McGill **Artificial Intelligence** Society

Lecture 3: Neural Networks

Slides based off of Machine Learning at Berkeley
https://github.com/mlberkeley/Machine-Learning-Decal-Fall-2018

Today’s Lesson Plan

Linear and Logistic Regression Review

Motivation

The Perceptron

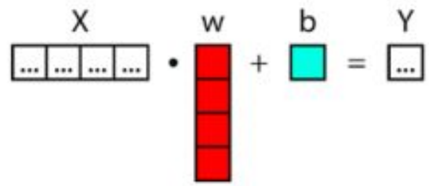
Feed-forward Neural Networks

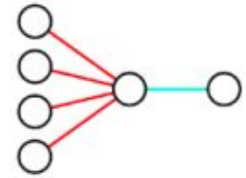
Learning In NNs

Coding Demo

Linear Regression

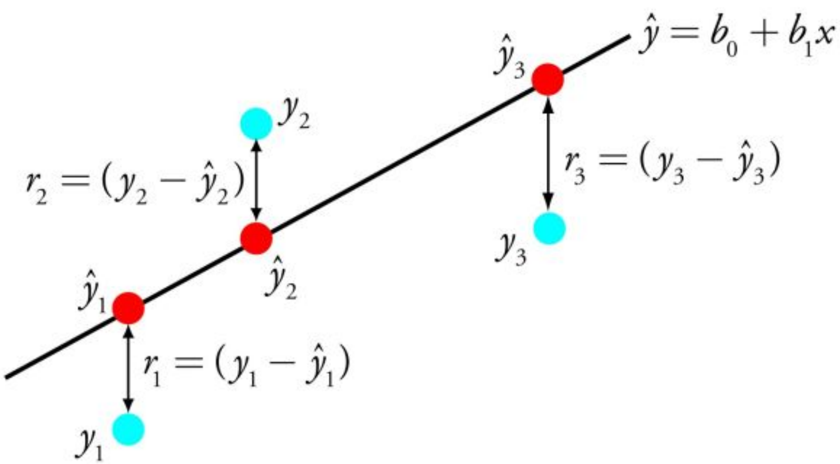
$$Y = f(X, w_1, b_1)$$
$$f(x, w, b) = x \cdot w + b$$





Let $\hat{y}_i = h(x) = b_0 + b_1 x$

$\min J(b_0, b_1)$

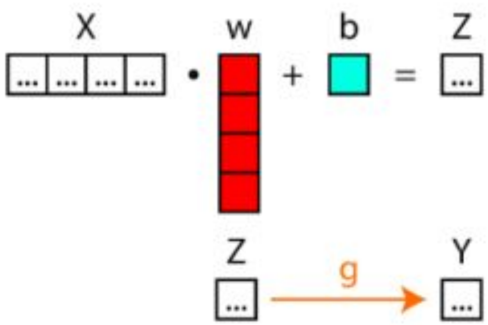


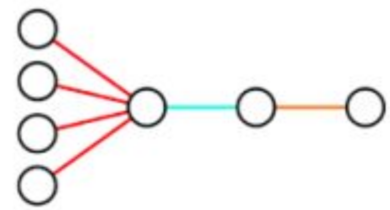
$$RSS(w) = \sum_{i=1}^{N} (y_i - w^T x_i)^2$$

$$J(b_0, b_1) = \frac{1}{2m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2$$

