Machine Learning Light Hypernuclei

Isaac Vidaña, INFN Catania

EXOTICO: EXOTIc atoms meet nuclear COllisions for a new frontier precision era in low-energy strangeness nuclear physics

ECT*, Trento (Italy), October 17th-21st 2022

This work in a sentence

We employ a feed-forward ANN to extrapolate at large model spaces the results of *ab-initio* hypernuclear NCSM calculations for the Λ separation energy B_Λ of the lightest hypernuclei, obtained in accessible HO basis spaces using chiral NN, NNN & YN interactions

Based on:

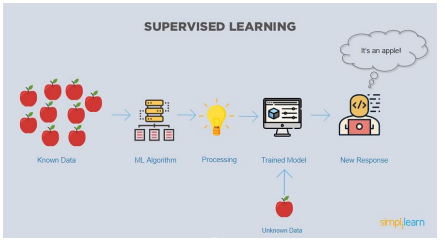
I.V. arXiv: 2203.11792

Machine Learning

Machine Learning is a branch of Artificial Intelligence whose scope is to *devise algorithms able to recognize patterns in previously unseen data without any explicit instructions by an external party*. Different types of ML include

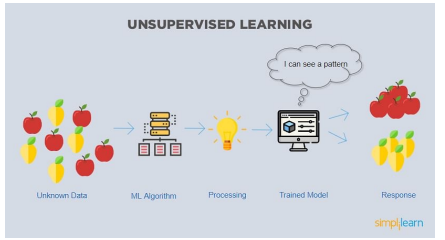
- **Supervised Learning**

  Known input-output (feature-label) relations are given to the machine learning algorithm to **trained** it and **infer a mapping therefrom**. Once the model is **trained based on the known data**, one can use unknown data into the model to get predictions. Used for **Classification** & **Regression** problems



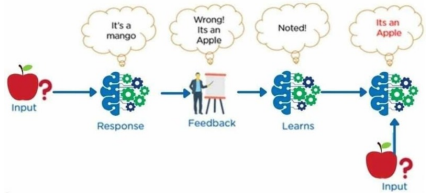
- **Unsupervised Learning**

  The output of the input training data is **unknown**. The input data is fed to the Machine Learning algorithm and is used to train the model which then is employed to **search for patterns in the data**. Used for **Clustering** & **Generation** problems



- **Reinforced learning**

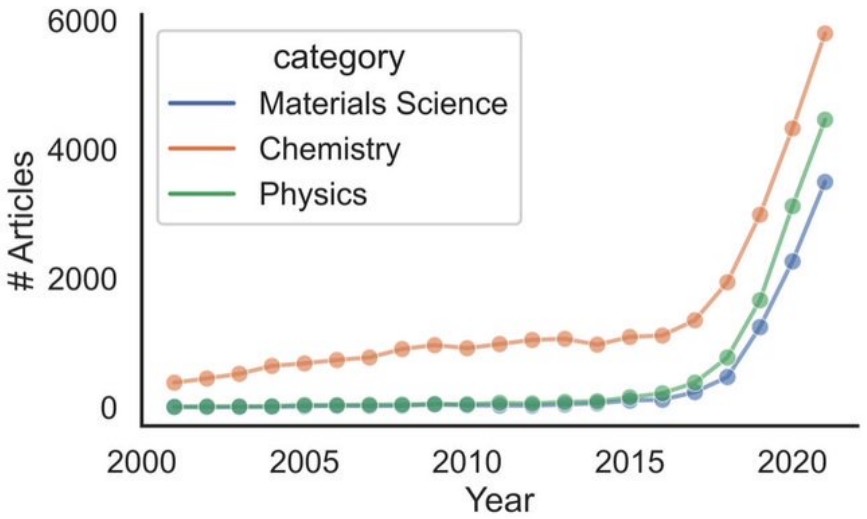
  Given a **framework of rules and goals**, an **agent** (algorithm) **learns in an interactive environment** by **trial and error** using **feedback from its own actions and experiences** and it gets **rewarded** or **punished** depending on which strategy it uses. Each **reward** reinforces the current strategy, while **punishment** leads to an adaptation of its policy. Example: **games** such as **Chess** or **Go**



Machine Learning in Physics

Machine Learning has been applied in different areas of physics that include among others:

- Condense matter
- Statistical physics
- Cold atoms
- Quantum many-body theory
- Quantum computing
- Cosmology
- Particle physics
- Nuclear physics
- …

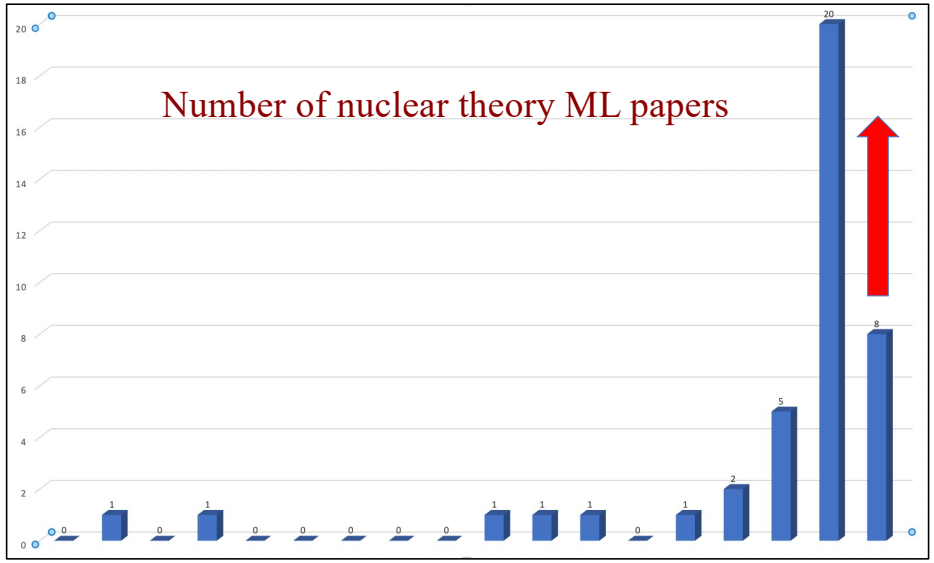


A **spectacular increase** of the number of publications related with AI or ML is observed in physical sciences in the last years

Machine Learning Applications in Nuclear Theory

Since the pioneering work of *Gazula et al., NPA 540 1 (1992)*, who employed a **feed forward neural network to study global nuclear properties across the nuclear landscape**, Machine Learning has been used to predict

