Food Recognition

Report of the project about image segmentation of food images

---

Riccardo Fava - riccardo.fava6@studio.unibo.it
Luca Bompani - luca.bompani4@studio.unibo.it
Ganesh Pavan Kartikeya Bharadwaj Kolluri - ganeshpavan.kolluri@studio.unibo.it

---

Github Repository: https://github.com/BeleRicks11/Food_Recognition

Abstract

The Food Recognition challenge is a very interesting topic that consists in training a Deep Learning model able to detect the food items in an image and classify them in different categories.
The kind of project falls into the category of the image segmentation tasks, that are based on the process of partitioning a digital image into multiple segments.

We tried to address this problem by trying different deep learning neural networks in order to experiment some specific techniques and then make a comparison between them.
We took inspiration from model architectures that reached very good results in the field of image segmentation and we tried to adapt them to our purpose.
The neural networks that we have implemented are U-Net, LinkNet and Mask-RCNN.

In this report we present our work based on the development of such models and the result that we have archived.

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.
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.
Food recognition remains however a difficult problem for a variety of reasons. Deep learning neural networks for this task have millions of parameters and for the training they require very big datasets and a lot of computational power. Another problem is the creation of the dataset that takes a lot of time due to the huge number of images to label.
Finally there are an enormous number of types of food that exist and some categories are very difficult to recognize because they are very similar.
For all the reasons mentioned above, food recognition is a difficult, but important task.
Algorithms who could tackle this problem would be extremely useful for everyone.
The goal of this challenge is to train models which can look at images of food and detect the individual food items present in them.

Here there is an example of an image segmented that represents our goal well.

2. Data

2.1 Dataset exploration

For our project we used the dataset made available by the AIcrowd community.
The dataset is composed in the following way:

- Training Set of 24120 RGB food images, along with their corresponding 39328 annotations in MS-COCO format

- Validation Set of 1269 RGB food images, along with their corresponding 2053 annotations in MS-COCO format

2.1.1 Structure of MS Coco Dataset

Coco is a large scale object detection, segmentation, and captioning dataset. Coco datasets annotations have Json format.

```
{
 "info": {…},
 "licenses": […],
 "images": […],
 "categories": […],
 "annotations": […]
}
```

- The Info section consists of metadata about the dataset.
- The License section consists of creative common licenses which associate to each image a licence.
- The Images section consists of image metadata, where there are information about license, name of the file, url, height, width, date, unique id.

- The Categories section consists of classes representing the objects that are present on the images as labeled data. Categories are grouped as supercategories.

```
[
  {"supercategory": "person", "id": 1, "name": "person"},
  {"supercategory": "vehicle", "id": 2, "name": "bicycle"},
  {"supercategory": "vehicle", "id": 3, "name": "car"},
  …
  {"supercategory": "indoor", "id": 90, "name": "toothbrush"}
]
```

- The annotations section consists of information about the segmentation of the items. Here, are stored the coordinates of the pixels that are part of the segmentation mask, the id of the corresponding image and the class of the item.


```
{
 "segmentation":
 [[
  239.97,
  260.24,
  222.04,
  …
 ]],
 "image_id": 558840,
 "bbox":
 [
  199.84,
  200.46,
  77.71,
  70.88
 ],
 "category_id": 58,
 "id": 156
}
```

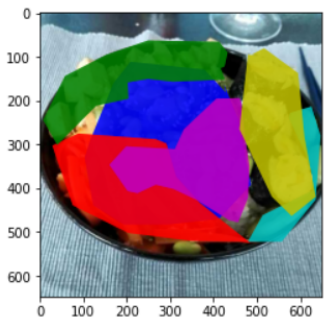

2.2 Preprocessing

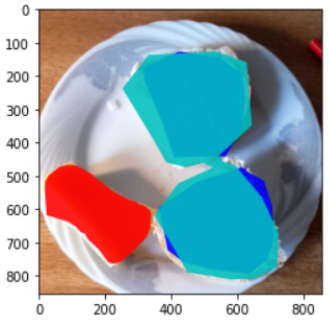
For manipulating the MS-COCO Dataset we used the official COCO python APIs called pycocotools version 2.0.
Due to the huge number of classes (273) that are part of this dataset, we decided to filter the images choosing the 16 most popular categories, in order to have a better management of the data and to avoid problems of memory.
Finally, the background class must be added to these categories.

Category Name	**ID**	**Train set images**	**Test set images**
Background	0	//	//
Water	1	1835	98
Bread-white	2	1273	65
Salad-leaf-salad-green	3	1189	64
Tomato	4	1069	52

Butter	5	1008	47
Bread-wholemeal	6	901	47
Carrot	7	893	39
Rice	8	659	29
Egg	9	626	30
Mixed-vegetables	10	624	34
Wine-red	12	545	23
Apple	13	504	38
Jam	14	502	25
Potatoes-steamed	15	450	25
Banana	16	412	28
Cheese	17	404	22

Once having filtered the images the training set contains 10454 images and the test set contains 539 images.
Before entering the main part of the project we have implemented a function to display the annotation of some images in order to have a better idea of how they are composed.
As we can see, an important property of the images in the dataset is that a lot of images have some food items that are overlapped. This is a particular property which we had to take into account.





2.2.1 Data Generators

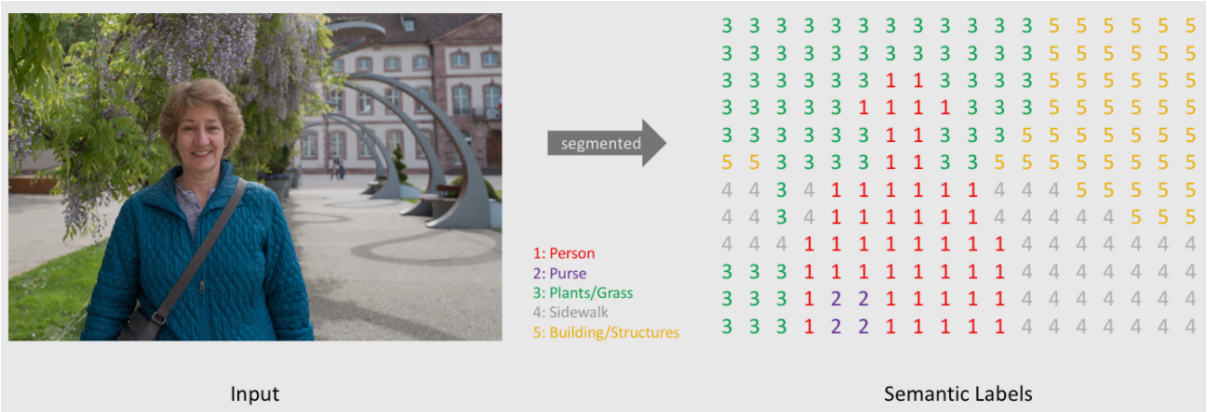
We have a huge number of images in our dataset and a problem that we encountered is loading the data for the preprocessing and training phase. Loading the entire dataset caused problems with the memory, so we decided to implement data generators.

Data generators load mini-batches of data and feed our model during training. They can generate the model input dynamically thus forming a pipeline from the storage to the RAM to load the data as and when it is required. Another advantage of this pipeline is, one can easily apply preprocessing routines on these mini-batches of data as they are prepared to feed the model.
We implemented our data generators importing ImageDataGenerator from tensorflow.keras.preprocessing.image .
So simply defining the batch size the data generators load a batch of images and prepare them for a future use. The next() function is invoked to retrieve a batch of data from the dataset.

