Universitat Autònoma de Barcelona

Title: Quantitative methods for Big Data: Neural Networks

Author: Marta Oliva Riera

Degree: Economia (en anglès)

Tutor: Michael Creel

Date: 08/06/2018

Table of Contents

1. INTRODUCTION

As it has already been proven, the use of big data has become a successful tool in the field of business. However, it has not yet become widely used in economics. The aim of this thesis is to analyse the possible ways in which big data techniques could be implemented in the field of economics. More specifically, I have focused on the use of neural networks, which are a machine learning technique that can be used for regression and classification with very large data sets.

The objectives I have for this project are the following:

- Explain what big data is
- Explore the main big data methods which may prove useful to economics
- Learn about how neural networks work
- Put some of these techniques into practice myself, specifically by training a neural network and using it for prediction in an economic model

With these objectives in mind, this project will start by introducing big data and its methods to perform predictive modelling, as well as the possibility of using them causal inference, in section 2. In section 3 I will give an in-depth explanation on what neural networks are and detail how they work. I will also present how to use the *neuralnet* package for R to train a neural network. Finally, in section 4, I will use the knowledge on neural networks to train one myself using the previously introduced methods.

An additional aim I hope to tackle with this project is to explain some of the quantitative methods used in Big data in way that can be understood by undergraduate students. I think it is important for economists to get acquainted with these methods but some of the literature in the topic seems quite advanced if you do not have a background in computer science.

2. BIG DATA

The term **Big Data** is defined as data sets that that are extremely large and sometimes too complex to be analysed using traditional data management tools. Using new tools, they can be analysed to reveal patterns, trends and associations especially relating to human behaviour and interaction. Big data is created from many different sources at an incredibly fast rate. Some of the sources of big data include computer-mediated transactions, e-mails, media, the Internet, Internet of Things (IoT) devices, sensor data, cell phones, among many others.

Economic data is now more abundant than ever, both from the private and public sectors. As businesses and the government work more with computers, an increasing amount data has been compiled on their activities at a whole new and previously unexpected degree. But how is this data different from what was previously available? According to Einav and Levin (2014) and Varian (2014a), the following are the main distinctive traits of big data:

- Very large volume of data, which reduces problems with statistical power.
- Real-time availability of data, which has proven to be very useful in business but not yet very exploited in economics.
- New types of variables are available that were not possible to know about before. This increases the number of potential regressors for our analysis as well as the complexity of the relations between variables.
- Data is less structured and has more dimensionality. This can be useful but requires manipulation to organize and select the data before it can be used.

Developing new techniques to deal work in these new settings is a challenge for economics research. Based on the previously mentioned characteristics, the new tools used by economists should allow for more powerful manipulation, better variable selection and more complex modelling of non-linear relations.

The tool often used to **process Big data** is a relational database, which works with Structured Query Language (SQL) to flexibly store and manipulate medium size datasets. To work with even larger data sets (including millions of observations) "NoSQL" databases are used, which are more primitive but can work with larger amounts of data.

Having processed it, the data set can be used for statistical analysis. If despite the manipulation the data set is still too large, a subsample for analysis can be selected through random sampling.

2.1 Big data analysis

Data science is the field that performs computer-assisted data analysis in order to extract useful information from big data. It uses several different techniques to perform prediction and summarization, as well as other tasks. Specifically, machine learning is the tool mostly used for prediction and data mining is used for summarization processes. **Machine learning** was first defined by Samuel (1959) as the field concerned with programming algorithms which give computers the ability to learn from the data available to them. Learning implies the progressive improvement of its performance in a specific task. On the other hand, **Data mining** is the process of examining large data sets to discover patterns and generate new information from them.

Regression analysis is the most frequently used tool in econometrics to find relationships between variables. However, borrowing some techniques from machine learning to perform predictive modelling could be better for economists when working with large datasets. As I previously defined it, machine learning uses an algorithm that can make the computer improve its predictive function based on input data that is provided to it. It is considered that the function gives good predictions when it is able to accurately forecast with out-of-sample data. Machine learning models are also able to update themselves when new data on the topic is provided to them, which could also be a useful feature.

Nevertheless, according to Einav and Levin (2014), a type of Lucas critique arises when economists think of using machine learning to predict the effects of new policy implementation. The Lucas critique says that it is naïve to try to predict the effects of a change in economic policy based only on the relationships observed in historical data. That's because people's decisions depend on the policies that are in place at the time, and changing them would also change their decision. Therefore, the usefulness of machine learning methods might depend a lot on the context in which they are used.

2.1.1 Causal inference

As stated by Varian (2014a) the area where there is more potential for collaboration between econometrics and machine learning is **causal inference**. Machine learning models so far have only focused on prediction, but their techniques have potential to include causality into their models. In fact, even if a predictive model cannot provide conclusions regarding causality, it is able to provide intuition into the causal impact of an action. Specifically, the model can predict what would have happened if that action had not been implemented, a sort of "control group prediction". Then the estimated causal effect can be measured by subtracting the real outcome of the action and the results from control group prediction, plus some selection bias. This is done in economics using difference-in-differences estimators as well as Instrumental variables, among other experiments. There is potential for machine learning to use this kind of econometrics techniques and provide possibly more accurate control group predictions.

2.1.2 Predictive modelling

Many firms have been using big data to build predictive models that help improve their efficiency, introduce new services and help in their decision-making process. There are many examples of these kinds of business applications of machine learning techniques, including Amazon's shopping recommendations based on past purchases. These methods that have become increasingly common in business are still relatively underused in the field of economics.

The goal of predictive modelling is to achieve good out-of-sample predictions. The **overfitting** problem refers to having a model that performs in-sample predictions well but does poorly for out-of-sample inputs. Machine learning has several different tools to solve the overfitting problem and make sure that the estimated models will be able to generalize their results to an out-of-sample data set.

First, there is the **cross-validation** method. It is a holdout method which consists of randomly dividing the data into "training" and "testing" subsets, which are arbitrarily sized. The training data set is used to for the estimation of the model. After that, the testing data set is used to evaluate how well the chosen model performs its prediction. It is also possible to increase the complexity of this method by using **K-fold cross-validation** instead. It works in a similar way, but it randomly divides the data set into k subsets instead of only two. K-1 subsets are used for training and the other one for testing. The process is repeated k times so that each of the subsets is used as the test set only once. The validation results are averaged once the cross-validation process is finished.

Repeating the cross-validation k times reduces the dependence of performance on the specific train-test division of data and reduces its variance.

Another method to combat overfitting is **regularisation.** It consists of penalizing models that are excessively complex because simple models tend to be better at generalizing their results. Lasso, Ridge, Lars and Elastic net regressions are all examples of regularization, each having a different penalization term for complexity. They all work by adding a penalty term to the estimation of the model's coefficients. For example, the penalty term for Lasso regression is $\lambda_1 \Sigma_j^p |\beta_j|$, where $\lambda_1 \geq 0$ is the penalty parameter. These also act as useful tools for variable selection when dealing with large samples because the penalties result in some of the variables' coefficients being zero – meaning they should not be included in the model.

Finally, another set of tools to improve predictive model performance are Bootstrapping, Bagging, and Boosting. All of these introduce randomness to the data, which helps reduce overfitting. **Bootstrapping** is based on the creation of random samples with replacement out of the data set, to estimate the distribution of a statistic. **Bagging** is the averaging of models estimated with different bootstrap samples, with the aim of improving the performance of the estimators. **Boosting** consists of repeating estimations where the misclassified observations are given increasing weights, and the final estimate is an average of the repetitions' results.

