UNIVERSITÀ DI PISA

SMART WASTE CLASSIFICATION

Computational intelligence and Deep learning project

A.Y. 2022/2023

Artificial Intelligence and Data Engineering
Matteo Mugnai, Filippo Puccini

1 SOMMARIO

2 INTRODUCTION

Waste management is a critical issue for modern society. The increasing amount of waste generated and the potential environmental impacts of improper disposal have made it necessary to adopt more efficient and sustainable waste management practices. One of the most effective approaches to reduce the amount of waste that ends up in landfills or oceans is through the implementation of selective waste collection systems.

Selective waste collection involves separating different types of waste materials, such as paper, plastic, glass, and metal, at the point of generation and collecting them separately for recycling or proper disposal. However, the success of selective waste collection depends on the ability to accurately identify and sort the different waste materials. This is where machine learning comes into play.

In the context of waste management, machine learning can be used to automatically classify waste materials based on images captured by cameras. By deploying cameras along the waste processing line (that monitor garbage flowing on a conveyor belt) and using machine learning algorithms to analyze the images in real-time, waste management facilities can identify and sort different types of waste more accurately and efficiently.

The benefits of automating waste classification using machine learning are numerous. First, it can increase the accuracy of waste sorting, reducing contamination and improving recycling rates. Second, it can reduce the need for manual sorting, which is labor-intensive and often prone to errors. Third, it can enable waste management facilities to process waste more quickly and efficiently, reducing costs and environmental impacts.

In conclusion, implementing machine learning-based waste classification systems can be a game-changer for waste management. By automating the waste sorting process, we can achieve higher recycling rates, reduce waste contamination, and improve the overall efficiency of waste management systems. This is the challenge that we are going to address in this project.

3 RELATED WORKS

Smart waste classification became popular few years ago. In fact, our dataset has been shared on Kaggle only two years ago. Different kind of studies have been proposed recently using different networks and techniques. Basically, two different kinds of problems have been deeply analysed: garbage recognition in nature (e.g., for plastic detection), brand recognition on wastes and garbage recognition in industry (e.g., wastes flowing on a conveyor belt). We decide to focus on the last problem because it's more useful, simple and feasible than the other. In fact, in a static context like a waste collection centre we don't have the problem of collecting a clear picture of an item because a camera is placed on a conveyor belt and take pictures from different angles. In the external environments this approach is not feasible for obvious reasons. Moreover, the background of an image can affect the correct classification while in our case background is always the same monochromatic one.

Let's discuss about some related works of past years.

One of the first analysis of such kind is in 2016 when Yang and Thung performed some experiments without obtaining good results. The SVM achieved better results than the CNN. It achieved a test accuracy of 63% using a 70/30 training/testing data split. The training error was 30%. The SVM is a relatively simpler algorithm than the CNN, which may attribute to its success in this task. The CNN seemed to not learn, as the test accuracy we achieved in the experiment described was only 22%.

In 2019 this paper was published in South Africa. They proposed an intelligent waste material classification system, which is developed by using the 50-layer residual net pre-train (ResNet-50) Convolutional Neural Network model which is a machine learning tool and serves as the extractor, and Support Vector Machine (SVM) which is used to classify the waste into different groups/types such as glass, metal, paper, and plastic etc. The proposed system is tested on the trash image dataset, which was developed by Gary Thung and Mindy Yang, and is able to achieve an accuracy of 87% on the dataset.

Again in 2019 at Stanford University this argument was tackled. They try to build a CNN from scratch and with the best model they reach stable test accuracies of 70% to 80%. They write: "The highest accuracy we achieved was 79.94% with Model 2 (SVM as last layer) using partial data augmentation. However, no matter what techniques we add on, we found it extremely hard to break through the 80% threshold with our scratch model. All the accuracies higher than 90% in the related works we read about were achieved by utilizing transfer learning techniques".

In 2020 at ITCC conference another study was published. In this paper, a method; based on deep learning and computer vision concepts, to classify wastes using their images into six different waste types (glass, metal, paper, plastic, cardboard and others) has been proposed. Multiple-layered Convolutional Neural Network (CNN) model, specifically the well-known Inception-v3 model has been used for classification of waste, with trained dataset obtained from online sources. High classification accuracy of 92.5% is achievable using the proposed method. It is envisaged that the proposed waste classification method would pave the way for the automation of waste segregation with reduced human involvement and therefore, helps with the waste recycling efforts.

Many other works are based on the classification between recyclable and organic wastes, but they are not interesting for our purposes.
What comes out from the state of the art is that the best way to solve this problem is the transfer learning approach. In fact, how will see in next sections the usage of pretrained CNN is very beneficial and it allows us to reach a good percentage of accuracy.

4 DATA PRE-PROCESSING

For this project we used a dataset found on Kaggle: Garbage Classification (12 classes) | Kaggle created 2 years ago.

The dataset contains 15,150 images from 12 different classes of household garbage: paper, cardboard, biological, metal, plastic, green-glass, brown-glass, white-glass, clothes, shoes, batteries, and trash. The total size is around 250 Mb.

Images stored in this dataset have been collected from different sources:

- The clothes category and 22 % of the shoes' category were obtained from the Clothing dataset https://www.kaggle.com/agrigorev/clothing-dataset-full.
- Around 29% of the other 9 classes combined was collected from the Garbage Classification dataset https://www.kaggle.com/asdasdasasdas/garbage-classification.
- All the rest of the images were obtained using Web Scrapping.

Pre-processing operations have been performed on a dedicated Colab notebook but first of all we edited manually the dataset by removing some classes. For “glass” we have the distinction in brown, white and green. We coalesce together these three classes under a unique “glass” folder. We merged “paper” and “cardboard” into a single “paper” class. The same process for “shoes” and “clothes”. We renamed the class “biological” into “organic”.

At the end of this process, we obtain 8 classes distributed in this way:

- Number of clothes images: 7302
- Number of glass images: 2011
- Number of organic images: 985
- Number of trash images: 697
- Number of plastic images: 865
- Number of paper images: 1941
- Number of metal images: 769
- Number of battery images: 945

Total number of images: 15515

Firstly, we reduce the numerosity of “clothes” class because there are too many samples, so we performed an under sampling of 75%.
Now we can split our dataset into training, validation and test using a 70/15/15 percentage.
Then we run some data balancing operations in order to have the same number of samples for all classes in the training set. Some classes (‘clothes’, ‘glass’, ‘paper’) have been under sampled and some other ('plastic', 'organic', 'metal', 'battery', 'trash') have been oversampled through data augmentation.
After this we store the custom dataset in separate folders, and we obtain the following sets:

- Training samples are: 7040.
- Validation samples are: 2327.
- Test samples are: 2328.

To conclude pre-processing phase, we also tried another approach: instead of balancing class samples at the same quantity with under sampling and data augmentation techniques, we kept class unbalanced with their original number of samples.

We need a way to tell the network that class are not equally balanced, so we adopted *"class_weight"* parameter in *keras.fit()* function. The *"class_weight"* parameter in the *"model.fit()"* method is used to handle the inequality in the distribution of classes in the training data. In some cases, there may be more instances of some classes than others in the training set, and this can lead to a model that pays more attention to classes with a higher number of instances than classes with a lower number of instances.

Using the *"class_weight"* parameter allows assigning different weights to classes during model training, so that the model considers the inequality in the distribution of classes and makes more accurate predictions for underrepresented classes. We decided to pass this parameter using a python dictionary where each class has its computed weight. Before doing this, we reduced the number of samples of "clothes" class because they are too many.

We compute weights in this way:

$$W_{class} = n_{classSamples} \div (n_{classes} * tot_{samples})$$

We obtained these results:

CLASS	SAMPLES	WEIGHT
BATTERY	945	1.547
CLOTHES	3473	0.421
GLASS	2011	0.727
METAL	769	1.901
ORGANIC	985	1.484
PAPER	1941	0.753
PLASTIC	865	1.690
TRASH	697	2.097

In chapter 6.6 we can see training results.

5 CNN FROM SCRATCH

In this section, we will build our neural network from scratch, experimenting with various possibilities and finally comparing the results. The models were constructed starting from simple networks and gradually adding complexity and depth to each model. We also aimed to minimize overfitting by introducing dropout layers and we also tested the models by adding Batch Normalization and Data Augmentation techniques. The parameters related to the input and batch size used for all models were as follows:

- **IMAGE_WIDTH** = 224
- **IMAGE_HEIGHT** = 224
- **BATCH_SIZE** = 128
- **NUM_CLASSES** = 8

For each model, we added a sequential layer as the first layer, which consists of two operations:

```
1. resize_and_rescale = tf.keras.Sequential([
2. layers.Resizing(IMAGE_HEIGHT, IMAGE_WIDTH),
3. layers.Rescaling(1./255)
4. ])
```

This layer sequentially applies two operations to the input. First, it resizes the input images to the specified IMAGE_HEIGHT and IMAGE_WIDTH dimensions. Then, it performs rescaling by dividing the pixel values by 255 to normalize them between 0 and 1. This pre-processing step helps prepare the input data for the subsequent layers of the neural network.
In addition to these models, which were built using standard methods of neural network architecture, we also experimented and tested three different types of architecture and training methodology:

- **Depthwise Separable Convolution**
- **Inception Layer**
- **Residual Module**

For these three architectures, we performed hyperparameter tuning to obtain the optimal models and achieve the best possible results.
In the following paragraphs, we will analyze the behavior of each model, providing an overview of their structure and the number of parameters. In the subsequent analysis, we will examine the performance of each model, including its structure and the number of parameters.

5.1 MODEL 1: STANDARD CNN

The first model is a standard CNN consisting of three convolutional layers followed by a final Dense layer with activation. The network has moderate complexity, as indicated by the limited number of parameters.
Before running the initial test, we performed a sanity check on our dataset. This involved training the model on a small batch of data to see if it would overfit. The purpose of the sanity check was to ensure that our model was capable of learning from the data without memorizing it too closely. By training the model on a limited amount of data, we could observe its performance and evaluate whether it was able to generalize well to new examples.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
sequential_1 (Sequential)	(None, 224, 224, 3)	0
conv2d (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_1 (Conv2D)	(None, 56, 56, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 28, 28, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73856
flatten (Flatten)	(None, 100352)	0
dense (Dense)	(None, 8)	802824

Total params: 896,072
Trainable params: 896,072
Non-trainable params: 0

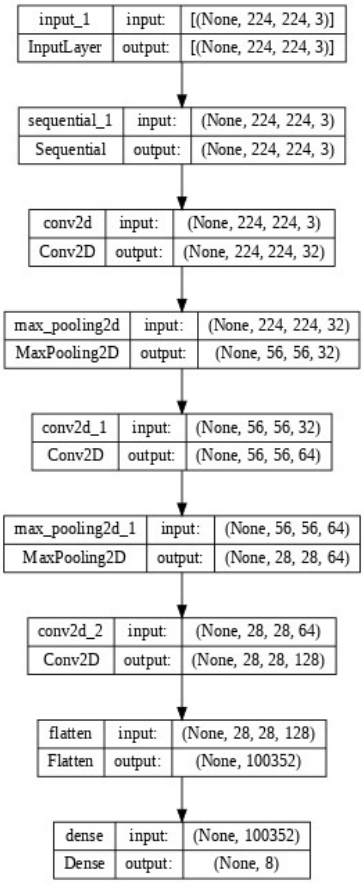


Figure 1 Model 1 Structure and parameters

Below, we can see the training results:

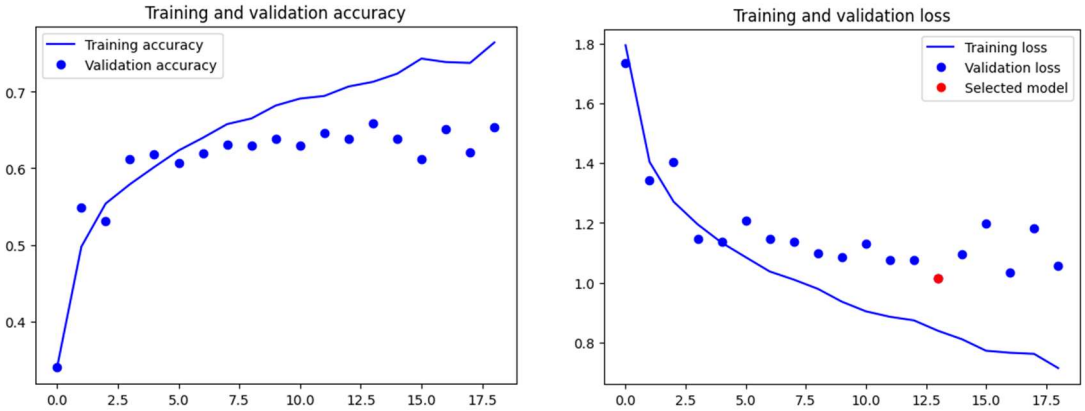


Figure 2 Training results Model 1

The results indicate that the model, trained for 19 epochs, starts overfitting early on, around epoch 5. After this point, there are only marginal improvements in terms of validation accuracy and loss. This behavior may be due to the model's simplicity, which is unable to capture more complex relationships, resulting in a maximum validation accuracy of 0.658.
When evaluating the model on the test set, we obtained results that align with those of the validation and training sets. The test accuracy of the model was found to be 0.673, with a

corresponding test loss of 0.965. These results suggest that the model's performance on unseen data is consistent with its performance during training and validation.

5.2 Model 2: One dense layer and four Conv2D layers

In this model, we attempted to address the issue of overfitting by introducing a Dense layer, allowing the model to better capture complex relationships. By incorporating a Dense layer, the model gains the ability to learn richer and more complex representations of the data, thereby enhancing its flexibility in modelling the relationships between input features and the desired

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 224, 224, 3)]	0
sequential_1 (Sequential)	(None, 224, 224, 3)	0
conv2d_7 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_6 (MaxPooling 2D)	(None, 56, 56, 32)	0
conv2d_8 (Conv2D)	(None, 56, 56, 64)	18496
max_pooling2d_7 (MaxPooling 2D)	(None, 28, 28, 64)	0
conv2d_9 (Conv2D)	(None, 28, 28, 128)	73856
max_pooling2d_8 (MaxPooling 2D)	(None, 14, 14, 128)	0
conv2d_10 (Conv2D)	(None, 14, 14, 256)	295168
max_pooling2d_9 (MaxPooling 2D)	(None, 7, 7, 256)	0
flatten_2 (Flatten)	(None, 12544)	0
dense_4 (Dense)	(None, 32)	401440
dense_5 (Dense)	(None, 8)	264

Total params: 790,120
Trainable params: 790,120
Non-trainable params: 0

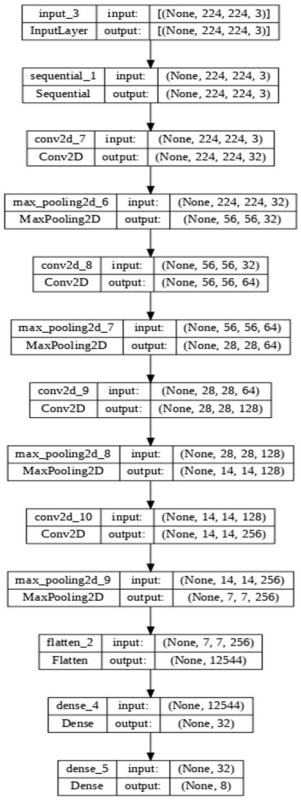


Figure 3 Model 2 Structure and parameters

output.

However, despite our efforts to enhance the model's ability to capture complex relationships, the improvement in predictive power compared to the previous model is limited. The validation

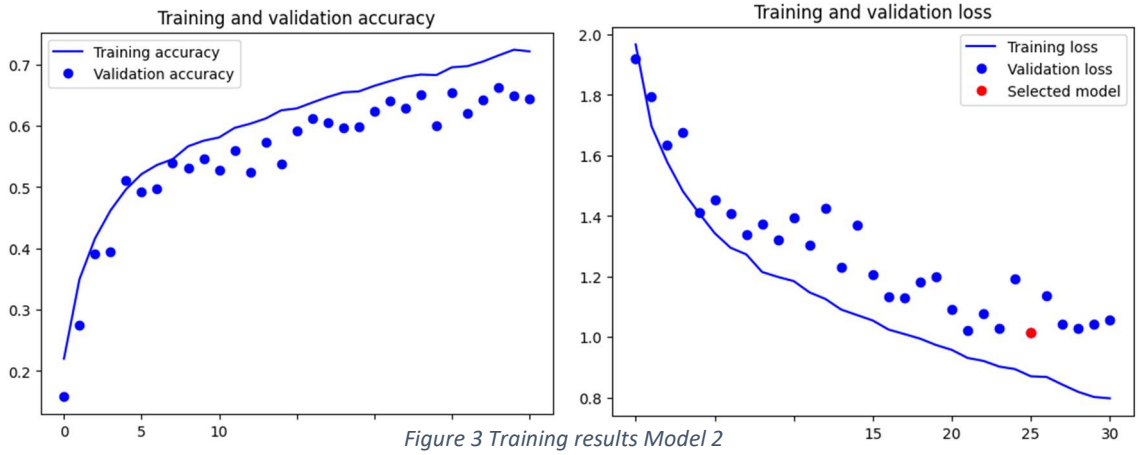


Figure 3 Training results Model 2

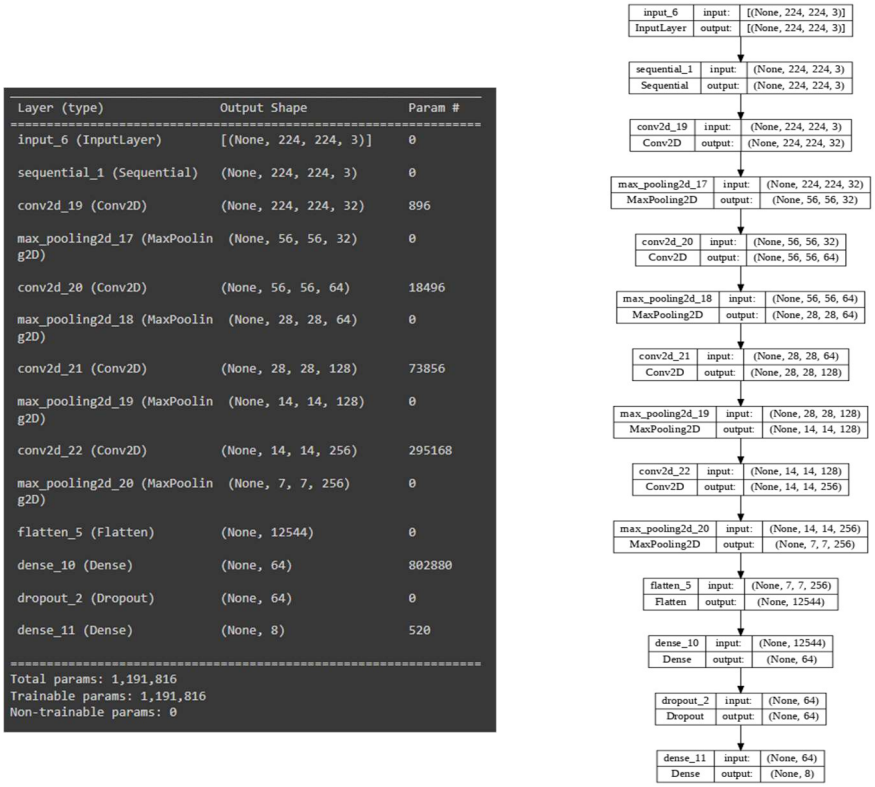
accuracy reaches a maximum of 0.653 after 31 epochs, suggesting that the model struggles to generalize well to unseen data. Although there is a reduction in the presence of overfitting compared to the previous model, it remains persistent to some degree. This indicates that the additional layer helps in mitigating overfitting to some extent, allowing the model to generalize slightly better.

The test set exhibits similar performance to the previous model, with a test accuracy of 0.670 and a test loss of 0.960.

Overall, despite efforts to increase the model's complexity, the results indicate that a more sophisticated architecture or additional techniques might be necessary to further improve its predictive power and overcome the challenge of overfitting.

