Prado Artworks Support Classification with Neural Networks

Leonardo Acquaroli

June 12, 2024

1 Introduction

The Prado Museum, located in Madrid, is the most important Spanish artistic hub and houses one of the world's finest collections of European art. This project focuses on classifying images of artworks from the Prado Museum using **Convolutional Neural Networks (CNNs)**. The classification task is about recognizing the **support type** of the artworks among three options paper (*papel*), canvas (*lienzo*), wood (*tabla*).

2 Dataset

The dataset used for this project is taken from Kaggle and is called the "Prado Museum Pictures" dataset[1]. The pictures cover a vast collection of European art dating from the 12th century to the early 20th century. This dataset includes a table (in .csv format) with the information about each artwork and a folder with images extracted from these artworks. The whole dataset has been split in a training folder a validation folder and a test folder.

2.1 Dataset description

The dataset consists of 13,487 images of artworks realized on tens of different supports. Each image is associated with metadata (stored in the `prado.csv` file) that include the type of support, which is the starting point of the classification task.

In particular, the 10 most frequent support types in the dataset are shown in Table 1 which shows that there are three dominant supports, which are the ones on which the classification task focuses: paper (*papel*), canvas (*lienzo*), wood (*tabla*).

Support Type	**Count**
Papel	6434
Lienzo	3355
Tabla	733
Marfil	149
Placa de vidrio	127
Lámina de cobre	82
Revestimiento mural trasladado a lienzo	44
Tejido	16
Cartón	14
Terciopelo	12

Table 1: The 10 most frequent supports in the Prado Museum Pictures dataset.

[1]`https://www.kaggle.com/datasets/maparla/prado-museum-pictures`

3 Data preparation

3.1 Dataset processing

After having reduced the entire dataset to the collection of artworks realized on **paper, canvas or wood** and dropped the records without support specification the number of observations available became 10,522. Yet the dataset still had a problem, **class imbalance**.

Given the skewness in the dataset highlighted by Table 1, also among the three most frequent supports, a basic **undersampling** method was employed to balance the dataset. This involved reducing the number of images in the majority classes (*Papel* and *Lienzo*) to match the number of images in the minority class (*Tabla*). Results are visually explained by Figure 1

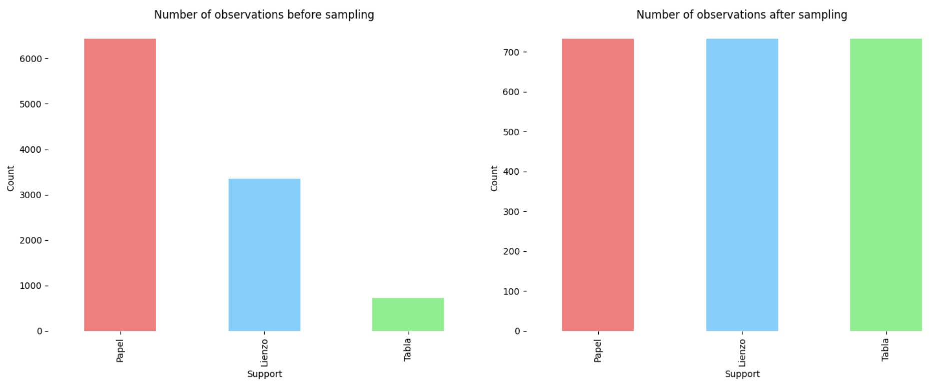


Figure 1: Number of observations before and after undersampling.

3.2 Image processing

Before the steps of the classification task (model selection, hyperparameters tuning, training, testing) images underwent a transformation pipeline made of three stages:

1. **Resizing** - from the original dimensions to a squared image of dimensions 224×224.
2. **Transformation into tensor** - in order to perform mathematical computations across three channels which are the colors channels red, green and blue (RGB).
3. **Normalization** - of pixel values using ImageNet averages and standard deviations for red green and blue, a common practice when using CNNs to allow a faster convergence during training[2].

4 Models

In this project two different **Convolutional Neural Network (CNN)** architectures are explored for the classification task: a Heavy CNN and a Light CNN.

