Predicting NBA players salary with Neural Networks coded from scratch

Nicolas García and Carlos Villa

June 2023

1 Abstract

In this project, we first implement in Python a neural network following the theory learned in the lessons, and then we use it to predict the salaries of NBA players based on their statistics. We compare results using different models, training algorithms and regularization techniques, and we finally compare the accuracy of our network with the one of a simple linear regression model.

2 Task

Our first goal is to code from scratch a shallow, fully connected, feed-forward neural network, following the ideas presented in the theoretical lectures of the course *Reti Neurali*.

Once that is done, we will train it on a dataset consisting of statistics of NBA players through the years 1995-2017. Adapting the dataset to train the model will be one of the problems we encounter.

Then, we will see how and why the model performs depending on the values of the parameters, the training method (GD or SGD), regularization... All of that with the purpose of getting a result at least at good as the one we would obtain by performing a linear regression.

3 Methods

3.1 Implementing the Neural Network

We have based all the models we use on a shallow, fully connected, feed-forward neural network.

The base Neural Network

First, we implemented a Neural Network that trains with full-batch backpropagation (standard Gradient Descent).

We created a `NeuralNetwork` class that takes the arguments `input_size`, `hidden_size` and `output_size`. We could have not used the argument `output_size` since for our purpose of doing a regression we just need it to be 1. However, we decided to add it since it was very similar to implement and that allows the network to be trained for a wider variety of problems.

The network initializes its weights following the Glorot's initialization.

After that, the `forward` and `backward` methods are defined in the conventional way: they both take the whole training dataset as an argument, `forward` computes the output of the network using the sigmoid activation function in the hidden neurons (it seemed to perform slightly better than ReLU) and ReLU in the output neurons; and `backward` takes also `lr` as an argument, and actualizes the weights after computing the errors following the standard backpropagation algorithm.

Then comes the `train` method, which takes as argument the training data, the number of epochs, and the learning rate. It simply consists of a `for` loop that for each epoch, does forward and backward propagation to the training data, and from time to time prints the epoch number and loss to monitor the training process.

Finally, there is a `predict` method, intended to be used after training the network, that returns the output after forward propagation on a given dataset (the validation dataset).

Implementing Mini-Batch Stochastic Gradient descent and regularization

After this, we wanted to implement mini batch SGD and L1 and L2 regularization. We wanted to do this in the same `NeuralNetwork` class, in a way that it allows us to choose whether to use GD or SGD, and the type of regularization (L1, L2 or none).

So we added a `create_minibatches(mb_size,X_train,y_train,shuffle=True)` method and added to the `train` method the arguments `mb_size=0,reg=0,reg_lambda=0`. The argument `reg` can take values 0, 1 or 2, depending on the type of regularization the user wants to apply (0 corresponds to none). We set all these arguments to 0 as default since, that way, if the user does not specify the desired values for these arguments, then standard GD is performed, with no regularization (we coded the method so that if `mb_size==0` then it sets `mb_size=X_train.shape[0]`, the size of the dataset, which is the same as performing full-batch propagation). .

Finally we modified the `backward` and `train` methods to perform SGD and update the weights with the desired regularization factor in each case.

3.2 Training on the NBA dataset

The dataset has been downloaded from the website Kaggle, and all the data in the dataset has been extracted from Basketball-Reference.

Preparing the dataset for the Neural Network

Before using our dataset in our neural network, we must make a number of changes in order to have homogeneous data samples. For example, eliminating irrelevant columns for our data analysis or those with empty data. We also do not consider data for those players who have played less than 15 matches and we unify the name of the teams because along the years some of them have changed their name.
After renaming some variables, we create a new column indicating the ratio of games started to games played, thus merging these two variables and having values between 0 and 1.
As there are players who have played a large number of matches and therefore have higher statistics than those who have only played for a short time, we transform all statistics into per-match statistics and not in total.
Finally, the dataset has total statistics for a player if he was on two teams in one season but we want to look at data on specific teams, so we eliminate the 'TOT' value because it wouldn't work for our analysis.

Evaluating and monitoring the performance of the models

To evaluate the performance of the models, we use the mean squared error (MSE) loss function.

To monitor the training process, we modify the `train` method in the `NeuralNetwork` class so that it saves the loss at each epoch in an array, and plots it afterwards. We add the argument `plot=0` to the possible inputs of the method, which can take values 0 (to not display anything) or 1 (to make such a plot). We also add `x_test` and `y_test` as arguments in order to plot also how the validation error evolves.

