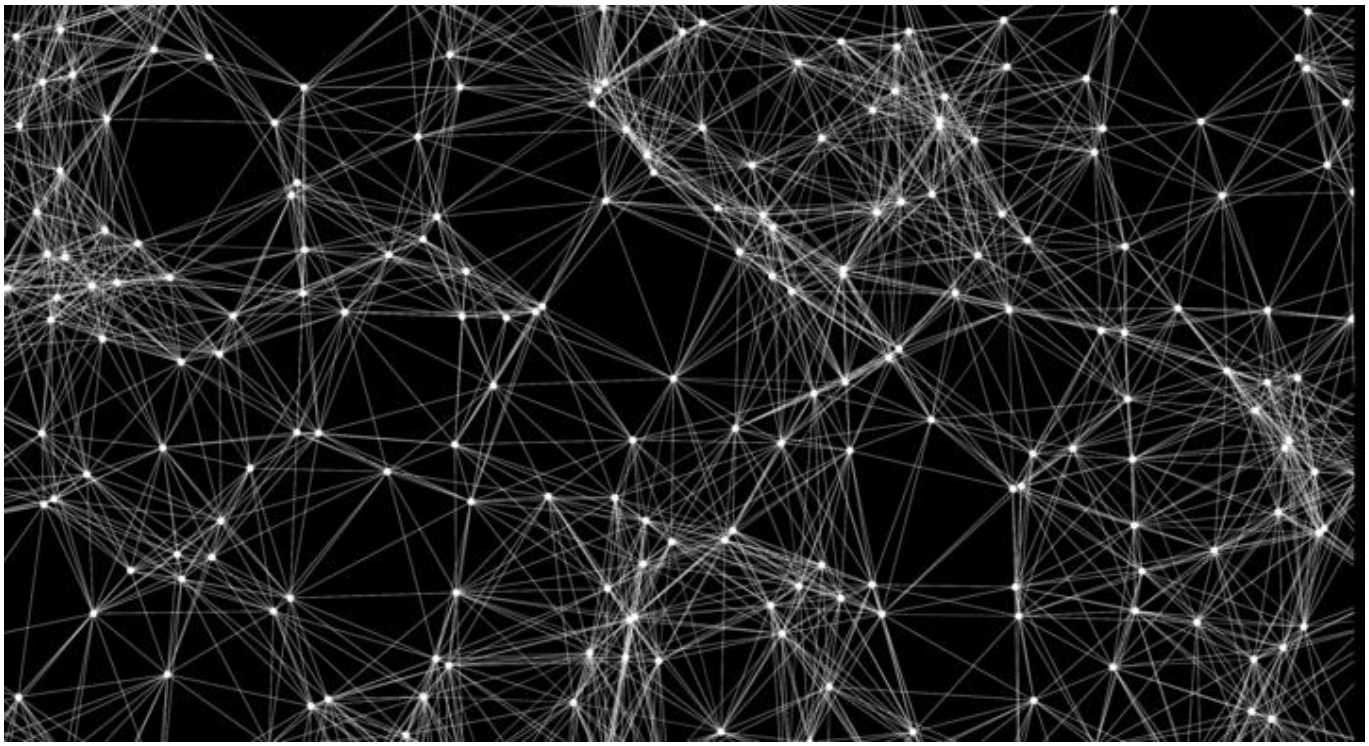
Neural Networks – Project 1



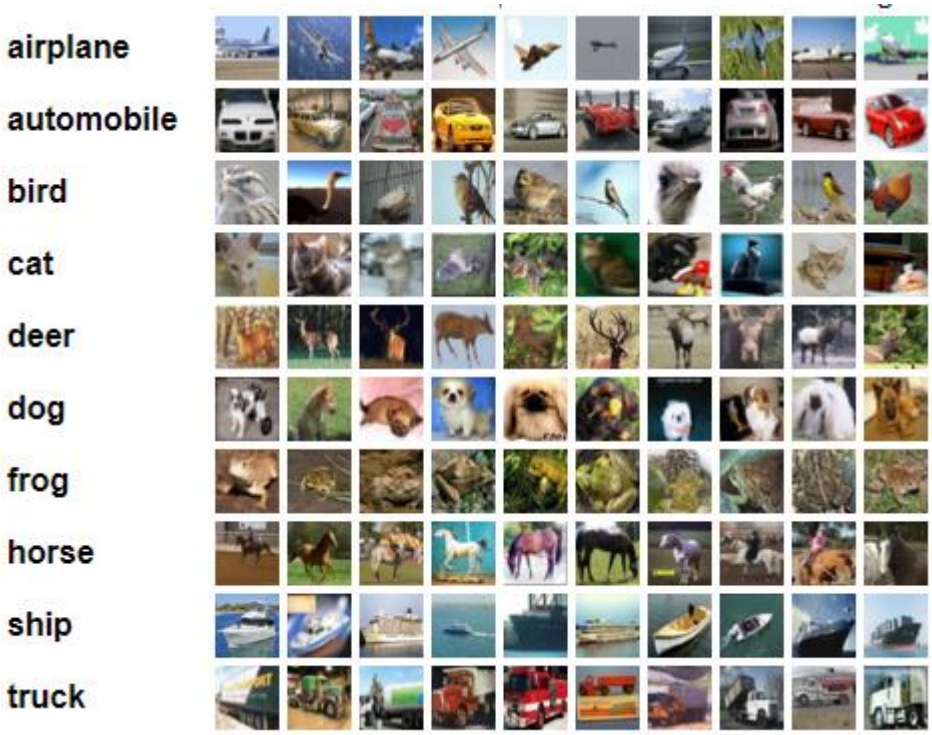
Nikolaos Konstas

konstasn@ece.auth.gr

In this work it was requested to implement the 1/3-Nearest, Neighbor and Nearest Centroid algorithms and a Neural Network to solve a classification problem and to compare the performances of the above classifiers, after first selecting the problem/dataset. For the above issues the Cifar-10 dataset has been selected and the implementation of the classifiers has been done in the python programming language and the Google Colab programming environment.

Cifar-10:

Cifar-10 is one of the most widespread datasets in the field of deep-learning and neural networks. It consists of 60,000 color (RGB) images of dimension 32x32 which belong to 10 different classes: Airplane, Car, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck.



Python & Google Colaboratory:

Python was chosen for the implementation of the work due to its simplicity and ease of use, combined with the incredible capabilities of the implemented libraries it offers. Google Colab was also chosen, which makes it easier for us as it has all the packages and libraries we need pre-installed and enables us to take advantage of Google's hardware and its GPUs together to speed up the training of neural networks.

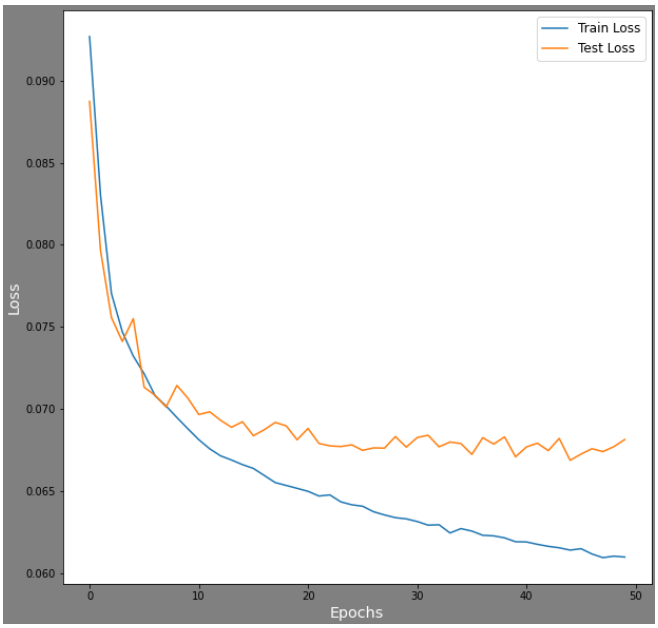
Multilayer Perceptrons

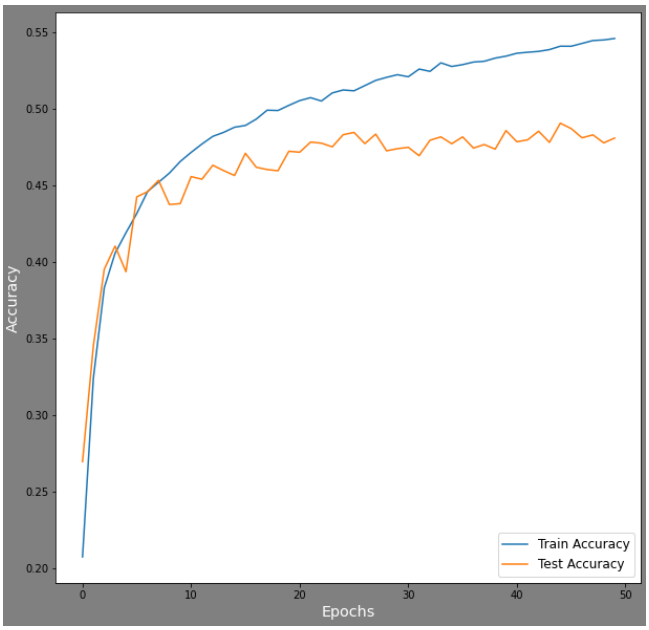
This is where our experiments with Multilayer Perceptrons begin. In the 1st part of our experiments with MLPs we try to familiarize ourselves with their hyperparameters and through them draw some conclusions about them, which will help us in the following work. For this reason we start with a very simple network which consists of a Hidden Layer with 100 neurons and the Output Layer with 10 neurons.

