COMPUTATIONAL INTELLIGENCE AND DEEP LEARNING

A.Y. 2022/2023

Technical Report

Carried out by:

- **Leonardo Bellizzi** – 643019 – l.bellizzi@studenti.unipi.it

[Project entirely written on Google Collab and loaded on personal GitHub with Report, Presentation and Models obtained by different experiments]

Contents

1. INTRODUCTION

Project goal is to create an image classifier that can recognize the number of fingers and relative hand starting from an image.

To develop an accurate system two CNN were used. Two CNN were written and developed from scratch, in the first part of the project, one shallow and the other one deeper, then for the second network a hyper tuning phase were set up to figure out if with some other parameters the Network behaves in the same way.

The second part of this project leverage on a pre-trained network (ResNet-50) that use ImageNet weights and has been modified to cope with the specific task of the project. Then with this pretrained network, two optimizers (SGD, Adam) were tested with and without Data Augmentation.

When data augmentation is used, all ResNet50 layers were frozen to figure out some advantages can be obtained like faster execution. Eventually, the two augmented networks with the two optimizers were fine tuned by unfrozen final layers.

The aim is to demonstrate that is possible to use multiple classifiers with different technologies to complete an image classification task.

2. DATASET

Dataset was found at this link: https://www.kaggle.com/datasets/koryakinp/fingers/. Dataset contains 21600 images of left and right hands fingers.

All images are 128x128 and images own to 12 classes.

Training set contains 18000 images (1500 images per class) and the Test set contains 3600 images.

Training set was already balanced and to use It on Google Collab the upload on Google Drive was necessary.

Images own to these classes (0L,1L,2L,3L,4L,5L,0R,1R,2R,3R,4R,5R)

3. PREPARING DATASET FOR LATER USAGE

Initially, dataset folder contained two subfolders train and test. Images were without labels so a function that takes label from the image path were used. For example, the path was ''/train/fff79b8e-4d29-…-4e98_2L.png" and the custom function taken the path returned the label.

Since one of the aspects to take in consideration is the overfitting, creation of a **validation set** was considered. To do this split, the function **train_test_split** method of **sklearn.model_selection** was used**.**

On Google Collab the maximum RAM allowed, before the forced arrest, is 12.7 GB and running the split function combined with the custom function to extract labels went the program out of RAM memory.

This happened because the **train_test_split** function use arrays that in case of large dataset is too expensive to use in term of computational power.

After some days of research another method was found that is **image_dataset_from_directory** of **keras.utils**. A drawback of this method is that the dataset must be organized in a hierarchical way, so a new script was written to organize images in subfolders.

The function to extract label was used and each folder contained the images of that label. Now subfolders are labels and inside, all images of that labels are in.

The use of **train_test_split** with the custom function brough the code to have a O(n^2) complexity so RAM run out fast. Then, the keras function was used and RAM was constantly on 2.5 GB of usage and was faster than the train_test_split function because that function works with arrays that are very slow (split and labelling with the classical function was about 13 minutes while with keras about 40 seconds).

Instead, the keras function transforms directly in tensors that are more efficient to use.

Moreover, keras function does provide several features that can improve the efficiency of working with image datasets such as:

- **Batching and shuffling** of the data to improve training performance.
- **Data augmentation,** which can help to reduce overfitting and improve overall performance.
- **Automatic detection of class labels** on the directory structure, which can save time and reduce the chance of errors compared to manual labelling.

That function was specifically designed to work with TensorFlow for further image processing, so it's optimized for that used and is completely integrated in the TensorFlow ecosystem.

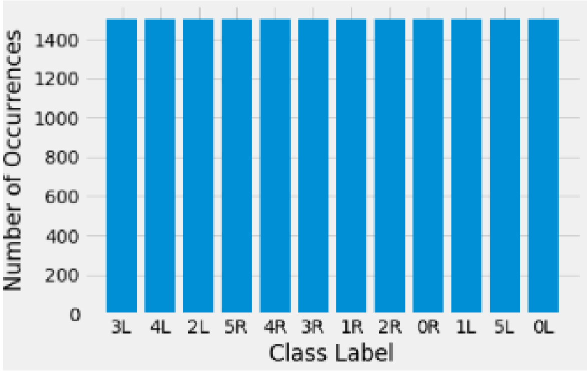
The strength is that on the split part now, some optimization method can be used like the **Autotune** provided by TensorFlow.

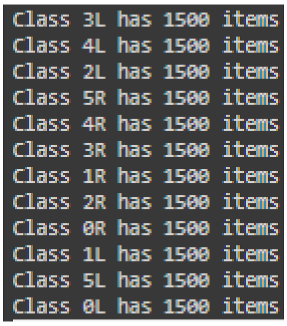
This method builds a performance model of the input pipeline and runs an optimization algorithm to find a good allocation of its CPU budget across all parameters. While the input pipeline is running, tf.data tracks the time spent in each operation, so that this time can be fed into the optimization algorithm.

No class weights and balancing operations were done since dataset was already balanced since the beginning.

Conclusion of this dataset analysis is that usage of **train_test_split** method is very restrictive because cannot be used for large dataset while the keras one can be used for dataset like the one used for this projects or for dataset that are 3 or 4 times bigger.

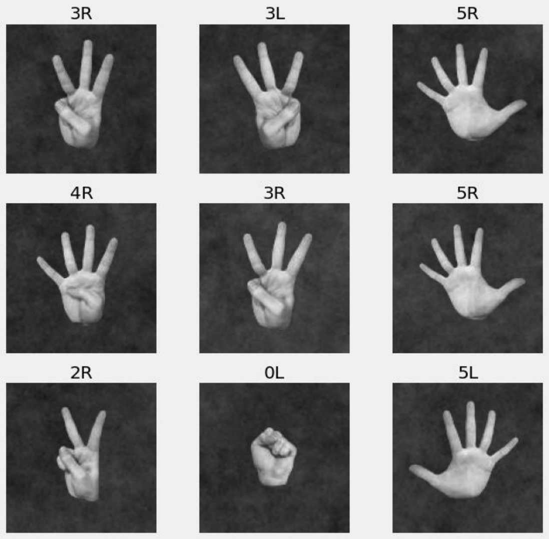
Training distribution:





```
Class 3L has 1500 items
Class 4L has 1500 items
Class 2L has 1500 items
Class 5R has 1500 items
Class 4R has 1500 items
Class 3R has 1500 items
Class 1R has 1500 items
Class 2R has 1500 items
Class 0R has 1500 items
Class 1L has 1500 items
Class 5L has 1500 items
Class 0L has 1500 items
```

This is how look like the dataset after keras function usage:



4. CODE STRUCTURE

Here the code structure is shown to make clear what was done after the train, validation and test splitting.

A **first CNN** from scratch was set up and tested, then some graphs of performance are shown and an explainability part was carried out.

A **second CNN** from scratch was set up and tested, this time more deeper with some performance graphs plotting with the explainability part.

In both cases data augmentation was added in the network construction.

The **second CNN** than was **Hyper Tuned.**

The **third section** leverage on **ResNet-50** and two optimizers were used Adam and SGD.

Practically speaking there is a section **no data augmentation with adam and SGD**. In the sections the pretrained network is used as feature extractor and then it's attached to custom classifier.

Moreover, **ADAM** and **SGD** are used with **data augmentation** with their relatives' plots. The **data augmentation** with the two optimizers is done by freezing all RESNET50 layers.

The **fourth** and last section has custom **RES-NET50 Fine Tuning** with ADAM and SGD with relatives plots and some explainability part, since **Transfer Learning** is used. Six layers were unfrozen.

5. FIRST CNN FROM SCRATCH

Here is a brief explanation of each layer:

keras.layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)): This layer rescales the input image data from its original range (0 to 255) to a range of 0 to 1, which can help to improve model performance and stability. The input_shape argument specifies the expected shape of the input data.

keras.layers.RandomRotation(0.2): This layer randomly rotates the input images by a random angle between 0 and 0.2 radians. This can help to increase the diversity of the training data and improve model robustness.

keras.layers.RandomZoom(0.5, fill_mode='reflect', interpolation='bilinear'): This layer randomly zooms in on the input images by a random factor between 0 and 0.5. The fill_mode and interpolation arguments control how the resulting image is filled in (e.g. with reflections of the existing data).

keras.layers.Conv2D(filters=32, kernel_size=2, activation='relu', input_shape = [128, 128,3]): This layer applies a 2D convolution to the input data, with 32 filters and a kernel size of 2. The activation argument specifies the activation function to use (in this case, the ReLU activation).

keras.layers.BatchNormalization(): This layer applies batch normalization to the data, which can help to stabilize the model training and improve performance.

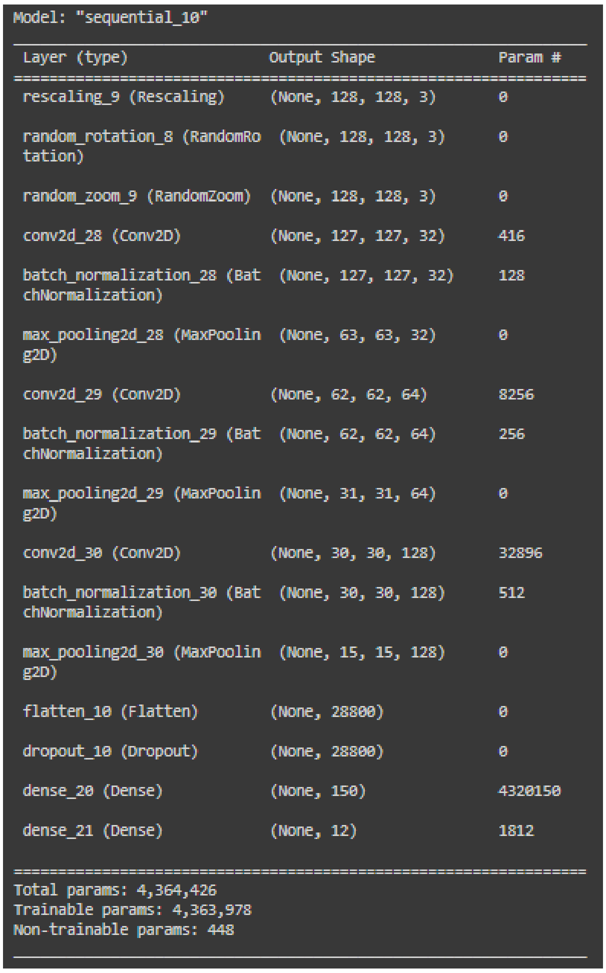
keras.layers.MaxPooling2D(pool_size=(2,2)): This layer performs max pooling on the data, which reduces the spatial dimensions of the data and can help to reduce the number of parameters in the model.

keras.layers.Flatten(): This layer flattens the data into a 1D tensor, which is necessary before passing the data through a dense layer.

keras.layers.Dropout(0.5): This layer applies dropout to the data, which can help to reduce overfitting by randomly dropping some of the data during training.

keras.layers.Dense(150, activation='relu'): This layer applies a dense (fully connected) layer to the data, with 150 units and a ReLU activation.

keras.layers.Dense(12, activation='softmax'): This is the final layer of the model, with 12 units and a softmax activation, which will output the class probabilities for the input data.



