CS-540: Intro to Artificial Intelligence

Multi-layer Neural Networks

Lecturer: Erin Winter

You survived the exam!

It will handed back shortly after the break. You'll also have midterm evaluations to do.

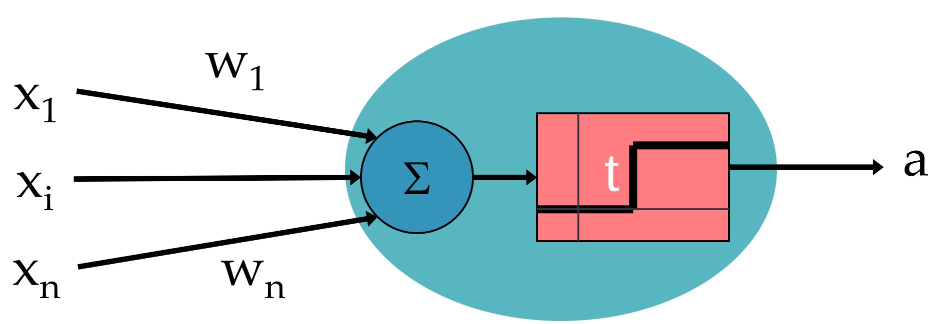
Perceptrons

"**1-layer network**": one or more *output units*

"Input units" don't count because they don't compute anything

Output units are all **linear threshold units** (**LTU**s)

- a unit's inputs, $\boldsymbol{x_i}$, are weighted, $\boldsymbol{w_i}$, and **linearly combined**
- **step** function computes binary output activation value, ***a***



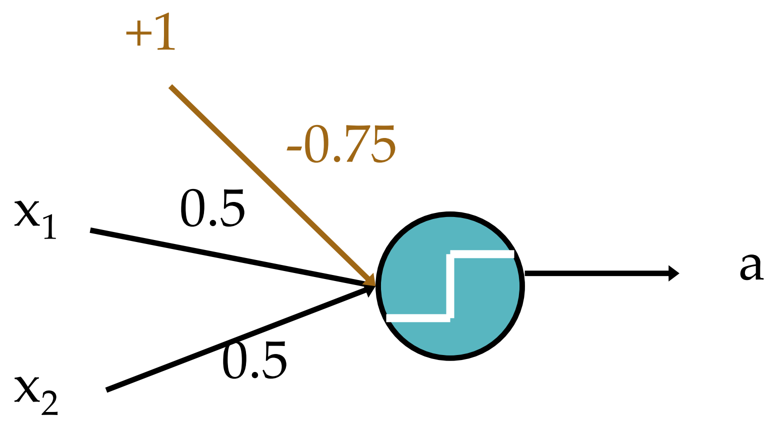
Perceptron Examples

“AND” Perceptron:

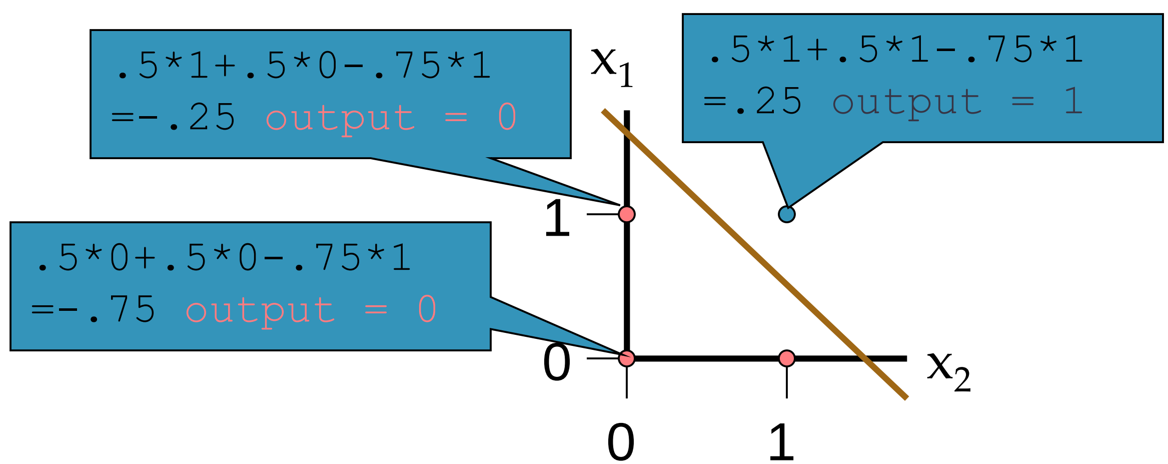
inputs are 0 or 1

output is 1 when

both x_1 and x_2 are 1



- 2D input space
  - 4 possible data points
  - weights define linear decision boundary



Perceptron Learning

How are the weights learned by a Perceptron?

- Programmer specifies:
  - numbers of units in each layer
  - connectivity between units
- Only unknown is the set of weights
- Learning of weights is ***supervised***
  - for each training example
    a list of values for each input units of the network
  - the correct output is given
    a list of values for the desired output of all output units

Perceptron Learning Algorithm

1. Initialize the weights in the network
   (usually with random values)

2. Repeat until all examples correctly classified
   or some other stopping criterion is met

   foreach example, ***e***, in the training set **do**

   O = neural_net_output(network, e);

   T = desired output; // Target or Teacher output

   update_weights(***e***, O, T);

- Each pass through *all* of the training examples is called an epoch
- Step 2 requires multiple epochs

Perceptron Learning Rule

How should the weights be updated?

$$w_i = w_i + \Delta w_i$$

where $\Delta w_i = \alpha\, x_i\, (T - O)$

x_i is the input associated with i^{th} input unit

α is a real-valued constant between 0.0 and 1.0 called the **learning rate**

Perceptron Learning Rule Properties

- $\Delta w_i = \alpha x_i (T - O)$ doesn't depend on w_i
- No change in weight (i.e., $\Delta w_i = 0$) if:

– **correct output**, i.e., $T = O$ gives $\times x_i \times 0 = 0$

– **0 input**, i.e., $x_i = 0$ gives $\alpha \times 0 \times (T - O) = 0$

- If T=1 and O=0, *increase* the weight

  so that maybe next time the result will exceed the output unit's threshold, causing it to be 1

