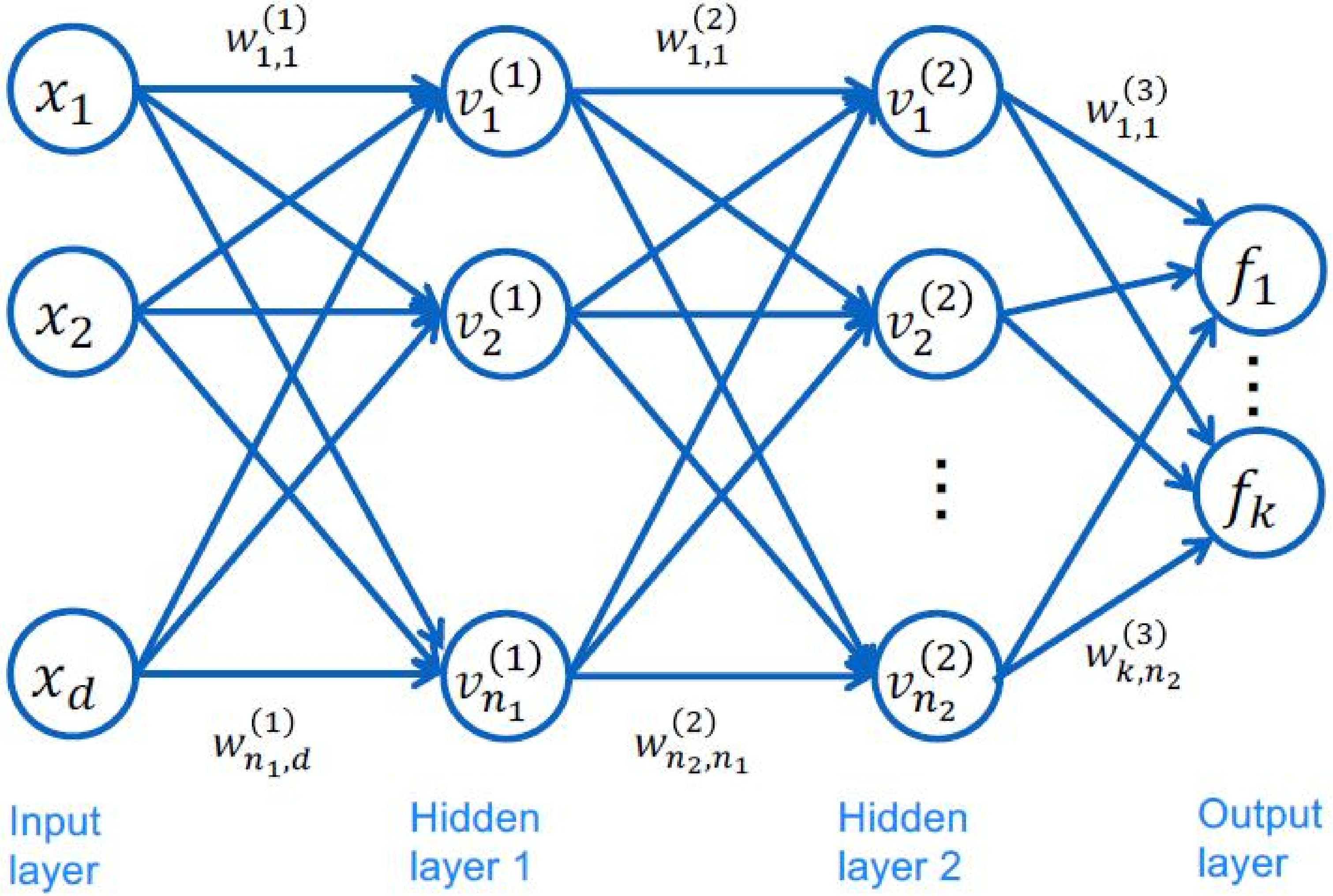
# Introduction to Deep Learning
## Calculus behind Backpropagation

