ASTRONET: DEEP LEARNING APPROACHES TO THE SKA SCIENCE DATA CHALLENGE 1

ASTRONET

repository link

Martina Rossini
martina.rossini3@studio.unibo.it

Vairo Di Pasquale
vairo.dipasquale@studio.unibo.it

Abstract

In this work we developed a series of deep learning models to detect and classify astronomical sources from radio images. In particular, we approached the problem both as an object detection and as an image segmentation task, implementing from scratch both YOLOv4 and U-Net. The first network did not achieve an optimal performance (partially due to shortage of training data), while the latter reached a an accuracy of 97.3%.

1. Introduction

The Square Kilometer Array (SKA) will be the world's largest radio telescope. SKA Challenges are international challenges that involve astronomers and software specialists from all over the world: they have been regularly issued to the community these last few years as a way to prepare for the analysis and management of the massive amount of data that SKA is going to produce. In this paper we deal with the SKA Science Data 1 (SDC1), which addresses source finding, characterization and classification for radio continuum sources starting from simulated sky images. Our approach includes two different deep learning methodologies: one of Object detection, where we rely on YOLOv4 [2] in particular, and another of Object segmentation, where we rely on U-Net [8]. We analyze these two different implementations, which, starting from the same given dataset, raise different weaknesses and strengths and, above all, produce different results.

The released train dataset consist of 3 image files in FITS format. Each file consists of a simulated SKA continuum image in total intensity at 3 frequencies:

1. 560 Mhz, representative of SKA Mid Band 1
2. 1.4 GHz, representative of SKA Mid Band 2
3. 9.2 GHz, representative of SKA Mid Band 5

Moreover, all the train images have a telescope integration of 1000h and are paired with a list of embedded light sources and their properties within 5% of the Field of View (FoV). In particular, this information was given in the form of an ASCII table with the columns shown in Table 1: as we can clearly see, the galaxies in the dataset are distributed in three different classes, Steep Spectrum AGN (SS-AGN, associated to index 1), Flat Spectrum AGN (FS-AGN, index 2) and Star-Forming Galaxies (SFG, index 3).

2. Data Preprocessing

FITS (Flexible Image Transport System) is the standard format for astronomical data [1]. We thus had to use a specific Python package, called astropy, in order to read the contents of our dataset and perform the necessary preprocessing steps on it. In particular, we decided to only use the

Column	ID	Unit of measure	Description
1	ID	none	Source ID
2	RA (core)	degs	Right ascension of the source core
3	DEC (core)	degs	DEcination of the source core
4	RA (centroid)	degs	Right ascension of the source centroid
5	DEC (centroid)	degs	Declination of the source centroid
6	FLUX	Jy	integrated flux density
7	Core frac	none	integrated flux density of core/total
8	BMAJ	arcsec	major axis dimension
9	BMIN	arcsec	minor axis dimension
10	PA	degs	PA (measured clockwise from the longitude-wise direction)
11	SIZE	none	1,2,3 for LAS, Gaussian, Exponential
12	CLASS	none	1,2,3 for SS-AGNs, FS-AGNs,SFGs
13	SELECTION	none	0,1 to record that the source has not/has been injected in the simulated map due to noise level
14	x	none	pixel x coordinate of the centroid, starting from 0
15	y	none	pixel y coordinate of the centroid,starting from 0

Table 1: Challenge training set

SKA Mid Band 1 and 2 data files, thus filtering out the Mid Band 5 one, because we found the latter to be very noisy and to contain only a small amount of interesting sources. Once each image and its corresponding label file were loaded in memory, we first filtered out the sources with `SELECTION = 0`, because they were not injected in the simulated sky map due to their noise level (see Table 1). Then we tried to extract only the sources with an high signal to noise ratio, which are more clearly visible in the image: this filter was initially implemented manually, by computing the `rms` of the whole image and removing from the labels the objects with an intensity lower than $rms \cdot k$, with k ranging from 2 to 5. However, the obtained result was still not satisfactory, thus we decided to use the pre-filtered label files offered by ICRAR, one of the teams that took part in the original competition.