- If T=0 and O=1, *decrease* the weight

  so that maybe next time the result won't exceed the output unit's threshold, causing it to be 0

Perceptron Learning Rule (PLR)

PLR is a “local" learning rule in that only local information in the network is needed to update a weight

PLR performs gradient descent (hill-climbing) in “weight space”

Iteratively adjusts all weights so that for each training example the error decreases (more correctly, error is monotonically non-increasing)

Perceptron Learning Rule (PLR)

Perceptron Convergence Theorem:

- *If a set of examples is learnable, then PLR will find an appropriate set of weights*
  - in a finite number of steps
  - independent of the initial weights
  - Using a sufficiently small value for α

- This theorem says that if a solution exists, PLR's gradient descent is guaranteed to find an optimal solution (i.e., 100% correct classification) for any 1-layer neural network

Limits of Perceptron Learning

What Can be Learned by a Perceptron?

- Perceptron's output is determined by the separating hyperplane (linear decision boundary) defined by

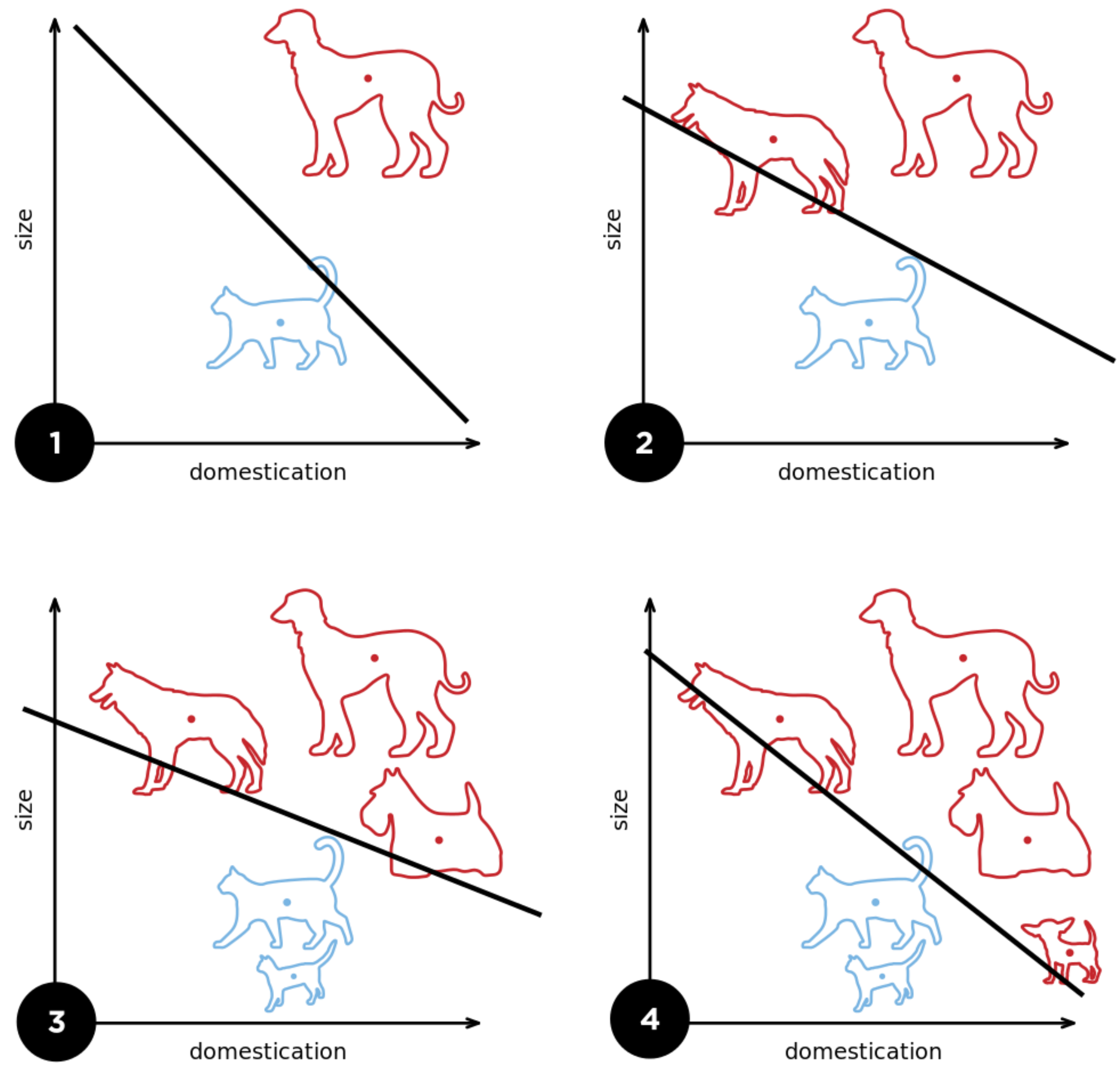
$$(w_1\, x_1) + (w_2\, x_2) + \ldots + (w_n\, x_n) \;=\; t$$

- So, Perceptrons can only learn functions that are linearly separable (in **input space**)

Beyond Perceptrons

Perceptrons are too weak a computing model because they **can only learn linearly-separable functions**

Most real-world data is *not* linearly-separable in d-dimensions aka input space



Beyond Perceptrons

- A ***Multi-Layer, Feed-Forward Network*** computes a function of the inputs and the weights
- Input units
  - Input values are given
- Output units
  - activation is the output result
- **Hidden units** (between input and output units)
  - cannot observe directly
- Perceptrons have input units followed by one layer of output units, i.e., no hidden units

