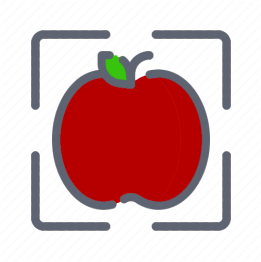
Alma Mater Studiorum · University of Bologna

Project 4 - Food Recognition[1]



91250 - Deep Learning

Giuseppe Murro (giuseppe.murro@studio.unibo.it)
Salvatore Pisciotta (salvatore.pisciotta2@studio.unibo.it)

June 24, 2021

[1]The code for the project is publicly available on GitHub

Contents

1 Introduction

Recognizing food from images is an extremely useful tool for a variety of use cases. In particular, it would allow people to track their food intake by simply taking a picture of what they consume. Current nutritional monitoring apps use traditional methods for food recognition, with algorithms based on classification. However, such applications are either semi-automatic, which identifies a group of possible types of food and requires user interaction. Image-based food recognition has in the past few years made substantial progress thanks to advances in deep learning. But food recognition remains a difficult problem for a variety of reasons.

2 Problem

2.1 Description

The goal of this project is to train models which can look at images of food items and detect, in the form of image segmentation, the individual food element. The categories of food to recognize are limited to a given set of categories decided in function of the dataset used. Once that the neural networks used for the food recognition are trained, prediction on test set of images are made and based on them results are computed using some metrics.

2.2 Dataset

The dataset used for this project is the dataset provided in the Food Recognition challenge proposed by AICrowd [4], in particular the dataset is divided in training set, val set and test set. The dataset is structured as follow:

- *Training Set* of 24120 (as RGB images) food images with their corresponding 39328 annotations in MS-COCO format
- *Validation Set* of 1269 (as RGB images) food images with their corresponding 2053 annotations in MS-COCO format
- *Test Set* where there are provided the same images as the validation set

3 Approach

3.1 Preprocessing

The first preprocessing operation that has been done is the filtering of the categories that model can detect. It has been necessary to do this since the dataset provides lots of categories and images that, if totally used for training a neural network, would require high computation capability. So, it has been agreed to reduce the number of categories down to 16 and in the same way include only those images of the dataset where these classes are present. Following this idea, for the training it has been created a subset of the entire dataset. The categories that has been taken into account are the most present in the dataset considering the entire distribution of categories that is shown in 3.1:

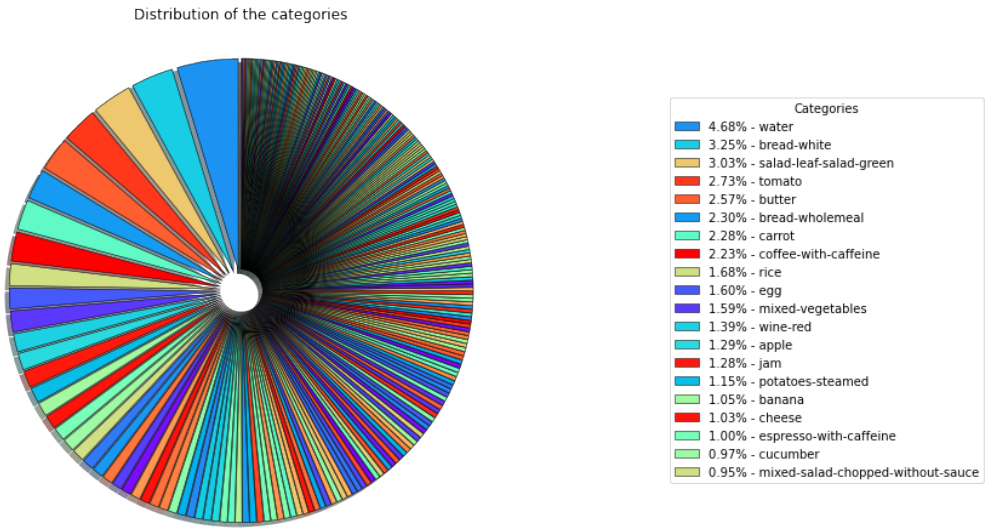


Figure 3.1: Pie chart of the distribution of the categories in the dataset

As we can see the most frequent categories are:

1. *water*
2. *bread-white*
3. *salad-leaf-salad-green*
4. *tomato*
5. *butter*
6. *carrot*
7. *coffee-with-caffeine*

8. *rice*
9. *egg*
10. *mixed-vegetables*
11. *wine-red*
12. *apple*
13. *jam*
14. *potatoes-steamed*
15. *banana*
16. *cheese*

These first 16 classes are the selected ones. Including the *background* class, the total number of categories has become 17. The dataset has been reduced as follow:

- *Train set* : from 24120 images to 10819 (and annotations from 39328 annotations to 12869)
- *Val set*: from 1269 images to 554
- *Test set* : from 1269 images to 554

3.2 Multichannel masks

One of the most important choice to do was to decide how to encode the classes present in the images into the corresponding masks. In particular how to represent the different categories that are inside a single image. The selected technique, in order to deal with multiclass images, is the use of the so-called multichannel masks associated to every image. This technique consists in creating an output channel for each of the possible classes, where every single channel represent the target by one-hot encoding the class labels. In order to apply this method it has been used Numpy matrices putting in the last dimension the encoding of every channel. A representation of this methodology is represented in 3.2:

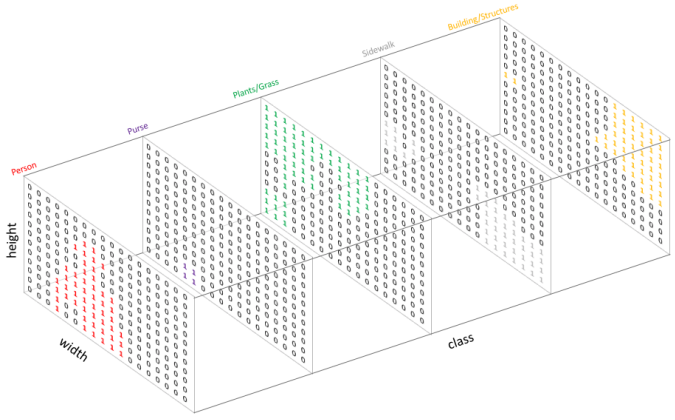


Figure 3.2: Example of multichannel mask with one-hot encoded channels for every category

3.3 Data Generator

Even if the dataset used for the training is a subset of the original, it is still too big to be loaded into memory at once and you could run out of RAM. That is the reason why there was the need to find other ways to do that task efficiently. A good solution has been found in the use of *data generators*, that can read images on the the fly when they are used for training.

Keras allow to create custom data generators inheriting the properties of *keras.utils.Sequence* in order to leverage nice functionalities such as multiprocessing. Every single *Sequence* implements the methods:

- *__getitem__*, that should return one batch of data *(X,y)*
- *__len__*, that should return the number of batches per epoch
- *on_epoch_end*, which is triggered once at the very beginning as well as at the end of each epoch. It allow to shuffle the dataset to make some randomness at each epoch

Each image and mask is processed by the *DataGenerator* using pycocotools API [8] to convert annotations in multichannel masks and then they are resized to *128x128*. So the shape of the input and output data in each batch become:

- *X.shape = [batch_size, 128, 128, 3]*
- *y.shape = [batch_size, 128, 128, 17]*

The channel 0 on the output represent the background and it is composed by all 1s where the category mask is absent. The example of how the input and output are formed is shown in figure 3.3.

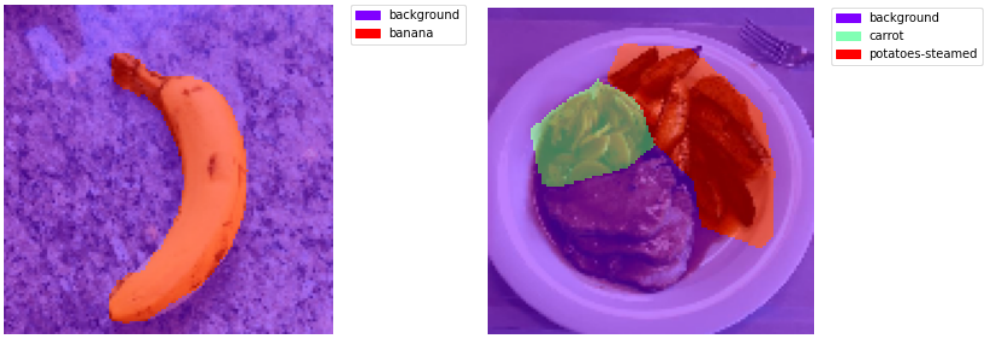


Figure 3.3: Examples of image masks

3.4 Data Augmentation

Data augmentation is the application of a wide range of techniques used to generate a new randomly transformed batch of training samples replacing the original one [12]. Furthermore, it is a good method in order to avoid the phenomenon of the overfitting that could affect the model in the training phase. In order to apply augmentation to the project, Data Generator allow to create a batch of augmented images and their correspondent masks. The modifications are applied to

the images using the Keras ImageDataGenerator class that, passing some parameters, applies the same transformations to the image and its related mask. The used transformations are:

- *Rotation* (rotation range = 5°)
- *Width shift* (width shift range = 1%)
- *Height shift* (height shift range = 1%)
- *Shear shift* (shear range = 1%)
- *Horizontal flip*
- *Vertical flip*
- *Zoom*(zoom range = from 90% to 100%)

The figure 3.4 shows an example of how augmentation is computed on the images of figure 3.3.

