Backpropagation

LING 572 Advanced Statistical Methods in NLP

March 2 2020

Outline

- Computation graphs, chain rule
- Backpropagation

Computation graphs

Derivative chain rule

$$z = f(y) \quad y = g(x)$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

or

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$$

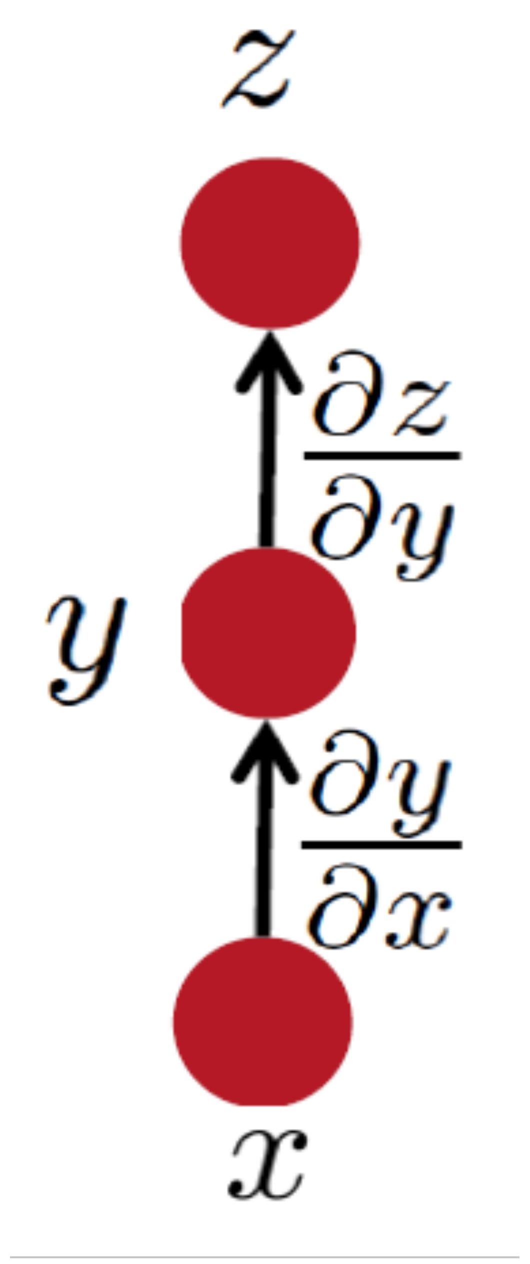
$z = (x^2+3)^4$

$z = y^4 \quad y=x^2+3$

$dz/dx = 4(x^2+3)^3 * 2x$

$dz/dy = 4y^3 \quad dy/dx=2x$

Simple chain rule



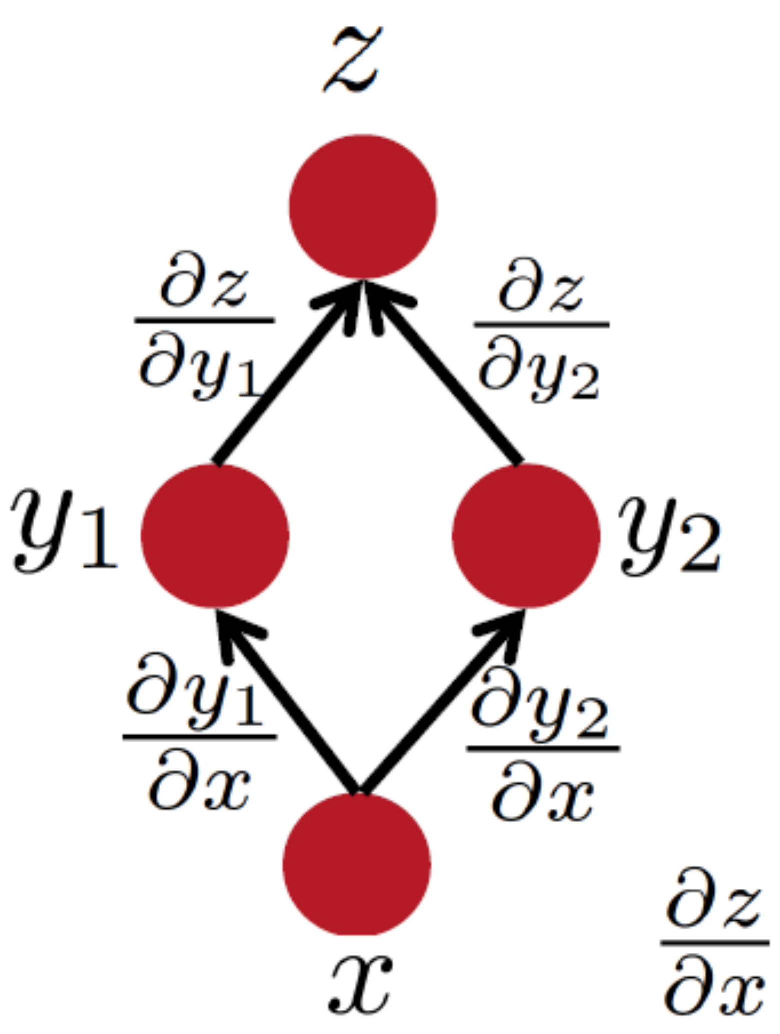
$$\Delta z = \frac{\partial z}{\partial y} \Delta y$$

$$\Delta y = \frac{\partial y}{\partial x} \Delta x$$

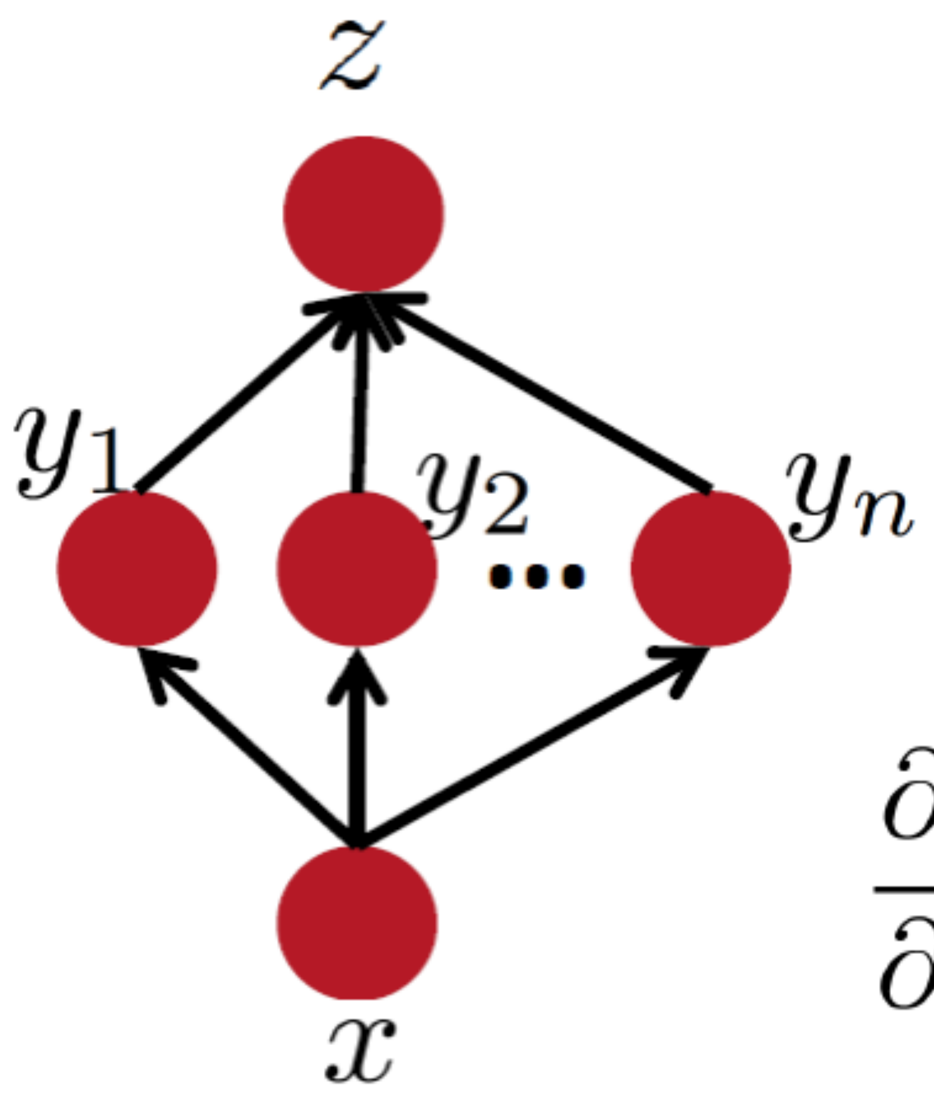
$$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Multiple paths chain rule

