Autoencoder

Mariajose Franco Orozco
mfrancoo@eafit.edu.co
EAFIT University
Mathematical Engineering
Medellín, Colombia

I. Introduction

Autoencoders were first introduced in 1986 by [1]. These are a type of neural network that consists of compressing the input and trying to get the most important features about it in order to reproduce it. It has 2 processes: encoder and decoder. The first step is that the autoencoder receives an input, then, it will encodes the input into a lower dimensional space in order to make a compressed representation of it. Finally, the decoder will pass that compressed representation to the original space [2].

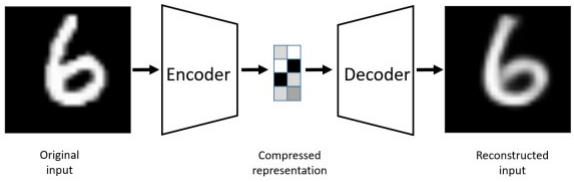

Fig. 1. Autoencoder example representation

Fig. 1 represents the process of an autoencoder. The idea behind them is that by forcing the network to learn a compressed representation of the input data, it will learn to extract the most important features of the data while filtering out noise and other extraneous information [2].

Autoencoders have a wide range of applications, including anomaly detection, image and audio compression, clustering, classification, generative models, and much more, proving to be effective in most of them [2].

II. Theoretical Framework

Autoencoders consist o 3 layers that form 2 neural networks mentioned above: the encoder and the decoder. The first layer is the input layer which is the one that receives the input data and has some related weights. In the input layer, the compression of the data is done and passed to the next layer. The second layer is the hidden layer, which is the one that receives the encoded input and is in charge of decoding the data and passing it to the last layer. Then, the output layer is the one that gives us the output, which is supposed to be similar to the initial input.

In this autoencoder, the bias is included. It is first initialized for every neuron in every layer in zeros. Then, for updating the bias, I performed the formula presented **here** which is the following:

$$\frac{\delta \varepsilon}{\delta b} = \frac{\delta \varepsilon}{\delta y}\frac{\delta y}{\delta V}\frac{\delta V}{\delta b} \tag{1}$$

but $\frac{\delta V}{\delta b} = 1$, resulting in

$$\frac{\delta \varepsilon}{\delta b} = \frac{\delta \varepsilon}{\delta y}\frac{\delta y}{\delta V} \tag{2}$$

Finally,

$$\Delta b_k = -e_k \phi_k^{'}(V_k) \tag{3}$$

In the next section, the dataset and methodology used are explained.

III. Methodology

The dataset used in this implementation was the *Rice Dataset Cammeo and Osmancik* which can be downloaded from **here**. It consists of a study made in 2019 by [3] in which 2 types of rice: Cammeo and Osmancik, were analyzed and some characteristics were extracted from each type of them resulting in 3810 data and 7 attributes for each one of them [4]. The attributes are:

- **Area:** Returns the number of pixels within the boundaries of the rice grain.
- **Perimeter:** Calculates the circumference by calculating the distance between pixels around the boundaries of the rice grain.
- **Major Axis Length:** The longest line that can be drawn on the rice grain, i.e. the main axis distance, gives.
- **Minor Axis Length:** The shortest line that can be drawn on the rice grain, i.e. the small axis distance, gives.
- **Eccentricity:** It measures how round the ellipse, which has the same moments as the rice grain, is.
- **Convex Area:** Returns the pixel count of the smallest convex shell of the region formed by the rice grain.
- **Extent:** Returns the ratio of the region formed by the rice grain to the bounding box pixels.

The output of the dataset is the class of rice, which is 'Cammeo' or 'Osmancik'. This was a categorical output, for making it a numerical output, 2 dummy variables were considered for the output:

- **Cammeo:** 1 if it is and 0 if it is not.
- **Osmancik:** 1 if it is and 0 if it is not.

This way, the resultant dataset consists of 7 inputs and 2 outputs, all of them numerical variables, ready to be

normalized.

