

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

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

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

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

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

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

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