Report on CNN/Scattering classification comparison

Alberto Carli Gabriele Roncolato Leonardo Zecchin

Contents

Chapter 1

Introduction

This document illustrates the project valid for the Visual Intelligence class of the academic year 2022/2023.
The assignment tests the knowledge gained in class by applying signal analysis methods, in particular we classified signals by using **Convolutional Neural Networks** and wavelet theory, specifically **Wavelet Scattering**.
In this project we implemented the code necessary to classify a given dataset first by training and testing a CNN, then by applying the Wavelet Scattering Transform and training a NN with the extracted features. Finally we compared the results obtained with these two methods in terms of accuracy against how many epochs were used to train the classifiers.
For the entire project, we followed the guidelines given during the laboratory lectures.

We started using an RGB dataset but we soon switched to the grayscale version because it is considered more challenging.
In this study, we initially used the Convolutional Neural Network (CNN) model provided during laboratory lectures as a baseline and compared its performance with a Neural Network (NN) model with wavelet scattering. We then sought to improve the results of both models by experimenting with different sets of parameters, with a particular focus on the NN and wavelet scattering parameters. These are the best parameters we found for the NN models:

- **Invariance scale**: $J = 6$
- **Order of scattering**: 2
- **Number of rotations**: $[8, 8]$
- **Quality factors**: $[2, 1]$
- **Learning rate**: 0.005
- **Momentum**: 0.9

After identifying the best parameters, we directed our attention to the CNN model.

Our findings revealed that the dataset was relatively small, and the loss and validation graphs suggested that the model was overfitting. To address this issue, we restructured the CNN model by introducing BatchNorm and Dropout layers. Results were unsatisfactory, suggesting that these changes alone were insufficient. To address the overfitting issue, we incorporated data augmentation into the training set, which significantly improved the results.
Our attempts to augment the dataset using translations and rotations were initially promising, but we found that the CNN filters were learning the black band generated by the translations. To avoid this, we opted to stick to rotations between -45 and 45 degrees.
This is the final set of parameters that we found to be optimal for this dataset:

- **batch_size**: 64
- **test_perc**: 0.2
- **num_samples**: 500
- **epoch_val**: 1
- **num_k_folds**: 3
- **augmentations**: 16
- **weight_decay**: 0.01
- **optimizer**: 0
- **learning_rate**: 0.005
- **momentum**: 0.9
- **num_epochs**: 200

Chapter 2

Objectives

The goal of this project is to explore the use of scattering transforms to improve the performance of neural networks on a given dataset.
Specifically, we aimed to compare the performance of a CNN trained on the original dataset to a NN trained on the scattering decomposition of the data.

We then visualized the filters learned by the CNN and compared them to the ones applied by the scattering transform to gain insight into the types of features that were extracted. By accomplishing these objectives we hoped to gain a better understanding of how the CNN learns which features to extract.

Chapter 3

Dataset

We used a dataset consisting of 128x128 RGB images divided into two categories: dogs and flowers. There are 1600 pictures of dogs and 1387 pictures of flowers. We converted everything to grayscale with each class n its specific folder and every file numbered.

Figure 3.1: Samples from the RGB dataset

Chapter 4

Framework

4.1 Convolutional Neural Network

4.1.1 What is a CNN

A Convolutional Neural Network (**CNN**) is a type of deep learning algorithm commonly used for image recognition and computer vision tasks. It is designed to automatically learn and extract relevant features from input images through convolutional and pooling layers, followed by fully connected layers that produce output predictions.
The `CNN_128x128` architecture consists of four **convolutional** layers and three fully connected layers. The first layer takes an input image with `input_channel` number of channels, and the output of the last layer is a vector with `num_classes` elements representing the probability of each class.

4.1.2 Architecture

```
CNN_128x128(
    (conv1): Conv2d(1, 16, kernel_size=(7, 7), stride=(2, 2), padding=(1, 1))
    (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(2, 2), padding=(1, 1))
    (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (batchnorm1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_ru
    (batchnorm2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_ru
    (batchnorm3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (drop1): Dropout(p=0.2, inplace=False)
    (flat): Flatten(start_dim=1, end_dim=-1)
    (fc1): Linear(in_features=256, out_features=64, bias=True)
    (drop2): Dropout(p=0.5, inplace=False)
    (fc2): Linear(in_features=64, out_features=2, bias=True)
    )
```

The model is named `CNN_128x128` and takes as input an image with a width and

height of 128 pixels and a number of channels specified by the `input_channel` parameter (typically 3 for RGB images or 1 for grayscale images). The number of output classes is specified by the `num_classes` parameter.
The CNN consists of three **convolutional layers** with increasing numbers of output channels (16, 32, and 64, respectively). Each convolutional layer is followed by a **batch normalization layer**, a **ReLU** activation function, and a **max pooling layer** with a kernel size of 2 and a stride of 2.
The output of the final convolutional layer is flattened into a vector and passed through two **fully connected** (FC) layers with 64 and `num_classes` output units, respectively. The FC layers are followed by dropout layers to prevent overfitting.

