Neural Networks

CMPUT 261: Introduction to Artificial Intelligence

P §3.1-3.6

Logistics

- **Quiz #2** is currently being marked
- **Assignment #3** will be released next week

Lecture Outline

1. Recap
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

Recap: Supervised Learning

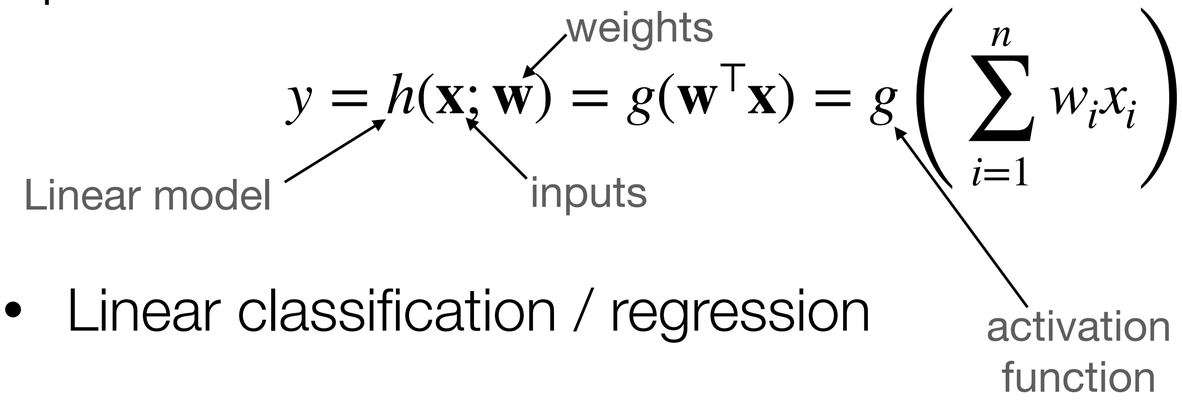
- **Supervised learning task:**
  predict the values of **target features** Y based on **input features** X
- Formally: Choose a hypothesis $h : \mathcal{X} \rightarrow \mathcal{Y}$ from a hypothesis space $\mathcal{H}$
- We use the value of a **loss function** L applied to a set of **training examples** $S = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$ to choose the hypothesis
  - **Regularization** penalty biases optimization toward simpler functions:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} L(h) = \sum_{i=1}^{n} \ell(h(\mathbf{x}^{(i)}), y^{(i)}) + \lambda \, \text{penalty}(h)$$

  - 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 MAP estimate with a particular prior

(Generalized) Linear Models

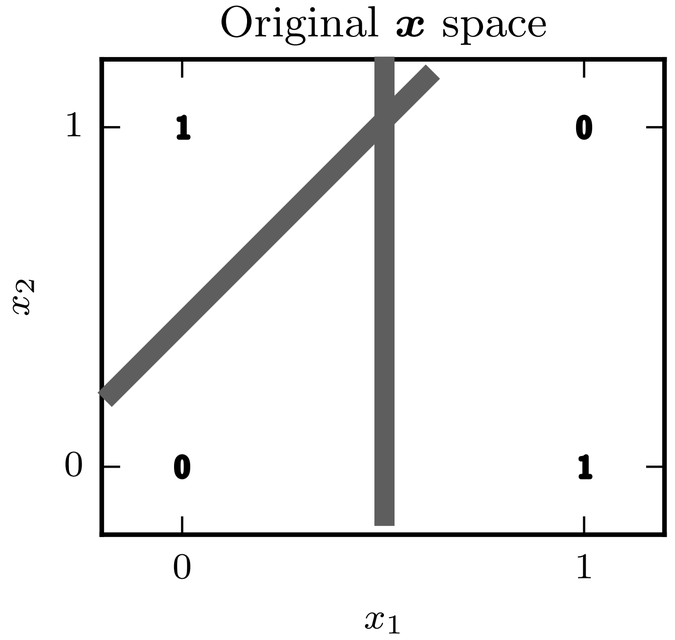
- Supervised models we have considered so far have been **linear:**



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

Example: XOR

- The function $h(x_1, x_2) = (x_1 \text{ XOR } x_2)$ 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?



