Neural Networks for Signal Analysis

November 15, 2023

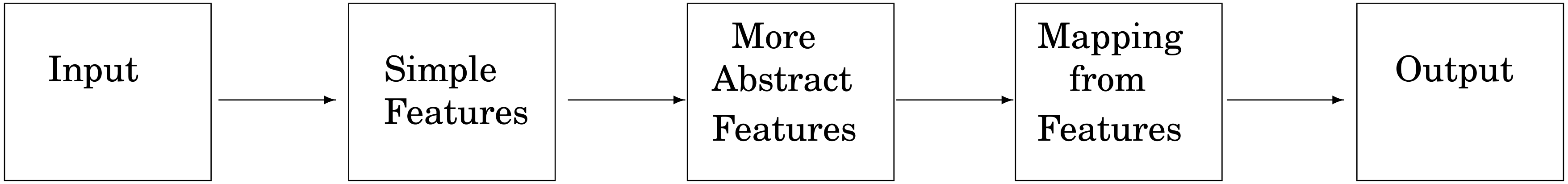
G. Castellanos-Dominguez

Electronics and Computing, UNAL, Manizales

Contents

Neural Networks for Signal Analysis

Heuristical Principles



$$\eta = f(\xi)$$

The dependence model f is built by heuristical principles and designed to simulate the human brain's problem-solving method.

$$\boxed{X \in \xi \overset{f}{\mapsto} Y \in \eta}$$

A neural network and the brain are similar in that both gain skills and knowledge as a result of training (*Learning*).

Neural Network structures tend to resemble anatomical and functional brain organization. NN includes basic computing units, which are interconnected through synaptic weights.

Multi-Layer Perceptron

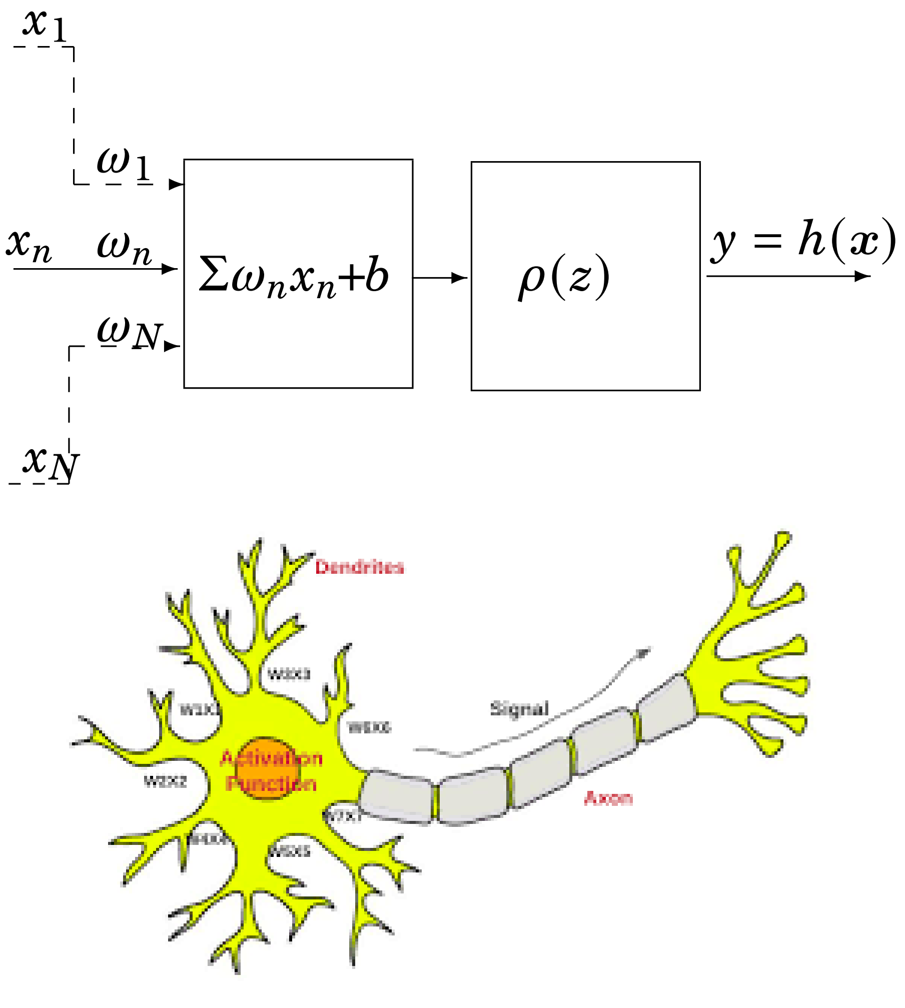
Neurons: Neurons are elementary units of a neural network, which perform the following steps on the collected information inputs x=$\{x_n$:n∈$N\}$:

i) Computation of the synaptic vector, $\omega \in \mathbb{R}^N$, for reweighting the inputs analogously to a biological neuron using a linear combination of x along with an added bias b; z=$\omega^\top x + b$.

ii) Learning a complex hypothesis through a rule $\rho(z)$ or *activation function*:

$$h(x) = \rho(\omega_1 x_1 + \ldots + \omega_n x_n + \ldots + \omega_N x_N + b)$$

Perceptron: the relationship between inputs and outputs $\rho(x)$=sign(x) maps any value nonlinearly to a couple of saturation values $h_{\min}$=0 and $h_{\max}$=1. That is, y∈$\{0, 1\}$.