In order to use the images as input to out networks, we then decided to crop them into smaller cutouts of 128x128 pixels and to apply the same image denoising method used in [10]: we computed mean, median and standard deviation of the cutout using astropy's sigma_clipped_stats and then set to zero all the pixels with intensity less than 3.5 times the standard deviation. This enabled us to highlight the contribution of the sources and to remove unwanted background noise. Finally, we also had to convert the original astronomical coordinates provided to us as a ground truth to bounding boxes in pixel coordinates. We used the pixel-wise x and y coordinates of the centroids, the angle `PA` and the major and minor axis of the elliptical source (respectively `BMAJ` and `BMIN`), which the challenge provided, in order to generate a tight bounding box for every object in the image. The code we used to this aim can be found in Listing 1.

```
def ellipse_to_box(phi, major, minor, x, y):
  axis_ux = major * np.cos(phi)
  axis_uy = major * np.sin(phi)
  axis_vx = minor * np.cos(phi + np.pi / 2)
  axis_vy = minor * np.sin(phi + np.pi / 2)

  box_halfwidth = np.sqrt(axis_ux ** 2 + axis_vx
    ** 2)
  box_halfheight = np.sqrt(axis_uy ** 2 + axis_vy
     ** 2)

  xmin, ymin = x - box_halfwidth, y -
    box_halfheight
  xmax, ymax = x + box_halfwidth, y +
    box_halfheight

  return (xmin, ymin, xmax, ymax)
```

Listing 1: Ellipse to Bounding Box's function

We then decided to save our preprocessed data in `TFRecord` format and to load it using the `tf.data.Dataset` API, which supports writing descriptive and efficient input pipelines. When building these compact data representations we explored two different approaches: initially, tried to convert each FITS cutout to a PNG image, thus rescaling the radio values between 0 and 255. However, we found it difficoult to obtain images with good contrast, something our models' performance suffered from: we then decided to simply store the corresponding numpy array. Moreover, note that while saving the data to `TFRecord` we changed the index assigned to every class label, decrementing it by one, due to a `tensorflow` requirement.

2.1. YOLO data processing

YOLOv4 [2] is able to detect objects at three different scales and, thus, expects three different ground truth labels, one for each stage. We then had to convert our ground truth boxes in labels of the form `(batch_size, grid_size, grid_size, N,`

`bbox, conf, classes)`, where `N` represents the number of anchors assigned to a scale (by default, 3). In order to do this we created an empty tensor of the correct shape and gradually filled it with values: we simply computed the IoU (Intersection over Union) between boxes and anchors and then matched each box with the anchor that got the highest score. In the corresponding tensor position we placed the box coordinates, a value of 1 for the confidence (indicating that we are certain of the presence of a galaxy in that location) and the object class one hot encoded.

Once the correct labels were created, we also implemented two different image augmentation steps, in order to try and mitigate the problems that arise while training a deep learning model on a limited amount of data. Inspired by [10], we first decided to rotate the image by a random value chosen between 90, 180 and 270 in 50% of the times and to either flip it horizontally or vertically. This was done using the Python library imgaug, which proved to be an effective tool to apply the transformations on both images and boxes at the same time. Finally, we also applied a random brightness transformation using the `tf.image` utilities. Note that we could not apply other color/contrast transformations, seeing as our input image (the numpy array containing radio values) has just one channel.

Finally, before passing the images as input to our network, we also normalized their pixel values using mean and standard deviation.

2.2. U-Net data processing

The U-Net [8] is a network for image segmentation, thus it expects labels in the form of segmentation masks. A simple way to generate these masks starting from our bounding boxes is the following: we consider a fourth class (with index 3.0) for the black background, instantiate a tensor of the same size of the image (128x128) and filled with '3's and then, for every box b and class index c in the training set, we fill the corresponding space in the tensor with c.

Moreover, since our dataset is strongly unbalanced in favour of class 2.0 (SFG), we also decided to compute the sample weights for every image. The easiest way to do this is to use the label as an index into a class_weight list (which was derived using the method compute_class_weight of `scikit-learn`). Again, before passing images and labels as input to our model we normalized them.

