



NN-PES, creating neural network-based solver for parametric equations

J-P. Merlet

INRIA Sophia-Antipolis

Objectives



- $F(\mathbf{Y}, \mathbf{X}) = 0$: system of parametric non-differential equations
- $\mathbf{Y}$ are **parameters** with arbitrary values in bounded domains
- $\mathbf{X}$ represents the n **unknowns**
- F is an arbitrary set of n equations, at least C^2

$\Rightarrow$ for a given $\mathbf{Y}$ we have a **square system** of size n that, usually, will have a finite number of solutions

Objectives



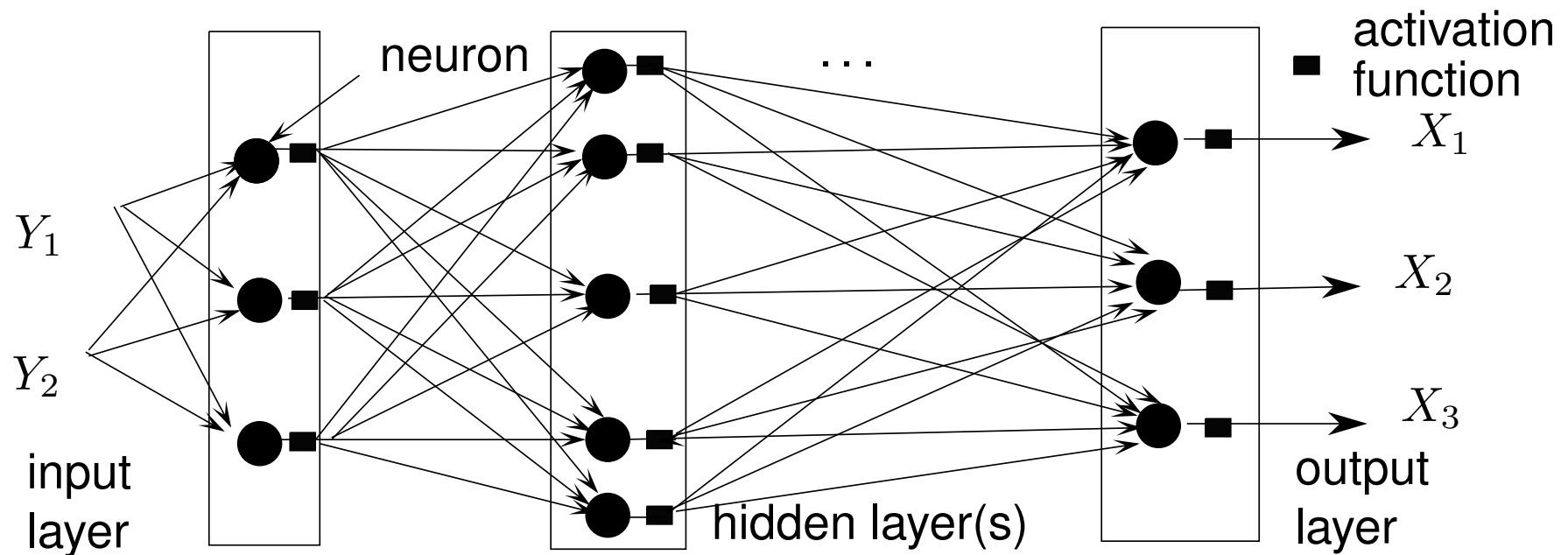
We want to create a specific solver for F that

1. **possibly** get all solutions of F for any given Y
 2. provides **exact/certified** solutions (solutions with an arbitrary large number of exact digits/even if solutions are close)
 3. is **fast** because it uses neural networks
- we don't make any assumption on the maximal number of solutions
 - approach **not intended** to be a substitute for analysis whenever it's possible

Neural networks



Focus on Multi-Layer Perceptron (MLP)



- input: a vector $\mathbf{Y}$, output: a prediction vector for $\mathbf{X}$
- layers: input, hidden, output
- each layer has **neurons** associated to an **activation function**
- **for a given $\mathbf{Y}$ we get a single prediction for $\mathbf{X}$**



Neural networks: MLP

each neuron of a layer receives a set of n_u inputs u_j and calculates its output S

$$S = AF\left(\sum_{k=1}^{k=n_u} a_k u_k + b_k\right)$$

- a_k, b_k : weight and bias
- AF : activation function, usually non linear



Neural networks: MLP

An MLP is a supervised neural network: requires a **learning set** i.e a set of N samples $(\mathbf{Y}_k, \mathbf{X}_k)$ that is used for its **training**

- **loss**: a statistical index which quantifies the average error between the MLP **prediction** $\hat{\mathbf{X}}_k$ and the expected solution $\mathbf{X}_k$ over the samples of the learning set

$$\text{loss} = \sqrt{\sum_{j=1}^{j=N} ||\mathbf{X}_j - \hat{\mathbf{X}}_j(\mathbf{Y}_j)||^2 / N}$$