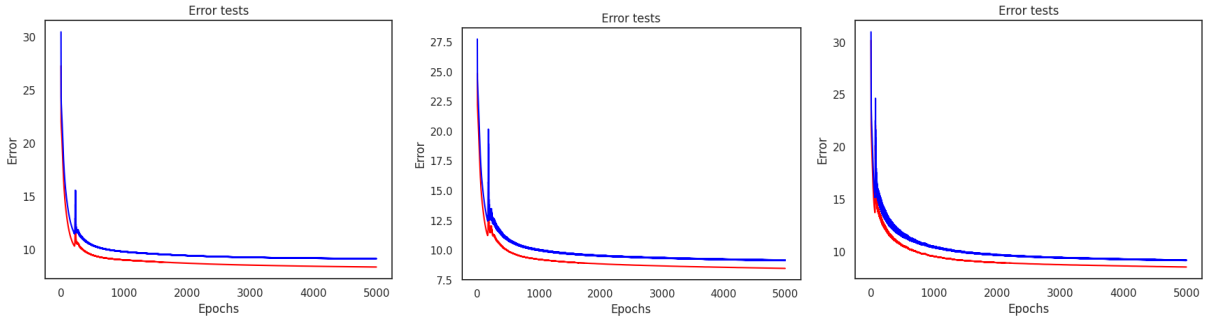
Feature selection for linear regression

In order to try to improve the results of our linear regression, we performed a previous study of the correlation of the variables with the salary. Firstly, we create a plot of the salaries in function of the categorical variables like "Season", "Position" and "Team" seeing that these variables have an effect on the salaries. Secondly, We create a correlation matrix to see which variables have less correlation coefficient with the salary, choosing those with a coefficient less than 0.1, in particular the variables'3PAr', 'FTr', 'ORB%', 'STL%', 'BLK%', '3P%', so we finally eliminated them.

4 Results

We first try the most basic model: full batch propagation, with no regularization. We find that a good value for the learning rate is 0.05. We also experiment with different numbers of neurons in the hidden layer, which do not yield very different results (see Figure 1).

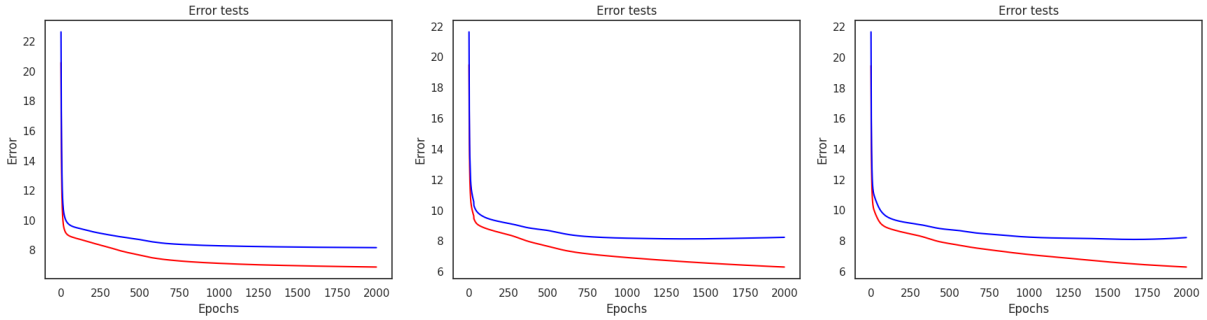
We measure the performance of the models with the mean squared error. These models ended with a training error of 8.38, 8.45 and 8.50, respectively; and a validation error of 9.16, 9.10, 9.18. The difference is not significant, however, we could say that the model with less neurons performed slightly better, since the graph shows a faster convergence.



(a) 8 hidden neurons (b) 20 hidden neurons (c) 50 hidden neurons

Figure 1: Evolution of the loss function during the training (5000 epochs). In red, the loss on the training set; in blue, the loss on the validation set.

Now, we use minibatch gradient descent to see if we can get better results. We have trained the same models as before with minibatch gradient descent, using minibatches of 32 datapoints, and we have reduced the learning rate to 0.005.



(a) 8 hidden neurons (b) 20 hidden neurons (c) 50 hidden neurons

Figure 2: Evolution of the loss function during the training (2000 epochs) with minibatches of size 32.

