Discovering Symbolic Models from Deep Learning with Inductive Biases

SAPIENZA
UNIVERSITÀ DI ROMA

Bernardo Ricci

In this paper it's presented a study and some revision attempts of the symbolic regression model described in the article ***"Discovering Symbolic Models from Deep Learning with Inductive Biases"*** by Miles Cranmer et al.

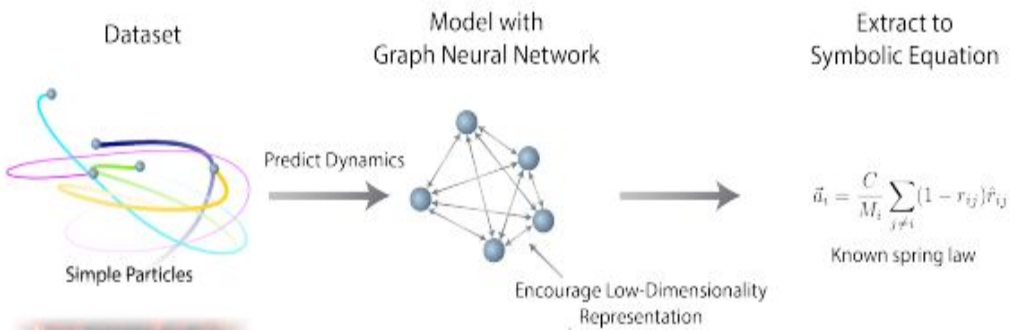
The aim of the work is to obtain the analytical expression of the interaction of a particle system from data on the dynamics of the system itself.

Therefore, illustrate how a neural network can be written that is able to impose a specific representation on an intermediate layer of neurons through the use of an inductive bias and that is able to obtain, from these, the analytical expression of the interactions.

In the study I put the focus on verifying the capabilities of the network by comparing the results obtained with interaction models and we have known to improve its performance by modifying its structure and loss function.

DATASET

The dataset is then set up from 10000 simulations of 4 particles living in a two-dimensional environment and which are each described by 6 features: *(x, y, v_x, v_y, q , m)* interacting under a potential.



Interaction	Potential
$r1$	$-m_1 m_2 log(r)$
$r2$	$-\frac{m_1 m_2}{r}$
$spring$	$(r-1)^2$
$charge$	$\frac{q_1 q_2}{r}$

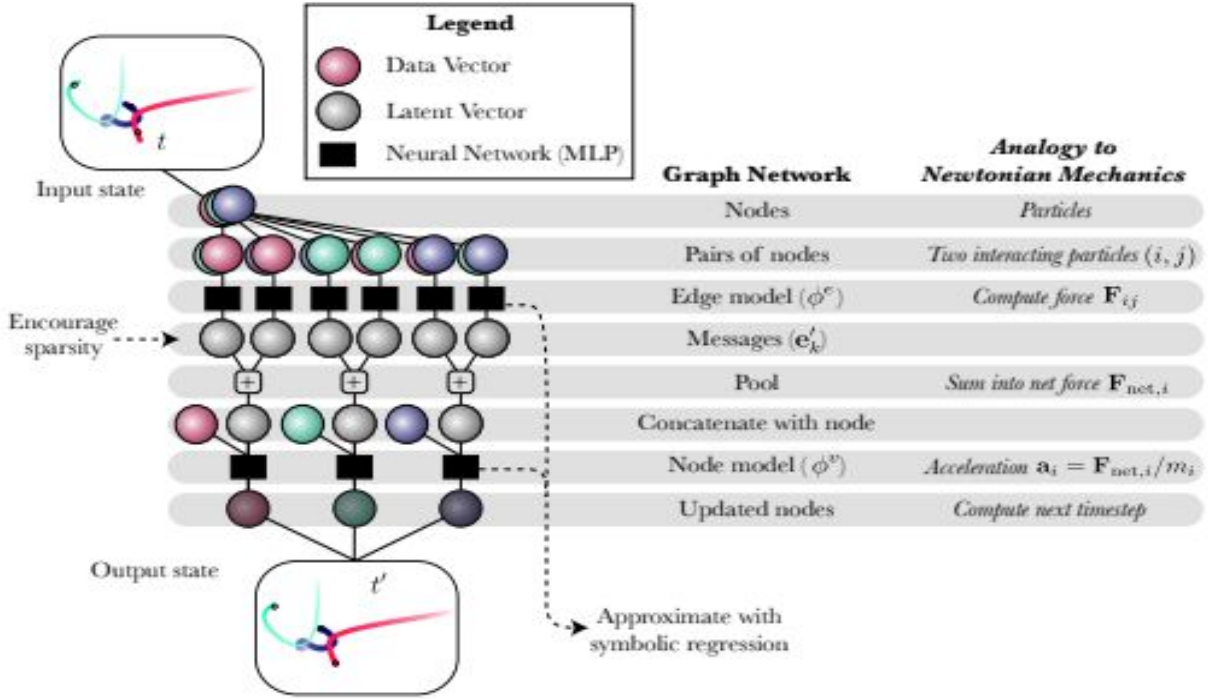
DATA REPRESENTATION

The network can be divided into two separate sections:

- The first is a supervised learning which has the objective of obtaining data regarding the interactions between the particles.
- The second is unsupervised learning with the aim of obtaining the analytical expression of this interaction

The program use Graph Neural Network with inductive bias. These algorithms are typically composed of three sections:

- The **edge model (ϕ_e)**: a pair of nodes (v_i, v_j) interacting through an edge is mapped through a “message function" ϕ_e in a message vector. This operation is repeated for each receiving node (i.e. that "undergoes" interaction) and the messages are then aggregated.
  The chosen aggregation operation is the vector sum on all nodes that they send the message;
- The **node model (ϕ_v)**: the aggregate message and the receiving node are processed by a new function ϕ_v which calculates the properties of this node is useful for updating its status;
- The **global model (ϕ_u)**: finally the new state of the graph is calculated using the information obtained



ϕ_e , ϕ_v, ϕ_u are approximated using multilayer perceptrons (MLP)

