ARISTOTLE UNIVERSITY OF THESSALONIKI

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Neural Networks: Using MLP for the Classification of Dementia through MRI

Aphrodite Latskou

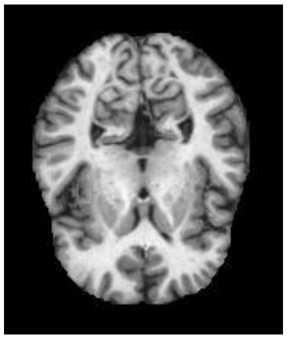
November 2024

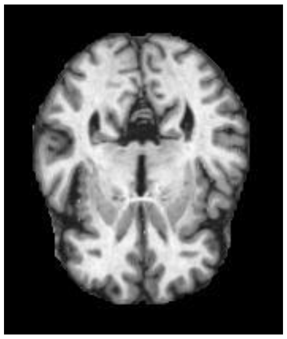
Table of Contents

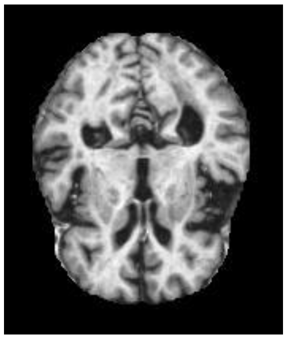
Chapter I: Dataset & Resources

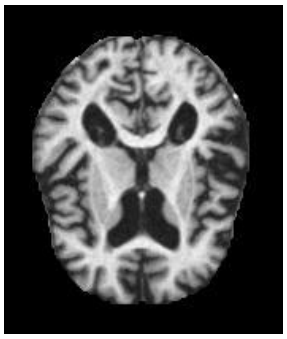
i. Dataset

For this assignment the dataset chosen was the Augmented Alzheimer MRI Dataset from Kaggle. It contains both real and augmented MRI imagery from patients with various dementia levels.


Not Demented


Very Mildly Demented


Mildly Demented


Moderately Demented

More specifically, the dataset contains the following files:

	Original	**Augmented**	**In Total**
Not Demented	3200	9600	12800
Very Mildly Demented	2240	8960	11200
Mildly Demented	896	8960	9856
Moderately Demented	64	6464	6528

Therefore, giving a total of 40384 samples. An important thing to note at this point, is that the samples belonging to each class are not equal. Between the non-demented patients and the moderately demented ones especially, there is a difference of approximately 6000 samples- *half* of the first class. When dealing with imbalanced datasets, it is very likely (*spoiler alert*) that the algorithm might have a bias towards the classes with the most samples, therefore learning to favor those categories when it comes to predicting.

ii. Preprocessing

While preprocessing, I used a number of steps to ensure the efficiency of the algorithm and the credibility of our results. I turned the images to true grayscale (since, even though they seem grayscale already, the image arrays had three channels) to have smaller input sizes and since color was non-existent anyways.

Since both the KNN and the MLP algorithm take one-dimensional inputs, I also flattened the images. The original images Ire of size 200x190, giving a vector size of 38000 for each sample. That was storage-prohibitive, since NumPy requires a contiguous block of memory to allocate the array, therefore when using the entire images, I would run out of memory often and it would also raise the running time of the algorithm. Through trials, the biggest size for time and space efficiency was found to be 100x95. After the flattening, each sample has a vector size of 2350.

The features Ire also scaled using the Min-Max method, since scaling is known to lead to better training. The labels Ire encoding using the One-Hot method in order to work with probabilities for each class in the prediction phase. That means that for example, the Very Mildly Demented class is encoded as [0, 1, 0, 0].

Finally, I split the data into training and testing sets with a percentage of 60% and 40% accordingly, using the train_test_split function from sklearn.

iii. Resources

The algorithms were partly run on a local AMD Ryzen 5 3500U CPU with 4 cores and 16GB RAM and partly using Google Collab with 12.7GB RAM available.

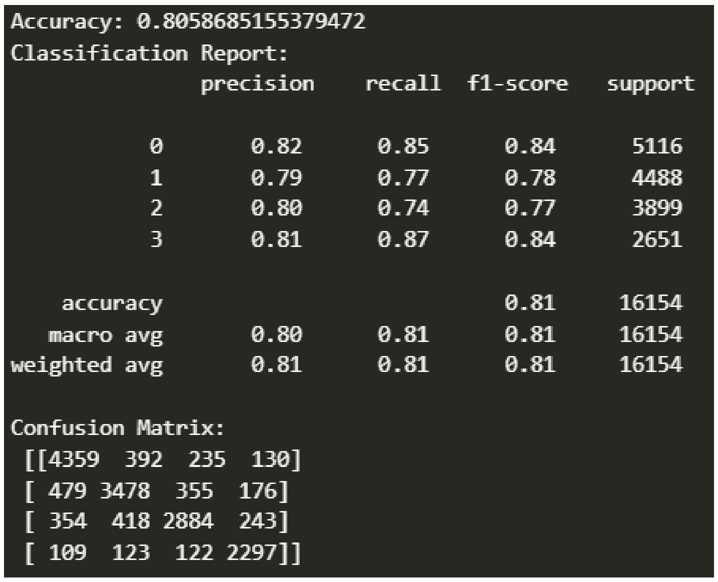
ChatGPT was also used for smaller code snippets (i.e. for plotting and result presentation), as well as the use of the KNN and Nearest Centroid algorithms. I did try the general prompt as input, but the output code was deemed extremely faulty and lacking.

Chapter II: KNN & Nearest Centroid

i. KNN

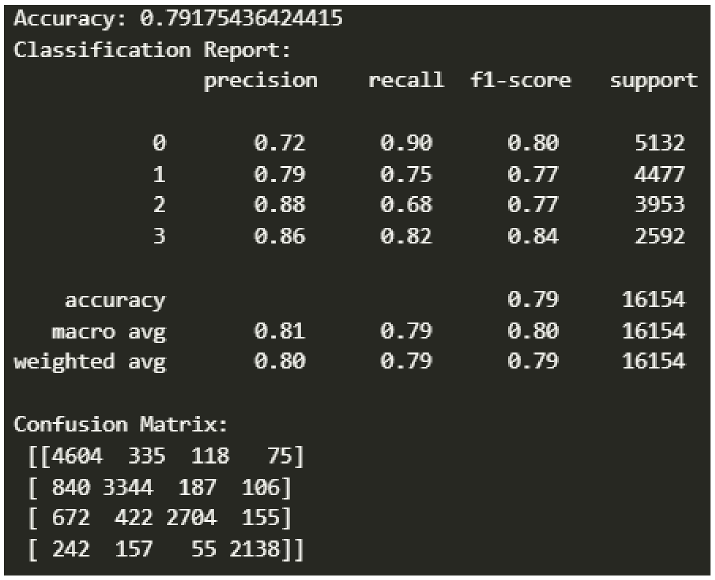
KNN algorithms classify a new sample based on its closest 'neighbors' based on some distance metric. Here, I used the Euclidean distance. The KNearestClassifer from sklearn was used with k = 1 and k = 3 and achieved the following results.

