

A mathematical introduction to Neural Networks and Neural Ordinary Differential Equations

Argimiro Arratia

argimiro@cs.upc.edu

<http://www.cs.upc.edu/~argimiro>

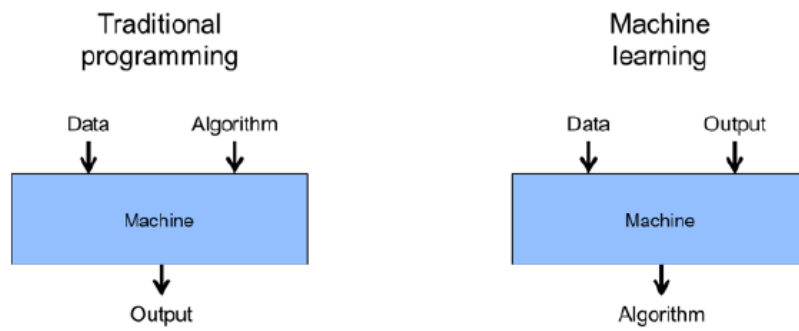
CS, Universitat Politècnica de Catalunya

Overview

- 1 Feed Forward Neural Networks. 1-layer and multi-layer. Optimization through Gradient Descent. Mathematical expressions for the gradient. General Nnet algorithm.
- 2 Pytorch Nnet programming
- 3 Deep Optimal Stopping problems (revisited with Pytorch code)
- 4 Universal Approximation Results for Neural Networks
- 5 Residual Neural Networks and other variants
- 6 Neural Ordinary Differential Equations. Adjoint Method. Numerical implementation. Errors in computation. Adaptive Adjoint. Pytorch implementation.
- 7 Neural Ordinary Differential Equations and universal systems. Applications.

Machine Learning Models

Machine Learning perspective



I. Feed Forward Neural Networks

Neural Networks (inspired by brain function)

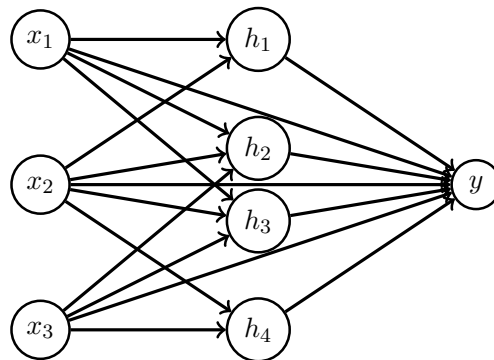


Figure: A 3-4-1 feed forward neural network with one hidden layer

x_1, x_2, x_3 input nodes; y output node;
 h_1, \dots, h_4 hidden nodes (neurons) in hidden layer (brain unit);
 h_j goes active and fires a signal to y if $z_j = \sum_{i \rightarrow j} \omega_{ij} x_i - b_j > 0$.
(Mathematically apply the activation function $\varphi(z) = \max(z, 0)$ to z_j)

Neural Networks: Single hidden layer case

We have d inputs $\mathbf{x} = (x_1, \dots, x_d)$, one (or many outputs), and one hidden layer with H_1 units. Set $H_0 = d$. For a single output:

$$\mathbf{F}(\mathbf{x}) = \varphi \left(\sum_{i=1}^{H_1} w_i^{[2]} a_i^{[1]} + b^{[2]} \right) \quad (1)$$

where






$$z_i^{[1]} = \sum_{j=1}^{H_0} w_{ij}^{[1]} x_j + b_i^{[1]}, \quad (2)$$

$$a_i^{[1]} = \varphi(z_i^{[1]}), \quad i = 1, 2, \dots, H_1 \quad (3)$$

$\varphi(\cdot)$ is a nonlinear activation function (e.g. $ReLU(x) = \max(0, x)$)¹. Think of $z^{[1]}$ and $a^{[1]}$ as the output at the hidden layer 1, before and after activation.