- an **optimizer** that incrementally determine the a_j, b_j that minimize the loss



MLP parameters

network parameters:

- number of hidden layers
- number of neurons in each layer (input, hidden, output)
- activation function for each neuron
- **no known rules** to determine these parameters

simplifying assumption: same activation function for all neurons



Solver creator input

Input : an initial solution set

- a limited number of samples $(\mathbf{Y}_j, S_j^1, S_j^2, \dots)$ (S_j^k is a solution for parameters $\mathbf{Y}_j$)
- not necessary to have **all** solutions for a given $\mathbf{Y}_j$
- $\mathbf{Y}_j$ distributed over the possible values of $\mathbf{Y}$

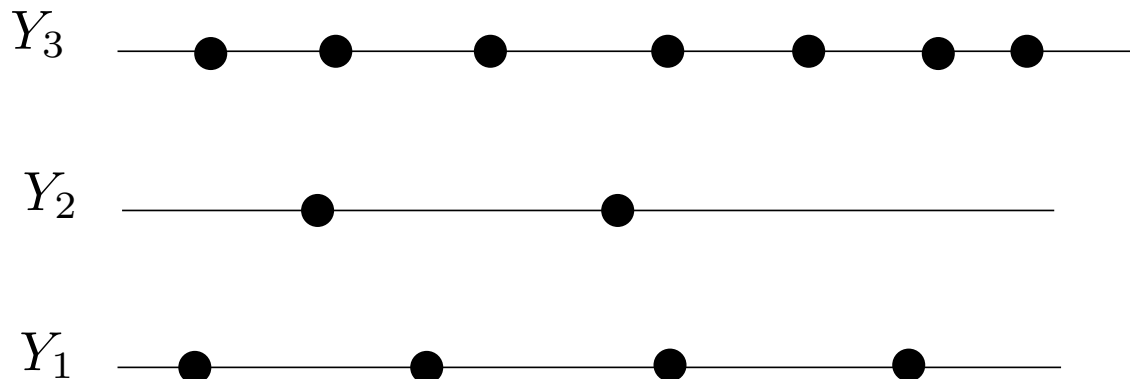


Solver creator input

Input : an initial solution set

- a limited number of samples $(\mathbf{Y}_j, S_j^1, S_j^2, \dots)$ (S_j^k is a solution for parameters $\mathbf{Y}_j$)
- not necessary to have **all** solutions for a given $\mathbf{Y}_j$
- $\mathbf{Y}_j$ distributed over the possible values of $\mathbf{Y}$

Graphical representation: an horizontal line, called a **level** $\rightarrow Y_i$,
a point on a line $\rightarrow S_j^k$, $(\mathbf{Y}_j, S_j^k)$ is a **node**
...





Learning sets

with the initial solution set it is **easy to create learning sets of almost arbitrary size**:

- consider a node (Y_j, S_j^k) and
 1. either a level $Y_m, m \neq j$
 2. or an arbitrary unit vector v in the Y space and
$$Y_m = Y_j + \lambda v$$
- use **continuation** to move from Y_j toward Y_m , tracking the solution starting with S_j^k
- store samples on the way

Training



we are done !

Training



not yet ... because of multiple solutions

Training



not yet ... because of multiple solutions

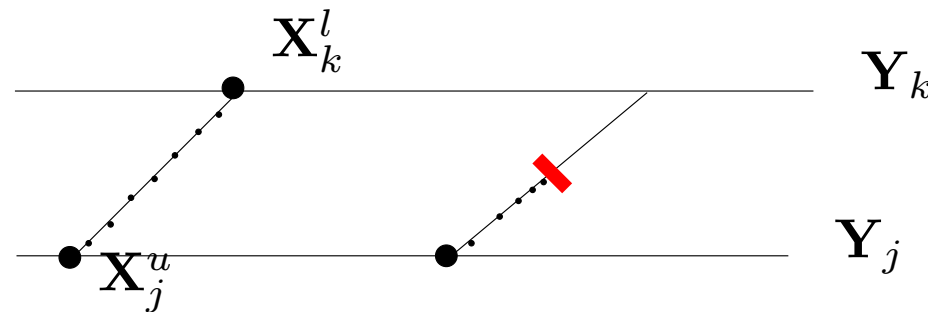
Example: intersection of 2 circles $(x_1, 0, r_1)$, $(x_2, 0, r_2)$ in a plane. If 2 solutions (x_s, y_s) determine learning sets $\mathcal{L}_+$ with $y_s > 0$ and $\mathcal{L}_-$ with $y_s < 0$ then train:

- final loss with $\mathcal{L}_+$ or $\mathcal{L}_-$: $1e-4$
- final loss with $\mathcal{L}_+ \cup \mathcal{L}_-$: 230



