January 2025

Sapienza Università di Roma

Machine Learning

Homework 2

1 Preliminary remarks

To properly solve this problem, it is very important to make some initial considerations before moving on to the architecture design of the convolutional neural network.

1.1 Unbalanced and limited dataset

Let us examine the training set.

- **Class 0** = 1000 images;
- **Class 1** = 1500 images;
- **Class 2** = 1500 images;
- **Class 3** = 2000 images;
- **Class 4** = 369 images;

As we can see, the training set contains a total of 6369 images. The images associated with class 3 are the most numerous, while classes 0 and 4 are the ones with the fewest samples. We can also observe that the training set is limited and unbalanced. Moreover, this situation is likely to get worse, as part of the training set will have to be reserved to become the validation set. Having a limited training set with underrepresented classes increases the difficulty of obtaining good results, and if we consider that using a CNN for this problem will not give us optimal performance, as it would be better to use Reinforcement Learning, the goal becomes even more complicated. To improve performances, to correctly train the model and to prevent overfitting, it will be very important to consider dataset augmentation, the CNN architecture, and evaluate regularization techniques. Finally, when evaluating the model, it will be important to consider not only the accuracy but also the other fundamental classification evaluation metrics.

1.2 Chosen approaches and hyperparameters

To solve this task, I decided to create two different CNN architectures and to use two different optimizers: Adam and SGD with momentum. The tests performed and the results obtained will now be described. The library I used for the implementation is PyTorch.

2 First approach

2.1 Data augmentation

As a first step, given the considerations made previously, I performed data augmentation by applying transformations to the images of the training set. These transformations include: **random rotation of 5 degrees, random horizontal flipping and random resize crop**. I decided not to use too "aggressive" transformations or to add any others in order to avoid adding too much noise and getting bad performances. I have therefore reserved 1/3 of the training set for the validation set and applied data augmentation only on the training set.

2.2 CNN architecture design

For the first architecture, I wanted to create a CNN with multiple convolutional layers, each followed by a pooling operation. This will allow to obtain feature maps with reduced height and width but with greater depth and this, just as defined by the guidelines, is a good way to obtain a significant detection of the most important features of each image. The **first convolutional layer** consists of **6 5x5x3** kernels. The third dimension set to 3 is obviously needed since RGB images have three channels to model the colors, while setting six small filters with width and height equal to 5 is a common effective choice for the first convolutional layer. I also set a **padding** initialized to 0 (of size 2 in this case, because the kernel width and height are set to 5) and stride equal to 1 so that the convolution is performed on all the pixels of the images. I made this choice because, given the limited dataset and the fact that the size of the images is quite small, I thought it was important not to leave out any pixels to avoid losing useful details. To introduce non-linearity, I used **Rectified Linear Units (ReLU)** activation function because it usually offers good performances and correctness. In addition, ReLU does not suffer significantly from saturation, and this facilitates network configuration and task completion. Now, to stabilize and speed up the model training, I added **batch normalization**. Finally, to obtain a reduction of the image width and height, I inserted a **pooling with size of 2x2 and stride of 2** which performs an average operation. I chose the average operation because, during the various tests I performed, the max operation led very quickly to overfitting. This may mean that this type of task, with this type of architecture, benefits more from an average operation that enhances the entire area rather than from a max operation that gives greater importance to the most marked details. The second and third convolutional layers are analogous to the first, except that they use 16 and 32 kernels, respectively. Additionally, for the third layer, I set the kernel width and height to 3 and consequently set the padding size to 1, so that the kernel size remain significantly smaller than the output of the second convolutional layer and thus better capture the small characteristics. At this point, I noticed that with additional convolutional layers, or by further increasing the depth of the feature maps, I was starting to get significant overfitting. So, I decided to include the fully connected network architecture here. I therefore added the **flattern operation**, **two layers with 64 and 32 neurons respectively** and an **output layer with 5 neurons** since a classification into five classes must be performed. I also inserted a **20% dropout** after these first two layers to both reduce overfitting and not penalizing too much the training loss. The **batch size is 64**, that is a common choice, the output activation function is obviously **softmax** and the loss function is **Cross Entropy Loss**. The Pytorch code snippet to create this architecture is the following (if Cross Entropy Loss is chosen, Pytorch will automatically use softmax after the output layer):

```
self.model = nn.Sequential(
            nn.Conv2d(3, 6, kernel_size = 5, stride = 1,
                        padding = 2),
            nn.ReLU(),
            nn.BatchNorm2d(6),
            nn.AvgPool2d(2, stride = 2),

            nn.Conv2d(6, 16, kernel_size = 5, stride = 1,
                        padding = 2),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.AvgPool2d(2, stride = 2),

            nn.Conv2d(16, 32, kernel_size = 3, stride = 1,
                        padding = 1),
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.AvgPool2d(2, stride = 2),

            nn.Flatten(),
            nn.Linear(32 * 12 * 12, 64),
            nn.ReLU(),
            nn.Dropout(0.2),

            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(32, 5)
        )
```

