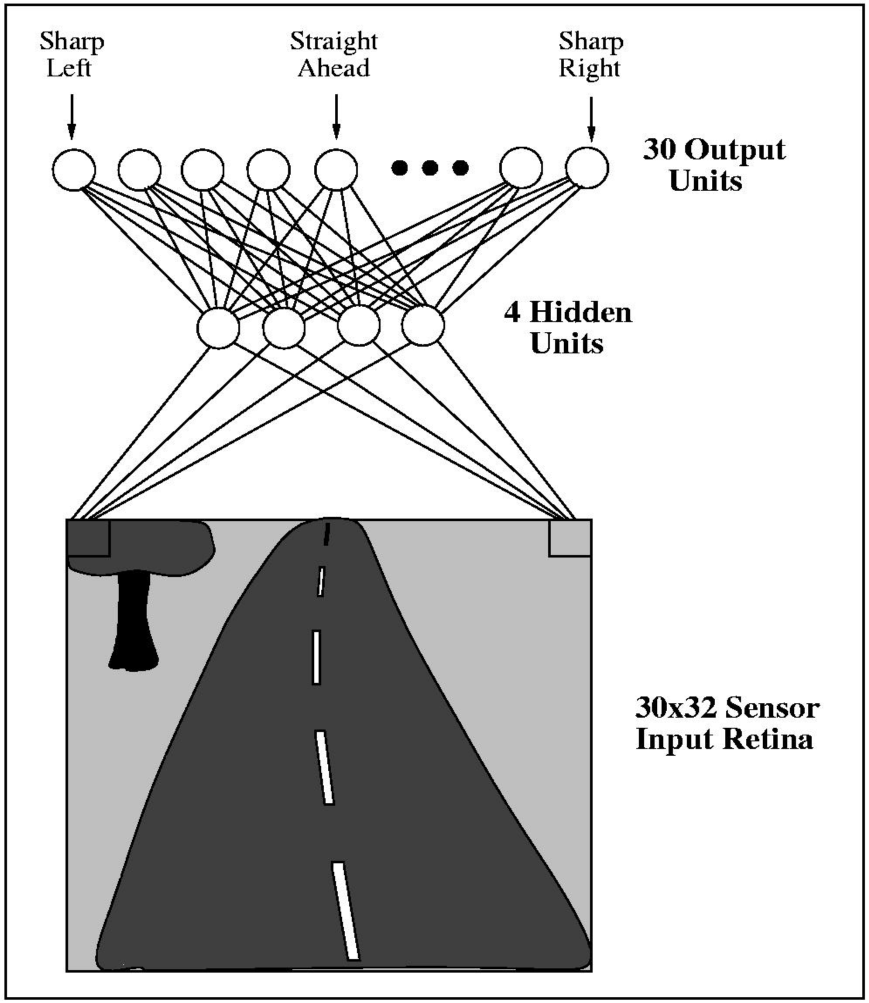
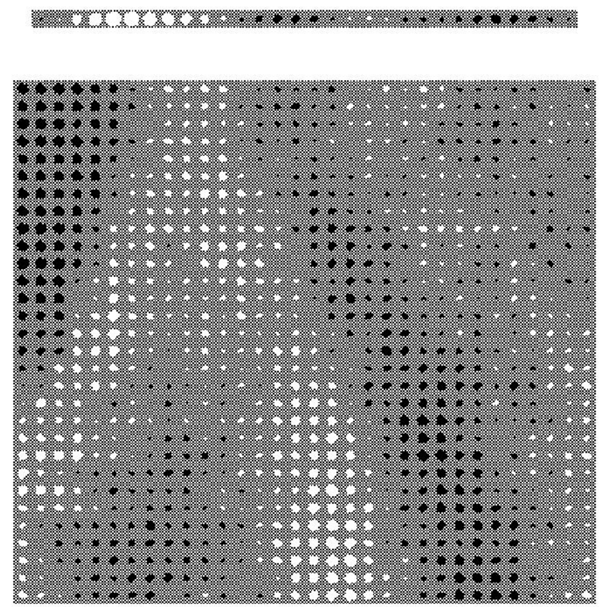
CSEP 517: NLP
Introduction to Neural Nets
Autumn 2018

Luke Zettlemoyer
University of Washington

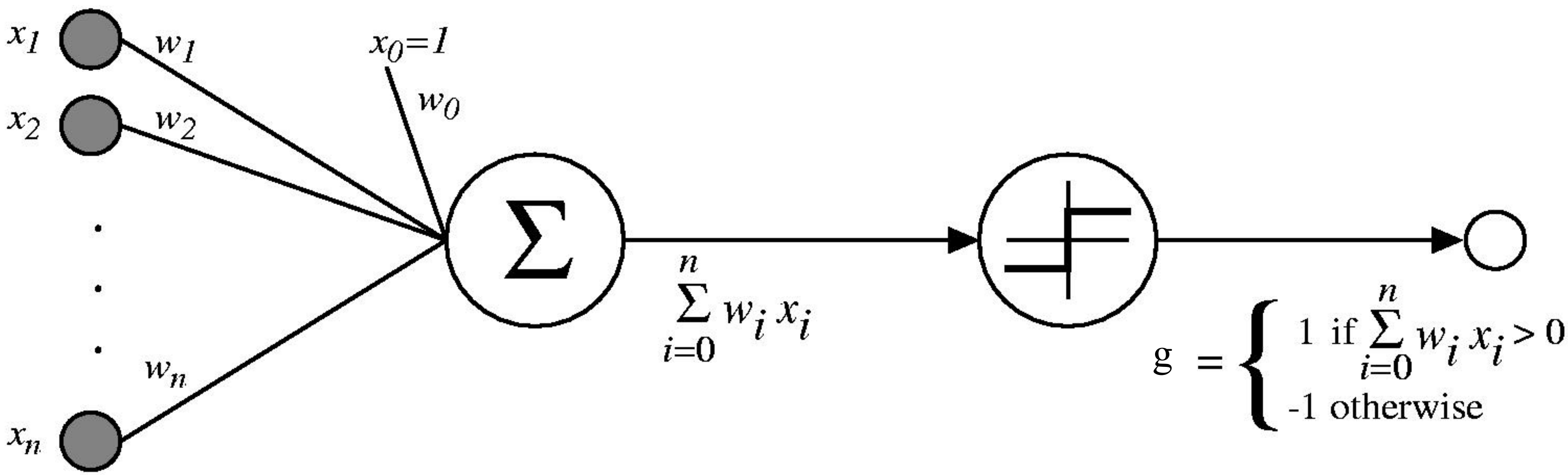
[Many slides from Yejin Choi, Carlos Guestrin]

Next several slides are from Carlos Guestrin, Luke Zettlemoyer


Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



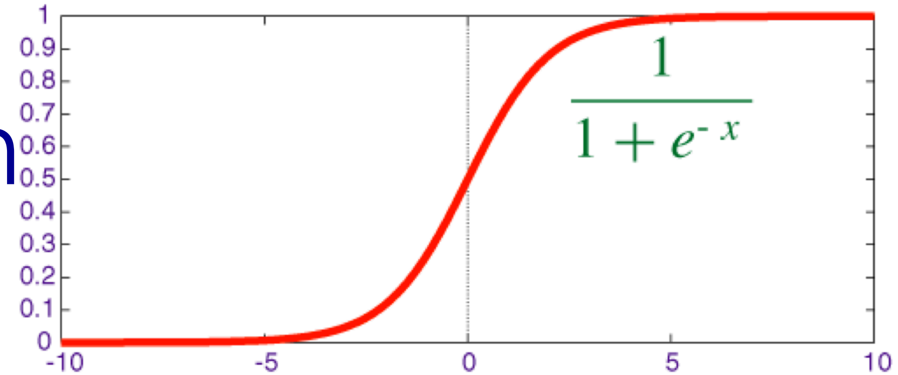
Perceptron as a Neural Network



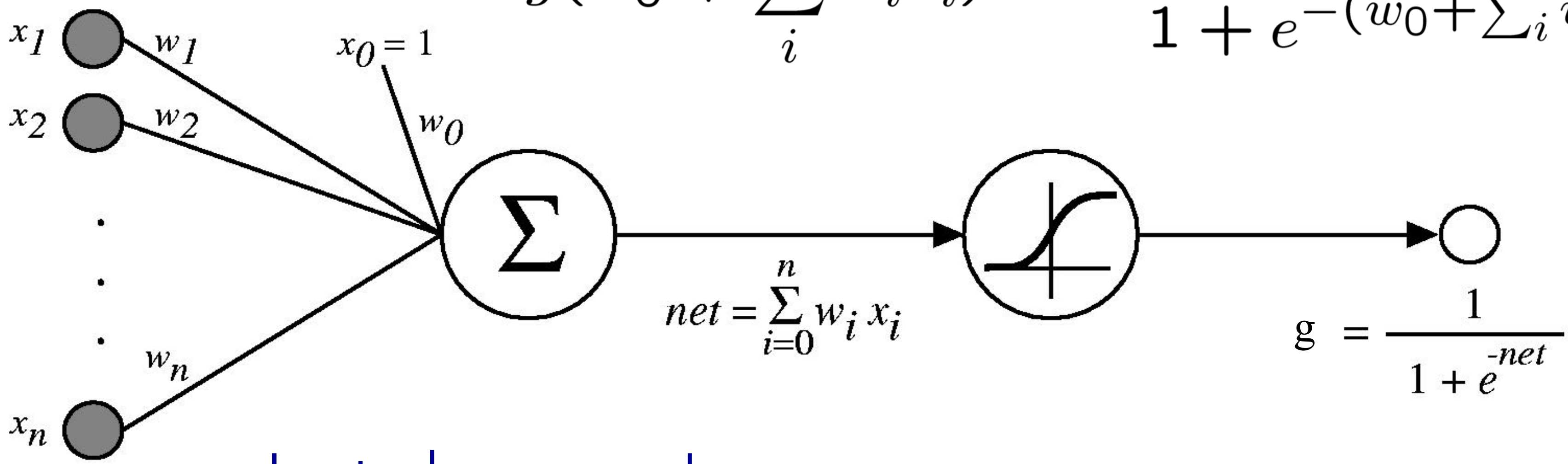
This is one neuron:

- Input edges x_1 ... x_n, along with basis
- The sum is represented graphically
- Sum passed through an activation function g

Sigmoid Neuron



$$g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$



Just change g!

