Neural networks basics

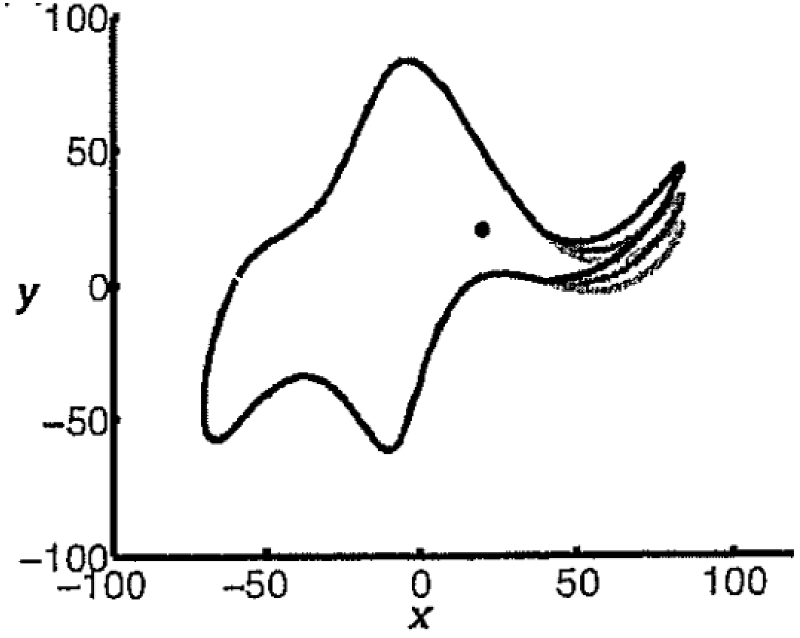
by C. Yao Lai for 2025 Princeton summer school

Outline

- Neural networks in problems governed by PDEs
- NN basics (Lecture 1-3)
  Universal function approximation
  Gradient descent
  Back propagation
- Physics-informed NN (Lecture 4-5)

Von Neumann's elephant

"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.“ -Von Neumann



An approximation using five complex parameters was found by Mayer, Khairy, Howard (2010) based on complex Fourier analysis.

$$x(t) = \sum_{k=0}^{\infty} (A_k^x \cos(kt) + B_k^x \sin(kt)), \quad (1)$$

$$y(t) = \sum_{k=0}^{\infty} (A_k^y \cos(kt) + B_k^y \sin(kt)), \quad (2)$$

Table I. The five complex parameters $p_1, \dots, p_5$ that encode the elephant including its wiggling trunk.

Parameter	Real part	Imaginary part
$p_1 = 50 - 30i$	$B_1^x = 50$	$B_1^y = -30$
$p_2 = 18 + 8i$	$B_2^x = 18$	$B_2^y = 8$
$p_3 = 12 - 10i$	$A_3^x = 12$	$B_3^y = -10$
$p_4 = -14 - 60i$	$A_5^x = -14$	$A_1^y = -60$
$p_5 = 40 + 20i$	Wiggle coeff. = 40	$x_{eye} = y_{eye} = 20$

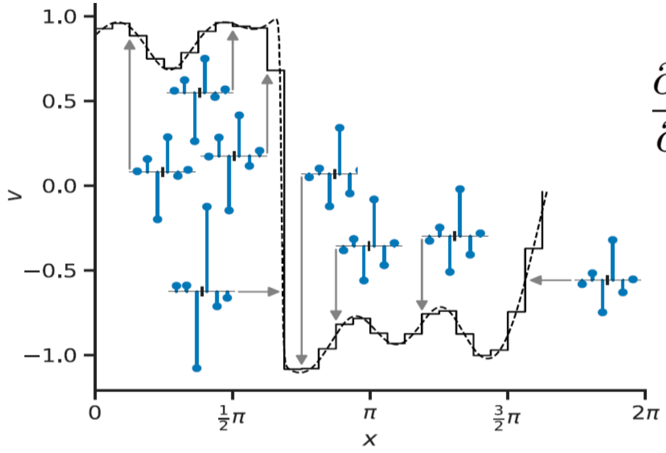
Early day ML-PDE

- **Leaning how to discretize PDEs (2019)**

Learning data-driven discretizations for partial differential equations

Yohai Bar-Sinai[a,1,2], Stephan Hoyer[b,1,2], Jason Hickey[b], and Michael P. Brenner[a,b]

Viscous Burgers eqn



$$\frac{\partial v}{\partial t} = F\left(t, x, v, \frac{\partial v}{\partial x_i}, \frac{\partial v}{\partial x_i \partial x_j}, \cdots\right)$$

$$\boxed{\frac{\partial^n v}{\partial x^n} \approx \sum_i \alpha_i^{(n)} v_i,}$$

ML maps discretized solution $(v_1 \ldots v_{N_x})$ to the stencil coefficients $(\alpha_1 \ldots \alpha_N)$: $\mathbb{R}^{N_x} \to \mathbb{R}^N$

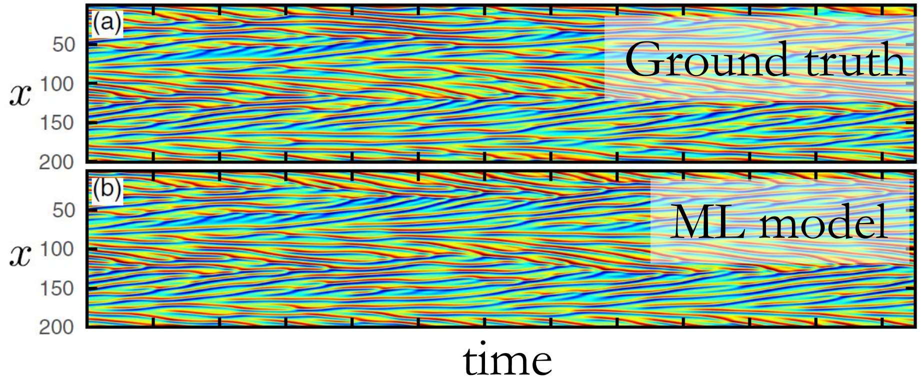
- **Learning a ML model to predict simple chaotic system (2018)**

PHYSICAL REVIEW LETTERS **120**, 024102 (2018)

Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach

Jaideep Pathak,[1,2,*] Brian Hunt,[3,4] Michelle Girvan,[1,3,2] Zhixin Lu,[1,3] and Edward Ott[1,2,5]

KS eqn



ML maps solution from one time step to the subsequent time step: $\mathbb{R}^{N_x} \to \mathbb{R}^{N_x}$


OpenAI Sora

ML weather forecast models

ECMWF is now running a series of data-driven forecasts as part of its experimental suite. These machine-learning based models are very fast, and they produce a 10-day forecast with 6-hourly time steps in approximately one minute. The outputs are available in graphical form.

Pangu-Weather

nature

Explore content ⌄ About the journal ⌄ Publish with us ⌄

nature › articles › article

Article | Open access | Published: 05 July 2023

Accurate medium-range global weather forecasting with 3D neural networks

Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu & Qi Tian

Nature **619**, 533–538 (2023) | Cite this article

Science

Current Issue First release papers Archive About

HOME › SCIENCE › VOL. 382, NO. 6677 › LEARNING SKILLFUL MEDIUM-RANGE GLOBAL WEATHER FORECASTING

RESEARCH ARTICLE | WEATHER FORECASTING

Learning skillful medium-range global weather forecasting

REMI LAM, ALVARO SANCHEZ-GONZALEZ, MATTHEW WILLSON, PETER WIRNSBERGER, MEIRE FORTUNATO, FERRAN ALET, SUMAN RAVURI, TIMO EWALDS, ZACH EATON-ROSEN, [...], AND PETER BATTAGLIA +8 authors Authors Info & Affiliations

SCIENCE • 14 Nov 2023 • Vol 382, Issue 6677 • pp. 1416-1421 • DOI: 10.1126/science.adi2336

Abstract

Global medium-range weather forecasting is critical to decision-making across many social and economic domains. Traditional numerical weather prediction uses increased compute resources to improve forecast accuracy but does not directly use historical weather data to improve the underlying model. Here, we introduce GraphCast, a machine learning–based method trained directly from reanalysis data. It predicts hundreds of weather variables for the next 10 days at 0.25° resolution globally in under 1 minute. GraphCast significantly outperforms the most accurate operational deterministic systems on 90% of 1380 verification targets, and its forecasts support better severe event prediction, including tropical cyclone tracking, atmospheric rivers, and extreme temperatures. GraphCast is a key advance in accurate and efficient weather forecasting and helps realize the promise of machine learning for modeling complex dynamical systems.

nature

Explore content ⌄ About the journal ⌄ Publish with us ⌄

nature › articles › article

Article | Open access | Published: 04 December 2024

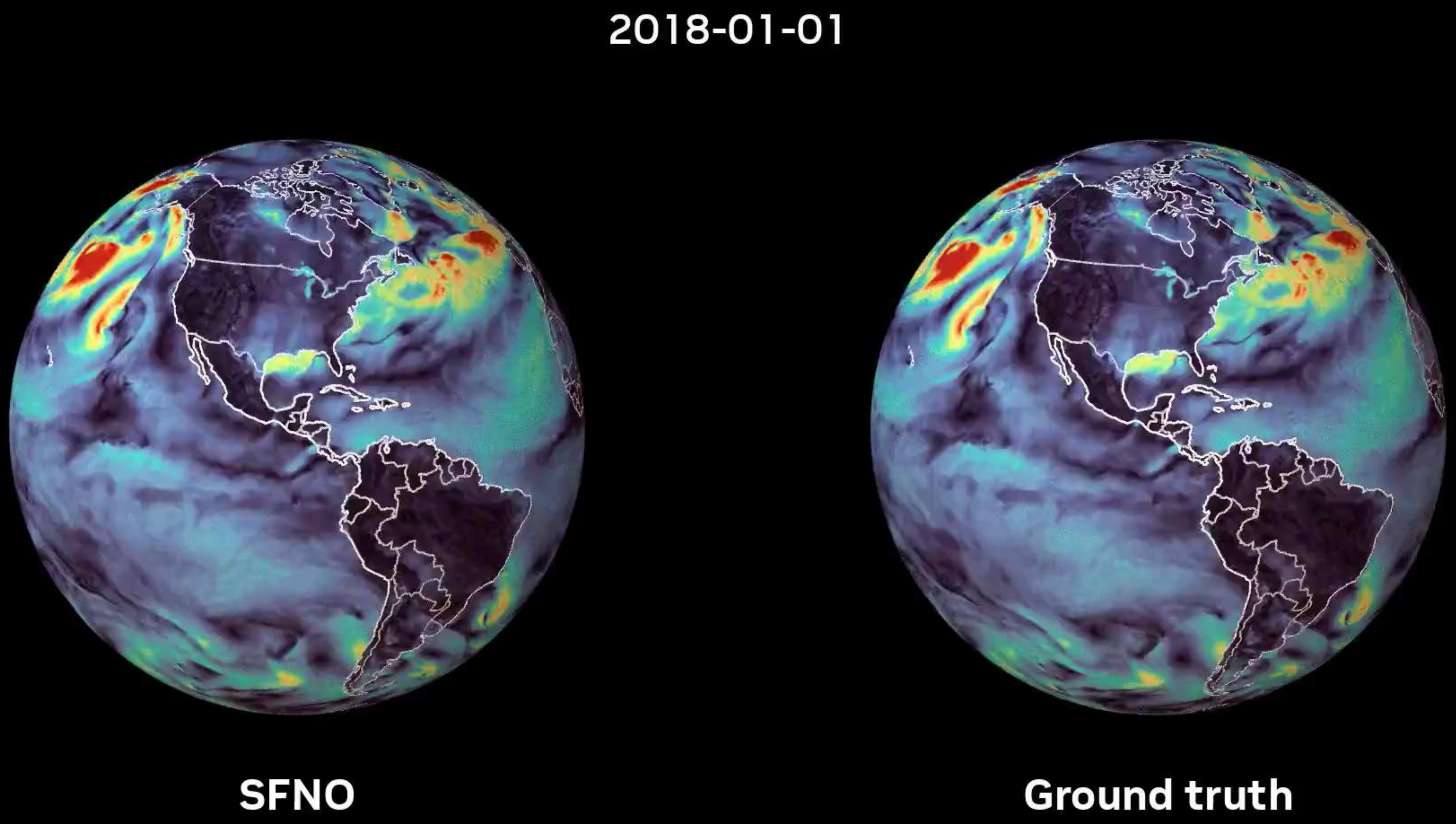
Probabilistic weather forecasting with machine learning

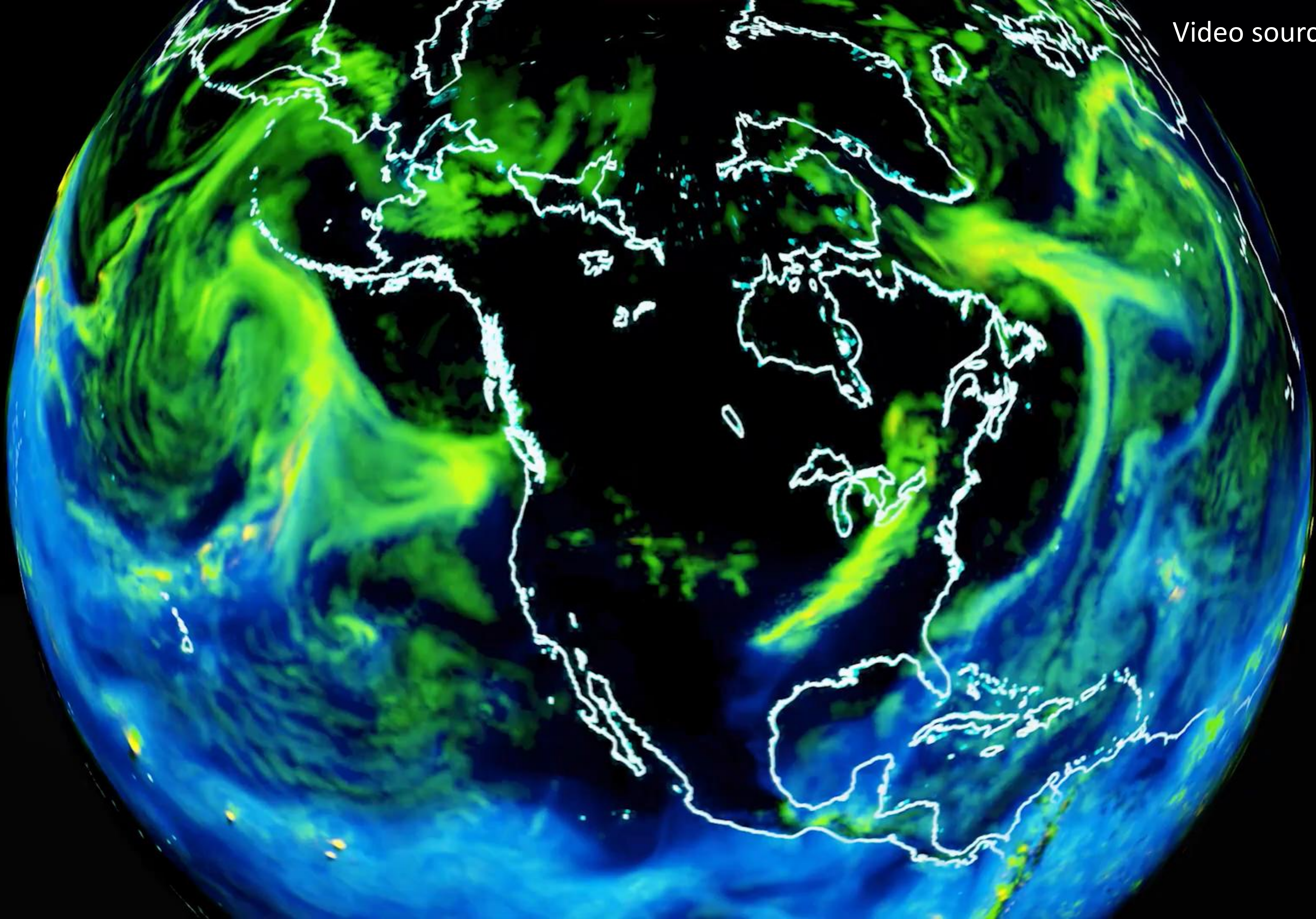
Ilan Price, Alvaro Sanchez-Gonzalez, Ferran Alet, Tom R. Andersson, Andrew El-Kadi, Dominic Masters, Timo Ewalds, Jacklynn Stott, Shakir Mohamed, Peter Battaglia, Remi Lam & Matthew Willson

However, these advances have focused primarily on single, deterministic forecasts that fail to represent uncertainty and estimate risk. Overall, MLWP has remained less accurate and reliable than state-of-the-art NWP ensemble forecasts. Here we introduce GenCast, a probabilistic weather model with greater skill and speed than the top operational medium-range weather forecast in the world, ENS, the ensemble forecast of the European Centre for Medium-Range Weather Forecasts[4]. GenCast is an ML weather prediction method, trained on

ML weather forecast models

Video source: NVIDIA





Video source: NVIDIA

In addition to generating known results faster, can we leverage neural networks to discovery new PDE solutions?