3. Object Detection Approach

For this first approach we choose to rely on YOLOv4, a one-stage object detection model which predicts the bounding boxes and classes for the whole image at once. We choose this model, even if it is known to struggle a bit in detecting small objects, for a couple of different reasons: firstly, we though that a little loss of accuracy could be acceptable, if we'd be able to get fast predictions on the huge amount of data that the SKA telescope is expected to output. Secondly, we though that the three different scales at which this model is able to predict could be extremely useful when we have to deal with astronomical data at different frequencies and levels of telescope integration, which result in images with distinct zoom levels.

In the following sections, we'll briefly touch on the structure of this network and the loss functions that we used during training.

3.1. Network Structure

As we already anticipated, YOLOv4 makes predictions at three different scales, which are given by downsampling the input image by 32, 16 and 8 respectively. Thus, for our 128x128 pictures the three final feature maps are respectively of size 4x4, 8x8 and 16x16. During our experiments, we found that, as expected, the 4x4 feature map is too small for our objects, which only take up an average of about 20 pixels each.

In order to find the location of many objects in the same image, YOLOv4 uses anchor boxes as a prior: by default, it uses three anchors per scale, but this number could also change depending on the size of the bounding boxes to predict. The prediction at each scale is made through a 1x1 convolution with a number of filters equal to the product between the number of anchors assigned to that scale and `NUM_CLASSES + 5` (where the 5 is due to the fact that we need 1 confidence value and 4 coordinates to locate the object).

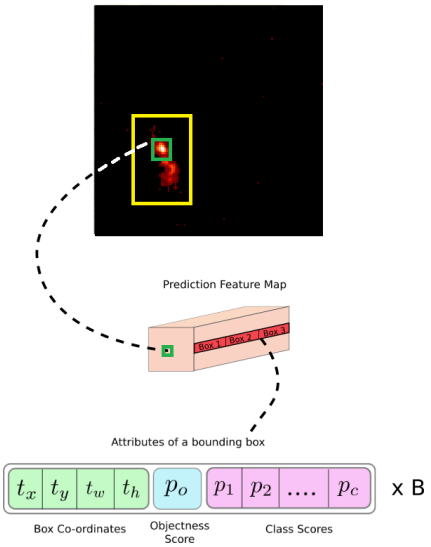

Figure 1: YOLO prediction example

3.1.1 Backbone

The backbone of an object detector is typically a deep convolutional neural network which aims to extract essential features from the input image. In the original YOLOv4 paper [2] the authors tried various backbone networks, among which they selected the CSPDarknet53 [9]: the code was originally written in C, so we had to adapt it to run in Python using TensorFlow 2.0.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 2: CSPDarknet53 with an input image of 256x256

3.1.2 Neck

Next, an object detection network needs to be able to mix and combine the features formed by the backbone in order to prepare for the actual localization and classification step: this is what the Neck part of the detector takes care of. YOLOv4's authors choose PANet [5] as a feature aggregator and then add an SPP [3] block immediately after the darknet in order to increase the receptive field of the neurons and to better separate the more important features detected by the backbone.

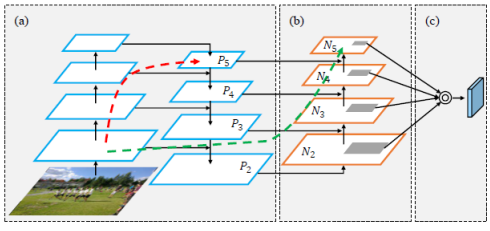

Figure 3: Graphic description of PANet feature aggregator

3.1.3 Head

Finally, for the actual anchor-based detection, YOLOv4 deploys the same head as YOLOv3 [6]: each cell in an output layer's feature map predicts one box per anchor and – as already anticipated – each box prediction consists of 4 values to locate the box (more precisely, two values for box center offset and two values for box size scale), 1 objectness score and a probability score for each class. In order to intuitively understand how these relative box values are used during the postprocessing stage to derive actual bounding boxes coordinates refer to Figure 4.

