COMS W4701: Artificial Intelligence

Lecture 10b: Neural Networks

Tony Dear, Ph.D.

Department of Computer Science

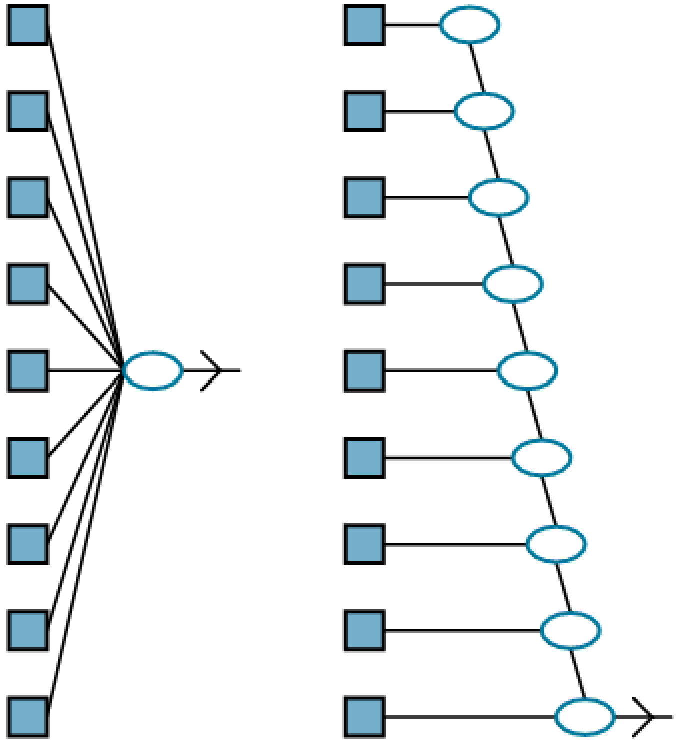
School of Engineering and Applied Sciences

Today

- Motivation for neural networks
- Neural network structure
- Deep learning considerations

“Shallow” Learning

- Linear models, and many simple nonlinear models, are relatively “shallow”
- While they can process high-dimensional inputs, they may not sufficiently represent *interactions* between inputs

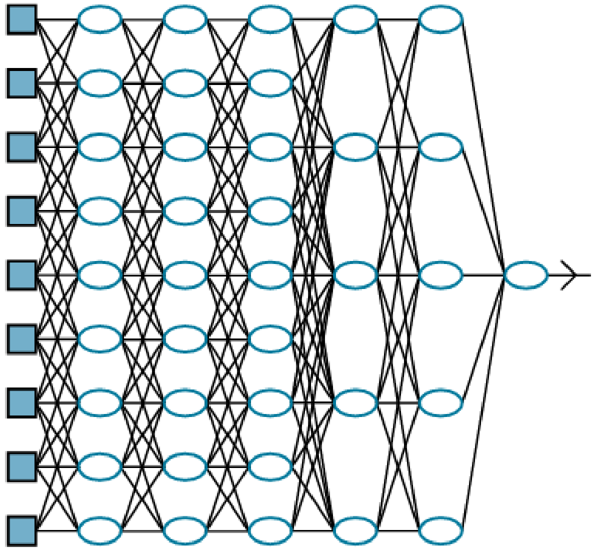


- Pros: Such models are easier to learn and interpret
- Learning is relatively efficient, not as data-hungry

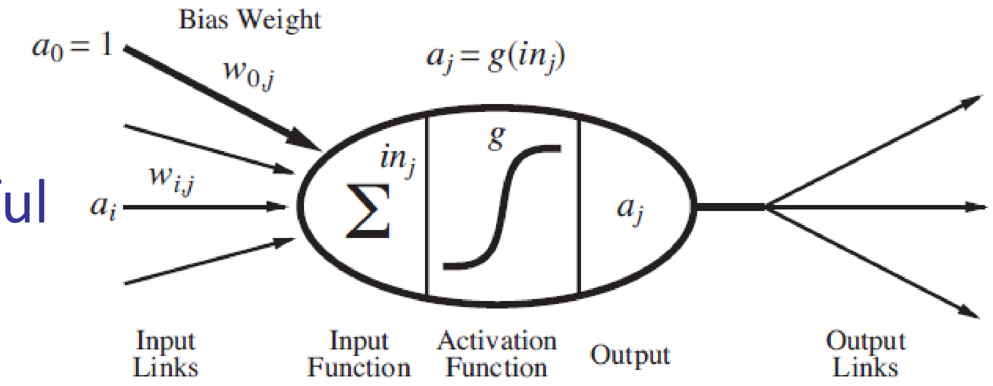
- Cons: Resultant hypothesis space may be much smaller than the space of complex real-world functions
- May have to perform manual *feature engineering*