In the following tests, only ReLu was used as the activation function in the hidden neuron layer and Adam as the optimizer, while different functions were tested in the output layer. Different loss functions were also tested but more emphasis was placed on learning rate & batch size, so as to find a balance between training time and efficiency.

Before presenting and commenting on the analytical results we see a single model from those tested which uses Adam optimizer with Mean Squared Error loss function, Sigmoid as activation function and is trained for 50 epochs with batch size 100 and learning rate 0.001.

The specific model categorizes the samples of the train set with a success rate of 53.6% and the test set with 48.65% while it takes 1 minute and 20 seconds to train.





In the learning curves of the MLP we notice that the phenomenon of overfitting appears, but at this stage it does not concern us, as we simply want to observe the behavior of the Neural Networks for the various values of the hyperparameters and in a later stage we will deal with the maximization of accuracy.

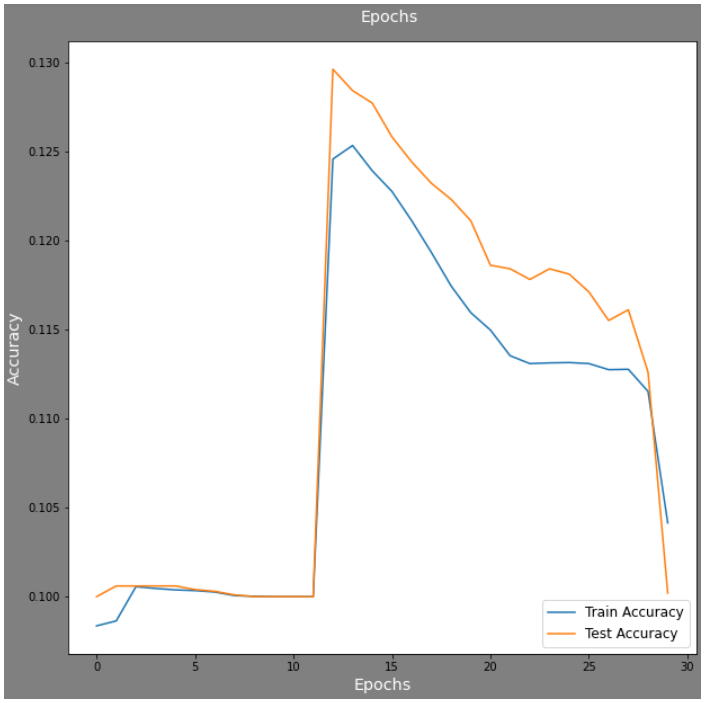
It should be noted that in all the tests the data is normalized in [0,1].

Here is a table with the detailed results of the tests performed:

Hidden Layer Activation	Output Layer Activation	Loss Function	Batch Size	Learning Rate	Epochs	Train Accuracy(%)	Test Accuracy(%)	Training Time
ReLU	Sigmoid	Mean Squared Error	10	10^{-5}	50	49,19	47,18	10min 22s
ReLU	Sigmoid	Mean Squared Error	10	$5*10^{-5}$	50	58,86	49,95	10min 22s
ReLU	Sigmoid	Mean Squared Error	10	10^{-4}	50	58,33	49,28	10min 22s
ReLU	Sigmoid	Mean Squared Error	10	$2*10^{-4}$	50	55,74	47,48	10min 22s
ReLU	Sigmoid	Mean Squared Error	10	$5*10^{-4}$	50	51,63	46,52	10min 22s
ReLU	Sigmoid	Mean Squared Error	10	10^{-3}	50	-	-	10min 22s
ReLU	Sigmoid	Mean Squared Error	25	10^{-5}	50	53,52	48,72	4 min 23s
ReLU	Sigmoid	Mean Squared Error	25	$5*10^{-5}$	50	56,27	49,63	4 min 23s
ReLU	Sigmoid	Mean Squared Error	25	10^{-4}	50	58,68	50,03	4 min 23s
ReLU	Sigmoid	Mean Squared Error	25	$5*10^{-4}$	50	54,79	48,71	4 min 23s
ReLU	Sigmoid	Mean Squared Error	25	10^{-3}	50	-	-	4 min 23s
ReLU	Sigmoid	Mean Squared Error	25	$1,5*10^{-3}$	50	-	-	4 min 23s
ReLU	Sigmoid	Mean Squared Error	50	10^{-4}	50	56,4	48,71	2min 10s
ReLU	Sigmoid	Mean Squared Error	50	$2,5*10^{-4}$	50	56,42	50,03	2min 10s
ReLU	Sigmoid	Mean Squared Error	50	$5*10^{-4}$	50	56,5	49,63	2min 10s
ReLU	Sigmoid	Mean Squared Error	50	10^{-3}	50	52,07	48,72	2min 10s
ReLU	Sigmoid	Mean Squared Error	50	$1,5*10^{-3}$	50	-	-	2min 10s
ReLU	Sigmoid	Mean Squared Error	50	$2*10^{-3}$	50	-	-	2min 10s
ReLU	Sigmoid	Mean Squared Error	100	$5*10^{-4}$	50	53,05	46,85	1min 20s
ReLU	Sigmoid	Mean Squared Error	100	10^{-3}	50	53,6	48,65	1min 20s
ReLU	Sigmoid	Mean Squared Error	100	$1,5*10^{-3}$	50	51,07	46,67	1min 20s
ReLU	Sigmoid	Mean Squared Error	100	$2*10^{-3}$	50	-	-	1min 20s
ReLU	Sigmoid	Mean Squared Error	200	$2,5*10^{-4}$	50	53,58	48,59	46s
ReLU	Sigmoid	Mean Squared Error	200	$5*10^{-4}$	50	55,25	49,51	46s
ReLU	Sigmoid	Mean Squared Error	200	$7,5*10^{-4}$	50	53,49	47,4	46s
ReLU	Sigmoid	Mean Squared Error	200	10^{-3}	50	54,56	47,92	46s
ReLU	Sigmoid	Mean Squared Error	200	$1,5*10^{-3}$	50	52,33	46,85	46s
ReLU	Sigmoid	Mean Squared Error	200	$2*10^{-3}$	50	-	-	46s
ReLU	Sigmoid	Mean Squared Error	500	10^{-4}	50	50,11	46,86	28s
ReLU	Sigmoid	Mean Squared Error	500	$2,5*10^{-4}$	50	52	48,03	28s
ReLU	Sigmoid	Mean Squared Error	500	$5*10^{-4}$	50	51,84	47,5	28s
ReLU	Sigmoid	Mean Squared Error	500	$5*10^{-4}$	100	57,63	49,69	1min 22s
ReLU	Sigmoid	Mean Squared Error	500	10^{-3}	50	53,52	48,38	28s
ReLU	Sigmoid	Mean Squared Error	500	$1,5*10^{-3}$	50	53,49	48,69	28s
ReLU	Sigmoid	Mean Squared Error	500	$1,75*10^{-3}$	50	51,43	46,21	28s
ReLU	Sigmoid	Mean Squared Error	500	$2*10^{-3}$	50	-	-	28s
ReLU	Sigmoid	Mean Squared Error	1000	$2,5*10^{-4}$	50	48,26	45,89	25s
ReLU	Sigmoid	Mean Squared Error	1000	$5*10^{-4}$	50	49,61	47,17	25s
ReLU	Sigmoid	Mean Squared Error	1000	$7,5*10^{-4}$	50	48,91	45,44	25s
ReLU	Sigmoid	Mean Squared Error	1000	10^{-3}	50	49,45	46,89	25s
ReLU	Sigmoid	Mean Squared Error	1000	$1,5*10^{-3}$	50	47,99	44,85	25s
ReLU	Sigmoid	Mean Squared Error	5000	10^{-4}	50	38,85	38,72	20s
ReLU	Sigmoid	Mean Squared Error	5000	$5*10^{-4}$	50	41,05	40,04	20s
ReLU	Sigmoid	Mean Squared Error	5000	10^{-3}	50	33,48	31,94	20s
ReLU	Sigmoid	Mean Squared Error	5000	$1,5*10^{-3}$	50	22,95	21,99	20s
ReLU	Sigmoid	Mean Squared Error	5000	$1,9*10^{-3}$	50	21,35	20,13	20s
ReLU	Sigmoid	Mean Squared Error	5000	$2*10^{-3}$	50	-	-	20s
ReLU	Sigmoid	Mean Squared Error	10000	10^{-4}	50	34,4	34,06	19s
ReLU	Sigmoid	Mean Squared Error	10000	10^{-4}	100	39,03	38,67	42s
ReLU	Sigmoid	Mean Squared Error	10000	$5*10^{-4}$	50	33,29	33,72	19s
ReLU	Sigmoid	Mean Squared Error	10000	$5*10^{-4}$	100	42,49	41,34	42s
ReLU	Sigmoid	Mean Squared Error	10000	10^{-3}	50	39,68	38,45	19s
ReLU	Sigmoid	Mean Squared Error	10000	10^{-3}	100	35,58	34,21	42s
ReLU	Sigmoid	Mean Squared Error	10000	$1,5*10^{-3}$	50	19,17	19,21	19s
ReLU	Sigmoid	Mean Squared Error	10000	$1,9*10^{-3}$	50	20,3	20,03	19s
ReLU	Sigmoid	Mean Squared Error	10000	$2*10^{-3}$	50	-	-	19s
ReLU	Softmax	Categorical Crossentropy	100	$5*10^{-5}$	30	51,13	48,18	51s
ReLU	Softmax	Categorical Crossentropy	100	10^{-4}	30	53,28	49,42	51s
ReLU	Softmax	Categorical Crossentropy	100	10^{-4}	50	57,02	50,6	1min 20s
ReLU	Softmax	Categorical Crossentropy	100	$3*10^{-4}$	30	52,28	47,6	51s
ReLU	Softmax	Categorical Crossentropy	100	$5*10^{-4}$	30	53,08	48,83	51s
ReLU	Softmax	Categorical Crossentropy	100	10^{-3}	30	47,35	44,22	51s
ReLU	Softmax	Categorical Crossentropy	100	$2*10^{-3}$	30	45,49	43,78	51s
ReLU	Softmax	Categorical Crossentropy	100	$3*10^{-3}$	30	33,04	29,64	51s
ReLU	Softmax	Categorical Crossentropy	100	$5*10^{-3}$	30	38,8	37,46	51s
500 Neurons on Hidden layer								
ReLU	Softmax	Categorical Crossentropy	100	10^{-4}	30	60,64	53,2	56s