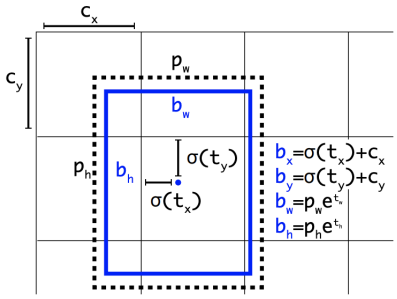

Figure 4: YOLOv3 box postprocessing

3.2. Loss Functions

During our experiments we tried different loss functions, both for box regression and for class and probability score estimation. In particular, for box regression we tried `CIoU`, `DIoU`, `GIoU` and the classic `L2`, while for class and confidence we employed first the `binary crossentropy`, then the `focal loss` and finally we also tried the `varifocal loss`. A brief explaination of all these metrics and of how we computed them can be found in Sections 3.2.1 and 3.2.2; note that most of these metrics were part of the Bag of Freebies introduced by the authors of the original YOLOv4 paper, but the `varifocal loss` is a quite recent metric that we decided to try and implement for this specific project.

3.2.1 Box Regression

The archetypal measure used to evaluate the quality of a predicted bounding box for an object is the `L2` loss, also known as the `least square error`, which can be computed as:

$$L2 = \sum_{i=1}^{n} (y_{true} - y_{pred})^2 \tag{1}$$

However, this measure tries to optimize each coordinate independently from the others. The `IoU` (Intersection over

Union) loss was then introduced in order to consider the bounding box as a unit and provide more accurate predictions: it's quite easily computed as the ratio between the intersection area of ground truth and prediction boxes and their union area. Starting from this second loss function, many variations can be derived, each new one addressing different shortcomings of the previous. A weakness of the `IoU` loss is that it evaluates to zero when the two boxes don't overlap but doesn't tell us anything about how far they are from each other. The `GIoU` (Genaralized IoU) [7] tries to solve this issue by also considering the smallest enclosing box: in this way the network can work to minimize the empty distance between the ground truth and the predicted box.

One issue we can encounter while using the `GIoU` loss is that it converges to `IoU` when the ground truth box is enclosed in the prediction, meaning that we may end up with too large predicted boxes. To solve this obstacle the `DIoU` (Distance IoU) [12] was introduced, where a penalty term is added to the `IoU` in order to reduce the distance between the central points of two bounding boxes, accelerating convergence. The formula to compute this metric is the following:

$$DIoU = 1 - IoU + \frac{p^2(b, b^{gt})}{c^2} \tag{2}$$

where c is the diagonal lenght of the smallest enclosing box covering two boxes and $\rho(b, b^{gt})$ is the distance between their central points.

Finally, `CIoU` (Complete IoU) [12] is just a variant of the `DIoU` with an additional term that also takes into consideration the boxes' aspect ratios. As we will shown in Section 3.4, the metric that worked better for our application is the `DIoU`.

3.2.2 Class and Confidence Estimation

The typical loss function for classification tasks is the `cross-entropy`. However, usually, object detection networks have to evaluate a huge amount of candidate box locations per image, and only a small portion of them actually contains interesting objects; moreover, the class imbalance problem is notoriously hard to solve for object detection datasets as it is not always straightforward to obtain new labeled images and/or to augment the images that only contain the minority class(es). The `cross-entropy` loss attempts to solve this imbalance problem by assigning more weights to difficult data points, with limited results.

The `focal loss` [4] is a simple alteration to the standard `cross-entropy` loss: it introduces a modulating factor that down-weights the contribution to the final loss of samples that are already well classified, thus letting the network focus on harder-to-solve instances. Its formula is expressed as:

$$FL(p, y) = \begin{cases} -\alpha(1-p)^\gamma \log(p) & \text{if } y = 1 \\ -(1-\alpha)p^\gamma \log(1-p) & \text{otherwise} \end{cases} \tag{3}$$

where p is the classification score and α and γ are hyperparameters that we need to tune.