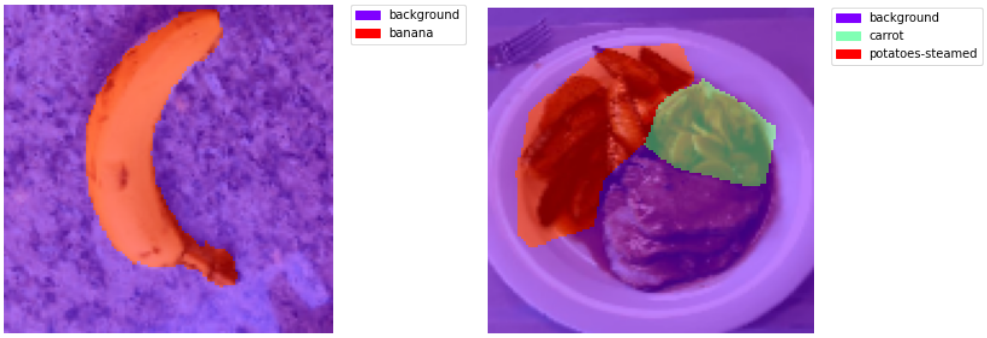


Figure 3.4: Examples of augmentation

3.5 Activation functions

The activation function of the last layer in a convolutional neural network is the one in charge of classifying the category each pixel belongs to. Since the output should be a probability, softmax and sigmoid are the principal candidates.

For *multi-label classification problem*, where there is more than one "right answer" (the outputs are not mutually exclusive), it is usually better to use a sigmoid function on each raw output independently. Indeed, the sigmoid allows to have high probability for all classes, for only some of them, or for none of them.

Instead for *multi-class classification problem*, where there is only one "right answer" (the outputs are mutually exclusive), it is usually better to use a softmax function. Indeed, the softmax enforces that the sum of the probabilities of output classes are equal to one, so in order to increase the probability of a particular class, the model must correspondingly decrease the probability of at least one of the other classes.

Upon an inspection of the provided dataset, it has been noticed that overlapping of masks is present in the images, for this reason it has been favored an output with sigmoid.

3.6 Loss function

Semantic segmentation models usually use a simple categorical cross-entropy loss function during training. However, sometimes slightly more advanced loss functions allow to get the granular information of an image. In particular the paper of Jadon S. [6] discusses the conditions to determine which loss function might be useful in a given scenario and she classifies loss functions for segmentation into 2 macro categories: distribution based and region based. The most interesting ones per each category, analyzing the dataset used in this project, are *focal loss* and *dice loss*.

3.6.1 Focal loss

The *focal loss* is a distribution based loss that works better with highly-imbalanced dataset. It is an improvement to the standard cross-entropy criterion. This is done by changing its shape such that the loss assigned to well-classified examples is down-weighted. In this loss function, the cross-entropy loss is scaled with the scaling factors decaying at zero as the confidence in the correct classes increases. The scaling factor automatically down weights the contribution of easy examples at training time and focuses on the hard ones. The definition given by Lin T. et al. [9] is:

$$FL(p_t) = -\alpha_t(1-p_t)^{\gamma} \cdot \log(p_t)$$

where $\gamma \in (0, 5]$ and α is a weighting factor $\in [0, 1]$

3.6.2 Dice loss

The Dice coefficient is a widely used metric in computer vision community to compute the similarity between two images. This measure ranges from 0 to 1 where a Dice coefficient of 1 denotes perfect and complete overlap. The Dice coefficient was originally developed for binary data, and it can be calculated as:

$$Dice = \frac{2|A \cap B|}{|A|+|B|}$$

where $|A \cap B|$ represents the common elements between sets A and B, and $|A|$ represents the number of elements in set A (and likewise for set B). In order to formulate a loss function which can be minimized, it can be simply used $1 - Dice$. This loss function is known as *soft Dice loss*. It is a region based loss and as Dice coefficient is non-convex in nature, so it has been modified to make it more tractable. It is called in this way because it directly uses the predicted probabilities instead of thresholding and converting them into a binary mask (see figure 3.5).

3.6.3 Combo Loss

The final loss that has been used in this project is a sum of *dice loss* and *focal loss*. It attempts to combine their strengths and mitigate their drawbacks.