Looking at the data the first thing we notice is the large fluctuations in training time. Specifically, the smaller the batch size, the longer the training time. This is to be expected since a smaller batch size implies more batches and therefore more error calculations. Even larger batch sizes benefit from the parallelism offered by tensorflow using GPUs, speeding up the training process even more.

Another conclusion we can draw is that in general, and with the appropriate learning rate, smaller batch sizes lead to better results. It should be noted at this point that for different batch sizes we obtain optimal results for different learning rates, for smaller batch sizes we need smaller learning rates, while there is also an upper limit on the value of the learning rate for which the MLP can be trained, the which in most cases is about 2x10-3. If we exceed this limit, the NN cannot be trained and the accuracy fluctuates around 10% (in the table above these cases are denoted by – in accuracy).



Although the scale is a bit misleading, looking at the accuracy values on the Y axis we notice that in all seasons it is close to 10%.

According to the above and taking a closer look we come to the conclusion that for a good speed-performance balance the best choice is batch size = 100 and a learning rate of the order of 10-4.

Finally, the combinations Mean Squared Error – Sigmoid & Categorical Crossentropy – Softmax seem to offer the same levels of accuracy but consulting the literature we prefer the 2nd combination for the rest of the tests.

Based on the above detailed tests, and as mentioned above, we choose:

- Activation Functions: ReLu -> Hidden Layers, Softmax -> Output Layer
- Loss Function: Categorical Crossentropy
- Optimizer: Adam

- Batch Size: 100
- Learning Rate: 10^{-4} – $1{,}5x10^{-4}$
- Epochs: 30

And we start experimenting with MLP architecture by adding Layers & Neurons.

We consider as a baseline our original model with 1 Hidden Layer and 100 neurons for which we have:

Train Accuracy	Test Accuracy	Train Time
53,28%	49,42%	51s

The first change we try is to increase the neurons from 100 to 500. The results after this change:

Train Accuracy	Test Accuracy	Train Time
60,64%	53,20%	56s

Our new MLP takes negligibly more training time, while offering a noticeable 4% increase in test accuracy.

In the second stage we increase the Hidden Layers to 2, but in the 1st Layer we return to 100 neurons, while in the 2nd Layer we give 50 neurons.

Train Accuracy	Test Accuracy	Train Time
54,34%	50,06%	50s

In this case we observe a small decrease in the training time, but also a noticeable drop in performance, which is reasonable, since the total trained parameters of the new NN are less than the above one.

Therefore we understand that the number of neurons in the 1st Layer is important and now we train a network with 500 neurons in the 1st and 50 in the 2nd.

Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	61,37%	52,63%	56s
$1.5*10^{-4}$	62,44%	53,22%	56s

Both the training time and accuracy of MLP are practically the same as the 1st Hidden Layer network, so we decide to increase the neurons in the 2nd Hidden Layer from 50 to 100 as well.

Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	63,39%	53,87%	1min 23s
$1.5*10^{-4}$	64,91%	54,04%	1min 23s

With these changes we get slightly better results, so we keep adding Layers, hoping to get further performance improvements.

The next MLP we test consists of 3 Hidden Layers which have 500-250-100 neurons respectively.

Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	66,77%	54,2%	1min 25s
$1.5*10^{-4}$	69,02%	54,36%	1min 25s

Again we see very small improvements, so this time we increase the 1st Hidden Layer neurons to 1000.

Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	70,46%	54,57%	1min 26s
$1.5*10^{-4}$	71,3%	54,26%	1min 26s

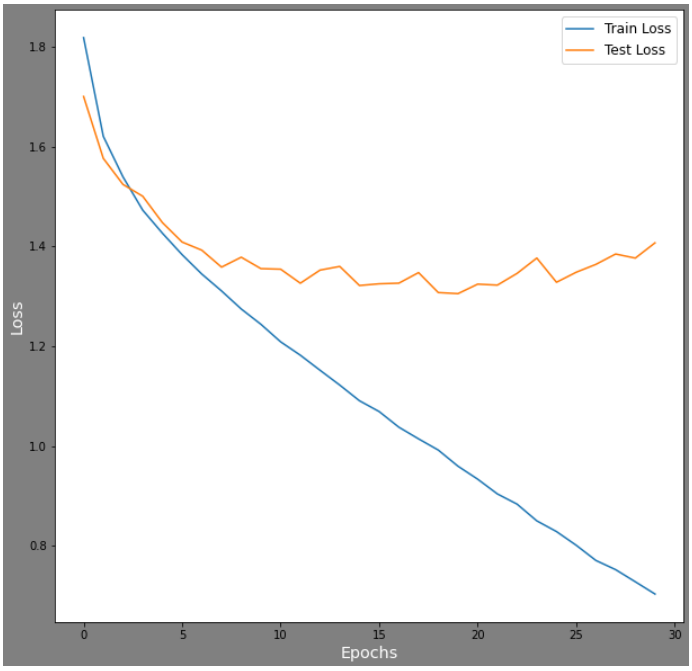
Even now we don't have significant improvements, so we try to increase the neurons in both 2nd & 3rd Hidden Layer and now we have an MLP with 1000/500/250 neurons in each Layer.