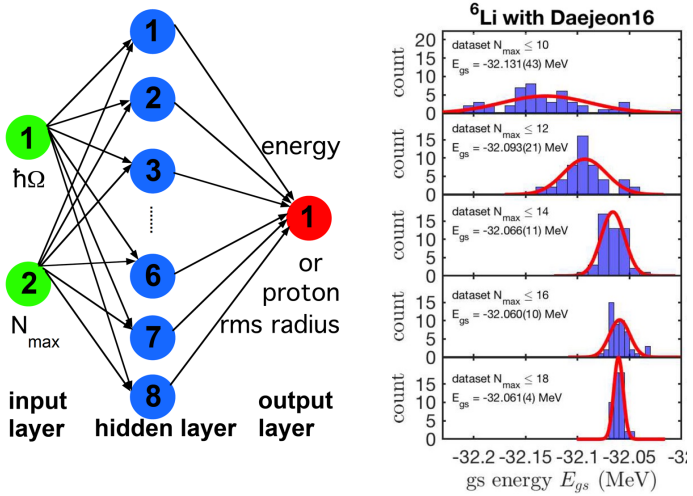
- Nuclear masses & charge radii
- α- & β-decay half-lives
- Fission yields
- Fusion reaction cross sections
- Isotropic cross-sections in proton-induced spallation reactions
- Ground and excited state energies
- Dripline locations
- The deuteron properties
- Proton radius
- Liquid-gas phase transition
- Nuclear energy density functionals
- Neutron star EoS
- The nucleon axial form factor from neutrino scattering
- Extrapolation of A-body results with ANN
- …



Machine Learning Applications in Nuclear Theory

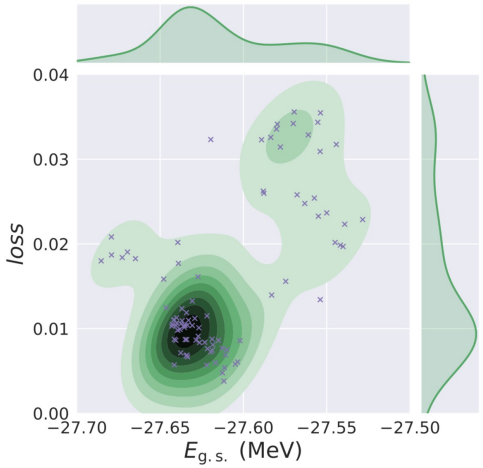
Recently, ANN have been employed to **extrapolate the results of *ab-initio* nuclear structure calculations in finite model spaces**. Particularly:

- *Negoita et al., PRC 99, 054308 (2019)* have used a feed-forward ANN method for predicting the **ground state energy and the ground state point proton root-mean-squared radius of ^{6}Li** training the network with NCSM results, obtained in accessible harmonic oscillator (HO) basis spaces. They showed that an ANN is able to predict correctly extrapolations of the NCSM results to very large model spaces of size $N_{max} \sim 100$.

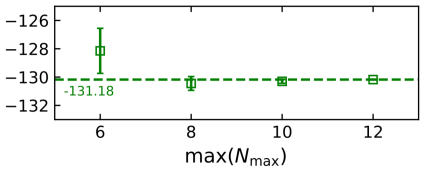


- Similarly, *Jiang et al., PRC 100, 054326 (2019)* have also employed an ANN to extrapolate the **ground state energy and radii of ^{4}He, ^{6}Li & ^{16}O** computed with the NCSM and the coupled-cluster (CC) methods.

^{4}He with NCSM+NNLO$_{opt}$



^{16}O with CC+NNLO$_{opt}$

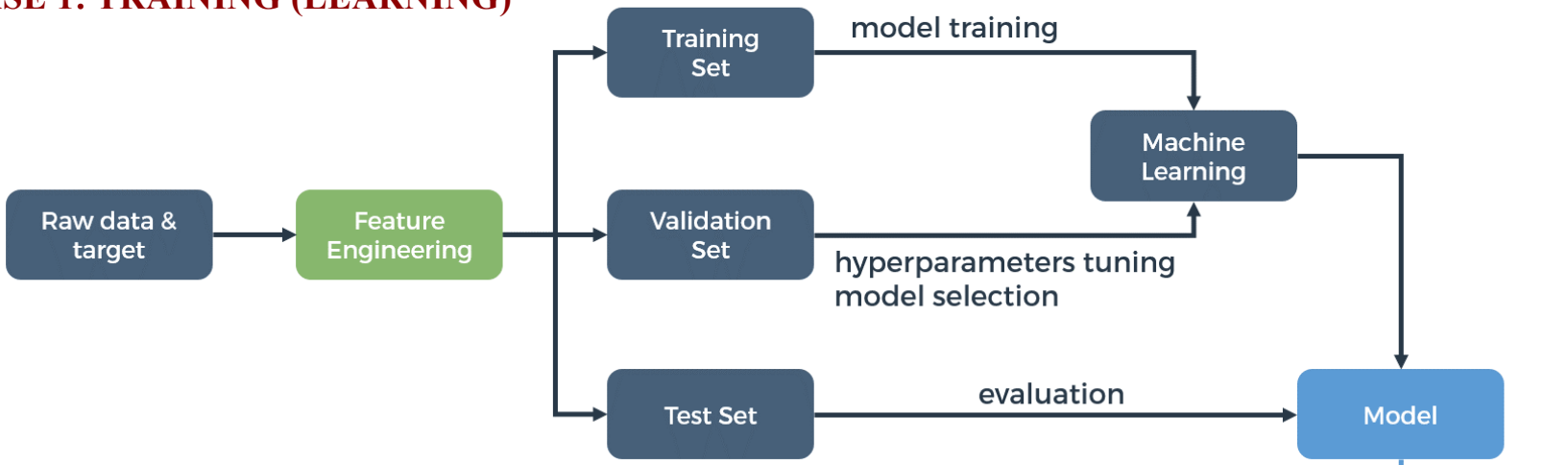


Here we follow the work of these authors, to extrapolate at large model spaces the results of *ab-initio* hypernuclear NCSM calculations for the Λ separation energy B_Λ of the lightest hypernuclei