Structure of generic artificial neuron

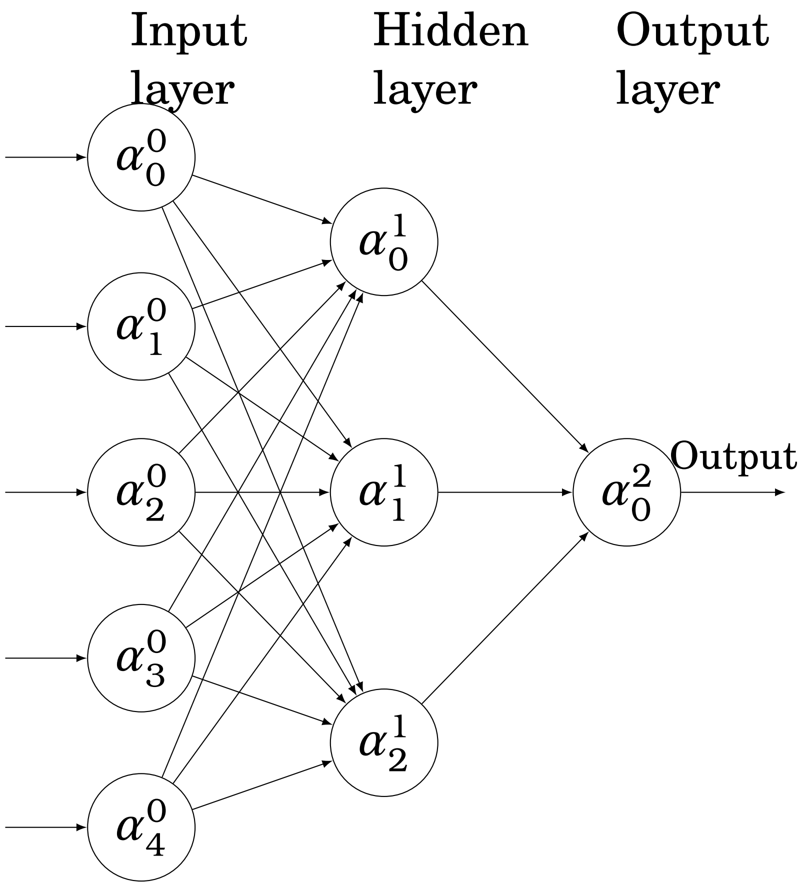
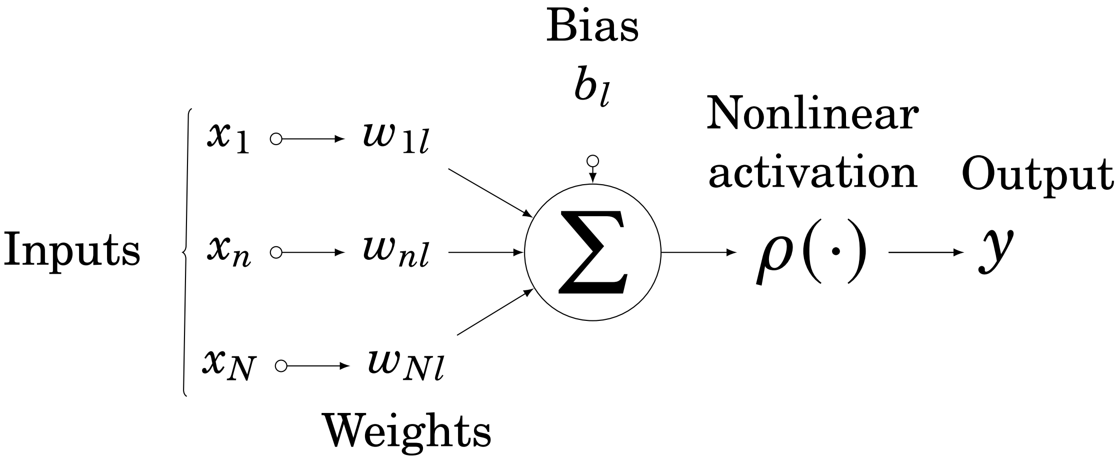


diagram of NN layers



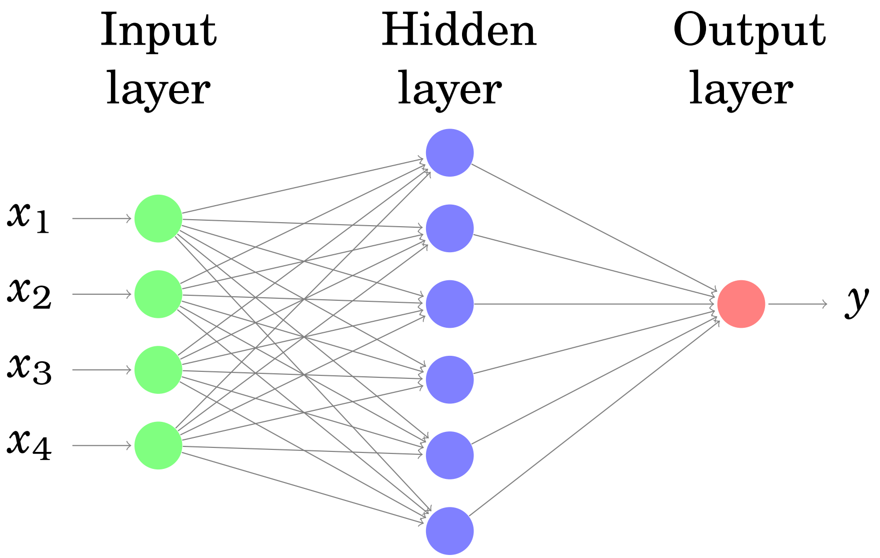
NN pipeline diagram

Perceptron Architecture 41aPerceptron

Backpropagation Perceptron Architecture

An elemental NN architecture of processing holds many layers of connected neurons, relaying data directly from the front to the back:

$$
\begin{aligned}
f: &\quad \mathbb{R}^n \mapsto \mathbb{R}^p\\
f(x) &= g(\ldots\rho(\boldsymbol{\omega}_2\rho(\boldsymbol{\omega}_1\boldsymbol{x}+b_1)+b_2))\\
f(x) &= \boxed{g\circ\cdots\circ f_2\circ f_1(x)}
\end{aligned}
$$



- $\boldsymbol{x}$ - input layer, $g()$ - output function,
- An arbitrary amount of hidden layers $\boldsymbol{h}$ with elements expressed as:

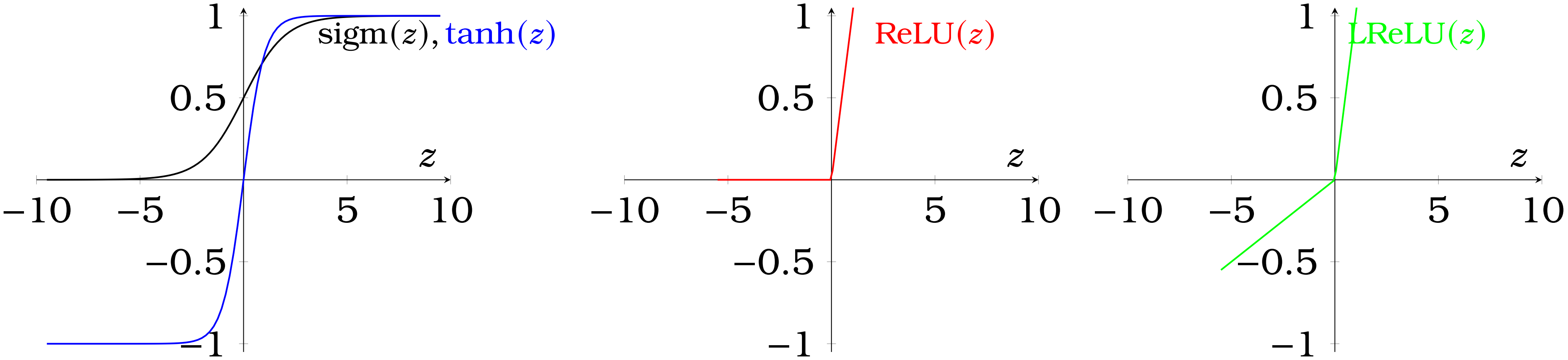
$$\rho\left(\sum\nolimits_k (w^l_{jk}\cdot h^{l-1}_k)+b^l_j\right)=\rho(z)=\begin{cases}1, & \text{if } z\geq 0\\ 0, & \text{Otherwise}\end{cases}$$

- Real-valued weights $\boldsymbol{\omega}^l$, a bias b^l of each l-layer, j- neuron; $0\leq h^{l-1}_k\leq 1$, and $z=\boldsymbol{\omega}_l\boldsymbol{h}^{l-1}+b^l$ is the Activation Potential.

Activation Function

AF determines whether a neuron is activated or not by calculating the weighted sum and adding bias with it, introducing nonlinearity into each layer's outputs.

Activation functions must be monotonic, differentiable, and quickly converge to the weights for optimization purposes. `41cActivation`



$$\mathrm{sigm}(z) = \frac{1}{1+\exp^{-z}}, \quad \tanh(z) = \frac{\exp^{z} - \exp^{-z}}{\exp^{z} + \exp^{-z}}, \quad \mathrm{ReLU}(z) = \max(0, z), \quad \max(\alpha z, z)$$

- **sigm**(z): It will give an analog activation, unlike the step function.

  Cons: Sigmoids saturate and tends to kill gradients (*vanishing gradients*), making the network refuse to learn further or be drastically slow.

- tanh(z): The gradient is stronger than sigmoid, i.e., derivatives are steeper, still susceptible to the vanishing gradient problem.

- ReLU(z): It avoids and rectifies the vanishing gradient problem. ReLu is less computationally expensive than tanh and sigmoid because it involves simpler math operations.

  Cons: It should only be used within hidden layers of a Neural Network Model. Some gradients can be fragile during training and can die. It can cause a weight update, making it never activate on any data point again. Thus, ReLu could even result in Dead Neurons.

- Leaky ReLU attempts to fix the "dying ReLU" problem by having a slight negative slope.

  Cons: it can not be used for complex classification because of its linearity.

The concrete use of activation depends on the problem type and the value range of the expected output.

Cost Function

A real-valued measure of fitting between the input and outputs of NN:

$$\mathscr{C}^* = \boxed{\operatorname{opt}_{\boldsymbol{W},B,} \mathbb{E}\left\{\mathscr{C}(\boldsymbol{W}, \boldsymbol{B}, \boldsymbol{x}^r, \boldsymbol{y}^r) : \forall r \in \boldsymbol{R}\right\}}$$

where $\boldsymbol{W}$ is our neural network's weights, $\boldsymbol{B}$ is our neural network's biases, $\boldsymbol{y}^r$ is the input of a single training sample, and $\boldsymbol{y}^r$ is the desired output of r- training sample.

In backpropagation architectures, the cost function computes the output layer error h_j^L via:

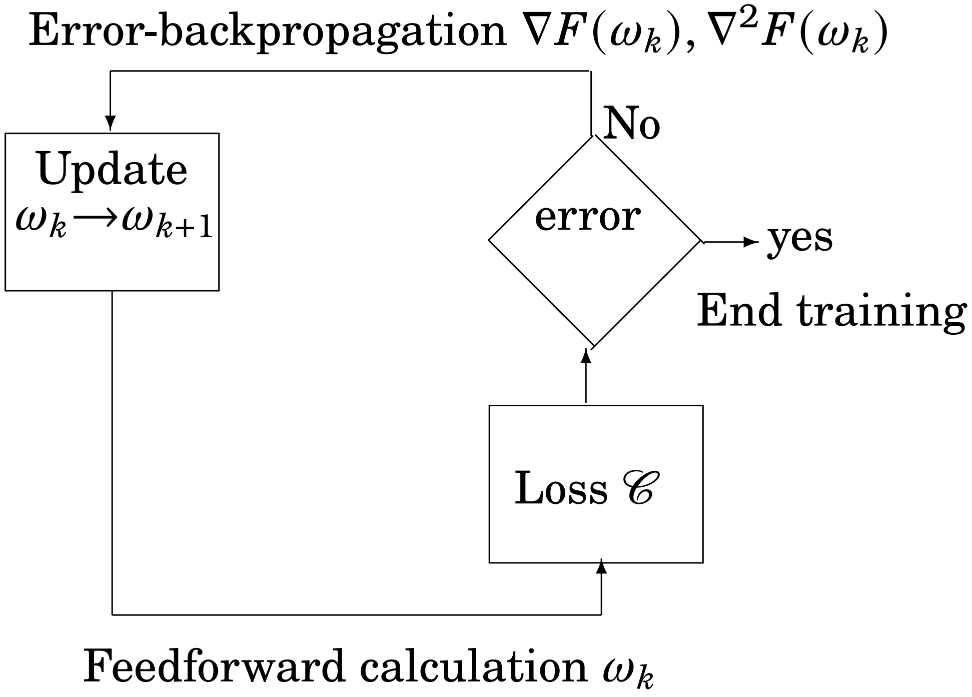
$$\mathscr{C}(\cdot) = \mathbb{E}\left\{(\mid h_j^L - y_j^r \mid) : \forall j\right\}, \quad \ell_1 - \text{cost and } \ell_1\text{-regularization (Lasso)}$$

$$\mathscr{C}(\cdot) = \mathbb{E}\left\{(h_j^L - y_j^r)^2 : \forall j\right\}, \ell_2 - \text{cost and } \ell_2\text{-regularization (Ridge)}$$

$$\mathscr{C}(\cdot) = \mathbb{H}(h_j^L, y_j^r)^2, \quad \text{Cross-Entropy}$$

Optimization by back-propagation

The backpropagation algorithm evolves iteratively downwards from the output to the input layer and closes the loop in the updating rule (activation functions).



