Machine Learning: MLP

Ilyass Taouil

02/11/2017

1 The Dataset

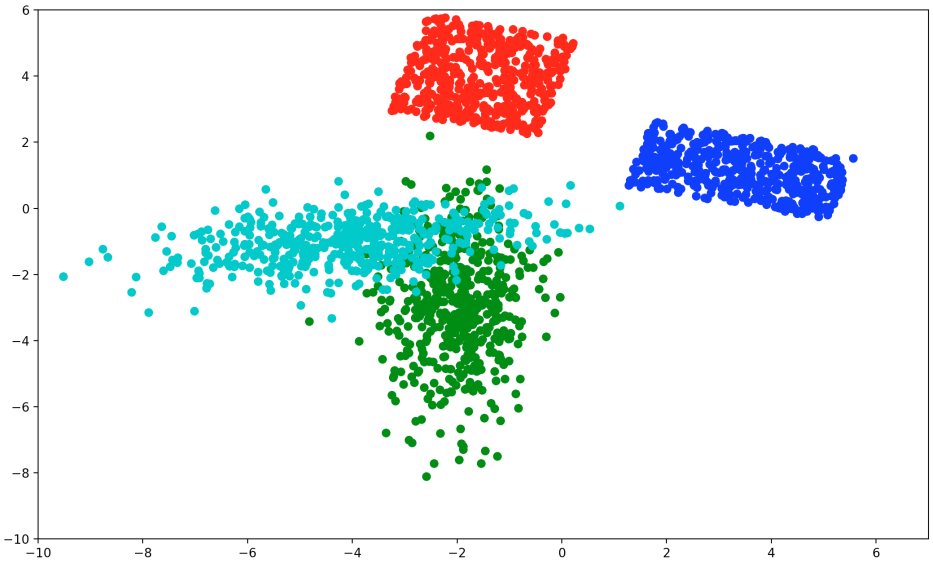

Figure 1: MLP's dataset generated.

The figure above shows the whole dataset example generated for the Multi Layer Perceptron (MLP). There are four distinguishable classes where a colour is assigned to each one of these. Respectively, red for class C1, blue for class C2, green for class C3 and cyan for class C4.

In the following two subsections the steps, methodologies and utilities used to reach the final result showed in Figure 1 will be explained with the help of small portion of code snippets.

1.1 C1 and C2

Both class C1 and class C2, which I will be referring to as class 1 and class 2 when needed, are two rectangular shapes whose distance, by imposed constraint, is at least 2 as to avoid overlapping between the two set of points. Having acknowledged the rectangular shape of the classes, generating a uniform set of data for both collections is a matter of defining the range of values for the X and Y sets, for both class 1 and class 2. These are the chosen ranges:

- C1:
  - X range: [2, 5]
  - Y range: [1, 4]

- C2:
  - X range: [1, 3]
  - Y range: [-5, -1]

The next logical step is to actually generate the data based on the range choices. To perform this step the numpy **np.random.uniform** is used, which spawns samples within the given interval with an equal likelihood, giving us a uniform range of data. The routine is used for both the set X and Y, for both classes. The obtained result, however, is not our desired output as the numpy function will return four different vectors of data, when we need two multidimensional array representing our class 1 and class 2. The numpy function **np.column_stack** is used for the matter, which will take in the separate **x** and **y** vectors and stack them together to make our class 1 and class 2 two different 2-D array.

C1 and C2 data generation code:

```
# Generate points for class1
c1 = stack(uniform(cf.data["c1_x_low"], cf.data["c1_x_high"], cf.data["size"]),
           uniform(cf.data["c1_y_low"], cf.data["c1_y_high"], cf.data["size"]))

# Generate points for class2
c2 = stack(uniform(cf.data["c2_x_low"], cf.data["c2_x_high"], cf.data["size"]),
           uniform(cf.data["c2_y_low"], cf.data["c2_y_high"], cf.data["size"]))
```

The figure below shows the obtained result in generating the points for C1 and C2:

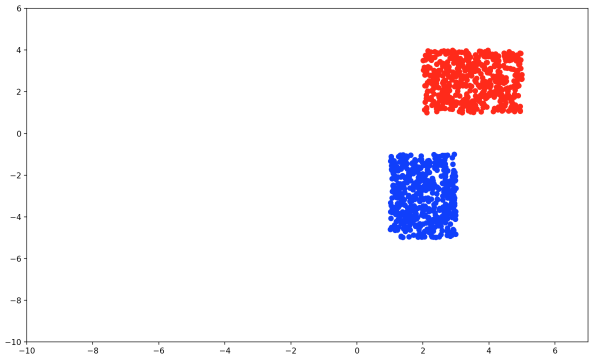

Figure 2: C1 and C2.

To obtain the final result for our C1 and C2 classes as shown in Figure 1, we will need to rotate the generated points with a certain angle. We hence define a rotation matrix **R** by taking advantage of Python's built in **math** module to compute trigonometric functions:

```
# Rotation matrix
R = np.array([ [math.cos(cf.data["angle"]), -math.sin(cf.data["angle"])],
               [math.sin(cf.data["angle"]), math.cos(cf.data["angle"])] ])
```

The angle choice denoted in the code via a configuration parameter is of a value of **-75** deg.

The last step in our classes generation consists in actually multiplying our C1 and C2 matrices by the previously defined rotation matrix. The numpy **np.dot** is used for that.

```
# Rotate class1 and class2 points
c1 = np.dot(c1, R)
c2 = np.dot(c2, R)
```

The final result is the following:

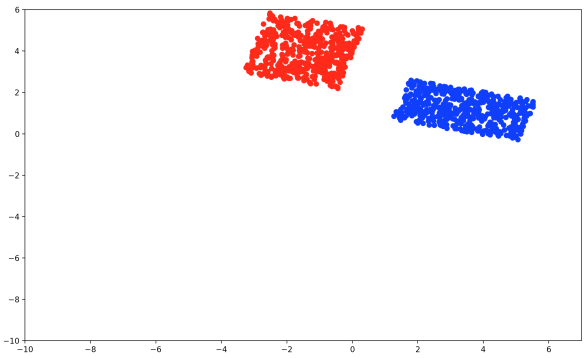

Figure 3: Cl and C2 with rotation matrix applied.

1.2 C3 and C4

The data for C3 and C4 are generated by leveraging the **numpy.random.multivariate_normal** numpy routine, which returns our set of data for class 3 and class 4. The matrices used for the covariance and mean for C3 and C4 are the following:

- C3 (Mean and Covariance):

$$\mathbf{m} = \begin{bmatrix} -2 & -3 \end{bmatrix}$$
$$\mathbf{c} = \begin{bmatrix} 0.5 & 0 \\ 0 & 3 \end{bmatrix}$$

- C4 (Mean and Covariance):

$$\mathbf{m} = \begin{bmatrix} -4 & -1 \end{bmatrix}$$
$$\mathbf{c} = \begin{bmatrix} 3 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

The result obtained is showed in the figure below:

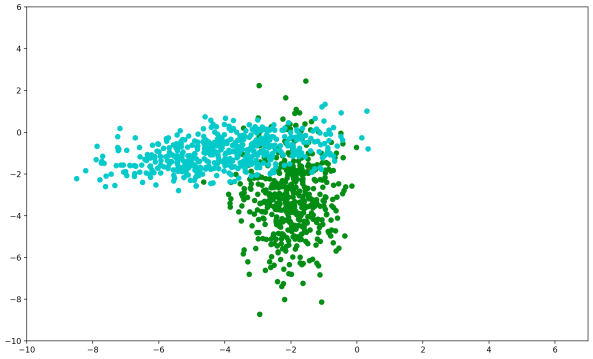

Figure 4: C3 and C4 dataset.

