4.2. Autograd

François Fleuret
https://fleuret.org/ammi-2018/
Sat Nov 10 10:06:29 UTC 2018
Conceptually, the forward pass is a standard tensor computation, and the DAG of tensor operations is required only to compute derivatives.
Conceptually, the forward pass is a standard tensor computation, and the DAG of tensor operations is required only to compute derivatives.

When executing tensor operations, PyTorch can automatically construct on-the-fly the graph of operations to compute the gradient of any quantity with respect to any tensor involved.
Conceptually, the forward pass is a standard tensor computation, and the DAG of tensor operations is required only to compute derivatives.

When executing tensor operations, PyTorch can automatically construct on-the-fly the graph of operations to compute the gradient of any quantity with respect to any tensor involved.

This “autograd” mechanism has two main benefits:

- Simpler syntax: one just need to write the forward pass as a standard sequence of Python operations,
- greater flexibility: since the graph is not static, the forward pass can be dynamically modulated.
A Tensor has a Boolean field `requires_grad`, set to False by default, which states if PyTorch should build the graph of operations so that gradients wrt to it can be computed.

```python
>>> x = torch.tensor([1., 2.])
>>> y = torch.tensor([4., 5.])
>>> z = torch.tensor([7., 3.])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```
A **Tensor** has a Boolean field `requires_grad`, set to **False** by default, which states if PyTorch should build the graph of operations so that gradients wrt to it can be computed.

The result of a tensorial operation has this flag to **True** if any of its operand has it to **True**.

```python
>>> x = torch.tensor([ 1., 2. ])
>>> y = torch.tensor([ 4., 5. ])
>>> z = torch.tensor([ 7., 3. ])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```
Only floating point type tensors can have their gradient computed.

```python
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: only Tensors of floating point dtype can require gradients
```
Only floating point type tensors can have their gradient computed.

```python
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: only Tensors of floating point dtype can require gradients
```

The method `requires_grad_(value = True)` set `requires_grad` to `value`, which is `True` by default.
torch.autograd.grad(outputs, inputs) computes and returns the gradient of outputs wrt inputs.

```python
>>> t = torch.tensor([1., 2., 4.]).requires_grad_()
>>> u = torch.tensor([10., 20.]).requires_grad_()
>>> a = t.pow(2).sum() + u.log().sum()
>>> torch.autograd.grad(a, (t, u))
(tensor([2., 4., 8.]), tensor([0.1000, 0.0500]))
```
torch.autograd.grad(outputs, inputs) computes and returns the gradient of outputs wrt inputs.

```python
>>> t = torch.tensor([1., 2., 4.]).requires_grad_()
>>> u = torch.tensor([10., 20.]).requires_grad_()
>>> a = t.pow(2).sum() + u.log().sum()
>>> torch.autograd.grad(a, (t, u))
(tensor([2., 4., 8.]), tensor([0.1000, 0.0500]))
```

inputs can be a single tensor, but the result is still a [one element] tuple.

If outputs is a tuple, the result is the sum of the gradients of its elements.
Every Tensor also has a field \texttt{grad}, itself a tensor of same size, type, and device (or \texttt{None}) used to accumulate gradients by some functions.
Every Tensor also has a field grad, itself a tensor of same size, type, and device (or None) used to accumulate gradients by some functions.

The function `torch.autograd.backward(tensors)` or `Tensor.backward()` accumulates the gradients in the grad fields of the tensors which are not results of operations, that is “leaf” tensors in the autograd graph.

This function is an alternative to `torch.autograd.grad(...)` and standard for training models.
Tensor.backward() accumulates the gradients in the different Tensors, so one may have to set them to zero before calling it.

This accumulating behavior is desirable in particular to compute the gradient of a loss summed over several “mini-batches,” or the gradient of a sum of losses.
So we can run a forward/backward pass on

\[
\begin{align*}
  \phi^1(x) &= x_1 \\
  \phi^2(x) &= x_2 \\
  \phi^3(x) &= x_3 \\
  f(x) &= x_3
\end{align*}
\]
So we can run a forward/backward pass on

\[
\phi^{(1)} \left( x^{(0)} ; w^{(1)} \right) = w^{(1)} x^{(0)}
\]

\[
\phi^{(2)} \left( x^{(0)} , x^{(1)} ; w^{(2)} \right) = x^{(0)} + w^{(2)} x^{(1)}
\]

\[
\phi^{(3)} \left( x^{(1)} , x^{(2)} ; w^{(1)} \right) = w^{(1)} \left( x^{(1)} + x^{(2)} \right)
\]
So we can run a forward/backward pass on

\[
\begin{align*}
\phi^{(1)} (x^{(0)}; w^{(1)}) &= w^{(1)} x^{(0)} \\
\phi^{(2)} (x^{(0)}, x^{(1)}; w^{(2)}) &= x^{(0)} + w^{(2)} x^{(1)} \\
\phi^{(3)} (x^{(1)}, x^{(2)}; w^{(1)}) &= w^{(1)} (x^{(1)} + x^{(2)})
\end{align*}
\]

w1 = torch.rand(5, 5).requires_grad_()
w2 = torch.rand(5, 5).requires_grad_()
x = torch.empty(5).normal_()
x0 = x
x1 = w1 @ x0
x2 = x0 + w2 @ x1
x3 = w1 @ (x1 + x2)
q = x3.norm()
q.backward()
We can look precisely at the graph built during a computation.

```python
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```
We can look precisely at the graph built during a computation.