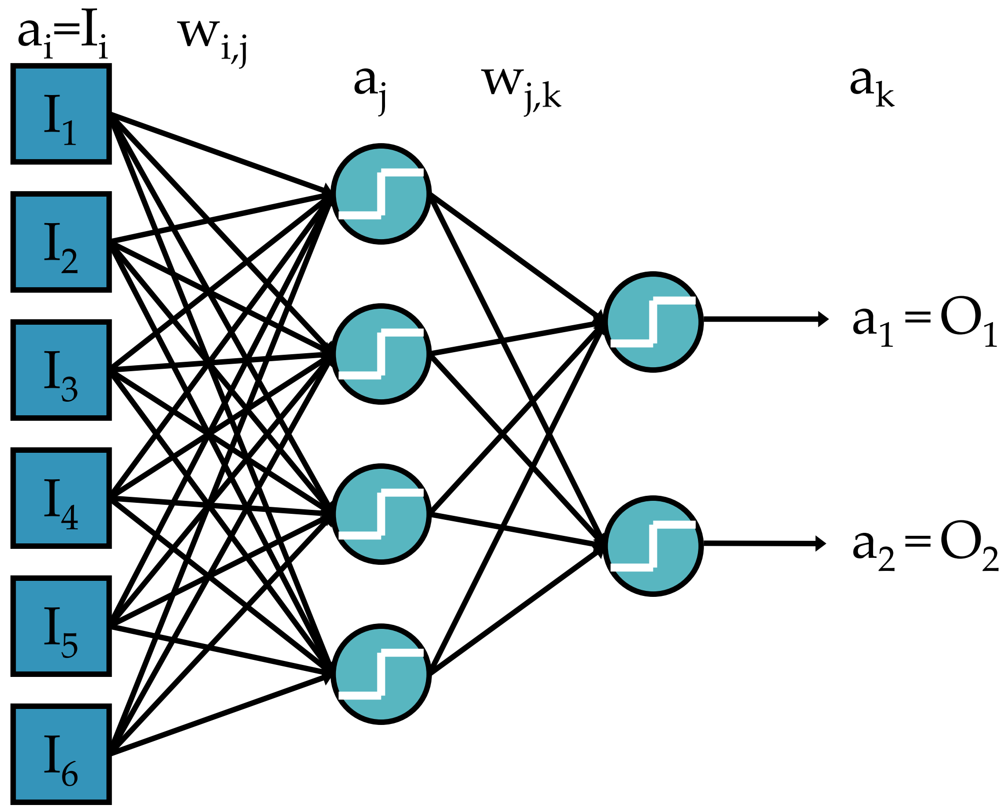
Two-Layer, Feed-Forward Neural Network

Input Units
Hidden Units
Output Units

Weights on links
from input to hidden

Weights on links
from hidden to output

Network Activations



Beyond Perceptrons

NN's with one hidden layer of a sufficient number of units, can compute functions associated with convex classification regions in input space

And can *approximate* any continuous function

NN's with two hidden layers are universal computing units that **can learn any function**, though the function complexity is limited by the number of units

If too few, the network will be unable to represent the function

If too many, the network can memorize examples and is subject to "overfitting"

Driverless Cars (of the 90's)

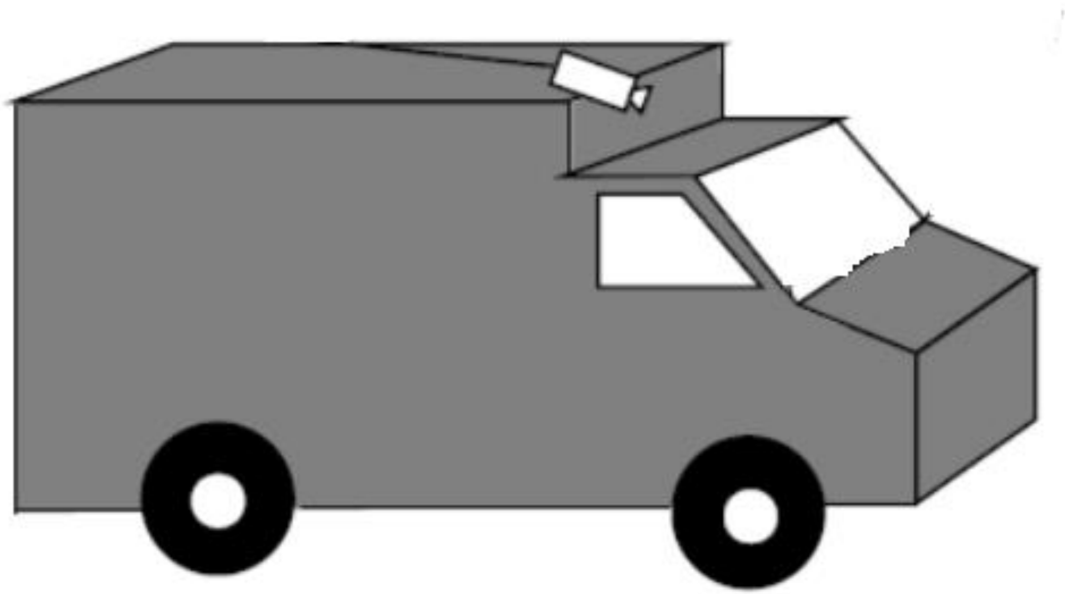
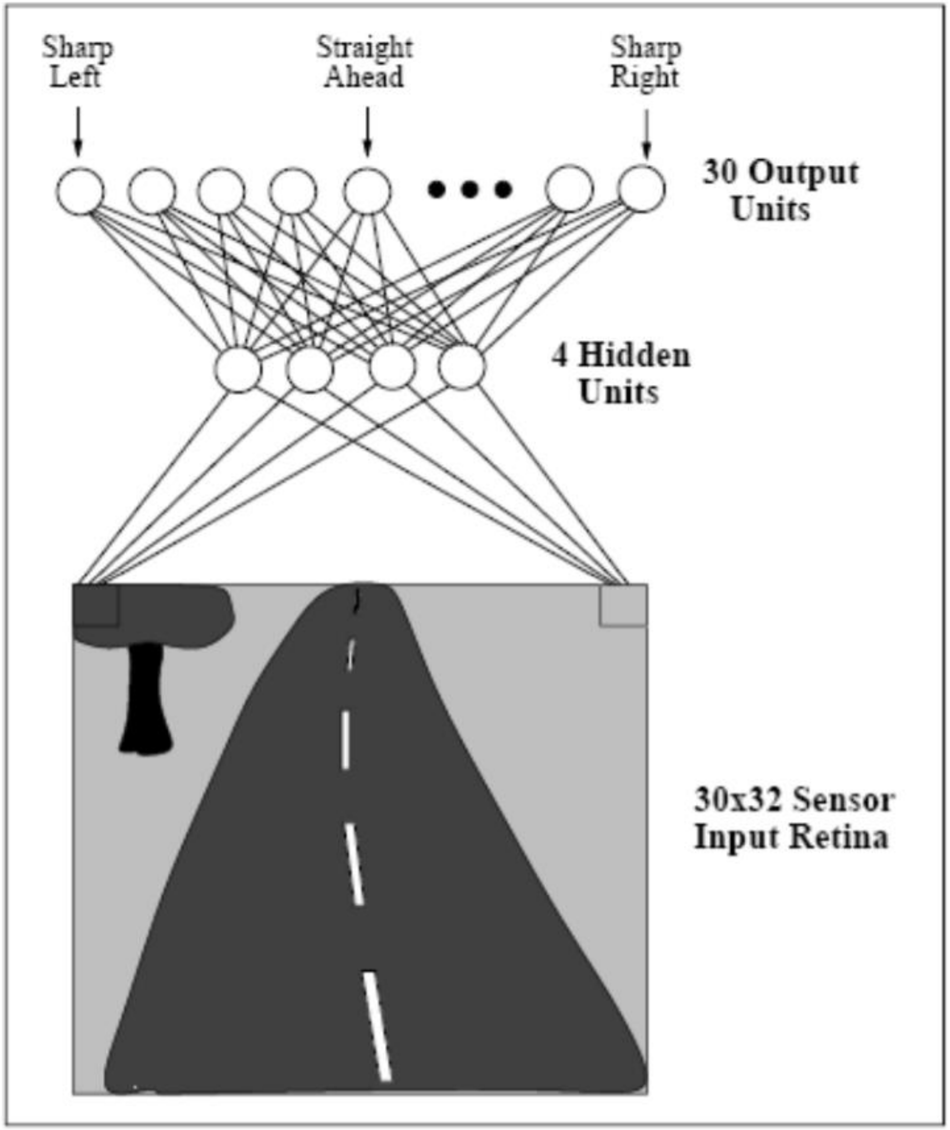


