ALMA MATER STUDIORUM
University of Bologna
School of Informatics - Science and Engigneering

Master in
Artifical Intelligence

Deep Learning project

FOOD RECOGNITION CHALLENGE

Students:
Omar Younis ID **1004765**
Mattia Bertè ID **0983469**

Accademic year 2020/2021

Contents

Introduction

"Food Recognition Challenge" is a competition organized by AI-crowd, in order to find the best technique to detect and classify food from images.
We can read on their site about the importance of this task:

"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. Food tracking can be of personal interest, and can often be of medical relevance as well. Medical studies have for some time been interested in the food intake of study participants but had to rely on food frequency questionnaires that are known to be imprecise.
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"

This is a typical example of image segmentation problem, in fact we are not only interested in determining the presence of a food but also in its position and we have to deal with the issue of having more type of food in the same photo.
Image segmentation can be seen as a particular type of classification task in which each pixel has to be classified.
We haven't taken part formally to the competition but we have studied this problem to understand which are the most common errors, how to solve them and to apply our theoretical knowledge to a practical problem.

These kind of problems require sophisticated hardware to be solved efficiently, in particular GPU optimized for convolutional operations, so we decided to work mainly on Google Colab to have at our disposal GPU pro-

vided by Google, usually more powerful than the ones present on common PC.
The idea was to run our code on Colab and to avoid problems related to the fact that we have to work on it remotely, we setup a *Google Drive* shared folder to have always last updated files. Google Colab have a strict usage limit so, to train many epochs, we use our Azure student subscription. However, both solutions have overall a limited amount of memory so we have studied a strategy to avoid memory overflow as we will explain better in section 1.4.

Chapter 1

Data set inspection and preprocessing

1.1 Data set inspection

The data set was provided by AI-crowd and it is composed of RGB images of different size with their correspondent annotations in the **COCO** format. COCO stand for **C**ommon **O**bject in **CO**ntext [1]. It contains images of different foods and the goal is to recognize and segment them. The test set of the competition is, obviously, private so we have splitten the validation set trying to preserve the proportions between classes.
The data set is divided as presented in Table 1.1.

	Number of images	**Number of annotations**
Training Set	24120	39328
Validation Set	649	1047
Test Set	620	1006

Table 1.1 - Data set division.

Since our results will be strictly related to the data set, it is always a good approach analyzing the data set before starting, in order to identify in advance some possible issue.
We opened the json file as *pandas DataFrames* to manipulate them and during the inspection, we have identified some possible source of problems.
First of all, the data set presents annotations related to 273 classes, however classes are very unbalanced: some of them are really specific and related to food that is very rare to find, such as the class **veggie burger** that counts just 36 elements, while others are very common, such as the class **water** that present 1836 examples.
To have an idea of the distribution of the examples between the classes we have just plotted the number of samples for each class (Figure 1.1). This is also useful to check that the validation set and the test set are correctly divided.

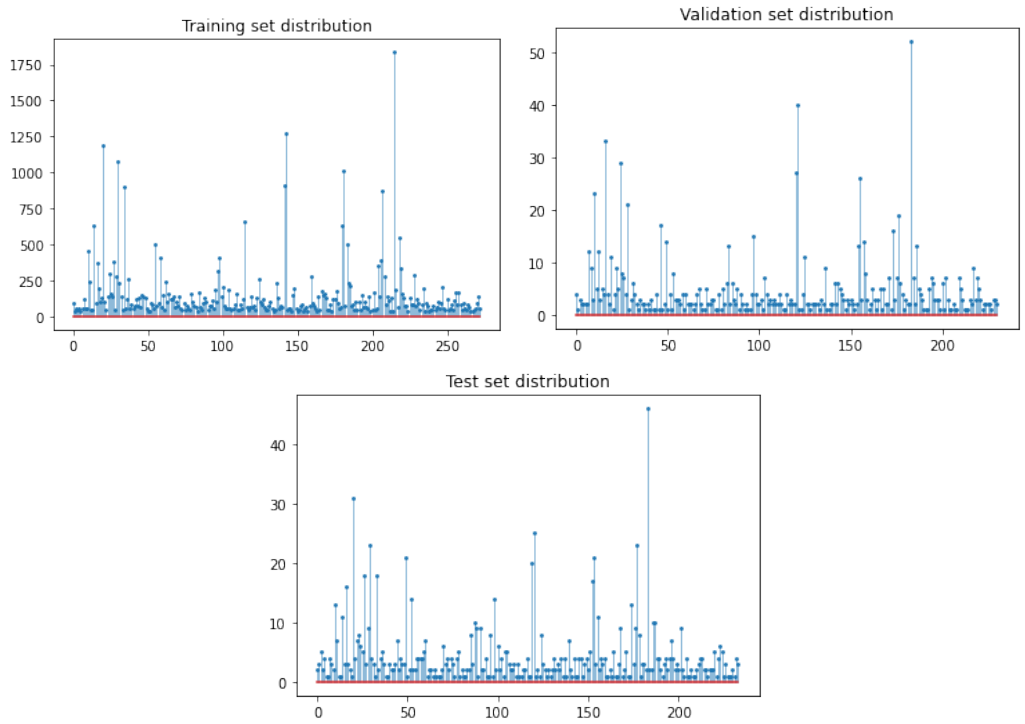

Figure 1.1 - *(Top-left)* Training set distribution. *(Top-right)* Validation set distribution. *(Bottom-center)* Test set distribution.

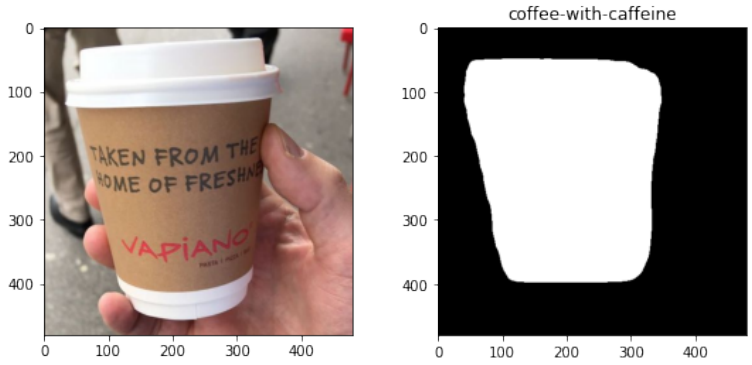

Figure 1.2 - An unconventional input image and its mask

Moreover we have subgroups of classes that are very similar to each other, such as *white-coffee-with-caffeine*, *espresso-with-caffeine*, *coffee-with-caffeine* and *ristretto-with-caffeine*.
Finally, the visual inspection of the data set revealed that another source of difficulties is the fact that some images are somehow unconventional as shown in Figure 1.2. In fact, generally, images of the class *coffee-with-caffeine* are cups with visible coffee and the segmentation is relative to the coffee and not to the cup.
Another example is in Figure 1.3, here we can see slices of apple that can be difficult to classify given that the majority of apple photo present an integer ones.