A 2D graph with 0 and 1 on the x axis and 0 and 1 on the y axis, representing XOR: at the 0,0 and 1,1 corners, there is a 0; at the 1,0 and 0,1 corners, there is a 1. A vertical line has both 0 and 1 on its left, and a 0 and 1 on its right; a diagonal line has a single 1 above it, and both 0s and a 1 below it.

(Image: Goodfellow 2017)

Nonlinear Features

$$h(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^\top \mathbf{x}) = g\left(\sum_{j=1}^{d} w_j x_j\right)$$

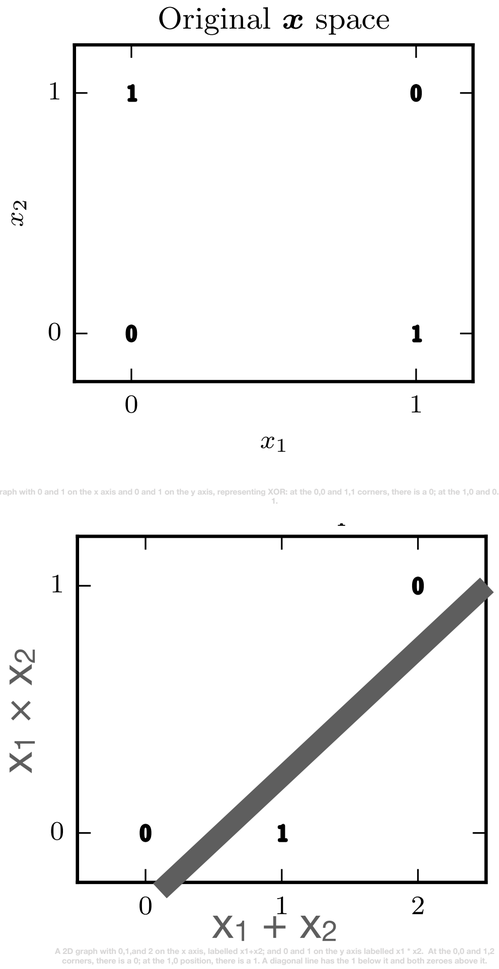
One option: Learn a linear model on **richer inputs**

1. Define a **feature mapping** $\boldsymbol{\phi}(\mathbf{x})$ that returns **functions** of the original inputs
2. Learn a linear model of the **features** instead of the **inputs**

$$h(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})) = g\left(\sum_{j=1}^{d} w_j [\boldsymbol{\phi}(\mathbf{x})]_j\right)$$

Nonlinear Features for XOR

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



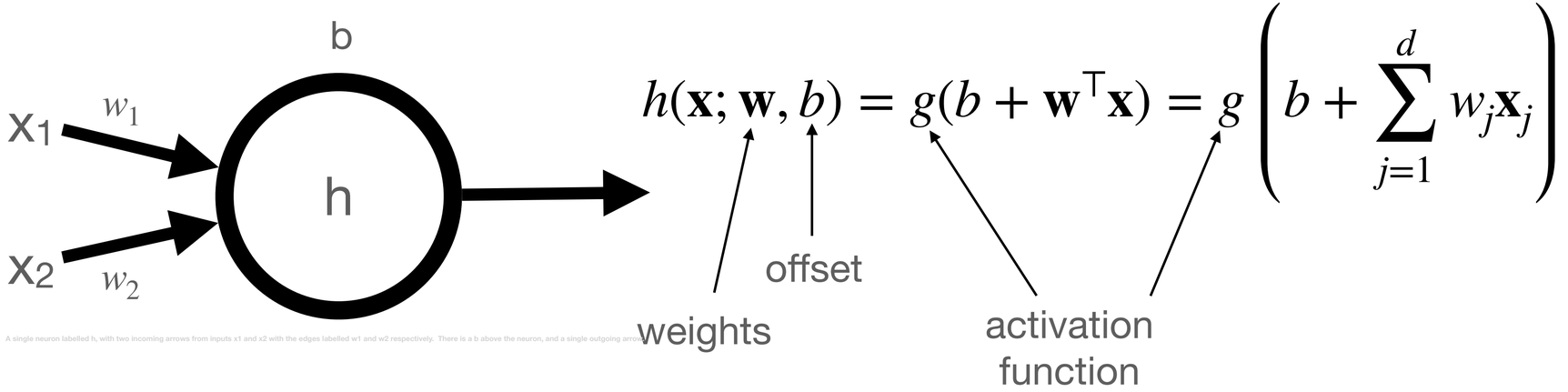
(Image: Goodfellow 2017)

