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

Outline

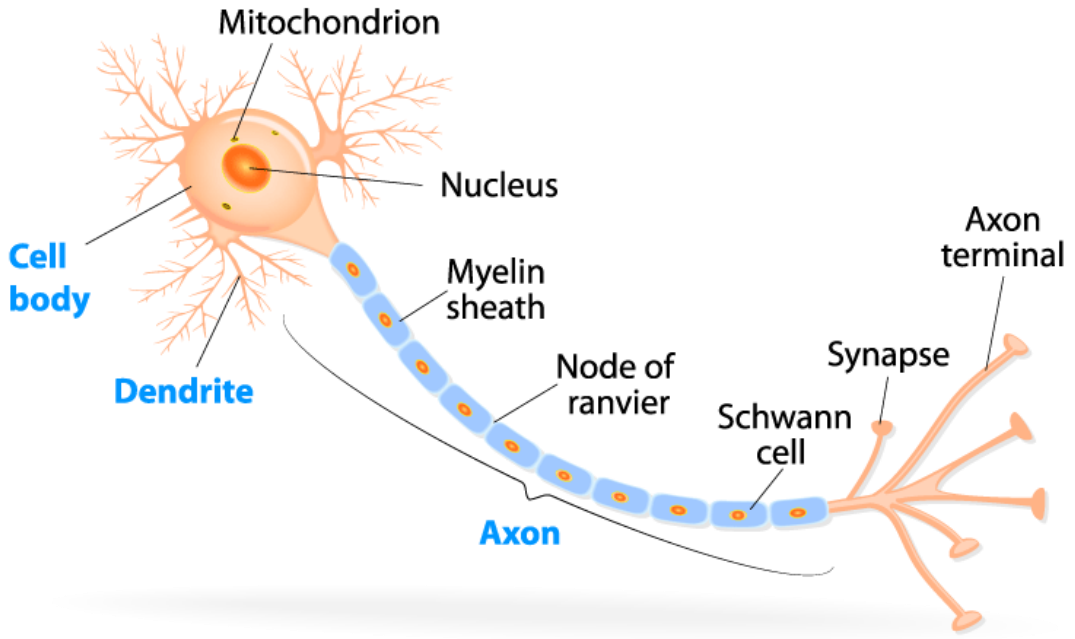
- Perceptron
- Activation functions
- Multilayer perceptron networks
- Training: backpropagation
- Examples
- Overfitting
- Applications

Brief history of artificial neural nets

- The First wave
  - 1943 McCulloch and Pitts proposed the McCulloch-Pitts neuron model
  - 1958 Rosenblatt introduced the simple single layer networks now called Perceptrons
  - 1969 Minsky and Papert's book *Perceptrons* demonstrated the limitation of single layer perceptrons, and almost the whole field went into hibernation
- The Second wave
  - 1986 The Back-Propagation learning algorithm for Multi-Layer Perceptrons was rediscovered and the whole field took off again
- The Third wave
  - 2006 Deep (neural networks) Learning gains popularity
  - 2012 made significant break-through in many applications

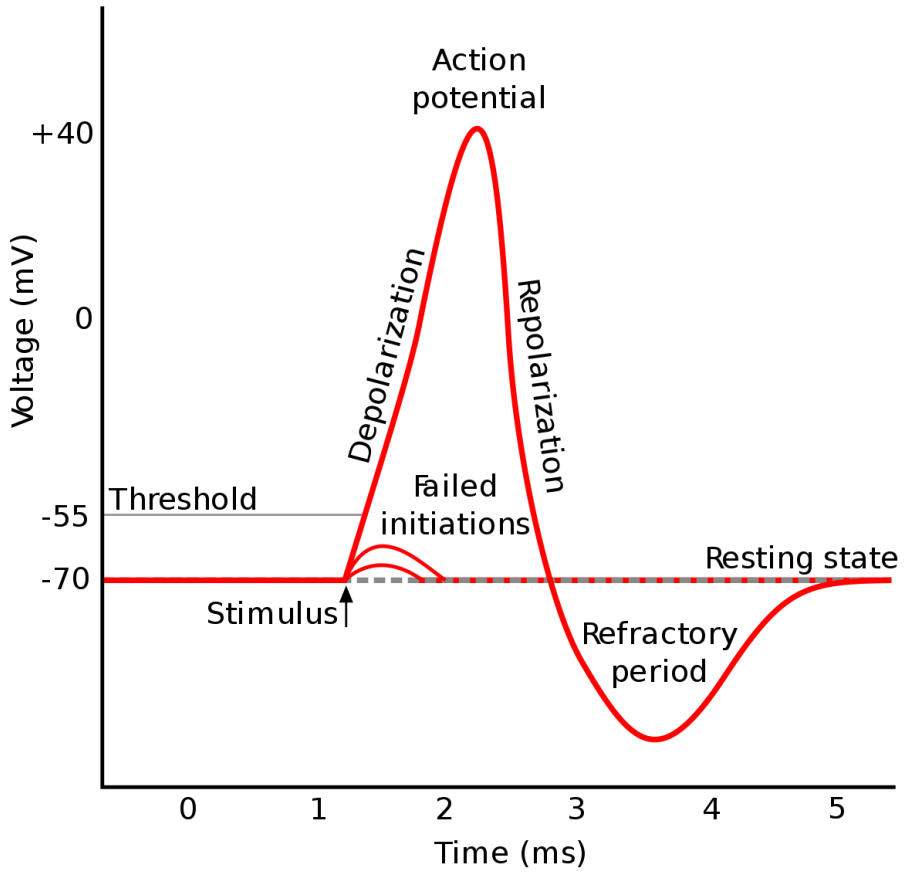
Biological neuron structure

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



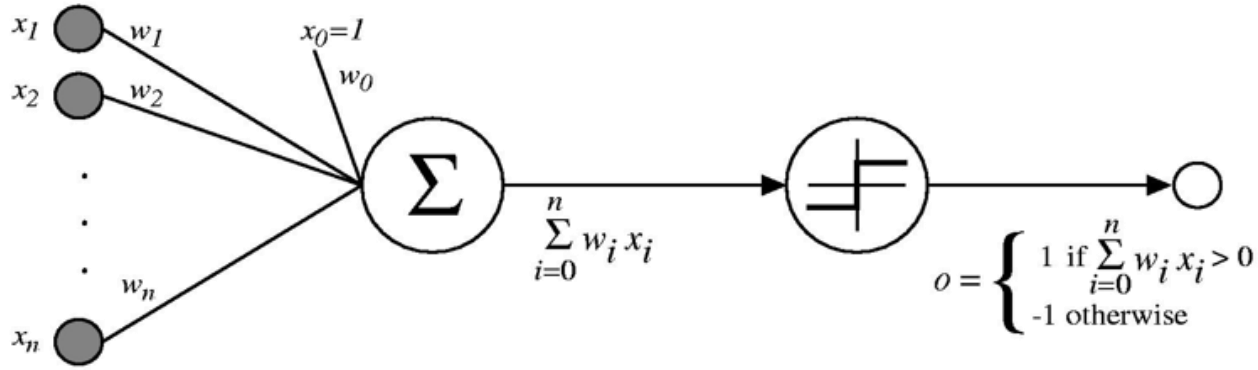
Biological neural communication

- 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



Perceptron

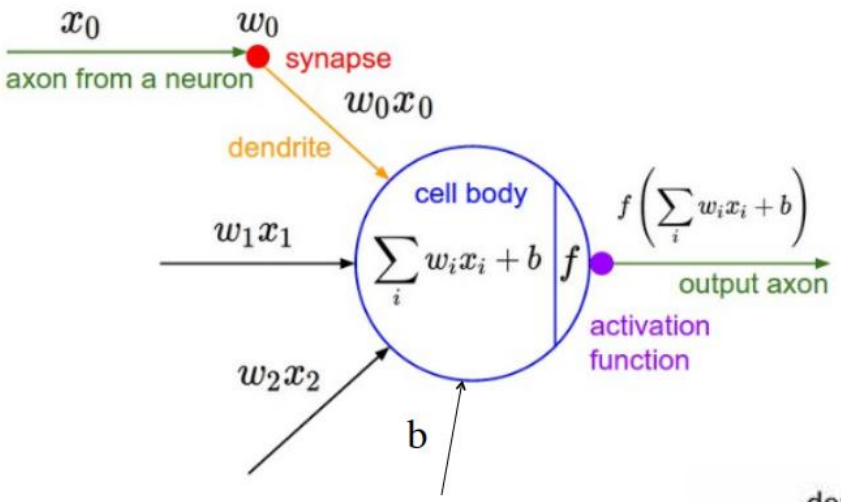
- Inspired by the biological neuron among humans and animals, researchers build a simple model called Perceptron

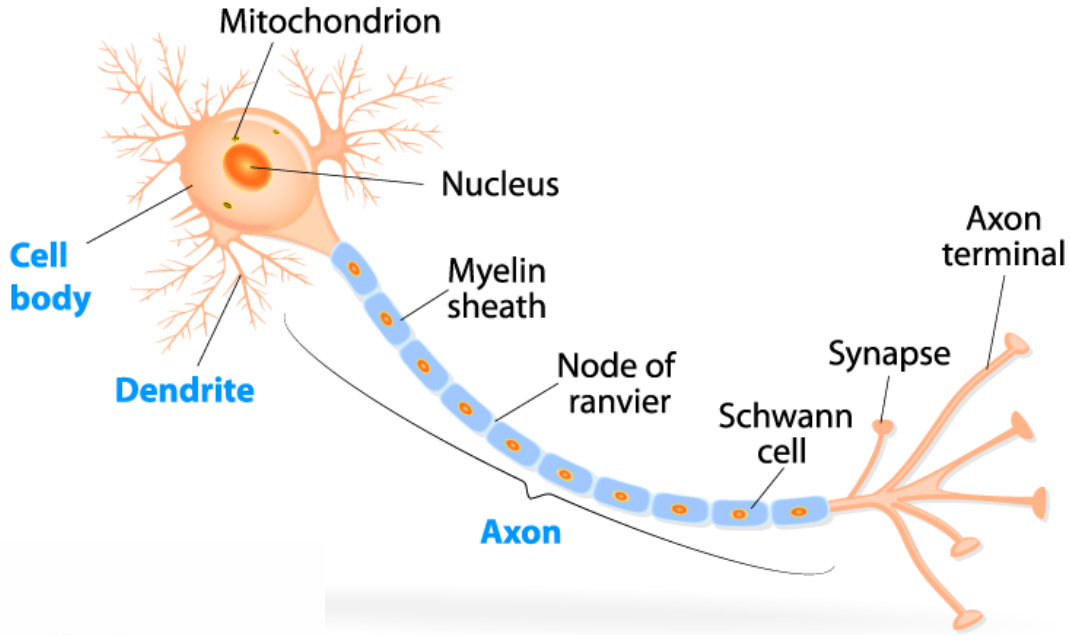


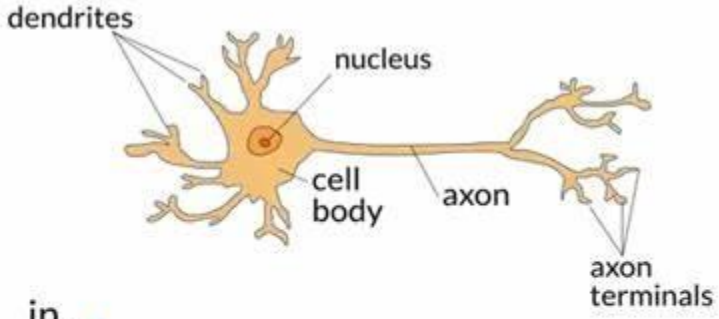
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

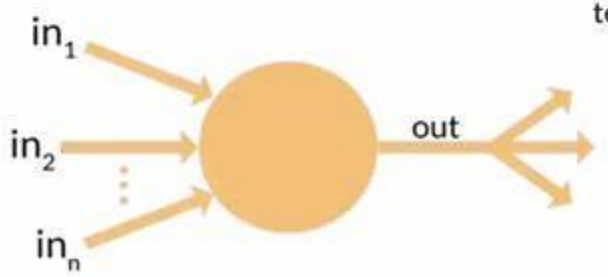
- It receives signals x_i's, multiplies them with different weights w_i, and outputs the sum of the weighted signals after an activation function, step function

Neuron vs. Perceptron



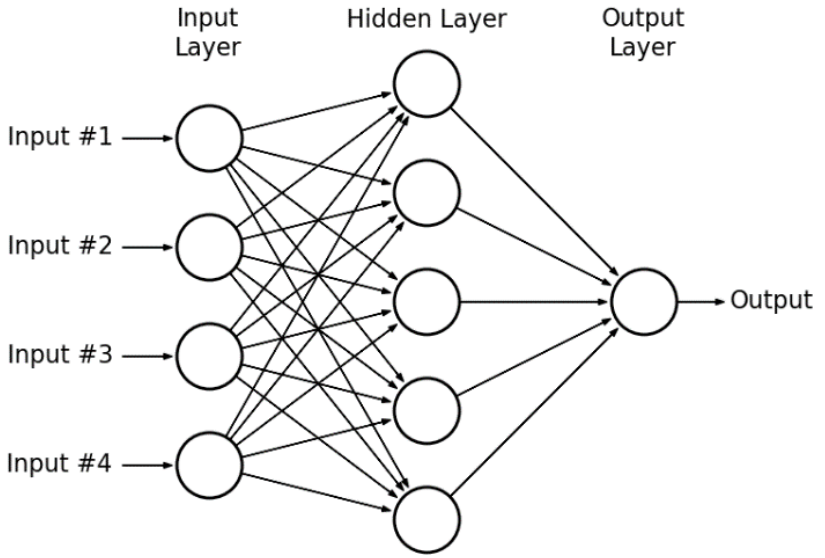


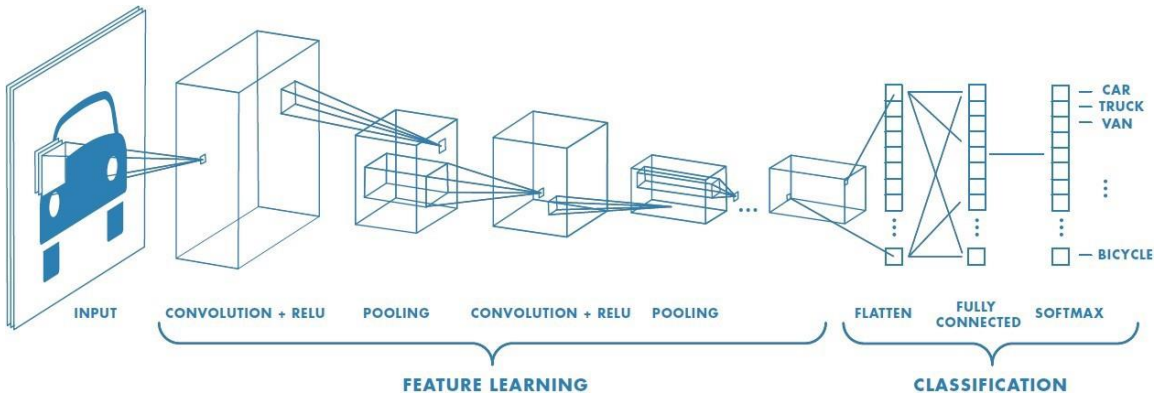


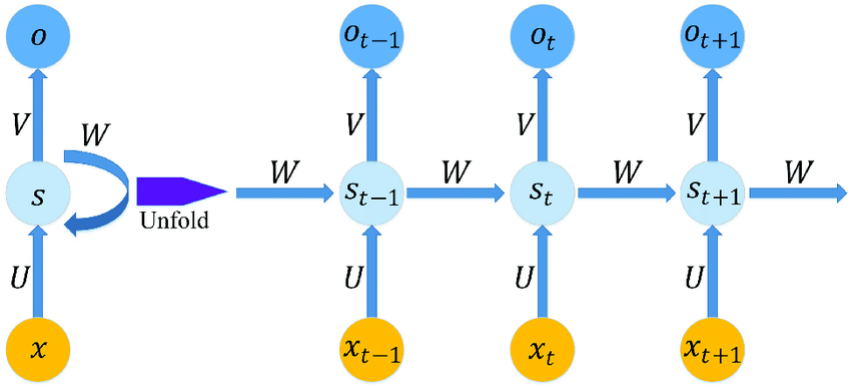


Artificial neural networks

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



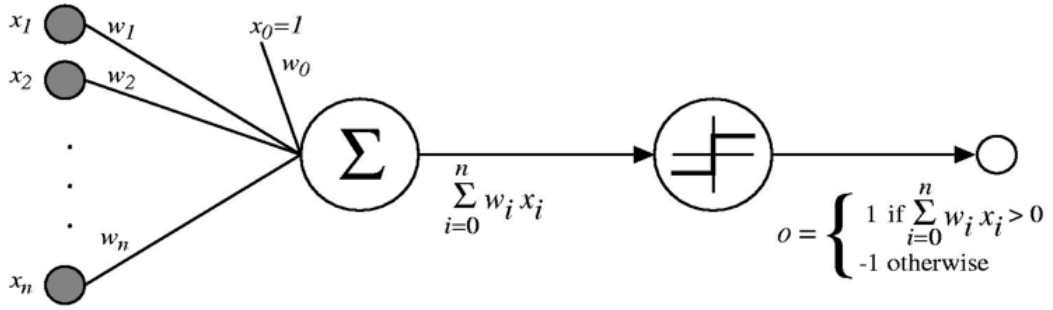




Perceptron

Training

- $w_i \leftarrow w_i - \eta(o - y)x_i$
  - y: the real label
  - o: the output for the perceptron
  - η: the learning rate