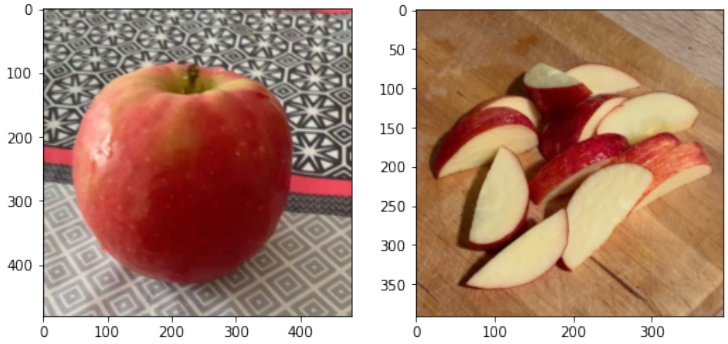

Figure 1.3 - *(Right)* Common apple image. *(Left)* Slices of apple which can be difficult to recognize.

1.2 Mask creation

During the data set inspection we have discovered another issue that could have created problems if it hasn't been noticed.
Some images present overlapping mask, that means some pixels belong to multiple classes, for example a category such as **cheese for raclette** can be found alone but also on a slice of bread, so these pixels will belong to both classes because they are classified as cheese but from a complessive point of view they are also above the bread, the same happened with bacon as can be seen in Figure 1.4

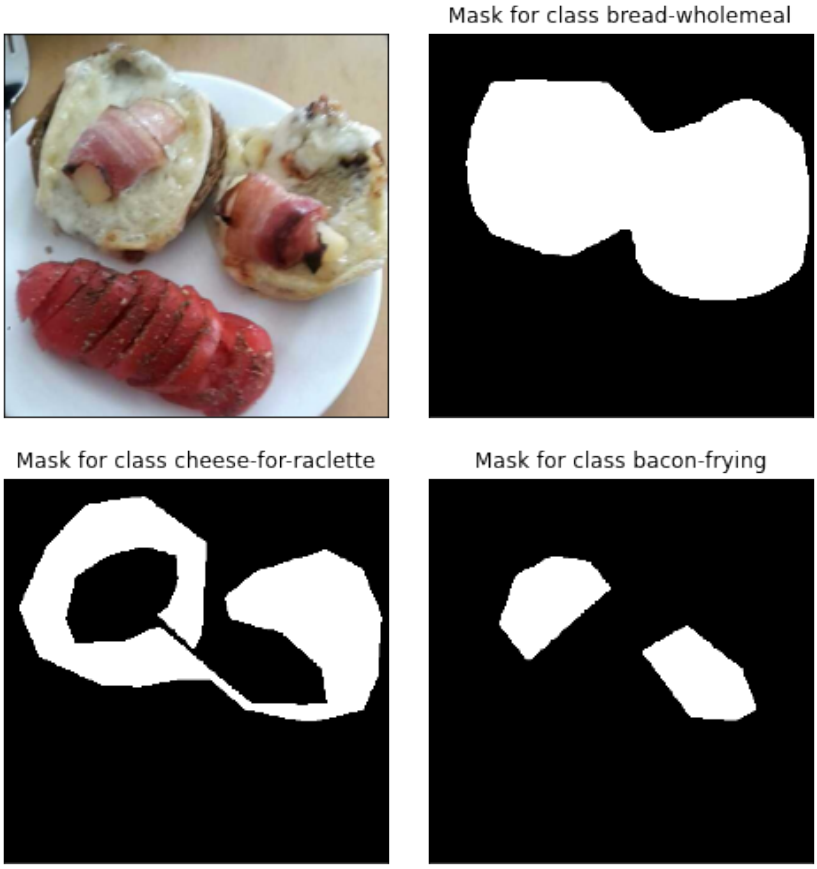

Figure 1.4 - Example of the original images and its overlapping mask.

To deal with this issue, the idea is to create a mask composed of a tensor of dimension $(H \times W \times 273)$, where H and W are height and width of the image and the number of channels equals to the number of categories.

$$mask(i,j,k) = \begin{cases} 1 & \text{if } p(i,j) \in \text{Class k} \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

Each channel k is a boolean matrix, since storing a single bit is cheaper than storing the values 1 and 0 in floating point notation, where the correspondent pixel $p(i,j)$ assumes value 1 if it belongs to the object of the class k, 0 otherwise. Each images will have 273 masks, but the majority of them will be composed of all zeros.
The maximum number of category present in the same images is 12 and in some images we have more distinct polygon for the same category.

1.3 Resizing

Another issue, encountered during the inspection, was that all images has a different size, so even if CNN architectures allows to work with them, all the images belonging to the same batch have to share the same dimensions. A resizing was necessary, so after some quick test with dimension $(512, 512, 3)$ and $(256, 256, 3)$ we decided for the second one, since some images are smaller than $(512, 512, 3)$ and enlarging could introduce some undesired artifacts; moreover smaller images allows us to use a larger batch size and lead to a faster training.
For what concern masks, we have computed them starting from the annotations related to original images and then we have resized them using **nearest neighbour interpolation** to avoid non binary pixel.

1.4 FoodSequence class

It is clearly impossible to load all the images and generate all the *segmentation masks* to fit our models. We have tried to use *ImageDataGenerator*

of Keras [2], but it is not suited for our purposes because it is designed for classification. In particular, we want to generate segmentation masks in a lazy mode (i.e. computing only a batch of masks just before passing them to training) and we want to adapt the segmentation masks after transforming training images for data augmentation. Given these considerations we decided upon write our own class *FoodSequence* that extends *Sequence* of Keras. The class Model can then work easily calling the method *__getitem__* with the batch's index to retrieve a batch. The batch size can be specified in the constructor. In subsection 1.4.1 we describe in details how this class work.

