ECE 693 – Special Topics:
AI for Radar System Design

Introduction to Deep Learning

Dr. Sevgi Zubeyde Gurbuz
szgurbuz@ua.edu

Feb. 11, 2022

THE UNIVERSITY OF ALABAMA®

BIOLOGICAL MOTIVATIONS

How Do Biological Systems Learn?

Study of neural computation inspired by the observation:

- Biological learning systems are built of very complex webs of interconnected neurons
- Each unit takes real-valued inputs (possibly from other units)
- Produces a single real valued output (which becomes the input to many other units)

A colored scanning electron micrograph (SEM) of a neuron (nerve cell).

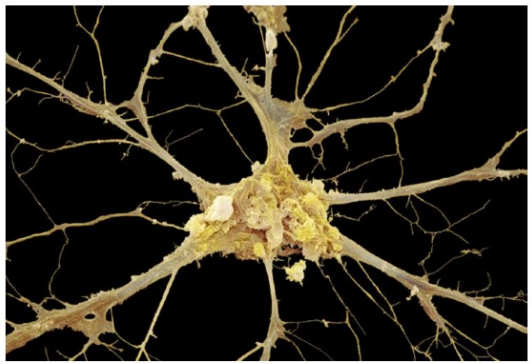
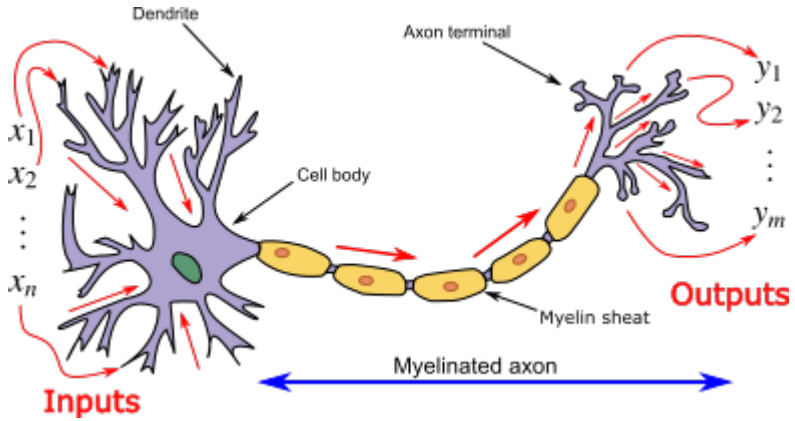


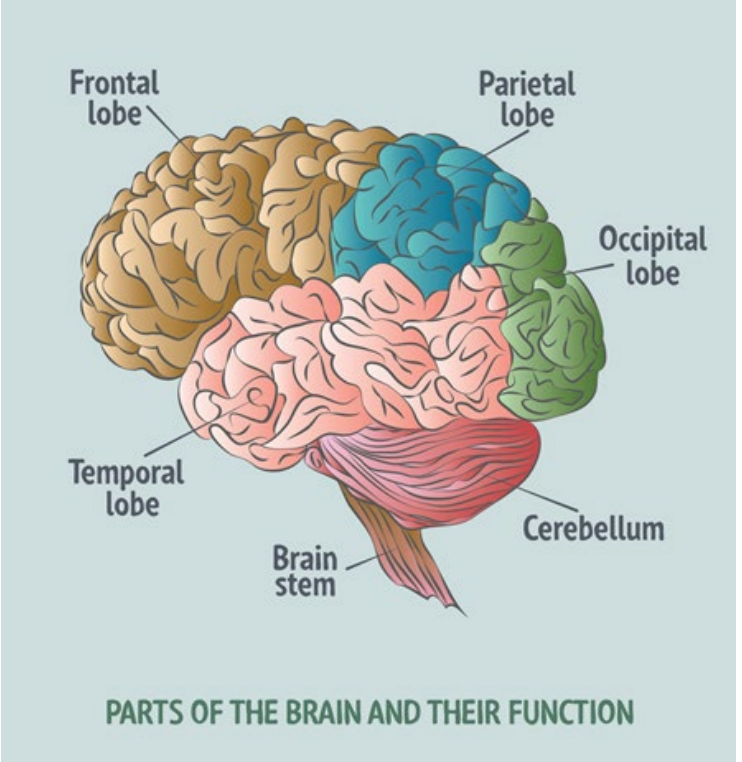
Image from S. Srihari



More than 10 billion neurons in human brain

The Human Brain

- Densely interconnected network of 10^{11} (100 billion) neurons
- Each connected to 10^{4} (10,000) others
- Fastest neuron switching time is 10^{-3} seconds
  - Slow compared to computer switching speed: 10^{-10} secs
- Activity excited or inhibited through connections to other neurons



Speed and Distributed Processing

- Humans can make certain decisions (visually recognize your mother) in 10^{-1} secs
- Implies that in 10^{-1} sec interval cannot possibly have more than a few hundred steps, given switch speed
- Highly parallel/distributed processing operations should exist

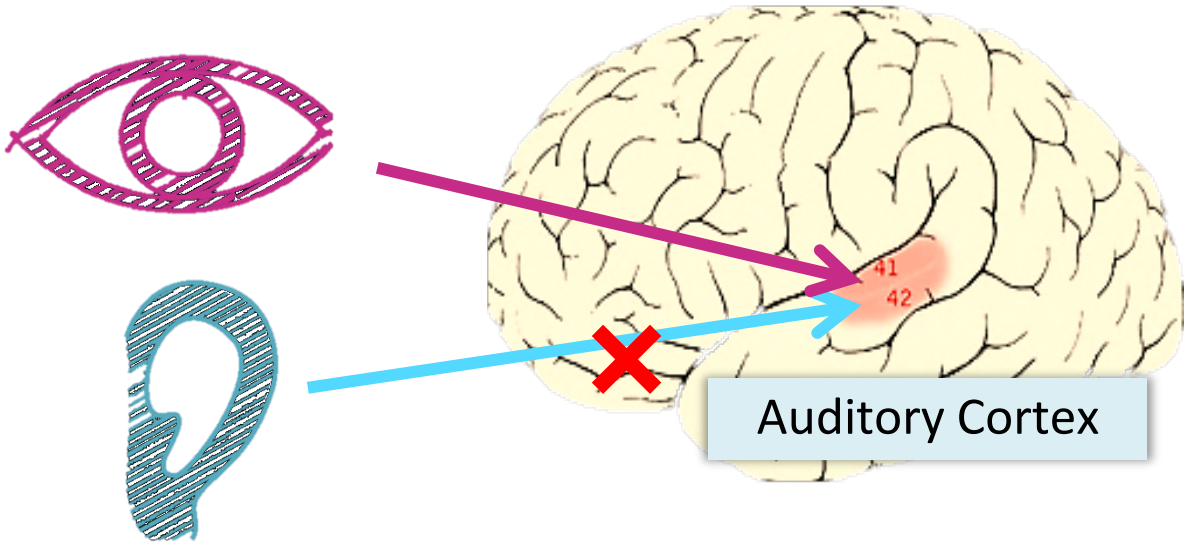
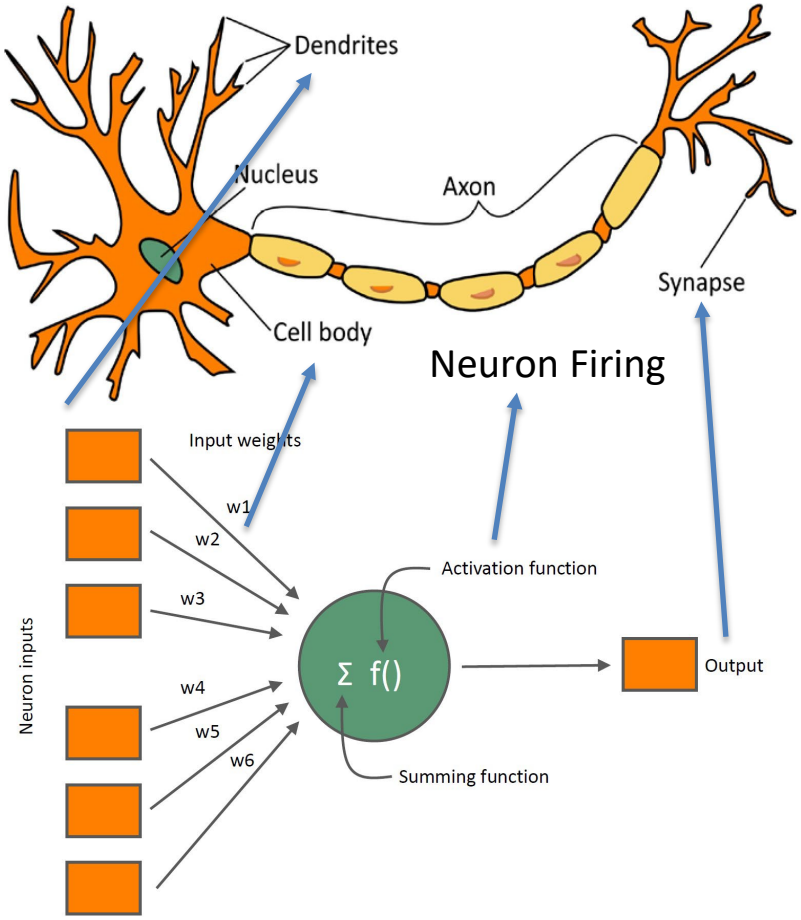


Figure by Andrew Ng.

