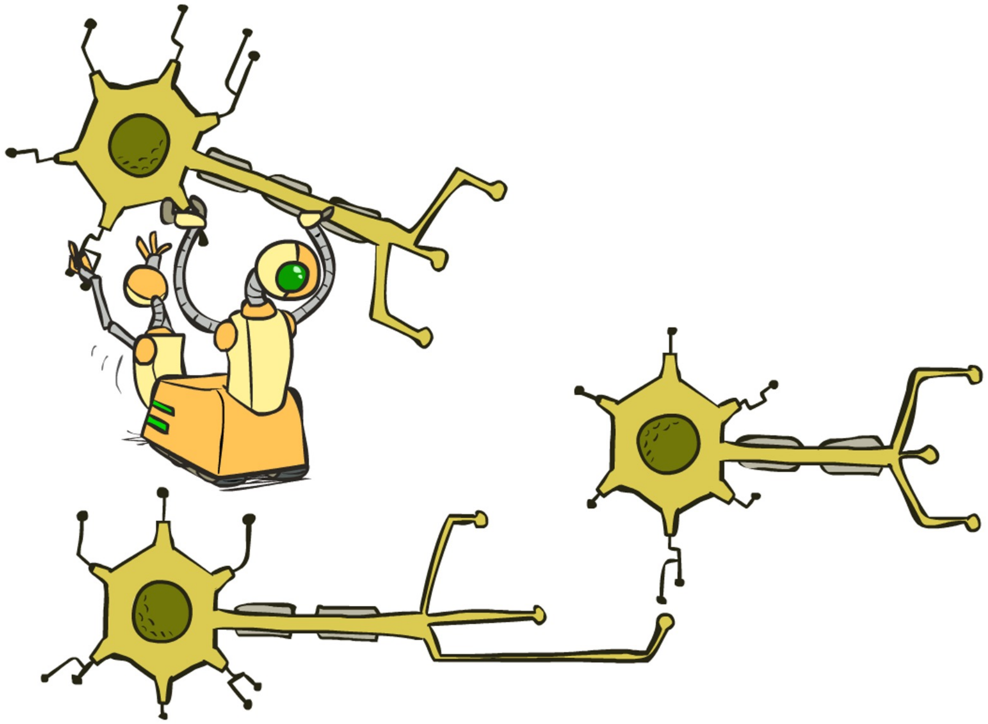
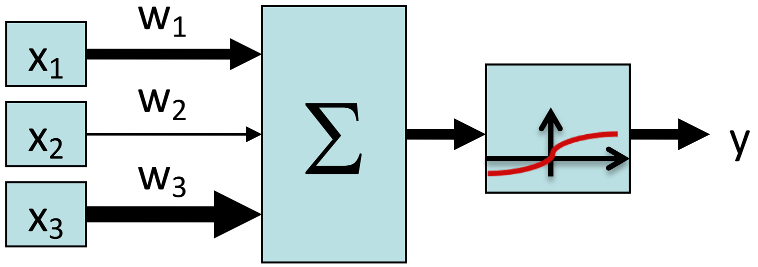
CS 188: Artificial Intelligence

Neural Networks



[These slides were created by Dan Klein, Pieter Abbeel, Anca Dragan for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

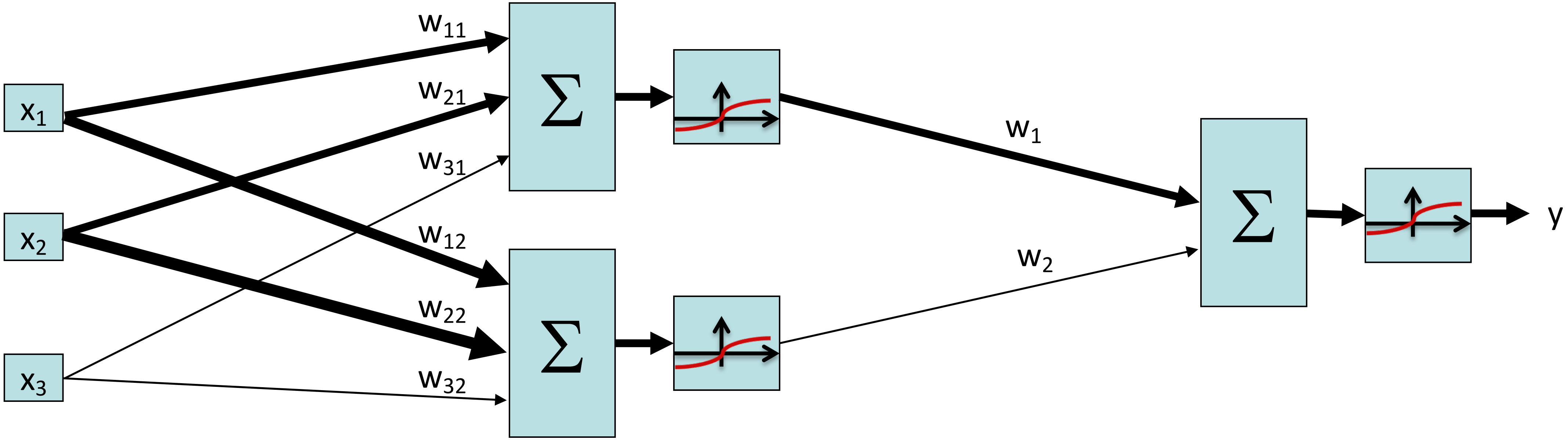
Recall: Perceptron with Sigmoid Activation



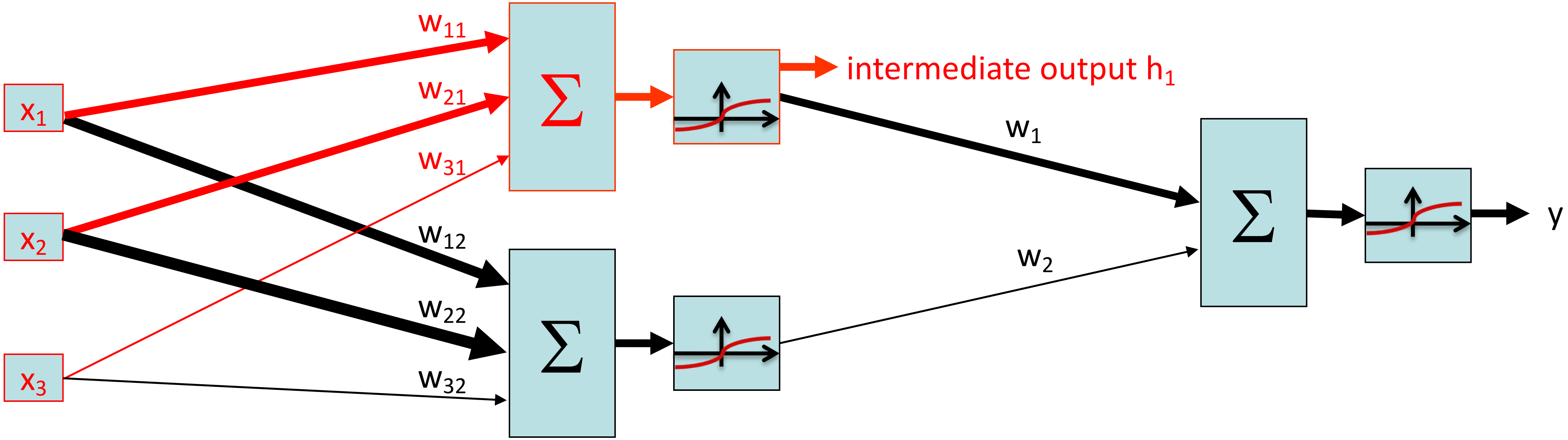
nonlinear (probability)

$$
\begin{aligned}
y &= \phi(w_1 x_1 + w_2 x_2 + w_3 x_3) \\
&= \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + w_3 x_3)}}
\end{aligned}
$$

Recall: 2-Layer, 2-Neuron Neural Network

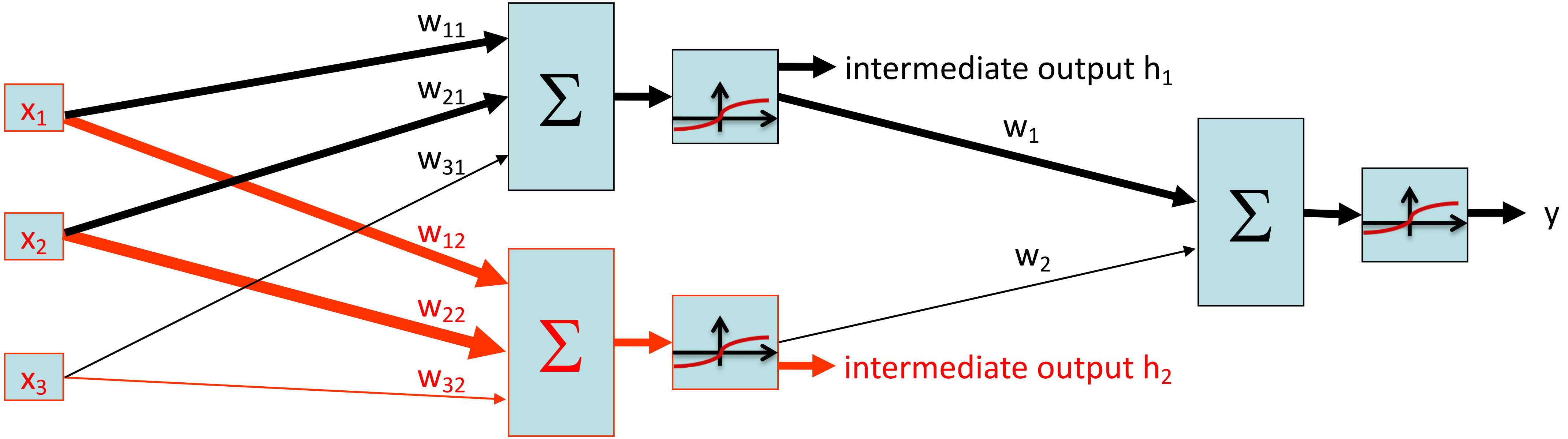


Recall: 2-Layer, 2-Neuron Neural Network



$$\text{intermediate output } h_1 = \phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) = \frac{1}{1 + e^{-(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)}}$$

