Università degli Studi di Milano

MSc. in Data Science and Economics

Algorithms for Massive Data Report

Multiclass Leaves Classification

Federico Vinci
federico.vinci1@studenti.unimi.it

July 16, 2023

Contents

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

The goal of this project is to classify images of 12 different species of leaves using a Convolutional Neural Network (CNN) architecture. The leaf images used for classification are sourced from the Kaggle dataset csafrit2/plant-leaves-for-image-classification. The report will commence with a comprehensive analysis of the data structure, focusing on the initial form of the data and the necessary data pre-processing steps. This analysis will involve organizing the directory structure to ensure the data is appropriately arranged. An exploratory data analysis (EDA) will be conducted to gain insights into the distribution of samples across each class. This analysis will help identify any class imbalances and determine if any class balancing techniques are required. Additionally, the report will provide a theoretical framework discussing the Neural Network layers employed in this study. This overview will enhance the understanding of the tools and concepts utilized throughout the project. In the Model Deployment section, two distinct architectures will be implemented: a "simple" architecture and a "deep" Neural Network architecture. The performance of these models will be assessed in terms of accuracy to determine the superior one for the leaf classification task. Furthermore, an attempt will be made to perform hyperparameter tuning to optimize the model's performance. This process involves adjusting the model's parameters to possibly find the best configuration for achieving higher accuracy.

2 Exploratory Data Analysis

2.1 Data Structure

The dataset utilized in this project comprises 4,503 leaf images belonging to 12 distinct plant species. These species include Mango, Arjun, Alstonia Scholaris (referred to as "Alstonia"), Guava, Bael, Jamun, Jatropha, Pongamia Pinnata, Basil, Pomegranate, Lemon, and Chinar.

Initially, the directory structure of the dataset consisted of separate folders for each leaf type, as well as subfolders indicating whether the leaf was diseased or healthy. Let's examine the organization of the directories:

```
data/
└── Plant2/
    ├── train/
    │   ├── Alstonia Scholaris diseased (P2a)/
    │   │   └── images/
    │   ├── Alstonia Scholaris healthy (P2b)/
    │   │   └── images/
    │   ├── Chinar diseased (P11b)/
    │   │   └── images/
    │   └── .../
    ├── test/
    │   ├── Alstonia Scholaris diseased (P2a)/
    │   │   └── images/
    │   ├── Alstonia Scholaris healthy (P2b)/
    │   │   └── images/
    │   ├── Chinar diseased (P11b)/
    │   └── .../
    └── valid/
        └── .../
```

The current directory structure is not compatible with the goals of the analysis. To ensure compatibility and improve readability, a revised directory structure is recommended. This revised structure should organize the leaves based on their leaf type only, rather than their health status. Additionally, it is advisable to use shorter and more concise class names for better readability. By implementing this revised directory structure, we can enhance compatibility with TensorFlow's data handling functions and streamline the classification process. Using `dirtree` LaTeX library can help visualize the proposed directory structure.

```
data/
└── Plant2/
    ├── train/
    │   ├── Alstonia Scholaris/
    │   │   └── images
    │   ├── Arjun/
    │   │   └── images
    │   └── Chinar
    │       └── images
    ├── test/
    │   ├── ...
    │   ├── Alstonia Scholaris/
    │   │   └── images
    │   ├── Arjun/
    │   │   └── images
    │   ├── Chinar/
    │   └── ...
    └── valid/
        └── .../
```

By restructuring the directory, accessing the folder containing the files through the `TensorFlow` library becomes much more convenient.

By utilizing the `image_dataset_from_directory` function in Keras, we can now load the image data from the specified directory path and store it in memory as `train_data`, `valid_data`, and `test_data`. This function does also automatically resizes the images to a specified dimension, which in this case is 64x64 pixels, ensuring uniformity in size for further processing and training. During the resizing process, the **Mitchell-Netravali** cubic interpolation method is applied to estimate pixel values between known pixels, achieving a balance between sharpness and smoothness in the resized images. To optimize efficiency and memory usage during training, we choose a batch size of 64. In summary, the data will be batched in a shape of 64x64x64, where the first two numbers refer to the height and weight while the last number to the batch size.

2.2 Data Exploration

It is fair to take a look at how images appear in the dataset. The next picture displays one picture for each folder (or leaf type) in the train directory.

