Lecture 6: Neural Networks

Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

https://shuaili8.github.io

https://shuaili8.github.io/Teaching/VE445/index.html

Last lecture

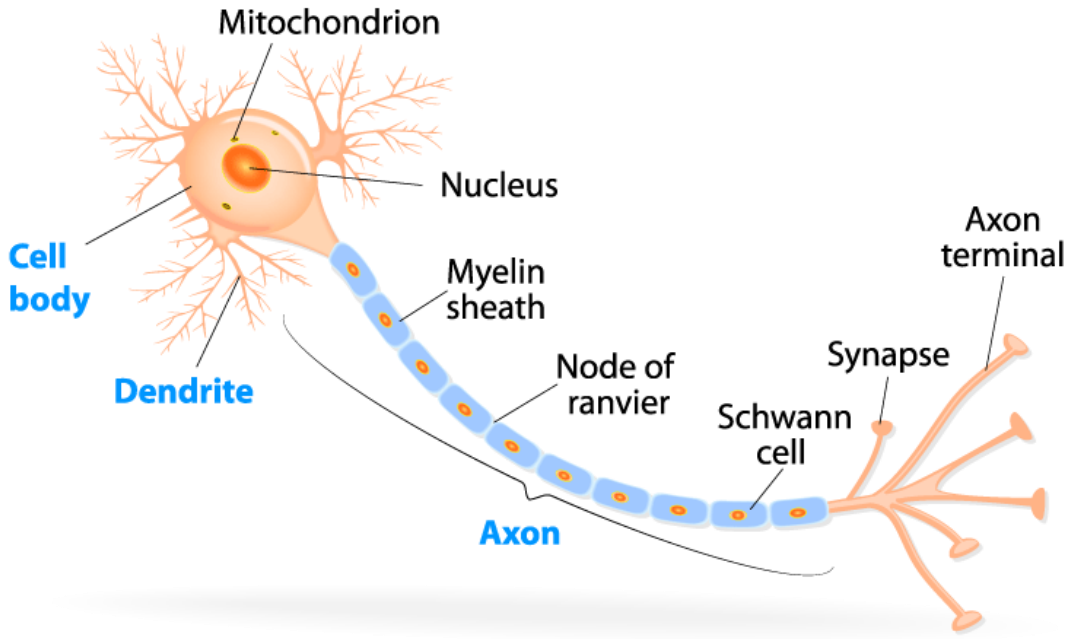
- Linear classifiers and the margins
- Objective of the SVM
- Lagrangian method in convex optimization
- Solve SVM by Lagrangian duality
- Regularization
- Kernel method
- SMO algorithm to solve the Lagrangian multipliers

Today’s lecture

- Perceptron
- Activation functions
- Layers
- Back propagation

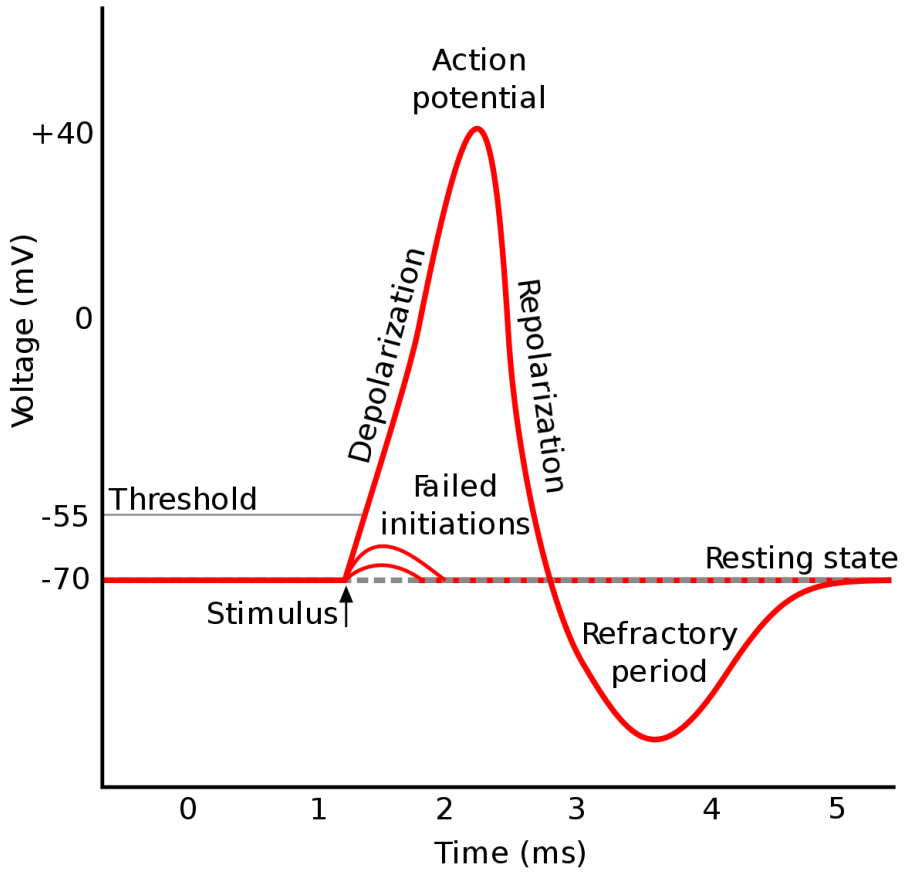
Biological neuron structure

- The neuron receives signals from their dendrites, and send its own signal to the axon terminal



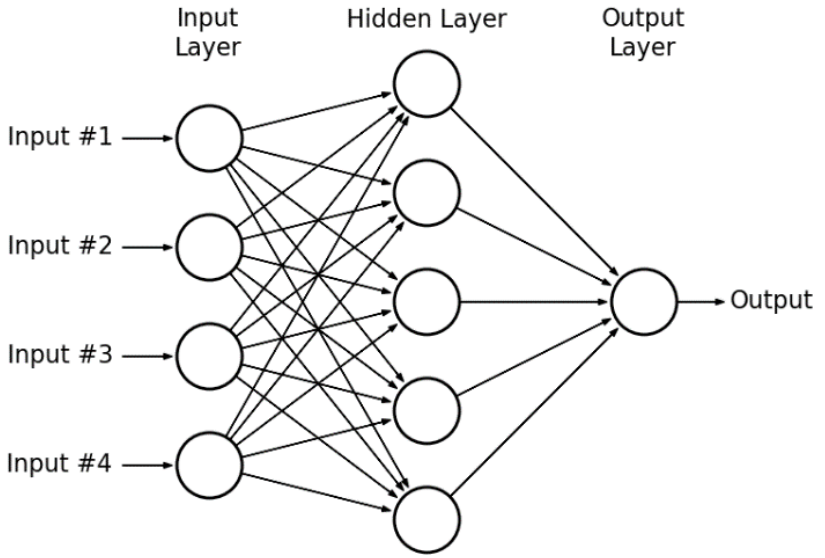
Biological neural communication

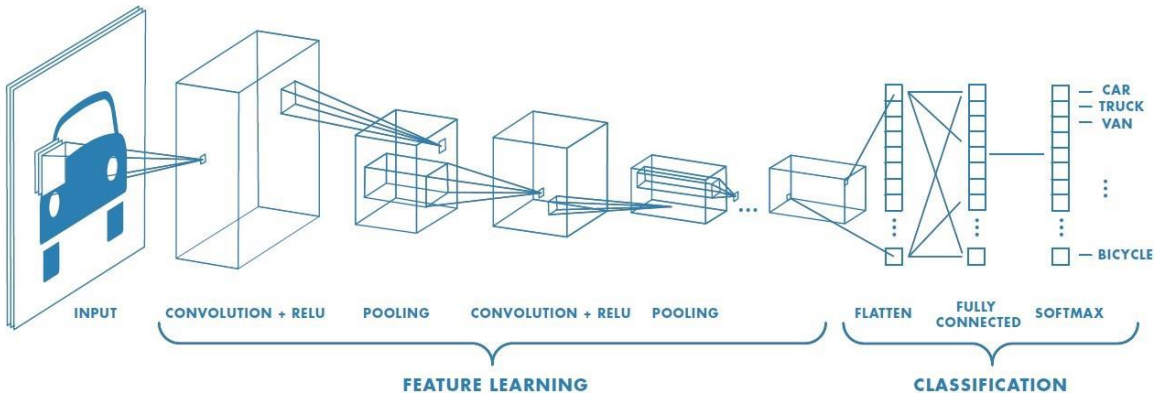
- Electrical potential across cell membrane exhibits spikes called action potentials
- Spike originates in cell body, travels down axon, and causes synaptic terminals to release neurotransmitters
- Chemical diffuses across synapse to dendrites of other neurons
- Neurotransmitters can be excitatory or inhibitory
- If net input of neuro transmitters to a neuron from other neurons is excitatory and exceeds some threshold, it fires an action potential

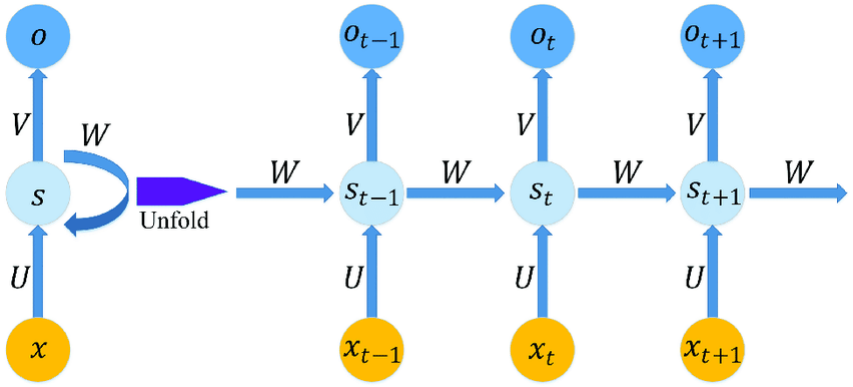


Artificial neural networks

- Multilayer perceptron network
- Convolutional neural network
- Recurrent neural network

