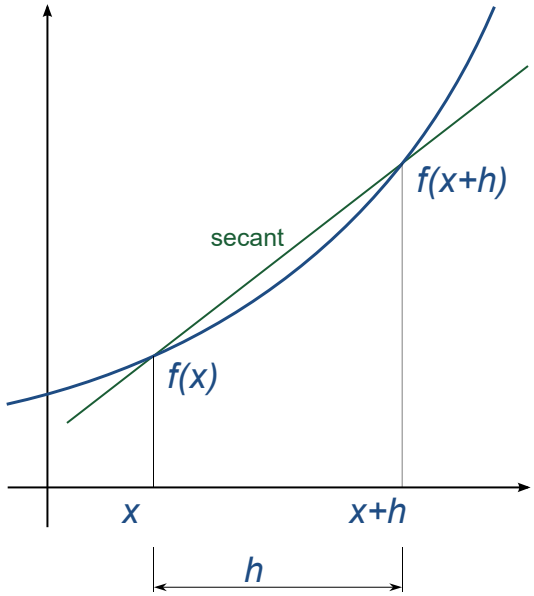
Autodiff & PyTorch for the gifted amateur

Autodiff – *Why should I care?*

- Reverse mode automatic differentiation (***autodiff***):
  - One of most important algorithms of the last century
  - Invented 1970 by Finish master student (Seppo Linnainmaa)
- Libraries (**autograd**, PyTorch, TF, …) allow us to write down forward path
- … and do the rest (**back propagation**) for us
- Still:
  - We should understand what we use all the time
  - Important to understand our tools (including its limitations!)
  - Helps to debug

Automatic vs. numeric / symbolic differentiation

- **Numeric** differentiation
  - Based on finite differences
  - Numerically unstable
  - Small errors accumulate
  - **#evaluations scales with #variables**
- **Symbolic** differentiation
  - General symbolic expression of differentiated function **for arbitrary values**
  - High computational & memory costs



- Automatic differentiation (***autodiff***)
  - Exact derivative of function for a specific value
  - **Requires only a single function evaluation**

Before we start ...

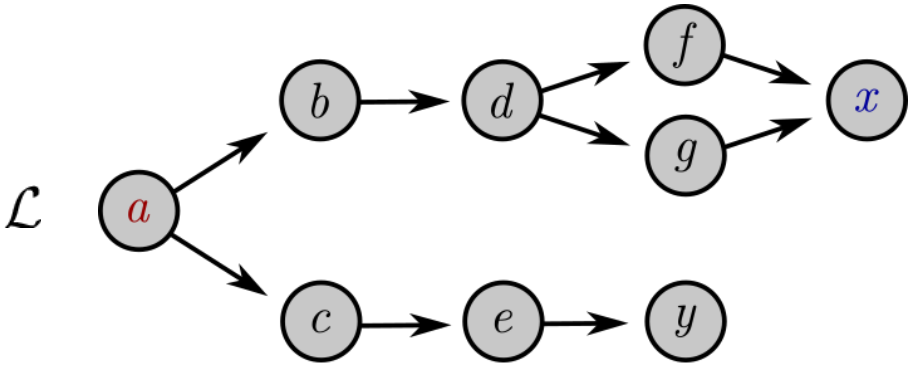
During this talk

- Autodiff for ML
  - autodiff ⟷ back-propagation
  - Loss function, $\mathcal{L}$ / *F* : multivariate function (*weights* / *x)* returning a scalar, *y*
  - Why? We need the gradient!

$$\vec{x}_{n+1} = \vec{x}_n - \gamma \underbrace{\vec{\nabla}\mathcal{L}(\vec{x}_n)}_{\text{grad } y}$$

- Ask questions!

Back propagation is a fancy name for chain rule

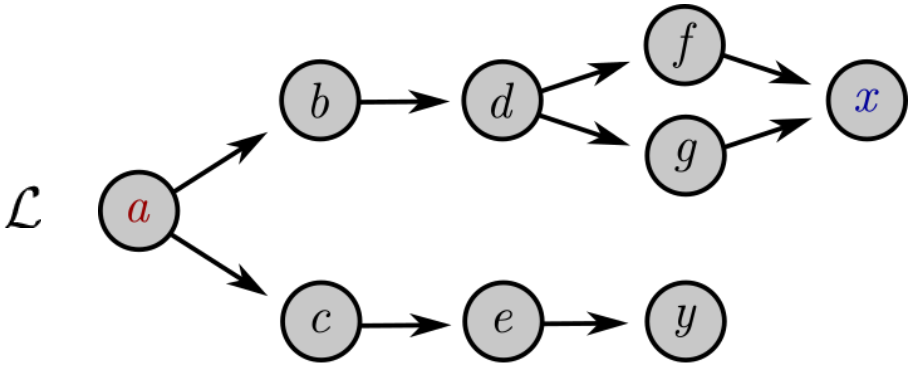


$$
\begin{aligned}
a &= A(b, c) \\
b &= B(d) \\
d &= D(f, g) \\
f &= F(x) \\
&\vdots
\end{aligned}
$$

$$
\frac{\partial \mathcal{L}}{\partial x} \equiv \frac{\partial a}{\partial x} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial a}{\partial c}\frac{\partial c}{\partial x}
$$

Goal: accumulate gradients for `x` in `x.grad`

Chain rule FTW

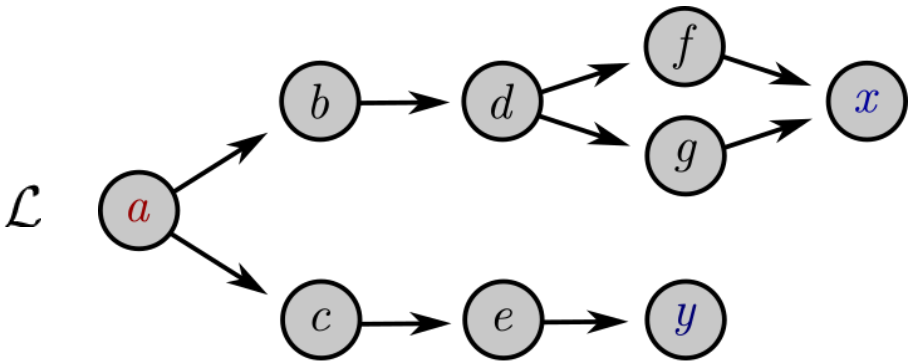


$$
\begin{aligned}
a &= A(b, c) \\
b &= B(d) \\
d &= D(f, g) \\
f &= F(x) \\
&\vdots
\end{aligned}
$$

$$
\frac{\partial \mathcal{L}}{\partial x} \equiv \frac{\partial a}{\partial x} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial x} + \underbrace{\frac{\partial a}{\partial c}\frac{\partial c}{\partial x}}_{=0}
$$

Goal: accumulate gradients for `x` in `x.grad`

Chain rule FTW



$$
\begin{aligned}
a &= A(b, c) \\
b &= B(d) \\
d &= D(f, g) \\
f &= F(x) \\
&\vdots
\end{aligned}
$$

$$
\frac{\partial \mathcal{L}}{\partial x} \equiv \frac{\partial a}{\partial x} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial d}\frac{\partial d}{\partial x} + \frac{\partial a}{\partial c}\frac{\partial c}{\partial e}\frac{\partial e}{\partial x}
$$

$$
\frac{\partial \mathcal{L}}{\partial y} \equiv \frac{\partial a}{\partial y} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial d}\frac{\partial d}{\partial y} + \frac{\partial a}{\partial c}\frac{\partial c}{\partial e}\frac{\partial e}{\partial y}
$$

Goal: accumulate gradients for `x` in `x.grad`
… and gradients for `y` in `y.grad`

Let's implement autodiff!

