Neural Networks

CMPUT 261: Introduction to Artificial Intelligence

GBC §6.0-6.4.1

Lecture Outline

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

After this lecture, you should be able to:

- define an activation function
- define a rectified linear activation and give an expression for its value
- describe how the units in a feedforward neural network are connected
- give an expression in matrix notation for a layer of a feedforward network
- explain (high level) what the Universal Approximation Theorem guarantees
- describe the basic procedure for training a neural network
- identify the parameters of a feedforward neural network

Logistics

- Assignment #2 due **Tuesday, Oct 25 at 11:59pm**
- Midterm is **Thursday, Nov 3**
  - Coverage: Everything up to and including lecture 18 (Nov 1: Image Data)

Recap: Supervised Learning

- **Supervised learning task:**
  predict the values of **target features** Y based on **input features** X
- Formally: Choose a hypothesis $h : \mathcal{X} \to \mathcal{Y}$ from a hypothesis space $\mathcal{H}$
- We use the value of a **loss function** L applied to a set of **training examples** $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ to choose the hypothesis
  - **Regularization** penalty biases optimization toward simpler functions:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} L(h) = \sum_{j=1}^{n} \ell(h(x_i), y_i) + \lambda \, \text{penalty}(h)$$

  - Simpler functions are more likely to **generalize**
- Generalization performance is evaluated on the **test set**
- Another way to reduce overfitting: Learn **distribution** over hypotheses (Bayesian)
  - Many regularization approaches amount to a particular prior