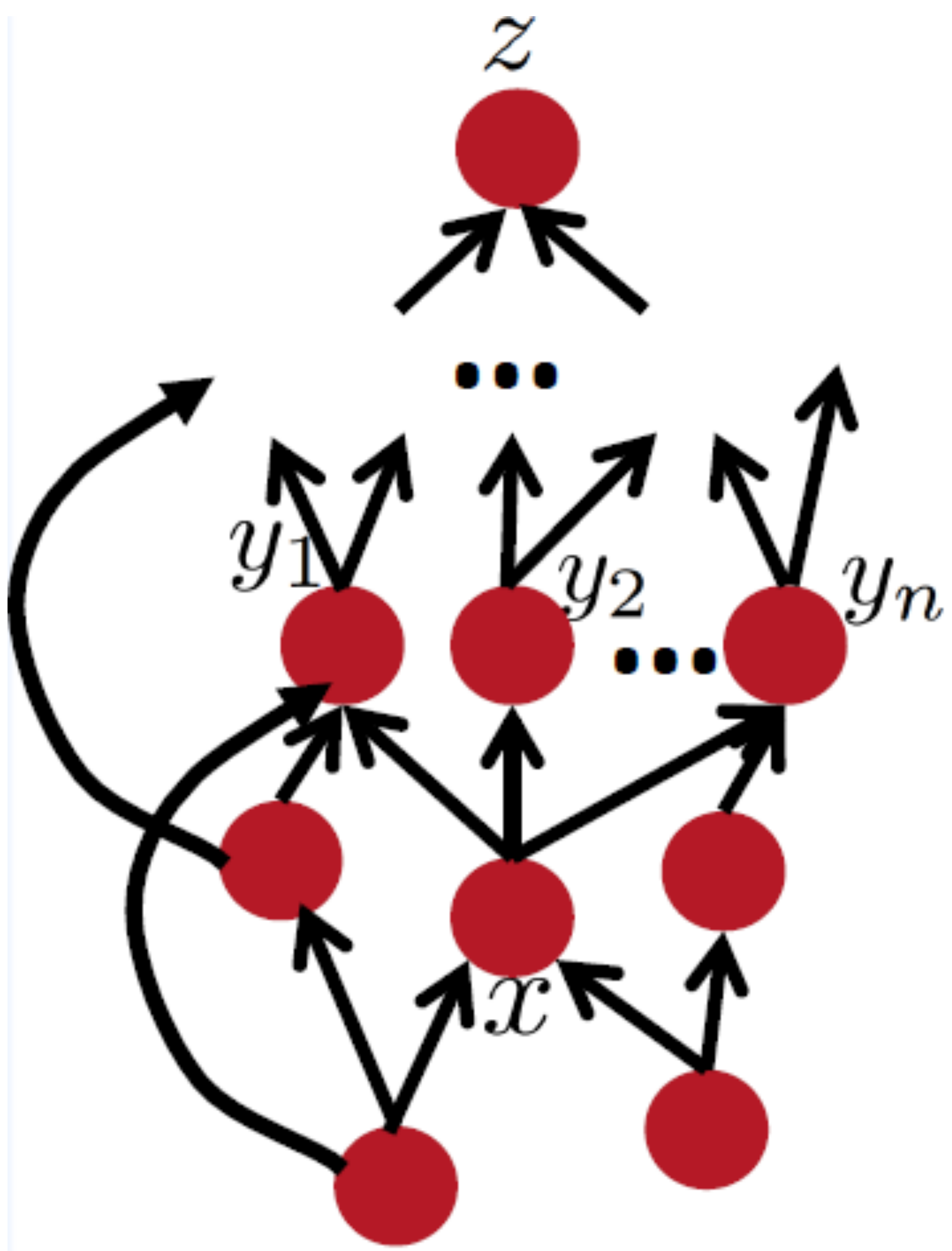


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

Chain rule in computation graph



Flow graph: any directed acyclic graph

node = computation result

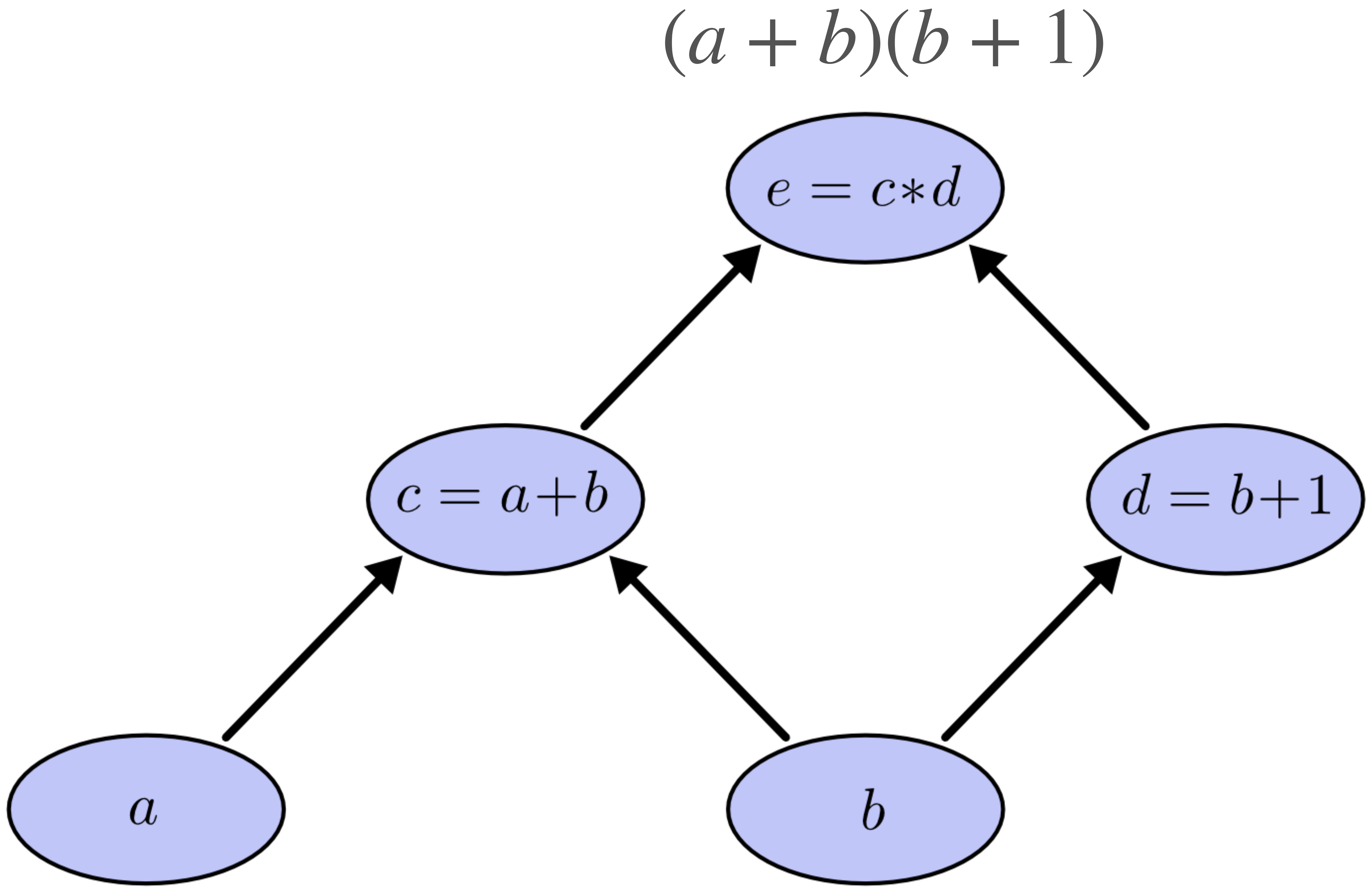
arc = computation dependency

$\{y_1, \; y_2, \; \ldots \; y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

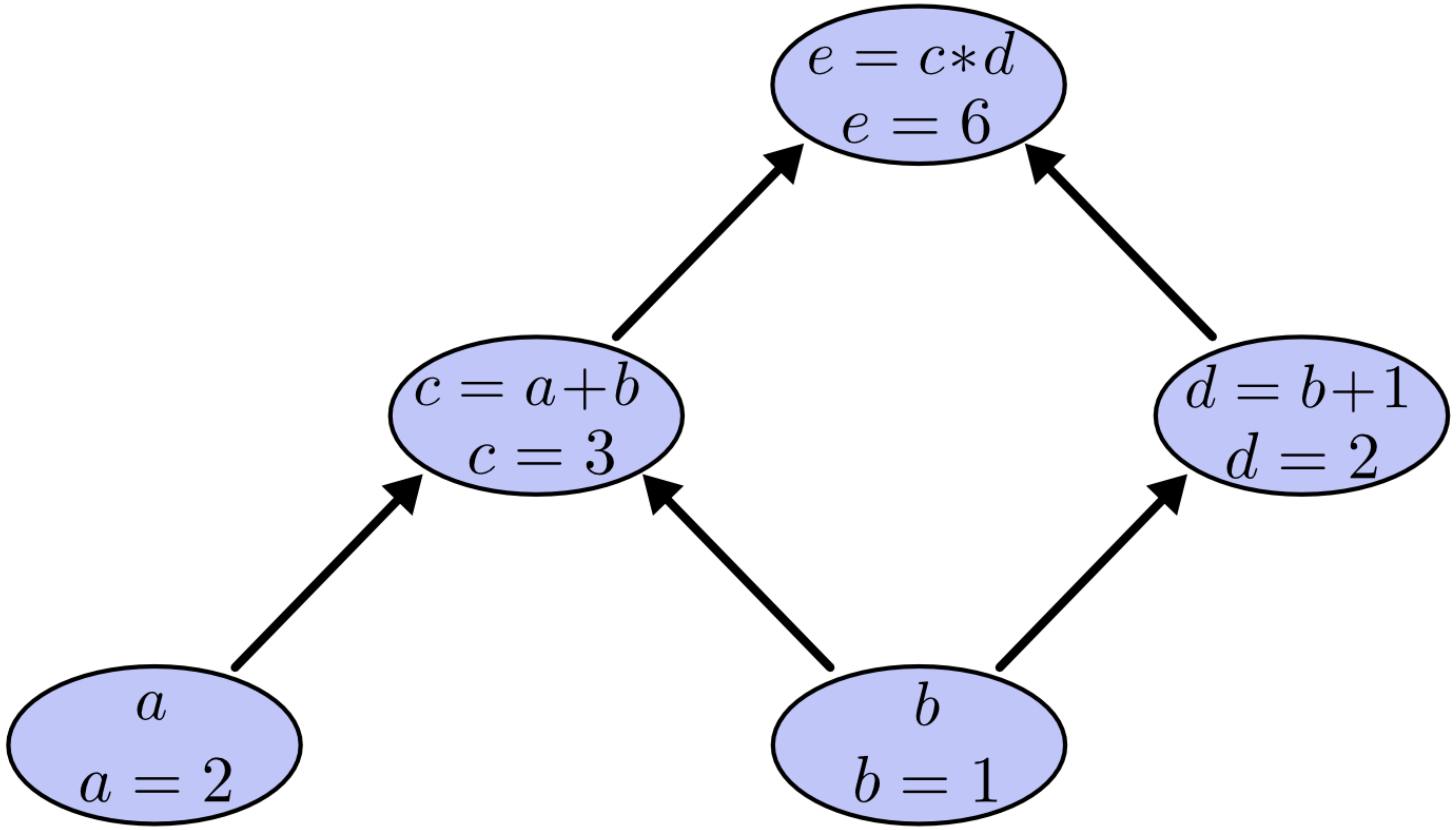
Computation graph

- Nodes: operations
- Edges: dependence
  - From A to B: output of A is an input of B



