Neural Networks

AERO 689: Machine Learning for Aerospace Engineers

Raktim Bhattacharya

Texas A&M University

Table of contents

Why Neural Networks?

Neural Networks

Activation Functions

Building Neural Networks in Python

Universal Approximation Theory

Impact of Weights and Biases

Deep Neural Networks

Specific Deep Learning Architectures

Other Architrectures

Physics-Informed Neural Networks

Summary

Why Neural Networks?

Motivation

Within data-driven ML, when should we use neural networks over classical methods?

Classical methods (Weeks 2–6): linear/regularized regression, logistic regression, PCA, POD, clustering

- These methods are **not inferior** — for many problems they are the right choice
- Ridge regression outperforms a NN on a 50-sample aerodynamic dataset
- Neural networks earn their place only when the problem **exceeds what the classical toolkit handles gracefully**

What Classical ML Cannot Do Easily

Limitation	Why it matters in aerospace
Fixed feature map — hand-engineering required	Wing pressure field: $O(10^5)$ DOF — no human can specify the right features
Shallow representation — hierarchical structure is expensive	Aero-acoustic noise: vortex roll-up $\rightarrow$ shear layer $\rightarrow$ far-field radiation
Does not scale — $n \times n$ matrices become intractable	Hyperspectral imagery, LiDAR, full-field strain maps
No structure sharing — inputs treated independently	Defect detection: pixel neighborhoods carry info a flat vector discards
No sequence memory — samples treated as i.i.d.	Flight dynamics, fault evolution over time, ATC commands

The Neural Network's Answer

1. Automatic feature learning — first layers extract low-level features; deeper layers compose abstractions. No domain expert needed to specify them.

2. Depth and compositionality [1], [2] — deep networks are *compact*: exponentially more efficient than shallow models for hierarchical functions.

3. Inductive biases that match physical structure

- **CNNs** — local, translation-invariant patterns (defect detection, pressure grids)
- **RNNs / LSTMs** — sequential data (flight dynamics, engine health trending)
- **GNNs** — unstructured meshes and sensor networks

4. Scaling laws [3] — classical $O(n^2)$–$O(n^3)$ methods are impractical for large n; NNs with SGD scale to billions of samples.

Why NNs Scale: SGD

Classical methods require the full dataset simultaneously:

Method	Cost per solve	Memory
Least squares	$O(np^2 + p^3)$	$O(np)$
Kernel / GP	$O(n^3)$	$O(n^2)$

SGD replaces the full gradient with a mini-batch estimate ($|\mathcal{B}| = m \ll n$):

$$\nabla_\theta \mathcal{L} \approx \frac{1}{m} \sum_{i \in \mathcal{B}} \nabla_\theta \, \ell(f_\theta(x_i),\, y_i)\,, \qquad \text{cost per step: } O(m \cdot p)$$

- Cost per update is **constant** in n — same whether $n = 10^4$ or 10^{10}
- NN loss **decomposes** as a sum over samples $\Rightarrow$ per-sample gradients are independent
- Backprop computes $\nabla_\theta \ell$ in $O(p)$ — no matrix inversion, no kernel matrix
- Mini-batch noise acts as **implicit regularization** (helps escape sharp minima)

Limitations: Optimization and Constraints

Optimization

- Loss is **non-convex** — SGD finds local minima; no global optimality guarantee
- Convergence is not guaranteed; sensitive to learning rate and initialization
- Classical methods (ridge, SVM, GP) solve *convex* problems with guaranteed global optima

Constraints

- No native constraint handling — classical methods satisfy $g(x) = 0$ exactly via KKT / Lagrange multipliers
- NNs must impose constraints via:
  - **Architecture** (hard) — output layer designed to enforce bounds or symmetry; exact but inflexible
  - **Penalty terms** (soft) — add $\lambda\, g(\theta)^2$ to the loss; satisfied only *approximately*; the weight λ requires tuning

Limitations: Data, UQ, and Hyperparameters

Data and compute

- **Data hungry** — classical methods generalize with $n \sim 10^2$; NNs typically need $n \gtrsim 10^4$
- Training large models requires GPUs and wall-clock time that classical solvers do not

Interpretability and uncertainty quantification

- **Black box** — individual weights carry no physical meaning; predictions are hard to audit
- **No built-in uncertainty** — linear regression gives exact confidence intervals; NNs require ensembles, MC dropout, or full Bayesian NNs to produce calibrated uncertainty estimates

Hyperparameters

- Architecture depth/width, learning rate, batch size, regularization strength, and optimizer must all be tuned
- No closed-form selection — typically requires cross-validation or neural architecture search (NAS)

When to Prefer Classical ML

Prefer classical methods when:

- **Small dataset** ($n \lesssim 10^3$): Ridge/Lasso generalizes better and gives exact confidence intervals
- **Smooth, low-dimensional response**: splines or polynomial regression — interpretable and sufficient
- **Interpretability required**: linear models / decision trees produce auditable rules
- **Tabular data with engineered features**: gradient boosted trees (XGBoost) consistently outperform NNs
- **Closed-form statistics needed**: linear regression gives exact confidence intervals; NNs require ensembles or dropout for uncertainty

Rule of Thumb

Use a neural network when **at least one** of the following holds:

1. Input is **raw and high-dimensional** — images, time series, fields, point clouds
2. Function is **highly nonlinear and hierarchical** — cannot be captured by a fixed kernel
3. **Sufficient data** — $n \gtrsim 10^4$, or transfer learning from a pretrained model is available
4. **Temporal or spatial structure** should be exploited automatically (RNN/LSTM, CNN)

Otherwise, start with the simplest model that works.

