Quick Intro to Neural Networks

COSC 6336: Natural Language Processing
Spring 2020

Some content in these slides has been adapted from Jurafsky & Martin 3rd edition, and lecture slides from Ray Mooney and the deep learning course by Manning and Socher and Greg Durret.

UNIVERSITY of HOUSTON

Today's Lecture

- ★ Neural network
  - Basic unit
  - Backpropagation

Neural Networks (NNs)

- ★ Inspired by human neurons
- ★ Feedforward NN
- ★ Modern NNs are called “deep”
- ★ Current NN models simultaneously induce features while learning the classification task

Neuron

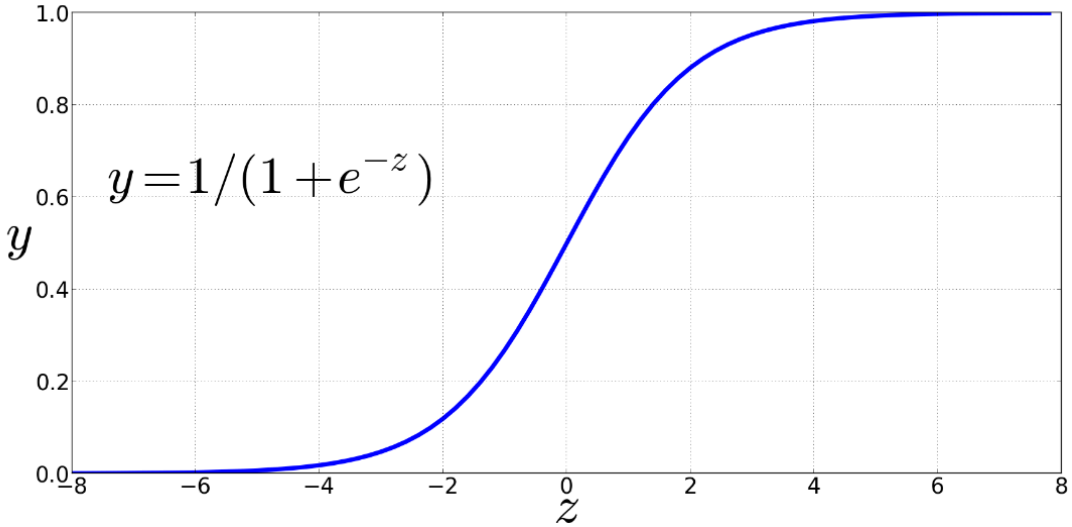
- The building block of an NN is a single unit