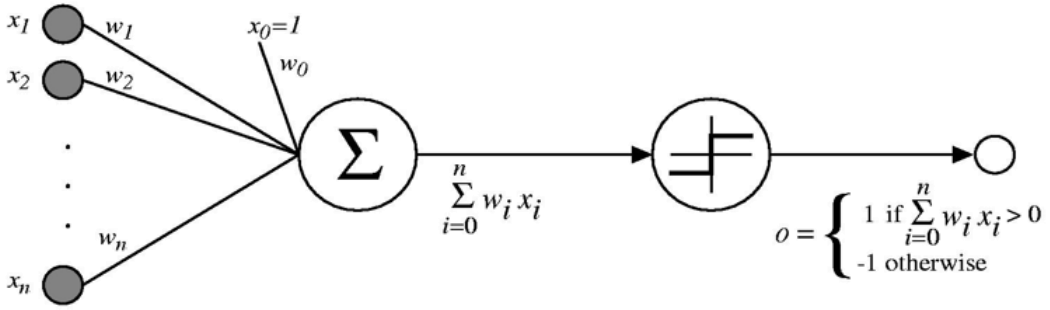


$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

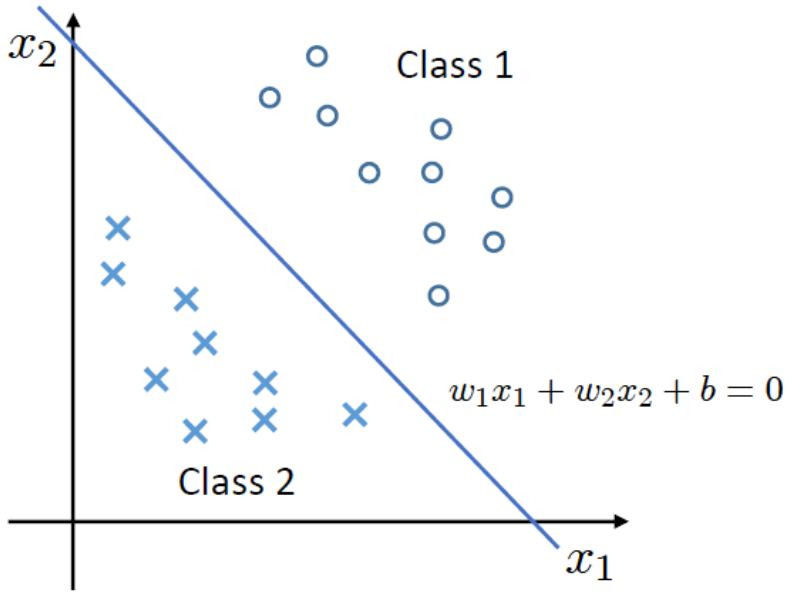
- Explanation
  - If the output is correct, do nothing
  - If the output is higher, lower the weight
  - If the output is lower, increase the weight

Properties

- Rosenblatt [1958] proved the training can converge if two classes are linearly separable and η is reasonably small



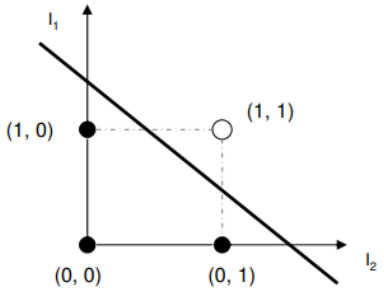
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$



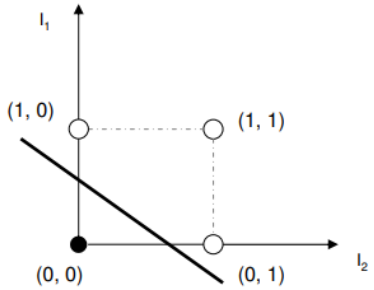
Limitation

- Minsky and Papert [1969] showed that some rather elementary computations, such as *XOR* problem, could not be done by Rosenblatt's one-layer perceptron
- However Rosenblatt believed the limitations could be overcome if more layers of units to be added, but no learning algorithm known to obtain the weights yet

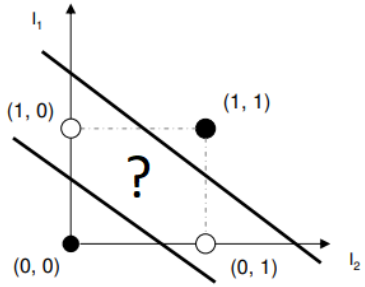
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



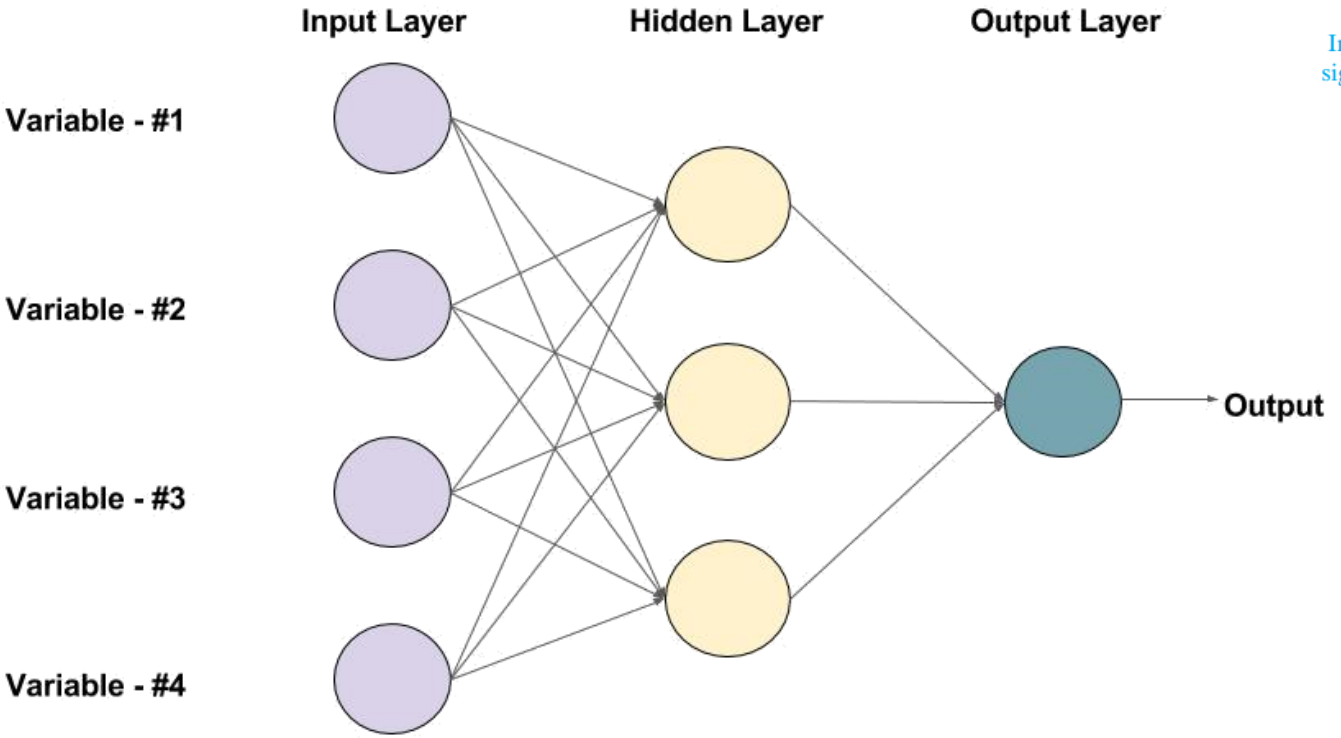
OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1



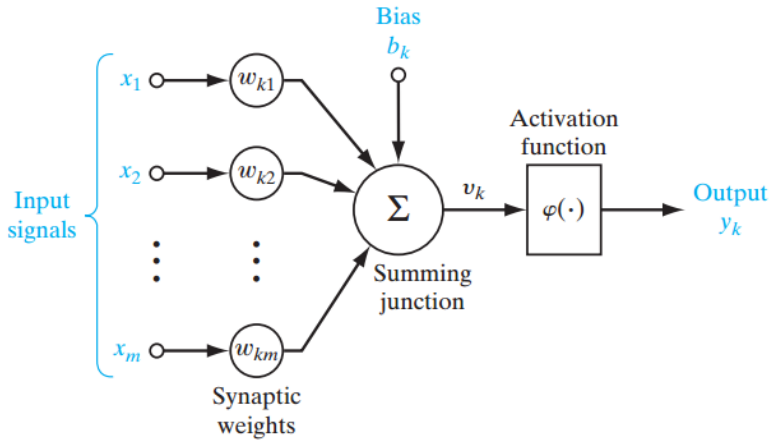
XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



Solution: Add hidden layers

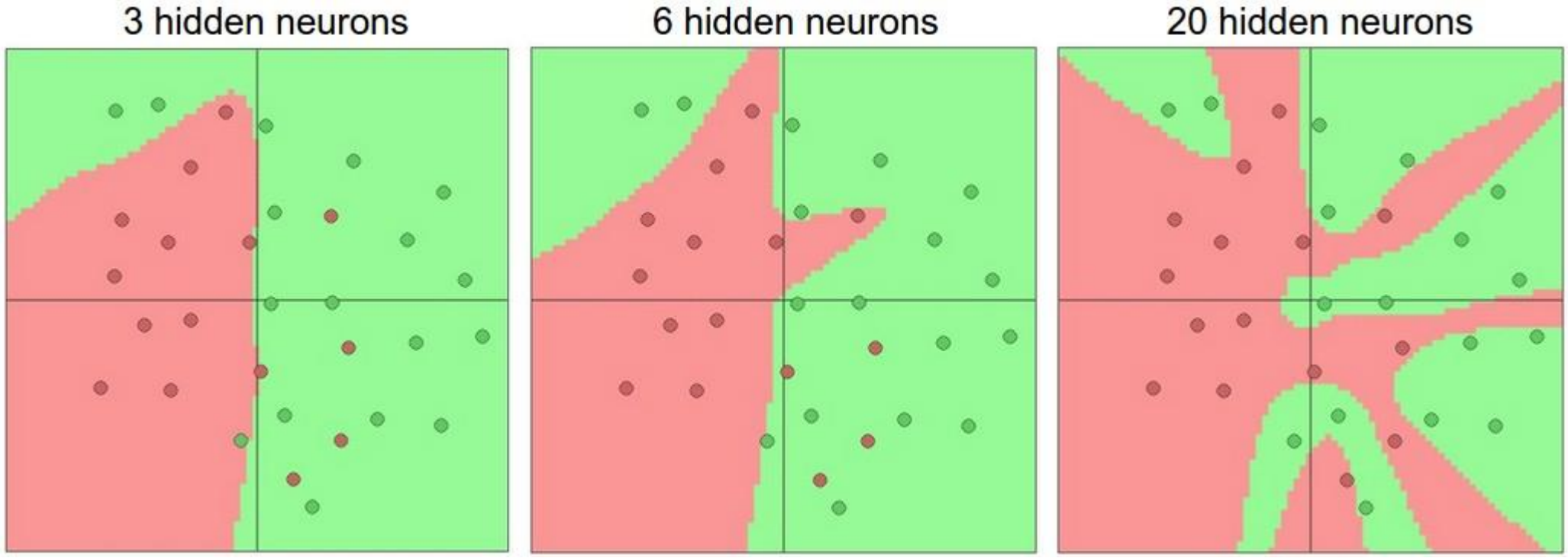


An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)



- Two-layer feedforward neural network

Demo



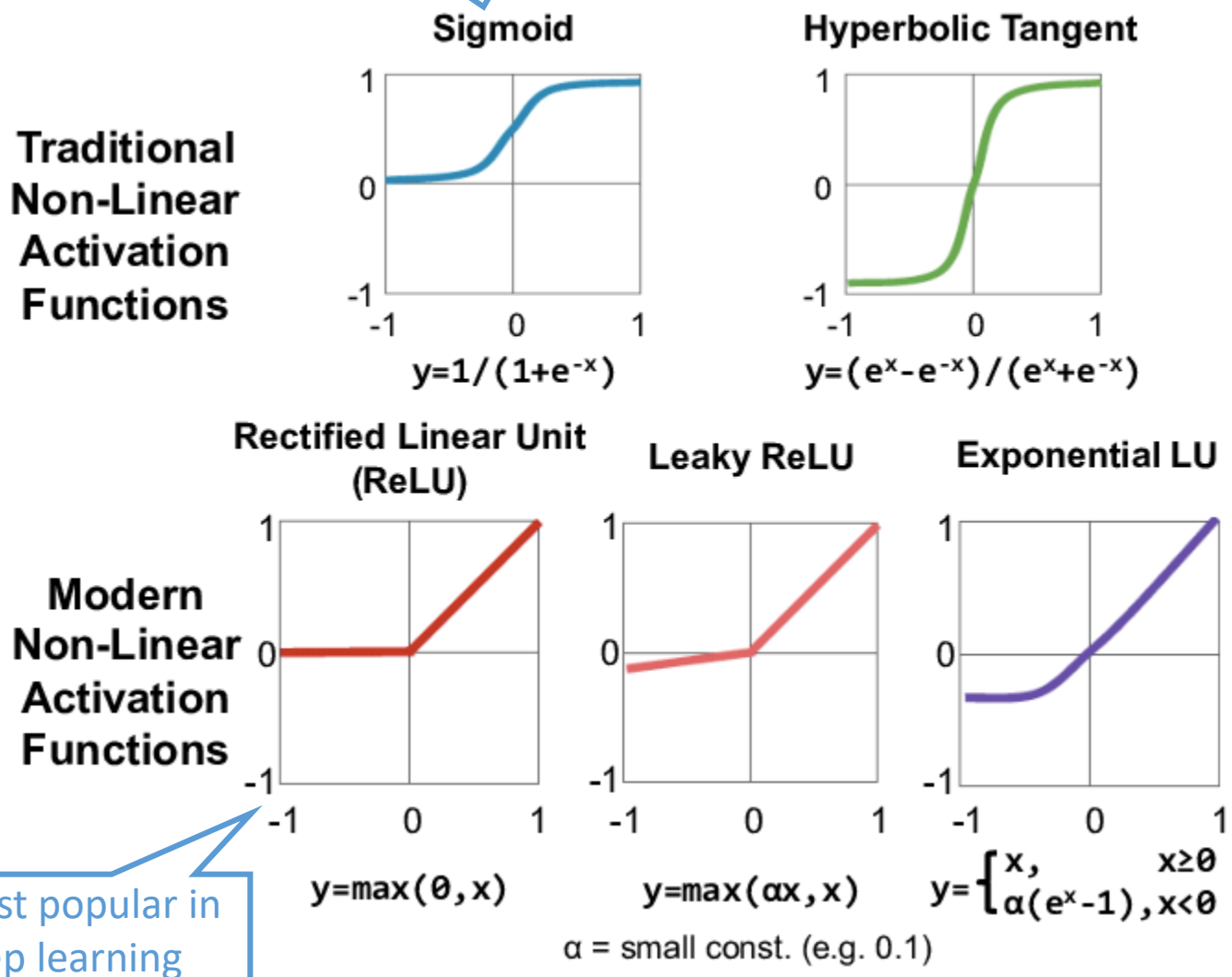
- Large Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath. You can play with these examples in this ConvNetsJS demo.