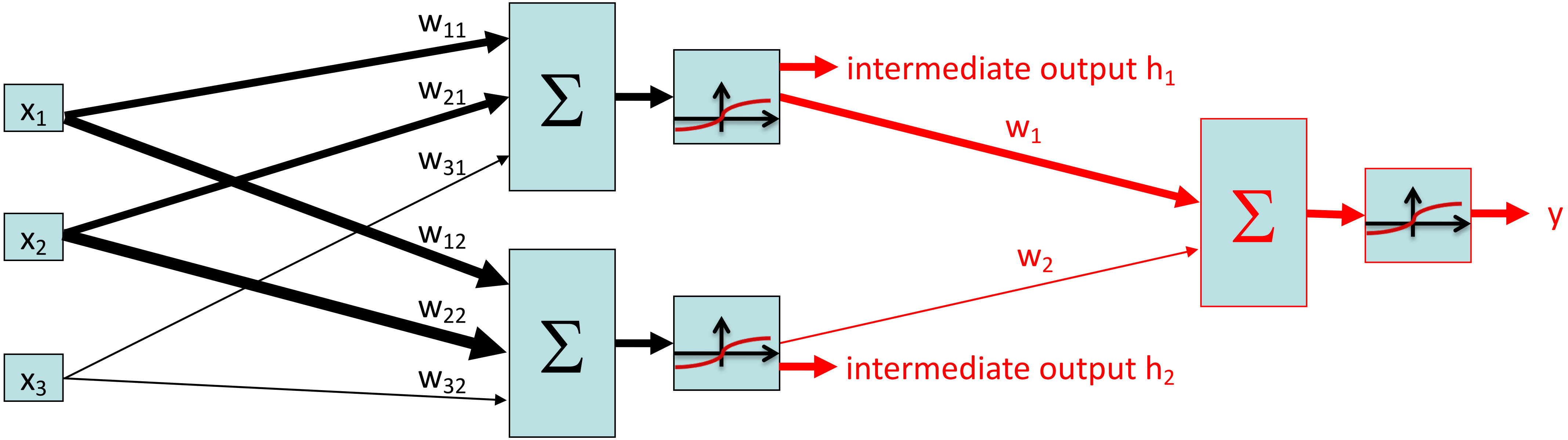
Recall: 2-Layer, 2-Neuron Neural Network



$$\text{intermediate output } h_2 = \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$

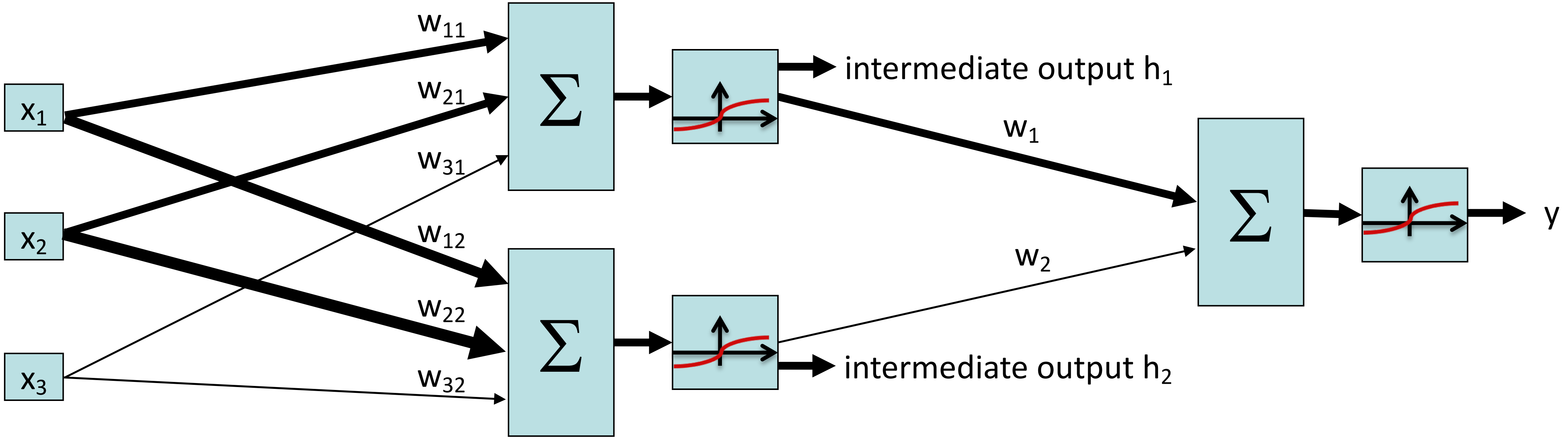
$$= \frac{1}{1 + e^{-(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)}}$$

Recall: 2-Layer, 2-Neuron Neural Network



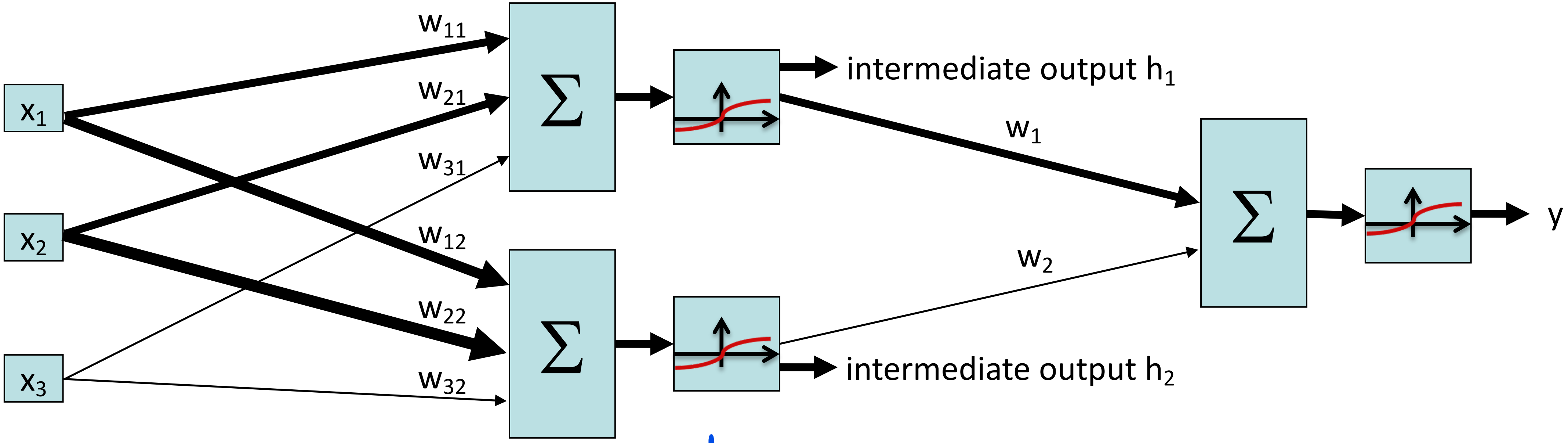
$$y = \phi(w_1 h_1 + w_2 h_2)$$
$$= \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}}$$

Recall: 2-Layer, 2-Neuron Neural Network



$$
\begin{aligned}
y &= \phi(w_1 h_1 + w_2 h_2) \\
&= \phi(w_1 \phi(w_{11} x_1 + w_{21} x_2 + w_{31} x_3) + w_2 \phi(w_{12} x_1 + w_{22} x_2 + w_{32} x_3))
\end{aligned}
$$

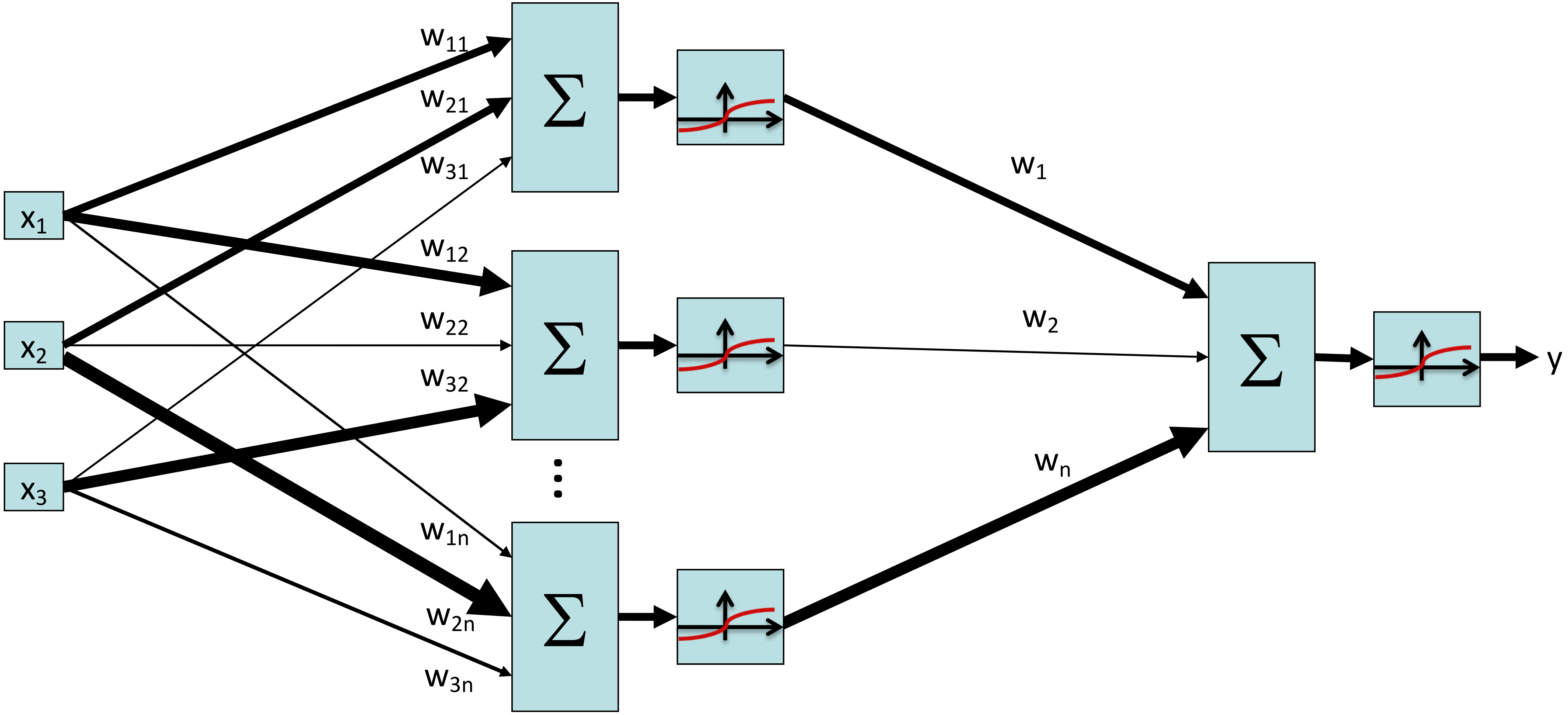
Recall: 2-Layer, 2-Neuron Neural Network