```
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```
We can look precisely at the graph built during a computation.

```python
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```

This graph was generated with

https://fleuret.org/git/agtreet2dot

and Graphviz.
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()

x = torch.rand(10)
h = torch.tanh(w1 @ x + b1)
y = torch.tanh(w2 @ h + b2)

target = torch.rand(5)

loss = (y - target).pow(2).mean()
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()
x = torch.rand(10)
h = torch.tanh(w1 @ x + b1)
y = torch.tanh(w2 @ h + b2)
target = torch.rand(5)
loss = (y - target).pow(2).mean()
w = torch.rand(3, 10, 10).requires_grad_()

def blah(k, x):
    for i in range(k):
        x = torch.tanh(w[i] @ x)
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```python
w = torch.rand(3, 10, 10).requires_grad_()

def blah(k, x):
    for i in range(k):
        x = torch.tanh(w[i] @ x)
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```
Although they are related, **the autograd graph is not the network’s structure**, but the graph of operations to compute the gradient. It can be data-dependent and miss or replicate sub-parts of the network.
The `torch.no_grad()` context switches off the autograd machinery, and can be used for operations such as parameter updates.

```python
torch.empty(10, 784).normal_(0, 1e-3).requires_grad_()
b = torch.empty(10).normal_(0, 1e-3).requires_grad_()

for k in range(10001):
    y_hat = x @ w.t() + b
    loss = (y_hat - y).pow(2).mean()

    w.grad, b.grad = None, None
    loss.backward()

    with torch.no_grad():
        w -= eta * w.grad
        b -= eta * b.grad
```
The `detach()` method creates a tensor which shares the data, but does not require gradient computation, and is not connected to the current graph.

This method should be used when the gradient should not be propagated beyond a variable, or to update leaf tensors.
a = torch.tensor(0.5).requires_grad_()
b = torch.tensor(-0.5).requires_grad_()

for k in range(100):
    l = (a - 1)**2 + (b + 1)**2 + (a - b)**2
    ga, gb = torch.autograd.grad(l, (a, b))
    with torch.no_grad():
        a -= eta * ga
        b -= eta * gb

    print('%.06f' % a.item(), '%.06f' % b.item())

prints

0.333333 -0.333333
a = torch.tensor(0.5).requires_grad_()
b = torch.tensor(-0.5).requires_grad_()

for k in range(100):
    l = (a - 1)**2 + (b + 1)**2 + (a.detach() - b)**2
    ga, gb = torch.autograd.grad(l, (a, b))
    with torch.no_grad():
        a -= eta * ga
        b -= eta * gb

print('%.06f' % a.item(), '%.06f' % b.item())

prints

1.000000 -0.000000
Autograd can also track the computation of the gradient itself, to allow higher-order derivatives. This is specified with `create_graph = True`:

```
>>> x = Tensor([1., 2., 3.]).requires_grad_()
>>> phi = x.pow(2).sum()
>>> g1, = torch.autograd.grad(phi, x, create_graph = True)
>>> g1
tensor([ 2., 4., 6.])
>>> psi = g1[0].exp() - g1[2].exp()
>>> g2, = torch.autograd.grad(psi, x)
>>> g2
tensor([14.7781, 0.0000, -806.8576])
```
Autograd can also track the computation of the gradient itself, to allow higher-order derivatives. This is specified with `create_graph = True`:

```python
>>> x = Tensor([1., 2., 3.]).requires_grad_()
>>> phi = x.pow(2).sum()
>>> g1, = torch.autograd.grad(phi, x, create_graph = True)
>>> g1
tensor([2., 4., 6.])
```

```python
>>> psi = g1[0].exp() - g1[2].exp()
```

```python
>>> g2, = torch.autograd.grad(psi, x)
```

```python
>>> g2
tensor([14.7781, 0.0000, -806.8576])
```
In-place operations may corrupt values required to compute the gradient, and this is tracked down by autograd.

```python
>>> x = torch.tensor([1., 2., 3.]).requires_grad_
>>> y = x.sin()
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y += 1
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y *= y
>>> l = y.sum()
>>> l.backward()
Traceback (most recent call last):
  .../
RuntimeError: one of the variables needed for gradient computation has
been modified by an inplace operation
```

They are also prohibited on so-called “leaf” tensors, which are not the results of operations but the initial inputs to the whole computation.
The end