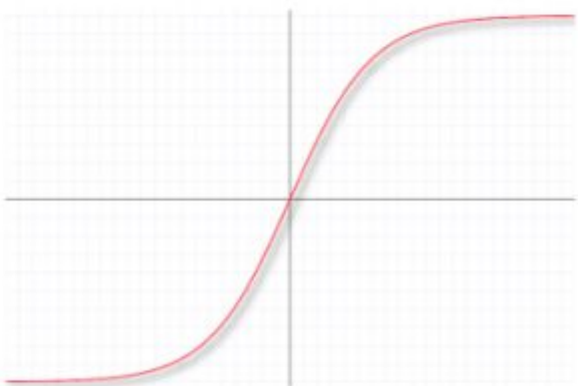
Logistic Regression

$$Z = f(X, w_1, b_1)$$
$$Y = g(Z)$$



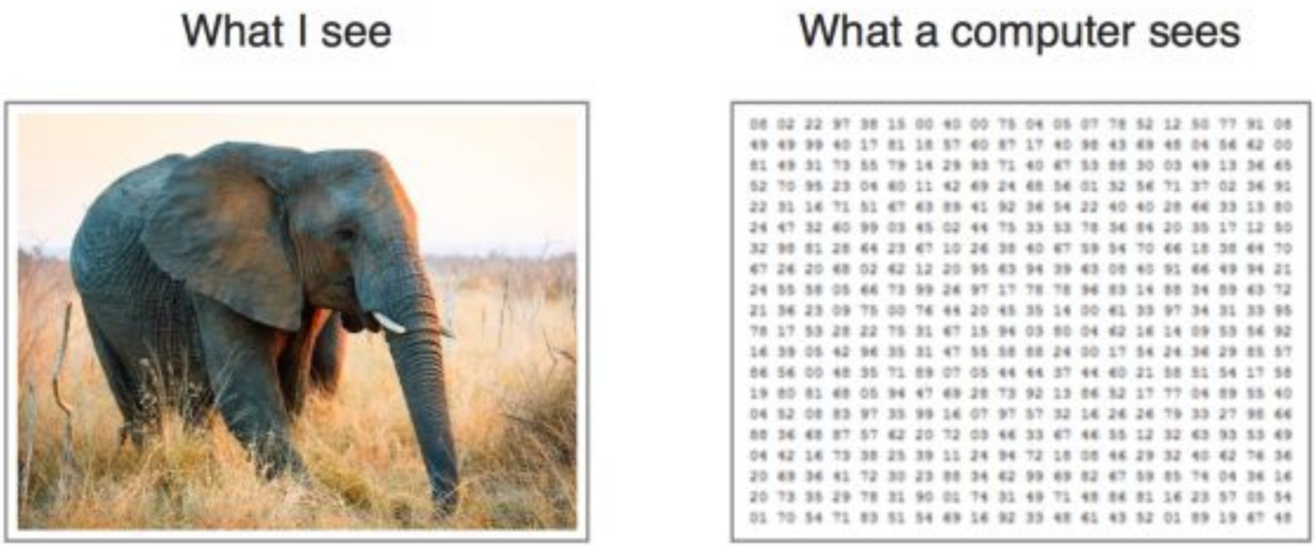


$$g(x) = \frac{1}{1+e^{-x}}$$



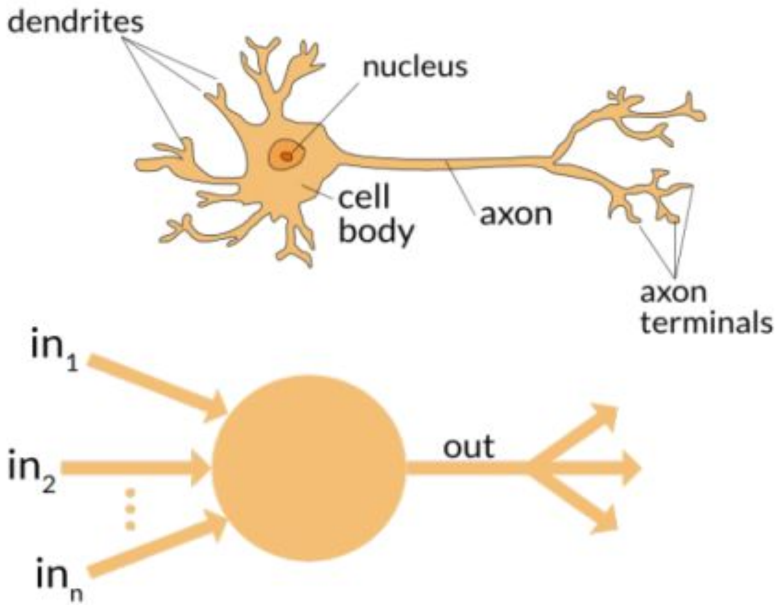
$$J(b) = -\sum_{i=1}^{m}\left(y^{(i)} \cdot \ln z^{(i)} + (1-y^{(i)}) \cdot \ln\left(1-z^{(i)}\right)\right)$$

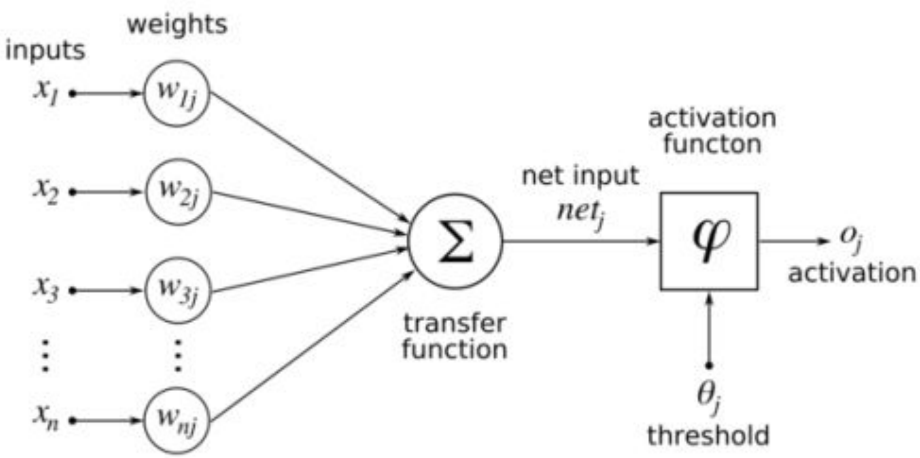
Motivation



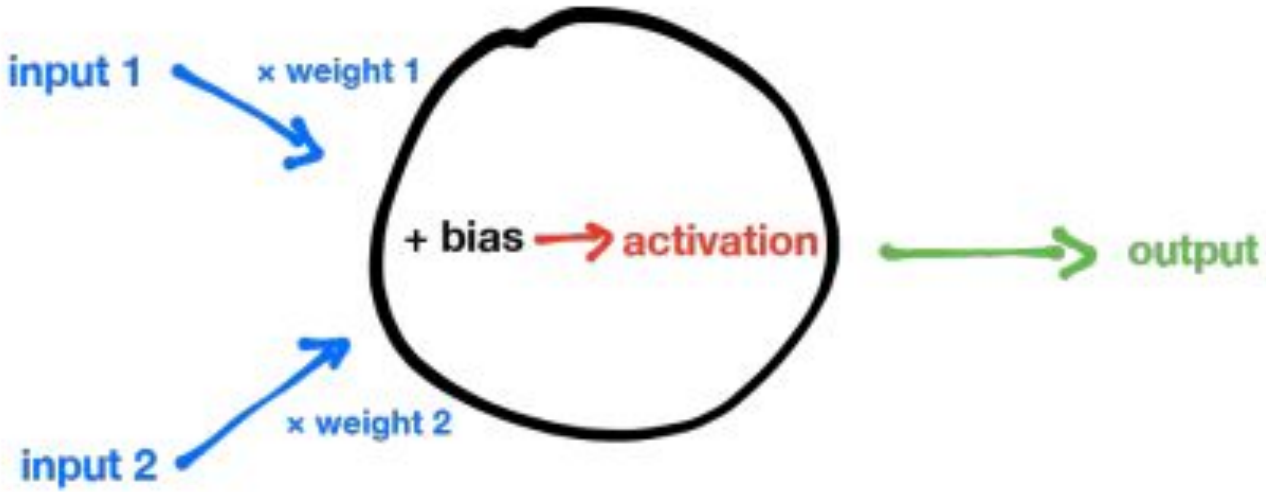
- Ability to learn complex, non-linear functions
- Does not impose fixed relationships in data
- Does not assume the inherent data distribution
    - Better at modeling non-constant variance

Biological Inspiration



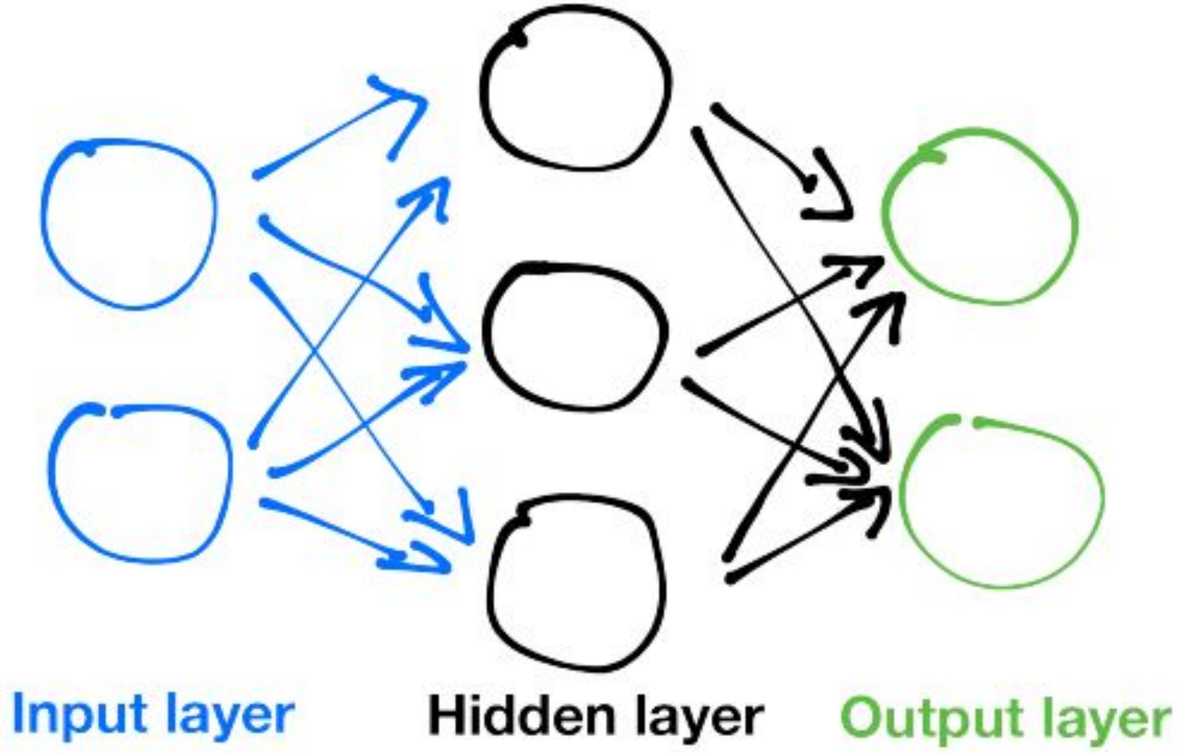


Biological Inspiration



- Inputs to the neuron are multiplied by weights
- Bias then summed with weighted inputs
- Non-linear activation function applied on wTx +b

