NEURAL NETWORKS

January 26, 2020

Alberto Grillo ID: 4229492
Alberto Ghiotto ID: 4225586

***Abstract*— Neural networks are multi-layer networks of connected neurons that can be used to classify things and make predictions. This biologically-inspired programming paradigm enables a computer to learn from observational data.**
They are widely used in applications where they can be trained via a dataset such as speech recognition and image analysis. The connections between the neuron are modeled as weights, which transform the input that will be given to the next neuron.
Neural networks and deep learning currently provide the best solutions to many problems such as the one mentioned above. In this projects we will use the neural network toolbox by MATLAB [1] to analyze different implementation of neural networks.

I. INTRODUCTION

In this assignment some guidelines on how to develop the project were given, along with some reference to various different dataset available in literature to be used as a benchmark for our tests. For the first two tasks, we selected a bigger dataset and a smaller one in order to test performances with different parameters and different dimensions of the datasets. The third task required to use the MNIST handwritten character database.
The first two tasks consisted in following two tutorials to get acquainted with the neural network toolbox by MATLAB. Both tutorials make use of shallow neural networks, the first [4] shows how to fit a function to some data while the second one [5] shows how to classify patterns of data.
The third and last task consisted in implementing a simple autoencoder network (i.e a multi-layer perceptron neural network with one input layer, one hidden layer and one output layer). To do so it was suggested to employ the two ad hoc MATLAB function *trainAutoencoder* and *encode* to develop the encoder and apply it to our data.
The workflow of the third experiment can be summarized as follows (taking the MNIST dataset as a benchmark):

- Given that the dataset is relatively large take only a subset of one tenth of the original dimension.
- Extract a training set with only 2 classes.
- Train an autoencoder on the new, reduced training set.
- Apply the obtained encoder to encode the different classes.
- Plot the data using the provided plotcl" function.

II. BACKGROUND

In this section, a brief explanation of the implemented principles will be given. The detailed theory can be found in the class slides.
As mentioned above, neural network are computational models based on networks of neurons built of several layers, each composed of homogeneous units. These simple units, called neurons, take an input(either from another neuron or from the input layer), make a decision (based on some function) and give an output. Learning in neurons occurs by slow modification of weights, a procedure that gradually changes them accordingly to experience acquired in the past.
In figure 1 it can be seen an example [6] of a simple neural network where, starting from the left we can see:

1) The input layer of our model in orange.
2) The first hidden layer of neurons in blue.
3) The second hidden layer of neurons in magenta.
4) The output layer of the model in green.

Usually a neural network with multiple layers has:

1) One input layer, which takes the external inputs and spreads them to the first hidden one.
2) One (or more) hidden layer. They are called hidden because they do not take and do not give any input and output from/to the outside of the network.
3) The output layer which provides the data to the outside of the network.

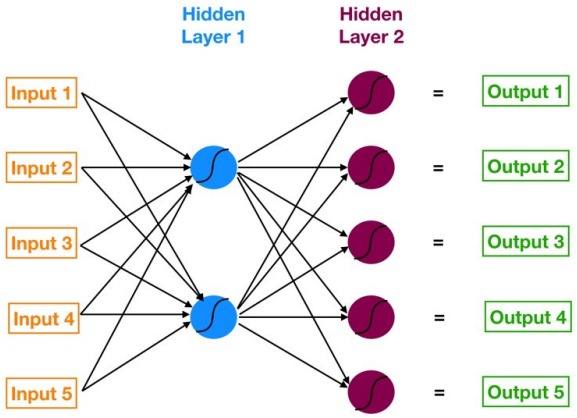

Fig. 1. A diagram of a simple neural network with five inputs, 5 outputs, and two hidden layers of neurons.

As the number of hidden layers rises, the results get better but the computational time becomes longer. Since error back-propagation does not work well with very deep structures, multi-layer networks put a limit to the process of learning. Shallow networks are considered to be sufficient to avoid this problem.

