Deep Learning
Ch6.5 Back-Propagation and Other Differentiation Algorithms

Authors: Ian Goodfellow, Yoshua Bengio, Aaron Courville

Reporter: Bo-Zen Xiao

2017-06-10

Contents

6.5 Back-Propagation and Other Differentiation Algorithms

- When we use a feedforward neural network to accept an input x and produce an output $\hat{y}$, information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{y}$. This is called *forward propagation*. During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$.
- The *back-propagation* algorithm, often simply called *backprop* , allows the information from the cost to then flow backwards through the network, in order to compute the gradient.
- Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

6.5 Back-Propagation and Other Differentiation Algorithms

- The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.
- Furthermore, back-propagation is often misunderstood as being specific to multilayer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined).
- Specifically, we will describe how to compute the gradient $\nabla_x f(x, y)$ for an arbitrary function f, where x is a set of variables whose derivatives are desired, and y is an additional set of variables that are inputs to the function but whose derivatives are not required.

6.5 Back-Propagation and Other Differentiation Algorithms

- In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_\theta J(\theta)$. Many machine learning tasks involve computing other derivatives, either as part of the learning process, or to analyze the learned model.
- The backpropagation algorithm can be applied to these tasks as well, and is not restricted to computing the gradient of the cost function with respect to the parameters.
- The idea of computing derivatives by propagating information th rough a network is very general, and can be used to compute values such as the Jacobian of a function f with multiple outputs. We restrict our description here to the most commonly used case where f has a single output.

6.5.1 Computational Graphs

- So far we have discussed neural networks with a relatively informal graph language. To describe the back-propagation algorithm more precisely, it is helpful to have a more precise *computational graph* language.
- Many ways of formalizing computation as graphs are possible.
  Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.
- To formalize our graphs, we also need to introduce the idea of an *operation*. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together.

6.5.1 Computational Graphs

- Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector.
- Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case in our description because it introduces many extra details that are not important to conceptual understanding.
- If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y.
- We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

6.5.1 Computational Graphs

Figure : (a) The graph using the$\times$ operation to compute $z = xy$. (b) The graph for the logistic regression prediction $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w} + b)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i-th such variable $\mathbf{u}^{(i)}$.

6.5.1 Computational Graphs

Figure : (c) The computational graph for the expression $\mathbf{H} = \max\{0, \mathbf{XW} + \mathbf{b}\}$, which computes a design matrix of rectified linear unit activations $\mathbf{H}$ given a design matrix containing a minibatch of inputs $\mathbf{X}$. (d) Examples a—c applied at most one operation to each variable, but it is possible to apply more than one operation. Here we show a computation graph that applies more than one operation to the weights $\mathbf{w}$ of a linear regression model. The weights are used to make the both the prediction $\hat{y}$ and the weight decay penalty $\lambda \sum_i w_i^2$.

6.5.2 Chain Rule of Calculus

- The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.
- Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

6.5.2 Chain Rule of Calculus

- We can generalize this beyond the scalar case. Suppose that $x \in R^m$, $y \in R^n$, g maps from R^m to R^n, and f maps from R^n to R . If $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

- In vector notation, this may be equivalently written as

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

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

6.5.2 Chain Rule of Calculus

- From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.
- Usually we do not apply the back-propagation algorithm merely to vectors, but rather to tensors of arbitrary dimensionality. Conceptually, this is exactly the same as back-propagation with vectors.
- The only difference is how the numbers are arranged in a grid to form a tensor. We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor.
- In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

6.5.2 Chain Rule of Calculus

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

$$(\nabla_{\mathbf{X}} z)_i = \sum_j (\nabla_{\mathbf{X}} \mathrm{Y}_j) \frac{\partial z}{\partial \mathrm{Y}_j}.$$

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. However, actually evaluating that expression in a computer introduces some extra considerations.
- Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times. An example of how these repeated subexpressions arise is given in
Fig. 6.9.

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- 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.

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

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

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

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

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- 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 Sec. 6.5.6, with Algorithm 6.5, where we also generalize to nodes that contain arbitrary tensors.

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- 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 ni input nodes $u^{(1)}$ to $u^{(n_i)}$. In other words we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1, 2, \ldots, n_i\}$.
- In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model.

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- 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

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

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

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

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

---

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

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

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

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

as specified by Algorithm 6.2.

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

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

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

as specified by Algorithm 6.2.

Thank You for your Attention