Deep Learning
Concepts & Algorithms

Lu Lu

Department of Chemical and Biomolecular Engineering
Penn Institute for Computational Science
University of Pennsylvania

Tianyuan Mathematical Center in Southeast China
Dec 8, 2021

Penn
University of Pennsylvania

Deep Learning for Science and Engineering Teaching Kit

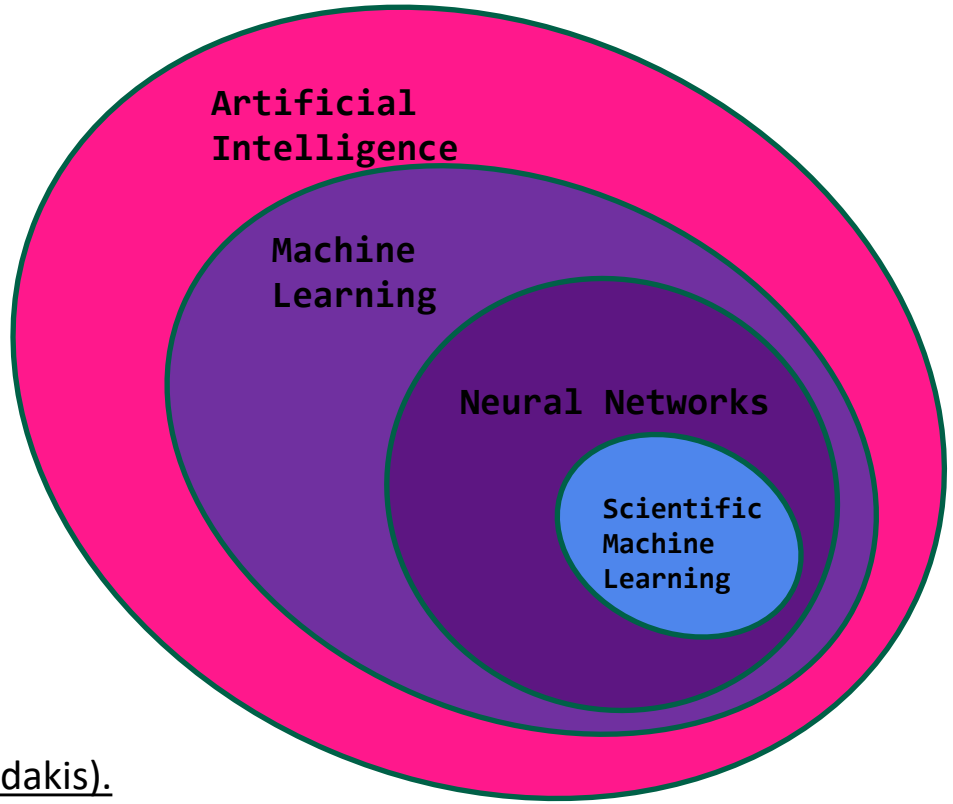
Deep Learning for Scientists and Engineers

Instructors: George Em Karniadakis, Khemraj Shukla, Lu Lu
Teaching Assistants: Vivek Oommen and Aniruddha Bora

(To be released in 2022)

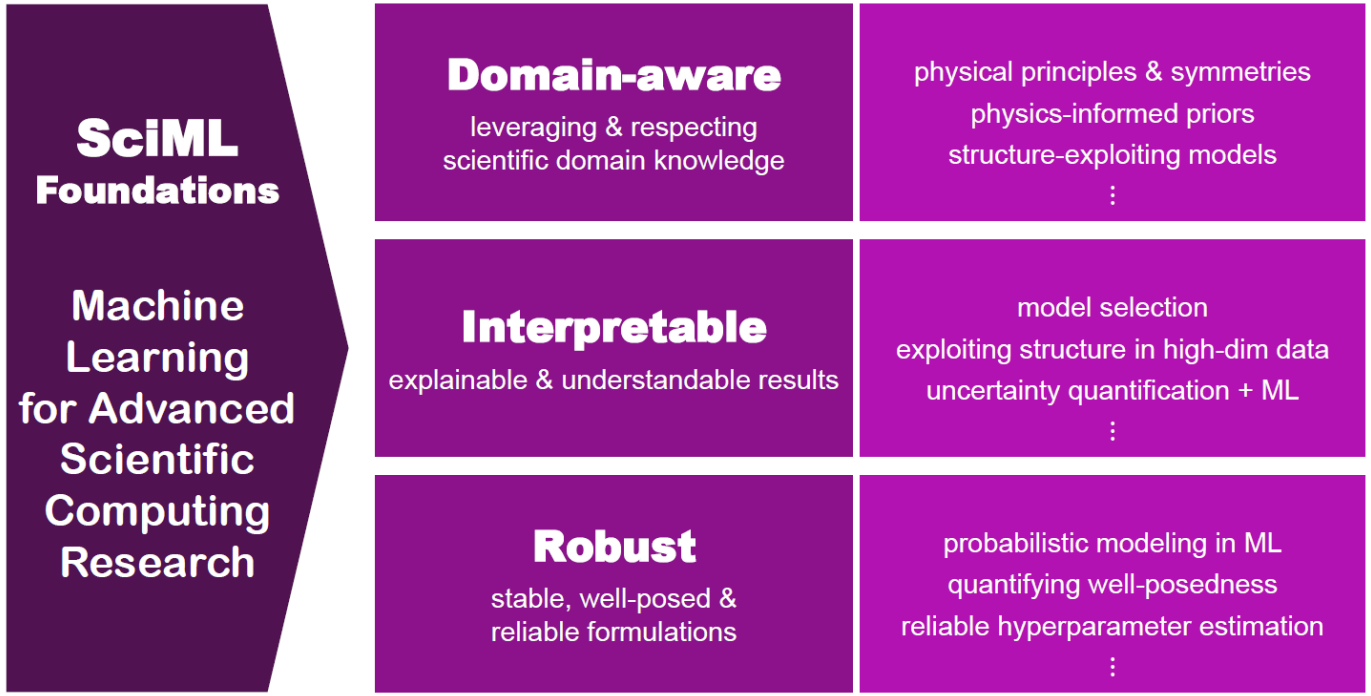
https://www.import.io/post/history-of-deep-learning/

- Artificial intelligence (AI) > Machine Learning (ML) > Deep Learning > Scientific Machine Learning (SciML).
- The expression "Deep Learning" was (probably) first used by Igor Aizenberg and colleagues around 2000.
- 1960s: Shallow Neural Networks.
- 1982: Hopfield Network – A Recurrent NN.
- 1988-89: Learning by backpropagation, Rumelhart, Hinton & Williams; hand-written text, LeCun.
- 1993 NVIDIA was founded; GeForce is the first GPU.
- 1990s Unsupervised Deep Learning.
- 1993: A Recurrent NN with 1,000 layers (Jürgen Schmidhuber)
- 1994: NN for solving PDEs, Dissanayake & Phan-Thien
- 1998: Gradient-based learning, LeCun.
- 1998: ANN for solving ODEs&PDEs, Lagaris, Likas & Fotiadis
- 1990-2000: Supervised Deep Learning.
- 2006: A fast learning algorithm for deep belief nets, Hinton.
- 2006-present: Modern Deep Learning.
- 2009: ImageNet: A large-scale hierarchical image database (Fei Fei).
- 2010: GPUs are only up to 14 times faster than CPUs (Intel).
- 2010: Tackling the vanishing/exploding gradients: Glorot & Bengio.
- 2011: AlexNet – Convolutional NN (CNN) - Alex Krizhevsky.
- 2014: Generative Adversarial Networks (GANs) – Ian Goodfellow.
- 2015: Batch normalization, Ioffe & Szegedy.
- 2017: PINNs: Physics-Informed Neural Networks (Raissi, Perdikaris, Karniadakis).
- 2019: Scientific Machine Learning (ICERM workshop Jan. 2019; DOE report, Feb 2019).
- 2019: DeepOnet – Operator regression (Lu, Jin, Karniadakis).



Basic Research Needs Workshop for Scientific Machine Learning
Core Technologies for Artificial Intelligence, DOE ASCR Report, Feb 2019

❑ Scientific machine learning (**SciML**) is a core component of artificial intelligence (AI) and a computational technology that can be trained, with scientific data, to augment or automate human skills.

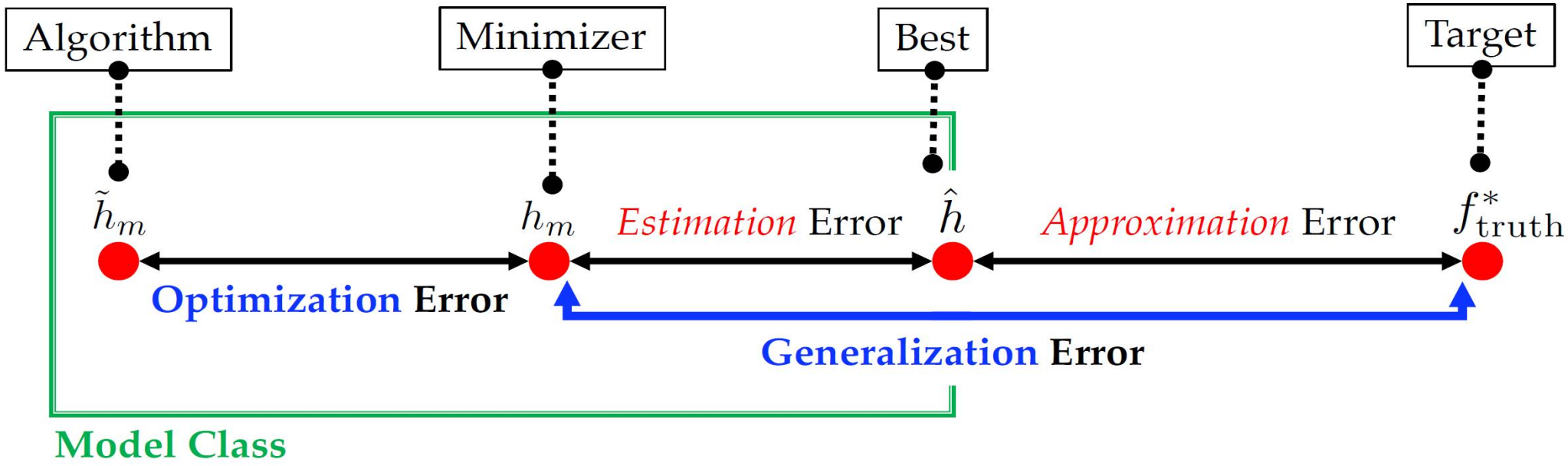


❑ SciML must achieve the same level of scientific rigor expected of established methods deployed in science and applied mathematics. Basic requirements include validation and limits on inputs and context implicit in such validations, as well as verification of the basic algorithms to ensure they are capable of delivering known prototypical solutions.

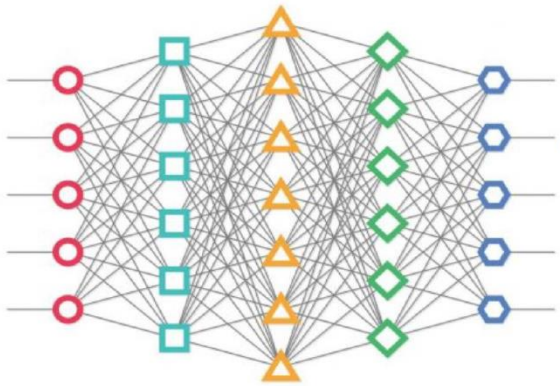
❑ Can SciML achieve Robustness?

- NSF/ICERM Workshop on SciML, January 28-30, 2019 (Organizers: J. Hesthaven & G.E. Karniadakis)
- https://icerm.brown.edu/events/ht19-1-sml/)

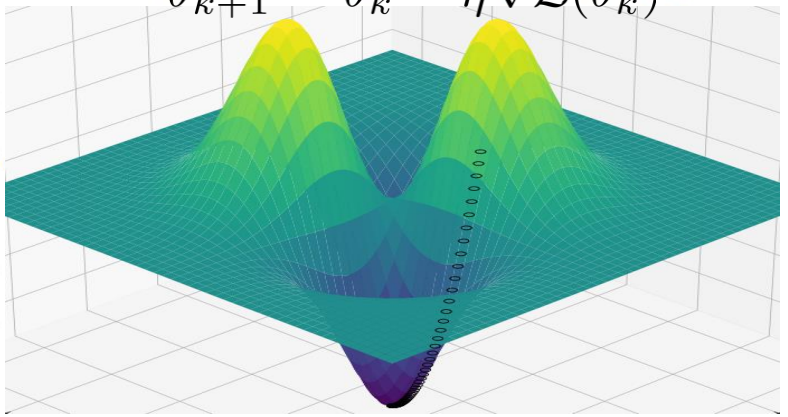
Fundamental Questions



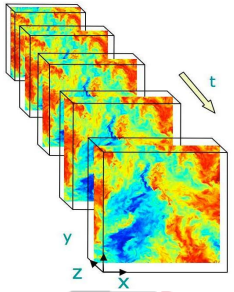
$$\theta_{k+1} = \theta_k - \eta \nabla \mathcal{L}(\theta_k)$$



Neural Networks



Gradient-based Optimization



NIH National Heart, Lung, and Blood Institute

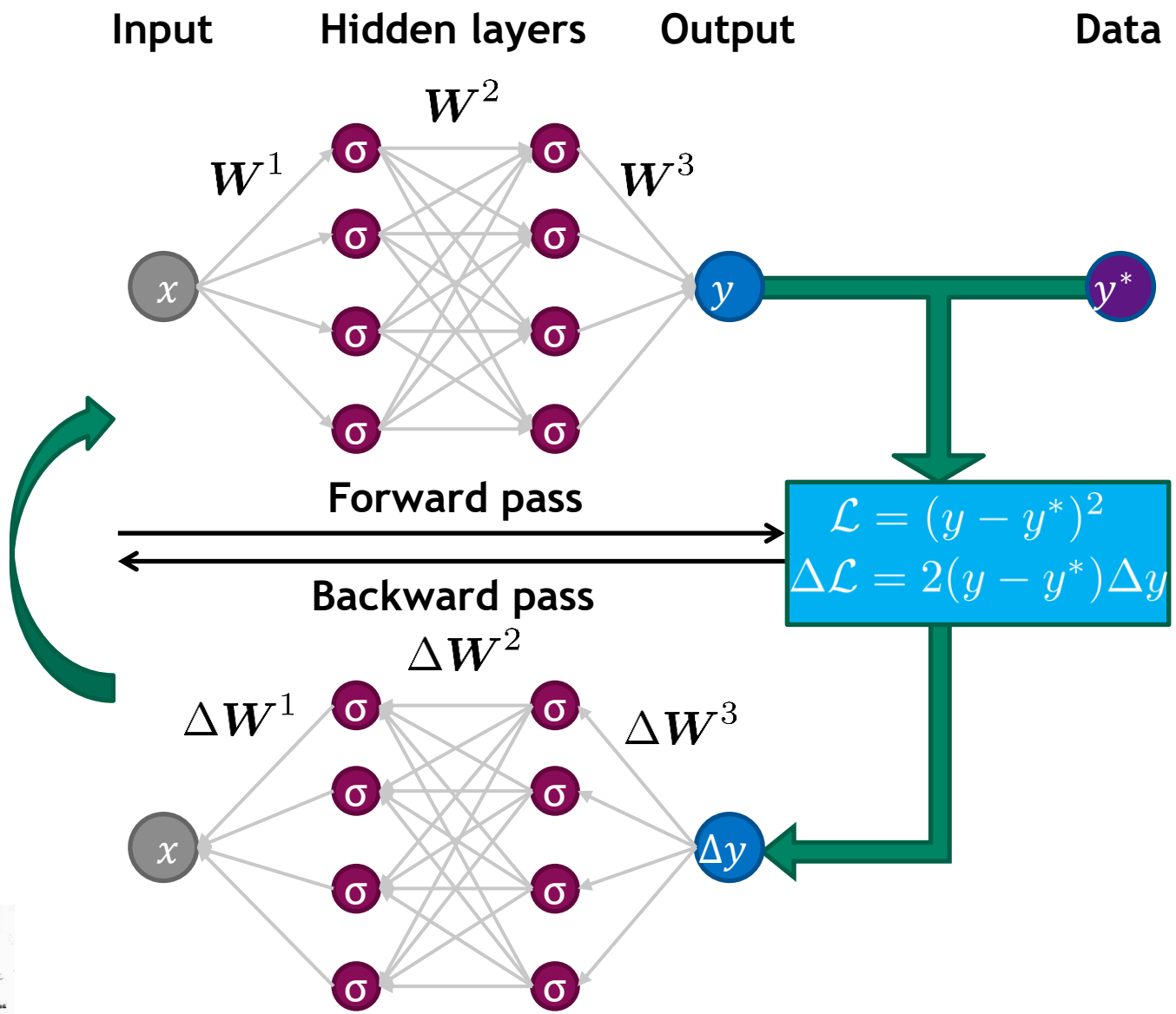
UCI Machine Learning Repository

Johns Hopkins Turbulence Database

BioData CATALYST

Data

Workflow in a Neural Network



- Input layer (layer 0):
  - $z^0 = x \in \mathbb{R}^d$
- Hidden layers:
  - Layer 1: $z^1 = \sigma(\boldsymbol{W}^1 x + \boldsymbol{b}^1) \in \mathbb{R}^{N_1}$
  - Layer 2: $z^2 = \sigma(\boldsymbol{W}^2 z^1 + \boldsymbol{b}^2) \in \mathbb{R}^{N_2}$
- Output layer (layer 3):
  - $y = z^3 = \boldsymbol{W}^3 z^2 + \boldsymbol{b}^3 \in \mathbb{R}$

