Neural Networks

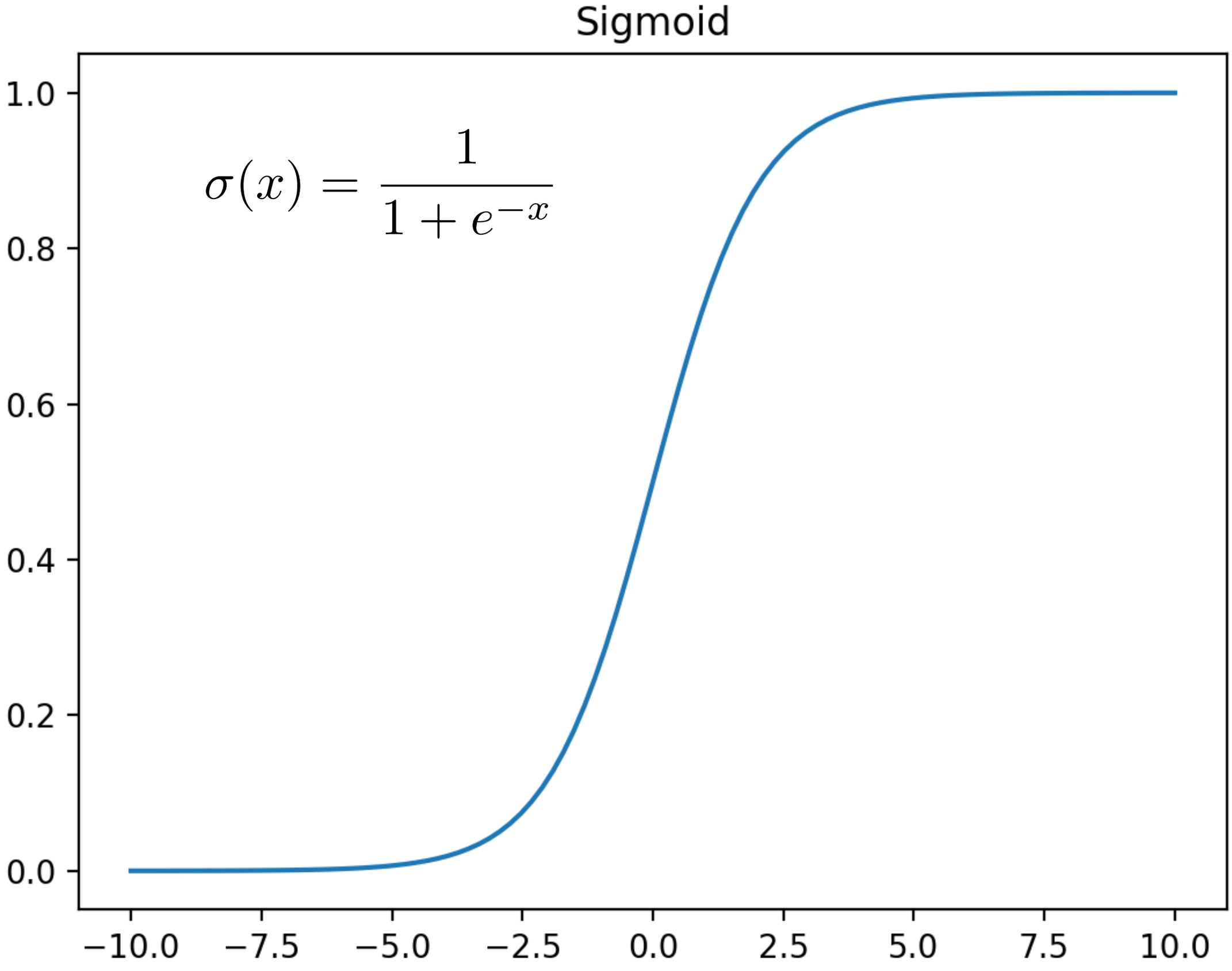
Logistic Regression

- Logistic Regression is a method for predicting values between 0 and 1 — often interpreted as probabilities.

- Because of this, logistic regression is most often used for binary classification.

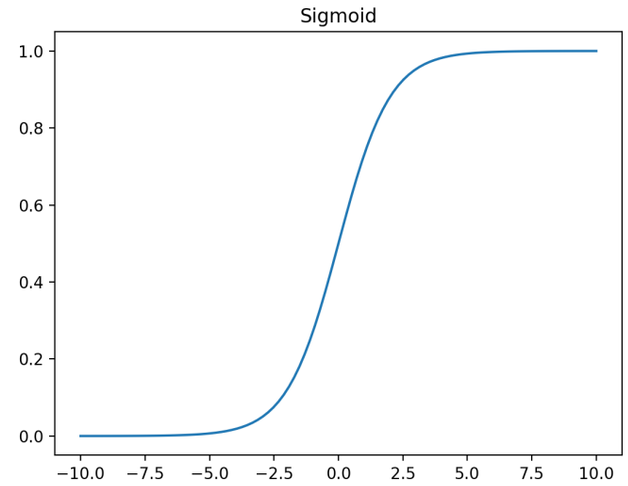
Logistic Regression

- We essentially perform linear regression and pass it through a function which squashes the output to the range (0,1) — which looks a lot like a probability.


Sigmoid
$\sigma(x) = \frac{1}{1 + e^{-x}}$
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Logistic Regression

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



$$Pr(y = true|x) = \frac{1}{1 + e^{-w \cdot \hat{x}}}$$

From Logistic Regression to Neural Networks

- Logistic regression takes a fixed number of inputs and calculates a probability — number in the range (0,1).

- This can be viewed as learning a single, real-valued feature.

- If we simultaneously learned many different logistic regression based features and another classifier which uses those features, then we would be using a specific kind of **neural network** called a **multilayer perceptron**. More next time.

Overview

- History of Neural Networks (beginnings until 1990s)
- Multilayer Perceptrons
    - Architecture
        - hidden units, activation functions, and layers
    - Learning
        - loss functions, gradient descent, backprop
- Later (if we have time):
    - Convolutional Networks
    - Recurrent Networks (Ellman and Hopfield networks)
    - Generative Adversarial Networks

Neural Networks

- A neural network represents a nonlinear function from a set of input variables **x** to a set of output variables **y** controlled by a set of adjustable parameters **W.**

$$f_W(x) : \mathcal{R}^{d_{in}} \rightarrow \mathcal{R}^{d_{out}}$$

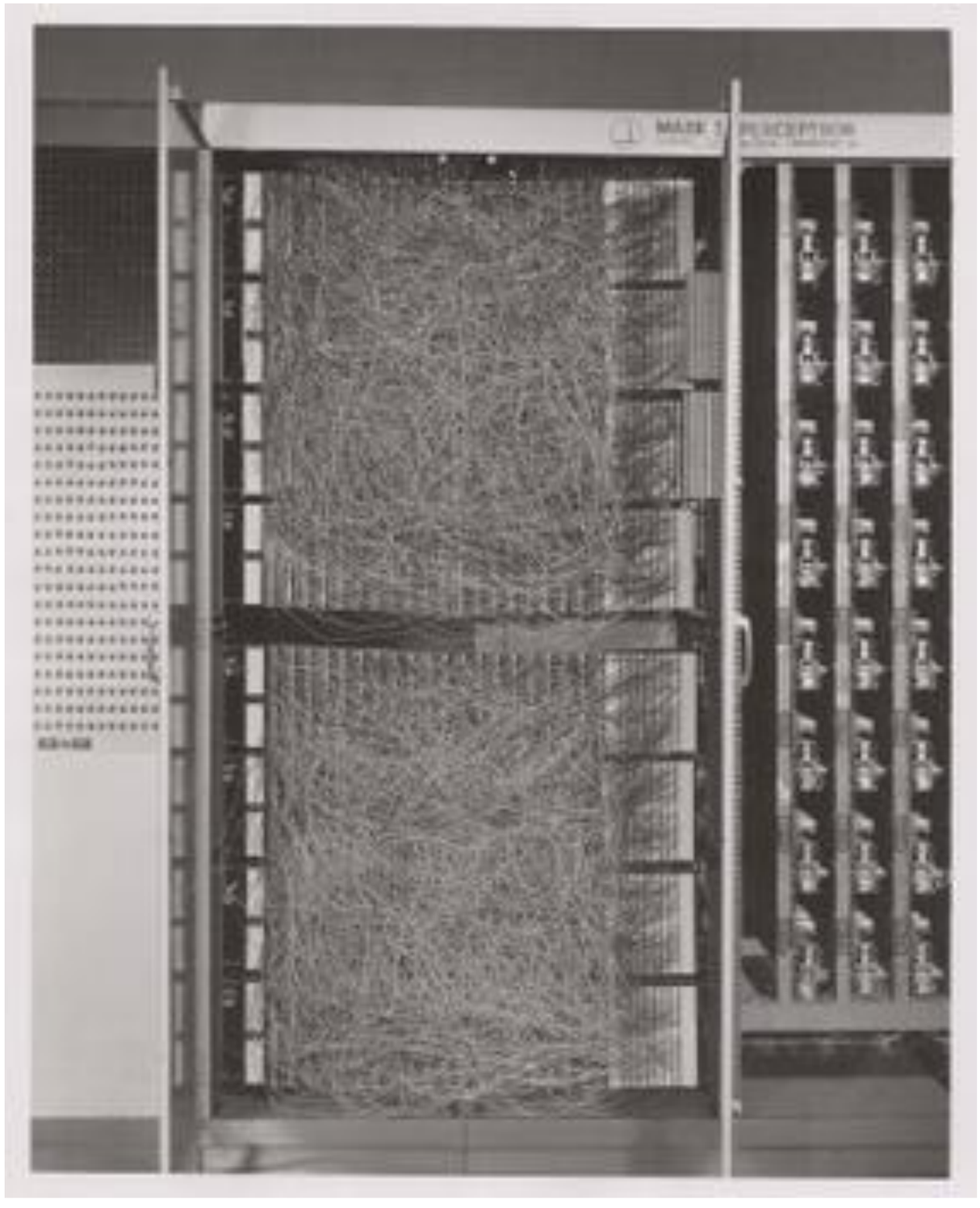
History

- The development of neural networks began early in the history of computing, with the work of McCulloch and Pitts [1943], who studied threshold logic computing as a model of cognitive activity, and the work of Rosenblatt [1958], who designed the Perceptron as a model for information storage and organization for the brain.

