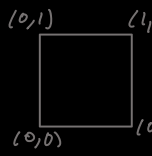
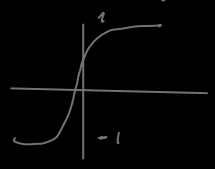
Laplacian: $-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2} = f = -\Delta u$



• Neural Network: $w = [\ ]_{8\times 8}$



$x \longrightarrow f \longrightarrow y$, $y = \tan^{-1}(wx+b)$, $y_1 = \tan^{-1}(y)$

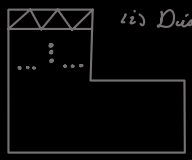
Physics-Informed-Neural-Networks (PINN): e.g. $m\frac{d^2\mu}{dx^2} + \mu\frac{d\mu}{dx} + k\mu = 0$

Forward Difference: $D^+ f(x) = \frac{f(x+h) - f(x)}{h}$, $x_{norm} = \frac{x - x_{mean}}{x_{std}}$

Lorenz System: $\begin{cases} \dot{x} = \sigma(y-z) \\ \dot{y} = x(\rho - z) - y \\ \dot{z} = xy - \beta z \end{cases}$, $x(t=0)=1$, $y(t=0)=1$, $\frac{dq}{dt} = f(q)$, $\frac{q(t+\Delta t) - q(t)}{\Delta t} = f(q)$, $q(t+\Delta t) = q(t) + \Delta t f(q)$ $t=\tau$

Finitie element method: $\int -\Delta u \cdot v \, d\Omega$ (v: test function)

$x = \begin{bmatrix} \vdots \end{bmatrix}_{100\times 1}$, $y = \begin{bmatrix} \ \end{bmatrix}_{100\times 1}$



(i) Discretize domain $u = \sum a_i \varphi_i(x)$; $\Delta u = \Delta(\sum a_i \varphi_i) = 0$, $-\int \Delta u \cdot v \, d\Omega = \int f \cdot v \, d\Omega$;

$m\frac{\partial^2 u}{\partial t^2} + d\frac{\partial u}{\partial t} - \nabla(c\nabla u) + au = f$

$x \in [-4, 4]$

$f(x) = x^2$, $y = x^2$

$N_x = 100$

(i) $y_{h_1} = W_1 x + b$

(ii) $y_{NL_1} = \phi(y_{h_1})$

$y_{h_2} = W_2 y_{NL_1} + b_2$, $y_{h_2} = \frac{1}{N}\sum (y_{pred} - Y)^2$

MSE: $\mathcal{J} = \frac{1}{N}\sum_i^N (\hat{y}_i - \hat{y}_D)^2$

$\min_{W,b} \mathcal{J}(y, \hat{y} \mid W, b)$ (convex), $W_{n+1} = W_n - \eta \frac{\partial \mathcal{L}}{\partial W_n}$ i.e. $\theta_{k+1} = \theta_k + \eta \nabla \mathcal{L}(\theta_k)$ (Gradient-based optimization)

NN for Regression: $T^l(x) = W^l x + b^l$, **Activation function** (σ): e.g. $\tanh(x)$, $\max\{0, x\}$, (iii) The $L-1$ hidden layers of a feedforward N-N.

$N_{HL} = \sigma \circ T^{l-1} \circ \dots \circ T^1(x)$, **NN for Classification:** $f_{SM}(\xi_i) = \frac{\exp(\xi_i)}{\sum_{j=1}^k \exp(\xi_j)}$, $0 \le f_{SM}(\xi_i) \le 1$, $\sum_i f_{SM}(\xi_i) = 1$. Convert any vector ξ_i to a probability vector

$N(x;\vec{\theta}) = f_{SM} \circ T^L \circ N_{HL}(x)$

$x \longrightarrow$ [ ] $\longrightarrow y$, $y = \varphi(wx+b)$. **Def:** We say σ is **discriminatory** if for measure $\mu \in \mathcal{M}(I_n)$ $\int_{I_N} \sigma(W'x + b')\, d\mu(x) = 0$ $\forall W' \in \mathbb{R}^{n\times N}$ and $b_i \in \mathbb{R}^n$ implies $\mu = 0$. (I_N compact on $\mathbb{R}^N$), ($M(I_n)$: Space of finite regular Borel measures on I_N)

Def: We say σ is sigmoidal if $\sigma(x) = \begin{cases} 1, & x \to +\infty \\ 0, & x \to -\infty \end{cases}$. **Theorem:** Let σ be any continuous discriminatory function, then finite sums of the form $y = \sum_{j=1}^N W_j^2 \sigma(W_j'x + b_j^1)$ are **dense** in $C(I_n)$: $\forall f \in C(I_n)$ and $\varepsilon > 0$, $\exists y = \sum^N W_j^2 \sigma(W_j'x + b_j^1)$ s.t. $|y - f(x)| < \varepsilon$ $\forall x \in I_N$

Theorem: Suppose U is a compact set in $C[a,b]$, f continuous functional defined on U, $\sigma(x)$

Adaptive Basis Viewpoint: $N_\theta : \mathbb{R}^d \longrightarrow \mathbb{R}$ consisting of $L-1$ hidden layers of width N composed with a final linear layer admitting the representation $N_\theta(x) = \sum_i^N W_i^L \Phi_i(x; \theta_H)$. **Loss Function:** (i) To learn $u: \Omega \longrightarrow \mathbb{R}$, given a dataset $\{(x_i, u(x_i))\}_{i=1}^m$

MSE loss: $\mathcal{L}(\theta) = \|u(x) - N(x,\theta)\|_2^2 \approx \frac{1}{m}\sum_{i=1}^m (u(x_i) - N(x_i,\theta))^2$. In general let $\{\mathcal{F}_k\}_{k=1}^K$ be a linear/nonlinear operator

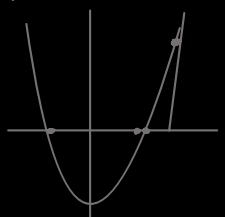
(i) $x \longrightarrow Wx + b \xrightarrow{(\tanh)} \varphi(Wx+b) = y_{nn}^1$, (ii) $\mathcal{J} = \frac{1}{N}\sum_{i=1}^N (y_{NN}^n - y_i)^2 \Longrightarrow \underset{[W,b]}{\operatorname{argmin}} \mathcal{J}$. (iii) $\begin{cases} W_{n+1} = W_n + \eta \frac{\partial \mathcal{J}}{\partial W_n} \\ b_{n+1} = b_n - \eta \frac{\partial \mathcal{J}}{\partial b_n} \end{cases}$

(Use GPU's for Matrix Multiplications)

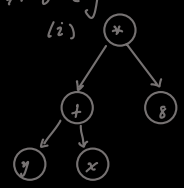
Suppose $f(x): y = x^2 - 2 \overset{!}{=} 0$

Taylor: $f(x) = f(x_n) + (x - x_n) f'(x) + \dots$

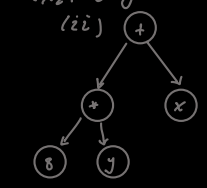
Newton's: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$



Computational Graph: ex_1: $8*(y+x)$

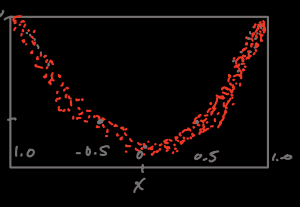


ex_2: $8*y+x$



L_p Norm: $\|X\|_p = \left(\sum_{i=1}^n x_i^p\right)^{1/p}$

$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \xrightarrow{f_{(NN)}} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$, $f: \mathbb{R}^{3\times 1} \longrightarrow \mathbb{R}^{3\times 1}$, $\partial f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_3} & \frac{\partial y_3}{\partial x_3} \end{bmatrix}_{3\times 3}$

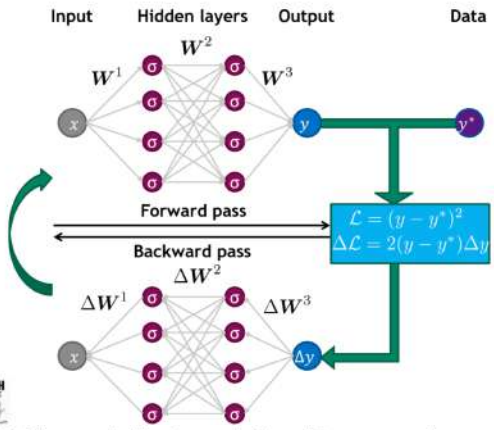


NN+ Function Approximation: $y = x^2 + \varepsilon$, $\varepsilon \sim \mathcal{U}[0,1)$, $x \in [-1,1]$, Noise = 10%

in_dim = 1, out_dim = 1

DL Networks: **Finite Difference:** $\frac{\partial f}{\partial x_i} \approx \frac{f(x + \Delta x_i) - f(x)}{\Delta x_i}$

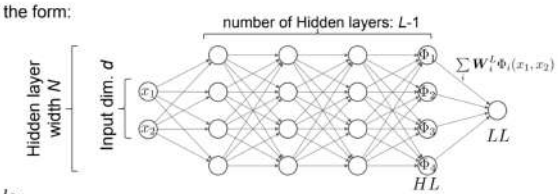
Workflow in a Neural Network



- 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

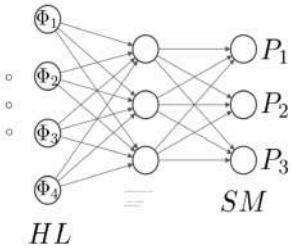
- Define the affine transformation in l-th layer
  $T^l(x) = \boldsymbol{W}^l x + \boldsymbol{b}^l$
- Activation function σ
  Popular choices: $tanh(x), max\{x, 0\}$ (Rectified Linear Unit, ReLU)
- The $L-1$ hidden layers of a feedforward neural network:
  $\mathcal{N}_{HL}(x) = \sigma \circ T^{L-1} \circ ... \circ \sigma \circ T^1(x)$
  Where $\circ$ denotes composition of functions
- For regression, a DNN Is typically of the form:
  $\mathcal{N}(x; \boldsymbol{\theta}) = T^L \circ \mathcal{N}_{HL}(x)$



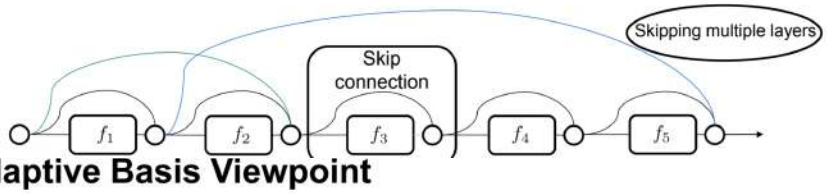
- Network parameters: $\boldsymbol{\theta} = \{\boldsymbol{W}^l, \boldsymbol{b}^l\}_{1 \le l \le L}$

A Neural Network for Classification

- For classification, define the **softmax function** for K classes
  - $f_{SM}(\xi_i) = \frac{exp(\xi_i)}{\sum_{j=1}^{K} exp(\xi_j)}$
  - $0 \le f_{SM}(\xi_i) \le 1$
  - $\sum_i f_{SM}(\xi_i) = 1$
  - Convert any vector ξ to a probability vector
- The DNN is typically of the form
  - $\mathcal{N}(x; \boldsymbol{\theta}) = f_{SM} \circ T^L \circ \mathcal{N}_{HL}(x)$



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



Adaptive Basis Viewpoint

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

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

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 $\boldsymbol{\theta}_H$.