Finally, the `varifocal loss` [11] is an improvement over the standard `focal loss` as it treats positive and negative examples asymmetrically: this means that the loss for the positive samples doesn't decrease by the same amount as the one for the negative samples. Basically, if a positive prediction has an high `IoU` with the ground truth, its contribution to the loss will be relatively big. This lets us focus the train on the high-quality positive examples that are important to get an high mAP. Finally, to balance the loss of positive examples with respect to that of the negative examples, a scaling factor α is added. The formula to compute this metric can be found in Equation 4.

$$VFL(p, q) = \begin{cases} -q(q\log(p)) + (1-q)\log(1-p)) & q > 0 \\ -\alpha p^\gamma \log(1-p) & q = 0 \end{cases} \tag{4}$$

with α and γ that are tunable hyperparameters and q that is the `IoU` score between ground truth and predicted bounding box. Finally, note that in the original paper p is the predicted `IoU` aware classification score (IACS); however, in our implementation – due to the output format of the YOLOv4 predictions – we simply used the class scores.

3.3. Anchors

YOLOv4 also comes with 9 pre-defined anchor boxes, three per prediction scale. During our experiments, we first used these standard anchors and simply re-scaled them in order to fit our input image's size of 128x128, as they were originally handcrafted for an higher resolution figure (416x416). However, seeing as the objects in our dataset all had very similar small shapes, we also tried to generate custom anchor boxes for them. For this purpose, we used a script adapted from the original YOLOv4 repository, which analyzes the height/width ratios of the bounding boxes in our dataset and makes use of the K-Means clustering algorithm in order to retrieve an optimal number of centroids/anchors. Executing this script on our training set produced a list of 10 anchors, which we assigned to the three scales according to the following criterion: the final layer at scale 16x16 got all the anchors with width smaller than 30, the final layer at scale 8x8 got all the anchors with width smaller than 60 and the last output layer got all the remaining anchors. Using this approach we obtained 7 prior boxes for the layer with feature map 16x16, 2 for the layer with map 8x8 and 1 for the remaining layer.

3.4. Results

During the train phase, we tried different combinations of loss functions and hyperparameters, in order to derive a best possible model; however, regardless of the framework we used, we always started from the original YOLOv4 weights trained on the COCO dataset, thus performing transfer learning.

In order to correctly interpret and analyze the plots we will show in this Section, it's important to note that the orange line always represents the train loss, while the blue one always represents the validation metric.

The first choice we had to make was between using the standard YOLOv4 anchors or our own custom ones: the results of these first two trainings can be found in Figure 5. Note that the only real change between them is in the choice of anchor and anchor masks, as they were both trained with `DIoU`, `focal loss` with $\gamma = 4.3$ and without performing any kind of data augmentation. As we can see from the plots, in our precise case, sticking to the standard YOLOv4 anchors proved to be beneficial, thus we kept using them for all our other trials. Also note that the first train ran for

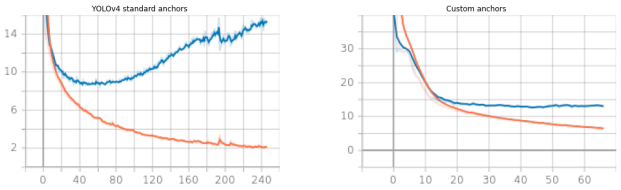

Figure 5: YOLOv4 trained on both standard and custom anchors.

a large number of epochs, because we did not use any form of early stopping. In the subsequent runs we indeed added this callback to prevent unnecessary train after the network starts overfitting.

Next, we also had to choose an optimal loss function for the class and confidence scores: we hypothesized that the `focal loss` could be a good choice, as it's reported to work well on unbalanced datasets, but we still tried to train the model also with `cross-entropy` and `varifocal loss`. The results we obtained are shown in Figure 6. We can see that – indeed – the `focal loss` was the best metric for our application, however, we believe that also the `varifocal loss` could have reached good results, if we'd be able to use it on the IACS instead of simply on the class scores. Still, all our experiments from now on will use `focal loss` as the designed metric for class and confidence during train.

