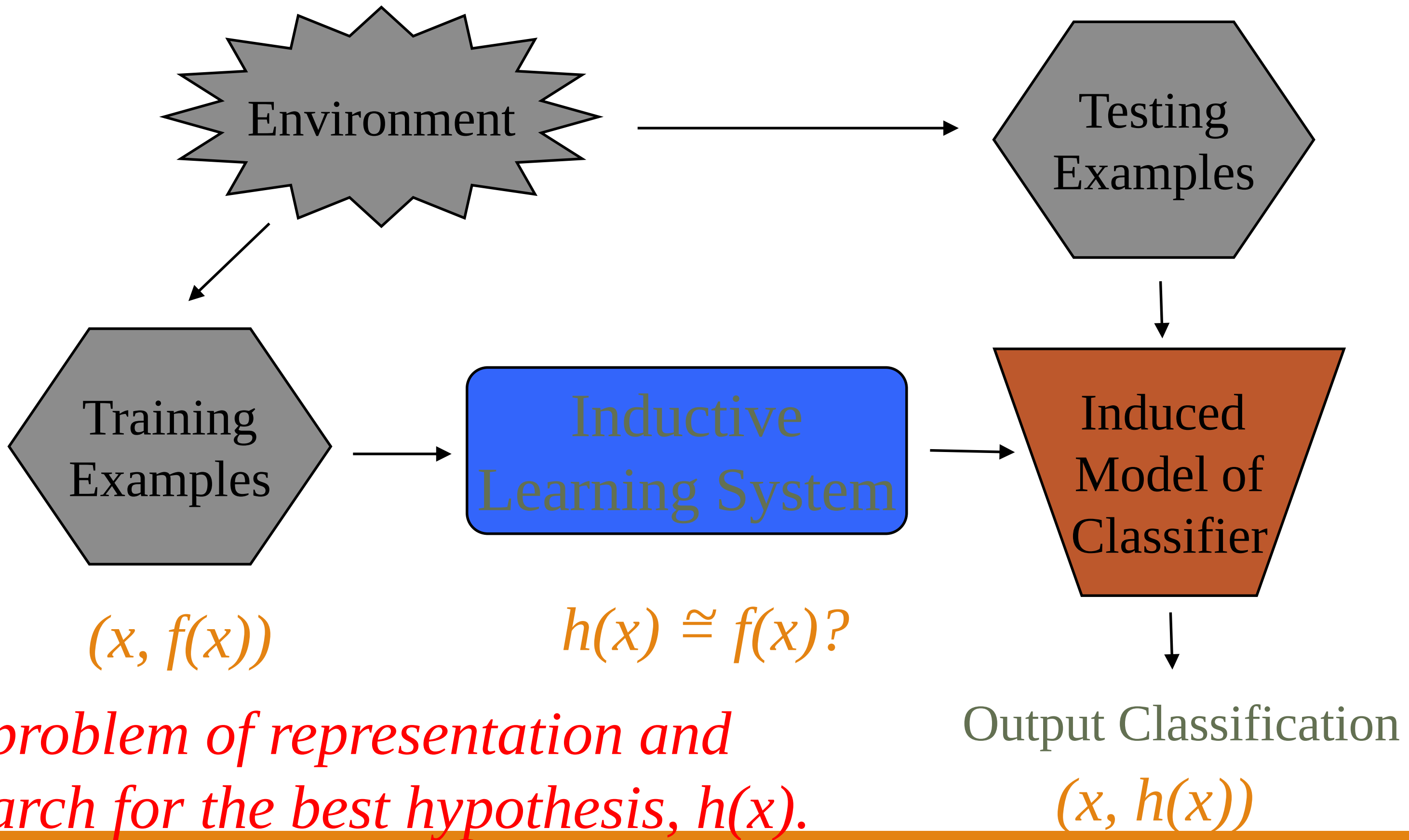
Artificial Neural Networks

Dr . M. SRIDEVI

Classification Systems

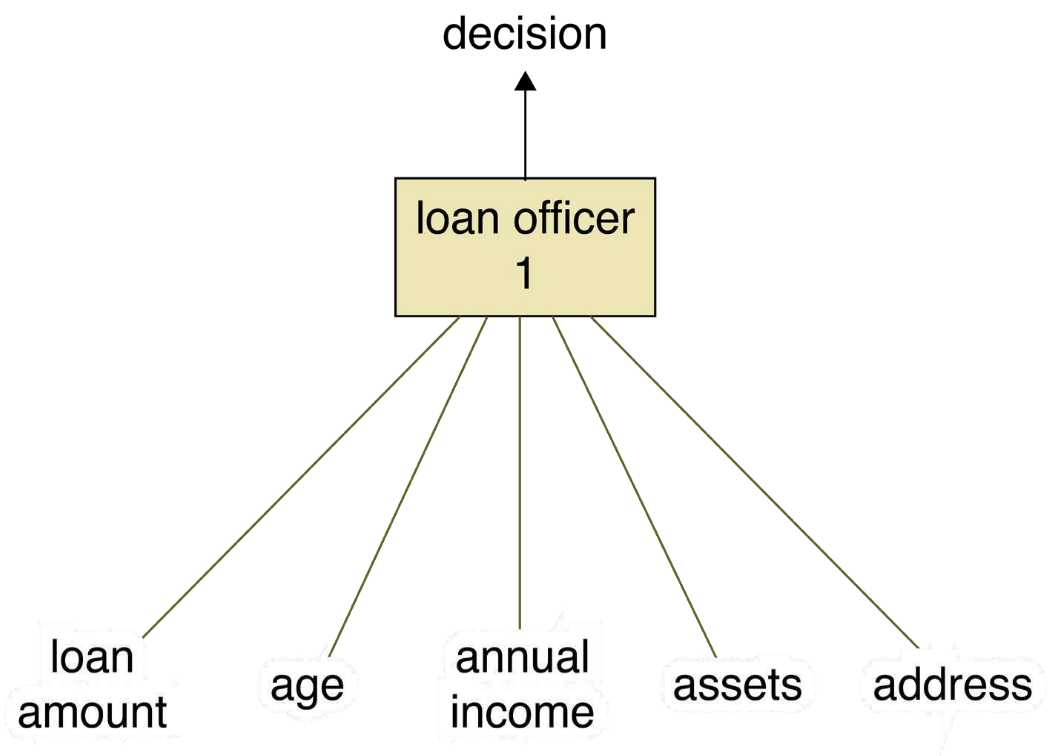
Basic Framework for Learning



A problem of representation and search for the best hypothesis, h(x).

Bank Loan Example with Single layer Neural Network

- Let's consider the process of getting a loan.
- In practice, any loan application is going to be a complex affair.
- Based on the applications for those loans, they came up with rules that would let them predict whether a loan was likely to end up getting paid back or not.
- This is of course just like what a perceptron does. The inputs are weighted and combined to produce a final score, as in Figure.
- Say, loan is rejected

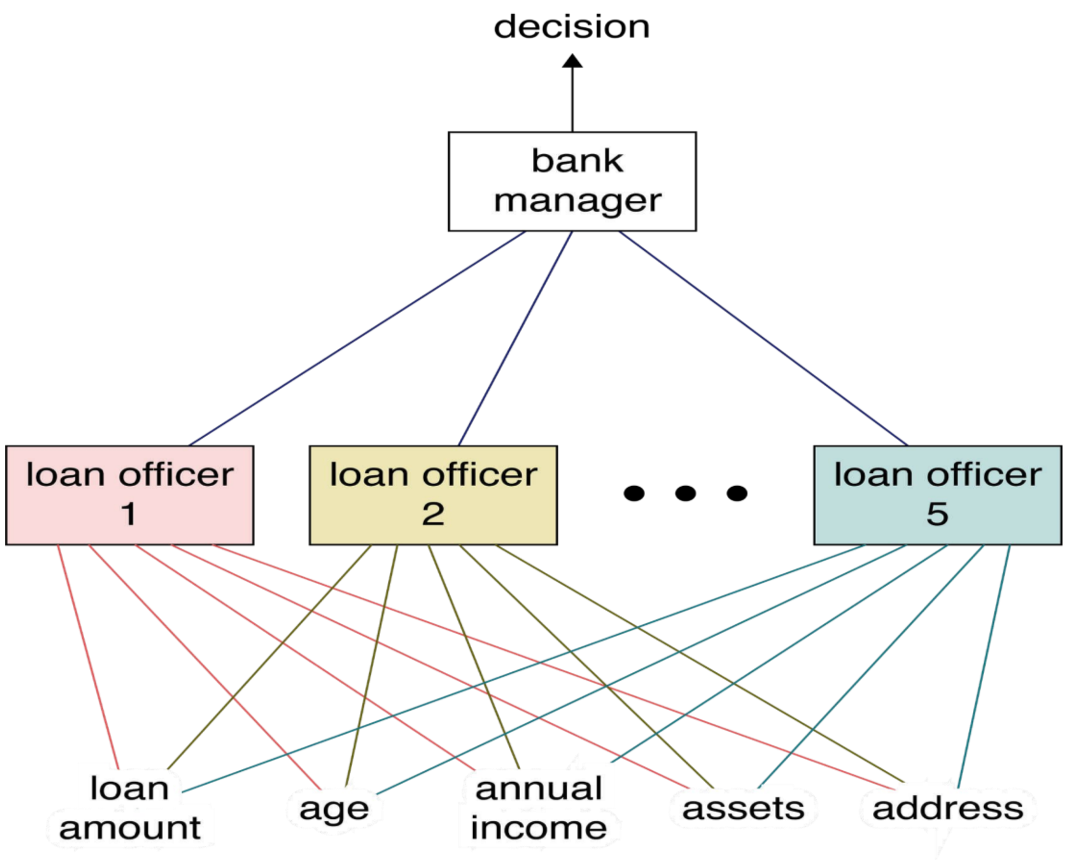


Bank Loan Example with 2 layer Neural Network

We might try our luck at another bank.

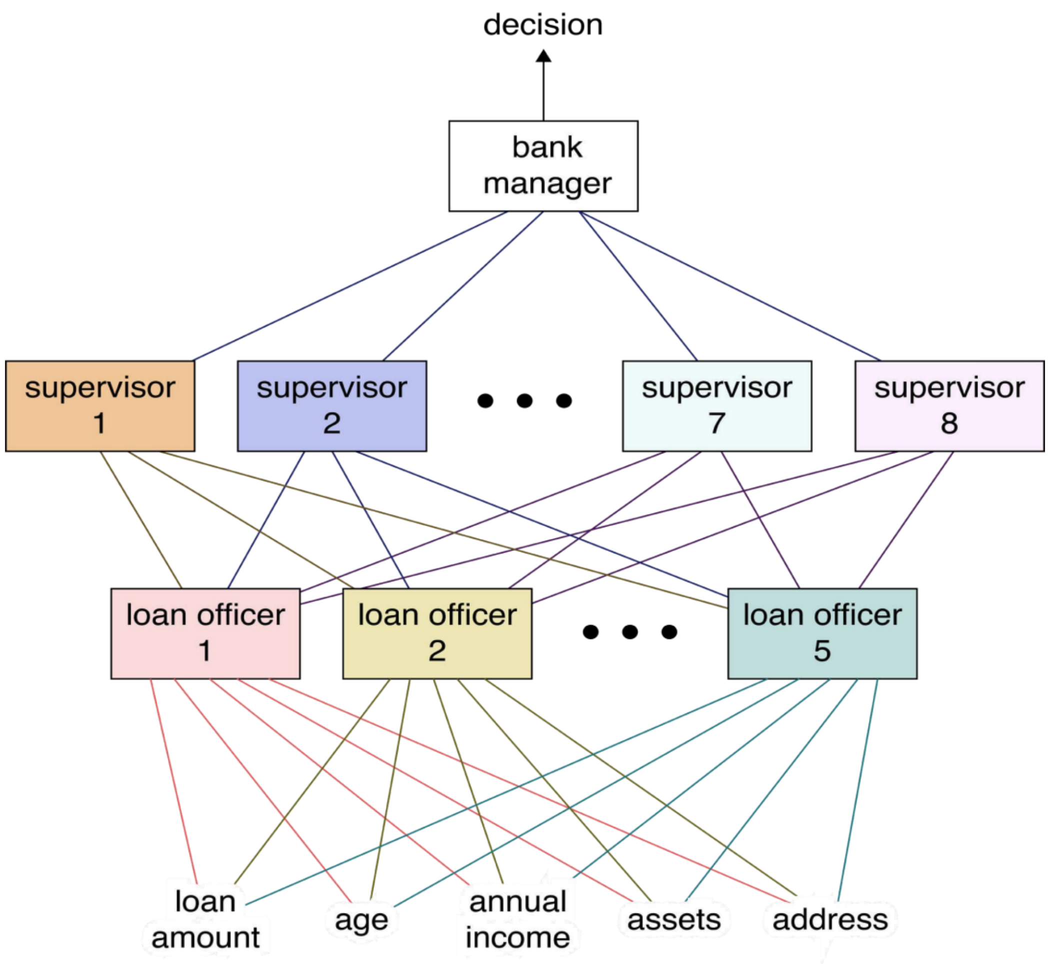
In this bank, suppose that there are 5 different loan officers, and each one has developed his own idiosyncratic procedure for evaluating the criteria that go into a loan.

We can draw this as a two-layer neural network.



Bank Loan Example with 3 layer Neural Network

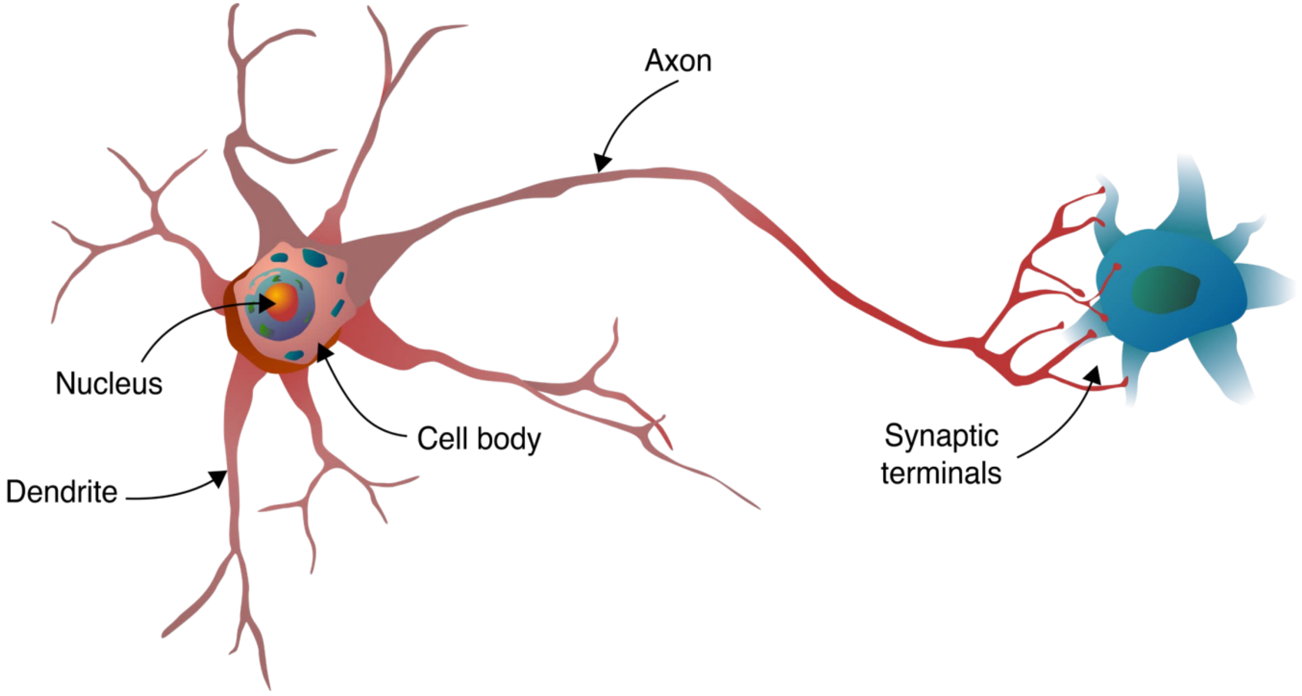
- Suppose bank hires a bunch of ssupervisors to sit between the loan officers and bank manager.
- Suppose we have 5 loan officers, 8 supervisors, and 1 bank manager.
- Then we have a 3-layer neural network to represent this process, as in Figure.
- Supervisors look at the decisions of the loan officers.
- Supervisors combine the results of the loan officers, and then pass their judgements up to the bank manager.
- Bank manager final decision is based on how he chooses to weight the conclusions of the supervisors.



