5 Differentiation with Automata

Modern deep learning relies on the fact that core functions are differentiable (or sub-differentiable). This is important to any type of gradient-based numerical optimization, of which the most commonly used are variations of stochastic gradient descent. The parameters, θ, of the model are specified in tensors. The objective function, $f(\theta, \mathbf{x}, \mathbf{y})$, is a function of the parameters and the input data $\mathbf{x}$ and $\mathbf{y}$. The parameters can then be optimized to improve the objective with variations of a simple update rule:

$$\theta_t = \theta_{t-1} + \alpha \nabla_\theta f(\theta_t, \mathbf{x}, \mathbf{y}),$$

where α is the learning rate, and ∇_θ is the gradient operator which we describe in more detail below. This update applies equally well to parameters in graphs as it does to tensors. The main challenge is computing gradients, the subject of this section.

5.1 Derivatives

Many operations used with vectors, matrices, and n-dimensional tensors are differentiable. This means we can compute the change in any of the output elements with respect to an infinitesimal change in any of the input elements. For example, consider a vector $\mathbf{z} = f(\mathbf{x}, \mathbf{y})$ which is the output of a function of two vectors $\mathbf{x}$ and $\mathbf{y}$. The Jacobian of $\mathbf{z}$ with respect to $\mathbf{x}$ is the matrix of partial derivatives with entries $\frac{\partial \mathbf{z}_i}{\partial \mathbf{x}_j}$. The gradient is defined as the tensor of partial derivatives of a scalar function. So if $f(\mathbf{x}) \in \mathbb{R}$ is a scalar function, then the gradient is:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_n} \right]^\top.$$

In the same way, we can compute partial derivatives of the arc weights of an output graph for a given operation with respect to the arc weights of any of the input graphs. Take the concatenation operation as an example. Suppose we are given two graphs, $\mathcal{A}$ and $\mathcal{B}$, and we construct the concatenated graph $\mathcal{C} = \mathcal{A}\mathcal{B}$ as in figure 5.1.

For each of the arc weights C_i in the concatenated graph $\mathcal{C}$, we can compute the partial derivative with respect to the arc weights of $\mathcal{A}$ and $\mathcal{B}$. For any arc in $\mathcal{C}$, it

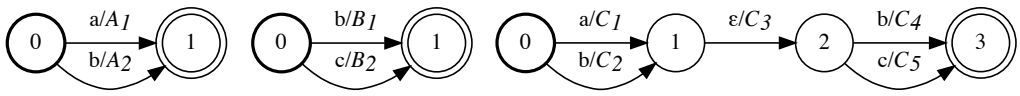

Figure 5.1: The concatenation of the graphs $\mathcal{A}$ (left) and $\mathcal{B}$ (middle) produces $\mathcal{C}$ (right). For each graph the arc weights are shown as variables on the edges.

either has a corresponding arc in $\mathcal{A}$ or $\mathcal{B}$ from which it gets its weight, or it has a weight of zero. The partial derivative of an output arc weight C_i with respect to an input arc weight A_j or B_j is 1 if the two arcs correspond and 0 otherwise. For example, in the graphs in figure 5.1 we have:

$$\frac{\partial C_1}{\partial A_1} = 1, \quad \frac{\partial C_2}{\partial A_2} = 1, \quad \frac{\partial C_4}{\partial B_1} = 1, \quad \text{and} \quad \frac{\partial C_5}{\partial B_2} = 1.$$

The remaining partial derivatives are all 0. For example, for C_1 we have:

$$\frac{\partial C_1}{\partial A_2} = 0, \quad \frac{\partial C_1}{\partial B_1} = 0, \quad \text{and} \quad \frac{\partial C_1}{\partial B_2} = 0.$$

In the following, I use the notation $\frac{\partial \mathcal{C}}{\partial \mathcal{A}}$ to generalize the Jacobian to graphs. This Jacobian is a data structure which contains the partial derivatives $\frac{\partial C_i}{\partial A_j}$ for all arc weights C_i in $\mathcal{C}$ and A_j in $\mathcal{A}$. Another way to view this Jacobian is as a set of graphs $\frac{\partial C_i}{\partial \mathcal{A}}$ indexed by i which are the same size as $\mathcal{A}$. Alternatively we can view the Jacobian as a set of graphs $\frac{\partial \mathcal{C}}{\partial A_j}$ indexed by j which are the same size as $\mathcal{C}$. This is analogous to viewing the Jacobian of a vector-valued function either as a set of columns or a set of rows.

Example 5.1. Compute the partial derivatives of the arc weights of the closure of the graph $\mathcal{A}$ from figure 5.1 with respect to the input arc weights. The closure, $\mathcal{A}^*$, is in figure 5.2.

