CS-540: Intro to Artificial Intelligence

Multi-layer Neural Networks Continued

Lecturer: Erin Winter

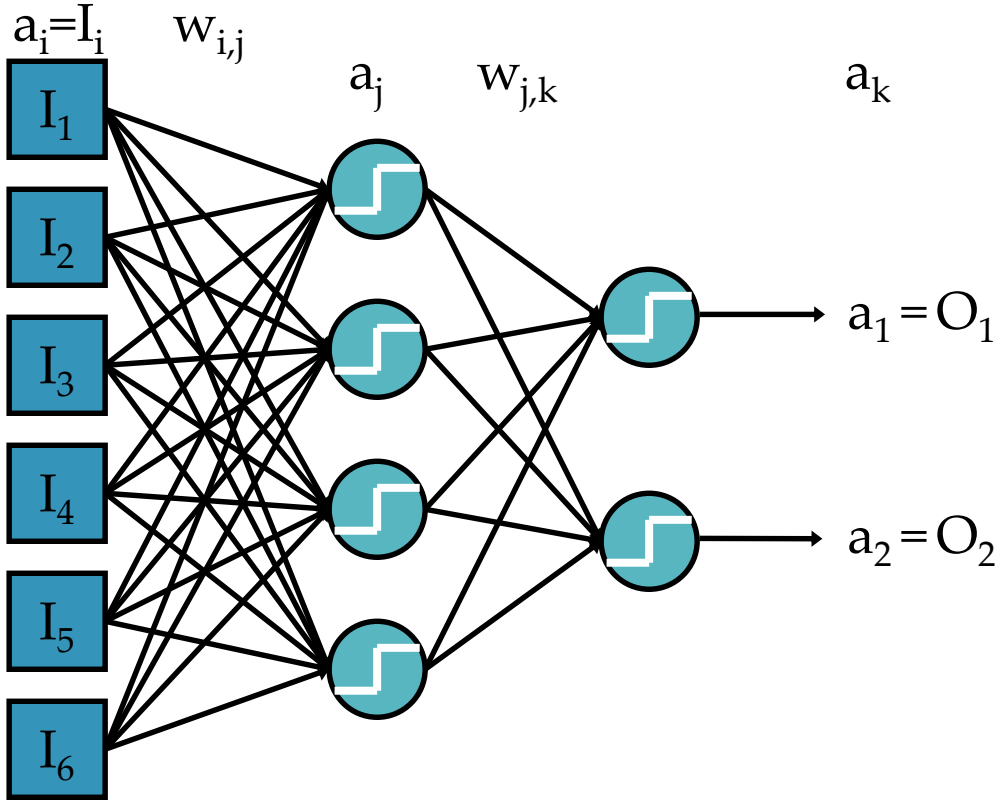
Two-Layer, Feed-Forward Neural Network

Input Units
Hidden Units
Output Units

Weights on links
from input to hidden

Weights on links
from hidden to output

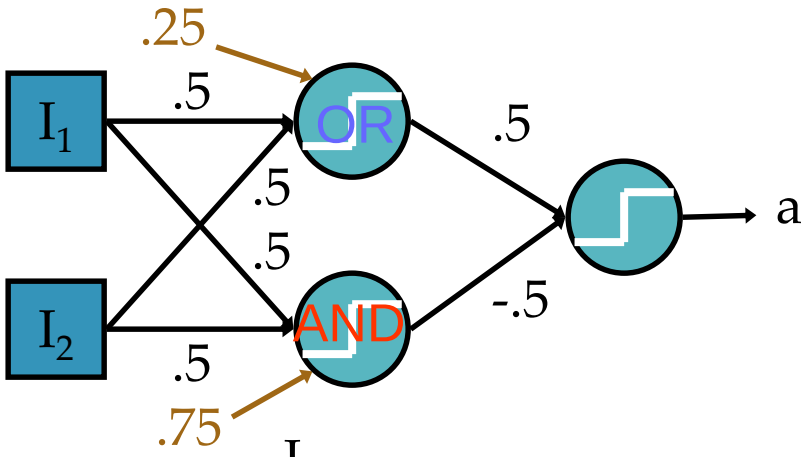
Network Activations



XOR Example

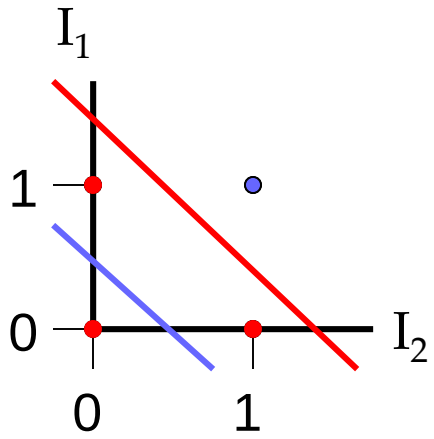
XOR 2-Layer Feed-Forward Network

- 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



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 $\Delta W_{p,q} = \Delta W_{p,q} + \Delta w_{p,q}$