(Generalized) Linear Models

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

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

weights

Linear model

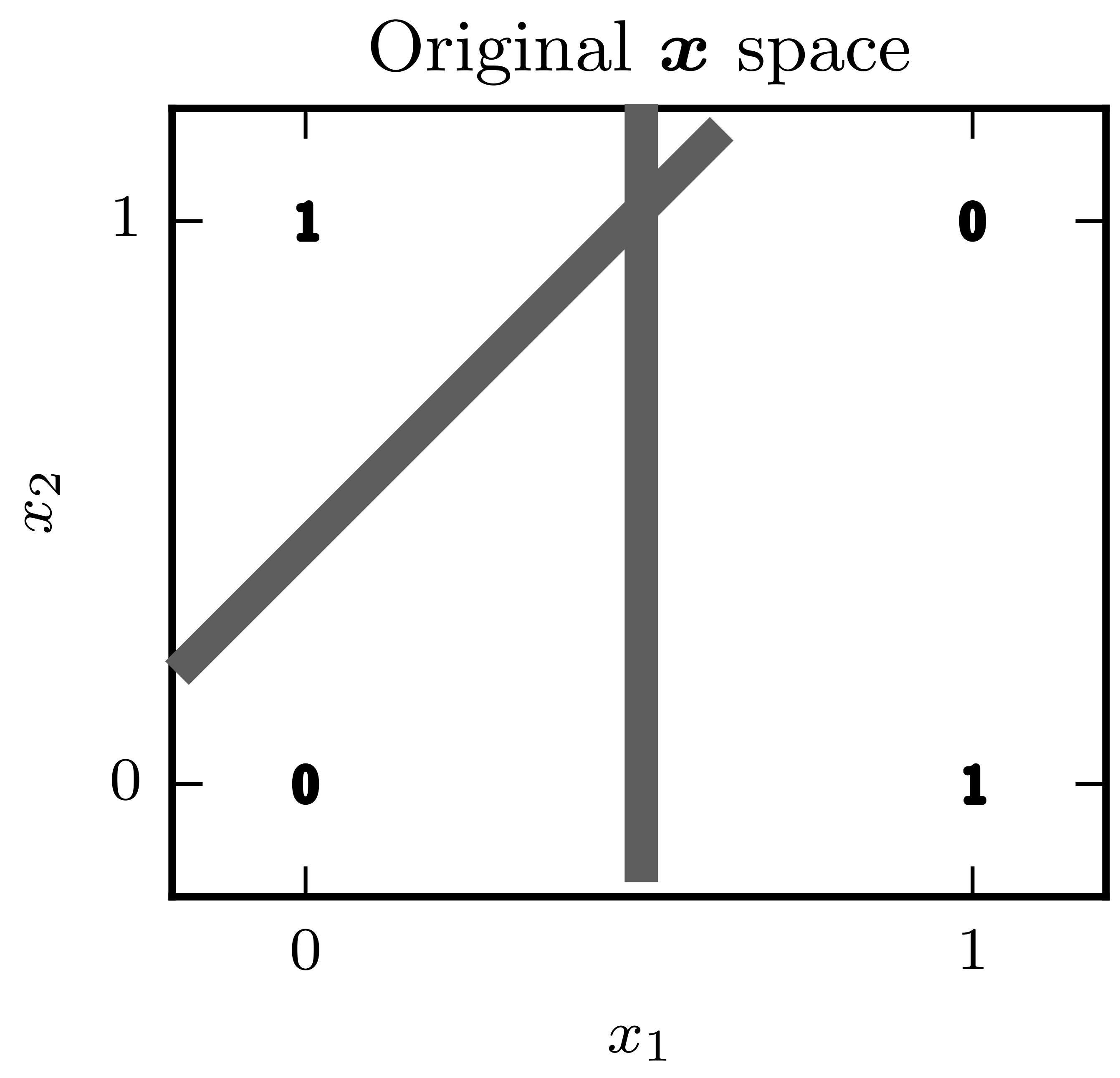
inputs

