computational graphs are often used in all types of unsupervised algorithms, because the cycles in these graphs help in relating other hidden nodes to the input nodes. For example, if the variables x and y are assumed to be hidden variables in Figure 11.16(b), this approach learns weights so that the two hidden variables correspond to the compressed representations of 5-dimensional data. The weights are often learned to minimize a loss function (or energy function) that rewards large weights when connected nodes are highly correlated in a positive way. For example, if variable x is heavily correlated with input a in a positive way, then the weight between these two nodes should be large. By learning these weights, one can compute the hidden representation of any 5-dimensional point by providing it as an input to the network.

The level of difficulty in learning the parameters of a computational graph is regulated by three characteristics of the graph. The first characteristic is the structure of the graph itself. It is generally much easier to learn the parameters of computational graphs without cycles

(which are always directed). The second characteristic is whether the variable in a node is continuous or discrete. It is much easier to optimize the parameters of a computational graph with continuous variables with the use of differential calculus. Finally, the function computed at a node of can be either probabilistic or deterministic. The parameters of deterministic computational graphs are almost always much easier to optimize with observed data. All these variations are important, and they arise in different types of machine learning applications. Some examples of different types of computational graphs in machine learning are as follows:

1. **Hopfield networks:** Hopfield networks are *undirected* computational graphs, in which the nodes always contain discrete, binary values. Since the graph is undirected, it contains cycles. The discrete nature of the variables makes the problem harder to optimize, because it precludes the use of simple techniques from calculus. In many cases, the optimal solutions to undirected graphs with discrete-valued variables are known[2] to be NP-hard [49]. For example, a special case of the Hopfield network can be used to solve the *traveling salesman problem*, which is known to be NP-hard. Most of the algorithms for such types of optimization problems are iterative heuristics.

2. **Probabilistic graphical models:** Probabilistic graphical models [74] are graphs representing the structural dependencies among random variables. Such dependencies may be either undirected or directed; directed dependencies may or may not contain cycles. The main distinguishing characteristic of a probabilistic graphical model from other types of graphical computational models is that the variables are probabilistic in nature. In other words, a variable in the computational graph corresponds to the *outcome* resulting from sampling from a *probability distributions that is conditioned on the variables in the incoming nodes*. Among all classes of models, probabilistic graphical models are the hardest to solve, and often require computationally intensive procedures like *Markov chain Monte Carlo sampling*. Interestingly, a generalization of Hopfield networks, referred to as *Boltzmann machines*, represents an important class of probabilistic graphical models.

3. **Kohonen self-organizing map:** A Kohonen self-organizing map uses a 2-dimensional *lattice-structured graph* on the hidden nodes. The activations on hidden nodes are analogous to the centroids in a k-means algorithm. This type of approach is a *competitive learning algorithm*. The lattice structure ensures that hidden nodes that are close to one another in the graph have similar values. As a result, by associating data points with their closest hidden nodes, one is able to obtain a 2-dimensional visualization of the data.

Table 11.2 shows several variations of the computational graph paradigm in machine learning, and their specific properties. It is evident that the methodology used for a particular problem is highly dependent on the structure of the computational graph, its variables, and the nature of the node-specific function. We refer the reader to [6] for the neural architectures of the basic machine learning models discussed in this book (like linear regression, logistic regression, matrix factorization, and SVMs).

[2]When a problem is NP-hard, it means that a polynomial-time algorithm for such a problem is not known (although it is unknown whether one exists). More specifically, finding a polynomial-time algorithm for such a problem would automatically provide a polynomial-time algorithm for thousands of related problems for which no one has been able to find a polynomial-time algorithm. The inability to find a polynomial-time problem for a large class of related problems is generally assumed to be evidence of the fact that the entire set of problems is hard to solve, and a polynomial time algorithm *probably* does not exist for any of them.

Table 11.2: Types of computational graphs for different machine learning problems. The properties of the computational graph vary according to the application at hand

Model	Cycles?	Variable	Function	Methodology
SVM Logistic Regression Linear Regression SVD Matrix Factorization	No	Continuous	Deterministic	Gradient Descent
Feedforward Neural Networks	No	Continuous	Deterministic	Gradient Descent
Kohonen Map	Yes	Continuous	Deterministic	Gradient Descent
Hopfield Networks	Yes (Undirected)	Discrete (Binary)	Deterministic	Iterative (Hebbian Rule)
Boltzmann Machines	Yes (Undirected)	Discrete (Binary)	Probabilistic	Monte Carlo Sampling + Iterative (Hebbian)
Probabilistic Graphical Models	Varies	Varies	Probabilistic (largely)	Varies

11.6 Summary

This chapter introduces the basics of computational graphs for machine learning applications. Computational graphs often have parameters associated with their edges, which need to be learned. Learning the parameters of a computational graph from observed data provides a route to learning a function from observed data (whether it can be expressed in closed form or not). The most commonly used type of computational graph is a directed acyclic graph. Traditional neural networks represent a class of models that is a special case of this type of graph. However, other types of undirected and cyclic graphs are used to represent other models like Hopfield networks and restricted Boltzmann machines.