Self-similar blow-up solution for Boussinesq eqn

$$f_1 = \Omega + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla\Omega - \Phi$$

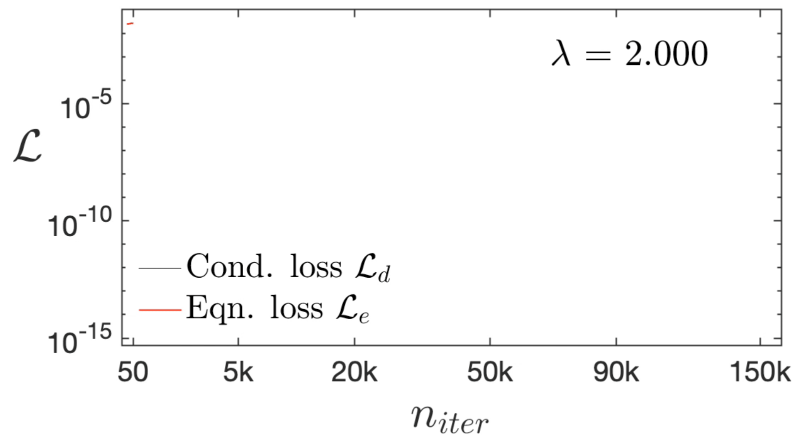
$$f_2 = (2 + \partial_{y_1} U_1)\Phi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla\Phi + \partial_{y_1} U_2 \Psi$$

$$f_3 = (2 + \partial_{y_2} U_2)\Psi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla\Psi + \partial_{y_2} U_1 \Phi$$

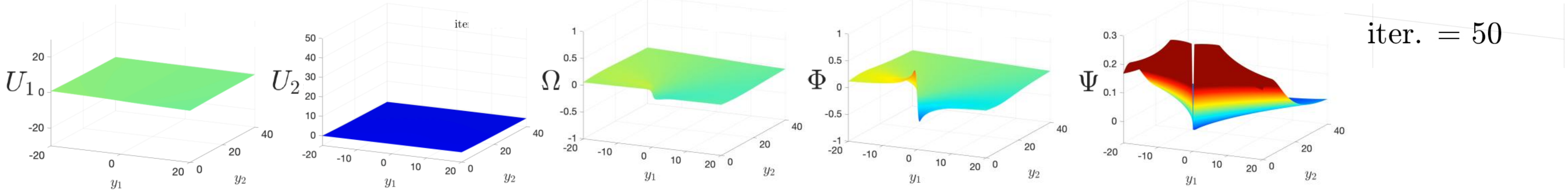
$$f_4 = \partial_{y_1} U_1 + \partial_{y_2} U_2$$

$$f_5 = \Omega - (\partial_{y_1} U_2 - \partial_{y_2} U_1)$$

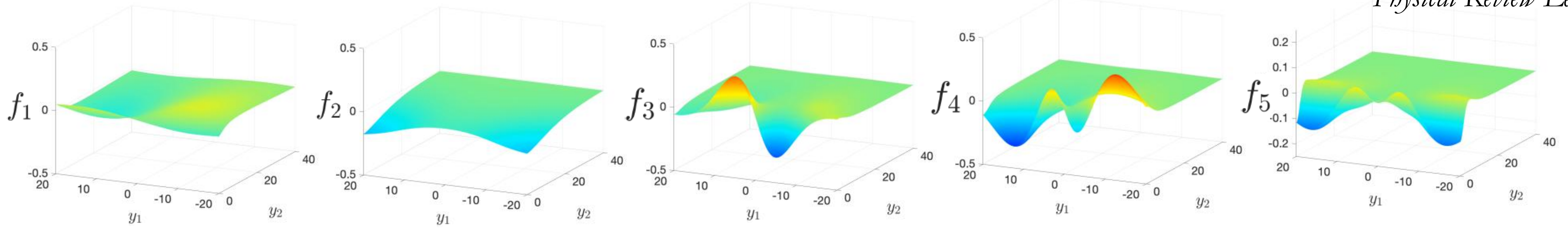
Loss function



Smooth self-similar solution



Uniform and small equation residues everywhere



Wang-Lai-GómezSerrano-Buckmaster, *Physical Review Letters* (2023)

So…why neural networks?

- Very often in math and science we want to know A as a function of B. NN has the flexibility to fit that function without prior assumptions about the functional form, particularly useful when the function is high dimensional (it is a function approximator).

- Leverages GPU (hardware) and open-source software (e.g. tensoflow, Pytorch, JAX). The optimization method is well-developed and user friendly. It can usually be easily adopted for different problems.

- We will talk about the challenges after learning about the basics of NN!

Outline

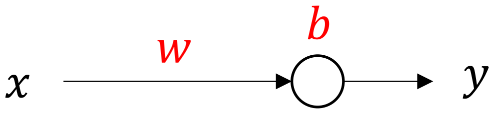
- Neural networks in problems governed by PDEs
- NN basics (Lecture 1-3)
  Universal function approximation
  Gradient descent
  Back propagation
- Physics-informed NN (Lecture 4-5)

What is a neural network?

An analytical model of output y as a function of input x, containing some fitting parameters

1. Linear Regression Model:

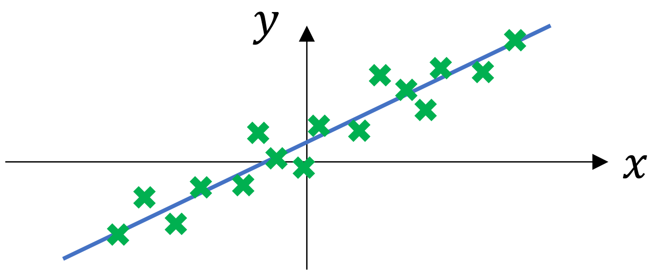
$\boxed{y = wx + b}$



Given observations of $\{x_d^i, y_d^i\}_i^n$

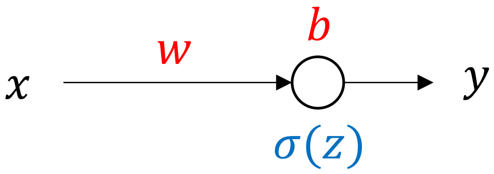
Find the w and b that minimizes

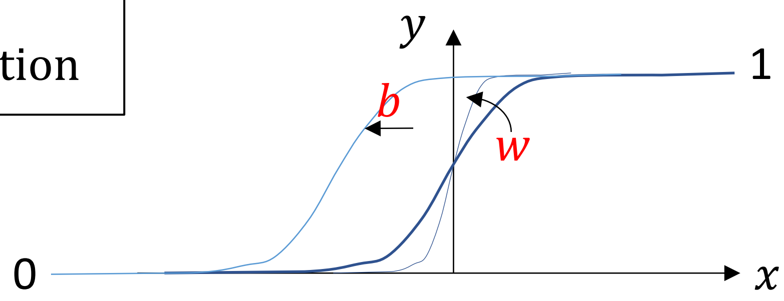
$J = \sum_{i=1}^{n} \left(y(x_d^i) - y_d^i \right)^2$



2. Logistic Regression Model: make output 0 to 1

$$\boxed{\begin{gathered} y = \sigma(wx + b), \\ \text{where } \sigma(z) = \frac{1}{1 + e^{-z}} \text{ is a sigmoid function} \end{gathered}}$$



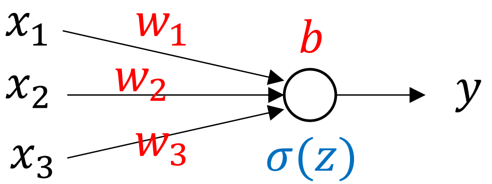


What is a neural network?

An analytical model of output y as a function of input x, containing some fitting parameters

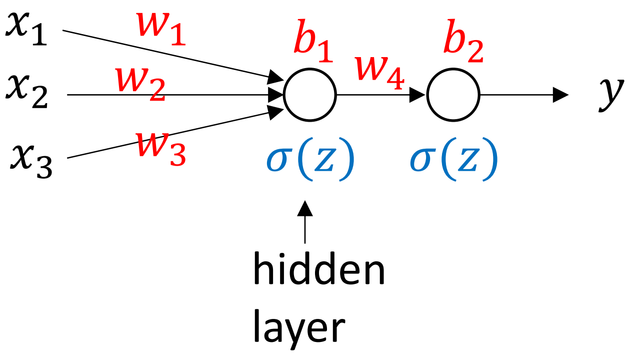
2. Logistic Regression Model: make output -1 to 1

$$y = \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3 + b),$$
$$\text{where } \sigma(z) = \frac{1}{1+e^{-z}} \text{ is a sigmoid function}$$



3. Neural network:

$$y = \sigma(w_4 \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3 + b_1) + b_2),$$
$$\text{where } \sigma(z) \text{ is a nonlinear activation function}$$



Common choices of $\boldsymbol{\sigma(z)}$

$sigmoid(z)$

$\left.\begin{array}{l} \sin(z) \\ \cos(z) \\ \tanh(z) \end{array}\right\}$ Output ranges from -1 to 1

…

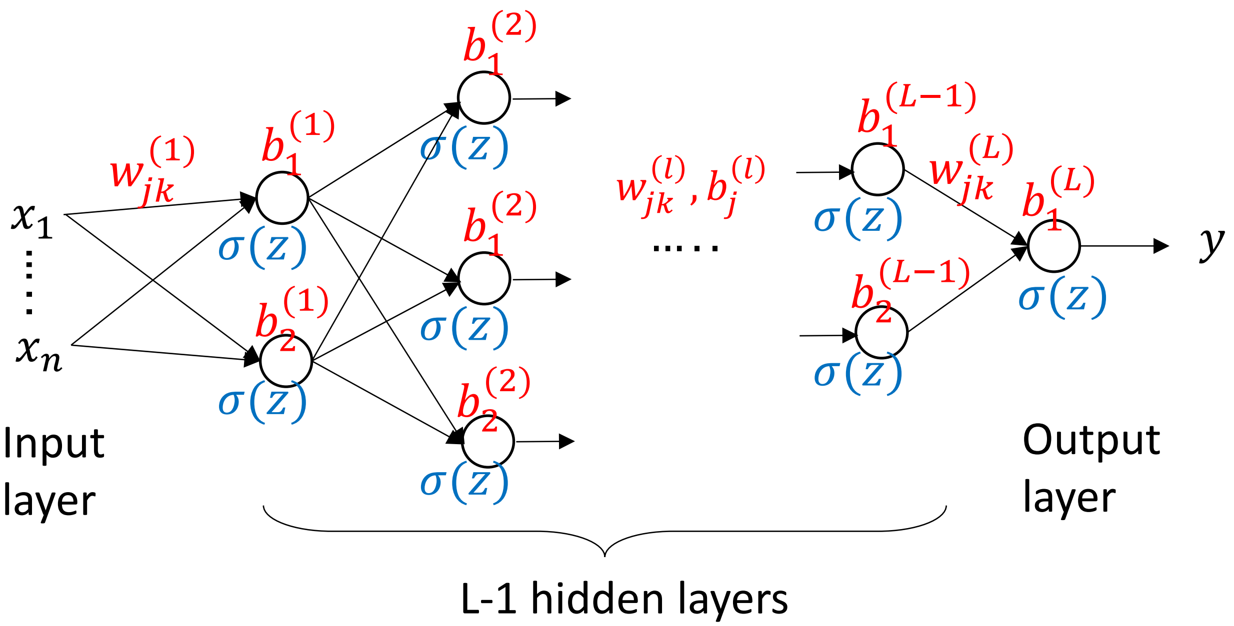
What is a neural network?

An analytical model of output y as a function of input x, containing some fitting parameters

3. Neural network:

More generally…

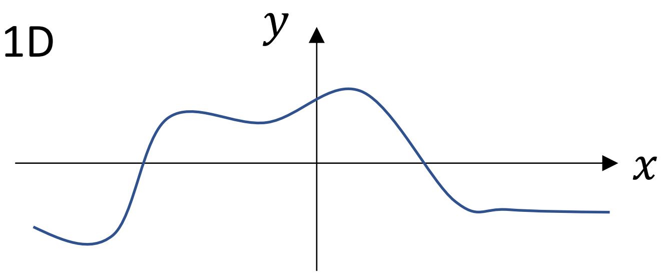
$$\boxed{\begin{array}{c} y = fn(w_{jk}^{(l)}, b_j^{(l)}, x_i), \\ \text{where } \sigma(z) \text{ is a nonlinear activation function} \end{array}}$$



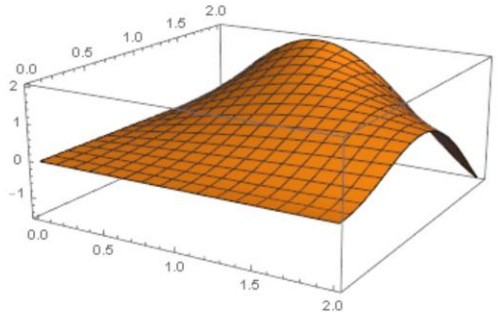
Neural network is a general function approximation

-> Universal function approximation

1D



2D

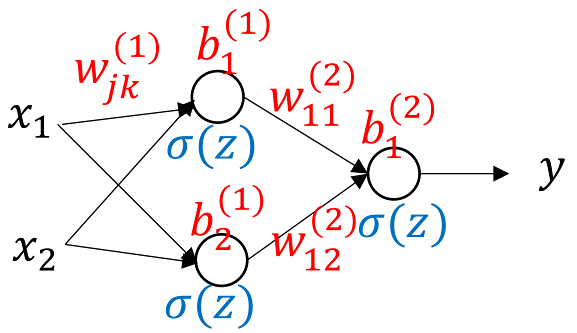


n-Dimension surface….

Why are activation function nonlinear?

Common choices of $\boldsymbol{\sigma(z)}$

$sigmoid(z)$
$\sin(z)$
$\cos(z)$
$\tanh(z)$
…



- If activation function is linear, NN can only represent a linear function

If activation fn is linear, e.g. $\sigma(z) = z$ …

Input x_1, x_2

In hidden layer, j=1,2

$$z_j^{(1)} = \sum_{k=1}^{2} w_{jk}^{(1)} x_k + b_j^{(1)}$$

$$a_j^{(1)} = \sigma\left(z_j^{(1)}\right) = z_j^{(1)}$$

In output layer, j=1

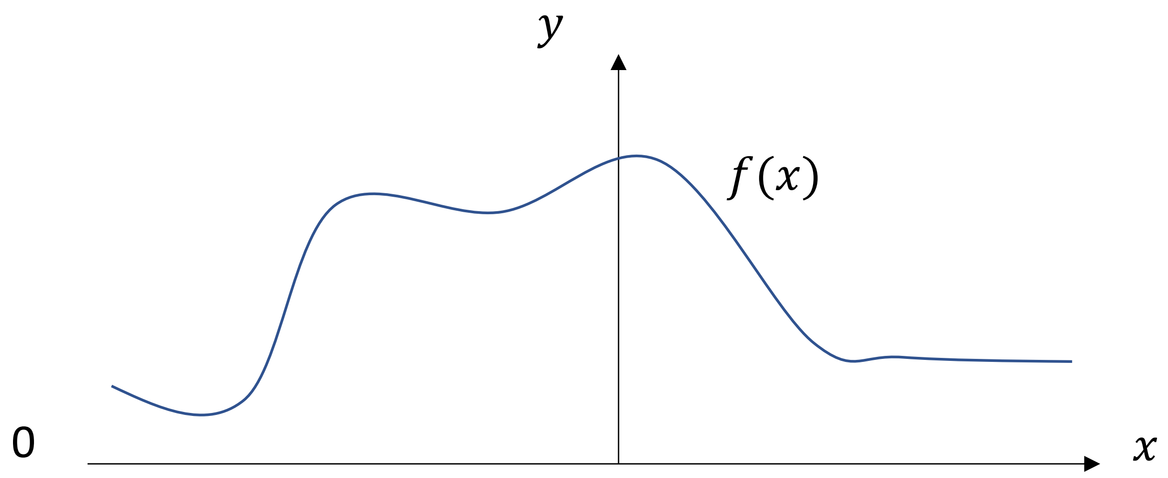
$$z_j^{(2)} = \sum_{k=1}^{2} w_{jk}^{(2)} a_k^{(1)} + b_j^{(2)}$$

$$a_j^{(2)} = \sigma\left(z_j^{(2)}\right) = z_j^{(2)}$$

$$y = a_1^{(2)} = \sum_{k=1}^{2} w_{1k}^{(2)} \left(\sum_{l=1}^{2} w_{kl}^{(1)} x_l + b_k^{(1)} \right) + b_1^{(2)} = Ax_1 + Bx_2 + C$$

NN is a Universal Function Approximator

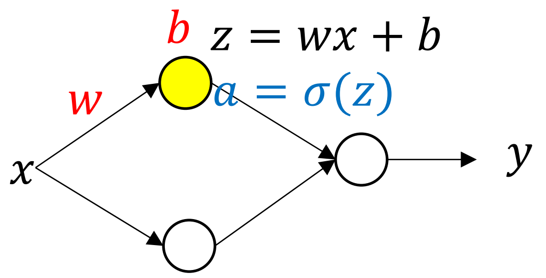
- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)