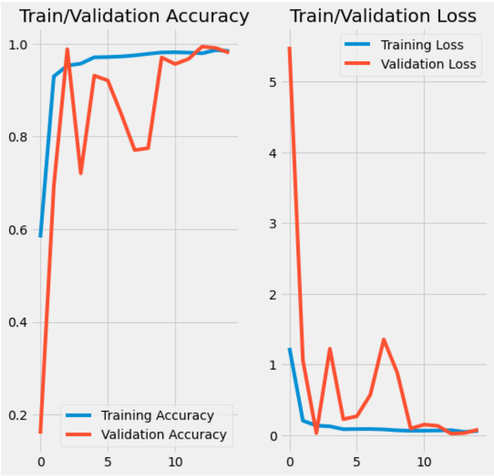
```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_9 (Rescaling)     (None, 128, 128, 3)       0

 random_rotation_8 (RandomRo  (None, 128, 128, 3)      0
 tation)

 random_zoom_9 (RandomZoom)  (None, 128, 128, 3)       0

 conv2d_28 (Conv2D)          (None, 127, 127, 32)      416

 batch_normalization_28 (Bat  (None, 127, 127, 32)     128
 chNormalization)

 max_pooling2d_28 (MaxPoolin  (None, 63, 63, 32)       0
 g2D)

 conv2d_29 (Conv2D)          (None, 62, 62, 64)        8256

 batch_normalization_29 (Bat  (None, 62, 62, 64)       256
 chNormalization)

 max_pooling2d_29 (MaxPoolin  (None, 31, 31, 64)       0
 g2D)

 conv2d_30 (Conv2D)          (None, 30, 30, 128)       32896

 batch_normalization_30 (Bat  (None, 30, 30, 128)      512
 chNormalization)

 max_pooling2d_30 (MaxPoolin  (None, 15, 15, 128)      0
 g2D)

 flatten_10 (Flatten)        (None, 28800)             0

 dropout_10 (Dropout)        (None, 28800)             0

 dense_20 (Dense)            (None, 150)               4320150

 dense_21 (Dense)            (None, 12)                1812

=================================================================
Total params: 4,364,426
Trainable params: 4,363,978
Non-trainable params: 448
_________________________________________________________________
```

Trainable params are those that are updated during train to adjust the model's weights to minimize the loss function.

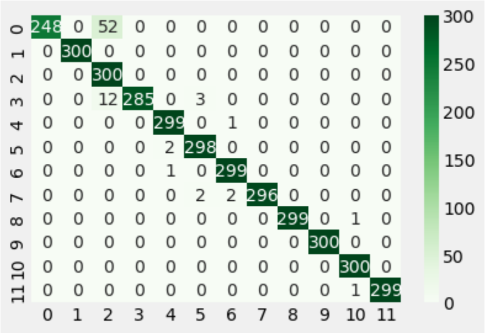
This model is compiled with **Adam** optimizer that stands for Adaptive Moment Estimation and is a variant of the gradient descent optimization algorithm. The Adam optimizer updates the model's parameters based on a combination of the gradient of the loss function with respect to the model's parameters and an exponentially weighted moving average of the gradients. This helps the optimizer to converge faster and more efficiently compared to standard gradient descent.

Moreover, the **Sparse Categorical Entropy** is used that is a loss function used in multi-class classification problems, where the number of classes is greater than two. The function measures the difference between the predicted probabilities and the true class label. Unlike categorical cross entropy loss, which requires that the target labels be one-hot encoded, sparse categorical cross entropy loss can be used with integer labels directly. It is called "sparse" because the target labels are assumed to be sparse, i.e., most elements in the target labels are zero and only one element is non-zero. The loss is used during training to drive the model towards making predictions that match the true class labels.

The model than was tested on **test set** and it achieved an accuracy of **0.97** and a loss of **0.08**.

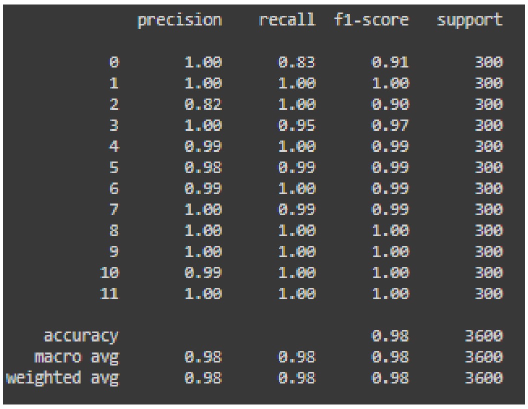


In the image above **Train/Validation Accuracy** and **Train/Validation Loss** curves are displayed and it's easy to figure out that Validation Accuracy and Validation Loss have more ups and downs in terms general behaviour. This is due to the model that oscillating around the optimal solution before finally converging to it. At the beginning of the training process, the model was complex and was fitting to the noise in the training data, resulting in a lower training loss but a higher validation loss. As the training process continued, the model started to learn the true underlying relationship in the data and started to generalize well to the validation set, resulting in a decrease in the validation loss. The variation is due to Adam optimizer that is more sensible to data noise.



	0	1	2	3	4	5	6	7	8	9	10	11
0	248	0	52	0	0	0	0	0	0	0	0	0
1	0	300	0	0	0	0	0	0	0	0	0	0
2	0	0	300	0	0	0	0	0	0	0	0	0
3	0	0	12	285	0	3	0	0	0	0	0	0
4	0	0	0	0	299	0	1	0	0	0	0	0
5	0	0	0	0	2	298	0	0	0	0	0	0
6	0	0	0	0	1	0	299	0	0	0	0	0
7	0	0	0	0	0	2	2	296	0	0	0	0
8	0	0	0	0	0	0	0	0	299	0	1	0
9	0	0	0	0	0	0	0	0	0	300	0	0
10	0	0	0	0	0	0	0	0	0	0	300	0
11	0	0	0	0	0	0	0	0	0	0	1	299

The picture above shows how the network behaved on unseen data. The bigger mistake is on the misclassification done for the 0L class, 52 samples were classified as 1L wrongly.



```
              precision    recall  f1-score   support

           0       1.00      0.83      0.91       300
           1       1.00      1.00      1.00       300
           2       0.82      1.00      0.90       300
           3       1.00      0.95      0.97       300
           4       0.99      1.00      0.99       300
           5       0.98      0.99      0.99       300
           6       0.99      1.00      0.99       300
           7       1.00      0.99      0.99       300
           8       1.00      1.00      1.00       300
           9       1.00      1.00      1.00       300
          10       0.99      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           0.98      3600
   macro avg       0.98      0.98      0.98      3600
weighted avg       0.98      0.98      0.98      3600
```

This precision, recall, f1-score and support results indicate that the model has high performance across all 12 classes. The overall accuracy is 98% which is very high. The macro average precision, recall and f1-score are all 0.98, indicating high performance for each class. The weighted average is also 0.98, indicating that the model's performance is balanced across all the classes.

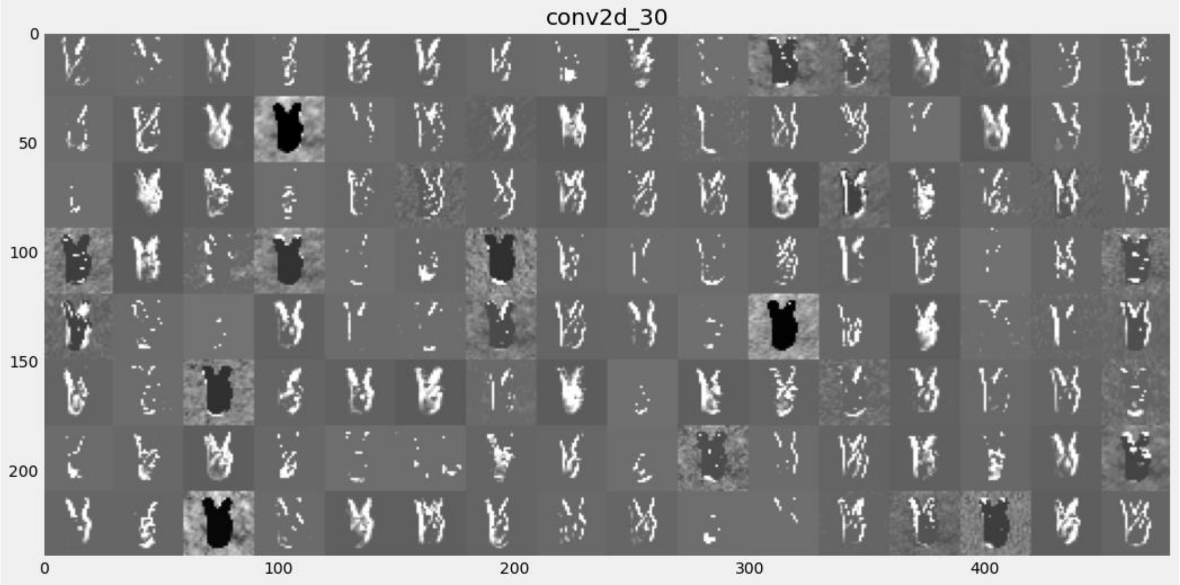
In general, these results suggest that the model can make accurate predictions for most of the classes.

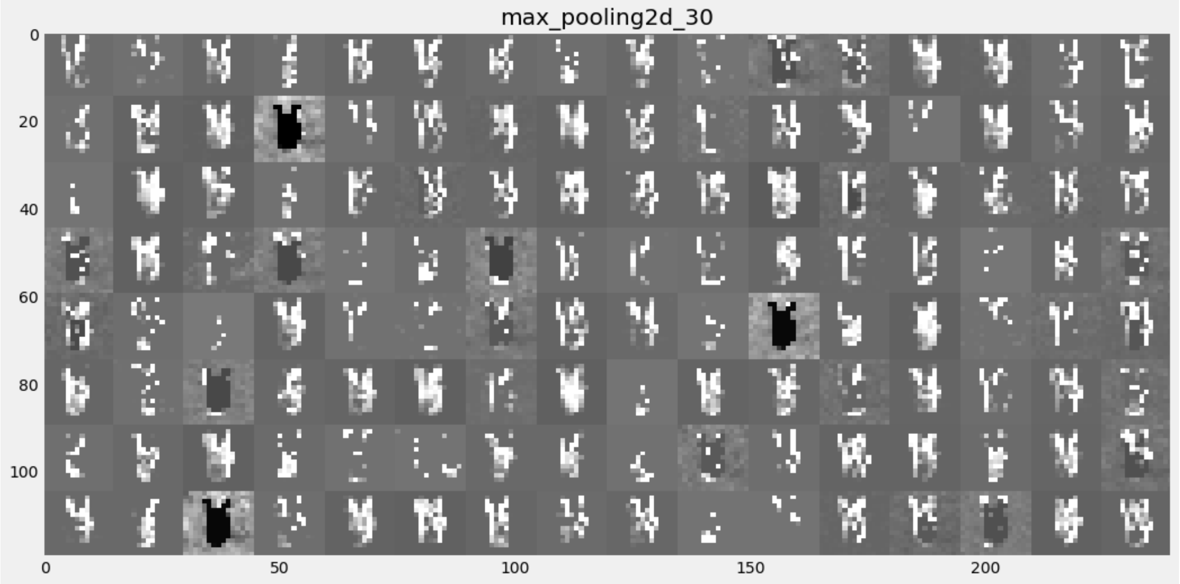
Explanability in deep learning refers to the ability to understand and interpret how a neural network makes its predictions. This can be helpful for gaining insights into the workings of the network and for identifying any potential biases or inaccuracies in the model.