$$z = b + \sum_i w_i x_i$$

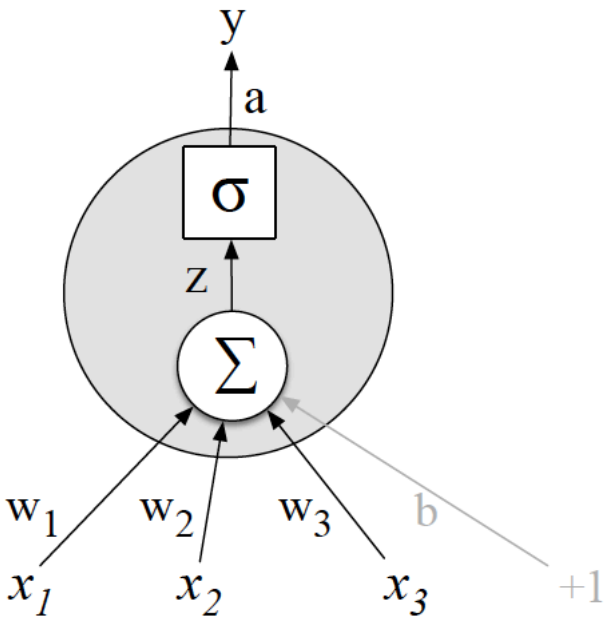
$$z = w \cdot x + b$$

Non-linearity

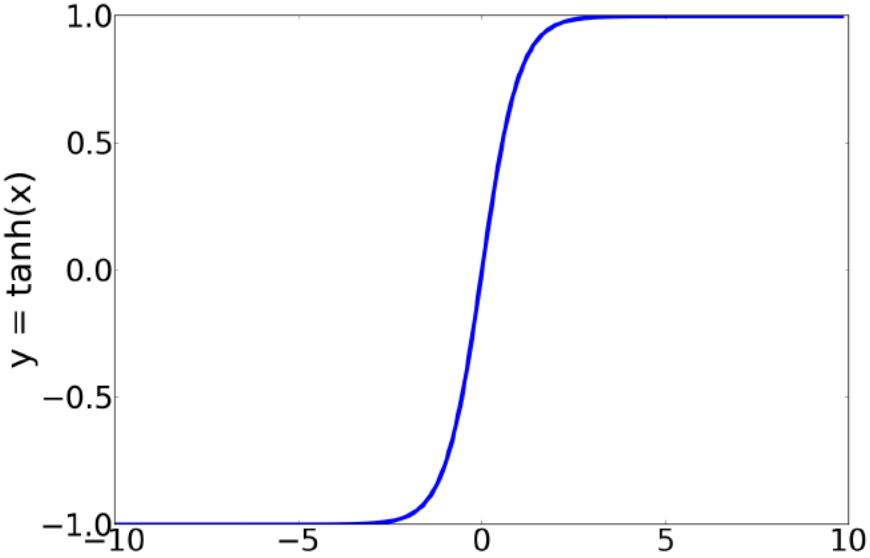
$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$



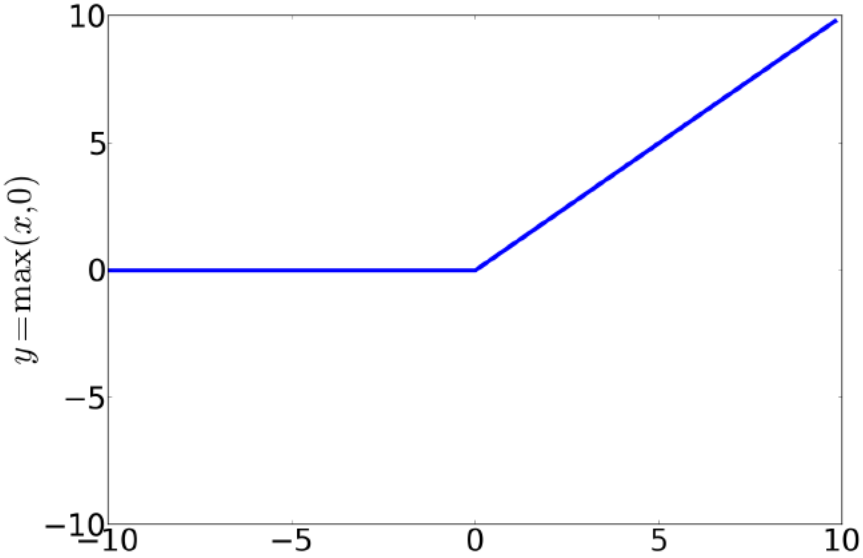
Graphical Representation of a single neuron



tanh and ReLU



$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



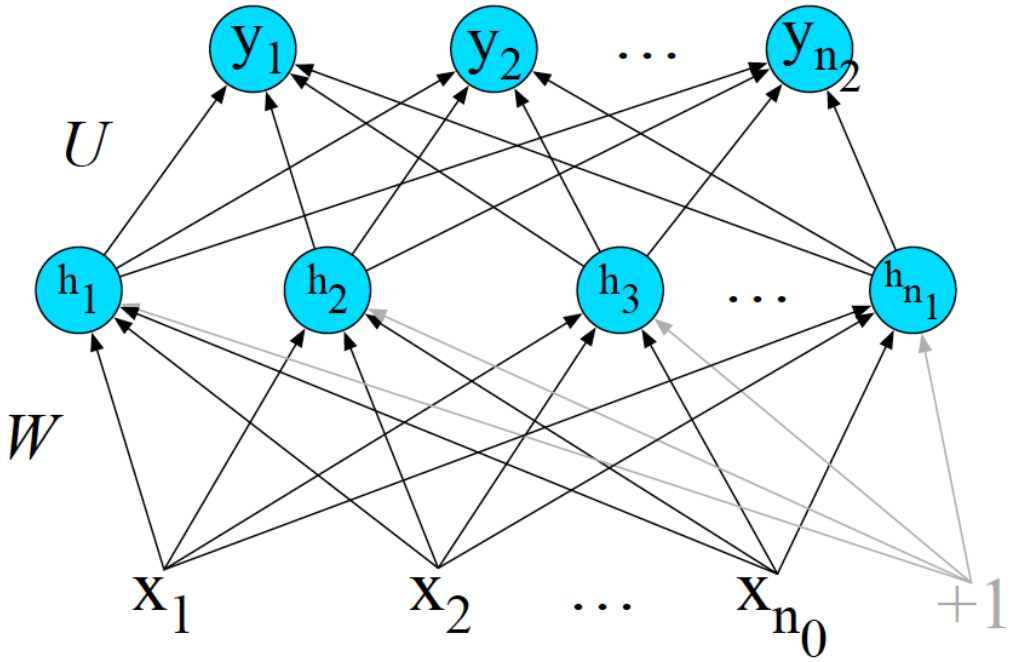
$$y = max(x, 0)$$

UNIVERSITY of HOUSTON

Feedforward Neural Networks

- Fully connected
- Three kinds of nodes:
  - Input units
  - Hidden unites
  - Output units

