UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Computational Intelligence and Deep Learning

Artist Identification with Convolutional Neural Networks

Project Documentation

TEAM MEMBERS:
Edoardo Fazzari
Mirco Ramo

Academic Year: 2020/2021

Contents

1 — Introduction

Artist identification is traditionally performed by *art historians* and *curators* who have expertise and familiarity with different artists and styles of art. This is a complex and interesting problem for computers because identifying an artist does not just require object or face detection; artists can paint a wide variety of objects and scenes. Additionally, many artists from the same time period will have similar styles, and some such as **Pablo Picasso** (see figure **??**) have painted in multiple styles and changed their style over time.

Figure 1: Both of these paintings were created by Pablo Picasso, but they have vastly different styles and content since they correspond to two different periods.

The aim of this project is to use Convolutional Neural Networks for the identification of an artist given a painting. In particular, the CNN networks will be modeled using multiple techniques: from scratch; via pretrained network; networks based on comparison with baseline input and using an ensemble network made of by the best classifiers found.

1.1 State of the Art

As mentioned, artist identification has primarily been tackled by humans. An example of that is the Artsy's Art Genome Project [1], which is led by experts who manually classify art. This strategy is not very scalable even if it is highly precise in the classification (the site is a marketplace of fine-arts for collects, you can find Pisarro, Bansky and other famous artists).

Most prior attempts to apply machine learning to this problem have been feature-based, aiming to identify what qualities most effectively distinguish artists and styles. Many generic image features have been used, including scale-invariant feature transforms (SIFT), histograms of oriented gradients (HOG), and more, but with the focus on *discriminating different style* in Fine-Art Painting[2].

The first time the problem of artist identification was really tackled was with J. Jou and S. Agrawal[3] in 2011, they applied several multi-class classification techniques like Naïve Bayes, Linear Discriminant Analysis, Logistic Regression, K-Means and SVMs and achieve a maximum classification accuracy of 65% for an unknown painting across 5 artists. Later on, the problem of identifying artists was retackled by the *Rijksmuseum Challenge*[4]. The objective of the challenge was to predict the artist, type, material and creation year (each of them was a different challenge) of the 112,039 photographic [5] (containing different viewpoints of an artwork, and different types of them: sculptures, paintings, saucers, etc.) reproductions of the artworks exhibited

[1] https://www.artsy.net/categories

[2] T. E. Lombardi. The classification of style in fine-art paint- ing. ETD Collection for Pace University, 2005

[3] J. Jou and S. Agrawal. Artist identification for renaissance paintings.

[4] T. Mensink and J. van Gemert. The rijksmuseum challenge: Museum-centered visual recognition. 2014

[5] The dataset contains 6,629 artists in total, with high variation in the number of pieces per artist. For example, Rembrandt has 1,384 pieces, and Vermeer has only 4. There are 350 artists with more than 50 pieces, 180 artists have around 100, and 90 artists have 200 pieces.

in the Rijksmuseum in Amsterdam (the Netherlands). For the artist classification challenge, the paper said they read a test accuracy of about 60%. The year later, Saleh and Elgammal's paper[6] was the first attempt to identify artists with a large and varied dataset, but still using generic features. The collection they used has images of 81,449 fine-art paintings from 1,119 artists ranging from fifteen centuries to contemporary artists, reaching an accuracy of 59%[7].

More recent attempts are related to the *Painter by Numbers*, a **Playground Prediction Competition** by *Kaggle*[8]. This competition used a pairwise comparison scheme: participants had to create an algorithm which needs to examine two images and predict whether the two images are by the same artist or not. Thus, it is not our same objective, however it can be consider the first application of Deep Learning to the problem. The real deal was taken by Nitin Viswanathan[9] in 2017. Viswanathan, using the same dataset of the mentioned *Kaggle Challenge*, proposed the use of ResNet with transfer learning (he first held the weights of the base ResNet constant and updated only the fully-connected layer for a few epochs). This trained network reached a train accuracy of 0.973 and a test accuracy of 0.898.