Deep Learning

- Idea: Use *compositions* of nonlinear functions to facilitate feature interactions and learn richer models



- **Neural networks** or **multilayer perceptrons** (MLPs) consist of successive *layers* of neurons (units)
- Each neuron applies a nonlinear *activation function* to a linear combination of input values



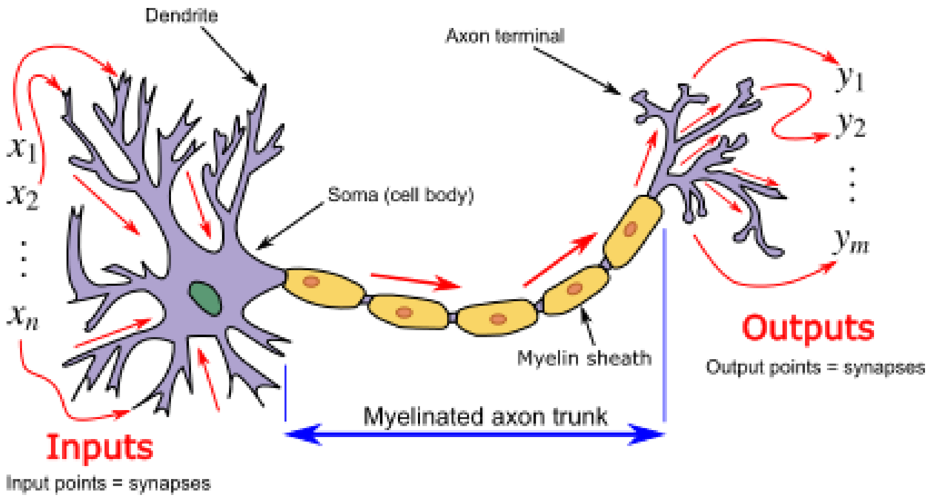
- Typically have input, hidden, and output layers
- We can think of each layer as a different but useful *representation* of the inputs

Neural Networks

- There is some (very loose) inspiration for neural networks from biology
- Brain neurons produce output signals in response to input signals

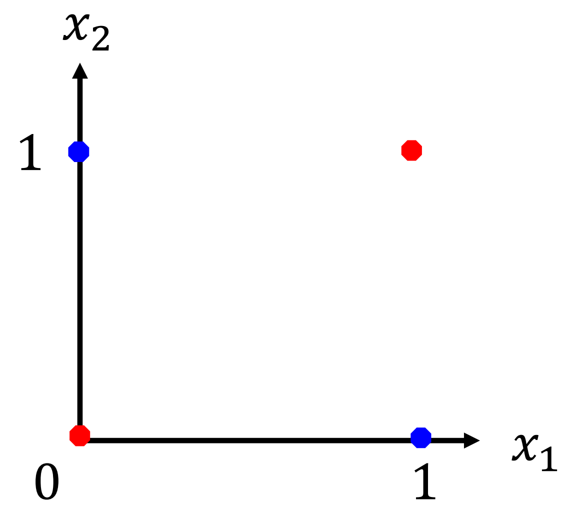
- Modern neural networks no longer models biology, although their applications do include biological and physical sciences