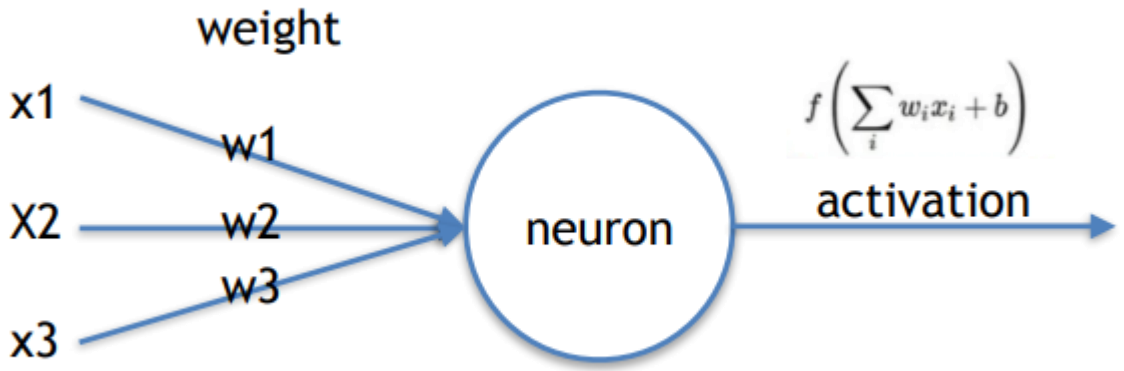






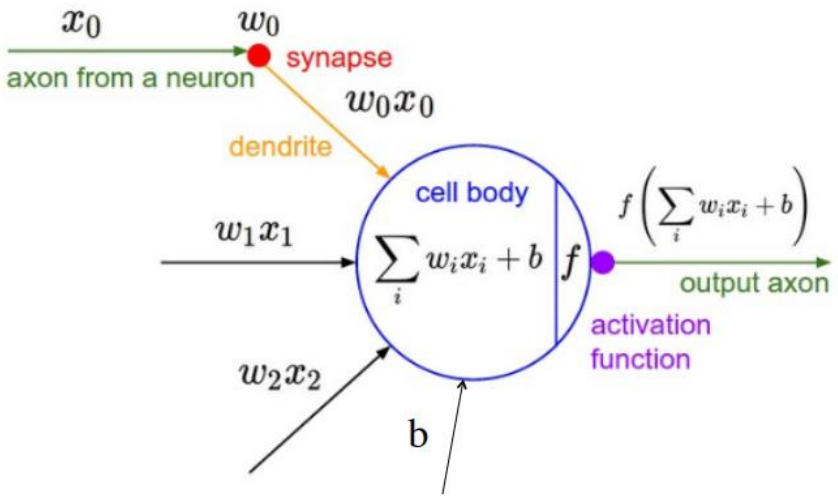
Perceptron

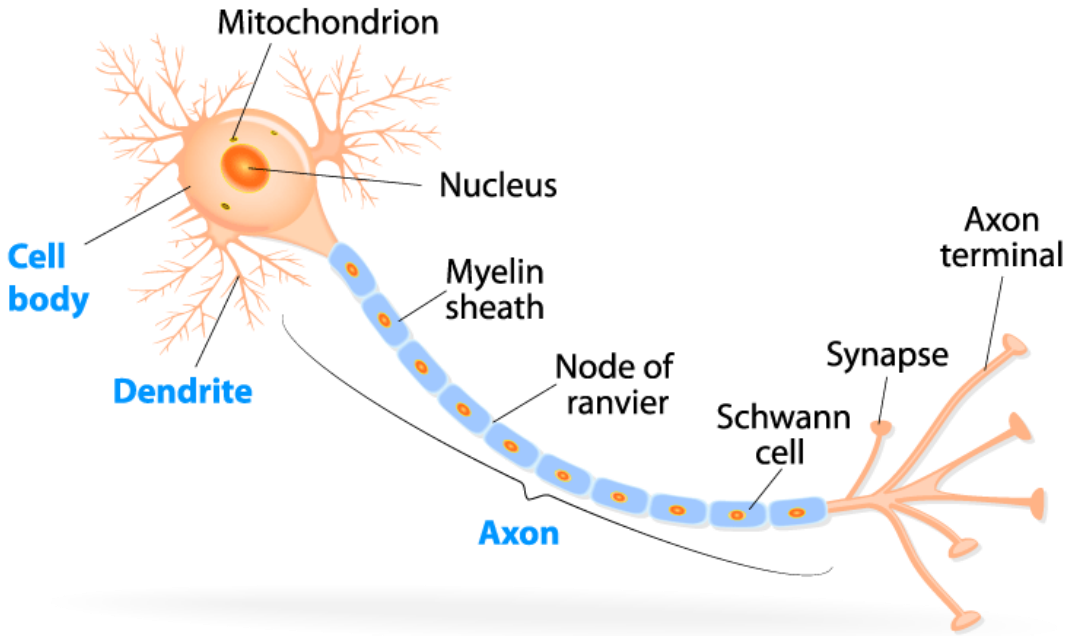
- Inspired by the biological neuron among humans and animals, researchers build a simple model called Perceptron



- It receives signals x_i, multiplies them with different weights w_i, and outputs the sum of the weighted signals after an activation function f

Neuron vs. Perceptron





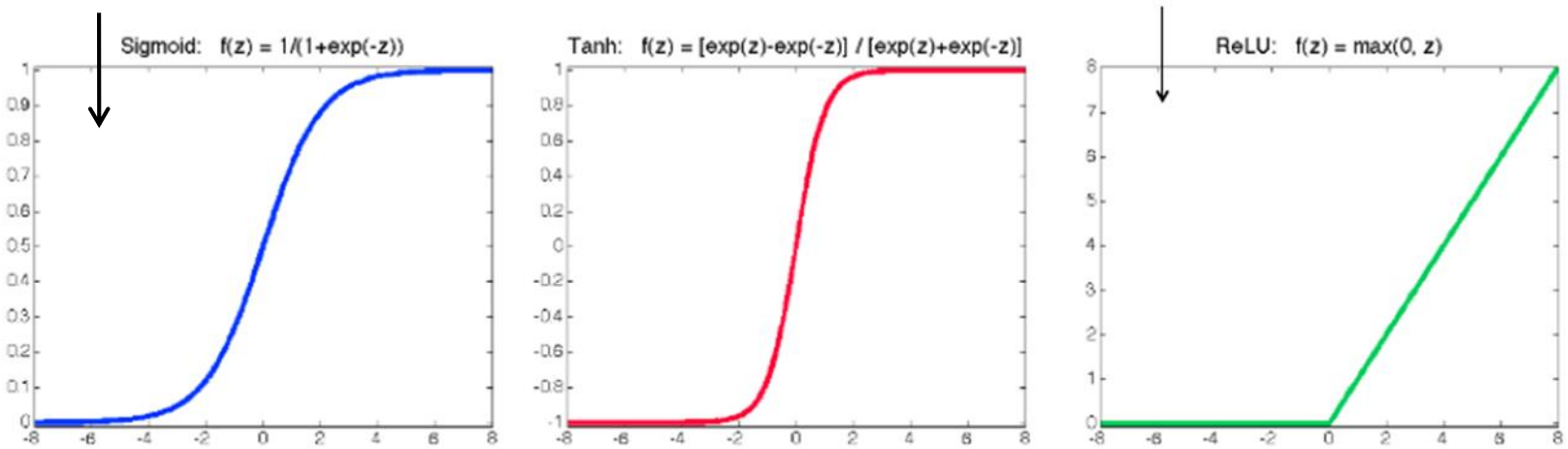
Activation Functions

Activation functions

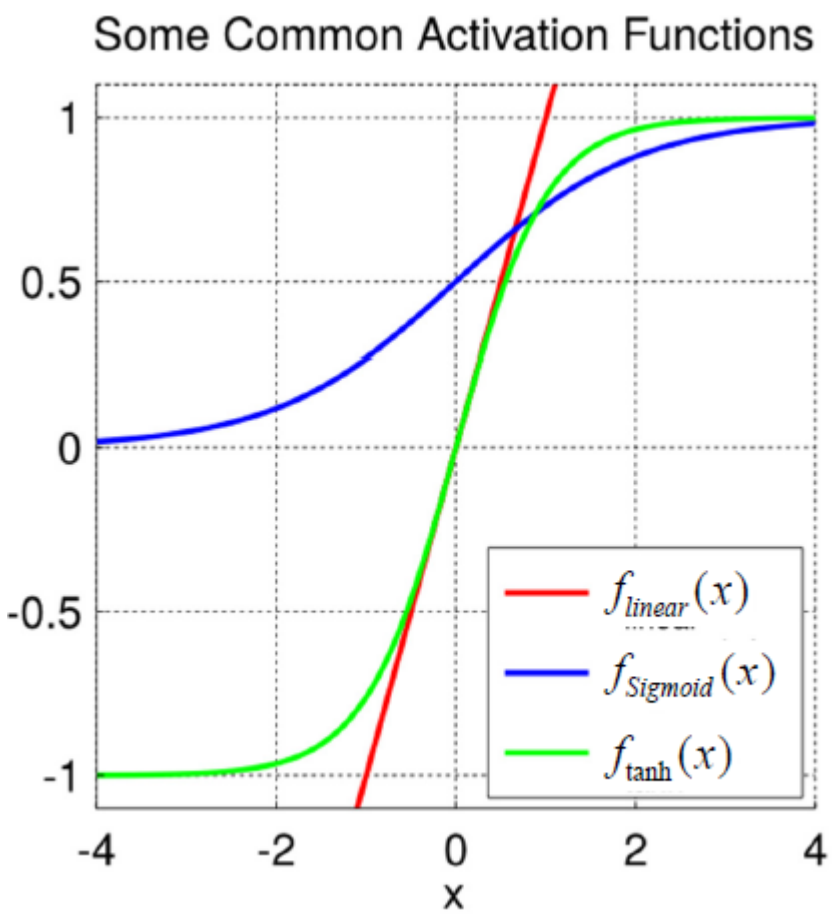
- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
- Tanh: $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- ReLU (Rectified Linear Unity): $\mathrm{ReLU}(z) = \max\ (0, z)$