Another choice we had to make was related to the data augmentation: as we did not have a huge amount of images to train on (only about 1600 pictures, with a great class imbalance), an augmentation step could prove to be extremely

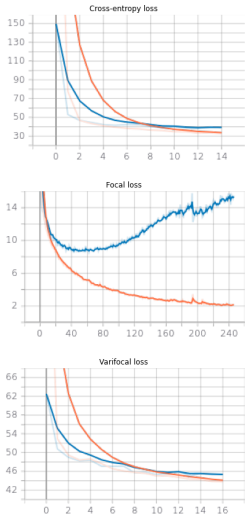

Figure 6: YOLOv4 trained with different losses for classes and confidence.

beneficial, thus we tried first to randomly rotate and flip the images, then to randomly change the pictures' brightness and finally to combine the two previous approaches. The results we obtained can be found in image 7. As we can clearly see, the flip and rotation method is able to prevent overfitting; however, using it, we were not able to drive the loss down below a certain threshold (around 14), thus getting stuck in a plateau. A similar problem is also encountered when we apply both types of augmentation. Then, our preferred augmentation method, which we'll use in the rest of the experiments, is the random brightness adjustment, which doesn't prevent overfitting quite as well as the other two methods but also doesn't cause the loss function to plateau.

Yet another point of experimentation was the loss function for the bounding box regression task: as anticipated in Section 3.2.1, we tried various options: the classic `L2` loss, `CIoU`, `DIoU` and `GIoU`. The results of the corresponding training sessions can be found in Figure 8: it's quite immediate to notice that the function which works best for our problem is the `DIoU`.

Finally, we also tried to change the value of the γ parameter for the focal loss, which regulates the weight given to well-classified examples (the higher the γ, the lower their weight is in the final loss value). In particular we tried two different values for this hyperparameter, 2.0 and 4.3: the results we obtained when training are shown in Figure 9.

We can thus conclude that the best model we were able to produce used the `DIoU` loss for box regression, the `focal loss` with $\gamma = 4.3$ for both class and confidence score and a simple data augmentation technique which randomly changes the image's brightness. With one such a model we

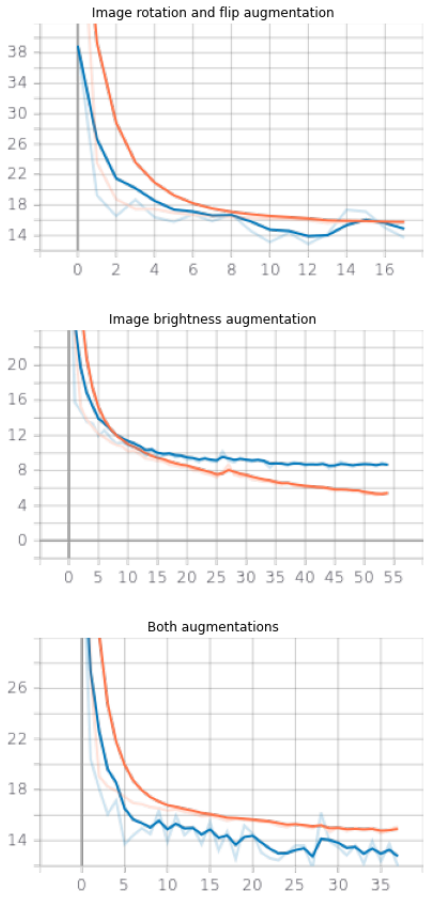

Figure 7: YOLOv4 results with different data augmentation strategies.

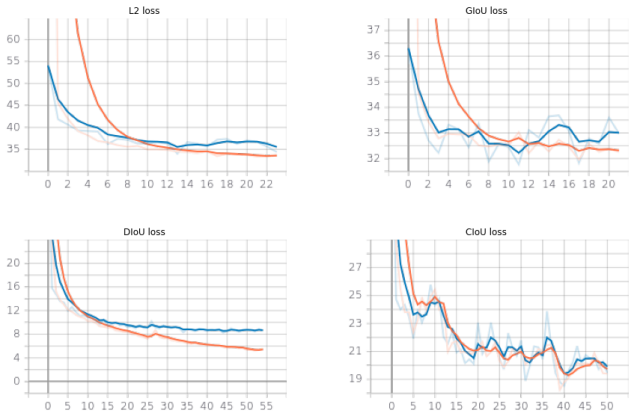

Figure 8: YOLOv4 results with different box regression losses.

were able to reach a global loss score on the validation set of 8.5, a location loss of 7.4, a confidence loss of 0.52 and a class loss of 0.59.