Lecturer: Dongyu Yao


$x_1$
$x_2$
$x_d$
$w_{1,1}^{(1)}$
$w_{n_1,d}^{(1)}$
$v_1^{(1)}$
$v_2^{(1)}$
$v_{n_1}^{(1)}$
$w_{1,1}^{(2)}$
$w_{n_2,n_1}^{(2)}$
$v_1^{(2)}$
$v_2^{(2)}$
$v_{n_2}^{(2)}$
$w_{1,1}^{(3)}$
$w_{k,n_2}^{(3)}$
$f_1$
$f_k$
Input layer
Hidden layer 1
Hidden layer 2
Output layer

# Forward propagation (short notation, with bias nodes)

- For input layer: $\boldsymbol{v}^{(0)} = [\boldsymbol{x};\ 1]$
- For each hidden layer $l = 1{:}L-1$

$$\mathbf{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{v}^{(l-1)} \quad \text{and} \quad \boldsymbol{v}^{(l)} = [\varphi(\mathbf{z}^{(l)});\ \ 1]$$

- For output layer:

$$\boldsymbol{f} = \boldsymbol{W}^{(L)}\boldsymbol{v}^{(L-1)}$$

- Predict $\boldsymbol{y} = \boldsymbol{f}$ for regression, or $y = \arg\max_j f_j$ for classification

# Computing the gradient

In order to apply SGD, need to compute

$$\nabla_{\boldsymbol{W}} \ell(\boldsymbol{W}; \boldsymbol{x}, y)$$

I.e., for each weight between any two connected units $i$ and $j$ on layer $l$ need

$$\frac{\partial}{\partial_{w_{i,j}^{(l)}}} \ell(\boldsymbol{W}; \boldsymbol{x}, y)$$

# Simple example

ANN with 1 output, 1 hidden and 1 input unit: $f(x;W) = f(x;[w,w']) = \underbrace{w\overbrace{\varphi(\overbrace{w'x}^{z})}^{v}}_{f}$

$$x \xrightarrow{w'} v \xrightarrow{w} f$$

Loss: $\ell(W; x,y)$, e.g. $\ell(W; x,y) = (y - f(x;W))^2 =: \ell_y(f)$

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial w} = \underbrace{\ell_y'(f)}_{\delta} \cdot v$$

Computed during forward pass

$$\frac{\partial \ell}{\partial w'} = \overbrace{\underbrace{\frac{\partial \ell}{\partial f}}_{\delta} \cdot \underbrace{\frac{\partial f}{\partial v}}_{w} \cdot \underbrace{\frac{\partial v}{\partial z}}_{\varphi'(z)}}^{\delta'} \cdot \underbrace{\frac{\partial z}{\partial w'}}_{x}$$

Want:

$$\nabla_W \ell(W; x,y) = \left[\frac{\partial \ell}{\partial w}, \frac{\partial \ell}{\partial w'}\right]^T$$

$z = w' \cdot x$

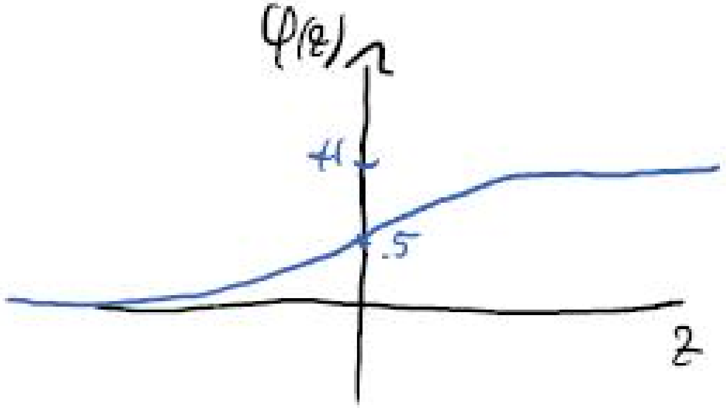
$v = \varphi(z)$

$f = w \cdot v$

$\ell_y'(f) = 2(f-y) =: \delta$

# Derivatives of activation functions

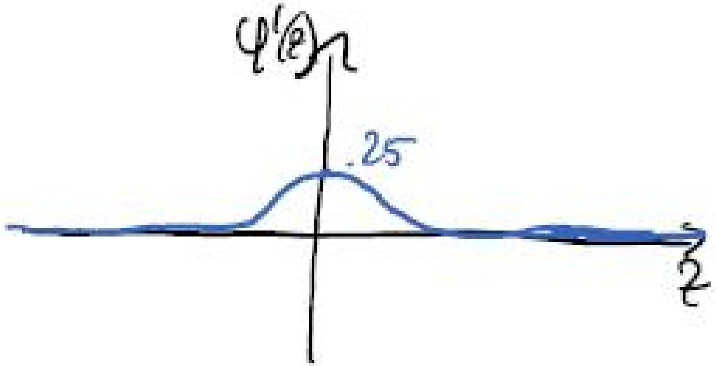
Sigmoid: $\varphi(z) = \frac{1}{1+\exp(-z)}$