Neurons

How real biological neurons inspired the artificial neurons we use in machine learning, and how those little bits of processing work alone and in groups.

- Neurons are the nerve cells that make up the brain, used for our cognitive abilities.
- Neurons are information processing machines.



A sketch of a biological neuron (in red) with a few major structures identified. This neuron's outputs are communicated to another neuron (in blue), only partially shown.

Neuron Model

Neuron collects signals from *dendrites*

Sends out spikes of electrical activity through an *axon*, which splits into thousands of branches.

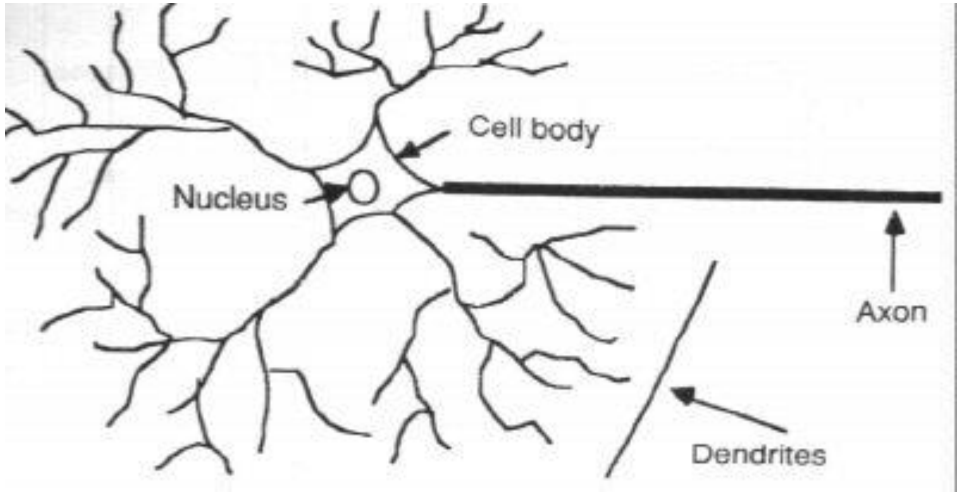
At end of each brand, a *synapses* converts activity into either exciting or inhibiting activity of a dendrite at another neuron.

Neuron *fires* when exciting activity surpasses inhibitory activity

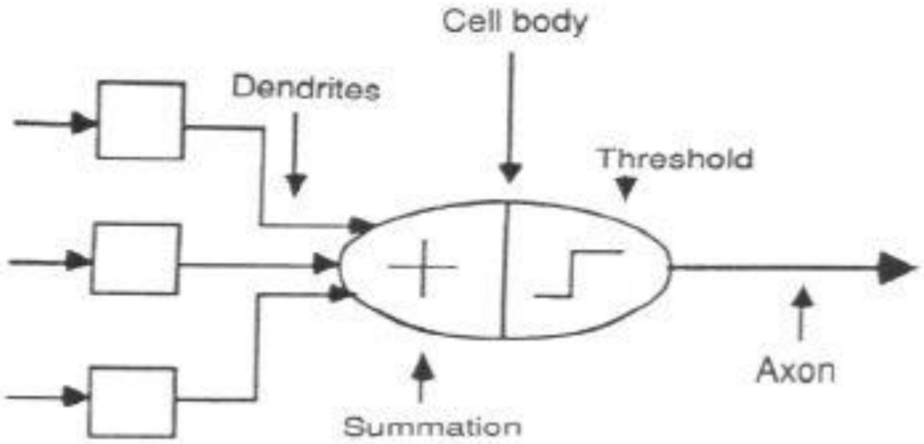
Learning changes the effectiveness of the synapses

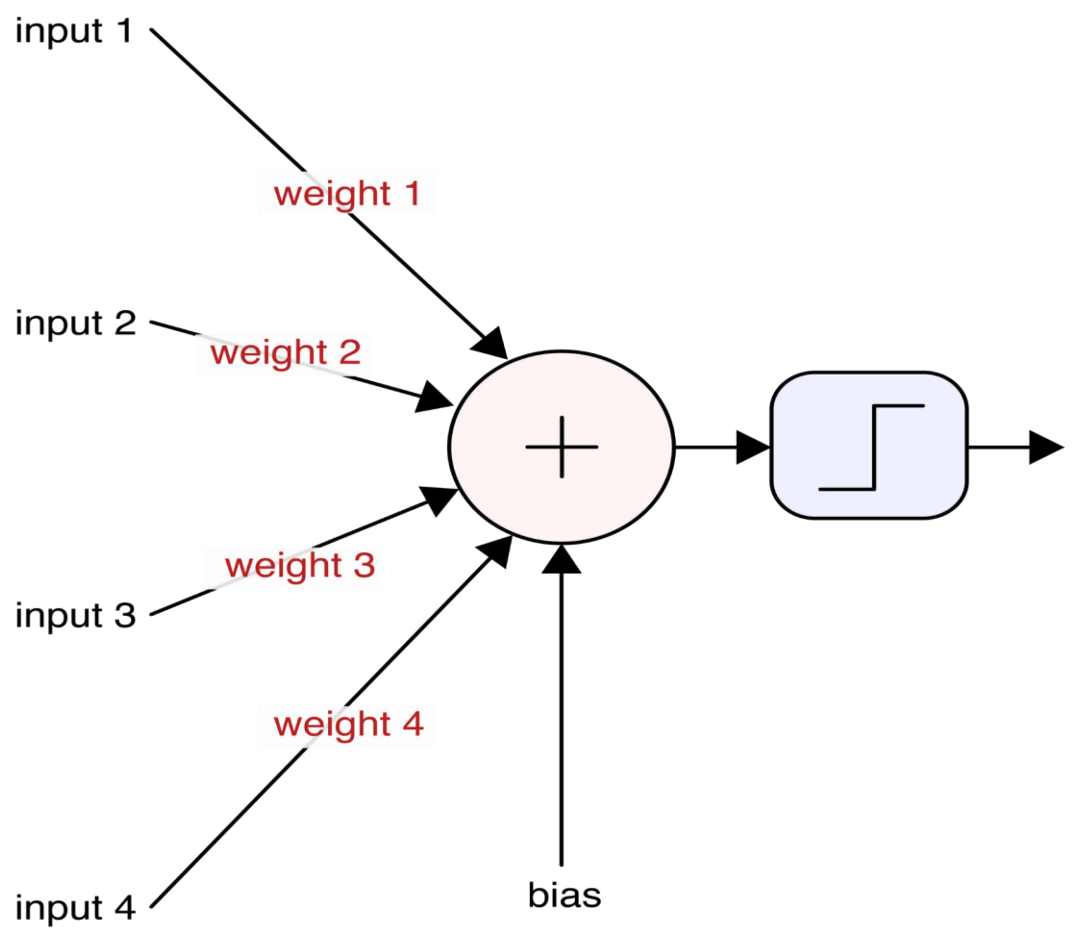
Neuron Model

Natural neurons



Abstract neuron model



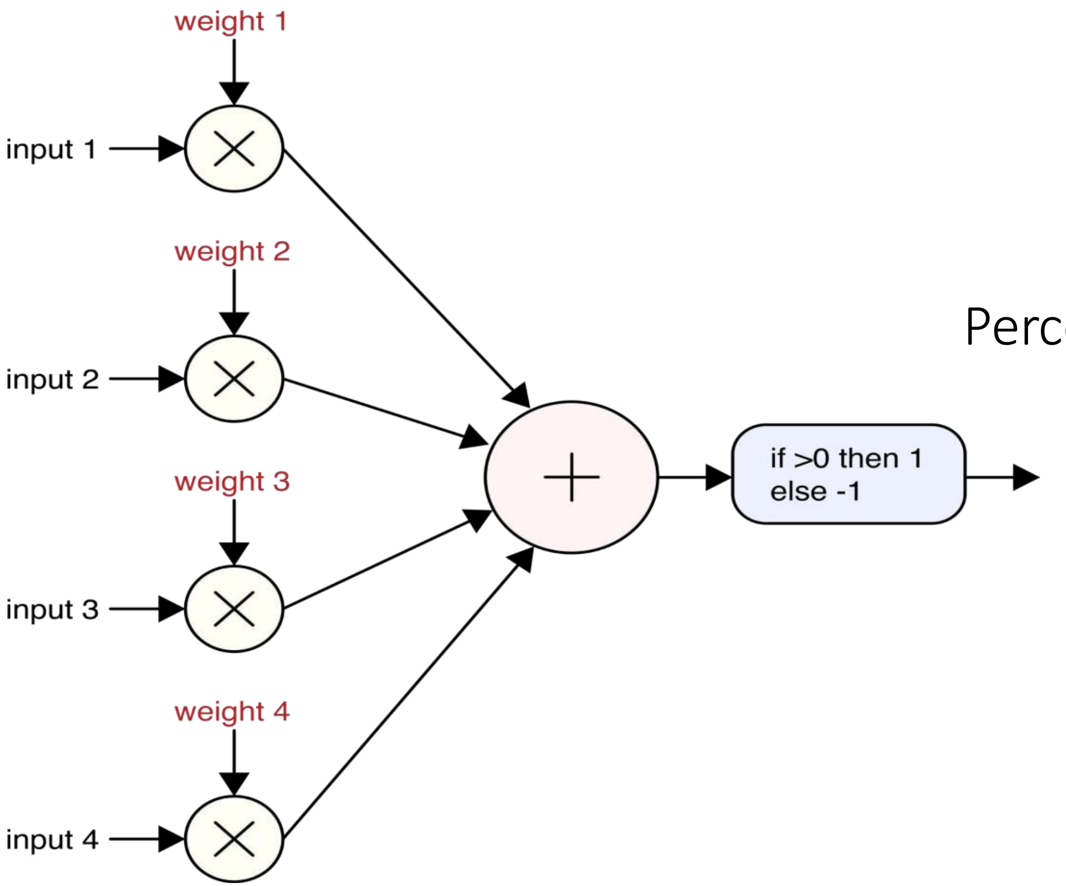


Note:

- In neural network diagrams, weights and the nodes where they multiply are not drawn
- Weights are always there and they always modifies the input . They're just not drawn.
- Output of the activation function might take on any value.
- Variety of activation functions each with its own pros and cons.
- We'll run through them later.

A neuron is often drawn with the weights on the arrows. This "implicit multiplication" is common in machine learning figures. We've also replaced the threshold function with a step, to remind us that any activation function can follow the sum

The "bias trick" in action. Rather than show the bias term explicitly. We pretend it's another input with its own weight.



Perceptron mimics the functional behaviour of neuron

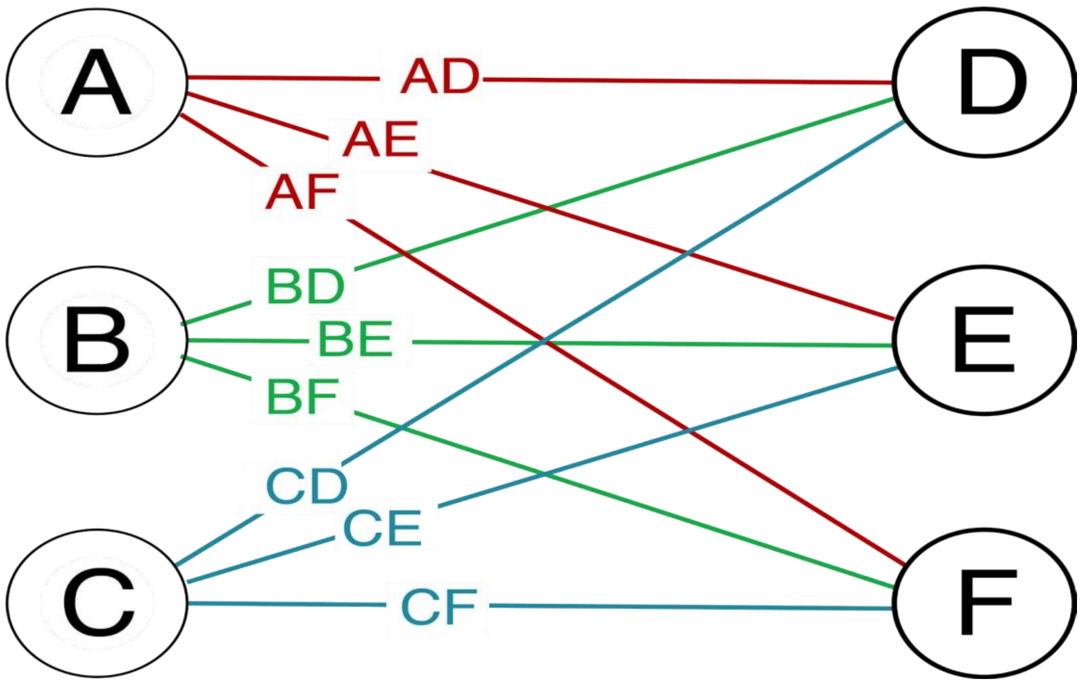
A schematic view of a perceptron. Each input is a single number, and it's multiplied by a corresponding real number called its weight.

The results are all added together, and then tested against a threshold.

If results are positive, perceptron outputs +1, else −1.

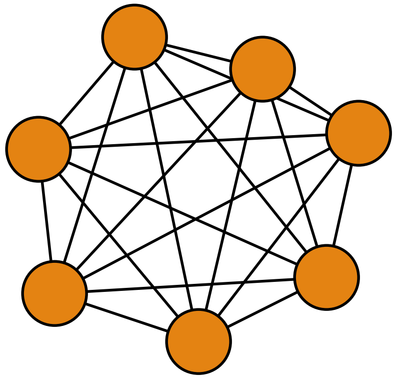
Modern Artificial Neurons

- **Modern** Neurons are only slightly generalized from the **original** perceptrons.
- Still called “perceptrons” or “**neurons”**
- Two changes to original perceptron: one at the input, and one at the output.
- 1. Input - Provide each neuron with one more input, called **bias**. It’s a number that is directly added into the sum of all the weighted inputs. Each neuron has its own bias.
- 2. Output- Replace threshold with an activation function, i.e., a mathematical **function** that takes the sum (including the bias) as input and returns a new floating-point value as output.

