CNN Classification

Authors: **Claudio Colturi, Luca Manzi**

Table of content

1. Introduction

In this laboratory session we experimented the usage of a convolutional neural network in order to classify the *10* classes in CIFAR10 dataset. This dataset is composed by *50000* training and *10000* test images of *32x32* pixels.
An example of the images in the dataset is shown in Figure 1.

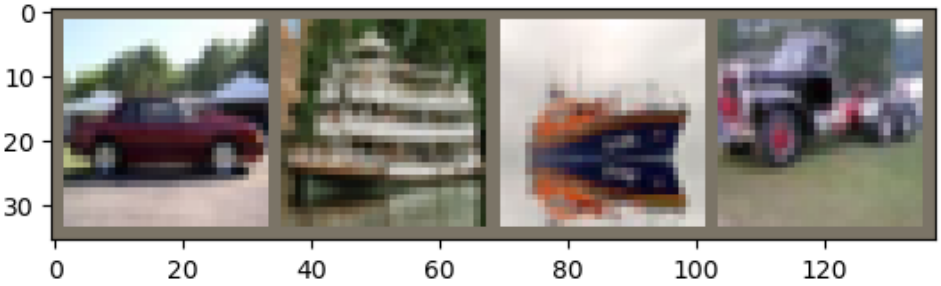

Figure 1: Example of the images in CIFAR10 dataset that we want to classify.

We tried to train different CNN changing both the training parameters and the network structure, in this paper we report the most significant experiments and the flow we followed in order to obtain our best model.