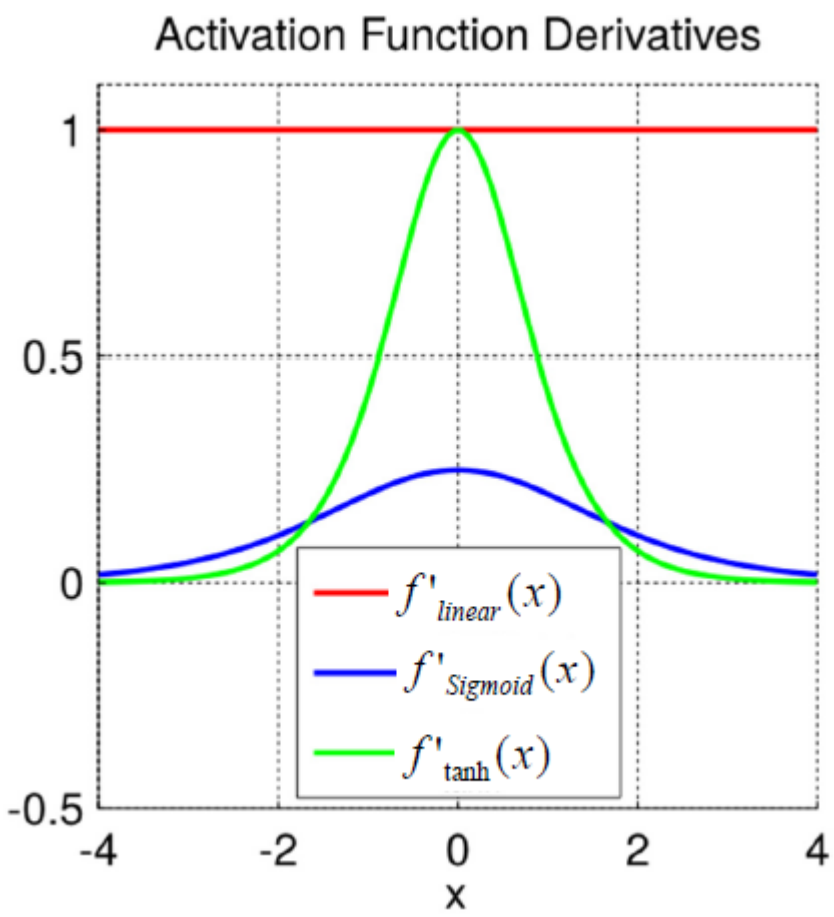
Most popular in fully connected neural network

Most popular in deep learning



Activation function values and derivatives

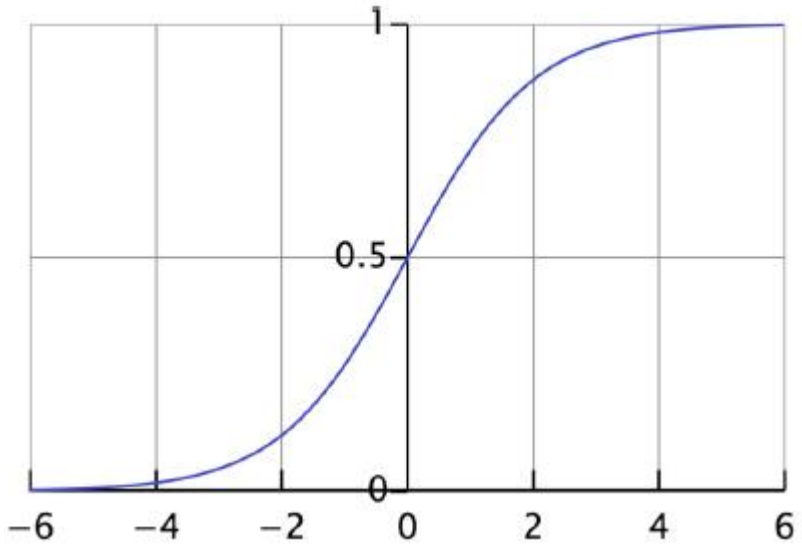




Sigmoid activation function

- Sigmoid

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



- Its derivative

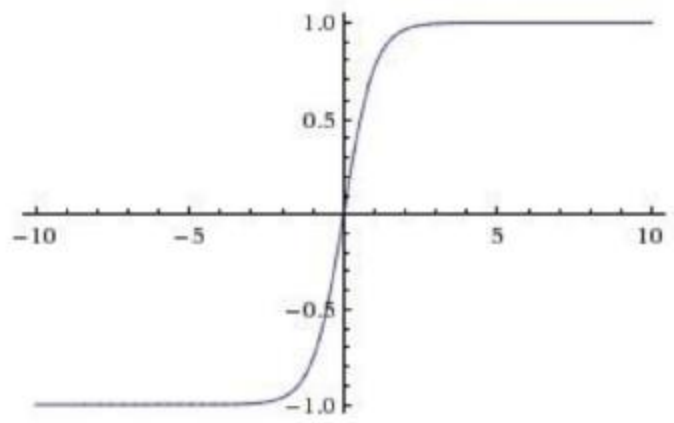
$$f'_{Sigmoid}(x) = f_{Sigmoid}(x)(1 - f_{Sigmoid}(x))$$

- Output range $(0,1)$
- Motivated by biological neurons and can be interpreted as the probability of an artificial neuron “firing” given its inputs
- However, saturated neurons make value vanished **(why?)**
  - $f\left(f\left(f(\cdots)\right)\right)$
  - $f([0,1]) \subseteq [0.5, 0.732)$
  - $f\left([0.5, 0.732)\right) \subseteq (0.378, 0.676)$

Tanh activation function

- Tanh function

$$f_{\tanh}(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



- Its derivative

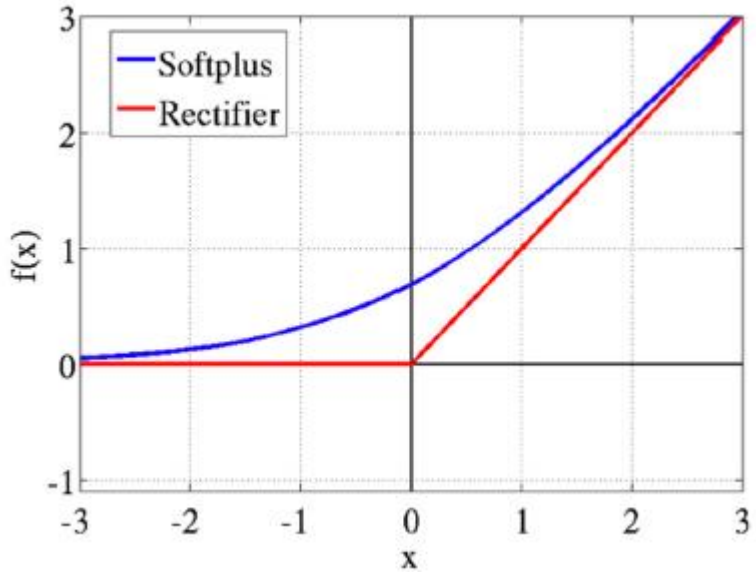
$$f_{\tanh}(x) = 1 - f_{\tanh}(x)^{2}$$

- Output range [-1,1]
- Thus strongly negative inputs to the tanh will map to negative outputs.
- Only zero-valued inputs are mapped to near-zero outputs
- These properties make the network less likely to get “stuck” during training

ReLU activation function

- ReLU function

$$f_{\mathrm{ReLU}}(x) = \max(0, x)$$



- Its derivative

$$f_{\mathrm{ReLU}}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

- Another version is Noise ReLU

$$f_{\mathrm{NoisyReLU}}(x) = \max(0, x + N(0, \delta(x)))$$

- ReLU can be approximated by softplus function

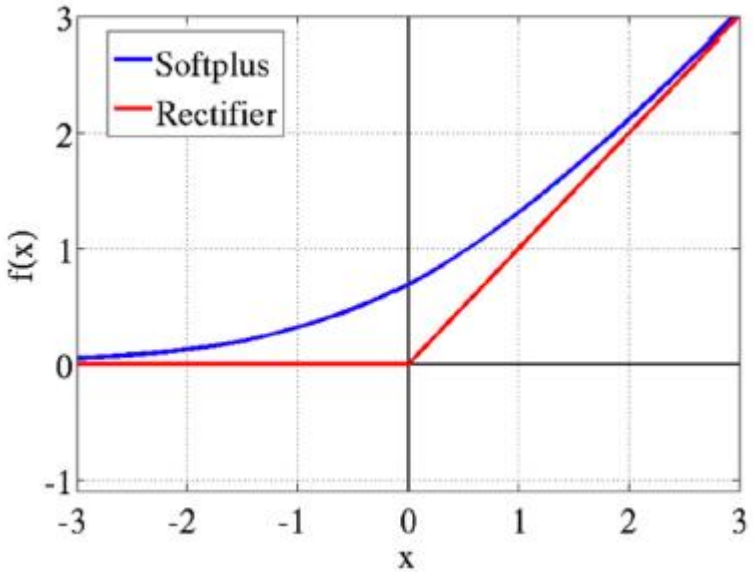
$$f_{\mathrm{Softplus}}(x) = \log(1 + e^{x})$$

- ReLU gradient doesn't vanish as we increase x
- It can be used to model positive number
- It is fast as no need for computing the exponential function
- It eliminates the necessity to have a “pretraining” phase

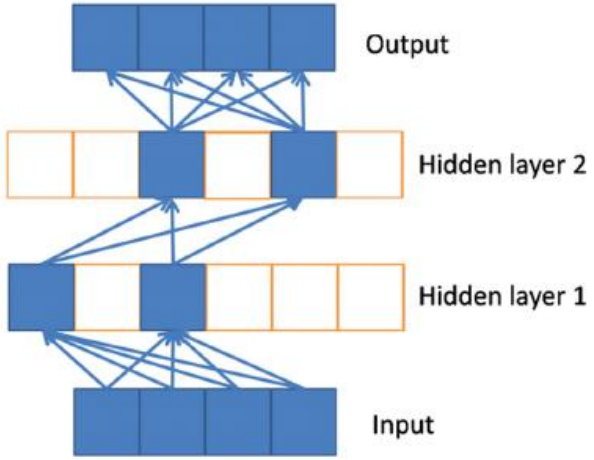
ReLU activation function (cont.)