One way to gain an understanding of the transformations performed by each filter in a neural network is using visualization techniques. These techniques typically involve creating visual representations of the filters and the activations they generate, which can help to uncover patterns and understand how the network is processing input data.

For example, layer activations visualization can be used to get an idea of what kind of features each filter is learning to detect in the input images.

By exploring the explainability of a neural network, it can gain insights into how it is processing information and identify areas for improvement, which can lead to a better understanding of the model and more accurate predictions.





In the images above transformations are displayed for the last conv2d layer and the max pooling layer linked to it. Is it possible to see that the image is still recognizable since the network is not so deep.

6. SECOND CNN FROM SCRATCH

Here the CNN structure that was used:

Rescaling layer: This layer scales the input image data by dividing it by 255. This is done to normalize the data to the range [0,1].

RandomRotation layer: This layer randomly rotates the input image by a specified degree (0.2 degrees in this case). This is done as data augmentation to increase the robustness of the model.

RandomZoom layer: This layer randomly zooms in and out of the input image. The fill_mode parameter sets the value to be used for padding the image after zooming, the **'reflect'** value sets the padding to reflect the input image, and the **'bilinear'** value sets the method for resizing the image to be bilinear interpolation.

Conv2D layer: This is the first Convolutional Neural Network (ConvNet) layer. It applies a set of filters to the input image to extract features. The **filters** parameter specifies the number of filters to be used (32 filters in this case). The **kernel_size** parameter specifies the size of the filter (2x2 in this case) and the **activation** parameter sets the activation function to be used (**'relu'** in this case).

BatchNormalization layer: This layer normalizes the activations of the previous layer. It helps in faster convergence of the model and regularization.

MaxPooling2D layer: This is the first Max Pooling layer. It reduces the spatial resolution of the input image by applying max pooling to non-overlapping windows. The **pool_size** parameter specifies the size of the pooling window (2x2 in this case).

Conv2D layer: This is the second ConvNet layer. It applies another set of filters to the output of the previous layer to extract more complex features.

BatchNormalization layer: This layer normalizes the activations of the previous layer.

MaxPooling2D layer: This is the second Max Pooling layer.

Conv2D layer: This is the third ConvNet layer.

BatchNormalization layer: This layer normalizes the activations of the previous layer.

MaxPooling2D layer: This is the third Max Pooling layer.

Conv2D layer: This is the fourth ConvNet layer.

BatchNormalization layer: This layer normalizes the activations of the previous layer.

MaxPooling2D layer: This is the fourth Max Pooling layer.

Conv2D layer: This is the fifth ConvNet layer.

BatchNormalization layer: This layer normalizes the activations of the previous layer.

MaxPooling2D layer: This is the fifth Max Pooling layer.

Flatten layer: This layer flattens the activations from the previous layer into a one-dimensional array.

Dense layer: This is the first fully-connected (dense) layer. It applies a set of weights to the activations from the previous layer. The **150** parameter specifies the number of neurons in this layer and use ReLU

Then there is the last final **Dense** of the model, with 12 units and a softmax activation, which will output the class probabilities for the input data.

By looking at the summary of the network, here are some observations and considerations one can make:

1. **Preprocessing**: The first layer rescales the pixel values of the input image by dividing each value by 255. This normalizes the pixel values and ensures that they lie in the range of 0 and 1.

2. **Data augmentation**: The second layer performs random rotations on the image to increase the diversity of the data. The third layer performs random zoom on the image to further increase the diversity of the data. These steps help prevent overfitting by increasing the size and diversity of the training data.

3. **Convolutional layers**: There are six convolutional layers in the network. Each convolutional layer has a specified number of filters (32, 64, 128, 256, 512, 512). The size of the filters is 2x2 and the activation function used is 'relu'. Convolutional layers are the building blocks of convolutional neural networks, and they perform feature extraction by sliding filters over the input image and creating feature maps.

4. **Batch normalization**: The network includes batch normalization layers after each of the convolutional layers. Batch normalization helps to regularize the network by standardizing the inputs to each layer, reducing the internal covariate shift and accelerating the training process.

5. **Max pooling layers**: There are six max pooling layers in the network. The pool size is 2x2. Max pooling reduces the spatial dimensions of the feature maps, increasing the translation invariance of the network and reducing the number of parameters.

6. **Flatten layer**: After the final max pooling layer, the network has a flatten layer that flattens the 2D feature maps into a 1D vector, which is then used as input to the dense layers.

7. **Dense layers**: The network has two dense layers. The first dense layer has 150 neurons and the activation function is 'relu'. The second dense layer has 12 neurons and the activation function is 'softmax', which is commonly used for multiclass classification. The softmax activation function returns the probability of each class, allowing the network to make predictions based on the class with the highest probability.

In general, this network appears to be a standard convolutional neural network architecture for image classification, with the added pre-processing steps and data augmentation.
In both architectures ReLU usage was preferred since it has several benefits compared to other activation functions, including:

1. **Simplicity**: ReLU is a simple activation function and is easy to implement. It only requires a threshold value and binary thresholding, making it computationally efficient.

2. **Non-linearity**: ReLU introduces non-linearity into the neural network, which is important for modeling complex relationships in the data.

3. **Faster convergence**: ReLU has been shown to converge faster than other activation functions like sigmoid and tanh. This is because it does not saturate for large input values, which means that the gradients do not vanish, and the learning can proceed faster.

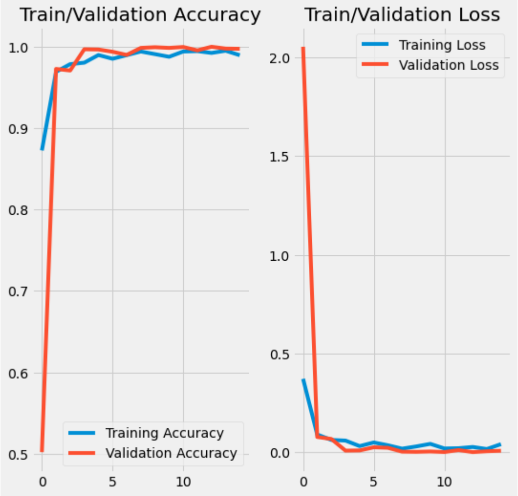
4. **Sparsity**: ReLU has the property of generating sparse activations, meaning that most neurons are either 0 or 1. This can help to reduce the computational load and make the neural network more efficient.

5. **No vanishing gradients**: Unlike sigmoid and tanh activation functions, ReLU does not suffer from the vanishing gradients problem. This means that the gradients do not become extremely small, which makes it easier to train deep neural networks.

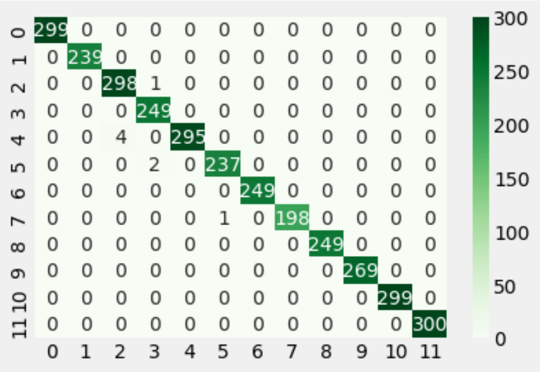
Note that in both networks **Early stopping criteria** was used to top training when a monitored quantity (here, the accuracy) has stopped improving. The **patience** argument determines the number of epochs to wait before ending the training. If during this time the accuracy does not improve, the training will be stopped. The **mode** argument is set to "max" so that the training will stop when the accuracy stops increasing. The **restore_best_weights** argument is set to **True** so that the best weights (weights with the highest accuracy) will be restored after the training has stopped. This helps ensure that the model's performance will not degrade over time due to overfitting.
Here the number of params of the second CNN:

- Total params: 2,061,962
- Trainable params: 2,058,954
- Non-trainable params: 3,008

After the training, the model was tested on unseen data with the following performance: **0.99%** of Accuracy and **0.008** of loss that is less than the loss of the first model.



Above the curves of accuracy and validation are displayed and w.r.t to the first model it's easy to observe that there are no oscillations around minima so the model has found instantly the minimum, moreover there is no overfitting situation since the **Validation Loss** is less than the **Training Loss** and the same is for the **Accuracies** so the model has generalized well since the very beginning of the model construction.
It's important to remark that some Data Augmentation was inserted in both Networks to prevent overfitting situations.



	0	1	2	3	4	5	6	7	8	9	10	11
0	299	0	0	0	0	0	0	0	0	0	0	0
1	0	239	0	0	0	0	0	0	0	0	0	0
2	0	0	298	1	0	0	0	0	0	0	0	0
3	0	0	0	249	0	0	0	0	0	0	0	0
4	0	0	4	0	295	0	0	0	0	0	0	0
5	0	0	0	2	0	237	0	0	0	0	0	0
6	0	0	0	0	0	0	249	0	0	0	0	0
7	0	0	0	0	0	1	0	198	0	0	0	0
8	0	0	0	0	0	0	0	0	249	0	0	0
9	0	0	0	0	0	0	0	0	0	269	0	0
10	0	0	0	0	0	0	0	0	0	0	299	0
11	0	0	0	0	0	0	0	0	0	0	0	300

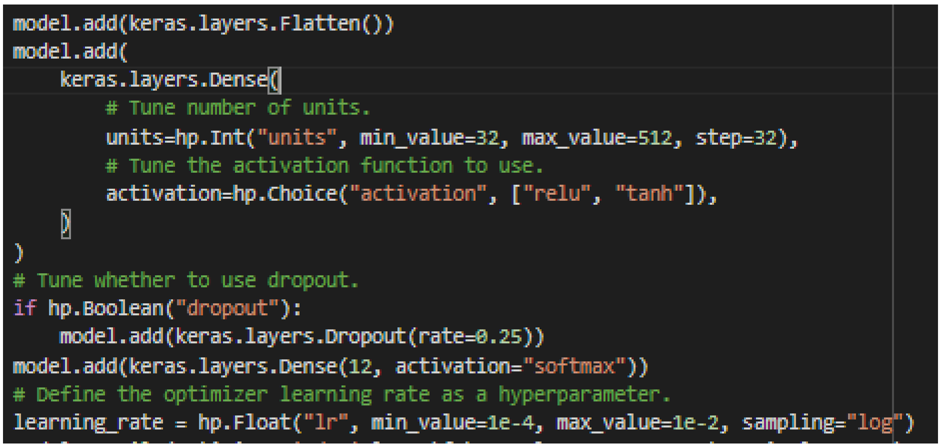
7. SECOND CNN HYPERTUNING

For the Hyper Tuning the entire second CNN was taken, and some parameters were tuned at the end of the CNN. The function uses the **Hyperparameters** object from the **hyperas** library to define several hyperparameters that can be tuned during the model training process. The number of units in the dense layer, the activation function used in the dense layer, whether to use dropout, and the learning rate of the optimizer are all defined as hyperparameters. This allows the model to be trained using a technique such as random search or Bayesian optimization to find the best set of hyperparameters for a given problem.