2. Standard model with 2 convolutional layers

We started experimenting with the standard model provided during the laboratory lesson. The summary of this model is reported in Figure 2:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             456
         MaxPool2d-2            [-1, 6, 14, 14]               0
            Conv2d-3           [-1, 16, 10, 10]           2,416
         MaxPool2d-4             [-1, 16, 5, 5]               0
            Linear-5                  [-1, 120]          48,120
            Linear-6                   [-1, 84]          10,164
            Linear-7                   [-1, 10]             850
================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
----------------------------------------------------------------
```

Figure 2: Summary of the CNN model used with 2 convolutional layers .

The parameters and activation functions used are:

- optimizer: SGD with momentum = 0.9
- criterion: Cross Entropy Loss
- learning rate: 0.001
- activation function convolutional layers: ReLu
- activation function linear layers: ReLu

The following plots shows the results obtained for the training and test loss (Figure 3, left) and the accuracy scored (Figure 3, right) in function of the epochs.

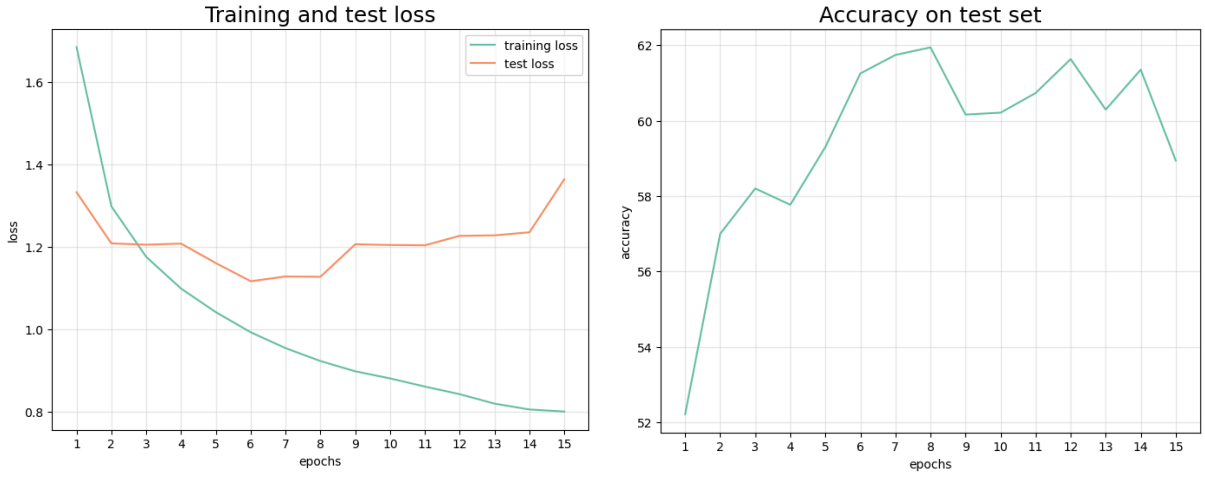

Figure 3: left: training and test loss;
right: accuracy on test set.

As we can see from the graphs above the test loss tends to saturate starting from the 8^{th} epoch: at this stage the accuracy scored is near 62%. After this epoch the training loss keeps on decreasing "regularly" (meaning that we chose a good learning rate) while the test loss tends to increase: this is a sign of overfitting that we should try to prevent in the next models.

3. Model selection and improving

The first step was a preliminary analysis for hyperparameters and activation functions, we fixed number of layers, and made a grid search using **MLFlow tracking**, trying multiple quickly trainable models, with different parameters.

3.1. Mlflow Tracking

Using MLFlow Tracking [1], we automated a training process for 4 models. The base architecture is composed by:

- 4 convolutional layers
  - kernel size 3
  - padding 1
  - with 3, 32, 64, 128 filters
- (2, 2) pooling layer to halve dimensions
- 3 fully connected layers

We tracked accuracy, epochs needed to achieve it, duration of the training phase. We used cap at 20 epochs, with 5 epochs of patience (see Section 3.2).

We tried all the different combinations of:

Models:

- **relu, no normalization** - relu for both convolutional and linear layers
- **relu, batch normaliz** - relu for both convolutional and linear layers, and batch normalization for convolutional layers
- **leaky relu, no dropout** - relu for convolutional layer and leaky relu for linear layers, and batch normalization for convolutional layers
- **leaky relu, dropout** - relu for convolutional layer and leaky relu for linear layers, batch normalization for convolutional layers, 0.2 dropout layers for linear layers

Learning rate:

- **0.001**
- **0.005**

Optimizer:

- **Adam**
- **SGD with momentum 0.5**
- **SGD with momentum 0.9**

Results are shown in Table 1.

	Model	Duration	Learning Rate	Momentum/Adam	Accuracy	Epochs Needed
0	leaky relu, dropout	4.5 min	0.001	0.9	67.39%	1
1	leaky relu, dropout	5.8 min	0.001	0.5	67.70%	2
2	leaky relu, dropout	9.8 min	0.001	Adam	67.12%	4
3	leaky relu, dropout	6.9 min	0.005	0.9	60.27%	4
4	leaky relu, dropout	5.4 min	0.005	0.5	60.77%	2
5	leaky relu, dropout	6.6 min	0.005	Adam	45.06%	2
6	leaky relu, no dropout	9.2 min	0.001	0.9	74.79%	7
7	leaky relu, no dropout	4.6 min	0.001	0.5	74.93%	1
