APL 745
Supervised Learning: Feedforward neural network

Dr. Rajdip Nayek

Block V, Room 418D

Department of Applied Mechanics

Indian Institute of Technology Delhi

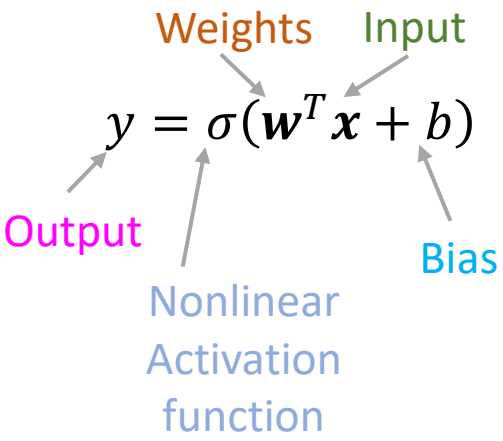
E-mail: rajdipn@am.iitd.ac.in

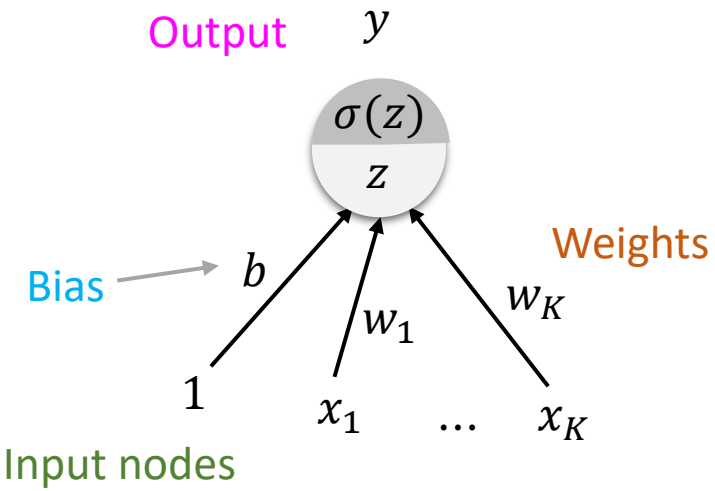
Learning goals

- Know the basic terminology for neural networks
- Given the weights and biases for a neural network, be able to compute its output from its input
- Know the types of activation functions used
- Be able to hand-design the weights of a neural net to represent functions like XOR
- Understand why shallow neural nets are expressive, and why this isn't necessarily very interesting

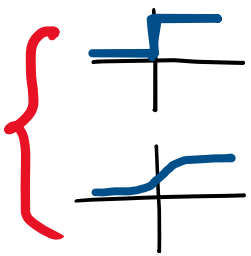
Overview

- In the first lecture, we introduced a neuron-like processing unit





- For **linear regression**: $\sigma(z) = z$
- For **linear classification**: $\sigma(z) =$



→ Perceptron

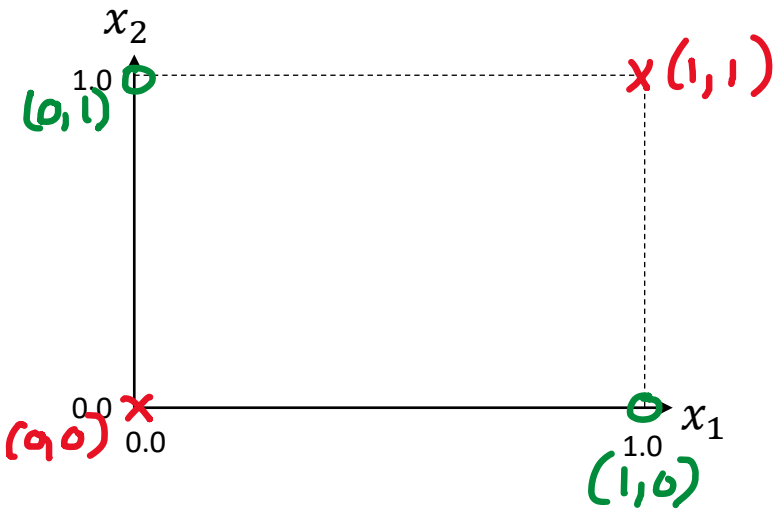
→ Sigmoid neuron

(Softmax for multi-class)

- A **neural network** is just a combination of lots of these units

Overview

- We noted that there are functions that can't be represented by linear models
  - Ex: Linear regression can not represent quadratic functions
  - Ex: Linear classifiers can not represent the XOR gate



- We also saw one particular way of avoiding the above issue
  - By defining **nonlinear features** or basis functions $\phi(\boldsymbol{x})$

Limitations of linear models

- While linear models are convenient, the way of defining nonlinear features is not very satisfying
  - The nonlinear features $\phi(\boldsymbol{x})$ have to be chosen and fixed manually and that requires expert domain knowledge
  - It might require a large number of nonlinear features to represent certain functions

- **Question**: Can we work with unrestricted non-linear models?
  - We don't want to work with a finite set of basis functions
  - We don't want to completely specify the basis functions beforehand

- **Answer**: Yes, now we are going to see how we can free ourselves from this restriction

Flexible nonlinear models

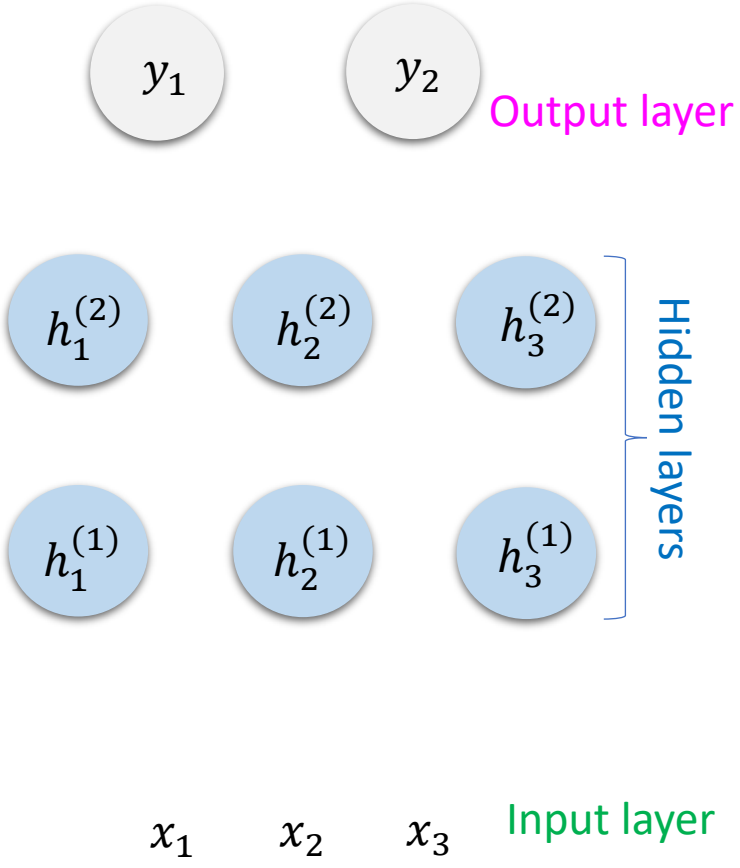
- **Idea 1**: Instead of choosing a fixed set of basis functions apriori, we can choose our basis functions to depend on the data and then allow a very large number (sometimes an infinite number) of them to be included without paying a computational price. This kind approach falls under **kernel** methods (e.g. **Support Vector Machines**)
  - Was a popular idea in the 2000s upto around 2010

- **Idea2**: Deep learning → Learn non-linear basis functions (e.g. **Multilayer Neural Networks**)
  - Became very popular from around 2010
  - Even though the number of basis functions are finite and fixed, the parameters of the basis functions are adapted (or learned) so as to effectively vary them based on the data

A multilayer neural network

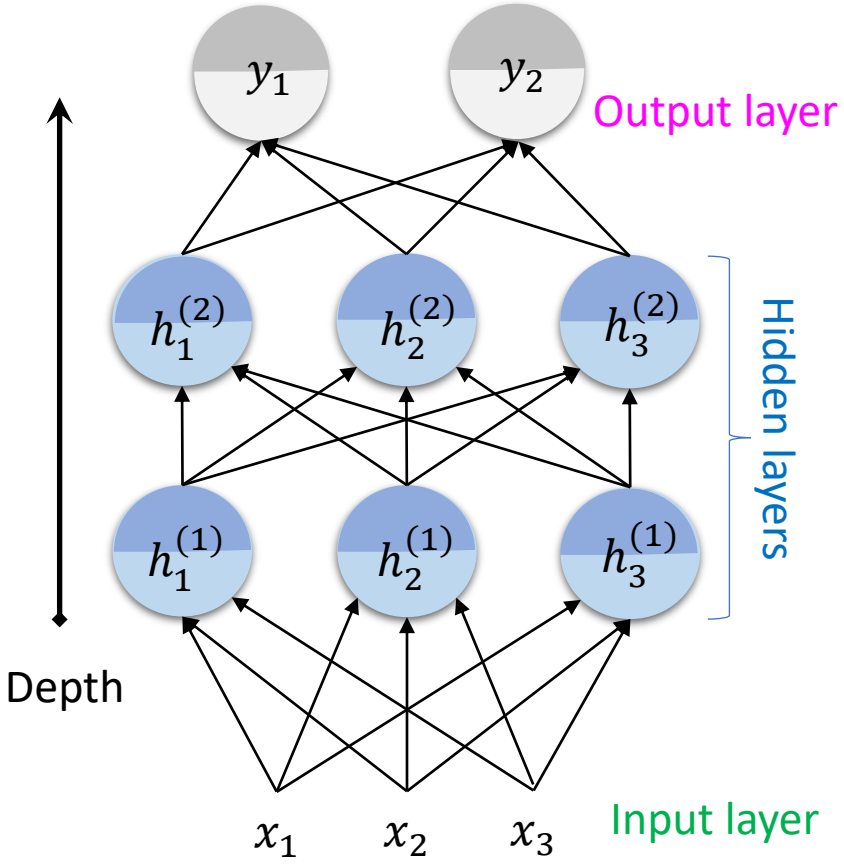
Terminology

- **Multilayer neural network** is a network of neuron-like units arranged in **layers**. This network in picture has 4 layers
- The layer containing the input features (x_1, x_2, x_3) is called the **input layer (Layer 0)**
- The two middle layers containing 3 units are called **hidden layers**
  - The outputs of the first hidden layer are denoted by $h_1^{(1)}, h_2^{(1)}, h_3^{(1)}$
  - The outputs of the second hidden layer are denoted by $h_1^{(2)}, h_2^{(2)}, h_3^{(2)}$
- The final layer is called the **output layer** (here two output units are shown)