Biological Inspiration - A Layered Architecture



Input Layer: takes in input data (size corresponds to input space)
Hidden Layer: neurons hidden from view (this is where the magic happens)
Output Layer: neurons in this layer provide the output of the network

The Perceptron

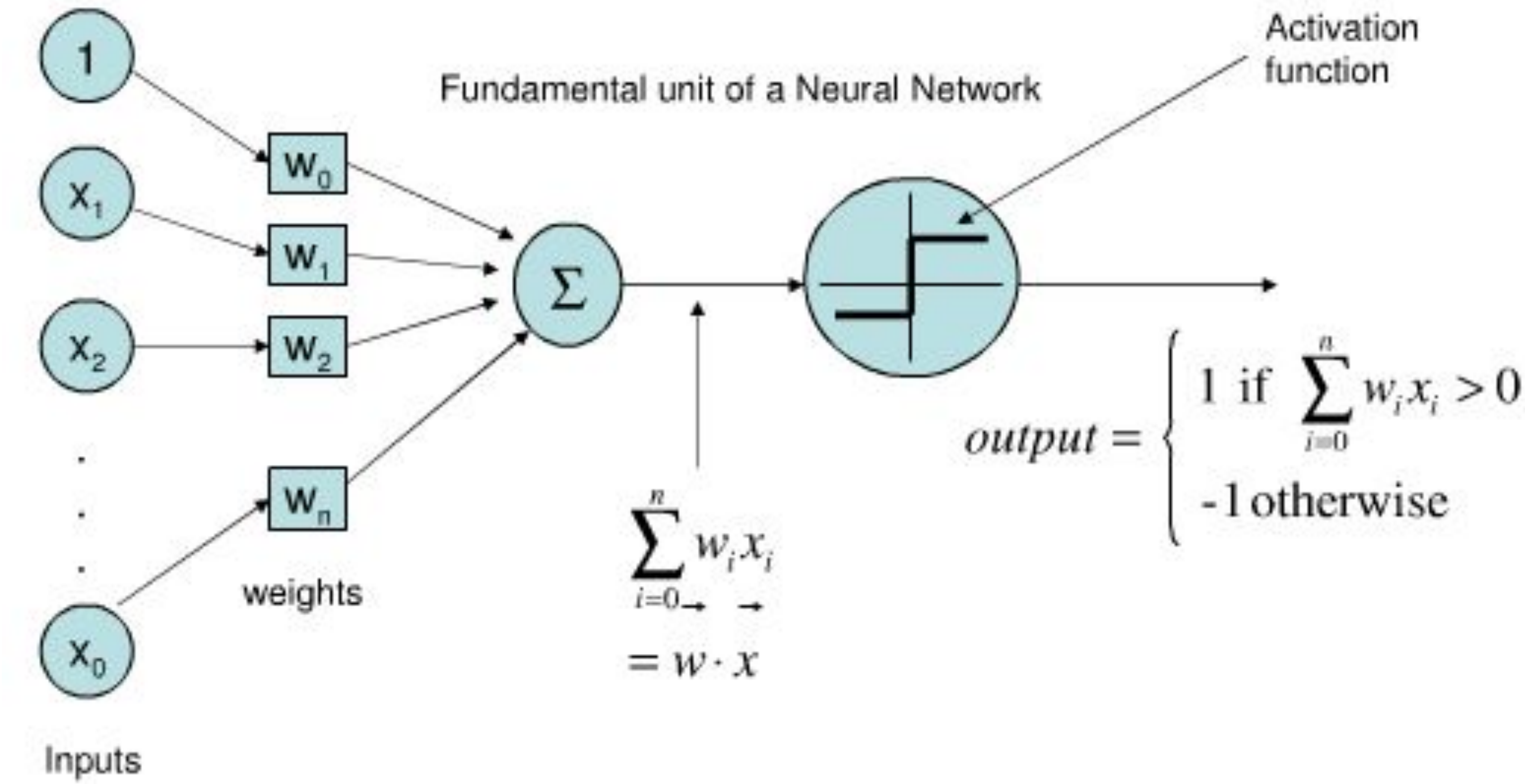
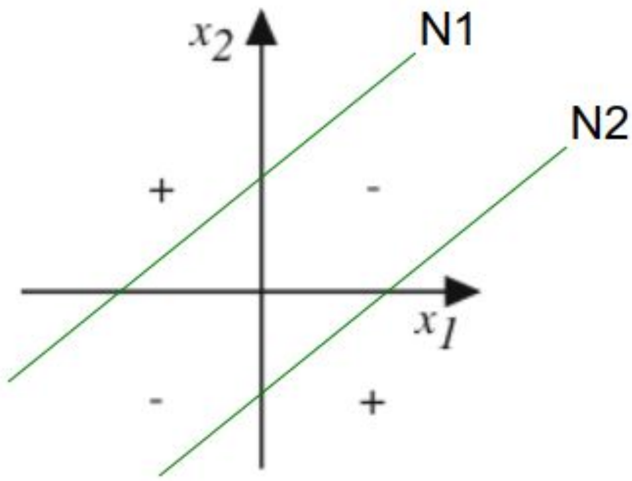