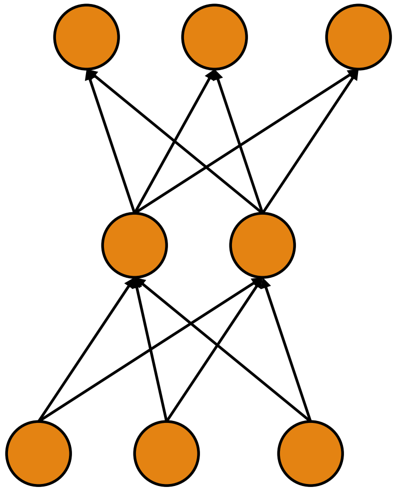


The weights are shown explicitly in this diagram. Following convention, each weight is given a two-letter name formed by combining the names of the neuron that produced the output on that weight's wire, with the name of the neuron that receives the value as input. For example, BF is the weight that multiplies the output of B for use by F

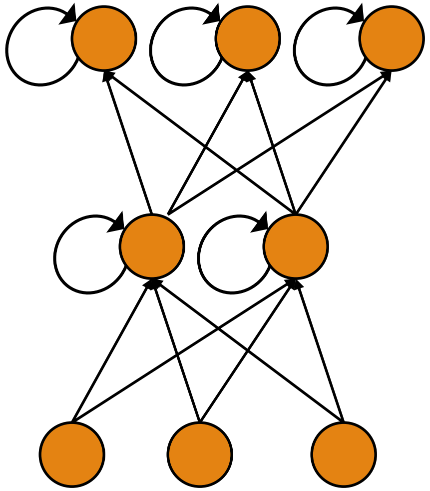
Topologies of Neural Networks



completely connected

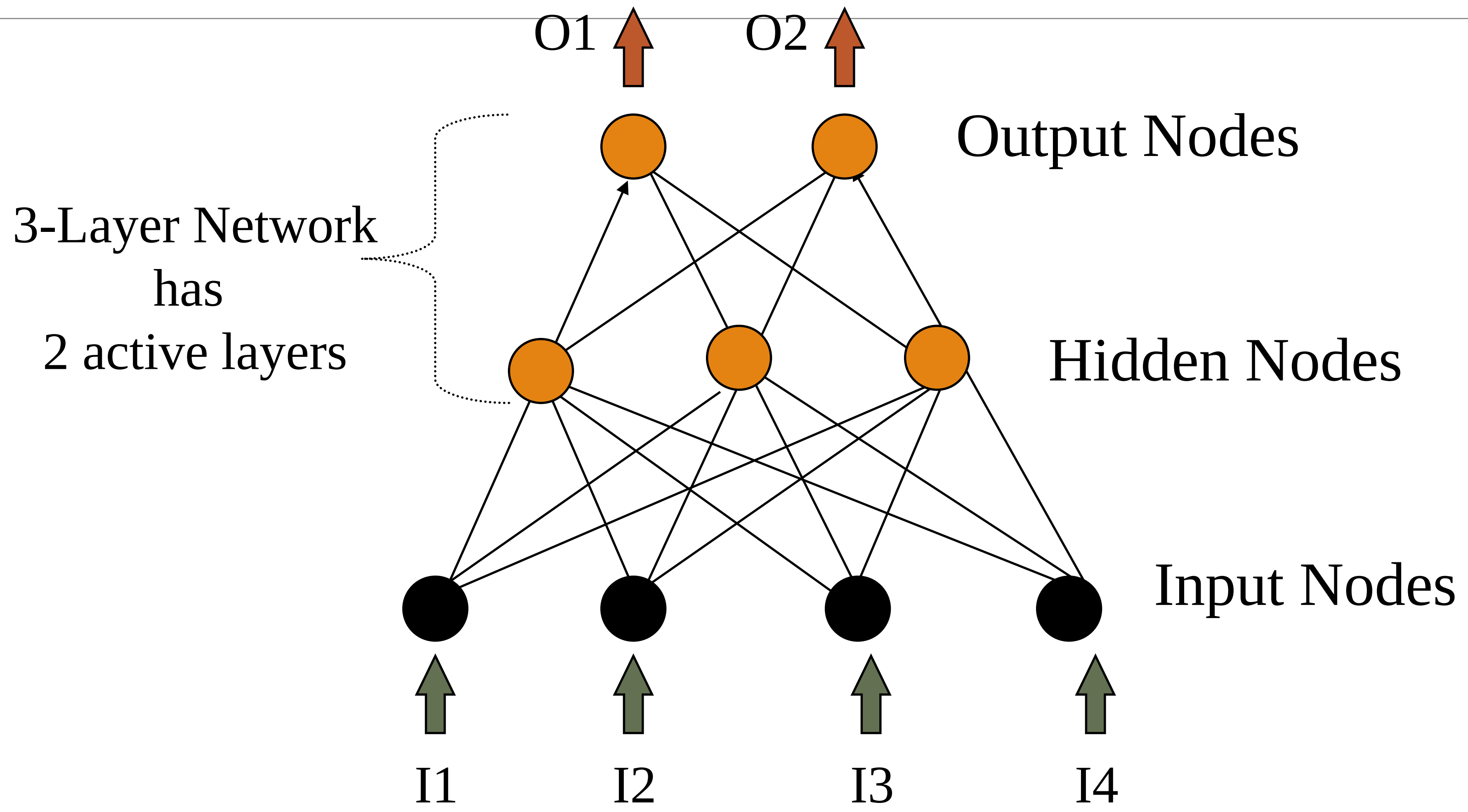


feedforward (directed, a-cyclic)

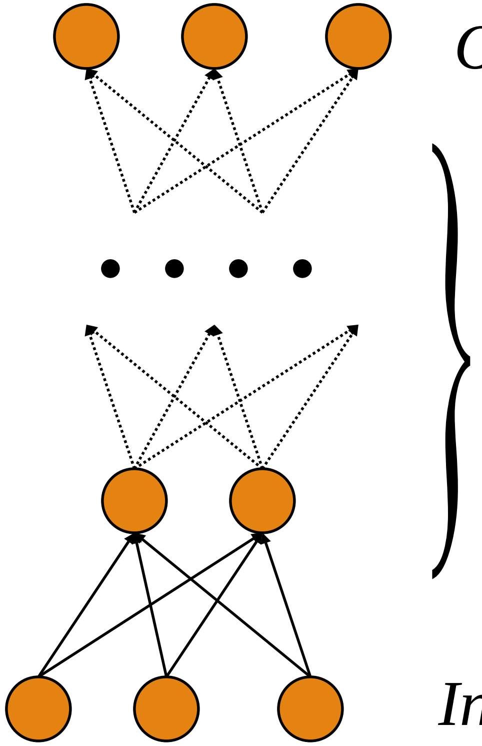


recurrent (feedback connections)

Artificial Neurons



Multilayer Perceptron



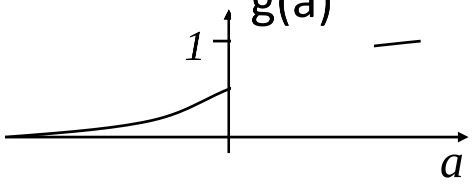
Output neurons

One or more layers of hidden units (hidden layers)

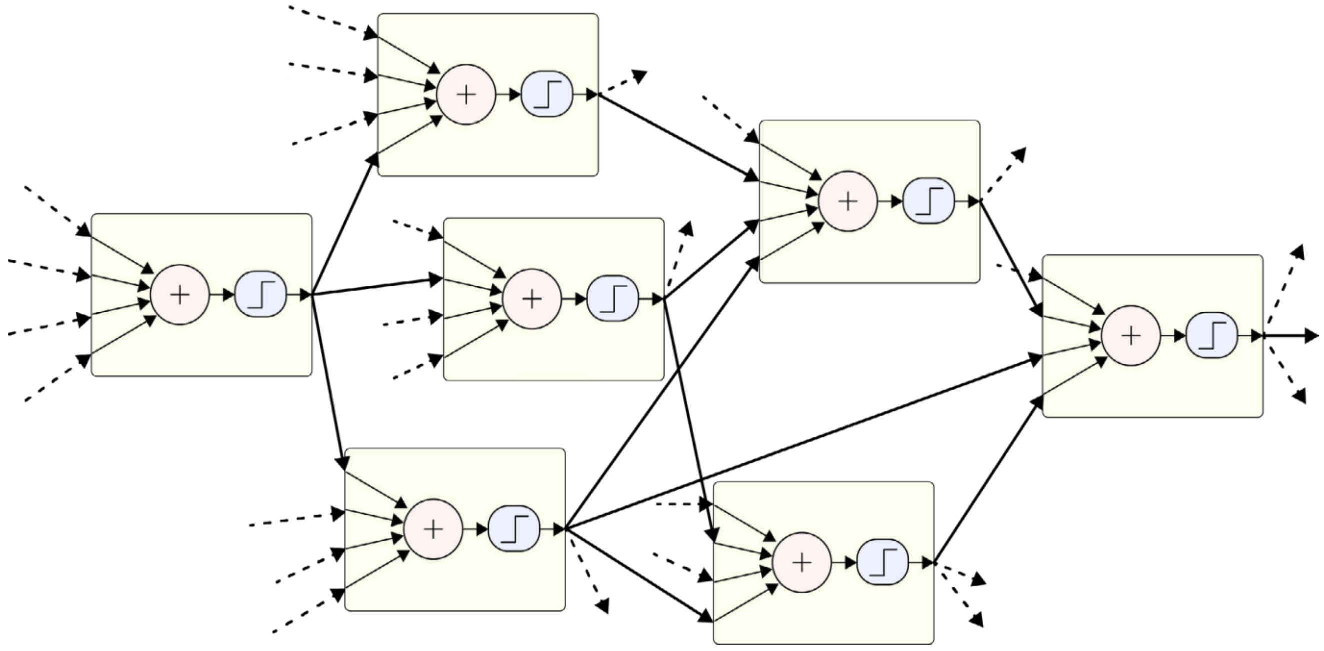
Input nodes

The most common output function (Sigmoid):

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



(non-linear squashing function)



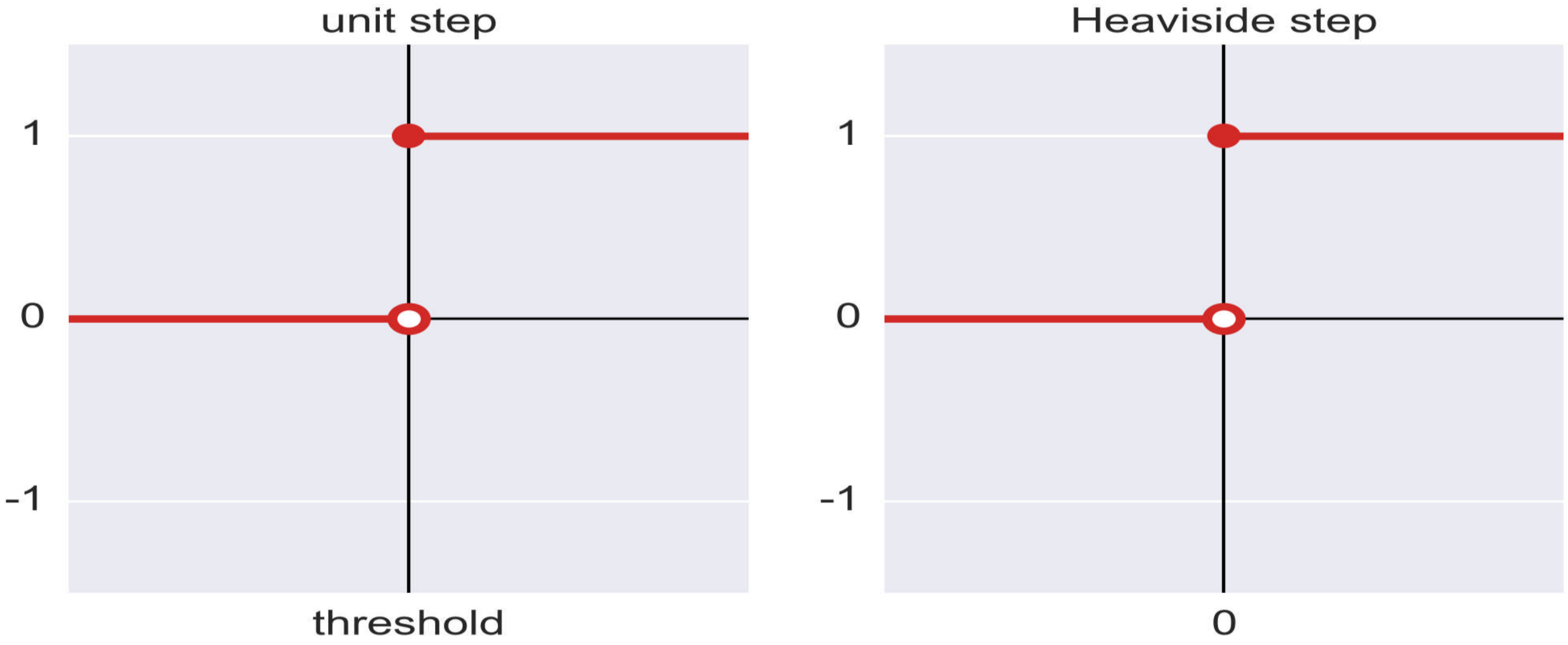
A piece of a larger network of artificial neurons. Each neuron receives its inputs from other neurons The dashed lines show connections coming from outside this little cluster.

Activation Functions

The last step in an artificial neuron is to apply an activation function to the value it computes.

Here we'll see a variety of popular activation functions in use today.

Activation function – Step

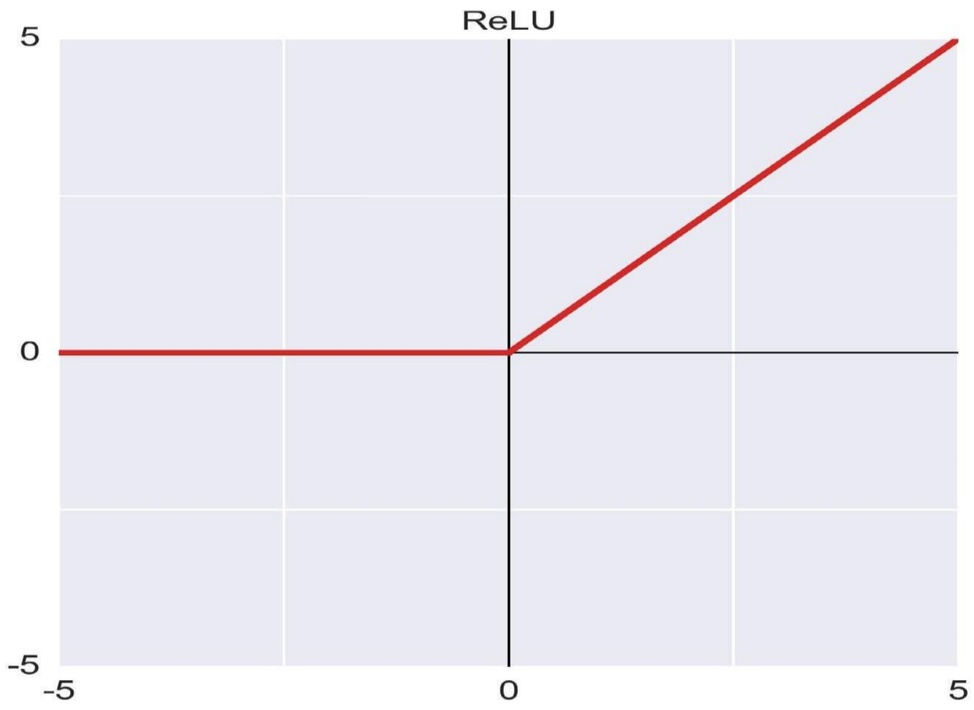


A couple of popular step functions.