8	leaky relu, no dropout	5.4 min	0.001	Adam	73.71%	1
9	leaky relu, no dropout	10.6 min	0.005	0.9	73.68%	9
10	leaky relu, no dropout	10.7 min	0.005	0.5	**75.69%**	9
11	leaky relu, no dropout	11.8 min	0.005	Adam	75.45%	8
12	relu, batch normaliz.	10.5 min	0.001	0.9	Not Converging	9
13	relu, batch normaliz.	14.9 min	0.001	0.5	Not Converging	20
14	relu, batch normaliz.	13.5 min	0.001	Adam	Not Converging	9
15	relu, batch normaliz.	15.1 min	0.005	0.9	Not Converging	19
16	relu, batch normaliz.	15.0 min	0.005	0.5	Not Converging	20
17	relu, batch normaliz.	26.2 min	0.005	Adam	Not Converging	20
18	relu, no normalization	12.9 min	0.001	0.9	Not Converging	20
19	relu, no normalization	12.8 min	0.001	0.5	Not Converging	16
20	relu, no normalization	16.6 min	0.001	Adam	Not Converging	16
21	relu, no normalization	13.1 min	0.005	0.9	Not Converging	20
22	relu, no normalization	13.1 min	0.005	0.5	Not Converging	19
23	relu, no normalization	26.4 min	0.005	Adam	Not Converging	20

Table 1: Rzesults for combinations of different models with different parameters, the best performance is in bold.

3.2. Dealing with number of epochs

Since we didn't know in advance how many epochs we needed for training phase, we implemented by hand a *patience* based approach getting a max number of epochs without improvement. After each epoch, we compute the accuracy on the training set:

- if the accuracy improves, we reset the *patience* and we update our best model.
- if it is not improving for n consecutive rounds, we stop the training, returning the best model.

3.3. Dealing with learning rate

Using an appropriate learning rate, we tested that we could train our model in less than 15 epochs. Nevertheless, we implemented an adaptive learning rate, using a scheduler, that makes it explore more in the beginning, and then it decreases the rate in the later steps.

3.4. Dealing with overfitting

When complicating the model, increasing trainable parameters, often we were reaching really low losses in the training set, but the losses of the test set were increasing. The model was starting to "learn" train set images, that is something we wanted to avoid.

We tried 2 techniques:

- adding Dropout layers
- using an activation function robust to overfitting, we used Gaussian ReLu

Dropout works forcing the network to make a prediction “switching off” some neurons during the training phase, that makes the network more able to generalize, focusing on the entire architecture instead of a specific area, as it may happen when the network “learns” an image.

4. Models with more convolutional layers

4.1. Model with 3 convolutional layers

The model used in this section is shown in Figure 4:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             896
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 13, 13]          18,496
         MaxPool2d-4            [-1, 64, 6, 6]               0
            Conv2d-5            [-1, 64, 4, 4]          36,928
         MaxPool2d-6            [-1, 64, 2, 2]               0
            Linear-7                 [-1, 128]          32,896
           Dropout-8                 [-1, 128]               0
            Linear-9                  [-1, 96]          12,384
          Dropout-10                  [-1, 96]               0
           Linear-11                  [-1, 84]           8,148
           Linear-12                  [-1, 10]             850
================================================================
Total params: 110,598
Trainable params: 110,598
Non-trainable params: 0
----------------------------------------------------------------
```

