CS6375: Machine Learning

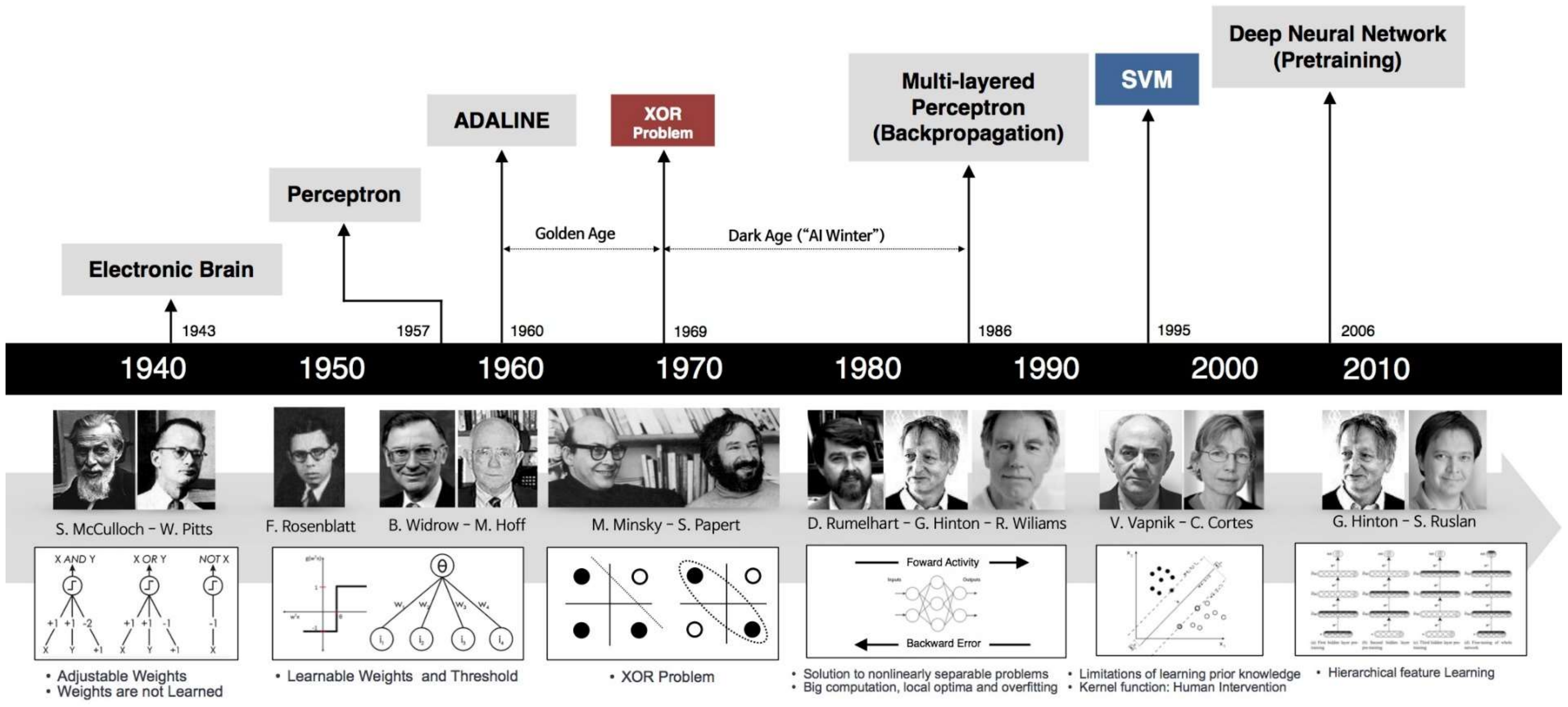
Gautam Kunapuli

TERMINATOR 2: JUDGMENT DAY (1991)
My CPU is a Neural Net processor... a learning computer!

THE UNIVERSITY OF TEXAS AT DALLAS

Erik Jonsson School of Engineering and Computer Science

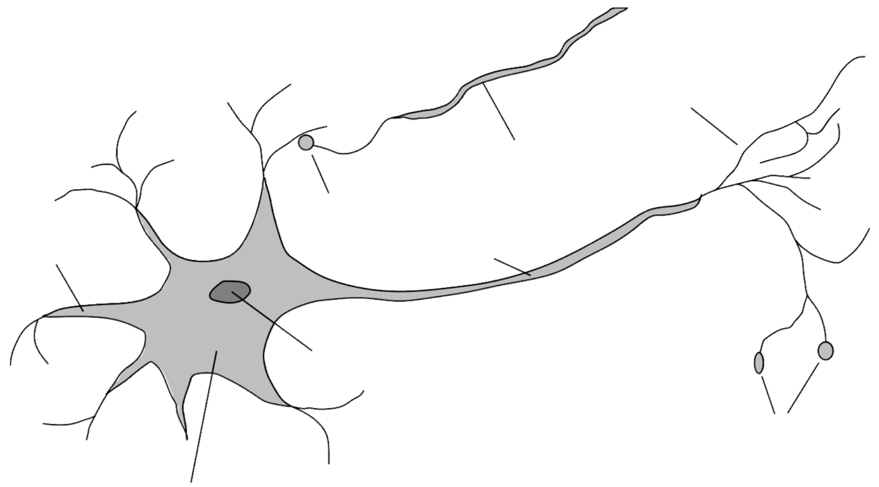
Neural Networks: A Brief History


Electronic Brain
Perceptron
ADALINE
XOR Problem
Multi-layered Perceptron (Backpropagation)
SVM
Deep Neural Network (Pretraining)
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Source: Unknown