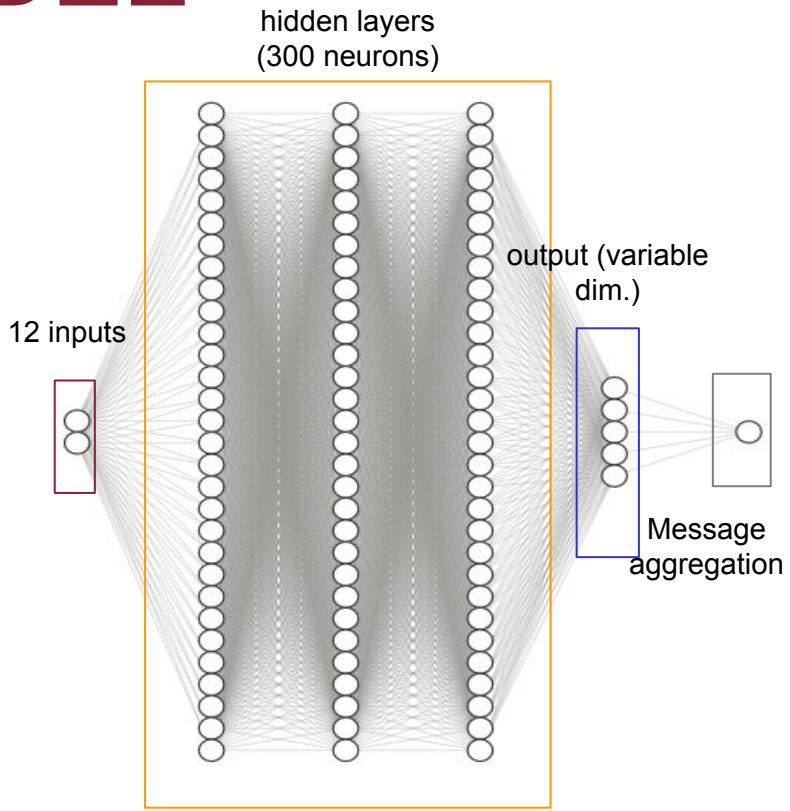
EDGE MODEL

The first MLP plays the role of the function ϕ_e:

- It is a structure with input features of two nodes, therefore 12 inputs;
- At least 3 hidden layers, of 300 neurons each;
- Output of variable size depending on the model used.

Activation function is **ReLU**

The output play a fundamental role in our study as it will represent the interaction between two particles.
This MLP is used for all pairs of particles fixed to a node and on the messages obtained it performs the operation aggregation.



AGGREGATION

Then messages are then pooled via element-wise summation for each receiving node into the summed message.

The interpretation we want to give to the messages, output of the first MLP, is to be the acceleration determined by the interaction between two nodes.

The inductive bias has the very purpose of forcing this representation and consists in using the messages as if they were actually the force between two nodes.

The aggregation operation must therefore add up all the message vectors relating to a node as, in Newtonian mechanics, to obtain the total acceleration of a body, all the forces acting on it are vectorically added.

In essence we will try to teach the algorithm what a force is.

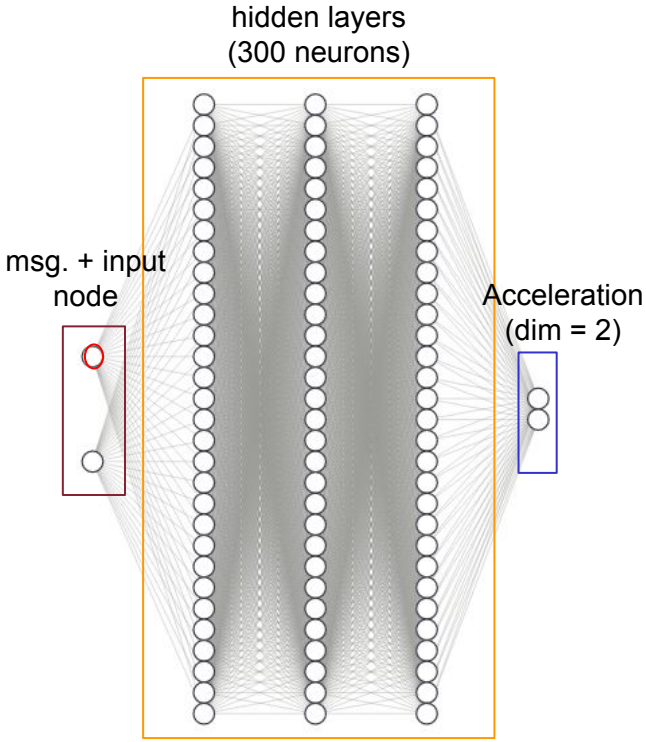
NODE MODEL

The second MLP has as the first 3 internal layers of 300 neurons and as output a two-dimensional vector, i.e. what we would like it to be the total acceleration of the particle.

- Takes in input the aggregation of first MLP and has dimensionality of the message + node;
- Has 3 layers of 300 neurons;
- The output is a 2D array.

In fact, it will be the output of this second level that will be compared with the expected acceleration value, training the network. This allows the system to succeed in the task of calculating the trajectory of the particle, even if the regression fails symbolic.

In fact, the acceleration calculated in this way is then used to find the position and speed of the particles in the following instant of time.



LOSS

The loss function is a function used to evaluate a possible candidate of the set of weights of a neural network, comparing the result obtained from the network with an expected result.

Defining it in this way is therefore only possible in supervised learning algorithms, where the goal is to minimize the distance (appropriately defined) between the network output and the expected one.

