Learning Equations for Extrapolation and Control

→ focus on learning mathematical expressions from data ≠ standard black box ML models

→ Instead of predicting outcomes, the goal is to identify **explicit** functional relationships – meaning the discovered model can be interpreted, extrapolated and used for control tasks

→ to achieve this, proposition of the EQL÷ (Equation Learner with Division) Model

(designed to → :

1- identify underlying equations governing a system (physic-based equations)
2- Generalize beyond training data (extrapolation)
3- Be useful in real-world control tasks, like robotics)

} EQL÷

Why learn equations instead of just predicting ? :

- Standard ML models like NN = black boxes + example of pendulum movement prediction & Newton's eqs of motion

⇒ • Instead of training a model just to make predictions, we want to discover the underlying mathematical equations governing a system

Core problem : Learning Equations from Data :

problem framed as a regression task + where the system follows an unknown analytical expression : $y = \phi(x) + \xi$

- x = input (system parameters)
- y = observed output (e.g. system response)
- $\phi(x)$ is the true eq. governing the system
- ξ = small noise in the measurements

Goal : learn an equation $\psi(x)$ that closely approximates $\phi(x)$, while remaining interpretable and generalizable

The EQL Model : NN for symbolic regression

— ML technique used to find math. expressions that best describe relationships within data (structure and parameters of the equ. that fits the data)

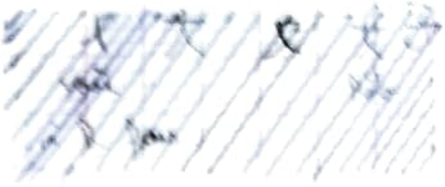
EQL : NN that learns equations

(hidden layers contains specific mathematical fcts $\left(\begin{smallmatrix} id \\ \sin, \cos \\ \times, \div \end{smallmatrix}\right)$

⇒ build complex equations by combining basic math operations

advantage : model naturally produces outputs in the form of interpretable equations

limitation : - could not handle division operations, which are crucial for many physical problems
- model selection process often picked overly complex or incorrect equations



Improvements in [illegible] EQL÷ :

1- Adds Division Operation to the model.
2- Stabilizes training ($\div$ can cause problems when denominators approach zero)
3- Improves Model selection to better identify the true underlying equation

Adding DIV units is challenging because:
- When den. approaches zero, the function becomes unstable
- Gradients in NN training can become very large, making optimization difficult

SOLUTION: regularized division function: $h_\theta(a,b) = \begin{cases} \frac{a}{b} & b > \theta \\ 0 & \text{otherwise} \end{cases}$

- θ a small threshold that ensures denominators never get too close to zero
  ↳ works because: - Prevents extreme values from destabilizing training
  - Encourages stable learning of division-based equations

Training EQL÷ :

↳ trained using gradient-based optimization (e.g. Adam optimizer) with
L1- regularization ($\times$: $L = \sum(y_i - \hat{y}_i)^2$ | $\checkmark$: $L = \sum(y_i - \hat{y}_i)^2 + \underset{\text{reg parameter}}{\lambda} \sum |w_i|$)
to encourage sparsity (most of the elements are $\neq 0$ or $= 0$)

Loss function: The total loss consists of:

1- MSE (measures how well the predicted eqn. matches data)
2- L_1 Regularization (encourages simpler equations by penalizing unnecessary terms)
3- Penalty for small denominators (ensures division remains stable)

↳ Key training strategy: - Initially, the model is highly regularized to encourage simple equations.
- As training progresses, reg. is gradually reduced, allowing finer adjustments

- another application: $y_2 = \frac{x_2+1}{3}\sin(\pi x_1) + x_2 x_3 x_4$, extrapolation, robotics: controlling a cart pendulum system

EPFL AI Internship Preparation

Why do we need Neural Operators (NO) ?