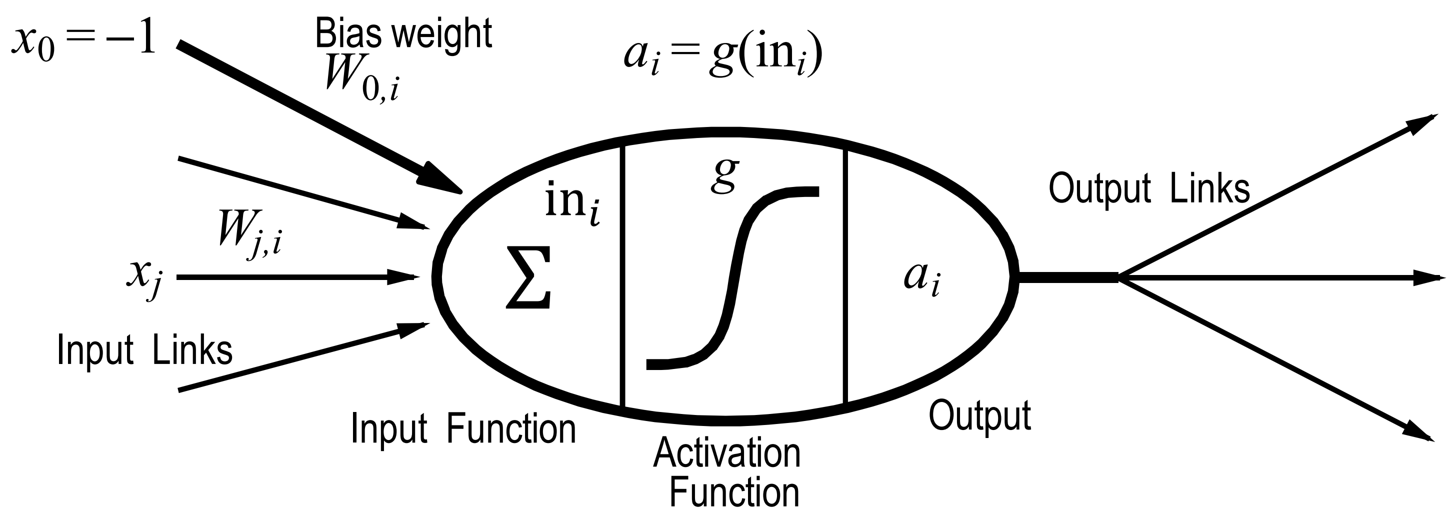
Neural Networks

- The basis of **neural networks** was developed in the 1940s-1960s
- The idea was to build **mathematical models** that might **"compute" in the same way that neurons** in the brain do
- As a result, neural networks are **biologically inspired**, though many of the **algorithms** developed for them are **not biologically plausible**
- Perform surprisingly well for many tasks



10^{11} neurons of more than 20 types, 10^{14} synapses, 1ms–10ms cycle time; signals are noisy "spike trains" of electrical potential

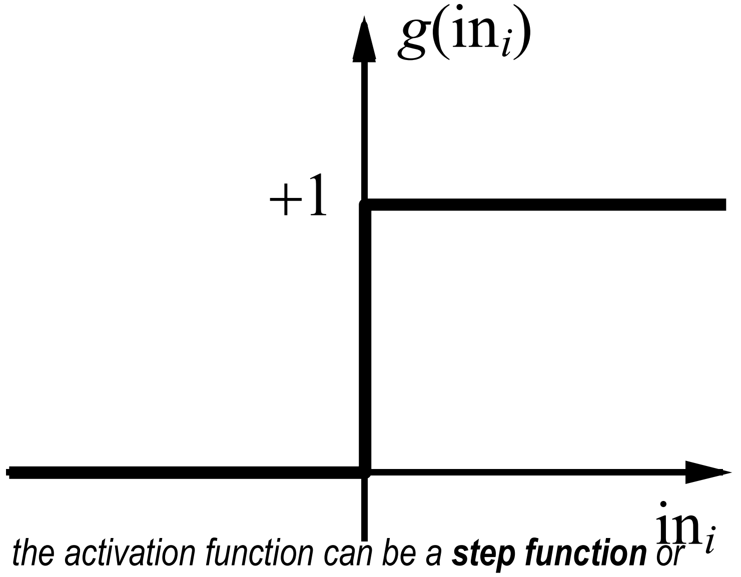
McCulloch-Pitts "unit": $a_i \leftarrow g(\text{in}_i) = g\left(\sum_j W_{j,i} x_j\right)$



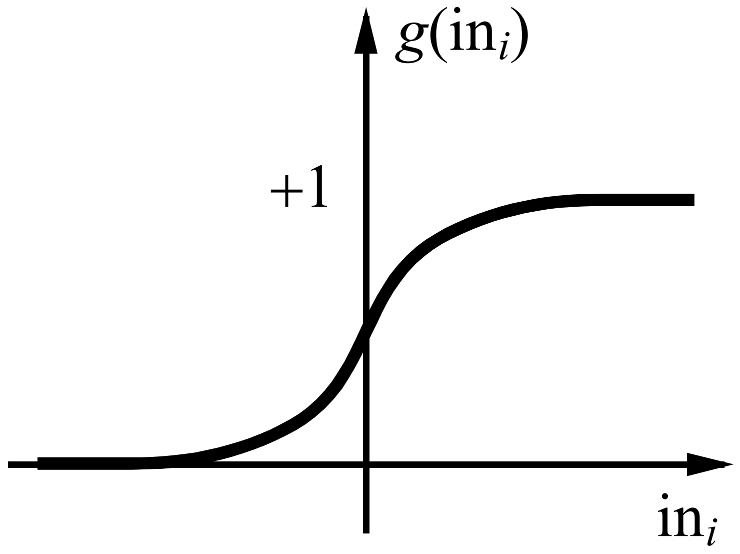
A gross oversimplification of real neurons, but its purpose is to develop an understanding of what networks of simple units can do

Neural Networks

- Neural networks consist of a **collection of artificial neurons**
- There are different types of neuron activation functions
    - the **perceptron** (one of the first studied)
    - the **sigmoid** neuron (one of the most common)
    - **rectified linear units** (deep learning)
- A neural network is a **directed graph** consisting of a collection of neurons (the **nodes**), directed **edges** (each with an associated **weight**), and a collection of **fixed binary inputs**



*the activation function can be a **step function** or **threshold function;** changing the bias weight $W_{0,i}$ moves the threshold location*



*the activation function can be a **sigmoid function**: $\frac{1}{1+e^{-x}}$; changing the bias weight $W_{0,i}$ moves the threshold location*

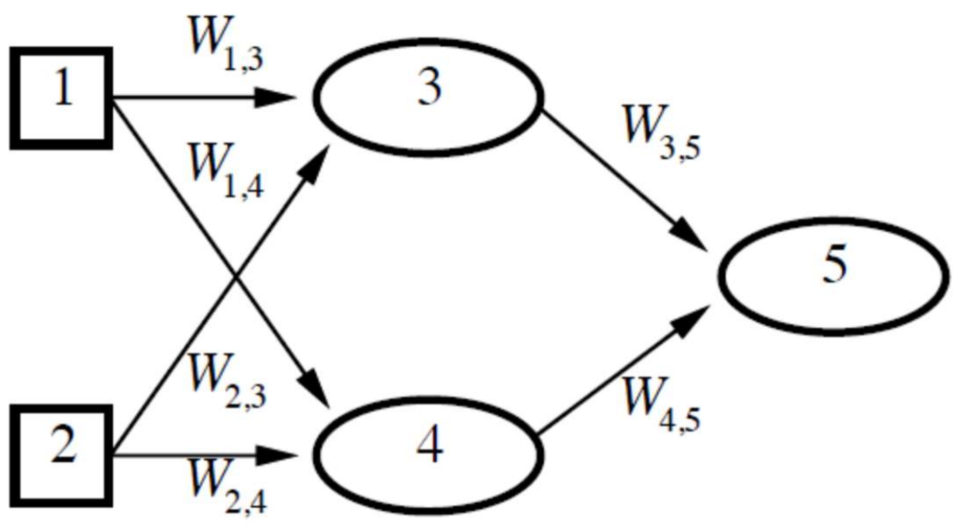
Network Architectures

Feed-forward networks implement functions, have no internal state
- single-layer perceptrons
- multi-layer perceptrons

Recurrent networks have directed cycles with delays
- have internal state (like flip-flops), can oscillate etc.