Figure 4: Summary of the CNN model used with 2 convolutional layers .

In this case we added a third convolutional layer followed by another pooling. Between the fully connected layers we added some dropouts layers in order to deal with overfitting.

The parameters and activation functions used are:

- optimizer: SGD with momentum = 0.9
- criterion: Cross Entropy Loss
- learning rate: 0.001
- activation function convolutional layers: ReLu
- activation function linear layers: GeLu

The results obtained are shown in Figure 5:

Figure 5: left: training and test loss;
right: accuracy on test set.

In this case we run the model for a greater number of epochs, but we can see from the images above that after the 5^{th} epoch the test tends to saturate until the 15^{th} epoch when it starts to increase. Looking also the accuracy graph, we can say that the best number of epochs to train this model is between 10 and 15, with a maximum of accuracy at **73%**.

4.2. Model with 5 convolutional layers

We decided to build up a more deep model with five convolutional layers. The structure of the network is showed in Figure 6

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 32, 32]             896
       BatchNorm2d-2          [-1, 32, 32, 32]              64
            Conv2d-3          [-1, 64, 32, 32]          18,496
       BatchNorm2d-4          [-1, 64, 32, 32]             128
         MaxPool2d-5          [-1, 64, 16, 16]               0
            Conv2d-6         [-1, 128, 16, 16]          73,856
       BatchNorm2d-7         [-1, 128, 16, 16]             256
         MaxPool2d-8           [-1, 128, 8, 8]               0
            Conv2d-9           [-1, 256, 8, 8]         295,168
      BatchNorm2d-10           [-1, 256, 8, 8]             512
           Conv2d-11           [-1, 256, 8, 8]         590,080
      BatchNorm2d-12           [-1, 256, 8, 8]             512
        MaxPool2d-13           [-1, 256, 4, 4]               0
          Dropout-14                 [-1, 4096]               0
           Linear-15                  [-1, 512]       2,097,664
