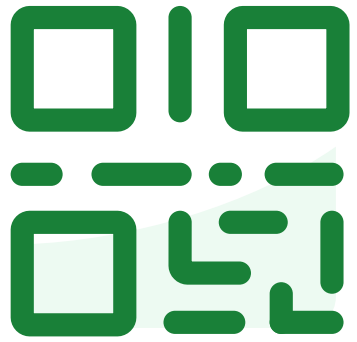


Do not edit
How to change the
design



**Join at slido.com
#2095864**

① The Slido app must be installed on every computer you're presenting from

slido

Lecture 12

Neural Networks (1)

Non-linearity, Architecture, Activation Functions, Output Layers, and Loss

EECS 189/289, Fall 2025 @ UC Berkeley

Joseph E. Gonzalez and Narges Norouzi



2095864

Recap: Linear Models

- Recall that (Generalized) Linear Models (GLM) are:

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^M w_j \phi_j(\mathbf{x}) + w_0\right)$$

- For classification: $f(\cdot)$ is a non-linear output activation function.
 - For regression: $f(\cdot)$ is an identity function.
-

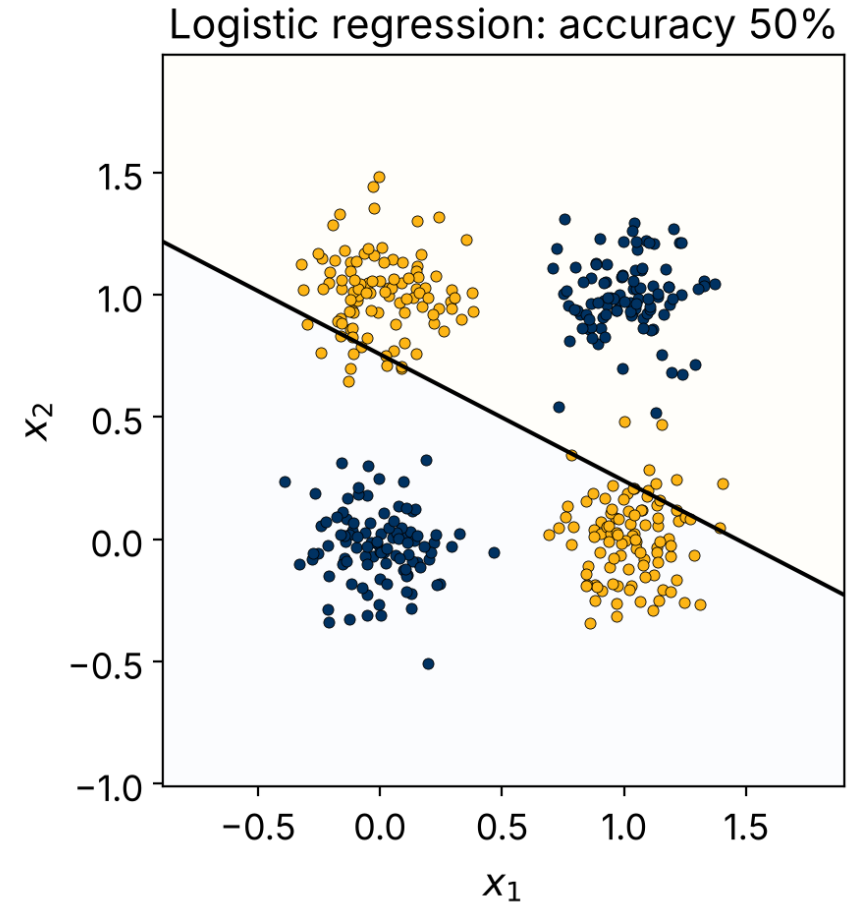
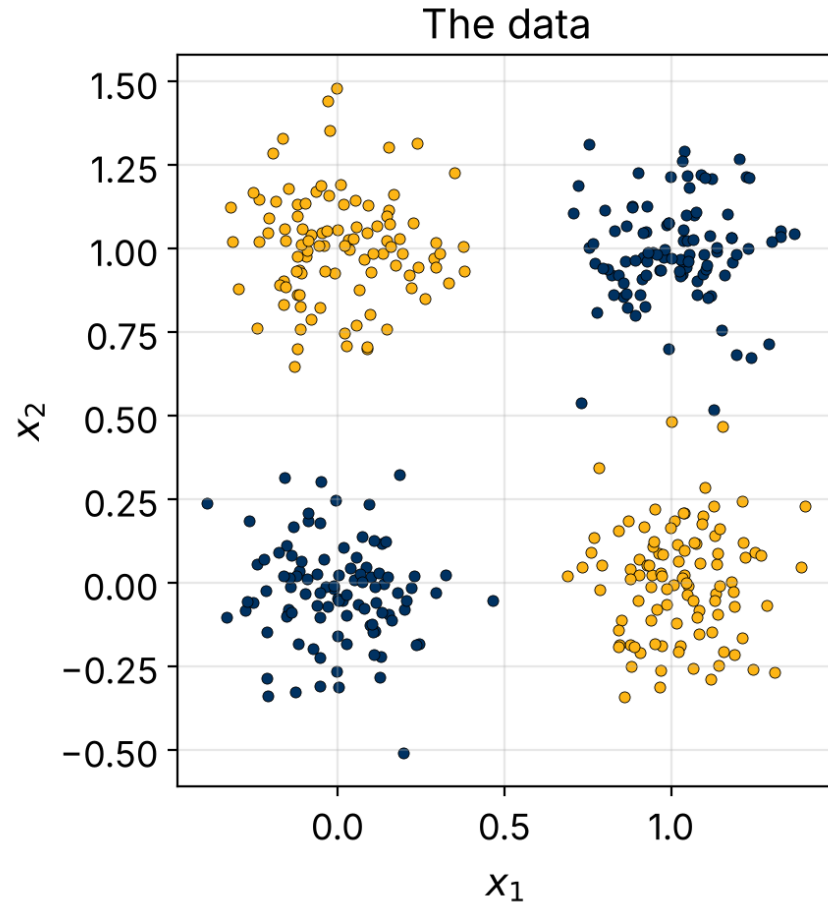
Can we approximate any function with sufficient accuracy if the set of $\phi_j(\mathbf{x})$ is rich enough?

In most cases **NO**, because $\phi_j(\mathbf{x})$ are fixed and independent of the training data.

Logistic Regression on XOR



2095864



$$E(w) = 0.6928 \approx \ln 2$$

Roadmap

- The Neuron
- Architecture
- Activation Functions
- Output Layers and Loss
- PyTorch (Optional)



2095864



2095864

The Neuron

- The Neuron
- Architecture
- Activation Functions
- Output Layers and Loss
- PyTorch (Optional)