2.2.2 Shape of the tensors

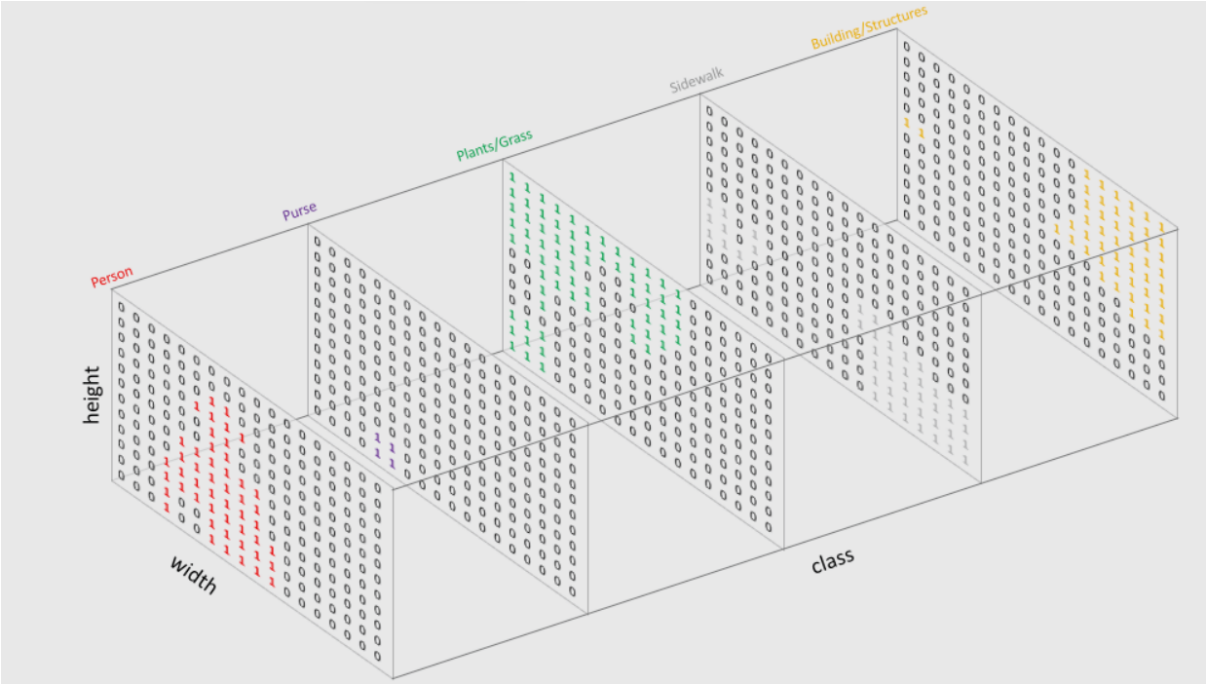
For the creation of our tensors to be given as input to the model we tried two different approaches. First of all we tried taking in input a RGB color image (height×width×3) and output a segmentation map where each pixel contains a class label represented as an integer (height×width×1).

The following image represents the approach defined before.



After several attempts, however, we decided to pass to a more accurate approach.
Taking in input a RGB color image (height×width×3) we created our target by one-hot encoding the class labels and essentially creating an output channel for each of the possible classes. This immediately saw that this approach gave us better results.

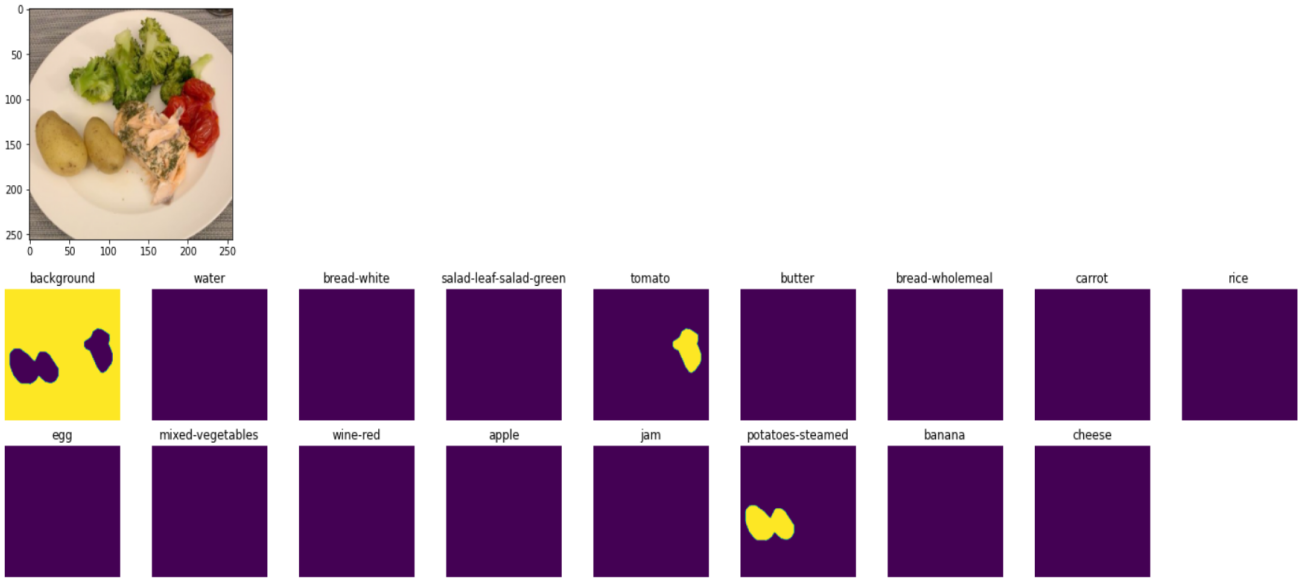
The following image represents the approach defined before.



So the output of our data generators is a 4D tensor composed by the batch_size, the image height, the image width and, for X (input image) the 3 channels of RGB, and for Y (true mask) the 16 classes + background class.

- *X.shape = [batch_size, 256, 256, 3]*
- *Y.shape = [batch_size, 256, 256, 17]*

The following image shows how class binary masks are composed.

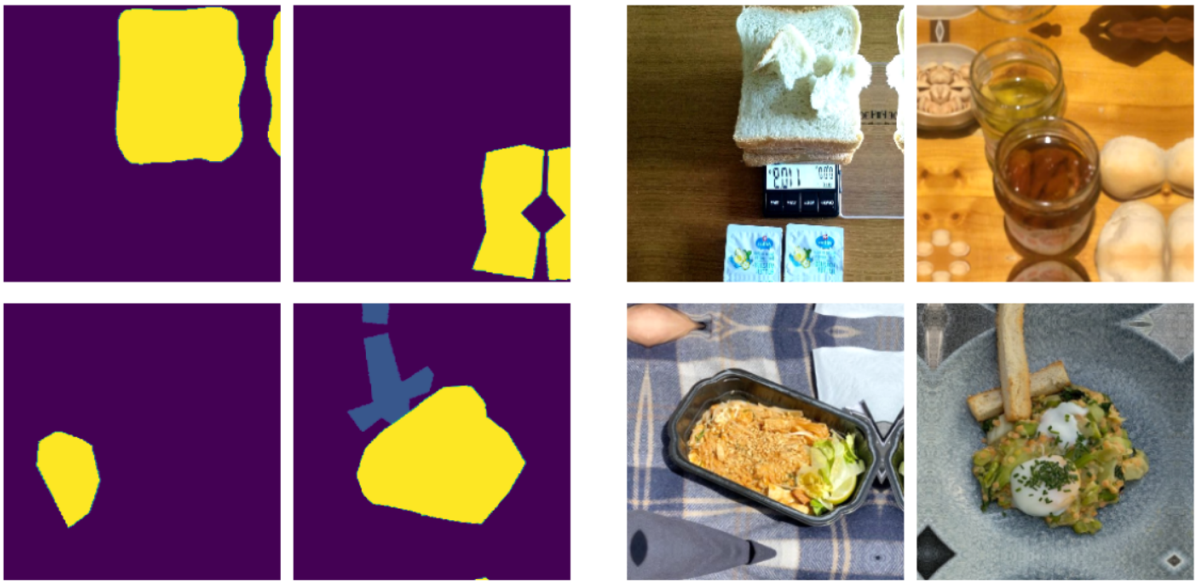


2.2.3 Data Augmentation

Data Augmentation is a technique where we can increase the images in a dataset by using some geometric transformations like cropping, padding, horizontal flipping and rotations, that will enhance the model performance. Neural networks need more data to better generalize the predictions and using the data augmentation technique can help to increase the dimensions of the dataset.
We implemented the data generators with augmentation, taking in input the images from the standard data generators and applying some transformations to them.
These are parameters that we used to apply these transformations are the following:

- horizontal shifting of 1%
- vertical shifting of 1%
- brightness in range = (0.8, 1.2)
- shear in range = 1%
- zoom in range = (1, 1.25)
- Horizontal flip
- fill_mode = 'reflect'
- rotation in range = 5°

The following image represents a batch of 4 images given as output of the data generators with augmentation.

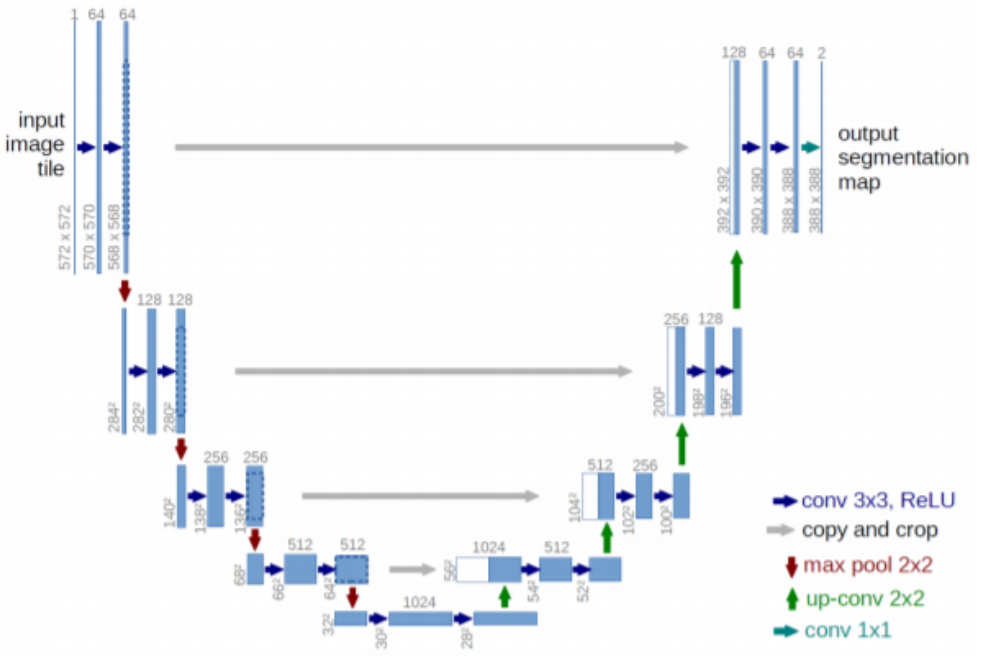


3. Models architectures

We decided to implement 3 different model architectures of neural networks in order to try different approaches and then make a comparison.
The neural networks that we implemented are U-Net, LinkNet and Mask-RCNN.