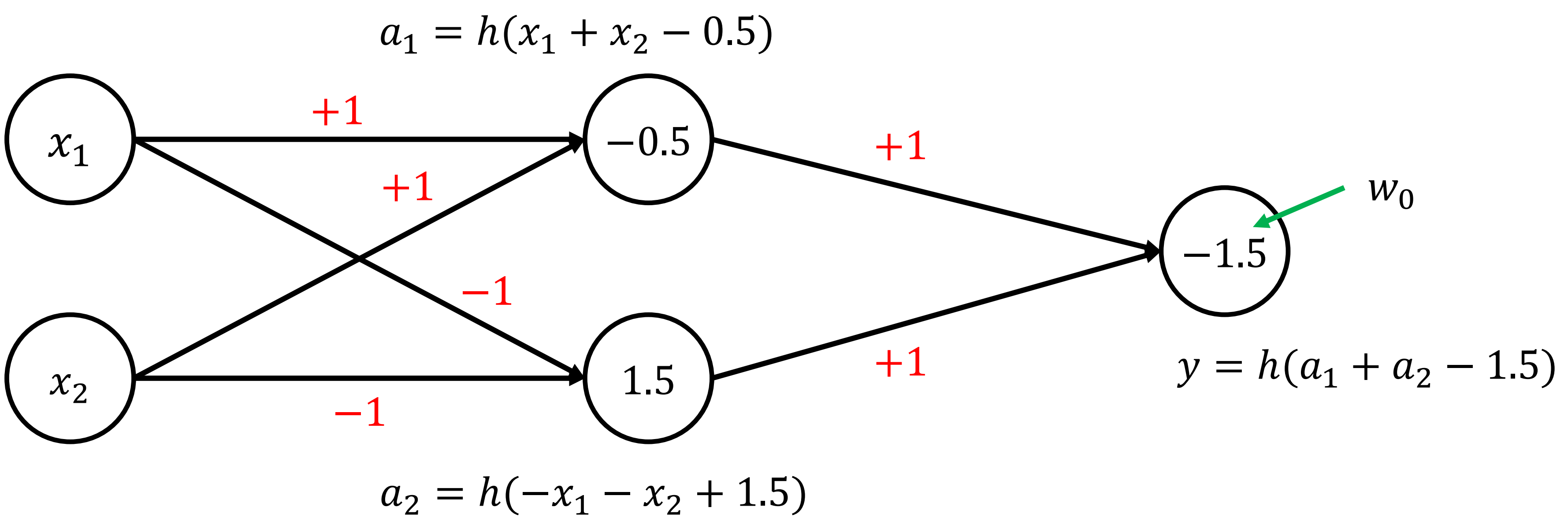
- Neural networks excel at **representation learning**
- Automated feature design in unstructured and perception tasks that difficult to manually describe, but have lots of associated training data



Example: XOR Function

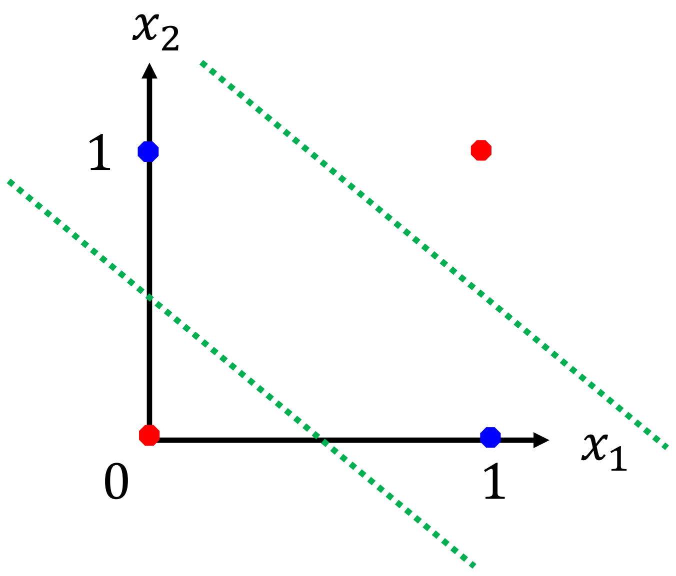
- No linear separator exists for these 4 points
- But suppose we compose three hard threshold linear classifiers together
- $h(z) = 1$ if $z \geq 0$, $h(z) = 0$ otherwise