Image from Camera → Steering Direction

Features: Pixels

Class: Ordinal Set of 30 Discrete Rotations

Neural Network Design

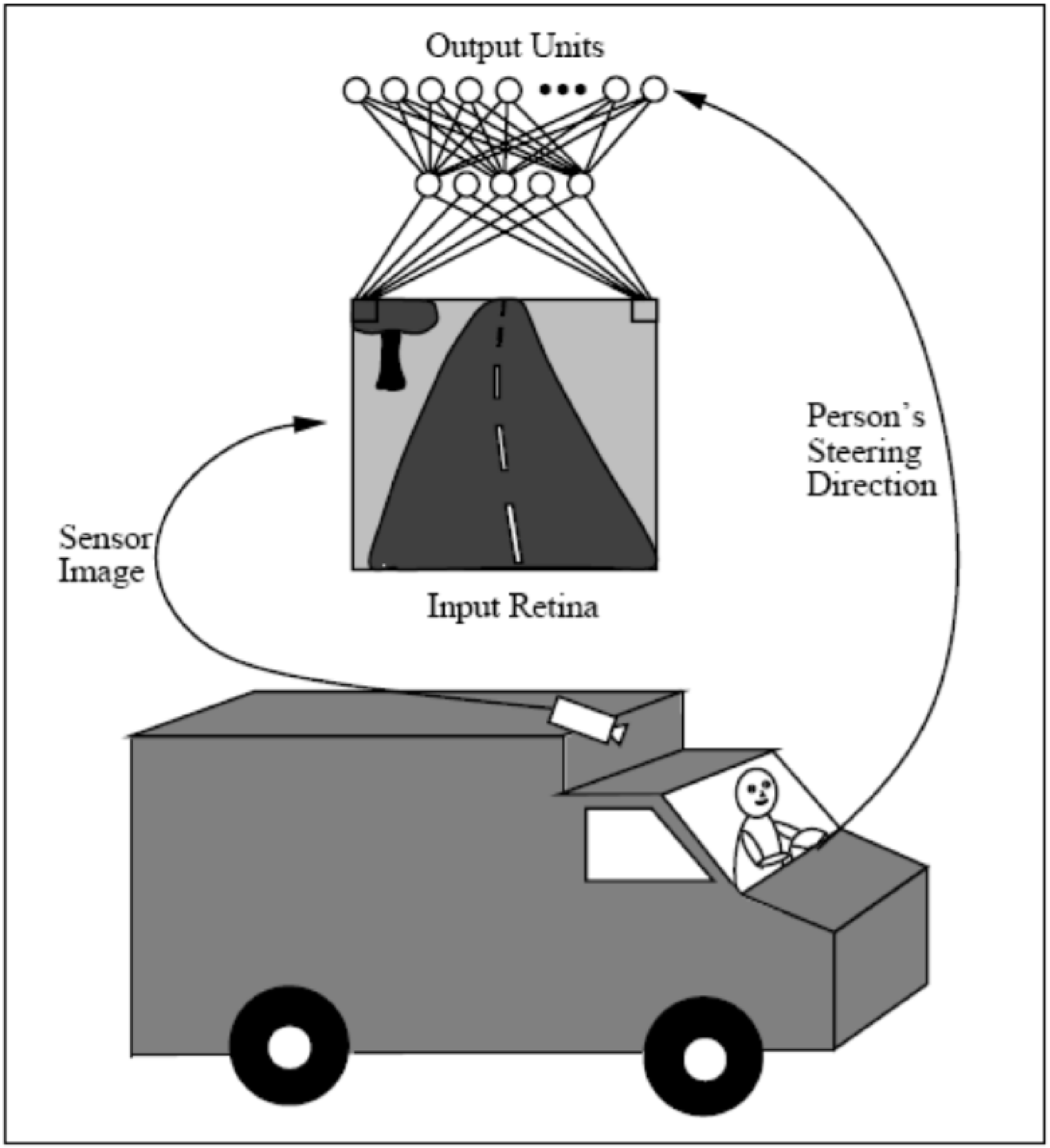


[Pomerleau, 1995]

This amounts to 960 features with continuous values representing the darkness or lightness of each pixel.