Let's implement autodiff!

- … for a rank 0 tensor (scalar)
- … supporting only addition and multiplication

Strategies:

1. Compiling
   - Read & generate source code ahead-of-time (e.g., Python to X)
   - Hard to implement
   - Libraries: TensorFlow
2. Taping / Recording
   - Monitor function execution at runtime
   - Easier to implement
   - Libraries: PyTorch, JAX

Let's implement autodiff!

```python
class Scalar:
    def __init__(self, val):
        self.val = float(val)
        self.grad = 0.0
        self.backward = lambda: None

    def __mul__(self, other):
        res = Scalar(self.val * other.val)

        def backward():...

        res.backward = backward
        return res

    def __add__(self, other):
        res = Scalar(self.val + other.val)

        def backward():...

        res.backward = backward
        return res
```

Let's implement autodiff!

```python
class Scalar:
    def __init__(self, val):
        self.val = float(val)
        self.grad = 0.0
        self.backward = lambda: None

    def __mul__(self, other):
        res = Scalar(self.val * other.val)

        def backward():
            self.grad += other.val * res.grad
            other.grad += self.val * res.grad
            self.backward()
            other.backward()

        res.backward = backward
        return res
```

$$f(a,b) = ab$$

$$\hookrightarrow \mathrm{d}f = \frac{\partial f}{\partial a}\,\mathrm{d}a + \frac{\partial f}{\partial b}\,\mathrm{d}b = b\,\mathrm{d}a + a\,\mathrm{d}b$$

Let's implement autodiff!

```python
class Scalar:
    def __init__(self, val):
        self.val = float(val)
        self.grad = 0.0
        self.backward = lambda: None

    def __mul__(self, other):...

    def __add__(self, other):
        res = Scalar(self.val + other.val)

        def backward():
            self.grad += res.grad
            other.grad += res.grad
            self.backward()
            other.backward()

        res.backward = backward
        return res
```

$$f(a,b) = a + b$$

$$\hookrightarrow \mathrm{d}f = \frac{\partial f}{\partial a}\,\mathrm{d}a + \frac{\partial f}{\partial b}\,\mathrm{d}b = \mathrm{d}a + \mathrm{d}b$$

Let's implement autodiff!

```
x1, x2 = Scalar(2), Scalar(3)
y = x1 * (x1 + x2)

y.backward()
print("(1):", x1.grad, x2.grad)
```

Output:

```
(1): 0.0, 0.0
```

```
1  class Scalar:
2      def __init__(self, val):
3          self.val = float(val)
4          self.grad = 0.0
5          self.backward = lambda: None
6
7      def __mul__(self, other):
8          res = Scalar(self.val * other.val)
9
10         def backward():
11             self.grad += other.val * res.grad
12             other.grad += self.val * res.grad
13             self.backward()
14             other.backward()
15
16         res.backward = backward
17         return res
18
19     def __add__(self, other):
20         res = Scalar(self.val + other.val)
21
22         def backward():
23             self.grad += res.grad
24             other.grad += res.grad
25             self.backward()
26             other.backward()
27
28         res.backward = backward
29         return res
```

Let's implement autodiff!

```
x1, x2 = Scalar(2), Scalar(3)
y = x1 * (x1 + x2)

y.backward()
print("(1):", x1.grad, x2.grad)

y.grad = 1.0
y.backward()
print("(2):", x1.grad, x2.grad)
```

Output:

```
(1): 0.0, 0.0
(2): 7.0, 2.0
```

```
1  class Scalar:
2      def __init__(self, val):
3          self.val = float(val)
4          self.grad = 0.0
5          self.backward = lambda: None
6
7      def __mul__(self, other):
8          res = Scalar(self.val * other.val)
9
10         def backward():
11             self.grad += other.val * res.grad
12             other.grad += self.val * res.grad
13             self.backward()
14             other.backward()
15
16         res.backward = backward
17         return res
18
19     def __add__(self, other):
20         res = Scalar(self.val + other.val)
21
22         def backward():
23             self.grad += res.grad
24             other.grad += res.grad
25             self.backward()
26             other.backward()
27
28         res.backward = backward
29         return res
```

Let's implement autodiff!

```
x1, x2 = Scalar(2), Scalar(3)
y = x1 * (x1 + x2)

y.backward()
print("(1):", x1.grad, x2.grad)

y.grad = 1.0
y.backward()
print("(2):", x1.grad, x2.grad)

y.grad = 1.0
y.backward()
print("(3):", x1.grad, x2.grad)
```

Output:

```
(1): 0.0, 0.0
(2): 7.0, 2.0
(3): 16.0, 6.0
```

```
class Scalar:
    def __init__(self, val):
        self.val = float(val)
        self.grad = 0.0
        self.backward = lambda: None

    def __mul__(self, other):
        res = Scalar(self.val * other.val)

        def backward():
            self.grad += other.val * res.grad
            other.grad += self.val * res.grad
            self.backward()
            other.backward()

        res.backward = backward
        return res

    def __add__(self, other):
        res = Scalar(self.val + other.val)

        def backward():
            self.grad += res.grad
            other.grad += res.grad
            self.backward()
            other.backward()

        res.backward = backward
        return res
```

Let's implement autodiff!

```
x1, x2 = Scalar(2), Scalar(3)
y = x1 * (x1 + x2)

y.backward()
print("(1):", x1.grad, x2.grad)

y.grad = 1.0
y.backward()
print("(2):", x1.grad, x2.grad)

y.grad = 1.0
y.backward()
print("(3):", x1.grad, x2.grad)

x1.grad = 0.0
x2.grad = 0.0
y.grad = 1.0
y.backward()
print("(4):", x1.grad, x2.grad)
```

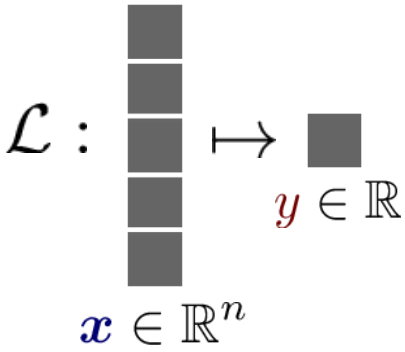
Output:

```
(1): 0.0, 0.0
(2): 7.0, 2.0
(3): 16.0, 6.0
(4): 11.0, 6.0
```

```
1  class Scalar:
2      def __init__(self, val):
3          self.val = float(val)
4          self.grad = 0.0
5          self.backward = lambda: None
6
7      def __mul__(self, other):
8          res = Scalar(self.val * other.val)
9
10         def backward():
11             self.grad += other.val * res.grad
12             other.grad += self.val * res.grad
13             self.backward()
14             other.backward()
15
16         res.backward = backward
17         return res
18
19     def __add__(self, other):
20         res = Scalar(self.val + other.val)
21
22         def backward():
23             self.grad += res.grad
24             other.grad += res.grad
25             self.backward()
26             other.backward()
27
28         res.backward = backward
29         return res
```

Multivariate autodiff

$$y = \mathcal{L}(\boldsymbol{x}) = (A \circ B \circ C \circ D)(\boldsymbol{x}) \equiv A(B(C(D(\boldsymbol{x}))))$$



