Regression Analysis and Neural Networks

Alessio Quercia

Abstract—In this paper I show the results of different regression problems dealt with using neural networks as training models.

Several experiments were made to test up regression problems with neural networks, such as using appropriate preprocessing techniques, varying the networks' topologies (and so their parameters) and the activation functions used by their layers, testing different training algorithms and loss functions. A validation loop over some validation epochs was used to obtain a good training algorithm's parameter (learning rate) for the subsequent training loop, and external cross-validation was used to obtain the model's accuracy as the mean of the different test errors computed as mean absolute error.

1 Introduction

Regression problems are supervised learning problems in which a model is asked to predict a numerical (real) value given some input of the form $sample_i = (x_i, y_{t_i})$, where $sample_i$ is a single input, $x_i \in \mathbb{R}^n$ is a vector of features and $y_{t_i} \in \mathbb{R}$ is the target, i.e. the desired output for that vector of features. The goal of a machine learning algorithm for regression problems is to build a system that is able to predict a value that is as close as possible to the target, given an input sample.

There are many ways to approach a regression problem. Linear regression, for example, is a parametric model whose output is a linear function of the inputs. A parametric model is a model that learns a function described by a parameter vector (the model's weights) whose size is finite and fixed before any data is observed [1]. Indeed, in a parametric model the learning phase is done by optimizing the model's parameters.

Another interesting approach to a regression problem is the logistic regression (also a parametric model), which consists in estimating the probability of a categorical outcome. This means that we are considering only a single outcome variable with two possible states, 1 or 0, a binary variable. Thus, the desired output for each input is 1 or 0, and logistic regression estimates the probability that a certain input belongs to 1 (or 0), resulting in being a generalization of linear regression to the classification scenario. Indeed, logistic regression is often used to solve binary classification problems.

Many other regression models exist in literature: robust regression, non-linear regression, least squares regression and ridge regression are just few examples.

Alessio Quercia, Statistical Methods For Machine Learning, A/A 2017-2018, Università degli Studi di Milano, via Celoria 18, Milano, Italy
E-mail: alessio.quercia@studenti.unimi.it

Artificial Neural Networks (ANNs) are learning models inspired by the biological brain, but, even though they are sometimes used to understand brain function, they are generally not designed to be realistic models of biological function [1].

Feedforwad neural networks (and their corresponding deep version) are the most common example of neural networks; they are widely used as models to solve simple classification and regression tasks.

The goal of a feedforward neural network is to approximate some function f^*, by defining a mapping from the input to the output $y = (x; \theta)$ and by learning the value of the parameters θ that result in the best function approximation.

Other kinds of neural networks exists, such as recurrent neural networks, which allows feedback connections, convolutional neural networks, that are deep feedforward neural networks inspired by the animal visual cortex, typically used in image classification problems, or Long Short-Term Memory neural networks (often called LSTM networks), recurrent neural networks composed by LSTM units that are typically used in natural language processing problems.

As a conclusion to this introduction, I would like to briefly introduce the method I used to make the experiments. It may be schematized into five main phases:

1) Preprocessing: processing the data before feeding them to the model.
2) Validation: optimizing the optimizer's parameter (learning rate) on the validation set.
3) Training: optimizing the model's weights (using the optimized parameter for the optimizer) on the training set.
4) Testing: testing the model on the test set.
5) Results Analysis: confronting the desired and predicted outputs.

The purpose of this paper is to show the results obtained by using different datasets as input to neural networks models built up to solve regression problems. I tested both the linear regression and the logistic regression approachs, depending on the dataset. I used TensorFlow's low level API to implement the models. For each test I used the cross-validation technique to compute the accuracy of the model in terms of mean error, repeating the above mentioned method on five different couples training-test sets.

2 Background

The following two subsections briefly describe what regression analysis and neural networks are and how they can be combined to make predictions.