3.1 U-Net Model

The model that we implemented is an architecture based on U-Net with a Residual Network (ResNet-34) backbone that has been pre-trained on the ImageNet dataset. We took inspiration from the following paper: *The 2ST-UNet for Pneumothorax Segmentation in Chest X-Rays using ResNet34 as a Backbone for U-Net* by Ayat Abedalla et al. [1].
The standard U-Net model is a fully convolutional network developed by Olaf Ronneberger et al. [2] for Bio Medical Image Segmentation. The architecture contains two main paths. The first path is the contraction path (also called as the encoder), which is used to capture the context and the features in the image. The second path is the symmetric expanding path (also called as the decoder) which is used to enable precise localization.
The following image shows the structure of the standard U-Net model.



In our network the encoder is built by inserting the layers of a ResNet-34 pre-trained using the ImageNet dataset. ResNet-34 was proposed by Kaiming He et al. in the paper: *Deep Residual Learning for Image Recognition*[3]. Our ResNet-34 model consists of one 7x7 convolution, followed by a max pooling step and by 4 blocks of similar behavior. Each of the blocks follow the same pattern: they perform some 3 x 3 convolution with a fixed filter dimension (64, 128, 256, 512 respectively), bypassing the input every 2 convolutions through

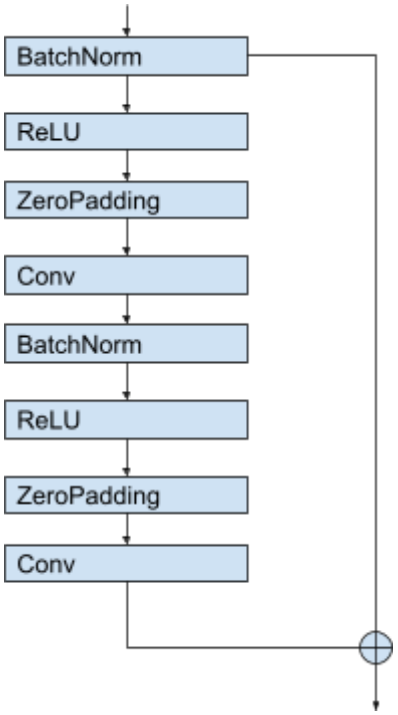
a residual connection. In the ResNet-34 these 4 blocks have a different number of layers: the first it's composed by 3 residual blocks with 6 convolutions, the second 4 residual blocks with 8 convolutions, the third 6 residual blocks with 12 convolutions and the last 3 residual blocks with 6 convolutions.

The decoder has five blocks, each consisting of a 2×2 up-sampling layer followed by two sets of layers, each containing a convolution, batch normalization layer, and Rectified Linear Unit (ReLU) activation layer. In the first four blocks of the decoder, the feature maps after up-sampling are concatenated using skip connections with the feature maps from the same sized part in the encoder.
Finally,we apply a 3×3 convolution layer followed by softmax activation to output the binary masks.

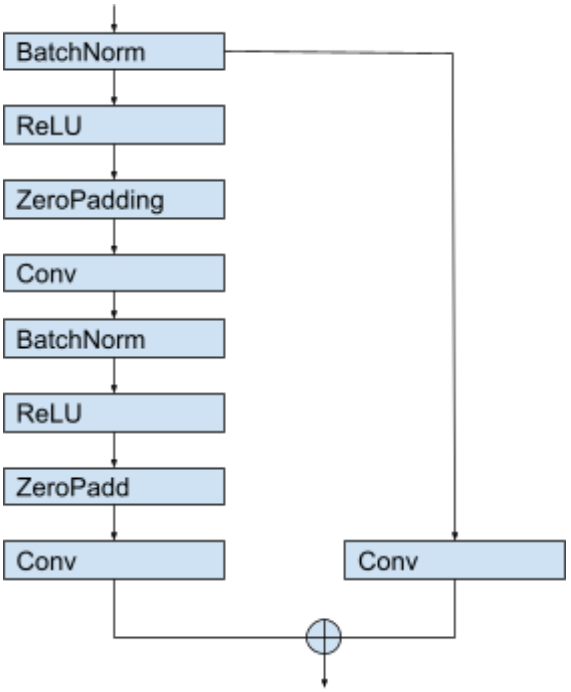
The detailed structure of our network is shown in the following table.

Layer Name	**Output Size**	**Filter Number**	**Remarks**
Input	256 x 256	//	//
Conv 1 MaxPool	128 x 128 64 x 64	64	Kernel Size (7 x 7) Kernel Size (3 x 3)
Encoder 1	64 x 64	64	Residual Block 1 Residual Block 2 Residual Block 3
Encoder 2	32 x 32	128	Residual Block 1 Residual Block 2 x 2 Residual Block 3
Encoder 3	16 x 16	256	Residual Block 1 Residual Block 2 x 4 Residual Block 3
Encoder 4	8 x 8	512	Residual Block 1 Residual Block 2 Residual Block 3
Decoder 1	16 x 16	256	Up-sample 1 (2 x 2) Skip: [Up-sample1, Encoder 3] Decoder Block
Decoder 2	32 x 32	128	Up-sample 2 (2 x 2) Skip: [Up-sample2, Encoder 2] Decoder Block
Decoder 3	64 x 64	64	Up-sample 3 (2 x 2) Skip: [Up-sample 3, Encoder 1] Decoder Block
Decoder 4	128 x 128	32	Up-sample 4 (2 x 2) Skip: [Up-sample4, Conv 1] Decoder Block
Decoder 5	256 x 256	16	Up-sample 5 (2 x 2) Decoder Block
Conv 2	256 x 256	17	kernel Size (3 × 3)

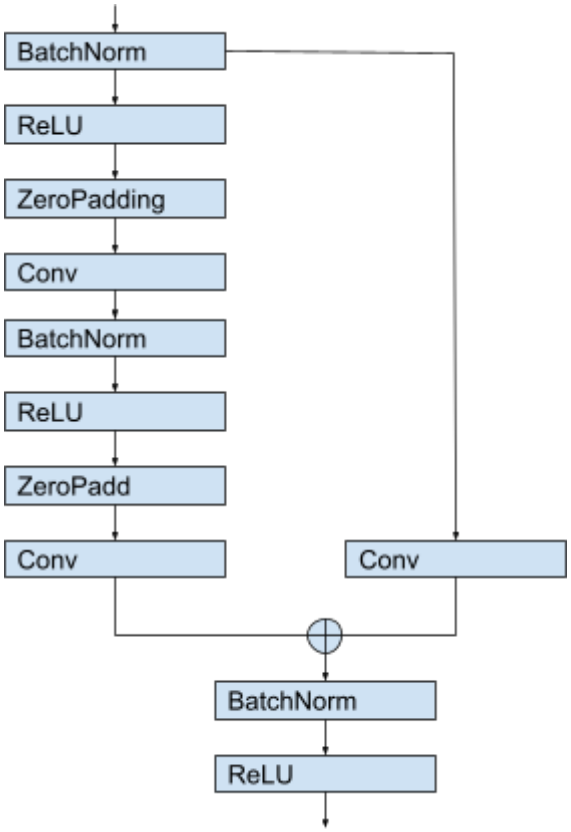
Here are shown the diagrams of the main blocks of the network described in the table.