During the forward pass, input images are passed through the convolutional layers, normalized and pooled before being flattened and passed through the FC layers. The output of the last FC layer is a probability distribution over the `num_classes` output classes, which can be used to make predictions about the input image's class.

4.2 Neural Network

4.2.1 Wavelet scattering

What is a wavelet scattering

Wavelet scattering is a mathematical technique for analyzing signals, such as sound or images, by decomposing them into different frequency bands and time scales. It works by applying a series of wavelet transforms to the signal, which effectively "filters" the signal into its constituent frequency components.
The key idea behind wavelet scattering is to create a representation of the signal that is invariant to translations and rotations. This allows us to extract features from the signal that are robust to changes in scale and position.

How we do it

Initially, we tried to use the Python library **Kymatio** to implement **2D wavelet scattering**, but then we decided to implement it through **MATLAB** because it allows us to specify scattering **parameters** better. We connected MATLAB to **Python** using the **matlab.engine** library. This is implemented in the `scatter_helper.py` file.

The purpose of the `get_scatterNet.m` file is to create a **scattering network** MATLAB object for input images using **Wavelet Scattering transform**.
The function calls the `waveletScattering2` MATLAB function, which is part of the Deep Learning Toolbox. WaveletScattering2 computes the scattering coefficients for a given image using a wavelet transform, which is then used to

construct the scattering network.
The input arguments of the `get_scatterNet` function are used to configure the waveletScattering2 function. Specifically, `invariance_scale` specifies the scale of invariance of the scattering coefficients, `quality_factors` specifies the quality factors for the wavelet transform, `num_rotations` specifies the number of rotations used in the wavelet transform, and `images_size` specifies the size of the input image.
Finally, the function returns the computed scattering network as **sn**.

Moreover, we have incorporated the `scattering.m` file that calculates the `scatter` function. This function applies the **Scatter Transform** to all images passed as an input argument `images`, using the Matlab method `scatteringTransform`. The `scatteringTransform` function computes the scattering transform for the input data, utilizing the provided scattering network (`sn`).

Finally, we reshaped the extracted features to keep only the highest-order scattering coefficients, after finding that our initial attempt to take the mean of the scattering coefficients over the spatial dimensions did not yield interesting results. We also found that keeping all the coefficients gave even worse results.

4.2.2 Architecture

```
NN_128x128(
  (flat): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=40960, out_features=64, bias=True)
  (drop2): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=64, out_features=2, bias=True)
)
```

This is a neural network with **fully-connected layers** designed for image classification tasks. Unlike Convolutional Neural Networks (CNNs), which are commonly used for image classification, this network does not have any convolutional layers.
The input to the network is an image with `input_channel` number of channels and size 128x128 pixels. The image is flattened into a vector using the`nn.Flatten()` layer, and then passed through two **fully-connected layers**. The first fully-connected layer has 64 nodes, and the second (output) layer has `num_classes` units, which correspond to the number of classes in the classification task.

The **forward method** defines the forward pass of the network, which simply applies the two fully-connected layers to the input vector, with ReLU activation applied to the output of the first layer and a dropout layer applied after the ReLU activation.

4.3 Pipeline

Which model to run can be specified when calling `configurable_classification.classify()` and passing `True` to the parameters *cnn* and *nn* respectively.

Each model is handled separately, each execution with its own quirks, as specified in the architecture section. Generally the pipeline involves:

1. dataset preparation
2. train-test splitting
3. eventual augmentation
4. train-validation splitting (eventually through k-fold)
5. training
6. validation
7. testing

In many points of the pipeline, behavior is controlled through parameters, which are written in the file *parameters.yaml*.
These parameters are overwritten at every execution by running the script *lib/scripts/make_settings.py*.
Scatter-specific parameters are too, written by the same script, but to another file named *scatter_parameters.yaml*.