Left: Unit step has a value of 0 to left of the threshold, and 1 to the right.

Right: The Heaviside step is a unit step where the threshold is 0.

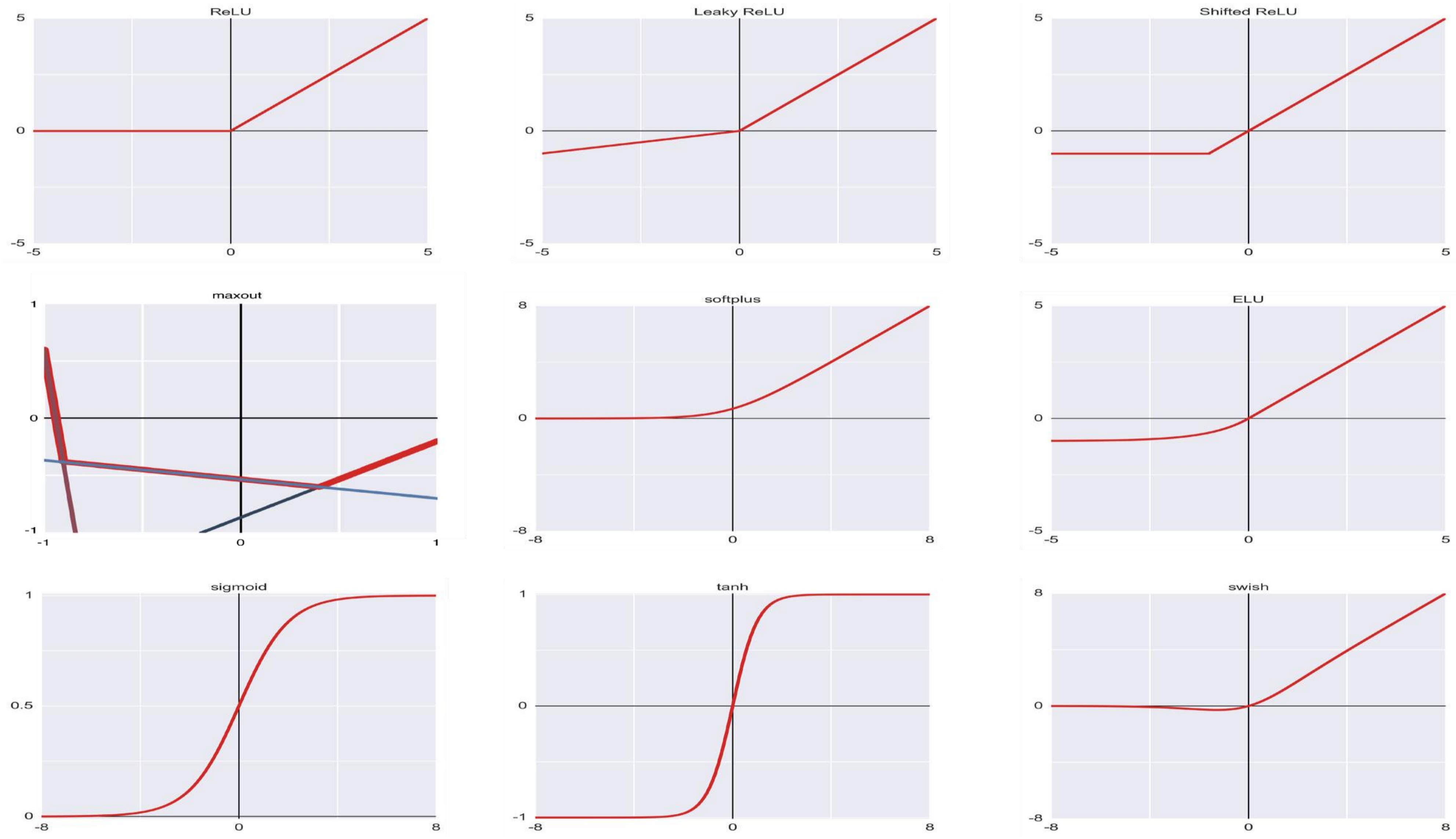
Activation function –ReLU



- ReLU is not a linear function.
- ReLU is popular because it's a simple and fast way to include a non-linearity in our artificial neurons.
- ReLU performs well and is often the first choice of activation function when building a new network.
- There are good mathematical reasons to use ReLU

- The ReLU, or rectified linear unit.
- Output is 0 for all negative inputs. Else, output = input

Functions



- Top row, left to right: ReLU, leaky ReLU, shifted ReLU.
- Middle row: maxout, softplus, ELU. Bottom row: sigmoid, tanh, swish.

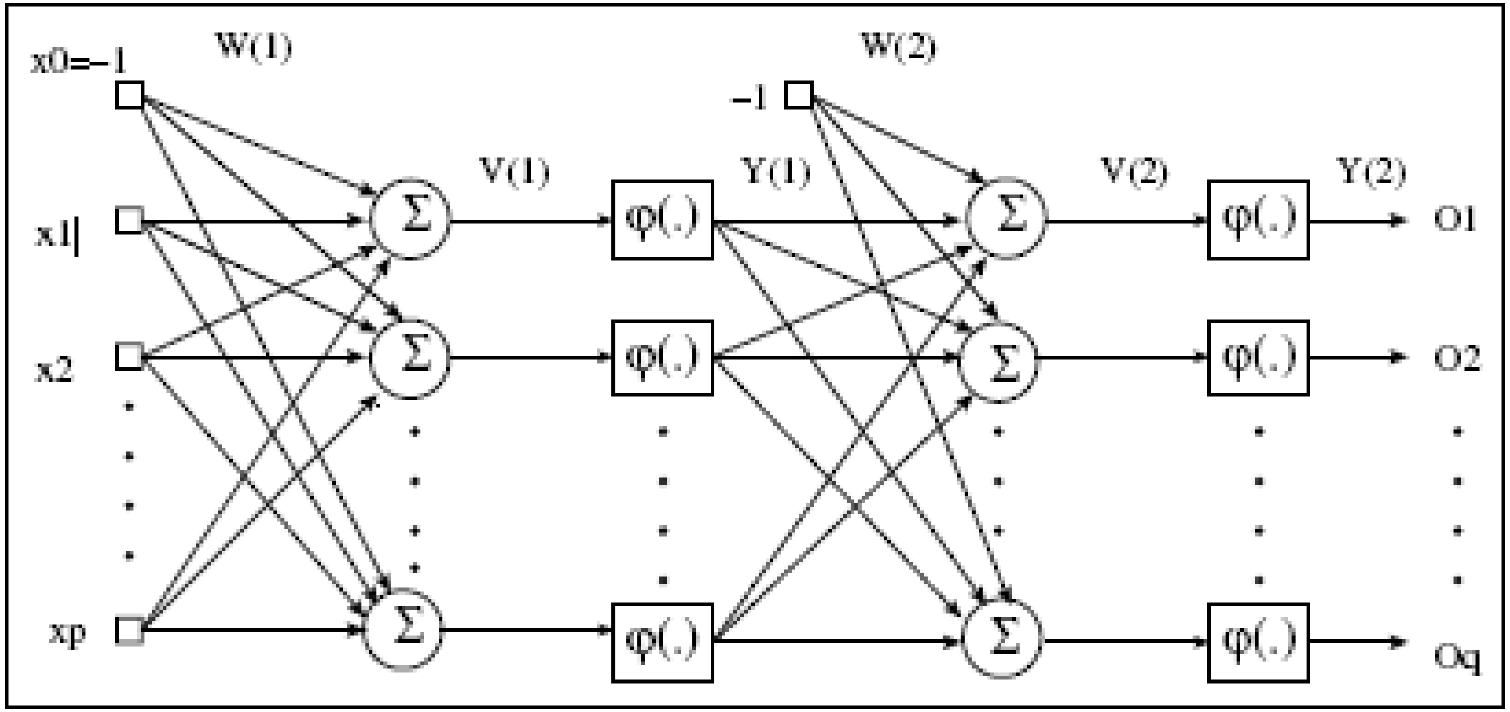
Learning in a ANN

Perceptron Learning Algorithm:

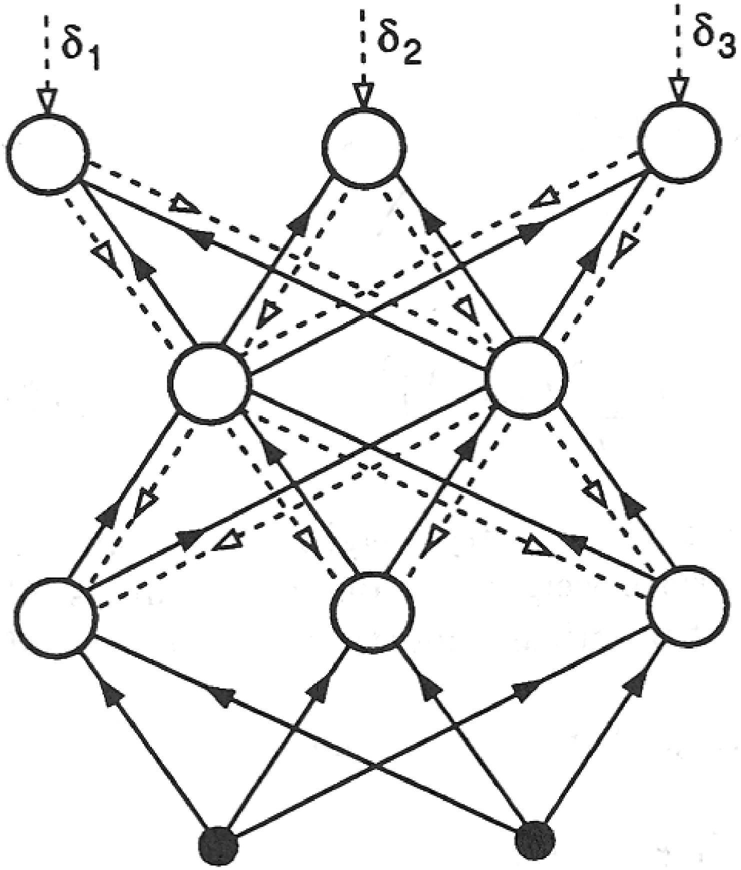
1. Initialize weights
2. Present a pattern and target output
3. Compute output :
4. Update weights :

Repeat starting at 2 until acceptable level of error

Forward Propagation



Back propagation



ANN Forward Propagation

Bias Nodes

- Add one node to each layer that has constant output

Forward propagation

- Calculate from input layer to output layer
- For each neuron:
  - Calculate weighted average of input
  - Calculate activation function

ANN Forward Propagation

Apply input vector **X** to layer of neurons.

Calculate

$$V_j(n) = \sum_{i=1}^{p} (W_{ji} X_i + Threshold)$$

- where X_i is the activation of previous layer neuron i
- W_{ji} is the weight of going from node i to node j
- p is the number of neurons in the previous layer

