

AD in PyTorch

Breandan Considine

February 5, 2021



What is automatic differentiation?

- ▶ AD is a higher-order function which either:
 - ▶ Accepts a function and returns a second function which gives, when evaluated, the sensitivity at *any location*, $AD : (f) \mapsto f'$
 - ▶ Accepts a function and an input to be evaluated, and evaluates the function and its derivative, *at that specific location*
 $AD : (f, x) \mapsto (f(x), f'(x))$
- ▶ These two views correspond to static and dynamic AD
 - ▶ For mathematical expressions, the static approach is preferred
 - ▶ For computer programs, the dynamic approach is preferred

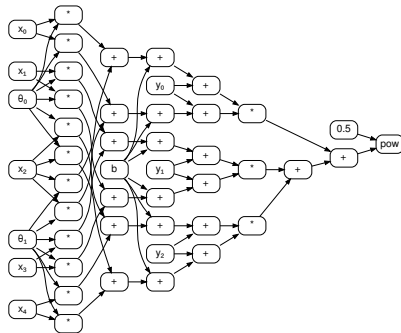
Static and Dynamic Representations

Programs are dynamical systems, graphs are a static objects.

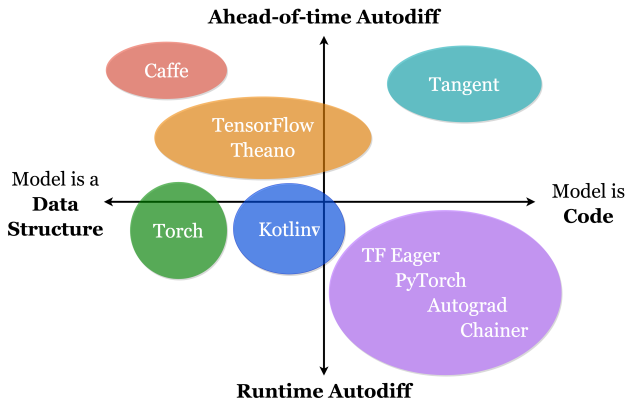
Program

```
sum = 0
l = [0, 0, 0, 0]
for i in range(0, 4):
    l[i] += t[i] * x[i]
for i in range(0, 4):
    l[i] -= y[i] - b
for i in range(0, 4):
    l[i] *= l[i]
for i in range(0, 4):
    sum += l[i]
l = sqrt(sum)
```

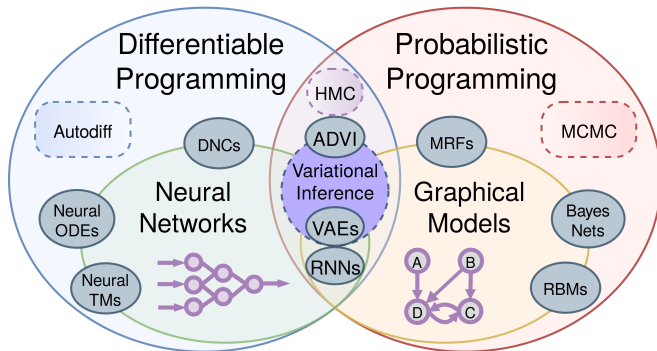
Computation Graph



Dimensions of AD frameworks



Applications of Automatic Differentiation



What is automatic differentiation?

To understand AD, you just need to remember two simple rules:

$$D(f + g) = D(f) + D(g)$$

$$D(f \cdot g) = D(f) \cdot g + f \cdot D(g)$$

We can think of AD as a *linear map* between function spaces.

$$D(f + g) = D(f) + D(g)$$

$$\alpha D(f) = D(\alpha f)$$

Picrograd / PyTorch in a single slide

```
class Var:
    def __init__(self, val, grad_fn=lambda: []):
        self.v, self.grad_fn = val, grad_fn

    def __add__(self, other):
        return Var(self.v + other.v,
                    lambda: [(self, 1.0), (other, 1.0)])

    def __mul__(self, other):
        return Var(self.v * other.v,
                    lambda: [(self, other.v), (other, self.v)])

    def grad(self, bp = 1.0, dict = {}):
        dict[self] = dict.get(self, 0) + bp
        for input, val in self.grad_fn():
            input.grad(val * bp, dict)
        return dict
```

Higher order and higher rank AD

The *gradient*, $\nabla : (\mathbb{R}^m \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^m$ maps a function Q to:

$$\nabla Q(q_1, \dots, q_m) = \left[\frac{\partial Q}{\partial q_1}, \dots, \frac{\partial Q}{\partial q_m} \right]$$

The *Jacobian*, $\mathcal{J} : (\mathbb{R}^m \rightarrow \mathbb{R}^n) \rightarrow \mathbb{R}^{n \times m}$ is a matrix of partials:

$$\mathcal{J} \circ \mathbf{f} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \dots & \frac{\partial \mathbf{f}}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_m \end{bmatrix}$$

Higher order and higher rank AD

Suppose we have a function $P(X) = p_k \circ p_{k-1} \circ \dots \circ p_1 \circ X$. The derivative of a linear composition can be expressed as a product:

$$\frac{dP}{dp_1} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \dots \frac{dp_2}{dp_1} = \prod_{i=1}^{k-1} \frac{dp_{i+1}}{dp_i}$$