I've tested three functions: *"Mean Squared Error"*, *"Mean Absolute Error"* and *"Huber Loss"* and compared the results obtained (only the MAE was present in the original work).

$$MSE : L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

$$MAE : L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

$$HUBER : L(y, \hat{y}) = \begin{cases} \frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 & |y_i - \hat{y}_i| < 1 \\ |y_i - \hat{y}_i| - 1 & otherwise \end{cases}$$

REGULARIZATION

The regularizer is a function which is added to the loss and which modifies its distribution of information to avoid overfitting.
In this case it encourages sparsity, i.e. using the potential of a large dimensional parameter space but then concentrating information in as few components as possible.

$$\mathcal{L} = \mathcal{L}_v + \alpha_1 \mathcal{L}_e + \alpha_2 \mathcal{L}_n, \text{ where}$$

the prediction loss is one between MSE, MAE or HUBER

$$\text{the message regularization is } \mathcal{L}_e = \frac{1}{N^e} \begin{cases} \sum_{k \in \{1:N^e\}} |\mathbf{e}'_k|, & \text{L}_1 \\ 0, & \text{Bottleneck} \end{cases}$$

$$\text{with the regularization constant } \alpha_1 = 10^{-2}, \text{ and the}$$

$$\text{regularization for the network weights is } \mathcal{L}_n = \sum_{l=\{1:N^l\}} |w_l|^2,$$

$$\text{with } \alpha_2 = 10^{-8},$$

This is the approach used for all models, except for KL

KL

KL model is a variational version of the GNN, which models the messages as distributions.

Has a normal distribution for each message component with a prior of μ = 0, σ = 1. The output of ϕ_e should now map to twice as many features as it is predicting a mean and variance.

The first half of the outputs of ϕ_e now represent the means, and the second half of the outputs represent the log variance of a particular message component.

Every time the GNN is run, the mean and log variance of messages is calculated, is sampled each message to calculate the output message non aggregated, and pass those samples through a sum (like the aggregation) and then pass that value through to compute ϕ_v a sample of the predicted value.

$$
\begin{aligned}
\boldsymbol{\mu}'_k &= \phi^e_{1:100}(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}), \\
\boldsymbol{\sigma}'^2_k &= \exp\left(\phi^e_{101:200}(\mathbf{v}_{r_k}, \mathbf{v}_{s_k})\right), \\
\mathbf{e}'_k &\sim \mathcal{N}(\boldsymbol{\mu}'_k, \mathrm{diag}(\boldsymbol{\sigma}'^2_k)), \\
\bar{\mathbf{e}}'_i &= \sum_{k \in \{1:N^e | r_k = i\}} \mathbf{e}'_k, \\
\hat{\mathbf{v}}'_i &= \phi^v(\mathbf{v}_i, \bar{\mathbf{e}}'_i),
\end{aligned}
$$

The regularization term is the following:

($\alpha_1 = 1$)

$$
\mathcal{L}_e = \frac{1}{N^e} \sum_{k=\{1:N^e\}} \sum_{j=\{1:L^{e'}/2\}} \frac{1}{2} \left(\mu'^2_{k,j} + \sigma'^2_{k,j} - \log\left(\sigma'^2_{k,j}\right) \right),
$$

PySR

PySR is an open-source library for practical symbolic regression built in the Julia environment, a type of machine learning which aims to discover human-interpretable symbolic models.

It use multi-population evolutionary algorithm, simulated annealing and gradient descent optimization to find the equation that best describes the data.

These techniques are similar to natural selection (genetic algorithms) where the 'fitness' of each expression is defined in terms of simplicity and accuracy and the space of expressions is explored through mutations of the same.

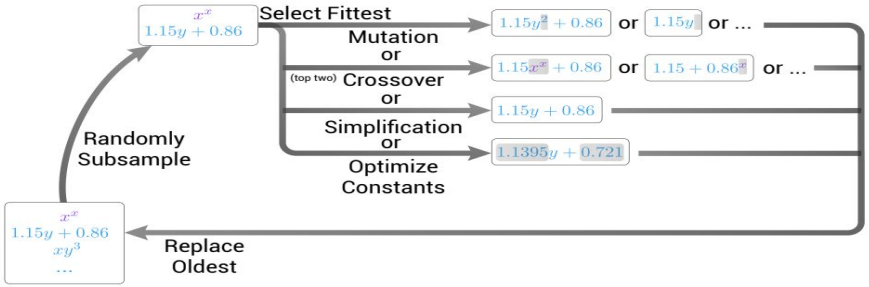
PySR build an equation starting from a closed set of operators.

GENETIC ALGORITHM

- Assume one has a population of individuals, a fitness function, and a set of mutation operators;
- Randomly select an n_s-sized subset of individuals from the population;
- Run the tournament by evaluating the fitness of every individual in that subset;
- Select the fittest individual as the winner with probability p. Otherwise, remove this individual from the subset and repeat this step again. If there is one remaining, select it.

  The probability for rejection is $p = \exp\left((L_F - L_E)/ \alpha T \right)$, for L_F and L_E the fitness of the mutated and original individual respectively, α a hyperparameter and $T \in [0,1]$ called Temperature.

- Create a copy of this selected individual, and apply a randomly-selected mutation from a set of possible mutations.
- Replace a member of the population with the mutated individual. It's replaced the oldest one (non-aging evolution). It allows not to focus too early on inaccurate structures and to observe a wider space of variables



Instead of searching the parameter space of a defined function, it's searched the space for all possible mathematical formulas, starting with a finite set of operators. Functions are represented as combinations of these in binary trees.

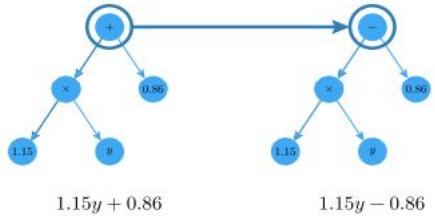


Figure 1: A mutation operation applied to an expression tree.

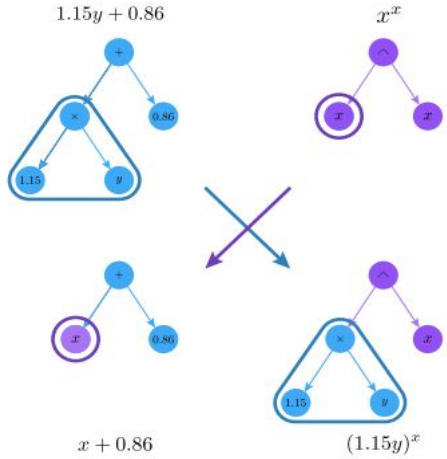


Figure 2: A crossover operation between two expression trees.

