CS 696 Applied Large Langauge Models
Spring Semester, 2025
Doc 2 NN Review
Jan 21, 2025

Acknowledgment

Some slides in this lecture are from Stanford's Speech and Language Processing (3rd ed. draft)
Dan Jurafsky and James H. Martin

https://web.stanford.edu/~jurafsky/slp3/

Language Model History

Linguists were building models by hand

1980's Statistical Models

2000s - 2010s neural language models

2017 - Transformer Architecture

2018 - Large Language Models

Neural Network Model Basic Idea

Given a sequence of words, predict the next word

Example:

It was a dark and stormy X

It was a dark and stormy X

X	Score
night	0.894
evening	0.045
day	0.039
morning	0.009

Attention

The chicken didn’t cross the road because it

The chicken didn’t cross the road because it was too tired

The chicken didn’t cross the road because it was too wide

Attention is All You Need

2017 paper from Google

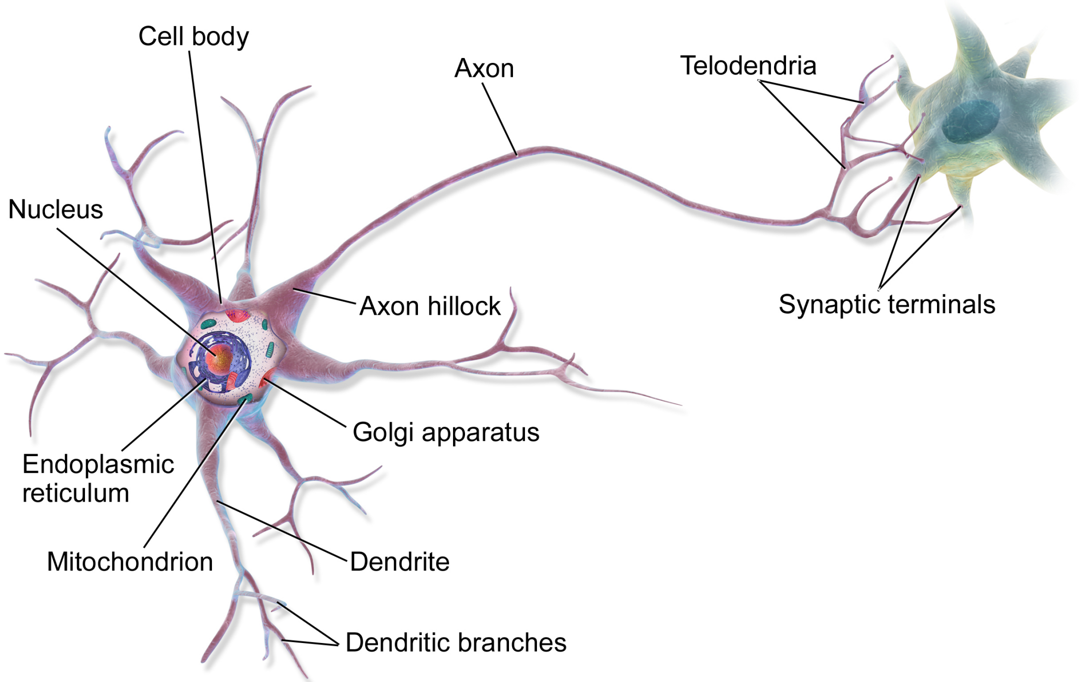
Introduced Transformer Architecture

Bases for Large Language Models

Uses context of a word

- Before and after

This is in your brain



Neural Network Unit

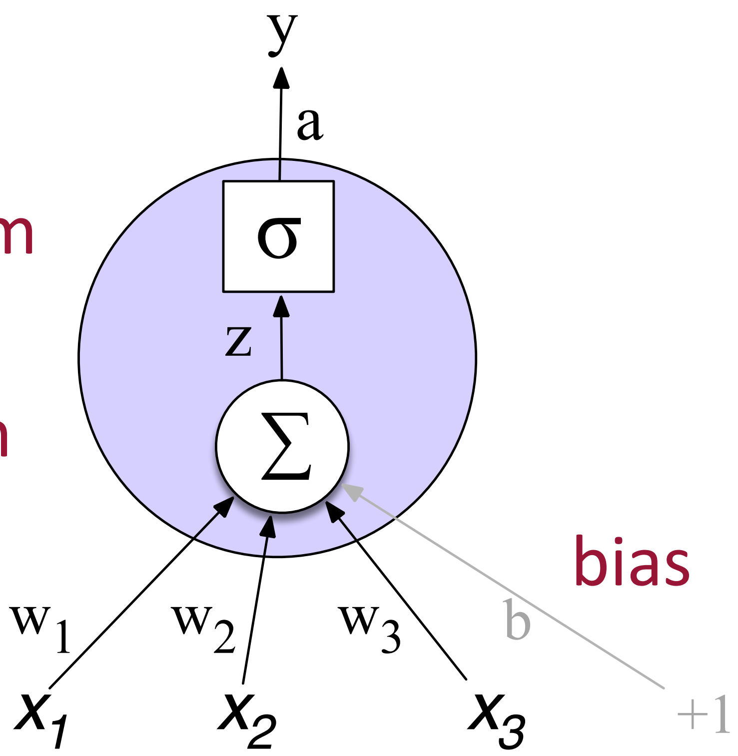
Output value

Non-linear transform

Weighted sum

Weights

Input layer



Neural unit

Take weighted sum of inputs, plus a bias

$$z = b + \sum_i w_i x_i$$

$$z = w \cdot x + b$$

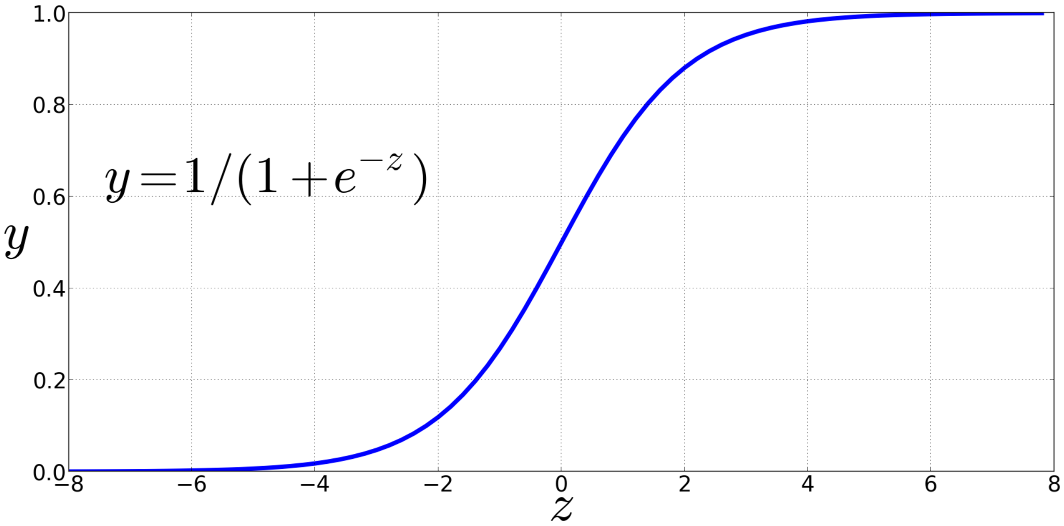
Instead of just using z, we'll apply a nonlinear activation function f:

