A Simple Function

Autograd

Data Science in Electron Microscopy

Philipp Pelz

2024

https://github.com/ECLIPSE-Lab/WS24_DataScienceForEM

Automatic Differentiation

- Recall from :numref:`sec_calculus` that calculating derivatives is the crucial step in all of the optimization algorithms that we will use to train deep networks. While the calculations are straightforward, working them out by hand can be tedious and error-prone, and this problem only grows as our models become more complex.

- Fortunately all modern deep learning frameworks take this work off of our plates by offering *automatic differentiation* (often shortened to *autograd*).

- framework builds a *computational graph* that tracks how each value depends on others

- To calculate derivatives, automatic differentiation works backwards through this graph applying the chain rule.

- The computational algorithm for applying the chain rule in this fashion is called *backpropagation*.

- Before exploring methods, let's first master the autograd package.

```
1  import torch
```

Let's assume that we are interested in (**differentiating the function $y = 2\mathbf{x}^{\top}\mathbf{x}$ with respect to the column vector $\mathbf{x}$.**) To start, we assign x an initial value.

```
1  x = torch.arange(4.0)
2  x
```

tensor([0., 1., 2., 3.])

- Before we calculate the gradient of y with respect to $\mathbf{x}$, we need a place to store it.

- In general avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters thousands or millions of times, and we might risk running out of memory.

- gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is vector-valued and has the same shape as $\mathbf{x}$.

```
1  # Can also create x = torch.arange(4.0, requires_grad=True)
2  x.requires_grad_(True)
3  x.grad  # The gradient is None by default
```

- **We now calculate our function of x and assign the result to y.**

```
1  y = 2 * torch.dot(x, x)
2  y
```
tensor(28., grad_fn=<MulBackward0>)

- **We can now take the gradient of y with respect to x** by calling its `backward` method. Next, we can access the gradient via x's `grad` attribute.

```
1  y.backward()
2  x.grad
```
tensor([0., 4., 8., 12.])

-**We already know that the gradient of the function $y = 2\mathbf{x}^{\top}\mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$.** - We can now verify that the automatic gradient computation and the expected result are identical.

```
1  x.grad == 4 * x
```
tensor([True, True, True, True])

- **Now let's calculate another function of x and take its gradient.**

- Note that PyTorch does not automatically reset the gradient buffer when we record a new gradient. Instead, the new gradient is added to the already-stored gradient. This behavior comes in handy when we want to optimize the sum of multiple objective functions. To reset the gradient buffer, we can call `x.grad.zero_()` as follows:

```python
x.grad.zero_()   # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

```
tensor([1., 1., 1., 1.])
```

Backward for Non-Scalar Variables

- When y is a vector, the most natural interpretation of the derivative of y with respect to a vector x is a matrix called the *Jacobian* that contains the partial derivatives of each component of y with respect to each component of x.

- Likewise, for higher-order y and x, the differentiation result could be an even higher-order tensor.

- While Jacobians do show up in some advanced machine learning techniques, more commonly we want to sum up the gradients of each component of y with respect to the full vector x, yielding a vector of the same shape as x.

- we often have a vector representing the value of our loss function calculated separately for each example among a *batch* of training examples.

- Here, we just want to (**sum up the gradients computed individually for each example**).

- deep learning frameworks vary in how they interpret gradients of non-scalar tensors, PyTorch takes some steps to avoid confusion.

- Invoking `backward` on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar.

- More formally, we need to provide some vector $\mathbf{v}$ such that `backward` will compute $\mathbf{v}^\top \partial_{\mathbf{x}} \mathbf{y}$ rather than $\partial_{\mathbf{x}} \mathbf{y}$.

- This next part may be confusing, but for reasons that will become clear later, this argument (representing $\mathbf{v}$) is named `gradient`. For a more detailed description, see Yang Zhang's Medium post.

```
1  x.grad.zero_()
2  y = x * x
3  y.backward(gradient=torch.ones(len(y)))   # Faster: y.sum().backward()
4  x.grad
```

```
tensor([0., 2., 4., 6.])
```

Detaching Computation

- Sometimes wish to [**move some calculations outside of the recorded computational graph.**]

- For example, say that we use the input to create some auxiliary intermediate terms for which we do not want to compute a gradient.

- In this case, we need to *detach* the respective computational graph from the final result.

- The following toy example makes this clearer: suppose we have z = x * y and y = x * x but we want to focus on the *direct* influence of x on z rather than the influence conveyed via y.

- In this case, we can create a new variable u that takes the same value as y but whose *provenance* (how it was created) has been wiped out.

- Thus u has no ancestors in the graph and gradients do not flow through u to x. For example, taking the gradient of z = x * u will yield the result u, (not 3 * x * x as you might have expected since z = x * x * x).