Activation Functions

Activation functions

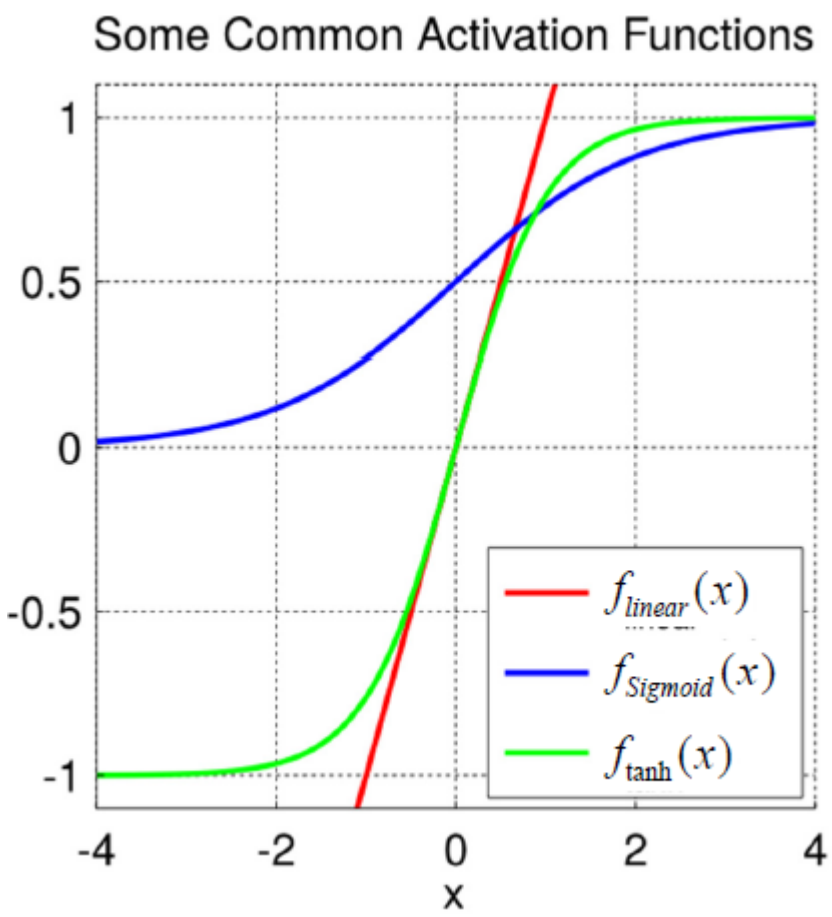
- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
- Tanh: $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- ReLU (Rectified Linear Unity): $\mathrm{ReLU}(z) = \max(0, z)$

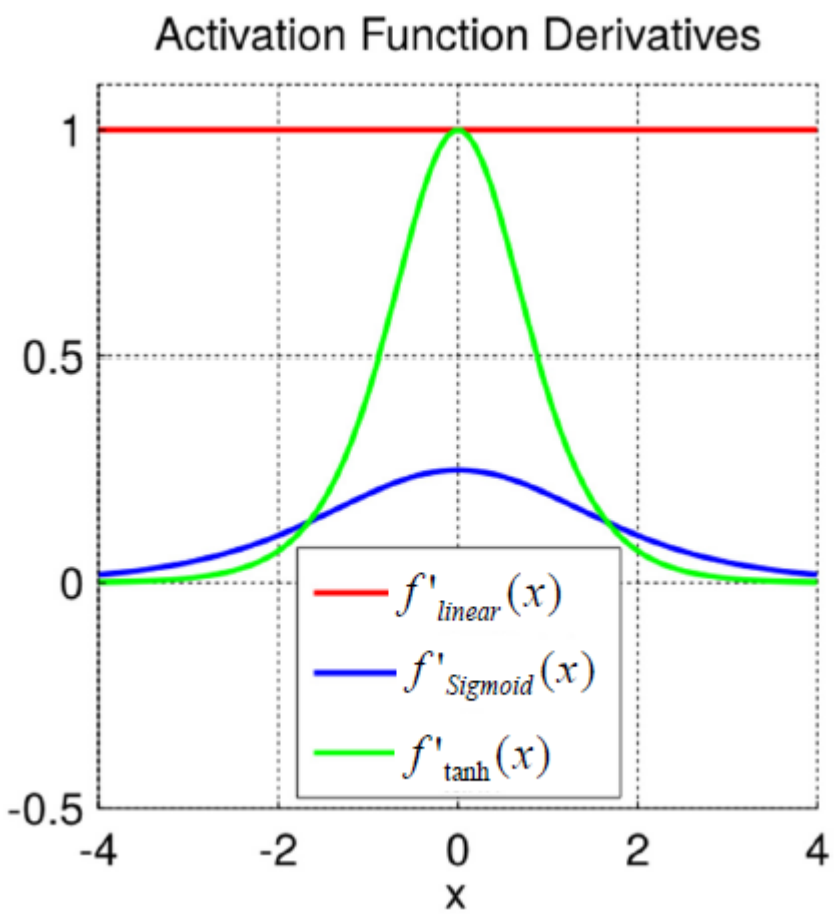
Most popular in fully connected neural network



Most popular in deep learning

Activation function values and derivatives

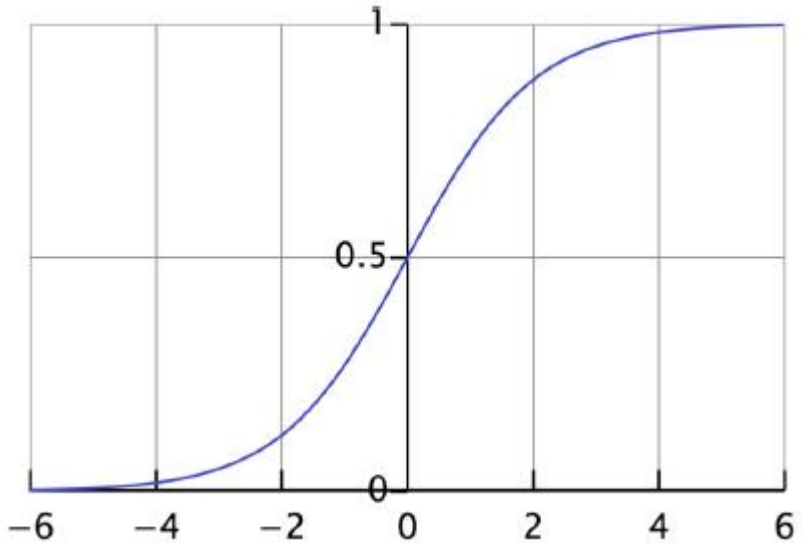




Sigmoid activation function

- Sigmoid

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

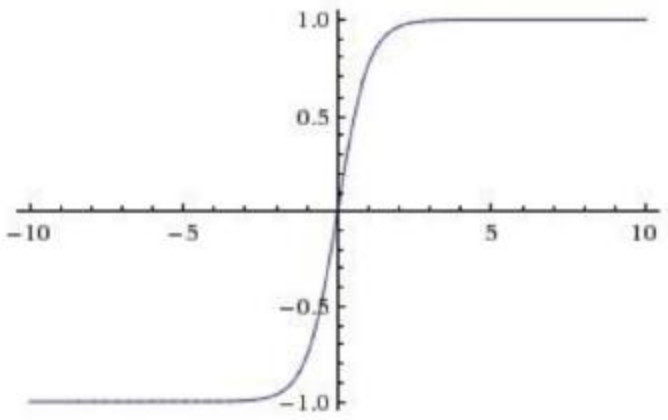


- Its derivative
  $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$
- Output range $(0,1)$
- Motivated by biological neurons and can be interpreted as the probability of an artificial neuron “firing” given its inputs
- However, saturated neurons make value vanished **(why?)**
  - $f\Big(f\big(f(\cdots)\big)\Big)$
  - $f([0,1]) \subseteq [0.5, 0.732)$
  - $f\big([0.5, 0.732)\big) \subseteq (0.622, 0.676)$

Tanh activation function

- Tanh function

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



- Its derivative

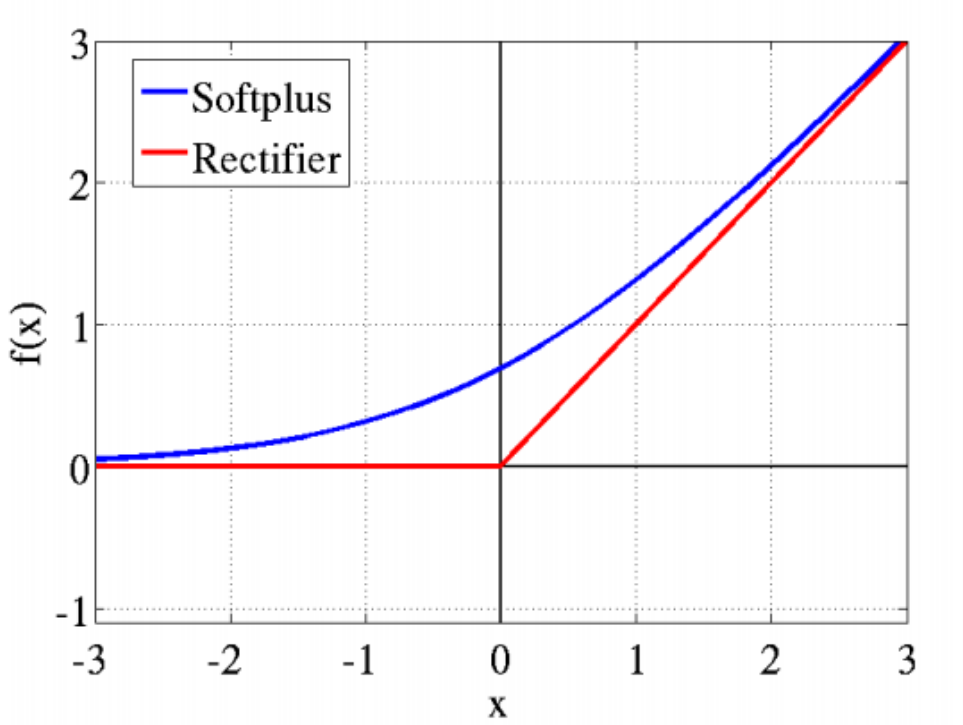
$$\tanh'(z) = 1 - \tanh^2(z)$$

- 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 (Rectified linear unity) function

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



- Its derivative

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

- ReLU can be approximated by softplus function

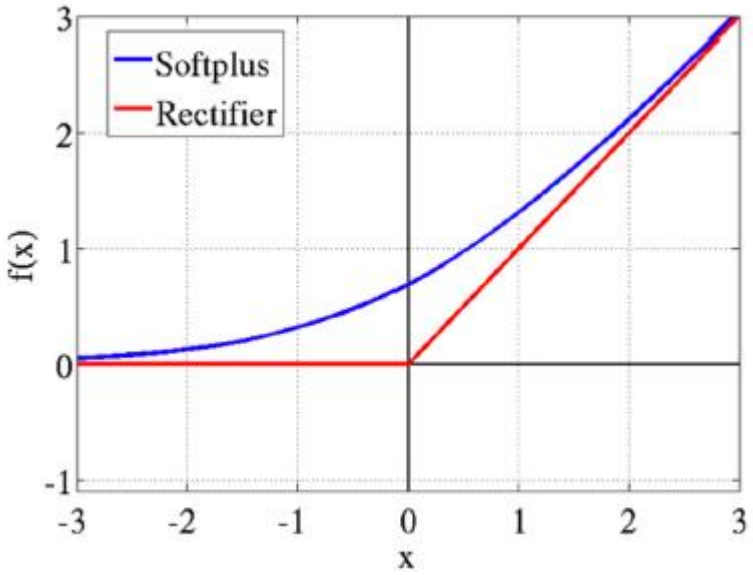
$$f_{\mathrm{Softplus}}(x) = \log(1 + e^{x})$$

- ReLU's gradient doesn't vanish as x increases
- Speed up training of neural networks
  - Since the gradient computation is very simple
  - The computational step is simple, no exponentials, no multiplication or division operations (compared to others)
- The gradient on positive portion is larger than sigmoid or tanh functions
  - Update more rapidly
  - The left "dead neuron" part can be ameliorated by Leaky ReLU

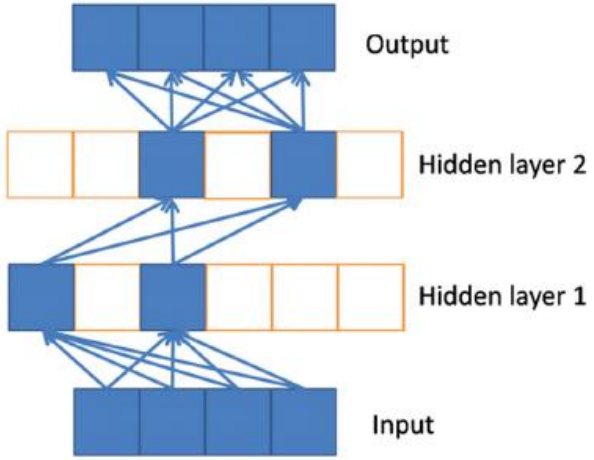
ReLU activation function (cont.)