11.7 Further Reading

Computational graphs represent a fundamental way of defining the computations associated with many machine learning models such as neural networks or probabilistic models. Detailed discussions of neural networks may be found in [6, 53], whereas detailed discussions of probabilistic graphical models may be found in [74]. Automatic differentiation in computational graphs has historically been used extensively in control theory [26, 71]. The backpropagation algorithm was first proposed in the context of neural networks by Werbos [131], although it was forgotten. Eventually, the algorithm was popularized in the paper by Rumelhart *et al.* [110]. The Hopfield network and the Boltzmann machine are both discussed in [6]. A discussion of Kohonen self-organizing maps may also be found in [6].

11.8 Exercises

1. Problem 11.2.2 proposes a loss function for the L_1-SVM in the context of a computational graph. How would you change this loss function, so that the same computational graph results in an L_2-SVM?

2. Repeat Exercise 1 with the changed setting that you want to simulate Widrow-Hoff learning (least-squares classification) with the same computational graph. What will be the loss function associated with the single output node?

3. The book discusses a vector-centric view of backpropagation in which backpropagation in linear layers can be implemented with matrix-to-vector multiplications. Discuss how you can deal with *batches* of training instances at a time (i.e., mini-batch stochastic gradient descent) by using matrix-to-matrix multiplications.

4. Let $f(x)$ be defined as follows:

$$f(x) = \sin(x) + \cos(x)$$

Consider the the function $f(f(f(f(x))))$. Write this function in closed form to obtain an appreciation of the awkwardly long function. Evaluate the derivative of this function at $x = \pi/3$ radians by using a computational graph abstraction.

5. Suppose that you have a computational graph with the constraint that specific sets of weights are always constrained to be at the same value. Discuss how you can compute the derivative of the loss function with respect to these weights. [Note that this trick is used frequently in the neural network literature to handle shared weights.]

6. Consider a computational graph in which you are told that the variables on the edges satisfy k linear equality constraints. Discuss how you would train the weights of such a graph. How would your answer change, if the variables satisfied box constraints. [The reader is advised to refer to the chapter on constrained optimization for answering this question.]

7. Discuss why the dynamic programming algorithm for computing the gradients will not work in the case where the computational graph contains cycles.

8. Consider the neural architecture with connections between alternate layers, as shown in Figure 11.13(b). Suppose that the recurrence equations of this neural network are as follows:

$$\overline{h}_1 = \text{ReLU}(W_1 \overline{x})$$
$$\overline{h}_2 = \text{ReLU}(W_2 \overline{x} + W_3 \overline{h}_1)$$
$$y = W_4 \overline{h}_2$$

Here, W_1, W_2, W_3, and W_4 are matrices of appropriate size. Use the vector-centric backpropagation algorithm to derive the expressions for $\frac{\partial y}{\partial \overline{h}_2}$, $\frac{\partial y}{\partial \overline{h}_1}$, and $\frac{\partial y}{\partial \overline{x}}$ in terms of the matrices and activation values in intermediate layers.

9. Consider a neural network that has hidden layers $\overline{h}_1 \ldots \overline{h}_t$, inputs $\overline{x}_1 \ldots \overline{x}_t$ into each layer, and outputs $\overline{o}$ from the final layer $\overline{h}_t$. The recurrence equation for the pth layer is as follows:

$$\overline{o} = U \overline{h}_t$$
$$\overline{h}_p = \tanh(W \overline{h}_{p-1} + V \overline{x}_p) \quad \forall p \in \{1 \ldots t\}$$

The vector output $\overline{o}$ has dimensionality k, each $\overline{h}_p$ has dimensionality m, and each $\overline{x}_p$ has dimensionality d. The "tanh" function is applied in element-wise fashion. The

notations U, V, and W are matrices of sizes $k \times m$, $m \times d$, and $m \times m$, respectively. The vector $\overline{h}_0$ is set to the zero vector. Start by drawing a (vectored) computational graph for this system. Show that node-to-node backpropagation uses the following recurrence:

$$\frac{\partial \overline{o}}{\partial \overline{h}_t} = U^T$$
$$\frac{\partial \overline{o}}{\partial \overline{h}_{p-1}} = W^T \Delta_{p-1} \frac{\partial \overline{o}}{\partial \overline{h}_p} \quad \forall p \in \{2 \ldots t\}$$

Here, Δ_p is a diagonal matrix in which the diagonal entries contain the components of the vector $\overline{1} - \overline{h}_p \odot \overline{h}_p$. What you have just derived contains the node-to-node backpropagation equations of a recurrent neural network. What is the size of each matrix $\frac{\partial \overline{o}}{\partial \overline{h}_p}$?