- Why would we want to do this?
- Notice new output range [0,1]. What was it before?
- Look familiar?

Optimizing a neuron

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

We train to minimize sum-squared error

$$\ell(W) \;=\; \frac{1}{2}\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

$$\frac{\partial l}{\partial w_i} = -\sum_j [y_j - g(w_0 + \sum_i w_i x_i^j)] \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j)$$

$$\frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j) = x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

$$\boxed{\frac{\partial \ell(W)}{\partial w_i} \;=\; -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]\; x_i^j \; g'(w_0 + \sum_i w_i x_i^j)}$$

Solution just depends on g': derivative of activation function!

Sigmoid units: have to differentiate g

$$\frac{\partial \ell(W)}{\partial w_i} = -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] \; x_i^j \; g'(w_0 + \sum_i w_i x_i^j)$$

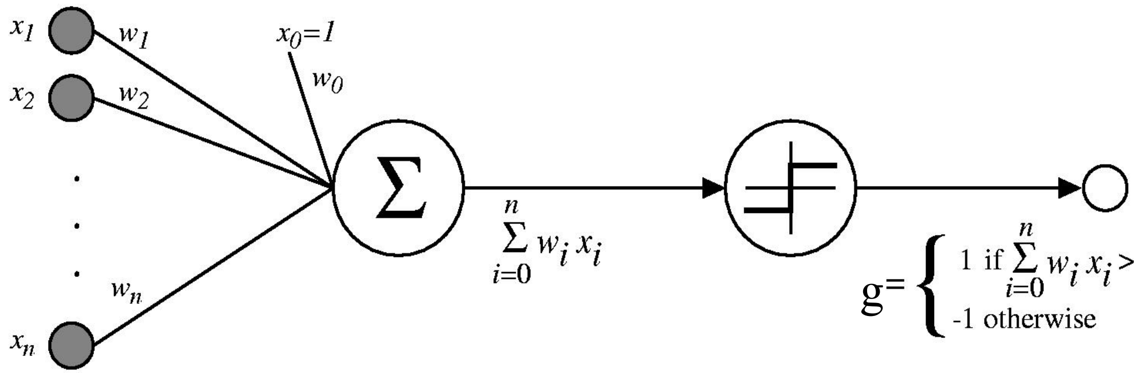
$$g(x) = \frac{1}{1+e^{-x}} \qquad g'(x) = g(x)(1-g(x))$$

$$\boxed{w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j}$$

$$\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)] g^j (1 - g^j)$$

$$g^j = g(w_0 + \sum_i w_i x_i^j)$$

Perceptron, linear classification, Boolean functions: $x_i \in \{0,1\}$



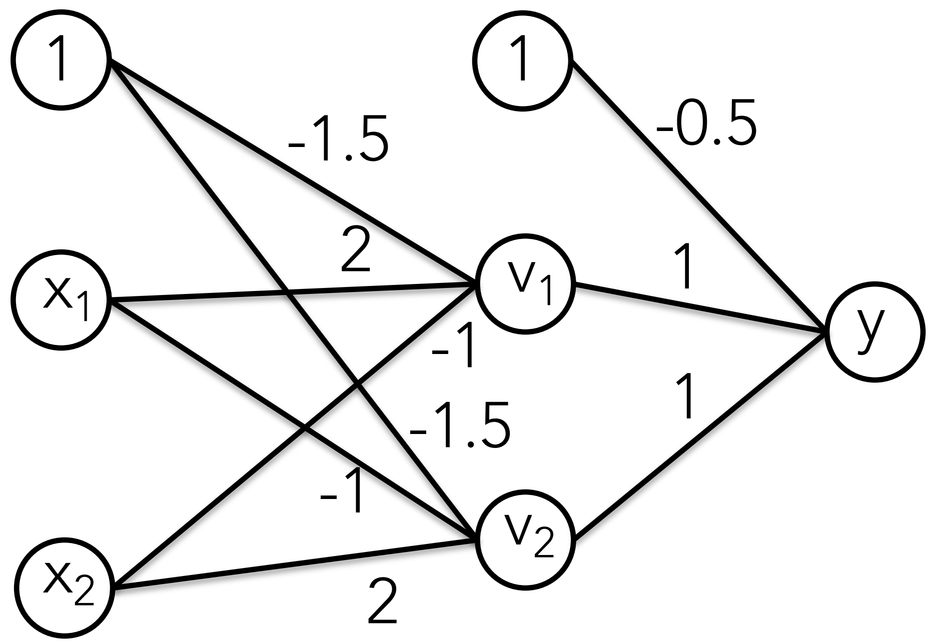
- Can learn $x_1 \vee x_2$?
  - $-0.5 + x_1 + x_2$
- Can learn $x_1 \wedge x_2$?
  - $-1.5 + x_1 + x_2$
- Can learn any conjunction or disjunction?
  - $0.5 + x_1 + \ldots + x_n$
  - $(-n+0.5) + x_1 + \ldots + x_n$
- Can learn majority?
  - $(-0.5*n) + x_1 + \ldots + x_n$
- What are we missing? The dreaded XOR!, etc.

Going beyond linear classification

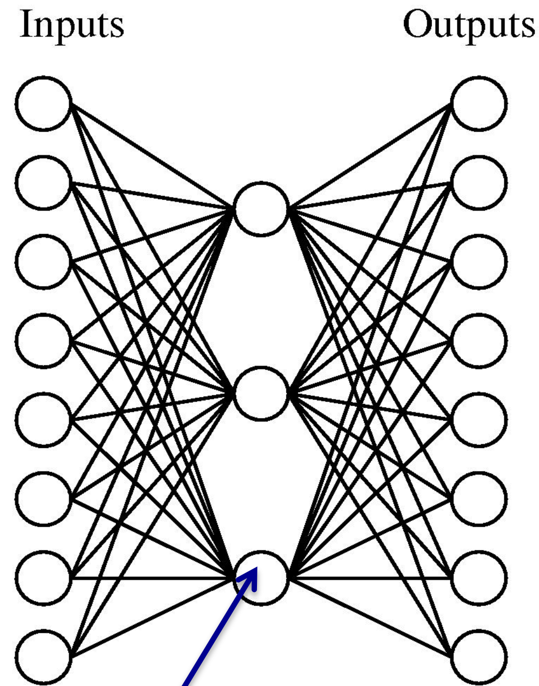
Solving the XOR problem

$y = x_1 \text{ XOR } x_2 = (x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_1)$

$$v_1 = (x_1 \wedge \neg x_2)$$
$$= -1.5 + 2x_1 - x_2$$
$$v_2 = (x_2 \wedge \neg x_1)$$
$$= -1.5 + 2x_2 - x_1$$
$$y = v_1 \vee v_2$$
$$= -0.5 + v_1 + v_2$$



Hidden layer



- Single unit:

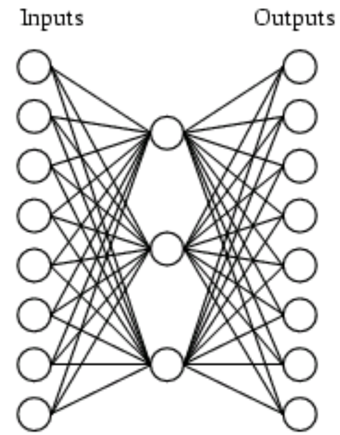
$$out(\mathbf{x}) \;=\; g(w_0 + \sum_i w_i x_i)$$

- 1-hidden layer:

$$out(\mathbf{x}) \;=\; g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$

- No longer convex function!

Example data for NN with hidden layer



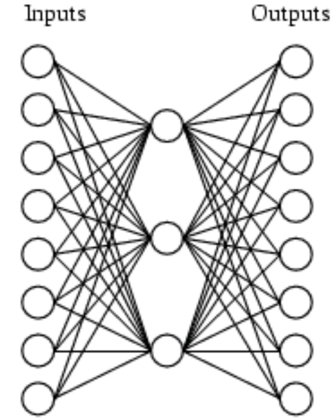
A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learned weights for hidden layer

A network:



Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Why "representation learning"?