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

Universal Function Approximation (1-layer):

Def: We say σ discriminatory if for measure $\mu \in M(I_N) = \{\lambda : \text{finite-signed-regular Borel measure on } I_N \subset \mathbb{R}^N\}$
$\int_{I_N} \sigma(W'x + b')\, d\mu(x) = 0 \quad \forall W' \in \mathbb{R}^{n \times N}, b' \in \mathbb{R}^n \Rightarrow \mu = 0$. ($I_N$ compact)

Def: σ sigmoidal if $\sigma(x) \to \begin{cases} 1, & x \to +\infty \\ 0, & x \to -\infty \end{cases}$. Theorem: Let σ continuous discriminatory, then $y = \sum_{j=1}^{N} W_j^2 \sigma(W_j' x + b_j')$ are dense in $C(I_N)$
$\therefore \forall f \in C(I_N), \varepsilon > 0\ \exists y = \ldots$ s.t. $|y - f(x)| < \varepsilon\ \forall x \in I_N$

Theorem: Let U compact in $C[a,b]$, f continuous functional on U, $\sigma(x)$ bounded sigmoidal function $\therefore \forall \varepsilon > 0$, $\exists m+1$ points $a = x_0 < \ldots < x_m = b$ $N \in \mathbb{N}$, constants $W_i^2, b_i, W_{ij}^1, b_{ij}^1, i = 1, 2, \ldots, N, j = 1, 2, \ldots, m$ s.t.
$\left| f(u) - \sum_{i=1}^{N} W_i^2 \sigma\left(\sum_{j=0}^{m} W_{ij}^1 u(x_j) + b_i\right) \right| < \varepsilon \quad \forall u \in U$. Loss Functions:
To learn $u: \Omega \to \mathbb{R}$, Given dataset $\{(x_i, u(x_i))\}_{i=1}^m$.
MSE: $\mathcal{L}(\theta) = \|u(x) - N(x;\theta)\|_2^2 \approx \frac{1}{m} \sum_{i=1}^{m} (u(x_i) - N(x_i;\theta))^2$
Let $\{F_k\}_{k=1}^K$ be linear/nonlinear operator $\mathcal{L}(\theta) = \sum_{k=1}^{K} \lambda_k \|F_k[u] - F_k[N]\|_2^2$

Classical Theory of Universal Approximation Theorem:
Def: (Tauber-Wiener function) If a function $\sigma: \mathbb{R}^m \to \mathbb{R}$ (continuous or discontinuous) satisfies that all the linear combinations $\sum_{i=1}^{N} c_i \sigma(W_i^T x + b_i)$, $W_i \in \mathbb{R}^{m \times m}$, $b_i \in \mathbb{R}^m$, $c_i \in \mathbb{R}$ are dense in $C(K)$ or $L^p(K)$, $K \subsetneq \mathbb{R}^m$ compact. Then σ is called a Tauber-Wiener (TW) function on $C(K)$ or $L^p(K)$
Theorem: Let K compact in $\mathbb{R}^m$. Then the Tauber-Wiener function on $C(K)$ or $L^p(K)$ exists. Sine $(x) \in TW$.
Approximation Theory (discontinuous functions)
Let $\sigma \in L^p[a,b]$ $\sigma \neq$ a polynomial a.e. Let $K \subset \mathbb{R}^d$ (compact)
Then $\forall f \in L^p(K)$ $\forall \varepsilon > 0$ $\exists$ real c_i's, W_i's, b_i's $1 \le i \le N$
s.t. $\left\| f - \sum_{i=1}^{N} c_i \sigma(W_i^T x + b_i) \right\|_{L^p(K)} < \varepsilon$, f (dis)continuous
$\Rightarrow \sigma$ a Tauber-Wiener function.
Maximum Norm: Let μ-finite on $(\mathbb{R}^d, B(\mathbb{R}^d))$
σ sigmoidal, $\forall$ Borel measurable $f: \mathbb{R}^d \to \mathbb{R}$
$\varepsilon > 0$, $\exists c_i, W_i, b_i, 1 \le i \le N$, K (compact) $\subset \mathbb{R}^d$
with $\mu(K) + \varepsilon > \mu(\mathbb{R}^d)$ s.t. $\sup_{x \in K} \left| f - \sum_{i=1}^{N} c_i \sigma(W_i^T x + b_i) \right| < \varepsilon$
Multilayer Neural Networks:
Kolmogorov Superposition Theorem: $\forall$ continuous f of n-variables on $[0,1]^n$ $\exists \lambda_1, \ldots, \lambda_n$, $\sum_{j=1}^{n} \lambda_j \le 1$ and $2n+1$ strictly continuous functions $\phi_1, \ldots, \phi_{2n+1}$ $\phi_i([0,1]) = [0,1]$
s.t. $f(x_1, \ldots, x_n) = \sum_{i=1}^{2n+1} g_f\left(\sum_{j=1}^{n} \lambda_j \phi_i(x_j)\right)$, $g_f \in C[0,1]$.
Maiorov-Pinkus Theorem: $\forall f \in C[0,1]^n$, $\varepsilon > 0$ $\exists d_i, c_{ij}, b_{ij}, \gamma_i$
$\vec{w}^{ij} \in \mathbb{R}^n$, σ (activation ρ_i) $\in C^\infty$ increasing-sigmoidal s.t.
$\max_{x \in [0,1]^n} \left| f(x) - \sum_{i=1}^{4n+3} d_i \sigma\left(\sum_{j=1}^{2n+1} c_{ij} \sigma(\vec{w}^{ij} \cdot x + b_{ij}) + \gamma_i\right) \right| < \varepsilon$

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

0. Data preparation

```
%% Code for Function regression
% Data Generation
% Define f(x) using anonymous definition
f = @(x) (x < 0).* (5.0 + sin(x) + sin(2.0*x) + ...
     sin(3.0*x) + sin(4.0*x)) + (x>=0) .* cos(10*x);

% Training and Testing data
x_in_l = linspace(-pi, -1.0e-3, 201);
x_in_r = linspace(0., pi, 501);
y_in_l = f(x_in_l);
y_in_r = f(x_in_r);
x_in = [x_in_l, x_in_r];
y_in = [y_in_l, y_in_r];
plot(x_in, y_in)
```

1. Initialize Neural Network Parameters

```
%% Parameters Initialization
parameters = struct;
numLayers = 2;
numNeurons = 40;
sz = [numNeurons 1];
parameters.fc1.Weights = initializeHe(sz,1,"single");
parameters.fc1.Bias = initializeZeros([numNeurons 1],"single");
for layerNumber=2:numLayers-1
    name = "fc"+layerNumber;
    sz = [numNeurons numNeurons];
    numIn = numNeurons;
    parameters.(name).Weights = initializeHe(sz,numIn,"single");
    parameters.(name).Bias = initializeZeros([numNeurons 1],"single");
end
sz = [1 numNeurons];
numIn = numNeurons;
parameters.("fc" + numLayers).Weights = initializeHe(sz,numIn, "single");
parameters.("fc" + numLayers).Bias = initializeZeros([1 1], "single");
```

2. Data preparation, mini-batches, and GPU placement

```
sz_in = size(X);
numEpochs = 60000;
miniBatchSize = sz_in(2);
executionEnvironment = "auto";
learningRate = 0.001;
ds = arrayDatastore(X);

mbq = minibatchqueue(ds, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFormat="CB", ...
    OutputEnvironment=executionEnvironment);
Y = dlarray(Y, "CB");

if (executionEnvironment == "auto" && canUseGPU) || ...
        (executionEnvironment == "gpu")
    X = gpuArray(X);
    Y = gpuArray(Y);
end
```

3. Training Loop

```
averageGrad = [];
averageSqGrad = [];
lossHist = zeros(numEpochs,1);
accfun = dlaccelerate(@modelLoss);
start = tic;
iteration = 0;
for epoch = 1:numEpochs
    reset(mbq);
    while hasdata(mbq)
        iteration = iteration + 1;
        X = next(mbq);
        [loss,gradients] = dlfeval(accfun, parameters,X,Y);
        [parameters,averageGrad,averageSqGrad] = adamupdate(parameters,...
            gradients,averageGrad, ...
            averageSqGrad,iteration,learningRate);
    end
    loss = double(gather(extractdata(loss)));
    lossHist(epoch) = loss;
    D = duration(0,0,toc(start),Format="hh:mm:ss");
    fprintf('On Epoch: %d, and Loss %f\n', epoch, loss)
end
toc;
```