Image Credits From StackExchange

Perceptron Decision Boundary



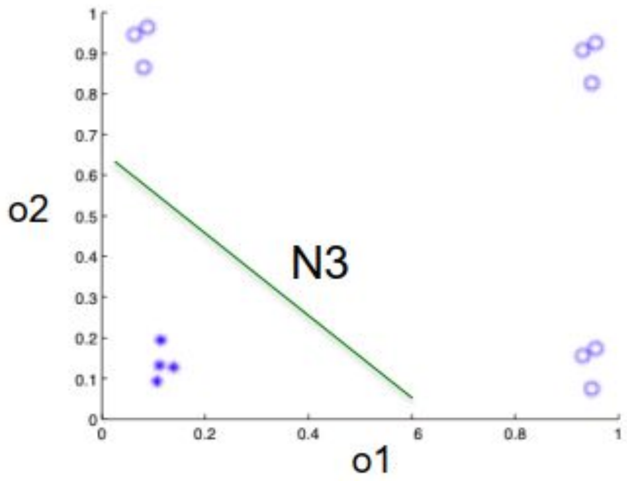
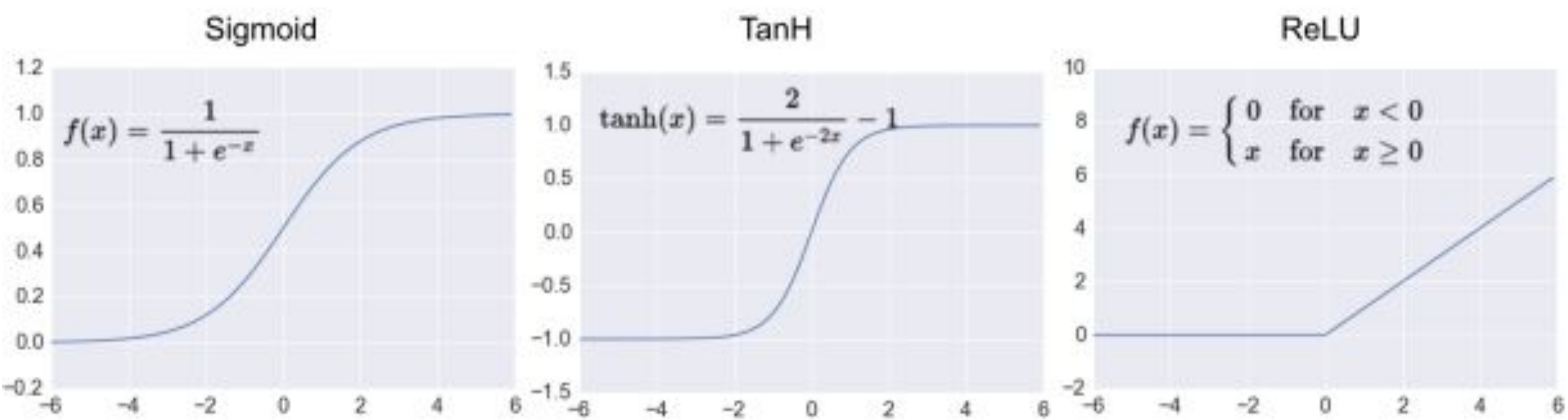


Image taken from Ryan Lowe's COMP551 Slides Lecture 14 Slide 5

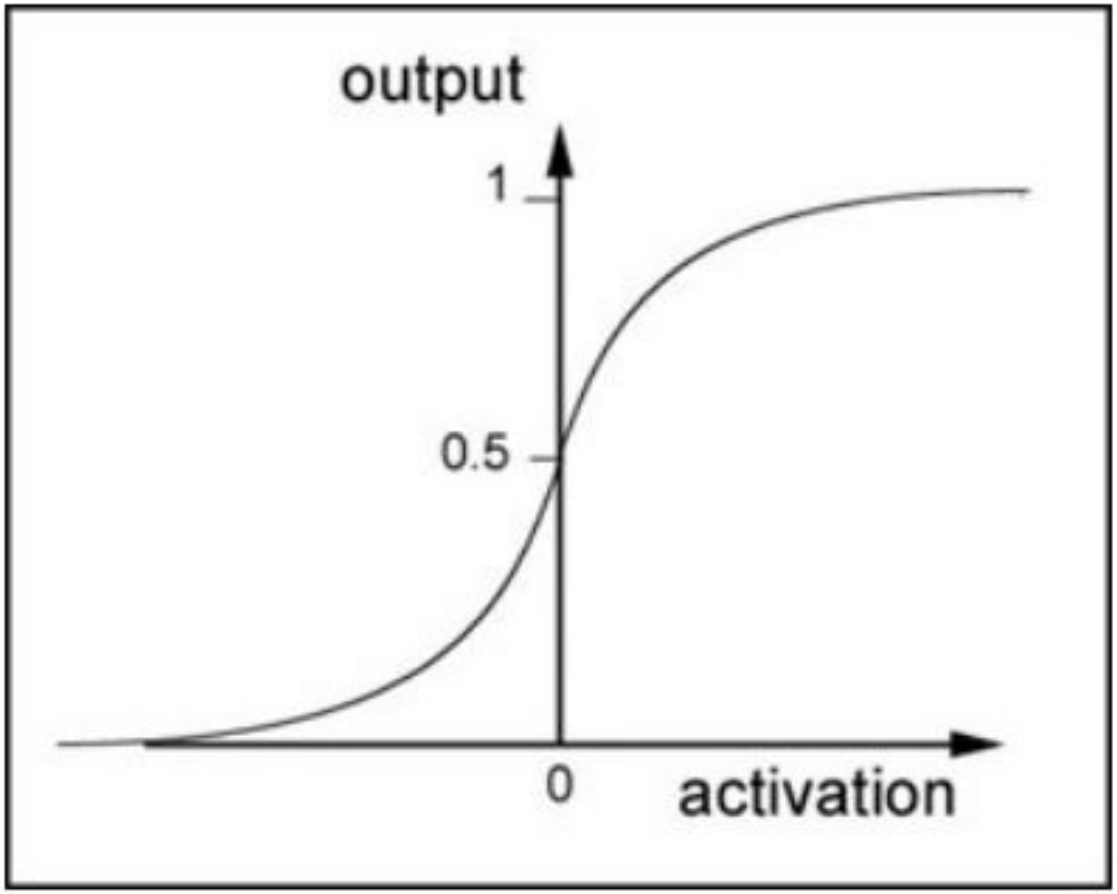
Activation Functions

For gradient descent to work, we need activation functions that are:

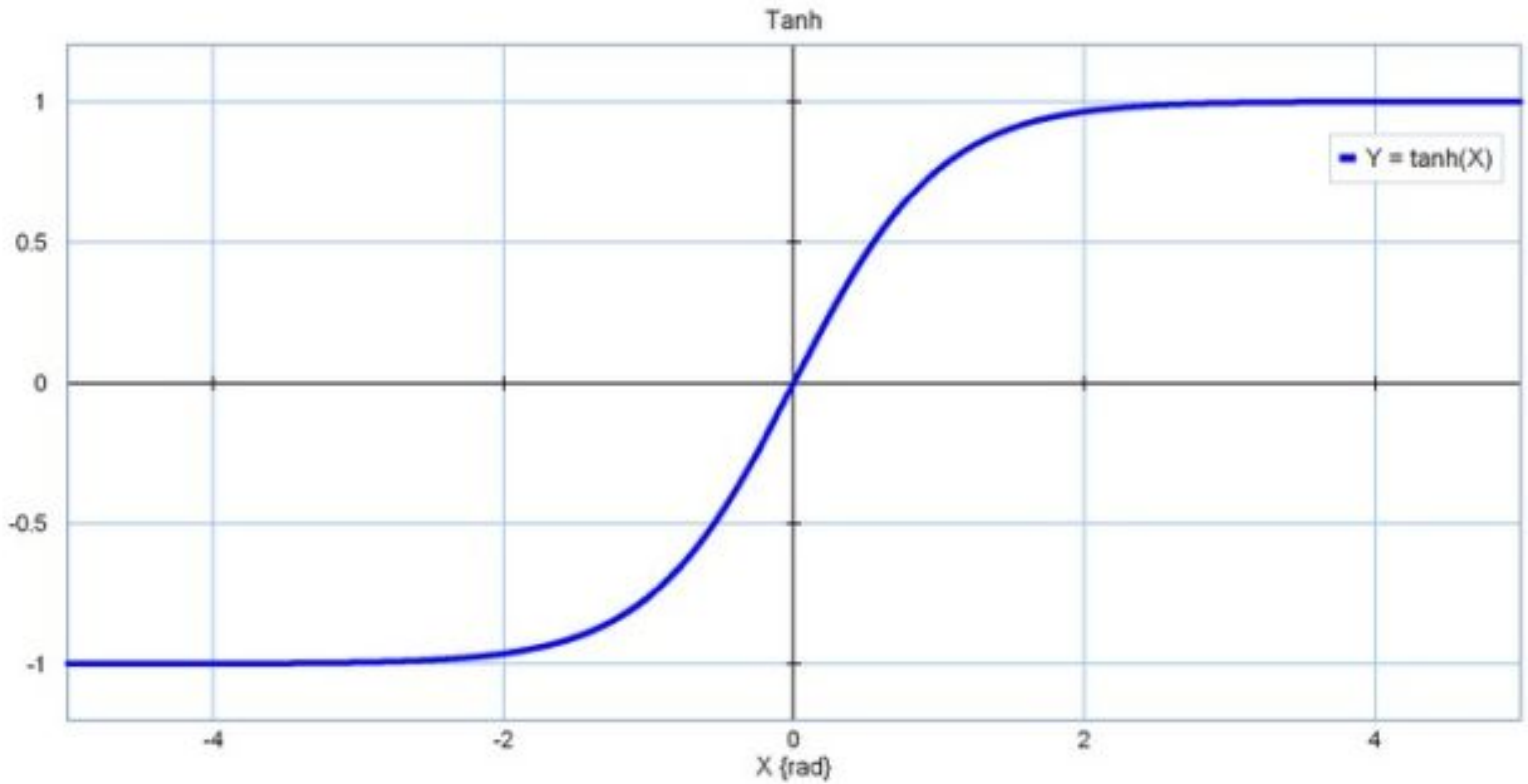
- Continuous
- Differentiable
- Monotonically Increasing



