Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

Lecture 3: Deep Neural Networks

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

Course Roadmap

Module-1 (Basics)

- Lecture 1: Introduction
- Lecture 2: A primer on Python, NumPy, SciPy and *jupyter* notebooks
- Lecture 3: Deep Learning Networks
- Lecture 4: A primer on TensorFlow and PyTorch
- Lecture 5: Training and Optimization
- Lecture 6: Neural Network Architectures

Module-2 (PDEs and Operators)

- Lecture 7: Machine Learning using Multi-Fidelity Data
- Lecture 8: Physics-Informed Neural Networks (PINNs)
- Lecture 9: PINN Extensions
- Lecture 10: Neural Operators

Module-3 (Codes & Scalability)

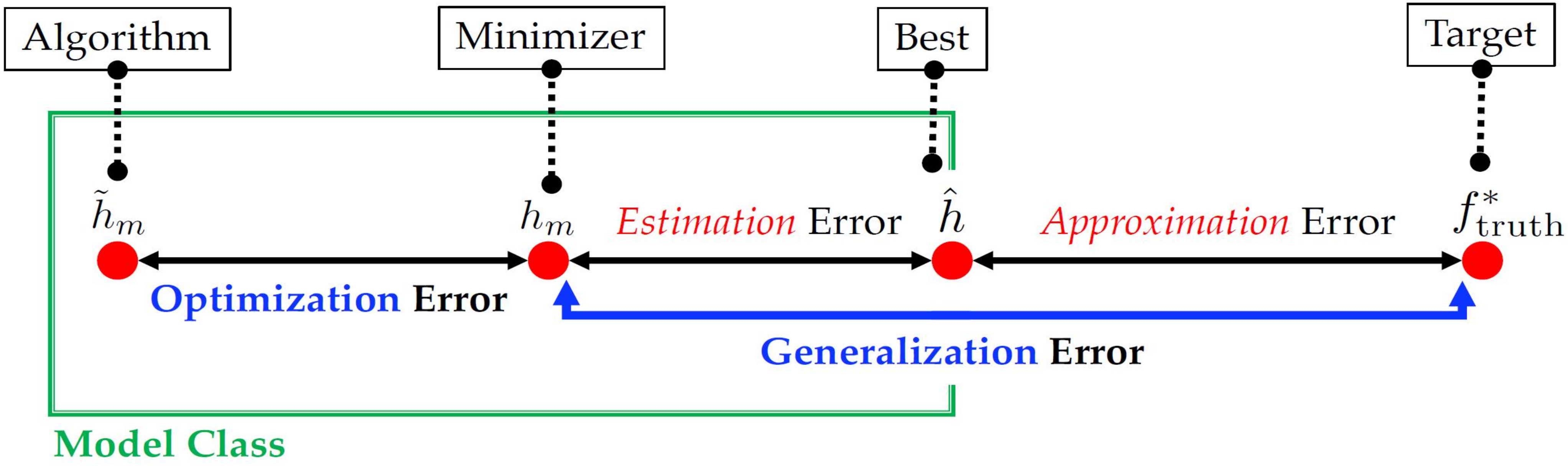
- Lecture 11: Multi-GPU SciML

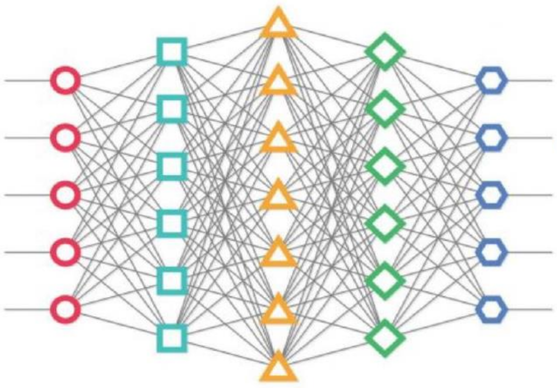
Contents

- Workflow in training a deep neural network
- Basic concepts and terminology
- Regression versus classification
- Universal approximation theorem for functions and functionals
- Example of a regression of a discontinuous/oscillatory function
- Fundamental approximation theory for shallow and deep neural networks
- Activation functions and adaptivity
- Loss functions (simple and advanced)
- Forward/backpropagation and automatic differentiation
- Connecting neural networks with finite elements
- Summary
- References

- Main references: Chapter 6 of the book by Goodfellow et al. (2016) and Chapters 4 & 11 of A. Geron (2019)
- See all references in last slide

