Neural Networks

CMPUT 366: Intelligent Systems

GBC §6.0-6.4.1

Lecture Outline

1. Recap
2. Nonlinear models
3. Feedforward neural networks

Recap: Calculus

- Derivatives can be used for **optimization**
  - **Minimization:** Increase x if derivative is **negative** & vice versa
- **Partial derivatives** are derivatives of "frozen" function:

$$\frac{\partial}{\partial x} f(x, y) = \frac{d}{dx}(f)_{y=y}(x)$$

- **Gradient** of a function is a **vector** of all its partial derivatives:

$$(\nabla f)(x, y) = \begin{bmatrix} \frac{\partial}{\partial x} f(x, y) \\ \frac{\partial}{\partial y} f(x, y) \end{bmatrix}$$

Linear Models

- Supervised models we have considered so far have been **linear:**

weights

$$y = f(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^T\mathbf{x}) = g\left(\sum_{i=1}^{n} w_i x_i\right)$$

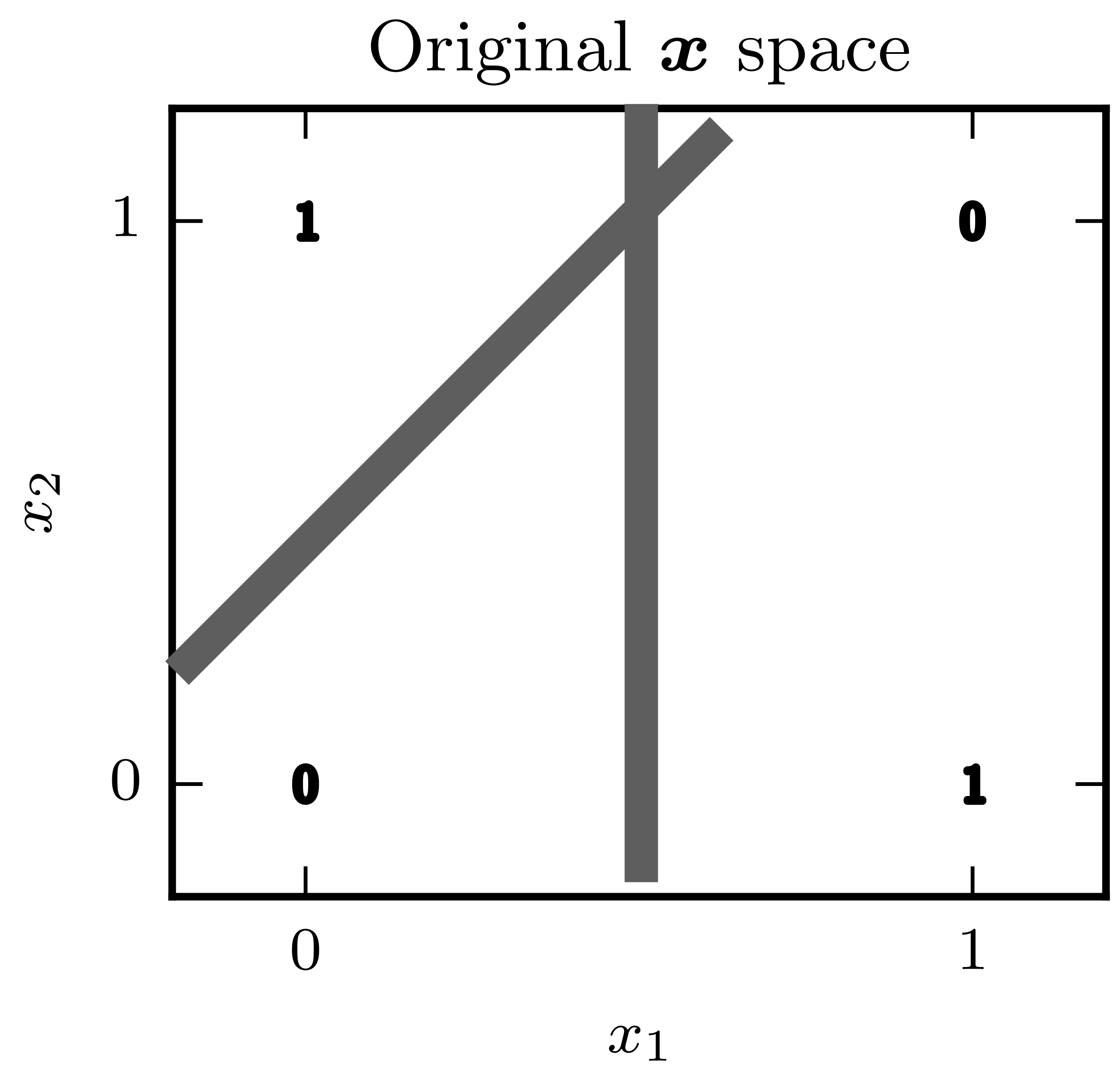
Linear model inputs

activation function

  - Linear classification / regression
  - Logistic regression
- **Advantages: Efficient** to fit (closed form sometimes!)
- **Disadvantages:** Can be really **limited**