2.3 CNN1 with Adam optimizer

The first optimizer I chose is Adam, as it is generally known to bring good results. I set the **learning rate to 0.0001** because, from the various tests performed, I noticed that with lr = 0.001 the loss on the validation set, after a quite reduced number of epochs, began to oscillate. This may be a sign of the fact that, for this problem and with this architecture, it is a too high learning rate that leads to difficulties for the model to converge. Let us now present and describe the results obtained.

Class	Precision	Recall	F1-Score	Support
0	0.583	0.244	0.344	659
1	0.647	0.632	0.639	1005
2	0.650	0.730	0.687	988
3	0.549	0.759	0.637	1344
4	0.550	0.044	0.081	250
Macro Avg	0.596	0.482	0.478	4246
Weighted Avg	0.601	0.600	0.571	4246

Table 1: Performance metrics of the training phase.

Class	Precision	Recall	F1-Score	Support
0	0.669	0.284	0.399	341
1	0.671	0.519	0.585	495
2	0.673	0.713	0.693	512
3	0.514	0.823	0.633	656
4	1.000	0.025	0.049	119
Macro Avg	0.706	0.473	0.472	2123
Weighted Avg	0.641	0.594	0.566	2123

Table 2: Performance metrics of the validation phase.

Metric	Value
Accuracy (Training)	0.600
Accuracy (Validation)	0.594

Table 3: Accuracy of the training and validation phases.

Figure 1: Training and validation loss.

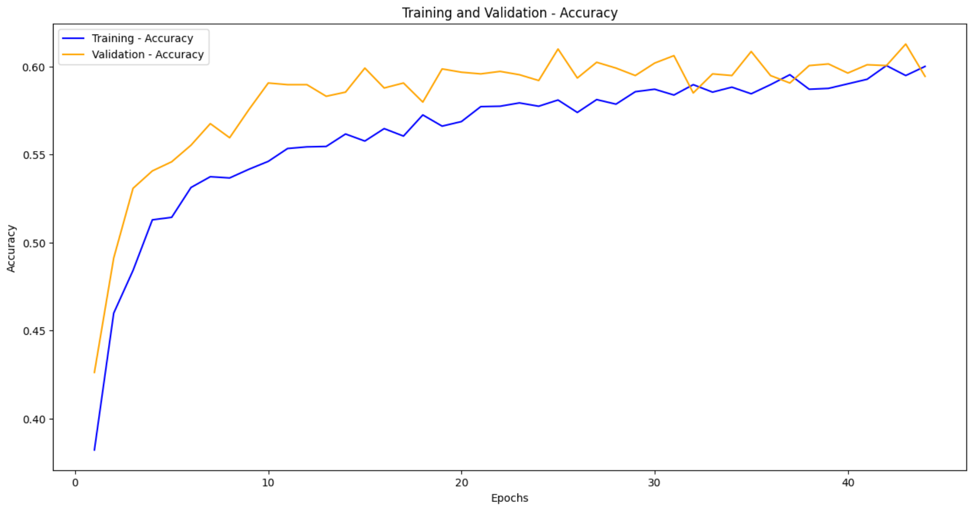

Figure 2: Training and validation accuracy.

The training phase took **12 minutes** for a total of **44 epochs** as I noticed that, with a larger number of epochs, the training and validation losses, along with the other metrics, did not improve significantly. Now, let us see the test results.

Class	Precision	Recall	F1-Score	Support
0	0.387	0.323	0.352	133
1	0.396	0.458	0.425	275
2	0.517	0.692	0.592	406
3	0.836	0.778	0.806	1896
4	0.000	0.000	0.000	39
Macro Avg	0.427	0.450	0.435	2749
Weighted Avg	0.711	0.701	0.703	2749

Table 4: Performance metrics of the test phase.

Metric	Value
Accuracy (Test)	0.701

Table 5: Accuracy of the test phase.