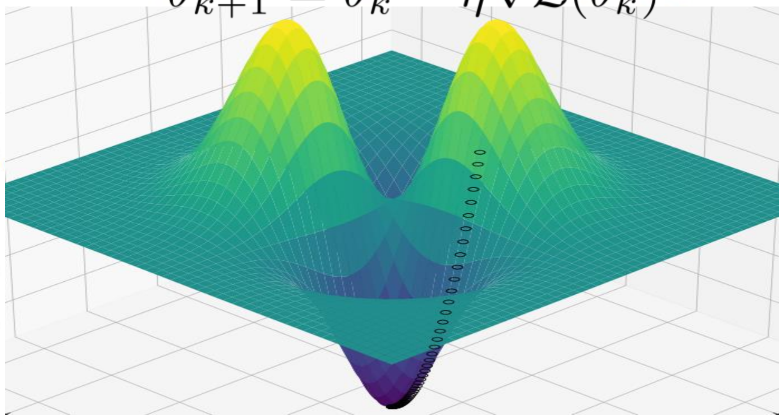
Fundamental Questions



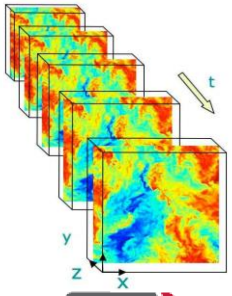


Neural Networks

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



Gradient-based Optimization



UCI Machine Learning Repository

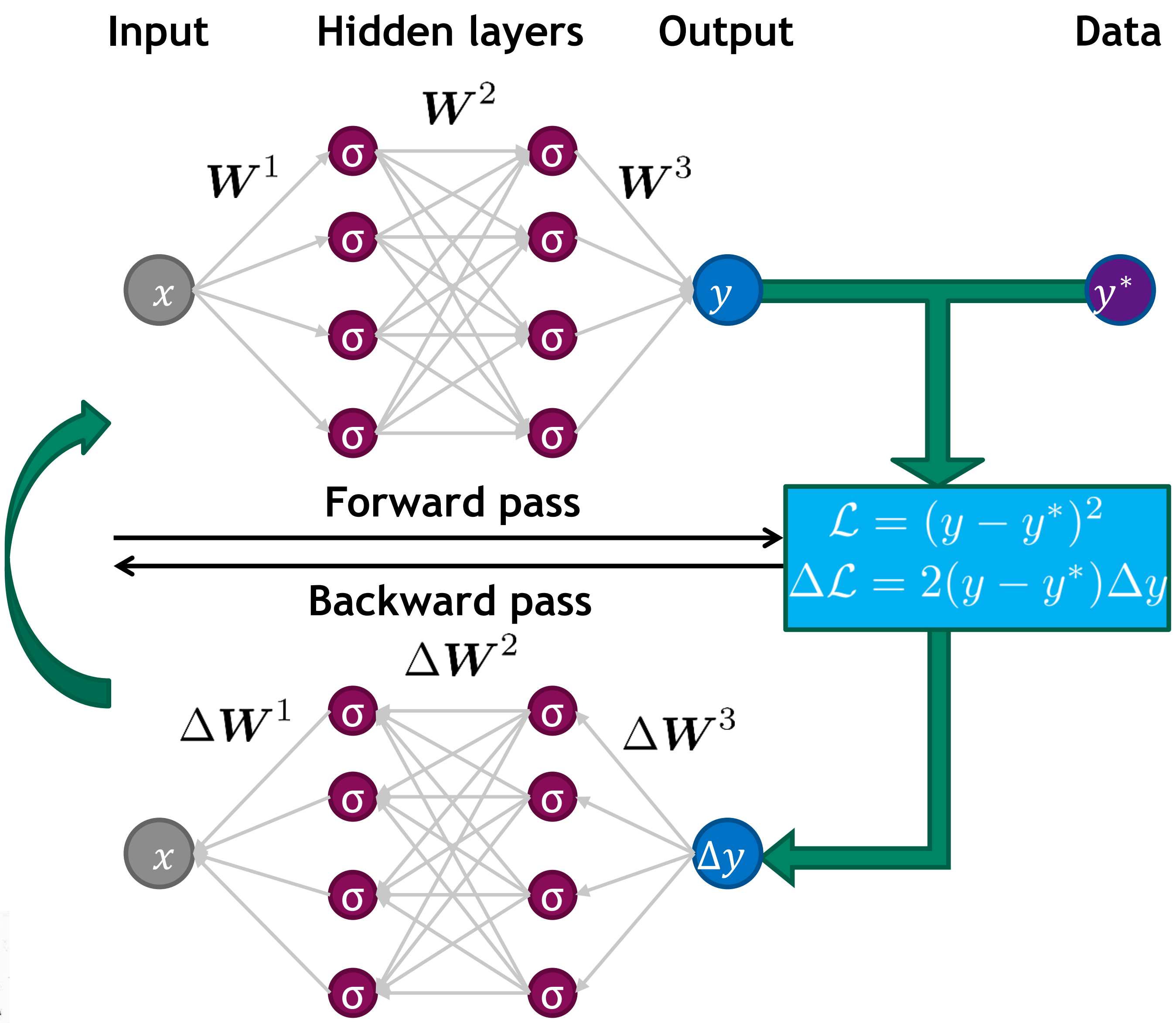
Johns Hopkins Turbulence Database

NIH National Heart, Lung, and Blood Institute

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)

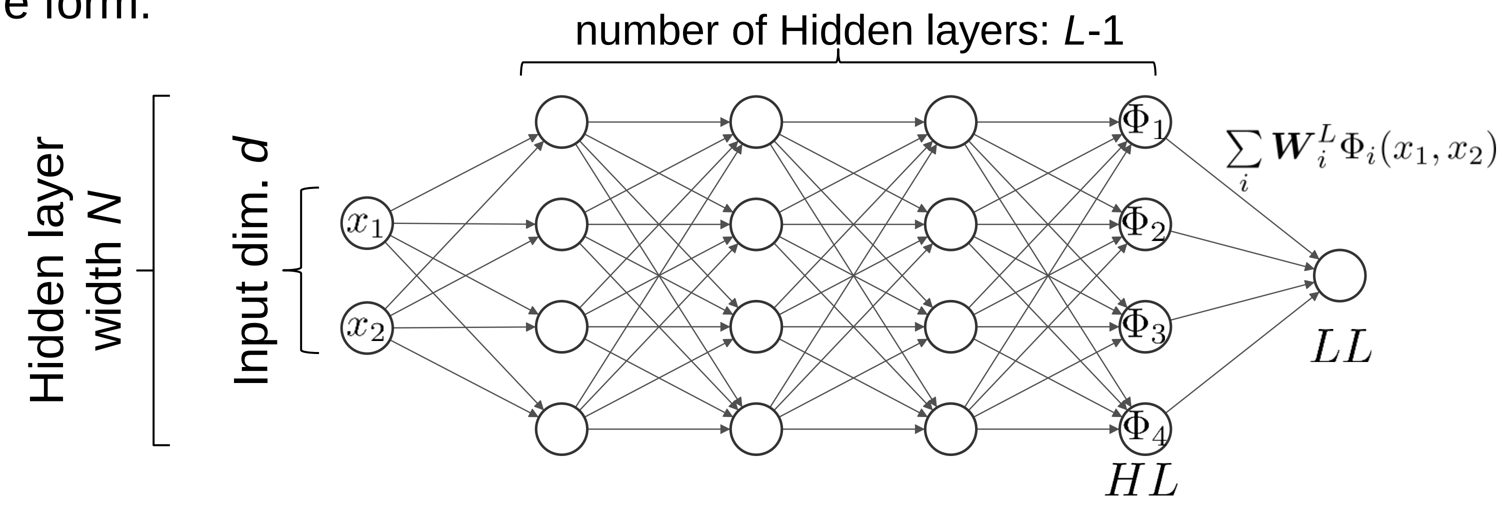
- The $L-1$ hidden layers of a feedforward neural network:

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

  Where $\circ$ denotes composition of functions

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

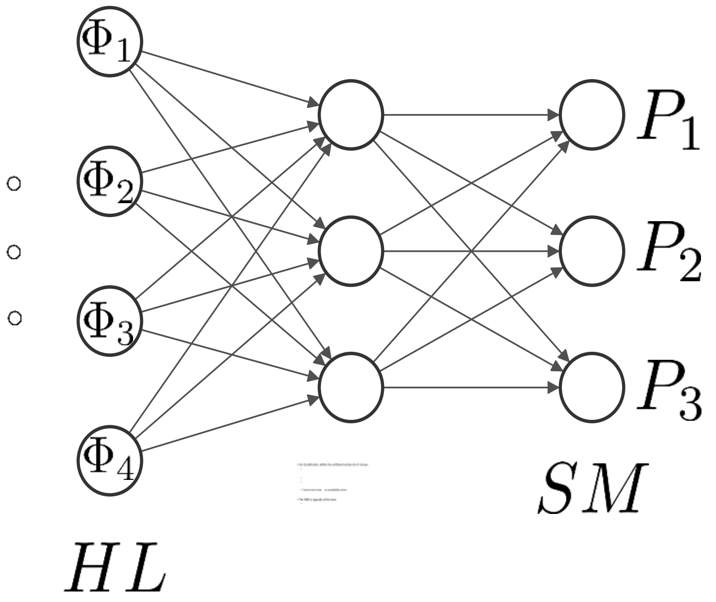
$$\mathcal{N}(x; \boldsymbol{\theta}) = T^L \circ \mathcal{N}_{HL}(x)$$



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

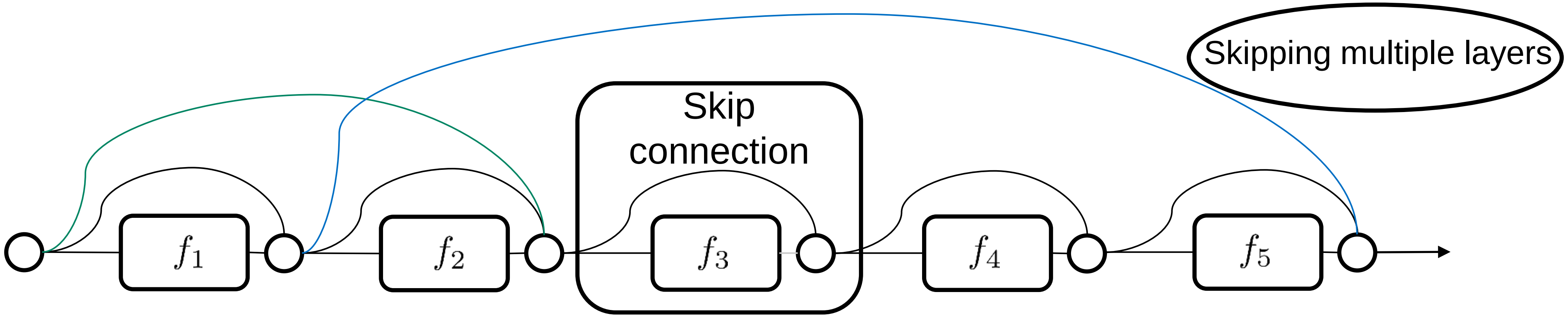
A Neural Network for Classification

- For classification, define the softmax function for K classes
  - $f_{SM}(\xi_i) = \dfrac{exp(\xi_i)}{\sum_{j=1}^{K} exp(\xi_j)}$
  - $0 \leq f_{SM}(\xi_i) \leq 1$
  - $\sum_i f_{SM}(\xi_i) = 1$
  - 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 $\mu \in M(I_N)$