Traditional NN : - great at mapping inputs to outputs
- BUT they work with FINITE-DIMENSIONAL spaces

· When a NN learns a mapping $\overbrace{(x,t) \mapsto u(x,t)}^{\text{a function}}$, the problem is ORIGINALLY an infinite-dimensional one (sol. are functions not just finite-dimensional vectors)

↳ For example Partial Differential Equations (PDEs):

$$F\left(x_1, x_2, \dots, x_n, \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_n}, \frac{\partial^2 u}{\partial x_i \partial x_j}, \dots\right) = 0$$

→ NO solve this by generalizing NN to learn mapping between function spaces, making them resolution-independent (= process inputs of different resolutions without requiring retraining or modifications) ⇔ varying discretization (different grid sizes)

What are Neural Operators ? :

Key Characteristics :

- Discretization Invariance : Do not depend on grid resolution
- Map from one function to another : learns R between entire functions (not vectors or images)
- Use integral operators : Instead of Matrix multiplication, NO use integral transforms to compute their results
- Universal approximation : NO can approximate any nonlinear continuous operators

↳ It matters because :
- Traditional ~~[illegible]~~ Deep learning Models work only on SPECIFIC GRID SIZES ⇒ need retraining for new resolutions
- NO allow training once and applying at any resolution
⇒ more efficient and scalable (eg. NO much faster than trad. PDE solvers

↳ - Zero-shot Super-Resolution : train on lower-resolution data, apply to high resolution problems without extra training

How do NO work ?:

• NO generalize standard NN by replacing their layers with integral operators layers.

Structure of a NO:

1. Lifting layer : Expands input function space to a higher-dimensional representation ← allows the NO to learn richer patterns

2. Integral Operator Layers: Apply a sequence of [integral transformations] instead of matrix multiplication
~~[illegible]~~

3. Nonlinear Activation : Uses pointwise (at each x) nonlinearities (Relu, sigmoid) for expressiveness

4. Projection Layer : Maps the high-dimensional function back to the output function ~~space~~ space.

Mathematical formulation of a NO:

$$G_\theta(a) = Q \circ \sigma \circ (K(a) + Wa + b) \circ P$$

$P: a \mapsto P(a)$

$x \mapsto \int_D K(x,y)\,a(y)\,dy$

integral kernel operator

- $K(a)$ = integral operator (core part)
- $Wa + b$ = standard affine transformation
- σ = nonlinear activation function
- P and Q = lifting and projection layers

Differents types of NO proposed :

1. GNO (Graph NO): $O(N^2)$ ← nodes: f values at discrete pts; edges define local dependencies ($f(x)$ depends on nearby points)
   - GNN to propagate infos between nodes
   - $K(a)$ approx. thanks to Nyström Method (computes integrals using a subset of sampled nodes)
- uses a graph-based structure to model interactions
- suitable for unstructured grids (irregular meshes in engineering app.)
- Computationally expensive but flexible
- requires a graph structure (⇒ less gen. for co fcts)

2. LNO (Low-Rank NO): $O(N)$
- approximates the integral operator using a low rank factorization ← $K(x,y) = \sum_{i=1}^{r} \phi_i(x)\psi_i(y)$
   ϕ_i, ψ_i = low rank basis functions
   ← good for diffusion type eqn. ; ex. $\frac{\partial u}{\partial t} = k\nabla^2 u$
   approx. high dimensional data by data using a small nb of basis components →
- Faster but less expressive (+ memory-efficient too)
- Bad gen. to highly non-local interactions

3. Multipole Graph NO (MGNO): $O(N \log(N))$
- Inspired by Fast Multipole Methods (FMM) ← • instead of FC Graphs = GNO, MGNO hierarchically group nodes based on their distance
   • fast multipole expansion to approx. long range interactions
- uses a multi-level hierarchical approx. for efficiency
- balances speed and accuracy (requires graph-based input / more complex than LNO)
   ↳ • kernel matrix factorized into diff. scales to handle both local and global dependencies

4. Fourier NO (FNO): $O(N\log(N))$ ← best performing variant, allowing high accuracy and speed

- uses the Fourier Transform to approximate the integral operator.
- fastest (~>>) and most accurate method for many PDE problems
- Works well for smooth functions (e.g. fluid dynamics) and periodic functions
  is $f \in C^\infty(\mathbb{R})$ = infinitely differentiable