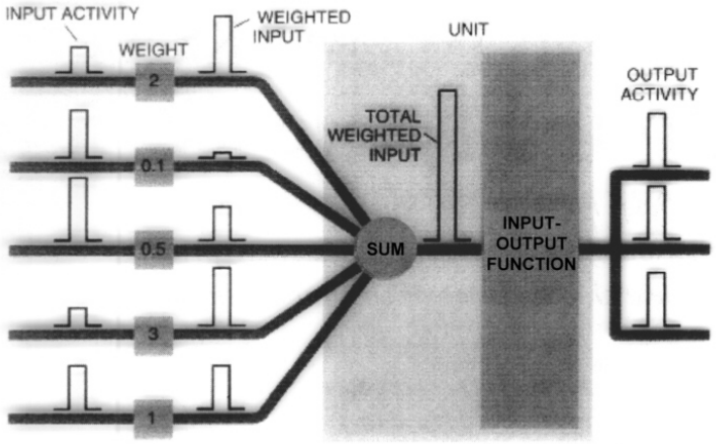
Seeing with your Tongue

Human Echolocation

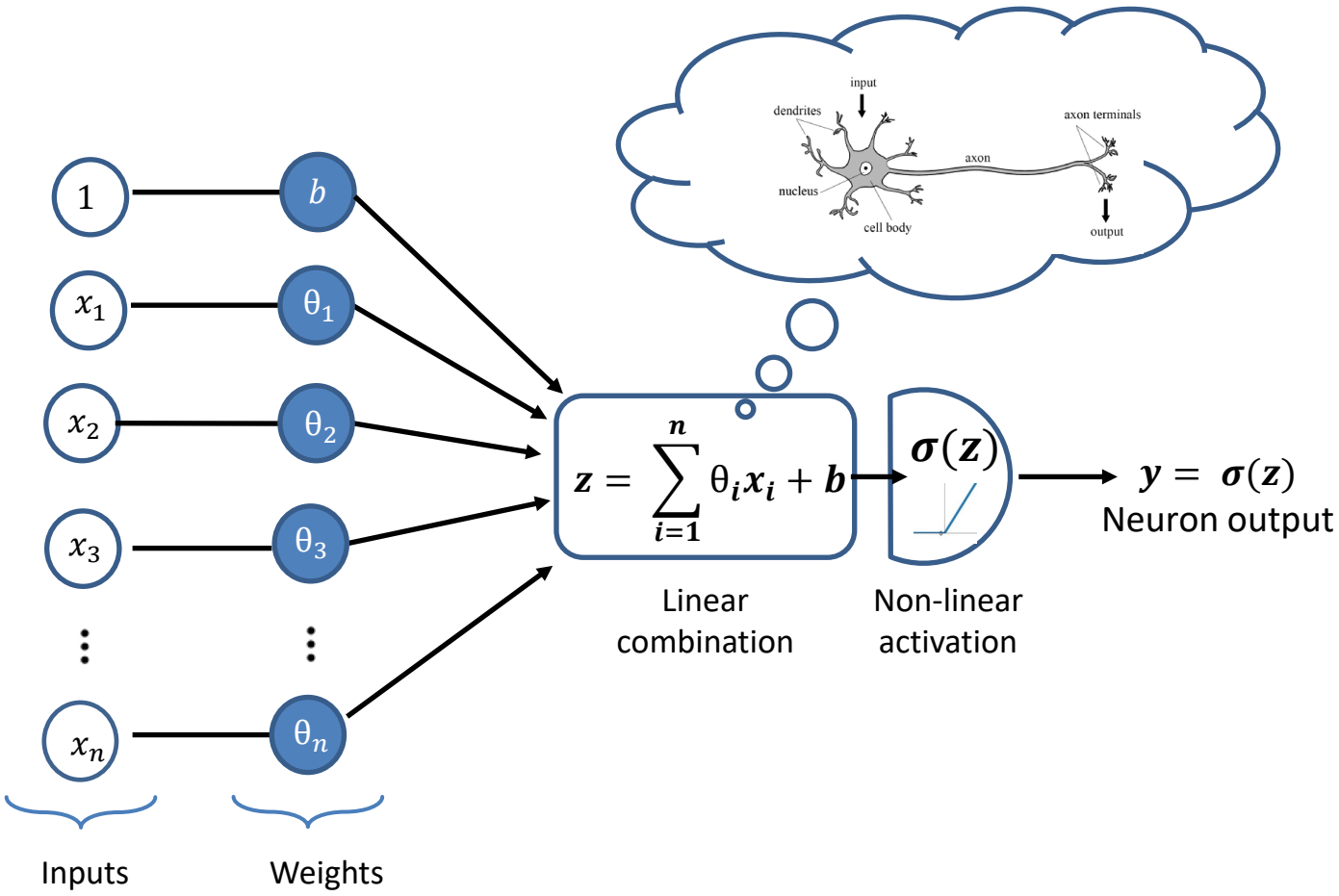
Our Hypothesis in NN



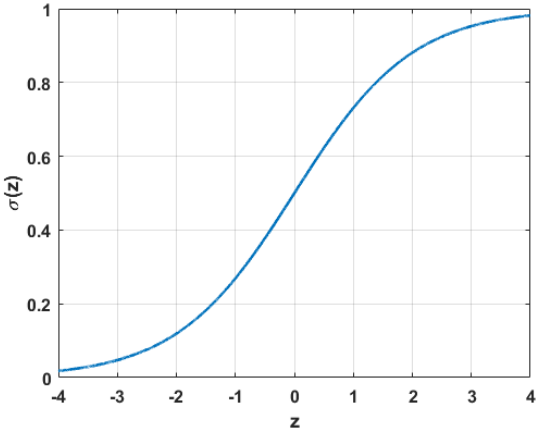
Idealization of A Neuron



Neural Network Structure



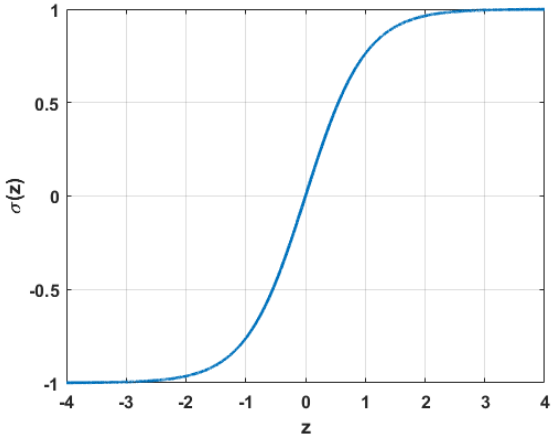
Activation Functions



(a)

Sigmoid

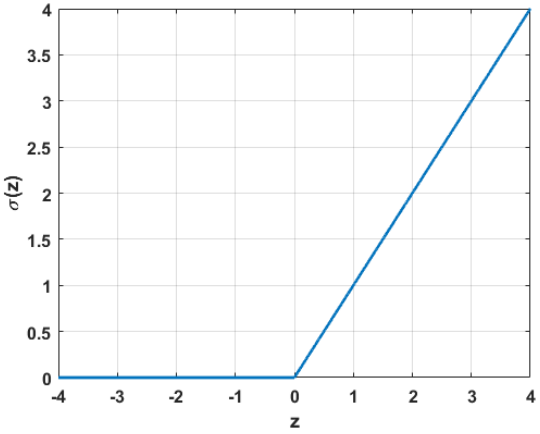
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



(b)

Tangent Hyperbolic

$$\sigma(z) = \tanh(z)$$



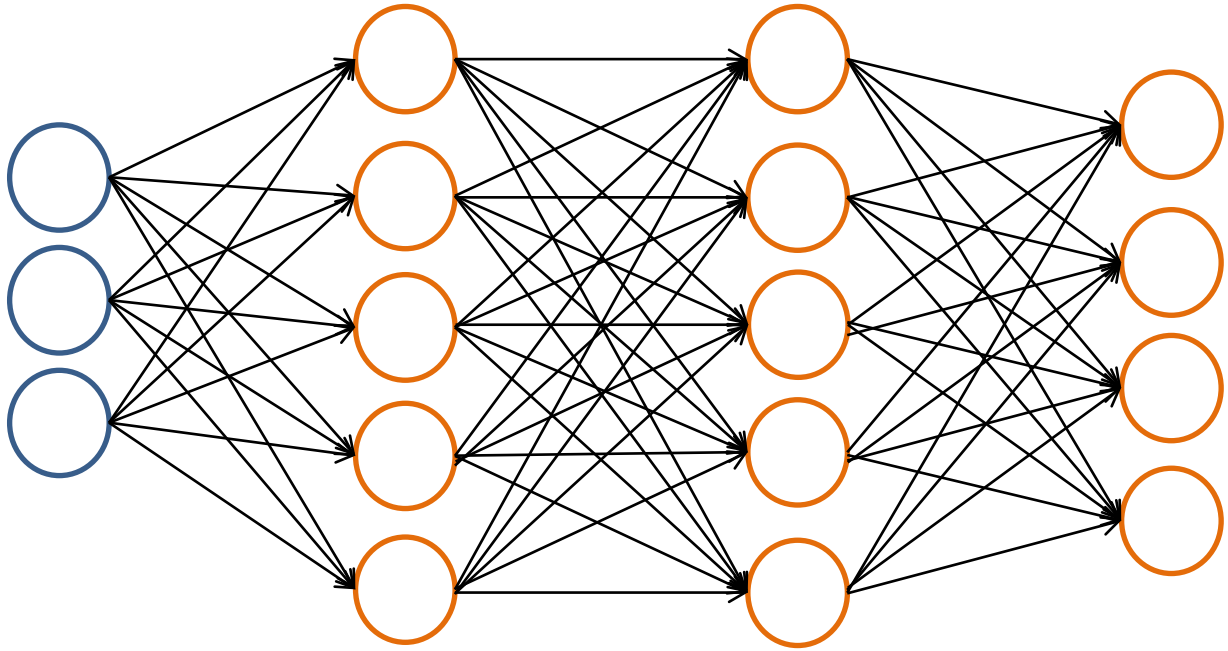
(c)