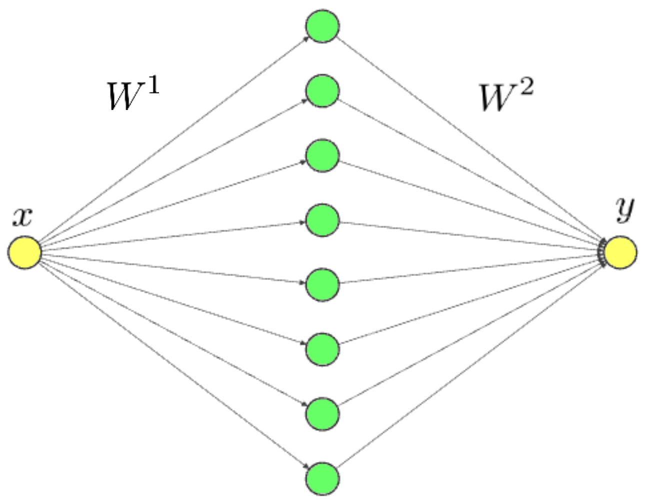
$$\int_{I_N} \sigma\left(W^1 x + b^1\right) d\mu(x) = 0$$

for all $W^1 \in \mathbb{R}^{n \times N}$ and $b^1 \in \mathbb{R}^n$ implies that $\mu = 0$. (I_N is compact on $\mathbb{R}^N$)

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

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

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

$$|y - f(x)| < \epsilon \quad \text{for all } x \in I_N$$

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 <u>Functional</u> 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 $a = x_0 < \ldots < x_m = b$, a positive integer N and constants $W_i^2, b_i, W_{i,j}, i = 1, 2, \ldots, N,$ $j = 1, 2, \ldots, m$

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 $u \in U$.

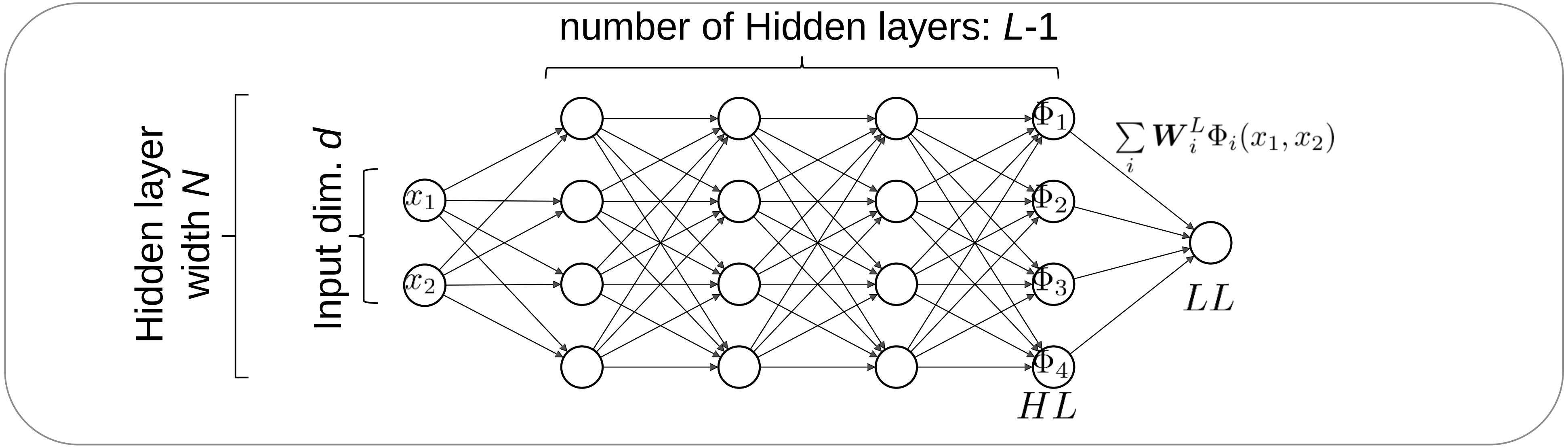
- T.P. Chen and H. Chen, Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Transactions on Neural Networks, 910-918, 4(6), 1993.

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.

Loss Functions

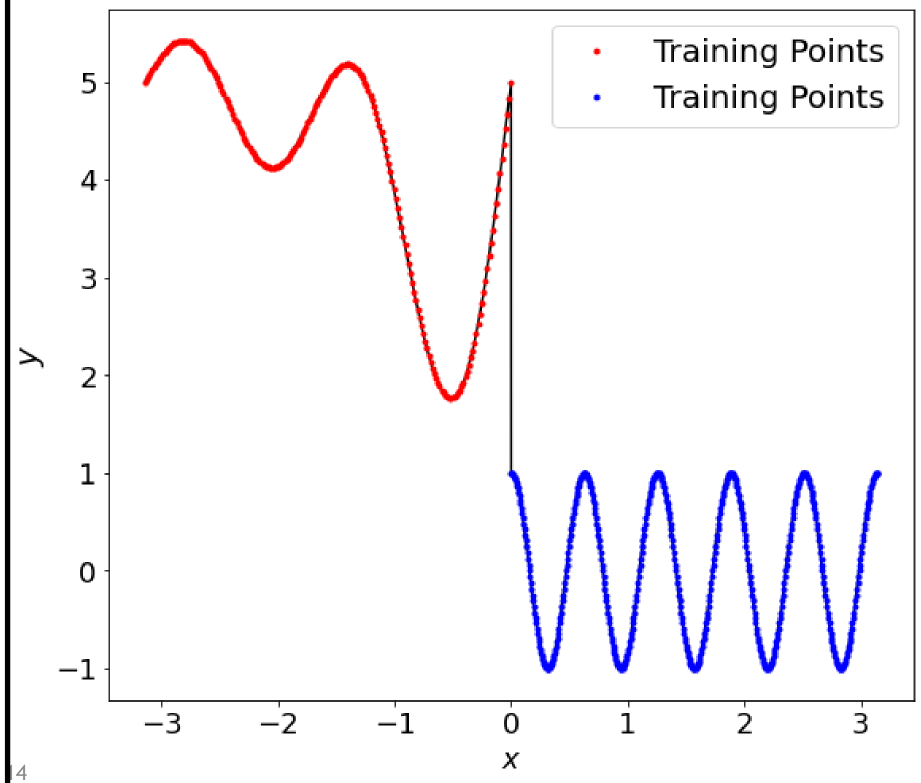
- To learn $u:\Omega \to \mathbb{R}$
- Given a dataset $\{(x_i, u(x_i))\}_{i=1}^{m}$
- Mean Squared Error (MSE) loss:
  - $\mathcal{L}(\boldsymbol{\theta}) = \|u(x) - \mathcal{N}(x;\boldsymbol{\theta})\|_2^2 \approx \frac{1}{m}\sum_{i=1}^{m}(u(x_i) - \mathcal{N}(x_i;\boldsymbol{\theta}))^2$
- In general,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

