ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS
ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΕΠΙΧΕΙΡΗΣΕΩΝ
SCHOOL OF BUSINESS
ΜΕΤΑΠΤΥΧΙΑΚΟ στην ΕΠΙΧΕΙΡΗΜΑΤΙΚΗ ΑΝΑΛΥΤΙΚΗ
MSc in BUSINESS ANALYTICS

.

Machine Learning and Content Analytics

Waste Management Segregation

Team Members:

Lida Vratsanou – P2822207
Konstantinos Velissaris – P2822204
Katerina Kalamara - P2822211

Professor:

Harris Papageorgiou

Assistant Professor:

George Perakis

Table of Contents

1. Introduction

1.1. Project Description and Problem Statement

In recent years, sustainability has become one of the most rapidly emerging issues. This effect has captured public awareness due to the negative impact of climate change. From undeniable extreme weather events to alarming shortages in natural resources, our society needs now, more than ever, to address this issue effectively.

As populations grow and economies expand, there is a constant pressure that continuously intensifies towards more sustainable and eco-friendly practices that will mitigate the negative consequences that ensue.

In response to this challenge, one solution that has garnered significant attention is Waste Management and Waste Segregation in factories, production sites and other sectors. Presently, waste management, or more accurately the lack of it, has led to serious environmental repercussions, posing a substantial threat to the ongoing climate disruption and climate change crisis. For instance, the accumulation and slow decomposition of non-biodegradable waste in landfills, improper disposal, and lack of effective recycling, have led to severe environmental degradation and contamination of vital natural sources.

More specifically, Waste Management falls short of its potential and the standards it should meet, leaving ample room for improvement. Numerous factories encounter resource constraints, lack of expertise or even challenging waste streams which demand substantial time and effort for proper treatment and segregation.

Upon careful examination of the current situation, there is an urgent need to identify and develop solutions to effectively counter all the above-mentioned threats as swiftly as possible.

1.2. Project Vision and Goals

As a response to the outlined challenges, Waste Management and Waste Segregation emerge as vital pillars in the broader framework of sustainability. Now, more than ever, the demand for a time-efficient and cost-effective solution towards achieving proper waste management and segregation is vital.

To address this need, the integration of systems and algorithms capable of automatically recognizing the composition of items and accurately categorizing them for segregation is vital. Such a solution will allow waste management sites to increase their productivity while reducing their environmental footprint and overall costs.

Through the advancements of machine learning and computer vision, it is now possible to create an algorithm, capable of scanning and analyzing each item as it goes through the production line by using an image recognition software/model. Having fulfilled that, the same system proposes the appropriate waste category for classification (e.g., paper, glass, plastic etc.), completely automating the process of Waste Segregation. In more detail, the goal of this project is to construct and train a deep learning model, able to discern and differentiate various waste images, all achieved through the integration of a camera at the production line. This will prevent the redundant disposal of waste. By implementing this intelligent system, the opportunity to enhance waste treatment is presented and the elevation of recycling rates can be achieved.

Moving forward with this project, our approach will be outlined, describing in detail the methodology that we followed to achieve our objectives and create the aforementioned system. A comprehensive evaluation of the utilized datasets will be carried out, followed by an in-depth description of the predictive models that were developed, along with their final assessments. We will also unveil our selected predictive model and the reasoning behind its selection for our intended purpose.

Ultimately, we will propose innovative approaches on how automatic waste image recognition can be integrated in waste management factories and provide ideas for further enhancement of the proposed process.

2. Way of Work

2.1 Data Collection

Initiating our project, one of the most important tasks was to source suitable data to fit our objectives. This is a very crucial step as the chosen dataset profoundly influences the subsequent stages of model training and testing, and the achievable outcome.

For our goal, we combined data from different datasets and resources, consisting of multiple different images spanning distinct categories of waste, such as photos of metal, food, paper, plastic etc. By assimilating diverse images, we aspired to construct a dataset that will reflect real-world complexity of waste.

More specifically, we merged datasets from two distinct Kaggle repositories, titled: **"Garbage Classification (12 classes)"**[1] and **"Waste Segregation Image Dataset"**[2], successfully acquiring a comprehensive assortment of images for training our models.

One extra option would be the inclusion of pictures taken by us in our dataset to augment its diversity. However, our existing dataset was already comprehensive enough to cover the necessary categories. Thus, this step was not performed since it would not add any significant value to the final result.

[1] **"Garbage Classification (12 classes)"** dataset**:** https://www.kaggle.com/datasets/mostafaabla/garbage-classification/code

[2] **"Waste Segregation Image Dataset":** https://www.kaggle.com/datasets/aashidutt3/waste-segregation-image-dataset

As previously mentioned, our datasets consisted of a substantial number of images spanning various waste categories. To optimize our workflow and avoid complexity and continuous model crashes, we decided to consolidate certain categories from the two distinct datasets while also eliminating less common categories. One specific example of the category consolidation involves merging various glass colors into a single "glass" category, instead of maintaining separate categories for each distinct glass color.

2.2 Dataset Overview

Having gathered all the required images, we ended up with a final dataset of 7,838 images in total, separated in 7 folders, each corresponding to a different category.
We first uploaded the zipped dataset including all the available images in our Google Drive. Going forward, we extracted the images by developing a Python script[3] that could extract them from the initial folder "Original_Images.zip" to the folder "Extracted_Images" while preserving the original folder structure and contents. Subsequently, we executed a secondary script, which renamed each individual image[4] ,ensuring uniform naming convention across all images to reflect the category they belong to.
More specifically, all the names follow the below format:
{class name}_{4-digit image number}.{image format}

Moving forward, the dataset consists of the below categories and number of images:

- **2,011** images of glass
- **1,187** images of plastic
- **1,050** images of paper
- **985** images of food waste
- **945** images of batteries
- **891** images of cardboard
- **769** images of metal

At this point, it is useful to mention that all the related steps and processes were performed by using python scripts on Google Collab, in order to secure higher performance and better collaboration between the team members.

2.3 Data Preprocessing

First, we uploaded all dataset images to Google Drive and finalized the renaming process. After, a Python script[5] was developed to prepare the images for further machine learning applications. The goal was to secure successful results and avoid overfitting the models or gain inaccurate results.

[3] Python script developed for Data Extraction: 0.Data-Extraction.ipynb

[4] Python script developed for Image Renaming: 1.Rename Images.ipynb

[5] Python script developed for Data Preprocessing: 2.Data-Preprocessing.ipynb

The actions we performed are the following:

1. Initially, a series of image integrity checks were executed prior to proceeding with image preprocessing, serving as a proactive measure for data validation. The goal of this step is to attempt to open all the images and verify their integrity. If an image can be opened and verified without any issue, it is considered valid, otherwise, it's considered not valid, and a relevant error message is generated to demonstrate its invalid status.
2. Proceeding with the image resizing phase, we developed an algorithm designed to iterate through each subfolder of our dataset, processing the images of each class. The goal of this resizing process is to achieve consistent pixel count across all images, resulting in uniform dimensions. We determined that the optimal image size for model training was 224x224 pixels.
3. After iterating through each folder, a corresponding class folder is generated in the output directory, where all the preprocessed images will be saved. Since many machine learning and deep learning frameworks, such as TensorFlow and PyTorch, work well with NumPy arrays, thanks to their efficiency and compatibility in handling multi-dimensional arrays and matrices, our first step was to convert our data into Numpy arrays. This conversion was important to facilitate the required preprocessing techniques, amplifying the image data usability and performance, while implementing different calculations and transformations.
4. Later on, we proceeded with the normalization of all the dataset images, a similar technique to resizing, aiming to adjust the pixel values of an image to a standard range. In our case, we decided to use [0,1], dividing each pixel value by the maximum value, which is 255 for 8-bit images. This step is important as it significantly helps algorithms converge faster during training and prevents certain features from dominating the learning process due to their larger scale.
5. Moreover, we saved the preprocessed images in the relevant output class folder, converting them back to their original PIL format, always maintaining the original folder structure.
6. Finally, to account for potential errors, we integrated a feature in the algorithm to print a relevant error message, as well as the path of the image where the pre-processing was not successful. Subsequently, the algorithm continues with the preprocessing of the next image. This error handling mechanism was implemented to streamline the debugging process and avoid the occurrence of unidentified errors during the preprocessing phase.