KNN with 1 Neighbor:



```
Accuracy: 0.8058685155379472
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.85      0.84      5116
           1       0.79      0.77      0.78      4488
           2       0.80      0.74      0.77      3899
           3       0.81      0.87      0.84      2651

    accuracy                           0.81     16154
   macro avg       0.80      0.81      0.81     16154
weighted avg       0.81      0.81      0.81     16154

Confusion Matrix:
 [[4359  392  235  130]
 [ 479 3478  355  176]
 [ 354  418 2884  243]
 [ 109  123  122 2297]]
```

KNN with 3 Neighbors:

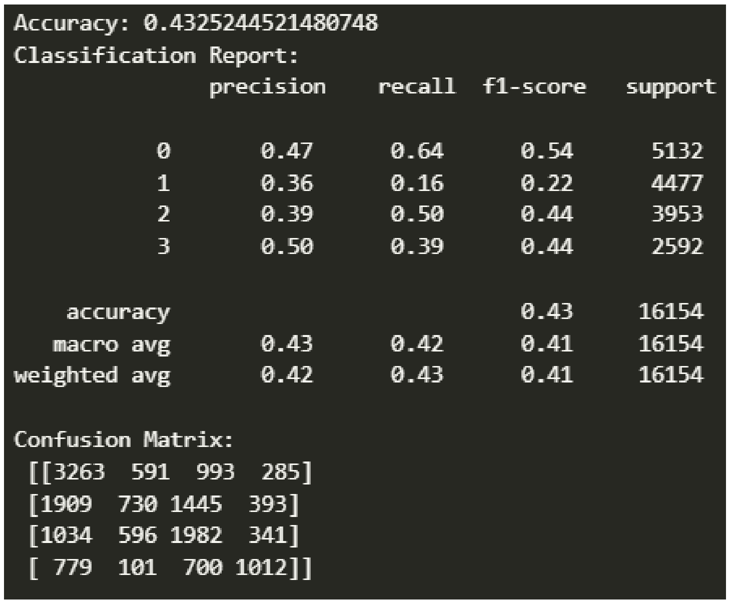


```
Accuracy: 0.79175436424415
Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.90      0.80      5132
           1       0.79      0.75      0.77      4477
           2       0.88      0.68      0.77      3953
           3       0.86      0.82      0.84      2592

    accuracy                           0.79     16154
   macro avg       0.81      0.79      0.80     16154
weighted avg       0.80      0.79      0.79     16154

Confusion Matrix:
 [[4604  335  118   75]
 [ 840 3344  187  106]
 [ 672  422 2704  155]
 [ 242  157   55 2138]]
```

The KNN algorithm performs surprisingly Ill, and notably, similarly either with 1 or 3 neighbors. Specifically, from the confusion matrices, it is clear that the best metrics are achieved on the Non-Demented and Moderately Demented classes, especially on the recall metric that describes the algorithm's ability to correctly classify a sample belonging to that class. That is always important when dealing with medical data.

ii. Nearest Centroid

Nearest Centroid algorithms calculate the centroid of every class (mean of the samples) and classify every new sample based on the distance from those points. The NearestCentroid from sklearn was used and achieved the following results.



```
Accuracy: 0.4325244521480748
Classification Report:
              precision    recall  f1-score   support

           0       0.47      0.64      0.54      5132
           1       0.36      0.16      0.22      4477
           2       0.39      0.50      0.44      3953
           3       0.50      0.39      0.44      2592

    accuracy                           0.43     16154
   macro avg       0.43      0.42      0.41     16154
weighted avg       0.42      0.43      0.41     16154

Confusion Matrix:
 [[3263  591  993  285]
 [1909  730 1445  393]
 [1034  596 1982  341]
 [ 779  101  700 1012]]
```

Clearly, the results are very different from the previous algorithms, with Nearest Centroid performing a lot worse in comparison for all 4 classes, and especially for the VeryMildDemented class.

Chapter III: MLP from scratch

To gain a better understanding of the Multilayer Perceptron, the model was built from scratch using primarily the NumPy library in python. The MLP algorithm takes the following steps:

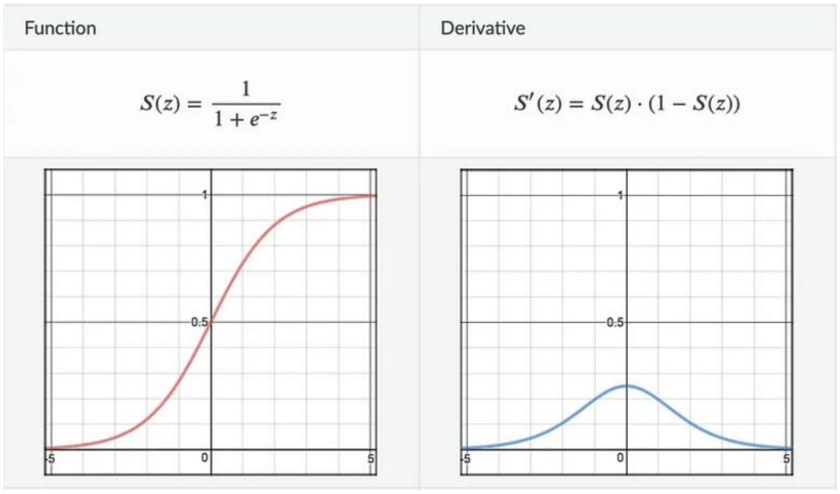
i. Initialization

Weights are initialized using the random.uniform() function in a range from -1 to 1. The biases for each layer are initialized as zeros.

ii. Activation function definitions

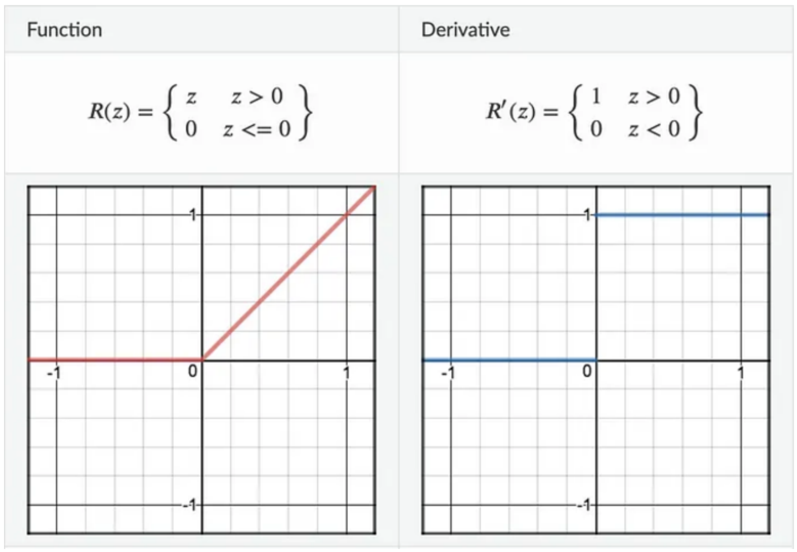
I define three different activation functions as Ill as their first derivatives:

- Sigmoid:



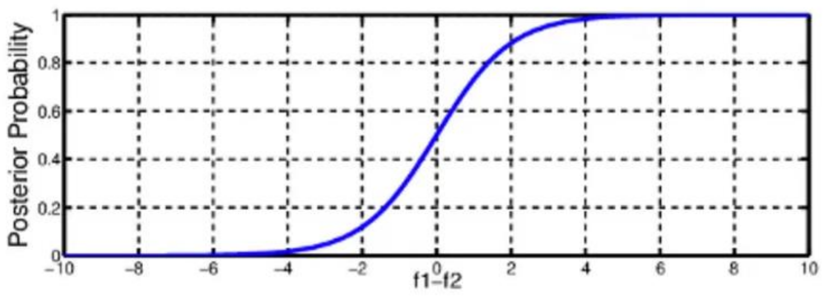
Function	Derivative
$S(z) = \frac{1}{1+e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$

