University of Rome "La Sapienza"
Department of Ingegneria Informatica, Automatica e Gestionale

Machine Learning
Homework 2

Image Classification

SAPIENZA
UNIVERSITÀ DI ROMA

Gianmarco Scarano

Matricola:
2047315

January 5, 2023

Contents

Chapter 1

Introduction

1.1 Image Classification

The main topic about this second homework is related to Image Classification.
We must choose a dataset of our choice and solve an Image Classification problem.

1.2 Requirements

As for the requirements, we can can choose any dataset or subset of a data set with the following features:

- At least 10 classes
- At least 150 images per class
- Educational datasets are not allowed (CIFAR, MNIST, etc.)

1.3 Goal

Our goal for this homework is to show and demonstrate the differences between the results given by different algorithms/approaches.

1.4 Code writing

The Jupyter Notebook related to this report has been written through Visual Studio Code and tested afterwards multiple times also on Google Colaboratory where cells did run perfectly fine on both machines. The framework used is `Tensorflow/Keras`.

Chapter 2

Dataset

2.1 LEGO Bricks

Various datasets have been explored, but in the end we decided to setup our problem on a simple and funny dataset downloaded from Kaggle: LEGO Bricks [1].

This dataset is composed by 40000 images of 50 different LEGO bricks (50 classes), rendered on a PC by 800 different angles.

2.2 Train & Validation

32000 images (*80%*) have been selected for the training phase, while 8000 (*20%*) have been selected for the validation phase. The validation images were already listed inside a file called `validation.txt`, created by the dataset developer himself.

The dataset can be easily downloaded via a simple variable called `downloadDataset`, which if set to *True*, uses the `gdown` library in order to download the LEGO Bricks dataset from Google Drive and places it automatically in the correct folder.

The `pandas` library has been used to generate two different DataFrames, one for training and one for validation.

Very simply, we did store the following variables for both DataFrames:

Image	**Path**	**ID**	**Name**	**Class**
⟨PIL Image ⟩	14719 flat tile corner 2x2 000L.png	000L	flat tile corner 2x2	14719
⟨PIL Image ⟩	15672 roof tile 1x2 000L.png	000L	roof tile 1x2	15672
⟨PIL Image ⟩	2420 plate corner 2x2 000L.png	000L	plate corner 2x2	2420
...	...	...	...	...
⟨PIL Image ⟩	99301 roof tile inside 3x3 079R.png	079R	roof tile inside 3x3	99301

2.3 Data augmentation

As for the data augmentation, 3 types of augmentation have been implemented, that goes as follows:

- 0 (No image pre-processing)
- 1 (Mid image pre-processing)
- 2 (High image pre-processing)

Please note that each and every one of these, implement the rescaling feature of pixels between 0 and 1 (meaning that the *0* augmentation implementation just implements the Rescale feature).

Plus, the images (which original size was 400 x 400) have been scaled down to a resolution of 200 x 200 for computational purposes.

	0 - No pre-processing	**1 - Mid pre-processing**	**2 - High pre-processing**
Rescale	✓	✓	✓
Zoom Range	X	0.1	0.1
Rotation Range	X	5	40
Horizontal Flip	X	X	0.3
Vertical Flip	X	X	0.2
Width Shift Range	X	X	0.2
Height Shift Range	X	X	0.1

In the next section, we'll see how `Weight & Biases` tools can help us in order to show these augmented images, along with metrics, etc.

2.3.1 Simple Example

Here comes a simple example of an image inside our LEGO Bricks dataset:

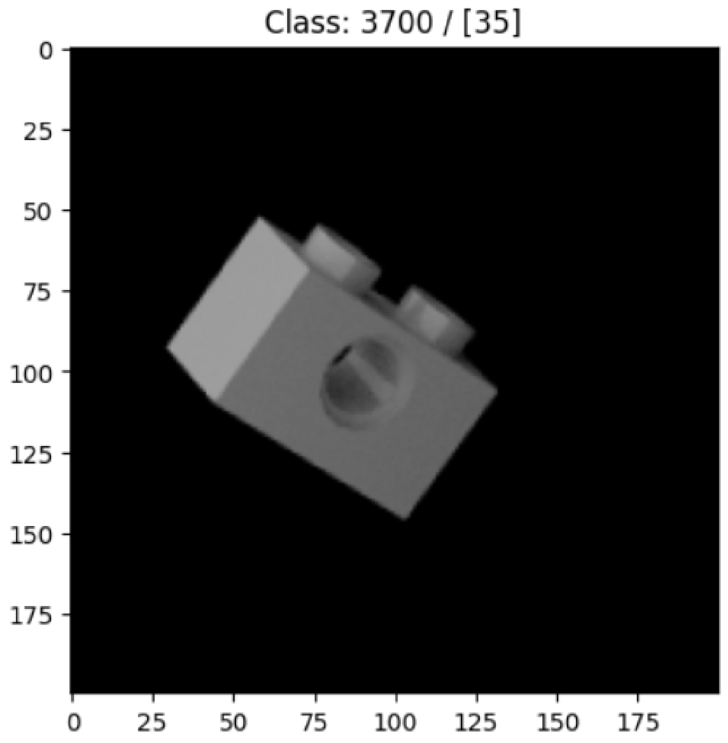


Figure 1: Simple example image taken from the LEGO Bricks dataset, along with its class and the position in the one-hot encoding vector.

Chapter 3

Weight & Biases

`Weight & Biases` [2] is a central dashboard to keep track of your hyperparameters, system metrics, and predictions so you can compare models live, and share your findings.

3.1 Helper functions

Various extra implementations have been manually coded and added aside from the standard WANDB tools.

3.1.1 Delete Artifacts

In WANDB, when saving all epochs models and the best model as well, those get saved in the WANDB servers, allowing the quota (*100GB*) to grow real quick. At the end of every training, we can manually delete the models we don't need one by one in order to save some space, but with this implementation, we delete useless models from the WANDB servers automatically, just by passing the `run_id` parameter. It skips the last epoch model (useful if we want to continue the training process later) and the latest best model saved.

3.1.2 Config changer in real-time

Sometimes, we might need to change a config value in real-time while the run is still running. This method edits a configuration parameter given a `run_id`. The importance of this will be explained later during the training phase section.

3.1.3 Resuming a run and a checkpoint for continuing the training phase

When we stop a training, we want to resume the stopped run and load the checkpoint of the latest epoch. This method does exactly this, finding automatically the latest epoch model and loading in the `model` variable, in order to continuing the training phase.