$$\varphi'(z) = \frac{-1}{(1+e^{-z})^2} \cdot e^{-z} \cdot (-1) = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$= \frac{e^{-z}}{(1+e^{-z})} \cdot \frac{1}{(1+e^{-z})} = (1-\varphi(z))\ \varphi(z)$$

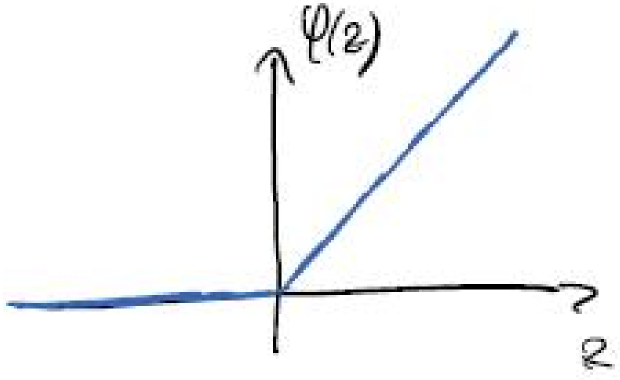
$$(1-v) \quad v$$



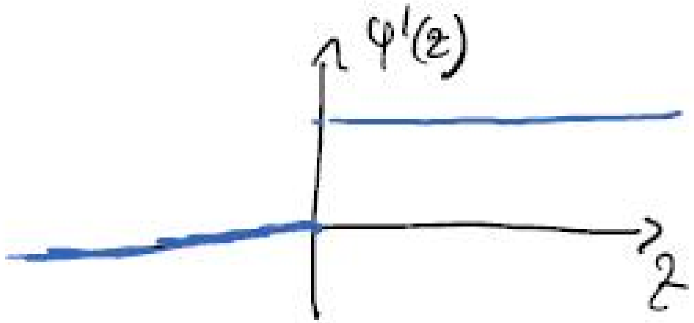

+ $\varphi'(z)$ is easy to compute given $\varphi(z) = v$
+ $\varphi'(z) \neq 0\ \forall z$, defined everywhere
− $\varphi'(z) \approx 0\ \forall z$ (except $z \approx 0$)

# Derivatives of activation functions

Rectified Linear Unit (ReLU): $\varphi(z) = \max(z, 0)$




$$\varphi'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$$




---

- not diff'able at $z=0$
  doesn't matter much in practice;
  e.g. simply define $\varphi'(0) := 0$
+ even easier to compute than sigmoid
+ $\neq 0 \quad \forall z > 0$

# Recall: Jacobians & multivariate chain rule

For a function $f: \mathbb{R}^n \to \mathbb{R}^m$ its Jacobian at x is given by

$$\frac{d}{d\boldsymbol{x}} f(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} =: J_f(\boldsymbol{x}) =: \frac{\partial f}{\partial \boldsymbol{x}}$$

Multivariate Taylor series: Given differentiable f, it holds that

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_0) + J_f(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0) + o\left(||\boldsymbol{x} - \boldsymbol{x}_0||\right)$$

Multivariate chain rule: Given differentiable $f: \mathbb{R}^n \to \mathbb{R}^m$ and $g: \mathbb{R}^m \to \mathbb{R}^p$, it holds that

$$\frac{d}{d\boldsymbol{x}} g\left(f(\boldsymbol{x})\right) = J_{g \circ f}(\boldsymbol{x}) = J_g\left(f(\boldsymbol{x})\right) J_f(\boldsymbol{x}) = \frac{\partial g}{\partial f} \frac{\partial f}{\partial \boldsymbol{x}}$$