Neural Networks

A Neural Network as a Function

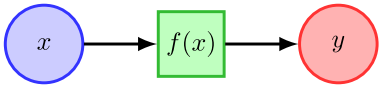


Figure 1: Simple: $y := f(x)$

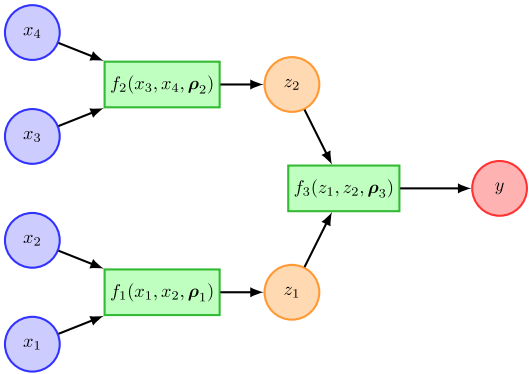
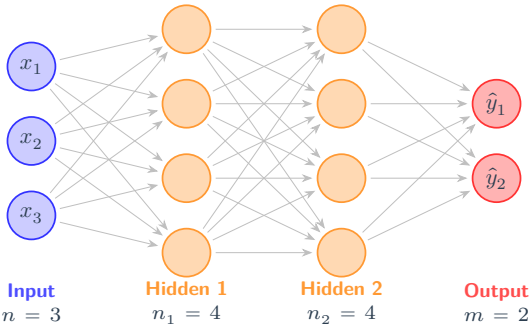


Figure 2: Complex: $y = f_3(f_1(\cdot),\ f_2(\cdot),\ \rho_3)$

In general: $y = f(x, \rho)$ where $x \in \mathcal{R}^n$ is the input and $\rho \in \mathcal{R}^p$ are the parameters.

Dense Feedforward Network

Every neuron is connected to every neuron in the next layer — hence *dense*.



Layer Operations

Each layer ℓ computes the **vector of activations**:

$$\underbrace{a^{(\ell)}}_{\text{output of layer } \ell} = \sigma\Big(W^{(\ell)}\, a^{(\ell-1)} + b^{(\ell)}\Big)$$

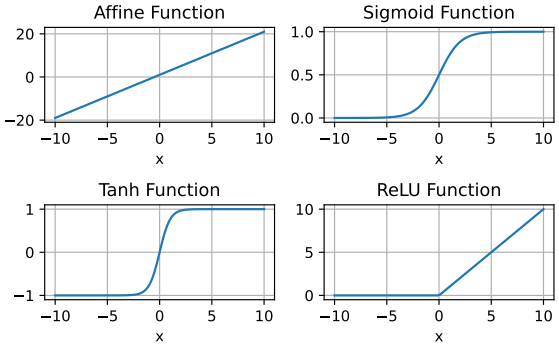
- $W^{(\ell)}$ — weight matrix; $b^{(\ell)}$ — bias vector
- σ — **activation function**, applied element-wise (broadcasting)
- "Activation of a vector" = "vector of activations" — same computation, different perspective
- The network is a *graphical architecture for composing parameterized nonlinear functions*

Activation Functions

Common Activation Functions

Name	Function
Affine	$W^T x + b$
Sigmoid	$\frac{1}{1+e^{-x}}$
Tanh	$\tanh(x)$
ReLU	$\max(0, x)$

Non-linear activations are essential: a composition of linear functions is still linear.



Building Neural Networks in Python

Framework Overview

Library	Backend	Primary use	By
TensorFlow / Keras	TF	Production, deployment	Google
PyTorch	PyTorch	Research, custom arch	Meta
scikit-learn MLP	NumPy	Rapid prototyping	Community

In practice: TensorFlow/Keras and PyTorch dominate.

TensorFlow / Keras

```
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense

fhat = Sequential([
    Dense(2,  input_dim=1),
    Dense(10, activation="tanh"),
    Dense(5,  activation="tanh"),
    Dense(1,  activation="sigmoid"),
])

# Reverse-mode AD
x = tf.linspace(-10, 10, 100)
with tf.GradientTape() as tape:
    tape.watch(x)
    y = fhat(x)
dydx = tape.gradient(y, x)
```

PyTorch: Model Definition

```
import torch
import torch.nn as nn

class FHat(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 2),
            nn.Linear(2, 10), nn.Tanh(),
            nn.Linear(10, 5), nn.Tanh(),
            nn.Linear(5, 1),  nn.Sigmoid(),
        )
    def forward(self, x):
        return self.net(x)

fhat = FHat()
```

PyTorch: Automatic Differentiation

```
x    = torch.linspace(-10, 10, 100,
                      requires_grad=True).unsqueeze(1)
y    = fhat(x)
y.sum().backward()       # dy/dx for all x simultaneously
dydx = x.grad
```

	TensorFlow / Keras	PyTorch
Training loop	`model.fit()` (hidden)	Explicit `loss.backward()`
Flexibility	High-level, concise	Fully customizable
Best for	Production / deployment	Research / custom arch

Universal Approximation Theory

Universal Approximation Theorem

Theorem (Cybenko [4]; Hornik [5]). A feedforward network with one hidden layer and finitely many neurons can approximate *any* continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy.

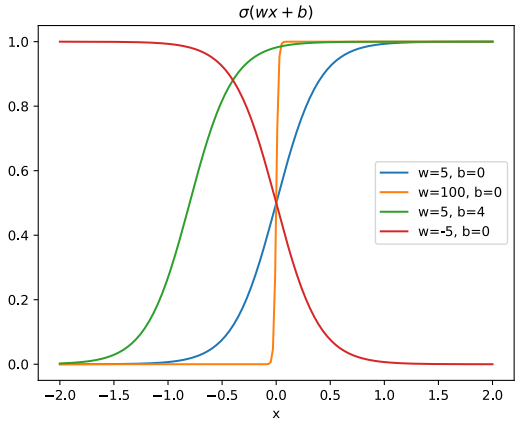
Key implications:

1. **Expressiveness** — NNs are not limited in the functions they can represent
2. **Depth efficiency** — deeper networks achieve the same accuracy with exponentially fewer parameters
3. **Non-linear activation is essential** — a composition of linear functions is still linear
4. **Does not address**: how to train, convergence speed, overfitting, or architecture choice

The theorem guarantees existence — not learnability.

Impact of Weights and Biases

Effect of Weights and Biases



Effect of w and b on $\sigma(wx + b)$.

- **Large** w, $b = 0$**:** step function — sharp transition
- **Nonzero** b**:** shifts the activation point left/right
- **Negative** w**:** deactivation — transitions from $1 \to 0$

Optimizing w and b controls the shape and location of each basis function.