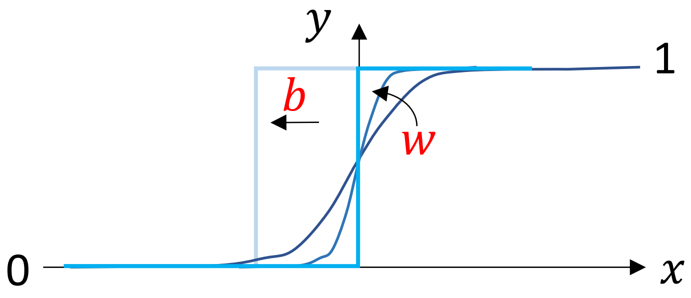


Goal:
Use a NN to approximate $f(x)$

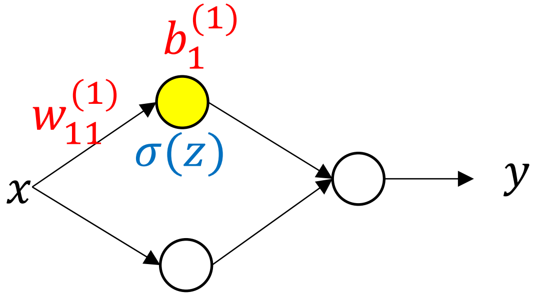
NN is a Universal Function Approximator

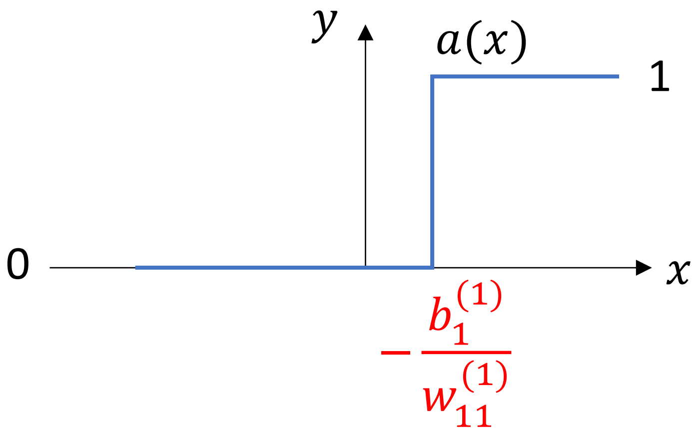
- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)





Large w gives a step

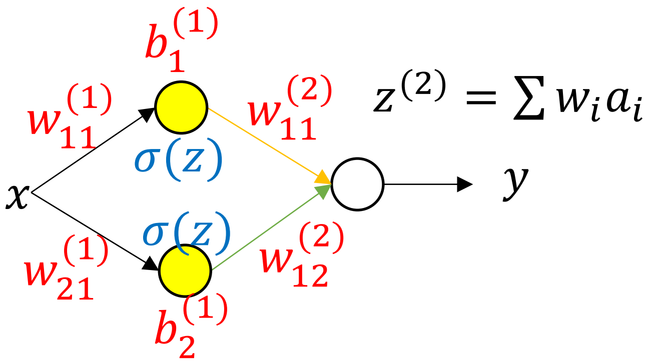


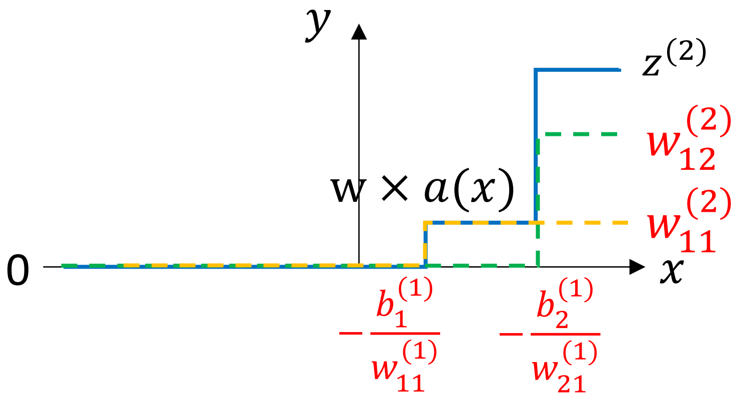


$-\frac{b}{w}$ determines the location of the step

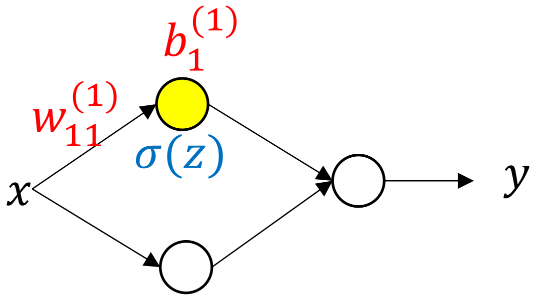
NN is a Universal Function Approximator

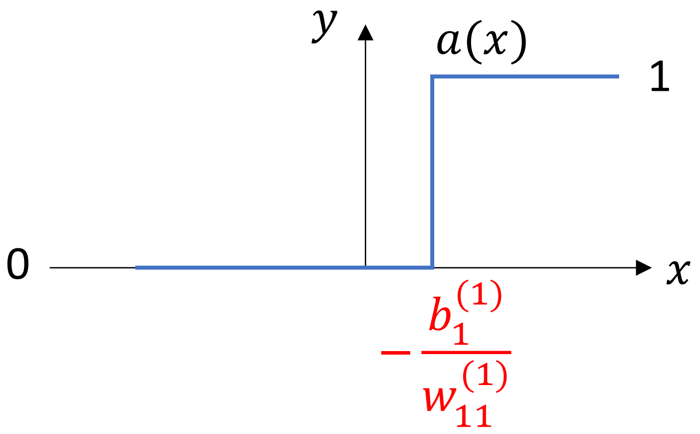
- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)





Superposition of two steps

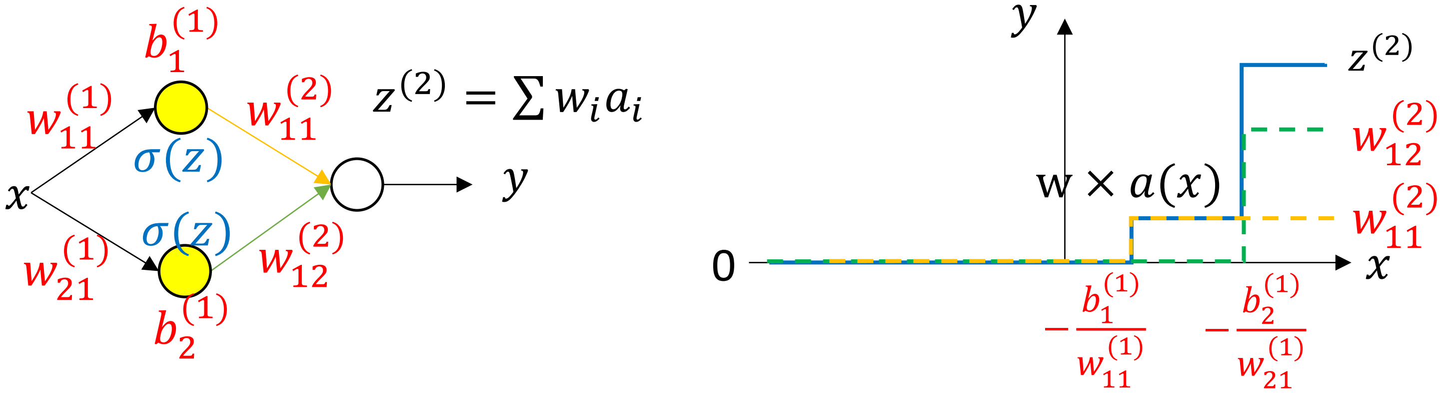




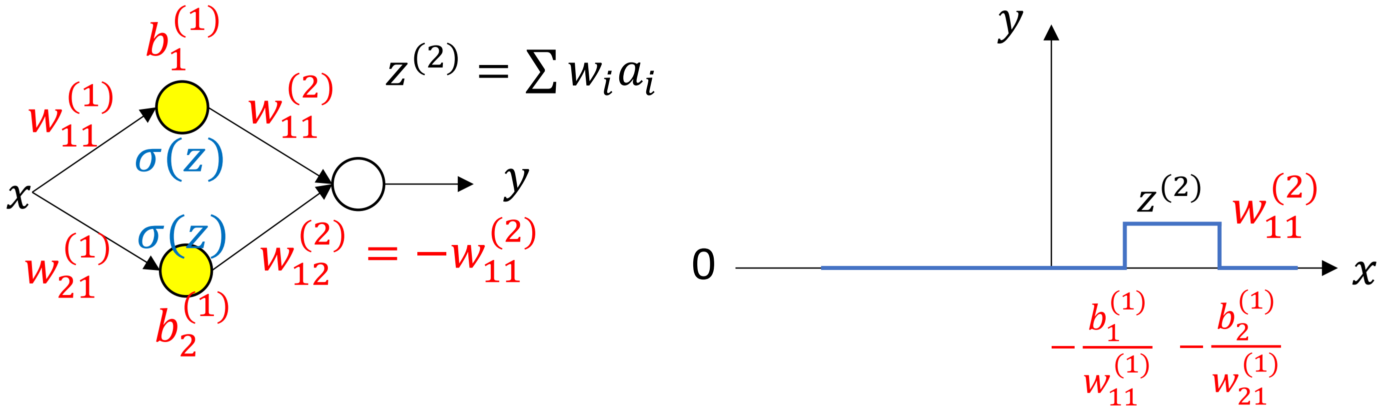
$-\frac{b}{w}$ determines the location of the step

NN is a Universal Function Approximator

- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)



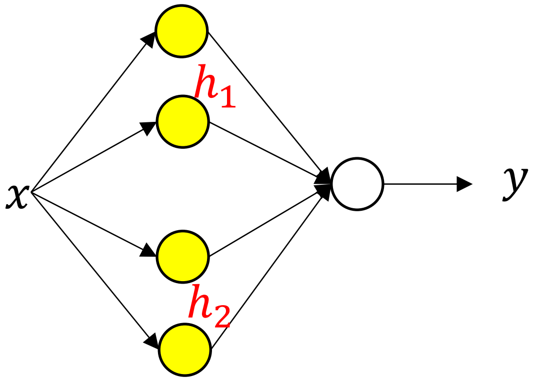
Superposition of two steps

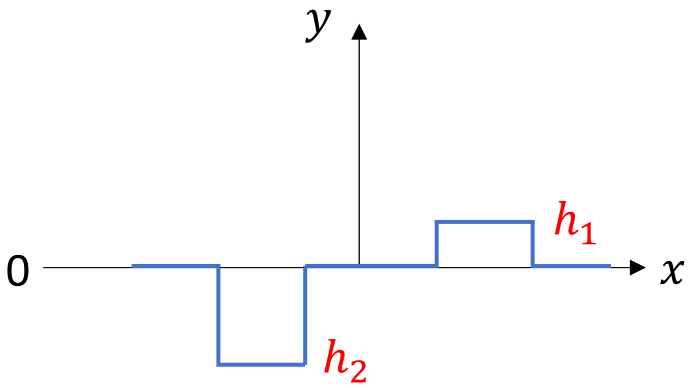


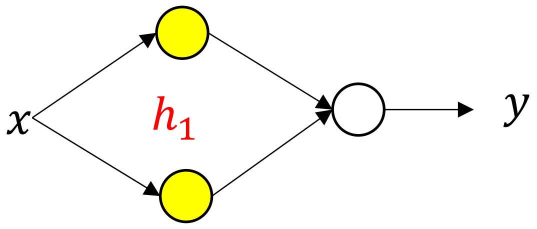
Create a column of height h

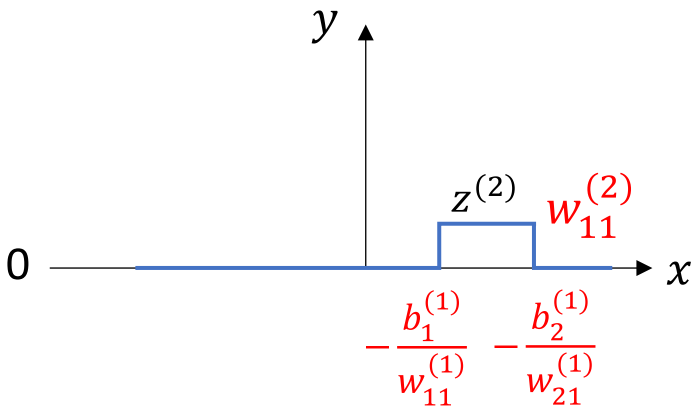
NN is a Universal Function Approximator

- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)





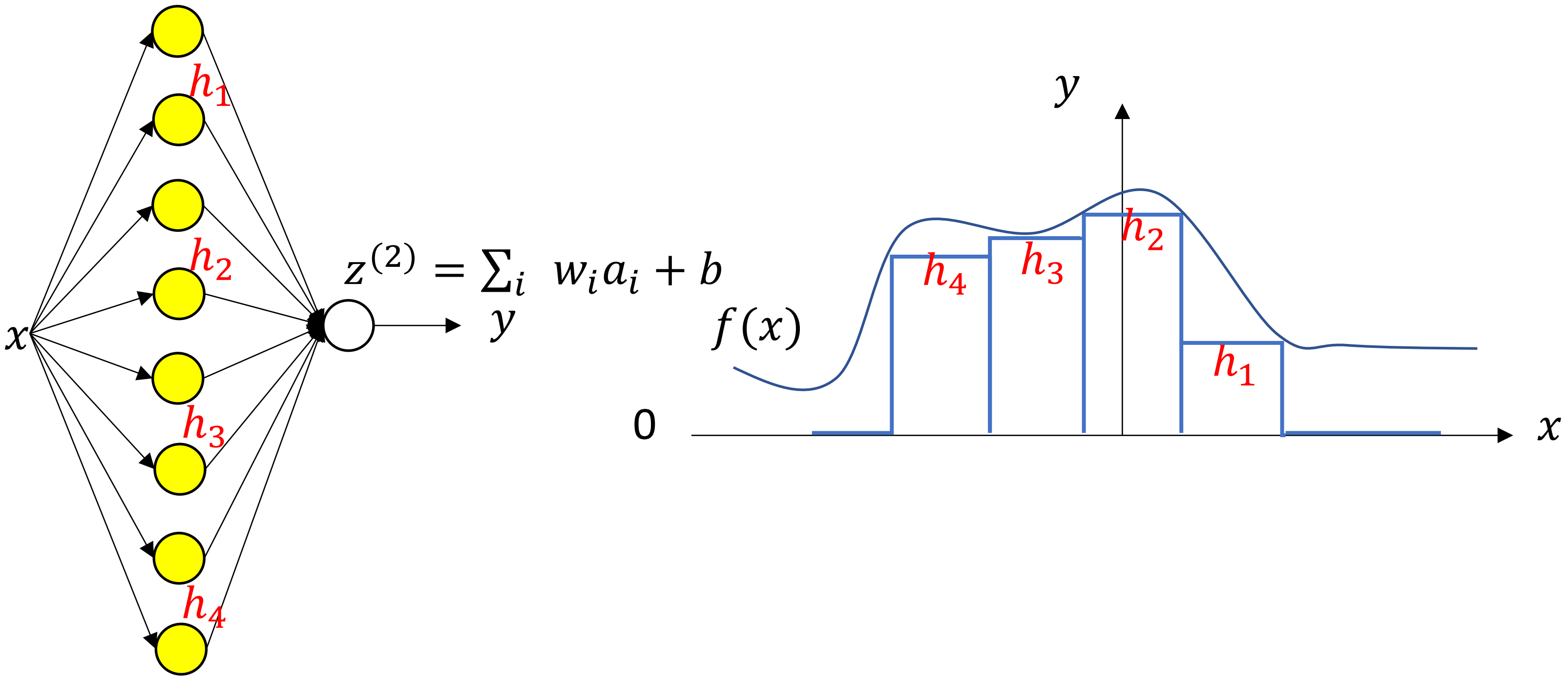




Create a column of height h

NN is a Universal Function Approximator

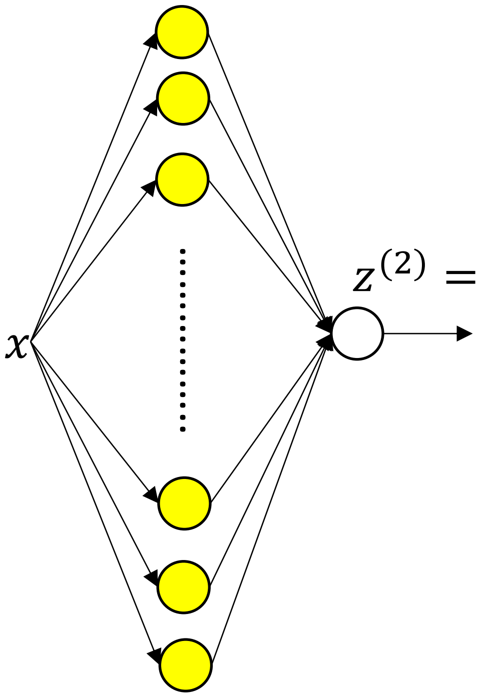
- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)