1.4.1 Loading images and generating segmentation masks

The class FoodSequence has an attribute *images* of type pandas DataFrame that contains all image identifiers with the respective file path. Another important attribute is *annotations*, a dict that contains for each images the map category-segmentation polygons. Using the attribute *images*, the batch images are loaded and then resized accordingly to the class attribute *img_size*. Then, for each image, the categories mask is computed using the dict *annotations*.
This class takes care of all the issues discovered during inspection and described in section 1.2 and 1.3

1.4.2 Perform data augmentation

In the constructor of the FoodSequence class is possible to specify a boolean parameter *data_augmentation* to activate/deactivate data augmentation [3]. We have not used one of the Keras built-in methods because they performs random data augmentation and we need to fulfill the same transformations on the categories masks except for contrast modifications and for the type of interpolation. So we code a simple method that performs these operations on the loaded batch:

- A horizontal flip (using *tensorflow.image.flip_left_right*) to both images

and categories masks at random with 50% of chance

- A vertical flip (using *tensorflow.image.flip_up_down*) to both images and categories masks at random with 50% of chance
- A contrast adjustment (using *tensorflow.image.adjust_contrast*) only to batch images using a random contrast factor between 0.6 to 1.4
- A rotation (using *tensorflow_addons.image.rotate*) to both images and categories masks using a random angle between $-30°$ and $+30°$
- A zoom (using the class *RandomZoom* of Keras) to both images and categories masks using a random scale between -0.3 and 0 (so only a zoom in). For categories masks we use a nearest neighbour interpolation.

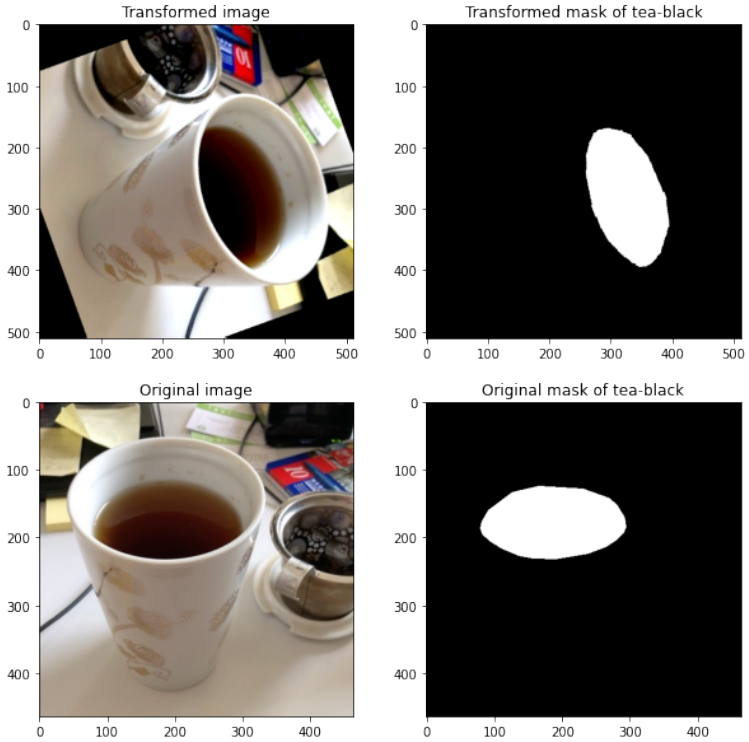

Figure 1.5 - Example of our data augmentation. *(Top)* The transformed image and relative categories mask. Notice also the change of contrast. *(Down)* The original image and his categories mask.

1.5 Loss function

Remembering that our goal is to maximize the intersection over union (IoU), we have tried to design a differentiable loss function that resemble it:

$$L_C(y, \hat{y}) = -2\frac{y \odot \hat{y}}{y + \hat{y}} \tag{1.2}$$

where $\odot$ indicates the element-wise multiplication. We have noticed that the training speed was very slow. We have then discovered that our loss is similar to the *dice loss*:

$$L_D(y, \hat{y}) = 1 - 2\frac{y \odot \hat{y} + 1}{y + \hat{y} + 1} \tag{1.3}$$

that it is known to be highly non-convex, as pointed out by S. Jadon [4].
In all of the architectures that we have used, the output is a 4d-tensor with shape $[M \times W \times H \times 273]$ with M equal to the batch size, W and H the input image size. The channels are 273 as the number of categories. We have used as output function the *sigmoid*, so the output represent the belief of the network to assign the corresponding pixel to the category associated with the channel. We have used the sigmoid because one pixel can belongs to multiple categories as pointed out in 1.1. Keeping this in mind, we have decided upon the *binary cross-entropy*.
At this point, the outputs were like in figure 1.6.
We have estimated from the training set that the unconditional probability of a pixel to belong to a category is roughly $\frac{1}{700}$. Therefore, the network is more confident to not assign a pixel to a category instead of the other way around since the output has 273 with the majority of them composed of all zeros matrix. To solve this problem we have chosen to use a *weighted binary cross-entropy*:

$$L_W(y, \hat{y}) = -\beta \cdot y \odot log(\hat{y} + \epsilon) + (1 - y) \odot log(1 - \hat{y} + \epsilon) \tag{1.4}$$

where ϵ is a small number. We have added ϵ to avoid overflow after computing the logarithm. Empirically, we have discovered that $\beta = 100$ seems to be a

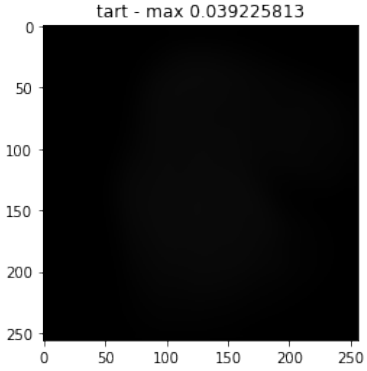

Figure 1.6 - Example of output image using binary cross entropy. The images seems all black because the outputs values are very small. This output was a prediction made after 4 epochs of training with batch size set to 8.

good choice.

Another issue we have encountered trying to minimize the loss function is related to the classes distribution, since they are very unbalanced the neural net will be more prone to learn the more frequent classes with the respect to the other. See Figure 1.1 for a clear idea of how the distribution look like. Thus, we have introduced weights associated to each class inversely proportional to their frequency to give more importance to rare class. In particular, we have used the pixel frequency, i.e. we have compute the sum of areas for each category.