Machine Learning Process: General Scheme

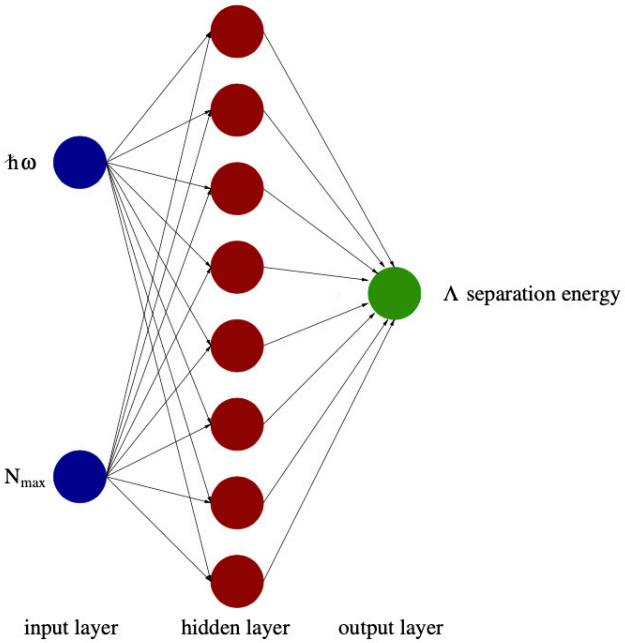
The task of **making a machine to learn** is made of **2 phases**

PHASE 1: TRAINING (LEARNING)



PHASE 2: PREDICTION

Architecture of the ANN



Architecture of our ANN. The input data are the HO spacings ħω and the maximum number of basis states N_{max} employed in hypernuclear NCSM calculations, whereas the output is the Λ separation energy

Numerical implementation with Python libraries Scikit-learn & Keras using a TensorFlow backend

- ANNs consist of a series of layers (input, hidden & output) each one contained a certain number of interconnected neurons
- In a feed-forward ANN, neurons do not form a cycle and the data propagates sequentially from the input to the output layer through all the hidden layers
- At each one of the N_k neurons i of a given layer k, the set of input data $\{x_j^{(k-1)}\}$ from the N_{k-1} neurons j of the layer k is transformed into

$$x_i^{(k)} = f\left(\sum_{j=1}^{N_{k-1}} W_{ij}^{(k)} x_j^{(k-1)} + b_i^{(k)}\right)$$

  - $f(z)$: activation function, introduces non-linearities on the neural network that enable it to capture complex non-linear relationships in the dataset. In this work we use a sigmoid activation function $f(z)=(e^z+1)^{-1}$
  - $W_{ij}^{(k)}, b_i^{(k)}$: fitting parameters of the ANN. Are the *weights* of the connections between the neurons of the two adjacent layers *k-1* & *k*, and the activation offset (*bias*) of each neuron of the layer *k*. The total number of fitting parameters n_p is

$$n_p = \sum_{k=1}^{L-1} (N_k + 1) N_{k+1}$$

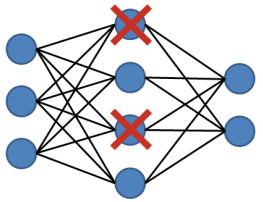
Neural Network Hyperparameters

Hyperparameters are the **variables which determine the network structure** (*e.g.,* number of hidden layers and neurons, type of regularization techique, initializatial values of weights & biases, type of activation function …) and the **variables which determine how the network is trained** (*e.g.,* learning rate, number of epochs (iterations), bach size, …). They are **set before the training** of the network

- **Hyperparameters related to the network structure**

  - **Number of hidden layers & neurons :** Many hidden layers and neurons layer can increase accuracy. Smaller number of hidden layers and neurons may cause **underfitting**

  - **Dropout:** is a regularization technique to avoid **overfitting** (seen later). It consist on dropping randomly neurons from the neural network during training in each iteration. The **number of dropped neurons** is another hyperparameter



  - **Initial values of weights & biases:** different weight initialization schemes can be used to start the training

  - **Type of activation function:** different types of activation function (seen later) can be used to introduce non-linearities

- **Hyperparameters related to the training of the network**

  - **Learning rate:** defines how quickly a network updates its parameters
  - **Number of epochs or iterations:** is the number of times the whole training data is shown to the network while training
  - **Batch size:** is the number of samples given to the network after which parameter update happens

The Learning Process of an ANN

The **learning (or training) process of an ANN** involves the **minimization of a cost** (also called **loss** or **error**) **function** (which compares the desired ouput (target) and the predicted one by the ANN) in order to **obtain the optimal set of fitting parameters** (**weights** and **biases**) of the network. The minimization is usually done by using **algorithms** such as the so-called **gradient descent**

Choice of a Cost Function

In general, the **choice of the cost function depends on the type of problem** one is solving with a neural network. In supervised learning, there are **two main types** of cost functions:

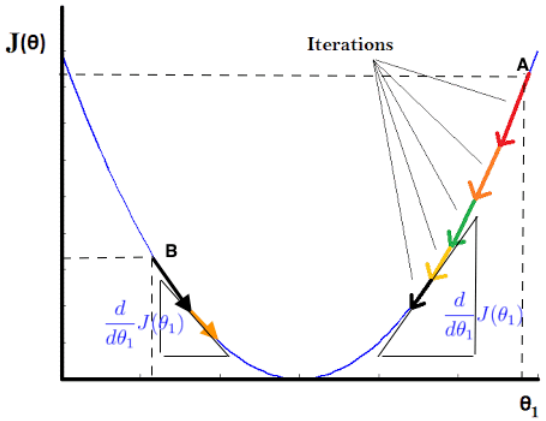
- **Regression Cost Functions** — used when solving a regression problem. Two examples of them are the *Mean Squared Error*, the *Mean Absolute Error*
- **Classification Cost Functions** — used when solving a classification problema. Among these type we can distinguish the *Binary Cross-Entropy* and the *Categorical Cross-Entropy*

Batch Gradient Descent

Batch Gradient Descent or simply **Gradient Descent** is an **iterative optimization algorithm** for finding a **local minimum** of a **differenciable function**

Idea: Take repeated steps in the opposite direction of the gradient since the **gradient of a multi-variable function $J(\vec{\theta})$ defines the direction of its maximum increase.** One starts with a guess $\vec{\theta}_o$ and considers the sequence $\vec{\theta}_1, \vec{\theta}_2, \vec{\theta}_3, \cdots$ according to

$$\vec{\theta}_{n+1} = \vec{\theta}_n - \eta \vec{\nabla} J(\vec{\theta}_n), \text{ with } \eta > 0$$