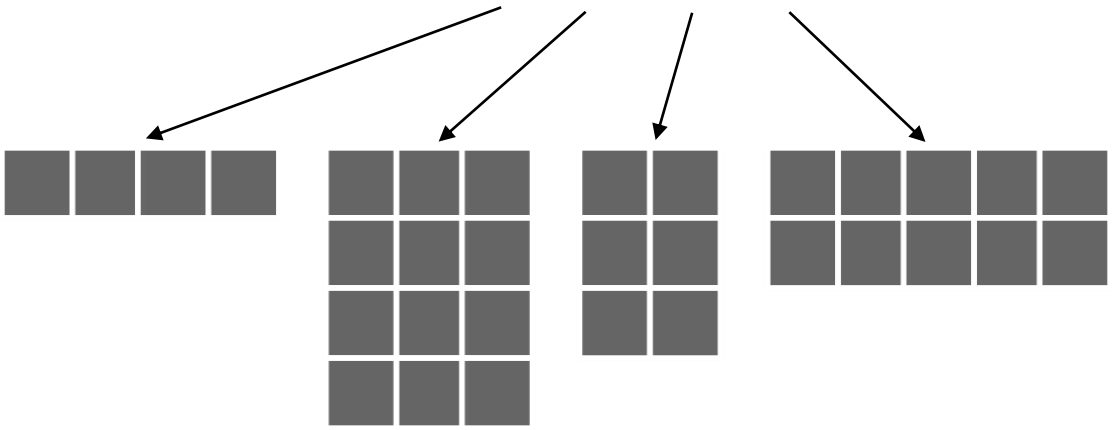
$$\mathcal{L}'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \left(\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n} \right) \in \mathbb{R}^{1 \times n}$$

Multivariate autodiff

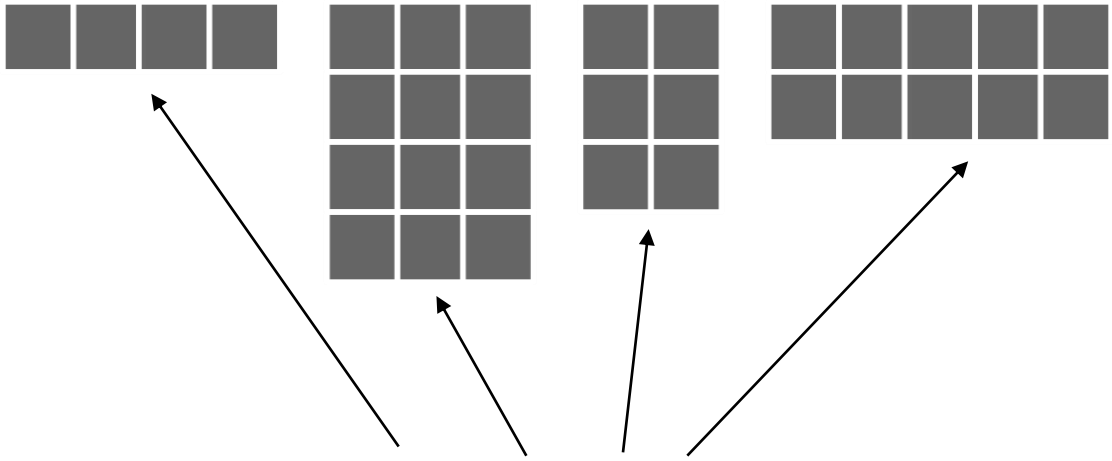
$$y = \mathcal{L}(\boldsymbol{x}) = (A \circ B \circ C \circ D)(\boldsymbol{x}) \equiv A(B(C(D(\boldsymbol{x}))))$$

$$y = A(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{d}), \quad \boldsymbol{d} = D(\boldsymbol{x})$$

$$\mathcal{L}'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \frac{\partial \boldsymbol{d}}{\partial \boldsymbol{x}}$$



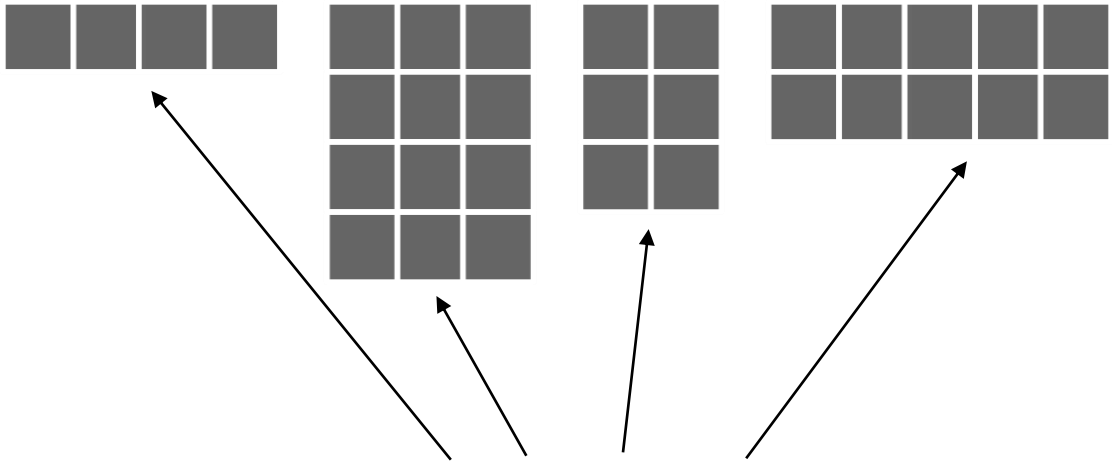
Multivariate autodiff (*forward accumulation*)



$$\mathcal{L}'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{b}} \left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{c}} \underbrace{\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \frac{\partial \boldsymbol{d}}{\partial \boldsymbol{x}} \right)} \right)$$

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial c_1}{\partial x_1} & \cdots & \frac{\partial c_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial c_m}{\partial x_1} & \cdots & \frac{\partial c_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

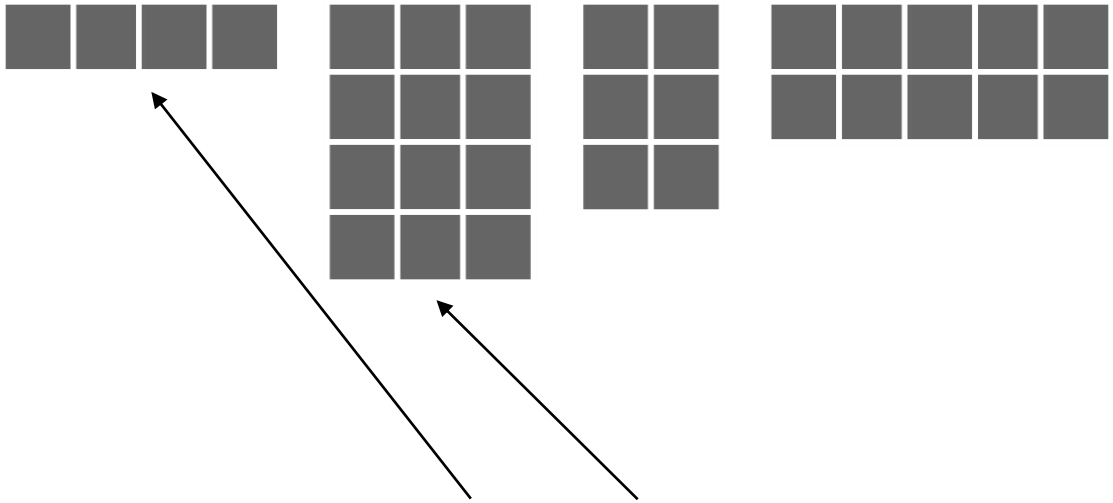
Multivariate autodiff (*reverse accumulation*)



$$\mathcal{L}'(\boldsymbol{x}) = \left(\underbrace{\left(\frac{\partial y}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{c}} \right)}_{} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \right) \frac{\partial \boldsymbol{d}}{\partial \boldsymbol{x}}$$

