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.

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.

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)

How we do it

Spiegare come viene applicata la wavelet scattering (utilizzato matlab). We found the best results with the following values:

- `J` = ...: parameter represents the number of scales to use in the wavelet transform .

- `order` = ...: is the maximum order of wavelet scattering coefficients to compute .

- `L` = ...: specifies the number of rotations to use in the wavelet transform .

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 hidden units, 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 Dataset preparation

We create 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 `train_test_split()` function from `sklearn.model_selection`.
A parameter controls the percentage of split between training and test sets.

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 going to validation.
The validation set is used to monitor the training process and to prevent overfitting.

4.4 Augmentation

The `data_handler` class provides a method to perform data augmentation on the dataset.
The augmentation is performed by randomly applying a series of transformations to the images in the dataset. Our transformations can be found in the file `lib/scripts/custom_augment.py` and consist in random rotation of angles between 0 and 90 degrees.

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

4.5 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 appropriate parameter `optimizer`: `0 = SGD` or `1 = Adam` to which are given parameters as required. SGD uses *learning_rate* and *momentum*, while Adam uses only *learning_rate*.
The training is performed by iterating through the training set `epochs` times.

4.6 Validation

4.7 Testing

Chapter 5

Results

cominciato in RGB, ma passato a grayscale perchè la prof è cattiva

5.1

trovati parametri ottimali per la scatter ma la cnn non partiva: i dati erano pochi provato a modificare la struttura della cnn (overfitting)(batchnorm, dropout, ecc) alla fine siamo riusciti ad avere buoni risultati con l'Augmentation provato autoaugment, solo traslazioni e solo rotazioni

Chapter 6

Conclusions

I parametri vanno meglio per la CNN o per la NN, richiedono parametri diversi
le rotazioni sono quelle che hanno dato risultati migliori + campioni -¿ training
noisy e filtri "sbiaditi", ma migliore generalizzazione