With this idea in mind the **weights ω_{jk}^l & biases b_j^l of the network are updated at each iteration** according to:

$$\omega_{jk}^l \to \omega_{jk}^l - \eta \frac{\partial C}{\partial \omega_{jk}^l}, \qquad b_j^l \to b_j^l - \eta \frac{\partial C}{\partial b_j^l}$$

where η is the so-called **learning rate,** one of the **hyperparameters** of the newtwork, and it scales the magnitude of the weighs and biases updates

Batch Gradient Descent Algorithm

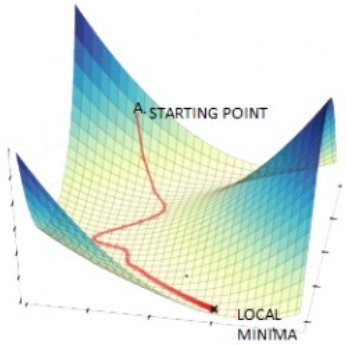
The Batch Gradient Descent Algorithm is quite simple. For **each epoch** (or **iteration**) of the training do the following steps:

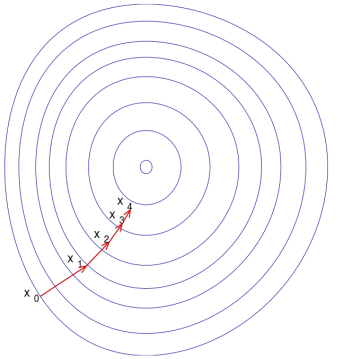
1. Feed the network with the **entire training input dataset** $\vec{x}$

2. Calculate the cost function and update the weigths & biases

$$\omega_{jk}^{l} \rightarrow \omega_{jk}^{l} - \eta \frac{\partial C}{\partial \omega_{jk}^{l}}, \qquad b_{j}^{l} \rightarrow b_{j}^{l} - \eta \frac{\partial C}{\partial b_{j}^{l}}$$

3. Repeat steps 1 – 2 until the convergence the cost function is substancially reduced

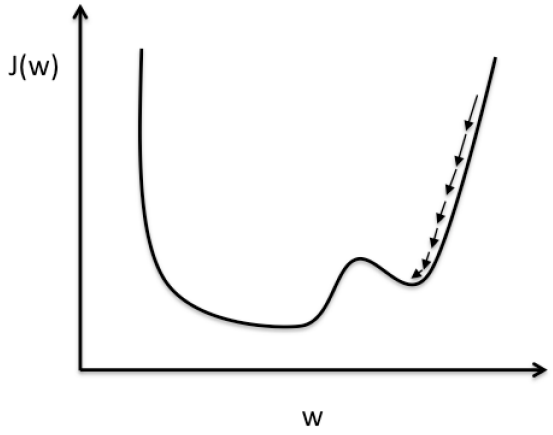




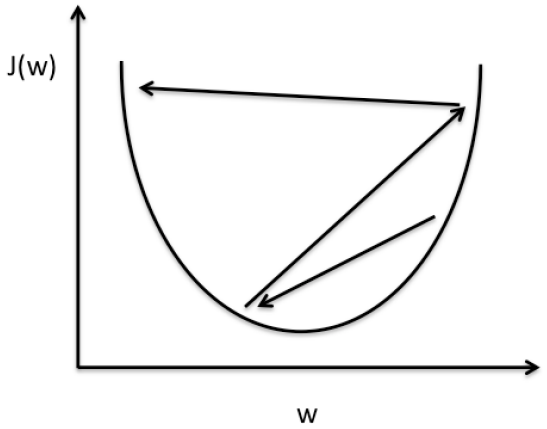
A comment in the learning rate η

A proper value of η plays a crucial role in gradient descent

- Choose η **too small** and the algorithm **will converge very slowly** or **get stuck in the local minima**
- Choose η **too big** and the algorithm **will never converge** either because it **will oscillate between around the minima** or it **will diverge by overshooting the range**

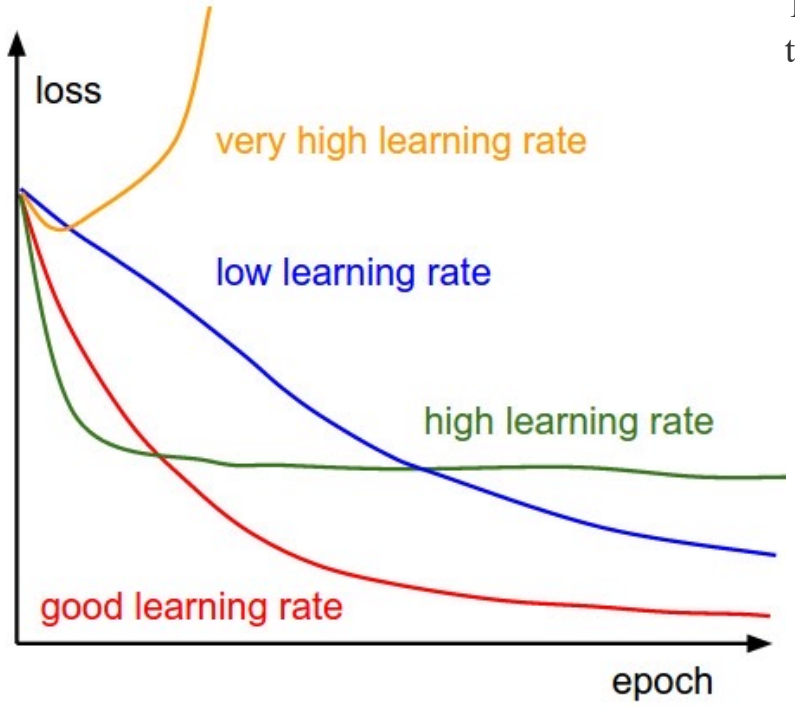


Small learning rate: Many iterations until convergence and trapping in local minima.



Large learning rate: Overshooting.