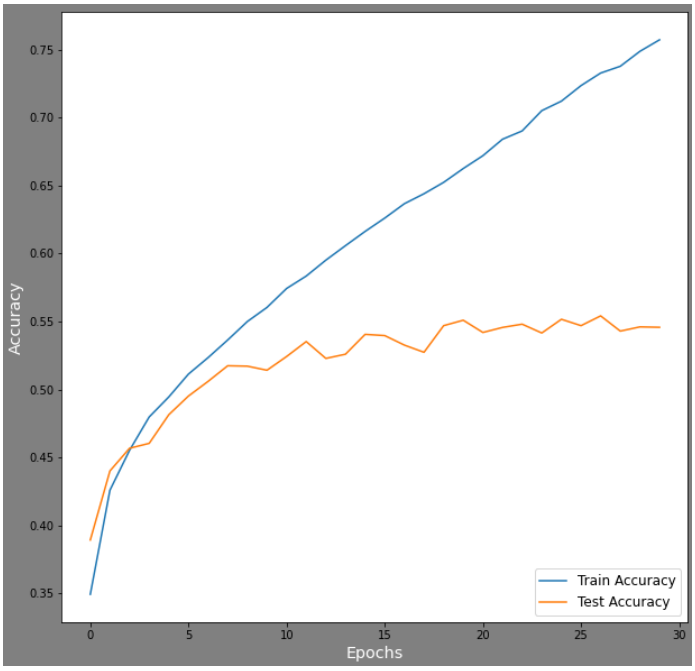
Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	73,87%	55,69%	1min 30s
$1.5*10^{-4}$	74,22%	55,22%	1min 30s

The new model shows a noticeable, but not particularly significant, increase in test accuracy. We notice that with the increase in Layers and neurons, the training accuracy has increased a lot and the phenomenon of poor generalization and overtraining is strong as can be seen in the learning curves below.

We add one more Hidden Layer and we have a total of 4 Hidden Layers with 1000-500-250-50 neurons:

Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	73,91%	55,8%	1min 38s
$1.5*10^{-4}$	76,4%	55,62%	1min 38s





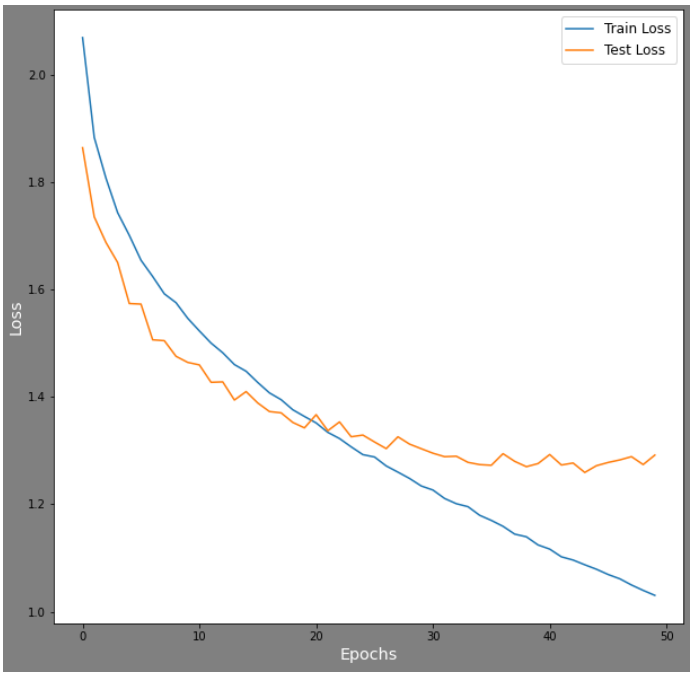
We do not see any noticeable improvement, so we understand that we are close to the limits of MLP in this particular problem, and there is no particular benefit to further increase the Hidden Layers and neurons.

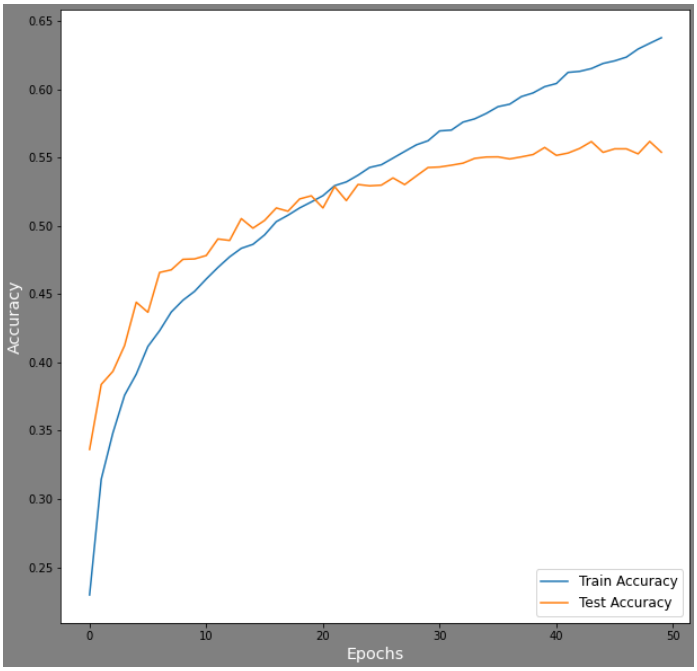
At this point we add Dropout layers to our MLP in order to eliminate over-fitting and possibly improve its performance. As the new network is trained harder we increase the epochs from 30 to 50.

First we try putting Dropout on the last model on all Layers with a 20% disable chance.

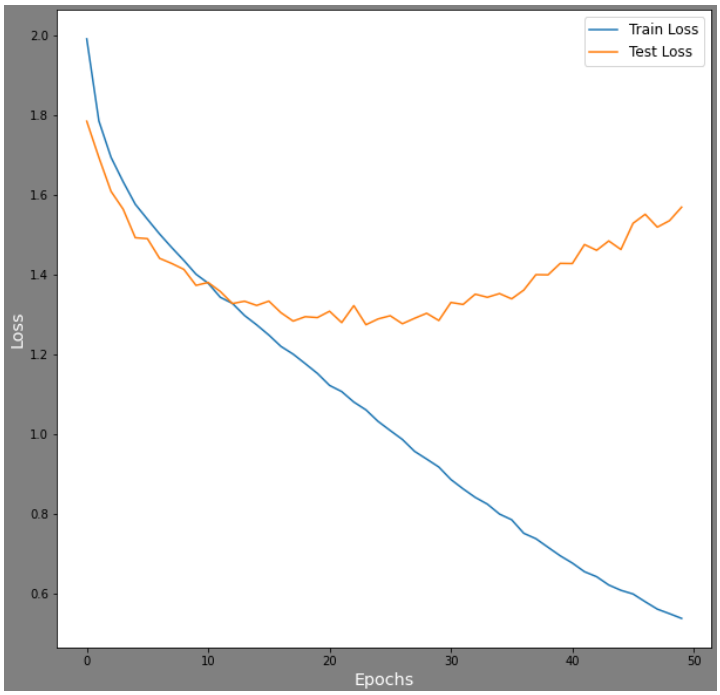
Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	63,32%	55,81%	2min 28s
$1.5*10^{-4}$	61,88%	55,93%	2min 23s

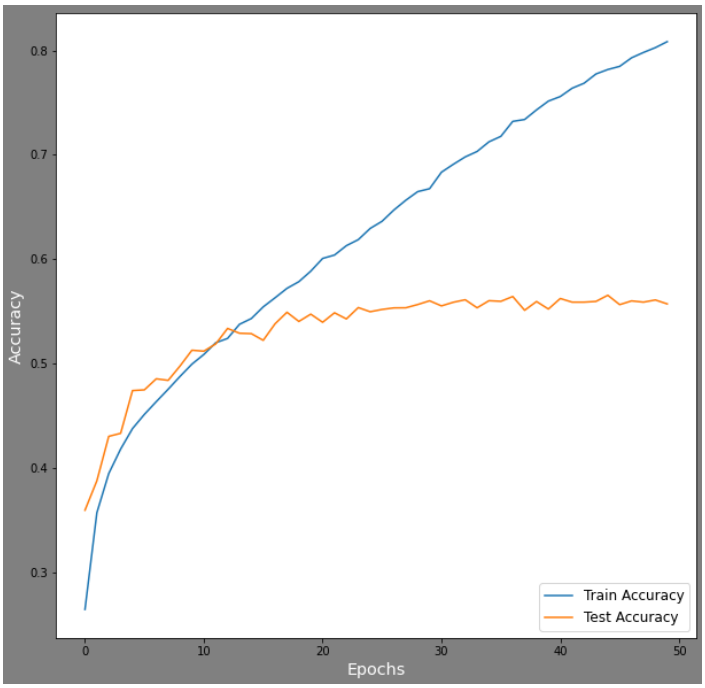
Although the accuracy of our MLP has not increased, with this modification we have managed to limit the phenomenon of over-fitting to a large extent. So now, we can add one more Hidden Layer and significantly increase the number of neurons. We train for 50 epochs an MLP with 5000-2500-1250-500-250 neurons.





Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	81,72%	56,63%	5min 23s
$1.5*10^{-4}$	80,86%	55,71%	5min





We see a very small improvement, but with the increased number of neurons the problem of over-fitting has become severe again. For this reason we increase the probability of Dropout to 30% and retrain.