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

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

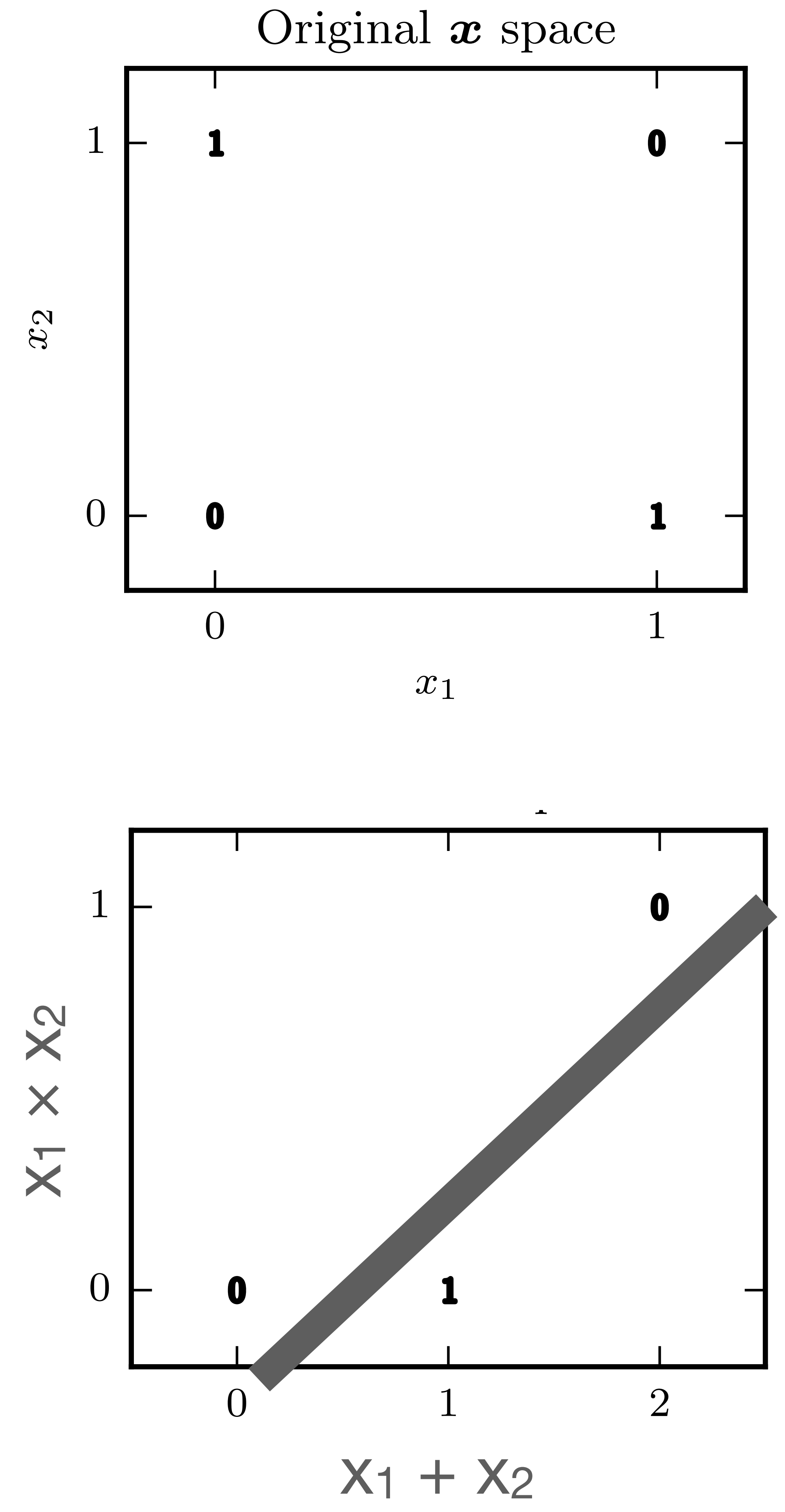
One option: Learn a linear model on **richer inputs**