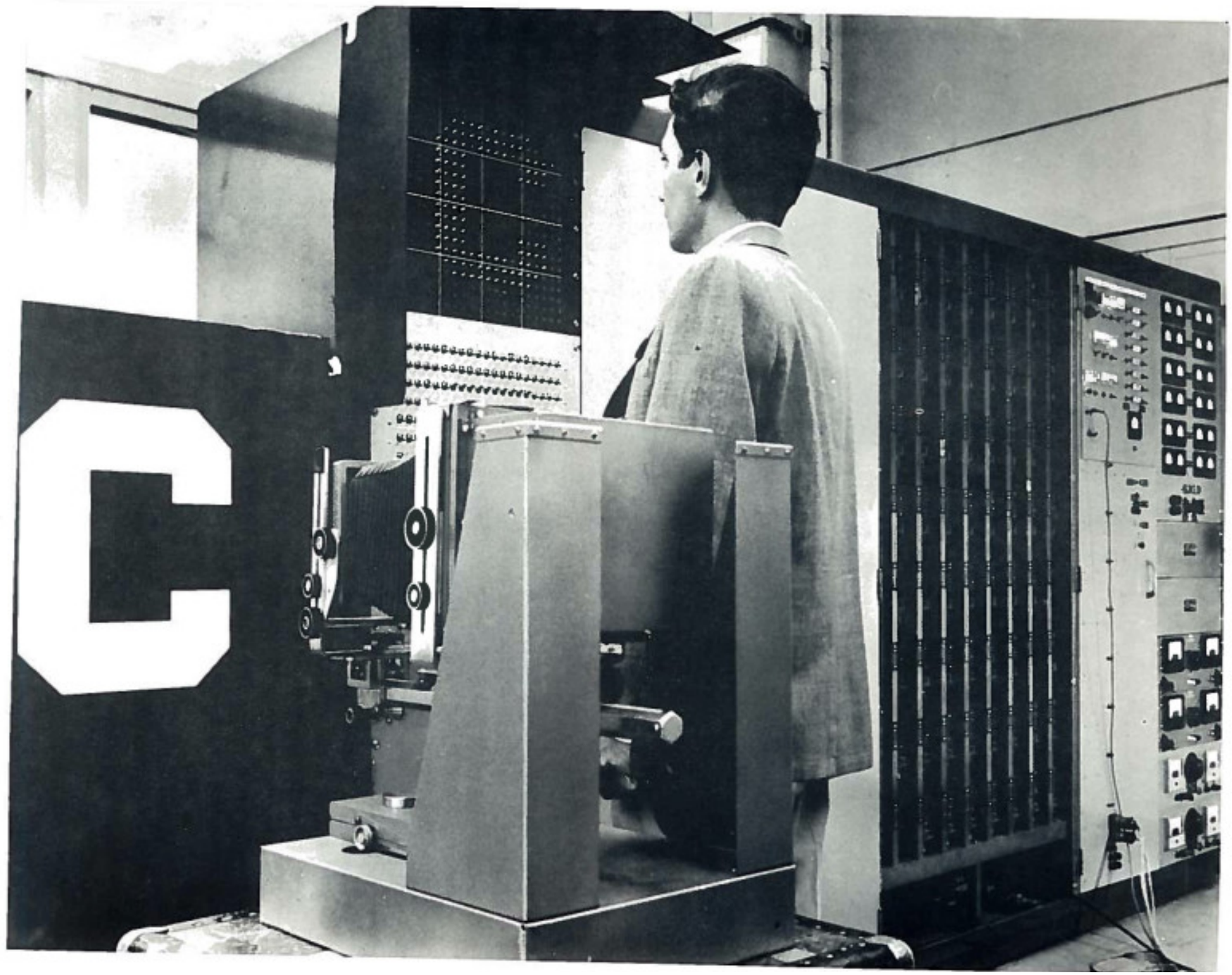
History

- A physical machine was constructed by Rosenblatt, the Mark I Perceptron, which took as input the voltages from a 20x20 array of photodetectors and used motor-driver potentiometers to calculate weighted sums of the inputs in order to visually recognize written digits.

Mark 1 Perceptron



https://digital.library.cornell.edu/catalog/ss:550351



http://www.glass-bead.org/article/machines-that-morph-logic/?lang=enview
(Image sourced from Cornell Library)

History

- Soon after, limitations of the perceptron were outlined by Minsky and Papert [1969], most notably with regard to their inability to solve the "XOR problem" – i.e., recognizing whether or not two inputs were different.

- Knowledge of this limitation is widely regarded as causing the "neural winter" of the subsequent two decades wherein neural network based research (and funding) was largely abandoned.

- However, it later became known that multilayer perceptrons were not subject to such a limitation, and in fact are instead members important class of functions known as universal approximators.

History

- Cybenko [1989], Hornik [1991], and Barron[1994] proved that, under relatively mild constraints, any continuous, bounded function can be approximated to arbitrary accuracy using a multilayer perceptron with a finite number of hidden units.

- The proof however is that such a network exists – given any particular function, determining the requisite number of hidden units is the task of model selection and identifying suitable weights can be tackled using the tools of optimization and machine learning.

UAT – Cybenko, 1989

- Let ϕ be a non-constant, bounded, and monotonically-increasing continuous function. Let f be any continuous, bounded function on the m-dimensional unit hypercube Let $\epsilon > 0$ any positive real.

- There exists an integer N, real numbers v_i, b_i and real vectors $w_i \in R^m$ such that for all inputs $x \in \mathrm{dom}(f)$:

$$|f(x) - \sum_{i=0}^{N} v_i \phi(w_i^T x + b_i)| < \epsilon$$

A neural network is just a nonlinear function controlled by a set of adjustable weights!

Historical References

- *A logical calculus of the ideas immanent in nervous activity,* Warren S McCulloch and Walter Pitts, The bulletin of mathematical biophysics, 1943.

- *The Perceptron: a probabilistic model for information storage and organization in the brain*, Rosenblatt, Psychological Review, 1958.

- *Perceptrons: an introduction to computational geometry*, Marvin L Minsky and Seymour A Papert , MIT Press, 1969.

- *Approximation by Superpositions of a Sigmoid Function*, George Cybenko, Math. Control Systems and Signals, 1989.

- *Approximation Capabilities of Multilayer Feedforward Networks*, Kurt Hornik, Neural Networks, 1991.

- *Approximation and estimation bounds for artificial neural networks*, Andrew Barron, Machine Learning, 1994

The Structure of Neural Networks

- We begin with a specific kind of neural network called a **multilayer perceptron** (MLP).

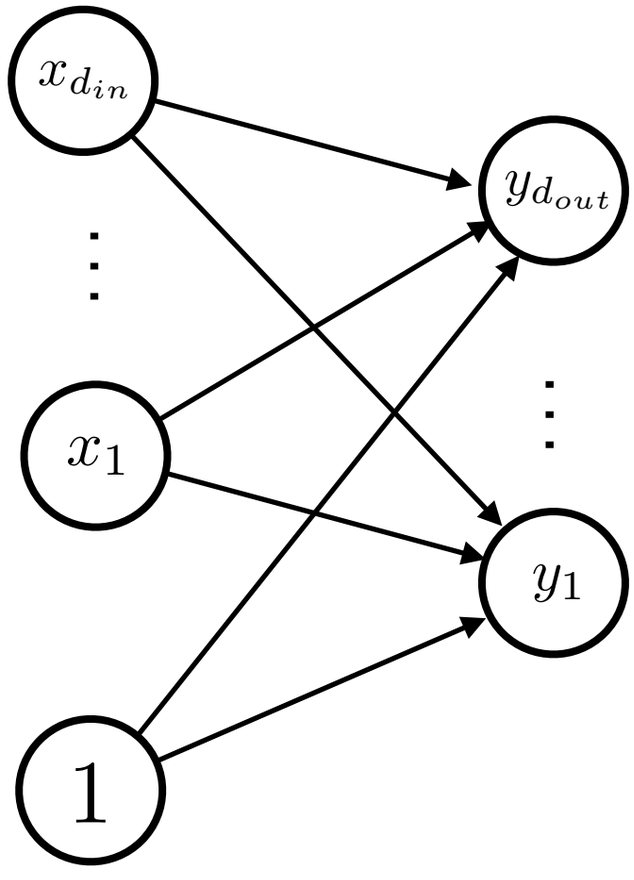
- As with all NNs, a MLP is a function from inputs x to outputs y controlled by a set W of parameters:

$$f_W(x) : \mathcal{R}^{d_{in}} \rightarrow \mathcal{R}^{d_{out}}$$

Structure: A Perceptron

Can only learn linearly separable classes.
E.g., cannot learn XOR.