Neural Basis Functions

By linearly combining sigmoid and affine functions — **locally supported functions**:

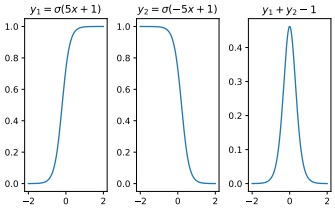


Figure 3: Bell: $y_1 + y_2 - 1$

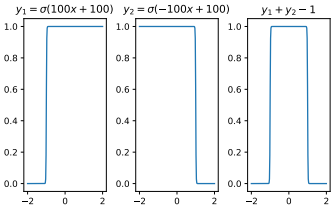


Figure 4: Box: large w

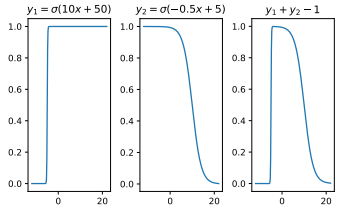


Figure 5: Asymmetric: $w_1 \neq w_2$

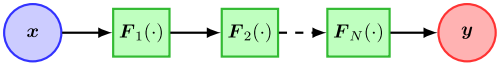
Optimizing weights and biases constructs a rich library of neural basis functions.

Deep Neural Networks

Feed-Forward Neural Network (FFNN / MLP)

$$F(x) = (F_N \circ \cdots \circ F_1)(x)$$

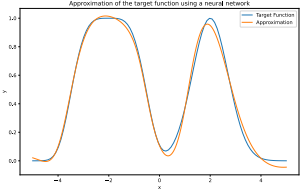
$$F_i(\cdot) = f_i(W_i\, F_{i-1}(\cdot) + b_i)$$



- W_i, b_i: weight matrix and bias of layer i
- f_i: activation function of layer i
- Well-suited for tabular data and supervised learning
- Address overfitting via weight decay or dropout
- See [6, Ch. 6] for a thorough treatment

FFNN Example: TensorFlow

```
def target_function(x):
    return np.exp(-(x-2)**2) + np.exp(-0.15*(x+2)**4)

x_train = np.linspace(-5, 5, 100).reshape(-1, 1)
y_train = target_function(x_train)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='tanh', input_shape=(1,)),
    tf.keras.layers.Dense(32, activation='tanh'),
    tf.keras.layers.Dense(32, activation='tanh'),
    tf.keras.layers.Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(x_train, y_train, epochs=1000)
```



FFNN Example: PyTorch

```
class FFNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 32), nn.Tanh(),
            nn.Linear(32, 32), nn.Tanh(),
            nn.Linear(32, 32), nn.Tanh(),
            nn.Linear(32, 1),
        )
    def forward(self, x): return self.net(x)

model     = FFNN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn   = nn.MSELoss()

for epoch in range(5000):
    optimizer.zero_grad()
    loss = loss_fn(model(x_train), y_train)
    loss.backward()
    optimizer.step()
```

Application: Aerodynamic Surrogate Modeling

Problem: Predict $\hat{C}_D(M, \alpha)$ — CFD requires hours per operating point.

Approach	Fails at	Advantage
Linear regression	Transonic drag rise ($M \approx 0.8$), stall ($\alpha > 12^\circ$)	Interpretable; fast
FFNN (2→64→64→1)	Extrapolation past training envelope	Captures strong non-linearities

Inputs: $[M, \alpha]$ **Output:** C_D **Data:** CFD sweep or wind-tunnel runs

Aerodynamic Surrogate: PyTorch

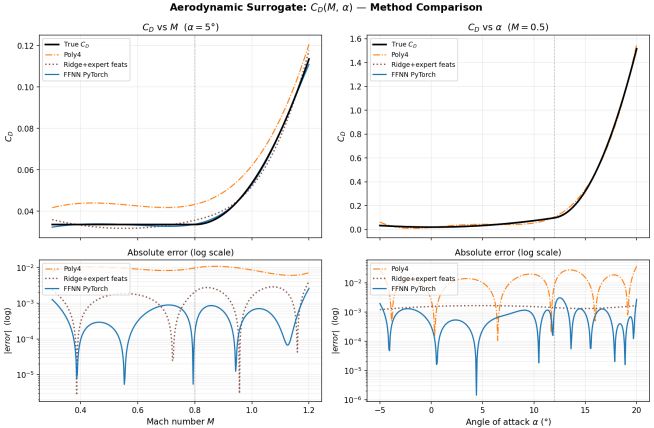
```
import torch, torch.nn as nn

class DragSurrogate(nn.Module):       # [M, alpha] -> C_D
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 1),
        )
    def forward(self, x): return self.net(x)

model     = DragSurrogate()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn   = nn.MSELoss()

# X_train: (N, 2) [M, alpha];  y_train: (N, 1) C_D
for epoch in range(5000):
    optimizer.zero_grad()
    loss = loss_fn(model(X_train), y_train)
    loss.backward()
    optimizer.step()
```

Aerodynamic Surrogate: Accuracy Results

Method	RMSE	R^2
Polynomial (deg 4)	0.0142	0.998
Ridge + expert feats	0.0027	1.000
FFNN PyTorch	**0.0024**	**1.000**

- 2,400 train / 600 test.
- $C_D \in [0.015,\ 1.52]$.

Takeaway: expert feats beat FFNN here — NNs earn their place when features are unknown.



Top: predictions; bottom: |error| (log scale). Poly4 and expert features track well; NNs match without feature engineering.

Specific Deep Learning Architectures

Sequence Prediction: Embedding the Past

Many aerospace problems are **sequential**: given a history of observations $x_1, x_2, \ldots, x_t$, predict what happens next.

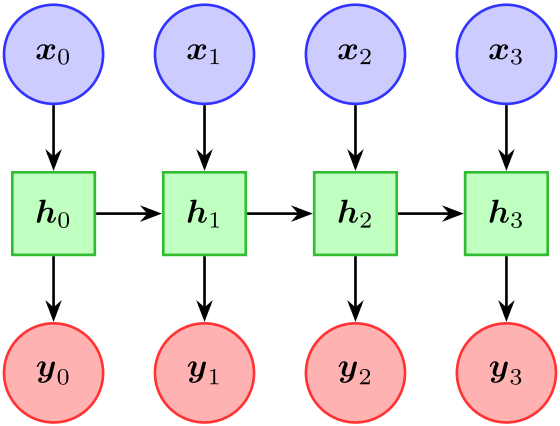
Application	Sequence (past/present)	Prediction (future)
Engine health	Sensor readings over T flights	Remaining useful life
Flight dynamics	States (α, q, V) over time	Next-step trajectory
Weather / turbulence	Gridded fields at $t_{-n} \ldots t_0$	Field at t_1

Core challenge: an FFNN takes a fixed-size input. How do we compress a variable-length sequence into a fixed representation — a **sequence embedding** — that retains the information needed for prediction?