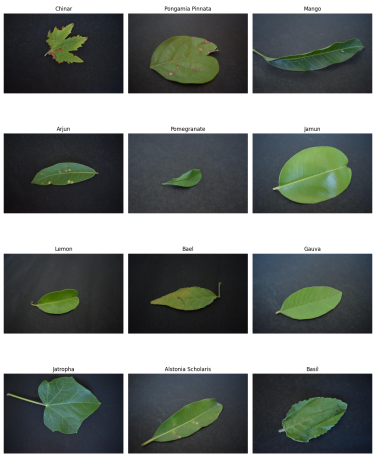

Figure 1: A glimpse at the images from our training dataset, one from each class.

In the Python code, the dictionary variable `train_count` contains the cardinality of each of the 12 classes. This cardinality is obtained by iterating through each folder in the directory and keeping up the count. Knowing the

cardinality helps address the problem of class imbalances. By examining **Figure 2**, we can identify which class is in the minority, namely Basil, Chinar, or Bael. Note that these classes correspond to leaves that have only one instance among the healthy or diseased leaf images.

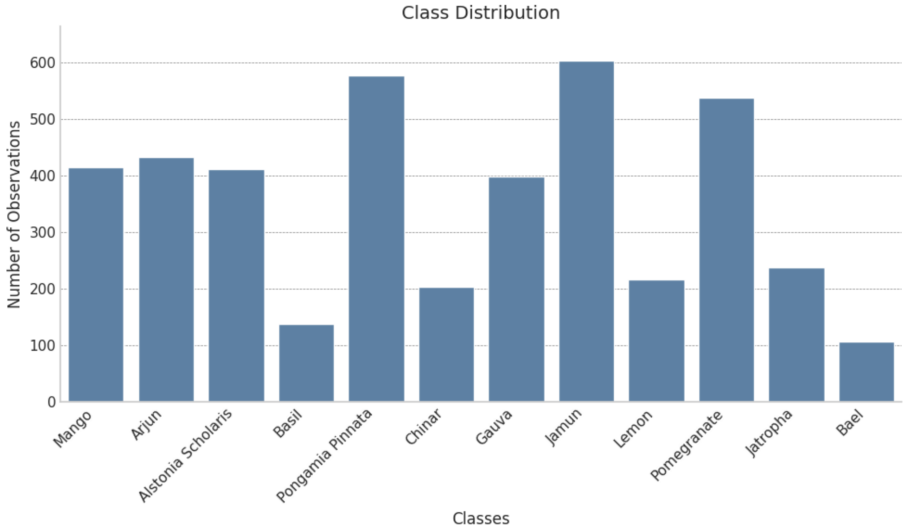

Figure 2: Classes Distribution

Class imbalance can result in biased model predictions, where the model tends to favor the majority class due to its prevalence in the training data. This bias can lead to poor performance on the minority classes, as we will see in the **Experimental Results** phase. To address this issue, assigning higher weights to the minority classes encourages the model to pay more attention to these classes during training. This helps achieve a more balanced representation and improves the model's ability to correctly classify samples from the minority classes. To implement this, we can manually compute the class weights and store them in a dictionary variable called `class_weight_dict`. The method that we will use is called *inverse class frequency*.

$$\text{Weight} = \frac{\max(\text{Frequency of any class})}{\text{Frequency of the class}}$$

The formula ensures that the weights are heavier on those classes that are less frequent. This gives more equality across the classes. The `class_weight_dict` dictionary will then be used during the training process to assign the computed weights to each class. By assigning higher weights to the minority classes, the model is encouraged to give equal importance to all classes, leading to better overall performance and mitigating the bias caused by class imbalance.

Figure 3: Classes' Weights

As it can be seen from the picture, those classes that were a minority now have an higher weight that will be taken into consideration during the training phase.

3 Theoretical Framework

3.1 *ConvNets*

Convolutional Neural Networks (ConvNets) are a class of Neural Networks that are mainly deployed for processing and analyzing images due to their ability to leverage the spatial characteristics of the data. Within their structure, ConvNets comprise different types of layers, namely Convolutional Layers, Pooling Layers, and Fully-Connected Layers, which work collaboratively to create an effective image analysis framework.

3.1.1 Convolutional Layers

Convolutional Layers play a crucial role in ConvNets by performing localized computations using `filters`. These filters are *convolved* with small receptive fields of the input data, calculating dot products and generating output volumes known as *feature maps*. Each filter captures specific patterns or features present in the input, enabling the network to learn relevant representations.