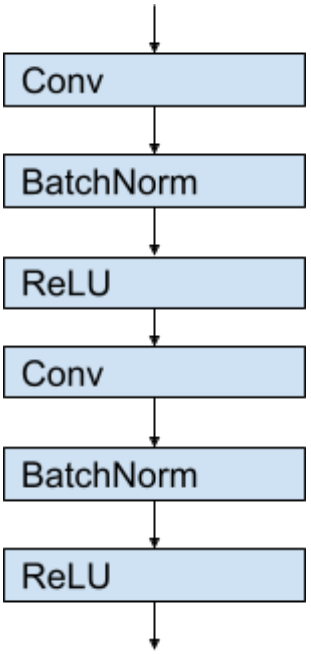
Residual Block 1



Residual Block 2



Residual Block 3



Decoder Block

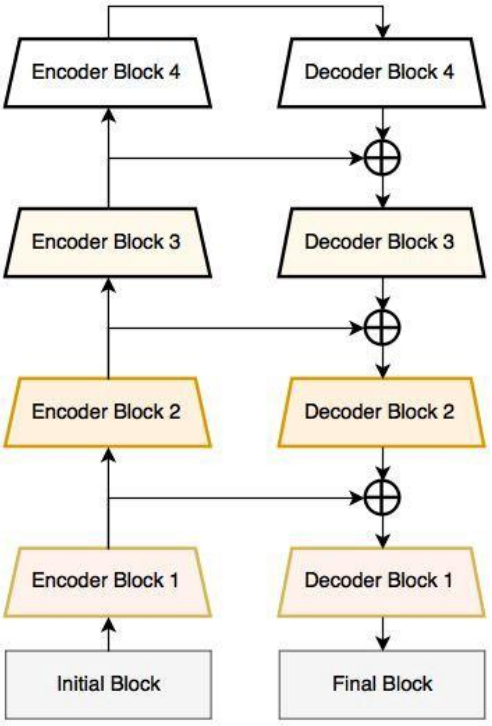
3.2 Link-Net Model

We Implemented Link-net model based on *LinkNet: Exploiting Encoder Representations for Efficient Semantic Segmentation*[4] research paper by Abhishek Chaurasai and Eugenio Culurciello. Our implementation of the LinkNet neural network uses 11.5 million parameters. The main objective of this model is to provide a neural network architecture which has fewer learnable parameters and which can produce substantial results.

This Linknet neural network is inspired by auto-encoders, which consists of encoder-decoder pairs as the core of their network architecture. Here the encoder encodes information into feature space and the decoder maps this information into spatial categorization to perform segmentation. Generally spatial information would be lost in the encoder due to pooling or strided convolution operation. It can be recovered by using pooling indices or by upsampling layers. But in the paper the researchers mentioned the technique of passing spatial information directly from the encoder to the corresponding decoder improving the accuracy along with a significant decrease in processing time. In this way, information that would have been lost at each level of the encoder will be preserved and no additional parameters or operations are wasted in regaining it. In addition, since the decoder is sharing knowledge learnt by the encoder at every layer, the decoder can use fewer parameters.

In LinkNet we are using ResNet18 as an encoder, which is a light network. In our decoder we use full-convolution technique to represent the output segmentation.
The initial block of the encoder performs convolution operation on the input with a kernel of size 7 x 7 and a stride of 2. In this block, we also have a max-pooling layer with size of 3 x 3 and with a stride of 2.
The initial block is followed by a chain of encoder blocks that are linked to the decoders using the Link previously described. Then this chain of encoders is followed by a chain of decoders and by a final block.

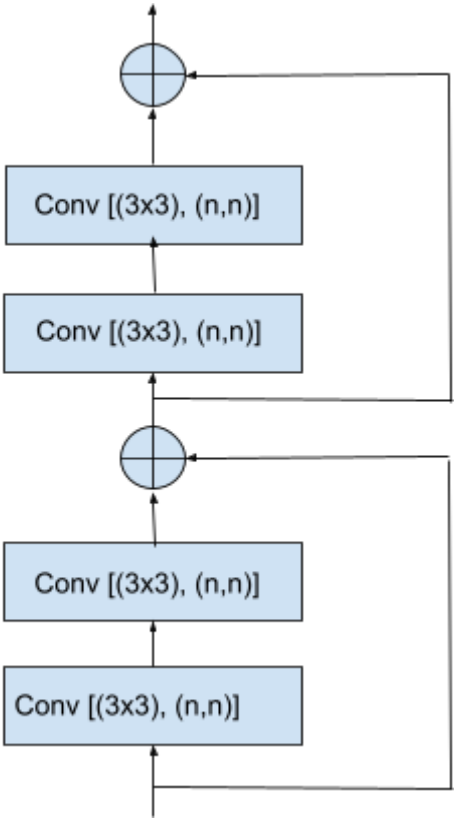
The following image describes the main structure of the LinkNet model architecture.



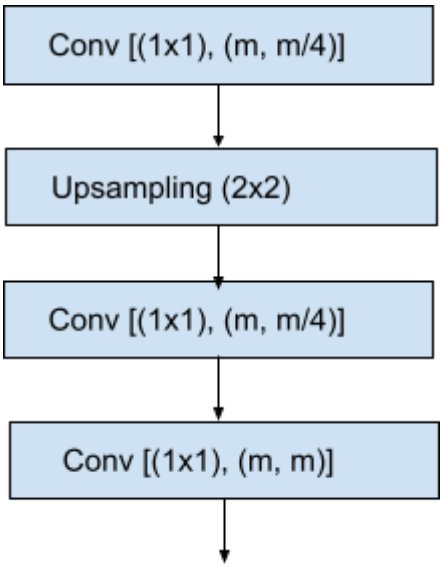
In the Encoder block, we used 2 subsequent times the following block of layers: BatchNormalization, ReLU activation function and then a convolution operation with “n” filters, kernel size of (3 x 3) and with stride of (2 x 2). Here we are performing downsampling with a factor of 2. Then we add the first residual connection.
Then we have again 2 subsequent times the following block of layers: BatchNormalization, ReLU activation function and then a convolution operation with “n” filters, kernel size of (3 x 3) and with stride of (2 x 2). Finally we have the second residual connection that becomes the output of the encoder block.
The Decoder block starts with the following layers: BatchNormalization followed by the ReLu Activation function and by a Convolution operation with a filter of size “input_filter/4” and kernel size of (1 x 1).
Then we used an UpSampling layer with an Upsampling factor of (2 x 2). This layer is followed by a BatchNormalization, a ReLU Activation function and then a convolution operation with a filter of size “input_filter/4” and kernel size of (3 x 3).
At the end of the decoder block, there is BatchNormalization followed by the ReLu Activation function and a final convolution with “n” filters and kernel size of (1 x 1).
The final block is composed by an UpSampling layer (2 x 2) followed by a BatchNormalization, a ReLu Activation function and by a Convolution operation with a filter of size 32 and kernel size of (3 x 3). Then we have another block of BatchNormalization, ReLu Activation function and Convolution with a filter of size 32 and kernel size of (3 x 3).

The following layers UpSampling layer (2 x 2) followed by a BatchNormalization, a ReLu Activation function. The final layer is a convolution operation with a filter equal to the number of classes, a kernel size of (2x2) and a softmax activation function.
In the following images the keyword " /2 " denotes downsampling by a factor of 2 which can be achieved by performing strided convolution operation and " *2 " means upsampling by a factor of 2.

Here there is shown the main structure of the encoder and decoder structure.



Encoder Block



Decoder Block

Here we can see the table of the output and input feature maps:

Block	Encoder		Decoder	
	m	n	m	n
1	64	64	64	64
2	64	128	128	64
3	128	256	256	128
4	256	512	512	256

Layer Name	Output Size	Filter Number	Remark
Input	256 x 256	//	//
Batch Normalization Activation (ReLu) Conv 1 MaxPool	 125 x 125 63 x 63	64	Kernel Size(7 x 7) Strides(2 x 2)
Encoder 1	32 x 32	64	Encoder Block
Encoder 2	16 x 16	128	Encoder Block
Encoder 3	8 x 8	256	Encoder Block
Encoder 4	4 x 4	512	Encoder Block
Decoder 4	8 x 8	256	Decoder Block
Decoder 3	16 x 16	128	Decoder Block Skip: [decoder_4,encoder_3]
Decoder 2	32 x 32	128	Decoder Block Skip: [decoder_3, encoder_2]
Decoder 1	64 x 64	64	Decoder Block Skip: [decoder_2, encoder_1]
Upsampling Batch Normalization Activation (ReLu) Conv 2	128 x 128 128 x 128	64 32	Upsampling factor(2,2) Kernel Size(3, 3)
Batch Normalization Activation (ReLu) Conv 3	128 x 128	32	Kernel Size(3, 3)
Upsampling Batch Normalization Activation (ReLu)	256 x 256	32	Upsampling factor(2,2)
Conv4	256 x 256	17	Kernel Size(2 x 2) Activation Function(Softmax)