A **feed-forward network** is a parameterized family of nonlinear functions; adjusting the weights changes the function

Learning problem: learn the weights for a given architecture

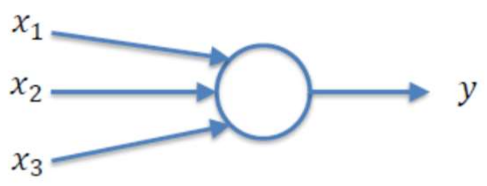


$$\begin{aligned} a_5 &= g(W_{3,5} \,.\, a_3 + W_{4,5} \,.\, a_4) \\ &= g(W_{3,5} \,.\, g(W_{1,3} \,.\, a_1 + W_{2,3} \,.\, a_2) + W_{4,5} \,.\, g(W_{1,4} \,.\, a_1 + W_{2,4} \,.\, a_2)) \end{aligned}$$

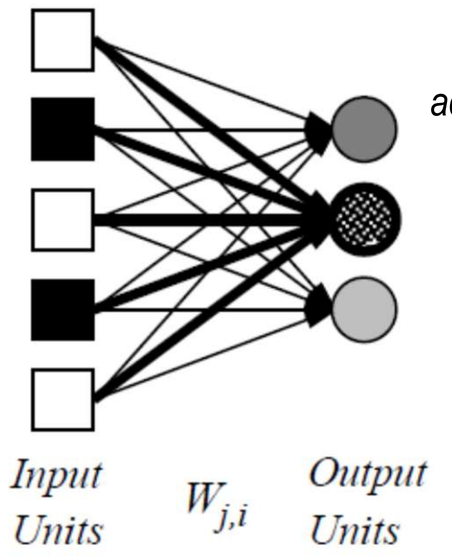
Single-Layer Perceptron

A **perceptron** is an artificial neuron that takes a collection of binary inputs and produces a binary output

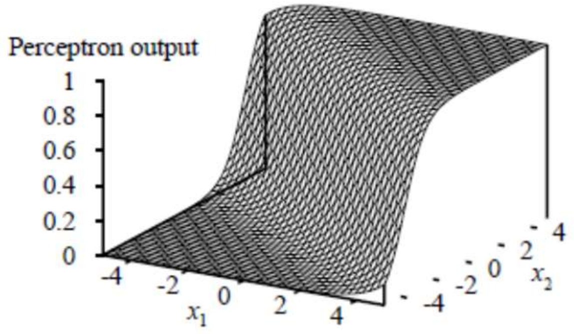
- The output of the perceptron is determined by summing up the weighted inputs and thresholding the result
- if the weighted sum is larger than the threshold, the output is one (and zero otherwise)
- the perceptron algorithm we previously studied uses the hard step function $g = \text{step}(\cdot)$



$$y = \begin{cases} 1 & w_1x_1 + w_2x_2 + w_3x_3 + b > 0 \\ 0 & otherwise \end{cases}$$



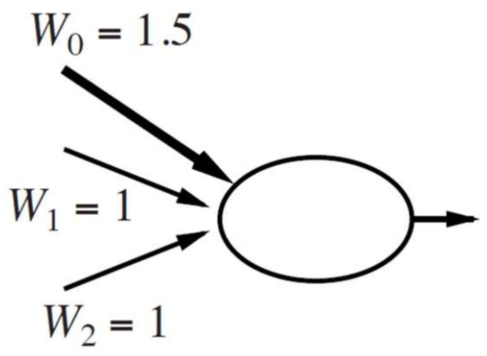
adjusting weights moves the location, orientation, and steepness of the thresholding cliff



Single-Layer Perceptron

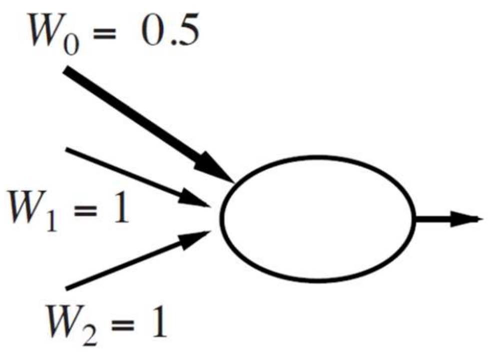
A perceptron can represent the **Boolean** functions **and**, **or** and **not** easily

$$y = \begin{cases} 1 & x_1 + x_2 > 1.5 \\ 0 & x_1 + x_2 \leq 1.5 \end{cases}$$



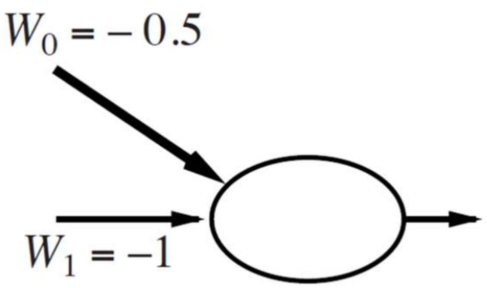
AND

$$y = \begin{cases} 1 & x_1 + x_2 > 0 \\ 0 & x_1 + x_2 \leq 0 \end{cases}$$



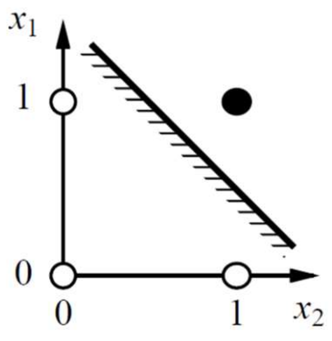
OR

$$y = \begin{cases} 1 & -x > -.5 \\ 0 & -x \leq -.5 \end{cases}$$

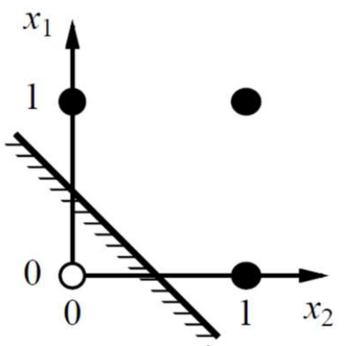


NOT

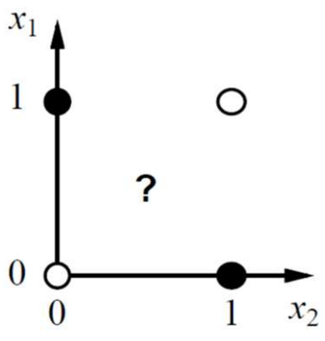
and/**or** can be represented as linear functions, but **xor**?