A Neural Network for Regression

- Define the affine transformation in l-th layer

$$T^l(x) = \boldsymbol{W}^l x + \boldsymbol{b}^l$$

- Activation function σ

  Popular choices: $tanh(x), max\{x, 0\}$ (Rectified Linear Unit, ReLU)

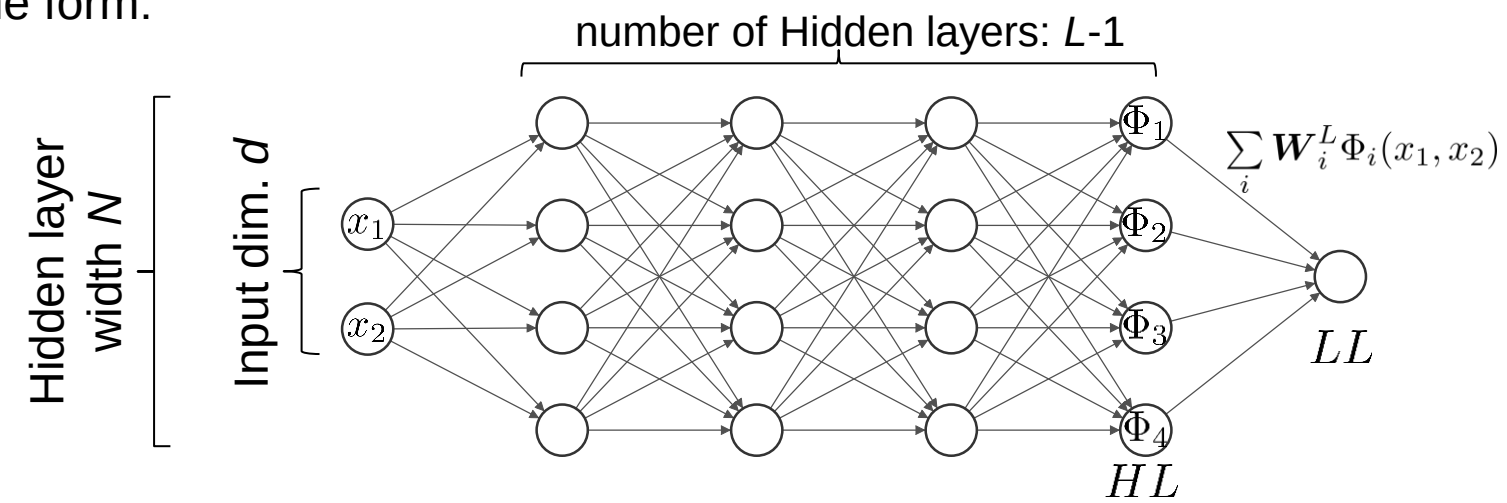
- The $L-1$ hidden layers of a feedforward neural network:

$$\mathcal{N}_{HL}(x) = \sigma \circ T^{L-1} \circ ... \circ \sigma \circ T^1(x)$$

  Where $\circ$ denotes composition of functions

- For regression, a DNN Is typically of the form:

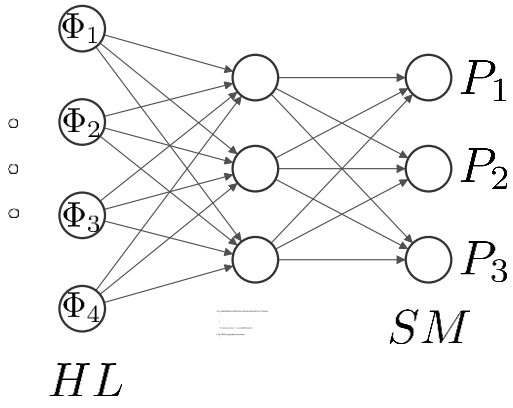
$$\mathcal{N}(x; \boldsymbol{\theta}) = T^L \circ \mathcal{N}_{HL}(x)$$



- Network parameters: $\boldsymbol{\theta} = \{\boldsymbol{W}^l, \boldsymbol{b}^l\}_{1 \le l \le L}$

A Neural Network for Classification

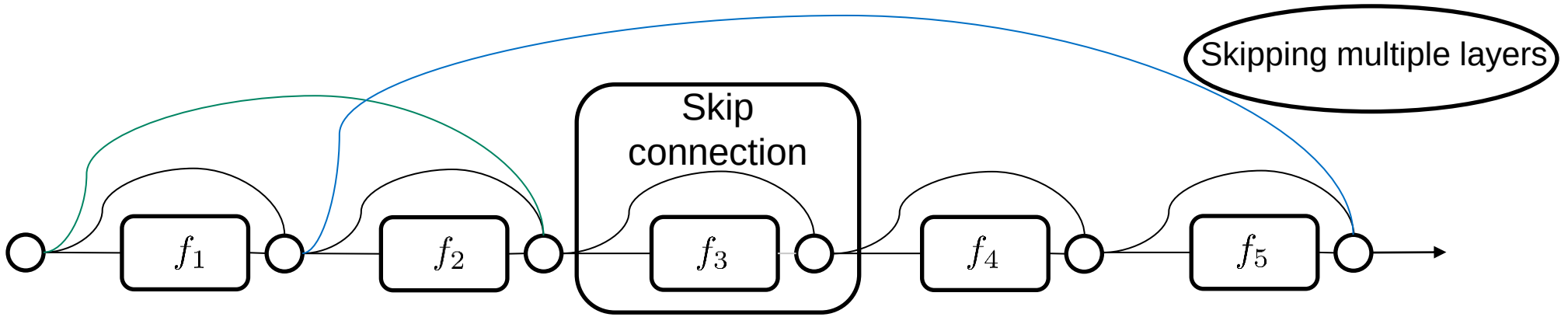
- For classification, define the softmax function for K classes
  - $f_{SM}(\xi_i) = \frac{exp(\xi_i)}{\sum_{j=1}^{K} exp(\xi_j)}$
  - $0 \leq f_{SM}(\xi_i) \leq 1$
  - $\sum_i f_{SM}(\xi_i) = 1$
  - Convert any vector ξ to a probability vector



- The DNN is typically of the form
  - $\mathcal{N}(x;\boldsymbol{\theta}) = f_{SM} \circ T^L \circ \mathcal{N}_{HL}(x)$

Building Different NNs: ResNet

- Residual network (**ResNet**)
- Replace $\sigma \circ T^l$ with $I + \sigma \circ T^l$
- I: Identity function
- $\mathcal{N}(x) = T^l \circ (I + \sigma \circ T^{L-1}) \circ ... \circ (I + \sigma \circ T^2) \circ \sigma \circ T^1(x)$



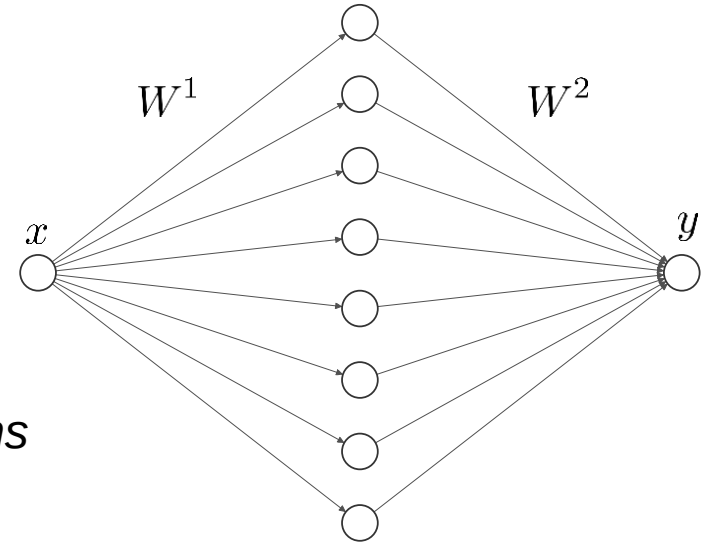
Universal Function Approximation (single layer)

Definition. We say that σ is *discriminatory* if for a measure $\mu \in M(I_N)$

$$\int_{I_N} \sigma(W^1 x + b^1) d\mu(x) = 0$$

for all $W^1 \in \mathbb{R}^n$ and $b^1 \in \mathbb{R}^n$ implies that $\mu = 0$

The space of finite, signed regular Borel measures on I_N is denoted by $M(I_N)$



Definition. We say that σ is *sigmoidal* if

$$\sigma(x) \to \begin{cases} 1 & \text{as } x \to +\infty \\ 0 & \text{as } x \to -\infty \end{cases}$$

Theorem 1. Let σ be *any continuous discriminatory function. Then finite sums of the form*

$$y = \sum_{j=1}^{N} W_j^2 \sigma(W_j^1 x + b_j^1)$$

are dense in $C(I_N)$. *In other words, given any* $f \in C(I_N)$ *and* $\epsilon > 0$, *there is a sum,* y, *of the above form, for which*

$$|y - f(x)| < \epsilon \qquad for\ all\ x \in I_N$$

- **Note**: The set of all functions y does not form a vector space since it is not closed under addition.

G. Cybenko, "Approximation by superpositions of a sigmoidal function", Mathematics of Control, Signals and Systems, 303-314, 2(4), 1989

Universal Functional Approximation (single layer)

Theorem (*Chen and Chen, 1993*):

Suppose that U is a compact set in $C[a,b], f$ is a continuous functional defined on U, and $\sigma(x)$ is a bounded sigmoidal function, then for any $\epsilon > 0$, there exist $m+1$ points $a = x_0 < ... < x_m = b$, a positive integer N and constants $W_i^2, b_i, W_{i,j}, i = 1, 2, ..., N,$ $j = 1, 2, ..., m$

Such that

$$\left| f(u) - \sum_{i=1}^{N} W_i^2 \sigma \left(\sum_{j=0}^{m} W_{i,j}^1 u(x_j) + b_i \right) \right| < \epsilon$$

holds for all $u \in U$.

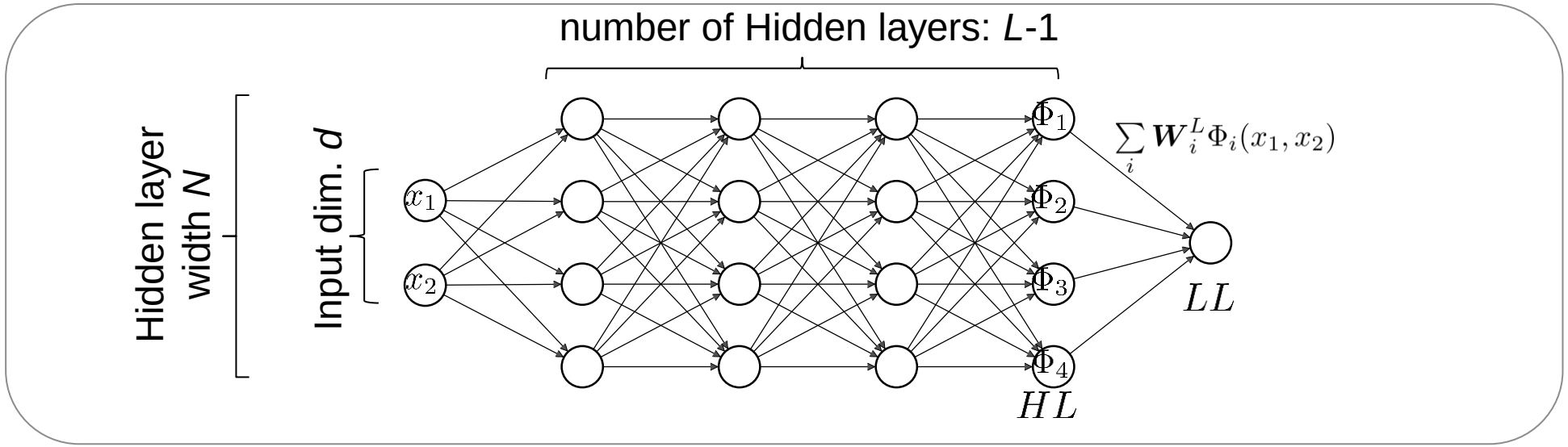
T.P. Chen and H. Chen, Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Transactions on Neural Networks, 910-918, 4(6), 1993.

Adaptive Basis Viewpoint

We consider a family of neural networks $\mathcal{N}_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}$ consisting of $L-1$ hidden layers of width N composed with a final linear layer, admitting the representation

$$\mathcal{N}_{\boldsymbol{\theta}}(x) = \sum_{i=1}^{N} \boldsymbol{W}_i^L \Phi_i(x; \boldsymbol{\theta}_H)$$

where $\boldsymbol{W}^L$ and $\boldsymbol{\theta}_H$ are the parameter corresponding to the final linear layer and the hidden layers respectively. We interpret $\boldsymbol{\theta}$ as a concatenation of $\boldsymbol{W}^L$ and $\boldsymbol{\theta}_H$.



This view point makes it clear that $\boldsymbol{\theta}_H$ parameterizes the basis (like FEM mesh & Shape functions), while $\boldsymbol{W}^L$ are just coefficients for these basis functions.

