**Classification and Regression, from linear and logistic regression to neural networks**

Table of Contents

[1. Introduction 1](#_Toc119312583)

[2. Data and Methods 1](#_Toc119312584)

[2.1. Theory 1](#_Toc119312585)

[2.1.1. Gradient methods 1](#_Toc119312586)

[2.1.2. Feeed-Forward Neural Networks (FFNN) 3](#_Toc119312587)

[2.2. Implementation in python 7](#_Toc119312588)

[2.3. Data 7](#_Toc119312589)

# Introduction

Simple models for regression (like those used in Project 1: Ordinary-Least-Squares (OLS), Ridge and Lasso); or classification can perform well in many cases where the target can be expressed as a linear combination of those functions. But their applicability is limited when the target behaves in a nonlinear way. Deep learning methods aims to extract linear combination of the inputs and model the target as a nonlinear function of them. Neural networks have become a popular method to solve problems within the supervised learning framework, with widespread applications in many fields.

In this project, we assess the performance of a Feed Forward Neural Network for both regression and classification problems, as an extension of the work done in Project 1. First, we use synthetic dataset generated by the Franke Function (Franke, 1979) to assess the performance on regression. In the second part, we use the Wisconsin Breast Cancer dataset to assess a classification problem. The structure of the Neural Network remains the same for both applications, just changes in the chosen activation functions are needed. Different activation functions and optimization techniques are discussed and tested in both parts

# Data and Methods

## Theory

In this work, we use classification and regression techniques by developing a feed-forward Neural Network (FFNN). In this network, the information moves forward through the layers.

### Gradient methods

* Gradient descent

Gradient descent is an optimization method, used to find the minima of the cost function. Gradient is defined as the derivative of a multivariable function, in our case the cost function. The main idea of the gradient descent is that a convex function decreases fastest when going from in the direction of the negarive gradient .

is the learning rate or step-size we use for moving in the direction of the gradient.

Taking steps iteratively in this direction the minima will be found if it exists. One common problem in these methods is to find local minima instead of global minima in non-convex functions. The simplest approach is to take a step size (or learning rate ).

In practice, the Gradient descent method is very sensitive to initial conditions (i.e. starting point for the iterations) and choice of learning rates. It is as well a method computationally expensive, especially for large datasets, since it computes the cost function and its gradient for all data points.

The performance can be accelerated by introducing momentum term , where the previous steps are remembered. This allows to build inertia in a direction in the search space and accelerate the search specially through noisy gradients or flat spots.

Linear regression has several desirable properties for calculating gradient descent methods: it has an analytical solution; the gradient can be also computed analytically, and the cost function is convex.

* Stochastic Gradient Descent

Stochastic Gradient Descent deals with the sensitivities of Gradient descent to the initial conditions, choice of learning rate or convexity of the cost function. It chooses a random instance in the training set at every step and computes the gradients based on only one instance (Geron). This algorithm is less regular than the Gradient Descent because of its stochastic nature. But it can help avoid the local minima, although it can make it harder to converge to a minimum. To solve this, it is common to use an adaptative learning rate, where the steps taken are larger at start, and get smaller as the algorithm approach the global minimum.

In addition, to further accelerate the computations, the gradient is computed over so called mini-batches of the training data, instead of computing the cost function and gradient for all points of the training dataset. This is especially useful with large training datasets

* Algorithms with adaptative learning rate

The learning rate is one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance. The momentum algorithm can mitigate these issues in part, but at the cost of introducing another hyperparameter (Goodfellow). Recently there has been developed different methods that can adaptively change the step size to match the search space without having to calculate or approximate the Hessian, that represents the second derivative of the cost function. They accomplished this by tracking the gradient plus the second moment of the gradient.