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

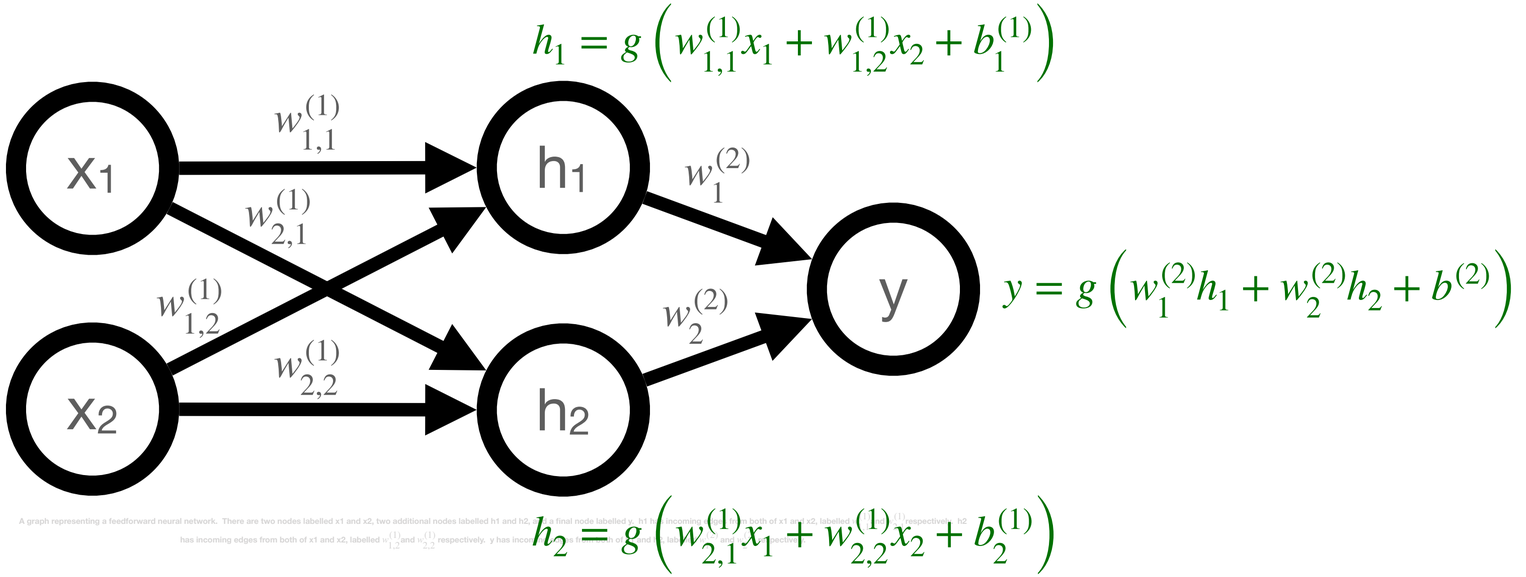
- Deep learning learns ϕ by **composing** little functions
- These function are called **units**



