Machine Learning Light Hypernuclei

Isaac Vidaña, INFN Catania



Departamento de Física Teórica, Universidad Complutense
March 30th 2023, Madrid

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

- Our results are in excellent agreement with those obtained using other extrapolation schemes of hypernuclear NCSM calculations, showing this that ANN is a reliable extrapolation method

Based on:



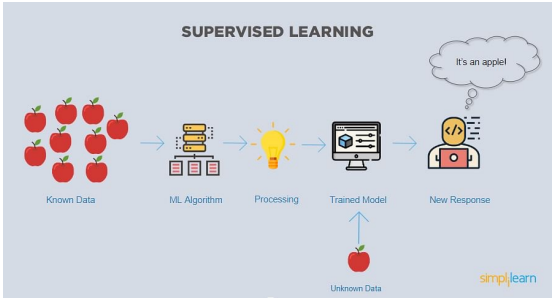
I.V. NPA 1032, 122625 (2023)

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

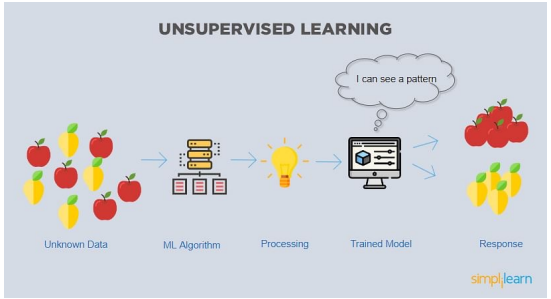
- **Supervised Learning**

  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



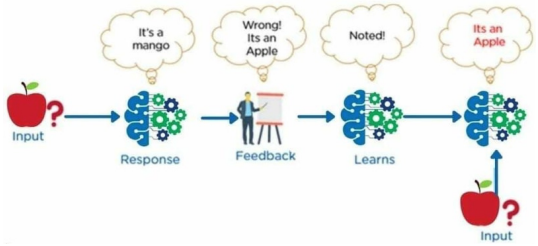
- **Unsupervised Learning**

  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**

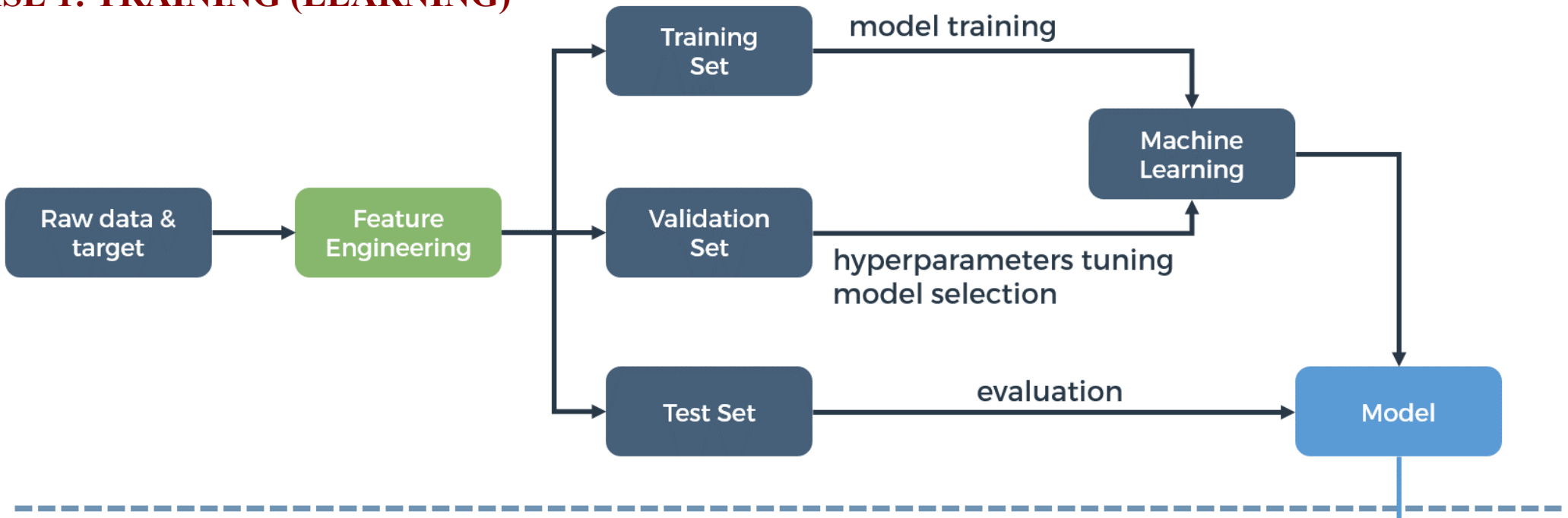
  Given a **framework of rules and goals**, an **agent** (algorithm) **learns in an interactive environment** by **trial and error** using **feedback from its own actions and experiences** and it gets **rewarded** or **punished** depending on which strategy it uses. Each **reward** reinforces the current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



Machine Learning Process: General Scheme

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

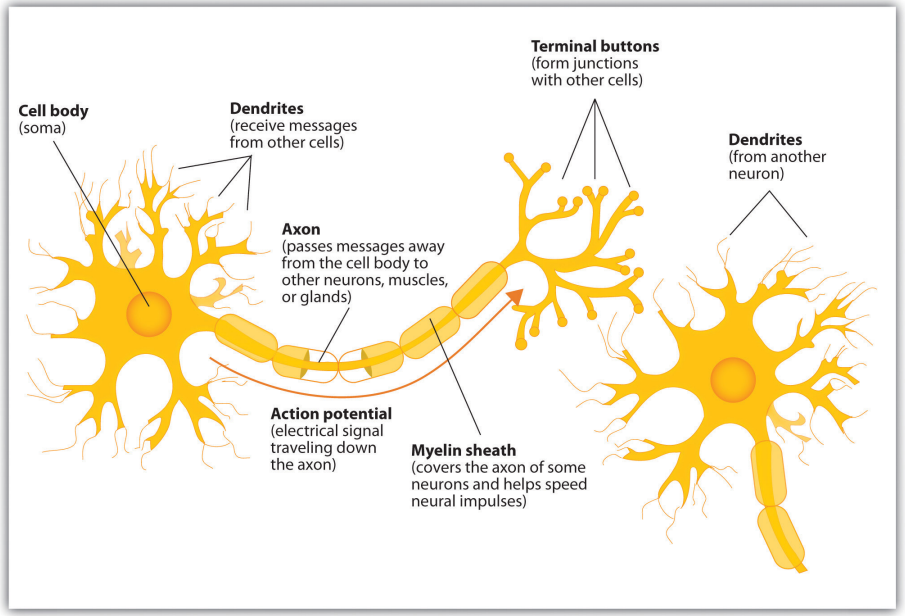
PHASE 1: TRAINING (LEARNING)



PHASE 2: PREDICTION



Artificial Neurons – a brief glimpse into the early history of ML



Biological Neurons are **interconnected nerve cells** in the brain that are involved in processing and transmitting chemical and electrical signals. Their main parts are:

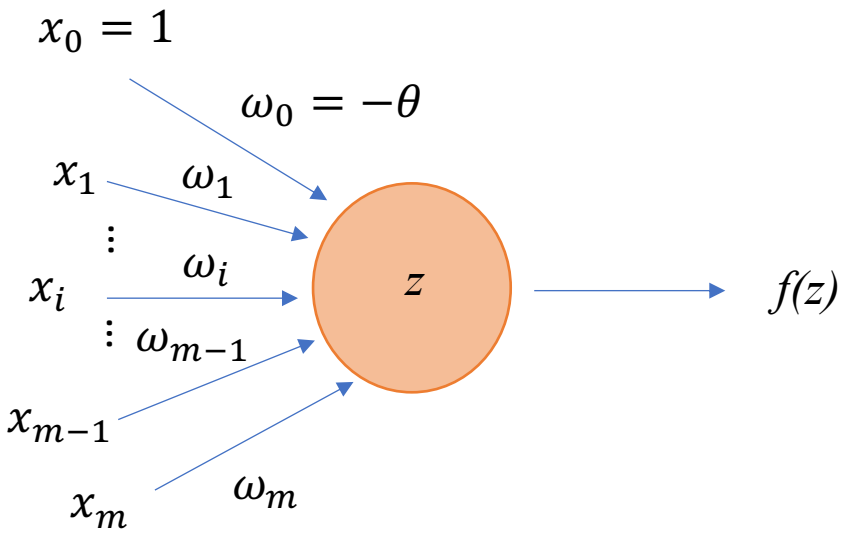
- **Dendrites** are branches that receive information from other neurons
- **Cell nucleus** or **Soma** processes the information received from dendrites
- **Axon** is a "cable" that is used by neurons to send information
- **Synapse** is the connection between an axon and other neuron dendrites.

- Trying to understant how the brain works, Warren McCulloch and Walter Pitts introduced the first artifical neutron, the so-called **McCulloch-Pitts (MCP) neuron** in 1943. They though it as a **simple logic gate with binary outputs**: multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon

- A few years later, in 1957, Frank Rosenblatt published the first concept of the **perceptron learning rule** based on the MCP neuron model. Rosenblatt **proposed an algorithm** that would **automatically learn** the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not

Formal Definition of an Artificial Neuron

- More formaly, we can put the idea behind **artifical neurons** into the context of a **binary classification task** where we refer to our two classes as ***1*** (positive class) and ***-1*** (negative class) for simplicity



- We can then define an **activation function** $f(z)$ that takes a linear combination $z = \sum_{i=1}^{m} \omega_i x_i$ (named **net input**) of certain **input values** $\vec{x} = (x_1, x_2, \cdots, x_m)$ where the coefficients ω_i are the so-called **weights** and it gives ***1*** or ***-1*** if the net input of a particular input value is larger or smaller than a certain defined **threshold** $\boldsymbol{\theta}$

$$f(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$

- For simplicity, we can bring the threshold θ to the left side of the equation and define a weight-zero as $\omega_0 = -\theta$ and $x_0 = 1$ so that we can write

$$z = \sum_{i=0}^{m} \omega_0 x_0 \quad \text{and} \quad f(z) = \begin{cases} 1 & if\ z \geq 0 \\ -1 & otherwise \end{cases}$$

Biological Neuron	Artificial Neuron
Cell nucleus (Soma)	Node
Dendrites	Input
Synapse	Weights or interconnections
Axon	Output

- In the Machine Learning literature the negative threshold or weight $\omega_0 = -\theta$ is usually called the ***bias unit*** and it is represented with the letter ***b***

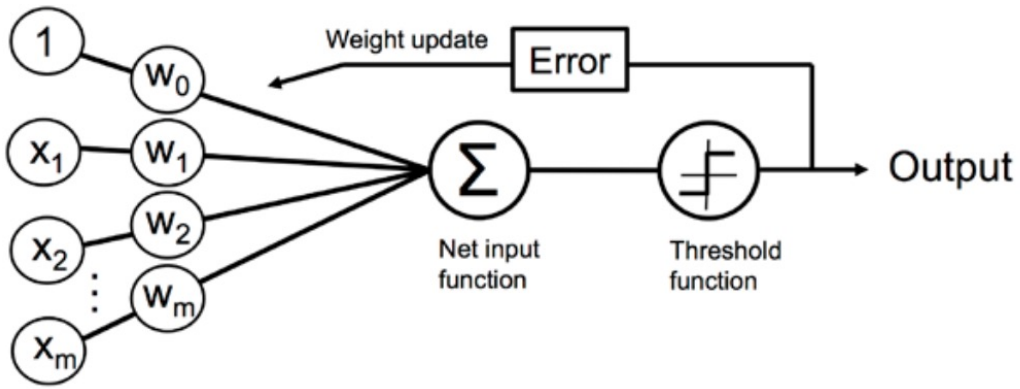
The Perceptron Learning Rule

The whole idea behind the ***MCP neuron*** and ***Rosenblatt's thresholded perceptron model*** is to use a reductionist approach to mimic how a single neuron in the brain works: ***it either fires or it doesn't***. Thus, Rosenblatt's initial perceptron rule is fairly simple, and the ***perceptron algorithm*** can be summarized by the following steps and illustrated by the diagram:

1. Initialize the weights to 0 or small random numbers

2. For each training example $\vec{x}$
   a. Compute the output value $\hat{y}$
   b. Update the weights according to the ***perceptron learning rule***

$$\omega_j \rightarrow \omega_j + \eta(y - \hat{y})x_j$$



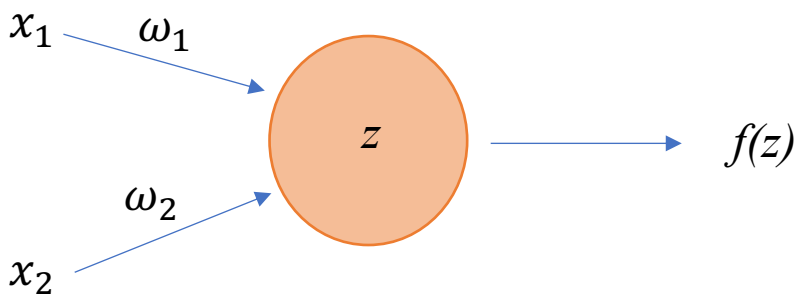
Here η is the ***learning rate*** (a constant between 0 and 1), y is the ***true class label*** and $\hat{y}$ the ***predicted class label***