As mentioned before, 3 layers were considered. The input layer has 7 neurons due to the inputs of the dataset, as well as the output layer, which has the same amount of neurons as the input layer. Then, for the hidden layer, 2 cases were considered: encoding in low-dimensions and high-dimensions. For low-dimensions, the hidden layer needs to have fewer neurons than the input layer, and for high dimensions needs to have more layers than the input layer. Also, 3 learning rates were considered: 0.2, 0.5, and 0.9, and a total of 50 epochs were performed. In the next section, the experimentation performed is presented.

IV. Experimentation

Two cases were considered for this experimentation: low-dimensions and high-dimensions. As it was mentioned before, the dataset used has 7 input variables, which means that the input and output layers will contain 7 neurons, but the hidden layer will vary the number of neurons in it. Also, bias was included in this project. For purposes of analyzing the effect of the bias, I performed the low-dimensions encoding with bias and without, as well as the high-dimensions encoding.

A. Low-Dimensions

For encoding in low-dimensions, the first thing we need to do is to encode the data, but for this compression, it is not straightforward to know the optimum number of hidden neurons. Because of that, I considered every architecture possible in order to see which one will be the best for encoding in low dimensions, varying it from 0 to 6 neurons in the hidden layer. The results obtained are presented in Fig. 2.

index	Learning Rate	Architecture	Average Energy
0	0.2	7,1,7	0.1059698372
1	0.2	7,2,7	0.130116713
2	0.2	7,3,7	0.1639425148
3	0.2	7,4,7	0.3212848483
4	0.2	7,5,7	0.4782901246
5	0.2	7,6,7	0.4831552033
6	0.5	7,1,7	0.096042698
7	0.5	7,2,7	0.0964405358
8	0.5	7,3,7	0.0970248462
9	0.5	7,4,7	0.1046265864
10	0.5	7,5,7	0.1284727875
11	0.5	7,6,7	0.1825585032
12	0.9	7,1,7	0.0950888468
13	0.9	7,2,7	0.0951815173
14	0.9	7,3,7	0.0947108486
15	0.9	7,4,7	0.0933305007
16	0.9	7,5,7	0.0938780673
17	0.9	7,6,7	0.0926293539

Fig. 2. Results per architecture (including bias)

The errors of the different architectures mentioned in Fig. 2 are presented in Figs. 3, 4, and 5. The graphs on the left are the errors of the different models with bias, and on the right, are the results obtained without including bias. In these Figures is evident the difference of including bias.

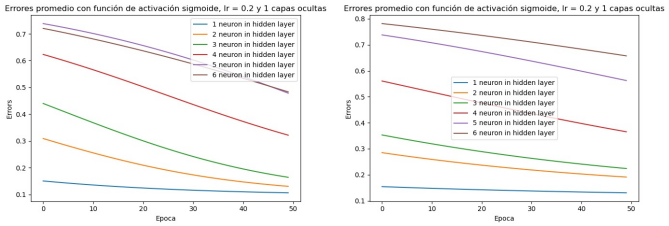

Fig. 3. Encoding in low-dimensions with vs without bias and a learning rate of 0.2

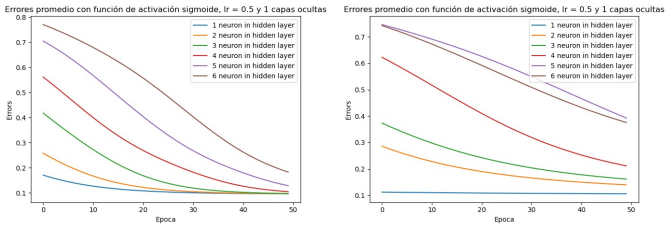

Fig. 4. Encoding in low-dimensions with vs without bias and a learning rate of 0.5