Rectified Linear Unit (ReLU)

$$\sigma(z) = \max(0, z)$$

DEEP NEURAL NETWORKS

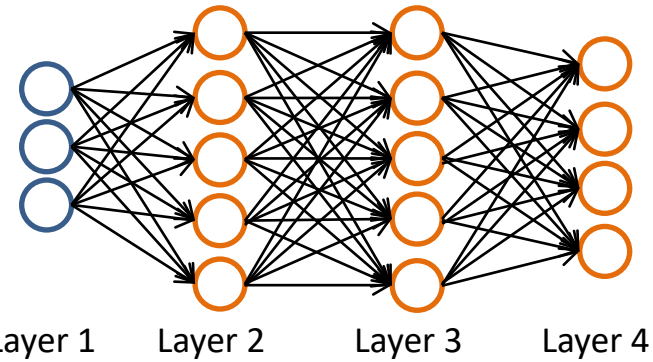
Artificial Neural Network

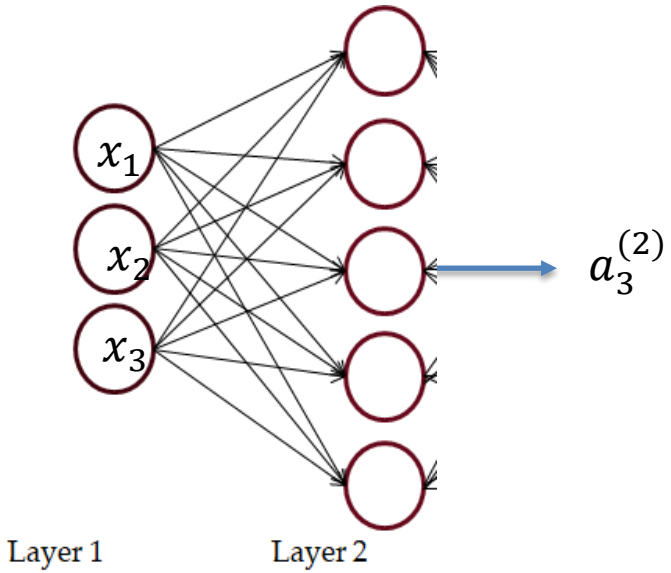


Feed Forward Neural Network Structure:

- The connections between the nodes do *not* form a cycle. No loops
- The information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes

First (Input) Layer





$a_i^{(j)}$ = "activation" of unit ***i*** in layer ***j***

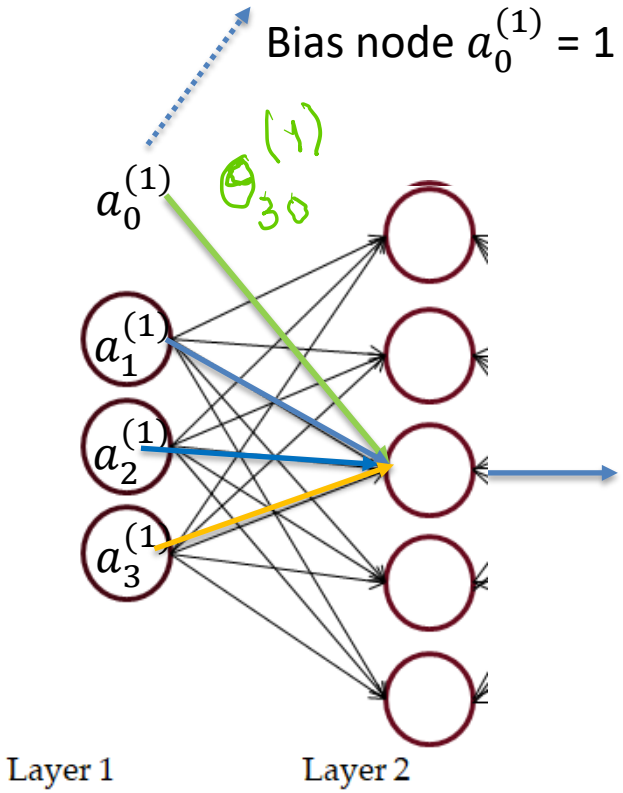
First layer (Input Layer):

- First layer activations are same as the inputs
- $a_1^{(1)} = x_1$
- $a_2^{(1)} = x_2$
- $a_3^{(1)} = x_3$

$$\boldsymbol{a^{(1)} = x}$$

$$\boldsymbol{a^{(1)}} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Propagate to 2nd Layer



Bias parameter

$$z_3^{(2)} = \theta_{30}^{(1)} a_0^{(1)} + \theta_{31}^{(1)} a_1^{(1)} + \theta_{32}^{(1)} a_2^{(1)} + \theta_{33}^{(1)} a_3^{(1)}$$

$$a_3^{(2)} = \sigma(z_3^{(2)})$$

$$a_3^{(2)} = \sigma(\theta_{30}^{(1)} a_0^{(1)} + \theta_{31}^{(1)} a_1^{(1)} + \theta_{32}^{(1)} a_2^{(1)} + \theta_{33}^{(1)} a_3^{(1)})$$

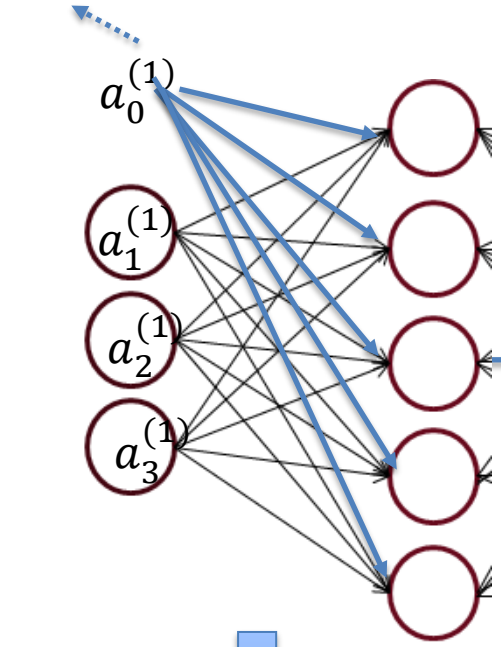
$$z_3^{(2)} = \begin{bmatrix} \theta_{30}^{(1)} & \theta_{31}^{(1)} & \theta_{32}^{(1)} & \theta_{33}^{(1)} \end{bmatrix} \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = (\boldsymbol{\theta}_{\mathbf{3},:}^{(\mathbf{1})})^T \boldsymbol{a}^{(\mathbf{1})}$$

$$a_3^{(2)} = \sigma(z_3^{(2)}) = \sigma((\boldsymbol{\theta}_{\mathbf{3},:}^{(\mathbf{1})})^T \boldsymbol{a}^{(\mathbf{1})})$$

Activations of 2nd Layer

Let's calculate all the activations in the next layer

Bias node $a_0^{(1)} = 1$



$$a_0^{(2)} = 1$$

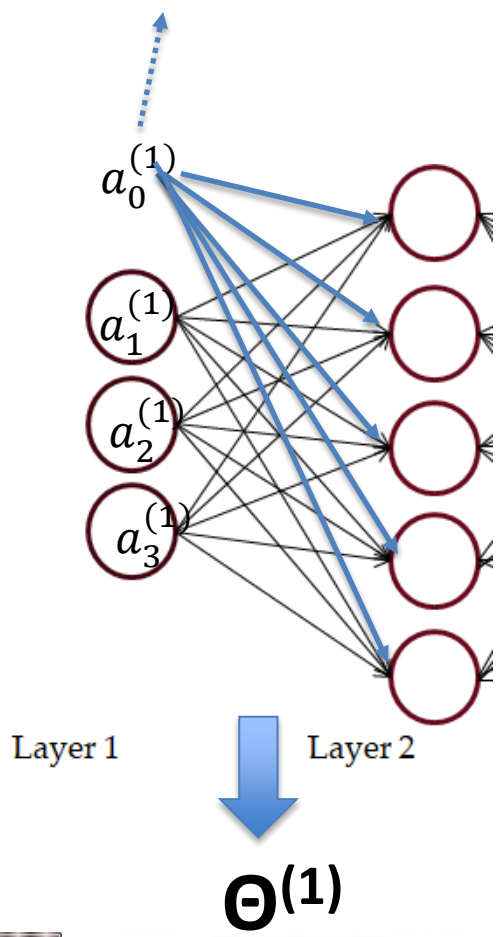
$$z_1^{(2)} = \theta_{10}^{(1)} a_0^{(1)} + \theta_{11}^{(1)} a_1^{(1)} + \theta_{12}^{(1)} a_2^{(1)} + \theta_{13}^{(1)} a_3^{(1)} \qquad a_1^{(2)} = \sigma(z_1^{(2)})$$

$$z_3^{(2)} = \theta_{30}^{(1)} a_0^{(1)} + \theta_{31}^{(1)} a_1^{(1)} + \theta_{32}^{(1)} a_2^{(1)} + \theta_{33}^{(1)} a_3^{(1)} \qquad a_3^{(2)} = \sigma(z_3^{(2)})$$

$$z_5^{(2)} = \theta_{50}^{(1)} a_0^{(1)} + \theta_{51}^{(1)} a_1^{(1)} + \theta_{52}^{(1)} a_2^{(1)} + \theta_{53}^{(1)} a_3^{(1)} \qquad a_5^{(2)} = \sigma(z_5^{(2)})$$