5.3 Model 3: One Dropout and Dense Layer

By adding a dropout layer and increasing the number of units in the dense layer, our goal was to alleviate overfitting and increase the complexity of the network. As a result, the model now has a higher number of parameters.



Layer (type)	Output Shape	Param #
input_6 (InputLayer)	[(None, 224, 224, 3)]	0
sequential_1 (Sequential)	(None, 224, 224, 3)	0
conv2d_19 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_17 (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_20 (Conv2D)	(None, 56, 56, 64)	18496
max_pooling2d_18 (MaxPooling2D)	(None, 28, 28, 64)	0
conv2d_21 (Conv2D)	(None, 28, 28, 128)	73856
max_pooling2d_19 (MaxPooling2D)	(None, 14, 14, 128)	0
conv2d_22 (Conv2D)	(None, 14, 14, 256)	295168
max_pooling2d_20 (MaxPooling2D)	(None, 7, 7, 256)	0
flatten_5 (Flatten)	(None, 12544)	0
dense_10 (Dense)	(None, 64)	802880
dropout_2 (Dropout)	(None, 64)	0
dense_11 (Dense)	(None, 8)	520

Total params: 1,191,816
Trainable params: 1,191,816
Non-trainable params: 0

Figure 4 Model 3 structure and parameters

The obtained results show improvement compared to the previous models. The validation accuracy is 0.694 with a validation loss of 0.8798. However, it is important to note that the model exhibits underfitting, suggesting that it may not be complex enough to capture the underlying patterns in the data effectively. One potential solution to address underfitting is to further increase the size and complexity of the network.

Turning our attention to the test set, we observe an increase in performance. The model achieves a test accuracy of 0.704 and a test loss of 0.843. These results demonstrate that the model's predictive power has improved, indicating its ability to generalize well to unseen data.

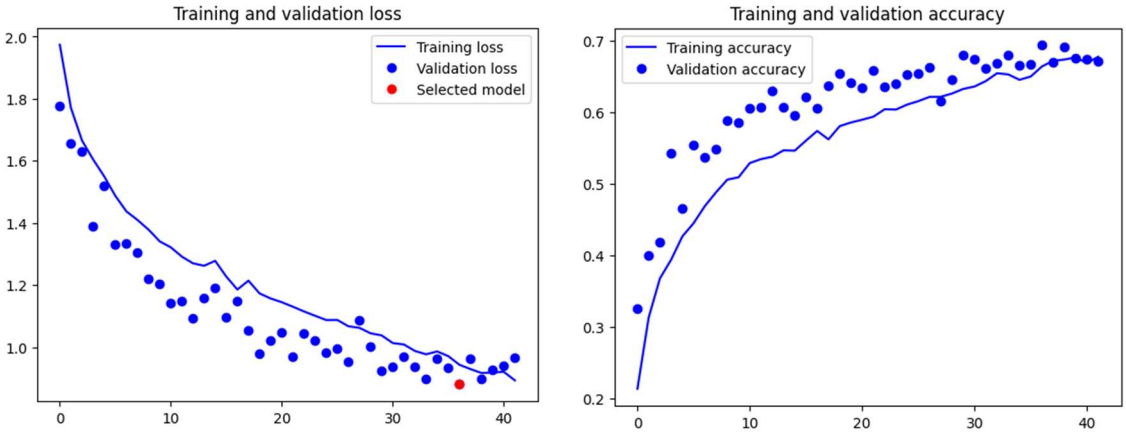


Figure 5 Training results Model 3

In summary, the addition of a dropout layer and the increase in the number of units in the dense layer have led to improved results in terms of accuracy and loss. However, the model still exhibits underfitting, suggesting the need for further adjustments to enhance its complexity and capture more intricate relationships within the data. The performance on the test set validates the advancements made, showcasing improved accuracy and a lower loss compared to previous models.

5.4 Model 4: Two Dropout and Dense layer

In this model, we introduced an additional dropout layer and a dense layer with 256 units to significantly amplify the complexity and predictive power. These modifications effectively reduced overfitting compared to earlier models and successfully addressed the underfitting issue observed in the previous model. Furthermore, the CNN demonstrates a substantial improvement in its predictive power.

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 224, 224, 3)]	0
sequential_1 (Sequential)	(None, 224, 224, 3)	0
conv2d_23 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_21 (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_24 (Conv2D)	(None, 56, 56, 64)	18496
max_pooling2d_22 (MaxPooling2D)	(None, 28, 28, 64)	0
conv2d_25 (Conv2D)	(None, 28, 28, 128)	73856
max_pooling2d_23 (MaxPooling2D)	(None, 14, 14, 128)	0
conv2d_26 (Conv2D)	(None, 14, 14, 256)	295168
max_pooling2d_24 (MaxPooling2D)	(None, 7, 7, 256)	0
flatten_6 (Flatten)	(None, 12544)	0
dropout_3 (Dropout)	(None, 12544)	0
dense_12 (Dense)	(None, 128)	1605760
dense_13 (Dense)	(None, 256)	33024
dropout_4 (Dropout)	(None, 256)	0
dense_14 (Dense)	(None, 8)	2056

Total params: 2,029,256
Trainable params: 2,029,256
Non-trainable params: 0

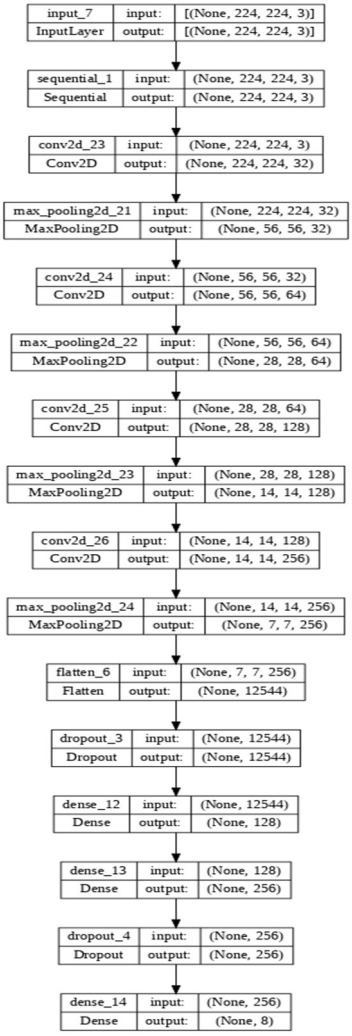


Figure 6 Model 4 structure and parameters

The results obtained from the training set showcase the exceptional performance of this model, surpassing all previous networks. With a higher validation accuracy (0.7340) and lower validation loss(0.8222) compared to its predecessors, this model demonstrates superior predictive capabilities. Moreover, the model exhibits stability throughout the training process, with a smooth and rapid convergence.

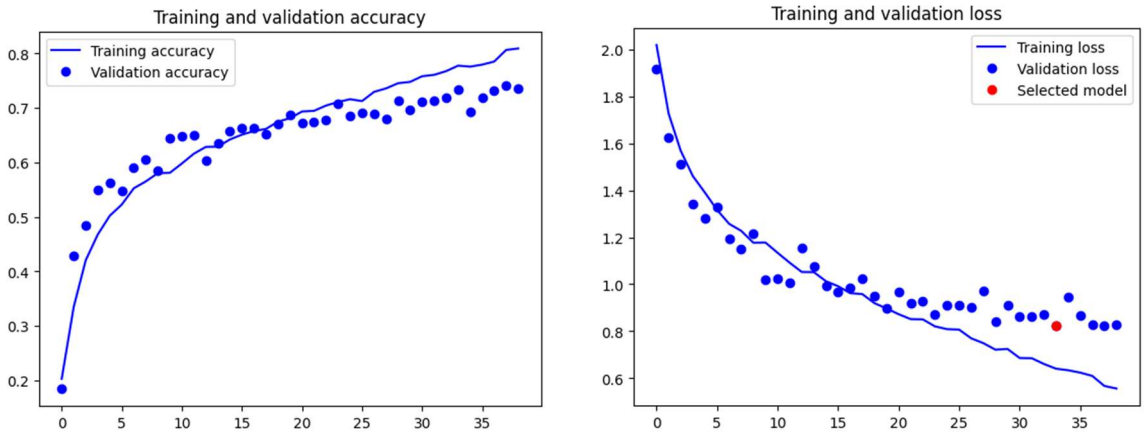


Figure 7 Training result Model 4

The test set results further validate the model's effectiveness, as evidenced by a significantly improved test accuracy of 0.753 and a reduced test loss of 0.765. These metrics indicate the model's ability to generalize well to unseen data and make accurate predictions.

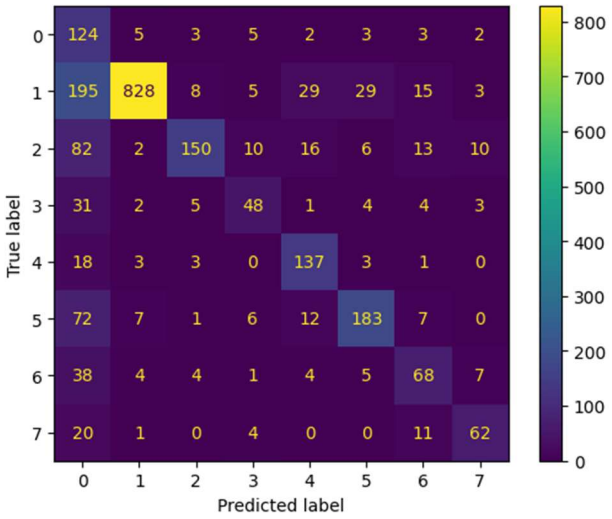


Figure 8 Confusion matrix Model 4

Given the remarkable performance of this model, we will use it as a reference point for the implementation of the next two models. Its success serves as a benchmark to guide the development and refinement of subsequent architectures.

5.5 Model 5: Batch Normalization

As mentioned earlier, this model (structure and parameters can be seen in the next page) is an extension of the previous one, with the addition of a Batch Normalization layer before the Activation layer. Batch Normalization is a technique commonly used in neural networks to improve training stability and performance. It normalizes the inputs within each mini batch during training, reducing the internal covariate shift and enabling smoother gradient flow.

The results demonstrate an increase in predictive power with the introduction of Batch Normalization (validation accuracy = 0.761, validation loss = 0.8644). However, it is important to

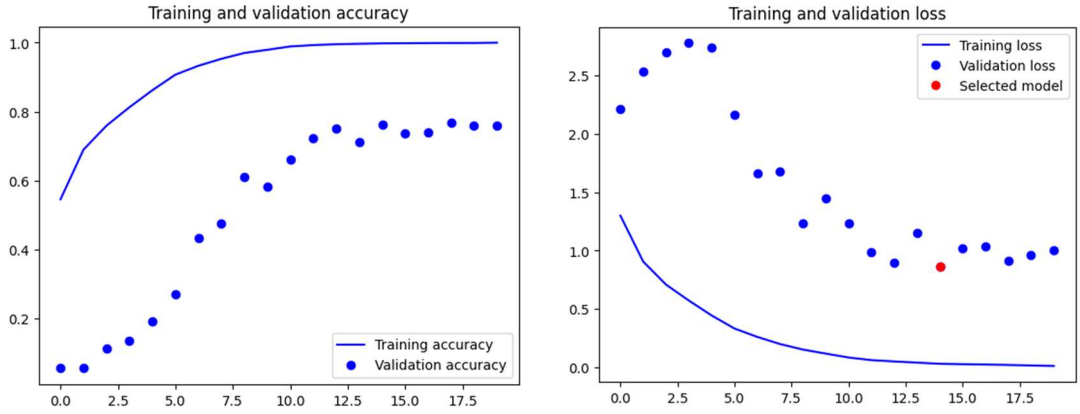


Figure 9 Training results Model 5

note that the model exhibits significant overfitting. Despite the improvement in performance, the extent of overfitting observed indicates that the use of this technique may not be justified in our case.

Layer (type)	Output Shape	Param #
input_13 (InputLayer)	[(None, 224, 224, 3)]	0
sequential (Sequential)	(None, 224, 224, 3)	0
conv2d_48 (Conv2D)	(None, 224, 224, 32)	896
batch_normalization_48 (BatchNormalization)	(None, 224, 224, 32)	128
activation_48 (Activation)	(None, 224, 224, 32)	0
max_pooling2d_39 (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_49 (Conv2D)	(None, 56, 56, 64)	18496
batch_normalization_49 (BatchNormalization)	(None, 56, 56, 64)	256
activation_49 (Activation)	(None, 56, 56, 64)	0
max_pooling2d_40 (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_50 (Conv2D)	(None, 14, 14, 128)	73856
batch_normalization_50 (BatchNormalization)	(None, 14, 14, 128)	512
activation_50 (Activation)	(None, 14, 14, 128)	0
max_pooling2d_41 (MaxPooling2D)	(None, 7, 7, 128)	0
conv2d_51 (Conv2D)	(None, 7, 7, 256)	295168
batch_normalization_51 (BatchNormalization)	(None, 7, 7, 256)	1024
activation_51 (Activation)	(None, 7, 7, 256)	0
flatten_12 (Flatten)	(None, 12544)	0
dense_36 (Dense)	(None, 128)	1605760
dense_37 (Dense)	(None, 256)	33024
dense_38 (Dense)	(None, 8)	2056

Total params: 2,031,176
Trainable params: 2,030,216
Non-trainable params: 960

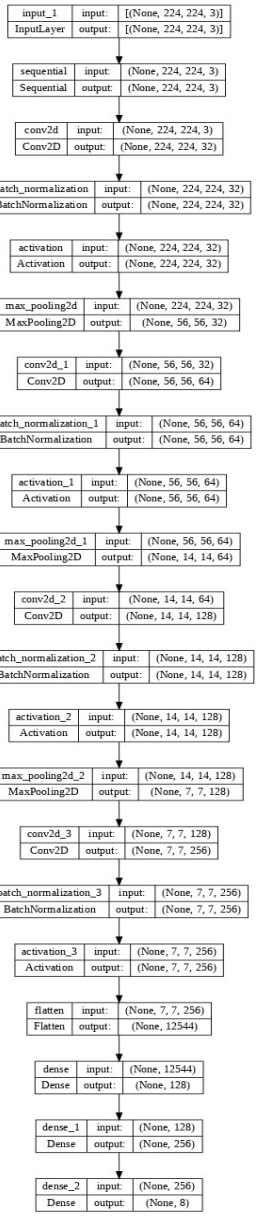


Figure 10 Model 5 structure and parameters

The performance on the test set (Test accuracy: 0.781, test loss: 0.785) shows improvement compared to the previous models. This suggests that the model with Batch Normalization was able to generalize well to our test set, but it is possible that the model's generalization performance may be different when applied to external test sets that differ from our current dataset. While the model has shown improved performance on our test set, its ability to generalize to completely new and unseen data remains uncertain.

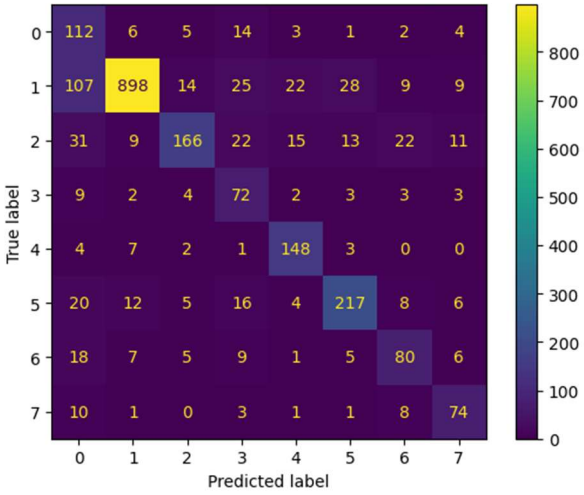


Figure 11 Confusion matrix model 5

5.6 Model 6: Data Augmentation

In this last model, we introduced an initial layer of data augmentation to model 4. Data augmentation is a technique where random modifications are applied to the images during training to reduce overfitting and expand the base dataset. Data augmentation allows for the generation of additional training examples by applying various random transformations to the images. In our case, we applied the following transformation:

- **Random flip**: applies horizontal flipping to a random 50% of the images.
- **Random rotation**: rotates the input images by a random value in the range[–10%, +10%] (fraction of full circle [-36°, 36°])
- **Random zoom**: zooms in or out of the image by a random factor in the range [-20%, +20%]

By applying these random modifications, the dataset is effectively expanded, providing the model with a more diverse set of training examples. This helps prevent overfitting by introducing variability and reducing the model's reliance on specific image characteristics. The results on validation are worse compared to the previous models, indicating that this technique is not suitable for our use case. We reach at best (epoch 21) a validation accuracy of 0.657, with a

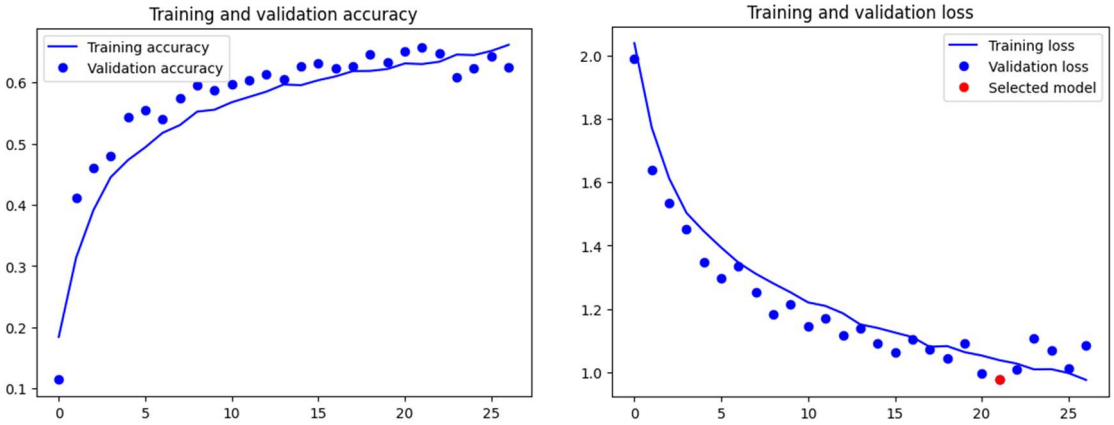


Figure 12 Training results on Model 6

validation loss of 0.9785. This probably happen because the modified images might have introduced noise or altered important features, leading to a decrease in performance.

Layer (type)	Output Shape	Param #
input_6 (InputLayer)	[(None, 224, 224, 3)]	0
sequential_1 (Sequential)	(None, 224, 224, 3)	0
sequential (Sequential)	(None, 224, 224, 3)	0
conv2d_19 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_18 (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_20 (Conv2D)	(None, 56, 56, 64)	18496
max_pooling2d_19 (MaxPooling2D)	(None, 28, 28, 64)	0
conv2d_21 (Conv2D)	(None, 28, 28, 128)	73856
max_pooling2d_20 (MaxPooling2D)	(None, 14, 14, 128)	0
conv2d_22 (Conv2D)	(None, 14, 14, 256)	295168
max_pooling2d_21 (MaxPooling2D)	(None, 7, 7, 256)	0
flatten_5 (Flatten)	(None, 12544)	0
dropout_5 (Dropout)	(None, 12544)	0
dense_11 (Dense)	(None, 128)	1605760
dense_12 (Dense)	(None, 256)	33024
dropout_6 (Dropout)	(None, 256)	0
dense_13 (Dense)	(None, 8)	2056

Total params: 2,029,256
Trainable params: 2,029,256
Non-trainable params: 0

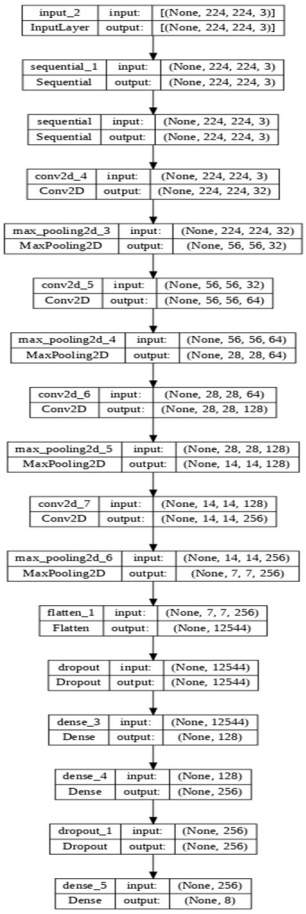


Figure 13 Structure and parameter Model 6

On test set we reach a value of accuracy equal to 0.672, with loss equal to 0.946, worse than previous models.