  }

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

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

}

backward pass

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 $\boldsymbol{W}$ has an associated total error, $\boldsymbol{E}$, on ***all*** the training data
  - $\boldsymbol{E}$ is the sum squared error (SSE) defined by

$$E = E_1 + E_2 + \ldots + E_m$$

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

- Given n output units in the network:

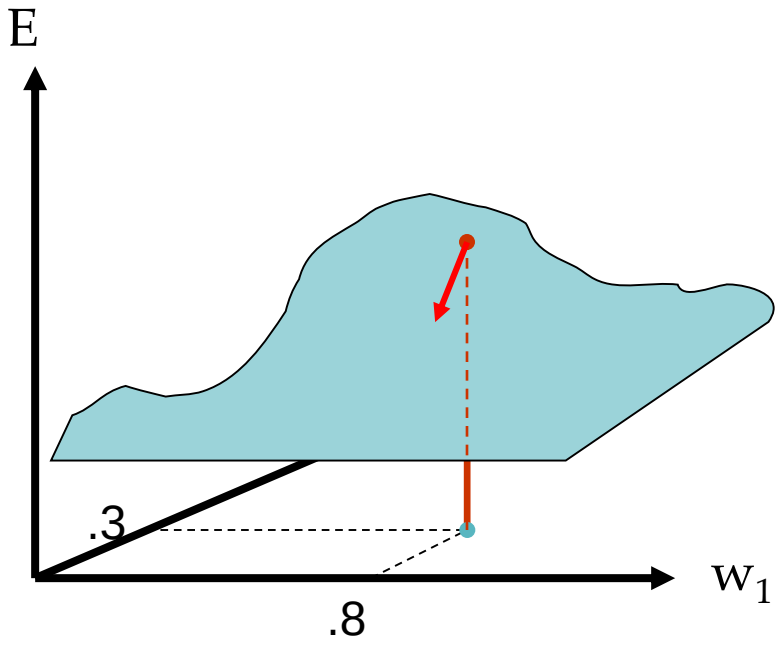
$$E_i = (T_1 - O_1)^2 + (T_2 - O_2)^2 + \ldots + (T_n - O_n)^2$$

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

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

Update the *i*th weight using

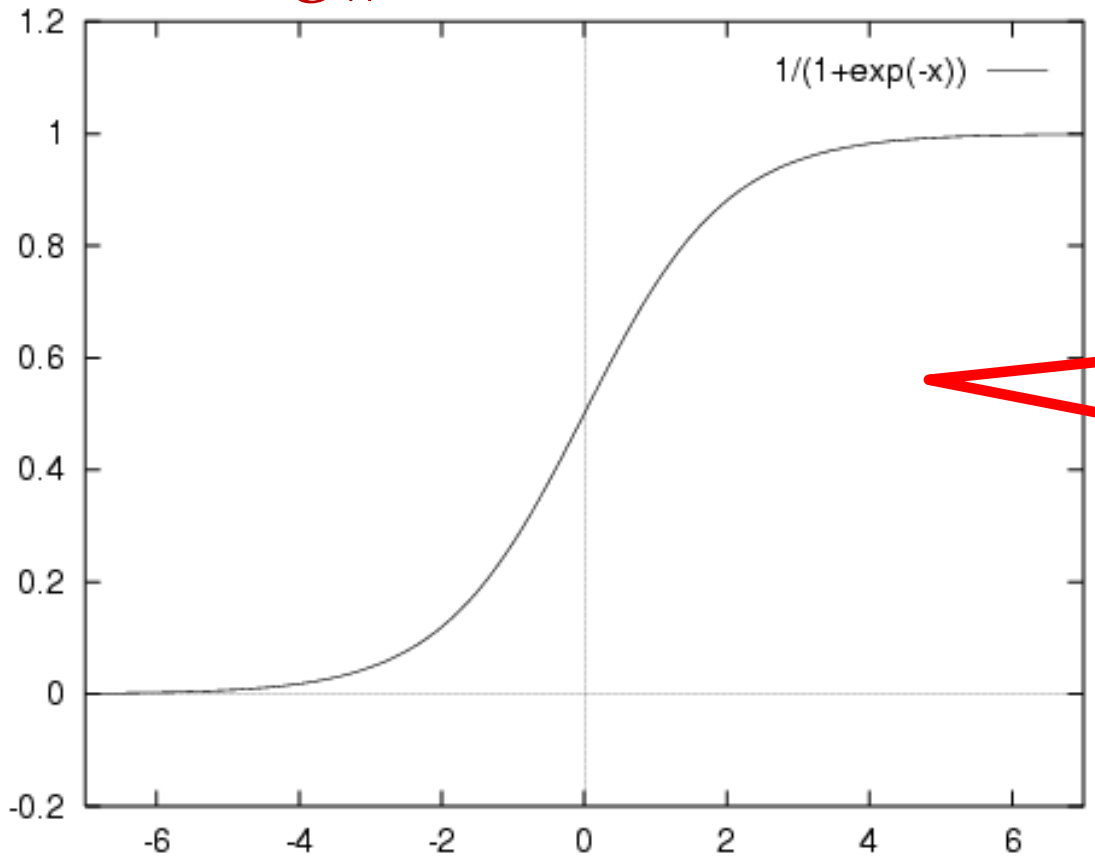
$$\Delta w_i = - \alpha \, \partial E / \partial w_i$$