$$y = a = f(z)$$

Non-Linear Activation Functions

Sigmoid

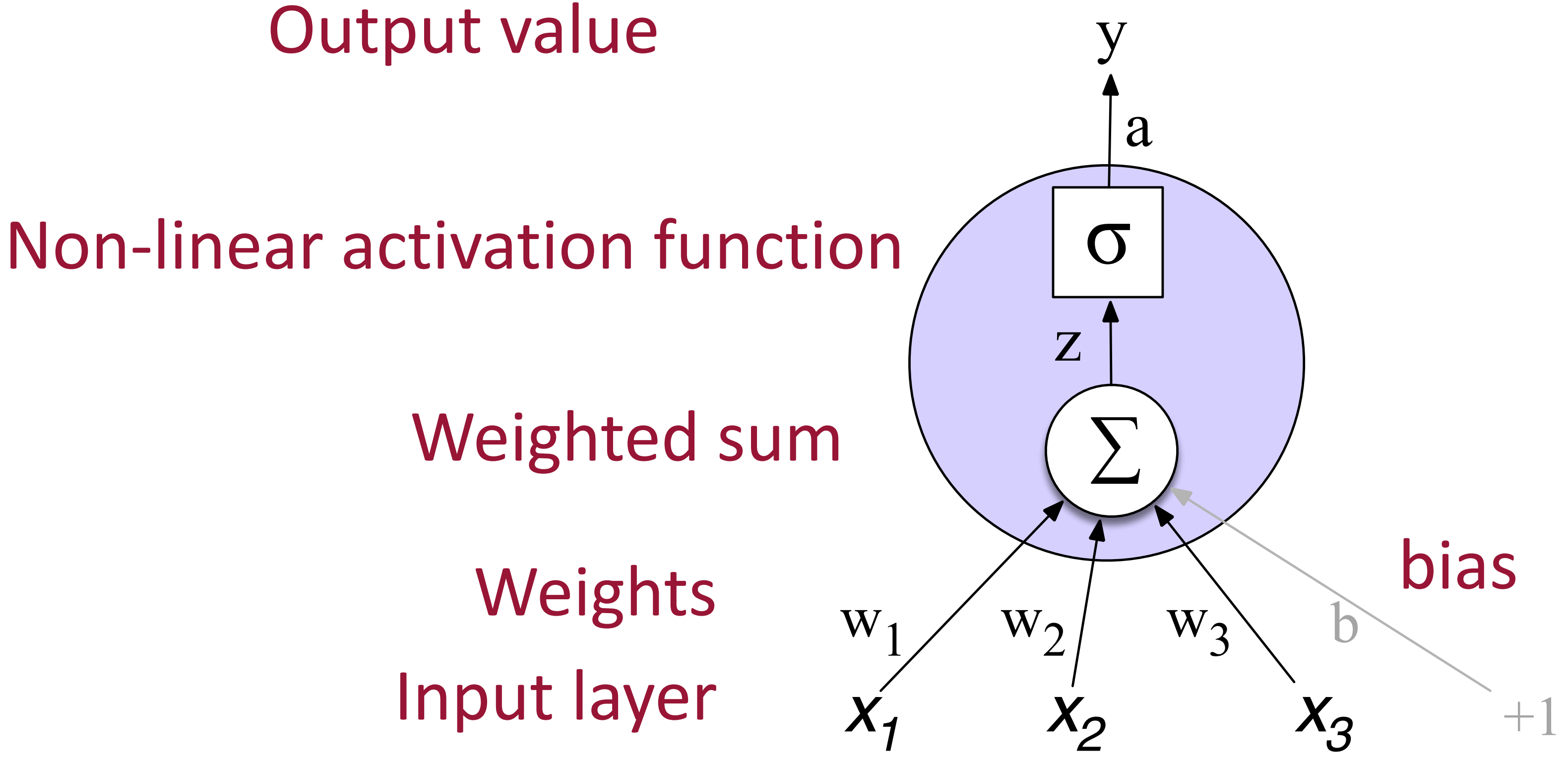
$$y = \sigma(z) = \frac{1}{1+e^{-z}}$$



Final function the unit is computing

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

Final unit again



An example

Suppose a unit has:

w = [0.2,0.3,0.9]

b = 0.5

What happens with input x:

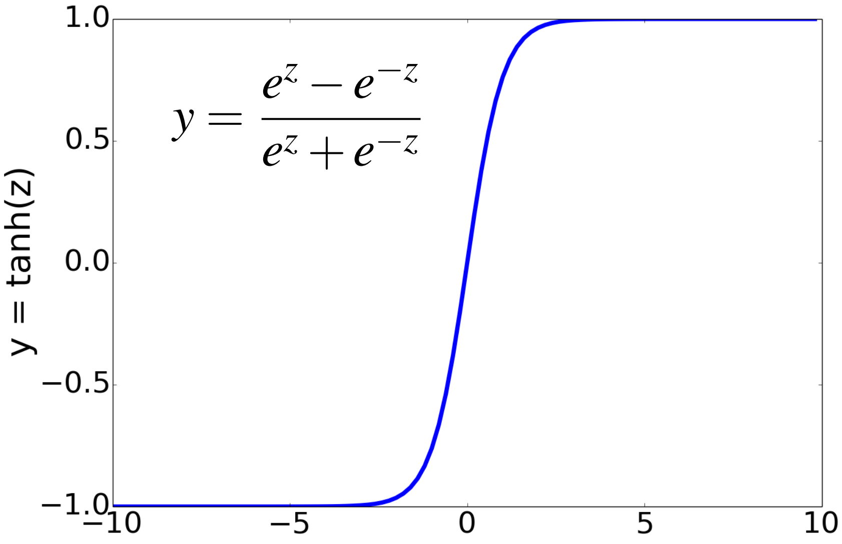
x = [0.5,0.6,0.1]

$$y = \sigma(w \cdot x + b) = \frac{1}{1+e^{-(w \cdot x + b)}} =$$

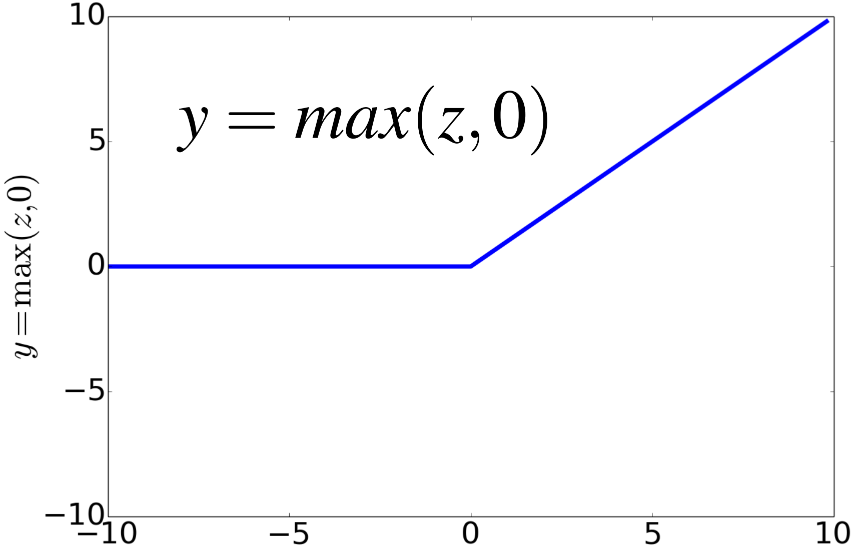
$$\frac{1}{1+e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1+e^{-0.87}} = .70$$

Non-Linear Activation Functions besides sigmoid

Most Common:



tanh



ReLU

Rectified Linear Unit

The XOR problem

Minsky and Papert (1969)

Can neural units compute simple functions of input?