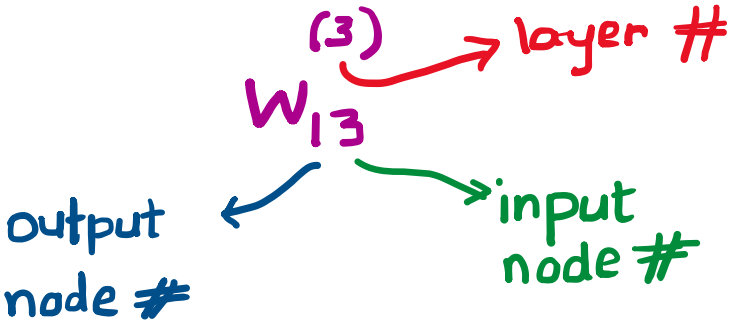
A multilayer neural network

- Each neuron in the hidden layers and the output layer can be split into two parts: pre-activation and activation
- **Fully connected** ← every unit in one layer is connected to every unit in the next layer
- **Depth** ← the number of computational layers
  - Three-layered network
  - Input layer is layer 0
- **Width** ← the number of units in a layer
  - Three-unit wide hidden layers (each one could have different width as well)
  - Three-unit wide input layer
  - Two-unit wide output layer
- Information flows in only forward direction, from input nodes through hidden nodes to output nodes, hence called **feed-forward neural network (FNN)**. No cycles or loops

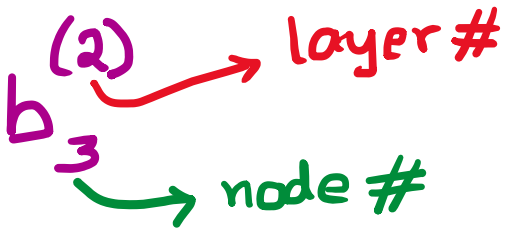


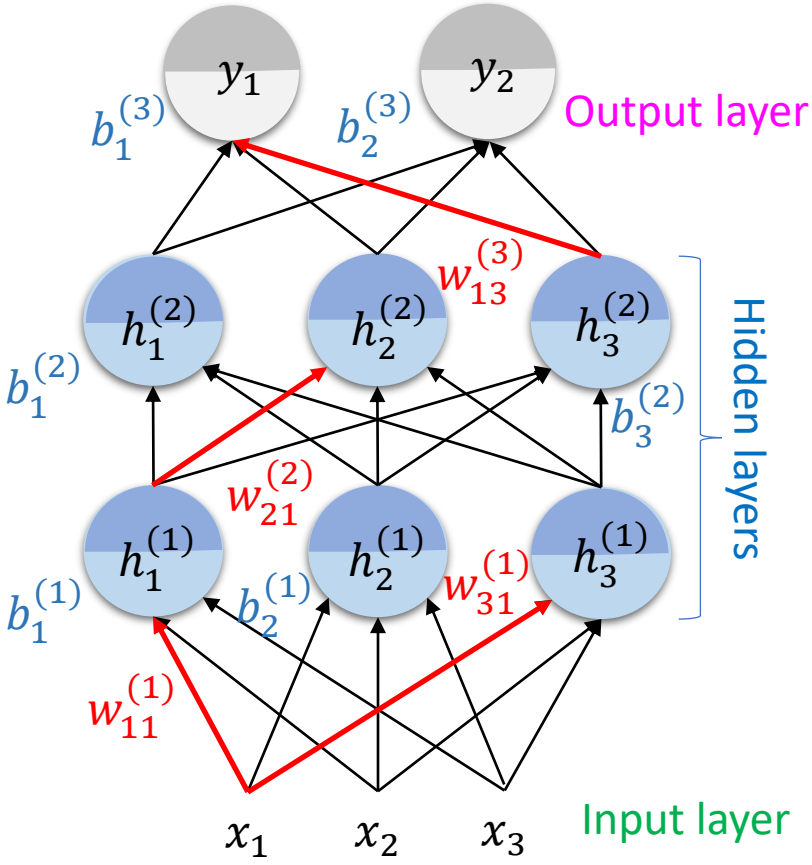
Computation in an FNN

- **Each link (arrow) is associated with a weight**
  - $w_{ij}^{(k)}$ ← k^{th} layer, i^{th} output, j^{th} input



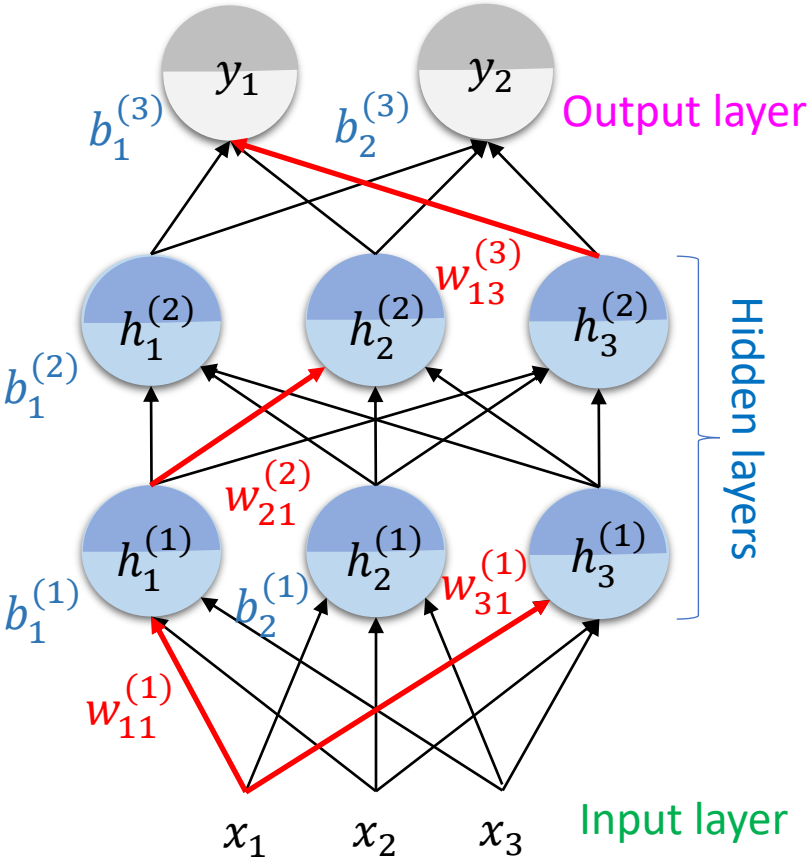
- **Each circular node has its own bias**
  - $b_i^{(k)}$ ← k^{th} layer, i^{th} unit





Computation in the layers of FNN

- Each link (arrow) is associated with a weight
  - $w_{ij}^{(k)} \leftarrow k^{\text{th}}$ layer, i^{th} output, j^{th} input
- Each circular node has its own bias (not shown)
  - $b_i^{(k)} \leftarrow k^{\text{th}}$ layer, i^{th} unit
- **Network computations**:
  - $h_i^{(1)} = \sigma^{(1)}\left(\sum_j w_{ij}^{(1)} x_j + b_i^{(1)}\right)$ (activation; Preactivation)
  - $h_i^{(2)} = \sigma^{(2)}\left(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}\right)$
  - $y_i = \sigma^{(3)}\left(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)}\right)$
- Note that $\sigma^{(1)}$ and $\sigma^{(2)}$ can be different because different layers may have different activation functions



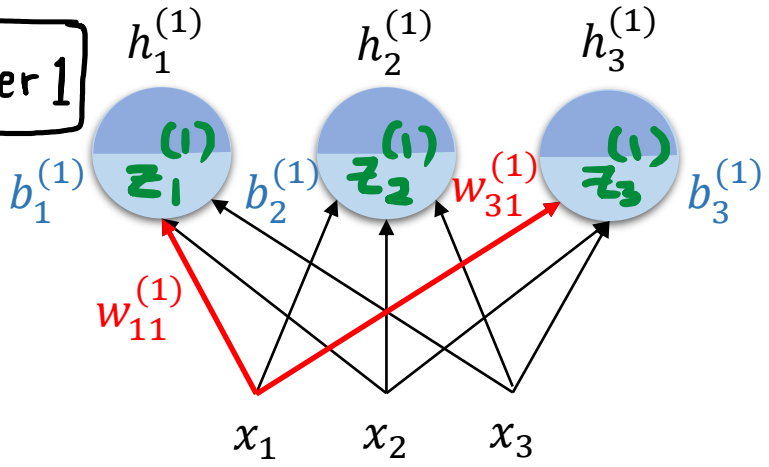
Computation in the layers of FNN

- Compute preactivation

$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b_1^{(1)}$$

$$z_2^{(1)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3 + b_2^{(1)}$$

$$z_3^{(1)} = w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + w_{33}^{(1)} x_3 + b_3^{(1)}$$



