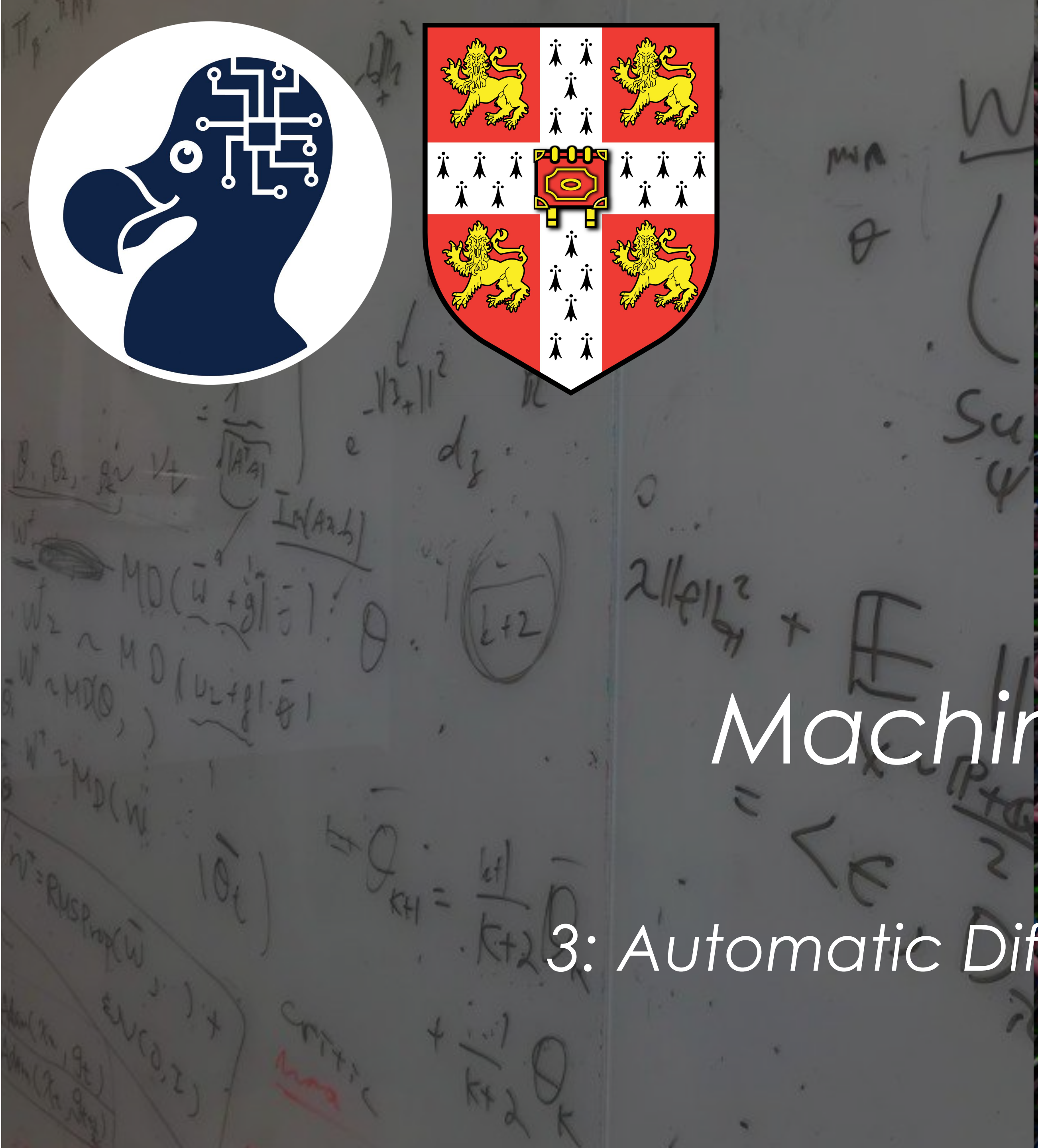
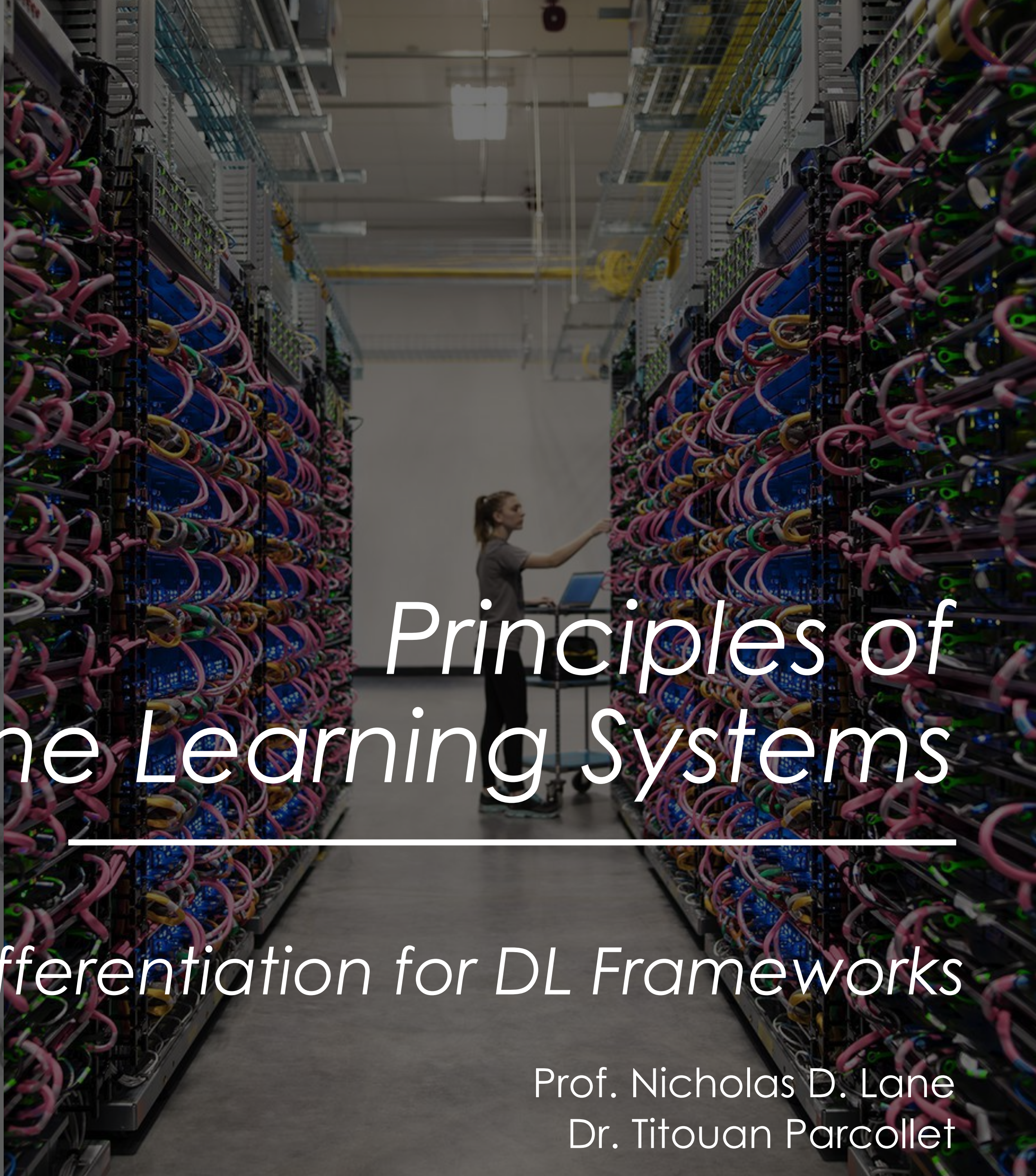



Principles of Machine Learning Systems

3: Automatic Differentiation for DL Frameworks

Prof. Nicholas D. Lane
Dr. Titouan Parcollet

Roadmap for Today

1. Why do we care about automatic differentiation?
2. The different types of differentiations.
3. Forward and reverse mode automatic differentiation.
4. Automatic differentiation in PyTorch.

Roadmap for Today

1. **Why do we care about automatic differentiation?**
2. The different types of differentiations.
3. Forward and reverse mode automatic differentiation.
4. Automatic differentiation in PyTorch.

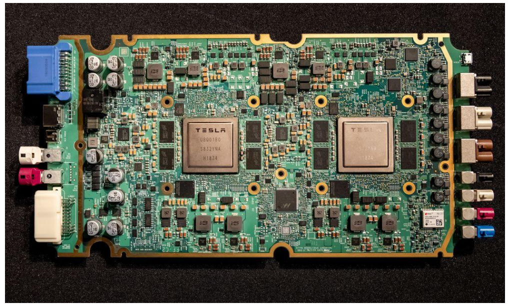
Why do we care about automatic differentiation (AD)?

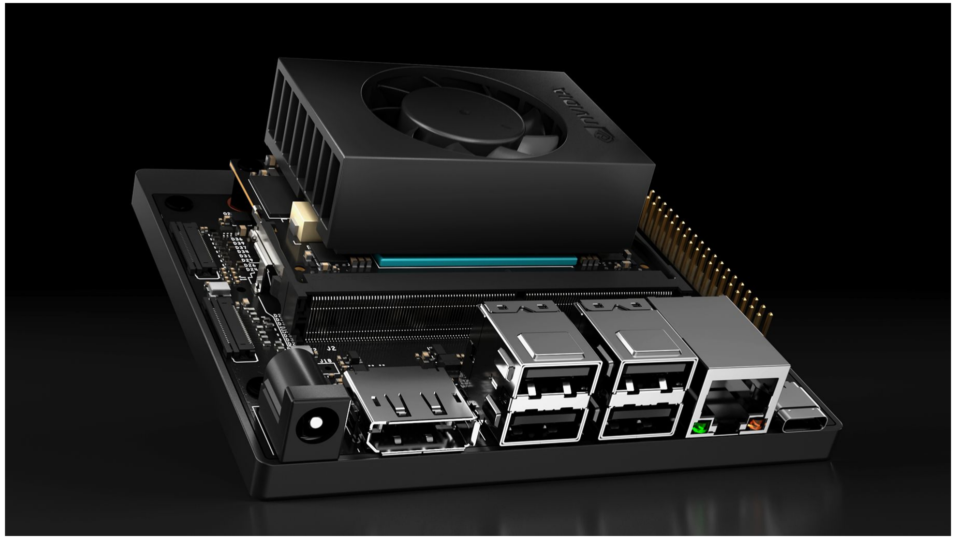
Most deep learning framework rely on automatic differentiation.
(In general, on the reverse mode of AD)

Understanding AD helps understanding them.

Why do we care about automatic differentiation (AD)?

Why do we care about AD for Machine Learning and Systems?

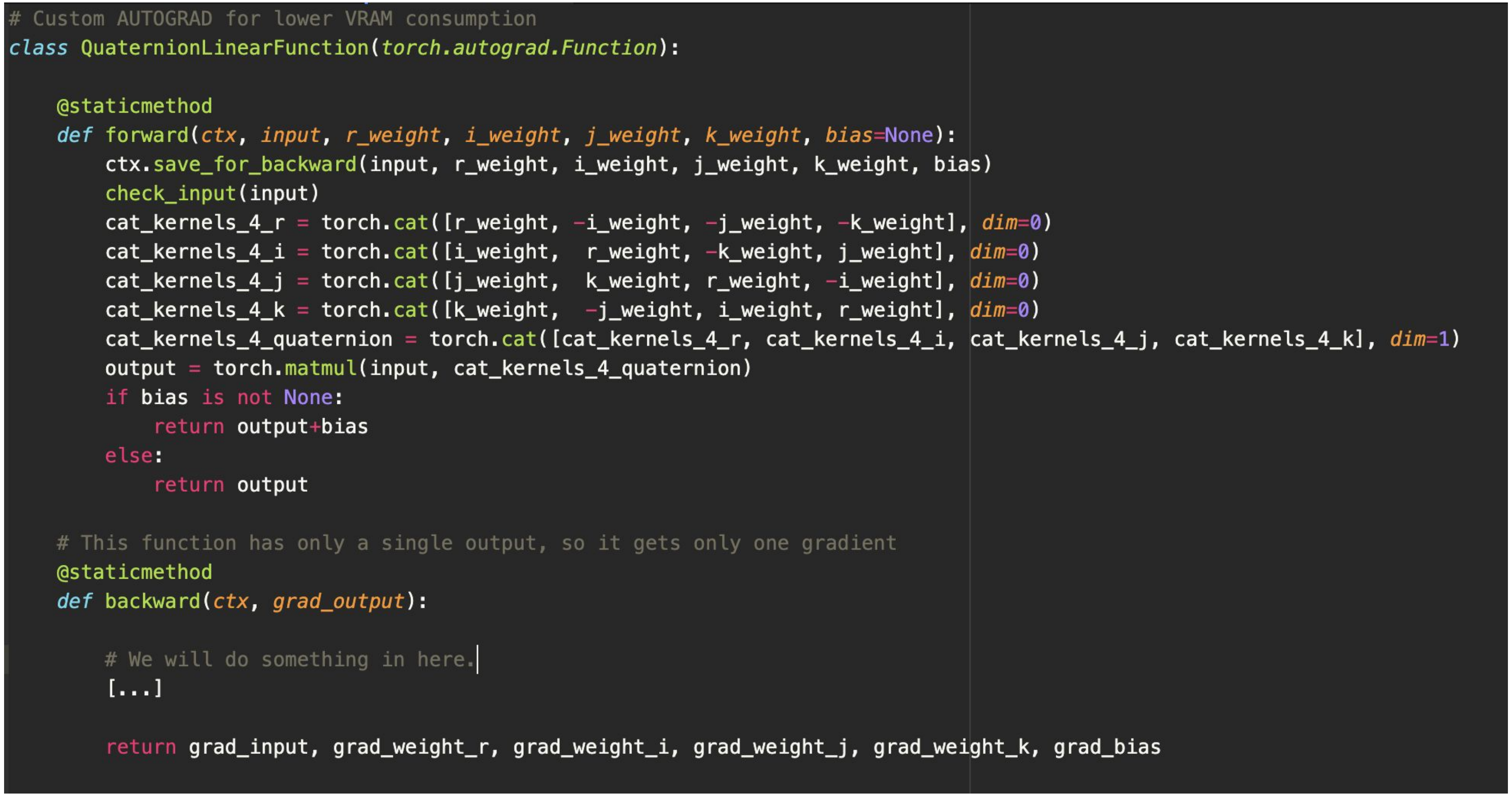




AD is one of the key for training optimisation.

Why do we care about automatic differentiation (AD)?