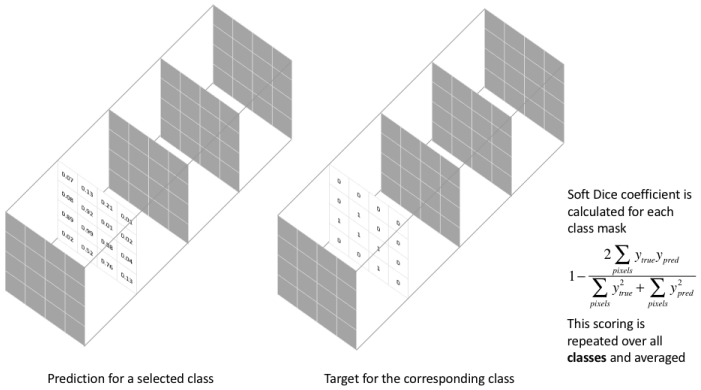


Figure 3.5: How Dice loss is computed [7]

3.7 Class weights

Semantic segmentation datasets can be highly imbalanced meaning that particular class pixels can be present more inside images than that of other classes. Since segmentation problems can be treated as per-pixel classification problems we can deal with imbalance problem by weighing the loss function to account for this. It's a simple and elegant way to deal with this problem. The class weights are computed for each category with the following formula [3]:

$$w(y) = \frac{n\_samples}{n\_classes \cdot counts(y)}$$

4 Model

4.1 Architectures

The chosen architecture for the problem resolution is U-Net, originally developed for segmenting biomedical images. When visualized its architecture looks like the letter U and hence the name U-Net. Its architecture is made up of two parts, the left part (the contracting path) and the right part (the expansive path). The purpose of the contracting path is to capture context, while the role of the expansive path is to aid in precise localization. Two U-Net like models have been investigated in order to compare their effectiveness on food recognition task.

4.1.1 Basic U-Net

The first architecture is based on the original one proposed by Olaf Ronneberger et al. [11], and it is is illustrated in figure 4.1.

The contracting path starts from an image of size *(128x128x3)* and follows the typical architecture of a convolutional network. It consists of four repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step the number of feature channels are doubled (starting from 32 up to 512), while the spatial dimensions is cutted by half (starting from 128 up to 8). The bottleneck uses two 3X3 convolutional layers to produce an internal encoding of the input image.

The expansion section consists of several expansion blocks with each block passing the input to two 3X3 convolutional layers and a 2x2 "up-convolution" that halves the number of feature channels. It also includes a concatenation with the correspondingly cropped feature map from the contracting path. The cropping is necessary due to the loss of border pixels in every convolution. In the end, 1X1 convolutional layer is used to produce 17 feature maps as same as the number of categories which are desired in the output.

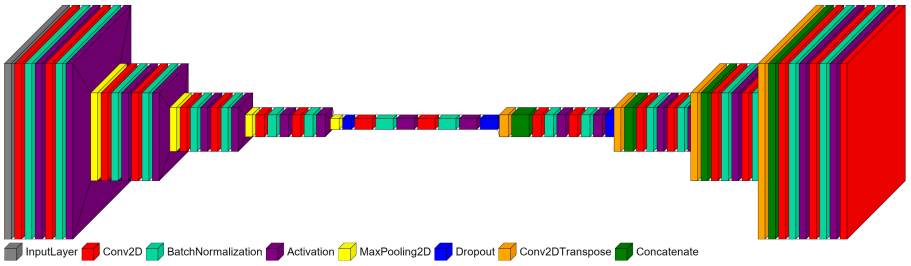


Figure 4.1: Architecture of the basic U-Net

4.1.2 Eff U-Net

The second architecture is inspired by Baheti B. et al. [2] which propose a novel approach for semantic segmentation that use EfficientNet as feature extractor in encoder and UNet as decoder.

The EfficientNet, as proposed in [13], consists of the compound coefficient which studied model scaling and adjusted the depth, width, and the resolution of the network for better performance. A new baseline architecture called EfficientNetB0 was designed initially and it is scaled up to generate family of EfficientNet by compound scaling method. Powered by this approach, there are eight variants of the EfficientNets, namely EfficientNetB0 to EfficientNetB7. As illustrated in figure 4.2 a good compromise between number of parameters (in order to avoid a too heavy training) and accuracy is EfficientNetB5 and it is used as encoder in the implemented model.

