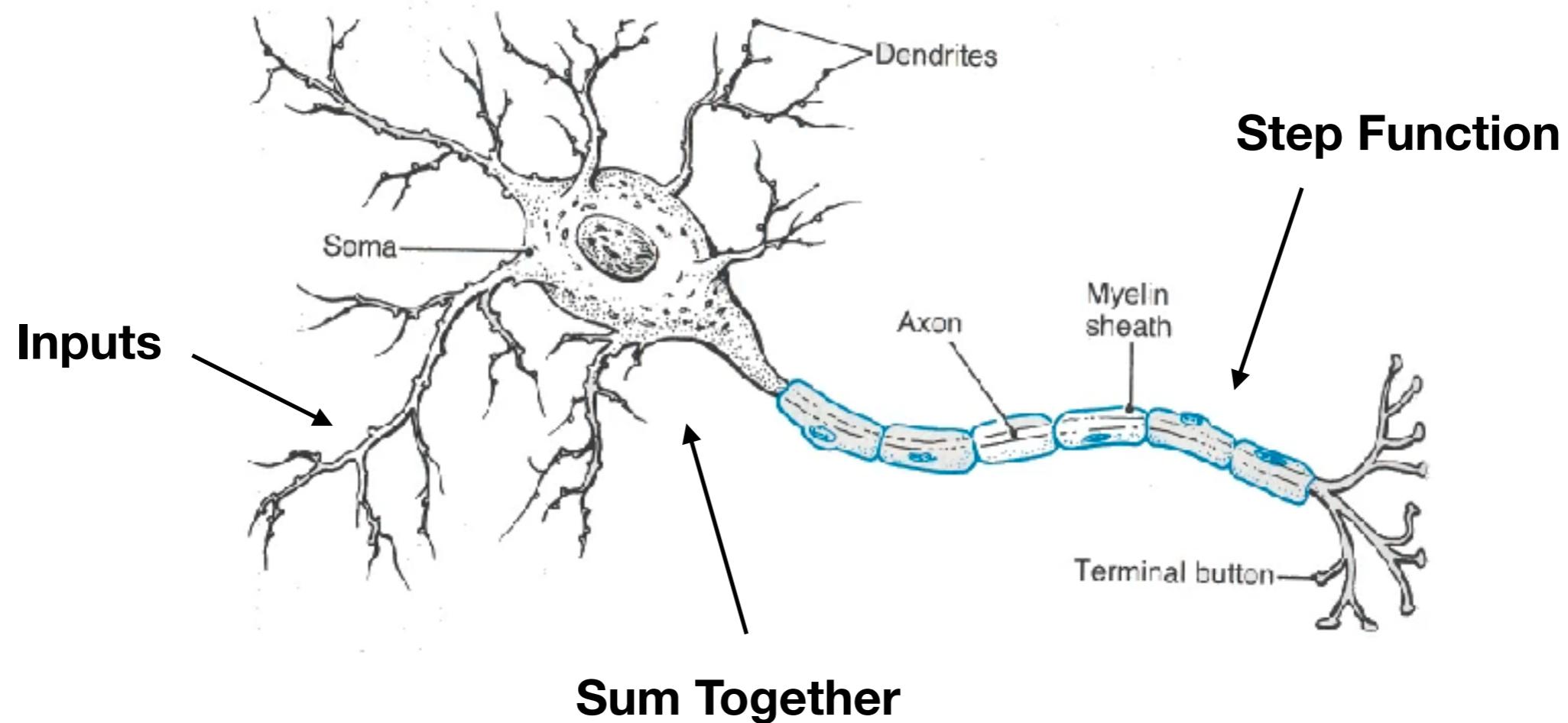
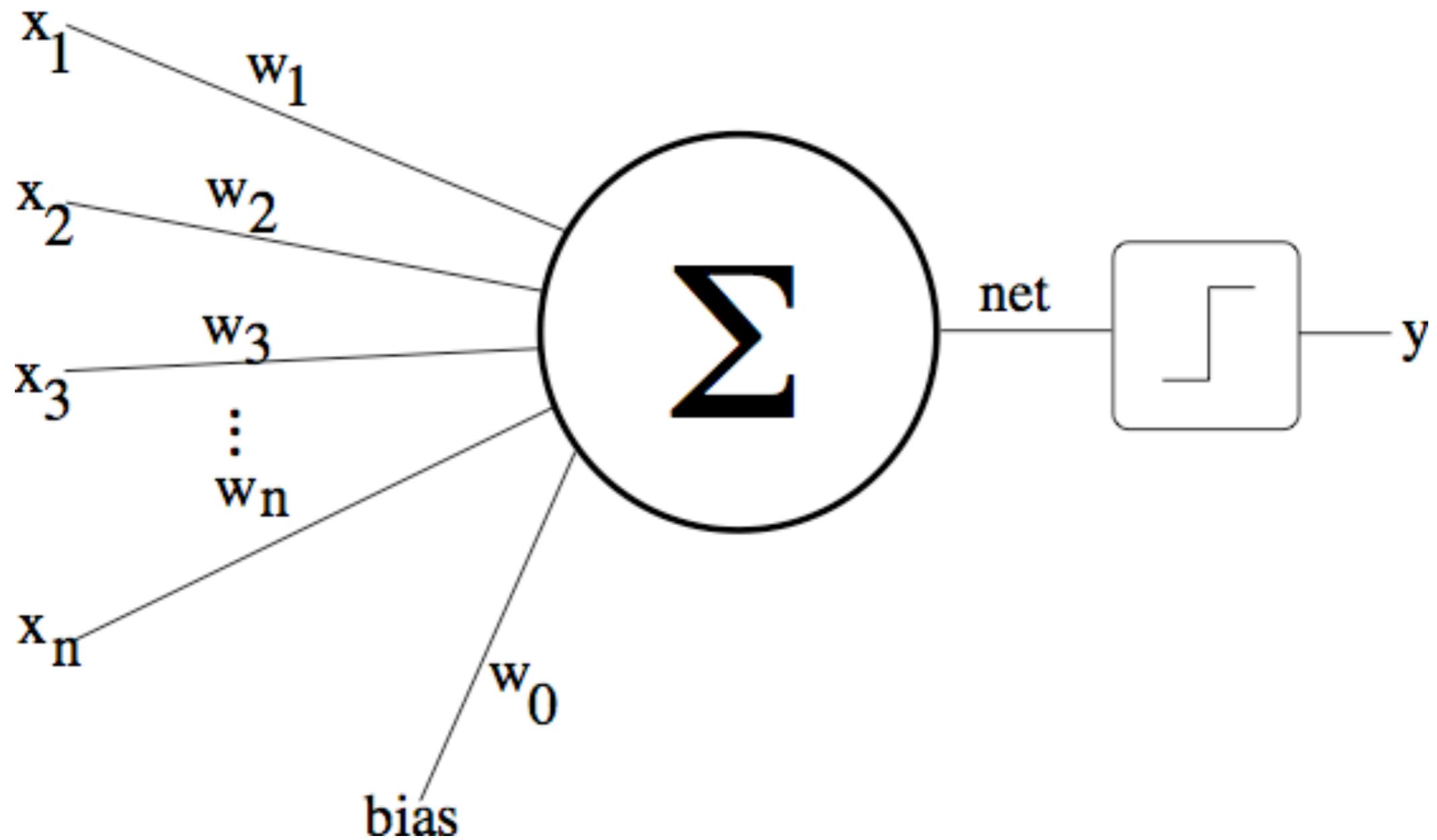