Structure

- consider a node $\mathcal{N}_j^u = (\mathbf{Y}_j, \mathbf{X}_j^u)$ and a level $\mathbf{Y}_k, k \neq j$
- we use a **continuation** scheme that moves $\mathbf{Y}$ from $\mathbf{Y}_j$ to $\mathbf{Y}_k$ and follows the evolution of the solution
- this scheme may
 1. **fail**: Newton does not converge at some point
 2. reach a solution $\mathbf{X}_k^l$ (possibly not in the initial solution set)

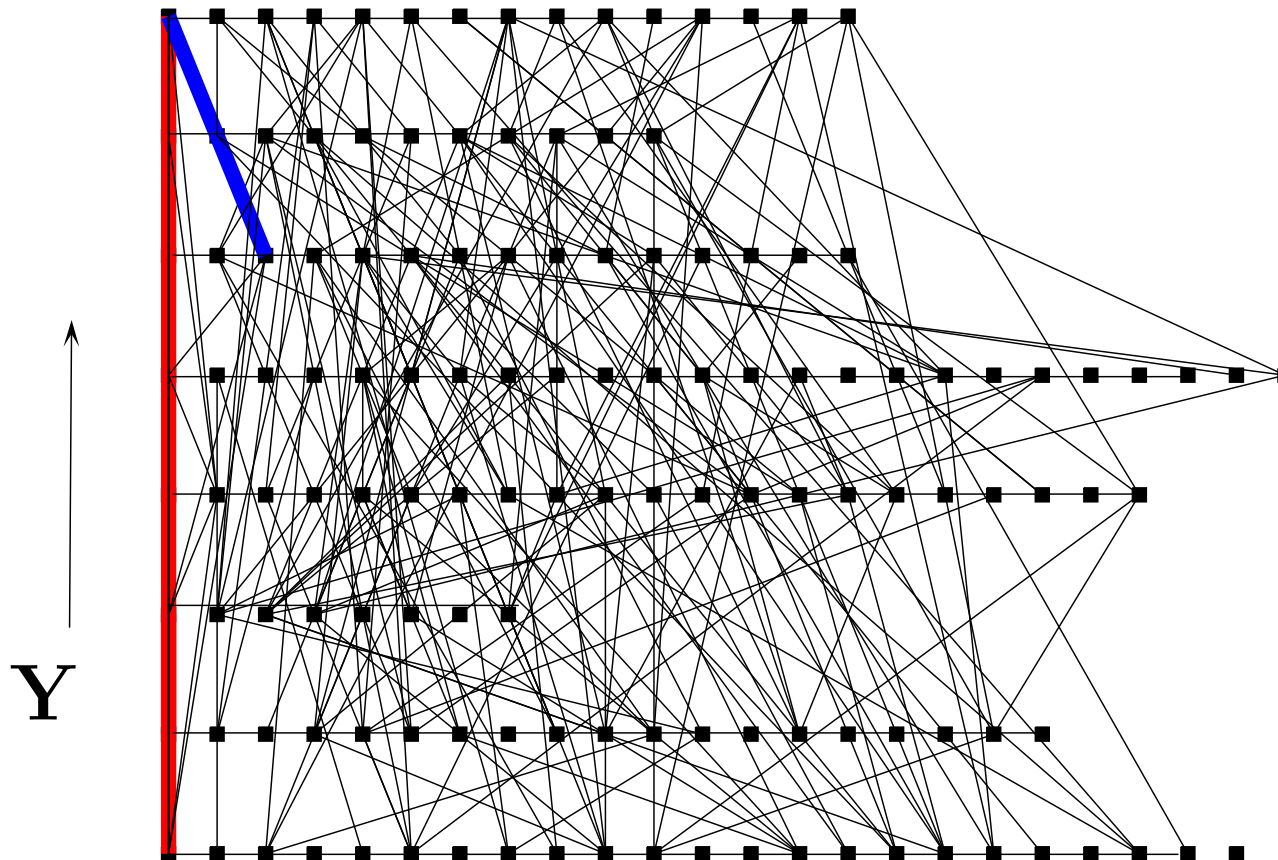


In the later case we say that the node $\mathcal{N}_k^l = (\mathbf{Y}_k, \mathbf{X}_k^l)$ is **connected** to the node $\mathcal{N}_j^u$



Structure

We may represent graphically the connection between 2 nodes by an **edge**. If we use all the nodes as starting point toward all the other levels we get the **solution graph**