Random Search was preferred to **Grid Search.** The first one is more efficient from a time perspective, since not all parameters combination are tested. The second type guarantees a complete combination parameter coverage but takes more time, so the first solution is better if retrieve a good approximation faster is required.

More in general the **Hyper Tuning** can lead to better use of resources, Increased robustness and better generalization which means the risk of overfitting is lower and the model can perform better on unseen data.

More in deep this was the Hyper Tuning in the CNN code:



```
model.add(keras.layers.Flatten())
model.add(
    keras.layers.Dense(
        # Tune number of units.
        units=hp.Int("units", min_value=32, max_value=512, step=32),
        # Tune the activation function to use.
        activation=hp.Choice("activation", ["relu", "tanh"]),
    )
)
# Tune whether to use dropout.
if hp.Boolean("dropout"):
    model.add(keras.layers.Dropout(rate=0.25))
model.add(keras.layers.Dense(12, activation="softmax"))
# Define the optimizer learning rate as a hyperparameter.
learning_rate = hp.Float("lr", min_value=1e-4, max_value=1e-2, sampling="log")
```

The **relu** activation function returns the input if it is positive and returns 0 if the input is negative. This helps the network to learn non-linear relationships between the inputs and outputs. The **tanh** activation function, on the other hand, returns the hyperbolic tangent of the input, which maps the input to the range of -1 to 1.

If model runs with the **tanh** activation function instead of **relu**, it can have a different impact on the model performance and the speed of convergence, depending on the dataset and the problem being solved. The **tanh** activation function may lead to slower convergence compared to the **relu** activation function, as it has a steeper saturation region around the minimum and maximum values. However, the **tanh** function can be useful in some cases as it can provide smoother and more centred outputs.

Number of units in the dense layer: The number of neurons in the dense layer that follows the convolutional layers is being tuned. The **hp.Int("units", min_value=32, max_value=512, step=32)** line of code defines the search space for this hyperparameter. The search will range from 32 to 512, with a step size of 32.

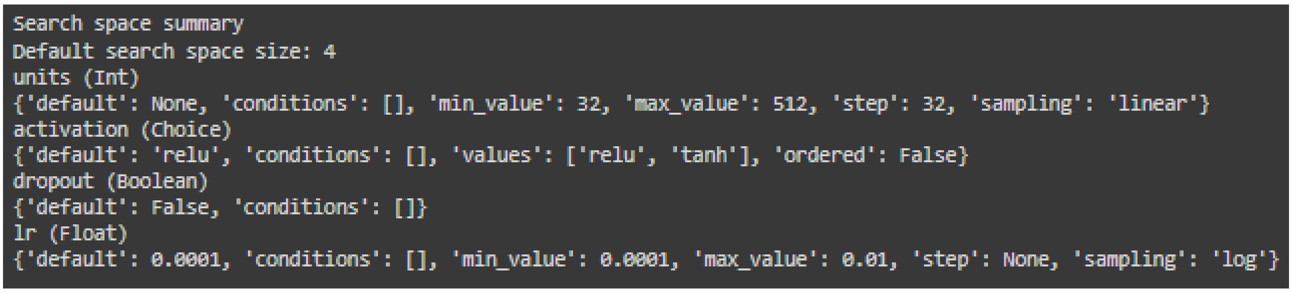
Activation function for the dense layer: The activation function used in the dense layer is being tuned. The **hp.Choice("activation", ["relu", "tanh"])** line of code defines the two choices for this hyperparameter, either 'relu' or 'tanh'.

Dropout layer: The presence of a dropout layer after the dense layer is being decided by the hyperparameter tuning. The **hp.Boolean("dropout")** line of code defines this hyperparameter, either True or False. If True, a dropout layer will be added with a dropout rate of 0.25.

Learning rate for the Adam optimizer: The learning rate used by the Adam optimizer is being tuned. The **hp.Float("lr", min_value=1e-4, max_value=1e-2, sampling="log")** line of code defines the search space for this hyperparameter. The search will range from a minimum value of 1e-4 to a maximum of 1e-2 and the sampling is logarithmic.

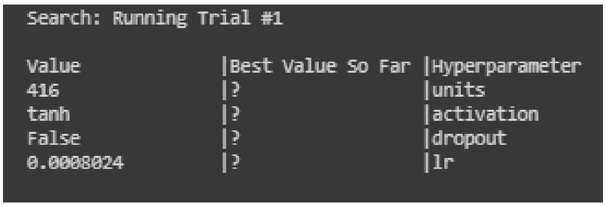
The **idea behind hyperparameter tuning** is to find the best combination of these hyperparameters that result in the best performance for the model on the task at hand. By tuning these hyperparameters, the model can improve its accuracy, reduce overfitting or underfitting, and generalize better to new data. The result of hyperparameter tuning can be seen in the difference in the misclassification of the classes. If the misclassification is improved from the main diagonal to other sparse blocks, it is a sign that the model is generalizing better to new data.

The following is the search space of the **Random Search**

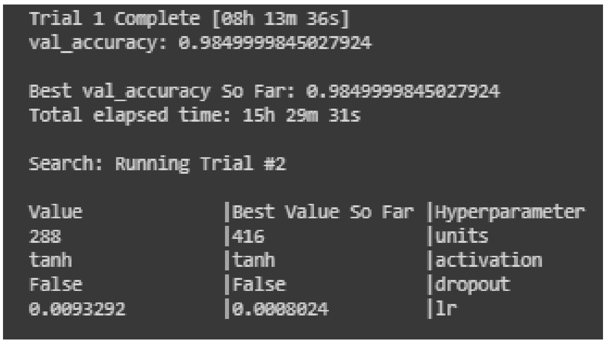


```
Search space summary
Default search space size: 4
units (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
activation (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'tanh'], 'ordered': False}
dropout (Boolean)
{'default': False, 'conditions': []}
lr (Float)
{'default': 0.0001, 'conditions': [], 'min_value': 0.0001, 'max_value': 0.01, 'step': None, 'sampling': 'log'}
```

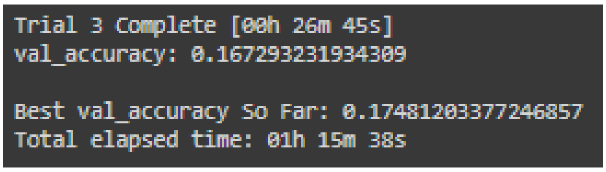
For time limitation only three trials were done and with the following parameters:



```
Search: Running Trial #1

Value             |Best Value So Far |Hyperparameter
416               |?                 |units
tanh              |?                 |activation
False             |?                 |dropout
0.0008024         |?                 |lr
```

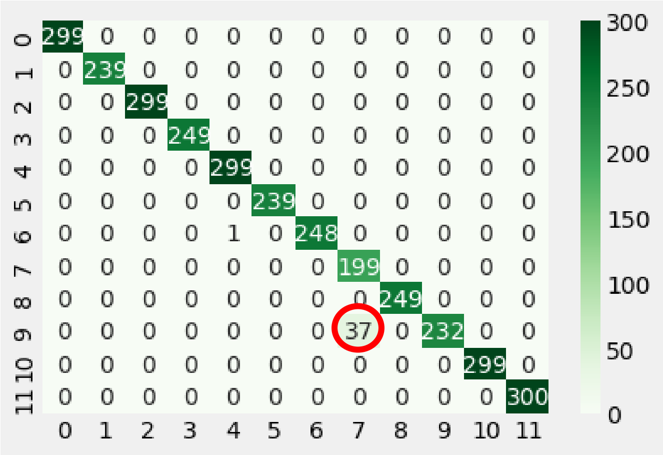


```
Trial 1 Complete [08h 13m 36s]
val_accuracy: 0.9849999845027924

Best val_accuracy So Far: 0.9849999845027924
Total elapsed time: 15h 29m 31s

Search: Running Trial #2

Value             |Best Value So Far |Hyperparameter
288               |416               |units
tanh              |tanh              |activation
False             |False             |dropout
0.0093292         |0.0008024         |lr
```

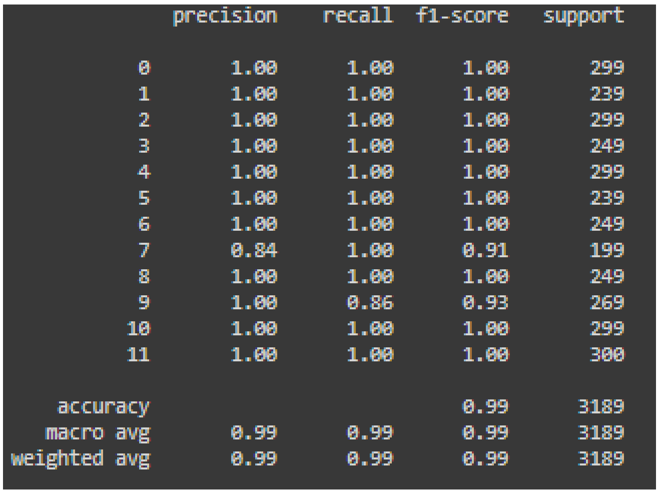


```
Trial 3 Complete [00h 26m 45s]
val_accuracy: 0.167293231934309

Best val_accuracy So Far: 0.17481203377246857
Total elapsed time: 01h 15m 38s
```

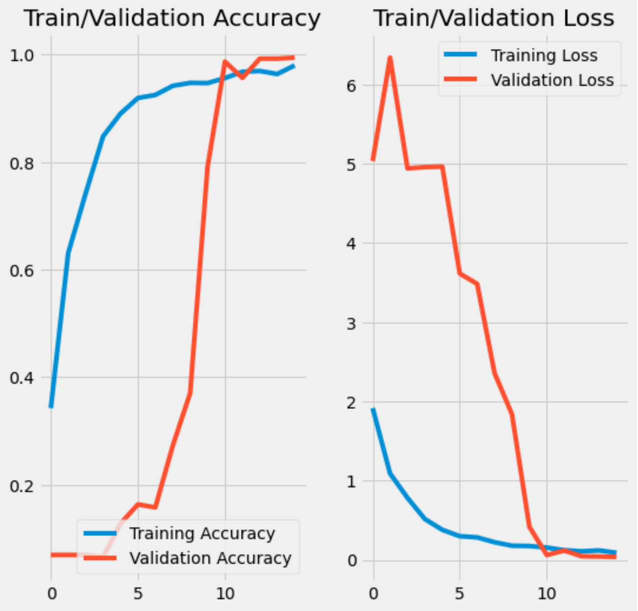
Parameters tuned in the first trial were taken to build the entire model.
From a **performance point of view** the Network achieved a **98%** of **Accuracy** and a **0.03** of loss that in terms of pure performance is worst w.r.t. to the Non-Hyper Tuned.