SCORE

To choose the best function proposed by PySR, I've choose the one with the best Score (fitness)

$$score = \frac{-\log(loss_i / loss_{i-1})}{(complexity_i - complexity_{i-1})}$$

Where complexity is related to the number of nodes and the loss is a MSE

PySR

APPLICATION

I've applied PySR to the best message i.e. the one with the lowest validation loss.

I've made 30 iterations starting from a population of 30 elements. That means hundreds of thousands of mutations and expression evaluations.

The set of operators is small to minimize the space of functions to search for

```
default_pysr_params = dict(
    populations=30,
    model_selection="best",   #selects the candidate model with the highest score among expressions with a loss better than at least 1.5x the most accurate model.
)

model_pysr = PySRRegressor(
    niterations= 30,
    binary_operators=["plus", "sub", "mult", "div"],
    unary_operators=["cube", "square"],
    **default_pysr_params
)
```

Complexity	Loss	Score	Equation
1	1.716e+01	5.001e-06	-0.8927252
3	1.622e+01	2.829e-02	(-2.918856 + r)
5	1.533e+01	2.812e-02	((dy + dx) ∕ r)
6	1.508e+01	1.682e-02	cube((dy + dx) ∕ r)
7	1.453e+01	3.674e-02	(((dy + dx) ∕ r) - 0.8953305)
8	1.393e+01	4.215e-02	((dy - r) ∕ (cube(r) + 0.034539152))
9	1.353e+01	2.932e-02	((dy - square(r)) ∕ (cube(r) + 0.008684006))
10	1.313e+01	3.020e-02	(((dy - r) ∕ (r + cube(r))) ∕ 0.20300612)
11	1.301e+01	9.357e-03	(((dy - r) + dy) ∕ (cube(r) + square(0.18225443)))
12	1.171e+01	1.050e-01	((((dx - r) + dy) ∕ (r + cube(r))) ∕ 0.21675915)
14	1.140e+01	1.320e-02	(((((dy ∕ 0.7800792) - r) + dx) ∕ (r + cube(r))) ∕ 0.22577576)
15	1.130e+01	8.885e-03	(((((dy ∕ square(0.7800792)) - r) + dx) ∕ (r + cube(r))) ∕ 0.2... 2577576)
16	1.122e+01	7.546e-03	(-2.1055453 * ((((dy ∕ r) * 2.284518) + (dx - 1.1395864)) ∕ (-... 0.469566 - square(r))))
17	1.117e+01	3.967e-03	((((((dy ∕ square(0.7800792)) - r) + dx) ∕ (r + cube(r))) ∕ 0.... 22577576) + 0.22577576)
18	1.094e+01	2.097e-02	(-2.1055453 * (((dy * (2.284518 ∕ r)) + ((dx ∕ 0.5545009) - 1....

MODIFICATIONS OF THE PROGRAM

- Tested Models
- Loss
- ACPCR and ACPER
- Regularization
- Standard Deviations over epochs

TESTED MODELS

For the first MLP I've tested 5 different GNN. The first three are already in the original work, the last 2 are implemented by myself:

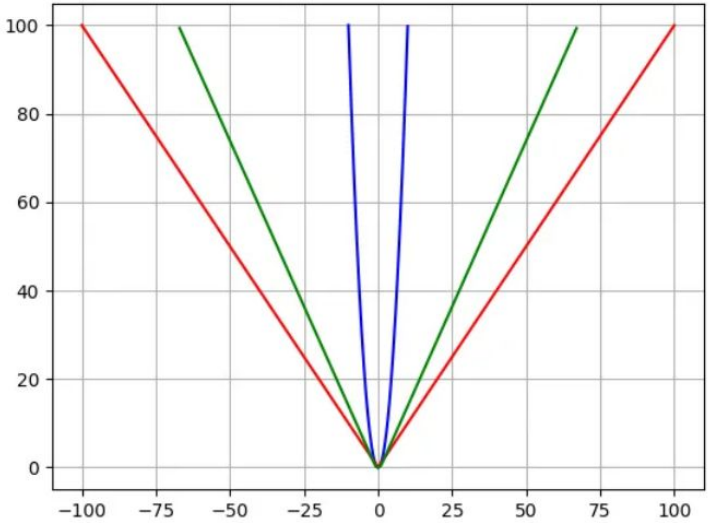
- "Bottleneck" model, a GNN with 3 hidden layers of 300 neurons in which the dimension of the output will have a dimension equal to that in which the particles live;
- "L1" model, a GNN with 3 hidden layers of 300 neurons and the output of 100 messages components and a L1 regularization loss term on the messages;
- "KL" model, same of "L1" with the Kullback-Leibler (KL) regularization;

- "Plus-Minus" model, composed of 5 consecutive linear layers of 300, 450, 600, 450, 300 neurons and an outgoing message of 100 messages components, with L1 regularization;
- "CUSTOM" model, denoting with X the output of the 1st linear dense layer, 2 independent linear layers are then instantiated to respectively perform linear transformations over $1/(1+\mathrm{ReLU}(X))$and $1/(1+\mathrm{ReLU}(X))^2$, where the unit is added to avoid 0 division errors.

  Also this has an output of 100 messages components and L1 regularization.

LOSS

The original work used only MAE, so I implemented and used MSE, MAE and Huber Loss



MAE (red), MSE (blue), and Huber (green) loss functions

ACPCR - ACPER

To evaluate how much a particular model works, I defined the Almost Correct Prediction Correct Rate (ACPCR) and Almost Correct Prediction Error Rate (ACPER)

$$ACPCR = \frac{1}{N}\sum_{i=1}^{N}\gamma_i$$

$$\begin{cases}\gamma_i = 1 & |y_i - \hat{y}_i| < 0.02 \\ 0 & otherwise\end{cases}$$

$$ACPER = \frac{1}{N}\sum_{i=1}^{N}\delta_i$$

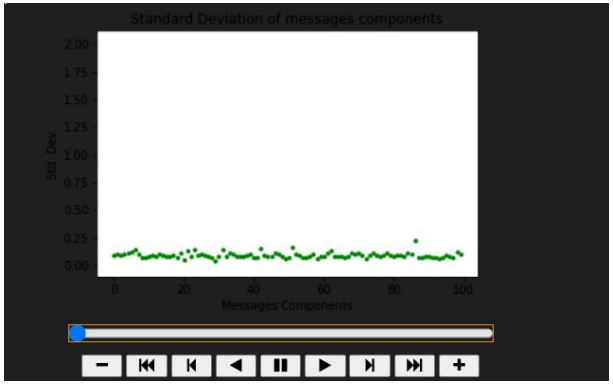
$$\begin{cases}\delta_i = 1 & |y_i - \hat{y}_i| > 0.02 \\ 0 & otherwise\end{cases}$$