2.4 Data Split

Upon completing the image preprocessing phase, the next step involved partitioning our dataset into 3 distinct subsets, each serving a unique purpose in our machine learning pipeline:

1) **Train**
2) **Test**
3) **Validation**

In the context of machine learning:

1) A training dataset is employed to train the model. During training, the model learns patterns, features, and relationships from the data to make accurate predictions. The training set should be large, to increase the model's understanding of various patterns.
2) A testing dataset is used to evaluate the performance of the model after training. It allows us to measure how well the model generalizes to new, unseen data.
3) A validation dataset helps fine-tune the model's hyperparameters and avoid overfitting, to optimize its predictive abilities.

The division process was simple, maintaining the original folder structure while incorporating a randomized assortment of images from each category. Considering the overall dataset size and optimal distribution, the following ratios were chosen: 70% for the Train dataset, 15% for the Test dataset and finally 15% for the Validation dataset.

Once again, to achieve our proper data split, a relevant Python script[6] was developed, ensuring a valid partitioning process. As a result, the algorithm divides each class folder into subsets suitable for training, validation, and testing. Notably, it uses a stratified sampling technique, preventing any one class from dominating a particular split. With this approach, we guarantee that each subset's class distribution accurately mirrors the distribution of classes in the original dataset. Furthermore, shuffling is integrated, to introduce randomness into the samples within each class before partitioning. This prevents potential biases stemming from the order of images in the original dataset.

By completing this step, we have finalized all the required preprocessing procedures, establishing the necessary groundwork to start training our models.

2.5 Data Info

To enhance our comprehension of the dataset and get clearer insights into its structure and contents, we will present some graphs and statistics[7].

As depicted in **Figure 1**, the histogram illustrates the category distribution within our dataset, in descending order.

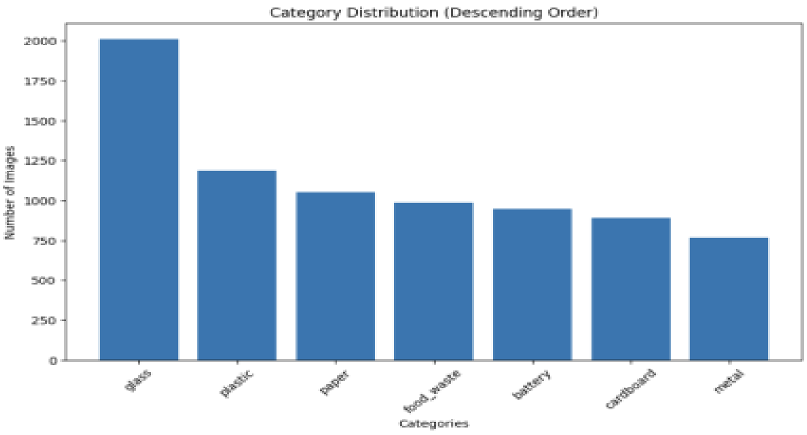


Figure 1: Category Distribution

[6] Python script developed for Data Split: 3.Data-Split.ipynb

[7] Python script developed for extracting Data Info: 4.Data Info Images.ipynb

These figures align with the statistics provided in Chapter 2.2 and can help us understand the different categories of waste included in our analysis, as well as the dataset's composition.

Additionally, to demonstrate the distribution of images among the different dataset categories we used the pie chart in **Figure 2**. Each slice corresponds to a category. The size of the slice represents the proportion of images belonging to that category compared to the total number of images.

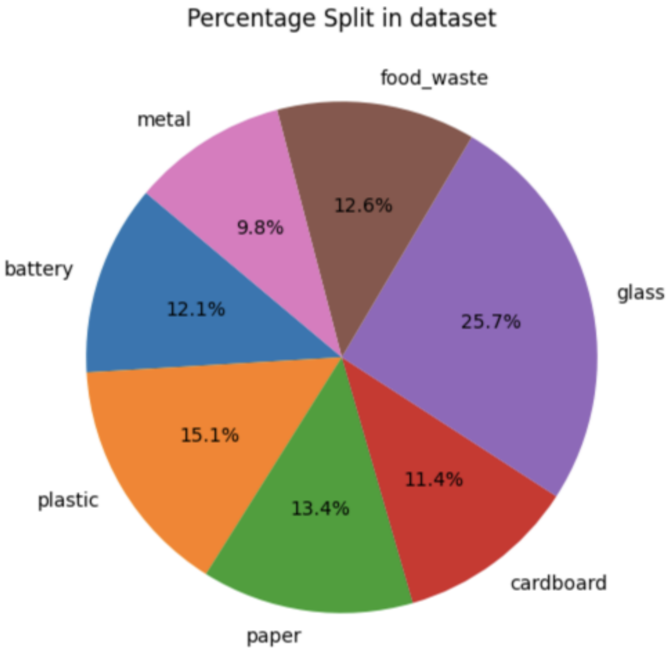
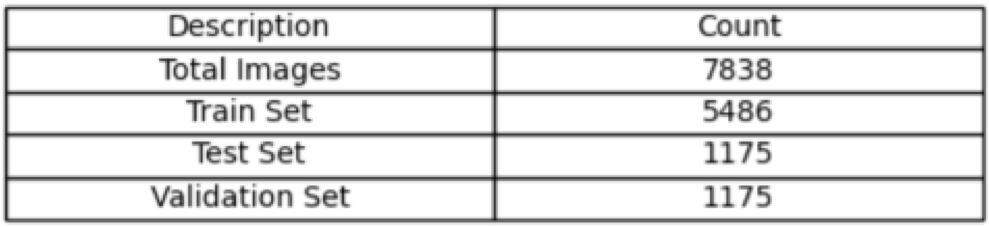


Figure 2: Percentage Split

Additionally, to demonstrate the chosen percentages of our train, test, and validation datasets, we present the corresponding results in **Table 1**, providing the exact number of images that is assigned to each dataset.



Description	Count
Total Images	7838
Train Set	5486
Test Set	1175
Validation Set	1175

Table 1: Data Split Count

At this point it is also useful to visualize the different data percentage compositions of the different waste categories that are defined, between the training, test and validation datasets, to ensure that a uniform distribution was achieved on all sets.

The below charts of **Figure 3** do indeed validate our intentions as they reflect an even distribution between the categories, as desired.