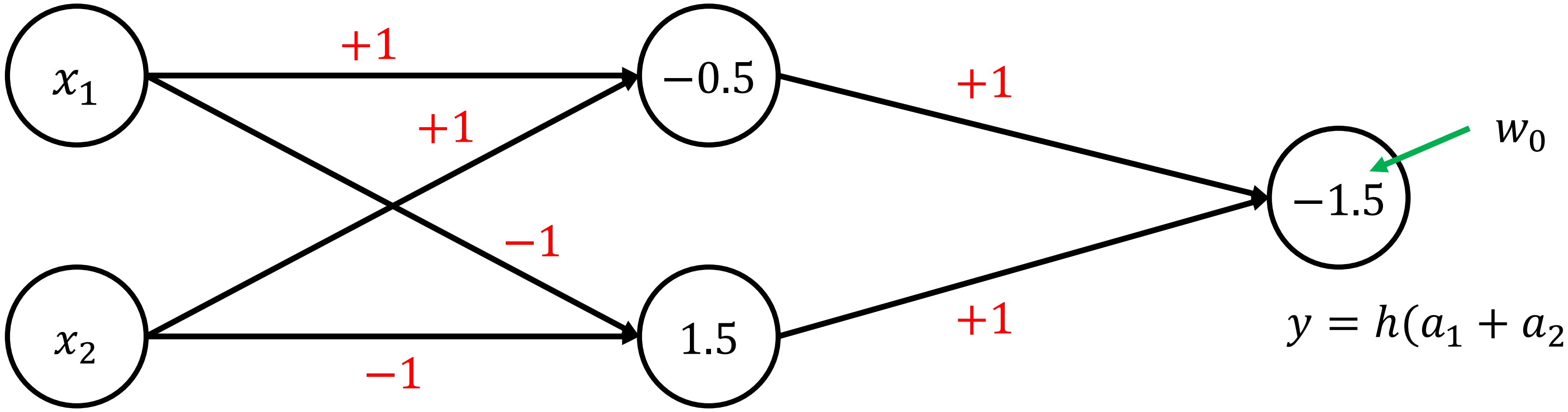


Example: XOR Function

- $\mathbf{x} = (0,0) \rightarrow a_1 = 0, a_2 = 1 \rightarrow y = 0$
- $\mathbf{x} = (1,1) \rightarrow a_1 = 1, a_2 = 0 \rightarrow y = 0$
- $\mathbf{x} = (0,1) \rightarrow a_1 = 1, a_2 = 1 \rightarrow y = 1$
- $\mathbf{x} = (1,0) \rightarrow a_1 = 1, a_2 = 1 \rightarrow y = 1$



$a_1 = h(x_1 + x_2 - 0.5)$

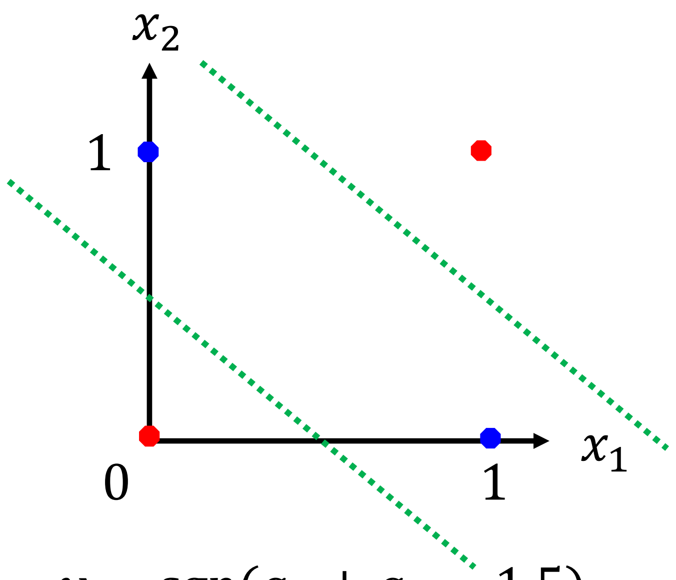


$y = h(a_1 + a_2 - 1.5)$

$a_2 = h(-x_1 - x_2 + 1.5)$

Example: XOR Function

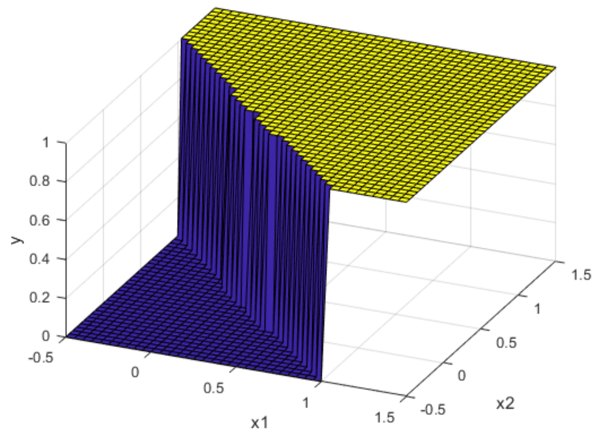
- $\mathbf{x} = (0,0) \rightarrow a_1 = 0, a_2 = 1 \rightarrow y = 0$
- $\mathbf{x} = (1,1) \rightarrow a_1 = 1, a_2 = 0 \rightarrow y = 0$
- $\mathbf{x} = (0,1) \rightarrow a_1 = 1, a_2 = 1 \rightarrow y = 1$
- $\mathbf{x} = (1,0) \rightarrow a_1 = 1, a_2 = 1 \rightarrow y = 1$