Three approaches, each building on the last:

1. **RNN** — fold the sequence into a hidden state, one step at a time
2. **LSTM** — add gated memory so the embedding can retain long-range information
3. **Transformer** — let every step attend to every other step in parallel

Recurrent Neural Networks (RNN)



$$h_t = F(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = g(W_{hy}h_t + b_y)$$

- Parameters shared across time steps
- **Hidden state** h_t is the sequence embedding — it accumulates information from $x_1, \ldots, x_t$
- At any time t, h_t is a fixed-size summary of everything seen so far

The Vanishing Gradient Problem

Training an RNN requires back-propagation through time (BPTT): gradients flow backward through every time step.

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial h_T} \prod_{k=t+1}^{T} \frac{\partial h_k}{\partial h_{k-1}}$$

The product $\prod \frac{\partial h_k}{\partial h_{k-1}}$ involves multiplying the same weight matrix $T - t$ times:

- If eigenvalues < 1: gradients **vanish** $\Rightarrow$ early time steps are forgotten
- If eigenvalues > 1: gradients **explode** $\Rightarrow$ training diverges

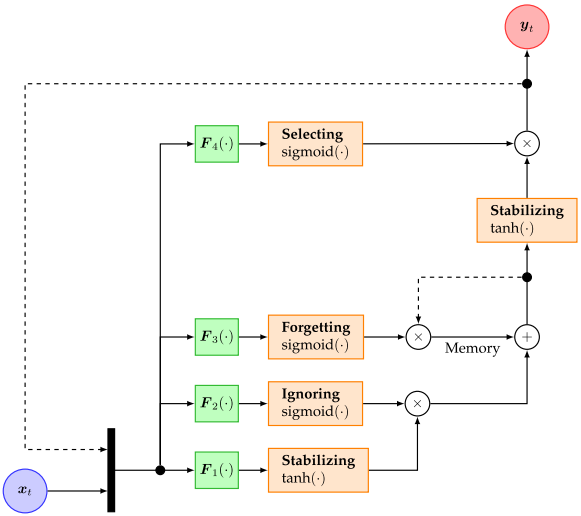
Vanishing Gradients: Consequence and Solution

Consequence: vanilla RNNs struggle to learn dependencies longer than $ $10–20 steps. The sequence embedding h_t loses early information.

Example: predicting engine failure from 200 cycles of sensor data — by the time the RNN reaches cycle 200, the gradient signal from cycle 1 has effectively disappeared.

Solution: add explicit memory with **gates** that control what to remember and what to forget $\Rightarrow$ **LSTM**.

Long Short-Term Memory (LSTM)



Three **gates** regulate memory [7]:

- **Forget gate** f_t — erases irrelevant memory
- **Input gate** i_t — writes new information
- **Output gate** o_t — controls what passes to h_t

The **cell state** c_t acts as a conveyor belt: gradients can flow through it unchanged, solving the vanishing gradient problem.

The sequence embedding is now (h_t, c_t) — richer than the RNN's h_t alone.

Periodic Forecasting Example

Goal: predict the next value of a periodic signal from a window of recent history.

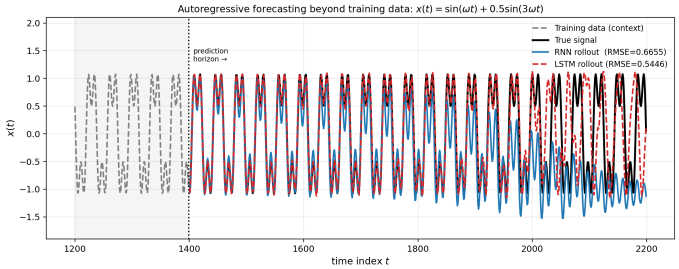
$$x_t = \sin(\omega t) + 0.5\sin(3\omega t), \quad \omega = 0.17$$

$$\hat{x}_{t+1} = f(x_{t-W+1}, \ldots, x_t), \quad W = 30$$

- Train: $t = 0, \ldots, 1399$; test rollout: $t = 1400, \ldots, 2199$
- Compare **vanilla RNN** and **LSTM** (same hidden size $= 32$)
- **Autoregressive rollout**: each prediction is fed back as the next input — no true future values used

```
def signal(t):
    return np.sin(omega * t) + 0.5 * np.sin(3 * omega * t)

X, y = make_windows(signal(t_train), window=30)

rnn  = nn.RNN(input_size=1,  hidden_size=32, batch_first=True)
lstm = nn.LSTM(input_size=1, hidden_size=32, batch_first=True)
```

Periodic Forecasting Results: RNN vs LSTM



- Gray region: last 200 training steps (context). Dashed line: end of training data.
- Both models roll out **800 steps** feeding only their own predictions — no true future values.
- Look-ahead is valid only for inputs known in advance (e.g., planned controls); using unknown future targets is leakage.
  - Valid only when future signals are genuinely available at inference time.

RNN: diverges quickly. The fixed-size hidden state h_t cannot retain phase information over many steps.

LSTM: stays accurate near the horizon, but also drifts eventually. The cell state slows error growth but cannot prevent it.

Key insight: autoregressive rollout accumulates errors at every step. Even the best recurrent model has a finite prediction horizon — this is a fundamental limitation, not just a training issue.

From Recurrence to Attention

LSTM processes the sequence **one step at a time** — step 50 must wait for steps 1–49. Two limitations:

1. **No parallelism** — training is slow on long sequences
2. **Information bottleneck** — all history must squeeze through the fixed-size state (h_t, c_t)

Key insight [8]: instead of passing information forward through a chain of hidden states, let every position in the sequence **look at every other position directly**.