- ReLU:



Function	Derivative
$R(z) = \begin{Bmatrix} z & z > 0 \\ 0 & z <= 0 \end{Bmatrix}$	$R'(z) = \begin{Bmatrix} 1 & z > 0 \\ 0 & z < 0 \end{Bmatrix}$

- Softmax:



$$softmax(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for\ j = 1, \dots, K$$

For Softmax specifically, I also added this line of code:

```
x_shifted = x - np.max(x, axis=1, keepdims=True)
```

This helps ensure the stability of our code and avoid any inf values and it was suggested by ChatGPT in order to avoid the overflow errors that I Ire facing.

iii. Loss function definition

For this assignment, the only loss function I defined was the cross-entropy loss function, described as:

```
-np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

following the mathematical expression:

$$cross\_entropy = \sum_{c=1}^{N} y_c \log(p_c)$$

The mean is added since we are handling more than one samples at a time.

iv. Training phase

For the forward pass, each input for each layer is multiplied (matrix multiplication) with the weights of said layer before the layer's biases are added to it. Then it is fed through the activation function and used as the input for the next layer.

Next, the loss is calculated using the function I have defined above.

During the backpropagation, the gradients for the weights and the biases are calculated based on the error of each layer. That error is the result of the gradient of the input of the activation functions multiplied by the product of the matrix multiplication between the error and weights of the next layer.

Finally, the weights and biases are updated using their respective gradients and the learning rate.

This happens as many times as the number of epochs decided upon the call of the function.

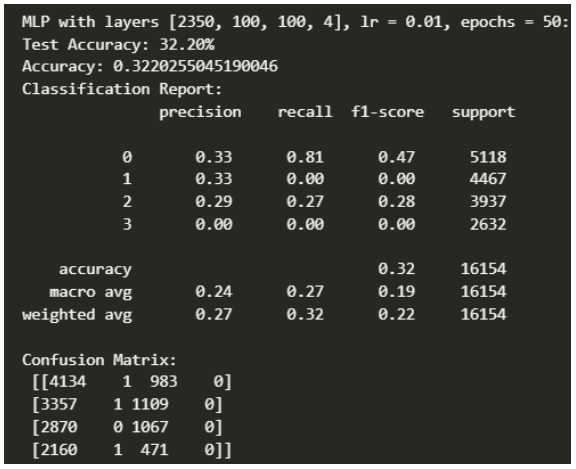
v. Prediction phase

During the prediction phase, basically only the forward pass of the training is used only once and the class with the highest probability is decided as the class of the sample.

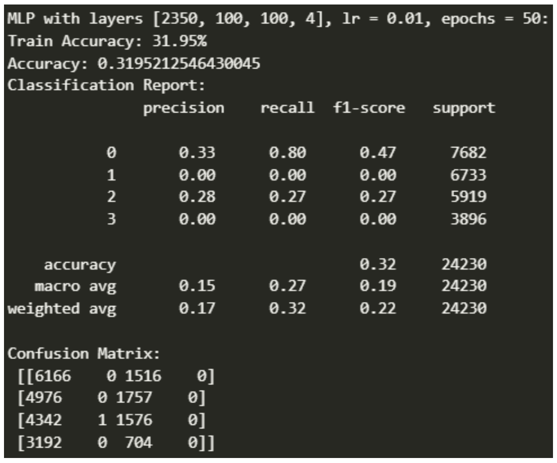
Chapter IV: Model and Parameter Variations

i. Variations in Architecture

In the first version of our algorithm, I used the entirety of the training data each time, with two layers of 1000/100 and 100/100 size and relatively high learning rate values. A for loop iterating over each sample was also naively implemented. That resulted in extremely high running times and extremely low metrics. One such example is shown below.



```
MLP with layers [2350, 100, 100, 4], lr = 0.01, epochs = 50:
Test Accuracy: 32.20%
Accuracy: 0.3220255045190046
Classification Report:
              precision    recall  f1-score   support

           0       0.33      0.81      0.47      5118
           1       0.33      0.00      0.00      4467
           2       0.29      0.27      0.28      3937
           3       0.00      0.00      0.00      2632

    accuracy                           0.32     16154
   macro avg       0.24      0.27      0.19     16154
weighted avg       0.27      0.32      0.22     16154

Confusion Matrix:
 [[4134    1  983    0]
 [3357    1 1109    0]
 [2870    0 1067    0]
 [2160    1  471    0]]
```



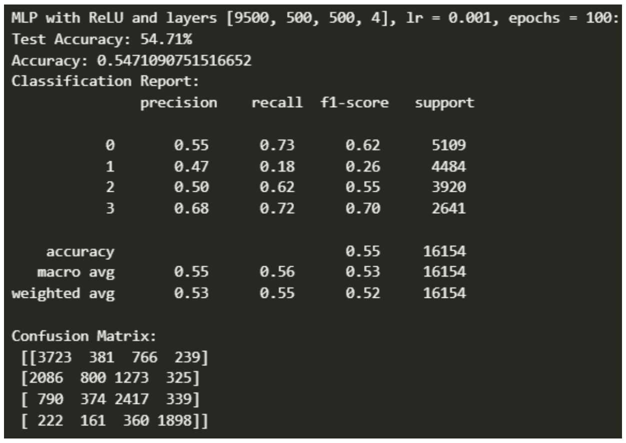
```
MLP with layers [2350, 100, 100, 4], lr = 0.01, epochs = 50:
Train Accuracy: 31.95%
Accuracy: 0.3195212546430045
Classification Report:
              precision    recall  f1-score   support

           0       0.33      0.80      0.47      7682
           1       0.00      0.00      0.00      6733
           2       0.28      0.27      0.27      5919
           3       0.00      0.00      0.00      3896

    accuracy                           0.32     24230
   macro avg       0.15      0.27      0.19     24230
weighted avg       0.17      0.32      0.22     24230