2.1 Regression Analysis

Regression analysis is a set of statistical processes for estimating the relationships among variables. Typically the focus is on the relationship between a dependent variable (Y) and one or more independent variables (X). Most commonly, regression analysis estimates the conditional expectation of the dependent variable, given the independent variables, that is, the average value of the dependent variable when the independent variables are fixed. Regression refers specifically to the estimation of continuous response (dependent) variables, that are a function of the independent variables (the regression function). While dealing with a regression problem, we want to build a system that is able to predict a numerical (real) value y (the value for our dependent variable) given some input x (values for our independent variables). Therefore we want to estimate a probability distribution $p(y|x)$, and we can do this by using maximum likelihood estimation to find the best parameter vector θ for a parametric family of distribution $p(y|x;\theta)$.

In the case of linear regression, we build our model in such a way that its output is a linear function of the input. To do so, we simply multiply each input for its respective weight (parameters of the model) and then sum up the results and, eventually, a bias term (which transform the linear function into an affine function, meaning that the plot of model's predictions still looks like a line, but it need not pass through the origin [1]). Thus, to design a good linear regression algorithm, we need to build an algorithm that will improve the initial values of the weights of our model. We want to optimize our model's parameters in order to predict values whose error is as small as possible. This means that we can minimize the error to optimize the weights. Indeed, an optimization algorithm is used to minimize the loss function (the function that computes the errors) during the model's training phase. The most used optimization algorithm is the Gradient Descent, which updates the weights by adding a value that is proportional to the negative of the gradient of the error; while the most used loss function is the Mean Squared Error (MSE), which measures the average of the squares of the errors, that is, the average squared difference between the expected values and the estimated values. Linear regression corresponds to the family of distribution

$$p(y|x;\theta) = \mathcal{N}(y;\theta^\top x, I) \tag{1}$$

that is a Normal (or Gaussian) distribution.

While using a logistic regression model, the output must be a (real) value between 0 and 1, a probability value. For this reason, we can't simply use a linear regression model, because its output will be a continuous value in $\mathbb{R}$. One way to solve this problem is to squash the output of the linear function into the interval $(0,1)$ by using the logistic sigmoid function and interpret the value as a probability:

$$p(y=1|x;\theta) = \sigma(\theta^\top x) \tag{2}$$

In logistic regression there is no closed-form solution for the optimal weights of the model. For this reason, we must search for them by maximizing the log-likelihood, which is the same as minimizing the negative log-likelihood (e.g. by using the Gradient Descent minimization method).

2.2 Neural Networks

Neural Networks are models composed by layers of units and connections between units. Though, they can be represented as graphs. To solve non-linearly separable tasks, at least an input, a hidden and an output layers are needed, where the input layer's units are the inputs themselves.

Feedforward neural networks are called feedforward because the information flows in a unique way from the input to the output. Indeed, these models have no feedback connections that allow a output to be used as input to a preceeding unit. Besides, they are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. Usually these functions are connected in a chain to form $f(x) = f^{(n)}(f^{(n-1)}(...f^{(2)}(f^{(1)})...))$, where $f^{(1)}$ is the first layer of the network (the input layer), $f^{(2)}$ the second, and so on until the last layer, which corresponds to the output layer [1]. Each unit of the network (except for the input units, that corresponds to the inputs themselves) compute its output as a weighted sum of its inputs (given by incoming connections) and the respective weights associated with them:

$$o_j = \sum_{i=1}^{n} w_i u_i \tag{3}$$

where o_j is the j-th unit's output, u_i is the i-th incoming input (if the unit j is in the first hidden layer, that u_i corresponds to the input x_i) and w_i is the weight on the connection between u_i and j. During the training, we drive $f(x)$ to match $f^*(x)$, that should be a close value to the label associated to the input. The learning algorithm must decide how to use each layer to produce the desired output (the label), by assigning each layer an activation function, which is used on the layer's outputs to map the outputs into a specific interval. Because the training data does not show the desired output for the intermediate layers, these layers are called hidden and their activation functions may be different.