Shallow networks vs Deep networks

- Universal approximator:
  - Shallow networks: width $\to \infty$
  - Deep networks: width $\sim d_{in} + d_{out}$ (for ReLU NN) [Hanin & Sellke, 2017]

- From approximation point of view: Deep networks perform better than shallow ones of comparable size [Mhaskar, 1996]
  - $\epsilon^{-d/p}$ neurons can approximate C^p functions with error ϵ [Mhaskar, 1996]
  - e.g., a 3-layer NN with 10 neurons per layer may be better than a 1-layer NN with 30 neurons

  - $\frac{size_{deep}}{size_{shallow}} \sim \epsilon^{d_{in}}$ [Mhaskar & Poggio, 2016]

  - There exist functions expressible by a small 2-hidden-layer NN, which cannot be approximated by any shallow NN with the same accuracy, unless its width is exponential in the dimension. [Eldan & Shamir, 2016]
  - The number of neurons needed by a shallow NN to approximate a function is exponentially larger than the number of neurons needed by a deep NN for a given accuracy level. [Liang & Srikant, 2017; Yarotsky, 2017]

Loss Functions

- To learn $u : \Omega \to \mathbb{R}$
- Given a dataset $\{(x_i, u(x_i))\}_{i=1}^m$
- Mean Squared Error (MSE) loss:
  - $\mathcal{L}(\boldsymbol{\theta}) = \|u(x) - \mathcal{N}(x;\boldsymbol{\theta})\|_2^2 \approx \frac{1}{m}\sum_{i=1}^{m}(u(x_i) - \mathcal{N}(x_i;\boldsymbol{\theta}))^2$
- In general, let $\{\mathcal{F}_k\}_{k=1}^K$ be a linear/nonlinear operators
  - $\mathcal{L}(\boldsymbol{\theta}) = \sum_{k=1}^{K} \lambda_k \|\mathcal{F}_k[u] - \mathcal{F}_k[\mathcal{N}]\|_2^2$
  - MSE is a special case with $\mathcal{F}$ be the identity
  - PINN loss uses the PDE residual as the operator

Activation Functions

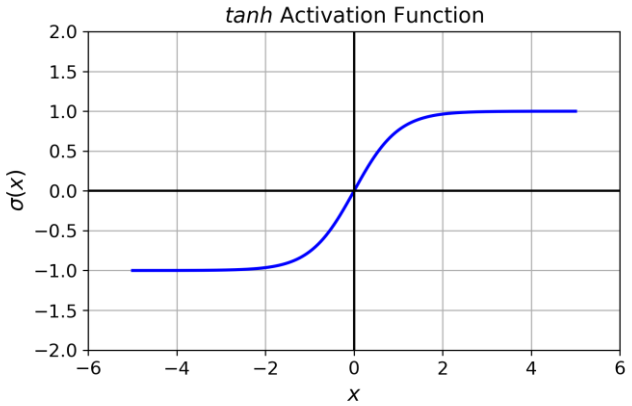
Conventional

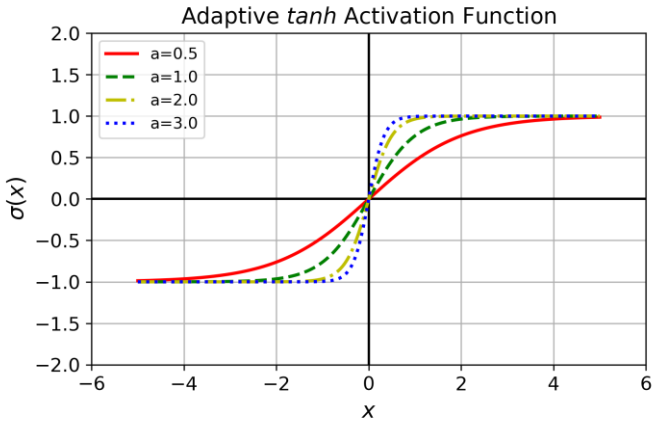
Parameterized*

Tanh

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

$$\sigma'(x) = 1 - \sigma^2(x)$$

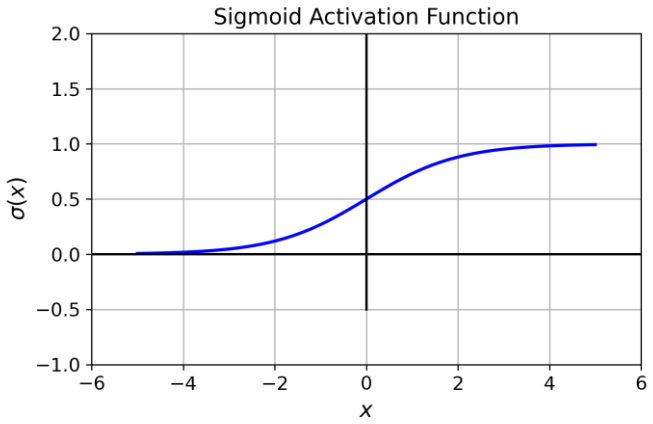


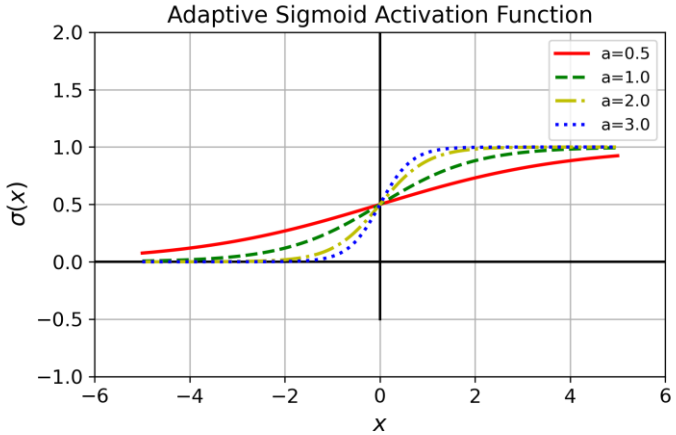


Sigmoid

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

$$\sigma^{'}(x) = \sigma(x)(1 - \sigma(x))$$



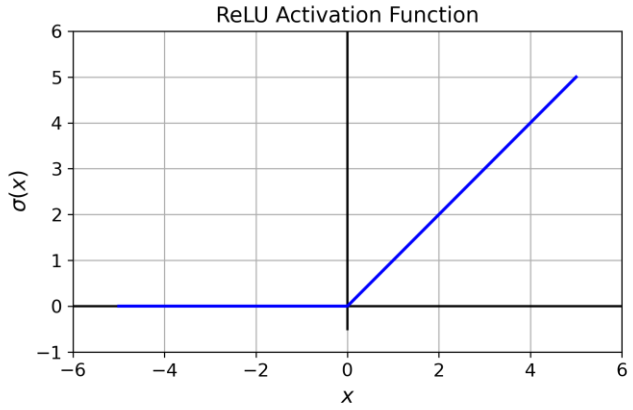


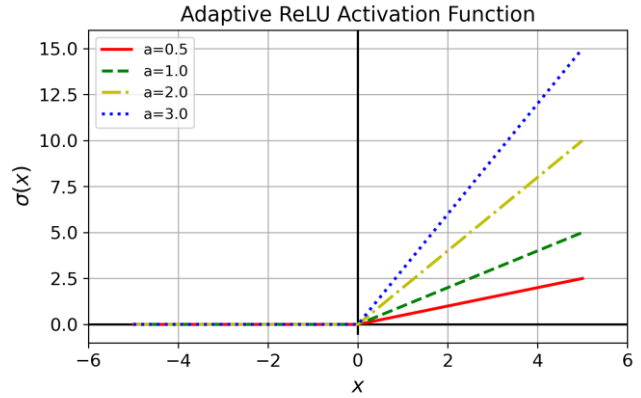
* In parametrization activation function consider $x \rightarrow ax$, where a is the trainable parameter.

ReLU

$$\sigma(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$



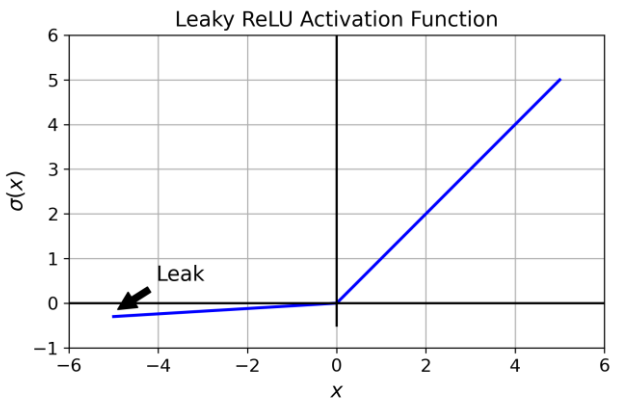


Leaky ReLU

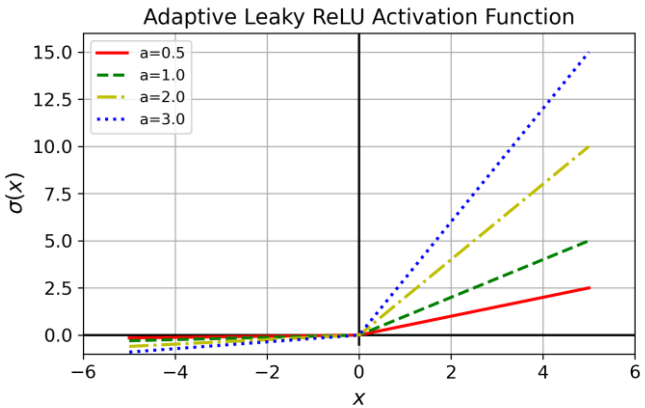
$$\sigma(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Conventional



Parameterized

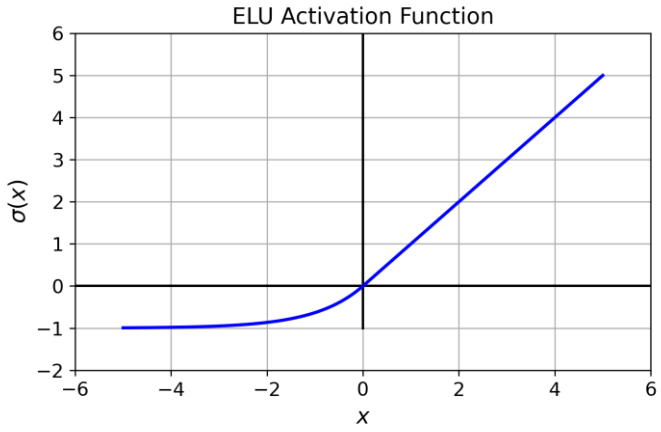


ELU

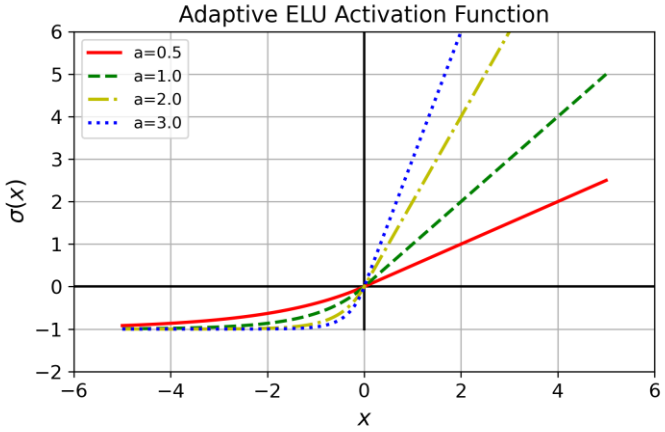
$$\sigma(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$$

Conventional



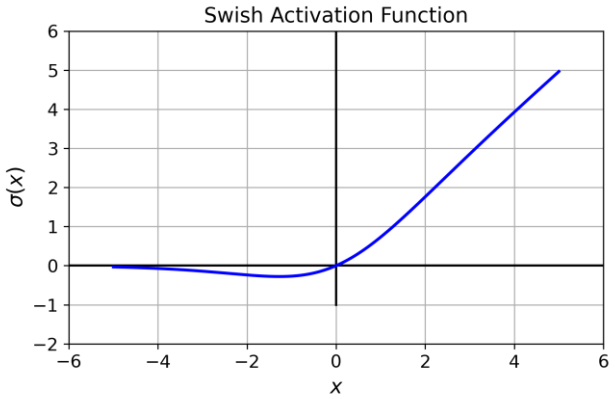
Parameterized

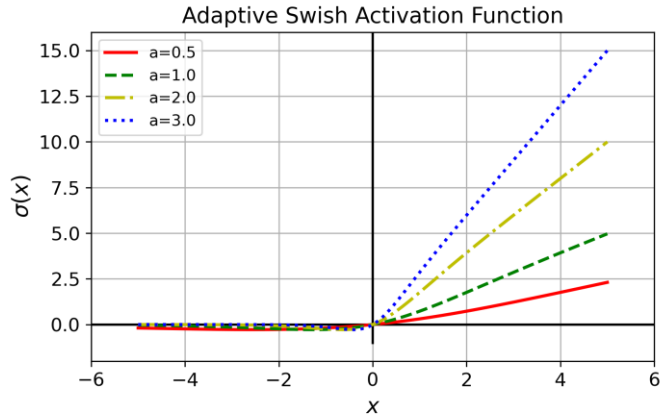


Swish

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

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





Differentiation: Four ways but only one counts: Automatic Differentiation (AD)

$$f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

- Hand-coded analytical derivative
  - Lots of human labor
  - Error prone
- Numerical approximations, e.g., finite difference $\frac{\partial f}{\partial x_i} \approx \frac{f(x+\Delta x_i)-f(x)}{\Delta x_i}$
  - Two function evaluations (forward pass) per partial derivative
  - Truncation errors
- Symbolic differentiation (used in software programs such as Mathematica, Maxima, Maple, and Python library SymPy)
  - Chain rule
  - Expression swell: Easily produce exponentially large symbolic representations
- Automatic differentiation (AD; also called algorithmic differentiation)
  - Symbolic differentiation simplified by numerical evaluation of intermediate sub-expressions
  - Does not provide a general analytical expression for the derivative
  - But only the value of the derivative for a specific input x

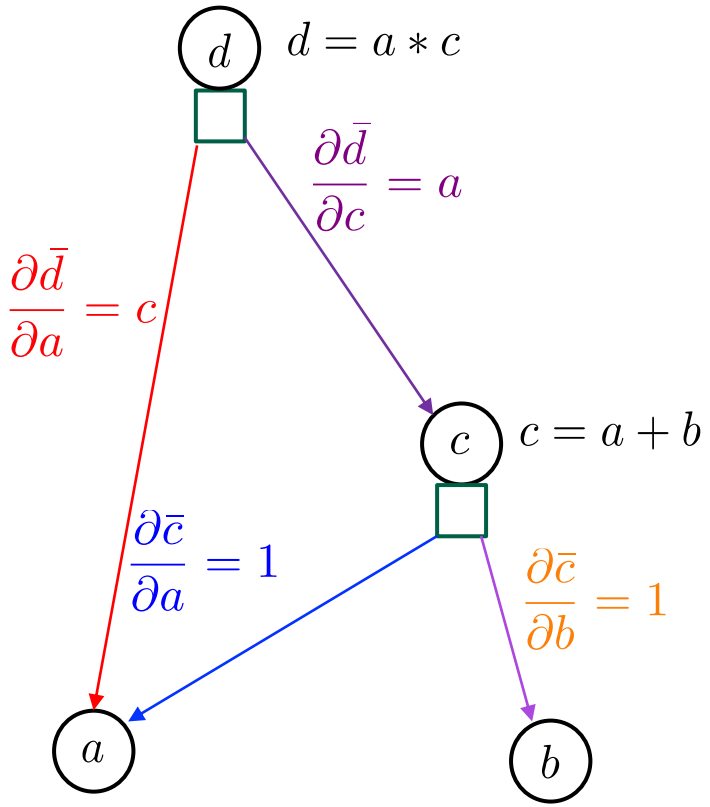
Automatic Differentiation

- Exploits the fact that all computations are compositions of a small set of elementary expressions with known derivatives
- Employs the chain rule to combine these elementary derivatives of the constituent expressions.