(a) x_1 **and** x_2



(b) x_1 **or** x_2



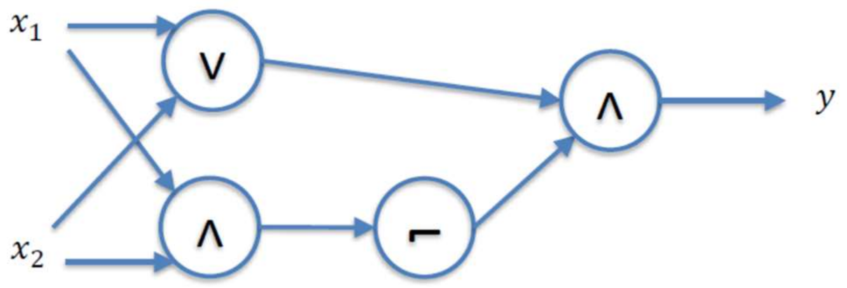
(c) x_1 **xor** x_2

Multi-Layer Perceptron

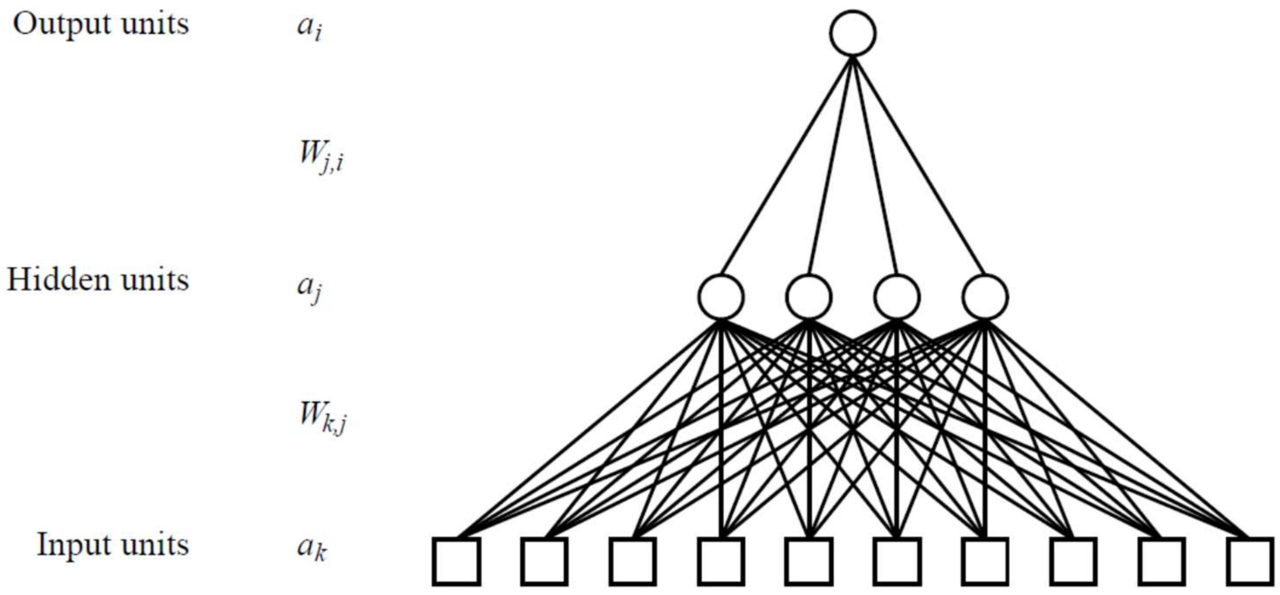
Recall that the **xor** function can be written as:

$$x_1 \oplus x_2 = (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$$

Can be expressed by combining **multiple perceptron units with multiple layers**!



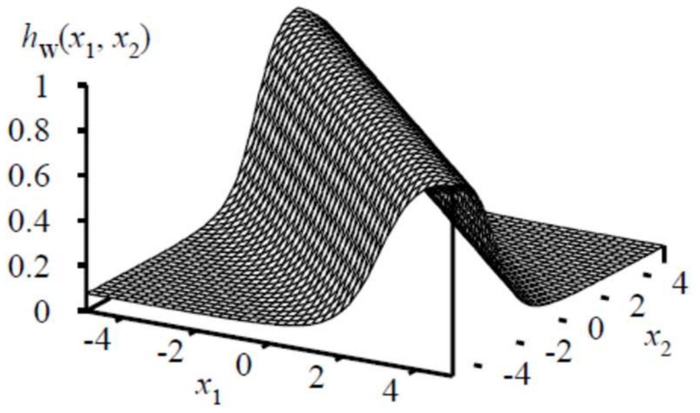
Gluing a bunch of perceptrons together gives us a **neural network**
- in general, neural nets have a collection of inputs and a collection of outputs; can be binary, continuous (need appropriate loss functions)
- layers are usually **fully connected**
- numbers of **hidden units** typically chosen by hand

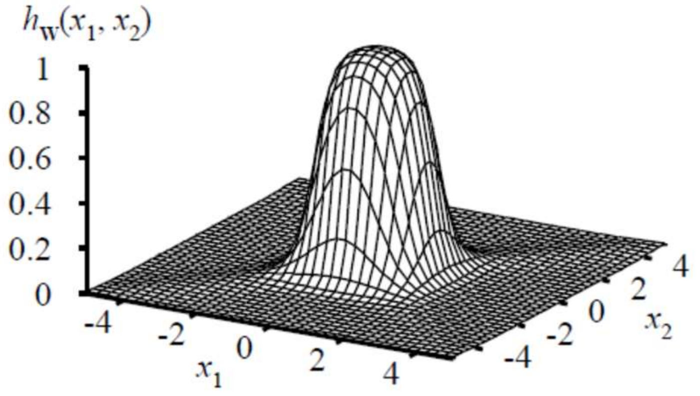


Multi-Layer Perceptron

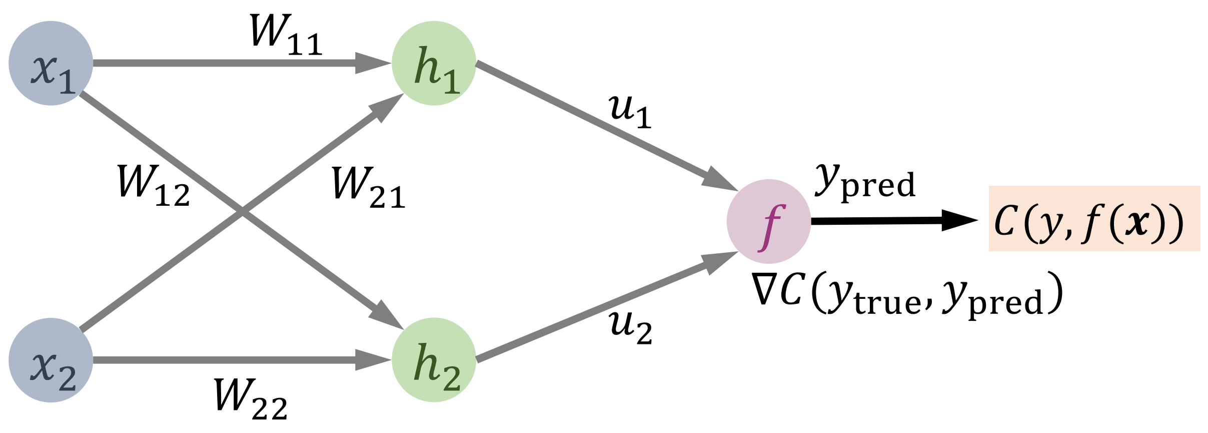
Multi-layer perceptrons can encode all continuous functions **with 2 layers**,
all functions **with 3 layers**