Sigmoid/Logistic

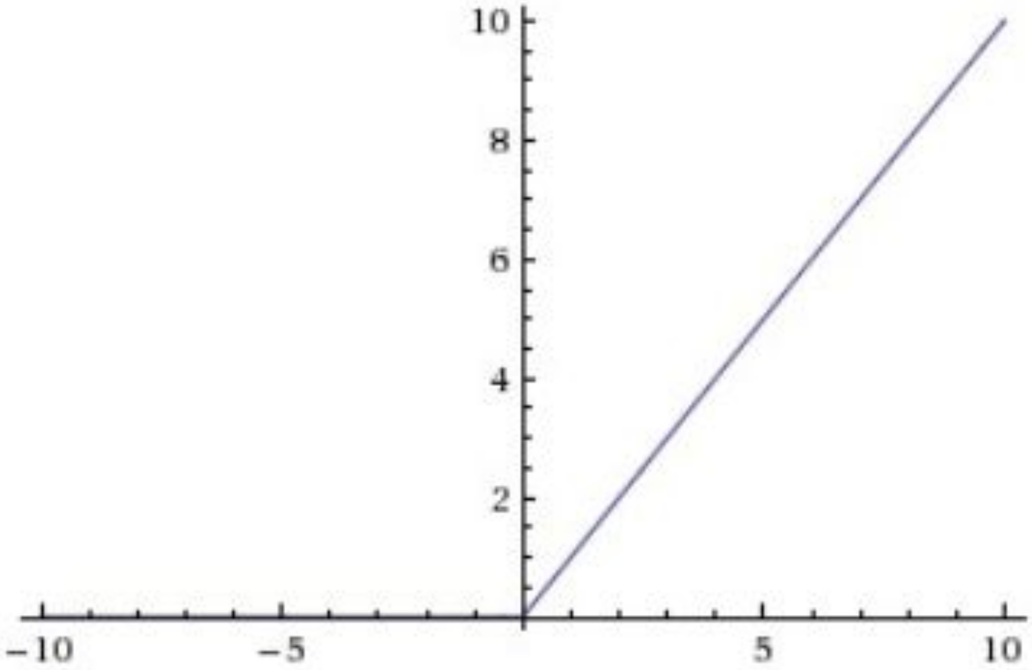


Hyperbolic tangent (tanh)