- Two ways to compute first-order derivative:
  - Forward mode AD (details not discussed)
    - Cost scales linearly w.r.t. the input dimension
    - Cost is constant w.r.t. the output dimension
  - Reverse mode AD
    - Cost is constant w.r.t. the input dimension
    - Cost scales linearly w.r.t. the output dimension

- In deep learning, **backpropagation** == Reverse mode AD
  - The input dimension of the loss function is # of parameters, e.g., millions
  - The output dimension is 1: the loss value

- High-order derivatives:
  - Nested-derivative approach: Apply first-order AD repeatedly
    - Cost scales exponentially in the order of differentiation
    - What we will use in this class, because the simplicity of implementation
  - More efficient approaches, such as Taylor-mode AD (high-order chain rule)
    - Not supported in TensorFlow/PyTorch yet

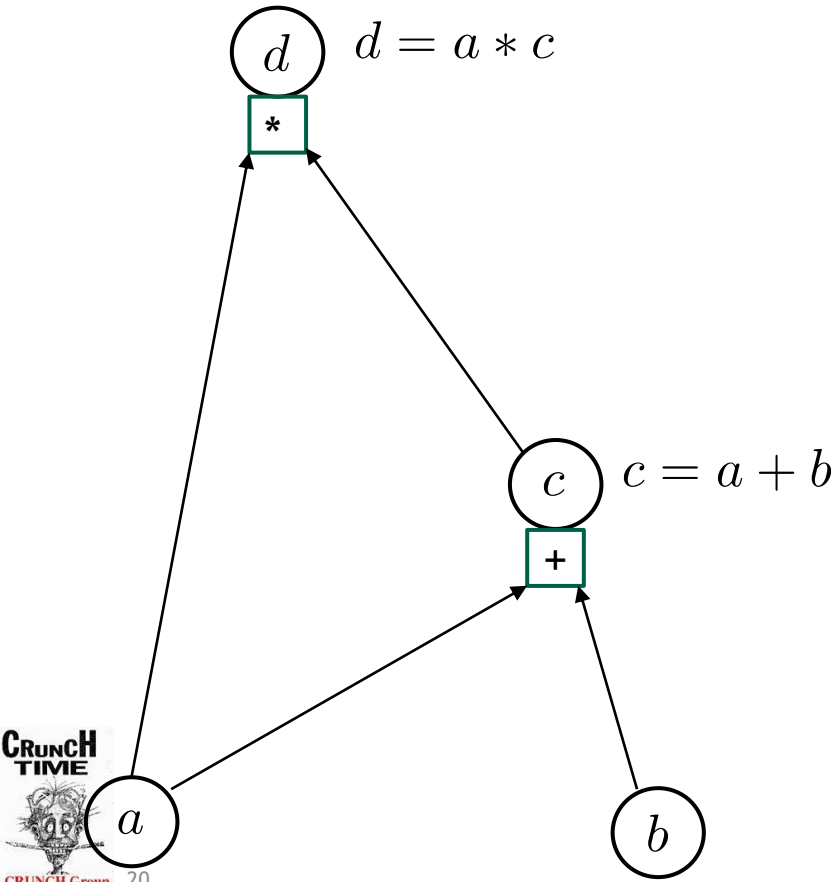


Backpropagation

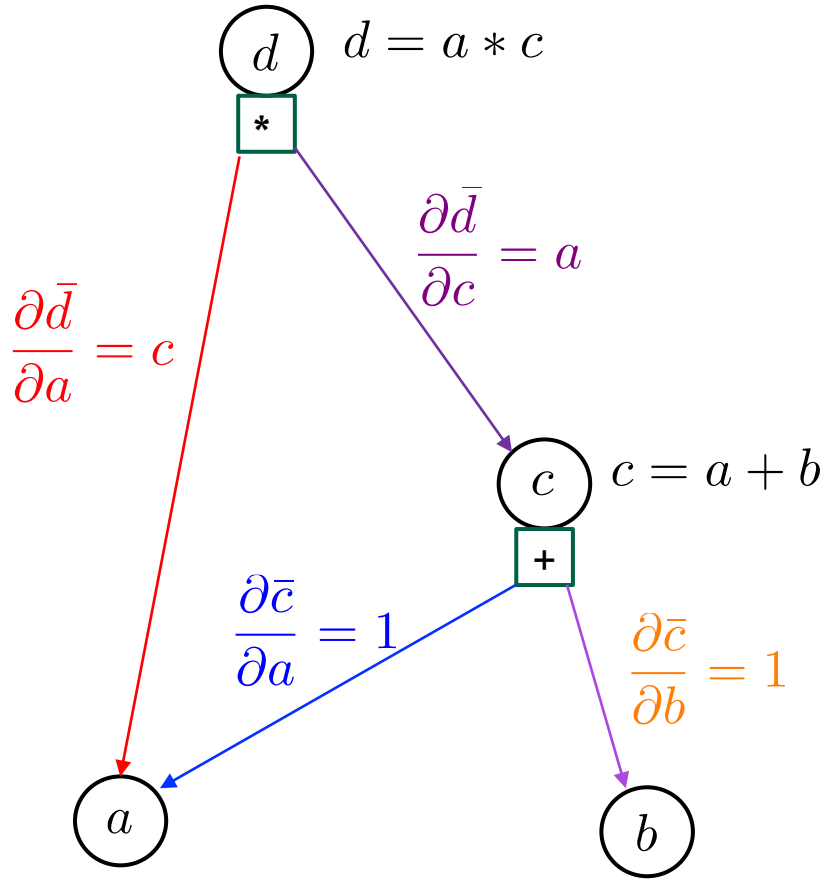
- We apply recursively the Chain rule to implement Backprop
- Use computational graphs to accomplish backprop

- Example: $d = a * (a + b)$

Forward pass



Backward pass



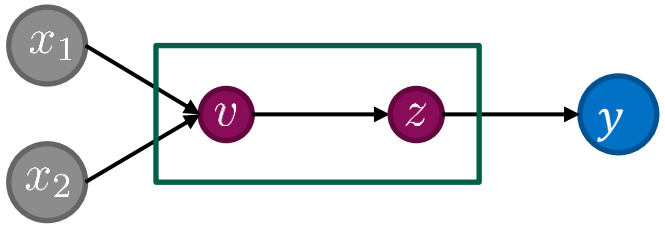
By chain rule:

$$\frac{\partial d}{\partial a} = \frac{\partial \bar{d}}{\partial a} + \frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}$$

$$= c + a$$

$$\frac{\partial d}{\partial b} = \frac{\partial d}{\partial c} * \frac{\partial c}{\partial b} = a * 1 = a$$

Backpropagation



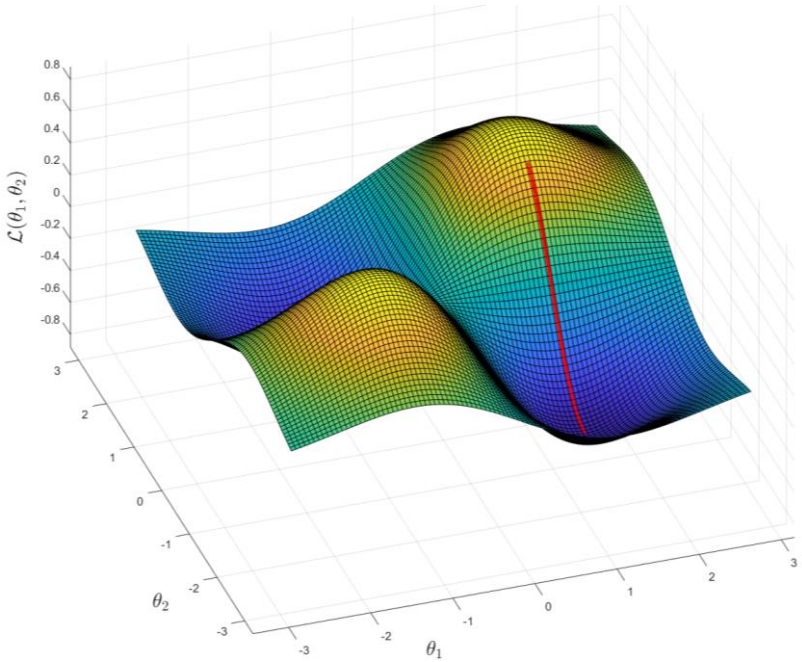
Forward Pass	Backward Pass
$x_1 = 2$ $x_2 = 1$	$\frac{\partial y}{\partial y} = 1$
$v = -2x_1 + 3x_2 + 0.5 = -0.5$ $z = tanh(v) \approx -0.462$	$\frac{\partial y}{\partial z} = \frac{\partial(2z-1)}{\partial z} = 2$ $\frac{\partial y}{\partial v} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial v} = \frac{\partial y}{\partial z} sech^2(v) \approx 1.573$
$y = 2h - 1$	$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v}\frac{\partial v}{\partial x_1} = \frac{\partial y}{\partial v} \times (-2) = -3.146$ $\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial v}\frac{\partial v}{\partial x_2} = \frac{\partial y}{\partial v} \times 3 = 4.719$

Lu et al. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 2021.

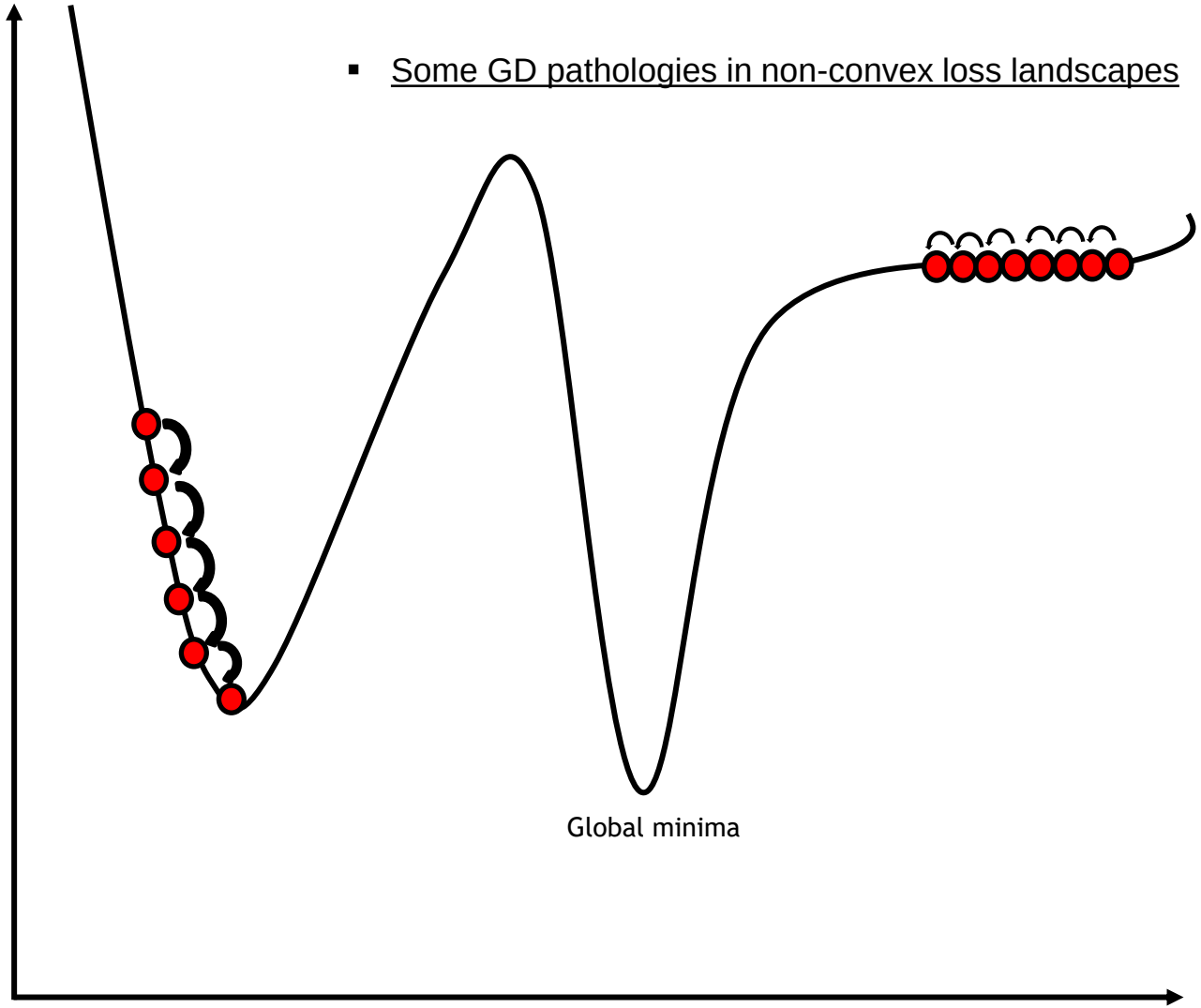
Gradient Descent (GD)

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$



- Some GD pathologies in non-convex loss landscapes

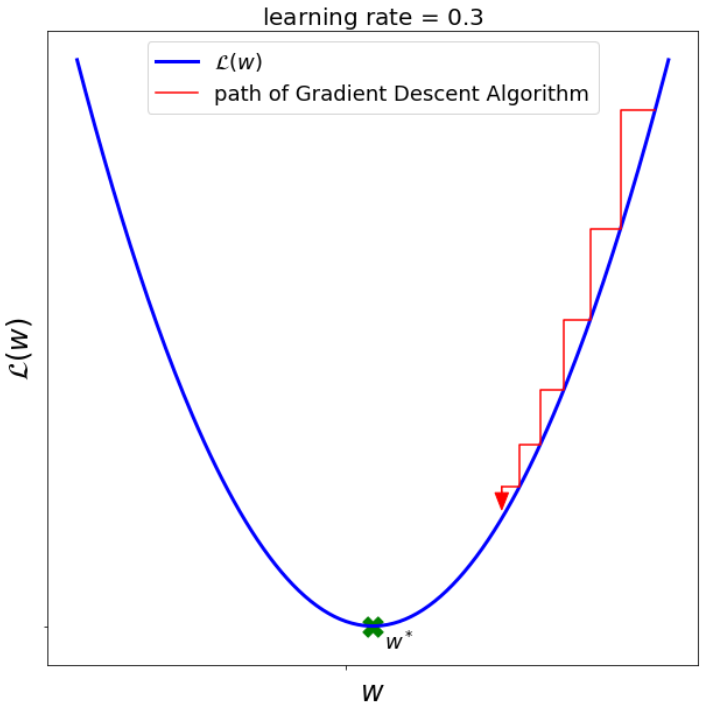


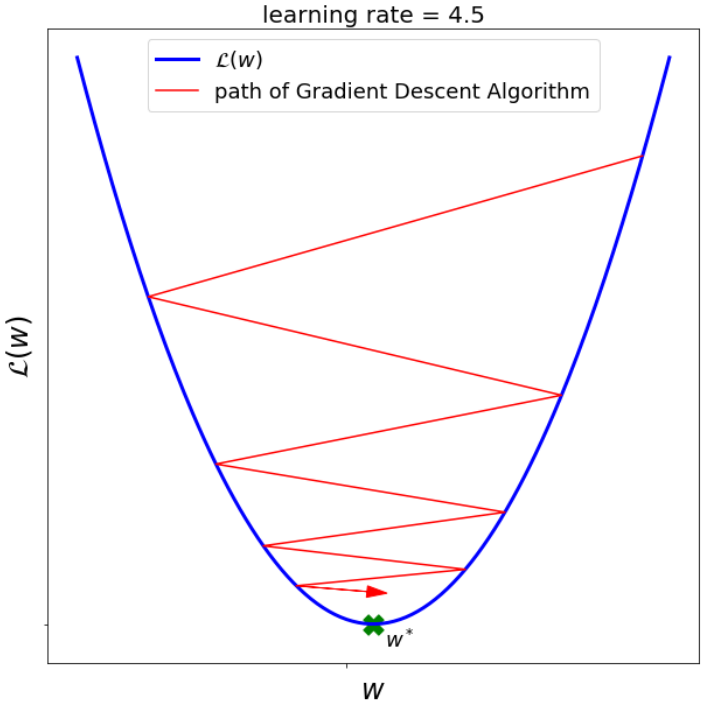
`tf.keras.optimizers.SGD(learning_rate=0.01)`