Training the Driverless Car

At fixed intervals, record the pixel values from the camera, then human driver response fractions of a second later. Possible to capture many training instances in a single driving session.



[Pomerleau, 1995]

Result: ALVINN

Two-Layer, Feed-Forward Neural Network

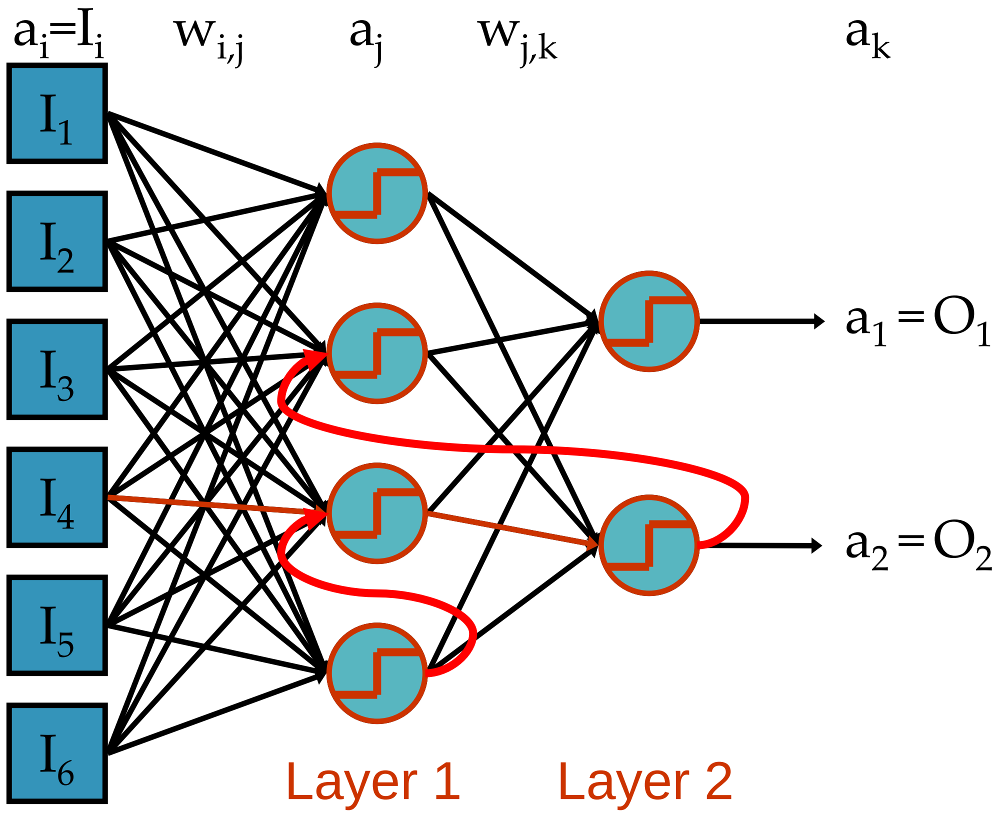
Two Layers:
count layers with units computing an activation

Feed-Forward:
each unit in a layer connects to all units in the next layer

no cycles

- links within the same layer
- links to prior layers

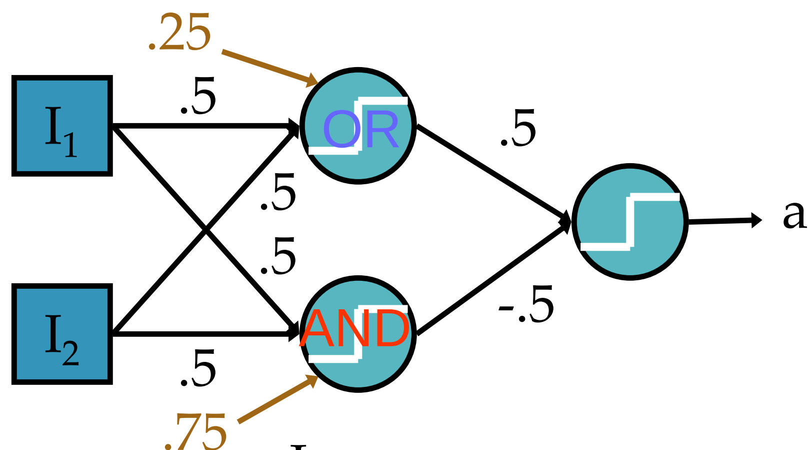
no skipping layers



XOR Example

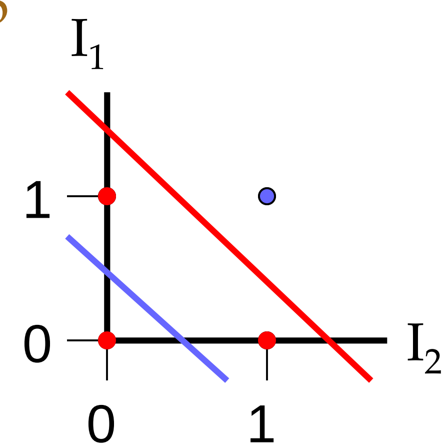
XOR 2-Layer Feed-Forward Network

- inputs are 0 or 1
- output is 1 when I_1 is 1 and I_2 is 0, or I_1 is 0 and I_2 is 1



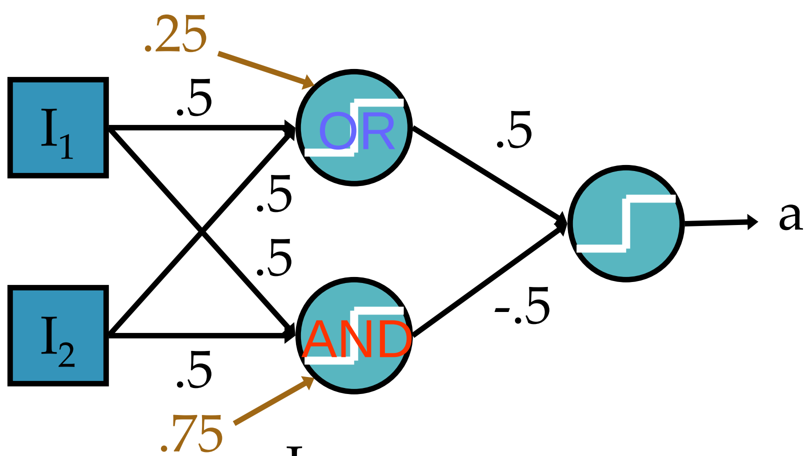
Each unit in hidden layer acts like a Perceptron learning a decision line