- ReLU function

$$f_{\mathrm{ReLU}}(x) = \max(0, x)$$



- The only non-linearity comes from the path selection with individual neurons being active or not
- It allows sparse representations:
  - for a given input only a subset of neurons are active



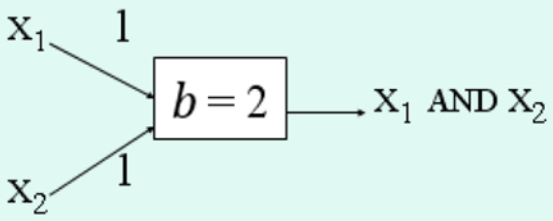
Sparse propagation of activations and gradients

Multilayer Perceptron Networks

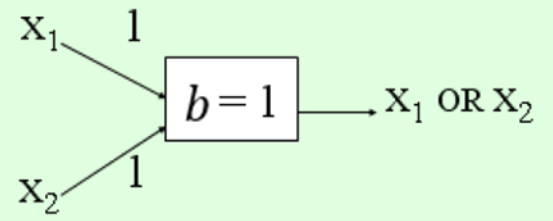
Limitation of single perception

- Single perception can distinguish the linearly separable data. For example, it can simulate basic logical gate AND and OR with proper weights

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

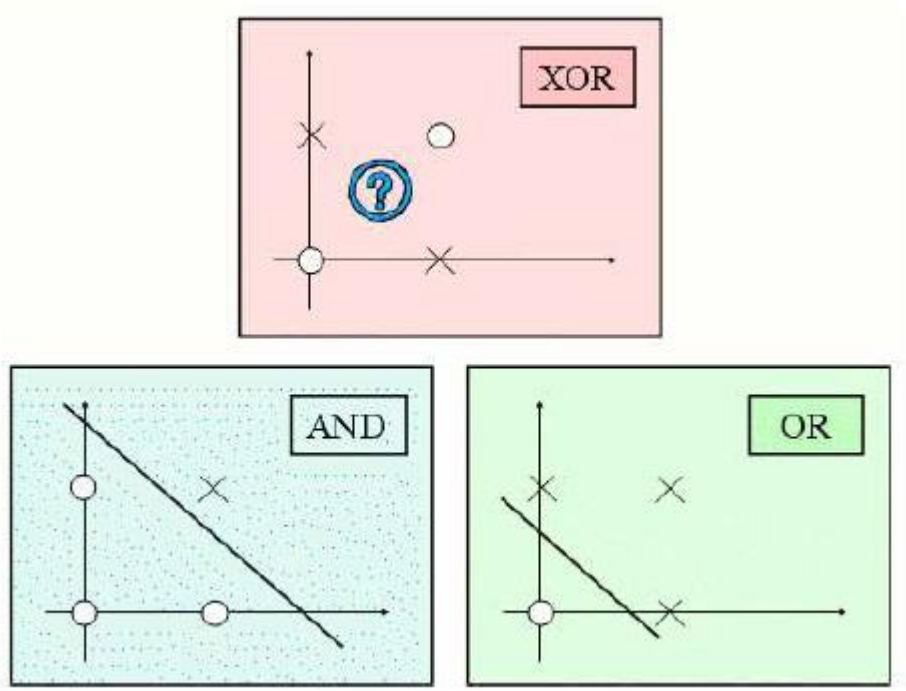


x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1



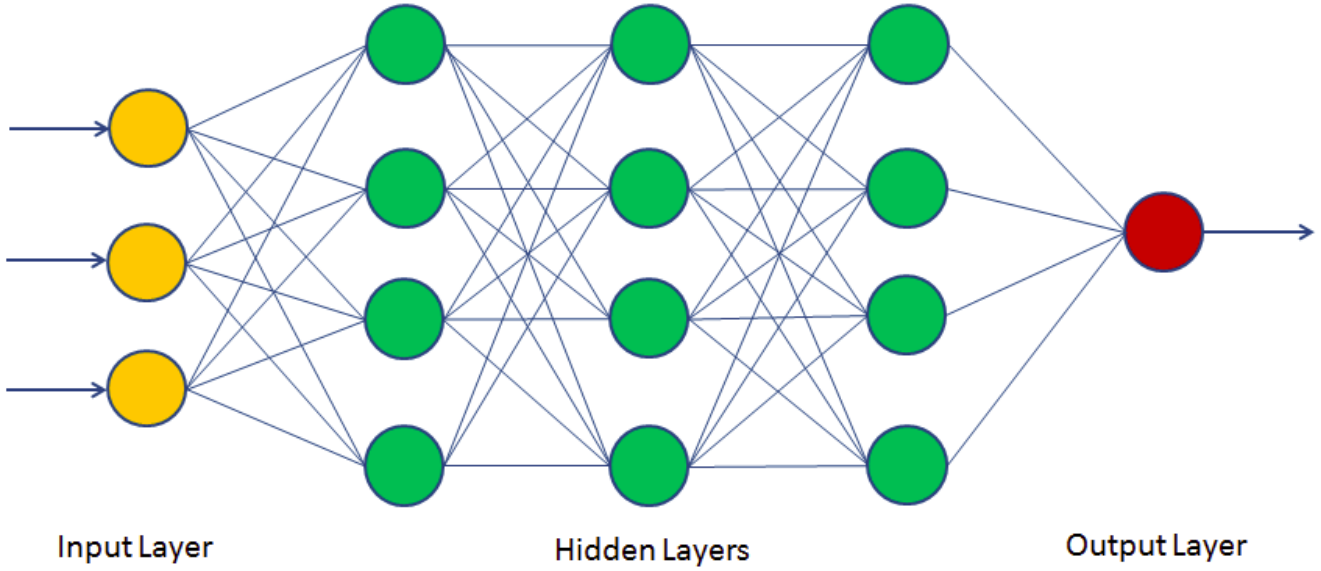
Limitation of single perception

- Single perception cannot distinguish linearly non-separable data, such as the XOR gate.



Neuron networks

- The solution to solve the limitation is connecting many perceptions together, layer by layer.



- And this is called neural network model

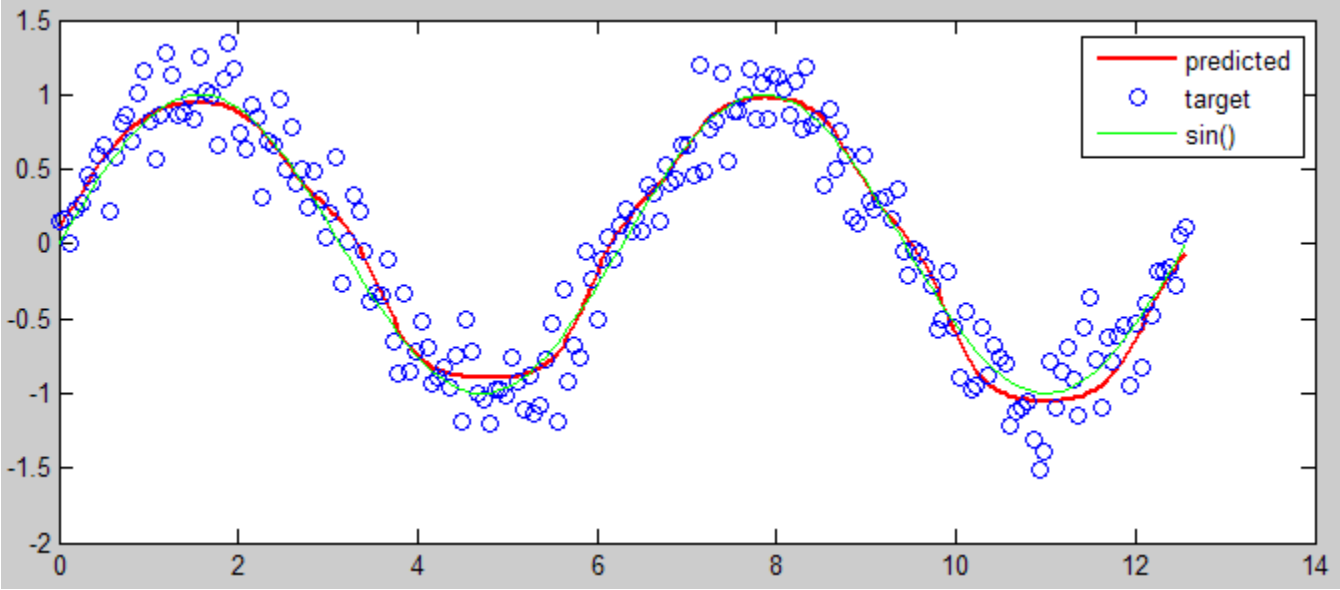
Neural network

- The there two important questions need to be answered.

1. What does the neural network do?
2. How can we train it?

What does the neural network do?

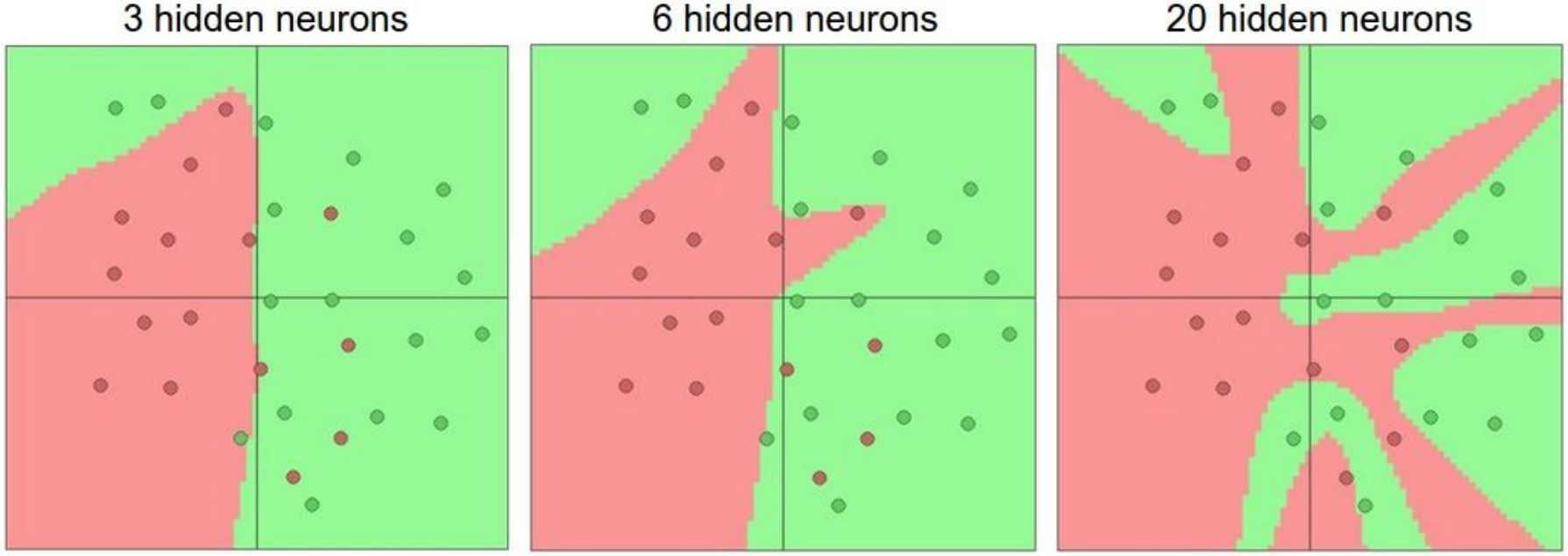
- For each sample, the predicted value of neural network model will approximate the target function gradually.



- Neural network is a kind of function approximation model.

What does the neural network do?

- With the increment in neural network layers, its approximation power increases.



- The decision boundary covers more details when the net goes deeper

How can we train it?

- As previous models, we use gradient descent method to train neural network

- Given the topology of the network (number of layers, number of neurons, their connections), find a set of **weights** to minimize the error function.