h

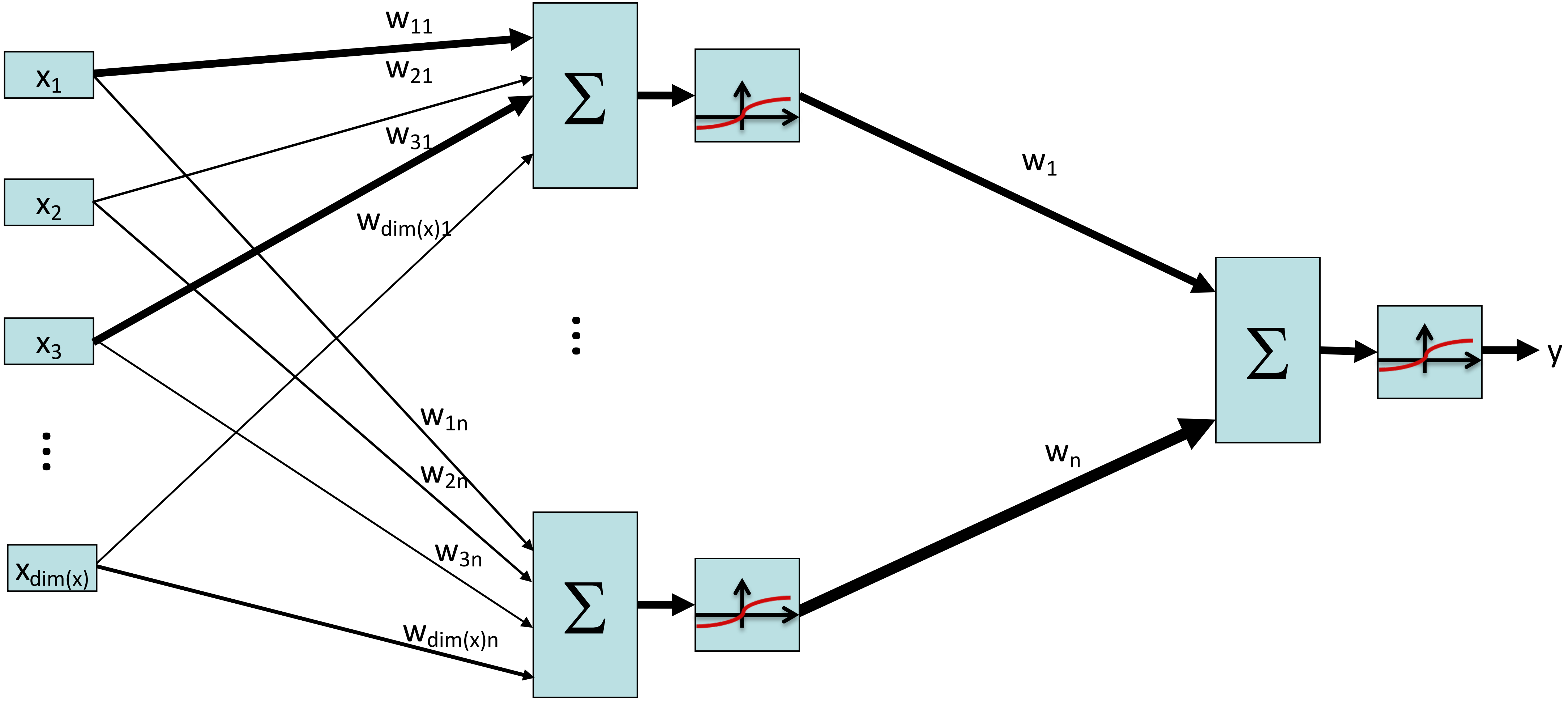
$$\phi(x \times W_{\text{layer 1}}) = h \qquad \phi(h \times W_{\text{layer 2}}) = y$$

Recall: generalize number of hidden neurons



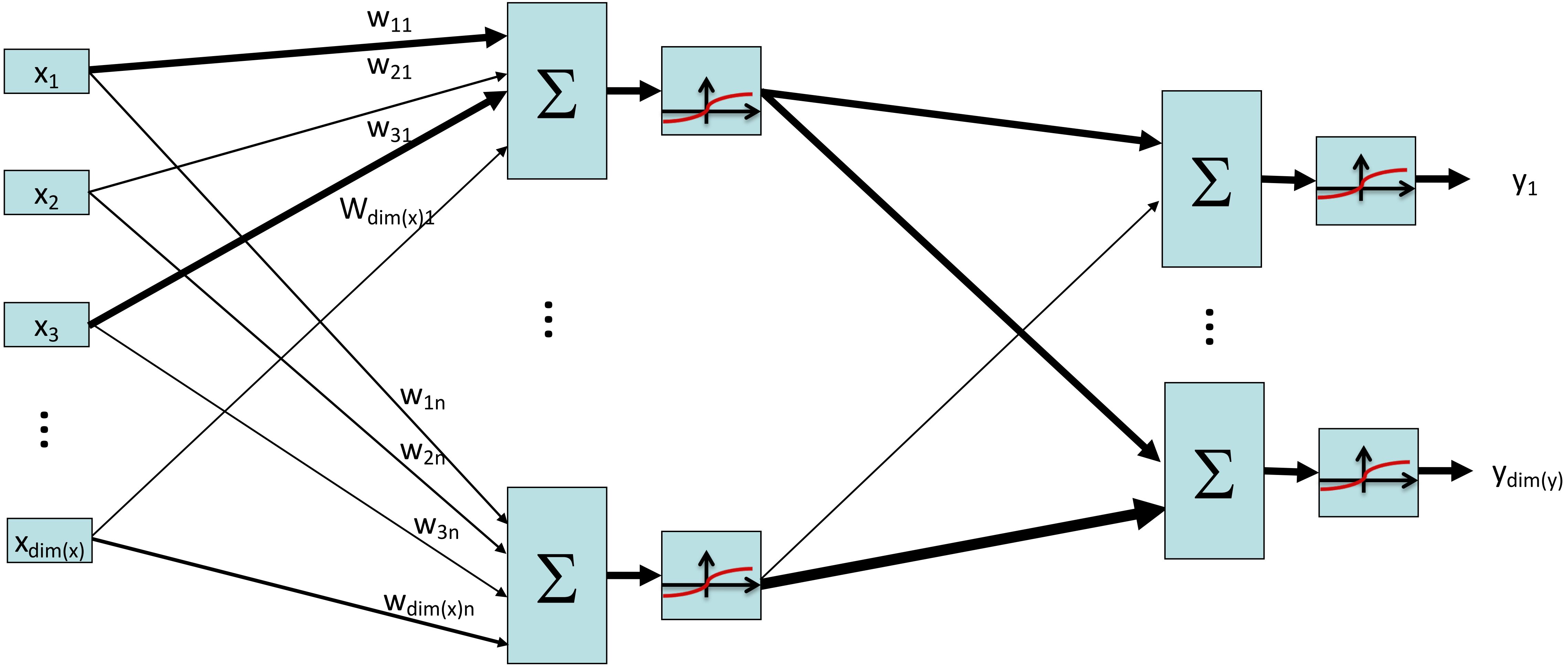
The hidden layer doesn’t necessarily need to have 3 neurons; it could have any arbitrary number *n* neurons.

Recall: generalize number of input features



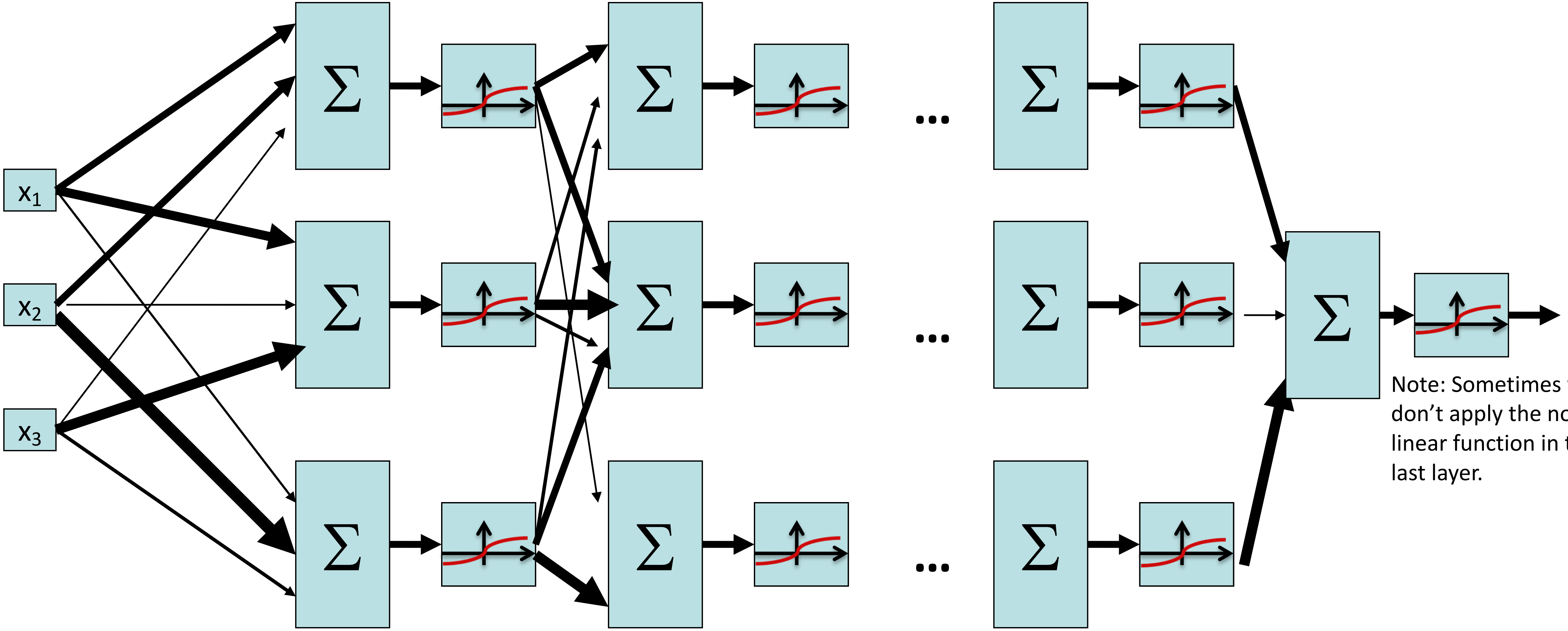
The input feature vector doesn't necessarily need to have 3 features; it could have some arbitrary number *dim(x)* of features.