5.7 DEPTHWISE SEPARABLE CONVOLUTION

After conducting these tests just discussed, using standard methods to define our models, we have decided to implement three different training techniques or types of architectures to evaluate their performance, predictive capabilities, and computational costs.
The first one is the **Depthwise Separable Convolution**, foundation for the Xception network, that is a convolutional technique used in convolutional neural networks to reduce the number of parameters of the model and improve computational performance. In the next paragraph, we will explain how this technique works and the benefits it can bring.

5.7.1 How it works

While standard convolution performs the channel wise and spatial-wise computation in one step, depthwise separable convolution splits the computation into two steps: depthwise convolution applies a single convolutional filter per each input channel and pointwise convolution is used to create a linear combination of the output of the depthwise convolution. So, this method deals

with not only the spatial dimensions but also the depth dimension, which refers to the number of channels in an image.
An input image may have 3 channels representing RGB colours. Each channel can be thought of as a particular interpretation of the image. For example, the "red" channel interprets the "redness" of each pixel, the "blue" channel interprets the "blueness" of each pixel, and the "green" channel interprets the "greenness" of each pixel. An image with 64 channels has 64 different interpretations of that image. Starting from the dimension of the input channel, we can split the convolution operation in two steps instead of just one, performing first the depthwise convolution:

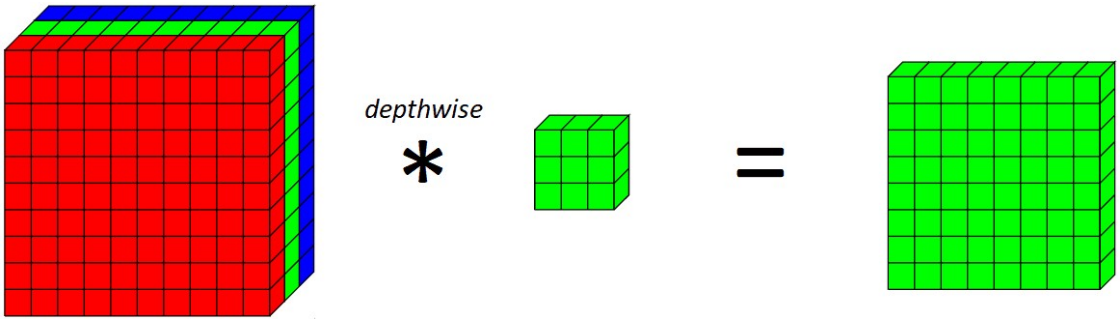


Figure 14 Depthwise convolution along one dimension

As already said, this operation must be applied to all dimensions of the image (e.g., height, width, and depth).

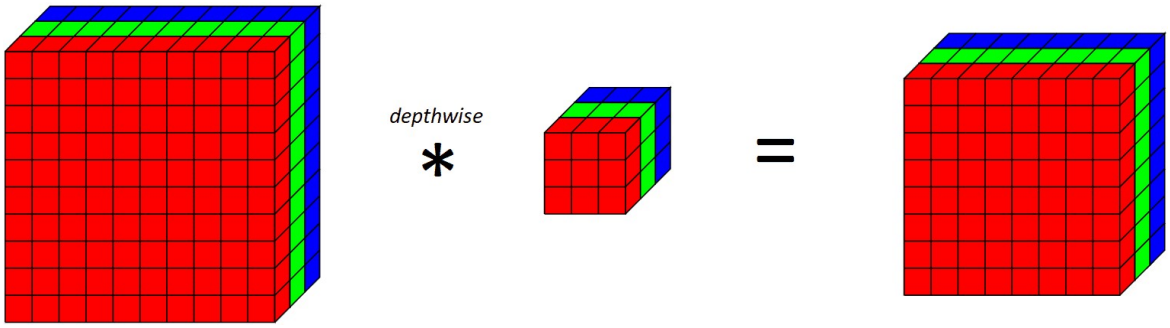


Figure 15 Depthwise convolution along three dimensions

Once the convolution operation has been performed for all dimensions of the image, a pointwise convolution can be performed to aggregate the results into a single output. The pointwise convolution is so named because it uses a 1x1 kernel, or a kernel that iterates through every single point. This kernel has a depth of however many channels the input image has; this involves computing the element-wise sum of the outputs for each channel of the image, resulting in a single output value.

Depthwise separable convolution offers several advantages over traditional convolution methods:

- **Reduced computational cost**: By separating the convolution into two steps - a depthwise convolution and a pointwise convolution - the model requires fewer computations. For example, Let's say we wanted to apply 64 convolutional filters to our RGB image to have 64

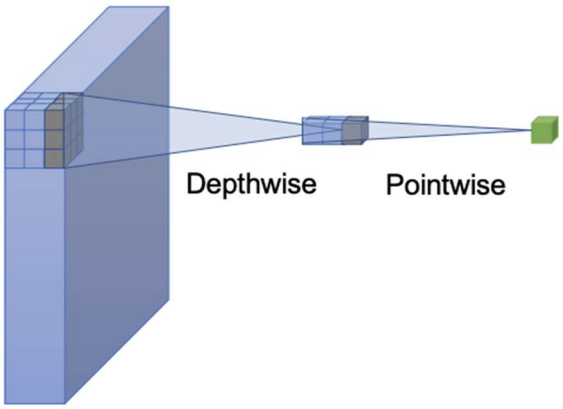


Figure 16 Pointwise convolution

channels in our output. Number of parameters in normal convolutional layer (including bias term) is **3 x 3 x 3 x 64 + 64 = 1792**. On the other hand, using a depthwise separable convolutional layer would only have **(3 x 3 x 1 x 3 + 3) + (1 x 1 x 3 x 64 + 64) = 286** parameters, which is a significant reduction, with depthwise separable convolutions having less than 6 times the parameters of the normal convolution.

- **Decreased memory usage**: With fewer parameters, depthwise separable convolution consumes less memory. This is particularly beneficial for models with limited memory resources or when running on devices with constraints. It allows for larger models or the ability to allocate more memory for other components of the network.
- **Improved generalization**: Depthwise separable convolution tends to have better generalization capabilities. By learning spatial features independently in the depthwise convolution and then combining them in the pointwise convolution, the model can capture more diverse and meaningful patterns. This enhances its ability to generalize well to unseen data and improves its robustness against variations and noise.

However, it is important to consider that this technique may not have the same accuracy as standard convolution. This is because dividing the convolution into multiple steps can lead the network to struggle in modelling complex relationships between different input channels.

5.7.2 Implementation

For the implementation, we took Model 4 and replaced the standard convolutional layers with layers that apply depthwise convolution using the SeparableConv2D() method provided by Keras. This technique significantly reduces the complexity of the network. While Model 4 had around 2

million parameters, utilizing depthwise separable convolution resulted in 1,686,467 parameters. Thus, the model is computationally much lighter compared to the original one.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
sequential (Sequential)	(None, 224, 224, 3)	0
separable_conv2d (SeparableConv2D)	(None, 224, 224, 32)	155
max_pooling2d (MaxPooling2D)	(None, 56, 56, 32)	0
separable_conv2d_1 (SeparableConv2D)	(None, 56, 56, 64)	2400
max_pooling2d_1 (MaxPooling2D)	(None, 28, 28, 64)	0
separable_conv2d_2 (SeparableConv2D)	(None, 28, 28, 128)	8896
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
separable_conv2d_3 (SeparableConv2D)	(None, 14, 14, 256)	34176
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 256)	0
flatten (Flatten)	(None, 12544)	0
dropout (Dropout)	(None, 12544)	0
dense (Dense)	(None, 128)	1605760
dense_1 (Dense)	(None, 256)	33024
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 8)	2056

Total params: 1,686,467
Trainable params: 1,686,467
Non-trainable params: 0

Figure 17 Model with depthwise separable convolution

The results on the validation (validation accuracy: 0.631, validation loss: 1.0164) and test sets (test accuracy: 0.651, test loss: 1.025) demonstrate, as expected, that the performance has degraded compared to Model 4. As explained earlier, this type of convolution is less powerful than the standard convolution and is more effective in much larger and more complex models.

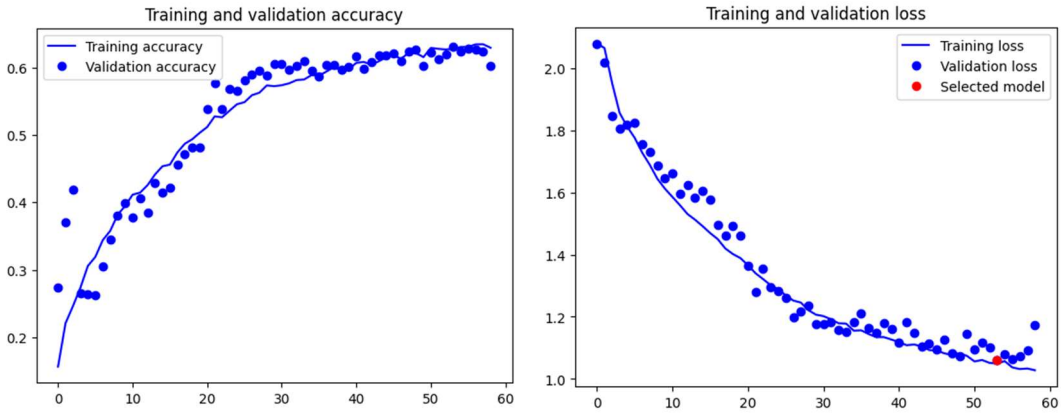


Figure 18 Training result model with depthwise separable convolution

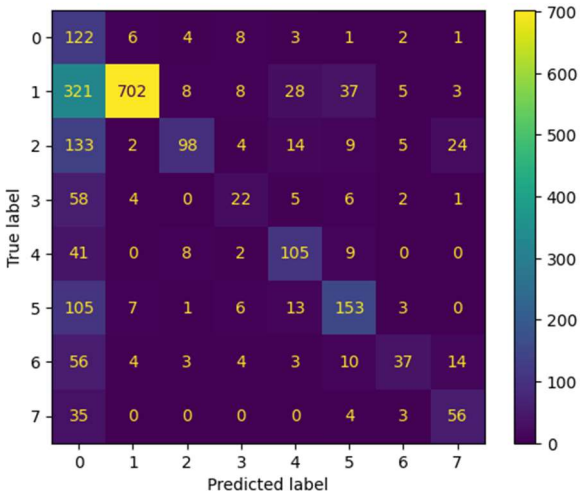


Figure 19 Confusion matrix model with depthwise separable convolution

5.7.3 Deeper network

To improve the performance of the previous model, we considered increasing its complexity by adjusting the "depth multiplier" parameter of the SeparableConv2D method, that controls the number of output channels in the depthwise convolution. It basically determines how many of these output feature maps are produced for each input channel.

```
x = layers.SeparableConv2D(filters=32, padding='same', kernel_size=3, depth_multiplier=2, activation="relu")(x)
```

Figure 20 Depth multiplier parameter in the SeparableConv method

For example, if the depth multiplier is set to 2, each input channel will generate two output feature maps instead of one. This effectively increases the number of output channels, allowing the model to capture more diverse and complex patterns from the input data.

By slightly increasing the number of parameters in the network to 1,736,532, we have improved the performance on both the validation set (validation accuracy: 0.661, validation loss: 0.9774) and the test set (test accuracy: 0.677, test loss: 0.932).

5.7.4 Hyperparameter tuning

In this section, we performed hyperparameter tuning using Keras Tuner. We defined ranges of values for certain parameters such as learning rate, units in Dense layers, dropout rate, and activation functions. The goal was to find the best possible combination of these parameters, building upon the previous model, to achieve the best validation accuracy.

```
activation_hp = hp.Choice('activation_function', values=['relu', 'elu', 'gelu'])
hp_units1 = hp.Int('units1', min_value=32, max_value=512, step=32)
hp_units2 = hp.Int('units2', min_value=32, max_value=512, step=32)
hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
dropout_rate_hp1 = hp.Float('dropout_rate1', min_value = 0, max_value = 0.5)
dropout_rate_hp2 = hp.Float('dropout_rate2', min_value = 0, max_value = 0.5)
```

Figure 21 Parameters tuned.

Below, you can see the performance of the best trial during the HP tuning.

```
Objective(name="val_accuracy", direction="max")

Trial 0025 summary
Hyperparameters:
activation_function: relu
units1: 384
units2: 32
learning_rate: 0.001
dropout_rate1: 0.27979051211721023
dropout_rate2: 0.03270814857365706
tuner/epochs: 25
tuner/initial_epoch: 9
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0022
Score: 0.725397527217865
```

Figure 22 Best trial for validation accuracy

Finally, we trained the model using the tuned parameters. The results are better compared to the initial model on the validation set, as we have a loss of 0.8619 and an accuracy of 0.7075. On the test set, we have a loss of 0.8619 and an accuracy of 0.7074.

5.8 Inception Layer

The second architecture we want to exploit is the inception layer. An inception network is a deep neural network with an architectural design that consists of repeating components referred to as Inception modules. The modules were designed to solve the problem of computational expense, as well as overfitting, among other issues. The solution, in short, is to take multiple kernel filter sizes within the CNN, and rather than stacking them sequentially, ordering them to operate on the same level.

5.8.1 How it works

Inception Modules are incorporated into convolutional neural networks (CNNs) as a way of reducing computational expense. As a neural net deals with a vast array of images, with wide variation in the featured image content, also known as the salient parts, they need to be designed appropriately. The most simplified version of an inception module works by performing a convolution on an input with not one, but three different sizes of filters (1x1, 3x3, 5x5). Also, max pooling is performed. Then, the resulting outputs are concatenated and sent to the next layer. By structuring the CNN to perform its convolutions on the same level, the network gets progressively wider, not deeper.
This is the structure of a basic Inception Module:

- Input layer
- 1x1 convolution layer
- 3x3 convolution layer
- 5x5 convolution layer
- Max pooling layer
- Concatenation layer

Below you can see the architecture of a Naive Inception Module:

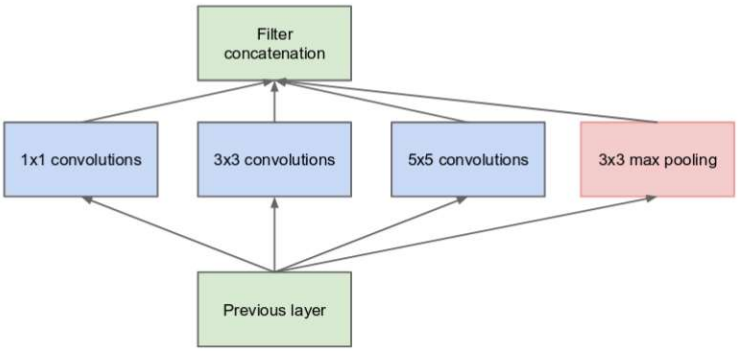


Figure 23 Inception module, naive version

To make the process even less computationally expensive, the neural network can be designed to add an extra 1x1 convolution before the 3x3 and 5x5 layers. By doing so, the number of input channels is limited and 1x1 convolutions are far cheaper than 5x5 convolutions. It is important to note, however, that the 1x1 convolution is added after the max-pooling layer, rather than before. This is the complete Inception module, with the addition of 1x1 convolutional layers:

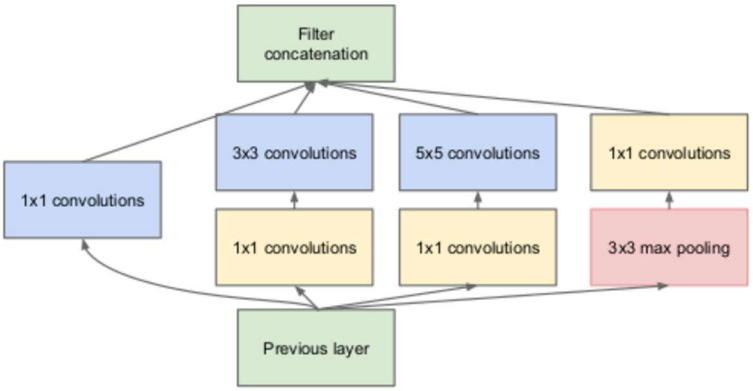


Figure 24 Inception module, complete version

In this way, we decrease by far the computational cost within the Inception Module, reducing depth in input and the number of multiplication operations. Now we will show the implementation of these two architectures.

5.8.2 Naïve inception module

We performed two different tests: first, we implemented the naive inception module, and then the complete inception module. The implementation of the first one involved constructing the naive inception module within a single method, which can be seen below; the method essentially applies three different convolution operations (as explained earlier) to the 'layer_in' and then returns the concatenation of these layers.

```
1. def naive_inception_module(layer_in, filter1, filter2, filter3):
2.   conv1 = layers.Conv2D(filter1, (1,1), padding='same', activation='relu')(layer_in)
3.   conv3 = layers.Conv2D(filter2, (3,3), padding='same', activation='relu')(layer_in)
4.   conv5 = layers.Conv2D(filter3, (5,5), padding='same', activation='relu')(layer_in)
```

```
5.  pool = layers.MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
6.  layer_out = layers.concatenate([conv1, conv3, conv5, pool], axis=-1)
7.  return layer_out
```

The model resulting quite complex respect to the previous ones, so we add a value of 'stride' equal to 4 in the first convolutional layer (the input to the inception block), to reduce the dimension of the network.