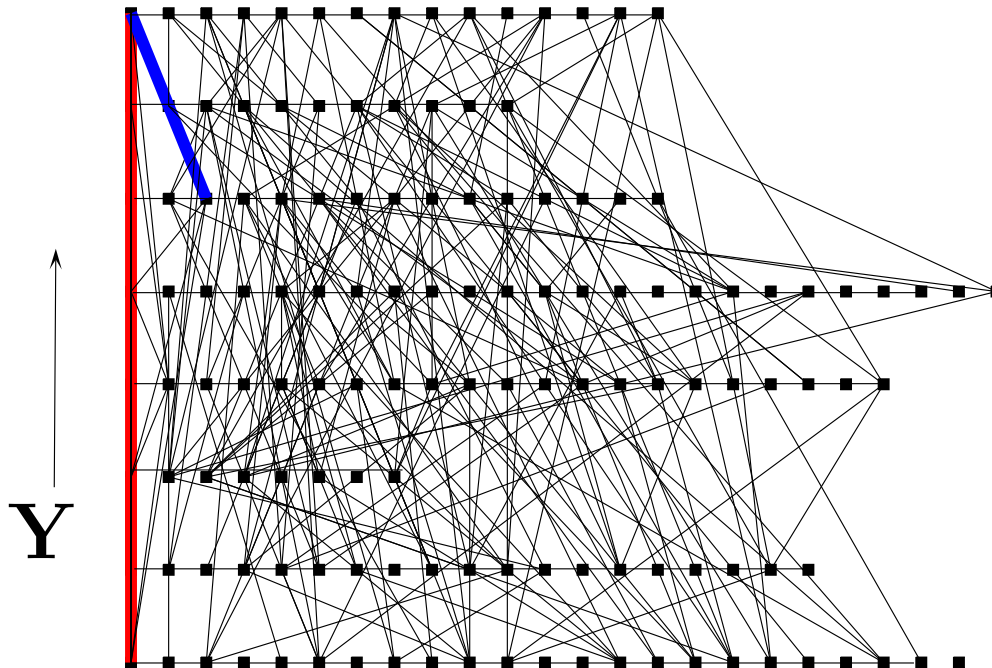


Structure

In the graph we have:

- **isolated nodes**: nodes not connected to any other nodes
- **branches**: sequentially connected node **with only one node** for a given Y_j

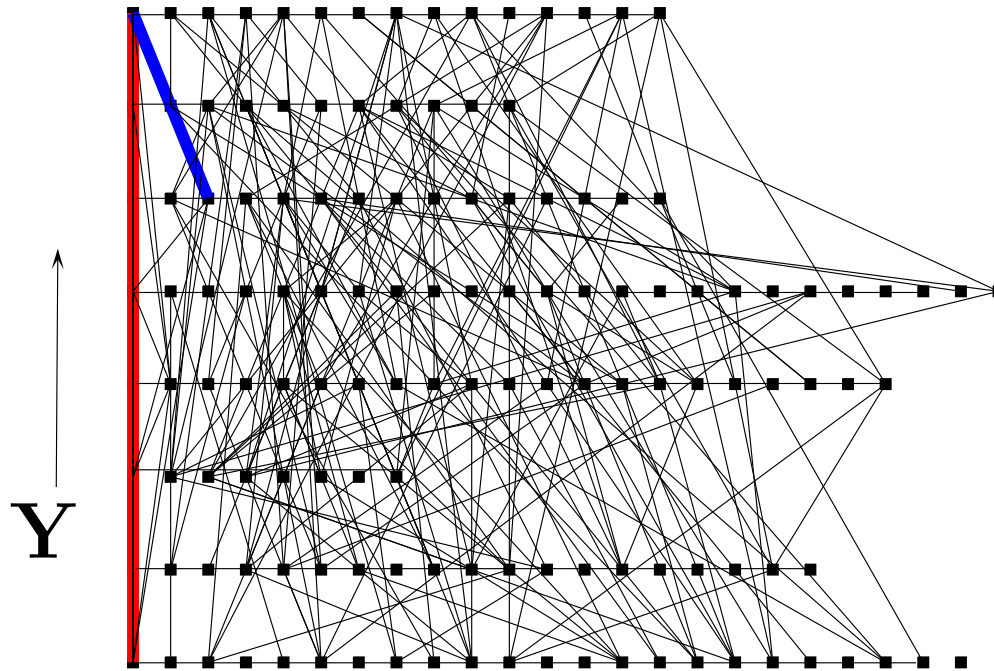
we create **learning sets** for each isolated node and branch



Structure



We get as many MLPs as the number $\mathcal{M}$ of isolated nodes and branches



Issues



classical learning strategy for a learning set

- loss is only an average error between the prediction and the expected solution over the samples of a learning set
- **issue 1**: they may be **still** large errors for some samples of the learning set even if the loss is small

We have $\mathcal{M}$ MLPs to which we provide as input a given $\mathbf{Y}$

- **issue 2**: among the $\mathcal{M}$ predictions provided by all the MLPs which one may be close to a solution ?