Error-backpropagation $\nabla F(\omega_k), \nabla^2 F(\omega_k)$

Update $\omega_k \rightarrow \omega_{k+1}$

No

error

yes

End training

Loss $\mathscr{C}$

Feedforward calculation ω_k

– A feed-forward propagation step to compute the network output.

– A backward propagation step in which the error at the network end is propagated backward through all the neurons while updating their parameters.

Non-linear activation function:

$$\rho(z_l) = \exp(-z_l^2),$$
$$\frac{d\rho(z_l)}{dz_l} = -2z_l \exp(-z_l^2) = -2z_l\rho(z_l)$$

Forward step

- Compute activations of hidden layer

$$\boldsymbol{h}^l = \rho(\boldsymbol{x} \cdot \boldsymbol{\omega}^l) = \exp(-(\boldsymbol{x} \cdot \boldsymbol{\omega}^l)^2)$$

- Compute activations of output, in this case, using the logistic classifier function:

$$\boldsymbol{y} = g(\boldsymbol{h} + b_0) = \frac{1}{1 + \exp(-(\boldsymbol{h} + b_0))}$$

Backward step: The gradient of the loss is propagated backward layer by layer, through the network to calculate the parameter updates.

- Compute the loss function, Cross-Entropy is selected.

$$\mathscr{C}(h_r^L, y_r) = -(y_r \log(h_r^L + (1 + y_r \log(1 - h_r^L))))$$

- Update the output layer. At the gradient output for sample r, the update for parameter b_0 can be worked out using the chain rule

$$\frac{\partial \mathscr{C}_r}{\partial b_0} = \frac{\partial \mathscr{C}_r}{\partial h_r}\frac{\partial h_r}{\partial z_{0r}}\frac{\partial z_{0r}}{\partial b_0} = (h_r - y_r) = \delta_{0r}, \quad z_{0r} = h_r + b_0$$

  $\delta_{0r} = \partial\mathscr{C}_r/\partial z_{0r}$ is the gradient of the error at the output layer concerning the input.

- Update the hidden layer. At the hidden layer parameter ω_h, the gradient for sample r is computed as:

$$\frac{\partial \mathscr{C}_r}{\partial w_h} = \frac{\partial \mathscr{C}_r}{\partial h_r}\frac{\partial h_r}{\partial z_{hr}}\frac{\partial z_{hr}}{\partial w_h} = x_r\delta_{hr}, \quad z_{jr} = \boldsymbol{x}_r\boldsymbol{\omega}_r$$

  $\delta_{hr} = \mathscr{C}_r/\partial z_{hr}$ is the gradient of the error at the input of the hidden layer for the input.

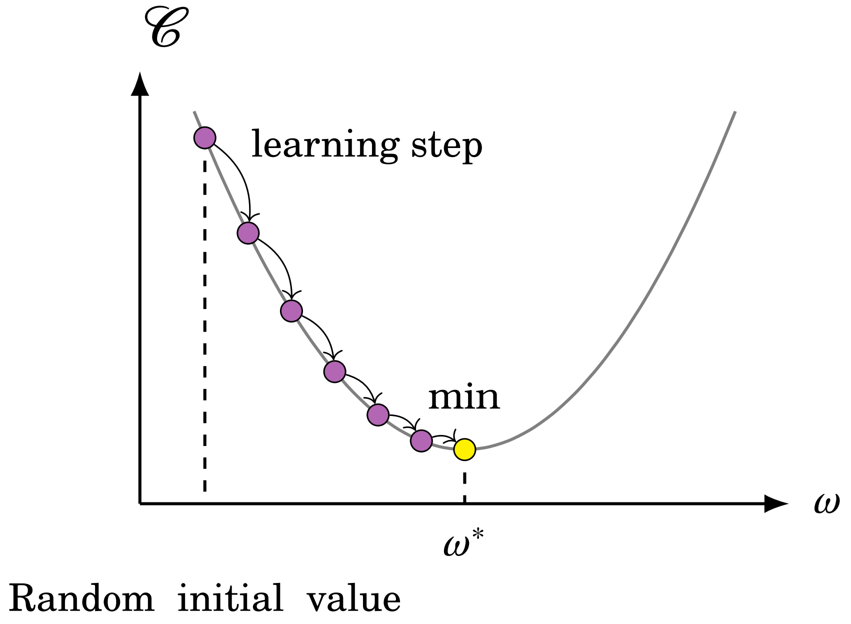
Backpropagation updates. Gradient descent minimizes the direction of the gradients of the loss concerning the parameters. Update of parameters $ve\omega_r$ and b_0 are scaled by the learning rate, $\mu{\in}\mathbb{R}^+$:

$$w_h(k+1) = w_h(k) - \Delta w_h(k+1) \quad \text{with } \Delta w_h = \mu\cdot\partial\mathscr{C}/\partial w_h$$
$$b_0(k+1) = b_0(k) - \Delta b_0(k+1) \quad \text{with } \Delta b_0 = \mu\cdot\partial\mathscr{C}/\partial b_0$$

The iterative first-order optimization algorithm finds a local minimum/maximum of a given function (cost/loss function).

Requirements for a function, f, to be optimized: i) differentiable ii) convex.

Gradient Descent Algorithm iteratively calculates the next point using gradient at the current position, then scales it (by a learning rate) and subtracts obtained value from the current location (makes a step):



$$p_{n+1} = \boxed{p_n - \eta \nabla f(p_n)}$$

Stochastic gradient descent (SGD) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g., differentiable or subdifferentiable) `41eGradDesc`

Developing Multi-Layer BP Neural Networks

Parameter Tuning

Number of layers: 0- only capable of representing linear separable functions or decisions; 1 - NN can approximate any function that contains a continuous mapping from one finite space to another, 2- NN can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy

There are many rule-of-thumb methods, such as the following `04bBackPropagation`:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.
- Yet, most problems can be solved by using a single hidden layer with the number of neurons equal to the mean of the input and output layers.

Back-propagated errors

In the pipelined backpropagation algorithm, numerical instabilities inherent to computational recurrences (very large variances differences in scale between inputs and outputs) and saturating nonlinearities cause the Exploding/Vanishing gradient problem:

- *Exploding gradients*. There is an exponential growth in the model parameters if the activations, for some reason, get out of control. The model weights may become NaN during training. The model experiences avalanche learning.
- *Vanishing gradients*. The parameters of the higher layers change significantly, whereas the parameters of lower layers would not change much (Covariate shift), frequently leading to saturated activation units and, consequentially, vanishing gradients. That is, the model weights may become 0 during training. The model learns very slowly, and perhaps the training stagnates very early, just after a few iterations.