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Perceptrons

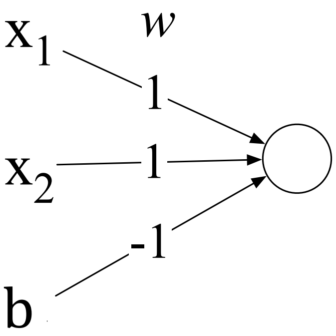
A very simple neural unit

- Binary output (0 or 1)
- No non-linear activation function

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

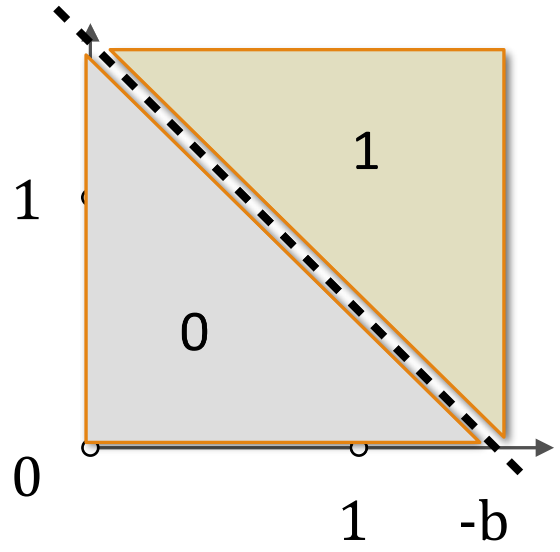
Easy to build AND with perceptrons

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



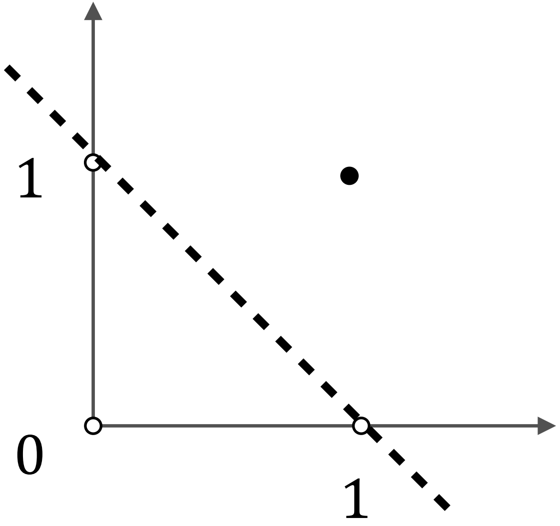
$$\mathrm{w \cdot x + b = 0}$$

Let w = 1

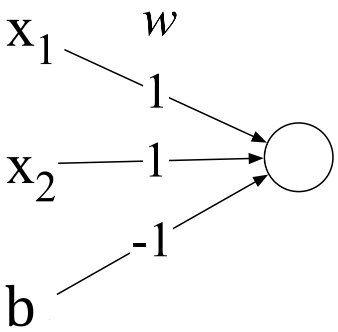
x1 + x2 + b = 0

x1 = -b - x2

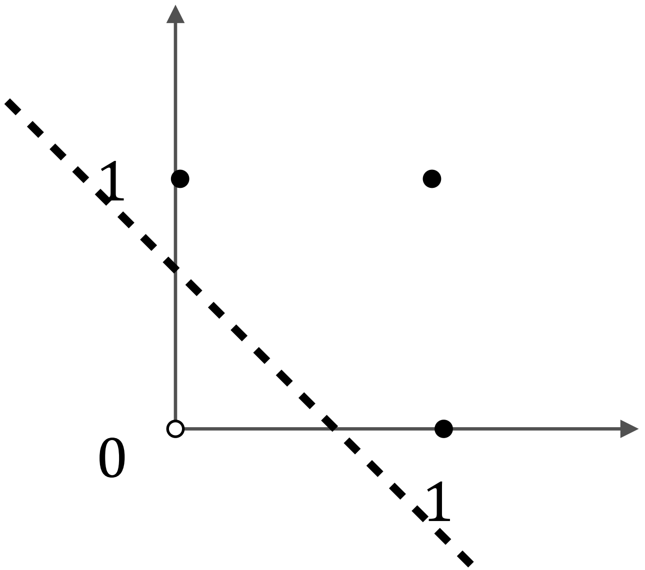
$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



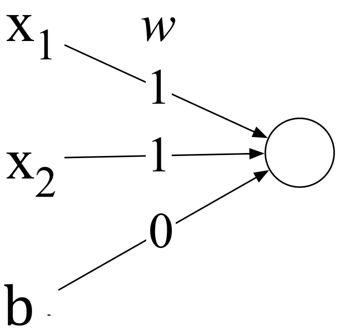
AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



$$y = \begin{cases} 0, & \text{if } w \cdot x + b \le 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

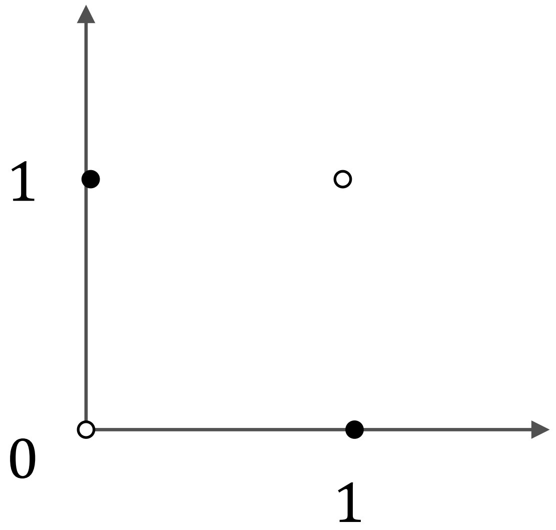


Floating Point Operations are Expensive

Early 2023, ChatGPT's daily electricity usage

560 MWh per day

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Why? Perceptrons are linear classifiers

Perceptron equation given x_1 and x_2, is the equation of a line

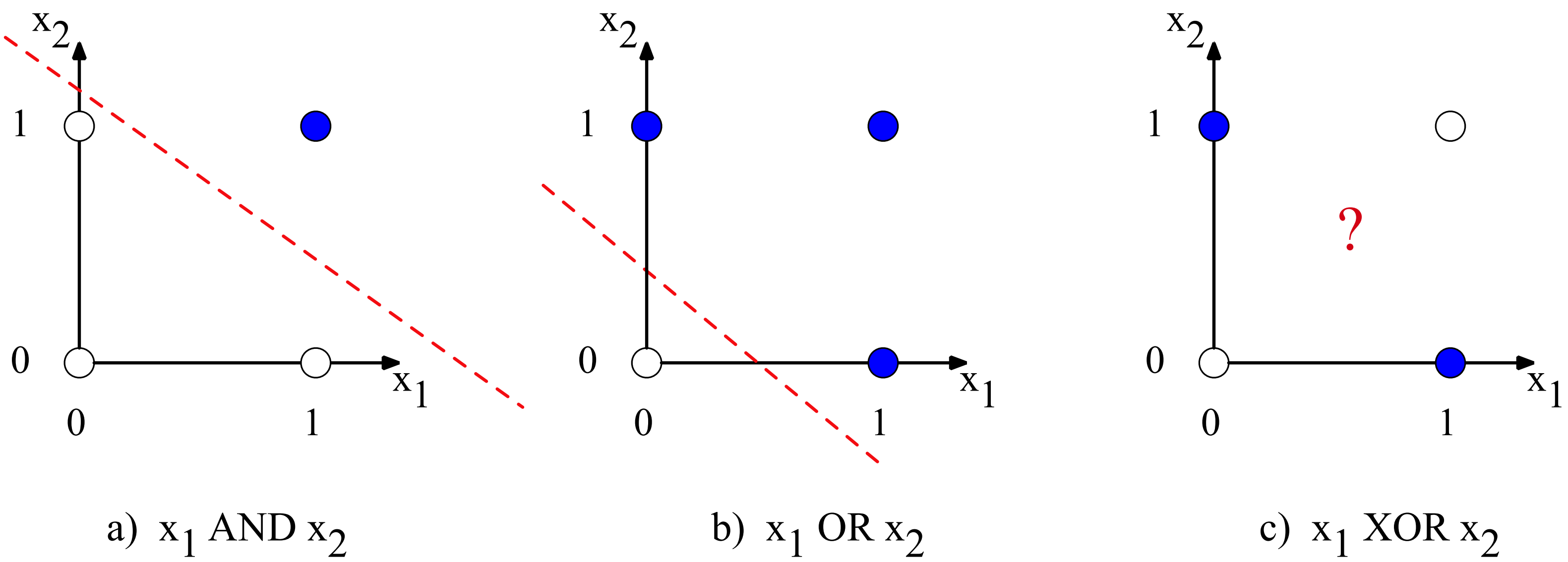
$$w_1x_1 + w_2x_2 + b = 0$$

(in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)

This line acts as a **decision boundary**

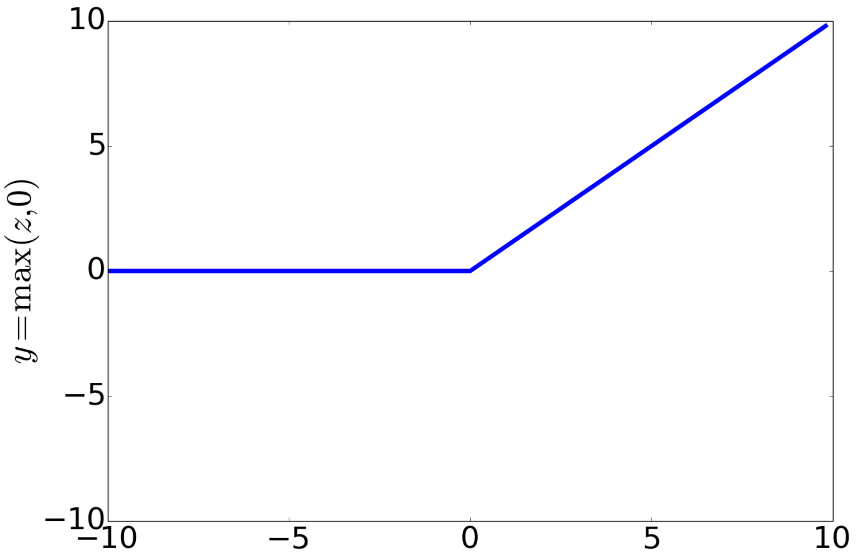
- 0 if input is on one side of the line
- 1 if on the other side of the line

Decision boundaries



XOR is not a **linearly separable** function!