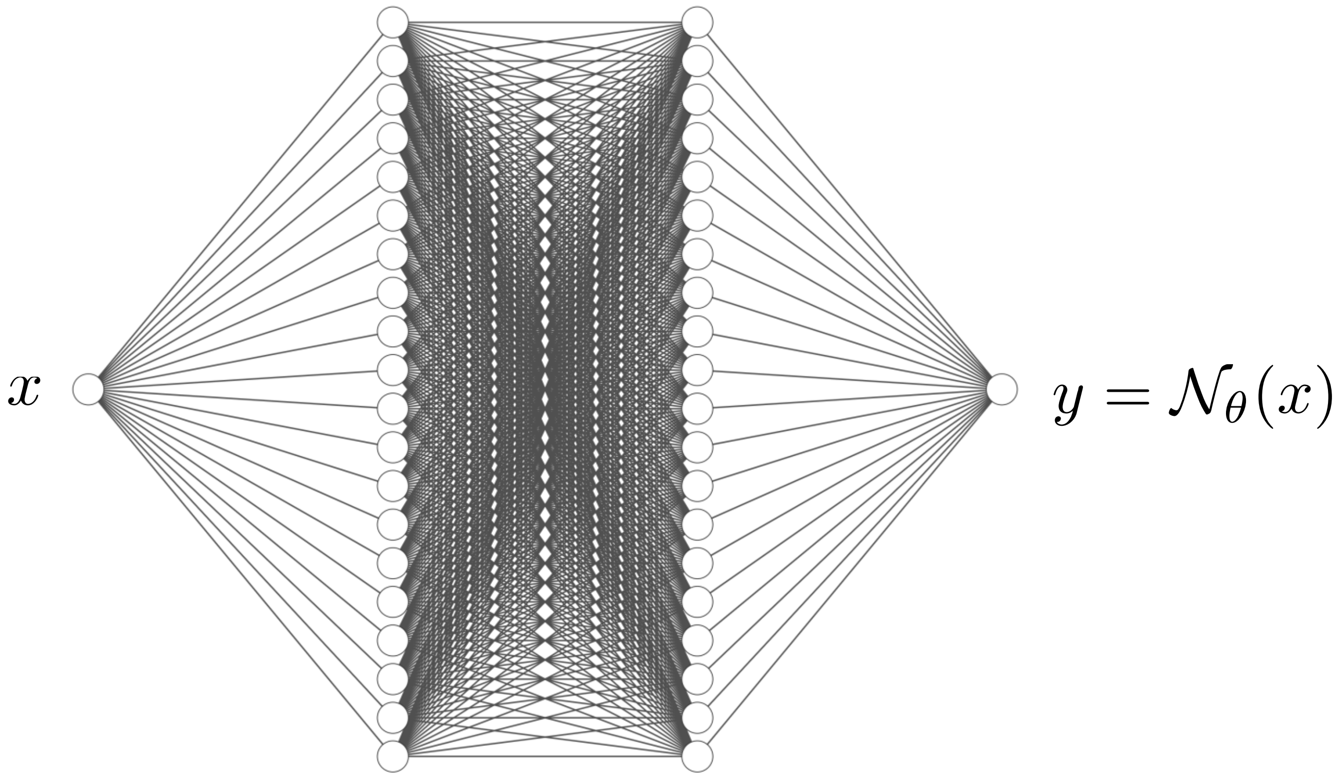
Regression of a Discontinuous/Oscillatory Function in Physical & Fourier Domains

- **Long Tail, Hierarchical training, Spectral Bias, Discontinuity**

$$y = 5 + \sum_{k=1}^{4} \sin(kx), \;\; x < 0$$

$$y = \cos(10x), \;\; x \geq 0$$





$$N = 20, \;\; L = 2, \;\; \text{and } m = 702$$

Regression of a Discontinuous/Oscillatory Function in Physical & Fourier Domains

Data preparation

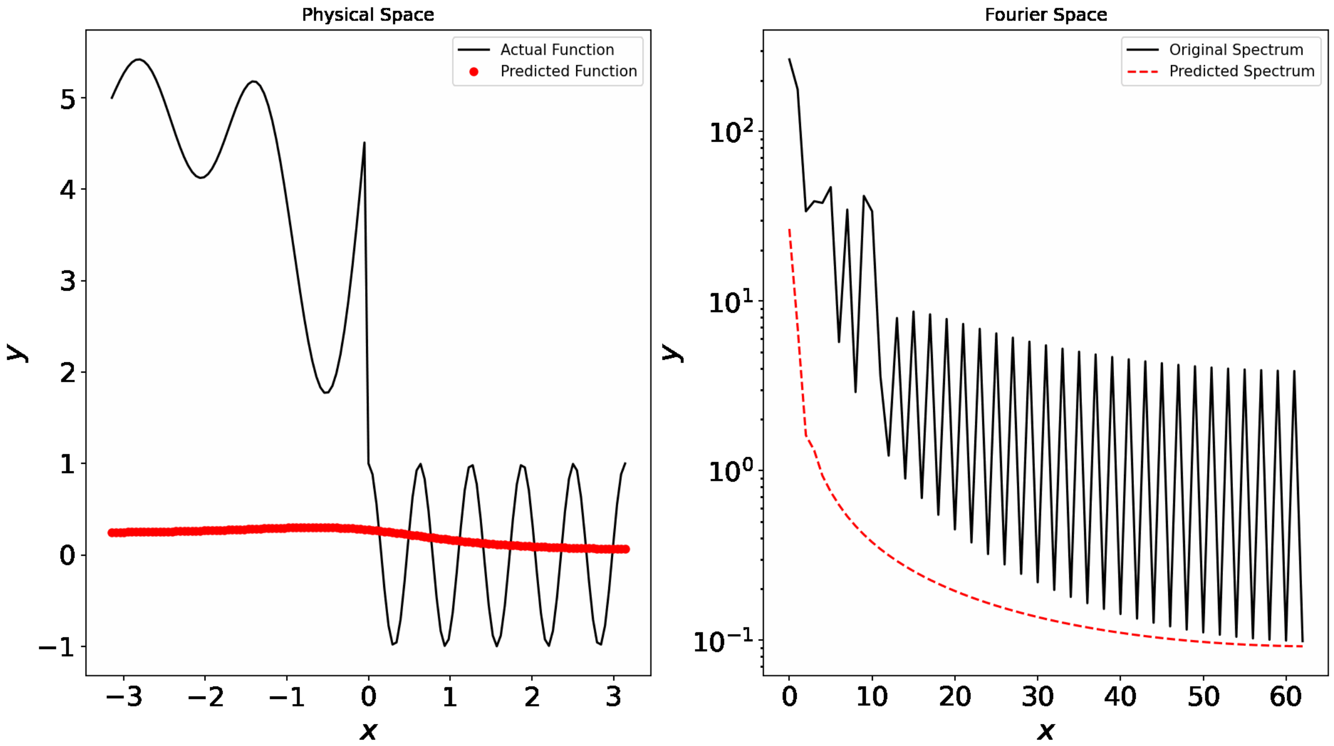
```python
if __name__ == "__main__":

    # Data for Traning and Testing
    x = np.linspace(-np.pi, np.pi, 129, dtype=np.float64)
    x = np.reshape(x, (-1, 1))
    y = np.array([fun_x(i) for i in x])

    F_y = F_transform(y[0:-1, 0])
    F_y_abs = abs(F_y)
    x_in_l = np.linspace(-np.pi, -1.0e-3, 201)
    x_in_r = np.linspace(0., np.pi, 501)
    x_in = np.concatenate((x_in_l, x_in_r), axis=0)
    y_in = np.array([fun_x(i) for i in x_in])


    N_i = 1
    N_o = 1
    L = 2
    N =20
    net = Net(N_i, L, N, N_o)
    x_train = torch.from_numpy(x_in.reshape(-1,1)).float()
    y_train = torch.from_numpy(y_in.reshape(-1,1)).float()
    x_test =  torch.from_numpy(x.reshape(-1,1)).float()

    optimizer = torch.optim.Adam(net.parameters(), lr=1.0e-03)
    loss_func = torch.nn.MSELoss()
```