However, what is the derivative of the LTU function?

The derivative is 0 everywhere up until 0, then undefined, then 0 everywhere until positive infinity…not useful for multiplying by alpha.

Sigmoid Activation Function

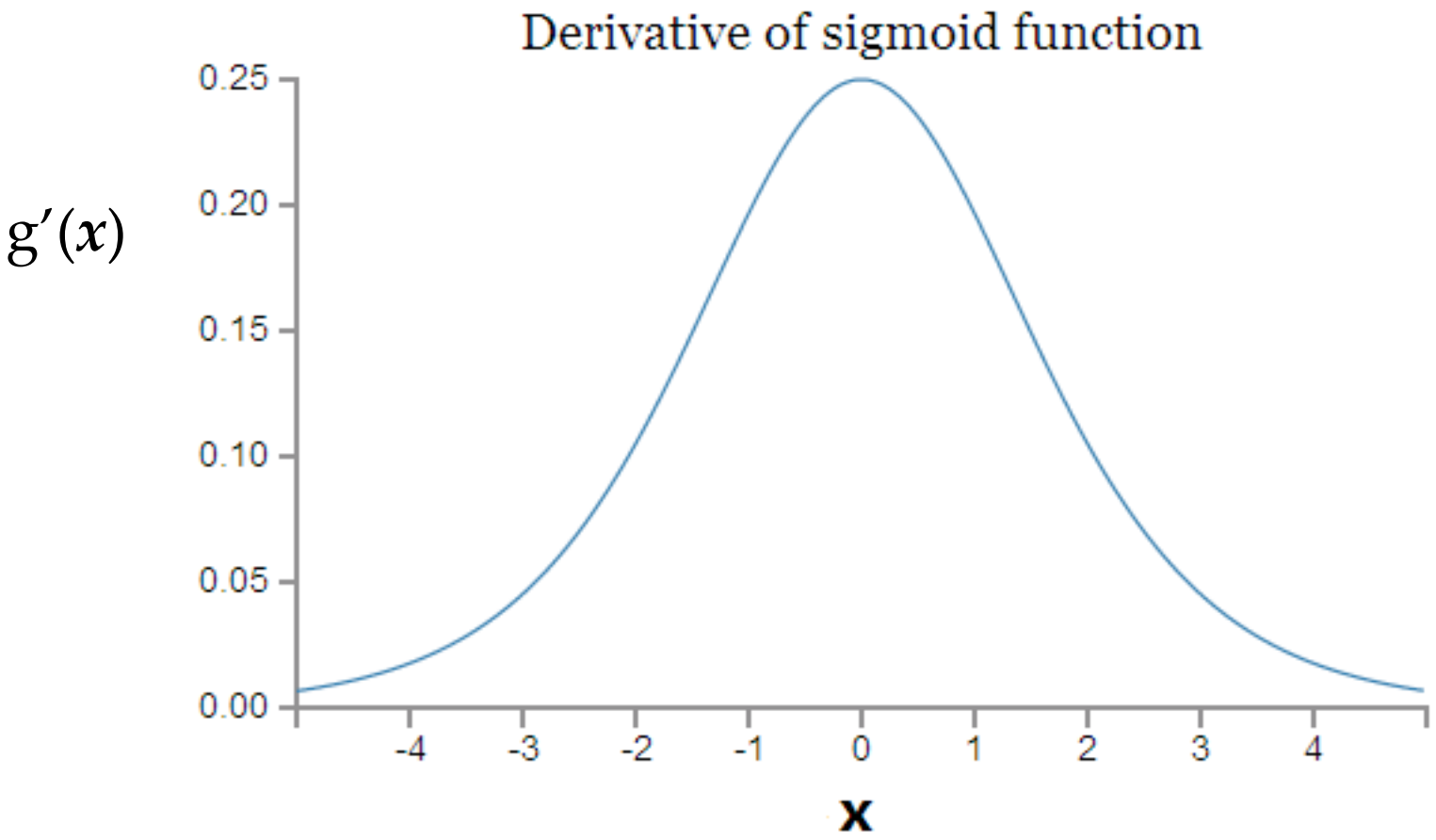
Solution: Replace with a smooth function such as Sigmoid function (aka Logistic Sigmoid function): $g_w(x) = 1 / (1 + e^{-wx})$



