Lecture 13: Backpropagation and ML Frameworks.

CS4787 — Principles of Large-Scale Machine Learning Systems

Recall: ReLU neural networks. We saw last time that we could express a ReLU neural network as

$$h(x) = W_{\mathcal{L}} \cdot \mathrm{ReLU}(W_{\mathcal{L}-1} \cdot \mathrm{ReLU}(\cdots) + b_{\mathcal{L}-1}).$$

We can write this more explicitly and more generally in terms of a recurrence relation $o_0 = x$ and

$$\begin{aligned} \forall l \in \{1, \ldots, \mathcal{L}\}, \quad a_l &= W_l \cdot o_{l-1} + b_l \\ \forall l \in \{1, \ldots, \mathcal{L}\}, \quad o_l &= \sigma_l(a_l), \end{aligned}$$

where σ_l is now the *non-linearity* or *activation function* used at the lth layer. It's clear that we could now try to learn this by differentiating this expression using the chain rule and then using the resulting expression in SGD. But does the simple mathematical expression we get from the chain rule give us the best way of *computing* the gradient?

A simple example. Imagine that we have some functions f, g, and h, and we are interested in computing the derivative of the function $f(g(h(x)))$ with respect to x. By the chain rule,

$$\frac{d}{dx} f(g(h(x))) = f'(g(h(x))) \cdot g'(h(x)) \cdot h'(x).$$

Imagine that we just copy-pasted this math into python. We'd have something like

```
def grad_fgh(x):
        return grad_f(g(h(x))) * grad_g(h(x)) * grad_h(x)
```

This code recomputes $h(x)$ twice! To avoid this, we could imagine writing something like this instead:

```
def better_grad_fgh(x):
        hx = h(x)
        return grad_f(g(hx)) * grad_g(hx) * grad_h(x)
```

This code only computes $h(x)$ once. But we had to do some manual work to pull out common subexpressions, work that goes beyond what the plain chain rule gives us.

Now you might ask the very reasonable question: **So what?** We just recomputed one function. **Could this really have a significant impact on performance?** To answer this question, suppose that we now have functions $f_1, f_2, \ldots, f_k$, and consider differentiating the more complicated function $f_k(f_{k-1}(\cdots f_2(f_1(x))\cdots))$. The chain rule will give us

$$\frac{d}{dx} f_k(f_{k-1}(\cdots f_2(f_1(x))\cdots)) = f_k'(f_{k-1}(\cdots f_2(f_1(x))\cdots)) \cdot f_{k-1}'(f_{k-2}(\cdots f_2(f_1(x))\cdots)) \cdots f_2'(f_1(x)) \cdot f_1'(x).$$

If each function f_i or f_i' takes $O(1)$ time to compute, how long can this derivative take to compute if

...I just copy-paste naively into python?	...I do something to save redundant computation?

Take-away point: computing a derivative by just applying the chain rule and then naively copying the expression into code can be **asymptotically slower** than other methods.

- Why? Derivatives have lots of redundant expressions.
- But we've seen so far that when we avoid redundant compute, we can compute the derivative in the same amount of time as it would take us to compute the original function.

Question: Can we figure out a way to remove redundant compute and get this fast code automatically? And can we do this to produce code written with matrix multiplies for gradients of DNNs? To answer this question, we need to talk about how a computer can represent the structure of a neural network.

Forward-Mode Automatic Differentiation.

Perhaps the easiest way to do this.

- Fix one input variable
- At each step, compute partial derivative of the expression we're computing with respect to that one input variable
- Can be done with dual numbers

Reverse-Mode Automatic Differentiation.

- Fix one output expression
- Compute partial derivative of that output with respect to the expression we're focusing on

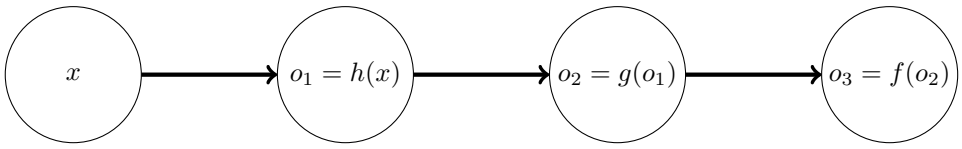
Much more common with machine learning! (Why?) Main way to do it is called *backpropagation*.

Neural networks as computational graphs.