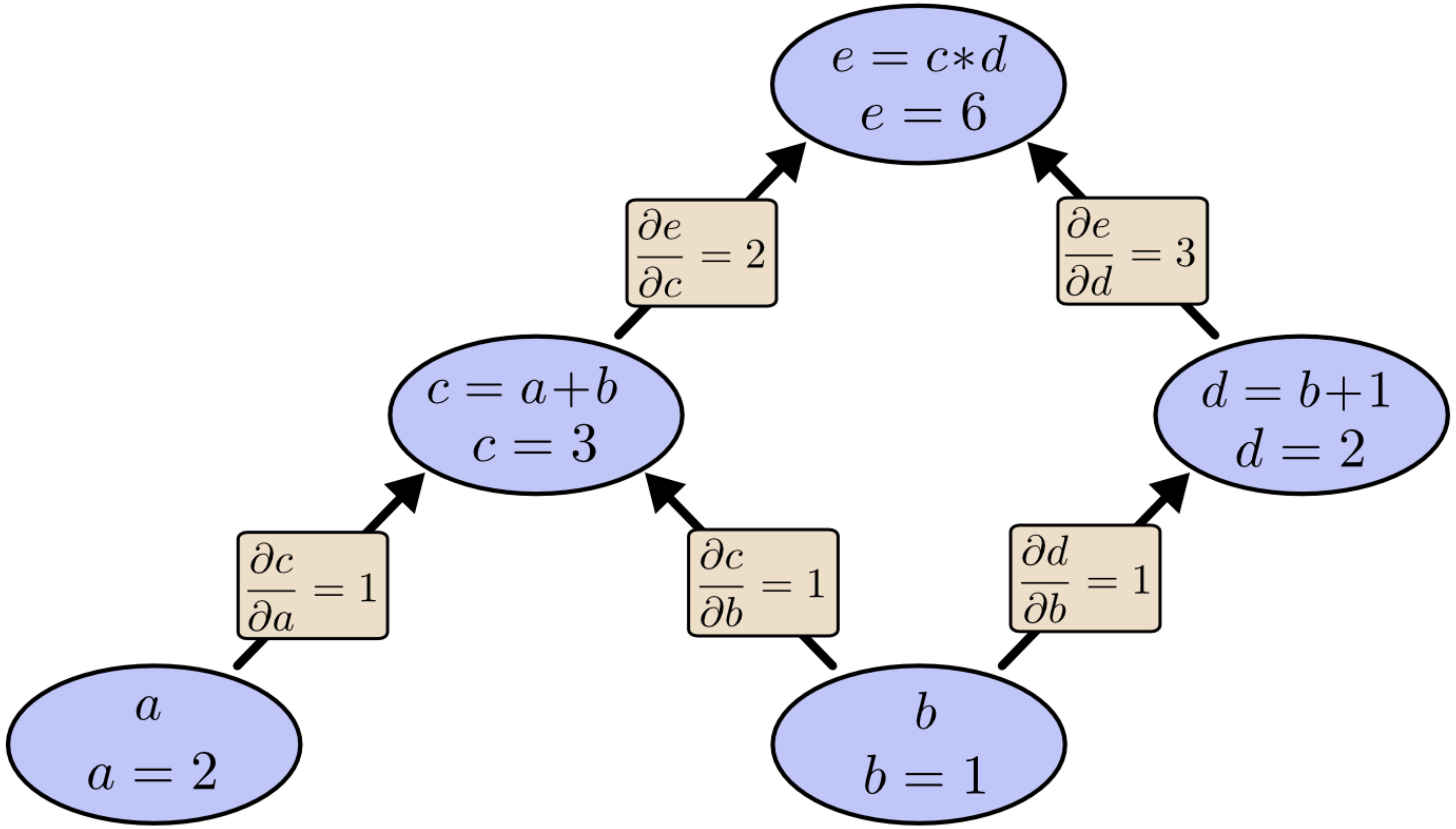
source

Computation Graph: Forward Pass



source

Computation Graph: Derivatives

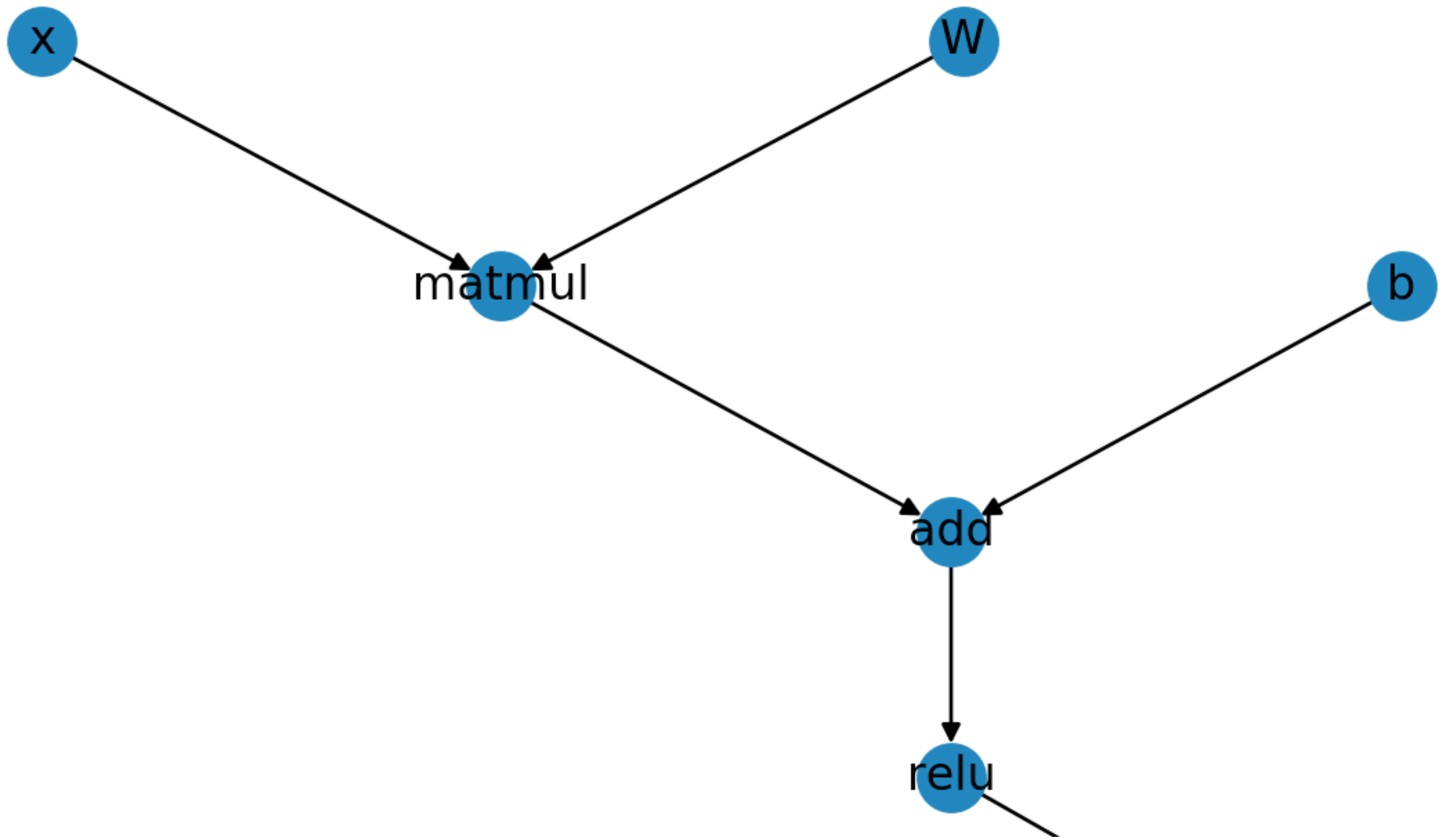


source

Computation Graph: operation API

```python
class Operation:

    def __init__(self, value=None, grads=None, name=None):
        self.value = value
        self.grads = grads
        self.name = name

    def forward(self, *args):
        raise NotImplementedError

    def backward(self, output_grad):
        raise NotImplementedError

    def __call__(self, *args):
        value = self.forward(*args)
        self.value = value
        return value
```