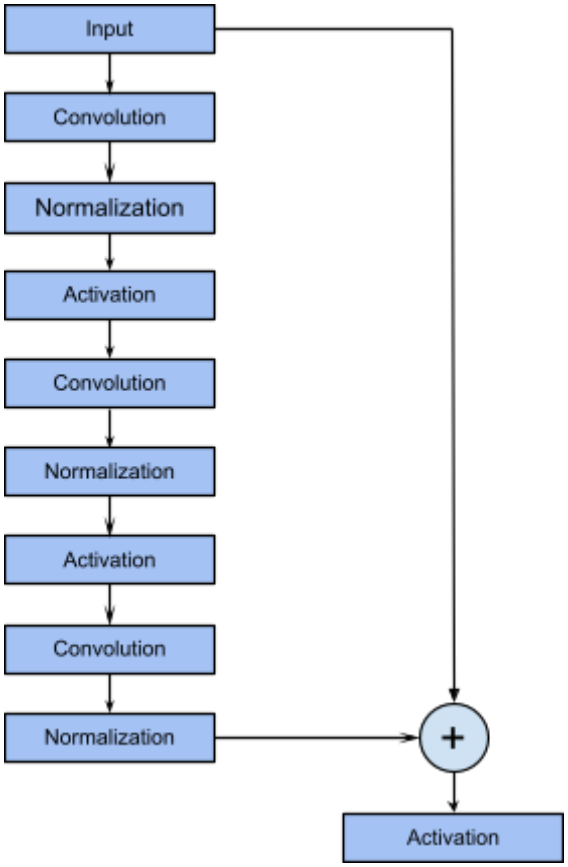
3.3 Mask-RCNN model

The mask RCNN model that we have implemented is based upon the paper *Mask RCNN* [5] with a Residual Network (ResNet-*5*) backbone. As the full model proposed in the paper would be too complex to implement on the limited resources provided by Google Colab we have simplified the model trying to retain the main aspects of the proposed architecture. With our simplifications the model is fully convolutional and can be divided into three main structures : the feature extraction, the bounding box proposals and the mask generation. All convolution should be assumed to have kernel size 3x3 if not differently specified.

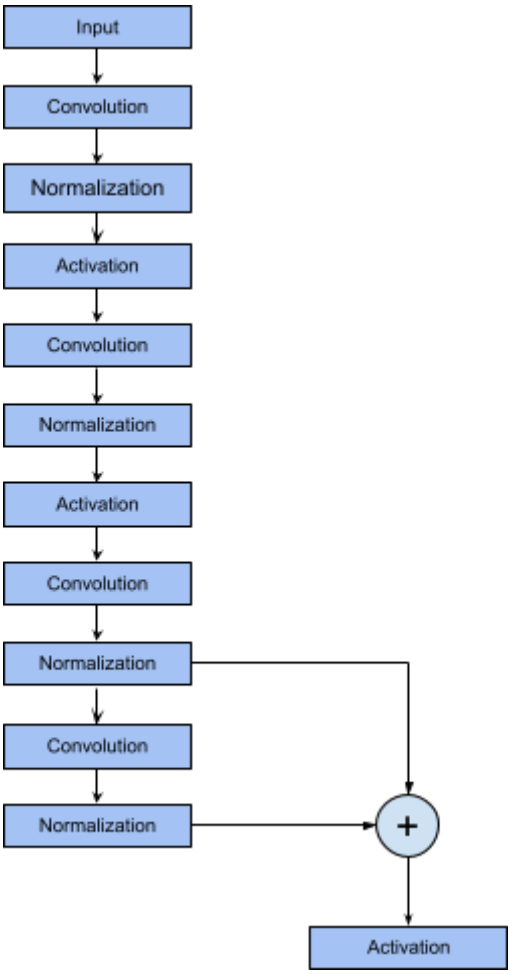
Feature extraction: Is in charge of extracting the features of the image and is represented by the ResNet-5 model. The model outputs at 5 different integration levels which define the size of the feature extracted (32x32 pixels, 16x16 pixels, 8x8 pixels, 4x4 pixels, 2x2 pixels). The ResNet-5 is built upon a convolutional layer with kernel size 7x7 followed by a max pooling layer the rest is formed of 16 blocks of layers which are formed by 3 convolution adding to the output of the block the input tensor in case of the identity_block or the input to the last convolution in case of the conv_block. In particular the number of the filters utilized by each convolution in the ResNet-5 doubles at the shrinking of the integration level. Lastly the add_block is applied to the outputs.
The table belows shows the fine structure of this part of the network

Layer Name	**Output Size**	**Filter Number**	**Remarks**
Input	128 x 128	//	
Batch Normalization Activation (ReLU)	32x32 32x32	// //	
Conv 1 MaxPool	64 x 64 32 x32	64	Kernel Size (7 x 7)
C2	32x32	64,64,256	conv_block stride 1 identity block x2
C3	16x16	128,128,512	conv_block stride 2 identity block x3
C4	8x8	256,256,1024	conv_block stride 2 identity block x5
C5	4x4	512,512,2048	conv_block stride 2 identity block x2
P2	32x32	128	adding _block

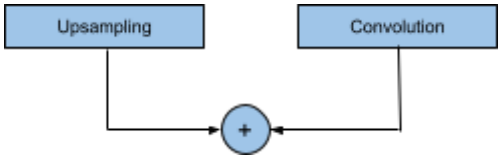
			(P3,C2)
P3	16x16	128	adding _block (P4,C3)
P4	8x8	128	adding _block (P5,C4)
P5-Conv	4x4	128	applied on C5
P6-MaxPooling	2x2		applied on P5



Identity block

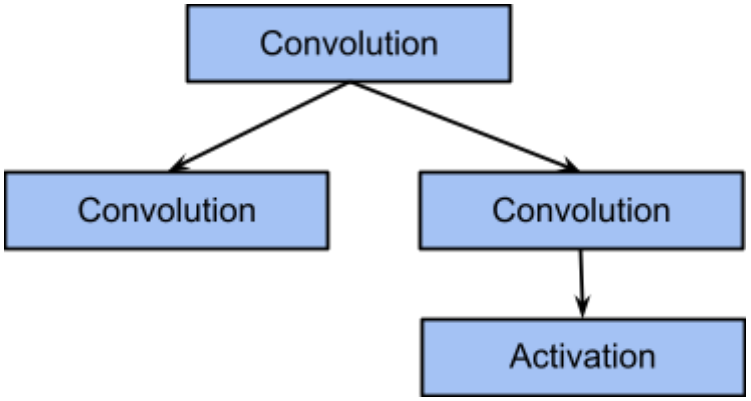


Conv_block



Addition_block

- **Region Proposal Network:** Is in charge of proposing the bounding boxes of the elements present in the image. It is formed by convolutional layers which receive the different feature maps produced by the Resnet and feed two other convolutional layers that output a score indicating the likelihood of an object to be in the bounding box and correction to the anchor boxes. The bounding boxes with the lowest scores are pruned leaving only (128,64,32,16,4) boxes respectively for the different integration levels. These numbers have been selected to both have a higher speed and lower memory footprint. A custom layer (called Proposal layer) applies the corrections to the original anchors and prunes away the boxes having a high intersection over union score with the other boxes outputting them to the loss function



- **Mask Generation:** is the final step executed in the model. The idea behind it is to utilize the bounding box produced in the previous step (before the non maxima suppression) and define binary masks weighted by the score of the boxes. These binary masks are applied to the various feature maps creating sub feature which contain the reduced features that should be correlated only to the object contained inside the bounding boxes. On each of the binary mask a convolution (with same weights for every mask) is applied. A summation over the binary masks is performed so that the sub-feature which have a higher score and appear in more bounding boxes would be more visible. At the same step the maximum among the different bounding boxes is extracted. After this step the resulting sub features map are concatenated with the maps coming from the ResNet and a convolution is applied to this concatenation. The same operations are then applied to the result of the convolutions for the different integration levels. Lastly four new convolutions are applied, the last outputting the different masks for the different classes of the model.

The table belows shows the fine structure of this part of the network.

Layer Name	**Output Size**	**Filter Number**	**Remarks**
Lambda	[32x32x128,16x16x64,8x8x32,4x4x16,2x2x4]		creation of binary masks based on the bounding boxes
Time distributed Convolution	[32x32x128,16x16x64,8x8x32,4x4x16,2x2x4]	128	the time distribution ensure that
Lambda-summation	[32x32,16x16,8x8,4x4,2x2]		summation over the different binary masks
Lambda-Maximum	[32x32,16x16,8x8,4x4,2x2]		maximum over the different binary masks
Concat	[32x32x256,16x16x256,8x8x256,4x4x256,2x2x256]		concatenation of features map with summation
Concat	[32x32x256,16x16x256,8x8x256,4x4x256,2x2x256]		concatenation of features map with Maximum
Conv Concatenation Upsampling Conv	 32x32	256 256	each concatenated block is convoluted and the resulting tensor concatenated and convoluted
Conv Upsampling Batch Normalization Activation (ReLU)	64x64	256	
Conv Upsampling Batch Normalization Activation (ReLU)	128x128	256	
Conv	128x128	2048	Kernel Size (1x1)
Conv		17	Kernel Size (1x1)

The model presented has a glaring difference from the mask RCNN, while the creation of the binary masks, as made, decouples (at least partially) the localization task from the identifying task, the different classes are not obtained in parallel as the model suggests. This is done on purpose as the semantic segmentation task takes into account the existence of the background as a class which is far easier classified in contrast with the other classes which, in turn, impacts also our localization of the objects.

