

# EE-559 – Deep learning

## 4.2. Autograd

François Fleuret

<https://fleuret.org/ee559/>

Mar 10, 2020

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 (Paszke et al., 2017) has two main benefits:

- Simpler syntax: one just needs 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 with respect to it can be computed.

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 with respect 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`.

```
>>> 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.

```
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
/.../
RuntimeError: only Tensors of floating point dtype can require gradients
```



Only floating point type tensors can have their gradient computed.

```
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
/.../
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 with respect to inputs.

```
>>> 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` with respect to `inputs`.

```
>>> 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.

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

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

```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.]
```

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

```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.]
```

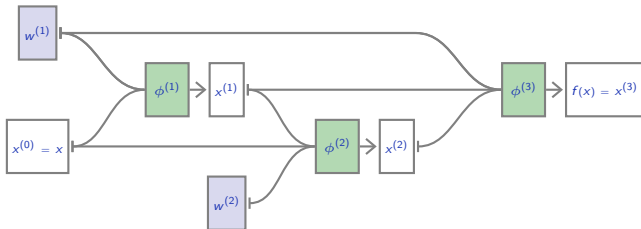
This function is an alternative to `torch.autograd.grad(...)` and standard for training models.



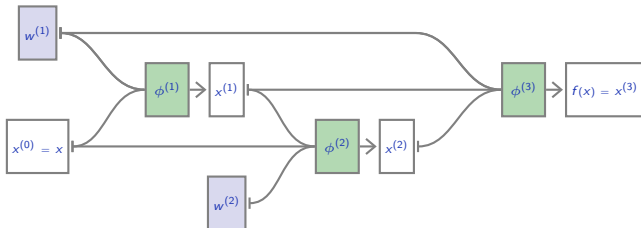
`Tensor.backward()` **accumulates** the gradients in the `grad` fields of tensors, so one may have to set them to zero before calling it.

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



So we can run a forward/backward pass on



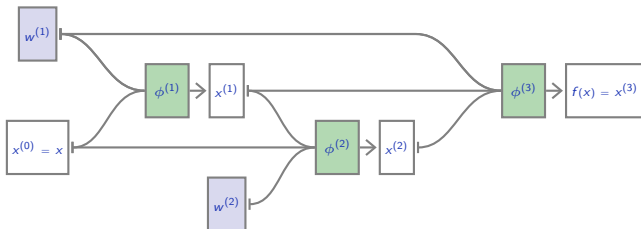
$$\phi^{(1)}(x^{(0)}; w^{(1)}) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) = x^{(0)} + w^{(2)}x^{(1)}$$

$$\phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)}) = w^{(1)}(x^{(1)} + x^{(2)})$$



So we can run a forward/backward pass on



$$\phi^{(1)}(x^{(0)}; w^{(1)}) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) = x^{(0)} + w^{(2)}x^{(1)}$$

$$\phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)}) = w^{(1)}(x^{(1)} + x^{(2)})$$

```
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()
```

## The autograd machinery

The autograd graph is encoded through the fields `grad_fn` of `Tensors`, and the fields `next_functions` of `Functions`.

```
>>> x = torch.tensor([ 1.0, -2.0, 3.0, -4.0 ]).requires_grad_()
>>> a = x.abs()
>>> s = a.sum()
>>> s
tensor(10., grad_fn=<SumBackward0>)
>>> s.grad_fn.next_functions
((<AbsBackward object at 0x7ffb2b1462b0>, 0),)
>>> s.grad_fn.next_functions[0][0].next_functions
((<AccumulateGrad object at 0x7ffb2b146278>, 0),)
```

The autograd graph is encoded through the fields `grad_fn` of `Tensors`, and the fields `next_functions` of `Functions`.

```
>>> x = torch.tensor([ 1.0, -2.0, 3.0, -4.0 ]).requires_grad_()
>>> a = x.abs()
>>> s = a.sum()
>>> s
tensor(10., grad_fn=<SumBackward0>)
>>> s.grad_fn.next_functions
((<AbsBackward object at 0x7ffb2b1462b0>, 0),)
>>> s.grad_fn.next_functions[0][0].next_functions
((<AccumulateGrad object at 0x7ffb2b146278>, 0),)
```

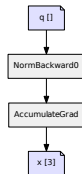
We will come back to this later to write our own `Functions`.