- combine two opposite-facing threshold functions to make a **ridge**
- combine two perpendicular ridges to make a **bump**
- add bumps of various sizes and locations to fit any **surface**
- proof requires **exponentially many hidden units**





Backpropagation: Forward Pass



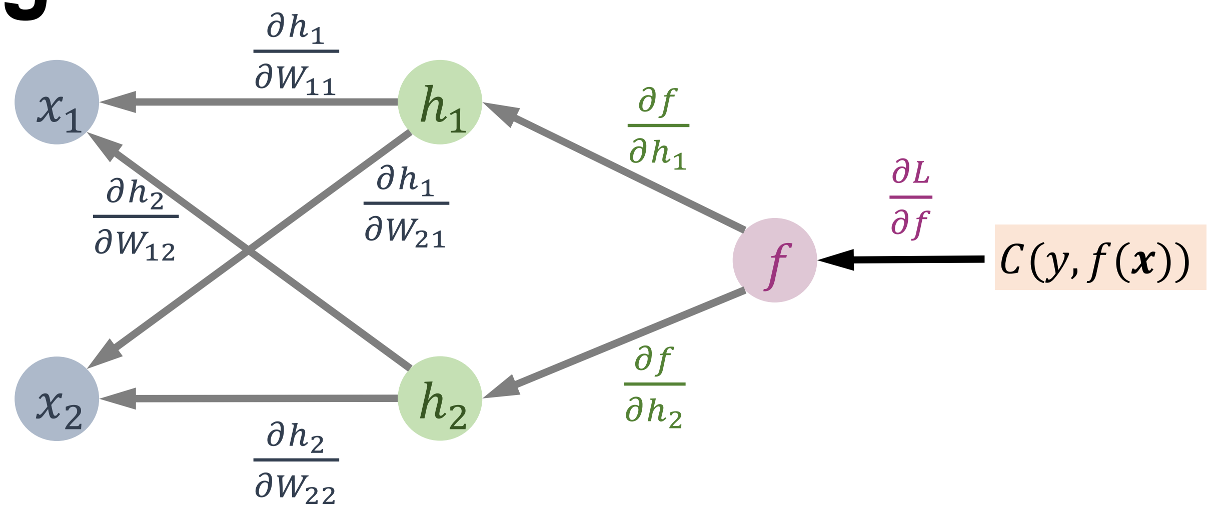
For **each example**, with the **current network parameters**, compute the prediction by **forward-propagating** the inputs through the network

- hidden layer values depend on the input layer: e.g., $h_1 = \sigma(W_{11}x_1 + W_{21}x_2) = \sigma(\boldsymbol{w}_1^T \boldsymbol{x})$
- output layer values depend on the hidden layer: $f = u_1 h_1 + u_2 h_2$
- activation function is sigmoid, $\sigma(z) = \frac{1}{1+e^{-x}}$

Use the **squared loss (cost)** to evaluate the prediction

$$C\left(y_{\text{true}}, y_{\text{pred}}\right) = \frac{1}{2}(y - f(\boldsymbol{x}))^2 = \frac{1}{2}(y - \boldsymbol{u}^T \sigma(W\boldsymbol{x}))^2$$

Backpropagation: Chain Rule

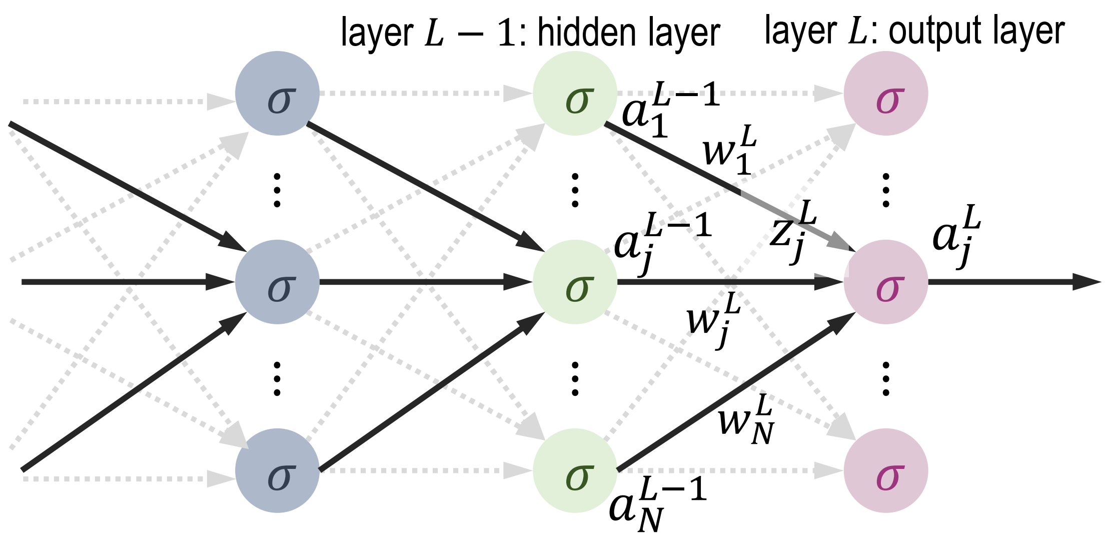

x_1
x_2
h_1
h_2
f
$C(y, f(\boldsymbol{x}))$
$\frac{\partial h_1}{\partial W_{11}}$
$\frac{\partial h_2}{\partial W_{12}}$
$\frac{\partial h_1}{\partial W_{21}}$
$\frac{\partial h_2}{\partial W_{22}}$
$\frac{\partial f}{\partial h_1}$
$\frac{\partial f}{\partial h_2}$
$\frac{\partial L}{\partial f}$

- **sigmoid functions** have a nice property, $\frac{\partial}{\partial z}\sigma(z) = \sigma(z)(1-\sigma(z))$
- we can **chain derivatives** to compute gradients e.g.,

$$\frac{\partial C}{\partial W_{11}} = \frac{\partial C}{\partial f} \cdot \frac{\partial f}{\partial h_1} \cdot \frac{\partial h_1}{\partial W_{11}}$$

$$= -(y - f(\boldsymbol{x})) \cdot \sigma(\boldsymbol{w}_1^T \boldsymbol{x})(1 - \sigma(\boldsymbol{w}_1^T \boldsymbol{x})) \cdot x_1$$