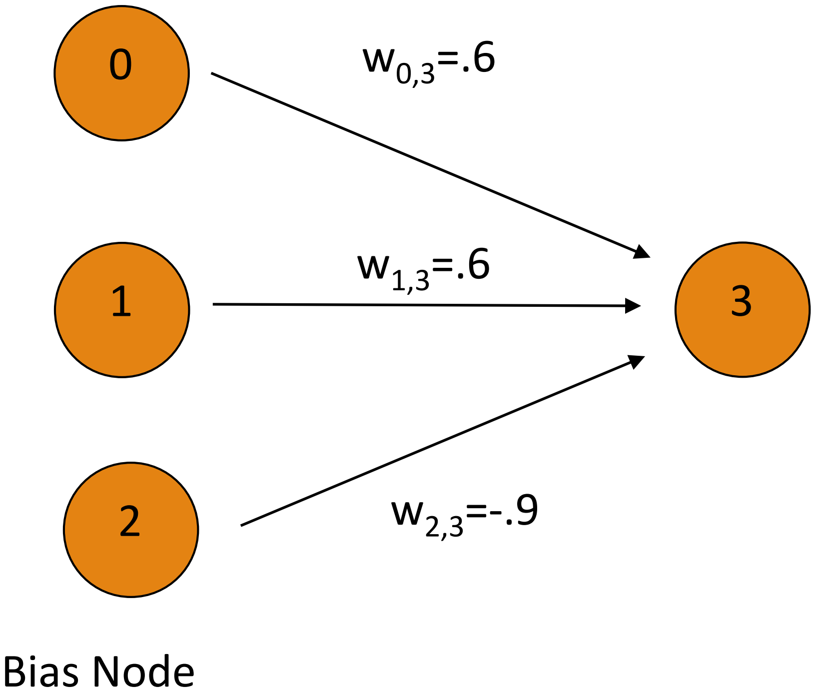
Calculate output activation

$$Y_j(n) = \frac{1}{1 + \exp(-V_j(n))}$$

ANN Forward Propagation

Example: ADALINE Neural Network

- Calculates and of inputs

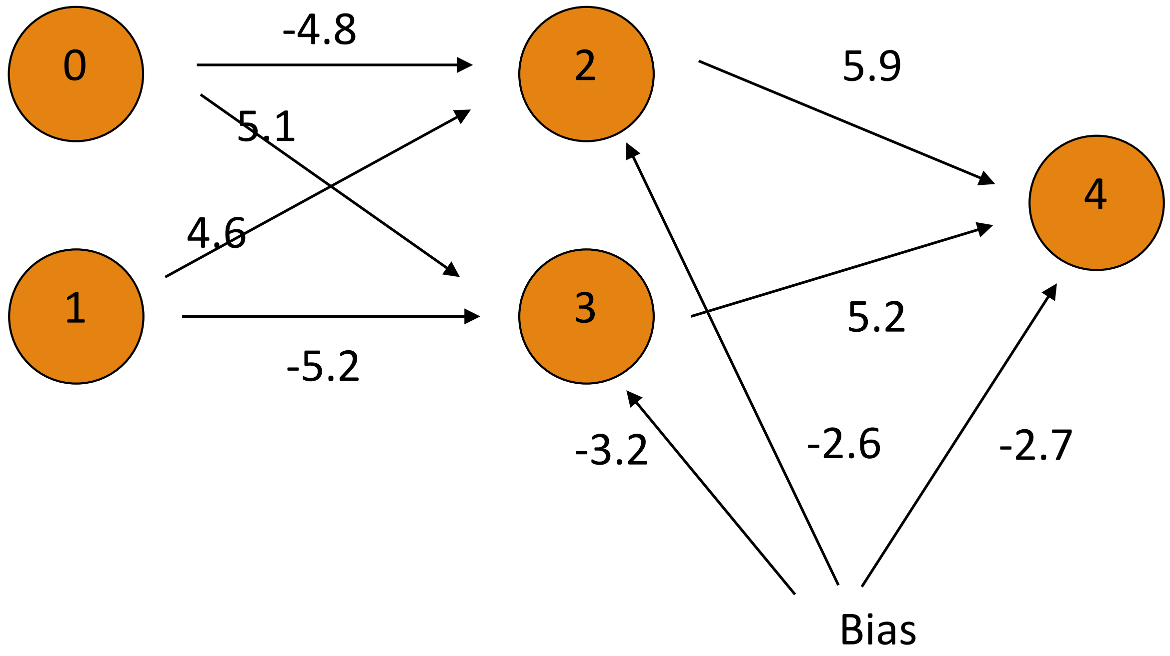


threshold function is step function

ANN Forward Propagation

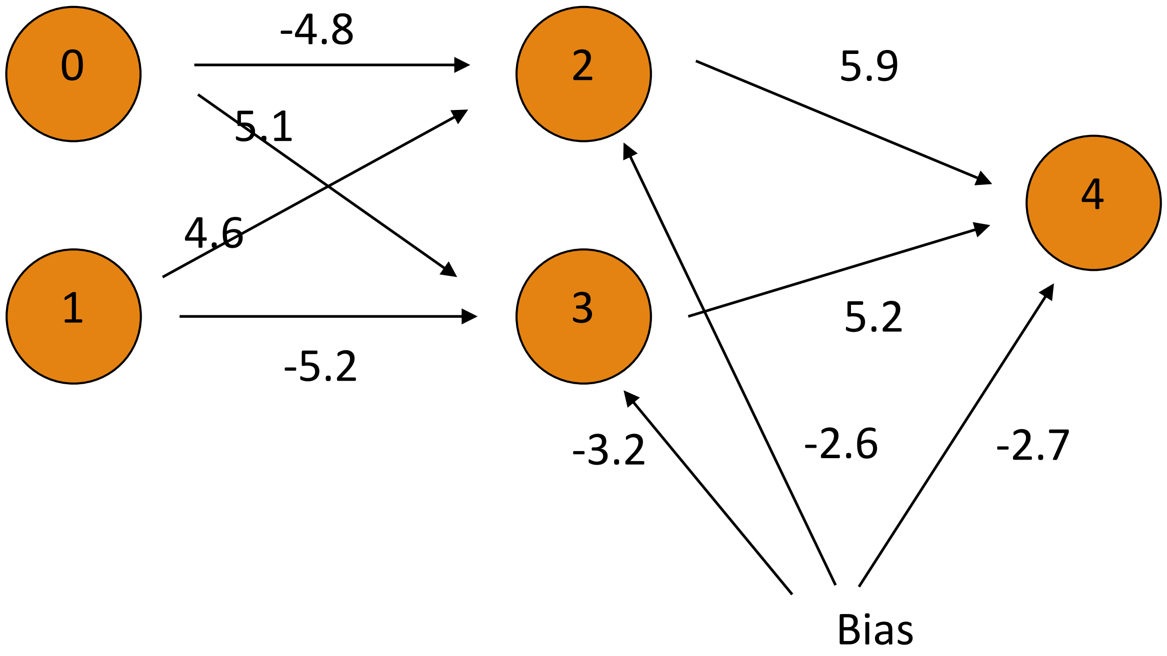
Example: Three layer network

- Calculates xor of inputs



ANN Forward Propagation

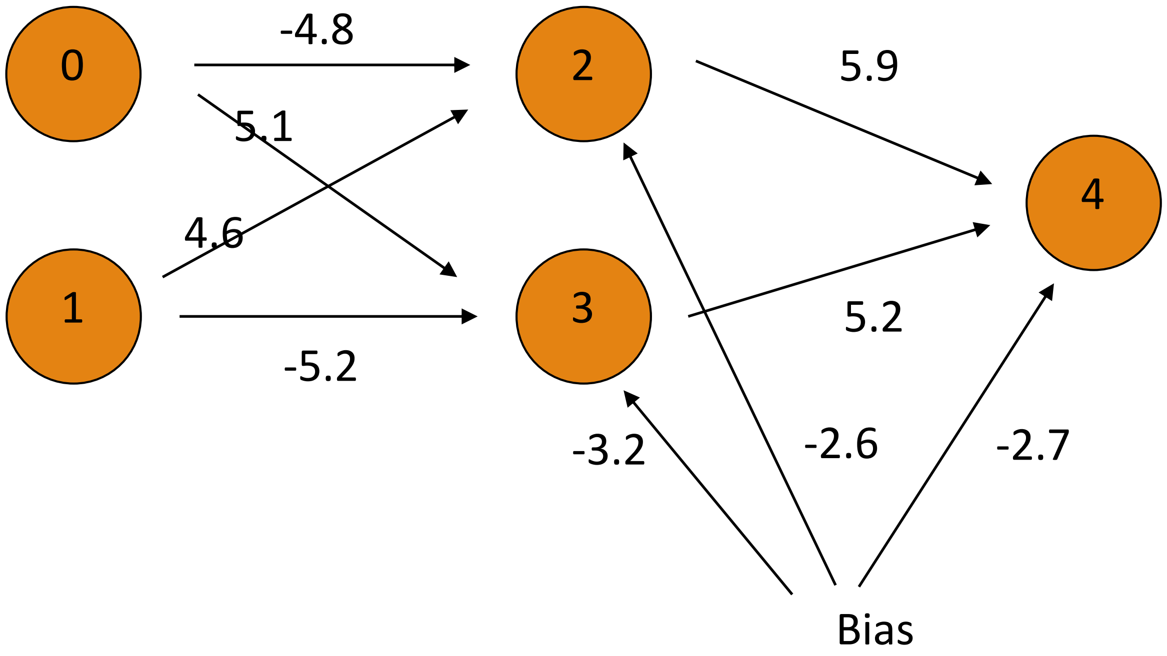
Input (0,0)



ANN Forward Propagation

Input (0,0)

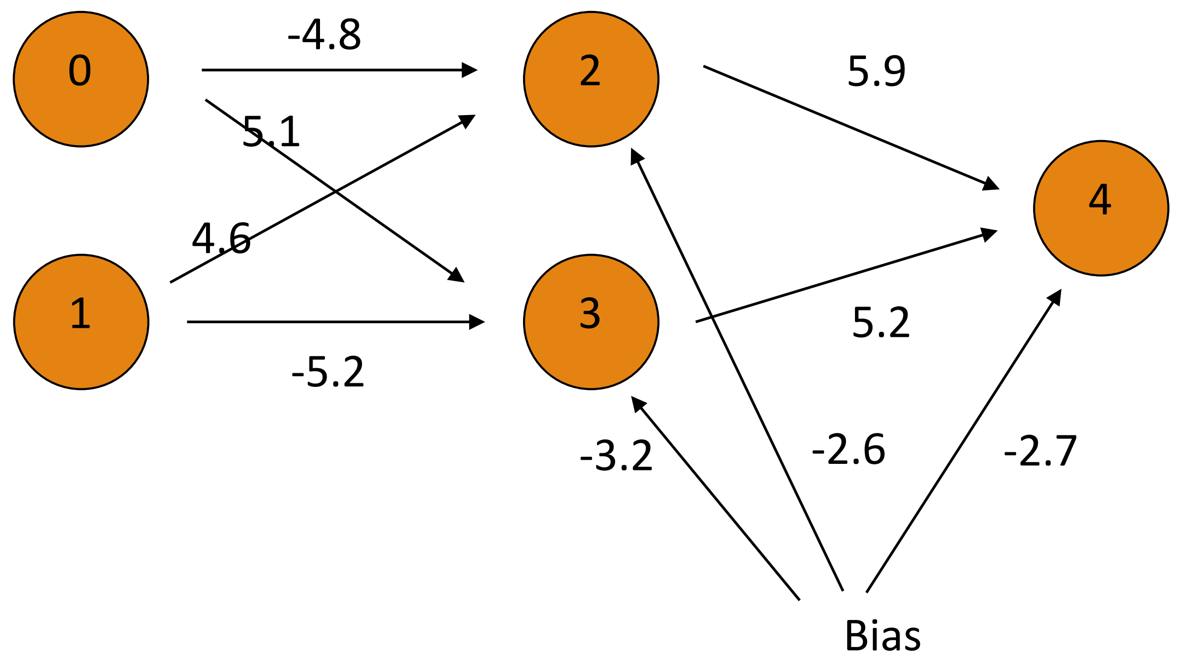
- Node 2 activation is $\varphi(-4.8 \cdot 0 + 4.6 \cdot 0 - 2.6) = 0.0691$



ANN Forward Propagation

Input (0,0)

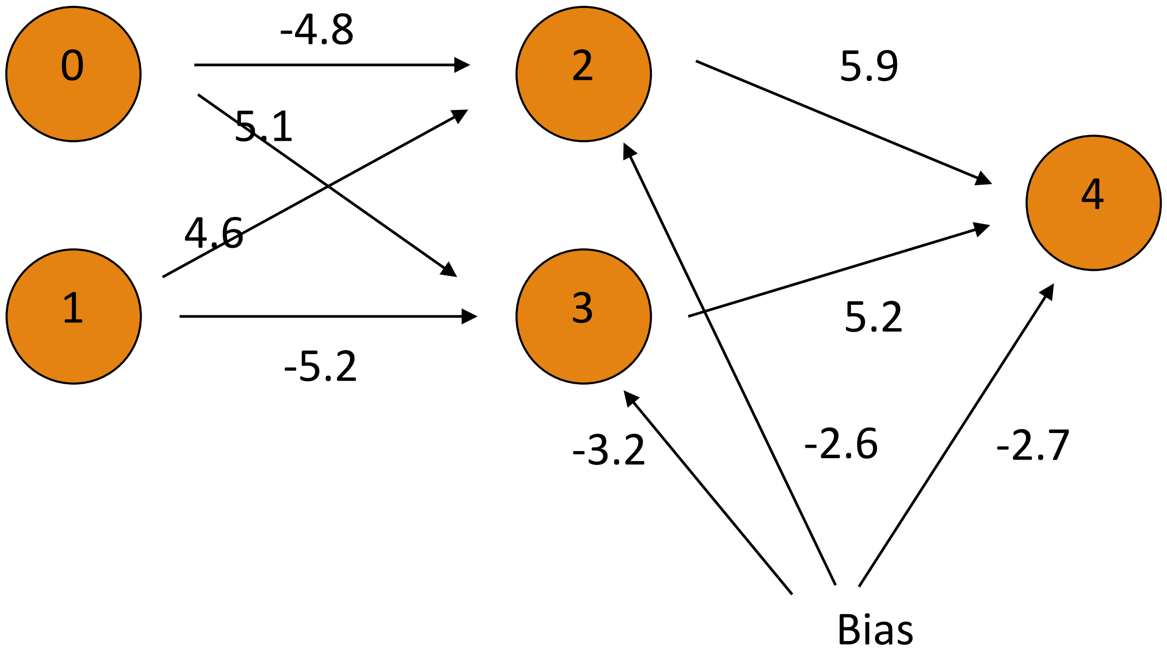
- Node 3 activation is $\varphi(5.1 \cdot 0 - 5.2 \cdot 0 - 3.2) = 0.0392$



ANN Forward Propagation

Input (0,0)

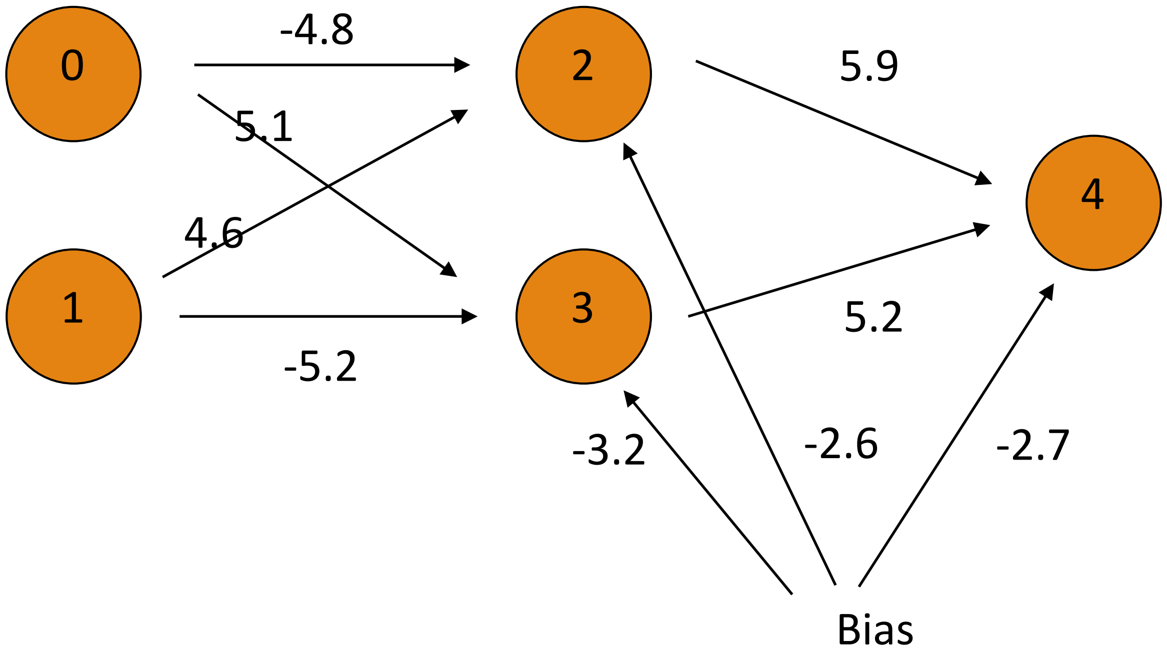
- Node 4 activation is $\varphi(5.9 \cdot 0.069 + 5.2 \cdot 0.069 - 2.7) = 0.110227$



ANN Forward Propagation

Input (0,1)

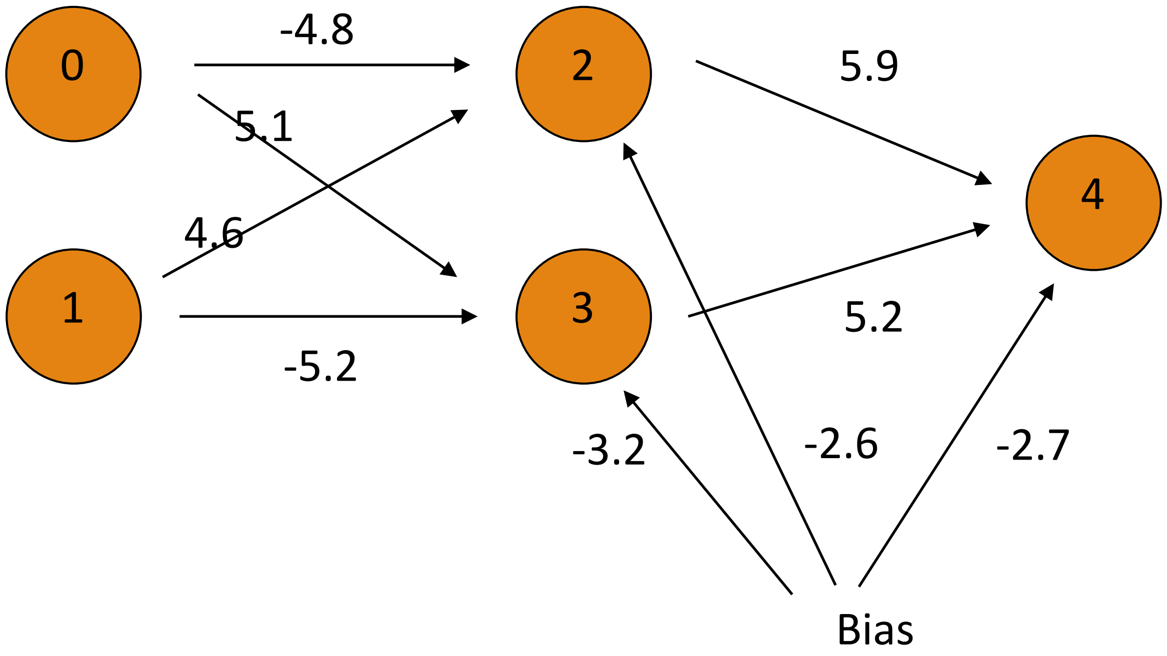
- Node 2 activation is φ(4.6 -2.6)= 0.153269