As a result, the gradient descent never converges to the optimum. Thus, the model gets stuck and inhibits it from learning.

Stochastic NN-based Learners

Recurrent LSTM

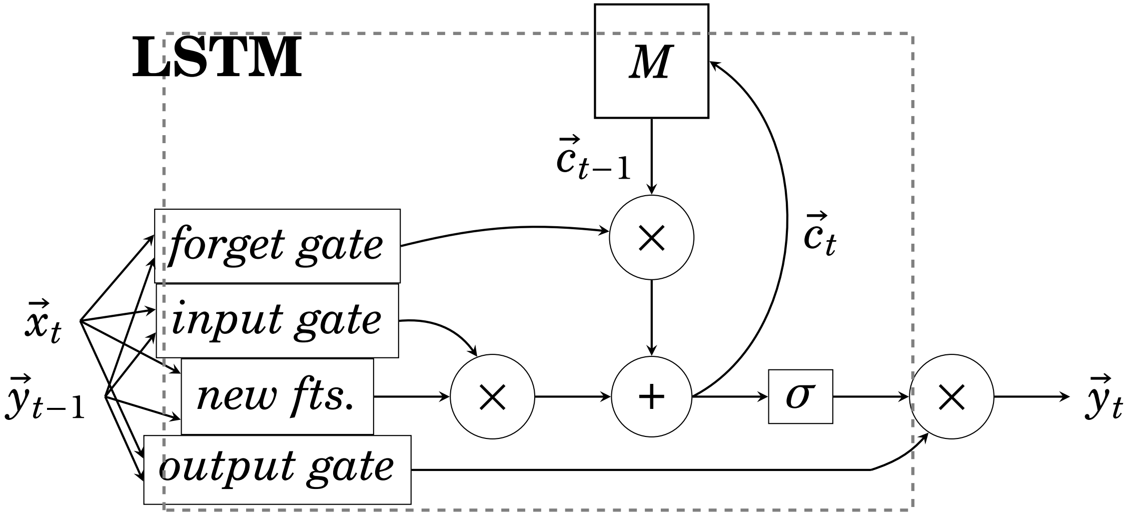


diagram representing a single LSTM block, in the context of recurrent neural networks

Autoencoders

(AE) – neural networks that compress (encoder – $g_\phi(x^i)$) the input x into a latent-space representation with a lower dimension to be expanded into the output replicating the input $\tilde{x} = f_\theta(g_\phi(x^i))$ (decoder): Encoder – hidden layer – decoder

$h = g((w \cdot x) + b)$, w - weights, b – bias, g – nonlinear activation. AE with linear activations is basically SVD

$\tilde{x} = f((w' \cdot h) + c)$

$\dim(h) < \dim(x)$ – under complete autoencoder

$\dim(h) < \dim(x)$ – over complete autoencoder

Goal : Output identical to input

Hyperparameters : Code (Bottleneck) size – compression rate

Number of layers – encoder/decoder depth

Number of nodes per layer – weight dimension per layer

Reconstruction Loss function – MSE, Binary Cross Entropy. Backpropagation-based minimization of error

- time-series AE

$$\mathcal{L}\{\theta, \phi\} = \mathbb{E}\left\{x^i - f_\theta(g_\phi(x^i))\right\}$$

- Convolutional AE: $C(x^i, y^j)$ 2D convolutional layer

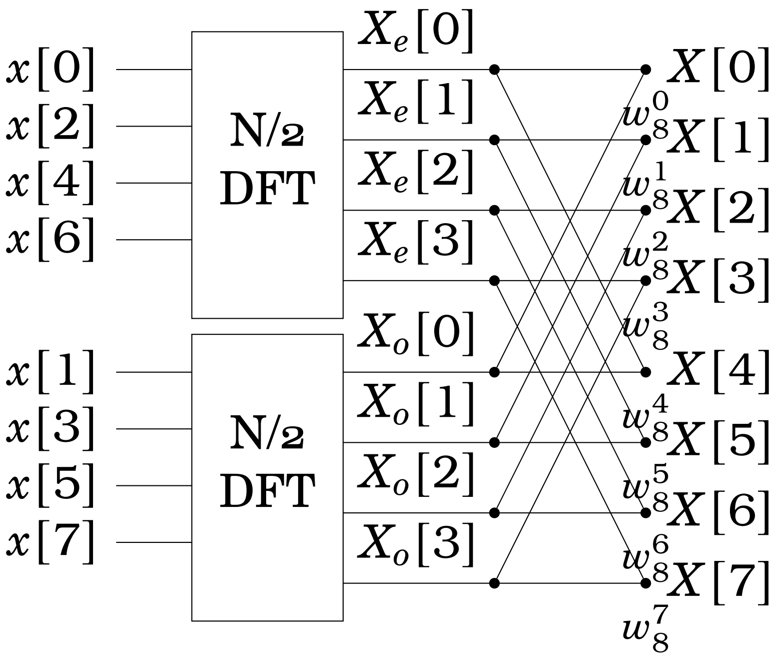
$$\mathcal{L}\{\theta, \phi\} = \mathbb{E}\left\{C(x^i, y^j) - g_\theta(g_\phi(C(x^i, y^j)))\right\}$$

NN-based Signal Processing

Discrete Fourier Transform as Linear MLP

DFT is computed by a single layer NN with a linear transfer (weighting) function W and particular weights values (*Butterfly*).
`42aLinearNNFFT`

The computational burden increases as the number of inputs rises. Because of linearity, models can be calculated analytically, resulting in low-cost complexity.



Butterfly computation in DFT

Applications

- Dimensionality reduction `41eRollingPCA,41fKompressAE`
- Feature extraction
- Denoising compresion
- Anomaly detection
- Missing value imputation

Signal Prediction by MLP

$$\mathbb{E}\left\{P\big(\eta(s_n)\mid\eta(s_{n-1}),\dots,\eta(s_{n-m})\big)\right\}:\quad \eta(s_{n-1}),\dots,\eta(s_{n-m})\overset{f}{\mapsto}\eta(s_n)\in\mathbb{R},\ \text{Prediction}$$

Multilayer Perceptron for forecasting involves predicting future values of a time-dependent variable based on its historical values. `03d1MLPredictionKeras` `03d2MLPredictionPytorch`

- *Preprocessing*: Data Preparation, Data Splitting: An observation sequence $\boldsymbol{x}=[x_t:t\in T]$ must be partitioned into multiple segments (samples), lasting $L<T$, from which the model can learn.