To use a linear or a logistic regression algorithm on a neural network model, the activation function of the output layer must be respectively a linear or a logistic activation function. This last one, also called sigmoid function (or curve) is used to squash the output between 0 and 1 as follows:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4}$$

Since neural networks models have a parameters' vector with fixed size, there is the problem of deciding the neural network structure (or topology). Indeed, there is no "correct" size for a neural network's model to solve a given problem, but may be that certain structures solve the problem in a smaller amount of time. However, there are some best practices on how to build a performing neural network model, such as the one that suggests building a model with a single hidden layer with a number of hidden units corresponding to the mean of the input and the output units. Furthermore, it seems that adding a new hidden layer with a small number of hidden units may perform better than adding a big number of units to the same hidden layer. Consequently, this imply that deep neural networks may perform better than simple neural networks. For a better precision, cross-validation can be used to estimate the "best" (or good enough) model's topology for a given problem, that is the number of hidden layers and the number of units per hidden layer that had the best performance on the validation set. In particular, I did not use this last technique due to its computational time cost; instead, I tried different fixed model's topologies.

3 Theoretical model

3.1 Datasets

I used different data sets to test both the linear and the logistic regression, available at UCI Machine Learning Repository [2].

I began testing linear regression on the Wine Quality Data Set, that has 12 different attributes, one of which could be used as an output variable, the 'quality' attribute. The other 11 variables were used as input variables to give the model enough information to predict the wine quality value (a value between 0 and 10) [3]. The Wine Quality Data Set variables are showed in the Table 1. All the attributes are real numbers, even though the quality values are cut to the integer value (to make the dataset available to classification problems too). Only the quality interval is given, the other attributes have no specified interval.

Another data set I used for linear regression is the Boston Housing Data Set, presenting 14 attributes, 13 inputs and 1 output. The input variables were used to predict the median value of owner-occupied homes in \$1000's, abbreviated as MEDV. The Boston Housing Data Set variables are showed in the Table 2. Every attribute is a real number, except for the CHAS attribute, which is a binary-valued attribute.

Lastly, I used the Wisconsin Diagnostic Breast Cancer Data Set to test logistic regression. Indeed this data set is composed by 34 attributes and the outcome is a binary value, representing whether the Breast Cancer is malignant (M) or benign (B). The first two attributes are the patient's id number and the diagnosis' outcome (M or B), the consecutive ones are ten real-valued features computed for each cell nucleus (30 attributes), resulting in 32 total attributes. The attributes are shown in the Table 3.

Attribute	Description
Fixed acidity	Value in $\mathbb{R}$
Volatile acidity	Value in $\mathbb{R}$
Citric acid	Value in $\mathbb{R}$
Residual sugar	Value in $\mathbb{R}$
Chlorides	Value in $\mathbb{R}$
Free sulfur dioxide	Value in $\mathbb{R}$
Total sulfur dioxide	Value in $\mathbb{R}$
Density	Value in $\mathbb{R}$
pH	Value in $\mathbb{R}$
Sulphates	Value in $\mathbb{R}$
Alcohol	Value in $\mathbb{R}$
Quality	Value in [0,10]

TABLE 1: Wine Quality Data Set.

Attribute	Description
CRIM	Per capita crime rate by town
ZN	Proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS	Proportion of non-retail business acres per town
CHAS	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX	Nitric oxides concentration (parts per 10 million)
RM	Average number of rooms per dwelling
AGE	Proportion of owner-occupied units built prior to 1940
DIS	Weighted distances to five Boston employment centres
RAD	Index of accessibility to radial highways
TAX	Full-value property-tax rate per \$10,000
PTRATIO	Pupil-teacher ratio by town
B	$1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
LSTAT	Percentage (%) of the lower status of the population
MEDV	Median value of owner-occupied homes in \$1000's

TABLE 2: Boston Housing Data Set.