Another interesting application of neural networks are autoencoders. An Autoencoder is an unsupervised artificial neural network that learns how to efficiently compress and encode data then learns how to reconstruct the data back from the reduced encoded representation to a representation that is as close to the original input as possible [7]. The simplest autoencoder network, visible in figure 2, is a multi-layer neural network which has one input layer, one hidden layer (n.hidden units¡n.inputs), and one output layer (n.output units=n.inputs). It is trained using the same pattern as both the input and the target An autoencoder learns an internal,

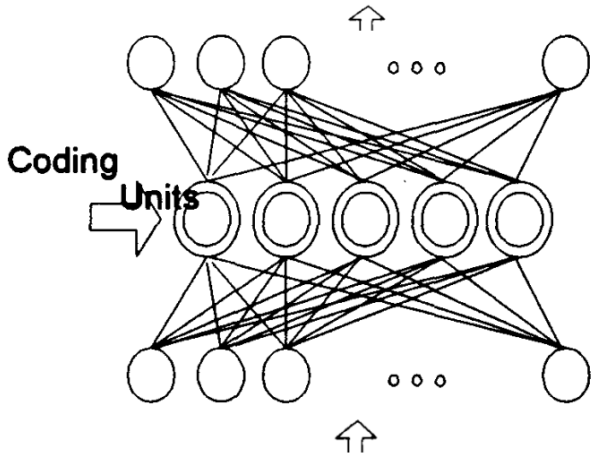

Fig. 2. A simple autoencoder.

compressed representation for the data, giving as output the value of its hidden layer. What we hope is that similar patterns will have similar representations, performing a sort of clustering with the evaluated classes.

III. Implementation and Results

Using the MATLAB Neural Network toolbox, we implemented neural networks and tested their performances in three different learning tasks.

The first two tasks were developed according to the tutorials suggested by the guidelines of the project, while the third one was about the implementation of an autoencoder for ”self-supervised” learning.

It’s important to highlight that the toolbox assumes a different convention for the data than the one used in previous projects. We usually assumed patterns as rows and variables as columns, while the toolbox functions expect patterns as columns and variables as rows. Hence, we had to make some minor compatibility modifications.

Moreover, when inputs are to be classified into N different classes, the target vector has to be normalized by arranging a matrix with N rows and as many columns as the number of observations. For each column, the element corresponding to the relative class label is set to 1 and the others to 0. As suggested by the tutorial, we decided to divide the datasets in three subsets:

1) Training, 70% of the dataset.
2) Test, 15% of the dataset.
3) Validation, 15% of the dataset.

A. Task 1: Fit data with Neural Network

The aim of this task was to use a neural network to fit a function to a dataset. For function fitting, we used a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. As explained in the tutorial, this can be done in two ways:

1) By selecting the Neural Network Fitting App in the GUI of the toolbox.
2) By generating scripts from the GUI and using command-line functions.

Since we used different datasets than the ones provided by the toolbox, we adapted them to the requested convention.

It is possible to validate network performance by displaying the regression plot, the overall performance plot and the error histograms.

The regression plots show the network outputs with respect to targets for training, validation, and test sets. For a perfect fit, the data should fall along a 45 degree line, where the network outputs are equal to the targets. The fit is reasonably good if the regression R values, measuring the correlation between outputs and targets, are near 1. An R value of 1 means a close relationship while a value of 0 implies a random relationship.

The histograms can give an indication of data points where the fit is significantly worse than the majority of data, called outliers. When outliers are valid data points, but different from the rest of the data, the network extrapolates (i.e estimates some values based on extending a known sequence of values or facts beyond the area that is certainly known) also with them, decreasing the overall performance.

The performance plot gives information about the overall performance in terms of final mean-square error and epochs. A good performance is achieved when:

1) The mean-square error is small.
2) The test set and the validation set errors have similar characteristics.
3) No significant overfitting has occurred by the iteration where the best validation performance occurs.

If the performance on the training set is good but the test set one is significantly worse, overfitting may have occurred. In this case, reducing the number of neurons can improve results. On the other hand, if training performance is poor, increasing the number of neurons can lead to better results.
We decided to use the "Building Energy" and the "Chemical Sensors" datasets from the toolbox. The former can be used to train a neural network to estimate the energy use of a building from time and weather conditions while the latter can be used to estimate one sensor signal from eight other sensor signals.
Our results, computed with different number of hidden neurons (10 or 30), are shown in the figures 3-14.
As we can see in the figure 3 (10 hidden units) and in the figure 4 (30 hidden units), the "Building Energy" dataset performance is slightly better when we use more hidden units. Nevertheless, when we increase the number of hidden units on a smaller dataset like the "Chemical Sensor" one, the overall performance is worse due to overfitting: this is in evidence in the figure 5 (10 hidden units) and in the figure 6 (30 hidden units).
The regression plots (fig. 7, 8, 9 and 10) and the error histograms(fig. 11, 12, 13, 14) show results coherent with our observation on overall performances: increasing the number of neurons produces worse performances on smaller dataset, but improves them on the bigger dataset.