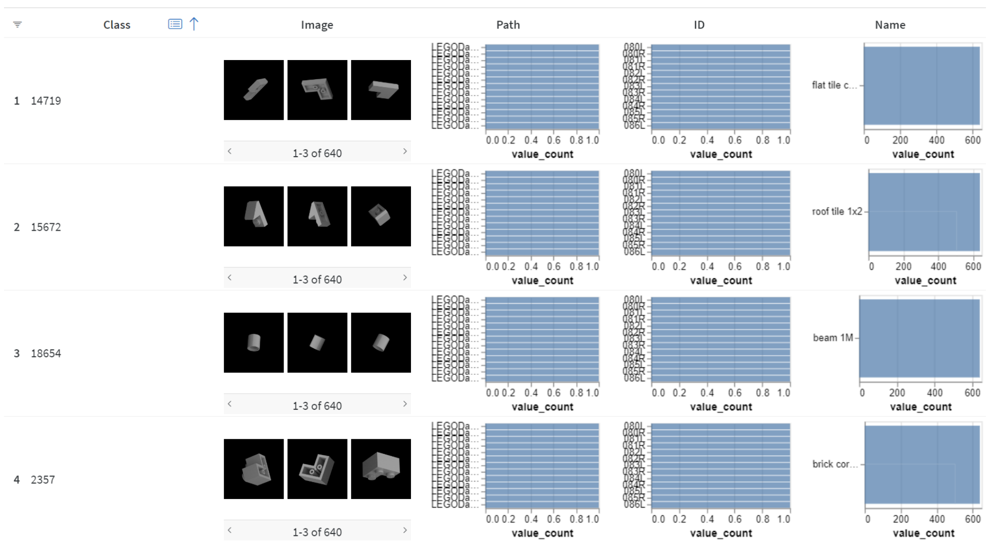
3.1.4 Resuming a run and retrieve the best model

This simple method, instead, retrieves the best model for a specific run passed by the `run_id` parameter. This is useful when we want to test our best configuration real quick.

3.2 Tables

We deployed two tables in our project in WANDB, which are the original images of the dataset and the augmented images through pre-processing techniques.

3.2.1 Original images

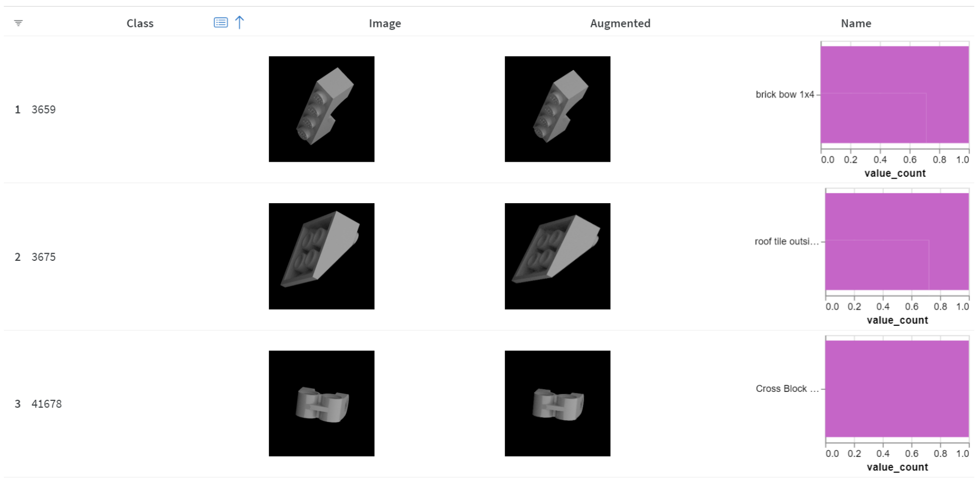


	Class	Image	Path	ID	Name
1	14719	1-3 of 640			flat tile c...
2	15672	1-3 of 640			roof tile 1x2
3	18654	1-3 of 640			beam 1M
4	2357	1-3 of 640			brick cor...

Figure 1: Table for original images in the dataset. The table is grouped by Class.

3.2.2 Augmented images

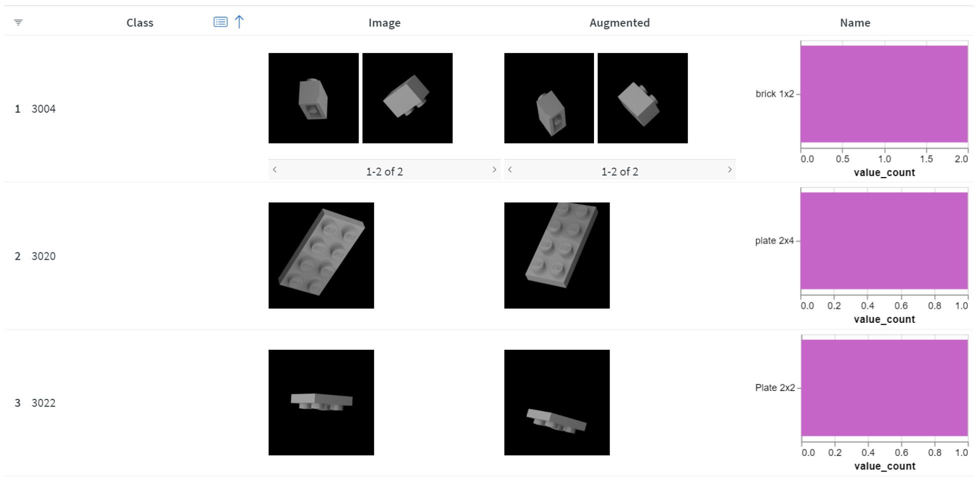
Mid pre-processing



	Class	Image	Augmented	Name
1	3659			brick bow 1x4
2	3675			roof tile outsi...
3	41678			Cross Block ...

Figure 2: Table for augmented images in the dataset through mid-preprocessing. We can see very few edits, due to how the mid-preprocessing technique was built. The table is grouped by Class.

High pre-processing



	Class	Image	Augmented	Name
1	3004	1-2 of 2	1-2 of 2	brick 1x2
2	3020			plate 2x4
3	3022			Plate 2x2

Figure 3: Table for augmented images in the dataset through high-preprocessing. We can see how the stronger pre-processing technique is working well, applying random flips and other enhancements. The table is grouped by Class.

Chapter 4

Approaches & Results

4.1 Training phase

As for the training phase, the parameters of each run are stored inside a dictionary, which contains:

- **Batch size:** 32
- **Pre-train weights:** Will be populated accordingly after choosing the model (Here we specify if our model should be pre-trained or not)
- **Epochs:** 50
- **Learning rate:** 0.001
- **LR Decay rate:** 0.1
- **Optimizer:** Adam
- **Model name:** Will be populated accordingly after choosing the model
- **Loss function:** Categorical Cross-Entropy
- **Metrics:** Accuracy
- **Patience:** 5 (5 epochs of patience in order to trigger the EarlyStopping method)
- **Data augmentation:** 1 (Here we specify which pre-processing technique we would like to apply)

Later on, we build our `DataLoaders` specifying that the train loader must be shuffled, while the validation loader mustn't be. Other than that, we specify that the class mode should be `categorical`.

We also define a Learning Rate Scheduler callback, which lowers the LR after the 8th epoch by this constant:

$$LR = LR \times e^{-0.1}$$

Along with this scheduler, we define other callbacks, which are:

- **Delete epochs:** (previously commented in 3.1.1)
- **Early Stopping:** Stops training after 5 epochs without any better result.
- **Model checkpoint:** Saves the model for each epoch and saves the best model as well.

Finally, we start our training through a WANDB run, which will get saved on our WANDB dashboard.