4.1 Heavy CNN

The Heavy CNN architecture is a **baseline model** built with few layers and designed with the usual convolutional layers followed by pooling layers and fully connected layers. It also includes a DropBlock layer to improve generalization by randomly dropping blocks of pixels during training[3].

[2]Read more about normaliztion on this PyTorch documentation page or start from this thread.
[3]DropBlock: A regularization method for convolutional networks

Listing 1: Heavy CNN Architecture

```
class HeavyCNN(nn.Module):
    def __init__(self, num_classes):
        super(HeavyCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding
           =1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding
           =1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1,
           padding=1)
        self.dropblock = DropBlock2D(block_size=5, drop_prob=0.3)
        self.fc1 = nn.Linear(128 * 28 * 28, 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.dropblock(x)
        x = x.view(-1, 128 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

4.2 Light CNN

Despite the second model being dubbed as "Light", this CNN architecture features a variety of **new layers** with respect to the baseline model that, in the end, make the model more compact by **reducing the number parameters**, allowing it to train faster and more efficiently. The techniques used to this aim are:

- **Batch normalization** - which standardized the input tensors to a mean of zero and standard deviation of one and is applied after each convolution and after the fully connected linear layer
- **Dropblock** - as in the Heavy CNN
- **Adaptive Average Pooling** - which adjusts the output size to a fixed dimension regardless of the input size, providing a smooth transition between layers.
- **Dropout** - which randomly zero-outs some nodes to help keeping only the impacting ones and which is applied before the linear layer that outputs the predictions for the three classes.

Listing 2: Light CNN Architecture