Approximation Rates: $B^n \triangleq \{x : \|x\|_2 = (x_1^2+\dots+x_n^2)^{1/2} \le 1\}$, **Sobolev Space** $W_p^m = W_p^m(B^n)$ as the completion of $C^m(B^n)$ w.r.t. $\|f\|_{m,p} = \begin{cases} \left(\sum_{0\le|k|\le m} \|D^k f\|_p^p\right)^{1/p}, & 1\le p<\infty \\ \max_{0\le|k|\le m} \|D^k f\|_\infty, & p=\infty \end{cases}$

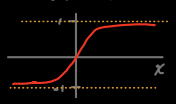
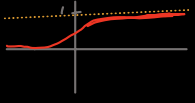
Theorem (down/upper bounds): $n\ge 2, m\ge 1$ For $M_N = \{g = \sum_{i=1}^{N} c_i \sigma(w_i^T x + b_i)\}$: $\sup_{g\in W_p^m, \|g\|_{m,p}\le 1} \inf_{g\in M_n} \|f-g\| \sim N^{-\frac{m}{(n-1)}}$ (**Shallow Networks**)

Theorem (deep Networks): Given $f\in C[0,1]^n$, $\forall L, N\in\mathbb{N}$ $p\in[1,\infty]$ $\exists$ ReLU forward Network f^N with width $C_1 = \max\{n\lfloor N^{1/n}\rfloor, N+2\}$ depth: $11L+C_2$ s.t. $\|f-f^N\|_{L^p([0,1]^n)} \le 131\sqrt{n}\,\omega\left(f, (N^2L^2\log_3(N+2))^{-1/n}\right)$, $C_1 = 16$, $C_2 = 18$ if $p<\infty$; $C_1 = 3^{n+3}$, $C_2 = 18+2n$ if $p=\infty$ and $\omega(f,\cdot)$ **modulus of continuity** $\omega(f,\delta) \triangleq \sup_{|z|\le\delta} \|f(\cdot+z)-f(\cdot)\|_{L^\infty}$, $x,y\in[0,1]^n$ when $|f(x)-f(y)|\le\lambda|x-y|^\alpha$, $0<\alpha\le 1$ $\|f-f^N\|_{L^p([0,1]^n)} \le 131\sqrt{n}\lambda(N^2L^2\log_3(N+2))^{-\alpha/n}$

Theorem: Given any positive numbers and a Hölder continuous function f on $[0,1]^n$, $|f(x)-f(y)|\le\lambda|x-y|^\alpha$, $\exists$ Floor-ReLU-network f^N width$(f^N)\le\max\{n, 5N+13\}$ depth $64nL+3$ $\max_{x\in[0,1]^n}|f-f^N| \le 3\lambda n^{\alpha/2}N^{-\frac{\alpha}{\sqrt{L}}}$

Hierarchical Binary Tree: e.g. $f(x_1,\dots,x_8) = h_3(h_{21}(h_{11}(x_1,x_2), h_{12}(x_3,x_4)), h_{22}(h_{13}(x_5,x_6), h_{14}(x_7,x_8)))$

Theorem: Let σ-infinitely differentiable-univariate which is not a polynomial on any subinterval of $\mathbb{R}$ $\because$ for $f\in W_p^m([-1,1]^n)$ f: compositional function conforming to a binary tree $\Rightarrow \exists$ hierarchical binary tree network $f^N \in D_N = \{\text{deep networks}\}$ s.t. $\max_{x\in[-1,1]^n}|f-f^N| \le cN^{-\frac{m}{n}}$ (**No curse of dimensionality**), For shallow networks with σ activation: $\le N^{-m/(n-2)}$

Activation Functions: (i) $\sigma(x) = \tanh(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}}$ $\sigma'(x) = 1-\sigma^2(x)$  , (ii) **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$, $\sigma'(x) = \sigma(x)(1-\sigma(x))$ 

(iii) **ReLU** $= \begin{cases} 0, & x\le 0 \\ x, & x\ge 0 \end{cases}$ $\sigma'(x) = \begin{cases} 0 & x<0 \\ 1 & x>0 \\ \text{undef.} & x=0 \end{cases}$, (iv) **Leaky ReLU**: $\sigma(x) = \begin{cases} 0.01x, & x\le 0 \\ x, & x\ge 0 \end{cases}$

(v) **ELU**: $\sigma(x) = \begin{cases} \alpha(e^x-1), & x\le 0 \\ x & x>0 \end{cases}$ (vi) **Swish**: $\sigma(x) = \frac{x}{1+e^{-x}}$

$\sigma'(x) = \begin{cases} \alpha e^x, & x<0 \\ 1, & x>0 \\ \text{undef.}, & x=0 \end{cases}$ $\wedge\alpha=1$ $\sigma'(x) = \frac{\sigma(x)}{x}(1+e^{-x}\sigma(x))$

Loss Functions: **Entropy**: $H(p) = -\sum_i p_i \log_2(p_i)$, **Cross-Entropy**: $H(p,q) = -\sum_i p_i \log_2(q_i)$, **KL-Divergence**: $D(p\|q) = H(p,q) - H(p)$

KL-Distribution-Convergence $\forall$ distributions $p(x), q(x)$ over X: $D[p(x)\|q(x)] = \sum_x p(x)\log\frac{p(x)}{q(x)} \ge 0$

Dual and Jacobian: Dual numbers $x = v+\dot{v}\varepsilon$ $\dot{u},\dot{v}\in\mathbb{R}$, $\varepsilon^2=0$, $f(x) = f(v+\dot{v}\varepsilon) = f(v)+f'(v)\dot{v}\varepsilon$ $\dot{u}=1$ computes $f'(x)|_{x=v}$

Chain-Rule: $f(g(v+\dot{v}\varepsilon)) = f(g(v)+g'(v)\dot{v}\varepsilon) = f(g(v)) + f'(g(v))g'(v)\dot{v}\varepsilon$, $\dot{v}=1 \Rightarrow (f(g(x)))'|_{x=v}$

Jacobian: $f:\mathbb{R}^n\to\mathbb{R}^m$, $\partial f(x) = \mathbb{R}^n\to\mathbb{R}^m$

$$\partial f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_{m(x)}}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{bmatrix} = \left[\frac{\partial f_i}{\partial x_j}(x)\right]_{i=1,j=1}^{m,n} \in \mathbb{R}^{m\times n}$$

JVP: directional derivative of $f:\mathbb{R}^n\to\mathbb{R}$ $|_x \in\mathbb{R}^n$ along $v\in\mathbb{R}^n$ is $\nabla f(x)^T v = \partial f(x)\,v$

$f(x) = \frac{1}{2}\|x\|_2^2 \Rightarrow \nabla f(x)^T v = x^T v$

VJP: $f:\mathbb{R}^n\to\mathbb{R}$ the Jacobian $|_x$ is $\partial f(x)\in\mathbb{R}^{1\times n}$, $\nabla f(x) = \partial f(x)^T 1$, $f(x) = \frac{1}{2}\|x\|_2^2$

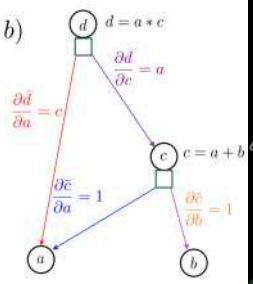
$\Rightarrow \nabla f(x) = x\cdot 1$

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

$$d = a * (a + b)$$

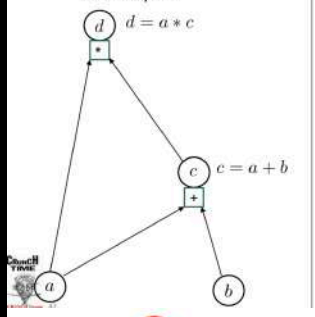


- In deep learning, **backpropagation** == Reverse mode AD
  - The input dimension of the loss function is # of parameters, e.g., millions
  - The output dimension is 1: the loss value
- High-order derivatives:
  - Nested-derivative approach: Apply first-order AD repeatedly
    - Cost scales exponentially 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 but it is included in JAX (jet)

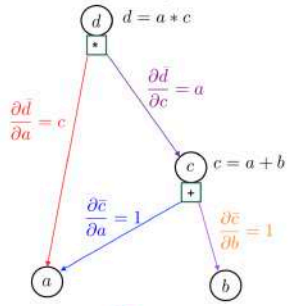
Backpropagation

- We apply recursively the chain rule to implement Backprop
- Use computational graphs to accomplish backprop
- Example: $d = a * (a + b)$

Forward pass



Backward pass



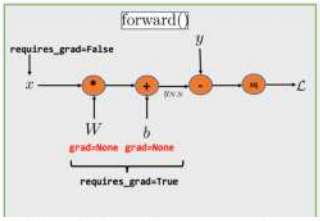
By chain rule:

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

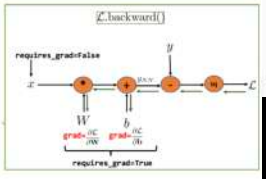
$$= c + a$$

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

1 2 3



$$y_{NN} = f(x) = \Phi(\mathbf{W}x + \mathbf{b})$$
$$y = f(x)$$
$$\mathcal{L} = \frac{1}{N}\sum_i (y_{NN} - y)^2$$



- 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.