Effect of the learning rate η in the convergence of the Gradient Descent Algorithm

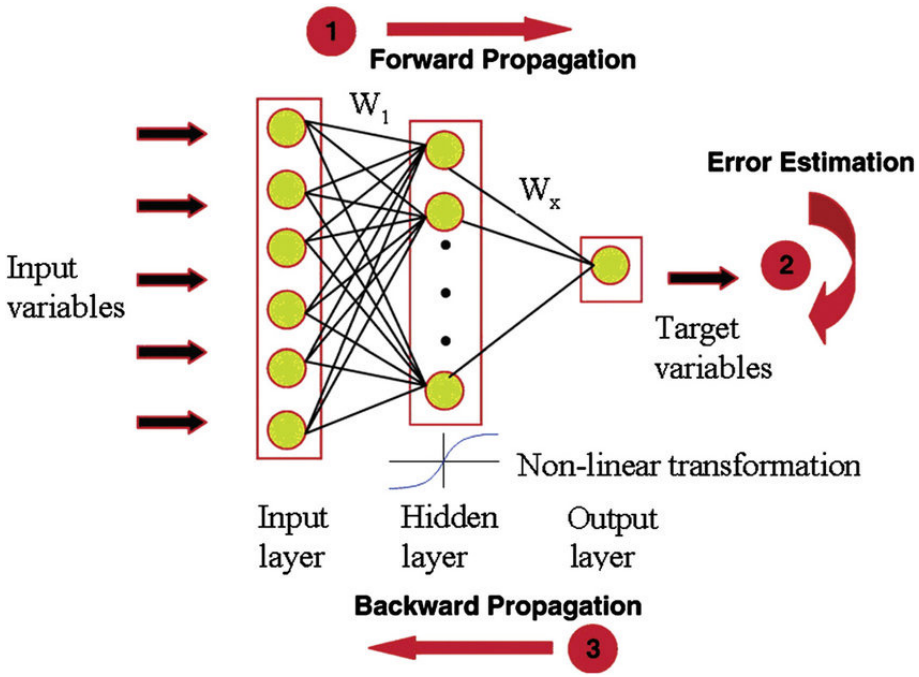


This figure tries to summarize the effect of η on the **convergence** of the gradient descent algorithm

- The **yellow** curve shows the **divergence** of the algorithm when the learning rate is really high wherein the learning steps overshoot.
- The **green** curve shows the case where learning rate is not as large as the previous case but is high enough that the steps keep **oscillating** at a point which is not the minima.
- The **red** curve would be the **optimum curve** for the cost drop as it drops steeply initially and then saturates very close to the optimum value.
- The **blue** curve is the least value of η and **converges very slowly** as the steps taken by the algorithm during update steps are very small.

The Backpropagation Algorithm: General Scheme

Backpropagation is a method used to **calculate efficiently the gradient** of the **cost function** and **adjust the connection weights & the biases** to reduce the error during the learning process

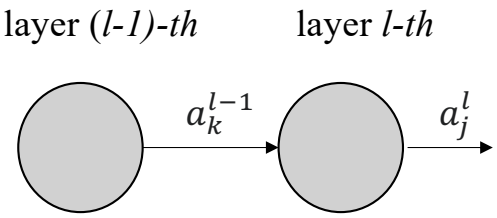


1. In a feed-forward ANN information **propagates sequentially** from through all the layers form the input to the output ones
2. The error (also known as **cost or loss function**) is evaluated
3. The error is **propagated backwards** to determine the new values of the fitting parameters at each layer & neuron
4. Steps 1 to 3 are repeated iteratively until a small error is reached

The Backpropagation Algorithm: Main Ingredients

Before presenting the backpropagation algorithm let us first recall the notation and present the main ingredients:

- ω_{jk}^{l}: **weight** between the neuron *k-th* in the layer *(l-1)-th* and the neuron *j-th* in the layer *l-th*
- b_j^l: **bias** of neuron *j-th* in the layer *l-th*
- $z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$: **weighted input** of neuron *j-th* in the layer *l-th*
- $a_j^l = f(z_j^l)$: **activation** (output) of neuron *j-th* in the layer *l-th* (note that $\boldsymbol{a_j^L = \hat{y}_j}$)



A little change Δz_j^l in the weighted input of neuron *j-th* in the layer *l-th* will propagate through later layers in the network, finally causing the overall cost to change by an amount $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$, where $\frac{\partial C}{\partial z_j^l}$ can be interpreted as a measurement the of the **error of neuron *j-th* in the layer *l-th***

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} f'(z_j^l) \qquad \text{(BP1)}$$

The Backpropagation Algorithm: Main Ingredients

Since the weighted inputs in the layer *(l+1)−th* (z_k^{l+1}) depend on the weighted inputs of the previous layer *l-th* (z_j^l), we can write

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Now, from $z_k^{l+1} = \sum_j \omega_{kj}^{l+1} a_j^l + b_k^{l+1}$ we have

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \omega_{kj}^{l+1} f'(z_j^l) \longrightarrow \boxed{\delta_j^l = \sum_k \delta_k^{l+1} \omega_{kj}^{l+1} f'(z_j^l)} \quad \text{(BP2)}$$

And therefore, the **gradient of the cost function** is simply given by

$$\boxed{\frac{\partial C}{\partial \omega_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial \omega_{jk}^l} = \delta_j^l a_k^{l-1}} \quad \text{(BP3)} \qquad \boxed{\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l} \quad \text{(BP4)}$$

The Backpropagation Algorithm: Summary

The backpropagation equations (BP1)-(BP4) provide us with a **fast way of computing the gradient of the cost function and adjusting the weights & biases**. Let's explicitly write it in the form of an algorithm

1. **Input** x: set the corresponding activation $a_j^1 = x_j$ for each neuron *j-th* of the input layer
2. **Feedforward**: for each layer $l = 2, 3, \cdots, L$ compute $z_j^l = \sum_k \omega_{jk}^l \, a_k^{l-1} + b_j^l$ and $a_j^l = f(z_j^l)$
3. **Output error** δ_j^L: compute the error of each neutron of the last layer L, $\delta_j^L = \frac{\partial C}{\partial a_j^L} f'(z_j^L) = \frac{\partial C}{\partial \hat{y}_j} f'(z_j^L)$
4. **Backpropagate the error**: for each layer $l = L-1, L-2, \cdots, 2$ compute $\delta_j^l = \sum_k \delta_k^{l+1} \omega_{kj}^{l+1} f'(z_j^l)$
5. **Gradient of the cost function**: $\frac{\partial C}{\partial \omega_{jk}^l} = \delta_j^l a_k^{l-1}$, $\frac{\partial C}{\partial b_j^l} = \delta_j^l$
6. **Update the weights & biases**: $\omega_{jk}^l \to \omega_{jk}^l - \eta \frac{\partial C}{\partial \omega_{jk}^l}$, $b_j^l \to b_j^l - \frac{\partial C}{\partial b_j^l}$
7. **Repeat steps 2 to 6 till convergence is achieved**

In which sense is Backpropagation a fast algorithm ?