Example: XOR

- The function $f(x_1, x_2) = (x_1 \text{ XOR } x_2)$ is not **linearly separable**
  - There is no way to draw a **straight line** with all of the 1's on one side and all of the 0's on the other
  - This means that no **linear model** can represent XOR exactly; there will always be some errors
- **Question:** What else could we do?



(Image: Goodfellow 2017)

Nonlinear Features

$$y = f(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^T\mathbf{x}) = g\left(\sum_{i=1}^{n} w_i x_i\right)$$

One option: Learn a linear model on **richer inputs**

1. Define a **feature mapping** $\phi(\mathbf{x})$ that returns **functions** of the original inputs

2. Learn a linear model of the **features** instead of the **inputs**

$$y = f(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^T\phi(\mathbf{x})) = g\left(\sum_{i=1}^{n} w_i [\phi(\mathbf{x})]_i\right)$$

Nonlinear Features for XOR

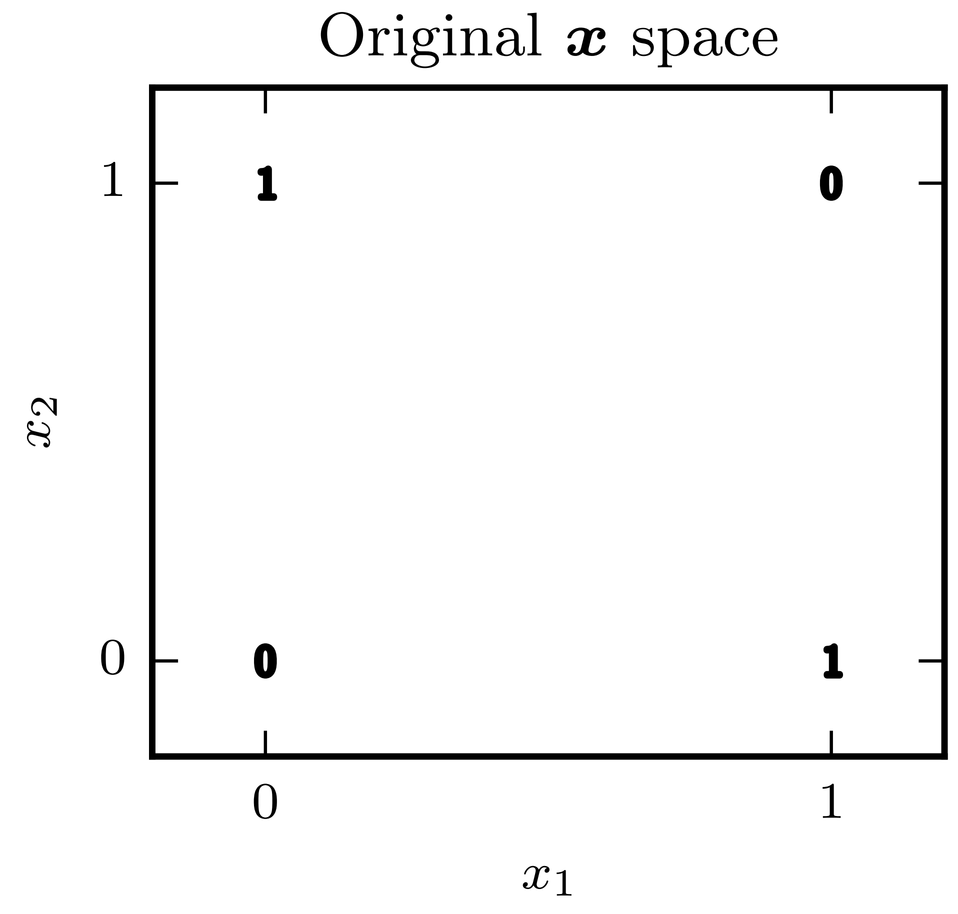
- **Question:**
  What additional features would help?