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

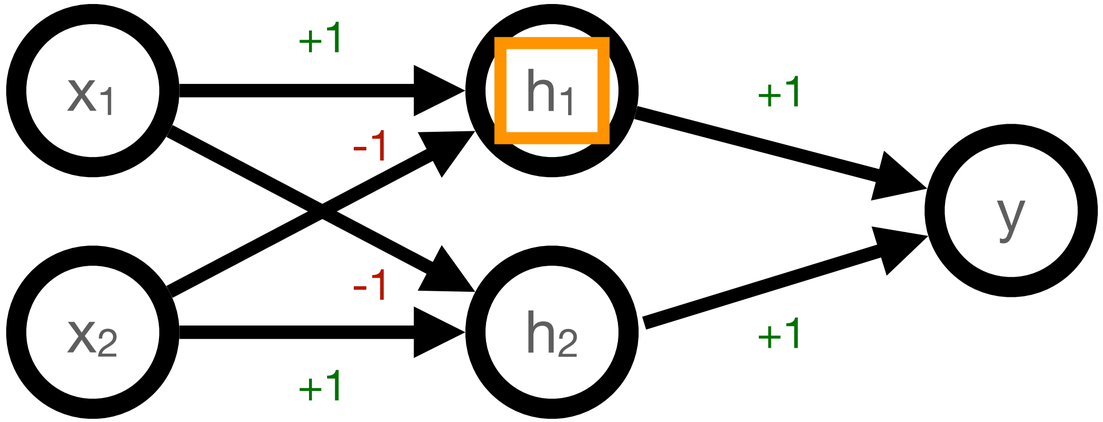
Feedforward Neural Network

- 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**



A graph representing a feedforward neural network. There are two nodes labelled x1 and x2, two additional nodes labelled h1 and h2, and a final node labelled y. h1 has incoming edges from both of x1 and x2, labelled $w_{1,1}^{(1)}$ and $w_{1,2}^{(1)}$ respectively. h2 has incoming edges from both of x1 and x2, labelled $w_{2,1}^{(1)}$ and $w_{2,2}^{(1)}$ respectively. y has incoming edges from both of h1 and h2, labelled $w_1^{(2)}$ and $w_2^{(2)}$ respectively.

Example: XOR network



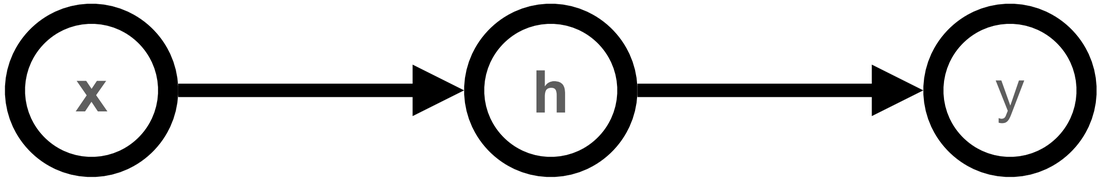
A graph representing a feedforward neural network. There are two nodes labelled x1 and x2, two additional nodes labelled h1 and h2, and a final node labelled y. h1 has an incoming edge from x1 labeled +1 and an incoming edge from x2 labeled -1. h2 has an incoming edge from x1 labeled -1 and an incoming edge from x2 labeled +1. y has incoming edges from h1 and h2, both labeled +1.