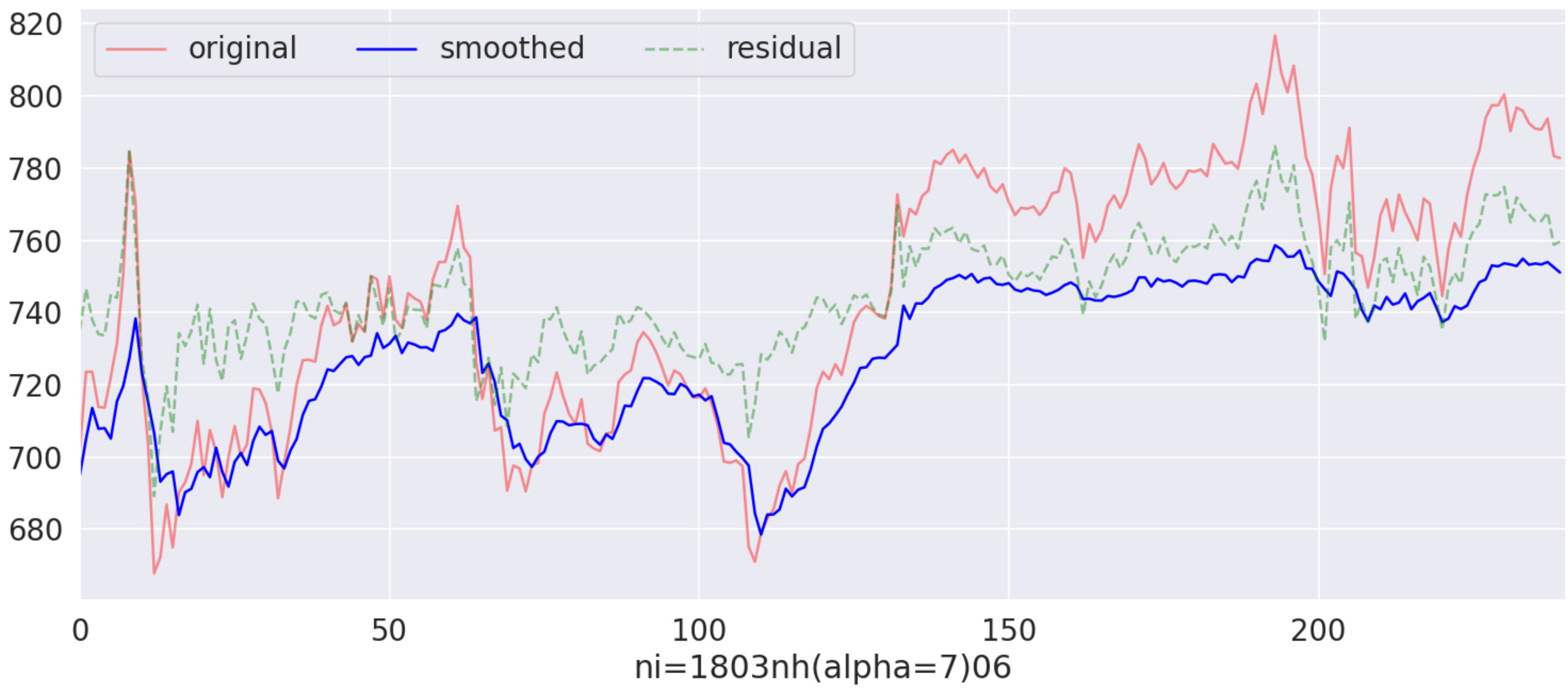
$$\boldsymbol{x}\to\tilde{y}$$

$$x_{L+1-n},\dots,x_{2L-n}\to x_{2L+1-n}: \qquad \text{one-step prediction}$$

$$x_{L+1-n},\dots,x_{2L-n}\to x_{2L+1-n},\dots,x_{2L+1-n+m}: \qquad m\text{-step prediction}$$

- *Model building*: Model Architecture, Hyperparameter Tuning, Model Training
- *Model Evaluation*: Validation, Forecasting

Signal Smoothing by NN



Tareas Q

- Multivariate Spectral Estimation using MLP
  - Import the required libraries (from scipy.signal import welch) and Python environment (pip install numpy scipy matplotlib)
  - Prepare your multivariate 2D data: each row represents a different trial, and columns represent the time series data.
  - Choose parameters for spectral estimation: the sampling frequency (fs) and the window function (e.g., Hamming window).
  - Calculate the power spectral density using the Welch method, dividing the data into overlapping segments and averaging their periodograms to reduce noise.
- Implementar el filtro de Wienner en forma multivariada
- Comparar ambos algoritmos de MLP en keras y pytorch para predicción
- Implementar en LSTM la extracción de la componente deterministica en `03eSmoothWT`. Incluir validación.

Signal Detection using NN

Detection of a signal with *unknown* constant:

$$\begin{aligned} H_0 : &\quad y_k = \eta_k, \\ H_1 : &\quad y_k = \alpha x_k + \eta_k; \quad \forall k \in [0, K-1] \end{aligned}$$

where x_i is a given function, α is unknown constant, and η_k is White Gaussian Noise having a known variance σ_k^2.

$$\Lambda(y|\hat{\alpha}) = \frac{P(y|\hat{\alpha})}{P(y)} = \frac{P(\boldsymbol{y}|\hat{\alpha})}{P(\boldsymbol{y})} \quad > \quad \gamma$$

where the a posterior probabilities are modeled as:

$$\begin{aligned} P(\boldsymbol{y}|\hat{\alpha}) &= \frac{1}{(2\pi\sigma_\eta^2)^{K/2}} \exp\left(-\frac{1}{2\sigma_\eta^2}(\boldsymbol{y} - \alpha\boldsymbol{x})^\top(\boldsymbol{y} - \alpha\boldsymbol{x})\right) \\ P(\boldsymbol{y}) &= \frac{1}{(2\pi\sigma_\eta^2)^{K/2}} \exp\left(-\frac{1}{2\sigma_\eta^2}\boldsymbol{y}^\top\boldsymbol{y}\right) \end{aligned}$$

$$\boldsymbol{y} = [y_0, y_1, \ldots, y_{K-1}]^\top, \quad \boldsymbol{x} = [x_0, x_1, \ldots, x_{K-1}]^\top$$

and $\hat{\alpha}$ is the value of α, estimated by the Maximum Likelihood (ML) rule as follows:

$$\begin{aligned}\frac{\partial}{\partial\alpha}\ln P(\boldsymbol{y}|\hat{\alpha}) &= 0\\ \frac{\partial}{\partial\alpha}(\boldsymbol{y}^2 - 2\alpha\boldsymbol{x}\boldsymbol{y} + \alpha^2\boldsymbol{y}^2) &= 0\\ -2\boldsymbol{x}\boldsymbol{y} + \alpha\boldsymbol{y}^2 &= 0\end{aligned}$$