Learning Rate	Train Accuracy	Test Accuracy	Train Time
10^{-4}	77,02%	56,77%	4min 26s

This particular modification did not bear fruit, as it reduced the overtraining effect very little and failed to increase the success rate of our model.

It is now clear that we have exhausted the capabilities of MLP and we need to use some other type of Neural Network to get better results.

The table below summarizes the results of the MLP tests.

Multi Layer Perceptron - Multiple Hidden Layers					
Optimizer: Adam, Loss Function: Categorical Crossentropy, Hidden Layers Activation: ReLU, Output Layer Activation, Batch Size: 100, Epochs: 30					
# of Hidden Layers	# of Neurons/Hidden Layer	Learning Rate	Train Accuracy(%)	Test Accuracy(%)	Training Time
1	500	10^{-4}	60,64	53,2	56s
2	100-50	10^{-4}	54,34	50,06	50s
2	500-50	10^{-4}	61,37	52,63	56s
2	500-50	$1.5*10^{-4}$	62,44	53,22	56s
2	500-100	10^{-4}	63,39	53,87	1min 23s
2	500-100	$1.5*10^{-4}$	64,91	54,04	1min 23s
3	500-250-100	10^{-4}	66,77	54,2	1min 25s
3	500-250-100	$1.5*10^{-4}$	69,02	54,36	1min 25s
3	1000-250-100	10^{-4}	70,46	54,57	1min 26s
3	1000-250-100	$1.5*10^{-4}$	71,3	54,26	1min 26s
3	1000-500-250	10^{-4}	73,87	55,69	1min 30s
3	1000-500-250	$1.5*10^{-4}$	74,22	55,22	1min 30s
4	1000-500-250-50	10^{-4}	73,91	55,8	1min 38s
4	1000-500-250-50	$1.5*10^{-4}$	76,4	55,62	1min 38s
4 / Dropout	1000-500-250-50	10^{-4}	63,32	55,81	2min 28s
4 / Dropout	1000-500-250-50	$1.5*10^{-4}$	61,88	55,93	2min 23s
5 / Dropout	5000-2500-1250-500-250	10^{-4}	81,72	56,63	4min 27s
5 / Dropout	5000-2500-1250-500-250	$1.5*10^{-4}$	80,86	55,71	4min 25s
5 / Dropout (30%)	5000-2500-1250-500-250	10^{-4}	77,02	56,77	4min 26s

Convolutional Neural Networks

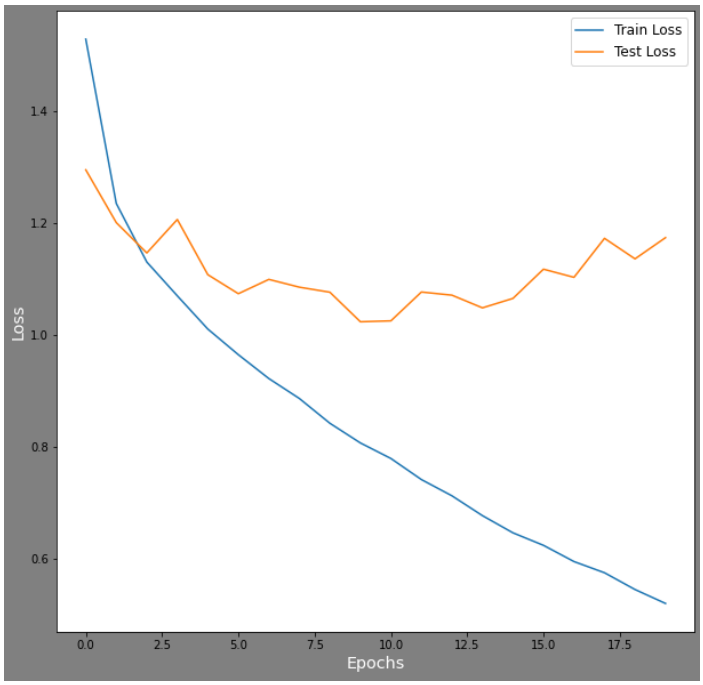
In the above part of the work, experimenting with the hyperparameters of the Multilayer Perceptrons, we managed to improve the accuracy of the NN, but without reaching a satisfactory level. To see further improvements and reach this satisfactory level we will use Convolutional Neural Networks which are usually the best choice when our problem has images as input.

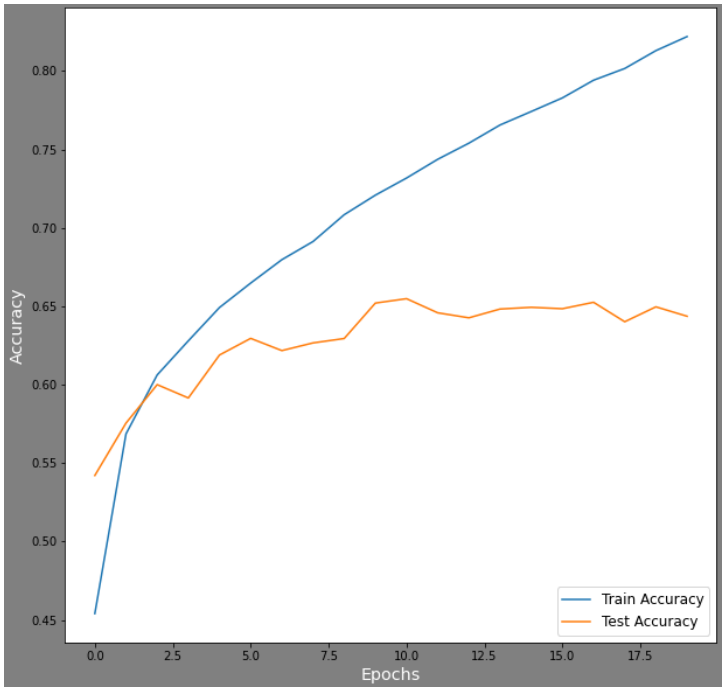
It should be noted that at this stage of the work we are not particularly concerned with the change of hyperparameters, but mainly we are testing architectures and other methods that can increase the efficiency of the NN. In all the following tests, optimizer is used: Adam, loss function: Categorical Crossentropy, learning rate = 0.001, batch size = 100.

We start with a simple network consisting of a Convolutional layer, a MaxPooling layer and finally an MLP with 1 Hidden Layer of 100 neurons which corresponds to the following code:

```
model = tf.keras.Sequential([
                            tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'),
                            tf.keras.layers.MaxPool2D(pool_size=(2,2)),
                            tf.keras.layers.Flatten(),
                            tf.keras.layers.Dense(100, activation= hidden_activation),
                            tf.keras.layers.Dense(10, activation= output_activation)
])
```

We train this model for 20 seasons. Below we see the training curves:



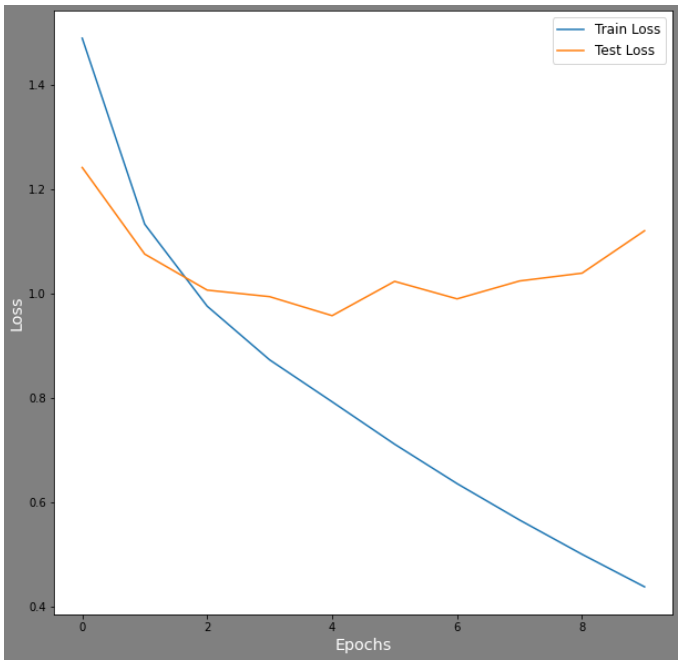


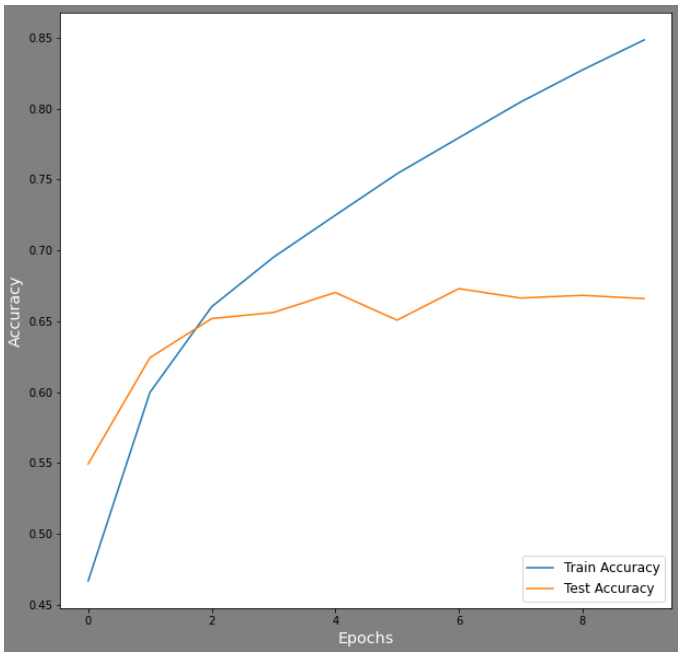
The CNN took only 36s to train, achieved train accuracy = 84.83% and test accuracy = 64.96%. Both in the results and in the curves it is clear that after the 10th epoch the phenomenon of over-fitting appears, so in the following tests we keep the training epochs at 10.

Remarkable is the fact that with such a simple CNN we have already managed to surpass by about 10% the most complex MLP we implemented.

Then we add an identical Convolutional layer before the MaxPooling Layer.

The new model, trained in 10 epochs, took 27 seconds and had train accuracy = 84.86% and test accuracy = 66.60%. We observe a small improvement in performance, but the problem of over-fitting continues, albeit to a lesser extent, but more importantly we see that the model does not generalize well.





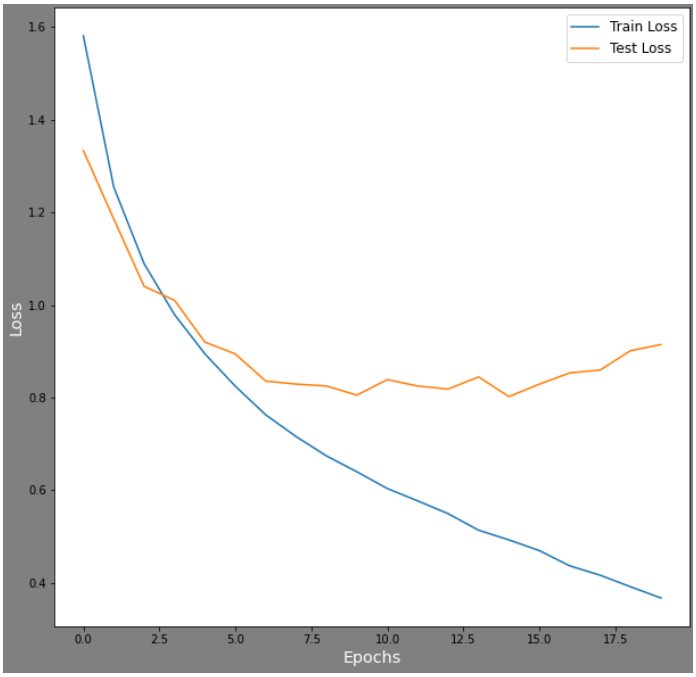
Now we test the same network by simply replacing the MaxPooling layer with an Average Pooling layer.

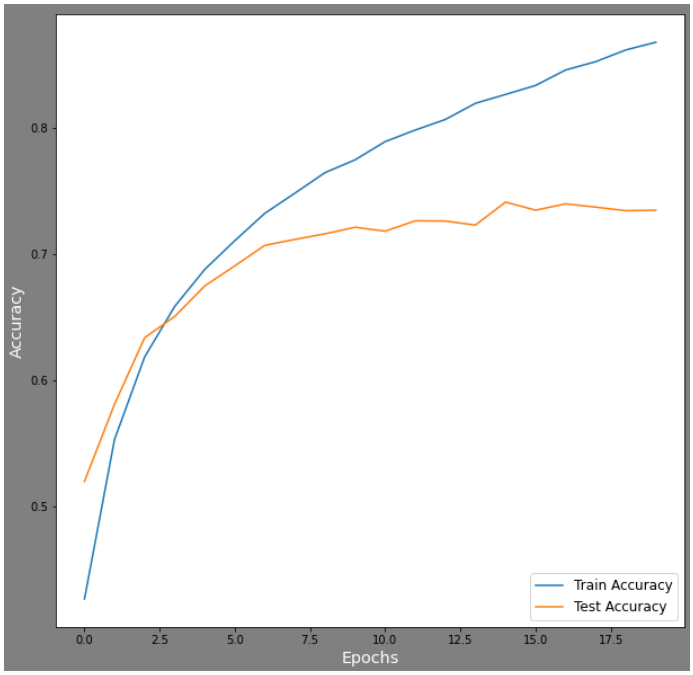
The learning curves of the specific NN are omitted as its results are similar to the above. Specifically we have: training time = 42 seconds, train accuracy = 82.97%, test accuracy = 66.09%.

Continuing, we take the "complete" Convolutional Layer we've created (2 Conv layers & 1 MaxPooling layer) and place an identical one after it. To train it we increase the epochs again to 20, as possibly this more complex model will be "harder" to train.

In this case we needed 59s while the train accuracy and test accuracy were respectively 86.75% and 73.46%. That is, with the addition of the new layers, a significant increase in test accuracy of 6% was observed.

Below we see the learning curves:





We notice that the problems of over-fitting and poor generalization are still present, but perhaps not so severe.

Now we are trying to increase the number of filters in the 2nd "complete" Convolutional Layer we use from 32 to 64.

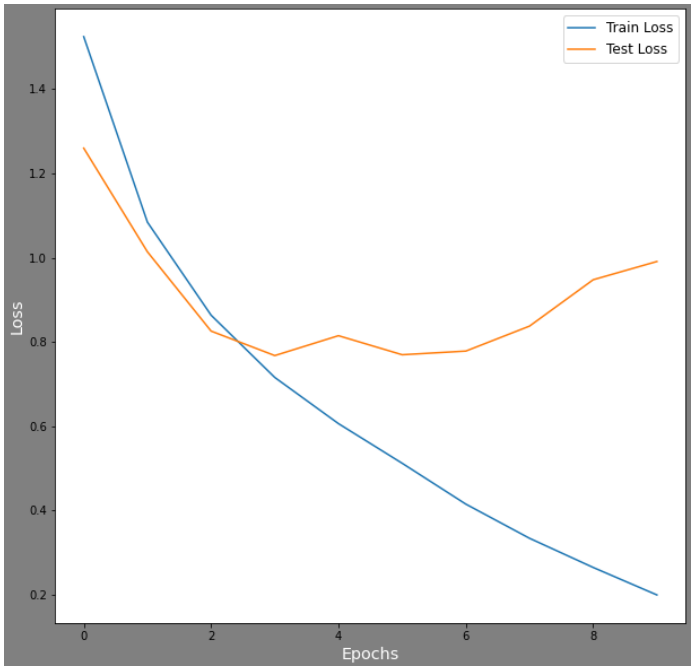
After 20 epochs, which in total lasted 1 minute and 23 seconds, we get train accuracy = 94.44% and test accuracy = 73.45%, essentially the same result as above.

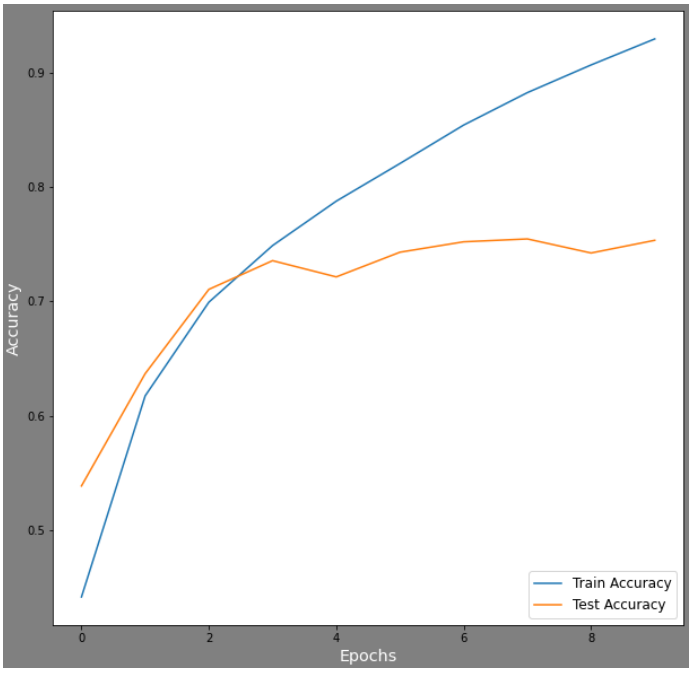
Observing that 10 seasons still seem enough, we return to this number.