Chapter 2

Images segmentation architectures

After data preprocessing, we have studied different architectures for this problem and we have found two interesting neural networks, the **SegNet** [5] and the **U-Net** [6].

2.1 SegNet

SegNet follows an *encoder-decoder* architecture. The encoder part is identical to VGG16 network [7]. In particular, the architecture uses many convolutions to detect features, followed by batch normalization [8] and $ReLU$ activation function layer. As pooling layer, it uses max pooling with size (2, 2), stride 2 and padding *same*. The decoder part of the architecture is symmetric to the encoder. Notice that we have used sigmoid instead of softmax as last activation function layer, as explanied in 1.5.

2.2 U-Net

This architecture has been proposed in 2015 but it still widely used. Its name derived from the shape with which it is presented, it is composed of two main path, the downsampling path during which the images are decreased in size,

through successive convolutional layers, activation layers (ReLU) and pooling one (Max Pooling) but in which the number of channels is doubled four times. Then we have the upsampling path, where the images are increased in size up to their original ones. Along this path there each layer doesn't process just the output of the previous layer but the concatenation of this one with the output of the correspondent layer in the downsampling path. The general architecture is presented in Figure 2.2.
BatchNormalization layers has been added to improve training.

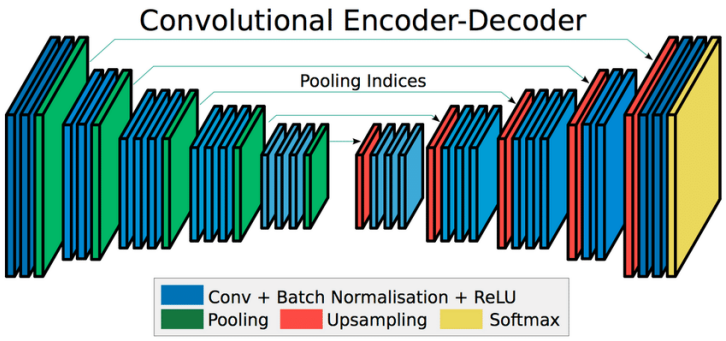

Figure 2.1 - SegNet architecture.

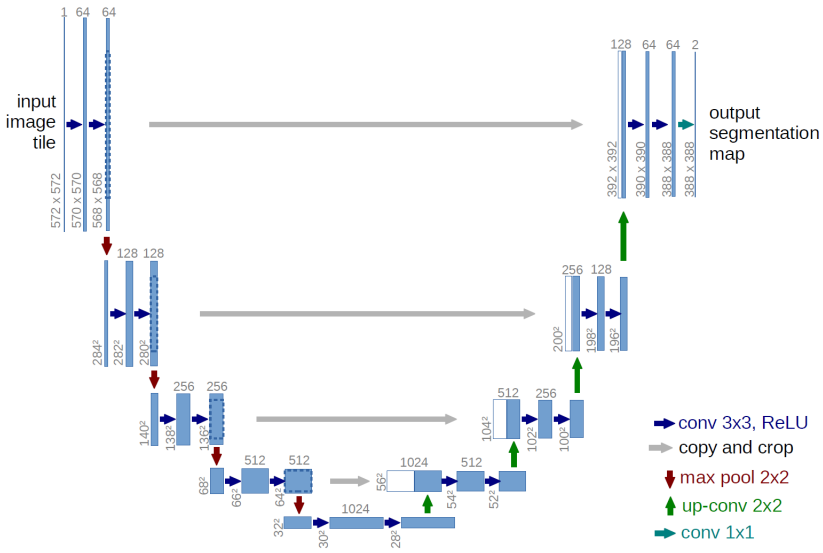

Figure 2.2 - U-net architecture.

Chapter 3

Results

3.1 Training

Once the two architectures has been defined, we started the training process.

3.1.1 Training without data augmentation

We first tried to train SegNet without using data augmentation. This lead soon, just after 5-6 epochs, to overfitting. This seems reasonable because our training set is not very large with respect to our network parameters. In principle, our network can interpolate the training set, as proved in [9]. Notice that SegNet architecture has 29.4 millions of parameters.
As can been seen in Figure 3.1 after 5-6 epochs, the validation set error start to increase while the training set error continue to decrease, that is a clear evidence that our net was in overfitting, that means it was losing its generalization ability.
Given the previous results, we avoided to try the same process with U-Net architecture.

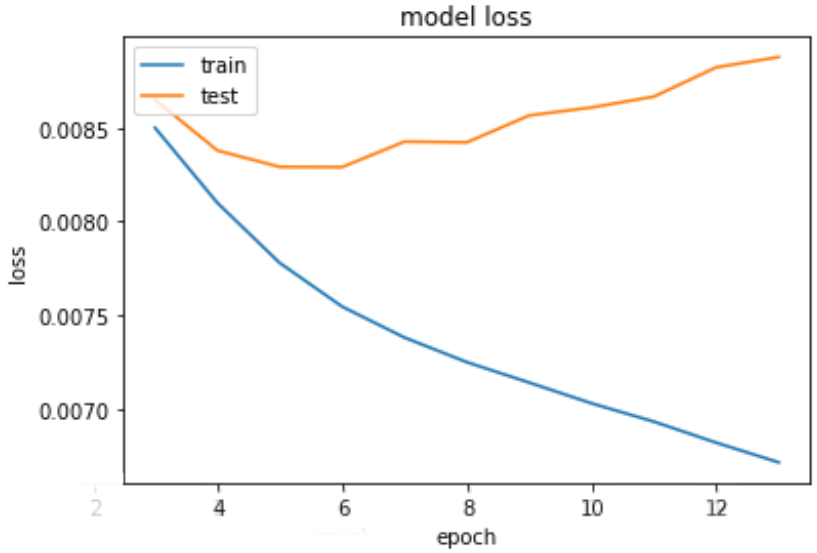

Figure 3.1 - A plot of the training loss (blue) and the validation loss (orange) of SegNet architecture

.