Execution results are produced as graphs of training and testing and saved in the folder *results* folder, each execution in its specific folder, numbered incrementally with the runs. The dump of the whole parameters file is saved as well. The output is a figure with the training and validation loss and accuracy, for every fold, a figure with the confusion matrices, side by side and a figure with the filters developed by the CNN, if present. If the parameter *display* is set to `True`, then graphs are displayed after execution too.

4.3.1 Dataset preparation

We created a class that handles the dataset `data_handler.py`. This class provides functionality to load the dataset from the disk, subsample it, and split it into training, validation, and test sets, wrap it into a batcher class from pytorch, transfer it to the GPU, and perform data augmentation.
Every random operation is seeded with 42 to ensure reproducibility.

When the data is loaded from disk, every class folder is read with the `cv2.imread()` function and a list of labels is generated assigning a number to the class, incrementally.

Then if the parameter `samples` is set (e.g. not None), the dataset is subsampled by randomly selecting an equal amount of images from each class up to the `samples` value specified.
The dataset and labels are saved as class attributes and returned as `pytorch.Tensor()`.

The dataset is then split into training, validation, and test sets using the `get_data_split()` function, which is just a wrapper for the `train_test_split()` function from `sklearn.model_selection` to better handle the data and seed it with a constant seed (42).
The `test_perc` parameter controls the percentage of split between train and test sets by specifying how much.

Then, if the folds parameter is set to more than 1, the training set is split and iterated through `folds` folds using the `sklearn.model_selection.KFold()`.
Otherwise, the training set is split into training and validation sets using the `train_test_split()` function from `sklearn.model_selection` with 0.2 of the dataset going to validation.
The validation set is used to monitor the training process and to detect overfitting.

4.3.2 Augmentation

The `data_handler` class provides a method to perform data augmentation on the dataset. Again, this is just a wrapper function for some `torchvision.transforms` all linked to a policy selected in the code. Aternative (or custom) policies can be found in the file `lib/scripts/custom_augment.py`.

The augmentation is performed by randomly applying a series of transformations to the images in the dataset. Our policy consists in random rotation of angles between 0 and 90 degrees.

In our pipeline, the transformations are applied to the training set only, before being split into training and validation sets, while the testing set is not augmented.
The original image is kept and then a number of new augmented images is generated as specified in the parameter `augmentations`.

4.3.3 Training

The training procedure is shared between the two models.
The training is performed by the `train()` function in the `lib/train_test.py` file.

Two optimizers can be selected with the `optimizer` parameter: `0` = `SGD` or `1` = `Adam` to which are given parameters as required. SGD uses *learning_rate* and *momentum*, while Adam uses only *learning_rate*.
Cross Entropy is used as loss function.

The training is performed by iterating through the training set `epochs` times. The function expects the dataset as a `torch.utils.data.DataLoader` and iterates by batch alternating classification, loss calculation and backpropagation.

If a GPU is available (and detected by pytorch), each batch will be loaded onto GPU and training will happen there.
The training is monitored by printing the loss and accuracy on the training set every epoch.
Training accuracy and loss are stored into lists and then returned after the training is done, to be used for plotting.

4.3.4 Validation

Validation can be enabled by setting the `epoch_val` parameter to a positive number, or disabled entirely by giving it value `None`. If enabled, the validation is performed every `epoch_val` epochs.

The validation is performed by setting the model to evaluation mode and running the model while iterating through the validation set.
A loss is calculated, but no backpropagation is performed.
If a GPU is available (and detected by pytorch), each batch will be loaded onto GPU and training will happen there.

The validation is monitored by printing the loss and accuracy on the validation set every time it is run.
Validation accuracy and loss are stored into lists and then returned after the training is done, to be used for plotting.

4.3.5 Testing

Testing is performed by the `test()` function in the `lib/train_test.py` file.
The function expects the dataset as a `torch.utils.data.DataLoader` and iterates by batch, classifying each image and storing the results.
Basically like validation, but with testing data. In case of data augmentation, testing data is NOT augmented.

If a GPU is available (and detected by pytorch), each batch will be loaded onto GPU and testing will happen there.

Testing predictions and real results are stored and returned after testing is done, to enable more flexible analysis and metrics calculation.
In our case, the `metrics` class handles calculation of statistics and plotting of useful graphs.