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**

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

Nonlinear Features for XOR

- **Question:**
  What additional features would help?
- The product of x_1 and x_2!
  - $\phi(x_1, x_2) = [1, x_1, x_2, x_1 x_2]^\top$
  - $\mathbf{w} = [-0.2, 0.5, 0.5, -2]^\top$
- $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^\top \phi(\mathbf{x}) > 0$ for $(0,1)$ and $(1,0)$
  $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^\top \phi(\mathbf{x}) < 0$ for $(1,1)$ and $(0,0)$



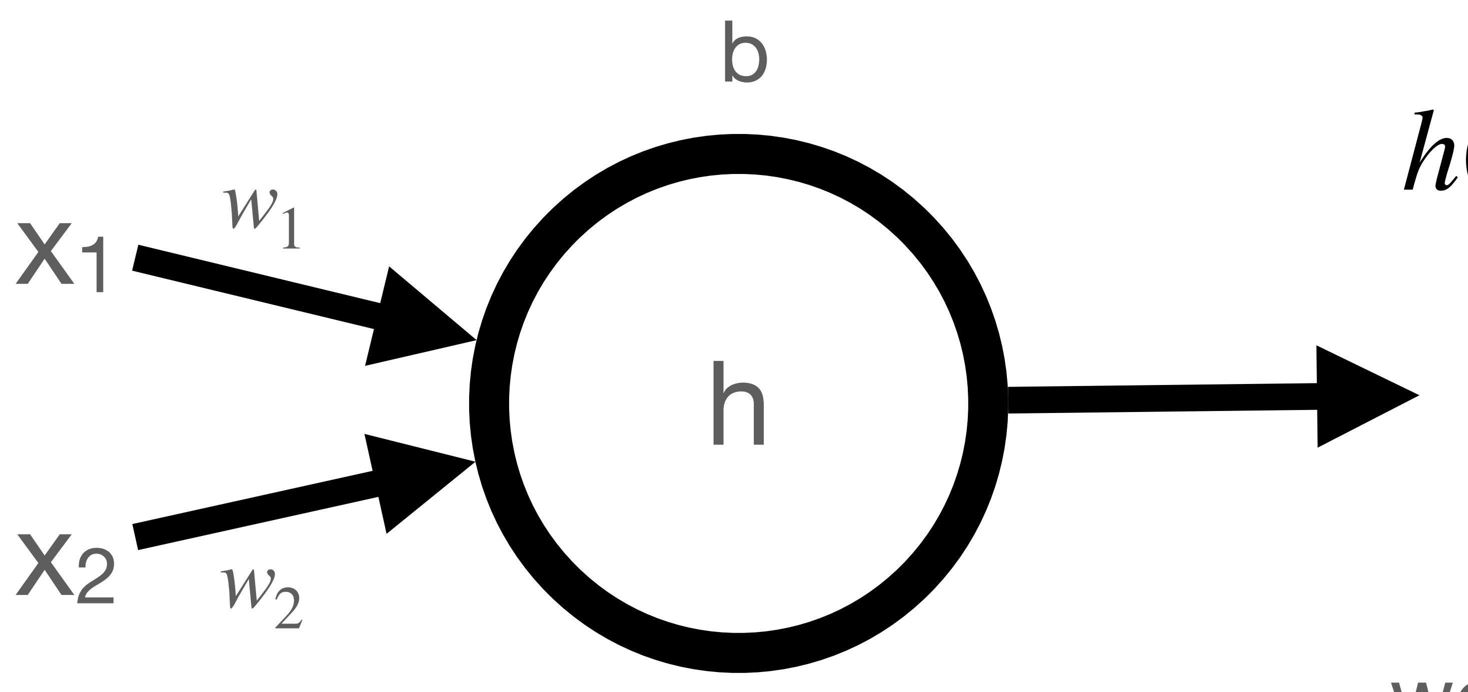
(Image: Goodfellow 2017)