Rectified Linear Unit - ReLU



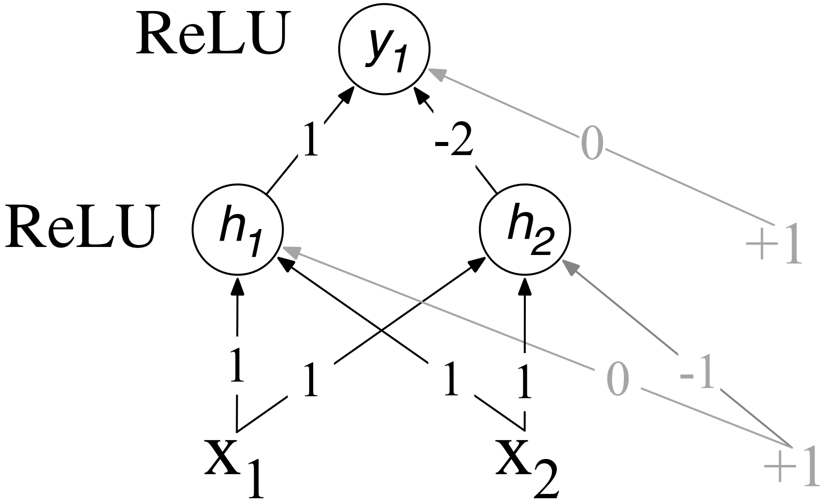
$$y = max(z, 0)$$

Solution to the XOR problem

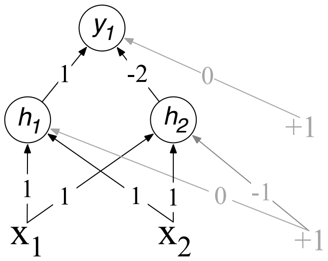
XOR **can't** be calculated by a single perceptron

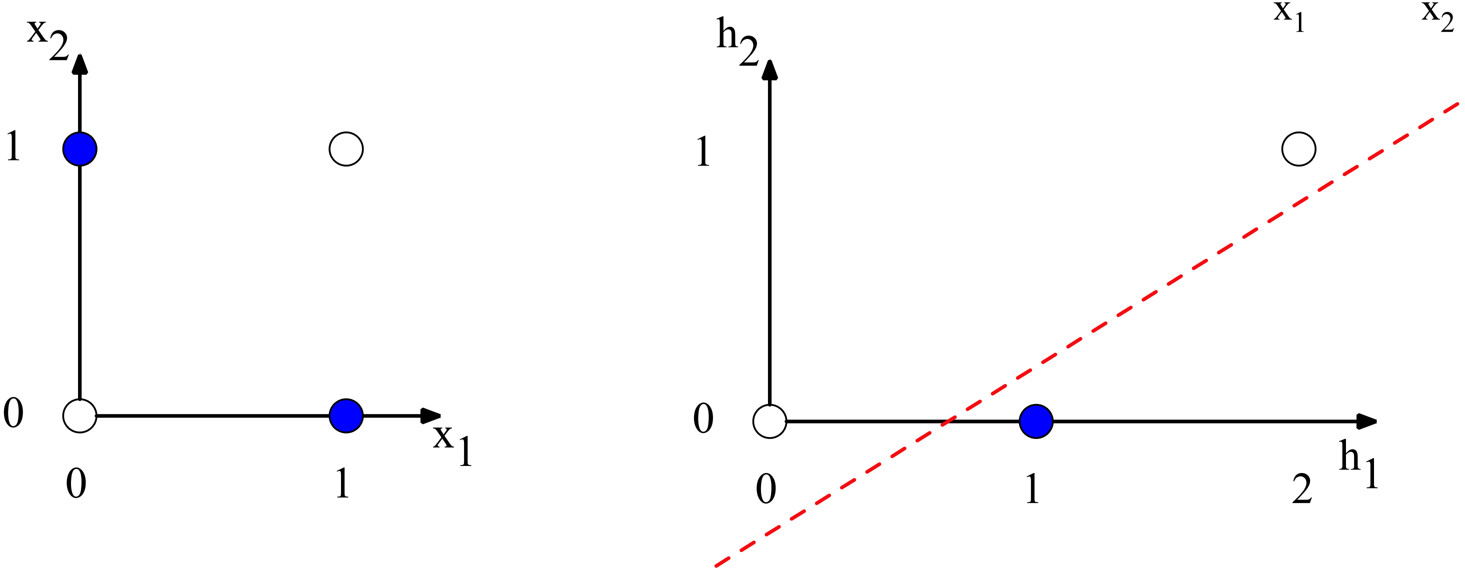
XOR **can** be calculated by a layered network of units.

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



The hidden representation h





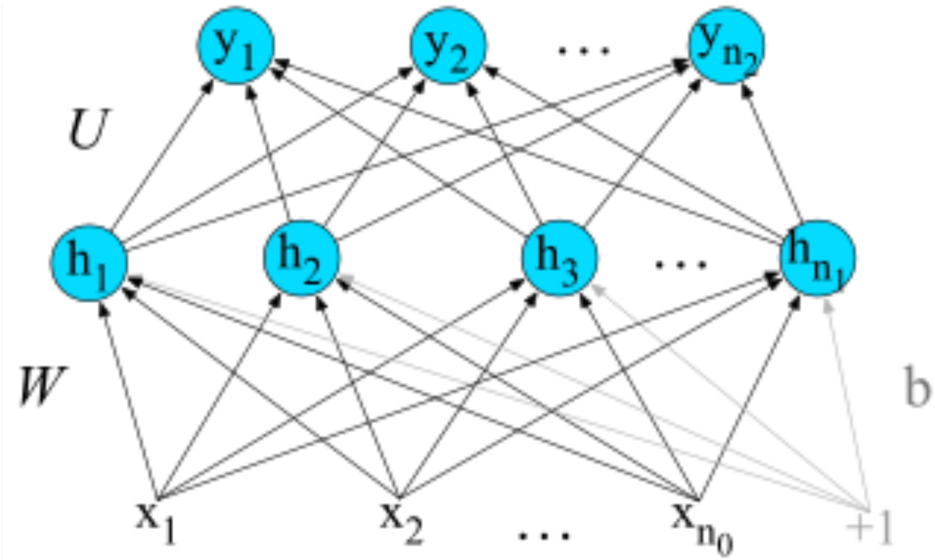
a) The original *x* space

b) The new (linearly separable) *h* space

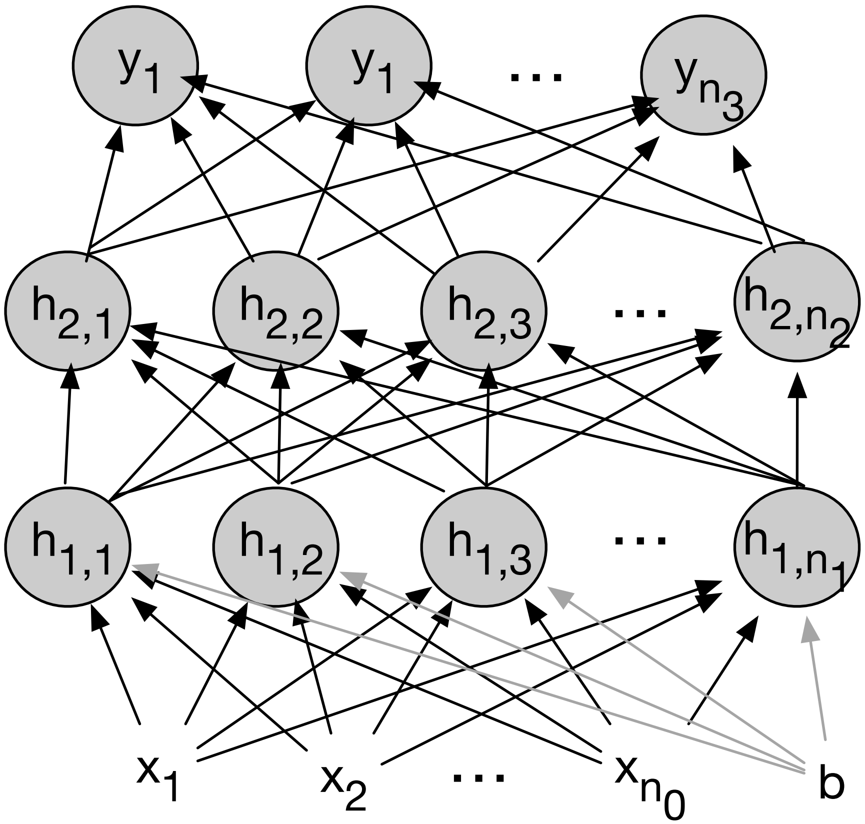
(With learning: hidden layers will learn to form useful representations)