Confusion Matrix:
 [[6166    0 1516    0]
 [4976    0 1757    0]
 [4342    1 1576    0]
 [3192    0  704    0]]
```

The tendency to favor the first class can be explained by the imbalance in the dataset, which was discussed earlier. Using the whole dataset also affects performance; the running time since I have multiplications with bigger matrices and the metrics since the neural network must learn from a bigger count of mistakes at once (think of a student trying to learn from a hundred mistakes in the final exam as opposed to the student learning from ten mistakes in each smaller exam throughout the term).

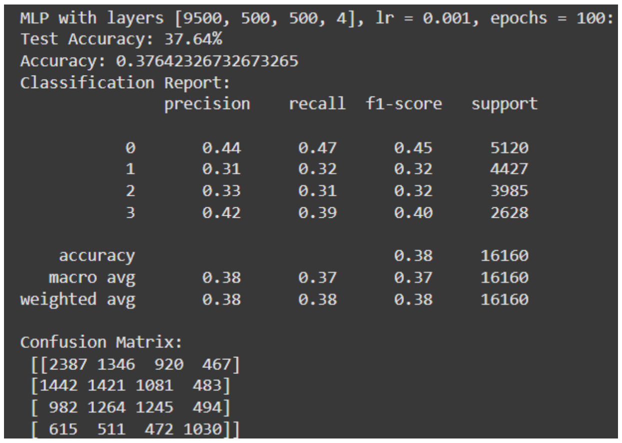
As such, I used matrix calculus and limited the 'for' use and I introduced batches in our architecture. As I'll soon find out, that improved our algorithm greatly.

ii. Variations in Activation Functions

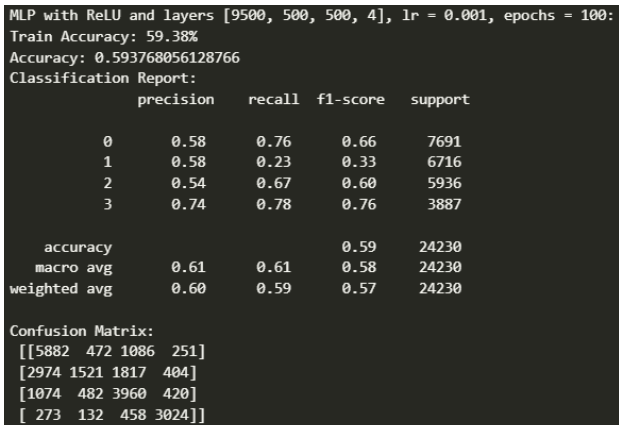
I tested the ReLU and Sigmoid activation functions against one another. For both, Softmax was the activation function in the final layer and a batch of 1000 was used each time.



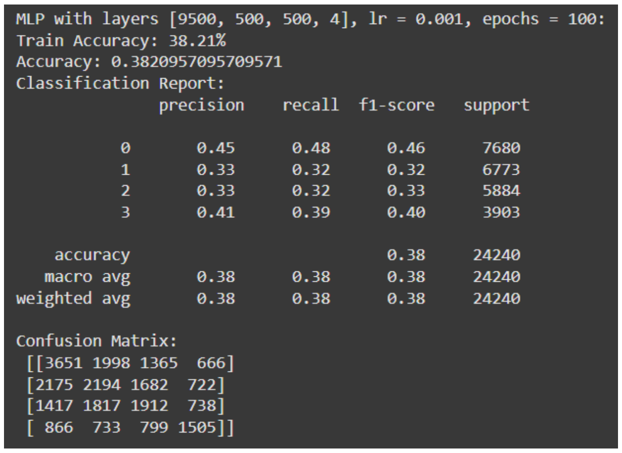
```
MLP with ReLU and layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Test Accuracy: 54.71%
Accuracy: 0.5471090751516652
Classification Report:
              precision    recall  f1-score   support

           0       0.55      0.73      0.62      5109
           1       0.47      0.18      0.26      4484
           2       0.50      0.62      0.55      3920
           3       0.68      0.72      0.70      2641

    accuracy                           0.55     16154
   macro avg       0.55      0.56      0.53     16154
weighted avg       0.53      0.55      0.52     16154

Confusion Matrix:
 [[3723  381  766  239]
 [2086  800 1273  325]
 [ 790  374 2417  339]
 [ 222  161  360 1898]]
```



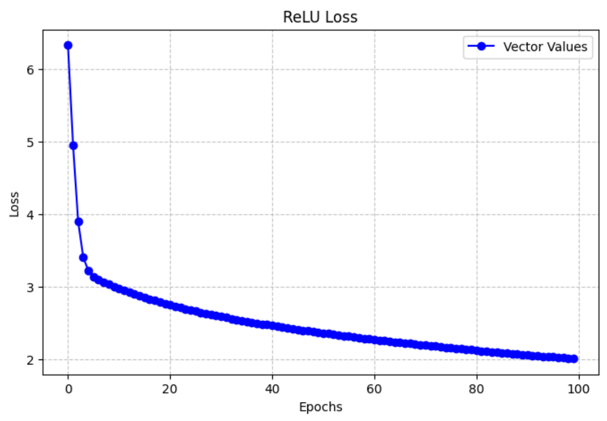
```
MLP with layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Test Accuracy: 37.64%
Accuracy: 0.37642326732673265
Classification Report:
              precision    recall  f1-score   support

           0       0.44      0.47      0.45      5120
           1       0.31      0.32      0.32      4427
           2       0.33      0.31      0.32      3985
           3       0.42      0.39      0.40      2628

    accuracy                           0.38     16160
   macro avg       0.38      0.37      0.37     16160
weighted avg       0.38      0.38      0.38     16160

Confusion Matrix:
 [[2387 1346  920  467]
 [1442 1421 1081  483]
 [ 982 1264 1245  494]
 [ 615  511  472 1030]]
```

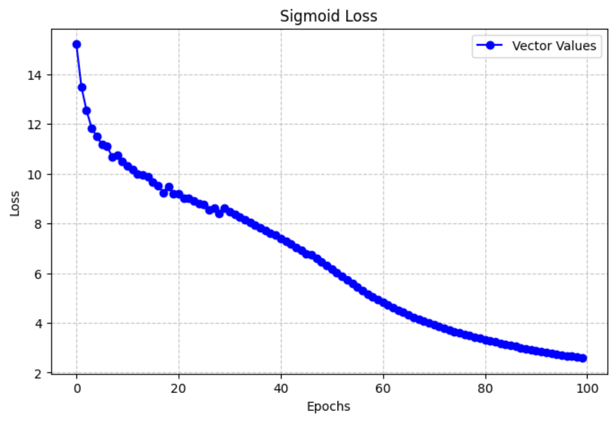


```
MLP with ReLU and layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Train Accuracy: 59.38%
Accuracy: 0.593768056128766
Classification Report:
              precision    recall  f1-score   support

           0       0.58      0.76      0.66      7691
           1       0.58      0.23      0.33      6716
           2       0.54      0.67      0.60      5936
           3       0.74      0.78      0.76      3887

    accuracy                           0.59     24230
   macro avg       0.61      0.61      0.58     24230
weighted avg       0.60      0.59      0.57     24230

Confusion Matrix:
 [[5882  472 1086  251]
 [2974 1521 1817  404]
 [1074  482 3960  420]
 [ 273  132  458 3024]]
```



```
MLP with layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Train Accuracy: 38.21%
Accuracy: 0.3820957095709571
Classification Report:
              precision    recall  f1-score   support

           0       0.45      0.48      0.46      7680
           1       0.33      0.32      0.32      6773
           2       0.33      0.32      0.33      5884
           3       0.41      0.39      0.40      3903

    accuracy                           0.38     24240
   macro avg       0.38      0.38      0.38     24240
weighted avg       0.38      0.38      0.38     24240