So, the optimal value yields the estimation $\hat{\alpha}$:

$$\boxed{\hat{\alpha}} = \frac{\boldsymbol{y}^\top\boldsymbol{x}}{\boldsymbol{x}^\top\boldsymbol{x}} = \frac{1}{\|\boldsymbol{x}\|}\boldsymbol{y}^\top\boldsymbol{x} = \boxed{\sum_{\forall k\in K} y_k x_k \Big/ \sum_{\forall k\in K} x_k^2}$$

Now, we calculate the log of ML inequality, $\Lambda(\boldsymbol{y}|\hat{\alpha}) > \gamma$:

$$-\frac{1}{2\sigma_\eta^2}\sum_{\forall k\in K}(-2\hat{\alpha}x_k y_k + \hat{\alpha}^2\boldsymbol{x}^2) > \ln\gamma$$

Replacing α by its estimate of $\hat{\alpha}$:

$$-\frac{1}{2\sigma_\eta^2}\left(-2\hat{\alpha}\hat{\alpha}\sum_{\forall k\in K} x_k^2 + \hat{\alpha}^2\sum_{\forall k\in K} x_k^2\right) > \ln\gamma$$

Therefore, the hypothesis H_1 about the present signal holds:

$$
\begin{aligned}
\hat{\alpha}^2 &\underset{H_0}{\overset{H_1}{\gtrless}} \frac{2\sigma_\eta^2 \ln \gamma}{\sum_{\forall k \in K} x_k^2} \\
\sum_{\forall k \in K} x_k^2 y_k^2 &\underset{H_0}{\overset{H_1}{\gtrless}} 2\sigma_\eta^2 \ln \gamma \\
\boxed{\left| \sum_{\forall k \in K} x_k y_k \right|} &\underset{H_0}{\overset{H_1}{\gtrless}} \boxed{\sqrt{2\sigma_\eta^2 \ln \gamma}}
\end{aligned}
$$

Issue in implementing the previous decision-making algorithm: vector x is not available practically.

Practical solution:

$$\begin{aligned}
&\mathfrak{N}(\mu_1, \sigma_\eta) + \mathfrak{N}(\mu_2, \sigma_\eta) = \mathfrak{N}((\mu_1 + \mu_2)/2, \sigma_\eta) \\
\Rightarrow &\mathfrak{N}(0, \sigma_\eta) + \mathfrak{N}(\mu_2, \sigma_\eta) = \mathfrak{N}(\mu_2/2, \sigma_\eta)
\end{aligned}$$

Thus, making $K_\alpha \geq K$, we estimate the amplitude as follows:

$$\boxed{\hat{\alpha}} \simeq 2 \sum_{\forall k \in K_\alpha} y_k$$

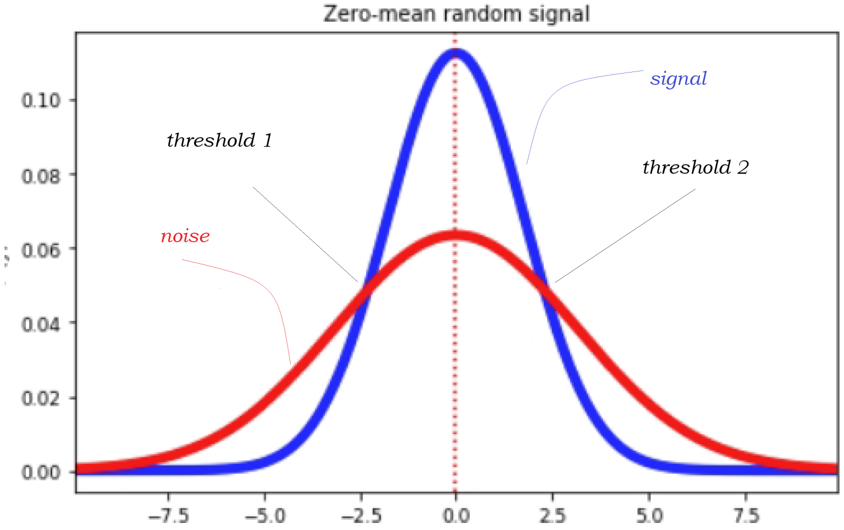
Detection of a zero-mean *random* signal: Detection of a signal with known variance:

$$\begin{aligned}
H_0 :&\quad y_k = \eta_k, \\
H_1 :&\quad y_k = \alpha_k + \eta_k; \quad \forall k \in [0, K-1]
\end{aligned}$$

where η_k is White Gaussian Noise ($\mathfrak{N}(0, \sigma_\eta)$) with zero-valued mean having a known variance σ_η^2, and α_k is random constant with $\mathfrak{N}(0, \sigma_a)$. *Assumption*: $\boxed{\sigma_a^2 > \sigma_\eta^2}$.

Therefore, it holds that:

$$\sigma_y^2 = \sigma_a^2 + \sigma_\eta^2,$$
$$\Rightarrow \Lambda(y|\hat{\alpha}) = \frac{P(y|\hat{\alpha})}{P(y|0)} = \frac{\mathfrak{N}(0, \sigma_y)}{\mathfrak{N}(0, \sigma_\eta)} > \gamma$$



Gaussian signal plus noise

From ML rule, after some simplifications, we have:

$$\sum_{\forall k \in K} y_k^2 \left(\frac{1}{\sigma_\eta^2} - \frac{1}{\sigma_y^2}\right) = K \ln(\sigma_y^2/\sigma_\eta^2 + \gamma)$$

$$\sum_{\forall k \in K} y_k^2 = K m_{2y} = \left(\frac{1}{\sigma_\eta^2} - \frac{1}{\sigma_y^2}\right)^{-1} K \ln(\sigma_y^2/\sigma_\eta^2 + \gamma)$$

$$\sqrt{m_{2y}} = \sqrt{\left(\frac{1}{\sigma_\eta^2} - \frac{1}{\sigma_y^2}\right)^{-1} \ln(\sigma_y^2/\sigma_\eta^2)}, \quad \text{making } \gamma = 1$$