Feedforward Neural Networks



$$
\begin{aligned}
h &= \sigma(Wx+b) \\
z &= Uh \\
y &= \text{softmax}(z)
\end{aligned}
$$

$$
\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \le i \le d
$$

Notation

- Superscripts denote layer numbers
- First layer is layer 0
- n_j is number of units in layer j
- g(•) is activation function
- $a^{[i]}$ is output from layer i
- $z^{[i]}$ is combination of weights and biases $W^{[i]}\, a^{[i-1]} + b^{[i]}$

$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

Computing the forward step

for i **in** 1..n

$$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}$$

$$a^{[i]} = g^{[i]}(z^{[i]})$$

$$\hat{y} = a^{[n]}$$

Training in NNs

- Common loss function is cross-entropy loss
- To tune parameters we will use gradient descent optimization
- This requires us to compute the gradient
- For logistic regression this is straightforward, but for NNs this is not the case
- We will use error backpropagation (reverse differentiation)

Loss function

- If we have a one layer NN in a binary classification task:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

- For c classes:

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

- which can be simplified as:

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i$$

Computing the gradient in NNs

- If we have one layer with a sigmoid output unit:

$$(\hat{y} - y)\, x_j$$

- If we have one layer wit

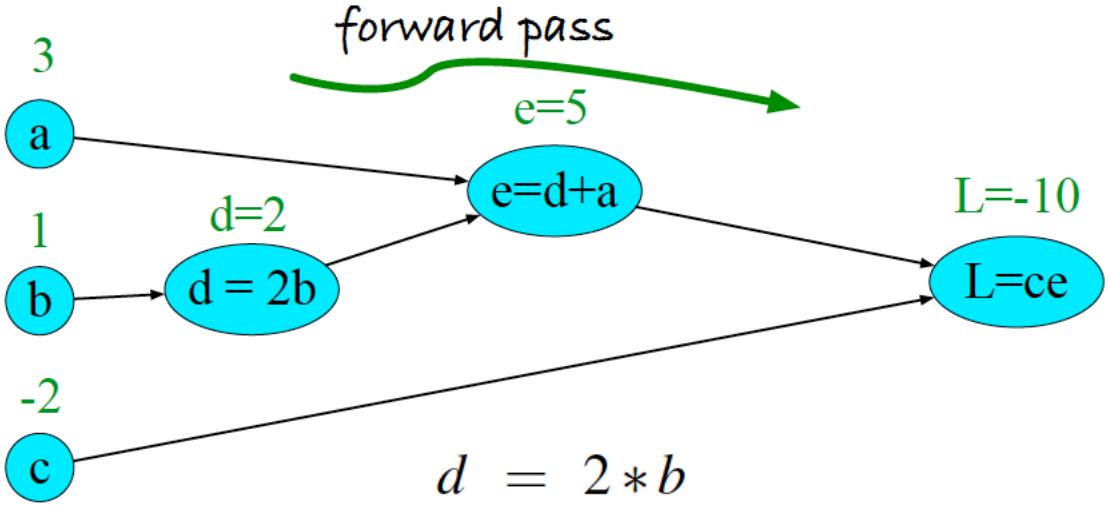
$$(\sigma(w \cdot x + b) - y)\, x_j$$

How do we compute gradients for networks with many hidden layers?

$$\frac{\partial L_{CE}}{\partial w_k} = (1\{y = k\} - p(y = k|x))x_k$$

$$= \left(1\{y = k\} - \frac{e^{w_k \cdot x + b_k}}{\sum_{j=1}^{K} e^{w_j \cdot x + b_j}}\right) x_k$$