$\Theta^{(1)}$ $\quad z^{(2)}$ $\quad \Theta^{(1)} a^{(1)}$

Let's Rewrite in Matrix Form



$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer (j+1)

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} a_0^{(1)} = 1 \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} \qquad \boldsymbol{z}^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \\ z_4^{(2)} \\ z_5^{(2)} \end{bmatrix} = \boldsymbol{\Theta}^{(1)}\boldsymbol{a}^{(1)}$$

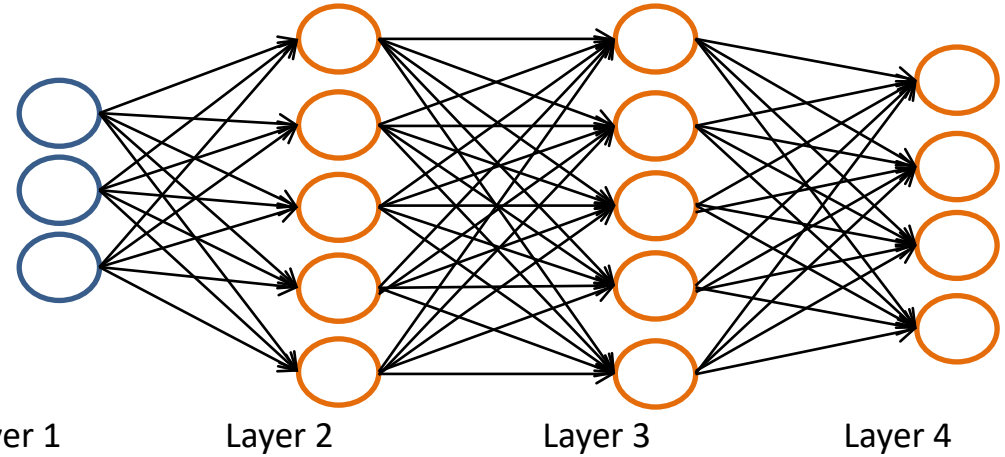
$$\boldsymbol{a}^{(2)} = \sigma(\boldsymbol{z}^{(2)})$$

↓ Elementwise

If the network has s_j units in layer j and s_{j+1} units in layer (j+1) then the size of $\Theta^{(j)}$ is **s_{j+1} x (s_j +1)**

$$\boldsymbol{\Theta}^{(1)} = \begin{bmatrix} \theta_{10}^{(1)} & \theta_{11}^{(1)} & \theta_{12}^{(1)} & \theta_{13}^{(1)} \\ \theta_{30}^{(1)} & \theta_{31}^{(1)} & \theta_{32}^{(1)} & \theta_{33}^{(1)} \\ \theta_{50}^{(1)} & \theta_{51}^{(1)} & \theta_{52}^{(1)} & \theta_{53}^{(1)} \end{bmatrix}$$

Number of Parameters in a DNN



$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer (j+1)

If the network has s_j units in layer j and s_{j+1} units in layer (j+1) then the size of $\Theta^{(j)}$ is s_{j+1} x (s_j +1)

$\Theta^{(1)}$ → 5 x 4 = 20

$\Theta^{(2)}$ → 5 x 6 = 30

$\Theta^{(3)}$ → 4 x 6 = 24

Total number of parameters:

20 + 30 + 24 = 74

Keep Propagating

Forward propagation:
For a given state of parameters $\Theta^{(1)}$, $\Theta^{(2)}$ and $\Theta^{(3)}$ we can calculate all the activations in each layer:

$$a^{(1)} = x$$

First layer activations are the input features $\mathbf{x}=\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = \sigma(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

Second Layer Activations

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = \sigma(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

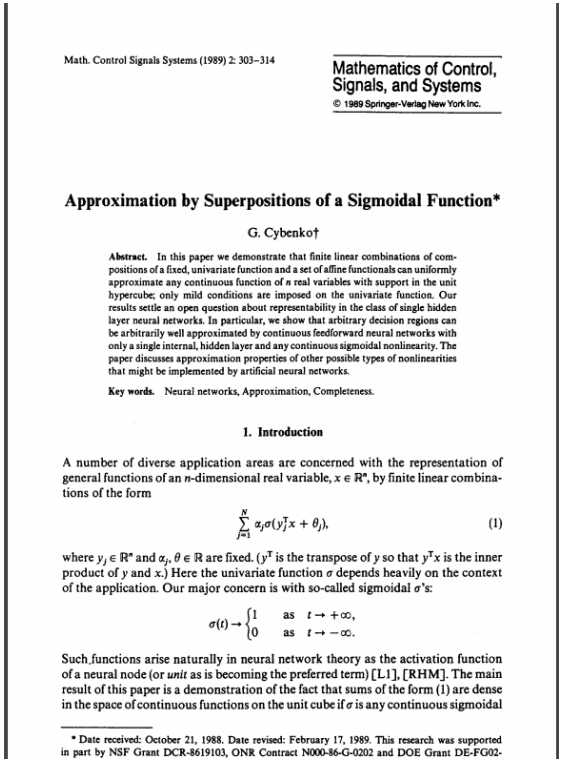
Third Layer Activations

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = \sigma(z^{(4)})$$

Output Layer Activations

What Can DNNs Represent?

- Two questions:
  - What is the representational power of this family of functions?
  - Are there functions that cannot be modeled with a Neural Network?



Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems
© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \qquad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-

Neural Networks with at least one hidden layer are *universal approximators*.

Given any continuous function g(x) and some $\epsilon > 0$, there exists a Neural Network $f_\Theta(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, |g(x) - f_\Theta(x)| < \epsilon$.

In other words, the neural network can approximate any continuous function.

This result holds even if the function has many inputs.

What if we go Deeper?

- If one hidden layer suffices to approximate any function, why use more layers and go deeper?

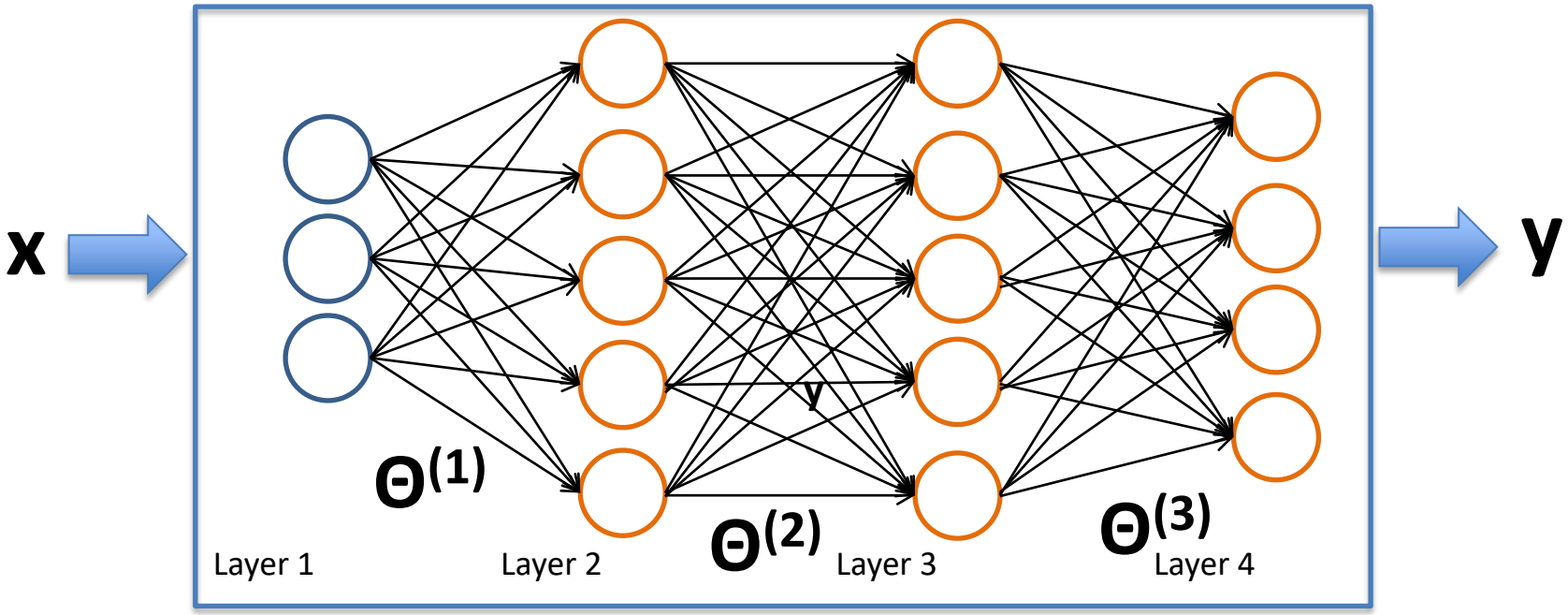
Two Caveats:

1) First, this doesn't mean that a network can be used to *exactly* compute any function. Rather, we can get an *approximation* that is as good as we want. By increasing the number of hidden neurons we can improve the approximation.

2) the class of functions which can be approximated in the way described are the *continuous* functions. If a function is discontinuous, i.e., makes sudden, sharp jumps, then it won't in general be possible to approximate using a neural net.

a more precise statement of the universality theorem is that neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision.