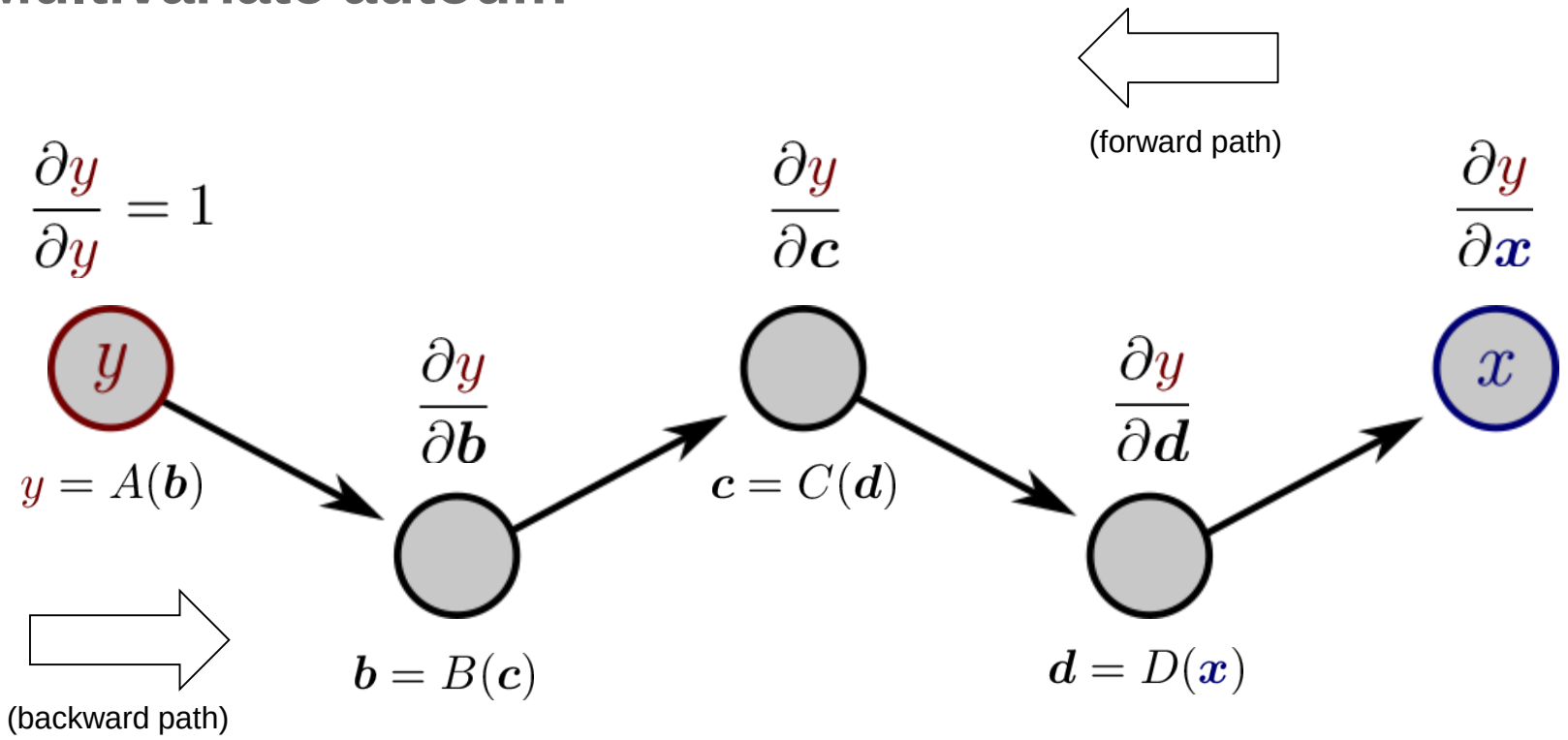
$$\frac{\partial y}{\partial \boldsymbol{c}} = \left(\frac{\partial y}{\partial c_1}, \cdots, \frac{\partial y}{\partial c_{m'}} \right) \in \mathbb{R}^{1 \times m'}$$

Multivariate autodiff (*reverse accumulation*)



More generally: this is a **vector-Jacobian** product (`vjp`)

Multivariate autodiff



$$\mathcal{L}'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \frac{\partial \boldsymbol{d}}{\partial \boldsymbol{x}}$$

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \equiv C'(\boldsymbol{d}) \equiv \left. \frac{\partial C(\tilde{\boldsymbol{d}})}{\partial \tilde{\boldsymbol{d}}} \right|_{\tilde{\boldsymbol{d}}=\boldsymbol{d}}$$

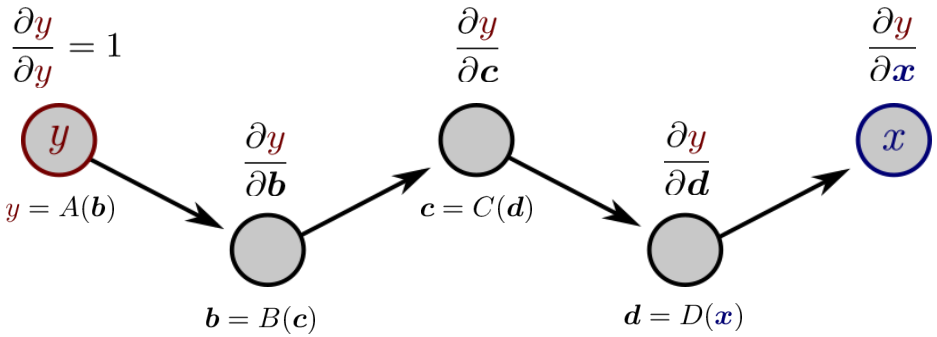
Multivariate autodiff

Forward (at each node)**:**

- Evaluate function to find y
- Save intermediate results

Backward (at each node)**:**

- Evaluate derivative using results from forward path
- Multiply with gradient of previous node



$$\mathcal{L}'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \frac{\partial \boldsymbol{d}}{\partial \boldsymbol{x}}$$

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{d}} \equiv C'(\boldsymbol{d}) \equiv \left. \frac{\partial C(\tilde{\boldsymbol{d}})}{\partial \tilde{\boldsymbol{d}}} \right|_{\tilde{\boldsymbol{d}}=\boldsymbol{d}}$$

Question

What about loops and branches?