Squashes numbers to range [0,1]

First Derivative of Sigmoid Function

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

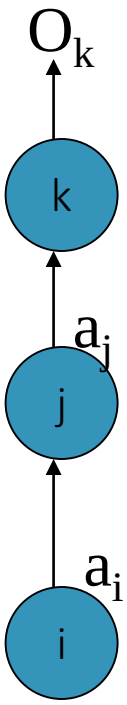


Updating the Weights

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.



output unit

hidden unit

input unit

Updating Weights in a 2-Layer Neural Network

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

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

$\Delta_k = \mathrm{Err}_k \times g'(\mathrm{in}_k)$

- $w_{j,k}$ 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*

Updating Weights in a 2-Layer Neural Network

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

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

{ O = neural_net_output(network, *e*)

forward pass

T = desired output, i.e., **T**arget or **T**eacher'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 $\Delta w_{i,j} = \alpha\, a_i\, \Delta_j = \alpha\, a_i\, g'(in_j) \sum_k w_{j,k} (T_k - O_k) g'(in_k)$

backward pass

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

}

Simplistic SGD: update all weights after each example

Neural Network (and AI) Applications

Because I know some of you skipped because it's almost Spring Break

Application: Handwriting Recognition

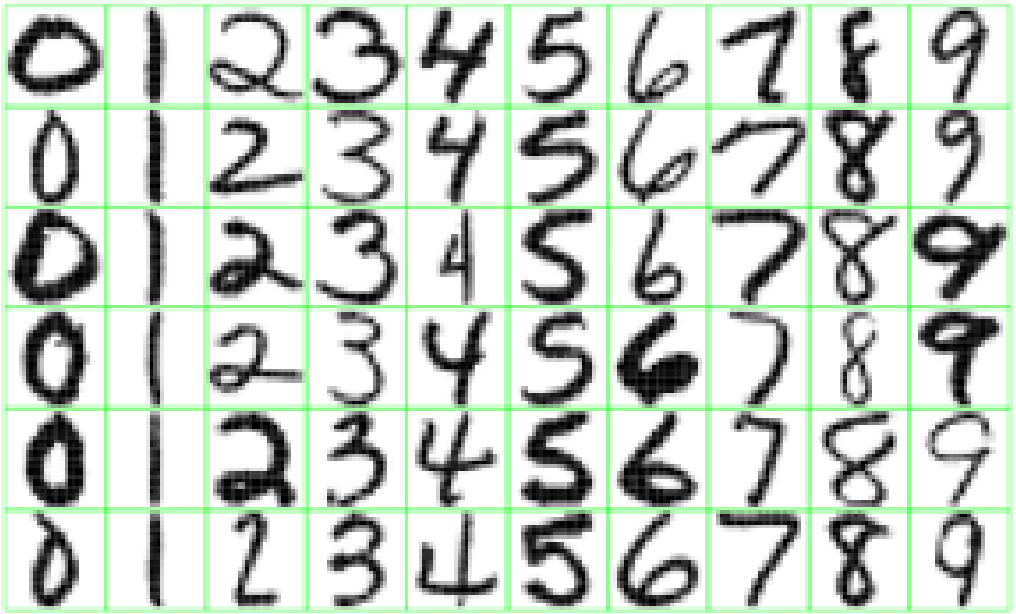


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

. For automatic interpretation of checks, postal addresses, and signatures.

”**The long term goal is to build an intelligent Angry Birds playing agent that can play new levels better than the best human players.** This is a very difficult problem as it requires agents to predict the outcome of physical actions without having complete knowledge of the world, and then to select a good action out of infinitely many possible actions.” - AIBirds.org