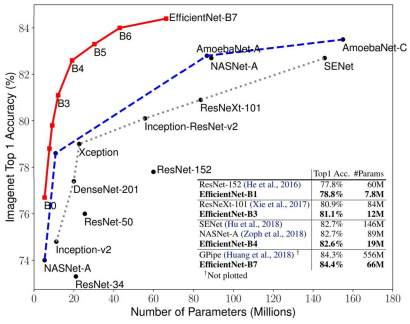


Figure 4.2: Eff-Nets comparison [13]

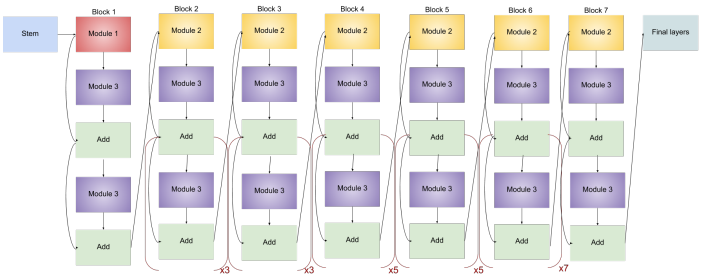


Figure 4.3: Architecture of EfficientNetB5 [1]

The structure of EfficientNetB5, illustrated in figure 4.3, is composed by 7 building blocks of several mobile inverted bottleneck convolutions (MBConv) with squeeze and excitation optimization. The final Eff U-Net model appear like in figure 4.4 where the decoder is similar to the original U-Net and as usual blocks from the contracting path are connected with the corresponding layers in the expansion path.

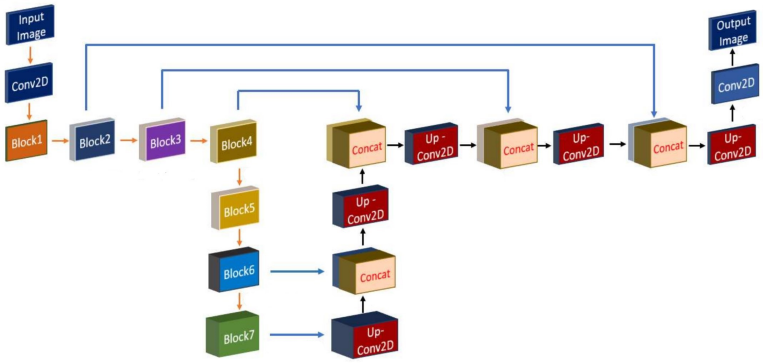


Figure 4.4: Architecture of Eff U-Net

4.2 Training

4.2.1 Hardware used

Training is a highly demanding task in terms of both computing and memory usage, so it is important to use properly the available resources. For that purpose, it is used a machine with those specifications:

- Intel® Core™ i7-10750H (6 core)
- RAM DDR4 16 GB
- NVIDIA® GeForce RTX™ 2060 6 GB
- Windows 10 Home 64

The training is done using *Keras* with *Tensorflow 2.4.1* as backend since it support GPU computation, which speeds up the training some orders of magnitude.

4.2.2 Experiments

During the training phase, several sessions have been scheduled in order to test the combination of different hyper-parameters. The common hyper-parameters fixed for each attempt are:

- batch size: 64
- epochs: 80
- optimizer: Adam with learning rate = 0.001
- class weights used for weighting the loss function
- metrics: IOUScore, Precision, Recall, FScore (provided by *segmentation_models* library [14])

The most promising models were the ones illustrated in table 4.1 where have been alternated the losses *Dice Loss (DL) + Binary Focal Loss (BFL)* and *Dice Loss (DL) + Categorical Focal Loss (CFL)* and also an attempt without augmentation has been done.

	Models description					
	Architecture	**Activation**	**Loss**	**Epochs**	**Trainable Params**	**Augmentation**
Model A	Basic U-Net	Sigmoid	BFL + DL	80	7.766.513	Yes
Model B	Basic U-Net	Sigmoid	BFL + DL	80	7.766.513	No
Model C	Basic U-Net	Softmax	CFL + DL	80	7.766.513	Yes
Model D	Eff U-Net	Sigmoid	BFL + DL	80	17.293.649	Yes

Table 4.1: Description of models

For the Eff U-Net training it has been applied the *transfer learning*, a popular method that allows to build accurate models in a timesaving way reusing models trained on a different but somehow similar problem. Indeed, instead of starting the learning process from scratch it relies on pre-trained weights based on ImageNet. In this way the convolutional base has been freezed on its original form and it uses pre-trained weights to get feature extraction.

As can be observed from the figure 4.5, the latter model seems to be the best and, thanks to transfer learning, the loss decreases much faster than other models in early epochs.