AdaGrad: short for Adaptative Gradients (Duchi et al., 2011), is an extension of the gradient descent. It uses an aggregate of the squares of previously observed gradients, and scales down the gradient vector along the steepest dimension. In practice, its significantly decrease learning rates on large gradient parameters, while the decrease is smaller on parameters with smaller gradients. Adagrad converged rapidly when we use it in convex functions, but it is not ideal with non-convex functions.

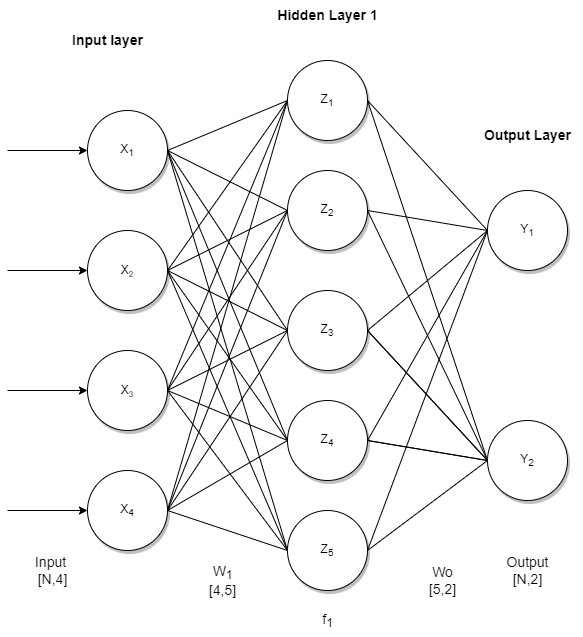
RMSProp: it accumulates the exponentially weighted moving average of just the most recent iterations, as opposed to the gradient accumulation since the beginning of training in Adagrad. This implies a better performance in non-convex functions, such as those typically used in machine learning and neural networks. RMSProp is widely used in practical optimization algorithms for neural networks (Goodfellow).

Adam: short from adaptive moment estimation, (Kingma and Ba, 2014) it combines properties from both RMSProp and gradient descent with momentum optimization. ADAM keeps track of the running average of the first and the second moment of the gradient and consequently adapt the learning rate for different parameters. In addition, ADAM includes a bias correction to account for the running average of the first and second moments.

### Feeed-Forward Neural Networks (FFNN)

Neural Networks are nonlinear, statistical models, that can learn to approximate some regression function or classification model through considering examples, i.e they are applied to both regression and classification. They are designed to mimic a human brain, in which an arbitrary number of neurons within a layer send signals to other neurons in a different layer. Each neuron accumulates its incoming signals, and they must exceed an activation threshold to yield an output. Each connection is represented by a weight variable (Hjortn-Jensen) plus some bias. They are called *feedforward* when the information flows just in one direction through the model, from the input, through the layers and produces an output. They are called *networks* because they can be composed by many different functions.

In Figure \_\_ is represented a network diagram of a multilayer perceptron (MLP), which is a fully connected feed-forward neural network with an input layer with four neurons (), one hidden layer (it could be more) with 5 neurons and an output layer with 2 neurons (). Each circle represents neurons, the weight parameters are represented by links between the neurons. Activation functions presented on the hidden layer (). For classification problems, there is often an output function on the output layer. For regression there is typically only one output, and the output function would often be the identity function. The length of the network gives the depth of the model, while the dimensionality of the hidden layers determines the width.



*Figure \_\_. Diagram of an example of single hidden layer, feed-forward neural network.*

The performance of the neural network is as follows. Each input is multiplied by a weight , that indicate the strength of the connection to each node in the next layer. To avoid outputs of only zeros, a small bias is added to each weight. These weights are usually initialized with random small numbers distributed around zero. The result is analyzed by an activation function (), and the result is the output for each node, which at the same time becomes the input for the next layer

The training data shows the behavior of the output layer: the output data should be close to . But it doesn’t specify the behavior of the hidden layers, the learning algorithm decides how to use them to end producing the desired output (that is the reason for calling them hidden layers).