Lecture 9: Multilayer Perceptrons

Artificial Intelligence
CS-GY-6613
Julian Togelius
julian.togelius@nyu.edu

The Neuron

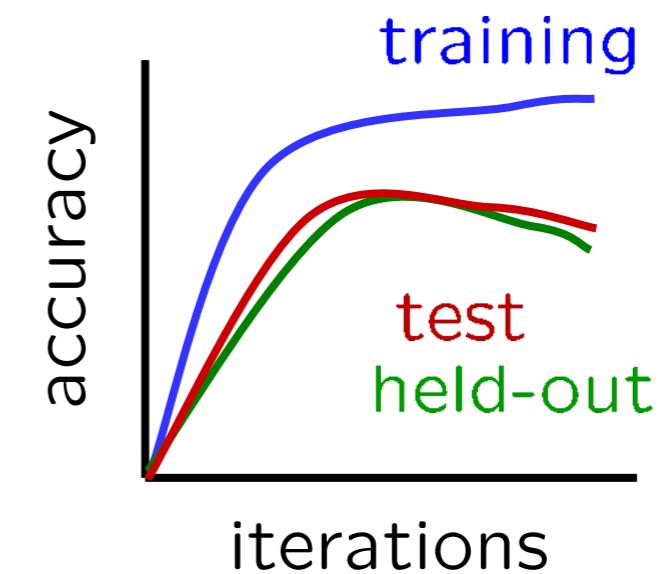
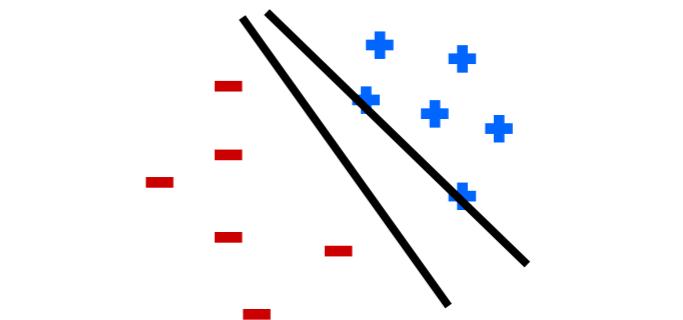
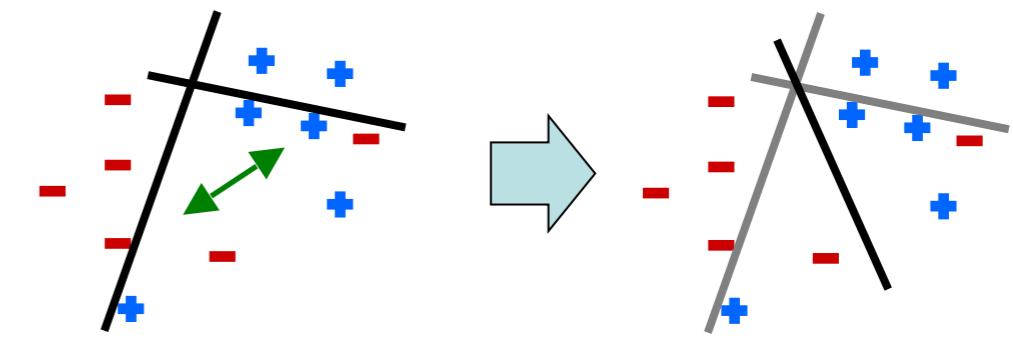




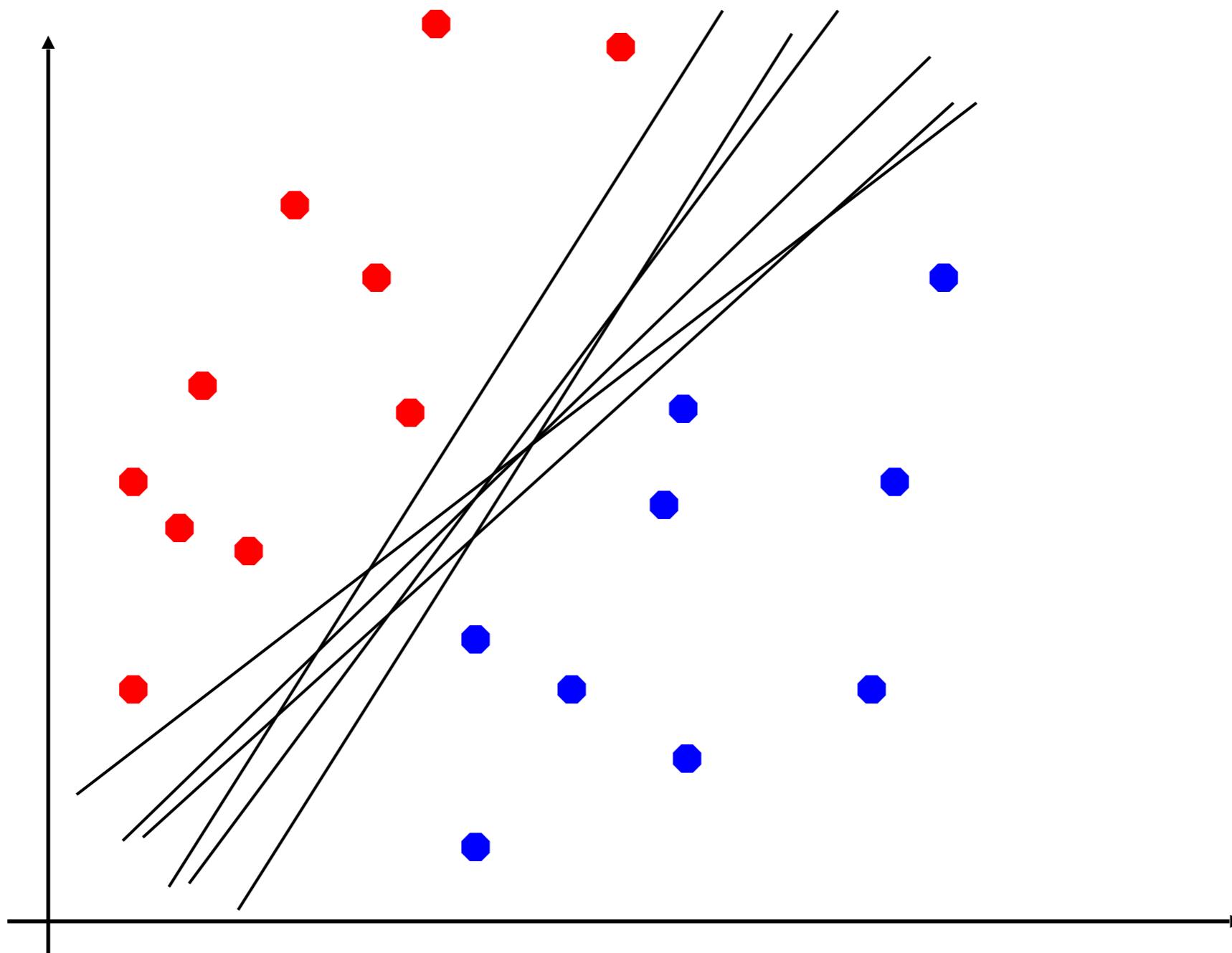
$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$

Perceptron problems

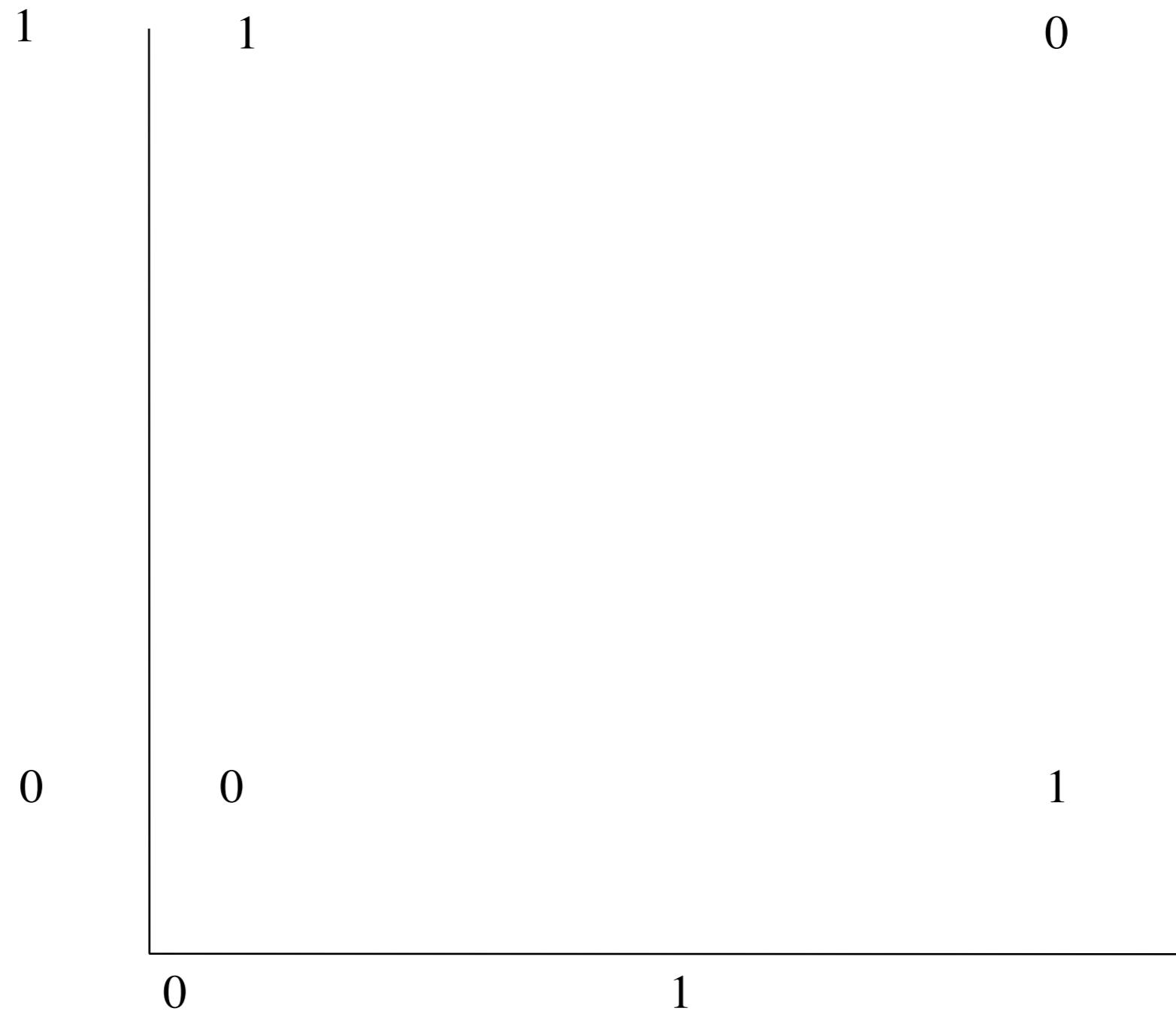
- Noise: if the data isn't separable, weights might thrash
- Mediocre generalization: finds a “barely” separating solution
- Overtraining: test / held-out accuracy usually rises, then falls



Which of these separators is optimal?