4. Experiments

In this section we present the experiments that we did using the different model architectures. As default parameters between our experiments we used:

- Adam Optimizer
- Learning rate of 0.0001
- The callback EarlyStopping(patience=5) based on the validation loss
- The callback ReduceLROnPlateau(factor=0.1, patience=3, min_lr=0.00001) based on the validation loss
- Class weights applied to the loss functions in order to avoid the problem of the imbalanced classes

We tried the following loss functions.
X is the predicted set of pixels and Y is the ground truth.

- Multiclass weighted squared dice loss

$$SQDL = 1 - \frac{2 \times |X \cap Y| \times class\ weights}{|X|^2 + |Y|^2 \times class\ weights}$$

  The multiplied by class weights should be interpreted as a dot product with a summation over the resulting values.

- Multiclass weighted dice loss [6]

$$DL = 1 - \frac{2 \times |X \cap Y| \times class\ weights}{|X| + |Y| \times class\ weights}$$

  The multiplied by class weights should be interpreted as a dot product with a summation over the resulting values.

- Multiclass weighted cross entropy [7]

$$WCE = -\Sigma X \times log(Y) \times class\ weights$$

- Multiclass focal loss [8]

$$FL = -\sum class\_weights \times (1 - Y)^{\gamma} \times X \times log(Y)$$

During the training the metrics that we used are:

- IoU score from the segmentation_models library [9]
- F1-score from the segmentation_models library

4.1 U-Net Model experiments

The first experimentation that we have done is to try a default U-Net model in order to have an idea of the performances that this architecture can reach. After some tests we decided to

pass to a more sophisticated structure of the neural network. So after reading the paper we decided to implement a U-Net model architecture using a ResNet-34 as encoder.
During the implementation of the network, for the tests, we used only 5 classes in order to reduce the time necessary to execute the training. Once we found a good configuration of the parameters of the network we passed to execute the training with 17 classes.
For the fine-tuning of the hyper-parameters we used 12 epochs due to the long time of the training and to the restrictions applied by the Google Colab platform on the use of the GPU. We tried first using a batch size of 16 but after several attempts we passed to 12 due to RAM problems, and then to a batch size of 8 which allowed us to achieve better results.
So the final model configurations that we present are the following:

- Model 1:
    - 12 epochs
    - Batch_size = 8
    - Multiclass weighted squared dice loss

- Model 2:
    - 12 epochs
    - Batch_size = 8
    - Multiclass weighted cross entropy

- Model 3:
    - 12 epochs
    - Batch_size = 8
    - Multiclass focal loss

4.2 Link-Net Model experiments

We implemented the Link-Net model architecture taking inspiration from the paper[4]. After building the model, we experimented with different configurations changing the batch size and loss function. Firstly we tried with a batch size of 16, but as happened with the U-Net, due to the limited ram size of the Google Colab this caused the crash of the environment. After some trial and error we noticed that a batch size of 8 could give us good results. We have tried three different configurations. Our first attempt was utilizing the Multiclass Weighted Cross Entropy loss function. This loss function did not give us good results. So in our second attempt we used the Multiclass Dice loss which improved greatly upon the previous results. In our final attempt we used the Multiclass Squared Dice loss which obtained good results as well.

- Model 1:
    - 15 epochs
    - Batch_size = 8
    - Multiclass weighted squared dice loss

- Model 2:
    - 15 epochs
    - Batch_size = 8
    - Multiclass weighted dice loss

- Model 3:
    - 15 epochs
    - Batch_size = 8
    - Multiclass weighted cross entropy

4.3 Mask-RCNN model experiments

During the creation of the model the main problematics presented have been in relation with the production of the binary masks. Different approaches have been tried to produce the binary mask and at the same time have a layer capable of propagating the Gradient. The first attempt utilized a custom layer to obtain the binary masks assigning zeros and ones to a Variable tensor. The approach had some good results but it became intractable with batches bigger than one as the Variable tensor cannot have mutable dimensions (as his purpose is to store the weights of the model). The current applied method is to produce the different maps by executing an outer product with between vectors (rank-1 tensor). The differentiability issue has been solved by outputting the bounding boxes and supplying them to an Intersection over union loss function directly.

Some tests performed with only 5 classes to receive results in more manageable times have shown that the batch size has a small impact with the resulting intersection over union. In the end we chose a batch size of 8 to balance between the run time of the training and obtain the best results. We have also tried to lower the learning rate as we noted that the model tends to have a change of slope in the loss that we thought was caused by a too high learning rate. The change didn't produce results so we have sticked with the10^{-4}.
As the model has 36 million parameters 10 epochs were deemed a reasonable choice to cope with the constraint of Google Colab. Taking in account the previous considerations we have tried the following models:

- Model 1:
    - 10 epochs
    - Batch_size = 8
    - Multiclass weighted squared dice loss

- Model2:
    - 10 epochs
    - Batch_size = 8
    - Multiclass focal loss

- Model 3:
    - 10 epochs
    - Batch_size = 8
    - Multiclass weighted dice loss

- Model 4:
    - 10 epochs
    - Batch_size = 8
    - Multiclass weighted cross entropy

5. Comparison between models

In this section we present first a comparison of the results obtained by the training configurations that we have discussed before, and then a comparison between the best configurations of our different model architectures.
For the evaluation of the models we followed what is written in the Discussion of the AiCrowd website.
We used the following metrics:

- IoU score from the segmentation_models library
- Precision from Keras
- Recall from Keras

We calculated these metrics for every image in the test set and then we made the average.

5.1 Comparison between different model configurations

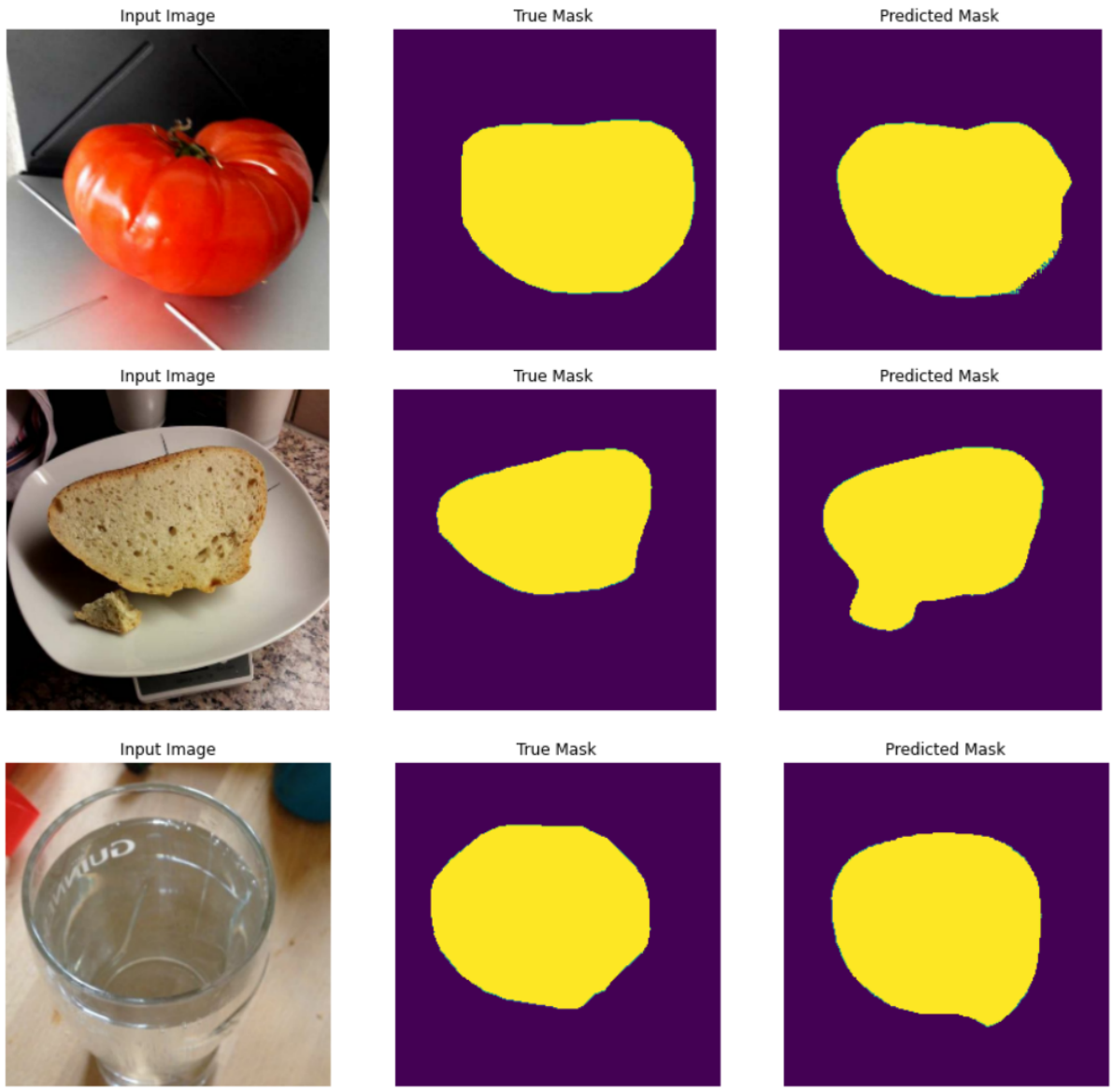
5.1.1 U-Net Model comparison

In the following table we present the results obtained by the different configurations presented before of the U-Net model.