1.2 Dataset

Unfortunately, the dataset provided by the *Kaggle Challange* is too huge to be used in Colab, in fact it is about 60GB unbearable on the free version of Colab, which provides only about 30GB of disk. Stated that, we decided to use a different dataset[10] with only 2GB of data and about 8k unique images.
The data downloaded from Kaggle has the following directories and csv file:

```
/
├── images
│   └── images
│       ├── Albrecht_Durer
│       ├── Alfred_Sisley
│       ├── Amedeo_Modigliani
│       ├── Andrei_Rublev
│       ├── Andy_Warhol
│       ├── Camille_Pissarro
│       ├── Caravaggio
│       ├── Claude_Monet
│       ├── Diego_Rivera
│       ├── Diego_Velazquez
│       ├── Edgar_Degas
│       ├── Edouard_Manet
│       ├── Edvard_Munch
│       └── an many others (total of 50 different artists)
├── resized
└── artists.csv
```

The *resized* directory is not useful for our studies, hence we deleted it to save space on the disk. On the other hand, we first use the *csv* file to select only the artists with at least 200 pieces, this operation was done to reduce the number of classes to a number per which the ratio between the number of artists and images was reasonable for learning. Even done that, the dataset was still unbalanced, e.g. Van Gogh's paintings are 877 against the 239 of Chagall's,

[6] B. Saleh and A. M. Elgammal. Large-scale classification of fine-art paintings: Learning the right metric on the right feature. CoRR, abs/1505.00855, 2015

[7] In the paper they tried to use also CNN, but reaching only an accuracy of 33.62%

[8] https://www.kaggle.com/c/painter-by-numbers/data

[9] Nitin Viswanathan, Artist Identification with Convolutional Neural Networks

[10] https://www.kaggle.com/ikarus777/best-artworks-of-all-time

thus we consider to compute **class weights** in order to use them in the *fit function*:

$$\text{class\_weights} = \frac{\text{Total number of paintings considered}}{\text{Number of artists considered} \cdot \text{Number of paintings per author}}$$

Then, we modified the structure of the *images/images* directory in order to create two directories, **train** and **test**, containing 90% and 10% of the images from each different artist's directory respectively (considering only the artists with at least 200 paintings). The newly created directories have the same structured of *images/images*. This was done in *python* in this way:

```
import os
import numpy as np
import shutil

rootdir= '/content/images/images' #path of the original folder
classes = os.listdir(rootdir)

for i, c in enumerate(classes, start=1):
  if c not in artists_top_name.tolist():
    shutil.rmtree(rootdir + '/' + c)
    continue
  if not os.path.exists(rootdir + '/train/' + c):
    os.makedirs(rootdir + '/train/' + c)
  if not os.path.exists(rootdir + '/test/' + c):
    os.makedirs(rootdir + '/test/' + c)

  source = os.path.join(rootdir, c)
  allFileNames = os.listdir(source)

  np.random.shuffle(allFileNames)

  test_ratio = 0.10
  train_FileNames, test_FileNames = np.split(np.array(allFileNames),
                                                        [int(len(allFileNames)*
                                            (1 - test_ratio))])

  train_FileNames = [source+'/'+ name for name in train_FileNames.tolist()]
  test_FileNames = [source+'/' + name for name in test_FileNames.tolist()]

  for name in train_FileNames:
    shutil.copy(name, rootdir +'/train/' + c)

  for name in test_FileNames:
    shutil.copy(name, rootdir +'/test/' + c)
```

After that we created the train/validation/test-sets using the *image_dataset_from_directory* function provided by **Keras** in the following way:

```
import tensorflow as tf

training_images = tf.keras.preprocessing.image_dataset_from_directory(
    TRAIN_DIR, labels='inferred', label_mode='categorical',
    class_names=None, color_mode='rgb', batch_size=BATCH_SIZE,
    image_size=(IMAGE_HEIGHT,  IMAGE_WIDTH), shuffle=True, seed=RANDOM_SEED,
    validation_split=VALIDATION_SPLIT, subset='training',
    interpolation='bilinear', follow_links=False
)

val_images = tf.keras.preprocessing.image_dataset_from_directory(
    TRAIN_DIR, labels='inferred', label_mode='categorical',
    class_names=None, color_mode='rgb', batch_size=BATCH_SIZE,
    image_size=(IMAGE_HEIGHT, IMAGE_WIDTH), shuffle=True, seed=RANDOM_SEED,
    validation_split=VALIDATION_SPLIT, subset='validation',
    interpolation='bilinear', follow_links=False
)
```

```
18
19 test_images = tf.keras.preprocessing.image_dataset_from_directory(
20     TEST_DIR, labels='inferred', label_mode='categorical',
21     class_names=None, color_mode='rgb', batch_size=BATCH_SIZE,
22     image_size=(IMAGE_HEIGHT, IMAGE_WIDTH), shuffle=True, seed=RANDOM_SEED,
23     interpolation='bilinear', follow_links=False
24 )
```

Where *VALIDATION_SPLIT* is equal to 0.1.
Obtaining in this way:

- 3478 files for training (belonging to 11 classes).
- 386 files for validation (belonging to 11 classes).
- 438 files for testing (belonging to 11 classes).