Lastly, I will quickly introduce a couple interesting **predictive modelling techniques** in machine learning that could be useful in the field of economics. **Classification and Regression Trees (CART)** are a machine learning tool used to create regression models that can be used to solve classification problems. They can classify multiple outputs and continuous dependent variables, the latter referring to Regression Trees. Their structure is similar to economics' decision trees, but in this case there is a classification choice at each node. CARTs are created by separating the data into parts and fit simple regressions in each node. They perform better when modelling non-linear relations, and can still be used if there is some missing data.

Bayesian Structural Time Series is an estimation method in machine learning that uses time series data. It is mostly applied for variable selection and prediction modelling. It uses Kalman filtering to estimate a time series model with different components (trends, random walks, seasonality, etc.) and the spike-and-slab variable selection method, after which the draws of posterior distribution of variables and coefficients are used to construct estimates and forecasts of the dependent variable. Choi and Varian (2012) used these methods to estimate the predictive power that Google queries have on different economic indicators.

There are many other methods of machine learning, one of which is artificial neural networks. As I have decided to focus this project on them, an in-depth description of their characteristics and use is given in section 3.

2.2 Opportunities and challenges of big data in economics

To conclude the theoretical framework on Big data, I will describe some of the opportunities and challenges of its applications in economics. These are based on the article by Einav and Levin (2014).

First and foremost, there are many opportunities in the application of big data both for economic research and economic policy decisions. Regarding research, large data sets with highly granular data can lead to studies that answer new questions which were previously not possible due to lack of detailed data. It also provides more credibility to any analysis, as there are more robustness measures. As I mentioned in the previous section, some machine learning techniques like regularisation and k-fold cross-validation would be valuable additions to an economist's toolkit. Furthermore, the use of big data has potential to incorporate heterogeneity into econometric models: taking advantage of how detailed the data is, the models could capture the response of several subgroups instead of just the average response.

With respect to economic policy, it will be interesting to use big data to create alternative measures to track private sector activity, other than traditional surveys. These could help estimate the inflation, employment, consumer spending, etc. faster than the surveys do, by using data sets on prices and spending that are available online, or indirectly, by looking into Google search data trends like Choi and Varian (2012) did.

On the other hand, there are also some challenges associated with using new techniques that are able to work with such large quantities of data. Firstly, economists will need to learn how to use new big data tools if they are to take advantage of the new data sets available to them. These include data management programmes, like SQL databases, as well as programming languages, like R, getting acquainted with different machine learning algorithms and other techniques. Another challenge will be to get broader access to more of these big data sets, as right now access to both public and private data is usually restricted due to privacy concerns. Public data is a powerful resource which is underutilised, and research could benefit from having greater access to it now that more tools are available. Finally, with such large datasets, it becomes harder and more time-consuming to summarize the data and to find possible meaningful relationships in its variables. The results can provide interesting insight but will also require a more complex analysis.

3. NEURAL NETWORKS

Artificial neural networks are computing systems that can be used to approximate any complex functional relationship. They are information processing machines made up of simple interconnected processing units, or neurons. Neurons work in a non-linear manner, through parallel processing of the information. They are used to infer meaning and detect patterns in complex data sets, without necessarily prespecifying the type of relationship between the regressors and response variables like you would need to do in a generalized regression model. Therefore, neural networks can be thought of as extensions to generalized regression models.

The idea of neural networks as computing machines was first introduced in 1943 by McCulloch and Pitts. Since that moment, neural networks have usually been compared to the human brain, because they acquire knowledge from their environment through a learning process and use connections between their neurons, or synaptic weights, to store the knowledge they have acquired. Learning is an iterative process in which the neural network learns from its environment (the data provided to it) in order to improve its performance in a specific task.

I will work with the ***neuralnet*** package for R, which uses a very flexible function to train neural networks. It can work with an arbitrarily large number of covariates and response variables, but as we will see the increased complexity might complicate the training process, causing it to stop it the maximum steps are reached before the algorithm converges. There are some other R packages available to train neural networks, including *nnet*, *RSNNS* and *AMORE*. However, *neuralnet* is specifically built to train neural networks in the context of regression analyses, and its algorithm is fastest for this purpose.

3.1 Basic structure

The design of a neural network is fundamentally based around neurons. A **neuron** is an information processing unit composed of the following:

- Summation: a neuron k takes all the input signals (x_j) multiplied by their respective synaptic weights (w_{kj}) and sums them, resulting in the linear combiner u_k.

- Synapses: they are connecting links between the neurons in each layer. They can only connect one neuron to another one in the subsequent layer. Each of them is characterized by a weight (w_{kj}), which represents the effect of the previous neuron on to the one it connects to. The weights are similar to the coefficients in a regression model. The weights are usually started at random values which are drawn from a normal distribution, and are later on adapted during the learning process.
- Bias nodes (b_k)*:* a bias is an externally applied effect to each of the hidden layers in the neural network, which modifies the input of the activation function. They act like intercepts in regression models (a constant learned outside of your input data) and allow us to shift the learned model. The bias is added to the summation's result to create the input for the activation function.

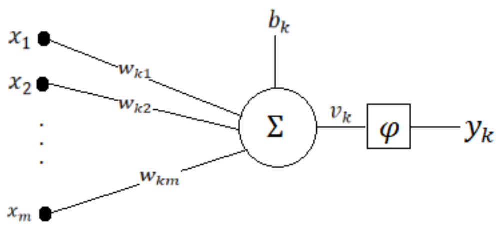
All data passing through the neural network does so as signals. These signals are first processed by the summation previously described and then by the activation function, which limits the neuron's output amplitude to some finite value. Activation functions are also referred to as squashing functions. I will give a more detailed description of activation functions in section 3.2.

Functions (1) and (2) and Figure 1 are the mathematical and graphical representations of neuron k, where u_k is the weighted summation of inputs to the neuron, v_k is the activation potential and y_k is the output signal, the latter being the outcome of the activation function (φ). Note that in all figures the signals flow from left to right.

$$u_k = \Sigma_{j=1}^{\mathrm{m}} w_{kj} x_j \tag{1}$$

$$y_k = \varphi(u_k + b_k) = \varphi(v_k) \tag{2}$$

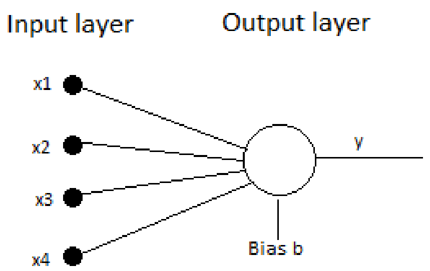
Figure 1: Graphical representation of neuron k.



There are three basic network architectures, or structures in which the neurons of a network are organized. A **perceptron**, or single layer feedforward neural network, is the most basic type of network: it only has an input layer and an output layer. It receives n input nodes (independent variables) and processes them on the output layer neurons using a weighted summation and an activation function, resulting in one or more output nodes (dependent variables). It is used to classify patterns that are linearly separable. If the network has a single neuron, it will only be able to classify its inputs into two categories. Rosenblatt first introduced the concept of a perceptron in 1957.

An example of a perceptron is illustrated in Figure 2, where the black circles are input nodes and the large circle is a neuron.