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3       0
                             )]

sequential (Sequential)      (None, 224, 224, 3)       0

conv2d_8 (Conv2D)            (None, 56, 56, 32)        896

max_pooling2d_4 (MaxPooling2D)  (None, 14, 14, 32)     0

conv2d_9 (Conv2D)            (None, 14, 14, 64)        2112

conv2d_10 (Conv2D)           (None, 14, 14, 128)       36992

conv2d_11 (Conv2D)           (None, 14, 14, 32)        25632

max_pooling2d_5 (MaxPooling2D)  (None, 14, 14, 32)     0

concatenate_2 (Concatenate)  (None, 14, 14, 256)       0



max_pooling2d_6 (MaxPooling2D)  (None, 7, 7, 256)      0

conv2d_12 (Conv2D)           (None, 7, 7, 128)         32896

conv2d_13 (Conv2D)           (None, 7, 7, 256)         590080

conv2d_14 (Conv2D)           (None, 7, 7, 64)          409664

max_pooling2d_7 (MaxPooling2D)  (None, 7, 7, 256)      0

concatenate_3 (Concatenate)  (None, 7, 7, 704)         0



conv2d_15 (Conv2D)           (None, 7, 7, 128)         811136

flatten_1 (Flatten)          (None, 6272)              0

dense_2 (Dense)              (None, 128)               802944

dropout_1 (Dropout)          (None, 128)               0

dense_3 (Dense)              (None, 8)                 1032

=================================================================
Total params: 2,713,384
Trainable params: 2,713,384
Non-trainable params: 0
_________________________________________________________________
```

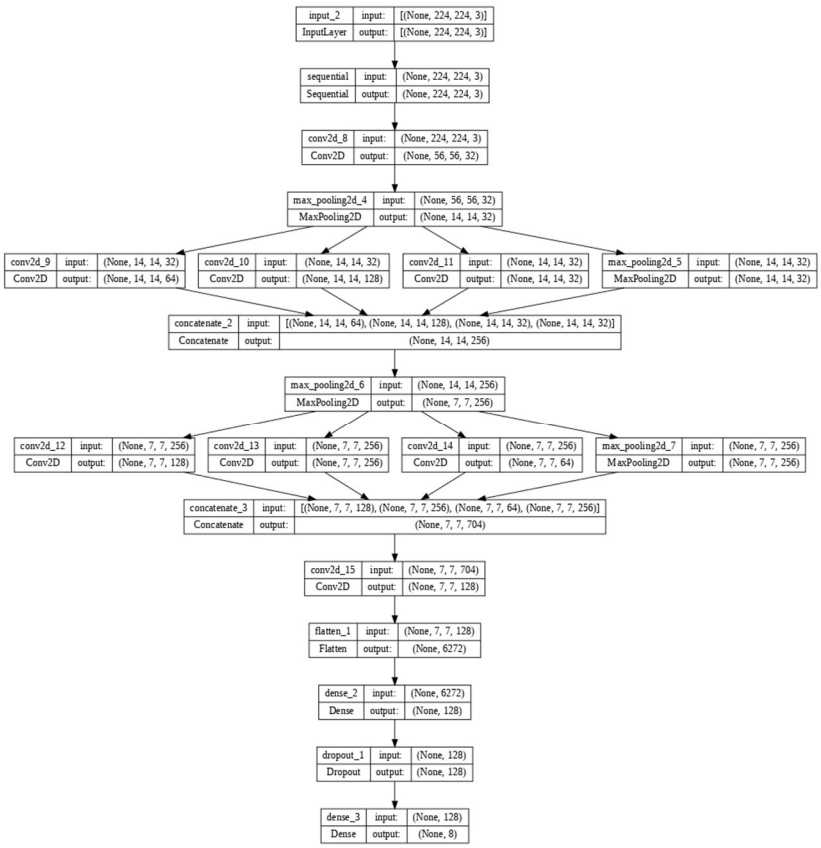


Figure 25 Parameters and structure of the model based on naive inception module

The results on both the training set and validation set demonstrate good performance, comparable to or slightly better than those observed in Model 4. The highest validation accuracy achieved is val_accuracy: 0.7520, with val_loss: 0.6964. However, the results from various iterations have some noise with fluctuations and inconsistencies throughout the training process. There is minimal overfitting observed, primarily in the final epochs.

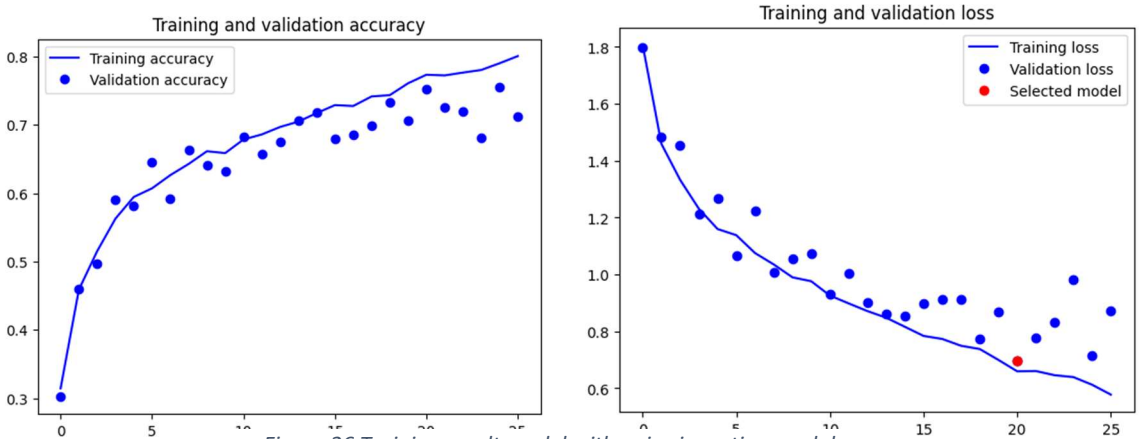


Figure 26 Training result model with naive inception module

The results on the test set are the best achieved among all the models trained from scratch. We achieved a test accuracy of 0.773 and a test loss of 0.655.

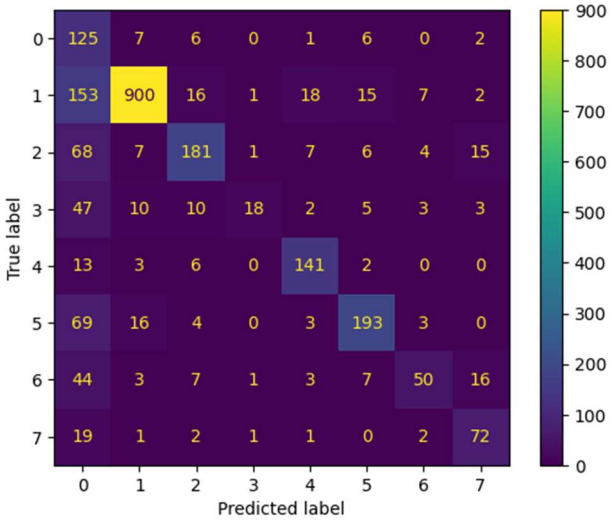


Figure 27 Confusion matrix model with naive inception module

5.8.3 Complete inception module

The second test we conducted involved a model based on the complete Inception module. We defined a method that implements the block and added a 1x1 kernel to the naïve Inception module components, namely the 3x3 and 5x5 convolutional layers. As explained earlier, the purpose of incorporating the 1x1 convolutional layers was to reduce the dimensionality of the input before applying the 3x3 and 5x5 convolutional layers. This approach aims to decrease the number of input channels and improve computational efficiency by utilizing more complex convolutional filters.

The method can be seen below:

```
1. def complete_inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
2.   conv1 = layers.Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
3.
4.   conv3 = layers.Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
5.   conv3 = layers.Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
6.
7.   conv5 = layers.Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
8.   conv5 = layers.Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
9.
10.  pool = layers.MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
11.  pool = layers.Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
12.
13.  layer_out = layers.concatenate([conv1, conv3, conv5, pool], axis=-1)
14.  return layer_out
```

The presented model demonstrates the reduction in network complexity achieved by introducing 1x1 kernels. We transitioned from a network with 2,713,384 parameters to one with 1,877,112 parameters. Consequently, we obtained a much less complex model compared to the previous one, while retaining the same base architecture.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3       0
                             )]
sequential (Sequential)      (None, 224, 224, 3)       0
conv2d (Conv2D)              (None, 56, 56, 32)        896
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
conv2d_2 (Conv2D)            (None, 14, 14, 96)        3168
conv2d_4 (Conv2D)            (None, 14, 14, 16)        528
max_pooling2d_1 (MaxPooling2D) (None, 14, 14, 32)      0
conv2d_1 (Conv2D)            (None, 14, 14, 64)        2112
conv2d_3 (Conv2D)            (None, 14, 14, 128)       110720
conv2d_5 (Conv2D)            (None, 14, 14, 32)        12832
conv2d_6 (Conv2D)            (None, 14, 14, 32)        1056
concatenate (Concatenate)    (None, 14, 14, 256)       0
max_pooling2d_2 (MaxPooling2D) (None, 7, 7, 256)       0
conv2d_8 (Conv2D)            (None, 7, 7, 128)         32896
conv2d_10 (Conv2D)           (None, 7, 7, 32)          8224
max_pooling2d_3 (MaxPooling2D) (None, 7, 7, 256)       0
conv2d_7 (Conv2D)            (None, 7, 7, 128)         32896
conv2d_9 (Conv2D)            (None, 7, 7, 192)         221376
conv2d_11 (Conv2D)           (None, 7, 7, 96)          76896
conv2d_12 (Conv2D)           (None, 7, 7, 64)          16448
concatenate_1 (Concatenate)  (None, 7, 7, 480)         0
conv2d_13 (Conv2D)           (None, 7, 7, 128)         553088
flatten (Flatten)            (None, 6272)              0
dense (Dense)                (None, 128)               802944
dropout (Dropout)            (None, 128)               0
dense_1 (Dense)              (None, 8)                 1032
=================================================================
Total params: 1,877,112
Trainable params: 1,877,112
Non-trainable params: 0
```

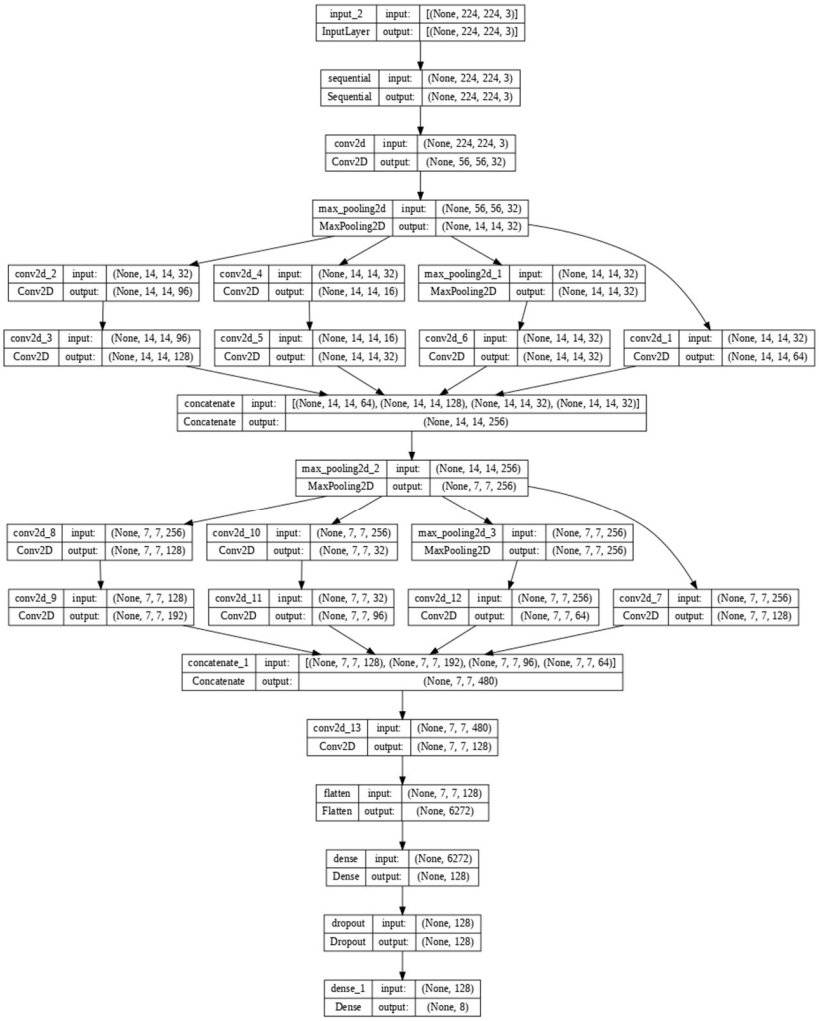


Figure 28 Parameters and structure model with complete inception module

The results demonstrate performance comparable to the previous model: in the best case, we achieved a val_loss of 0.7599 and a val_accuracy of 0.7323. The model exhibits a slight overfitting tendency and, like the previous model, displays some noise during the training process. One possible solution could be the addition of Batch Normalization layers to improve regularization throughout the training. Overall, the performance is satisfactory, considering the reduced complexity of the model compared to its predecessor

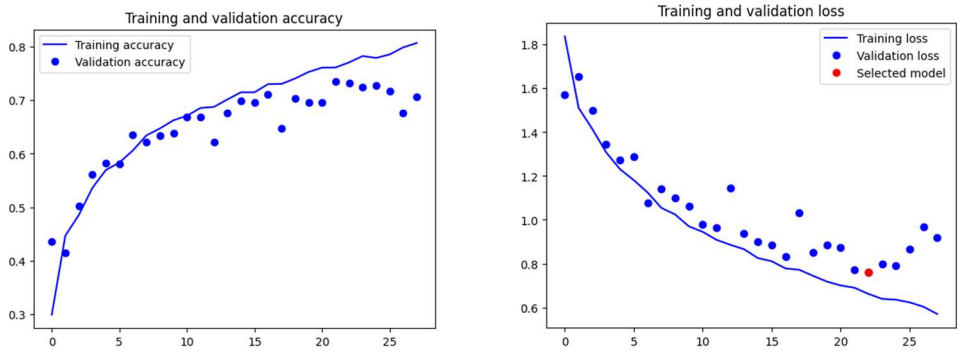


Figure 29 Training result on model with complete inception module

The results on the test set are promising: Test accuracy: 0.766, test loss: 0.677. Although slightly lower than the performance of the previous model, they are still considered satisfactory.

5.8.4 Hyperparameter tuning

We performed hyperparameter tuning on the model based on complete inception module, to maximize the validation accuracy. Parameters considered are learning rate, dropout rate, activation function and units of dense layers.

```
Search space summary
Default search space size: 4
activation_function (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'elu', 'gelu'], 'ordered': False}
units (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'ordered': True}
dropout_rate (Float)
{'default': 0.0, 'conditions': [], 'min_value': 0.0, 'max_value': 0.5, 'step': None, 'sampling': 'linear'}
```

Figure 30 Parameters tuned

Below, you can see the performance of the best trial during the HP tuning, that are much better than the model unoptimized.

```
Objective(name="val_accuracy", direction="max")

Trial 0024 summary
Hyperparameters:
activation_function: gelu
units: 320
learning_rate: 0.001
dropout_rate: 0.24713996898052154
tuner/epochs: 25
tuner/initial_epoch: 9
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0019
Score: 0.7730984091758728
```

Figure 31 Best trial in validation accuracy

Lastly, we trained the model with optimized parameters, and the results on the test set align with those obtained on the validation set, yielding a test loss of 0.733 and a test accuracy of 0.771.

5.9 RESIDUAL MODULE

For our final test, we opted to delve into training a neural network using residual modules, drawing inspiration from the ResNet architecture. Residual modules, which incorporate residual learning, provide a solution to the vanishing gradient problem, and enhance the network's capacity to capture complex patterns. We implemented the residual block and integrated it into our custom model to assess its performance and effectiveness.

5.9.1 How it works

Residual modules are based on the introduction of a shortcut connections that allow information to bypass certain layers in the network. Instead of solely relying on the output of the previous layer, residual modules incorporate the original input data as well. By adding the input data to the output of a layer, the module enables the network to learn residual functions that focus on the differences between the input and output. This approach facilitates the learning process,

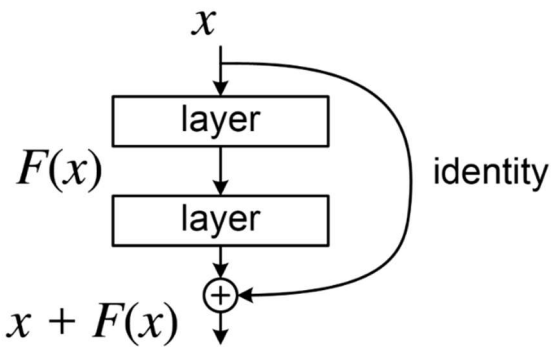


Figure 32 Residual block

especially when dealing with deep networks, by mitigating the vanishing gradient problem. Residual modules have proven effective in improving gradient flow, enhancing feature extraction, and enabling better model optimization, ultimately leading to improved performance in various tasks.

5.9.2 Residual module implementation

Below, you can observe the method implementing the residual block. It comprises two convolutional layers applied to the input layer, with their output being summed with the input layer itself. This mechanism enables the preservation of information across various layers. In cases where the input layer's depth differs from the number of filters provided as an argument, a 1x1 convolution is employed to adjust the dimensions.

```
1. def residual_module(layer_in, filters):
2.  merge_input = layer_in
3.
4.  if layer_in.shape[-1] != filters:
5.   merge_input = layers.Conv2D(filters, (1,1), padding='same', activation='relu')(layer_in)
6.
7.  conv1 = layers.Conv2D(filters, (3,3), padding='same', activation='relu')(layer_in)
8.  conv2 = layers.Conv2D(filters, (3,3), padding='same', activation='linear')(conv1)
9.
10.  layer_out = layers.add([conv2, merge_input])
11.  layer_out = layers.Activation('relu')(layer_out)
12.  return layer_out
```

```
Model: "model_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_18 (InputLayer)       [(None, 224, 224, 3       0
                             )]

 sequential_3 (Sequential)   (None, 224, 224, 3)       0

 conv2d_193 (Conv2D)         (None, 224, 224, 32       896
                             )

 conv2d_194 (Conv2D)         (None, 224, 224, 32       9248
                             )

 conv2d_192 (Conv2D)         (None, 224, 224, 32       128
                             )

 add_64 (Add)                (None, 224, 224, 32       0
                             )

 activation_128 (Activation) (None, 224, 224, 32       0
                             )

 max_pooling2d_64 (MaxPooling2D  (None, 56, 56, 32)    0
 )

 conv2d_196 (Conv2D)         (None, 56, 56, 64)        18496

 conv2d_197 (Conv2D)         (None, 56, 56, 64)        36928

 conv2d_195 (Conv2D)         (None, 56, 56, 64)        2112

 add_65 (Add)                (None, 56, 56, 64)        0

 activation_129 (Activation) (None, 56, 56, 64)        0

 max_pooling2d_65 (MaxPooling2D  (None, 28, 28, 64)    0
 )

 conv2d_199 (Conv2D)         (None, 28, 28, 128)       73856

 conv2d_200 (Conv2D)         (None, 28, 28, 128)       147584

 conv2d_198 (Conv2D)         (None, 28, 28, 128)       8320

 add_66 (Add)                (None, 28, 28, 128)       0

 activation_130 (Activation) (None, 28, 28, 128)       0

 max_pooling2d_66 (MaxPooling2D  (None, 14, 14, 128)   0
 )

 conv2d_202 (Conv2D)         (None, 14, 14, 256)       295168

 conv2d_203 (Conv2D)         (None, 14, 14, 256)       590080

 conv2d_201 (Conv2D)         (None, 14, 14, 256)       33024

 add_67 (Add)                (None, 14, 14, 256)       0

 activation_131 (Activation) (None, 14, 14, 256)       0

 max_pooling2d_67 (MaxPooling2D  (None, 7, 7, 256)     0
 )

 dropout_32 (Dropout)        (None, 7, 7, 256)         0

 flatten_16 (Flatten)        (None, 12544)             0

 dense_48 (Dense)            (None, 128)               1605760

 dense_49 (Dense)            (None, 256)               33024

 dropout_33 (Dropout)        (None, 256)               0

 dense_50 (Dense)            (None, 8)                 2056

=================================================================
Total params: 2,856,680
Trainable params: 2,856,680
Non-trainable params: 0
_________________________________________________________________
```

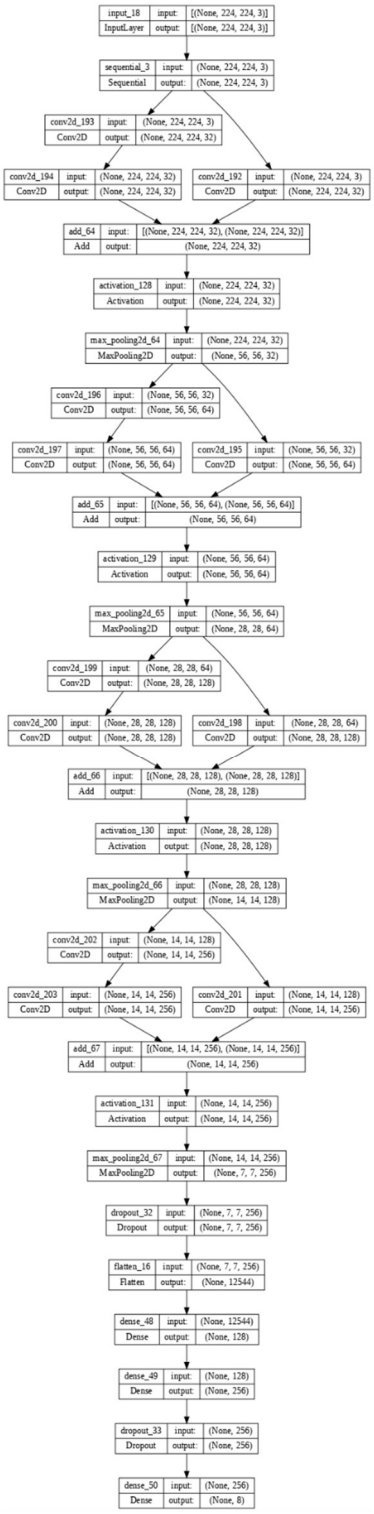


Figure 33 Parameters and structure of model with residual module

The results on the training set demonstrate that the model attains a val_loss of 0.7678 and a val_accuracy of 0.749, which are good compared to the previous models. The training process is also quite regular, with only a slight overfitting in last epochs.