Hence, we have a total of 4299 different pictures.

2 — General Information Useful for Training

In the following chapters we will make use of different strategies:

- Class Weights (already talked about)
- Data augmentation
- Regularization
- Dropout
- Multiple activation functions
- Multiple optimizers
- Genetic Algorithms

In order to allow a better and faster reading of the tests done, in the following paragraph the mentioned strategies are discussed.

2.1 Data Augmentation

Data augmentation takes the approach of generating more training data from existing training samples by augmenting the samples via a number of random transformations that yield believable-looking images. The goal is that, at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data so it can generalize better. In Keras, this can be done by adding a number of data augmentation layers at the start of your model. In our model, we included the following transformation:

```
data_augmentation = ks.Sequential(
    [
        layers.RandomFlip('horizontal'),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
        layers.RandomHeight(0.1),
        layers.RandomWidth(0.1)
    ]
)
```

2.2 Regularization

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation. This is called "regularizing" the model, because it tends to make the model simpler, more "regular", its curve smoother, more "generic"; thus it is less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data. A common way to mitigate overfitting is to put constraints on the complexity of a model by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called *weight regularization*, and it's done by adding to the loss function of the model a cost associated with having large weights. This cost comes in two flavors:

1. *L1 regularization*—The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).

2. *L2 regularization*—The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights).

3. *L1_L2 regularization*—Combine L1 and L2.

2.3 Dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks; it was developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.

In **keras** can be set using the *layers.Dropout* function passing as parameter the *dropout rate*, in our case always 0.5.

2.4 Activation Functions

In the studies done in the following chapters we used three different activation functions:

- $ReLU$: $\max(0, x)$
- ELU: $\max(0.2x, x)$
- *Leaky ReLU*: $f(x) = \begin{cases} x & x > 0 \\ \alpha(\exp(x) - 1) & x \leq 0 \end{cases}$

They will be useful in the genetic algorithm analysis done fore the *scratch architecture* and the *VGG16*

2.5 Optimizers

An optimizer is the mechanism through which the model will update itself based on the training data it sees, so as to improve its performance. In our project we make use of:

- *RMSprop*: the gist of RMSprop is to:
  - Maintain a moving (discounted) average of the square of gradients
  - Divide the gradient by the root of this average
  - It uses plain momentum, not Nesterov momentum.
- *Adam*: stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

2.6 Genetic Algorithms

Genetic algorithms are a family of search algorithms inspired by the principles of evolution in nature. By imitating the process of natural selection and reproduction, genetic algorithms can produce high-quality solutions for various problems involving search, optimization, and learning. At the same time, their analogy to natural evolution allows genetic algorithms to overcome some of the hurdles that are encountered by traditional search and optimization algorithms, especially for problems with a large number of parameters and complex mathematical representations. Thus, they come in handy for optimizing our networks. In order to make use of genetic algorithms we must decide some components, which are:

- Genotype
- Population

- Fitness Function
- Selection Algorithm
- Crossover Algorithm
- Mutation Algorithm
- Elitism

All of these components are implemented using the python library **deap**[11].

2.6.1 Genotype

The *genotype* is a collection of genes that are grouped into chromosomes. In our specific case, our genes are bounded real-valued encoded and they represent three different parameters:

- *activation_function*: bounded between 0, 1.999, 2.999. The integer part stands for ReLU (0), ELU (1), Leaky Relu (2);
- *optimizer*: bounded between 0 and 1.999. The integer part stands for rmsprop (0) and adam (1);
- *learning rate*: bounded between 0.001 and 0.1.