- top hidden unit acts like an OR Perceptron
- bottom hidden unit acts like an AND Perceptron



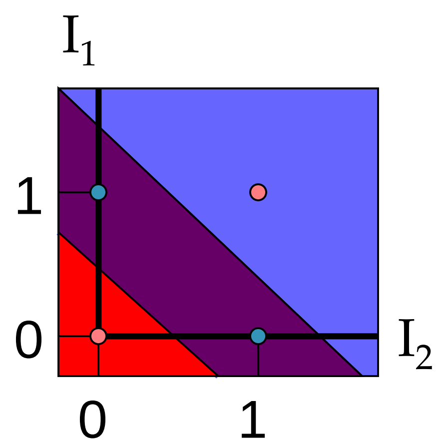
XOR Example

To classify an example each unit in the output layer combines these decision lines by intersecting their "half-planes”:



when OR is 1 and AND is 0

then output, a, is 1



Correct classifier is the intersection of two hyperplanes

Training neural networks with hidden layers

- Perceptron Learning Rule doesn't work in multi-layered feed-forward nets because the desired target values for the hidden units are *not known*

- Must again solve the **Credit Assignment Problem**
  - determine which weights to credit/blame for the output error in the network, and how to update them

Learning in Multi-Layer, Feed-Forward Neural Nets

Back-Propagation

- Method for learning weights in these networks
- Generalizes Perceptron Learning Rule to learn weights in hidden layers

Approach

- **Gradient-descent algorithm** to minimize the total error on the training data
- Errors are propagated through the network starting at the output units and working *backwards* towards the input units

Back-Propagation Algorithm

Initialize the weights in the network (usually random values)

Repeat until stopping criterion is met {

forall p,q in network, $\Delta W_{p,q} = 0$

foreach example ***e*** in training set **do** {

O = neural_net_output(***network***, ***e***) // forward pass

Calculate error (T - O) at the output units // T = teacher output

Compute $\Delta w_{j,k}$ for all weights from hidden to output layer

Compute $\Delta w_{i,j}$ for all weights from inputs to hidden layer

forall p,q in network $\Delta W_{p,q} = \Delta W_{p,q} + \Delta w_{\boldsymbol{p,q}}$

}

backward pass

for all p,q in network $\Delta W_{p,q} = \Delta W_{p,q}$ / num_training_examples

network = update_weights(***network***, $\Delta W_{p,q}$)

}

Note: Uses average gradient for all training examples when updating weights

Back-Prop using Stochastic Gradient Descent (SGD)

Most practitioners use SGD to update weights using the *average gradient computed using a small batch of examples*, and repeating this process for many small batches from the training set

In extreme case, update after *each* example

Called *stochastic* because each small set of examples gives a noisy estimate of the average gradient over *all* training examples

Updating the Weights

Back-Propagation performs a ***gradient descent search*** in "weight space" to learn the network weights

Given a network with *n* weights:

each configuration of weights is a vector, ***W***, of length *n* that defines an instance of the network

W can be considered a point in an *n*-dimensional **weight space**, where each dimension is associated with one of the connections in the network

Updating the Weights

- Given a training set of *m* examples:
  - Each network defined by the vector ***W*** has an associated total error, E, on ***all*** the training data
  - E is the sum squared error (SSE) defined by

$$\boldsymbol{E = E_1 + E_2 + \ldots + E_m}$$

where $\boldsymbol{E_i}$ is the squared error of the network on the $\boldsymbol{i}^{\text{th}}$ training example

- Given n output units in the network:

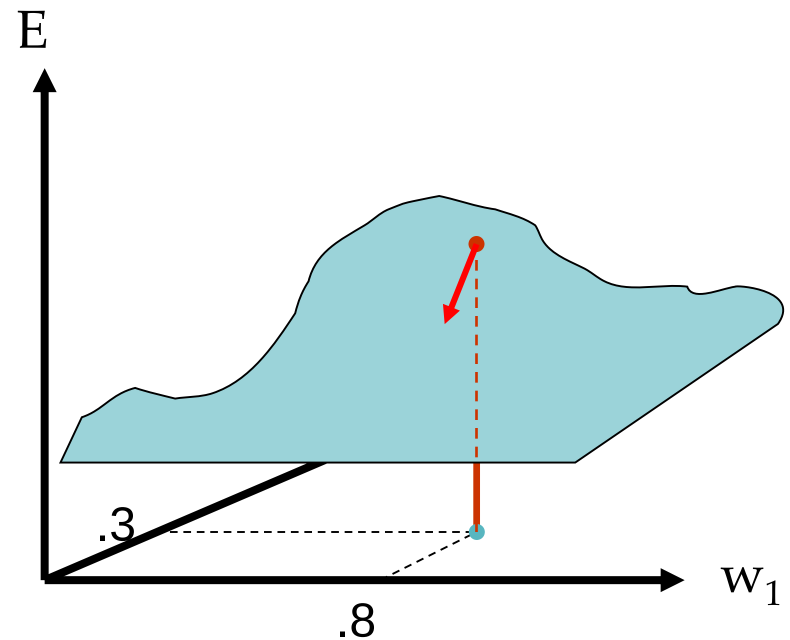
$$E_i = (T_1 - O_1)^2 + (T_2 - O_2)^2 + \ldots + (T_n - O_n)^2$$

  - $\boldsymbol{T_i}$ is the target value for the $\boldsymbol{i}^{\text{th}}$ example
  - $\boldsymbol{O_i}$ is the network output value for the $\boldsymbol{i}^{\text{th}}$ example

Updating the Weights

Visualized as a 2D error surface in “weight space”

- Each point in w_1 w_2 plane is a weight configuration
- Each point has a total error E
- 2D surface represents errors for all weight configurations
- Goal is to find a lower point on the error surface (local minimum)
- Gradient descent follows the direction of steepest descent, i.e., where E decreases the most