- Compute activation

$$h_1^{(1)} = \sigma^{(1)}(z_1^{(1)})$$

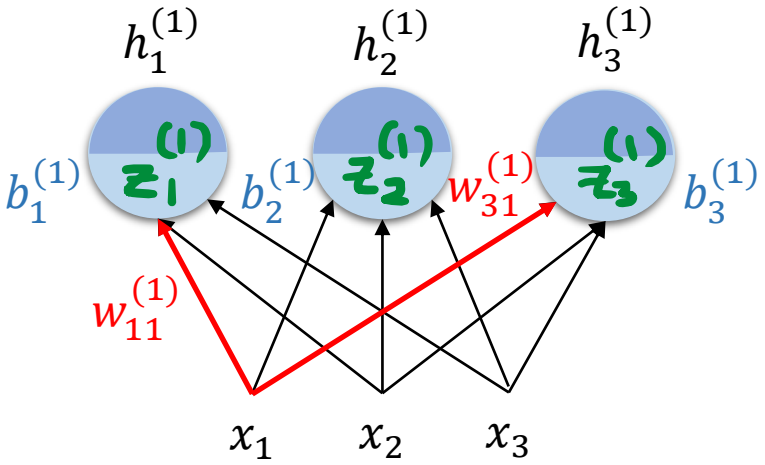
$$h_2^{(1)} = \sigma^{(1)}(z_2^{(1)})$$

$$h_3^{(1)} = \sigma^{(1)}(z_3^{(1)})$$

Computation in the layers of FNN

- Compute preactivation

$$\begin{pmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{pmatrix} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix}$$



$$\underset{\sim}{z}^{(1)} = \underset{\approx}{W}^{(1)} \underset{\sim}{x} + \underset{\sim}{b}^{(1)}$$

← Vectorized

- Compute activation

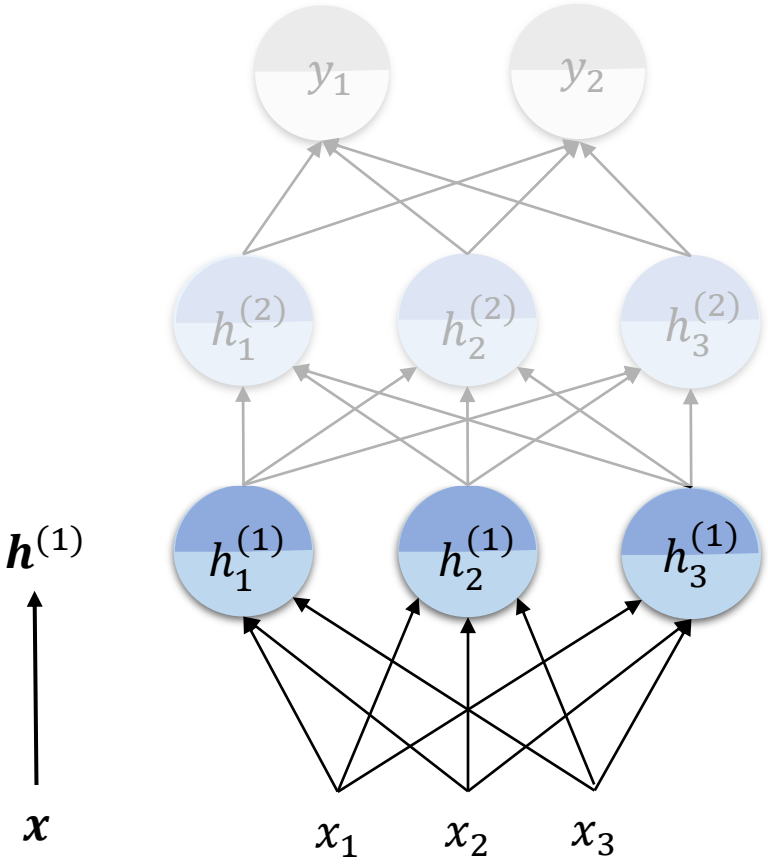
$$\underset{\sim}{h}^{(1)} = \sigma^{(1)}\left(\underset{\sim}{z}^{(1)}\right)$$

$$\begin{pmatrix} h_1^{(1)} \\ h_2^{(1)} \\ h_3^{(1)} \end{pmatrix} = \begin{pmatrix} \sigma^{(1)}(z_1^{(1)}) \\ \sigma^{(1)}(z_2^{(1)}) \\ \sigma^{(1)}(z_3^{(1)}) \end{pmatrix}$$

Computation in the layers of FNN

- $\boldsymbol{x} = [x_1 \quad x_2 \quad x_3]^T, \boldsymbol{y} = [y_1 \quad y_2]^T$
- $\boldsymbol{h}^{(1)} = \left[h_1^{(1)} \quad h_2^{(1)} \quad h_3^{(1)}\right]^T$
- $\boldsymbol{b}^{(1)} = \left[b_1^{(1)} \quad b_2^{(1)} \quad b_3^{(1)}\right]^T$
- $\mathbf{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{13}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix}$
- **Vectorized computation for 1st layer**

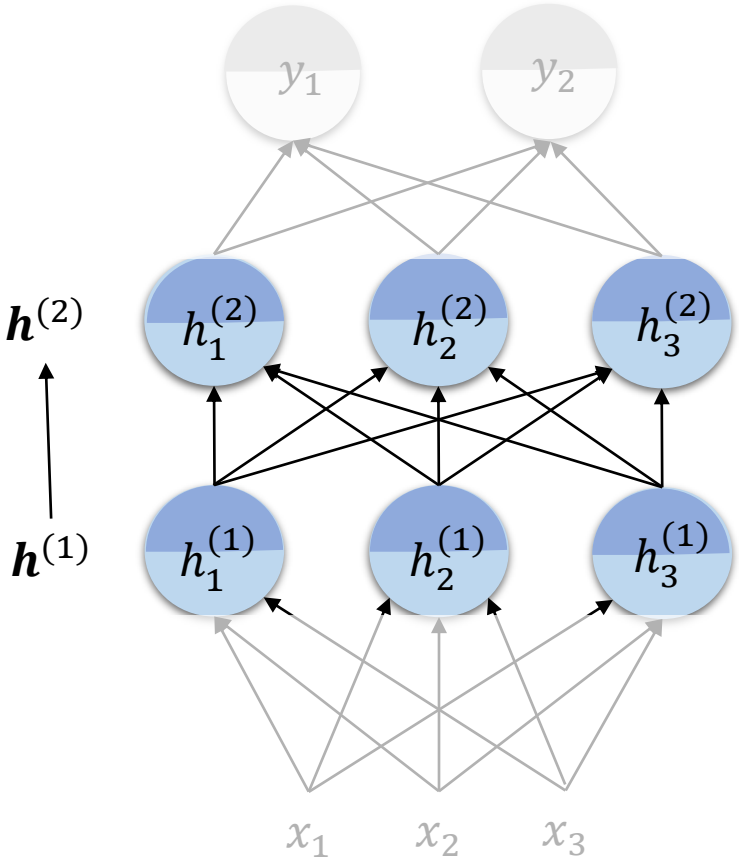
$$\boldsymbol{h}^{(1)} = \sigma^{(1)}\left(\mathbf{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$



Computation in the layers of FNN

- $\boldsymbol{h}^{(1)} = \begin{bmatrix} h_1^{(1)} & h_2^{(1)} & h_3^{(1)} \end{bmatrix}^T$
- $\boldsymbol{h}^{(2)} = \begin{bmatrix} h_1^{(2)} & h_2^{(2)} & h_3^{(2)} \end{bmatrix}^T$
- $\boldsymbol{b}^{(2)} = \begin{bmatrix} b_1^{(2)} & b_2^{(2)} & b_3^{(2)} \end{bmatrix}^T$
- $\mathbf{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{23}^{(2)} \\ w_{13}^{(2)} & w_{32}^{(2)} & w_{33}^{(2)} \end{bmatrix}$
- **Vectorized computation for 2nd layer**

$$\boldsymbol{h}^{(2)} = \sigma^{(2)}\left(\mathbf{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

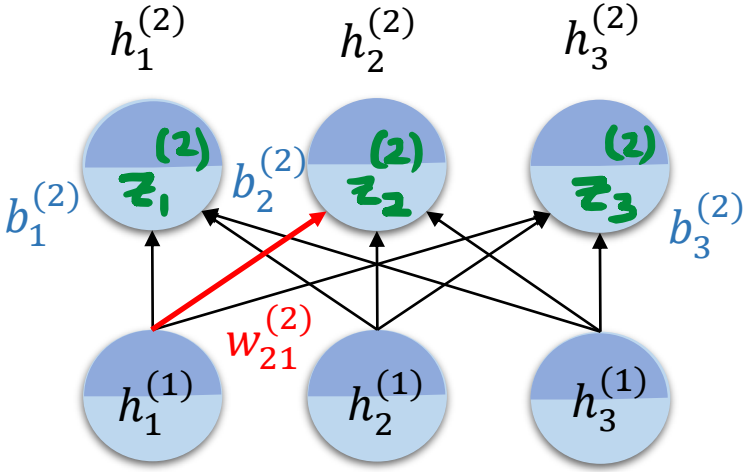


Computation in the layers of FNN

- Compute preactivation

$$\begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{pmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(1)} \\ w_{21}^{(2)} & w_{22}^{(2)} & w_{23}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} & w_{33}^{(2)} \end{bmatrix} \begin{pmatrix} h_1^{(1)} \\ h_2^{(1)} \\ h_3^{(1)} \end{pmatrix} + \begin{pmatrix} b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(2)} \end{pmatrix}$$