Computation Graphs



$$d = 2 * b$$

$$e = a + d$$

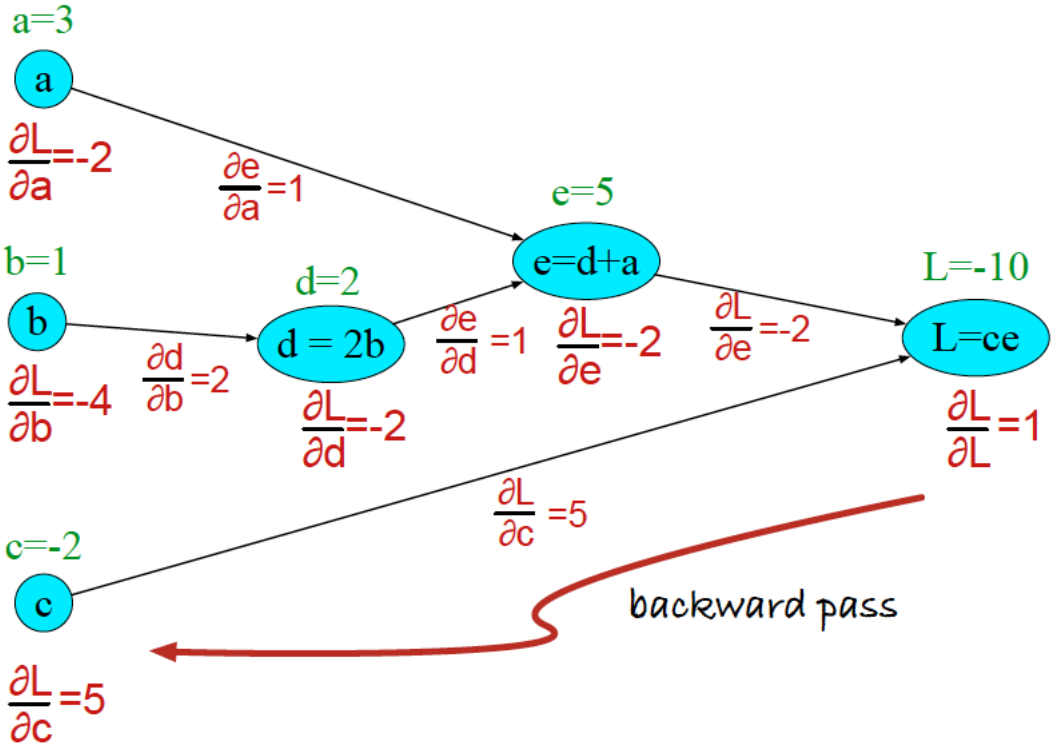
$$L = c * e$$

Computation Graphs: backward differentiation

- Let's compute the derivate of output function L with respect to each of the input variables $\partial L/\partial a$, $\partial L/\partial b$, and $\partial L/\partial c$.
- We use the chain rule of calculus, the derivative of composite function $f(x) = u(v(w(x)))$:

$$df/dx = du/dv \cdot dv/dw \cdot dw/dx$$

Computation Graphs



$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$L = ce \ : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

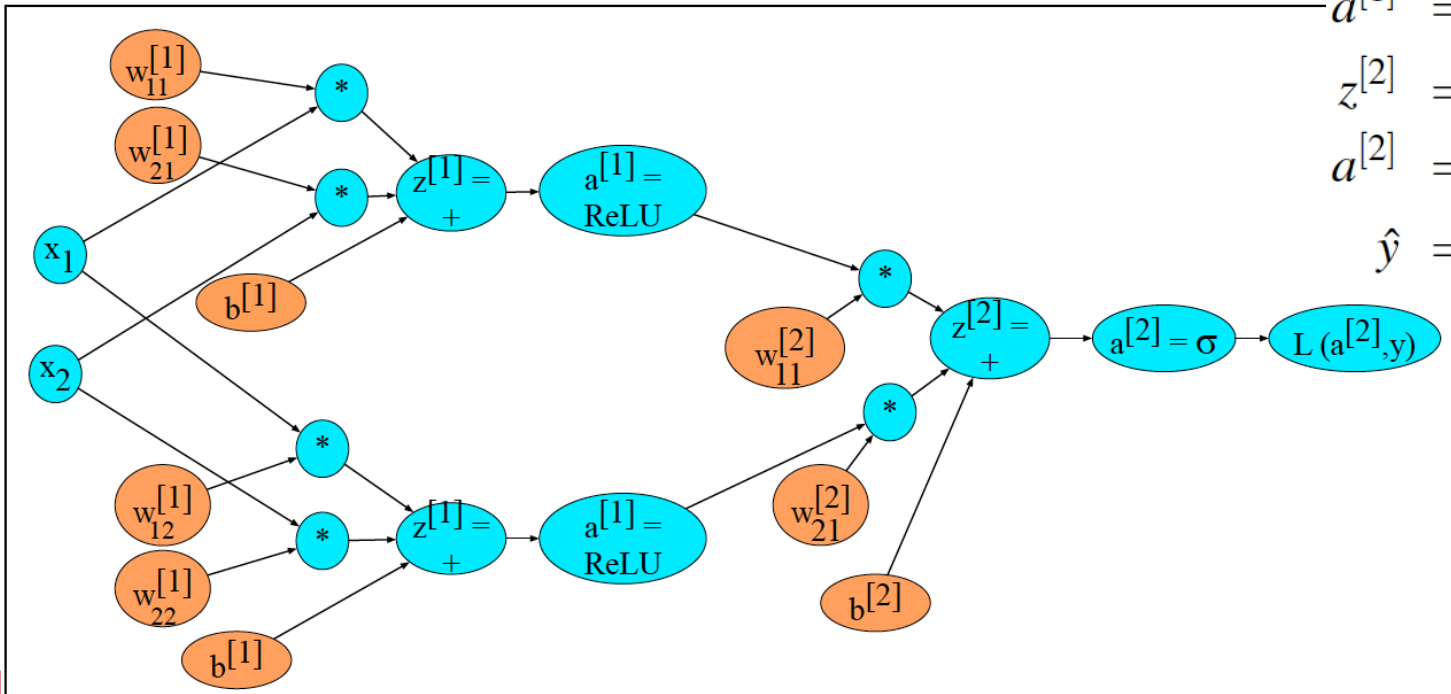
$$e = a + d \ : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \ : \quad \frac{\partial d}{\partial b} = 2$$