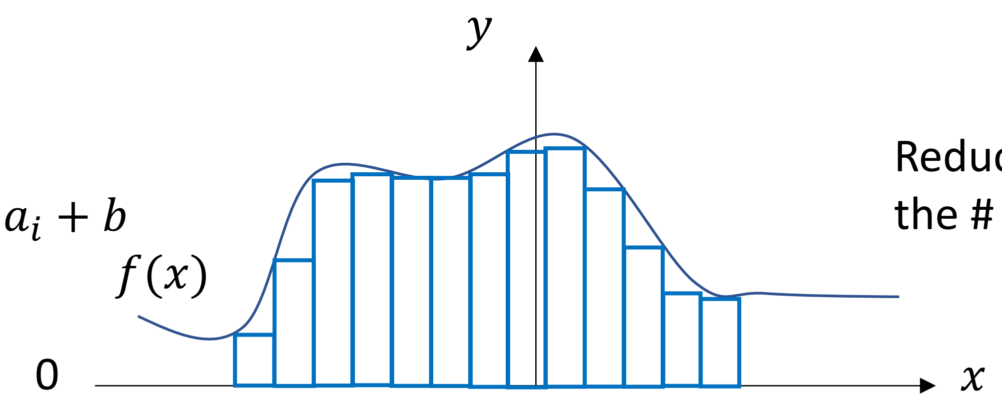
Superposition of columns

NN is a Universal Function Approximator

- NN can approximate continuous and smooth functions
  A visual proof (for sigmoid activation)



$z^{(2)} = \sum_i w_i a_i + b$



Reduce column widths + increasing the # of neurons to approximate $f(x)$

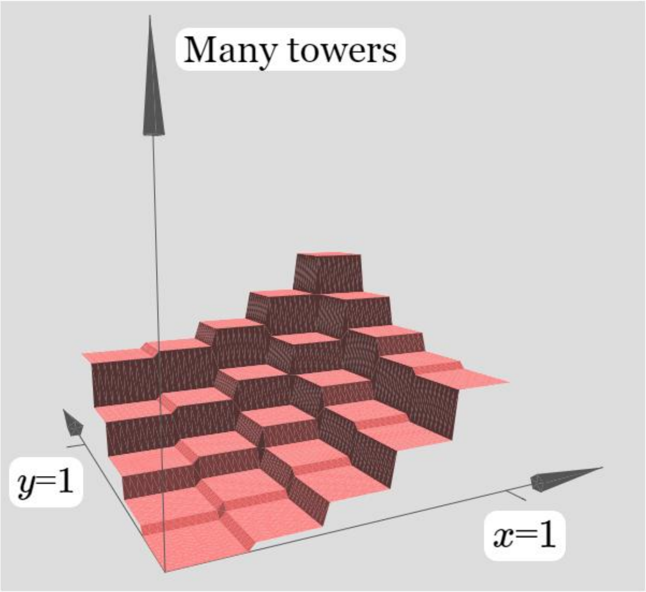
For a given continuous smooth $f(x)$ and an arbitrarily small $\epsilon > 0$
There exist $z^{(2)}$ so that

$$\int |z^{(2)}(x) - f(x)|\, dx < \epsilon$$

NN is a Universal Function Approximator

- Approximate a 2D surface

What should be the input/output units of NN?



http://neuralnetworksanddeeplearning.com/chap4.html

NN is a Universal Function Approximator

In summary

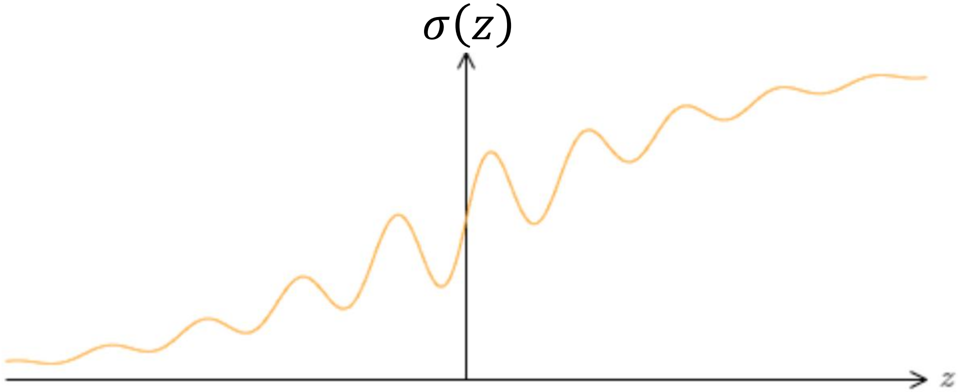
- Dimension of the approximated surface is determined by …

  Number of input and output units

- Column size is determined by …

  Number of hidden units, values of weights and biases

- Could the following activation function instead of a sigmoid function approximate a step?



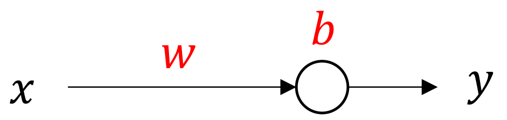
$\sigma(z) \to s_1,\ z \to \infty$
$\sigma(z) \to s_2,\ z \to -\infty$
$s_1 \neq s_2$

- Could activation function $\sigma(z) = z$ instead of a sigmoid function approximate a step?

Now we know it is possible to tune weights and biases in a NN to approximate functions. We still need an automated method to find the correct weights and biases.

How to find w and b?

- E.g., Model $y = \sigma(\color{red}{w}x + \color{red}{b})$

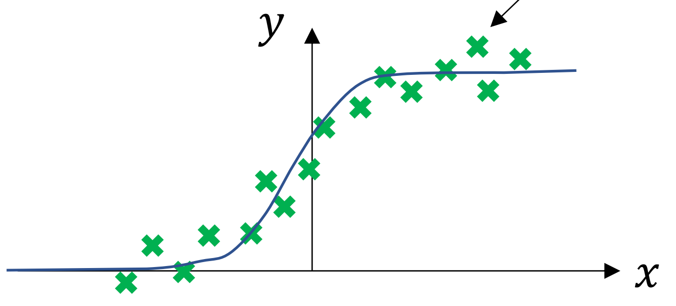


Given observations of $\{x_d^i, y_d^i\}_i^m$

Our goal is to find the model parameters w, b that minimizes the cost function

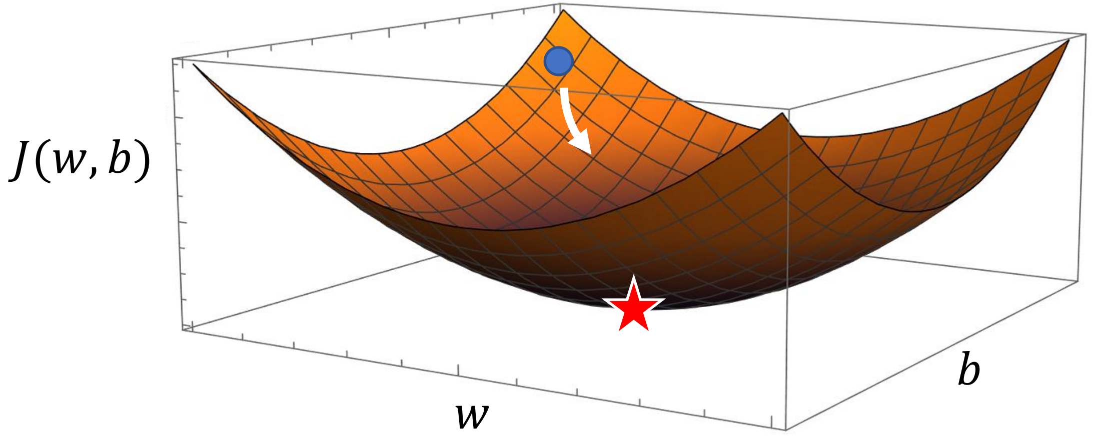
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2$$

Training data



How to find w and b?

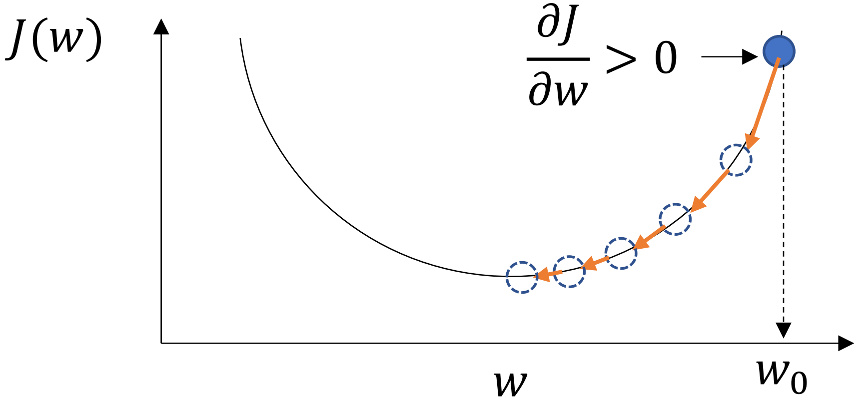
- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w, b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2$



Find the w, b that minimizes $J(w, b)$
i.e. $\frac{\partial J}{\partial w} = \frac{\partial J}{\partial b} = 0$

How to find w and b? Gradient descent

- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \left(y\left(x_d^i\right) - y_d^i\right)^2$



I. calculate the slope $\frac{\partial J}{\partial w}$ corresponds to an initial w

II. Adjust w according to the local slope

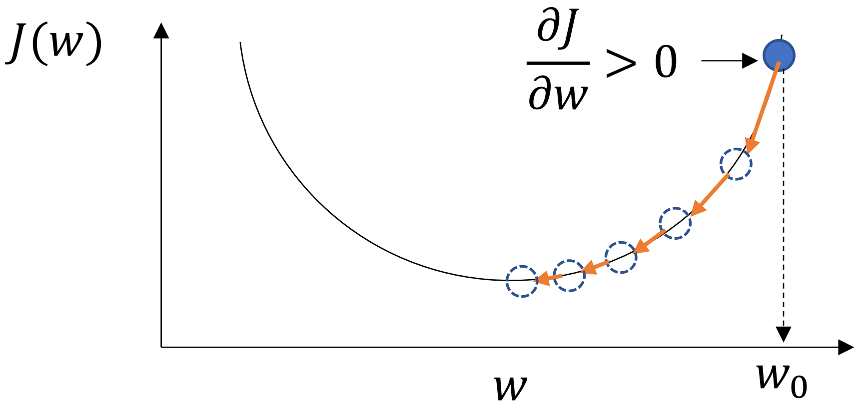
$w_{new} = w_{old} - \alpha \frac{\partial J}{\partial w}$, $\alpha > 0$ is the learning rate

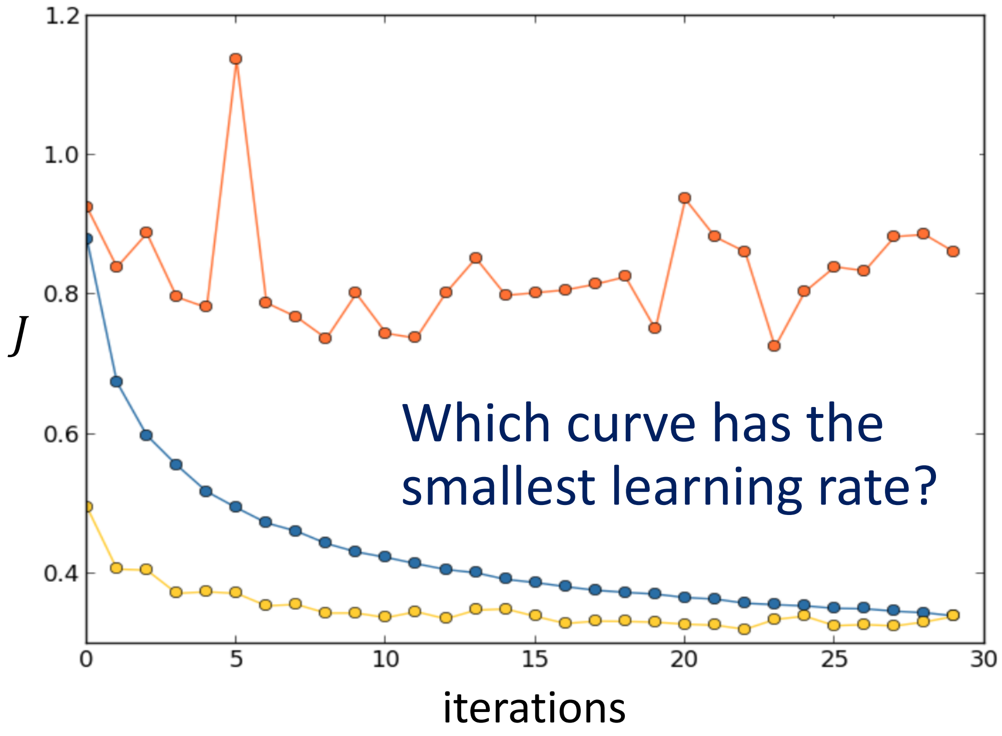
III. Iterate until $\frac{\partial J}{\partial w} = 0$

https://developers.google.com/machine-learning/crash-course/fitter/graph

How to find w and b? Gradient descent

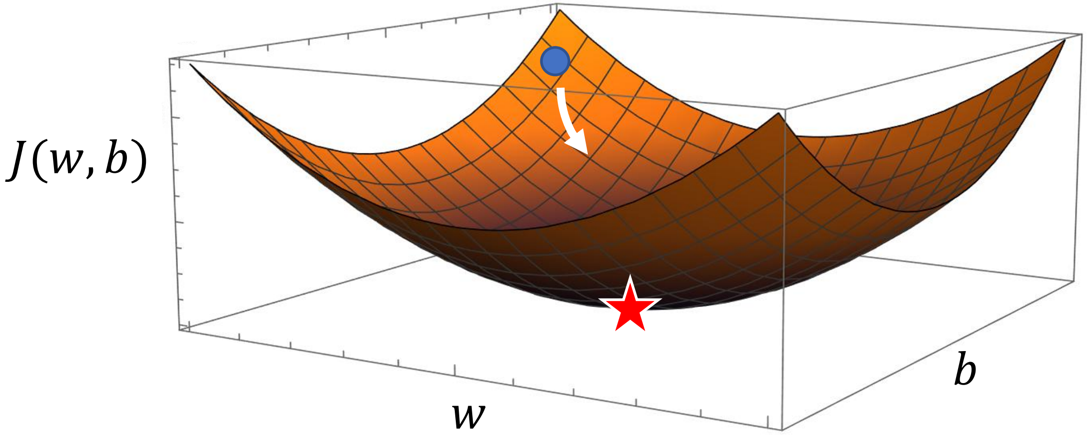
- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w, b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2$





How to find w and b? Gradient descent

- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2$



Gradient on a surface

- $-\nabla J$ gives the direction of the steepest decrease of J

$$-\nabla J(w,b) = -\left(\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}\right)$$

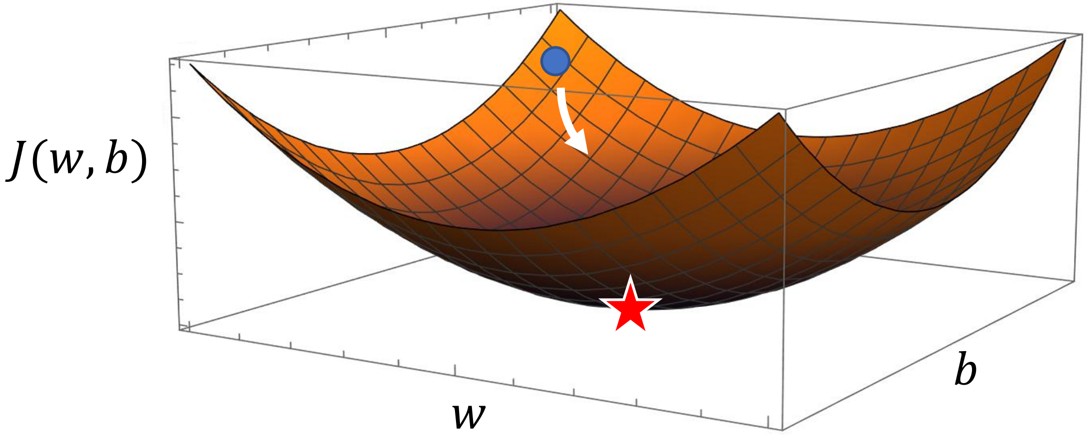
e.g. $-\nabla J(w_0, b_0) = -(10,1)$ What does this mean?

Changing w reduces J 10 times faster than changing b

- $-\nabla J$ tells you which weights and biases reduce cost function J the fastest!