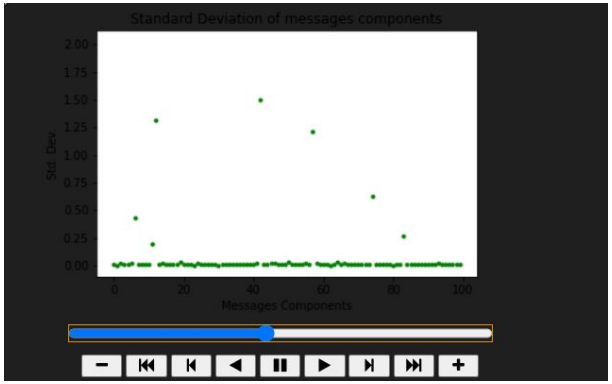
$$ACPER + ACPCR = 1$$

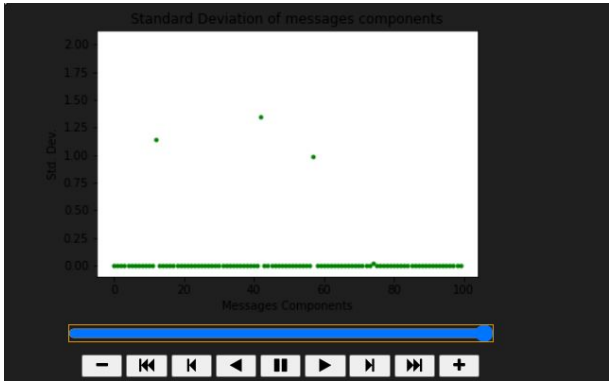
Higher the ACPCR, better the model. The opposite for the ACPER.

REGULARIZATION EFFECT

To observe the effect of the regularizer and identify the relevant messages I used the standard deviation of the message components related to the data over all the available simulations and times. The idea is that the only entries that would be non-zero are those whose value changes from interaction to interaction.





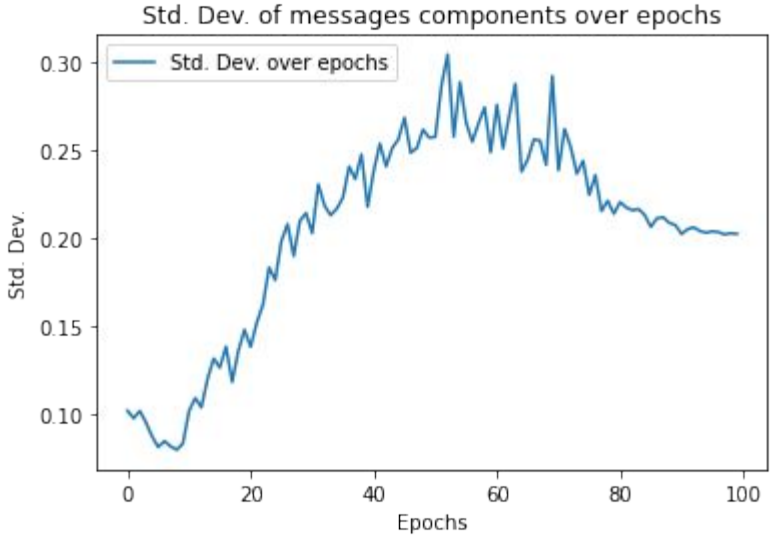


If I check the mean, all components are going to zero, and this is what I expect because the mean value of all interactions of a system without external forces would have to be really zero.

EPOCHS CHOICE

Based on this behaviour of Standard Deviation, I studied the overall standard deviation of all messages components over epochs. From a certain point the standard deviation goes almost asymptotically around a certain value. It means that the std. deviation of most of messages components are now at zero and only the interesting entries survived.

I found that 100 epochs can be a good choice. More epochs is for sure better, but the compilation time will be too long.



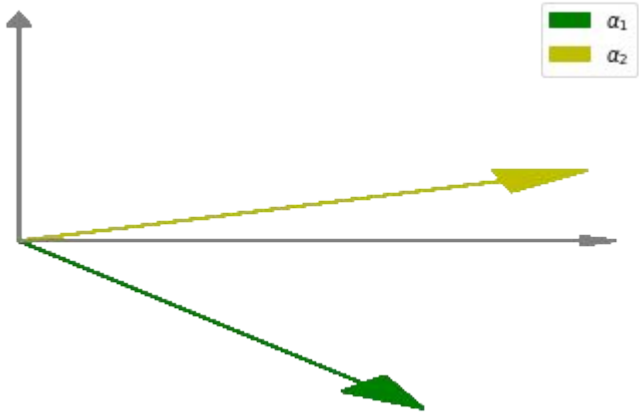
RELATION WITH PHYSICAL FORCES

Let's call m_1 and m_2 the relevant messages components, it's possible to write them as linear combination of the force.

$$m_i = \alpha_{i,x} F_x + \alpha_{i,y} F_y + \alpha_{i,bias}$$

So is possible to reconstruct $\vec{\alpha}_1$ and $\vec{\alpha}_2$ with a linear fit on m_1 and m_2.

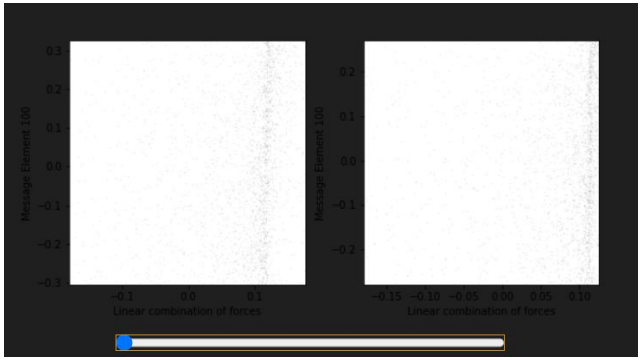
To do that the only way is to know the analytic expression of the force, that is possible to obtain thanks to PySR.