X1	X2	Y
F	F	F
F	T	T
T	F	T
T	T	F



$$y_k = \sum_j w_{kj} x_j$$

Structure: How many layers?

- The original perceptron was effectively a linear classifier. In order to get the universal approximation properties, we need at least one hidden layer of **hidden units**.

Structure: Hidden Units

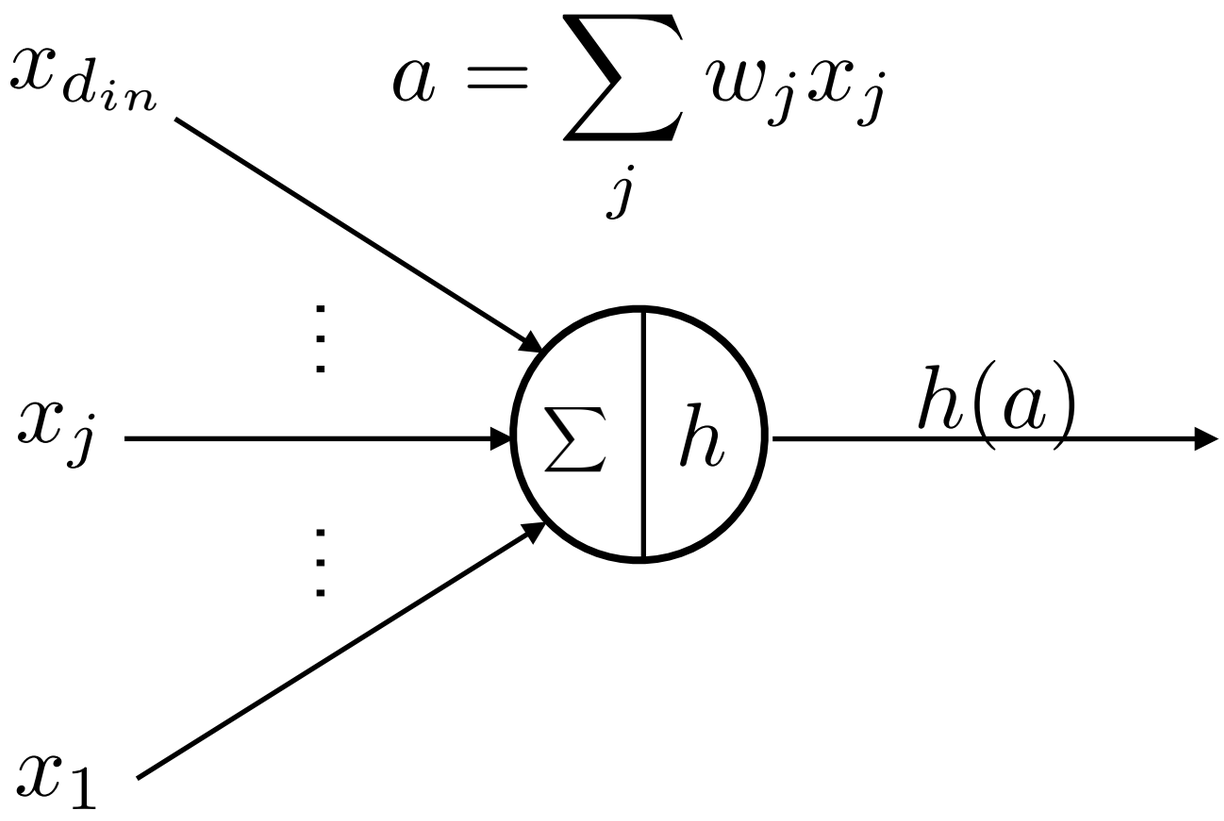
- A multilayer perceptron has at least one layer of **hidden units**, which compute a nonlinear function of their weighted inputs:

$$h(a)$$

- The weighted sum is called the **activation** of the unit.