          Dropout-16                  [-1, 512]               0
           Linear-17                  [-1, 128]          65,664
          Dropout-18                  [-1, 128]               0
           Linear-19                   [-1, 84]          10,836
           Linear-20                   [-1, 10]             850
================================================================
Total params: 3,154,982
Trainable params: 3,154,982
Non-trainable params: 0
----------------------------------------------------------------
```

Figure 6: Summary of the CNN model used with 2 convolutional layers .

The hyperparameters tuning lead to: The parameters and activation functions used are:

- optimizer: SGD with momentum = 0.8
- criterion: Cross Entropy Loss
- learning rate: 0.001
- activation function convolutional layers: ReLu
- activation function linear layers: GeLu

As we can see in the following picture (Figure 7) this model provided good performance on the test set, reaching an accuracy score near to **85%**.

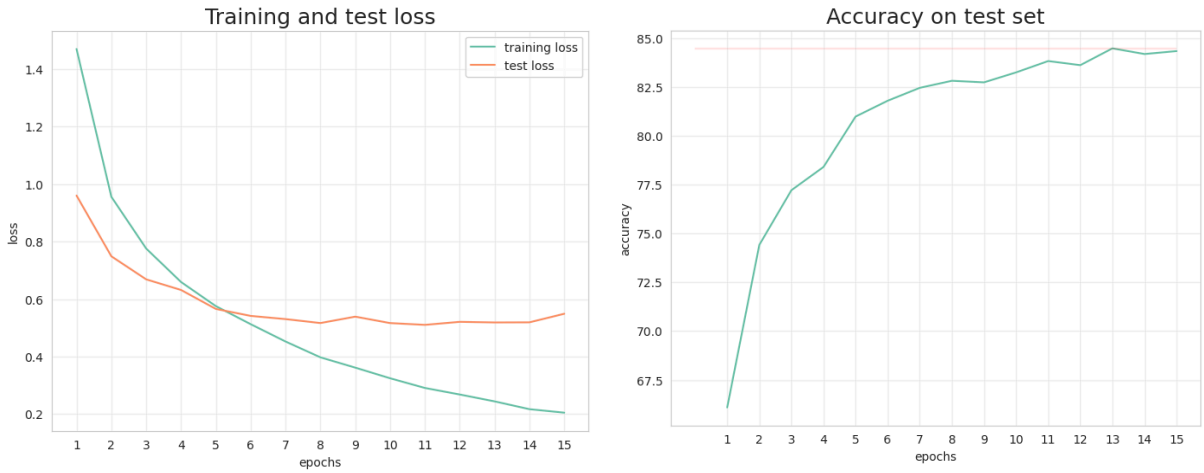

Figure 7: left: training and test loss;
right: accuracy on test set.

The ability of this model to generalize better to new data is due to the large number of parameters (over 3 milions). The drawback of such big model is the fact that we had to deal with overfitting, in particular we had to consider both batch normalization and dropout for the model.

4.3. Even Deeper models: CerberusNet

After a first analysis seems like that the go-to approach to improve performance was to add more convolutional layers. It keeps the training more efficient than using a lot of linear layers, and leaverages the spaciality of data.

A notable exception to the rule, was a model [2] that scored **86%** of accuracy, using 4 convolutional layers with padding, only one pooling layer and 3 huge linear layers. This model has 9 million trainable parameters, that had a completely different approach, and focused more on the complexity of the linear layers. This was something we wanted to avoid.