This also holds in higher dimensions, for example $\mathbf{P}_k : \mathbb{R}^m \rightarrow \mathbb{R}^n$:

$$\begin{aligned} \mathcal{J}\mathbf{P}_k &= \prod_{i=1}^k \mathcal{J}p_i = \underbrace{\left(\left((\mathcal{J}p_k \mathcal{J}p_{k-1}) \dots \mathcal{J}p_2 \right) \mathcal{J}p_1 \right)}_{\text{Reverse mode, VJP, Pullback}} \\ &= \underbrace{\left(\mathcal{J}p_k \left(\mathcal{J}p_{k-1} \dots (\mathcal{J}p_2 \mathcal{J}p_1) \right) \right)}_{\text{Forward mode, JVP, Pushforward}} \end{aligned}$$

Gradients in PyTorch

Suppose we have a scalar-valued vector function, $f : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$:

```
x = torch.randn(2, requires_grad=True)
t = torch.tensor([2., 3.], requires_grad=True)
y = torch.randn(2)
f = sum((x*t - y)**2)**0.5
```

```
torch.autograd.grad(f, inputs=(x, t))
```

```
(tensor([-0.60,  2.85]), tensor([0.23, 1.47]))
```

What is a vectorizing map? (vmap)

Suppose we have a scalar-valued function, $f : \mathbb{R} \rightarrow \mathbb{R}$. **vmap** is a function which takes f and returns a function $g : \mathbb{R}^* \rightarrow \mathbb{R}^*$ that accepts a tensor $t : \mathbb{R}^*$, and maps f over the tensor, elementwise.

```
function = lambda x: x**2 + x
tensor = torch.ones(3, 3, 3) * 2
vmap(function)(tensor)
```

```
torch.dot          # [ D ], [ D ] -> S
vd = vmap(torch.dot) # [ N,D ], [ N,D ] -> [ N ]
vvd = vmap(vd)      # [N,D,C], [N,D,C] -> [N,D]
x, y = torch.ones(3, 2, 5), torch.ones(3, 2, 5)
vvd(x, y)
```

Jacobians in PyTorch

```
def jacobian(fun, x) -> torch.Tensor:
    x = x.detach().requires_grad_()
    y = fun(x)
    vjp = lambda v: torch.autograd.grad(y, x, v)[0]
    vs = torch.eye(y.numel())\
        .view(y.numel(), *y.shape)
    result = vmap(vjp)(vs)
    return result.detach()
```

```
f = lambda x: x ** 3
jacobian(f, torch.ones(3))
```

```
tensor([[3., 0., 0.],
        [0., 3., 0.],
        [0., 0., 3.]])
```

Higher order and higher rank AD

The *Hessian* $\mathbf{H} : (\mathbb{R}^m \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^{m \times m}$ maps scalar functions to ∂^2 :

$$\mathbf{H}(Q) = \begin{bmatrix} \frac{\partial^2 Q}{\partial x_1^2} & \frac{\partial^2 Q}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 Q}{\partial x_1 \partial x_m} \\ \frac{\partial^2 Q}{\partial x_2 \partial x_1} & \frac{\partial^2 Q}{\partial x_2^2} & \cdots & \frac{\partial^2 Q}{\partial x_2 \partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 Q}{\partial x_m \partial x_1} & \frac{\partial^2 Q}{\partial x_m \partial x_2} & \cdots & \frac{\partial^2 Q}{\partial x_m^2} \end{bmatrix}$$

The Hessian and Jacobian are related by $\mathbf{H}(Q)^\top = \mathcal{J} \circ \nabla Q$.

Hessians in PyTorch

```
def hessian(fun, x) -> torch.Tensor:
    def grad0(x: torch.Tensor):
        y = fun(x)
        assert y.dim() == 0
        return torch.autograd.grad(y, x,
                                     create_graph=True)[0]
    return jacobian(grad0, x)

g = lambda x: (x ** 3).sum()
hessian(g, torch.ones(3))
```

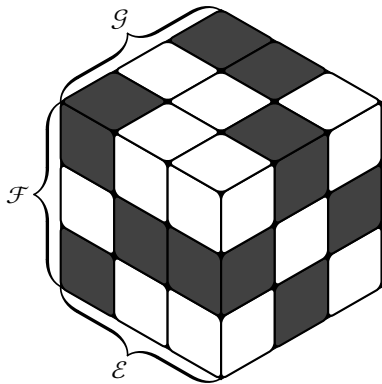
```
tensor([[6., 0., 0.],
        [0., 6., 0.],
        [0., 0., 6.]])
```

What is a tensor?

Rank-2

$$\begin{bmatrix} 1 & 0 & \dots & 1 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 1 \end{bmatrix}$$

Rank-3



Checking matrix multiplication

Suppose we have two tensors, $A : \mathbb{R}^{x \times y \times \dots}$ and $B : \mathbb{R}^{y \times z \times \dots}$
Then $C = A @ B$ has type $C : \mathbb{R}^{x \times z \times \dots}$. For example:

```
state = torch.ones(9, 5, names=('batch', 'D'))  
trans = torch.randn(5, 5, names=('in', 'out'))  
next_state = state @ trans  
print(next_state.names)
```

('batch', 'out')

Runtime type checking: name mismatch

What happens if we try to sum dimensions with different names?

```
x = torch.ones(3, names=('X',))
y = torch.ones(3, names=('Z',))
z = x + y
```

----**RuntimeError**

Traceback (most recent call last)[...]

```
2 x = torch.ones(3, names=('X',))
```

```
3 y = torch.ones(3, names=('Y',))
```

```
----> 4 xpz = x + z
```

RuntimeError: Error attempting to broadcast
dims ['X'] and dims ['Y']: dim 'X' and dim 'Y'
are at the same position from the right