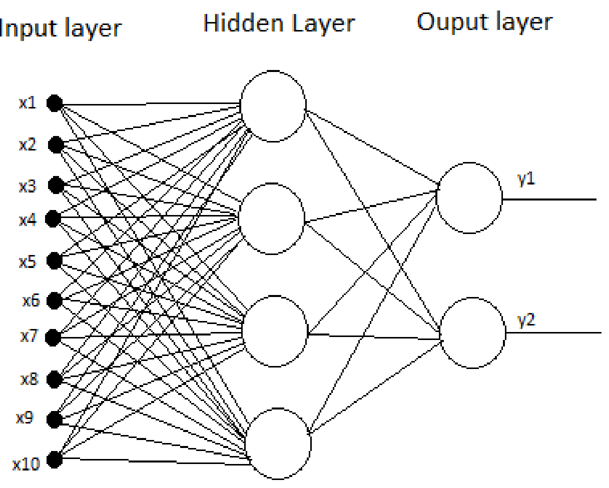
Figure 2: Perceptron with 4 input nodes and 1 output neuron



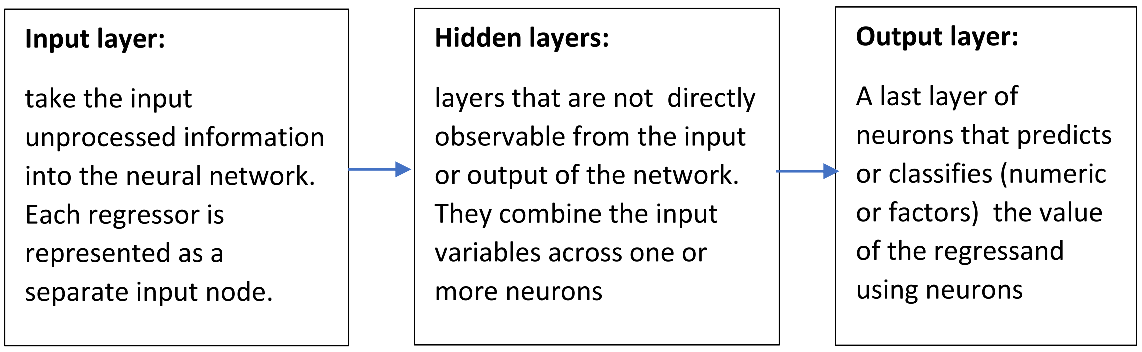
Multi-layer feedforward neural networks consist of neurons organized in one or more hidden layers, as well as the input and output layers, all connected through synapses. They overcome the limitations of the perceptron and are used to model more complex relationships between the variables, as they can extract higher-order statistics from the inputs provided. Multilayer neural networks have a high degree of connectivity.

As it can be seen in Figure 3, each neuron in any layer is connected to all the neurons or nodes in the previous layer. The input nodes are processed in the hidden layer neurons, whose output is then passed on to the following layer (in this case the output layer) as inputs. If there were more hidden layers, the process of feeding the data forward would go until it reached the output layer, whose output represents the overall response of the network to the initial inputs. Multi-layer perceptrons are the cornerstone of *neuralnet.*

Figure 3: Multi-layer feedforward network with 10 input nodes, one hidden layer and 2 output neurons



In addition to those network architectures, we can also find **Recurrent Neural Networks.** These are neural networks that include at least one feedback loop, through which the output is fed back to the input nodes to update them through a unit-time delay element. These may or may not include hidden layers. The presence of loops has a profound impact on the learning capabilities of the neural network. In fact, the type of network architecture chosen is always closely related to the learning algorithm used to train it.



Input layer:

take the input unprocessed information into the neural network. Each regressor is represented as a separate input node.

Hidden layers:

layers that are not directly observable from the input or output of the network. They combine the input variables across one or more neurons

Output layer:

A last layer of neurons that predicts or classifies (numeric or factors) the value of the regressand using neurons

Figure 4: Layers of a neural network

Figure 4 shows how the layers of a neural network are ordered and what function each of them performs. As I have previously mentioned, a perceptron does not have any hidden layers. However, hidden layers can be included to increase a model's flexibility. As Hornik, Stinchcombe and White (1989) proved in their article about Multi-layer feed-forward networks, that a neural network with just one hidden layer and a finite number of neurons is enough to model any piecewise continuous function. This means that, under certain assumptions on the activation function, these networks have potential to be universal approximators.

The structure of a neural network can be referred to as a formula describing the dependent and independent variables used in the network (determining the nodes in the input layer and neurons in the output layer) and a vector define the hidden layers, where the number of elements in the vector determines the number of layers and each number sets the neurons that layer has. These are used to set up a neural network in *neuralnet,* as I will explain in section 3.5.1. As an example, a neural network with two hidden layers and 5 and 3 neurons in each hidden layer would be represented by the following vector: $hidden = (5, 3)$.

3.2 Activation function

The activation function defines the output of a neuron as a function of its activation potential (v_k). It determines whether neurons in the following hidden layer will be activated or deactivated. Right now, *neuralnet* uses a single activation function for all their neurons. Some common activation functions include the following:

- **Threshold function**: if the activation potential is positive or zero, it outputs 1. It outputs zero otherwise. It is the simplest activation function, although it is not used a lot in current implementations.

$$\varphi(v) = \begin{cases} 1 & if\ v \geq 0 \\ 0 & if\ v < 0 \end{cases} \tag{3}$$

- **Sigmoid function***:* this is the most common type of activation function used in the setting of neural networks. It is strictly increasing and S-shaped. In contrast to the threshold function, it offers a continuous range of variables from 0 to 1 and is also differentiable.

$$\varphi(v) = \frac{1}{1 + e^{-av}} \tag{4}$$

Where a is the slope parameter.

If we were interested in getting an output ranging from -1 to 1 instead of 0 to 1 we could redefine the previous functions in the following way:

- **Signum function**:

$$\varphi(v) = \begin{cases} 1 & if\ v > 0 \\ 0 & if\ v = 0 \\ -1 & if\ v < 0 \end{cases} \tag{5}$$

- **Hyperbolic tangent function**:

$$\varphi(v) = \tanh(v) \tag{6}$$

If we wanted to base our analysis on a stochastic neural model instead of a deterministic one, we could add probability to the previous activation functions to determine if the neuron is "fired" or not.

3.3 The Learning process

One of the main characteristics of neural networks is their ability to learn from their environment to improve their performance. Haykin (1999) defined the **learning process** of neural networks in the following way:

"Leaning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter change takes place".

So, in the process of learning, the environment stimulates the neural network, which optimizes its weights based on a learning algorithm, and changes the way it interacts with the environment. This process is iterated and the network tends to gradually improve its performance. It ends when a pre-specified condition is fulfilled, such as minimizing the error signal or reaching the maximum number of iteration steps.

I will focus my analysis on a specific type of learning process: error correction learning. The type of learning process you follow depends on the type of neural network you work with. In my case, error correction learning is used for both single layer and multilayer feedforward neural networks.

In this learning process, the error signal measures the difference between the desired result and the actual output signal of the neural network:

$$e_k = d_k(n) - y_k(n) \tag{7}$$

Where the output neuron k in a feedforward neural network receives the input signal $x(n)$ from the previous hidden layers, $y(n)$ is the neuron's output and n represents the time step of the training process. **Error correcting learning** will adjust the values of the neuron's synaptic weights, in an iterative manner, until the actual output is close to the desired response. The adjustment will depend on the learning algorithm we have chosen to use, but they all attempt to minimize the error. The learning rate will determine the speed at which the desired outcome is achieved: a low learning rate will result in a slow convergence; a high learning rate might result in missing the minimum error point. Therefore, the choice of learning rate is crucial for the convergence of the process, but it is set by the user and there is no rule to know which learning rate will work best in each specific context.