Feedforward Neural Networks

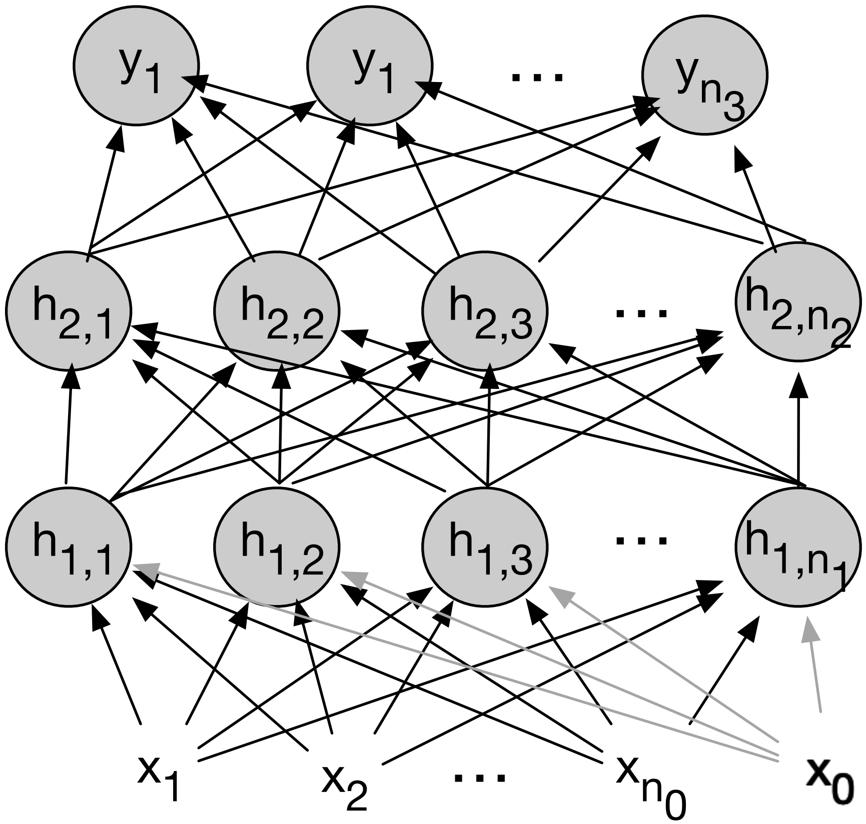
Can also be called **multi-layer perceptrons** (or **MLPs**) for historical reasons



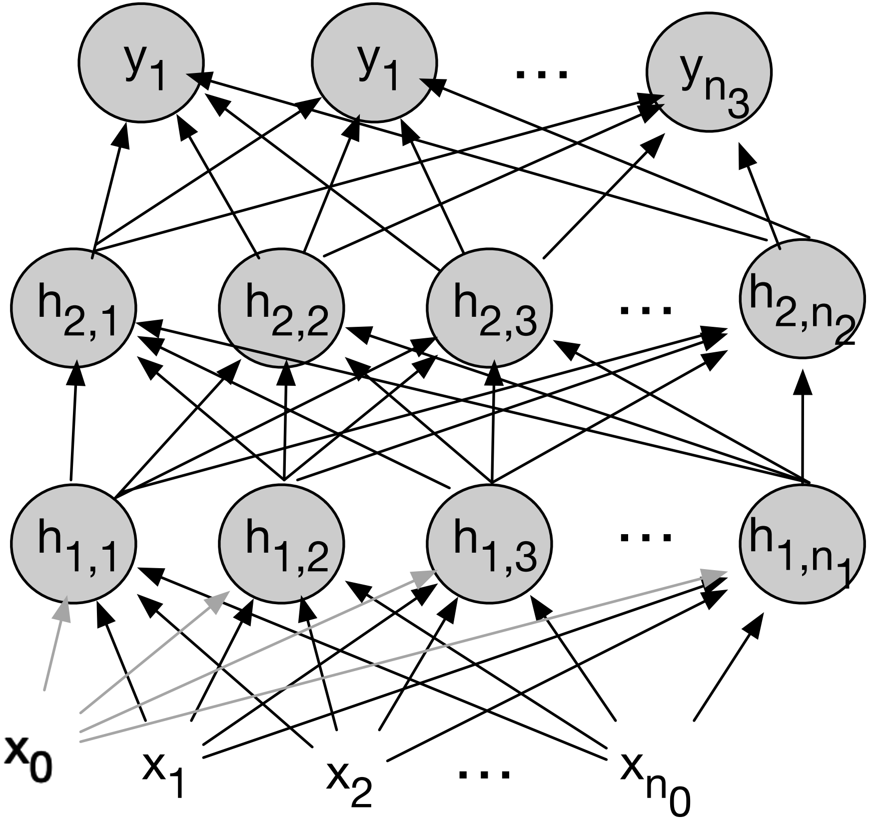
Feedforward Neural Networks



Feedforward Neural Networks



Feedforward Neural Networks



Final unit again

Output value

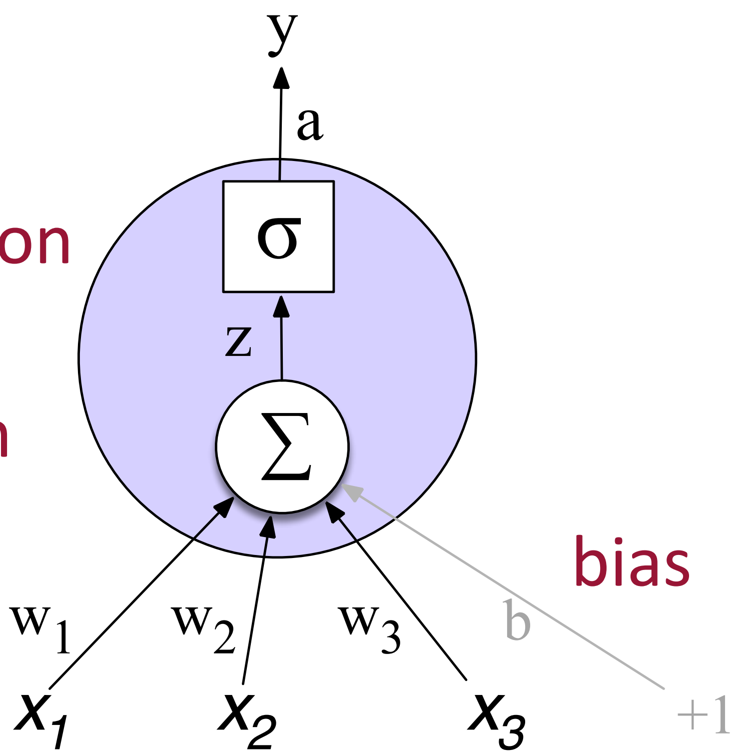
Non-linear activation function

Weighted sum

Weights

Input layer

bias



Final unit again

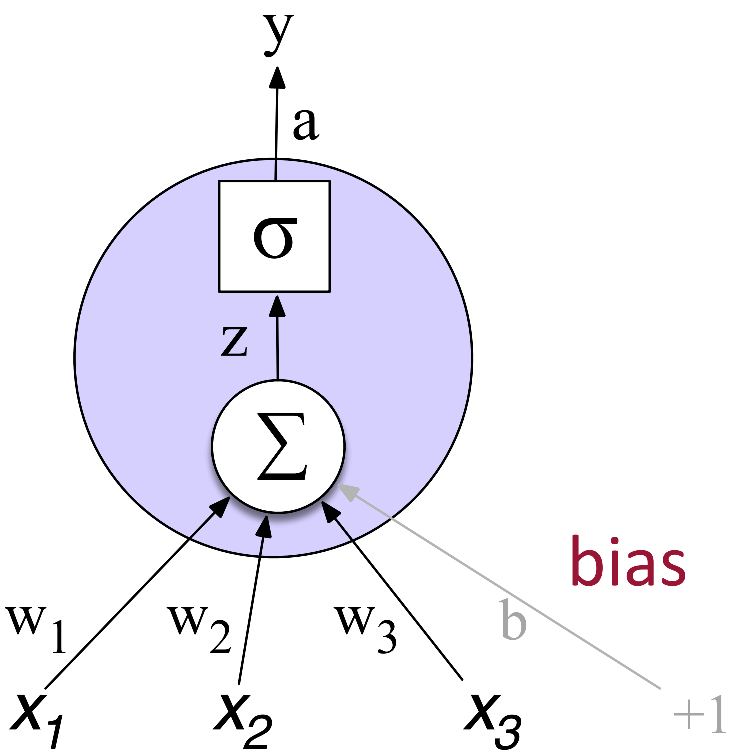
$z = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n$