To answer this question, suppose we want to compute the gradient of the cost function C by simply using the approximation

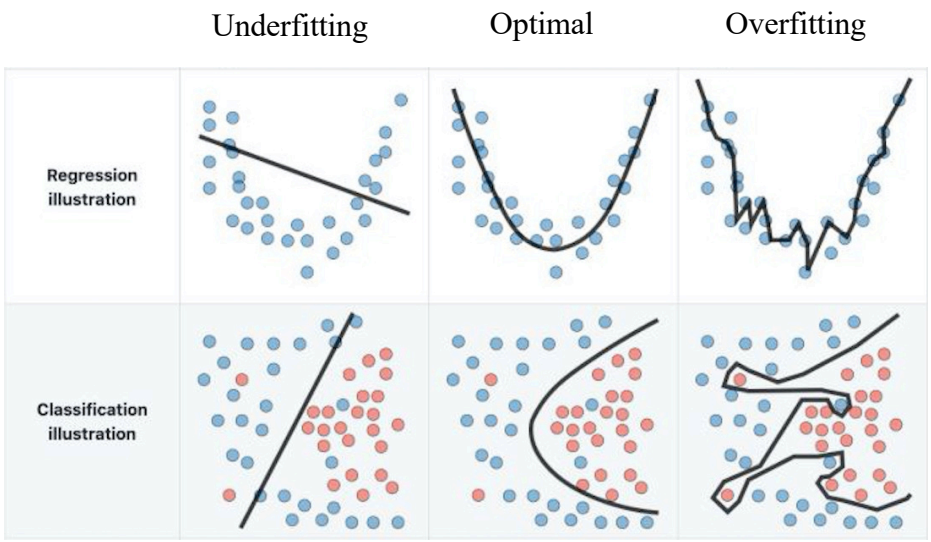
$$\frac{\partial C}{\partial w_{jk}^l} \approx \frac{C\left(\omega_{jk}^l - \epsilon e_{jk}^l, b_j^l\right) - C\left(\omega_{jk}^l, b_j^l\right)}{\epsilon} \quad , \quad \frac{\partial C}{\partial b_j^l} \approx \frac{C\left(\omega_{jk}^l, b_j^l - \epsilon e_j^l\right) - C\left(\omega_{jk}^l, b_j^l\right)}{\epsilon}$$

where $\epsilon > 0$ is a small positive number and e_{jk}^l $\left(e_j^l\right)$ is a unit vector in the direction of ω_{jk}^l $\left(b_j^l\right)$

This looks very promissing, we only have to compute $C\left(\omega_{jk}^l, b_j^l\right)$, $C\left(\omega_{jk}^l - \epsilon e_{jk}^l, b_j^l\right)$ and $C\left(\omega_{jk}^l, b_j^l - \epsilon e_j^l\right)$ for each distincts weight ω_{jk}^l and bias b_j^l. **However,** this is **extremelly expensive computationally speaking,** specially for neural networks with a exteme large number (millions) of weights and biases

What is clever about the backpropagation algorithm is that it **enables us to compute simultaneously all the partial derivatives $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ using just one forward pass through the network followed by one backward pass through the network**, *i.e.,* the computational cost of the forward and backward passes is the same. The **numerical cost of backpropagation** is **roughly the same as** making **just two forward passes**

Overfitting of an ANN

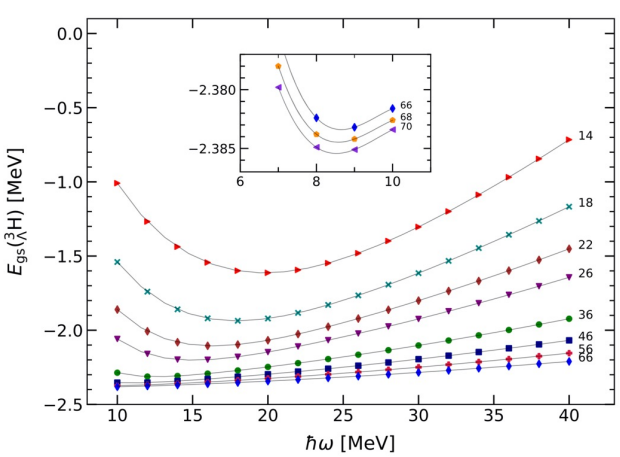


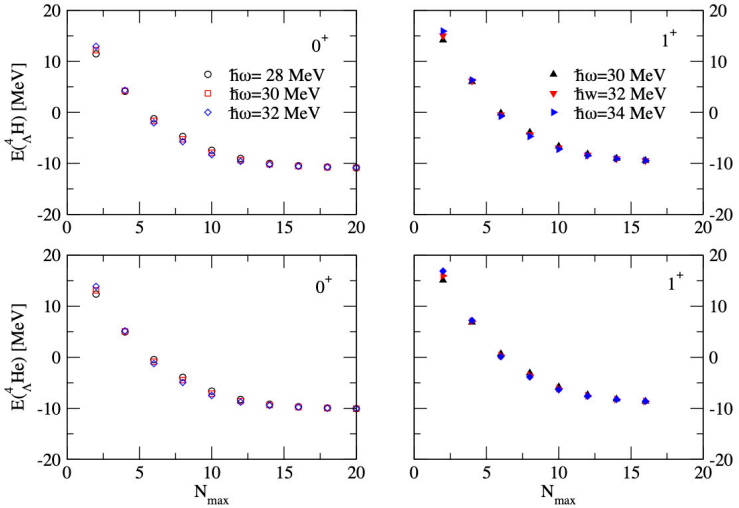
- A major issue in the development of an ANN is overfitting (also known as overtraining), which basically means that the network, due to its high flexibility to approximate complex non-linear functions, tries to fit the data entirely and ends up memorizing all the data patterns.
- Due to overfitting the predictability of the network on testing data becomes questionable

- Strategies to avoid overfitting include among others:
  - early stopping of the training: stops the training process once the model performance stops improving on the validation dataset
  - dropout: reduce overfitting by dropping randomly neurons from the neural network during training in each iteration
- In addition to these which can be used together, overfitting can be reduced by:
  - enlarging the input dataset (specially in those case where the input dataset is not large enough)
  - adding noise to the input dataset making the network less able to memorize data patterns since they change randomly during the training