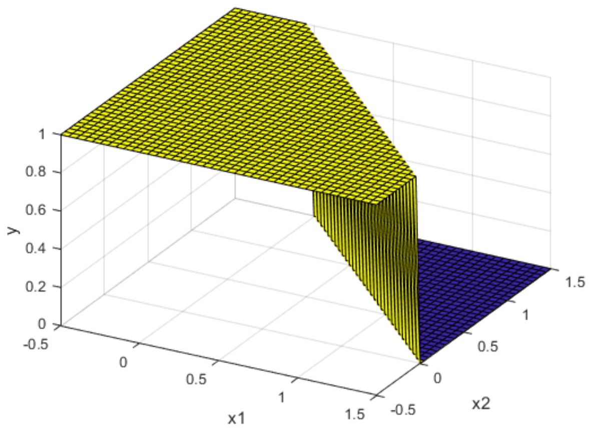


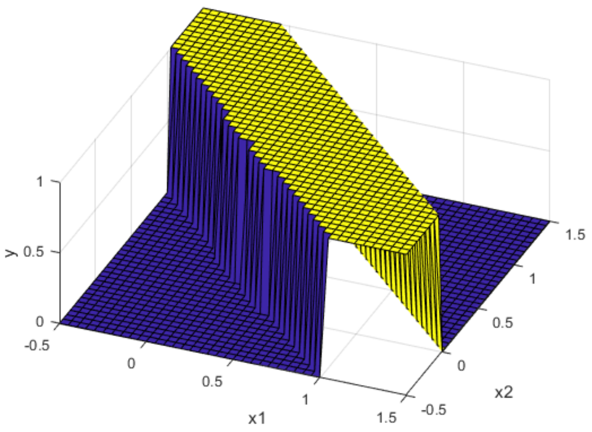
$a_1 = \text{sgn}(x_1 + x_2 - 0.5)$

$a_2 = \text{sgn}(-x_1 - x_2 + 1.5)$

$y = \text{sgn}(a_1 + a_2 - 1.5)$







Input Layer

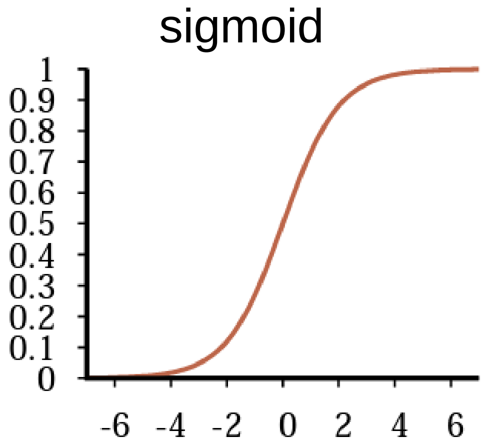
- We will generally have an input unit for each input data attribute

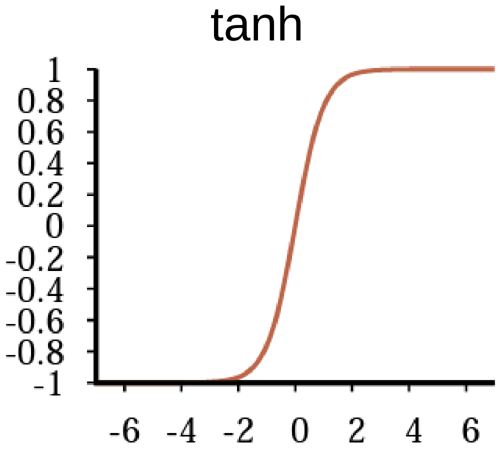
- For categorical attributes, create an input node for each possible value
- **One-hot encoding**: Set the node value corresponding to the input to 1 and other node values to 0