Test loss is **0.8798**.
Analyzing the results of the testing phase, we can notice, as anticipated, that performances are not optimal. However, considering the use of CNN and the limitations of the training and test sets, we can conclude that, for this classification problem with five classes, an accuracy of 0.701 is quite good. Looking at the other metrics, we can see that the quality of the results is proportional to the number of samples associated with the respective classes. In particular, class 3, the one with the most samples, shows good values of all the metrics: Precision, Recall and F1-Score. On the contrary, class 4, which is the one with only 39 samples, shows the worst performances and unfortunately the model was not able to correctly classify any image in this class. However, if we look at the performances of the validation phase, we can notice that the model was indeed able to make correct predictions also for this class. In particular, a validation Precision $= 1$ shows that the model correctly classified all the true positives it was able to find, but the low recall shows difficulty in identifying most of them. Another interesting aspect is that, even though predictions on class 1 and 2 showed coherent training and validation metrics, they decreased quite significantly during the testing phase. This, as the following

covariance matrix will illustrate, led to an increase in the number of inaccuracies concerning those classes. These differences between validation set and test set may be due to the very few samples in the test set. Following these results, it is logical to see that the macro average of the performances is significantly lower than the weighted average since the latter takes into account the number of samples for each class. The weighted average also shows improvement on the test set, demonstrating that the model can generalize quite well. Finally, the charts show how the loss correctly decreases following the gradient descent and how all the other metrics correctly increase.
Let us now analyze the covariance matrix of the test phase.

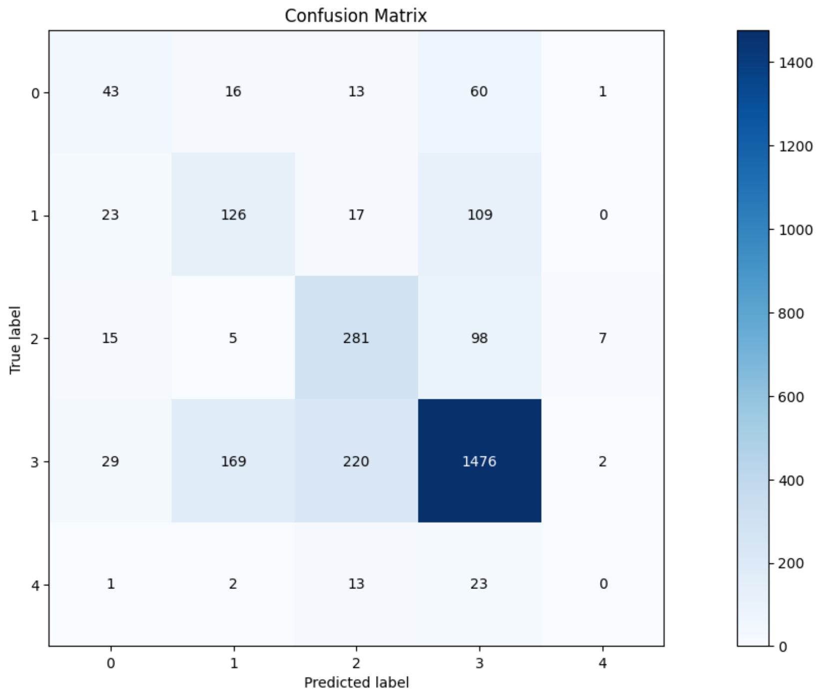

Figure 3: Covariance Matrix.

The image shows that the results obtained are perfectly in line with what has just been said. An index of sufficient performance can be seen by noting that, except for classes 0 and 4 which are the least represented, the diagonal of the matrix shows that most of the predictions have been performed correctly.

2.4 CNN1 with SGD and momentum

Let us now see the performance of this same architecture, with the same number of epochs, using SGD as the optimizer. To make the comparison, I decided to set the same learning rate as in the previous case, but adding a **momentum of 0.99** because SGD is generally more prone to getting stuck in local minima compared to Adam.

Class	Precision	Recall	F1-Score	Support
0	0.537	0.167	0.255	659
1	0.607	0.607	0.607	1005
2	0.627	0.725	0.672	988
3	0.534	0.751	0.624	1344
4	1.000	0.004	0.008	250
Macro Avg	0.661	0.451	0.433	4246
Weighted Avg	0.601	0.576	0.538	4246

Table 6: Training Results

Class	Precision	Recall	F1-Score	Support
0	0.630	0.284	0.392	341
1	0.697	0.432	0.534	495
2	0.682	0.746	0.713	512
3	0.502	0.843	0.629	656
4	0.000	0.000	0.000	119
Macro Avg	0.502	0.461	0.453	2123
Weighted Avg	0.583	0.587	0.554	2123

Table 7: Validation Results

Class	Precision	Recall	F1-Score	Support
0	0.423	0.308	0.357	133
1	0.430	0.378	0.402	275
2	0.493	0.727	0.588	406
3	0.828	0.792	0.810	1896
4	0.000	0.000	0.000	39
Macro Avg	0.435	0.441	0.431	2749
Weighted Avg	0.708	0.706	0.703	2749

Table 8: Test Results

Type	Accuracy
Training	0.576
Validation	0.587
Test	0.706

Table 9: Accuracy Results

Figure 4: Training and validation loss.