- MaxEnt (multinomial logistic regression):

$$y = \mathrm{softmax}(w \cdot \underline{f(x,y)})$$

You design the feature vector

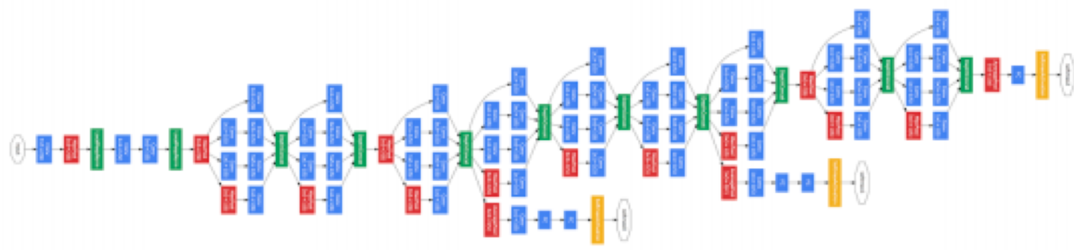
- NNs:

$$y = \mathrm{softmax}(w \cdot \underline{\sigma(Ux)})$$

$$y = \mathrm{softmax}(w \cdot \underline{\sigma(U^{(n)}(...\sigma(U^{(2)}\sigma(U^{(1)}x))))})$$

Feature representations are "learned" through hidden layers

Very deep models in computer vision



[1]Inception 5 (GoogLeNet)

Inception 7a

[1]Going Deeper with Convolutions, [C. Szegedy et al, CVPR 2015]

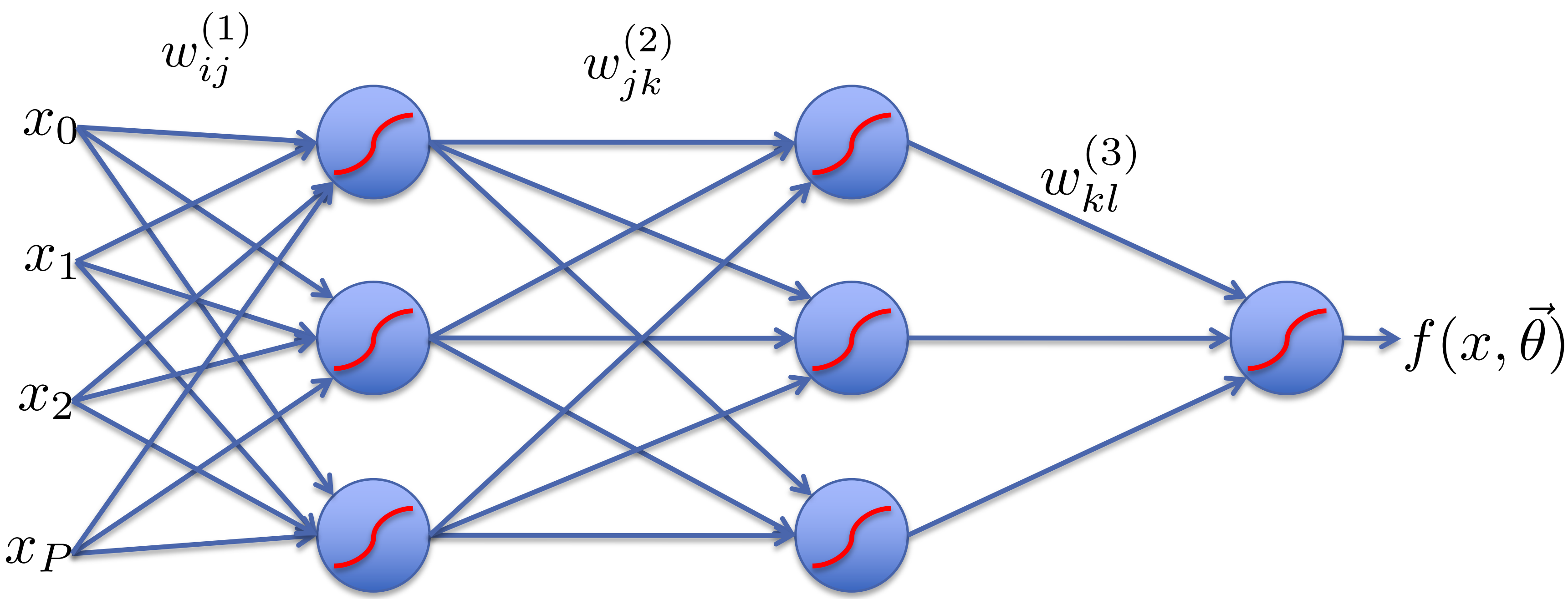
LEARNING: BACKPROPAGATION

Next 10 slides on back propagation are adapted from Andrew Rosenberg

Error Backpropagation

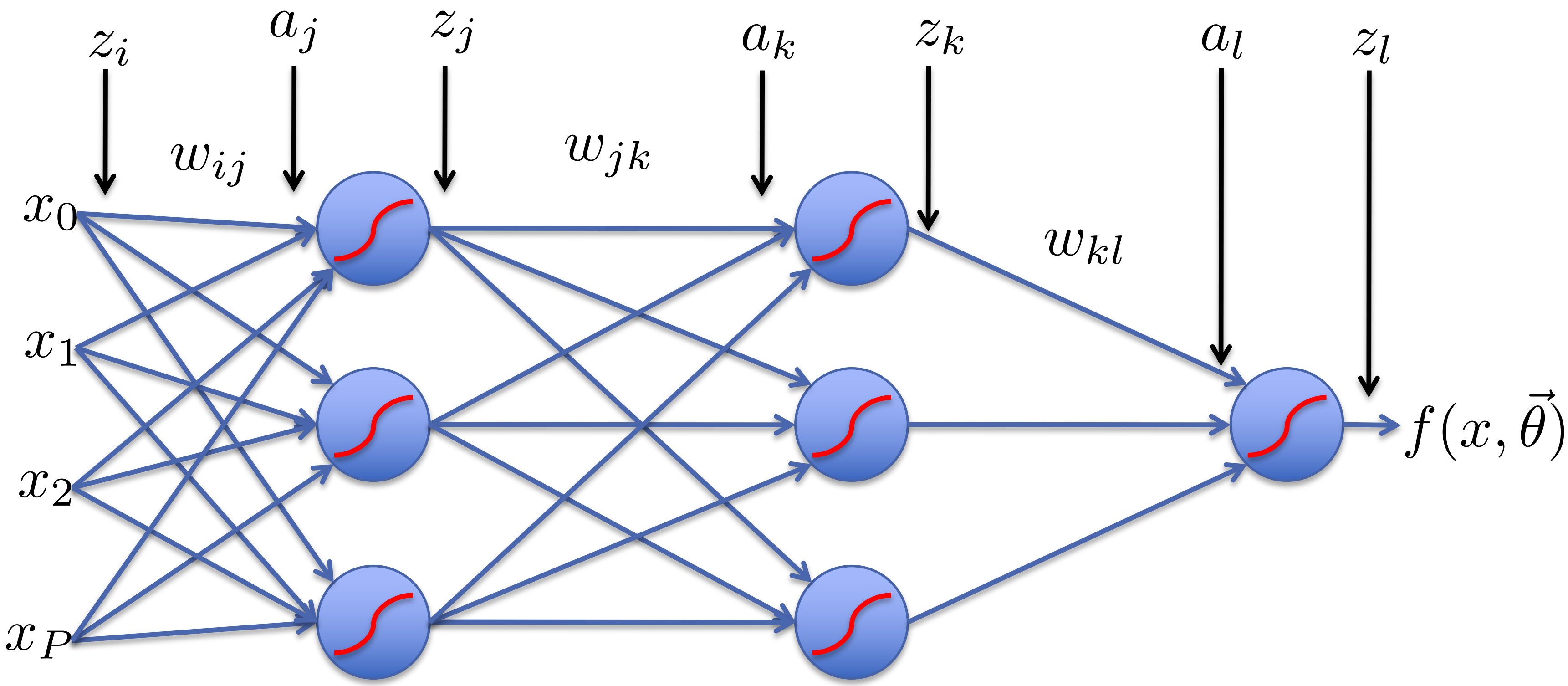
- Model parameters: $\vec{\theta} = \{w_{ij}^{(1)}, w_{jk}^{(2)}, w_{kl}^{(3)}\}$