Cost Functions for Training



Θ

$y = f_\Theta(x)$

Training data:
$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$,

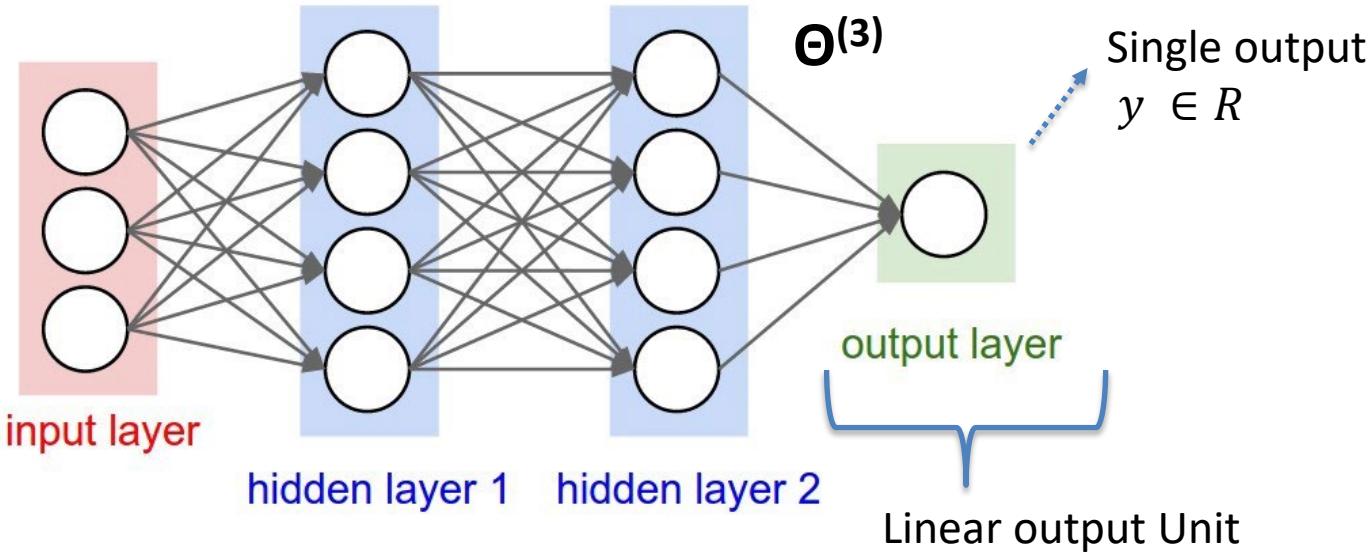
Need a cost function to say how well the approximation for given parameters Θ

How to Define Cost?

- It depends on the application and NN structure
  - Regression
  - Binary classification (Logistic Activation)
  - Multi-class classification (K-binary) (Logistic Activation)
  - Standard Multi-class (Softmax Activation)

Each case has a different cost function
(again basically depending on the probabilistic view of the data and Maximum likelihood estimation of parameters)

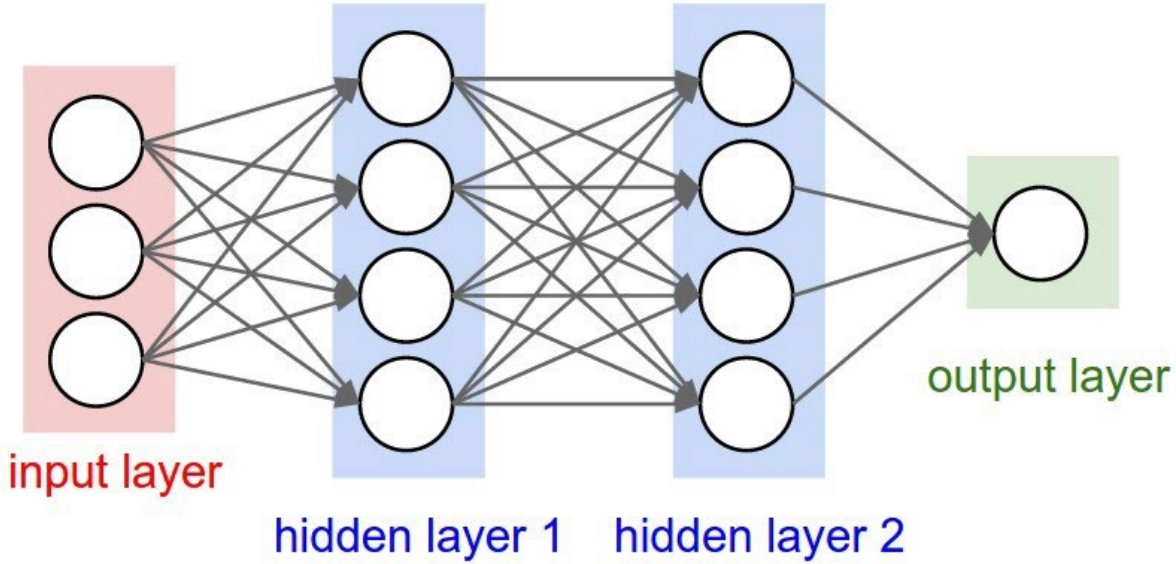
Example: Cost in Regression



$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{M}(f(x^{(i)};\Theta) - y^{(i)})^2$$

Assuming $y^{(i)}$ are independent and Gaussian distributed with mean $f(x^{(i)};\Theta)$. This cost is the same obtained in Maximum Likelihood Estimation of Θ

Why? Take Derivative at Output

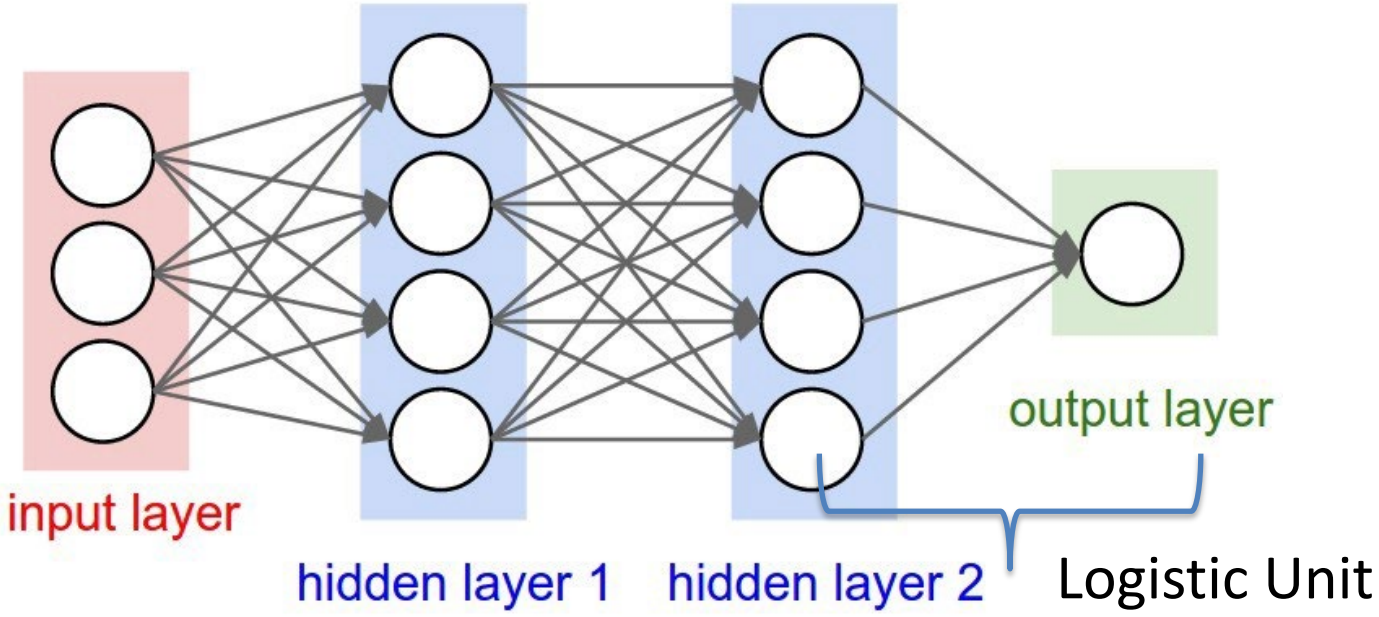


Derivative at the output node will be back-propagated to update Θ. Hence it is important.

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{M}(z^{(3,i)} - y^{(i)})^2 \Rightarrow \frac{J(\Theta)}{dz^{(i)}} = (z^{(3,i)} - y^{(i)})$$

The error ($f\left(x^{(i)};\Theta\right) - y^{(i)}$) for each data sample is actually the derivative at the output node wrt z

Another Example: Binary Classification



This time output is either y=0 or y=1 (Binary!)