Backward differentiation for a neural network

Here layer 0 has 2 inputs, n_1 =2 and n_2=1

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



Derivatives for activation functions

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z)$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad x<0 \\ 1 & for \quad x\geq 0 \end{cases}$$

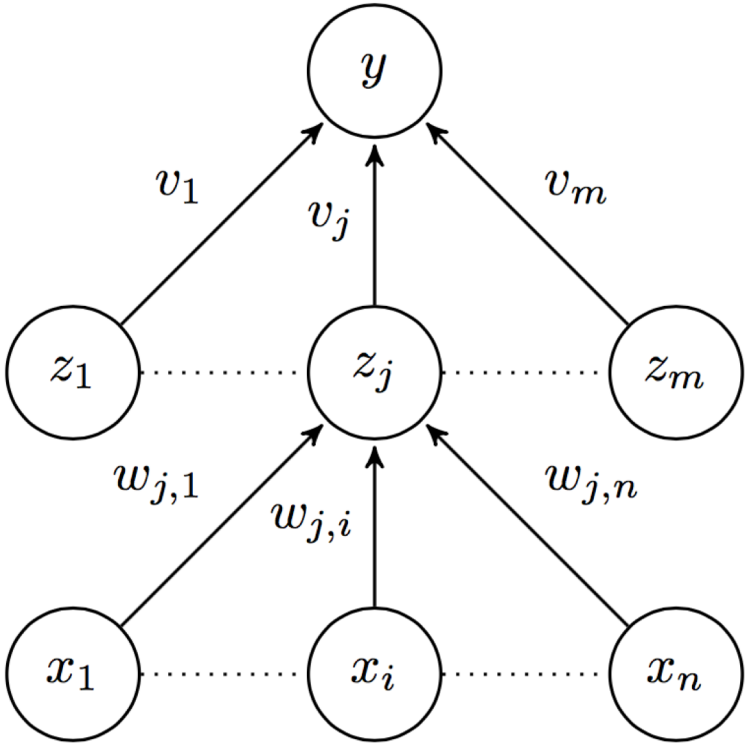
Neural Network Training

Training multilayer networks

- ★ We will use gradient descent as well.
- ★ We need to calculate

$$\frac{\partial E_\ell}{\partial w_{ji}}$$

- ★ An analytical solution gets very complicated even for a small NN
- ★ Backpropagation is an efficient strategy for gradient calculation



Rumelhart, D.; Hinton, G.; Williams, R. (1986). "Learning representations by back-propagating errors". *Nature*. **323** (6088): 533–536.