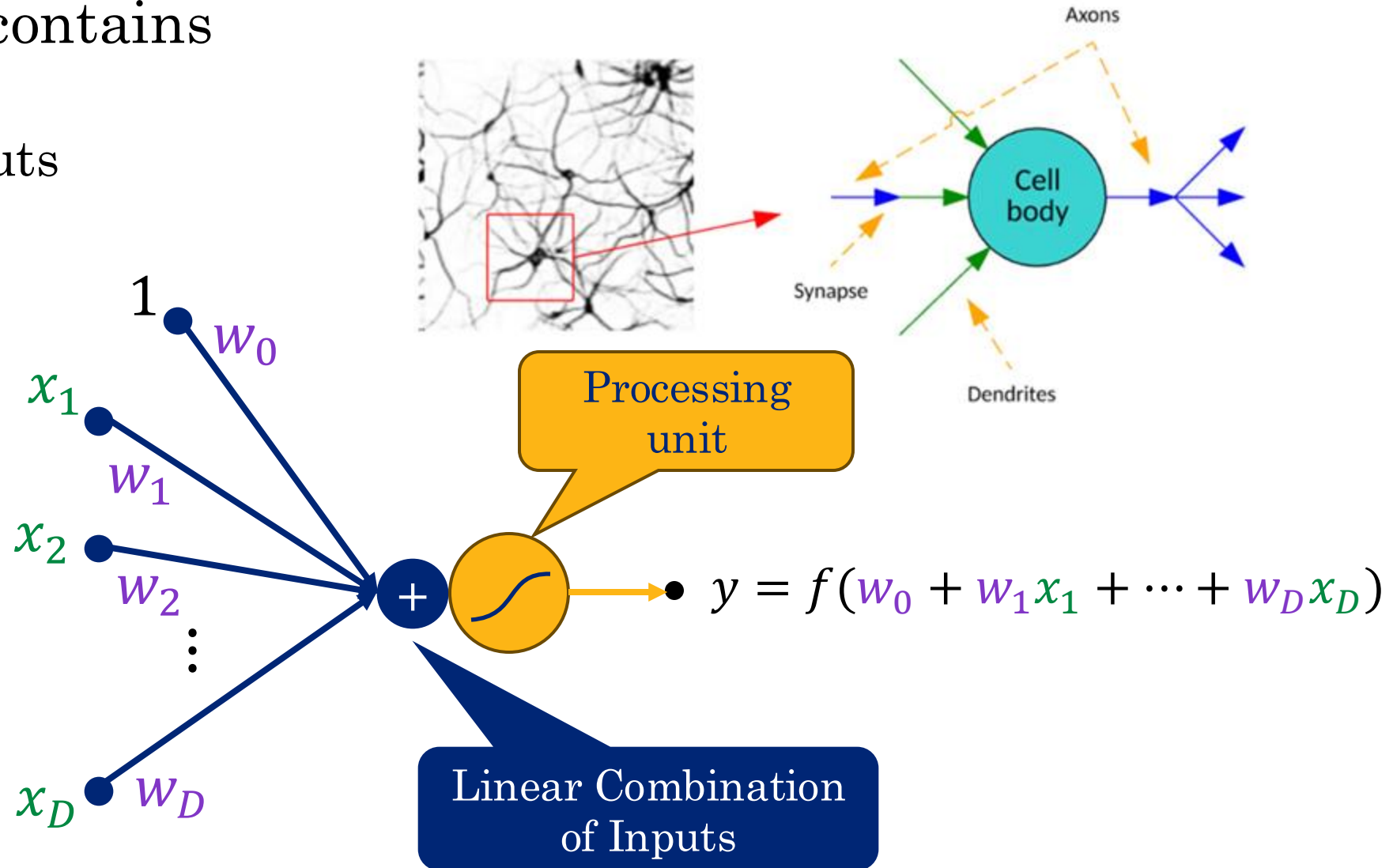
2095864

From Biological to Artificial Neurons

- Biological neuron contains

- Inputs
- Summation of inputs
- Processing unit
- Output

- Artificial neuron



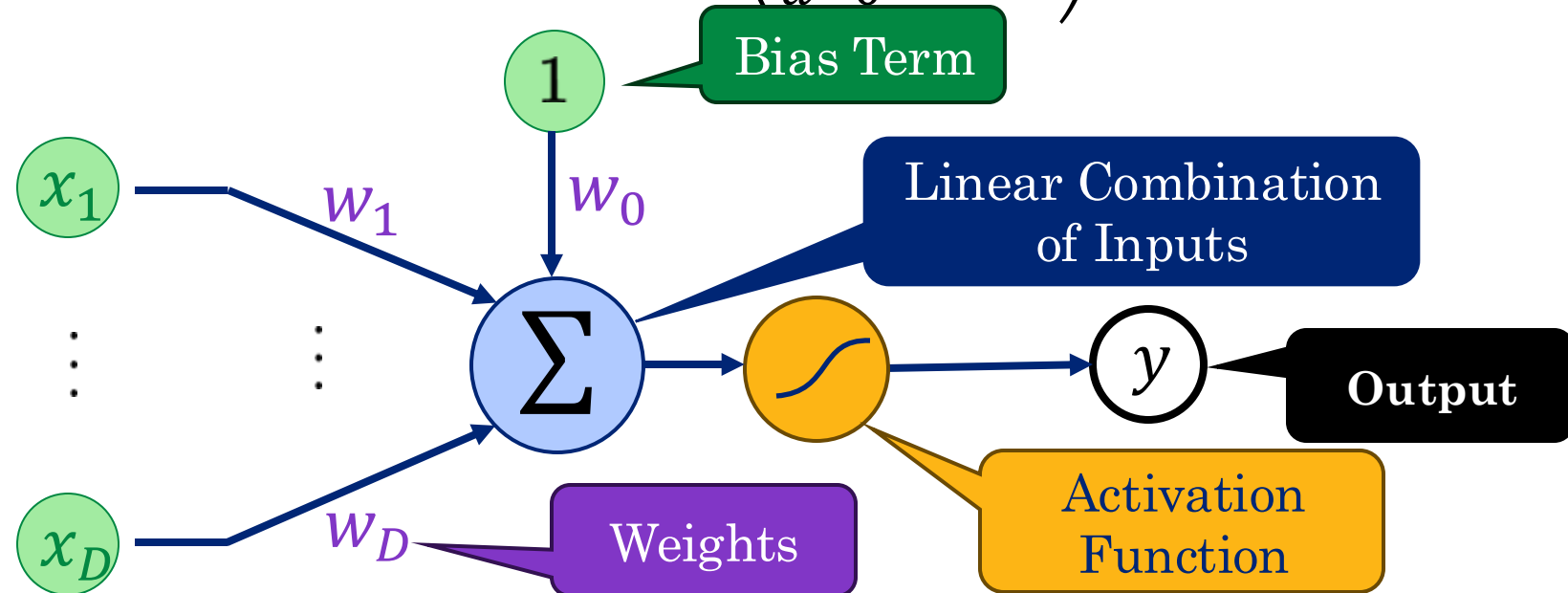


2095864

Neuron – Activation Function

- An artificial neuron is a **summation unit** + **an activation function**.
- The weighted summation is passed to the activation function to calculate the neuron's activation.

$$y = f(\vec{w} \cdot \vec{x}) = f\left(\sum_{d=0}^D w_d x_d\right)$$



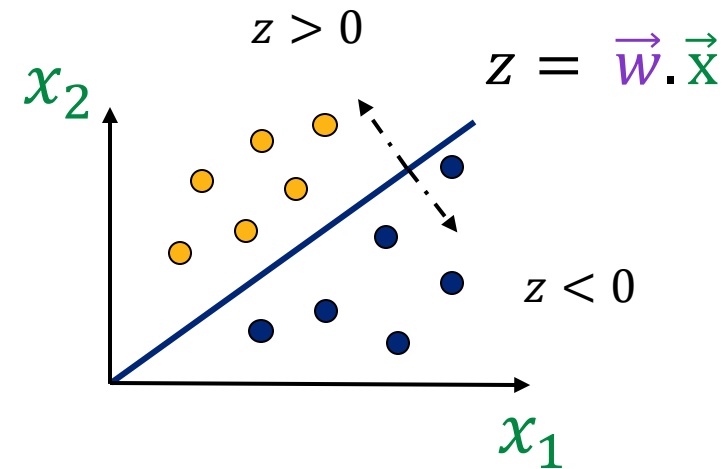
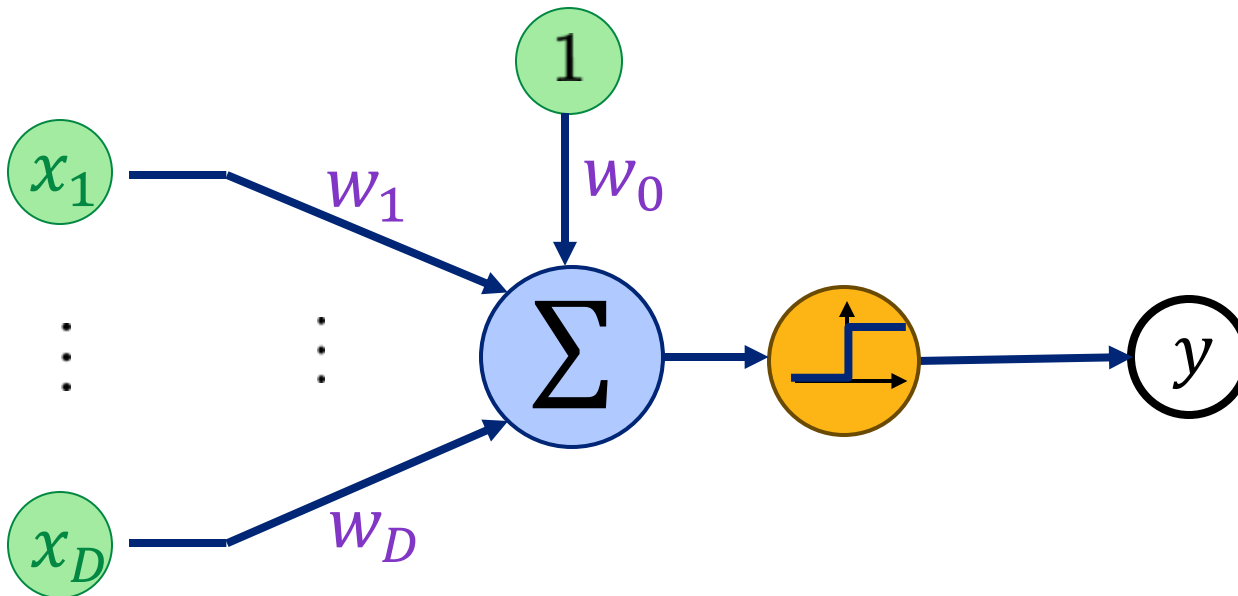
Perceptron



2095864

- Assume the activation function to be a **step function**.
 - If the weighted combination of inputs is **negative**, the output is **0** → neuron is **not activated**.
 - If the weighted combination of inputs is **positive**, the output is **1** → neuron is **activated**.

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$$

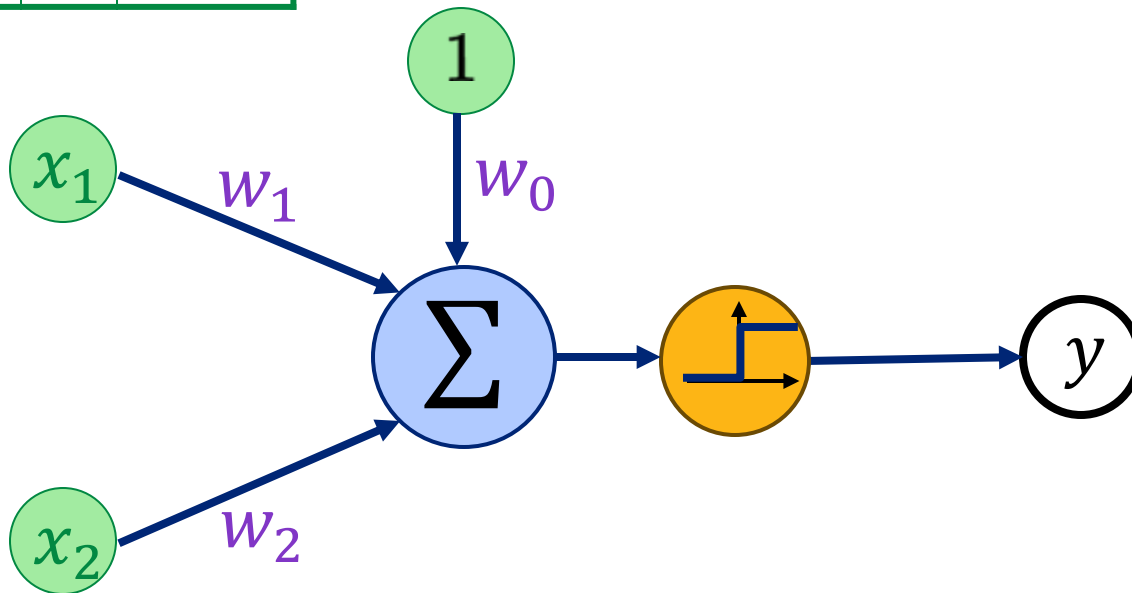
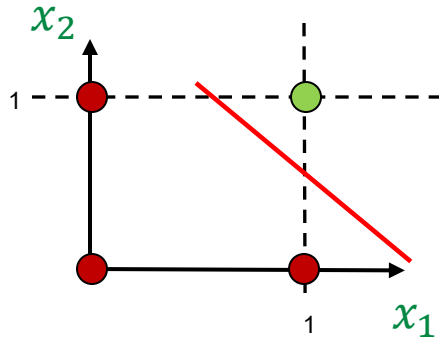