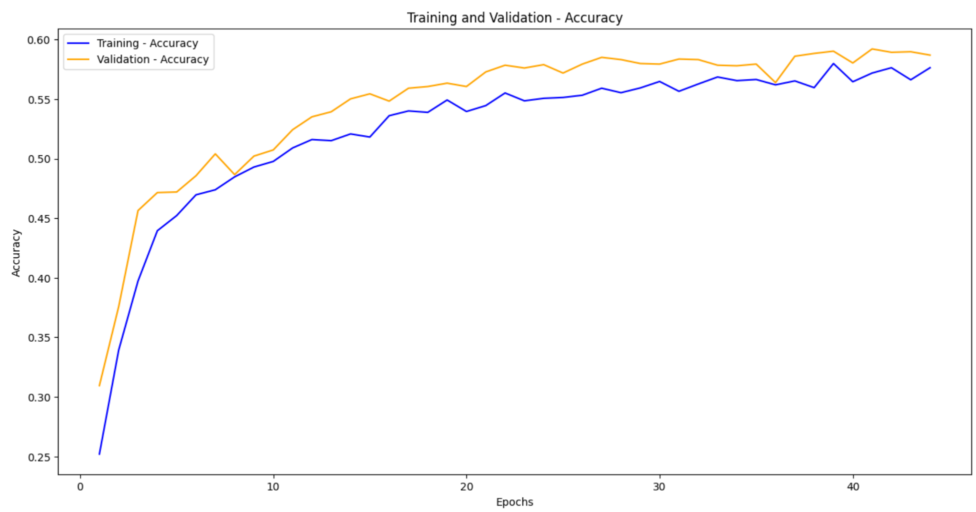

Figure 5: Training and validation accuracy.

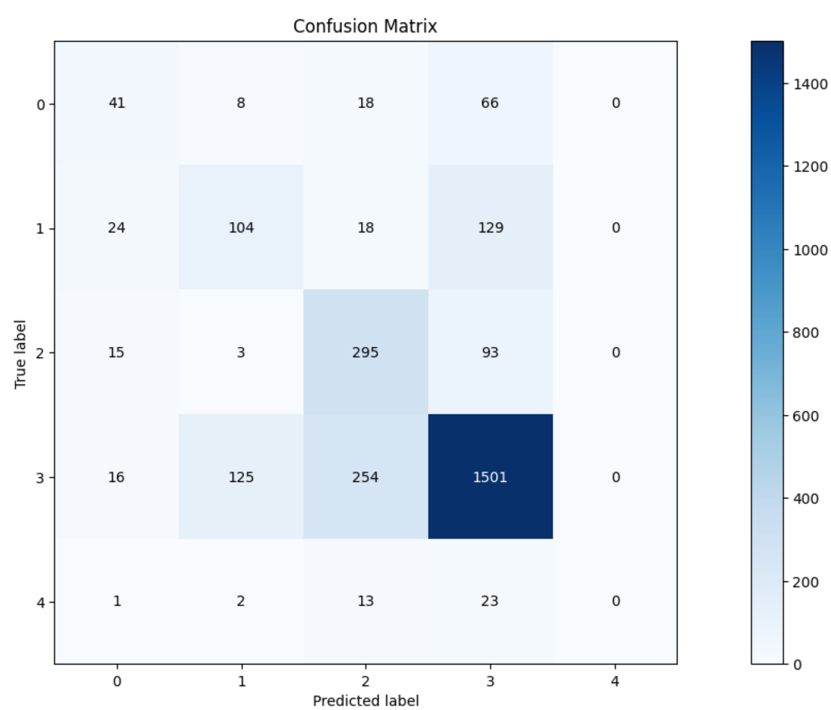

Figure 6: Testing Covariance Matrix.