Example: 4x reduction in VRAM consumption during training.



```
# Custom AUTOGRAD for lower VRAM consumption
class QuaternionLinearFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input, r_weight, i_weight, j_weight, k_weight, bias=None):
        ctx.save_for_backward(input, r_weight, i_weight, j_weight, k_weight, bias)
        check_input(input)
        cat_kernels_4_r = torch.cat([r_weight, -i_weight, -j_weight, -k_weight], dim=0)
        cat_kernels_4_i = torch.cat([i_weight,  r_weight, -k_weight, j_weight], dim=0)
        cat_kernels_4_j = torch.cat([j_weight,  k_weight, r_weight, -i_weight], dim=0)
        cat_kernels_4_k = torch.cat([k_weight,  -j_weight, i_weight, r_weight], dim=0)
        cat_kernels_4_quaternion = torch.cat([cat_kernels_4_r, cat_kernels_4_i, cat_kernels_4_j, cat_kernels_4_k], dim=1)
        output = torch.matmul(input, cat_kernels_4_quaternion)
        if bias is not None:
            return output+bias
        else:
            return output

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):

        # We will do something in here.
        [...]

        return grad_input, grad_weight_r, grad_weight_i, grad_weight_j, grad_weight_k, grad_bias
```

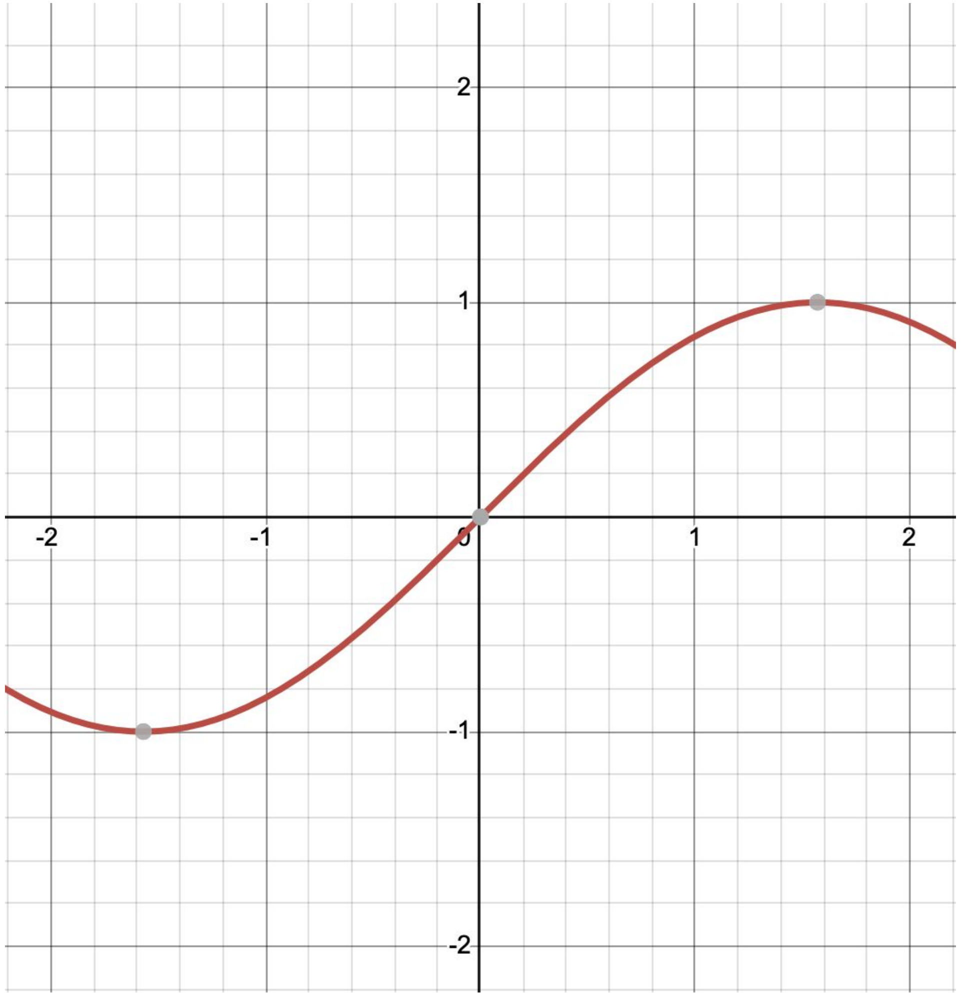
24 Gb to 6-8Gb.

Roadmap for Today

1. Why do we care about automatic differentiation?
2. **The different types of differentiations.**
3. Forward and reverse mode automatic differentiation.
4. Automatic differentiation in PyTorch.

The different types of differentiation.

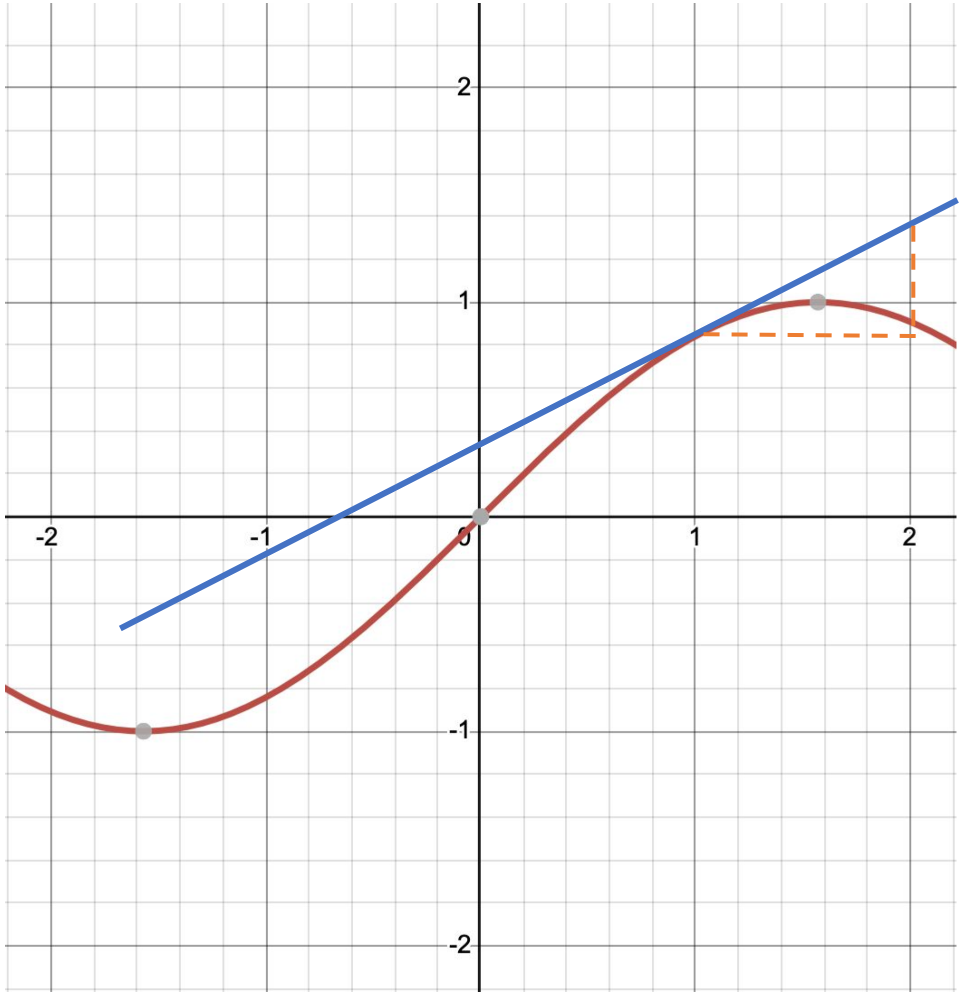
Starting from the beginning: differentiation and gradient.



The gradient can be seen as a measure of steepness or rate of change.

The different types of differentiation.

Starting from the beginning: differentiation and gradient.



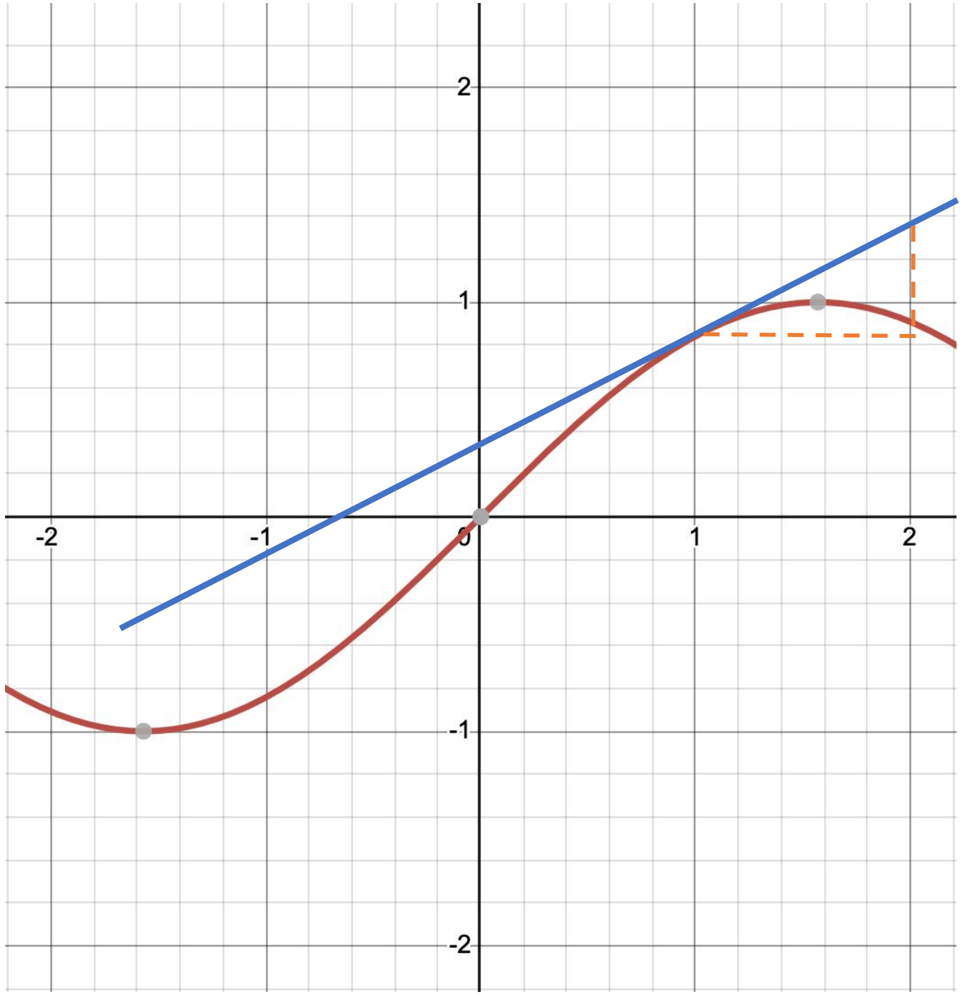
The gradient can be seen as a measure of steepness or rate of change.

The gradient of a curve at a given point is equal to the gradient of the tangent at the curve from this point.

For a move in x of 1, y increases by roughly 0.5. The gradient is 0.5.

The different types of differentiation.

Starting from the beginning: differentiation and gradient.

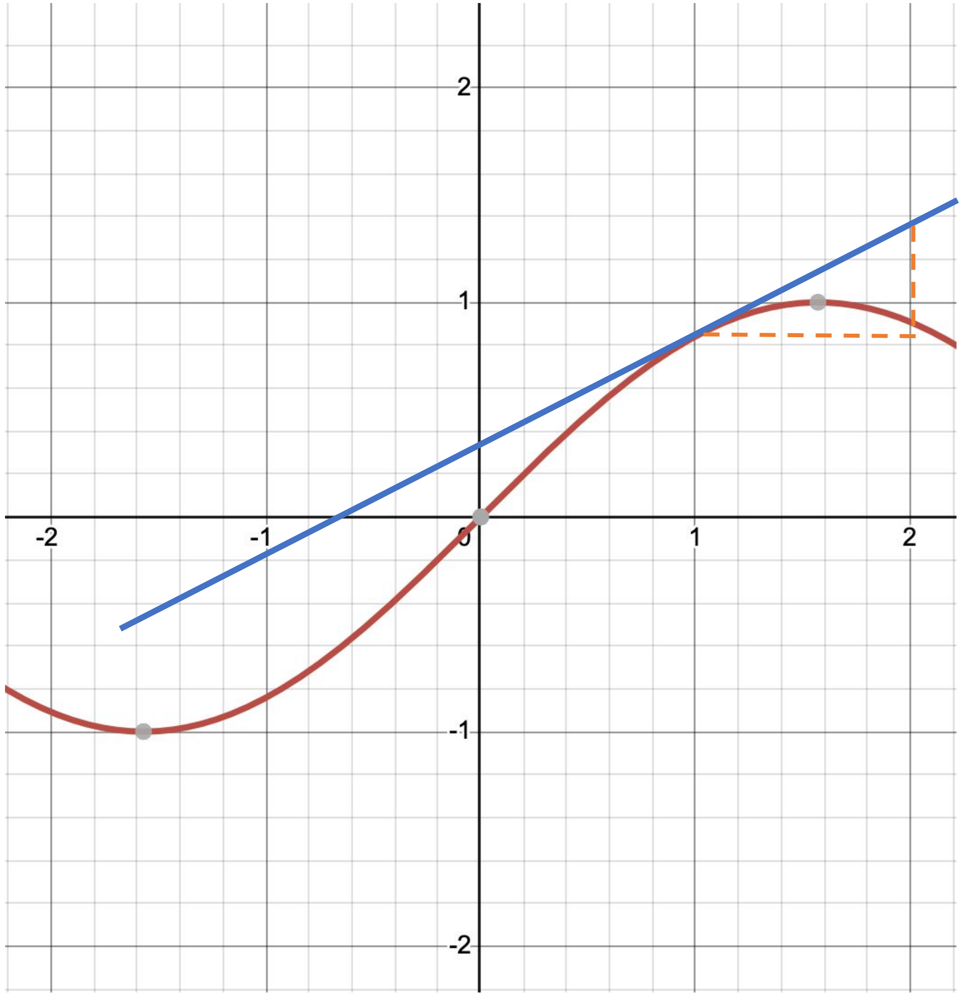


Defining the gradient function.

$$y = f(x) = 2x^3 - 6x^2 + x + 1$$

The different types of differentiation.

Starting from the beginning: differentiation and gradient.



Defining the gradient function.

$$y = f(x) = 2x^3 - 6x^2 + x + 1$$

$$\frac{dy}{dx} = f^{'}(x) = 6x^2 - 12x + 1$$

The differentiation of f w.r.t x.
or the gradient function.

The gradient at x=2 is 1.

The different types of differentiation.

Starting from the beginning: differentiation and gradient.

But wait, in DL, most functions are multivariable!

$$f(x_0, ..., x_n) = dense$$

The different types of differentiation.

Starting from the beginning: differentiation and gradient.

The gradient stores all the partial derivatives of a multivariable function.

At any given point, we know the direction of the steepest change.

Going to the opposite direction = gradient descent.

The different types of differentiation.

Starting from the beginning: differentiation and gradient.

The gradient stores all the partial derivatives of a multivariable function.

$$f(x, y) = 2x^2 + 3y^2$$

$$\nabla f(x, y) = (\frac{df}{x}, \frac{df}{y})$$

The different types of differentiation.

Starting from the beginning: differentiation and gradient.

The gradient stores all the partial derivatives of a multivariable function.

$$f(x, y) = 2x^2 + 3y^2$$

$$\nabla f(x, y) = (\frac{df}{x}, \frac{df}{y})$$

$$\frac{df}{dx} = 4x \qquad \frac{df}{dy} = 6y$$

If x=4 and y =5, then (15, 30) points in the direction of greatest increase of the function f.

The different types of differentiation.

Analytical (manual) differentiation

$$f(x, y) = 2x^2 + 3y^2$$

$$\nabla f(x, y) = (\frac{df}{x}, \frac{df}{y})$$

$$\frac{df}{dx} = 4x \qquad \frac{df}{dy} = 6y$$

This would be the first type: manual differentiation.

Ok for simple functions, but try it with a deep neural network.

The different types of differentiation.

Analytical (Manual)

Numerical

Symbolic

Automatic

The different types of differentiation.

Analytical (Manual)

Numerical

Symbolic

Automatic

The different types of differentiation.

Numerical differentiation or finite difference calculation.