Recall: generalize number of outputs



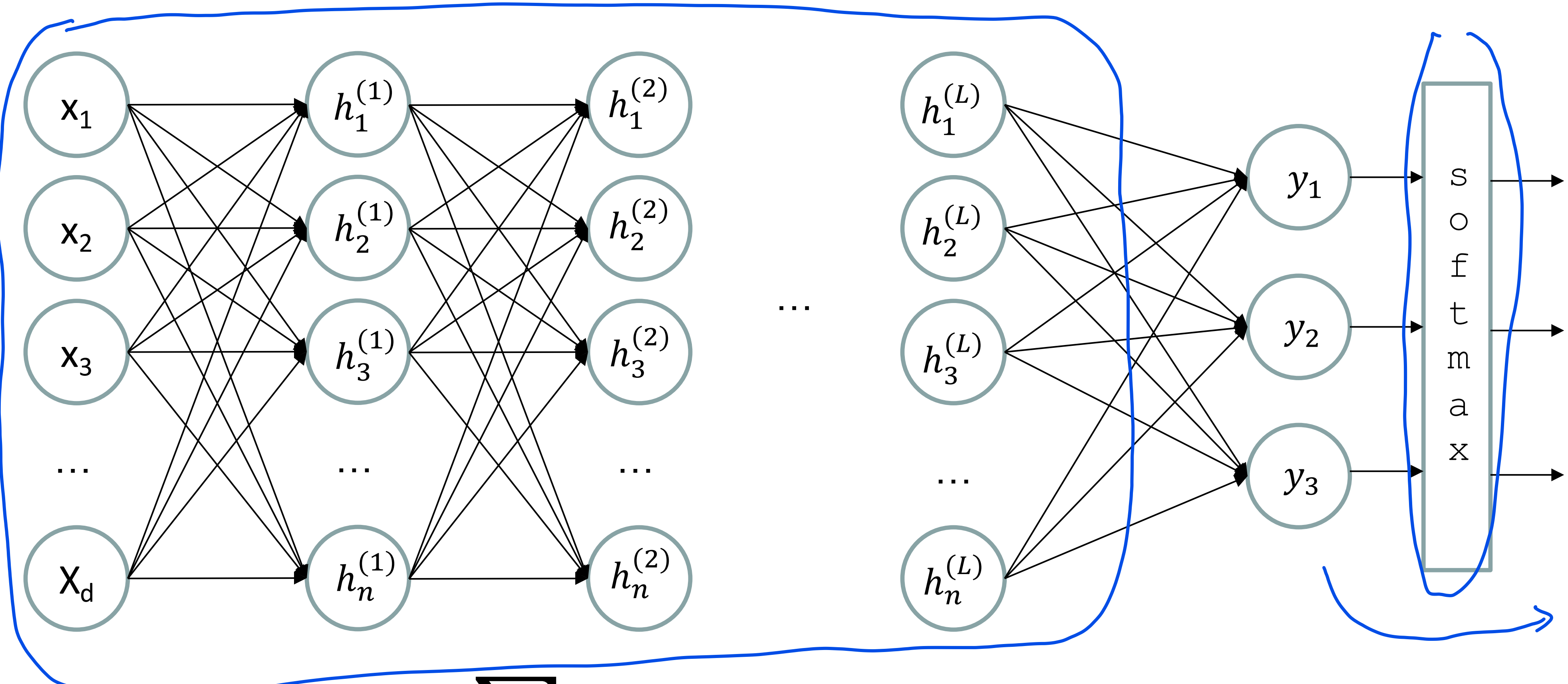
The output doesn't necessarily need to be just one number; it could be some arbitrary *dim(y)* length vector.

Recall: generalize number of layers



Note: Sometimes we don't apply the non-linear function in the last layer.

Deep Neural Network for 3-way classification



$$P(y_1|x;w) = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

$$P(y_2|x;w) = \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

$$P(y_3|x;w) = \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

→ multiplicative class

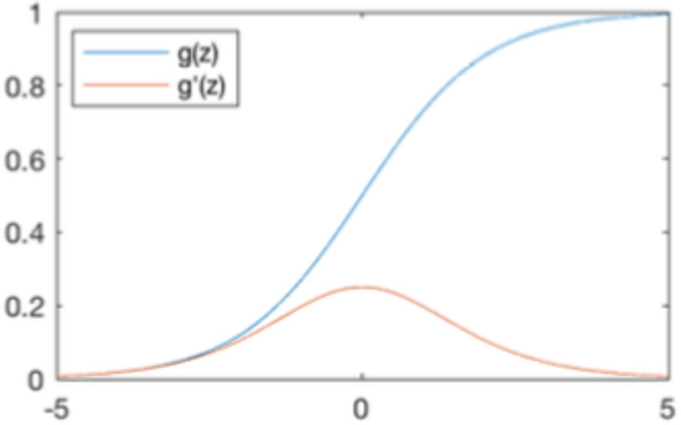
$$h_i^{(\text{layer } l)} = \phi\Big(\sum_j w_{ji}^{(\text{layer } l)} \cdot h_j^{(\text{layer } l-1)}\Big)$$

ϕ = **nonlinear activation function**

- Neural network with L layers
- $h^{(l)}$: activations at layer l
- $w^{(l)}$: weights taking activations from layer l-1 to layer l

Recall: Common Activation Functions

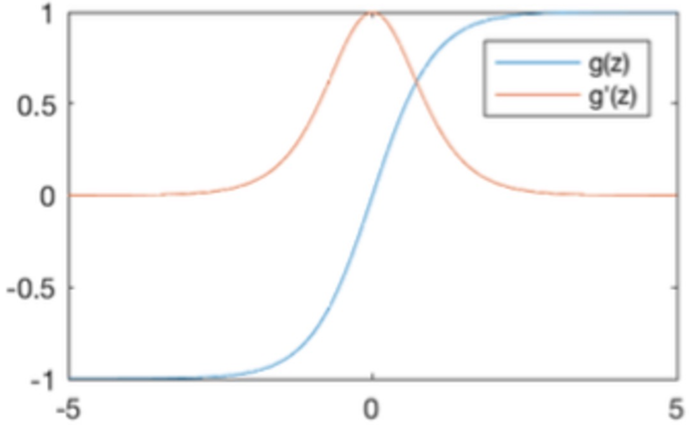
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

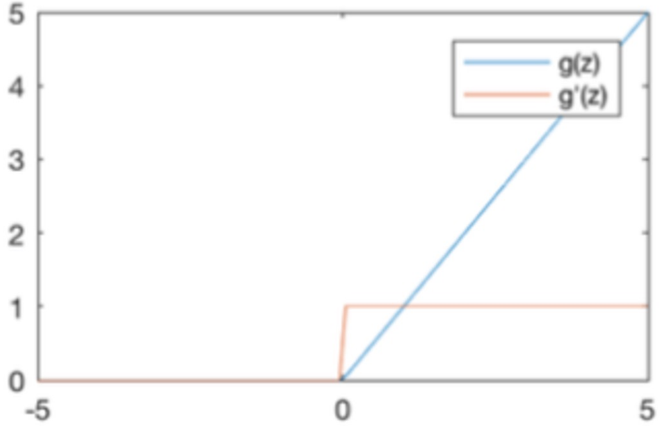
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

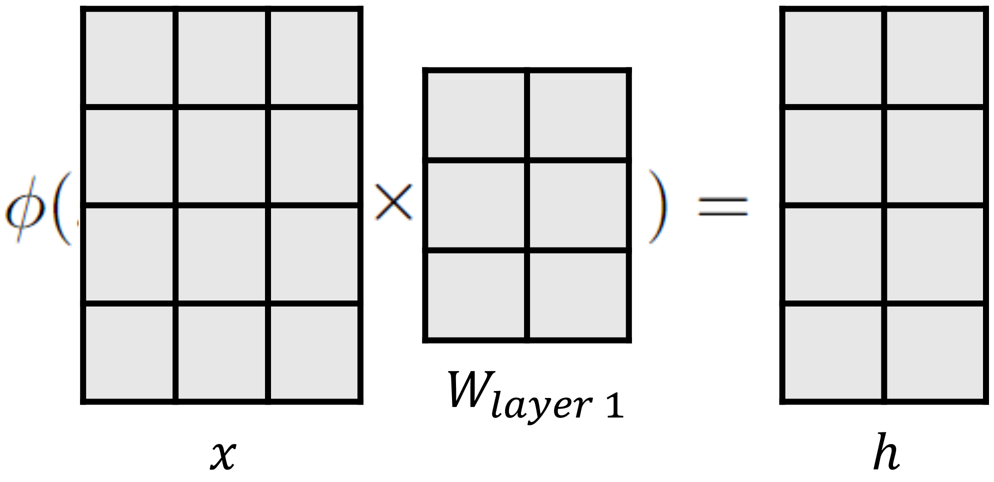


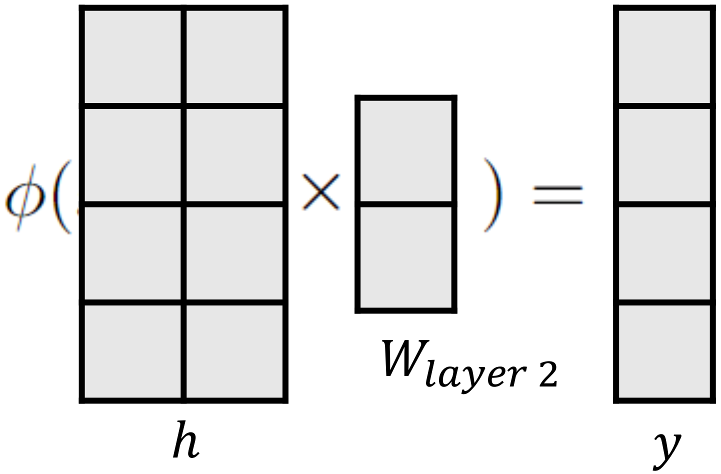
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