* Activation functions

The choice of activation layers can have big influence on the performance of the neural network. In a FFNN the activation functions are required to be non-constant, bounded, monotonically increasing and continuous (Hjortn-Jensen) for the network to work well in complex datasets. Common choices include sigmoid function, the Rectified Linear Unit function (ReLU), the hyperbolic tangent function (than) or the Leaky ReLU.

Sigmoid function: The Sigmoid (or Logistic) function is the preferred one for classification problems with binary outcome, which aims to predict outputs in form of discrete variables (yes/no, true/false etc.). The minimization on the cost function leads in this case to a non-linear equation that can’t be solve analytically. The optimization of logistic regression is then done by minimization algorithms, like gradient descent methods.

Considering the case where the response variables are discrete and only take values from two classes , the sigmoid function gives the probability that a data point belongs to one of the categories:

The sigmoid function gives outputs in the range , which it’s a good fit to classify a binary problem. In the case of having more than two classes (or categories), the softmax function is commonly used. This function also gives outputs in between 0 and 1, with the particularity that all the outputs sum 1, so they can be interpreted as probabilities that the output is in a particular class.

To measure the performance of the classification network, we apply an accuracy score to some data that has not been part of the training dataset. This term is simply the number of outputs classified correctly divided by the total. The expression is as follows:

Where is the indicator function, 1 if , and 0 otherwise.

Softmax function: The Softmax function, also called Multinomial Logistic Regression, can be used for outputs with multiple classes. It produces positive probabilities that sum to one:

ReLU function: This function only activates when the input variable that goes through it is positive:

ReLU works well in deep neural networks and it’s quite fast to compute. However, it suffers from a problem known as dying ReLU. This is due to some neurons stop outputting anything other than 0. To solve this problem, there is a variant of this method called Leaky ReLU.

Leaky ReLU: This form of ReLU has a small positive gradient for negative values:

Than: The hyperbolic tangent activation function is related to the sigmoid. The difference is the range of outputs is from (-1 to 1). The main advantage with the sigmoid is that tanh is faster.

* Output and backpropagation

After one feed-forward pass through the network (i.e., one iteration), we need a mechanism that adjust the weights and biases that connect the neurons in the network, to minimize the errors in the output. After performing the feed-forward, the output error (last layer ) is given by

where is the derivative of the activation function of the last neuron, is the output data of the last neuron and is the cost function.

The back-propagate error can be computed for each layer as

Weights and biases can now be updated using gradient descent for each according to

Where is the learning rate

## Implementation in python

All the algorithms were implemented in python 3.8. The packages we used were numpy, seaborn, scikit-learn and tensorflow (keras). To have a full understanding of all processes described in 2.1, we wrote our own neural network code for both regression and classification problem, in form of a python class. We have interpreted the network as an object using a python class. Different classes were created for all variants tested as an effort for documenting all the process and quality check the results. However, we could have used the same code and replacing in each run the activation functions.

* + 1. Benchmarks

The results from our neural network classes were compared with ready functionalities from Scikit-Learn and Keras. Our regression problem, using the Franke function data was benchmarked against Scikit-Learn’s *MLPRegressor* and tensorflow/keras. In the case of our classification problem, we compared our results with scikit-Learn’s *MLPClassifier* and with tensorflow/keras.

## Data

* + 1. Regression analysis

The dataset used to study regression problems is generated using the Franke function. The Franke function is a function of two variables where and is expressed as follows:

The Franke Function is depicted in Fig. \_\_ for two cases, with and without normally distributed noise. We use a normally distributed noise with a variance of 0.1.