3.1.2 Training with data augmentation

As described in section 1.4.2, we introduced data augmentation to decrease the probability of overfitting and we reached our purpose, infact we have been able to train the two nets for more than 100 epochs without this problem. In Figure 3.2 and Figure 3.3 we have plotted training loss and validation loss during training. Notice that the validation loss is lesser than training loss. This is mainly because during training we have applied data augmentation; moreover during validation inference batch normalization is turned off.
After this great number of epoch we decided to stop the training because the loss was decreasing very slow, even if we weren't in overfitting yet. Moreover, our Azure credit sufficed for just about 50 epochs so we were forced to continue on Google Colab which has GPU usage limit and training with a simple CPU is too time consuming.

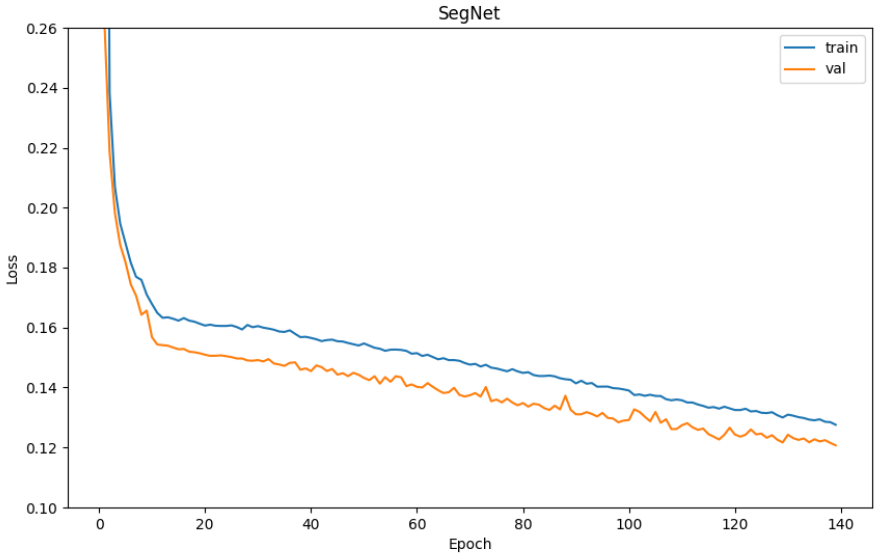

Figure 3.2 - Plot of training loss (blue) and validation loss (orange) during the training of the SegNet

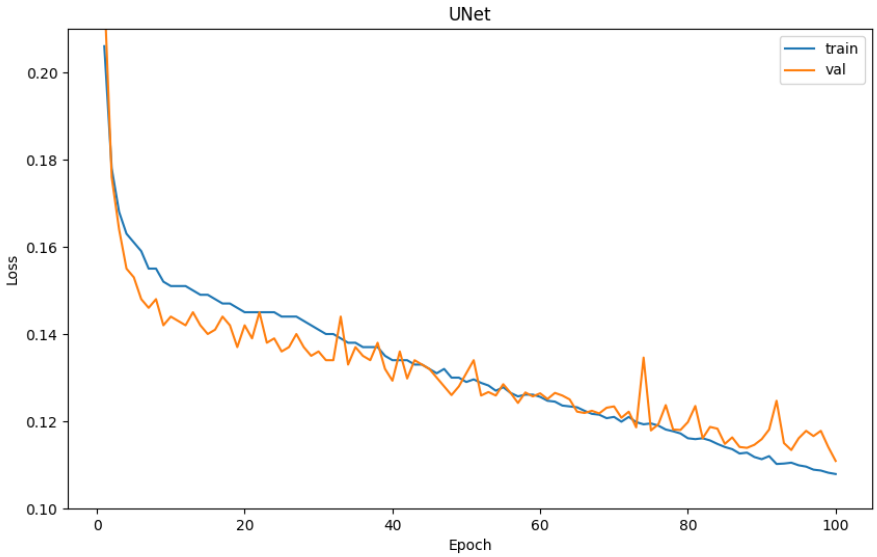

Figure 3.3 - Plot of training loss (blue) and validation loss (orange) during the training of the U-Net

3.2 Tuning

3.2.1 Thresold tuning

After the training process, we had to find a threshold in order to make the output of our nets a binary mask. The output is the result of the application of the sigmoid function to the output of the last layer. The sigmoid squashes all the values between 0 and 1 but they are still real values and we need a boolean mask.
The idea is to apply a threshold T such that:

$$pixel(i,j) = \begin{cases} 1 & \text{if } pixel(i,j) \geq T \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

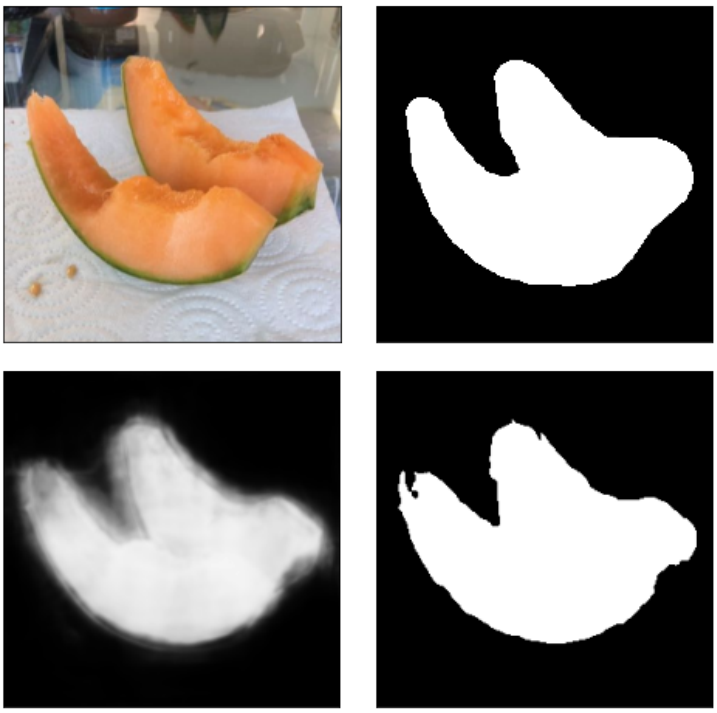

Figure 3.4 - *(Top-left)* The original image in input. *(Top-right)* The real mask.*(Bottom-lef)* The output of the net. *(Bottom-right) The binarized output.*

To speed up the tuning process, we have used *coarse-to-fine* technique [10]. At the beginning we have plotted IoU trying different thresholds between 0.4 and 0.9 with step of 0.1. After discovering that the best value for the threshold is between 0.5 and 0.7, we have computed IoUs in that range with a step of 0.02. We found 0.59 ± 0.01 as the best value. In Figure 3.5 you can find the results of applying this technique for the SegNet.