Chapter 5

Results

The project started from the RGB dataset images and with the models as seen during laboratory lectures.
We started by using the Convolutional Neural Network (CNN) model provided during laboratory lectures from which we extracted the fully connected section and thus defined the **NN_128x128** model.
These models were using RGB images, but after a short while we converted everything to grayscale images.

We started working with the parameters as given in the laboratory lectures, together with the models.

5.1 Tuning scatter

In this first part of the study, we focused on the NN model with wavelet scattering.
We started by tuning the parameters of the scattering transform, which are the invariance scale J, the order of scattering Q, the number of rotations R, and the quality factors Q.

But using kymatio, we found that the library does not support the quality factors, so, after a bit of fruitless parameter tuning, we opted to use the `scattering2d` function from `kymatio.scattering2d` instead, for better control.

5.2 Models tuning

Then we tried to tune the parameters of the optimizer of the models.
We tried different learning rates and momentum values for the SGD optimizer, and finally found a combination that led to the networks actually learning.

- **Learning rate**: 0.01
- **Momentum**: 0.5

So, in this situation we could actually try and find some good values for the MATLAB scatter transform:

- **Invariance scale**: $J = 6$
- **Order of scattering**: 2
- **Number of rotations**: $[8, 8]$

We found that with these values, the neural network performed well at around 80% accuracy.

5.3 CNN

With the scatter sorted, we now focused on the CNN model, ready to find new, different, parameters that would better suit the different model.
We figured that the model was overfitting a lot on training data so, to mitigate that, we tried to change the structure of the model, adding Batch Normalization and Dropout layers, as well as adding/reducing layers size and number.
In the meantime, through testing, we figured that the best parameters for the SGD optimizer were the following:

- **Learning rate**: 0.005
- **Momentum**: 0.9

At this point we realized that batch normalization and dropout layers were contributing to stability of the training, so we decided to keep them.

Even though the accuracy was good and we solved the overfitting issue, the filters were not quite what we expected (something like edge detection), so we tried to change parameters of the model, but we observed some sort of tradeoff between the training curves (loss and accuracy) and the quality of the filters.
We also tried to use the Adam optimizer, but the results were not as good as with SGD and could not determine consistent differences between the two.

5.4 Augmentation

With augmentation we finally managed to get both reasonable training curves and good filters.
We tried to use the `autoaugment` function from `torchvision.transforms`, and

it gave very nice results.

But we didn't yet understand why, so we tried to use only a single type of transformation at a time and try to understand what was happening.
Notably, when augmentation involved only translations, the filters presented some sort of weird black edge around, so we figured they didn't help much.

Finally, we tried to use only rotations, and the results were very good: we could see both good edge filters and decent training curves.
Moreover, model's accuracy was very good, at about 85%.
We also noticed some sort of positive correlation between the number of augmented (rotated) images and the accuracy of the model.

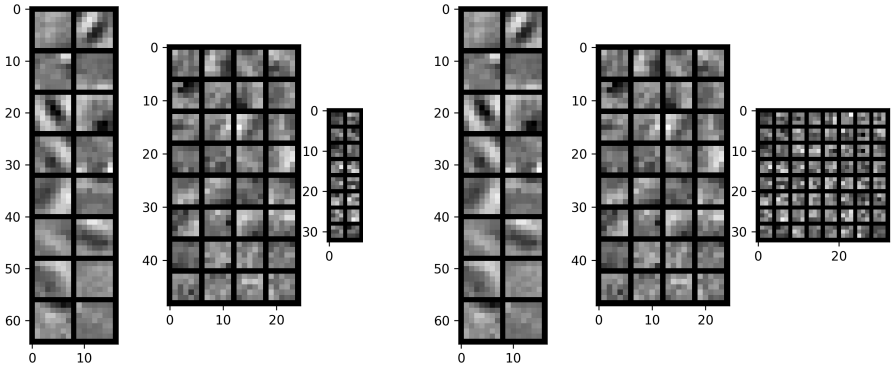
Chapter 6

Conclusions

Our study has revealed several key findings regarding the impact of parameter selection and data augmentation techniques on the performance of Convolutional Neural Networks (CNNs) and Neural Networks (NNs). The optimal parameters varied between the two models, but our results suggest that rotations were the most effective parameter for improving performance. However, the reasons for this improvement are not entirely clear, as it is uncertain whether it is due to the rotations themselves or the increase in the number of training examples. While using more samples can lead to a training process that is noisier and with "faded" filters, it can also result in better generalization. Indeed, data augmentation techniques such as rotations can produce filters that are more generalizable to unseen data, as the model learns to identify more robust features that are less dependent on specific input configurations. Nonetheless, it is crucial to find the right balance between the number and type of augmented samples and the complexity of the model to avoid overfitting. Overall, our findings highlight the importance of careful parameter selection and the potential benefits of using rotation as a data augmentation technique for improving the performance and generalization of CNNs and NNs.