B. Task 2: Classify with feedforward multi-layer networks

In this task we employed a neural network to classify datasets by recognizing patterns.
In order to recognize patterns, we made use of a two-layer feedforward network, with a sigmoid transfer function in the hidden layer, and a softmax transfer function in the output layer.
We generated the script from the GUI and used command-line functions to implement this task because we wanted to add a function to adapt other datasets to the toolbox convention.
It is possible to validate network performance by displaying the confusion matrix plots.
The confusion matrices plots show classification results for training, testing, validation set and the results for the three kinds of data combined. For sake of simplicity, in this report we presented only the confusion matrix of the combined subsets.If the network outputs are accurate, the number of correct classifications will be high while the number of incorrect classifications will be low.
Moreover, it is possible to plot and analyze overall performances and error histograms in the same way of task 1.

For this task, we used the "Thyroid" dataset from the toolbox and the "Wine" dataset from the UCI Machine Learning Repository. The former can be used to create a neural network that classifies patients thyroid as normal, hyperfunctioning or subnormal functioning, while the latter can be used to create a neural network that classifies wines from three wineries in Italy based on constituents found through chemical analysis.
By checking the confusion matrices produced from the "Thyroid" dataset (figure 15 and figure 16), we can see that the more the units are, the better the results are. Again, since the "Wine" dataset is smaller, the improvement is almost neglectable (figure 17 and figure 18).

C. Task 3: Autoencoder

In this last task, we implemented an autoencoder using the appropriate toolbox functions in a MATLAB script. The results of the self-supervised training on the datasets are plotted using the MATLAB function "plotcl" as suggested by the project guidelines.
This function will plot a 2D representation of the clustered data. We can appreciate that, when there are "easy" classes to distinguish, the dots are linearly separate, while results are worse for classes that have very similar features pattern.
For this last task, we used the MNIST dataset in order to create an autoencoder that learns to classify by clustering similiar data. We used 2 hidden units in order to plot the classification between two or three classes in 2D, while 3 hidden units are required for a 3D representation of the clusters.
By observing the plots we can recognize that the dots are linearly separate when the classes are easy to distinguish, e.g. handwritten digits 1 and 5 (figure 19). Instead, with "difficult" pairs, e.g. handwritten digits 0 and 8, the results are worse (figure 20) or even really bad when they are very similiar, e.g. handwritten digits 4 and 9 (figure 21). In the the figures 22 and 23 we show the results of the same three class comparison (handwritten digits 0, 1, 5) with respectively 2 or 3 hidden units. When we use 3 hidden units, the autoencoder is able to build a 3D representation of the clusters.

IV. CONCLUSIONS

Each time a neural network is trained, it can yield different results due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, one will have to retrain it several times. If the network's performance are not satisfying, it's possible to improve them by doing one of the following:

1) Train it again.
2) Increase the number of neurons.
3) Get a larger training data set.
4) Reset the initial network weights and biases to new values and train again.
5) Try a different training algorithm.

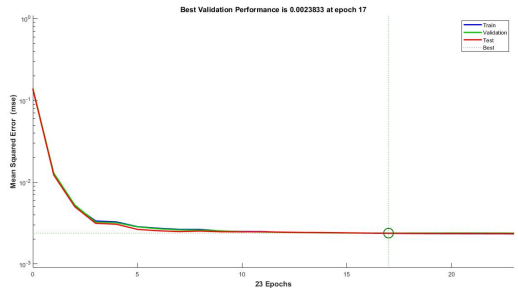

Fig. 3. Performance with 10 neurons on ”Building Energy” dataset.

1

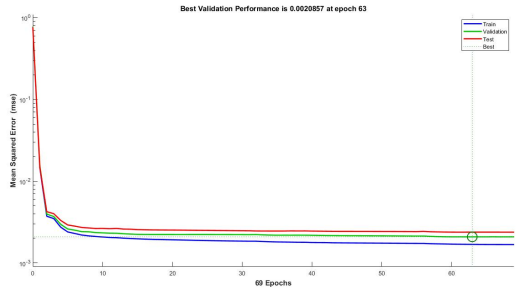

Fig. 4. Performance with 30 neurons on ”Building Energy” dataset.