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



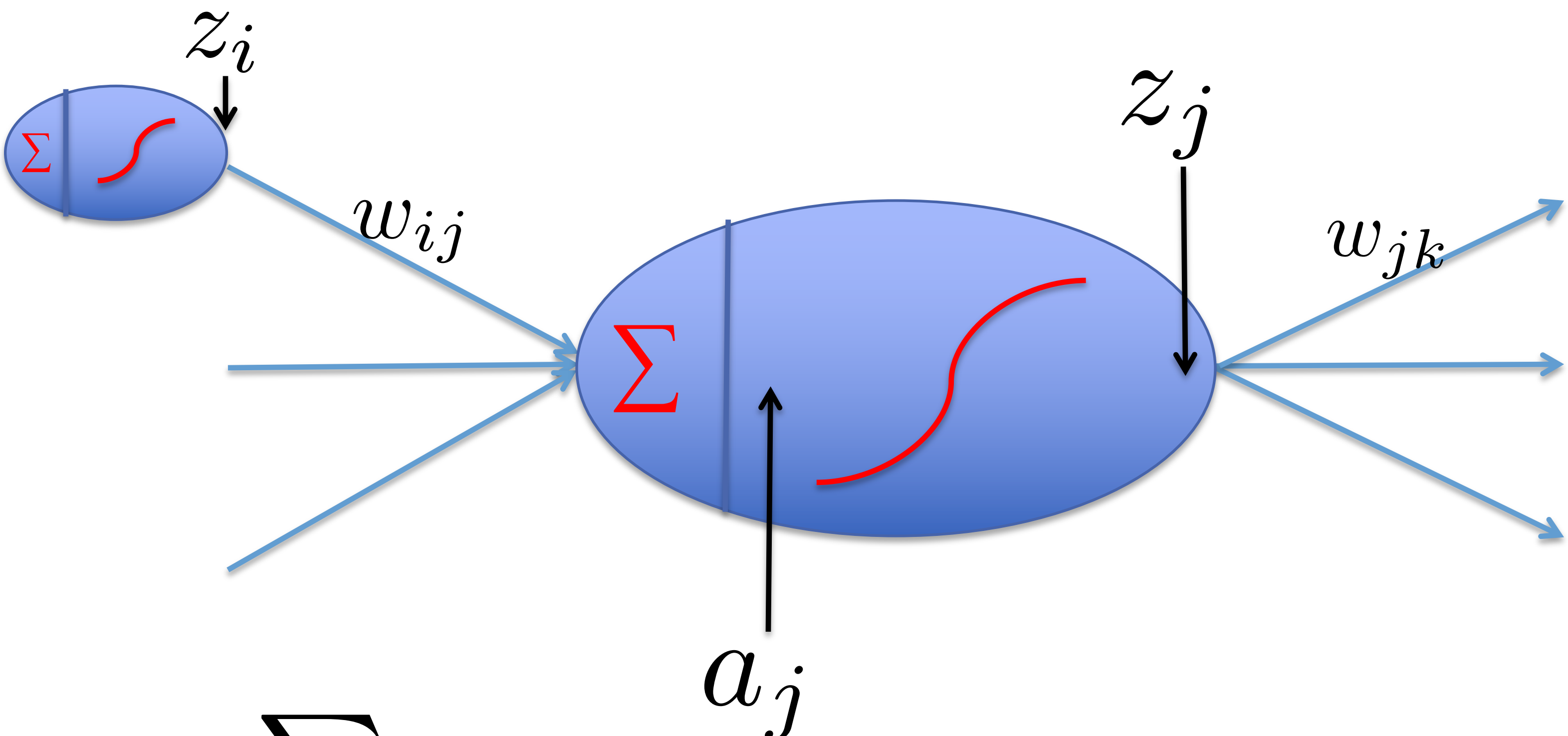
Error Backpropagation

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



Error Backpropagation

$$z_j = g(a_j)$$

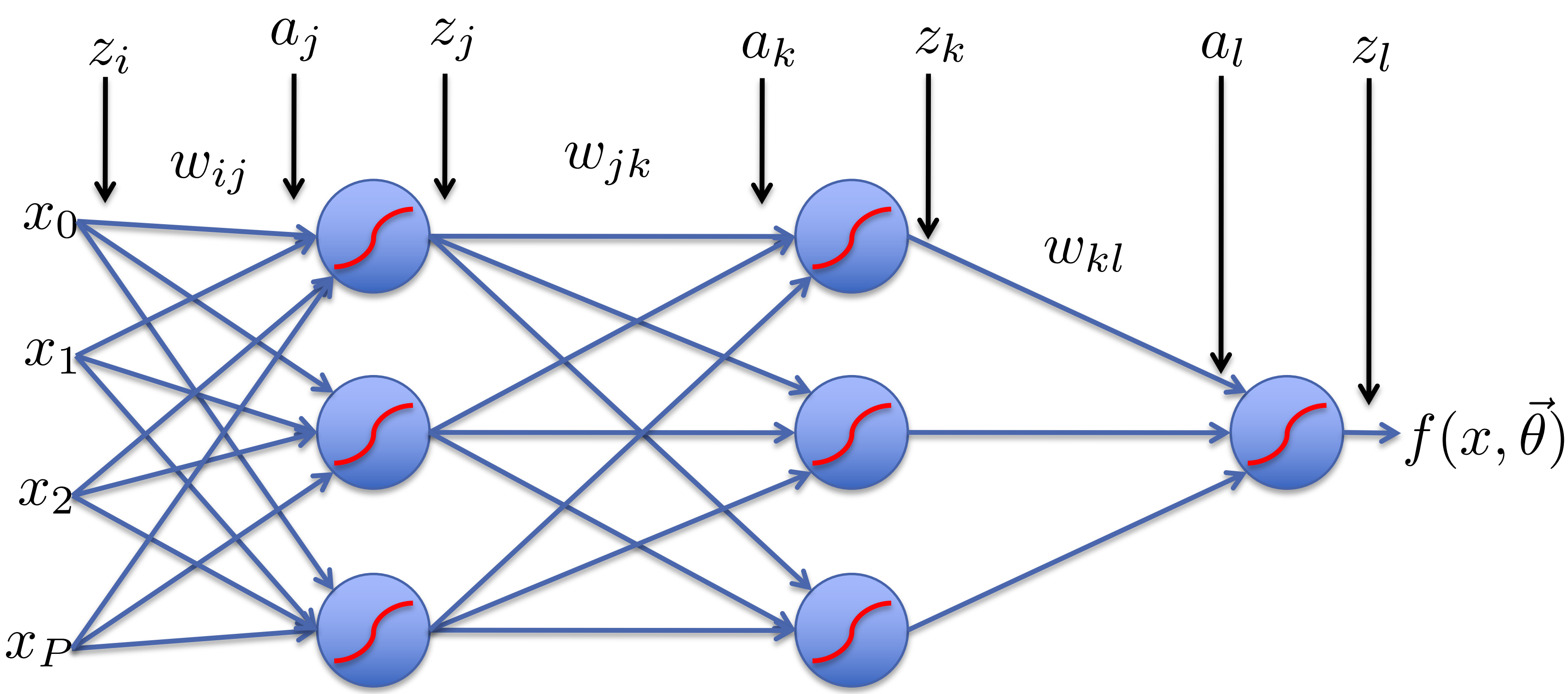


$$a_j = \sum_i w_{ij} z_i$$

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

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

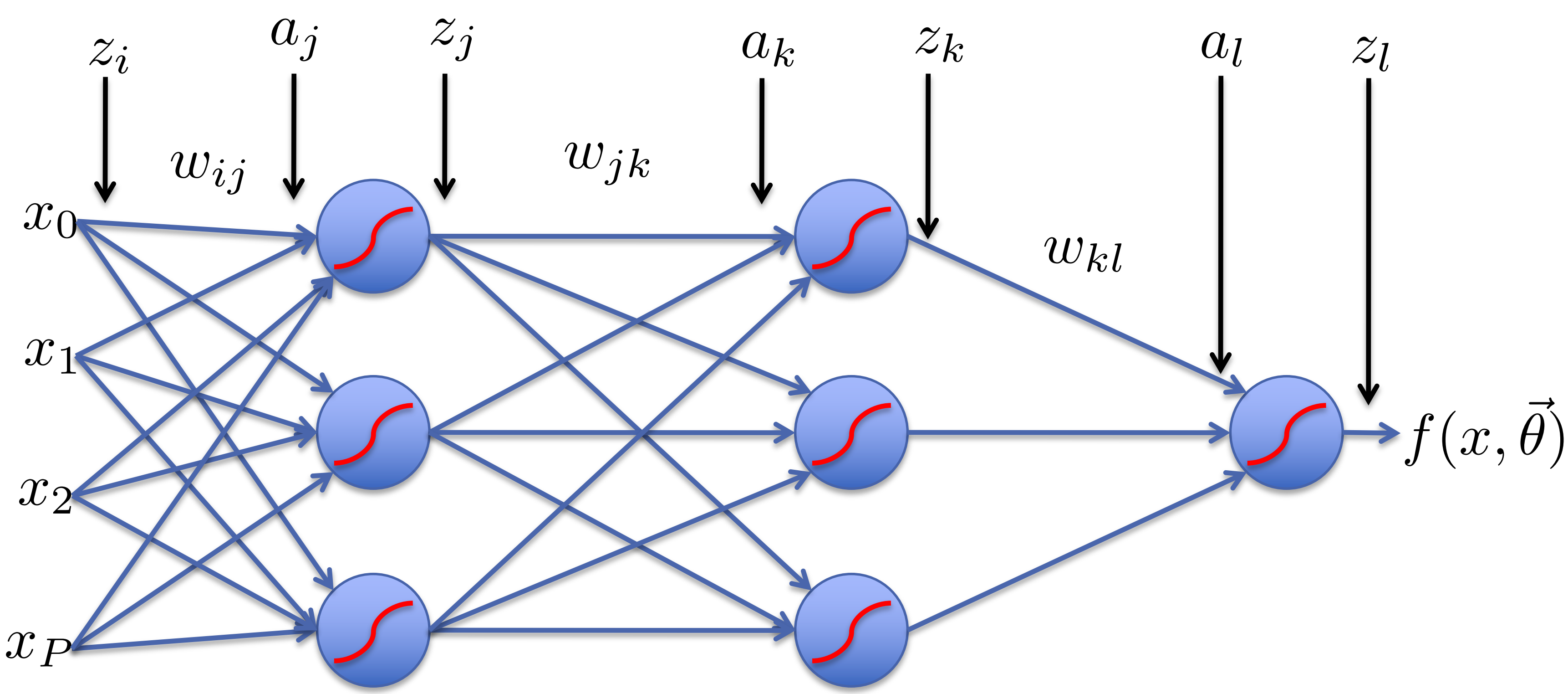
$$z_j = g(a_j) \qquad z_k = \boxed{} \qquad z_l = \boxed{}$$