In a computational graph,

- A **node** represents a (possibly vector or matrix) value.
- An **edge** represents a data dependency. Edges are directed.
- The value of a node is a function of the values of the nodes connected to all its **incoming edges**.

An example: for the function $f(g(h(x)))$ above we can use the computational graph



We can equivalently think of this in terms of *object orientation*.

- Each node is an **object**.
- A node object contains **references** to some number of other node objects, on which it depends (these are the incoming edges).
- A node has a method that **can compute its value**, given the values of the other nodes on which it depends.

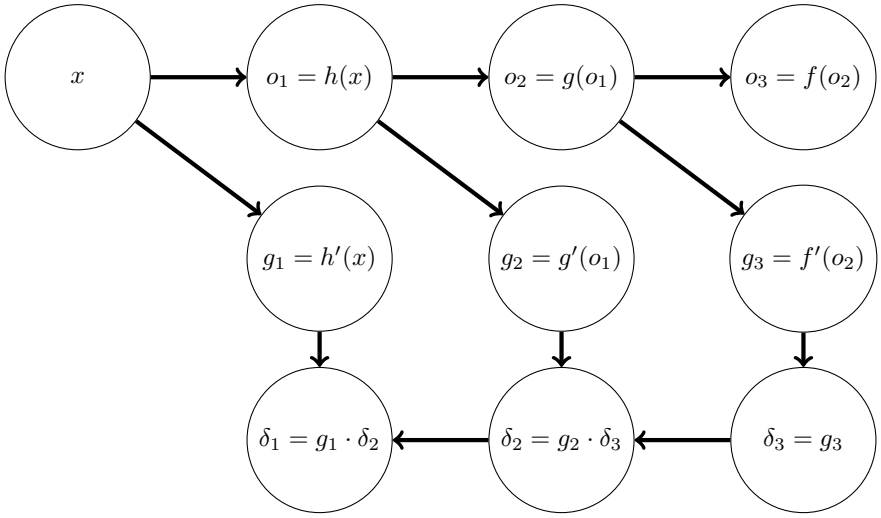
Demo: Object orientation for computational graphs.

How can this help us remove redundant computation while computing the gradient? Let's just look at how one node participates in gradient descent. For example, let's say that our functions h and g depended on some parameters w_1 and w_2. The node $o_3 = f(o_2)$, by the chain rule, has

$$\frac{\partial o_3}{\partial w_1} = f'(o_2) \cdot \frac{\partial o_2}{\partial w_1}.$$

What if we differentiate with respect to w_2 instead? What computation is redundant (for just this chunk of the computation) between the computation of these two partial derivatives?

What could a computational graph for the gradient of this function look like?



Here, the derivative of the whole function is the δ_1 node at the bottom left. One simple strategy for computing this graph without recomputing anything is to compute the top row first (and *cache* all the computed values within the node objects) and then use this to compute the rest of the nodes on-demand. This is an example of the **backpropagation** technique.

In practice, we usually don't construct a separate computational graph object to represent the gradient. Instead, we make each node object responsible for computing not only its value (in a forward pass) but also its gradient (in a backward pass). Specifically,

- In the forward pass, a *non-leaf node* takes as input the values of the other nodes it depends on, and computes its value given the parameters.
- In the backward pass, a *non-leaf node* takes as input the values of the other nodes it depends on, and (2) for each of the nodes that depend on it, the gradient of the objective with respect to its value through that node. It outputs, for each of the nodes it depends on, the gradient of the objective with respect to that node's value.

In mathematical notation, suppose that the goal is to differentiate some function f, and part of the computational graph of f is a node that has value g, which depends on nodes with values $v_1, v_2, \ldots, v_k$. Since the node with value g is part of the computational graph of f, there must be some other nodes with values $h_1, h_2, \ldots, h_m$ which depend on that value g, such that f depends on g only through $h_1, h_2, \ldots, h_m$. Then by the **multivariable chain rule**,

$$\frac{\partial f}{\partial g} = \sum_{i=1}^{m} \frac{\partial h_i}{\partial g} \cdot \frac{\partial f}{\partial h_i}.$$

And consequently

$$\frac{\partial g}{\partial v_j} \cdot \frac{\partial f}{\partial g} = \underbrace{\frac{\partial g}{\partial v_j}}_{\text{can compute given values of } v_1, \ldots, v_k} \cdot \left(\sum_{i=1}^{m} \underbrace{\frac{\partial h_i}{\partial g} \cdot \frac{\partial f}{\partial h_i}}_{\text{can be sent from node with value } h_i} \right).$$

This is an expression that node g "knows" how to compute, given values that can be sent to it from adjacent nodes in the computation graph.