Training Loop

```python
for n in range(N_iter):
    prediction = net(inputs)
    loss = loss_func(prediction, outputs)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

CRUNCH TIME
CRUNCH Group
Brown University

DEEP LEARNING INSTITUTE
BROWN

Approximation Rates

Convergence rate of shallow networks

Approximating functions defined on B^n, where

$$B^n = \left\{ \mathbf{x} : \|\mathbf{x}\|_2 = \left(x_1^2 + \cdots + x_n^2\right)^{1/2} \leq 1 \right\}$$

The Sobolev space $\mathcal{W}_p^m = \mathcal{W}_p^m\left(B^n\right)$ as the completion of $C^m\left(B^n\right)$ with respect to the norm

$$\|f\|_{m,p} = \begin{cases} \left(\sum_{0\leq|\mathbf{k}|\leq m} \| D^{\mathbf{k}} f|_p^p\right)^{1/p}, & 1 \leq p < \infty \\ \max_{0\leq|\mathbf{k}|\leq m} \left\|D^{\mathbf{k}} f\right\|_\infty, & p = \infty \end{cases}$$

Theorem (Maiorov 1999, lower and upper bounds) Let $n \geq 2$ and $m \geq 1$. For $M_N = \left\{ g = \sum_{i=1}^N c_i \sigma\left(w_i^\top x + b_i\right) \right\}$

$$\sup_{f\in\mathcal{W}_p^m, \|f\|_{m,p}\leq 1} \inf_{g\in M_N} \|f - g\| \sim N^{-m/(n-1)}.$$

CRUNCH TIME

CRUNCH Group

Brown University

DEEP LEARNING INSTITUTE

BROWN

Approximation Theory

The power of deep neural networks: convergence in depth vs width

Theorem (Shen et al 2021) Given any positive integers and a Hölder continuous function f on $[0,1]^n |f(x) - f(y)| \leq \lambda |x-y|^\alpha$, there exists a FloorReLU network $f^{\mathcal{N}}$ with width $\max\{n, 5N+13\}$ and depth $64nL+3$

$$\max_{x\in[0,1]^n} \left|f - f^{\mathcal{N}}\right| \leq 3\lambda n^{\alpha/2} N^{-\alpha\sqrt{L}}$$

Here a Floor-ReLU network refers to a fully connected network built with Floor $\lfloor \cdot \rfloor$ and ReLU activation functions (must have both). Idea of proof: using composition and nonlinearity of deep networks

- Exponential convergence in depth, algebraic convergence in width
- Lifting the curse of dimensionality

See also Yarotsky and Zhevnerchuk (2020), The phase diagram of approximation rates for deep neural networks, NeurIPS.

Approximation Theory

Some remarks

- The networks considered here are fully connected and feedforward networks, unless explicitly stated.
- Convergence rates are mostly characterized by the number of nonzero weights and biases in the literature.
- No optimization errors are included.
- Some references on approximation theory of neural networks.