Self-Attention Mechanism

Given an input sequence $x_1, \ldots, x_T$, compute three projections per position:

- $\mathbf{Q}$ (Query) — "what am I looking for?"
- $\mathbf{K}$ (Key) — "what do I contain?"
- $\mathbf{V}$ (Value) — "what information do I pass along?"

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

The softmax produces a **weight matrix** over all positions — each output is a weighted combination of all inputs, where the weights are learned from data.

Dividing by $\sqrt{d_k}$ prevents dot products from growing too large in high dimensions.

Transformer Architecture

Key ideas:

- **Self-attention** — each output is an attention-weighted sum of *all* inputs
- **Multi-head attention** — h parallel heads learn different relationships simultaneously
- **Positional encoding** — injects sequence order (no recurrence)
- Scales to long sequences; backbone of modern LLMs and vision models

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Watch: 3Blue1Brown

Attention in transformers, visually explained

https://www.youtube.com/watch?v=eMlx5fFNoYc

Self-study: Watch before the next class. We will discuss aerospace applications in lecture.

Sequence Models: Comparison

	RNN	LSTM	Transformer
Long-range memory	Poor	Good	Excellent
Parallel training	No	No	Yes
Sequence embedding	h_t	(h_t, c_t)	attention-weighted
Complexity per layer	$O(T \cdot d^2)$	$O(T \cdot d^2)$	$O(T^2 \cdot d)$

- **RNN:** simple, fast per step, but forgets early inputs
- **LSTM:** gated memory solves vanishing gradients; standard for moderate-length sequences
- **Transformer:** best for long-range dependencies and large-scale data; $O(T^2)$ cost per layer

Active research: linear attention, sparse attention, and state-space models aim to reduce the $O(T^2)$ cost while keeping the benefits of attention.

RNN Example: Frequency Estimation

```
from tensorflow.keras.layers import SimpleRNN, Dense

model = Sequential([
    SimpleRNN(50, input_shape=(50, 1), return_sequences=True),
    SimpleRNN(50),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=10, batch_size=32, validation_split=0.2)
```

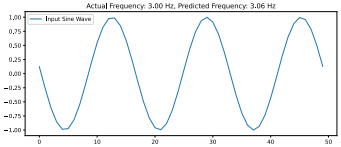


Figure 6: Frequency classification from sine-wave time series.

Application: Engine Health — NASA C-MAPSS

C-MAPSS = Commercial Modular Aero-Propulsion System Simulation [9].

NASA simulated turbofan engines running from healthy to failure under realistic conditions.

- One **cycle** = one start → operate → shutdown sequence (i.e. one flight).
- Each row in the dataset is a snapshot of all sensors during one cycle.
- Each engine has a **different initial wear** and **unknown fault onset**, so trajectories vary in length (e.g. 128–362 cycles).

Item	Description
Subset used	FD001 — single operating condition, single fault mode (HPC degradation)
Training / Test	100 / 100 engines (run-to-failure in train; cut-off in test)
Sensor channels	21 (temperatures, pressures, fan/core speeds, bypass ratio, …)
Operating settings	3 (altitude, Mach, throttle resolver angle)
Target	Remaining Useful Life (RUL) — cycles until failure, capped at 125

Free download: `https://ti.arc.nasa.gov/c/6/`

Engine Health: Problem Setup

Goal: predict how many cycles an engine has left before failure.

- **Piece-wise linear RUL:** true RUL decreases linearly (299, 298, ...), but early in life the sensors look identical — no degradation signal.
  - Capping at 125 means: any RUL > 125 is set to 125, so the model only learns the **degradation phase** where sensors actually change.
- **Rolling window ($w = 30$ cycles):** at each time step we feed the last 30 sensor readings to the model.

Inputs	Output	Why LSTM?
14 sensors + 3 settings over 30 cycles (30×17 matrix)	RUL (cycles)	Hidden state captures gradual degradation trend across the window

Engine Health: Classical Baseline

Strategy: Extract hand-crafted features from a 30-cycle window, then Ridge regression with polynomial interaction terms.

```
# For each window of 30 cycles, extract:
#   rolling mean, rolling std, trend (Δ), cycle number
# per sensor channel → 52 features per sample

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import RidgeCV

ridge = Pipeline([
    ("poly", PolynomialFeatures(degree=2, interaction_only=True)),
    ("scl",  StandardScaler()),
    ("ridge", RidgeCV(alphas=[0.01, 0.1, 1.0, 10.0, 100.0])),
]).fit(X_train, y_train)
```

52 hand-crafted features $\rightarrow$ 1,378 polynomial interaction terms $\rightarrow$ Ridge selects the best linear combination.

Engine Health: Feature Engineering

A 30-cycle window of raw sensor data is a 30×17 matrix. Ridge regression needs a flat vector, so we summarize each sensor column:

Feature	What it captures
Rolling mean	Current average state of the sensor
Rolling std	How noisy / volatile the sensor is
Trend (Δ)	Is the sensor drifting up or down? (degradation direction)
Cycle number	Where in the engine's life we are

This yields $\sim$ 52 scalar features per sample — but **discards the time ordering** within the window.

Engine Health: Why Polynomial Ridge?

Polynomial interactions (`degree=2, interaction_only=True`) — creates all pairwise products of the 52 features, e.g. $\overline{T}_{30} \times \sigma_{\mathrm{P}_{15}}$.

- This lets a linear model capture nonlinear patterns like "temperature rising *while* pressure becomes noisy $\Rightarrow$ imminent failure."
- 52 features $\rightarrow$ 1,378 interaction terms

Ridge (L2) regularization prevents overfitting on 1,378 terms; `RidgeCV` auto-selects the penalty strength via cross-validation.

Key limitation: the temporal **sequence** inside each window is lost — LSTM addresses this.