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \nabla f(\boldsymbol{x}_n)$$

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \boldsymbol{H}_n^{-1} \nabla f(\boldsymbol{x}_n)$$

First Order	Second Order
Momentum	Newton Method
Nestrov	Quasi-Newton Methods: BFGS
Adagrad	Quasi- Newton Methods: L-BFGS
RMSProp	
Adam	
Nadam	
Rprop	
Lion	

Optimization: Stationary Points: $\nabla \mathcal{L}(\theta) = 0$ w.r.t. θ

$\theta^* = \arg\min_\theta \mathcal{L}(\theta)$, $\theta_{n+1} = \theta_n - \eta \nabla_\theta \mathcal{L}(\theta)$

$\mathcal{L}(\theta - \eta \nabla \mathcal{L}(\theta)) = \mathcal{L}(\theta) - \eta \|\nabla \mathcal{L}(\theta)\|_2^2 + O(\eta^2)$

Saddle Points: (i) **Degenerate**: Hessian positive semidefinite and 0-eigenvalue, **Hessian**: Suppose $f: \mathbb{R}^n \to \mathbb{R}$. If all second order partial derivatives of f exist. $\Rightarrow H_{n\times n}$:

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

positive definite if $x^T M x > 0$

$\forall \vec{x} \neq 0 \in \mathbb{R}^n$, H_f positive d. $\iff x^T H_f x \ \forall x \in \mathbb{R}^n \setminus \{0\}$

Batch Normalization:

Algorithm:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^m x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Learning rate Scheduling: (i) Train model for a few hundred iterations, exponentially increasing $\eta(t)$

Dynamic Scheduling:

$$\begin{cases} \eta(t) = \eta_i, & t_i \le t \le t_{i+1} : \text{Piecewise Decay} \\ \eta(t) = \eta_0 \exp(-\lambda t) : \text{Exponential decay} \\ \eta(t) = \eta_0 (\beta t + 1)^{-\alpha} : \text{Polynomial decay} \end{cases}$$

Theoretical Analysis: N^L will die in probability as $L \to \infty$

Theorem Let N^L be a ReLU-NN with L layers each having $N_1, \dots, N_L$ neurons. Suppose weights are initialized from a symmetric distribution around zero (ii) Biases are 0 or from a symmetric distribution $\therefore \mathbb{P}(N^L(x) \text{ dies}) \le \prod_{l=1}^{L-1} (1 - (1/2)^{N_l})$. Furthermore, assuming $N_l = N \ \forall l$: $\lim_{L\to\infty} \mathbb{P}(N^L \text{ dies}) = 1$, $\lim_{N\to\infty} \mathbb{P}(N^L \text{ dies}) = 0$

Proof: **Lemma** Let N^L - ReLU-NN L-layers. Suppose weights ind. from distribution s.t. $\mathbb{P}(W_j^1 z = 0) = 0 \ \forall z \neq 0 \in \mathbb{R}^{N_{L-1}}$ and any jth row of W_L. Then $\mathbb{P}(N^L(x) \text{ die}) = \mathbb{P}(\exists l \in \{1, \dots, L-1\}$ s.t. $\phi(N^l(x)) = 0 \ \forall x \in D)$ For a given x

$\mathbb{P}(W_s^j \phi(N^{j-1}(x)) + b_s^j < 0 \mid \tilde{A}_{j-1}^c x) = 1/2$

$\tilde{A}_{l,x}^c = \{\forall 1 \le j < l, \phi(N^j(x)) = 0\}$

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

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

- After taking the gradient

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

- 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 terms in the 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.

Summary

- ❑ The main difficulty in training a NN is due to back propagation and the need to minimize a high-dimensional non-convex function.
- ❑ Stochastic gradient descent (SGD) and mini-batch GD with decaying learning rate are effective and practical methods.
- ❑ Underfitting occurs for a low-capacity model while overfitting occurs for a high-capacity model.
- ❑ The Xiavier and He initialization as well as the data normalization are key components of effective NN training.
- ❑ Batch normalization can lead to faster convergence, avoid vanishing gradients, allows higher learning rates and act as regularizer.
- ❑ There are many optimizers but the combination of Adams first, followed by L-BFGS is usually the winner.
- ❑ Decaying learning rate and cyclical learning rate are important in accelerating convergence and avoiding bad minima.
- ❑ L2 regularization leads to better generalization while L1 encourages sparsity; which one to use is problem-dependent.
- ❑ Dropout regularization is effective but depends stronlgy on the dropout rate, which should be variable across the layers.
- ❑ Information bottleneck theory suggests a phase transition in deep learning, from fitting to early layers to true learning in the higher layers.
- ❑ Deep neural networks are expressive but they can collapse during training yielding erroneous results, e.g. a deep and narrow ReLu network.

Structure of deep convolutional neural networks

- Rectified linear unit (ReLU): $\sigma(u) = (u)_+ = \max\{u, 0\}$
- sequence of convolutional filter masks $\boldsymbol{w} = \{w^{(j)} = (w_k^{(j)})_{k=-\infty}^{\infty}\}_j$
- **filter length** s: Assume $w^{(j)}$ is suported in $\{0, 1, ..., s\}$
- **convolution** $w * x = (\sum_k w_k x_{i-k} = \sum_{k=0}^{s} w_{i-k}x_k)_i$ supported in $\{0, 1, ..., N + s\}$, so $(w * x)_{i=0}^{N+s} = W^w x$ with $W^w = (w_{i-k})_{0\le i\le N+s, 0\le k\le N}$

$$W^w = \begin{bmatrix} w_0 & 0 & 0 & 0 & \dots & \dots & 0 \\ w_1 & w_0 & 0 & 0 & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ w_s & w_{s-1} & \dots & w_0 & 0 & \dots & 0 \\ 0 & w_s & w_1 & w_0 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & w_s & \dots & w_1 & w_0 \\ 0 & \dots & 0 & 0 & w_s & \dots & w_1 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & w_s \end{bmatrix} \in \mathbb{R}^{(N+s)\times N}$$

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:

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

FNN vs CNN

FNN

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

CNN

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

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

Properties of CNN

- Efficiency: Much fewer parameters $(W \times H)^2 \to K^2$
- 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 $\boldsymbol{H}$

- Two probaility distributions P and Q.
- Total variation (TV) distance

$$\delta(P, Q) = \sup|P(A) - Q(A)|$$

- Kullback-Leibler (KL) divergence (also called relative entropy)
  - Discrete probability distributions

$$D_{KL}(P||Q) = \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{P(x)}{Q(x)}\right)$$

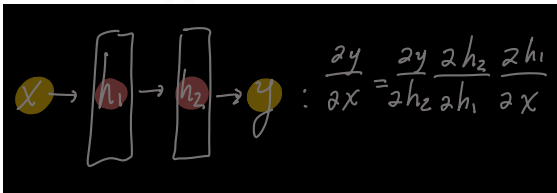
  - Continuous random variable

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x)\log\left(\frac{p(x)}{q(x)}\right)dx$$

  - $H(P, Q)$: cross entropy of P and Q

$$D_{KL}(P||Q) = H(P, Q) - H(p),$$ **Asymmetric!**

where $H(P)$ denotes the entropy of P.



A **deep CNN of depth** L is an L-layer neural network with widths $\{N_j = N + js\}$ having a **convolutional sparse structure**:

$$z^{(j)}(x) = \sigma(W^{(j)} z^{(j-1)}(x) - b^{(j)}, \quad j = 1, 2, ..., L$$

where $W^{(j)} = W^{w^{(j)}}$ is a Toeplitz type $N_j \times N_{j-1}$ matrix generated by the filter mask $w = w^{(j)}$

The generated **hypothesis space** of functions:

$$\mathcal{H}_L^{\boldsymbol{w},\boldsymbol{b}} = \left\{ \sum_{k=1}^{N+Ls} c_k z_k^{(L)}(x) : c = (c_i)_{i=1}^{N+Ls} \in \mathbb{R} \right\}$$

Take $b^{(j)}$ of the form $b = [b_1 \dots b_{s-1} b_s b_s \dots b_s b_{N_j - s+1} \dots b_{N_j}]^T$
Bias vector sequence $\boldsymbol{b} = \{b^{(j)}\}_{j=1}^{L}$
Total number of free parameters: $\mathcal{N} = 3s(L-1) + 2N + 2L$

Theorem

Let $2 \le s \le N$. For and compact subset Ω of $\mathbb{R}^N$ and any $f \in C(\Omega)$, there exist sequences $\boldsymbol{w}$ of filter masks, $\boldsymbol{b}$ of bias vectors, and $f_L^{\boldsymbol{w},\boldsymbol{b}} \in \mathcal{H}_L^{w,b}$ such that $\lim_{L\to\infty} ||f - f_L^{\boldsymbol{w},\boldsymbol{b}}||_{C(\Omega)} = 0$

- **Learning an autonomous dynamical system:**

$$\dot{z}(t) = F(z),\ z \in \mathbb{R}^d,\ z(0) = z_0$$

where $F(z)$ is the unknown function.

- Trajectory data: $k = 1, ..., K$
the k-th trajectory $= \{(t_0, z_0^{(k)}), (t_1, z(t_1; z_0^{(k)})), ..., (t_N, z(t_N; z_0^{(k)}))\}$
- Goal: Approximate $F(z)$ using data and neural networks $F^\theta(z)$

1. For given θ, generate trajectories from $\frac{d\tilde{z}}{dt} = F^\theta(\tilde{z}),\ \tilde{z}(0) = z_0$
the k-th trajectory $= \{(t_0, z_0^{(k)}), (t_1, \tilde{z}^{(k)}(t_1)), ..., (t_N, \tilde{z}^{(k)}(t_N))\}$
2. Define a loss: $\mathcal{L}(\theta) = \frac{1}{K}\sum_{k=1}^{K} \frac{1}{N}\sum_{i=1}^{N} ||\tilde{z}^{(k)}(t_i) - z(t_i)||^2$
3. $\min_\theta \mathcal{L}(\theta) \to \theta^* \to F^{\theta^*}(z)$

Pure Data Driven | Q: How to incorporate physics?

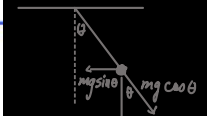
A: Embed physical laws into $F^\theta(z)$

Dynamical Systems:
(1) Data Driven
(2) Physics Informed Data Driven:
Soft Constrained — Hard Constrained

$\mathcal{L} = \sum_{f_D} (U_{NN} - u)^2 + (\nabla \cdot U_{NN})^2$, s.t. $\nabla \cdot U_{NN} = 0$

$\frac{dq}{dt} = F(q,t)$: "Discovery of closed form governing laws - equations with sparse/scarce and noisy data"
To solve $\frac{dq}{dt} = F(q,t)$ we need $q(t=0) = q_0$
Bounded Domain Ω
"Black Box to Grey Box"
Problem: design NN ϕ s.t. $x(t_i) \approx \phi(x(t_i - dt))$.