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

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

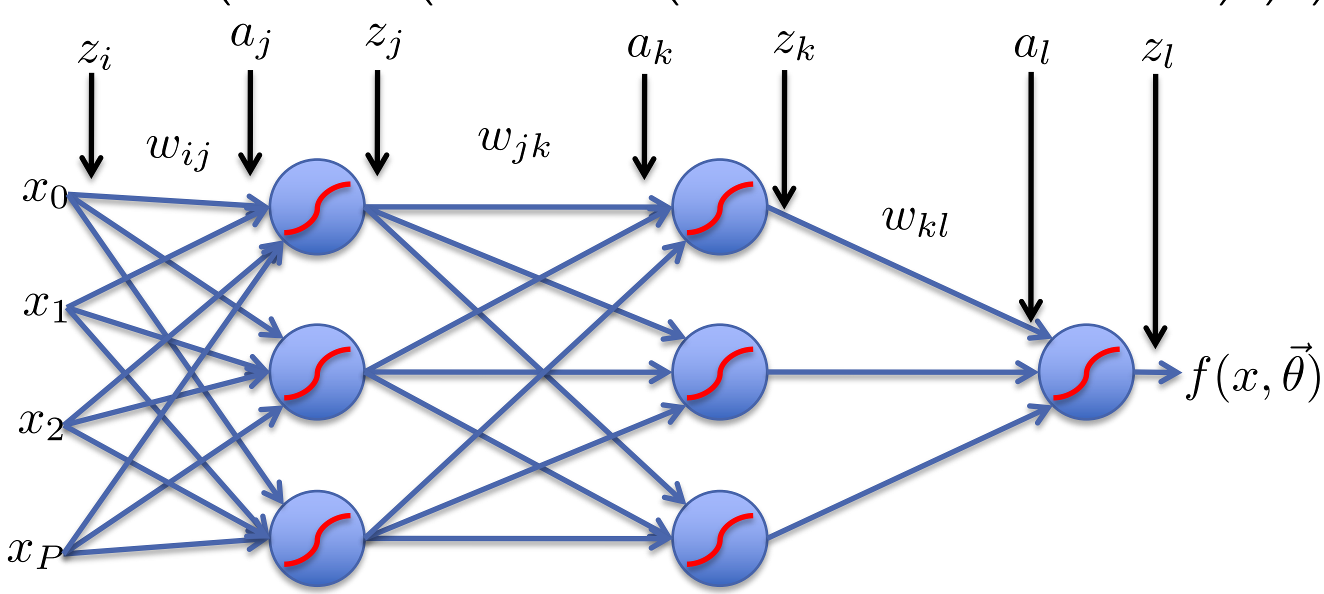
$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