They are **unrolled!**

PyTorch

An engine for computing `vjp`s

PyTorch

- Library around a fork of `autograd` (wrapper of `numpy` function)
- Core data structure: `torch.tensor`
  - Akin to `np.array`
  - Default data type: `float32`
  - Different devices

```
x1 = torch.tensor(2.0, requires_grad=True)
x2 = torch.tensor(3.0, requires_grad=True)

y = x1 * (x1 + x2)
y.backward()
print("(1):", x1.grad, x2.grad)
```

Output:

```
(1): tensor(7.), tensor(2.)
```

```
>>> x = np.random.rand(4, 2, 3)
>>> sel = [False, False, True]
>>> x[0, :, sel]
array([[0.7545524 , 0.85684194]])
>>> torch.tensor(x)[0, :, sel]
tensor([[0.7546],
        [0.8568]], dtype=torch.float64)
```

DLR

PyTorch

```
x1 = torch.tensor(2.0, requires_grad=True)
x2 = torch.tensor(3.0, requires_grad=True)

y = x1 * (x1 + x2)
y.backward()
print("(1):", x1.grad, x2.grad)

y.backward()
print("(2):", x1.grad, x2.grad)
```

RuntimeError: Trying to backward through the graph a second time (or directly access saved variables after they have already been freed). Saved intermediate values of the graph are freed when you call `.backward()` or `autograd.grad()`. **Specify `retain_graph=True` if you need to backward** through the graph a second time or if you need to access saved variables after calling backward.

PyTorch

```
x1 = torch.tensor(2.0, requires_grad=True)
x2 = torch.tensor(3.0, requires_grad=True)

y = x1 * (x1 + x2)
y.backward()
print("(1):", x1.grad, x2.grad)

y = x1 * (x1 + x2)
y.backward()
print("(2):", x1.grad, x2.grad)
```

Output:

```
(1): tensor(7.), tensor(2.)
(2): tensor(14.), tensor(4.)
```

PyTorch

```
x1 = torch.tensor(2.0, requires_grad=True)
x2 = torch.tensor(3.0, requires_grad=True)

y = x1 * (x1 + x2)
y.backward()
print("(1):", x1.grad, x2.grad)

x1.grad.zero_()
x2.grad.zero_()
y = x1 * (x1 + x2)
y.backward()
print("(2):", x1.grad, x2.grad)
```

Code smell!

Output:

```
(1): tensor(7.), tensor(2.)
(2): tensor(7.), tensor(2.)
```

A new PyTorch module

```python
import torch
import torch.nn as nn


class NN(nn.Module):
    def __init__(self, n: int):
        super().__init__()
        self.weights = nn.Parameter(torch.rand(n)) ** 2

    def forward(self, x):
        return x * self.weights
```

```python
if __name__ == "__main__":
    f = NN(5)
    x = torch.rand(5)

    f.zero_grad()
    y = f(x)
    y.backward()

    f.zero_grad()
    y = f(x)
    y.backward()
```

RuntimeError: grad can be implicitly created only for scalar outputs

A new PyTorch module

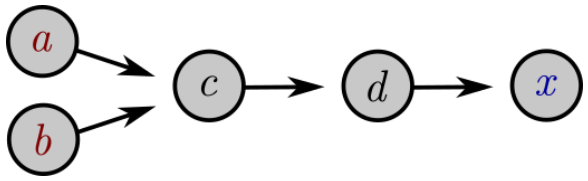
```python
import torch
import torch.nn as nn


class NN(nn.Module):
    def __init__(self, n: int):
        super().__init__()
        self.weights = nn.Parameter(torch.rand(n)) ** 2

    def forward(self, x):
        return x * self.weights
```

```python
if __name__ == "__main__":
    f = NN(5)
    x = torch.rand(5)

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()
```

RuntimeError: Trying to backward through the graph a second time […]

Excursus: backward's retain_graph

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    d = x ** 2
    c = d * 2.0
    b = c.mean()
    a = c.sum()
```



$$a = \sum_{i=1}^{n} 2x_i^2 \equiv n\,b \quad \rightarrow \quad \frac{\partial a}{\partial x_j} = 4x_j$$

What is the output of this program?

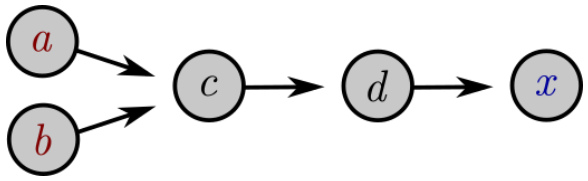
```python
    b.backward(retain_graph=True)
    grads_exp = 4.0 * x.detach() / n
    print("(1)", torch.allclose(x.grad, grads_exp))

    a.backward()
    print("(2)", torch.allclose(x.grad, n * grads_exp))
```

Output:

```
(1) True
(2) False
```

Excursus: `backward`'s `retain_graph`

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    d = x ** 2
    c = d * 2.0
    b = c.mean()
    a = c.sum()
```



$$a = \sum_{i=1}^{n} 2x_i^2 \equiv n\,b \quad \rightarrow \quad \frac{\partial a}{\partial x_j} = 4x_j$$

What is the output of this program?

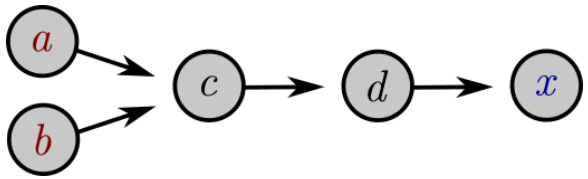
```python
    b.backward(retain_graph=True)
    grads = x.grad.clone()
    grads_exp = 4.0 * x.detach() / n
    print("(1)", torch.allclose(grads, grads_exp))

    a.backward()
    print("(2)", torch.allclose(x.grad - grads, n * grads_exp))
```

Output:

```
(1) True
(2) True
```

Excursus: `backward`'s `retain_graph`

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    d = x ** 2
    c = d * 2.0
    b = c.mean()
    a = c.sum()
```



$$a = \sum_{i=1}^{n} 2x_i^2 \equiv n\,b \quad \rightarrow \quad \frac{\partial a}{\partial x_j} = 4x_j$$

What is the output of this program?

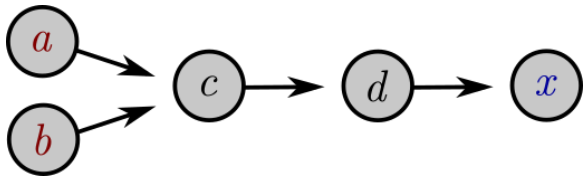
```python
    b.backward(retain_graph=True)
    grads_exp = 4.0 * x.detach() / n
    print("(1)", torch.allclose(x.grad, grads_exp))

    x.grad.zero_()
    a.backward()
    print("(2)", torch.allclose(x.grad, n * grads_exp))
```

Output:

```
(1) True
(2) True
```

Excursus: `backward`'s `retain_graph`

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    d = x ** 2
    c = d * 2.0
    b = c.mean()
    a = c.sum()
```



$$a = \sum_{i=1}^{n} 2x_i^2 \equiv n\,b \quad \rightarrow \quad \frac{\partial a}{\partial x_j} = 4x_j$$

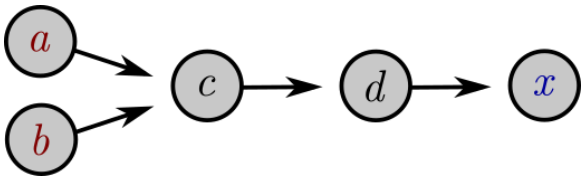
Still, `retain_graph` is a code smell...

```python
b.backward(retain_graph=True)
a.backward(retain_graph=True)
grads = x.grad.clone()

x.grad.zero_()
loss = a + b
loss.backward()
assert torch.allclose(x.grad, grads)
```