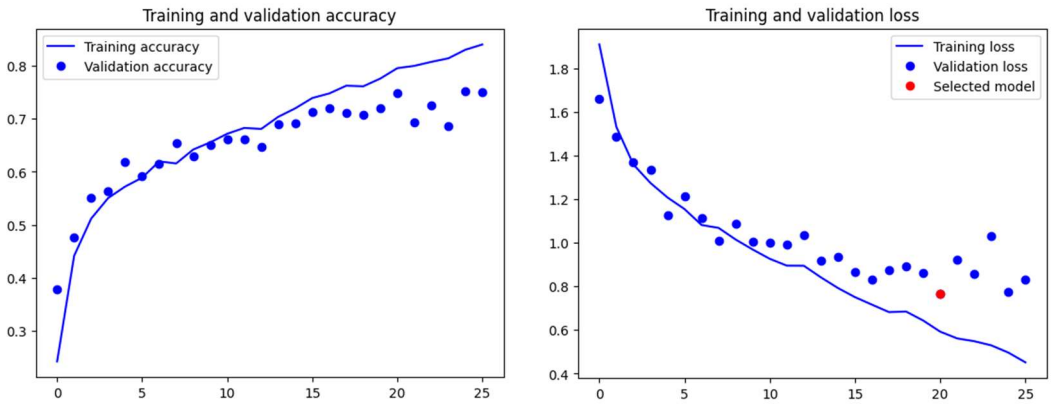


Figure 34 Training result on model based on residual module

The performance on the test set is similar, with a Test accuracy of 0.757 and a test loss of 0.701.

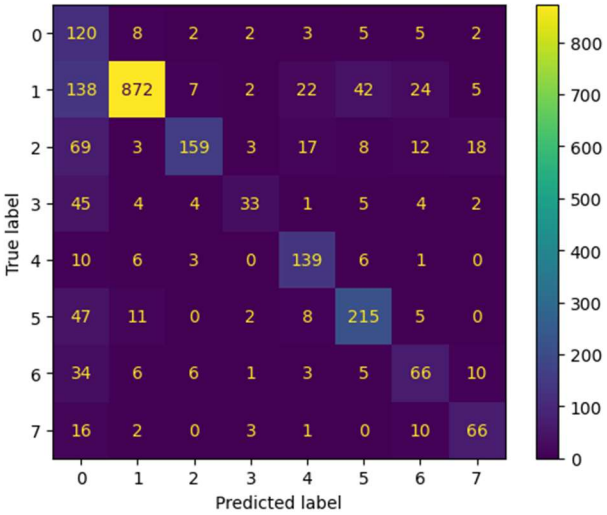


Figure 35 Confusion matrix model based on residual module

5.9.3 Residual module deeper

As a final test, we aimed to increase the complexity of the previous network to enhance its predictive power. To achieve this, we added additional residual modules to the network architecture. As a result, the size of the network increased from 2,856,680 parameters to 3,299,560 parameters. By incorporating more residual modules, we aimed to provide the network with a greater capacity to learn complex representations and capture intricate patterns in the data. This increase in complexity allows the network to potentially improve its predictive performance and ability to generalize to unseen data (parameters and structure can be seen in Colab notebook).

The increase in model complexity not improved very much the results of the training process. We observed a val_loss of 0.7973 and a val_accuracy of 0.7434, which are in line with the previous model. However, it's important to note that towards the end of the training, the model exhibited a noticeable overfitting tendency.

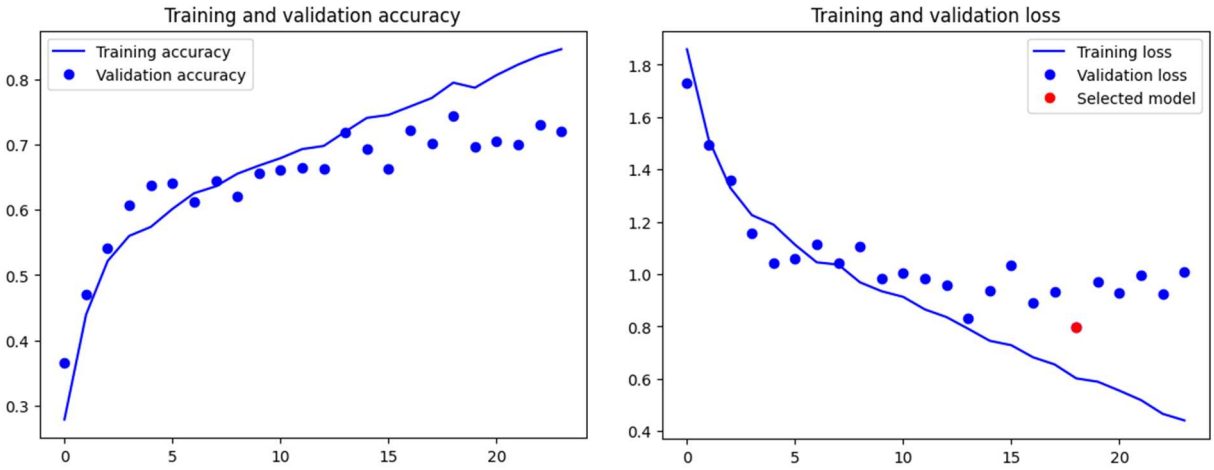


Figure 36 Training results on deeper model based on residual module

The results on the test set are good, with a test accuracy of 0.758 and a test loss of 0.733.
In conclusion, the residual module has proven to be a valuable addition to the model, enabling it to achieve notable improvements in accuracy and loss metrics. Further fine-tuning and regularization strategies can be explored to strike a balance between model complexity and generalization, ultimately optimizing the performance of the residual module in future experiments.

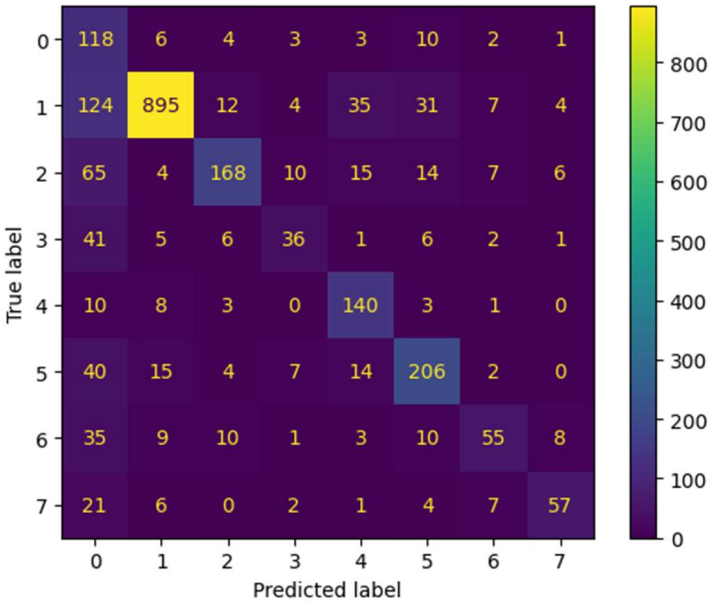


Figure 37 Confusion matrix deeper model based on residual module

5.9.4 Hyperparameter tuning

We performed hyperparameter tuning to optimize the model and achieve its maximum predictive power. Below, you can find the parameters we considered for optimization:

```
Search space summary
Default search space size: 6
activation_function (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'elu', 'gelu'], 'ordered': False}
units1 (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
units2 (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'ordered': True}
dropout_rate1 (Float)
{'default': 0.0, 'conditions': [], 'min_value': 0.0, 'max_value': 0.5, 'step': None, 'sampling': 'linear'}
dropout_rate2 (Float)
{'default': 0.0, 'conditions': [], 'min_value': 0.0, 'max_value': 0.5, 'step': None, 'sampling': 'linear'}
```

Figure 38 Parameters tuned

We performed hyperparameter tuning with the aim of maximizing accuracy. After the hyperparameter tuning process, we trained the model using the tuned parameters. As a result, we achieved a test accuracy of 0.7512 and test loss of 0.7376. These results indicate the improved performance of the model after hyperparameter tuning.

```
Objective(name="val_accuracy", direction="max")

Trial 0023 summary
Hyperparameters:
activation_function: relu
units1: 192
units2: 352
learning_rate: 0.001
dropout_rate1: 0.45443640433752847
dropout_rate2: 0.21661425812447588
tuner/epochs: 25
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.7271164655685425
```

Figure 39 Best trial during hp training

5.10 SUMMARY

In this section, we report the performance of the models we have developed to compare them and draw some conclusions. The table below shows the results obtained during the various tests:

	Test Accuracy	Test Loss
Standard CNN	0.673	0.965
One Dense Layer and Four Conv2d Layers	0.670	0.960
One Dropout and Dense Layer	0.704	0.843
Two Dropout and Dense Layer	0.753	0.765
Batch Normalization	0.781	0.785
Data Augmentation	0.672	0.946
Depthwise Separable convolution	0.651	1.025

Depthwise Separable convolution deeper	0.677	0.932
Naïve Inception Module	0.773	0.655
Complete Inception Module	0.766	0.677
Residual Module	0.757	0.701
Residual Module deeper	0.758	0.733

As can be seen in the table, we have managed to achieve good performance by starting with simple and uncomplicated networks and, with each subsequent model, addressing the shortcomings of the previous one. We have also implemented three techniques that have helped us improve the results.
The overall best performing network is the one based on Inception Layers, as it provides good performance (especially in terms of loss) while maintaining a moderate model complexity, particularly in the case of the Complete Inception Module. We also achieve good results with models using Residual Modules and those with two layers of dropout and two dense layers. However, the network with batch normalization layers shows a different outcome: we obtain very high accuracy values, but the model demonstrates significant overfitting, as indicated by the high loss.

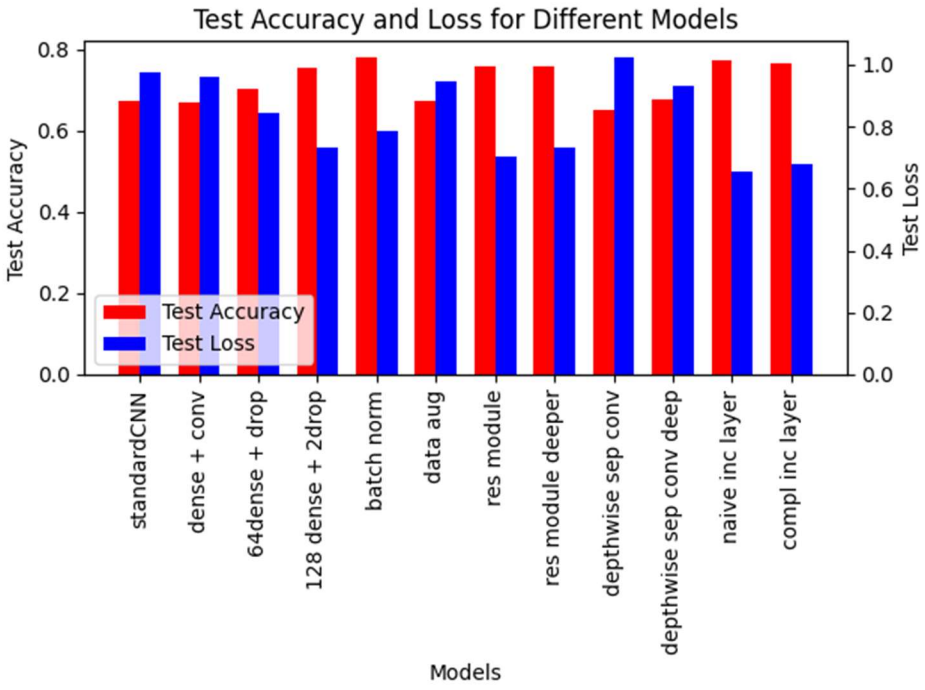


Figure 40 Comparison of models on performance on the test set

6 PRE-TRAINED CNN

We focused our work on ResNet50v2 pretrained CNN by using the transfer learning approach. We picked such a network because, giving a look at related works section of this dataset on Kaggle, it has the most promising results with 96% of accuracy. Moreover, from the literature we learn that in a 2019 ImageNet competition called the "ImageNet Large Scale Visual Recognition Challenge" (ILSVRC), ResNet50v2 achieved a top-1 accuracy of 79.3% and a top-5 accuracy of 94.6%, which place it among the most accurate neural networks for image classification. Since our dataset contains images of items, very often common and domestic ones, it's not very different from the famous "ImageNet" one, so we hope to get promising results for our classification task. We implemented transfer learning also on two other CNNs like VGG-16 and MobileNetv2 without performing several trials or any kind of parameter optimization. We test these models only because we need them in order to implement a classifier based on ensemble method.

ResNet50v2 is a deep convolutional neural network (CNN) that is part of the ResNet (Residual Network) family of networks and was developed for training on large images such as the ImageNet dataset.

The ResNet50v2 architecture consists of 50 layers of neurons, with "v2" indicating the second version of the ResNet50 architecture, which introduced some modifications compared to the first version. The main feature of the ResNet network is the use of skip connections, called "residual connections", which allow the signal to bypass one or more layers of the network, going directly to the next level. This helps to prevent the problem of "vanishing gradient" that can occur in deep neural networks.

The ResNet50v2 architecture uses 3x3 and 1x1 convolutions, maximum and median pooling, batch normalization, and dropout to prevent overfitting. The architecture consists of five convolutional blocks, each of which contains multiple convolution and batch normalization layers, followed by a ReLU activation function. There are also "bottleneck" convolution blocks that use 1x1 convolutions to reduce the dimensionality of the feature space, followed by 3x3 and 1x1 convolutions to expand the feature space dimensionality.

The network ends with a global pooling layer, which reduces the size of the feature maps to a one-dimensional feature vector, followed by a fully connected classification layer with 1000 neurons (corresponding to the 1000 classes in ImageNet). This classification layer can be replaced with a new custom classification layer to adapt the network to a specific target dataset.

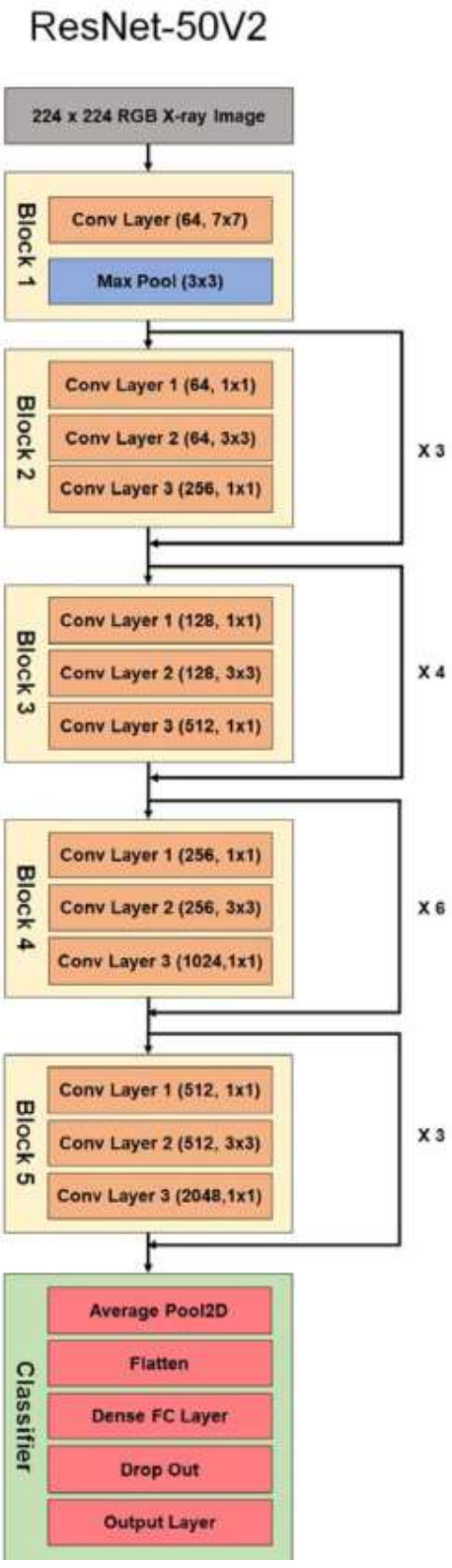


Figure 41 ResNet50v2 structure

In summary, the ResNet50v2 architecture is complex but highly effective for training on large images such as the ImageNet dataset, thanks to its ability to prevent the vanishing gradient problem using skip connections and to use various techniques to prevent overfitting.

6.1 TEST 1: CLASSICAL RESNET50V2 (FEATURE EXTRACTION)

The original ResNet50 comes with a GlobalAveragePooling2D and a prediction layer soon after. In this test we used the same approach, resizing the prediction layer to the number of classes we have. Moreover, we add a pre-processing layer to resize and rescale our input images.

- **BATCH SIZE = 32**
- **IMAGE HEIGHT = 224**
- **IMAGE WIDTH = 224**
- **NUM_CLASSES = 8**

```
num_classes = len(train_dataset.class_names)

inputs = keras.Input(shape=(IMAGE_HEIGHT, IMAGE_WIDTH, 3))
x = resize_and_rescale(inputs)
x = keras.applications.resnet_v2.preprocess_input(inputs)
x = conv_base(x)
x = layers.GlobalAveragePooling2D(name='my_glo_avg_pool')(x)
outputs = layers.Dense(num_classes, activation="softmax", name='predictions')(x)
model = keras.Model(inputs, outputs)

model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.Adam(learning_rate=0.0001),
              metrics=["accuracy"])
```

Figure 432 Base model overview

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 224, 224, 3)]     0

 tf.math.truediv_2 (TFOpLamb  (None, 224, 224, 3)      0
 da)

 tf.math.subtract_2 (TFOpLam  (None, 224, 224, 3)      0
 bda)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 my_glo_avg_pool (GlobalAver  (None, 2048)             0
 agePooling2D)

 predictions (Dense)         (None, 8)                 16392

=================================================================
Total params: 23,581,192
Trainable params: 16,392
Non-trainable params: 23,564,800
```

Figure 42 Base model layers' shape

The results obtained after training are:

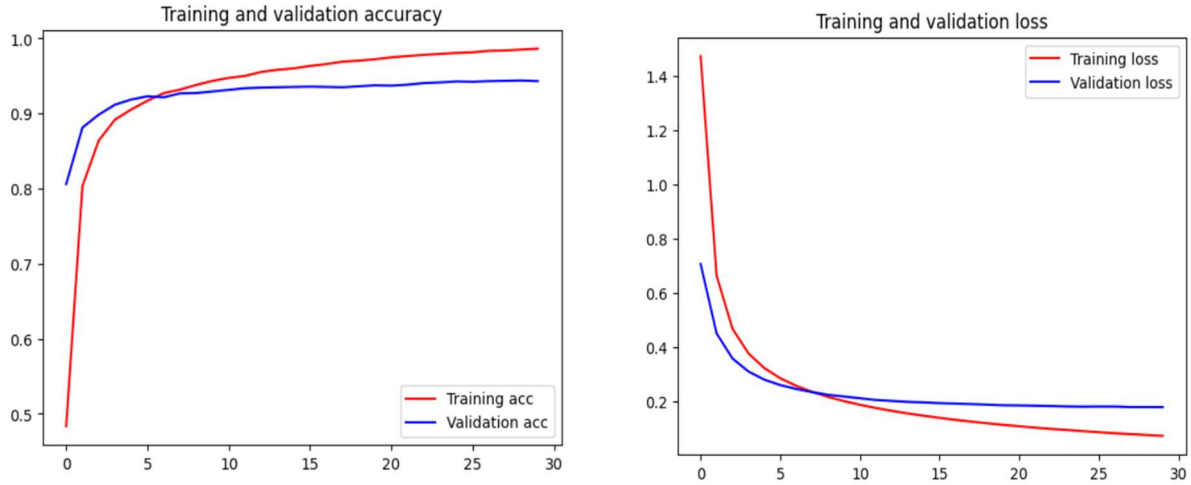


Figure 44 Training and loss plots

This test runs for 27 epochs and accuracy increases quite linearly while loss decreases uniformly. Best results are obtained in the last epoch:

```
loss: 0.0811 - accuracy: 0.9834 - val_loss: 0.1702 - val_accuracy: 0.9394
```

For what concern overfitting the gap between train and validation is not large, so results are quite satisfactory considering that this model is the simpler one. The reason for such performances maybe stands in the fact that images of our dataset are very similar to those one of "ImageNet" dataset on which ResNet50v2 has been pre-trained.

Let's give a look also at the confusion matrix computed on the test set:

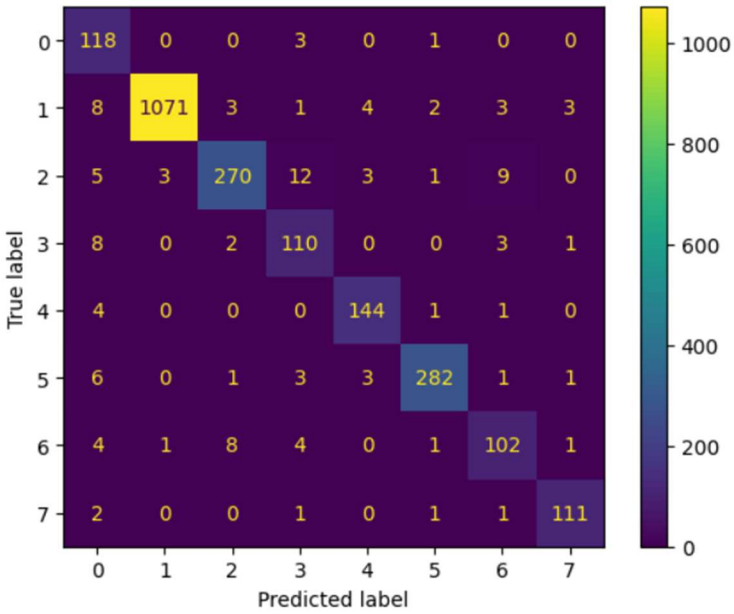


Figure 45 Confusion matrix

We plot also the ROC curves for each class and we computed the relative AUC.

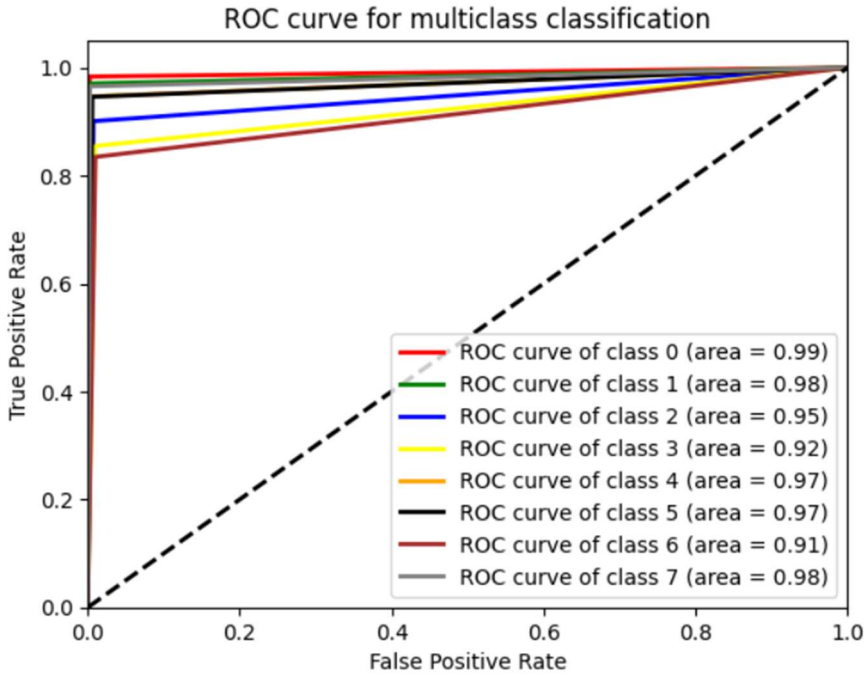


Figure 46- ROC plot

Altogether we obtained very good results, in fact all areas are very high and close to 1.00. The lowest value came from class 6 (related to “plastic” category) and it is 0.91. Let’s analyse this worst case.

An AUC (Area Under the Curve) value of 0.91 suggests that the classifier has a high discriminatory power and performs very well in distinguishing between the classes.

With an AUC of 0.91, the classifier is capable of correctly ranking instances from the positive class higher than instances from the negative class in approximately 91% of the cases. This indicates a strong ability to make accurate predictions and suggests that the classifier is effective at separating the positive and negative instances.

In practical terms, an AUC of 0.91 indicates a highly reliable classifier with a high true positive rate and a low false positive rate. It demonstrates that the classifier has a high probability of correctly identifying positive instances and a relatively low likelihood of misclassifying negative instances.

Overall, an AUC of 0.91 is considered very good and suggests a robust classifier with excellent performance in the task of distinguishing between the classes.

We run the same test with a 12-class model in order to see if our choice of reducing the number of classes is beneficial or not.

The results obtained after training are the following:

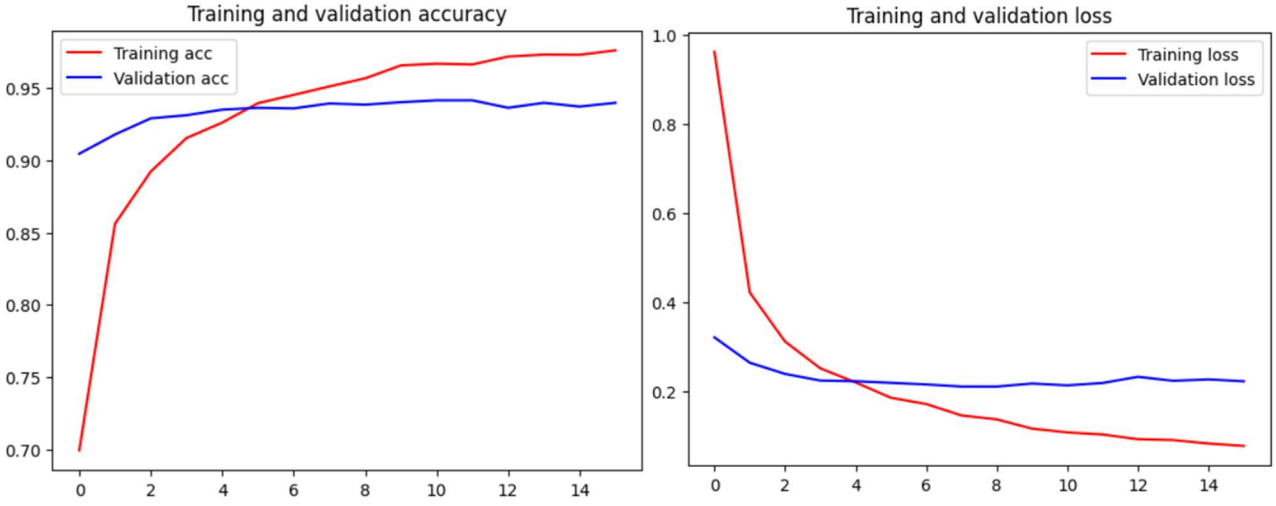


Figure 47 Training and loss plots

Performances are similar but slightly worst both in terms of accuracy and loss. Best result is:

```
loss: 0.1190 - accuracy: 0.9745 - val_loss: 0.2272 - val_accuracy: 0.93
```

From the comparison comes out that the 8-class model is more performant (each metric is better) so in the next experiments we'll use always this one.
Previous tests have been carried out with a simple hold-out validation, so with a fixed train/validation split.
We tried also to run the same test with a 5-fold cross validation method in order to compare the results, but simulation went always out of memory and disconnected every time.
Having only a standard version of Google Colab we have limited resources so we can't rely on this approach. In the next tests we'll use always simple hold-out validation method for this reason.

6.2 Test 2: Classical ResNet50V2 + Dense layer + Dropout layer (Feature Extraction)

In this test we try to add some extra layer to the base architecture to see if the training process has some benefits or not. First, we try to add a Dense layer in this way:

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_6 (InputLayer)        [(None, 224, 224, 3)]     0

 tf.math.truediv_3 (TFOpLamb  (None, 224, 224, 3)      0
 da)

 tf.math.subtract_3 (TFOpLam  (None, 224, 224, 3)      0
 bda)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 my_glo_avg_pool (GlobalAver  (None, 2048)             0
 agePooling2D)

 my_dense2 (Dense)           (None, 512)               1049088

 predictions (Dense)         (None, 8)                 4104

=================================================================
Total params: 24,617,992
Trainable params: 1,053,192
Non-trainable params: 23,564,800
```

Figure 48 Model overview

The procedure run for 24 epochs and altogether the metrics are improved apart from validation loss that resulted in a slightly higher value. In general results are satisfactory. These are the final plot of accuracy and loss:

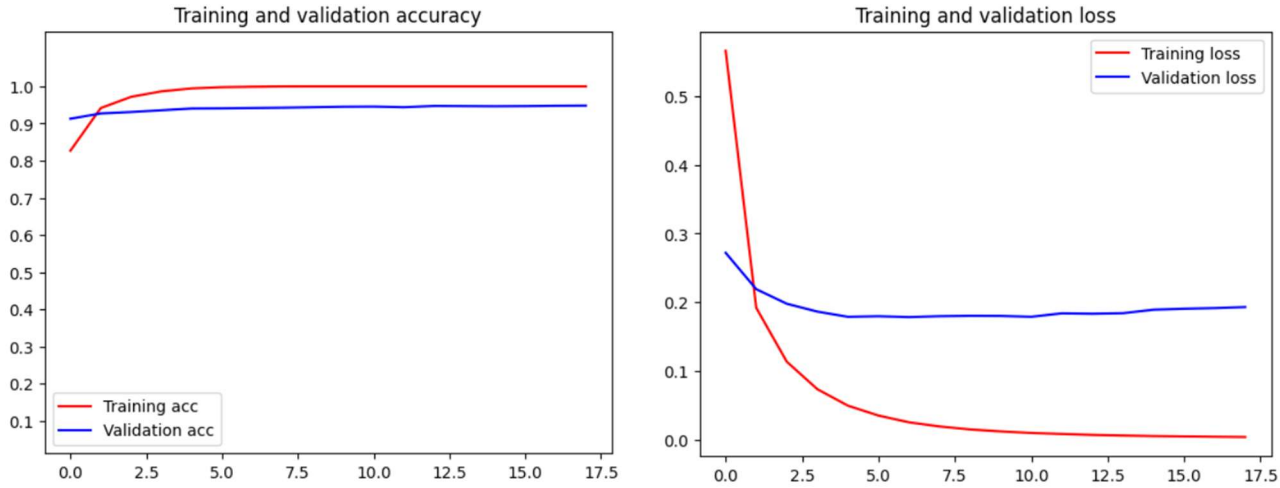


Figure 49 Training and loss plots

Then we tried to also add a dropout layer before the dense one inserted right now.

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 224, 224, 3)]     0

 tf.math.truediv_4 (TFOpLamb  (None, 224, 224, 3)      0
 da)

 tf.math.subtract_4 (TFOpLam  (None, 224, 224, 3)      0
 bda)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 my_glo_avg_pool (GlobalAver  (None, 2048)             0
 agePooling2D)

 dropout_1 (Dropout)         (None, 2048)              0

 my_dense2 (Dense)           (None, 512)               1049088

 predictions (Dense)         (None, 8)                 4104

=================================================================
Total params: 24,617,992
Trainable params: 1,053,192
Non-trainable params: 23,564,800
```

Figure 50 Model overview

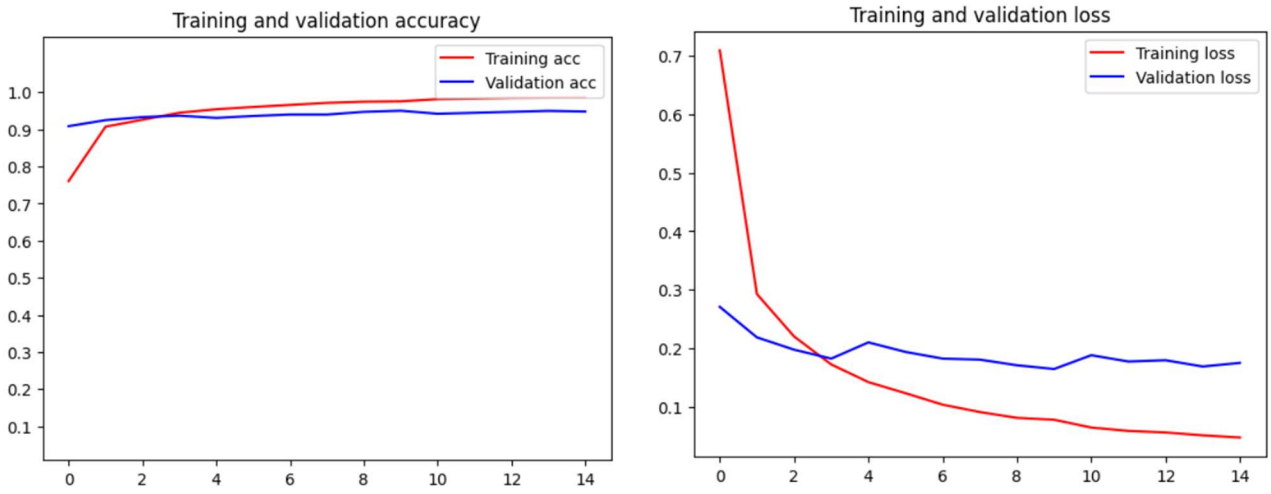


Figure 51 Training and loss plots

Again, results are not so different and network performances are very similar to the previous case, but we have a slight benefit in terms of validation loss that resulted lower than before. Best result has been obtained at epoch 10:

```
loss: 0.0774 - accuracy: 0.9754 - val_loss: 0.1642 - val_accuracy: 0.9502
```

6.3 Test 3: Fine tuning 1 block

ResNet50V2 is made of multiple big blocks (i.e., 5) so called conv in the model. These blocks are made of sub-blocks connected by each other by an add layer which connects the processed input (e.g., processed by Conv2D, Padding, Pooling, BatchNormalization) and the residual input. We finetuned considering these under-blocks, hence in this paragraph we finetuned the *conv5 block3*.

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 224, 224, 3)]     0

 tf.math.truediv_3 (TFOpLamb  (None, 224, 224, 3)      0
 da)

 tf.math.subtract_3 (TFOpLam  (None, 224, 224, 3)      0
 bda)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 my_glo_avg_pool (GlobalAver  (None, 2048)             0
 agePooling2D)

 my_dense2 (Dense)           (None, 512)               1049088

 predictions (Dense)         (None, 8)                 4104

=================================================================
Total params: 24,617,992
Trainable params: 5,511,688
Non-trainable params: 19,106,304
```

Figure 52 Model overview

Making this choice 5.5 million of parameters are now ready be trained.

Looking at the results the convergence is fast because training lasted only 11 epochs and results epoch by epoch remained constant. In conclusion all metrics are improved except for validation

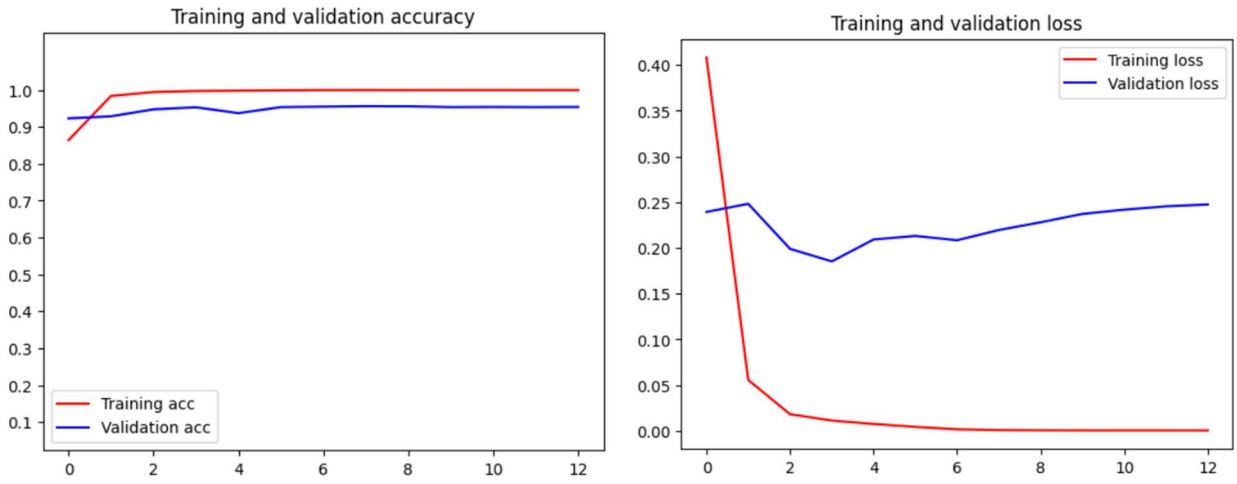


Figure 53 Training and loss plots

loss that suffered a remarkable increase with a peak of 0.2486.

6.4 TEST 4: FINE TUNING 2 BLOCKS

In this test we tried to fine tune another block, but results are not good. The reason is that unfreezing another block too many parameters are trained (almost 10 million in this case). Now, in addition to "*block5 conv3"* we unfreeze also "*block5 conv2"*.
The convergence of the training is faster because it lasted only 9 epochs.

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 224, 224, 3)]	0
tf.math.truediv_3 (TFOpLambda)	(None, 224, 224, 3)	0
tf.math.subtract_3 (TFOpLambda)	(None, 224, 224, 3)	0
resnet50v2 (Functional)	(None, 7, 7, 2048)	23564800
my_glo_avg_pool (GlobalAveragePooling2D)	(None, 2048)	0
my_dense2 (Dense)	(None, 512)	1049088
predictions (Dense)	(None, 8)	4104

Total params: 24,617,992
Trainable params: 9,970,184
Non-trainable params: 14,647,808

Figure 54 Model overview

In term of accuracy results are still good but for what concern loss we obtain a shaky trend with a peak of 0.373 in validation loss. However, in terms of validation accuracy we got the best result so far 0.9630.

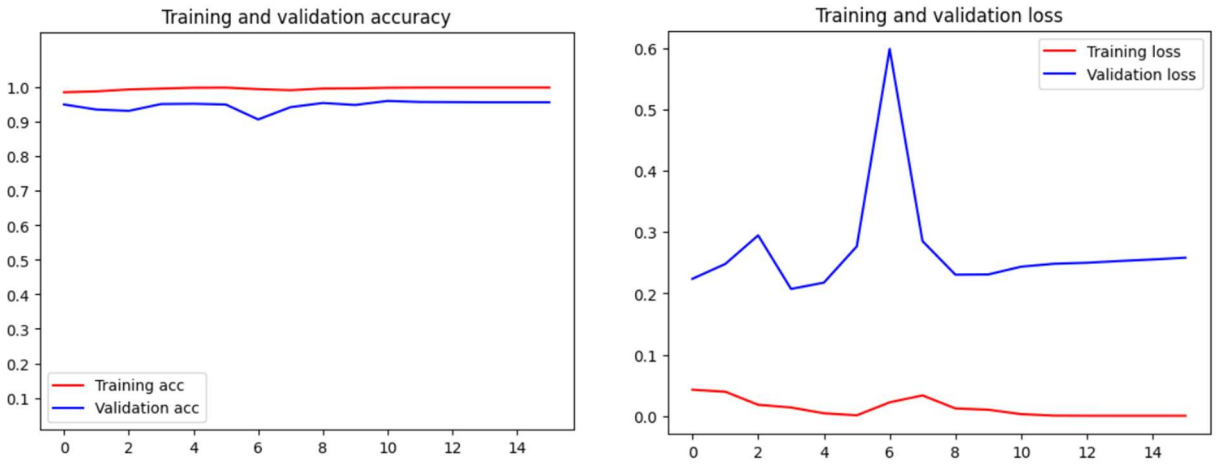


Figure 55 Training and loss plots

6.5 TEST 5: FINE TUNING ONLY 2 LAST LAYERS

In this test we tried to reduce the number of trainable parameters to see whether there is different behaviour during training. We unfreeze only the two last layers of block5_conv3 that are: "*conv5 block3 3conv"* and "*conv5 block3 out"*.