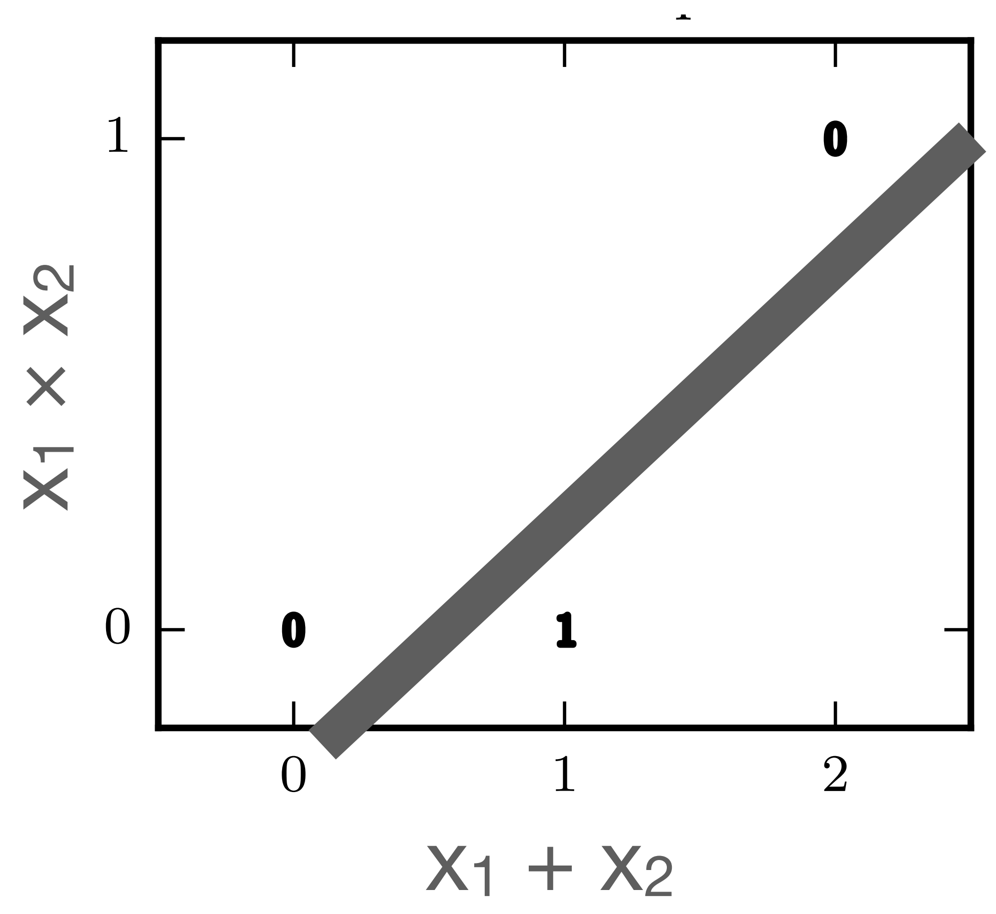
- The product of x_1 and x_2!

  - $\phi(x_1, x_2) = [1, x_1, x_2, x_1 x_2]$

  - $\mathbf{w} = [-0.2, 0.5, 0.5, -2]$

- $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T \phi(\mathbf{x}) > 0$ for $(0,1)$ and $(1,0)$
  $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T \phi(\mathbf{x}) < 0$ for $(1,1)$ and $(0,0)$