Learning Autonomous Dynamical System:
$\dot{z}(t) = F(z),\ z \in \mathbb{R}^d,\ z(0) = z_0,\ F(z)$ unknown



$\therefore -mg\sin\theta = ma,\ s = l\theta$
$a = l\frac{d^2\theta}{dt^2}$
$\Rightarrow -mg\sin\theta = ml\frac{d^2\theta}{dt^2}$
$\therefore \frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0$

Lagrangian: $L(x,v) := T - U$
Hamiltonian $H(q,p) = \sum_i p_i q_i - L(q,\dot{q},t),\ p_i = \frac{\partial L}{\partial \dot{q}_i}$
Euler-Lagrangian Eq: $\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) = \frac{\partial L}{\partial \theta}$

Approach 1: Neural ODE

Question: How to get $\tilde{z}$ from $\frac{d\tilde{z}}{dt} = F^\theta(\tilde{z}),\ \tilde{z}(0) = z_0$
Using numerical integrator

- Forward Euler: Residual block in the ResNet, (He et al., 2016)

$$\tilde{z}(t_{i+1}) = \tilde{z}(t_i) + hF^\theta(\tilde{z}(t_i))$$

- Runge-Kutta: (Rico-Martinez et al., 1992)

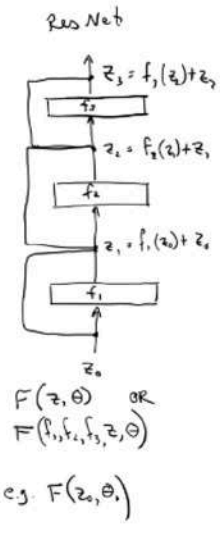
$$k_1 = hF^\theta(\tilde{z}(t_i)), \quad k_2 = hF^\theta(\tilde{z}(t_i) + \frac{k_1}{2})$$

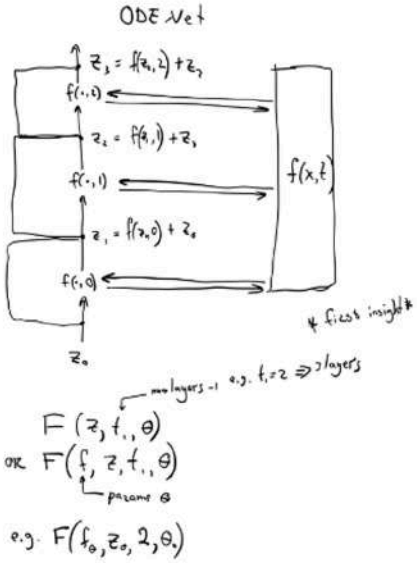
$$k_3 = hF^\theta(\tilde{z}(t_i) + \frac{k_2}{2}), \quad k_4 = hF^\theta(\tilde{z}(t_i) + k_3)$$

$$\tilde{z}(t_{i+1}) = \tilde{z}(t_i) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- Black-box/ adaptive integrators: Neural ODE (Chen et al., 2018

$$\tilde{z}(t_{i+1}) = \text{ODESolve}(\tilde{z}(t_i), F^\theta; h)$$





Adjoint method

Consider we are dealing with a constrained optimization problem

$$\min_{\mathbf{p}} g(\mathbf{u}(\mathbf{p}), \mathbf{p})$$

where $\mathbf{u}(\mathbf{p}) \in \mathbb{R}^N$ is the solution to some equation

$$\mathbf{f}(\mathbf{u}(\mathbf{p}), \mathbf{p}) = \mathbf{0}$$

If we wish to optimize g using some gradient-type algorithm, we need an efficient way to compute the derivative of g w.r.t. $\mathbf{p} \in \mathbb{R}^M$:

$$\frac{dg}{d\mathbf{p}} = \frac{\partial g}{\partial \mathbf{p}} + \frac{\partial g}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{p}} \tag{1}$$

The adjoint method is a general procedure to reduce the computation of $\frac{dg}{d\mathbf{p}}$ to solving one adjoint equation.

- $\left(\mathbf{c}^T \mathbf{A}^{-1}\right) \frac{\partial \mathbf{b}}{\partial \mathbf{p}}$. This results in a computational cost of $O(N(N+M))$.

- $\mathbf{c}^T \left(\mathbf{A}^{-1} \frac{\partial \mathbf{b}}{\partial \mathbf{p}}\right)$. This results in a computational cost of $O(NM(1+N))$.

Clearly, the first method is more efficient as $M > 1$.

Summarizing, we can re-formulate the above as

$$\frac{\partial g}{\partial \mathbf{p}} = \boldsymbol{\lambda}^T \frac{\partial \mathbf{b}}{\partial \mathbf{p}}$$

where $\boldsymbol{\lambda}$ is the solution to

$$\mathbf{A}^T \boldsymbol{\lambda} = \mathbf{c}$$

This is basically the main idea of the adjoint method. Notice that, in fact, we moved to solving a linear system whose matrix is the adjoint of the original linear system (2).

Example: A linear problem

Consider the case of a linear function g

$$g(\mathbf{u}) = \mathbf{c}^T \mathbf{u}$$

where $\mathbf{u}(\mathbf{p})$ is the solution to a (parametric) linear system

$$\mathbf{A}\mathbf{u}(\mathbf{p}) = \mathbf{b}(\mathbf{p}) \tag{2}$$

where $\mathbf{b} : \mathbb{R}^M \to \mathbb{R}^N$ and $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a fixed matrix. $\mathbf{u}(\mathbf{p})$ is the solution to the system and is thus given by $\mathbf{u}(\mathbf{p}) = \mathbf{A}^{-1}\mathbf{b}(\mathbf{p})$. If we derive $\mathbf{u}$ w.r.t. to $\mathbf{p}$, we get

$$\frac{\partial \mathbf{u}}{\partial \mathbf{p}} = \mathbf{A}^{-1} \frac{\partial \mathbf{b}}{\partial \mathbf{p}}$$

Plugging this into equation (1), we get

$$\frac{dg}{d\mathbf{p}} = \frac{\partial g}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{p}} = \mathbf{c}^T \mathbf{A}^{-1} \frac{\partial \mathbf{b}}{\partial \mathbf{p}} \tag{3}$$

Recall that $\mathbf{c}$ is a N dimensional vector, $\mathbf{A}$ is a $N \times N$ matrix, and $\frac{\partial \mathbf{b}}{\partial \mathbf{p}}$ is a $N \times M$ matrix. There are two possible ways to compute the matrix product in the RHS of equation (3):

Neural ODE and backpropagation

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta) dt\right) = L\left(\text{ODESolve}\left(\mathbf{z}(t_0), f, t_0, t_1, \theta\right)\right)$$

- To optimize L, we require gradients with respect to θ.
- The first step is to determining how the gradient of the loss depends on the hidden state $\mathbf{z}(t)$ at each instant. This quantity is called the adjoint and given as
- $\mathbf{a}(t) = \partial L / \partial \mathbf{z}(t)$.
- Its dynamics are given by another ODE, which can be thought of as the instantaneous analog of the chain rule:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^\top \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}$$

- We can compute $\partial L / \partial \mathbf{z}(t_0)$ by another call to an ODE solver. This solver must run backwards, starting from the value of $\partial L / \partial \mathbf{z}(t_1)$.
- One complication is that solving this ODE requires the knowing value of $\mathbf{z}(t)$ along its entire trajectory. However, we can simply recompute $\mathbf{z}(t)$ backwards in time together with the adjoint, starting from its final value $\mathbf{z}(t_1)$.

Algorithm Complete reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_1)}^\top f(\mathbf{z}(t_1), t_1, \theta)$$ ▷ Compute gradient w.r.t. t_1

$$s_0 = \left[\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}, -\frac{\partial L}{\partial t_1}\right]$$ ▷ Define initial augmented state

def aug_dynamics $([\mathbf{z}(t), \mathbf{a}(t), \cdot, \cdot], t, \theta)$: ▷ Define dynamics on augmented state

return $\left[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\top \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\top \frac{\partial f}{\partial \theta}, -\mathbf{a}(t)^\top \frac{\partial f}{\partial t}\right]$ ▷ Compute vector-Jacobian products

$$\left[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}\right] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷ Return all gradients

Adjoint Proof

Let $\mathbf{z}(t)$ follow the differential equation $\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta)$, where θ are the parameters. We will prove that if we define an adjoint state

$$\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$$

then the adjoint state satisfies the differential equation

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{4}$$

For ease of notation, we denote vectors as row vectors, whereas the main text uses column vectors. The adjoint state is the gradient with respect to the hidden state at a specified time t. In standard neural networks, the gradient of a hidden layer $\mathbf{h}_t$ depends on the gradient from the next layer $\mathbf{h}_{t+1}$ by chain rule

$$\frac{dL}{d\mathbf{h}_t} = \frac{dL}{d\mathbf{h}_{t+1}} \frac{d\mathbf{h}_{t+1}}{d\mathbf{h}_t}$$

With a continuous hidden state, we can write the transformation after an ε change in time as

$$\mathbf{z}(t+\varepsilon) = \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta) dt + \mathbf{z}(t) = T_\varepsilon(\mathbf{z}(t), t)$$

and chain rule can also be applied

$$\frac{dL}{d\mathbf{z}(t)} = \frac{dL}{d\mathbf{z}(t+\varepsilon)} \frac{d\mathbf{z}(t+\varepsilon)}{d\mathbf{z}(t)} \quad \text{or} \quad \mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \frac{\partial T_\varepsilon(\mathbf{z}(t), t)}{\partial \mathbf{z}(t)}$$

The proof follows from the definition of derivative:

$$\begin{aligned}
\frac{d\mathbf{a}(t)}{dt} &= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \frac{\partial}{\partial \mathbf{z}(t)} T_\varepsilon(\mathbf{z}(t))}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \frac{\partial}{\partial \mathbf{z}(t)} \left(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \left(I + \varepsilon \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon \mathbf{a}(t+\varepsilon) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} -\mathbf{a}(t+\varepsilon) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon) \\
&= -\mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}
\end{aligned}$$

