Report on CNN/Scattering classification comparison

Alberto Carli Gabriele Roncolato Leonardo Zecchin

Contents

Chapter 1

Introduction

This report illustrates the results of the project for the Visual Intelligence class of 2022/2023.
The assignment is designed to test the knowledge gained in class regarding signal analysis and wavelet theory: in particular we classified signals using **Convolutional Neural Networks** and wavelet functions, specifically the **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 classifier NN with the extracted features. Finally we compared the results obtained with these two different 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 classification models 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 transform of the data.

We then visualized the filters built by the CNN and compared them to the ones built by the scattering transform to gain some insight into what types of features each model extracted. By accomplishing these goals we have gained a better understanding of how a CNN can iteratively learn which features are more meaningful.

Chapter 3

Dataset

We used a dataset consisting of 128×128 RGB images split into two categories: dogs and flowers. There are 1600 pictures of dogs and 1387 pictures of flowers. In order to make the task less trivial we converted all samples to grayscale.

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 class of neural networks commonly used for visual analysis. It is designed to automatically build specialized convolutional filters and extract relevant features from input images through convolution and pooling: the extracted features can then be used to compute an output through a series of fully connected layers.
The `CNN_128×128` architecture consists of four **convolutional** layers and three fully connected layers. The networks takes an image with `input_channel` channels as input and returns a vector with `num_classes` elements representing the probability of each class as output.

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_128×128`: it takes a 128×128 image with a number of

channels specified by the `input_channel` parameter as input (typically 3 channels for RGB images or 1 channel for grayscale images). The number of output classes is specified by the `num_classes` parameter.
The CNN consists of three **convolutional layers** with an increasing number 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 and passed through two **fully connected** (FC) layers with 64 and `num_classes` output units, respectively. A dropout layer is present to prevent overfitting.

During the forwarding, each input image is convoluted, normalized and pooled before being flattened and forwarded to 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 class of the input image.

4.2 Neural Network

4.2.1 Wavelet scattering

What is a wavelet scattering

The wavelet scattering can be used to analyze signals, such sounds or images, by decomposing them into different frequency bands and time/space scales. It applies a series of wavelet transforms to the signal and breaks it down 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 resistant to changes in scale and position.

How we do it

Initially, we tried using the Python library **Kymatio** to apply a **2D wavelet scattering**, but some limitations in the Kymatio implementation led us to implement our own transform in **MATLAB**, mainly because the Kymatio library didn't allow us to specify some critical **parameters**. We were then able to execute MATLAB code from **Python** using the **matlab.engine** library. This implementation can be found in the `scatter_helper.py` file.

The purpose of the code contained in `get_scatterNet.m` is to create a **scattering network** MATLAB object for input images using **Wavelet Scattering transform**.
This function calls the `waveletScattering2` MATLAB function from 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**.

In addition to that, we have incorporated the `scattering.m` file which calculates the `scatter` function. This function applies the **Scatter Transform** to all of the images which are passed as the input argument `images` using the Matlab function `scatteringTransform`.
The `scatteringTransform` function computes the scattering transform of the input images using the scattering network (`sn`).

Finally, we reshaped the extracted features in order 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 of the coefficients gave 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. Unlike Convolutional Neural Networks (CNNs), which are commonly used for image classification, this network does not have any convolutional layer. The input of this network is an image with `input_channel` channels and a size of 128×128 pixels. The image is flattened using the`nn.Flatten()` layer and then passed through two **fully-connected layers**. The first fully-connected layer has 64 nodes and the second layer (which is also the output layer) has `num_classes` units, which correspond to the total number of classes for the classification task.

The **forward method** defines the forwarding of the network: the input vector passes through the first FC layer, a ReLU activation function and finally a dropout layer.

4.3 Pipeline

It is possible to specify which model to employ by calling `configurable_classification.classify()` with the value `True` for the *cnn* and *nn* parameters.