Layer 2



$$\underset{\sim}{z}^{(2)} = \underset{\approx}{W}^{(2)} \underset{\sim}{h}^{(1)} + \underset{\sim}{b}^{(2)}$$

← Vectorized

- Compute activation

$$\underset{\sim}{h}^{(2)} = \sigma^{(2)}\left(\underset{\sim}{z}^{(2)}\right)$$

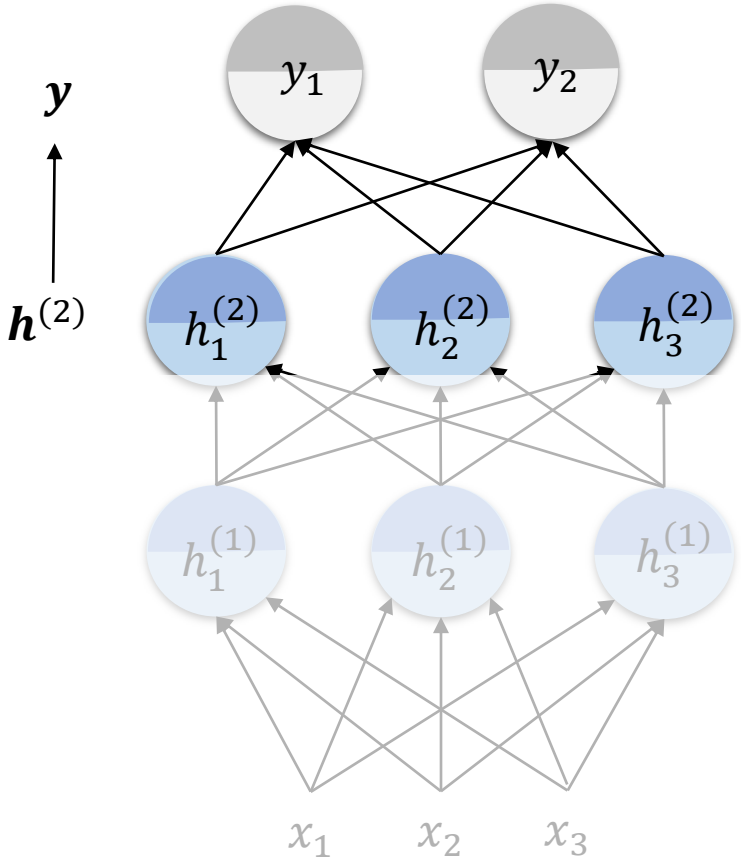
$$\begin{pmatrix} h_1^{(2)} \\ h_2^{(2)} \\ h_3^{(2)} \end{pmatrix} = \begin{pmatrix} \sigma^{(2)}(z_1^{(2)}) \\ \sigma^{(2)}(z_2^{(2)}) \\ \sigma^{(2)}(z_3^{(2)}) \end{pmatrix}$$

Vectorized computation in FNN

- $\boldsymbol{x} = [x_1 \quad x_2 \quad x_3]^T, \boldsymbol{y} = [y_1 \quad y_2]^T$
- $\boldsymbol{h}^{(2)} = \left[h_1^{(2)} \quad h_2^{(2)} \quad h_3^{(2)}\right]^T$
- $\boldsymbol{b}^{(3)} = \left[b_1^{(3)} \quad b_2^{(3)} \quad b_3^{(3)}\right]^T$
- $\mathbf{W}^{(3)} = \begin{bmatrix} w_{11}^{(3)} & w_{12}^{(3)} & w_{13}^{(3)} \\ w_{12}^{(3)} & w_{22}^{(3)} & w_{23}^{(3)} \\ w_{13}^{(3)} & w_{32}^{(3)} & w_{33}^{(3)} \end{bmatrix}$
- **Vectorized computation for 3rd layer**

$$\boldsymbol{y} = \sigma^{(3)}\left(\mathbf{W}^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$

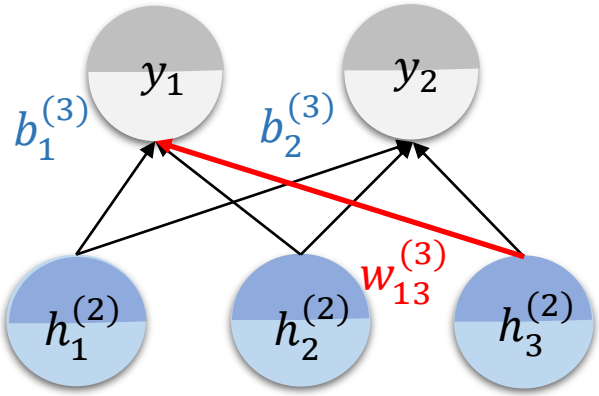
- For the output layer, the type of activation function will depend upon the task at hand. It could be linear for regression or sigmoid for classification.



Computation in the layers of FNN

- Compute preactivation

$$\begin{pmatrix} z_1^{(3)} \\ z_2^{(3)} \end{pmatrix} = \begin{bmatrix} w_{11}^{(3)} & w_{12}^{(3)} & w_{13}^{(3)} \\ w_{21}^{(3)} & w_{22}^{(3)} & w_{23}^{(3)} \end{bmatrix} \begin{pmatrix} h_1^{(2)} \\ h_2^{(2)} \\ h_3^{(2)} \end{pmatrix} + \begin{pmatrix} b_1^{(3)} \\ b_2^{(3)} \end{pmatrix}$$



$$\underset{\sim}{z}^{(3)} = \underset{\approx}{W}^{(3)} \underset{\sim}{h}^{(2)} + \underset{\sim}{b}^{(3)}$$

← Vectorized

- Compute activation

$$\underset{\sim}{y} = \sigma^{(3)}\left(\underset{\sim}{z}^{(3)}\right)$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \sigma^{(3)}(z_1^{(3)}) \\ \sigma^{(3)}(z_2^{(3)}) \end{pmatrix}$$

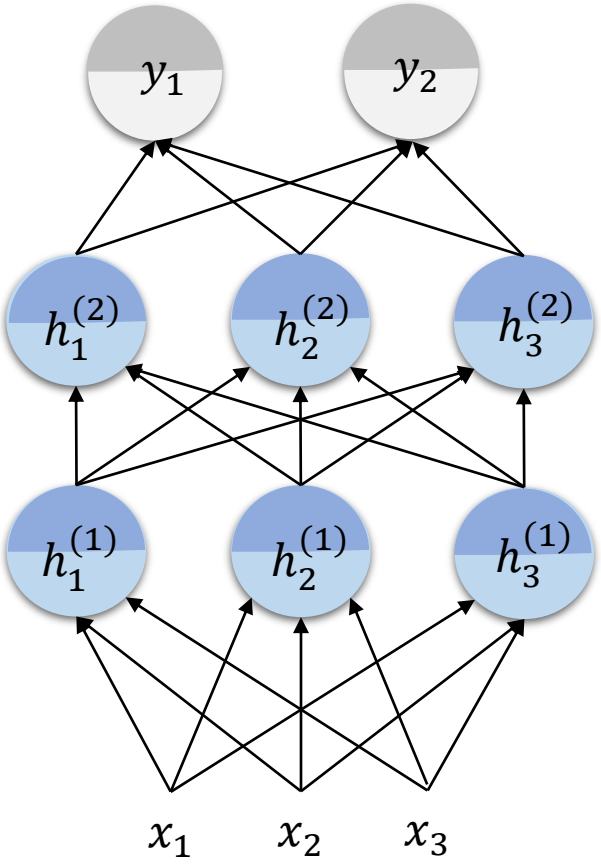
Vectorized computation in FNN

- **Compact vectorized representation**

$$\boldsymbol{h}^{(1)} = \sigma^{(1)}\left(\mathbf{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = \sigma^{(2)}\left(\mathbf{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{y} = \sigma^{(3)}\left(\mathbf{W}^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$



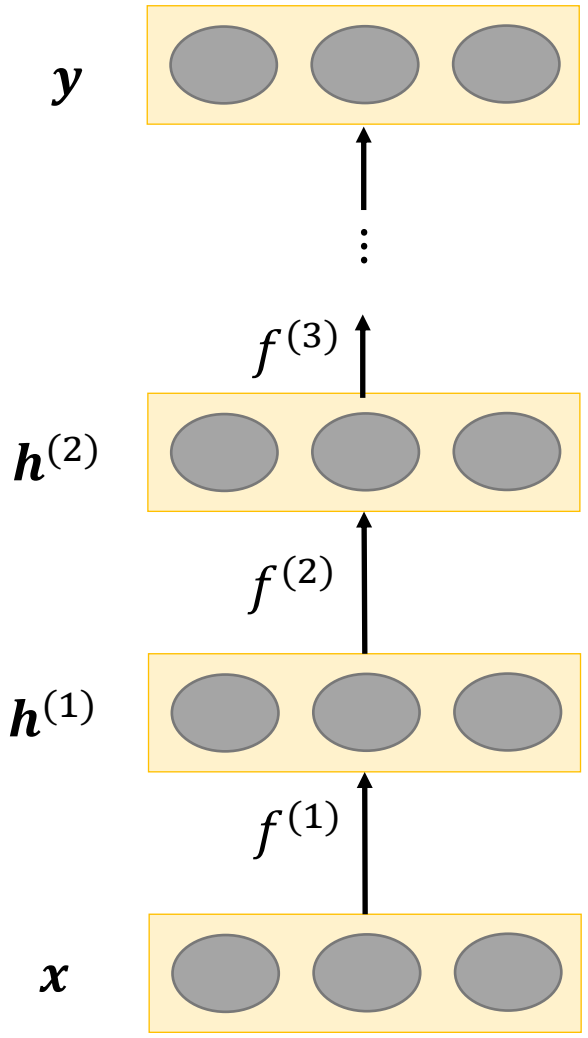
FNN as a function of functions

- How does the output $\boldsymbol{y}$ related to the input $\boldsymbol{x}$
- Each layer computes a function, so the network computes a composition of functions:

$$\boldsymbol{h}^{(1)} = f^{(1)}(\boldsymbol{x})$$
$$\boldsymbol{h}^{(2)} = f^{(2)}\left(\boldsymbol{h}^{(1)}\right)$$
$$\vdots$$
$$\boldsymbol{y} = f^{(L)}\left(\boldsymbol{h}^{(L-1)}\right)$$