Configuration	Number of Epochs	Loss Function	Batch size	IoU score	Precision	Recall
1	12	Squared Dice Loss	8	0.93	0.88	0.83
2	12	Cross Entropy	8	0.83	0.60	0.59
3	12	Focal Loss	8	0.87	0.67	0.37

As we can see the best configuration is the one using the multiclass squared dice loss. This particular loss function is able to guide the learning in the direction of the optimization of the Iou coefficient in a very good way.
The configurations using the multiclass weighted cross entropy and the multiclass focal loss istead reached not very excellent results.
The same discussion holds for the prediction of the masks.

These are the Predicted Masks we got from using our best configuration.



5.1.2 Link-Net Model comparison

In the following table we present the results obtained by the different configurations presented before of the LinkNet model.

Configuration	Number of Epochs	Loss Function	Batch size	IoU score	Precision	Recall
1	15	Cross Entropy	8	0.61	0.60	0.58
2	15	Dice Loss	8	0.89	0.77	0.76
3	15	Squared Dice Loss	8	0.87	0.81	0.71

Here we can observe that the best configuration is one using the multiclass weighted dice loss. Using the multiclass weighted squared dice loss the results are quite similar, there is not a big difference. By using cross entropy loss instead, the model didn't learn much and obtained poor results.
The same discussion holds for the prediction of the masks. The mask predicted using the third configuration is of course the best one, the predictions of the fist model are not very precise.

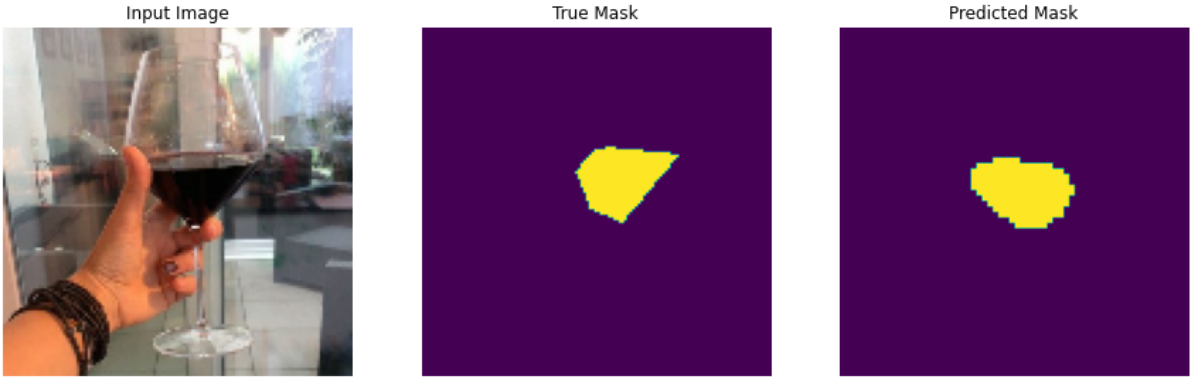
These are the Predicted Masks we got from using our best configuration.

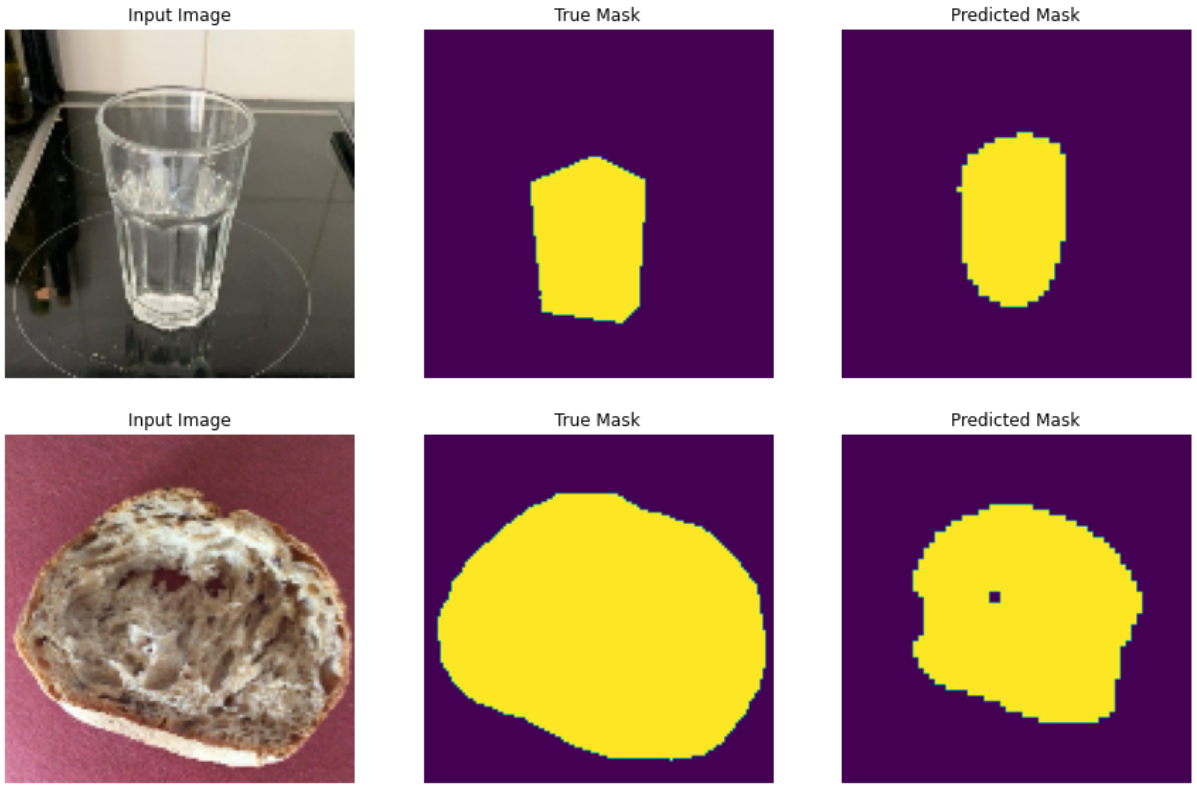
5.1.3 Mask-RCNN model comparison

in the following table we present the different results obtained by the different configurations applied to the Mask-RCNN model

Configuration	Number of Epochs	Loss Function	Batch size	IoU score	Precision	Recall
1	10	Squared Dice Loss	8	0.90	0.79	0.74
2	10	Dice Loss*	8	0.91	0.74	0.71
3	10	Focal Loss	8	0.82	0.83	0.08
4	10	Cross Entropy	8	0.68	0.64	0.59

These are some Predicted Masks we got from using our best configuration. As can be seen the square dice loss has the best results among the utilized metrics. The cross entropy seems unable to both identify and localize to a satisfactory degree the data. The focal loss while obtaining good precision and localization was able to correctly recognize an extremely low number of images. Lastly the Dice Loss behaviour was puzzling because while the metrics are good (similar to the squared dice loss) the masks produced are similar to white noise.





5.2 Comparison between our best models

In this section we make a comparison between the best configuration of the different model architecture that we tried.

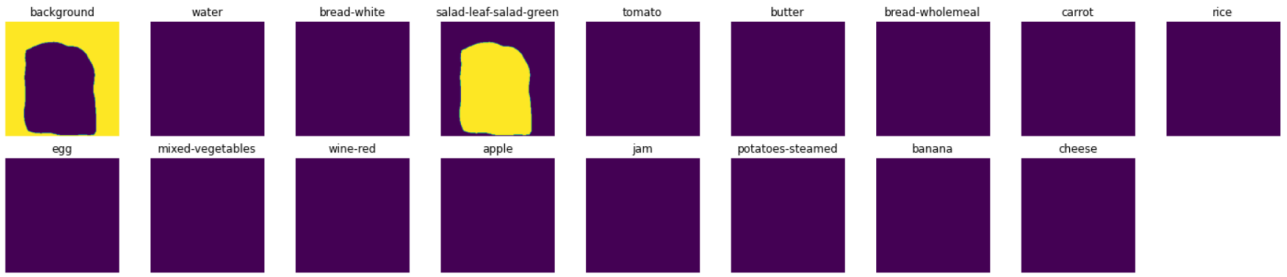
Model	Number of Epochs	Loss Function	Batch size	IoU score	Precision	Recall
U-Net	12	Squared Dice Loss	8	0.93	0.88	0.83
Mask-RCNN	10	Squared Dice Loss	8	0.90	0.79	0.74
Link-Net	15	Dice Loss	8	0.89	0.77	0.74

The U-Net as we can see is the model architecture that archives best results, this probably because we have used as encoder a Resnet-34 pre-trained using the Imagenet dataset. Using the transfer learning technique gave a push up to the performances of the U-net.
On the other hand the simplified Mask-RCNN model, while having a similar intersection over union, has reduced precision and recall. This is probably due to the necessary simplifications applied to the original network to allow it to run with the limited resources at hand. Finally, the Link-Net model performs in a quite similar way to the Mask-RCNN model reaching very good results for the IoU score and quite good results for precision and recall.