At this point, due to the increased complexity of the Convolutional Network, and with the increase of filters, the increased dimension of the vector managed by the Dense NN we increase the complexity of the latter as well and add a Hidden Layer with 500 neurons. Our network has now taken this form:

```
model = tf.keras.Sequential([
                            tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'),
                            tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'),
                            tf.keras.layers.MaxPool2D(pool_size=(2,2)),
                            tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation='relu'),
                            tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation='relu'),
                            tf.keras.layers.MaxPool2D(pool_size=(2,2)),
                            tf.keras.layers.Flatten(),
                            tf.keras.layers.Dense(500, activation= hidden_activation),
                            tf.keras.layers.Dense(100, activation= hidden_activation),
                            tf.keras.layers.Dense(10, activation= output_activation)
])
```

Indeed, with the changes in the network we managed to improve the test accuracy a little more, reaching 75.34% (runtime = 36s, train accuracy = 92.95), but also by reducing the seasons we also limited the phenomenon of over-fitting a little , as shown below:





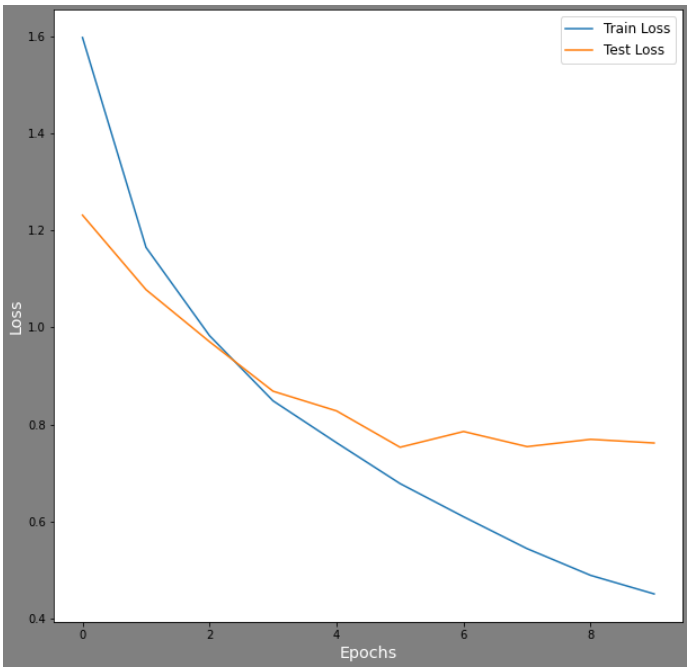
EAdditionally, we try the same architecture by simply replacing the MaxPooling Layers with AveragePooling Layers, as before. CNN again takes 35 seconds in the training stage and achieves train accuracy = 90.16% and test accuracy = 72.81%, slightly worse than the results using MaxPooling.

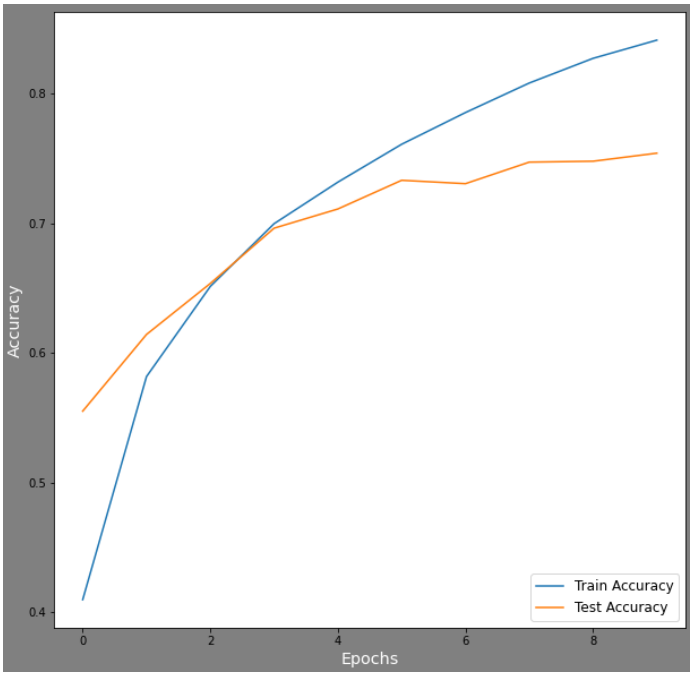
Then we try to improve the performance by adding additional convolutional layers. Due to the low dimension to which our data has been driven from the previous layers (5x5) we cannot add a "full" Convolutional Layer like the above, so we add a Convolutional Layer with 128 filters and a kernel (2,2) which is followed by from a MaxPooling Layer.

This particular CNN is trained in 37 seconds, with train accuracy = 84.10% and test accuracy = 75.39%, i.e. in the test set it has the same performance as the above network. The most important improvement is the reduction of over-fitting and better generalization to unknown data.

Continuing, in order to increase the dimension of the data reaching the MLP we remove the last MaxPooling Layer.

This test is not very successful returning:





Train Accuracy	Test Accuracy	Train Time
89,62%	73,69%	37s

As mentioned above in the 3rd convolutional block we can apply only one convolutional layer as the dimensions of the data do not allow us to use a 2nd one. To keep the dimension a little higher and to be able to apply more filters we use padding = 'same'.

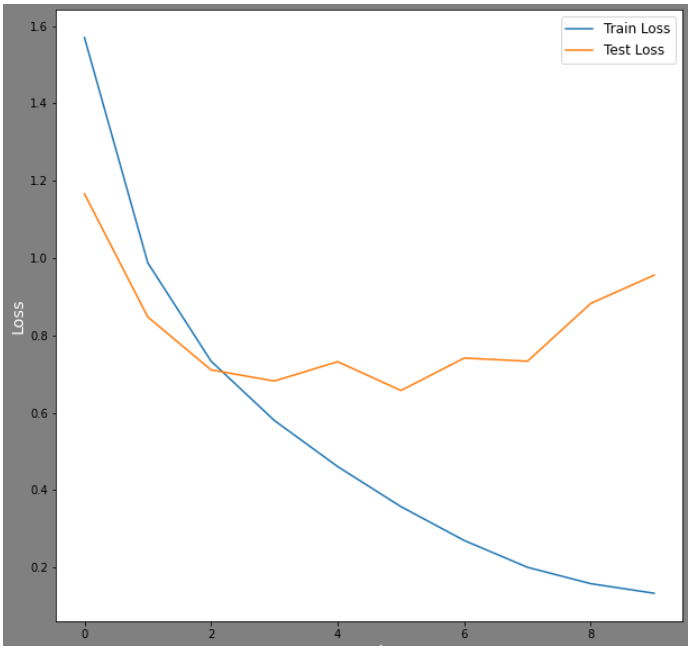
Train Accuracy	Test Accuracy	Train Time
94,24%	76,83%	52s

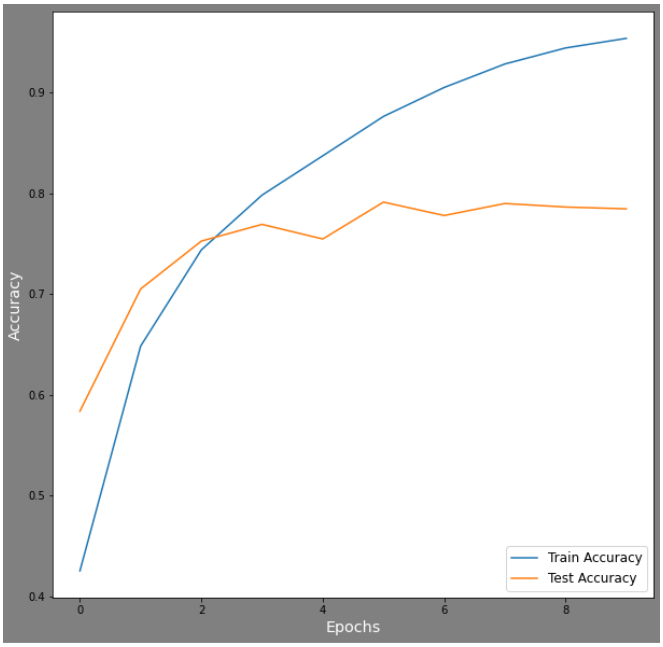
As we expected, the new model slightly increased the success rate with a small cost in training time, which is however very small and does not concern us.