We point out the similarity between adjoint method and backpropagation. Similarly to backpropagation, ODE for the adjoint s needs to be solved backwards in time. We specify the constraint on the last time point, which is simply the gradient of the loss the last time point, and can obtain the gradients with respect to the hidden state at any time, including the initial value.

$$\underbrace{\mathbf{a}(t_N) = \frac{dL}{d\mathbf{z}(t_N)}}_{} \qquad \underbrace{\mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0} \frac{d\mathbf{a}(t)}{dt} dt = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} dt}_{}$$

Know F, solve the following equations for z

$$\sum_{m=0}^{M} \alpha_m z(t_{n-m}) = h \sum_{m=0}^{M} \beta_m F(z(t_{n-m}))$$

How to choose α and β?
We have 3 families:

- Adams-Bashforth(AB)
- Adams-Moulton (AM)
- Backwards Differentiation Formula (BDF)

$$z(t_n) \approx z(t_{n-1}) + \int_{t_{n-1}}^{t_n} \sum_{k \in \Lambda} F(t_{n+k}) l_{k,n}(t; \Lambda) dt$$

where,

$$l_{k,n}(t; \Lambda) = \prod_{i \in \Lambda \setminus \{k\}} \frac{t - t_{n+i}}{t_{n+k} - t_{n+i}}, k \in \Lambda$$

If $\Lambda_0 = \{-M+1, -M+2, ..., -1, 0\}$, we get AB
If $\Lambda_1 = \{-M+2, -M+3, ..., 0, 1\}$, we get AM
BDF:

$$\sum_{k \in \Lambda} z(t_{n+k}) \frac{dl_{k,n}}{dt}(t_n; \Lambda_1) \approx \frac{d}{dt} z(t_n) = F(z(t_n))$$

Consider the IVP: $\frac{dU}{dt} = \mathcal{F}(U, t)$, where $U(t=0) = U_0$

Examples:

- 2nd -order Adams-Bashforth: $\frac{U^{n+1} - U^n}{\Delta t} = \frac{3}{2}\mathcal{F}^n - \frac{1}{2}\mathcal{F}^{n-1}$
- 3rd -order Adams-Bashforth: $\frac{U^{n+1} - U^n}{\Delta t} = \frac{23}{12}\mathcal{F}^n - \frac{16}{12}\mathcal{F}^{n-1} + \frac{5}{12}\mathcal{F}^{n-2}$
- 3rd -order Backwards differentiation:

$$\frac{U^{n+1} - \frac{1}{11}\left(18U^n - 9U^{n-1} + 2U^{n-2}\right)}{\Delta t} = \frac{6}{11}\mathcal{F}^{n+1}, \text{ implicit}$$

- 3rd -order Adams-Moulton method:

$$\frac{U^{n+1} - U^n}{\Delta t} = \frac{5}{12}\mathcal{F}^{n+1} + \frac{8}{12}\mathcal{F}^n - \frac{1}{12}\mathcal{F}^{n-1}$$

Typically, the coefficients of the Adams-Moulton (AM) methods are smaller than the coefficients of the Adams-Bashforth (AB) method, which means lower truncation errors and round-off. Also, for the same accuracy AM uses fewer points but it is implicit.

General stiffly stable schemes (due to GEAR):

$$U^{n+1} = \sum_{j=0}^{k} \alpha_{j+1} U^{n-j} + \Delta t \beta_0 \mathcal{F}^{n+1}$$

k	2	3	4
β_0	2/3	6/11	12/25
α_1	4/3	18/11	48/25
α_2	−1/3	−9/11	−36/25
α_3	-	2/11	16/25
α_4			−3/25

Consistency and stability of AB, AM and BDF

Definition
A linear multistep method is consistent with the differential equation for dynamics discovery provided $\|\tau_h\|_\infty \to 0$ as $h \to 0$, and it is strongly consistent if $\|\tau_h\|_1 \to 0$ as $h \to 0$.

Definition
A linear M-step method for the dynamics discovery is called stable if there exists a constant $K < \infty$, not depending on N, such that, for any two grid functions u, v, we have

$$\|u - v\|_1 \le K \left(\max_{1 \le i \le M-1} \|u_i - v_i\|_\infty + \left\| \hat{R}_h(u - v) \right\|_1 \right)$$

Definition
A linear M-step method for the dynamics discovery is called strongly stable if there exists a constant $K < \infty$, not depending on N, such that, for any two grid functions u, v, we have

$$\|u - v\|_\infty \le K \left(\max_{1 \le i \le M-1} \|u_i - v_i\|_\infty + \left\| \hat{R}_h(u - v) \right\|_\infty \right)$$

Theorem
If a linear multistep method is strongly consistent and stable, or consistent and strongly stable, then it is convergent.

Pf:

$$\begin{aligned} \|\boldsymbol{f} - \hat{\boldsymbol{f}}\|_W &\le K_W \left(\max_{0 \le i \le M-1} \left\| \boldsymbol{f}_i - \hat{\boldsymbol{f}}_i \right\|_\infty + \left\| R_h \boldsymbol{f} - R_h \hat{\boldsymbol{f}} \right\|_W \right) \\ &\le K_W \left(o(1) + \|\boldsymbol{\tau}_h\|_W \right) \end{aligned}$$

Let's use this theorem to see if AB, AM, BDF are convergent for inverse problem.

Implementation of Multistep Neural Network: Problem Setup

- **Multistep Neural Network**: We define the ODE for a non-linear dynamic system:

$$\frac{\mathrm{d}}{dt}\boldsymbol{x}(t) = \boldsymbol{f}(\boldsymbol{x}(t)),$$

where $\boldsymbol{x}(t) \in \mathbb{R}^D$ denotes the state of system at time t and function $\boldsymbol{f}$ describes the evolution of the system.

- **Goal**: Given noisy measurements of the state $\boldsymbol{x}(t)$ of the system at several time instances $t_1, t_2, \ldots, t_N$, our goal is to determine the function $\boldsymbol{f}$ consequently discover the underlying dynamical system from data.
- The general form of a linear multistep method with M steps is

$$\sum_{m=0}^{M} [\alpha_m \boldsymbol{x}_{n-m} + \Delta t \beta_m \boldsymbol{f}(\boldsymbol{x}_{n-m})] = 0, \qquad n = M, \ldots, N,$$

where $\boldsymbol{x}_{n-m}$ is state of the system $\boldsymbol{x}(t_{n-m})$ at time t_{n-m}. Different choices for the parameters α_m and β_m result in specific schemes. For example, trapezoidal rule

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} + \frac{1}{2}\Delta t(\boldsymbol{x}_n + \boldsymbol{f}(\boldsymbol{x}_{n-1})), \qquad n = 1, \ldots, N.$$

- The above corresponds to the case where $M = 1$, $\alpha_0 = -1$ $\alpha_1 = 1$ and $\beta_0 = \beta_1 = 0.5$. We put neural netowork prior on $\boldsymbol{f}$. The parameters of neural network can be learned by optimizing the mean squared loss error function defined as:

$$MSE := \frac{1}{N - M + 1} \sum_{n=M}^{N} |\boldsymbol{y}_n|^2,$$

where $\boldsymbol{y}_n := \sum_{m=0}^{M} [\alpha_m \boldsymbol{x}_{n-m} + \Delta t \beta_m \boldsymbol{f}(x_{n-m})]$

Neural ODE, PNN, SympNet

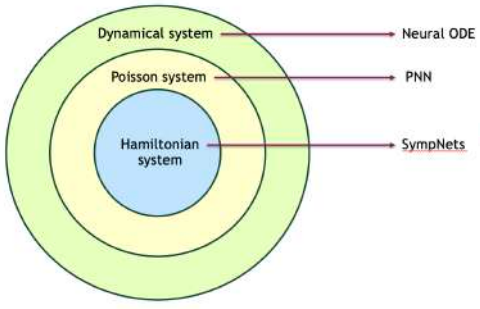


Figure: Neural ODE, PNN and SympNet. Prior knowledge used: SympNet > PNN > Neural ODE; Model expressivity: Neural ODE > PNN > SympNet.

When a model is overfitting, it has large gradient norms compared to the function that we want to reconstruct.

This intuition has been formalized in the work of Oberman et al., 2018

Idea: Force the model to have a small Lipschitz constant, while still be close to the observed data (Negrini et al., 2021).

$$\mathrm{Loss}(x, \theta) = \frac{1}{N} \sum_{i=1}^{N} \|f(x_i) - N(\theta, x_i)\|^2 + \alpha \,\mathrm{Lip}(N)$$

$$\left(\hat{R}_h \hat{f}\right)_n := -\sum_{m=0}^{M} \beta_m \hat{f}_{n-m} \qquad \mathrm{Lip}(N_{int}) = \|\nabla N_{int}\|_{L^\infty(\mathbb{R}^{d+1})}$$

Hamiltonian Systems: SympNets

Definition

A matrix $H \in \mathbb{R}^{2d \times 2d}$ is called symplectic if $H^T J H = J \qquad J := \begin{bmatrix} 0 & I_d \\ -I_d & 0 \end{bmatrix}$

Definition

A differentiable map $\phi : U \to \mathbb{R}^{2d}$ (where $U \subset \mathbb{R}^{2d}$ is an open set) is called symplectic if the Jacobian matrix $\frac{\partial \phi}{\partial x}$ is everywhere symplectic, i.e.

$$\left(\frac{\partial \phi}{\partial x}\right)^T J \left(\frac{\partial \phi}{\partial x}\right) = J$$

Approach 4: Structure-Preserving Neural Networks

The three approaches discussed so far are general purpose models. They are expressive enough to approximate arbitrary dynamical systems, but may not generalize well on specific problems.