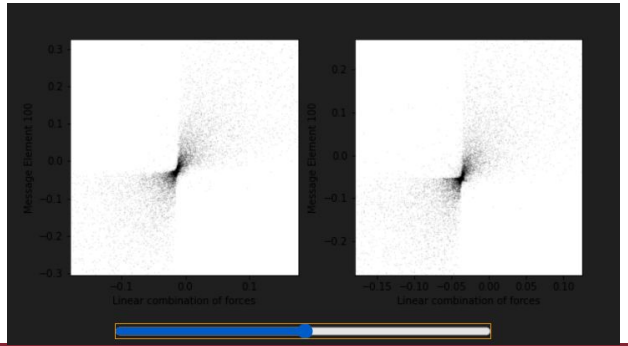
RELATION BETWEEN FORCE AND MESSAGE

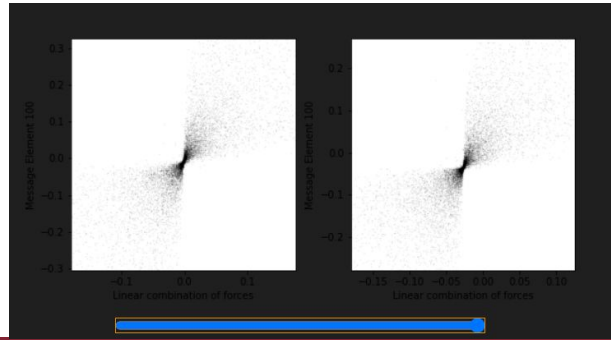
The regression does not assure that the components of the message actually represent the forces.

We could in fact stay linearly inserting a nonlinear relation, resulting in expressions without any meaning.



The relationship between physical forces and relevant components of the message becomes ever closer to the identity going forward in the training of the network.