4.2 Personal Model (LEGOModel)

We built our personal model (LEGOModel) starting from scratch. Several attempts have been made, changing and editing small parts of its architecture. In the end, the best configuration is the one consisting in 3 convolutional blocks, configured as follows:

- `Conv2D` layer with 16, 32, 64 kernels respectively of size (4, 4) and stride (1, 1).

  We use `ReLU` as the activation function, along with padding 'same', meaning that the output size will be the same as the input one.
- `Batch Normalization` on the output of the previous `Conv2D` layer.
- `AVG Pooling` with a Pool Size of (2, 2).

  We use the padding 'valid', meaning that our output shape will be of: (100, 100, 16)

In this way, we end up with a (25, 25, 64) tensor.

After this, we `Flatten` our previous tensor and instantiate a `Fully Connected Layer` (Dense) with 2048 units and `ReLU` as activation function on top.

Later, we simply apply a `BatchNormalization()` and a `Dropout Layer` with value 0.2.

`Dropout` rate is very low, otherwise the validation accuracy would be higher than training (Due to the fact that while in evaluate mode, the `Dropout Layer` gets deactivated).

We then apply another `FC Layer` with 1024 units and lastly, we build the last `FC Layer` with 50 units (classes) called Prediction_Layer with a simple `SoftMax` activation function on top.

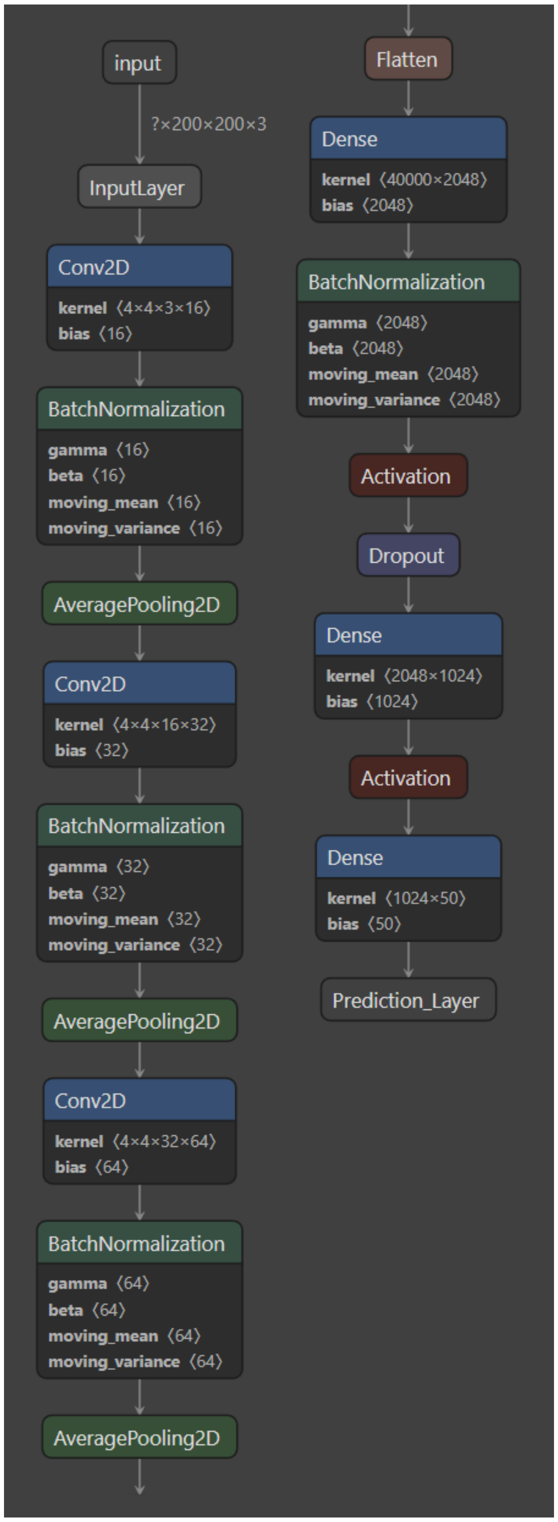


Figure 1: A simple visualization of the LEGOModel (personal model) architecture.

4.2.1 Results

Our model performed discreetly well on our Image Classification problem, returning an accuracy of **88.17%** and a total loss of **0.50** on the *validation* set. As stated before, various attempts have been made for this model by changing and editing small parts of its architecture. All tests can be reviewed through this simple legend generated by the WANDB tools.

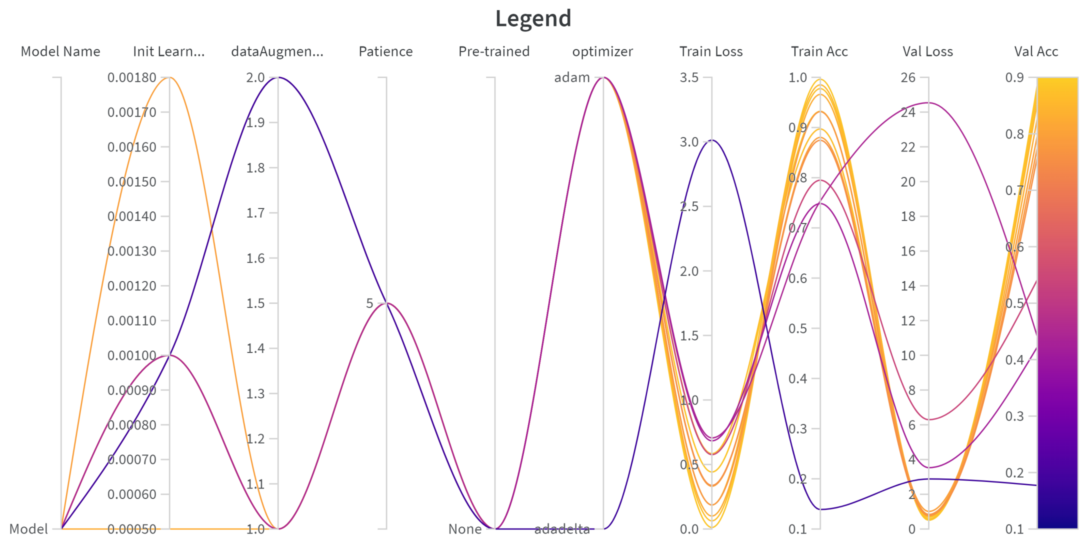


Figure 2: A simple and efficient legend for all the tests carried out with the LEGO-Model.

From this legend, we can see that few configurations did have some worse results, while we can observe that the yellow lines gradually brought us to an optimal solution, since optimization tests have been carried out. Let's remember that the weights of each configuration start from scratch, so there is no pre-training on any dataset (such as ImageNet). The reason behind the worse results lies in the model architecture, since in most of them `LeakyReLU` have been used, along with a `Dropout Layer` after each `Conv2D` layer.

In the other worse results w.r.t the best LEGOModel architecture, two `Conv2D` layers have been stacked together in the 3 blocks, as well as using `MaxPooling` instead of `AVGPooling`.

The data augmentation of level 2 (*high*) also could be the culprit of such low accuracy, because most probably the model confused samples of various classes with samples coming from other classes due to the high pre-processing techniques, where a class could be mistaken with another very easily, since the overall background of the images is almost the same for all the 40000 samples.