Examples

$$z = z(x) = Wx \quad \Rightarrow \quad \overbrace{J_z(x)}^{\frac{\partial z}{\partial x}} = W$$

$$\varphi'(x) = [\varphi'(x_1) \dots \varphi'(x_n)]$$

$$\varphi(x) = [\varphi(x_1), \dots, \varphi(x_n)]^T \quad \Rightarrow \quad J_\varphi(x) = \overbrace{\begin{pmatrix} \varphi'(x_1) & 0 & \cdots & 0 \\ & \varphi'(x_2) & & 0 \\ 0 & & \ddots & \varphi'(x_n) \end{pmatrix}}^{diag(\varphi'(x))}$$

$$f(x) = \varphi(\underbrace{Wx}_{z}) \quad \Rightarrow \quad J_f(x) = J_\varphi(z)\, J_z(x) = diag(\varphi'(z)) \cdot W$$

# Backpropagation

Consider neural network $f(x;W)$ with weights $W = [W^{(1)}, W^{(2)}, \dots, W^{(L)}]$

Want to compute gradient of loss $\ell(W;x,y)$ w.r.t., weights $W^{(i)}$

$$(\nabla_{W^{(L)}}\ell)^T = \frac{\partial \ell}{\partial W^{(L)}} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial W^{(L)}}$$

$$(\nabla_{W^{(L-1)}}\ell)^T = \frac{\partial \ell}{\partial W^{(L-1)}} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial W^{(L-1)}}$$

$$(\nabla_{W^{(L-2)}}\ell)^T = \frac{\partial \ell}{\partial W^{(L-2)}} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial z^{(L-2)}}\frac{\partial z^{(L-2)}}{\partial W^{(L-2)}}$$

$$\vdots$$

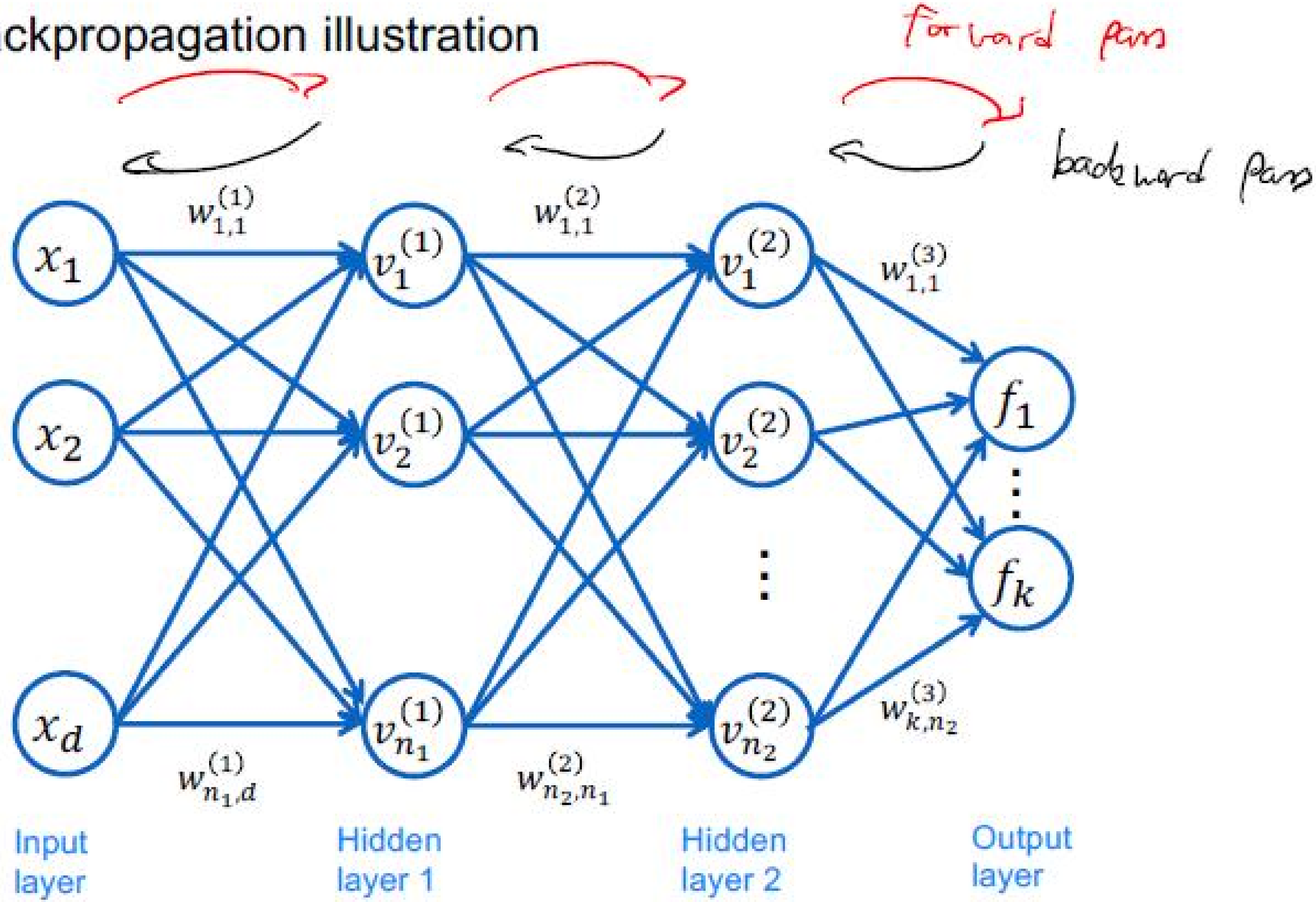
$$(\nabla_{W^{(i)}}\ell)^T = \frac{\partial \ell}{\partial W^{(i)}} = \frac{\partial \ell}{\partial f}\frac{\partial f}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial z^{(L-2)}}\cdots\frac{\partial z^{(i+1)}}{\partial z^{(i)}}\frac{\partial z^{(i)}}{\partial W^{(i)}}$$