Engine Health: LSTM (PyTorch)

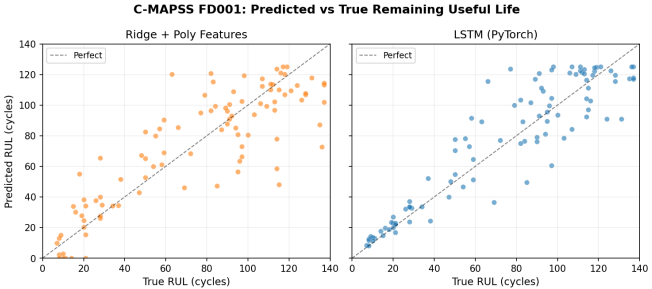
Strategy: Feed raw 30-cycle sensor sequences directly — let the LSTM learn temporal features.

```
class RULPredictor(nn.Module):
    def __init__(self, n_feat=17, h1=64, h2=32):
        super().__init__()
        self.lstm1 = nn.LSTM(n_feat, h1, batch_first=True)
        self.lstm2 = nn.LSTM(h1, h2, batch_first=True)
        self.fc    = nn.Linear(h2, 1)

    def forward(self, x):                # x: (batch, 30, 17)
        out, _ = self.lstm1(x)
        out, _ = self.lstm2(out)
        return self.fc(out[:, -1, :])    # last hidden → RUL

model     = RULPredictor()
optimizer = torch.optim.Adam(model.parameters(), lr=5e-4)
for epoch in range(80):
    for X_b, y_b in train_loader:
        optimizer.zero_grad()
        loss = nn.MSELoss()(model(X_b), y_b)
        loss.backward()
        optimizer.step()
```

Engine Health: Results

Method	RMSE	R^2
Ridge + poly feats	21.5	0.73
LSTM	**17.2**	**0.83**

- LSTM: **20% lower RMSE** — no feature engineering needed
- Ridge needs domain knowledge to build 52 hand-crafted features
- LSTM learns temporal patterns directly from raw sensors



Predicted vs true RUL (cycles). LSTM points cluster tighter around the diagonal.

Other Architrectures

Watch these videos

Watch: 3Blue1Brown

- **Word vectors:** https://youtu.be/FJtFZwbvkI4
- **Convolution:** https://youtu.be/KuXjwB4LzSA
- **Transformers:** https://youtu.be/wjZofJX0v4M
- **Attention in Tranformers:** https://www.youtube.com/watch?v=eMlx5fFNoYc

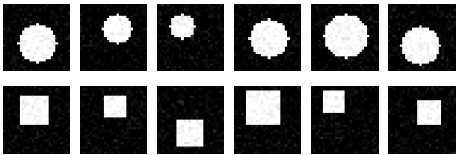
Important Architectures

Watch: 3Blue1Brown

- Convolutional Neural Networks
- Autoencoders
- Transformers

How to use them in aerospace applications?

CNN Intuition: Circles vs. Squares



2000 synthetic 28×28 images (1000 per class). Shapes at random position and size, with noise.

What separates them in pixel space?

- **Circle:** curved boundary — no right-angle gradients anywhere
- **Square:** horizontal + vertical edges, 4 sharp corners

A plain MLP sees 784 *independent* pixel values and must re-learn position from scratch.

A **CNN** slides a small window across the image and detects local patterns *wherever they appear*.

ShapeCNN: Data & Architecture

```python
import numpy as np, torch, torch.nn as nn

IMG = 28   # pixels

def make_circle():
    img = np.zeros((IMG, IMG), np.float32)
    r  = np.random.randint(5, 10)
    cx = np.random.randint(r+2, IMG-r-2)
    cy = np.random.randint(r+2, IMG-r-2)
    yy, xx = np.ogrid[:IMG, :IMG]
    img[(xx-cx)**2+(yy-cy)**2<=r**2] = 1.
    return img

def make_square():
    img = np.zeros((IMG, IMG), np.float32)
    s  = np.random.randint(8, 16)
    x0 = np.random.randint(2, IMG-s-2)
    y0 = np.random.randint(2, IMG-s-2)
    img[y0:y0+s, x0:x0+s] = 1.
    return img

circles = [make_circle() for _ in range(1000)]
squares = [make_square() for _ in range(1000)]
X = np.array(circles+squares)[:,None] # (2000,1,28,28)
y = np.array([0]*1000 + [1]*1000)
```

```python
class ShapeCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 8, 5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2),      # 28 → 14
            nn.Conv2d(8,16,3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),      # 14 → 7
        )
        self.head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16*7*7, 32),
            nn.ReLU(),
            nn.Linear(32, 2),     # circle / square
        )
    def forward(self, x):
        return self.head(self.features(x))
```

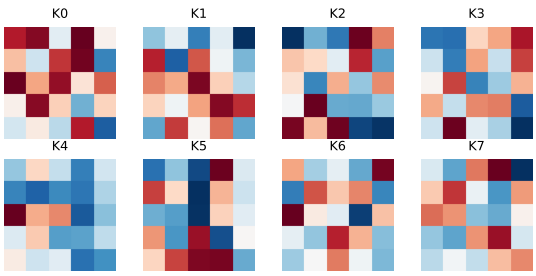
Conv2d = 2D sliding filter — the 2D analogue of the 1D C_p case.\[3pt] **MaxPool2d** = translation tolerance: the circle can be anywhere.

ShapeCNN: Training & Learned Filters

```
model   = ShapeCNN()
opt     = torch.optim.Adam(
              model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

for epoch in range(50):
    opt.zero_grad()
    loss = loss_fn(model(X_tr), y_tr)
    loss.backward(); opt.step()

model.eval()
with torch.no_grad():
    preds = model(X_te).argmax(1)
acc = (preds == y_te).float().mean()
print(f"Accuracy: {acc:.1%}")  # → 99.5%
```

Class	Prec.	Rec.
Circle	1.00	0.99
Square	0.99	1.00

Layer-1 learned filters (5×5, Conv2d)

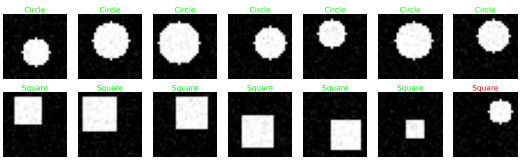


Filters emerge as **oriented edge detectors** at different angles and frequencies — the CNN automatically reinvented Sobel/Gabor filters.

Key takeaway: no one programmed "look for curves" or "look for corners." The network learned these from labeled examples alone.