`torch.optim.SGD(params, lr=0.01)`

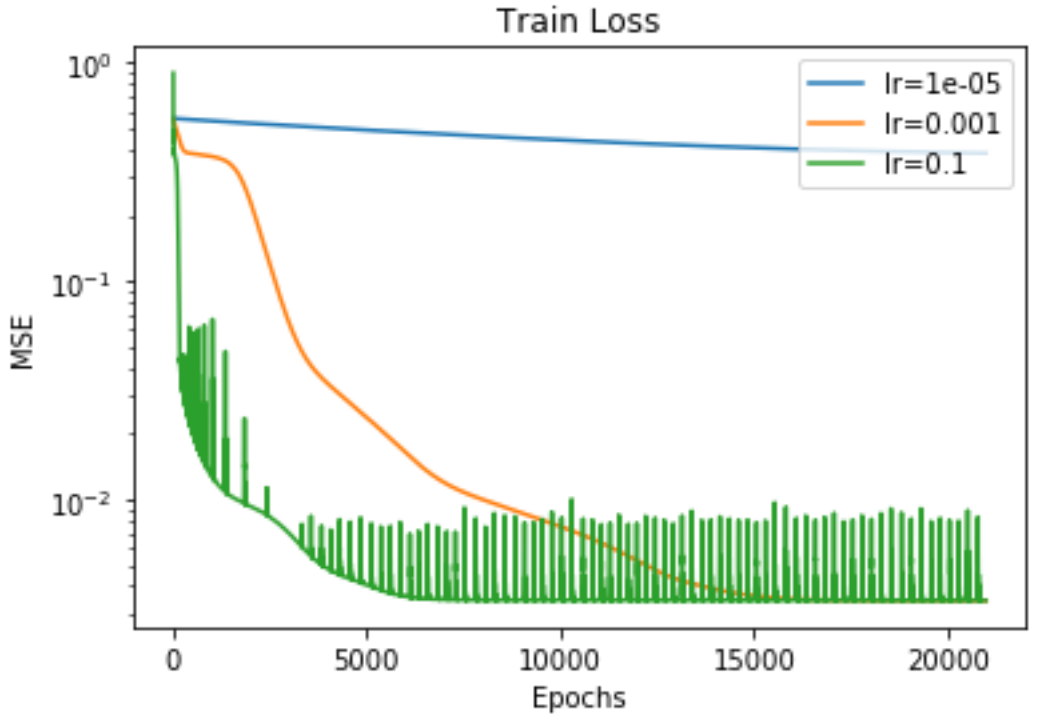
Effect of Learning Rate

❑In linear regression we have convexity (hence global minimum) but still we should scale all features for faster convergence





Loss plot associated with learning the sine function with noise using a fully connected neural network



Learning rate is too small

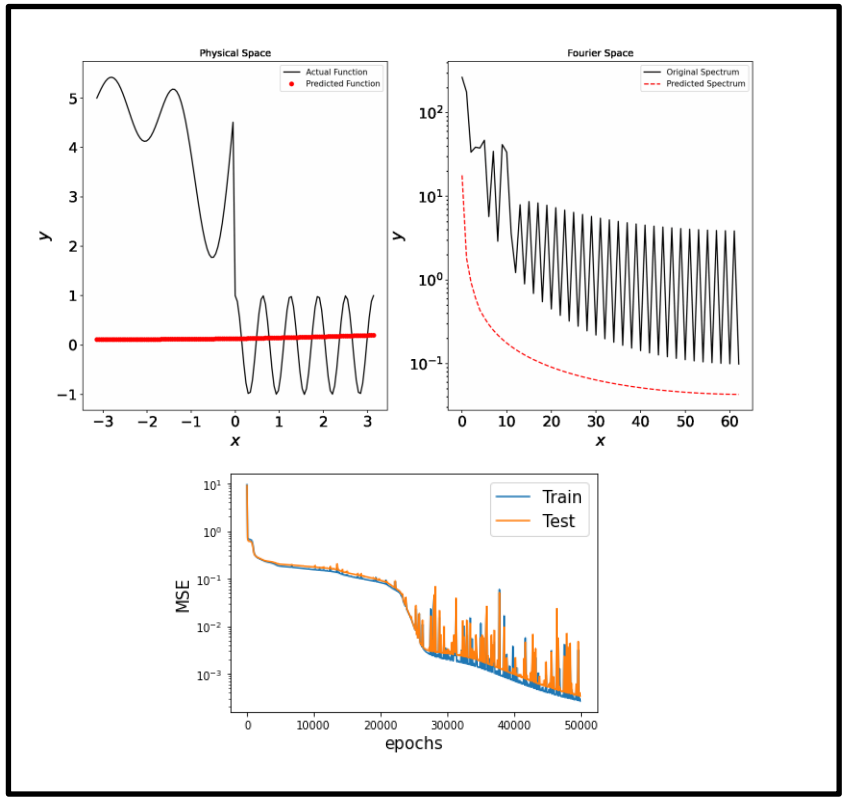
Learning rate is too large

Convergence depends strongly on the lr

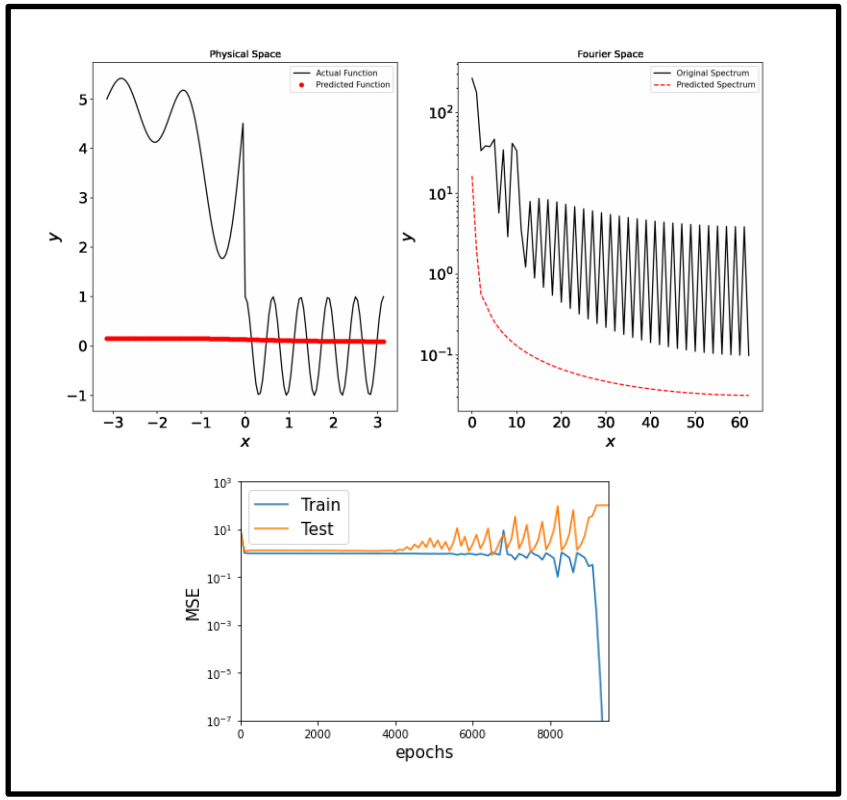
❑ **An effective strategy is to use a variable/decaying learning rate**

Underfitting vs. Overfitting

- Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set (low-capacity models).
- Overfitting occurs when the gap between the training error and test error is too high (high-capacity models).



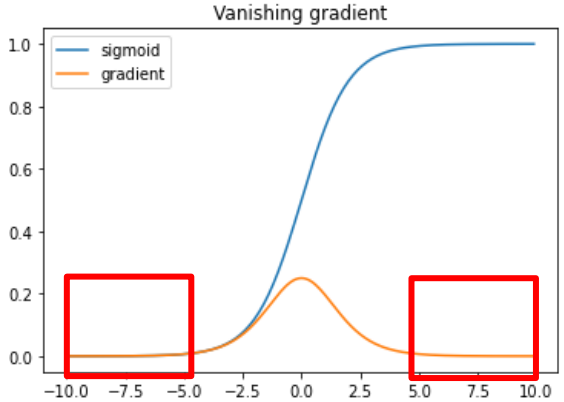
Overfitting



- The neural network is forced to overfit by considering only 10 training points
- The predicted function passes through the 10 training points, making training loss ~ 10^-7
- Model fails to learn the underlying function

Vanishing and Exploding Gradients

- Different layers may learn at hugely different rates: for most NN architectures the gradients become smaller and smaller in back propagation, hence leaving the weights of lower layers unaffected (vanishing gradients). In recurrent NN in addition the weights may explode.
- **Exploding gradient: multiplying** 100 Gaussian random matrices



(All linear layers)

```
A single matrix
 tensor([[ 0.7948, -0.5345,  2.2011, -1.2147, -0.8642],
         [-1.0132, -0.3668,  1.3064, -0.3739,  0.3137],
         [-0.7110,  0.0634, -3.0735, -0.8933, -0.2504],
         [ 0.2037, -0.6473,  1.2173,  0.6089, -1.1243],
         [ 0.7627,  0.8086, -0.9196, -1.1723,  0.4238]])
After multiplying 100 matrices
 tensor([[-5.9421e+28,  2.5019e+28, -4.7395e+27,  9.2365e+28, -9.0353e+28],
         [-4.1826e+28,  1.7611e+28, -3.3361e+27,  6.5015e+28, -6.3598e+28],
         [ 8.4344e+28, -3.5513e+28,  6.7274e+27, -1.3111e+29,  1.2825e+29],
         [-2.4234e+28,  1.0204e+28, -1.9328e+27,  3.7670e+28, -3.6850e+28],
         [ 2.1614e+28, -9.1011e+27,  1.7237e+27, -3.3597e+28,  3.2865e+28]])
```

- This was the main obstacle in training DNNs until the early 2000s.
- 2010: Breakthrough paper of Xavier Glorot & Youshua Bengio *"Understanding the difficulty of training Deep neural nerworks"*, Proc 13th Int. Conf. on AI and Statistics pp. 249-256.
- The main reasons were the then popular sigmoid activation function and the normal distribution of initialized weights $\mathcal{N}(0,1)$. The variance of each layer from input to output increases monotonically and then the activation function saturates at 0 and 1 in the deep layers. Note that the mean of this activation function is 0.5.

Xavier and He - Weight Initializations

- Variance of output of each layer = Variance of inputs to that layer.
- Gradients should have equal variance before and after flowing through a layer in the reverse direction (fan-in/fan-out) – this led to *Xavier (or Glorot) initializations.*

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{1}{\text{fan}_{\text{avg}}}}\right)$$

$$\text{fan}_{\text{avg}} = 0.5(\text{fan}_{\text{in}} + \text{fan}_{\text{out}})$$

- He Normal
- *He initialization* is similar with ReLU $\mathcal{N}(0, \sqrt{\frac{2}{\text{fan}_{\text{in}}}})$

Glorot Initialization
`torch.nn.init.xavier_normal_(w)`

Example:
w = torch.empty(5, 5)
nn.init.xavier_normal_(w)

He Initialization
`torch.nn.init.kaiming_normal_(w)`

Example:
w = torch.empty(5, 5)
torch.nn.init. kaiming_normal_(w)

Glorot Initialization
`tf.initializers.GlorotNormal()`

Example:
init = tf.initializers.GlorotNormal()
w = initializer(shape=(4, 4))

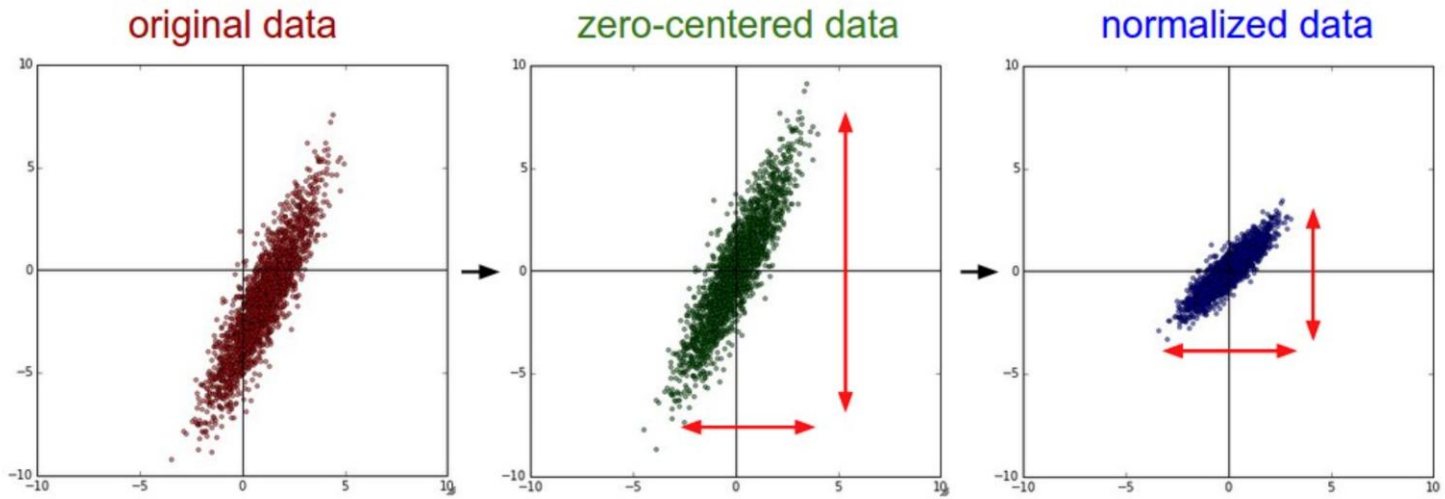
He Initialization
`tf.initializers.HeNormal()`

Example:
init = tf.initializers.HeNormal()
w = initializer(shape=(4, 4))

"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", 2015, IEEE Conference on Computer Vision, pp. 1026-1034

Data Normalization

Ref: https://zaffnet.github.io/batch-normalization



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

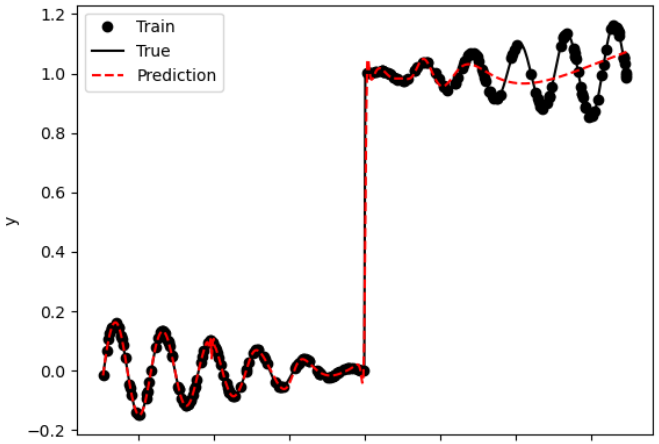
In 1998, Yan LeCun in his paper, Effiecient BackProp, highlighted the importance of normalizing the inputs. Preprocessing of the inputs using normalization is a standard machine learning procedure and is known to help in faster convergence. Normalization is done to achieve the following objectives:

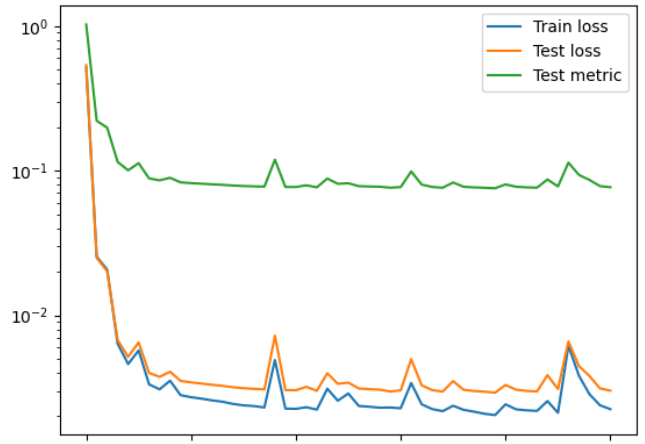
- ❑ The average of each input variable (or feature) over the training set is close to zero (Mean subtraction).
- ❑ Covariances of the features are same (Scaling).
- ❑ Decorrelate the features (Whitening – not required for CNNs).