Key insight: Can reuse computations from forward propagation and from layer i+1 to compute $W^{(i)}$

# Backpropagation

1. For each unit $j$ on the output layer
   - Compute error signal $\delta_j^{(L)} = \frac{\partial \ell}{\partial f_j}$
   - For each unit $i$ on layer $L$, $\frac{\partial}{\partial_{w_{j,i}^{(L)}}} \ell = \delta_j^{(L)} v_i^{(L-1)}$
2. For each unit $j$ on hidden layer $l = L - 1 : -1 : 1$
   - Compute error signal $\delta_j^{(l)} = \varphi'\left(z_j^{(l)}\right) \sum_{i=1}^{n_{l+1}} w_{i,j}^{(l+1)} \delta_i^{(l+1)}$
   - For each unit $i$ on layer $l$: $\frac{\partial}{\partial_{w_{j,i}^{(l)}}} \ell = \delta_j^{(l)} v_i^{(l-1)}$

# Backpropagation illustration



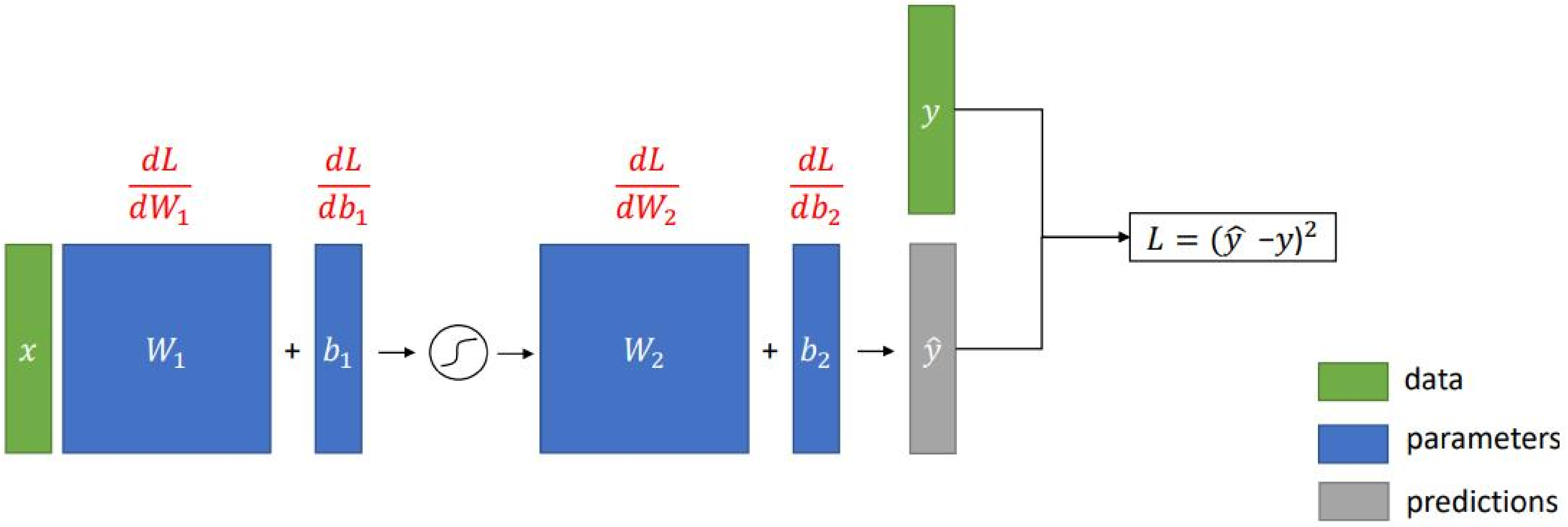
# Autodifferentiation Example

Modern deep learning frameworks (eg, TensorFlow, PyTorch) allow to specify the computation graph (neural network architecture), and then automatically calculate gradients!