Input Dataset

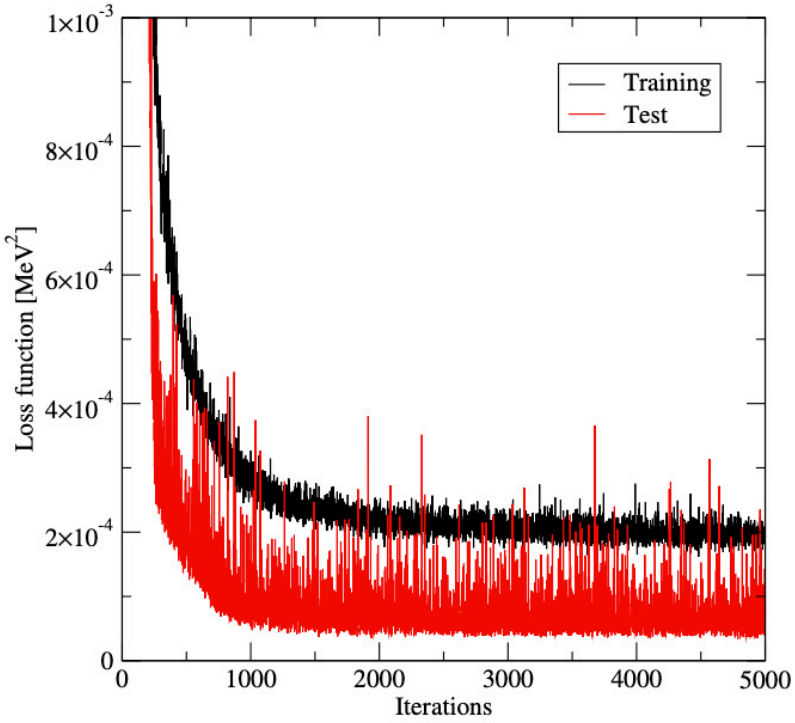
- We employ as input dataset the hypernuclear NCSM results of Gazda *et al.* (PRC 97 (2018) 064315, Few-Body Syst. 62 (2021) 94) for the Λ separation energy of $^{3}_{\Lambda}$H, $^{4}_{\Lambda}$H & $^{4}_{\Lambda}$He obtained with chiral NN & NNN interactions at N^{3}LO and N^{2}LO, respectively both with a regulator cutoff of 500 MeV, and YN potentials at LO with a cutoff of 600 MeV





- Due to the small size of the original input dataset to avoid overfitting we have:
  - enlarged it by performing a cubic interpolation in the HO spacing $\hbar\omega$ at each given value of N_{max}
  - introduced a Gaussian noise in the enlarged input dataset during the training of the network
- We use the 80% (10 % of it used for validation) of the enlarged input dataset to train the network and leave the 20% of it for testing

Performance of the ANN



Loss function of the training & test datasets as a function of the number of iterations in the calculation of the Λ separation energy of the ground state of $^3_\Lambda$H

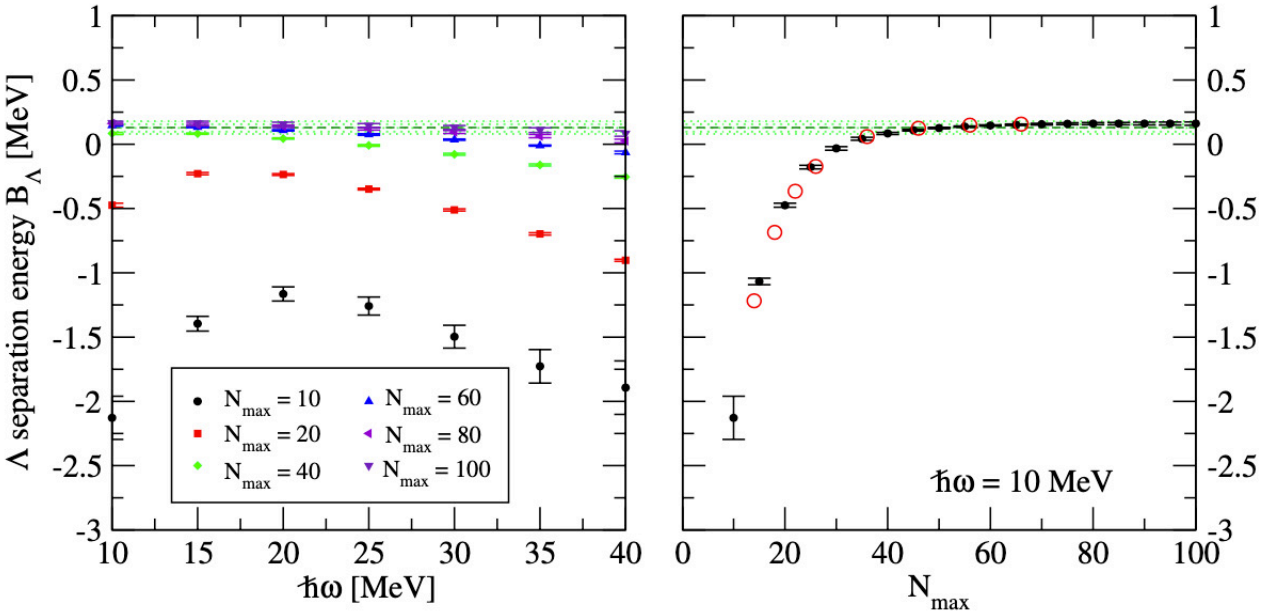
- The learning process of an ANN involves the minimization of a loss function in order to obtain the optimal set of parameters $(\boldsymbol{W}, \boldsymbol{a}) \equiv \left\{\boldsymbol{W}_{\boldsymbol{ij}}^{(\boldsymbol{k})}, \boldsymbol{a}_{\boldsymbol{i}}^{(\boldsymbol{k})}\right\}$. In the case of a regression-type problem, as in our case, a common choice is the mean squared error (MSE)

$$\mathcal{L}(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i(\boldsymbol{W}, \boldsymbol{b}) - y_i)^2$$

  - N: number of data points used in the minimization procedure
  - $\hat{y}_i(\boldsymbol{W}, \boldsymbol{b}) \equiv \boldsymbol{x}_{\boldsymbol{i}}^{(\boldsymbol{L})}$: prediction of the ANN
  - $\boldsymbol{y_i}$: actual output of the input data

- Very fast decrease during the first 500 iterations becoming (on average) essentially constant at about 1000 iterations and above it.
- The loss function of the test dataset is smaller that that of the training one, indicating that overfitting has been substantially reduced.
- Similar good performance for $^4_\Lambda$H and $^4_\Lambda$He