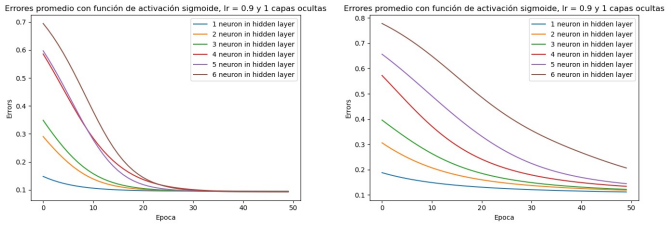

Fig. 5. Encoding in low-dimensions with vs without bias and a learning rate of 0.9

B. High-Dimensions

For the high-dimension case, we have infinite possibilities of determining how many neurons to include in the hidden layer, but for this project, I decided to consider a maximum of 6 neurons more in the hidden layer than the input layer. Because of this, the hidden neurons will vary between 8 and 13 neurons. The results are the following:

index	Learning Rate	Architecture	Average Energy
0	0.2	7,8,7	0.7987634844
1	0.2	7,9,7	0.8682497575
2	0.2	7,10,7	0.9056906043
3	0.2	7,11,7	0.9397220248
4	0.2	7,12,7	0.9607974581
5	0.2	7,13,7	0.9535359275
6	0.5	7,8,7	0.6107038372
7	0.5	7,9,7	0.753947826
8	0.5	7,10,7	0.8652544499
9	0.5	7,11,7	0.9609414417
10	0.5	7,12,7	0.9439588006
11	0.5	7,13,7	0.9674346952
12	0.9	7,8,7	0.2669708833
13	0.9	7,9,7	0.2516633133
14	0.9	7,10,7	0.3236089134
15	0.9	7,11,7	0.5583837549
16	0.9	7,12,7	0.7827477913
17	0.9	7,13,7	0.7875797644

Fig. 6. Results per architecture (including bias)

The errors of the different architectures mentioned in Fig. 6 are presented in Figs. 7, 8, and 9. The graphs on the left are the errors of the different models with bias, and on the right, are the results obtained without including bias. In these Figures is evident the difference of including bias.

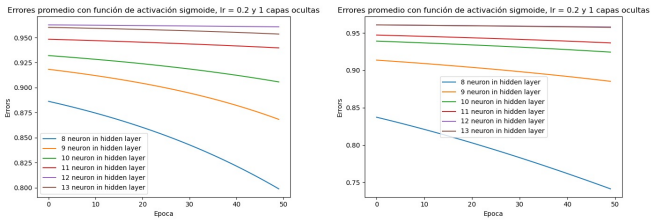

Fig. 7. Encoding in high-dimensions with vs without bias and a learning rate of 0.2

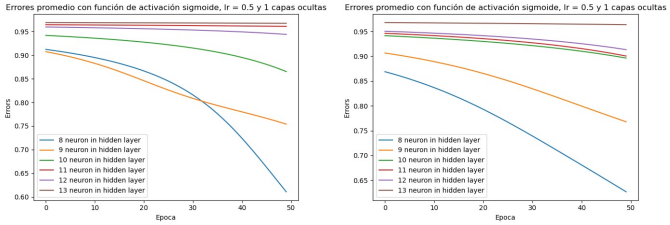

Fig. 8. Encoding in high-dimensions with vs without bias and a learning rate of 0.5

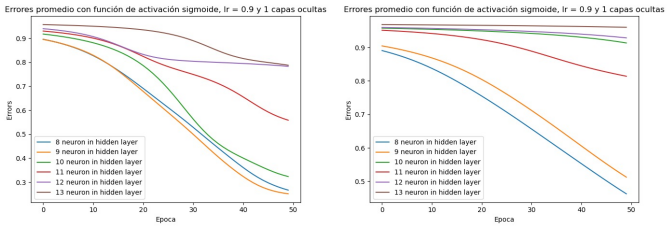

Fig. 9. Encoding in high-dimensions with vs without bias and a learning rate of 0.9

V. Results

After obtaining the different results with the autoencoder, the next step is to determine which is the best and the worst model. And then, the final step is to train an MLP with the code obtained in the low-dimension best performance, and the high-dimension best performance.