Computation Graph: feedforward layer

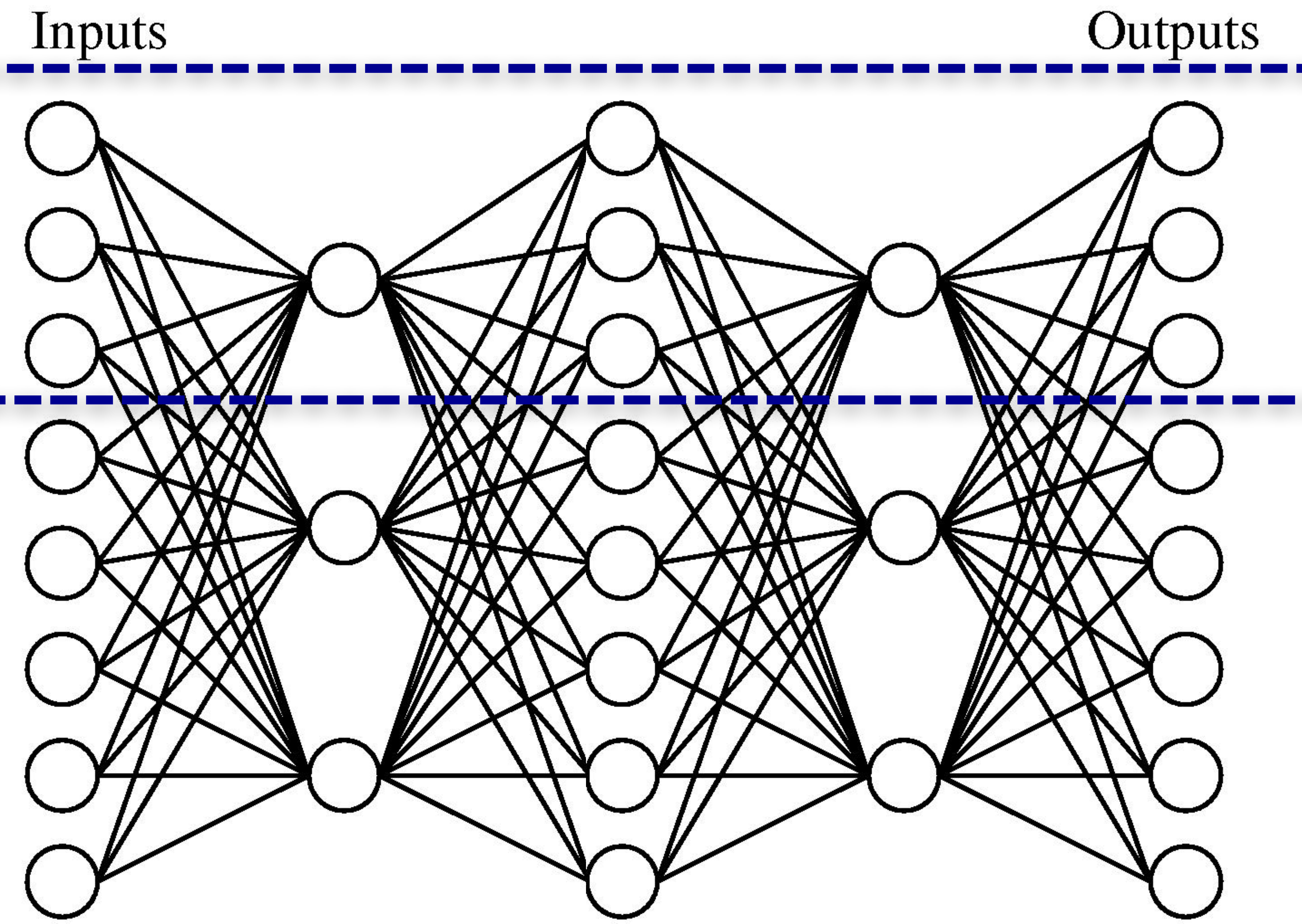


Outline

- Computation graphs, chain rule
- Backpropagation

Computing gradients in NNs: Backpropagation

Forward



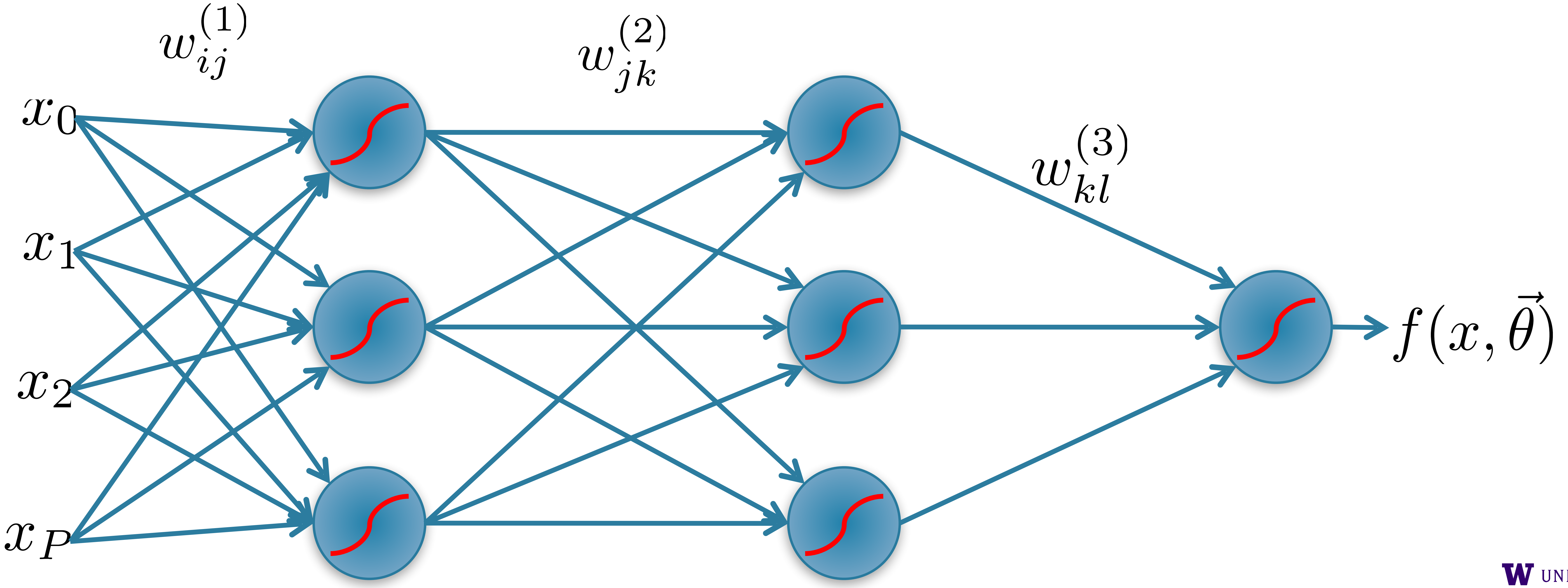
Gradient

Next 10 slides on back propagation are adapted from Andrew Rosenberg

Error Backpropagation

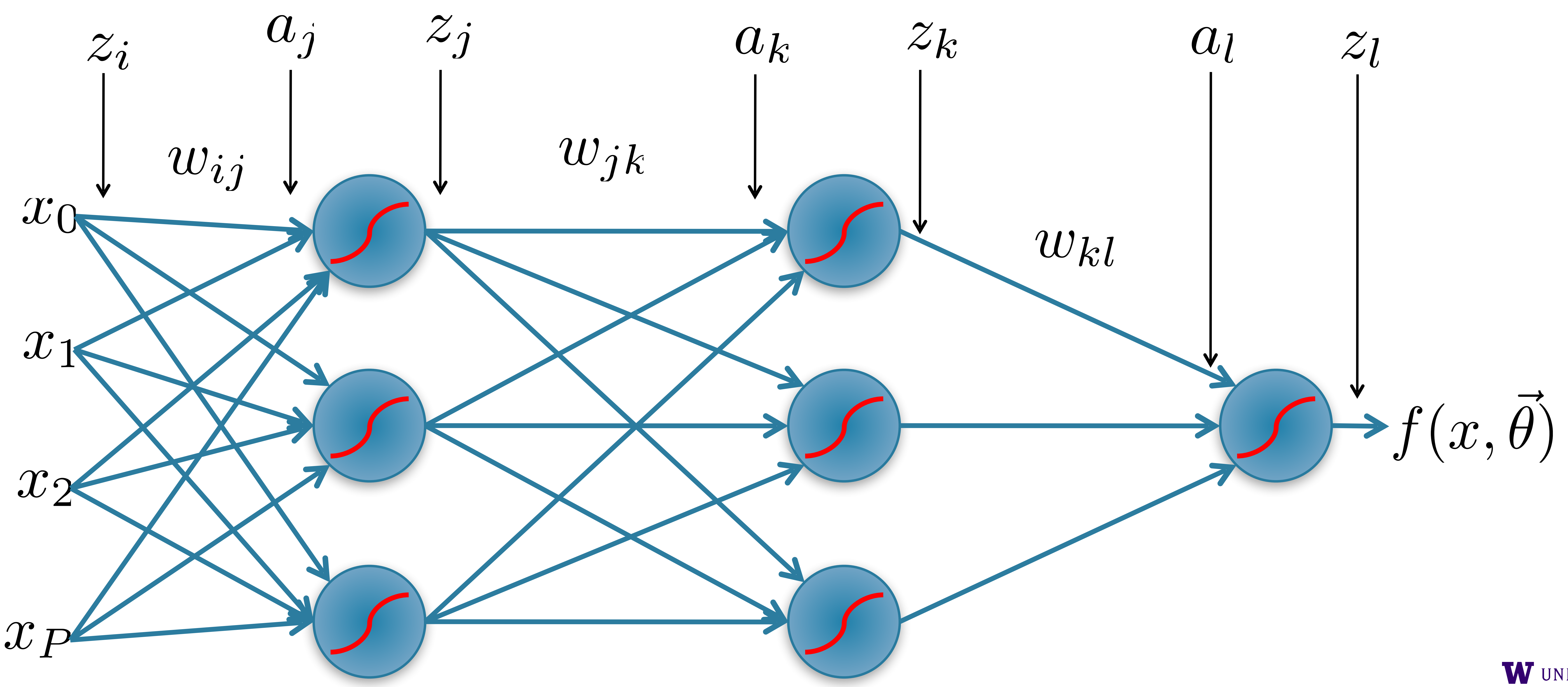
- Model parameters: $\vec{\theta} = \{w_{ij}^{(1)}, w_{jk}^{(2)}, w_{kl}^{(3)}\}$

  for brevity: $\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$



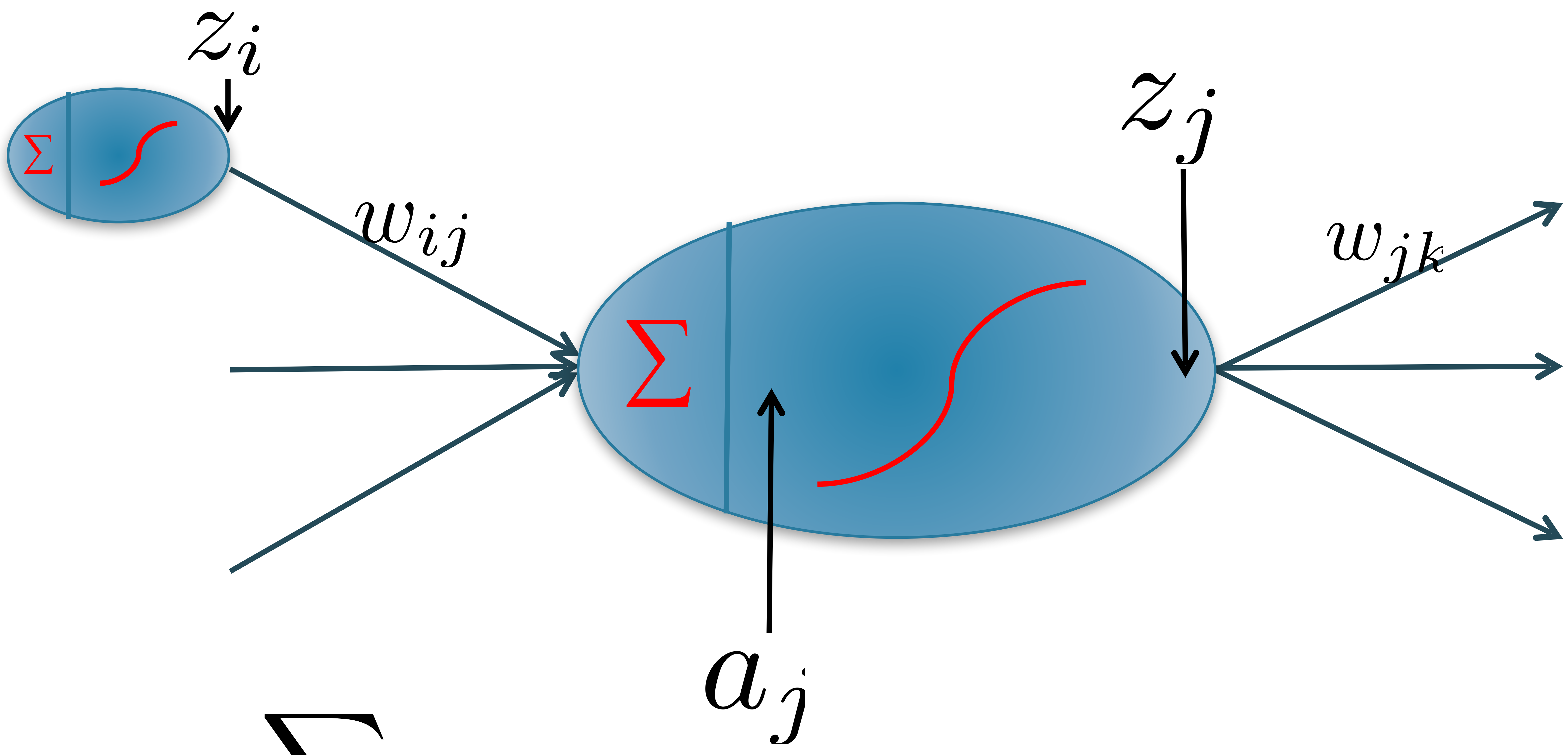
Error Backpropagation

- Model parameters: $\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$
- *Let a and z be the input and output of each node*



Error Backpropagation

$$z_j = g(a_j)$$

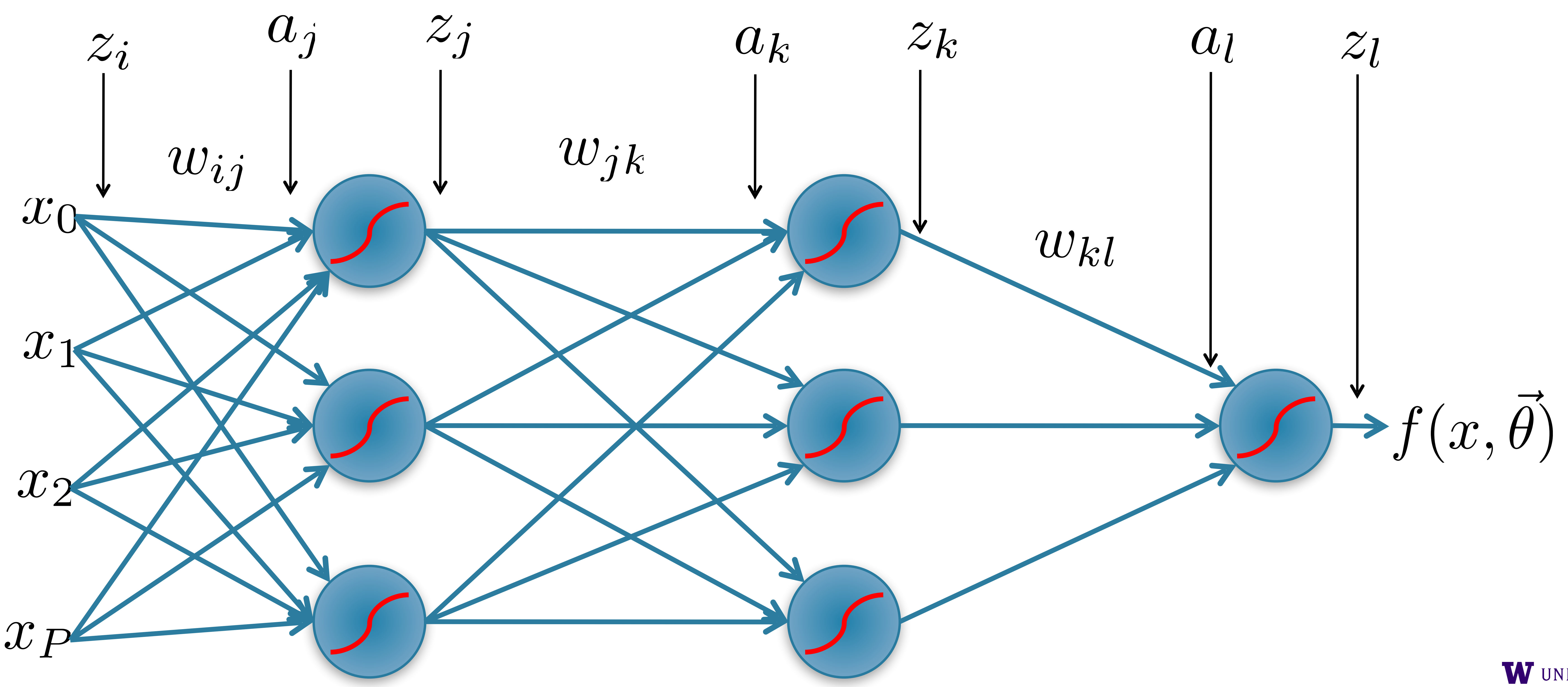


$$a_j = \sum_i w_{ij} z_i$$

- *Let* a and z *be the input and output of each node*

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \boxed{} \qquad a_l = \boxed{}$$