ANN Forward Propagation

Input (0,1)

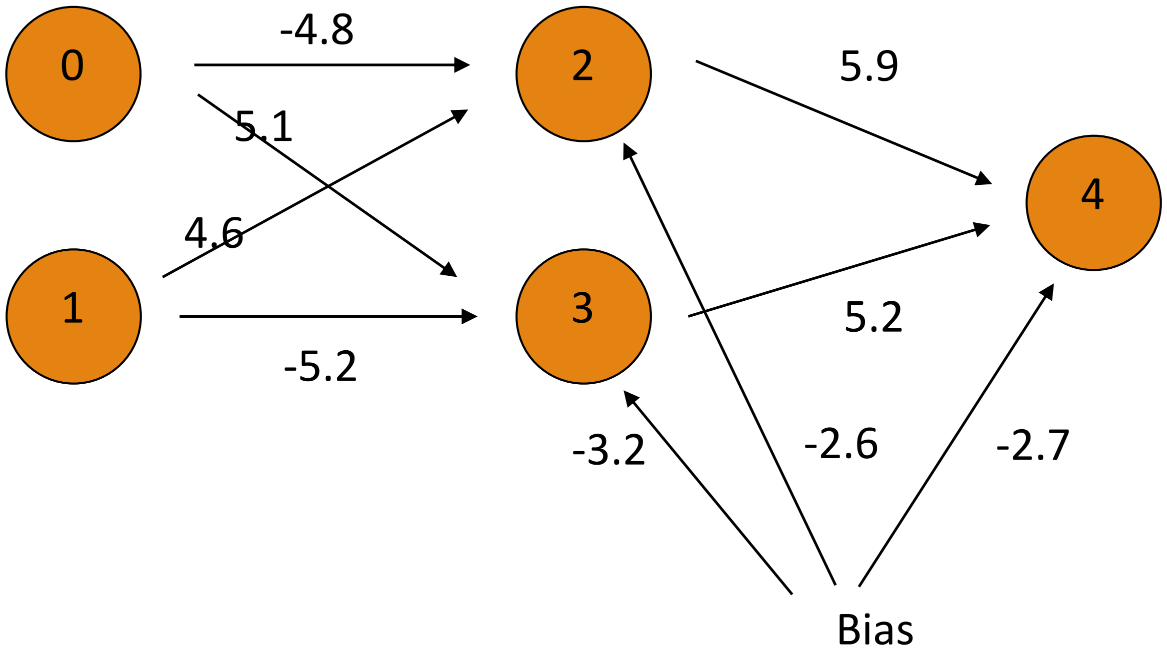
- Node 3 activation is φ(-5.2 -3.2)= 0.000224817



ANN Forward Propagation

Input (0,1)

- Node 4 activation is $\varphi(5.9 \cdot 0.153269 + 5.2 \cdot 0.000224817 - 2.7) = 0.923992$



Neuron Model

Firing Rules:

- Threshold rules:
  - Calculate weighted average of input
  - Fire if larger than threshold
- Perceptron rule
  - Calculate weighted average of input input
  - Output activation level is

$$\phi(\nu) = \begin{cases} 1 & \nu \geq \frac{1}{2} \\ \nu & 0 \leq \nu \leq \frac{1}{2} \\ 0 & \nu \leq 0 \end{cases}$$

Neuron Model

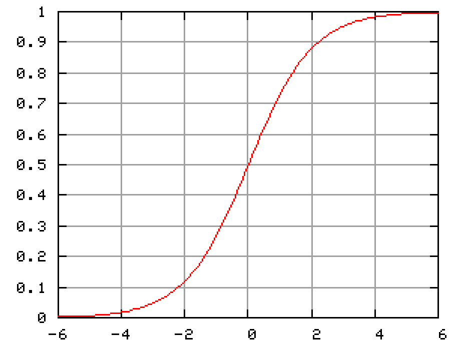
Firing Rules: Sigmoid functions:

- Hyperbolic tangent function

$$\varphi(v) = \tanh(v/2) = \frac{1-\exp(-v)}{1+\exp(-v)}$$

- Logistic activation function

$$\varphi(v) = \frac{1}{1+\exp(-v)}$$



ANN Forward Propagation

Network can learn a non-linearly separated set of outputs.

Need to map output (real value) into binary values.

ANN Training

Weights are determined by *training*

- Back-propagation:
  - On given input, compare actual output to desired output.
  - Adjust weights to output nodes.
  - Work backwards through the various layers
- Start out with initial random weights
  - Best to keep weights close to zero (<<10)

ANN Training

Weights are determined by *training*

- Need a training set
  - Should be representative of the problem
- During each training epoch:
  - Submit training set element as input
  - Calculate the error for the output neurons
  - Calculate average error during epoch
  - Adjust weights

ANN Training

Error is the mean square of differences in output layer

$$E(\vec{x}) = \frac{1}{2}\sum_{k=1}^{K}(y_k(\vec{x}) - t_k(\vec{x}))^2$$

y – observed output

t – target output

ANN Training

Error of training epoch is the average of all errors.

ANN Training

Update weights and thresholds using

- Weights

$$w_{j,k} = w_{j,k} + (-\eta)\frac{\partial E(\vec{x})}{\partial w_{jk}}$$

- Bias

$$\theta_k = \theta_k + (-\eta)\frac{\partial E(\vec{x})}{\partial \theta_k}$$

- η is a possibly time-dependent factor that should prevent overcorrection

ANN Training

Using a sigmoid function, we get

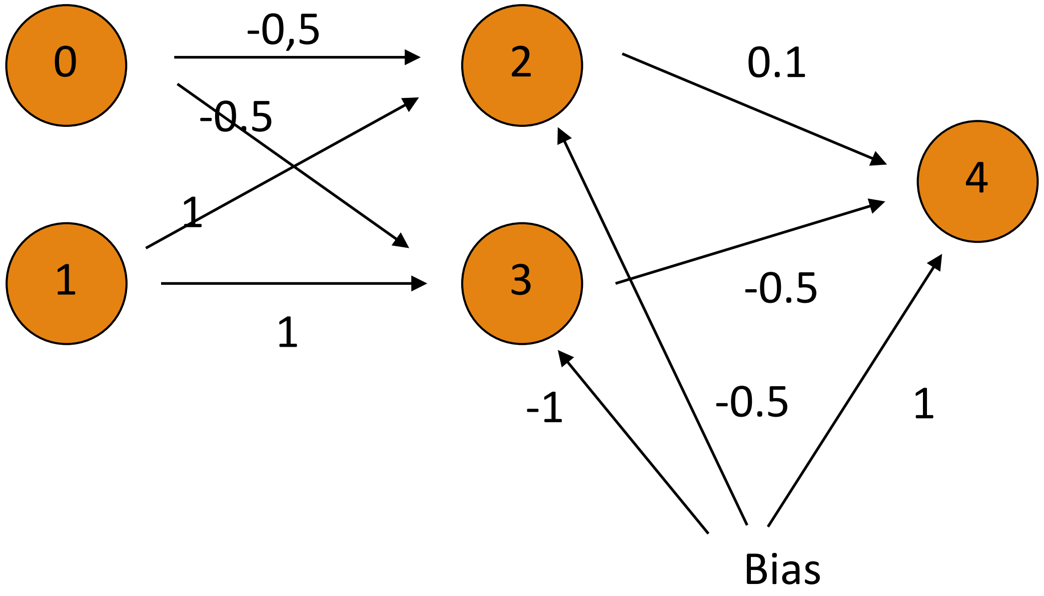
$$\frac{\partial E(\vec{x})}{\partial w_{jk}} = -y_j \delta_j$$

$$\delta_j = f'(\text{net}_j)(t_j - y_j)$$

- Logistics function φ has derivative φ’(t) = φ(t)(1- φ(t))

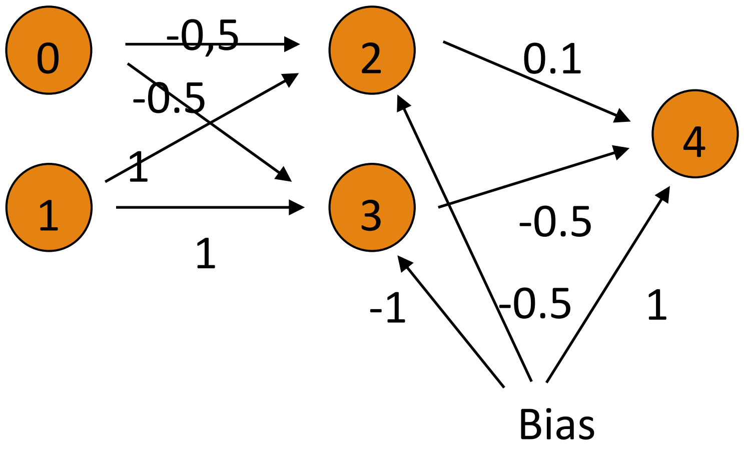
ANN Training Example

Start out with random, small weights



x1	x2	y
0	0	0.687349
0	1	0.667459
1	0	0.698070
1	1	0.676727

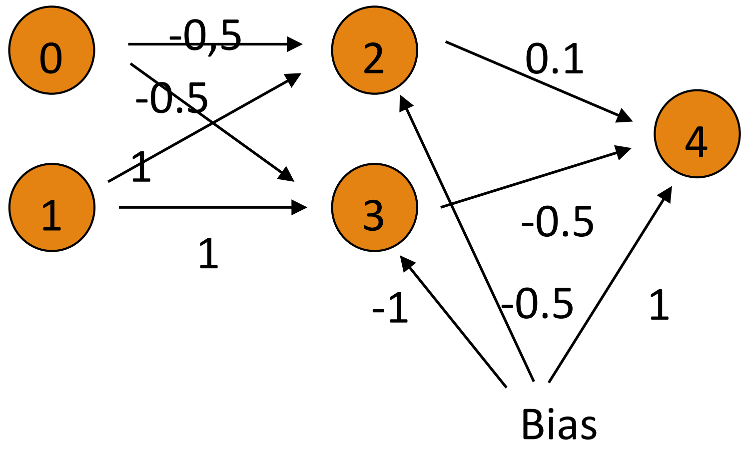
ANN Training Example



x1	x2	y	Error
0	0	0.69	0.472448
0	1	0.67	0.110583
1	0	0.70	0.0911618
1	1	0.68	0.457959

Average Error is 0.283038

ANN Training Example



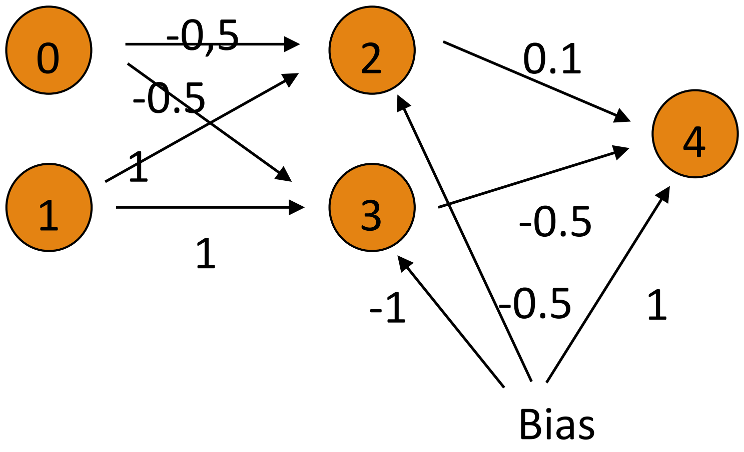
x1	x2	y	Error
0	0	0.69	0.472448
0	1	0.67	0.110583
1	0	0.70	0.0911618
1	1	0.68	0.457959

Average Error is 0.283038

ANN Training Example

Calculate the derivative of the error with respect to the weights and bias into the output layer neurons

ANN Training Example



New weights going into node 4

We do this for all training inputs, then average out the changes

net_4 is the weighted sum of input going into neuron 4:

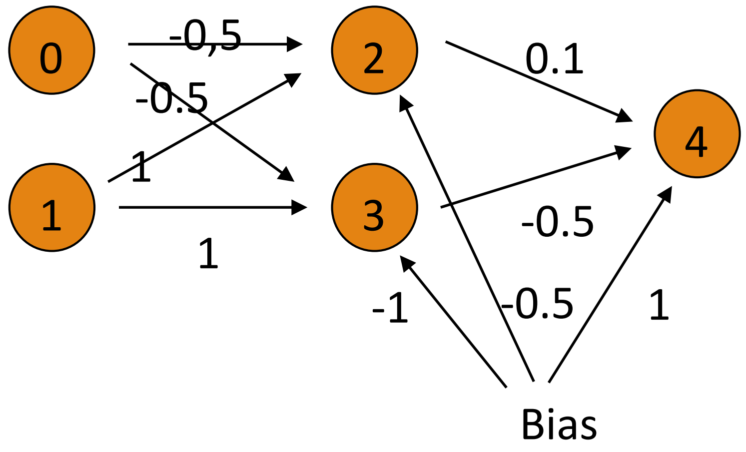
$net_4(0,0)$= 0.787754

net4(0,1)= 0.696717

net4(1,0)= 0.838124

net4(1,1)= 0.73877

ANN Training Example



New weights going into node 4

We calculate the derivative of the activation function at the point given by the net-input.