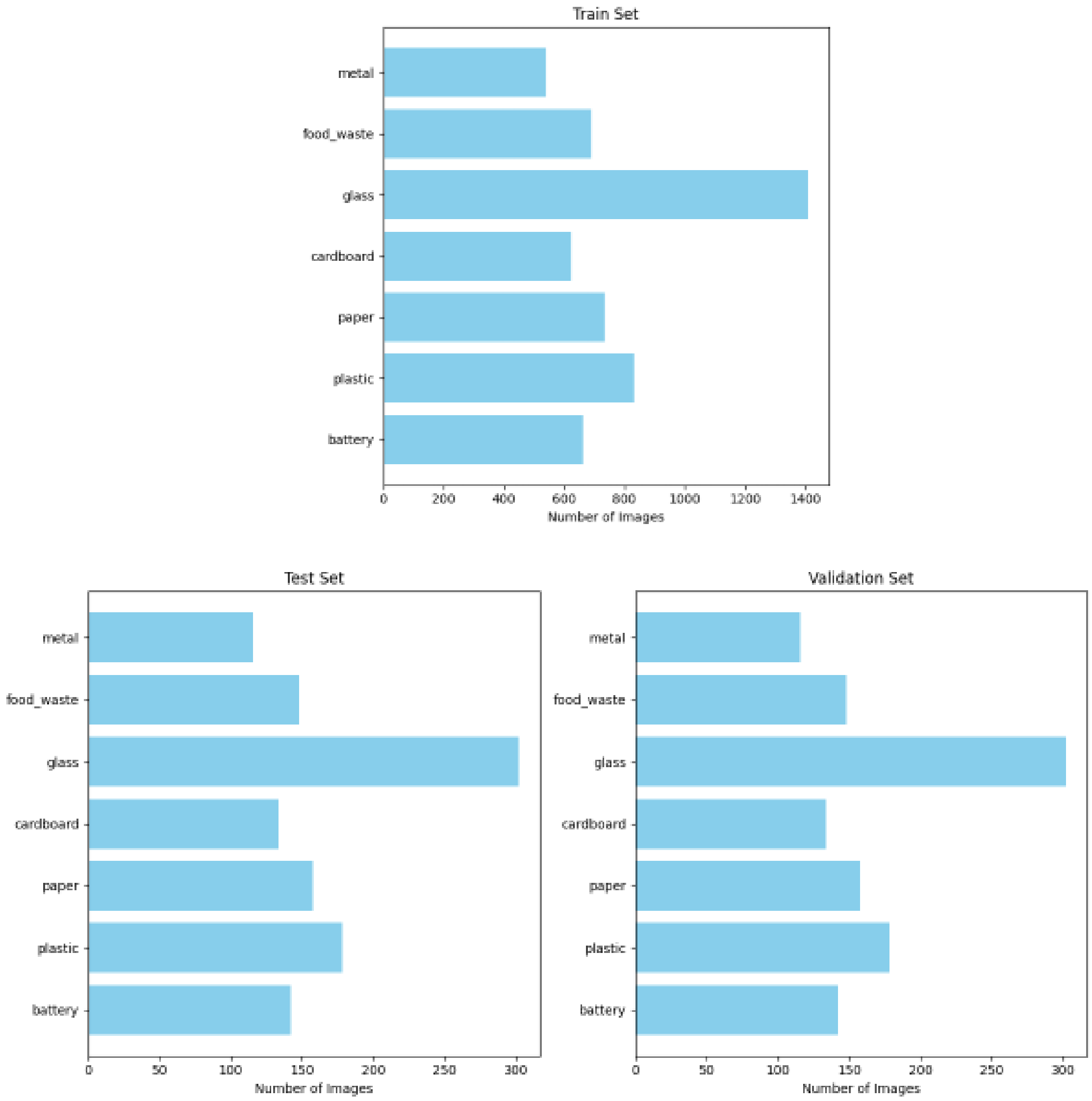


Figure 3: Data split between Training, Test and Validation Datasets

Finally, depicted in **Figure 4**, is a collection of representative instances from each waste category. These examples serve as reference points, allowing us to familiarize ourselves with the images utilized during the upcoming model training phases.

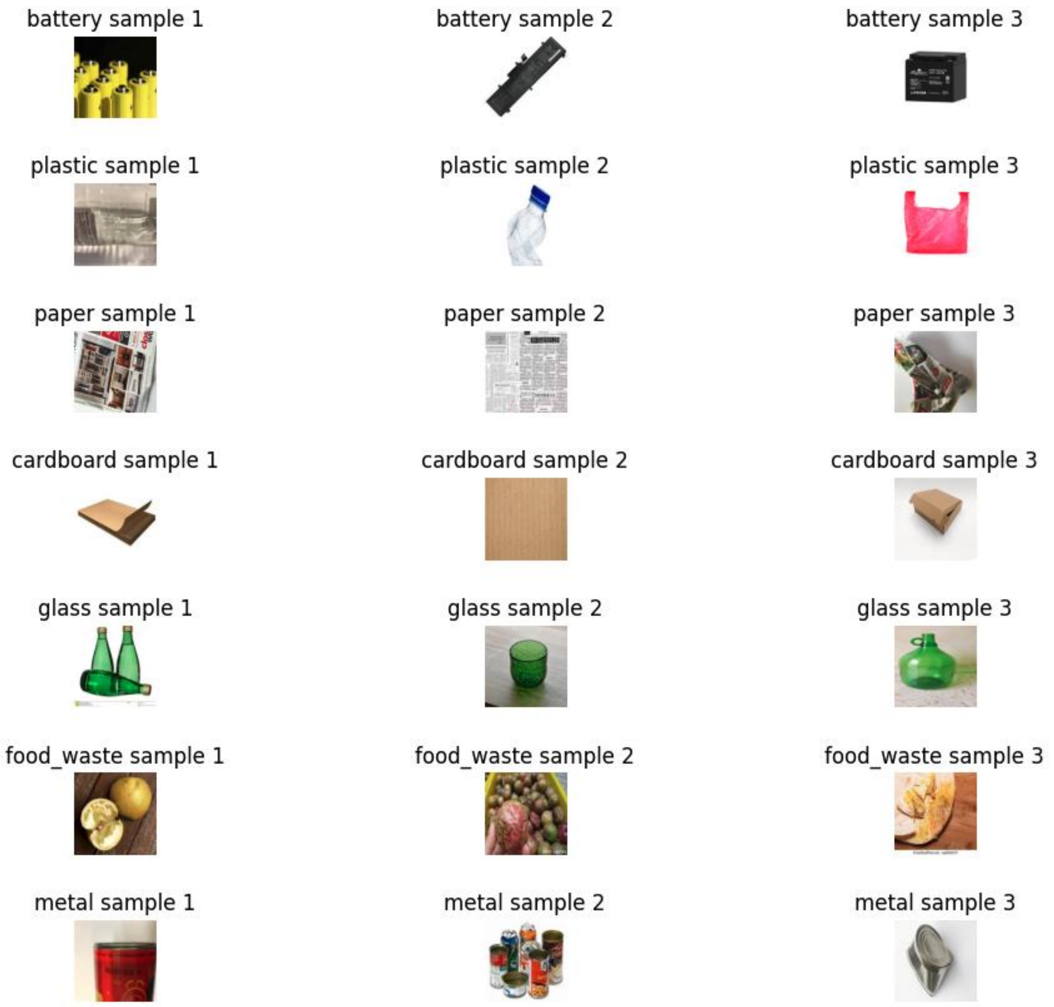


Figure 4: Sample Images Display

3. Methodology & Models Used

In this part of the project, we will describe the implementation of deep learning models, and the contribution of neural networks , in performing complex computations on vast volumes of image data, with the goal to create robust image classification and object detection models.

In general, image classification is a computer vision problem, as computers are called to recognize visual content and generate information from a given image. For our project, supervised algorithms for image classification will form the backbone of our approach, to end up with an algorithm able to detect and assign different images into separate waste categories. This kind of training enables the algorithm to categorize unknown data accurately. More analytically, the training of the machine learning model will be implemented by using a labeled dataset, where each image is already associated with a corresponding category. In that way, the model learns from these labeled examples and becomes able to make accurate predictions on new, unseen data. The models learn by using specific learning patterns and features along with the help of neural networks and can identify and separate different categories, e.g., separate plastic from paper.

3.1. NLP Architectures and systems

In this chapter, we will analyze further the Natural Language Processing architectures and systems that we used for the implementation of our task. These refer to frameworks and methodologies that are designed to help computers process and understand things like humans would do. One of the most notable NLP architectures that is also used in our project are the **CNNs - Convolutional Neural Networks.** These deep learning models are commonly used for analyzing and processing visual data such as images or videos, due to their unlimited capability to learn features and patterns of increasing complexity directly from pixel data.

While working with CNNs, we are called to choose to work with a specific CNN architecture that will fit our purpose and the available resources, as it will differentiate the arrangement and configuration of the different layers and components within the CNN model. For our purpose, we will use the MobileNetV2 architecture in order to define the data flow through the network, the feature extraction and the prediction making process.