```
1  x.grad.zero_()
2  y = x * x
3  u = y.detach()
4  z = u * x
5
6  z.sum().backward()
7  x.grad == u
```
tensor([True, True, True, True])

- Note that while this procedure detaches y's ancestors from the graph leading to z, the computational graph leading to y persists and thus we can calculate the gradient of y with respect to x.

```
1  x.grad.zero_()
2  y.sum().backward()
3  x.grad == 2 * x
```

```
tensor([True, True, True, True])
```

Gradients and Python Control Flow

- So far we reviewed cases where the path from input to output was well-defined via a function such as `z = x * x * x`.

- Programming offers us a lot more freedom in how we compute results.

- For instance, we can make them depend on auxiliary variables or condition choices on intermediate results.

- One benefit of using automatic differentiation is that [**even if**] building the computational graph of (**a function required passing through a maze of Python control flow**) (e.g., conditionals, loops, and arbitrary function calls), (**we can still calculate the gradient of the resulting variable.**)

- To illustrate this, consider the following code snippet where the number of iterations of the `while` loop and the evaluation of the `if` statement both depend on the value of the input `a`.

```python
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

- Below, we call this function, passing in a random value as input. Since the input is a random variable, we do not know what form the computational graph will take. However, whenever we execute `f(a)` on a specific input, we realize a specific computational graph and can subsequently run `backward`.

```python
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

- Even though our function `f` is a bit contrived for demonstration purposes, its dependence on the input is quite simple: it is a *linear* function of `a` with piecewise defined scale.

- As such, `f(a) / a` is a vector of constant entries and, moreover, `f(a) / a` needs to match the gradient of `f(a)` with respect to `a`.

```python
a.grad == d / a
```
```
tensor(True)
```

- Dynamic control flow is very common in deep learning.

- For instance, when processing text, the computational graph depends on the length of the input.

- In these cases, automatic differentiation becomes vital for statistical modeling since it is impossible to compute the gradient a priori.

Discussion

- The development of libraries for calculating derivatives both automatically and efficiently has been a massive productivity booster for deep learning practitioners, liberating them to focus on loftier concerns.

- autograd permits us to design massive models for which pen and paper gradient computations would be prohibitively time consuming.

- Interestingly, while we use autograd to *optimize* models (in a statistical sense) the *optimization* of autograd libraries themselves (in a computational sense) is a rich subject of vital interest to framework designers.

- Here, tools from compilers and graph manipulation are leveraged to compute results in the most expedient and memory-efficient manner.

For now, try to remember these basics:

1. attach gradients to those variables with respect to which we desire derivatives;

2. record the computation of the target value;

3. execute the backpropagation function; and

4. access the resulting gradient.

Exercises

1. Why is the second derivative much more expensive to compute than the first derivative?

2. After running the function for backpropagation, immediately run it again and see what happens. Why?

3. In the control flow example where we calculate the derivative of d with respect to a, what would happen if we changed the variable a to a random vector or a matrix? At this point, the result of the calculation `f(a)` is no longer a scalar. What happens to the result? How do we analyze this?

4. Let $f(x) = \sin(x)$. Plot the graph of f and of its derivative f'. Do not exploit the fact that $f'(x) = \cos(x)$ but rather use automatic differentiation to get the result.

5. Let $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$. Write out a dependency graph tracing results from x to $f(x)$.

6. Use the chain rule to compute the derivative $\frac{df}{dx}$ of the aforementioned function, placing each term on the dependency graph that you constructed previously.

7. Given the graph and the intermediate derivative results, you have a number of options when computing the gradient. Evaluate the result once starting from x to f and once from f tracing back to x. The path from x to f is commonly known as *forward differentiation*, whereas the path from f to x is known as backward differentiation.

8. When might you want to use forward differentiation and when backward differentiation? Hint: consider the amount of intermediate data needed, the ability to parallelize steps, and the size of matrices and vectors involved.