AUTOGRAD: AUTOMATIC DIFFERENTIATION

Central to all neural networks in PyTorch is the `autograd` package. Let's first briefly visit this, and we will then go to training our first neural network.

The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.

Let us see this in more simple terms with some examples.

Tensor

`torch.Tensor` is the central class of the package. If you set its attribute `.requires_grad` as `True`, it starts to track all operations on it. When you finish your computation you can call `.backward()` and have all the gradients computed automatically. The gradient for this tensor will be accumulated into `.grad` attribute.

To stop a tensor from tracking history, you can call `.detach()` to detach it from the computation history, and to prevent future computation from being tracked.

To prevent tracking history (and using memory), you can also wrap the code block in `with torch.no_grad():`. This can be particularly helpful when evaluating a model because the model may have trainable parameters with `requires_grad=True`, but for which we don't need the gradients.

There's one more class which is very important for autograd implementation - a `Function`.

`Tensor` and `Function` are interconnected and build up an acyclic graph, that encodes a complete history of computation. Each tensor has a `.grad_fn` attribute that references a `Function` that has created the `Tensor` (except for Tensors created by the user - their `grad_fn is None`).

If you want to compute the derivatives, you can call `.backward()` on a `Tensor`. If `Tensor` is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to `backward()`, however if it has more elements, you need to specify a `gradient` argument that is a tensor of matching shape.

```
import torch
```

Create a tensor and set `requires_grad=True` to track computation with it

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
```

Out:

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

Do a tensor operation:

```
y = x + 2
print(y)
```

Out:

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

`y` was created as a result of an operation, so it has a `grad_fn`.

```
print(y.grad_fn)
```

Out:

```
<AddBackward0 object at 0x7f2451155828>
```

Do more operations on `y`

```
z = y * y * 3
out = z.mean()

print(z, out)
```

Out:

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>) tensor(27., grad_fn=<MeanBackward0>)
```

`.requires_grad_( ... )` changes an existing Tensor's `requires_grad` flag in-place. The input flag defaults to `False` if not given.

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
```

Out:

```
False
True
<SumBackward0 object at 0x7f2451164208>
```

Gradients

Let's backprop now. Because `out` contains a single scalar, `out.backward()` is equivalent to `out.backward(torch.tensor(1.))`.

```
out.backward()
```

Print gradients d(out)/dx

```
print(x.grad)
```

Out:

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

You should have got a matrix of `4.5`. Let's call the `out` *Tensor* "oo". We have that $o = \frac{1}{4}\sum_i z_i$o=14∑izi, $z_i = 3(x_i + 2)^2$zi=3(xi+2)2 and $z_i\big|_{x_i=1} = 27$zi|xi=1=27. Therefore, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$∂o∂xi=32(xi+2), hence $\frac{\partial o}{\partial x_i}\big|_{x_i=1} = \frac{9}{2} = 4.5$∂o∂xi|xi=1=92=4.5.

Mathematically, if you have a vector valued function $\vec{y} = f(\vec{x})$y→=f(x→), then the gradient of $\vec{y}$y→ with respect to $\vec{x}$x→ is a Jacobian matrix:

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

J=(∂y1∂x1⋯∂y1∂xn⋮⋱⋮∂ym∂x1⋯∂ym∂xn)

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector $v = \begin{pmatrix} v_1 & v_2 & \cdots & v_m \end{pmatrix}^T$v=(v1v2⋯vm)T, compute the product $v^T \cdot J$vT·J. If vv happens to be the gradient of a scalar function $l = g\left(\vec{y}\right)$l=g(y→), that is, $v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T$v=(∂l∂y1⋯∂l∂ym)T, then by the chain rule, the vector-Jacobian product would be the gradient of ll with respect to $\vec{x}$x→:

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

JT·v=(∂y1∂x1⋯∂ym∂x1⋮⋱⋮∂y1∂xn⋯∂ym∂xn)(∂l∂y1⋮∂l∂ym)=(∂l∂x1⋮∂l∂xn)

(Note that $v^T \cdot J$vT·J gives a row vector which can be treated as a column vector by taking $J^T \cdot v$JT·v.)

This characteristic of vector-Jacobian product makes it very convenient to feed external gradients into a model that has non-scalar output.

Now let's take a look at an example of vector-Jacobian product:

```
x = torch.randn(3, requires_grad=True)

y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)
```

Out:

```
tensor([-285.0968, -947.5601, -235.9010], grad_fn=<MulBackward0>)
```

Now in this case `y` is no longer a scalar. `torch.autograd` could not compute the full Jacobian directly, but if we just want the vector-Jacobian product, simply pass the vector to `backward` as argument:

```
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(v)

print(x.grad)
```

Out:

```
tensor([1.0240e+02, 1.0240e+03, 1.0240e-01])
```

You can also stop autograd from tracking history on Tensors with `.requires_grad=True` either by wrapping the code block in `with torch.no_grad():`

```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

Out:

```
True
True
False
```

Or by using `.detach()` to get a new Tensor with the same content but that does not require gradients:

```
print(x.requires_grad)
y = x.detach()
print(y.requires_grad)
print(x.eq(y).all())
```

Out:

```
True
False
tensor(True)
```

Read Later:

Document about `autograd.Function` is at https://pytorch.org/docs/stable/autograd.html#function

Total running time of the script: (0 minutes 3.567 seconds)

‹ Previous Next ›