Training the neural networks like this works clearly better: in less than half the epochs we get similar (slightly better) results: the training errors after 2000 epochs are 6.83, 6.27 and 6.28, respectively, and the validation errors are 8.14, 8.21 and 8.19.

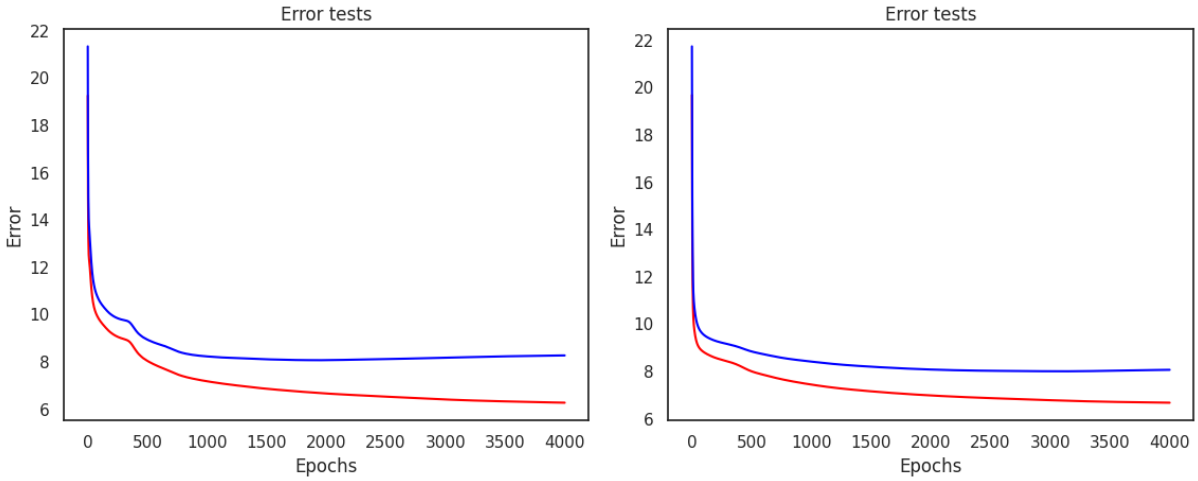
It is worth noticing that in the two last models, some overfitting was starting to appear: the best validation error was a few hundred epochs before 2000, with 8.11 and 8.19 being the least values for the error (for the models with 20 and 50 hidden neurons, respectively).

That is not at all a significant overfitting but anyways, it leads us to try, for example, the model with 20 neurons, during more epochs and using L1 and L2 regularization (see Figure 3).

Both work better than not using regularization: overfitting appears later, specially with L2 regularization. We could have set λ to higher values in order to prevent even more the overfitting process. However, doing this resulted in a too slow convergence and it took a lot of time to obtain the desired loss values. Now, with these values of λ, the best validation errors we reach are 8.02 (L1, at epoch 2200 approximately, after which overfitting starts) and **7.98** (L2, at around epoch 3000). That is the best value that we can obtain with our Neural Network. We can interpret it better taking the square root: our model usually predicts a salary that differs from the real one by around $\sqrt{7.98} = 2.82$ million dollars.

Now, we may compare it to the results obtained with a simple linear regression. Applying linear regression directly to the training dataset yields a MSE of 10.84, considerably higher than any of the results obtained before.

After exploring the correlation between all the variables in order to do some feature engineering, we want to see if even after that, our network performs at least as well as linear regression. After that we obtain an error in the linear regression of 10.82, practically equal to the previous linear regression model and still higher than any result obtained before, and with an R-squared value of approximately 0.57 which means that it is not a very accurate model but not a bad one either.



(a) L1 regularization with $\lambda = 0.0005$.

(b) L2 regularization with $\lambda = 0.001$.

Figure 3: Evolution of the loss function during the training with minibatch gradient descent and regularization.

5 Conclusions

The neural network performed better in general than linear regression, thanks to the versatility of this kind of models, that allows to solve a huge variety of problems by choosing the right architecture and parameters.

This, however, does not mean that linear regression would not be a good solution for our problem. It would be worth studying (but deviates from the main focus of this project) the performance of a linear regression model after an exhaustive feature engineering. Maybe dedicating the same efforts to the linear regression problem as we did with neural network would yield as good results.

Anyways, we can say that our goal has been achieved: we built a neural network from scratch that allowed us to solve a problem better than a direct linear regression.