Other possible types of learning processes include memory-based learning, where all past experiences are stored in a large memory of classified input-output examples; Hebbian learning, which is based on associative learning and increases synaptic efficiency as a function of the correlation between the neurons on either side of it; Competitive learning, where output neurons compete to be activated; and Boltzmann learning, which uses a stochastic learning algorithm and neurons that operate binarily – being on or off.

3.3.1 Learning paradigms

In addition to the type of learning process, there are three possible learning paradigms, or environments in which a neural network can operate:

1. **Supervised learning:** or learning with a teacher, refers to the situation in which the neural network has no knowledge of the environment. The teacher has knowledge on the environment and provides the desired response that the network should have to a training vector. The training process is used to transmit the knowledge of the environment from the teacher to the network.
   Specifically, within error-correcting learning, supervised learning starts with synaptic weights that are random values drawn from a normal distribution, and the network adapts them according to the learning algorithm chosen in order to reach the desired response. When the error is minimized the process of training

the network is completed, as it has achieved to emulate the teacher's desired outcome.

There are two main methods of supervised learning with multi-layer perceptrons. In the on-line method, the adjustments to the synaptic weights are performed on an example-by-example basis: a first random pair of input vector and desired response are presented, the weights are adjusted and the process is repeated until all training vectors have been used. This method is simple to implement and effectively solves large-scale pattern classification problems. The other method of supervised learning, the batch method, presents all the training sample example to the network at the same time and then adjusts the weights. The process is repeated and at each period the training samples are presented in a random order. This method provides a more accurate estimation of the gradient vector and allows for the parallelization of the learning process. It is useful to solve nonlinear regression problems.

2. **Reinforced learning:** training is performed through continued interaction with the environment, in order to minimize the performance index. This process is built around a critic: it receives a temporal sequence of signals from the environment and transforms them into heuristic reinforcement signals after a delay (delayed reinforcement). Despite the additional difficulty it involves compared to supervised learning, reinforced learning is interesting because it develops the network's ability to interact with its environment and learn to perform a task based only on its own experience.
3. **Unsupervised learning:** there is no teacher or critic to supervise the network's training process. Instead, it uses a task-independent measure to evaluate the quality of representation that the network should learn. Based on that measure it optimizes the networks' synaptic weights.

3.3.2 Learning algorithms

A **learning algorithm** is a set of well-defined rules that the neural network follows in order to complete its learning process. There are many possible algorithms to use, they all aim to optimize the synaptic weights but each adjusts them in a different way.

The *neuralnet* package allows us to switch between different algorithms: backpropagation; resilient backpropagation without weight backtracking; and GRProp, the globally convergent version by Anastasiadis, Magoulas and Vrahatis (2005). The latter was a modification to the resilient backpropagation algorithm, which aims at improving the convergence speed and stability of the previous algorithms.

I will focus on error correcting algorithms for single layer and multi-layer neural networks that are under supervised learning, based on the descriptions by Haykin (2009).

First, the **Perceptron algorithm**. Remember that for the perceptron to work properly, the patterns it classifies must be linearly separable (in the case of two classes, they should be easily separable with a straight line, the decision boundary). To make the previously introduced notation more straightforward, I will treat the bias (b_k) as a fixed input whose weight is equal to one. Therefore, the input ($x(n)$) and weight ($w(n)$) vectors have m+1 elements each instead of m, which is the number of inputs. The number of steps in the training process is denoted by n.

Consider a perceptron whose inputs are classifiable into two categories (C_1 and C_2). We provide it with sets of training vectors H_1 and H_2, corresponding to each of the classes respectively. The training process will imply the adjustment of the weights vector w until the two classes are linearly separable for all input vectors x. Equation (8) will hold for all input vectors belonging to C_1, while equation (9) will hold for all input vectors belonging to C_2.

$$w^T x > 0 \tag{8}$$

$$w^T x \leq 0 \tag{9}$$

Therefore, if the training set $x(n)$ is correctly classified by weight vector $w(n)$ on the nth iteration of the process, the weight vector will not be adjusted ($w(n+1) = w(n)$). Being correctly classified implies that if we know the vector comes from the set H_1, $w^T(n)x(n) > 0$ will hold, so that the network is classifying the input vector into C_1. However, if $x(n)$ is not correctly classified, the weight vector will be updated in the following way:

$$w(n+1) = w(n) \pm \eta(n)x(n) \tag{10}$$

where $\eta(n)$ is the learning rate, a parameter that adjusts the change in weight. If $w^T x > 0$ but $x(n)$ belonged to C_2, the weight should be decreased; if $w^T x \leq 0$ but $x(n)$

belonged to C_1, the weight should be increased. The process should be repeated until the training input vectors are correctly classified.

The learning rate can take any positive value, but if it is fixed to a constant value then the **Fixed-increment adaptation algorithm** for the perceptron is used, as I have just explained. If the learning rate is variable then the **Perceptron convergence algorithm** (Lippman, 1987) should be considered. For the example of the latter algorithm, assume that the initial weights are set at zero. The learning rate can be changed but should be taking values between zero and one. Then activate the perceptron by providing it with the training input vectors $x(n)$ and a desired response $d(n)$. In this case, the activation function is the signum function, so the actual output and the desired response will be described by the following functions:

$$y(n) = sng[w^T(n)x(n)] \tag{11}$$

$$d(n) = \begin{cases} 1 \quad if\ x(n)\ belongs\ to\ C_1 \\ -1 \quad if\ x(n) belongs\ to\ C_2 \end{cases} \tag{12}$$

Then the weights will be adjusted by the rule in equation (13). The process will be repeated until the desired response is achieved, which will be the point where $d(n) - y(n) = 0$ so that $w(n+1) = w(n)$.

$$w(n+1) = w(n) \pm \eta[d(n) - y(n)]x(n) \tag{13}$$

Secondly, I will introduce the **Least Mean Square (LMS) algorithm**. This algorithm works on neural networks called adaptive filters, which are characterized by arbitrary initial weights, continuous adjustments to the weights are made based on the network's behaviour with respect to the desired response, and the computation of adjustments being completed in a single period. These neural networks create a feedback loop around their neurons, as their output is used to compute the error signal and that is in turn used to adjust the initial weights, resulting in a new output and error signal. The LMS algorithm was developed by Widrow and Hoff in 1960 and it can be used for prediction problems. This algorithm is computationally efficient, simple to code and robust to external disturbances.

