Autograd

Autograd is now a core torch package for automatic differentiation. It uses a tape based system for automatic differentiation.

In the forward phase, the autograd tape will remember all the operations it executed, and in the backward phase, it will replay the operations.

Tensors that track history

In autograd, if any input `Tensor` of an operation has `requires_grad=True`, the computation will be tracked. After computing the backward pass, a gradient w.r.t. this variable is accumulated into `.grad` attribute.

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 variable has a `.grad_fn` attribute that references a function that has created a function (except for Tensors created by the user - these have `None` as `.grad_fn`).

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 tensor), you don't need to specify any arguments to `backward()`, however if it has more elements, you need to specify a `grad_output` 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.]])
```

```
print(x.data)
```

Out:

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

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

Out:

```
None
```

```
print(x.grad_fn)  # we've created x ourselves
```

Out:

```
None
```

Do an operation of x:

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

Out:

```
tensor([[ 3.,  3.],
        [ 3.,  3.]])
```

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

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

Out:

```
<AddBackward0 object at 0x7fb9f72740f0>
```

More operations on y:

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

print(z, out)
```

Out:

```
tensor([[ 27.,  27.],
        [ 27.,  27.]]) tensor(27.)
```

`.requires_grad_( ... )` changes an existing Tensor's `requires_grad` flag in-place. The input flag defaults to `True` 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 0x7fb9f7504f60>
```

Gradients

let's backprop now and print gradients d(out)/dx

```
out.backward()
print(x.grad)
```

Out:

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

By default, gradient computation flushes all the internal buffers contained in the graph, so if you even want to do the backward on some part of the graph twice, you need to pass in `retain_variables = True` during the first pass.

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
y.backward(torch.ones(2, 2), retain_graph=True)
# the retain_variables flag will prevent the internal buffers from being freed
print(x.grad)
```

Out:

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

```
z = y * y
print(z)
```

Out:

```
tensor([[ 9.,  9.],
        [ 9.,  9.]])
```

just backprop random gradients

```
gradient = torch.randn(2, 2)

# this would fail if we didn't specify
# that we want to retain variables
y.backward(gradient)

print(x.grad)
```

Out:

```
tensor([[ 0.6683,  2.2307],
        [ 1.3618,  0.7551]])
```

You can also stops autograd from tracking history on Tensors with requires_grad=True 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
```

Total running time of the script: (0 minutes 0.003 seconds)

Download Python source code: autograd_tutorial.py

Download Jupyter notebook: autograd_tutorial.ipynb