In [3], the authors describe a process of optimization of convolutional neural network parameters for image classification. They used stacks of building blocks composed by convolutional layers and ReLu activation functions, changing only kernel size, learning rate, number of filters and number of the building blocks (layers) added. For the CIFAR10 dataset, they reach the best accuracy (**81%**) combining 14 layers, using 28 filters with a kernel size of 5x5.

The model we proposed **CerberusNet** is a deep model composed by 12 convolutional layers and 2 linear ones. The convolutional part is composed by 3 macro blocks (hence the name), each of them composed by 4 blocks of this single unit:

- convolutional layer (kernel 3x3, padding 1, up to 256 filters)
- ReLu activation function
- Batch Normalization

Then after each macro block, we applied a Pooling layer to halve the dimensions. After the convolutional part, we applied 2 highly regularized fully connected layers, using dropout (to avoid overfitting).

The architecture can be seen in Figure 8.

The dimension in the convolutional part remains the same (due to padding and kernel size), halving 3 times, until obtaining a 4 x 4 x 256 filters.

As expected, having a high latent space paid off, also we didn't get deep enough to incur in vanishing gradient, and we obtained an accuracy of **82.1%**.

The main drawback was the really high computational cost, due to the 3 million parameters to optimize.

Something we understood is that there is no golden rule, no "free lunch", but it is necessary a lot of tuning and exploration, also depends heavily on the dataset and the task, to find an acceptable tradeoff.

Layer (type:depth-idx)	Output Shape	Param #
├─Conv2d: 1-1	[-1, 64, 32, 32]	1,792
├─BatchNorm2d: 1-2	[-1, 64, 32, 32]	128
├─Conv2d: 1-3	[-1, 64, 32, 32]	36,928
├─BatchNorm2d: 1-4	[-1, 64, 32, 32]	128
├─Conv2d: 1-5	[-1, 64, 32, 32]	36,928
├─BatchNorm2d: 1-6	[-1, 64, 32, 32]	128
├─Conv2d: 1-7	[-1, 64, 32, 32]	36,928
├─BatchNorm2d: 1-8	[-1, 64, 32, 32]	128
├─MaxPool2d: 1-9	[-1, 64, 16, 16]	--
├─Conv2d: 1-10	[-1, 128, 16, 16]	73,856