```
model = torch.nn.Sequential(        # define ANN with 5 input, 3 hidden
        torch.nn.Linear(5, 3),      # units and ReLU activations
        torch.nn.ReLU(),
        torch.nn.Linear(3,1),
        torch.nn.Flatten(0, 1)
)
loss_fn = torch.nn.MSELoss()        # train with gradient descent on squared loss
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
for t in range(1000):
        y_pred = model(x)           # forward pass
        loss = loss_fn(y_pred, y)   # compute and print loss
        optimizer.zero_grad()       # initialize gradient
        loss.backward()             # compute gradient via backprop
        optimizer.step()            # parameter update
```

# What is Autodiff?

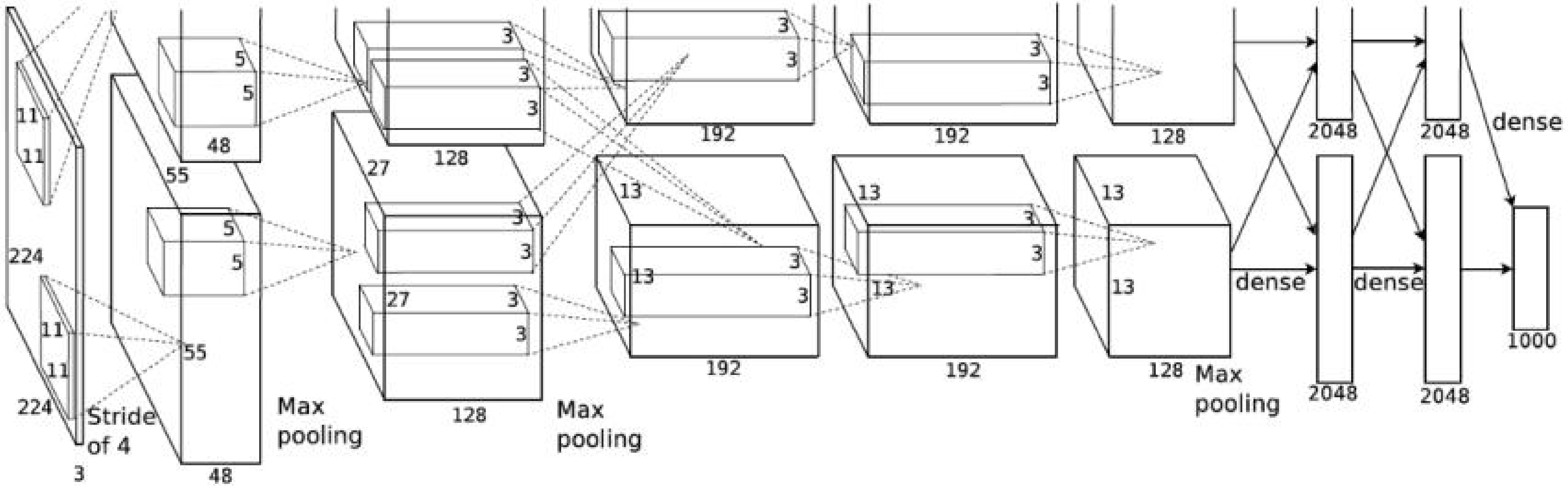
- We only need to implement the **forward pass** (i.e., chain of operations from input to output)
- PyTorch automatically computes the derivatives for us → **backward pass**