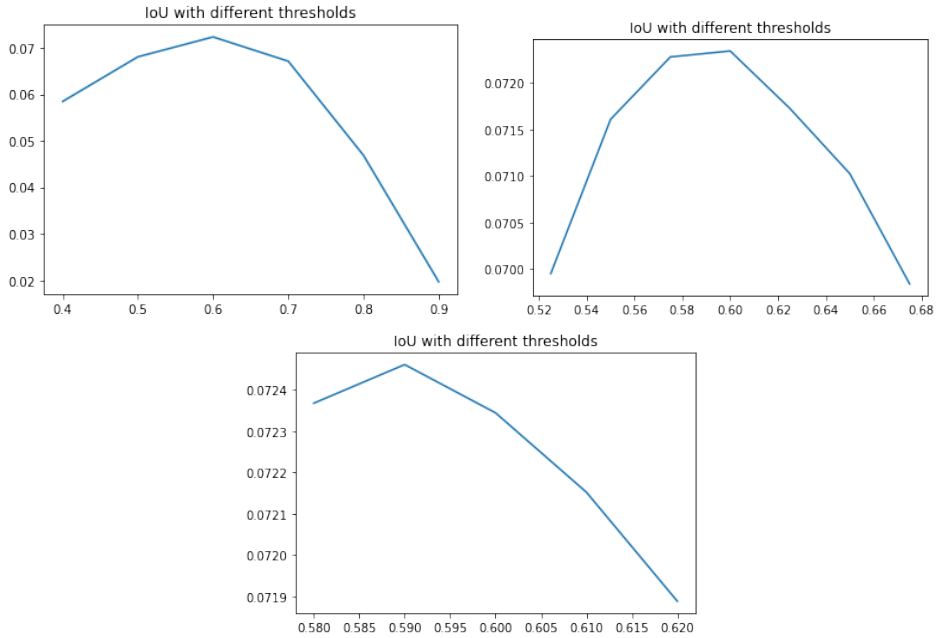

Figure 3.5 - IoU plot with different thresholds for SegNet architecture. *(Top-left)* The threshold is between 0.4 and 0.9 with step of 0.1. *(Top-right)* The threshold is between 0.50 and 0.70 with a step of 0.02. *(Bottom-center)* The threshold is between 0.580 and 0.620 with a step of 0.01

The same procedure has been applied to U-Net, as can be seen in Figure 3.6. In this case the threshold that optimizes the overall IoU results to be equal to 0.635.

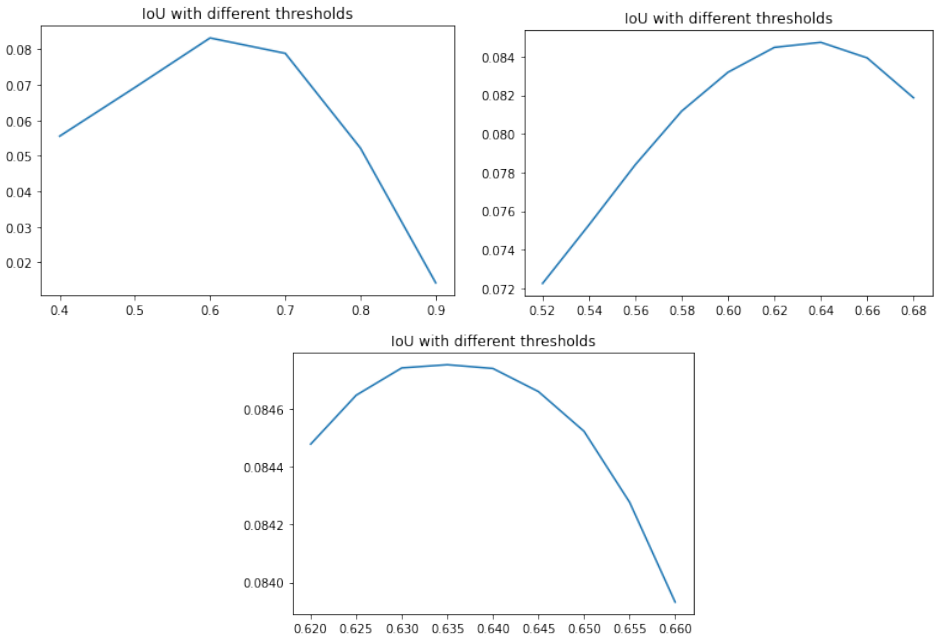

Figure 3.6 - IoU plot with different thresholds for U-Net architecture. *(Top-left)* The threshold is between 0.4 and 0.9 with step of 0.1. *(Top-right)* The threshold is between 0.52 and 0.68 with a step of 0.02 *(Bottom-center)* The threshold is between 0.620 and 0.660 with a step of 0.005

At this point, after choosing the right threshold, we have computed the overall Intersection over Union on the test set. We report the results in Table 3.1.

	IoU
SegNet	0.0748
U-Net	0.0901

Table 3.1 - Overall IoUs for the two architectures after tuning the threshold.

3.3 Tuning of two different thresholds

Since the results with a simple thresholding doesn't appear very satisfactory, due to the fact that for each image we had a great number of false positive masks, we adopted a different approach and we introduced a second threshold. This one has the purpose to keep only the channels whose **max** is higher than the threshold. Doing so, a large number of channels is discarded.
For tuning this value we decided to maximize the precision:

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

infact the problem with our nets is that they tend to detect objects that aren't present, i.e. the *false positive* (fp). So maximizing the precision we tend to minimize the number of false positive. We have tuned two thresholds for the SegNet. In Figure 3.7 we have plotted the precision values during tuning of the threshold for categories discrimination, the ones which provides the best performance is 0.77. Again, we have used coarse-to-fine method.