3.2 Preprocessing

Each dataset is split into two disjoint sets, training set (80%) and test set (20%). Before analysing the data by feeding them as input to a neural network, it is common to normalize them. Data normalization can be easily done when the values of every attribute belong to the same known interval, by subtracting every attribute the minimum value of the interval and then dividing the result by the difference between the interval's maximum and minimum values. If we consider an attribute to be a vector of values that belong to the interval [min, max],

Attribute	Description
ID number	Integer value
Diagnosis	(M = malignant, B = benign)
Radius	Mean of distances from center to points on the perimeter
Texture	Standard deviation of gray-scale values
Perimeter	Value in $\mathbb{R}$
Area	Value in $\mathbb{R}$
Smoothness	Local variation in radius lengths
Compactness	$Perimeter^2/Area - 1.0$
Concavity	Severity of concave portions of the contour
Concave points	Number of concave portions of the contour
Symmetry	Value in $\mathbb{R}$
Fractal dimension	"$Coastline\ approximation$" $- 1$

TABLE 3: Wisconsin Diagnostic Breast Cancer Data Set.

the normalized attribute can be computed as follows:

$$attribute_{normalized} = \frac{attribute - min}{max - min} \tag{5}$$

Often the datasets are composed by multivariate data and each variable has its own interval, that may be unknown. A normalization technique can be used to normalize an attribute without knowing its interval:

$$attribute_{normalized} = (attribute - mean)/std \tag{6}$$

where the $mean$ and the std (standard deviation) are computed over all the values of that attribute in the considered dataset.

The datasets I used have attributes with unknown intervals. For this reason I used mostly the second technique, applying it both on the training and the test set separately.

3.3 Validating, Training and Testing

Once the training and the test sets are normalized, they can be fed to the neural network model, respectively during the training phase and the test phase.

Before describing the validation, training and testing phases in detail, it is worth to describe the neural network model itself, that is the core of the regression algorithm used to solve the regression problems specified by the different datasets.

I built up a fully connected feedforward neural network as a general model for regression analysis. As already mentioned, the neural network topology choise is still an open problem, therefore there is not an a-priori "perfect" model topology. It has been demonstrated that a feedforward neural network with a single hidden layer containing a finite number of units is able to approximate any continuous function on compact subsets of $\mathbb{R}^n$ (*Universal approximation theorem*, Cybenko [4]). This does not imply that a model with a single hidden layer will approximate a function f faster than a model with more hidden layers (deep neural network). Indeed, it happens that deep neural network models perform better than single hidden layer ones. For this reason, I built a function that allows to build a model with an arbitrary number of hidden layers as well as an arbitrary number of hidden units for each one of them, and tested several different network topologies.

If we consider W_i to be the i-th weights matrix, that is the weights associated to the incoming connections to the units of the i-th layer (from the output of the $(i-1)$-th units layer), and b_i to be the i-th bias vector, containing the weights associated to the connections between the $(i-1)$-th bias and the i-th units layer, then, the output of the i-th units layer can be computed as follows:

$$o_i = W_i u_i + b_i \tag{7}$$

where u_i and o_i are respectively the input and the output vectors (or matrices U_i and O_i if the input contains multiples samples) of the i-th units layer. Then, an activation function a is applied to the i-th output layer, obtaining

$$o_i = a(o_i) \tag{8}$$

that is the i-th activated output vector.

A different activation function can be chosen for each hidden layer. Some of the most common activation functions used for regression analysis are showed in the Table 4. When using a linear regression model the output must be a linear function of the input, therefore the Identity function is usually used as activation function for the output layer; while dealing with logistic regression, the Logistic function is used instead. Besides, in this last case, an appropriate loss function need to be used, as already mentioned. Indeed, while in the linear regression's case *Mean Squared Error* (MSE) and *Mean Absolute Error* (MAE) are commonly used loss function, they can't be used in logistic regression, where the optimal parameters are estimated by maximizing the log-likelihood, that is by minimizing the negative of the Logaritmic loss. The mentioned loss function are shown in the Table 5.

