
---

---

Program: Data Science for Economics
2023/24

*Author:*
Pietro Padovese

*ID:*
12356A

*Course Coordinator:*
Prof. Dario Malchiodi

November 14, 2024

# Contents

# 1 Introduction

## 1.1 Task and Dataset

The primary objective of this project is to design, implement, and evaluate various neural network architectures to develop an effective image classification model. The aim is to classify each painting based on its artist, allowing the model to identify the creator of a given artwork solely from its visual characteristics. By testing different neural network structures, we seek to identify the architecture that offers the highest accuracy and reliability in artist classification using the Prado Museum's collection of paintings.

The dataset used for this project consists of paintings by the five most frequently represented artists in the Prado Museum's collection. This subset was chosen to focus on the artists with the most substantial representation, ensuring a balanced dataset for training and evaluation. A total of 2.058 paintings are included, distributed among the artists as follows:

| | **Goya** | **Bayeu** | **Haes** | **Pizarro** | **Ribera** | **Total** |
|---|---|---|---|---|---|---|
| **Train** | 757 | 329 | 225 | 185 | 151 | 1,647 |
| **Validation** | 176 | 69 | 64 | 62 | 40 | 411 |
| **Total** | 933 | 398 | 289 | 247 | 191 | 2058 |

## 1.2 Preprocessing

Prior to being fed into the neural networks, the images underwent several preprocessing steps designed to standardize and enhance the dataset for optimal model performance. Initially stored in JPEG format, all images were resized to a uniform dimension of 224x224 pixels and converted to RGB color mode, ensuring consistency in input shape and color channels across the dataset. The pixel values, originally ranging from 0 to 255, were then rescaled to a [0,1] interval. This normalization helps to streamline computational efficiency and ensures uniform input values, aiding in the model's learning process during training and evaluation.

In addition to rescaling, the images were augmented with random horizontal flips and rotations. These transformations introduce variations in the dataset by creating altered versions of the original images, which is particularly beneficial when working with a relatively small dataset, as in this case. Such data augmentation techniques encourage the model to learn more robust features that are invariant to small positional changes,

thereby improving generalization to new images.

Finally, each image was subject to per-image standardization, a technique that adjusts each image to have a mean of zero and a unit standard deviation. This process further normalizes the images by reducing brightness and contrast variations, allowing the neural network to focus on the most distinctive features across the images. Altogether, these preprocessing steps prepare the images to be efficiently processed by the neural networks and enhance the model's ability to recognize distinctive artist-specific features across diverse paintings.

## 1.3 Class Imbalance

To account for class imbalance in the dataset, the models use two techniques: output bias initialization and class weights during training. These adjustment aim to improve the models' performance, particularly on underrepresented classes, and to avoid bias toward more frequent classes.

The models' output layer, includes an output bias initialiazer. The output bias is calculated based on class frequencies, aiming to initialize the models' predictions closer to the actual class distribution. We begin by calculating the relative frequency ($f_i$) of each class. To ensure that the models' softmax outputs approximate these frequencies, we initialize the bias terms $b_i$ in the output layer so that the softmax probabilities $pi = \frac{exp(b_i)}{\sum_{j=1}^{5} exp(b_j)}$ are close to $f_i$. This leads to a system of equations where $p_i - f_i = 0$, or equivalently, $exp(b_i) = f_i \cdot \sum_{j=1}^{5} exp(b_j)$. Since the system is nonlinear, we solve for the $b_i$ values using numerical methods, specifically the SciPy's fsolve function, which iteratively finds values of $b_i$ that approximate the target distribution.

To further address class imbalance during training, class weights are applied to encourage the model to give more attention to underrepresented classes. These weights are calculated based on the inverse frequency of each class, assigning higher weights to less frequent classes. Mathematically, for each class $i$, the weight $w_i$ is computed as $w_i = \frac{1}{n_i} \cdot \frac{N}{C}$, where $n_i$ is the sample count for class $i$, $N$ is the total number of samples, and $C$ is the total number of classes. These class weights are then incorporated into the training process. This weighting scheme enables the model to treat misclassifications in underrepresented classes as more impactful, thus reducing the chance of the model being biased towards more frequent classes.

## 1.4 Dataset caching and optimization

To optimize the data loading process during model training, the TensorFlow `AUTOTUNE` feature is employed to automatically adjust the prefetching and caching settings for the training and validation datasets. Caching the dataset involves storing it in memory, which significantly reduces disk I/O operations and improves data access speed. In this case, the entire dataset is loaded into memory after the first read, allowing for faster retrieval in subsequent epochs. By employing the prefect method, the data pipeline fetches batches of data asynchronously while the model is actively training, effectively overlapping data preparation with model computation.