Functional Completeness - AND



2095864

x_1	x_2	y
1	0	0
1	1	1
0	0	0
0	1	0





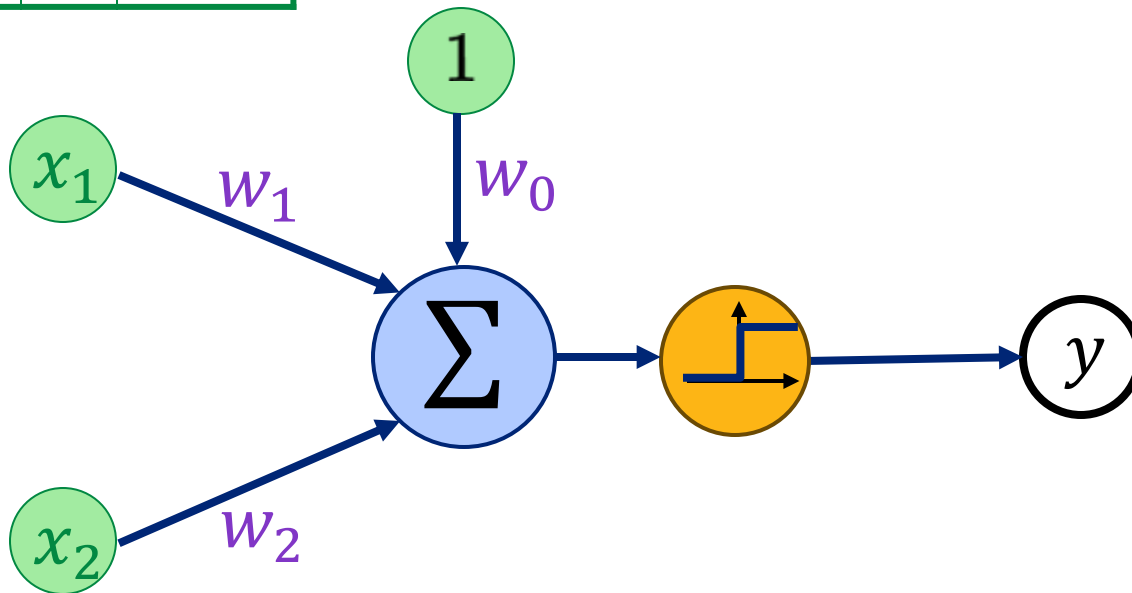
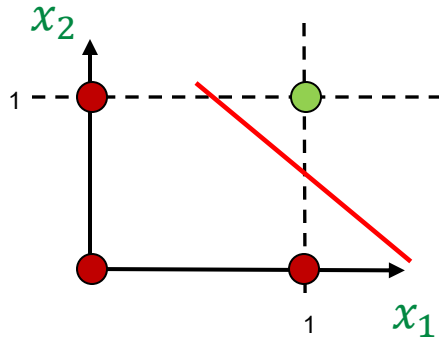
**What are example parameters
for the AND gate?
 $w_0=?$, $w_1=?$, $w_2=?$**

Functional Completeness - AND



2095864

x_1	x_2	y
1	0	0
1	1	1
0	0	0
0	1	0

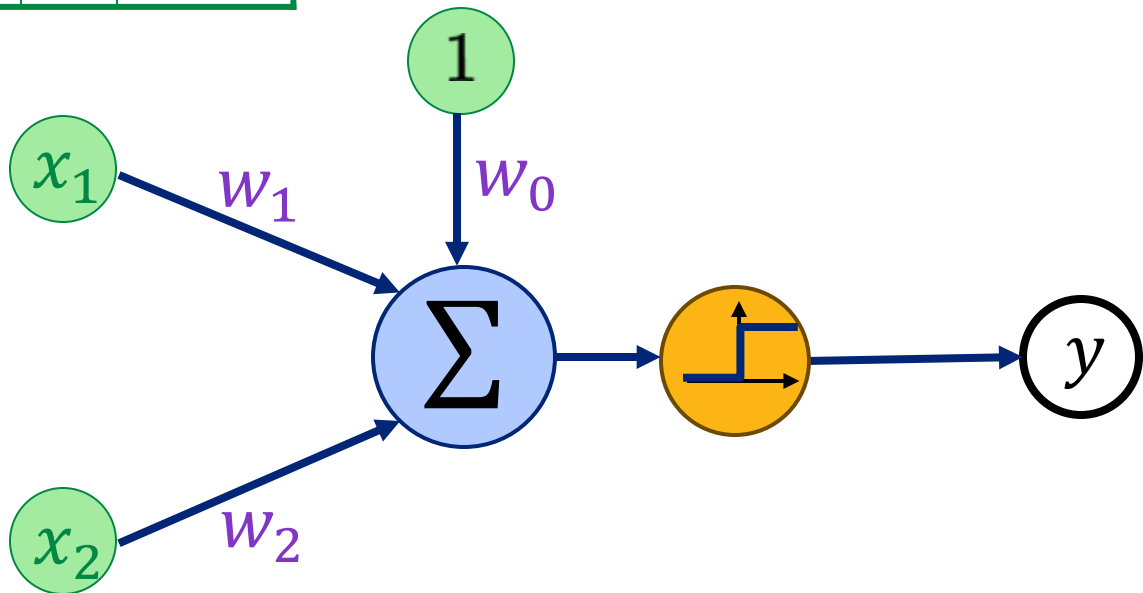
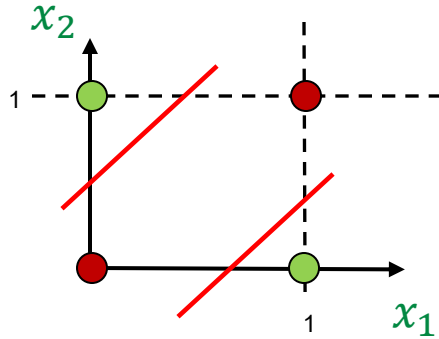


Functional Completeness - XOR



2095864

x_1	x_2	y
1	0	1
1	1	0
0	0	0
0	1	1





**Can we build the XOR gate
with only one neuron?**

Architecture

- The Neuron
- Architecture
- Activation Functions
- Output Layers and Loss
- PyTorch (Optional)



2095864

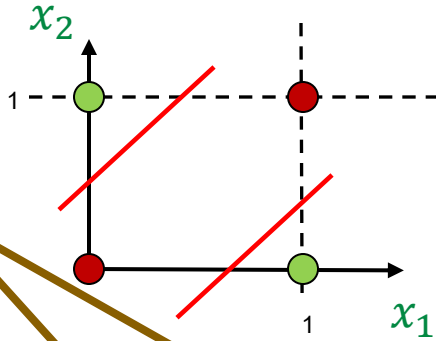
XOR with Two Layers



2095864

x_1	x_2	y
1	0	1
1	1	0
0	0	0
0	1	1

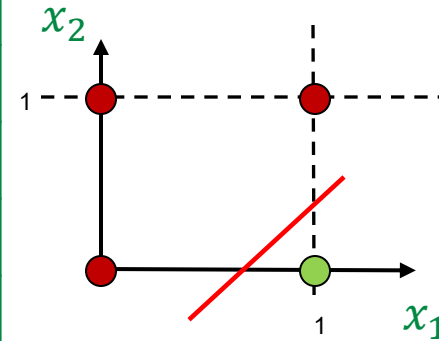
$x_1 \text{ XOR } x_2$



Non-linear Decision Boundary
(No linear separability)

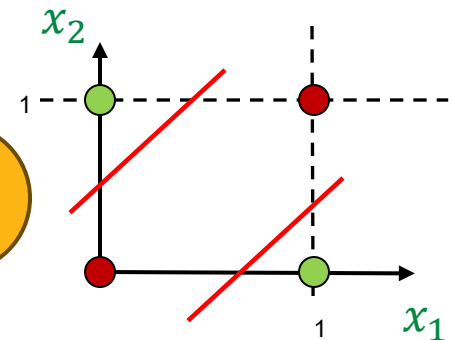
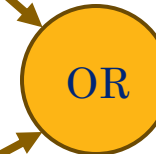
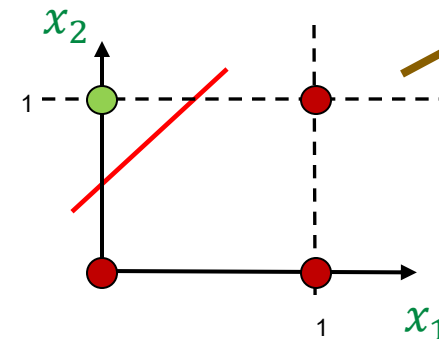
$x_1 \text{ AND } \sim x_2$