Confusion Matrix:
 [[3651 1998 1365  666]
 [2175 2194 1682  722]
 [1417 1817 1912  738]
 [ 866  733  799 1505]]
```

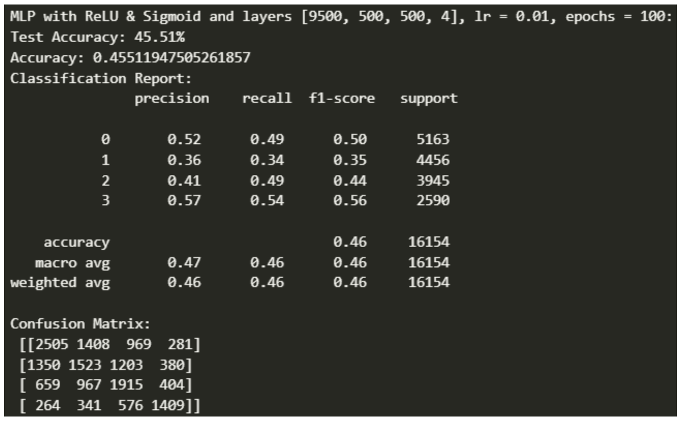




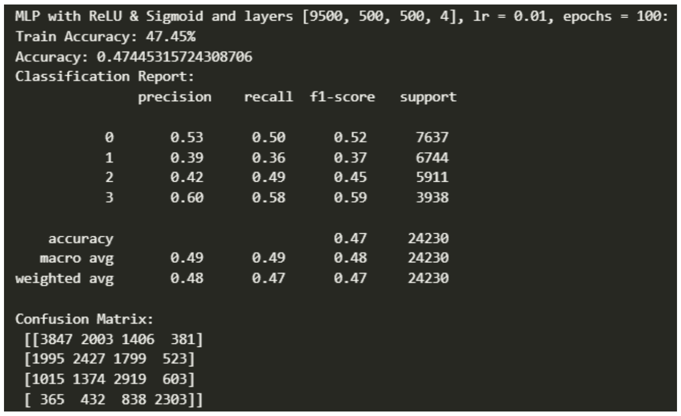
We find that ReLU performed significantly better in both train and test datasets as Ill as converged a lot faster and also took a lot less time to run, since it does not need to compute the exponential. ReLU was therefore found to be the better choice of the two, for both good time and better performance.

We also note that, in comparison to the previous architecture, the model performs in a more efficient way and while the first class is still slightly favored, it is nowhere near as extreme as before.

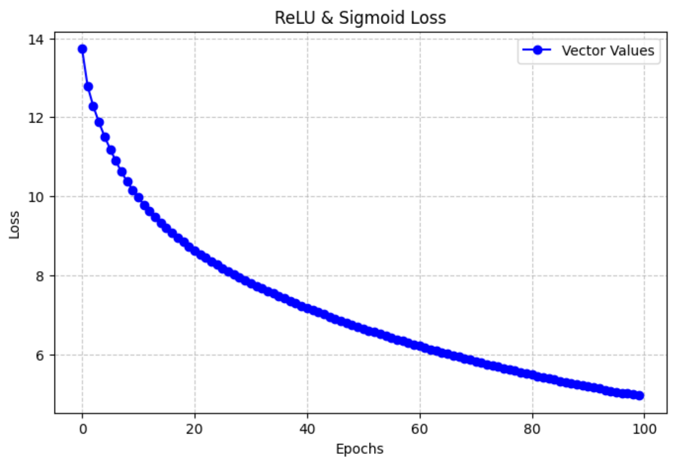
I also wanted to test what would happen if ReLU and Sigmoid were both used in different layers of the same model. Therefore, an MLP with a ReLU, Sigmoid and Softmax layer was implemented. (learning rate appears mistakenly 0.01, should be 0.001)



```
MLP with ReLU & Sigmoid and layers [9500, 500, 500, 4], lr = 0.01, epochs = 100:
Test Accuracy: 45.51%
Accuracy: 0.45511947505261857
Classification Report:
               precision    recall  f1-score   support

           0       0.52      0.49      0.50      5163
           1       0.36      0.34      0.35      4456
           2       0.41      0.49      0.44      3945
           3       0.57      0.54      0.56      2590

    accuracy                           0.46     16154
   macro avg       0.47      0.46      0.46     16154
weighted avg       0.46      0.46      0.46     16154

Confusion Matrix:
 [[2505 1408  969  281]
 [1350 1523 1203  380]
 [ 659  967 1915  404]
 [ 264  341  576 1409]]
```



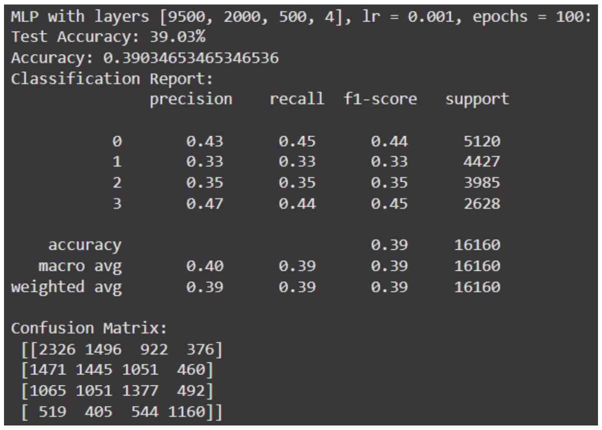
```
MLP with ReLU & Sigmoid and layers [9500, 500, 500, 4], lr = 0.01, epochs = 100:
Train Accuracy: 47.45%
Accuracy: 0.47445315724308706
Classification Report:
               precision    recall  f1-score   support

           0       0.53      0.50      0.52      7637
           1       0.39      0.36      0.37      6744
           2       0.42      0.49      0.45      5911
           3       0.60      0.58      0.59      3938

    accuracy                           0.47     24230
   macro avg       0.49      0.49      0.48     24230
weighted avg       0.48      0.47      0.47     24230

Confusion Matrix:
 [[3847 2003 1406  381]
 [1995 2427 1799  523]
 [1015 1374 2919  603]
 [ 365  432  838 2303]]
```



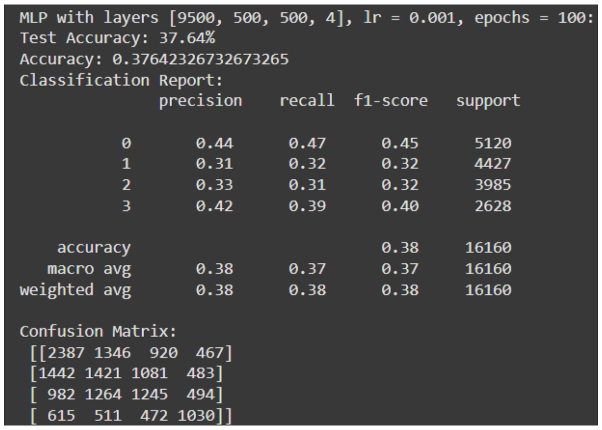
The performance was somewhere between the one each activation function achieved separately, with ReLU still holding the best results.

iii. Variations in Number of Neurons

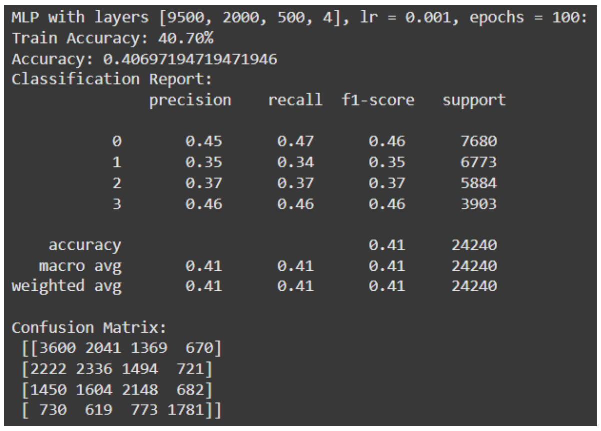
I tried a few different combinations of neurons, namely 1000/100, 2000/500, 500/500 and 100/100. Generally, the 2000/500 case performed the best. However, using big layer sizes also led to bigger running times without enough improvement to the metrics to justify that choice. Below, I focus on the comparison between the 500/500 and 2000/500 cases.



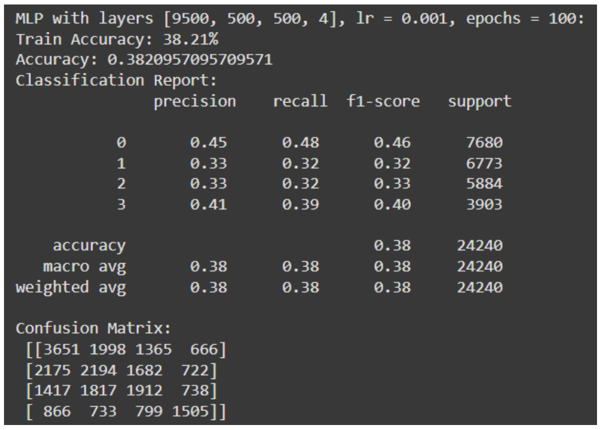
```
MLP with layers [9500, 2000, 500, 4], lr = 0.001, epochs = 100:
Test Accuracy: 39.03%
Accuracy: 0.39034653465346536
Classification Report:
              precision    recall  f1-score   support