  - ◇ Pinkus, A. (1999), Approximation theory of the MLP model in neural networks, Acta Numerica (1999), pp. 143-195.
  - ◇ Gühring, I., Raslan, M., and Kutyniok, G. (2020), Expressivity of Deep Neural Networks, arxiv: 2007.04759
- Other important topics:
  - Approximation of singular functions,
  - Approximation in various metrics.
  - Approximation with given data
  - . . .

Shallow networks vs Deep networks

- □ Universal approximator:
  - – Shallow networks: width $\to \infty$
  - – Deep networks: width $\sim d_{\text{in}} + d_{\text{out}}$ (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 □$\frac{\text{size deep}}{\text{size } e_{\text{shallow}}} \sim \epsilon^{d_{\text{in}}}$ [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]

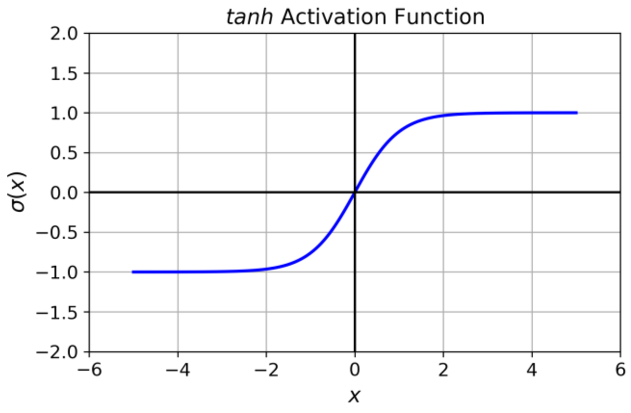
Activation Functions

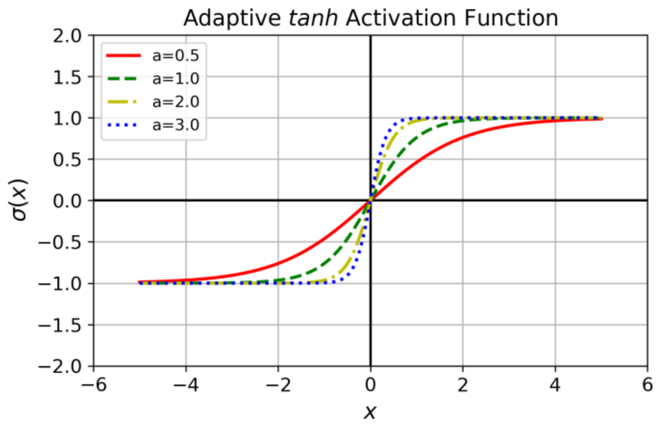
Conventional | Parameterized*

Tanh

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

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

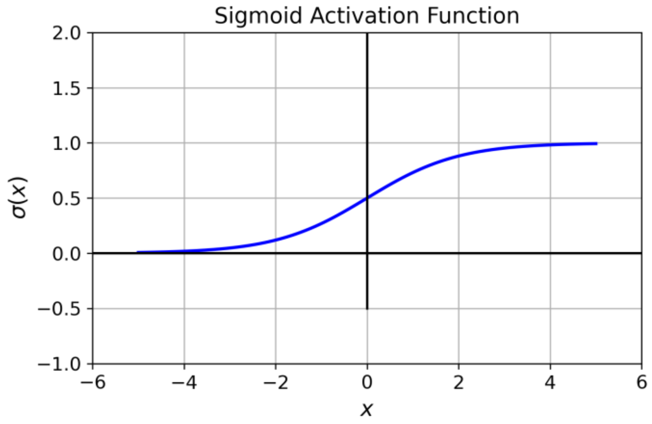


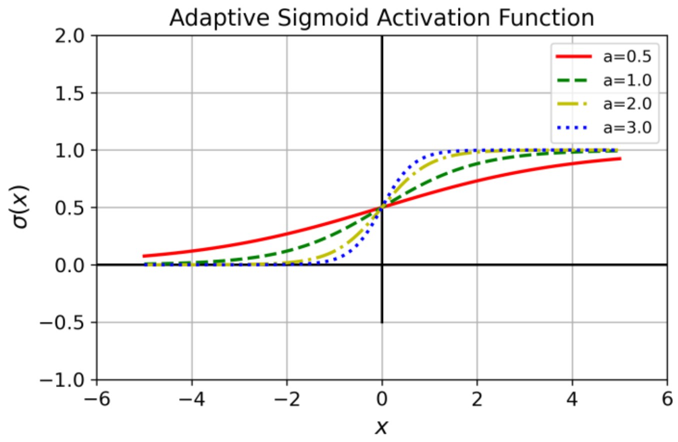


Sigmoid

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

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



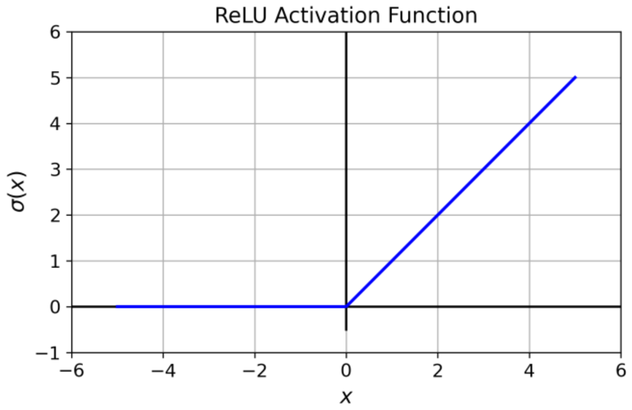


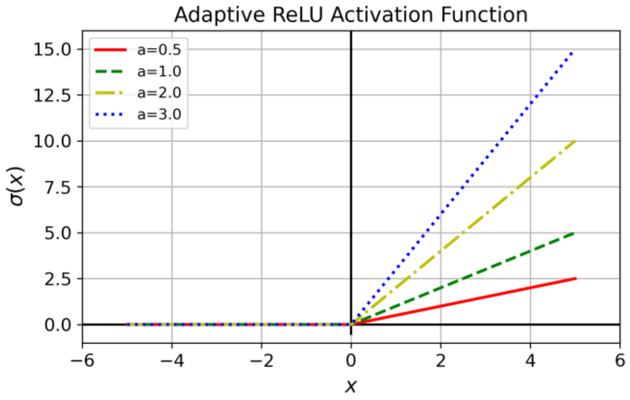
* In parametrization activation function consider $x \to ax$, where a is the trainable parameter.

ReLU

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

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



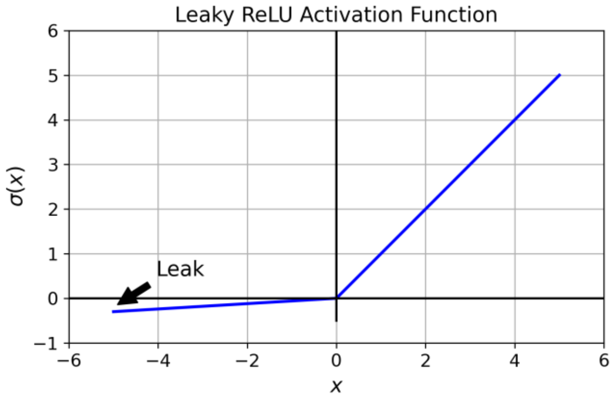


Leaky ReLU

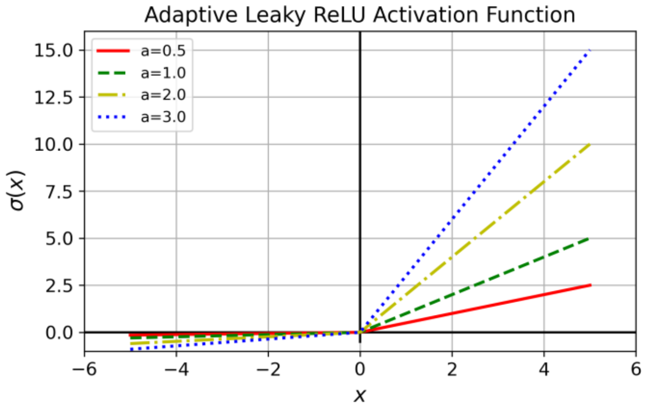
$$\sigma(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

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

Conventional



Parameterized

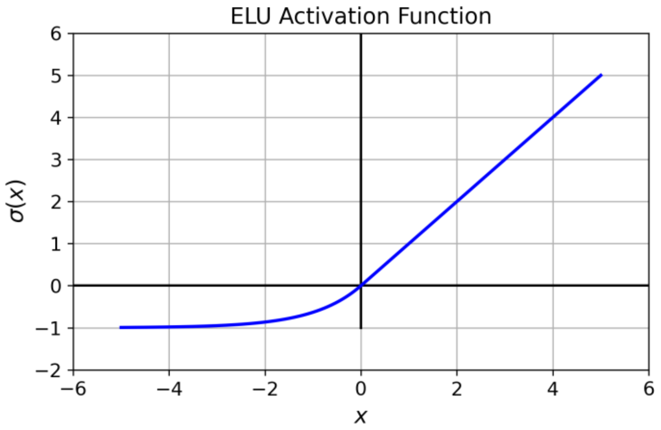


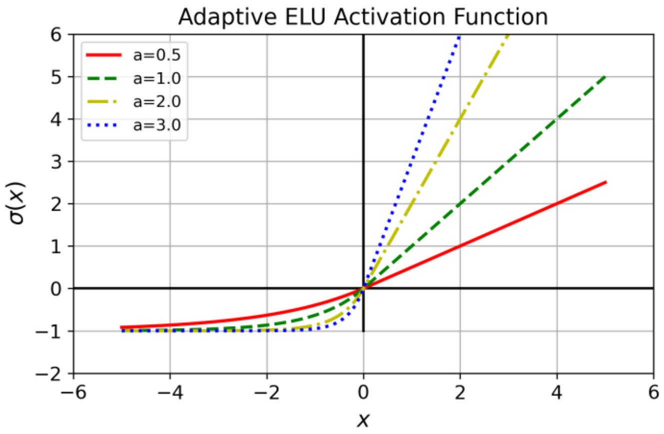
ELU

Conventional | Parameterized

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

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

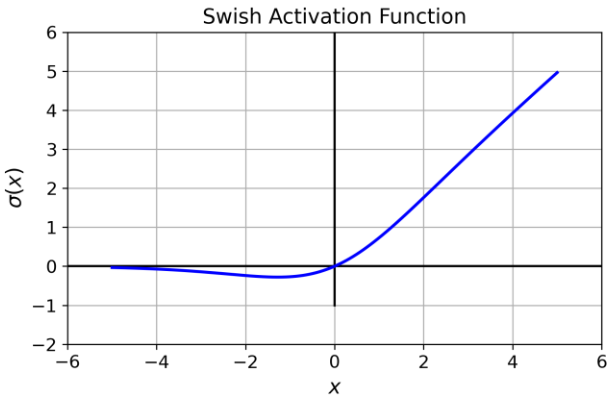


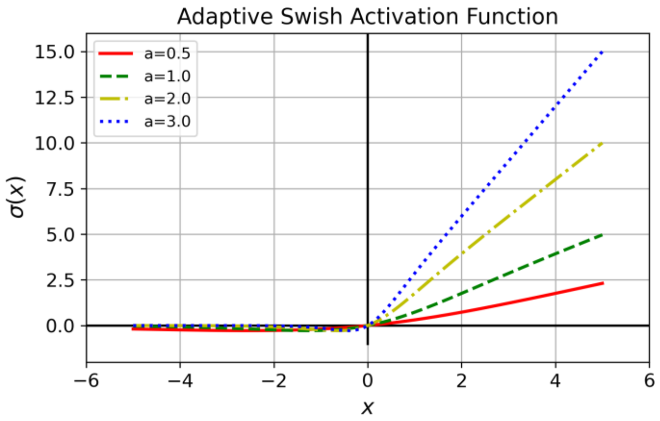


Swish

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

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





Activation Functions: A Recap

Adaptive Activation Functions: sin(x)

Adaptive Activation Functions: tanh(x)

Loss Functions: Implementation

TensorFlow 2.0

Mean Absolute Error Loss:

```
import tensorflow as tf
mae = tf.keras.losses.MeanAbsoluteError()
loss = mae(y_true, y_pred)
```

Mean Square Error Loss:

```
import tensorflow as tf
mse = tf.keras.losses.MeanSquaredError()
loss = mse(y_true, y_pred)
```

Huber Loss:

```
import tensorflow as tf
hl= tf.keras.losses.Huber(delta=1.0)
loss = hl(y_true, y_pred)
```

Mean Absolute Error Loss:

```
import torch.nn as nn
mae = nn.L1Loss()
loss = mae(y_pred, y_true)
```

Mean Square Error Loss:

```
import torch.nn as nn
mse = nn.MSELoss()
loss = mse(y_pred, y_true)
```

Huber Loss:

```
import torch.nn as nn
hl = nn.HuberLoss(delta=1.0)
loss = hl(y_pred, y_true)
```

** **delta** Specifies the threshold at which to change between delta-scaled L_1 and L_2 loss

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

- Hand-coded analytical derivative $f(x) : \mathbb{R}^n \to \mathbb{R}^m$
  - 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

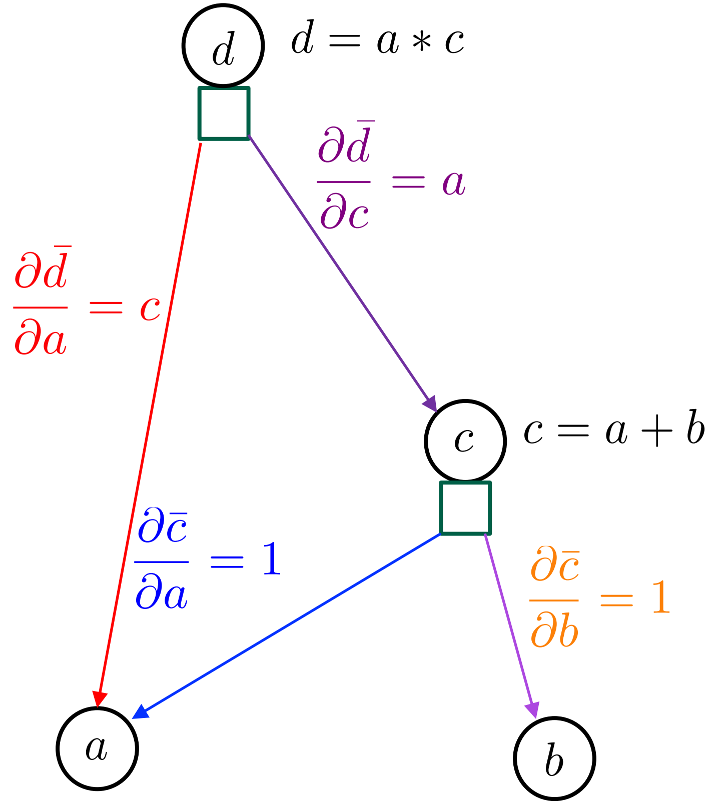
Automatic Differentiation

- 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

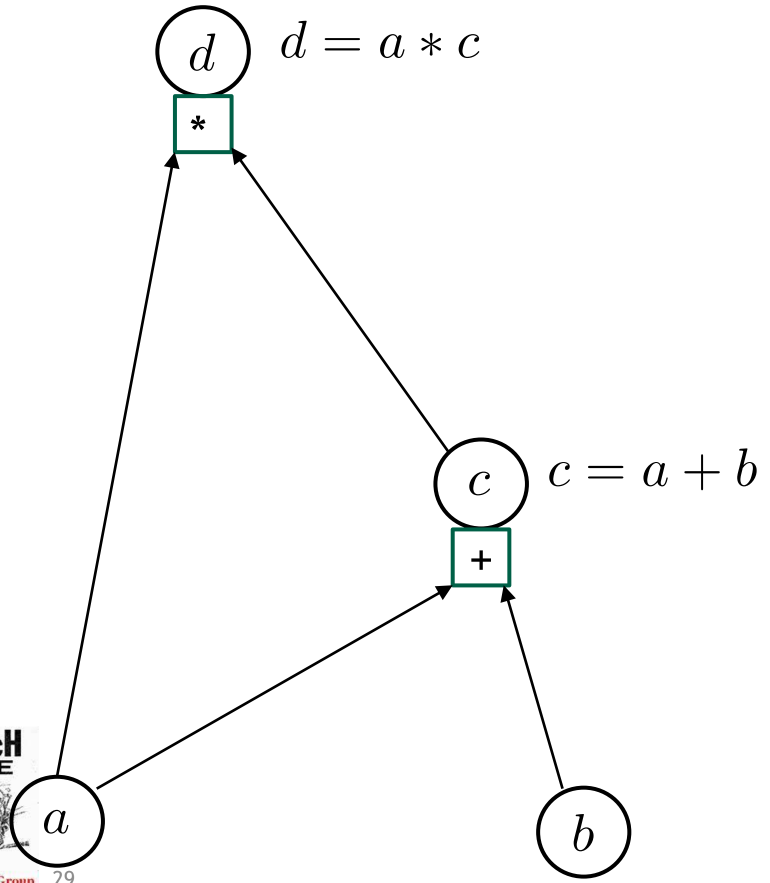


Backpropagation

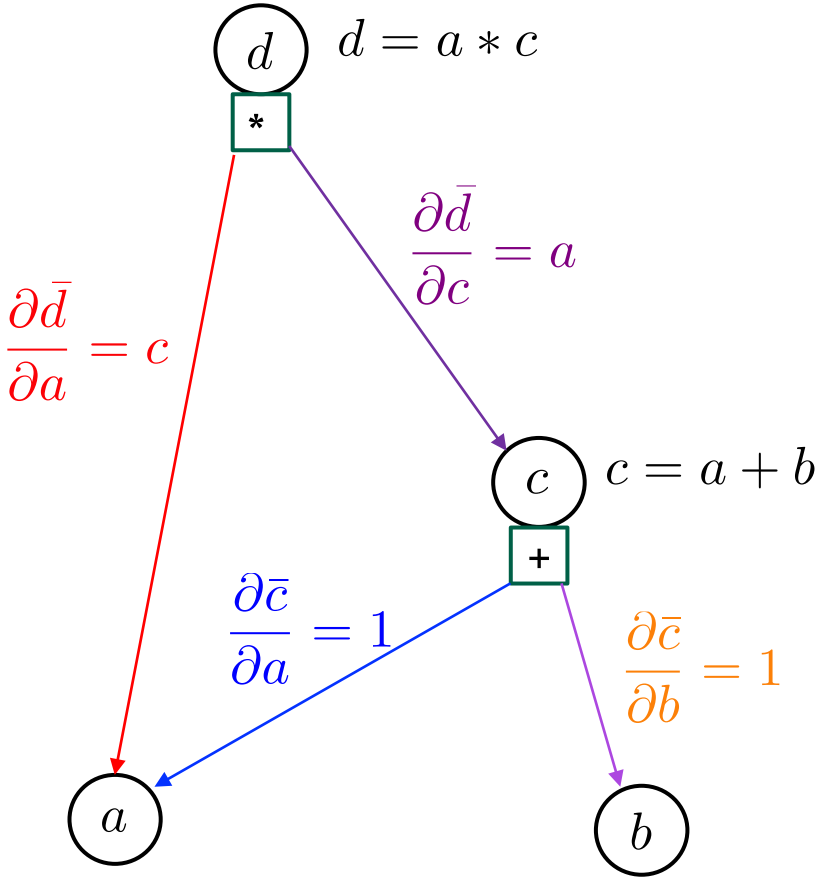
- We apply recursively the chain rule to implement Backprop
- Use computational graphs to accomplish backprop

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

Forward pass



Backward pass



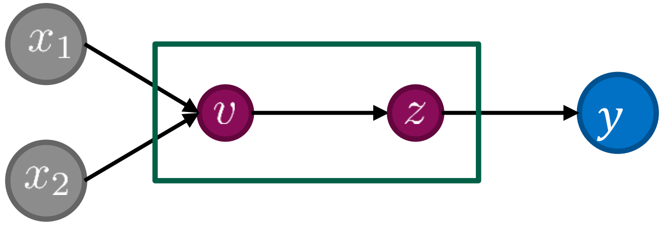
By chain rule:

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

$$= c + a$$

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

Backpropagation



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

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

Forward Pass:

$$Y = \phi\left(W * X + b\right)$$