Updating the Weights

The **gradient** is defined as

$$\nabla E = [\partial E / \partial w_1, \partial E / \partial w_2, \ldots, \partial E / \partial w_n]$$

Update the ith weight using

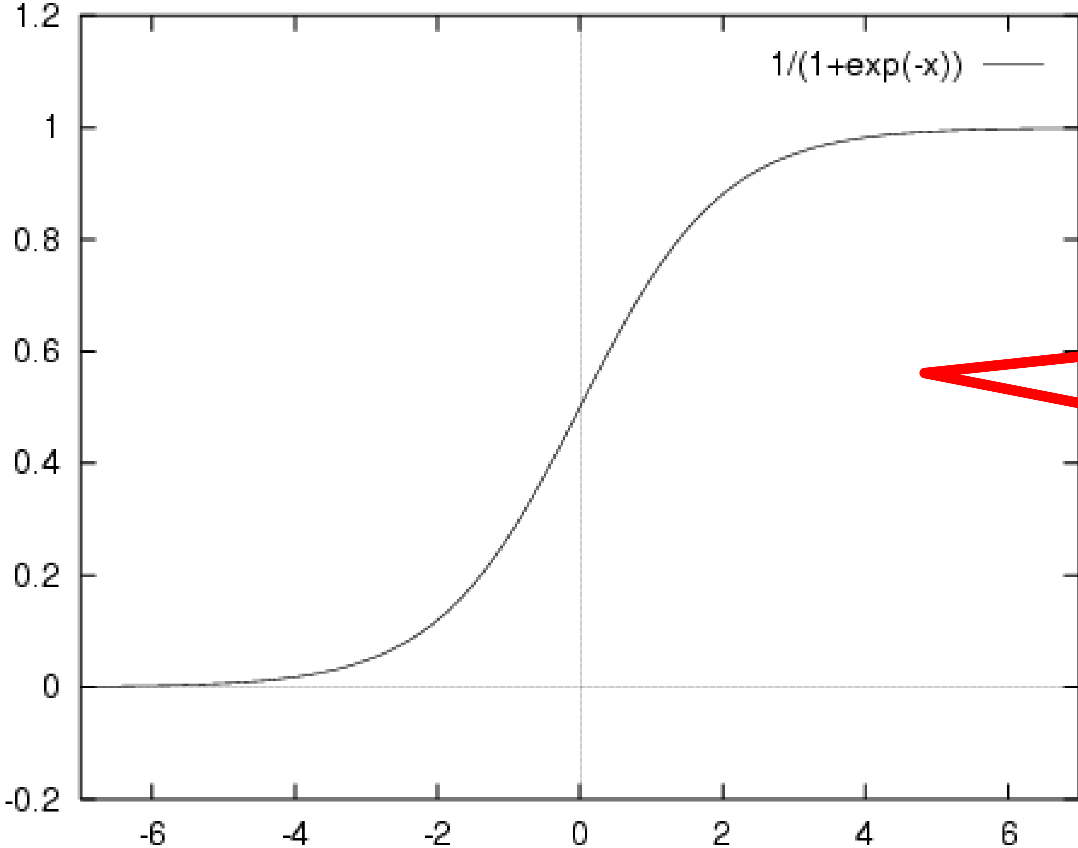
$$\mathrm{D}w_i = - \boldsymbol{a} \, \partial E / \partial w_i$$

Can't use the Step function in LTU's because it’s derivative is 0 everywhere

Instead, let’s use (for now) the **Sigmoid function**

Sigmoid Activation Function

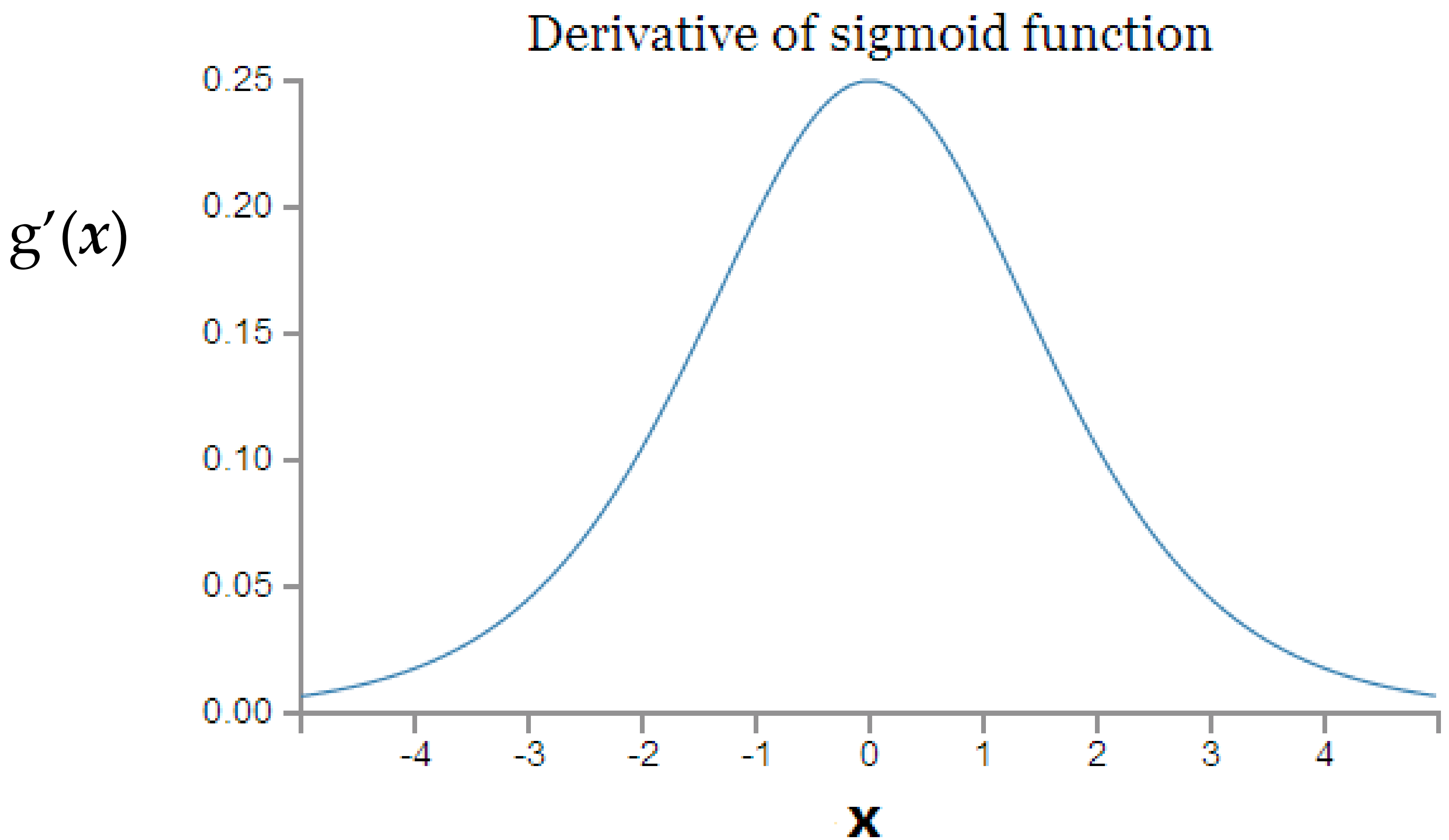
Solution: Replace with a smooth function such as Sigmoid function (aka Logistic Sigmoid function): $g_w(x) = 1 / (1 + e^{-wx})$