The convolutional layer receives an input volume, which is typically a 3D array representing an image. The input volume has a certain height, width, and depth (number of channels). For example, an RGB image has a depth of 3 (one channel for red, green, and blue). The convolution operation is the core operation of a convolutional layer. It involves applying multiple filters to the input volume. Each filter has a smaller size than the input volume, such as a 3x3 filter. The filter is **convolved** with the input volume by sliding it spatially across the input.

At each spatial position, the filter performs a dot product between its values and the corresponding input values it is currently overlaid on. This dot product is computed separately for each channel of the filter and input, and the results are summed. The resulting single scalar value represents the activation or response of that filter at that spatial position. The convolution operation generates multiple such activation values as the filter slides across the input volume. These values are combined to form an output volume called a **feature map**. The feature map retains the spatial dimensions (height and width) of the input but has a different depth (number of filters) based on the number of filters applied.

Each filter in the feature map captures a different pattern or feature from the input. After the convolution step, a bias term is added to each activation value in the feature map. This bias allows the ConvNet to learn the offset or baseline for each filter's response. Finally, an activation function is applied element-wise to the feature map. The activation function introduces non-linearity, enabling the ConvNet to model complex relationships and capture

non-linear patterns. The receptive field of a neuron in a convolutional layer refers to the region of the input volume that contributes to its activation. It corresponds to the spatial extent of the filter. As the filters are convolved across the input volume, they cover different receptive fields, allowing the ConvNet to capture local patterns and gradually learn higher-level representations.

3.1.2 Fully-Connected Layers

Fully-Connected Layers connect neurons to the preceding volume, enabling the network to make class predictions. In ConvNets, these layers are typically placed at the end of the architecture. Before passing through the fully-connected layers, the feature maps from the previous layers are flattened into a one-dimensional vector. Each neuron in the fully-connected layers corresponds to a specific class, and the output of these layers represents the class scores, indicating the likelihood of the input belonging to each class.

3.1.3 Loss Functions

During the training of *ConvNets*, the model's performance is evaluated using a loss function. Sparse categorical cross-entropy is a loss function commonly used in machine learning, particularly in multi-class classification tasks where the classes are mutually exclusive.

In sparse categorical cross-entropy, the predicted probability distribution and the target variable are compared to calculate the loss. The predicted probability distribution is obtained by applying a *softmax* activation function to the output layer of a neural network. The target variable is a vector of integers representing the true class labels.

The loss is calculated as the negative log-probability of the correct class label. It encourages the model to assign a high probability to the correct class and lower probabilities to the incorrect classes. The loss is defined as follows:

$$L = -\sum_{i=1}^{C} \mathbb{I}(y = i) \cdot \log(p_i)$$

where C is the total number of classes in the classification task, $\mathbb{I}$(y=i) is an indicator function that evaluates to 1 if the true class label y matches class i, and 0 otherwise. Essentially, it selects the predicted probability p_i, corresponding to the true class label. Finally, $\log(p_i)$ computes the natural logarithm of the predicted probability p_i for class i.

3.1.4 Optimizers

Optimizers are algorithms or methods used in training neural networks to adjust the model's parameters based on the gradients of the loss function. Their goal is to minimize the loss function and improve the model's performance. The choice of optimizer can significantly impact the convergence speed and the quality of the final model.

The **Adam** optimizer is a popular optimization algorithm used in deep learning, including ConvNets. It combines the advantages of Adaptive Gradient Algorithm (*AdaGrad*) and Root Mean Square Propagation (RMSProp). **Adam** maintains a learning rate for each parameter and calculates adaptive learning rates based on the historical gradients. It uses moving averages of past gradients to update the parameters during training, resulting in better optimization.

In the context of ConvNets, the Adam optimizer plays a crucial role in achieving better convergence and performance, especially when dealing with complex ConvNets and large-scale datasets. By efficiently updating the network's parameters, Adam helps the model learn relevant representations and improve its ability to analyze and extract meaningful features from images.

Optimizers, like Adam, are essential components in the training process of ConvNets and other deep learning models. Their proper selection and configuration can greatly impact the model's effectiveness and efficiency in capturing patterns and making accurate predictions.

3.2 Hyperparameter Tuning: *RandomSearch*

During the training of ConvNets, selecting appropriate *hyperparameters* is crucial for achieving optimal performance. Hyperparameter tuning is the process of finding the best combination of hyperparameters for a given algorithm. One popular method for hyperparameter tuning is **RandomSearch**. RandomSearch involves randomly sampling from a predefined search space of hyperparameters and evaluating the model's performance using cross-validation or a validation set. This process is repeated for a defined number of iterations, and the hyperparameter configuration that yields the best performance is selected. RandomSearch hyperparameter tuning can be applied to ConvNets

by exploring different hyperparameters such as learning rate, batch size, dropout rate, number of filters, filter sizes, and more. By randomly sampling these hyperparameters, it allows for a broad search across the hyperparameter space, providing a good chance of finding an optimal configuration.