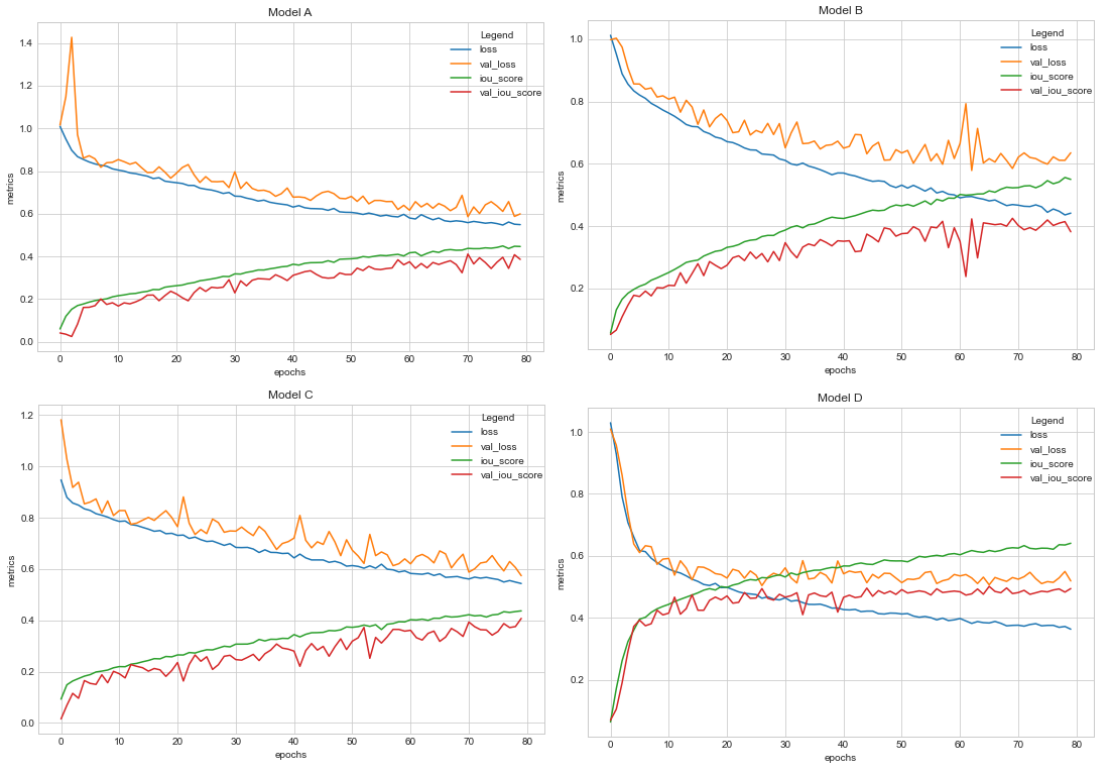


Figure 4.5: History of models during the training

5 Evaluation

All the predictive masks before the evaluation have been processed by a method that given in input the masks produced by the neural network, where it is associated a probability to each pixel, return a binarized masks where each pixel is 1 or 0 depending if the probability is over a certain threshold. The used threshold value, empirically selected, is 0.4 .

5.1 Metrics

In order to evaluate results produced by different models' predictions on the test set of images, it has been used three metrics that are required by the project and that are the most used at the state of art [10]

- *IoU*: Intersection over Union
- *Precision*
- *Recall*

5.1.1 IoU

The intersection over union 5.1, known also as Jaccard index, is the ratio between the area of overlap and the area of union of the ground truth and prediction, where the overlap can be calculated as the intersection between the two masks.

In the evaluation of the results of prediction, the IoU has been calculated for every different classes present in the image and the final image IoU is the mean of the IoU associated to every single class. The final result is the mean of every single image IoU that is called mIoU.

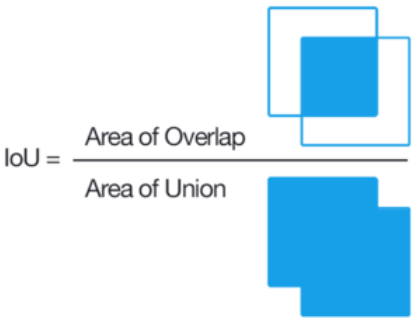


Figure 5.1: Intersection over Union

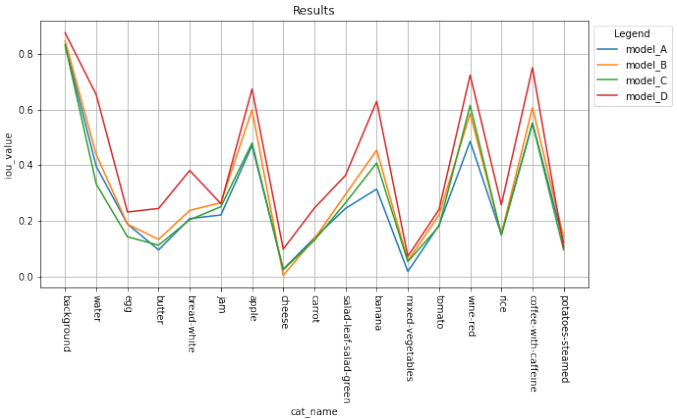


Figure 5.2: IoU values computed by for every category

5.1.2 Precision

The precision effectively describes the purity of our positive detections relative to the ground truth. Precision is calculated as the ratio between true positive and the sum between true positive and false positive results.

$$Precision = \frac{TP}{TP + FP}$$

In this project it has been used the Keras Precision class passing the ground truth mask and the predicted mask. Similarly to IoU results, also the precision results have been computed initially for each single image, as the mean of the precision of every class present in it, and then the mean of all the images precision has given the final precision value computed by the model on the predictions of the test set.