Tests have been carried out also with different optimizers such as `Adadelta` with data augmentation of level 2, but we can see that the accuracy is very poor (**17.71%**).

4.2.2 Classification Report

As we said, we reached an accuracy of **88.17%** with our LEGOModel, but we have to make sure that this accuracy is reflected in all the classes and not just in one class (synonym of overfitting issues). That's why we need two methods called Classification Report and Confusion Matrix, which will help us visualizing the *Precision*, *Recall* and *F1-Score* of each class.

Table 4.1: Classification Report for the LEGOModel model

Class	Precision	Recall	F1-Score	Support
14719	0.8701	0.8375	0.8535	160
15672	0.9568	0.9688	0.9627	160
18654	1.0000	1.0000	1.0000	160
2357	0.7586	0.8250	0.7904	160
2420	0.8774	0.8500	0.8635	160
2780	0.9521	0.9938	0.9725	160
27925	0.9241	0.9125	0.9182	160
3001	0.8256	0.8875	0.8554	160
3002	0.7206	0.9187	0.8077	160
3003	0.6667	0.8625	0.7520	160
3004	0.8882	0.8438	0.8654	160
3005	0.9390	0.9625	0.9506	160
3010	0.8311	0.7688	0.7987	160
3020	0.9494	0.9375	0.9434	160
3021	0.9068	0.9125	0.9097	160
3022	0.7399	0.8000	0.7688	160
3023	0.8537	0.8750	0.8642	160
3024	0.8977	0.9875	0.9405	160

Continued on next page

Table 4.1: Classification Report for the LEGOModel model (Continued)

Class	Precision	Recall	F1-Score	Support
3037	0.8571	0.7500	0.8000	160
3038	0.6914	0.7562	0.7224	160
3039	0.7451	0.7125	0.7284	160
3040	0.8466	0.8625	0.8545	160
3045	0.8696	0.7500	0.8054	160
3046	0.6643	0.5813	0.6200	160
3062	0.9938	1.0000	0.9969	160
3063	0.9038	0.8812	0.8924	160
3068	0.9423	0.9187	0.9304	160
3069	0.8383	0.8750	0.8563	160
3070	0.9341	0.9750	0.9541	160
3298	0.8657	0.7250	0.7891	160
33909	0.7950	0.8000	0.7975	160
3622	0.8289	0.7875	0.8077	160
3623	0.8727	0.9000	0.8862	160
3659	0.9252	0.8500	0.8860	160
3675	0.9608	0.9187	0.9393	160
3700	0.8788	0.9062	0.8923	160
3794	0.9615	0.9375	0.9494	160
4150	0.9937	0.9875	0.9906	160
41677	0.9576	0.9875	0.9723	160
41678	1.0000	0.9875	0.9937	160
4274	0.9938	1.0000	0.9969	160
4286	0.8981	0.8812	0.8896	160
43093	1.0000	0.9500	0.9744	160
43857	1.0000	0.9938	0.9969	160

Continued on next page

Table 4.1: Classification Report for the LEGOModel model (Continued)

Class	Precision	Recall	F1-Score	Support
4490	0.8704	0.8812	0.8758	160
54200	0.9325	0.9500	0.9412	160
6143	1.0000	1.0000	1.0000	160
6632	0.9744	0.9500	0.9620	160
85984	0.9714	0.8500	0.9067	160
99301	0.9067	0.8500	0.8774	160
Accuracy			0.8860	8000
Macro avg	0.8886	0.8860	0.8861	8000
Weighted avg	0.8886	0.8860	0.8861	8000

4.2.3 Confusion Matrix

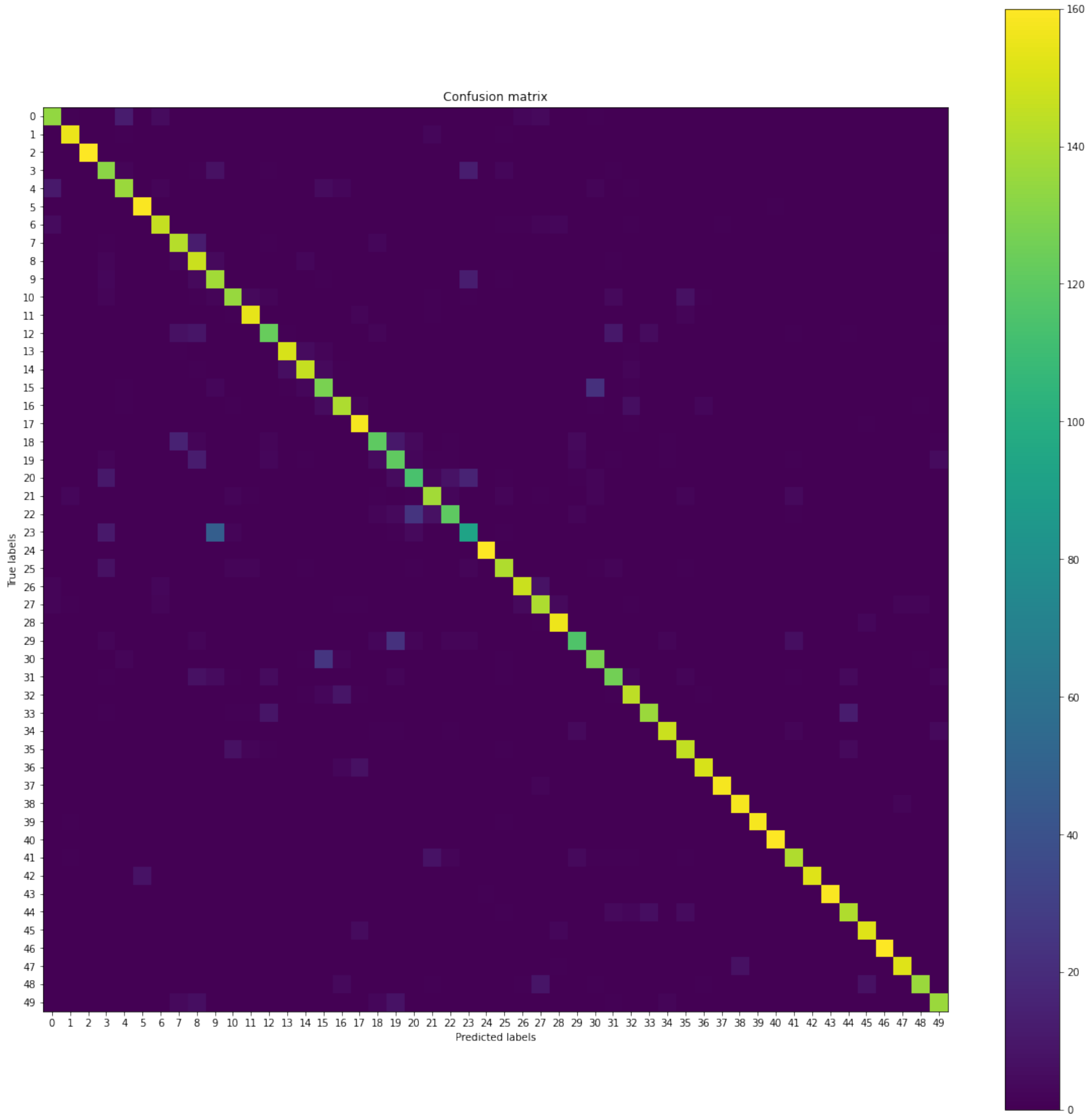


Figure 3: Confusion Matrix for the LEGOModel.