Training: minimize loss

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

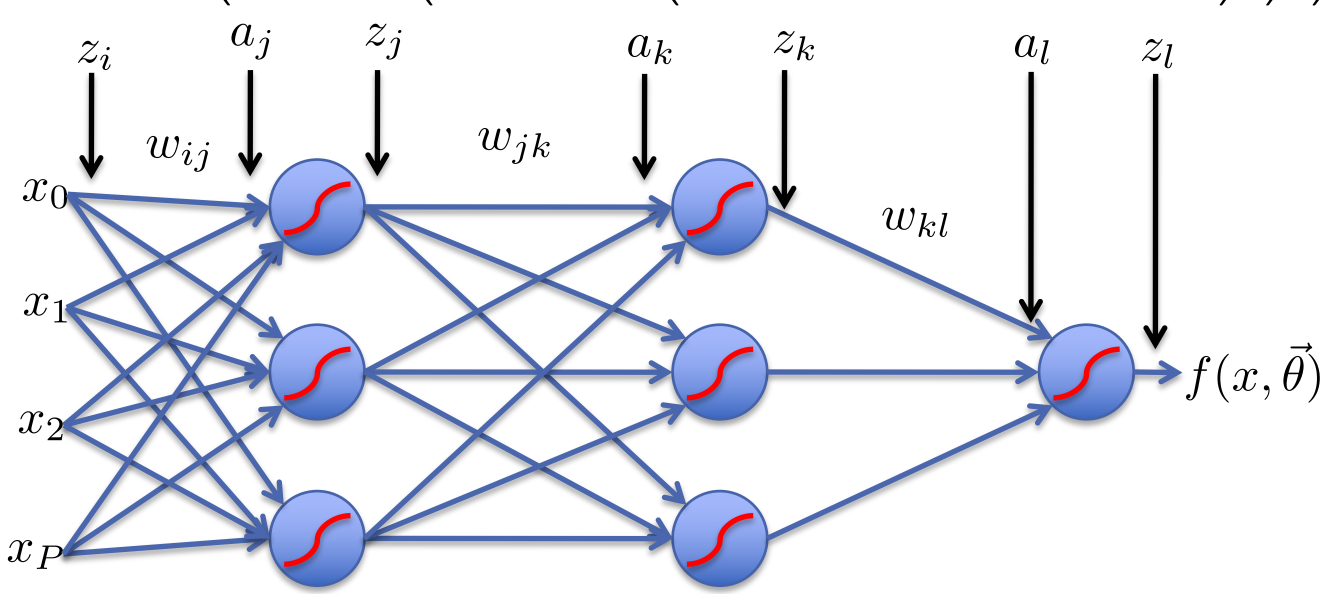
Empirical Risk Function



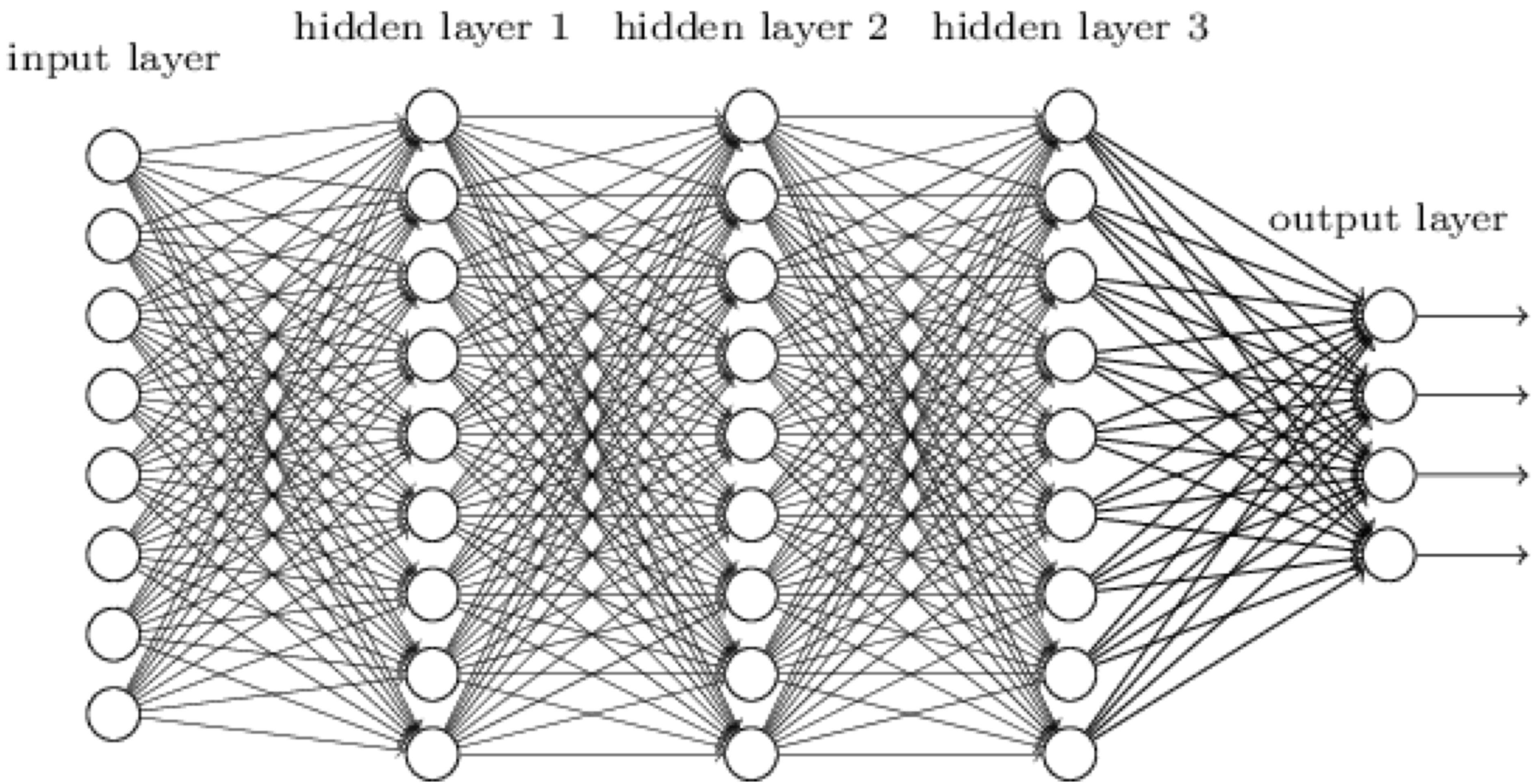
Training: minimize loss

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

Empirical Risk Function



Taking Partial Derivatives…



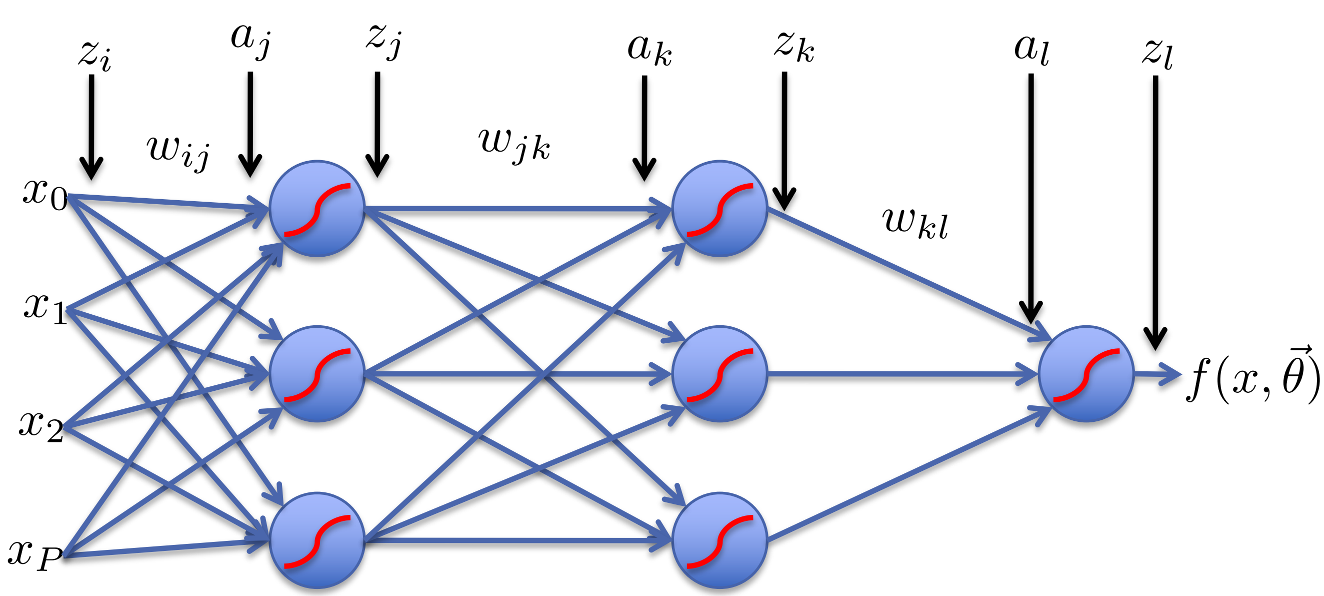
Error Backpropagation

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

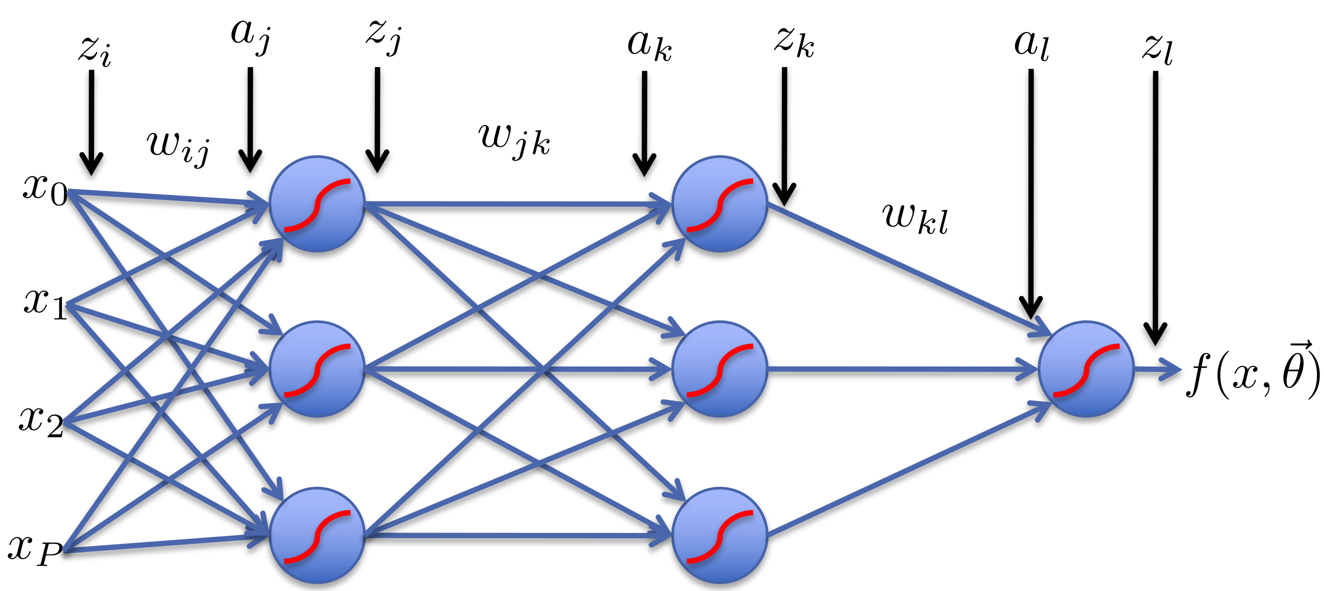
Optimize last layer weights w_{kl}

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

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

Calculus chain rule

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



Error Backpropagation

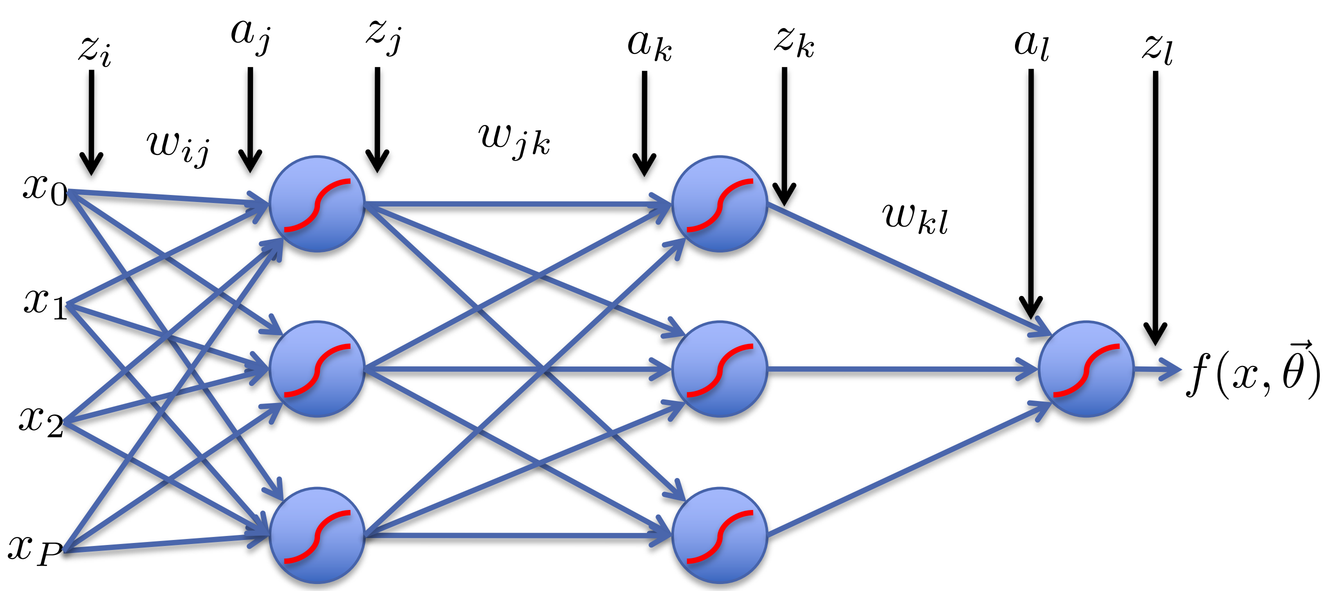
Optimize last layer weights w_{kl}

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

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