The LMS algorithm is set to minimize the instantaneous value of the cost function defined in equation (14). In this case, we also assume that the initial synaptic weights are set to zero. LMS uses the Method of Steepest Descent for its optimization, meaning that the adjustments to the weights are in the direction of steepest descent, opposite to the direction of the gradient vector $\nabla\varepsilon(n)$. This method converges to the optimal weights slowly and learning rate has a large influence on its convergence behaviour. An instantaneous estimate of the gradient vector is written in equation (15), considering that the error signal is defined as $e(n) = d(n) - x^T(n)\widehat{w}(n)$.

$$\varepsilon(n) = \frac{1}{2}e^2(n) \tag{14}$$

$$\frac{\delta\varepsilon(\widehat{w})}{\delta\widehat{w}(n)} = e(n)\frac{\delta e(n)}{\delta\widehat{w}} = -x(n)e(n) \tag{15}$$

Using the gradient vector in equation (15) for the method of steepest descent, the weights will be adjusted according to the Widrow-Hoff rule or Delta rule, described in equation (16). The Delta rule states that the synaptic weight adjustment of a neuron is proportional to the product of the error signal and the input at time n.

$$\widehat{w}(n+1) = \widehat{w}(n) + \eta x(n)e(n) \tag{16}$$

This process of weight adjustment will be repeated until the cost function (a function of the error signal) is minimized. Basing the learning process on this algorithm implies the assumption that the error signals are directly measurable and that the desired response is externally provided, like in a supervised training setting. Assigning credit for overall outcomes becomes difficult when there are hidden neurons involved.

Finally, I will present the **Backpropagation training algorithm**. It is commonly used in the learning process of multi-layer perceptrons. This algorithm solves the credit-assignment problem that arises when training a multi-layer perceptron with error-correcting learning, due to the fact that the output of the hidden neurons is not observed. It does so by computing the gradients differently depending on whether the neuron is at a hidden layer or at the output layer.

Based on the on-line method of supervised learning, the backpropagation algorithm works in the following way. The synaptic weights are initially set to values of a uniform distribution whose mean is zero and variance makes the standard deviation of the neuron's activation potentials lie between the linear and standard parts of the sigmoid activation function. The cost function is the total instantaneous error energy, defined by equation (14). Then present a set of training examples to the network, formed by an input vector $x(n)$ and a desired response vector $d(n)$. In the forward computation phase, the input vector is applied to the input layer and the desired response to the output layer, so that the error signal can be computed. The error signal for neuron j in the output layer L is computed in equation (17), where $d_j(n)$ is the jth element of the desired response vector, or desired response for neuron j, and $o_j(n) = y_j^L$ is neuron j's output signal.

$$e_j(n) = d_j(n) - o_j(n) \tag{17}$$

Once we have the error signals, the Backwards computation phase can be started, where the local gradients of the network are computed in the following manner, depending on whether neuron j is in a hidden layer l or in the output layer L:

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\varphi'_j\left(v_j^{(L)}(n)\right) & for\ neuron\ j\ in\ output\ layer\ L \\ \varphi'_j\left(v_j^{(l)}(n)\right) \Sigma_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & for\ neuron\ j\ in\ hidden\ layer\ l \end{cases} \tag{18}$$

where the activation potential is defined as $v_j^l(n) = \Sigma_i w_{ji}^l(n) y_i^{l-1}(n)$, with i being a neuron in the previous layer; and φ'_j is the activation function's differentiation with respect to the argument.

Finally, the synaptic weights in layer l are adjusted according to the generalized delta rule, described as $\Delta w_{ji}(n)$ in equation (19), where α is the momentum constant and η is the learning rate. The Widrow-Hoff rule presented for the perceptron is a specific case of the generalized delta rule, where α was set to zero. The addition of a momentum provides stability to the algorithm, as it controls the feedback loop around the weight modification.

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha\left[\Delta w_{ji}^{(l)}(n-1)\right] + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n) \tag{19}$$

The Forward and Backwards computation phases should be iterated while presenting new training data sets so that the synaptic weights will be adjusted until one of the chosen stopping criteria is met. These stopping criteria must be set because the backpropagation algorithm cannot be shown to converge on its own. The most reasonable stopping criterion is reaching a minimum in the error surface, which happens when the gradient vector of the error surface with respect to the weights is sufficiently small or when the rate of change of the cost function $\varepsilon_{av}(w)$ is sufficiently small. Another possible reason for the learning process to stop is if it reaches a maximum number of iterations before having completed any of the previously defined convergence criteria.

A variation of the Backpropagation algorithm that can be used in the *neuralnet* package is the **Resilient Backpropagation** algorithm. It differs from traditional backpropagation in that it uses a different learning rate for each weight. This rate can also be changed during the process. This solves the problem of setting a learning rate that is appropriate for the overall learning process, which can be hard for complex networks. Moreover, resilient backpropagation only uses the sign of the gradient to modify the weights, not their magnitude. That ensures that the learning rate will have equal influence over the whole network. The learning rate will be increased if the sign of the gradient stays the same, and decreased if the sign changes (as that would imply that the minimum point was missed), in order to speed up the convergence to the local minimum.

3.4 neuralnet

As described in its documentation (Fritsch, Guenther, Suling, and Mueller, 2016), *neuralnet* is a package that can be used to train neural networks using different algorithms and in a flexible manner, as it allows us to choose the error and activation functions we prefer.

3.4.1 *Fitting a neural network*

The following are some basic steps on how to fit a neural network with *neuralnet*. I have written in italics some of the basic functions required to use this package, as well as examples of how to code the different steps.

0. Set Working Directory and read the data: *setwd(direction)* to the direction where your data files are saved, then set *read.csv("file")* to open the data file itself. Or work with one of R's datasets, in which case you just need to load the package it is related to.

1. Decide which **dependent** and **independent variables** out of your data set you will use in your neural network.

2. Check that there are no observations missing. If there are, fix the data set to either fill them or remove them.

3. **Data preprocessing**: normalise the data so that our results are not misled by the scale of the some of the variables, and the accuracy of our prediction is higher. You can do so by min-max normalisation, Z-score normalisation, median and MAD, and tan-h estimators. The *scale()* function in R could also be used. I use min-max normalisation, which transforms the data into a common range while maintaining the original distribution. It can be used by creating the *normalise* function below, where *df* is a data frame with the variables to normalize:

```
normalise <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
 }
dfnormalised <- as.data.frame(lapply(df, normalize))
```

   Additionally, if we have factor or character variables, we should convert them to numerical variables (dummy variables). This is done using the *model.matrix()* function.

4. Divide the data into **training** and **test sets** to perform the learning process. These should be assigned using random sampling, with the function *sample()*. Remember to use *set.seed()* every time you use random sampling or the results will not be reproducible.

   a. Create an index with the function *sample().* In the following example the training sample would contain 75% of the observations in our data:

   ```
   index <- sample(1:nrow(data), round(0.75*nrow(data)))
   ```

   b. Specify the variables *trainNN* and *testNN* by using the index in square brackets on the normalized data:

   ```
   trainNN <- dfnomalized[index, ]
   testNN <- dfnormalized[-index, ]
   ```

You can also fit a **linear regression model** and test it on the test set, using the Mean squared error to see how far our predictions are from the real data and use it to later compare with the MSE resulting from testing the neural network.

5. **Fit the neural network** using the scaled data:

    a. Install the package *neuralnet: install.packages("neuralnet")*

    b. Load the package: *library(neuralnet)*

    c. Decide how many layers and neurons there will be in the network. One hidden layer is usually enough. However, there is no rule of thumb to decide which setting will fit your model best, so the only option is to experiment with different configurations.

    d. Fit the neural network, using the function *neuralnet().*

    Its basic arguments are the formula describing the function to be fitted (*f,* which should be previously defined *as.formula(y~x1+x2+...+xn)*), where the function's data comes from the train subset (*data=trainNN)* , and the hidden layers and neurons (*hidden=(,)).* If hidden is equal to a scalar there is only one hidden layer with however many neurons the scalar indicates; if it is a vector each number in it determines how many neurons there are in each subsequent hidden layer.