- Same idea for high-dimensional input data, like the pixels of an image!
- For other models, pixels may be a poor input representation, but *deep* networks are able to make sense of these

Hidden Layers

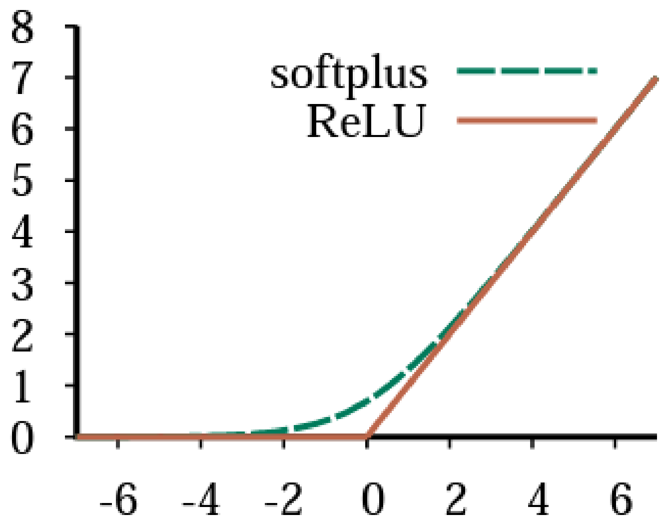
- Hidden layer units implement *nonlinear* functions of weighted inputs, e.g., sigmoid or tanh
- $\sigma(x) = \frac{1}{1+e^{-x}}$, $\tanh(x) = 2\sigma(2x) - 1$
- Pros: Continuous and differentiable everywhere (unlike hard threshold), bounded output values
- Disadvantages: Saturated values and small derivatives for inputs away from zero, making learning slow





Rectified Linear Units

- Most modern networks use the **ReLU** activation function
- $g(x) = \max\{0, x\}$ outputs 0 if $x < 0$ and x otherwise



- Piecewise linear, low computational cost
- Nonvanishing derivative when active ($x > 0$)

- Variations modify the function to make derivative nonzero for $x < 0$ as well
- E.g., Softplus (smoothed ReLU): $g(x) = \log(1 + e^x)$

Output Layer

- The output layer of a network produces values in the expected format
- E.g., for regression tasks it can contain linear node units

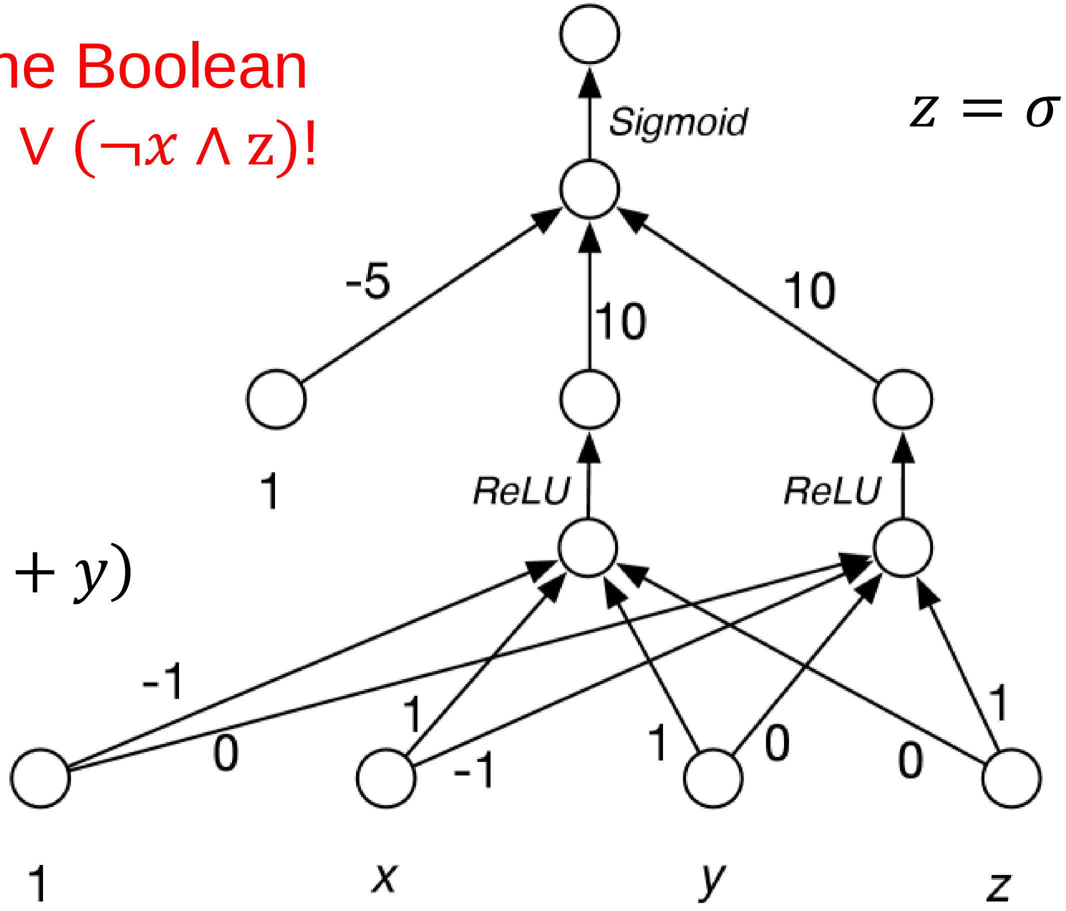
- For binary classification tasks, sigmoid units like logistic regression
- Or softmax units for multi-valued classification, with one unit per class
- Very common for image classification and language prediction tasks