Λ separation energy of the ground state of $^{3}_{\Lambda}$H



Open circles in the right panel show the NCSM results used for the training of the ANN

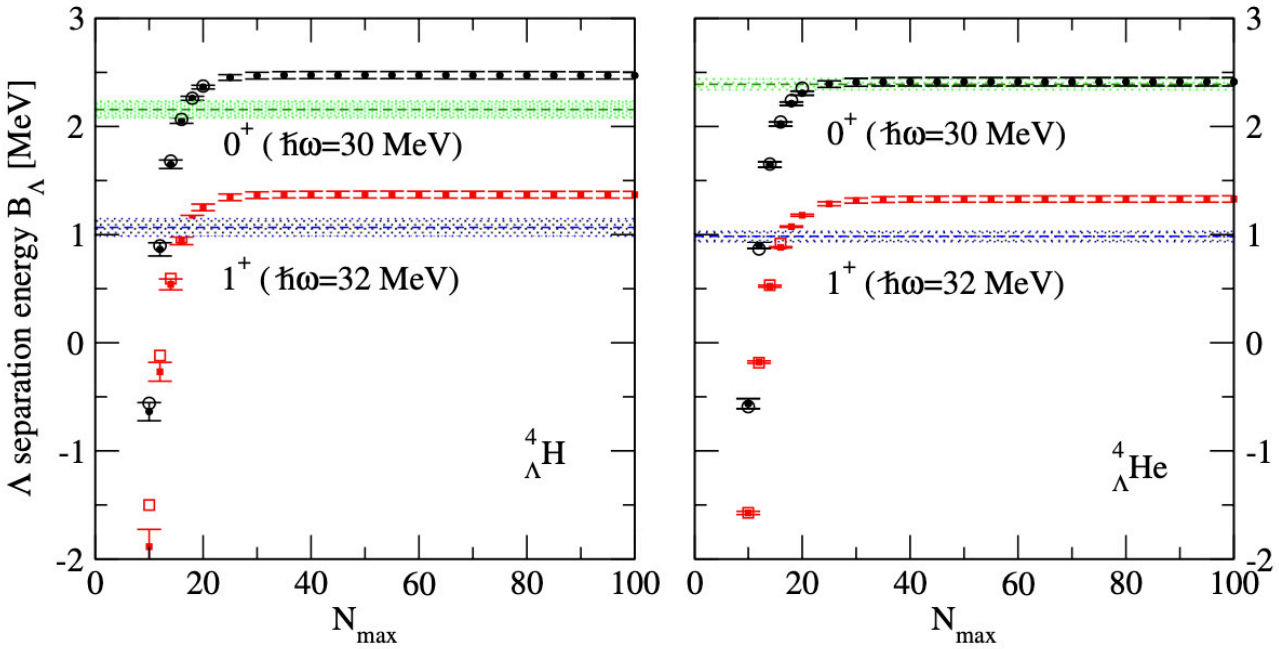
- Slow convergence due to the extremely weak binding energy of $^{3}_{\Lambda}$H
- Considerably reduction of the B_Λ dependence with ħω with the increase of N_{max}
- Good extrapolation to the experimental result for large values of the model space size N_{max}

ANN prediction for $N_{max} = 100$

$$B_\Lambda\left(^{3}_{\Lambda}H\right) = 0.16 \pm 0.01 \text{ MeV}$$

N.B.: A typical run of an ANN starts with random values of the weights & biases of the network. Therefore, different runs can lead to slightly different results. Because of this we have performed 25 independent runs of the ANN and taken the average and the standard deviation of all these runs as the prediction of the network & their corresponding error

Λ separation energy of the 0^+ & 1^+ states of ${}^4_\Lambda$H & ${}^4_\Lambda$He



ANN prediction for $N_{max} = 100$

$B_\Lambda\left({}^3_\Lambda H(0^+)\right) = 2.47 \pm 0.03$ MeV $B_\Lambda\left({}^3_\Lambda He(0^+)\right) = 2.41 \pm 0.04$ MeV

$B_\Lambda\left({}^3_\Lambda H(1^+)\right) = 1.37 \pm 0.03$ MeV $B_\Lambda\left({}^3_\Lambda He(1^+)\right) = 1.33 \pm 0.03$ MeV

- Convergence faster than in the ${}^3_\Lambda$H case. Good convergence already for $N_{max} > 25$
- Well extrapolation of the ANN prediction for the 0^+ state of ${}^4_\Lambda$He to the experimental value
- ANN prediction for the 0^+ & 1^+ states of ${}^4_\Lambda$H & 1^+ of ${}^4_\Lambda$He off of the experiment by about 0.3 MeV.
- Charge symmetry breaking (CSB) in these two A=4 mirror hypernuclei not explained because CSB effects are not included in the NCSM calculations used to train the ANN. Therefore, the ANN cannot account for them

This work in few words

- We employ a feed-forward ANN to extrapolate at large model spaces the results of *ab-initio* hypernuclear NCSM calculations for the Λ separation energy B_Λ of the lightest hypernuclei, obtained in accessible HO basis spaces using chiral NN, NNN & YN interactions
- The overfitting problem is avoided by enlarging the size of the input dataset & by introducing a Gaussian noise during the training process of the neural network
- We find that a network with a single hidden layer of eight neurons is enough to extrapolate correctly the value of B_Λ to model spaces of size N_{max}=100

Hypernucleus	ANN Prediction	Experimental Vaue
${}^{3}_{\Lambda}\mathrm{H}$	0.16 ± 0.01	0.13 ± 0.05
${}^{4}_{\Lambda}\mathrm{H}(0^+)$	2.47 ± 0.03	2.157 ± 0.077
${}^{4}_{\Lambda}\mathrm{H}(1^+)$	1.37 ± 0.03	1.067 ± 0.08
${}^{4}_{\Lambda}\mathrm{He}(0^+)$	2.41 ± 0.04	2.39 ± 0.05
${}^{4}_{\Lambda}\mathrm{He}(1^+)$	1.33 ± 0.03	0.984 ± 0.05

✧ You for your time & attention

✧ The organizers for their invitation & support

✧ Specially Daniel Gazda for providing me with the NCSM results used to train the ANN

"This project has received funding from the Helmholtz Institute Mainz and the European Union's Horizon 2020 research and innovation programme under grant agreement No 824093"