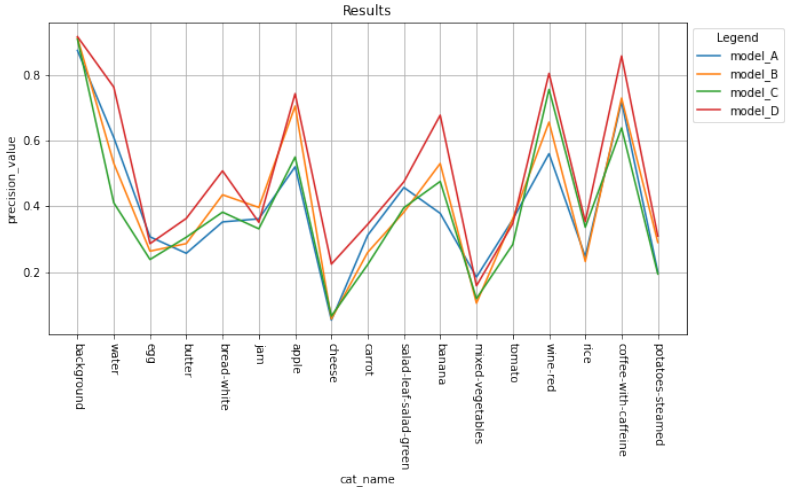


Figure 5.3: Precision values computed by every model for every category

5.1.3 Recall

The recall describes the completeness of our positive predictions relative to the ground truth. Recall is calculated as the ratio between true positive and the sum between true positive and false negative results.

$$Recall = \frac{TP}{TP + FN}$$

Also in this case it has been used the Keras Recall class passing the ground truth mask and the predicted mask and again, similarly to IoU and precision results, also the recall results have been computed initially for each single image, as the mean of the recall of every class present in it, and then the mean of all the images has given the final recall value of the predictions made by model on the test set.

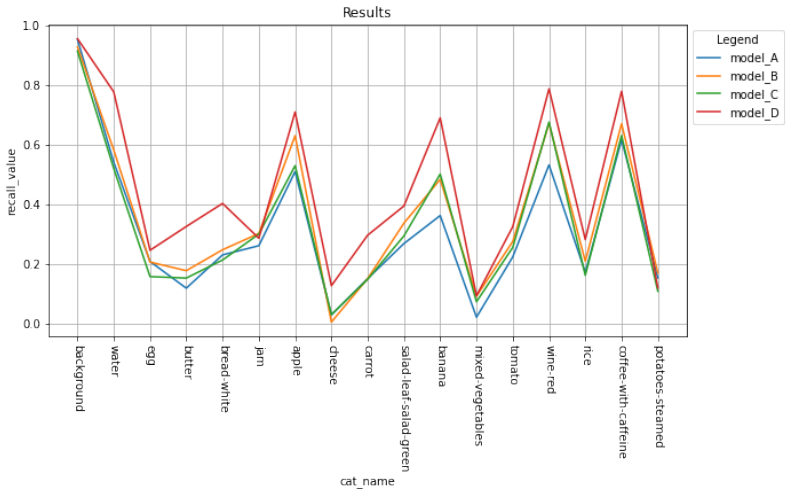


Figure 5.4: Recall values computed by every model for every category

5.2 Results

The results that can be seen in the table 5.1 are the final values of every metric computing the mean of every image values. It is important to say that these results take into account all the classes present in every single image (including the background class). Following some resources it can be seen that there are conflicting opinions on whether or not to take the background class into account in the calculation of the metrics values. The final decision was to take into account also the background class following the suggestions of an AICrowd forum discussion about the evaluation criteria [5].