How to find w and b? Gradient descent

- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w, b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2$



Gradient on a surface

- $-\nabla J$ gives the direction of the steepest decrease of J

$$-\nabla J(w, b) = -\left(\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}\right)$$

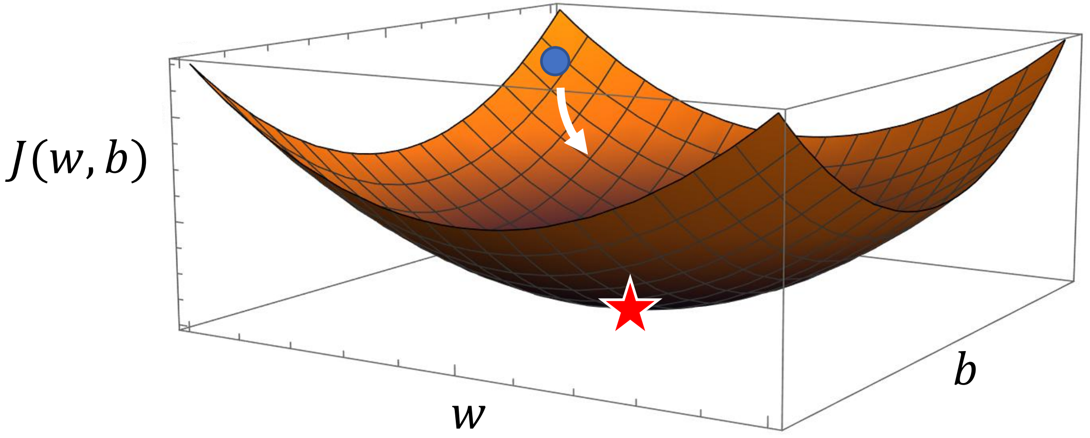
- $(w_{new}, b_{new}) = (w_{old}, b_{old}) - \alpha\, \nabla J(w, b)$, α is the learning rate

- Iterate until $\nabla J = 0$

https://www.deeplearning.ai/ai-notes/optimization/

How to find w and b? Gradient descent

- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2$



Gradient on a surface

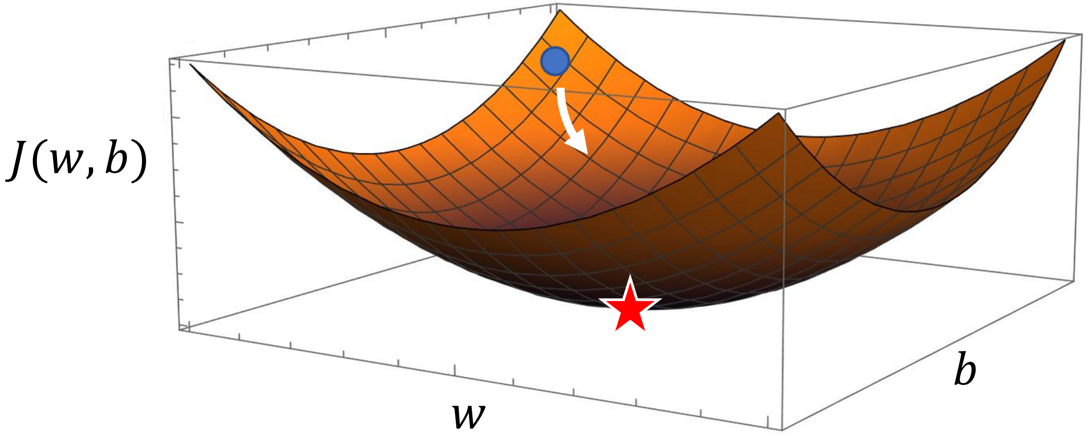
- $-\nabla J$ gives the direction of the steepest decrease of J

$$-\nabla J(w,b) = -\left(\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}\right)$$

But how are $\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}$ calculated?

How to find w and b? Gradient descent

- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w, b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2 = \frac{1}{m}\sum_{i=1}^{m} \overbrace{L\left(y^i, y_d^i\right)}^{\text{Defined for every example i}}$



What would $\frac{dL}{dw}$ be for a linear model $\sigma(z) = z$?

$$\boxed{L \equiv (y(x) - y_d)^2, \quad J = \frac{1}{m}\sum_{i=1}^{m} L \;\rightarrow\; \frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w}}$$

Back propagation (chain rule)

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial (wx+b)}\frac{\partial (wx+b)}{\partial w} = 2(y - y_d)\sigma' x$$

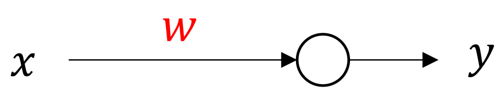
1) For one example:

$$\frac{\partial L}{\partial w}\left(\mathrm{w}, \mathrm{b}, \mathrm{x}_\mathrm{d}^i, \mathrm{y}_\mathrm{d}^i\right) = 2\left(\sigma\left(w\mathrm{x}_\mathrm{d}^i + b\right) - \mathrm{y}_\mathrm{d}^i\right)\sigma'\, \mathrm{x}_\mathrm{d}^i$$

2) For the full data set:

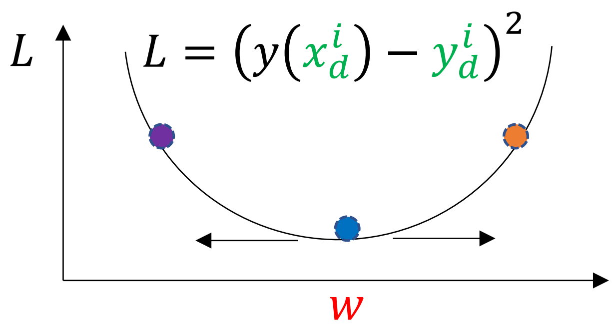
$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} 2\left(\sigma\left(w\mathrm{x}_\mathrm{d}^i + b\right) - \mathrm{y}_\mathrm{d}^i\right)\sigma'\mathrm{x}_\mathrm{d}^i$$

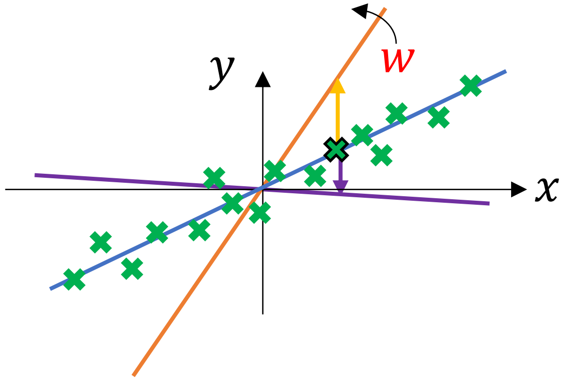
Example:



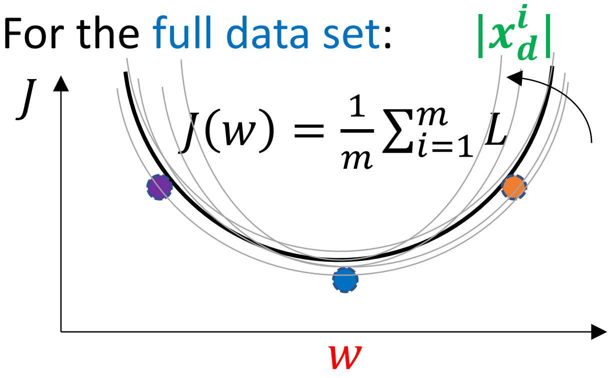
- Linear Regression Model $y = wx$
- Cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2 = \frac{1}{m}\sum_{i=1}^{m} \overbrace{L\left(y^i, y_d^i\right)}^{\text{Defined for every example i}}$

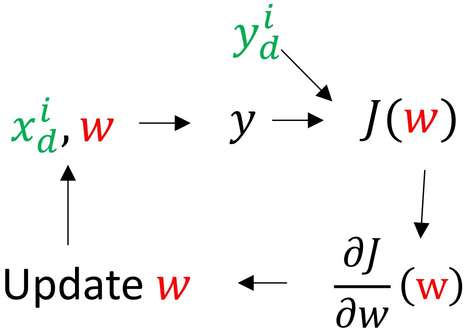
1) For one example:





2) For the full data set:





$$\boxed{L \equiv (y(x) - y_d)^2,\ J = \frac{1}{m}\sum_{i=1}^{m} L \rightarrow \frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L}{\partial w}}$$

Back propagation (chain rule)

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial (wx)}\frac{\partial (wx)}{\partial w} = 2(y - y_d)x$$

1) For one example:

$$\frac{\partial L}{\partial w}\left(\mathrm{w}, \mathrm{b}, \mathrm{x}_{\mathrm{d}}^{i}, \mathrm{y}_{\mathrm{d}}^{i}\right) = 2\left(wx_d^i - \mathrm{y}_{\mathrm{d}}^{i}\right)\mathrm{x}_{\mathrm{d}}^{i}$$

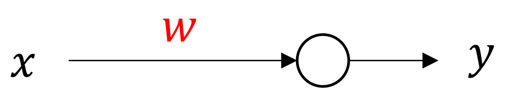
This gradient is defined for every example i, and is a function of w

2) For the full data set:

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} 2\left(wx_d^i - \mathrm{y}_{\mathrm{d}}^{i}\right)\mathrm{x}_{\mathrm{d}}^{i}$$

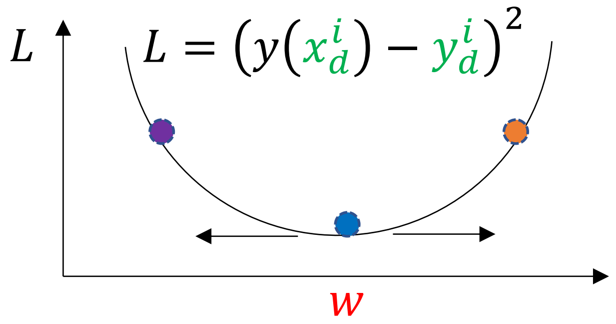
https://www.deeplearning.ai/ai-notes/optimization/

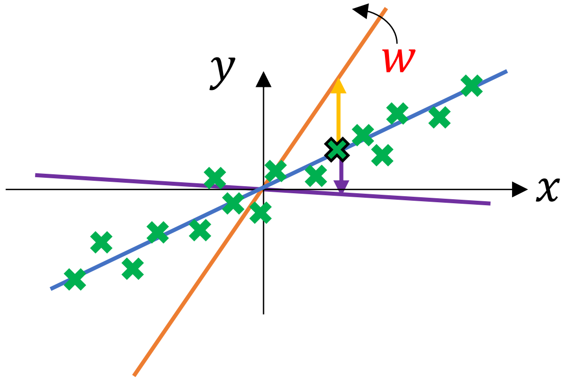
Example:



- Linear Regression Model $y = wx$
- Cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2 = \frac{1}{m}\sum_{i=1}^{m} \overbrace{L\left(y^i, y_d^i\right)}^{\text{Defined for every example i}}$

1) For one example:





$$\boxed{L \equiv (y(x) - y_d)^2,\ J = \frac{1}{m}\sum_{i=1}^{m} L \rightarrow \frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w}}$$

Back propagation (chain rule)

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial (wx)}\frac{\partial (wx)}{\partial w} = 2(y - y_d)x$$

1) For one example:

$$\frac{\partial L}{\partial w}\left(\mathrm{w}, \mathrm{b}, \mathrm{x}_\mathrm{d}^i, \mathrm{y}_\mathrm{d}^i\right) = 2\left(wx_d^i - \mathrm{y}_\mathrm{d}^i\right)\mathrm{x}_\mathrm{d}^i$$

This gradient is defined for every example i, and is a function of w

2) For the full data set:

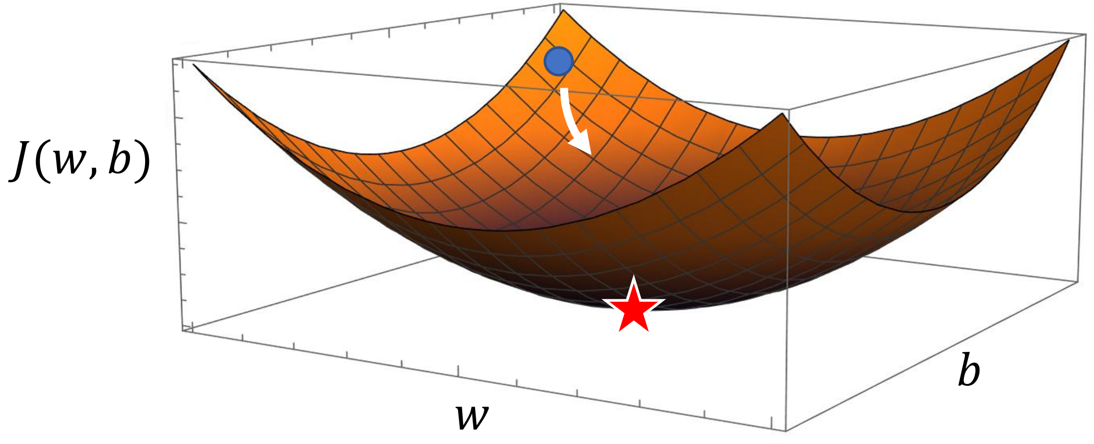
$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} 2\left(wx_d^i - \mathrm{y}_\mathrm{d}^i\right)\mathrm{x}_\mathrm{d}^i$$

What would happen if x_d^i are very large/small?

Note that $\frac{\partial J}{\partial w}$ is analytical fn of $\mathrm{w}, x_d^i, \mathrm{y}_\mathrm{d}^i$!

How to find w and b? Gradient descent

- E.g., Model $y = \sigma(wx + b)$
- Cost function $J(w, b) = \frac{1}{m}\sum_{i=1}^{m}\left(y\left(x_d^i\right) - y_d^i\right)^2 = \frac{1}{m}\sum_{i=1}^{m} L(y^i, y_d^i)$



- $-\boldsymbol{\nabla} J(w, b) = -\left(\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}\right)$ **for a given data set at a given $\boldsymbol{w}, \boldsymbol{b}$ is known analytically**
- $(w_{new}, b_{new}) = (w_{old}, b_{old}) - \alpha\, \nabla J(w, b)$, α is the learning rate
- Iterate until $\nabla J = 0$

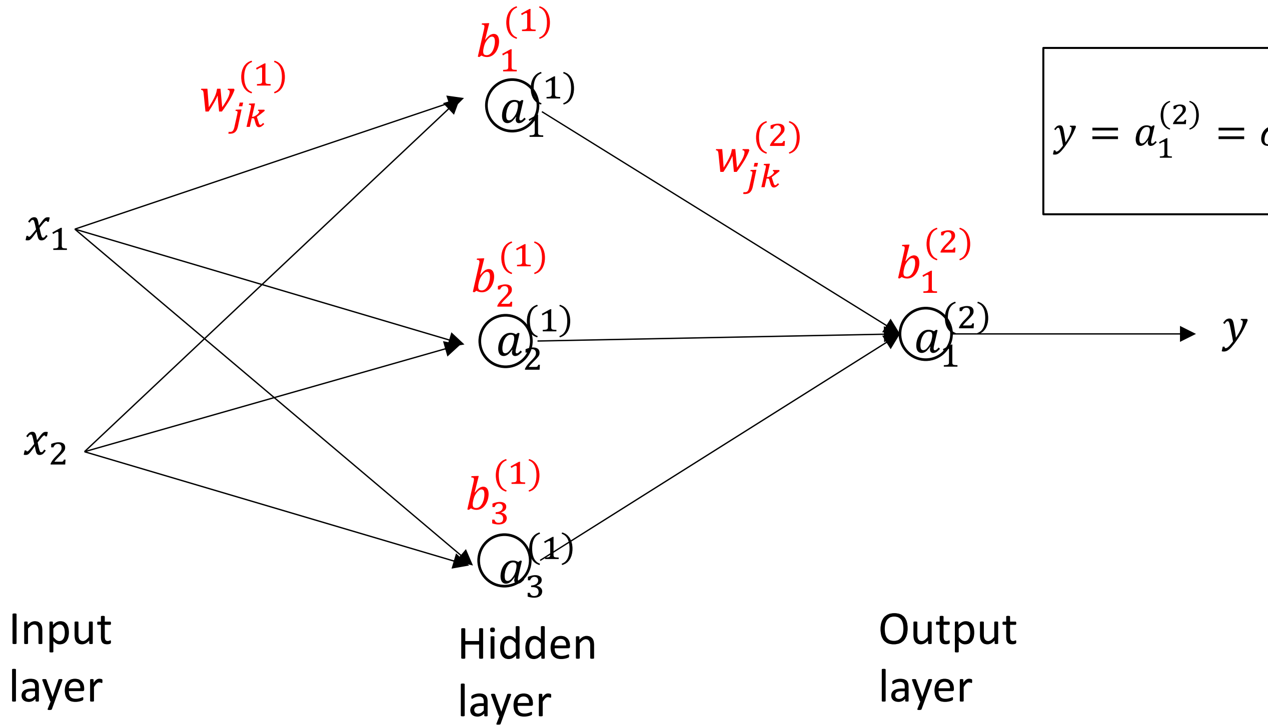
So far we have discussed…

- Universal function approximator
- Gradient descent: a method to find weights and biases that minimize J
- Calculate $-\boldsymbol{\nabla} J(w, b) = -\left(\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}\right)$ for a simple model $y = \sigma(wx + b)$
  - One example
  - A full dataset

We know how to find w and b for a simple model $y = \sigma(wx + b)$.

What about finding w and b in a neural network?