# 2 CNN-01

## 2.1 Architecture

The first architecture examined incorporates a basic convolutional neural network (CNN). After preprocessing, as detailed above, this model comprises four convolutational block, each containing 3 convolutional layers, with an increasing number of filters, and followed by an average pooling layer. The output of the final convolutional block undergoes flattening and pass throguh two additional Dense layer. These two layers can enhance the feature aggregations capabilities of the model. After extracting complex features with the convolutional layers, they are aggregated and combined in the dense layer. Having more dense layer means increasing the number of non-linear transformations to the feature space that the model can capture. Finally to the last dense layer which return the desired output. Below, each component is analysed in greater detail.

### Convolutional Layers

The goal of the convolutional layer is to extract features from the input data. It consists of a set of filters (or kernels) which are small-sized matrices that slide over the input data. Each filter detects specific patterns by performing a convolution operation. During this operation a dot product is computed between the filter weights and the corresponding input values. A process that is repeated across the entire image, producing a feature map that highlights the presence of different patterns or features.

All three convolutional layers of this network use a kernel size of 3x3 with padding, a technique that adds zeros around the edges of the image, in order to ensure that the spatial dimensions of the ouput feature maps remain the same as the input data.

The layers, on the other hand, differ in the number of filters applied. They have been structured with an increasing number of filters. The idea behind this choice is that as we move deeper into the network, the complexity of the features that the model can recognize increases, allowing the network to achieve a more refined understanding of the images.

### ReLU

the output of the convultional layers, before being passed to the subsequent layer, is transform by the ReLU (Rectified Linear Unit) activation function.

$$f(x) = max(0, x)$$

This not only improve the computational efficiency by introducing sparsity in the network, but also tackles the problem of diminished gradients, which may emerge in a deeper model like this one. This problem refers to the fact that during the backpropagation process, since the gradients are computed through the chain rule of calculus from the output layer, backward through the network, the repeated multiplications that happens in each layer may lead the gradient to diminish significantly in magnitude, slowing down the process of training. For how the ReLU is constructed, multiplying for its gradient is equivalent to a multiplication by 1 when the input is positive and zero otherwise, meaning that it will either stays as it is or become exactly zero.

The choice to use the ReLU activation function has also led to reflections on the weight initialization process. The optimization path of adjusting the weight associated to each parameter, needs indeed a starting point, a choice that can strongly affect the training process. Although randomly drawing weights from a distribution in a specific range is a possibility, more tailored systems have been developed that consider the type of activation function used.

The He initializer control the variance of the gradients. Since the ReLU reduces the negatives value to zero, halving the variance, He initializer compensates for this reduction by doubling the scale of the weight's variance. In this sense the weights are drawn from a normal distribution distribution with mean zero and standard deviation $\sqrt{2/n_{in}}$. This initializer helps in avoiding the dead neurons problem, which happens when a neuron outputs zero for all inputs. They can occur when the initial weights are too small. The He initializer should ensure that the initial weights are larger enough to keep the majority of neurons active.

### BatchNormalisation

Between the convolutional layer and the activation function a regularisation method has been added. Now the feature maps undergo the process of BatchNormalisation, which helps to improve training stability and speed up the convergence. During the training process, the distribution of activations within a neural network shifts, a problem known as internal covariate shifts. This can cause the optimization algorithms some problems

since it needs continually to adapt to the changing distribution. By normalizing the activations of each layer within a mini-batch, we ensure that the distribution of the activations remains stable during training.

#### Average Pooling Layer

Each convolutional block is followed by an average pooling layer with a kernel size of 2x2. These layers reduces each 2x2 patch to its average value, effectively downsampling it by a factor of 4. This operation not only reduces the computational costs, but also makes the features learned by the model more robust to position and orientation changes in the input image, while preserving the information from the input feature map.

### 2.1.1 Compiling

Before starting with the training process, we need first to configure the model specifying various parameters that govern how the model learns and how it optimizes its performance.

#### Loss function

The loss function quantifies the distance between the model predictions and the actual labels. It serves as feedback to the optimizer to guide its process of adjusting the parameters of the model.

In multiclass classification problems, a common choice for the loss function, and the one adopted in this project, is the sparse categorical cross-entropy loss function. This function measures the confidence of predictions by comparing the predicted probabilities across multiple classes to the actual class labels, providing more nuanced information than simple class labels alone. Sparse categorical cross-entropy is particularly useful when dealing with integer-encoded class labels, as it allows for efficient computation without requiring one-hot encoding.