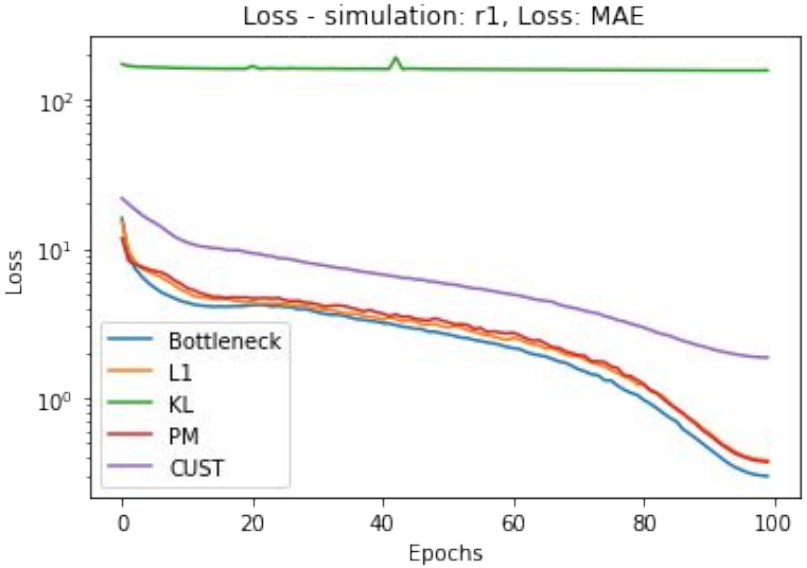


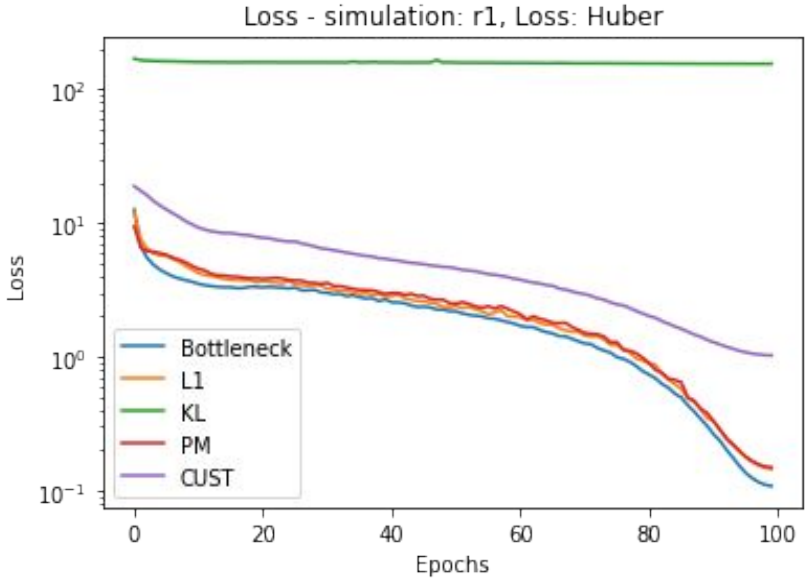
RESULTS

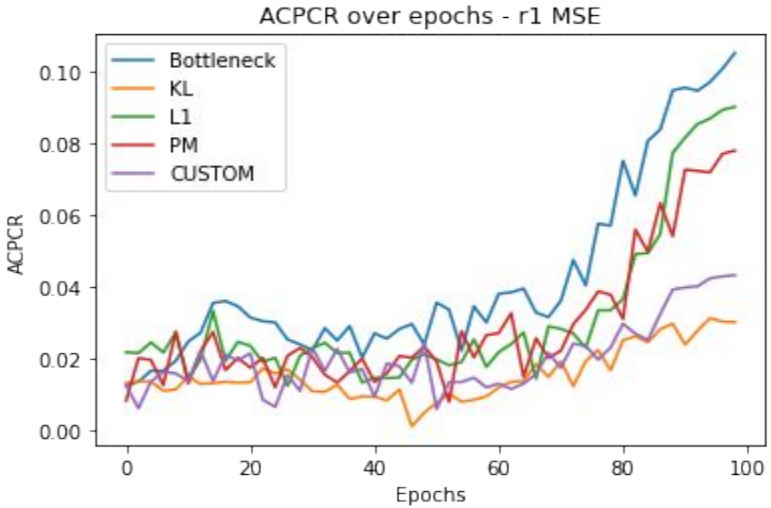
- Comparison between Models

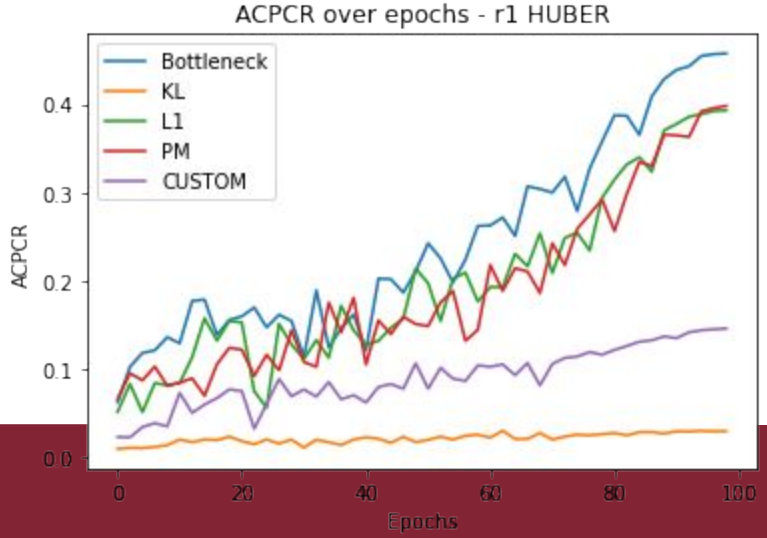
- Analytical expressions

LOSS FOR DIFFERENT MODELS



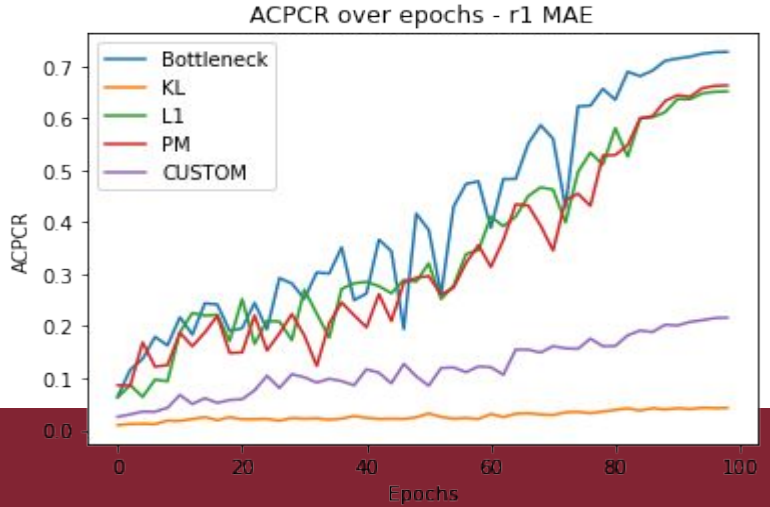




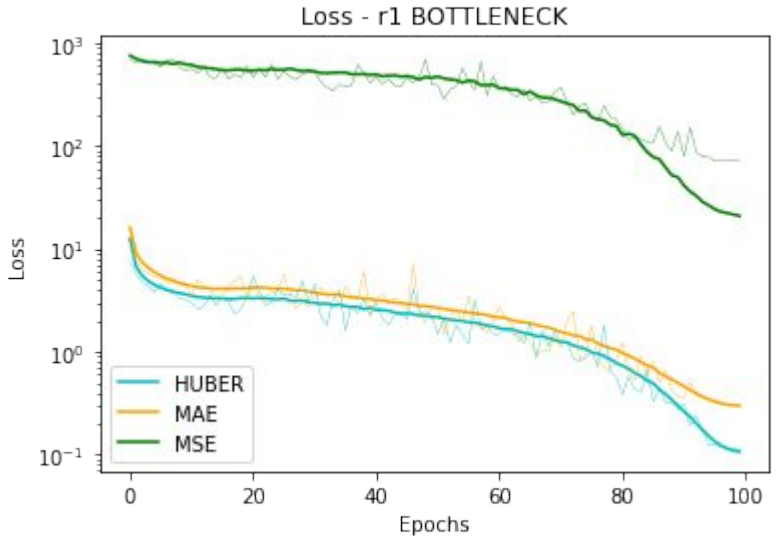


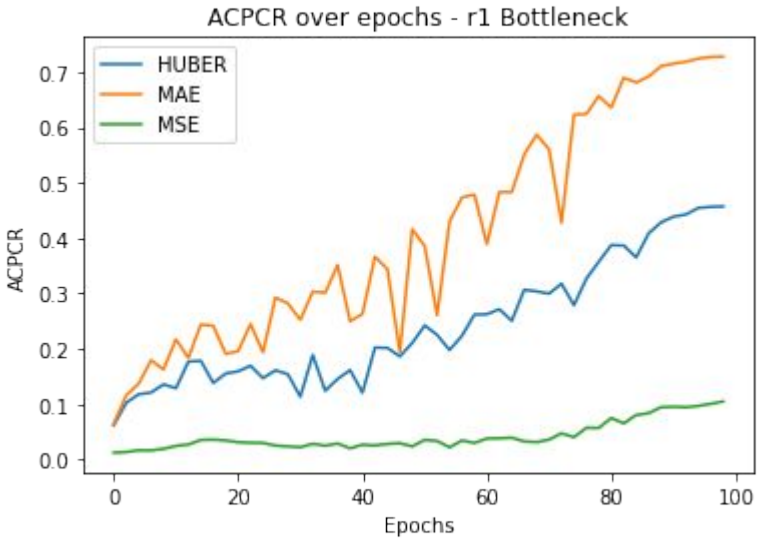
At fixed simulation, I've studied the ACPCR for different losses