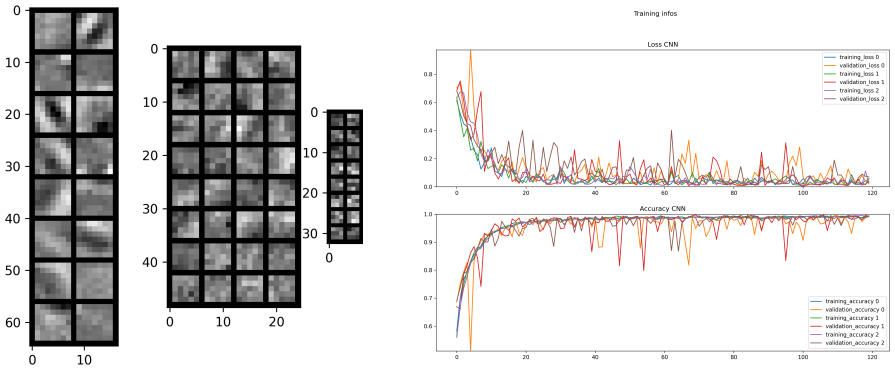
Part I

Appendix



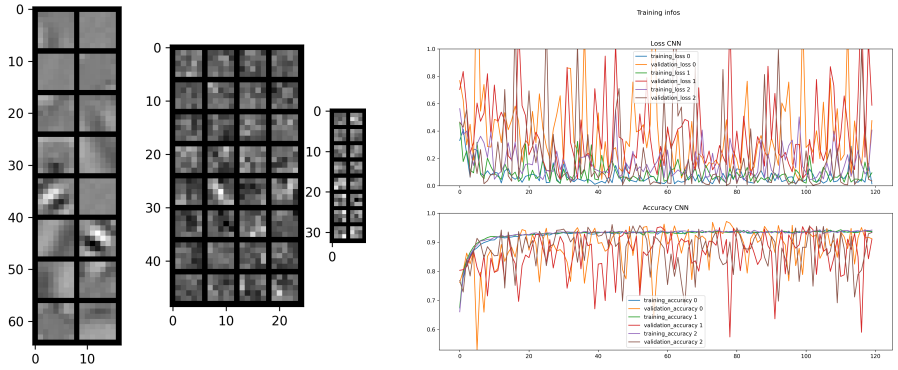
(a) Caption for Image 1 (b) Caption for Image 2

Figure 1: Caption for the whole figure (TEMPLATE)



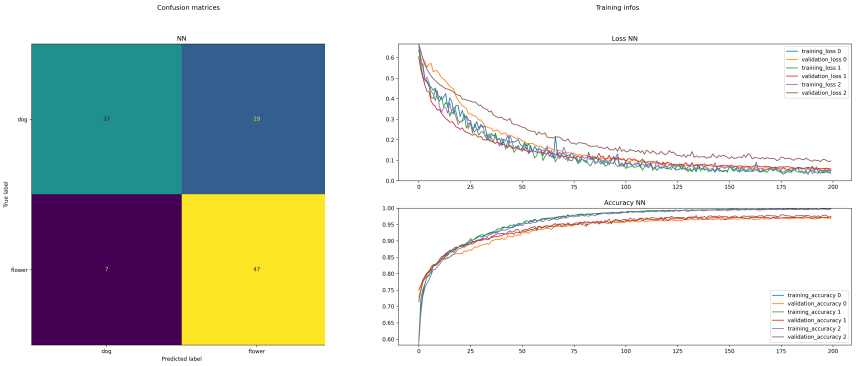
(a) CNN filters (b) Loss and Validation graphs

Figure 2: These are the results obtained with the best parameters found for the CNN model. The **accuracy** reached in test is 0.82.