The training phase took **12** minutes and the **test loss is 0.9073**.
As we can see, the results obtained are very similar to the previous case. The most significant differences are the higher loss and the fact that this model did not make it to perform correct predictions for class 0 even on the validation phase. This may be due to the fact that, as shown by the metrics tables, contrary to the previous case, here the model obtained Precision = 1 in the predictions related to class 4 in the training phase which then resulted in a Precision = 0 in the validation phase. This may mean

that the model, during training, overfitted the Precision of those predictions and this led it to not have enough generalization power. In fact, the most marked difference is the validation Precision, which is significantly lower than the previous model, meaning that Adam showed greater ability to correctly predict the class of the samples and less tendency to overfit. So, we can say that the previous model overall obtained better performances than this one. Anyway, also in this case, considering everything, the results obtained are still quite satisfactory.

3 Second approach

3.1 CNN architecture design

During the various tests I performed, I noticed that adding more convolutional layers resulted in a performance reduction, probably because the architecture was starting to be too complex for this problem. Therefore, for the second approach, I decided to remove a convolutional layer. Moreover, I removed dropouts and added a **L2 regularization** with a parameter **lambda = 0.01**. Since this architecture is "simpler" than the previous one, I decided to continue the training phase for **50 epochs**. In a clearer way, the Pytorch code snippet to create this architecture is the following:

```
self.model = nn.Sequential(
            nn.Conv2d(3, 6, kernel_size = 5, stride = 1,
                        padding = 2),
            nn.ReLU(),
            nn.BatchNorm2d(6),
            nn.AvgPool2d(2, stride = 2),

            nn.Conv2d(6, 16, kernel_size = 3, stride = 1,
                        padding = 1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.AvgPool2d(2, stride = 2),

            nn.Flatten(),
            nn.Linear(16 * 24 * 24, 64),
            nn.ReLU(),

            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 5)
        )
```

3.2 CNN2 with Adam optimizer

Let's analyze the performance of this architecture using **Adam with lr = 0.001** as optimizer. Training phase took **8 minutes**, so less than before.

Class	Precision	Recall	F1-Score	Support
0	0.632	0.287	0.395	659
1	0.666	0.614	0.639	1005
2	0.662	0.727	0.693	988
3	0.549	0.776	0.643	1344
4	0.706	0.096	0.169	250
Macro Avg	0.643	0.500	0.508	4246
Weighted Avg	0.625	0.610	0.587	4246

Table 10: Training Results

Class	Precision	Recall	F1-Score	Support
0	0.635	0.352	0.453	341
1	0.651	0.584	0.616	495
2	0.661	0.758	0.706	512
3	0.563	0.762	0.648	656
4	0.533	0.067	0.119	119
Macro Avg	0.609	0.505	0.508	2123
Weighted Avg	0.617	0.615	0.593	2123

Table 11: Validation Results

Class	Precision	Recall	F1-Score	Support
0	0.371	0.368	0.370	133
1	0.360	0.498	0.418	275
2	0.472	0.746	0.578	406
3	0.860	0.711	0.779	1896
4	0.080	0.051	0.062	39
Macro Avg	0.429	0.475	0.441	2749
Weighted Avg	0.718	0.669	0.683	2749

Table 12: Test Results

Dataset	Accuracy
Training	0.610
Validation	0.615
Test	0.669

Table 13: Accuracy for Training, Validation, and Test

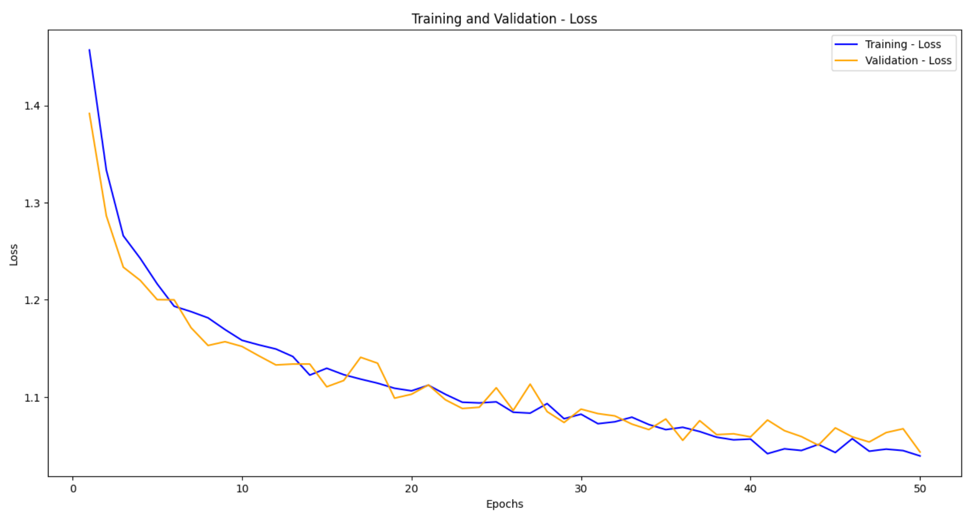

Figure 7: Training and validation loss.