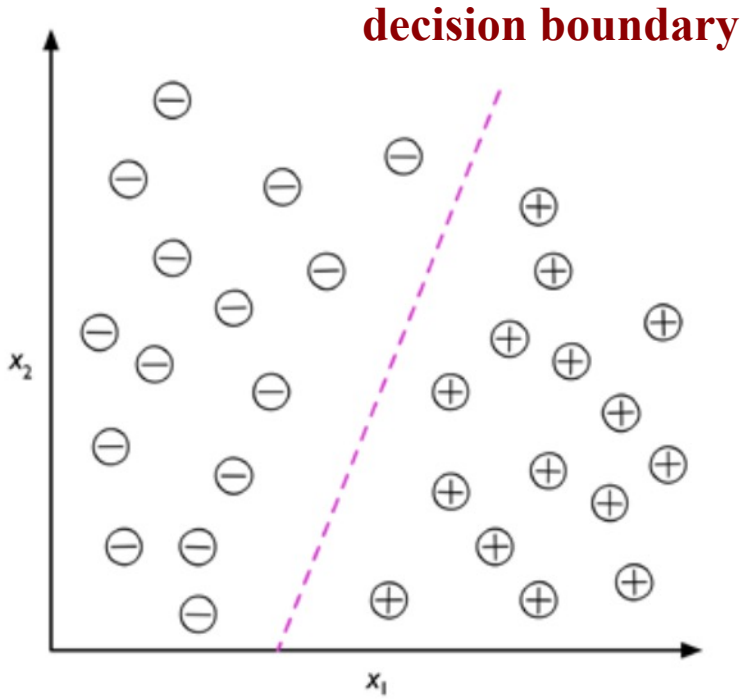
Example of a Binary Classification Task

- Consider 30 training examples: 15 of which are labeled as **negative class (-)** and the other 15 as **positive class (+)**
- The dataset is **two-dimensional**, which means that each training exampe has two values associated to it: x_1 and x_2
- Applying the Perceptron Learning Rule is possible to find the **decision boundary**

$$\omega_1 x_1 + \omega_2 x_2 + b = 0$$

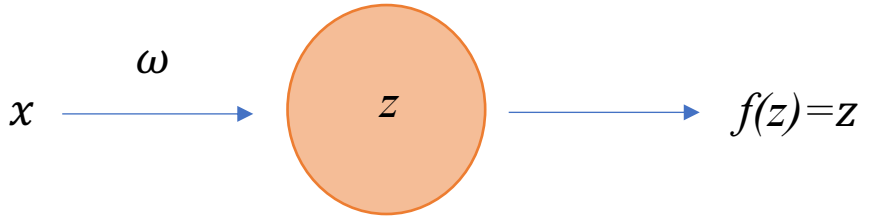
that can **separate those two classes** and will allow to **classify new data** into each of those two cathegories given its values x_1 and x_2





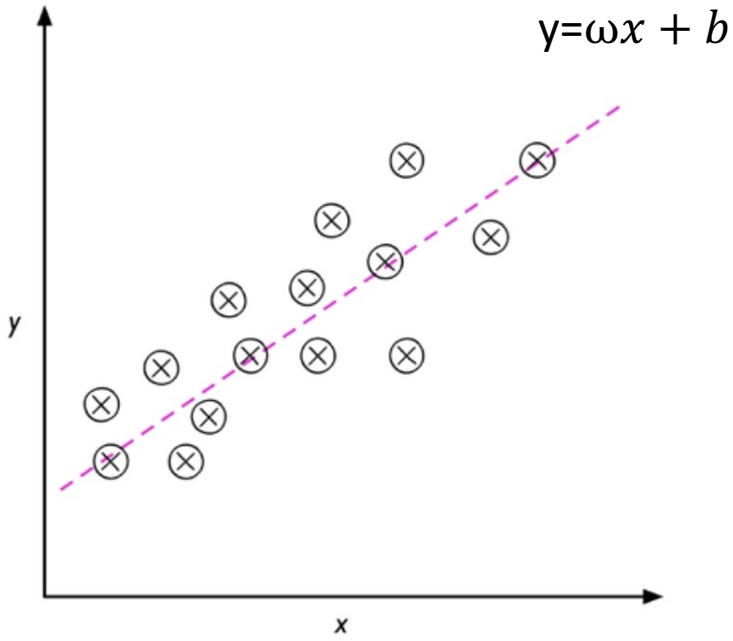
Preceptron & Linear Regression

- The perceptron can be used to **predict the outcome of (linear) continuous functions** if the **identity function** $f(z) = z$ is used as **activation function** intead of the unit step function



- Given a set of **features** variables x and their corresponding **target** values y that are linearly related

x_1	y_1
x_2	y_2
...	...
x_m	y_m

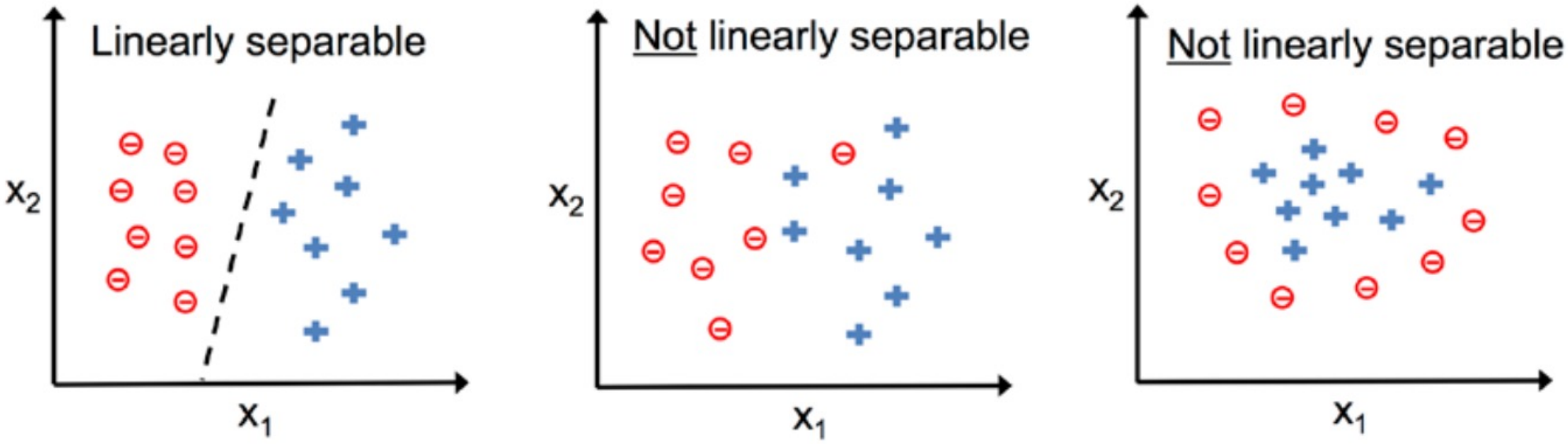


By applying the Perceptron Learning Rule is possible to find the **strainght line** that minimizes the distance – most commonly the average square distance – between the data points and the fitted line and make predictions for new data

$$y=\omega x + b$$

Convergence of the Perceptron Algorithm

It is important to note that the **convergence** of the perceptron is **only guaranteed if the two classes are linearly separable** and the **learning rate is sufficiently small**. If the two classes cannot be separated by a linear decision boundary the perceptron **would never stop updating the weights**. In this case, we can set a maximum number of passes over the training dataset (**epochs or iterations**) and/or a threshold for the number of tolerated misclassifications

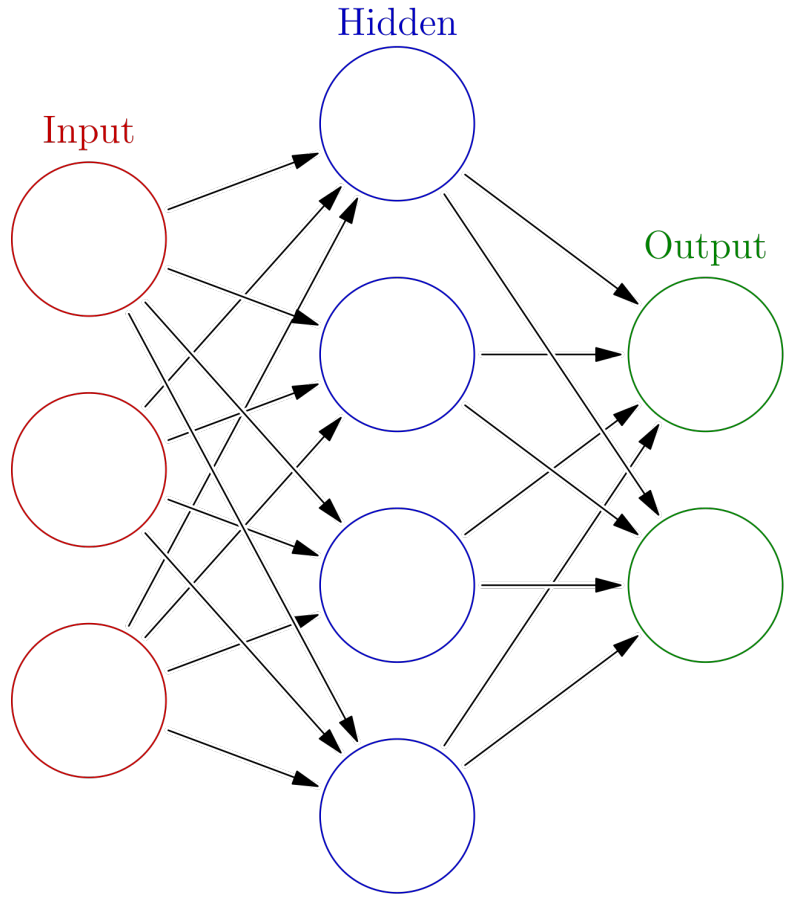


The solution, as we will see later, will imply the introduction of **non-linear activation functions**

General Architecture of an Artificial Neural Network

The architecture of an artificial neural network (ANN) consists of an **input layer**, one or more **hidden layers**, and an **output layer** of several **interconnected artificial neurons**

- **Input layer:** Made up of those neurons that introduce input patterns into the network. No processing occurs in these neurons
- **Hidden layers:** Formed by those neurons whose inputs come from previous layers and whose outputs pass to neurons of later layers
- **Output layer:** Made up by the neurons whose output values correspond to the outputs of the entire network

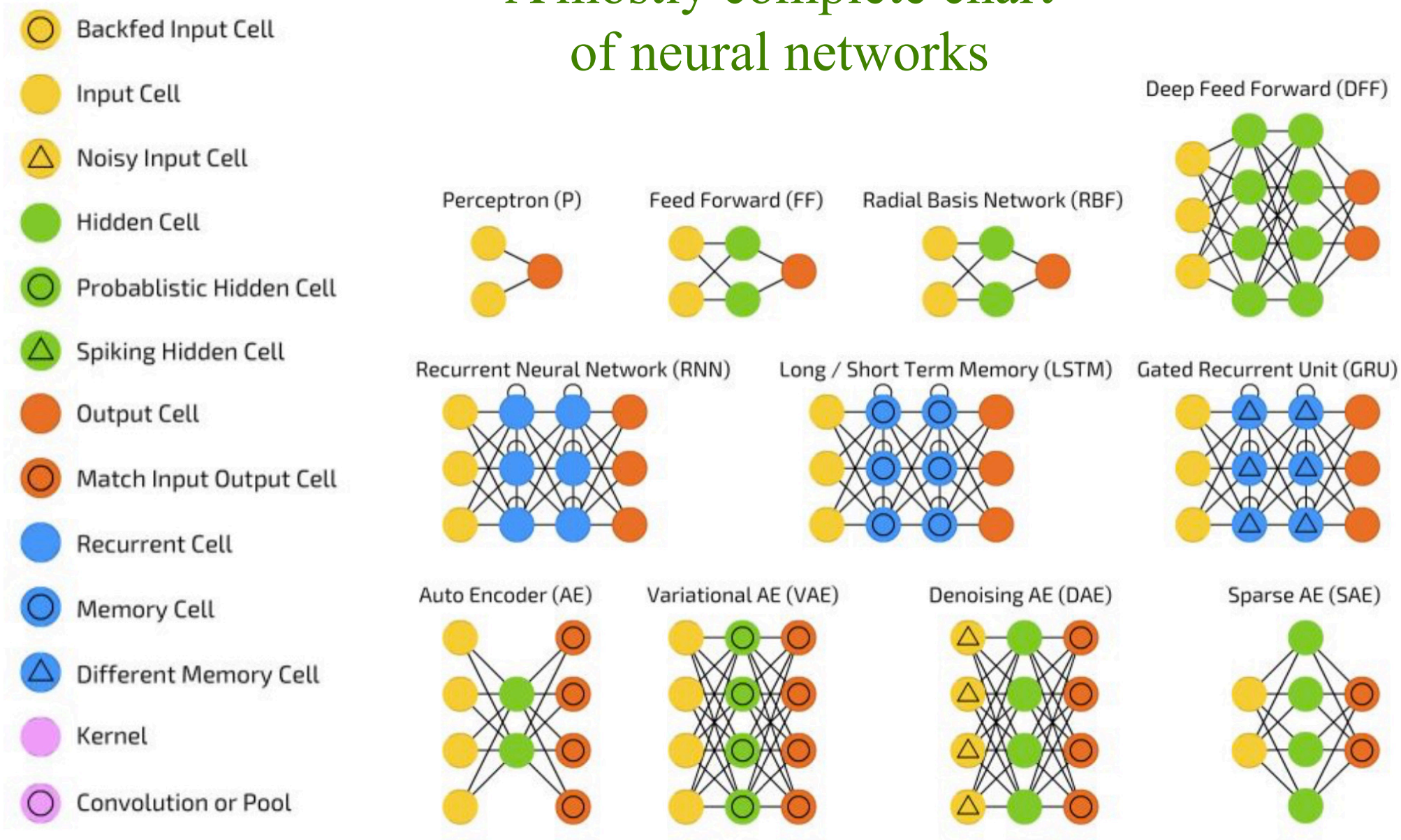


Terminology:

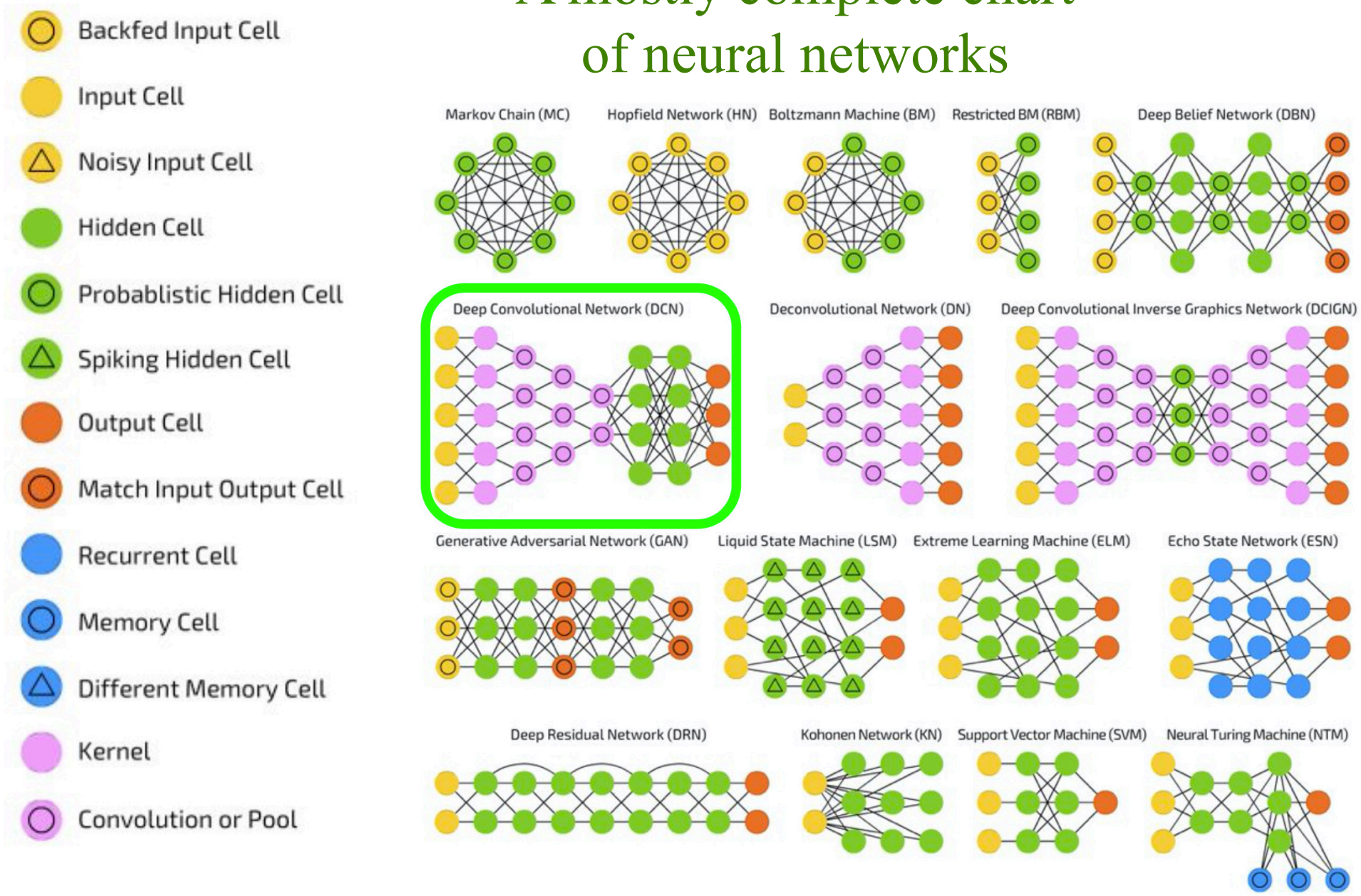
- ω_{jk}^{l}: **weight** between the neuron *k-th* in the layer *(l-1)-th* and the neuron *j-th* in the layer *l-th*
- b_j^l: **bias** of neuron *j-th* in the layer *l-th*
- $a_j^l = f(z_j^l)$: **activation** (output) of neuron *j-th* in the layer *l-th* (note that $a_j^L = \hat{y}_i$)
- $z_j^l = \sum_k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

Total number of fitting parameters: $$n_p = \sum_{k=1}^{L-1} (N_k + 1) N_k$$

A mostly complete chart of neural networks



A mostly complete chart of neural networks



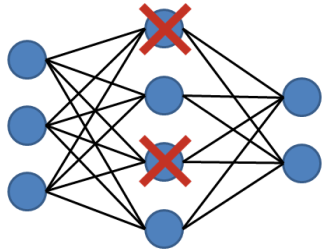
Neural Network Hyperparameters

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

Activation Functions

- As mentioned before one of the **limitations of the Perceptron Algorithm** is that it **only converges for linear** (classification or regression) **problems**. The same is true for any **ANN made of perceptrons** since, it will be simply **a linear combination of perceptrons**

- In order to pick up the **non-linearities of the input data** that enable ANN to capture **complex non-linear relationships in the dataset** and make new predictions, ANN should employ **non-linear activation functions**

- There **exist several possible choices** for the activation function **depending on the particular problem** one is trying to solve. The choice of the activation function has a large impact on the capability and performance of the neural network, and **different activation functions may be used in different parts of the model.** All hidden layers typically use the same activation function. The output layer will typically use a different activation function from the hidden layers and is dependent upon the type of prediction required by the model

- Activation functions are typically **differentiable** *(i.e.*, the first-order derivative can be calculated for a given input value). This is required given that neural networks are typically trained using the **backpropagation of error algorithm** that requires the derivative of prediction error in order to update the weights of the model

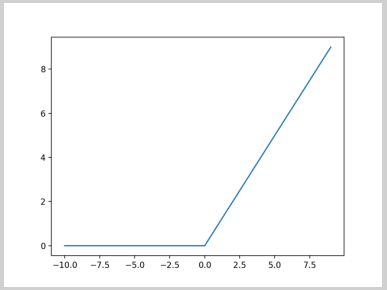
- There are many **different types of activation functions** used in neural networks, **although** perhaps only **a small number of functions used in practice for hidden and output layer**

Activation Functions

Hidden Layer Activation Functions

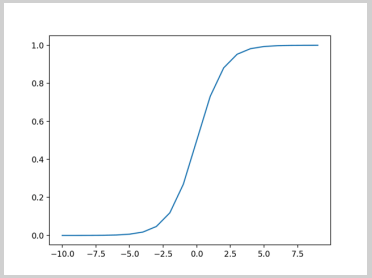
- **Rectified Linear Activation (ReLU) function**

$$f(z) = \max(0, z)$$



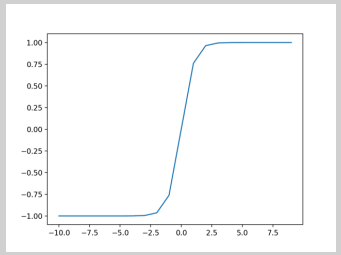
- **Sigmoid Activation function**

$$f(z) = \frac{1}{1 + e^{-z}}$$



- **Hyperbolic tangent Activation function**

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



Output Layer Activation Functions

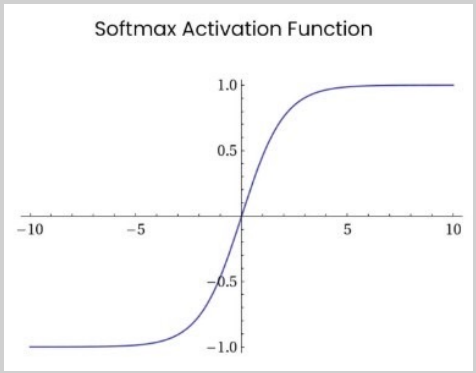
- **Linear or identity activation function**

$$f(z) = \mathrm{z}$$

- **Sigmoid Activation function**

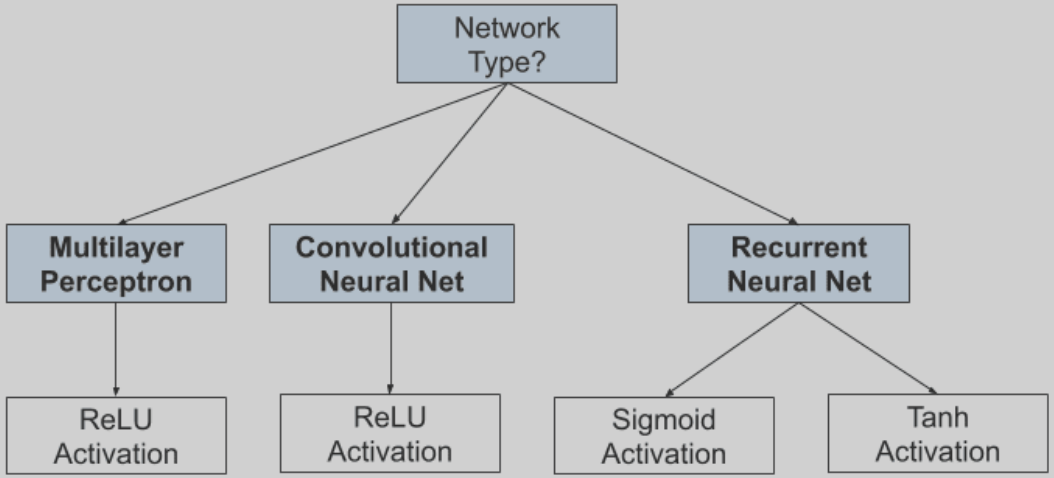
- **Softmax Activation function**

$$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

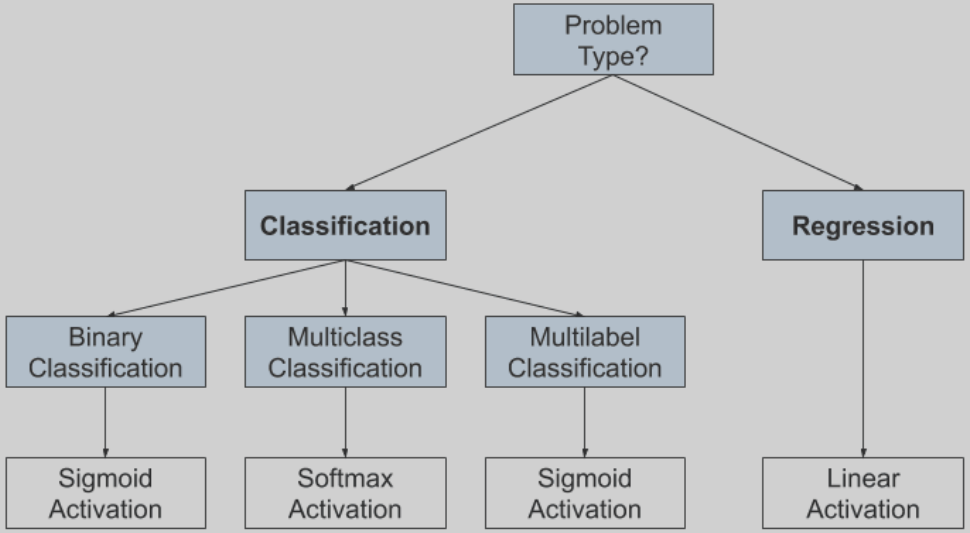


How to Choose the Activation Functions

Hidden Layer Activation Functions



Output Layer Activation Functions



The Learning Process of an ANN

The **learning (or training) process of an ANN** involves the **minimization of a loss** (also called **cost** 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**) $(\boldsymbol{W}, \boldsymbol{b}) \equiv \left\{\boldsymbol{\omega}_{\boldsymbol{ij}}^{(\boldsymbol{k})}, \boldsymbol{b}_{\boldsymbol{i}}^{(\boldsymbol{k})}\right\}$ of the network.