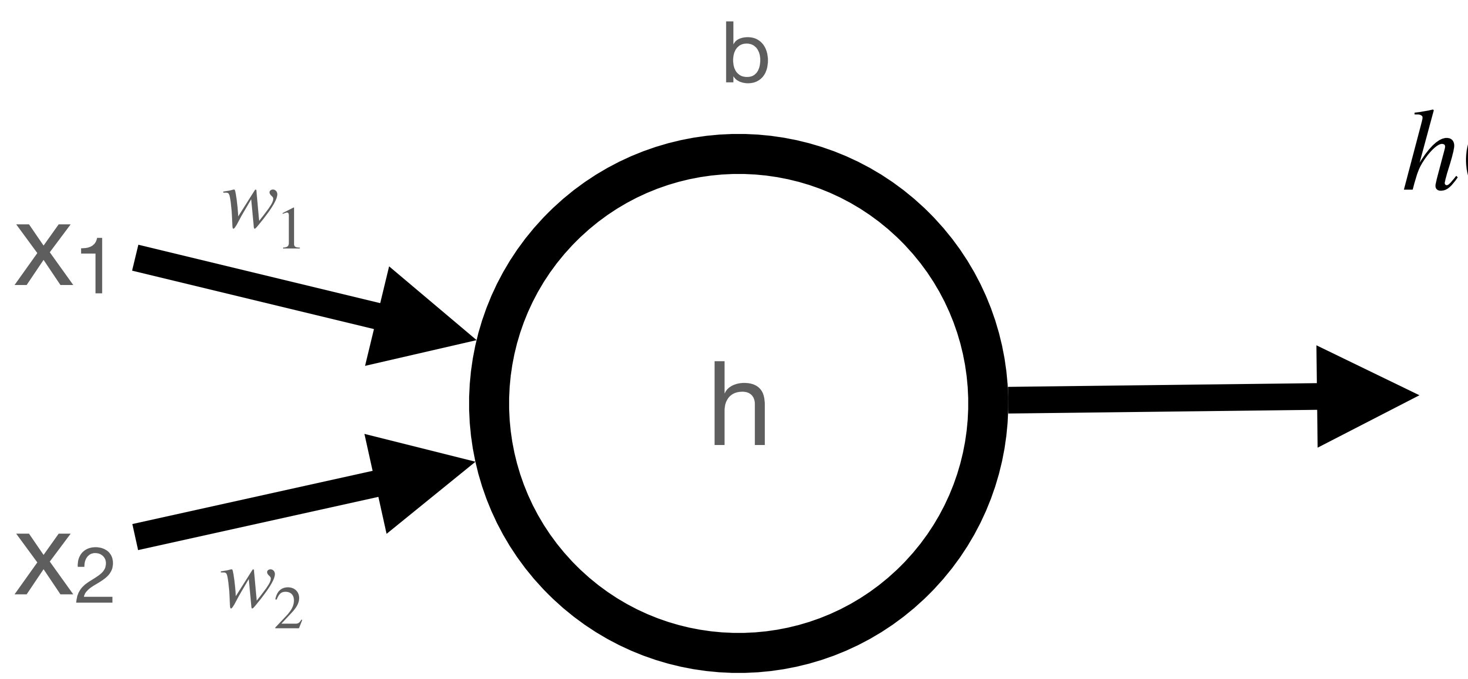
(Image: Goodfellow 2017)

Learning Nonlinear Features

- Manually constructing good features is extremely **hard**
- Manually constructed features are not **transferrable** between domains
    - e.g., SIFT features were a revolution in computer vision, but are **only** for computer vision
- Deep learning aims to learn ϕ **automatically** from the data

Neural Units

- Deep learning learns ϕ by **composing** little functions
- These function are called **units**



$$h(\mathbf{x}; \mathbf{w}, b) = g(b + \mathbf{w}^T\mathbf{x}) = g\left(b + \sum_{i=1}^{n} w_i \mathbf{x}_i\right)$$