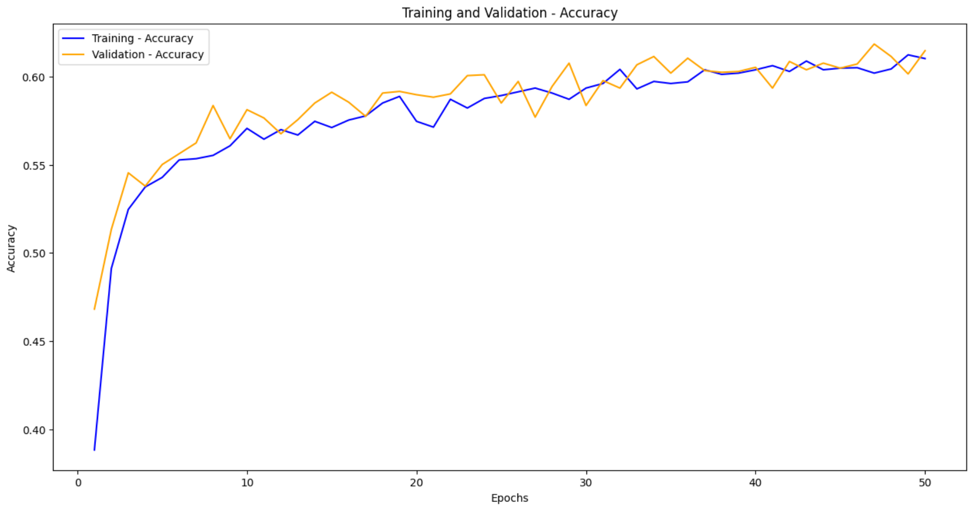

Figure 8: Training and validation accuracy.

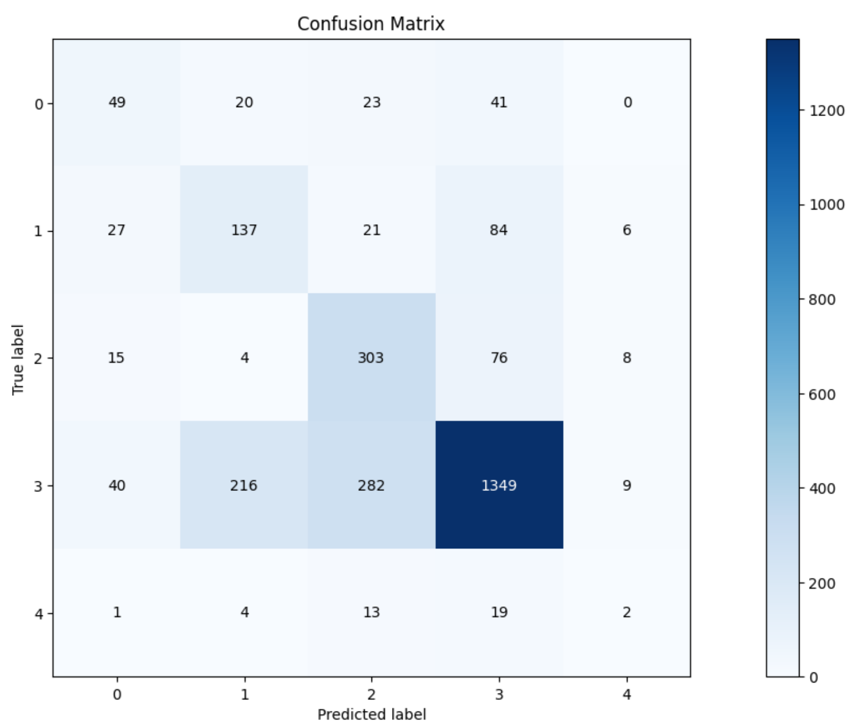

Figure 9: Testing Covariance Matrix.

The test loss is **0.9715**.
As we can observe, the model achieved performances similar to the previous cases. These performances are not optimal, but still quite good. Charts show how the loss correctly decreases following the gradient descent and how all the other metrics correctly increase. It is interesting to notice that, differently from the previous moodels, here the training phase continued to improve for 50 epochs. This may be due to the fact that this network is shallower than the previous ones, so the training is able to last

longer without starting overfitting. The most interesting aspect is, looking at the covariance matrix, that except for class 3, this model is the one that correctly predicted the highest number of samples for each class. It is also the only one that has so far managed to correctly predict some samples for class 4 also in testing phase. Looking at the metrics reported in the tables, specifically those of the validation set, we can see that this model got better performances in Recall metric. This means that the model was able to correctly identify a greater number of positive samples for each class, especially those related to number 4. Consequently, even though the precision of this model is slightly lower than that first one, its higher Recall allowed it to correctly identify and consequently classify a larger number of samples for most classes, resulting in overall greater performances. This shows that, for a problem with few data, a shallow architecture can have a better generalization power and that, in an unbalanced classification problem, accuracy is certainly not the only metric that matters.