├─BatchNorm2d: 1-11	[-1, 128, 16, 16]	256
├─Conv2d: 1-12	[-1, 128, 16, 16]	147,584
├─BatchNorm2d: 1-13	[-1, 128, 16, 16]	256
├─Conv2d: 1-14	[-1, 128, 16, 16]	147,584
├─BatchNorm2d: 1-15	[-1, 128, 16, 16]	256
├─Conv2d: 1-16	[-1, 128, 16, 16]	147,584
├─BatchNorm2d: 1-17	[-1, 128, 16, 16]	256
├─MaxPool2d: 1-18	[-1, 128, 8, 8]	--
├─Conv2d: 1-19	[-1, 256, 8, 8]	295,168
├─BatchNorm2d: 1-20	[-1, 256, 8, 8]	512
├─Conv2d: 1-21	[-1, 256, 8, 8]	590,080
├─BatchNorm2d: 1-22	[-1, 256, 8, 8]	512
├─Conv2d: 1-23	[-1, 256, 8, 8]	590,080
├─BatchNorm2d: 1-24	[-1, 256, 8, 8]	512
├─Conv2d: 1-25	[-1, 256, 8, 8]	590,080
├─BatchNorm2d: 1-26	[-1, 256, 8, 8]	512
├─MaxPool2d: 1-27	[-1, 256, 4, 4]	--
├─Linear: 1-28	[-1, 128]	524,416
├─Dropout: 1-29	[-1, 128]	--
├─Linear: 1-30	[-1, 10]	1,290

Total params: 3,223,882
Trainable params: 3,223,882
Non-trainable params: 0
Total mult-adds (M): 379.78

Input size (MB): 0.01
Forward/backward pass size (MB): 7.00
Params size (MB): 12.30
Estimated Total Size (MB): 19.31

Figure 8: Architecture of the model in our analysis, that scored 82.1% of accuracy.

We trained the model using both concept of patience Section 3.2, and decaying learning rate Section 3.3, the plot of the losses on the train and test set is shown in Figure 9.

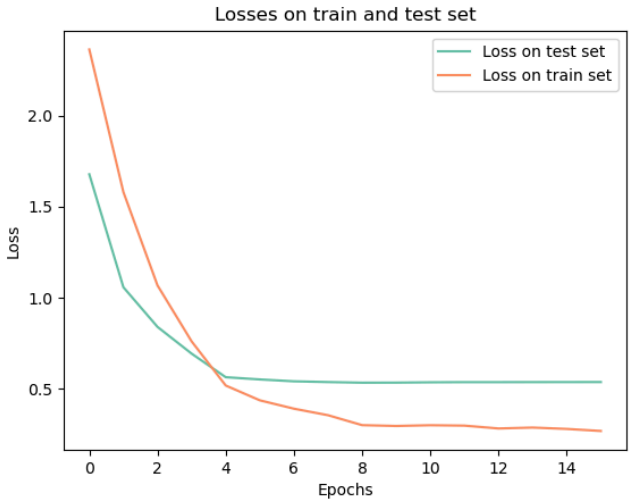

Figure 9: Plot showing how MSE losses evolved for both training and test set over the course of the epochs. We can observe how, since epoch 4, test set does not decrease any more

5. Error analysis

We can analyse prediction errors, plotting a confusion matrix Figure 10. It has been plotted fitting the model described at Section 4.3 . On the y axis we have true labels, and on the x axis the predictions obtained with our model.

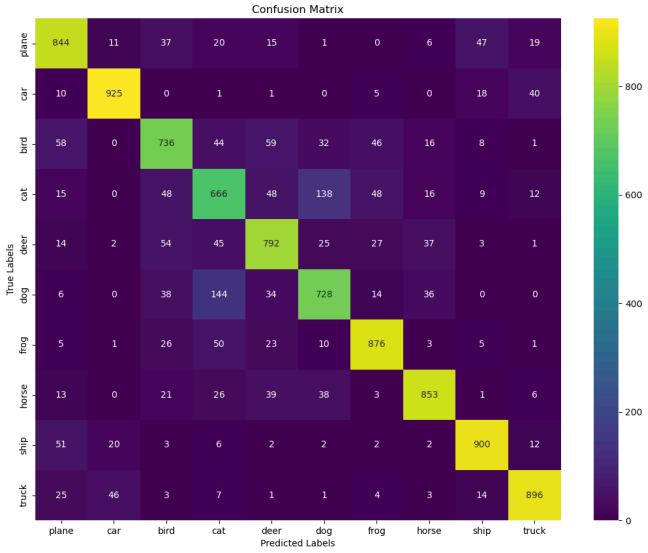

Figure 10: Confusion matrix obtained after fitting a 12-Layer model CerberusNet to the dataset CIFAR10, with an accuracy for the predictions of 82%.

We can remove the diagonal, to have a better understanding of the misclassification errors, and we obtain the plot Figure 11.

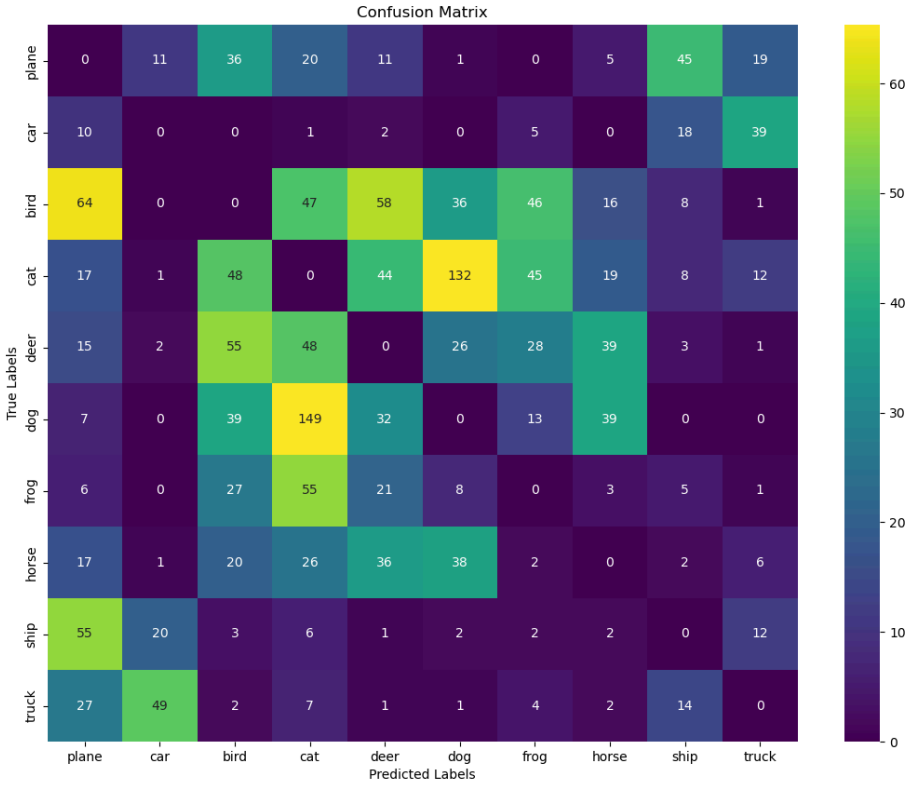