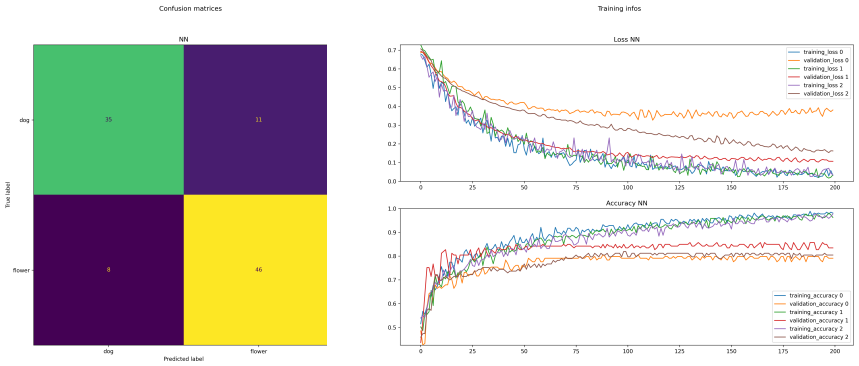
(a) CNN filters (b) Loss and Validation graphs

Figure 3: Then we tried with 2500 samples. The **accuracy** reached in test is 0.87



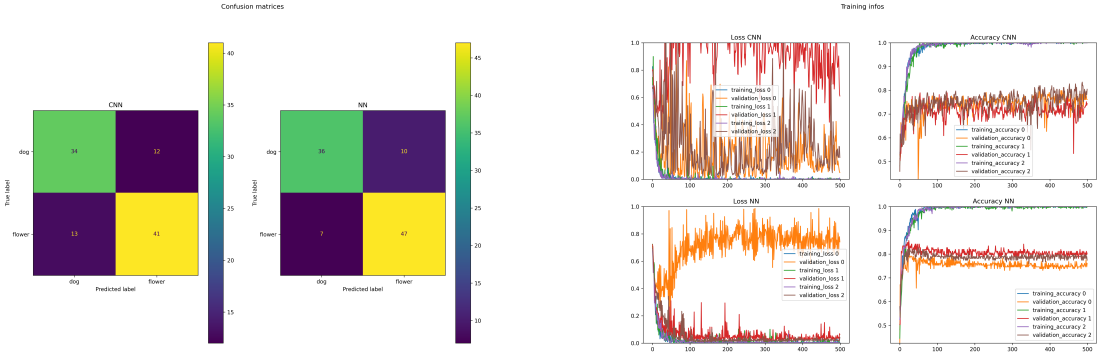
(a) Confusion matrix (b) Loss and Validation graphs

Figure 4: NN executions with agumentation. The **accuracy** reached in test is 0.74.



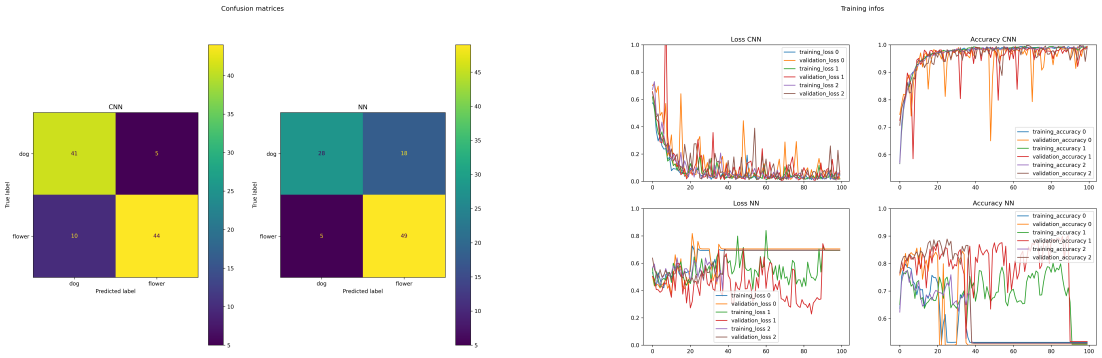
(a) Confusion matrix (b) Loss and Validation graphs

Figure 5: NN executions without agumentation. The **accuracy** reached in test is 0.81.



(a) Confusion matrix comparison (b) Traning graphs comparison

Figure 6: Execution with 500 samples and no augmentation. The NN reaches a better accuracy with 83%, while the CNN reaches 75%. No augmentation privileges scatter+NN



(a) Confusion matrix comparison

(b) Traning graphs comparison

Figure 7: Execution with 500 samples and 16 augmentations. The CNN reaches a better accuracy with TODO%, while the NN reaches TODO%. Augmentation privileges CNN