Data Normalization - Example

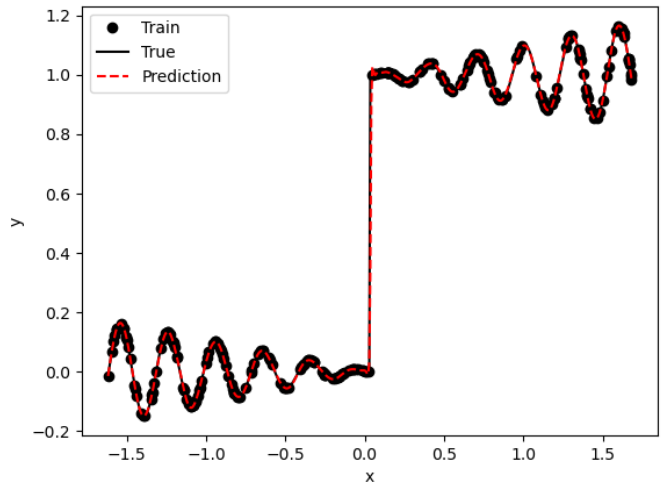
Without data-normalization

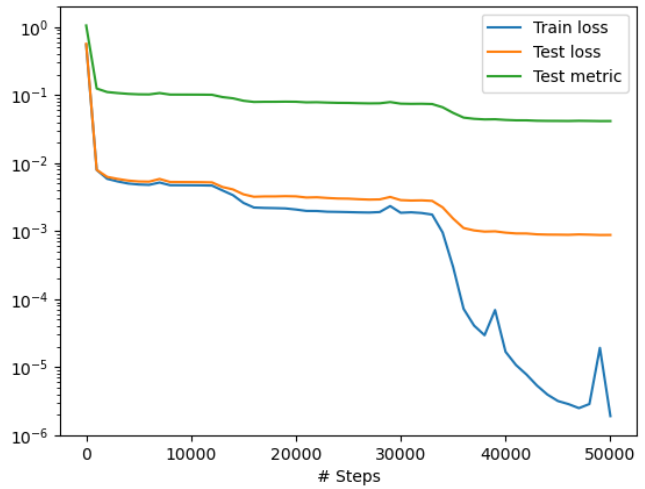
$$y = \begin{cases} \frac{1}{10} x sin(20x) & \text{if } x < 0 \\ 0.5 & \text{if } x = 0.5 \\ 1 + \frac{1}{10} x sin(20x) & \text{if } x > 0 \end{cases}$$





With data-normalization

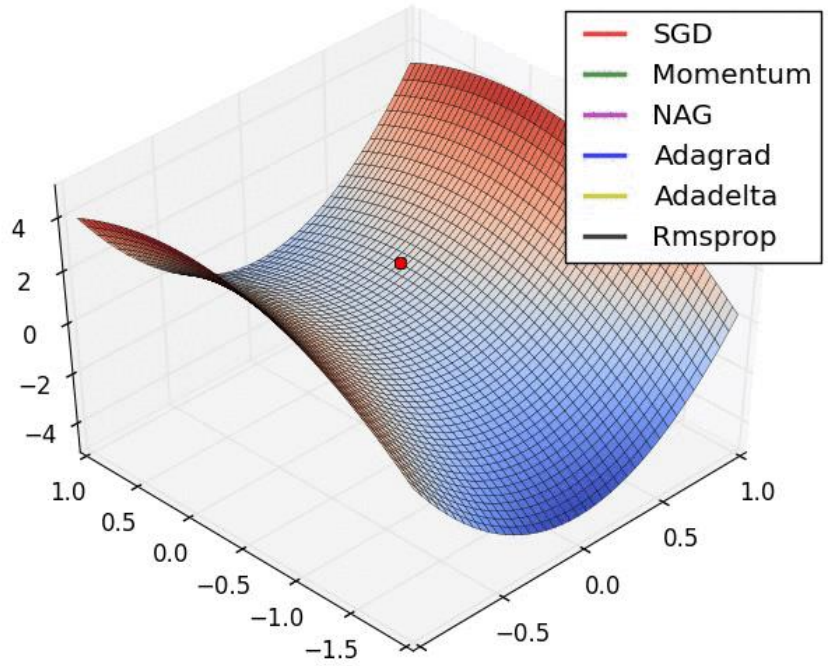




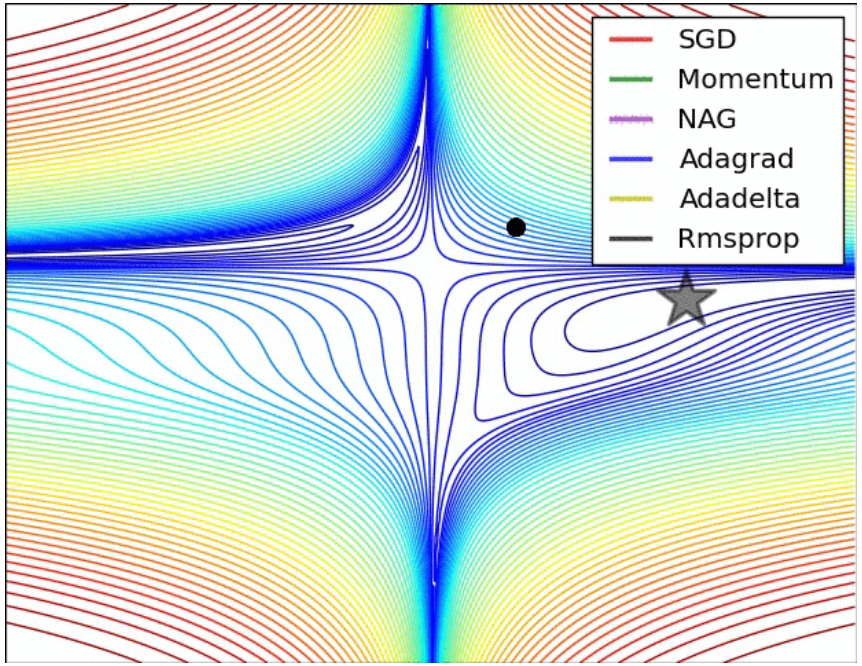
An Overview of Gradient Descent Optimization Algorithms

https://arxiv.org/abs/1609.04747

- This post explores how many of the most popular gradient-based optimization algorithms actually work.



A. This movie shows the **behaviour of the algorithms at a saddle point** Notice that SGD, Momentum, and NAG find it difficulty to break symmetry, although the two latter eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope.



B. In this movie, we see their **behavior on the contours of a loss surface (the Beale function) over time**. Note that Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast, while Momentum and NAG are led off-track, evoking the image of a ball rolling down the hill. NAG, however, is quickly able to correct its course due to its increased responsiveness by looking ahead and heads to the minimum.

- These two animations (Images credit: Alec Radford) provide some intuitions towards the optimization behavior of most of the presented optimization methods.

What Optimizer to Use?

Adam Optimizer: Adaptive moment based optimizer

- Adam = *adaptive moment estimation* is a hybrid method and combines the ideas of momentum optimization and RMSProp.
- Similar to momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients.
- Steps 1, 2, and 5 in algorithm (below) reveal Adam's close similarity to both momentum optimization and RMSProp.

1. $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^{\top}}$
4. $\widehat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^{\top}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\boldsymbol{m}} \oslash \sqrt{\widehat{\boldsymbol{s}} + \epsilon}$

$\top$ denotes iteration number starting at 1.

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
opt= opt_class.Adam(model.parameters(),lr=0.01,betas=(0.9,0.999))
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.Adam(lr=0.001, beta_1=0.9,
beta_2=0.999)
```

What Optimizer to Use?

Adam Optimizer: Adaptive moment based optimizer

- Steps 3 and 4 can be explained as follows: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

- The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. The smoothing term ε is usually initialized to a small number such as 10^{-7}.

- Since Adam is an adaptive learning rate algorithm, it requires less tuning of the learning rate hyperparameter η. We can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

1. $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^T}$
4. $\widehat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^T}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\boldsymbol{m}} \oslash \sqrt{\widehat{\boldsymbol{s}} + \epsilon}$

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
opt= opt_class.Adam(model.parameters(),lr=0.01,betas=(0.9,0.999))
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.Adam(lr=0.001, beta_1=0.9,
beta_2=0.999)
```

What Optimizer to Use?

- Liu DC, Nocedal J. On the limited memory BFGS method for large scale optimization. Mathematical programming. 1989 Aug;45(1):503-28.

L-BFGS optimizer

❑ BFGS is the most popular of all Quasi-Newton methods and have storage complexity.

❑L-BFGS (Limited memory BFGS), which does not require to explicitly to store H^{-1} but instead stores the previous data $\{(x_i, \nabla f(x_i))\}_{i=1}^{k}$ and manages to compute $d = H^{-1}\nabla f(x)$ directly from this data. L-BFGS has storage complexity of $\mathcal{O}(n)$.

❑ L-BFGS implementation is not straightforward in PyTorch and TF2. A detailed implementation will be discussed in Lecture 4, but here we provide a simple API for both.

Algorithm 1: BFGS Algorithm

Input : initial $\boldsymbol{x}_0 \in \mathbb{R}^n$, functions $f(\boldsymbol{x})$, $\nabla f(\boldsymbol{x})$. tolerance θ
Output: x

initialize $\boldsymbol{H}^{-1} \leftarrow \mathbb{I}$
do
  compute $\boldsymbol{s} = \boldsymbol{H}^{-1}\nabla f(\boldsymbol{x})$
  perform a line search $\min_\alpha f(x + \alpha d)$
  $\boldsymbol{s} \leftarrow \alpha \boldsymbol{d}$
  $\boldsymbol{y} \leftarrow \nabla f(\boldsymbol{x} + \boldsymbol{s}) - \nabla f(\boldsymbol{x})$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{s}$
  update $\boldsymbol{H}^{-1} \leftarrow \left(\mathbb{I} - \frac{\boldsymbol{y}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}\right)\boldsymbol{H}^{-1}\left(\mathbb{I} - \frac{\boldsymbol{y}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}\right) + \frac{\boldsymbol{s}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}$
while *until* $||s||_\infty < \theta$;

```
torch.optim.LBFGS(params, lr=1, max_iter=
20, max_eval=None, tolerance_grad=1e-
07, tolerance_change=1e-
09, history_size=100, line_search_fn=None
)
```

L-BFGS implementation in TensorFlow is provided through TF Probability package

```
tfp.optimizer.lbfgs_minimize(f,
initial_position=self.get_weights(),
num_correction_pairs=50,
max_iterations=2000)
```

Loss Regularizers L^2

- Regularization strategy is used to reduce test errors for new inputs but may increase the training errors.
- Loss function $\mathcal{L}$ with L^2 regularizer is expressed as

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- After taking the gradient

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, y) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, y)$$

- Single gradient step to update the weights is expressed as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta(\alpha \boldsymbol{w} + \nabla_w \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}])$$

- After rearranging the term in above expression

$$\boldsymbol{w} \leftarrow (1 - \eta\alpha)\boldsymbol{w} - \eta \nabla_w \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

Loss Regularizers: L^1

- The L^1 regularization model parameter $\boldsymbol{w}$ is defined as

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1$$

- Thus, the regularized loss function $\boldsymbol{\theta}$ is expressed as

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha||\boldsymbol{w}1||_1$$

- The gradient of L^1 regularized loss function is expressed as

$$\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{X}, y; \boldsymbol{w})$$

  where $\text{sign}(\boldsymbol{w})$ is sign $\boldsymbol{w}$ applied element-wise.

Collapse of Deep and Narrow ReLU Neural Networks

Training of NNs

- NP-hard [Sima, 2002]
- Local minima [Fukumizu & Amari, 2002]
- Bad saddle points [Kawaguchi, 2016]

ReLU

- Dying ReLU neuron: stuck in the negative side

Deep ReLU nets?

Dying ReLU network

NN is a constant function after initialization

Collapse

NN converges to the "mean" state of the target function during training

L. Lu, Y. Shin, Y. Su, & G. E. Karniadakis. Dying ReLU and initialization: Theory and numerical examples. *Communications in Computational Physics*, 28(5), 1671–1706, 2020.