[source: MIT 6.S191 introtodeeplearning.com]

Recall: Sizes of neural networks

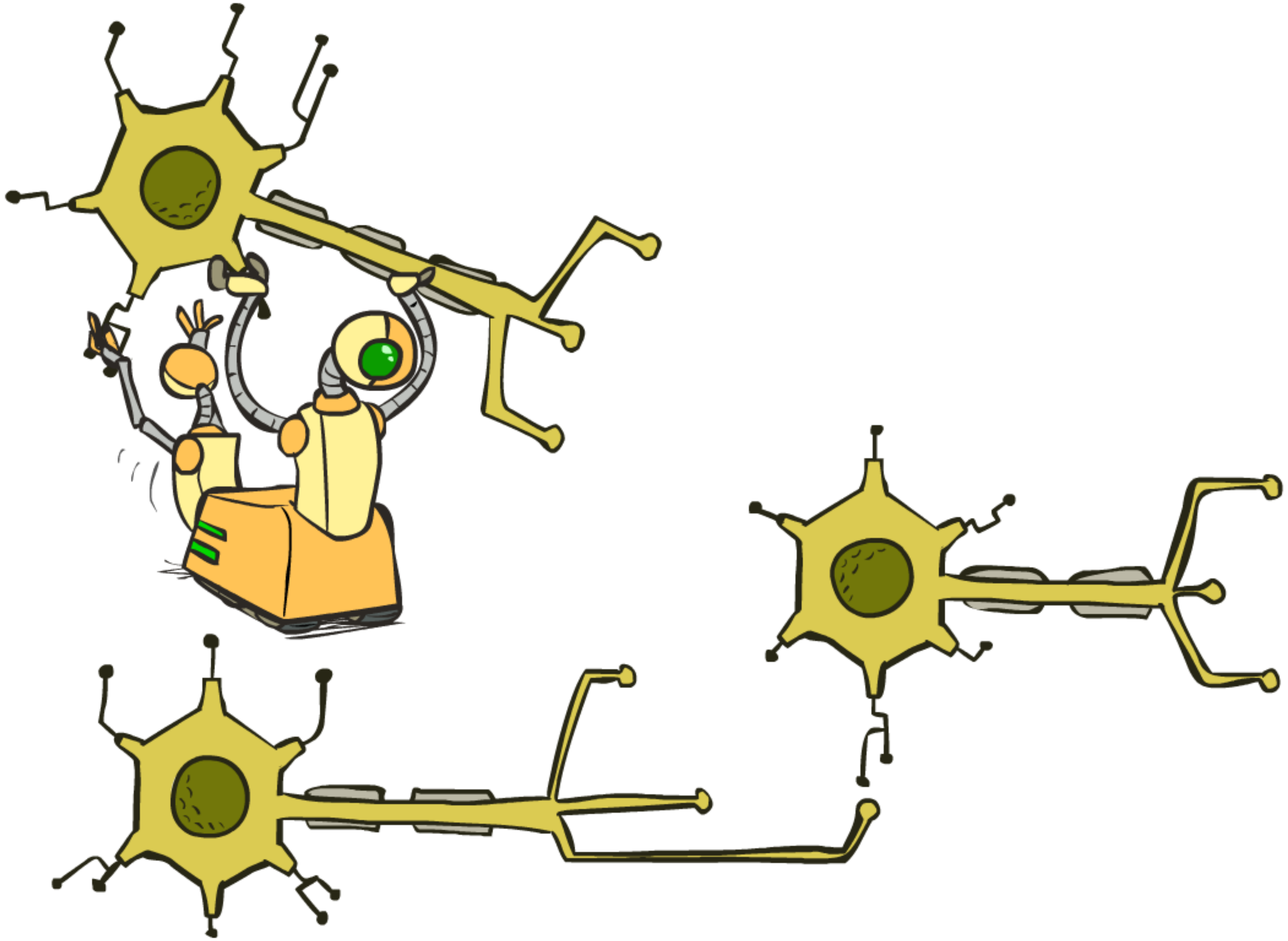




We have a neural network with the matrices drawn.

1. How many layers are in the network?
   2
2. How many input dimensions dim(x)?
   3
3. How many hidden neurons n?
   2
4. How many output dimensions dim(y)?
   1
5. What is the batch size?
   4

Training Neural Networks



Recall: Deep Neural Network Training

Training the deep neural network is just like logistic regression:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

just w tends to be a much, much larger vector

-> just run gradient ascent

+ stop when log likelihood of hold-out data starts to decrease

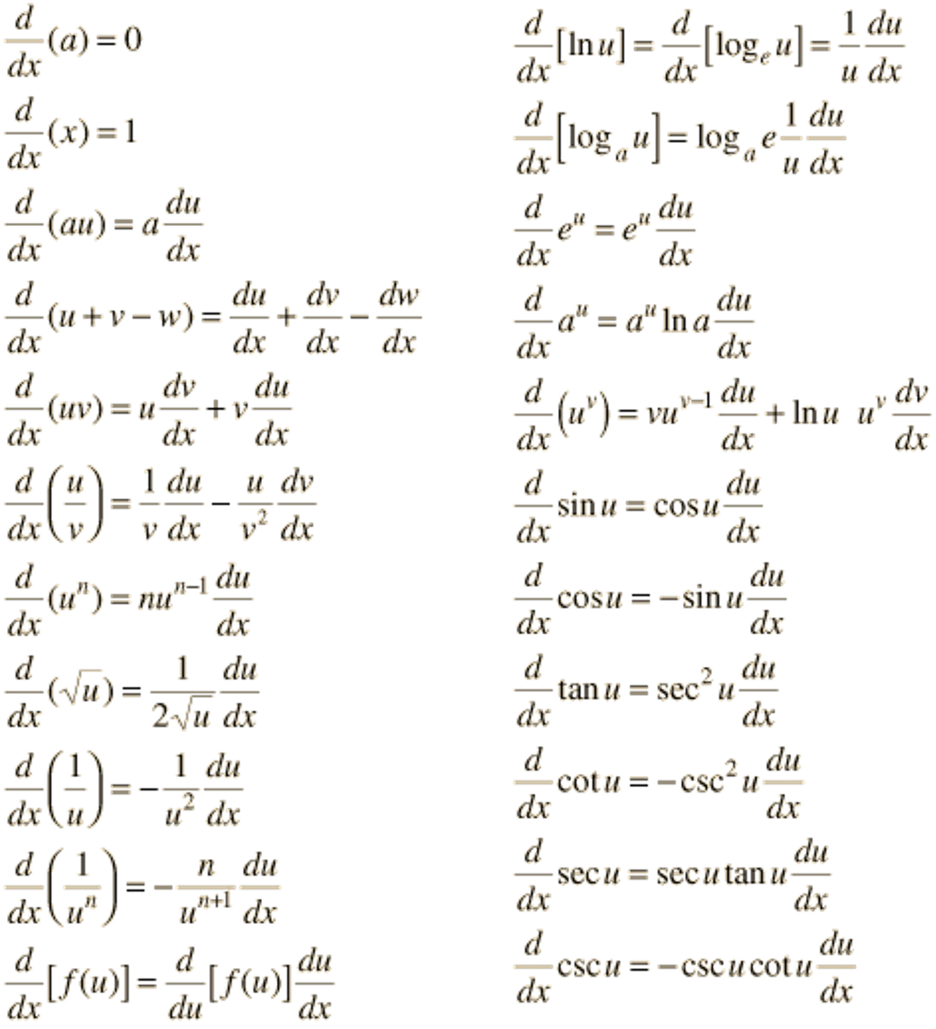
Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w \quad ll(w) = \max_w \quad \underbrace{\sum_i \log P(y^{(i)}|x^{(i)};w)}_{g(w)}$$

```
init $w$
for iter = 1, 2, …
    $w \leftarrow w + \underset{10^{-3}}{\alpha} * \underbrace{\sum_i \nabla \log P(y^{(i)}|x^{(i)};w)}_{\nabla g(w)}$
```

How about computing all the derivatives?

Derivatives tables:



$$\frac{d}{dx}(a) = 0$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(au) = a\frac{du}{dx}$$

$$\frac{d}{dx}(u + v - w) = \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{1}{v}\frac{du}{dx} - \frac{u}{v^2}\frac{dv}{dx}$$

$$\frac{d}{dx}(u^n) = nu^{n-1}\frac{du}{dx}$$

$$\frac{d}{dx}(\sqrt{u}) = \frac{1}{2\sqrt{u}}\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u}\right) = -\frac{1}{u^2}\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u^n}\right) = -\frac{n}{u^{n+1}}\frac{du}{dx}$$

$$\frac{d}{dx}[f(u)] = \frac{d}{du}[f(u)]\frac{du}{dx}$$

$$\frac{d}{dx}[\ln u] = \frac{d}{dx}[\log_e u] = \frac{1}{u}\frac{du}{dx}$$

$$\frac{d}{dx}\left[\log_a u\right] = \log_a e\frac{1}{u}\frac{du}{dx}$$

$$\frac{d}{dx}e^u = e^u\frac{du}{dx}$$

$$\frac{d}{dx}a^u = a^u \ln a\frac{du}{dx}$$

$$\frac{d}{dx}\left(u^v\right) = vu^{v-1}\frac{du}{dx} + \ln u \quad u^v\frac{dv}{dx}$$

$$\frac{d}{dx}\sin u = \cos u\frac{du}{dx}$$

$$\frac{d}{dx}\cos u = -\sin u\frac{du}{dx}$$

$$\frac{d}{dx}\tan u = \sec^2 u\frac{du}{dx}$$

$$\frac{d}{dx}\cot u = -\csc^2 u\frac{du}{dx}$$

$$\frac{d}{dx}\sec u = \sec u \tan u\frac{du}{dx}$$

$$\frac{d}{dx}\csc u = -\csc u \cot u\frac{du}{dx}$$

[source: http://hyperphysics.phy-astr.gsu.edu/hbase/Math/derfunc.html

How about computing all the derivatives?