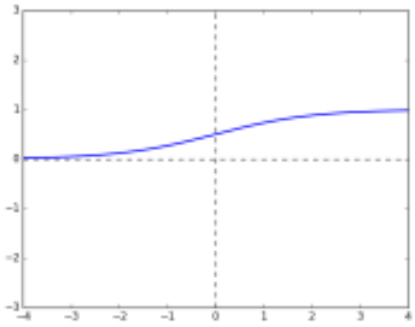
- Or more simply

$$\boldsymbol{y} = f^{(L)}\left(f^{(L-1)}\left(\cdots f^{(1)}(\boldsymbol{x})\right)\right)$$



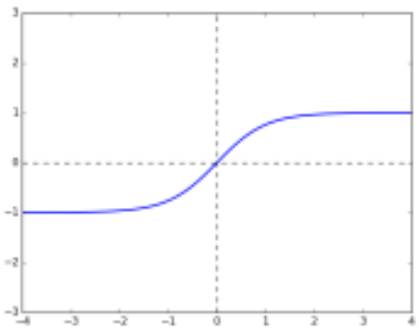
Activation functions

- We are free to use any kind of differentiable activation function we like at each layer
- A common choice was to use a sigmoid (logistic) function, which can be seen as a smooth approximation to the step function used in a perceptron



Logistic

$$y = \frac{1}{1 + e^{-z}}$$



Hyperbolic Tangent (tanh)

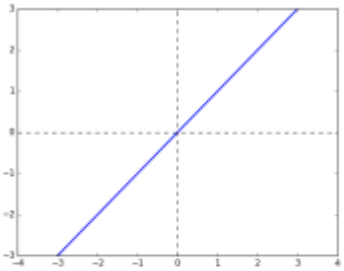
$$y = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

- However, the **logistic** function saturates at 1 for large positive inputs, and at 0 for large negative inputs
- The **tanh function** has a similar shape, but saturates at -1 and +1. In these regimes, the gradient of the output w.r.t. the input will be close to zero, so any gradient signal from higher layers will "vanish", as we will discuss later
- Tanh is better than sigmoid because it is zero-centred

Activation functions

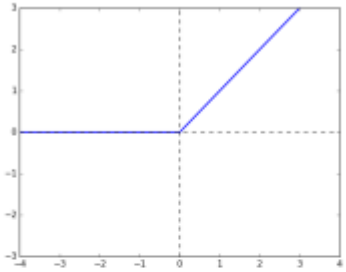
- Non-saturating activation functions
- The most common one is rectified linear unit or **ReLU**

$$\text{ReLU}(z) = \max(z, 0)$$



Linear

$y = z$



Rectified Linear Unit (ReLU)

$y = \max(0, z)$

- The ReLU function simply "turns off" negative inputs, and passes positive inputs unchanged. This turns out to have a gradient of 1, at least for positive inputs; this helps avoid vanishing gradients
- Unfortunately, for negative inputs, the gradient of ReLU units is 0, so the unit will never get any feedback signal to help it escape from this parameter setting; this is called the "dying ReLU" problem
- ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations

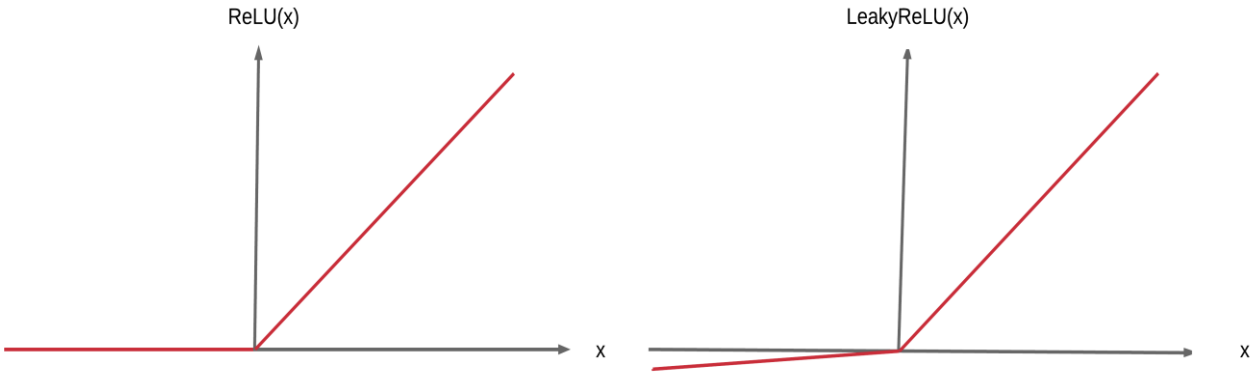
Activation functions

- One simple fix is to use the **leaky ReLU**. This is defined as

$$\text{LReLU}(z;\alpha) = \max(\alpha z, z), \qquad 0 < \alpha < 1$$

- The slope of this function is 1 for positive inputs, and α for negative inputs, thus ensuring there is some signal passed back to earlier layers, even when the input is negative.

- Leaky ReLU is non-smooth, it has a kink at $z = 0$

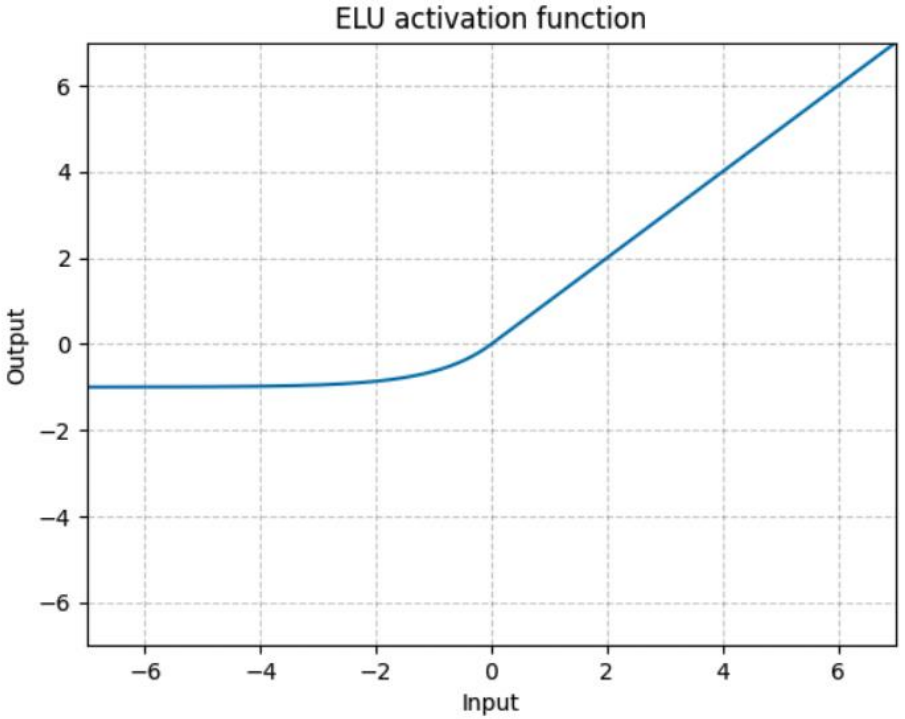


Activation functions

- Another popular choice is the **ELU**

$$ELU(z\,;\,\alpha) = \begin{cases} \alpha\left(e^{\{z\}} - 1\right) & if\ z \leq 0 \\ z & if\ z > 0 \end{cases}$$

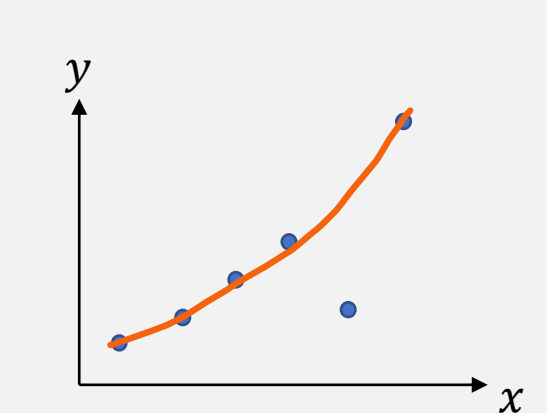
- This has the advantage over leaky ReLU of being a smooth function



Feature learning

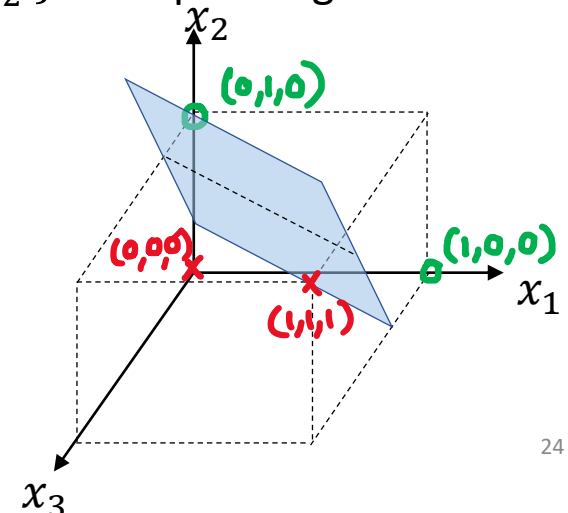
- Recall nonlinear feature maps in linear regression and classification
- For instance, the feature mapping $\phi(x) = \{1, x, x^2, x^3\}$ can represent third-degree polynomials in regression.