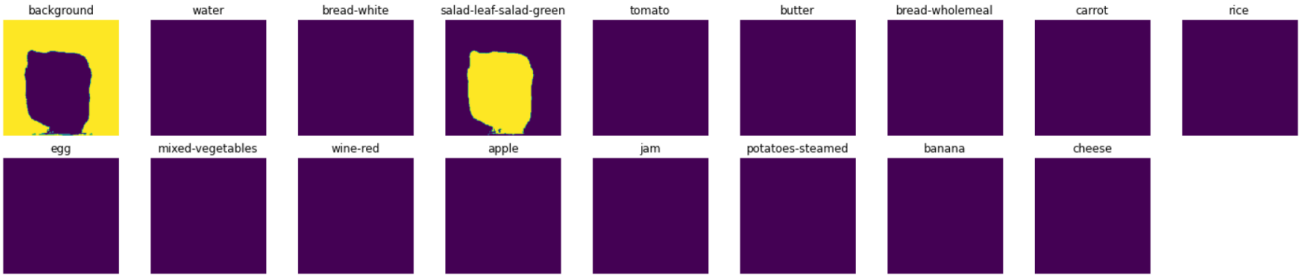
5.2.1 Display of the model predictions

Here we show how our models predict the masks by making a comparison based on the same image.

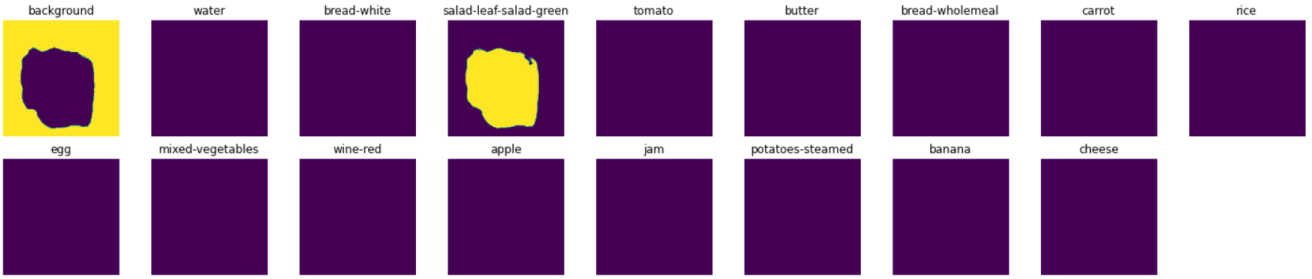
This is the true mask:



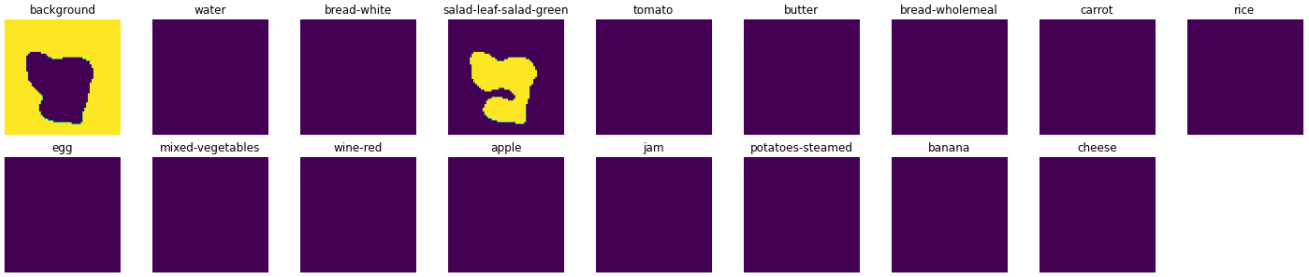
This is the prediction of the U-Net model:



This is the prediction of the Link-Net model:



This is the prediction of the Mask-RCNN model:



6 Conclusion and future works

As the three different models are based on different logics our conclusions differ based on the peculiarities of our models. However, it can be seen that in all the different network architectures the dice like losses produced much more precise results with respect to other metrics, being tailored for this particular kind of task. Also we noticed that the masks presented in the dataset are not consistent which caused some ulterior difficulties to the localization and the classes are not mutually exclusive. So first of all the quality of the annotations in the dataset should be improved, secondly increasing the number of images of some classes can help to improve the performances of the models.
Finally we were limited by the computational power. Working on machines with a higher computational power and without the Google Colab limitations can of course give us a boost in training our models. Surely training our models for more epochs could have led us to archiving better results.

6.1 Mask-RCNN conclusions

The simplified Mask-RCNN model while obtaining some good results in the localization task (as can be seen by the high intersection over union score) have mediocre results in the identification task, we individuate the problematics and hypothesize on possible solution in the following issues:

- **Tendency to favor elements in the center of the image:** this is probably due to the summing and the method of pruning of the bounding boxes based on the score because very large boxes would be advantaged as they naturally have a higher probability to contain an object. So having lots of them overlapping could cause this problem. A possible solution may be achieved by varying the non maxima suppression parameters present in the model or make them learnable by the model itself. Also the very reduced size of the features maps, and consequently of the binary maps, can be to blame too even though it may not be solved easily as it ultimately depends on the available hardware

- **Problem with overlapping elements:** this derives from the nature of the last layer in our model. More precisely because the assignment of the class is conducted in a single layer the different classes compete in their discrimination in case of multiple object present localized near each other the competition prevents the model to lean toward one. A possible solution would be to decouple at the binary mask creation time the different classes and so perform the search for each class in parallel. This approach, Alas, still impacts heavily on the underlying hardware even though more on its computational capabilities as it requires the creation and the reduction of an increasing number of tensors linear with the number of classes.

- **Problem with multiple elements:** to reinforce the prediction between the masks the best feature maps with the highest values are selected this seem to increase the

certantanty of the predicted object but reduces the certainty of other prediction it will be solved by taking not only the maximum but have a more lenient criteria to include binary masks.

Lastly for the maskRCNN we think that it would produce better results if a more performant network than the ResNet-5 for features extraction.

6.2 Link-Net conclusions

The LinkNet model obtained a good accuracy in this segmentation task. In Linknet model architecture, we have actually used the architecture of ResNet-18 as an encoder block, but we have not used the transfer learning technique. So as possible future work we can try to do a pre-training of the encoder using the Imagenet dataset and then load the weights .
Due to time limits we have not fine-tuned all the parameters of our network, so with a more accurate trial and error approach our model might obtain better results. For example a possible future work can be trying a combination of our loss function or make the same test using different optimizers or learning rate.

6.3 U-Net conclusions

The U-Net model achieved good results in this particular task. Possible future works can be for example trying a more performant neural network as encoder, in order to increase the quality of the feature extracted and consequently the prediction of the mask.
Another possible change that can be done is to replace in the decoder block the Upsampling layers with a Transposed convolution layer (but we don’t know if this can improve our results).
Moreover we think that a very important possible future work can be the reduction of the number of parameters of the neural network. This could help to make the model lighter and can reduce the time taken from every training of the network.
Finally a more accurate fine-tuning might help to increase the performance of the model; for example we have not tried to change the learning rate and the optimizer for time problems. Trying all the possible combinations of the hyper-parameters of the network can lead to finding the best configuration of the model.

To conclude we think that the most important improvement can be done by refining the annotation of the dataset. The quality of the data is crucial; the principle garbage in, garbage out always holds.

References

1. *The 2ST-UNet for Pneumothorax Segmentation in Chest X-Rays using ResNet34 as a Backbone for U-Net* by Ayat Abedalla et al. https://arxiv.org/pdf/2009.02805.pdf

2. *U-Net: Convolutional Networks for Biomedical Image Segmentation by* Olaf Ronneberger et al. https://arxiv.org/abs/1505.04597

3. *Deep Residual Learning for Image Recognition* by Kaiming He et al. https://arxiv.org/pdf/1512.03385.pdf

4. *Exploiting Encoder Representations for Efficient Semantic Segmentation* by Abhishek Chaurasai et al. https://arxiv.org/pdf/1707.03718.pdf

5. *Mask R-CNN* by Kaiming He et al. https://arxiv.org/pdf/1703.06870.pdf

6. *Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations* by Carol H Sudre et al. https://arxiv.org/abs/1707.03237

7. *Multiclass Weighted Loss for Instance Segmentation of Cluttered Cells* by Fidel A. et al. https://arxiv.org/abs/1708.02002

8. *Focal Loss for Dense Object Detection* by Tsung-Yi Lin et al. https://arxiv.org/abs/1708.02002

9. Segmentation models library: https://github.com/qubvel/segmentation_models