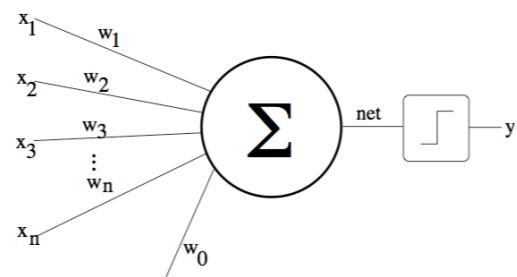


Problem: learn this!

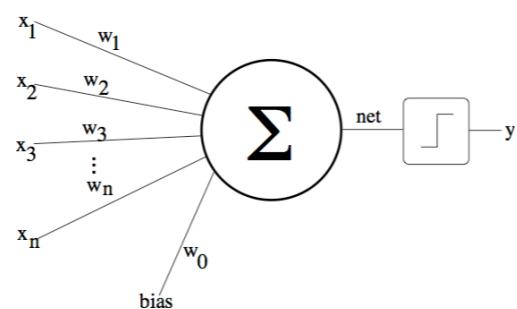


With all of these problems and with so many other learning algorithms, why were people still interested in pursuing this technique?

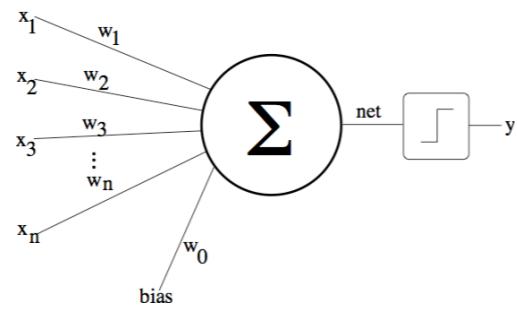
Could we just train a bunch of perceptrons connected to each other?



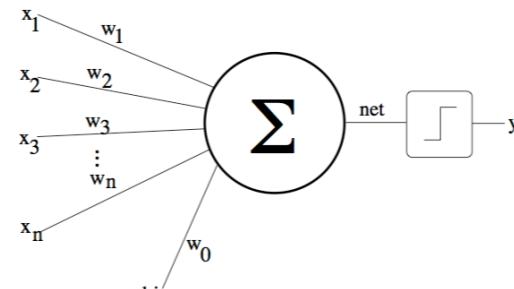
$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$



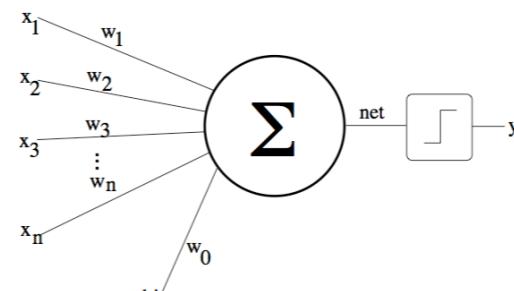
$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$



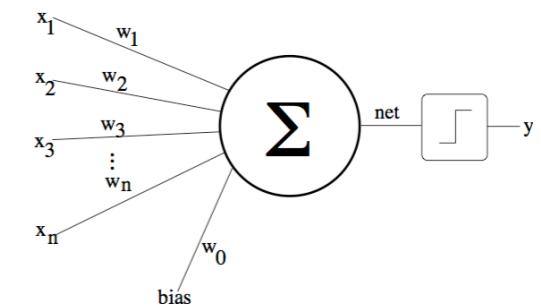
$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$



$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$



$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$



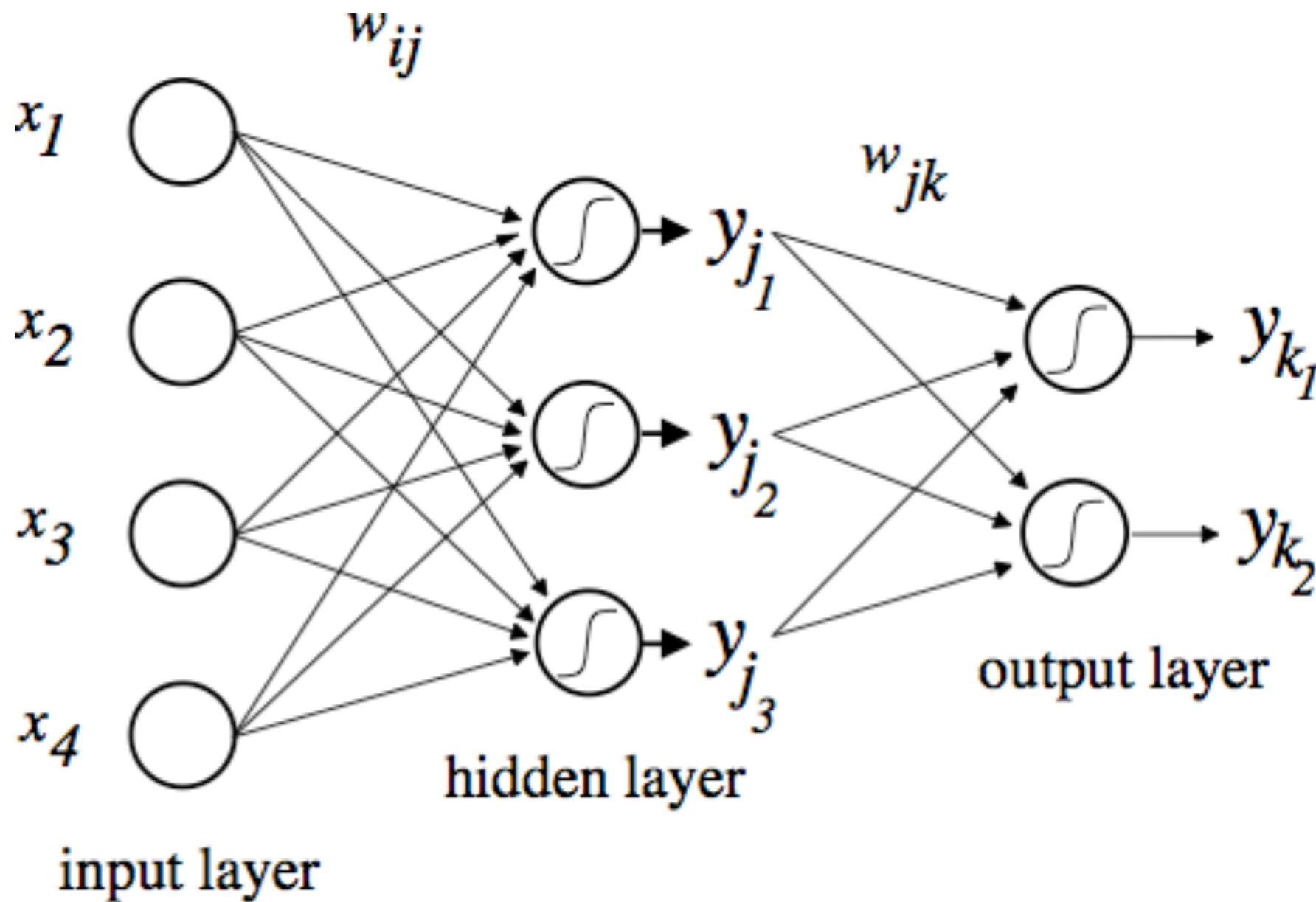
$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{else} \end{cases}$$

Multilayer Perceptron

“When an axon of cell A is near enough cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased”

–Donald Hebb

Multi-layer Perceptron (MLP)