10. Show that if we use the loss function $L(\overline{o})$ in Exercise 9, then the loss-to-node gradient can be computed for the final layer $\overline{h}_t$ as follows:

$$\frac{\partial L(\overline{o})}{\partial \overline{h}_t} = U^T \frac{\partial L(\overline{o})}{\partial \overline{o}}$$

The updates in earlier layers remain similar to Exercise 9, except that each $\overline{o}$ is replaced by $L(\overline{o})$. What is the size of each matrix $\frac{\partial L(\overline{o})}{\partial \overline{h}_p}$?

11. Suppose that the output structure of the neural network in Exercise 9 is changed so that there are k-dimensional outputs $\overline{o}_1 \ldots \overline{o}_t$ in each layer, and the overall loss is $L = \sum_{i=1}^{t} L(\overline{o}_i)$. The output recurrence is $\overline{o}_p = U\overline{h}_p$. All other recurrences remain the same. Show that the backpropagation recurrence of the hidden layers changes as follows:

$$\frac{\partial L}{\partial \overline{h}_t} = U^T \frac{\partial L(\overline{o}_t)}{\partial \overline{o}_t}$$
$$\frac{\partial L}{\partial \overline{h}_{p-1}} = W^T \Delta_{p-1} \frac{\partial L}{\partial \overline{h}_p} + U^T \frac{\partial L(\overline{o}_{p-1})}{\partial \overline{o}_{p-1}} \quad \forall p \in \{2 \ldots t\}$$

12. For Exercise 11, show the following loss-to-weight derivatives:

$$\frac{\partial L}{\partial U} = \sum_{p=1}^{t} \frac{\partial L(\overline{o}_p)}{\partial \overline{o}_p} \overline{h}_p^T, \quad \frac{\partial L}{\partial W} = \sum_{p=2}^{t} \Delta_{p-1} \frac{\partial L}{\partial \overline{h}_p} \overline{h}_{p-1}^T, \quad \frac{\partial L}{\partial V} = \sum_{p=1}^{t} \Delta_p \frac{\partial L}{\partial \overline{h}_p} \overline{x}_p^T$$

What are the sizes and ranks of these matrices?

13. Consider a neural network in which a vectored node $\overline{v}$ feeds into two distinct vectored nodes $\overline{h}_1$ and $\overline{h}_2$ computing different functions. The functions computed at the nodes are $\overline{h}_1 = \text{ReLU}(W_1\overline{v})$ and $\overline{h}_2 = \text{sigmoid}(W_2\overline{v})$. We do not know anything about the values of the variables in other parts of the network, but we know that $\overline{h}_1 = [2, -1, 3]^T$ and $\overline{h}_2 = [0.2, 0.5, 0.3]^T$, that are connected to the node $\overline{v} = [2, 3, 5, 1]^T$. Furthermore, the loss gradients are $\frac{\partial L}{\partial \overline{h}_1} = [-2, 1, 4]^T$ and $\frac{\partial L}{\partial \overline{h}_2} = [1, 3, -2]^T$, respectively. Show that the backpropagated loss gradient $\frac{\partial L}{\partial \overline{v}}$ can be computed in terms of W_1 and W_2 as follows:

$$\frac{\partial L}{\partial \overline{v}} = W_1^T \begin{bmatrix} -2 \\ 0 \\ 4 \end{bmatrix} + W_2^T \begin{bmatrix} 0.16 \\ 0.75 \\ -0.42 \end{bmatrix}$$

What are the sizes of W_1, W_2, and $\frac{\partial L}{\partial \overline{v}}$?

14. **Forward Mode Differentiation:** The backpropagation algorithm needs to compute node-to-node derivatives of *output* nodes with respect to all other nodes, and therefore computing gradients in the backwards direction makes sense. Consequently, the pseudocode on page 460 propagates gradients in the backward direction. However, consider the case where we want to compute the node-to-node derivatives of all nodes with respect to *source* (input) nodes $s_1 \ldots s_k$. In other words, we want to compute $\frac{\partial x}{\partial s_i}$ for each non-input node variable x and each input node s_i in the network. Propose a variation of the pseudocode of page 460 that computes node-to-node gradients in the forward direction.