Squashes numbers to range [0,1]

First Derivative of Sigmoid Function

$$g'(x) = g(x)\,(1 - g(x))$$

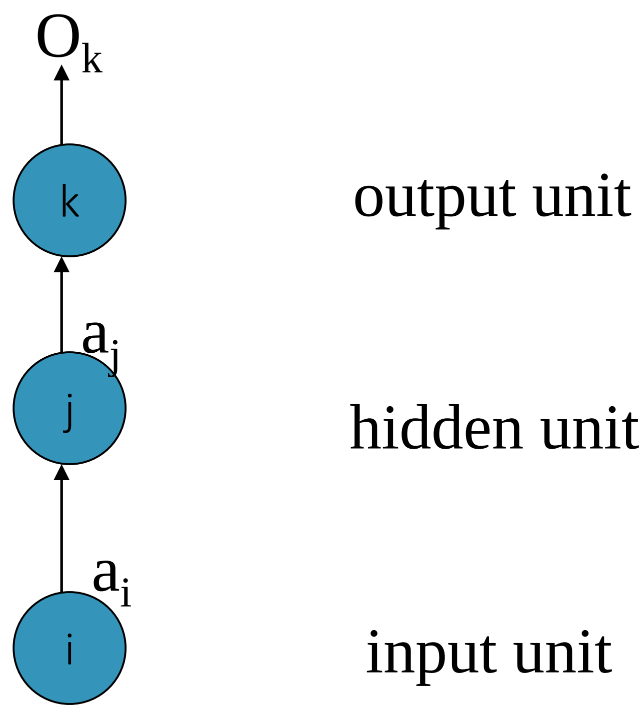


Updating the Weights

Training Label for Instance k

T_k

Propagate the error from layers j,k (hidden to output) to layers i,j (input to hidden). Then update weights using the computed gradient.



Updating Weights in a 2-Layer Neural Network

For **weights between hidden and output units**, generalized PLR for sigmoid activation is

$$Dw_{j,k} = - a\, \partial E / \partial w_{j,k}$$
$$= -\alpha\, -a_j\, (T_k - O_k)\, g'(in_k)$$
$$= \alpha\, a_j\, (T_k - O_k)\, O_k\, (1 - O_k)$$
$$= \alpha\, a_j\, \Delta_k$$

$$\boxed{\Delta_k = Err_k \times g'(in_k)}$$

- $w_{j,k}$ weight on link from hidden unit j to output unit k
- α learning rate parameter
- a_j activation (i.e. output) of hidden unit j
- T_k teacher output for output unit k
- O_k actual output of output unit k
- g' derivative of the sigmoid activation function, which is $g' = g(1 - g)$

Updating Weights in a 2-Layer Neural Network

For **weights between input and hidden units**:

- we don't have teacher-supplied correct output values
- infer the error at these units by "back-propagating"
- error at an output unit is "distributed" back to each of the hidden units in proportion to the weight of the connection between them
- total error is distributed to all of the hidden units that contributed to that error

- *Each hidden unit accumulates some error from each of the output units to which it is connected*

For **weights between inputs and hidden units**:

$$\Delta w_{i,j} = -\alpha\, \partial E / \partial w_{i,j}$$

$$= -\alpha(-a_i) g'(in_j) \sum_k w_{j,k}(T_k - O_k) g'(in_k)$$

$$= \alpha\ a_i\ a_j(1 - a_j) \sum_k w_{j,k}(T_k - O_k)\ O_k(1 - O_k)$$

$$= \alpha\ a_i\ \Delta_j \quad \text{where} \quad \Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$$

$w_{i,j}$ weight on link from input i to hidden unit j

$w_{j,k}$ weight on link from hidden unit j to output unit k

α learning rate parameter

a_j activation (i.e. output) of hidden unit j

T_k teacher output for output unit k

O_k actual output of output unit k

a_i input value i

g' derivative of sigmoid activation function, which is $g' = g(1\text{-}g)$

Back-Propagation Algorithm

Initialize the weights in the network (usually random values)

Repeat until stopping criterion is met

forward pass

foreach example, *e,* in training set **do**

{ O = neural_net_output(network, *e*)

T = desired output, i.e., Target or Teacher's output

calculate error (T - O) at all the output units

compute $\Delta w_{j,k} = \alpha\, a_j\, \Delta_k = \alpha\, a_j\, (T_k - O_k)\, g'(in_k)$

compute $\Delta w_{i,j} = \alpha\, a_i\, \Delta_j = \alpha\, a_i\, g'(in_j) \sum_k w_{j,k}(T_k - O_k) g'(in_k)$

forall *p*, *q* in network $w_{p,q} = w_{p,q} + \Delta w_{p,q}$

}

backward pass

Simplistic SGD: update all weights after each example