2.6.2 Population

At any point in time, genetic algorithms maintain a population of individuals (i.e., chromosomes)– a collection of candidate solutions for the problem at hand. Since for each individual we train a model and evaluate its performance, we decided to use only 20 individuals per generation to limit the computation.

2.6.3 Fitness Function

At each iteration of the algorithm, the individuals are evaluated using a fitness function (also called the target function). This is the function we seek to optimize or the problem we attempt to solve. In our problem, the fitness function is the the function which calculate the maximum validation accuracy reached by the model. Our objective is maximizing the validation accuracy.

2.6.4 Selection Algorithm

After calculating the fitness of every individual in the population, a selection process is used to determine which of the individuals in the population will get to reproduce and create the offspring that will form the next generation. The selection algorithm used by as is *tournament selection*. In each round of the tournament selection method, two individuals are randomly picked from the population, and the one with the highest fitness score wins and gets selected. We decided to select only two individuals since our population is small and selecting more could cause an abuse in exploitation.

2.6.5 Crossover Algorithm

To create a pair of new individuals, two parents are chosen from the current generation, and parts of their chromosomes are interchanged (crossed over) to create two new chromosomes representing the offspring. There exists different algorithms applicable to real-value encoded

[11] https://deap.readthedocs.io/en/master/

individuals, we decided to use the *Simulated Binary Bounded Crossover*, which is a bounded version of the Simulated Binary Crossover (SBX)[12].

The idea behind the simulated binary crossover is to imitate the properties of the single-point crossover that is commonly used with binary-coded chromosomes. One of these properties is that the average of the parents' values is equal to that of the offsprings' values. When applying SBX, the two offspring are created from the two parents using the following formula:

$offspring_1 = \frac{1}{2}[(1+\beta)parent_1 + (1-\beta)parent_2]$

$offspring_2 = \frac{1}{2}[(1-\beta)parent_1 + (1+\beta)parent_2]$

Here, β is a random number referred to as the *spread factor*.
This formula has the following notable properties:

- The average of the two offspring is equal to that of the parents, regardless of the value of β.
- When the β value is 1, the offspring are duplicates of the parents.
- When the β value is smaller than 1, the offspring are closer to each other than the parents were.
- When the β value is larger than 1, the offspring are farther apart from each other than the parents were.

The probability to mate is set equal to 0.9.

2.6.6 Mutation Algorithm

The purpose of the mutation operator is to periodically and randomly refresh the population, introduce new patterns into the chromosomes, and encourage search in uncharted areas of the solution space. As mutation algorithm we decided to use the *Polynomial Bounded* method, which is a bounded mutation operator that uses a polynomial function for the probability distribution.
The probability to mutate is set equal to 0.5.

2.6.7 Elitism

While the average fitness of the genetic algorithm population generally increases as generations go by, it is possible at any point that the best individual(s) of the current generation will be lost. This is due to the selection, crossover, and mutation operators altering the individuals in the process of creating the next generation. In many cases, the loss is temporary as these individuals (or better individuals) will be re-introduced into the population in a future generation.

However, if we want to guarantee that the best individual(s) always make it to the next generation, we can apply the optional elitism strategy. This means that the top n individuals (n being a small, predefined parameter, in our case 5) are duplicated into the next generation before we fill the rest of the available spots with offspring that are created using selection, crossover, and mutation. The elite individuals that were duplicated are still eligible for the selection process so they can still be used as the parents of new individuals.

Elitism is made possible in our code thanks to the function *eaSimpleWithElitism*, which is a modification of the function *eaSimple* present in the **Deap** framework.

[12] https://content.wolfram.com/uploads/sites/13/2018/02/09-2-2.pdf

3 — CNN from Scratch

4 — Pre-Trained Models

This section describes the results obtained using different pre-trained architecture and strategies[13]. The pre-trained networks here tested are:

- VGG16
- ResNet50V2
- ResNet101V2
- InceptionV3

4.1 VGG16

VGG16**??** is a convolutional neural network model proposed by Simonyan et al., with several 3x3 convolutional layers in cascade occasionally interleaved with 2x2 max-pooling layers forming the so called *blocks*. Developed for the ILSVRC2014 challenge, it was able to achieve a top-5 accuracy of 92.7 on ImageNet.