Neural Networks with Prior Physical Knowledge

- Hamiltonian neural networks, symplectic networks
  Assume the dynamics are from a Hamiltonian system $\frac{d\tilde{z}}{dt} = J^{-1}\nabla H^\theta(\tilde{z})$
- Lagrangian neural network
  Assume the dynamics follow the Lagrangian Formalism.
- Poisson neural network
  Assume the dynamics are from a Poisson system $\frac{d\tilde{z}}{dt} = L(\tilde{z})\nabla H^\theta(\tilde{z})$
- Graph Neural ODE (Poli et al., 2019)
  Assume the RHS of the ODE have certain graph structure (works for molecular dynamics)
- GENERIC formalism informed neural network
  Assume the dynamics follow the GENERIC formalism
  $\frac{d\tilde{z}}{dt} = L(\tilde{z})\nabla E^\theta(\tilde{z}) + M(\tilde{z})\nabla S^\theta(\tilde{z})$

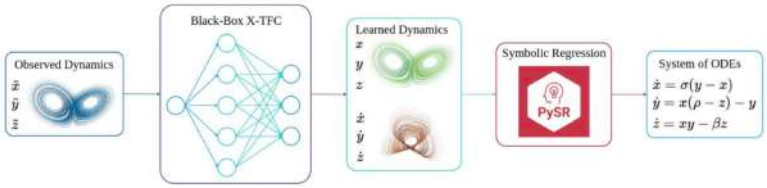
Summary

- ❑ We presented four different approaches to learn dynamical systems from data.
- ❑ Approach 1: In Neural ODE the key is to use the adjoint method for backpropagation.
- ❑ Approach 2: Use the structure of multi-step methods to learn the coefficients of the RHS from data.
- ❑ Approach 3: The recurrent neural network (RNN) family is very effective, especially the Seq2Seq models.
- ❑ Approach 4: Neural networks with prior physical knowledge or constrains can be used for Hamiltonian systems, Lagrangian systems, Poisson systems, and general systems by imposing constraints on the structure, e.g. the symplectic structure or the GENERIC formalism of non-equilibrium thermodynamics, which will lead to better generalization.
- ❑ Lipschitz regularization of deep neural networks enhances convergence and generalization.

Distillation of NN: $\begin{cases} \frac{dx}{dt} = \sigma(y - x) = NN(x, y, t, \theta, \sigma) \\ \frac{dy}{dt} = x(\rho - z) - y = NN(x, y, z, t, \theta, \rho) \\ \frac{dz}{dt} = xy - \beta z = NN(x, z, t, \theta, \rho) \end{cases}$ given $x(t=0) = x_0$, $y(t=0) = y_0$, $z(t=0) = z_0$

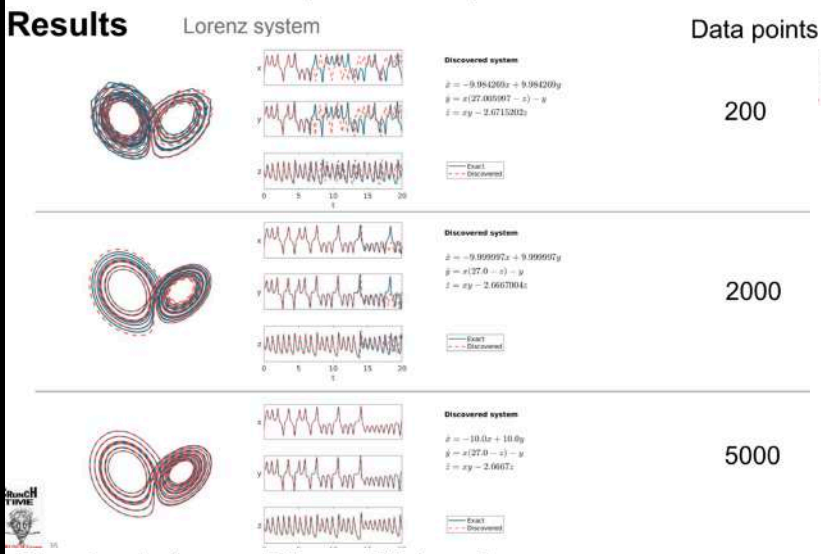
Symbolic Regression: (ML) Fit model to predict y given x, $\{(y, x)_i\}_i^N$, $y \approx f(x_1, x_2, x_3, \ldots)$
$f \in$ Analytic equations

AI-Lorenz framework



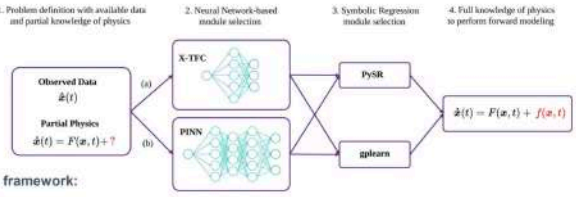
The observed dynamics are used to train the neural network in the black-box X-TFC framework, which learns the dynamics and their rates of change in time. The latter are used as input in the symbolic regression algorithm (PySR), which autonomously distills the explicit mathematical terms that build the system of differential equations.

Results



Gray-Box model Identification
AI-Aristotle

Daryakenari, Nazanin Ahmadi, Mario De Florio, Khemraj Shukla, and George Em Karniadakis. "AI-Aristotle: A Physics-Informed framework for Systems Biology Gray-Box Identification." *PLoS computational biology* (2024).

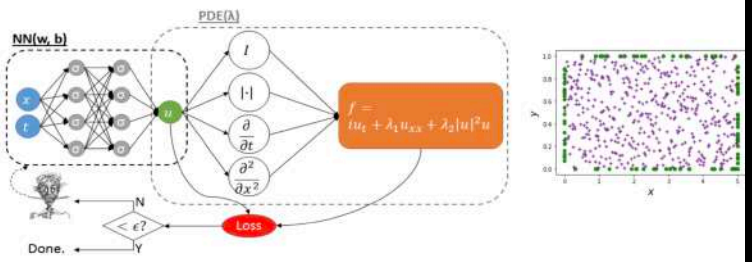


AI-Aristotle framework:

1. The observed data and the partial knowledge of physics are used to train the selected NN-based module.
2. The selection of the NN-based module needs to be done between **(a)** X-TFC, recommended for high-resolution data and missing terms discovery, and **(b)** PINN, recommended for sparse data and parameter estimation. The NN outputs are the time-dependent representations of the missing terms of the dynamical systems, which are fed into the SR algorithm.
3. The selected SR module identifies the mathematical expressions of the missing terms. It is recommended to use both symbolic regressors for cross-validation.
4. The full knowledge of physics is now available, allowing forward modeling performance.

What is a PINN? Physics-Informed Neural Network

We employ two (or more) NNs that share the same parameters



➢ Minimize:

$$MSE = MSE_u + MSE_f,$$

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2,$$

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2.$$

Physics Informed Neural Networks

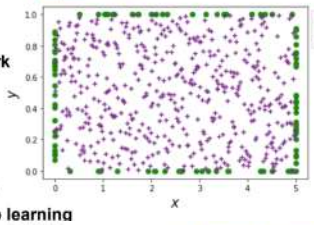
Idea: Encode "physics equations" and "data" into the neural net $h_\theta(\mathbf{x})$

- **DEF**: Let $\boldsymbol{m} = (m_r, m_b)$. The prototype PINN loss is

$$\text{Loss}_{\boldsymbol{m}}^{\text{PINN}}(\theta) = \frac{1}{m_r}\sum_{i=1}^{m_r}\Big(\underbrace{\mathcal{L}[h_\theta](\mathbf{x}_r^i)}_{\text{physics}} - \underbrace{f(\mathbf{x}_r^i)}_{\text{data}}\Big)^2 + \frac{1}{m_b}\sum_{i=1}^{m_b}\Big(\underbrace{\mathcal{B}[h_\theta](\mathbf{x}_b^i)}_{\text{physics}} - \underbrace{g(\mathbf{x}_b^i)}_{\text{data}}\Big)^2$$

- **Method**: $\underset{\theta}{\text{minimize}}\ \text{Loss}_{\boldsymbol{m}}^{\text{PINN}}(\theta) \implies \theta_{\boldsymbol{m}}^*$

  $h_{\boldsymbol{m}} := h_{\theta_{\boldsymbol{m}}^*}$ is a **physics informed neural network**



Why PINNs?

- **Simple and easy** to implement it
- Solve PDEs without **meshes**
- Great potential in **solving *inverse* problems**
- Take advantage of rapid development of **deep learning**

Approach 1: PINNs with Soft Constraints

- Loss = Boundary Losses + Residual Loses

$$\mathcal{L} = \lambda_{B_1}\mathcal{L}_{B_1} + \lambda_{B_2}\mathcal{L}_{B_2} + \lambda_f\mathcal{L}_f$$

Approach 2: Self-Adaptive PINNs

- McClenny L, Braga-Neto U. Self-adaptive physics-informed neural networks using a soft attention mechanism. arXiv preprint arXiv:2009.04544. 2020.