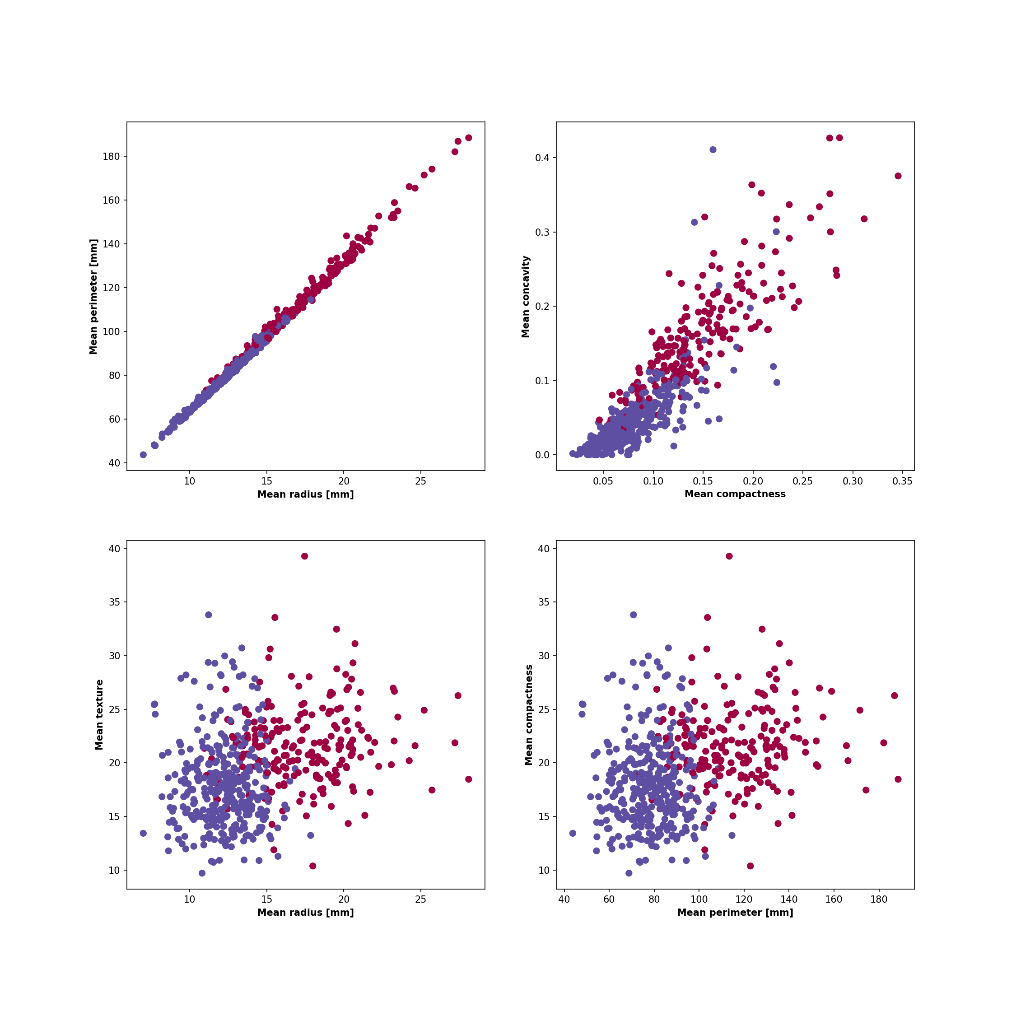
Chart, surface chart

Description automatically generated

**Fig. \_\_** 3D visualization of the Franke Function without added noise (left panel) and with normally distributed noise with variance 0.1 and mean 0 (right panel).

* + 1. Classification analysis

The dataset used for classification problem is the Wisconsin Breast Cancer dataset, and it was uploaded directly in our python script through the scikit-learn package. The dataset has a series of input variables extracted from patients with solid breast masses, such as mean radius, mean perimeter, mean texture, or mean compactness. Figure \_\_ visualize the correlation in between some of these variables.



**Fig. \_\_**  visualization of the correlation in between 4 relevant variables from the Wisconsin Breast Cancer dataset

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online

learning and stochastic optimization. *Journal of Machine Learning Research*. 307

Kingma, D. P., & Welling, M. (2014). Auto-Encoding Variational Bayes. International Conference on Learning Representations (ICLR).