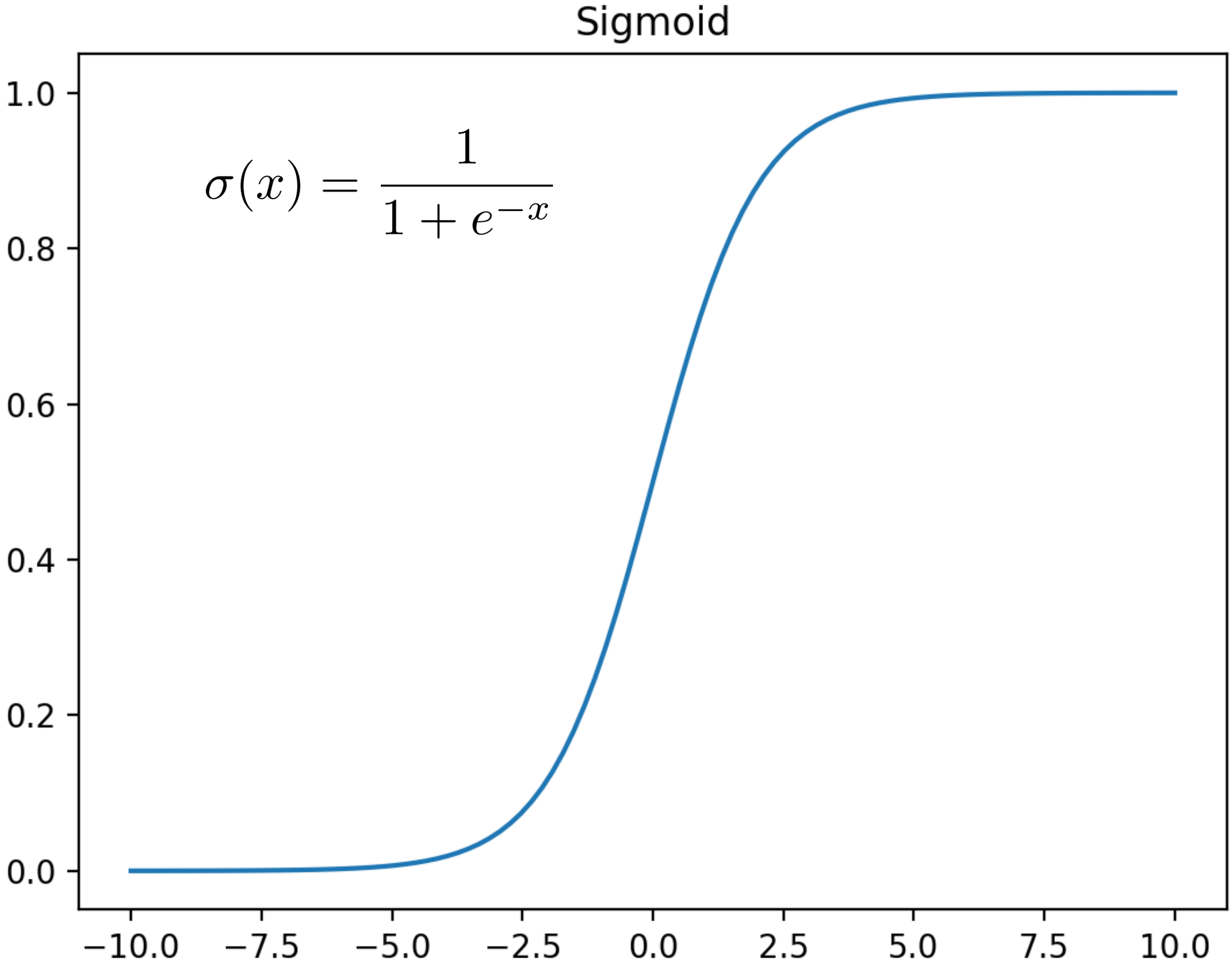
$$a = \sum_j w_j x_j$$

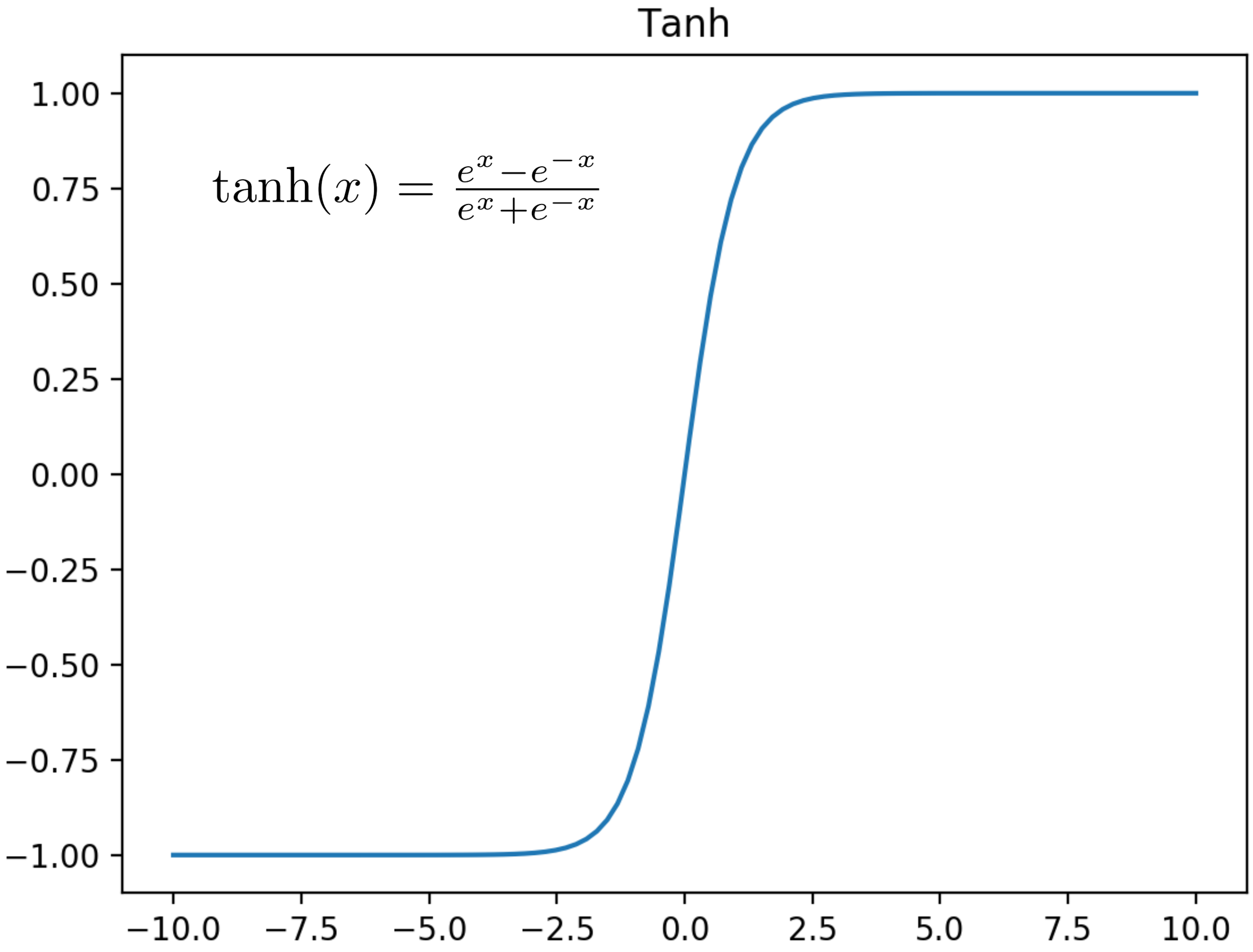
Structure: Hidden Units

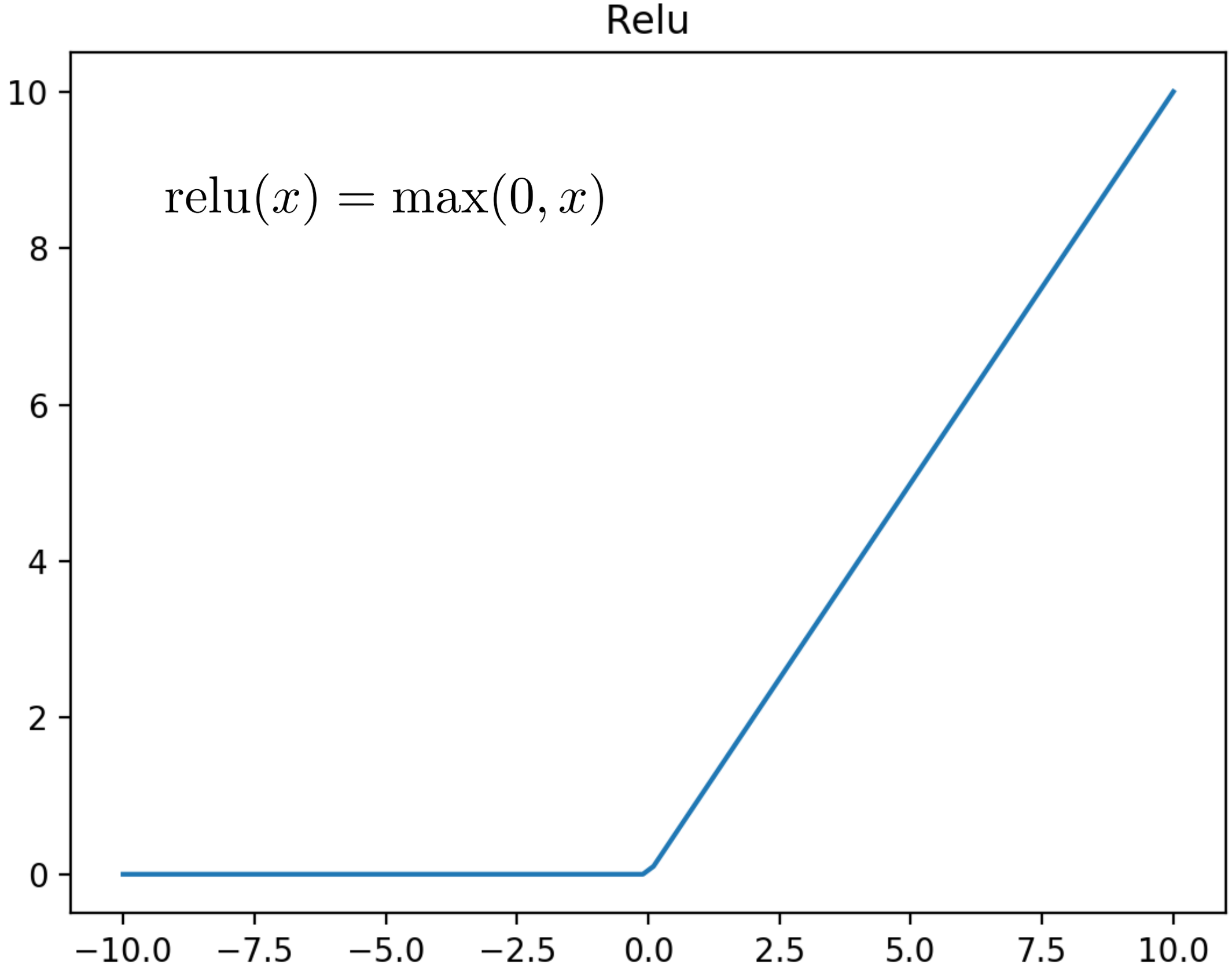


The Structure: Activation Functions

- Example activation functions:
  - $\sigma(x) = \frac{1}{1+e^{-x}}$ ← Sigmoid
  - $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ ← Hyperbolic Tangent
  - $\max(0, x)$ ← Rectified Linear Unit (ReLU)


Sigmoid
$\sigma(x) = \frac{1}{1 + e^{-x}}$
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Tanh
$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Relu
$\mathrm{relu}(x) = \max(0, x)$
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

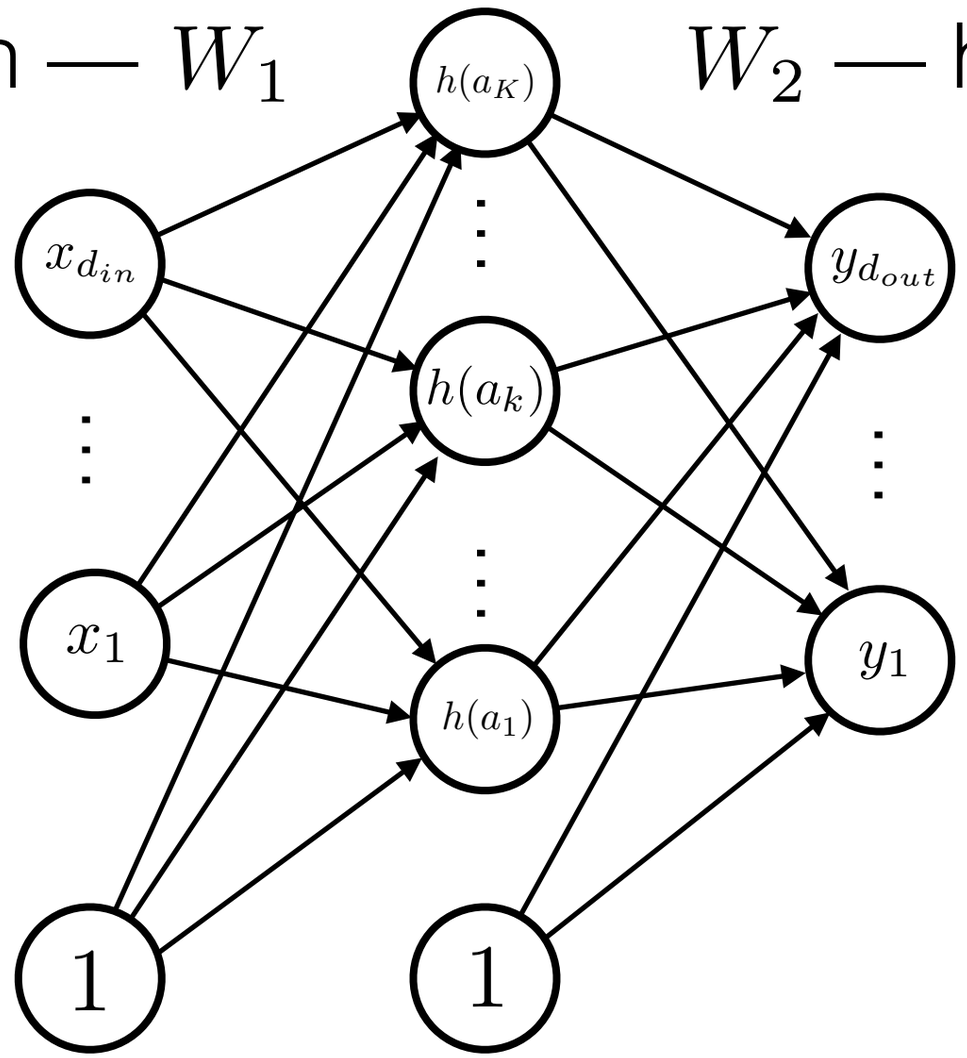
Structure: A single-layer MLP

- Let’s look at an MLP with a single hidden layer.