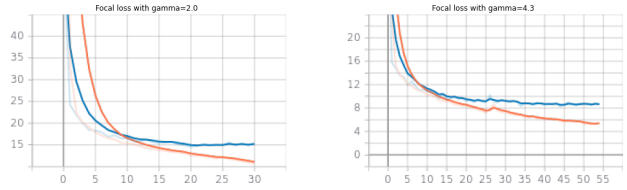

Figure 9: YOLOv4 results with different values for the γ hyperparameter.

4. Image Segmentation Approach

Seeing as the object detection approach did not meet our expectations, we decided to look at the problem from another perspective and tried to implement an image segmentation deep neural network. Among the many possible architectures, we choose to develop the U-Net [8], a convolutional neural network originally intended to be used for biomedical image segmentation. Its complete structure can be found in Figure 10: we can easily see that it consists of two connected paths, a contractive one, which acts as a sort of encoder, and an expansive one, which acts as a decoder.

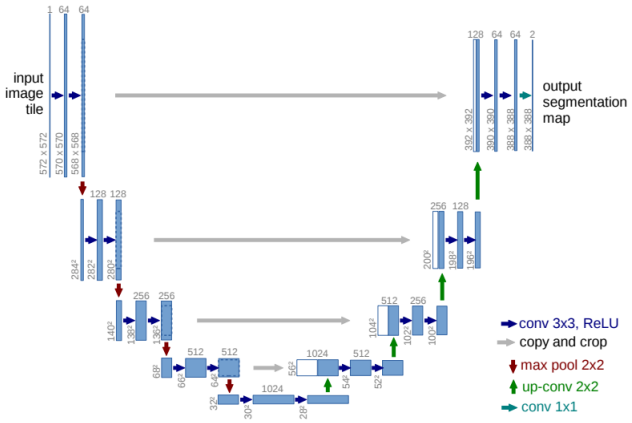

Figure 10: U-net architecture (example for 32x32 pixels in the lowest resolution)

The contractive part consists of 5 steps, each one composed of two unpadded `Conv2D` with a 3x3 kernel size and ReLU activation followed by one `MaxPooling2D` for the downsampling of the input image. The number of channels for the initial convolutions is 64 and they double at every step. Each step in the expansive path instead is composed of a `Conv2DTranspose` with kernel size 2x2 and strides 2, that upsamples the input feature map and halves the number of channels, followed by a concatenation with the corresponding feature map from the contracting path, and two 3x3 convolutions with ReLU activation function. The final layer, which produces the output segmentation mask, is

given by a 1x1 convolution with a number of filters equal to the amount of classes to detect.

Finally, note that we also implemented a smaller variation of this network, where the encoder and the decoder both have one less step with respect to the original structure: this was done in order to reach a final feature map of 16x16 after the contractive layers instead of a smaller one of 8x8. In the rest of this paper, we'll refer to this smaller network as Tiny U-Net.

4.1. Loss Function

Since this network aims to solve a multiclass classification problem, during training we use the classic loss function for this type of task, which is the `categorical cross-entropy`. In particular, we actually employ a variation of this metric, the `sparse categorical cross-entropy`, because our labels are given as scalar integers and not as vectors of scores (i.e.: they are not one-hot encoded).

Finally, during training we also keep track of the `accuracy` score of our network, in order to get a complete idea of the performance of our model.

4.2. Results

We trained both the classic U-Net model and our custom Tiny U-Net in two variations, which differ from each other only in the use (or not) of the class weights for loss and metrics computation. The results obtained with these two approaches for the classic U-Net model are shown in Figures 11 and 12, respectively, while those for the Tiny U-Net model can be found in Figures 13 and 14.