Calculus chain rule

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



Error Backpropagation

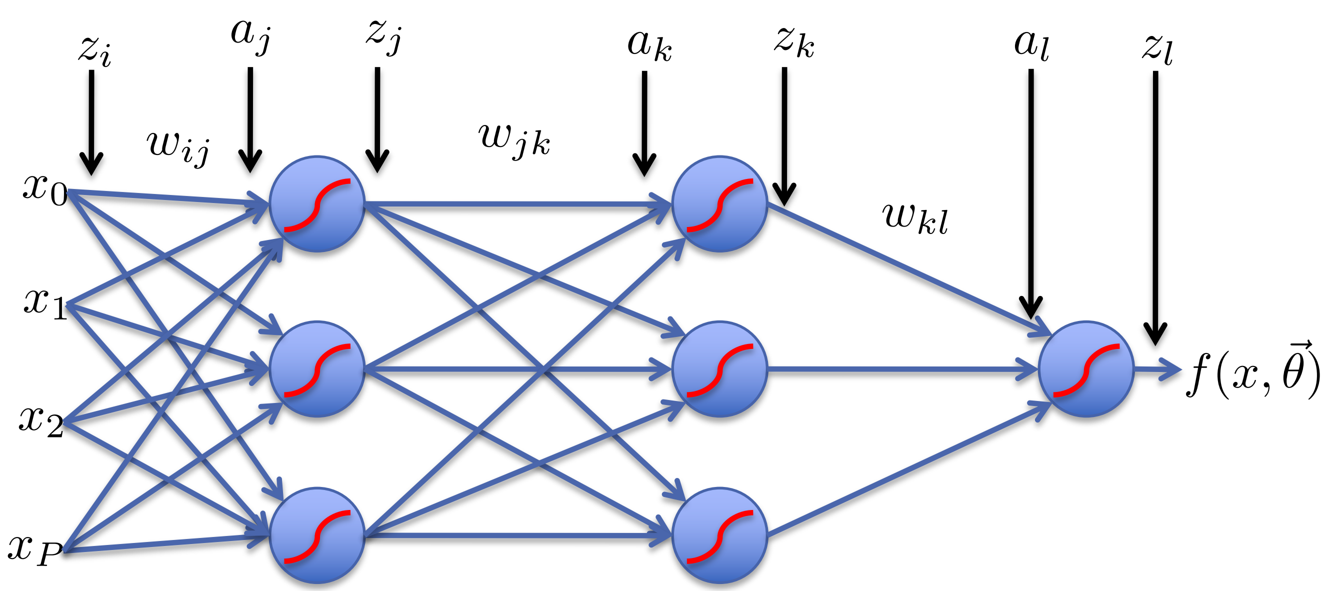
Optimize last layer weights w_{kl}

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

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

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

$\boxed{\text{Optimize last layer weights } w_{kl}}$

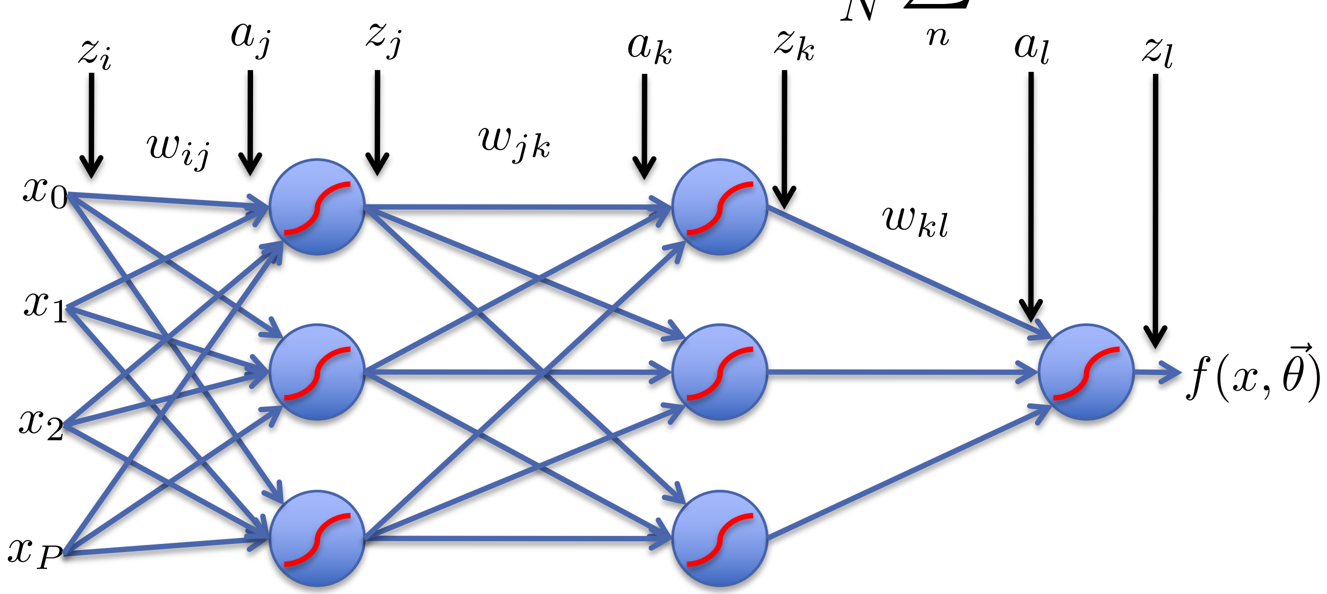
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

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

$\boxed{\text{Calculus chain rule}}$

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

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



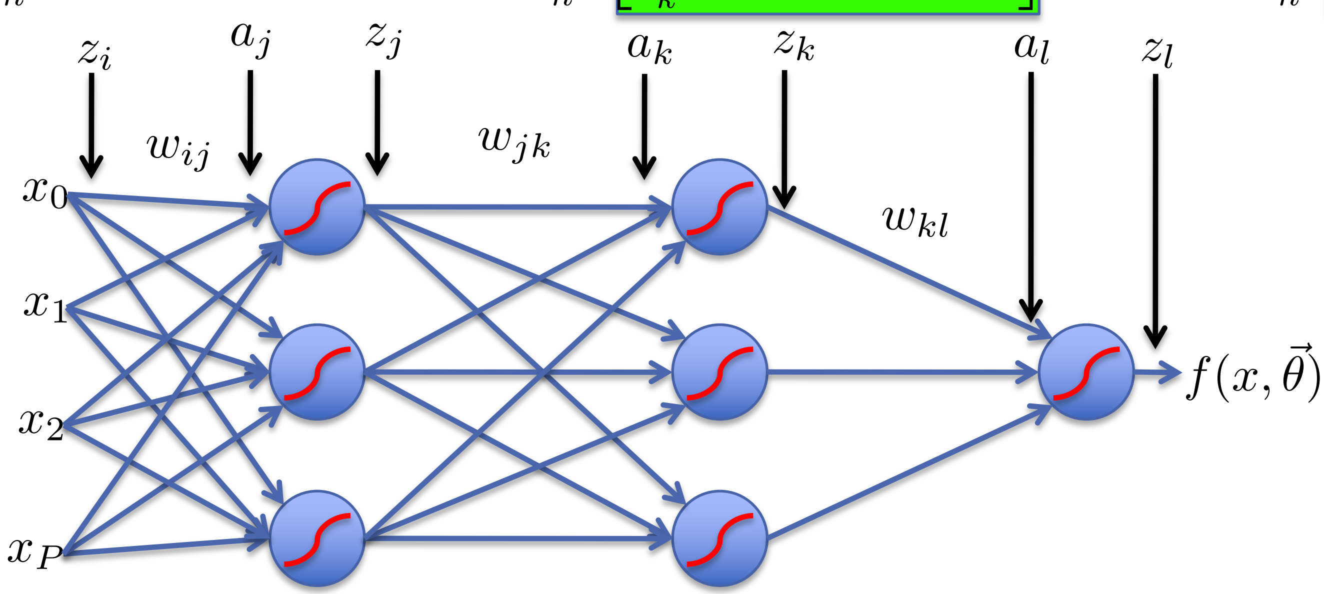
Error Backpropagation

Repeat for all previous layers

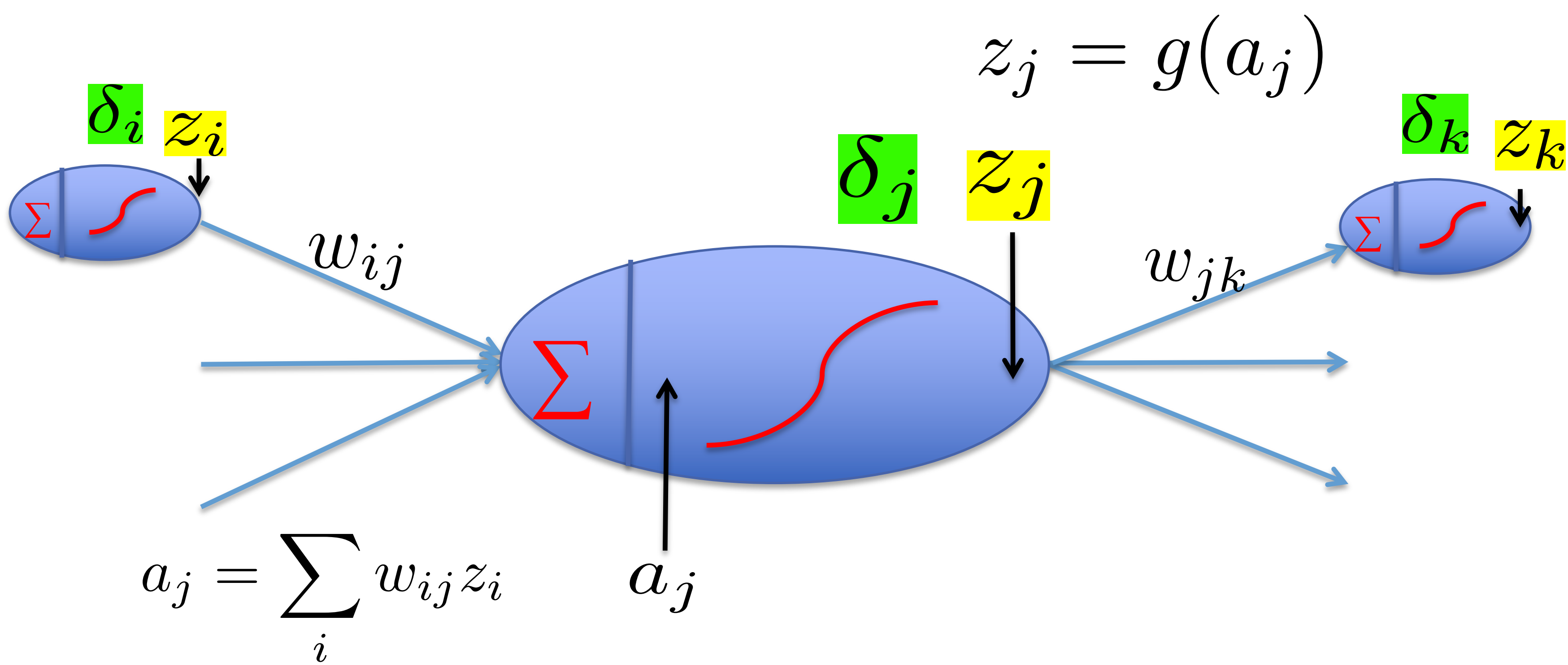
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