2 Multi Layer Perceptron

2.1 Network Size

The network size experimentation, where only the number of hidden neurons are changed with a constant learning rate of **0.01**, has been carried on three MLP instances, the following:

- 2 Neurons
- 7 Neurons
- 14 Neurons

2.1.1 2 Neurons

The first instance performed really well with only two neurons and with a relatively small learning rate as shown in the confusion matrix and the performance analysis below:

$$\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 124 & 0 & 0 & 0 \\ 0 & 133 & 0 & 0 \\ 2 & 0 & 118 & 16 \\ 0 & 0 & 18 & 89 \end{pmatrix} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \end{array}$$

The **precision** value showing how many classified elements are relevant and the **recall** value displaying how many of the classified instances are relevant are shown below for each class:

- C1:
  - Precision : 124 / 126 = 98%
  - Recall : 124 / 124 = 100%
- C2:
  - Precision : 133 / 133 = 100%
  - Recall : 133 / 133 = 100%
- C3:
  - Precision : 118 / 136 = 87%
  - Recall : 118 / 136 = 87%
- C4:
  - Precision : 89 / 105 = 85%
  - Recall : 89 / 107 = 83%

What we derive from the above results is that the MLP is able to classify extremely well both C1 and C2, however the performances decrease by almost 13%/15% when it comes to classify correctly C3 and C4, and this was somehow expected due to the overlapping distribution of class 3 and class 4.

2.1.2 7 Neurons

In the following instance the MLP did not perform as well as the first one. The analysis below goes onto demonstrating that and the possible causes.

$$\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 110 & 0 & 0 & 0 \\ 0 & 133 & 0 & 0 \\ 10 & 17 & 108 & 0 \\ 41 & 25 & 5 & 51 \end{pmatrix} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \end{array}$$

The **precision** value showing how many classified elements are relevant and the **recall** value displaying how many of the classified instances are relevant are shown below for each class:

- C1:
  - Precision : 110 / 161 = 68%
  - Recall : 110 / 110 = 100%
- C2:
  - Precision : 133 / 175 = 76%
  - Recall : 133 / 135 = 98%
- C3:
  - Precision : 108 / 113 = 96%
  - Recall : 108 / 135 = 80%
- C4:
  - Precision : 51 / 51 = 100%
  - Recall : 51 / 121 = 42%

Class 1 and class 2 have a lot of false positives classified as part of the class. Class 3 and class 4 present a remarkable misclassification of truthful instances to the class, hence, a lot of (**False Negatives**).

It is worth noting that the seen variations might be due to the different set of training data and weights used for the training step, as the these are randomly generated every time the Neural Network is run.

2.1.3 14 Neurons

The confusion matrix for the following instance is shown below:

$$\begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 133 & 0 & 0 & 0 \\ 0 & 130 & 0 & 0 \\ 3 & 0 & 93 & 15 \\ 11 & 5 & 1 & 109 \end{pmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

Precision and recall values for each class are shown below:

- C1:
  - Precision : 133 / 147 = 90%
  - Recall : 133 / 133 = 100%
- C2:
  - Precision : 130 / 135 = 96%
  - Recall : 130 / 130 = 100%
- C3:
  - Precision : 93 / 94 = 99%
  - Recall : 93 / 111 = 84%
- C4:
  - Precision : 109 / 124 = 88%

– Recall : 109 / 126 = 87%

C1 and C2 remain by far the best classified classes due to the ease in their linear separation, while C3 and C4 are still the most difficult ones to separate, with a lot of **False Negatives**.

2.1.4 Conclusion

Overall we noticed the good performance of the three instances in classifying class 1 and class 2, and their difficulty in achieving the same for C3 and C4 for questions already presented. To improve the given scenario it would be possible to put a constraint when these classes are generated as done with C1 and C2. Nonetheless, we acknowledge the fact that this is not usually possible in real world problems, and trade-offs need to be found.

2.2 Learning Rate

The best MLP out of the ones introduced in the **Network Size** section is the instance with the two neurons, in terms of errors on the training set, hence, the following subsection will be based on it using the following three learning rates:

- 0.01
- 0.02
- 0.03

The following plots displays the error performance on the training set for the three chosen learning rates.