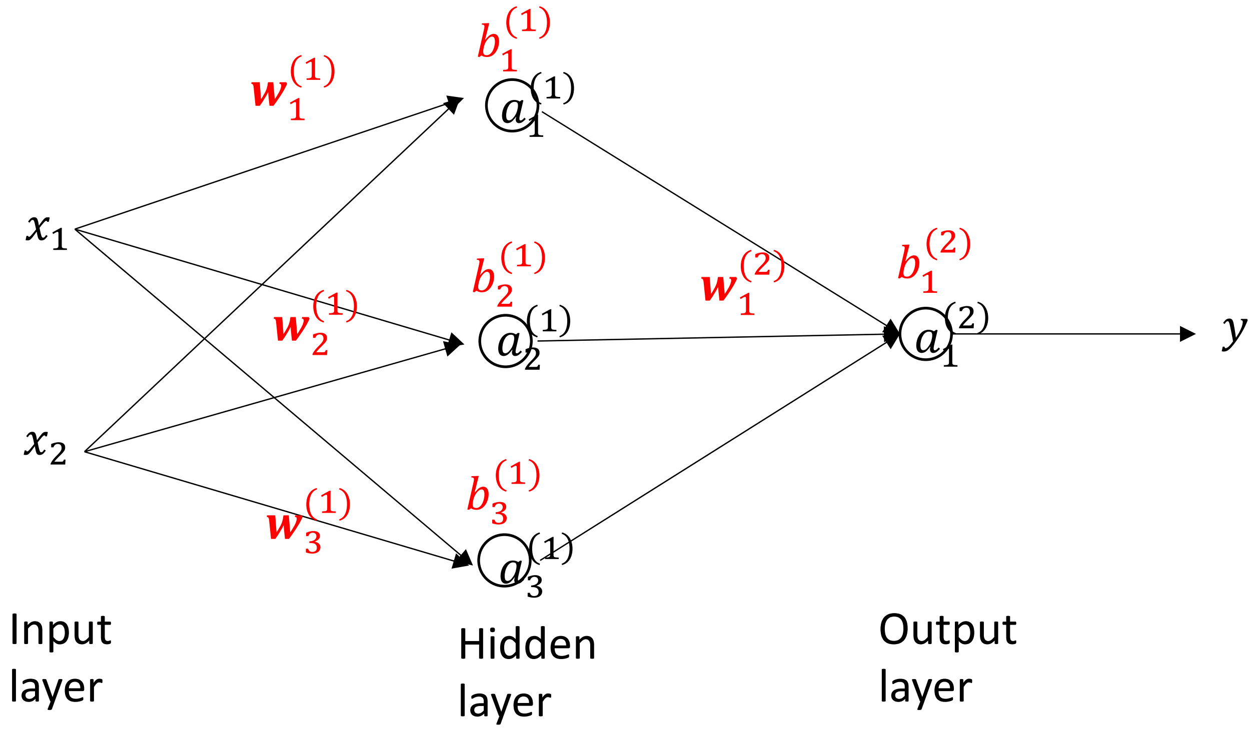
Writing a neural network in terms of vectors



$$y = a_1^{(2)} = \sigma\left(\sum_{k=1}^{3} w_{1k}^{(2)} \sigma\left(\sum_{l=1}^{2} w_{kl}^{(1)} x_l + b_k^{(1)}\right) + b_1^{(2)}\right)$$

# of units	*j=1,2*	*j=1,2,3*	*j=1*
Forward propagation	$a_j^{(0)} = x_j$	$z_j^{(1)} = \sum_{k=1}^{2} w_{jk}^{(1)} a_k^{(0)} + b_j^{(1)}$ $a_j^{(1)} = \sigma\left(z_j^{(1)}\right)$	$z_j^{(2)} = \sum_{k=1}^{3} w_{jk}^{(2)} a_k^{(1)} + b_j^{(2)}$ $a_j^{(2)} = \sigma\left(z_j^{(2)}\right)$

Writing a neural network in terms of vectors

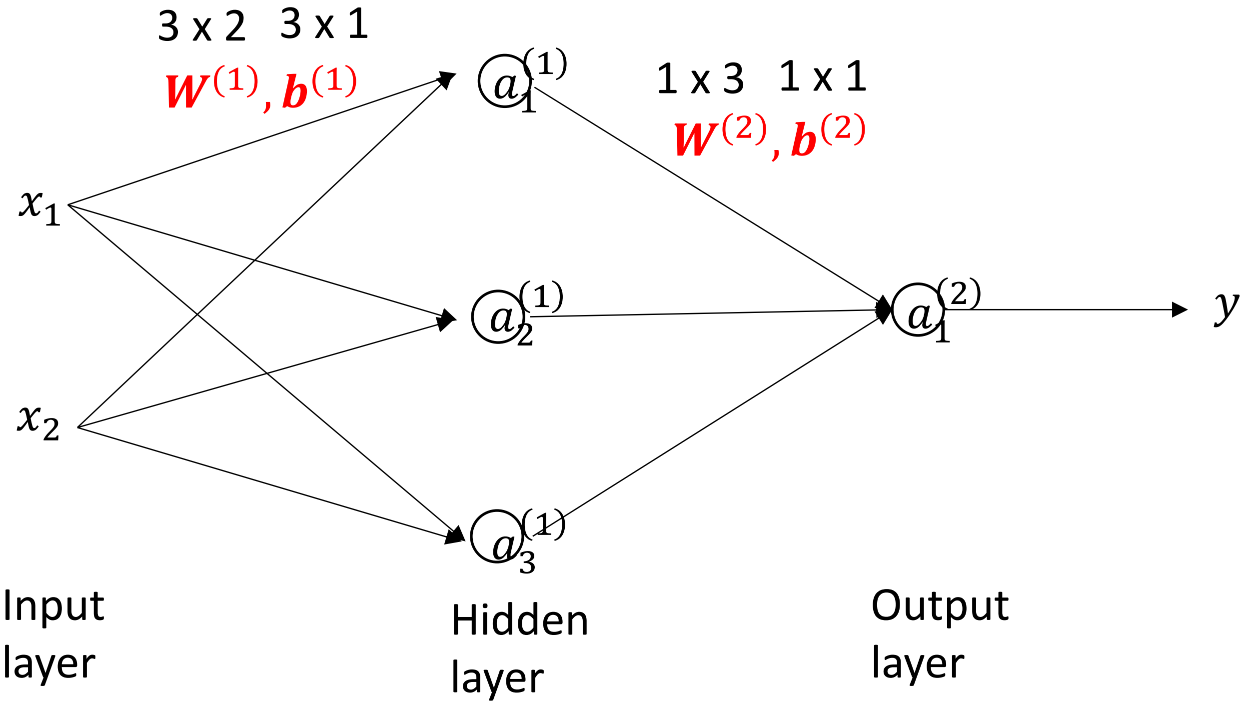


$$\boldsymbol{a}^{(0)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \boldsymbol{w}_j^{(1)} = \begin{bmatrix} w_{j1}^{(1)} \\ w_{j2}^{(1)} \end{bmatrix}$$

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix}, \boldsymbol{w}_j^{(2)} = \begin{bmatrix} w_{j1}^{(2)} \\ w_{j2}^{(2)} \\ w_{j3}^{(2)} \end{bmatrix}$$

# of units	*j=1,2*	*j=1,2,3*	*j=1*
Forward propagation	$\boldsymbol{a}^{(0)} = \boldsymbol{x}$	$z_j^{(1)} = \mathbf{w}_j^{(1)T}\boldsymbol{a}^{(0)} + b_j^{(1)}$ $a_j^{(1)} = \sigma(z_j^{(1)})$	$z_j^{(2)} = \mathbf{w}_j^{(2)T}\boldsymbol{a}^{(1)} + b_j^{(2)}$ $a_j^{(2)} = \sigma(z_j^{(2)})$

Writing a neural network in terms of vectors



$$\boldsymbol{a}^{(0)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \boldsymbol{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix}$$

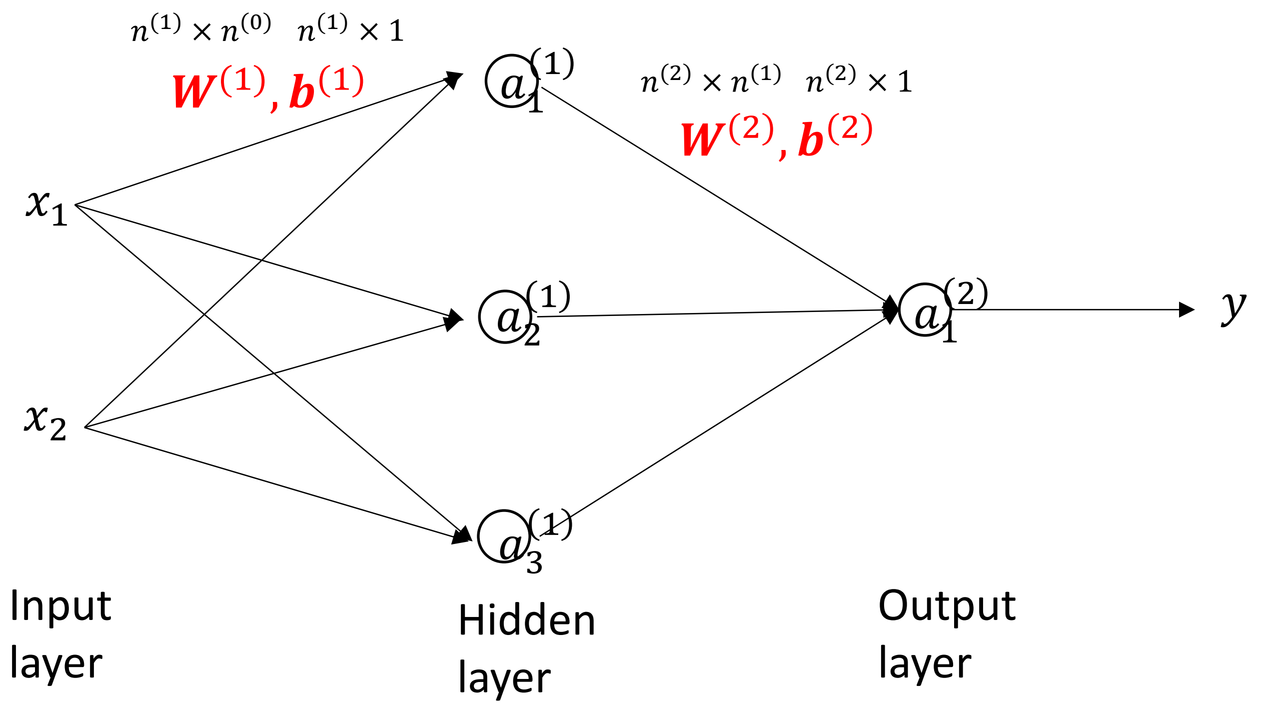
$$\boldsymbol{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}, \boldsymbol{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix},$$

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix},$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \end{bmatrix}$$

# of units	*j=1,2*	*j=1,2,3*	*j=1*
Forward propagation	$\boldsymbol{a}^{(0)} = \boldsymbol{x}$	$\boldsymbol{z}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{a}^{(0)} + \boldsymbol{b}^{(1)}$ $\boldsymbol{a}^{(1)} = \sigma(\boldsymbol{z}^{(1)})$	$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(2)}$ $\boldsymbol{a}^{(2)} = \sigma(\boldsymbol{z}^{(2)})$

Writing a neural network in terms of vectors



$$\boldsymbol{a}^{(0)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \boldsymbol{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix}$$

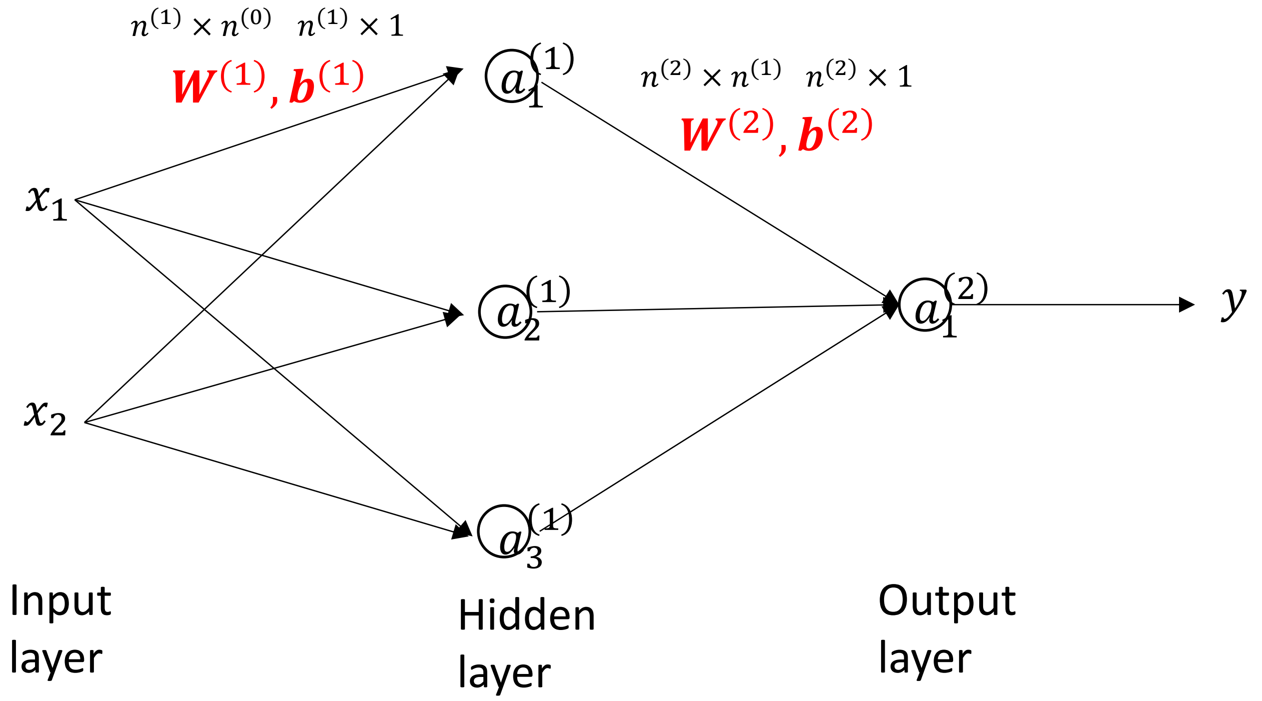
$$\boldsymbol{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}, \boldsymbol{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix},$$

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix},$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \end{bmatrix}$$

# of units	$n^{(0)} = 2$	$n^{(1)} = 3$	$n^{(2)} = 1$
Forward propagation	$\boldsymbol{a}^{(0)} = \boldsymbol{x}$	$\boldsymbol{z}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{a}^{(0)} + \boldsymbol{b}^{(1)}$ $\boldsymbol{a}^{(1)} = \sigma(\boldsymbol{z}^{(1)})$	$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(2)}$ $\boldsymbol{a}^{(2)} = \sigma(\boldsymbol{z}^{(2)})$

Writing a neural network in terms of vectors



$$\boldsymbol{a}^{(0)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \boldsymbol{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix}$$

$$\boldsymbol{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}, \boldsymbol{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix},$$

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix},$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \end{bmatrix}$$

In short, $y = a_1^{(2)} = \sigma(\sum_{k=1}^{3} w_{1k}^{(2)} \sigma\left(\sum_{l=1}^{2} w_{kl}^{(1)} x_l + b_k^{(1)}\right) + b_1^{(2)})$

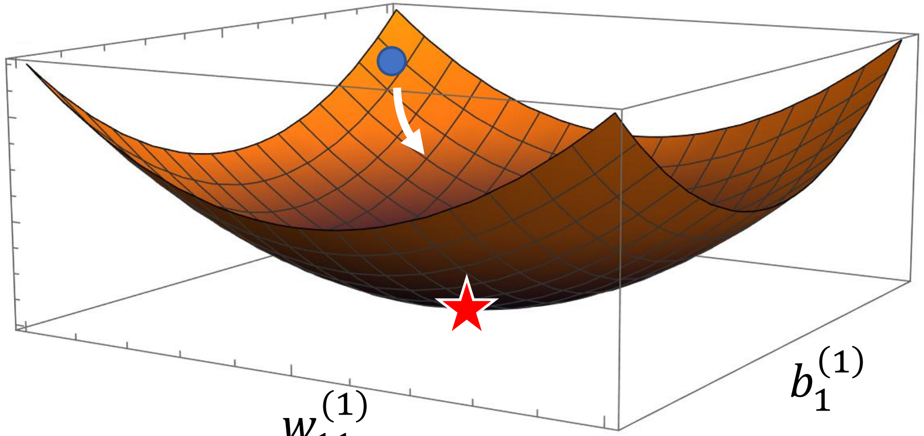
$\longrightarrow$ $y = \boldsymbol{a}^{(2)} = \sigma(\boldsymbol{W}^{(2)} \sigma(\boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)})$

We know how to find w and b for a simple model $y = \sigma(wx + b)$.

What about finding w and b in a neural network?
$y = a^{(2)} = \sigma(\boldsymbol{W}^{(2)}\sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)})$

How to find w and b in a neural network?