3.3 Simple ConvNets vs. Deep ConvNets

One important distinction that needs to be taken into consideration is between deep and non-complex ConvNets, as these two model structures will be taken into consideration during the Model Deployment phase. When comparing complex ConvNets to non-complex ConvNets, the key distinction lies in the architectural complexity and computational requirements. Non-complex ConvNets have simpler designs within each layer, often with fewer layers, making them suitable for tasks with less complexity or limited computational resources. These networks are easier to train, have a smaller number of parameters, and can be trained using optimization algorithms such as the Adam optimizer.

4 Model Deployment

Two models will be deployed, a "simple" one and a "deep" one, to put into practice what has been discussed about during the **Theoretical Framework** part. Following, another attempt will be performed as to try to perform an hyper-parameter tuning on the best model between the two. Each model is trained for 10 epochs (this number has been chosen for hardware/computational reasons) by using ad Adam Optimizer. `early_stopping` is further implemented as to save up time in case the training phase wouldn't improve the validation accuracy within 3 epochs.

4.1 Simple Model

The first model to be assessed is going to be a simple one. This model present the following structure:

1. **Convolutional Layer 1**: This layer applies 16 filters of size (3, 3) to the input image. It uses the sigmoid activation function. The input shape is (HEIGHT, WIDTH, 3), where `HEIGHT = 64` and `WIDTH = 64`. The activation function is gonna be ReLU.
2. **Max Pooling Layer 1**: This layer performs max pooling with a pool size of (2, 2) to downsample the spatial dimensions of the previous layer's output.
3. **Convolutional Layer 2**: Similar to the first convolutional layer, this layer applies 32 filters of size (3, 3) to the input image. Sigmoid activation function.
4. **Max Pooling Layer 2**: Another max pooling layer with a pool size of (2, 2).
5. **Flatten Layer**: This layer flattens the output from the previous layer into a 1D vector, converting the multi-dimensional feature maps into a single long feature vector.
6. **Dense Layer 1**: A fully connected layer with 128 units and ReLU activation is used. It learns complex patterns and relationships from the flattened features.
7. **Dense Layer 2**: The final dense layer with `NUM_CLASSES` units (the number of classes in the classification problem) and softmax activation is applied to produce the probability distribution over the classes.

The model is then trained for 10 epochs (this number has been chosen for hardware/computational reasons). `early_stopping` is further implemented as to save up time in case the training phase wouldn't improve the validation accuracy. The Adam optimizer was utilized to train the model.

4.2 Deep Model

The second model deployed is a Deep Model that increases the complexity of the internal layers. The layers are the following:

1. **Rescaling Layer**: This layer rescales the input image pixels by dividing them by 255. It normalizes the pixel values to the range of [0, 1]. The input shape will then be of (64, 64, 3), representing the height, width, and color channels of the image. Rescaling to achieve faster convergence during training and improves the stability of the network. This normalization technique ensures that the pixel values are within a consistent and standardized range, making them more suitable for training the Convolutional Neural Network.

2. **Convolutional Layer 1**: Applies 32 filters of size (3, 3) to the input. RELu activation function.
3. **Max Pooling Layer 1**: Performs a downsampling operation using a pool size of (2, 2).
4. **Convolutional Layer 2**: Similar to the first convolutional layer, this layer applies 64 filters of size (3, 3) to the input, followed by the ReLU activation function.
5. **Max Pooling Layer 2**: Another max pooling layer with a pool size of (2, 2).
6. **Convolutional Layer 3**: This layer applies 128 filters of size (3, 3) to the input image. RELu activation function.
7. **Max Pooling Layer 3**: The third and final max pooling layer with a pool size of (2, 2) is applied.
8. **Flatten Layer**: Flatten layer, same as the one discussed earlier.
9. **Dense Layer 1**: A fully connected layer with 256 units and ReLU activation. It learns complex patterns and relationships from the flattened features.
10. **Dense Layer 2**: The final dense layer with `NUM_CLASSES` units (the number of classes in the classification problem) and softmax activation is used to produce the probability distribution over the classes.

Similarly, this model is also tuned for 10 epochs.