We compute the partial derivatives using the Newton’s quotient:

$$\frac{df}{dx} \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Problems:

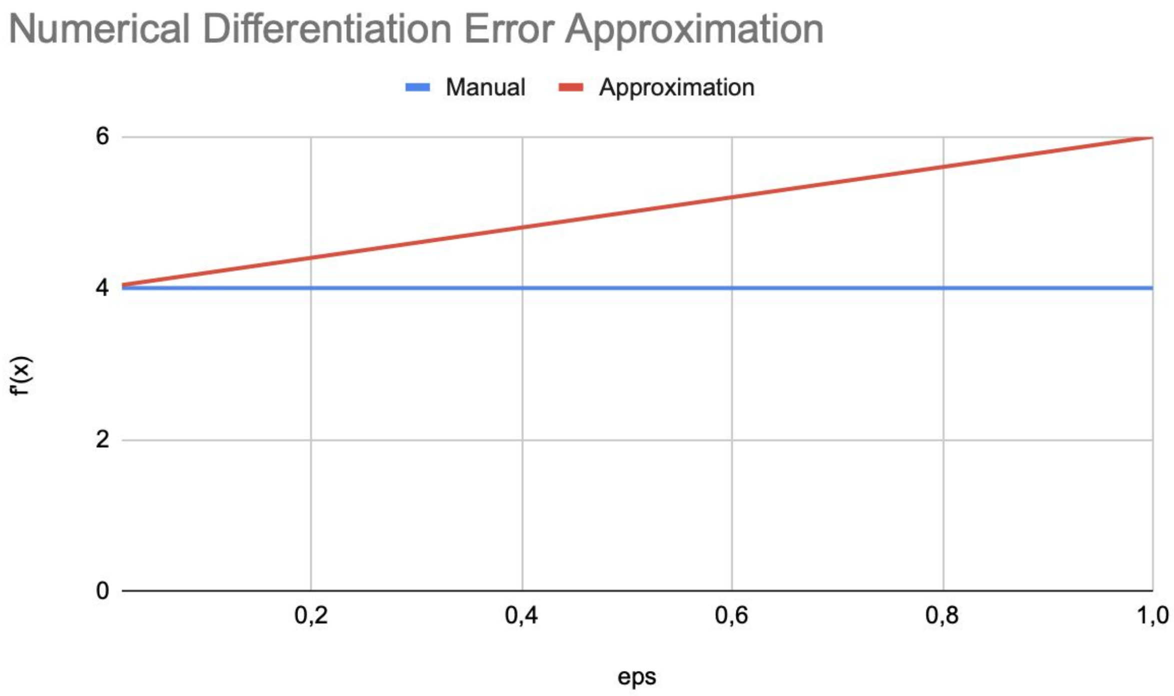
1. It’s an approximation highly dependent on the value of epsilon.
2. If epsilon is too small, we might end up in precision underflow.
3. If epsilon is too big, the error in the approximation will increase.
4. The complexity is O(n), with n parameters we need to compute f(x) n +1 times!

The different types of differentiation.

Numerical differentiation or finite difference calculation.

Example with: $f(x, y) = 2x^2 + 3y^2$

$$\frac{df}{dx} =$$



The different types of differentiation.

Analytical (Manual)

Numerical

Symbolic

Automatic

The different types of differentiation.

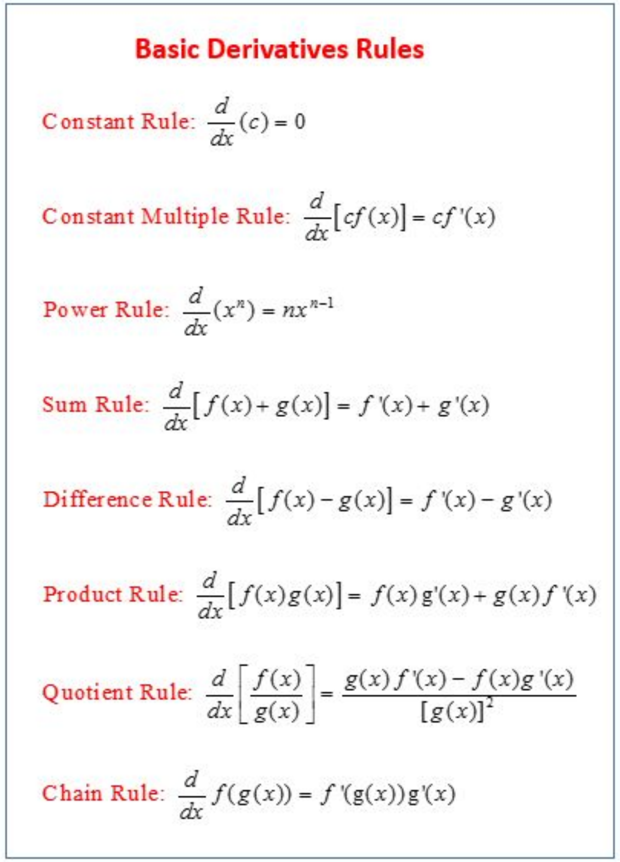
Symbolic differentiation.

Nothing more than an automated version of the manual differentiation.

The different types of differentiation.

Symbolic differentiation.

Nothing more than an automated version of the manual differentiation.



Basic Derivatives Rules

Constant Rule: $\frac{d}{dx}(c) = 0$

Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$

Power Rule: $\frac{d}{dx}(x^n) = nx^{n-1}$

Sum Rule: $\frac{d}{dx}[f(x)+g(x)] = f'(x)+g'(x)$

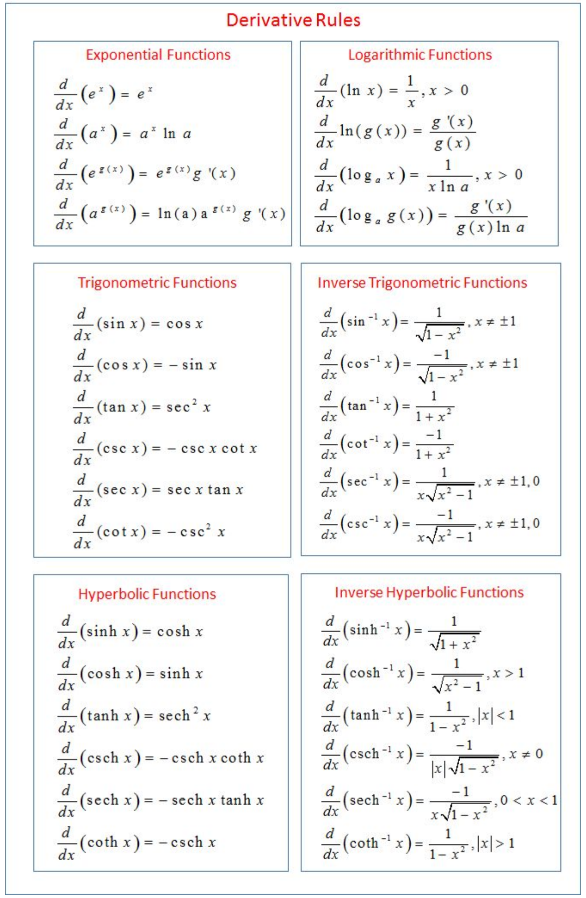
Difference Rule: $\frac{d}{dx}[f(x)-g(x)] = f'(x)-g'(x)$

Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x)+g(x)f'(x)$

Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x)-f(x)g'(x)}{[g(x)]^2}$

Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

1. Decompose your function.
2. Apply rules.



Derivative Rules

Exponential Functions

$\frac{d}{dx}(e^x) = e^x$

$\frac{d}{dx}(a^x) = a^x \ln a$

$\frac{d}{dx}(e^{g(x)}) = e^{g(x)}g'(x)$

$\frac{d}{dx}(a^{g(x)}) = \ln(\mathrm{a})\,\mathrm{a}^{g(x)}\,g'(x)$

Logarithmic Functions

$\frac{d}{dx}(\ln x) = \frac{1}{x}, x > 0$

$\frac{d}{dx}\ln(g(x)) = \frac{g'(x)}{g(x)}$

$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}, x > 0$

$\frac{d}{dx}(\log_a g(x)) = \frac{g'(x)}{g(x)\ln a}$

Trigonometric Functions

$\frac{d}{dx}(\sin x) = \cos x$

$\frac{d}{dx}(\cos x) = -\sin x$

$\frac{d}{dx}(\tan x) = \sec^2 x$

$\frac{d}{dx}(\csc x) = -\csc x \cot x$

$\frac{d}{dx}(\sec x) = \sec x \tan x$

$\frac{d}{dx}(\cot x) = -\csc^2 x$

Inverse Trigonometric Functions

$\frac{d}{dx}(\sin^{-1} x) = \frac{1}{\sqrt{1-x^2}}, x \neq \pm 1$

$\frac{d}{dx}(\cos^{-1} x) = \frac{-1}{\sqrt{1-x^2}}, x \neq \pm 1$

$\frac{d}{dx}(\tan^{-1} x) = \frac{1}{1+x^2}$

$\frac{d}{dx}(\cot^{-1} x) = \frac{-1}{1+x^2}$

$\frac{d}{dx}(\sec^{-1} x) = \frac{1}{x\sqrt{x^2-1}}, x \neq \pm 1, 0$

$\frac{d}{dx}(\csc^{-1} x) = \frac{-1}{x\sqrt{x^2-1}}, x \neq \pm 1, 0$

Hyperbolic Functions

$\frac{d}{dx}(\sinh x) = \cosh x$

$\frac{d}{dx}(\cosh x) = \sinh x$

$\frac{d}{dx}(\tanh x) = \operatorname{sech}^2 x$

$\frac{d}{dx}(\operatorname{csch} x) = -\operatorname{csch} x \coth x$

$\frac{d}{dx}(\operatorname{sech} x) = -\operatorname{sech} x \tanh x$

$\frac{d}{dx}(\coth x) = -\operatorname{csch} x$

Inverse Hyperbolic Functions

$\frac{d}{dx}(\sinh^{-1} x) = \frac{1}{\sqrt{1+x^2}}$

$\frac{d}{dx}(\cosh^{-1} x) = \frac{1}{\sqrt{x^2-1}}, x > 1$

$\frac{d}{dx}(\tanh^{-1} x) = \frac{1}{1-x^2}, |x| < 1$

$\frac{d}{dx}(\operatorname{csch}^{-1} x) = \frac{-1}{|x|\sqrt{1-x^2}}, x \neq 0$

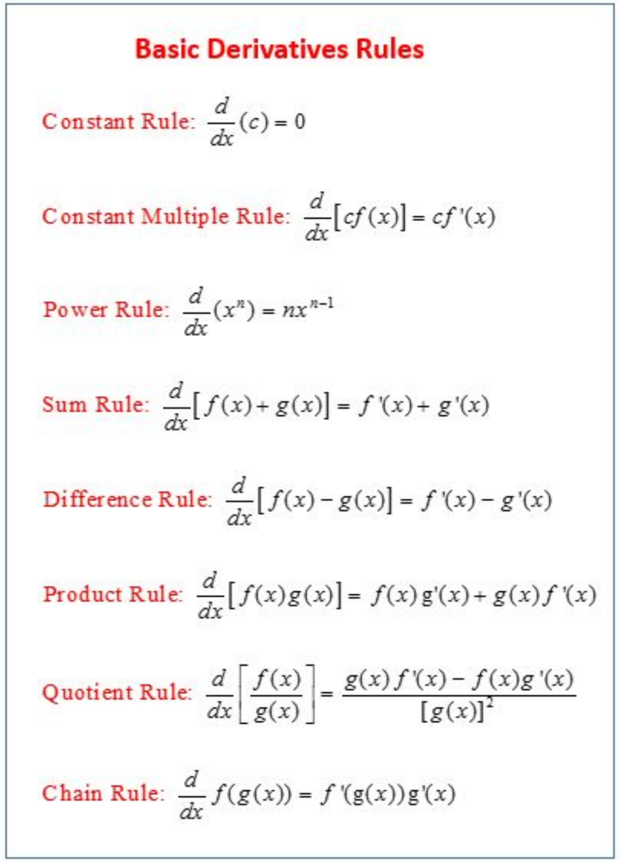
$\frac{d}{dx}(\operatorname{sech}^{-1} x) = \frac{-1}{x\sqrt{1-x^2}}, 0 < x < 1$

$\frac{d}{dx}(\coth^{-1} x) = \frac{1}{1-x^2}, |x| > 1$

The different types of differentiation.

Symbolic differentiation.

Nothing more than an automated version of the manual differentiation.



Basic Derivatives Rules

Constant Rule: $\frac{d}{dx}(c) = 0$

Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$

Power Rule: $\frac{d}{dx}(x^n) = nx^{n-1}$

Sum Rule: $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$

Difference Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$

Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$

Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$

Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

1. Decompose your function.
2. Apply rules.

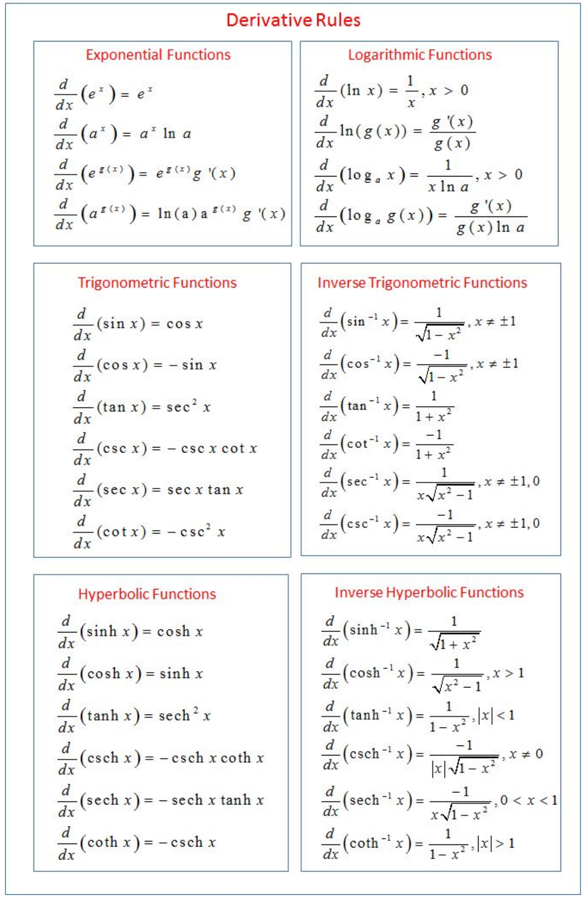
Pros:

Exact value up to numerical precision!

Cons:

Expression swell…

The derivative becomes much more complex that the initial function.



Derivative Rules

Exponential Functions

$\frac{d}{dx}(e^x) = e^x$

$\frac{d}{dx}(a^x) = a^x \ln a$

$\frac{d}{dx}(e^{g(x)}) = e^{g(x)} g'(x)$

$\frac{d}{dx}(a^{g(x)}) = \ln(a) a^{g(x)} g'(x)$

Logarithmic Functions

$\frac{d}{dx}(\ln x) = \frac{1}{x}, x > 0$

$\frac{d}{dx}\ln(g(x)) = \frac{g'(x)}{g(x)}$

$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}, x > 0$

$\frac{d}{dx}(\log_a g(x)) = \frac{g'(x)}{g(x) \ln a}$

Trigonometric Functions

$\frac{d}{dx}(\sin x) = \cos x$

$\frac{d}{dx}(\cos x) = -\sin x$

$\frac{d}{dx}(\tan x) = \sec^2 x$

$\frac{d}{dx}(\csc x) = -\csc x \cot x$

$\frac{d}{dx}(\sec x) = \sec x \tan x$

$\frac{d}{dx}(\cot x) = -\csc^2 x$

Inverse Trigonometric Functions

$\frac{d}{dx}(\sin^{-1} x) = \frac{1}{\sqrt{1-x^2}}, x \neq \pm 1$

$\frac{d}{dx}(\cos^{-1} x) = \frac{-1}{\sqrt{1-x^2}}, x \neq \pm 1$

$\frac{d}{dx}(\tan^{-1} x) = \frac{1}{1+x^2}$

$\frac{d}{dx}(\cot^{-1} x) = \frac{-1}{1+x^2}$

$\frac{d}{dx}(\sec^{-1} x) = \frac{1}{x\sqrt{x^2-1}}, x \neq \pm 1, 0$

$\frac{d}{dx}(\csc^{-1} x) = \frac{-1}{x\sqrt{x^2-1}}, x \neq \pm 1, 0$

Hyperbolic Functions

$\frac{d}{dx}(\sinh x) = \cosh x$

$\frac{d}{dx}(\cosh x) = \sinh x$

$\frac{d}{dx}(\tanh x) = \text{sech}^2 x$

$\frac{d}{dx}(\text{csch}\, x) = -\text{csch}\, x \coth x$

$\frac{d}{dx}(\text{sech}\, x) = -\text{sech}\, x \tanh x$

$\frac{d}{dx}(\coth x) = -\text{csch}\, x$

Inverse Hyperbolic Functions

$\frac{d}{dx}(\sinh^{-1} x) = \frac{1}{\sqrt{1+x^2}}$

$\frac{d}{dx}(\cosh^{-1} x) = \frac{1}{\sqrt{x^2-1}}, x > 1$

$\frac{d}{dx}(\tanh^{-1} x) = \frac{1}{1-x^2}, |x| < 1$

$\frac{d}{dx}(\text{csch}^{-1} x) = \frac{-1}{|x|\sqrt{1-x^2}}, x \neq 0$

$\frac{d}{dx}(\text{sech}^{-1} x) = \frac{-1}{x\sqrt{1-x^2}}, 0 < x < 1$

$\frac{d}{dx}(\coth^{-1} x) = \frac{1}{1-x^2}, |x| > 1$

The different types of differentiation.