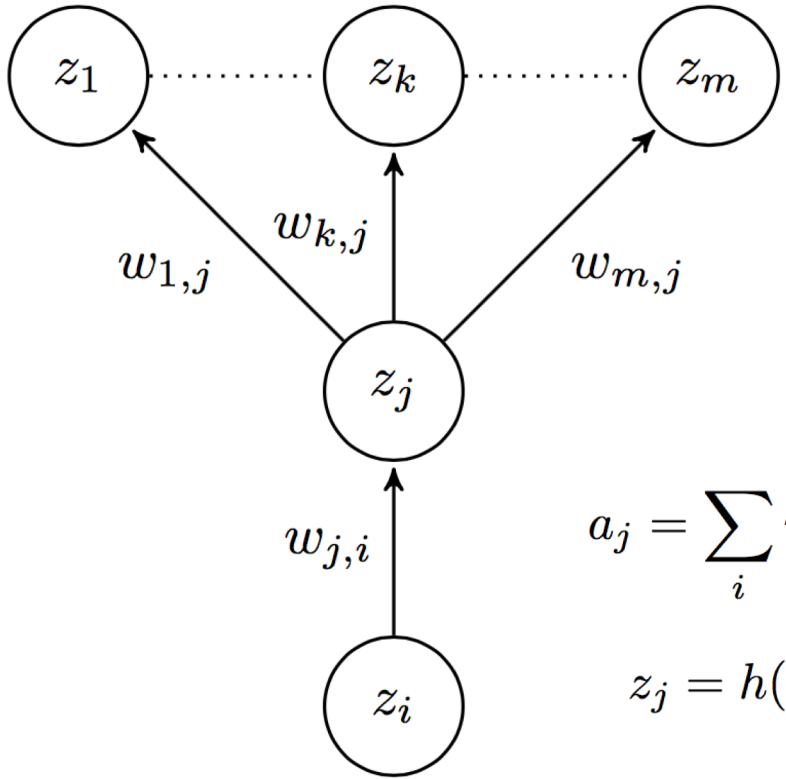
Gradient calculation

- ★ General case for a neuron z_j and z_i in layers n and n-1 respectively
- ★ We want to calculate the gradient:

$$\frac{\partial E_\ell}{\partial w_{ji}}$$

- ★ General strategy: re-express the gradient as a function of two values

$$\frac{\partial E_\ell}{\partial w_{ji}} = \delta_j z_i$$

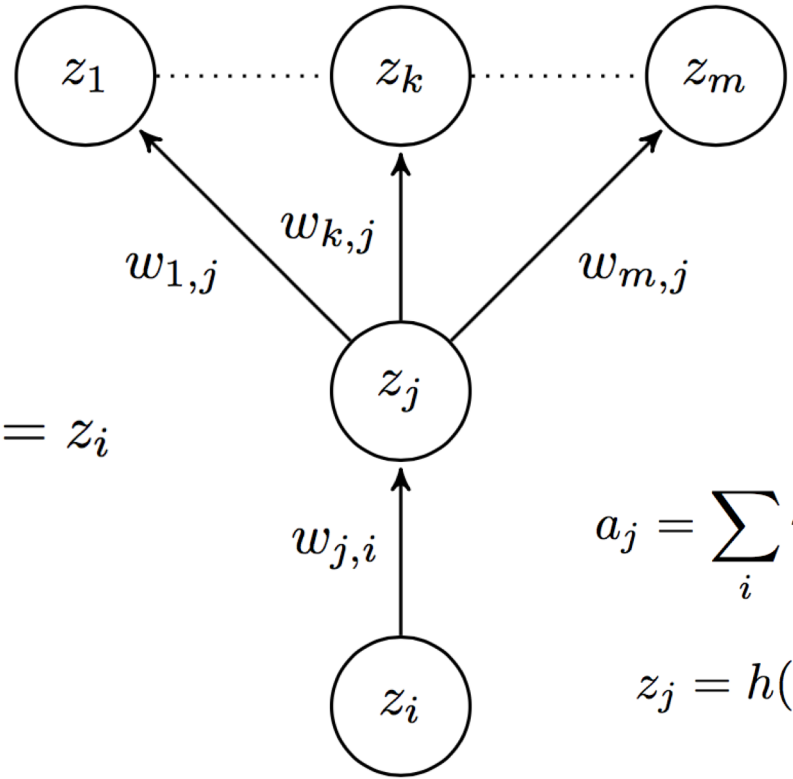


Gradient decomposition

$$\frac{\partial E_\ell}{\partial w_{ji}} = \frac{\partial E_\ell}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$ (chain rule)

$$\delta_j = \frac{\partial E_\ell}{\partial a_j} \qquad \frac{\partial a_j}{\partial w_{ji}} = \frac{\partial \sum_i w_{ji} z_i}{\partial w_{ji}} = z_i$$

$$\frac{\partial E_\ell}{\partial w_{ji}} = \delta_j z_i$$



Generalized (multidimensional) chain rule (GCR)

$$y = f(u_1, \ldots, u_m)$$

$$\mathbf{u} = g(x_1, \ldots, x_n)$$

$$\frac{\partial y}{\partial x_i} = \sum_{\ell=1}^{m} \frac{\partial y}{\partial u_\ell} \frac{\partial u_l}{\partial x_i}$$

δ_j recursive calculation

$$\delta_j = \frac{\partial E_\ell}{\partial a_j} = \sum_{k=1}^{m} \frac{\partial E_\ell}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

Applying GCR since $E_\ell = f(a_1, \ldots, a_k)$

$$= \sum_{k=1}^{m} \delta_k \frac{\partial a_k}{\partial a_j}$$

Definition of δ_j

$$= \sum_{k=1}^{m} \delta_k \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j}$$