… and there are alternatives

Excursus: backward's retain_graph

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    d = x ** 2
    c = d * 2.0
    b = c.mean()
    a = c.sum()
```



$$a = \sum_{i=1}^{n} 2x_i^2 \equiv n\,b \quad \rightarrow \quad \frac{\partial a}{\partial x_j} = 4x_j$$

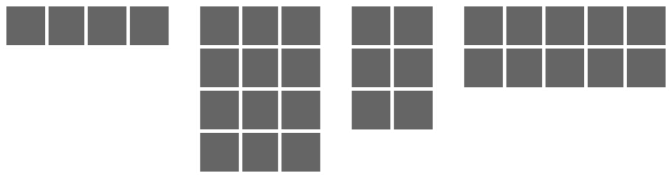
$$a = \sum_{i} c_i \quad \rightarrow \quad \left(\frac{\partial a}{\partial \boldsymbol{c}}\right)_i \equiv \frac{\partial a}{\partial c_i} = 1$$

… and there are alternatives

```python
a.backward(retain_graph=True)
grads_exp = x.grad.clone()

x.grad.zero_()
c.backward(gradient=torch.ones_like(x))
assert torch.allclose(x.grad, grads_exp)
```

$$\frac{\partial a}{\partial \boldsymbol{c}} = (1, \ldots, 1)$$

(In theory: Build Jacobian one row at a time with one-hot vectors)



A new PyTorch module

```python
import torch
import torch.nn as nn


class NN(nn.Module):
    def __init__(self, n: int):
        super().__init__()
        self.weights = nn.Parameter(torch.rand(n)) ** 2

    def forward(self, x):
        return x * self.weights
```

```python
if __name__ == "__main__":
    f = NN(5)
    x = torch.rand(5)

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()
```

RuntimeError: Trying to backward through the graph a second time […]

A new PyTorch module

```python
import torch
import torch.nn as nn


class NN(nn.Module):
    def __init__(self, n: int):
        super().__init__()
        self.weights = nn.Parameter(torch.rand(n))

    def forward(self, x):
        return x * (self.weights ** 2)
```

```python
if __name__ == "__main__":
    f = NN(5)
    x = torch.rand(5)

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()
```

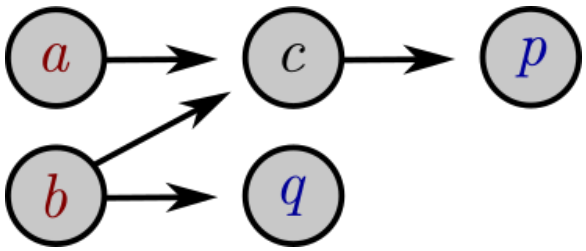
A new PyTorch module

```python
import torch
import torch.nn as nn


class NN(nn.Module):
    def __init__(self, n: int):
        super().__init__()
        self.weights = nn.Parameter(torch.rand(n) ** 2)

    def forward(self, x):
        return x * self.weights
```

```python
if __name__ == "__main__":
    f = NN(5)
    x = torch.rand(5)

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()

    f.zero_grad()
    y = torch.sum(f(x))
    y.backward()
```

detach() & clone()

$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$

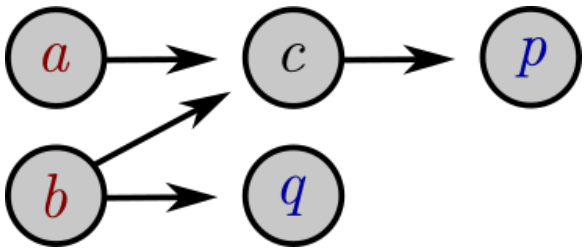



```
6   p_val, q_val = 2.0, 3.0
7   p = torch.tensor(p_val, requires_grad=True)
8   q = torch.tensor(q_val, requires_grad=True)
9
10  c = 2 * p
11  b = c * q
12  a = c ** 2
13
14  a.backward()
15  b.backward()
16
17  assert np.isclose(p.grad.item(), 8 * p_val)
18  assert np.isclose(q.grad.item(), 2 * p_val)
```

RuntimeError: Trying to backward through the graph a second time […]

detach() & clone()

$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$



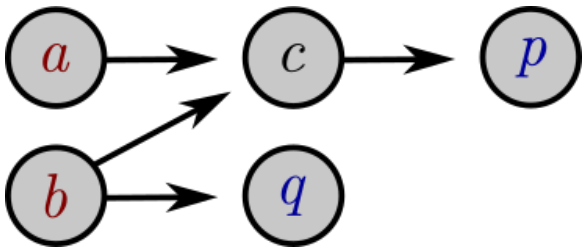

```
6   p_val, q_val = 2.0, 3.0
7   p = torch.tensor(p_val, requires_grad=True)
8   q = torch.tensor(q_val, requires_grad=True)
9
10  c = 2 * p
11  c_cpy = c.detach()
12  b = c_cpy * q
13  a = c ** 2
14
15  a.backward()
16  b.backward()
17
18  assert np.isclose(p.grad.item(), 8 * p_val)
19  assert np.isclose(q.grad.item(), 2 * p_val)
20
21  print("(1)", c_cpy)
22  c.mul_(2)
23  print("(2)", c_cpy)
```

Output:

```
(1) tensor(4.)
(2) tensor(8.)
```

detach() & clone()

$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$



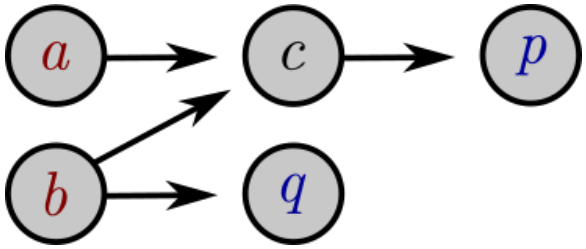

```
6   p_val, q_val = 2.0, 3.0
7   p = torch.tensor(p_val, requires_grad=True)
8   q = torch.tensor(q_val, requires_grad=True)
9
10  c = 2 * p
11  c_cpy = c.detach().clone()
12  b = c_cpy * q
13  a = c ** 2
14
15  a.backward()
16  b.backward()
17
18  assert np.isclose(p.grad.item(), 8 * p_val)
19  assert np.isclose(q.grad.item(), 2 * p_val)
20
21  print("(1)", c_cpy)
22  c.mul_(2)
23  print("(2)", c_cpy)
```

Output:

```
(1) tensor(4.)
(2) tensor(4.)
```

detach() & clone()

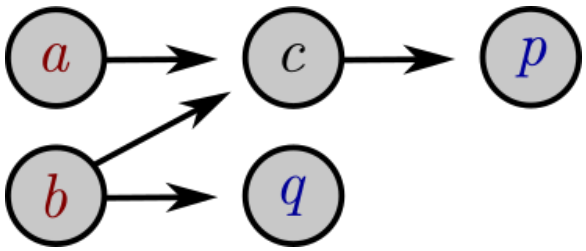
$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$




```
6   p_val, q_val = 2.0, 3.0
7   p = torch.tensor(p_val, requires_grad=True)
8   q = torch.tensor(q_val, requires_grad=True)
9
10  c = 2 * p
11  c.mul_(1.0)
12  c_cpy = c.detach()
13  b = c_cpy * q
14  a = c ** 2
15
16  a.backward()
17  b.backward()
18
19  assert np.isclose(p.grad.item(), 8 * p_val)
20  assert np.isclose(q.grad.item(), 2 * p_val)
```

detach() & clone()

$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$

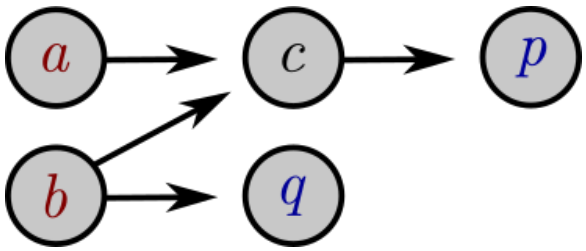