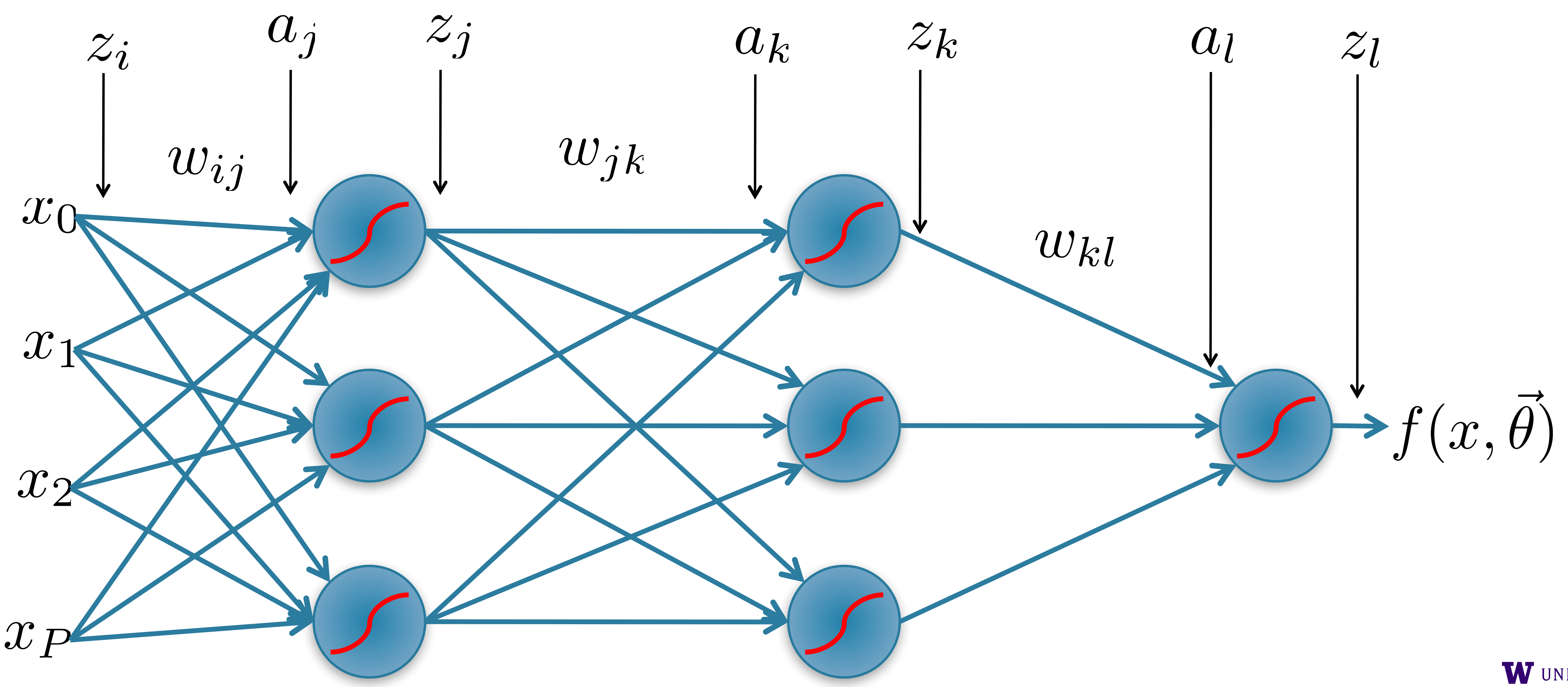
$$z_j = g(a_j) \qquad z_k = \boxed{} \qquad z_l = \boxed{}$$



- *Let* a and z *be the input and output of each node*

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



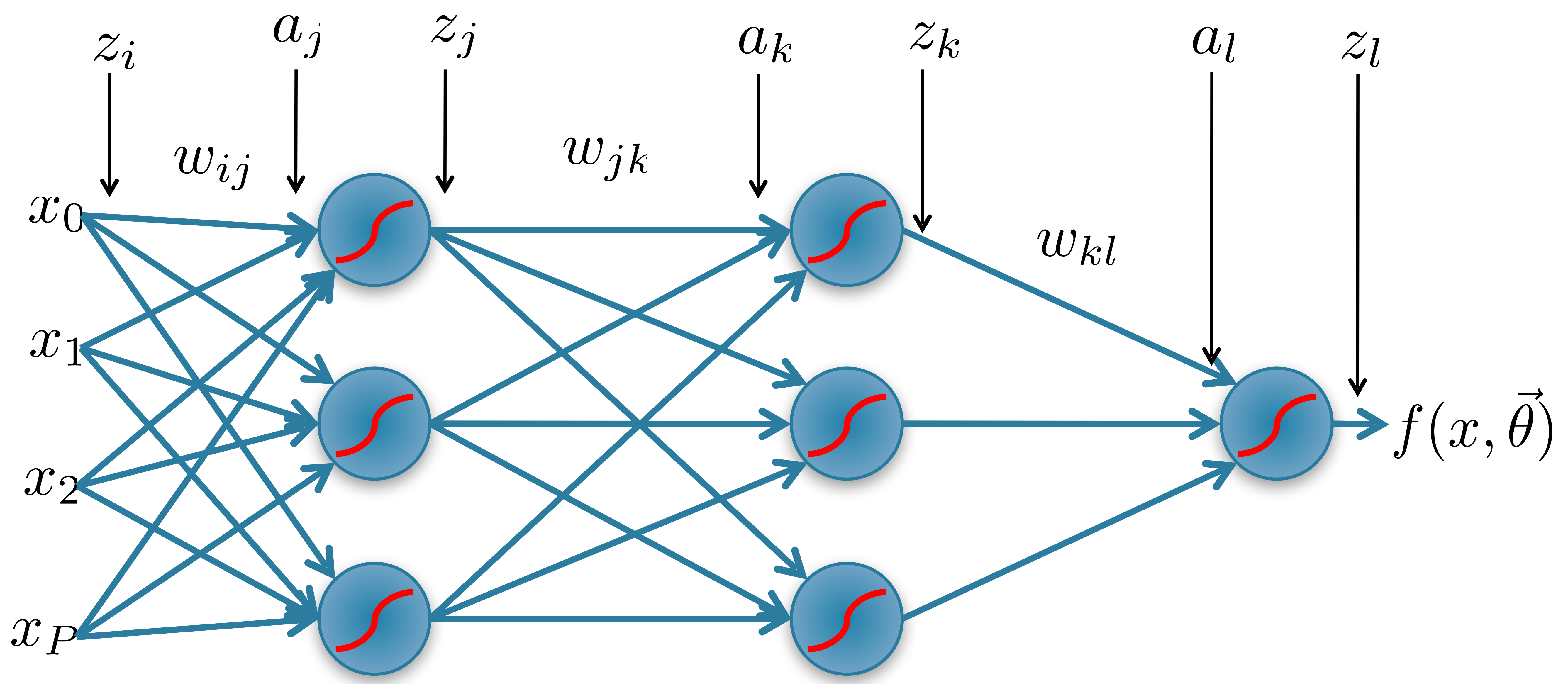
Training: minimize loss

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\quad g\left(\quad g\left(\quad x_{n,i}\right)\right)\right)\right)^2$$

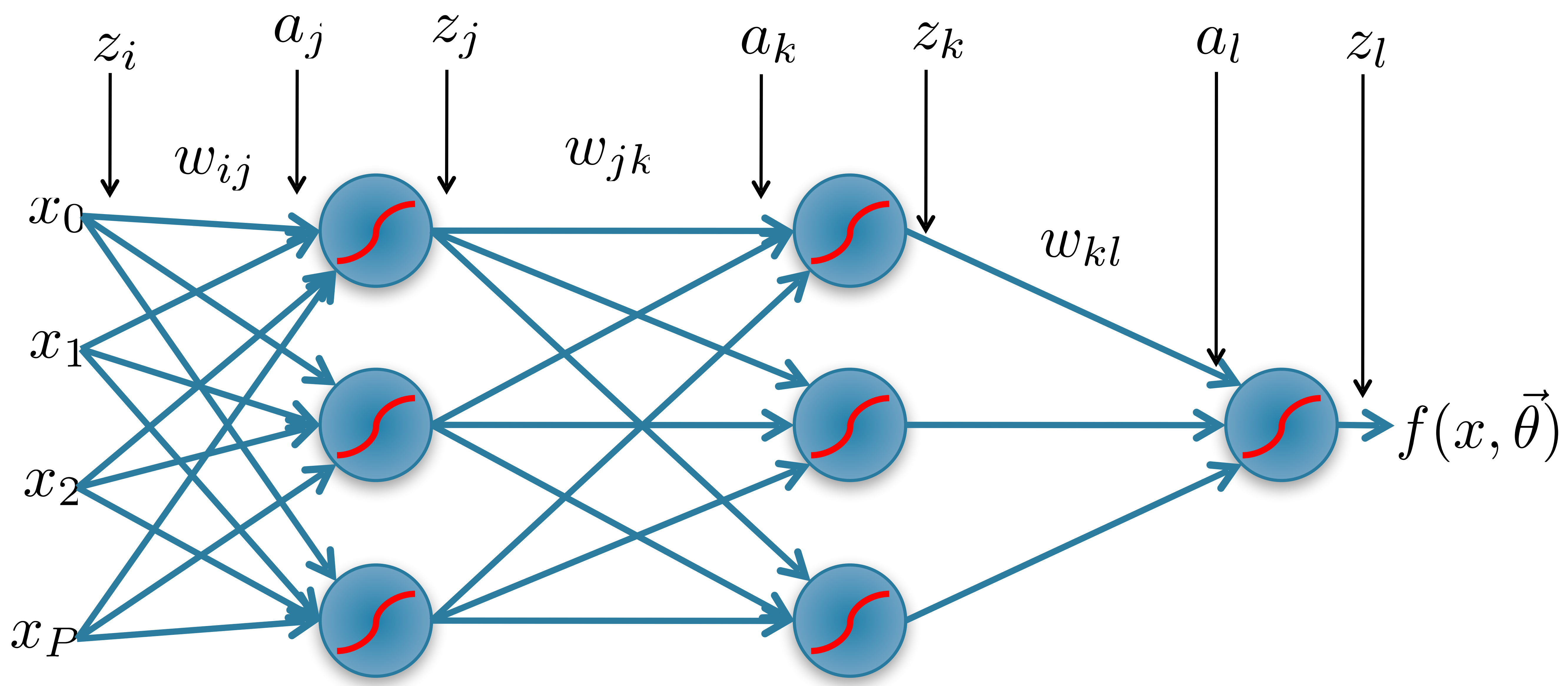
Empirical Risk Function



Training: minimize loss

$$
\begin{aligned}
R(\theta) &= \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n)) \\
&= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2 \\
&= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl}\, g\left(\sum_j w_{jk}\, g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2
\end{aligned}
$$