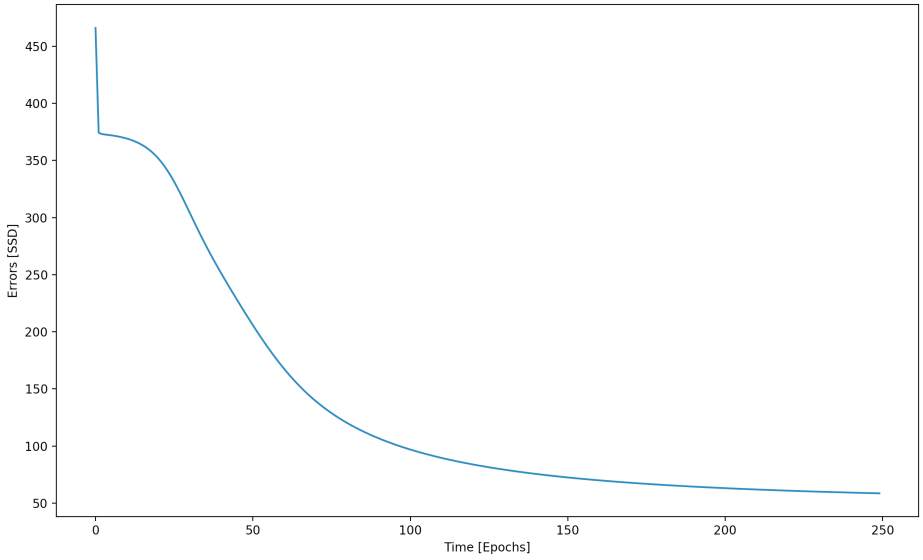

Figure 5: MLP training error with a 0.01 eta.

It is observable how by increasing the learning rate even if by a minimum amount the time needed to reach a convergence decreases with the growth of this one. Nonetheless, the quality of the MLP's generalisation does not necessarily improve.

In the first case, showed in Figure 5, the errors at the end of the training step are around a value of 55, while in the next two cases (Figure 6 and Figure 7) the number of errors increased. We observe as well

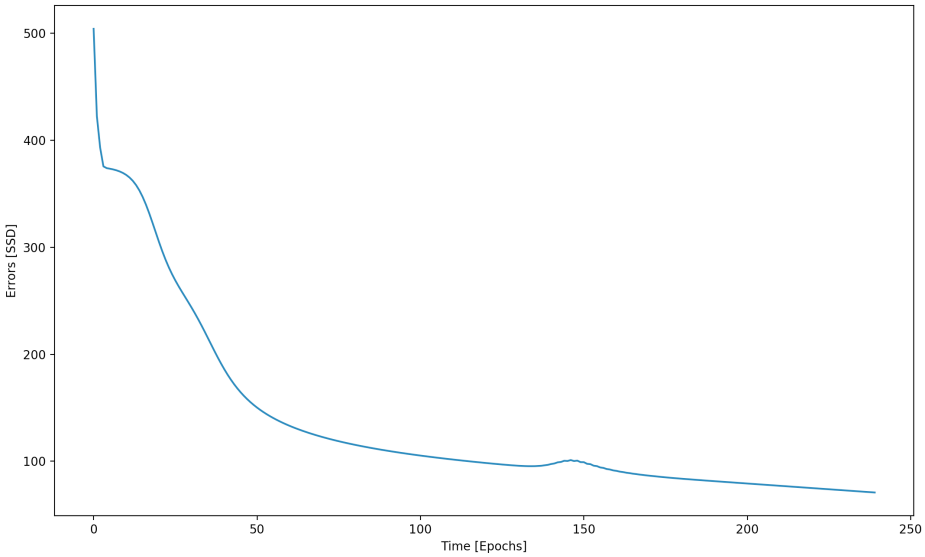

Figure 6: MLP training error with a 0.02 eta.