- ReLU function

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



- 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

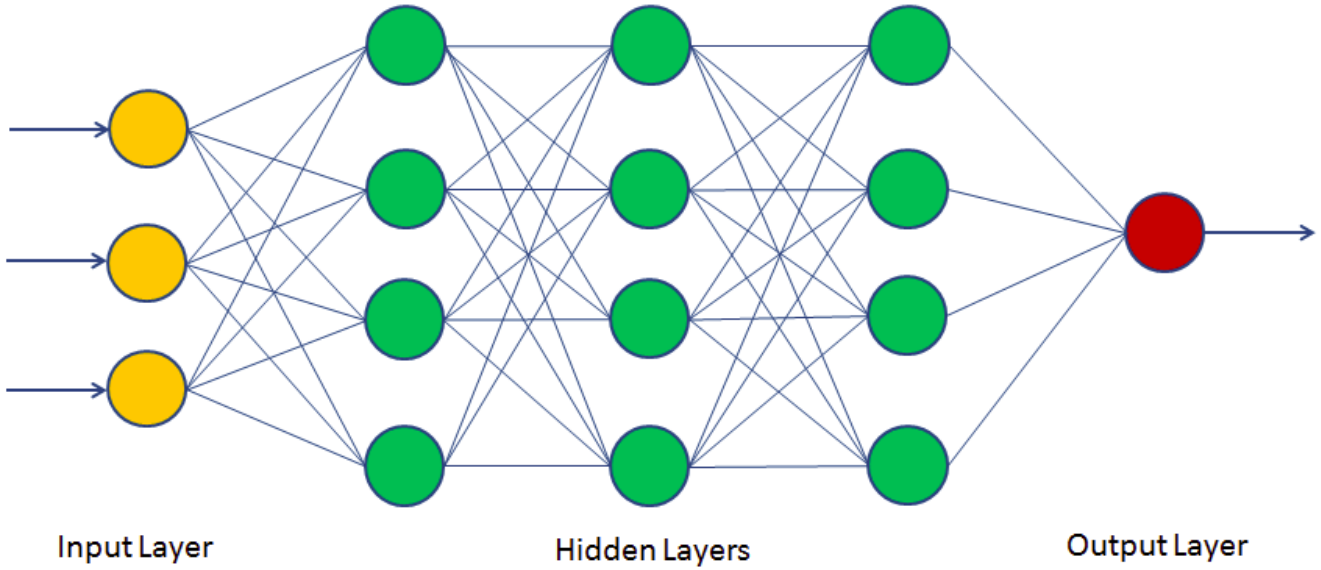


Sparse propagation of activations and gradients

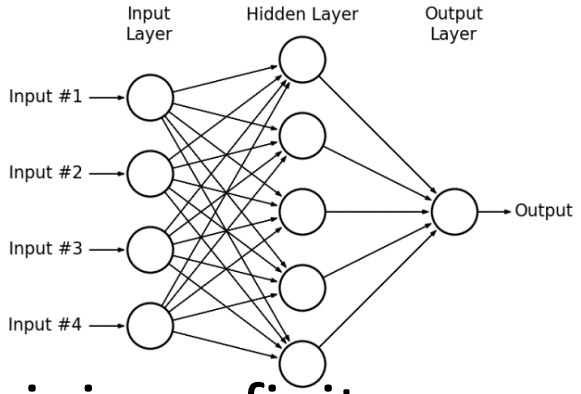
Multilayer Perceptron Networks

Neural networks

- Neural networks are built by connecting many perceptrons together, layer by layer

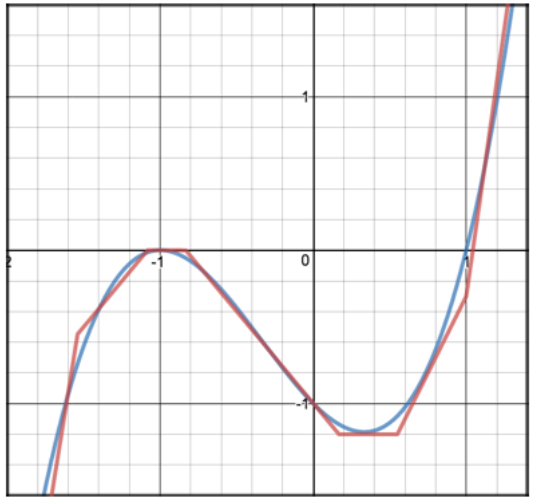


Universal approximation theorem



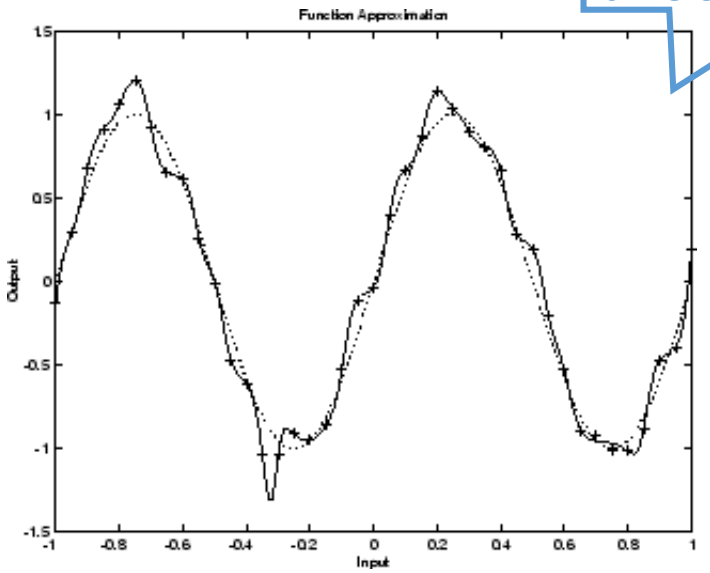
- A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions

1-20-1 NN approximates a noisy sine function



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

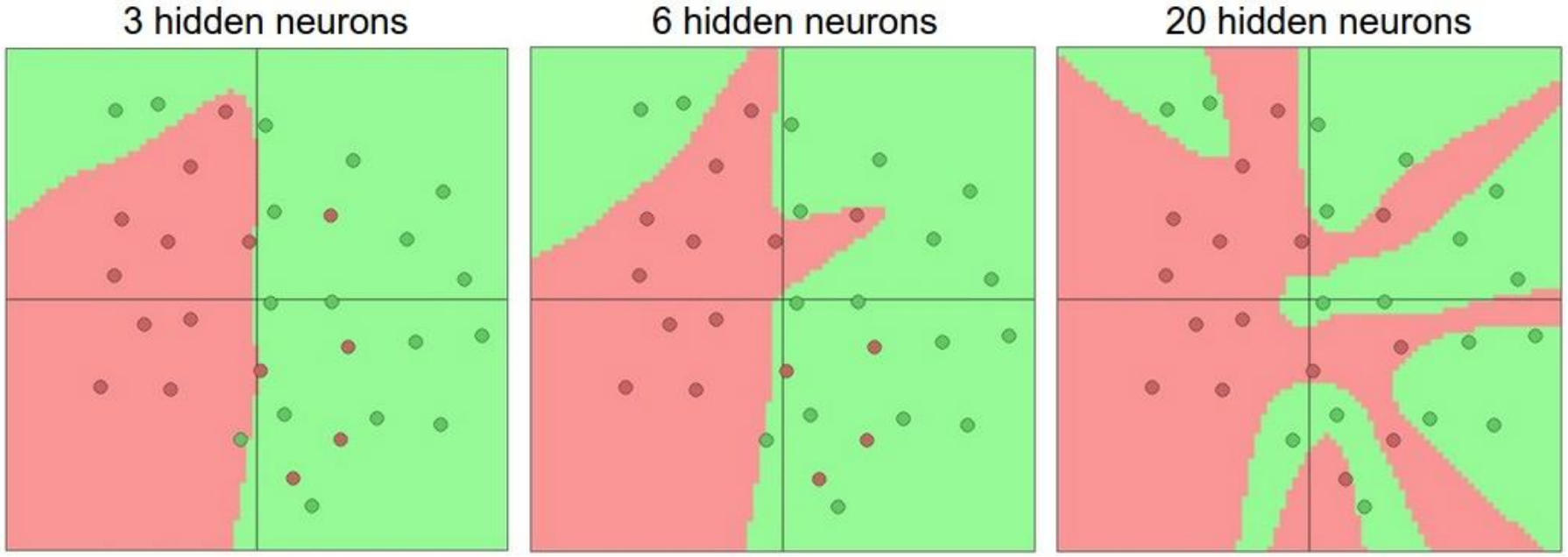
$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$



Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." *Neural networks* 2.5 (1989): 359-366

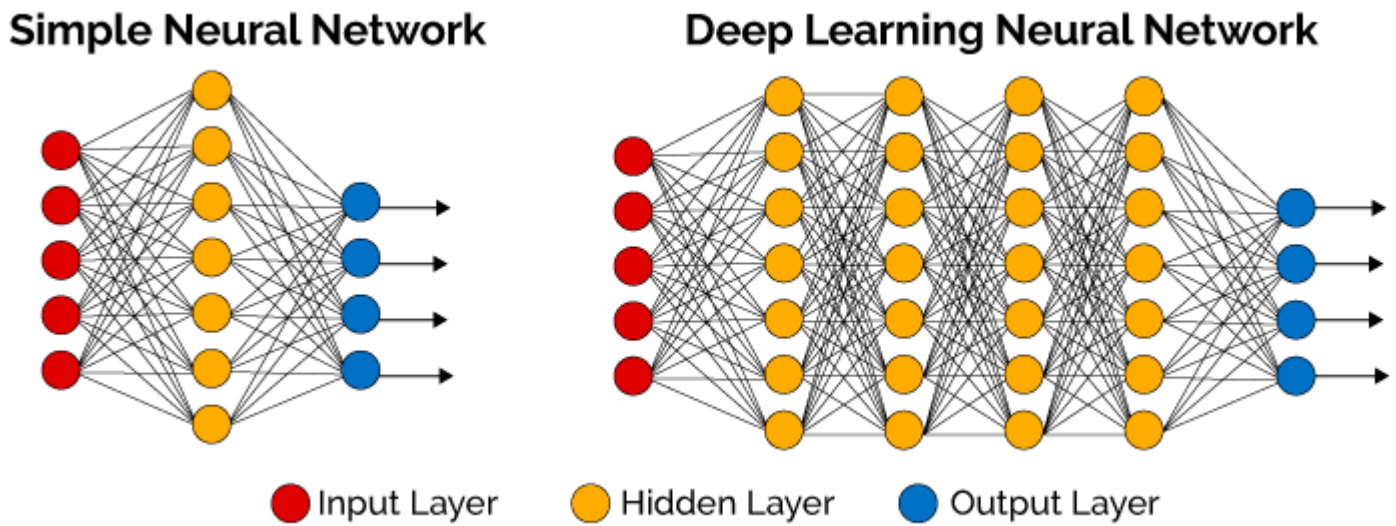
Increasing power of approximation

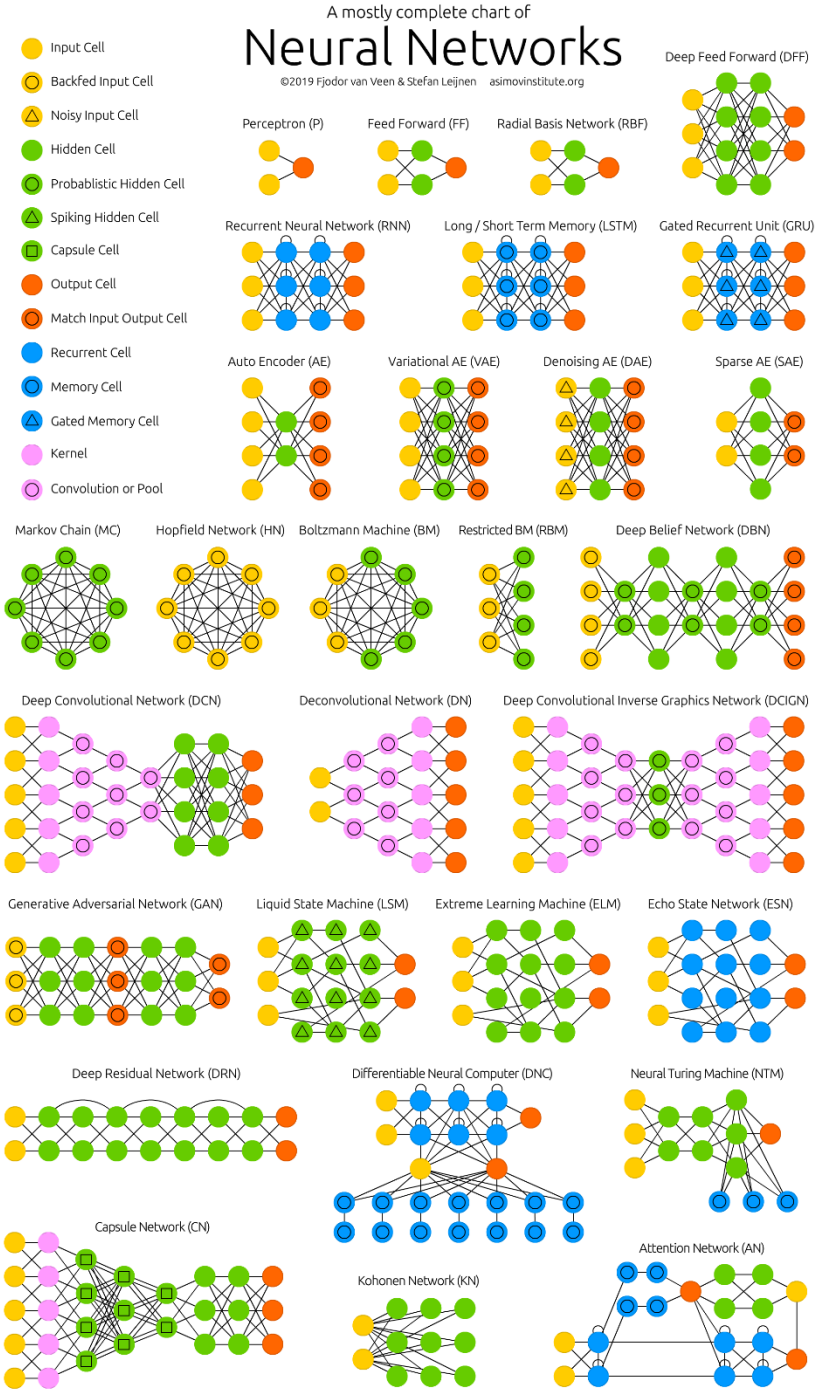
- With more neurons, its approximation power increases. The decision boundary covers more details



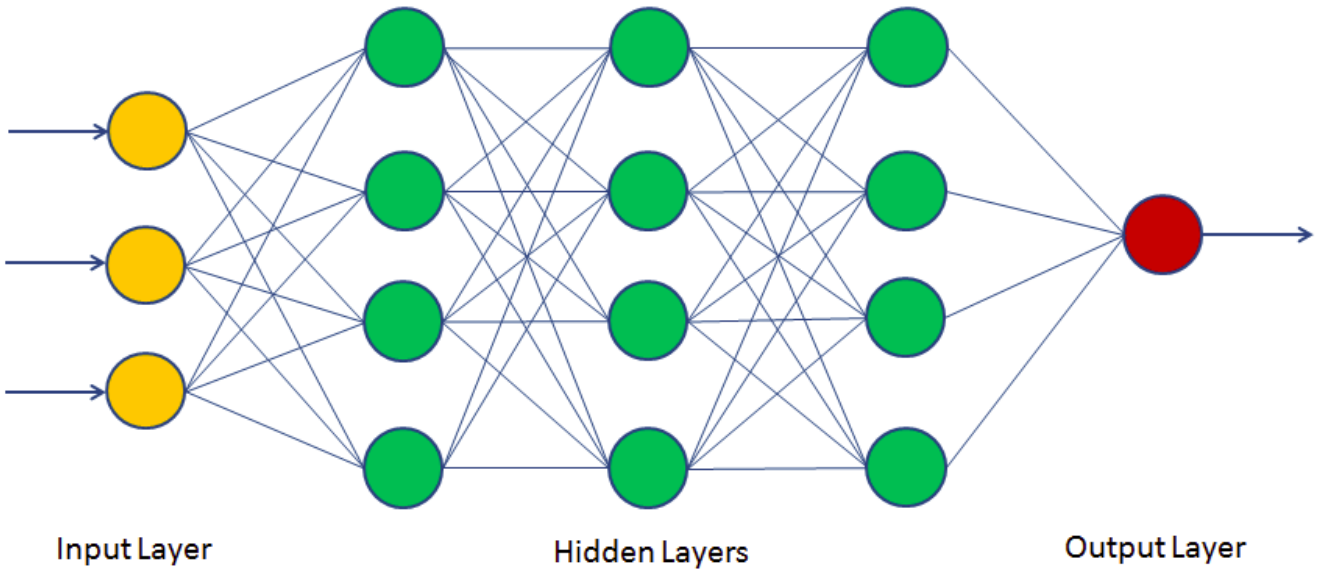
- Usually in applications, we use more layers with structures to approximate complex functions instead of one hidden layer with many neurons

Common neuron network structures at a glance





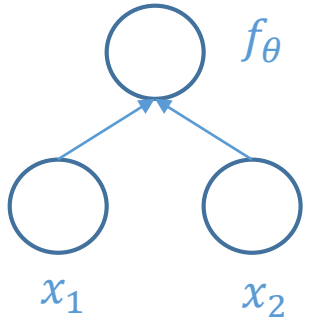
Multilayer perceptron network



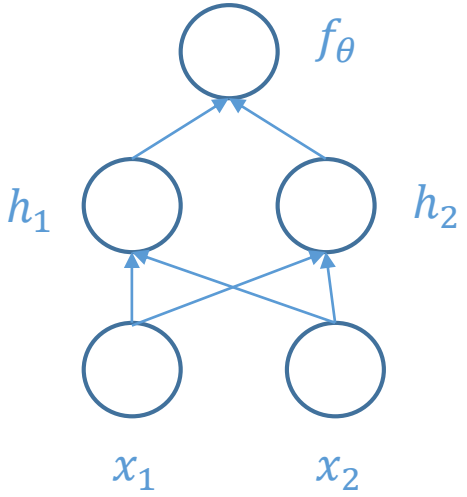
Single / Multiple layers of calculation

- Single layer function

$$f_\theta(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$



- Multiple layer function
  - $h_1(x) = \sigma(\theta_0^1 + \theta_1^1 x_1 + \theta_2^1 x_2)$
  - $h_2(x) = \sigma(\theta_0^2 + \theta_1^2 x_1 + \theta_2^2 x_2)$
  - $f_\theta(h) = \sigma(\theta_0 + \theta_1 h_1 + \theta_2 h_2)$



Training

Backpropagation

How to train?