$\bar{w} = [w_0, w_1, \ldots w_n]$

$\bar{x} = [x_0, x_1, \ldots x_n]$

$z = \bar{w} \cdot \bar{x}$ or $\bar{w} * \bar{x}^T$

$a = \sigma(\bar{w} \cdot \bar{x})$



For a Layer

$\bar{W}_1 = [\bar{w}_{1,1}$

$\bar{w}_{1,2}$

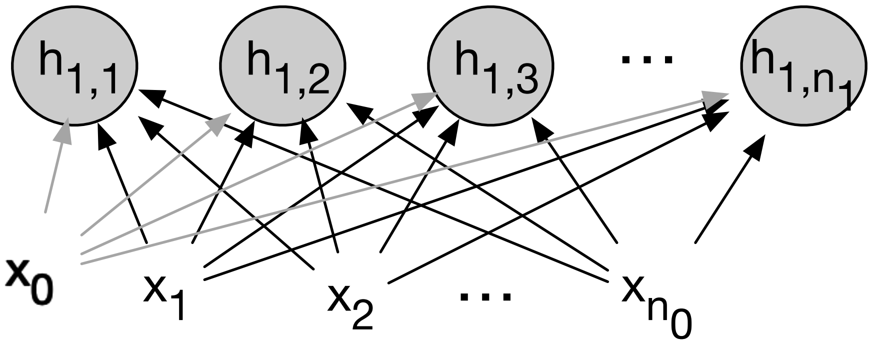
…

$\bar{w}_{1,n1}]$

$\bar{W}_{1*}\bar{x}^T$

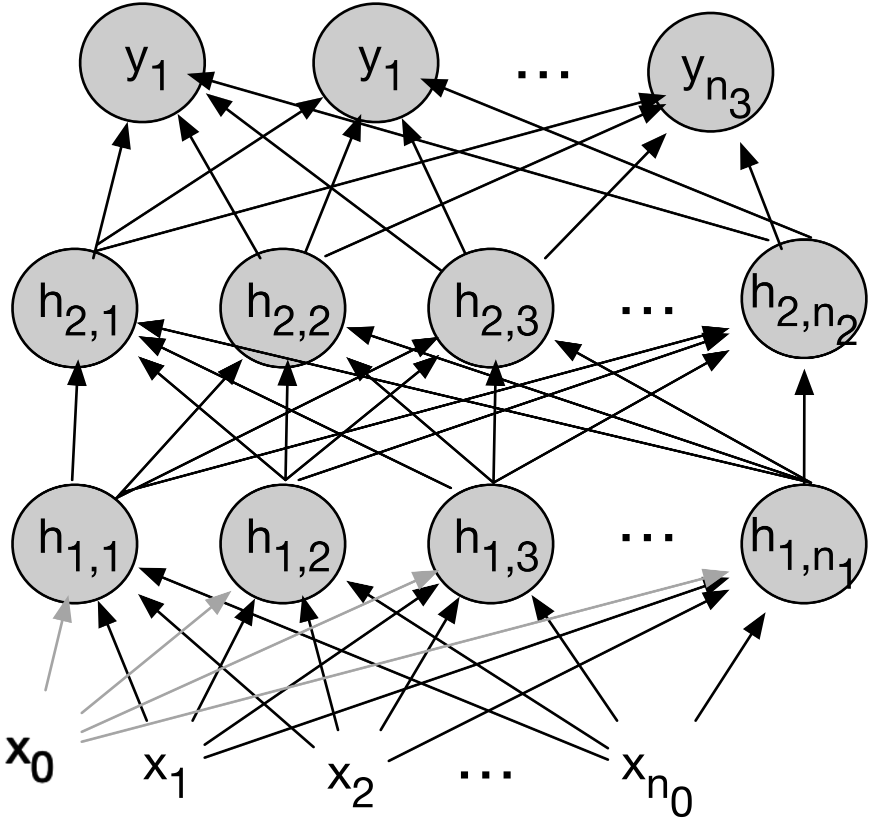
$\sigma(\bar{W}_{1*}\bar{x}^T)$

$\sigma(\bar{w}_{1,1} \cdot \bar{x})$ $\sigma(\bar{w}_{1,2} \cdot \bar{x})$ $\sigma(\bar{w}_{1,3} \cdot \bar{x})$ $\sigma(\bar{w}_{1,n1} \cdot \bar{x})$



For A Network of Layers

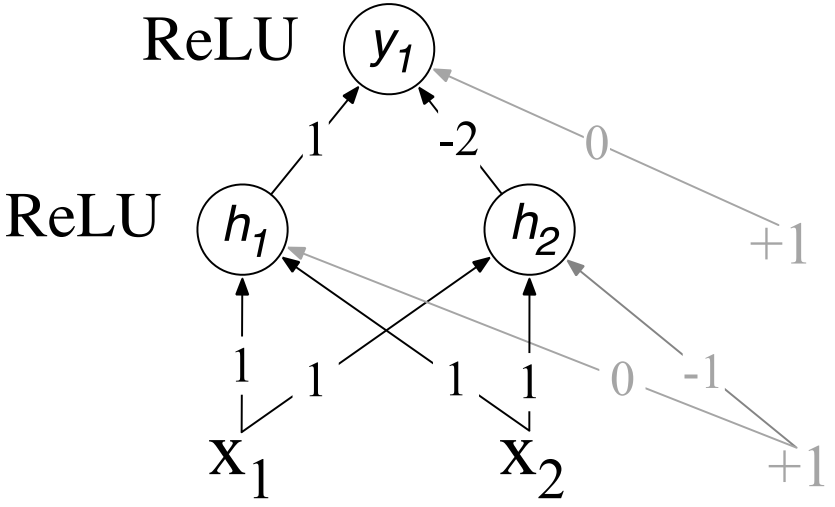
$\sigma(\bar{W}_3 * \sigma(\bar{W}_2 * \sigma(\bar{W}_{1*} \bar{x}^T)))$



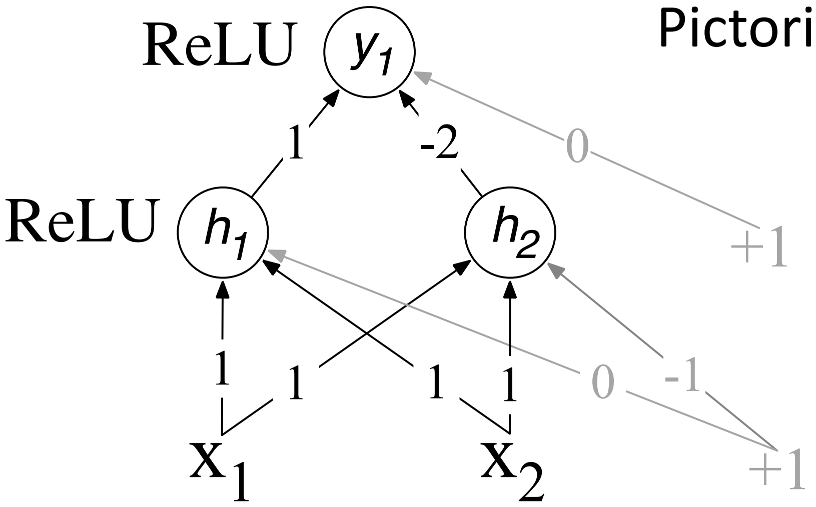
Recall XOR Solution

```
using LinearAlgebra

σ(x) = max(x, 0)
W̄1 = [0 1 1;
      -1 1 1]
W̄2 = [1 -2]
xorNN(x) = σ.(W̄2 * σ.(W̄1 * x))

using Test
@test xorNN([1, 0, 0]) == [0]
@test xorNN([1, 0, 1]) == [1]
@test xorNN([1, 1, 0]) == [1]
@test xorNN([1, 0, 0]) == [0]
```

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



What is a Neural Net?



Pictorial representation of Neural Net

This is Not a Pipe

The Treachery of Images by René Magritte

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

$$\bar{W}1 = \begin{bmatrix} 0 & 1 & 1; \\ -1 & 1 & 1 \end{bmatrix}$$

$\bar{W}2 = [1\ -2]$

$xorNN(x) = \sigma(\bar{W}2 * \sigma(\bar{W}1 * x))$

Downloading a Model

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    attn_implementation='eager',
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")
```

A Pipeline

```
from transformers import pipeline

# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=True,
    max_new_tokens=500,
    do_sample=False
)
```

Running the Model

```
# The prompt (user input / query)
messages = [
    {"role": "user", "content": "Create a funny joke about chickens."}
]

# Generate output
output = generator(messages)
print(output[0]["generated_text"])
```

Why did the chicken join the band? Because it had the drumsticks!

Downloading a Model

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    attn_implementation='eager',
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")
```

What was downloaded?