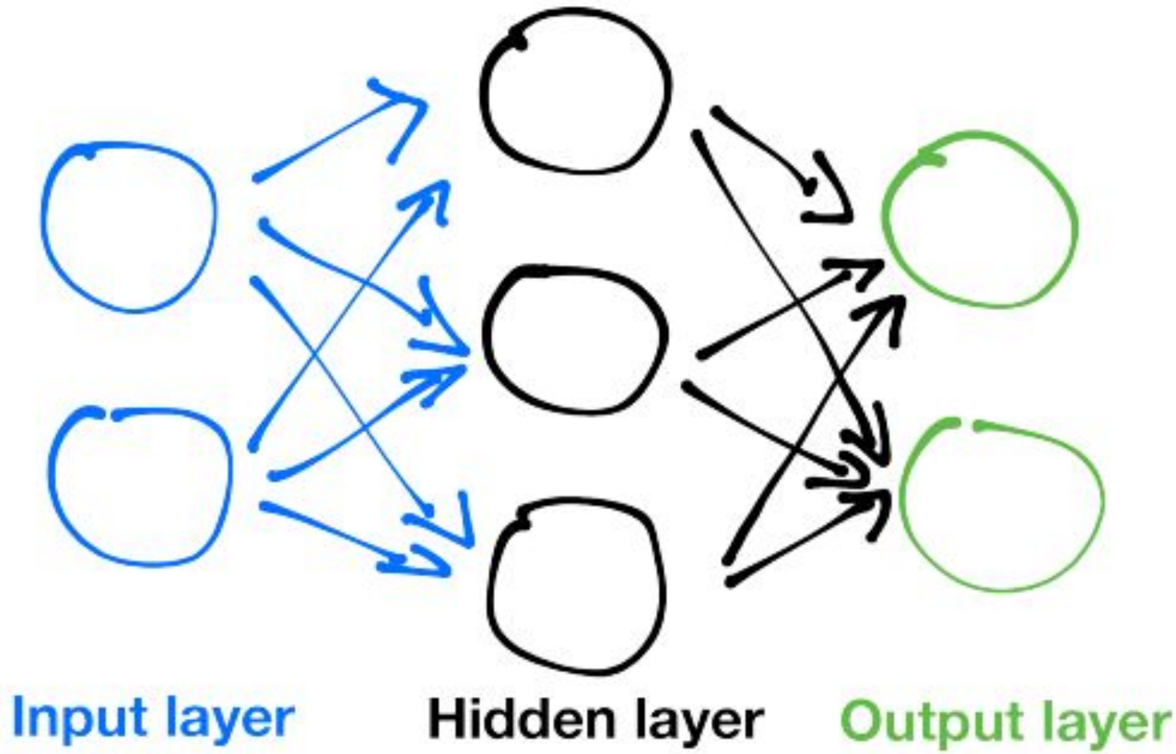


Rectified Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

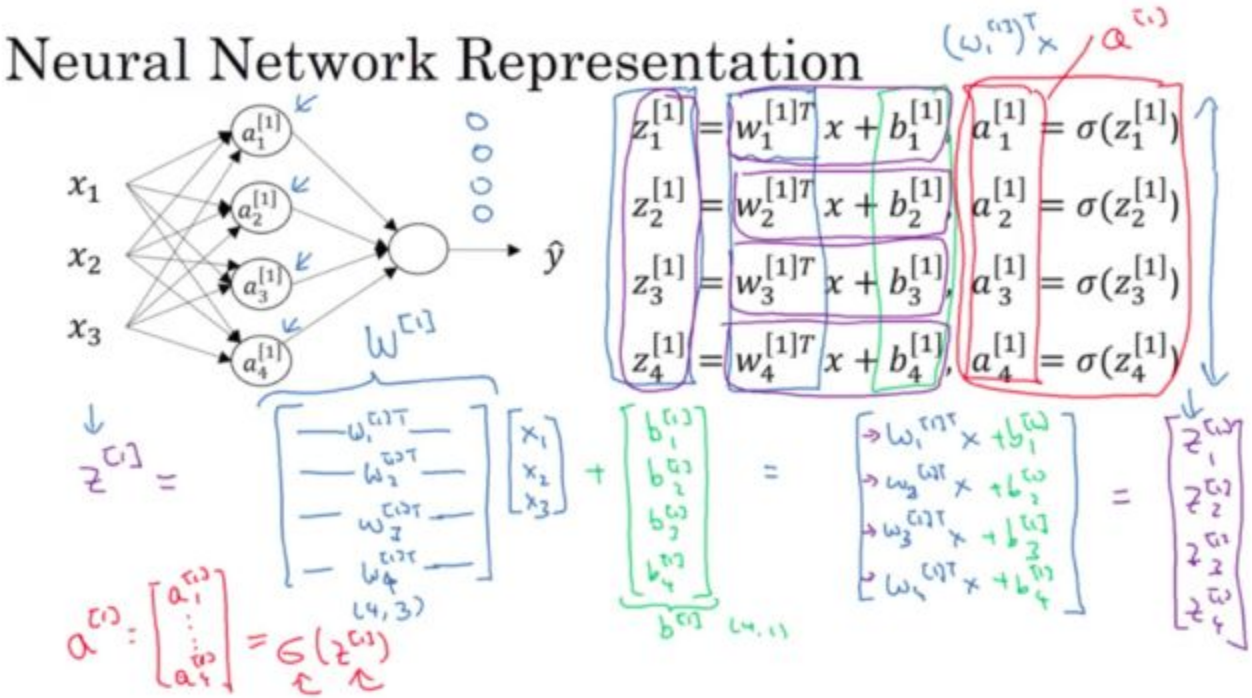


Neural Networks

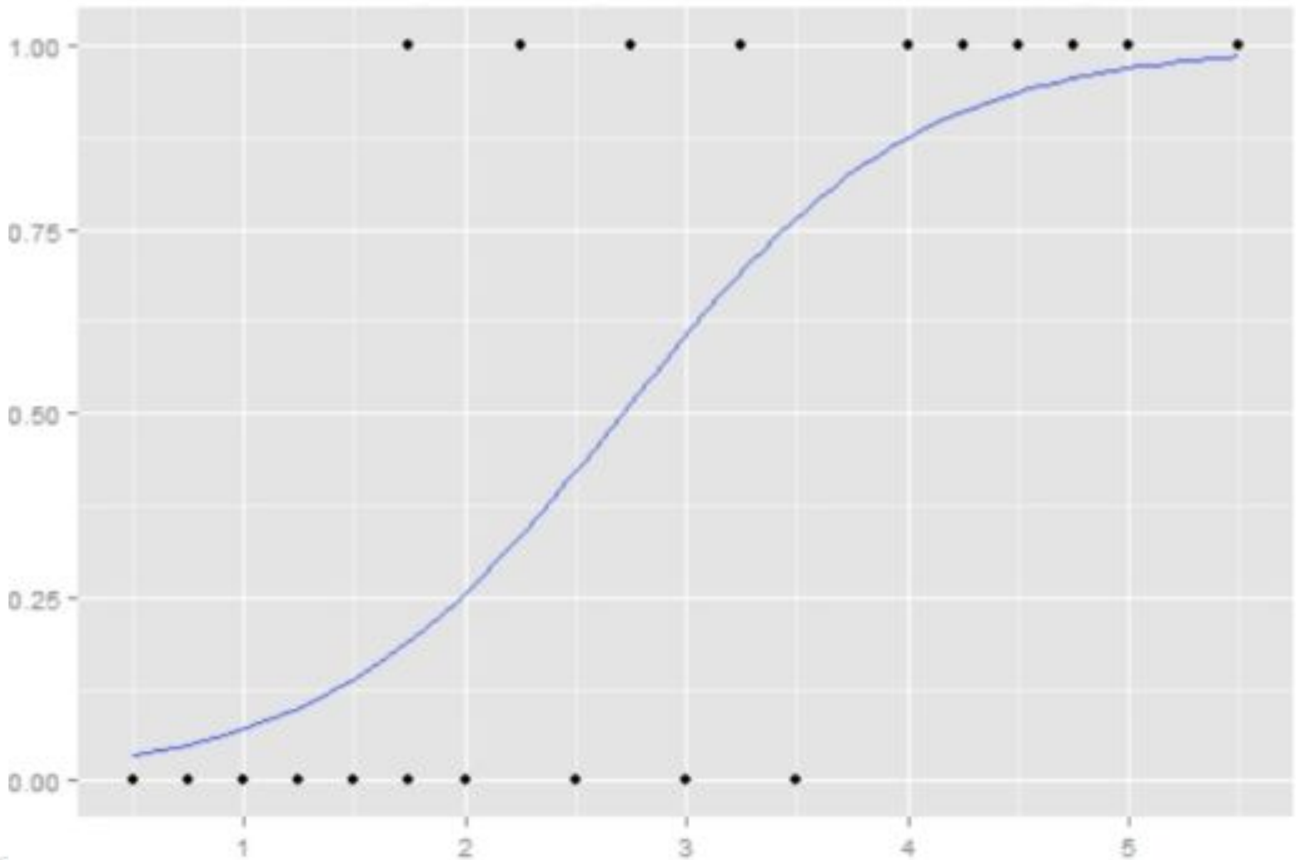


Input Layer: takes in input data (size corresponds to input space)
Hidden Layer: neurons hidden from view (this is where the magic happens)
Output Layer: neurons in this layer provide the output of the network