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

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

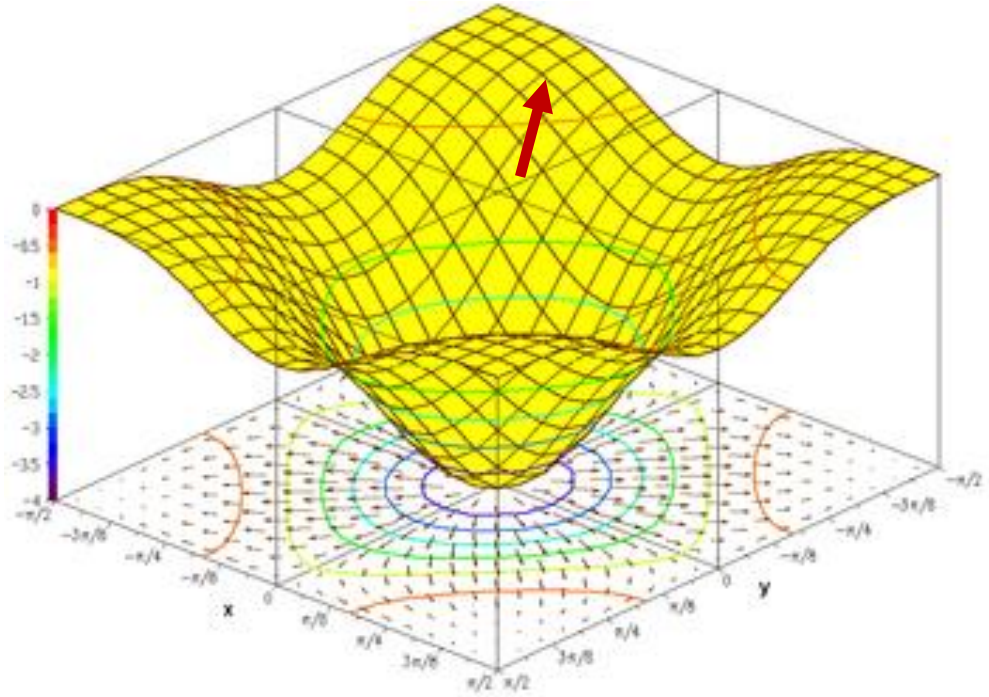
The set of training examples

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

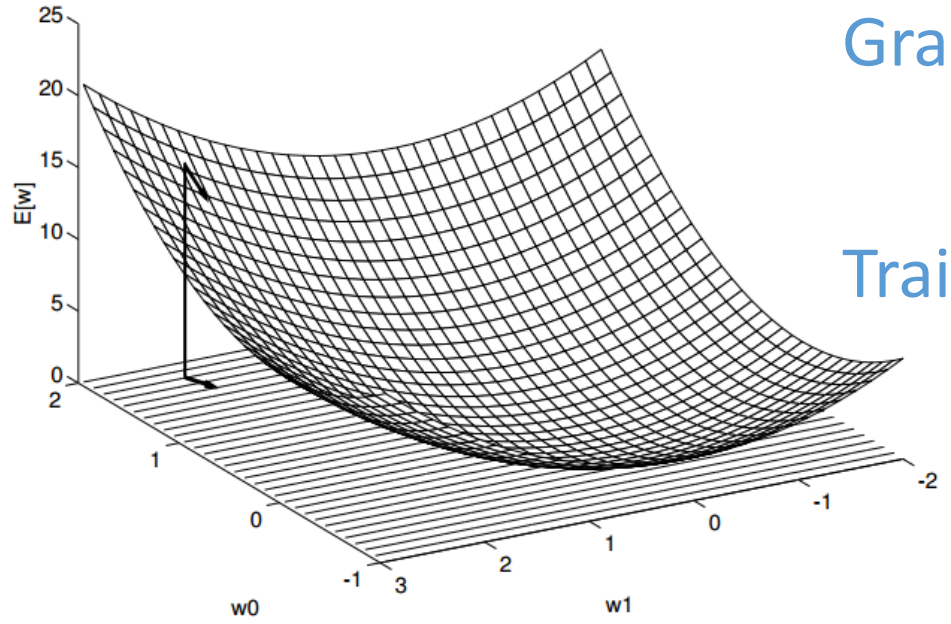
- To find a (local) minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or an approximation) of the function at the current point

- For a smooth function $f(x)$, $\frac{\partial f}{\partial x}$ is the direction that f increases most rapidly. So we apply

$$x_{t+1} = x_t - \eta \frac{\partial f}{\partial x}(x_t)$$

  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

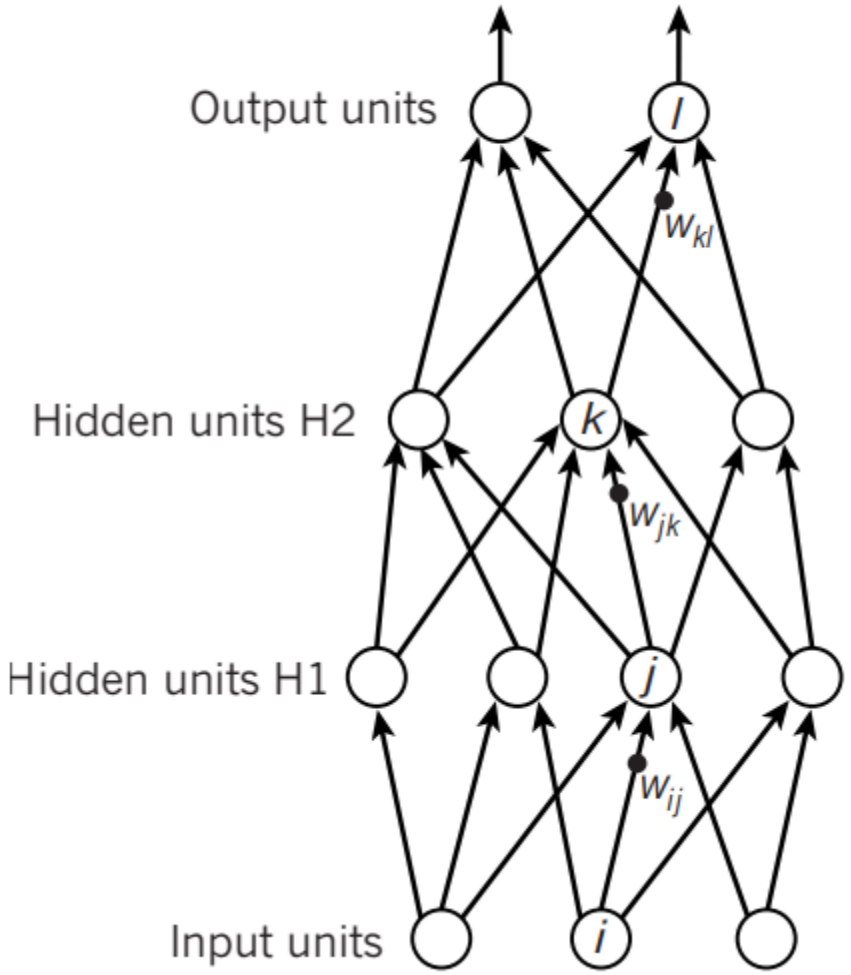
- The challenge in neural network model is that we only know the target of the output layer, but don't know the target for hidden and input layers, how can we update their connection weights using the gradient descent?

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

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

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

Feed forward vs. Backpropagation



$$y_l = f(z_l)$$

$$z_l = \sum_{k \,\varepsilon\, \text{H2}} w_{kl}\, y_k$$

$$y_k = f(z_k)$$

$$z_k = \sum_{j \,\varepsilon\, \text{H1}} w_{jk}\, y_j$$

$$y_j = f(z_j)$$

$$z_j = \sum_{i \,\varepsilon\, \text{Input}} w_{ij}\, x_i$$

Compare outputs with correct answer to get error derivatives

$$\frac{\partial E}{\partial y_l} = y_l - t_l$$

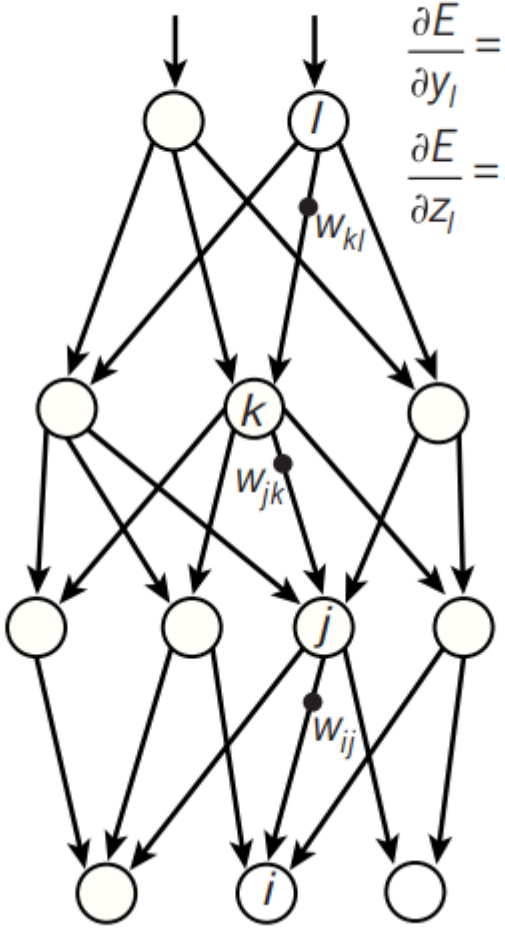
$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial y_k} = \sum_{l \,\varepsilon\, \text{out}} w_{kl}\frac{\partial E}{\partial z_l}$$

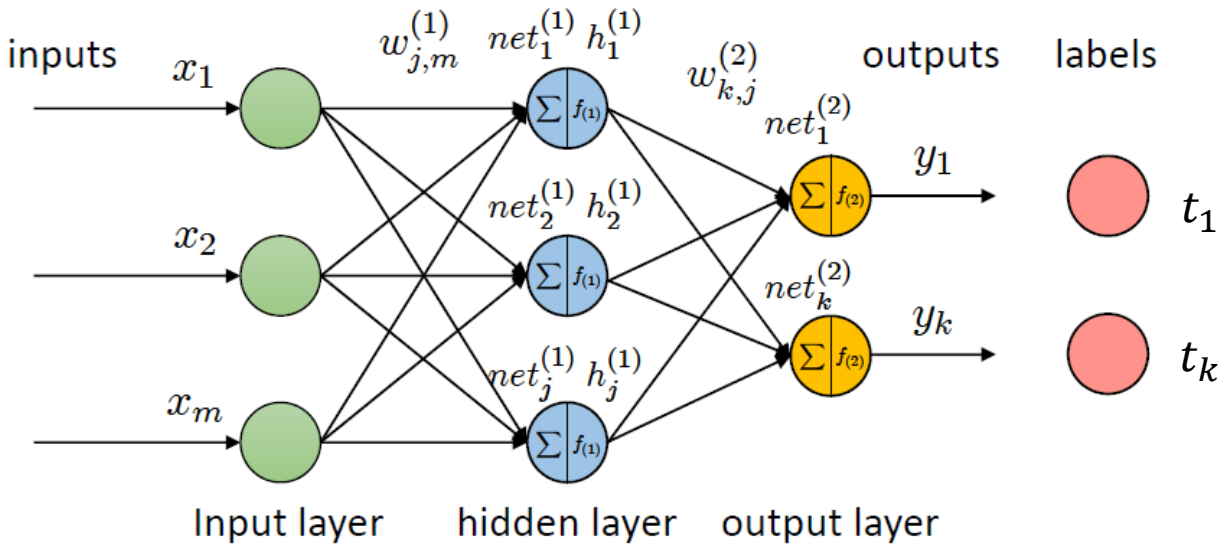
$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \,\varepsilon\, \text{H2}} w_{jk}\frac{\partial E}{\partial z_k}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$$



Make a prediction



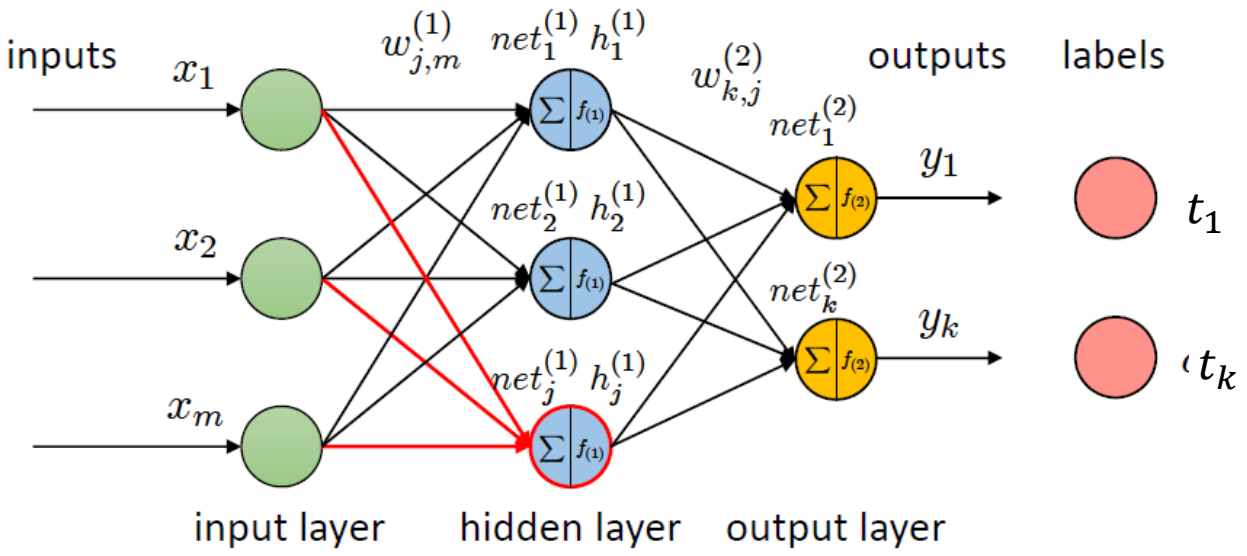
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)} x_m\right) \longrightarrow y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(1)} h_j^{(1)}\right)$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $\qquad$ $net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Make a prediction (cont.)