Figure 11: Confusion matrix obtained after fitting a 12-Layer model CerberusNet to the dataset CIFAR10, with an accuracy for the predictions of 82%, removing the correct predictions.

We can note that:

- a lot of true cats are mistaken as dogs, and vice versa Figure 12
- it has some troubles classifying birds, maybe is due to the variety of colors. Expecialy we can note a lot of birds classified as planes Figure 13 (but more of the double with respect to planes-birds misclassifications, a point of investigation)

Figure 12: Misclassification between dogs and cats.

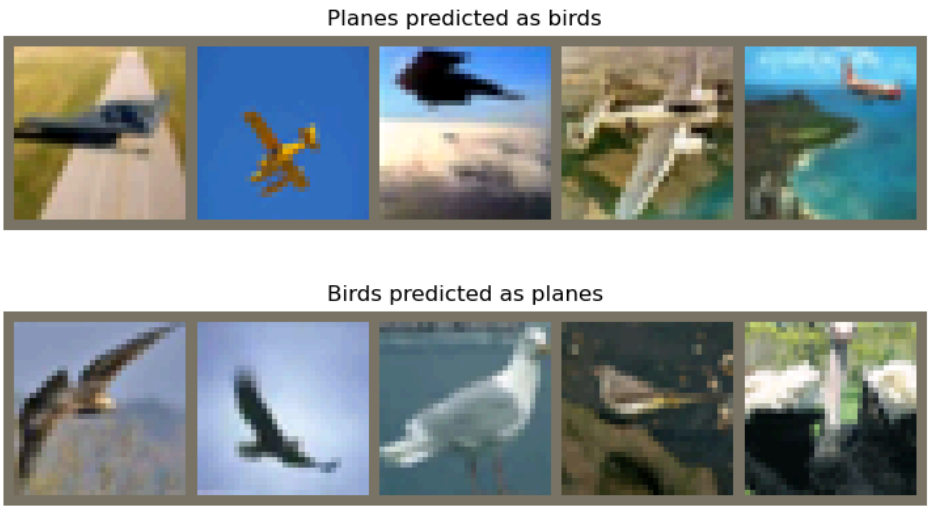

Figure 13: Misclassifications between birds and planes.

We could visualize it as an histogram in Figure 14, where, in the other hand, we can clearly see that the model performs very well in classificating cars, ships and trucks.

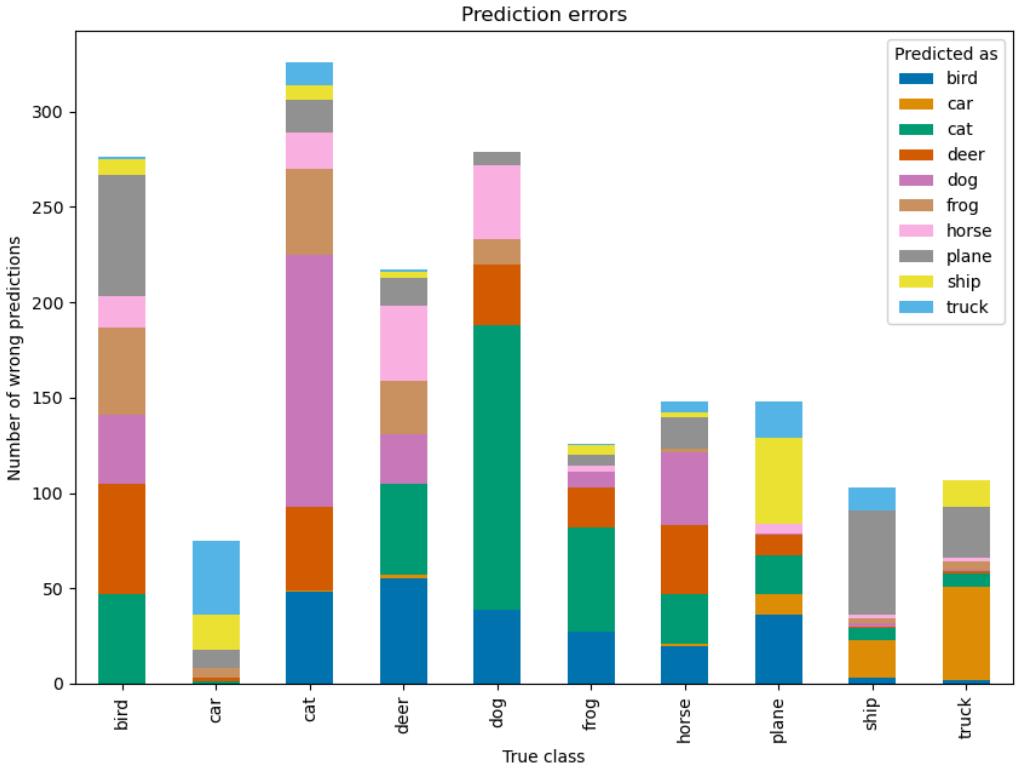

Figure 14: Histogtram containing prediction errors

6. Conclusion

In this project we explored different CNN structures and parameters in order to understand better how they work. We saw that a deeper model with more parameters is able to predict better the test set with respect to the first example provided at lesson. In particular we saw that instead of using

ReLu activation function turned out that a smoother activation function (like Gelu and leaky Relu) helps to obtain better results.
In order to deal with overfitting we implemented batch normalization on convolutional layers and dropout on the fully connected part of the network.
The best result has been obtained by the architecture presented in Section 4.2, were we achieved a test accuracy near **85%**.The result is similar to that obtained with CerberusNet Section 4.3, model for which we studied performance by error analysis.

Bibliography

[1] mlflow, "MLFlow." GitHub, 2024.

[2] jameschengpeng, "PyTorch-CNN-on-CIFAR10." GitHub, 2019.

[3] T. Sinha, B. Verma, and A. Haidar, "Optimization of convolutional neural network parameters for image classification," in *2017 IEEE symposium series on computational intelligence (SSCI)*, 2017, pp. 1–7.