Implementation of NN Learners

Estimation Tasks

Machine Learning simulates the operation of a human brain and enables computing/processing units to learn from data to solve statistical inference tasks:

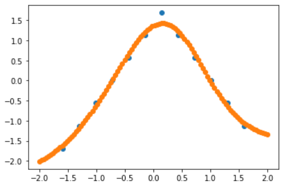
$$P(\xi;\theta): \quad \xi(\theta) \overset{f}{\mapsto} \hat{\theta}\in\mathbb{R}, \text{ Parameter Estimation}$$

$$P(\eta \mid \xi): \quad \xi \overset{f}{\mapsto} \eta\in\mathbb{R}, \text{ Regression}$$

$$P(\eta(s_n) \mid \eta(s_{n-1}),\ldots,\eta(s_{n-m})): \quad \eta(s_{n-1}),\ldots,\eta(s_{n-m}) \overset{f}{\mapsto} \eta(s_n)\in\mathbb{R}, \text{ Prediction}$$

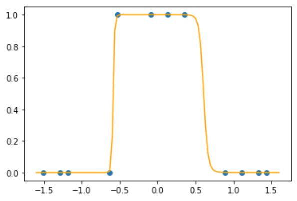
$$P(\eta \mid \xi): \quad \xi \overset{f}{\mapsto} \eta\in\mathbb{N}, \text{ Classification}$$

$$f(x) = \frac{w_{21}}{1+e^{-(w_{11}x+b_{11})}} + \frac{w_{22}}{1+e^{-(w_{12}x+b_{12})}} + b2$$

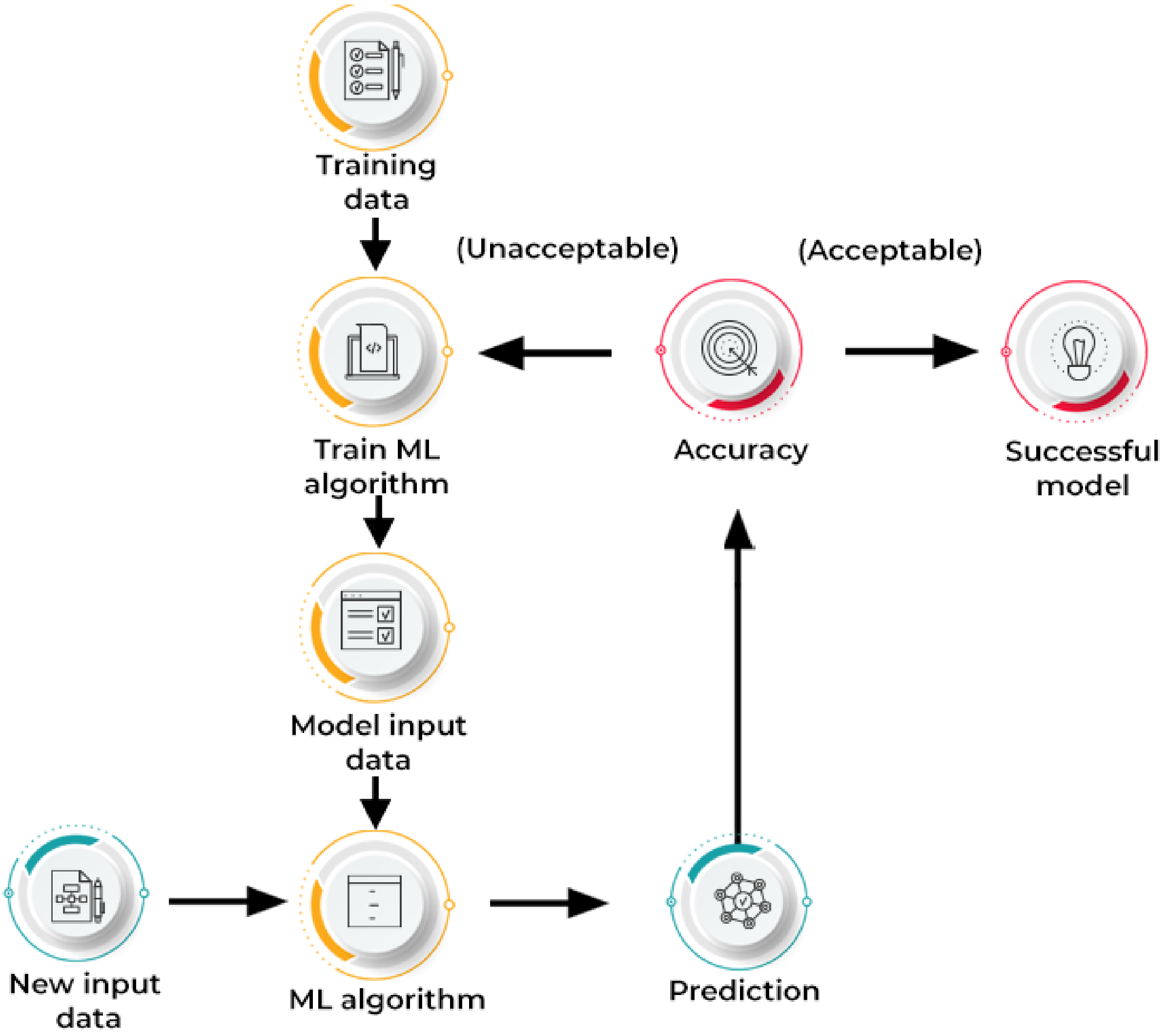


Regressor

$$f(x) = \frac{1}{1+e^{-\left(\frac{w_{21}}{1+e^{-(w_{11}x+b_{11})}} + \frac{w_{22}}{1+e^{-(w_{12}x+b_{12})}} + b2\right)}}$$



Classifier



Machine Learning Framework

Fixing NN training issues

- Exploding Gradients. Reject outcomes above a fixed threshold (Gradient Clipping). Regularization strategies can be used. Batch Normalization and Using Non-saturating Activation Functions.
- Datasets having target values on very different scales. Scale (z-scoring or StandardScaler and MinMax scaler) and normalize data.
- Datasets with very large outliers. Remove outliers and data cleaning.
- Loss is unable to get traction on the training dataset. Check for the designed minimization framework: choice of optimizer function, Batch Size, Learning rate, Loss...