Structure: A single-layer MLP

weights from input to hidden — W_1

weights from W_2 — hidden to output



$$y = W_2 h(W_1 x)$$

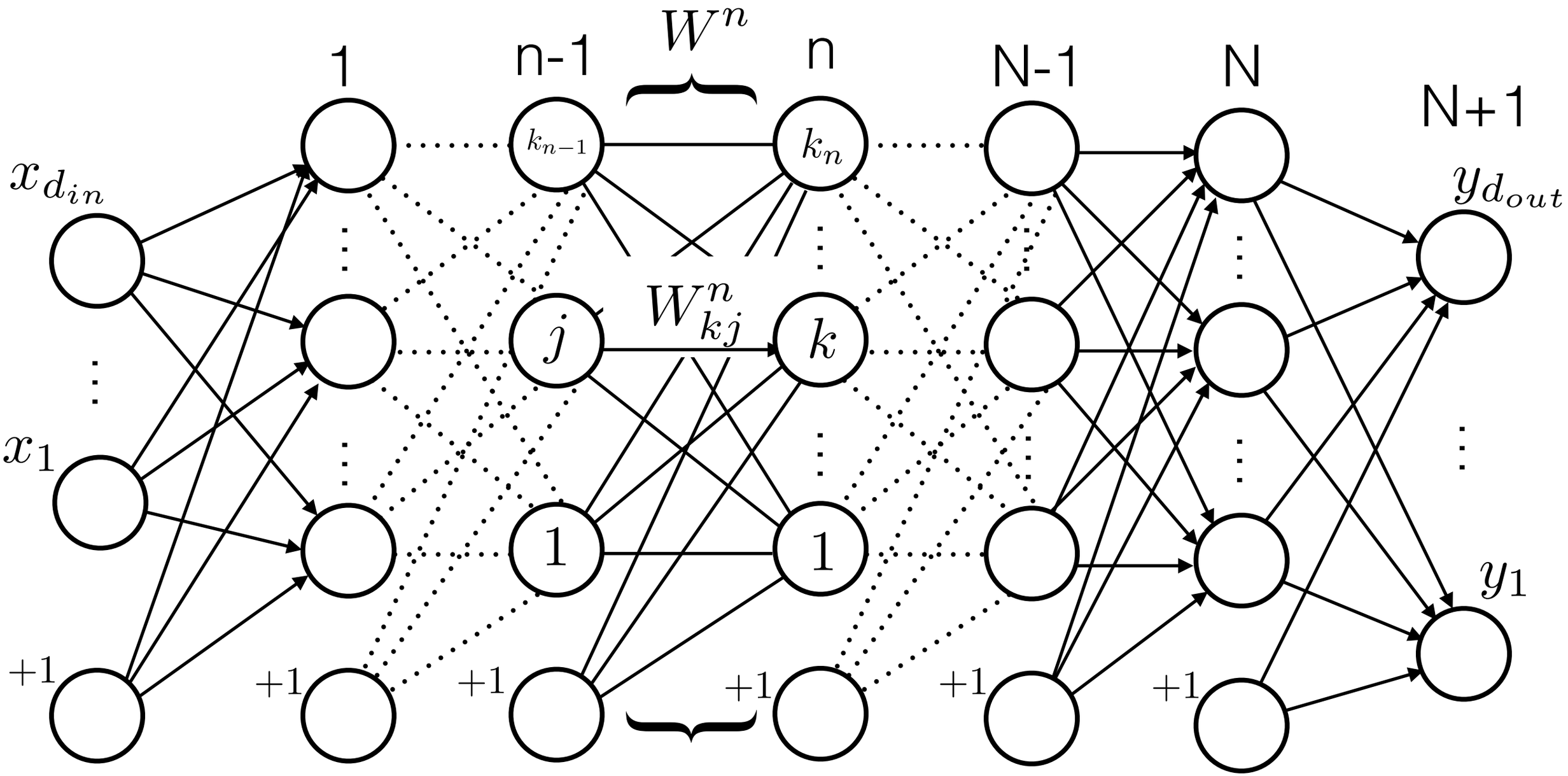
$$\dim(W_1) = K \times (d_{in} + 1)$$

$$\dim(W_2) = d_{out} \times (K + 1)$$

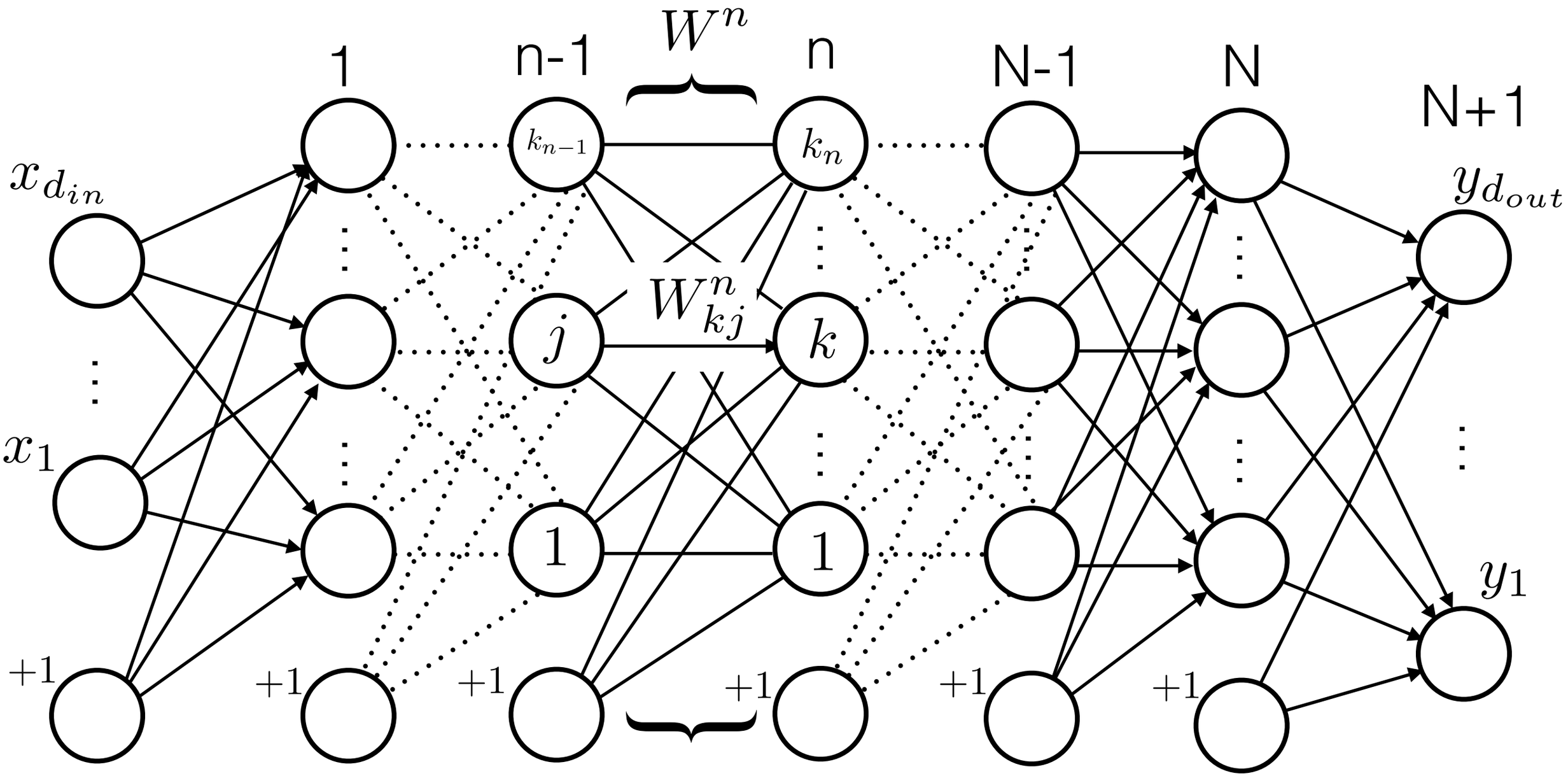
Extension – Additional Layers

- We can extend the multilayer perceptron to one with arbitrarily many layers. We must decide:
  - the number of layers,
  - the number of hidden units in each layer, and
  - the activation function.