```
class LightCNN(nn.Module):
    def __init__(self, num_classes):
        super(LightCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding
            =1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding
            =1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1,
            padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.dropblock = DropBlock2D(block_size=5, drop_prob=0.3)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.fc1 = nn.Linear(128, 128)
        self.bn_fc1 = nn.BatchNorm1d(128)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = self.dropblock(x)
        x = self.global_pool(x)
        x = x.view(-1, 128)
        x = self.dropout(F.relu(self.bn_fc1(self.fc1(x))))
        x = self.fc2(x)
        return x
```

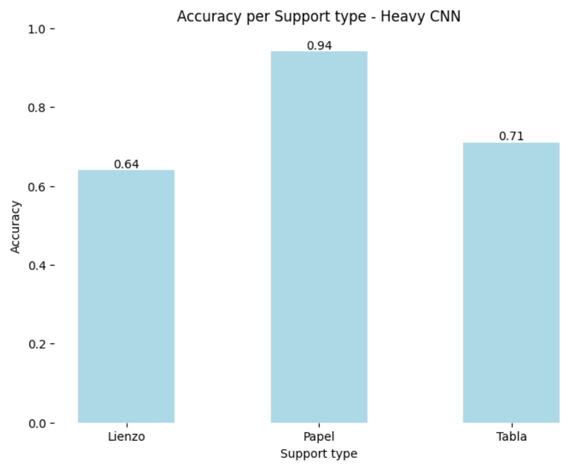
4.3 Model Selection

The first step toward achieving a satisfying classification accuracy was model selection. The performance of the Heavy and Light CNN models in terms of accuracy and model size was compared by assessing them on the validation set and **the model with the best trade-off between accuracy and size was chosen**.

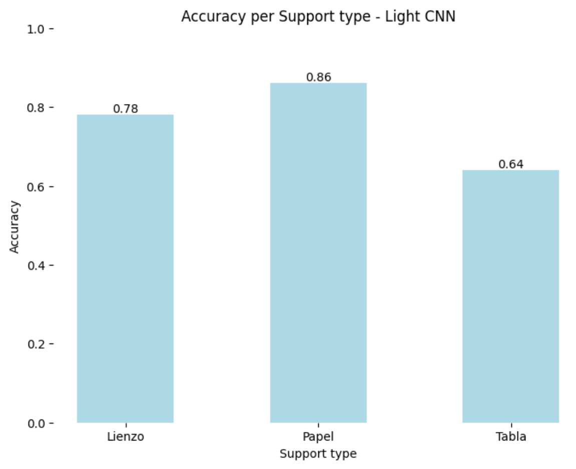
The Heavy CNN achieved an accuracy of 76.14% with a model size of $\sim 50MB$. The Light CNN, instead, achieved a slightly lower accuracy of 75.91% but made the model size almost null, $447KB$.

A model which ensures a substantially unchanged performance in the support classification task with a file size of almost the 100% smaller allows for faster experimentation, easier sharing and of course runtime saving.

Thus, in the rest of the experiment the **Light CNN architecture was used**. To grasp a deeper insight about the model comparison, Figure 2 shows how the models performed among the three classes highlighting some differences in the behavior of the two architectures. For example, the Light CNN classifies with greater success the artworks on canvas (*Lienzo*) but does a poorer job on paper (*Papel*) and wood (*Tabla*) pieces.



(a) Heavy CNN



(b) Light CNN

Figure 2: Model comparison by support type.

4.4 Light CNN advantages

In order to understand how a way smaller model can obtain a substantially equal accuracy in the classification task in this subsection, that concludes Section 4, the function and the effect of the additional layers with respect to the baseline model are explained.

Batch Normalization is used to stabilize and accelerate the training process of deep neural networks. It normalizes the input of each layer so that they have a mean of zero and a variance of one. This normalization is done on mini-batches of data. In this CNN architecture, batch normalization is applied after each convolutional layer and after the fully connected linear layer. This helps to reduce internal covariate shift, making the optimization process smoother and more efficient. The main benefits of batch normalization include allowing higher learning rates, reducing sensitivity to initialization which is especially important when the number of training epochs is reduced like in this project[4].