x_1	x_2	y
1	0	1
1	1	0
0	0	0
0	1	0



$\sim x_1 \text{ AND } x_2$

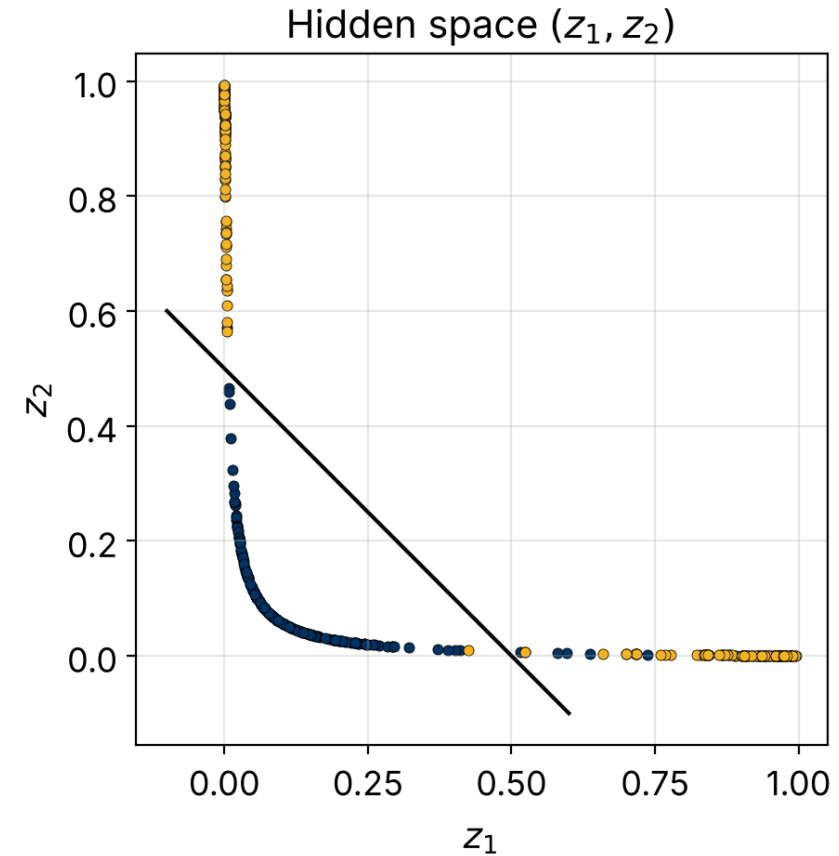
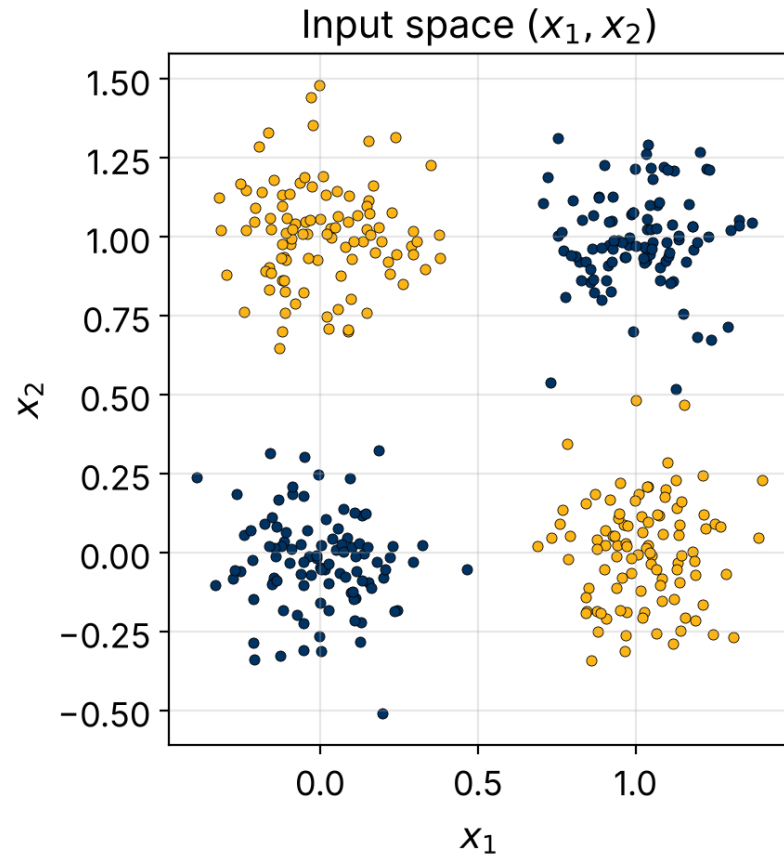
x_1	x_2	y
1	0	0
1	1	0
0	0	0
0	1	1



Hidden Units Are New Features



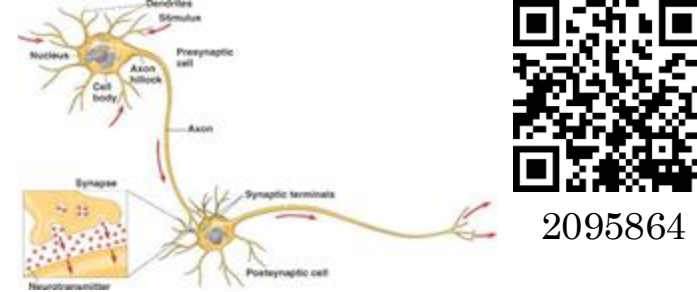
2095864



$$z_1 = h(x_1 - x_2 - 0.5)$$

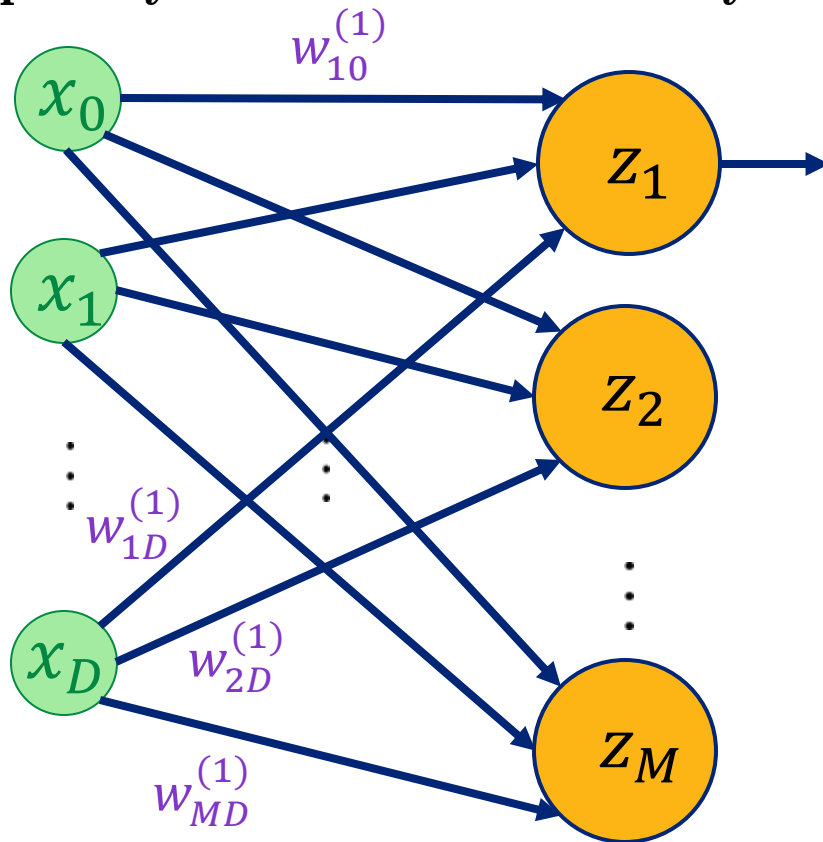
$$z_2 = h(-x_1 + x_2 - 0.5)$$

Neural Network



Input layer

Hidden layer



Superscript (1) indicates that corresponding parameters are in the first 'layer' of the network.

Activations (M linear combination of inputs)

$$a_i^{(1)} = \sum_{j=1}^D w_{ij}^{(1)} x_j + w_{i0}^{(1)}$$

Pre-activation or logits

Weights

Bias Term

$i \in \{1, \dots, M\}$

$$z_i^{(1)} = h(a_i^{(1)})$$

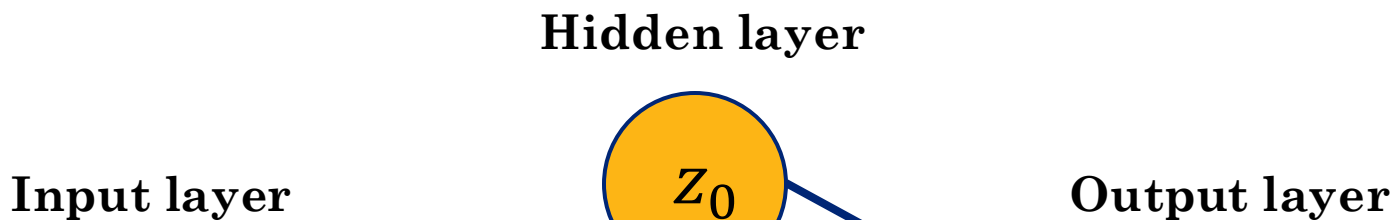
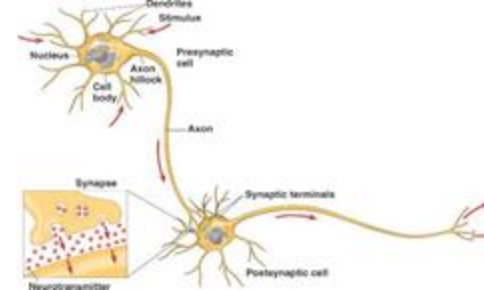
Output of basis functions

Activation Function

$i \in \{1, \dots, M\}$



2095864



$$a_k^{(2)} = \sum_{i=1}^M w_{ki}^{(2)} z_i^{(1)} + w_{k0}^{(2)} \quad k \in \{1, \dots, K\}$$

$$y(x, w) = f(a_k^{(2)})$$

$$= f\left(\sum_{i=1}^M w_{ki}^{(2)} z_i^{(1)} + w_{k0}^{(2)}\right)$$

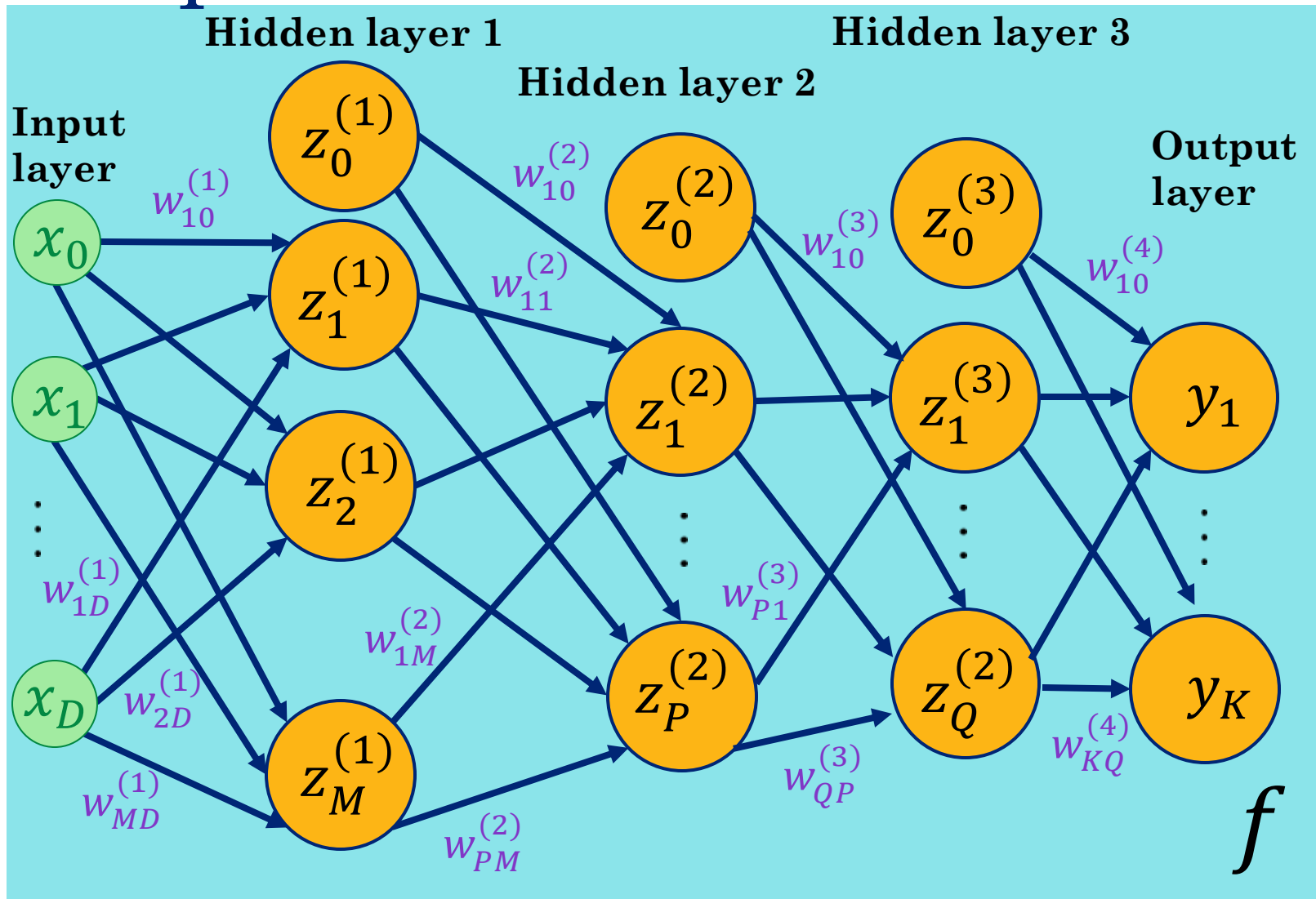
$$= f\left(\sum_{i=1}^M w_{ki}^{(2)} h\left(\sum_{j=1}^D w_{ij}^{(1)} x_j + w_{i0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