Chain rule

$$= \sum_{k=1}^{m} \delta_k w_{kj} h'(a_j)$$

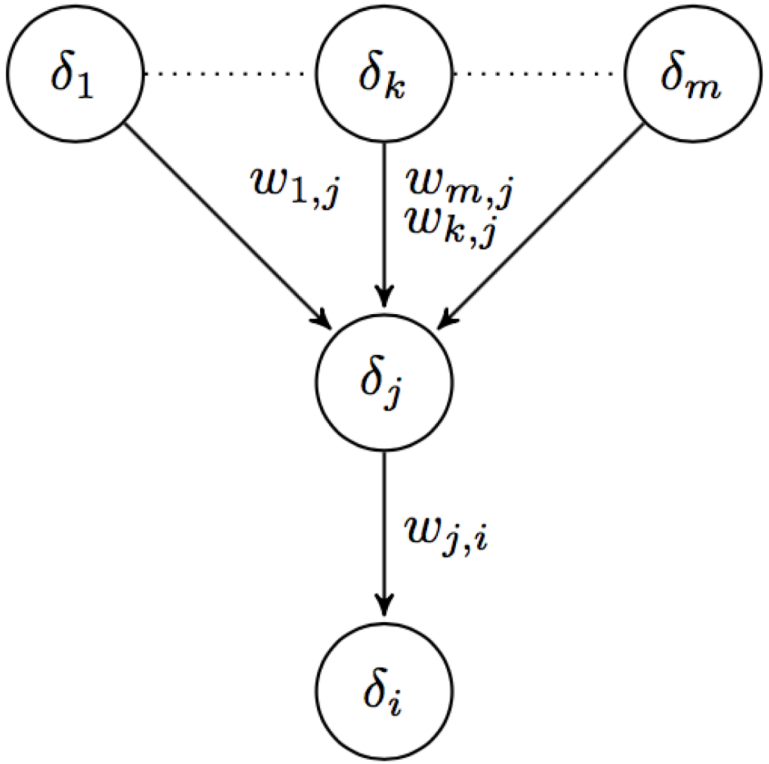
$$= h'(a_j) \sum_{k=1}^{m} \delta_k w_{kj}$$

UNIV
HO

δ_j backpropagation

Deltas in layer *n* are calculated from deltas in layer *n + 1*:

$$\delta_j = h'(a_j) \sum_{k=1}^{m} \delta_k w_{kj}$$



Backpropagation algorithm

1: Initialize $\mathbf{w}$
2: for n = 1 to num epochs
3: for all $x^{\ell} \in D$
4: Forward propagate x^{ℓ} through the network to calculate the a_j and z_j values
5: Calculate $\delta_o = \frac{\partial E_{\ell}}{\partial a_o}$ for all the output neurons
6: Backward propagate δ_j values $\delta_j = h'(a_j) \sum_{k=1}^{m} \delta_k w_{kj}$
7: for all $w_{ji} \in \mathbf{w}$
8: $\Delta w_{ji} \leftarrow \delta_j z_i$
9: $w_{ji} \leftarrow w_{ji} - \eta_n \Delta w_{ji}$