Each model is handled separately as specified in the architecture section. The global pipeline involves:

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

The behavior of the pipeline is controlled through the parameters specified in the *parameters.yaml* file.
These parameters are overwritten at every execution by running the *lib/scripts/make_settings.py* script.
Scatter-specific parameters are also written by the same script to a different file named *scatter_parameters.yaml*.

The results for each execution consist of training graphs and testing measures, which are saved in the *results* folder: each set of results is saved in its own specific folder, which is incrementally numbered at each execution. A dump containing all of the parameters used is also saved in a txt file.
The output consists of a graph showing the training and validation loss and accuracy for every fold, a figure containing the confusion matrices side by side and a figure showing the filters built by the CNN. If the *display* parameter is set to `True`, then these results are displayed before the execution has terminated.

4.3.1 Dataset preparation

We created a class which handles the dataset `data_handler.py`: this class provides a function to load the dataset from the disk, subsample it, split it into separate training, validation, and test sets, wrap it into a batcher class from pytorch, transfer it to the GPU and perform data augmentation on it.
Each random operation is seeded with 42 to ensure consistent results.

When the data is loaded from the disk, every class folder is read using the `cv2.imread()` function and a list of labels is then generated by assigning a

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

Then, if the folds parameter is set to greater than 1, the training set is split and iterated through `folds` folds using the `sklearn.model_selection.KFold()` function.
Otherwise, the training set is split into a training set and a validation set using the `train_test_split()` function from `sklearn.model_selection` with 20% of the dataset reserved for 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. This is just a wrapper function for some `torchvision.transforms` linked to an augmentation policy. 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 dataset images. Our policy consists of random rotations between -45 and +45 degrees.

The transformations are only applied to the training set before it is split into training and validation, while the testing set is not augmented.
The newly generated augmented images are added to the dataset: the amount of generated images is specified by the `augmentations` parameter.

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.

One out of two different optimizers can be selected using the `optimizer` parameter: `0` = `SGD` or `1` = `Adam` 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 training function expects a `torch.utils.data.DataLoader` object as training set and iterates on each batch alternating classification, loss calculation and backpropagation.

If a GPU is available (and detected by pytorch) each batch will be loaded onto the GPU and the training will take place there.
The training is monitored by printing the loss and accuracy on the training set every epoch.
The 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 the `None` value. If enabled, the validation is performed once every `epoch_val` epochs.

The validation is performed by setting the model to evaluation mode and iterating the model 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 the GPU and the validation will take place there.

The validation is monitored by printing the loss and the accuracy on the validation set each time it is executed.
The 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.
This function expects the dataset as a `torch.utils.data.DataLoader` object and it iterates on each batch, classifying each image and storing the results in the same way as the validation but using the test set. In case of data augmentation, the testing set is NOT augmented.

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

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

Chapter 5

Results

Our initial dataset consisted of the RGB images and our initial models were derived from CNN defined during the laboratory lectures.
Starting off from this Convolutional Neural Network we extracted the fully connected layers and used those to define the **NN_128×128** model.
Eventually we converted our initial dataset to grayscale images in order to make the classification task less trivial.

5.1 Tuning scatter

At first we focused on the NN model and on the 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.

As mentioned above, we initially tried using kymatio to apply the scattering transform but we found that the library does not support a quality factors parameter for the 2D transform: for this reason, after some fruitless parameter tuning, we opted to use the `scattering2d` function from `kymatio.scattering2d` which granted us more control.

5.2 Models tuning

We then tried tuning the parameters of the optimizer.
We tried using different learning rates and momentum for the SGD optimizer, and finally found the optimal values which allowed the models to start learning.

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

Then we finally found some good values for the MATLAB scatter transform parameters:

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

Using these values the neural network reached TODO accuracy in testing.