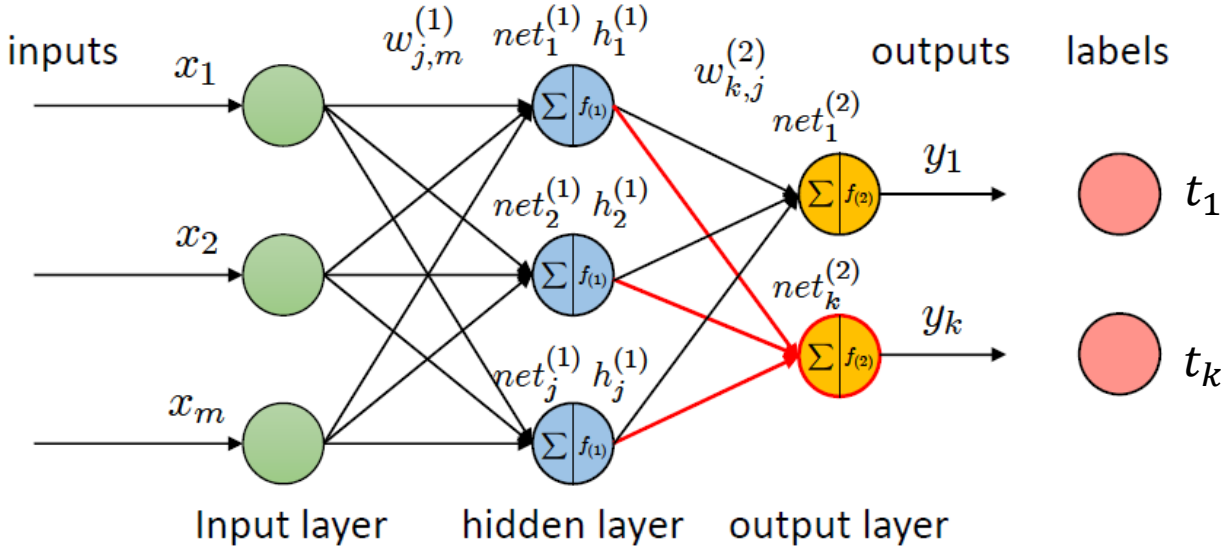
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \xrightarrow{h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}(\sum_m w_{j,m}^{(1)} x_m)} h_j^{(1)} \xrightarrow{y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}(\sum_j w_{k,j}^{(1)} h_j^{(1)})} y_k$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Make a prediction (cont.)



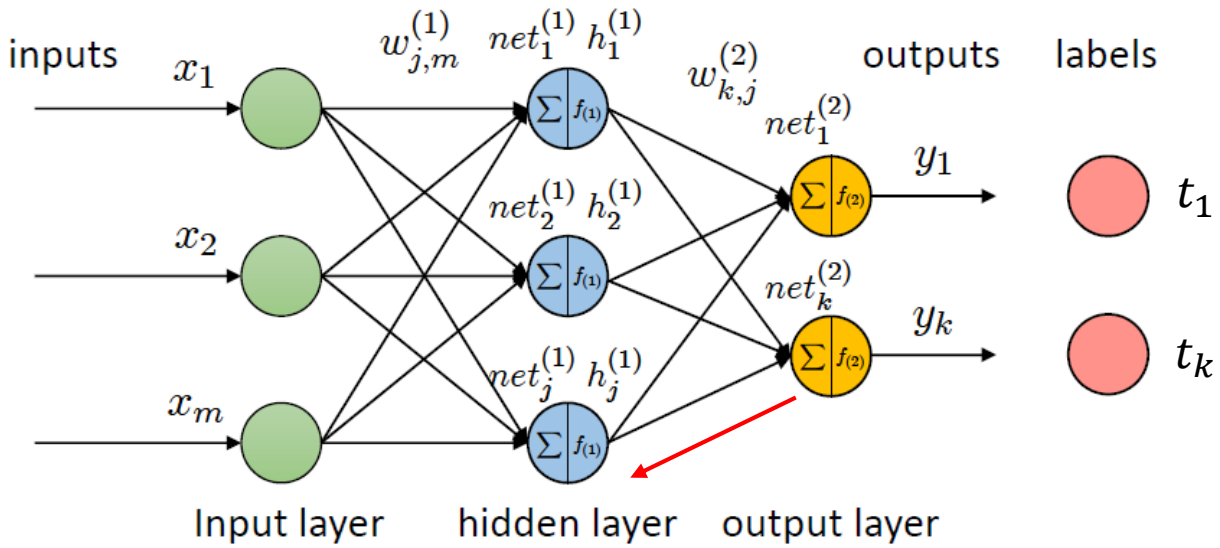
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)} x_m\right) \longrightarrow y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(1)} h_j^{(1)}\right) \longrightarrow y_k$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $\qquad$ $net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Backpropagation



Two-layer feedforward neural network

- Assume all the activation functions are sigmoid
- Error function $E = \frac{1}{2}\sum_k (y_k - t_k)^2$
- $\frac{\partial E}{\partial y_k} = y_k - t_k$
- $\frac{\partial y_k}{\partial w_{k,j}^{(2)}} = f'_{(2)}\left(net_k^{(2)}\right) h_j^{(1)} = y_k(1-y_k)h_j^{(1)}$
- $\Rightarrow \frac{\partial E}{\partial w_{k,j}^{(2)}} = -(t_k - y_k)y_k(1-y_k)h_j^{(1)}$
- $\Rightarrow w_{k,j}^{(2)} \leftarrow w_{k,j}^{(2)} + \eta \delta_k^{(2)} h_j^{(1)}$

$\delta_k^{(2)}$

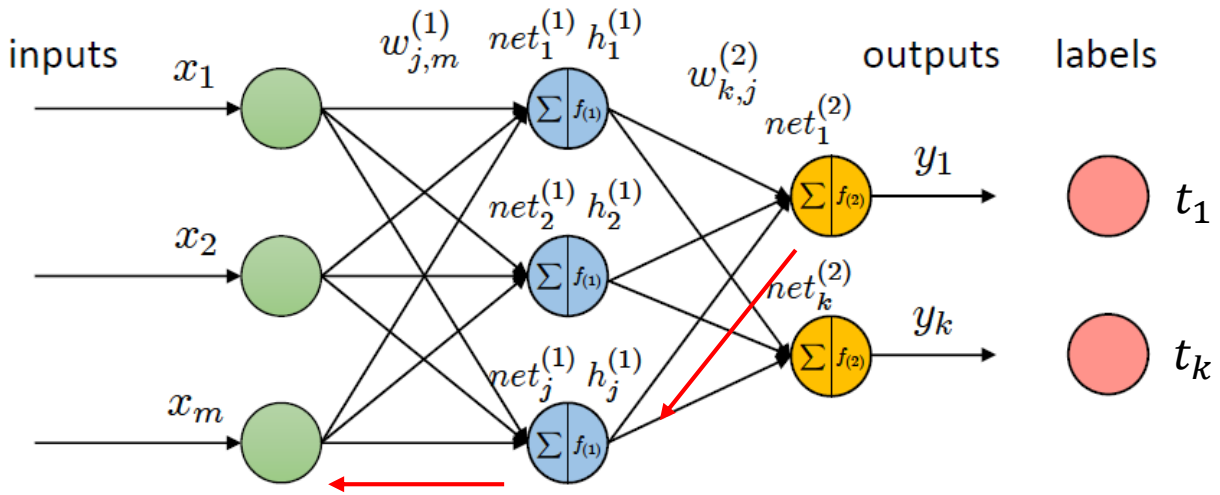
Output of unit j

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)} x_m\right) \longrightarrow y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(1)} h_j^{(1)}\right) \longrightarrow y_k$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $\quad net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Backpropagation (cont.)



Two-layer feedforward neural network

- Error function $E = \frac{1}{2}\sum_k (y_k - t_k)^2$
- $\frac{\partial E}{\partial y_k} = y_k - t_k$
- $\delta_k^{(2)} = (t_k - y_k) y_k (1 - y_k)$
- $\Rightarrow w_{k,j}^{(2)} \leftarrow w_{k,j}^{(2)} + \eta \delta_k^{(2)} h_j^{(1)}$
- $\frac{\partial y_k}{\partial h_j^{(1)}} = y_k(1 - y_k) w_{k,j}^{(2)}$
- $\frac{\partial h_j^{(1)}}{\partial w_{j,m}^{(1)}} = f'_{(1)}\left(net_j^{(1)}\right) x_m = h_j^{(1)}\left(1 - h_j^{(1)}\right) x_m$
- $\frac{\partial E}{\partial w_{j,m}^{(1)}} = -h_j^{(1)}\left(1 - h_j^{(1)}\right) \sum_k w_{k,j}^{(2)} (t_k - y_k) y_k (1 - y_k)\, x_m$

$$= -h_j^{(1)}\left(1 - h_j^{(1)}\right) \sum_k w_{k,j}^{(2)} \delta_k^{(2)}\, x_m$$

- $\Rightarrow w_{j,m}^{(1)} \leftarrow w_{j,m}^{(1)} + \eta \delta_j^{(1)} x_m$

$\delta_j^{(1)}$

Feed-forward prediction:

$$h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)} x_m\right) \qquad y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(1)} h_j^{(1)}\right)$$

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} \longrightarrow y_k$$

where $\quad net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m \qquad net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Backpropagation algorithms

- Error function $E = \frac{1}{2}\sum_k (y_k - t_k)^2$
- $\delta_k^{(2)} = (t_k - y_k) y_k (1 - y_k)$
- $\Rightarrow w_{k,j}^{(2)} \leftarrow w_{k,j}^{(2)} + \eta \delta_k^{(2)} h_j^{(1)}$
- $\delta_j^{(1)} = h_j^{(1)} \left(1 - h_j^{(1)}\right) \sum_k w_{k,j}^{(2)} \delta_k^{(2)}$
- $\Rightarrow w_{j,m}^{(1)} \leftarrow w_{j,m}^{(1)} + \eta \delta_j^{(1)} x_m$

- Activation function: sigmoid

Initialize all weights to small random numbers

Do until convergence

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

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

  3. For each hidden unit j, o_j is the output of unit j

$$\delta_j \leftarrow o_j(1 - o_j) \sum_{k \in next\ layer} w_{k,j} \delta_k$$

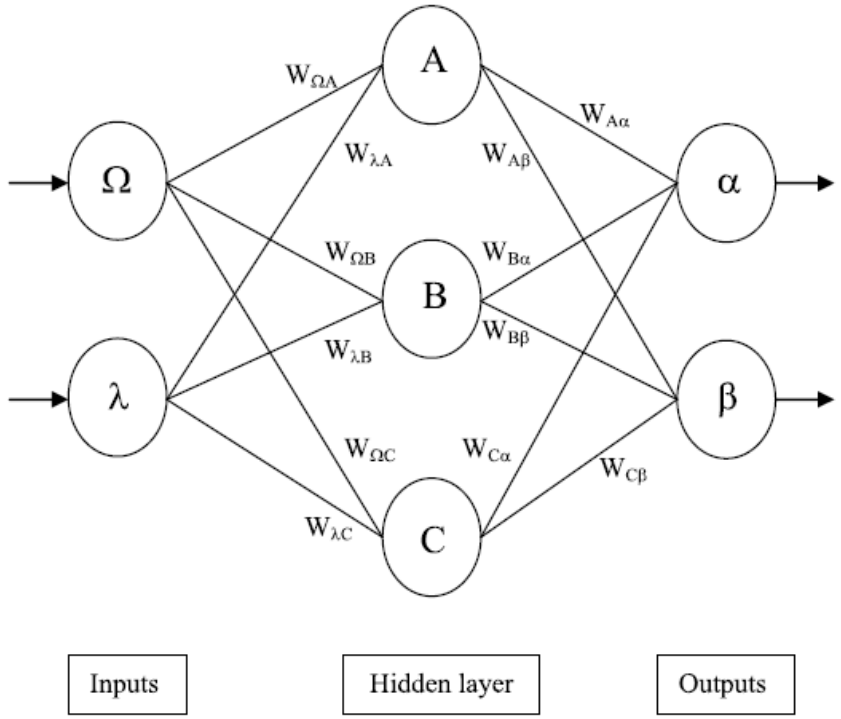
  4. Update each network weight, where x_i is the output for unit i

$$w_{j,i} \leftarrow w_{j,i} + \eta \delta_j x_i$$

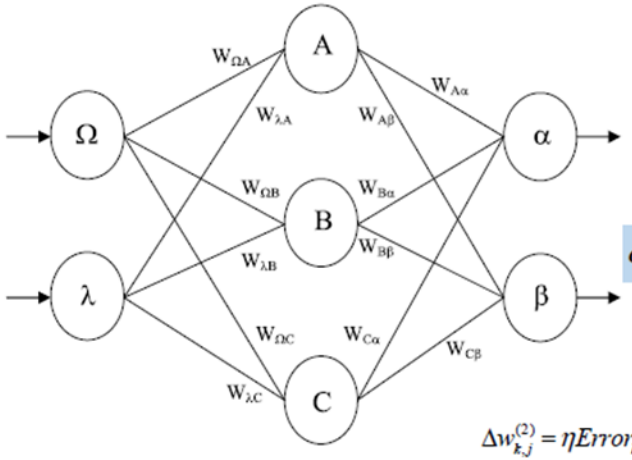
See the backpropagation demo

- https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/

Formula example for backpropagation



Formula example for backpropagation (cont.)



1. Calculate errors of output neurons

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

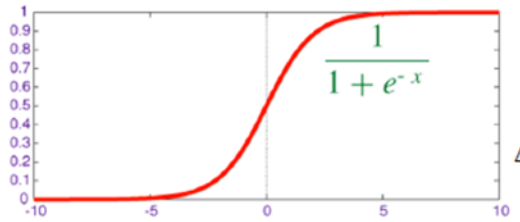
$$\delta_\alpha = out_\alpha\ (1 - out_\alpha)\ (Target_\alpha - out_\alpha)$$
$$\delta_\beta = out_\beta\ (1 - out_\beta)\ (Target_\beta - out_\beta)$$

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

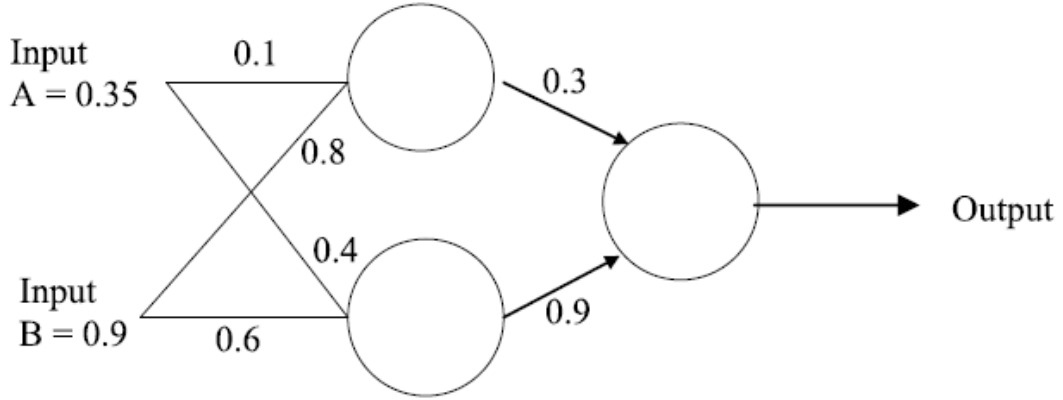
$$\Delta w_{j,m}^{(1)} = \eta Error_j Output_m = \eta \delta_j x_m$$

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

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

Calculation example

- Consider the simple network below:



- Assume that the neurons have sigmoid activation function and
  - Perform a forward pass on the network and find the predicted output
  - Perform a reverse pass (training) once (target = 0.5) with $\eta = 1$
  - Perform a further forward pass and comment on the result

Calculation example (cont.)