weights

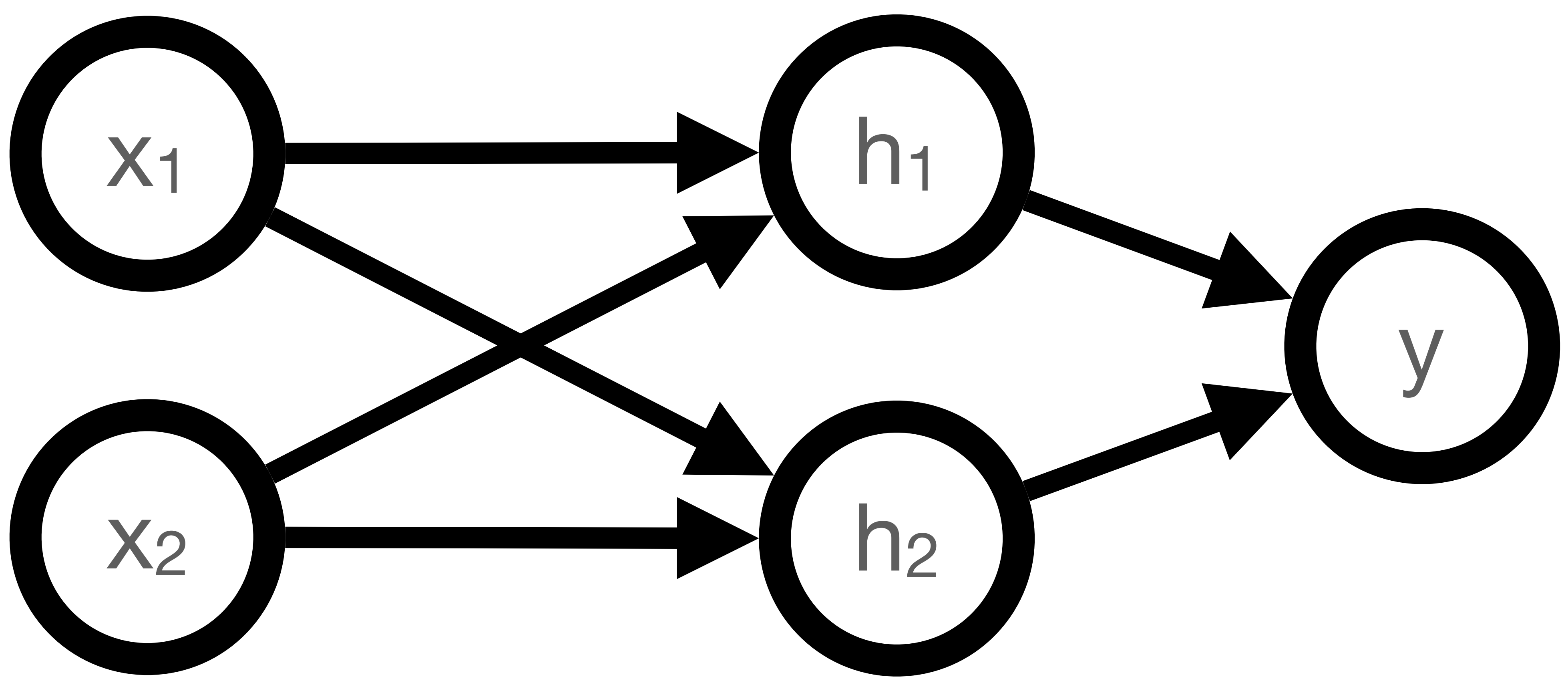
offset

activation function

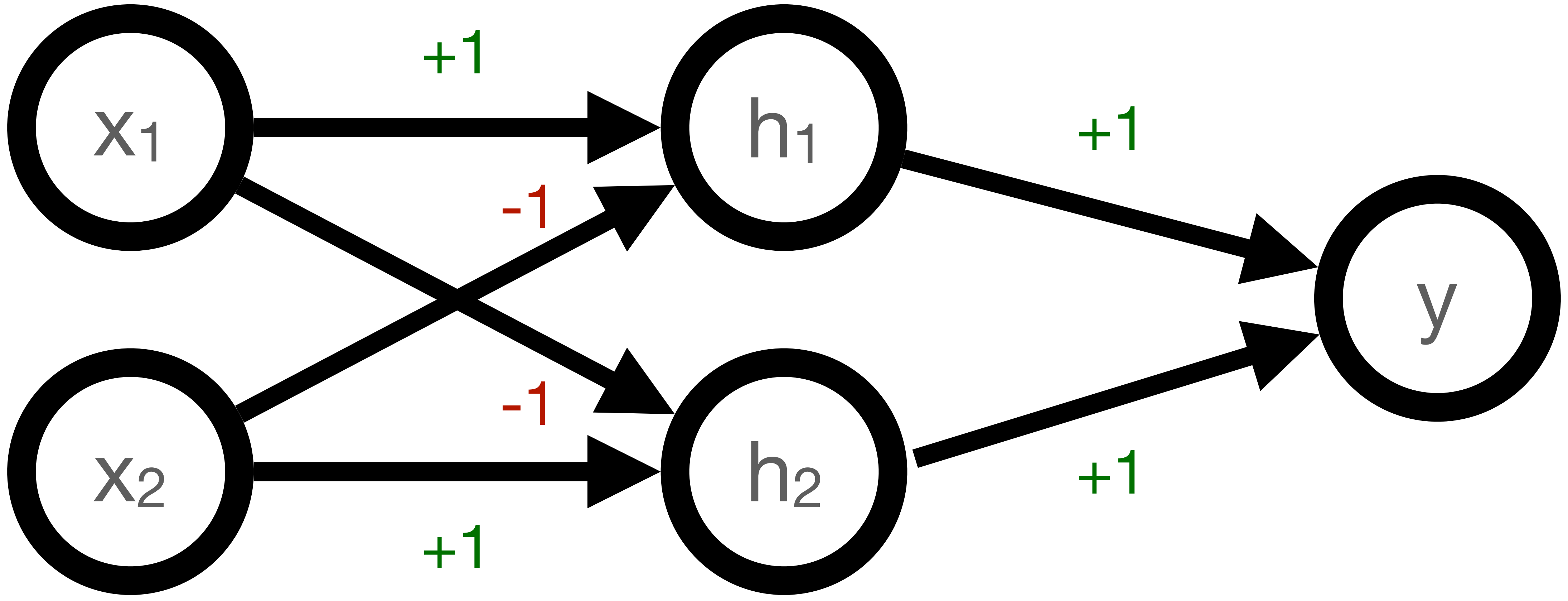
- **Question:** How is this different from a linear model?