MobileNetV2 is a significant architecture in the world of deep learning, particularly when efficient computation is required, like in mobile vision applications.
Because it is designed for mobile devices, it provides a good trade-off between computational efficiency and model accuracy.
MobileNetV2 is preferred for various mobile and edge applications, including object detection, image classification, facial recognition, etc.

3.2. Models Set-Up and Final Model decision

3.2.1. Initial Model Set-Up

While experimenting with different deep learning model architectures for image classification, we embarked on numerous attempts with no satisfactory results or efficient performance. For illustration purposes, we are going to present a manual model building process[8] that was developed, which proved to be quite complex and inefficient compared to the one that was chosen as our final model architecture. In this scenario, instead of leaning on a pre-existing pretrained model, we opted to construct a custom model, trained only on our available dataset, allowing us to tailor its architecture and parameters.
Our initial model was initiated with a function designed to create and return a TensorFlow/Keras model. At the start, an input layer was initiated for the neural network, the input tensor, with the given input shape of 224x224 and 3 RGB channels.
Then, the input tensor “inputs” pass through a data augmentation pipeline, consisting of two simple augmentation operations. These operations are **random flipping**, that flips images horizontally with a certain probability and **random rotation**, that applies random rotations to the images. Further details on data augmentation techniques will be provided in the next chapter (3.2.2).

[8] Python script developed for the initial model building: 6.initial_model.ipynb

Going forward, our entry block was formed, by implementing a step to rescale the input pixel values (originally in the range [0,255] representing the color intensity of the image pixels) to the range [0,1], assuring the standardization of the input data.

In our last step, we applied our first convolutional block which consisted of:

- **A 2D convolutional layer** to process the input data with 32 filters, a kernel size of 3x3, stride of 2, simultaneously adding a padding. This layer slides a filter over the input data (such as an image) to produce a feature map, capturing spatial information like edges, corners, and textures. Each filter will learn to recognize a different feature in the image. So, with 32 filters, the layer can potentially identify 32 different features. The kernel has dimensions of 3 pixels by 3 pixels. This is a common size for convolutional kernels and means that, when the network is looking at the image, it's considering 3x3 patches of pixels at a time. The stride determines how much the filter "steps over" as it slides across the input image. A stride of 2 means that the filter moves over by 2 pixels each time. This has the effect of down-sampling or reducing the spatial dimensions of the resulting feature map by roughly half, compared to a stride of 1. Last, padding is used to preserve the spatial dimensions of the input volume. By adding padding, the spatial dimensions of the output volume (after the convolution) can be made the same as the input volume. The padding mentioned here is "same" padding, which means enough zeros are added around the input image so that the output feature map has the same width and height as the input. This is particularly useful when we want the output feature map to have the same spatial dimensions as the input, even after the convolution.
- **Batch normalization** is applied to normalize the activations and improve the stability and efficiency of the training. Batch normalization normalizes the output of a previous layer by subtracting the batch mean and dividing by the batch standard deviation. This can help in stabilizing and accelerating the training of deep networks.
- **ReLU (Rectified Linear Unit) activation function** is applied to the output of the batch normalization layer. This function helps introduce non-linearity into the network and is commonly used in CNNs to model complex relationships between features.

This block was expected to serve as an early feature extractor in the network. Typically, in the initial layers of a CNN, the network learns low-level features like edges and simple textures. The stride of 2 reduces the spatial dimensions of the feature map, thus making the network more computationally efficient and also increasing its receptive field.

- **previous_block_activation = x** with this line, the algorithm stores the current tensor x to a variable named previous_block_activation to be used for residual connections later on.

- **A Middle block** is introduced, by creating a sequence of layers, with each layer utilizing a number of filters (128, 256, 512, 728), to achieve an iterative “deepening”, while preserving connections to the previous layers, known as residual connections. This loop iterates through different filters of various sizes, each time applying several operations to deepen the neural network. This procedure helps the network to hierarchically learn more complex features as it goes deeper into its architecture.

- **Operations in the Middle block:** The middle block begins with the application of ReLU activation function, introducing non-linearity into the network and allowing it to capture complex relationships within the data. A depthwise separable convolution splits the typical convolution process into two parts: a Depthwise Convolution and a Pointwise Convolution. This convolution can be more efficient than the standard one, although achieving similar or even better performance in some cases. Here, we used a kernel of size 3x3 and the same padding. As a result, the output will have the same spatial dimensions as the input. Then we normalize the activations from the previous layer, before applying a max pooling layer. With Max Pooling, we can reduce the spatial dimensions of the feature map, thus concentrating the information and making the network more computationally efficient. A kernel size of 3x3 and stride of 2 is used, which roughly halves the spatial dimensions.

- Following investigation, we decided to add the **Residual Connections** logic. Residual connections, or skip connections, are a technique introduced in the ResNet architecture to combat the vanishing gradient problem in deep networks. Here, the output of a previous layer (before all the transformations in the loop) is directly added to the output of the current loop iteration. This direct path helps in easier backpropagation of gradients and improves convergence. The layers.Conv2D(size, 1, strides=2, padding="same") is a projection that matches the spatial dimensions and depth of the residual to the main path, so they can be added together.

- In the **Final Convolutional block**, we mirror the earlier layers but with a significant increase in the number of filters to 1024. This was used to extract a wider range of features before the final classification step.

- For that matter, a **Global Average Pooling Layer** is inserted, in order to help us compute the average value of each feature map. This reduces the dimensions and provides a flat vector, making it suitable for feeding into a dense layer for classification.

- **A dropout of 50%** was applied so as to help fine-tune the model’s accuracy, before finally feeding the output to a final fully connected dense layer. This is the final layer, where the actual classification takes place. It has as many neurons as there are classes (num_classes), and it uses a softmax activation to produce probability distributions over the classes.

By utilizing the model.summary()command, we got to monitor and evaluate the behavior intended above, regarding the dimensionality of our feature maps.

An accuracy of **82%** was achieved, leading to pretty successful predictions.

However, upon completion of the model construction, due to the model’s complexity and the increased validation loss, as well as in combination with the low validation accuracy monitored up until the 15th epoch, this model was dropped. Since this particular model architecture was impractical and efficient, it was not selected, while other alternatives (such as DesNet169, VGG16 etc.) were tested. After thorough assessment, we arrived at the architecture outlined in section 3.2.2 as the most suitable choice for our final model, that exploits the rich input of transferred learning.

3.2.2. Final Model Set-Up

Having tested various alternatives, we concluded that the best option was to use an already pre-trained model, based on the **MobileNetV2** architecture as mentioned in chapter 3.1, since with this model we could get the best results in terms of accuracy and efficiency. Below is a thorough explanation of the procedure that was followed and the steps that were implemented to reach the final results with the relevant script that was developed[9].

Step 1: Defining Basic Model Constants

Initially, after importing all the necessary libraries from TensorFlow.Keras framework that we will utilize for our model building, we defined some basic model constants. Such constants are the ones described below:

- **data_dir = '/content/drive/MyDrive/Colab Notebooks/Preprocessed_Images'**
  Definition of the directory to which our model should refer so as to iterate over our test and validation sets of images

- **IMG_SIZE = (224, 224)**
  The input layer of the neural network is defined with images of accepted shape of 224x224 – the tuple representing the height and width of the image.

- **NUM_CLASSES = 7**
  The number of classes that we will train our model to classify the images of random garbage.
- **BATCH_SIZE = 64**
  Batch size determines the number of samples that will be processed before updating the model's weights.Taking into consideration that we were working with no limited memory constraints (by using Google Colab's T4 GPU which has ample memory capacity), we aimed at larger sized batches of images to be processed at a time, since smaller batches can introduce noise in the weight updates. In general, a larger batch size means more samples are processed together before updating weights. This can lead to more stable weight updates, as the influence of individual noisy samples is averaged out. Furthermore, while experimenting with the model, it was observed that smaller batches might even lead to unstable training. Last but not least, larger batches can often result in faster convergence in terms of wall-clock time since matrix operations can be parallelized more effectively.