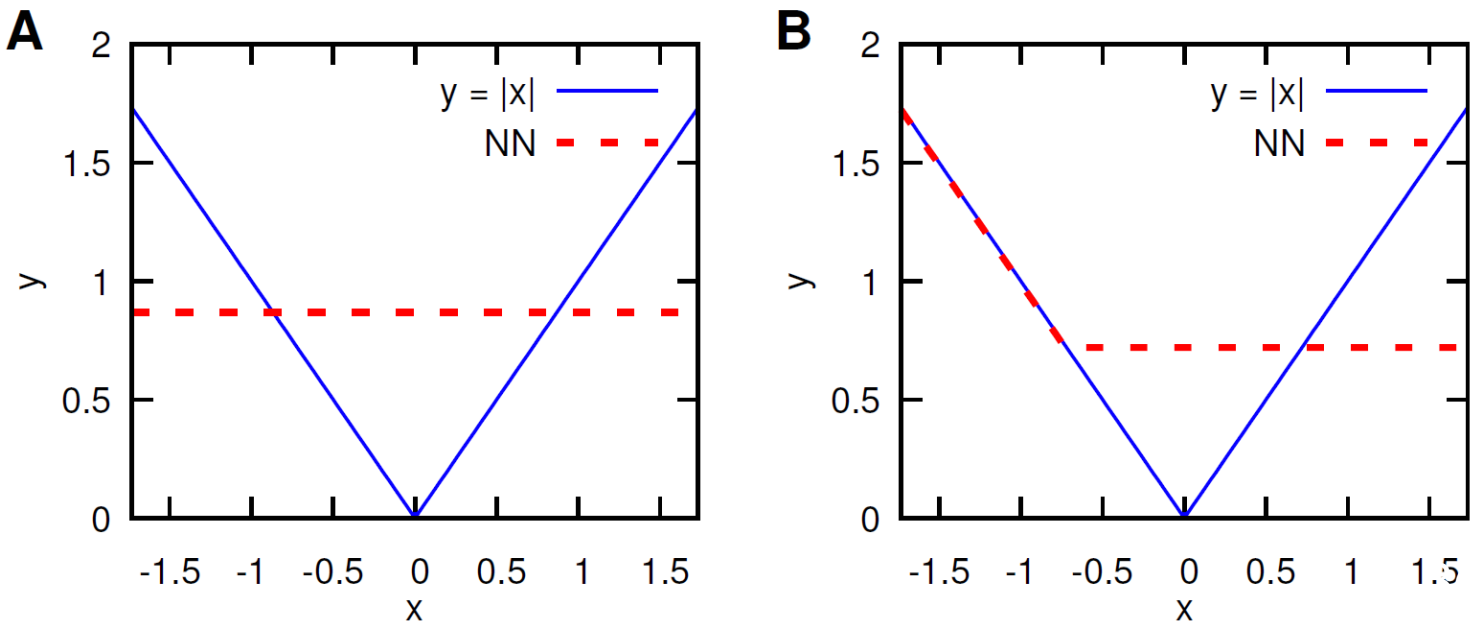
ReLU Collapse: One-Dimensional Examples

$$f(x) = |x|$$

- $|x| = \text{ReLU}(x) + \text{ReLU}(-x) = \begin{bmatrix} 1 & 1 \end{bmatrix} \text{ReLU}(\begin{bmatrix} 1 \\ -1 \end{bmatrix} x)$
- 2-layer with width 2

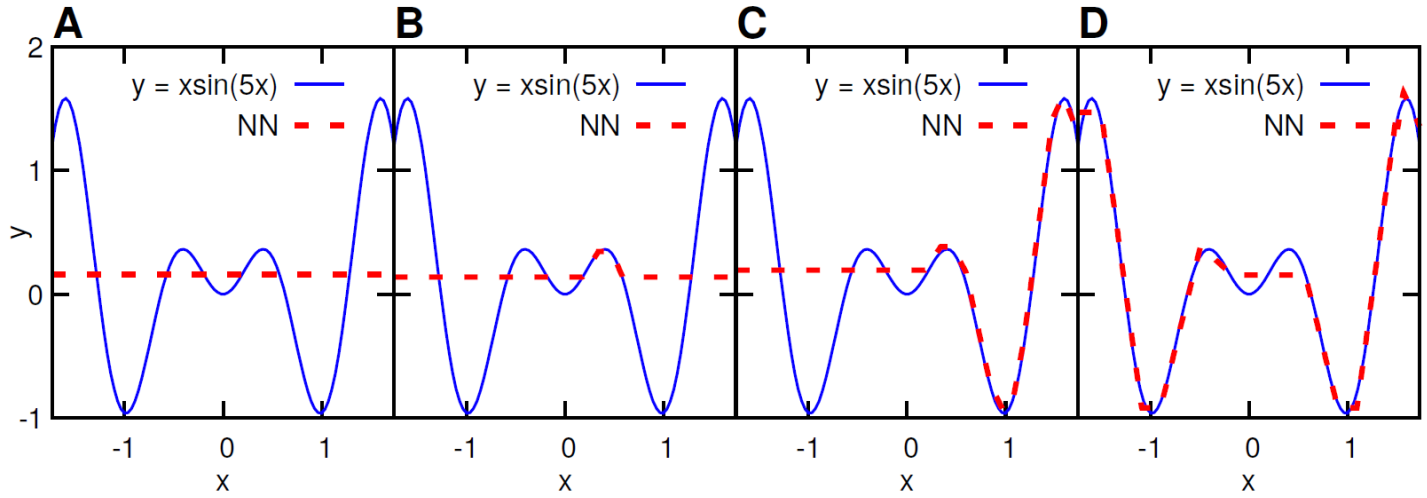
Train a 10-layer ReLU NN with width 2 (MSE loss, whatever optimizer)

- Collapse to the mean value (A): ∼93%
- Collapse partially (B)

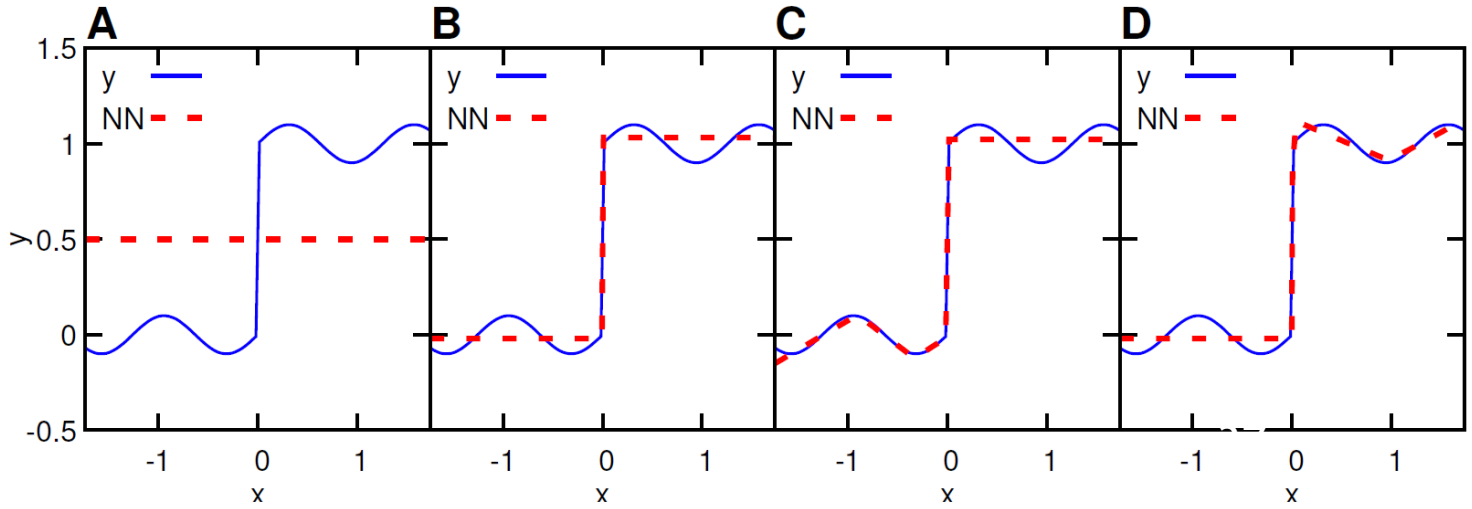


ReLU Collapse: One-Dimensional Examples

$$f(x) = x \sin(5x)$$

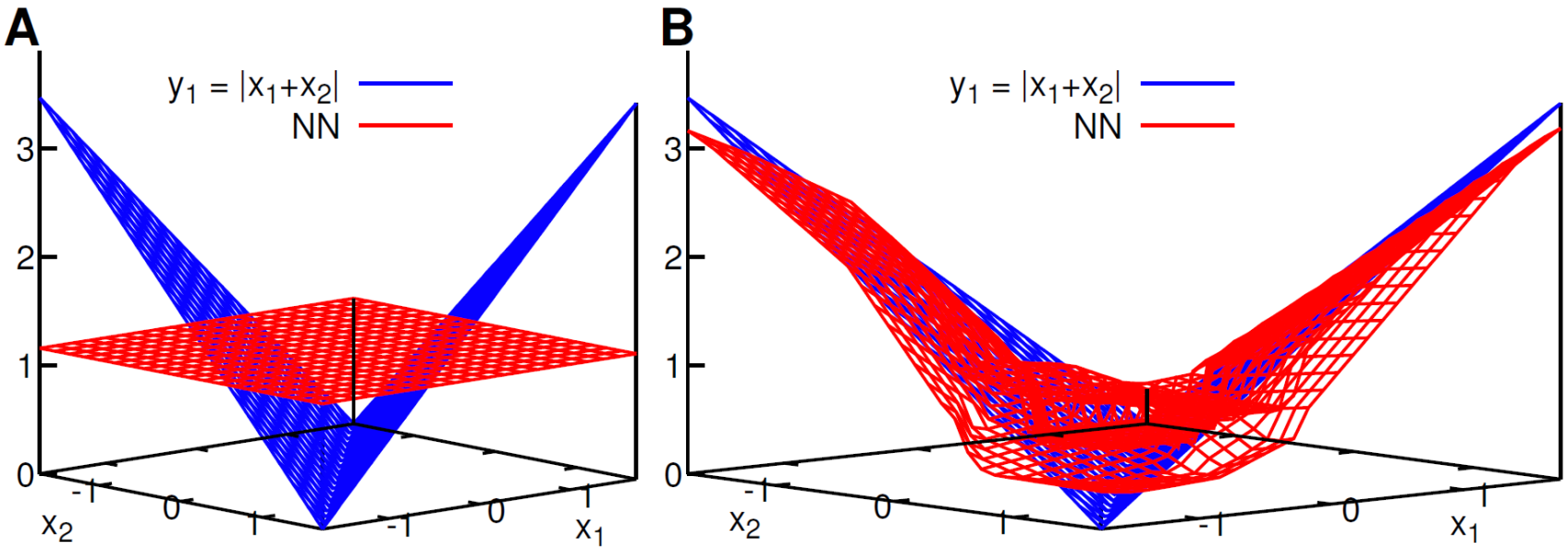


$$f(x) = 1_{\{x>0\}} + 0.2 \sin(5x)$$



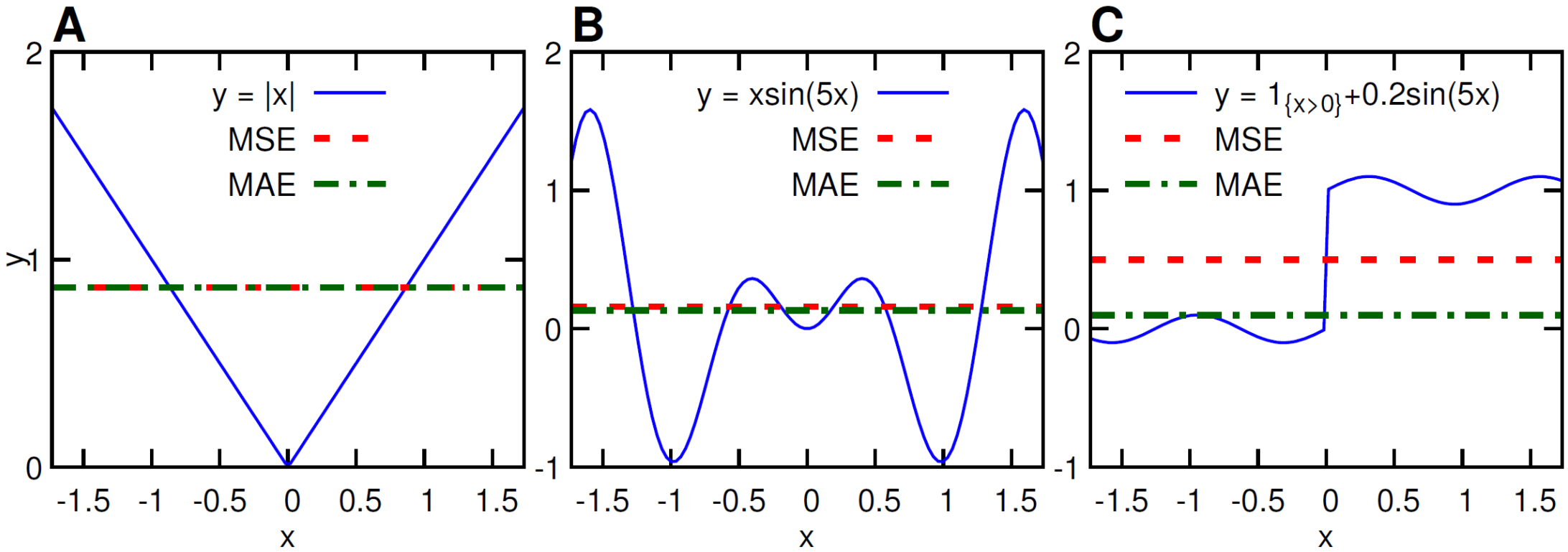
ReLU Collapse: Two-Dimensional Example

$$f(\mathbf{x}) = \begin{bmatrix} |\mathbf{x}_1 + \mathbf{x}_2| \\ |\mathbf{x}_1 - \mathbf{x}_2| \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ & \\ & \\ & 1 & 1 \end{bmatrix} \text{ReLU}\left(\begin{bmatrix} 1 & 1 \\ -1 & -1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{x} \right)$$



Does the Loss Type Matter?

- Mean squared error (MSE) $\Rightarrow$ mean
- Mean absolute error (MAE) $\Rightarrow$ median



Theoretical Analysis - I

$\mathcal{N}^L$ will eventually Die in probability as $L \to \infty$

Theorem 1

Let $\mathcal{N}^L(\mathbf{x})$ be a ReLU NN with L layers, each having $N_1, \cdots, N_L$ neurons. Suppose

1. Weights are independently initialized from a symmetric distribution around 0,
2. Biases are either from a symmetric distribution or set to be zero.

Then

$$P(\mathcal{N}^L(\mathbf{x}) \text{ dies}) \leq 1 - \prod_{\ell=1}^{L-1} (1 - (1/2)^{N_\ell}).$$

Furthermore, assuming $N_\ell = N$ for all ℓ,

$$\lim_{L\to\infty} P(\mathcal{N}^L(\mathbf{x}) \text{ dies}) = 1, \qquad \lim_{N\to\infty} P(\mathcal{N}^L(\mathbf{x}) \text{ dies}) = 0.$$

Proof

Lemma 1

Let $\mathcal{N}^L(\mathbf{x})$ be a ReLU NN of L-layers. Suppose weights are independently from distributions satisfying $P(\boldsymbol{W}_j^\ell \boldsymbol{z} = \mathbf{0}) = 0$ for any nonzero $\boldsymbol{z} \in \mathbb{R}^{N_{\ell-1}}$ and any j-th row of $\boldsymbol{W}^\ell$. Then

$$P(\mathcal{N}^\ell(\mathbf{x}) \text{ dies}) = P(\exists\, \ell \in \{1, \ldots, L-1\} \text{ s.t. } \phi(\mathcal{N}^\ell(\mathbf{x})) = \mathbf{0}\ \forall \mathbf{x} \in \mathcal{D}).$$

- For a given $\mathbf{x}$,

$$P\left(\boldsymbol{W}_s^j \phi(\mathcal{N}^{j-1}(\mathbf{x})) + \boldsymbol{b}_s^j < 0 | \tilde{A}_{j-1,\mathbf{x}}^c\right) = \frac{1}{2},$$

where $\tilde{A}_{\ell,\mathbf{x}}^c = \{\forall 1 \leq j < \ell,\ \phi(\mathcal{N}^j(\mathbf{x})) \neq \mathbf{0}\}$

Dead Networks would Collapse

Theorem 2

Suppose the ReLU NN dies. Then for any loss $\mathcal{L}$, the network is optimized to a constant function by any gradient based method.