# Why Autodiff?

- Neural network architectures can be quite complicated

  → Deriving the gradients by hand is tedious, error prone and computationally less efficient than Autodiff



# How does Autodiff work?

## 1) Forward Pass:

- Construct computation graph (DAG)
- Store intermediate computed values at each node of the graph

```
def f(x, y):
  c = x + y
  d = c**2
  return x * d
```

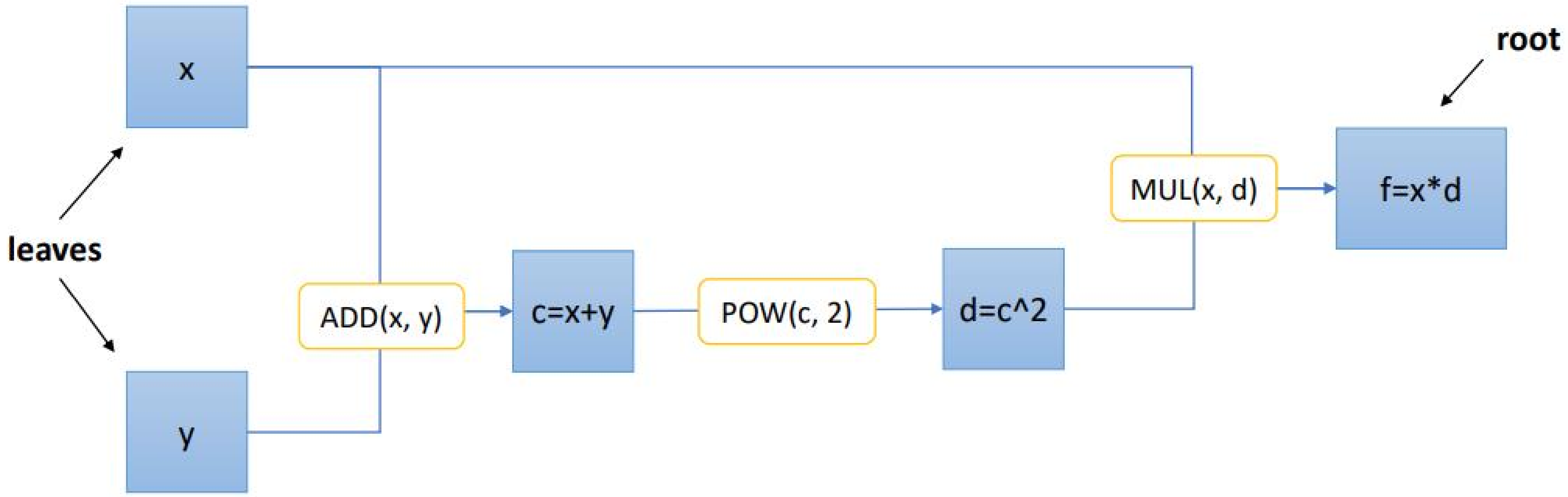
```
x = torch.tensor([3.0], requires_grad=True)
y = torch.tensor([-1.0], requires_grad=True)
f_val = f(x, y)
```

# How does Autodiff work?

## 2) Backward Pass:

`f_val.backward()`

- Go back from the root (i.e. loss) to the leafs (i.e. parameters)