The next thought we have to increase the performance of the CNN is to increase the number of filters applied to the Convolutional Layers. Thus we increase the filters to: 1st block 64-64, 2nd block 128-128, 3rd block 256-256.

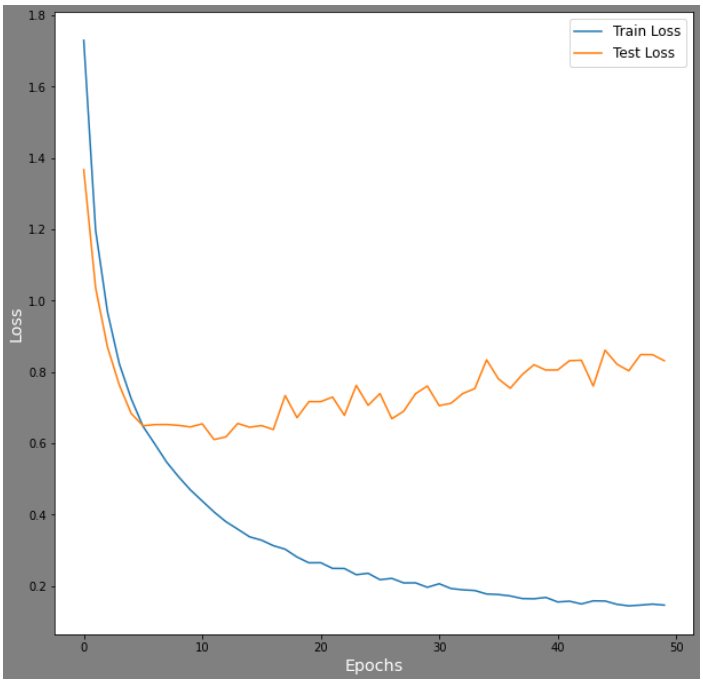
Train Accuracy	Test Accuracy	Train Time
95,34%	78,45%	2min

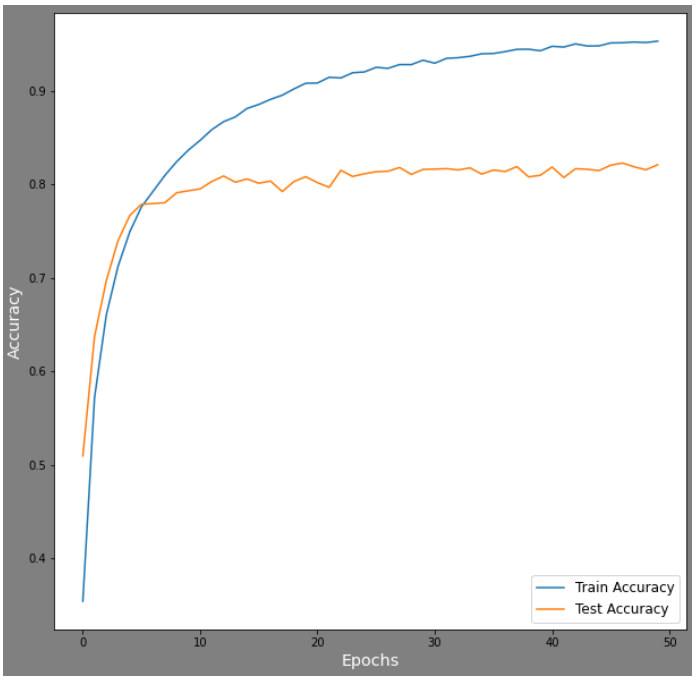
Indeed, the new network increases the accuracy, but the phenomenon of over-fitting is now stronger than ever, so we introduce Dropout Layers (20%) and train for 50 seasons.





Train Accuracy	Test Accuracy	Train Time
95,34%	82,11%	9min



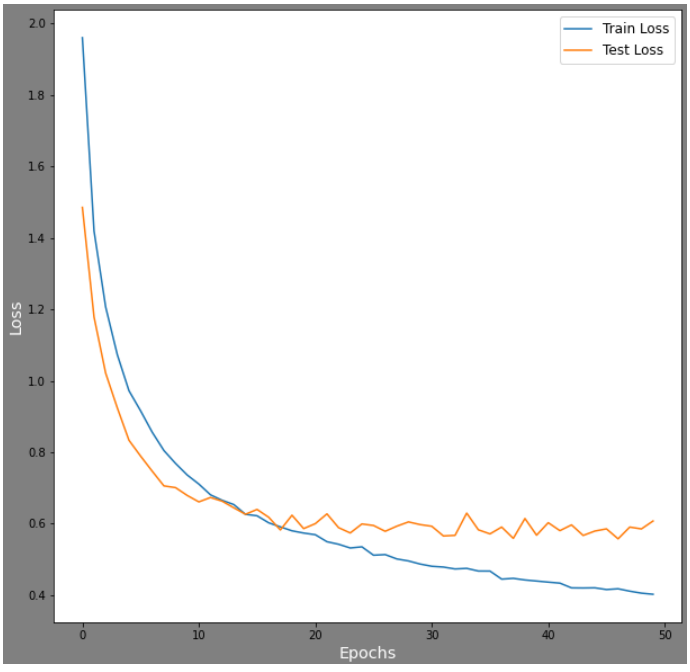


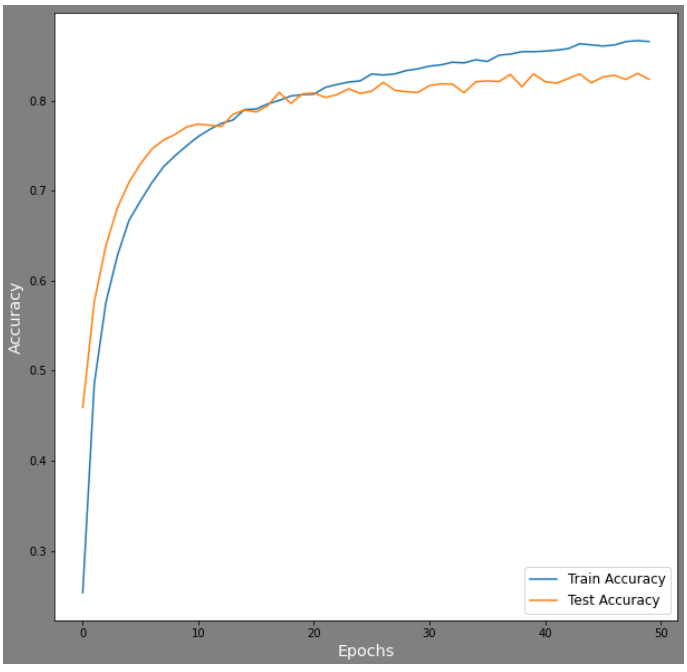
We notice that with these changes, the accuracy of the NN has increased significantly and overtraining has been limited to a certain extent, but it still exists. For this reason we do a final test, where in each layer we increase the probability of Dropout (0.2->0.3->0.4 in Convolutional & 0.5 in Dense).

Train Accuracy	Test Accuracy	Train Time
86,57%	82,39%	9min

Indeed, our final model has managed to limit overtraining to a point where it does not concern us, as can be seen in the learning curves.

Comparison to kNN & NC





From Neural Networks, the last CNN implemented is selected:

Convolutional Neural Network		
Train Accuracy	Test Accuracy	Train Time
86,57%	82,39%	9min

We also choose for our comparison an MLP, namely the one with 4 Hidden Layers and Dropout:

Multilayer Perceptron		
Train Accuracy	Test Accuracy	Train Time
63,32%	55,81%	2min 28s

The results of kNN and CNN algorithms:

1-Nearest Neighbor		
Metric	Accuracy	Runtime
Euclidean	35,39%	1min 35s
Manhattan	38,59%	26min

3-Nearest Neighbor			
Metric	Weights	Accuracy	Runtime
Euclidean	Uniform	33,03%	1min 35s
Euclidean	Distance	35,69%	1min 35s
Manhattan	Uniform	36,25%	26min
Manhattan	Distance	39,39%	25min 58s

Nearest Centroid		
Metric	Accuracy	Runtime
Euclidean	27,74%	0s
Manhattan	27,39%	8s

Taking a look at the accuracy of the models alone, it is clear that the NNs are the best choice. Even MLP, which lags far behind CNN, manages to beat the kNN & NC classifiers comfortably. The training time is quite low in all cases so it does not concern us. Finally, a fact that must be taken into account is the time it took to develop the final model in each case with that of NN being much longer than the others, but with the accuracy they achieve we cannot even consider the problem solvable, and so the obvious choice is Neural Networks with CNN.