Feedforward Neural Network

- A **neural network** is many units **composed** together
- **Feedforward neural network:** Units arranged into **layers**
  - Each layer takes outputs of **previous layer** as its **inputs**

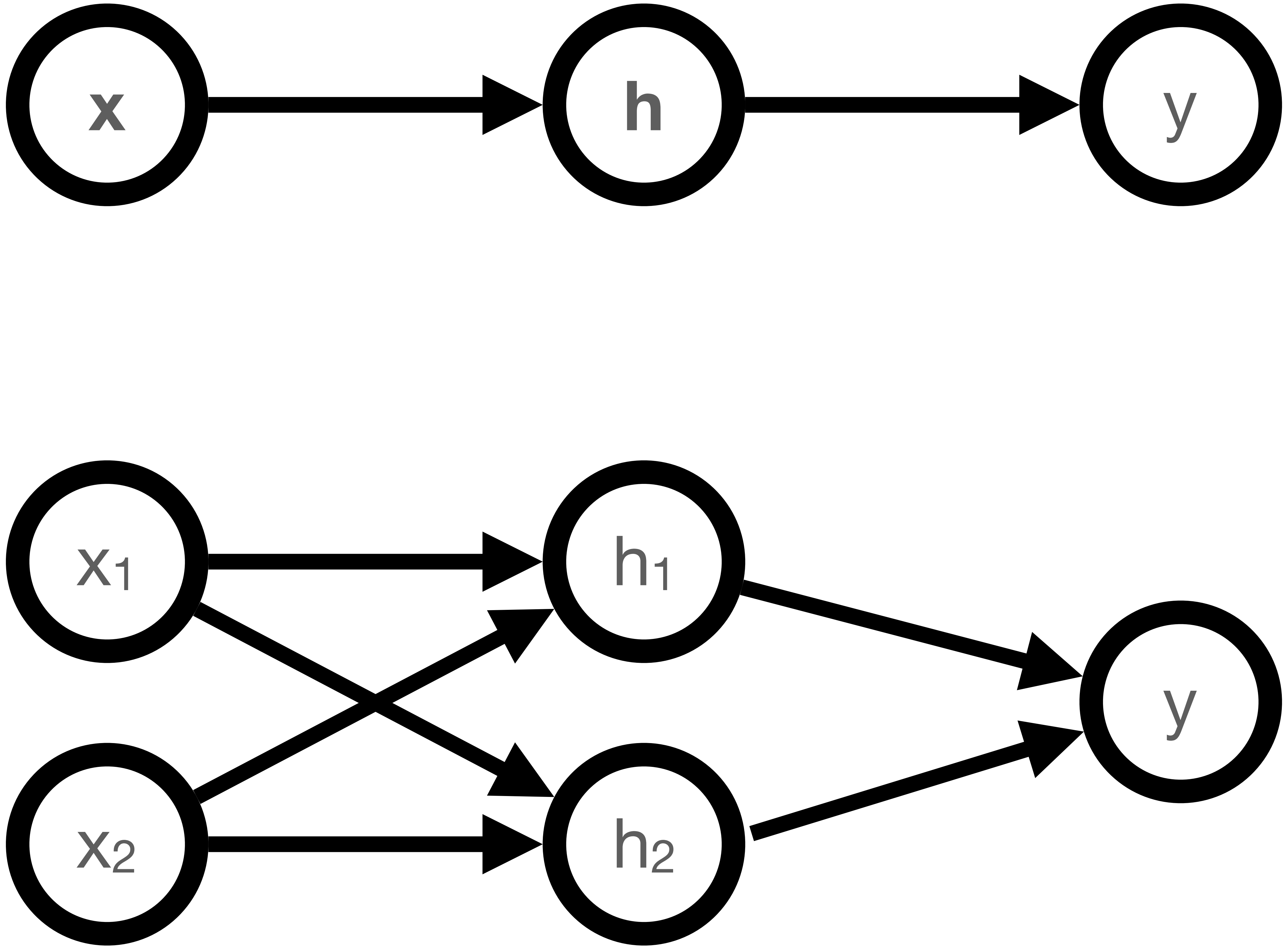


Example: XOR network



- *Activation*: $g(z) = \max\{0, z\}$ ("recified linear unit")
- Weights:
  - $[+1, -1]$ for h_1; $[-1, +1]$ for h_2
  - $[+1, +1]$ for y