Structure: A many-layer MLP



Structure: A many-layer MLP



Structural Equations of MLPs

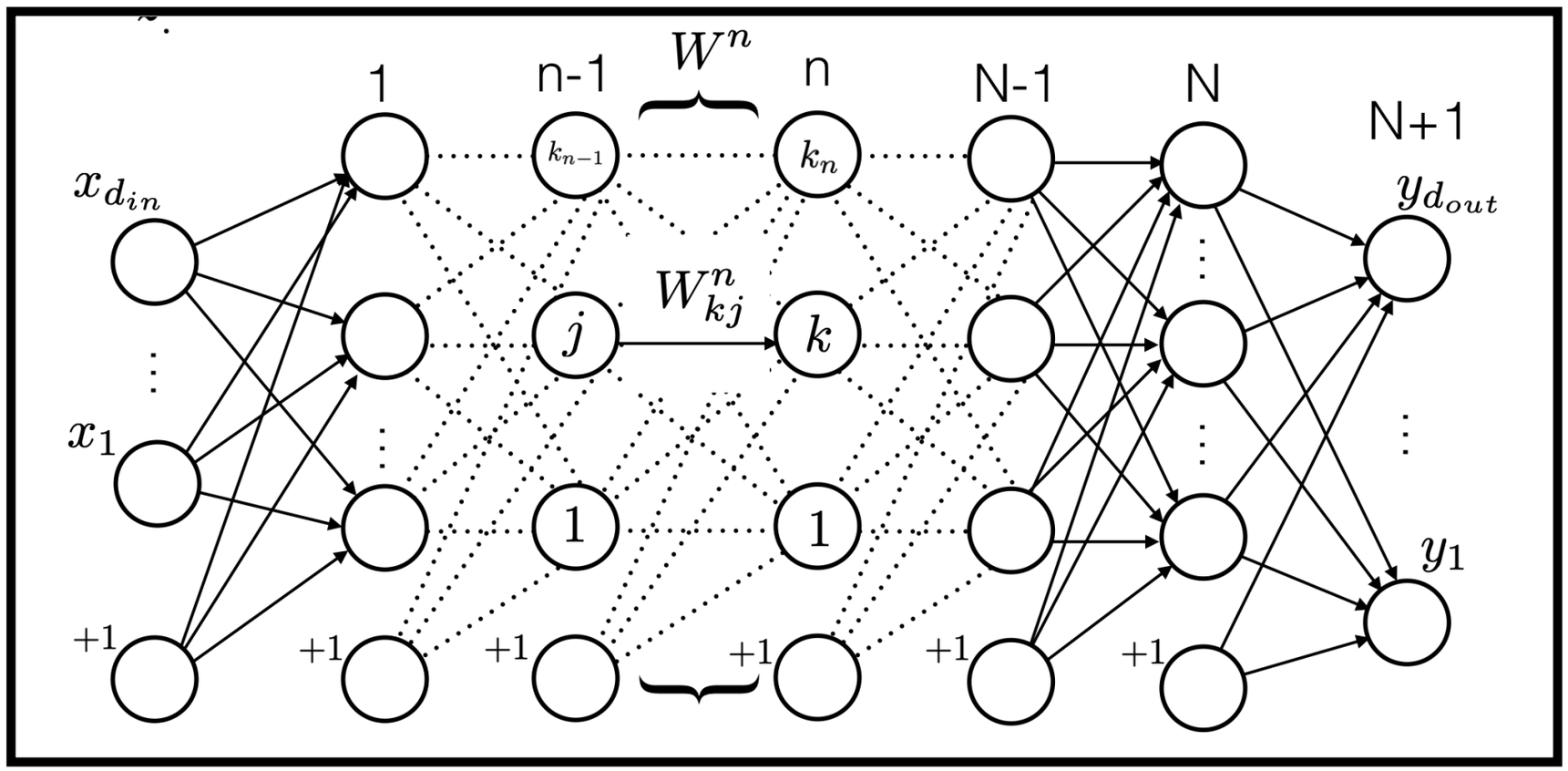
Unit Outputs

$$z$$

$$a_k^n = \sum_{j=1}^{k_j} W_{kj}^n \cdot z_j^{n-1}$$

Activations

$$a$$



Network Input

$$z^0 = x$$

Propagation

$$z^n = h(a^n)$$

$$a^n = w^n \cdot z^{(n-1)}$$

Network Output

$$a^{(N+1)} = y$$

Learning

- Clearly, neural networks with *random* parameters aren't exactly great.

- We need learning — a technique for minimizing the number of errors made by a predictive algorithm by (approximately) solving the fundamental equation:

$$\arg\min_{w} E[L(w)]$$

Learning

- If we can quantify how the performance of our neural network depends on the specific parameter settings, then we can use local search techniques to find better parameters from randomly initialized values.

Learning: Loss Functions

- A **loss function** quantifies the loss, or error, when using a neural network to approximate a function.

- Different loss functions are used for regression and classification tasks. A great variety of loss functions have been studied, but the most popular are:

- Sum of Squares

- Cross-Entropy

Learning: Loss Functions

- Typical loss functions for $\hat{y} = f(x)$:
  - $I(y = \hat{y})$ ← 0/1 loss
  - $\sum_k (y_k - \hat{y}_k)^2$ ← Sum-of-Squares
  - $-\sum_k y_k \log(\hat{y}_k)$ ← Cross-Entropy

Learning: Gradient Descent

- Given an available set of training data and a choice of loss function, one approach to learning is to start with a random set of parameters and iteratively improve them with **gradient descent.**

Learning: Gradient Descent

- Training a neural network means to adjust the parameters (weights) such that the error on the training data is minimal.

- Therefore we take the **gradient of the error (loss) function with respect to the weights**, and iteratively search for a local minima.

Learning: Gradient Descent

Choose an initial set of parameters w_0.

Until converged do:

$$w_{t+1} := w_t - \gamma \nabla E(w_t)$$

Either a fixed number of iterations or by gradient magnitude. E.g., $(\nabla E(w_t))^2 < \epsilon$

Scale the update by given **learning rate.**

Find weights that minimize **expected loss**.

Learning: Gradient Descent

- How to calculate $\nabla E(w)$?
  - Numerically
  - Analytically

Learning: The Gradient

- What about a many-dimensional gradient?

- Such as of prediction error with respect to our parameters?

Learning: The Gradient

$$\nabla E(w) =< \frac{\partial E}{\partial w_N}, \ldots, \frac{\partial E}{\partial w_0} >$$

$$w + \Delta w_n =< w_N, \ldots, w_{n+1}, w_n + \Delta w_n, w_{n-1}, \ldots, w_0 >$$

$$\frac{\partial E}{\partial w_n} = \lim_{\Delta w_n \to 0} \frac{E(w + \Delta w_n) - E(w)}{\Delta w_n}$$

Calculating the Gradient

- Consider using finite differences to evaluate ∇E at a particular point w. I.e., for every dimension of w we will approximate the derivative along the dimension by incrementing by
  Δ along a basis vector e_i :

$$\frac{E(w + \Delta e_i) - E(w)}{\Delta}$$

- Issues: slow, approximate, redundant
  Improvement — backprop (soon)

Backprop

- Clearly there are a lot of redundant calculations when evaluating the gradient using finite differences.

- Enter: backpropagation.

How to Train a Neural Network

- Choose error function $E(W)$.
- Find best parameters by solving $\arg\min_W E(W)$.
- Use random initialization to get W_0.
- Loop until converged or satisfied:

$$W_{t+1} := W_t + \gamma \cdot \nabla E(W)$$

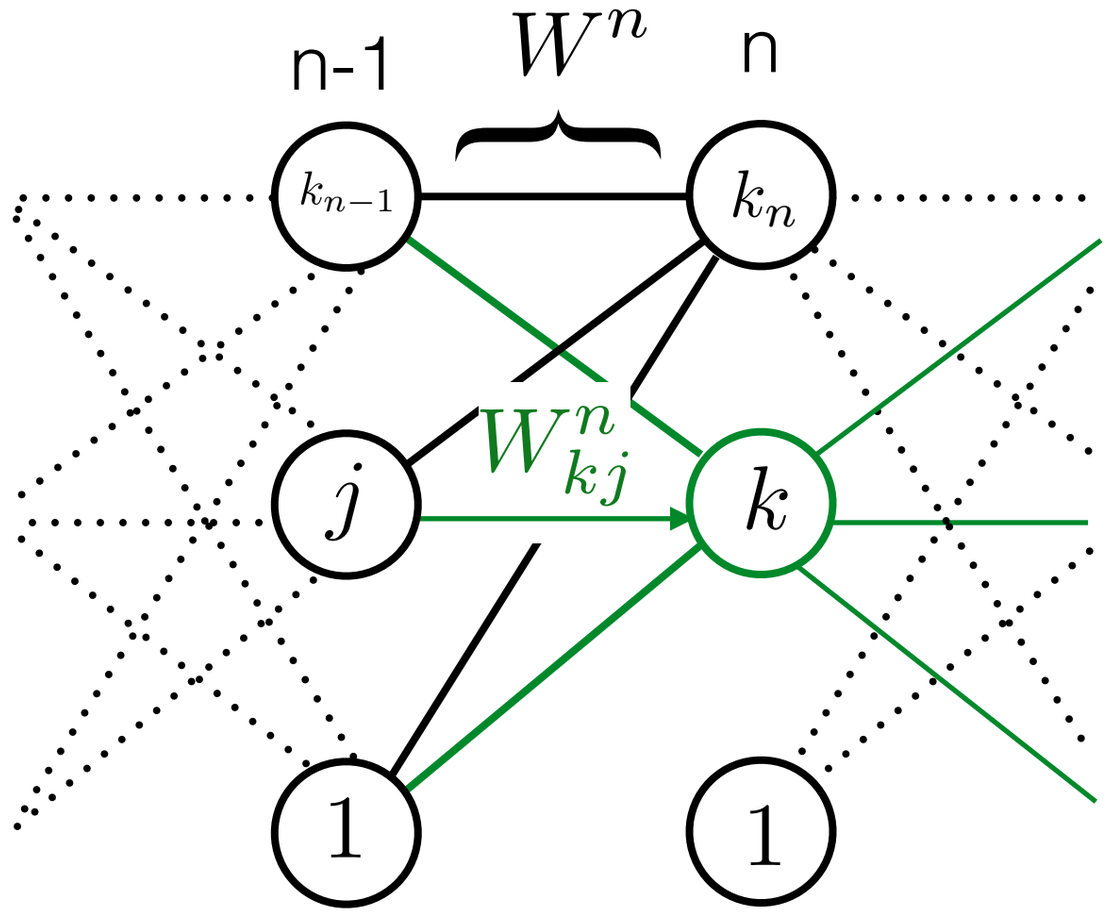
- To find $\nabla E(W)$, use backprop.