We can visualize the full graph built during a computation.

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

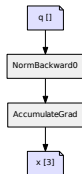
We can visualize the full graph built during a computation.

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



We can visualize the full graph built during a computation.

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



This graph was generated with

<https://fleuret.org/git/agtreetodot>

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()
```



```

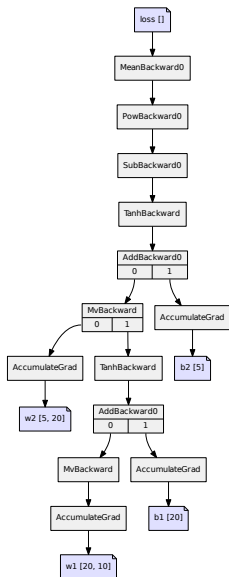
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()

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_()
```

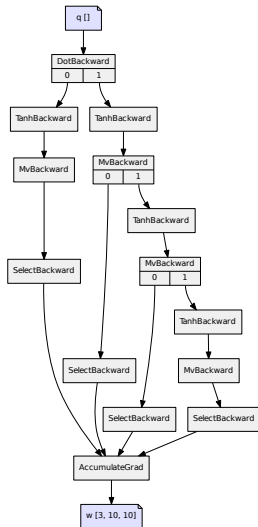
```
def blah(k, x):  
    for i in range(k):  
        x = torch.tanh(w[i] @ x)  
    return x
```

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

```
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.

```
w = 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(a, b)

prints

tensor(0.3333, requires_grad=True) tensor(-0.3333, requires_grad=True)
```

```
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(a, b)

prints

tensor(1.0000, requires_grad=True) tensor(-8.2480e-08, requires_grad=True)
```



By default, autograd deletes the computational graph when it is used.

```
>>> x = torch.tensor([1.]).requires_grad_()
>>> z = 1/x
>>> torch.autograd.grad(z, x)
(tensor([-1.]),)
>>> torch.autograd.grad(z * z, x)
Traceback (most recent call last):
/.../
RuntimeError: Trying to backward through the graph a second time, but
the buffers have already been freed.
```

By default, autograd deletes the computational graph when it is used.

```
>>> x = torch.tensor([1.]).requires_grad_()
>>> z = 1/x
>>> torch.autograd.grad(z, x)
(tensor([-1.]),)
>>> torch.autograd.grad(z * z, x)
Traceback (most recent call last):
/.../
RuntimeError: Trying to backward through the graph a second time, but
the buffers have already been freed.
```

The flag `retain_graph` indicates to keep it.

```
>>> x = torch.tensor([1.]).requires_grad_()
>>> z = 1/x
>>> torch.autograd.grad(z, x, retain_graph = True)
(tensor([-1.]),)
>>> torch.autograd.grad(z * z, x)
(tensor([-2.]),)
```

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

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

$$\begin{aligned}\psi(x_1, x_2) &= \log(x_1) + x_2^2 \\ \|\nabla\psi\|_2^2 &= \left(\frac{1}{x_1}\right)^2 + (2x_2)^2 \\ \nabla\|\nabla\psi\|_2^2 &= \left(-\frac{2}{x_1^3}, 8x_2\right)\end{aligned}$$

```
>>> x = torch.tensor([2., 3.]).requires_grad_()
>>> psi = x[0].log() + x[1].pow(2)
>>> g, = torch.autograd.grad(psi, x, create_graph = True)
>>> torch.autograd.grad(g.pow(2).sum(), x)
(tensor([-0.2500, 24.0000]),)
```



In-place operations may corrupt values required to compute the gradient, and this is tracked down by autograd.

```
>>> x = torch.tensor([1., 2., 3.]).requires_grad_()
>>> 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

## References

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. **Automatic differentiation in PyTorch**. In Proceedings of the NIPS Autodiff workshop, 2017.