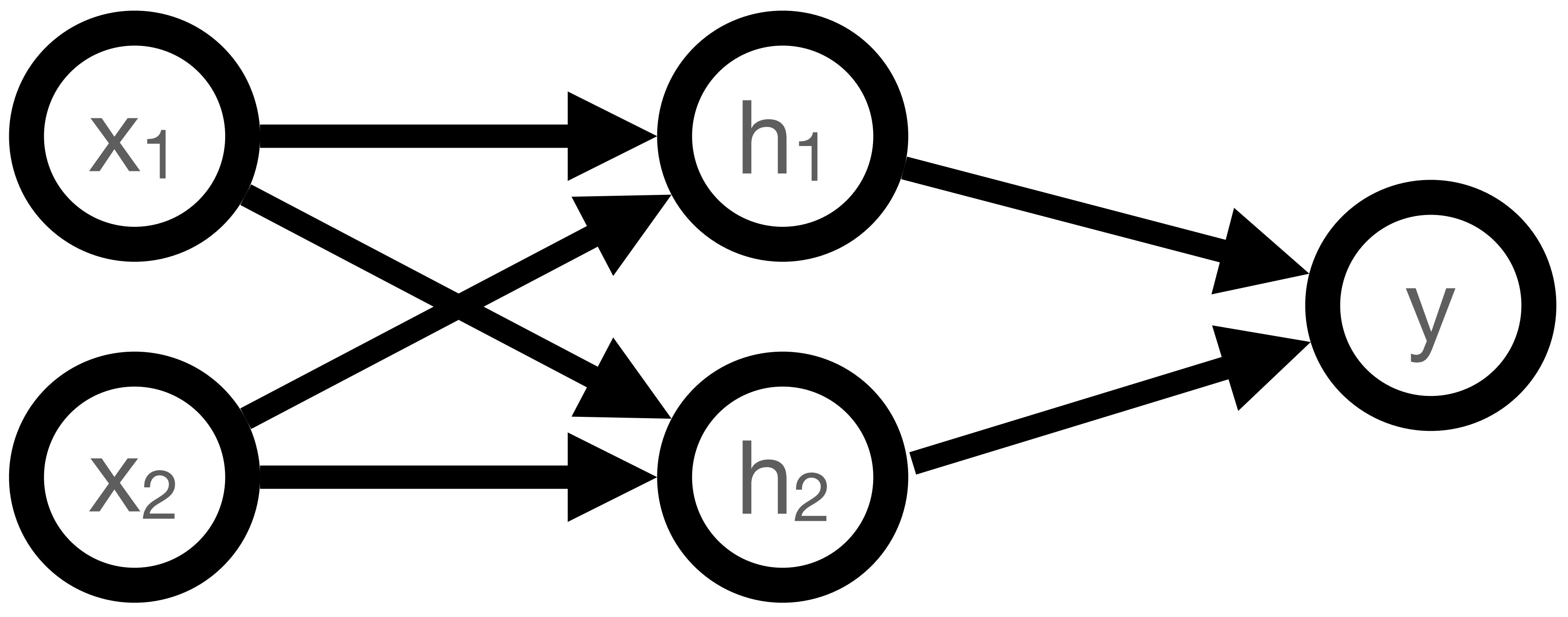
Matrix Representation



- You can think of the **outputs** of each layer as a **vector** $\mathbf{h}$
- The **weights** from all the outputs of a previous layer to each of the units of the layer can be collected into a **matrix** **W**
- The **offset term** for each unit can be collected into a vector $\mathbf{b}$:

$$\mathbf{h} = g\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$

Architecture



Design decisions:

1. **Depth:** number of layers
2. **Width:** number of nodes in each layer
3. Fully connected?

Universal Approximation Theorem

Theorem: (Hornik et al. 1989; Cybenko 1989; Leshno et al. 1993)
A feedforward network with **one hidden layer** with a "squashing" activation or rectified linear activation and a linear output layer can approximate **any function** to within **any given error bound**, given enough hidden units.