Universal Approximator

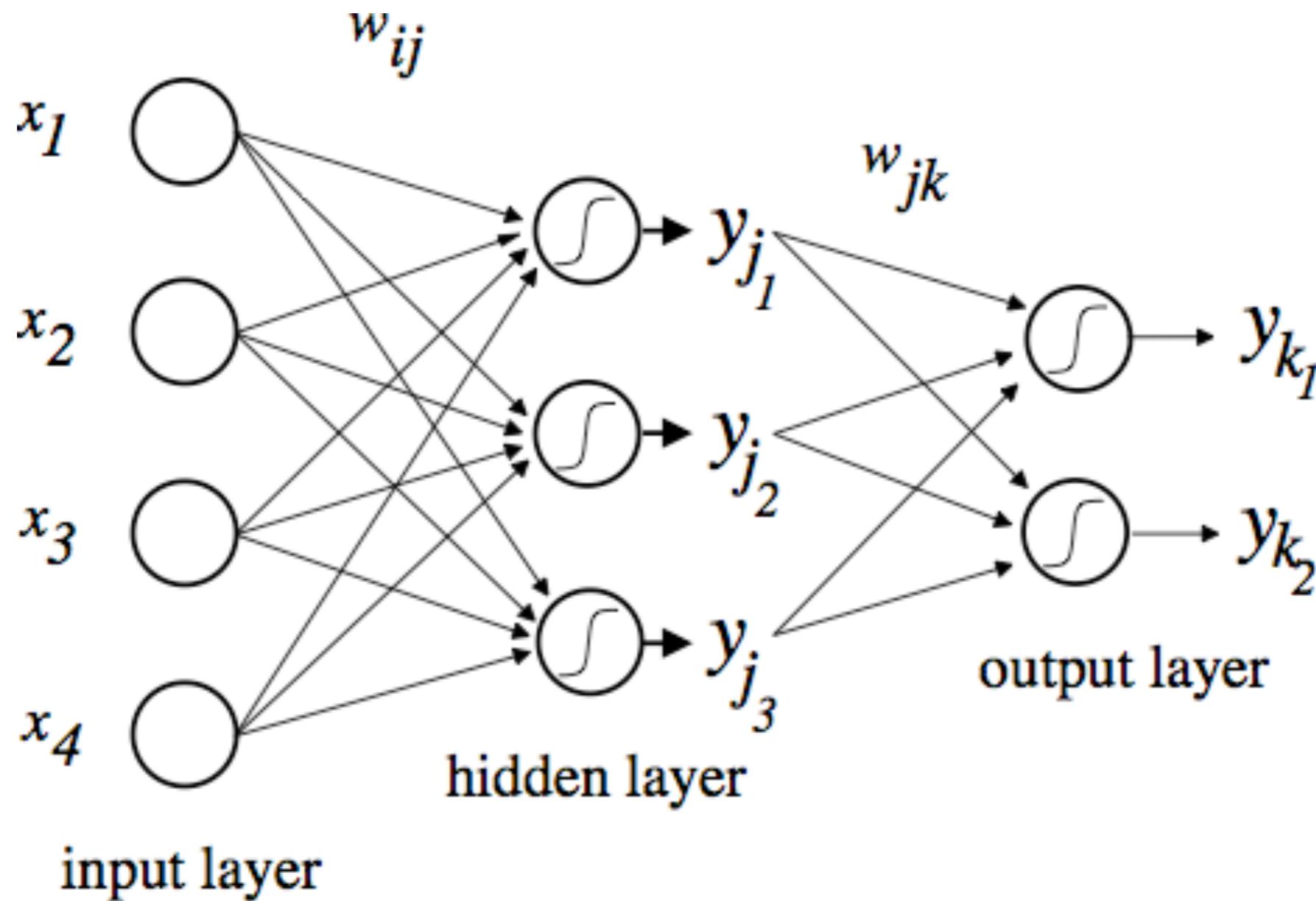
- George Cybenko in 1989 provided one of the first proofs for the Universal Approximation Theorem for Neural Networks
- States that a feedforward network with at least one hidden layer of a finite size can approximate any continuous function on compact subsets of R^n
- Says nothing about learnability
 - How do you train a hidden layer? What is its loss?

Training a Neural Network

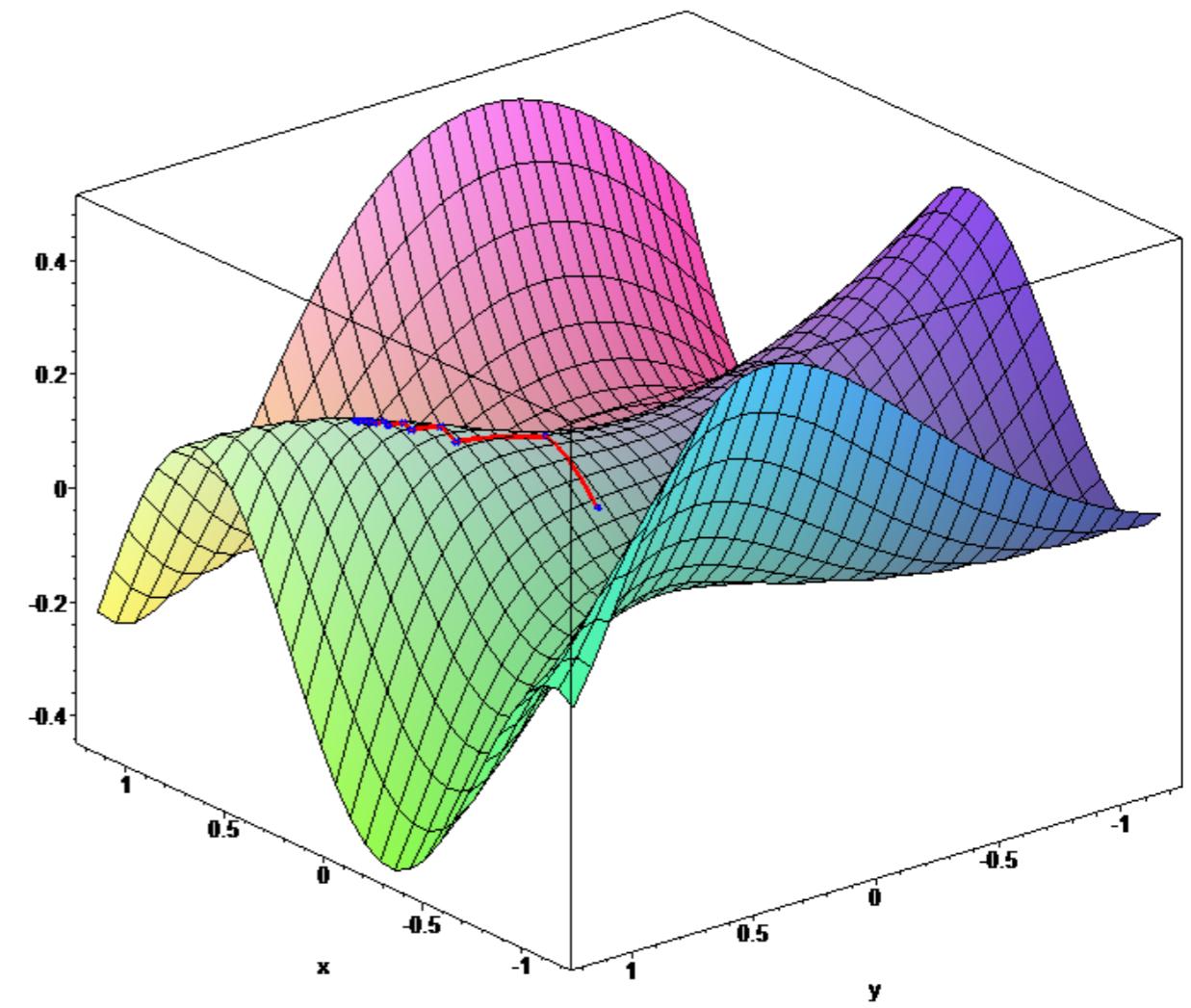
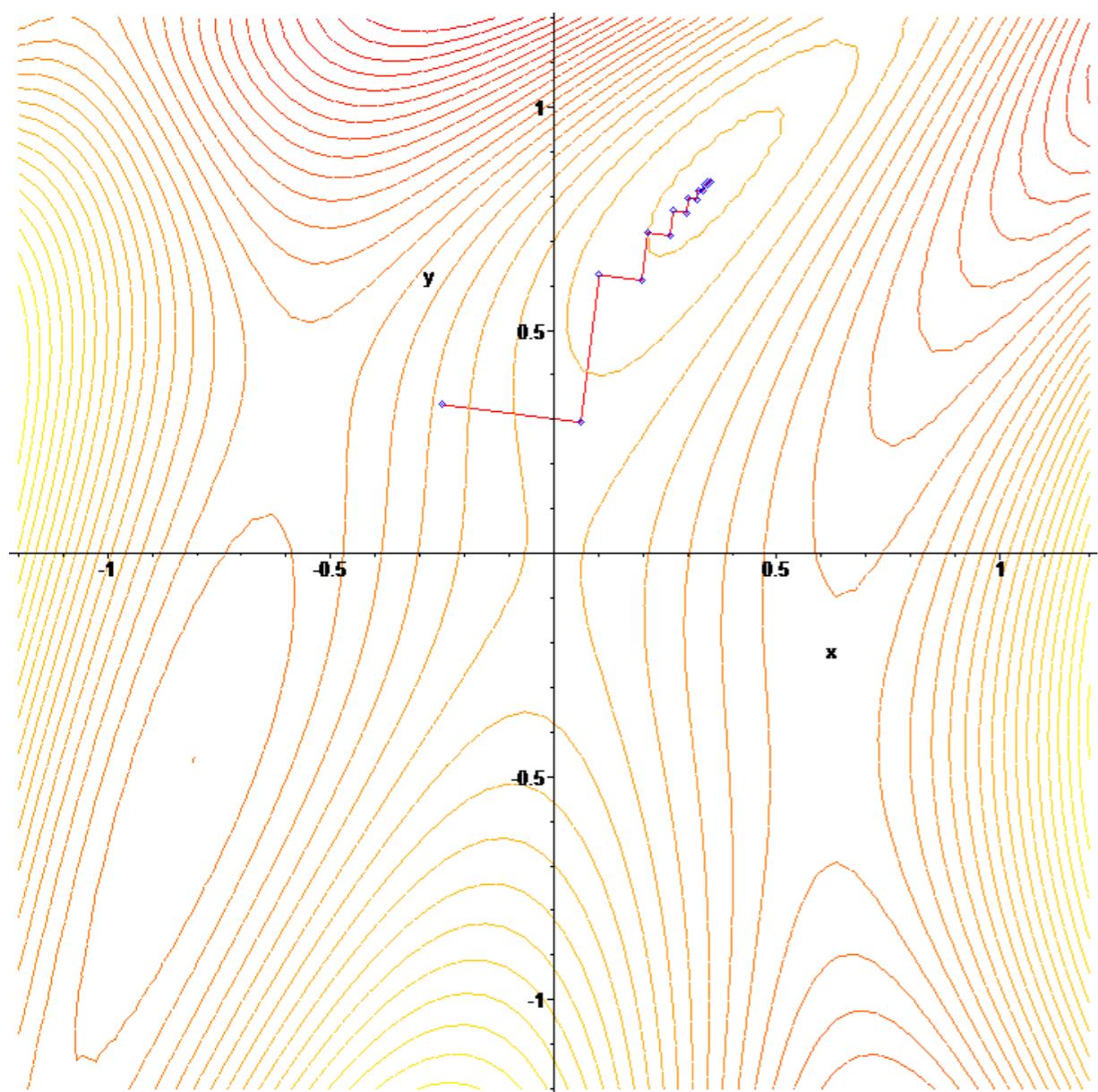
Backpropagation