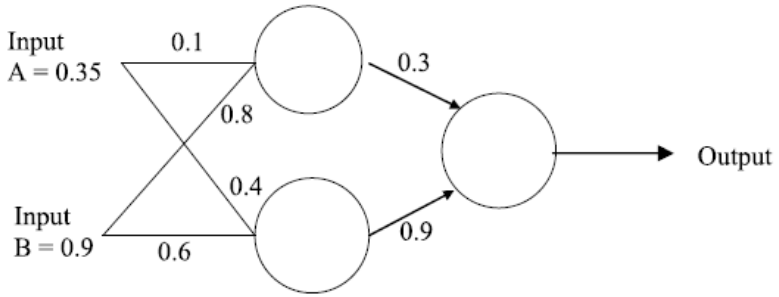
Answer:
(i)
Input to top neuron = (0.35x0.1)+(0.9x0.8)=0.755. Out = 0.68.
Input to bottom neuron = (0.9x0.6)+(0.35x0.4) = 0.68. Out = 0.6637.
Input to final neuron = (0.3x0.68)+(0.9x0.6637) = 0.80133. Out = 0.69.

(ii)
Output error δ=(t-o)(1-o)o = (0.5-0.69)(1-0.69)0.69 = -0.0406.

New weights for output layer
$w1^{+}$ = w1+(δ x input) = 0.3 + (-0.0406x0.68) = 0.272392.
$w2^{+}$ = w2+(δ x input) = 0.9 + (-0.0406x0.6637) = 0.87305.

Errors for hidden layers:
δ1 = δ x w1 = -0.0406 x 0.272392 x (1-o)o = -2.406×10^{-3}
δ2= δ x w2 = -0.0406 x 0.87305 x (1-o)o = -7.916×10^{-3}

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



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

- For each output unit k, o_k is the output of unit k

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

- For each hidden unit j, o_j is the output of unit j

$$\delta_j \leftarrow o_j(1 - o_j) \sum_{k \in next\ layer} w_{k,j}\delta_k$$

- Update each network weight, where x_i is the input for unit j

$$w_{j,i} \leftarrow w_{j,i} + \eta\delta_j x_i$$

Calculation example (cont.)

- Answer (i)
  - Input to top neuron $= 0.35 \times 0.1 + 0.9 \times 0.8 = 0.755$. Out=0.68
  - Input to bottom neuron $= 0.35 \times 0.4 + 0.9 \times 0.6 = 0.68$. Out= 0.6637
  - Input to final neuron $= 0.3 \times 0.68 + 0.9 \times 0.6637 = 0.80133$. Out= 0.69
- (ii) It is both OK to use new or old weights when computing δ_j for hidden units
  - Output error $\delta = (t - o)o(1 - o) = (0.5 - 0.69) \times 0.69 \times (1 - 0.69) = -0.0406$
  - Error for top hidden neuron $\delta_1 = 0.68 \times (1 - 0.68) \times 0.3 \times (-0.0406) = -0.00265$
  - Error for top hidden neuron $\delta_2 = 0.6637 \times (1 - 0.6637) \times 0.9 \times (-0.0406) = -0.008156$
  - New weights for the output layer
  - $w_{o1} = 0.3 - 0.0406 \times 0.68 = 0.272392$
  - $w_{o2} = 0.9 - 0.0406 \times 0.6637 = 0.87305$
  - New weights for the hidden layer
  - $w_{1A} = 0.1 - 0.00265 \times 0.35 = 0.0991$
  - $w_{1B} = 0.8 - 0.00265 \times 0.9 = 0.7976$
  - $w_{2A} = 0.4 - 0.008156 \times 0.35 = 0.3971$
  - $w_{2B} = 0.6 - 0.008156 \times 0.9 = 0.5927$
- (iii)
  - Input to top neuron $= 0.35 \times 0.0991 + 0.9 \times 0.7976 = 0.7525$. Out=0.6797
  - Input to bottom neuron $= 0.35 \times 0.3971 + 0.9 \times 0.5927 = 0.6724$. Out= 0.662
  - Input to final neuron $= 0.272392 \times 0.6797 + 0.87305 \times 0.662 = 0.7631$. Out= 0.682
  - New error is -0.182, which is reduced compared to old error -0.19

- For each output unit k, o_k is the output of unit k

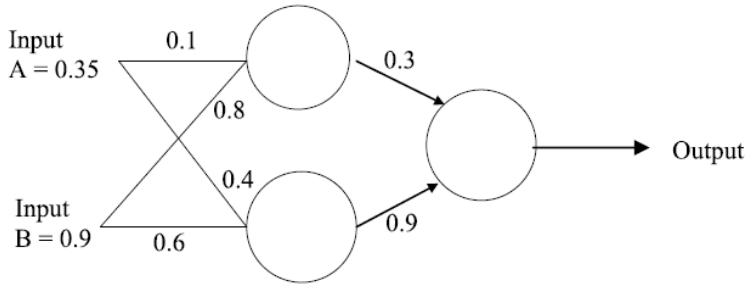
$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

- For each hidden unit j, o_j is the output of unit j

$$\delta_j \leftarrow o_j(1 - o_j) \sum_{k \in next\ layer} w_{k,j}\delta_k$$

- Update each network weight, where x_i is the input for unit j

$$w_{j,i} \leftarrow w_{j,i} + \eta\delta_j x_i$$



More on backpropagations

- It is gradient descent over entire network weight vectors
- 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

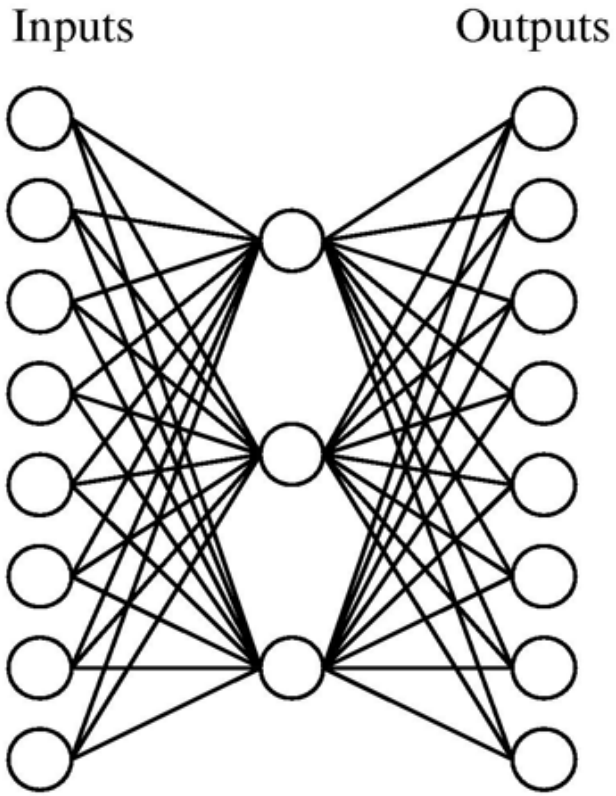
$$\Delta w_{j,i}(n) \leftarrow \eta \delta_j x_i + \alpha \Delta w_{j,i}(n-1)$$

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

An Training Example

Network structure

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



Input and output

- A target function:

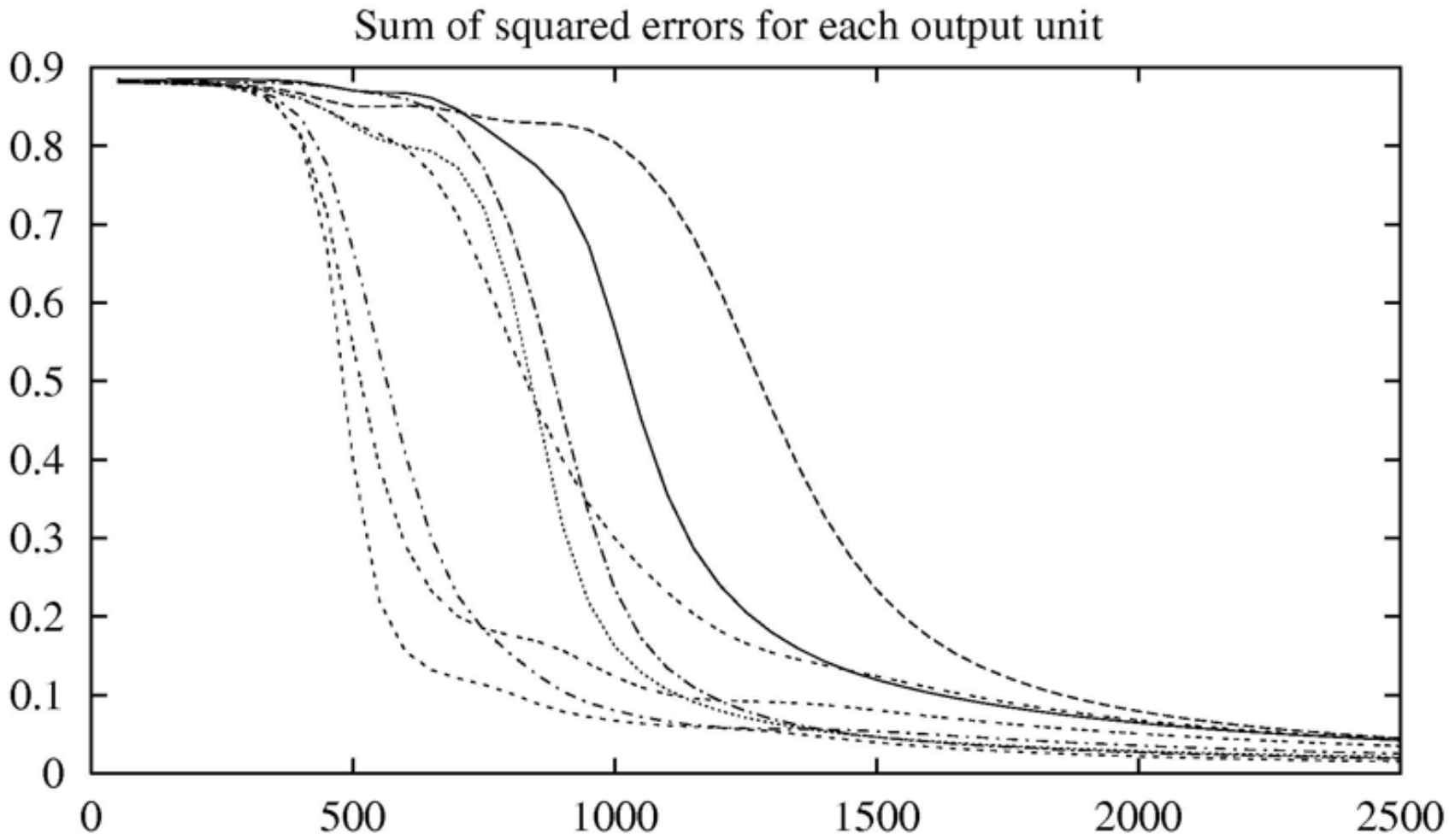
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

- Can this be learned?