$$z_i^{(1)} = h\left(\sum_{j=1}^D w_{ij}^{(1)} x_j + w_{i0}^{(1)}\right) \quad i \in \{1, \dots, M\}$$

Now, let's add output layer.

$$= f(W^{(2)} h(W^{(1)} \mathbf{x}))$$

Deep Neural Network



$$l \in \{1, \dots, L\}$$

$$z^{(l)} = h^{(l)}(W^{(l)} z^{(l-1)})$$

Activation in
layer l

Weights in
layer l

$$z^{(0)} = \mathbf{x}$$

$$z^{(L)} = \mathbf{y}$$

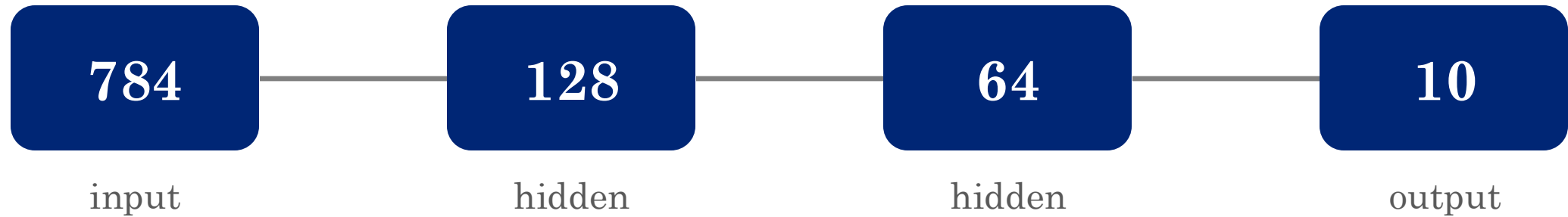


2095864

Count the Parameters



2095864

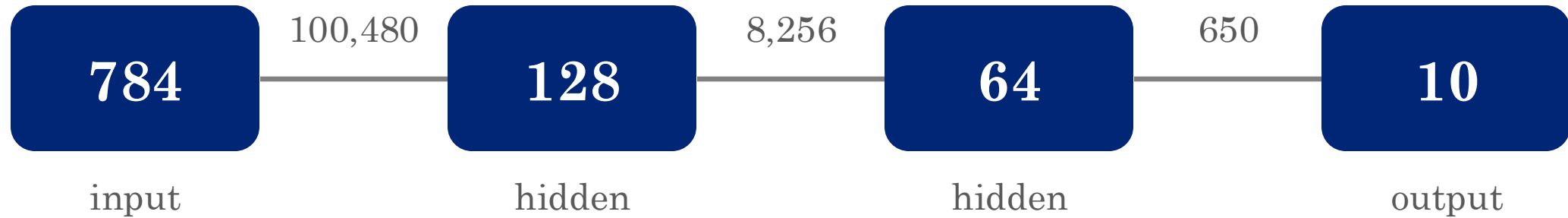


How many weights and biases?

Count the Parameters



2095864



$$(784 \times 128 + 128) + (128 \times 64 + 64) + (64 \times 10 + 10) = 109,386$$

Universality Theorem

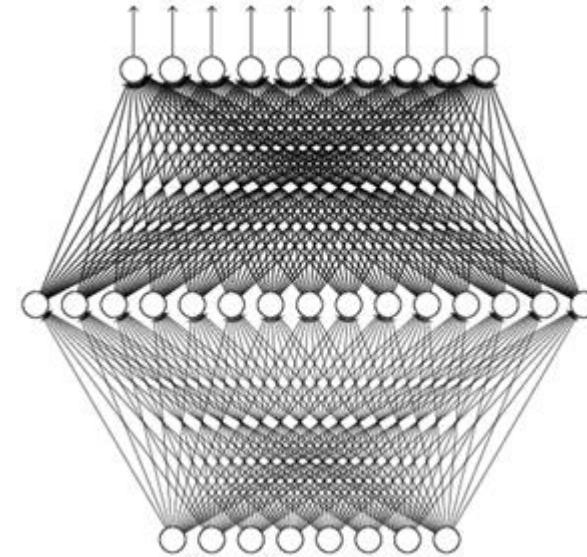


2095864

$$f: \mathbb{R}^D \rightarrow \mathbb{R}^K$$

Can be realized by a network
with one hidden layer

(given enough hidden neurons)



Visual proof:

<http://neuralnetworksanddeeplearning.com/chap4.html>

Neural networks are said to be universal approximators.

Why “Deep” neural network not “Fat” neural network?

Why Deep? Analogy

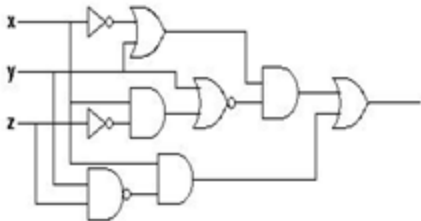


2095864

Logic circuits

- Logic circuits consists of **gates**.
- **Two layers of logic gates** can represent **any Boolean function**.
- Using multiple layers of logic gates to build some functions are much simpler.

→ less gates needed



Neural network

- Neural network consists of **neurons**.
- **A hidden layer network** can represent **any continuous function**.
- Using multiple layers of neurons to represent some functions are much simpler.



Activation Functions

- The Neuron
- Architecture
- Activation Functions
- Output Layers and Loss
- PyTorch (Optional)

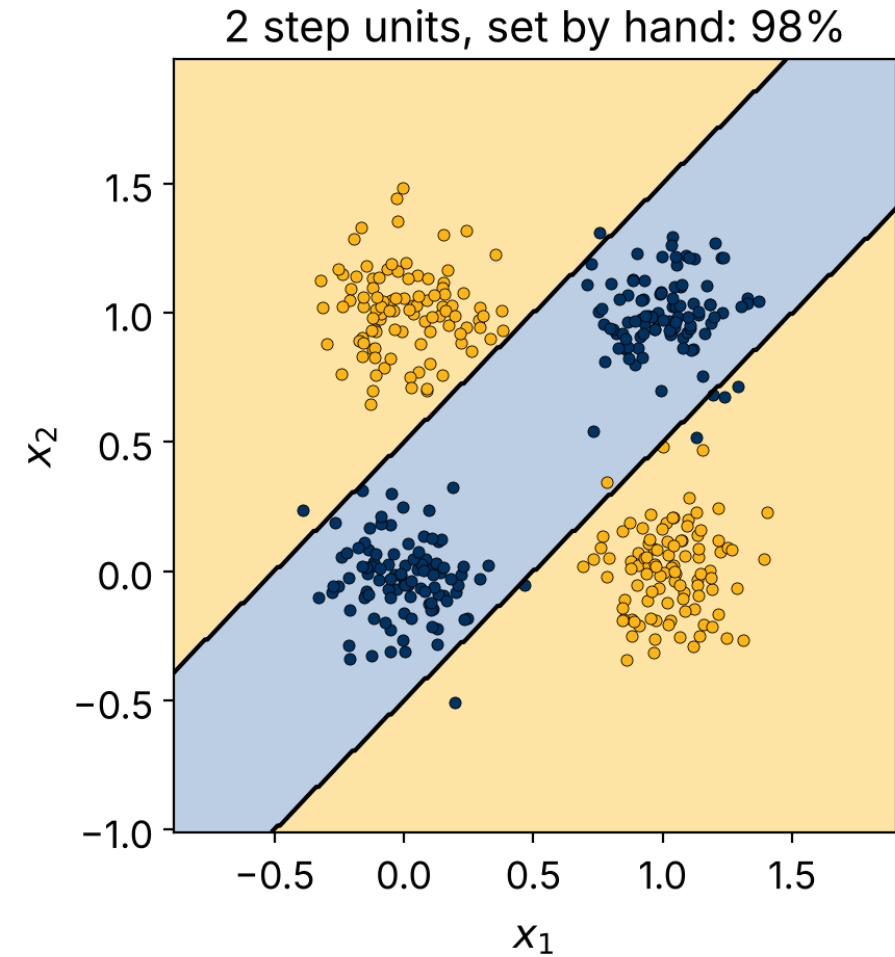


2095864

Depth Alone Does Not Help



2095864



Why Linear Layers Collapse



2095864

$$\begin{aligned} y &= W^{(2)} (W^{(1)} x + b^{(1)}) + b^{(2)} \\ &= W^{(2)} W^{(1)} x + W^{(2)} b^{(1)} + b^{(2)} \\ &= W x + b \end{aligned}$$