Hybridization



A prediction provide a guess for a solution

predictions of the MLP used as **guess** for the Newton method



Hybridization

predictions of the MLP used as **guess** for the Newton method

Results for a given learning set/hybrid MLP

- N convergences of Newton **toward the expected solution** for the M samples of the learning set
- **success rate (SR)**: $\frac{100N}{M}$
- if $SR=100 \rightarrow$ **optimal MLP**
- **but** SR calculated for the MLP with the minimal loss is usually low: $\approx 2\%$



Hybridization

let's go back to the **training process**:

- at iteration k of the training process the optimizer creates an MLP, M_k with loss loss_k
- we have calculated the SR only for the minimal loss



Hybridization

let's go back to the **training process**:

- at iteration k of the training process the optimizer creates an MLP, M_k with loss loss_k
- we have calculated the SR only for the minimal loss

Question:

relationship between loss_k and the success rate SR_k of M_k ?



Hybridization

let's go back to the **training process**:

- at iteration k of the training process the optimizer creates an MLP, M_k with loss loss_k
- we have calculated the SR only for the minimal loss

Question:

relationship between loss_k and the success rate SR_k of M_k ?

- at each optimizer iteration k where the loss decreases we compute SR_k and store $(\text{loss}_k, \text{SR}_k)$
- we have points on the curve $\text{SR}(\text{loss})$



Hybridization

let's go back to the **training process**:

- at iteration k of the training process the optimizer creates an MLP, M_k with loss loss_k
- we have calculated the SR only for the minimal loss

Question:

relationship between loss_k and the success rate SR_k of M_k ?

- at each optimizer iteration k where the loss decreases we compute SR_k and store $(\text{loss}_k, \text{SR}_k)$
- we have points on the curve $\text{SR}(\text{loss})$

we expect a curve where $\text{loss} \searrow \Rightarrow \text{SR} \nearrow$

Hybridization



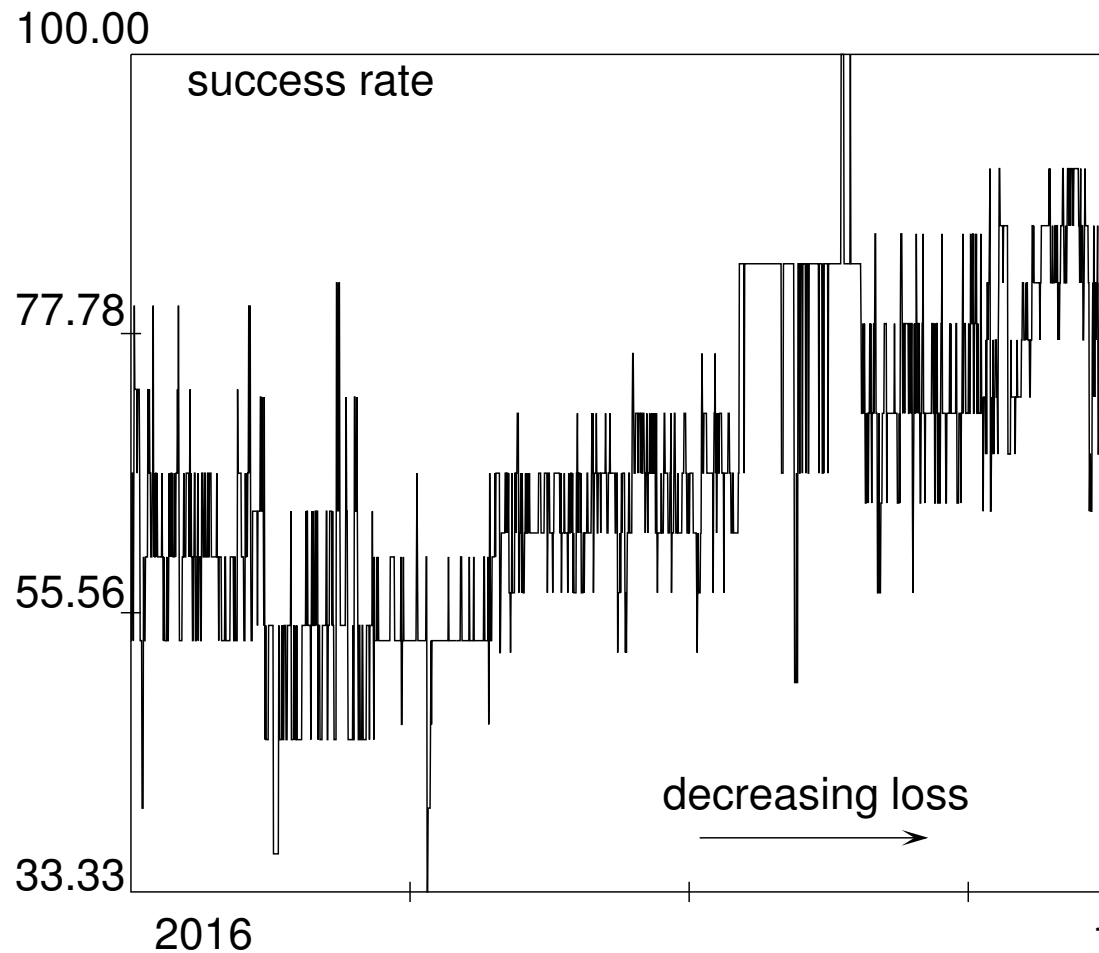
we expect a curve where loss $\searrow \Rightarrow$ SR $\nearrow$