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*

Cost Functions

Regression Cost Functions

✓ Mean Squared Error (MSE)

$$C_{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

✓ Mean Absolute Error (MAE)

$$C_{MAE} = \frac{1}{N}\sum_{i=1}^{N}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

Classification Cost Functions

✓ Binary Cross-Entropy

$$C_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i log\left(p(y_i)\right) + (1 - y_i) log\left(1 - p(y_i)\right)\right)$$

✓ Categorical Cross-Entropy

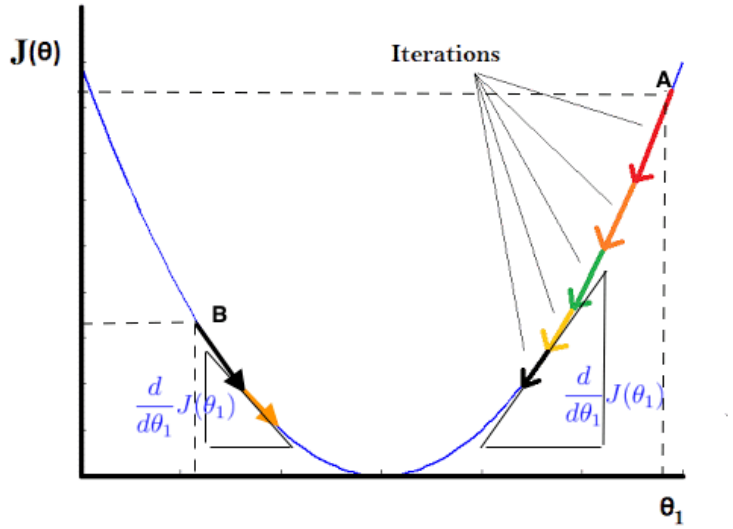
$$C_{CCE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log\left(p(y_{ij})\right)$$

Gradient Descent

Most of the **minimization algorithms** employed in ANN are based on the so-called **gradient descent algorithm**

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, that controls how fast or how slow the network parameters are updated

Gradient Descent Algorithm

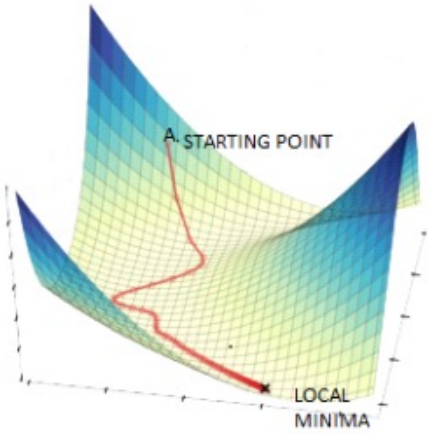
The Gradient Descent Algorithm is quite simple. For **each epoch** (or **iteration**) of the training do the following steps:

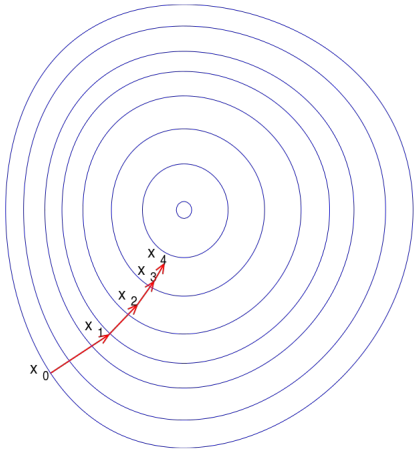
1. Feed the network with the **entire training input dataset** $\vec{x}$

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

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

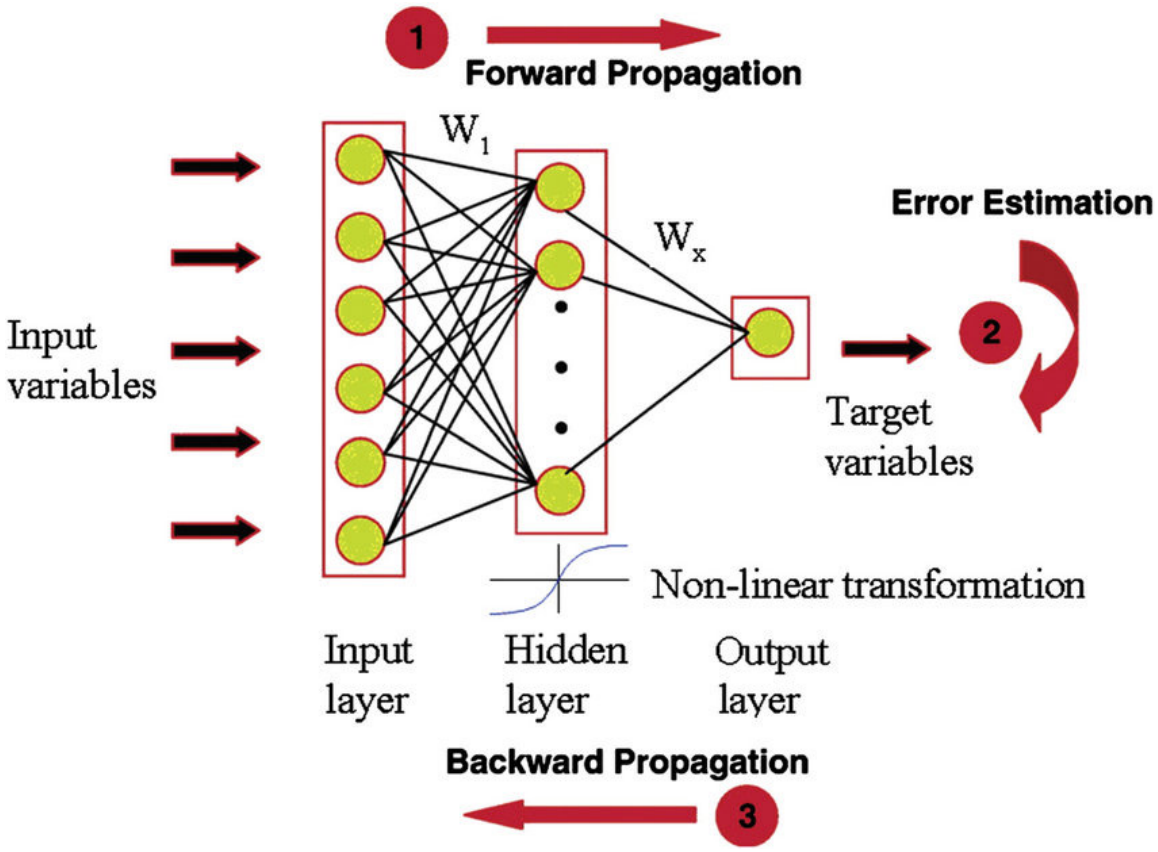
3. Repeat steps 1 – 2 until the convergence the cost function is substancially reduced





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

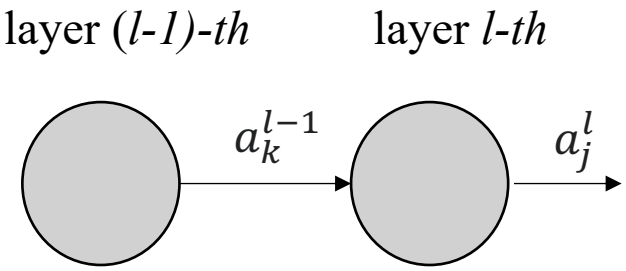


1. In a feed-forward ANN information **propagates sequentially** from through all the layers form the input to the output ones
2. The error (also known as **cost or loss function**) is evaluated
3. The error is **propagated backwards** to determine the new values of the fitting parameters at each layer & 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*
- $a_j^l = f(z_j^l)$: **activation** (output) of neuron *j-th* in the layer *l-th* (note that $\boldsymbol{a_j^L = \hat{y}_j}$)
- $z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$: **weighted input to the activation function** of neuron *j-th* in the layer *l-th*



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* depends on the weighted inputs of the previous layer *l-th*, 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, $\delta_j^L = \frac{\partial C}{a_j^L} f'(z_j^L) = \frac{\partial C}{\partial \hat{y}_j} f'(z_j^L)$
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 $\left(b_j^l\right)$

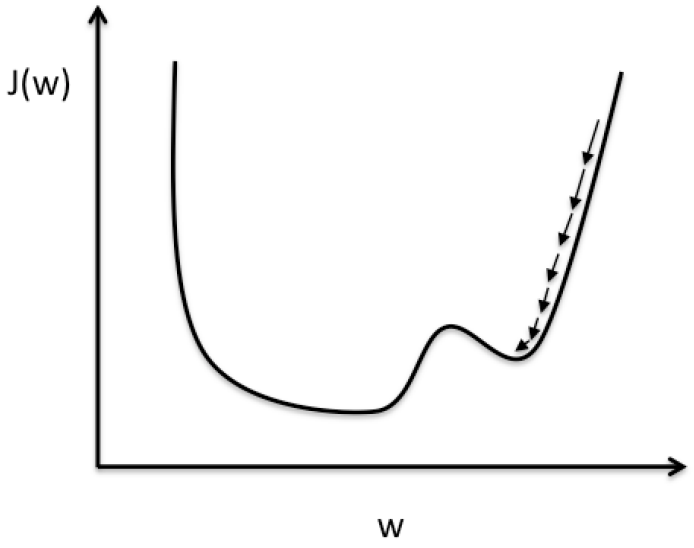
This looks very promissing, we only have to compute $C\left(\omega_{jk}^l, b_j^l\right)$, $C\left(\omega_{jk}^l - \epsilon e_{jk}^l, b_j^l\right)$ and $C\left(\omega_{jk}^l, b_j^l - \epsilon e_j^l\right)$ for each distincts weight ω_{jk}^l and bias b_j^l. **However,** this is **extremelly expensive computationally 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**

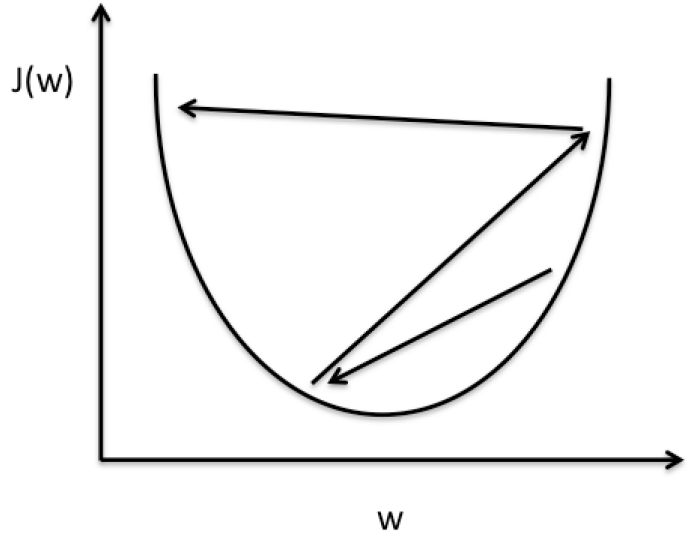
A comment in the learning rate η

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**

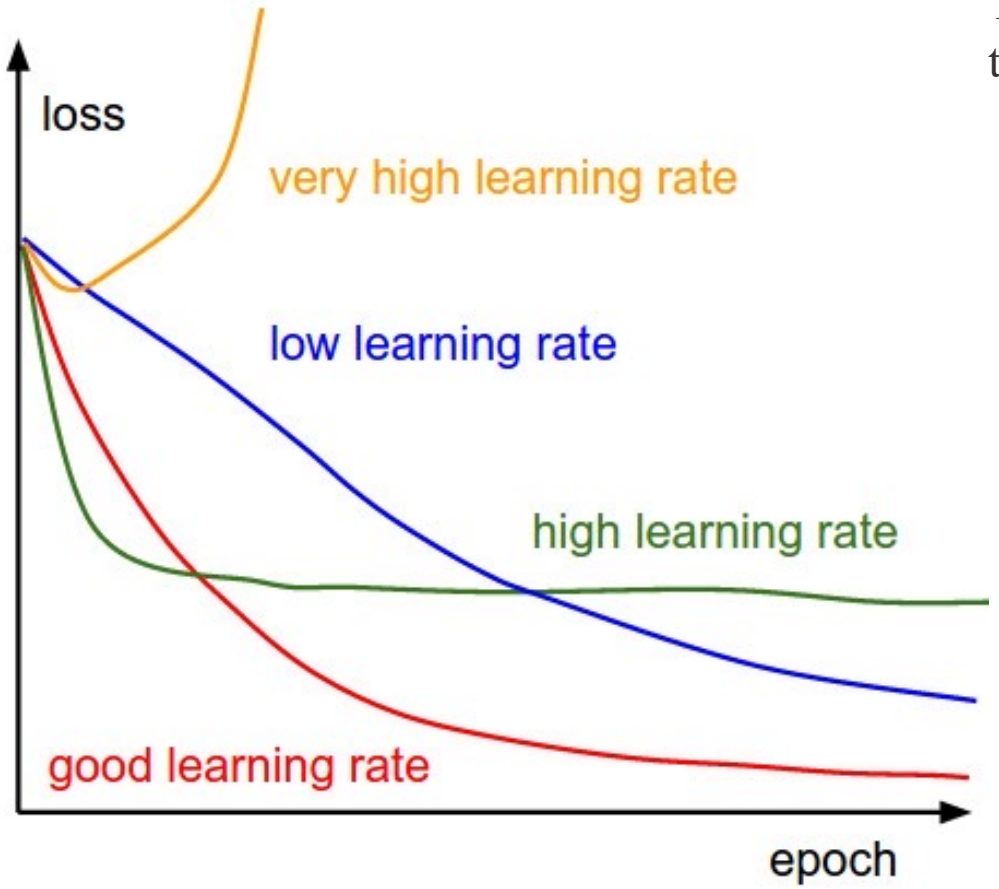


Small learning rate: Many iterations until convergence and trapping in local minima.