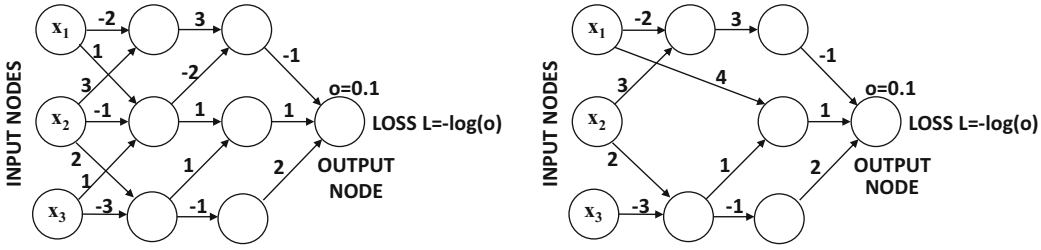
15. **All-pairs node-to-node derivatives:** Let $y(i)$ be the variable in node i in a directed acyclic computational graph containing n nodes and m edges. Consider the case where one wants to compute $S(i, j) = \frac{\partial y(j)}{\partial y(i)}$ for all pairs of nodes in a computational graph, so that at least one directed path exists from node i to node j. Propose an algorithm for all-pairs derivative computation that requires at most $O(n^2 m)$ time. [Hint: The pathwise aggregation lemma is helpful. First compute $S(i, j, t)$, which is the portion of $S(i, j)$ in the lemma belonging to paths of length exactly t. How can $S(i, k, t + 1)$ be expressed in terms of the different $S(i, j, t)$?]

16. Use the pathwise aggregation lemma to compute the derivative of $y(10)$ with respect to each of $y(1)$, $y(2)$, and $y(3)$ as an algebraic expression (cf. Figure 11.9). You should get the same derivative as obtained using the backpropagation algorithm in the text of the chapter.

17. Consider the computational graph of Figure 11.8. For a particular numerical input $x = a$, you find the unusual situation that the value $\frac{\partial y(j)}{\partial y(i)}$ is 0.3 for each and every edge (i, j) in the network. Compute the numerical value of the partial derivative of the output with respect to the input x (at $x = a$). Show the computations using both the pathwise aggregation lemma and the backpropagation algorithm.

18. Consider the computational graph of Figure 11.8. The upper node in each layer computes $\sin(x + y)$ and the lower node in each layer computes $\cos(x + y)$ with respect to its two inputs. For the first hidden layer, there is only a single input x, and therefore the values $\sin(x)$ and $\cos(x)$ are computed. The final output node computes the product of its two inputs. The single input x is 1 radian. Compute the numerical value of the partial derivative of the output with respect to the input x (at $x = 1$ radian). Show the computations using both the pathwise aggregation lemma and the backpropagation algorithm.

19. **Matrix factorization with neural networks:** Consider a neural network containing an input layer, a hidden layer, and an output layer. The number of outputs is equal to the number of inputs d. Each output value corresponds to an input value, and the loss function is the sum of squared differences between the outputs and their corresponding inputs. The number of nodes k in the hidden layer is much less than d. The d-dimensional rows of a data matrix D are fed one by one to train this neural network. Discuss why this model is identical to that of unconstrained matrix factorization of rank-k. Interpret the weights and the activations in the hidden layer in the context of matrix factorization. You may assume that the matrix D has full column rank. Define weight matrix and data matrix notations as convenient.



(a) Exercise 21 (b) Exercise 22

Figure 11.17: Computational graphs for Exercises 21 and 22

20. SVD with neural networks: In the previous exercise, unconstrained matrix factorization finds the same k-dimensional subspace as SVD. However, it does not find an orthonormal basis in general like SVD (see Chapter 8). Provide an iterative training method for the computational graph of the previous section by gradually increasing the value of k so that an orthonormal basis is found.

21. Consider the computational graph shown in Figure 11.17(a), in which the local derivative $\frac{\partial y(j)}{\partial y(i)}$ is shown for each edge (i, j), where $y(k)$ denotes the activation of node k. The output o is 0.1, and the loss L is given by $-\log(o)$. Compute the value of $\frac{\partial L}{\partial x_i}$ for each input x_i using both the path-wise aggregation lemma, and the backpropagation algorithm.

22. Consider the computational graph shown in Figure 11.17(b), in which the local derivative $\frac{\partial y(j)}{\partial y(i)}$ is shown for each edge (i, j), where $y(k)$ denotes the activation of node k. The output o is 0.1, and the loss L is given by $-\log(o)$. Compute the value of $\frac{\partial L}{\partial x_i}$ for each input x_i using both the path-wise aggregation lemma, and the backpropagation algorithm.

23. Convert the weighted computational graph of Figure 11.2 into an unweighted graph by defining additional nodes containing $w_1 \ldots w_5$ along with appropriately defined hidden nodes.

24. Multinomial logistic regression with neural networks: Propose a neural network architecture using the softmax activation function and an appropriate loss function that can perform multinomial logistic regression. You may refer to Chapter 4 for details of multinomial logistic regression.

25. Weston-Watkins SVM with neural networks: Propose a neural network architecture and an appropriate loss function that is equivalent to the Weston-Watkins SVM. You may refer to Chapter 4 for details of the Weston-Watkins SVM.