From this simple Confusion Matrix, along with the Classification Report, we can see the 50 classes and how well we can distinguish the oblique line, meaning that the model almost correctly identified all classes. There is one class (Class 23), though, which has not been correctly identified. This thesis is supported by the fact that if we check the above Classification Report, we can notice that the class 3046 (which in the one-hot encoding vector has position 23), has an F1-Score of `0.6200` versus all other classes which have an average of `0.9579` of F1-Score.

4.2.4 Confusion Matrix with error percentage

We can now proceed to analyze a table of error percentages, starting from the Confusion Matrix method. With this new table, we are able to check exactly which classes our model didn't properly predict, so that we can do further analysis on the faulty ones.

N.B. We will display only errors strictly greater than 5 and with a percentage strictly greater than 0.06% otherwise the list would be very very long.

Table 4.2: Error percentages for the LEGOModel model

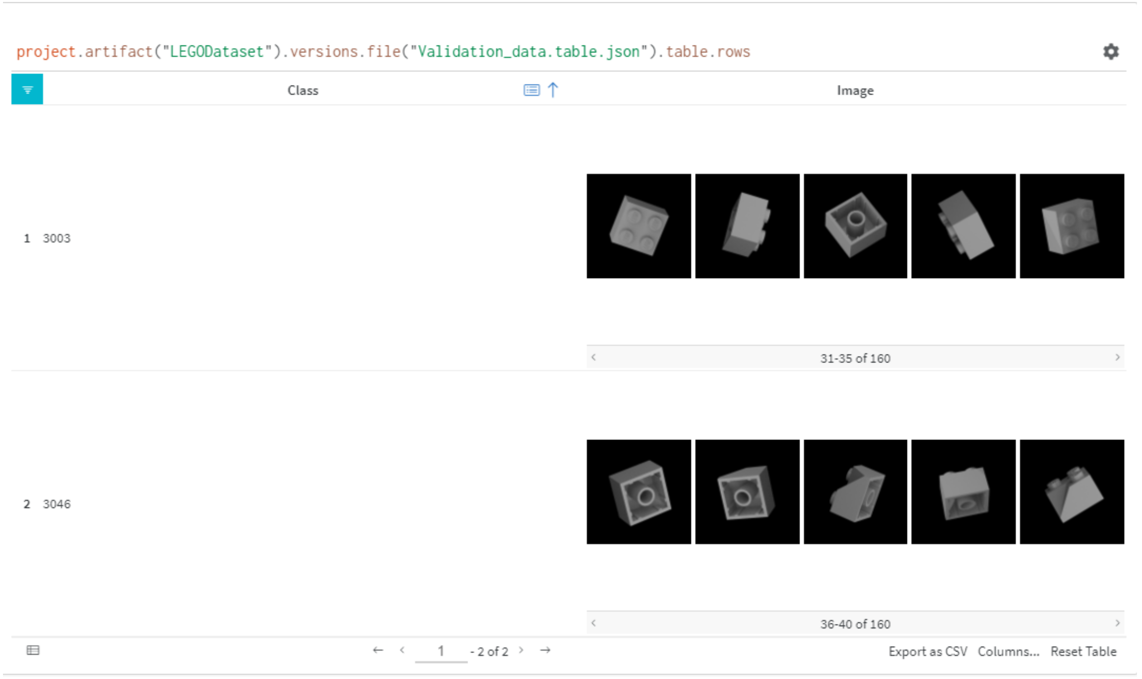
True	Predicted	Errors	Percentage error
3046	3003	48	0.60%
33909	3022	25	0.31%
3045	3039	24	0.30%
3298	3038	23	0.29%
3022	33909	22	0.28%
3039	3046	16	0.20%
3037	3001	15	0.19%
3003	3046	13	0.16%
2357	3046	13	0.16%
14719	2420	12	0.15%
3001	3002	12	0.15%
3659	4490	12	0.15%
3038	3002	12	0.15%
2420	14719	11	0.14%
3046	2357	11	0.14%
3010	3622	10	0.12%
3037	3038	10	0.12%
3039	2357	10	0.12%

Continued on next page

Table 4.2: Error percentages for the LEGOModel model (Continued)

True	Predicted	Errors	Percentage error
3659	3010	9	0.11%
3010	3002	9	0.11%
85984	3069	9	0.11%
3623	3023	9	0.11%
3039	3045	8	0.10%
99301	3038	8	0.10%
43093	2780	8	0.10%
3068	3069	8	0.10%
4286	3040	8	0.10%
3700	3004	7	0.09%
3004	3700	7	0.09%
3010	3001	7	0.09%
3794	3024	7	0.09%
3063	2357	7	0.09%
6632	41677	7	0.09%
85984	54200	7	0.09%
2357	3003	7	0.09%
3045	3040	7	0.09%
3622	3002	7	0.09%
3023	3623	6	0.07%
3298	4286	6	0.07%
99301	3002	6	0.07%
4490	3659	6	0.07%
3021	3020	6	0.07%

As expected, class 3046 has been misclassified 48 times as class 3003. Most probably this means that the two classes are very similar or have very similar features. For this purpose, we use the WANDB Table which help us identify the two classes through a simple query followed by a filter for showing only the two classes:



```
project.artifact("LEGODataset").versions.file("Validation_data.table.json").table.rows
```

	Class	Image
1	3003	31-35 of 160
2	3046	36-40 of 160

Figure 4: Table which shows the two classes which the model easily confused.

We can *clearly* see why the model is confusing the two classes. The bottom part of the LEGO brick is pretty much the same, while the top is slightly different (in 3003 there is a flat top, while in 3046 there is a rounded corner). There is a very high chance that the model correctly identifies class 3046 instead of 3003 when it looks at the top of the brick, while there is high uncertainty when it anaylizes the bottom part of the brick. This approach works for all the classes who have been misclassified, but the differences will get smaller and smaller as the percentage error decreases.

(For example, the same thing happens between class 3022 and 33909, where the model confused 25 samples when analizying the bottom part of the brick, which is identical on both classes).

4.3 Transfer Learning

Another test, called Transfer Learning, has been carried out along with our personal model. Basically, it is a Machine Learning method where a model developed for a task is reused as the starting point for a model on a second task. With this being said, the models reused as starting point are listed here below:

- EfficientNet-L2 [3]
- MobileNetV2 [4]
- MobileNetV3Small [5]
- ResNet50 [6]
- VGG16 [7]
- VGG19 [7]
- Xception [8]

4.3.1 Results

Starting off with a simple legend, we show the overall performances of each of these models used as starting point for our Image Classification task.