Large learning rate: Overshooting.

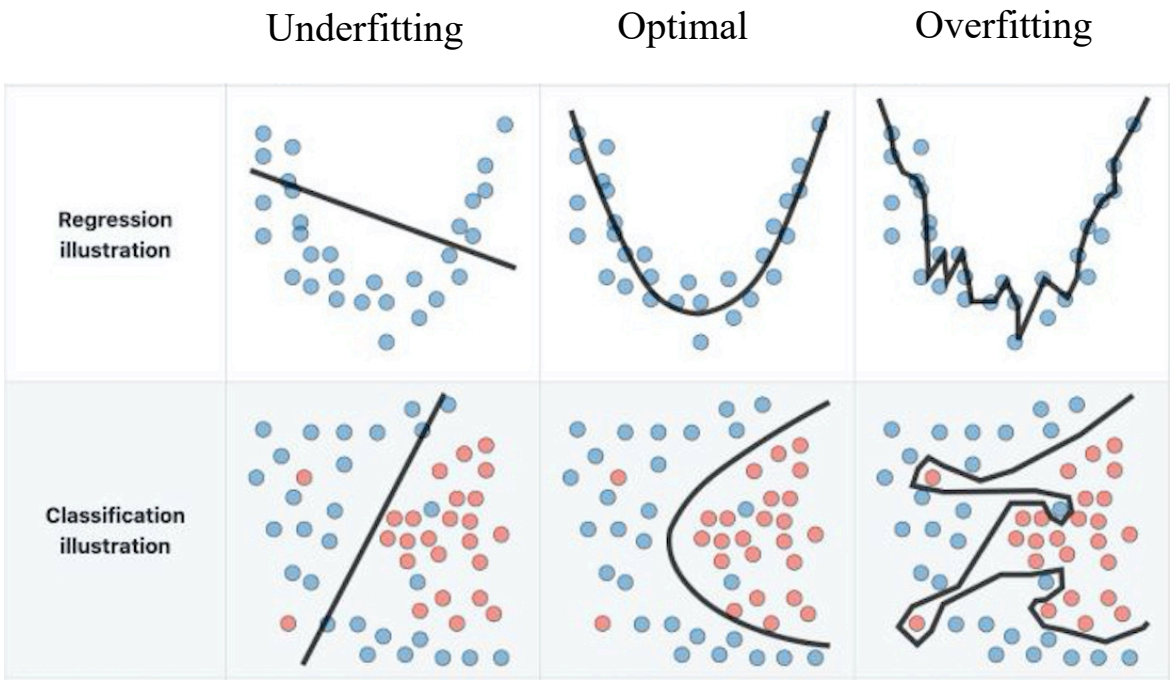
Effect of the learning rate η in the convergence of the Gradient Descent Algorithm



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 **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 **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.

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

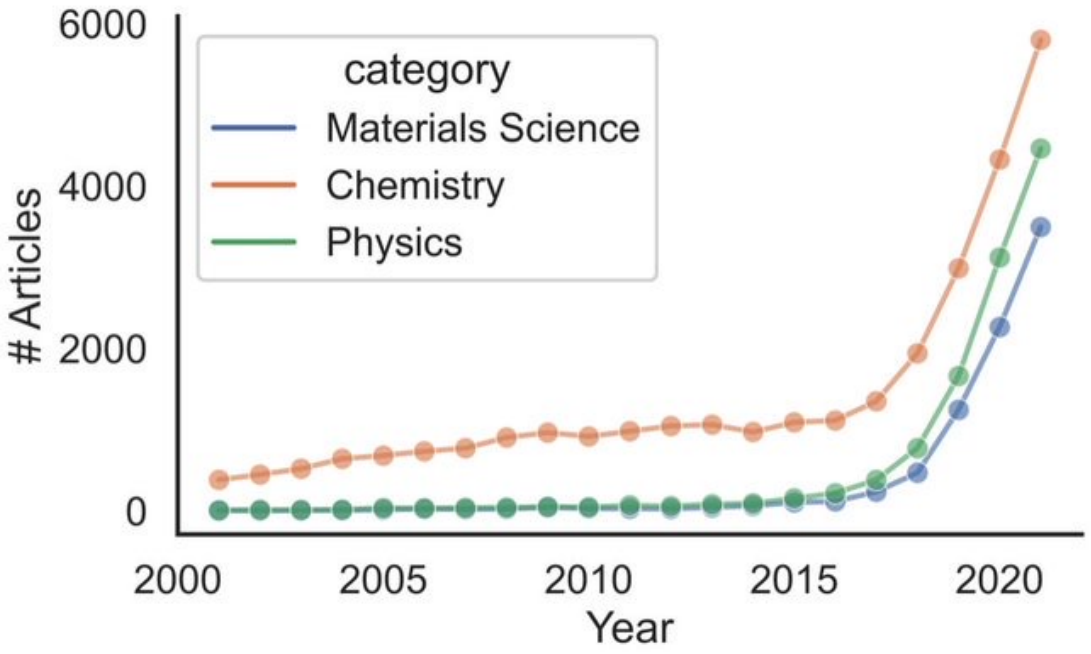
- Strategies to avoid overfitting include among others:
  - early stopping of the training: stops the training process once the model performance stops improving on the validation dataset
  - dropout: reduce overfitting by dropping randomly neurons from the neural network during training in each iteration

- 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

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
- …

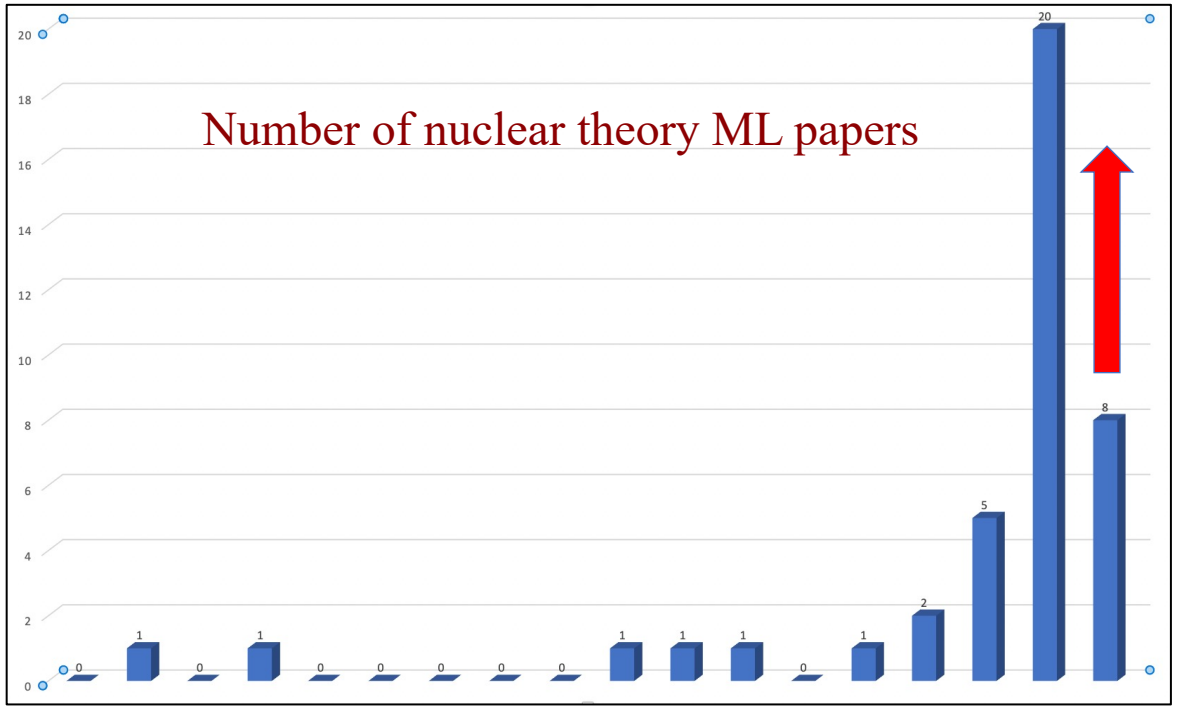


A **spectacular increase** of the number of publications related with AI or ML is observed in physical sciences in the last years

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

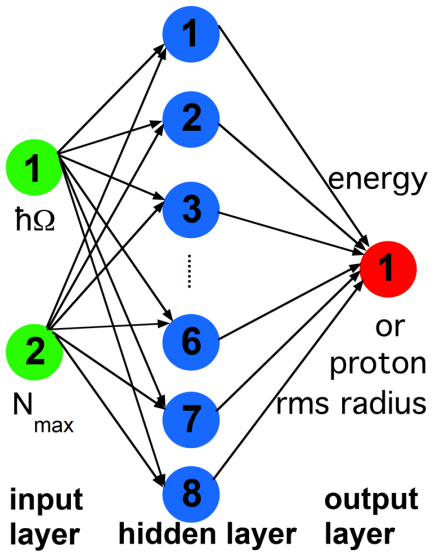
- Nuclear masses & charge radii
- α- & β-decay half-lives
- Fission yields
- Fusion reaction cross sections
- Isotropic cross-sections in proton-induced spallation reactions
- Ground and excited state energies
- Dripline locations
- The deuteron properties
- Proton radius
- Liquid-gas phase transition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

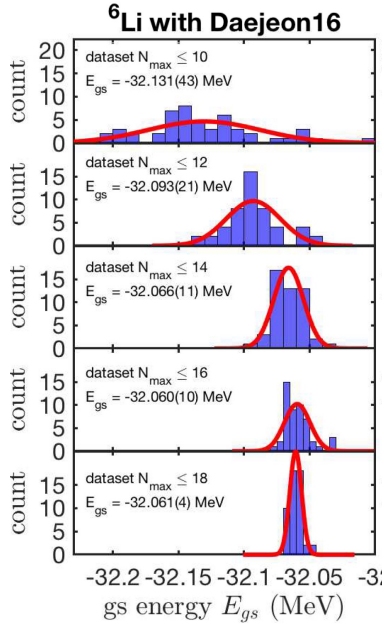


Machine Learning Applications in Nuclear Theory

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

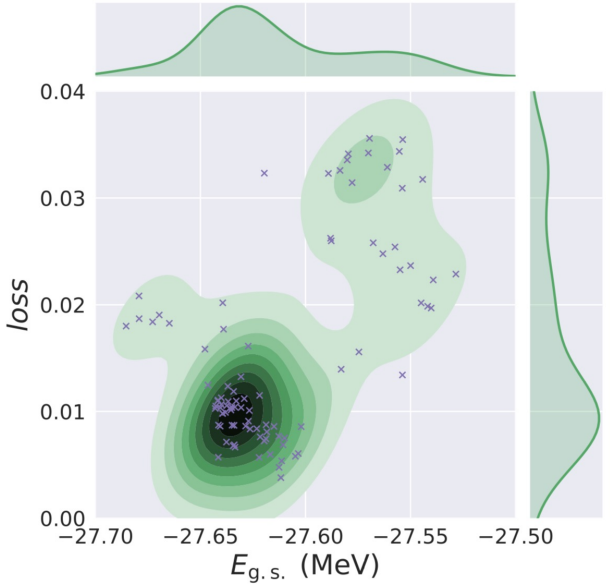
- *Negoita et al., PRC 99, 054308 (2019)* have used a feed-forward ANN method for predicting the **ground state energy and the ground state point proton root-mean-squared radius of ^{6}Li** training the network with NCSM results, obtained in accessible harmonic oscillator (HO) basis spaces. They showed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



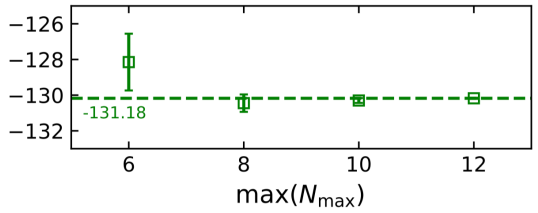


- Similarly, *Jiang et al., PRC 100, 054326 (2019)* have also employed an ANN to extrapolate the **ground state energy and radii of ^{4}He, ^{6}Li & ^{16}O** computed with the NCSM and the coupled-cluster (CC) methods.