Empirical Risk Function



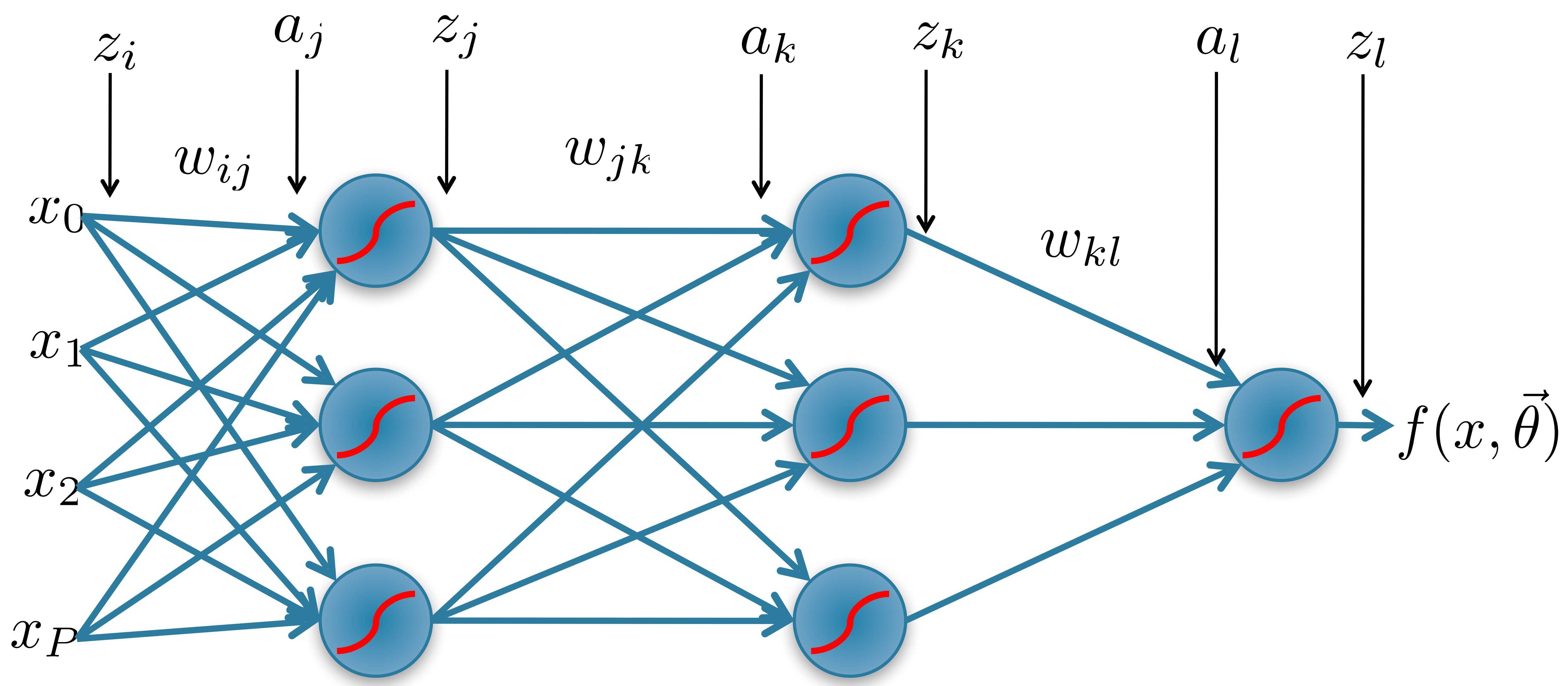
Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

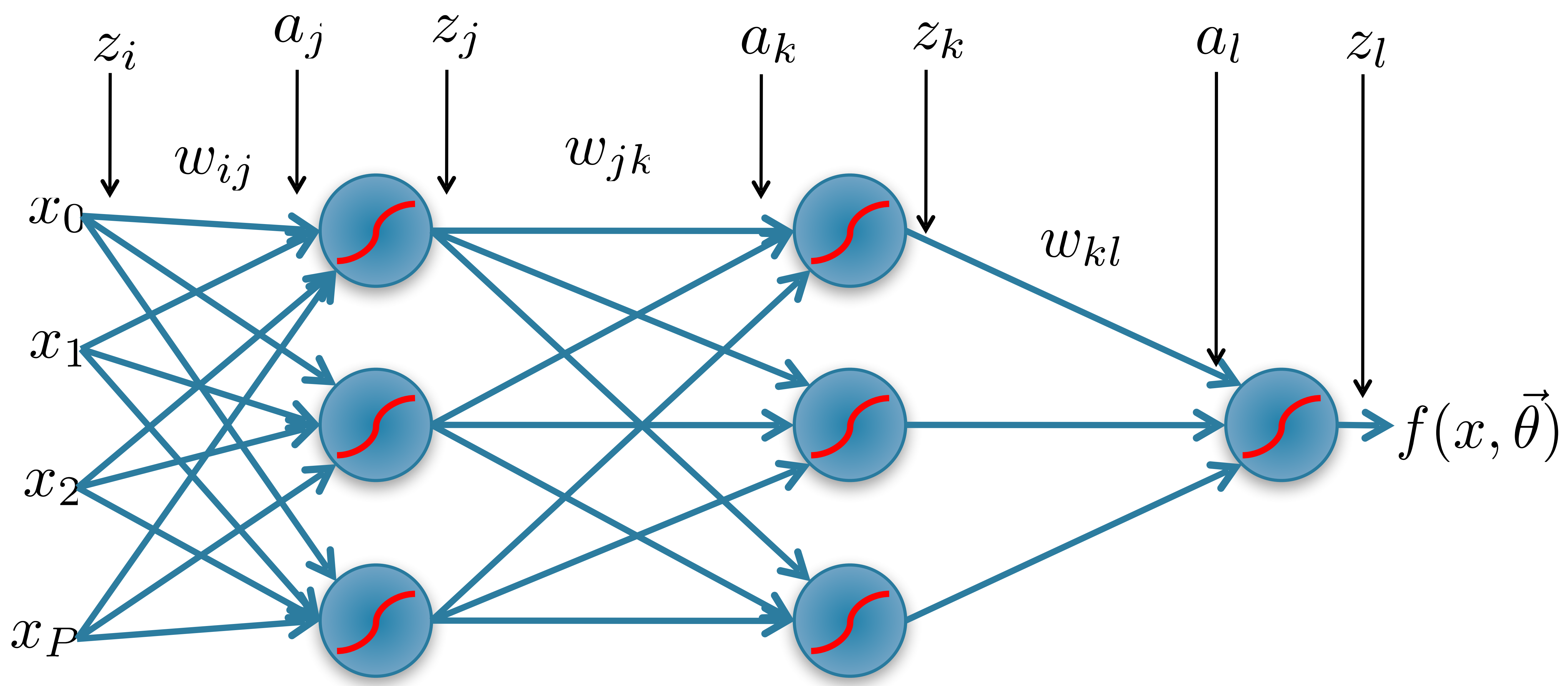
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

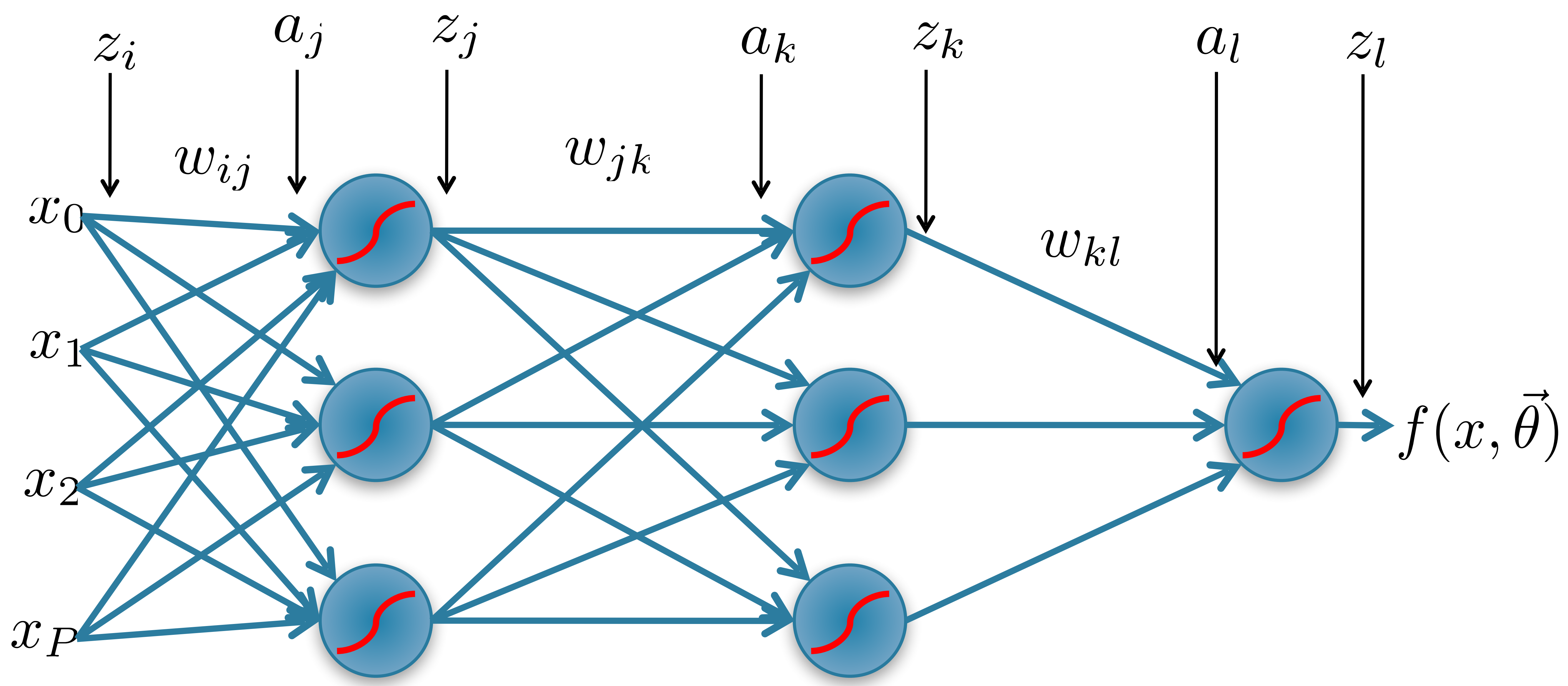
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