    Another possibility is to change the algorithm used (*algorithm=).* Different algorithms may require additional arguments in order to work. Some other relevant arguments include *threshold, stepmax, startweights, learningrate, act.fct,* and *rep.* The package's documentation (Fritsch et al., 2016) can be checked for more details on all the possible arguments, which allow for a lot of customization. The only compulsory arguments to specify are the formula and the source of the data, so the documentation also details what the default values of the optional arguments are. For example, the default number of hidden layers is one, with a single neuron, the default algorithm is resilient backpropagation with weight backtracking and the default activation function is the logistic function.

    NN <- neuralnet(f, data, hidden, ...)

After using the function *neuralnet(),* all information about the training process and the trained neural network is stored in NN (or any name you have given to the trained neural network). The information about the network can be explored by printing the elements in NN, which include: *NN$net.result, NN$result.matrix, NN$weights, NN$generalized.weights, NN$startweights. net.result* is a list of the neural network's output, equivalent to fitted values. *result.matrix* is a matrix containing a summary of the main result of the network, including the error, reached threshold, number of needed steps, information criteria and the weights. The other elements are lists of the final, generalized and starting weights.

6. **Prediction**: once the network has been trained, it can be used to predict values of the dependent variable. This is done in this package using the *compute()* function, which calculates and summarizes the output of all neurons in the network. If it is provided with a vector of new covariate combinations (*NNtest*) that were not in the train set of the neural network, the compute function can be used to calculate the new outputs.

   newoutput <- compute(NN, NNtest ,...)

   newoutput$Net.result

   Net.result is a list of the predicted outputs. The results will be scaled due to the previous normalization, so they must be transformed before we can compare them to the real values. These comparisons can easily be done through visualization.

   We can also evaluate it using just the Mean Squared Error, and compare it to the previous results using the linear regression model, if we estimated that beforehand. If the MSE of the neural network is smaller than the MSE of the linear model then the network is doing a better job at predicting the dependent variable. However, keep in mind that these results depend on the train-test set split that has been performed above.

3.4.2 Additional features

Some additional features available when using *neuralnet* to fit a neural network include the ability to visualize the results in different ways as well as the computation of confidence intervals for the weights that the network has estimated.

There are three possible ways to **visualize** the results of the learning process of a neural network. The first one is simply using the *plot()* function on our neural network *NN*. It results in a graph showing the structure of the trained neural network: all covariates and result variables, layers, neurons, and synapses, along with their corresponding weight. It also includes the resulting error and the number of steps involved in the training process. It is a good option to visualize the structure of the neural network, but it can look very cluttered and not convey much information if the neural network is very complex. To make larger neural networks' plot clearer, we can customize it using the parameters *dimension* and *plot* to edit the size of the plot and that of each neuron.

Another possibility is to plot the generalized weights using *gwplot.* It plots the generalized weights with respect to each covariate, so it gives us multiple plots. It allows visualizing the possible linear relations within the data in a clearer way.

The final possibility regarding visualization is to plot the output results of the neural network (y_k for all output layer neurons). This should provide a visual approximation to the performance of the trained neural network. This can be plotted alongside the results of the equivalent linear model, to visually compare them. Less dispersion in the output implies better predictions from the model.

On the other hand, the *confidence.interval()* function in the *neuralnet* package allows us to compute the **confidence intervals** for each of the weighs in the trained neural network. That is possible as long as the weights of the neural network follow a normal distribution, which is the case when the network is identified (it does not include neurons that have no effect or ones that are a linear combination of other neurons in the input or hidden layers); and the error function equals the negative log-likelihood. The *confidence.interval()* function does not control for these conditions being satisfied, so the user should be careful when interpreting the results, as they will not be meaningful unless the assumptions are satisfied.

3.5 Benefits and challenges of neural networks

As a point of closure to my theoretical analysis of neural networks, I will discuss some of the benefits and challenges related to this machine learning method, mostly based on Haykin (2009).

First, I will mention some of the beneficial characteristics of neural networks. They are non-linear in their processing of information and have a parallel distributed structure. As I have described in detail in section 3.3, one of the network's most characteristic features is their ability to learn from the training set of inputs and afterwards generalize, so that they can predict reasonable outputs for inputs that are out of the training sample. These characteristics make neural networks able to find good solutions to complex problems. In addition to that, neural networks do not require making any prior assumptions on a statistical model for the input data, in a similar manner to nonparametric statistical inference in statistics: the networks learn from the training set they are provided with and construct an input-output mapping for the specific problem they are working with. Networks are also very adaptable to changes in their environment, so they can be retrained when they are presented with new conditions, and they will adjust the synaptic weights accordingly. Another benefit arising from the network's distributed information structure is their fault tolerance: if a neuron or synapsis is damaged, the network should be able to mostly keep up their performance, degrading slowly rather than having a catastrophic failure. Finally, their analysis and design are uniform, using the same notation and structure for all of its different applications.

On the other hand, there are also some challenges involved with the use of neural networks. Their hidden layers and neurons of multi-layer perceptrons act like black boxes, which, along with their nonlinearity and full connectivity, makes their analysis difficult to tackle. On a related subject, the neural network training process can be computationally expensive and sometimes that additional cost might not yield better results than simpler methods would. Regarding the algorithm chosen for the learning process, as it works toward the minimum in the error surface it might get caught in a local minimum and miss the global minimum. Some algorithms address this by including random jumps to different sections of the error surface. To conclude, a difficulty that I have mentioned several times: there is no set rule to optimally set the learning rate and number of hidden neurons and layers, the only way to find what is optimal for each setting is experimentation.

4. EMPIRICAL EXERCISE

4.1 Introduction to the exercise and data

As an empirical exercise to complement my analysis of big data methods applied to economics, I have decided to train a multi-layer feedforward neural network in R, using the *neuralnet* package which I have previously introduced.

To train the model, I have used a dataset that is preloaded into an R package. Specifically, the *Wages* data set from the *Ecdat* package. My data set includes panel data on individual wages as well as other characteristics of the individuals, like their experience, years of education and marital status. It has data on 595 United States' individuals in the period from 1976 to 1982, a total of 4165 observations. The *Ecdat* package includes many econometrics data sets ready to use in R, which are all from published books or articles that used them in their models. My dataset was originally used in an article by Cornell and Rupert (1988) published in the Journal of Applied Econometrics, and more recently it was published with the book Econometric Analysis of Panel data by Baltagi (2003).

A dataset with only slightly over 4000 observations cannot be considered Big Data. However, I have limited computing power available so I am restricted to using smaller data sets. In addition to that, I want this exercise to be focused on my ability to use new tools, specifically neural networks and R, to solve a problem similar to those I have seen during the Bachelor's degree. That is why I have decided to use a data set from R rather than collect my own data set from a statistical service, which would have been difficult to summarize and manage if I wanted a dataset with as many observations as the one I have used. My focus has been placed on being able to successfully apply these new techniques and solve the problems that arise when using new methods, rather than in getting an economically relevant interpretation out of my results. However, I am still interested in using economics' data during my training with neural networks, which is why I have chosen this dataset.

4.2 Results

In this section I will introduce the results of my analysis. The script I used to achieve these results can be found in the Annex, along with a short description of the variable names. During my analysis I followed the steps I detailed on section 3.4 that explain how to work with the package *neuralnet* to train an artificial neural network.