```
6     p_val, q_val = 2.0, 3.0
7     p = torch.tensor(p_val, requires_grad=True)
8     q = torch.tensor(q_val, requires_grad=True)
9
10    c = 2 * p
11    c_cpy = c.detach()
12    b = c_cpy * q
13    c.mul_(1.0)
14    a = c ** 2
15
16    a.backward()
17    b.backward()
18
19    assert np.isclose(p.grad.item(), 8 * p_val)
20    assert np.isclose(q.grad.item(), 2 * p_val)
```

RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation […]

detach() & clone()

$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$

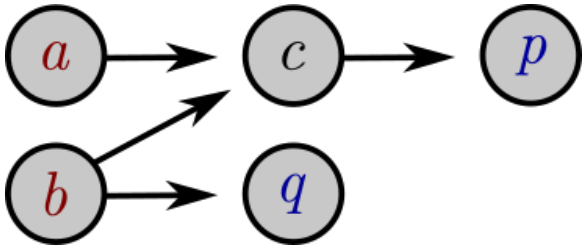



```
6     p_val, q_val = 2.0, 3.0
7     p = torch.tensor(p_val, requires_grad=True)
8     q = torch.tensor(q_val, requires_grad=True)
9
10    c = 2 * p
11    c_cpy = c.detach()
12    b = c_cpy * q
13    c *= 1.0
14    a = c ** 2
15
16    a.backward()
17    b.backward()
18
19    assert np.isclose(p.grad.item(), 8 * p_val)
20    assert np.isclose(q.grad.item(), 2 * p_val)
```

RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation […]

detach() & clone()

$$c = 2p \qquad \frac{\partial a}{\partial p} = 8p$$
$$b = qc \qquad \frac{\partial b}{\partial q} = 2p$$
$$a = c^2$$




```
6     p_val, q_val = 2.0, 3.0
7     p = torch.tensor(p_val, requires_grad=True)
8     q = torch.tensor(q_val, requires_grad=True)
9
10    c = 2 * p
11    c_cpy = c.detach()
12    b = c_cpy * q
13    c = c * 1.0
14    a = c ** 2
15
16    a.backward()
17    b.backward()
18
19    assert np.isclose(p.grad.item(), 8 * p_val)
20    assert np.isclose(q.grad.item(), 2 * p_val)
```

Rule of thumb: avoid in-place operations and be explicit when doing overwrites.

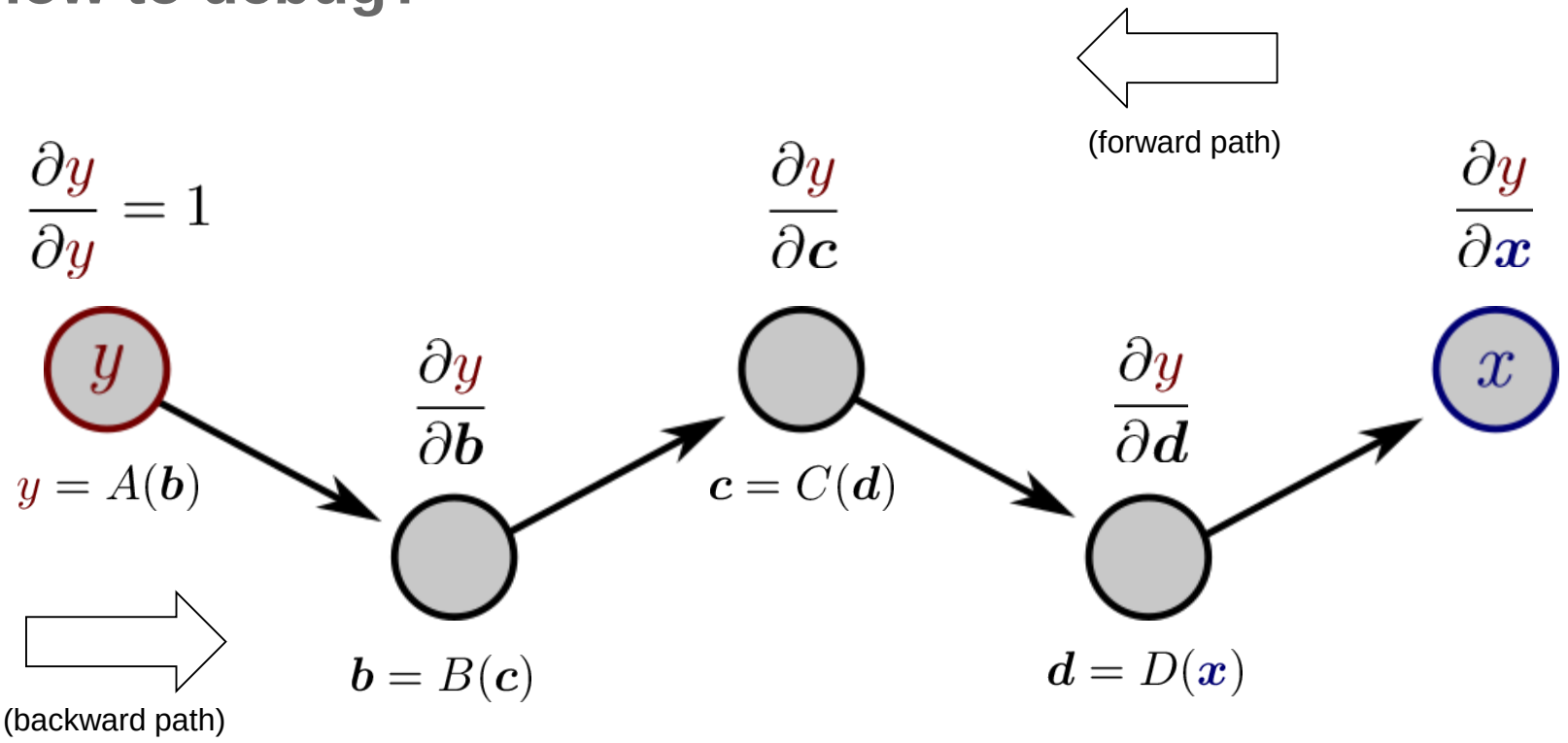
!!! WARNING: indexing is also in-place !!!

- Avoid indexing whenever possible:
  - Select and indexing yield views
  - There are in-place correctness checks but they are not perfect
- That is:
  - `… = x[i]  #` **code smell**
  - `x[i] = …  #` **code smell**
  - Be explicit and use `torch.cat` if necessary