Recall our cool formula

$$\varphi'(t) = \varphi(t)(1-\varphi(t))$$

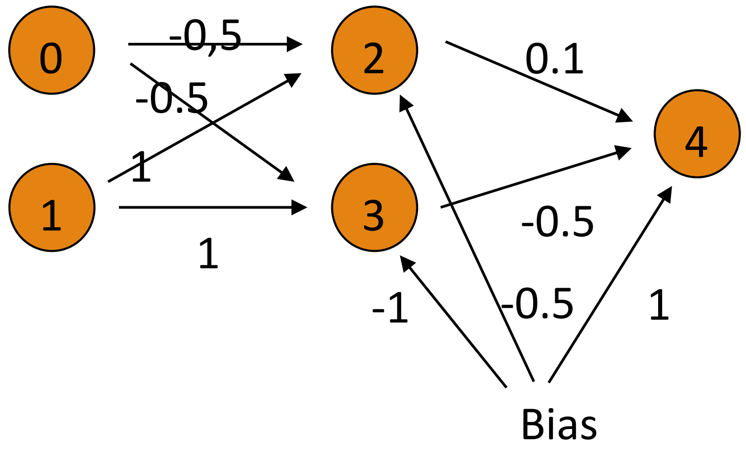
$\varphi'(\, net_4(0,0)) = \varphi'(\, 0.787754) = 0.214900$

$\varphi'(\, net_4(0,1)) = \varphi'(\, 0.696717) = 0.221957$

$\varphi'(\, net_4(1,0)) = \varphi'(\, 0.838124) = 0.210768$

$\varphi'(\, net_4(1,1)) = \varphi'(\, 0.738770) = 0.218768$

ANN Training Example



$$\frac{\partial E(\vec{x})}{\partial w_{jk}} = -y_j \delta_j$$

$$\delta_j = f'(\text{net}_j)(t_j - y_j)$$

New weights going into node 4

We now obtain δ values for each input separately:

Input 0,0:

$\delta_4 = \varphi'(\ \text{net}_4(0,0))\ *(0-y_4(0,0)) = -0.152928$

Input 0,1:

$\delta_4 = \varphi'(\ \text{net4}(0,1))\ *(1-y_4(0,1)) = 0.0682324$

Input 1,0:

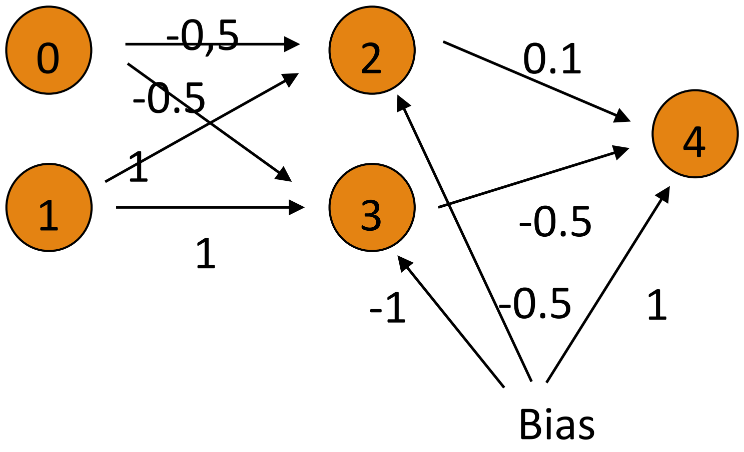
$\delta_4 = \varphi'(\ \text{net}_4(1,0))\ *(1-y_4(1,0)) = 0.0593889$

Input 1,1:

$\delta_4 = \varphi'(\ \text{net}_4(1,1))\ *(0-y_4(1,1)) = -0.153776$

Average: $\delta_4 = -0.0447706$

ANN Training Example



$$\frac{\partial E(\vec{x})}{\partial w_{jk}} = -y_j \delta_j$$

$$\delta_j = f'(\text{net}_j)(t_j - y_j)$$

New weights going into node 4

Average: δ_4 = -0.0447706

We can now update the weights going into node 4:

Let's call: E_{ji} the derivative of the error function with respect to the weight going from neuron *i* into neuron *j*.

We do this for every possible input:

$E_{4,2}$ = - output(neuron(2)* δ_4

For (0,0): $E_{4,2}$ = 0.0577366

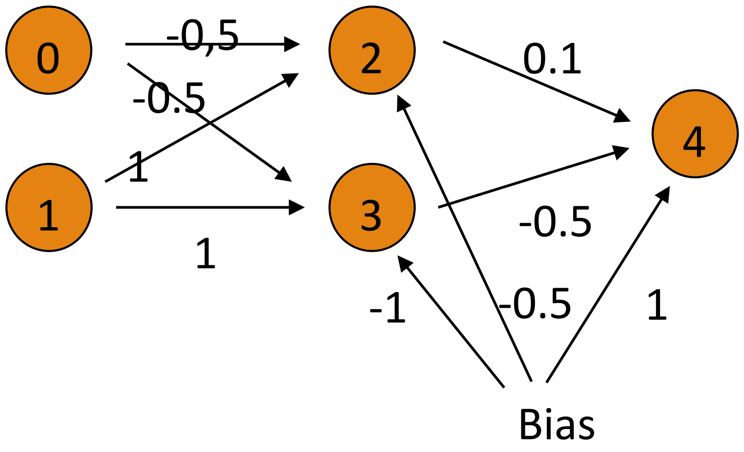
For (0,1): $E_{4,2}$ = -0.0424719

For(1,0): $E_{4,2}$ = -0.0159721

For(1,1): $E_{4,2}$ = 0.0768878

Average is 0.0190451

ANN Training Example

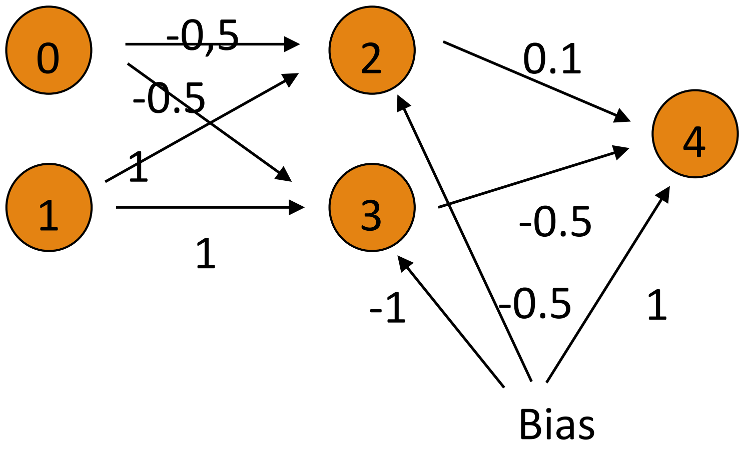


New weight from 2 to 4 is now going to be 0.1190451.

$$\frac{\partial E(\vec{x})}{\partial w_{jk}} = -y_j \delta_j$$

$$\delta_j = f'(\text{net}_j)(t_j - y_j)$$

ANN Training Example



New weights going into node 4

For (0,0): $E_{4,3}$ = 0.0411287

For (0,1): $E_{4,3}$ = -0.0341162

For(1,0): $E_{4,3}$ = -0.0108341

For(1,1): $E_{4,3}$ = 0.0580565

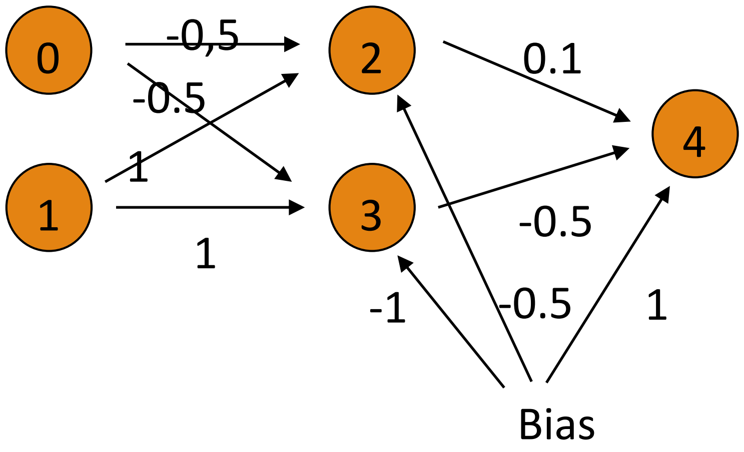
Average is 0.0135588

New weight is -0.486441

$$\frac{\partial E(\vec{x})}{\partial w_{jk}} = -y_j \delta_j$$

$$\delta_j = f'(\text{net}_j)(t_j - y_j)$$

ANN Training Example



$$\frac{\partial E(\vec{x})}{\partial w_{jk}} = -y_j \delta_j$$

$$\delta_j = f'(\text{net}_j)(t_j - y_j)$$

New weights going into node 4:

We also need to change the bias node

For (0,0): $E_{4,B}$ = 0.0411287

For (0,1): $E_{4,B}$ = -0.0341162

For(1,0): $E_{4,B}$ = -0.0108341

For(1,1): $E_{4,B}$ = 0.0580565

Average is 0.0447706

New weight is 1.0447706

ANN Training Example

We now have adjusted all the weights into the output layer.

Next, we adjust the hidden layer

The target output is given by the delta values of the output layer

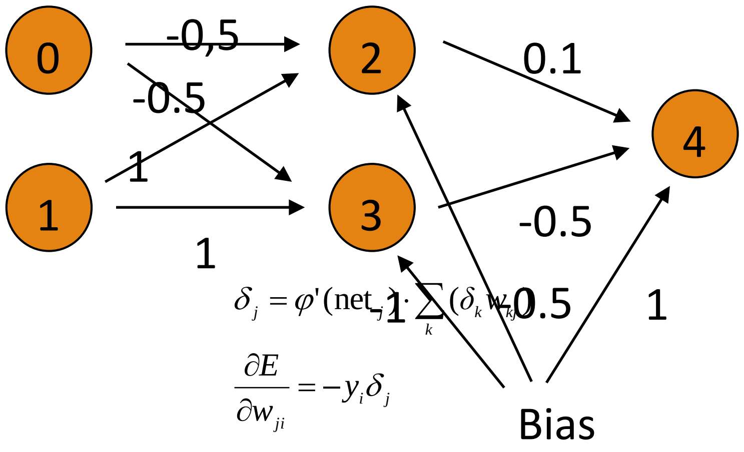
More formally:

- Assume that *j* is a hidden neuron
- Assume that δ_k is the delta-value for an output neuron *k*.
- While the example has only one output neuron, most ANN have more. When we sum over *k,* this means summing over all output neurons.
- w_{kj} is the weight from neuron *j* into neuron *k*

$$\delta_j = \varphi'(\text{net}_j) \cdot \sum_k (\delta_k w_{kj})$$

$$\frac{\partial E}{\partial w_{ji}} = -y_i \delta_j$$

ANN Training Example



$$\delta_j = \varphi'(\text{net}_j) \cdot \sum_k (\delta_k w_{kj})$$

$$\frac{\partial E}{\partial w_{ji}} = -y_i \delta_j$$

We now calculate the updates to the weights of neuron 2.

First, we calculate the net-input into 2.

This is really simple because it is just a linear functions of the arguments x_1 and x_2

$net_2 = -0.5\ x_1 + x2 - 0.5$

We obtain

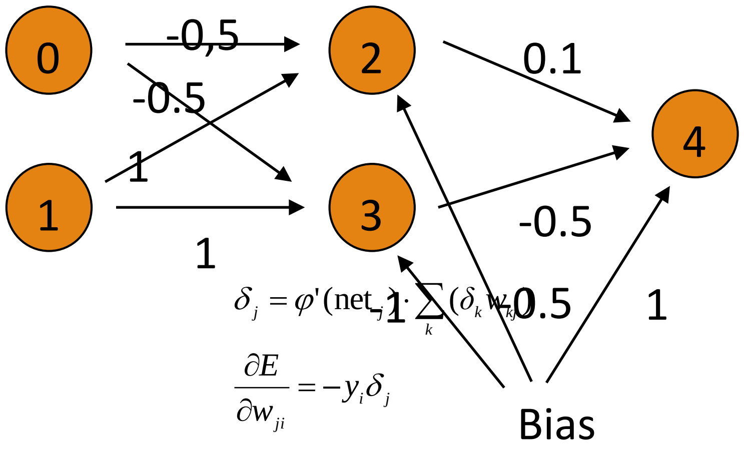
$\delta_2\ (0,0) = -\ 0.00359387$

$\delta_2(0,1) = \ \ 0.00160349$

$\delta_2(1,0) = \ \ 0.00116766$

$\delta_2(1,1) = \ -\ 0.00384439$

ANN Training Example



Call E_{20} the derivative of E with respect to w_{20}. We use the output activation for the neurons in the previous layer (which happens to be the input layer)

$E_{20}(0,0) = -\varphi(0)\cdot\delta_2(0,0) = 0.00179694$

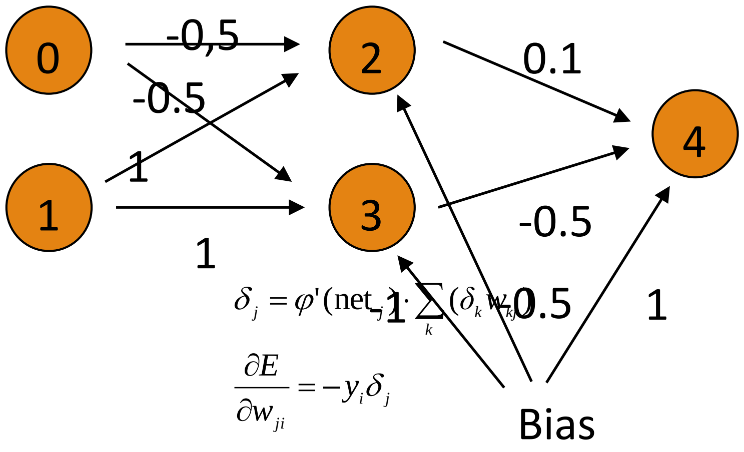
$E_{20}(0,1) = 0.00179694$

$E_{20}(1,0) = -0.000853626$

$E_{20}(1,1) = 0.00281047$

The average is 0.00073801 and the new weight is -0.499262

ANN Training Example



$$\delta_j = \varphi'(\mathrm{net}_j) \cdot \sum_k (\delta_k w_{kj})$$

$$\frac{\partial E}{\partial w_{ji}} = -y_i \delta_j$$

Call E_{21} the derivative of E with respect to w_{21}. We use the output activation for the neurons in the previous layer (which happens to be the input layer)

$E_{21}(0,0) = -\varphi(1) \cdot \delta_2(0,0) =$ 0.00179694

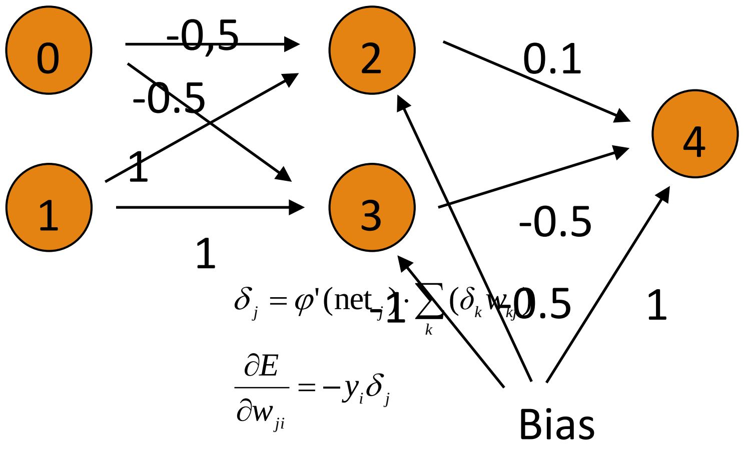
$E_{21}(0,1) =$ -0.00117224

$E_{21}(1,0) =$ -0.000583829

$E_{21}(1,1) =$ 0.00281047

The average is 0.000712835 and the new weight is 1.00071

ANN Training Example



$$\delta_j = \varphi'(\mathrm{net}_j) \cdot \sum_k (\delta_k w_{kj})$$

$$\frac{\partial E}{\partial w_{ji}} = -y_i \delta_j$$

Call E_{2B} the derivative of E with respect to w_{2B}.
Bias output is always -0.5

$E_{2B}(0,0) = - -0.5 \cdot \delta_2(0,0) =$ 0.00179694

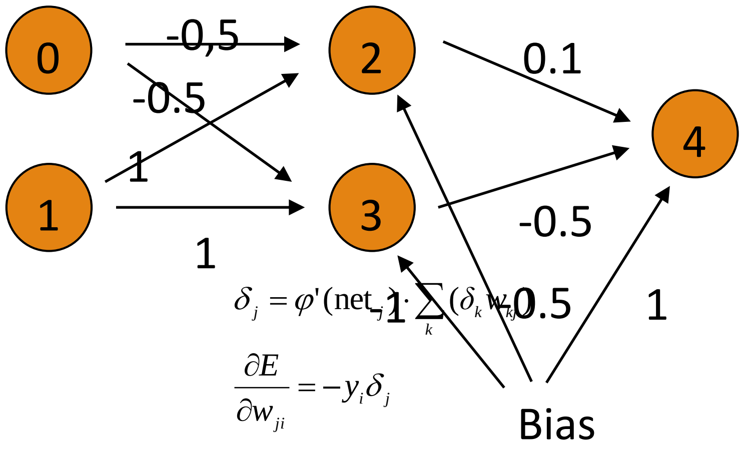
$E_{2B}(0,1) =$ -0.00117224

$E_{2B}(1,0) =$ -0.000583829

$E_{2B}(1,1) =$ 0.00281047

The average is 0.00058339 and the new weight is -0.499417

ANN Training Example



$$\delta_j = \varphi'(\text{net}_j) \cdot \sum_k (\delta_k w_{kj})$$

$$\frac{\partial E}{\partial w_{ji}} = -y_i \delta_j$$

We now calculate the updates to the weights of neuron 3.