Dropblock is a regularization technique designed to prevent overfitting by randomly masking parts of the feature maps during training. It works similarly to dropout, but instead of dropping individual neurons, it drops contiguous regions of a feature map (blocks), effectively removing spatially correlated information. In the Light model, Dropblock is integrated to enhance its regularization capabilities, which is particularly useful in convolutional layers where spatial correlation of features can lead to overfitting. Dropblock helps to improve generalization by encouraging the network to learn more robust and diverse features, rather than relying on any specific part of the feature map[5].

Adaptive Average Pooling is used to reduce the spatial dimensions of the feature maps to a fixed size, regardless of the input size. Unlike regular pooling layers, which reduce dimensions by a fixed factor, adaptive average pooling adjusts the pooling regions dynamically to output a specified dimension. Based on the input size and the target output size, the module automatically calculates the appropriate kernel size and stride to evenly divide the input into the specified number of regions.
This technique is particularly useful in CNN architectures that need to handle varying input sizes. By applying adaptive average pooling, the model ensures that the feature maps are consistently sized before they are fed into fully connected layers. The benefits of adaptive average pooling include providing flexibility in handling inputs of different sizes and ensuring a consistent input size to the subsequent layers, simplifying the model design and improving computational efficiency[6].

Dropout is another regularization technique used to prevent overfitting by randomly dropping units (neurons) during the training phase. During each forward pass, a subset of neurons is randomly set to zero, which prevents the model from relying too heavily on any particular neuron and encourages it to learn more robust features. In the Light CNN architecture, dropout is applied before the linear layer that outputs the predictions for the three classes. This ensures that the final feature representation

[4]Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
[5]DropBlock: A regularization method for convolutional networks
[6]Multiple references for Adaptive Average Pooling: Zenn, Stack Overflow, Medium

is not overly reliant on any single neuron, promoting generalization. The main benefits of dropout include reducing overfitting, improving the robustness of the model, and leading to better performance on unseen data by ensuring that the model does not become too reliant on specific paths through the network[7].

Together, these techniques contribute to making the Light CNN model more efficient and effective by reducing the number of parameters, improving training stability, and enhancing generalization, all while maintaining almost unchanged model performance.

Finally, a waterfall graphical representation of the neural network is provided in Figure 3 showing all its building blocks.

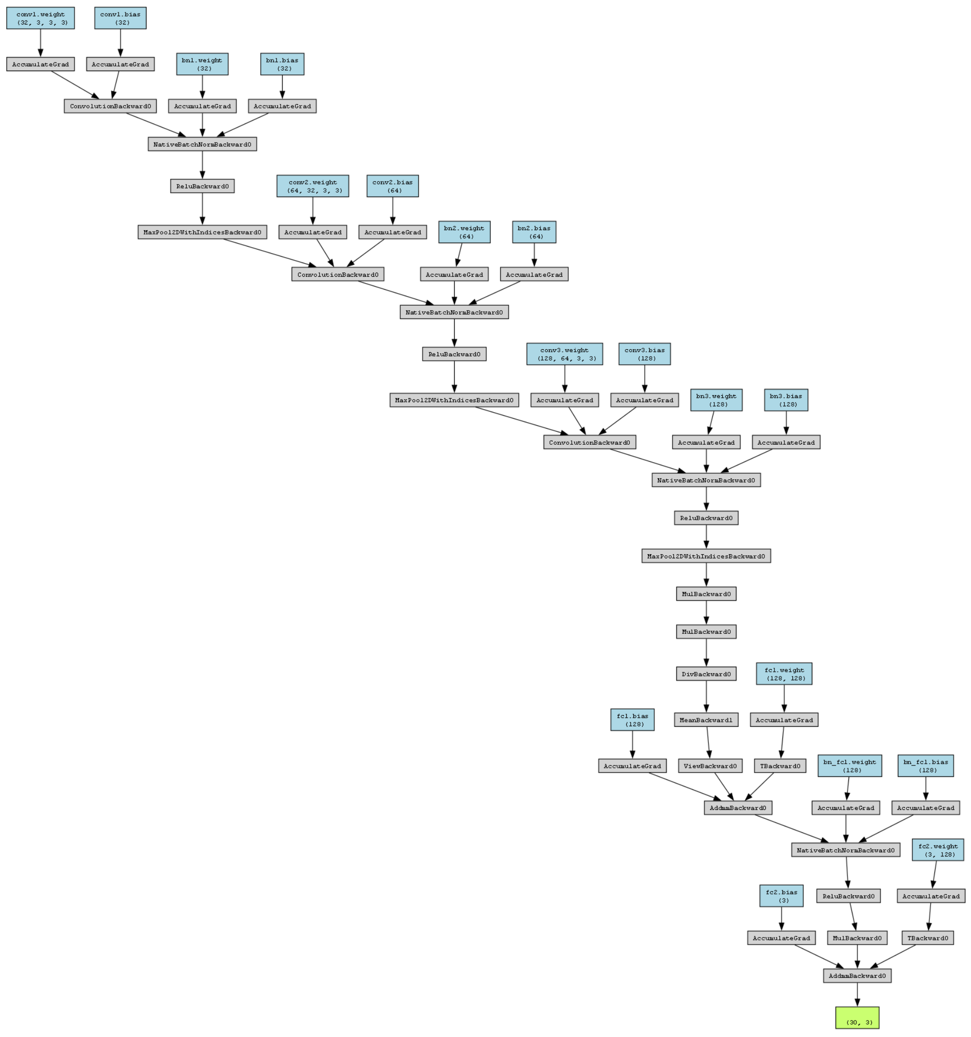


Figure 3: Tochviz representation of the Light CNN.

5 Hyperparameters tuning

To further improve the model performance, a job of hyperparameters tuning was performed using Bayesian optimization through the library Optuna[8].