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

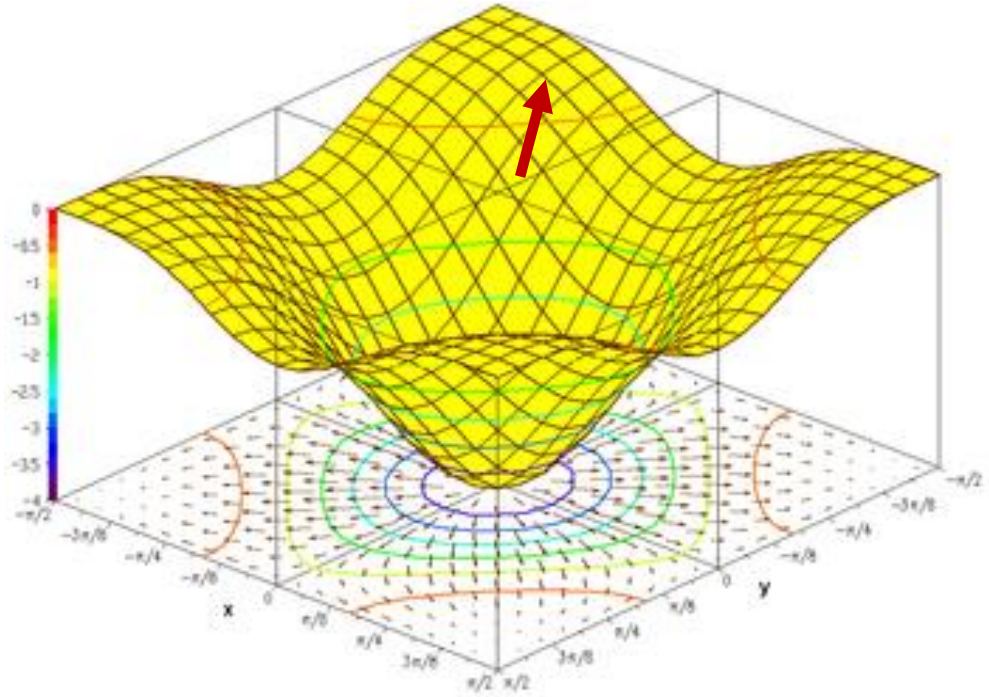
The set of training examples

Target

Output

Gradient interpretation

- Gradient is the vector (the red one) along which the value of the function increases most rapidly. Thus its opposite direction is where the value decreases most rapidly.

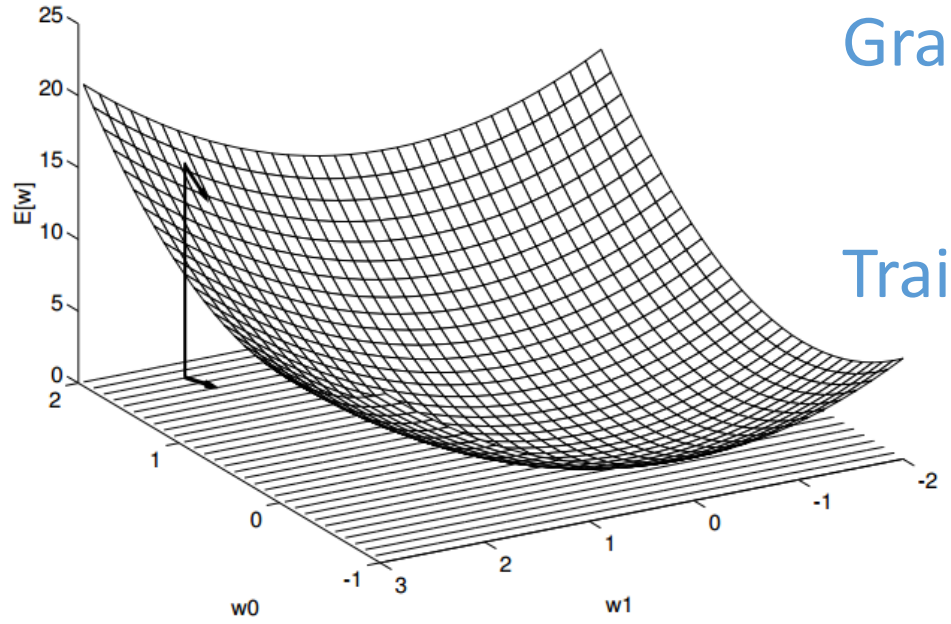


Gradient Descent

- To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point.

- For a smooth function $f(x)$, $\frac{\partial f}{\partial x}$ is the direction that f increases most rapidly. So we apply $x_{t+1} = x_t - \lambda \frac{\partial f}{\partial x}$ until x converges.

Gradient Descent



Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training Rule

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$ → Partial derivatives are the key

Update

$$w_i \leftarrow w_i + \Delta w_i$$

The Chain Rule

- The challenge in neural network model is that we only know the target of the output layer, and don't know the target for hidden and input layers, how can we update their connection weights using gradient descent?

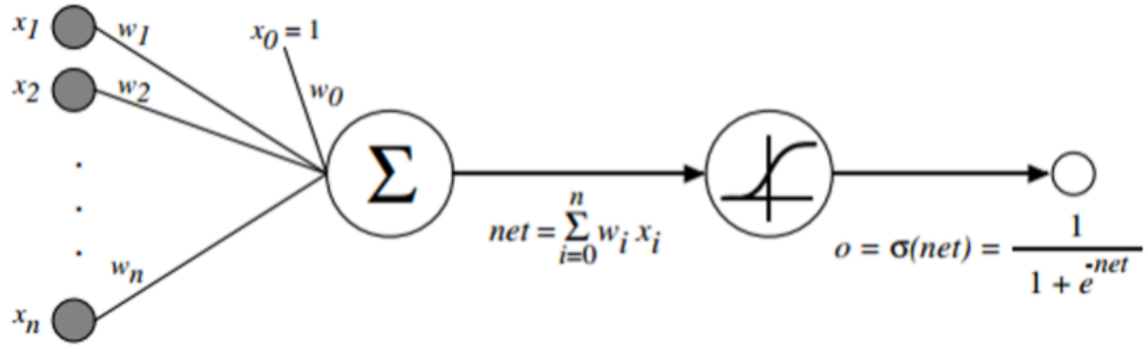
- The answer is the chain rule that you have learned in calculus.

$$y = f(g(x))$$

$$\Rightarrow \frac{dy}{dx} = f'(g(x))g'(x)$$

Output Neuron

- For the output layer, we have:



Input x ➡ Net input $\sum w_i x_i$ ➡ output $= \frac{1}{1+e^{-net}}$ ➡ Error $E = \frac{1}{2}(t - output)^2$

$$\frac{\partial Err}{\partial w_i} = \frac{\partial Err}{\partial out} * \frac{\partial out}{\partial net} * \frac{\partial net}{\partial w_i}$$

Output neuron

Expression	Derivative
$Err = \frac{1}{2}(t - output)^2$	$\frac{\partial Err}{\partial out} = (t - out)$
$out = \frac{1}{1 + e^{-net}}$	$\frac{\partial out}{\partial net} = \frac{e^{-net}}{(1+e^{-net})^2} = out(1 - out)$
$net = \sum w_i x_i$	$\frac{\partial net}{\partial w_i} = x_i$

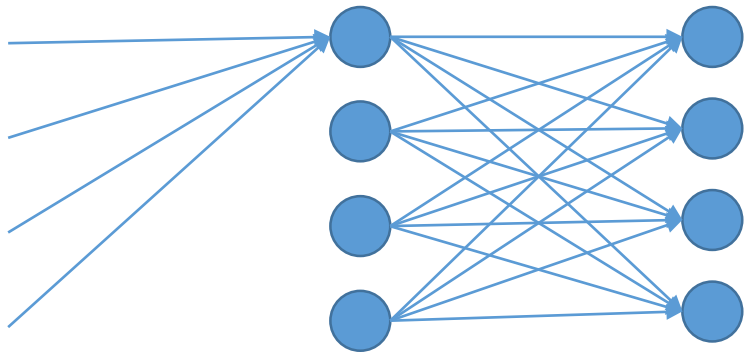
- Finally, we have $\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$

Hidden layer

- Suppose we have updated the input weight for the neuron in output layer using the following equation:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

- What about the hidden layer? More generally, suppose we have updated the input weights for a layer, how to update the input weights of previous layer?



How?

updated

Hidden layer

- Let k denote the neuron in next layer, and j denote the neuron in current layer. Let $\delta_k = -\frac{\partial E}{\partial net_k}$
- We have

$$\begin{aligned}
\frac{\partial E}{\partial net_j} &= \sum_{k \in Outs(j)} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial net_j} && \text{Chain rule} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial net_j} && \text{replacement} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} && \text{Chain rule} \\
&= \sum_{k \in Outs(j)} -\delta_k \, w_{kj} \frac{\partial o_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)
\end{aligned}$$

- Where $\delta_j = -\frac{\partial E}{\partial net_j} = o_j(1 - o_j) \sum_{k \in Outs(j)} \delta_k \, w_{kj}$

Backpropagation algorithms

Initialize all weights to small random numbers

Do until convergence

- For each training example:
  1. Input it to network and compute the network output
  2. For each output unit k

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

  3. For each hidden unit h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

  4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

Where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

More on back propagations

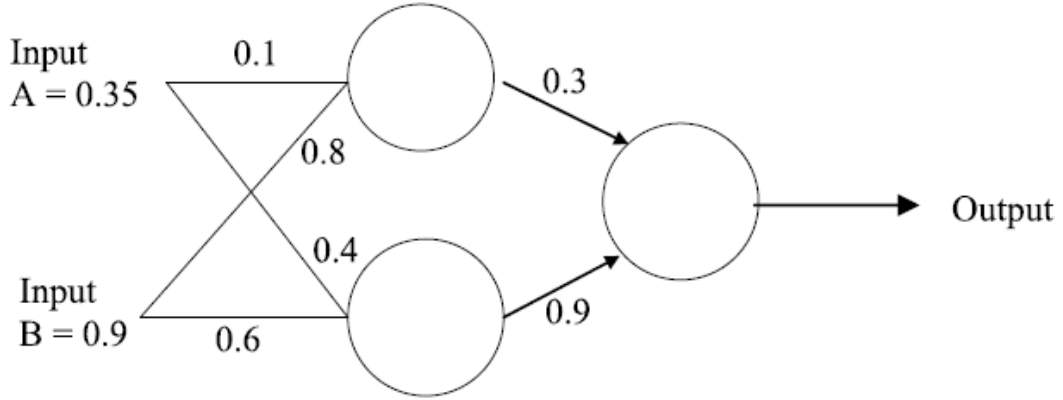
- Gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often work well (can run multiple times)
- Often include weight momentum α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations, slow!
- Using network after training is very fast.