- Sigmoid and softmax output *probabilities*, which give us likelihood functions that we can then optimize

Example

This implements the Boolean expression $(x \wedge y) \vee (\neg x \wedge z)$!

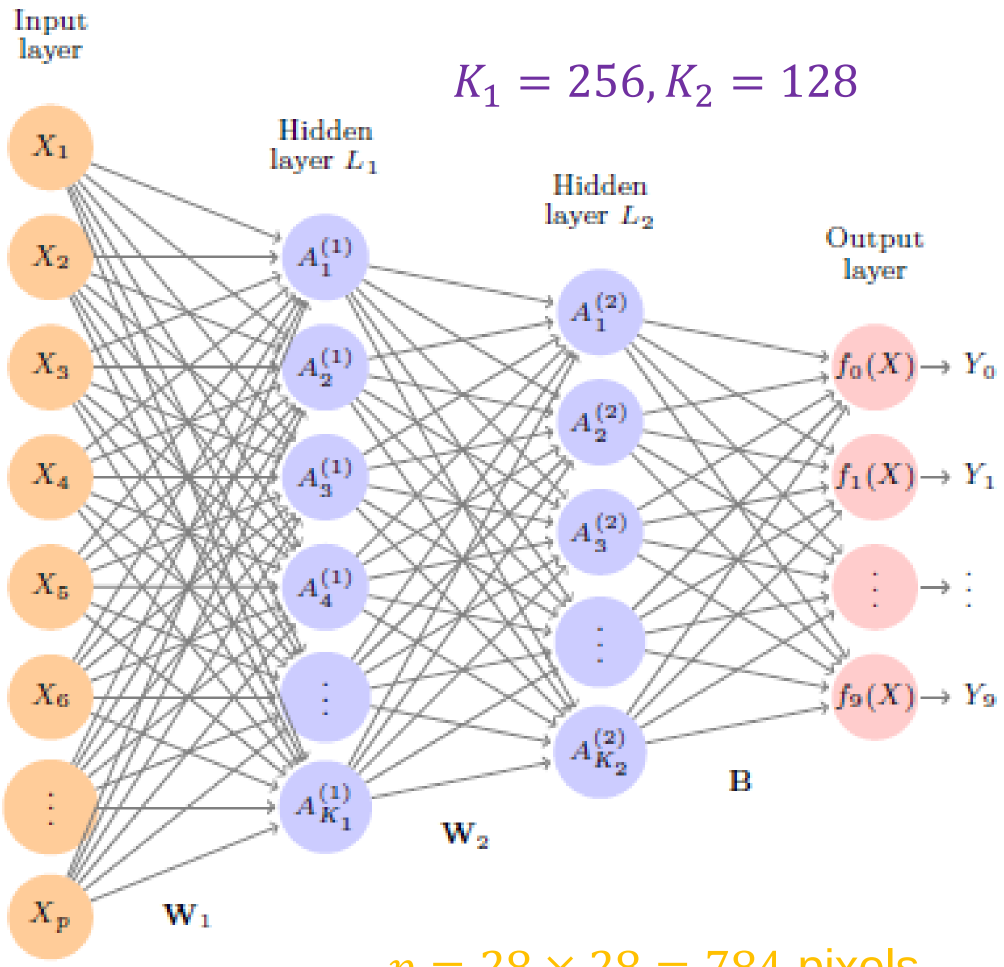
$z = \sigma(-5 + 10h_1 + 10h_2)$



$h_1 = ReLU(-1 + x + y)$

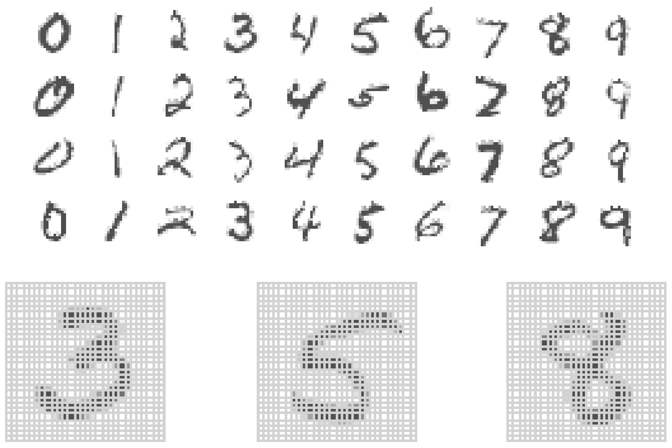
$h_2 = ReLU(-x + z)$

Example: MNIST



$K_1 = 256, K_2 = 128$

$p = 28 \times 28 = 784$ pixels



Softmax:

$$f_i(X) = \Pr(Y = i|X) = \frac{e^{X_i}}{e^{X_0} + \cdots + e^{X_9}}$$

James et al., *Introduction to Statistical Learning.*

Architecture Design

- **Universal approximation theorem**: A network with at least one hidden layer and sufficiently many neurons can approximate any function

- Still no guarantee that a network can *learn* any function
- Required network size may still be intractably large

- A shallow network may require exponentially more neurons to represent the same function(s) expressed by a deeper network
- Deep networks perform multiple compositions and transformation steps

Deep Learning vs Other Models

- Deep neural networks have seen unprecedented success in a variety of ML tasks: image classification, language translation, generative AI

- However, on simpler tasks it may be desirable to use tried and true methods that are easier to learn and interpret

- Disadvantages of deep learning: Fewer guidelines for model selection; many, *many* parameters; requires *lots* of data to learn; less interpretable
- Can be a good approach if have lots of data and time to begin with

Loss Functions