…

ANN Training

ANN Back-propagation is an empirical algorithm

ANN Training

XOR is too simple an example, since quality of ANN is measured on a finite sets of inputs.

More relevant are ANN that are trained on a training set and unleashed on real data

ANN Training

Need to measure effectiveness of training

- Need training sets
- Need test sets.

There can be no interaction between test sets and training sets.

- Example of a Mistake:
  - Train ANN on training set.
  - Test ANN on test set.
  - Results are poor.
  - Go back to training ANN.
- After this, there is no assurance that ANN will work well in practice.
  - In a subtle way, the test set has become part of the training set.

ANN Training

Convergence

- ANN back propagation uses gradient decent.
- Naïve implementations can
  - overcorrect weights
  - undercorrect weights
- In either case, convergence can be poor

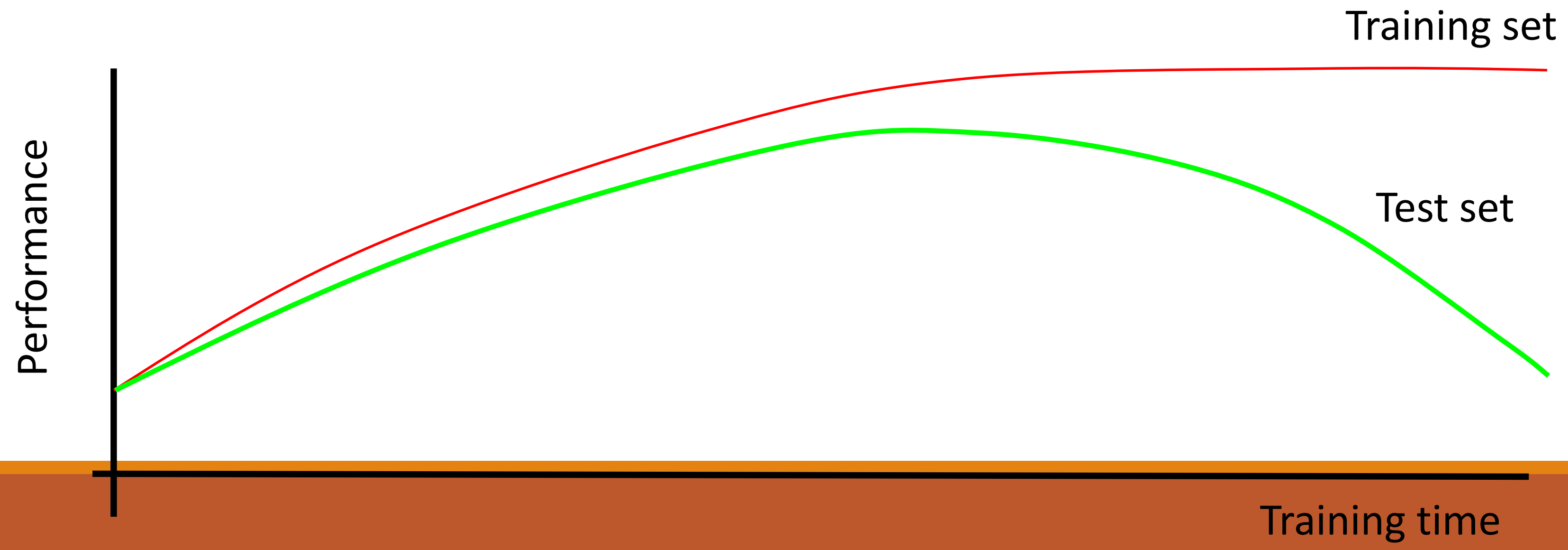
Stuck in the wrong place

- ANN starts with random weights and improves them
- If improvement stops, we stop algorithm
- No guarantee that we found the best set of weights
- Could be stuck in a local minimum

ANN Training

Overtraining

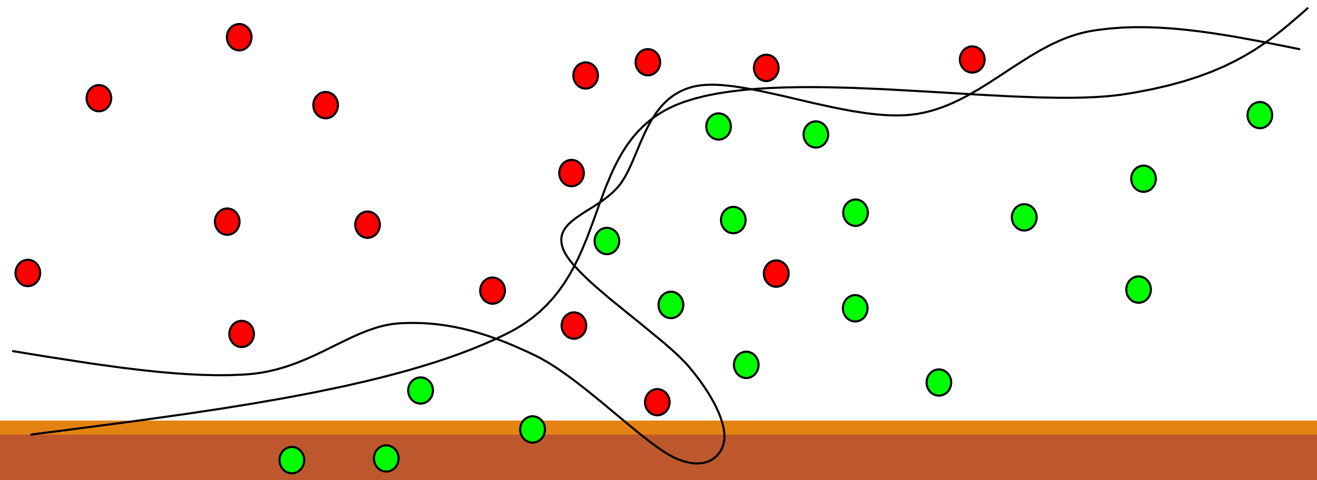
- An ANN can be made to work too well on a training set
- But loose performance on test sets



ANN Training

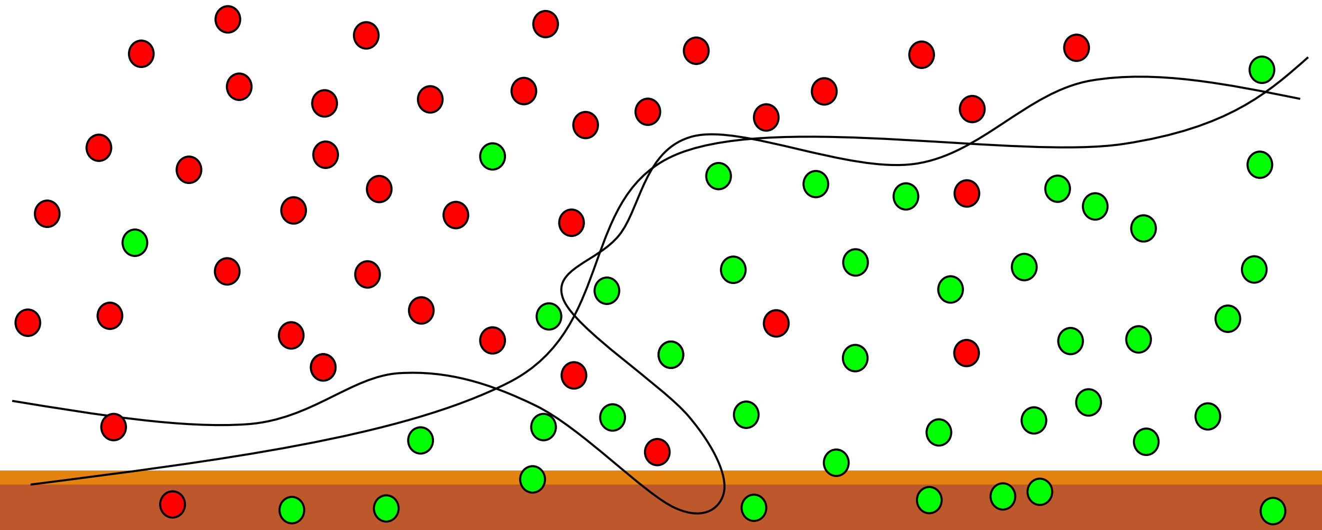
Overtraining

- Assume we want to separate the red from the green dots.
- Eventually, the network will learn to do well in the training case
- But have learnt only the particularities of our training set



ANN Training

Overtraining



ANN Training

Improving Convergence

- Many Operations Research Tools apply
  - Simulated annealing
  - Sophisticated gradient descent

ANN – Issues

Number of layers

- Apparently, three layers is almost always good enough and better than four layers.
- Also: fewer layers are faster in execution and training

How many hidden nodes?

- Many hidden nodes allow to learn more complicated patterns
- Because of overtraining, almost always best to set the number of hidden nodes too low and then increase their numbers.

ANN - Issues

Interpreting Output

- ANN's output neurons do not give binary values.
  - Good or bad
  - Need to define what is an accept.
- Can indicate *n* degrees of certainty with *n*-1 output neurons.
  - Number of firing output neurons is degree of certainty

ANN Applications

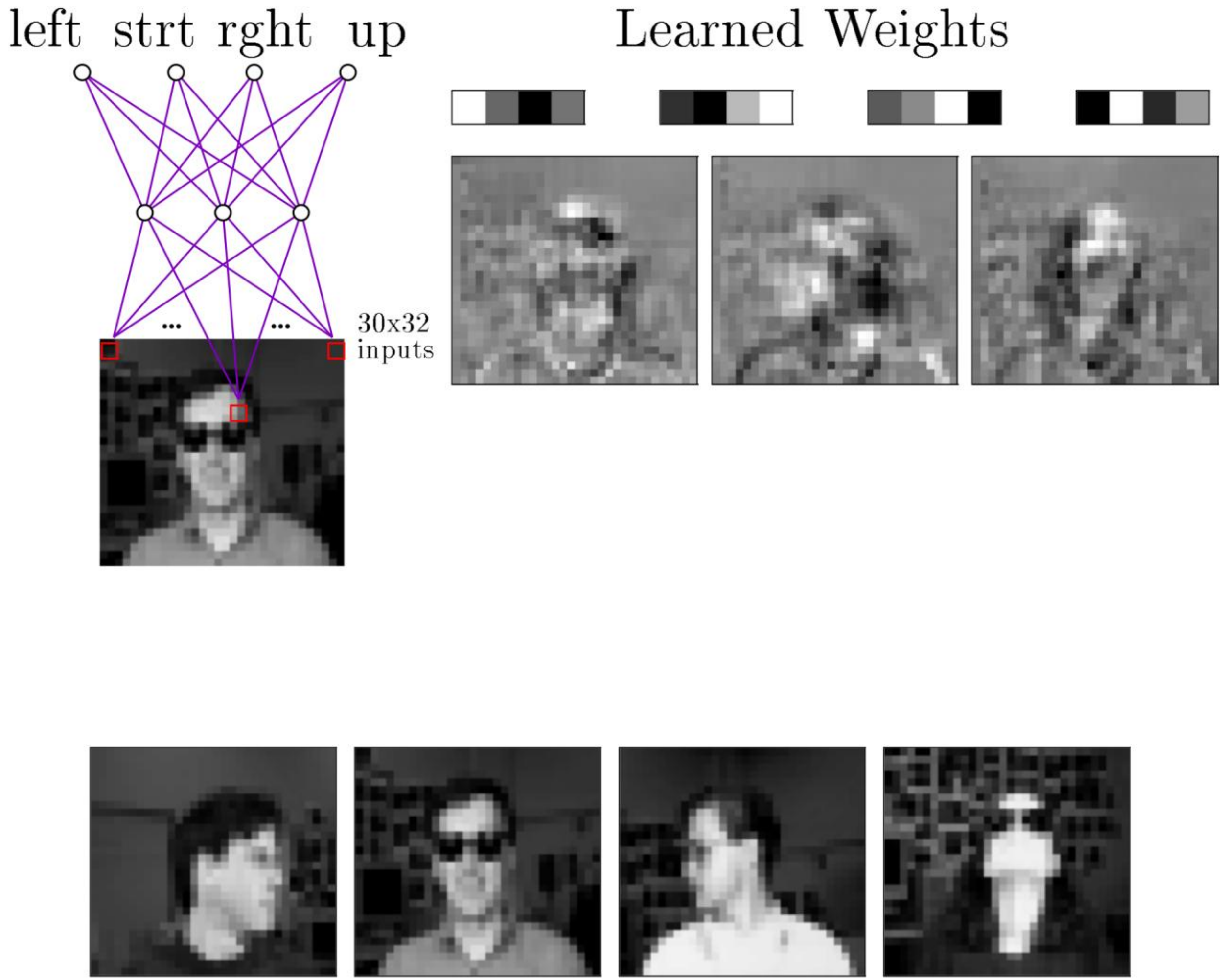
Pattern recognition

- Network attacks
- Breast cancer
- Object and face detection
- handwriting recognition

Pattern completion

Auto-association

- ANN trained to reproduce input as output
  - Noise reduction
  - Compression
  - Finding anomalies



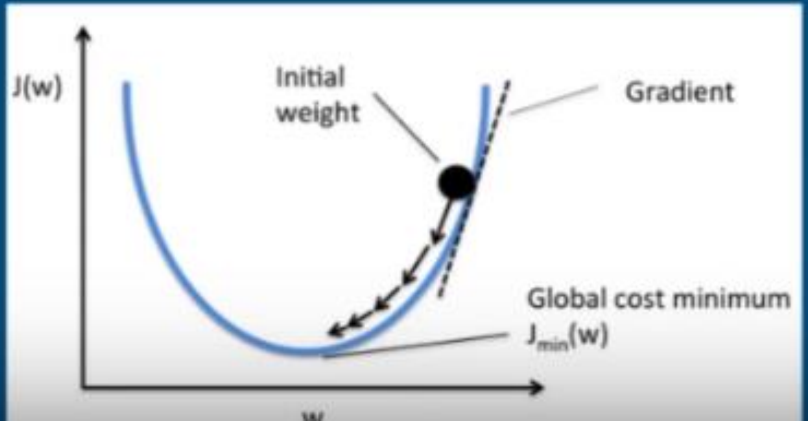
Typical input images

http://www.cs.cmu.edu/~tom/faces.html

Thank You

Gradient Descent Tutorial

- Optimization method
- Used extensively in deep learning, useful in a wide variety of situations
- Idea:
- You have a function you want to minimize, J(w) = cost or error
- Can maximize things too, just switch signs



Example

$J = w^2$

(we know min is at w = 0, but let's pretend we don't)

$dJ/dw = 2w$, set initial $w = 20$, learning rate = 0.1

Iteration 1: $w \leftarrow 20 - 0.1*40 = 16$
Iteration 2: $w \leftarrow 16 - 0.1*2*16 = 12.8$
Iteration 3: $w \leftarrow 12.8 - 0.1*2*12.8 = 10.24$