Now that the model is introduced, the different phases of the regression analysis can be properly described. If we define as *optimizer* an optimization algorithm (e.g. the Gradient Descent optimizer), that is an algorithm that tries to minimize a given function, and *loss function* l (or *cost function*, e.g. the MSE function) as a function that maps values of one or more variables onto a real number representing their loss (or cost), we can use an optimizer to minimize the loss between the desired output and the model's predicted output. The Gradient Descent optimizer, as an example, at each training step:

1) Propagates forward through the network to generate the output value(s) (output calculation)

$$o_i = a(W_i u_i + b_i) \tag{9}$$

2) Calculates the loss (error calculation)

$$l(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{10}$$

where n is the number of output units, $\hat{y}_i$ is the predicted value for the i-th unit of the output layer, y_i is the target (or desired output) for the i-th unit and the considered loss function is the MSE function.

3) Propagates the output activations back through the network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons (backpropagation)

$$\delta_j = \begin{cases} (\hat{y}_j - y_j)\hat{y}_j(1-\hat{y}_j) \text{ if } j \text{ is an output unit,} \\ (\sum_{i \in I} w_{ji}\delta_i)\hat{y}_j(1-\hat{y}_j) \text{ if } j \text{ is an inner unit} \end{cases} \quad (11)$$

4) Multiplies the weight's output delta and input activation to find the gradient of the weight (weights gradients calculation)

$$\frac{\partial l(\hat{y}, y)}{\partial w_{ij}} = \delta_j \hat{y}_i \quad (12)$$

5) Subtracts a ratio (percentage defined by the learning rate parameter η) of the weight's gradient from the weight (weights update)

$$\Delta w_{ij} = -\eta \frac{\partial l(\hat{y}, y)}{\partial w_{ij}} \quad (13)$$

After having normalized the training and the test sets, a subset of the training set is used to test the predictors produced by the training of the model on the remaining (disjoint) subset of the training set using a certain value as learning rate (the optimizer's parameter). Repeating this for a certain number (validation epochs) of times in a loop that we can call Validation Phase, we can obtain the best value for the learning rate by choosing the learning rate's value that produced the predictor with the lowest validation error, that is the lowest error on the validation set.

Once a good value for the optimizer's parameter is obtained, the Training Phase starts by feeding the model (with random initialized parameters) the training set. During this phase, the optimizer tries to minimize the training error, that is the error on the training set.

When we've done enough training, e.g. when the training error reached a good threshold or just when the model has been trained for a certain number of epochs (training epochs), we can test the model by feeding it the test set (Testing Phase). In this way we can compute the test error, that is the error on inputs that have not been used to train the model, by comparing the model's predictions and the respective desired outputs.

4 Simulations and experiments

I used *cross-validation* to compute the prediction accuracy of different predictors, obtained by training the model on different subsets of the data set. Indeed I split the training set into 5 portions, each one containing 20% of the dataset's data, and I used the *leave-one-out* technique as a cross-validation method. Therefore I iterated for 5 times the whole validation, training and testing phases by changing each time the four portions of datasets composing the training set (resulting in being 80% of the dataset) and using each time a different portion (subset) as test set (20% of the dataset). At the end of the cross-validation loop, it was possible to compute the accuracy of the model, by computing the mean test error as mean of the test errors produced by the different predictors obtained during the loop.

Name	Equation	Derivative
Identity	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$
Rectifier	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$
Softplus	$f(x) = \ln(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

TABLE 4: Activation Functions.

Name	Equation
Mean Absolute Error	$l(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n} \lvert\hat{y}_i - y_i\rvert$
Mean Squared Error	$l(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2$
Logaritmic Loss	$l(\hat{y}, y) = -\frac{1}{n}\sum_{i=1}^{n} y\ln(\hat{y}) + (1-y)\ln(1-\hat{y})$

TABLE 5: Loss Functions.