To begin with the analysis, I opened the data and checked if it had any missing values, which was not the case. Then I proceeded to normalize my data set so that the magnitudes of the variables did not affect the learning process of the neural network and it would converge. This was not as straightforward as I expected because the factor variables had to be excluded from the normalization and later turned into dummy variables so they could be included in the neural network. The creation of dummy variables implied the renaming of all the factor variables. For example, bluecol became bluecolyes, which means that the dummy is equal to one for the individuals that are blue collar workers.

After the pre-processing of the data set was completed, I created the training and testing sets. Having done that, I usedthe *neuralnet()* function to create my first artificial neural network. I decided that I wanted the network to have only one hidden layer with six hidden neurons. This initial decision is quite arbitrary, as there is no optimal rule to set the number of hidden neurons. The decision to only include one hidden layer to start with was based on the study by Hornik et al. (1989) that proved that a neural network have potential to be universal approximators – a network with only one hidden layer and a finite number of neurons is able to model any piece-wise function. The dependent variable is the logarithm of wages and the independent variables are all of the others in the data set: experience, work hours, years of education, being a blue-collar worker, employed in manufacturing, black, living in the south, living in a metropolitan area, marital status, gender and union membership. I used the default values for the rest of arguments in the *neuralnet()* function, so my neural network was trained using the resilient backpropagation algorithm with weight backtracking, and the activation function was the logistic function. The error was computed as the Sum squared of the residuals.

The trained neural network, with all of its adjusted synaptic weights can be seen in Figure 5. The large number of independent variables makes the first layer of weights impossible to distinguish. The bias nodes are represented in blue. More detailed results are given by the *NN6$netresults* table which, in addition to all the synaptic weights seen in the figure, contains interesting details on the learning process of the neural network. The network needed 91988 steps to reach the minimum in the error surface, which is 10.7.

Figure 6 First neural network, with 6 hidden layers

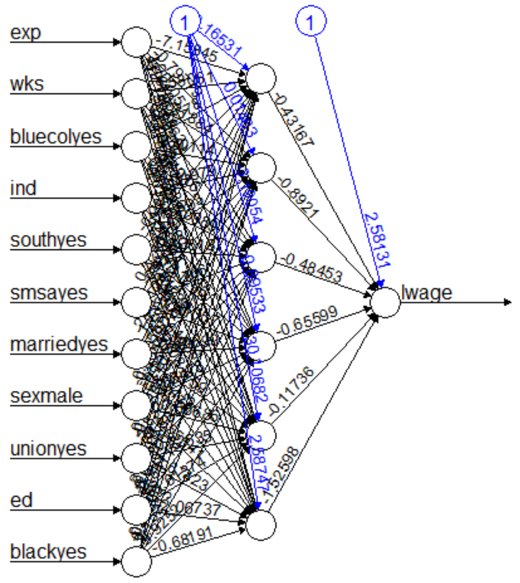
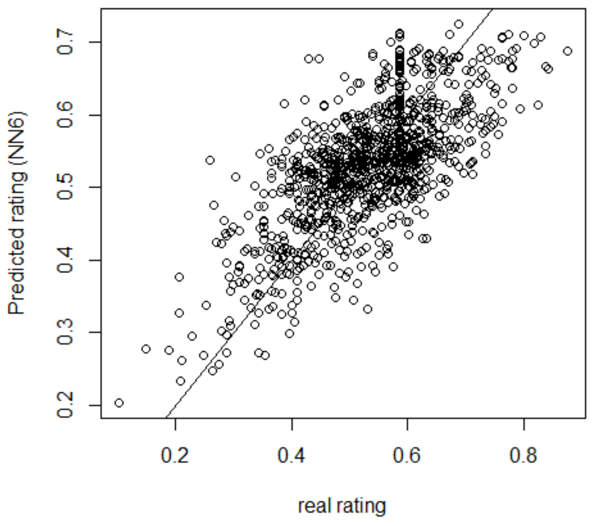


Figure 5 Predicted and real values of the test set (NN6)



Having completed the training process, I used the network to predict the values of the train set by using the *compute()* function. After reversing the normalization process that I used to train the network on the results, I illustrated them against the real values in the training set in Figure 6. If the points were all on the straight line, it would mean that the predictions were exactly equal to the real values of observations in the train set. That did not happen, but the points are quite clustered around the line which suggests the prediction is not bad.

As a last part of my exercise, I decided to train the network again by changing its architecture. This second version had two hidden layers, with eight and three hidden neurons respectively. I repeated the whole process in the same way, and the results are

Figure 8 Second network, with two hidden layers

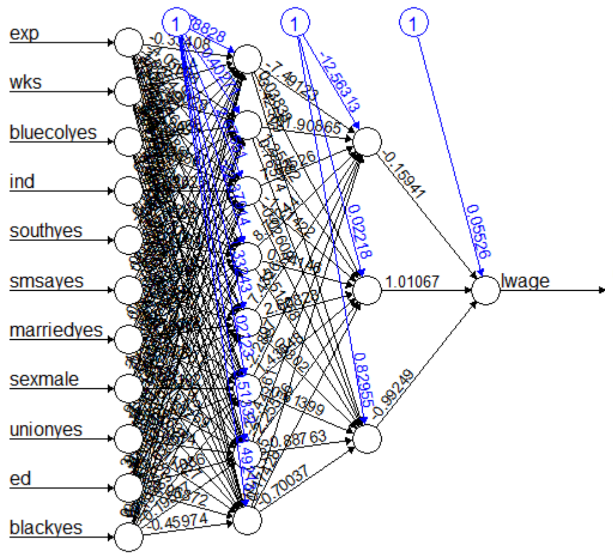
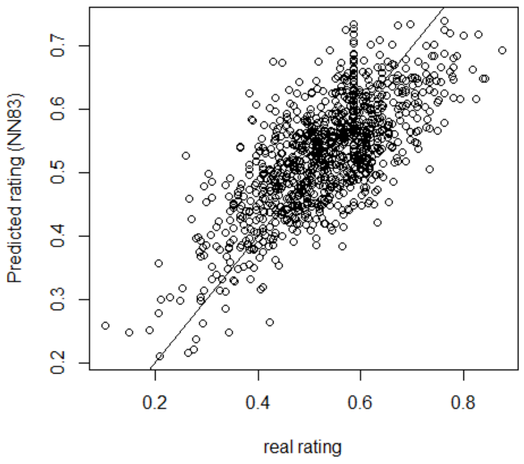


Figure 7Predicted and real values of the test set (NN83)



illustrated in figures 7 and 8. In terms of the comparison between predicted results and actual values of the test set, there does not seem to be a lot of difference between the two specifications of the networks.

As a final measure in the analysis of these networks, I computed the Mean squared error of both, to compare the results. The first one's is 0.0965 and the second's is 0.0925 – also very similar. I initially intended to use the same data in a linear regression model to compare the results, but I do not think it would make sense considering that the data used is panel data and would require another type of model.

5. CONCLUSIONS

This project has taught me a lot about big data and machine learning, and how truly large the opportunities for its application in economics are.

I have learnt about big data and some of the tools that could become useful to economists, like cross-validation and regularization, as well as new predictive models like CART and neural networks. I have also read about a lot of interesting research that relates the large amount of data now available to economic topics, and how big data has allowed for new questions to be considered that were previously not possible due to data limitations.