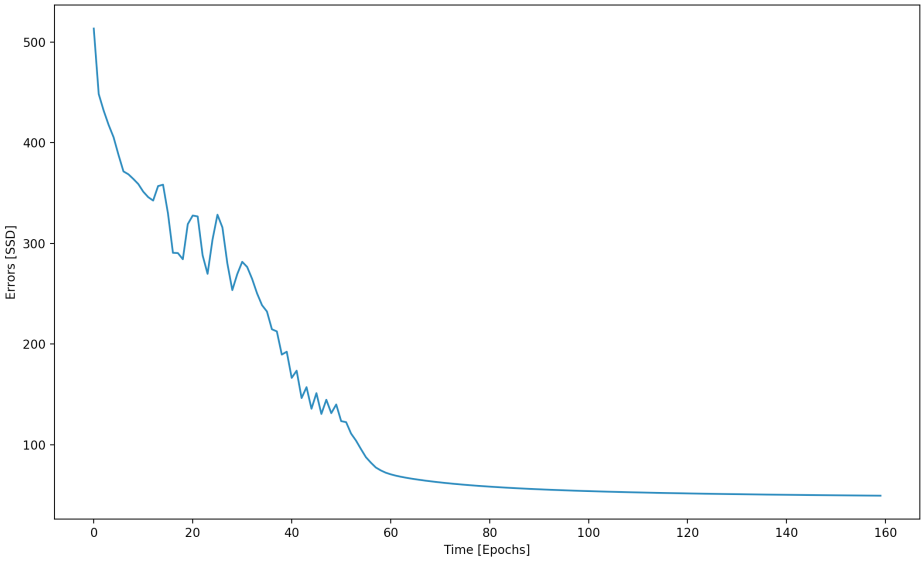

Figure 7: MLP training error with a 0.03 eta.

how with a greater learning rate fluctuations start happening because of the overshoot caused by the increase of the learning rate.

In conclusion, the parameters tuning process is an art in itself and it depends heavily on what the problem being tackled is. However, we can notice a trade-off, where with a greater step size we converge faster but not necessarily to a qualitatively good solution.

2.3 Test Set Plot

The following picture shows the test set classified by the best MLP (2 neurons), with 30 errors made in the test set.

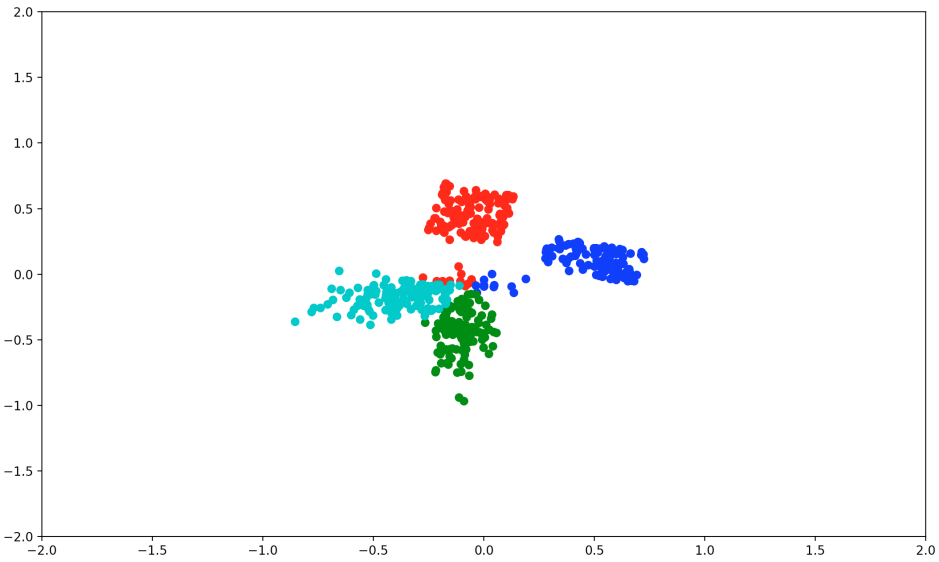

Figure 8: Test set plot with best MLP.