Hybridization



We expect a curve where loss $\searrow \Rightarrow$ SR $\nearrow$

what we get

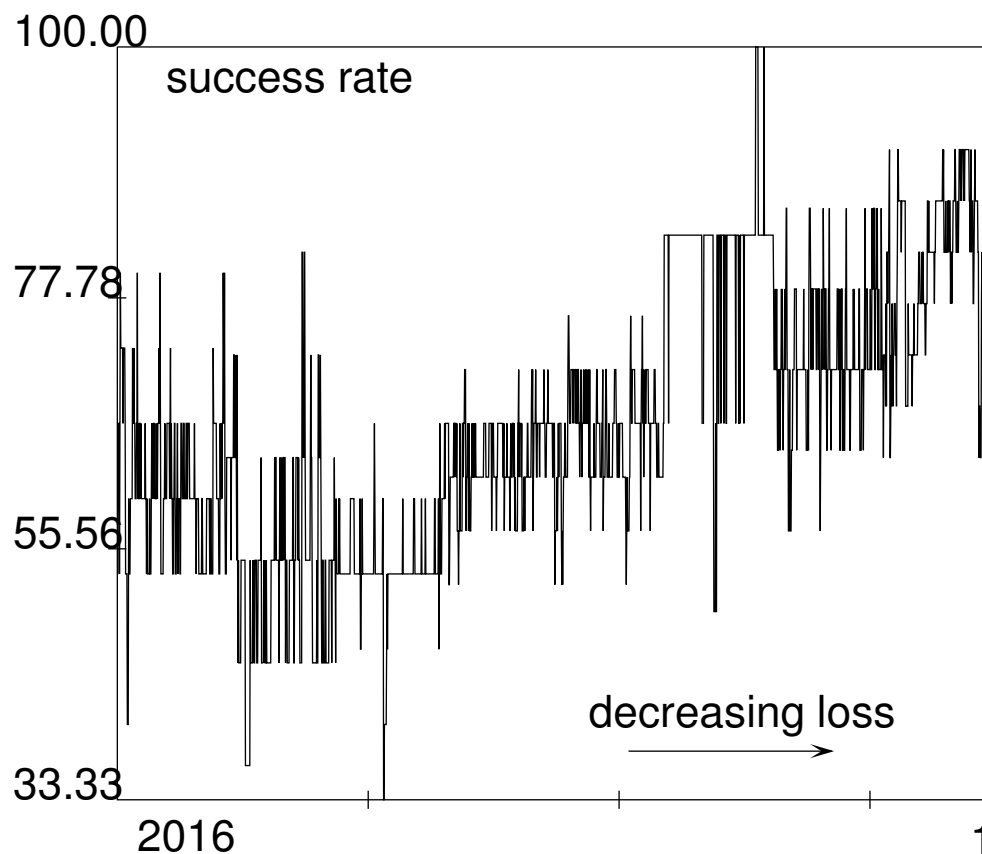


Hybridization



- **chaotic** behavior
- left part: loss $\searrow$, SR $\searrow$
- then loss $\searrow$, SR trend $\nearrow$, SR = 100 for some MLPs
- extreme right part: SR trend $\searrow$, SR $\neq$ 100 for min loss

mathematically
explainable





New training strategy

- still use the optimizer to decrease the loss
- compute the SR at each decrease of the loss

but we use the following rules to select MLP:

- if SR reaches 100 select the MLP with SR=100 and lowest loss
- if not, select up to 3 MLPs
 - select the MLP with maximal SR
 - select the MLP with the lowest number of Newton failure
 - select the MLP for which Newton has converged the less time to a solution $\neq$ expected solution



New training strategy

We have **missed solution(s)** for a learning set after training

- check if the missed solutions are not found by other MLPs
- **if not**
 - create a learning set for a missed solution using the strategy used for isolated node
 - train the MLP: same training strategy + ensure that the missed solution is found by the created hybrid MLP
- repeat until there is no more missed solutions

Result

a set of hybrid MLPs that find **all solutions** for **all samples of all learning sets**



Exact solutions

For now we get a solution $\mathbf{X}_a$ if $\|\mathbf{F}(\mathbf{X}_a)\| \leq \epsilon$, ϵ is a "small" threshold compatible with the available numerical accuracy



Exact solutions

Let X_r be a true solution

- exact solution: a X_a such that
 - X_a may be refined at will so that we have $\|X_a - X_r\| \leq \epsilon$ where ϵ is an almost arbitrary low number (limit: $10^{-38654705646}$)