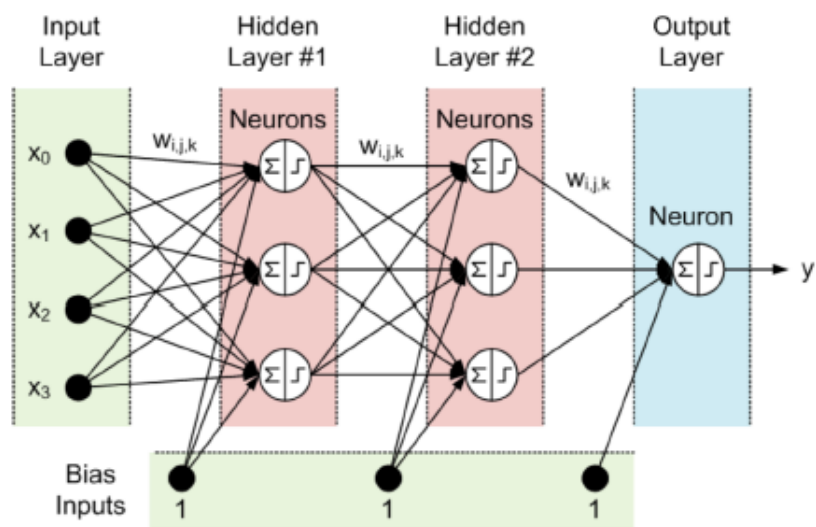
¹We can have different activation functions but for sake of simplicity we work with one

Nnet: Activation functions

Name	Visualization	$f(x) =$	Notes
Linear (= Identity)		x	Not useful for hidden layers
Heaviside Step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	Not differentiable
Rectified Linear (ReLU)		$\begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	Surprisingly useful in practice
Tanh		$\frac{2}{1+e^{-2x}} - 1$	A soft step function; ranges from -1 to 1
Logistic ('sigmoid')		$\frac{1}{1+e^{-x}}$	Another soft step function; ranges from 0 to 1

Deep Neural Networks

Deep Neural Networks (aka. multilayer neural networks)



Deep Neural Networks

In vectorial notation (2), (3) are expressed as

$$\mathbf{z}^{[1]} = W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}, \quad \mathbf{a}^{[1]} = \varphi(\mathbf{z}^{[1]}) \in \mathbb{R}^{H_1}$$

And the output of 1-layer Nnet (Eq. (1)) as

$$\mathbf{F}(\mathbf{x}) = \varphi(W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}) =: \mathbf{a}^{[2]}$$

where $\varphi(\mathbf{z}) = (\varphi(z_1), \dots, \varphi(z_{H_1}))$ is activation function

Deep Neural Networks

If there are $D > 1$ hidden layers, each labelled by $\mu = 1, \dots, D$, and with H_μ neurons in each, the recursion can be written as

$$\begin{aligned} \mathbf{a}^{[0]} &= \mathbf{x} \\ \mathbf{z}^{[\mu]} &= W^{[\mu]}\mathbf{a}^{[\mu-1]} + \mathbf{b}^{[\mu]} \\ \mathbf{a}^{[\mu]} &= \varphi(\mathbf{z}^{[\mu]}) \in \mathbb{R}^{H_\mu} \end{aligned}$$

And the final output is the vector

$$\mathbf{F}(\mathbf{x}) = \varphi(W^{[D+1]}\mathbf{a}^{[D]} + \mathbf{b}^{[D+1]}) =: \mathbf{a}^{[D+1]}$$

Forward evaluation (training)

consists of choosing weights and biases such that the output approaches the actual values associated to input

Nnet Backward propagation (tuning)

Let training data $\{(\mathbf{x}^{[i]}, \mathbf{y}^{[i]}) : i = 1, \dots, N\}$ of N inputs $\mathbf{x}^{[i]} \in \mathbb{R}^{H_0}$ and corresponding N outputs $\mathbf{y}^{[i]} \in \mathbb{R}^{H_D}$.

The parameters (e.g. weights and biases) are chosen so that some **error** measure is minimized (e.g. mean square error MSE).