```
Y = tf.nn.tanh(tf.add(tf.matmul(X, W), b))
```

Auto-differentiation:

$$\frac{\partial Y}{\partial X}$$

```
with tf.GradientTape(persistent=True) as tape:
       tape.watch([X])
       Y = ϕ(W*X + b)
Y_x = tape.gradient(Y, X)
del tape
```

Backward Pass + Optimization:

$$L = Y - Y_{actual}$$

$$\frac{\partial L}{\partial W} = W_0 - \eta * \frac{\partial L}{\partial W}$$

η Learning Rate

```
grads_loss = tape.gradient(L, [W,b])
opt = tf.optimizers.Adam(learning_rate=1e-3)
opt.apply_gradients(zip(grads_loss, [W, b])
```

PyTorch

```
import torch
import torch.nn as nn
```

Forward Pass: $Y = \phi\left(W * X + b\right)$

```
net = nn.Sequential()
net.add_module(‘ll1’, nn.Linear(X.shape[0], W.shape[0]))
net.add_module('tanh_layer_1', nn.Tanh())
Y = net(X)
```

Auto-differentiation: $\frac{\partial Y}{\partial X}$

```
Yx = torch.autograd.grad(Y, X, create_graph=True)
```

Backward Pass + Optimization:

$$L = Y - Y_{actual}$$

$$\frac{\partial L}{\partial W} = W_0 - \eta * \frac{\partial L}{\partial W}$$

$$\eta$$

```
optimizer = torch.optim.Adam([W, b], lr = learning_rate)
optimizer.zero_grad()
L.backward()
optimizer.step()
```

CRUNCH TIME
CRUNCH Group
Brown University

NVIDIA. DEEP LEARNING INSTITUTE | BROWN

Connecting Neural Networks with Finite Elements (FEM) – 1D

Recall the hat function

$$\varphi(x) = \begin{cases} 2x & x \in \left[0, \frac{1}{2}\right], \\ 2(1-x) & x \in \left[\frac{1}{2}, 1\right], \\ 0, & \text{otherwise.} \end{cases}$$

$$\varphi(x) = 2\,\mathrm{ReLU}(x) - 4\,\mathrm{ReLU}(x - \frac{1}{2}) + 2\,\mathrm{ReLU}(x-1).$$

- Linear finite element basis function φ_i

$$\varphi_i = \varphi\left(\frac{x - x_{i-1}}{2h}\right) = \varphi\left(w_h x + b_i\right), \quad w_h = \frac{1}{2h}, \quad b_i = \frac{-(i-1)}{2}.$$

$$\varphi_i \in \mathrm{Span}\left\{\mathrm{ReLU}(wx+b), w, b \in \mathbb{R}\right\}$$

- The Linear FEM space in 1D is a subspace of shallow neural networks with the ReLU activation function.

Connection to FEM: d-D

♢ For $d \geq 2$, the linear FEM space is not a subspace of shallow ReLU networks (He et al. 2020)

$$\left\{ \sum_{i=1}^{n} a_i \,\mathrm{ReLU}(w_i^T x + b_i), w_i \in \mathbb{R}^{1\times d},\ a_i, b_i \in \mathbb{R} \right\}$$

♢ For $d \geq 2$, the linear FEM space is a subspace of deep ReLU networks (Arora et al. 2016).

- Arora R, Basu A, Mianjy P, Mukherjee A. Understanding deep neural networks with rectified linear units. arXiv preprint arXiv:1611.01491. 2016.
- He J, Li L, Xu J, Zheng C. ReLU deep neural networks and linear finite elements. arXiv preprint arXiv:1807.03973. 2018 Jul 11.

Summary

- Neural Networks can approximate functions and functionals with arbitrary accuracy.
- Deep Neural Networks are more expressive and can beat the curse-of-dimensionality.
- Activation functions are important for accuracy and convergence speed.
- Deep Neural Networks can be thought as nonlinear approximations with adaptive basis functions.
- Regression and classification use different loss functions – loss functions can be meta-learned.
- Forward/Backpropagation and Automatic Differentiation have optimal cost and machine precision.
- Generalization and Optimization errors are often greater than Approximation errors.
- Traditional methods, like finite elements, can be related to adaptive Deep Neural Networks.

References

- Chapter 6 of the book by Goodfellow et al. (2016).
- Arora R, Basu A, Mianjy P, Mukherjee A. Understanding deep neural networks with rectified linear units. arXiv preprint arXiv:1611.01491. 2016.
- Attali, J.-G. and Pages, G. (1997). Approximations of functions by a multilayer perceptron: a new approach. Neural Networks, 10(6):1069-1081.
- Burton, R. M. and Dehling, H. G. (1998). Universal approximation in p-mean by neural networks. Neural Networks, 11(4):661-667.
- Carpenter, M. et al. The Stan Math Library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164. 2015 Sep 23.
- Chen, T.P. and Chen, H.. Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Transactions on Neural Networks, 910-918, 4(6), 1993.
- Cybenko, G.. Approximation by superpositions of a sigmoidal function, Mathematics of Control, Signals and Systems, 303-314, 2(4), 1989.
- Eldan, R. and Shamir, O.. The power of depth for feedforward neural networks. In Conference on Learning Theory, pp. 907–940, 2016.
- Hanin, B. and Sellke, M.. Approximating continuous functions by ReLU nets of minimal width. arXiv preprint arXiv:1710.11278, 2017.
- He J, Li L, Xu J, Zheng C. ReLU deep neural networks and linear finite elements. arXiv preprint arXiv:1807.03973. 2018 Jul 11.
- Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2(5):359-366.
- Jiang, C. and Chen, T. (1999). Denseness of superposition of one function with its translation and dilation in Sobolev space W^m_2 (R^n). Neural Networks, 42(3):495 -500.
- Kolmogorov, A. N. (1957), On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition, Doklady Akademii Nauk SSSR, 114, pp. 953-956.
- Liang, S. and Srikant, R.. Why deep neural networks for function approximation? In International Conference on Learning Representations, 2017.
- Maiorov, V. (1999). On best approximation by ridge functions. Journal of Approximation Theory, 99(1):68-94.
- Maiorov, V. and Pinkus, A. (1999), Lower bounds for approximation by MLP neural networks, Neurocomputing 25, 81-91.
- Mhaskar, H. Neural networks for optimal approximation of smooth and analytic functions. Neural Computation, 8(1):164–177, 1996.
- Mhaskar, H. and Poggio, T. Deep vs. shallow networks: An approximation theory perspective. Analysis and Applications, 14(06):829–848, 2016.

References

- Shen, Z., Yang, H., and Zhang, S. (2021). Optimal approximation rate of ReLU networks in terms of width and depth. arXiv 2103.00502.
- Shen, Z., Yang, H., and Zhang, S. (2021). Deep network with approximation error being reciprocal of width to power of square root of depth. Neural Computation, 33(4):1005{1036.
- Suzuki, T. (2018). Adaptivity of deep ReLU network for learning in Besov and mixed smooth Besov spaces: optimal rate and curse of dimensionality, arXiv:1810.08033.
- Whitney, H. (1934), Analytic extensions of differentiable functions defined in closed sets, Trans. Am. Math. Soc., 36 (1): 63-89.
- Yarotsky, D.. Error bounds for approximations with deep ReLU networks. Neural Networks, 94: 103–114, 2017.

Thanks to: Prof. Zhongqiang Zhang, WPI

Dr. Apostolos Psaros, Fidelity/Brown U

Deep Learning for Science and Engineering Teaching Kit

Thank You