$$y = w_1 x + w_2 x^2 + w_3 x^3 + b$$



- In another instance, we used $\phi(x_1, x_2) = \{x_1, x_2, x_1 x_2\}$ for separating XOR function

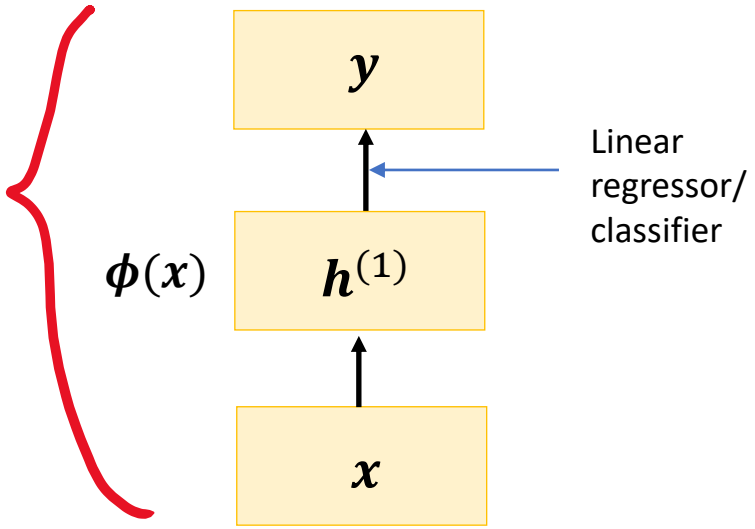
Input x_1	Input x_2	Input $x_3 = x_1 x_2$	Output y	
0	0	0	0	✗
1	0	0	1	○
0	1	0	1	○
1	1	1	0	✗



Feature learning

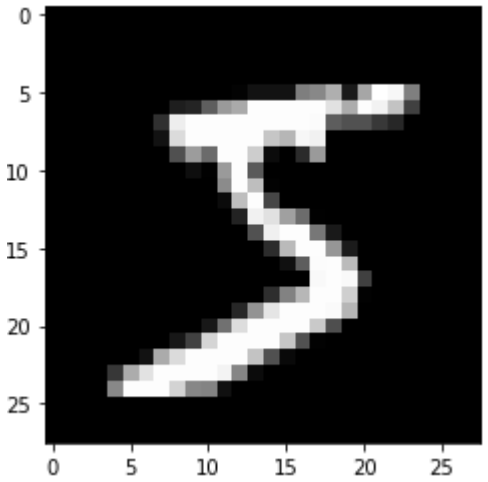
- Recall nonlinear feature maps in linear regression and classification
- For instance, the feature mapping $\phi(x) = \{1, x, x^2, x^3\}$ can represent third-degree polynomials in regression.
- In another instance, we used $\phi(x_1, x_2) = \{x_1, x_2, x_1 x_2\}$ for separating XOR function

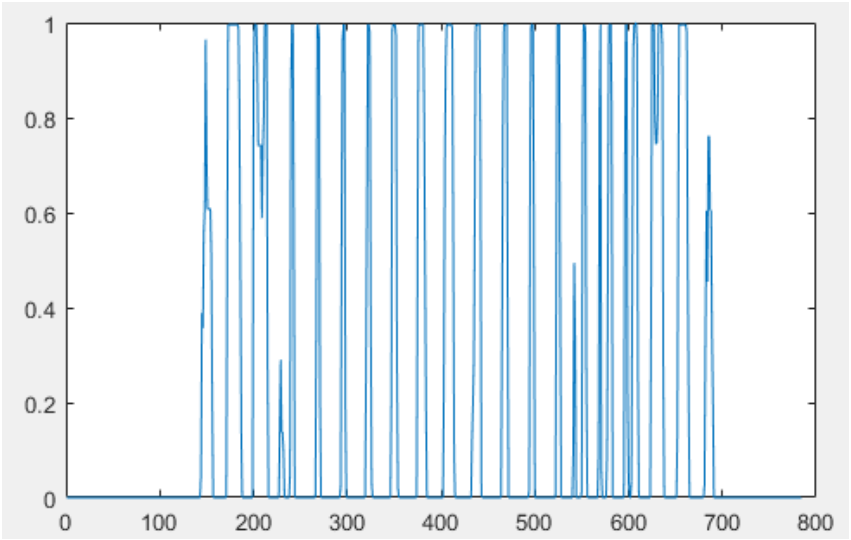
- **Neural nets** can be viewed as a way of learning nonlinear features
- For example, the hidden layer can be thought of as a feature map $\phi(x)$, and the output layer weights can be thought of as a linear model using those features



Feature learning example using MNIST

A multi-class classification task with 10 categories, one for each digit class





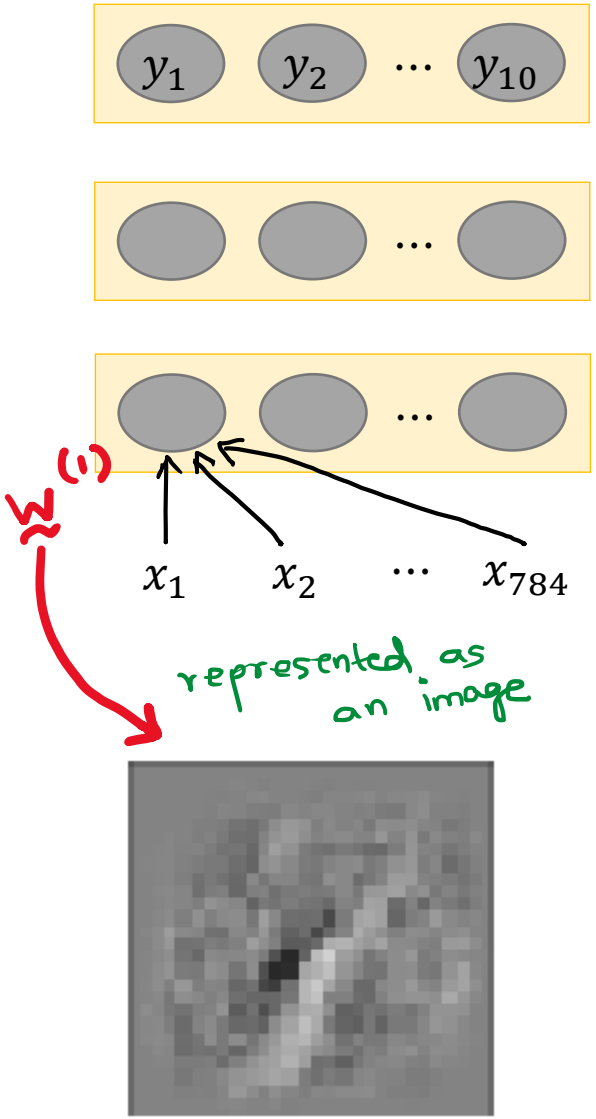
28 x 28 grayscale image

$\underset{\sim}{x}$

784 x 1 input vector

Feature learning example using MNIST

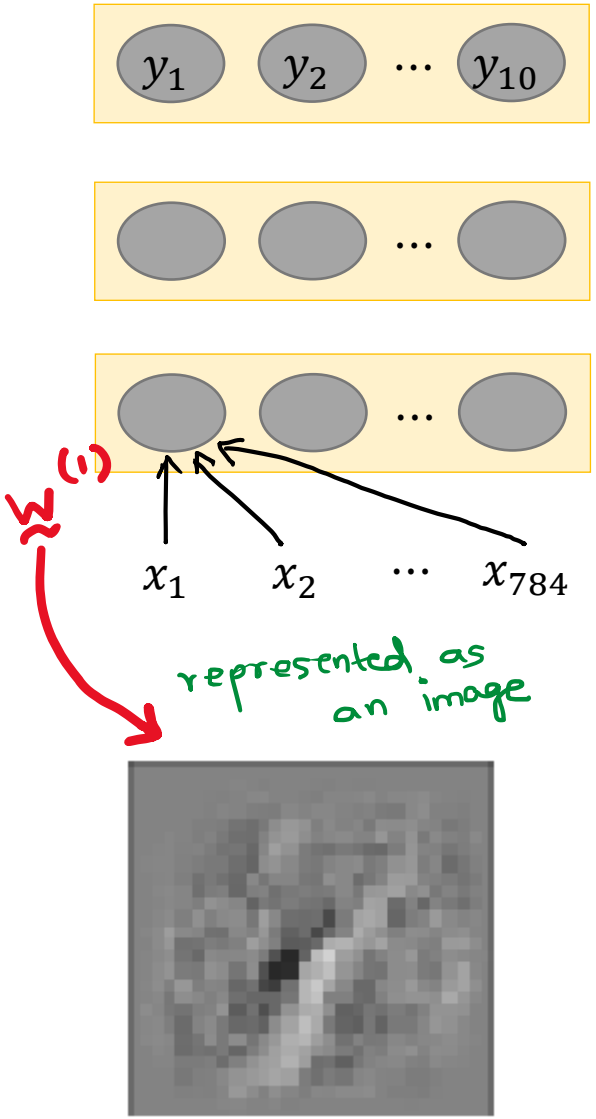
- Suppose we train an MLP with two hidden layers
- Each hidden unit receives inputs from each of the pixels and computes $\sigma(\boldsymbol{w}_i^T \boldsymbol{x})$
- Let's see what the first layer of hidden units computes by visualizing the weights
- The weights $\boldsymbol{w}^{(i)} \in R^{784\times 1}$ feeding into each hidden unit of the first layer can be reshaped into a 28×28 matrix and displayed as images
- Weight image: gray $= 0$, white $\rightarrow +$, black $\rightarrow -$