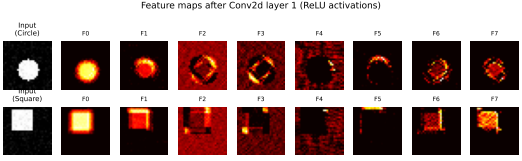
ShapeCNN: Does It Work?

Test predictions (green = correct, red = wrong)



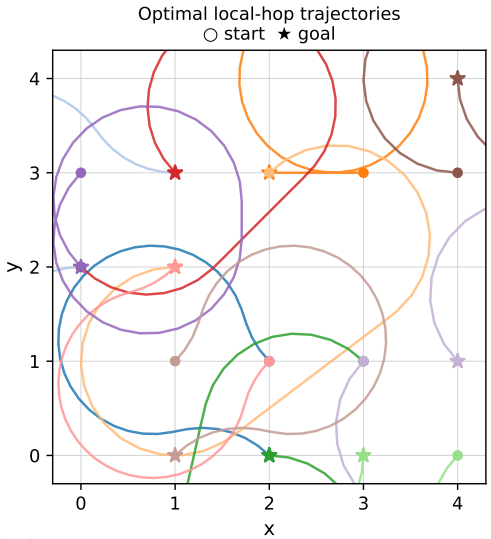
13/14 shown correct; overall test accuracy **99.5%**.

Feature maps after Conv2d layer 1



F2/F3 trace the **curved boundary** of a circle. Same filters produce **straight-edge** responses for a square — that's how the network separates the two classes.

Transformer Example: Dubins Car Path Planning



Dubins car: constant speed $v = 1$, steer with $u \in [-1, 1]$ rad/s.

Dataset: solve a **minimum-time OCP** (CasADi + IPOPT) for every local hop on a 5×5 grid $\times$ all 8×8 heading combos ;($\theta_k = k\pi/4$):

$$\min_{u(\cdot),\,T_f} \; T_f \quad \text{s.t.} \; \begin{cases} \dot{x} = v\cos\theta,\ \dot{y} = v\sin\theta,\ \dot{\theta} = u \\ |u| \le 1 \end{cases}$$

Key design: train only on **nearest-neighbour hops** — the transformer must **stitch hops** to reach distant goals (otherwise it is just a look-up table).

7573 trajectories solved in $ $288 s on 8 cores.

Transformer: State Encoding & Architecture

```
class DubinsTransformer(nn.Module):
  def __init__(self, d=64, nhead=4, nlayers=3):
    # encode each raw state R^3 -> R^d
    self.state_enc = nn.Linear(3, d)
    self.goal_enc  = nn.Linear(3, d)
    layer = nn.TransformerEncoderLayer(
        d, nhead, dim_feedforward=256,
        batch_first=True, norm_first=True)
    self.tf   = nn.TransformerEncoder(layer, nlayers)
    self.head = nn.Linear(d, 3)  # R^d -> next (x,y,th)

  def forward(self, states, goal):
    g_tok = self.goal_enc(goal).unsqueeze(1) # (B,1,d)
    s_tok = self.state_enc(states)           # (B,t,d)
    seq   = cat([g_tok, s_tok], dim=1)       # (B,t+1,d)
    mask  = causal_mask(seq)
    out   = self.tf(seq, mask=mask)
    return self.head(out[:, 1:]) # skip goal token
```

State encoding projects $\mathbf{s}_t \in \mathbb{R}^3$ into a compact $d{=}64$ space:

$$\mathbf{z}_t = W_s\,\mathbf{s}_t + \mathbf{b}_s \;\in\; \mathbb{R}^{64}$$

Goal token prepended — every state attends to the goal at all 3 layers.

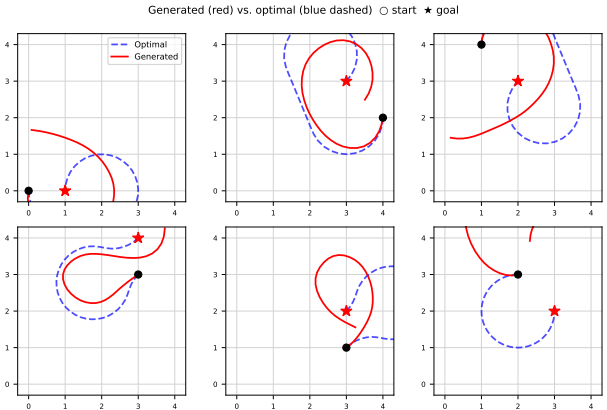
Causal mask prevents future look-ahead, enabling autoregressive rollout at test time.

Training: teacher forcing, cosine-annealed Adam, 500 epochs, batch 512:

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T} \|\hat{\mathbf{s}}_t - \mathbf{s}_t\|^2$$

Final loss: 1.2×10^{-4}

Transformer: Generated Trajectories



Autoregressive rollout:

$$\hat{\mathbf{s}}_{t+1} = f_\theta(\hat{\mathbf{s}}_{0:t},\, \mathbf{g})$$

No ground truth at test time — each prediction feeds back as the next input.

- Trained only on **local hops** (1-step neighbors)
- Test cases include longer hops the model never saw
- The transformer **attends to the goal token** at every layer to stay on course
- Replicates the curved, arc-shaped structure of min-time Dubins paths

Physics-Informed Neural Networks

When Data Alone Is Not Enough

Classical NNs need labeled data $\{(x_i, y_i)\}$. In many aerospace problems:

- **Data is scarce** — each CFD run or flight test is expensive
- **Physics is known** — governing equations (ODEs, PDEs) encode what we already understand
- **Extrapolation matters** — a data-only NN can predict nonsense outside the training envelope

Idea (Raissi et al., 2019): embed the governing equation directly into the loss function.

$$\mathcal{L} = \underbrace{\mathcal{L}_{\text{data}}}_{\text{fit observations}} + \lambda \underbrace{\mathcal{L}_{\text{physics}}}_{\text{satisfy PDE/ODE}}$$

The network is trained to satisfy *both* the data and the physics simultaneously.

PINN Formulation

Let $u(t;\,\theta)$ be the NN approximation of the true solution $u^*(t)$.