4.3 Deep Tuned Model

1. **Rescaling Layer**, same as deep model.
2. **Convolutional Layer 1**: filter parameter space is 32, 3x3 kernel size. RELu activation function.
3. **Max Pooling Layer 1**: pool size 2x2.
4. **Convolutional Layer 2**: filter parameter space is 32 or 64, 3x3 kernel size. RELu activation function.
5. **Max Pooling Layer 2**: pool size 2x2.
6. **Convolutional Layer 3**: filter parameter space 64, 96 or 128, 3x3 kernel size. RELu activation function.
7. **Max Pooling Layer 3**: pool size 2x2.
8. **Flatten Layer**: same as before.
9. **Dense Layer 1**: 64, 96 or 128 dense units.
10. **Dense Layer 2**: `NUM_CLASSES` units (the number of classes in the classification problem). Activation function softmax.

The RandomSearch algorithm, applied to the "Deep Tuned Model", aims to monitor the validation accuracy. It will perform a maximum of 8 trials to achieve convergence.

5 Experimental Results

5.1 Simple Model

The simple model fails to achieve an acceptable degree of accuracy in the multiclass classification task. The train loss reached accounts for 1.41, while the accuracy for almost 49%. Things do change in worse when we move onto the validation set and test set, where we record an accuracy level of 39% and 35% respectively. The loss function follows the trend of the accuracy level (downward trend). It seems like not implementing the class weights has produced poor learning performance. In the following graph we can take a look at the model history during the training phase.

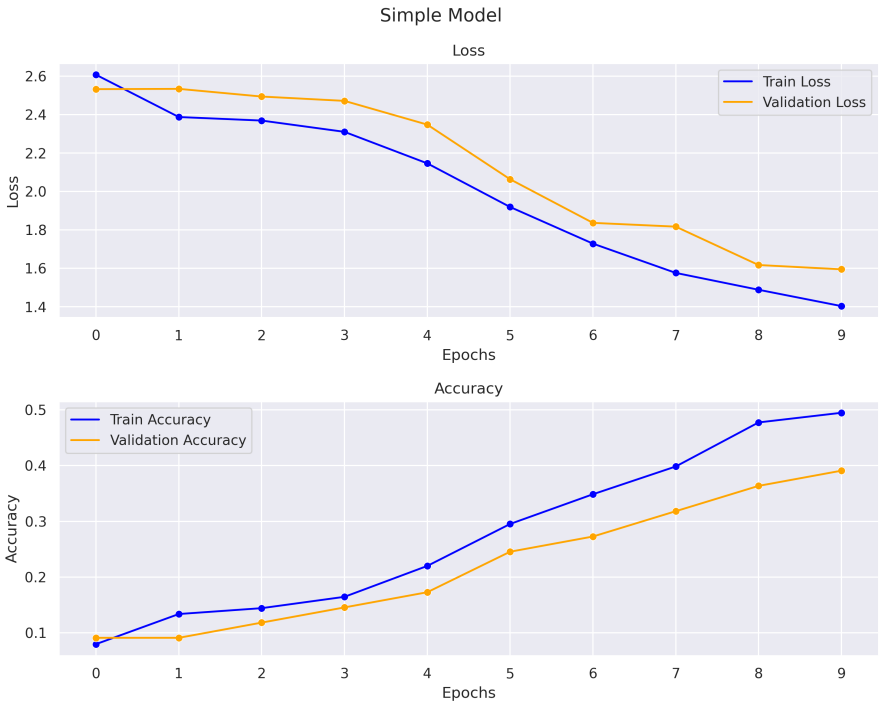

Figure 4: Simple Model's Training History

It is therefore necessary to opt for a more complex model such as the "deep" one, as well as implementing class weights during the training phase.

5.2 Deep Model

In the Model Deployment phase, a rescaling layer was added as first layer of the model. This rescaling layer has proven to be beneficial for the model's performance on each dataset provided. Additionally, increasing the number of filters in each layer has increased the model's capacity to better understand the features laying in the images.

During the training process, the training error steadily decreased over the course of 10 epochs, reaching a value of 0.20. The accuracy of the model on the training set peaked at 93%, demonstrating its ability to classify the leaf images accurately. Within the validation set, the results are pretty satisfying too: the loss function is below zero while the train accuracy reaches a solid 81%. Furthermore, in the testing phase, the model achieved an accuracy of 87% on the test set, which is a satisfactory percentage. The model also achieved a low value for the loss function, indicating that it was able to make accurate predictions on the test data.