5.3 CNN

After finding the optimal scatter parameters we finally focused on the CNN model.
A different architecture meant, of course, the need for different parameters which could work well with the additional convolutional layers, but the model would always overfit or underfit. After many attempts at finding an optimal configuration we concluded that the CNN model needed to be further adjusted: the addition of Batch Normalization and Dropout layers, as well as increasing or reducing the number of layers and their size, was critical in avoiding overfitting. We then concluded that the best parameters for the SGD optimizer were the following:

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

We also attempted to use the Adam optimizer, but SGD consistently provided better results.

The model was now reaching a good accuracy during testing, but the filters were not quite what we expected: instead of obtaining a set of filters which resembled some generic edge detection filters (like the ones used by the scatter transform), we were obtaining some very noisy matrices. We also started noticing a consistent tradeoff between the quality of training/accuracy and the quality of the filters. Eventually we came to realize that, when using a small dataset, a convolutional network can either build very generic filters, which perform poorly on a very specific and biased dataset, or very specific filters, which greatly diverge from common edge detection ones but perform well on the given dataset.

5.4 Augmentation

The results showed that the only way to obtain both a good accuracy and the expected filters is to increase the training dataset. This was not possible however, as one of the project goals was to use a small dataset: for this reason

we decided to artificially increase the amount of data using data augmentation. This allowed us to reach both reasonable training curves and the expected filters.

The `autoaugment` function from `torchvision.transforms` with the IMAGENET augmentation policy gave very good results.
In order to understand why this was the case, we tried using only a single type of transformation at a time (rotations, solarization, translations, ...).
Notably, when augmentation involved only translations, the filters presented some black edges and the accuracy was not as good: we concluded that the wrong type of augmentations didn't allow the model to learn any further.

Finally we tried using only rotations: we could see both good edge filters and decent training curves.
Moreover, the model's accuracy also increased.
We noticed a positive correlation between the number of augmented images and the quality of the filters.

5.5 Summary

We now present a brief summary of the parameters used to train our models.

The following are the best parameters we found for the NN model:

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

The following are the best parameters we found for the CNN model:

- **batch_size**: 64
- **test_perc**: 0.2
- **num_samples**: 500
- **epoch_val**: 1
- **num_k_folds**: 3
- **augmentations**: 16 (number of augmented images per original image)
- **weight_decay**: 0.01

- **optimizer**: 0 (0- SGD, 1- Adam)
- **learning_rate**: 0.005 (scale for how much new weights are evaluated)
- **momentum**: 0.9 (scale for past experience to not be perturbed by new ones)
- **num_epochs**: 200

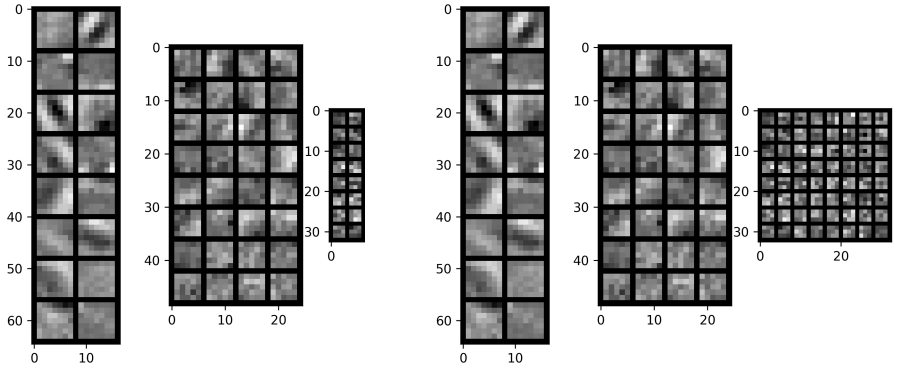
Chapter 6

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 on our particular dataset. The addition of augmented images to the dataset not only increased the overall accuracy but also allowed the CNN model to build a set of filters which resemble generic edge detection filters, in particular those used by the scatter transform. This led us to conclude that data augmentation not only increases the capability for generalization in the fully connected layers, but also in the filters used for the convolutional layers. Data augmentation techniques such as rotations can indeed produce filters that are more generalizable to new data, as the model learns to identify more robust features that are less dependent on a specific input. The robustness to certain properties strictly depends on the type of augmentation used: for example, the use of rotations led the CNN to build filters which could work well with rotated subjects, while the use of translations led the CNN to build filters which could work with black edges (although not optimal for this classification task). 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 parameters selection and the potential benefits of using rotations as a major data augmentation technique for improving the performance and generalization of CNNs and NNs.