	0	1	2	3	4	5	6	7	8	9	10	11
0	299	0	0	0	0	0	0	0	0	0	0	0
1	0	239	0	0	0	0	0	0	0	0	0	0
2	0	0	299	0	0	0	0	0	0	0	0	0
3	0	0	0	249	0	0	0	0	0	0	0	0
4	0	0	0	0	299	0	0	0	0	0	0	0
5	0	0	0	0	0	239	0	0	0	0	0	0
6	0	0	0	0	1	0	248	0	0	0	0	0
7	0	0	0	0	0	0	0	199	0	0	0	0
8	0	0	0	0	0	0	0	0	249	0	0	0
9	0	0	0	0	0	0	0	37	0	232	0	0
10	0	0	0	0	0	0	0	0	0	0	299	0
11	0	0	0	0	0	0	0	0	0	0	0	300



```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       299
           1       1.00      1.00      1.00       239
           2       1.00      1.00      1.00       299
           3       1.00      1.00      1.00       249
           4       1.00      1.00      1.00       299
           5       1.00      1.00      1.00       239
           6       1.00      1.00      1.00       249
           7       0.84      1.00      0.91       199
           8       1.00      1.00      1.00       249
           9       1.00      0.86      0.93       269
          10       1.00      1.00      1.00       299
          11       1.00      1.00      1.00       300

    accuracy                           0.99      3189
   macro avg       0.99      0.99      0.99      3189
weighted avg       0.99      0.99      0.99      3189
```



It’s possible to see how the model misclassifies more the non-tuned version. Until the tenth epoch the model seems to work bad but then is aligned to what was expected.

8. CNN USING RESNET-50

ResNet50 architecture is used to test a pre-trained network on the dataset project by means of **Transfer Learning.** The main challenge here is to set up a working ecosystem even if the dataset is not so deep like maybe ResNet50 would like to have, in fact it has almost 50 million of trainable parameters that maybe are not so suitable for a 20k image dataset.

Two optimizers are used here, with and without **Data Augmentation.**

When **no data augmentation** is applied the network is not frozen and this allow layers to be completely trainable during the training phase that means that all weights associated to these layers can be tailored on new data. This could be an advantage if new data are different from those on which the pretrained network was trained since network could adapt itself to fit completely to new features. Non-frozen technique has some drawbacks like overfitting or slower training time. In this situation the ResNet50 network is used as **Feature reuse** since it extracts feature and then the final layers are modelled to answer to this specific task.

Infact a method **extract_feature_with_resnet** has been defined and then final custom layers are attached to ResNet network to give a 12 classes output. Feature reuse has a lot of advantages like **reduced computational cost** since using a pre-trained model as a feature extractor, you can leverage the learned features in the lower layers of the model to represent the input data in a higher-level, more abstract space, allowing you to solve the problem with less data and computational resources compared to training a model from scratch; **improve performance with limited data since** it allows you to take advantage of the pre-trained model's understanding of the task at hand. This can help to improve the performance of the model and reduce the risk of overfitting.

With **data augmentation**, it was decided to freeze all layers of the network. By freezing all the layers all weights associated to those layers cannot be modified. It allows more **stability** since knowledge is preserved during training.

Drawbacks could be **poor performance** since by freezing all layers the network cannot fit well the new data if data are very different from those on which it was pretrained and could be **overfitting** that is a consequence of poor fitting on the new data.

The **augmented networks with the Adam and SGD**, eventually are **Fine Tuned** that is a concept completely different from the **Hypertuning** already seen in the first part of the project.

Fine-tuning consists of changing the weights of a pre-trained neural network to a new data set. This can be done by freezing some layers of the network and unfreezing others, thus allowing the network to adapt to the new data. Fine-tuning is a way of using the knowledge gained from a pre-trained neural network to improve performance on a new dataset.

Hyperparameter tuning consists of optimising the values of certain parameters that control the behaviour of a neural network during training. These parameters are not part of the network weights but influence how the weights are changed during training. For example, the values of learning rate, batch size, number of epochs, and so on, are all examples of hyperparameters. Hyperparameter tuning is a way of improving the performance of a neural network during training.

The pretrained network used is the **ResNet50,** that is a deep residual network architecture designed to address the problem of vanishing gradients in very deep neural networks. It was introduced in 2015 by Microsoft researchers and has since become a popular architecture for computer vision tasks, such as image classification.

The ResNet50 architecture is built using residual blocks, which are the building blocks of the network. A residual block is a building block that adds the input to the output of the block, and it consists of several convolutional layers and shortcut connections. The shortcut connections allow the network to skip over the convolutional layers and use the identity function, which can help prevent the vanishing gradient problem and enable the network to learn more complex features.

The ResNet50 architecture consists of 50 layers and is trained on large datasets such as ImageNet. The architecture uses multiple residual blocks with different numbers of convolutional layers, and the blocks are stacked on top of each other to form the network. The first few layers of the network consist of standard convolutional layers that are designed to extract low-level features from the input image. The later layers consist of residual blocks that are designed to learn higher-level features.

The ResNet50 architecture has been trained on large datasets, and it has been shown to perform well on a variety of computer vision tasks. It has become a popular starting point for many computer vision projects, and it is often used as a baseline for comparison with other architectures.

The ResNet50 convolutional base was instantiated and then a classifier was applied on top of it.

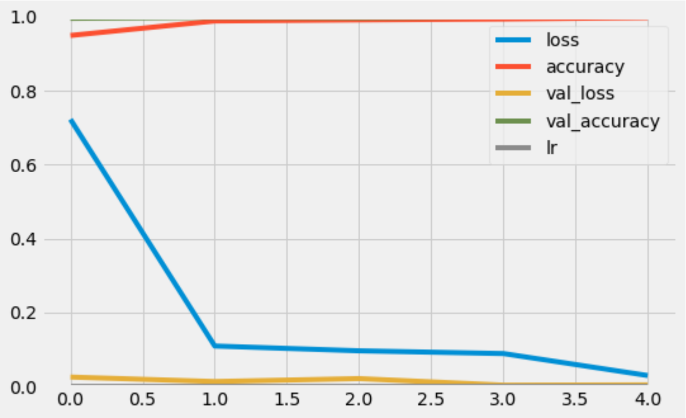
From now on the **ReduceLROnPlateau** is used that is a function that reduces the learning rate of the optimizer when the validation loss has stopped improving. This helps to avoid overfitting and speeding up convergence.

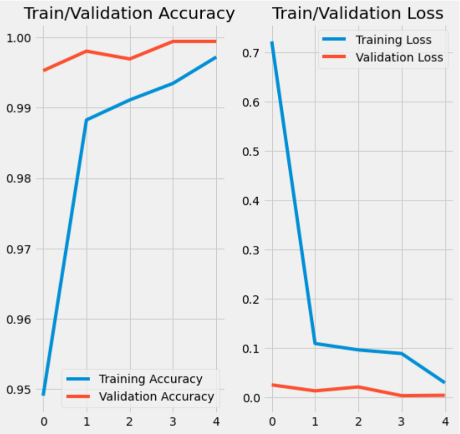
The ReduceLROnPlateau callback works by monitoring the validation loss during training. If the validation loss does not improve for a certain number of epochs (as specified by the "patience" parameter), the learning rate of the optimizer is reduced by a factor specified by the "factor" parameter. The reduction in learning rate continues until the learning rate reaches the minimum value specified by the "min_lr" parameter.

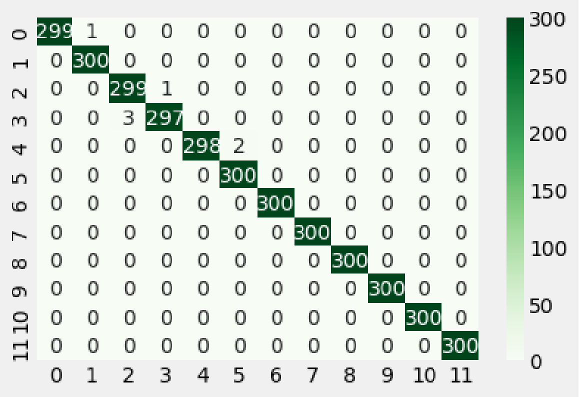
The ReduceLROnPlateau callback can be a useful tool to improve the performance of a deep learning model, as it allows to fine-tune the learning rate based on the progress of the model during training. By reducing the learning rate when the validation loss has stopped improving, you can avoid overfitting and speed up convergence, which can lead to better performance on the validation set and the test set.

Feature extraction with ResNet50 function was made and then the optimizers with and without data augmentation were applied.

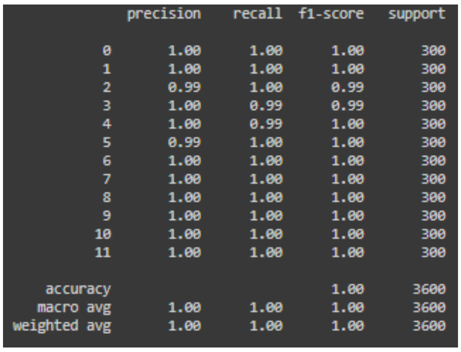
- **Adam without Data Augmentation:**





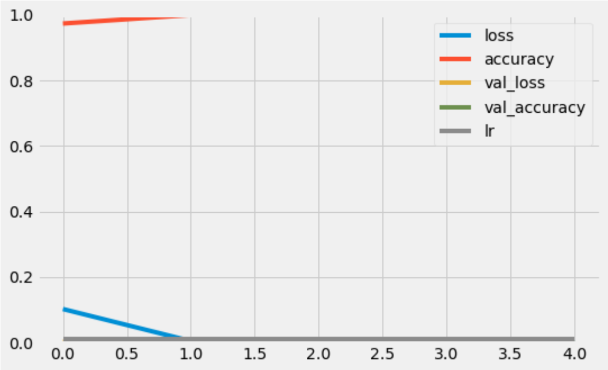


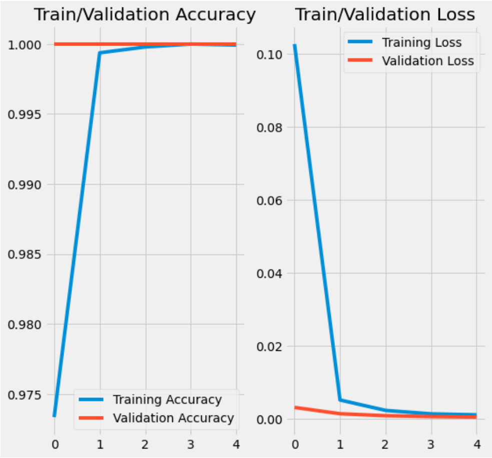
	0	1	2	3	4	5	6	7	8	9	10	11
0	299	1	0	0	0	0	0	0	0	0	0	0
1	0	300	0	0	0	0	0	0	0	0	0	0
2	0	0	299	1	0	0	0	0	0	0	0	0
3	0	0	3	297	0	0	0	0	0	0	0	0
4	0	0	0	0	298	2	0	0	0	0	0	0
5	0	0	0	0	0	300	0	0	0	0	0	0
6	0	0	0	0	0	0	300	0	0	0	0	0
7	0	0	0	0	0	0	0	300	0	0	0	0
8	0	0	0	0	0	0	0	0	300	0	0	0
9	0	0	0	0	0	0	0	0	0	300	0	0
10	0	0	0	0	0	0	0	0	0	0	300	0
11	0	0	0	0	0	0	0	0	0	0	0	300