In general we have cost (or loss) function \mathcal{C} on parameters θ and measure of error

$$Cost(\theta) = \frac{1}{N} \sum_{i=1}^N \mathcal{C}(\mathbf{y}^{[i]} - \mathbf{F}(\mathbf{x}^{[i]}))$$

(e.g. in the case of quadratic cost, the objective to be minimized is

$$Cost(\theta) = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} \|\mathbf{y}^{[i]} - \mathbf{F}(\mathbf{x}^{[i]})\|_2^2$$

Optimization through Gradient Descent

Expand the cost objective using Taylor series ($\theta \in \mathbb{R}^s$):

$$\begin{aligned} Cost(\theta + \Delta\theta) &\approx Cost(\theta) + \sum_{i=1}^s \frac{\partial Cost(\theta)}{\partial \theta_i} \Delta\theta_i \\ &= Cost(\theta) + \nabla Cost(\theta)^\top \Delta\theta \end{aligned}$$

where $\nabla Cost(\theta)$ is the gradient vector and need to choose $\Delta\theta$ so that $\nabla Cost(\theta)^\top \Delta\theta$ is most negative at each iteration. This is achieved by updating with small step size η :

$$\theta \rightarrow \theta - \eta \nabla Cost(\theta)$$

layer through layer (gradient descent)

Summary: Neural Network paradigm

- Forward evaluation (training)

$$\mathbf{F}(\mathbf{x}) = \psi(W^{[D+1]}\mathbf{a}^{[D]} + \mathbf{b}^{[D+1]})$$

with $z^{[\mu]} = W^{[\mu]}\mathbf{a}^{[\mu-1]} + \mathbf{b}^{[\mu]}$ and $\mathbf{a}^{[\mu]} = \varphi(\mathbf{z}^{[\mu]})$, $\mu = 1, \dots, D$.

- Measure of quality of approximation (Cost function)

$$Cost(\theta) = \frac{1}{N} \sum_{i=1}^N \mathcal{C}(\mathbf{y}^{[i]} - \mathbf{F}(\mathbf{x}^{[i]}))$$

- Backward propagation to improve approximation. By gradient descent update through layers

$$\theta \leftarrow \theta - \eta \nabla Cost(\theta)$$

Remark: The functions in $Cost$ are known and differentiable.

Neural Network training.
Formal details and algorithm

We derive a formula for the gradient. Consider a Deep Neural Network of depth $D + 2$ (D hidden layers and input -layer 0- and output -layer $D + 1$) with activation function φ , a dataset \mathcal{D} and MSE loss function. To drop the dependence of the loss function on the input, we consider a single training point $(x, y) \in \mathcal{D}$ and write

$$L = \frac{1}{2} \|y - a^{[D+1]}\|_2^2 \quad (4)$$

Let $\delta^{[\mu]} \in \mathbb{R}^{H_\mu}$ be defined by

$$\delta_j^{[\mu]} = \frac{\partial L}{\partial z_j^{[\mu]}} \quad j \in \{1, \dots, H_\mu\}, \quad \mu \in \{1, \dots, D + 1\} \quad (5)$$

which we call the *error* of the neurons at layer μ . Consider the Hadamard or element-wise vector product, \odot :
if $x, y \in \mathbb{R}^n$, $x \odot y \in \mathbb{R}^n$, and $(x \odot y)_i = x_i y_i$.

Formulae for the gradient

Lemma

$$\delta^{[D+1]} = \varphi'(z^{[D+1]}) \odot (a^{[D+1]} - y) \quad (6)$$

$$\delta^{[\mu]} = \varphi'(z^{[\mu]}) \odot (W^{[\mu+1]})^T \delta^{[\mu+1]} \quad \text{for } \mu \in \{1, \dots, D\} \quad (7)$$

$$\frac{\partial L}{\partial b_j^{[\mu]}} = \delta_j^{[\mu]} \quad \text{for } \mu \in \{1, \dots, D + 1\} \quad (8)$$

$$\frac{\partial L}{\partial \omega_{jk}^{[\mu]}} = \delta_j^{[\mu]} a_k^{[\mu-1]} \quad \text{for } \mu \in \{1, \dots, D + 1\} \quad (9)$$