Neural Networks in Matrix Form



Neural Network Expressivity



Neural Network Expressivity

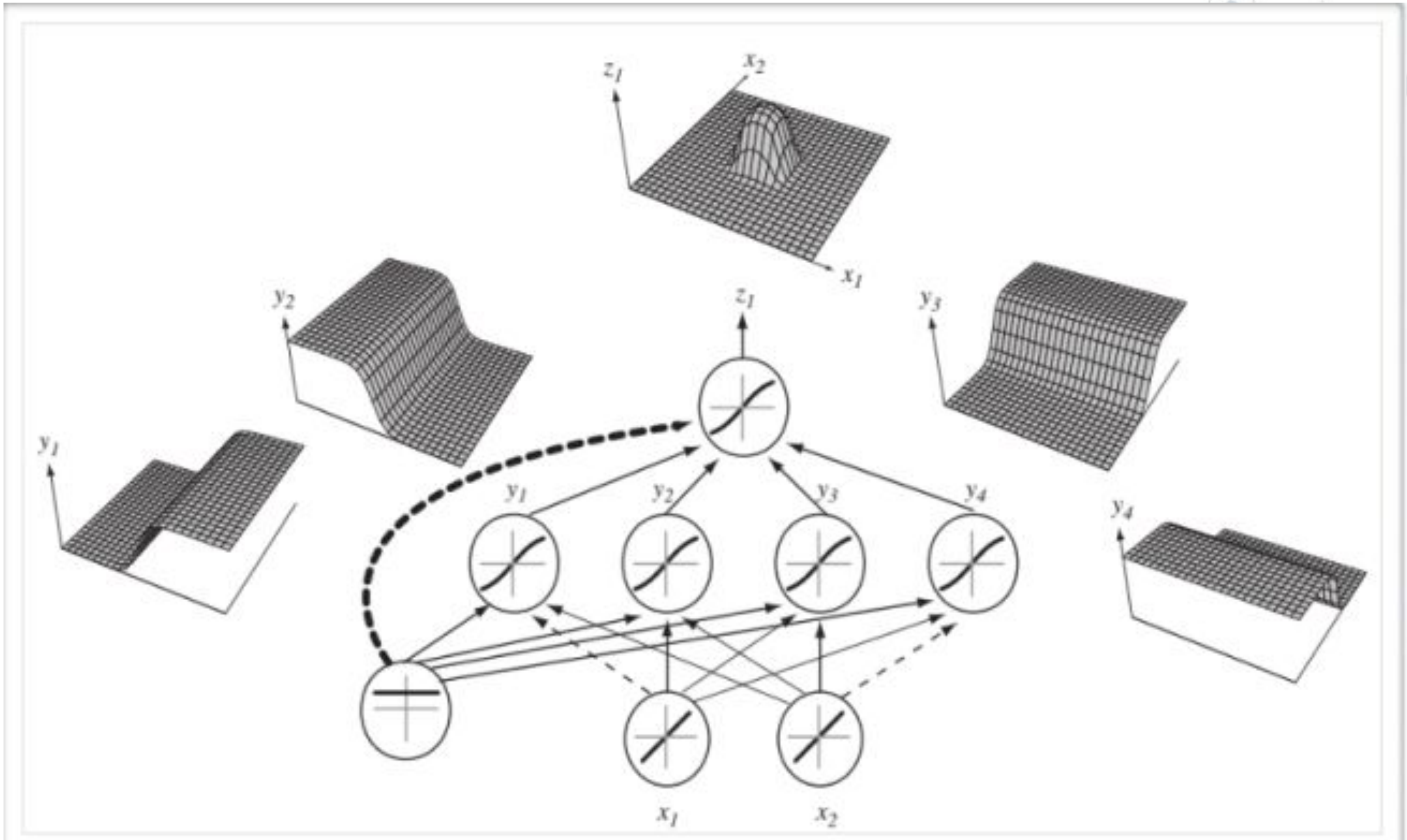


Image taken from Hugo Larochelle's Slides

Neural Network Expressiveness

- Neural networks approximate functions without assuming an initial data distribution
- NNs learn function approximations (anything that maps an input, to a single output)

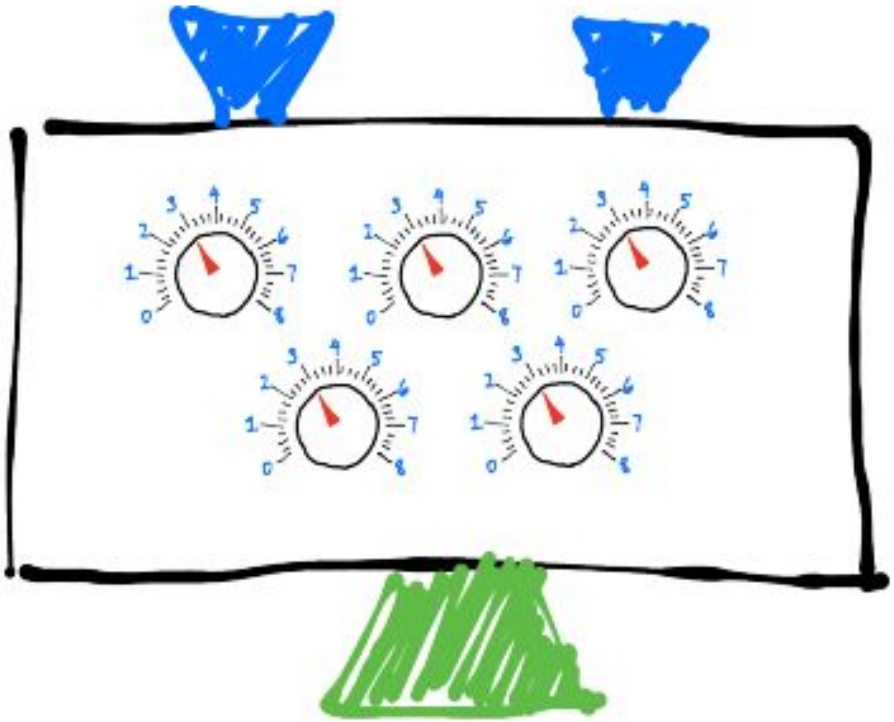
$$f : X \rightarrow Y$$

Universal Approximation Theorem

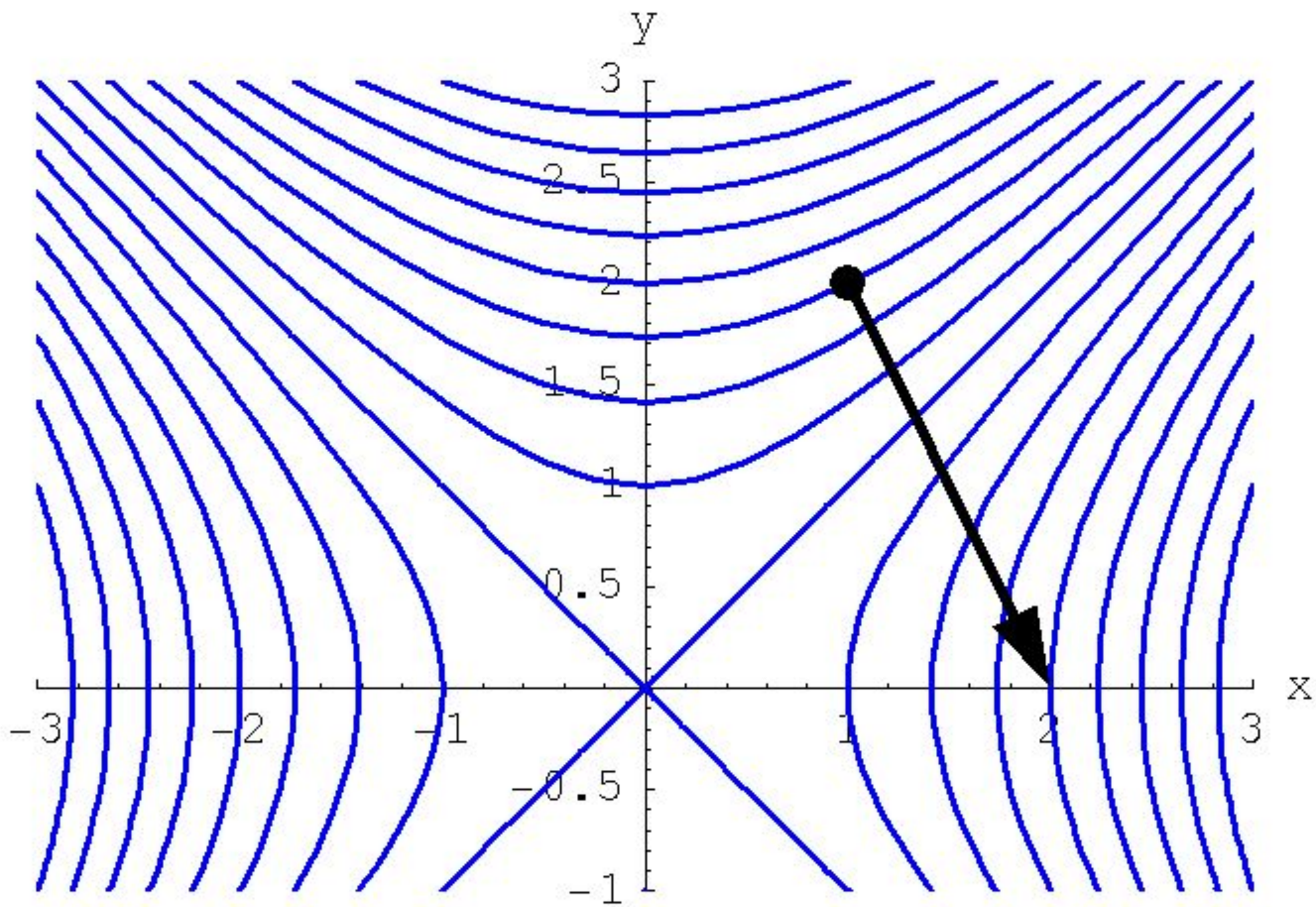
"Every bounded continuous function can be approximated with
arbitrary precision by a single-layer neural network"
(Hornik, 1991)