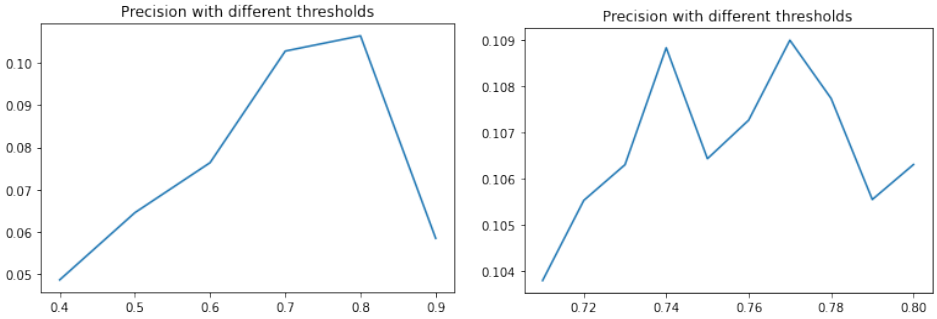

Figure 3.7 - Precision plot with different thresholds for channel selection of SegNet. *(Left)* The threshold is between 0.4 and 0.9 with step of 0.1. *(Right)* The threshold is between 0.71 and 0.80 with a step of 0.01.

After that, we tuned the second threshold for pixel discrimination and we choose for this value 0.4 as can been seen in Figure 3.8.

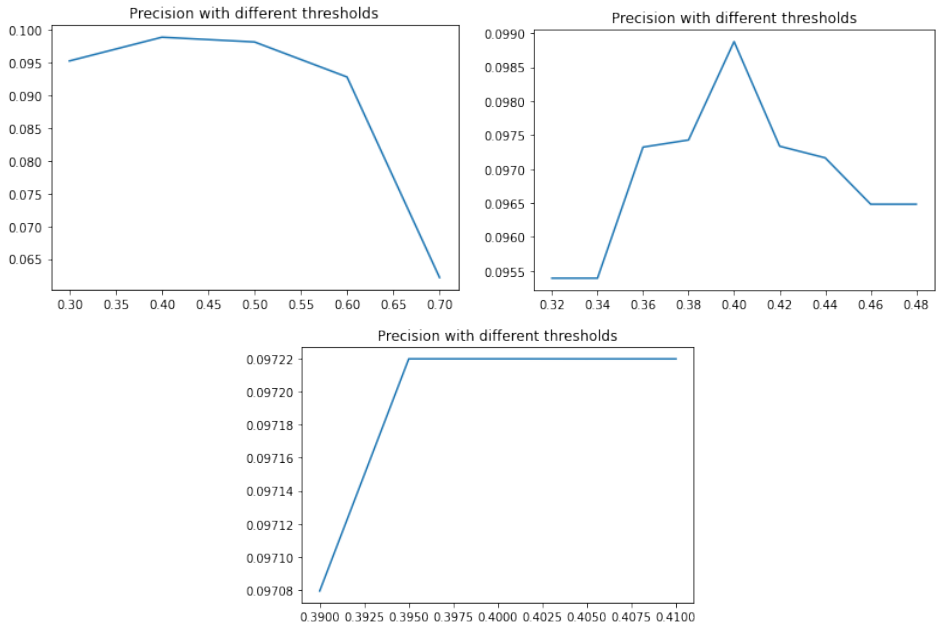

Figure 3.8 - Precision plot with different thresholds for pixels selection of SegNet. *(Top-left)* The threshold is between 0.3 and 0.7 with step of 0.1. *(Top-right)* The threshold is between 0.32 and 0.48 with a step of 0.02. *(Bottom-center)* The threshold is between 0.3900 and 0.4100 with a step of 0.005

Using the same technique for U-Net architecture, the optimal threshold for category discrimination is 0.790 as in Figure 3.9.
While the threshold for pixels selections is 0.41 Figure 3.10.
Summing up:

	Threshold for channels	**Threshold for pixels**
SegNet	0.770	0.400
U-Net	0.790	0.410

Table 3.2 - Threshold values for the two architectures.

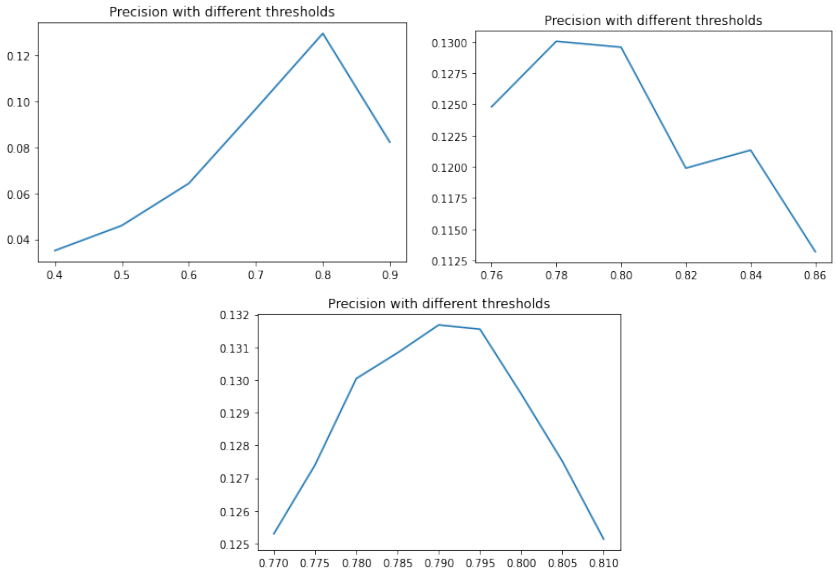

Figure 3.9 - Precision plot with different thresholds for channels selection of UNet. *(Top-left)* The threshold is between 0.2 and 0.9 with step of 0.1. *(Top-right)* The threshold is between 0.32 and 0.48 with a step of 0.02 *(Bottom-center)* The threshold is between 0.405 and 0.435 with a step of 0.005