3.3 CNN2 with SGD and momentum

Finally, let us test this architecture with **SGD with lr = 0.0001** and **momentum = 0.99**. The training phase took **8 minutes**, as the previous one.

Class	Precision	Recall	F1-Score	Support
0	0.601	0.294	0.395	659
1	0.664	0.628	0.645	1005
2	0.657	0.699	0.678	988
3	0.549	0.774	0.642	1344
4	0.692	0.072	0.130	250
Macro Avg	0.633	0.493	0.498	4246
Weighted Avg	0.618	0.606	0.583	4246

Table 14: Training Results

Class	Precision	Recall	F1-Score	Support
0	0.641	0.287	0.397	341
1	0.581	0.667	0.621	495
2	0.627	0.828	0.714	512
3	0.584	0.646	0.614	656
4	0.000	0.000	0.000	119
Macro Avg	0.487	0.486	0.469	2123
Weighted Avg	0.570	0.601	0.570	2123

Table 15: Validation Results

Class	Precision	Recall	F1-Score	Support
0	0.310	0.301	0.305	133
1	0.338	0.596	0.432	275
2	0.417	0.791	0.546	406
3	0.864	0.622	0.724	1896
4	0.000	0.000	0.000	39
Macro Avg	0.386	0.462	0.401	2749
Weighted Avg	0.707	0.620	0.638	2749

Table 16: Test Results

Dataset	Accuracy
Training	0.606
Validation	0.601
Test	0.620

Table 17: Accuracy for Training, Validation, and Test

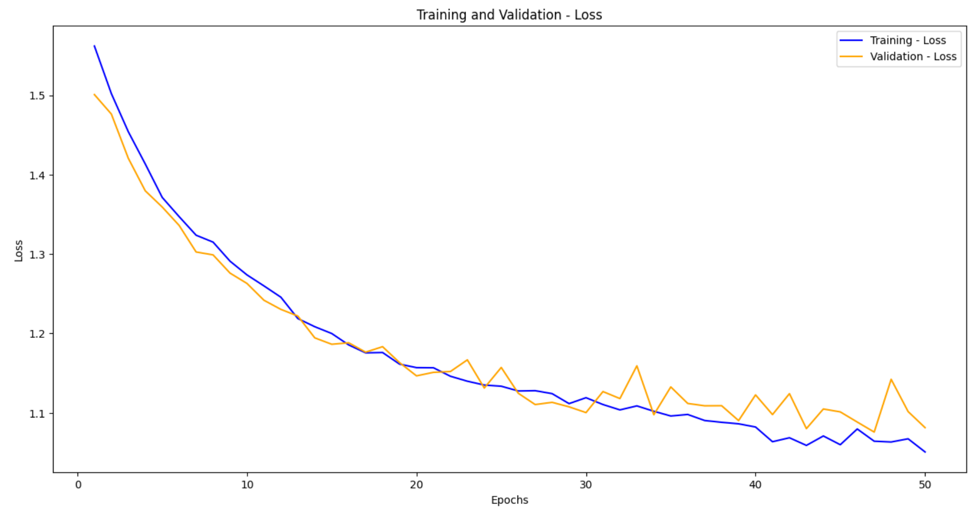

Figure 10: Training and validation loss.