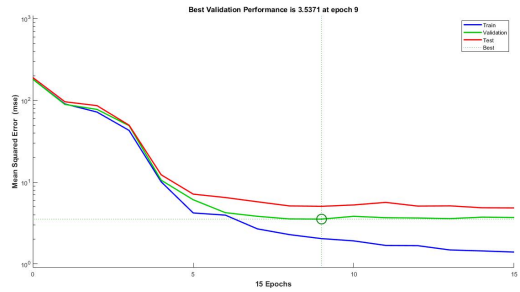

Fig. 5. Performance with 10 neurons on ”Chemical Sensors” dataset.

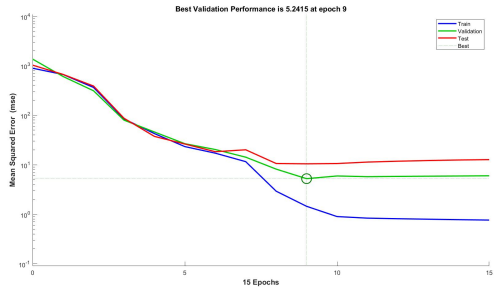

Fig. 6. Performance with 30 neurons on ”Chemical Sensors” dataset.

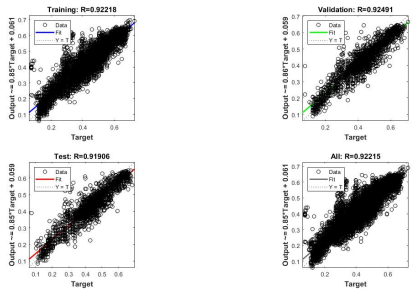

Fig. 7. Regression values with 10 neurons on ”Building Energy” dataset.

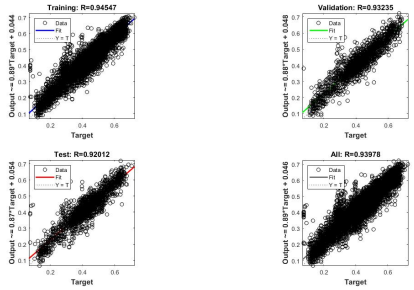

Fig. 8. Regression values with 30 neurons on ”Building Energy” dataset.

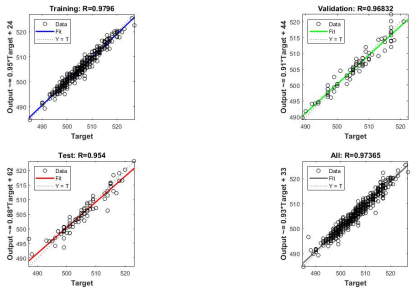

Fig. 9. Regression values with 10 neurons on ”Chemical Sensors” dataset.

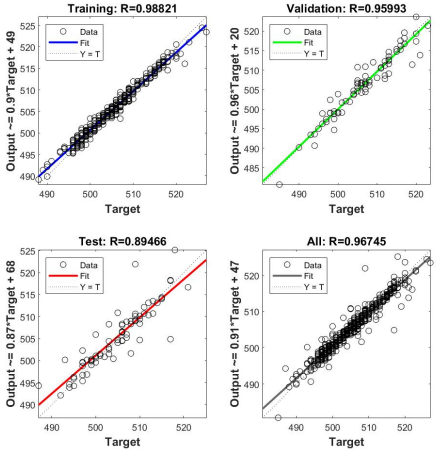

Fig. 10. Regression values with 30 neurons on ”Chemical Sensors” dataset.

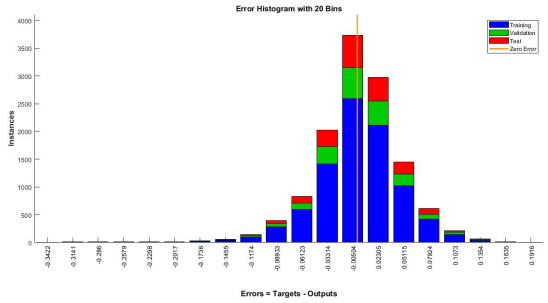

Fig. 11. Error histograms with 10 neurons on ”Building Energy” dataset.

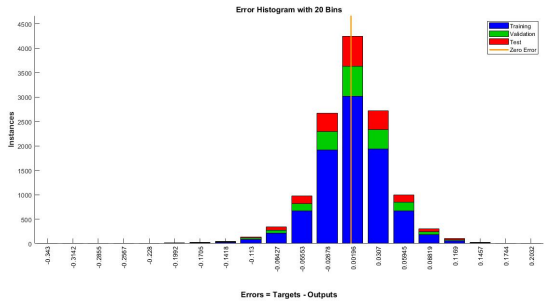

Fig. 12. Error histograms with 30 neurons on ”Building Energy” dataset.