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



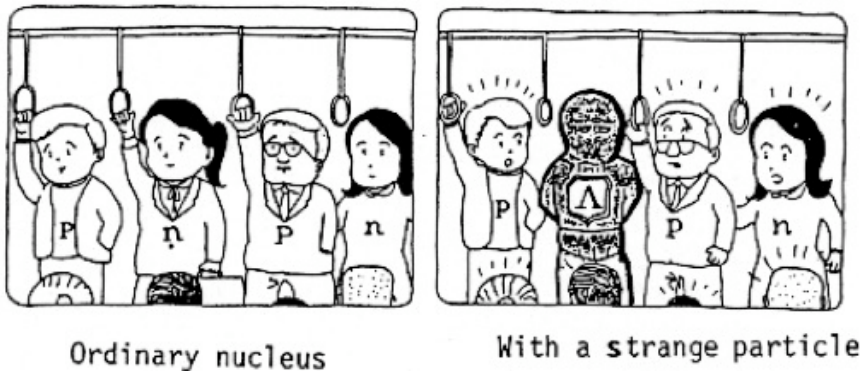
^{16}O with CC+NNLO$_{opt}$



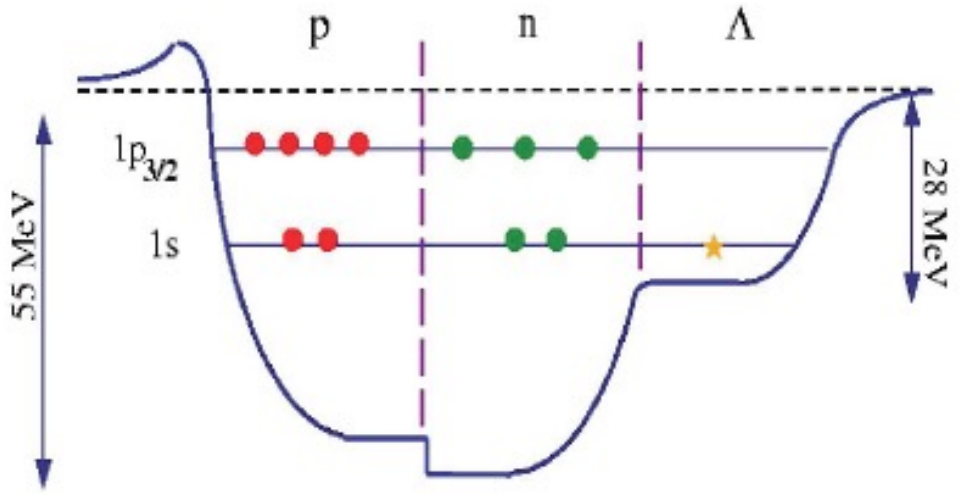
Here we follow the work of these authors, to extrapolate at large model spaces the results of *ab-initio* hypernuclear NCSM calculations for the Λ separation energy B_{Λ} of the lightest hypernuclei

What is an hypernucleus ?

- A hypernucleus is a bound system of nucleons with one or more strange baryons (Λ, Σ, Ξ, Ω^- hyperons).



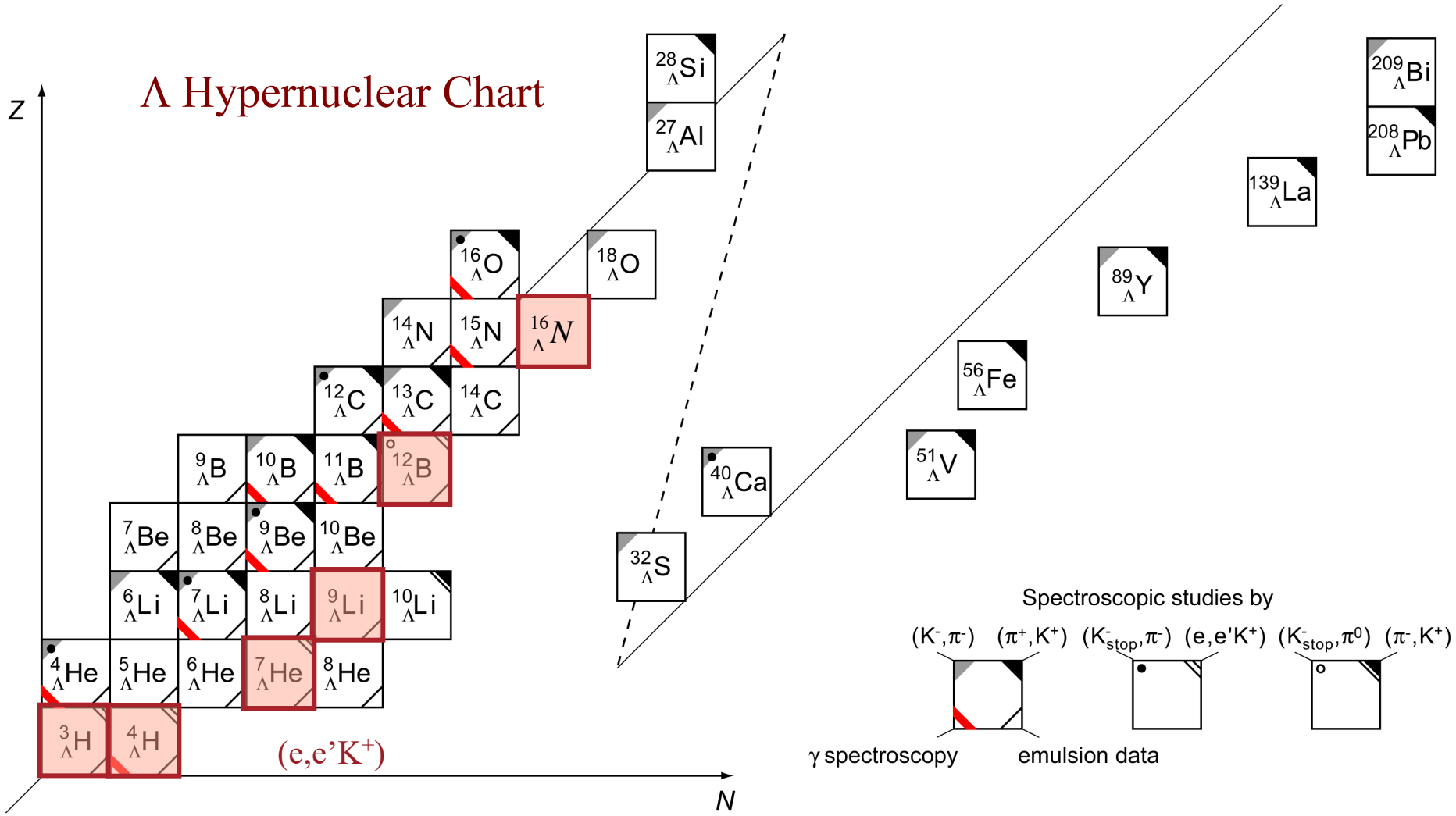
H. Bando, PARITY 1, 54 (1986)



Simple s.p. model of ${}^{12}_{\Lambda}C$

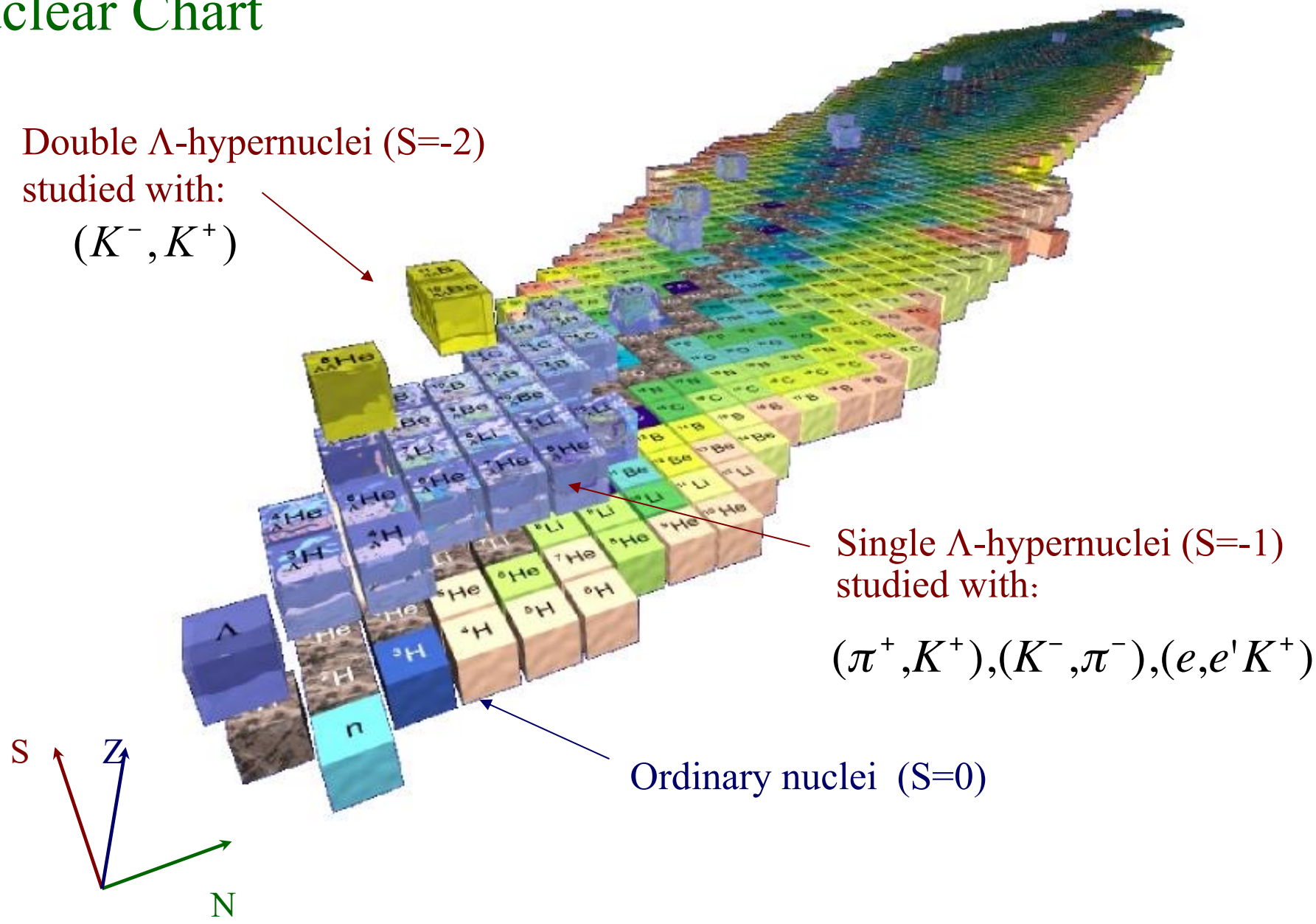
- In a simple single-particle model: protons, neutrons and hyperons are considered distinguishable particles placed in independent effective potential wells in which Pauli exclusion principle is applied.

Present status of Λ hypernuclear spectroscopy

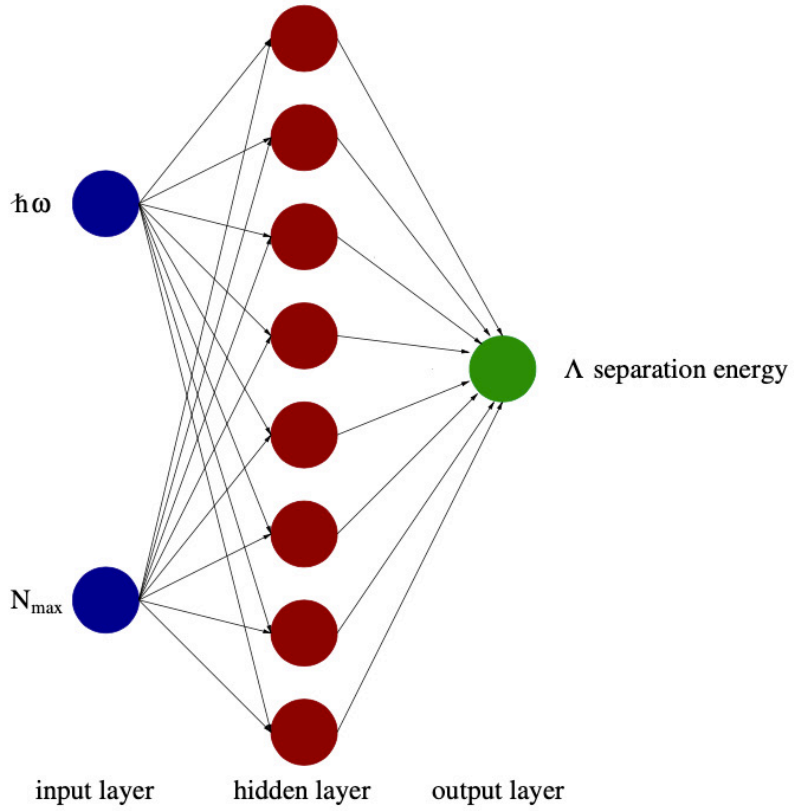


O. Hashimoto and H. Tamura, Prog. Part. Nucl. Phys. 57, 564 (2006)

The 3D Nuclear Chart



Architecture of our Feed-forward Artificial Neural Network



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

- Our network is a feed-forward ANN, tha means that neurons do not form a cycle and the data propagates sequentially from the input to the output layer through all the hidden layers
  - In this work we use a sigmoid activation function

$$f(z)=(e^{z}+1)^{-1}$$

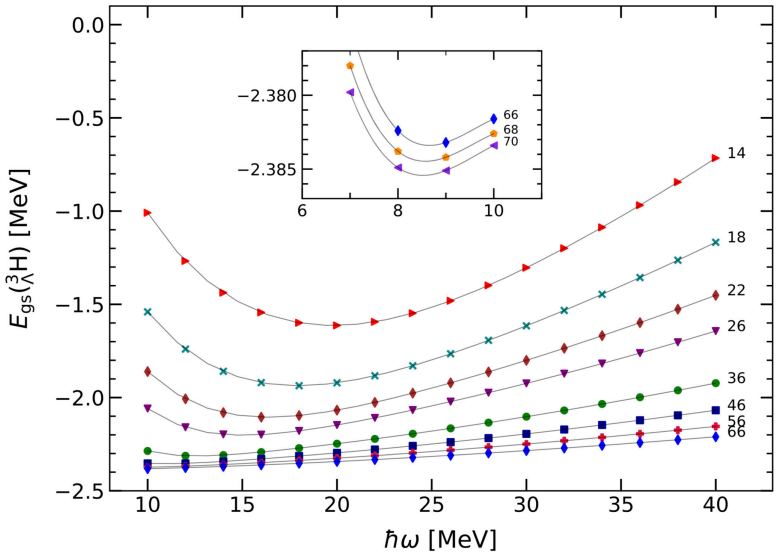
- In this work we are solving a regression-type problem and, therefore, we chose the *mean squared error* (MSE) as cost function