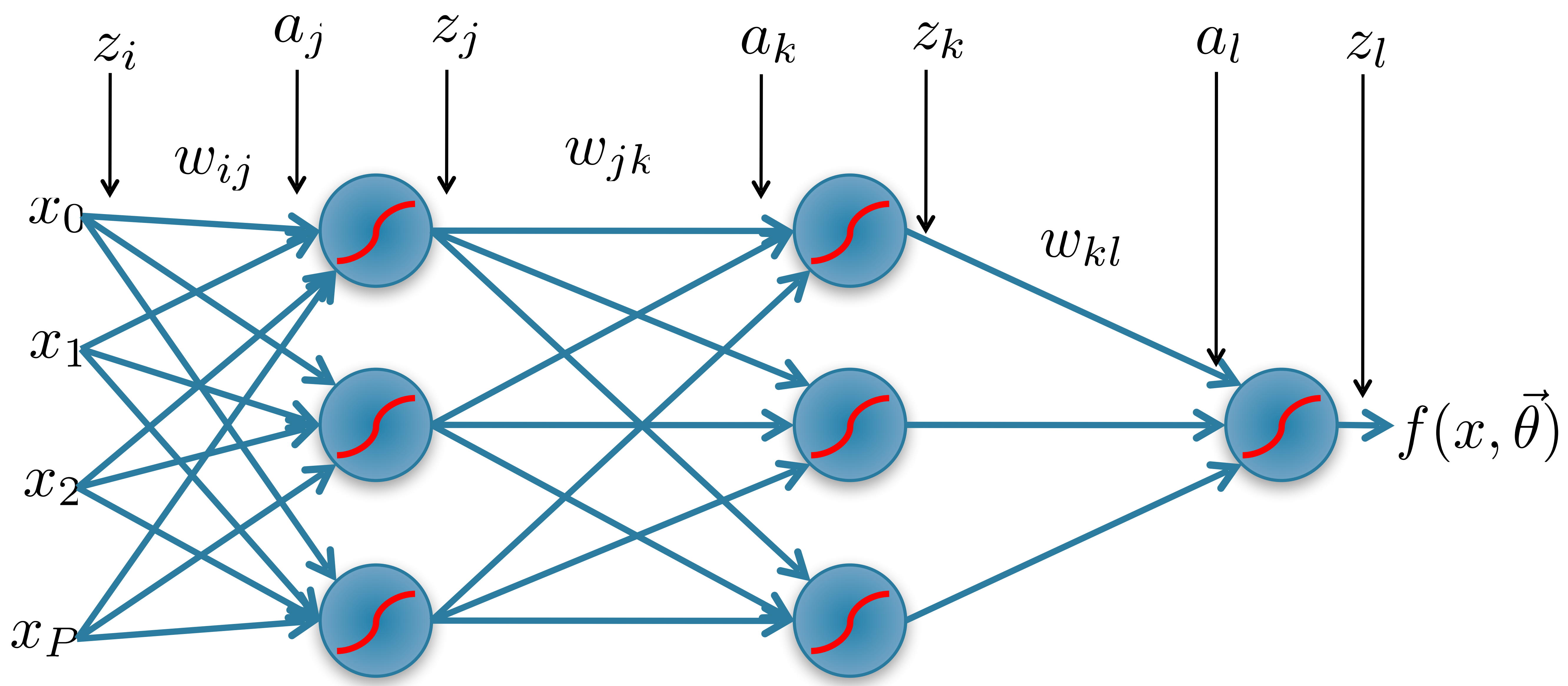
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



Error Backpropagation

Optimize last layer weights w_{kl}

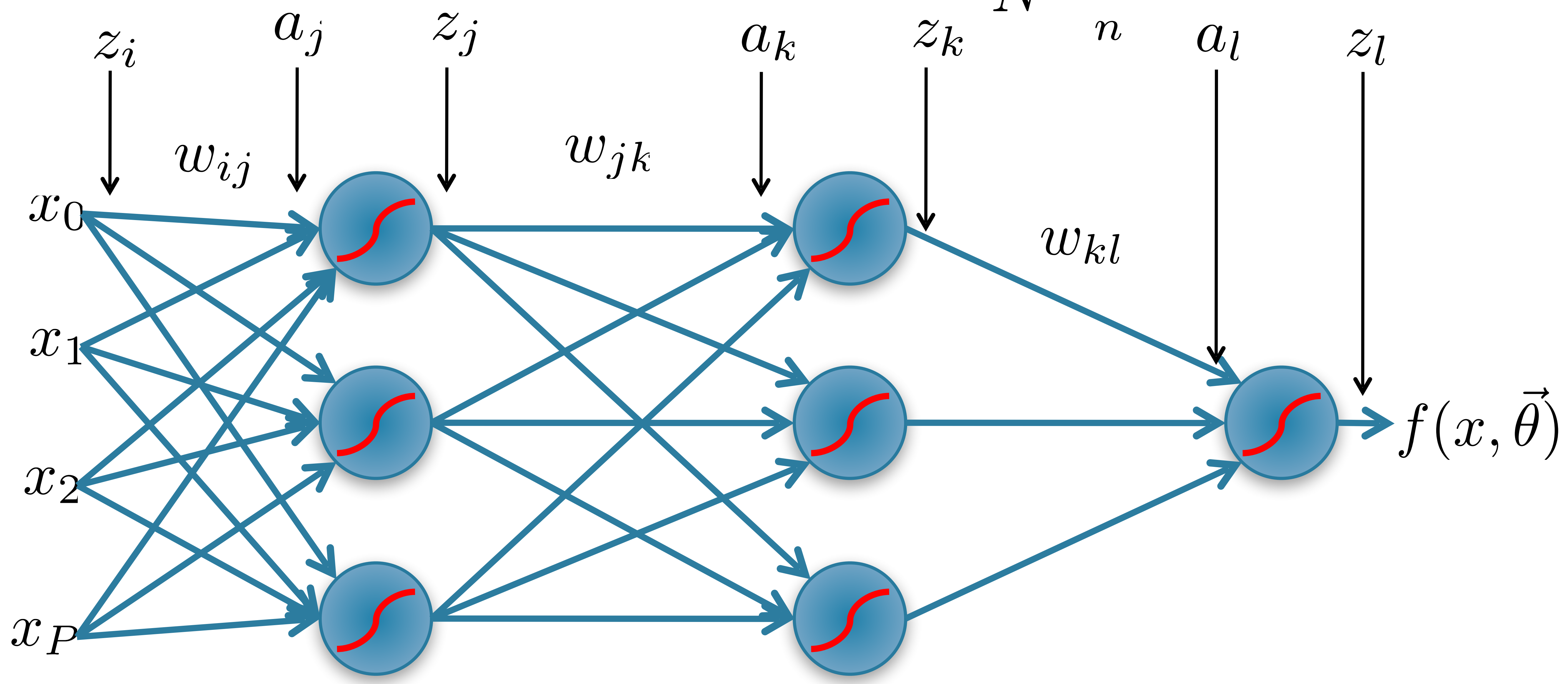
Calculus chain rule

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \boxed{\left[-(y_n - z_{l,n})g'(a_{l,n})\right]} z_{k,n}$$