Exact solutions

Let X_r be a true solution

- **exact solution**: a X_a such that
 - X_a may be refined at will so that we have $\|X_a - X_r\| \leq \epsilon$ where ϵ is an almost arbitrary low number (limit: $10^{-38654705646}$)
- **certified and exact solution**: a X_a such that
 - X_a may be refined at will
 - there is a ball centered at X_a of radius B_0 which includes a single true X_r
 - B_0 is an almost arbitrary low number (limit: $10^{-38654705646}$)



Exact solutions

Getting exact solutions requires **extended arithmetic** and **interval arithmetic** (available in Maple, Mathematica, C)

We use **Kantorovitch theorem** at $\mathbf{X}_a$

- Jacobian of $\mathbf{F}$ has an inverse $\Gamma(\mathbf{X}_a)$ and $||\Gamma(\mathbf{X}_a)|| = A_0$
- $||\Gamma(\mathbf{X}_a)\mathbf{F}(\mathbf{X}_a)|| = B_0$, $\mathcal{B}$: ball centered at $\mathbf{X}_a$, radius B_0
- $||Hessian(\mathbf{X})|| \leq C \forall \mathbf{X}$ in $\mathcal{B}$
- if $2nA_0B_0C < 1$ then
 - there is a single root of $\mathbf{F}$ in $\mathcal{B}$
 - Newton with guess $\mathbf{X}_a$ will converge to this root

C calculated with interval arithmetic

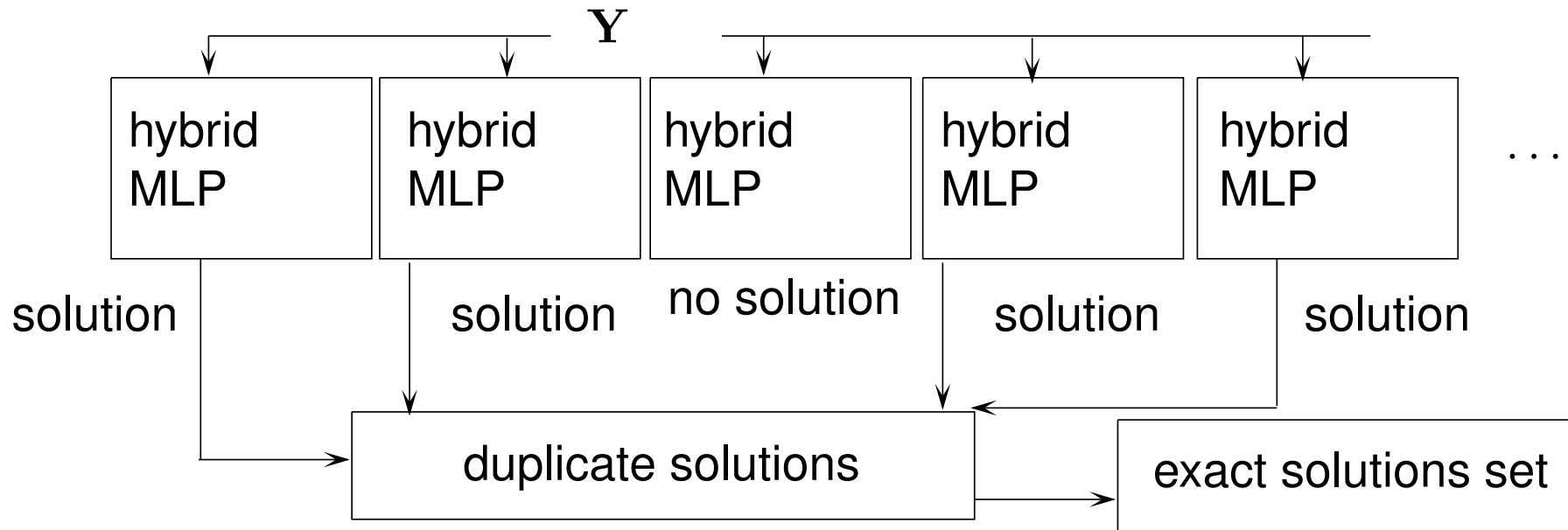


Exact solutions

Getting $\mathbf{X}_a$ with accuracy ϵ , calculation accuracy in digits: A
we assume $2nA_0B_0C < 1$ but $B_0/2 > \epsilon$

- increase A
- run Newton with residues $< 10^{-A/2} \rightarrow F(\mathbf{X}_a) \searrow \rightarrow B_0 \searrow$
- stop if $B_0/2 \leq \epsilon$, otherwise repeat

The solver





Verification

Current solver provides all solutions for all learning sets, what about a random $\mathbf{Y}$?