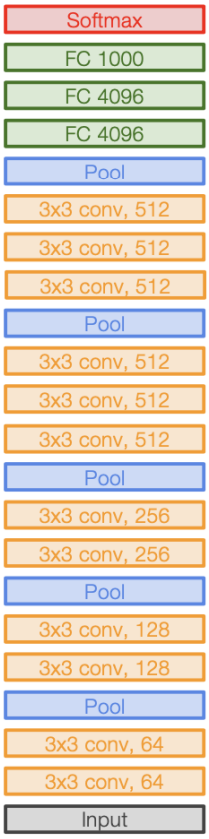

Figure 2: VGG16 Architecture

4.1.1 Test 1: Classical VGG16 (Feature Extraction)

The original VGG16 comes with a couple of 4096 FC layers followed by 1000 softmax neurons, which is alright for ImageNet but definitely oversized for our purpose. Hence, the convolutional

[13]The data augmentation strategy is always used since we have very little data

base is left as it is, and the fully-connected block is replaced by the a shrunk version with only 256 neurons per layer, followed by our prediction layer made of 11 neurons**??**.

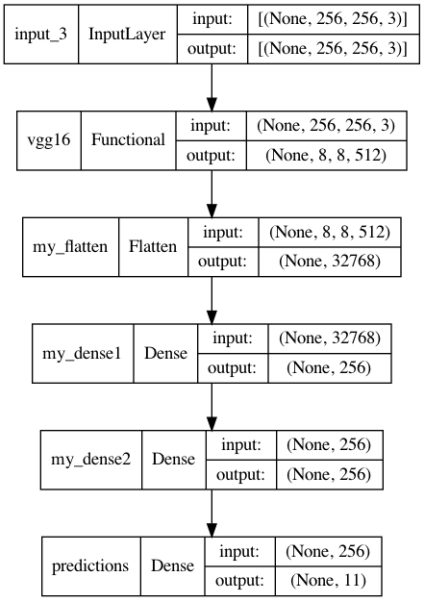

Figure 3: Our Feature Extraction Network

The result obtained, using RMSprop as optimizer, are:

Feature Extraction				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
16	0.7927	0.7471	2.5532	3.6978

The network begin to overfit very fast, hence some regularization methods are needed.

4.1.2 Test 2: Adding dropout to Test 1

We have two possible positions to use the dropout layer in our network and they are after each 256-dense layer, thus we change our previous network in the following way:

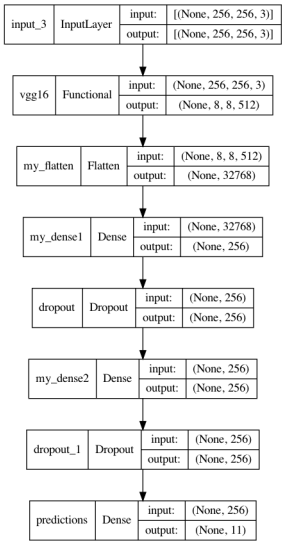

Figure 4: Our Feature Extraction Network + Dropout

4.1.3 Test 3: Completely New Output Layers Architecture (Feature Extraction)

4.1.4 Test 4: Test 4 with Data Augmentation

4.1.5 Test 5: Test 4 with Data Augmentation + Regularization

4.1.6 Test 6: Fine Tuning with One Layer

4.1.7 Test 7: Fine Tuning with Two Layers

4.1.8 Test 8: Genetic Algorithm for Hyper-parameters and Architecture Optimization

4.2 ResNet50V2

4.2.1 Test 1: Classical ResNet50V2 (Feature Extraction)

On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when you're working with image data.

4.2.2 Test 2: Completely New Output Layers Architecture

4.2.3 Test 3: 1 and 2 with Data Augmentation

4.2.4 Test 4: Fine Tuning with One Layer

4.2.5 Test 5: Fine Tuning with Two Layers

4.3 ResNet101V2

4.3.1 Test 1: Classical ResNet101V2 with 50 classes

4.3.2 Test 2: Completely Newly Output Layers Architecture

4.3.3 Test 3: Fine Tuning with One Layer

4.3.4 Test 4: Fine Tuning with Two Layers

4.4 InceptionV3

4.4.1 Test 1: Classical ResNet101V2 with 50 classes

4.4.2 Test 2: Completely Newly Output Layers Architecture

4.4.3 Test 3: Fine Tuning with One Layer

4.4.4 Test 4: Fine Tuning with Two Layers