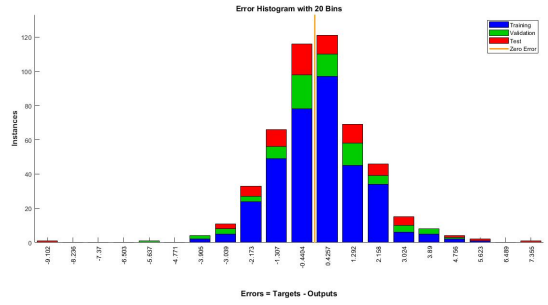

Fig. 13. Error histograms with 10 neurons on ”Chemical Sensors” dataset.

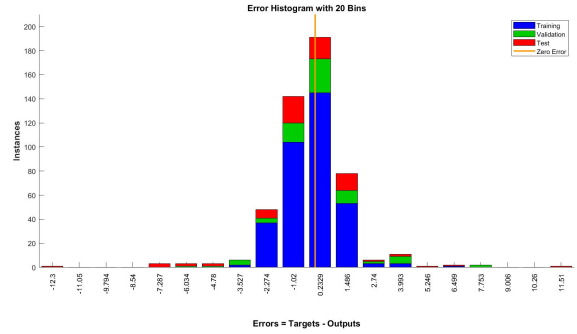

Fig. 14. Error histograms with 30 neurons on ”Chemical Sensors” dataset.

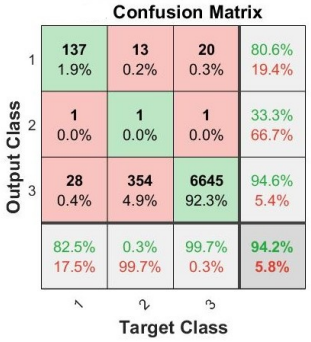

Fig. 15. Total confusion matrix with 5 neurons on ”Thyroid” dataset.

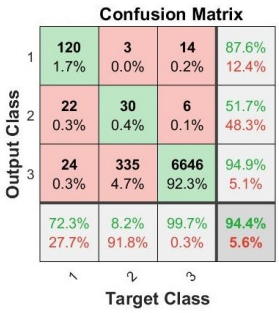

Fig. 16. Total confusion matrix with 30 neurons on ”Thyroid” dataset.

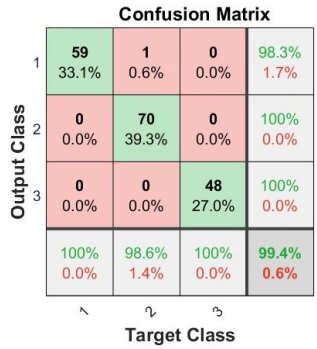

Fig. 17. Total confusion matrix with 5 neurons on ”Wine” dataset.

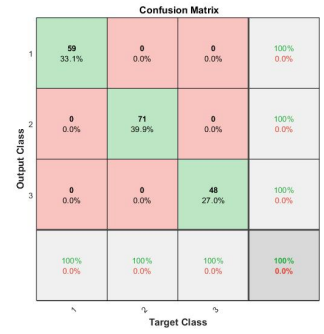

Fig. 18. Total confusion matrix with 30 neurons on ”Wine” dataset.

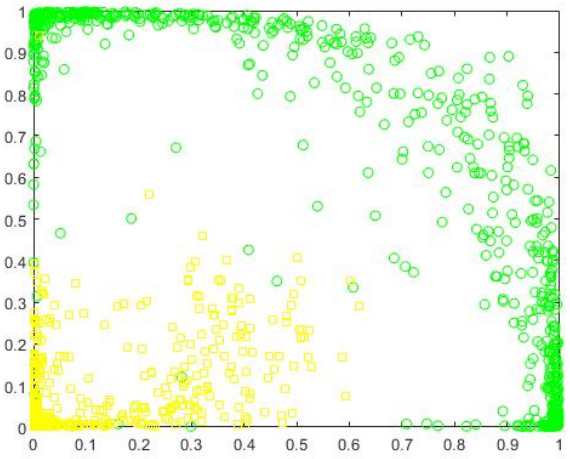

Fig. 19. Autoencoder clustering results with digit 1 and 5.