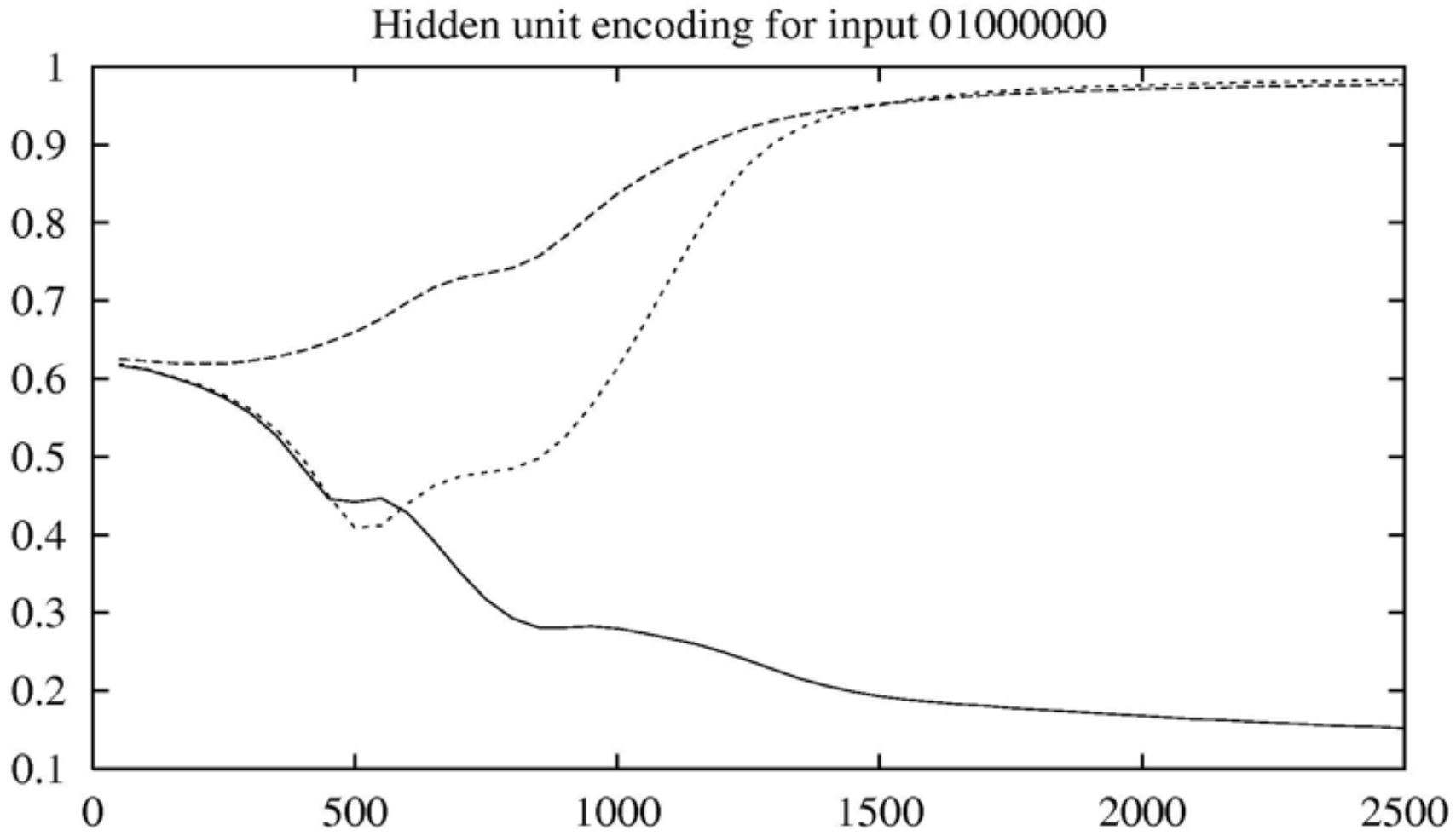
Learned hidden layer

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

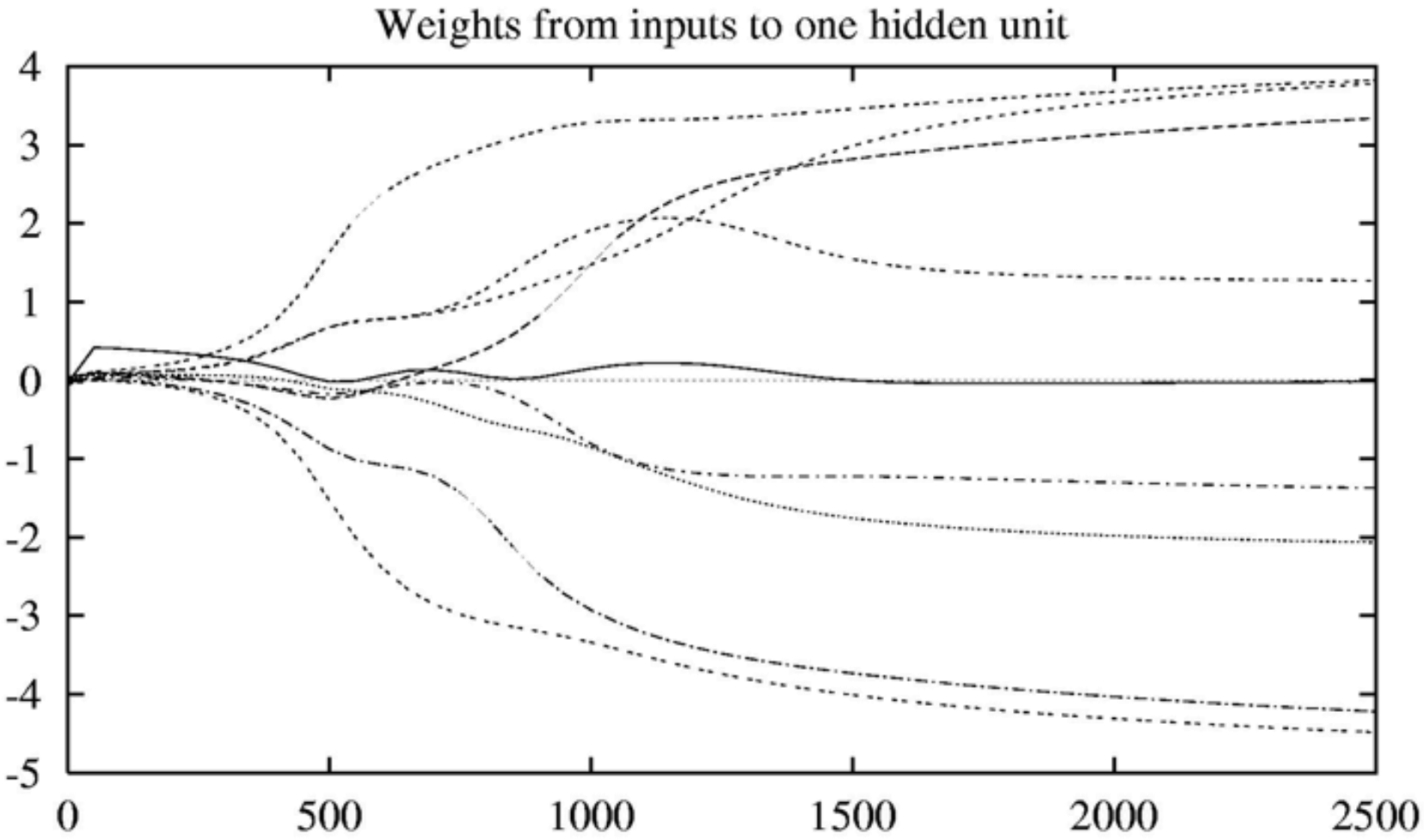
Convergence of sum of squared errors



Hidden unit encoding for 01000000



Convergence of first layer weights



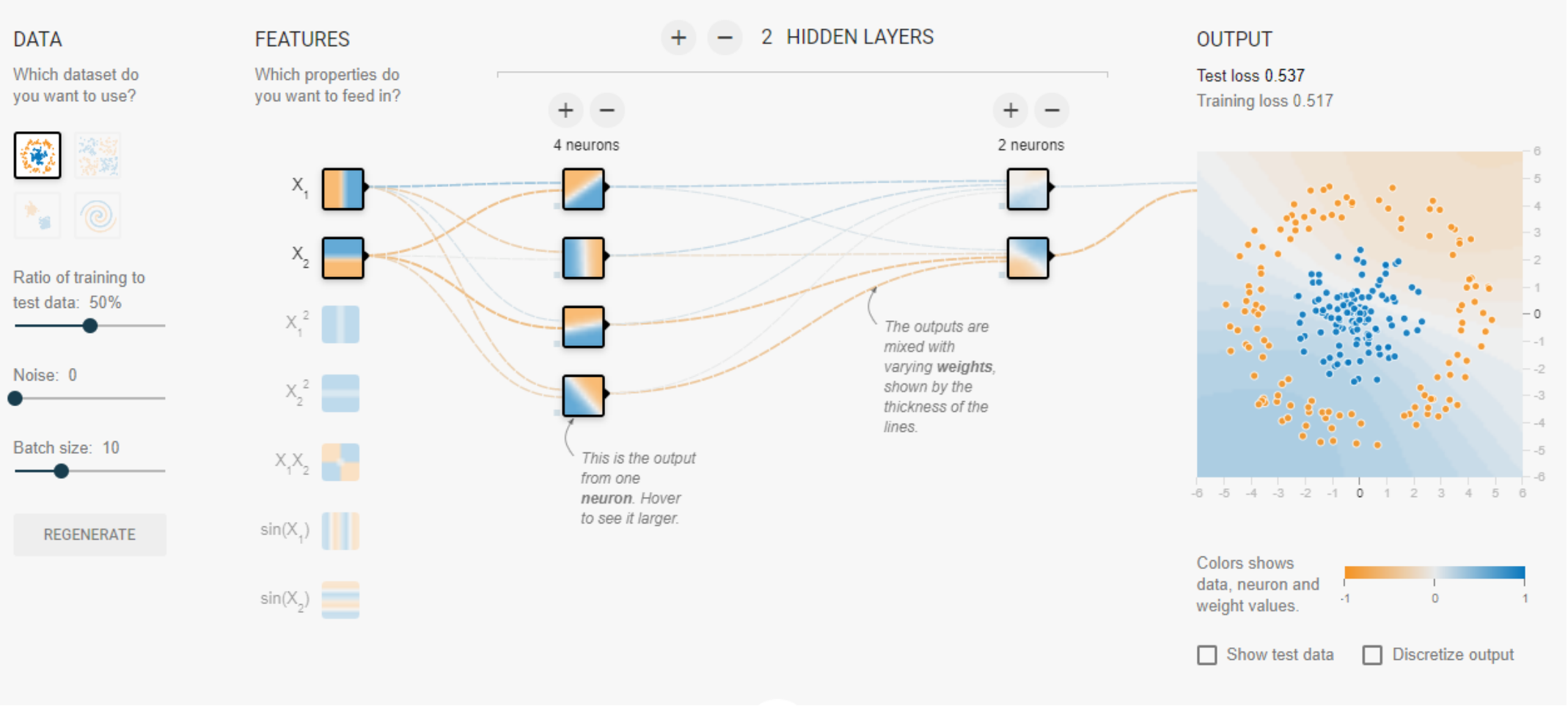
Convergence of backpropagation

- 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 approximate non-linear functions 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

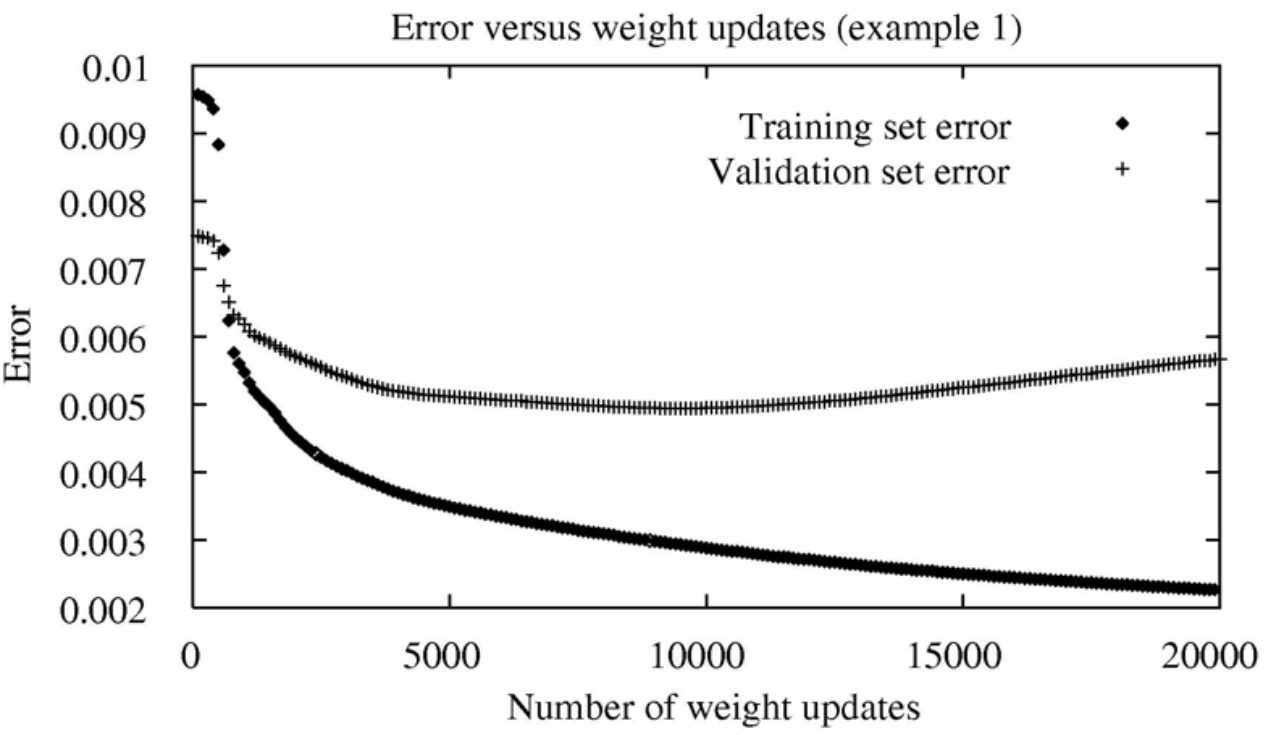
Tensorflow playground

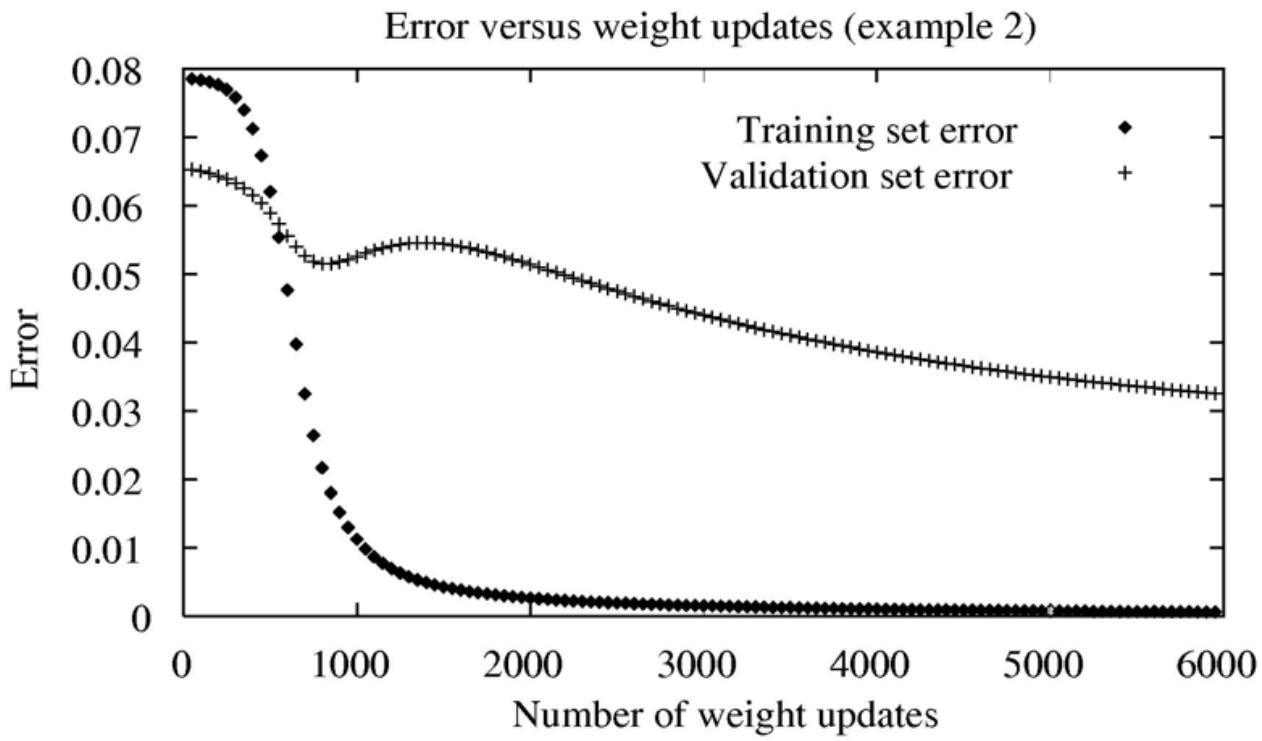


- http://playground.tensorflow.org/

Overfitting

Two examples of overfitting





Avoid overfitting

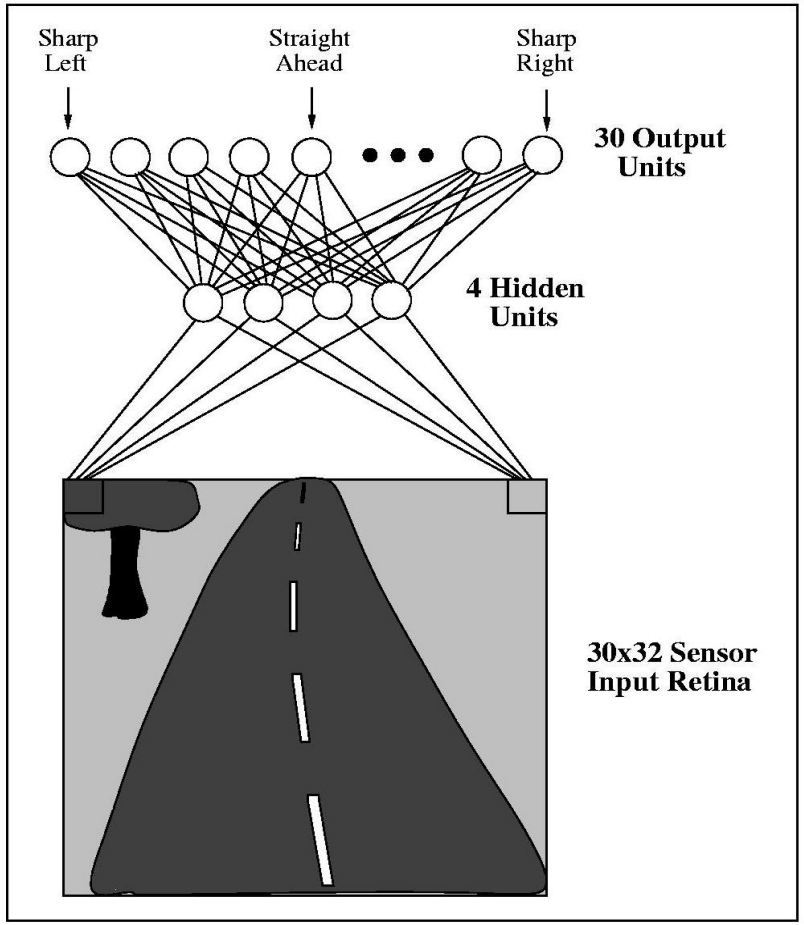
- Penalize large weights: $$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$
- Train on target slopes as well as values: $$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$
- Weight sharing
  - Reduce total number of weights
  - Using structures, like convolutional neural networks
- Early stopping
- Dropout

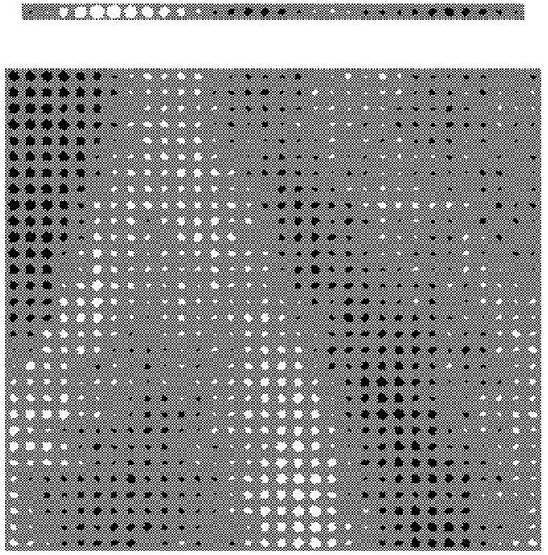
Applications

Autonomous self-driving cars

- Use neural network to summarize observation information and learn path planning

Autonomous self-driving cars (cont.)





A recipe for training neural networks

- By Andrej Karpathy
- https://karpathy.github.io/2019/04/25/recipe/