PDE is Defined as

$$\begin{aligned}\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] &= 0, \quad \boldsymbol{x}\in\Omega, t\in[0,T]\\ u(\boldsymbol{x},t) &= g(\boldsymbol{x},t), \quad \boldsymbol{x}\in\partial\Omega, t\in[0,T]\\ u(\boldsymbol{x},0) &= h(\boldsymbol{x}), \quad \boldsymbol{x}\in\Omega\end{aligned}$$

Self-adaptive PINN utilizes the following loss function

$$\mathcal{L}(w,\lambda_r,\lambda_b,\lambda_0) = \mathcal{L}_s(w) + \mathcal{L}_r(w,\lambda_r) + \mathcal{L}_b(w,\lambda_b) + \mathcal{L}_0(w,\lambda_0)$$

where $\lambda_r = \{\lambda_r^1,\ldots,\lambda_r^{N_r}\}$, $\lambda_b = \{\lambda_b^1,\ldots,\lambda_b^{N_b}\}$, and $\lambda_0 = \{\lambda_0^1,\ldots,\lambda_0^{N_0}\}$ are trainable selfadaptation weights for the initial, boundary, and collocation points, respectively, and

$$\mathcal{L}_r(w,\lambda_r) = \frac{1}{N_r}\sum_{i=1}^{N_r}\left[\lambda_r^i\, r\left(x_r^i,t_r^i;w\right)\right]^2$$

$$\mathcal{L}_b(w,\lambda_b) = \frac{1}{N_b}\sum_{i=1}^{N_b}\left[\lambda_b^i\left(u\left(x_b^i,t_b^i;w\right)-g_b^i\right)\right]^2$$

$$\mathcal{L}_0(w,\lambda_0) = \frac{1}{N_0}\sum_{i=1}^{N_0}\left[\lambda_0^i\left(u\left(x_0^i,0;w\right)-h_0^i\right)\right]^2$$

The key feature of self-adaptive PINNs is that the loss $\mathcal{L}(w,\lambda,\lambda_b,\lambda_0)$ is minimized with respect to the network weights w, as usual, but is maximized with respect to the self-adaptation weights $\lambda_r,\lambda_b,\lambda_0$; in other words, training seeks a saddle point

$$\min_w \max_{r,\lambda_b,\lambda_0} \mathcal{L}(w,\lambda_r,\lambda_b,\lambda_0).$$

This can be accomplished by a gradient descentascent procedure, with updates given by:

gradient descent

$$w^{k-1} = w^k - \eta_k\nabla_w\mathcal{L}\left(w^k,\lambda_r^k,\lambda_b^k,\lambda_0^k\right)$$

gradient ascent

$$\begin{aligned}\lambda_r^{k+1} &= \lambda_r^k + \eta_k\nabla_{\lambda_r}\mathcal{L}\left(w^k,\lambda_r^k,\lambda_b^k,\lambda_0^k\right)\\ \lambda_b^{k+1} &= \lambda_b^k + \eta_k\nabla_{\lambda_b}\mathcal{L}\left(w^k,\lambda_r^k,\lambda_b^k,\lambda_0^k\right)\\ \lambda_0^{k+1} &= \lambda_0^k + \eta_k\nabla_{\lambda_0}\mathcal{L}\left(w^k,\lambda_r^k,\lambda_b^k,\lambda_0^k\right)\end{aligned}$$

Approach 3: Dynamic Weights for PINNs

- Wang S, Yu X, Perdikar... PINNs fail to train: A ne... perspective. Journal of ... Physics. 2022 Jan 15;4...

- Loss = Boundary Losses + Residual Loses

$$\mathcal{L} = \lambda_B\mathcal{L}_B + \lambda_f\mathcal{L}_f$$

- Parameters update

$$\theta^{(k+1)} = \theta^{(k)} - \eta\nabla_\theta\mathcal{L}_f - \eta\alpha^{(k+1)}\nabla_\theta\mathcal{L}_B,$$

where $\alpha^{(k+1)} = (1-\lambda)\alpha^{(k)} + \lambda\hat{\alpha}^{(k+1)}$, with $\hat{\alpha}^{(k+1)} = \dfrac{\overline{|\nabla_\theta\mathcal{L}_f|}}{|\nabla_\theta\mathcal{L}_B|}$,

- $\overline{|\nabla_\theta\mathcal{L}_B|}$ denote the means of $|\nabla_\theta\mathcal{L}_B|$.
- $0 \le \lambda \le 1$.

Loss function

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w}) + \mathcal{L}_0(\boldsymbol{w})$$

$$\mathcal{L}_s(\boldsymbol{w}) = \frac{1}{N_s}\sum_{i=1}^{N_s}\left|\underbrace{u\left(\boldsymbol{x}_s^i,t_s^i;\boldsymbol{w}\right)}_{\text{physics}} - \underbrace{y_s^i}_{\text{data}}\right|^2$$

$$\mathcal{L}_r(\boldsymbol{w}) = \frac{1}{N_r}\sum_{i=1}^{N_r}\underbrace{r\left(\boldsymbol{x}_r^i,t_r^i;\boldsymbol{w}\right)}_{\text{physics}}^2$$

$$\mathcal{L}_b(\boldsymbol{w}) = \frac{1}{N_b}\sum_{i=1}^{N_b}\left|\underbrace{u\left(\boldsymbol{x}_b^i,t_b^i;\boldsymbol{w}\right)}_{\text{physics}} - \underbrace{g_b^i}_{\text{data}}\right|^2$$

$$\mathcal{L}_0(\boldsymbol{w}) = \frac{1}{N_0}\sum_{i=1}^{N_0}\left|\underbrace{u\left(\boldsymbol{x}_0^i,0;\boldsymbol{w}\right)}_{\text{physics}} - \underbrace{h_0^i}_{\text{data}}\right|^2$$

Setup for PINNs

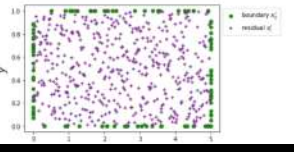
Consider the partial differential equations (PDEs):

$$\begin{aligned}\mathcal{L}[u](\mathbf{x}) &= f(\mathbf{x}) \text{ in } U\subset\mathbb{R}^d\\ \mathcal{B}[u](\mathbf{x}) &= g(\mathbf{x}) \text{ in } \partial U\end{aligned}$$

Goal: Find a neural net $h_\theta(\mathbf{x})$ that approximates the PDE solution.

What are known to us?

- Pointwise PDE data: i.i.d. samples
- Residual data: $\left(\mathbf{x}_r^i, f\left(\mathbf{x}_r^i\right)\right)$ where $\mathbf{x}_r^i \in U$
- Boundary data: $\left(\mathbf{x}_b^i, g\left(\mathbf{x}_b^i\right)\right)$ where $\mathbf{x}_b^i \in \partial U$
- Physics Equations: $\mathcal{L}[u] - f = 0$ and $\mathcal{B}[u] - g = 0$



Physics-Informed Neural Networks

Idea: Encode "physics equations" and "data" into the neural net $h_\theta(\mathrm{x})$

Physics

Data

DEF: Let $\boldsymbol{m} = (m_r, m_b)$. The prototype PINN loss is

$$\mathrm{Loss}_m^{\mathrm{PINN}}(\theta) = \frac{1}{m_r}\sum_{i=1}^{m_r}\left(\mathcal{L}\left[h_\theta\right](\mathbf{x}_r^i) - f(\mathbf{x}_r^i)\right)^2 + \frac{1}{m_b}\sum_{i=1}^{m_b}\left(\mathcal{B}\left[h_\theta\right](\mathbf{x}_b^i) - g(\mathbf{x}_b^i)\right)^2$$

DEF: The Hölder regularized PINN loss is

$$\mathrm{Loss}_m^{\mathrm{Höld}}(\theta) = \mathrm{Loss}_m^{\mathrm{PINN}}(\theta) + \lambda_r^R\left([\mathcal{L}\left[h_\theta\right]]_{\alpha;U}\right)^2 + \lambda_b^R\left([\mathcal{B}\left[h_\theta\right]]_{\alpha;\partial U}\right)^2$$

Hölder constant → $[u]_{\alpha;U} = \sup_{x\neq y\in U}\frac{\|u(x)-u(y)\|}{\|x-y\|^\alpha}$

Goal: $\underset{\theta}{\text{minimize}}\,\mathrm{Loss}_m^{\mathrm{Höld}}(\theta) \Longrightarrow \theta_m^*$

$h_m := h_{\theta_m^*}$ is a physics informed neural network $\Longrightarrow \{h_m\}$

Q: Does $\{h_{\boldsymbol{m}}\}$ converge to the PDE solution? In what norm?

Hölder Regularization: Motivation

Ideally, we wish to minimize the expected PINN loss :

$$\mathrm{Loss}_\infty^{\mathrm{PINN}}(h) = \mathbb{E}\left[\mathrm{Loss}_m^{\mathrm{PINN}}(h)\right]$$

Theorem: (Shin, Darbon, Karniadakis, 20) Suppose iid samples. Then, with high probability over iid samples,

$$\mathrm{Loss}_\infty^{\mathrm{PINN}}(\theta) \le C_m\,\mathrm{Loss}_m^{\mathrm{Höld}}(\theta) + C'\left(m_r^{-\frac{\alpha}{d}} + m_b^{-\frac{\alpha}{d-1}}\right)$$

Expected PINN loss — Hölder Regularized empirical loss

- $C_m = \Theta\left(\max\left\{\sqrt{m_r}, \sqrt{m_b}\right\}\right)$
- C' is a universal constant
- $\lambda^R = (\lambda_r^R, \lambda_b^R)$, $\quad \lambda_r^R = \Theta\left(C_m^{-1} m_r^{-\alpha/d}\right)$, $\quad \lambda_b^R = \Theta\left(C_m^{-1} m_b^{-\alpha/(d-1)}\right)$.

Summary

- PINN is simple and can be implemented in a few lines of code for any PDE
- Boundary conditions need special treatment and can be implemented via self-adaptive or dynamic weights
- Weighted residual methods can be implemented following a Petrov-Galerkin projection
- Domain decomposition can enhance the accuracy and flexibility of PINNs
- PINN converges for boundary values problems even in the absence of BCs
- Several works have obtained a priori and a posteriori errors for PINNs
- There are three main errors: Approximation, Generalization and Optimization
- PINN is more accurate and faster that the Deep Ritz Method

Hard Constraints: Dirichlet BC and IC

Dirichlet BC or IC : $\quad u_i(\mathbf{x}) = g_0(\mathbf{x}), \quad \mathbf{x}\in\Gamma_D,$

Trial solution: $\quad \hat{u}_i(\mathbf{x};\boldsymbol{\theta}_u) = g(\mathbf{x}) + \ell(\mathbf{x})\mathcal{N}(\mathbf{x};\boldsymbol{\theta}_u),$

$g(x)$ is a continuous extension of $g_0(x)$ from Γ_D to Ω where $\mathcal{N}(\mathbf{x};\boldsymbol{\theta}_u)$ is the network output, and ℓ is a function satisfying the following two conditions:

$$\begin{cases}\ell(\mathbf{x}) = 0, & \mathbf{x}\in\Gamma_D\\ \ell(\mathbf{x}) > 0, & \mathbf{x}\in\Omega-\Gamma_D\end{cases}$$

For example: $\quad u(0)=0, u(1)=1 : g(x) = x, \ell(x) = x(1-x)$