- **EPOCHS = 80**
  Regarding the number of epochs, we opted for 80,which is an arbitrary moderate selection. In a nutshell, epochs signify how many times the entire training dataset passes through the model during the training. The main expectation while choosing this specific number of epochs, in the earlier convergence of the model, due to the callbacks, which are specialized functions, added later on in the process to monitor the training and make relevant decisions, as well as to reduce the prediction errors.

[9] Python script developed for the final model building: 5.Final_Model.ipynb

Step 2: Data Augmentation

Given that our dataset is a rather small one, we experimented with data augmentation techniques.
Data augmentation is a strategy used to increase the diversity of your training data without actually collecting new data. By applying small transformations to the input images, it artificially enlarges the training dataset and helps capture more angles of the items in the pictures. This helps to prevent overfitting, as the model will see slightly different variations of the images during training, making it generalize better to unseen data.

In our case, we established a pipeline where such techniques were both defined and incorporated into our model as well.

In detail:

- **RandomFlip("horizontal"):**
  This layer randomly flips the input image horizontally (left-to-right). With each training iteration, there's a 50% chance that an image will be flipped. This augmentation can be useful if the orientation of objects in the image doesn't matter. For instance, a cat might be facing left in one picture and right in another, and it's still a cat.

- **RandomRotation(0.1):**
  This layer randomly rotates the input image by a factor. The factor 0.1 implies a random rotation in the range of [-10% * 180 degrees, 10% * 180 degrees], which means the image could be rotated anywhere between -18 degrees to +18 degrees.

- **RandomZoom(0.2):**
  This layer is esigned to perform random zooms in each input image. A factor of 0.2 indicates a random zoom in the range between [1-0.2, 1+0.2] (or 80% to 120%). As a result, the image will be zoomed in or out by this random factor within the above specified range. By zooming in an image, we crop parts of the image, while zooming out introduces new extended borders.

- **RandomTranslation(0.1, 0.1):**
  With this layer, we shift the input image vertically, horizontally, or both ways. The provided values(0.1, 0.1) , signify that the image can be shifted randomly up to 10% of its initial width or height, towards any direction., as previously described.

Step 3: Building a custom classification model using transfer learning

- To begin with the process of creating our machine learning model for image classification, we will use as **Input Layer** the processed images, with the updated dimensions with input shape of 224x224 for 3 RGB channels.

- Proceeding to the next step with the **Data Augmentation** technique that is applied to the input of our final model, as described in Step 2.

- At this point, it is time to **load our Pre-Trained Model**: MobileNetV2 model has been extensively trained on a large and diverse dataset called ImageNet. Due to this training, it has acquired a deep understanding of multiple image features and patterns that can be leveraged for different tasks. This can lead to faster convergence and might even require less data. In our case as a starting point, we utilized this pre-trained model in the logic of transfer learning. In other words, it is being used as a feature extractor, which means that instead of using the entire MobileNetV2 model for classification, only its earlier layers, responsible for capturing image features, are used. MobileNetV2 model is loaded without its top (classification) layer, since this layer needs to be replaced with a layer specific for our current task (and not ImageNet oriented), and its weights are initialized from 'imagenet'.

- The layers of MobileNetV2 are frozen *(made non-trainable)*. This is because these layers already contain valuable information and updating them might lead to forgetting those patterns. In more detail, this means that during training, only the weights of the new layers that we added, are updated. This helps in preserving the generic features learned by the pre-trained model and quickly adapting the new layers to the new task.

- Going forward with further processing of the model, additional layers are added on top of the base model. These additional layers are designed to tailor the model’s architecture to the dataset's classification problem. Below are presented the extra layers that are added to our model:

    - **<u>GlobalAveragePooling2D</u>** layer effectively reduces the spatial dimensions, focusing on the important features. It also reduces the number of parameters, which can prevent overfitting. Global Average Pooling is a type of pooling operation used in Convolutional Neural Networks (CNNs), but instead of focusing over a local neighborhood (as in MaxPooling or AveragePooling), it pools over the entire height and width of each feature map.

      In that way we achieve:

        - *<u>Dimensionality Reduction:</u>* The primary utility of GAP is reducing the spatial dimensions of the feature maps. This is particularly useful in scenarios where the spatial dimensions of the feature maps can vary, and you want to connect the features to a fully connected layer which expects fixed-size inputs.
        - *<u>Preventing Overfitting:</u>* By reducing the spatial dimensions, GAP reduces the number of parameters when connecting to dense layers. Fewer parameters often mean less chance of overfitting.
        - *<u>Spatial Invariance:</u>* Since GAP computes the average of all the spatial positions, it allows the network to focus on the presence of features rather than their exact locations. This can be useful in tasks like image classification where the exact spatial location of a feature might not be as relevant as its presence or absence.

    - **<u>Additional dense layers</u>** (with 1024 and 512 units) increase the model's capacity. This means that the model can learn more complex patterns, which can be beneficial if the task at hand is more complex than the original task of the pre-trained model. As a result, with these extra layers we combine the already learned features from the earlier layers to make final decisions on the classification. With these layers we risk having more parameters, which means a higher chance of overfitting, especially with small datasets, like ours. However, this risk is mitigated with the dropout[10] layers placed after each dense layer. In these layers, we have selected the Rectified Linear Unit (ReLU) activation function. The ReLU activation function is popular in deep learning because of its simplicity and efficiency. It helps models learn complex patterns. This activation function basically introduces non-linearity into the model. The ReLU function is mathematically represented as f(x)=max(0,x). This means that if the input is positive, it returns the input value; if the input is negative or zero, it returns zero.

- **Prediction Layer** as the final prediction layer with 7 output units and a softmax activation function is added. This layer is actually responsible for producing the actual predictions based on the learned features by the preceding layers. Softmax Activation function is commonly used in multi-class classification tasks, where the main goal is to assign each image to one between several categories.

Step 4: Data Loading

ImageDataGenerator is used to perform real-time data augmentation during loading images in batches. The *training* and *validation* datasets are prepared using directory paths, and they're rescaled to have pixel values between 0 and 1.

Step 5: Model Training

- Model compilation: The model is compiled with Adam optimizer. This is a crucial step before training the neural network as it specifies various parameters on how the model will be trained and optimized. Adam optimizer specifically, is an optimization algorithm, commonly used for the purpose described above.
- Loss function: set to "sparse_categorical_crossentropy", which is appropriate for multiclass classification tasks with integer class labels. This function determines how well the model's predictions match with the true labels of the images during the training.
- Metrics: The model's performance metric is set to "Accuracy". This metric measures the ratio of correctly predicted images to the total number of images.

The model is then trained using the training dataset and validated with the validation dataset.

[10] **Dropout** is a regularization technique that helps in preventing overfitting. With dropout, during training, a fraction of neurons is randomly set to zero, ensuring that the network doesn't rely too heavily on any particular neuron. This means that for each forward and backward pass during training, the network architecture changes slightly as some neurons are deactivated. This prevents any single neuron from becoming overly specialized or overly reliant on particular neurons in the previous layer.