The output unit is a logistic unit

$$\vec{z} = \theta^T \vec{x} \rightarrow y = \sigma(\vec{z}) = \frac{1}{1-e^{-z}}$$

Cost Function for Binary Classification

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log f\left(x^{(i)};\Theta\right) + \left(1-y^{(i)}\right)\log\left(1-f\left(x^{(i)};\Theta\right)\right)\right]$$

Derivative at the output node: $\frac{J(\Theta)}{dz^{(i)}} = f\left(x^{(i)};\Theta\right) - y^{(i)}$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h^{(i)} + \left(1-y^{(i)}\right)\log\left(1-h^{(i)}\right)\right] \qquad h^{(i)} = \frac{1}{1+e^{-z^{(i)}}}$$

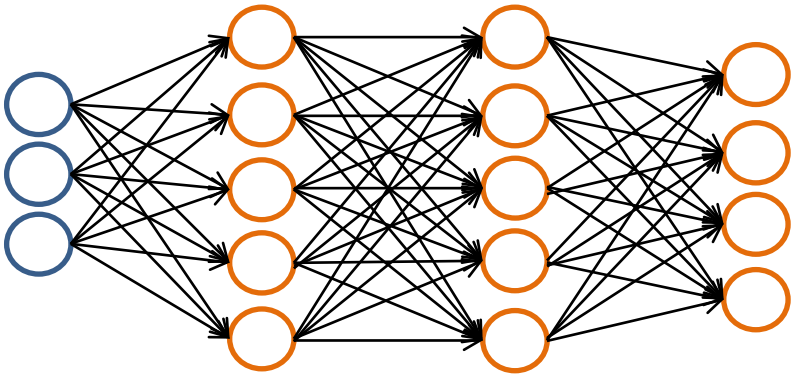
$$\frac{J(\Theta)}{dh^{(i)}} = \frac{-y^{(i)}}{h^{(i)}} + \frac{1-y^{(i)}}{1-h^{(i)}} = \frac{h^{(i)}-y^{(i)}}{h^{(i)}(1-h^{(i)})}$$

$$\frac{dh^{(i)}}{dz^{(i)}} = h^{(i)}(1-h^{(i)})$$

$$\frac{J(\Theta)}{dz^{(i)}} = h^{(i)} - y^{(i)}$$

(Same as for linear regression)

Standard Multi-Class Classification



Output Layer is a **softmax layer**

Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

$$h_j^{(i)} = \frac{e^{z_j^{(i)}}}{\sum_k^K z_k^{(i)}}$$

softmax activation

k = 1 to K number of classes

Network outputs are interpreted as probabilities

$$p(y = j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T \mathbf{x} + b_j)}}{\sum_{k \in K} e^{(\mathbf{w}_k^T \mathbf{x} + b_k)}}$$

Cost for Standard Multiclass

$$J(\Theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \quad \log h_k^{(i)}\right]$$

Derivation is a little longer but for softmax case the derivative wrt the activation and the output layer is similar: just the error

$$\frac{J(\Theta)}{dz_k^{(i)}} = h_k^{(i)} - y_k^{(i)}$$

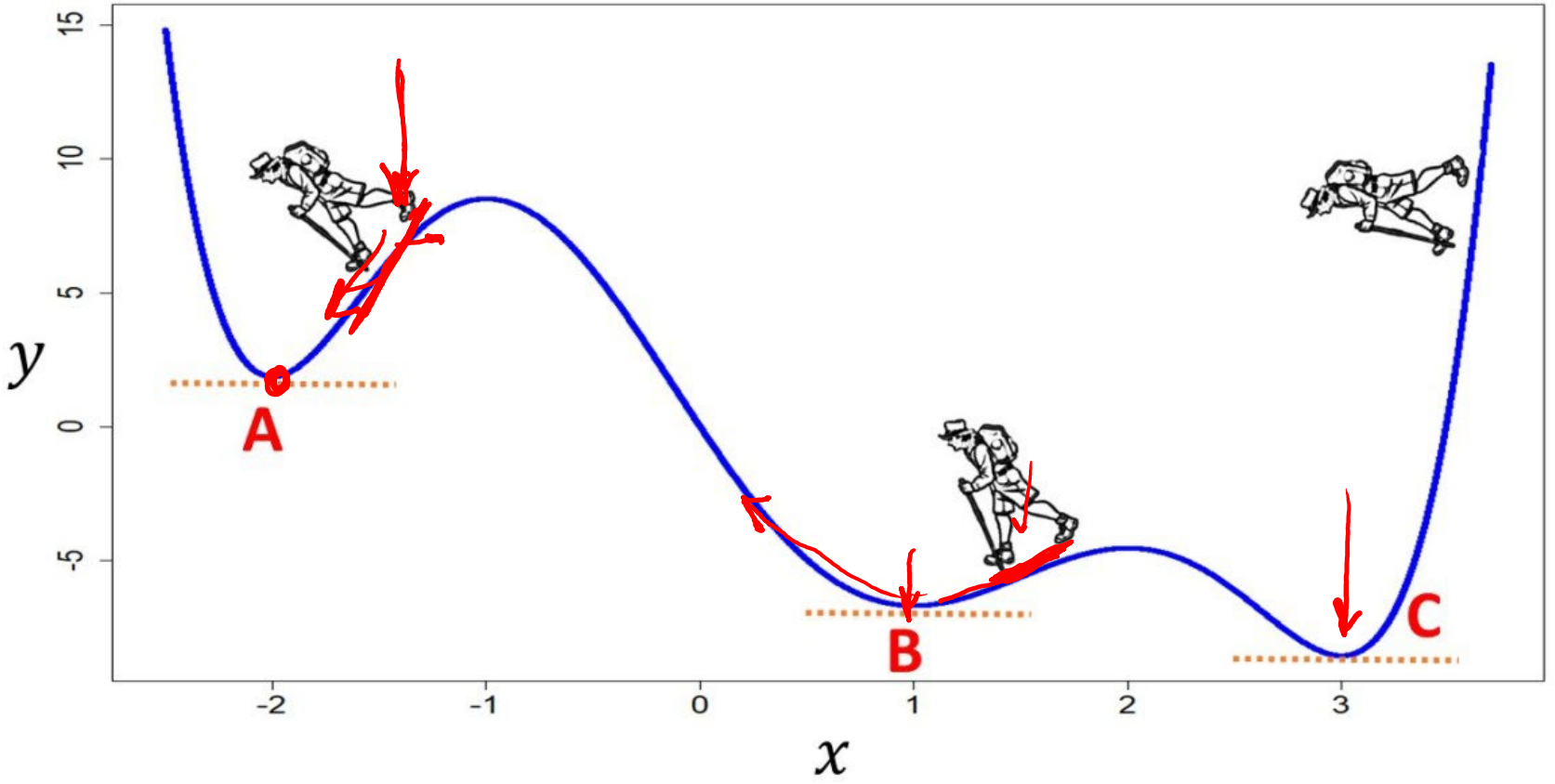
Regularized Cost:

$$J_R(\Theta) = J(\Theta) + \frac{\lambda}{2m}\sum(\Theta_{ij}^{l})^2$$

MATHEMATICS OF TRAINING:

1. Gradient Descent
2. Backpropagation

Gradient Descent: How to Find Maxima/Minima



Gradient Descent

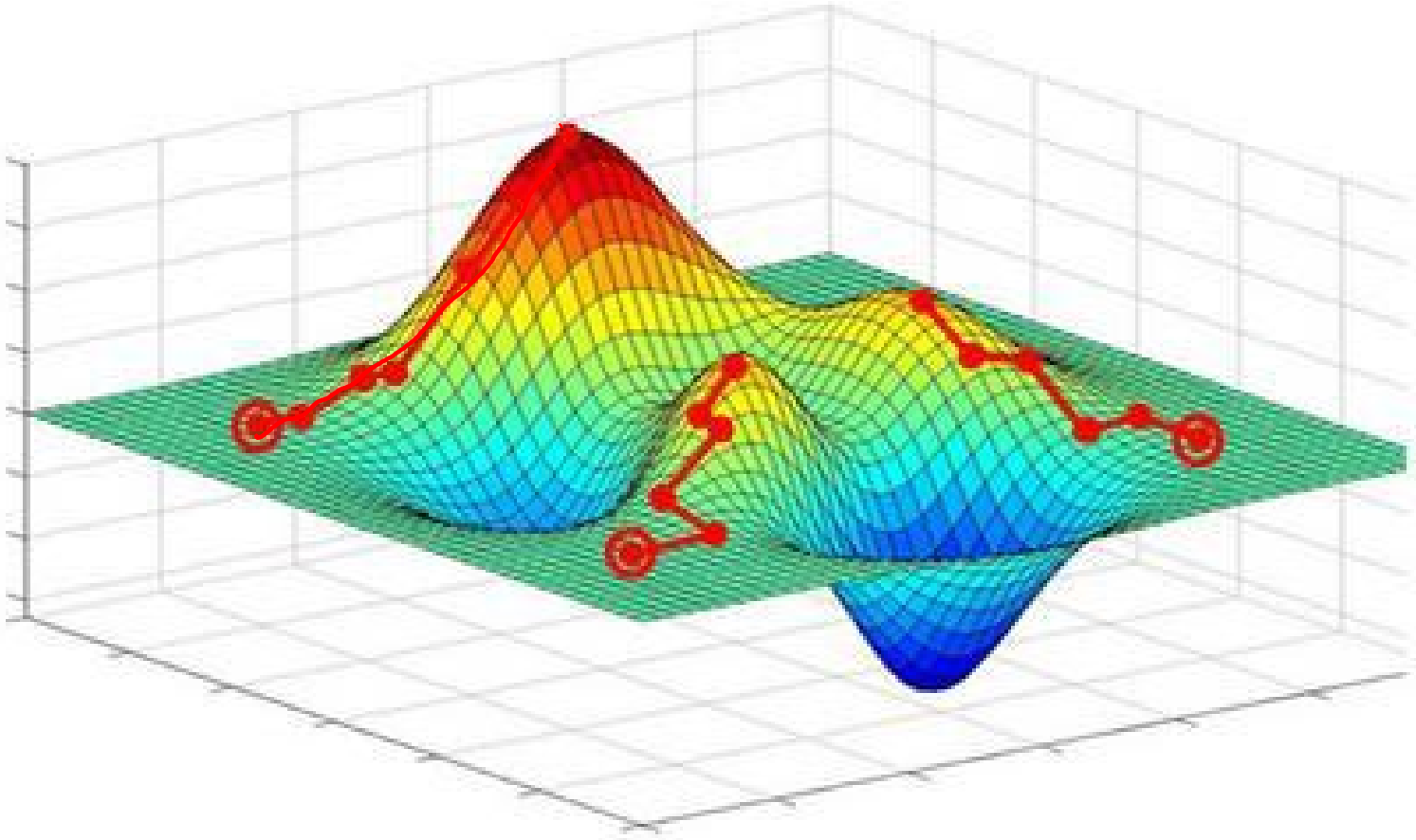
- Want to minimize $J(\theta_0, \theta_1)$ over parameters θ_0, θ_1