Learning Nonlinear Features

- Manually constructing good features is **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**



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

weights

offset

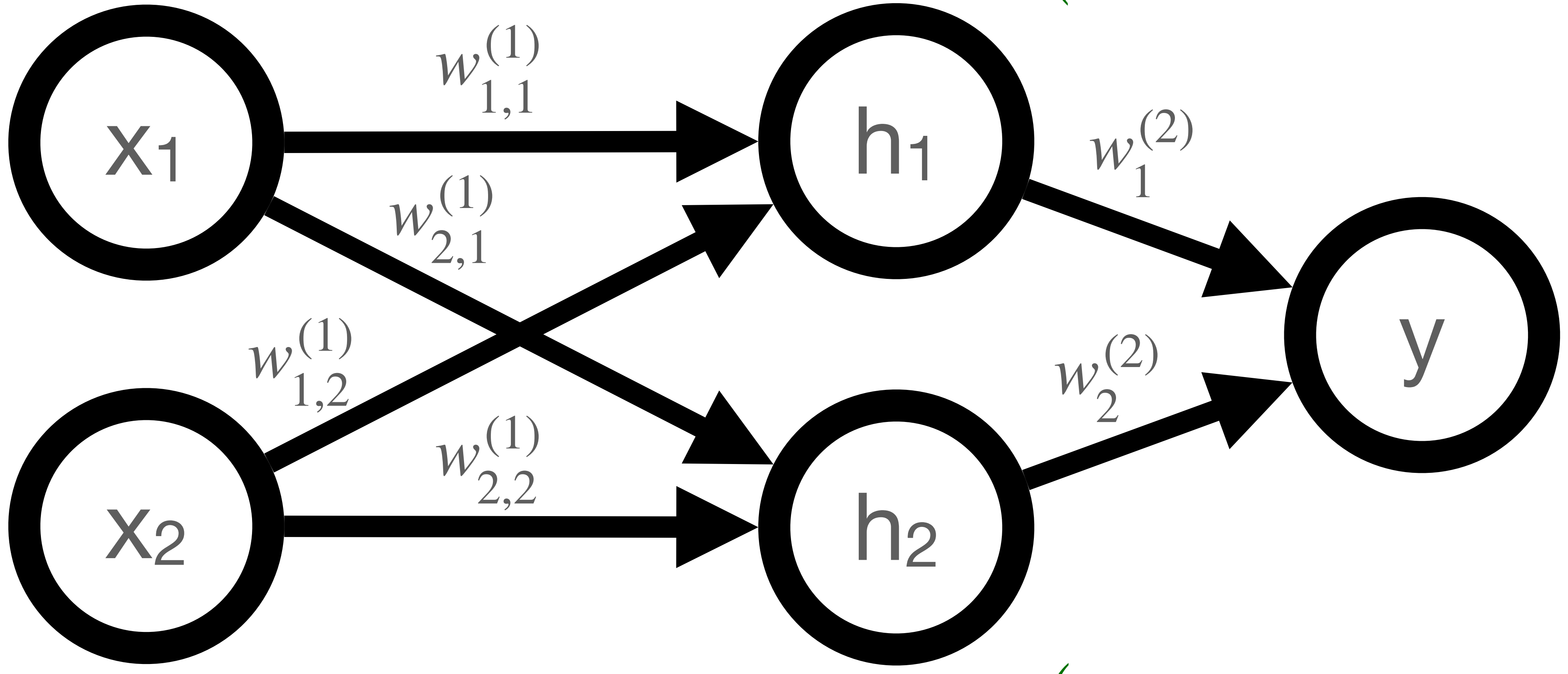
activation function

- **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**

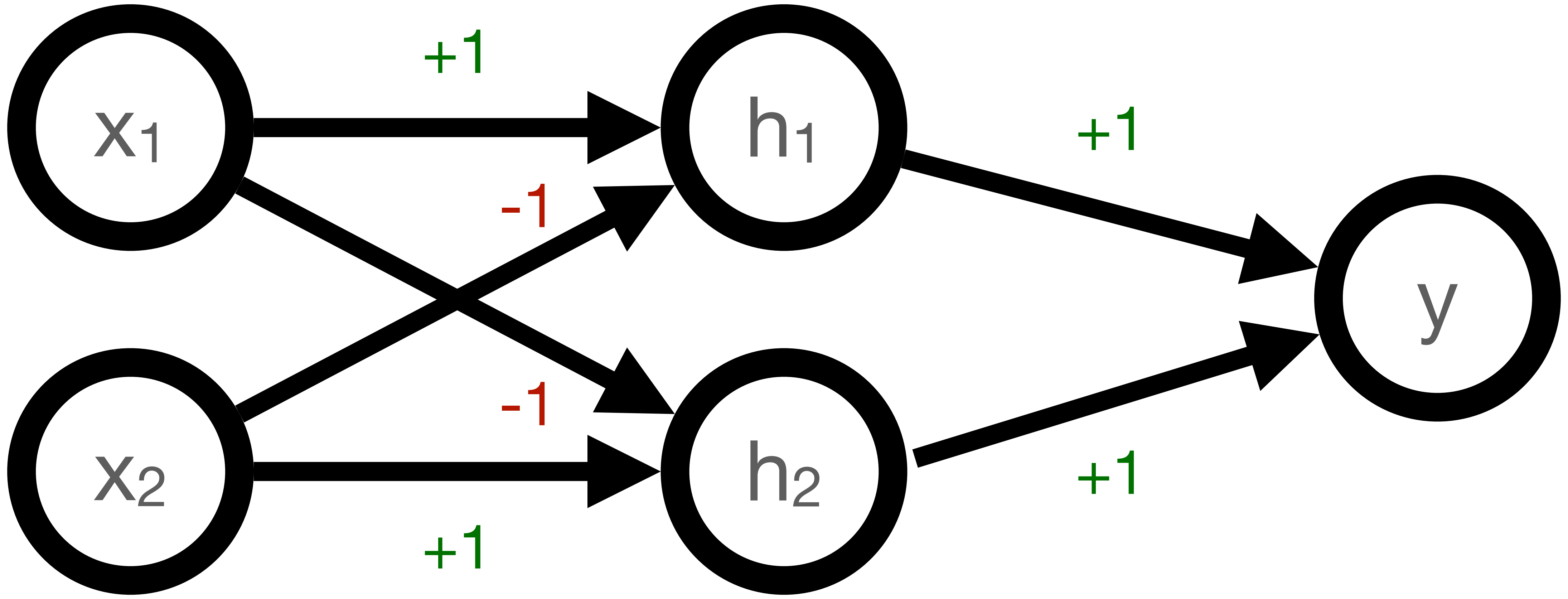
$$h_1 = g\left(w_{1,1}^{(1)}x_1 + w_{1,2}^{(1)}x_2 + b_1^{(1)}\right)$$