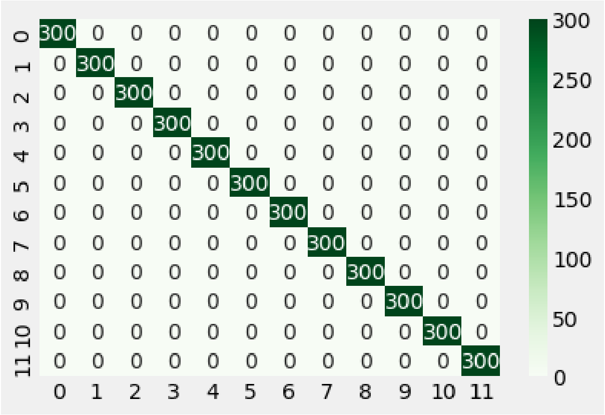


```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       300
           1       1.00      1.00      1.00       300
           2       0.99      1.00      0.99       300
           3       1.00      0.99      0.99       300
           4       1.00      0.99      1.00       300
           5       0.99      1.00      1.00       300
           6       1.00      1.00      1.00       300
           7       1.00      1.00      1.00       300
           8       1.00      1.00      1.00       300
           9       1.00      1.00      1.00       300
          10       1.00      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           1.00      3600
   macro avg       1.00      1.00      1.00      3600
weighted avg       1.00      1.00      1.00      3600
```

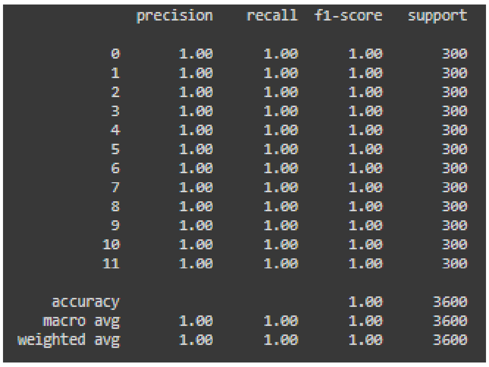
- **SGD without Data Augmentation**







	0	1	2	3	4	5	6	7	8	9	10	11
0	300	0	0	0	0	0	0	0	0	0	0	0
1	0	300	0	0	0	0	0	0	0	0	0	0
2	0	0	300	0	0	0	0	0	0	0	0	0
3	0	0	0	300	0	0	0	0	0	0	0	0
4	0	0	0	0	300	0	0	0	0	0	0	0
5	0	0	0	0	0	300	0	0	0	0	0	0
6	0	0	0	0	0	0	300	0	0	0	0	0
7	0	0	0	0	0	0	0	300	0	0	0	0
8	0	0	0	0	0	0	0	0	300	0	0	0
9	0	0	0	0	0	0	0	0	0	300	0	0
10	0	0	0	0	0	0	0	0	0	0	300	0
11	0	0	0	0	0	0	0	0	0	0	0	300



```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       300
           1       1.00      1.00      1.00       300
           2       1.00      1.00      1.00       300
           3       1.00      1.00      1.00       300
           4       1.00      1.00      1.00       300
           5       1.00      1.00      1.00       300
           6       1.00      1.00      1.00       300
           7       1.00      1.00      1.00       300
           8       1.00      1.00      1.00       300
           9       1.00      1.00      1.00       300
          10       1.00      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           1.00      3600
   macro avg       1.00      1.00      1.00      3600
weighted avg       1.00      1.00      1.00      3600
```

For **Adam** a test accuracy of 99% was registered with a loss of 0.02, while for **SGD** an accuracy of 100% with a loss of 0.0012**.**

It was decided to compare **SGD** and **Adam** because they are both optimizations' algorithms used for Neural Networks training.

SGD is a simple algorithm that uses only one setting value, the learning rate, to control how fast the model adapts to the training data. It has a simple implementation and is efficient in terms of computational resources but may be more sensitive to setting values and local minima than other optimisation algorithms.

Adam, on the other hand, is a more advanced algorithm that uses different settings to control the speed of model fitting. Adam considers the current direction of the gradient, the average gradient in the last training steps and the variance of the gradient to control the speed of model fitting. This makes it less sensitive to local minima than SGD and more suitable for complex models.

Now for the **Data Augmentation** applied to the two different optimizers all layers have been freeze. Freezing the layers of a pre-trained neural network means freezing the weights of those layers during the training process of a new model. This is useful when use a pre-trained network as the basis for a new model is wanted, but only want to train some of the innermost layers to fit the new training data.

Freeing the outer layers of a pre-trained network is useful because these layers have often learnt to recognise general features of the data, such as shapes, textures, and patterns. This information is often relevant for many different applications, so it is useful to keep this information fixed while training the new model.

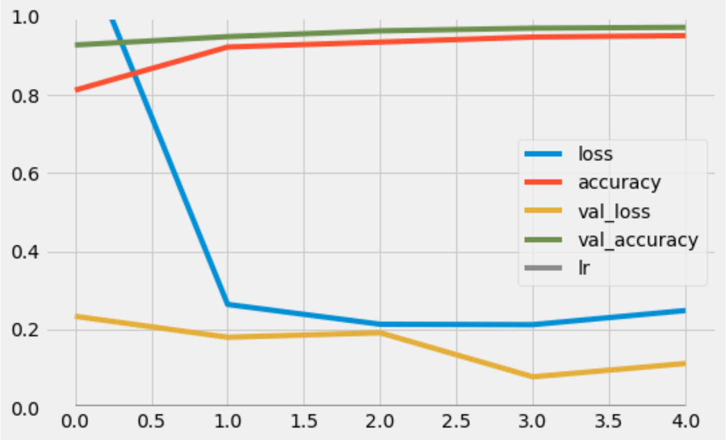
On the other hand, the innermost layers are often more specialised for a particular application, so it is often useful to train them on new data to adapt the model to the new application.

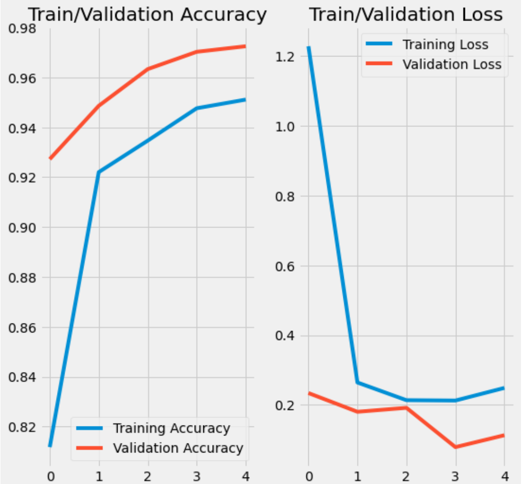
In summary, the technique of freezing layers in a pre-trained neural network is useful for exploiting the information learned from a pre-trained model in a new model, retaining some of the features learned from the original model and training only the parts best suited to the new application.

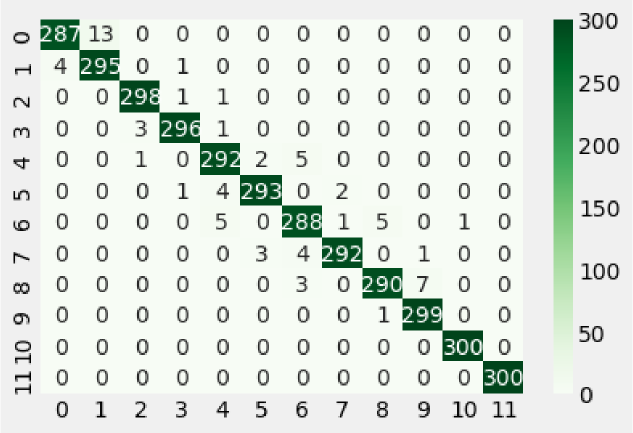
Freezing layers here is done to compare performance obtained from both system in the two optimizers configurations. For now, training the network without the augmentation lead to a good performance system, so now the goal is to train again the model in this other configuration.
From a theoretical perspective these are some observations that can be done by put those techniques side by side:

- Overfitting: If the unfrozen pre-trained neural network suffers from overfitting, i.e., if the results on the training data are very good but the results on the test data are poor, this could be a sign that the network is learning too much information specific to the training data and is not able to generalise well to the new data. In this case, layer freezing could help prevent overfitting.
- Performance: If the frozen pre-trained neural network performs worse than the unfrozen network, it could be a sign that the frozen layers are not suitable for the new training data. In this case, it may be necessary to train all layers for optimal performance.
- Training time: Freezing the layers may also have an impact on the training time. If the frozen neural network takes less time to train than the unfrozen network, it may be an advantage for some applications with time or computing power limitations.

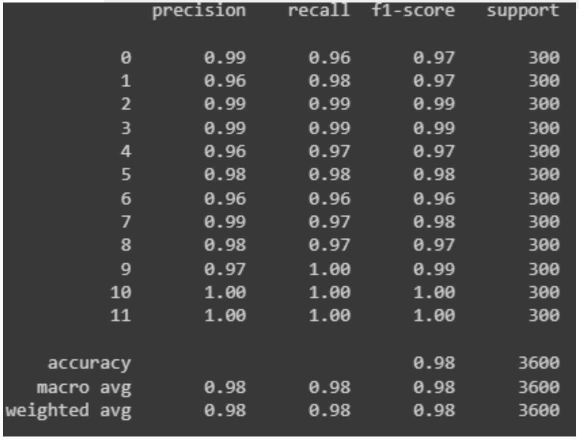
- **Adam with Data Augmentation**







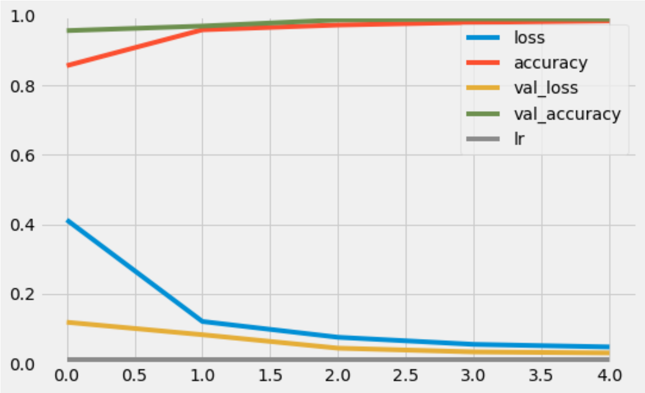
	0	1	2	3	4	5	6	7	8	9	10	11
0	287	13	0	0	0	0	0	0	0	0	0	0
1	4	295	0	1	0	0	0	0	0	0	0	0
2	0	0	298	1	1	0	0	0	0	0	0	0
3	0	0	3	296	1	0	0	0	0	0	0	0
4	0	0	1	0	292	2	5	0	0	0	0	0
5	0	0	0	1	4	293	0	2	0	0	0	0
6	0	0	0	0	5	0	288	1	5	0	1	0
7	0	0	0	0	0	3	4	292	0	1	0	0
8	0	0	0	0	0	0	3	0	290	7	0	0
9	0	0	0	0	0	0	0	0	1	299	0	0
10	0	0	0	0	0	0	0	0	0	0	300	0
11	0	0	0	0	0	0	0	0	0	0	0	300