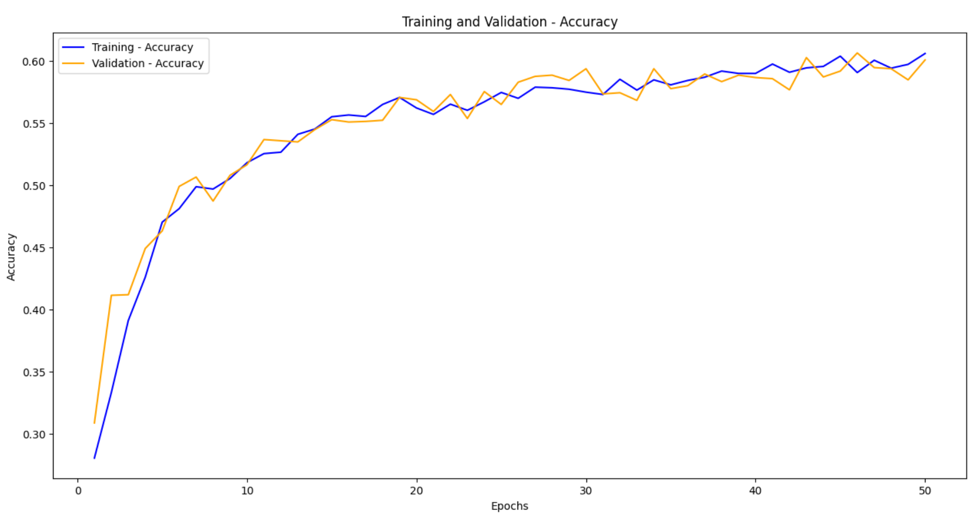

Figure 11: Training and validation accuracy.

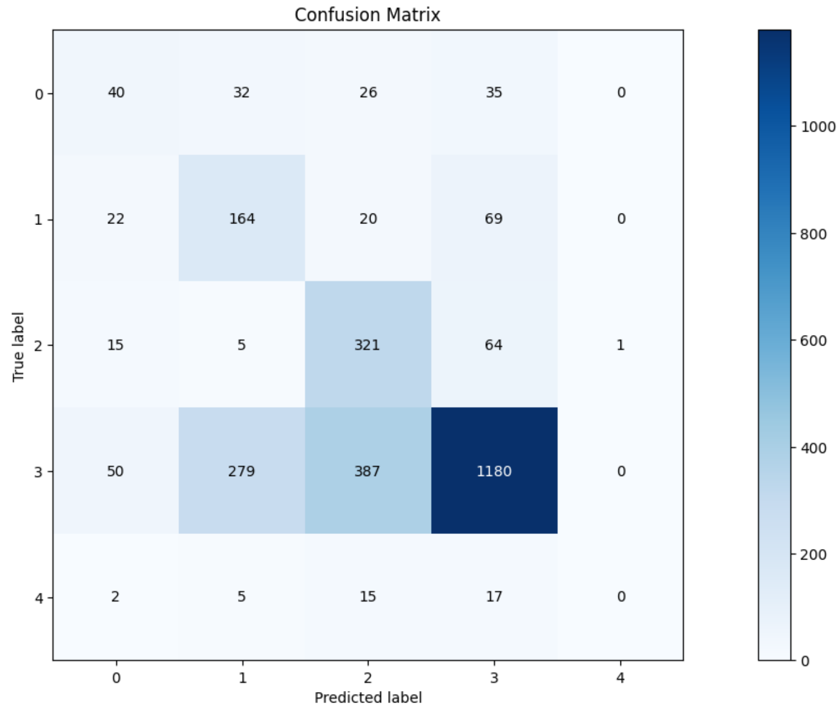

Figure 12: Test covariance matrix.

Test loss is **1.1157**.
Also in this last case, the performances are similar to the previous ones. Examining the test metrics, the results are still acceptable: an accuracy of 0.620 and a F1-Score weighted average of 0.638 with a CNN and limited and unbalanced training and test sets are not bad results. The charts show that the loss is consistent with the gradient descent and that the other metrics grow stably. Anyway, the loss tends not to improve significantly in the last epochs, which indicates that setting an earlier stopping may improve the results. The covariance matrix is consistent with what was just said, but looking at it, it is interesting to notice that this model performed best of all the others exclusively for class 2. Looking at the metrics tables, especially the validation ones, it can be noticed, compared to the previous models, that a training value not significantly high for the Recall of class 2 led to a significant improvement in the validation phase, thus improving performance. However, the lack of generalization power towards the class 4 samples led to an inability to make correct predictions about that class, and this significantly lowered the average performances values. So, once again, if the metrics are quite similar to the previous cases, Adam again conferred a better generalization power to the model.

4 Conclusion and future works

The results obtained show that, the models with Adam optimizer perform slightly better than SGD as they showed a greater generalization power especially for Recall in samples associated to class 4. Anyway, the most important thing to notice is that for a problem with a limited and unbalanced dataset, a shallow CNN can perform better than a deep one because its ”simpler” architecture will not excessively focus on the training data and will have a greater generalization power.
This task also demonstrated how using a non-optimal method and having limited and unbalanced training and test sets can significantly lower performance and make it difficult to define a good architecture for the model. In particular, compared to the previous homework, it was much more difficult to find the best architecture that I could identify as any change could worsen the performance or cause overfitting. It will be very interesting to solve this problem with Reinforcement Learning and compare the obtained performances.