To be able to control the training process more dynamically and save model progress, several callbacks were added. These callbacks are used to ensure efficient training adaptation and retention of the best model. These Callbacks are the following:

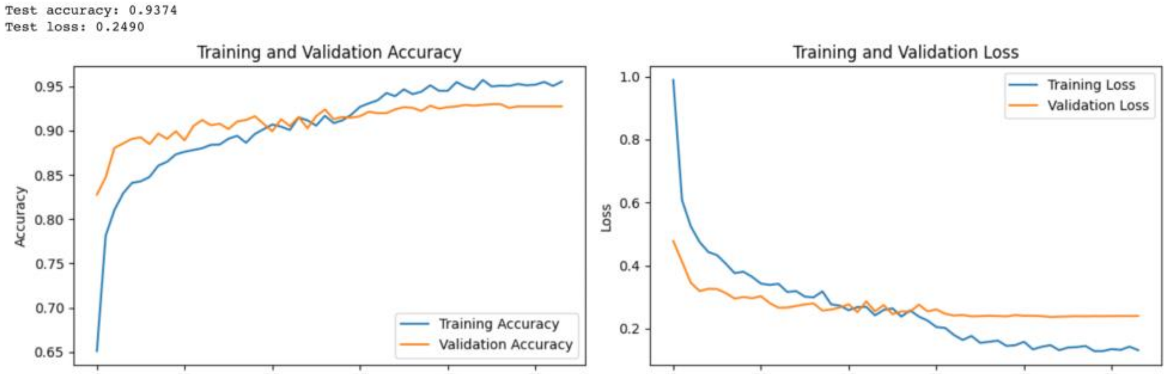
- **EarlyStopping:** This callback monitors the validation loss and stops training if the validation loss doesn't improve for 10 epochs . The function restores the best model weights when training is stopped. This prevents overfitting.
- **ReduceLROnPlateau**: This callback reduces the learning rate of the validation loss plateaus. The learning rate is reduced by a factor of 0.2 if no improvement is observed in the validation loss for 5 epochs. This helps with fine-tuning and avoids getting stuck in local minima by improving models converge rate.
- **ModelCheckpoint**: This callback saves only the best model weights to a file named 'best_model.h5' based on the validation loss.

Step 6: Fine Tuning

After the initial training, the last 20 layers of the base model (MobileNetV2) were unfrozen, allowing them to be trained. This means that during subsequent training, these layers will also have their weights updated. Thus, the model is then recompiled (meaning that it's made ready for training again), and further trained for another 10 epochs, with a lower learning rate. The reason for a lower learning rate is to ensure that the already learned weights in the unfrozen layers don't change drastically. We want to make small adjustments to them, not large shifts, in an effort to fine-tune our model's accuracy. With the above fine-tuning process we help the model to adapt to this new task while maintaining the learned features of the initial training, leading to better predictive ability. After training and fine-tuning, the model and its weights are saved to Google Drive for future usage and manipulation.

Step 7: Model Evaluation

Our saved test dataset is loaded, and the model's performance is evaluated on it, monitoring the training and validation accuracy and loss metrics, which we plotted using matplotlib to visualize how the model performed during training.
By looking at **Figure 5**, we note that the predictive accuracy of our model has increased significantly, reaching almost **94%**.
According to the Accuracy graph, we observe a consistent upward pattern for both Training and Validation curves, as they tend to smoothly increase. In fact, the validation curve closely follows the training accuracy curve, suggesting good generalization, and mitigating the risk of potential overfitting.
As depicted in the Loss curve, a sharp decline is evident in the training loss curve, and correspondingly the validation loss curve follows the same trend with a smoother downward path. Once more these two curves maintain proximity, signifying a progressive improvement to our model over time.
By combining the gained insights from both accuracy and loss graphs, where both training and validation accuracy ascend, while training and validation loss descend over epochs, we can conclude that we have achieved effective learning to our model.



```
Test accuracy: 0.9374
Test loss: 0.2490
```

Figure 5: Accuracy and Loss graphs

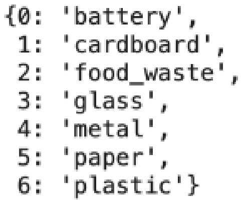
4. Results & Visualizations

Having completed the training phase of our final model and having evaluated its accuracy, it is useful to proceed with some random manual testing on a selection of test images, to check and re-confirm its predictive capability. To achieve this objective, we proceeded with loading, visualizing, and passing the images through the model, so as to get predictions.

Fortunately, all the performed tests were successful and the model indeed managed to correctly predict the class/category of the given waste images, without any losses. Below, we will illustrate some of the successfully conducted predictions to better demonstrate the initial objective of our model.

Before proceeding with the relevant images and their corresponding predictions, it is useful to mention that each class was assigned a specific number/label in order for the algorithm to be able to identify and separate the classes. As a result, the expected outcome after loading an image, is the relevant class label (e.g 0, 1,2 etc) along with the corresponding class category name.

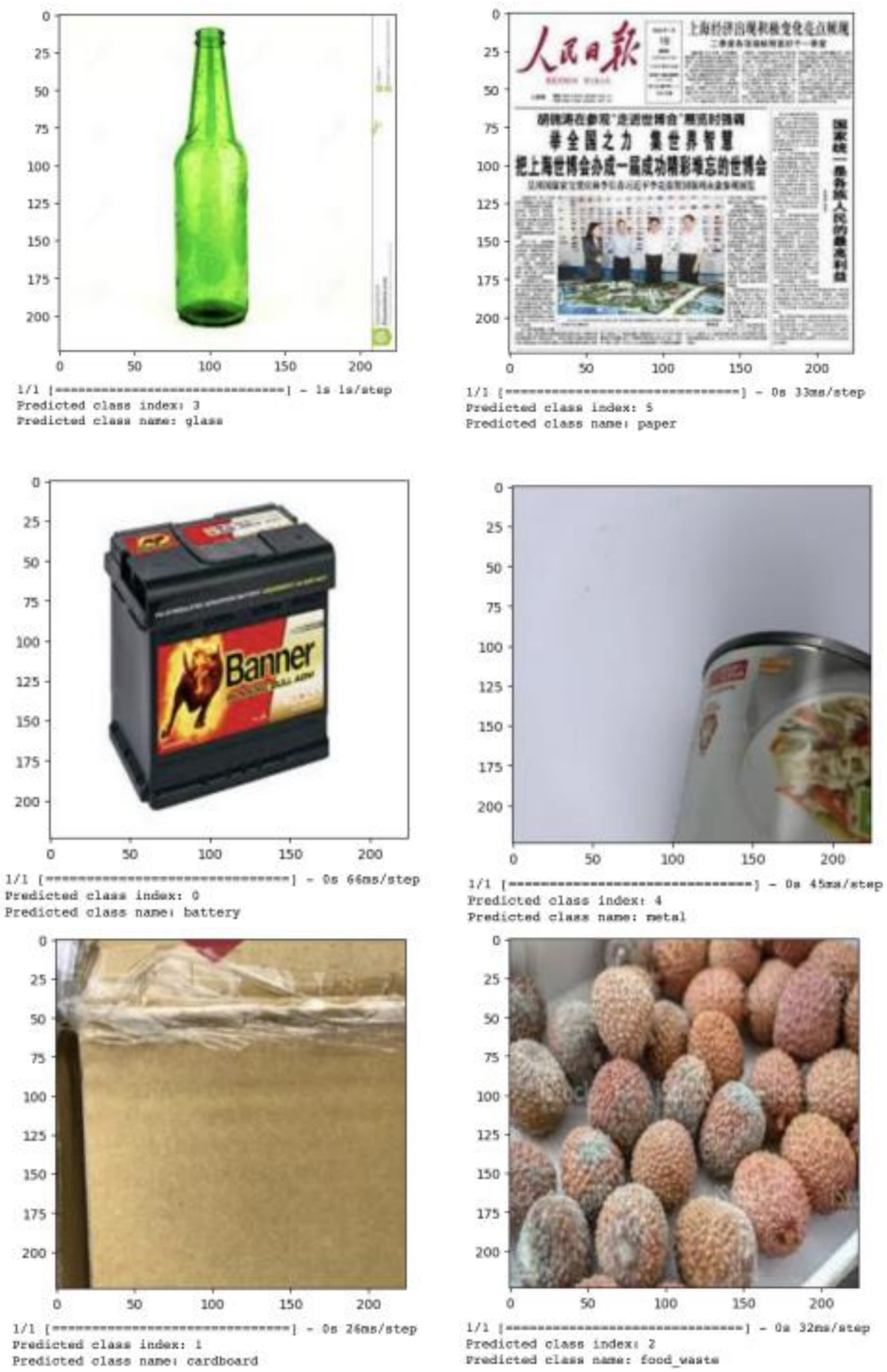
In **Figure 6**, we can see the exact class labels, and their corresponding class names.



```
{0: 'battery',
 1: 'cardboard',
 2: 'food_waste',
 3: 'glass',
 4: 'metal',
 5: 'paper',
 6: 'plastic'}
```