- E. g., Model $y = a^{(2)} = \sigma(\boldsymbol{W}^{(2)}\sigma(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)})$
- Cost function $J(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}) = \frac{1}{m}\sum_{i=1}^{m} L(y^i, y_d^i)$



- For the full data set (i=1…m), compute

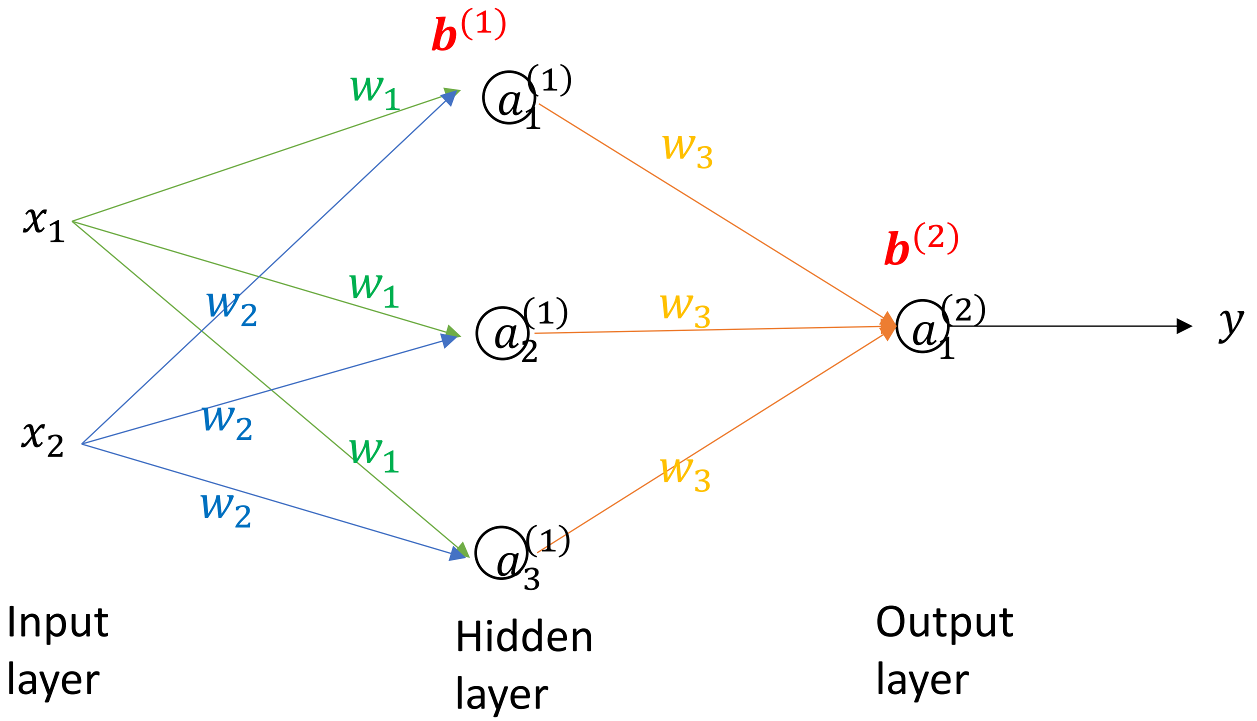
$$-\boldsymbol{\nabla} J = -\left(\frac{\partial J}{\partial w_{jk}^{(l)}}, \dots, \frac{\partial J}{\partial b_j^{(l)}}\right)$$

- ${w_{jk}}_{new}^{(l)} = {w_{jk}}_{old}^{(l)} - \alpha \frac{\partial J}{\partial w_{jk}^{(l)}}$, ${b_j}_{new}^{(l)} = {b_j}_{old}^{(l)} - \alpha \frac{\partial J}{\partial b_j^{(l)}}$

  α is the learning rate

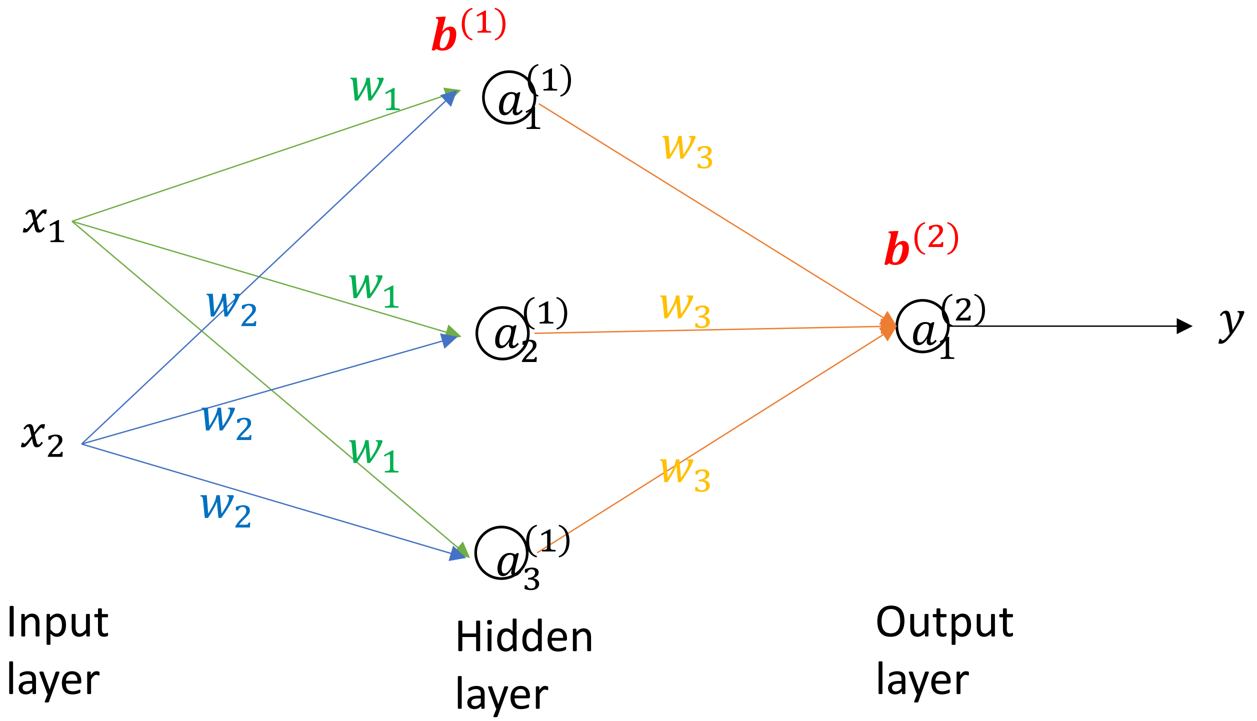
- Iterate until $\boldsymbol{\nabla} J = 0$

NN weight Initialization



Q: What would happen if all initial weights are chosen to be the same (e.g., all zeros)?

NN weight Initialization



More specifically, if weights are symmetric:

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} w_1 & w_2 \\ w_1 & w_2 \\ w_1 & w_2 \end{bmatrix}, \boldsymbol{W}^{(2)} = \begin{bmatrix} w_3 & w_3 & w_3 \end{bmatrix}$$

(e.g. all zero weights, all constant weights)

Then $a_1^{(1)} = a_2^{(1)} = a_3^{(1)}$

and

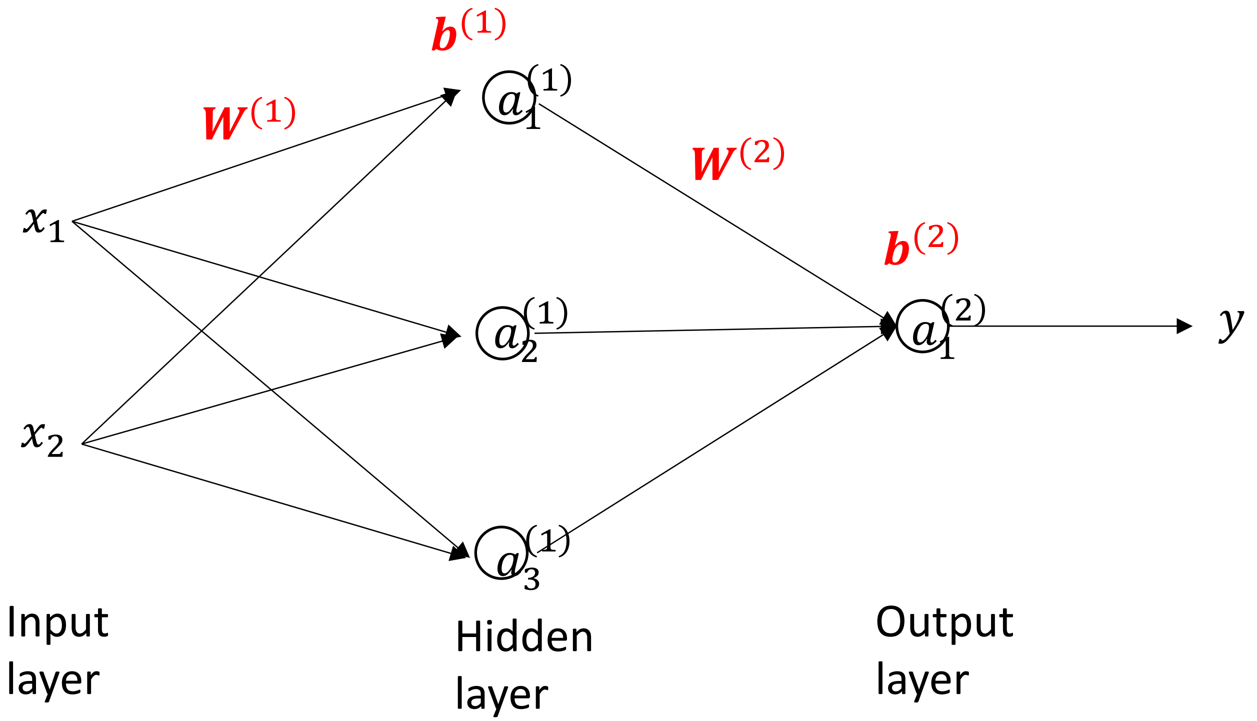
$$\begin{bmatrix} w_{1new} & w_{2new} \\ w_{1new} & w_{2new} \\ w_{1new} & w_{2new} \end{bmatrix} = \begin{bmatrix} w_1 & w_2 \\ w_1 & w_2 \\ w_1 & w_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial J}{\partial w_1} & \frac{\partial J}{\partial w_2} \\ \frac{\partial J}{\partial w_1} & \frac{\partial J}{\partial w_2} \\ \frac{\partial J}{\partial w_1} & \frac{\partial J}{\partial w_2} \end{bmatrix}$$

The NN never learns to have non-symmetric weights -> $a_1^{(1)} = a_2^{(1)} = a_3^{(1)}$ during training

If the NN weights are **symmetric** → Gradient descent will update these hidden units in the same way → All hidden units in the same layer will be identical throughout training iterations → Equivalent to NN with just 1 hidden unit.

https://www.deeplearning.ai/ai-notes/initialization/

NN weight Initialization



To make the different hidden units approximate different functions

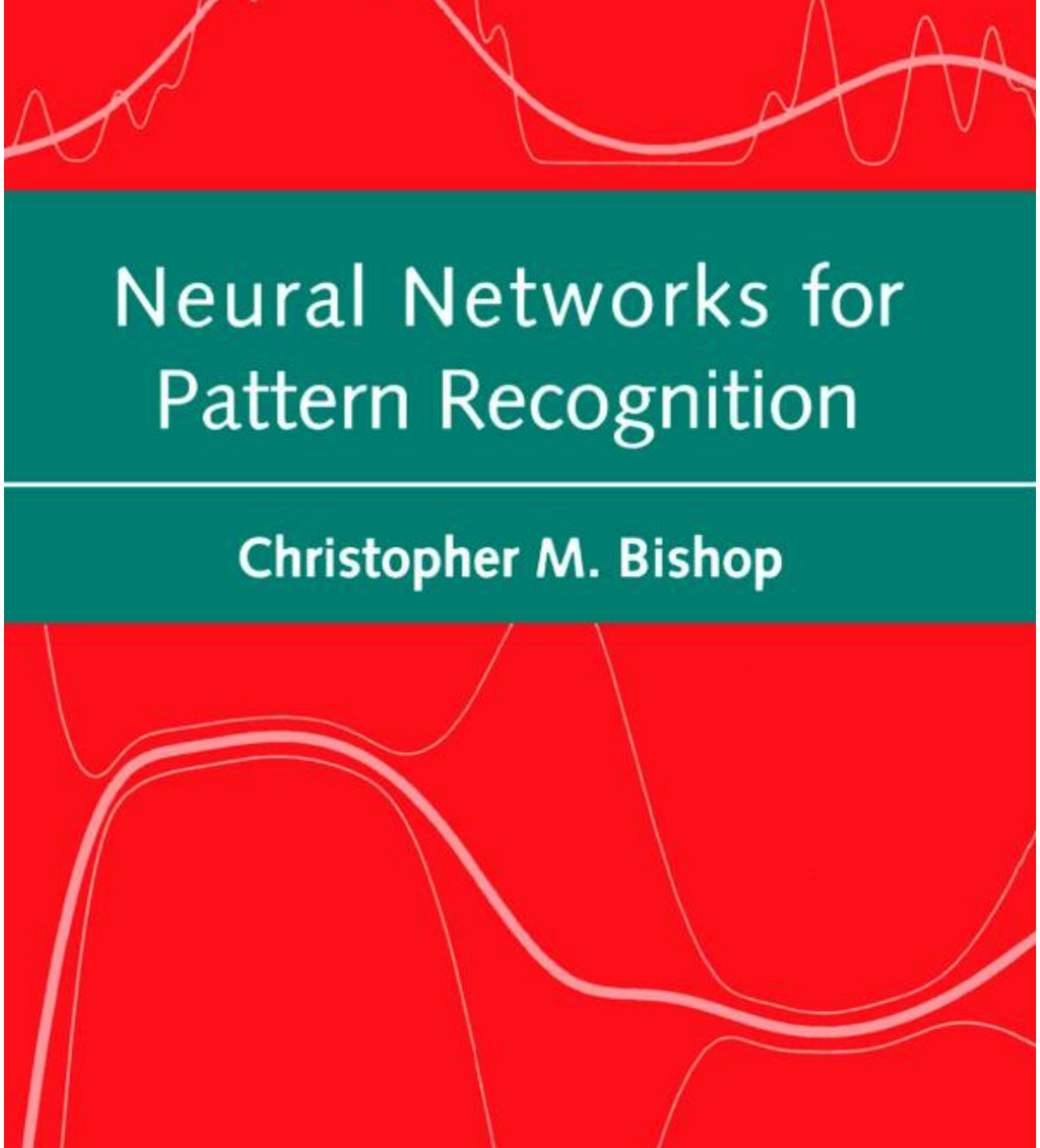
→ Initialized weights $w_{jk}^{(l)}$ to random values

What about biases?

→ Biases can just be zeros

$$\boldsymbol{b}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \ \boldsymbol{b}^{(2)} = [0]$$

If the NN weights are **symmetric** → Gradient descent will update these hidden units in the same way → All hidden units in the same layer will be identical throughout training iterations → Equivalent to NN with just 1 hidden unit.

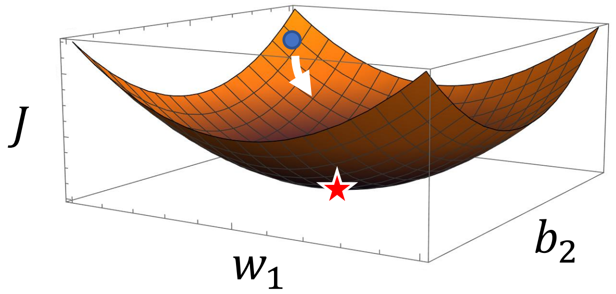


Chapter 4
The Multi-layer Perceptron

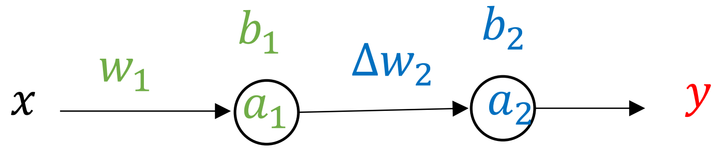
Forward & Back Propagation: 1. single neuron

Exercise:

Model: $y = \sigma(w_2\sigma(w_1 x + b_1) + b_2)$



∇J vary with w_1, w_2, b_1, b_2



How important is w_2 for changing the cost function J?

Forward propagation →

$z_1 = w_1 x + b_1$
$a_1 = \sigma(z_1)$

$z_2 = w_2 a_1 + b_2$
$a_2 = \sigma(z_2)$

$y = a_2$

Cost function:
$J = (y - y_d)^2$

$\frac{\partial a_2}{\partial w_2}$ $\frac{\partial J}{\partial a_2}$

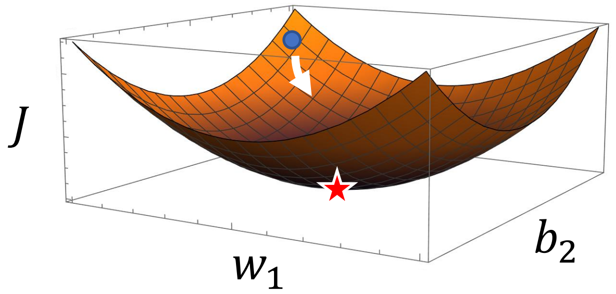
← *Back propagation*

$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial a_2}\frac{\partial a_2}{\partial w_2}$$
$$= 2(y - y_d)\sigma'(z_2)a_1$$

Forward & Back Propagation: 1. single neuron

Exercise:

Model: $y = \sigma(w_2\sigma(w_1 x + b_1) + b_2)$



∇J vary with w_1, w_2, b_1, b_2

$$x \xrightarrow{\Delta w_1} \overset{b_1}{(a_1)} \xrightarrow{w_2} \overset{b_2}{(a_2)} \longrightarrow y$$

How important is w_1 for changing the cost function J?