- Forward Pass: present training input pattern to network and activate network to produce output (can also do in batch: present all patterns in succession)
- Backward Pass: calculate error gradient and update weights starting at output layer and then going back

Multi-layer Perceptron (MLP)



Gradient Descent



Gradient Descent

Input : list of n training examples $(x_0, d_0), \dots, (x_n, d_n)$
where $\forall i : d_i \in \{+1, -1\}$

Output : classifying hyperplane w

Algorithm :

Randomly initialize w ;

While makes errors on training set **do**

for (x_i, d_i) **do**

 let $y_i = \text{MLP}(w, x_i)$;

$loss \leftarrow \text{Mean}((d_i - y_i)^2)$

$w' \leftarrow \text{Backprop}(w, loss)$

$w \leftarrow w - \eta w'$

end

end

*x and w are vectors;
i is the instance index*

Building a Neural Network

Weight Update Equation

$$\theta_w' = \theta_w - \eta * \frac{\partial \text{TC}}{\partial \theta_w}$$



New weight



Learning Rate

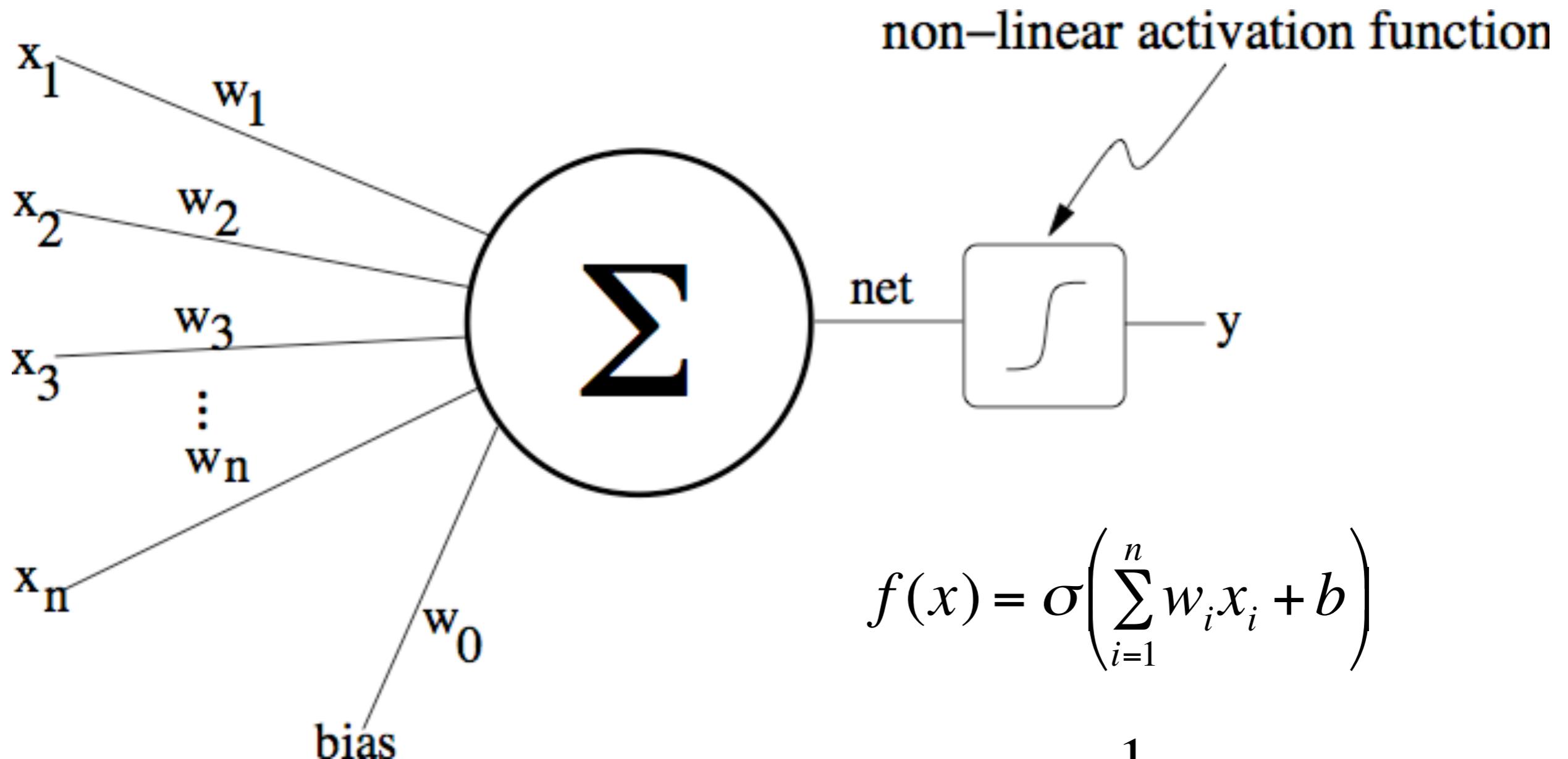
```

function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector  $\mathbf{x}$  and output vector  $\mathbf{y}$ 
  network, a multilayer network with  $L$  layers, weights  $w_{i,j}$ , activation function  $g$ 
  local variables:  $\Delta$ , a vector of errors, indexed by network node

  repeat
    for each weight  $w_{i,j}$  in network do
       $w_{i,j} \leftarrow$  a small random number
    for each example  $(\mathbf{x}, \mathbf{y})$  in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node  $i$  in the input layer do
         $a_i \leftarrow x_i$ 
      for  $\ell = 2$  to  $L$  do
        for each node  $j$  in layer  $\ell$  do
           $in_j \leftarrow \sum_i w_{i,j} a_i$ 
           $a_j \leftarrow g(in_j)$ 
      /* Propagate deltas backward from output layer to input layer */
      for each node  $j$  in the output layer do
         $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$ 
      for  $\ell = L - 1$  to 1 do
        for each node  $i$  in layer  $\ell$  do
           $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ 
      /* Update every weight in network using deltas */
      for each weight  $w_{i,j}$  in network do
         $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ 
  until some stopping criterion is satisfied
  return network

```

Activation Functions



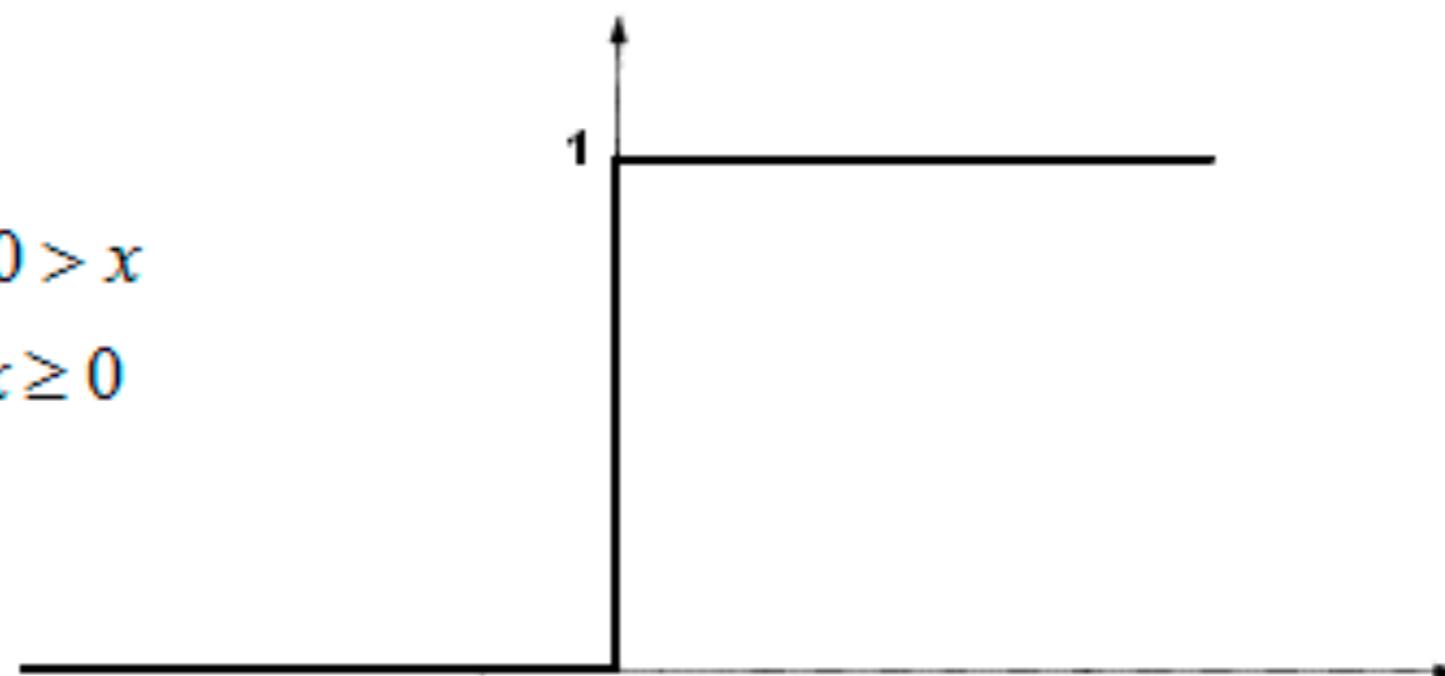
$$f(x) = \sigma\left(\sum_{i=1}^n w_i x_i + b\right)$$