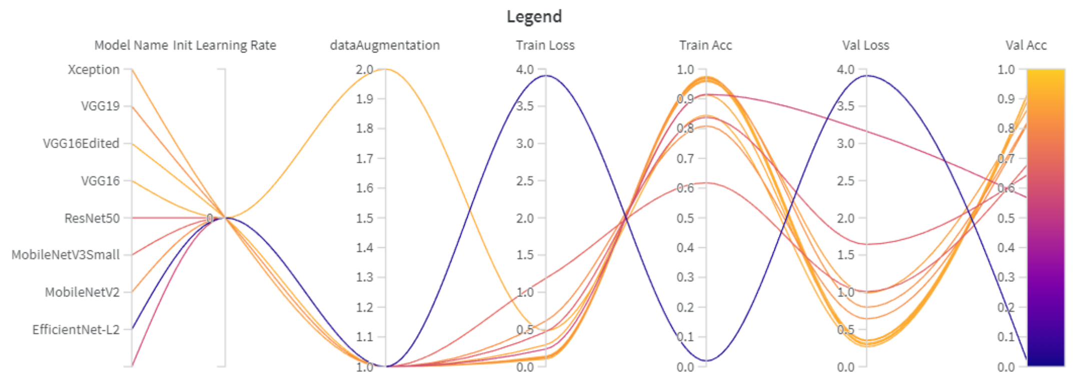


Figure 5: A simple and efficient legend for all the tests carried out with the Transfer Learning method.

All of these models weights are pre-trained on ImageNet and have been trained on the `Adam` optimizer. There are some remarks about few models though, like `Xception`, a 2017 model developed by Google, which uses the `SGD` optimizer as per its paper, or `EfficientNet-L2`, which has a total of 502,869,746 parameters. In fact, for this last model, its initialization has been stopped at the 3rd Convolutional block, since it will take around 30 minutes for a single epoch if we consider the whole EfficientNet-L2 architecture:

Epoch	Loss	Accuracy	Time per epoch
1	3.9122	0.0200	12 min
2	3.9124	0.0201	12 min

This is not good for our testing, but this was done to demonstrate that such big models can be loaded and trained without problems with our LEGOBricks dataset. It is not computationally great, though.

As per the other models, we can state that `VGG16` has been the model with the best accuracy and best loss compared to all other models, with a total of **91.35%** for accuracy and **0.29** for loss on the validation set.

Below we can find a chart of the various accuracies of the various runs (related to all the models). We can clearly see how the light-blue one (`VGG16`) finishes its 50 epochs with the highest accuracy, while all other ones have been stopped by the Early Stopping callback.

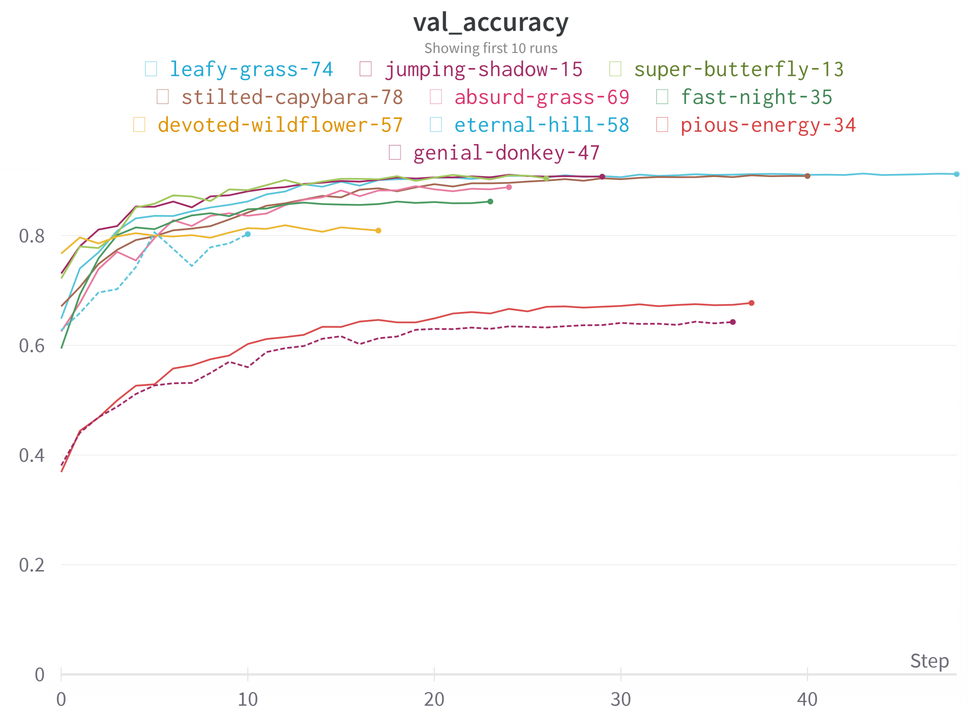


Figure 6: A plot of all the various accuracies of all the various runs in WANDB.

One important thing to mention is that all models have been downloaded and loaded without the top final layers through the variable `include_top=False`, as we would like to place our final layers instead, which are:

- `AveragePooling2D` layer with pool size (4, 4)
- `Flatten Layer` followed by a `Batch Normalization Layer`
- `Dropout Layer` with `p = 0.2`
- 2 `Dense Layers` (of 100 units and 50 units respectively) with `Batch Normalization` and `ReLU` on top.
- Final `SoftMax` activation function.

Here follows a visualization of its full architecture:

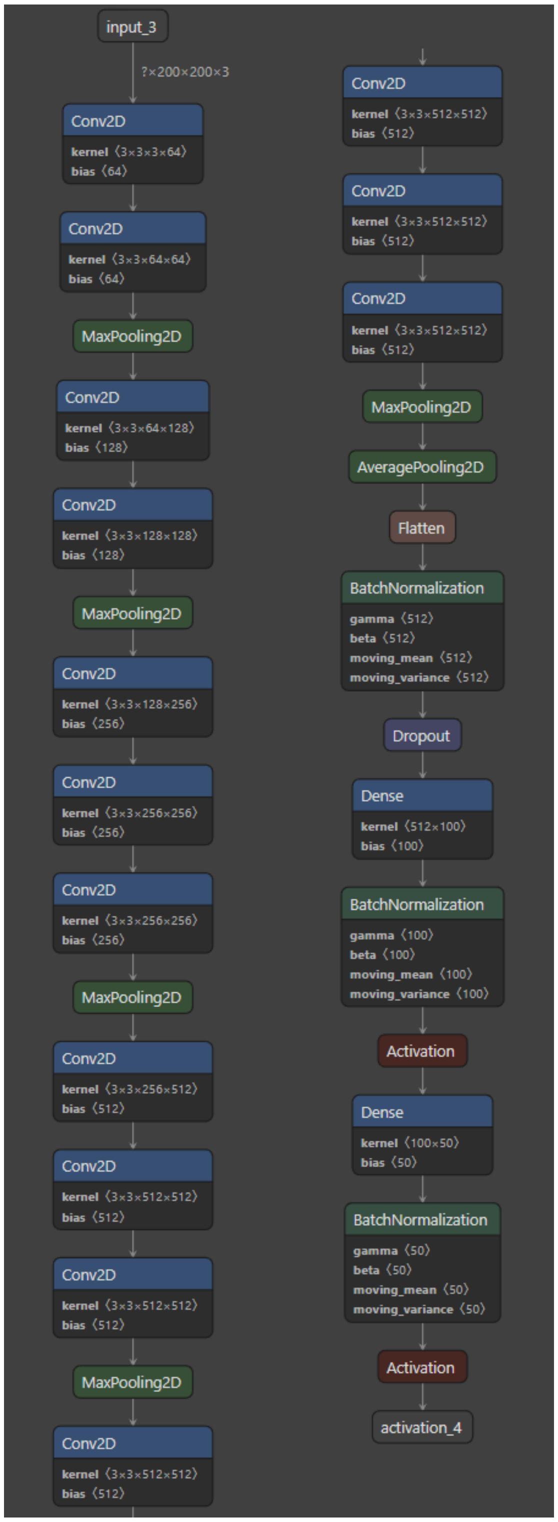


Figure 7: A simple visualization of the VGG16 (edited) architecture.