Feature learning example using MNIST

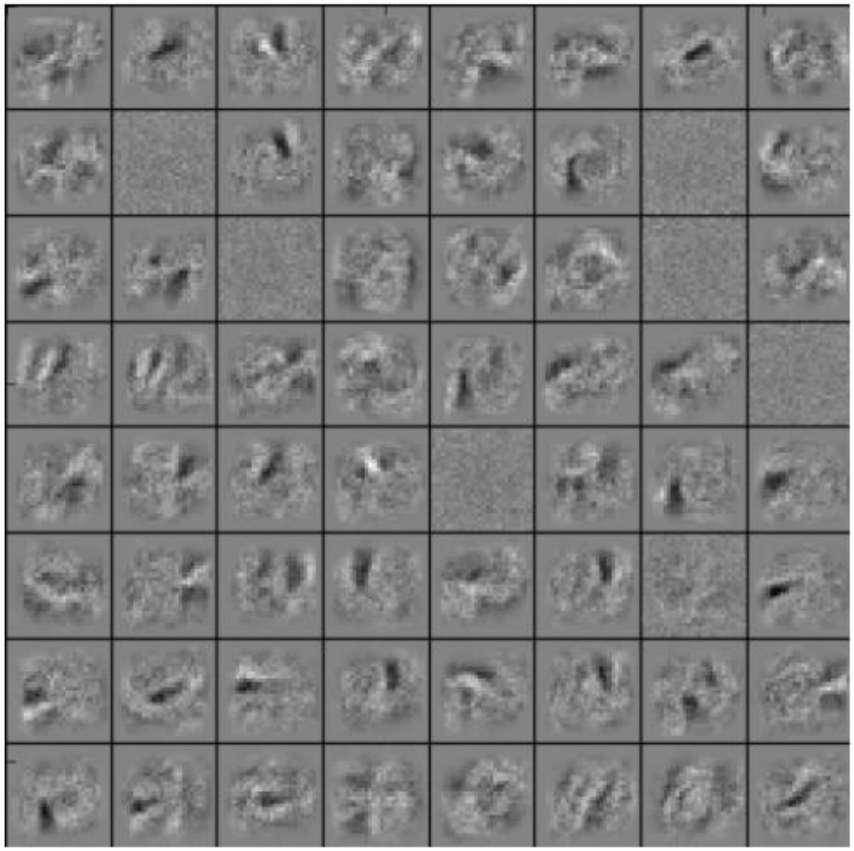
What do the weights tell us?

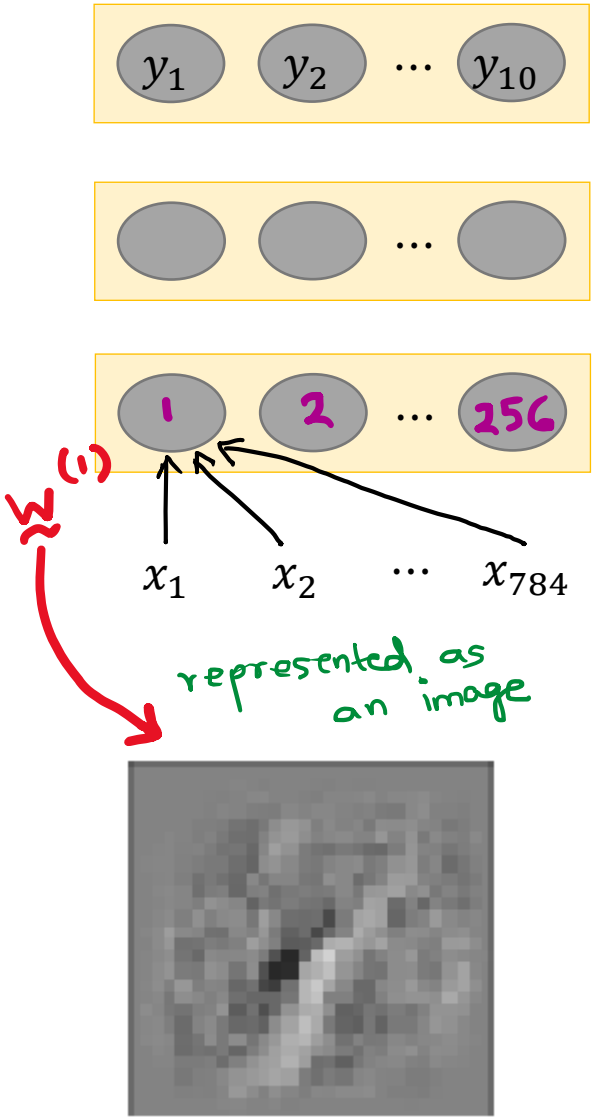
- In the weight visualization, positive values are lighter, and negative values are darker
- Each hidden unit computes the dot product of these weight vectors with the input image, and then passes the result through the activation function
- The weight images are mostly found to correspond to some **oriented edges**
- Thus the weights carry useful information about locations and orientations of strokes



Feature learning example using MNIST

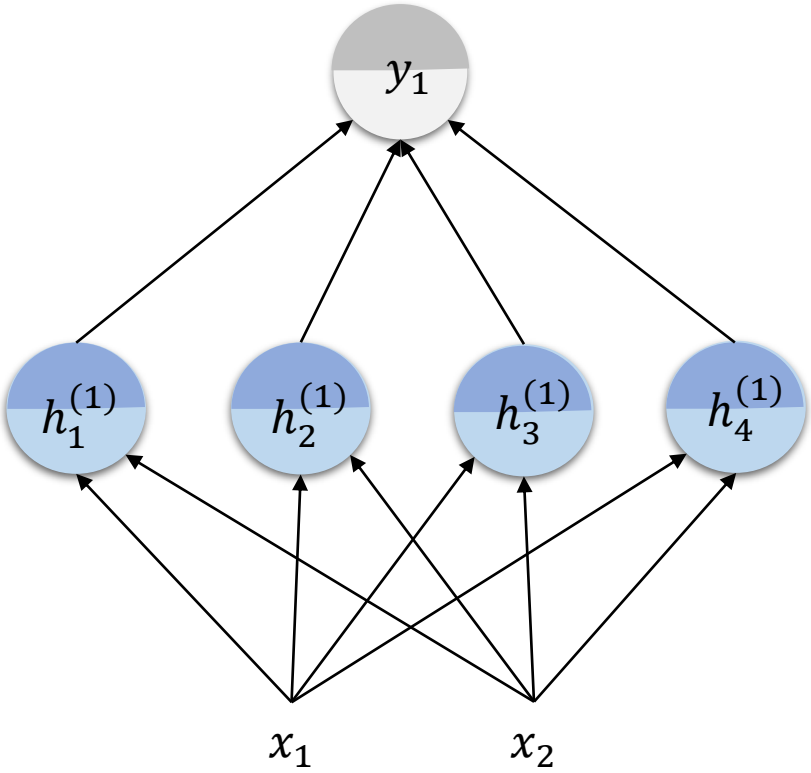
- There are 256 units in the first hidden layer and hence there are 256 images of weight vectors. Here are some more of images of the weight vectors for the 1st layer





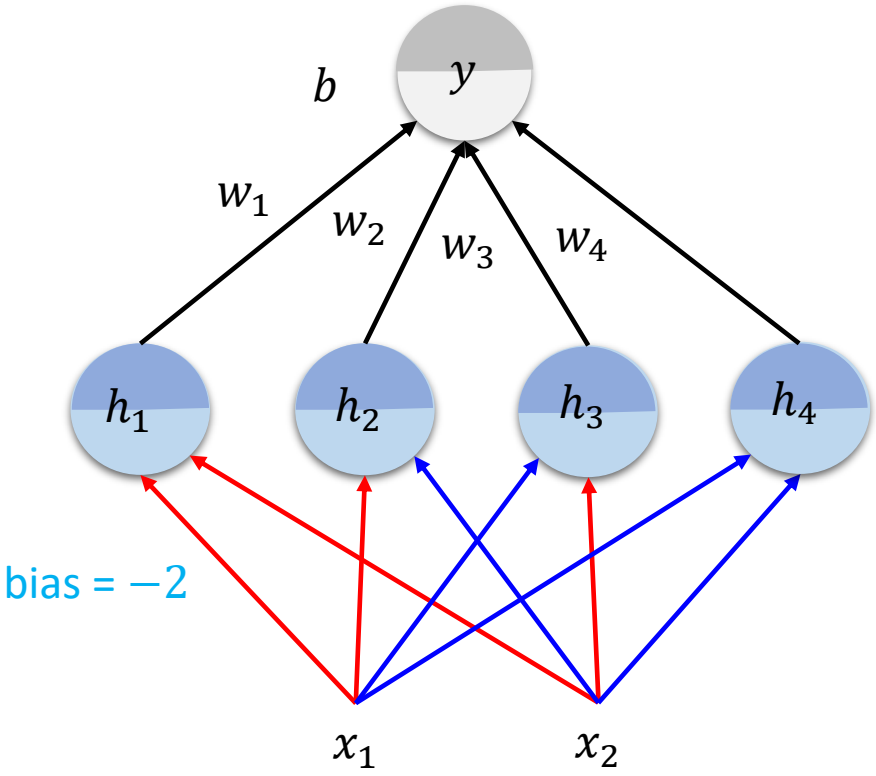
FNNs are very expressive!