```
5   class MyFancyModel(nn.Module):
6       def __init__(self):
7           super().__init__()
8
9       def forward(self, x):
10          y = torch.empty(3)
11          y[:2] = x
12          y[2] = x[0] * y[0]
13          return y[0]
14
15
16  if __name__ == "__main__":
17      x = torch.rand(2, requires_grad=True)
18
19      model = MyFancyModel()
20      y = model(x)
21      y.backward()
```

RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation […]

How to debug?



Forward path: `print`
Backward path: **hooks**

Inspecting the backward path

$$\frac{\partial y}{\partial b_i} = \frac{\partial}{\partial b_i}\left(\sum_i b_i\right) = 1 \qquad \frac{\partial y}{\partial \boldsymbol{b}} = (1, \ldots, 1)$$

$$\frac{\partial b_i}{\partial a_j} = \frac{\partial}{\partial a_j}(2a_i) = 2\,\delta_{ij} \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = \begin{pmatrix} 2 & & \\ & \ddots & \\ & & 2 \end{pmatrix}$$

$$\frac{\partial a_i}{\partial x_j} = \frac{\partial}{\partial x_j}(x_i^2) = 2x_i\,\delta_{ij} \qquad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \begin{pmatrix} 2x_1 & & \\ & \ddots & \\ & & 2x_n \end{pmatrix}$$

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    a = x ** 2
    b = a * 2.0
    y = b.sum()

    a.retain_grad()
    b.retain_grad()
    y.retain_grad()

    y.backward()
    print("grad of y:", y.grad)
    print("grad of b:", b.grad)
    print("grad of a:", a.grad)
    print("grad of x:", x.grad)
    print("Exp. grad of x:", x * 4)
```

$$a_i = x_i$$
$$b_i = 2a_i$$
$$y = \sum_i b_i$$

```
grad of y: tensor(1.)
grad of b: tensor([1., 1., 1., 1.])
grad of a: tensor([2., 2., 2., 2.])
grad of x: tensor([2.28, 3.58, 2.06, 3.20])
Exp. grad of x: tensor([2.28, 3.58, 2.06, 3.20])
```

$$\frac{\partial y}{\partial \boldsymbol{x}} = \frac{\partial y}{\partial \boldsymbol{b}}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = (1, \ldots, 1)\begin{pmatrix} 2 & & \\ & \ddots & \\ & & 2 \end{pmatrix}\begin{pmatrix} 2x_1 & & \\ & \ddots & \\ & & 2x_n \end{pmatrix} = (4x_1, \ldots 4x_n)$$

Inspecting the backward path

```python
if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    a = x ** 2
    b = a * 2.0
    y = b.sum()

    a.retain_grad()
    b.retain_grad()
    y.retain_grad()

    y.backward()
    print("grad of y:", y.grad)
    print("grad of b:", b.grad)
    print("grad of a:", a.grad)
    print("grad of x:", x.grad)
    print("Exp. grad of x:", x * 4)
```

```python
def hook(msg, grad):
    print(msg, grad)


def register_hooks(**kwargs):
    for k in kwargs:
        v = kwargs[k]

        msg = f"grad of {k}:"
        v.register_hook(functools.partial(hook, msg))


if __name__ == "__main__":
    n = 4
    x = torch.rand(n, requires_grad=True)
    a = x ** 2
    b = a * 2.0
    y = b.sum()

    register_hooks(y=y, b=b, a=a, x=x)

    y.backward()
    print("Exp. grad of x:", x * 4)
```

- Hooks are called **each time** the node is called **during back propagation**
- Hooks are allow to change gradients

Inspecting the backward path

$$\frac{\partial y}{\partial b_i} = \frac{\partial}{\partial b_i}\left(\sum_i b_i\right) = 1 \qquad \frac{\partial y}{\partial \boldsymbol{b}} = (1, \ldots, 1)$$

$$\frac{\partial b_i}{\partial a_j} = \frac{\partial}{\partial a_j}(2a_i) = 2\,\delta_{ij} \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = \begin{pmatrix} 2 & & \\ & \ddots & \\ & & 2 \end{pmatrix}$$

$$\frac{\partial a_i}{\partial x_j} = \frac{\partial}{\partial x_j}(x_i^2) = 2x_i\,\delta_{ij} \qquad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \begin{pmatrix} 2x_1 & & \\ & \ddots & \\ & & 2x_n \end{pmatrix}$$

```python
class MyFancyModel(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return x ** 2


def make_hook(msg):
    return lambda *args: print(f"{msg}:", *args)


if __name__ == "__main__":
    f = MyFancyModel()
    f.register_forward_hook(make_hook("fwd"))
    f.register_full_backward_hook(make_hook("bwd"))

    x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
    a = f(x)
    b = a * 2.0
    y = b.sum()

    y.backward()
```

Output:

```
fwd: MyFancyModel() (tensor([1., 2., 3.],) tensor([1., 4., 9.],)
bwd: MyFancyModel() (tensor([ 4.,  8., 12.]),) (tensor([2., 2., 2.]),)
```

Custom `autograd` functions

$$
\begin{aligned}
a &= 2x \\
b &= 3x \\
f &= a \sin b \\
y &= f^2
\end{aligned}
$$

```python
class MyFancyFun(torch.autograd.Function):
    @staticmethod
    def forward(ctx, a, b):
        ctx.save_for_backward(a, b)
        return a * torch.sin(b)

    @staticmethod
    def backward(ctx, grad):
        a, b = ctx.saved_tensors
        return torch.sin(b) * grad, a * torch.cos(b) * grad
```

$$
\begin{aligned}
\frac{\partial y}{\partial x} &= \frac{\partial y}{\partial f} \left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b} \right) \begin{pmatrix} \frac{\partial a}{\partial x} \\ \frac{\partial b}{\partial x} \end{pmatrix} \\
&= 2f \ (\sin b, \ a \cos b) \begin{pmatrix} 2 \\ 3 \end{pmatrix}
\end{aligned}
$$

```python
if __name__ == "__main__":
    fun = MyFancyFun().apply

    x = torch.tensor([0.5], requires_grad=True)
    a = 2.0 * x
    b = 3.0 * x
    f = fun(a, b)
    y = f ** 2
    y.backward()

    x_val = x.detach().clone()
    s = torch.sin(3.0 * x_val)
    c = torch.cos(3.0 * x_val)
    grad_exp = 4 * x_val * s * (2.0 * s + 6.0 * x_val * c)
    assert torch.isclose(x.grad, grad_exp)
```

Custom `autograd` functions

$$a = 2x$$
$$b = 3x$$
$$f = a \sin b$$
$$y = f^2$$

```python
class MyFancyFun(torch.autograd.Function):
    @staticmethod
    def forward(ctx, a, b):
        res = a * torch.sin(b)
        ctx.save_for_backward(a, b, res)
        return res

    @staticmethod
    def backward(ctx, grad):
        a, b, res = ctx.saved_tensors
        sin_b = res / a
        sign = torch.where(torch.abs(b) < math.pi / 2.0, +1.0, -1.0)
        return sin_b * grad, sign * a * torch.sqrt(1.0 - sin_b ** 2) * grad
```

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial f} \left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b} \right) \begin{pmatrix} \frac{\partial a}{\partial x} \\ \frac{\partial b}{\partial x} \end{pmatrix}$$
$$= 2f \ (\sin b, \ a \cos b) \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

```python
if __name__ == "__main__":
    fun = MyFancyFun().apply

    x = torch.tensor([0.5], requires_grad=True)
    a = 2.0 * x
    b = 3.0 * x
    f = fun(a, b)
    y = f ** 2
    y.backward()

    x_val = x.detach().clone()
    s = torch.sin(3.0 * x_val)
    c = torch.cos(3.0 * x_val)
    grad_exp = 4 * x_val * s * (2.0 * s + 6.0 * x_val * c)
    assert torch.isclose(x.grad, grad_exp)
```

Thank you for your attention!