Now we have around 2 million trainable parameters. Let's give a look at the results:

The process run for 8 epochs only. Accuracy and validation accuracy have a linear trend and they are very high and constant. There is no overfitting from this point of view. Validation is very low, but validation loss is augmented again compared to previous tests.

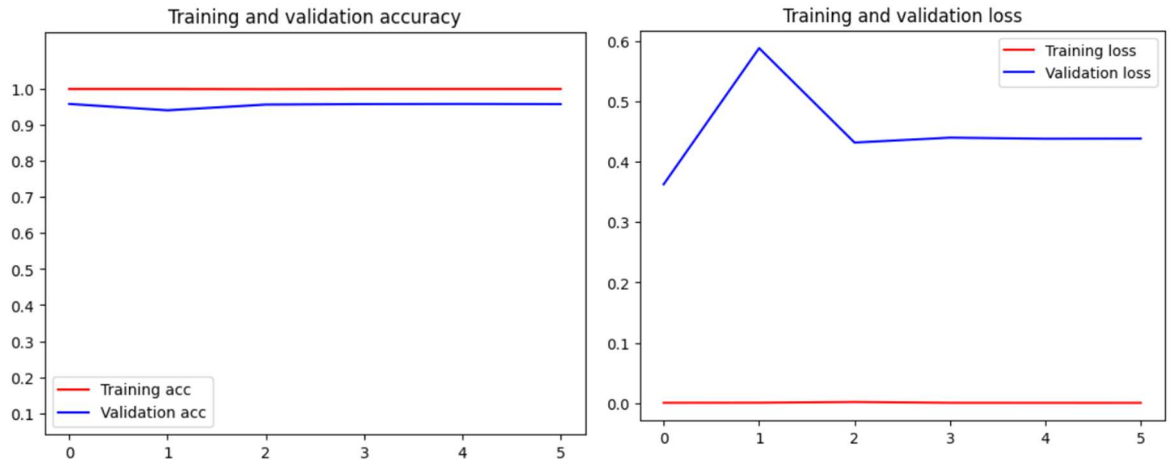


Figure 56 Training and loss plots

6.6 TEST 6: CLASSICAL RESNET50V2 WITH CLASS WEIGHTS

As we mentioned in pre-processing chapter, in this section, we explore the possibility to consider a weight for each class based on number of samples belonging to that class. We tried this approach on the classical ResNet50v2. Below we can see the resulting plots:

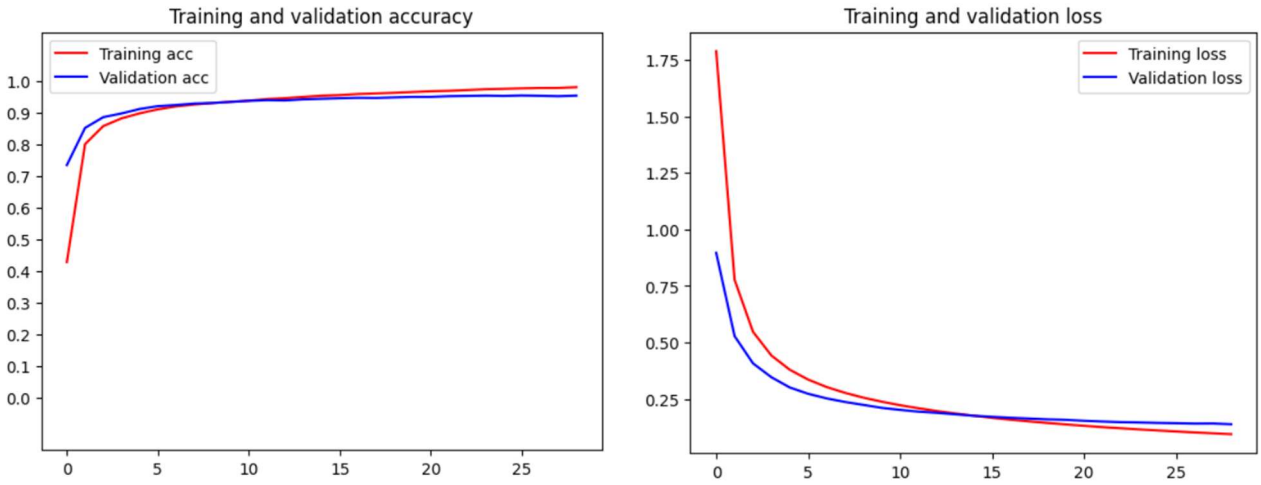


Figure 57 Training and loss plots

As we can observe results are very good, even better than all the other tests.

The simulation run for 29 epochs and best statistics were in the last one:

```
loss: 0.0947 - accuracy: 0.9798 - val_loss: 0.1386 - val_accuracy: 0.9532
```

The overfitting is minimal and a validation loss of 0.1386 is the best result obtained so far.

6.7 HYPERPARAMETER OPTIMIZATION

In this section we run some experiments with Keras Tuner to find the optimal configuration for the hyperparameters of CNN selected for this project. We try to play with parameters such as learning rate of optimizer, number of hidden units of dense layer, activation function of dense layer and the presence or not of a dropout layer. We run two different simulations: one aimed to maximize validation accuracy and the other aimed to minimize validation loss.

```
inputs = keras.Input(shape=(IMAGE_HEIGHT, IMAGE_WIDTH, 3))
x = resize_and_rescale(inputs)
x = keras.applications.resnet_v2.preprocess_input(inputs)
x = conv_base(x)
x = keras.layers.GlobalAveragePooling2D(name='my_glo_avg_pool')(x)

x = keras.layers.Dropout(dropout_rate_hp)(x)

x = keras.layers.Dense(units=hp_units, activation=activation)(x)

outputs = keras.layers.Dense(8, activation="softmax", name='predictions')(x)
model = keras.Model(inputs, outputs)
```

Figure 58 Model overview

```
Search space summary
Default search space size: 4
dropout_rate (Float)
{'default': 0.0, 'conditions': [], 'min_value': 0.0, 'max_value': 0.5, 'step': 0.05, 'sampling': 'linear'}
activation (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'elu', 'gelu'], 'ordered': False}
units (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'ordered': True}
```

Figure 59 Involved parameters

```
Objective(name="val_accuracy", direction="max")

Trial 0020 summary
Hyperparameters:
dropout_rate: 0.30000000000000004
activation: gelu
units: 352
learning_rate: 0.001
tuner/epochs: 7
tuner/initial_epoch: 0
tuner/bracket: 1
tuner/round: 0
Score: 0.9535883069038391
```

```
Objective(name="val_loss", direction="min")

Trial 0024 summary
Hyperparameters:
dropout_rate: 0.30000000000000004
activation: relu
units: 160
learning_rate: 0.0001
tuner/epochs: 20
tuner/initial_epoch: 7
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0019
Score: 0.15341199934482574
```

Figure 60 Validation accuracy and validation loss best results

The above results are not so far from those obtained in tests 1, 2 and 3. Best values were obtained in test 3 where we reached 0.9502 for validation accuracy and 0.1642 for validation loss. For sure we'll set dropout rate to 0.3 since it is in common between the two optimal configurations.

6.8 SUMMARY

In this section we want to show a recap of the metrics of all the models after running an evaluate() method on the test set in order to draw some conclusions.

Model	Test Accuracy	Test Loss
base	0.95189	0.152345
base + dense	0.949313	0.147765
base + dense + drop	0.955326	0.146311
base weights	0.973797	0.0922346
base ft one block	0.953608	0.147772
base ft two block	0.957904	0.17789
base ft two layers	0.964777	0.181658

Figure 61 Evaluation results on test set

As we can see best results were obtained in test number 4 where we train the model by using class weights instead of balancing the number of samples per class. We didn't expect such result, but this approach seems to be very promising. If we look at the last three models (those related to fine tuning) we can see that accuracy experienced a benefit, instead loss worsen test after test except for what concern the first test of fine tuning. In fact, when we fine tune only the last convolutional block test loss slightly improves. We expected this kind of behaviour because tests on fine tuning had a bit of overfitting in the loss plot. In conclusion all these models are very accurate and we are happy of the results that we got.

7 ENSEMBLE METHOD

An ensemble classifier is a machine learning model that combines multiple individual models to improve the overall prediction accuracy and robustness of the classifier. In essence, the idea behind ensemble classifiers is to harness the strengths of multiple models and minimize their weaknesses by combining their predictions. Ensemble classifiers are often used in practice because they can improve the generalization of the model and reduce overfitting. By combining multiple models, ensemble classifiers can also provide a more robust prediction than a single model, especially when the individual models are diverse and complementary to each other.

In this section we try to use this approach with the best model shown in the previous chapters.

We adopt four models (three pre-trained networks and the best CNN built from scratch):

- ResNet50v2 fine-tuned model
- MobileNetv2 model
- VGG16 model
- CNN from scratch (with two dropout and dense layer)

We run tf.keras.model predict method on the test set for each model. Then we concatenate all the results (predictions). For each image of test set we sum up all four different predictions and we average them to obtain only one value per image. This method is called average voting.

$$X = \frac{M1 + M2 + M3 + M4}{4}$$

Classification report:

	precision	recall	f1-score	support
0	0.9520	0.9754	0.9636	122
1	0.9973	0.9954	0.9963	1095
2	0.9655	0.9241	0.9444	303
3	0.8992	0.9355	0.9170	124
4	0.9610	0.9867	0.9737	150
5	0.9898	0.9764	0.9831	297
6	0.8672	0.9174	0.8916	121
7	0.9741	0.9741	0.9741	116
accuracy			0.9738	2328
macro avg	0.9508	0.9606	0.9555	2328
weighted avg	0.9743	0.9738	0.9739	2328

Figure 62 Classification results

Finally, we can evaluate performances by plotting the confusion matrix:

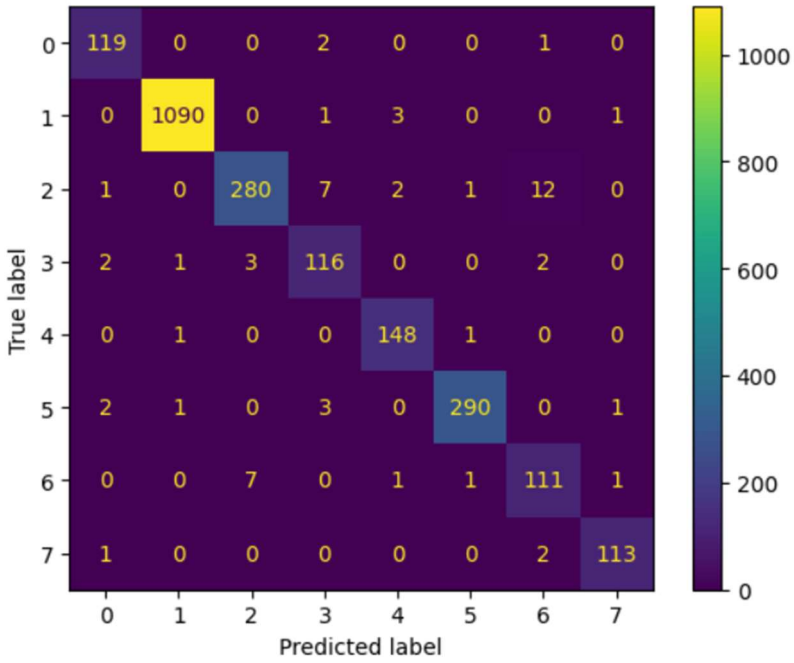


Figure 63 Confusion matrix

As we expected the overall accuracy is improved, reaching 97.38%. Precision and recall are very good for all classes. In conclusion this model is the best found so far.

7.1 MISCLASSIFICATION ANALYSIS

In this paragraph we looked at the predictions of the four models used in ensemble classifier and we focus our attention on misclassified images. Test set contains 2328 samples.

```
Misclassified images from all 4 classifiers:  18
Misclassified images from 3 classifiers:  53
Misclassified images from 2 classifiers:  127
Misclassified images from 1 classifiers:  584
```

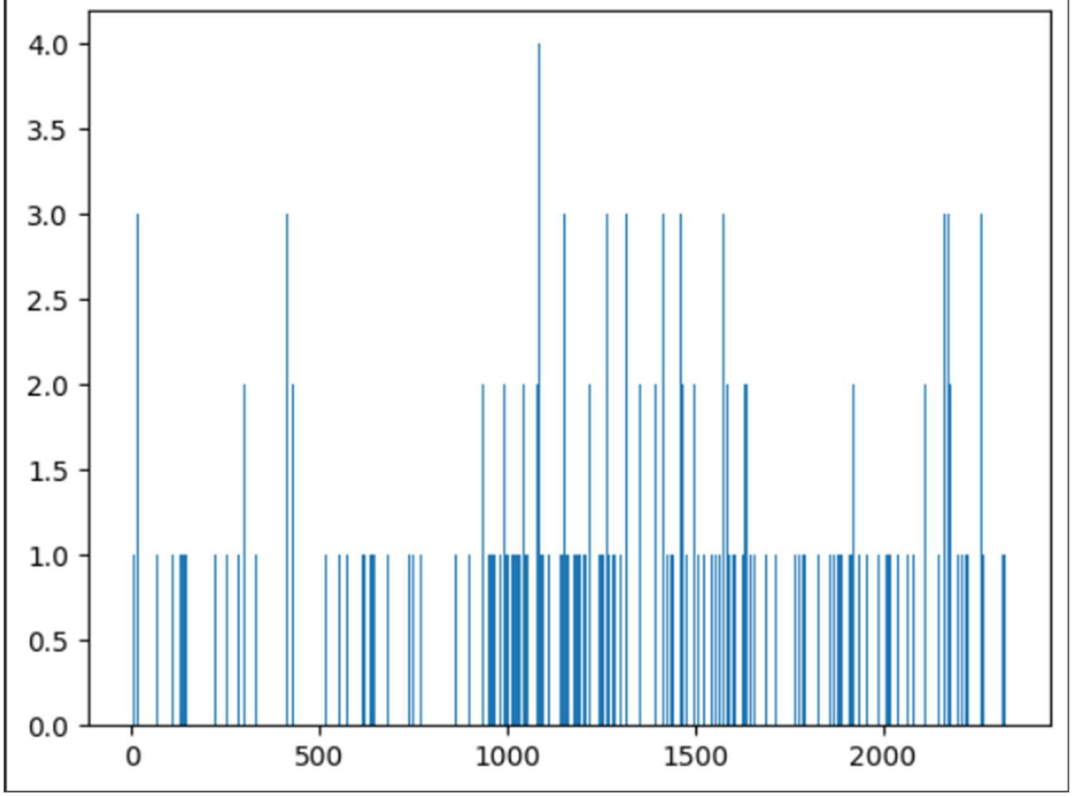


Figure 64 Classification errors image per image

The overall behaviour of our models is very good, in fact only 18 images were misclassified by all classifiers.

The number of images correctly recognised by all classifiers are 1546 (66% of data).

From these results we can see that ensemble method is very useful and beneficial and by using it we reduce a lot the number of errors.

```
Images that are misclassified by all classifiers:  18
--------
Images that are misclassified only by VGG16:  32
Images that are correctly classified only by VGG16:  9
--------
Images that are misclassified only by ResNet50v2:  9
Images that are correctly classified only by ResNet50v2:  24
--------
Images that are misclassified only by MobileNetv2:  37
Images that are correctly classified only by MobileNetv2:  10
--------
Images that are misclassified only by CNN from scratch:  506
Images that are correctly classified only by CNN from scratch:  10
```

Figure 65 Classification errors on all models

By investigating errors model by model, we get the following result:
As we expected ResNet50v2 is the best model since it got the highest accuracy and lowest loss both in training and in validation. On the contrary CNN from scratch performances are not so good because it misclassified 506 images (21.7% of total data). From this table we can conclude that ensemble method is beneficial because all models perform in a different way, processing images in a unique way and detecting different kind of features network by network.
Now let's analyse misclassified images to see whether there is a common factor that explains the reasons of the error.

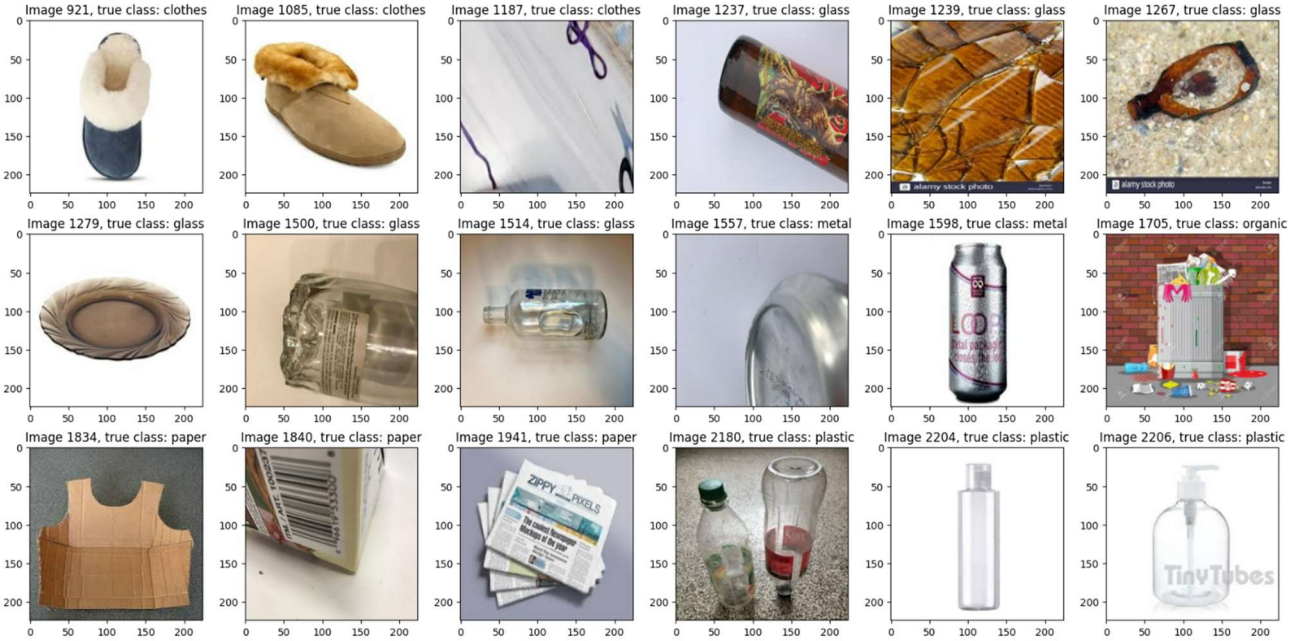


Figure 66 Images misclassified by all four models

Most of errors were in "glass" class maybe because some images contain fragments of broken glass and some other contains transparent glass that may be confused with plastic. Some other images are meaningless or noisy and even human reasoning don't help in distinguish them. An interesting case is the one of image 1834 that depicts a dress made by cutting a piece of cardboard. The true label in fact is "paper" but models classified it as "clothes" because of its shape. This case helps us understand the difference between human classification and computer vision in our opinion.