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

Activation Functions

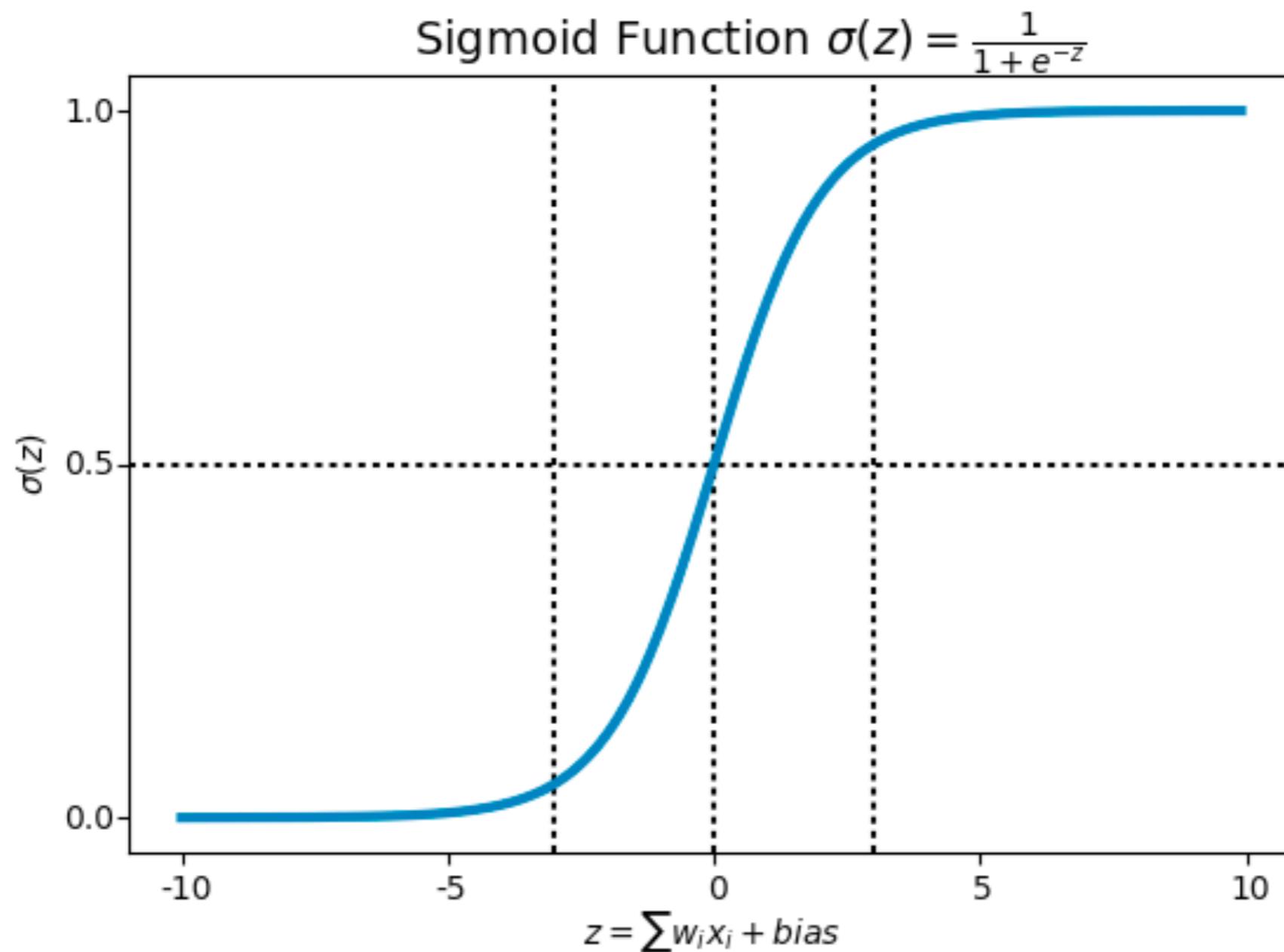
Unit step (threshold)

$$f(x) = \begin{cases} 0 & \text{if } x > 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