$$y = g\left(w_1^{(2)}h_1 + w_2^{(2)}h_2 + b^{(2)}\right)$$

$$h_2 = g\left(w_{2,1}^{(1)}x_1 + w_{2,2}^{(1)}x_2 + b_2^{(1)}\right)$$

Example: XOR network

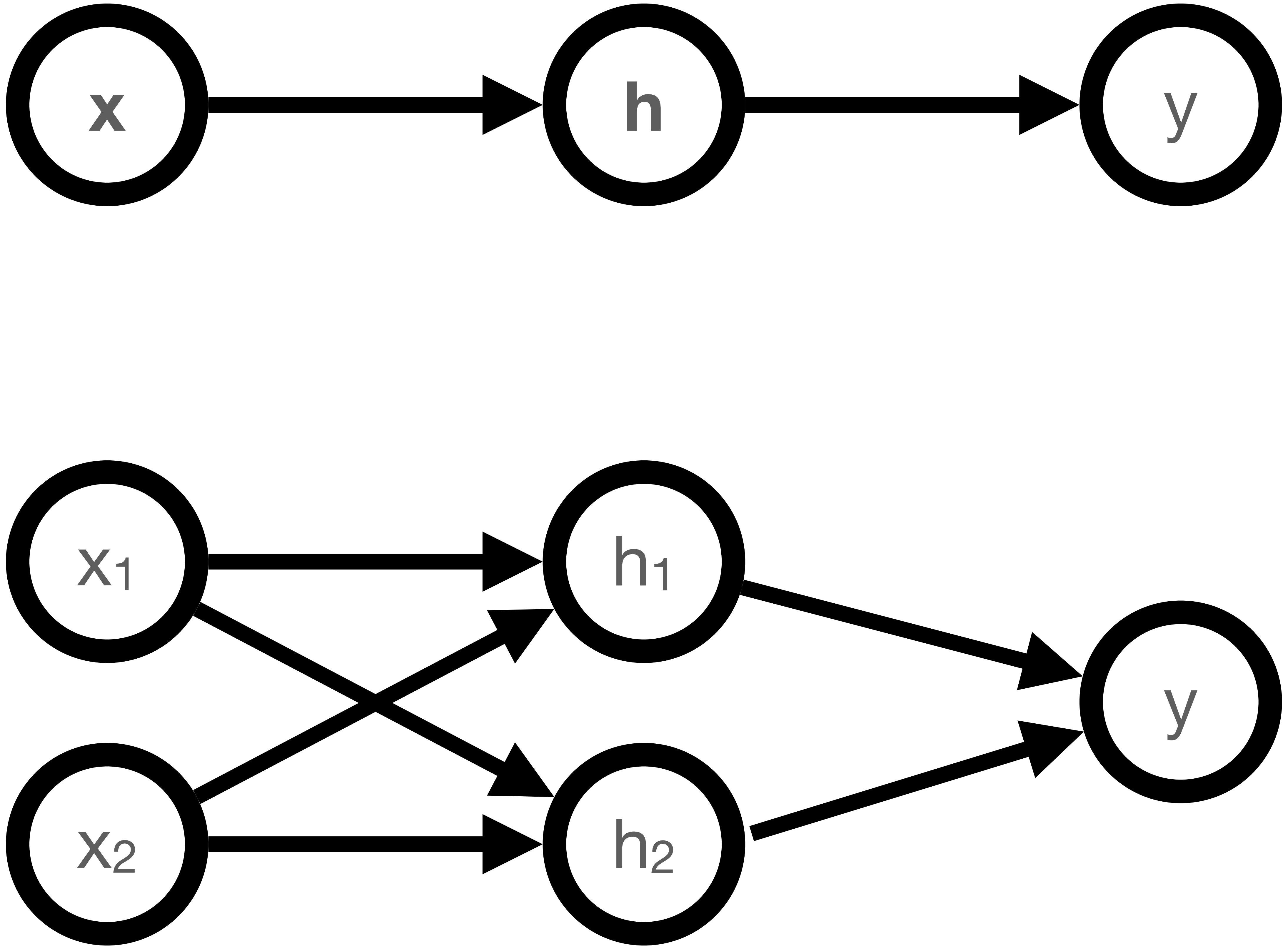


- *Activation*: $g(z) = \max\{0, z\}$ ("**rectified linear unit**")
- *Offsets:* 0
- *Weights:*
  - $[+1, -1]$ for h_1; $[-1, +1]$ for h_2
  - $[+1, +1]$ for y

Question:

When does $h_1 = 1$?