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

Outline:

- Start with some θ_0, θ_1 (could be arbitrary)
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minimum

GD in Multiple dimensions



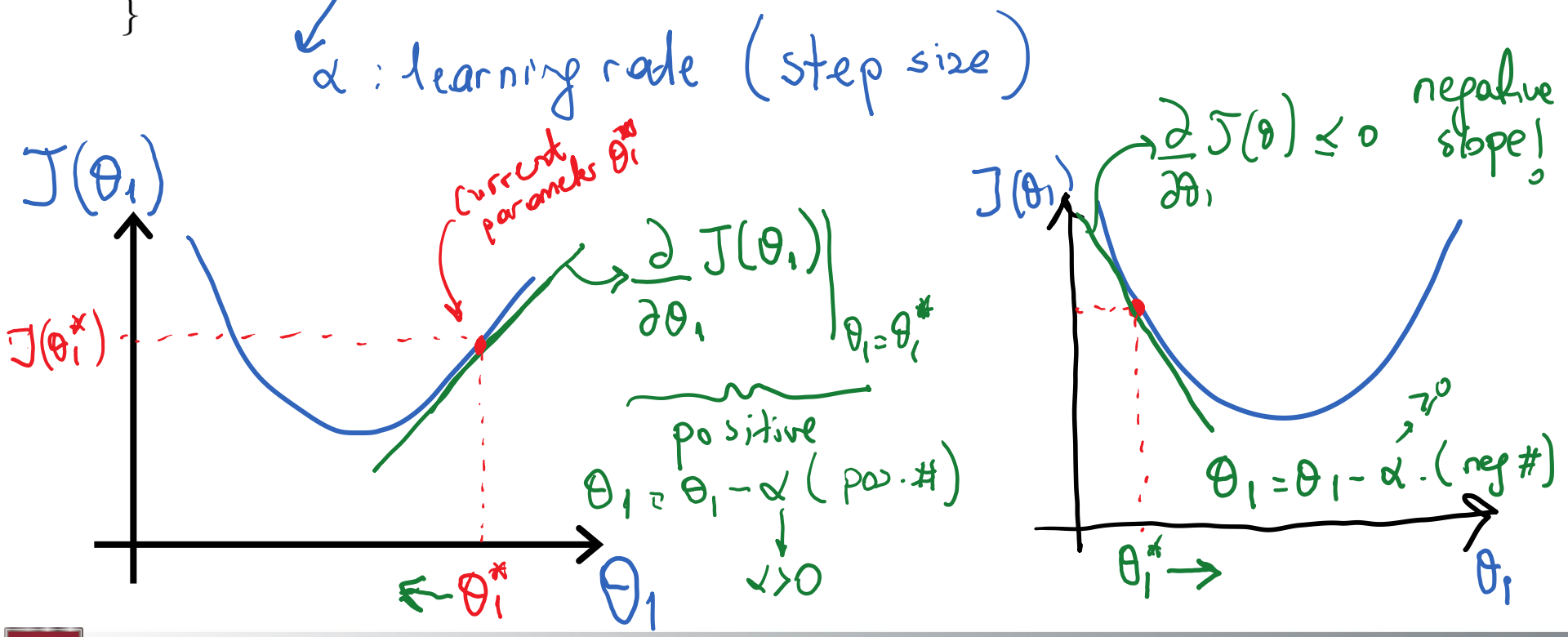
Simply Gradient Descent

Start with arbitrary θ_0, θ_1

repeat until convergence {

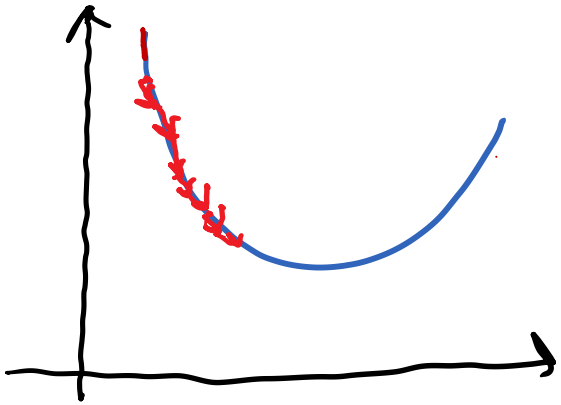
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

}



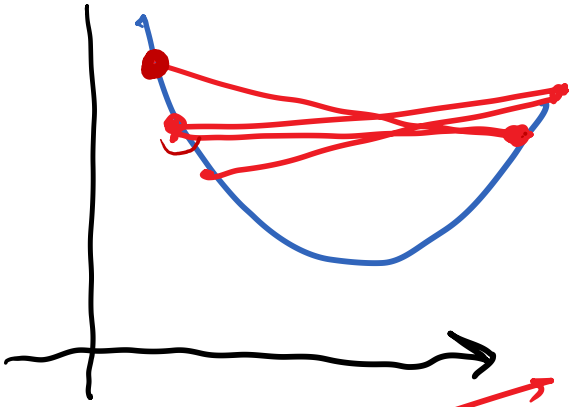
Effect of Learning Rate (Step Size)

Learning Rate α small:

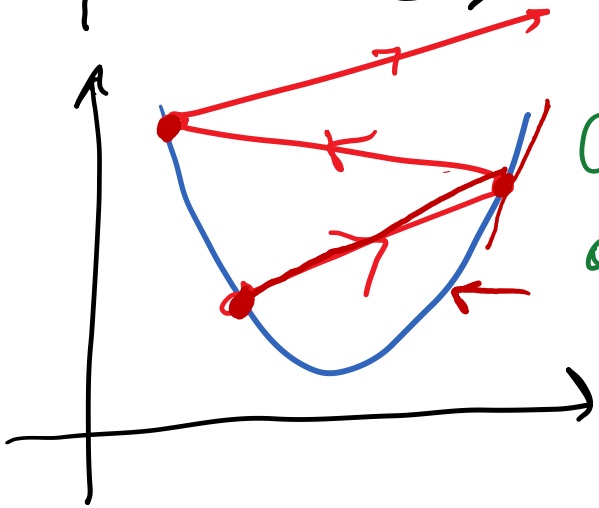


- Too many iterations to find the minimum
- α too small ⇒ GD is slow!

Learning Rate α too large:



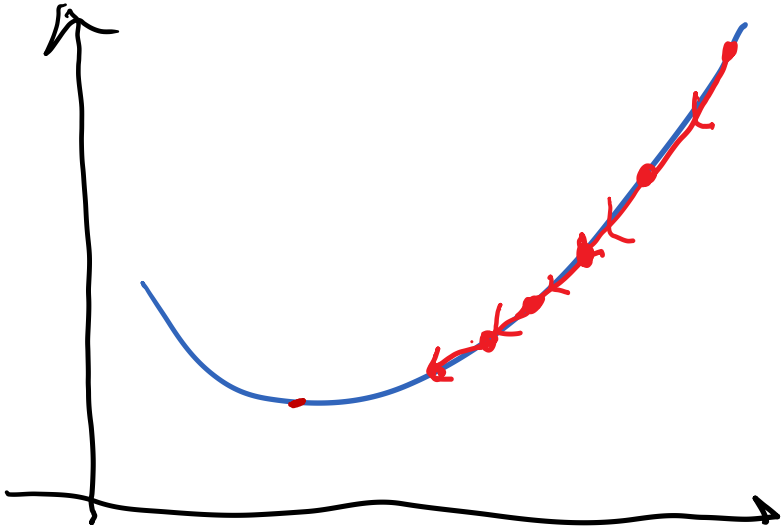
may not converge at all.



Could also diverge

Effect of Learning Rate (Step Size)

Learning Rate α fixed or variable



Since gradient gets smaller update on the parameter gets smaller even using a fixed α.

Other considerations

- Might decrease α over time.

  $\alpha_{t+1} = 0.9\,\alpha_t$?

- Different α on θ_1 or θ_2?

In general a fixed suitable α will work. But it is still an additional parameter we need to select.

How to choose α

- For sufficiently small α , J(θ) should decrease at every iteration
- Very small α converges slowly
- Very large α might not converge
- Test different α:
  - start small, then gradually increase

When to stop?