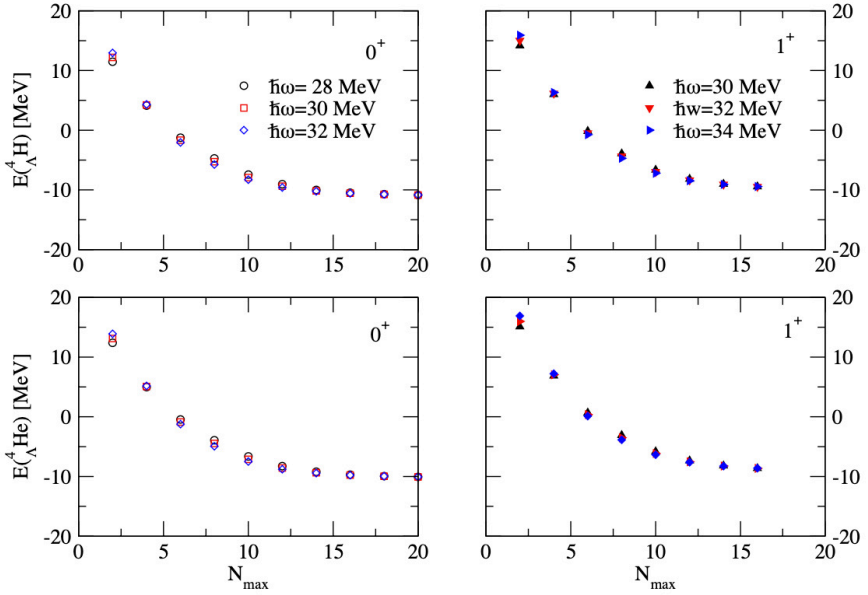
$$C(\boldsymbol{W},\boldsymbol{b})=\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i(\boldsymbol{W},\boldsymbol{b})-y_i)^2$$

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

Input Dataset

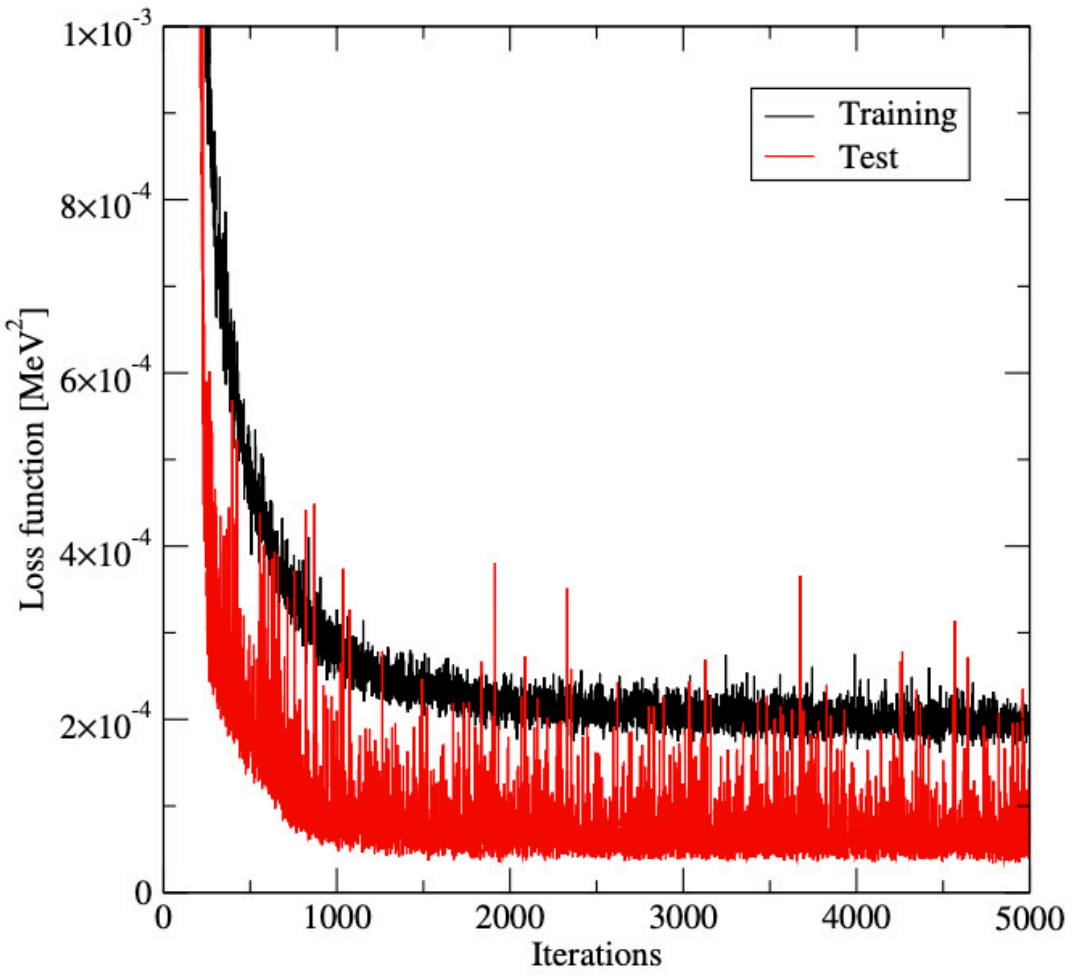
We employ as input dataset the hypernuclear NCSM results of Gazda *et al.* (PRC 97 (2018) 064315, Few-Body Syst. 62 (2021) 94) for the Λ separation energy of ${}^{3}_{\Lambda}$H, ${}^{4}_{\Lambda}$H & ${}^{4}_{\Lambda}$He obtained with chiral NN & NNN interactions at N^{3}LO and N^{2}LO, respectively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

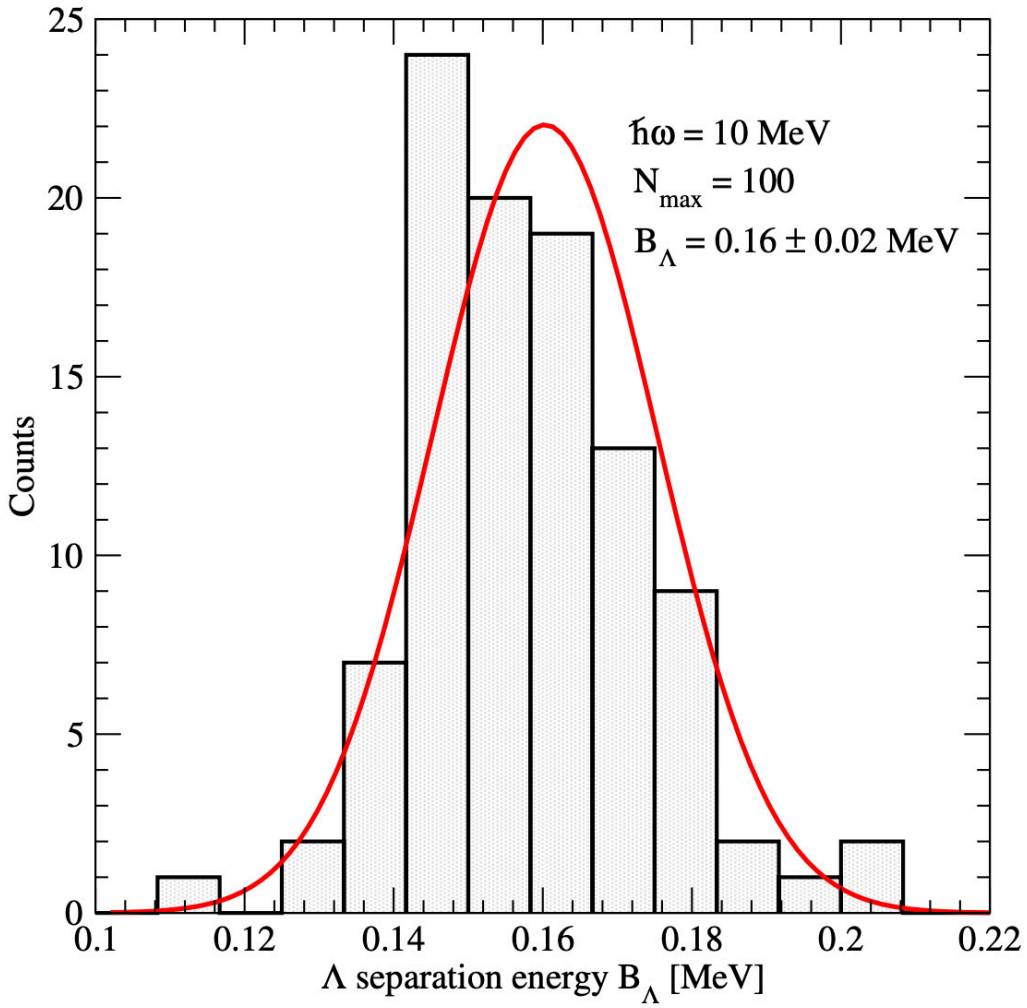
Performance of the ANN



To illustrate the performance of the network we show the loss function $L(\boldsymbol{W}, \boldsymbol{b})$ 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 ${}^{3}_{\Lambda}\mathrm{H}$

- 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 ${}^{4}_{\Lambda}\mathrm{H}$ and ${}^{4}_{\Lambda}\mathrm{He}$

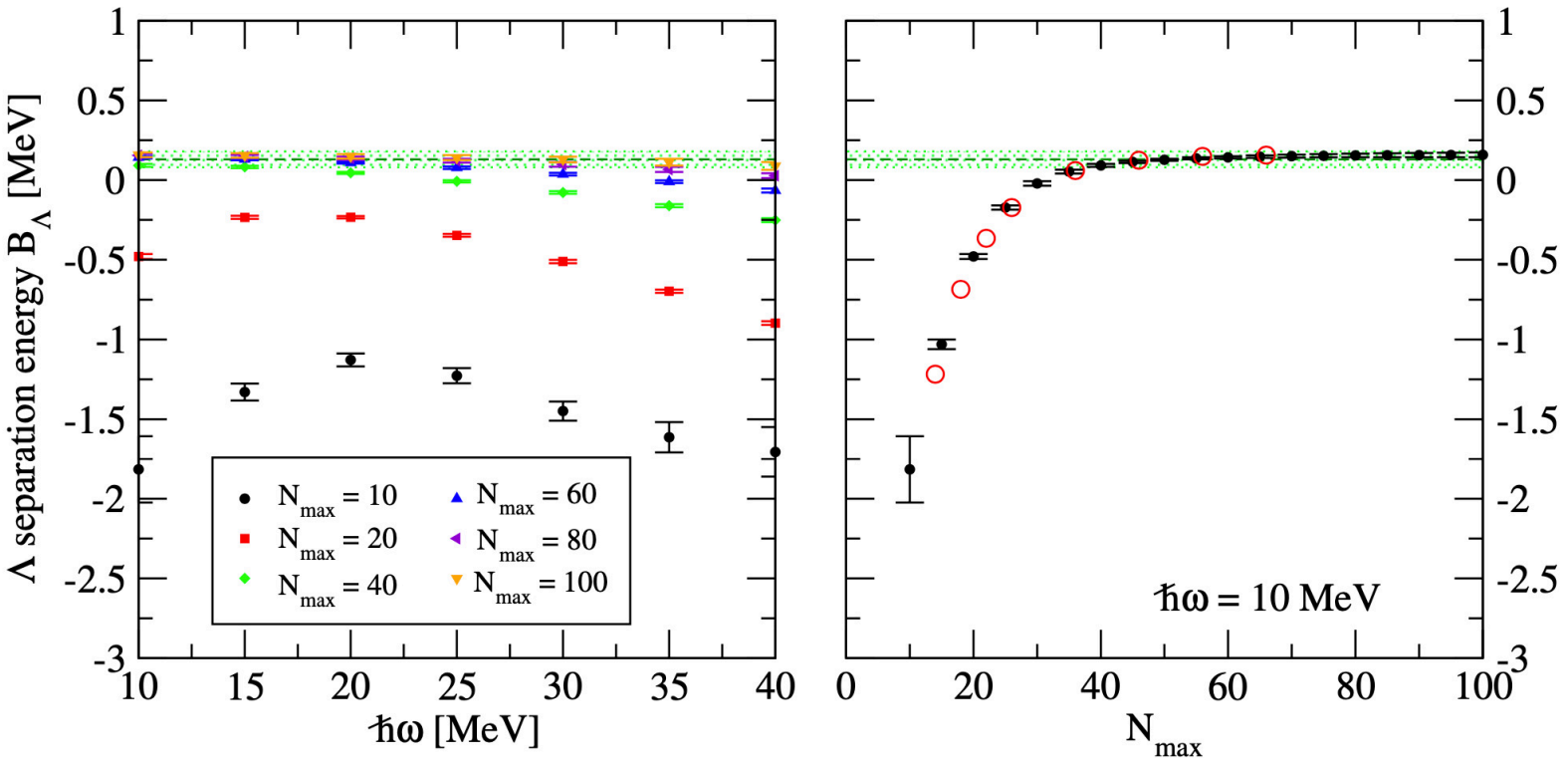
A General Comment



- A typical run of an ANN starts with random values of the weights & biases of the network.
- The random inizialization of the weights & biases is not accidental but an important feature of the network training that introduces a certain degree of stochasticity that reduces the risk that during the optimization process of the network parameters it gets stuck in a local minimum
- Consequently, different runs of the ANN lead to slightly different results as seen in the figure
- Because of this, for each hypernucleus, we have performed 100 independent runs of the ANN and taken the average and the standard deviation of all these runs ad the predictions of the network and their corresponding error, respectively

Statistical distribution of the results for the Λ separation energy B_Λ of the ground state of ${}^{3}_{\Lambda}\mathrm{H}$ predicted by 100 independent runs of the ANN for an HO spacing $\hbar\omega$= 10 MeV and a model space size N_{max} = 100. The continuous line shows a Gaussian fit of the histogram.