- But neural net f is never one of those?
  - No problem: CHAIN RULE:

If

$$f(x) = g(h(x))$$

Then

$$f'(x) = g'(h(x))h'(x)$$

Derivatives can be computed by following well-defined procedures

Automatic Differentiation

Automatic differentiation software

e.g. TensorFlow, PyTorch, Jax

Only need to program the function g(x,y,w)

Can automatically compute all derivatives w.r.t. all entries in w

This is typically done by caching info during forward computation pass of f, and then doing a backward pass = "backpropagation"

Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass

Need to know this exists

How this is done? Details outside of scope of CS188, but we'll show a basic example

Backpropagation*

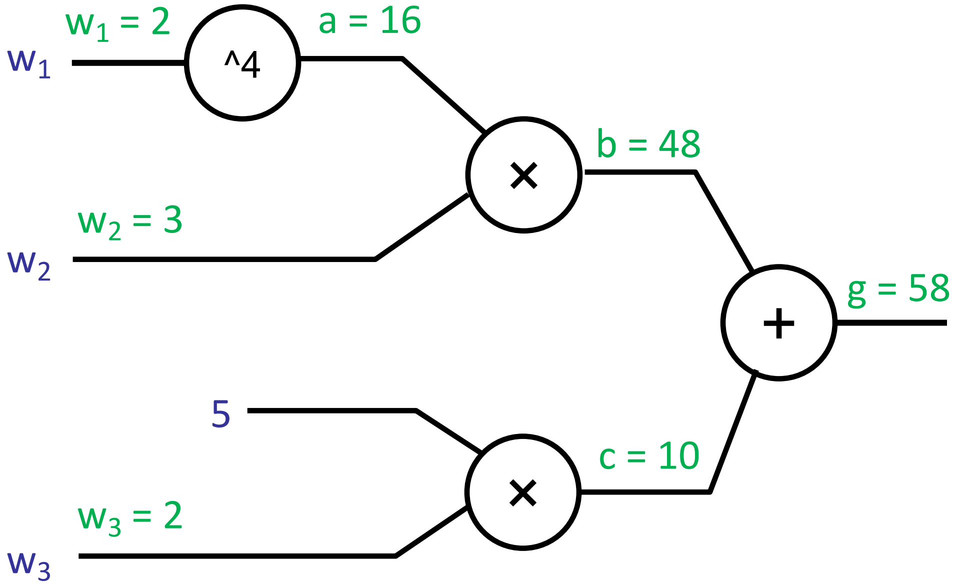
- Gradient of $g(w_1, w_2, w_3) = w_1^4 w_2 + 5w_3$ at $w_1 = 2$, $w_2 = 3$, $w_3 = 2$
- Think of *g* as a composition of many functions
  - Then, we can use the chain rule to compute the gradient
- g = b + c

  $$\frac{\partial g}{\partial b} = 1, \frac{\partial g}{\partial c} = 1$$
- $b = a \times w_2$

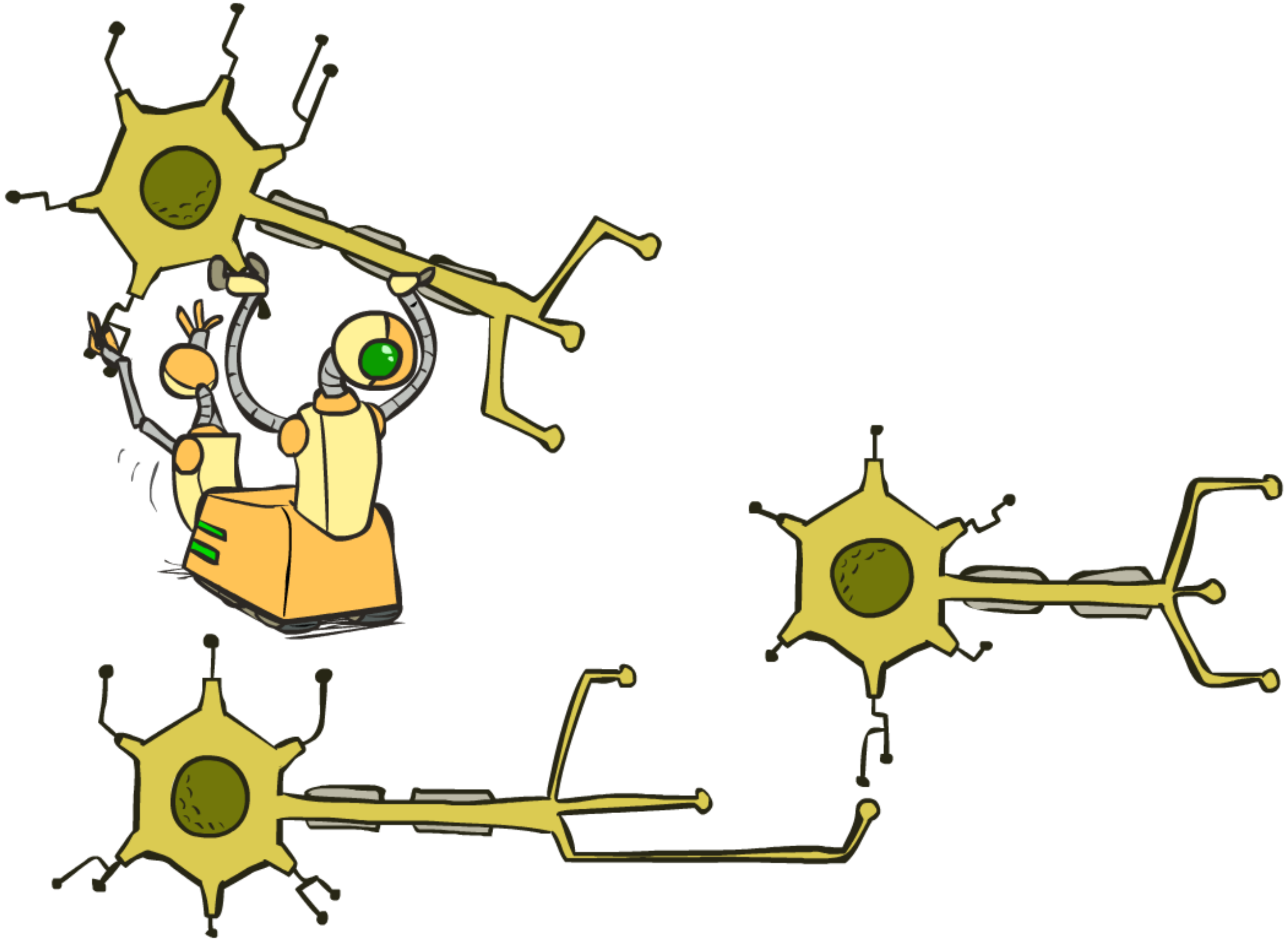
  $$\frac{\partial g}{\partial a} = \frac{\partial g}{\partial b}\frac{\partial b}{\partial a} = 1 \cdot w_2 = 3 \qquad \frac{\partial g}{\partial w_2} = \frac{\partial g}{\partial b}\frac{\partial b}{\partial w_2} = 1 \cdot a = 16$$
- $a = w_1^4$

  $$\frac{\partial g}{\partial w_1} = \frac{\partial g}{\partial a}\frac{\partial a}{\partial w_1} = 3 \cdot 4w_1^3 = 96$$
- $c = 5w_{\overset{3}{1}}$

  $$\frac{\partial g}{\partial w_3} = \frac{\partial g}{\partial c}\frac{\partial c}{\partial w_3} = 1 \cdot 5 = 5$$