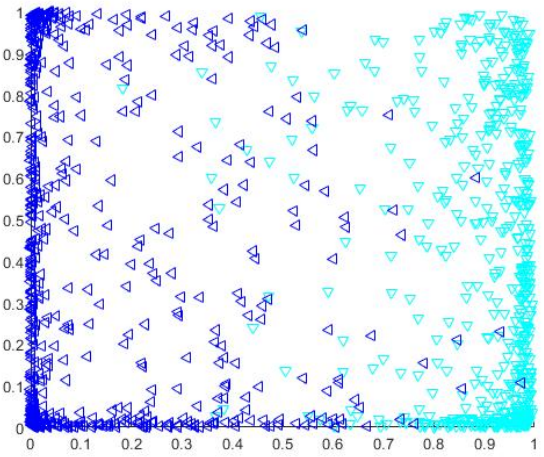

Fig. 20. Autoencoder clustering results with digit 0 and 8.

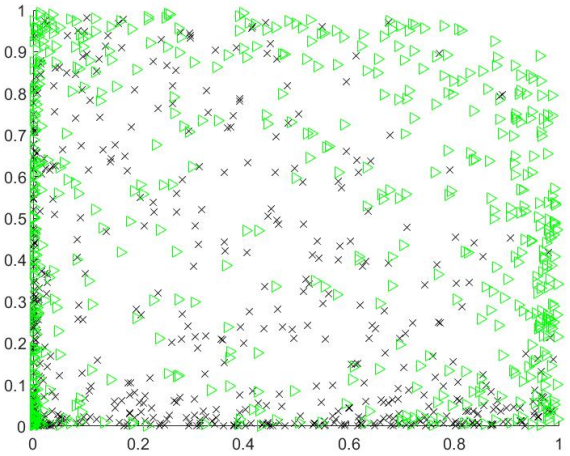

Fig. 21. Autoencoder clustering results with digit 4 and 9.

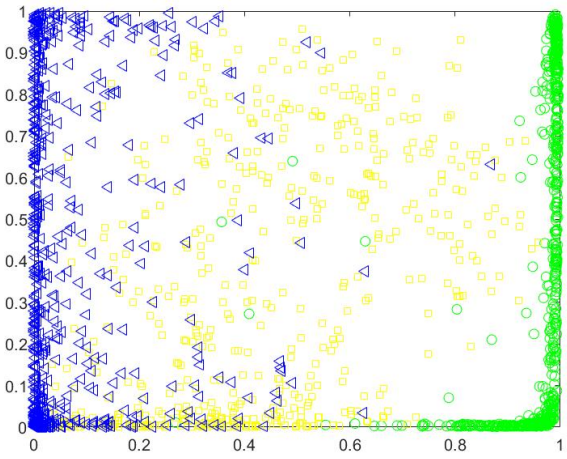

Fig. 22. Autoencoder clustering results with digit 0, 1 and 5 with 2 hidden neurons

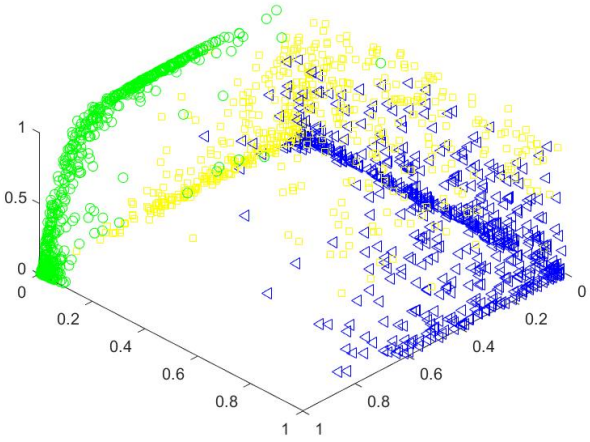

Fig. 23. Autoencoder clustering results with digit 0, 1 and 5 with 3 hidden neurons.

REFERENCES

[1] Neural Network Toolbox
https://it.mathworks.com/help/deeplearning/index.html?s_tid=CRUX_lftnav

[2] Wine dataset
http://archive.ics.uci.edu/ml/datasets/Wine

[3] Iris Dataset
http://archive.ics.uci.edu/ml/datasets/Iris

[4] Tutorial 1
https://it.mathworks.com/help/deeplearning/gs/fit-data-with-a-neural-network.html

[5] Tutorial 2
https://it.mathworks.com/help/deeplearning/gs/classify-patterns-with-a-neural-network.html

[6] Neural Network example
https://towardsdatascience.com/understanding-neural-networks-19020b758230

[7] Autoencoder
https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used\-for-part-1-3e5c6f017726