- Declare that the algorithm has converged when the cost $J(\theta)$ decreases by less than a predefined error in one iteration

Summary of Backpropagation

1. Define a training set $\mathcal{T}$, which consists of N training samples with the corresponding desired outputs

$$\mathcal{T} = \{\mathbf{x}(n), \mathbf{d}(n); n = 1, 2, \ldots, N\}.$$

2. Random initialization of the network weights $w_{ji}^{(l)}$.

   Set $y_i^{(0)} = x_i, n = 1, y_0^{(l-1)} = 1, l = 1, 2, \ldots, L.$

3. Forward propagation of training sample $\mathbf{x}(n)$

$$v_j^{(l)} = \sum_{i=0}^{m_{l-1}} w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

$$y_j^{(l)}(n) = \varphi_j^{(l)}\left(v_j^{(l)}\right).$$

Backpropagation (cont'd)

4. Calculate the local gradient for all neurons. Equations (3.20) and (3.25).

$$\delta_j^{(l)}(n) = \begin{cases} \frac{\partial \mathscr{E}}{\partial y_j^{(L)}} \cdot \varphi_j'^{(L)}(v_j^{(L)}(n)) & \text{if} \quad l = L \\ \varphi_j'^{(l)}(v_j^{(l)}(n)) \sum_{k=1}^{m_{l+1}} \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{else} \end{cases}.$$

5. Compute gradient vector of weights according to (3.28)

$$\frac{\partial \mathscr{E}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} \cdot y_i^{(l-1)}$$

6. Weight update according to (3.16)

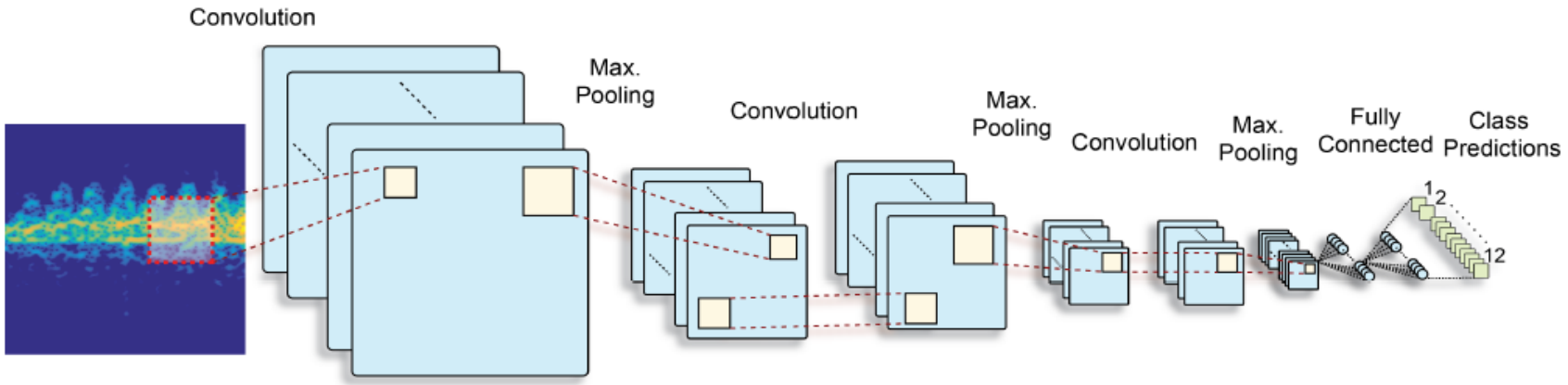
$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) - \eta \frac{\partial \mathscr{E}}{\partial w_{ji}^{(l)}}.$$

7. Iterate $n \rightarrow n+1$ and repeat steps 3–6. until stopping critarion is met.

DNN ARCHITECTURES

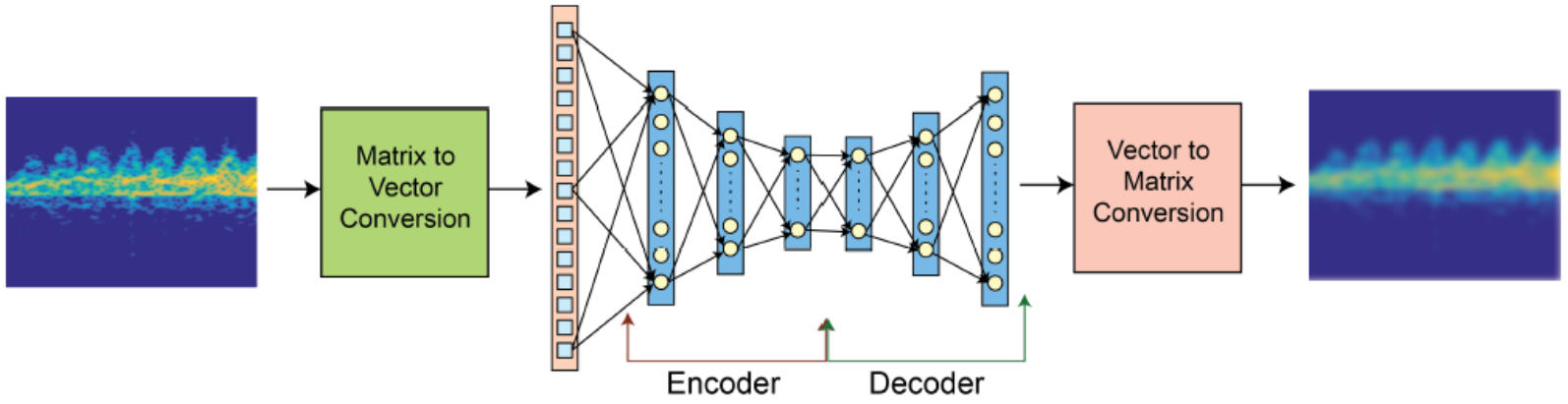
Convolutional Neural Networks

- *Consist of a number of convolutional and subsampling layers optionally followed by fully connected layers.*



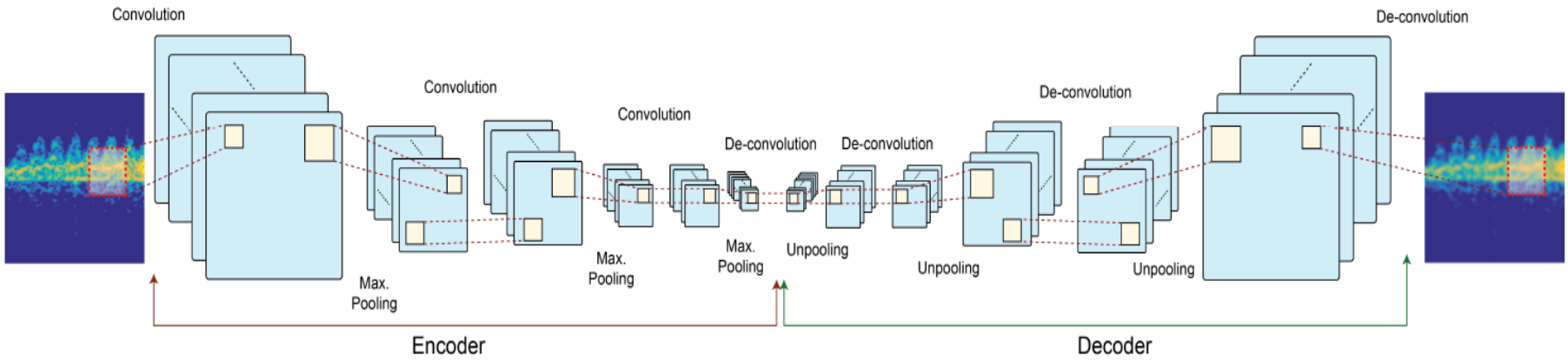
Auto-Encoders (AEs)

- Neural network that takes unlabeled data and tries to learn the identity function, i.e. optimizes weights such that output approximates input.



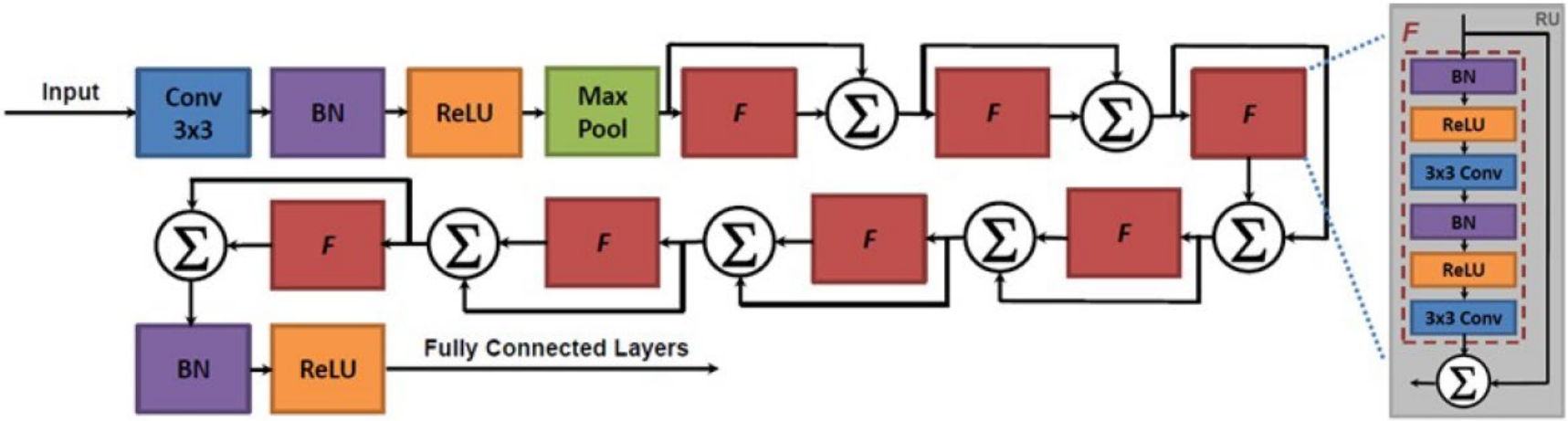
Convolutional Auto-Encoders (CAEs)

- CAEs combine the benefits of convolutional filtering in CNN's with unsupervised pre-training of autoencoders



Residual Neural Networks

- Residual units include a short cut path
  - If no new information is learned by adding layers, identity mapping can be used



Next Time…

… Examples from radar applications