- Recall, the gradient $\nabla E(W)$ is a vector of partials

$$\frac{\partial E}{\partial W_{kj}^{n}}$$ for each parameter in the network.

The Backprop Equations

$$\frac{\partial E}{\partial W^n_{kj}} = \frac{\partial E}{\partial a^n_k} \cdot \frac{\partial a^n_k}{\partial W^n_{kj}}$$

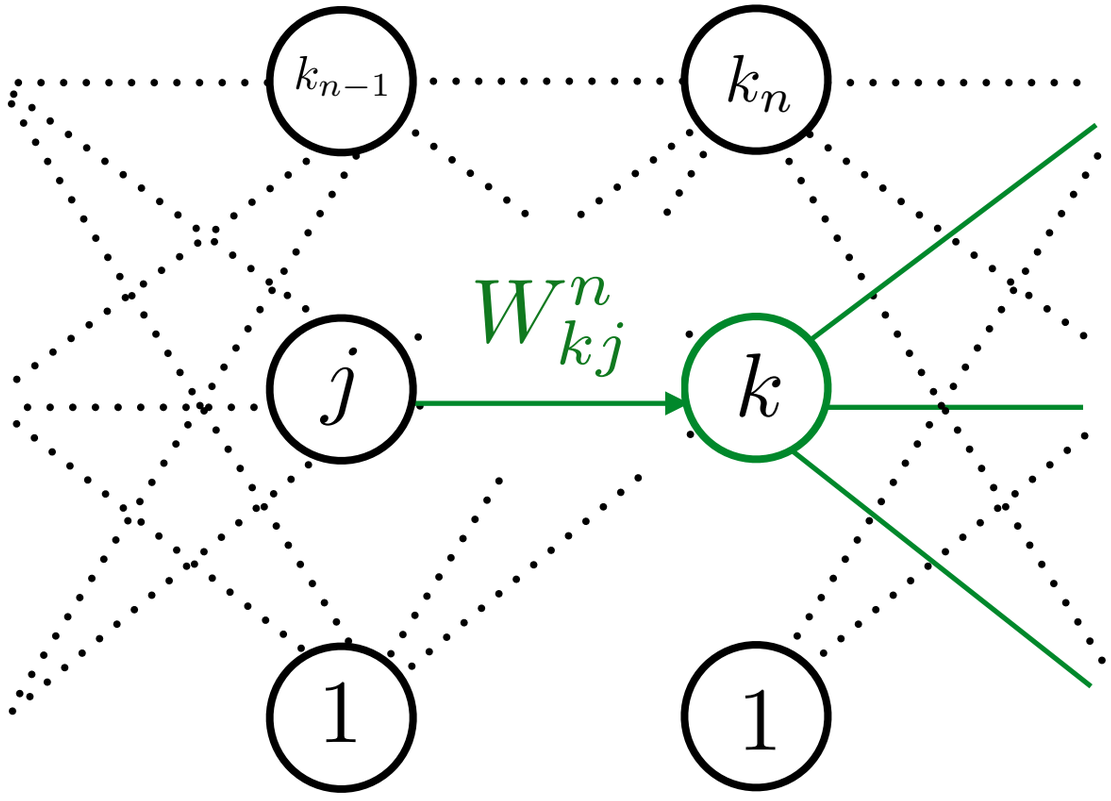


change in error wrt parameter	=	(change in error wrt unit activation)	x	(change in unit activation wrt parameter)

Backprop – Delta Introduction

$$\frac{\partial E}{\partial W_{kj}^n} = \frac{\partial E}{\partial a_k^n} \cdot \frac{\partial a_k^n}{\partial W_{kj}^n}$$

Define $\delta_k^n := \frac{\partial E}{\partial a_k^n}$



Then we must solve for:

δ_k^n — 1

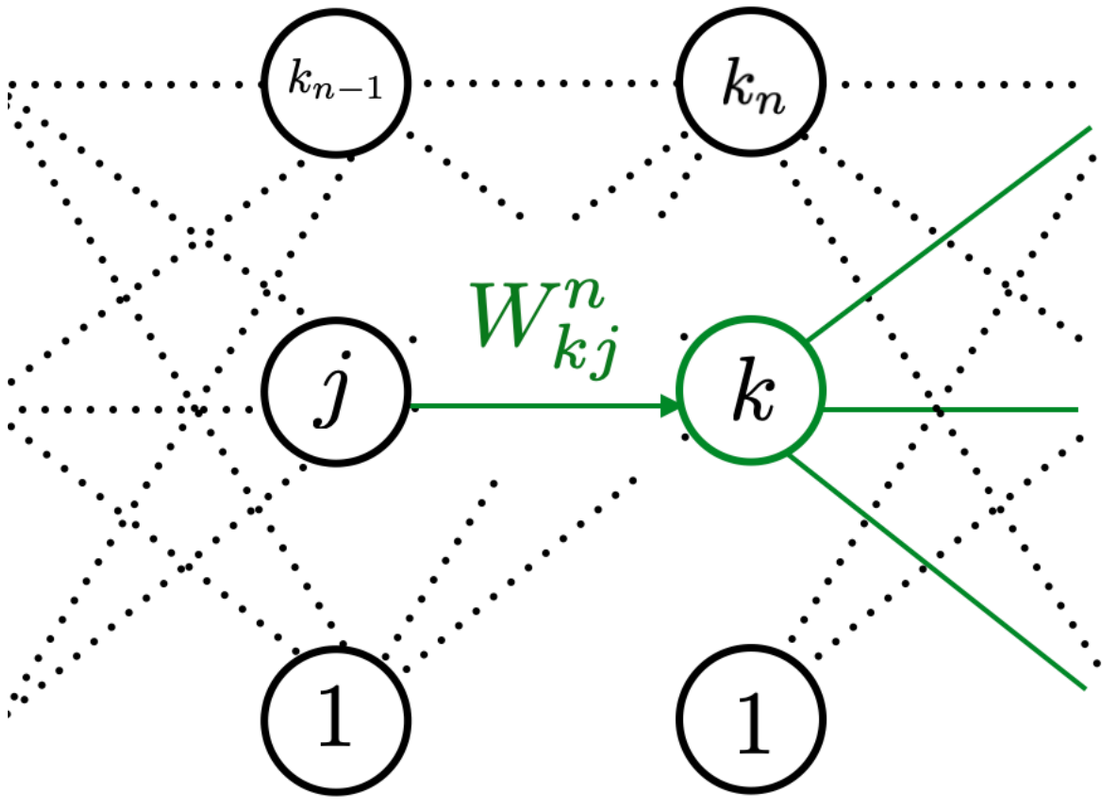
and

$\frac{\partial a_k^n}{\partial W_{kj}^n}$ — 2

Backprop – Solving for Delta

$$\delta_k^n := \frac{\partial E}{\partial a_k^n} = \sum_i \frac{\partial E}{\partial a_i^{(n+1)}} \cdot \frac{\partial a_i^{(n+1)}}{\partial a_k^n}$$

This is the most difficult substitution for me to accept.



It is the total derivative.

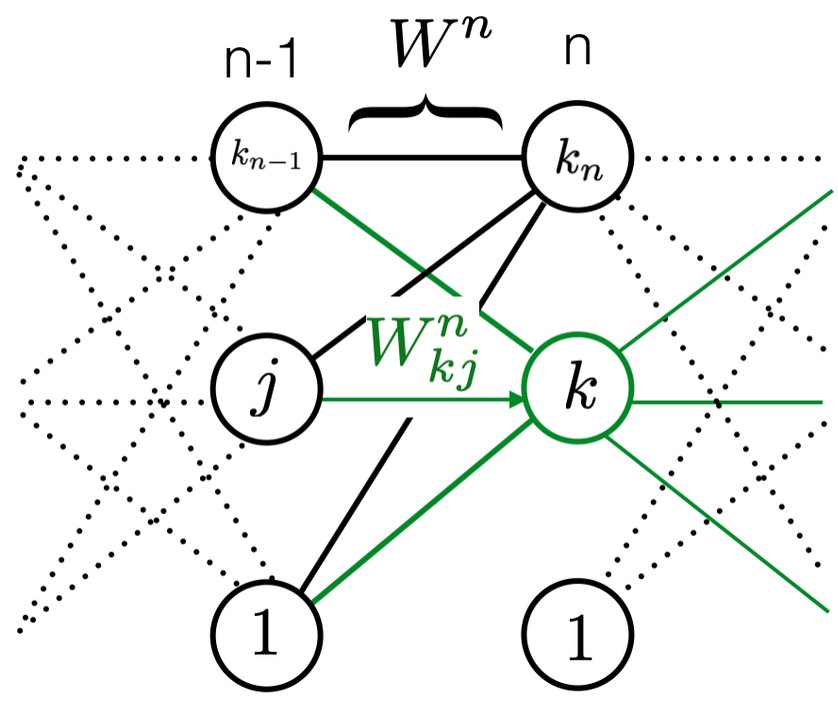
Assuming linearity for small enough changes.