4.1 System architecture

I ran the simulations on my laptop, characterized by an Intel Core i7-6700HQ CPU, a Nvidia Geforce GTX 950M GPU and 8 GB RAM. The implemented algorithm uses both CPU and GPU.

4.2 Implementation details

The implementation is based on the general method scheme represented in the Figure 1.

The implemented algorithm also takes in consideration the *cross-validation* as shown in the pseudo-code in the Algorithm 1.

I implemented the algorithm in python, importing and using the *Tensorflow* library. Different optimizers (and optimizers parameters' values), loss functions, activation functions, neural network topologies were tested in several experiments. I also imported the *MinMaxScaler* method from *sklearn.preprocessing* to scale each dataset's feature such that it is in the range $(0, 1)$, i.e. between 0 and 1. I tried training and testing the models on normalized labels too, but most of the times the results were worse than those obtained on pure labels.

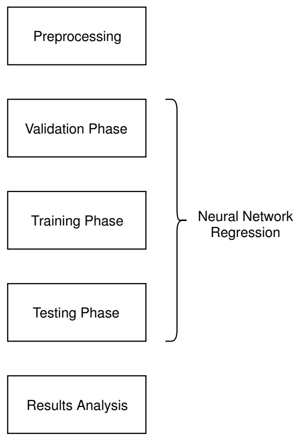

Fig. 1: Method scheme.

Algorithm 1 Neural Network Regression Pseudo Code

```
loss = CHOSEN_LOSS
learning_rate = 0.0001
optimizer = CHOSEN_OPTIMIZER(learning_rate)
train_step = optimizer.minimize(loss)
model = CHOSEN_MODEL.set(optimizer)
leave_out = splits_number
while leave_out >= 0 do
    initialize_all_variables()
    val_errors = [ ]
    learning_rates = [ ]
    for each epoch in VALIDATION_EPOCHS do
        model.train((training-validation)_set, train_step)
        validation_error = model.test(validation_set)
        val_errors.append(validation_error)
        learning_rates.append(learning_rate)
        learning_rate + = 0.0001
        optimizer.set(learning_rate)
    opt_learning_rate = best_learning_rate(val_errors, learning_rates)
    optimizer.set(opt_learning_rate)
    initialize_all_variables()
    for each epoch in TRAINING_EPOCHS do
        for each sample in training_set do
            model.train(sample, train_step)
        test_error = model.test(test_set)
```

5 Results

I tested linear regression on two datasets (Wine Quality and Boston Housing), testing both MSE and MAE loss functions and using four different functions as model's hidden layers' activation functions (Rectifier, TanH, Sigmoid and Softplus). Besides, I varied many times the model's topology to choose one that could fit the problem. For each dataset fixed two different topologies a “Single” one and a “Deep” one. The first depicts a neural network with a single hidden layer, while the second depicts a neural network with more hidden layers. The Single topology is composed by a fixed number of units in each of the three problems:

$$hidden\_units = \frac{input\_units + output\_units}{2} \tag{14}$$

The Deep one has been adapted to the considered problem. As I already mentioned, I did not use cross-validation to estimate the number of hidden layers and the number of units for each hidden layer due to its computational cost. For this reason, the results shown, especially those concerning the Deep topologies, may be not the best results.

Linear regression's results on the Wine Quality Data Set and on the Boston Housing Data Set are showed respectively in Table 6 and Table 2; while logistic regression's result are showed in Table 8. The tables shows the cross-validation results as a mean of the produced predictors performances on the test set. Indeed, the Mean Test Error is the mean error computed on the test set using the loss function specified, while the Mean Error depicts the mean absolute difference between the predicted values and the labels on the test set. In the Table 8, the Mean Accuracy is computed as the mean of the accuracies (computed as $correct\_predictions/total\_predictions$) of the predictors obtained during cross-validation.

From the tests, I was able to notice that the Sigmoid and TanH functions give better results (lower errors) when used as hidden layer activation functions in models with a single hidden layer; while the Rectifier and Softplus functions perform better when using a model with more then one hidden layer.