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



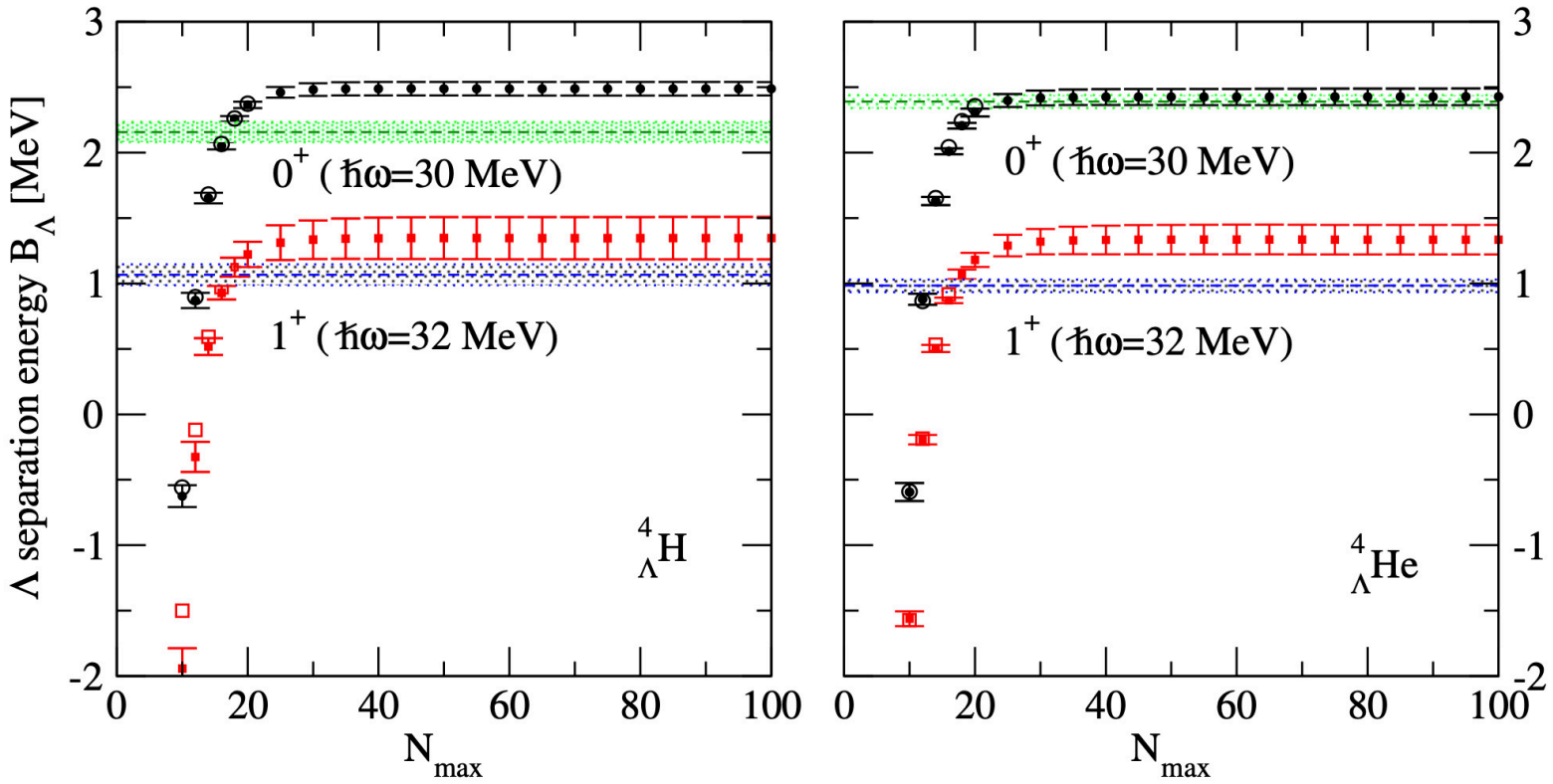
Open circles in the right panel show the NCSM results used for the training of the ANN

- Slow convergence due to the extremely weak binding energy of ${}^{3}_{\Lambda}H$
- Considerably reduction of the B_Λ dependence with $\hbar\omega$ 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 $N_{max} = 100$

$$B_\Lambda\left({}^{3}_{\Lambda}H\right) = 0.16 \pm 0.02 \text{ MeV}$$

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



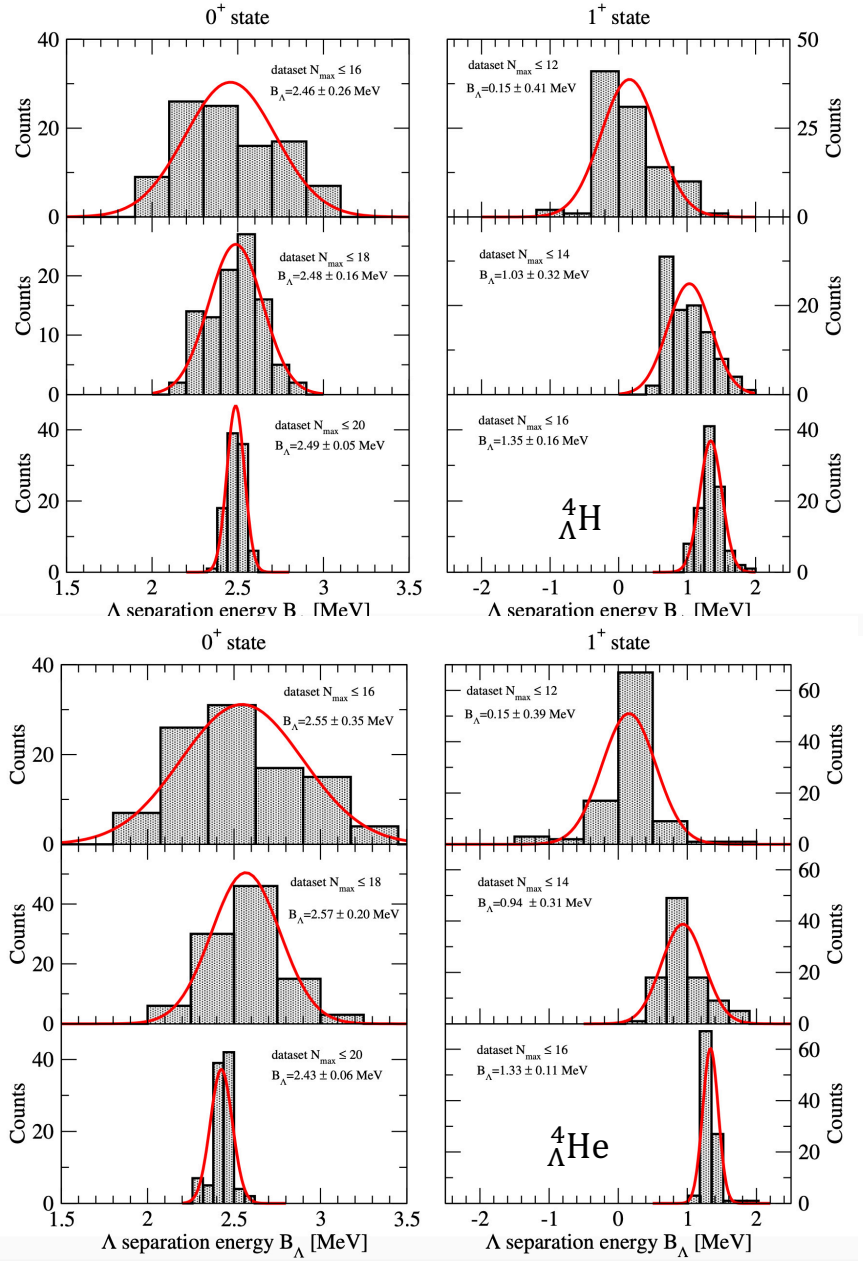
ANN prediction for $N_{max} = 100$

$B_\Lambda\left({}^4_\Lambda H(0^+)\right) = 2.49 \pm 0.05$ MeV $B_\Lambda\left({}^4_\Lambda He(0^+)\right) = 2.43 \pm 0.06$ MeV

$B_\Lambda\left({}^4_\Lambda H(1^+)\right) = 1.35 \pm 0.16$ MeV $B_\Lambda\left({}^4_\Lambda He(1^+)\right) = 1.33 \pm 0.11$ MeV

- Faster convergence than in the ${}^3_\Lambda$H case. Good convergence already for $N_{max} \gtrsim 25$
- 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. This should not be attributed to the performance of the ANN but to the Hamiltonian used & the symmetries assumed in the NCSM calculations
- 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

Convergence of the extrapolated results of the 0^+ & 1^+ states at N_{max}=100



To further check the convergence of the extrapolated value of B_Λ for the 0^+ & 1^+ states of ${}^{4}_{\Lambda}$H & ${}^{4}_{\Lambda}$He at N_{max}=100 we show the statistical distributions of the results of 100 independent runs of the ANN for several choices of the maximal value of N_{max} taken into account in the training dataset

We observe that:

- The dispersion of the ANN predictions becomes narrower & narrower when the maximal value of N_{max} included in the training dataset increases
- Successive extrapolates are consistent consistent with the previous ones within the given uncertainties

From this one can conclude that the extrapolated values of B_Λ for the two states of both hypernuclei show a rather well convergence in terms of the maximal value of N_{max} included in the training dataset, being this an indication that ANN are a reliable method to extrapolate the results of hypernuclear NCSM calculations to large model spaces

Comparison with other extrapolation schemes

The main goal of this work is mainly focused on discerning whether ANN is a reliable scheme to extrapolate NCSM results at larger model spaces rather than its accuracy on reproducing the experimental results. To such end we compare our results with the extrapolated ones of Htun *et al.*, FBS 62 (2021) 94 [1] & Wirth *et al.*, PRC 97 (2018) 064315 [2]

Hypernucleus	ANN Prediction	Extrapolated results of [1] & [2]	Experimental Vaue
${}^{3}_{\Lambda}\mathrm{H}$ (g.s.)	0.16 ± 0.02	0.158 [1]	0.13 ± 0.05
${}^{4}_{\Lambda}\mathrm{H}(0^{+})$	2.49 ± 0.05	2.48 ± 0.04 [2]	2.157 ± 0.077
${}^{4}_{\Lambda}\mathrm{H}(1^{+})$	1.35 ± 0.16	1.40 ± 0.28 [2]	1.067 ± 0.08
${}^{4}_{\Lambda}\mathrm{He}(0^{+})$	2.43 ± 0.06	2.45 ± 0.04 [2]	2.39 ± 0.05
${}^{4}_{\Lambda}\mathrm{He}(1^{+})$	1.33 ± 0.11	1.34 ± 0.28 [2]	0.984 ± 0.05

As it is seen our results are in excellent agreement with those of Htun *et al.* & With *et al.*, confirming this that ANN is a reliable method to extrapolate results of hypernuclear NCSM calculations to large model spaces

Take home message

Summary & Conclusions

- 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
- Our results are in excellent agreement with those obtained using other extrapolation schemes of hypernuclear NCSM calculations, showing this that ANN is a reliable extrapolation method

Future Perspectives

- Compare the ANN results with those obtained with other extrapolation schemes such that of infrared extrapolation (IR) where the model space parameter $\hbar\omega$ & N_{max} are translated into an IR length scale L_{eff} and an ultraviolet (UV) cutoff scale Λ_{UV}
- Analyze the ANN performance, particularly the convergence of the ANN results, on heavier hypernuclei

- ✧ You for your time & attention
- ✧ Felipe for his kind invitation
- ✧ 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"