Topics

Architecture

Training

Input

Performance

Architecture

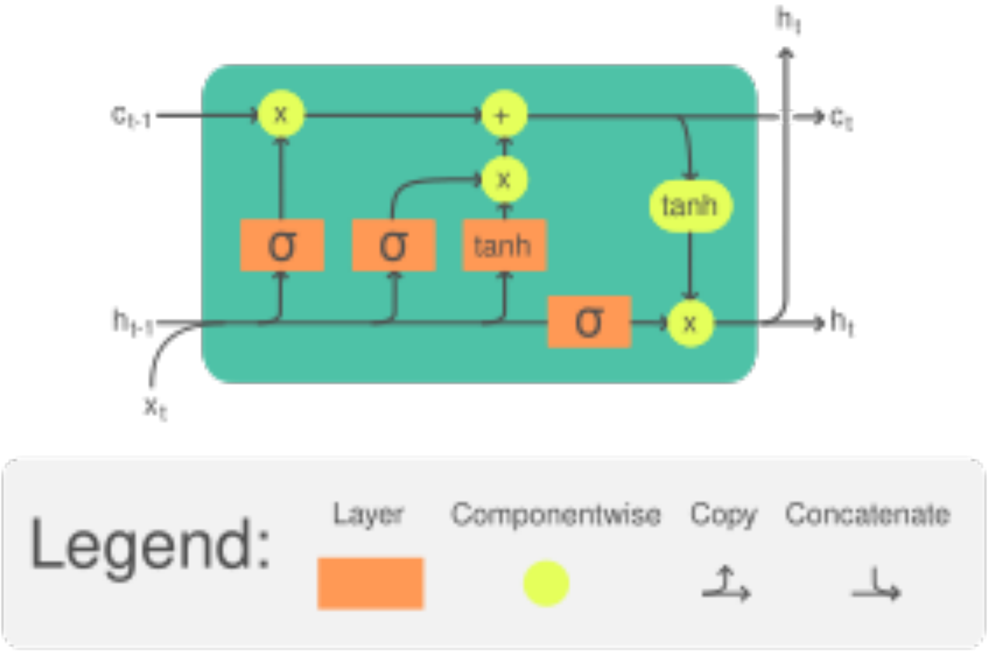
Number of Layers

Size of each layer

Connections

Structure of Nodes

Performance



Training

How to determine the value of the parameters

- Supervised
- Unsupervised

Input

- Different types and sizes
  - Text
  - Images
  - Sound
  - Video
- How to encode for the NN

Performance

Memory & Time

Llama 403b

- 30,840,000 GPU hours to train on Nvidia h800

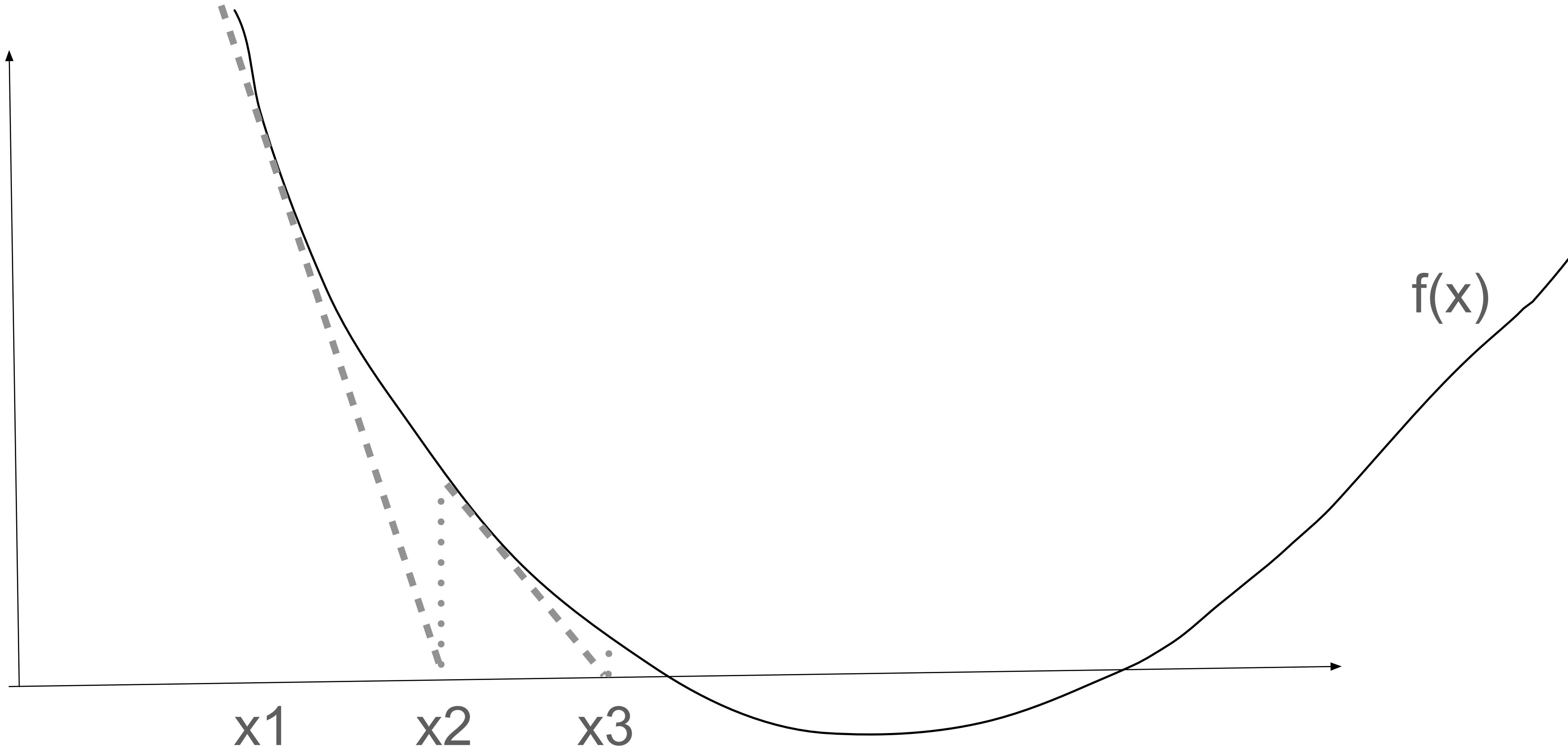
Microsoft - $80 Billion on AI data centers in 2025