Loss	Topology	Activation	Mean Test Error	Mean Error
MSE	Single	Rectifier	1.000	0.559
MSE	Single	TanH	0.785	0.590
MSE	Single	Sigmoid	0.685	0.624
MSE	Single	Softplus	0.951	0.543
MSE	Deep	Rectifier	0.890	0.529
MSE	Deep	TanH	0.716	0.687
MSE	Deep	Sigmoid	0.664	0.687
MSE	Deep	Softplus	0.956	0.543
MAE	Single	Rectifier	0.790	0.591
MAE	Single	TanH	0.684	0.668
MAE	Single	Sigmoid	0.715	0.698
MAE	Single	Softplus	0.803	0.615
MAE	Deep	Rectifier	0.766	0.622
MAE	Deep	TanH	0.708	0.708
MAE	Deep	Sigmoid	0.708	0.708
MAE	Deep	Softplus	0.789	0.613

TABLE 6: Results obtained by applying linear regression on the Wine Quality Data Set. The hidden layer in the Single topology was formed by 6 units, while the Deep topology was composed by 3 layers of 2 units each.

Loss	Topology	Activation	Mean Test Error	Mean Error
MSE	Single	Rectifier	126.279	7.675
MSE	Single	TanH	149.855	7.369
MSE	Single	Sigmoid	145.234	6.353
MSE	Single	Softplus	183.645	7.040
MSE	Deep	Rectifier	108.787	7.350
MSE	Deep	TanH	134.661	8.69
MSE	Deep	Sigmoid	134.571	8.685
MSE	Deep	Softplus	157.785	8.189
MAE	Single	Rectifier	8.439	5.952
MAE	Single	TanH	7.417	6.627
MAE	Single	Sigmoid	7.080	6.044
MAE	Single	Softplus	8.521	6.036
MAE	Deep	Rectifier	7.990	5.804
MAE	Deep	TanH	7.631	7.631
MAE	Deep	Sigmoid	7.639	7.639
MAE	Deep	Softplus	7.901	5.602

TABLE 7: Results obtained by applying linear regression on the Boston Housing Data Set. The hidden layer in the Single topology was formed by 7 units, while the Deep topology was composed by 3 layers of 2 units each.

Topology	Activation	Mean Accuracy
Single	Rectifier	-
Single	TanH	94%
Single	Sigmoid	76%
Single	Softplus	-
Deep	Rectifier	90%
Deep	TanH	-
Deep	Sigmoid	-
Deep	Softplus	86%

TABLE 8: Results obtained by applying logistic regression on the Wisconsin Diagnostic Breast Cancer Data Set. The hidden layer in the Single topology was formed by 15 units, while the Deep topology was composed by 10 layers of 2 units each. From the tests, it can be noticed that TanH and Sigmoid works well as hidden layers activation functions while using a single hidden layer's model. On the other hand, while using a model with more than one hidden layer, it is difficult to train the model with them, so Rectifier or Softplus functions (that in single hidden layer's model can't be used because of their gradients exploding) have to be used as activation functions. In this case, the loss function used is the Logaritmic loss.

6 Conclusions

The results obtained indicates that it is possible to predict a wine's quality with a margin of error of 0.6 on a range of real values between 0 and 10, given the features described in the Table 1. It is also possible to predict the price of a Boston house in 1970 in 1000\$'s, making a mistake of 6000\$-8000\$, given the features in the Table 2. Eventually, we can predict a breast cancer diagnosis with an accuracy of 90%, relying on the information presented in the Table 3.

To conclude, regression analyis combined with neural networks allows us to solve many statistical problems, from simple ones, like simple predictions/classifications problems, to more complicated ones, like nonlinear regression problems.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[3] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Modeling wine preferences by data mining from physicochemical properties," *Decision Support Systems*, vol. 47, no. 4, pp. 547 – 553, 2009, smart Business Networks: Concepts and Empirical Evidence. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923609001377

[4] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.