4.3.2 Classification Report

We can now concentrate on exploring the results, metrics and predictions of the VGG16 models. As we did with our LEGOModel, we are able to show a classification report for the VGG16 model, allowing us to check whether the model correctly classified all classes.

Table 4.3: Classification Report for the VGG16 model

Class	Precision	Recall	F1-Score	Support
14719	0.9146	0.9375	0.9259	160
15672	0.9816	1.0000	0.9907	160
18654	0.9938	1.0000	0.9969	160
2357	0.8555	0.9250	0.8889	160
2420	0.9313	0.9313	0.9313	160
2780	0.9938	1.0000	0.9969	160
27925	0.9930	0.8875	0.9373	160
3001	0.8772	0.9375	0.9063	160
3002	0.8712	0.8875	0.8793	160
3003	0.7943	0.8688	0.8299	160
3004	0.8526	0.8313	0.8418	160
3005	0.9176	0.9750	0.9455	160
3010	0.9189	0.8500	0.8831	160
3020	0.9383	0.9500	0.9441	160
3021	0.9245	0.9187	0.9216	160
3022	0.8442	0.8125	0.8280	160
3023	0.9281	0.8875	0.9073	160
3024	0.8579	0.9812	0.9155	160
3037	0.9078	0.8000	0.8505	160
3038	0.8155	0.8562	0.8354	160
3039	0.7588	0.8063	0.7818	160

Continued on next page

Table 4.3: Classification Report for the VGG16 model (Continued)

Class	Precision	Recall	F1-Score	Support
3040	0.8800	0.8250	0.8516	160
3045	0.9058	0.7812	0.8389	160
3046	0.7605	0.7937	0.7768	160
3062	1.0000	1.0000	1.0000	160
3063	0.9869	0.9437	0.9649	160
3068	0.9366	0.8313	0.8808	160
3069	0.8140	0.8750	0.8434	160
3070	0.8857	0.9688	0.9254	160
3298	0.9110	0.8313	0.8693	160
33909	0.8046	0.8750	0.8383	160
3622	0.8580	0.9062	0.8815	160
3623	0.9313	0.9313	0.9313	160
3659	0.9542	0.9125	0.9329	160
3675	0.9539	0.9062	0.9295	160
3700	0.8848	0.9125	0.8985	160
3794	0.9299	0.9125	0.9211	160
4150	0.9938	0.9938	0.9938	160
41677	0.9873	0.9750	0.9811	160
41678	1.0000	0.9938	0.9969	160
4274	1.0000	1.0000	1.0000	160
4286	0.8650	0.8812	0.8731	160
43093	1.0000	0.9938	0.9969	160
43857	0.9936	0.9750	0.9842	160
4490	0.9024	0.9250	0.9136	160
54200	0.9367	0.9250	0.9308	160
6143	0.9877	1.0000	0.9938	160

Continued on next page

Table 4.3: Classification Report for the VGG16 model (Continued)

Class	Precision	Recall	F1-Score	Support
6632	0.9812	0.9812	0.9812	160
85984	0.9524	0.8750	0.9121	160
99301	0.8951	0.9062	0.9006	160
Accuracy			0.9135	8000
Macro avg	0.9153	0.9135	0.9136	8000
Weighted avg	0.9153	0.9135	0.9136	8000

4.3.3 Confusion Matrix

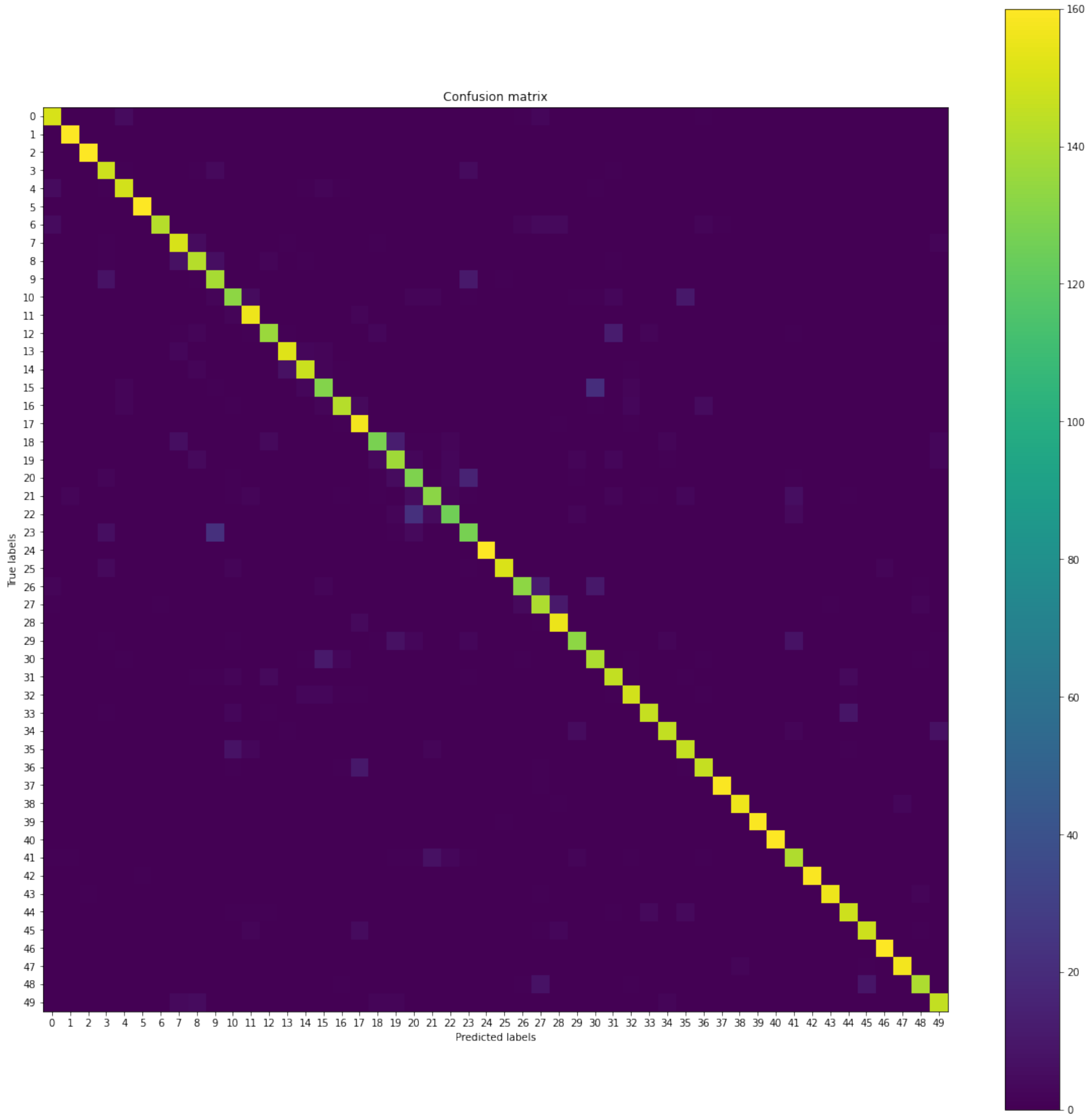


Figure 8: Confusion Matrix for the VGG16 model.