The non-zero partial derivatives are:

$$\frac{\partial w_2}{\partial A_1} = 1 \quad \text{and} \quad \frac{\partial w_3}{\partial A_2} = 1.$$

The remaining partial derivatives are zero:

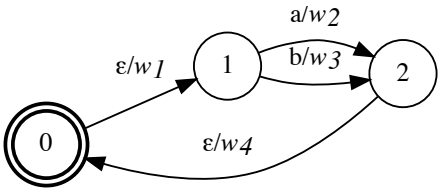

Figure 5.2: The closure of the graph $\mathcal{A}$ from figure 5.1. The weights are denoted by the variables w_i.

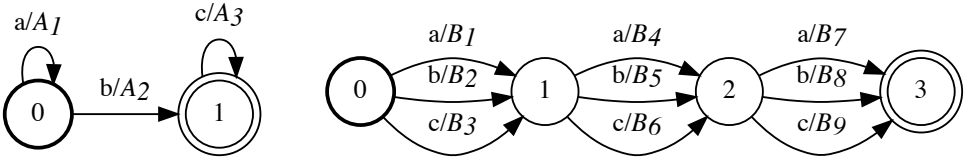

Figure 5.3: We would like to compute the derivative of the intersected graph's arc weights with respect to the arc weights in the two input acceptors $\mathcal{A}$ and $\mathcal{B}$. The arc weights are labeled with the variable names A_j and B_k.

$$\frac{\partial w_1}{\partial A_1} = 0, \quad \frac{\partial w_1}{\partial A_2} = 0, \quad \frac{\partial w_2}{\partial A_2} = 0, \quad \frac{\partial w_3}{\partial A_1} = 0, \quad \frac{\partial w_4}{\partial A_1} = 0, \quad \text{and} \quad \frac{\partial w_4}{\partial A_2} = 0.$$

■

Example 5.2. Compute the partial derivatives of the intersected automata weights w_i with respect to the input arc weights A_j for graph $\mathcal{A}$ and B_k for graph $\mathcal{B}$ shown in figure 5.3.

The partial derivative $\frac{\partial w_i}{\partial A_j}$ is 1 if the weight w_i came from A_j and zero otherwise. The derivatives with a value of 1 for graph $\mathcal{A}$ are:

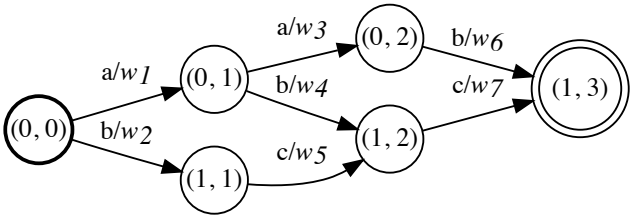

Figure 5.4: The intersected graph of the two acceptors $\mathcal{A}$ and $\mathcal{B}$ in figure 5.3. The weights are denoted as variables w_i on the edges.

$$\frac{\partial w_1}{\partial A_1}, \frac{\partial w_2}{\partial A_2}, \frac{\partial w_3}{\partial A_1}, \frac{\partial w_4}{\partial A_2}, \frac{\partial w_5}{\partial A_3}, \frac{\partial w_6}{\partial A_2}, \text{ and } \frac{\partial w_7}{\partial A_3}.$$

The derivatives with a value of 1 for graph $\mathcal{B}$ are:

$$\frac{\partial w_1}{\partial B_1}, \frac{\partial w_2}{\partial B_2}, \frac{\partial w_3}{\partial B_4}, \frac{\partial w_4}{\partial B_5}, \frac{\partial w_5}{\partial B_6}, \frac{\partial w_6}{\partial B_8}, \text{ and } \frac{\partial w_7}{\partial B_9}.$$

The remaining derivatives for both graphs are zero. ■

5.2 Automatic Differentiation

In the previous section we saw how to compute derivatives for some common automata operations. Automatic differentiation greatly simplifies the process of computing derivatives for arbitrary compositions of operations. In this section, we will discuss *reverse-mode* automatic differentiation at a high-level.

Reverse-mode automatic differentiation proceeds in two steps. First, a forward pass computes all of the operations. Then, a backward pass computes the gradients. During the forward pass the composition of operations is stored in a computation graph (not to be confused with an automata). Data and metadata are also cached during the forward pass to make the gradient computation more efficient. There is often a trade-off between memory and compute in that we can save more intermediate data to reduce the computation required during but increase the memory required during the backward pass.