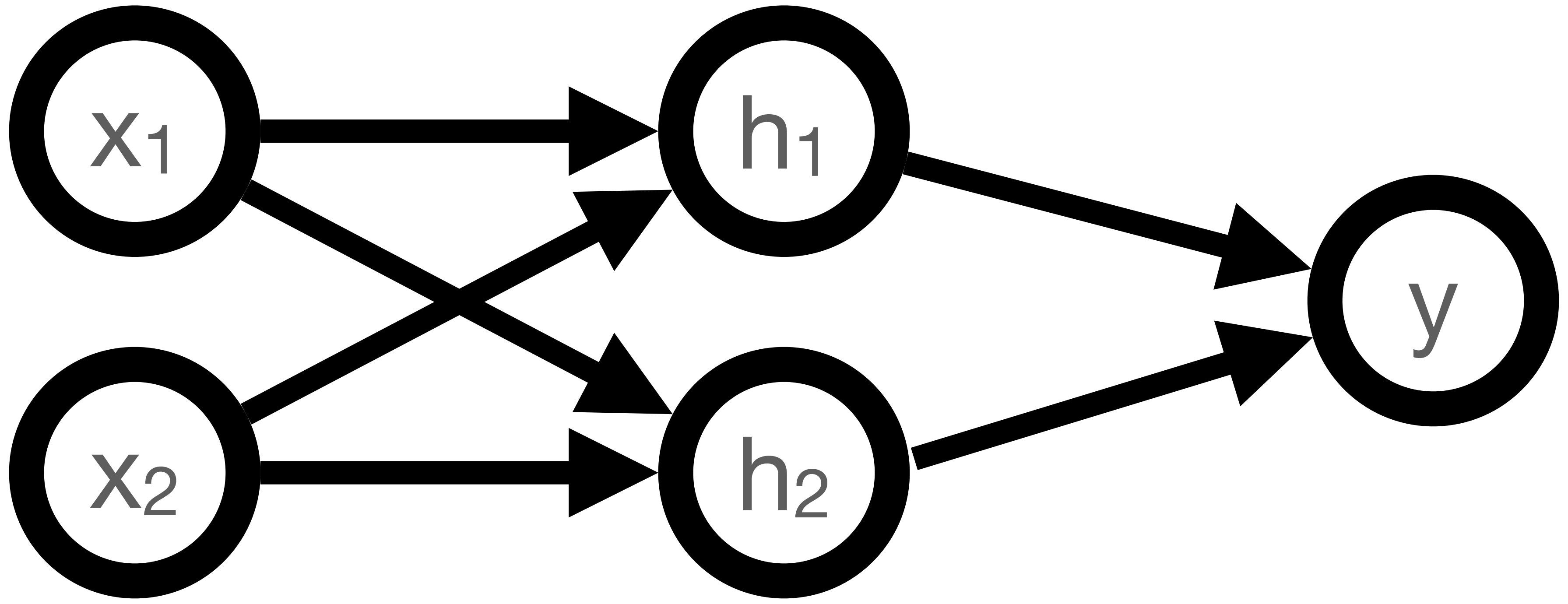
Matrix Representation of Layers



- 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** $\mathbf{W}$
- 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)

Neural Network Parameters

$$y = f(x; \theta)$$

A neural network is just a **supervised model**

- It is a function that takes **inputs** $\mathbf{x}$, and computes an output y based on **parameters** θ
- **Question:** What is θ in a feedforward neural network?

Training Neural Networks

- Specify a **loss** L and a set of **training examples:**

$$E = (\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})$$

- Training by **gradient descent**:

1. Compute **loss** on training data: $L(\mathbf{W}, \mathbf{b}) = \sum_i \ell\left(f(\mathbf{x}^{(i)}; \mathbf{W}, \mathbf{b}), y^{(i)}\right)$

   Loss function (e.g., squared error)