Image taken from Hugo Larochelle's Slides

Learning in Neural Networks



Optimization Via Gradient Descent



Gradient Descent

Define a differentiable, convex objective to minimize:

$$C(x, \text{parameters}) = \frac{1}{2}(y - \hat{f}(x))^2$$

Where:

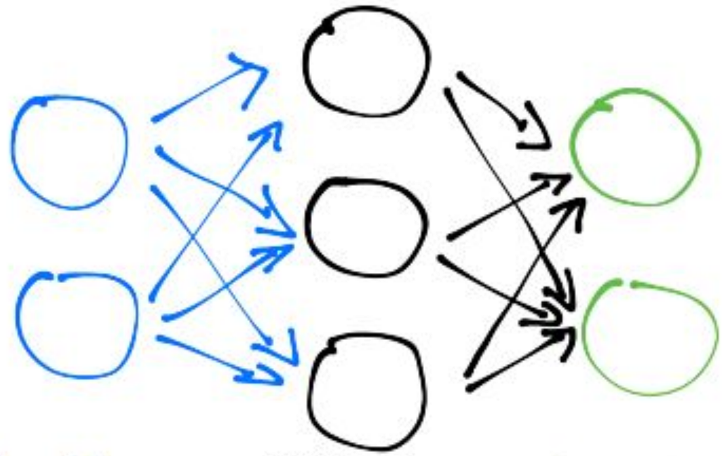
- y is the actual output
- f_hat is our hypothesis function (output of the network) dependent on our learned parameters and inputs

:

$$J(\mathbf{w}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2 = \frac{1}{2}(y - o_{N+H+1})^2$$

We use gradient descent to allow us to find a local minima of C given the derivative of C with respect to its parameters

Backpropagation



Neural network:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Stochastic Gradient Descent

- Initialize all weights to small random numbers.

Initialization

- Repeat until convergence:
  - Pick a training example.
  - Feed example through network to compute output $o = o_{N+H+1}$

Forward pass

  - For the output unit, compute the correction:

$$\delta_{N+H+1} \leftarrow o(1-o)(y-o)$$

  - For each hidden unit *h*, compute its share of the correction:

$$\delta_h \leftarrow o_h(1-o_h) w_{N+H+1,h} \delta_{N+H+1}$$

Backpro-pagation

  - Update each network weight:

$$w_{h,i} \leftarrow w_{h,i} + \alpha_{h,i} \delta_h x_{h,i}$$

Gradient descent

Image taken from Ryan Lowe's COMP551 Slides Lecture 14 Slide 17

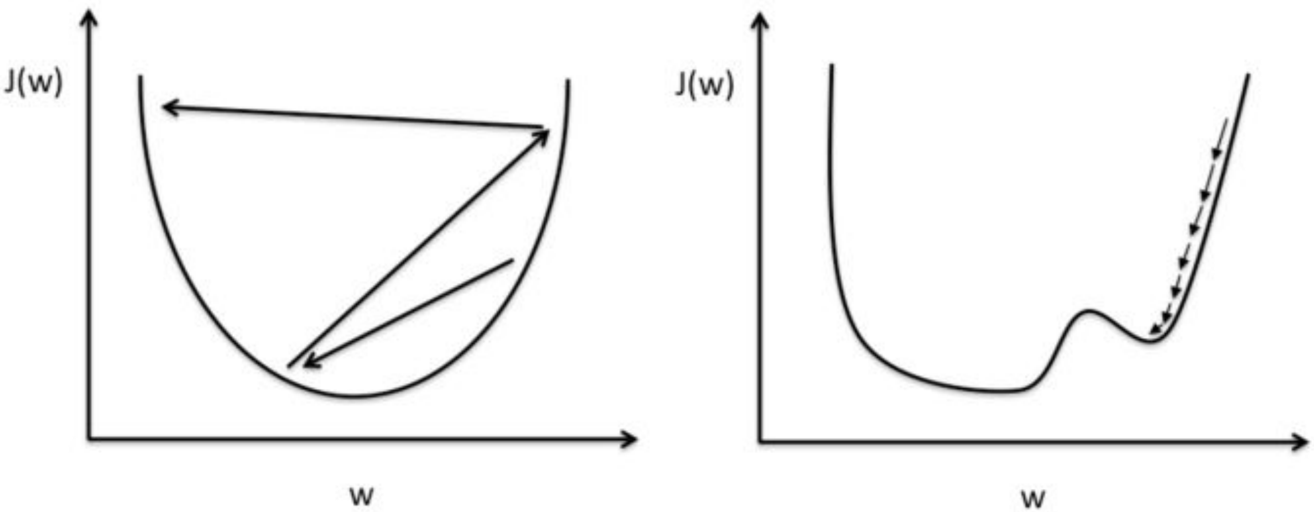
Forms of Gradient Descent

Stochastic Gradient Descent: compute error using a single sample at a time, update weights, repeat

Batch Gradient Descent: compute error on all examples, update weights based on error, repeat

Mini-batch Gradient Descent: randomly select a subset from the training data, calculate error on subset, update weights, repeat

Picking α - Adaptive Learning Rates



ADAM, RMSProp, Adagrad, etc.

[More Information on Stochastic Gradient Descent and Different Optimization Methods]

Coding Demo

Thanks!

Any questions?

Reminders:

Homework 2 due before next lecture.