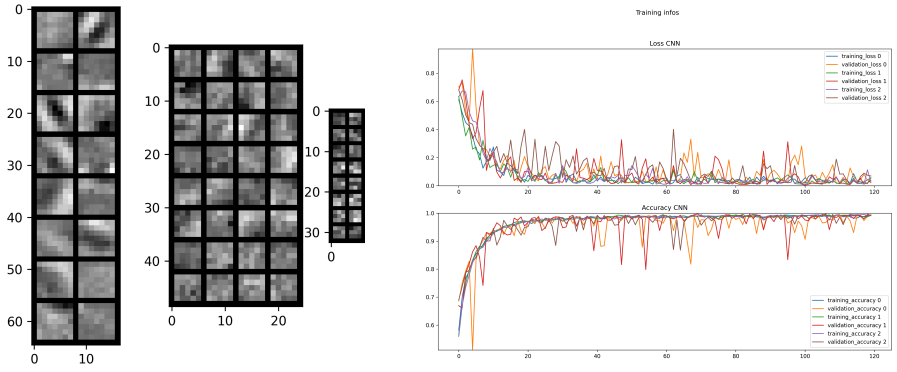
Part I

Appendix



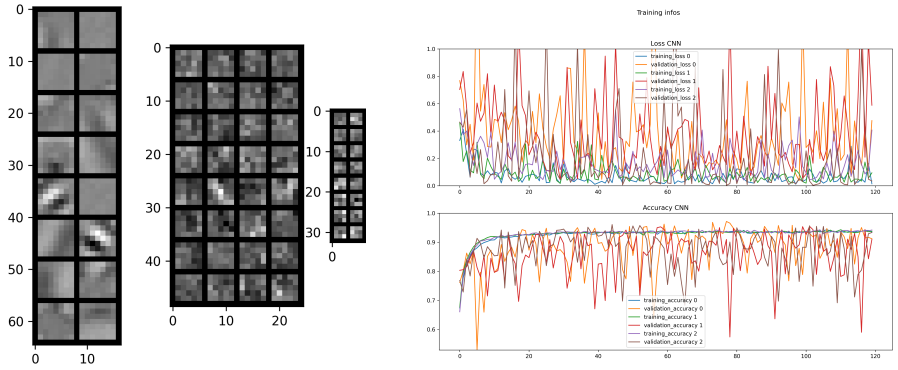
(a) Caption for Image 1 (b) Caption for Image 2

Figure 1: Caption for the whole figure (TEMPLATE)



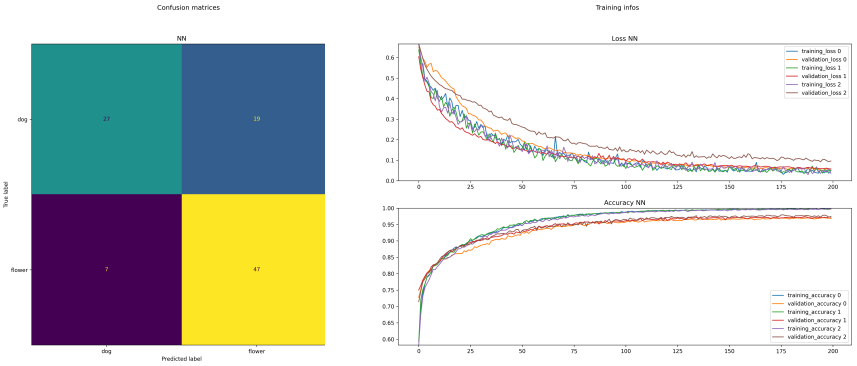
(a) CNN filters (b) Loss and Validation graphs