- *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

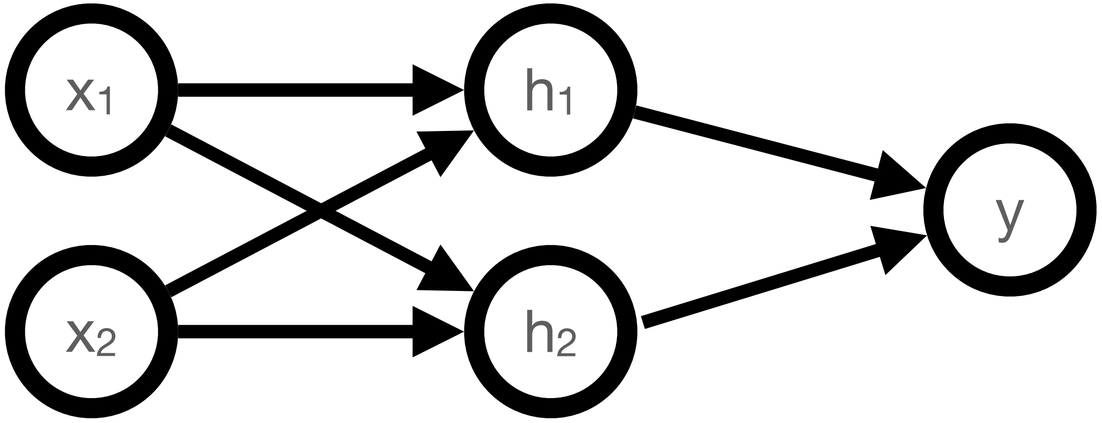
Question:

When does $h_1 = 1$?

Matrix Representation of Layers



A graph representing a feedforward network. One node is labelled x, one is labelled h, and one is labelled y. There is an edge from x to h, and an edge from h to y.

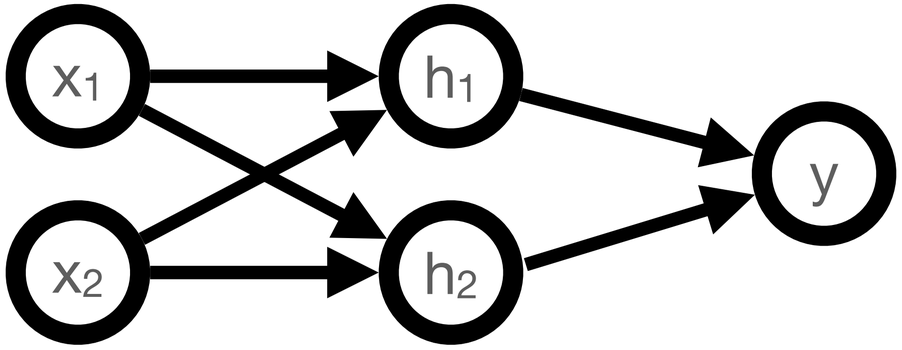


A graph representing a feedforward neural network. There are two nodes labelled x1 and x2, two additional nodes labelled h1 and h2, and a final node labelled y. h1 has an incoming edge from x1 and an incoming edge from x2. h2 has an incoming edge from x1 and an incoming edge from x2. y has incoming edges from h1 and h2.

- 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** $\mathbf{W} \in \mathbb{R}^{k \times d}$ (# hidden units, # inputs)
- The **offset term** for each unit can be collected into a vector $\mathbf{b} \in \mathbb{R}^k$:

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

Architecture



A graph representing a feedforward neural network. There are two nodes labelled x1 and x2, two additional nodes labelled h1 and h2, and a final node labelled y. h1 has an incoming edge from x1 and an incoming edge from x2. h2 has an incoming edge from x1 and an incoming edge from x2. y has incoming edges from h1 and h2.

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

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

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:**

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

- Training by **gradient descent**:

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

Loss function (e.g., squared error)

Prediction

Target

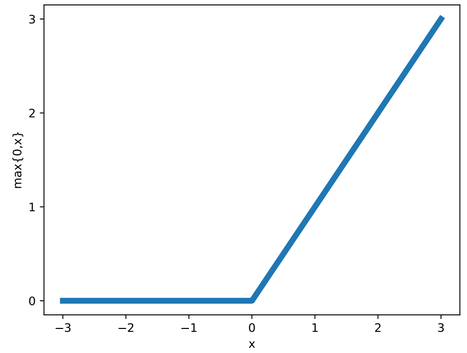
2. Compute **gradient** of loss: $\nabla L(\mathbf{W}, \mathbf{b})$ (Subsequent lecture)