Figure 6: Class Labels

Going forward, in **Figure 7**, we see some of the above-mentioned manual checks where the algorithm managed to correctly predict the waste class.



```
1/1 [==============================] - 1s 1s/step
Predicted class index: 3
Predicted class name: glass
```

```
1/1 [==============================] - 0s 33ms/step
Predicted class index: 5
Predicted class name: paper
```

```
1/1 [==============================] - 0s 66ms/step
Predicted class index: 0
Predicted class name: battery
```

```
1/1 [==============================] - 0s 45ms/step
Predicted class index: 4
Predicted class name: metal
```

```
1/1 [==============================] - 0s 26ms/step
Predicted class index: 1
Predicted class name: cardboard
```

```
1/1 [==============================] - 0s 32ms/step
Predicted class index: 2
Predicted class name: food_waste
```

Figure 7: Manual Prediction Tests

As we can see, the algorithm managed to make correct predictions for all the available categories of waste, based on the provided images.

Last, but not least, in **Figure 8** we observe a detailed illustration of the architecture of our trained neural network model using Keras. More specifically, the below figure includes the layers, their corresponding connections, shapes, and layer names by providing an analytical view of the overall model's structure.

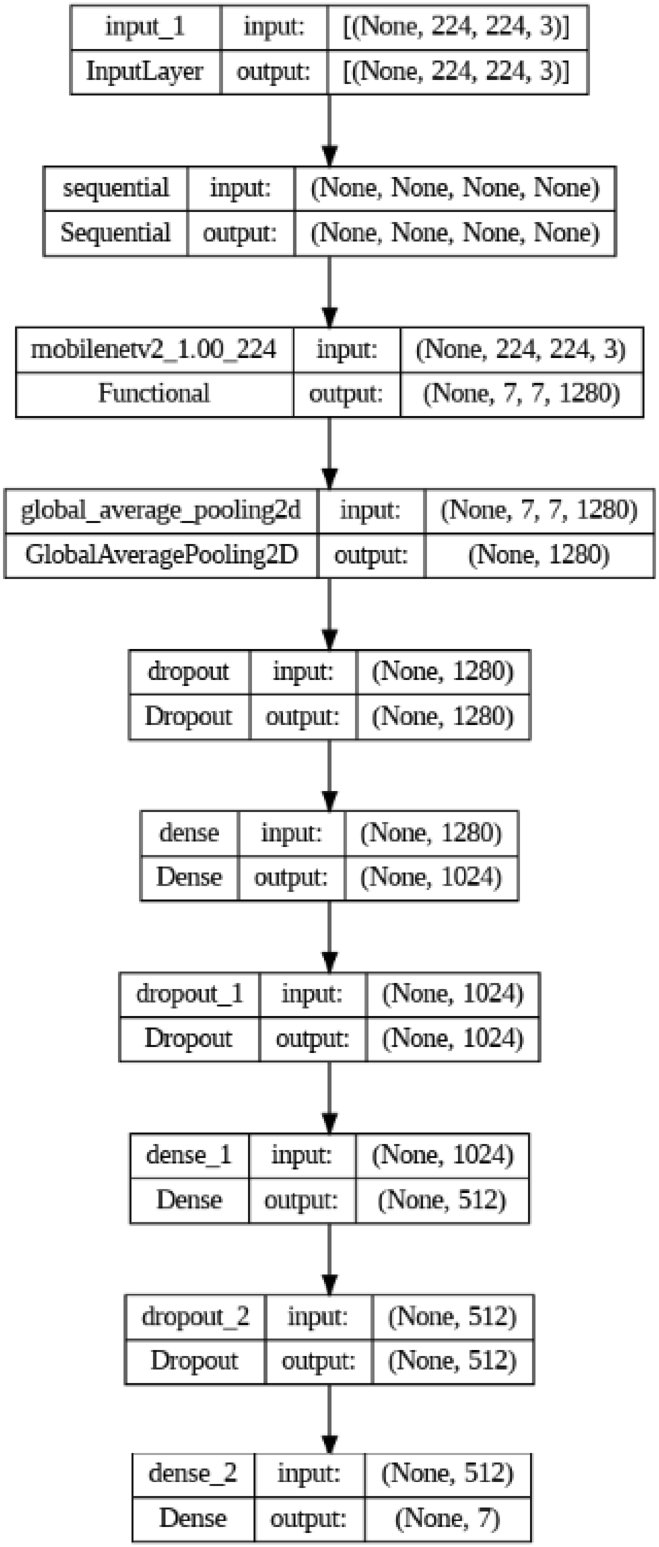

input_1 | input: | [(None, 224, 224, 3)]
InputLayer | output: | [(None, 224, 224, 3)]
sequential | input: | (None, None, None, None)
Sequential | output: | (None, None, None, None)
mobilenetv2_1.00_224 | input: | (None, 224, 224, 3)
Functional | output: | (None, 7, 7, 1280)
global_average_pooling2d | input: | (None, 7, 7, 1280)
GlobalAveragePooling2D | output: | (None, 1280)
dropout | input: | (None, 1280)
Dropout | output: | (None, 1280)
dense | input: | (None, 1280)
Dense | output: | (None, 1024)
dropout_1 | input: | (None, 1024)
Dropout | output: | (None, 1024)
dense_1 | input: | (None, 1024)
Dense | output: | (None, 512)
dropout_2 | input: | (None, 512)
Dropout | output: | (None, 512)
dense_2 | input: | (None, 512)
Dense | output: | (None, 7)

Figure 8: Model Architecture

5. Error Analysis

5.1. Runtime disruptions - Resources Requirement

In the long process of trial and error, one of the major issues that we encountered was the lack of the necessary computational resources required by some of the versions with which we were experimenting along the way. Making use of Google's Colaborary - a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs, we came across too many connectivity disruptions and timeouts that forced us to repeat our training process multiple times.

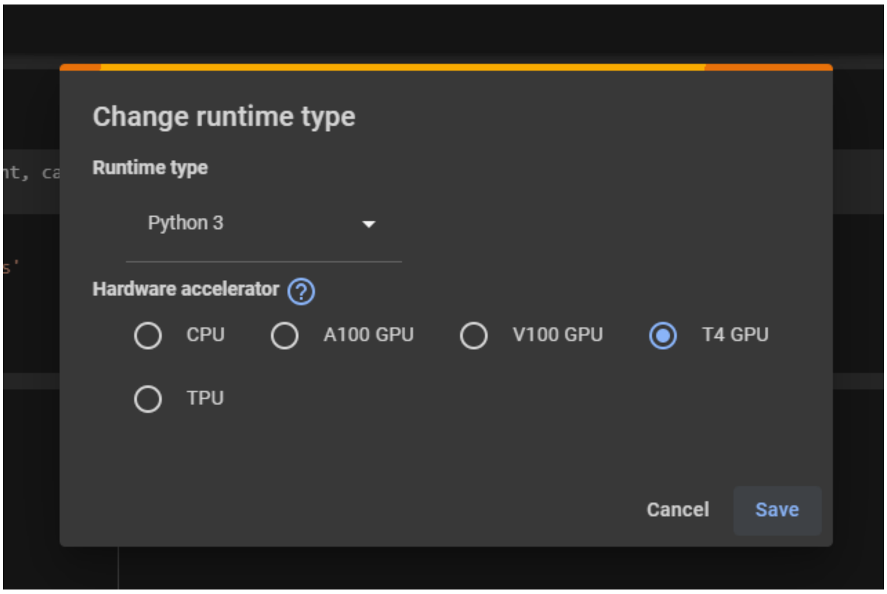
We had to restart our runtime multiple times to get our preferred GPU. Remember, the time and resources you get with the GPU in Colab are shared and have limitations, especially on the free tier. To resolve this, we selected the option to pay for unit consumption as we go, unlocking more powerful resources.