Forward propagation →

$z_1 = w_1 x + b_1$ $a_1 = \sigma(z_1)$	⇄	$z_2 = w_2 a_1 + b_2$ $a_2 = \sigma(z_2)$	⇄	$y = a_2$	⇄	Cost function: $J = (y - y_d)^2$
	$\frac{\partial a_1}{\partial w_1}$		$\frac{\partial a_2}{\partial a_1}$		$\frac{\partial J}{\partial a_2}$	

← *Back propagation*

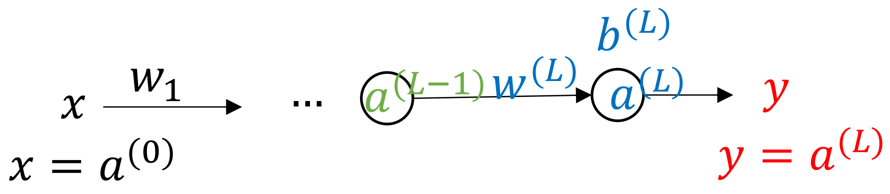
$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial a_2}\frac{\partial a_2}{\partial a_1}\frac{\partial a_1}{\partial w_1}$$
$$= 2(y - y_d)\sigma'(z_2)w_2\sigma'(z_1)x$$

When training a neural network

Repeat these steps:

1. Forward propagate an input
2. Compute the cost function
3. Compute the gradients of the cost with respect to parameters using backpropagation
4. Update each parameter using the gradients, according to the optimization algorithm

Forward & Back Propagation: 1. single neuron



Forward propagation

Input: $a^{(0)} = x$

...

$z^{(L-1)} = w^{(L-1)} a^{(L-2)} + b^{(L-1)}$

$a^{(L-1)} = \sigma(z^{(L-1)})$

$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$

$a^{(L)} = \sigma(z^{(L)})$

Output: $y = a^{(L)}$

Cost function $J = (y - y_d)^2$

Back propagation (use chain rules)

$$\frac{\partial J}{\partial w^{(L)}} =$$

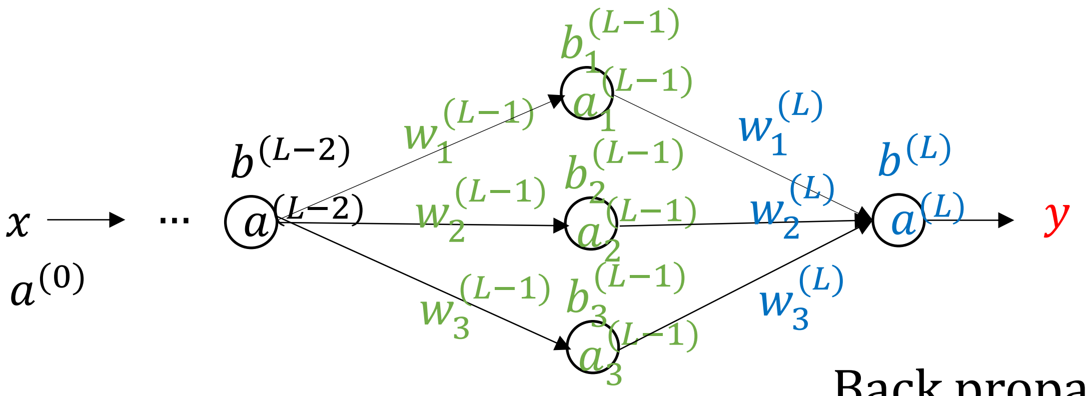
$$\frac{\partial J}{\partial b^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial b^{(L)}} =$$

$$\frac{\partial J}{\partial w^{(L-1)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial w^{(L-1)}}$$

$$\frac{\partial J}{\partial b^{(L-1)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial b^{(L-1)}}$$

$$= 2(a^{(L)} - y_d)\, \sigma'(z^{(L)}) w^{(L)} \sigma'(z^{(L-1)})$$

Forward & Back Propagation: 2. multiple hidden units



$x = a^{(0)}$

Forward propagation

Input: $a^{(0)} = x$

…

$z_k^{(L-1)} = w_k^{(L-1)} a^{(L-2)} + b_k^{(L-1)}$

$a_k^{(L-1)} = \sigma(z_k^{(L-1)})$

$z^{(L)} = \sum_{k=1}^{3} w_k^{(L)} a_k^{(L-1)} + b^{(L)}$

$a^{(L)} = \sigma(z^{(L)})$

Output: $y = a^{(L)}$

Cost function $J = (y - y_d)^2$

Back propagation (use chain rules)

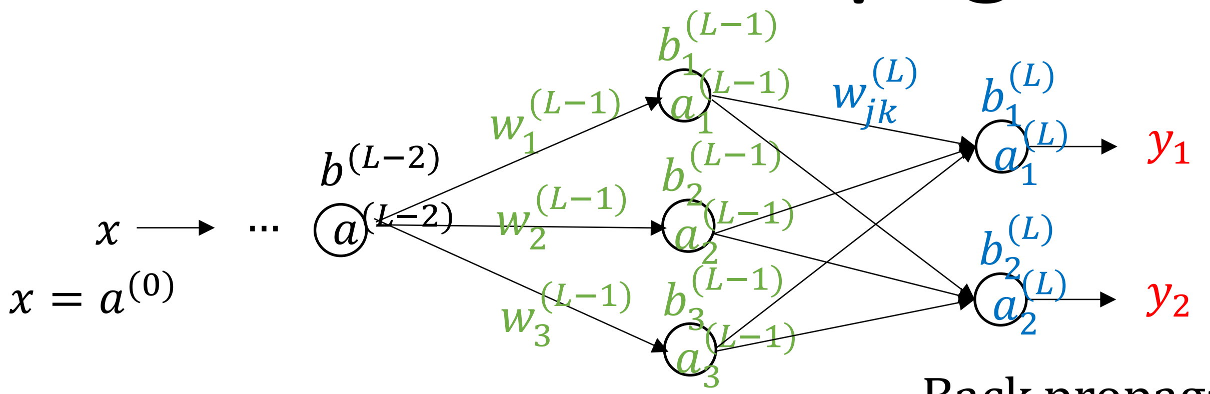
$$\frac{\partial J}{\partial w_k^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial w_k^{(L)}} = 2(a^{(L)} - y_d)\, \sigma'(\mathrm{z}^{(\mathrm{L})})\, a_k^{(L-1)}$$

$$\frac{\partial J}{\partial b^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial b^{(L)}} = 2(a^{(L)} - y_d)\, \sigma'(\mathrm{z}^{(\mathrm{L})})$$

$$\frac{\partial J}{\partial w_k^{(L-1)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_k^{(L-1)}}{\partial w_k^{(L-1)}}$$
$$= 2(a^{(L)} - y_d)\, \sigma'(\mathrm{z}^{(\mathrm{L})}) w_k^{(L)} \sigma'(z_k^{(L-1)}) a^{(L-2)}$$

$$\frac{\partial J}{\partial b_k^{(L-1)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_k^{(L-1)}}{\partial b_k^{(L-1)}}$$
$$= 2(a^{(L)} - y_d)\, \sigma'(\mathrm{z}^{(\mathrm{L})}) w_k^{(L)} \sigma'(z_k^{(L-1)})$$

Forward & Back Propagation: 3. multiple outputs



Forward propagation

Input: $a^{(0)} = x$

...

$z_k^{(L-1)} = w_k^{(L-1)} a^{(L-2)} + b_k^{(L-1)}$

$a_k^{(L-1)} = \sigma(z_k^{(L-1)})$

$z_j^{(L)} = \sum_{k=1}^{3} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$

$a_j^{(L)} = \sigma(z_j^{(L)})$

Output: $\mathrm{y_j} = a_j^{(L)}$

Cost function $J = \sum_{j=1}^{2} (y_j - y_{d,j})^2$

Back propagation (use chain rules)

$$\frac{\partial J}{\partial w_{jk}^{(L)}} = \frac{\partial J}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial w_{jk}^{(L)}} = 2(a_j^{(L)} - y_{d,j})\, \sigma'(\mathrm{z_j^{(L)}}) a_k^{(L-1)}$$

$$\frac{\partial J}{\partial b_j^{(L)}} = \frac{\partial J}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial b_j^{(L)}} = 2(a_j^{(L)} - y_{d,j})\, \sigma'(z_j^{(L)})$$

$$\frac{\partial J}{\partial w_k^{(L-1)}} = \sum_{j=1}^{2} \frac{\partial J}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_k^{(L-1)}}{\partial w_k^{(L-1)}}$$
$$= \sum_{j=1}^{2} 2(a_j^{(L)} - y_{d,j})\, \sigma'(z_j^{(L)}) \mathrm{w}_{jk}^{(L)}\ \sigma'(z_k^{(L-1)}) a^{(L-2)}$$

$$\frac{\partial J}{\partial b_k^{(L-1)}} = \sum_{j=1}^{2} \frac{\partial J}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_k^{(L-1)}}{\partial b_k^{(L-1)}}$$
$$= \sum_{j=1}^{2} 2(a_j^{(L)} - y_{d,j})\, \sigma'(z_j^{(L)}) \mathrm{w}_{jk}^{(L)}\ \sigma'(z_k^{(L-1)})$$

Summary

- Universal function approximator
- Gradient descent: a method to find weights and biases that minimize J
- Calculate $\boldsymbol{\nabla} J$
  - One example
  - A full dataset
- Weight initialization
- Forward and backpropagation (a clever way to get gradients of J)

Outline

- Neural networks in problems governed by PDEs
- NN basics (Lecture 1-3)
  Universal function approximation
  Gradient descent
  Back propagation
- Advanced topics: NN's spectral bias
- Physics-informed NN (Lecture 4-5)

Spectral bias: A Fourier analysis of NN

Xu et al. (2022)

For a single-hidden-layer NN with input $x \in \mathbb{R}$ and output $u \in \mathbb{R}$

$$u(x) = \sum_{j=1}^{m} w_j^{(1)} \sigma(w_j^{(0)} x + b_j), \quad \text{where} \quad \theta_j = \{w_j^{(0)}, w_j^{(1)}, b_j\}$$

The mean square loss measures the distance between the NN output $u(x)$ and ground truth $u_g(x)$

$$J \equiv \int_{-\infty}^{\infty} \tfrac{1}{2}(u(x) - u_g(x))^2 dx = \int_{-\infty}^{\infty} \tfrac{1}{2}|\hat{u}(k) - \hat{u}_g(k)|^2 dk$$

, where $\hat{u}(k)$ denotes the Fourier transform of $u(x)$

The loss at frequency k is $J(k) = \frac{1}{2}|\hat{u} - \hat{u}_g|^2$

For tanh activation function, the gradient of loss at $J(k)$ with respect to NN parameters θ_j is

$$\left|\frac{\partial J(k)}{\partial \theta_j}\right| \approx A(k) e^{-\left|\frac{\pi k}{2 w_j^{(0)}}\right|}$$

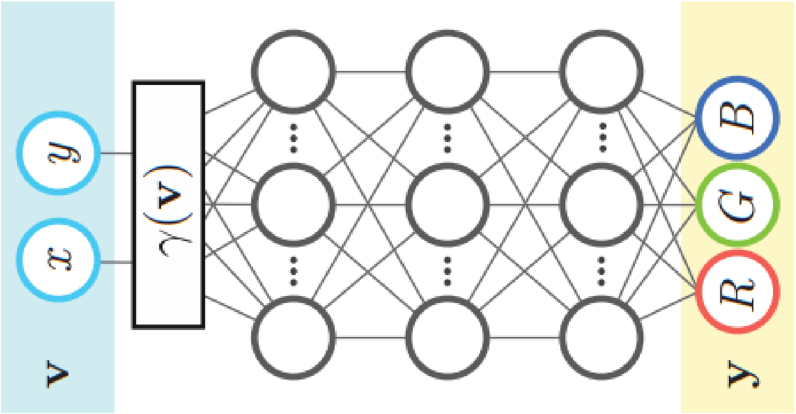
where $A(k) \in [0, +\infty)$ is the amplitude of $\hat{u} - \hat{u}_g$

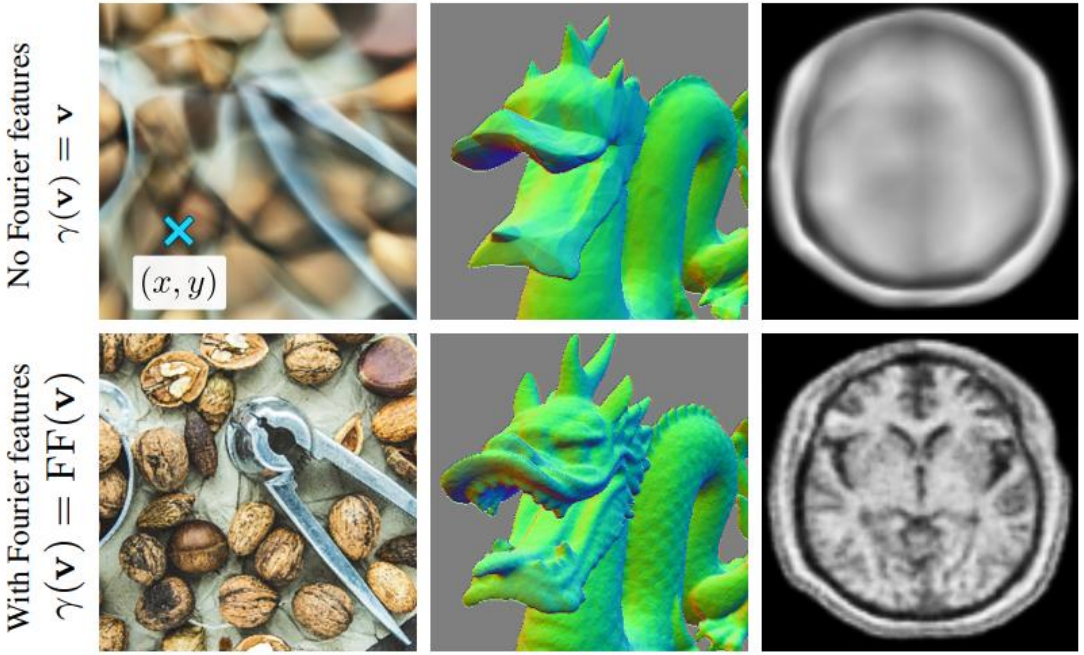
This gradient decay rapidly with frequency k!! This is a simple explanation of the spectral bias.

Spectral bias is bad for multiscale problems

- One solution: Fourier feature $\gamma(\mathbf{v})$ Tancik et al (2020), Wang et al (2021)
  $\gamma(\mathbf{v})$ maps the input coordinates to a feature space before passing them to NN layers.

$$\boxed{\gamma(\mathbf{v}) = \left[a_1 \cos(2\pi \mathbf{b}_1^{\mathrm{T}} \mathbf{v}), a_1 \sin(2\pi \mathbf{b}_1^{\mathrm{T}} \mathbf{v}), \ldots, a_m \cos(2\pi \mathbf{b}_m^{\mathrm{T}} \mathbf{v}), a_m \sin(2\pi \mathbf{b}_m^{\mathrm{T}} \mathbf{v})\right]^{\mathrm{T}}}$$





(b) Image regression $(x,y) \to$ RGB
(c) 3D shape regression $(x,y,z) \to$ occupancy
(d) MRI reconstruction $(x,y,z) \to$ density

Main idea

Wang & Lai, J. Comput. Phys. (2024)

- When NN training error is small, the learning becomes really slow. NN learns better when $u \approx O(1)$. Inspired by the principals of perturbation theory, MSNN includes a superposition of multi-stage NNs, with each stage using a new NN to fit the residue rescaled to O(1) from the previous NN.

$$\boxed{u_c^{(n)}(x) = u_0(x) + \epsilon_1 u_1(x) + \epsilon_2 u_2(x) + \ldots + \epsilon_n u_n(x) = \sum_{j=0}^{n} \epsilon_j u_j(x)}$$

where $1 \gg \epsilon_1 \gg \epsilon_2 \gg \ldots \gg \epsilon_n$

$u_0(x)$, $u_1(x)$ and $u_n(x)$ are all of order $\mathcal{O}(1)$ NNs of different stages

Related work with similar flavors: Multi-level neural networks: Aldirany et al. (2023), Multifidelity neural networks: Howard et al. (2023, 2024), Precision Machine Learning: Michaud et al. (2023)…etc

Multistage-NN (MSNN)

Wang & Lai, J. Comput. Phys. (2024)