Symbolic differentiation.

Expression swell.

$$h(x) = f(x)g(x)$$

$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

The different types of differentiation.

Symbolic differentiation.

Expression swell.

$$h(x) = f(x)g(x)$$

$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

$$f(x) = u(x)v(x)$$

$$h'(x) = (u'(x)v(x) + v'(x)u(x))g(x) + (x)v(x)g'(x)$$

Can become intractable.
Also, it requires closed form expressions! (no loop, if statements etc…)

The different types of differentiation.

Analytical (Manual)

Numerical

Symbolic

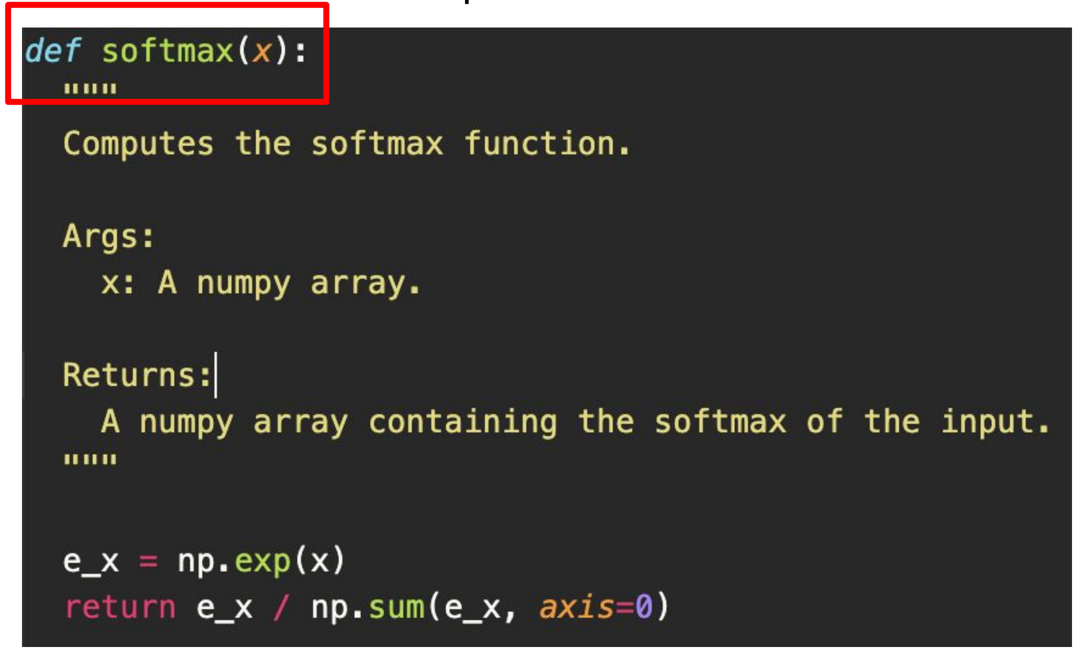
Automatic

The different types of differentiation.

Automatic differentiation.

The key behind AD:

Implemented differentiable functions are composed of primitive operations whose derivatives are known and the chain rule enables us to compose them.



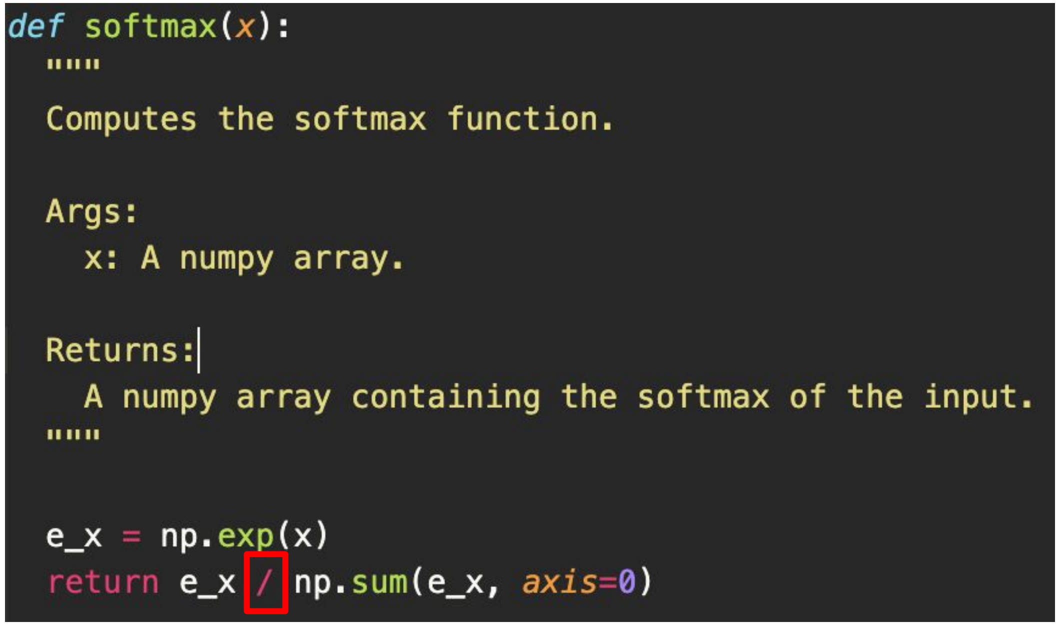
```
def softmax(x):
  """
  Computes the softmax function.

  Args:
    x: A numpy array.

  Returns:
    A numpy array containing the softmax of the input.
  """

  e_x = np.exp(x)
  return e_x / np.sum(e_x, axis=0)
```

The different types of differentiation.

Automatic differentiation.

The key behind AD:

Implemented differentiable functions are composed of primitive operations whose derivatives are known and the chain rule enables us to compose them.



```python
def softmax(x):
  """
  Computes the softmax function.

  Args:
    x: A numpy array.

  Returns:
    A numpy array containing the softmax of the input.
  """

  e_x = np.exp(x)
  return e_x / np.sum(e_x, axis=0)
```

The different types of differentiation.

Automatic differentiation.

The key behind AD:

Implemented differentiable functions are composed of primitive operations whose derivatives are known and the chain rule enables us to compose them.

$$y = cos(x^2) \qquad u = x^2 \qquad y = cos(u)$$

The different types of differentiation.

Automatic differentiation.

The key behind AD:

Implemented differentiable functions are composed of primitive operations whose derivatives are known and the chain rule enables us to compose them.

$$y = cos(x^2) \qquad u = x^2 \qquad y = cos(u)$$

$$\frac{du}{dx} = 2x \qquad \frac{dy}{dx} = -sin(u)$$

The different types of differentiation.

Automatic differentiation.

The key behind AD:

Implemented differentiable functions are composed of primitive operations whose derivatives are known and the chain rule enables us to compose them.

$$y = cos(x^2) \qquad u = x^2 \qquad y = cos(u)$$

$$\frac{du}{dx} = 2x \qquad \frac{dy}{dx} = -sin(u)$$

$$\frac{dy}{dx} = \boxed{\frac{dy}{du} \times \frac{du}{dx}} = -sin(u) \times 2x = -2xsin(x^2)$$

Roadmap for Today

1. Why do we care about automatic differentiation?
2. The different types of differentiations.
3. **Forward and reverse mode automatic differentiation.**
4. Automatic differentiation in PyTorch.

Forward and reverse mode automatic differentiation.

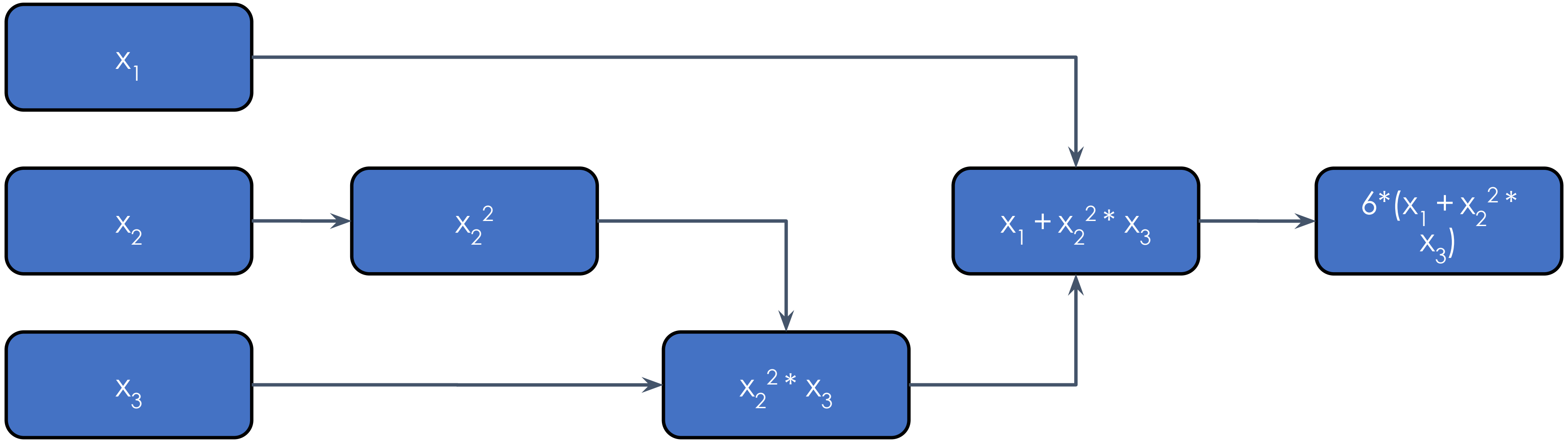
Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$

Forward and reverse mode automatic differentiation.

Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$

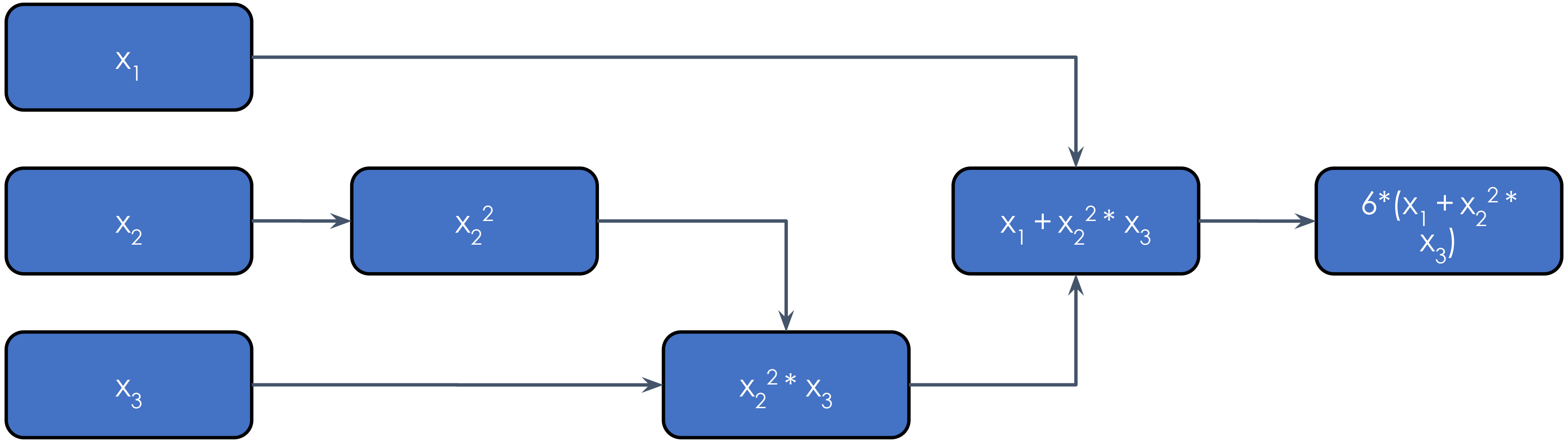


Compute the output and the corresponding derivative at each node.

Forward and reverse mode automatic differentiation.

Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$

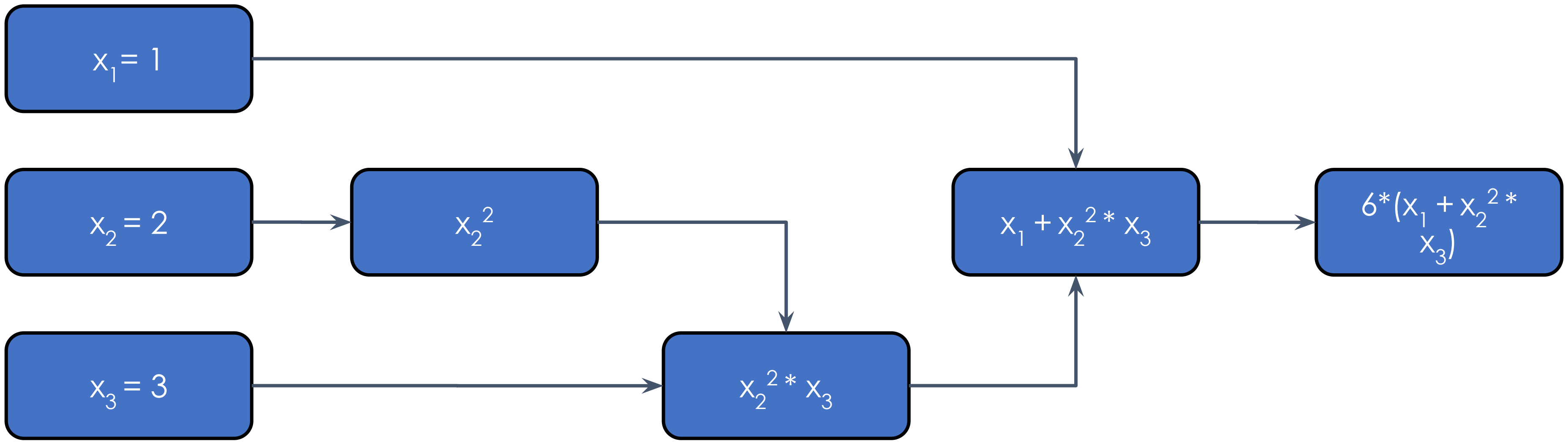


$$\frac{df}{dx_1}$$ at (1,2,3)?

Forward and reverse mode automatic differentiation.

Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



$$\frac{dx_1}{dx_1} = 1$$

$x_1 = 1$

$$\frac{dx_2}{dx_1} = 0$$

$x_2 = 2$

$$\frac{dx_3}{dx_1} = 0$$

$x_3 = 3$

x_2^2

$x_2^2 * x_3$

$x_1 + x_2^2 * x_3$

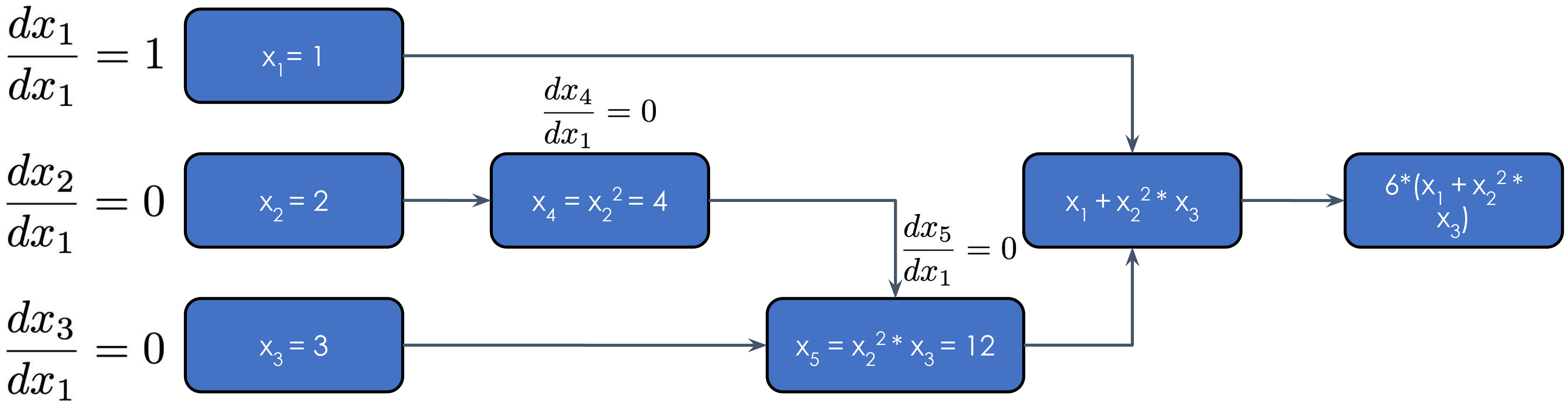
$6*(x_1 + x_2^2 * x_3)$

$$\frac{df}{dx_1}$$ at (1,2,3)

Forward and reverse mode automatic differentiation.

Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



$\frac{dx_1}{dx_1} = 1$

$\frac{dx_2}{dx_1} = 0$

$\frac{dx_3}{dx_1} = 0$

$x_1 = 1$

$x_2 = 2$

$x_3 = 3$

$\frac{dx_4}{dx_1} = 0$

$x_4 = x_2^2 = 4$

$\frac{dx_5}{dx_1} = 0$

$x_5 = x_2^2 * x_3 = 12$

$x_1 + x_2^2 * x_3$

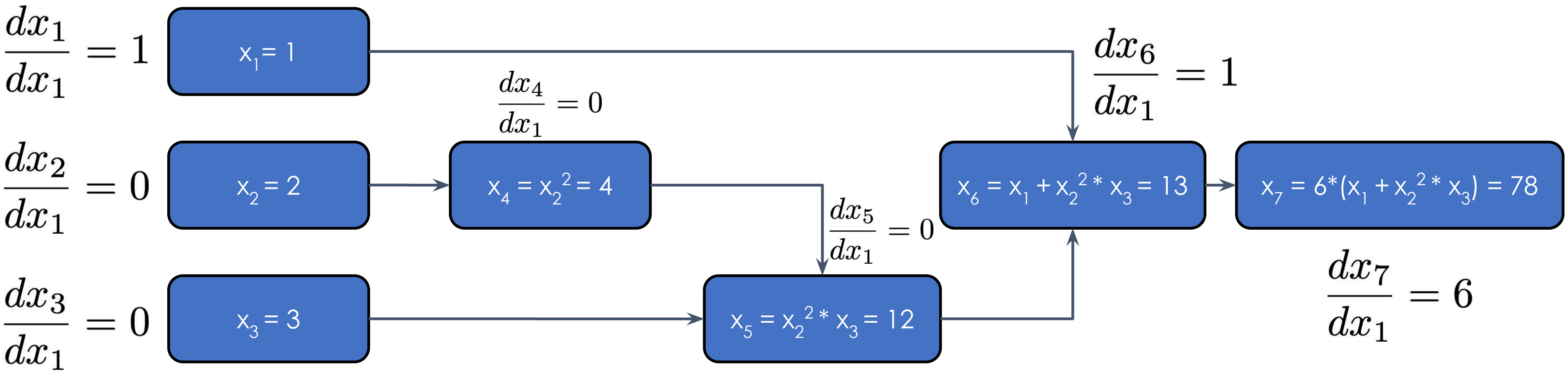
$6*(x_1 + x_2^2 * x_3)$

$\frac{df}{dx_1}$ at (1,2,3)

Forward and reverse mode automatic differentiation.

Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



$\frac{dx_1}{dx_1} = 1$

$x_1 = 1$

$\frac{dx_2}{dx_1} = 0$

$x_2 = 2$

$\frac{dx_3}{dx_1} = 0$

$x_3 = 3$

$\frac{dx_4}{dx_1} = 0$

$x_4 = x_2^2 = 4$

$\frac{dx_5}{dx_1} = 0$

$x_5 = x_2^2 * x_3 = 12$

$\frac{dx_6}{dx_1} = 1$

$x_6 = x_1 + x_2^2 * x_3 = 13$

$x_7 = 6*(x_1 + x_2^2 * x_3) = 78$

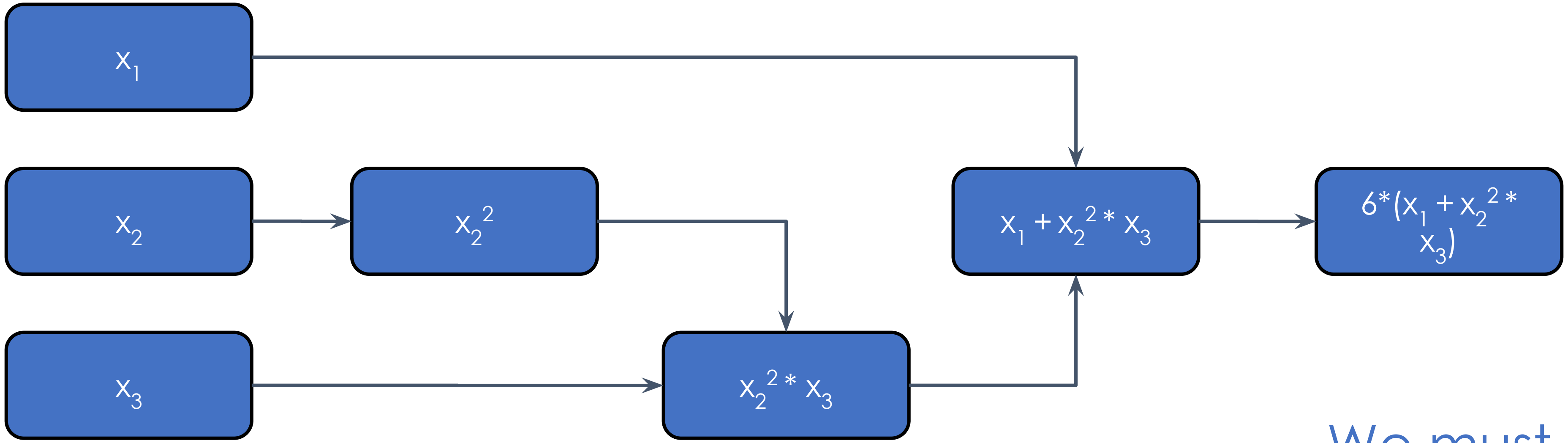
$\frac{dx_7}{dx_1} = 6$

$\frac{df}{dx_1}$ at (1,2,3)

Forward and reverse mode automatic differentiation.

Forward mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



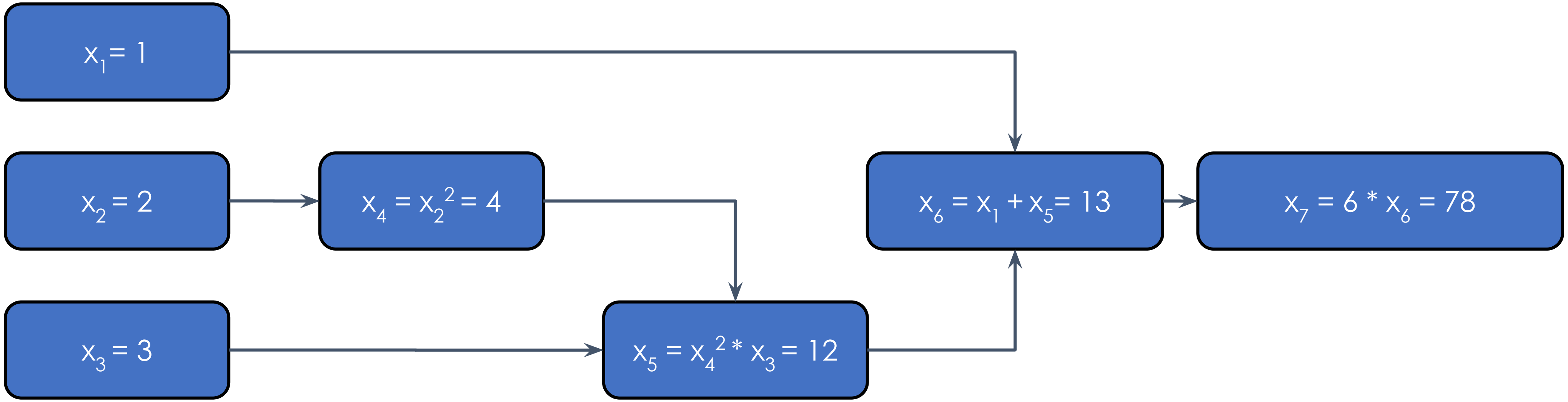
$\frac{df}{dx_2}$ at (1,2,3)?

We must recompute the whole graph!

Forward and reverse mode automatic differentiation.

Reverse mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$

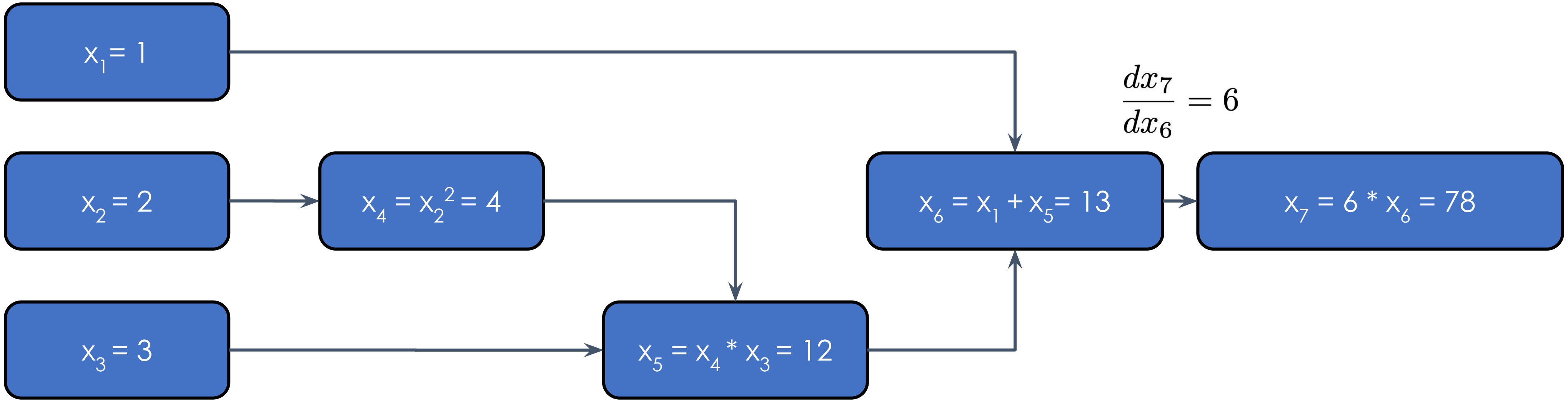


Compute the outputs during a forward pass. We also rewrite the graph, this will be useful in the next step.

Forward and reverse mode automatic differentiation.

Reverse mode.

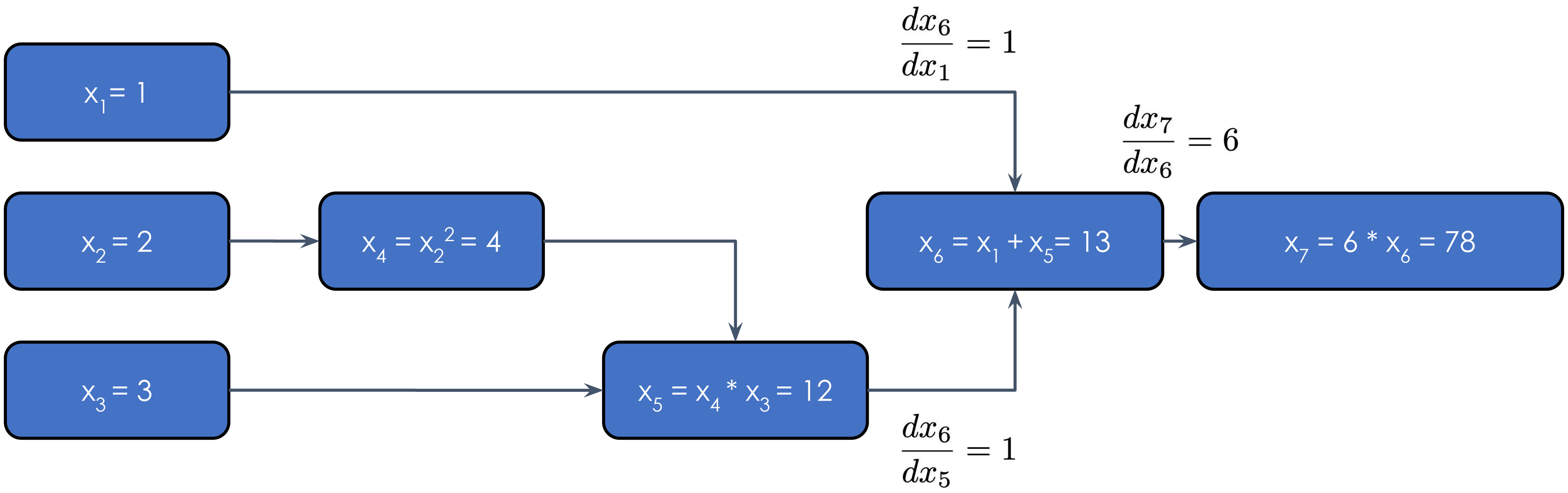
$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



Forward and reverse mode automatic differentiation.

Reverse mode.

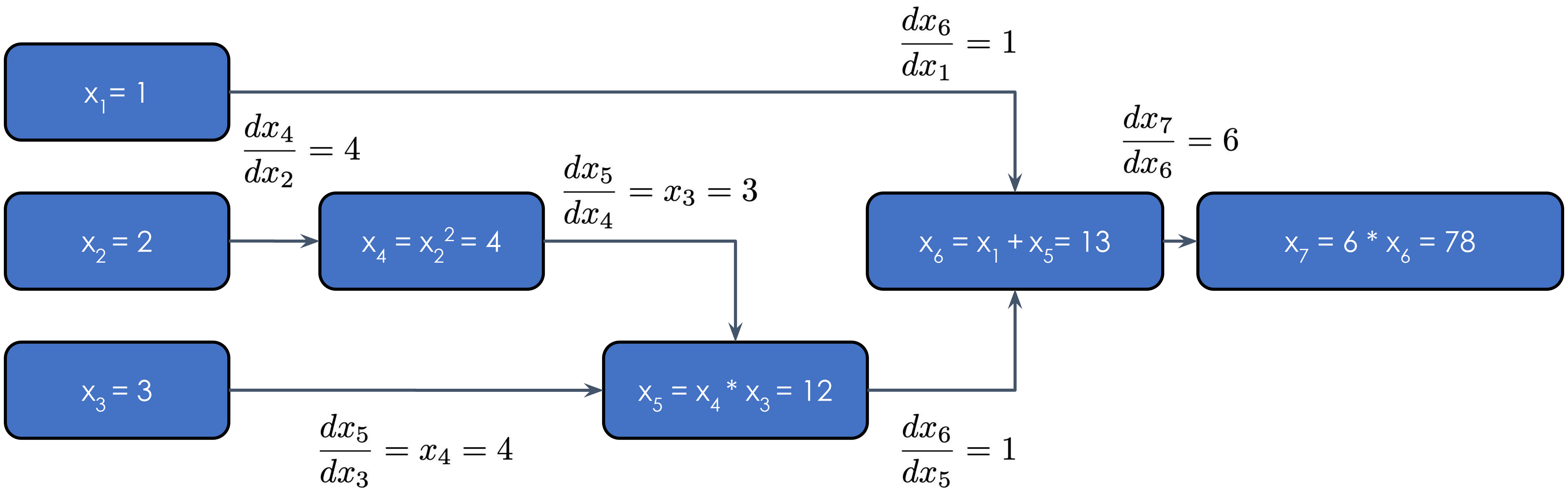
$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



Forward and reverse mode automatic differentiation.