   Prediction — Target

2. Compute **gradient** of loss: $\nabla L(\mathbf{W}, \mathbf{b})$

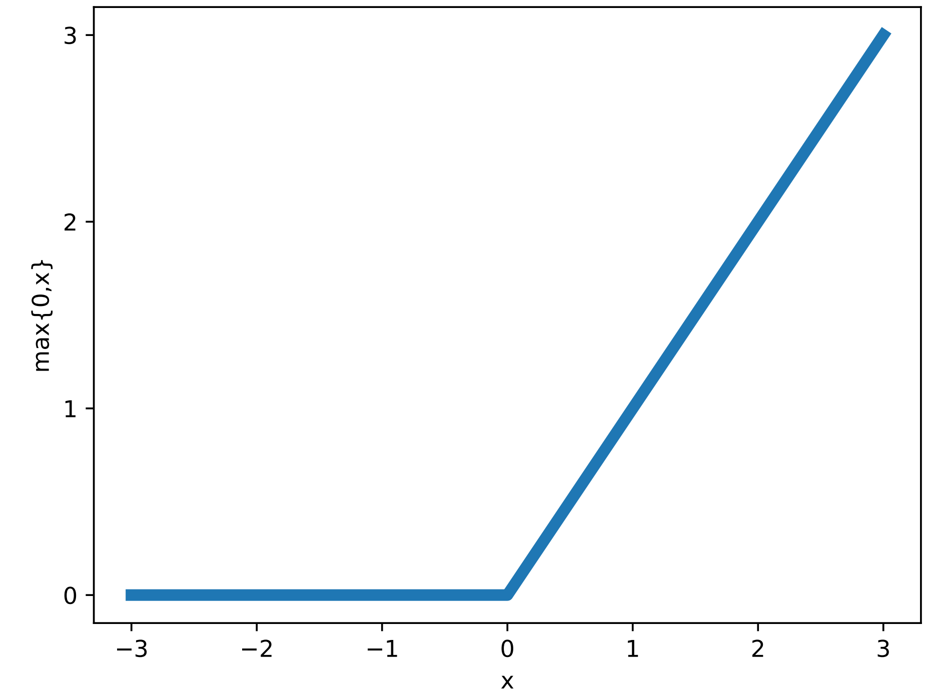
   (Subsequent lecture)

3. **Update parameters** to make loss smaller:

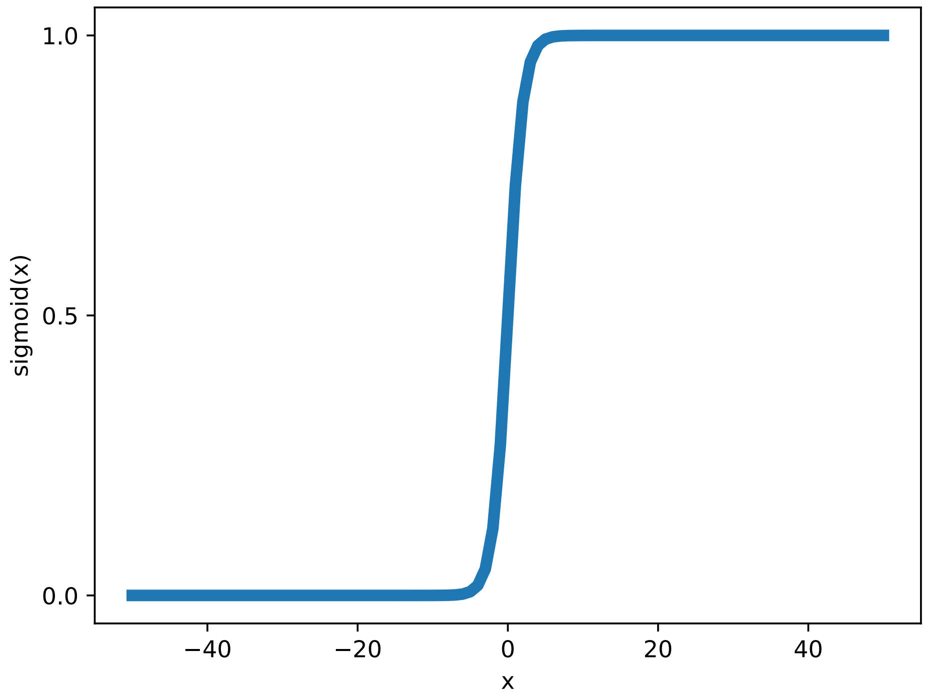
$$\begin{bmatrix} \mathbf{W}^{new} \\ \mathbf{b}^{new} \end{bmatrix} = \begin{bmatrix} \mathbf{W}^{old} \\ \mathbf{b}^{old} \end{bmatrix} - \eta \nabla L(\mathbf{W}^{old}, \mathbf{b}^{old})$$

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** for many applications
- 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