- we may create **verification sets** with samples $(\mathbf{Y}_r, \{\mathbf{X}_r^1, \mathbf{X}_r^2, \dots, \mathbf{X}_r^m\})$
 - choose randomly $\mathbf{Y}_r$, different from every $\mathbf{Y}$ in the learning sets
 - use the elements $(\mathbf{Y}_k, \mathbf{X}_k^j)$ of the initial solution set as starting point for a continuation process from $\mathbf{Y}_k$ to $\mathbf{Y}_r$ to find the solutions for $\mathbf{Y}_r$



Verification

Current solver provides all solutions for all learning sets, what about a random $\mathbf{Y}$?

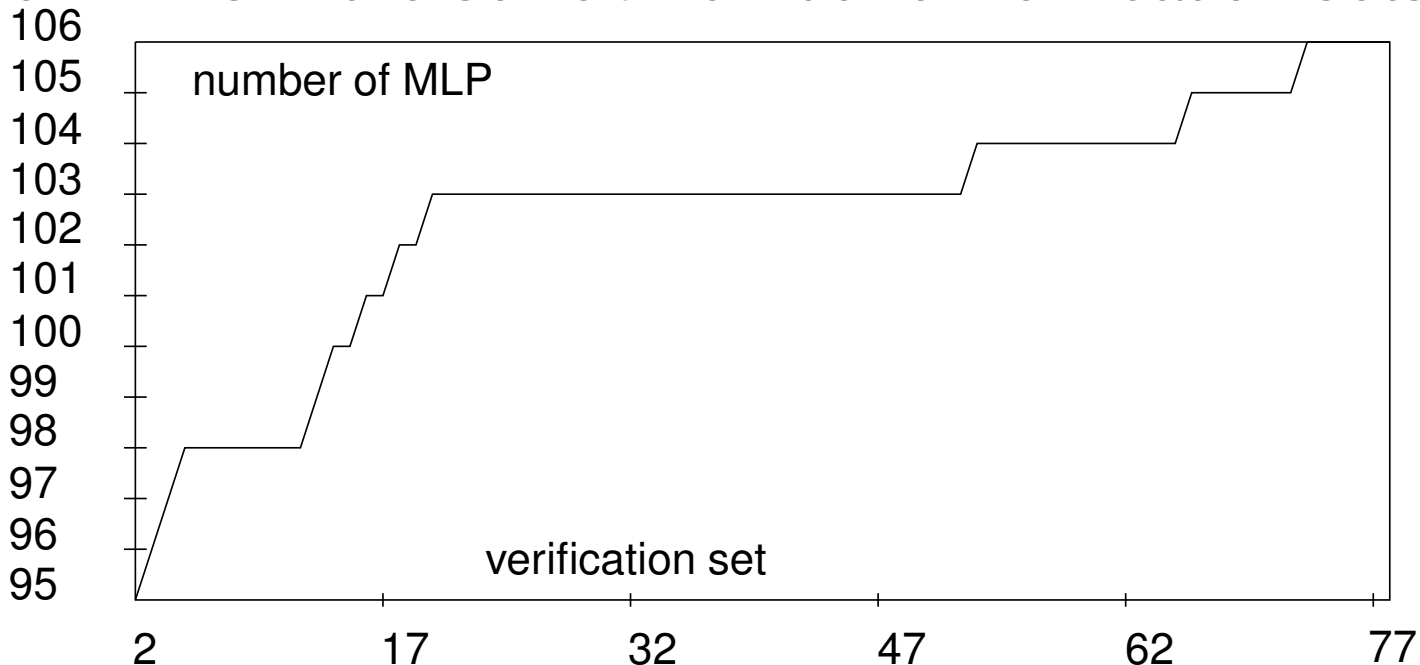
- we may create **verification sets**: $\mathbf{Y}_r, \{\mathbf{X}_r^1, \mathbf{X}_r^2, \dots, \mathbf{X}_r^m\}$
- check with the solver
- if missed solution(s)
 - create learning set from the missed solution(s)
 - train an hybrid MLP
 - add the MLP to the solver

Note: it is possible to trim the solver MLPs so that we have less MLPs while still recovering all solutions for all learning and verification sets

Verification



number of MLPs in the solver/ number of verification sets



20 verification sets $\rightarrow$ 103 MLPs, after 78 verification sets 3 missed solutions $\rightarrow$ 0.0005% of the total number of solutions

we have a **self-learning** capacity



Conclusion

- tested on various problems in engineering
- **solver creation time**: a few hours
- **solving time**: between 1ms and 0.5 seconds/samples
- solver devoted to the case where multiple occurrences of Y have to be considered
- solver provides
 - **exact** solutions
 - **but no guarantee** to find them all

software freely available at <https://gitlab.inria.fr/merlet/nn-pes>

- software developed with the help of the 3IA Techpool
- version 0 will be available end of October 2026
- version 1: creation/solving time ↘, soon ...