Backpropagation: Multiple Layers



loss at the j-th output node:

$$C = \frac{1}{2}\left(y_j - a_j^L\right)^2$$

z_j^L: **input** to the j-th neuron in the L-th layer

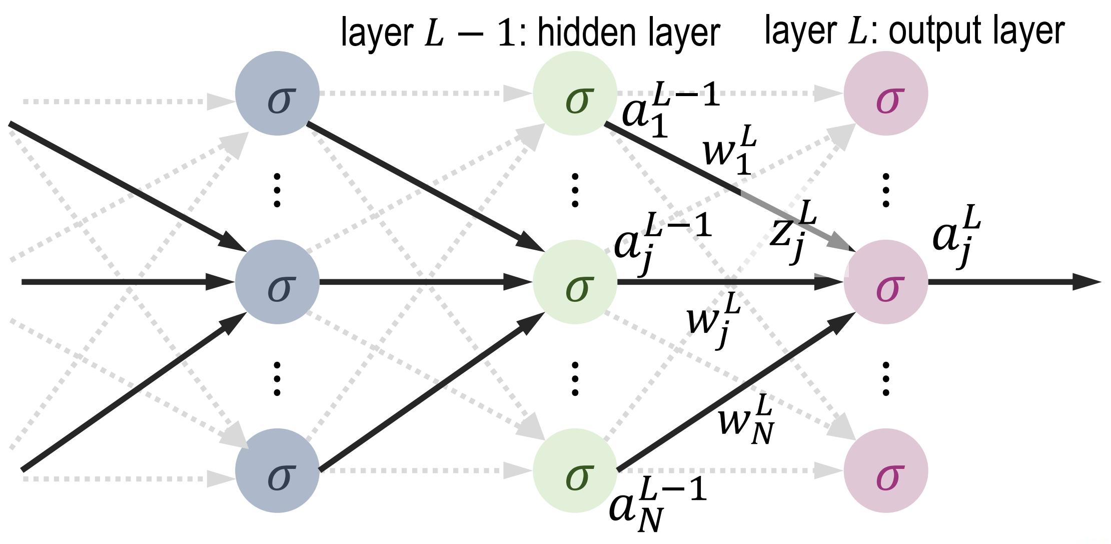
$$z_j^L = \sum_j w_j^L a_j^{L-1}$$

a_j^L: **output** of the j-th neuron in the L-th layer

$$a_j^L = \sigma(z_j^L)$$

$$\begin{aligned}\frac{\partial C}{dz_j^L} &= -\left(y_j - a_j^L\right)\frac{\partial a_j^L}{\partial z_j^L} \\ &= -\left(y_j - a_j^L\right)\frac{\partial \sigma\left(z_j^L\right)}{\partial z_j^L} \\ &= -\left(y_j - a_j^L\right)\sigma\left(z_j^L\right)\left(1-\sigma\left(z_j^L\right)\right) \\ &= \delta_j^L\end{aligned}$$

Backpropagation: Multiple Layers



z_j^L: **input** to the j-th neuron in the L-th layer

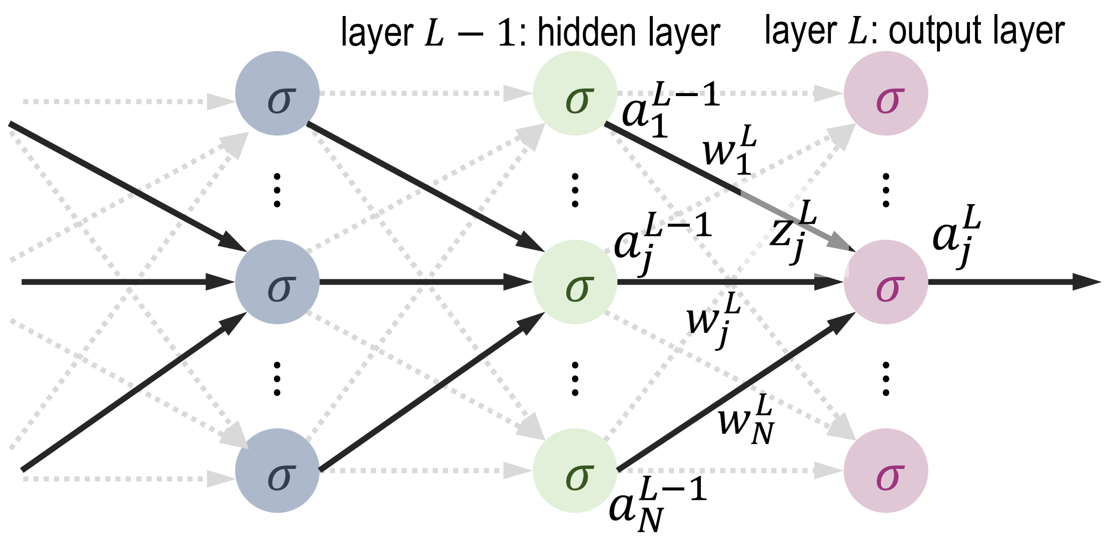
$$z_j^L = \sum_j w_j^L a_j^{L-1}$$

a_j^L: **output** of the j-th neuron in the L-th layer

$$a_j^L = \sigma(z_j^L)$$

$$\begin{aligned}
\frac{\partial C}{dz_j^{L-1}} &= \sum_j \left(a_j^L - y_j\right) \frac{\partial a_j^L}{\partial z_k^{L-1}} \\
&= \sum_j \left(a_j^L - y_j\right) \frac{\partial \sigma\left(z_j^L\right)}{\partial z_k^{L-1}} \\
&= \sum_j \left(a_j^L - y_j\right) \sigma\left(z_j^L\right)\left(1-\sigma\left(z_j^L\right)\right) \frac{\partial z_j^L}{\partial z_k^{L-1}} \\
&= \sum_j \left(a_j^L - y_j\right) \sigma\left(z_j^L\right)\left(1-\sigma\left(z_j^L\right)\right) \frac{\partial \sum_{k'} w_{jk'}^L a_{k'}^{L-1} + b_j^L}{\partial z_k^{L-1}} \\
&= \sum_j \left(a_j^L - y_j\right) \sigma\left(z_j^L\right)\left(1-\sigma\left(z_j^L\right)\right) \sigma\left(z_k^{L-1}\right)\left(1-\sigma\left(z_k^{L-1}\right)\right) w_{jk}^L \\
&= \left(\left(\delta^L\right)^T w_{*k}^L\right)\left(1-\sigma\left(z_k^{L-1}\right)\right)\sigma\left(z_k^{L-1}\right)
\end{aligned}$$

Backpropagation: Multiple Layers



Can use stochastic gradient descent to update gradients one example at a time!