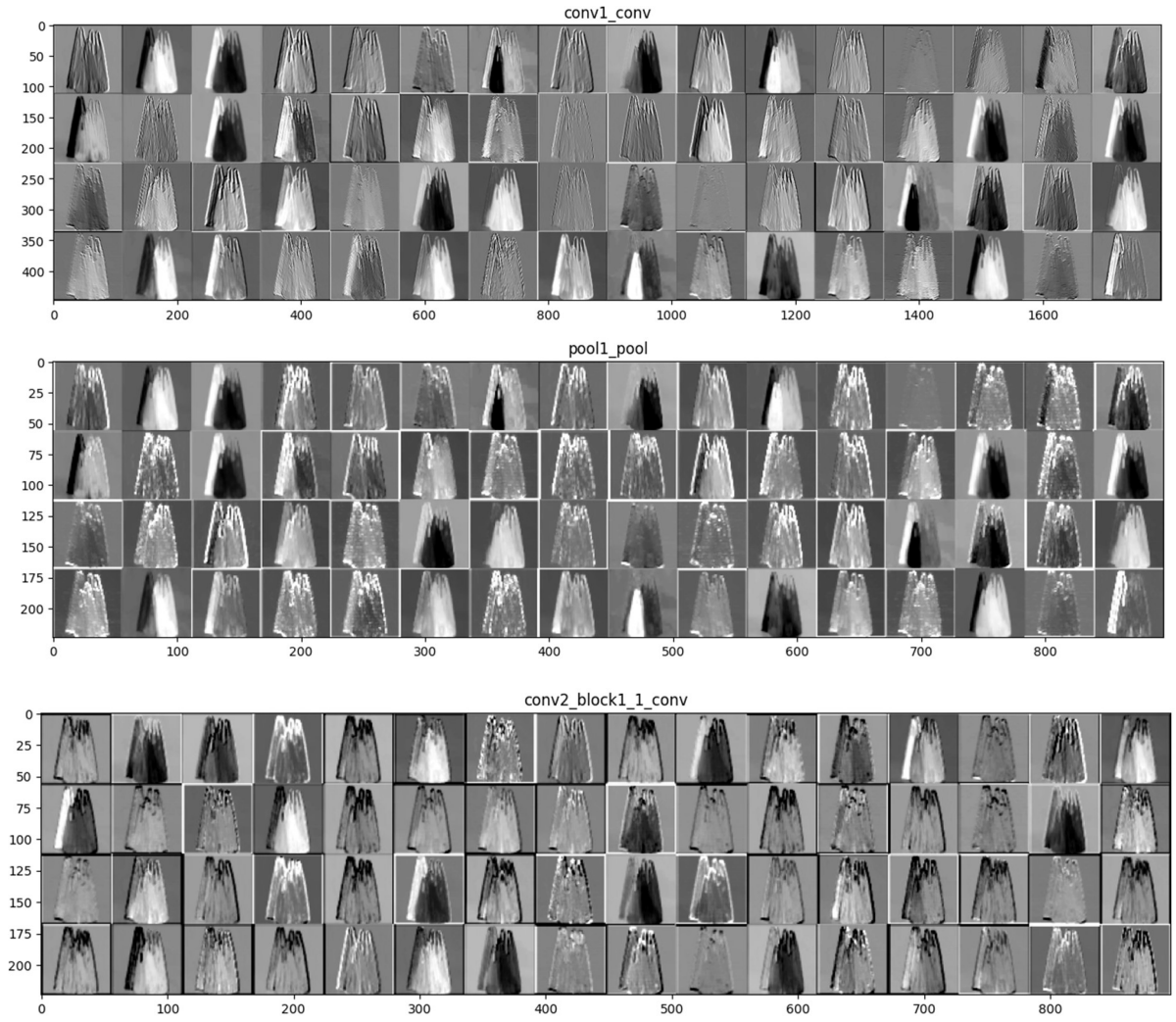
8 CONVNET BEHAVIOUR

In this section, we will explore the behavior of our neural network by examining its intermediate activations, visualizing convnet filters, and analysing heatmaps of class activation. Understanding how a neural network processes information and makes predictions can provide valuable insights into its decision-making process and aid in model interpretation. Tests are conducted using the model based on ResNet50v2.

8.1 INTERMEDIATE ACTIVATIONS

Intermediate activations refer to the output of intermediate layers in a neural network during the forward propagation process. As data flows through the network, it undergoes a series of transformations and computations in each layer. The intermediate activations capture the internal representations learned by the network at different stages of processing, so, layer by layer, we can observe how the network extracts and encodes features from the input data.

Below we present the intermediate activations of the first layers in our network:



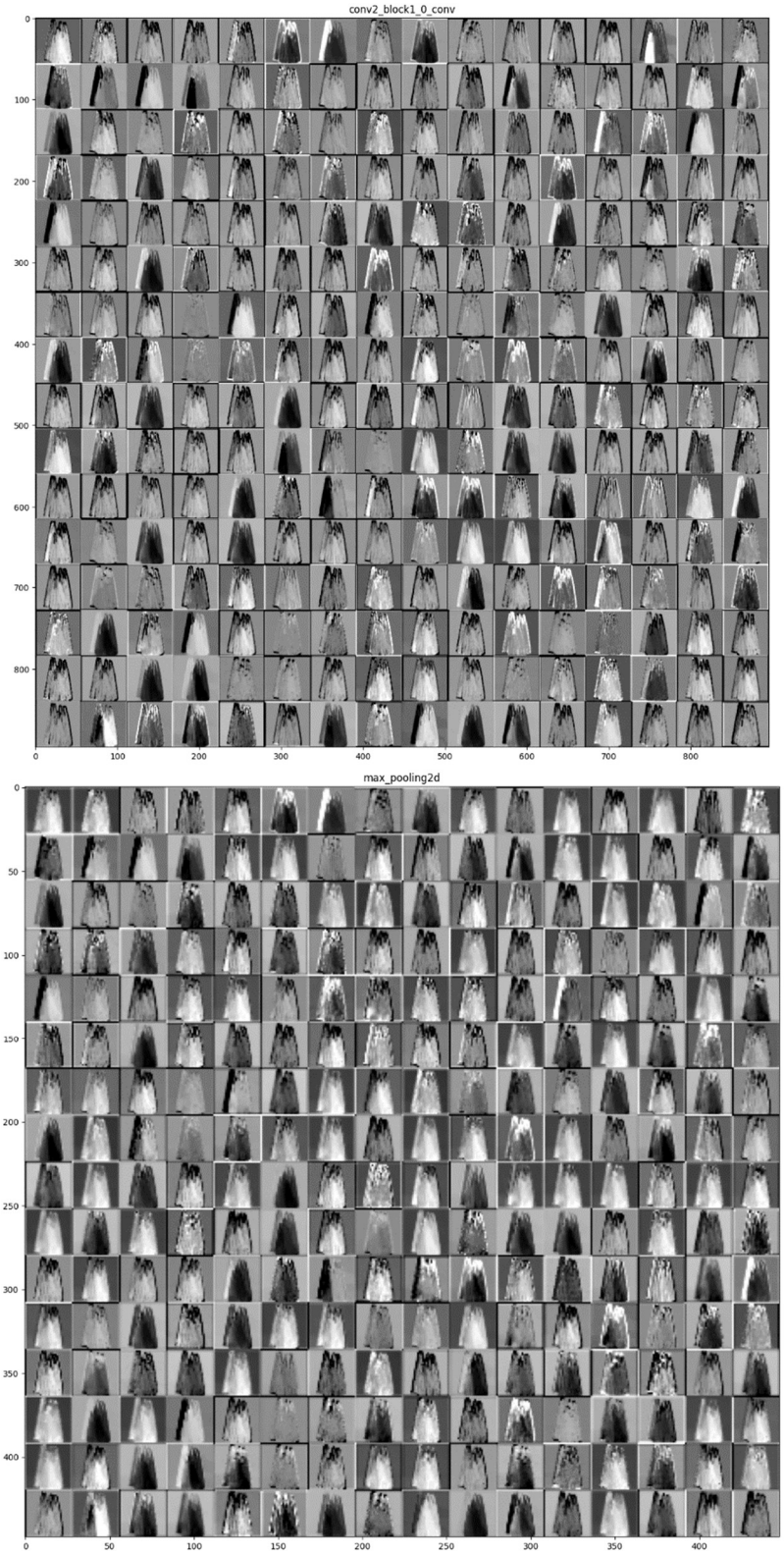


Figure 67 Intermediate activations

The initial layer appears to preserve the complete shape of our sample image, although there are some filters that remain inactive and are almost blank. At this stage, the activations retain nearly all the information from the original image. As we delve deeper into the subsequent layers, the activations become increasingly abstract and less visually interpretable. They start encoding higher-level concepts such as individual borders, corners, and angles. The higher-level

representations carry progressively less information about the visual details of the image and more information related to the image's class or category.

8.2 VISUALIZING CONVNET FILTERS

Visualizing convolutional filters involves examining the learned filters in CNNs and understanding the visual patterns to which they respond. Our goal is to determine the input image that maximizes the response of a specific filter. To achieve this, we employ a technique called gradient ascent in the input space. By applying gradient ascent to the input image of a convnet, we iteratively adjust its values to maximize the response of the desired filter. This process starts with a blank input image and produces an image that elicits the highest response from the chosen filter. We define a loss function that maximizes the value of a given filter in a specific convolution layer. This loss function is used during the gradient ascent process to update the values of the input image, thereby maximizing the activation value.
First, we randomly selected an image from the dataset on which to perform the operations, then we retrieve the pretrained model based on ResNet, listing all layers' names.

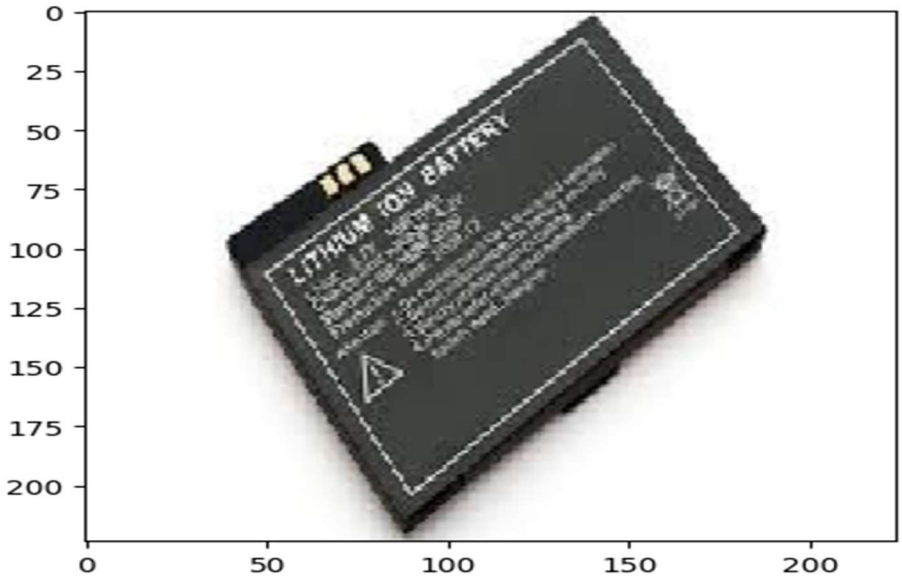


Figure 68 Sample image

The loss function and the function implementing the gradient ascent process were implemented using Gradient Tape from TensorFlow. The result is displayed below (more examples can be found in the notebook).
This operation can be performed for each layer and for each filter, helping us gain a better understanding of what the convolutional layers perceive, allowing us to explore the hierarchical nature of feature extraction in convolutional neural networks and provides valuable insights into the inner workings of these models.

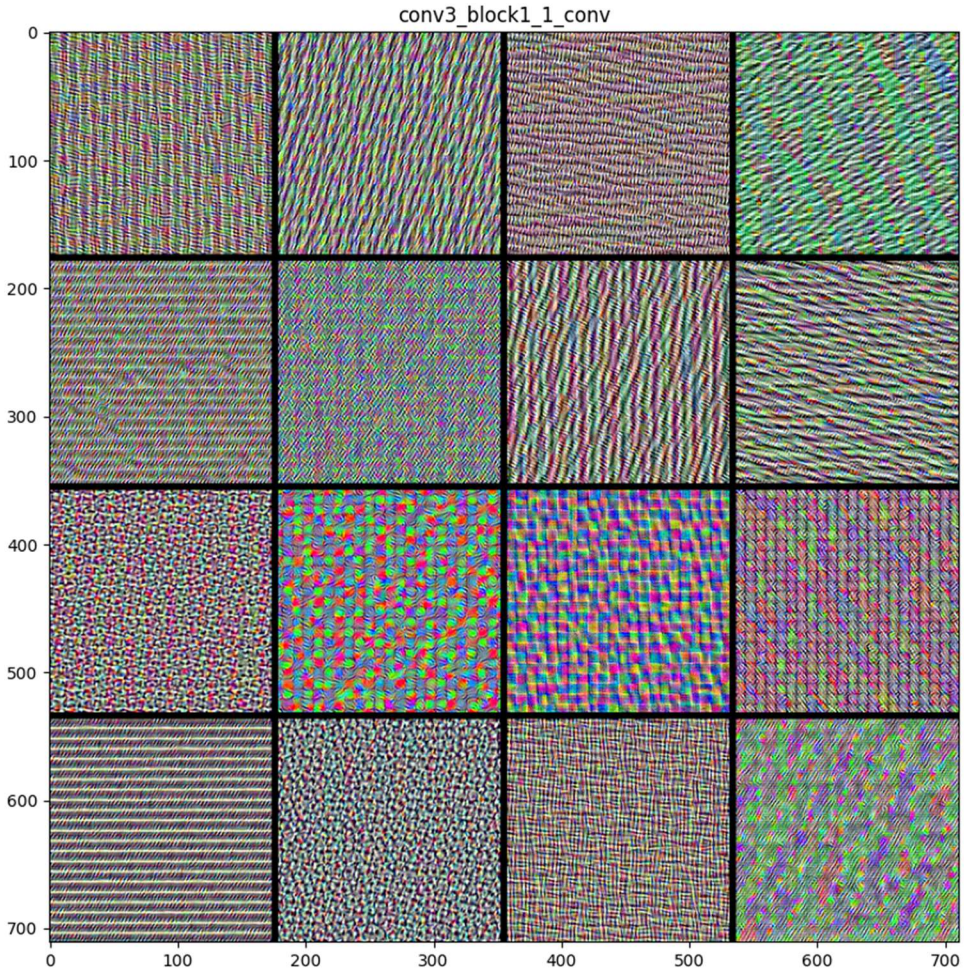


Figure 69 Preferred path from conv3_block1_1_conv

8.3 HEATMAP OF CLASS ACTIVATION

The heatmap of class activation is the last method we use to understand the behaviour of our model. It is built using the Grad-CAM (Gradient-weighted Class Activation Mapping) method, that is a visualization technique that highlights the important regions of an image that influence the prediction of a specific class in a CNN. Grad-CAM combines the gradient information from the final convolutional layer and the global average pooling layer of the CNN. It computes the gradients of the predicted class with respect to the feature maps in the final convolutional layer. These gradients are then used to obtain the importance weights for each feature map, indicating the contribution of that map towards the predicted class. By multiplying the importance weights with the corresponding feature maps, a weighted combination is obtained, representing the class activation map. This map indicates the regions in the image that strongly activate the prediction for the specific class. It highlights the areas where the CNN is focusing its attention to make the classification decision. The resulting heatmap can be overlayed on the original image, visually indicating the regions that the CNN considers most relevant for the predicted class.

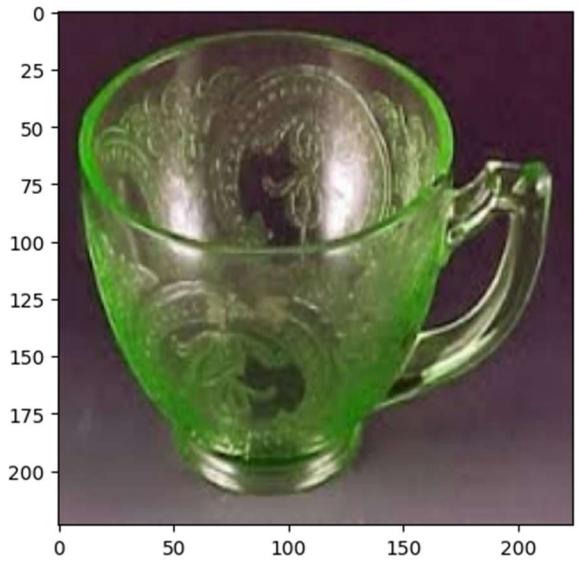


Figure 71 Sample image

Applying this technique to a random image, we obtained the results shown below. In our case, the image belongs to the 'glass' class: the network is likely focusing on the handle of the cup to recognize its class.

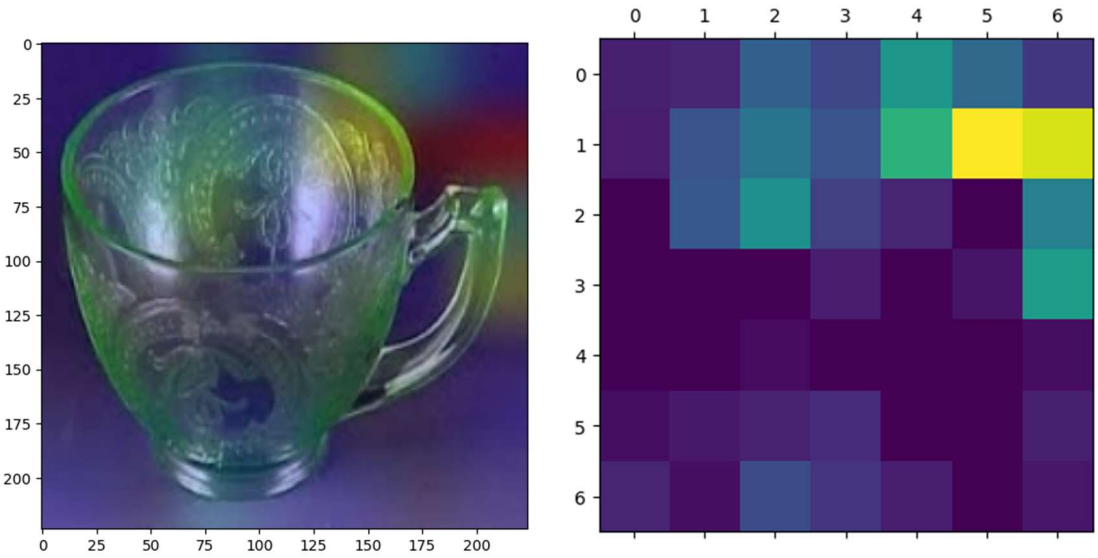


Figure 70 Heatmap of the sample image

9 CONCLUSIONS

We conducted several tests by developing both CNNs from scratch and pretrained, ultimately combining them through ensemble methods, yielding encouraging results. However, there are still ample opportunities for improvement. Firstly, limited computational resources slowed down our progress and prevented us from conducting all the desired tests. Additionally, a larger dataset would have significantly enhanced the performance of our networks.
Despite these limitations, our experiments showcased the potential of both approaches. Building networks from scratch allowed us to tailor architectures and optimize parameters to our specific task, while pretrained networks provided a valuable starting point by leveraging prelearned features. The ensemble method further augmented our results by combining the strengths of different models. We utilized a hyperparameter tuner for optimizing the hyperparameters, but only on a subset of models. A more comprehensive application of the tuner would have likely resulted in improved performance for all the networks.
Finally, acquiring a larger and diverse dataset would greatly improve the generalization capabilities of our networks: the data we utilized predominantly consists of images collected from the web, with a significant portion of them being of low quality. For these types of networks, access to high-quality images is crucial for the success of the project.

10 REFERENCES

Garbage Classification (12 classes) | Kaggle

Intelligent Waste Classification System Using Deep Learning Convolutional Neural Network - ScienceDirect

Waste Classification using Convolutional Neural Network | Proceedings of the 2020 2nd International Conference on Information Technology and Computer Communications

CNN Basic Architecture for Classification & Segmentation - Data Analytics

cs230.stanford.edu/projects_spring_2020/reports/38847029.pdf

2202.12258.pdf

https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/

https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec

https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/

https://deepai.org/machine-learning-glossary-and-terms/inception-module

https://machinelearningmastery.com/using-depthwise-separable-convolutions-in-tensorflow/

https://soroushhashemifar.medium.com/depth-wise-separable-convolution-explained-in-tensorflow-9be6aeaa4f8b