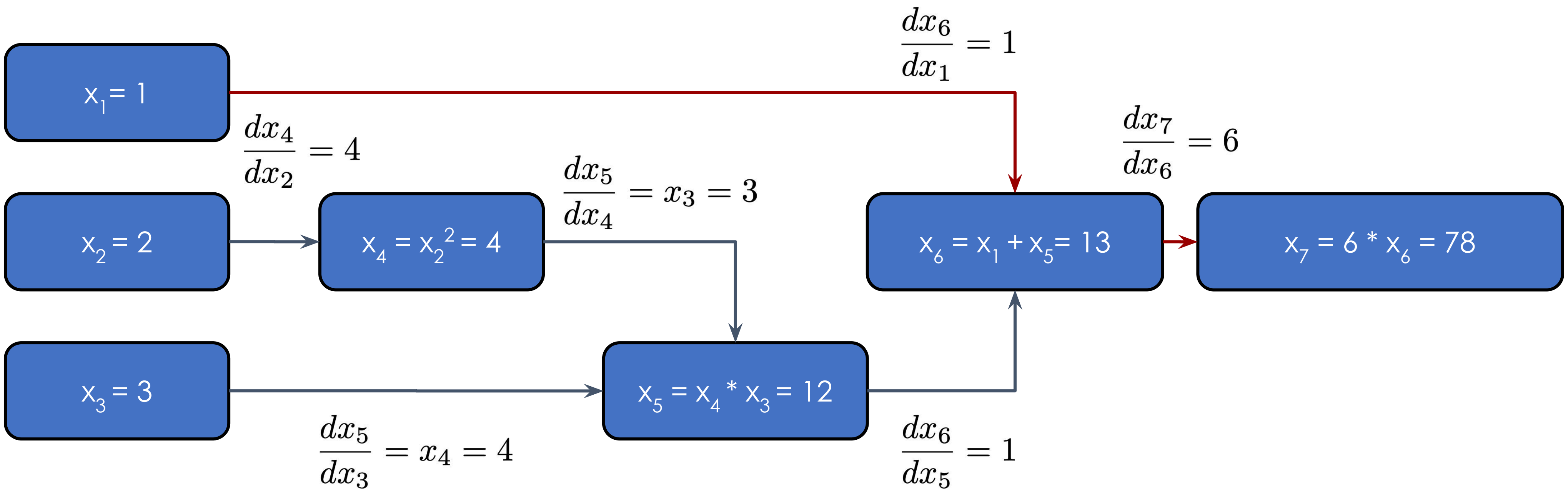
Reverse mode.

$$f(x_1, x_2, x_3) = 6(x_1 + x_2^2 \times x_3)$$



Forward and reverse mode automatic differentiation.

Reverse mode.

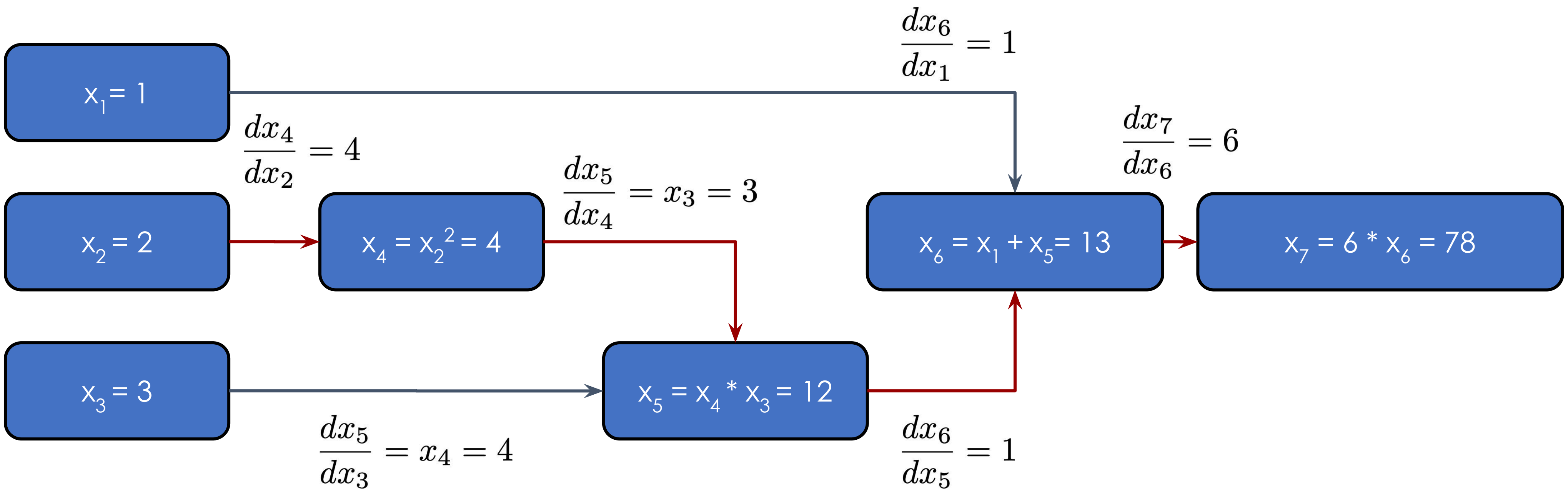


We then apply the Chain Rule to get the derivatives.

$$\frac{df}{dx_1} = \frac{dx_7}{dx_6}\frac{dx_6}{dx_1} = 6$$

Forward and reverse mode automatic differentiation.

Reverse mode.



$x_1 = 1$

$x_2 = 2$

$x_3 = 3$

$x_4 = x_2^2 = 4$

$x_5 = x_4 * x_3 = 12$

$x_6 = x_1 + x_5 = 13$

$x_7 = 6 * x_6 = 78$

$$\frac{dx_4}{dx_2} = 4$$

$$\frac{dx_5}{dx_4} = x_3 = 3$$

$$\frac{dx_5}{dx_3} = x_4 = 4$$

$$\frac{dx_6}{dx_1} = 1$$

$$\frac{dx_6}{dx_5} = 1$$

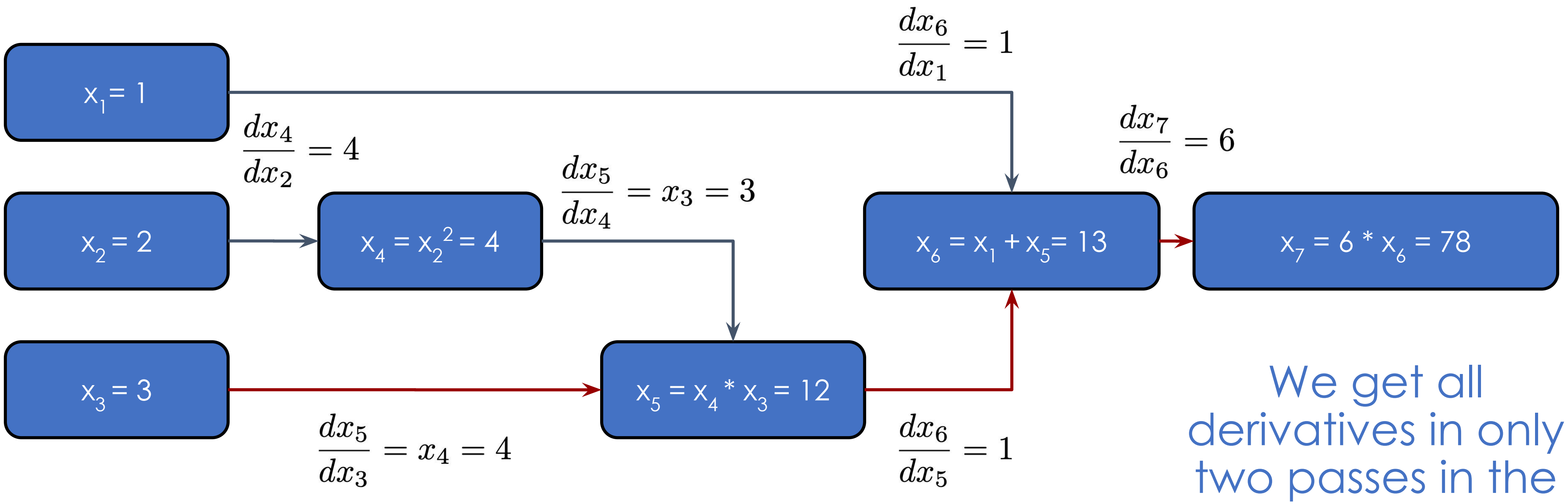
$$\frac{dx_7}{dx_6} = 6$$

$$\frac{df}{dx_1} = \frac{dx_7}{dx_6}\frac{dx_6}{dx_1} = 6$$

$$\frac{df}{dx_2} = \frac{dx_7}{dx_6}\frac{dx_6}{dx_5}\frac{dx_5}{dx_4}\frac{dx_4}{dx_1} = 72$$

Forward and reverse mode automatic differentiation.

Reverse mode.



$x_1 = 1$

$x_2 = 2$

$x_3 = 3$

$x_4 = x_2^2 = 4$

$x_5 = x_4 * x_3 = 12$

$x_6 = x_1 + x_5 = 13$

$x_7 = 6 * x_6 = 78$

$\frac{dx_6}{dx_1} = 1$

$\frac{dx_4}{dx_2} = 4$

$\frac{dx_5}{dx_4} = x_3 = 3$

$\frac{dx_7}{dx_6} = 6$

$\frac{dx_5}{dx_3} = x_4 = 4$

$\frac{dx_6}{dx_5} = 1$

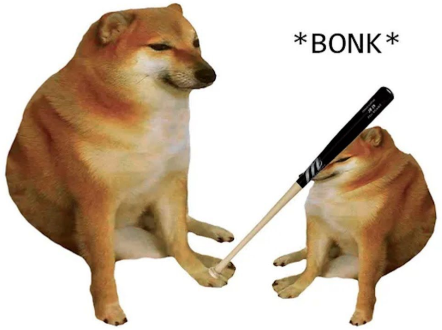
We get all derivatives in only two passes in the graph!

$$\frac{df}{dx_1} = \frac{dx_7}{dx_6}\frac{dx_6}{dx_1} = 6 \qquad \frac{df}{dx_2} = \frac{dx_7}{dx_6}\frac{dx_6}{dx_5}\frac{dx_5}{dx_4}\frac{dx_4}{dx_1} = 72 \qquad \frac{df}{dx_3} = \frac{dx_7}{dx_6}\frac{dx_6}{dx_5}\frac{dx_5}{dx_3} = 24$$

Roadmap for Today

1. Why do we care about automatic differentiation?
2. The different types of differentiations.
3. Forward and reverse mode automatic differentiation.
4. **Automatic differentiation in PyTorch** .

Automatic differentiation in PyTorch

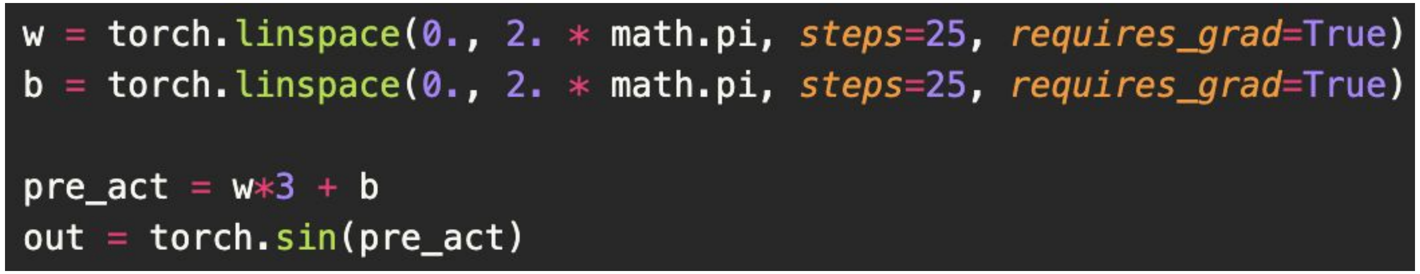


Automatic differentiation in PyTorch

Take what we did manually in the Reverse Mode slides, then convert it to Python — PyTorch Autograd.

Automatic differentiation in PyTorch

Step by step example.

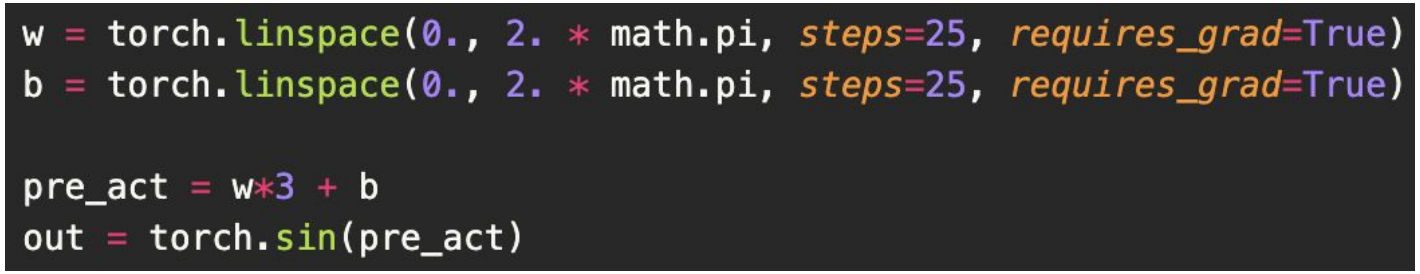


```
w = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)
b = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)

pre_act = w*3 + b
out = torch.sin(pre_act)
```

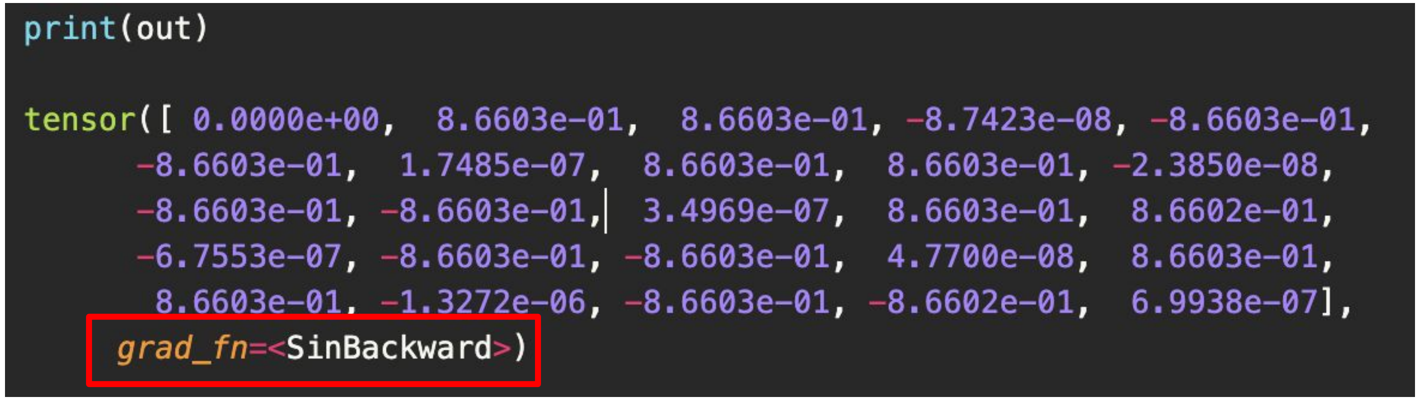
Let’s define a simple dense non-linear transformation and extract the backward compute graph from it.