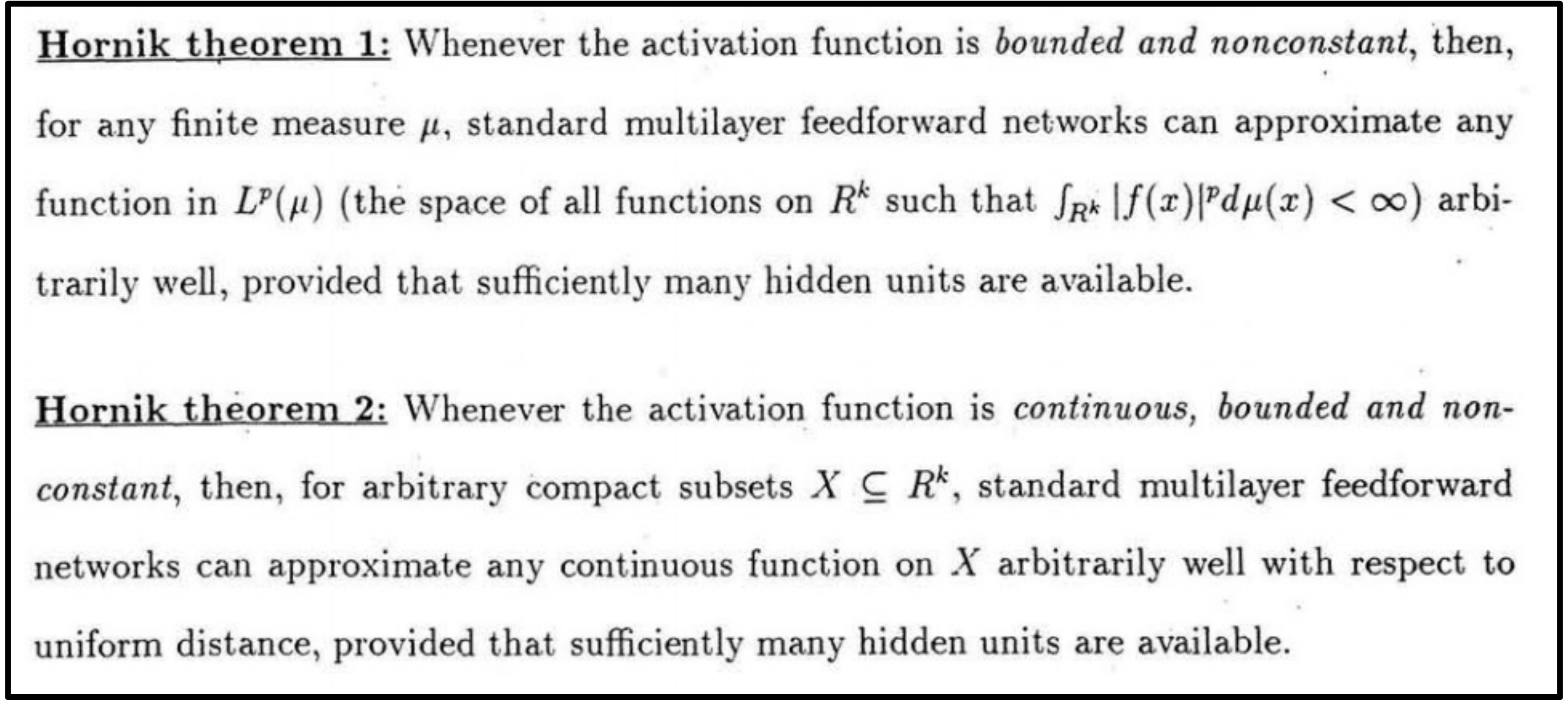
Properties of Neural Networks



Neural Networks Properties

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

Universal Function Approximation Theorem*



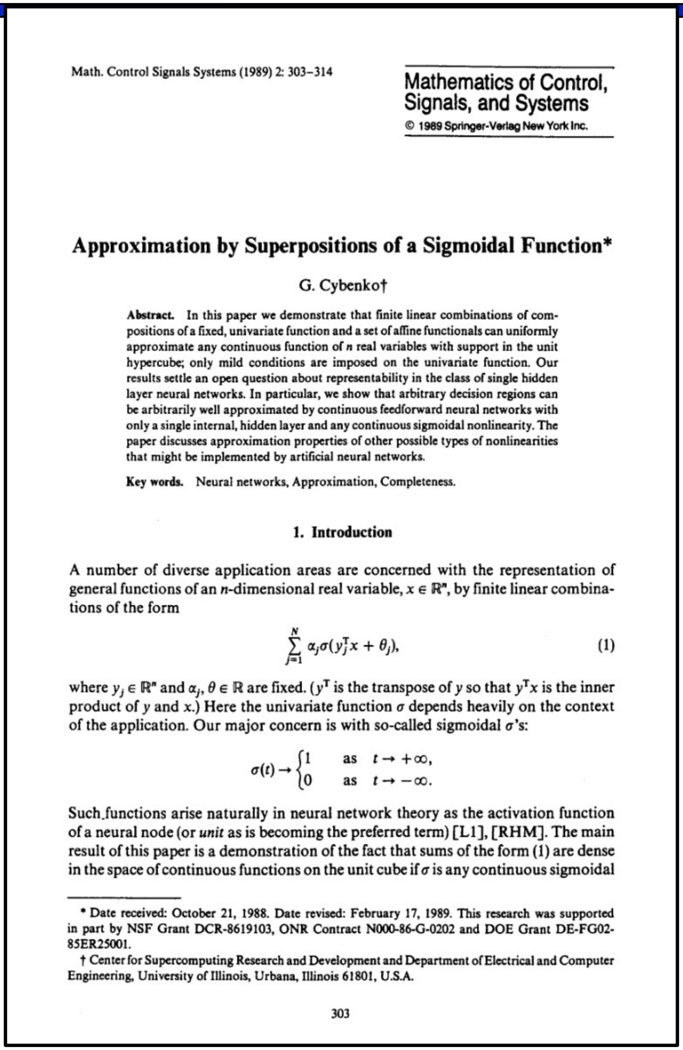
__Hornik theorem 1:__ Whenever the activation function is *bounded and nonconstant*, then, for any finite measure μ, standard multilayer feedforward networks can approximate any function in $L^p(\mu)$ (the space of all functions on R^k such that $\int_{R^k} |f(x)|^p d\mu(x) < \infty$) arbitrarily well, provided that sufficiently many hidden units are available.

__Hornik theorem 2:__ Whenever the activation function is *continuous, bounded and nonconstant*, then, for arbitrary compact subsets $X \subseteq R^k$, standard multilayer feedforward networks can approximate any continuous function on X arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- __In words:__ Given any continuous function f(x), if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate f(x).

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”
Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”
Leshno and Schocken (1991) ”Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”

Universal Function Approximation Theorem*



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

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

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 $y^{\mathrm{T}}x$ 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:

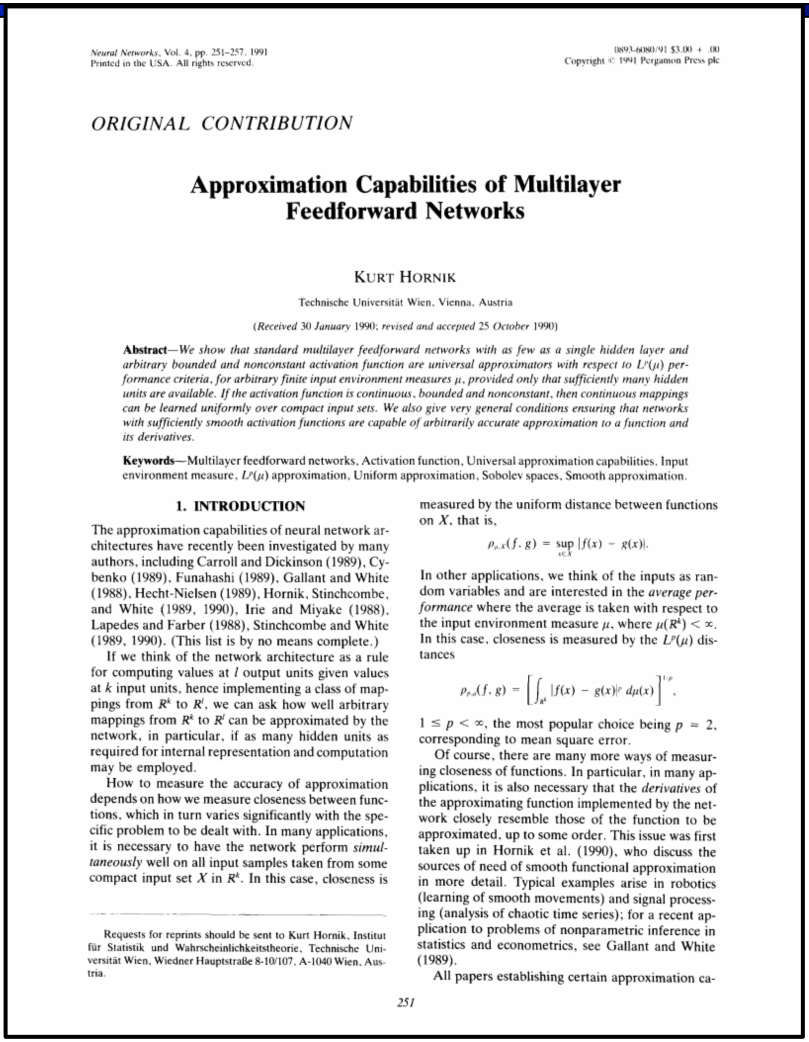
$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

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-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

303



Neural Networks, Vol. 4, pp. 251–257, 1991
Printed in the USA. All rights reserved.

0893-6080/91 $3.00 + .00
Copyright © 1991 Pergamon Press plc

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Technische Universität Wien, Vienna, Austria

(*Received 30 January* 1990; *revised and accepted 25 October* 1990)

Abstract—*We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to $L^p(\mu)$ performance criteria, for arbitrary finite input environment measures μ, provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonconstant, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.*

Keywords—Multilayer feedforward networks, Activation function, Universal approximation capabilities, Input environment measure, $L^p(\mu)$ approximation, Uniform approximation, Sobolev spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural network architectures have recently been investigated by many authors, including Carroll and Dickinson (1989), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989, 1990), Irie and Miyake (1988), Lapedes and Farber (1988), Stinchcombe and White (1989, 1990). (This list is by no means complete.)

If we think of the network architecture as a rule for computing values at l output units given values at k input units, hence implementing a class of mappings from R^k to R^l, we can ask how well arbitrary mappings from R^k to R^l can be approximated by the network, in particular, if as many hidden units as required for internal representation and computation may be employed.

How to measure the accuracy of approximation depends on how we measure closeness between functions, which in turn varies significantly with the specific problem to be dealt with. In many applications, it is necessary to have the network perform *simultaneously* well on all input samples taken from some compact input set X in R^k. In this case, closeness is measured by the uniform distance between functions on X, that is,

$$\rho_{\mu,X}(f, g) = \sup_{x \in X} |f(x) - g(x)|.$$

In other applications, we think of the inputs as random variables and are interested in the *average performance* where the average is taken with respect to the input environment measure μ, where $\mu(R^k) < \infty$. In this case, closeness is measured by the $L^p(\mu)$ distances

$$\rho_{p,\mu}(f, g) = \left[\int_{R^k} |f(x) - g(x)|^p \, d\mu(x) \right]^{1/p},$$

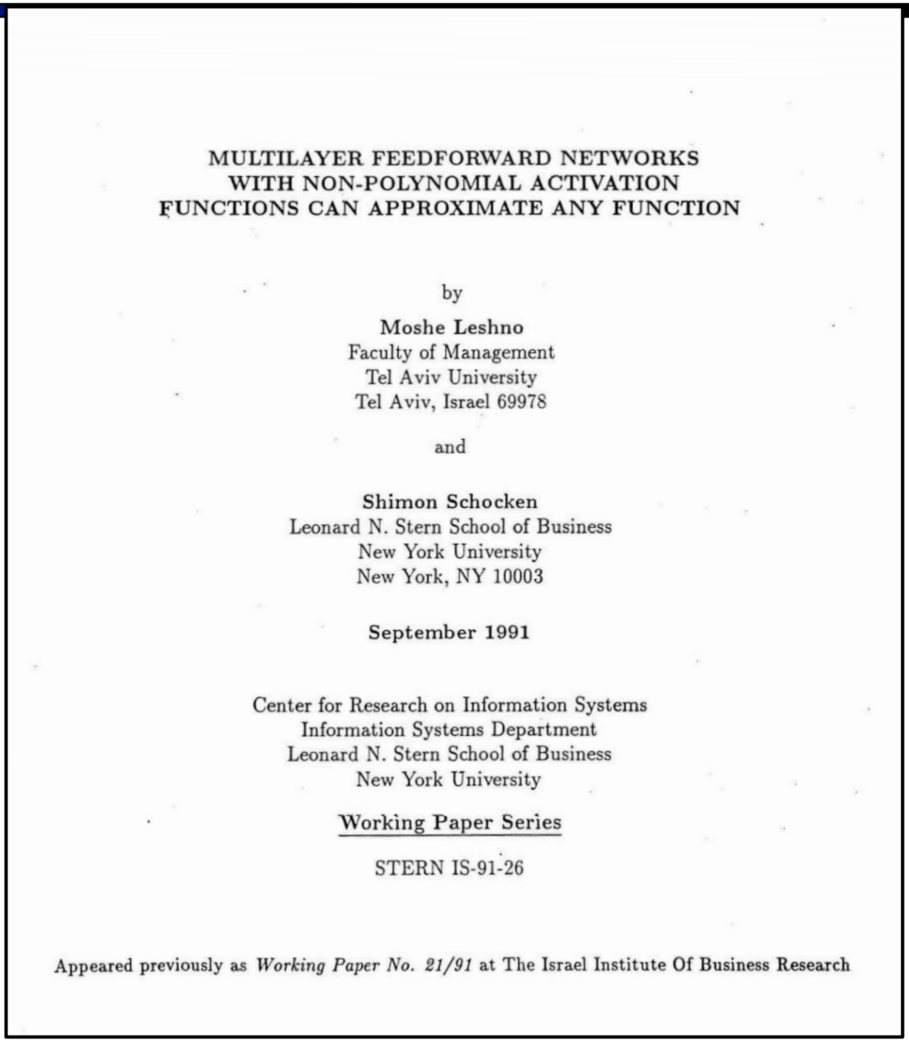
$1 \le p < \infty$, the most popular choice being $p = 2$, corresponding to mean square error.