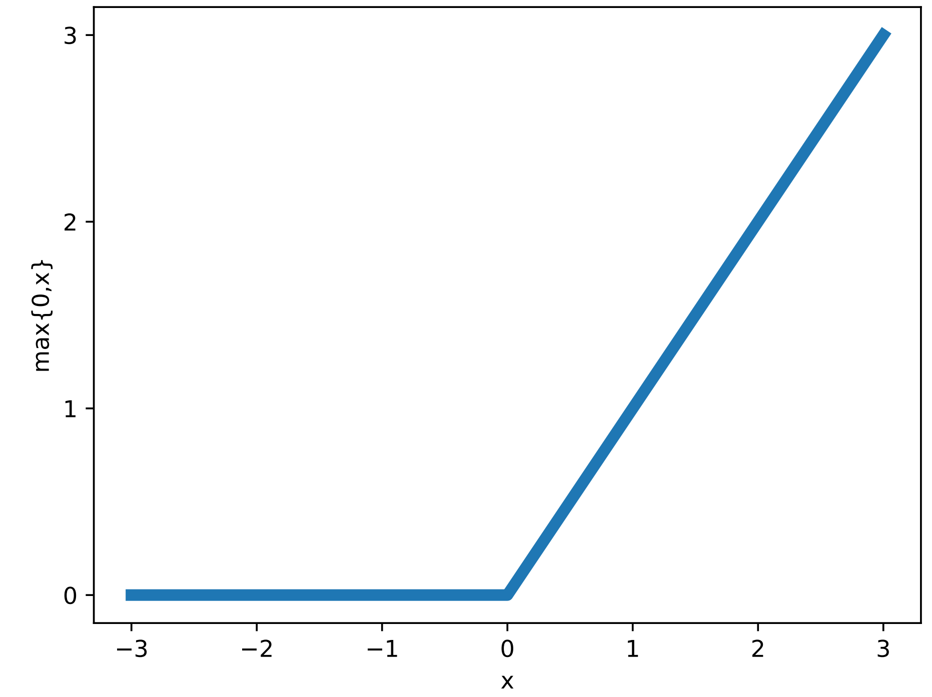
- So a **wide but shallow** feedforward network can **represent** any function we're trying to learn!
- **Question:** Why bother with multiple layers? (i.e., depth > 1)

Training

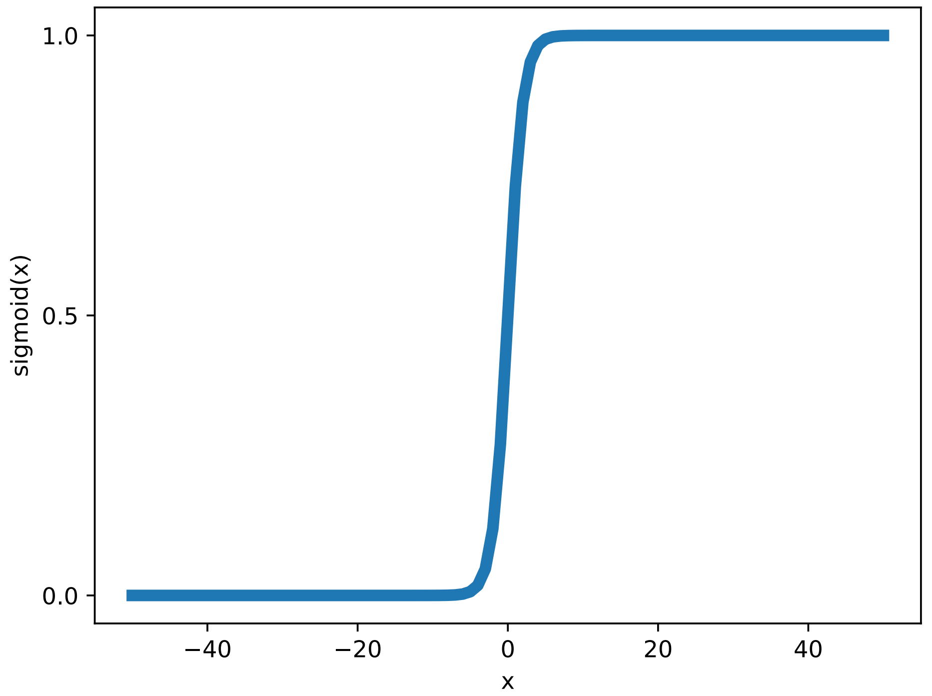
- Neural networks are trained using variants of **gradient descent**
  - e.g., stochastic gradient descent
- **Back propagation** is an algorithm that allows for efficient computation of the **gradient**
- Modern frameworks can compute the gradient in other ways (e.g., **automatic differentiation**) even for complicated units

Hidden Unit Activations

- Default choice: Rectified linear units (ReLU)
  $g(z) = \max\{0, z\}$



- Other common types:
  - $\tanh(z)$
  - $\dfrac{1}{1+e^{-z}}$ (sigmoid)



- Sigmoid suffers from **vanishing gradients**; ReLU does not

Summary

- Generalized linear models are **insufficiently expressive**
- Composing GLMs into a network is **arbitrarily expressive**
  - A neural network with a **single hidden layer** can approximate **any function**
  - But the network might need to be impractically large, prone to overfitting, or inefficient to train
- Neural networks are trained using variants of **gradient descent**
- **Architectural choices** can make a network easier to train, less prone to overfitting