Consider the graph equation:

$$\text{LSE}\left[((\mathcal{A}_1 + \mathcal{A}_2) \circ \mathcal{X}^*)\right]. \tag{4}$$

This equation can be represented in the computation graph in figure 5.5.

The leaves of the computation graph (the solid circular nodes with no incoming arrows) are either parameter graphs or input data graphs. In this case, let's assume $\mathcal{A}_1$ and $\mathcal{A}_2$ are the parameter graphs, and $\mathcal{X}$ is the graph of input data. The square nodes are operations, and they are always followed by output graphs, which are dashed circular nodes.

During the backward pass, the gradients are computed from the output ($\mathcal{G}_4$ in figure 5.5) following the arrows in the computation graph backwards. A graph can only compute its gradient once all of the graphs downstream of it have had

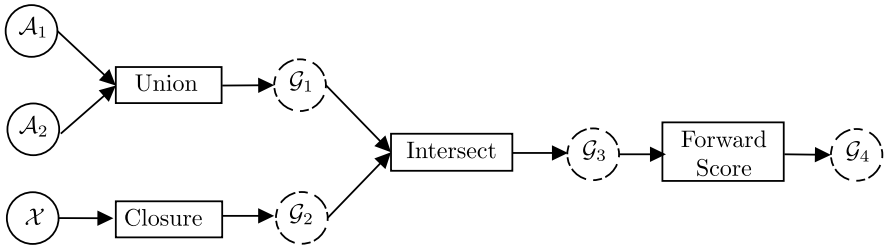

Figure 5.5: An example of a compute graph for equation 4. The solid circular nodes are leaves. The graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ are parameters, and $\mathcal{X}$ is input data. The rectangular nodes are operations. The dashed circular nodes are graphs computed as the result of an operation.

their gradients computed. Thus the backward pass must be done as a reverse topological traversal of the computation graph.

For example, assume the gradient of the output $\mathcal{G}_4$ with respect to graph $\mathcal{G}_1$ has been computed. This gradient $\frac{\partial \mathcal{G}_4}{\partial \mathcal{G}_1}$ is then used to compute $\frac{\partial \mathcal{G}_4}{\partial \mathcal{A}_1}$ and $\frac{\partial \mathcal{G}_4}{\partial \mathcal{A}_2}$. Assuming we know how to differentiate $\mathcal{G}_1$ with respect to $\mathcal{A}_1$ and $\mathcal{A}_2$, then we can compute the desired gradients essentially using the chain rule. Assume g_i are the arc weights of $\mathcal{G}_1$ and a_j are the arc weights of $\mathcal{A}_1$, then the chain rule gives:

$$\frac{\partial \mathcal{G}_4}{\partial a_j} = \sum_i \frac{\partial \mathcal{G}_4}{\partial g_i} \frac{\partial g_i}{\partial a_j}.$$

This is done recursively at every node in the computation graph until the gradients for all the leaf graphs are available.

At a high-level any implementation of reverse-mode automatic differentiation is the same. However, implementations often differ in the details. One simple approach is to have every output graph record the input graphs from which it was generated as well as a gradient computation function. The input graphs and gradient computation function (and any other metadata) can be set during the forward pass by the operation itself.

For example, after the execution of union, the data structure which holds the output graph $\mathcal{G}_1$ could also hold pointers to the inputs $\mathcal{A}_1$ and $\mathcal{A}_2$ and a pointer to the gradient computation function for union. A simplified example of what this data structure might look like is shown in figure 5.6.

Once the gradient for $\mathcal{G}_1$ is available, the union's gradient function is called with

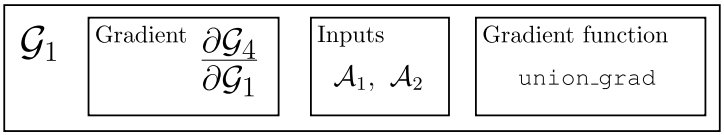

Figure 5.6: The data structure which holds the graph $\mathcal{G}_1$ as well as the data needed to compute the gradient of its inputs. In this case $\mathcal{G}_1$ was the output of a union of $\mathcal{A}_1$ and $\mathcal{A}_2$.

the inputs $\mathcal{A}_1$, $\mathcal{A}_2$, and $\frac{\partial \mathcal{G}_4}{\partial \mathcal{G}_1}$. For $\mathcal{A}_1$, the union gradient function will compute $\frac{\partial \mathcal{G}_1}{\partial \mathcal{A}_1}$ and use the chain rule as described above to assemble the desired gradient $\frac{\partial \mathcal{G}_4}{\partial \mathcal{A}_1}$. It will do the same for $\mathcal{A}_2$.