Of course, there are many more ways of measuring closeness of functions. In particular, in many applications, it is also necessary that the *derivatives* of the approximating function implemented by the network closely resemble those of the function to be approximated, up to some order. This issue was first taken up in Hornik et al. (1990), who discuss the sources of need of smooth functional approximation in more detail. Typical examples arise in robotics (learning of smooth movements) and signal processing (analysis of chaotic time series); for a recent application to problems of nonparametric inference in statistics and econometrics, see Gallant and White (1989).

All papers establishing certain approximation ca-

Requests for reprints should be sent to Kurt Hornik, Institut für Statistik und Wahrscheinlichkeitstheorie, Technische Universität Wien, Wiedner Hauptstraße 8-10/107, A-1040 Wien, Austria.

251



MULTILAYER FEEDFORWARD NETWORKS WITH NON-POLYNOMIAL ACTIVATION FUNCTIONS CAN APPROXIMATE ANY FUNCTION

by

Moshe Leshno
Faculty of Management
Tel Aviv University
Tel Aviv, Israel 69978

and

Shimon Schocken
Leonard N. Stern School of Business
New York University
New York, NY 10003

September 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-26

Appeared previously as *Working Paper No. 21/91* at The Israel Institute Of Business Research

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”
Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”
Leshno and Schocken (1991) ”Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”

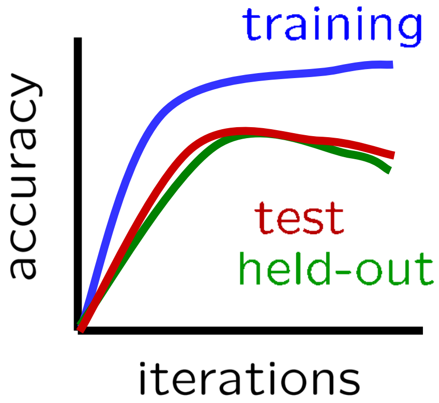
Neural Networks Properties

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

- Practical considerations
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting
    - (hence early stopping!)

Preventing Overfitting in Neural Networks

Early stopping:



Weight regularization

Weight Regularization

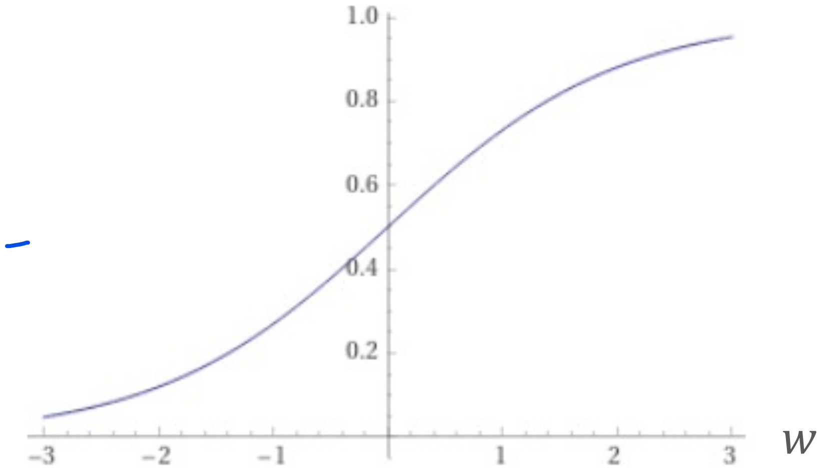
What can go wrong when we maximize log-likelihood?

Example: logistic regression with only one datapoint: f(x)=1, y=+1

$$\max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

- $P(y = +1|x; w) = \frac{1}{1+e^{-w\cdot f(x)}}$

$$\max_w \log \frac{1}{1+(\underset{\downarrow}{e}^{-w})^{\uparrow}}$$



$$\log\frac{1}{1+e^{-w}} = \log 1 - \log(1+e^{-w})$$

$$= 0 - \log(1+e^{-w})$$

w can grow very large and lead to overfitting and learning instability

Weight Regularization

What can go wrong when we maximize log-likelihood?

$$\max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

w can grow very large

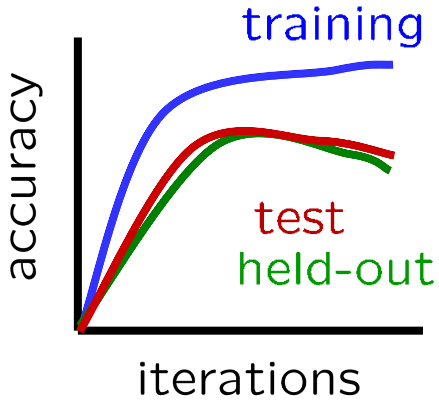
Solution: add an objective term to penalize weight magnitude

$$\max_w \sum_i \log P(y^{(i)} | x^{(i)}; w) - \frac{\lambda}{2} \sum_j w_j^2$$

λ is a hyperparameter (typically 0.1 to 0.0001 or smaller)

Preventing Overfitting in Neural Networks

Early stopping:

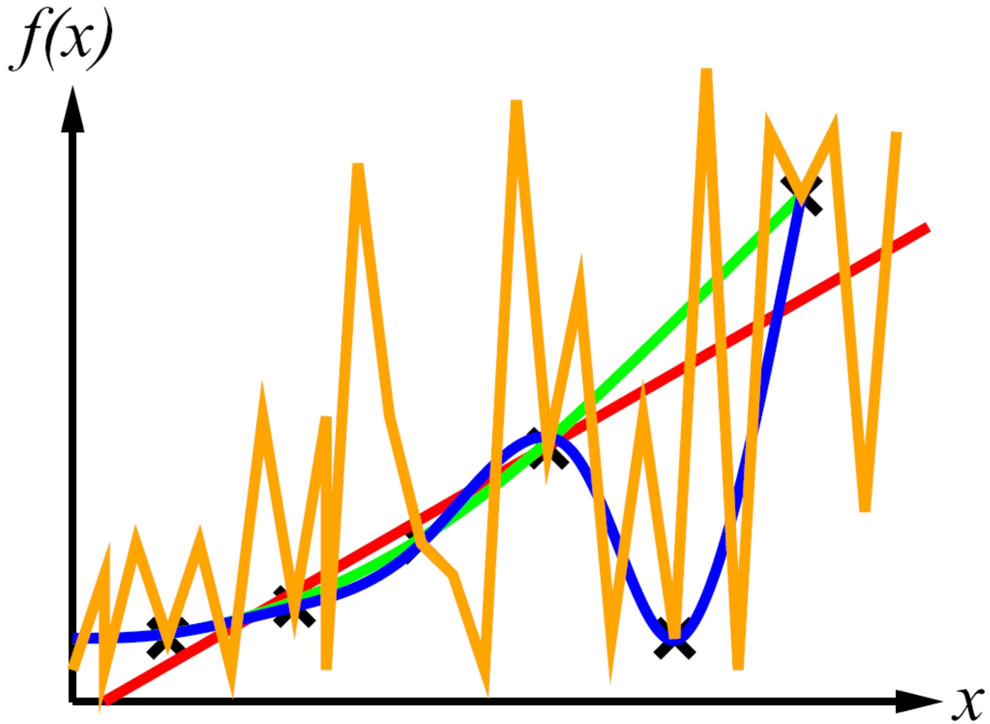


Weight regularization: $\max_{\mathrm{w}} \sum_i \log P(y^{(i)} | x^{(i)}; w) - \frac{\lambda}{2} \sum_j w_j^2$

Dropout

Consistency vs. Simplicity

- Example: curve fitting (regression, function approximation)



- Consistency vs. simplicity
- Ockham’s razor

Consistency vs. Simplicity

- Usually algorithms prefer consistency by default (why?)

- Several ways to operationalize “simplicity”
  - Reduce the hypothesis/model space
    - Assume more: e.g. independence assumptions, as in naïve Bayes
    - Fewer features or neurons
    - Other limits on model structure
  - Regularization
    - Laplace Smoothing: cautious use of small counts
    - Small weight vectors in neural networks (stay close to zero-mean prior)
    - Hypothesis space stays big, but harder to get to the outskirts

Fun Neural Net Demo Site

Demo-site:

- http://playground.tensorflow.org/

Summary of Key Ideas

Optimize probability of label given input

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

Continuous optimization

Gradient ascent:

- Compute steepest uphill direction = gradient (= just vector of partial derivatives)
- Take step in the gradient direction
- Repeat (until held-out data accuracy starts to drop = “early stopping”)

Deep neural nets

Last layer = still logistic regression

Now also many more layers before this last layer

- = computing the features
- the features are learned rather than hand-designed

Universal function approximation theorem

- `If` neural net is large enough
- `Then` neural net can represent any continuous mapping from input to output with arbitrary accuracy
- But remember: need to avoid overfitting / memorizing the training data ? early stopping!

Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)

Next: How well does deep learning work?