           0       0.43      0.45      0.44      5120
           1       0.33      0.33      0.33      4427
           2       0.35      0.35      0.35      3985
           3       0.47      0.44      0.45      2628

    accuracy                           0.39     16160
   macro avg       0.40      0.39      0.39     16160
weighted avg       0.39      0.39      0.39     16160

Confusion Matrix:
 [[2326 1496  922  376]
 [1471 1445 1051  460]
 [1065 1051 1377  492]
 [ 519  405  544 1160]]
```



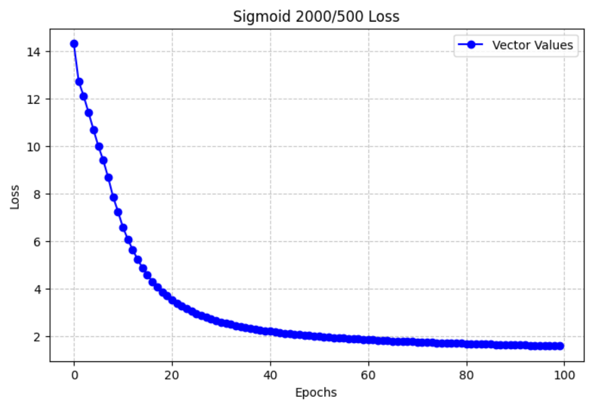
```
MLP with layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Test Accuracy: 37.64%
Accuracy: 0.37642326732673265
Classification Report:
              precision    recall  f1-score   support

           0       0.44      0.47      0.45      5120
           1       0.31      0.32      0.32      4427
           2       0.33      0.31      0.32      3985
           3       0.42      0.39      0.40      2628

    accuracy                           0.38     16160
   macro avg       0.38      0.37      0.37     16160
weighted avg       0.38      0.38      0.38     16160

Confusion Matrix:
 [[2387 1346  920  467]
 [1442 1421 1081  483]
 [ 982 1264 1245  494]
 [ 615  511  472 1030]]
```

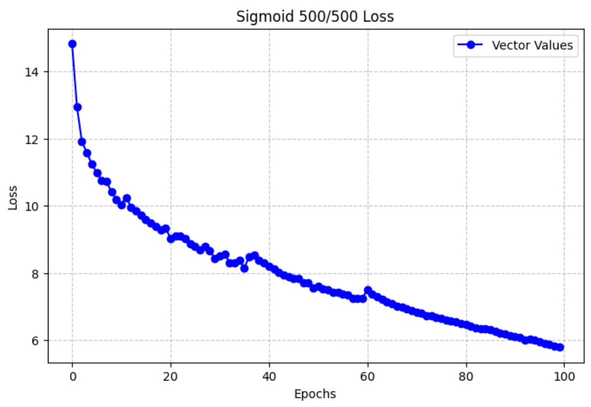


```
MLP with layers [9500, 2000, 500, 4], lr = 0.001, epochs = 100:
Train Accuracy: 40.70%
Accuracy: 0.40697194719471946
Classification Report:
              precision    recall  f1-score   support

           0       0.45      0.47      0.46      7680
           1       0.35      0.34      0.35      6773
           2       0.37      0.37      0.37      5884
           3       0.46      0.46      0.46      3903

    accuracy                           0.41     24240
   macro avg       0.41      0.41      0.41     24240
weighted avg       0.41      0.41      0.41     24240

Confusion Matrix:
 [[3600 2041 1369  670]
 [2222 2336 1494  721]
 [1450 1604 2148  682]
 [ 730  619  773 1781]]
```



```
MLP with layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Train Accuracy: 38.21%
Accuracy: 0.3820957095709571
Classification Report:
              precision    recall  f1-score   support

           0       0.45      0.48      0.46      7680
           1       0.33      0.32      0.32      6773
           2       0.33      0.32      0.33      5884
           3       0.41      0.39      0.40      3903

    accuracy                           0.38     24240
   macro avg       0.38      0.38      0.38     24240
weighted avg       0.38      0.38      0.38     24240