If we take into consideration the LEGOModel Confusion Matrix, we can see that the errors are almost similar, but the frequency of these errors is way less in VGG16. There are few classes who have been correctly classified with an accuracy of 100,00%, but taking into account the Class 3045, which is class 22 in the one-hot encoding vector, we can see that it has been misclassified, returning an F1-Score of `0.8389`. Following the same path we used on the LEGOModel, we can now check the error percentage and its error frequency in the Error Percentage table, starting from this Confusion Matrix.

4.3.4 Confusion Matrix with error percentage

N.B. We will display only errors strictly greater than 5 and with a percentage strictly greater than 0.06% otherwise the list would be very very long.

Table 4.4: Error percentages for the VGG16 model

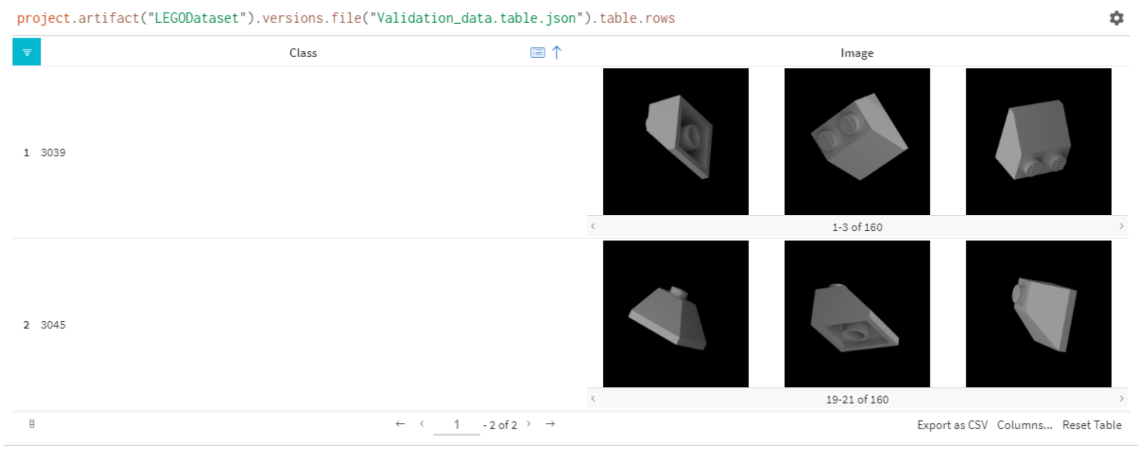
True	Predicted	Errors	Percentage error
3046	3003	22	0.28%
3045	3039	22	0.28%
3022	33909	21	0.26%
3039	3046	16	0.20%
3037	3038	13	0.16%
3010	3622	12	0.15%
3068	3069	12	0.15%
3069	3070	11	0.14%
33909	3022	11	0.14%
3003	3046	11	0.14%
3794	3024	10	0.12%
3004	3700	10	0.12%
3068	33909	10	0.12%
85984	54200	9	0.11%
3659	4490	9	0.11%
85984	3069	8	0.10%
3298	4286	8	0.10%
3700	3004	8	0.10%
3003	2357	8	0.10%
3298	3038	7	0.09%

Continued on next page

Table 4.4: Error percentages for the VGG16 model (Continued)

True	Predicted	Errors	Percentage error
3021	3020	7	0.09%
3002	3001	7	0.09%
4286	3040	7	0.09%
3002	3003	6	0.07%
3040	4286	6	0.07%
3046	2357	6	0.07%
3037	3001	6	0.07%

We already checked before the correlation between Class 3046 and Class 3003, so let's focus our attention to Class 3039 and 3045, always through a WANDB Table which will help us identify the two classes through a simple query followed by a filter for showing only the two classes:



```
project.artifact("LEGODataset").versions.file("Validation_data.table.json").table.rows
```

	Class	Image
1	3039	1-3 of 160
2	3045	19-21 of 160

Figure 9: Table which shows the two classes which the model easily confused.

Again, we can see how the models performs worse when recognizing the bottom part of the bricks. If the bottom part of two bricks are almost identical, then it will be tough for the model to correctly classify which one is the right one, since this happened for 3 classes already. The model has no problem in identifying the correct one when looking at the top part of the brick. If we want to pursue this thesis, a test where we select only the top part of the brick could be deployed, expecting the accuracy to be way higher than the best one we got (**91.35%**).

4.4 Final considerations

The goal of this homework has been reached: We tested two different configurations (in this case we tested multiple architectures but focused our researches on two models) and compared their results through Confusion Matrices, Images, Plots, etc.

Overall, **VGG16** has performed better than our personal model, with a difference of `0.0318` points in accuracy and `0.21` points in loss. This is due to the fact that *VGG16* has a stronger and deeper architecture, running 5 Convolutional Blocks (2 of these with 2 stacked `Conv2D Layers` and the last 3 with 3 stacked `Conv2D Layers`) unlike our personal model which runs 3 simple Convolutional Blocks (where each of them is built with a single `Conv2D Layer`).

All other metrics, such as loss, learning rate, legend, parameter importance etc. can be easily analyzed through the WANDB dashboard of this Homework, which is public.

Bibliography

[1] J. Hazelzet, “Legobricks: 40,000 images of 50 different lego bricks.” `https://www.kaggle.com/datasets/joosthazelzet/lego-brick-images`, 2020.

[2] L. Biewald, “Experiment tracking with weights and biases.” `https://www.wandb.com/site`, 2020. Software available from wandb.com.

[3] M. Tan and Q. V. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks.” `http://arxiv.org/abs/1905.11946`, 2019.

[4] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks.” `http://arxiv.org/abs/1801.04381`, 2018.

[5] A. Howard, M. Sandler, G. Chu, L. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, “Searching for mobilenetv3.” `http://arxiv.org/abs/1905.02244`, 2019.

[6] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition.” `http://arxiv.org/abs/1512.03385`, 2015.

[7] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition.” `https://arxiv.org/abs/1409.1556`, 2014.

[8] F. Chollet, “Xception: Deep learning with depthwise separable convolutions.” `http://arxiv.org/abs/1610.02357`, 2016.