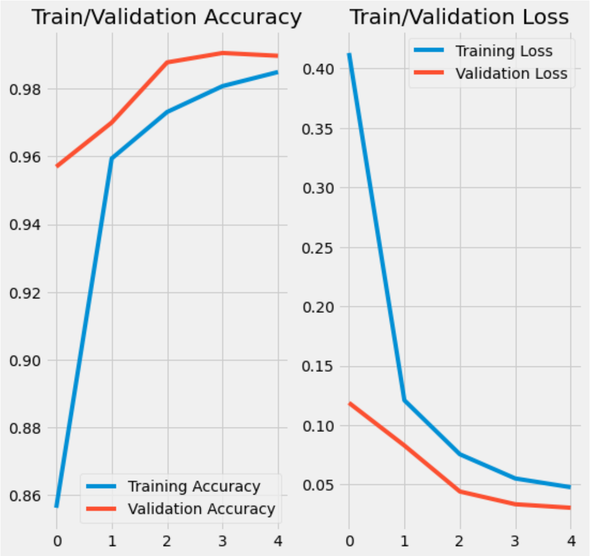


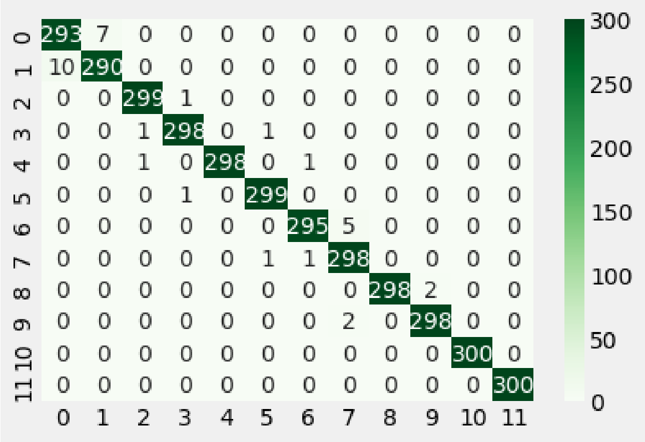
```
              precision    recall  f1-score   support

           0       0.99      0.96      0.97       300
           1       0.96      0.98      0.97       300
           2       0.99      0.99      0.99       300
           3       0.99      0.99      0.99       300
           4       0.96      0.97      0.97       300
           5       0.98      0.98      0.98       300
           6       0.96      0.96      0.96       300
           7       0.99      0.97      0.98       300
           8       0.98      0.97      0.97       300
           9       0.97      1.00      0.99       300
          10       1.00      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           0.98      3600
   macro avg       0.98      0.98      0.98      3600
weighted avg       0.98      0.98      0.98      3600
```

The model behaved well even if a small and continuous gap is visible between the training and validation accuracies curves. This could be a sign of a light overfitting situation; The gap is of 2 tenth so it's not an issue. The model, however, performed on the set with a **97%** of Accuracies and a loss of **0.09.**

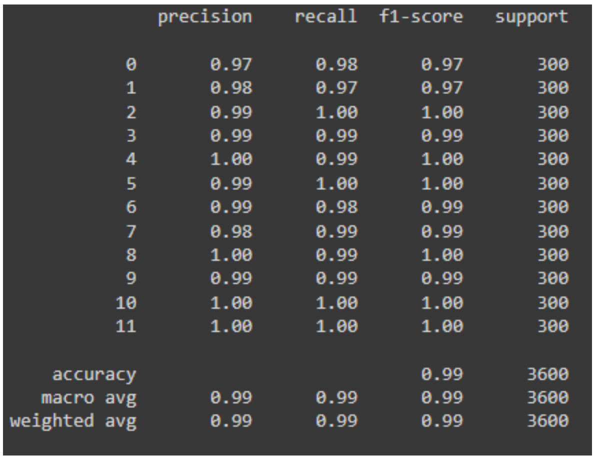
- **SGD with Data Augmentation**







	0	1	2	3	4	5	6	7	8	9	10	11
0	293	7	0	0	0	0	0	0	0	0	0	0
1	10	290	0	0	0	0	0	0	0	0	0	0
2	0	0	299	1	0	0	0	0	0	0	0	0
3	0	0	1	298	0	1	0	0	0	0	0	0
4	0	0	1	0	298	0	1	0	0	0	0	0
5	0	0	0	1	0	299	0	0	0	0	0	0
6	0	0	0	0	0	0	295	5	0	0	0	0
7	0	0	0	0	0	1	1	298	0	0	0	0
8	0	0	0	0	0	0	0	0	298	2	0	0
9	0	0	0	0	0	0	0	2	0	298	0	0
10	0	0	0	0	0	0	0	0	0	0	300	0
11	0	0	0	0	0	0	0	0	0	0	0	300



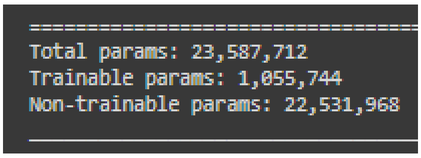
```
              precision    recall  f1-score   support

           0       0.97      0.98      0.97       300
           1       0.98      0.97      0.97       300
           2       0.99      1.00      1.00       300
           3       0.99      0.99      0.99       300
           4       1.00      0.99      1.00       300
           5       0.99      1.00      1.00       300
           6       0.99      0.98      0.99       300
           7       0.98      0.99      0.99       300
           8       1.00      0.99      1.00       300
           9       0.99      0.99      0.99       300
          10       1.00      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           0.99      3600
   macro avg       0.99      0.99      0.99      3600
weighted avg       0.99      0.99      0.99      3600
```

This model performed better than the previous one and the gap between curves is smaller indicating a good generalization level.

9. FINETUNING ON CNN RESNET BASED

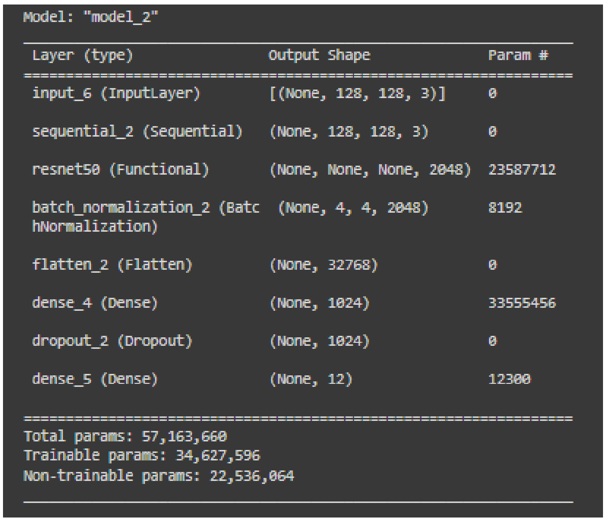
Fine-tuning is a popular approach in transfer learning, where a pre-trained neural network is adapted to a new task by updating some of its layers. The process of fine-tuning involves keeping some layers of the pre-trained network frozen, while others are updated during training. Unfrozen layers in fine-tuning refer to those layers of the pre-trained network that are updated during training. The choice of which layers to unfreeze depends on several factors, including **training dataset size, similarity** between pre-training task and the target task, and **network architecture**.

The following picture shows how many params are trainable after six unfrozen layers for RESNET50 network.



```
==================================
Total params: 23,587,712
Trainable params: 1,055,744
Non-trainable params: 22,531,968
__________________________________
```

Some changes are required to adapt the pretrained network to specific task of the project, so the following custom network was created starting from the unfrozen RESNET50.



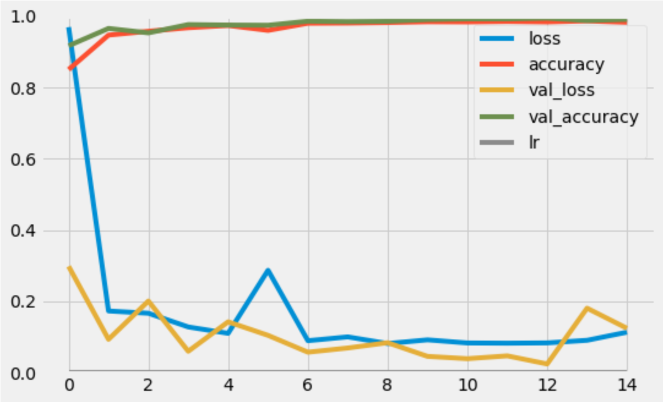
```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_6 (InputLayer)        [(None, 128, 128, 3)]     0

 sequential_2 (Sequential)   (None, 128, 128, 3)       0

 resnet50 (Functional)       (None, None, None, 2048)  23587712

 batch_normalization_2 (Batc  (None, 4, 4, 2048)       8192
 hNormalization)

 flatten_2 (Flatten)         (None, 32768)             0

 dense_4 (Dense)             (None, 1024)              33555456

 dropout_2 (Dropout)         (None, 1024)              0

 dense_5 (Dense)             (None, 12)                12300

=================================================================
Total params: 57,163,660
Trainable params: 34,627,596
Non-trainable params: 22,536,064
_________________________________________________________________
```

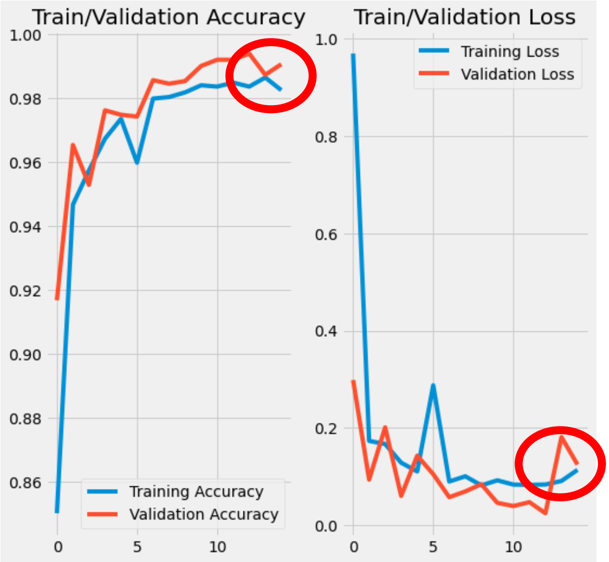
Fine-tuning involves a trade-off between the desire to update as many layers as possible to improve pure performance on the target task and the risk of overfitting if too many layers are updated.

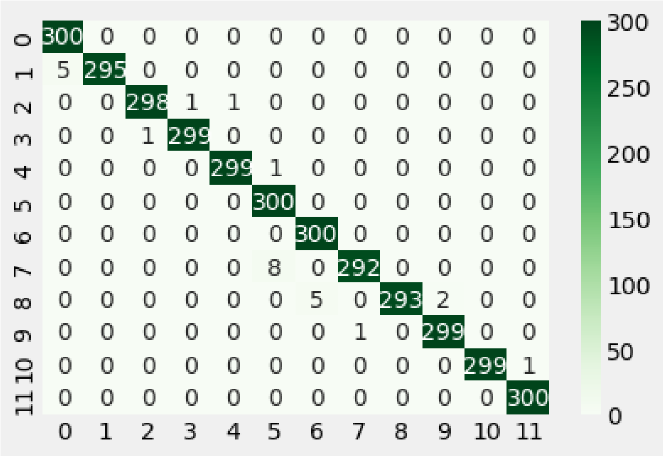
After monitoring some experiments, for this specific project task it was found that the maximum layers to unfroze were 6 like said before, since after model start to overfit like it's possible to see from the plots below.