- A **loss function** $L(y, \hat{y})$ measures the closeness between a predicted value $\hat{y}$ and the actual value y
  - Absolute ($\boldsymbol{L_1}$) loss: $L(y, \hat{y}) = |y - \hat{y}|$
  - Squared ($\boldsymbol{L_2}$) loss: $L(y, \hat{y}) = (y - \hat{y})^2$
  - **0-1** loss: $L(y, \hat{y}) = 1$ if $y \neq \hat{y}$, and 0 otherwise

- The *empirical loss* can be computed over a data set can be computed by taking the sum or the mean of one of the functions above
- L_1 and L_2 losses are useful for real values, 0-1 loss for categorical values

Training Neural Networks

- The parameters of a neural network are the weights along the connections
- As before, we can train a network using **stochastic gradient descent**

- Divide data into minibatches, process them for some number of epochs
- Forward-feed a minibatch through the network and compute the *loss*

- Gradient wrt all weights is (slightly) complicated due to network structure
- **Backpropagation** is an efficient algorithm for computing all derivatives
- Then update all weights according to gradient descent and repeat

Summary

- Neural networks *compose* nonlinear transformations of inputs
- Great for representation learning, automated feature design
- Not so great for interpretability, data and training efficiency

- Many design choices in number of layers/neurons, activation functions
- Common choices: ReLU for hidden, sigmoid or softmax for output

- Neural networks can be trained using stochastic gradient descent
- Compute losses and gradient on training data, update weight parameters