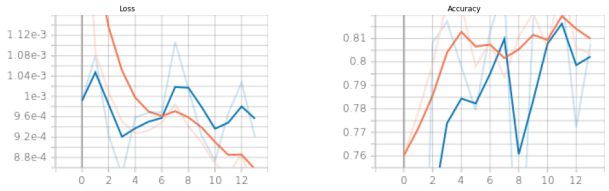

Figure 11: U-net results using class weights

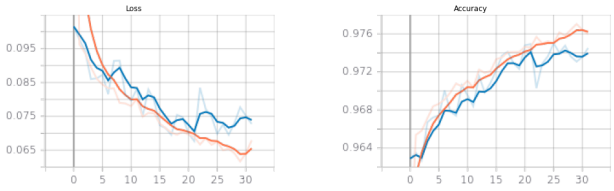

Figure 12: U-net results without using class weights

Obviously, our results show a greater accuracy when the weights are not used, as we then give equal significance

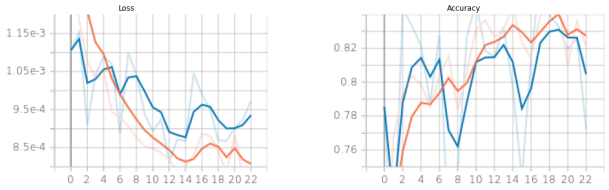

Figure 13: Tint U-net results using class weights

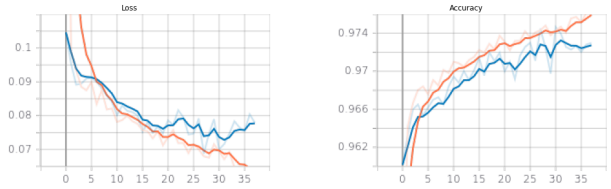

Figure 14: Tiny U-net results without using class weights

to all the classes, even if two of them (namely, the background and the Star-Forming Galaxies) are much easier to classify than the other two and also much more present in the training data. Thus, we think that the more accurate results should be those where we took class weights into consideration.

However, what we saw when examining the predictions, points into a different direction: the masks we got evaluating the networks on the test data were much worse when we trained using weights for the classes. Indeed, classes 0.0 and 1.0 get overclassified a lot, while the background often gets misclassified as an SFG. This is probably due to the weights we ended up choosing for the classes, which may have been too extreme. Given this premise, among our trained modes, the best one is the Tiny U-Net trained without the class weights, which reaches a loss of 0.075 and an accuracy of about 97.3% on the validation set. A mask predicted by this model on the test set, along with the corresponding image and ground truth label can be found in Figure 15.

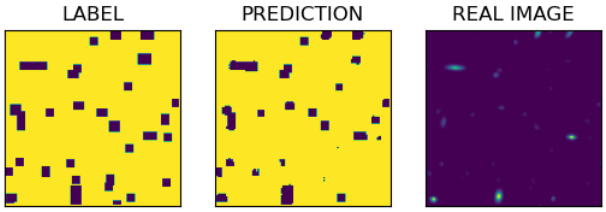

Figure 15: Example of Tiny U-Net prediction

5. Conclusions and Future Work

Between the two approaches we tried, the best one ended up being the image segmentation, which produces quite im-

pressive results.

Overall, a great challenge we faced when working with this data was related to the difficulty of correctly preprocessing radio images and to the great imbalance of the provided classes. Indeed, even our more successful model struggles when it comes to classifying SS-AGNs and FS-AGNs, which are very much minority classes. Regarding possible future improvements to this work, we think it might be interesting to try using different loss functions for the image segmentation method; for the object detection task, we think that implementing in the network some of the changes found in a recent work [10] dealing with a similar problem could prove to be beneficial. Also, other trials could be done using different image sizes in order to increase resolution or changing the input pipeline.

References

[1] FITS Documentation Page. 1

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020. 1, 2, 4

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *Lecture Notes in Computer Science*, page 346–361, 2014. 4

[4] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018. 5

[5] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation, 2018. 4

[6] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. 4

[7] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression, 2019. 5

[8] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. 1, 3, 7

[9] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of cnn, 2019. 4

[10] Xingzhu Wang, Jiyu Wei, Yang Liu, Jinhao Li, Zhen Zhang, Jianyu Chen, and Bin Jiang. Research on morphological detection of fr i and fr ii radio galaxies based on improved yolov5. *Universe*, 7(7), 2021. 2, 3, 9

[11] Haoyang Zhang, Ying Wang, Feras Dayoub, and Niko Sünderhauf. Varifocalnet: An iou-aware dense object detector, 2021. 5

[12] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression, 2019. 5