Neural networks have turned out to be a method with a lot more complexity than I was expecting, but which is still approachable with some effort. It has taught me about machine learning, from which economists could borrow many methods. My empirical exercise using neural networks consisted of using two neural networks to predict the values of wages for individuals. The predictions were not perfect, but the example helped me put to practice some of the concepts which I had previously described, which was very interesting. I am sure that using more complex datasets and taking more advantage of *neuralnet*'s customization options could result in very interesting results.

6. ANNEX

Meaning of the variable names in the dataset *Wages*:

- exp: years of full-time work experience
- wks: weeks worked
- bluecol: blue collar worker (yes/no)
- ind: works in a manufacturing industry (yes/no)
- south: resides in the south (yes/no)
- smsa: resides in a standard metropolitan statistical area (yes/no)
- married: the individual is married (yes/no)
- sex: the gender of the individual (male/female)
- union: whether the individual's wage set by a union contract (yes/no)
- ed: years of education
- black: whether the individual is African-American (yes/no)
- lwage: logarithm of wage

This is the R script detailing everything I used in the empirical exercise (section 4).

#Remove everything that was in the previous workspace and install packages needed: Ecdat and neuralnet

rm(list=ls())

install.packages(c("Ecdat", "neuralnet"))

#Load Ecdat and neuralnet and open the data set of interest, Wages

library(Ecdat)

data(Wages)

#Check the srtucture of the data set and if it contains any missing values

str(Wages)

sum(is.na(Wages))

#Normalise the data frame, now called wagesn, and check the results with str(). Include model.matrix function at the end to turn all the factor variables to dummy variables. After

```
the dummies have been included, I turn the data set back to a data frame instead of a
matrix
normalise <- function(x){
                        if(is.numeric(x)) {
                        return((x-min(x))/(max(x)-min(x)))
                        } else {
                        x
                        }
                        }
wagesn <- as.data.frame(lapply(Wages, normalise))
wages.matrix <-
model.matrix(~exp+wks+bluecol+ind+south+smsa+married+sex+union+ed+black+l
wage, data=wagesn)
wages <- as.data.frame(wages.matrix)

#Check the results of the normalisation. Comparing wages and wagesn, I can see that the
variables that were factor have been renamed, and a 1 indicates the option that has been
added to its name
head(wages)
head(wagesn)

#Create the test and train subsets
set.seed(42)
index <- sample(1:nrow(wages), round(0.75*nrow(wages)))
trainNN <- wages[index, ]
testNN <- wages[-index, ]

#Fit the neural network with the training subset
library(neuralnet)
set.seed(24601)
f <-
as.formula(lwage~exp+wks+bluecolyes+ind+southyes+smsayes+marriedyes+sexmale
+unionyes+ed+blackyes)
NN6 <- neuralnet(f, data=trainNN, hidden=6)
```

```
#Plot and see the results of the trained network
plot(NN6)
NN6$result.matrix
#Predict using the test set
predwages <- compute(NN6, testNN[ ,2:12])
predwages$net.result

#Reverse the normalization process on the predicted output
predwages2 <- (predwages$net.result*(max(Wages$lwage)-
min(Wages$lwage)))+min(Wages$lwage)
#Plot the prediction against the real values of the test sample
plot(testNN$lwage, predwages$net.result, ylab = "Predicted rating (NN6)", xlab =
"real rating")
abline(0,1)

#Retrain the neural network with a different architecture: 8 and 3 hidden nodes, and
repeat all previous steps done with NN6
set.seed(24601)
NN83 <- neuralnet(f, data=trainNN, hidden=c(8,3))
plot(NN83)
NN83$result.matrix
predwages83 <- compute(NN83, testNN[ ,2:12])
predwages83$net.result
predwages832 <- (predwages83$net.result*(max(Wages$lwage)-
min(Wages$lwage)))+min(Wages$lwage)

#Plot the new network
plot(testNN$lwage, predwages83$net.result, ylab = "Predicted rating (NN83)", xlab =
"real rating")
abline(0,1)

#MSE of both neural network specifications, which requires having the test samples
without normalisation
```

```
set.seed(42)
index2 <- sample(1:nrow(Wages), round(0.75*nrow(Wages)))
train2 <- Wages[index2, ]
test2 <- Wages[-index2, ]

MSE.NN <- sum((predwages2 - test2$lwage)^2)/nrow(test2)
MSE.NN83 <- sum((predwages832 - test2$lwage)^2)/nrow(test2)
```

7. BIBLIOGRAPHY

Alice, M., 2015. Fitting a neural network in R, neuralnet package. [online] R-bloggers. Available at: https://www.r-bloggers.com/fitting-a-neural-network-in-r-neuralnet-package/. Accessed 4 March 2018.

Anastasiadis, A., Magoulas, G. and Vrahatis, M., 2005. New globally convergent training scheme based on the resilient propagation algorithm. *Neurocomputing*, 64: pp.253-270.

Choi, H. and Varian, H., 2012. Predicting the Present with Google Trends. *Economic Record*, 88: pp.2-9.

Croissant, Y. (2016). Package ‘Ecdat’. [online] Available at: https://cran.r-project.org/web/packages/Ecdat/Ecdat.pdf. Accessed 2 May 2018.

Einav, L. and Levin, J., 2014. The Data Revolution and Economic Analysis. *Innovation Policy and the Economy*, 14: pp.1-24.

Fritsch, S., Guenther, F., Suling, M. and Mueller, S., 2016. Package 'neuralnet'. [online] Available at: https://cran.r-project.org/web/packages/neuralnet/neuralnet.pdf. Accessed on 1 May 2018.

Grogan, M., 2018. neuralnet: Train and Test Neural Networks Using R. [online] Michael Grogan. Available at: http://www.michaeljgrogan.com/neural-network-modelling-neuralnet-r/. Accessed 15 Mar. 2018.

Günther, F. and Fritsch, S., 2010. neuralnet: Training of Neural Networks. *The R Journal*, [online] 2(1). Available at: https://journal.r-project.org/ Accessed on 16 March 2018.

Haykin, S., 1999. Neural networks: a comprehensive foundation. 2nd ed. London: Prentice-Hall International.

Haykin, S., 2009. Neural Networks and Learning Machines. 3rd ed. Pearson Prentice Hall.

Hornik, K., Stinchcombe, M. and White, H., 1989. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5): pp.359-366.

James, G., Witten, D., Hastie, T. and Tibshirani, R., 2013. An introduction to statistical learning. New York: Springer.

Kuan, C. and White, H., 1994. Artificial neural networks: an econometric perspective. *Econometric Reviews*, 13(1): pp.1-91.

Lippmann, R., 1987. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(2): pp.4-22.

McCulloch, W., and Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4): pp.115-133.

Portilla, J., 2016. A Beginner's Guide to Neural Networks with R. [online] Kdnuggets.com. Available at: https://www.kdnuggets.com/2016/08/begineers-guide-neural-networks-r.html Accessed 4 Mar. 2018.

Rosenblatt, F., 1957. The Perceptron: a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

Sagar, C., 2017. Creating & Visualizing Neural Network in R. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2017/09/creating-visualizing-neural-network-in-r/. Accessed 3 March 2018.

Samuel, A., 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3): pp.210-229.

Varian, H., 2014a. Big Data: New Tricks for Econometrics. *Journal of Economic Perspectives*, 28(2): pp.3-28.

Varian, H., 2014b. Beyond Big Data. *Business Economics*, 49(1): pp.27-31