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



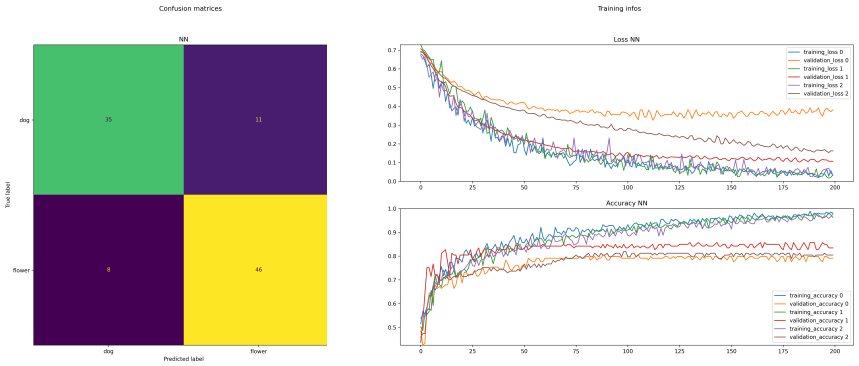
(a) CNN filters (b) Loss and Validation graphs

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



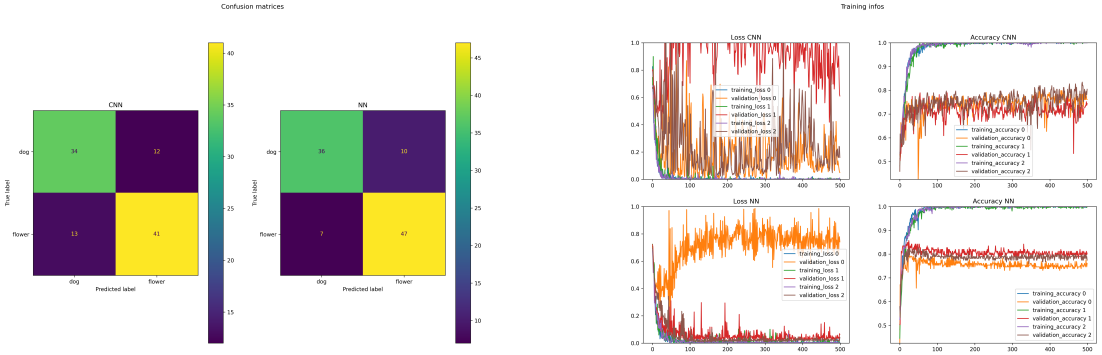
(a) Confusion matrix (b) Loss and Validation graphs

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



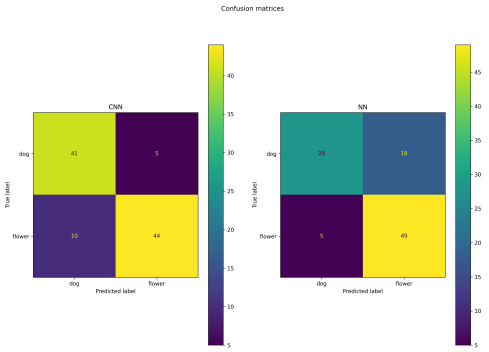
(a) Confusion matrix (b) Loss and Validation graphs

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

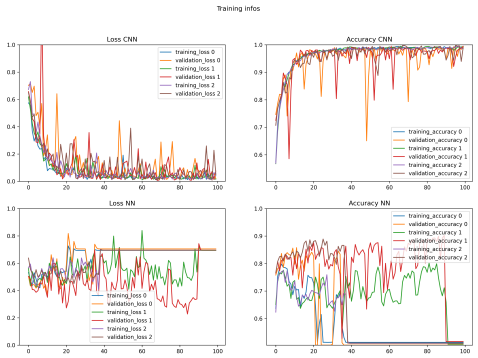


(a) Confusion matrix comparison (b) Traning graphs comparison

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