EE-559 – Deep learning

5.7. Writing an autograd function

François Fleuret

https://fleuret.org/ee559/

Wed Aug 29 14:57:38 UTC 2018

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

We have seen how to write new `torch.nn.Module`s. We may have to implement new functions usable with autograd, so that `Module`s remain defined through their forward pass alone.

This is achieved by writing sub-classes of `torch.autograd.Function`, which have to implement two static methods:

- `forward(...)` takes as argument a context to store information needed for the backward pass, and the quantities it should process, which are `Tensor`s for the differentiable ones, but can also be any other types. It should return one or several `Tensor`s.

This is achieved by writing sub-classes of `torch.autograd.Function`, which have to implement two static methods:

- `forward(...)` takes as argument a context to store information needed for the backward pass, and the quantities it should process, which are `Tensor`s for the differentiable ones, but can also be any other types. It should return one or several `Tensor`s.
- `backward(...)` takes as argument the context and as many `Tensor`s as `forward` returns `Tensor`s, and it should return as many values as `forward` takes argument, `Tensors`s for the tensors and `None` for the others.

This is achieved by writing sub-classes of `torch.autograd.Function`, which have to implement two static methods:

- `forward(...)` takes as argument a context to store information needed for the backward pass, and the quantities it should process, which are `Tensor`s for the differentiable ones, but can also be any other types. It should return one or several `Tensor`s.
- `backward(...)` takes as argument the context and as many `Tensor`s as `forward` returns `Tensor`s, and it should return as many values as `forward` takes argument, `Tensors`s for the tensors and `None` for the others.

Evaluating such a `Function` is done through its `apply(...)` method, which takes as many arguments as `forward(...)`, context excluded.

If you create a new `Function` named `Dummy`, when `Dummy.apply(...)` is called, autograd first adds a new node of type `DummyBackward` in its graph, and then calls `Dummy.forward(...)`.

If you create a new `Function` named `Dummy`, when `Dummy.apply(...)` is called, autograd first adds a new node of type `DummyBackward` in its graph, and then calls `Dummy.forward(...)`.

To compute the gradient, autograd evaluates the graph and calls `Dummy.backward(...)` when it reaches the corresponding node, with the same context as the one given to `Dummy.forward(...)`.

If you create a new `Function` named `Dummy`, when `Dummy.apply(...)` is called, autograd first adds a new node of type `DummyBackward` in its graph, and then calls `Dummy.forward(...)`.

To compute the gradient, autograd evaluates the graph and calls `Dummy.backward(...)` when it reaches the corresponding node, with the same context as the one given to `Dummy.forward(...)`.

This machinery is hidden to you and this level of details should not be required for normal operations.

Consider a function to set to zero the first n components of a tensor.

```
class KillHead(Function):
    @staticmethod
    def forward(ctx, input, n):
        ctx.n = n
        result = input.clone()
        result[:, 0:ctx.n] = 0
        return result

    @staticmethod
    def backward(ctx, grad_output):
        result = grad_output.clone()
        result[:, 0:ctx.n] = 0
        return result, None

killhead = KillHead.apply
```

It can be used for instance

```
y = torch.empty(3, 8).normal_()
x = torch.empty(y.size()).normal_().requires_grad_()

criterion, eta = nn.MSELoss(), 1e-0

for k in range(5):
    r = killhead(x, 2)
    loss = criterion(r, y)
    if k > 0: x.grad.zero_()
    loss.backward()
    print(k, loss.item())
    with torch.no_grad():
        x.sub_(eta * x.grad)
```

It can be used for instance

```
y = torch.empty(3, 8).normal_()
x = torch.empty(y.size()).normal_().requires_grad_()

criterion, eta = nn.MSELoss(), 1e-0

for k in range(5):
    r = killhead(x, 2)
    loss = criterion(r, y)
    if k > 0: x.grad.zero_()
    loss.backward()
    print(k, loss.item())
    with torch.no_grad():
        x.sub_(eta * x.grad)
```

prints

```
0 1.5175858736038208
1 1.310139536857605
2 1.1358269453048706
3 0.9893561005592346
4 0.8662799000740051
```

The `torch.autograd.gradcheck(...)` function checks numerically that the backward function is correct, *i.e.*

$$\forall i, j, \ \left| \frac{f_i(x_1, \dots, x_j + \epsilon, \dots, x_D) - f_i(x_1, \dots, x_j - \epsilon, \dots, x_D)}{2\epsilon} - (J_f(x))_{i,j} \right| \leq \alpha$$

The `torch.autograd.gradcheck(...)` function checks numerically that the backward function is correct, *i.e.*

$$\forall i, j, \ \left| \frac{f_i(x_1, \dots, x_j + \epsilon, \dots, x_D) - f_i(x_1, \dots, x_j - \epsilon, \dots, x_D)}{2\epsilon} - (J_f(x))_{i,j} \right| \leq \alpha$$

```
x = torch.empty(10, 20, dtype = torch.float64).uniform_(-1, 1).requires_grad_()
input = (x, 4)

if gradcheck(killhead, input, eps = 1e-6, atol = 1e-4):
    print('All good captain.')
else:
    print('Ouch')
```

The `torch.autograd.gradcheck(...)` function checks numerically that the backward function is correct, *i.e.*

$$\forall i, j, \ \left| \frac{f_i(x_1, \dots, x_j + \epsilon, \dots, x_D) - f_i(x_1, \dots, x_j - \epsilon, \dots, x_D)}{2\epsilon} - (J_f(x))_{i,j} \right| \leq \alpha$$

```
x = torch.empty(10, 20, dtype = torch.float64).uniform_(-1, 1).requires_grad_()
input = (x, 4)

if gradcheck(killhead, input, eps = 1e-6, atol = 1e-4):
    print('All good captain.')
else:
    print('Ouch')
```

⚠ It is advisable to use `torch.float64`s for such a check.

Consider a function that takes two similar sized `Tensor`s and apply component-wise

$$(u, v) \mapsto |uv|.$$

Consider a function that takes two similar sized `Tensor`s and apply component-wise

$$(u, v) \mapsto |uv|.$$

The backward has to compute two tensors, and the forward must keep track of the input to compute the derivatives in the backward.

Consider a function that takes two similar sized Tensors and apply component-wise

$$(u, v) \mapsto |uv|.$$

The backward has to compute two tensors, and the forward must keep track of the input to compute the derivatives in the backward.

```
class Something(Function):
    @staticmethod
    def forward(ctx, input1, input2):
        ctx.save_for_backward(input1, input2)
        return (input1 * input2).abs()

    @staticmethod
    def backward(ctx, grad_output):
        input1, input2 = ctx.saved_variables
        return grad_output * input1.sign() * input2.abs(), \
               grad_output * input1.abs() * input2.sign()

something = Something.apply
```

The end