We can compute these derivatives one layer at a time

$$\frac{\partial C}{dz_j^{L-1}} = \delta^{L-1} = \left((\delta^L)^T w^L\right)\left(1-\sigma(z^{L-1})\right)\sigma(z^{L-1})$$

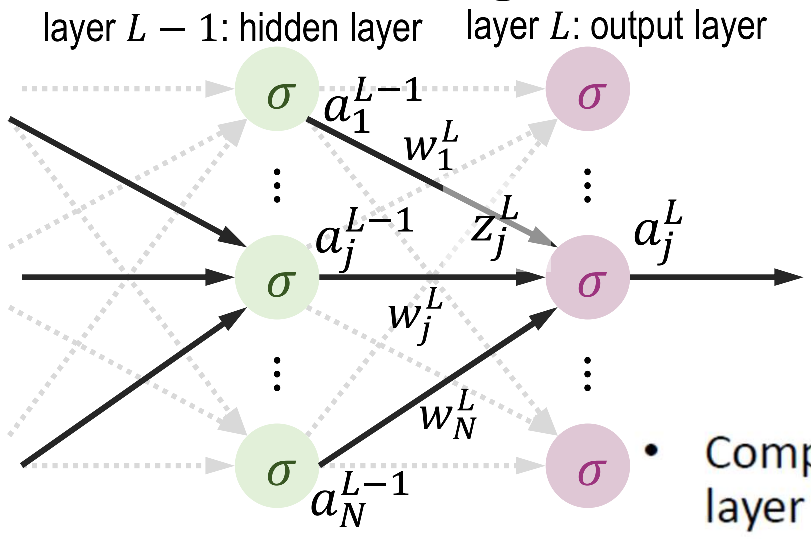
$$\delta^l = \left((\delta^{l+1})^T w^{l+1}\right)\left(1-\sigma(z^l)\right)\sigma(z^l)$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l$$

bias term is implicit in each node

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$$

Backpropagation



- Backpropagation converges to a **local minimum** (loss is not convex in the weights and biases)
- Like EM, can just run it several times with **different initializations**
- Training can take a **very long time**
  - even with stochastic gradient descent
- **Prediction after learning is fast**
- Sometimes include a **momentum** term α in the gradient update

$$w(t) = w(t-1) - \gamma \cdot \nabla_w C(t-1) + \alpha(-\gamma \cdot \nabla_w C(t-2))$$

- Compute the inputs/outputs for each layer by starting at the input layer and applying the sigmoid functions
- Compute δ^L for the output layer

$$\delta^L = -(y_j - a_j^L)\,\sigma(z_j^L)\left(1-\sigma(z_j^L)\right)$$

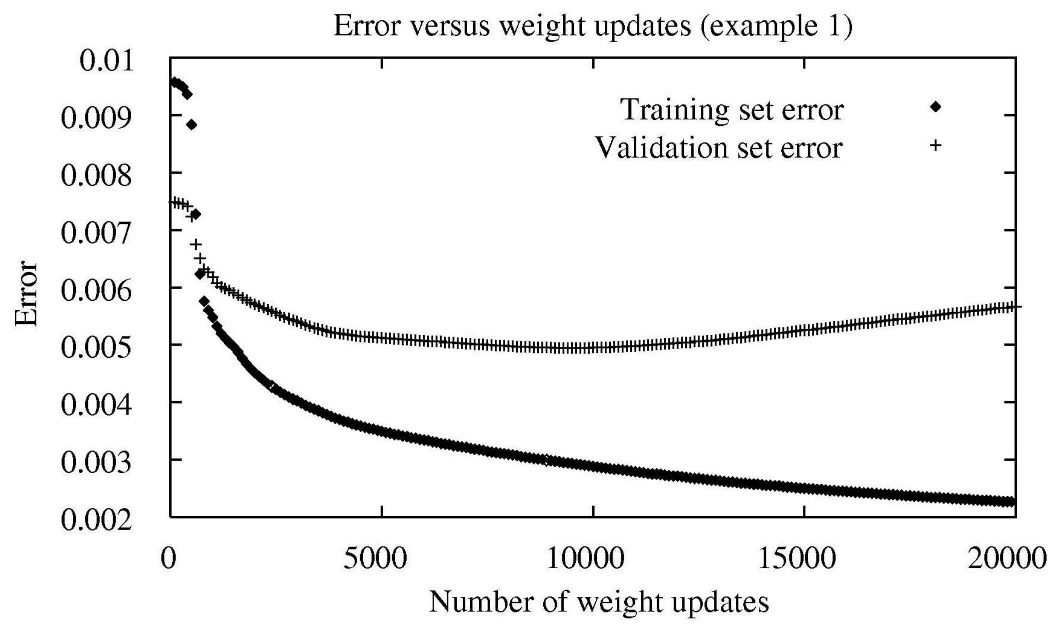
- Starting from $l = L-1$ and working backwards, compute

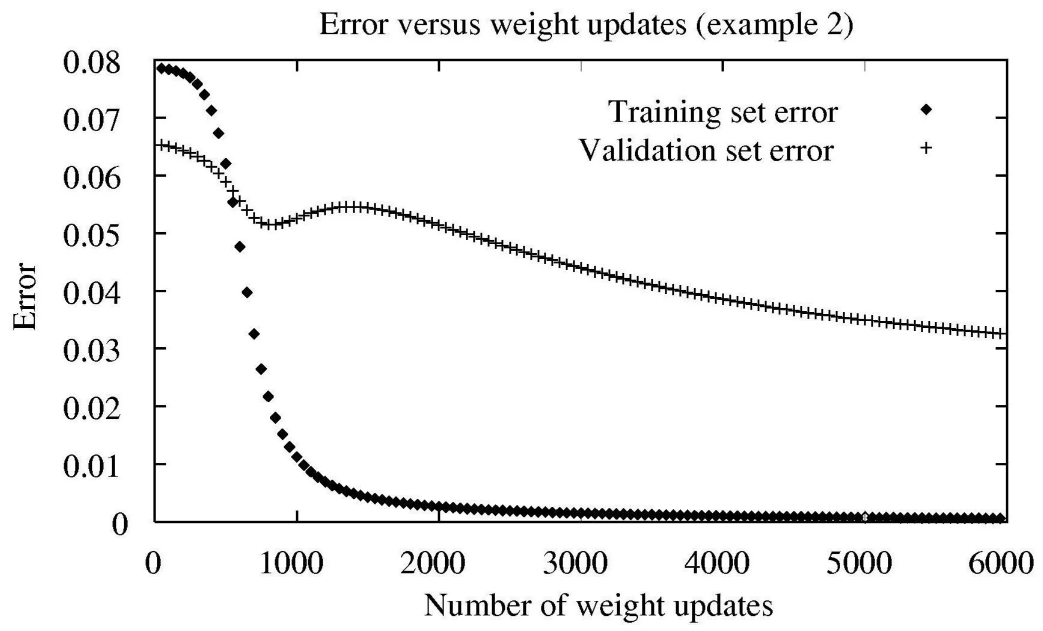
$$\delta^l = \left((\delta^{l+1})^T w^{l+1}\right)\sigma(z^l)\left(1-\sigma(z^l)\right)$$

- Perform gradient descent

$$b_j^l = b_j^l - \gamma \cdot \delta_j^l$$
$$w_{jk}^l = w_{jk}^l - \gamma \cdot \delta_j^l a_k^{l-1}$$