UNIVERS
HOUSTON

Two layers NN with sigmoid activation

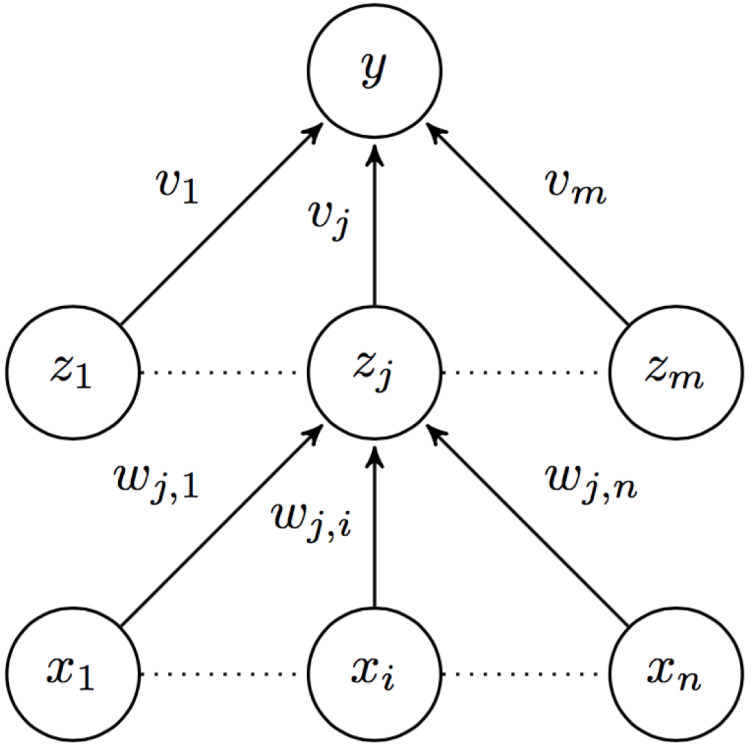
$$a_j = \sum_i w_{ji} x_i$$

$$z_j = \sigma(a_j)$$

$$a_y = \sum_j v_j z_j$$

$$y = \sigma(a_y)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



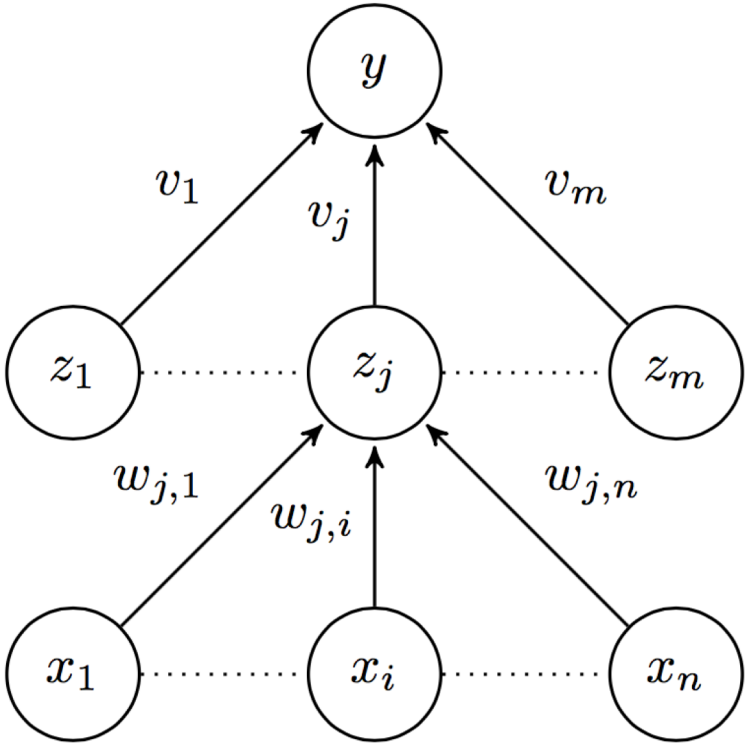
UNIVERSI
HOUST

Two layers NN with sigmoid activation

$$E_\ell(w) = -r^\ell \log y^t - (1 - r^\ell) \log(1 - y^t)$$

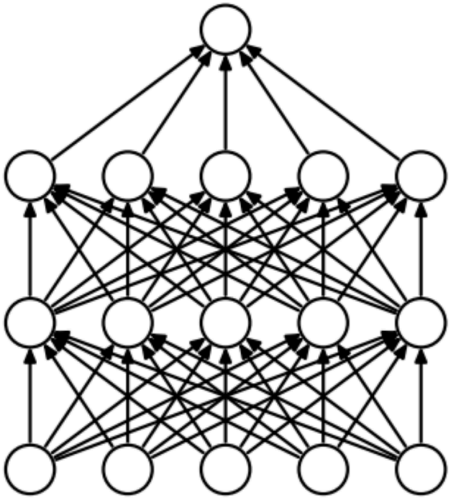
$$\delta_y = \frac{\partial E_\ell}{\partial a_y} = \sigma(a_y) - r^\ell = y - r^\ell$$

$$\delta_j = \sigma'(a_j)\delta_y v_j = \sigma(a_j)(1 - \sigma(a_j))\delta_y v_j = z_j(1 - z_j)\delta_y v_j$$

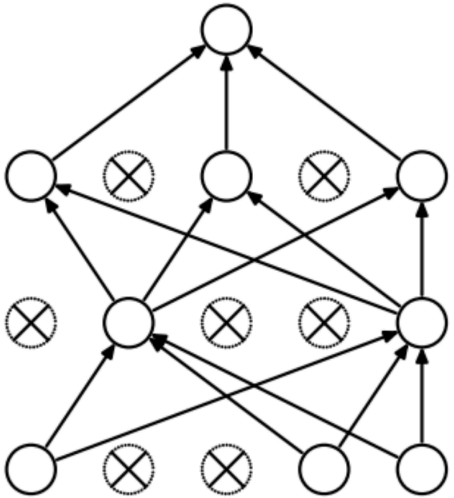


Training Tricks

- Parameter
- Hyperparameters
- Batch backpropagation
- Initialization
- Batch normalization
- Drop out
- Optimizer
- Gradient clipping



(a) Standard Neural Net



(b) After applying dropout.