- Multilayer feed-forward neural nets with *nonlinear* activation functions are universal approximators: they can approximate any function arbitrarily well
- For example, FNN with **just one hidden layer can represent the any Boolean function** (or logic gates)
- Let's try this network for representing the XOR gate



FNNs are very expressive!

Note: Here we consider hard threshold activation functions for this example



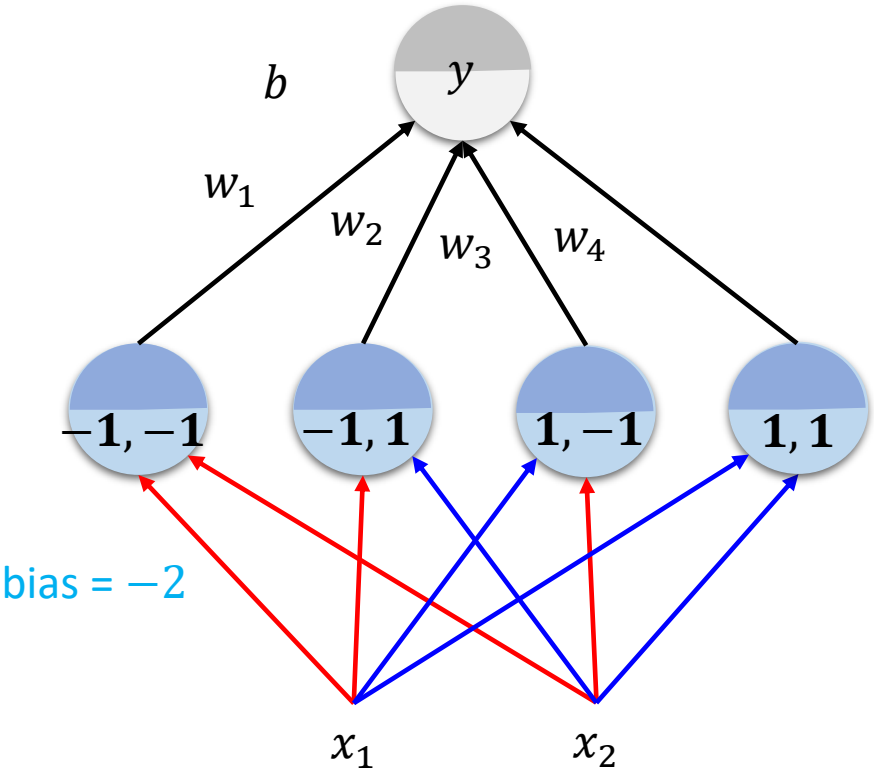
Red edge indicates $w = -1$

Blue edge indicates $w = +1$

- For this discussion, we will assume True = +1 and False = -1
- We consider two inputs and four perceptrons in hidden layer
- The weights of the first layer are manually fixed for illustration
- The bias for each perceptron is set to -2
- The weights w_i's and bias for the output perceptron are to be learned (in this case)
- **Question**: Can we find $\{w_1, w_2, w_3, w_4, b\}$ such that the FNN can represent any Boolean function?

FNNs are very expressive!

Note: Here we consider hard threshold activation functions for this example

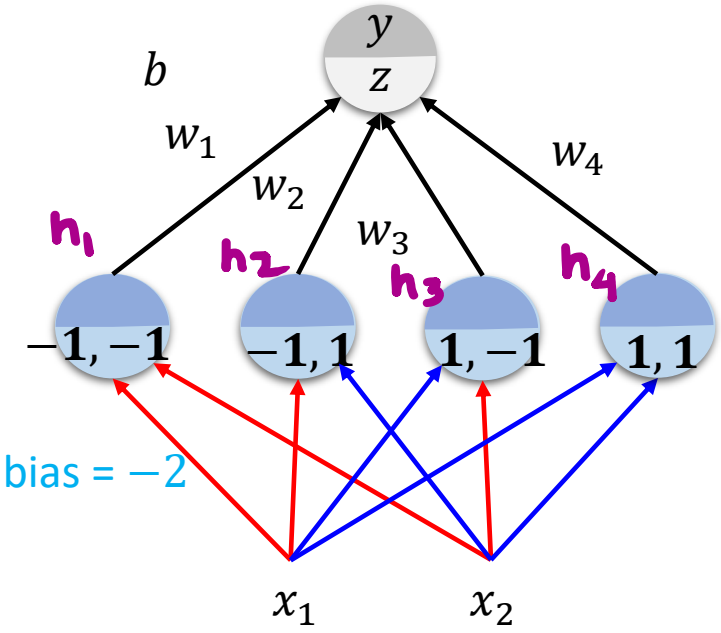


Red edge indicates $w = -1$

Blue edge indicates $w = +1$

- For this discussion, we will assume True = +1 and False = -1
- **Question**: Can we find $\{w_1, w_2, w_3, w_4, b\}$ such that the FNN can represent any Boolean function?
- Each hidden unit fires for a specific input (no two perceptron fire for the same input)
- The first hidden perceptron fires for inputs {False, False} → {-1,-1}
- The second hidden perceptron fires for inputs {False, True} → {-1,1}

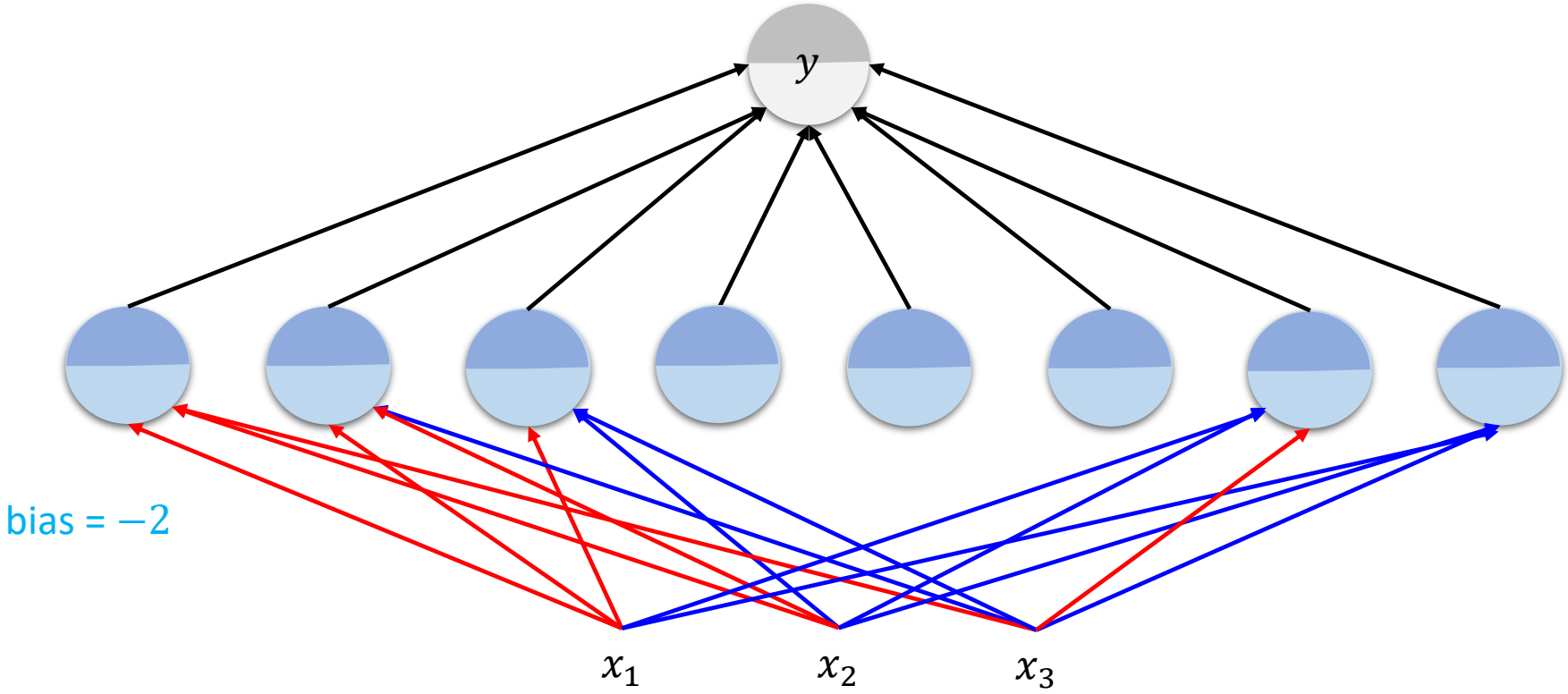
FNNs are very expressive!



- Can we adjust the weights to get whatever output we need?
- **Say we want to represent XOR**
- $b = -1$
- $w_1 = 0, \ \ w_2 = 1, \ \ w_3 = 1, \ \ w_4 = 0$

x_1	x_2	XOR	h_1	h_2	h_3	h_4	$z = \sum w_i h_i + b$	$y = \sigma(z)$
-1	-1	0	1	0	0	0	$w_1 - 1$	
-1	1	1	0	1	0	0	$w_2 - 1$	
1	-1	1	0	0	1	0	$w_3 - 1$	
1	1	0	0	0	0	1	$w_4 - 1$	

FNNs are very expressive!



Red edge indicates $w = +1$

Blue edge indicates $w = -1$

For D inputs, **2^D units needed in hidden layer**

Shallow vs Deep neural networks

- Shallow neural networks
  - Shallow neural networks have **one hidden layer** of neurons
  - They are can very expressive, but that may require a large width
  - Having an extremely wide neural network can be computationally very costly

- Deep neural networks
  - Deep neural networks have **more than one hidden layer** of neurons
  - They are **expressive** as well as **compact**, i.e. don't need to be extremely wide
  - Deep neural networks are therefore more practically computationally