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

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



Backprop Recursion

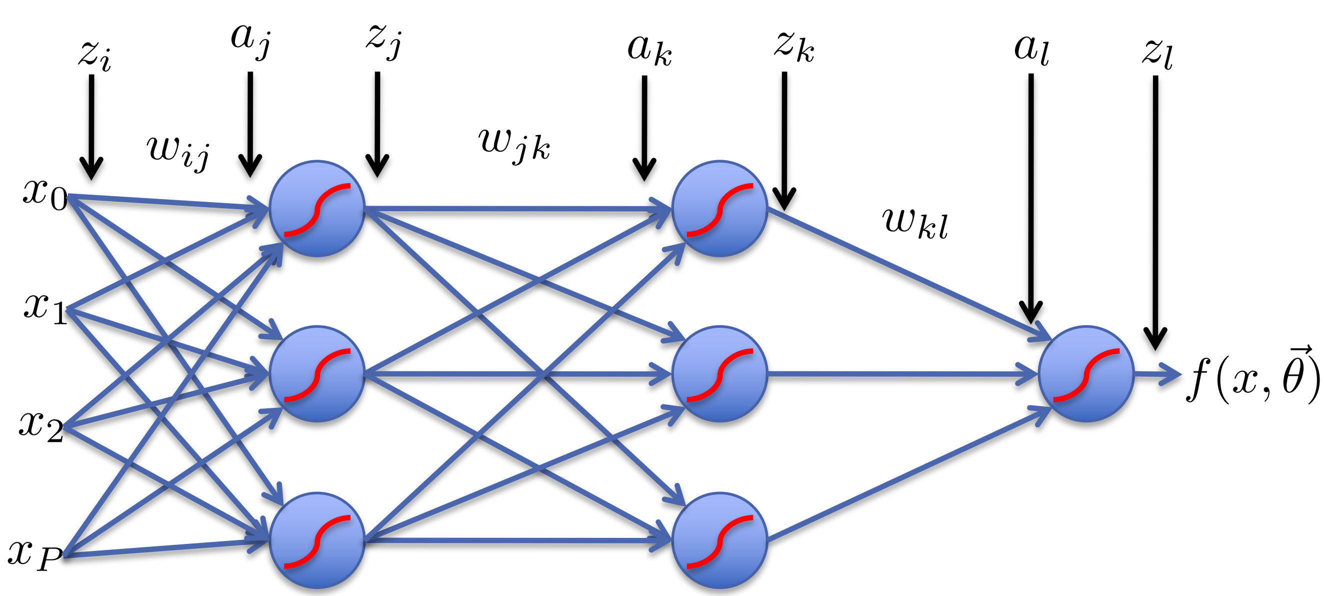


$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

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

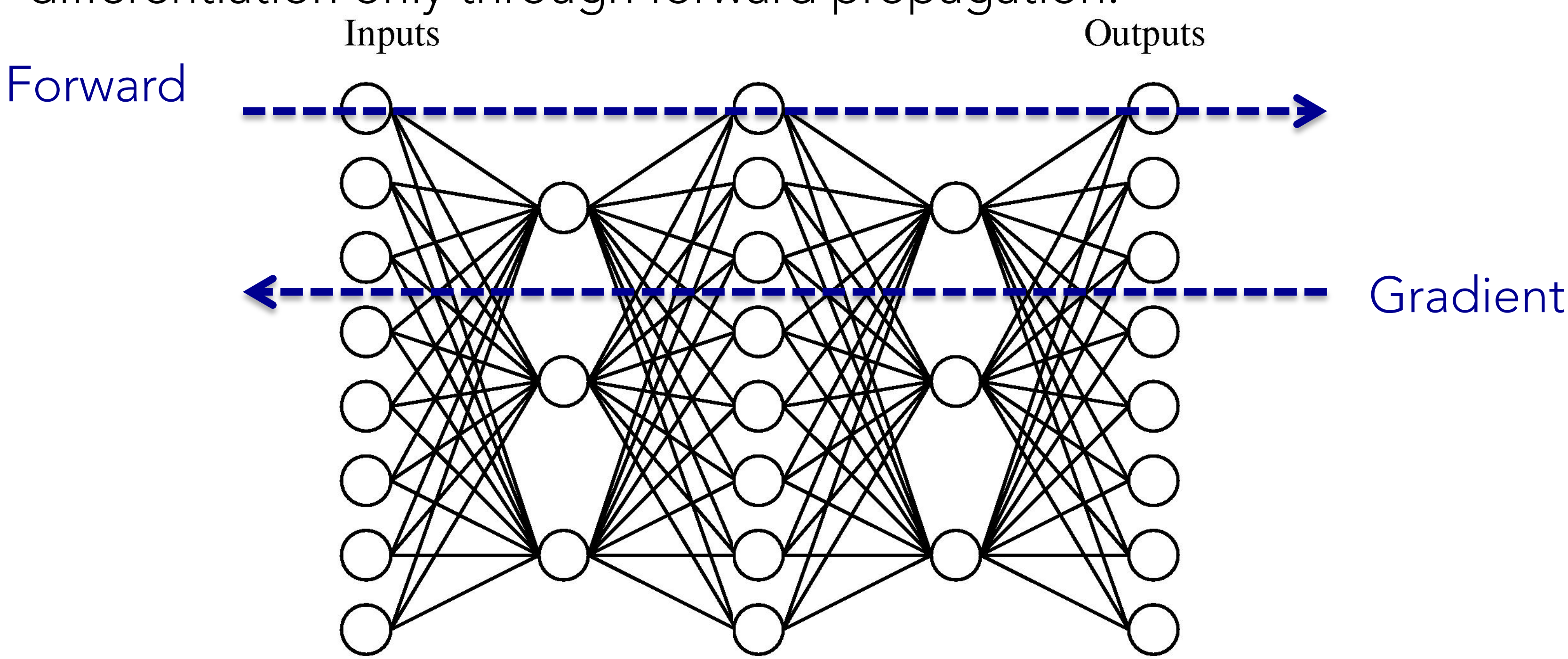
Learning: Gradient Descent

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



Backpropagation

- Starts with a forward sweep to compute all the intermediate function values z_i
- Through backprop, computes the partial derivatives recursively δ_j $\frac{\partial R}{\partial w_{ij}}$
- 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.

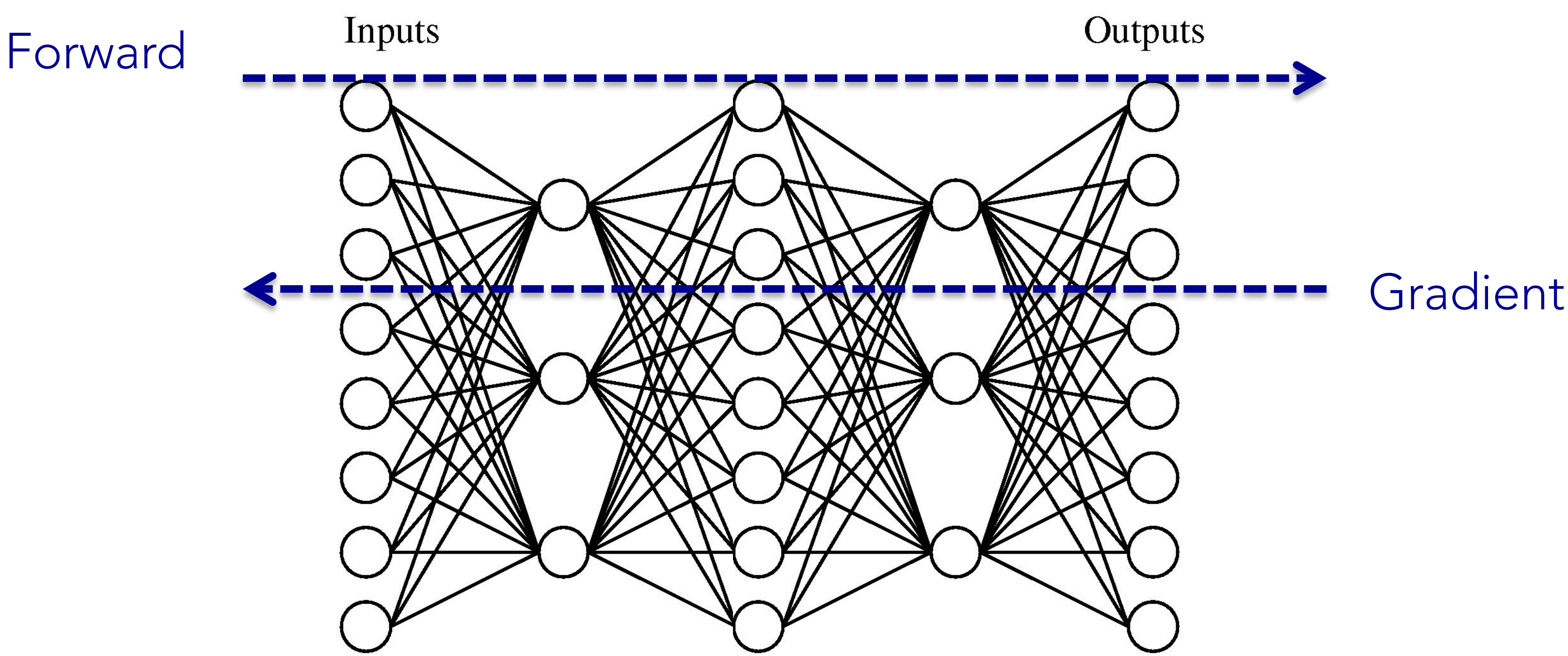


Backpropagation

- TensorFlow (https://www.tensorflow.org/)
- Torch (http://torch.ch/)
- Theano (http://deeplearning.net/software/theano/)
- CNTK (https://github.com/Microsoft/CNTK)
- cnn (https://github.com/clab/cnn)
- Caffe (http://caffe.berkeleyvision.org/)

Primary Interface Language

- Python
- Lua
- Python
- C++
- C++
- C++



Cross Entropy Loss (aka log loss, logistic loss)

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

Predicted prob

True prob

- Related quantities
  - Entropy $$H(p) = \sum_y p(y) \log p(y)$$
  - KL divergence (the distance between two distributions p and q)

$$D_{KL}(p||q) = \sum_y p(y)\ \log\ \frac{p(y)}{q(y)}$$

$$H(p,q) = E_p[-\log\ q] = H(p) + D_{KL}(p||q)$$

- Use Cross Entropy for models that should have more probabilistic flavor (e.g., language models)
- Use Mean Squared Error loss for models that focus on correct/incorrect predictions

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

Regularization

- Regularization by objective term

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \max\{0, 1 - (\hat{y}_c - \hat{y}_{c'})\} + \lambda||\theta||^2$$

  – Modify loss with L1 or L2 norms

- Less depth, smaller hidden states, early stopping
- Dropout
  – Randomly delete parts of network during training
  – Each node (and its corresponding incoming and outgoing edges) dropped with a probability p
  – P is higher for internal nodes, lower for input nodes
  – The full network is used for testing
  – Faster training, better results

Convergence of backprop

- Without non-linearity or hidden layers, learning is convex optimization
  - Gradient descent reaches **global minima**
- Multilayer neural nets (with nonlinearity) are **not convex**
  - Gradient descent gets stuck in local minima
  - Selecting number of hidden units and layers = fuzzy process
  - NNs have made a HUGE comeback in the last few years
    - Neural nets are back with a new name
      - Deep belief networks
      - Huge error reduction when trained with lots of data on GPUs