# How does Autodiff work?

## 2) Backward Pass:

- Go back from the root (i.e. loss) to the leaves (i.e. parameters)
- Each elementary operation has a 'backward function' that computes its derivatives
- Apply **chain rule** along each path

$$\frac{d\,f(g(x))}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

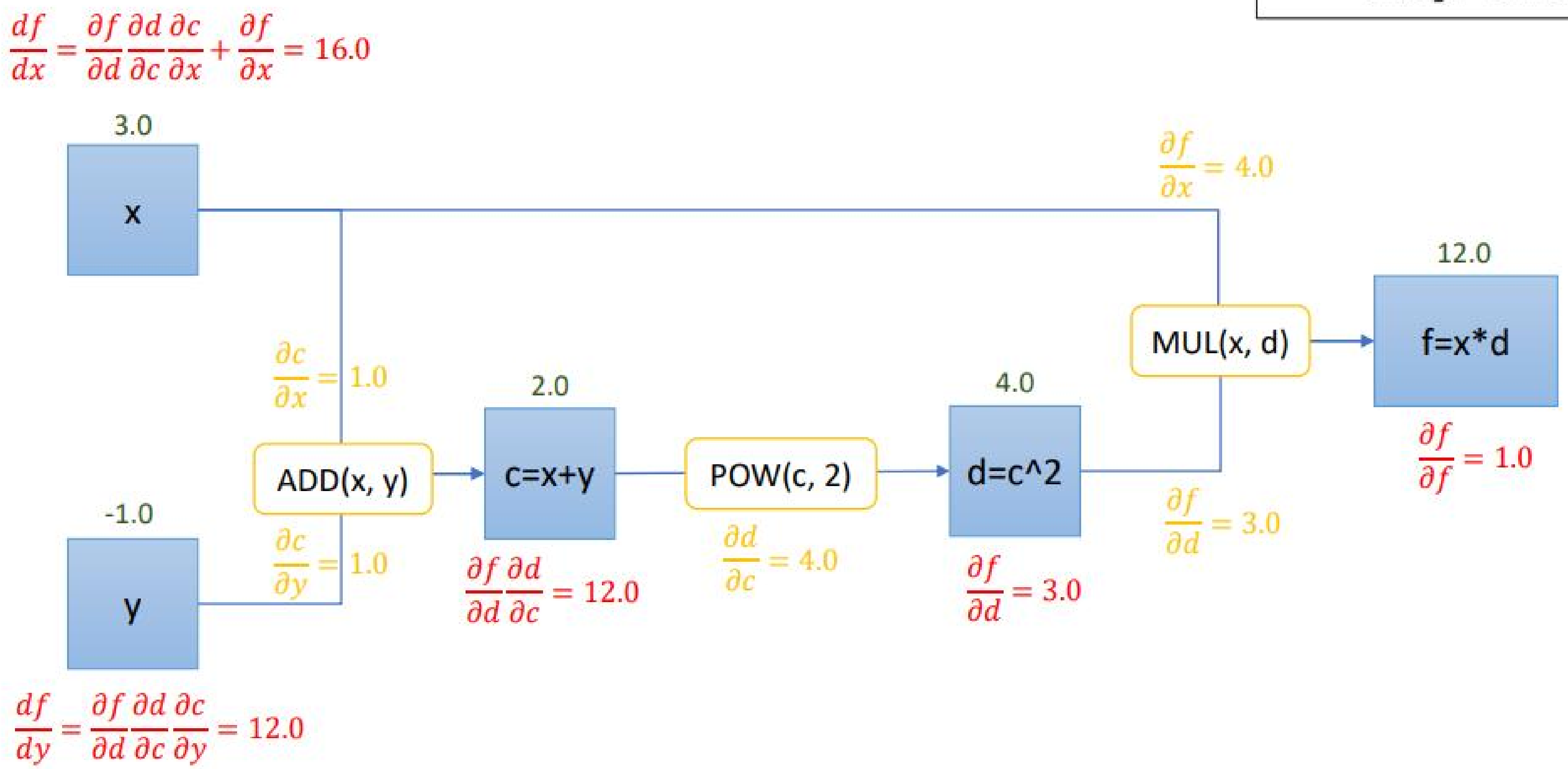
→ Make use of stored intermediate computation results from forward pass

- Total derivative is **sum of all path derivatives** from the root to the leaf

# Backward Pass

```
[31] f_val.backward()
     print('df/dx:', x.grad)
     print('df/dy:', y.grad)

     df/dx: tensor([16.])
     df/dy: tensor([12.])
```



# Other Resource:

https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

Review on Backpropagation

Video tutorials:
https://www.youtube.com/watch?v=Ilg3gGewQ5U
https://www.youtube.com/watch?v=tIeHLnjs5U8
https://www.youtube.com/watch?v=i94OvYb6noo

Thank You!