Confusion Matrix:
 [[3651 1998 1365  666]
 [2175 2194 1682  722]
 [1417 1817 1912  738]
 [ 866  733  799 1505]]
```

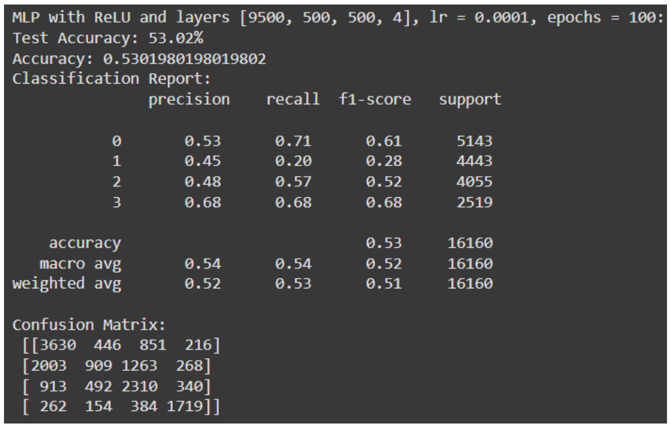




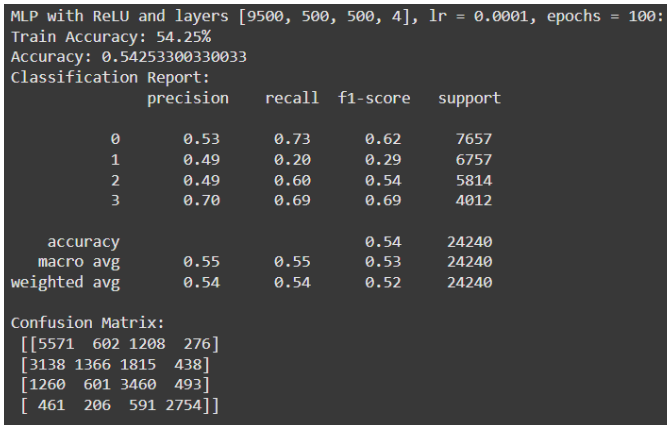
We see that the first case performed slightly better than the second and also had faster convergence to a lower loss

iv. Variations in Learning Rate

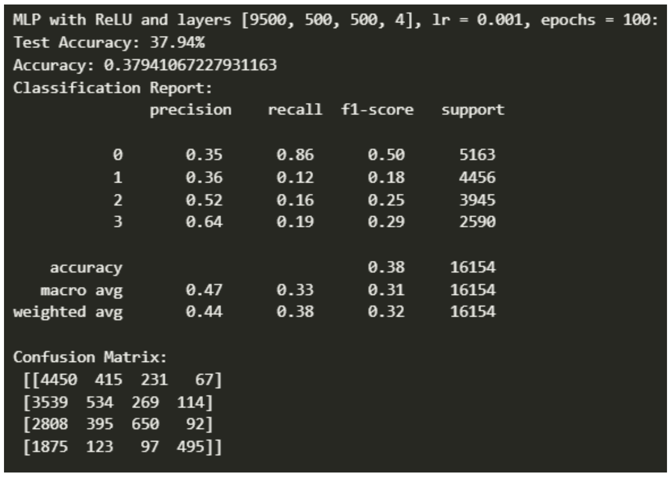
As for the learning rate, I tested on the values of 0.01, 0.001, 0.0001. By far the worst performing was the biggest one, as expected, therefore the comparison will focus on the other two.



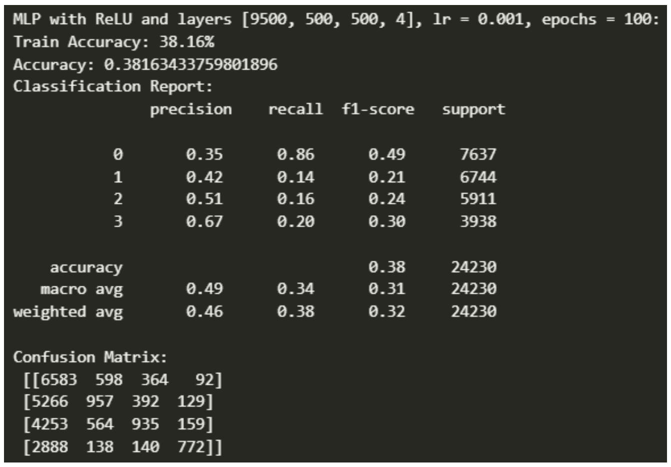
```
MLP with ReLU and layers [9500, 500, 500, 4], lr = 0.0001, epochs = 100:
Test Accuracy: 53.02%
Accuracy: 0.5301980198019802
Classification Report:
              precision    recall  f1-score   support

           0       0.53      0.71      0.61      5143
           1       0.45      0.20      0.28      4443
           2       0.48      0.57      0.52      4055
           3       0.68      0.68      0.68      2519

    accuracy                           0.53     16160
   macro avg       0.54      0.54      0.52     16160
weighted avg       0.52      0.53      0.51     16160

Confusion Matrix:
 [[3630  446  851  216]
 [2003  909 1263  268]
 [ 913  492 2310  340]
 [ 262  154  384 1719]]
```



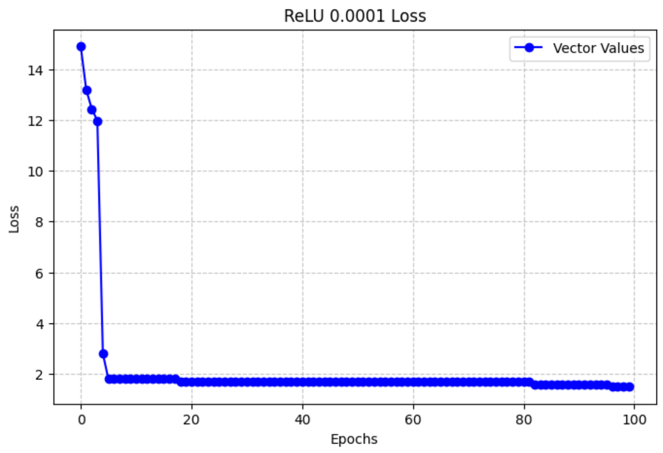
```
MLP with ReLU and layers [9500, 500, 500, 4], lr = 0.0001, epochs = 100:
Train Accuracy: 54.25%
Accuracy: 0.54253300330033
Classification Report:
              precision    recall  f1-score   support

           0       0.53      0.73      0.62      7657
           1       0.49      0.20      0.29      6757
           2       0.49      0.60      0.54      5814
           3       0.70      0.69      0.69      4012

    accuracy                           0.54     24240
   macro avg       0.55      0.55      0.53     24240
weighted avg       0.54      0.54      0.52     24240

Confusion Matrix:
 [[5571  602 1208  276]
 [3138 1366 1815  438]
 [1260  601 3460  493]
 [ 461  206  591 2754]]
```

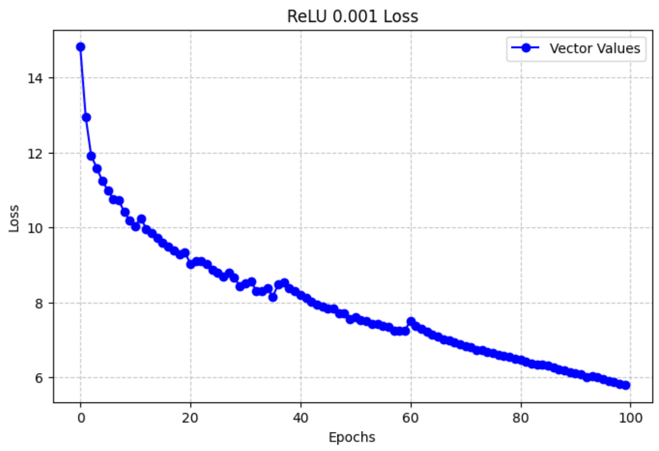


```
MLP with ReLU and layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Test Accuracy: 37.94%
Accuracy: 0.37941067227931163
Classification Report:
              precision    recall  f1-score   support

           0       0.35      0.86      0.50      5163
           1       0.36      0.12      0.18      4456
           2       0.52      0.16      0.25      3945
           3       0.64      0.19      0.29      2590

    accuracy                           0.38     16154
   macro avg       0.47      0.33      0.31     16154
weighted avg       0.44      0.38      0.32     16154

Confusion Matrix:
 [[4450  415  231   67]
 [3539  534  269  114]
 [2808  395  650   92]
 [1875  123   97  495]]
```



```
MLP with ReLU and layers [9500, 500, 500, 4], lr = 0.001, epochs = 100:
Train Accuracy: 38.16%
Accuracy: 0.38163433759801896
Classification Report:
              precision    recall  f1-score   support

           0       0.35      0.86      0.49      7637
           1       0.42      0.14      0.21      6744
           2       0.51      0.16      0.24      5911
           3       0.67      0.20      0.30      3938

    accuracy                           0.38     24230
   macro avg       0.49      0.34      0.31     24230
weighted avg       0.46      0.38      0.32     24230