The output $a^{[D+1]}$ is evaluated from a *forward pass* through the network, starting at $a^{[0]}$ and computing $z^{[1]}, a^{[1]}, z^{[2]}, a^{[2]}, \dots, a^{[D+1]}$. Formulas 8-9 show that to compute the gradient we need the sequence $\{\delta^{[\mu]}\}$ and from 6-7 we see that after a forward evaluation and y we can get $\delta^{[D+1]}$, and then $\delta^{[D]}, \delta^{[D-1]}, \dots, \delta^{[1]}$, in a process known as *backward pass*

Algorithm 1: Training of a Neural Network

Input: neural network object of depth $D + 2$ with :
weights $W^{[\mu]}$ and biases $b^{[\mu]}$, $\mu \in \{1, \dots, D + 1\}$
activation function φ and its derivative φ'
data points $\mathcal{D} = \{(\mathbf{x}^{[k]}, \mathbf{y}^{[k]}) \in \mathbb{R}^n \times \mathbb{R}^m\}_{k \in \{1, \dots, N\}}$
number of epochs $Niter$; learning rate η

```
for epoch = 1 to Niter do
  Choose an integer  $k$  uniformly at random from  $\{1, 2, 3, \dots, N\}$ ;
   $a^{[0]} \leftarrow x^{[k]}$ ;
  for  $l = 1$  upto  $D + 1$  do
     $z^{[l]} \leftarrow W^{[l]}a^{[l-1]} + b^{[l]}$ ;
     $a^{[l]} \leftarrow \varphi(z^{[l]})$ ;
  end
   $\delta^{[D+1]} \leftarrow \varphi'(z^{[D+1]}) \odot (a^{[D+1]} - y^{[k]})$ ;
  for  $l = D$  downto 1 do
     $\delta^{[l]} \leftarrow \varphi'(z^{[l]}) \odot (W^{[l+1]})^T \delta^{[l+1]}$ ;
  end
  for  $l = D + 1$  downto 1 do
     $W^{[l]} \leftarrow W^{[l]} - \eta \delta^{[l]} a^{[l-1]T}$ ;
     $b^{[l]} \leftarrow b^{[l]} - \eta \delta^{[l]}$ ;
  end
end
```

Forecasting Time Series with NNet

Practical issues

Model Selection

The performance of the NNet -based forecasters depends on the choice of the free parameters.

These are,

- **NNets**: the size of hidden layer; the thresholds for activating hidden nodes; connection bias; the weights for inputs; decay factor (or learning rate).

Adapting the parameters is referred to as model selection.

R packages for Nnet: `nnet`: considers only one layer;

`neuralnet` is multilayer;

in `caret` + `RSNNS` has `mlp` a multilayer perceptron; `keras`

Python: `sklearn.neural_network`; `pytorch`; `tensorflow`

Model training and testing

Let $\{(x_t, r_t) : t = 1, \dots, T\}$ be the available data

r_t is the return of some financial asset (but we could target PRICE also) and x_t is vector of inputs or **features**²:

- lags of the series (its past behavior);
- volume,
- variance (volatility)
- any fundamental indicator of the series (e.g. Price-to-Earnings, Dividend-to-Price)

Model fitting for a NNet requires division of the data into

TRAINING ($\pm 75\%$) **TESTING** ($\pm 25\%$)

²One assumes inputs lie in some feature space, which one not need to know

Training

In this step build a few models by choosing the parameters (e.g., weights, the thresholds and connection bias) so that some forecasting error measure is minimized

For NNet: use the mean squared error

$$MSE(\mathbf{w}) = \frac{1}{N} \sum_{t=1}^N (r_t - modelFit(\mathbf{w}, \mathbf{x}_t))^2$$

Testing

The best fitted model build in training step is tested on the subsample of data reserved for testing to predict some values and compare estimations with actual sample values.

Usual measures of forecasting accuracy

$$MSE = \frac{1}{N} \sum_{t=1}^N (r_t - pred(r_t))^2, \quad MAE = \frac{1}{N} \sum_{t=1}^N |r_t - pred(r_t)|,$$

$$RMSE = \sqrt{MSE}$$

But some experts recommend to use

Normalized RMSE

$$NRMSE = \frac{\sqrt{MSE}}{\sqrt{\frac{1}{N} \sum_t (r_t - \hat{\mu}_r)^2}} = \sqrt{\frac{SE}{(N-1)Var(r_t)}}$$