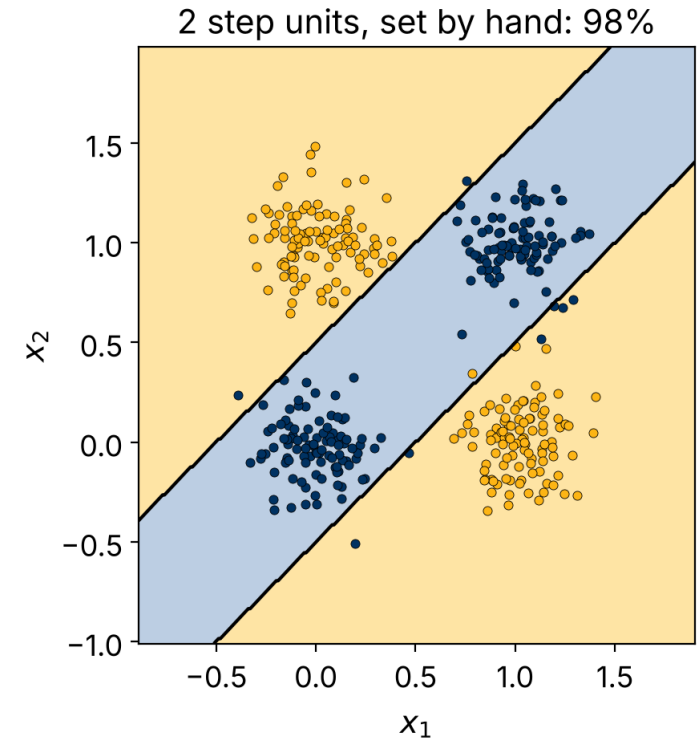
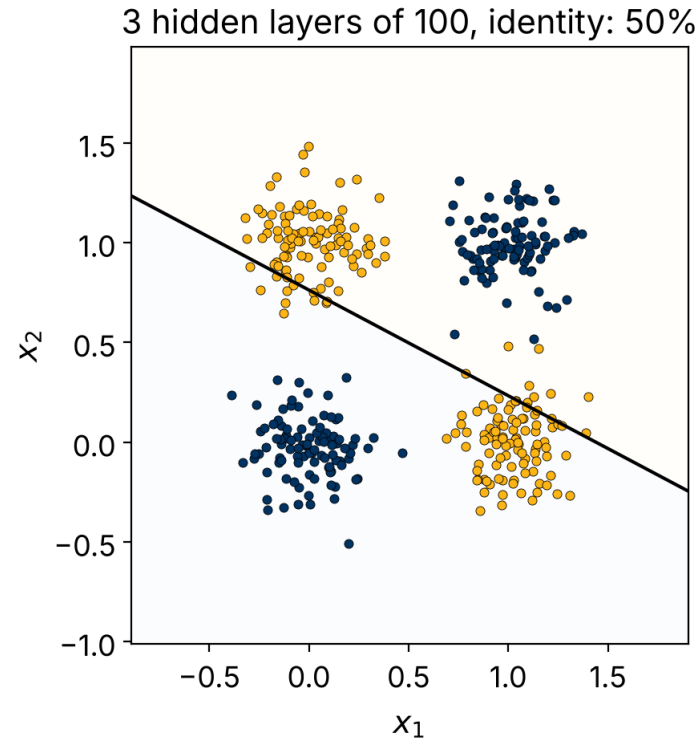
$$W = W^{(L)} W^{(L-1)} \dots W^{(1)}$$



2095864

Depth Alone Doesn't Help

Demo in the code



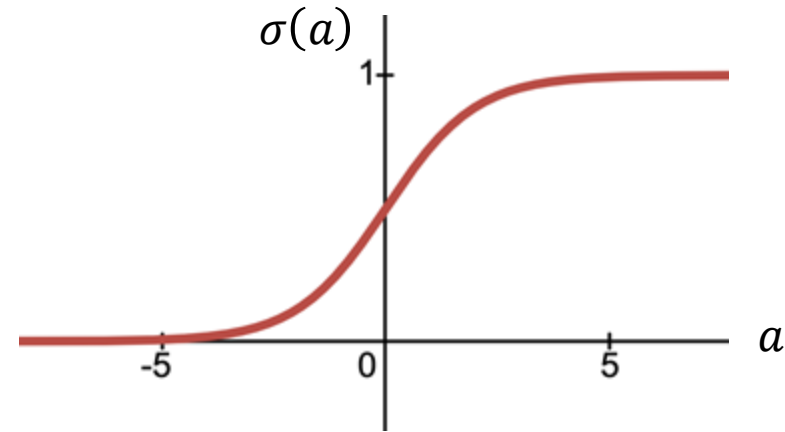


2095864

Activation Functions

- The simplest option for a hidden unit activation function is the identity function.
 - But it is redundant to have a hidden layer if all activations are identity.
- Simple non-linear differentiable activation function is sigmoid.

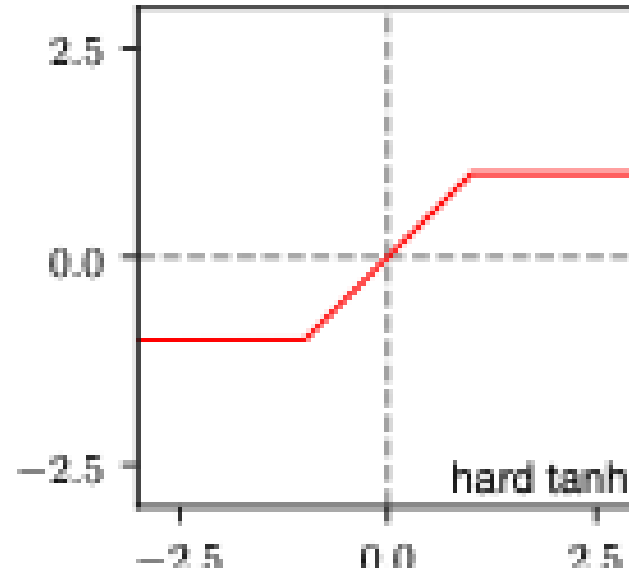
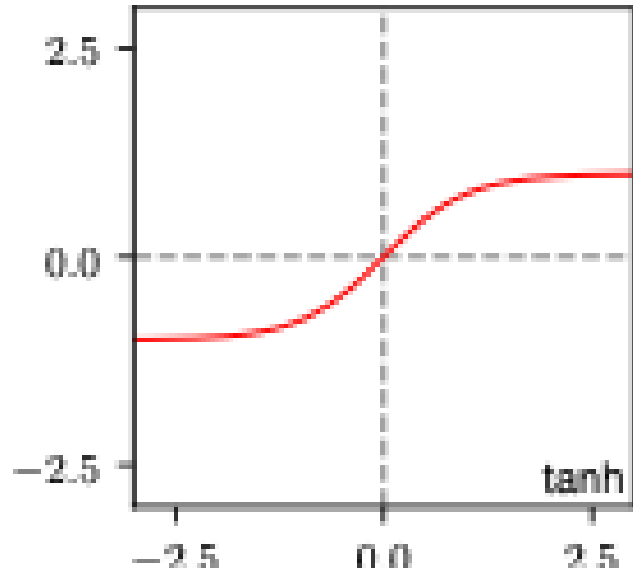
$$\sigma(a) = \frac{1}{1 + e^{-a}}$$





2095864

Other Activation Functions



$$\tanh(a) = 2\sigma(2a) - 1 \quad h(a) = \max(-1, \min(1, a))$$

$$= \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

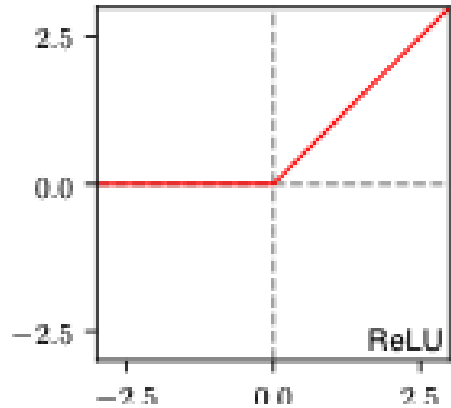
One issue with tanh and sigmoid family action function is that gradients go to 0 when magnitude of activations are large.

We need an activation that has non-zero gradient at least when activations are large positive values.

ReLU and Its Variants

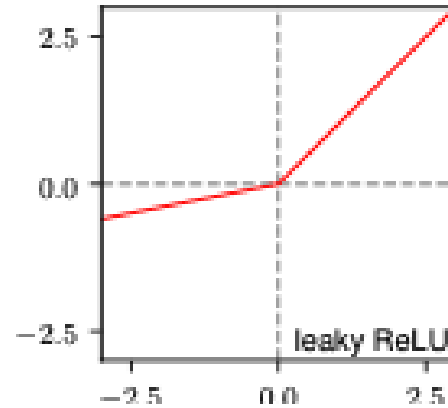


2095864



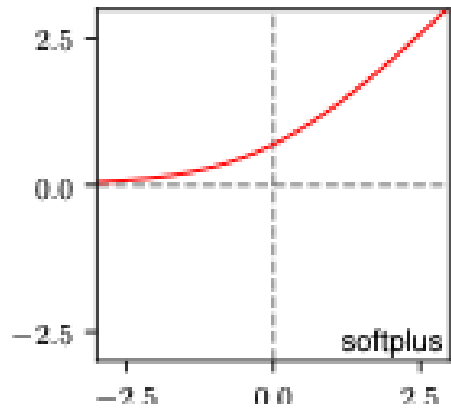
$$h(a) = \max(0, a)$$

Neurons with negative activations will receive "no error signal".



$$h(a) = \max(0, a) + \alpha \min(0, a)$$

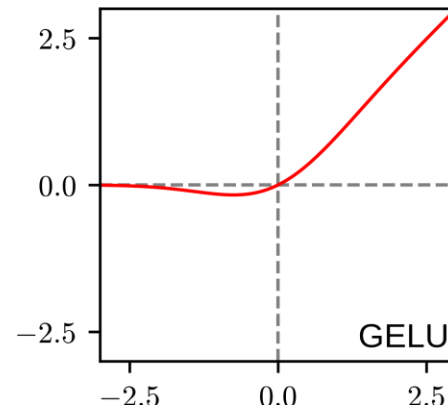
A small slope for negative inputs keeps the gradient nonzero.