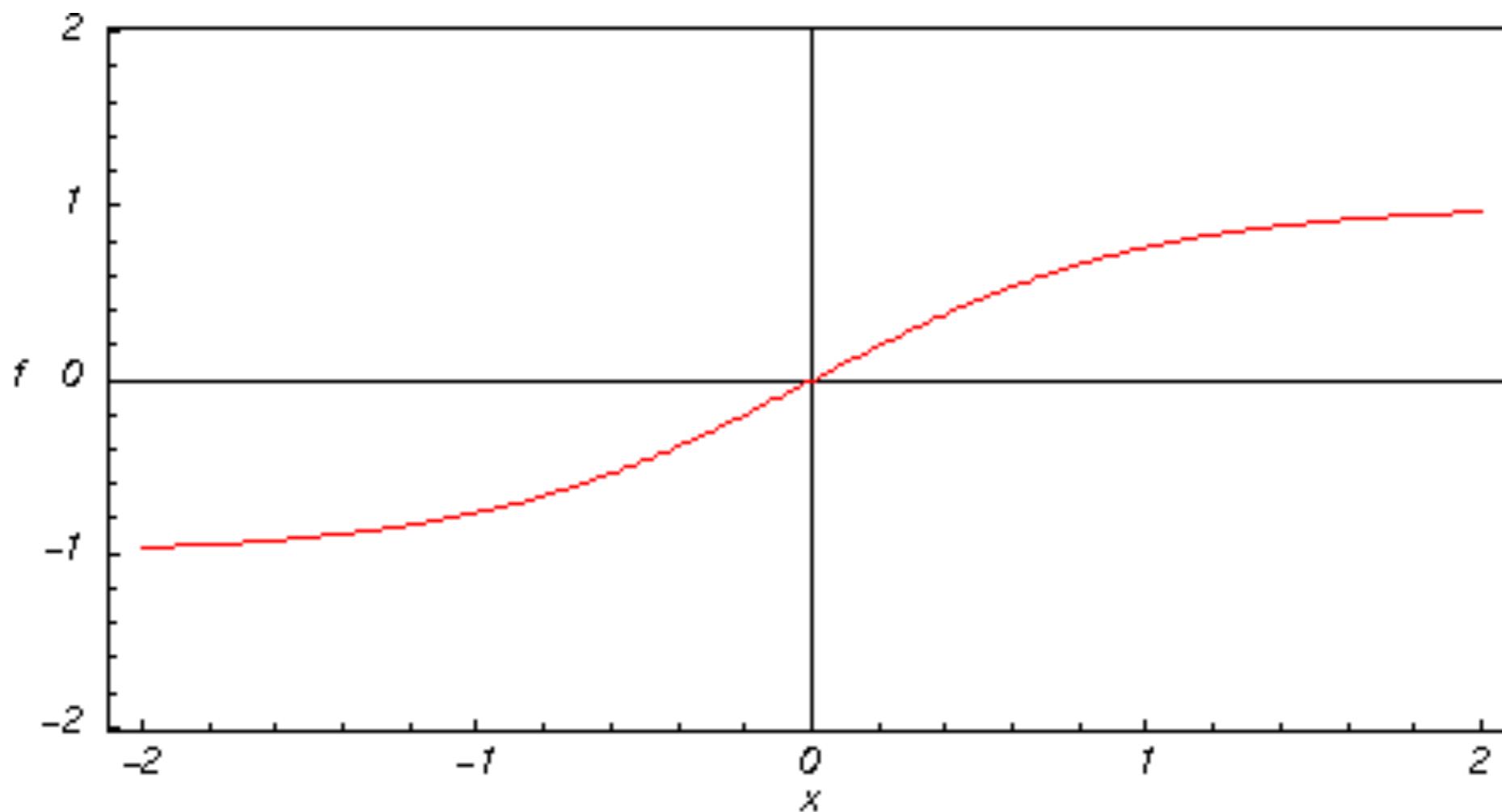
Derivate: N/A

Activation Functions



Derivative: $\sigma'(z) = \sigma(z)(1-\sigma(z))$

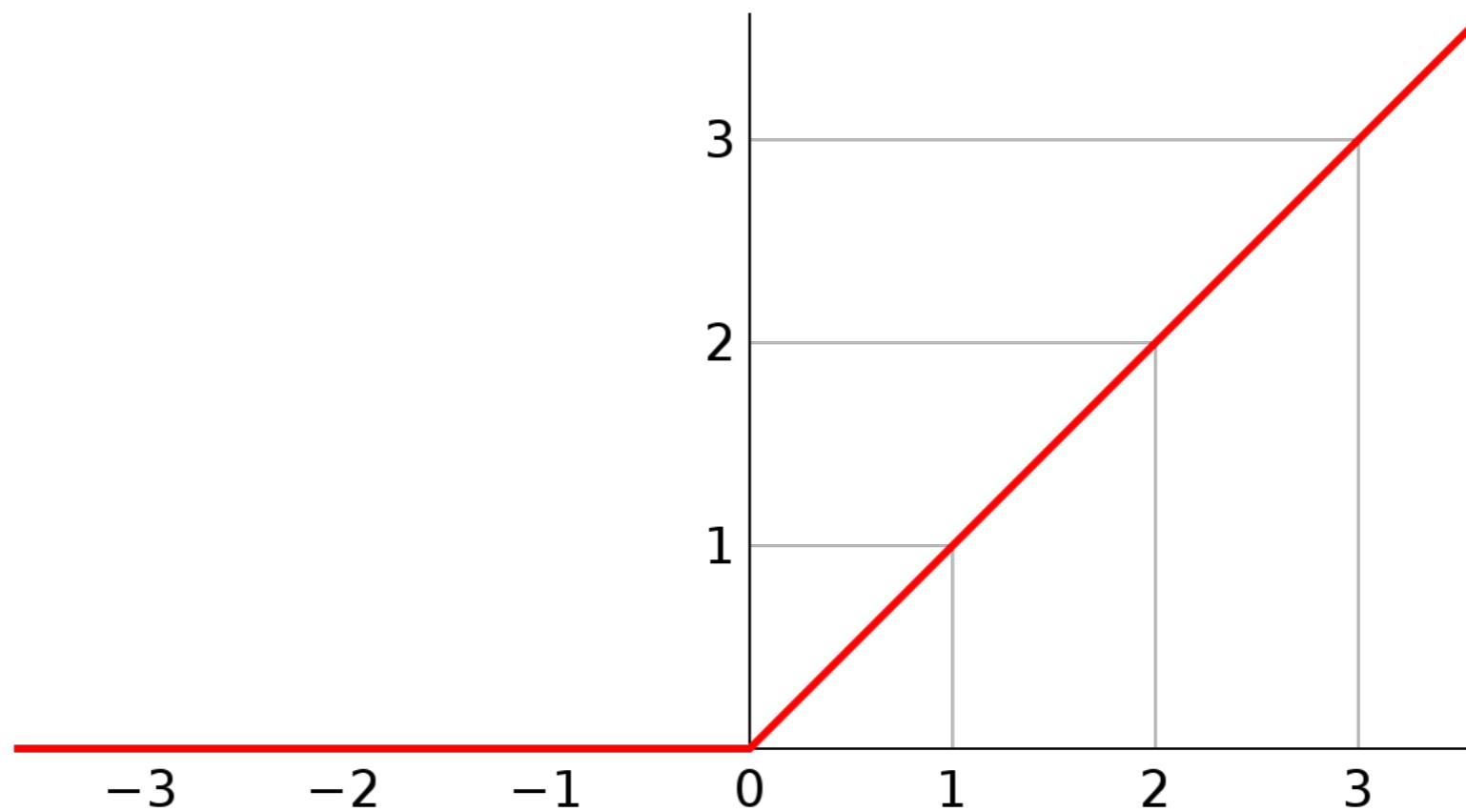
Activation Functions



Hyperbolic Tangent

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

Activation Functions

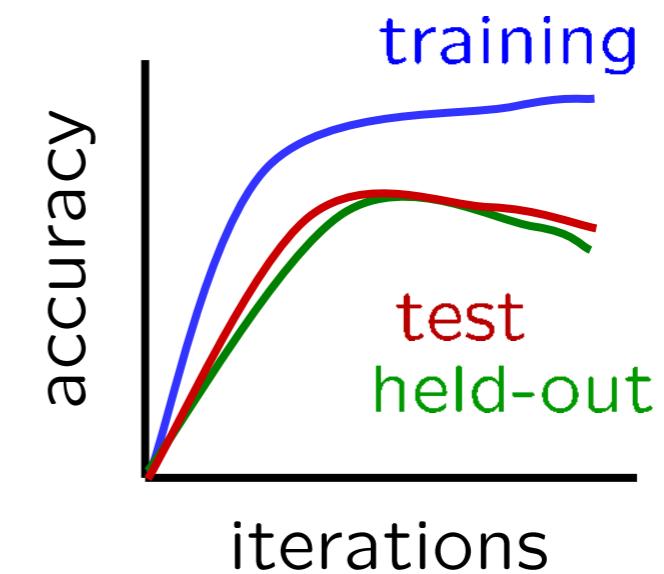
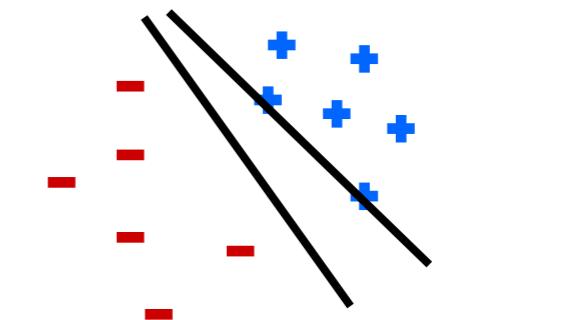
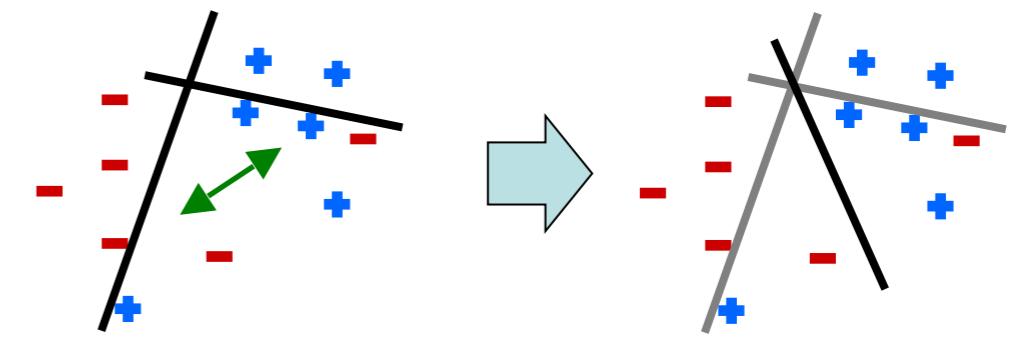