MAE gives the best ACPCR

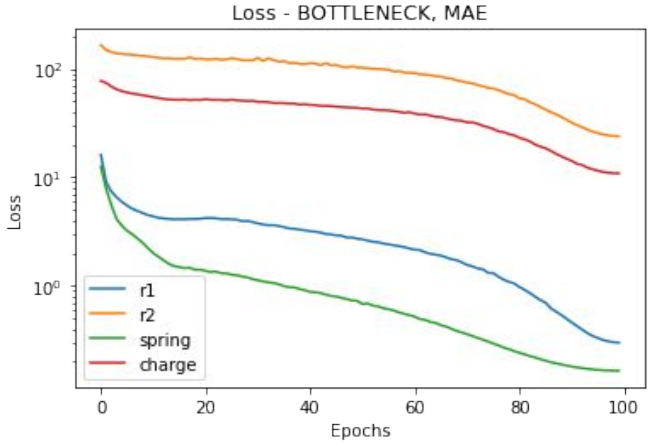


Fixed model and simulations, behaviour of different losses

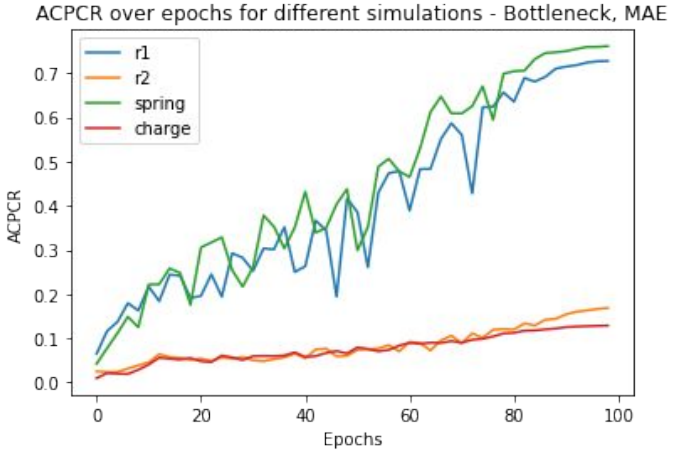


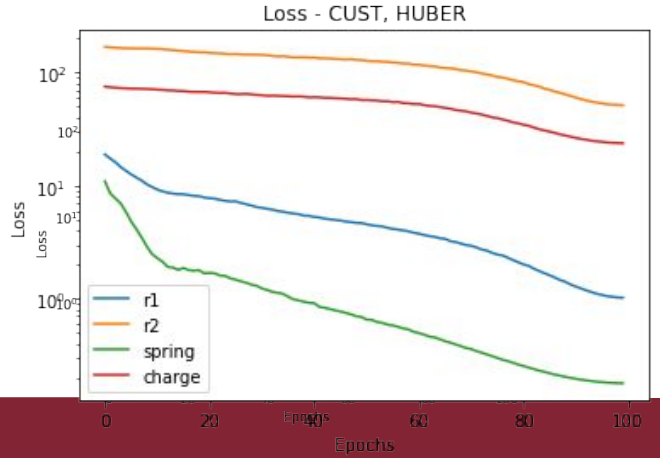


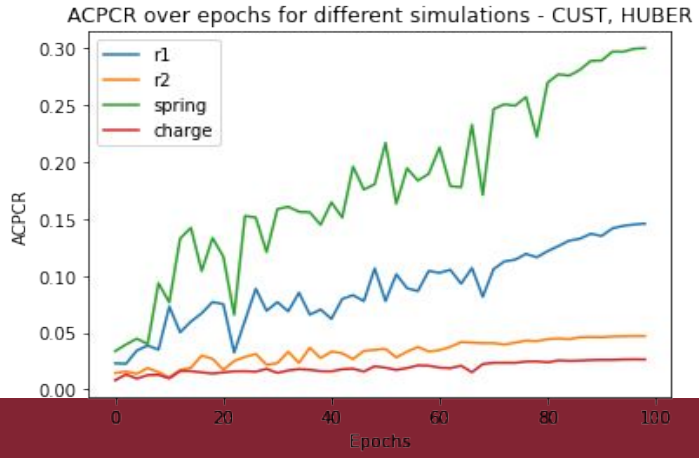
Fixed model and loss



r1 and spring are the simulations that works better







ANALYTICAL EXPRESSIONS

Interaction	ACPCR	Equation expected	PySR
r1	0.73	$-m_2 \frac{dx+dy}{r^2}$	$m_2 \frac{dy-dx}{r^2}$
r2	0.17	$-m_2 \frac{dx+dy}{r^3}$	$\frac{-0.40086803}{r^2} dx$
spring	0.76	$-m_2(r-1)\frac{dx+dy}{r}$	$-1.0401107dy$
charge	0.13	$q_1 q_2 m_2 \frac{dx+dy}{r^3}$	$\frac{-0.4539162}{r^2} dx$

Table 1: Bottleneck MAE

Interaction	ACPCR	Equation expected	PySR
r1	0.15	$-m_2 \frac{dx+dy}{r^2}$	$\frac{m_2}{(1.2337741+r+dx)^3}$
r2	0.05	$-m_2 \frac{dx+dy}{r^3}$	$\frac{r-dy}{r^3+0.005402562}$
spring	0.30	$-m_2(r-1)\frac{dx+dy}{r}$	$-0.34305963dy$
charge	0.03	$q_1 q_2 m_2 \frac{dx+dy}{r^3}$	$\frac{0.21675915(dx-r+dy)}{r+r^3}$

Table 2: CUST HUBER

CONCLUSIONS

- Bottleneck is the model that works better. Probably this is due to the fact that the output is of the same dimension of the physical system;
- MAE is the best loss;
- Charge and r2 doesn’t gives good results, the GNN doesn’t works so well to reconstruct these datasets;
- As expected PySR doesn’t work well for Charge and r2. Unexpected bad result for Spring. Best results with r1.

The inefficiency for Spring potential probably is due to a too much restricted set of operators;

Possible upgrades:

- Create models that brings output to the dimension of the physical system like Bottleneck;
- Try to use models with more layers and more neurons, it could help to extract more informations;
- Try other losses, such as RMSE;
- Insert a second inductive bias in PySR: that of the dimensional analysis of the interaction force. This bias would severely limit the size of the analytic function space that PySR explore, making it much more effective.