Proof

- Lemma 1 $\Rightarrow \exists\, \ell \in \{1, \ldots, L-1\}$ s.t. $\phi(\mathcal{N}^\ell(\mathbf{x})) = \mathbf{0}\ \forall \mathbf{x} \in \mathcal{D}$
- Gradients of $\mathcal{L}$ wrt the weights/biases in the $1, \ldots, l$-th layers vanish

Assuming training data are iid from $P_{\mathcal{D}}$, the optimized network is

$$\mathcal{N}^L(\mathbf{x}; \theta^*) = \underset{\boldsymbol{c} \in \mathbb{R}^{N_L}}{\operatorname{argmin}}\, \mathbb{E}_{\mathbf{x} \sim P_{\mathcal{D}}} \left[\ell(\boldsymbol{c}, f(\mathbf{x}))\right]$$

- MSE/$L^2 \Rightarrow \mathbb{E}[f(\mathbf{x})]$
- MAE/$L^1 \Rightarrow$ median of $f(\mathbf{x})$

Probability of Dying when $d_{in} = 1$

Theorem 3

Let $\mathcal{N}^L(\mathbf{x})$ be a bias-free ReLU NN with $L \geq 2$ layers, each having N neurons at $d_{\text{in}} = 1$. Suppose weights are independently initialized from continuous symmetric distributions around 0. Then

$$1 - \prod_{\ell=1}^{L-1} (1 - (1/2)^N) \geq P(\mathcal{N}^L(x) \text{ dies})$$

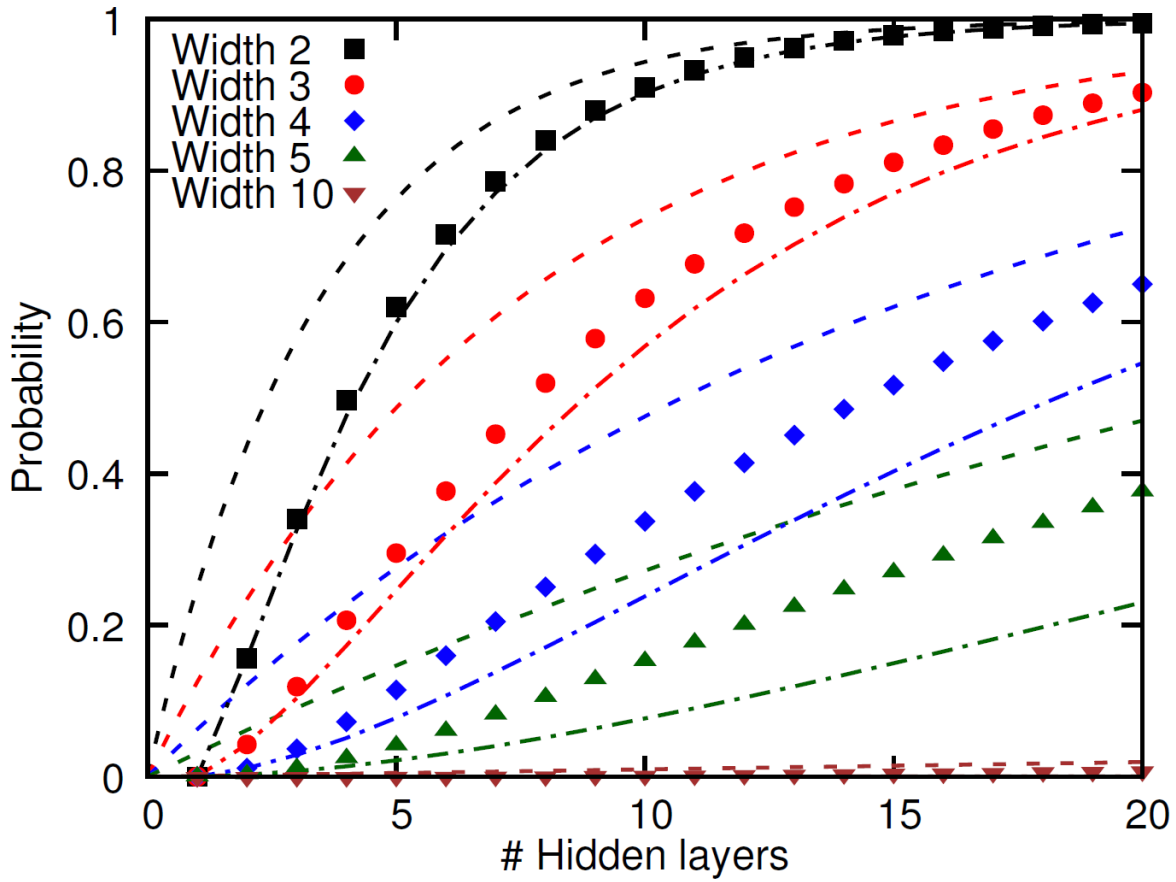
$$\geq 1 - (\mathcal{P}_{22})^{L-2} - \frac{(1 - 2^{-N+1})(1 - 2^{-N})}{1 + (N-1)2^{-N}} ((\mathcal{P}_{22})^{L-2} - (\mathcal{P}_{33})^{L-2})$$

where $\mathcal{P}_{22} = 1 - \frac{1}{2^N}$ and $\mathcal{P}_{33} = 1 - \frac{1}{2^{N-1}} - \frac{N-1}{4^N}$.

Theoretical versus Numerical Results

- A ReLU NN with $d_{in} = 1$
- Weights randomly initialized from symmetric distributions
- Biases are initialized to 0

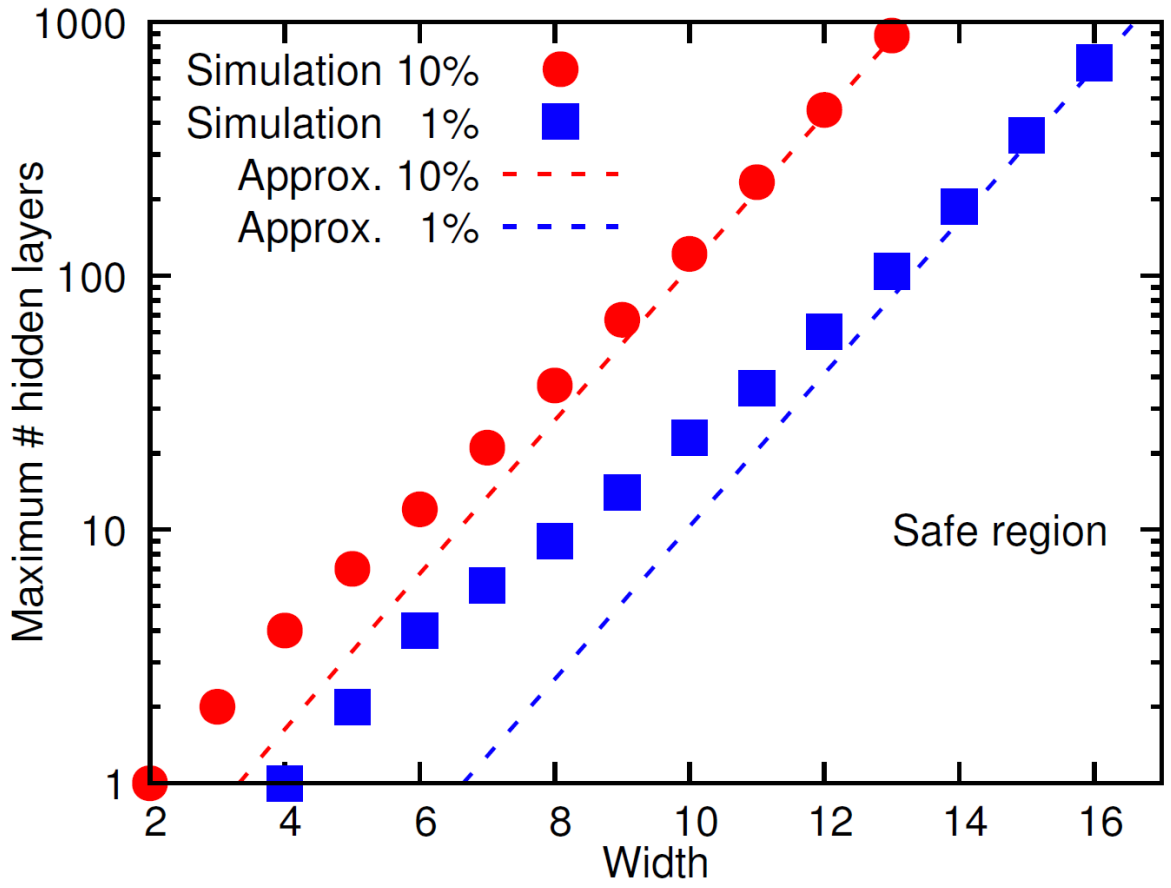
More likely to die when it is deeper and narrower



Theoretical versus Numerical Results

Safe Operating Region for a ReLU NN

Keep the dying probability $< 10\%$ or 1%



Motivation and need for other neural networks

Cause: Diverse data formats and learning tasks
Rescue: Different neural network architectures

Datatype: Tabular data

Breast Cancer Wisconsin (Prognostic) Data Set

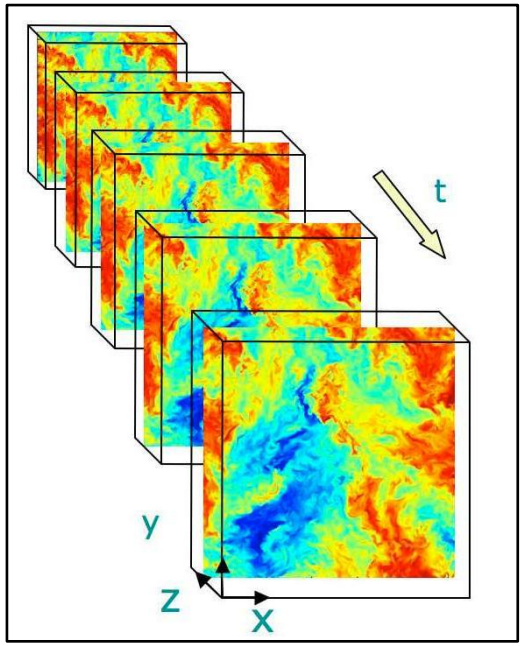
1) ID number
2) Outcome (R = recur, N = nonrecur)
3) Time (recurrence time if field 2 = R, disease-free time if field 2 = N)
4-33) Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)"

UCI Machine Learning Repository

Datatype: Image data

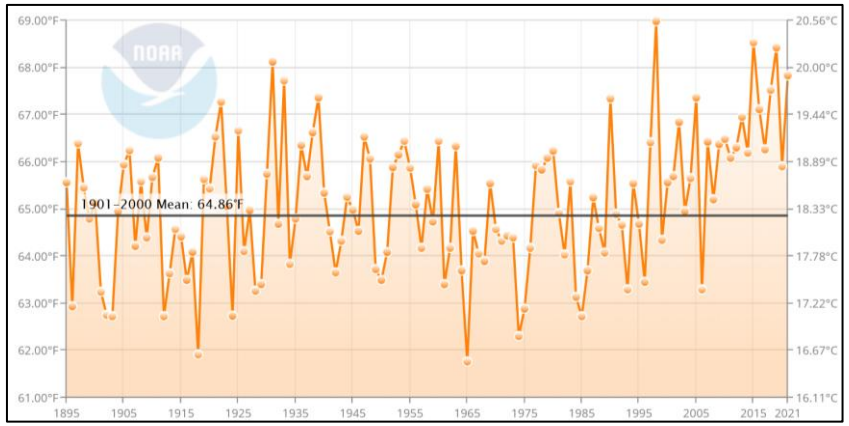
Velocity and pressure for fluid flow



Johns Hopkins Turbulence Databases

Datatype: Time series data

Contiguous U.S. Average Temperature

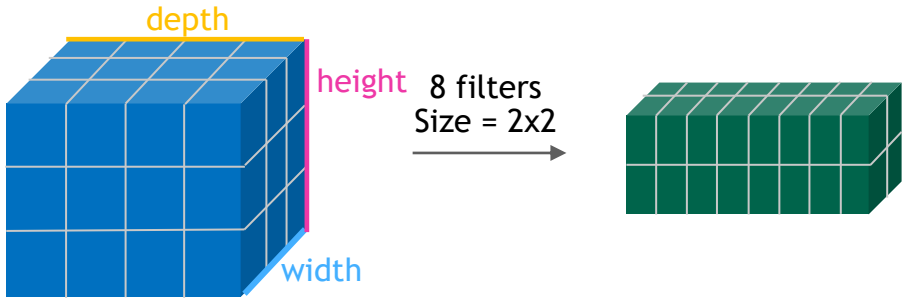


NOAA NATIONAL CENTERS FOR ENVIRONMENTAL INFORMATION
NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION

Convolutional Neural Network (CNN)

- Input: A 3D tensor (width, height, depth); e.g., if the input is an image, the depth is 3, i.e., Red, Green, Blue channels.
- Output: a 3D tensor (width, height, depth)
- A convolutional layer: Convolution between two functions f and g is defined as

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}$$

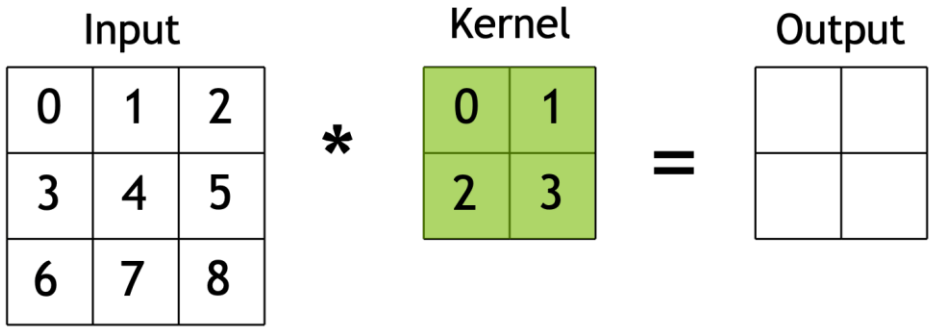


- For 1D discrete case, the integral turns into a sum

$$(f * g)(i) = \sum_a f(a) g(i - a)$$

- For 2D case

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b)$$



- Rewrite above as

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}$$

- For multiple channels:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}$$

FNN vs CNN

FNN

$$T^l(x) = \boldsymbol{W}^l x + \boldsymbol{b}^l$$

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathbf{W}]_{i,j,k,l}[\mathbf{X}]_{k,l}$$

$$= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathbf{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b}$$

CNN

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}$$

Properties of CNN

- Efficiency: Much fewer parameters $(W \times H)^2 \rightarrow K^2$
- Locality:
  1. Nearby pixels are typically related to each other
  2. We should not have to look very far away from location (i, j) to compute $H(i, j)$
- Translation Invariance: A shift in the input $\boldsymbol{X}$ should simply lead to a shift in the hidden representation $\boldsymbol{H}$

CNN kernel vs finite difference stencil

Partial derivative

$$\frac{\partial \phi}{\partial x} = \frac{\phi(x+\delta x) - \phi(x-\delta x)}{2\delta x} + O(\delta x)^2 \quad \longleftrightarrow \quad \frac{\partial \phi}{\partial x} = \begin{bmatrix} -\frac{1}{2\delta x} & 0 & \frac{1}{2\delta x} \\ -\frac{1}{2\delta x} & 0 & \frac{1}{2\delta x} \\ -\frac{1}{2\delta x} & 0 & \frac{1}{2\delta x} \end{bmatrix}$$

2D Laplacian (Δ)

Standard 5-point stencil finite difference kernel

Learned kernel

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_{i,j} + \left.\frac{\partial^2 T}{\partial y^2}\right|_{i,j} \rightarrow \quad \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 0 & -4 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{2,3} \\ k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Resources

- Deep Learning. Ian Goodfellow, Yoshua Bengio, & Aaron Courville. MIT Press, 2016. http://www.deeplearningbook.org
- Dive into Deep Learning. https://d2l.ai
- https://github.com/lululxvi/tutorials