	Test		
	IoU	**Precision**	**Recall**
Model A	0.488	0.591	0.559
Model B	0.525	0.618	0.591
Model C	0.455	0.549	0.526
Model D	0.624	0.703	0.688

Table 5.1: Test results of every model

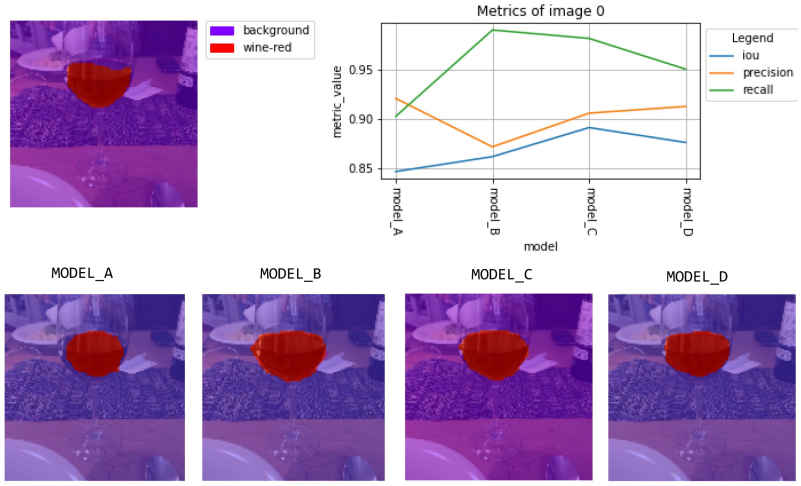


Figure 5.5: An example of a well done prediction

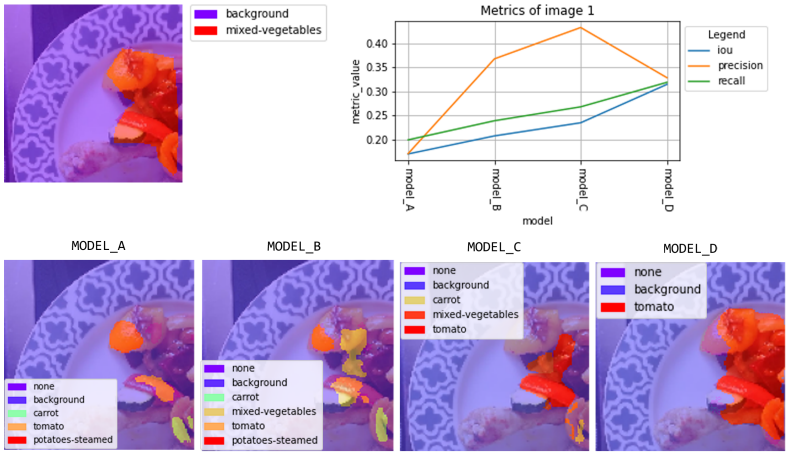


Figure 5.6: An example of a bad done prediction

6 Conclusions and future works

From the results it is possible to see that the Eff U-Net is the model that gives the best results with respect to the others considered. In order to improve the work that has been done in this project the first step could be extend the training not only for some categories but to the entire dataset. Of course this requires machines with more computational capabilities than the one that was used for this project. For this reason, it is more realistic to improve the project looking for other models from literature (for instance Mask RCNN, DeepLabv3, etc.) that could give better results for the project purposes. Moreover, it can be a good idea also to use different metrics in order to analyze in a more detailed way the results obtained by the various used models. The last improvement that can be done is on the ground truth annotations, since there are some images where they are not perfectly done.

Bibliography

[1] Vardan Agarwal. *Complete Architectural Details of all EfficientNet Models*. 2020. URL: https://towardsdatascience.com/complete-architectural-details-of-all-efficientnet-models-5fd5b736142.

[2] Bhakti Baheti, Shubham Innani, Suhas Gajre, and Sanjay Talbar. "Eff-UNet: A Novel Architecture for Semantic Segmentation in Unstructured Environment". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2020.

[3] *Calculate class weights*. URL: https://www.tensorflow.org/tutorials/structured_data/imbalanced_data#calculate_class_weights.

[4] AI Crowd. *Food Recognition Challenge*. URL: https://www.aicrowd.com/challenges/food-recognition-challenge.

[5] AI Crowd Forum. *Food Recognition Challenge Forum, Evaluation Criteria Discussion*. URL: https://discourse.aicrowd.com/t/evaluation-criteria/2668.

[6] Shruti Jadon. "A survey of loss functions for semantic segmentation". In: *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*. 2020, pp. 1–7. DOI: 10.1109/CIBCB48159.2020.9277638.

[7] Jeremy Jordan. *An overview of semantic image segmentation*. URL: https://www.jeremyjordan.me/semantic-segmentation/.

[8] Tsung-Yi Lin. *COCO API*. URL: https://github.com/cocodataset/cocoapi.

[9] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. "Focal Loss for Dense Object Detection". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2999–3007.

[10] Angelo Monteux. *Metrics for semantic segmentation*. URL: https://ilmonteux.github.io/2019/05/10/segmentation-metrics.html.

[11] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi. Cham: Springer International Publishing, 2015, pp. 234–241.

[12] Adrian Rosebrock. *Keras ImageDataGenerator and Data Augmentation*. URL: https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/.

[13] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[14] Pavel Yakubovskiy. *Segmentation Models*. URL: https://github.com/qubvel/segmentation_models.