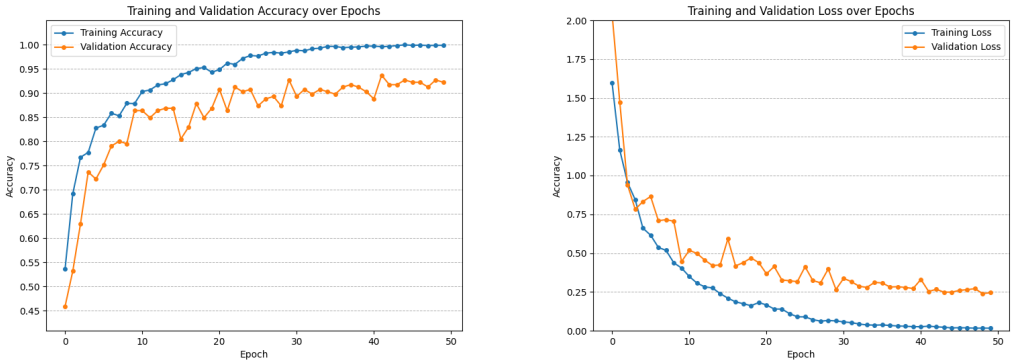
#### Optimizer

The purpose of the optimizer is to adjust the parameters of the model iteratively during the training in order to minimize the define loss function. The chosen optimizer was the Adaptive Moment Estimation (Adam). This algorithm can be though as an "extension" of the Gradient Descent, which minimize the loss function by moving the weights in the direction of the steepest descent of the function's gradient. The size of this steps is determined by the learning rate, which determines how much we adjust the parameters. Traditional gradient descent methods use a fixed learning rate for all parameters, and

remains constant for the whole training process. With Adam instead individual learning rates are used for each parameter and are updated at each iteration using the first and second moment of the gradients. The first moment helps in smoothing the optimization trajectory by accumulating past gradients, the second allows to adjust the rates based on the variance of the gradient, allowing parameters with large gradients to have smaller updates and vice versa.

## 2.2 Result analysis

Figure 2.1: Sequential Results




(a) Accuracy

(b) Loss

Overall, the Sequential model performs well, showing solid accuracy on the validation set despite its simpler architecture. After an initial phase of learning, the training and validation accuracies begin to diverge around the 10th epoch, a common sign of overfitting as the model starts to specialize on the training data. However, by the 30th epoch, validation accuracy stabilizes within a strong range of 0.89-0.93, indicating that the model has effectively learned the key patterns in the data. Similarly, the validation loss decreases and reaches a consistent level near 0.25, reinforcing the stability of the model's performance. While more complex architectures may offer further improvements, the Sequential model's results are commendable, especially given the straightforward design.

# 3 Residual Learning

## 3.1 Architecture

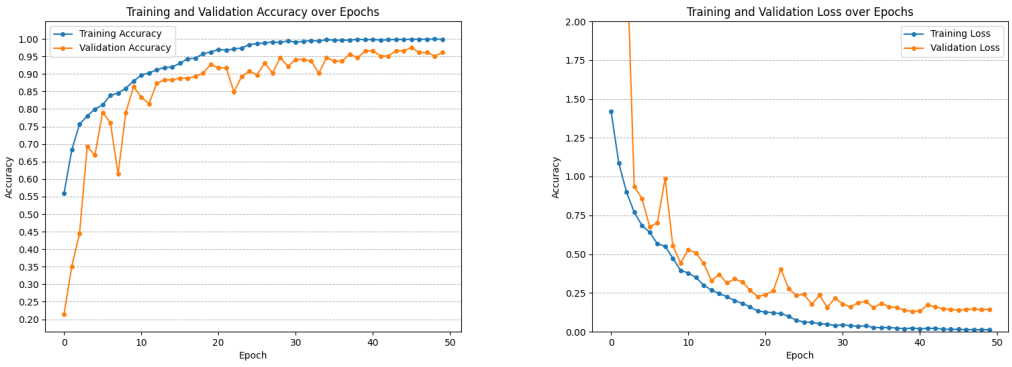
For the second model a different approach has been taken regarding the architecture of the network. This new implementation revolves around the concept of residual learning, introduced with ResNet model. This framework aims at easing the training of deep neural networks by introducing skipping connections within network layers. If in traditional CNN architectures the aim is to transform the input data into a representation progressively closer to the desired output, in residual learning the focus shift to learning residual function that capture the discrepancy between the desired output and input to a particular layer.

The core element in this architecture is the identity/convolutional block (with the difference being whether or not the input activation has the same dimension as the output). Within this block, the input will follow two paths. The "main path", which is a standard CNN flow, and the "Skip connection path",that allows the input to bypass this stream. If we denote the input as $X$, and the desired mapping as $H(X)$, we are basically letting the network fitting $F(X) = H(X) - X$, the residual mapping. The output of the fitting and the initial input are finally added together and passed to the activation function. Since operating on the residual it is easier than figuring out from scratch the entire input-output mapping, this technique should help our model to converge faster.