Google Colab provides access to NVIDIA GPUs and TPUs for accelerating deep learning tasks. Two such options we experimented with:

- V100 GPU, offering high-performance deep learning and computational power with advanced Tensor Cores.

- T4 GPU, offering efficient inference and moderate deep learning training with versatile precision modes.

Our project was delivered with T4 GPU in the end, as depicted in **Figure 9**.



Change runtime type

Runtime type

Python 3

Hardware accelerator

○ CPU ○ A100 GPU ○ V100 GPU ● T4 GPU

○ TPU

Figure 9: Change of Runtime type

5.2. Classes re-arrangement and limitation

As mentioned in our introduction, with the datasets we sourced and manipulated from Kaggle, we had originally formed 15 classes/categories of waste. We initially chose to use them all, quickly facing and recognizing the significant computational demands and prolonged epoch durations that this entailed during the training phase. Plus, our initial models underperformed, largely because the intricate details in the images led to complications. For instance, distinctions like white_glass, green_glass, and brown_glass added undue complexity to our training process. To address this, we consolidated these into a singular "glass" category, refining our project's scope to classify just 7 categories as per our final deliverable.

6. Final Comments and Future Steps

Having finalized our project and reaching the final stage of evaluating and re-assessing potential future actions, the significance and usability of the developed tool were more obvious than ever. It is proof of how AI, machine learning and the overall proper use of technology, in combination with human actions, can contribute in creating a better world, with less challenges, waste or operational inefficiencies. More specifically, this waste segregation tool, by leveraging the power of automatic image recognition, would significantly contribute in refining planning and management, particularly within the field of production and manufacturing at first. On a later stage, its applicability could extend beyond these sectors, having the potential to revolutionize the waste management operations of different business areas as well. By implementing such add-ons in a production line, it would help businesses reduce their waste, increase their recycling levels, and reduce their overall footprint. Consequently, it would significantly assist those businesses to comply with the newly emerging rules and regulations, as well as reduce their unexpected costs caused due to fines or violations.

However, if we want to consider the practical implementation of the tool within the production line, this demands more tests, changes, and fine tuning that we need to pay attention to before a full-scale deployment. Regarding the backend infrastructure, it would be crucial to obtain appropriate tools, with sufficient computational power, memory, and speed. This would facilitate the expansion of the tool’s capabilities, to be able to identify even more waste categories and make the whole waste segregation procedure more accurate and detailed. Additionally, the same algorithm, with proper alterations and the necessary computational power, could potentially recognize and segregate different waste categories from one single image, instead of having to strictly have each item in a separate photo. All the above would make the whole procedure more practical and streamlined for the human operators in the production line.

Apart from the programming part, some extra steps would need to be implemented in the on-site infrastructure of the production line as well. For example, the operators of the organization where the tool would be installed, need to have already placed machinery that will capture pictures of the wastes that will later send them to our AI tool, as well as install machinery that will actually segregate the waste as indicated by the algorithm, after the class recognition. All these assets should be inter-connected in order to collaborate for the waste segregation goal autonomously.

If the above prerequisites are met, then this could be a revolutionary solution that could secure better treatment for the environment, and significant cost savings.

7. Team Members & Time Plan

Our team consisted of 3 members with diverse backgrounds to secure efficiency and completeness for all the different stages that were required for this project. The team members were Lida Vratsanou, Kostas Velissaris and Katerina Kalamara. All of the 3 members were involved in all the stages of the project, each emphasizing more in different aspects of it during the whole journey.
Specifically, data collection was performed collaboratively, since no manual pictures were taken, or any particular additions/edits were performed on the found datasets. Lida Vratsanou and Kostas Velissaris were mostly responsible for the programming part of the project, in terms of data pre-processing and different models' development and evaluation, respectively.
Katerina Kalamara mostly dealt with the Business section of the project, by producing the final report, presentation, and some fine tuning.
Considering this informal separation of the tasks, we want to highlight once more the great cooperation and support within our team, since despite the different tasks that were assigned to each individual, all team members supported each other during the implementation of all the different steps of the procedure, in order to gain holistic knowledge, and secure inclusiveness of the thoughts and opinions of all members.

With this collective effort, we managed to keep all 3 members updated for all the steps of the project and not leave unanswered questions or unclear points at any part of the project.
Apart from efficient collaboration, in order to successfully finalize this project, a well-organized time plan was followed, to avoid delays, re-work or risking not meeting the deadline. The Gantt Chart of **Figure 10** presents the analytical time plan that was followed for the successful and timely completion of the project.

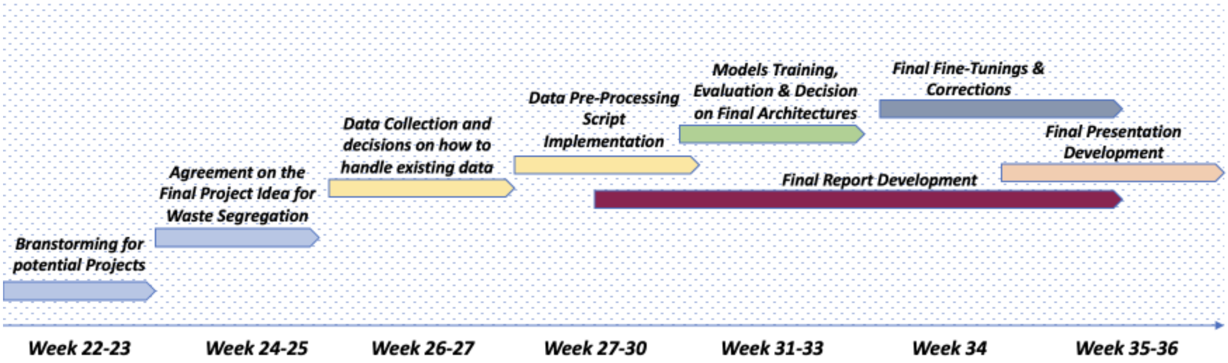


Figure 10: Implementation Time plan

As can be seen, the first weeks were dedicated to deciding the goal of the project, finding the data, and agreeing on the way we will handle data for our project. The next weeks were dedicated to data pre-processing, and the model's construction and evaluation. At the same time, we had already begun writing the first parts of the report, and once new content became available, we updated the relevant chapters. With these simultaneous actions, we established efficient completion of all the tasks. After finalizing the coding parts, all members were dedicated to fine tuning and finalizing the report by applying corrections or changes that may be needed. The last step of the journey was the creation of the final presentation , in order to complete all the required deliverables for our project. Having completed all the above steps in the pre-set time, we have successfully finalized our project.

8. Bibliography

1. *Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich Google Inc. University of North Carolina, Chapel Hill University of Michigan, Ann Arbor Magic Leap Inc., Going Deeper with Convolutions,* *https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf*
2. *Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos, Image Segmentation Using Deep Learning: A Survey,* *https://arxiv.org/pdf/2001.05566.pdf*
3. *Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen, MobileNetV2: Inverted Residuals and Linear Bottlenecks,* *https://arxiv.org/abs/1801.04381v4*
4. *Vladimir Iglovikov,Alexey Shvets, TernausNet: U-Net with VGG11 Encoder Pre-Trained on ImageNet for Image Segmentation,* *https://arxiv.org/pdf/1801.05746.pdf*
5. *MobileNet, MobileNetV2, and MobileNetV3,* *https://keras.io/api/applications/mobilenet/*
6. https://www.e2enetworks.com/blog/nlp-101-nlp-architecture-explained