Automatic differentiation in PyTorch

Step by step example.



```
w = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)
b = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)

pre_act = w*3 + b
out = torch.sin(pre_act)
```

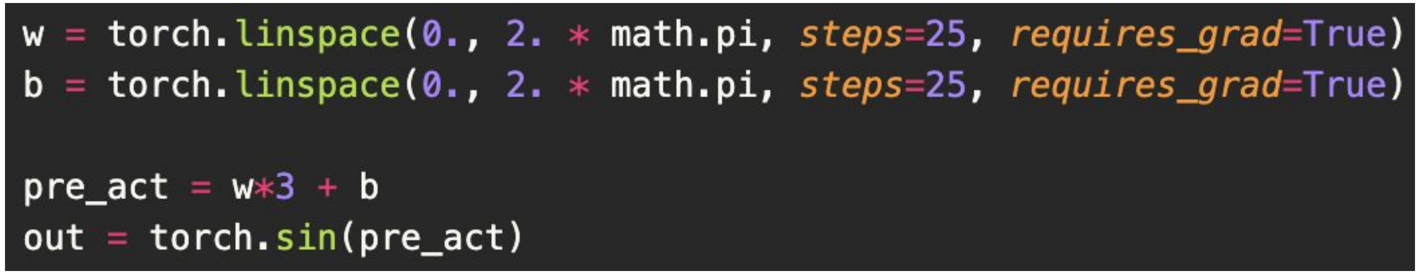


```
print(out)

tensor([ 0.0000e+00,  8.6603e-01,  8.6603e-01, -8.7423e-08, -8.6603e-01,
        -8.6603e-01,  1.7485e-07,  8.6603e-01,  8.6603e-01, -2.3850e-08,
        -8.6603e-01, -8.6603e-01,  3.4969e-07,  8.6603e-01,  8.6602e-01,
        -6.7553e-07, -8.6603e-01, -8.6603e-01,  4.7700e-08,  8.6603e-01,
         8.6603e-01, -1.3272e-06, -8.6603e-01, -8.6602e-01,  6.9938e-07],
       grad_fn=<SinBackward>)
```

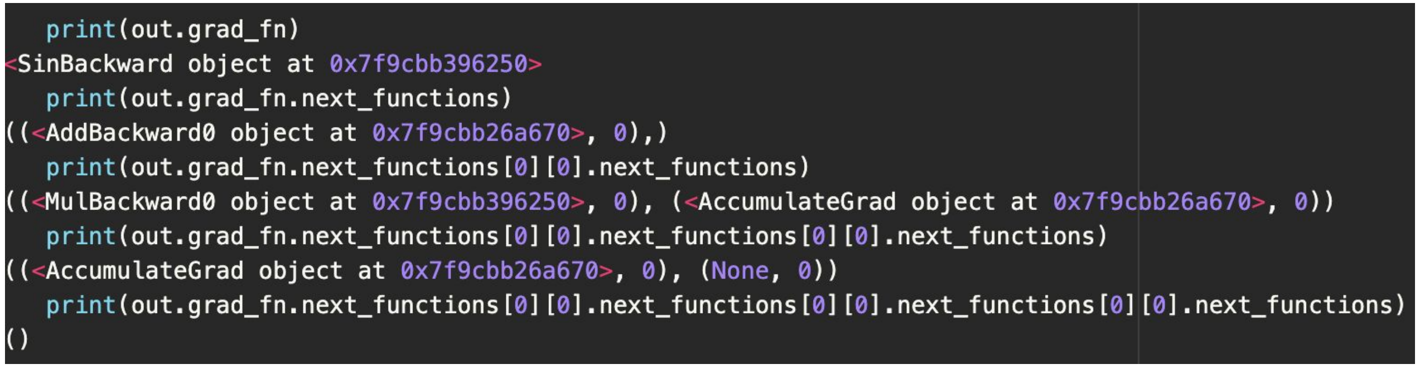
Looking at the out variable, we can spot the last backward function that needs to be executed. This will give us a cos.

Automatic differentiation in PyTorch

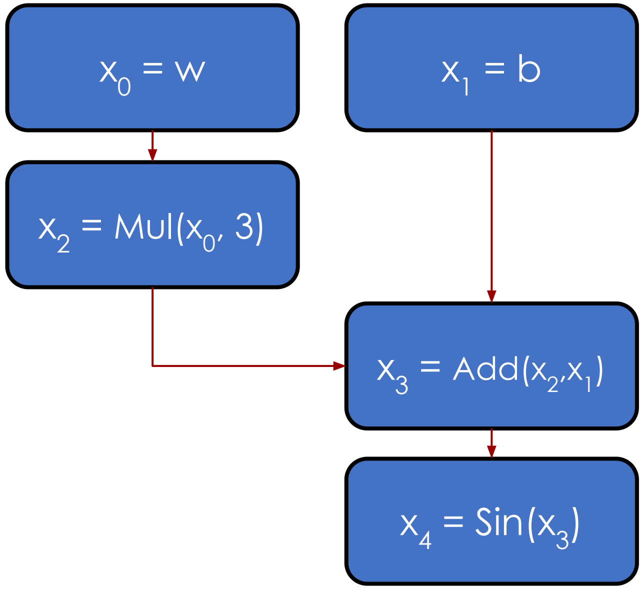
Step by step example.



```
w = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)
b = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)

pre_act = w*3 + b
out = torch.sin(pre_act)
```



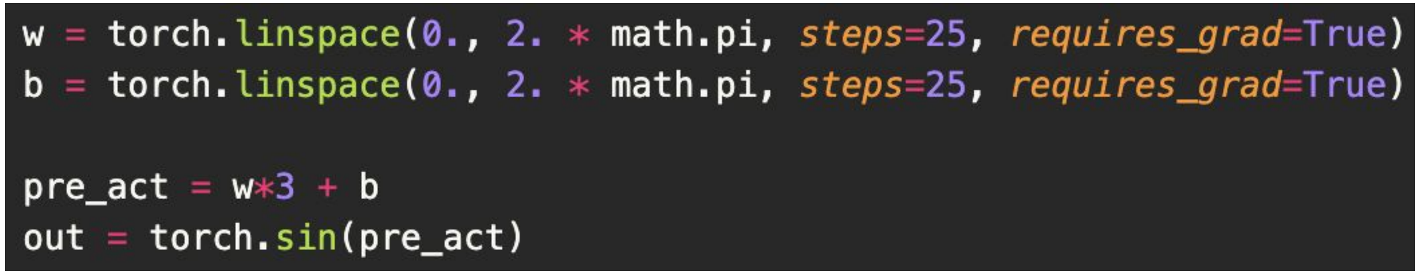
```
   print(out.grad_fn)
<SinBackward object at 0x7f9cbb396250>
   print(out.grad_fn.next_functions)
((<AddBackward0 object at 0x7f9cbb26a670>, 0),)
   print(out.grad_fn.next_functions[0][0].next_functions)
((<MulBackward0 object at 0x7f9cbb396250>, 0), (<AccumulateGrad object at 0x7f9cbb26a670>, 0))
   print(out.grad_fn.next_functions[0][0].next_functions[0][0].next_functions)
((<AccumulateGrad object at 0x7f9cbb26a670>, 0), (None, 0))
   print(out.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions)
()
```



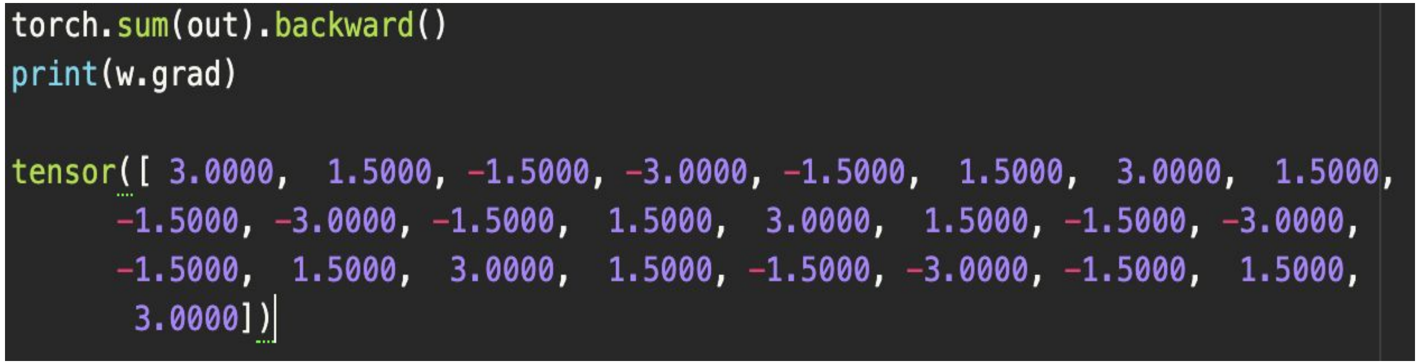
You can easily reconstruct the graph from the grad_func.

Automatic differentiation in PyTorch

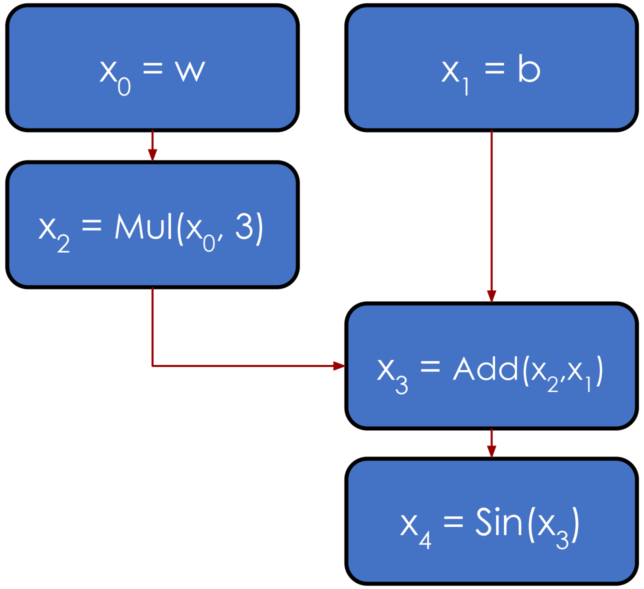
Step by step example.



```
w = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)
b = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)

pre_act = w*3 + b
out = torch.sin(pre_act)
```



```
torch.sum(out).backward()
print(w.grad)

tensor([ 3.0000,  1.5000, -1.5000, -3.0000, -1.5000,  1.5000,  3.0000,  1.5000,
        -1.5000, -3.0000, -1.5000,  1.5000,  3.0000,  1.5000, -1.5000, -3.0000,
        -1.5000,  1.5000,  3.0000,  1.5000, -1.5000, -3.0000, -1.5000,  1.5000,
         3.0000])
```

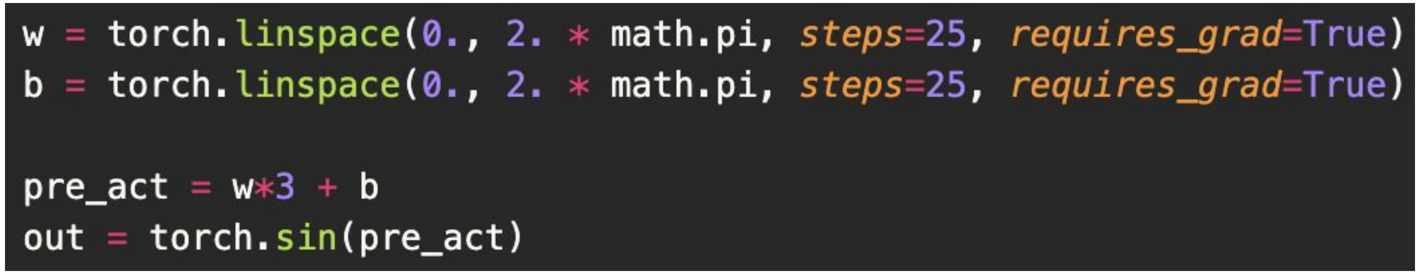


And we get the gradient by calling .backward().

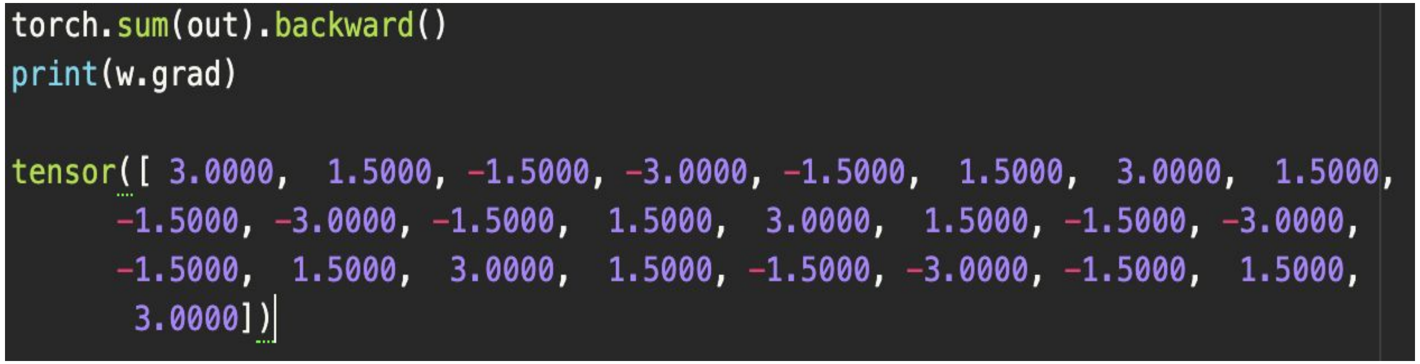
$$w.grad() = \frac{d_{out}}{d_w} = \frac{dx_4}{dx_3}\frac{dx_3}{dx_2}\frac{dx_2}{dx_0}$$

Automatic differentiation in PyTorch

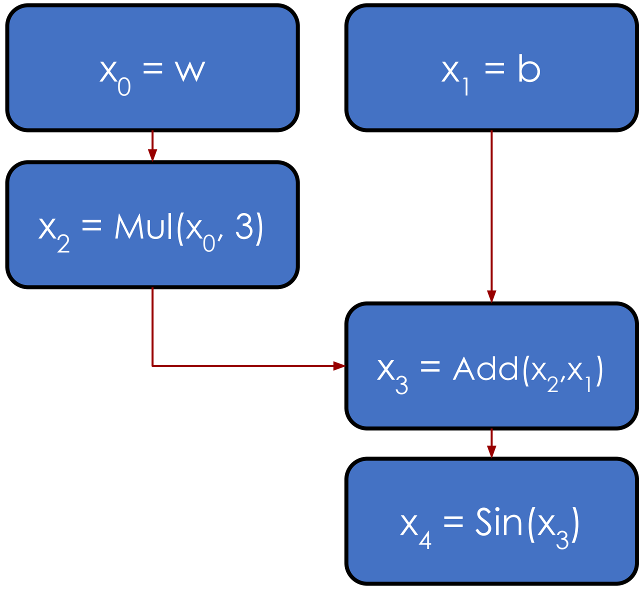
Step by step example.



```
w = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)
b = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)

pre_act = w*3 + b
out = torch.sin(pre_act)
```



```
torch.sum(out).backward()
print(w.grad)

tensor([ 3.0000,  1.5000, -1.5000, -3.0000, -1.5000,  1.5000,  3.0000,  1.5000,
        -1.5000, -3.0000, -1.5000,  1.5000,  3.0000,  1.5000, -1.5000, -3.0000,
        -1.5000,  1.5000,  3.0000,  1.5000, -1.5000, -3.0000, -1.5000,  1.5000,
         3.0000])
```