A. Low Dimensions

TABLE I
BEST AND WORST MODELS FOR ENCODING IN LOW-DIMENSIONS

	Architecture	Learning Rate	Average Error
BEST	[7, 6, 7]	0.9	0.092629
WORST	[7, 6, 7]	0.2	0.483155

The best model is the one that has 6 neurons in the hidden layer and a learning rate of 0.9, and the worst model has the same architecture but with a learning rate of 0.2.

Then, using the sklearn library, an MLP was trained with the output obtained with the best model shown in Fig. I. Then, test data was passed to this MLP in order to see if it can predict the output. The RMSE obtained was **0.4955** and the loss curve is presented in the following figure.

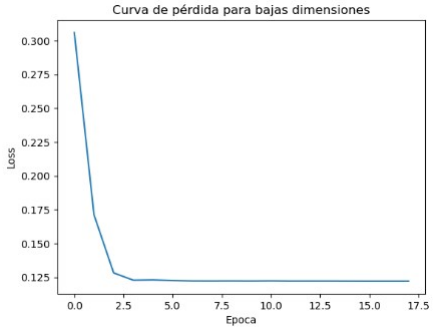

Fig. 10. Loss curve for low dimensions

B. High Dimensions

TABLE II
BEST AND WORST MODELS FOR ENCODING IN HIGH-DIMENSIONS

	Architecture	Learning Rate	Average Error
BEST	[7, 9, 7]	0.9	0.251663
WORST	[7, 13, 7]	0.5	0.967434

The best model is the one that has 9 neurons in the hidden layer and a learning rate of 0.9, and the worst model has 13 neurons in the hidden layer and a learning rate of 0.5.

As mentioned above, sklearn library was used to train an MLP with the output obtained with the best model shown in Fig. II. Then, test data was passed to this MLP in order to see if it can predict the output. The RMSE obtained was **0.4947** and the loss curve obtained is shown below.

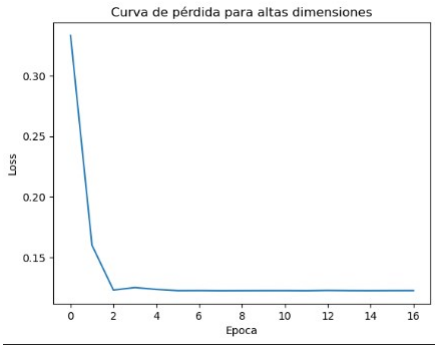

Fig. 11. Loss curve for high dimensions

VI. Conclusions

In Figs. I and II we can see that the best models in both cases were the ones with higher learning rates, as we could see in the last class project about MLP. Also, we can notice that, even though the results obtained encoding with high-dimensions were not bad, the best results obtained were when encoding with low-dimensions. According to this, what I can understand is that our data variables considered were representative of the data and it is not needed to increment the space to obtain more features about them.

When training the MLP with the outputs of the autoencoder in low and high dimensions, I could notice that the results were not the desired outputs, but they were not bad. The loss curve for low-dimensions significantly decreases and converges to 0.125 approximately. For high-dimensions, the loss curve converges more rapidly and to a lower value of the loss.

In the last class project, the bias was not included in the MLP, but for this project it was. It definitely makes an improvement in the prediction, as we could see in the figures where the errors are presented, when using bias, the error decreases much more.

A great understanding was achieved with this project about the idea behind an autoencoder and finally, we can conclude that the autoencoder implemented is able to learn the input data correctly in order to extract the most relevant features of it.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations Error Propagation," Tech. Rep., 1986.

[2] W. H. Lopez Pinaya, S. Vieira, R. Garcia-Dias, and A. Mechelli, "Autoencoders," *Machine Learning: Methods and Applications to Brain Disorders*, pp. 193–208, 2019.

[3] M. Koklu and I. Cinar, "Classification of Rice Varieties Using Artificial Intelligence Methods," *International Journal of Intelligent Systems and Applications in Engineering*, 2019.

[4] U. M. L. Repository, "Rice (Cammeo and Osmancik) Data Set," 2019. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Rice+{%}28Cammeo+and+Osmancik{%}29{#}