$$= \frac{1}{N}\sum_n \delta_{l,n} \, z_{k,n}$$



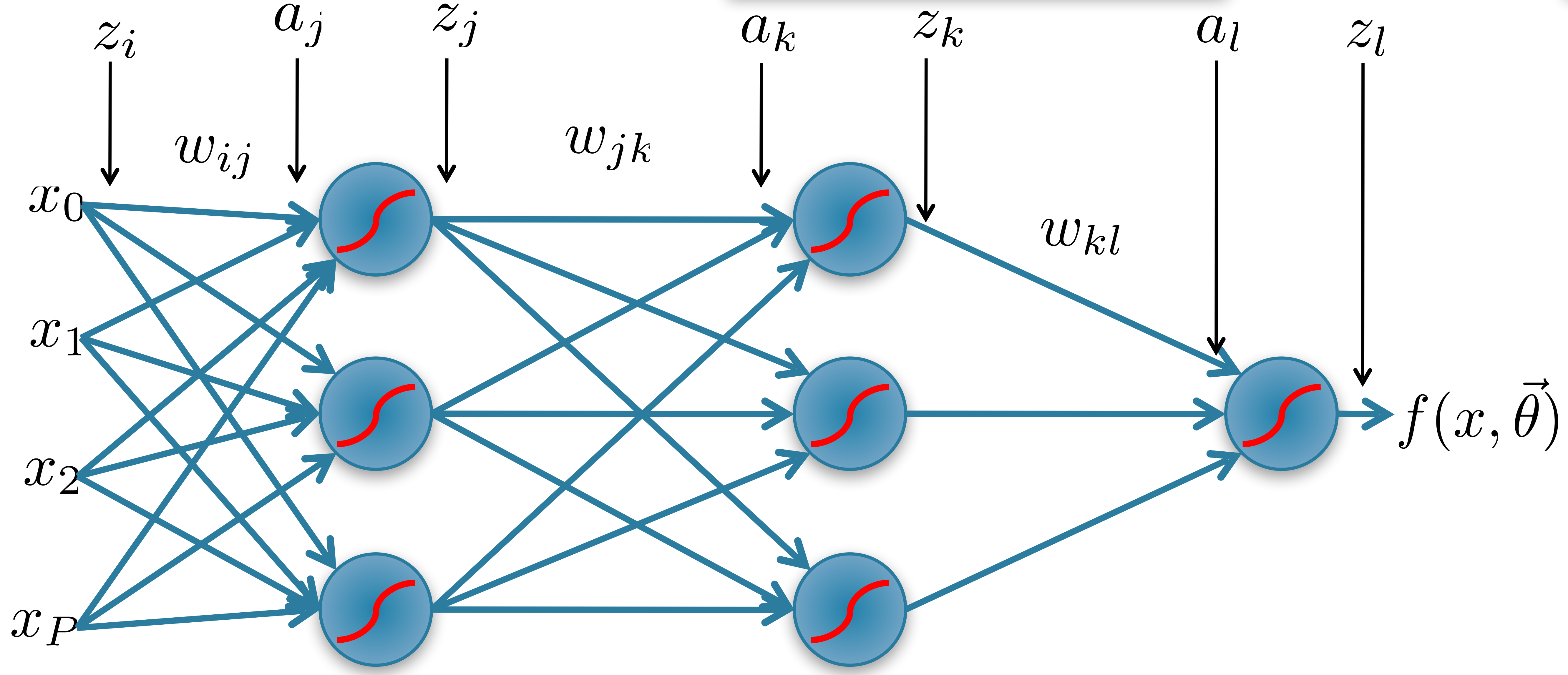
Error Backpropagation

Repeat for all previous layers

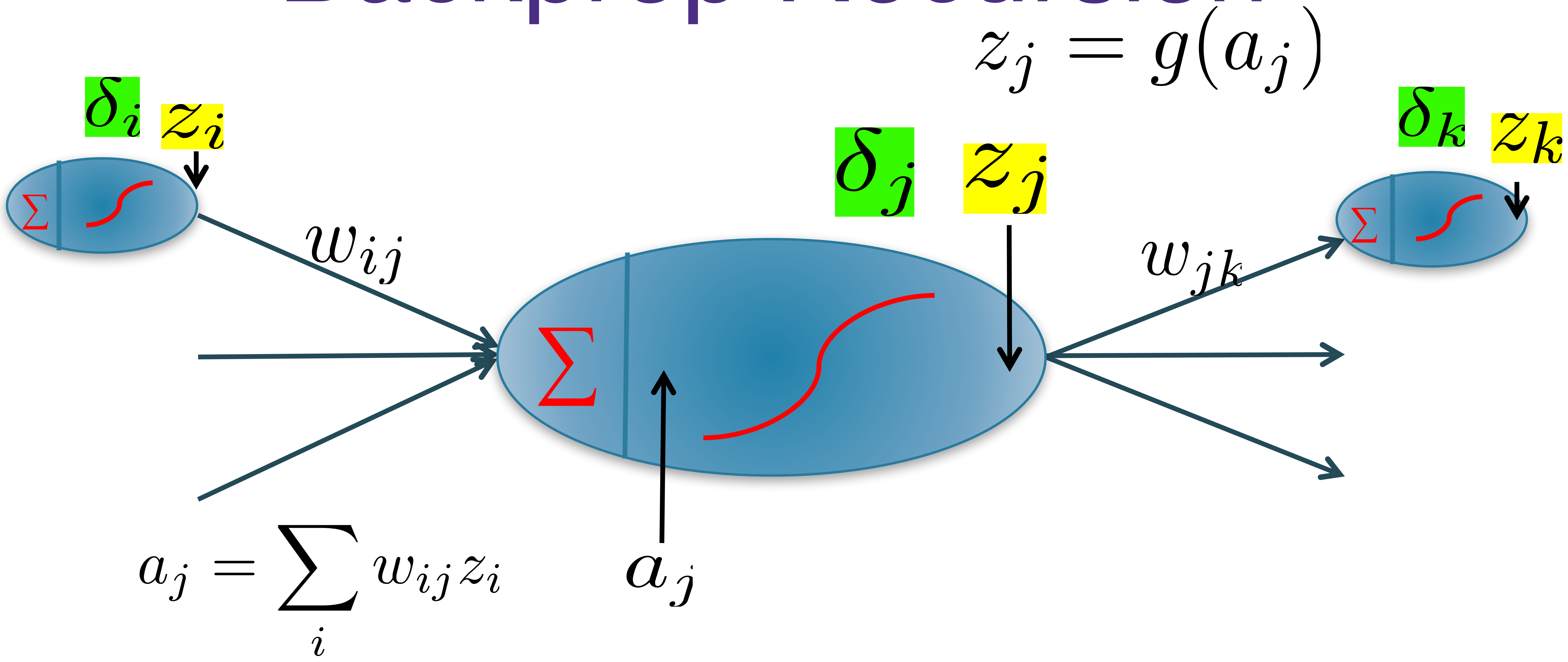
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



Backprop Recursion

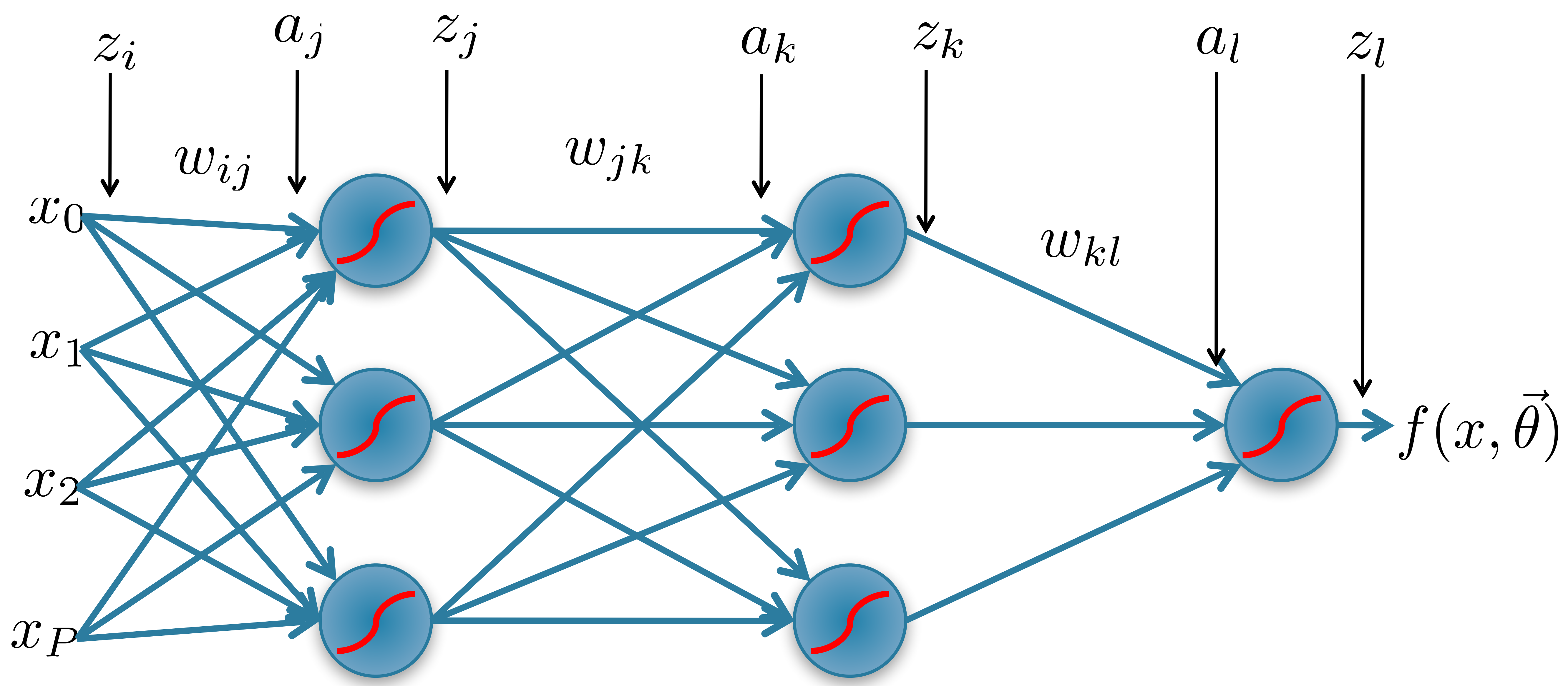


$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\boxed{\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right]} = \frac{1}{N}\sum_n \boxed{\left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right]} z_{i,n} = \frac{1}{N}\sum_n \boxed{\delta_{j,n}}\,\boxed{z_{i,n}}$$

Learning: Gradient Descent

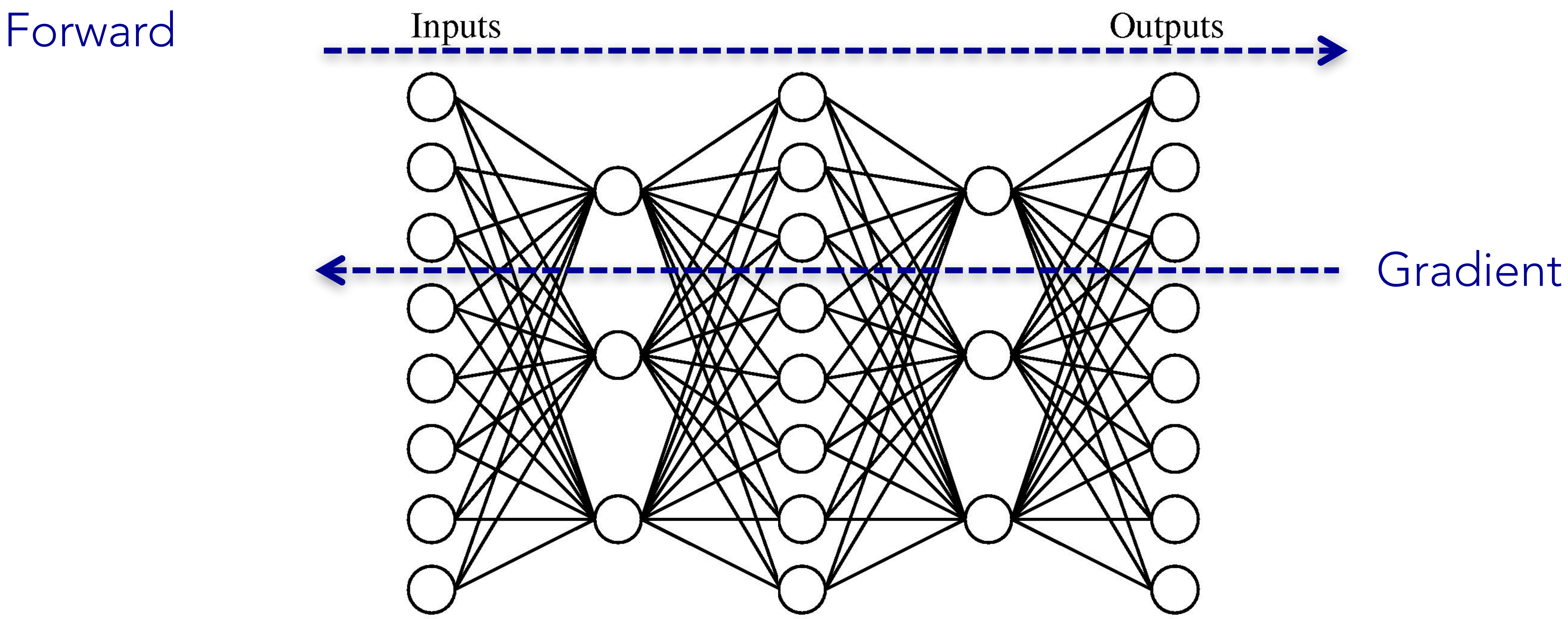
$$
\begin{aligned}
w_{ij}^{t+1} &= w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}} \\
w_{jk}^{t+1} &= w_{jk}^{t} - \eta \frac{\partial R}{w_{kl}} \\
w_{kl}^{t+1} &= w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}}
\end{aligned}
$$



Backpropagation

- Starts with a forward sweep to compute all the intermediate function values
- Through backprop, computes the partial derivatives recursively $\frac{\partial R}{\partial w_{ij}} = \delta_j \, z_i$
- A form of dynamic programming
  - Instead of considering exponentially many paths between a weight w_ij and the final loss (risk), store and reuse intermediate results.
- A type of automatic differentiation. (There are other variants e.g., recursive differentiation only through forward propagation.)

Backpropagation



Backpropagation: general graphs

- Construct graph (two approaches: static vs. **dynamic**)
- Forward:
  - Loop over nodes in the graph's topological order
    - Computing value of nodes given inputs
      - Store any values needed for gradient computation
- Backward:
  - Loop over nodes in the graph's *reverse* topological order
  - Compute derivative of output w/r/t all inputs of a node

Backpropagation

- Major libraries like TensorFlow and PyTorch have auto-diff built-in
  - Define (now, dynamically) computation graph, get backprop “automatically”

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Backprop the loss!

Update the parameters

[Yes, you should understand backdrop!]()

Cross Entropy Loss (aka log loss, logistic loss)

- Cross Entropy

$$H(p,q) = -\sum_y p(y)\, \log\, q(y)$$

Predicted prob

True prob

In classification:

$$= -\log \hat{y}(y_{\text{true}})$$

- Use Cross Entropy for models that should have more probabilistic flavor (e.g., language models), incl classifiers

- Use Mean Squared Error loss for regression-like models

$$\text{MSE} = \frac{1}{2}(y - f(x))^2$$

- Fancier applications / tasks —> specialized losses

Other backprop resources

- This is a lot of complex material. No one understands it the first time!
  - Read lots of different expositions, triangulate, find what works for you
- From course website:
  - DL book ch 6.5: https://www.deeplearningbook.org/contents/mlp.html
  - CS231n notes: http://cs231n.github.io/optimization-2/
  - CS231n notes on vector derivatives: http://cs231n.stanford.edu/vecDerivs.pdf
- The matrix calculus you need: https://explained.ai/matrix-calculus/
- Calculus on computation graphs: https://colah.github.io/posts/2015-08-Backprop/
- Minimal (not fully general!) example in pure numpy: http://cs231n.github.io/neural-networks-case-study/