* instead of computing the integral $K(a)$ directly, FNO applies a Fourier Transform

$K(a)(x) = \mathcal{F}^{-1}(R \cdot \mathcal{F}(a))$, R a learnable weight matrix that operates in a Fourier Space

$$\mathcal{F}(a)(Y) = \int_{-\infty}^{\infty} a(x)\, e^{-2\pi i x Y}\, dx$$

- requires uniform grids (not as flexible as GNOs)
- not great for discontinuous solutions

Challenges and Future Directions

- Handling Non-Smooth Solutions: NO struggle with discontinuous functions like shocks in hyperbolic PDEs
- Better Theoretical Understanding: more research is needed on error bounds and expressiveness
- Hybrid Approaches: Combining NO with traditional solvers could improve robustness

NOMTO: Neural Operator-based symbolic Model Approximation and discOvery

Why do we need NOMTO?:

- same motivation as EQL and EQL÷ pendulum and Newton's law of motion example ⇒ need of symbolic regression (SR)

- Problems with existing symbolic regression methods:

1- Traditional SR (like genetic programming) searches for equations heuristically, making it slow and inefficient

2- DL-based SR models require large training datasets and often fails to generalize to unseen functions

3- Models like EQL, while more flexible, struggle with complex operations (e.g. : $\underbrace{\text{divisions}}_{\text{not EQL÷}}$, exponentials, derivatives)

- NOMTO is meant to solve this, through this Key Idea :

NOMTO (Neural Operator-based symbolic Model approximaTion and discOvery) combines the power of Neural Operators with SR to efficiently discover equations that include :

- Singularities (division, exponentials)
- Special functions (Gamma function, Airy function) $\longleftrightarrow$ satisfying $\frac{d^2y}{dx^2} - xy = 0$
- Differential operators (time derivative, Laplacians)

⇒ provides interpretable equations thanks to NO specifically used and modified to learn PDEs directly from data

↳ slightly harder to train but far more generalizable

How does NOMTO work ?

1- Trains NO to approximate fundamental operations (multiplication, sin, etc)

2- Builds a computational graph where each node is an operation learned by NO

3- Optimizes the graph to extract the simplest possible symbolic equation

⇒ unlike black box DL-models, NOMTO discovers equations explicitly, making them interpretable

In detail, How does NOMTO work?:

1- Train Neural Operators to approximate basic Operations:

- NOs (like FNOs and CNOs) learn basic mathematical operations.
- These operations form a library of symbolic functions:

  $\text{library} = \{+, -, \times, \div, \partial/\partial t, \partial^2/\partial x^2, \sin, \cos, \exp, \underbrace{\text{Airy Function}, \Gamma}_{[illegible]}\}$

→ important step because traditional SR only handles simple algebraic functions, while NOMTO learns even differential operators and special functions

2- Construct the Computational Graph:

- After training NOs, the building blocks are combined into a graph-based representation of a symbolic equation.
- The edges in the graph represent learned relationships between operations

ex.: if the true eqn is $y = \sin(x) + \frac{x^2}{1+x}$ → the NOMTO graph would dynamically discover that:
- y depends on x
- sin(x) and x^2 are important
- a division is necessary

3- Optimize the graph to Minimize Loss

- The system optimizes the structure of the graph to find the simplest possible equation that fits the data (+ gradient based optim.)
- Uses sparsity contraints to avoid overcomplicated expressions

→ advantage: traditional methods struggle to simplify equations, while NOMTO actively removes unnecessary terms

~~Ex: [illegible]~~
Differences:
NO (FNO): given fluid flow at time t, predicts fluids flow there a time t+1
NOMTO: given observations of fluid flow, discover the Navier-Stokes Equation

4- Extract the final Symbolic Expression

- The optimized computational graph is translated into an explicit symbolic equation.
- This equation is interpretable and generalizable beyond the training data

⇒ NOMTO significantly outperforms existing SR methods in discovering complex mathematical models.

- Revolutionary for PDE learning: accurately finds heat equation, Burgers equation and [illegible]