Here the number of epochs were reduced to 5.

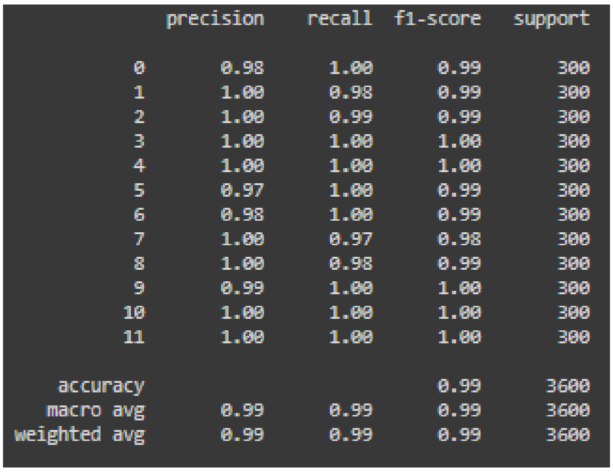
- **Adam**







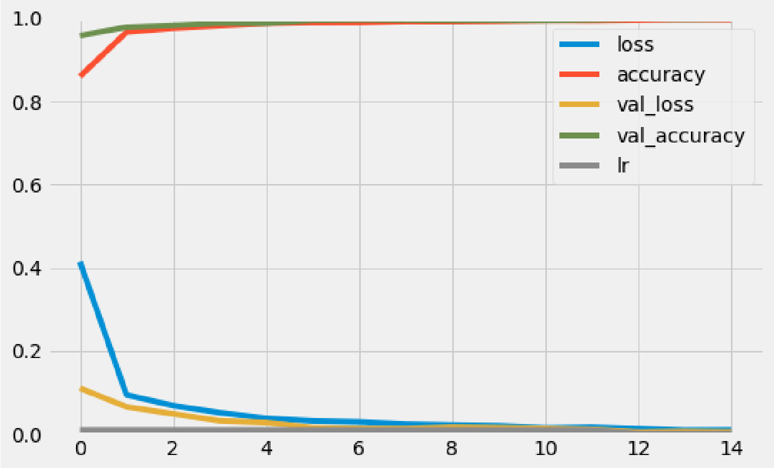
	0	1	2	3	4	5	6	7	8	9	10	11
0	300	0	0	0	0	0	0	0	0	0	0	0
1	5	295	0	0	0	0	0	0	0	0	0	0
2	0	0	298	1	1	0	0	0	0	0	0	0
3	0	0	1	299	0	0	0	0	0	0	0	0
4	0	0	0	0	299	1	0	0	0	0	0	0
5	0	0	0	0	0	300	0	0	0	0	0	0
6	0	0	0	0	0	0	300	0	0	0	0	0
7	0	0	0	0	0	8	0	292	0	0	0	0
8	0	0	0	0	0	0	5	0	293	2	0	0
9	0	0	0	0	0	0	0	1	0	299	0	0
10	0	0	0	0	0	0	0	0	0	0	299	1
11	0	0	0	0	0	0	0	0	0	0	0	300

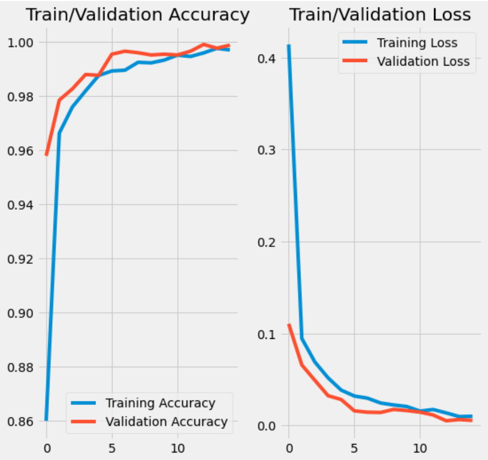


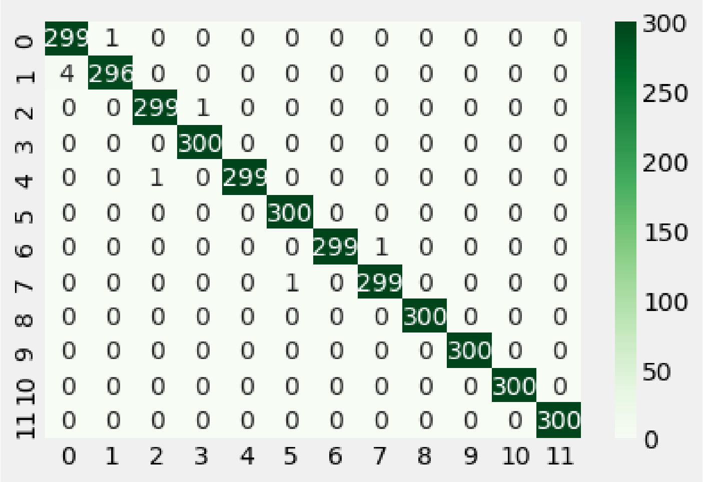
```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       300
           1       1.00      0.98      0.99       300
           2       1.00      0.99      0.99       300
           3       1.00      1.00      1.00       300
           4       1.00      1.00      1.00       300
           5       0.97      1.00      0.99       300
           6       0.98      1.00      0.99       300
           7       1.00      0.97      0.98       300
           8       1.00      0.98      0.99       300
           9       0.99      1.00      1.00       300
          10       1.00      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           0.99      3600
   macro avg       0.99      0.99      0.99      3600
weighted avg       0.99      0.99      0.99      3600
```

By observing plots there are some light overfitting signs. The gap between curves after an instant in which collides that means that the model is in an ideal situation, start to become larger and this is one of the drawbacks of unfroze some layers of a pretrained network. When a model is in overfitting one of the best cures is to reduce model complexity. However, the model performed with a **99** of accuracy and a loss of **0.08** on test set.

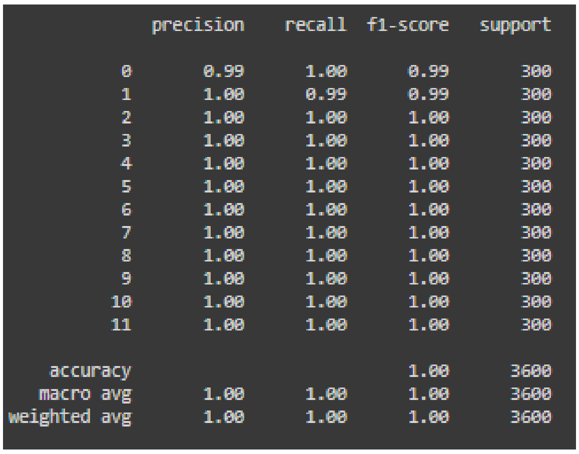
- **SGD**







	0	1	2	3	4	5	6	7	8	9	10	11
0	299	1	0	0	0	0	0	0	0	0	0	0
1	4	296	0	0	0	0	0	0	0	0	0	0
2	0	0	299	1	0	0	0	0	0	0	0	0
3	0	0	0	300	0	0	0	0	0	0	0	0
4	0	0	1	0	299	0	0	0	0	0	0	0
5	0	0	0	0	0	300	0	0	0	0	0	0
6	0	0	0	0	0	0	299	1	0	0	0	0
7	0	0	0	0	0	1	0	299	0	0	0	0
8	0	0	0	0	0	0	0	0	300	0	0	0
9	0	0	0	0	0	0	0	0	0	300	0	0
10	0	0	0	0	0	0	0	0	0	0	300	0
11	0	0	0	0	0	0	0	0	0	0	0	300



```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       300
           1       1.00      0.99      0.99       300
           2       1.00      1.00      1.00       300
           3       1.00      1.00      1.00       300
           4       1.00      1.00      1.00       300
           5       1.00      1.00      1.00       300
           6       1.00      1.00      1.00       300
           7       1.00      1.00      1.00       300
           8       1.00      1.00      1.00       300
           9       1.00      1.00      1.00       300
          10       1.00      1.00      1.00       300
          11       1.00      1.00      1.00       300

    accuracy                           1.00      3600
   macro avg       1.00      1.00      1.00      3600
weighted avg       1.00      1.00      1.00      3600
```

SGD seems to behave better than ADAM optimizer since it doesn't overfit, and the training phase is more stable than the ADAM one that means that model is more robust and ready to face unseen data. Accuracy score on test set is **99.7%** with **0.06** of loss so it's slightly better than ADAM.

10. FINAL CONSIDERATIONS

Project goal has been reached out since several model with different technologies have been built up. Moreover, some sophisticated techniques were exploited like using directly Tensors instead of array to make computation lighter since RAM memory in Collab is not so high.

Low RAM usage allowed to execute models in row without some memory crashes allowing faster developing time. That's an important aspect when this technology will be applied on dataset 5 or 6 times bigger w.r.t. that used in this project.

The question is **which model works better** in this scenario? Here an overview:

- The **first CNN** from scratch even if is not deeper as the second has more parameters and observing the results, it shows a convergence at the end of the training phase but a lot of instability maybe due to number of parameters. It seems to oscillate around minimum at the beginning but in the end, it is stable towards the final convergence. Its training time are 1s/step. It does errors in classification but without making confusion between classes that are instantly shows by the confusion matrix.
- The **second CNN** from scratch it's the deepest of the two and it's reasonably slow, it needs 2s/step. This CNN seems to be better than the first one maybe due to the right depth of the network itself since it made less misclassifications.
- The **third CNN** is a Hyper Tuned version of the second CNN. Hyper tuning with Random Search found a pool of "optimal" parameters, even if it performed slightly worst then the non-hyper tuned parameters. The research here was done to figure out if the model with some different parameters it would get better, but it wasn't the case.
- The **fourth** (built with Adam) and the **fifth** (built with SGD) **models** leverage on ResNet50 using it as feature extractor and without data augmentation. From a training phase point of view the model is built linearly but with slow training times due to ResNet depth. The results of both models are better than all those obtained from the previous architecture because they made less than 10 classification error with an accuracy close to 100%. Here the SGD is the best.
- The **sixth** (built with Adam) and the **seventh** (built with SGD) **models** leverage on a completely frozen ResNet50 architecture with data augmentation. The model built with Adam performed with an accuracy of 98% but it did more misclassification errors w.r.t to its non-augmented version. SGD

model performed better than the Adam with an accuracy of 99% and less errors. Here SGD is the best.

- Last two networks are **fine tuned versions** built with Adam and SGD. ResNet here is used with 6 unfrozen layers. Both models have an accuracy of 99% but that built with Adam during the training phase is more unstable w.r.t that built with SGD.

After the overview it's not easy to choose the best. Each one has its peculiarities but from the pretrained network pool, those with SGD are the best while from those built from scratch the second network works better with this kind of problem. The Hyper tuned version could be discarded.

SGD worked better for its native structure because it's less sensitive to local minima w.r.t Adam and it could be observable from the plot especially from the first CNN built from scratch.

Adam takes information by mean and gradient variance to adapt dynamically the learning rate, but this kind of structure can make Adam more sensitive to data noise.

In the end if network params are high (more then 5 million) it's better to pickup SGD models while if params are less than 5 millions Adam could be picked up.