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

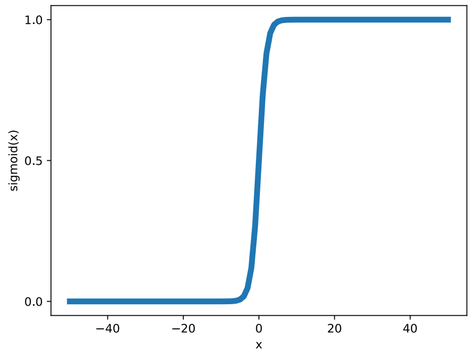
$$\begin{bmatrix} \mathbf{W}^{new} \\ \mathbf{b}^{new} \end{bmatrix} = \begin{bmatrix} \mathbf{W}^{old} \\ \mathbf{b}^{old} \end{bmatrix} - \eta \nabla L(\mathbf{W}^{old}, \mathbf{b}^{old})$$

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

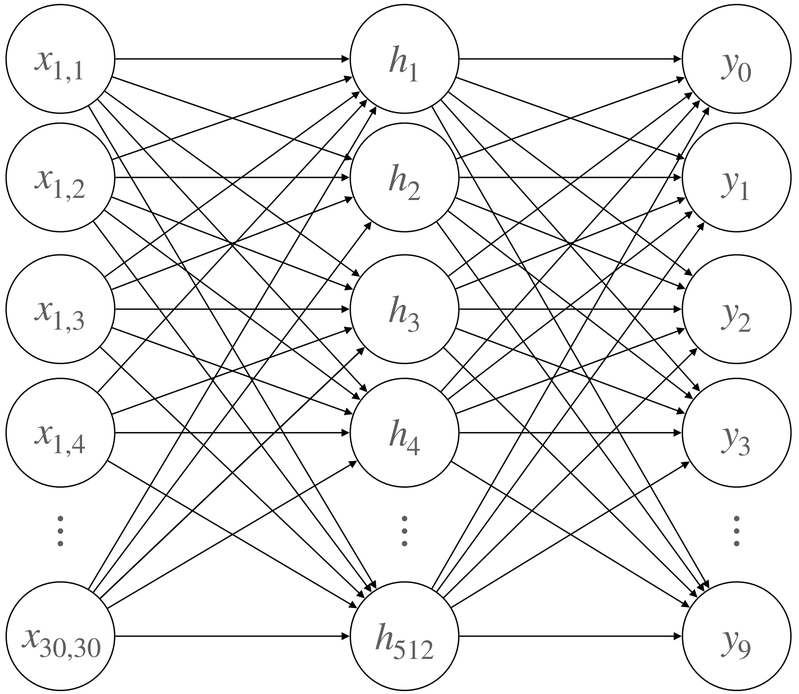


A graph representing the linear rectified unit activation function. Left of 0, the function is flat at 0; right of 0, the function is a straight 45-degree diagonal line.



The graph for the sigmoid function; the x-axis plots t from -5 to 5, and the y-axis plots the sigmoid function with a range within 0 to 1.

Example: Feedforward network for MNIST



- The MNIST dataset has grayscale images of handwritten individual digits
- Each image is 30×30 pixels
- Possible network for recognizing digits:
  - Each pixel is an input
  - One hidden layer of 512 nodes with ReLU activation
  - Output layer of 10 nodes with a combined softmax activation

Torch: Representating Layers

```
# Input will be a tensor with dimension `(n_datapoints, n_channels, height, width)`
# So `(64, 1, 30, 30)` in the case of MNIST dataset
self.layers = Sequential(
    Flatten(),                                  # -> (64, 900)
    Linear(in_features=30*30, out_features=512),  # -> (64, 512)
    ReLU(),                                     # -> (64, 512)
    Linear(in_features=512, out_features=10),     # -> (64, 10)
    LogSoftmax(dim=1)                           # -> (64, 10)
    )
```

Torch thinks about **operations** rather than **units**

- We've been thinking about a unit as "weighted sum and then activation"
- Torch specifies the weighting (`Linear`) and then the activation (`ReLU`) **separately**
- This is especially handy for output layers, where you often want to normalize by the sum of all the outputs (e.g., `LogSoftmax`)

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