Calculation exercise

- Consider the simple network below:



- Assume that the neurons have sigmoid activation function and
  - Perform a forward pass on the network
  - Perform a reverse pass (training) once (target = 0.5)
  - Perform a further forward pass and comment on the result

Calculation exercise

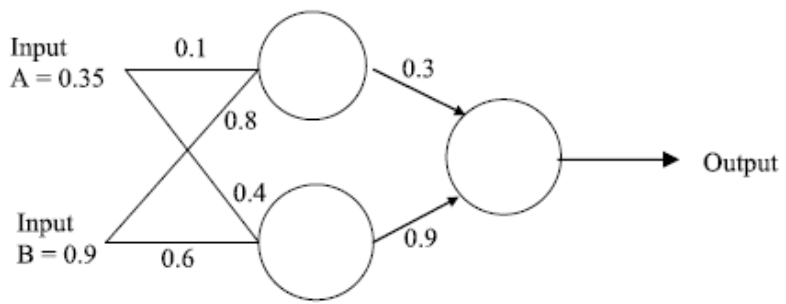
Answer:
(i)
Input to top neuron = (0.35x0.1)+(0.9x0.8)=0.755. Out = 0.68.
Input to bottom neuron = (0.9x0.6)+(0.35x0.4) = 0.68. Out = 0.6637.
Input to final neuron = (0.3x0.68)+(0.9x0.6637) = 0.80133. Out = 0.69.

(ii)
Output error $\delta=(t-o)(1-o)o = (0.5-0.69)(1-0.69)0.69 = -0.0406$.

New weights for output layer
$w1^+ = w1+(\delta \times input) = 0.3 + (-0.0406x0.68) = 0.272392$.
$w2^+ = w2+(\delta \times input) = 0.9 + (-0.0406x0.6637) = 0.87305$.

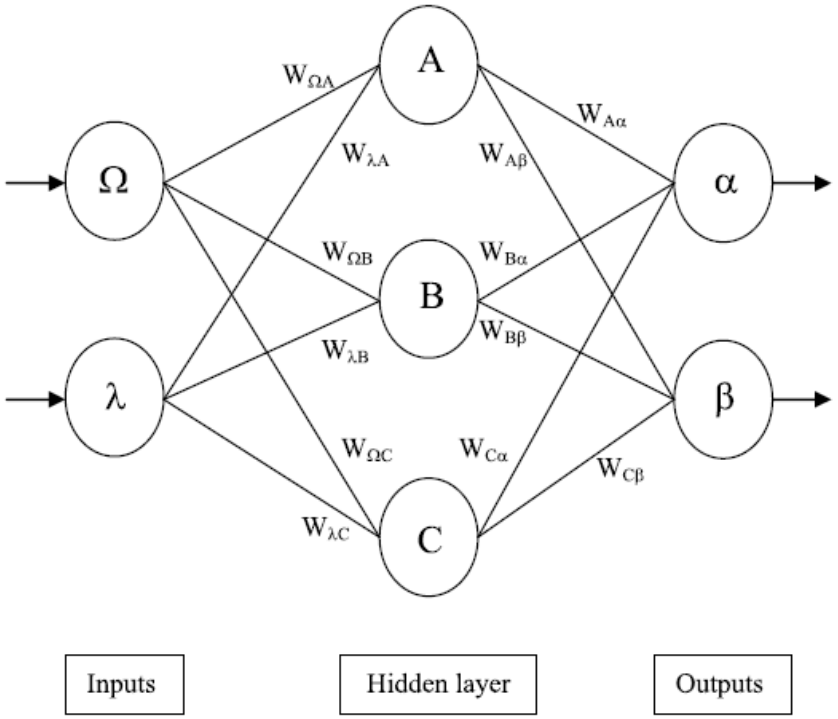
Errors for hidden layers:
$\delta1 = \delta \times w1 = -0.0406 \times 0.272392 \times (1-o)o = -2.406x10^{-3}$
$\delta2 = \delta \times w2 = -0.0406 \times 0.87305 \times (1-o)o = -7.916x10^{-3}$

New hidden layer weights:
$w3^+=0.1 + (-2.406 \times 10^{-3} \times 0.35) = 0.09916$.
$w4^+ = 0.8 + (-2.406 \times 10^{-3} \times 0.9) = 0.7978$.
$w5^+ = 0.4 + (-7.916 \times 10^{-3} \times 0.35) = 0.3972$.
$w6^+ = 0.6 + (-7.916 \times 10^{-3} \times 0.9) = 0.5928$.

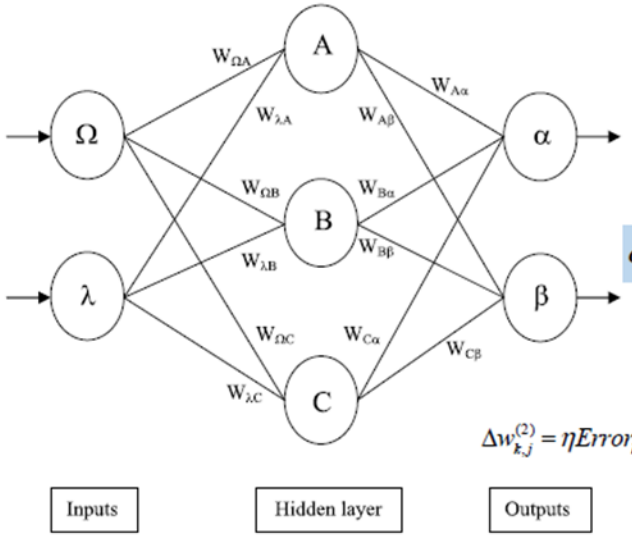


(iii)
Old error was -0.19. New error is -0.18205. Therefore error has reduced.

An example for Backprop



An example for Backprop



1. Calculate errors of output neurons

$$\delta_k = (d_k - y_k)\ f_{(2)}'(net_k^{(2)})$$

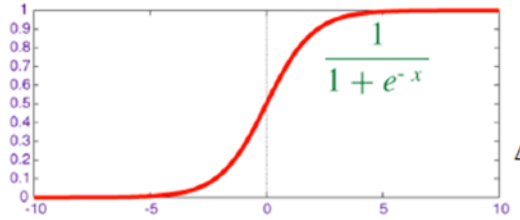
$$\delta_\alpha = out_\alpha\ (1 - out_\alpha)\ (Target_\alpha - out_\alpha)$$
$$\delta_\beta = out_\beta\ (1 - out_\beta)\ (Target_\beta - out_\beta)$$

2. Change output layer weights

$$\Delta w_{k,j}^{(2)} = \eta Error_k Output_j = \eta \delta_k h_j^{(1)}$$

$$W^+_{A\alpha} = W_{A\alpha} + \eta\delta_\alpha\, out_A \qquad W^+_{A\beta} = W_{A\beta} + \eta\delta_\beta\, out_A$$
$$W^+_{B\alpha} = W_{B\alpha} + \eta\delta_\alpha\, out_B \qquad W^+_{B\beta} = W_{B\beta} + \eta\delta_\beta\, out_B$$
$$W^+_{C\alpha} = W_{C\alpha} + \eta\delta_\alpha\, out_C \qquad W^+_{C\beta} = W_{C\beta} + \eta\delta_\beta\, out_C$$

Consider sigmoid activation function $f_{Sigmoid}(x) = \dfrac{1}{1+e^{-x}}$



3. Calculate (back-propagate) hidden layer errors

$$\delta_j = f_{(1)}'(net_j^{(1)}) \sum_k \delta_k w_{k,j}^{(2)}$$

$$\delta_A = out_A\ (1 - out_A)\ (\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = out_B\ (1 - out_B)\ (\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = out_C\ (1 - out_C)\ (\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

4. Change hidden layer weights

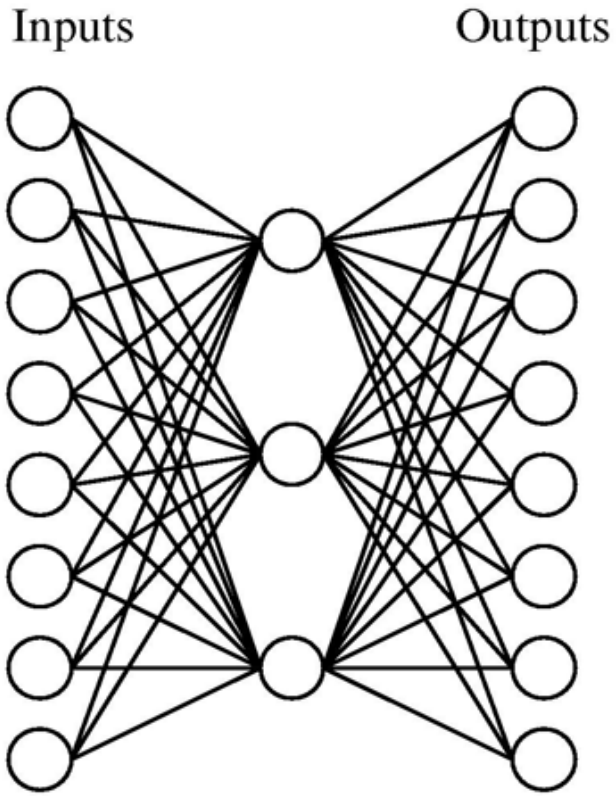
$$\Delta w_{j,m}^{(1)} = \eta Error_j Output_m = \eta \delta_j x_m$$

$$W^+_{\lambda A} = W_{\lambda A} + \eta\delta_A\, in_\lambda \qquad W^+_{\Omega A} = W^+_{\Omega A} + \eta\delta_A\, in_\Omega$$
$$W^+_{\lambda B} = W_{\lambda B} + \eta\delta_B\, in_\lambda \qquad W^+_{\Omega B} = W^+_{\Omega B} + \eta\delta_B\, in_\Omega$$
$$W^+_{\lambda C} = W_{\lambda C} + \eta\delta_C\, in_\lambda \qquad W^+_{\Omega C} = W^+_{\Omega C} + \eta\delta_C\, in_\Omega$$

$$f'_{Sigmoid}(x) = f_{Sigmoid}(x)(1 - f_{Sigmoid}(x))$$

Learning examples

- What happens when we train a network?
- Example:



Example: input and output

- A target function:

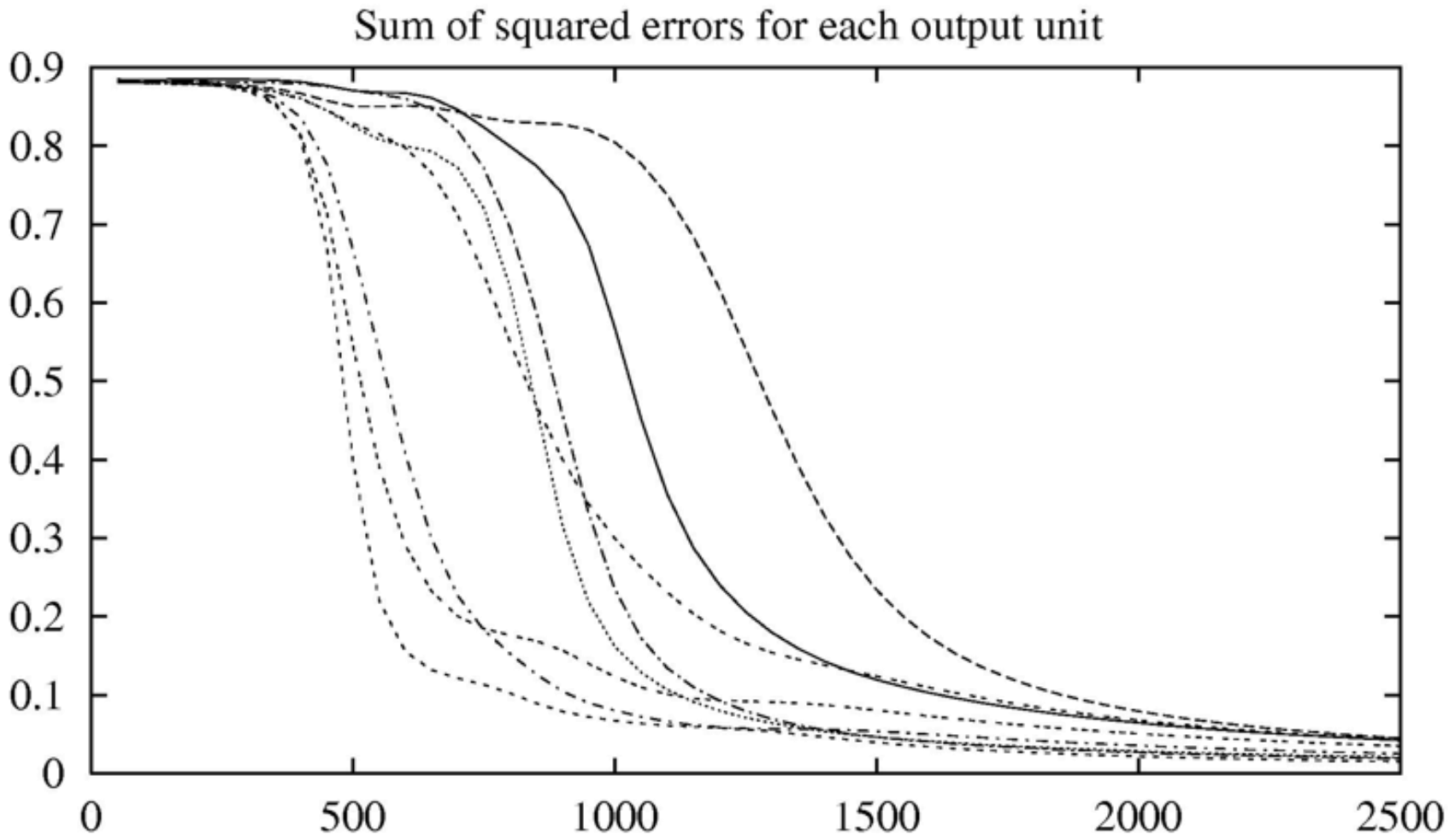
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

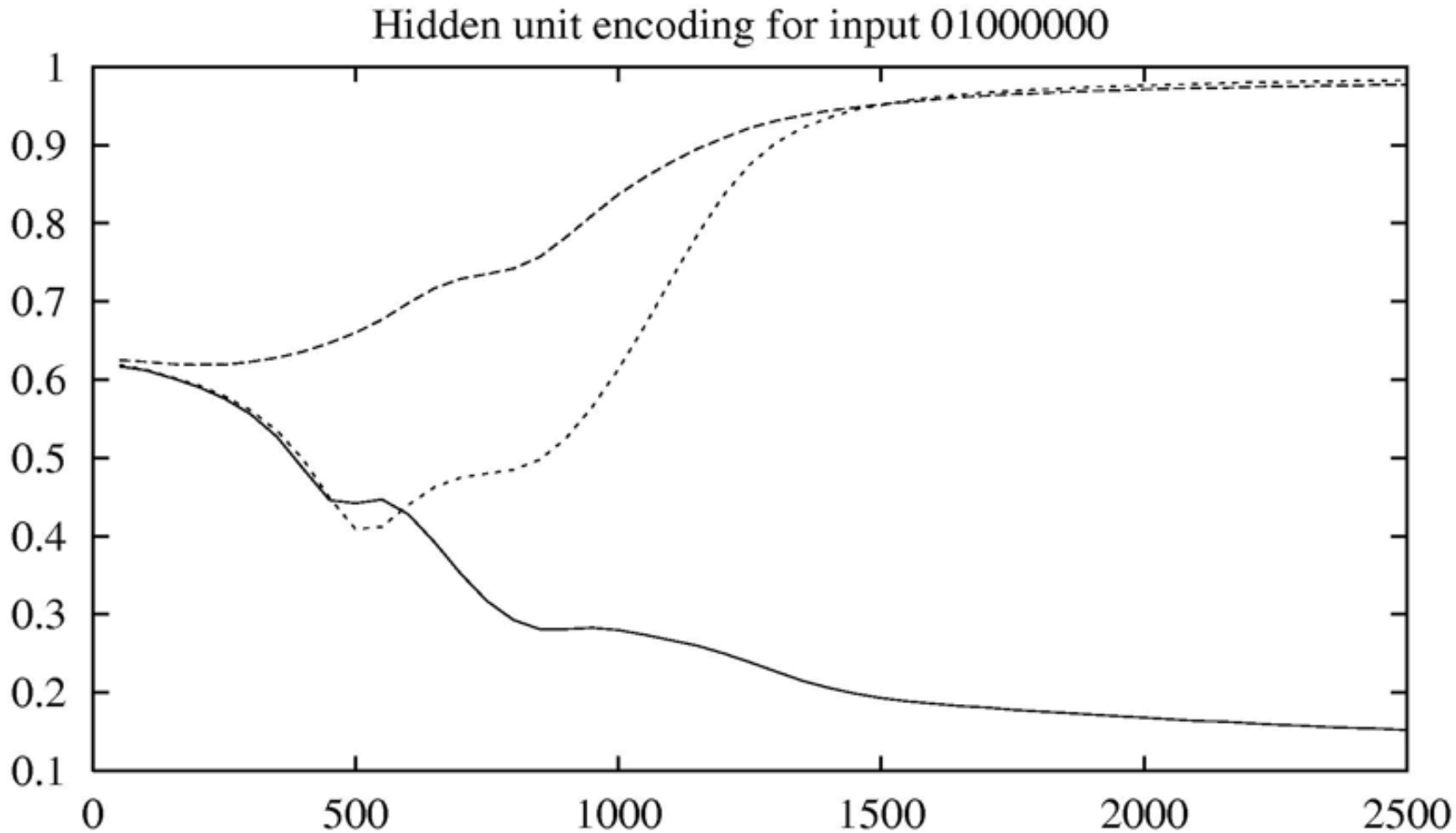
- Can this be learned?

Example: Learned hidden layer

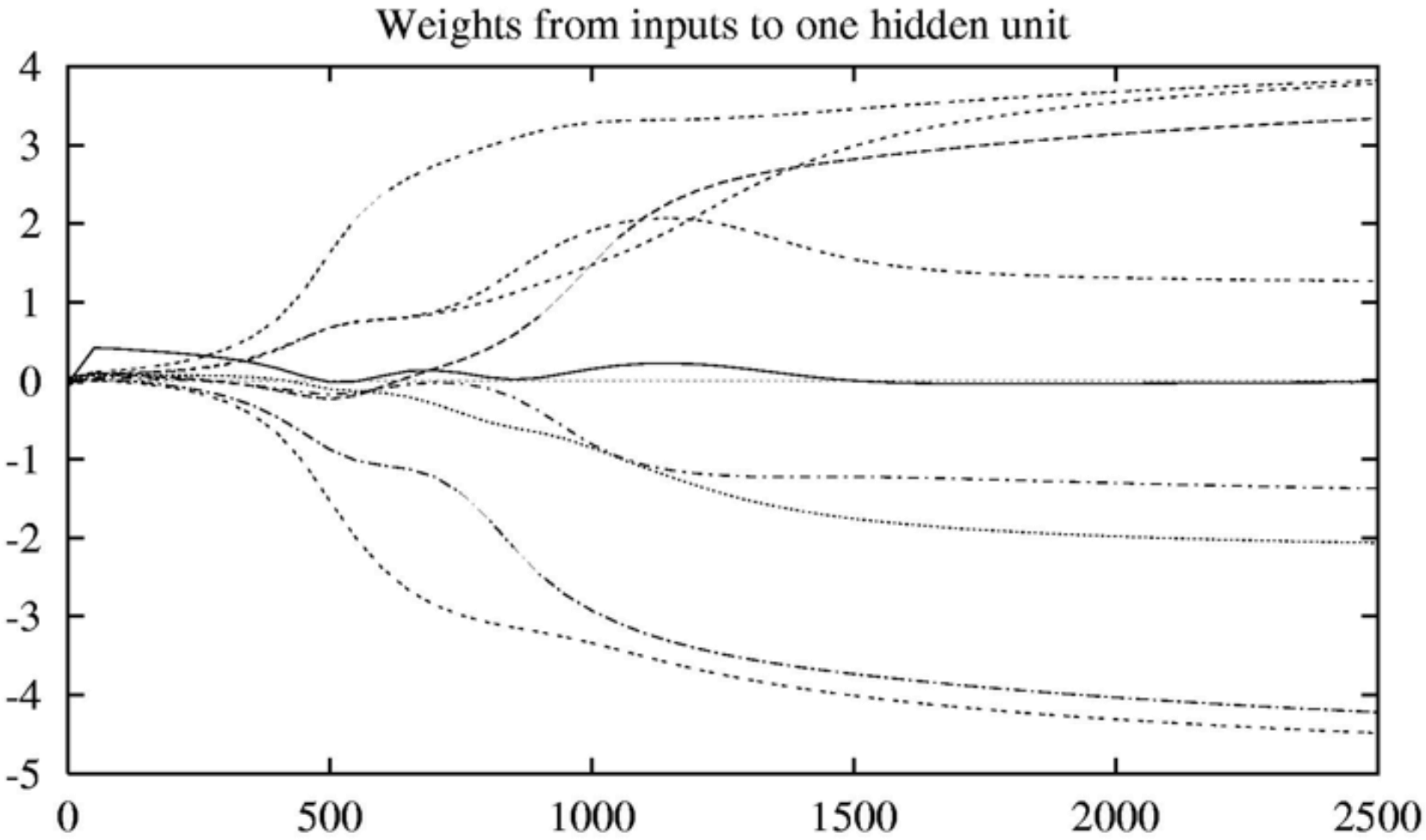
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Example: training




Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

Example: training



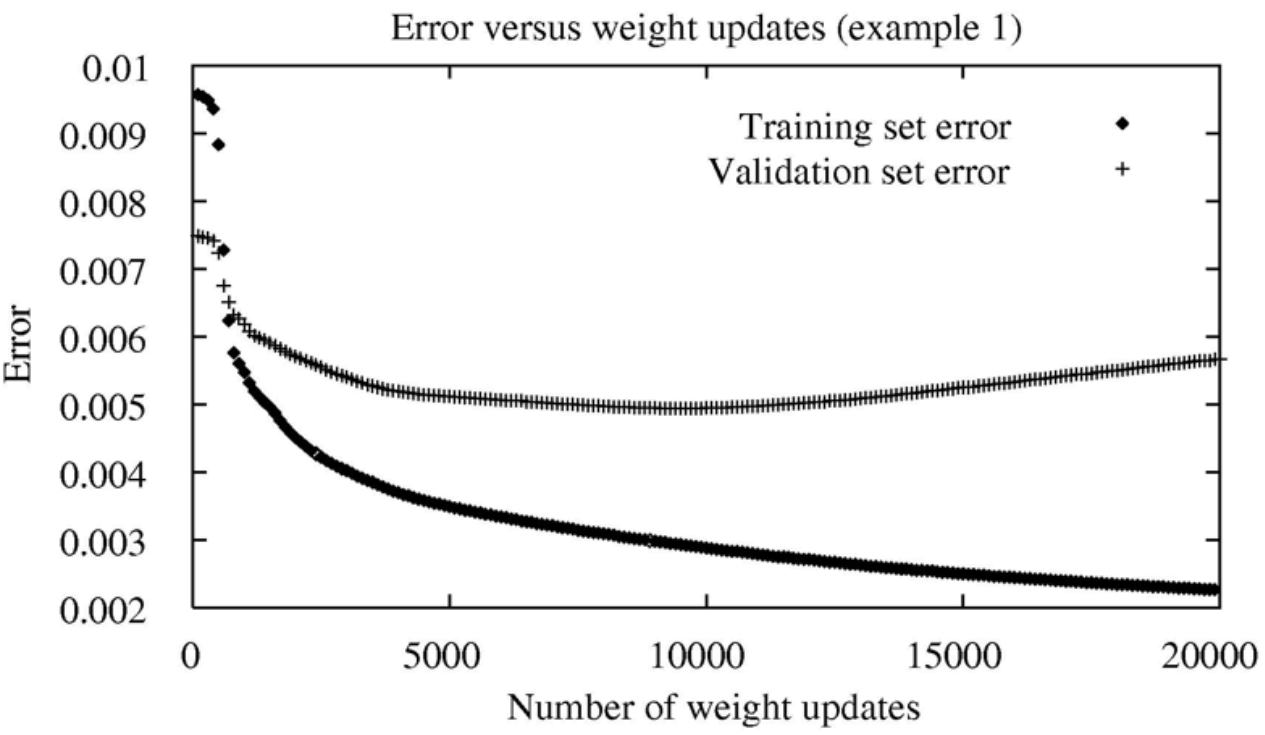
Convergence of Backpropagation

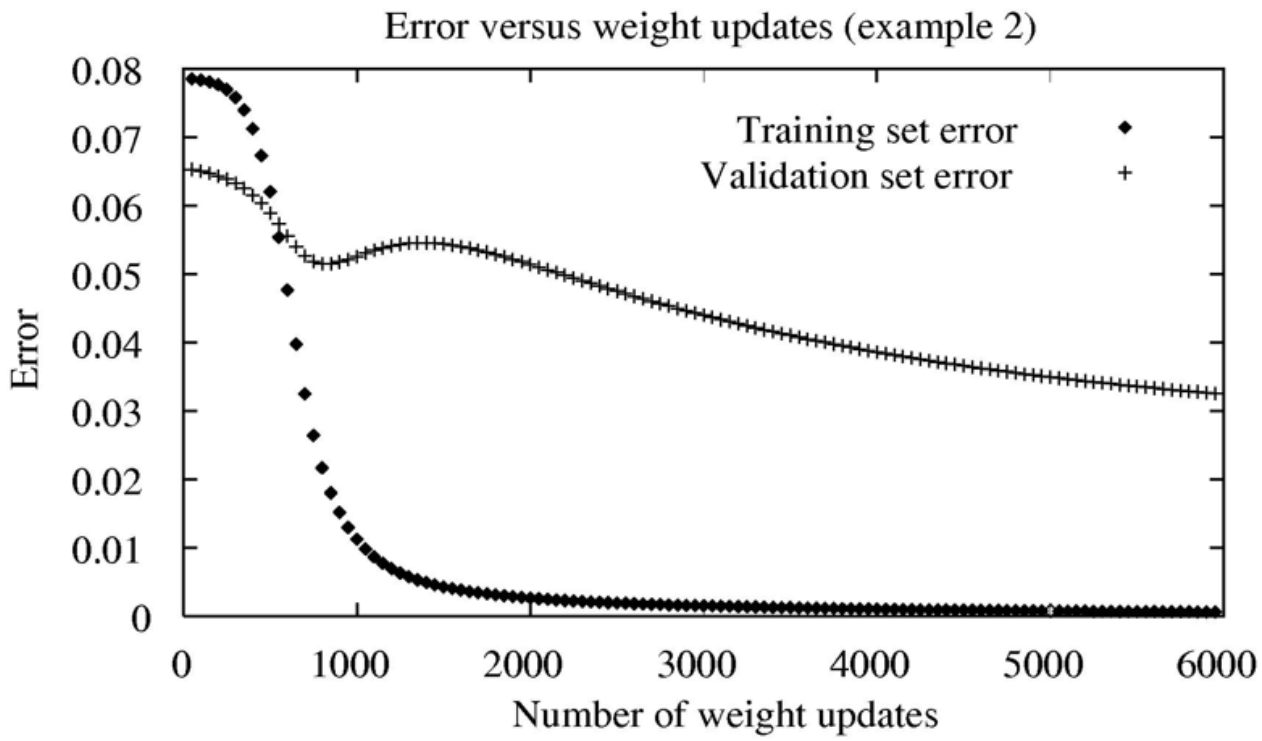
- Gradient descent to some local minimum
  - Perhaps not global minimum
  - Add momentum
  - Stochastic gradient descent
  - Train multiple nets with different initial weights
- Nature of convergence
  - Initialize weights near zero
  - Therefore, initial networks near-linear
  - Increasingly non-linear functions possible as training progresses

Expressiveness of Neural Nets

- Boolean functions:
  - Every Boolean function can be represented by network with single hidden layer
  - But might require exponential (in number of inputs) hidden units
- Continuous functions:
  - Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
  - Any function can be approximated to arbitrary accuracy by network with two hidden layers

Overfitting in Neural Nets





Avoid overfitting

- Penalize large weights: $$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

- Train on target slopes as well as values:

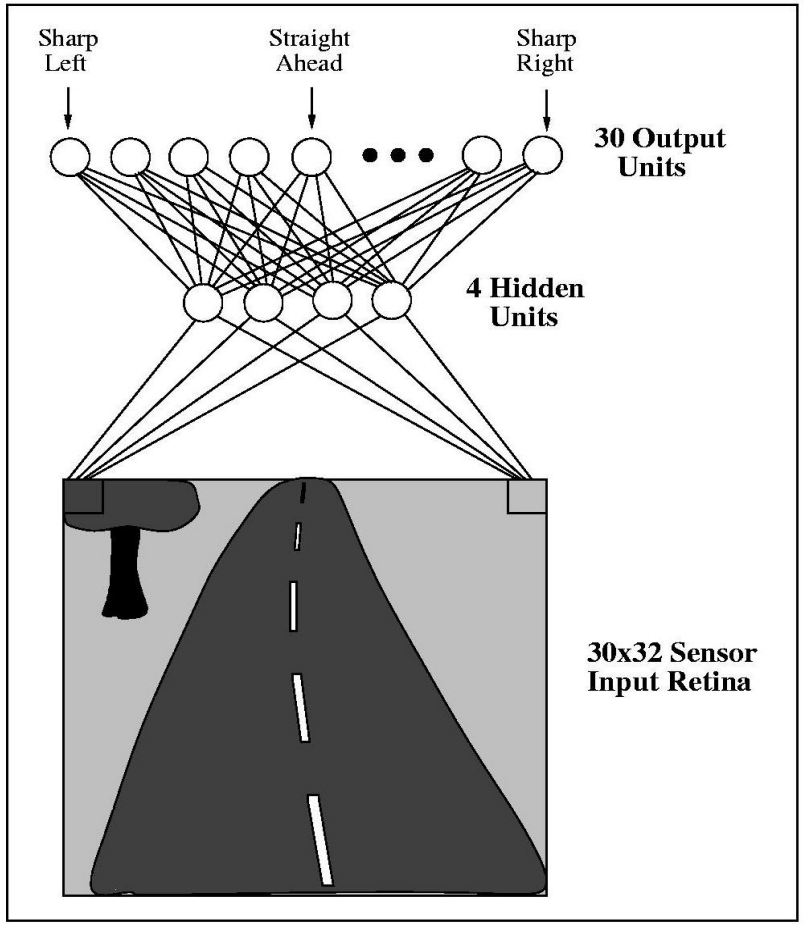
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

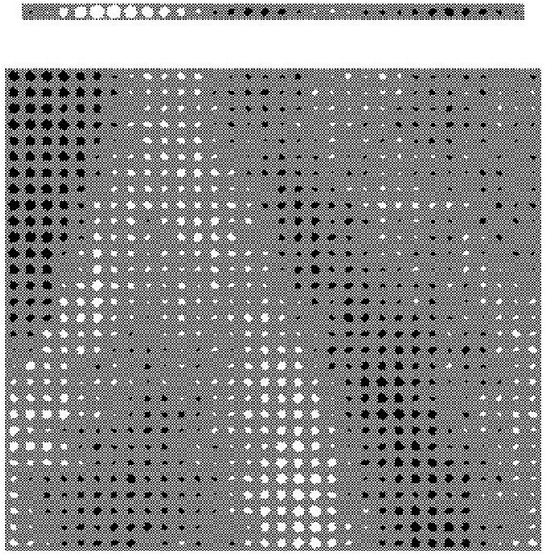
- Weight sharing
- Early stopping

Applications

- Autonomous Self-driving Cars: use neural network to learn from human drivers.

Applications





Shuai Li

https://shuaili8.github.io

Questions?

https://shuaili8.github.io/Teaching/VE445/index.html