## 3.2 Result analysis

The ResNet model demonstrates strong and sustained learning throughout training, with metrics continuing to improve for a more extended period compared to the Sequential model. Validation accuracy consistently surpasses the 0.95 mark, while validation loss settles around a low 0.20 range. The improved results indicate that the model effectively leverages its increased capacity to capture complex patterns, largely due to skip connections. These connections allow each layer to refine features rather than completely relearn them, enabling the model to achieve a deeper and more nuanced understanding of the data.

Figure 3.1: ResNet results




(a) Accuracy

(b) Loss

# 4 DenseNet

## 4.1 Architecture

For this model, a novel approach has been taken with the adoption of DenseNet-style architecture, a framework designed to enhance information flow across the network through dense connectivity. While traditional convolutional networks process data sequentially through each layer, DenseNet introduces an interconnected structure in which each layer receives input from all previous layers within the same block. This architecture not only strengthens feature propagation but also mitigates the vanishing gradient problem, a frequent challenge in deep networks where gradients can diminish as they propagate back through layers.

The essential building block in DenseNet is the dense block. Within each dense block, the network operates in a unique "feed-forward" style: each layer's output is concatenated with the outputs of all preceding layers and is passed as input to all subsequent layers. Thus, each layer has access to a cumulative feature set, which grows as the network depth increases. For example, if we denote the feature maps produced by the $l$-th layer as $Xl$, then the input to the $(l+1)$-th layer would be a concatenation of all previous feature maps, $[X0, X1, ..., Xl]$ By retaining and reusing these features throughout the network, DenseNet encourages feature reuse and reduces the need for redundant learning across layers.

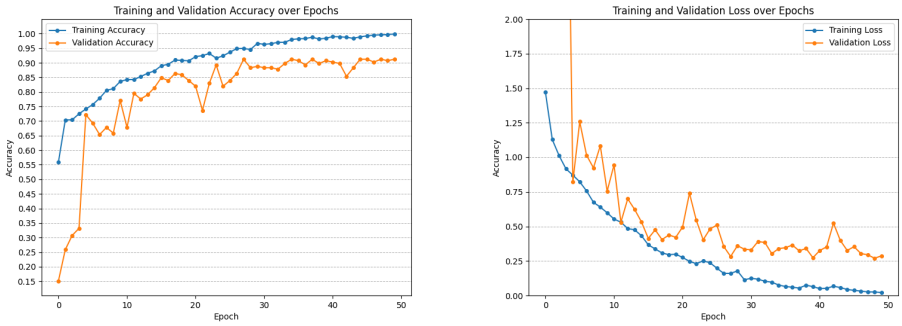
To manage the growth in the number of features from layer to layer, DenseNet includes transition layers between dense blocks. These transition layers apply convolutional and pooling operations to reduce the spatial and feature dimensions, keeping the network computationally efficient and scalable. The design of DenseNet makes it particularly resource-efficient, as it achieves comparable or improved performance with fewer parameters than more conventional deep networks.

Through its dense connectivity, DenseNet facilitates efficient gradient flow and allows each layer to build upon the representations learned by all previous layers.

## 4.2 Result analysis

The DenseNet model is performing worse than both the Sequential and ResNet models in this case. It takes longer to converge, with accuracy staying just above 0.90 and loss fluctuating significantly during training. Despite its potential for better feature reuse, DenseNet's complexity, with dense connections between layers, increases the number of parameters and computations, making it harder to train effectively. This results in slower convergence and unstable loss, which likely contributes to its underperformance. In comparison, the simpler Sequential and ResNet models are able to achieve more stable and faster training with better generalization.

Figure 4.1: DenseNet results




(a) Accuracy

(b) Loss

# 5 Conclusion

In conclusion, among the three architectures analyzed, the ResNet model emerges as the best performer. Its deeper architecture, enhanced by skip connections, enables it to capture more complex patterns in the data, leading to superior accuracy and better generalization. The model consistently improves over a longer training period, with stable learning and low validation loss, making it the most effective at handling the complexity of the dataset.

The Sequential model, while simpler, still demonstrates solid performance with stable training and reasonable accuracy. Although it is prone to some overfitting as training progresses, it maintains a good balance between performance and complexity, making it a strong contender for simpler tasks.

The DenseNet model, while theoretically offering better feature reuse through its dense connections, struggles with slower convergence and unstable loss due to its increased complexity and larger number of parameters. This makes it less efficient and effective compared to both the Sequential and ResNet models. Therefore, while DenseNet has potential, it underperforms in this analysis, with ResNet being the clear winner overall.