Overall, the addition of the rescaling layer, the implementation of class weights and the increase in the number of filters in each layer has improved the model's performance, leading to higher accuracy and lower loss values on both the training and test sets. Here's a picture of the training history of the Deep Model.

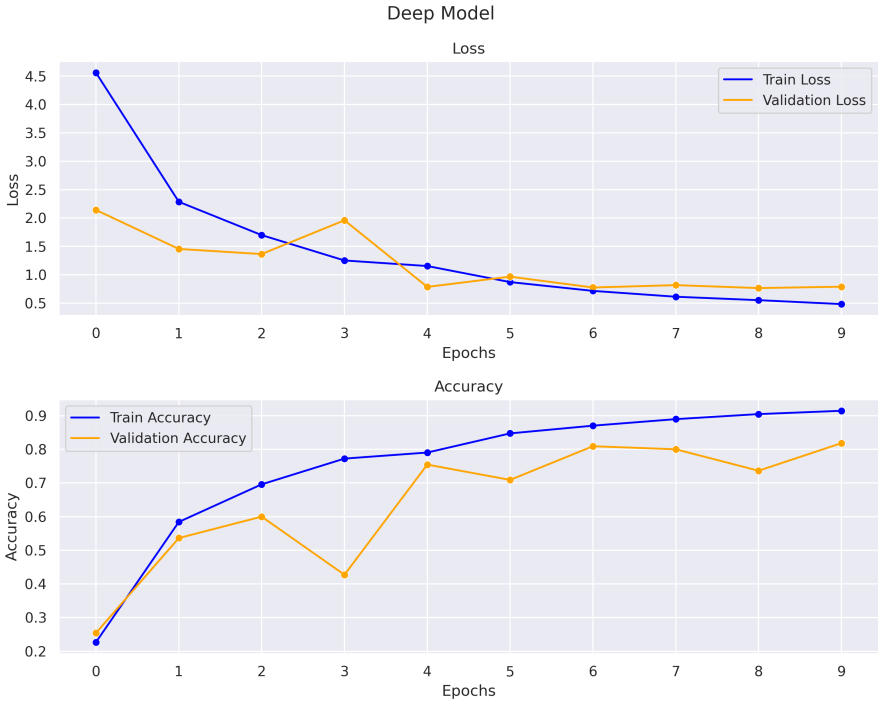

Figure 5: Deep Model's Training History

5.3 Hyper Parameter Tuning

Given the optimal results obtained in the Deep Model, we decided to tune the model for parameters in order to decrease the complexity of each layer and see whether similar results could be obtained in terms of accuracy and loss functions.

The *RandomSearch* algorithm outputs this Model structure as the one which achieves the highest validation accuracy amongst the parameter space selected during the Model Deployment phase.

The `model.summary()` function outputs the following:

- **Rescaling (Rescaling)**
- **conv2d (Conv2D):** 32 filters of size 3x3.
- **max_pooling2d (MaxPooling2D):** Down-sampling by taking the maximum value within a 2x2 window.
- **conv2d_1 (Conv2D):** 32 Filters of size 3x3.
- **max_pooling2d_1 (MaxPooling2D).** Down-sampling by taking the maximum value within a 2x2 window.
- **conv2d_2 (Conv2D):** 64 filters of size 3x3.
- **max_pooling2d_2 (MaxPooling2D):** Down-sampling by taking the maximum value within a 2x2 window.
- **flatten (Flatten):** Vector of length 2304.
- **dense (Dense):** Dense fully connected layer of 128 neurons.
- **dense_1 (Dense):** Final output layer of size 12.

As expected, the tuned model demonstrated acceptable performance both in-sample and out-of-sample. However, the train accuracy slightly decreased compared to the previous performance of the deep model. The trend remained consistent: during validation, the accuracy dropped by 10%. On the other hand, the test accuracy increased to 83%.

6 Conclusion

The results of the models can be summed up in the following table:

		Loss	Accuracy
SIMPLE	Train	1.41	49%
	Val	1.59	39%
	Test	1.42	35%
DEEP	Train	0,14	93%
	Val	0,55	81%
	Test	0,33	87%
DEEP TUNED	Train	0,02	90%
	Val	0,77	78%
	Test	0,47	83%

Figure 6: Model Results

If we prioritize out-of-sample performance, the "deep" model would be the best choice. However, it comes with higher computational costs and a heavier overall model, even though it could potentially achieve higher accuracy. On the other hand, if one prefers a slightly lighter model, the "deep tuned" model could be selected. Although it would be faster, it might not be as accurate as the deeper model.