Data loss (initial/boundary conditions or sparse measurements):

$$\mathcal{L}_{\text{data}} = \frac{1}{N_d}\sum_{i=1}^{N_d}\big(u(t_i;\theta) - u_i^*\big)^2$$

Physics residual (evaluated at collocation points $\{t_j\}$, no labels needed):

$$\mathcal{L}_{\text{phys}} = \frac{1}{N_c}\sum_{j=1}^{N_c}\big[\mathcal{N}[u](t_j;\theta)\big]^2$$

where $\mathcal{N}[\cdot]$ is the differential operator. Gradients through $\mathcal{N}$ are computed by **automatic differentiation** — exactly the same `autograd` used for backprop.

Total loss: $\mathcal{L} = \mathcal{L}_{\text{data}} + \lambda\,\mathcal{L}_{\text{phys}}$

Application: Spring-Mass Oscillator

Problem: learn the solution $x(t)$ of the spring-mass ODE

$$\ddot{x} + \omega_n^2\, x = 0, \qquad x(0) = 1,\ \dot{x}(0) = 0, \qquad \omega_n = 2\pi \text{ rad/s}$$

- **Exact solution:** $x^*(t) = \cos(\omega_n t)$
- **Only 5 IC measurements** are given to the NN as data
- **2,000 collocation points** enforce the ODE residual
- The network takes t as input and outputs $\hat{x}(t)$
- Analogy: vibrating structure, landing gear dynamics, aeroelastic mode

```
omega_n = 2 * np.pi                        # natural frequency [rad/s]
t_ic    = torch.tensor([[0.0]])
x_ic    = torch.tensor([[1.0]])            # x(0) = 1
xdot_ic = torch.tensor([[0.0]])            # xdot(0) = 0
t_col   = torch.linspace(0, 2, 2000).unsqueeze(1).requires_grad_(True)
```

PINN: Model Definition

```
class PINN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 1),
        )
    def forward(self, t): return self.net(t)

def ode_residual(model, t):
    """Compute x_tt + omega_n^2 * x at collocation points."""
    t = t.requires_grad_(True)
    x     = model(t)
    x_t   = torch.autograd.grad(x.sum(), t,
                                create_graph=True)[0]
    x_tt  = torch.autograd.grad(x_t.sum(), t,
                                create_graph=True)[0]
    return x_tt + omega_n**2 * x          # residual; should be 0
```

`autograd.grad` differentiates the *network output* w.r.t. the *input* t — not the weights. This is automatic differentiation used as a calculus tool.

PINN: Training Loop

```
model     = PINN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
lam       = 1.0                          # physics weight

for epoch in range(10000):
    optimizer.zero_grad()

    # Data loss: initial conditions
    x_pred   = model(t_ic)
    xd_pred  = torch.autograd.grad(x_pred.sum(), t_ic,
                   create_graph=True)[0]
    L_data   = (x_pred - x_ic).pow(2).mean() \
             + (xd_pred - xdot_ic).pow(2).mean()

    # Physics loss: ODE residual at collocation points
    L_phys   = ode_residual(model, t_col).pow(2).mean()

    loss     = L_data + lam * L_phys
    loss.backward()
    optimizer.step()

print(f"Final loss: {loss.item():.6f}")
```

PINN: Results

```
t_test = torch.linspace(0, 2, 500).unsqueeze(1)
model.eval()
with torch.no_grad():
    x_pred = model(t_test).numpy()
x_exact = np.cos(omega_n * t_test.numpy())
rmse = np.sqrt(((x_pred - x_exact)**2).mean())
print(f"RMSE vs exact: {rmse:.5f}")
```

Output (typical):

```
Final loss: 0.000031
RMSE vs exact: 0.00412
```

- With **5 data points** and 2,000 collocation points, the PINN recovers $\cos(\omega_n t)$ to $< 0.5\%$ error over $[0, 2]$ s
- A data-only NN with 5 points would wildly overfit — the physics loss is the regularizer
- The same framework extends to PDEs (heat equation, Euler, Navier-Stokes) by adding spatial inputs to the network

PINN: Why Not Just Integrate the ODE?

For a simple ODE, classical integrators (RK4) are faster and more accurate. PINNs earn their place when:

Situation	Why PINN helps
Incomplete physics (unknown forcing)	Physics loss constraints what is known; data fills the gap
Inverse problems	Identify unknown parameters ω_n, c from sparse measurements
High-dimensional PDEs	Mesh-free collocation avoids the curse of dimensionality
Data + physics fusion	Sensor data and known dynamics combined in one loss

Aerospace examples: identifying structural damping from flight data, solving aeroelastic flutter equations with sparse strain gauge measurements, turbulence closure modeling.

Summary

Architecture Overview

Architecture	Best suited for	Aerospace example
FFNN / MLP	Tabular data, regression	$C_D(M, \alpha)$ surrogate; 10^4 CFD samples
RNN	Short sequences	Frequency estimation from telemetry
LSTM	Long sequences, forecasting	Engine RUL (NASA C-MAPSS); 80+ cycles
Transformer	Long-range dependencies	Turbulence intensity forecasting
CNN	Spatial / field data	$C_p(x/c)$ flow regime classification
Autoencoder	Anomaly detection, dim. red.	IMU anomaly detection without labels
PINN	Physics + sparse data	Spring-mass ODE; structural damping ID

All examples in this lecture run on a laptop CPU in under 2 minutes.

References

[1] M. Telgarsky, "Benefits of depth in neural networks," in *Proceedings of the 29th annual conference on learning theory (COLT)*, 2016, pp. 1517–1539.

[2] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao, "Why and when can deep—but not shallow—networks avoid the curse of dimensionality: A review," *International Journal of Automation and Computing*, vol. 14, no. 5, pp. 503–519, 2017.

[3] J. Kaplan *et al.*, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[4] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

References

[5] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. Available: https://www.deeplearningbook.org

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[8] A. Vaswani *et al.*, "Attention is all you need," in *Advances in neural information processing systems (NeurIPS)*, 2017.

[9] A. Saxena, K. Goebel, D. Simon, and N. Eklund, "Damage propagation modeling for aircraft engine run-to-failure simulation," in *International conference on prognostics and health management (PHM)*, IEEE, 2008, pp. 1–9.