Bayesian optimization is an efficient method for hyperparameters tuning that builds a probabilistic model of the objective function (in this case the accuracy of the predictions) and uses it to select the most promising hyperparameters to evaluate in the true objective function. Unlike random or grid search, which can be inefficient as they do not use information from previous evaluations, Bayesian optimization

[7]Dropout Regularization in Deep Learning Models with Keras and "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

[8]Optuna: an open source hyperparameter optimization framework to automate hyperparameter search

uses a surrogate model (in this case a Gaussian process) to predict the performance of hyperparameters sets and to balance exploration of new areas of the hyperparameters space with exploitation of known good areas. This approach significantly reduces the number of evaluations needed to find near-optimal hyperparameters, even though it can only be run sequentially thus reducing the utility of parallel computing using GPUs in the tuning phase.

For the sake of runtime and because of finite computational power availability, the number of epochs and the number of images included in a batch were fixed to, respectively, 15 and 32 for this tuning process even if both could have been treated as hyperparameters. Also, in order not to exceed GPUs' available runtime, 5 iterations of Bayesian optimization were performed.

The two hyperparameters tuned are then the learning rate which sets the pace to which the parameters are adjusted after each training epoch and the momentum accelerates convergence and reduce oscillations during optimization by accumulating an exponentially decaying moving average of past gradients and making the weights continuing to move in their direction[9].

The best accuracy achieved was 79.27% using a learning rate of approximately 3.13×10^{-4} and a momentum of approximately 0.94.

6 Results

Finally the optimized model was evaluated on a test set made of 100 images distributed as 34 pictures of class *Papel* and 33 for both *Lienzo* and *Tabla*.

The final test accuracy achieved by the best model is 79%. The detailed accuracy for each support type is illustrated in Figure 4 which shows that the wood support is the one where the model makes the majority of mistakes.

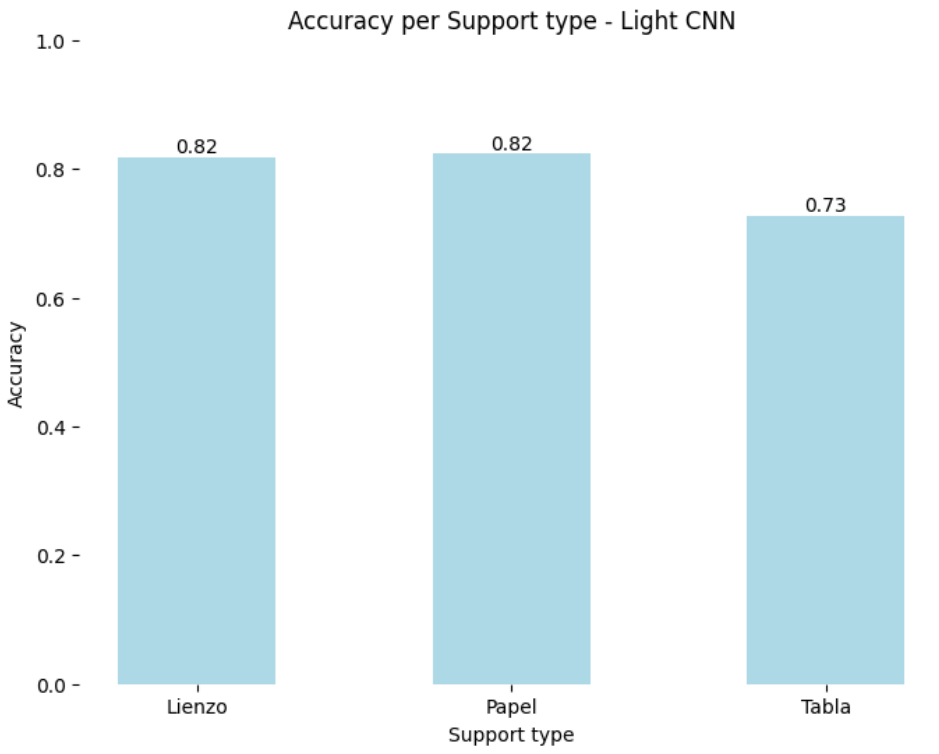


Figure 4: Test accuracy for each support type with the optimized model.

[9]Deep Learning, Ian Goodfellow and Yoshua Bengio and Aaron Courville

7 Conclusion

This project tackled the classification of Prado Museum artworks based on their support type using convolutional neural networks.

After having processed the dataset to obtain a balanced class distribution and having processed the images for performing an efficient training, model selection, and hyperparameter tuning allowed to achieve a satisfactory testing accuracy of 79%.

The model chosen for the task was a light version of the baseline model employed in a first stage becauase it guaranteed a fair trade-off between model size and classification accuracy.

Future work could involve experimenting with different data augmentation techniques, exploring more advanced neural network architectures and tuning a different set of hyperparameters.