$x_0 = w$

$x_1 = b$

$x_2 = \mathrm{Mul}(x_0, 3)$

$x_3 = \mathrm{Add}(x_2, x_1)$

$x_4 = \mathrm{Sin}(x_3)$

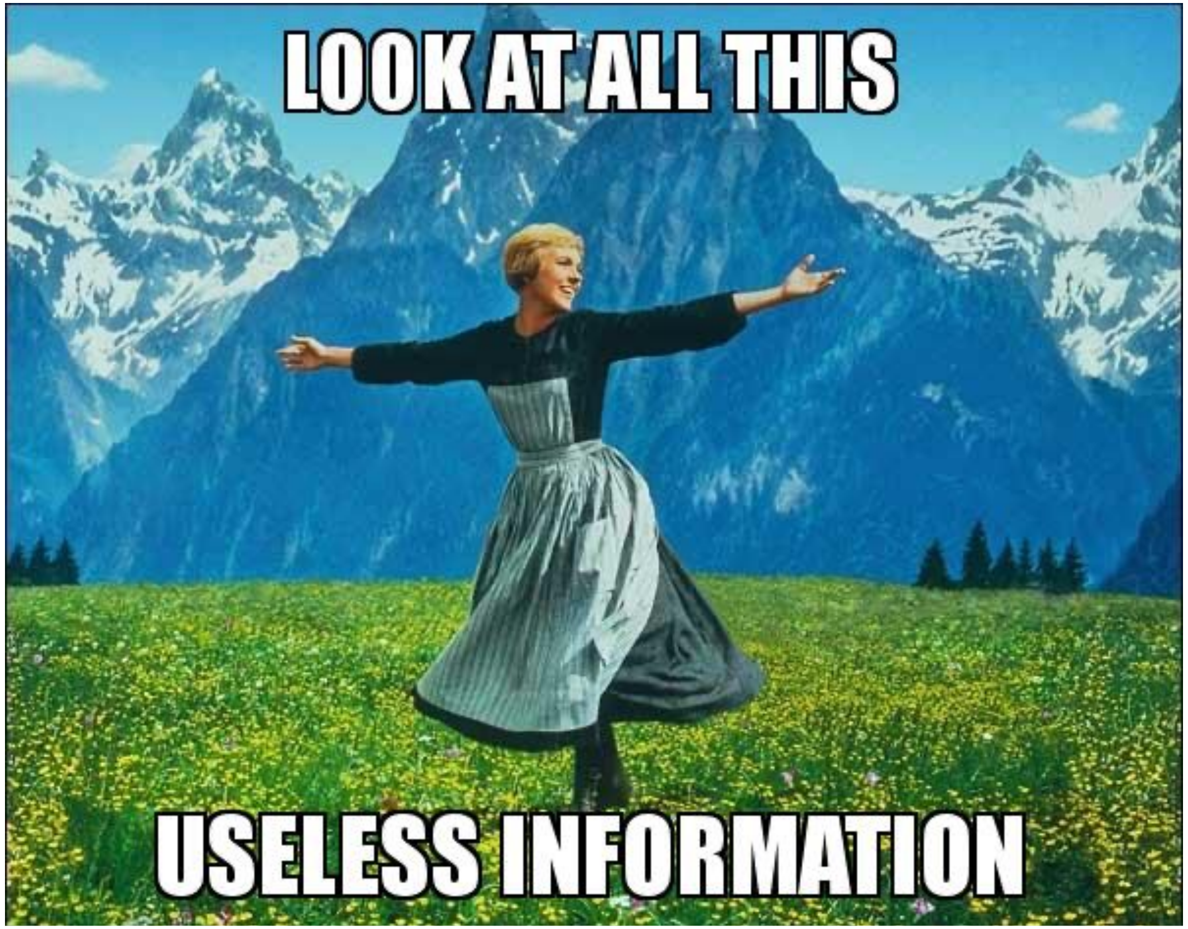
And we get the gradient by calling .backward().

Note: the graph is recomputed at every forward() call.

$$w.grad() = \frac{d_{out}}{d_w} = \frac{dx_4}{dx_3}\frac{dx_3}{dx_2}\frac{dx_2}{dx_0}$$

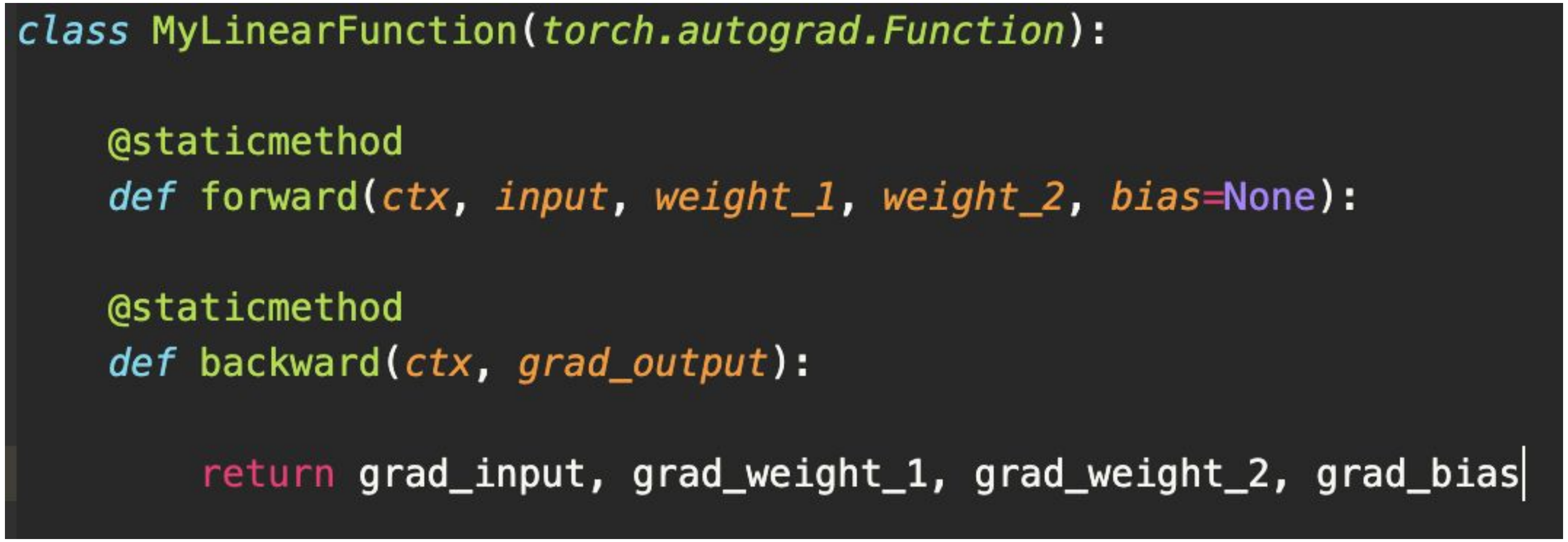
Automatic differentiation in PyTorch

But why?



Automatic differentiation in PyTorch

A practical example.



```python
class MyLinearFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input, weight_1, weight_2, bias=None):

    @staticmethod
    def backward(ctx, grad_output):

        return grad_input, grad_weight_1, grad_weight_2, grad_bias
```

The way you define a forward() function of a Module affects the way PyTorch computes the backward graph.

You may want to define your own derivatives.

Automatic differentiation in PyTorch

A practical example.

```
class QuaternionLinearFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input, r_weight, i_weight, j_weight, k_weight, bias=None):
        ctx.save_for_backward(input, r_weight, i_weight, j_weight, k_weight, bias) # Save the tensors in memory for backward
        cat_kernels_4_r = torch.cat([r_weight, -i_weight, -j_weight, -k_weight], dim=0)
        cat_kernels_4_i = torch.cat([i_weight,  r_weight, -k_weight, j_weight], dim=0)
        cat_kernels_4_j = torch.cat([j_weight,  k_weight, r_weight, -i_weight], dim=0)
        cat_kernels_4_k = torch.cat([k_weight,  -j_weight, i_weight, r_weight], dim=0)
        cat_kernels_4_quaternion = torch.cat([cat_kernels_4_r, cat_kernels_4_i, cat_kernels_4_j, cat_kernels_4_k], dim=1)
        output = torch.matmul(input, cat_kernels_4_quaternion)
        if bias is not None:
            return output+bias
        else:
            return output
```

This is a standard forward call of a ComplexLinear transformation.

PyTorch, to save time, will store the cat_kernels_4_quaternion matrix by default. This is extremely memory inefficient.

Automatic differentiation in PyTorch

A practical example.

```
class QuaternionLinearFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input, r_weight, i_weight, j_weight, k_weight, bias=None):
        ctx.save_for_backward(input, r_weight, i_weight, j_weight, k_weight, bias) # Save the tensors in memory for backward
        cat_kernels_4_r = torch.cat([r_weight, -i_weight, -j_weight, -k_weight], dim=0)
        cat_kernels_4_i = torch.cat([i_weight,  r_weight, -k_weight, j_weight], dim=0)
        cat_kernels_4_j = torch.cat([j_weight,  k_weight, r_weight, -i_weight], dim=0)
        cat_kernels_4_k = torch.cat([k_weight,  -j_weight, i_weight, r_weight], dim=0)
        cat_kernels_4_quaternion = torch.cat([cat_kernels_4_r, cat_kernels_4_i, cat_kernels_4_j, cat_kernels_4_k], dim=1)
        output = torch.matmul(input, cat_kernels_4_quaternion)
        if bias is not None:
            return output+bias
        else:
            return output
```

This is a standard forward call of a ComplexLinear transformation.

PyTorch, to save time, will store the cat_kernels_4_quaternion matrix by default. This is extremely memory inefficient.

The ctx call asks PyTorch to only store the components of this matrix so that we can do the efficient backward pass by ourselves.

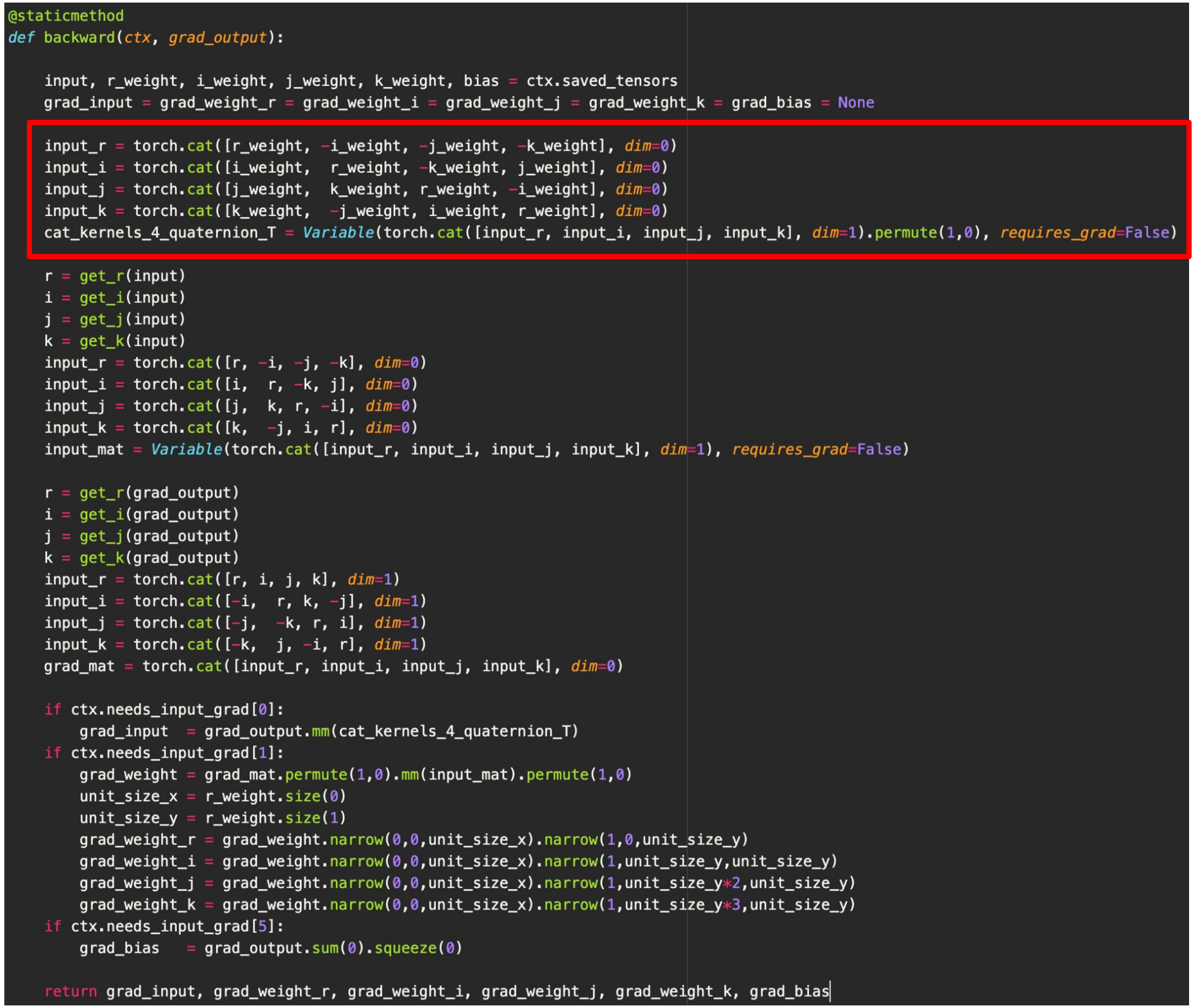
Automatic differentiation in PyTorch

A practical example.

Instead of storing the large matrix we take the tensors saved in the context and we reconstruct this matrix every time.

The rest of the operations are standard quaternion derivatives.

This change induces a reduction of two to three in the VRAM consumption while only slowing down the training by 20%.



```python
@staticmethod
def backward(ctx, grad_output):

    input, r_weight, i_weight, j_weight, k_weight, bias = ctx.saved_tensors
    grad_input = grad_weight_r = grad_weight_i = grad_weight_j = grad_weight_k = grad_bias = None

    input_r = torch.cat([r_weight, -i_weight, -j_weight, -k_weight], dim=0)
    input_i = torch.cat([i_weight,  r_weight, -k_weight, j_weight], dim=0)
    input_j = torch.cat([j_weight,  k_weight, r_weight, -i_weight], dim=0)
    input_k = torch.cat([k_weight,  -j_weight, i_weight, r_weight], dim=0)
    cat_kernels_4_quaternion_T = Variable(torch.cat([input_r, input_i, input_j, input_k], dim=1).permute(1,0), requires_grad=False)

    r = get_r(input)
    i = get_i(input)
    j = get_j(input)
    k = get_k(input)
    input_r = torch.cat([r, -i, -j, -k], dim=0)
    input_i = torch.cat([i,  r, -k, j], dim=0)
    input_j = torch.cat([j,  k, r, -i], dim=0)
    input_k = torch.cat([k,  -j, i, r], dim=0)
    input_mat = Variable(torch.cat([input_r, input_i, input_j, input_k], dim=1), requires_grad=False)

    r = get_r(grad_output)
    i = get_i(grad_output)
    j = get_j(grad_output)
    k = get_k(grad_output)
    input_r = torch.cat([r, i, j, k], dim=1)
    input_i = torch.cat([-i,  r, k, -j], dim=1)
    input_j = torch.cat([-j,  -k, r, i], dim=1)
    input_k = torch.cat([-k,  j, -i, r], dim=1)
    grad_mat = torch.cat([input_r, input_i, input_j, input_k], dim=0)

    if ctx.needs_input_grad[0]:
        grad_input  = grad_output.mm(cat_kernels_4_quaternion_T)
    if ctx.needs_input_grad[1]:
        grad_weight = grad_mat.permute(1,0).mm(input_mat).permute(1,0)
        unit_size_x = r_weight.size(0)
        unit_size_y = r_weight.size(1)
        grad_weight_r = grad_weight.narrow(0,0,unit_size_x).narrow(1,0,unit_size_y)
        grad_weight_i = grad_weight.narrow(0,0,unit_size_x).narrow(1,unit_size_y,unit_size_y)
        grad_weight_j = grad_weight.narrow(0,0,unit_size_x).narrow(1,unit_size_y*2,unit_size_y)
        grad_weight_k = grad_weight.narrow(0,0,unit_size_x).narrow(1,unit_size_y*3,unit_size_y)
    if ctx.needs_input_grad[5]:
        grad_bias   = grad_output.sum(0).squeeze(0)

    return grad_input, grad_weight_r, grad_weight_i, grad_weight_j, grad_weight_k, grad_bias
```

In brief

1. Gradients give the direction of the steepest change of a differentiable multivariable function at a certain point.
2. Diff. methods are: Manual, Numeric, Symbolic or Automatic.
3. AD has two modes: forward and reverse.
4. Most frameworks use reverse mode AD.
5. The idea is to create a forward compute graph and differentiate it using the Chain Rule from the output to the variable of interest.

To go beyond the lecture

1. https://e-dorigatti.github.io/math/deep%20learning/2020/04/07/autodiff.html
2. https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
3. https://www.jmlr.org/papers/volume18/17-468/17-468.pdf