Backprop – Solving for Delta

$$\delta_k^n := \frac{\partial E}{\partial a_k^n} = \sum_i \frac{\partial E}{\partial a_i^{(n+1)}} \cdot \frac{\partial a_i^{(n+1)}}{\partial a_k^n} = \sum_i \delta_i^{(n+1)} \frac{\partial a_i^{(n+1)}}{\partial a_k^n}$$

$$\frac{\partial a_i^{(n+1)}}{\partial a_k^n} = \frac{\partial}{\partial a_k^n}[\sum_{k'} W_{ik'}^{(n+1)} h(a_{k'}^n)] = W_{ik}^{(n+1)} h'(a_k^n)$$

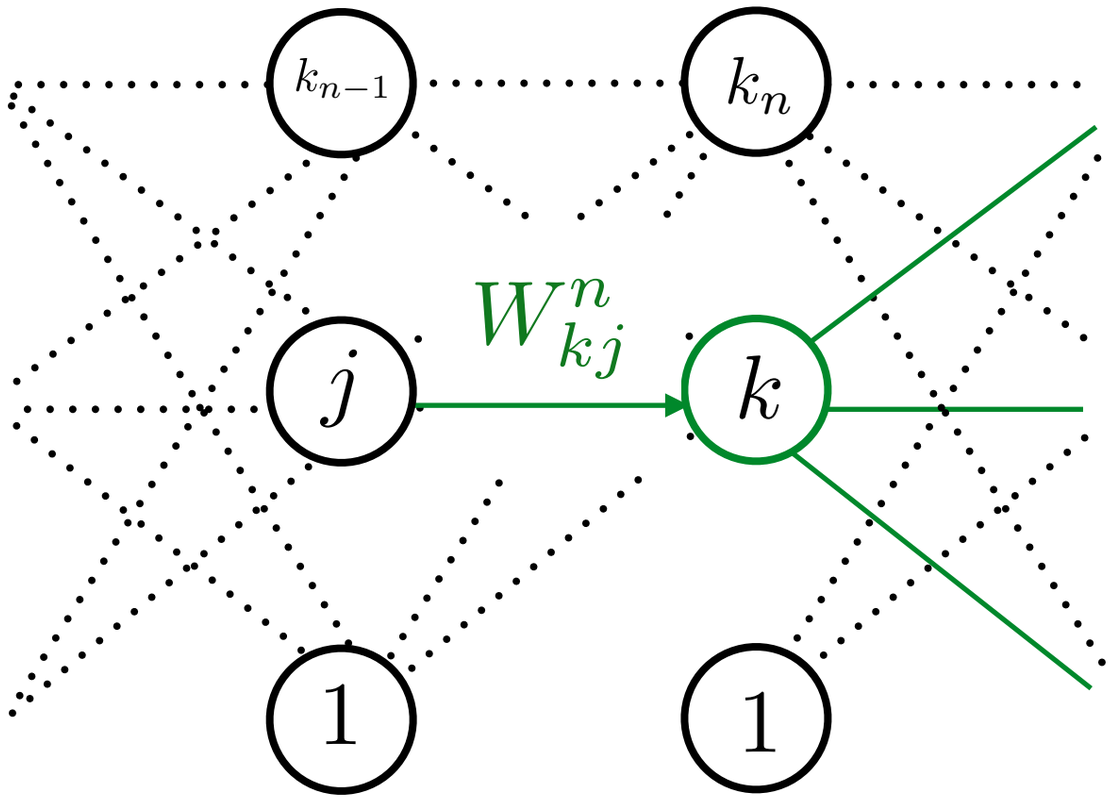
$$\therefore \ \delta_k^n = h'(a_k^n) \sum_i \delta_i^{(n+1)} W_{ik}^{(n+1)}$$



Backprop – Delta Introduction

$$\frac{\partial E}{\partial W_{kj}^n} = \frac{\partial E}{\partial a_k^n} \cdot \frac{\partial a_k^n}{\partial W_{kj}^n}$$

Define $\delta_k^n := \frac{\partial E}{\partial a_k^n}$



Then we must solve for:

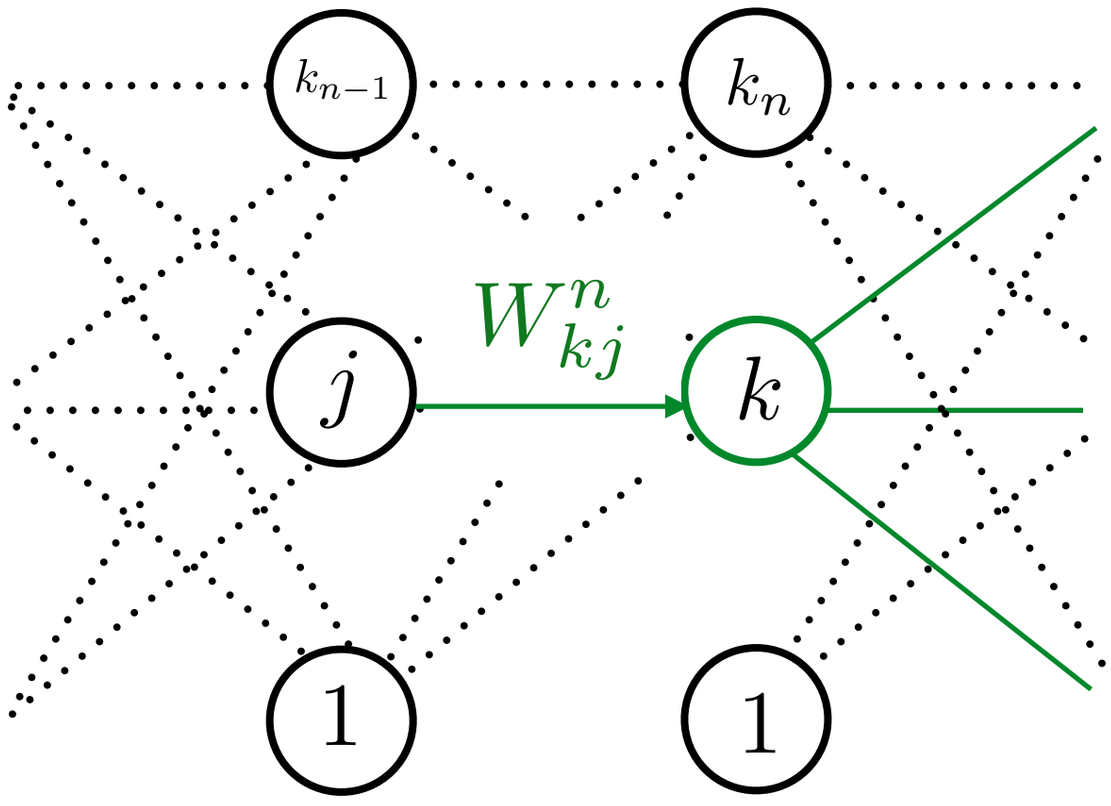
δ_k^n 1

and

$\frac{\partial a_k^n}{\partial W_{kj}^n}$ 2

Backprop – Solving Part 2

$$\frac{\partial a_k^n}{\partial W_{kj}^n} = \frac{\partial}{\partial W_{kj}^n}[\sum_{j'} W_{kj'}^n h(a_{j'}^{(n-1)})] = h(a_j^{(n-1)})$$



Recall:

$$a_k^n = \sum_{j'} W_{kj'}^n h(a_{j'}^{(n-1)})$$

The Backprop Equations

$$\frac{\partial E}{\partial W_{kj}^n} = \delta_k^n \cdot h(a_j^{(n-1)})$$

where

$$\delta_k^n = h'(a_k^n) \sum_i \delta_i^{(n+1)} W_{ik}^{(n+1)}$$

The Rest of the Algorithm

- To calculate the gradient for a single example (x,y), we first **forward evaluate** the network on x, caching the activations, then we can set $\delta^{(N+1)}$ using the MLP outputs and real targets, then recursively evaluate the remaining parameters in a **backward** fashion.

$$\frac{\partial E}{\partial W^n_{kj}} = \delta^n_k \cdot h(a^{(n-1)}_j)$$

$$\delta^n_k = h'(a^n_k) \sum_i \delta^{(n+1)}_i W^{(n+1)}_{ik}$$

$$\delta^{(N+1)}_k = \frac{\partial E}{\partial \hat{y}_k}$$

Intro to Neural Networks – Summary

- Universal Approximators
- Architectures
- Loss functions, softmax, cross-entropy, sum-squares.
- Gradient descent
- Back-propagation of errors.

Project 4 – Machine Learning

- Due last day of class… NEXT WEDNESDAY!

- Let's discuss it. If