**Rectified Linear Units
(ReLU)**

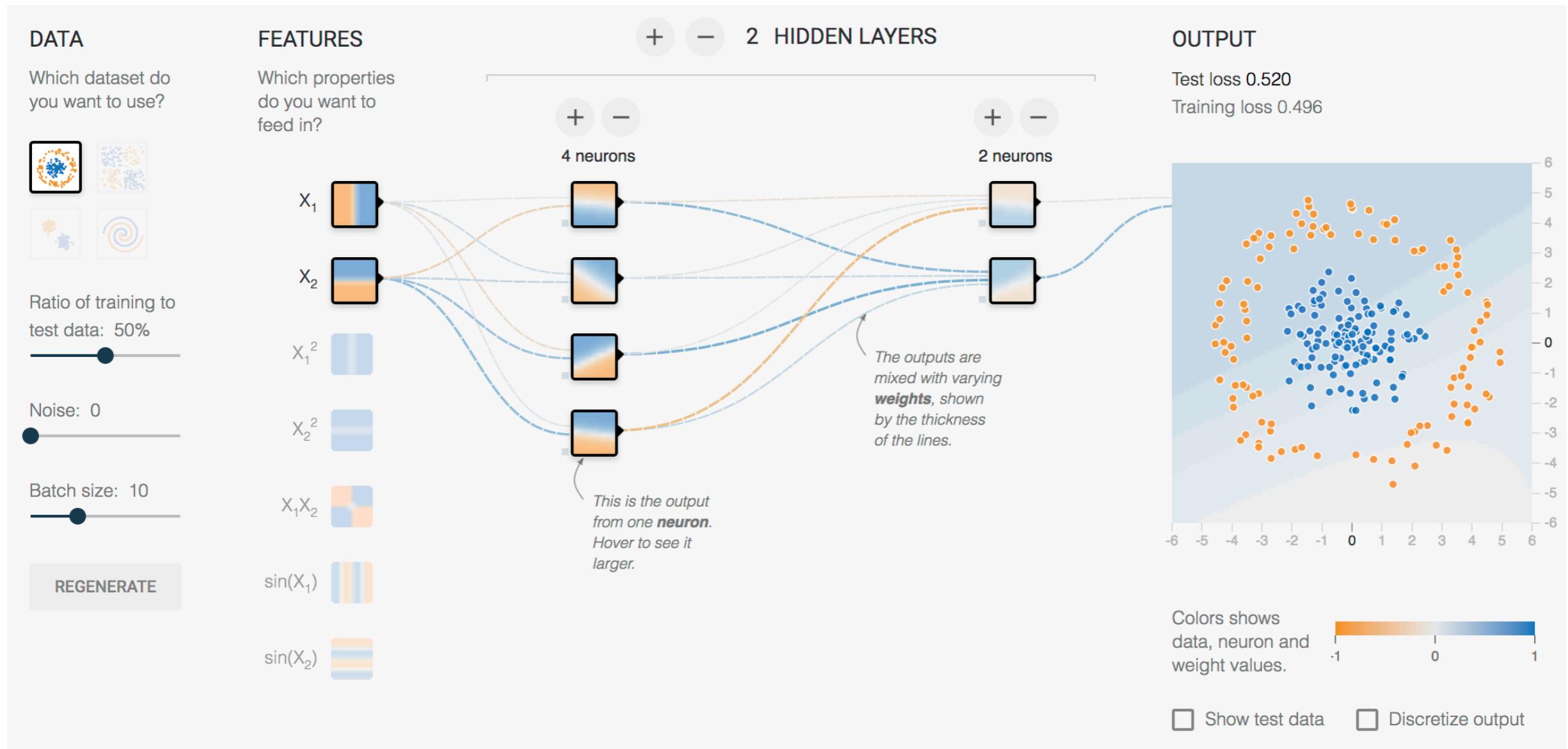
Derivative: 1, if $x > 0$; otherwise 0

Perceptron problems

- Noise: if the data isn't separable, weights might thrash
- Multiple layers allow it to work on non-linearly separable data
- Mediocre generalization: finds a “barely” separating solution
- More data and a continuous loss function greatly improve generalization
- Overtraining: test / held-out accuracy usually rises, then falls
- Large data, many layers, and batch training help



Elements of a Neural Network



<http://playground.tensorflow.org/>

Choosing the learning rate

- This is a subtle art.
- Too small: can take days instead of minutes to converge
- Too large: diverges (MSE gets larger and larger while the weights increase and usually oscillate)
- Sometimes the “just right” value is hard to find.

Improving Simple Gradient Descent

Momentum

Don't just change weights according to the current datapoint.
Re-use changes from earlier iterations.

Let $\Delta\mathbf{w}(t)$ = weight changes at time t .

Let $-\eta \frac{\partial E}{\partial w}$ be the change we would make with regular gradient descent.

Instead we use

$$\Delta\mathbf{w}(t+1) = -\eta \frac{\partial E}{\partial \mathbf{w}} + \alpha \Delta\mathbf{w}(t)$$
$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t)$$

Momentum damps oscillations.

A hack? Well, maybe.

momentum parameter

Batch learning

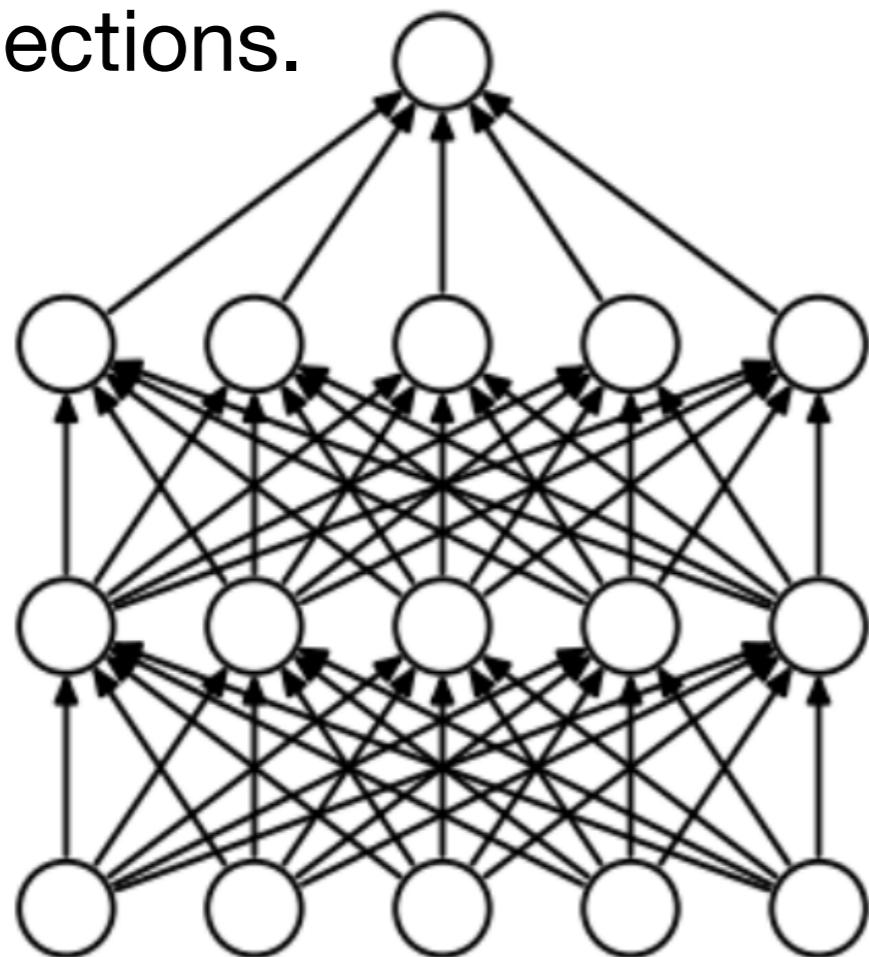
- All the training data is taken into consideration to take a single step
 - Take the average gradient of all the training examples, and use that to update the weights
- Pros: smooth updates, not hopping around
- Cons: Inefficient
- Minibatch gradient descent: use a random subset of the dataset for each batch

Modern optimizers

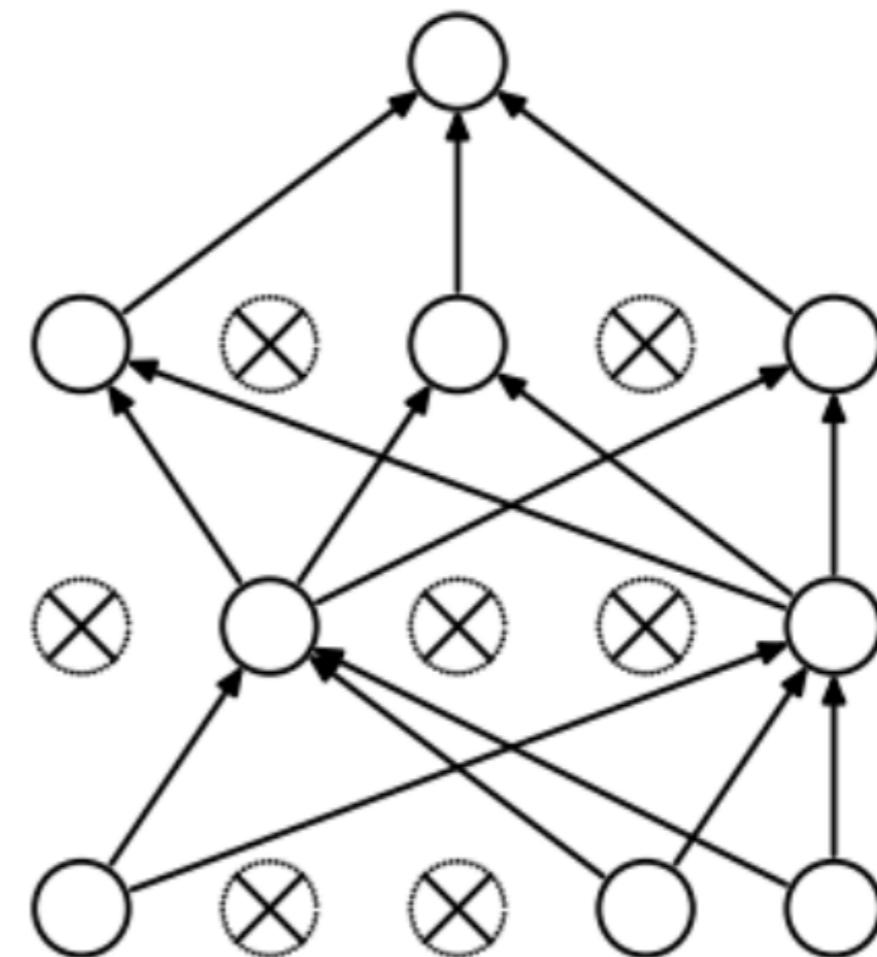
- Gradient Descent:
 - Batch gradient descent
 - Stochastic gradient descent
 - Mini-batch gradient descent
- Adaptive:
 - Adagrad
 - Adadelta
 - RMSprop
 - Adam

Dropout

In training, block out a random subsample of connections (temporarily set weights to 0). In inference, use all the connections.



(a) Standard Neural Net



(b) After applying dropout.