$$h(a) = \ln(1 + e^a)$$

$$h(a) \simeq a \text{ when } a \gg 1$$

Also called soft ReLU.



$$h(a) = a \Phi(a)$$

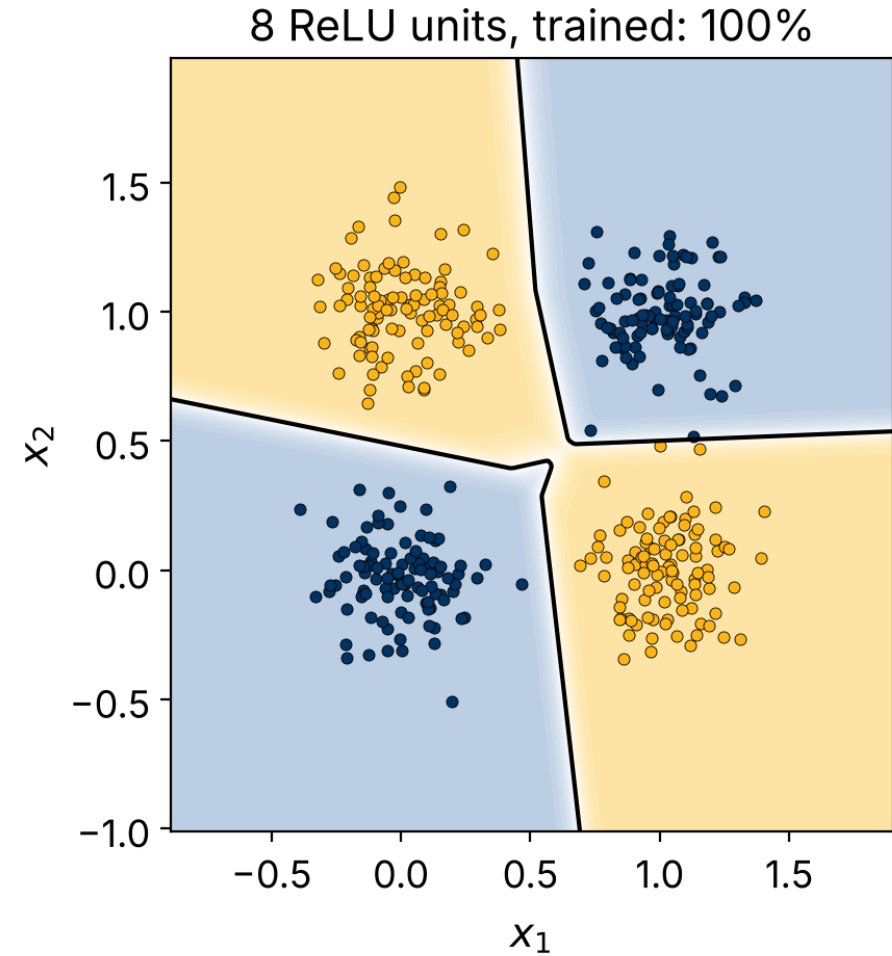
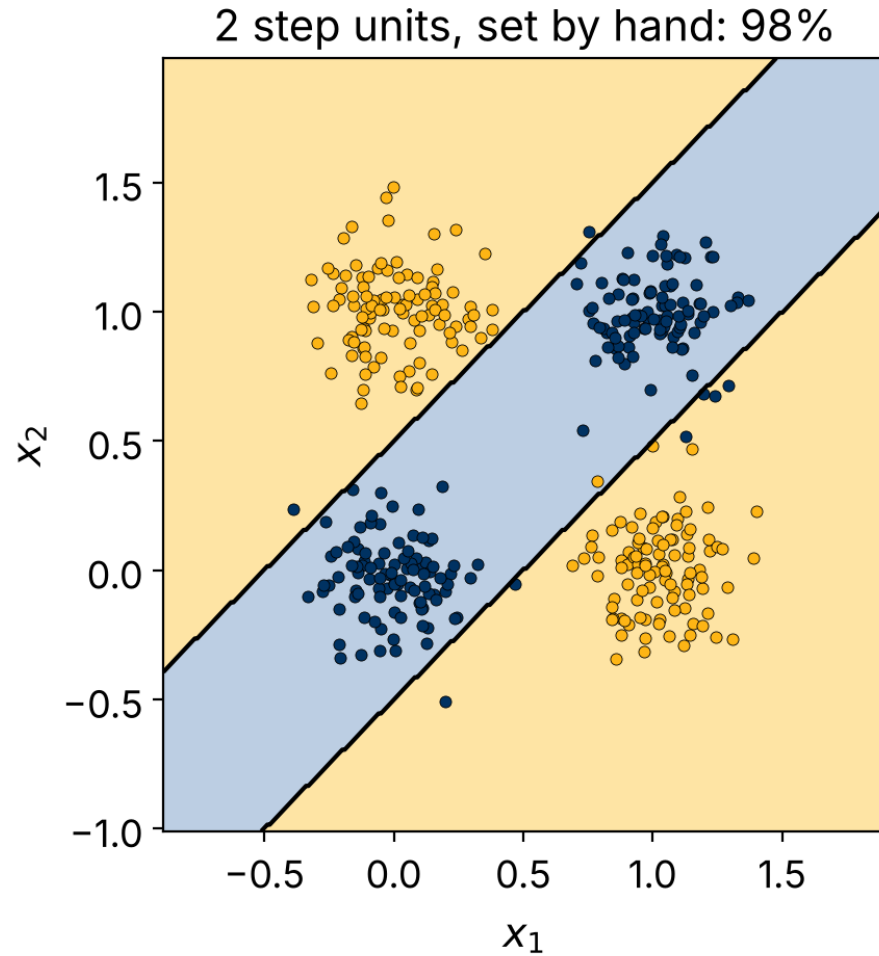
$$\Phi(a) = P(Z \leq a), \quad Z \sim \mathcal{N}(0,1)$$

A smooth ReLU, used in transformers. It dips slightly below 0 for negative inputs.

Training a ReLU Network on XOR



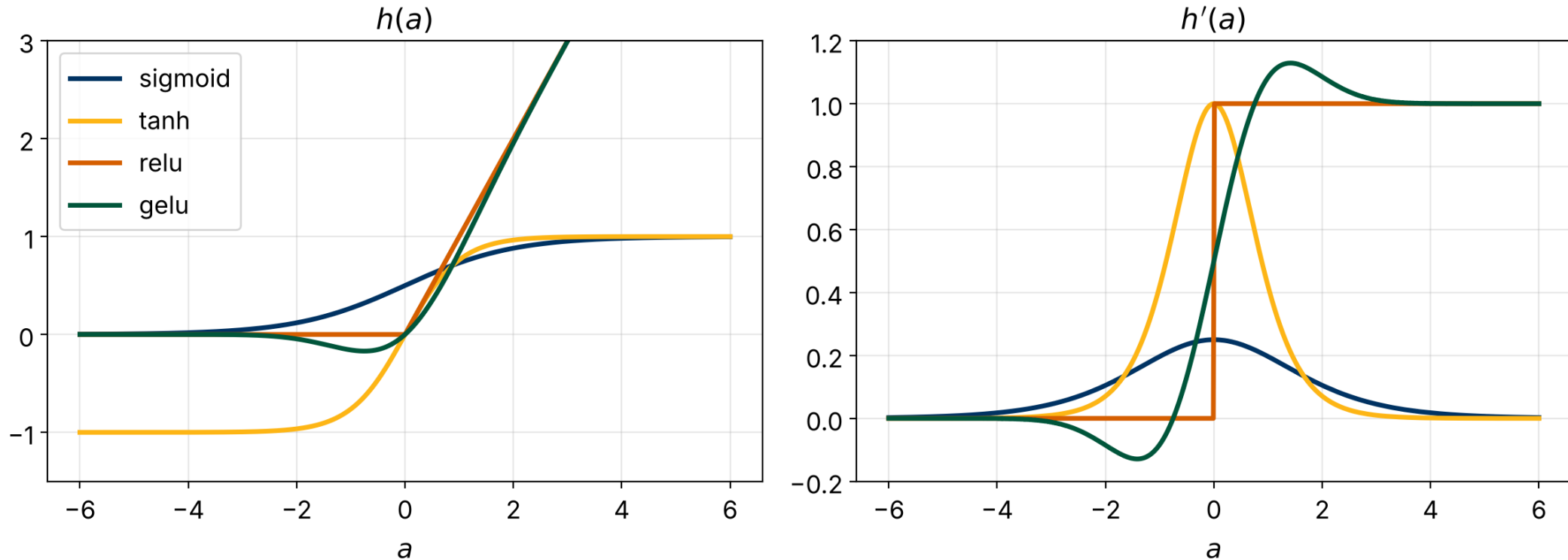
2095864



Where Gradients Vanish



2095864



$$\sigma'(a) = \sigma(a)(1 - \sigma(a)) \leq 1/4 \quad \tanh'(a) = 1 - \tanh^2(a) \quad \text{ReLU}'(a) = \begin{cases} 1 & a > 0 \\ 0 & a < 0 \end{cases}$$