This all works for vector-valued nodes too! If we overload our partial derivative notation to mean a gradient when taken with respect to a vector, then we can write

$$\underbrace{\frac{\partial g}{\partial v_j} \cdot \frac{\partial f}{\partial g}}_{\text{shape: length}(v_j)} = \underbrace{\frac{\partial g}{\partial v_j}}_{\text{shape: length}(g)\times\text{length}(v_j)} \cdot \left(\sum_{i=1}^{m} \underbrace{\frac{\partial h_i}{\partial g} \cdot \frac{\partial f}{\partial h_i}}_{\text{shape: length}(g)} \right).$$

So for a deep neural network, we can do the following, which is called the **backpropagation** algorithm. First, we compute the *forward pass*, by going from $l = 1$ to $\mathcal{L}$, where $\mathcal{L}$ is the number of layers, and computing

$$a_l = W_l \cdot o_{l-1} + b_l \qquad\qquad o_l = \sigma_l(a_l).$$

While we are doing this, we need to *save* the values of a_l and o_l for later use. Then, we compute the *backward pass*, by going from $l = \mathcal{L}$ down to 1 and computing something like

$$\frac{\partial f}{\partial a_l} = D\sigma_l(a_l) \cdot \frac{\partial f}{\partial o_l} \qquad\qquad \frac{\partial f}{\partial o_{l-1}} = W_l^T \cdot \frac{\partial f}{\partial a_l}$$

$$\nabla_{W_l} f = \frac{\partial f}{\partial a_l} \cdot o_{l-1}^T \qquad\qquad \nabla_{b_l} f = \frac{\partial f}{\partial a_l}.$$

This is also written nicely out as Algorithm 6.4 in the *Deep Learning Book*.

Take-away point: automatic differentiation lets us write the gradient in terms of fast matrix-multiply operations, as long as the original computational graph was vector-valued. And as an example of special interest, we can do this for deep neural networks.

Mini-batching and tensors. Typically, for learning, we don't just want to compute the gradient for a single training example, but rather for *multiple* training examples at once in a minibatch. How can we do it with the above automatic differentiation setup?

One way to do it is by adding an *extra dimension* to all our arrays, which stores the batch index. We can then continue to compute backpropagation using fast linear algebra routines from a linear algebra library, except now we've got an extra dimension (the minibatch index b) for all our computations. This sort of higher-dimensional array-of-numbers object is called a *tensor*. Tensors are not only useful for enabling minibatching; they are also useful in other cases, such as when the vectors of features we want to classify have a natural array structure. A classic example of this is images, which are typically stored in a tensor of dimension (# rows) $\times$ (# cols) $\times$ 3.

There are **two equivalent ways of thinking about a tensor**. The "CS" way is to think about it as a multidimensional array. The number of dimensions of a tensor is called its **order** (or sometimes, confusingly, the **rank** or **degree**). We can also think about a tensor in the "math" way as a multilinear map. The set of order-k tensors in $\mathbb{R}^{d_1 \times d_2 \times \cdots \times d_k}$ is equivalent to the set of functions $T : \mathbb{R}^{d_1} \times R^{d_2} \times \cdots \times \mathbb{R}^{d_k} \to \mathbb{R}$ that are multilinear; i.e. that satisfy

$$T(x_1, x_2, \ldots, \alpha x_i + \beta y_i, \ldots, x_k) = \alpha T(x_1, x_2, \ldots, x_i, \ldots, x_k) + \beta T(x_1, x_2, \ldots, y_i, \ldots, x_k).$$

Machine learning frameworks like TensorFlow, PyTorch, and MxNet combine (1) automatic differentiation via backprop, (2) automatic compilation of matrix multiplies to GPUs for fast compute, and (3) built-in functions and learning examples that make it easy to write and train neural networks. Mostly use **Python** as the front-end interface. These frameworks make it easy to train deep neural networks and get good performance and scalability, even for people who do not understand the principles behind their operation. This is a major driving force behind the deep learning revolution!