Confusion Matrix:
 [[6583  598  364   92]
 [5266  957  392  129]
 [4253  564  935  159]
 [2888  138  140  772]]
```

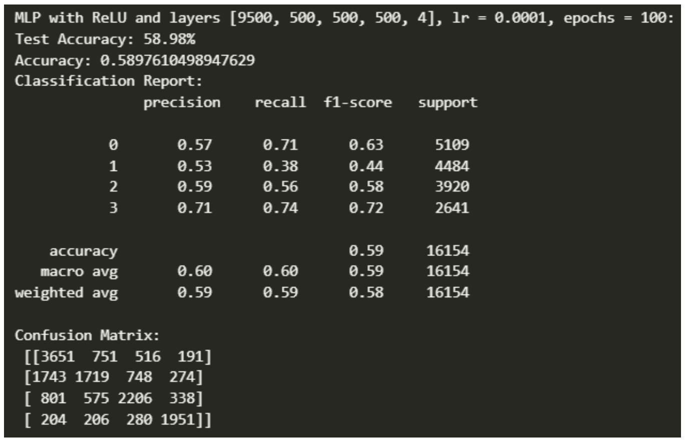




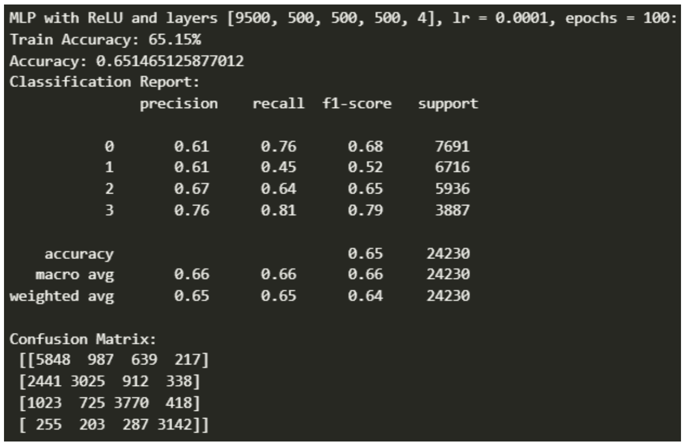
We can assume from the results that the smaller value of 0.0001 performs better than the alternative, giving us higher metrics. The running time was very comparable between the two. However, it is worth noting that with the choice of lr = 0.0001 we have faster convergence to a lower loss and problems surrounding the stability of the algorithm and the unexpected nan values that would sometimes appear were both solved.

v. Variations in Number of Layers

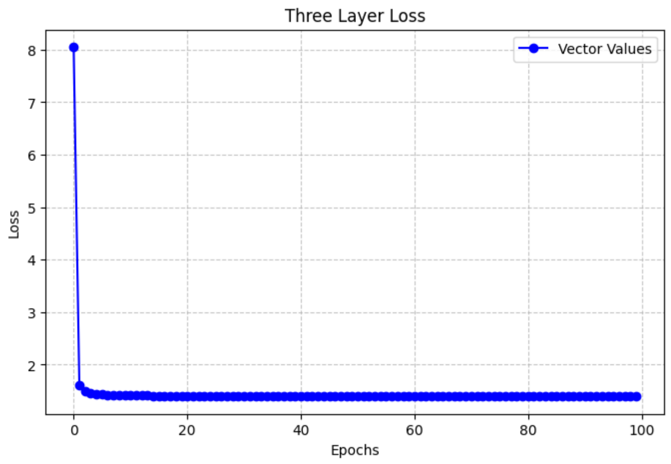
Finally, one more layer was added to test the effect the number of layers have on the efficiency and accuracy of the algorithm. Naturally, this rendition took the longest to run but also provided the best results. I tried an MLP with layers 2000/1000/100 and one with 500/500/500. The first one performed poorly, achieving an accuracy of approximately 34% and is therefore not presented.



```
MLP with ReLU and layers [9500, 500, 500, 500, 4], lr = 0.0001, epochs = 100:
Test Accuracy: 58.98%
Accuracy: 0.5897610498947629
Classification Report:
              precision    recall  f1-score   support

           0       0.57      0.71      0.63      5109
           1       0.53      0.38      0.44      4484
           2       0.59      0.56      0.58      3920
           3       0.71      0.74      0.72      2641

    accuracy                           0.59     16154
   macro avg       0.60      0.60      0.59     16154
weighted avg       0.59      0.59      0.58     16154

Confusion Matrix:
 [[3651  751  516  191]
 [1743 1719  748  274]
 [ 801  575 2206  338]
 [ 204  206  280 1951]]
```



```
MLP with ReLU and layers [9500, 500, 500, 500, 4], lr = 0.0001, epochs = 100:
Train Accuracy: 65.15%
Accuracy: 0.651465125877012
Classification Report:
              precision    recall  f1-score   support

           0       0.61      0.76      0.68      7691
           1       0.61      0.45      0.52      6716
           2       0.67      0.64      0.65      5936
           3       0.76      0.81      0.79      3887

    accuracy                           0.65     24230
   macro avg       0.66      0.66      0.66     24230
weighted avg       0.65      0.65      0.64     24230

Confusion Matrix:
 [[5848  987  639  217]
 [2441 3025  912  338]
 [1023  725 3770  418]
 [ 255  203  287 3142]]
```



Chapter V: Conclusions

i. Comparison and Discussion

In the end, the observations from the various tests performed can be summarized by the following:

- Using the dataset in batches is important to make the algorithm's learning more effective, as well as avoid prohibitive running times.
- The ReLU activation function has many advantages over the Sigmoid, namely time, simplicity, convergency speed and overall performance.
- Extreme values, whether high or low, should be avoided as layer sizes and layers with the same size often perform better or at least almost as best as layers with descending sizes.
- A lower learning rate is important not only to achieve better results but also to limit errors and improve the stability of the algorithm.
- Depth -of course withing reason- generally improves performance and accuracy, leading to better, more reliable results.

The MLP algorithm created managed to reach as high as 60% accuracy on the testing set and 65% accuracy on the training set, with the correct combination of parameters and architecture, managing therefore to beat the Nearest Centroid algorithm. Of course, that is not a great achievement, seeing as the KNN algorithm, which is generally considered the ground base when it comes to neural networks due to its 'brute force' approach, performed significantly better on the same dataset.

If anything, that goes to show that, when building a neural network, creating the skeleton is the easiest part. In order to transform it into a trustworthy tool that can be used easily and efficiently to solve real-life problems, feature engineering, optimization and hyperparameter tuning is arguably the most crucial step and takes the longest in the creating process.

ii. Limitations

The attempt to create an MLP from scratch was a good learning experience, but still many flaws and limitations in my implementation should be acknowledged. Firstly, the source code is far from optimized and some naïve and ineffective coding choices were made which I will improve in a second version, outside of this assignment. Even so, the code is fully functional and mathematically correct.

Ideally, more steps should have been taken in order to have safer, more dependable conclusions, such as more experimentation with the number of layers and neurons as well as different loss functions, addition of a validation set as well as cross-validation to detect overfitting and also it would have been useful to test the dataset on an already optimized MLP algorithm with PyTorch to compare the achieved results and have a better understanding of how much the quality of the code affects the results.

Finally, despite my best efforts, it was not possible to do all the comparisons with the same data split, which very often affects the performance of the algorithm. That is why it would have been better to have cross-validation as well.

References

https://www.geeksforgeeks.org/backpropagation-in-neural-network/

https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning

https://medium.com/@cmukesh8688/activation-functions-sigmoid-tanh-relu-leaky-relu-softmax-50d3778dcea5

https://medium.com/deep-learning-study-notes/multi-layer-perceptron-mlp-in-pytorch-21ea46d50e62