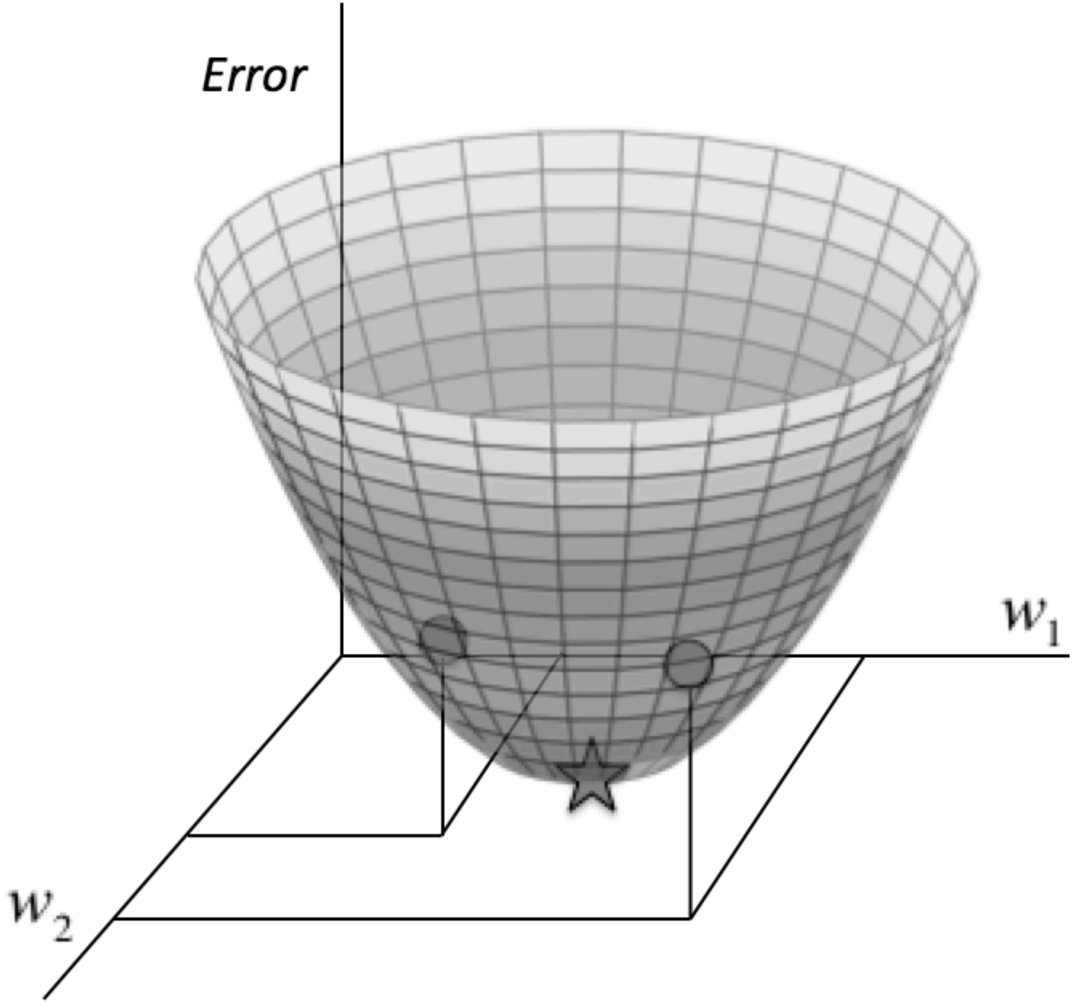
How to run models on

- Laptops
- Phones

Training - Supervised

First a review of gradient descent


f(x)
x1
x2
x3


Error
w_1
w_2

Systematic way to change weights

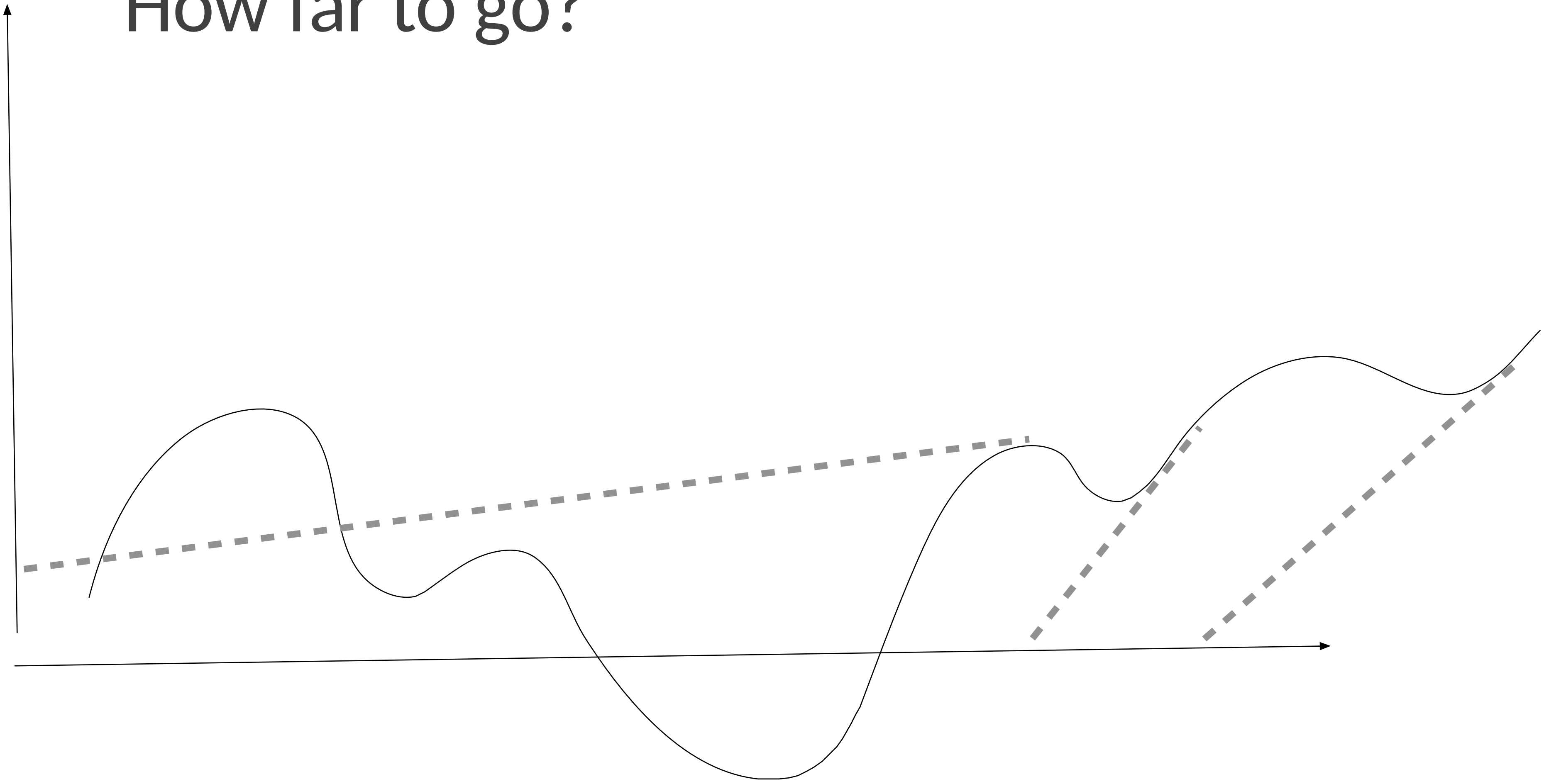
$f(x1, x2) = w1*x1 + w2*x2 + b$

Take derivative of activation function get gradient

Use the slope in the x1 dimension to adjust w1

Use the slope in the x2 dimension to adjust w2

How far to go?



Learning Rate

To avoid overshooting multiply the gradient by a factor - say 0.1
This is called the learning rate

Take derivative of activation function to get gradient
Use the slope in the x1 dimension * learning rate to adjust w1
Use the slope in the x2 dimension * learning rate to adjust w2

Example

```
import numpy as np

def loss_function(w):
    return (w - 3)**2

def gradient(w):
    return 2 * (w - 3)
```

```
learning_rate = 0.1
epochs = 20
w = 0.0
print(f"Initial weight: {w}")
for epoch in range(epochs):
    grad = gradient(w)
    w = w - learning_rate * grad
    loss = loss_function(w)
    print(f"Weight = {w:.4f}, Loss = {loss:.4f}")
```

Initial weight: 0.0	Initial weight: 100.0
Initial weight: 0.0 Learning Rate: 0.01 Epoch 1: Weight = 0.0600, Loss = 8.6436 Epoch 20: Weight = 0.9972, Loss = 4.0113	Initial weight: 100.0 Learning Rate: 0.01 Epoch 1: Weight = 98.0600, Loss = 9036.4036 Epoch 20: Weight = 67.7580, Loss = 4193.5951
Initial weight: 0.0 Learning Rate: 0.1 Epoch 1: Weight = 0.6000, Loss = 5.7600 Epoch 20: Weight = 2.9654, Loss = 0.0012	Initial weight: 100.0 Learning Rate: 0.1 Epoch 1: Weight = 80.6000, Loss = 6021.7600 Epoch 20: Weight = 4.1183, Loss = 1.2507
Initial weight: 0.0 Learning Rate: 1.0 Epoch 1: Weight = 6.0000, Loss = 9.0000 Epoch 2: Weight = 0.0000, Loss = 9.0000 Epoch 3: Weight = 6.0000, Loss = 9.0000	Initial weight: 100.0 Learning Rate: 1.0 Epoch 1: Weight = -94.0000, Loss = 9409.0000 Epoch 2: Weight = 100.0000, Loss = 9409.0000 Epoch 3: Weight = -94.0000, Loss = 9409.0000
Initial weight: 0.0 Learning Rate: 10.0 Epoch 1: Weight = 60.0000, Loss = 3249.0000 Epoch 2: Weight = -1080.0000, Loss = 1172889.0000 Epoch 3: Weight = 20580.0000, Loss = 423412929.0000	Initial weight: 100.0 Learning Rate: 10.0 Epoch 1: Weight = -1840.0000, Loss = 3396649.0000 Epoch 2: Weight = 35020.0000, Loss = 1226190289.0000 Epoch 3: Weight = -665320.0000, Loss = 442654694329.0000