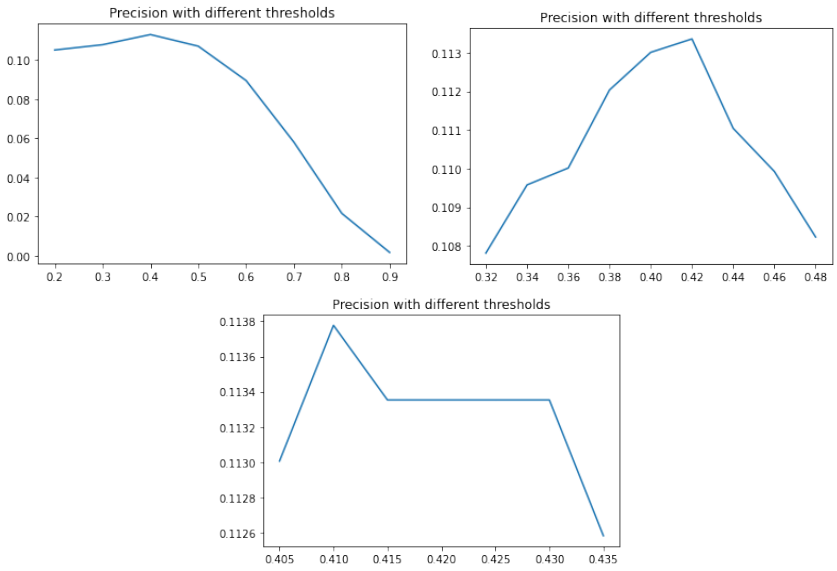

Figure 3.10 - Precision plot with different thresholds for pixels selection of UNet. *(Top-left)* The threshold is between 0.4 and 0.9 with step of 0.1. *(Top-right)* The threshold is between 0.76 and 0.86 with a step of 0.02 *(Bottom-center)* The threshold is between 0.770 and 0.810 with a step of 0.005

3.4 Metrics

Once the different thresholds have been tuned we computed *True Positive* (*TP*), *True Negative* (*TN*), *False Positve* (*FP*) and *False Negative* (*FN*) and from these all the metrics presented in Table 3.3

	SegNet	**UNet**
Accuracy	98.33%	98.33%
Sensitivity	20.68%	24.39%
Specificity	98.77%	98.74%
Precision	8.83%	9.84%
NPV	99.54%	99.57%
F1 Score	12.60%	14.02%

Table 3.3 - Table of metrics for both architectures.

Looking at the metrics, the U-Net seems to works slightly better. In particular it is more sensible to detect positives at the cost of having a tiny worse specificity.

3.5 Boosting

Given that the results weren't too good, we tried to improve them using the boosting [11], that means combine the two classifiers.
From a theoretical point of view, in order to obtain the best results from boosting, the classifiers have to be independent. If it was the case, we could have expect an accuracy improvement like this:

$$Acc_{Ens} = 1 - (1 - Acc_{SegNet})(1 - Acc_{UNet}) = 99.97\%$$

Obviously, this is a theoretical upper bound since our networks are far from being independent. Nevertheless, it results in better performances as shown in Table 3.4.

	Boosting
Accuracy	99.04%
Sensitivity	18.20%
Specificity	99.51%
Precision	17.87%
NPV	99.52%
F1 Score	18.03%

Table 3.4 - Table of metrics for boosting method.

The overall metrics are evidently better, but it is worthy to notice that the sensitivity is worse. This is because the ensemble net have a higher false negative rate, i.e. tend to be more conservative in the response.

Chapter 4

Conclusion

In the end, we have to admit that our results aren't too satisfactory with the respect to the neural network that have taken part to the real competition. Obviously, our nets have problem to handle this great number of classes, many of which would be difficult to distinguish even for an human eye as said in section 1.1, for examples coffee with caffeine and coffee without caffeine. They have also difficulties when the number of objects inside the image is greater than one as for example with mixed salad. Probably, some percentage point could have been gained training the nets up to the overfitting. We discuss, here, some other techniques that can be applied in order to improve the performances.

First of all, it is common in challenges to use *test-time augmentation*, i.e. apply data transformation to the test set, predict every transformed data and merge the results. For more detail about that technique, see *When and why test-time augmentation works*[12]. We avoid to try that because it highly increase the inference time and, because of this, it is not commonly used outside competitions.

We discovered a *Generalized Intersection over Union* that can be used as loss to improve IoU metric [13]. This loss seems to do not have problems that we have discussed in Section 1.5. It would be a good attempt to try this loss and see if it really improve performances in our task.

Finally, we tested the most widespread architectures for image segmenta-

tion and tried to do our best, but nowadays there are more advanced neural networks that represent the state of the art such *DeepLabV3* [14] and *Detectron2* [15] which is a pretrained network, thus, we avoided to try these networks because, for didactic reasons we want to try to train our network from scratch. For similar consideration, we avoided to try transfer learning [16].

Afterall, we are glad with our project because, despite the metrics, we have successfully overcome many obstacles (sometimes with a lot of effort) and we learned a lot of practical stuff that it would not have been possible by studying theory alone.

Bibliography

[1] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollar, *Microsoft coco: Common objects in context*, 2015. arXiv: 1405.0312v3 [cs.CV].

[2] *Keras*, 2015. [Online]. Available: https://keras.io.

[3] L. Perez and J. Wang, *The effectiveness of data augmentation in image classification using deep learning*, 2017. arXiv: 1712.04621 [cs.CV].

[4] S. Jadon, "A survey of loss functions for semantic segmentation," *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, 2020. DOI: 10.1109/cibcb48159.2020.9277638. [Online]. Available: http://dx.doi.org/10.1109/CIBCB48159.2020.9277638.

[5] V. Badrinarayanan, A. Kendall, and R. Cipolla, *Segnet: A deep convolutional encoder-decoder architecture for image segmentation*, 2016. arXiv: 1511.00561 [cs.CV].

[6] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, May 2015.

[7] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[8] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG].

[9] J. Berner, P. Grohs, G. Kutyniok, and P. Petersen, *The modern mathematics of deep learning*, 2021. arXiv: 2105.04026 [cs.LG].

[10] A. Ng, *Course 2 week 3 lecture 01 - deeplearning.ai*, 2017. [Online]. Available: https://youtu.be/AXDByU3D1hA.

[11] D. Opitz and R. Maclin, "Popular ensemble methods: An empirical study," *Journal of Artificial Intelligence Research 11*, 1999.

[12] D. Shanmugam, D. Blalock, G. Balakrishnan, and J. Guttag, *When and why test-time augmentation works*, 2020. arXiv: 2011.11156 [cs.CV].

[13] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, *Generalized intersection over union: A metric and a loss for bounding box regression*, 2019. arXiv: 1902.09630 [cs.CV].

[14] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, *Rethinking atrous convolution for semantic image segmentation*, 2017. arXiv: 1706.05587 [cs.CV].

[15] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, *Detectron2*, https://github.com/facebookresearch/detectron2, 2019.

[16] S. Bozinovski and A. Fulgosi, *The influence of pattern similarity and transfer learning upon the training of a base perceptron b2.* 1976.