Overfitting





Neural Networks in Practice

Many ways to improve weight learning in NNs

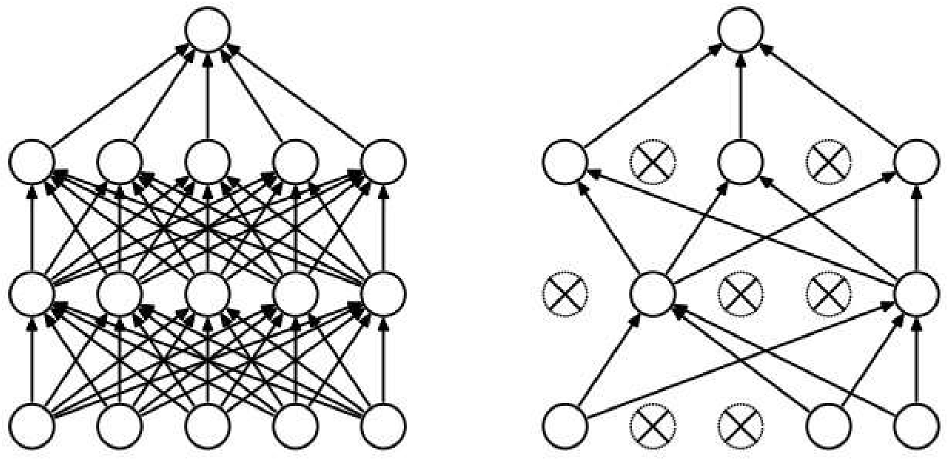
- Use **regularized squared loss (cost)** prediction (can still use backpropagation in this setting)

$$C(y_{\text{true}}, y_{\text{pred}}) = \frac{1}{2}(y - f(\boldsymbol{x}; \boldsymbol{w}, b))^2 + \frac{\lambda}{2}\|w\|_2^2$$

  - L_1 regularization can also be useful
  - $\lambda > 0$ should be chosen with a **validation set**
- Try other loss functions, e.g., the **cross entropy**
  - $C(y_{\text{true}}, y_{\text{pred}}) - y \log f(\boldsymbol{x}) - (1 - y) \log(1 - f(x))$
- **Initialize weights** of the network more cleverly
  - Random initializations are likely to be far from optimal
- Learning procedure can have **numerical difficulties** if there are a **large number of layers**
  - **Early stopping**: stop the learning early in the hopes that this prevents overfitting

Drop out: A **heuristic bagging-style approach** applied to neural networks to **counteract overfitting**

- **Randomly remove** a certain percentage of **neurons from the network** and then train only on the remaining neurons
- networks **recombined using an approximate averaging**
- keeping around too many networks and doing proper bagging can be costly in practice



(a) Standard Neural Net (b) After applying dropout.

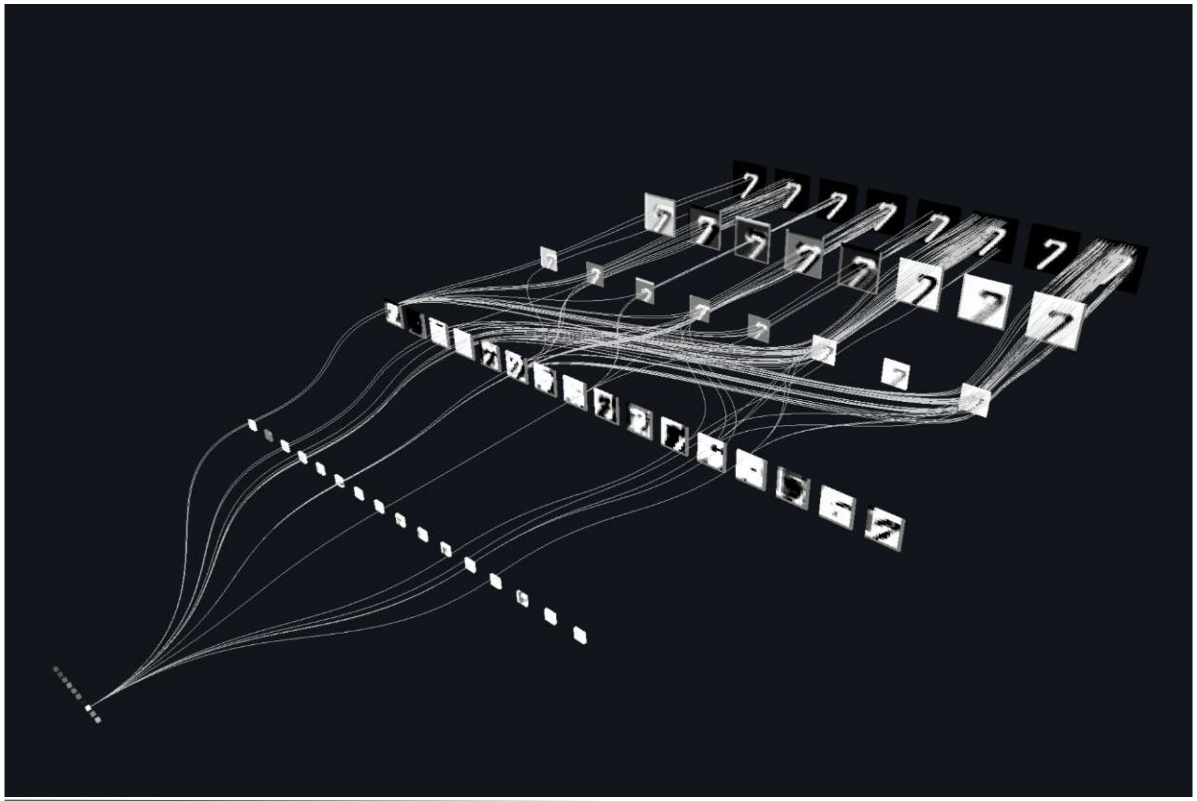
Parameter Tying

Parameter tying: Assume some of the weights in the model are the same to reduce the **dimensionality** of the learning problem;

- Also a way to learn "simpler" models
- Can lead to significant compression in neural networks (i.e., >90%)

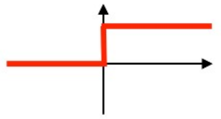
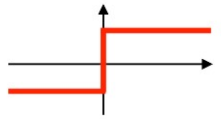
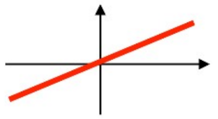
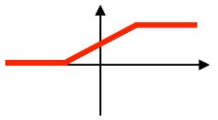
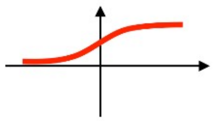
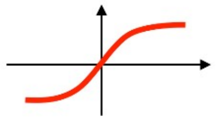
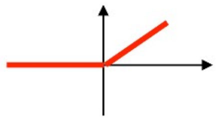
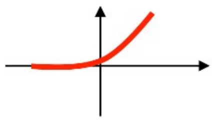
Convolutional neural networks

- Instead of the output of every neuron at layer ℓ being used as an input to every neuron at layer $\ell + 1$, edges between layers are chosen more locally
- Many tied weights and biases
  - convolution nets apply the same process to many different local chunks of neurons
- Often combined with pooling layers
  - layers that replacing small regions of neurons with their aggregated output
- Used extensively for image classification tasks



Topological Visualization of a Convolutional Neural Network by Terence Broad http://terencebroad.com/nnvis.html

Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Example: Self Driving Cars