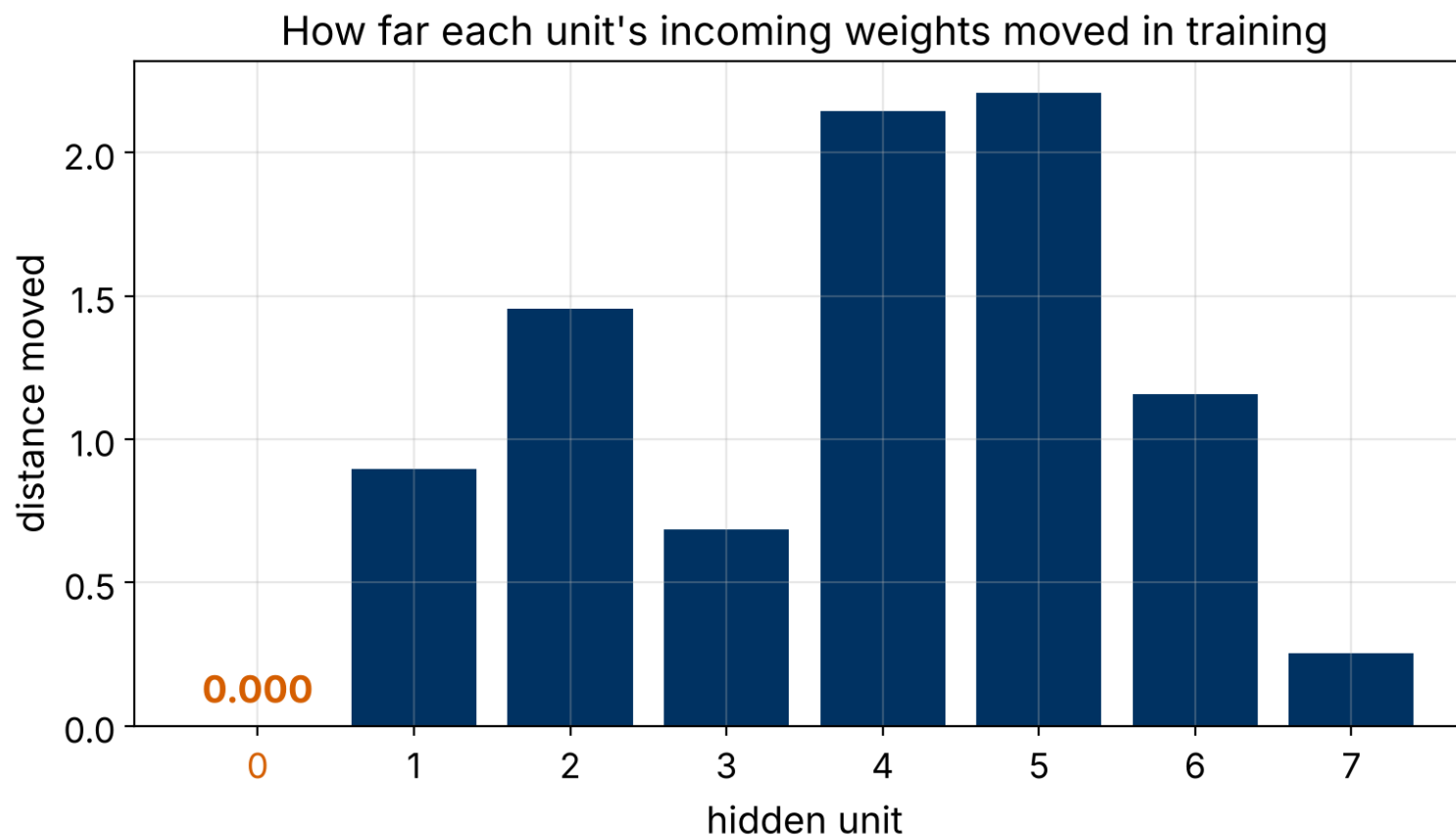


In a ReLU network, one hidden unit's pre-activation a is negative for every training point. What happens to that unit's incoming weights during gradient descent?

A Dead ReLU Unit



2095864



Unit 0 has a negative pre-activation on all 400 training points.

Output Layers and Loss

- The Neuron
- Architecture
- Activation Functions
- Output Layers and Loss
- PyTorch (Optional)



2095864



2095864

The Output Layer Is a GLM

From linear
regression
lectures

$$y = f\left(\sum_j w_j \phi_j(x) + w_0\right)$$

Neural network

$$y = f(W^{(L)} z^{(L-1)} + b^{(L)})$$

$$z^{(L-1)} = \phi(x; W^{(1)}, \dots, W^{(L-1)})$$



2095864

Match the Output Layer to the Target

Target	Output activation	Likelihood	Loss	PyTorch loss
a real number	identity	Gaussian	squared error	<code>nn.MSELoss</code>
one of 2 classes	sigmoid	Bernoulli	binary cross-entropy	<code>nn.BCEWithLogitsLoss</code>
one of K classes	softmax	categorical	cross-entropy	<code>nn.CrossEntropyLoss</code>



2095864

Multi-Label: One Sigmoid per Label

One of K classes

a digit from 0 to 9

$$p_k = \frac{e^{a_k}}{\sum_j e^{a_j}}$$

$$\sum_k p_k = 1$$

$$E = - \sum_k t_k \ln p_k$$

Any subset of K labels

every animal in a photo

$$p_k = \sigma(a_k)$$

$$E = - \sum_k [t_k \ln p_k + (1 - t_k) \ln (1 - p_k)]$$



2095864

- The Neuron
- Architecture
- Activation Functions
- Output Layers and Loss
- PyTorch

Building It in PyTorch



2095864

Big Concepts in PyTorch

- **torch.tensor** is the basic datatype used to encode data, weights, and activations and is very similar `numpy.ndarrays`.
- Neural networks are built using the many primitives in the **torch.nn** module and defining the **forward** process of making a prediction.
- PyTorch uses automatic differentiation to efficiently compute the **gradient** (Lecture 13).
- Training is accomplished by **(you) implementing** some variation of gradient descent.



2095864

torch.nn.Module

When building a new model you extend the nn.Module

```
class MLP(nn.Module):
```

Extend nn.Module

```
    def __init__(self, dims, activation="relu"):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(dims[i], dims[i+1]) for i in range(len(dims)-1)])
        if activation == "relu": self.act = nn.functional.relu
        else: self.act = torch.tanh

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            x = layer(x)
            if i < len(self.layers)-1:
                x = self.act(x)

        return x
```

Add modules to your module. These add parameters.

Compute the **forward** pass using the layer modules.



2095864

Implement Gradient Descent with Optimizer

In PyTorch you implement **your own gradient descent loop** but you can use built in update rules (e.g., Adam) called **optimizers**.

```
loader = DataLoader(tr_data, batch_size=32, shuffle=True)
```

```
optimizer = torch.optim.Adam(model.parameters(), eta)
```

```
for epoch in range(nepochs):
```

```
    for (x, t) in loader:
```

```
        optimizer.zero_grad()
```

```
        loss = loss_fn(model(x), t)
```

```
        loss.backward()
```

```
        optimizer.step()
```

Minibatch SGD
DataLoader

Adam optimizer

Epoch is one pass over data

Load one minibatch at a time

Compute the gradient
of the loss

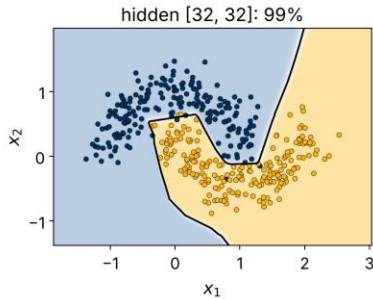
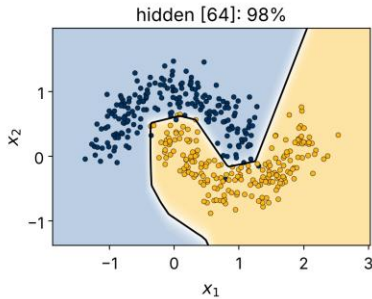
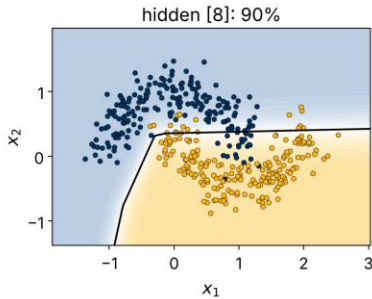
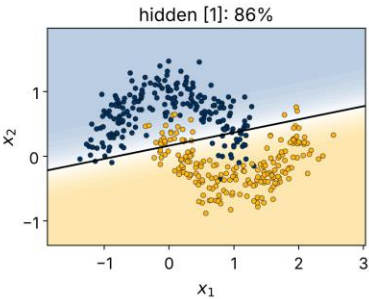
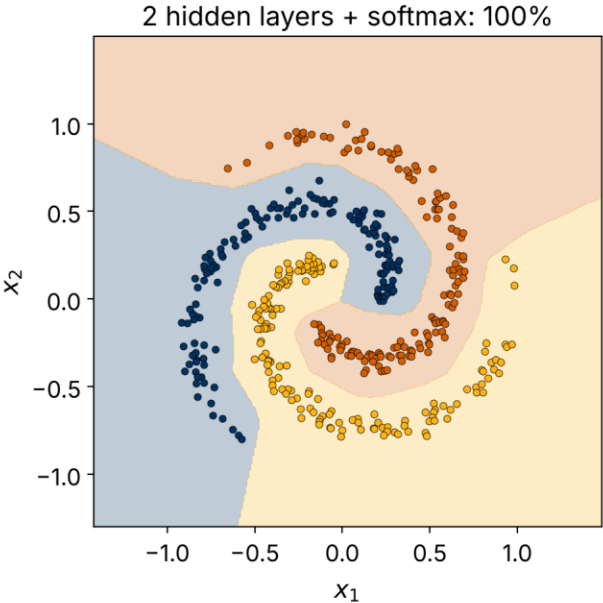
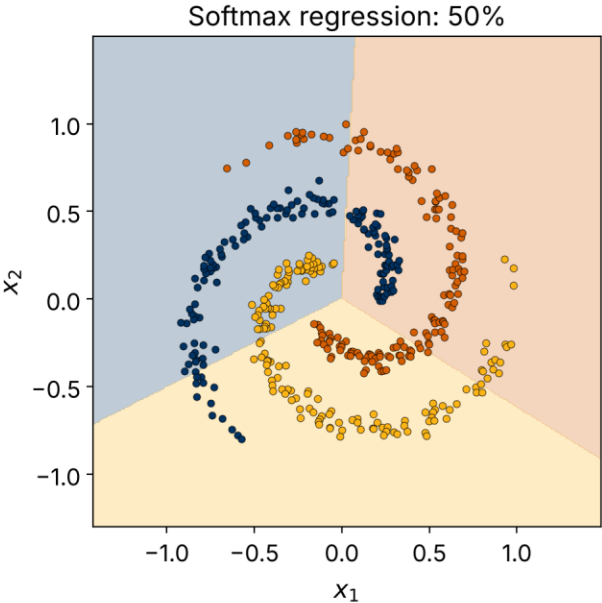
Use optimizer to update all the parameters (1-step)



2095864

Demo

Width, depth, and activation



Lecture 12

Neural Networks (1)

Credit: Joseph E. Gonzalez and Narges Norouzi

Reference: Bishop & Bishop, *Deep Learning: Foundations and Concepts*, Ch. 6, Sections 6.1, 6.2.1–6.2.3, 6.3.1, 6.4