Understanding Clouds from Satellite Images

Mincato Emanuele
emanuele.mincato@studenti.unipd.it

Abstract

Shallow clouds play a huge role in determining the Earth's climate. They are also difficult to understand and to represent in climate models. Part of the reason is that shallow clouds are not just the result of the global circulation of the atmosphere. Rather, they have a life of their own and arrange themselves in a variety of patterns. For many of these patterns, the basic mechanisms behind them are poorly understood. After some discussion, scientists agreed that there are four common patterns and called them Sugar, Flower, Fish and Gravel. By classifying these types of cloud organization, researchers hope to improve our physical understanding of these clouds, which in turn will help us build better climate models, in order to have better prediction of climate change or forecasting weather update

1. Introduction

The aim of this project is to identify regions in satellite images that contain certain cloud formations, with names: Fish, Flower, Gravel, Sugar. For each image in the test set, we need to segment the regions of each cloud formation. This is a multiclass segmentation task where we have to find 4 different cloud patterns in the images. On the other hand, we make predictions for each pair of image and label separately, so this could be treated as 4 binary segmentation tasks. At each pixel in fact could correspond more than one label so, at the end, we consider each image four time to see if there is one particular cloud formation and produce the relative mask. Also from Kaggle rules we know that each image has at least one cloud formation, and can possibly contain up to all four. The project is divided into two main parts. In the first one, I implemented a U-Net architecture from scratch and tried to find the best combination of hyperparameters to get the best results. Instead, in the second part I compare the U-Net architecture built from scratch with more complicated and already implemented networks, in order to see if they can achieve better results.

2. Related Work

This task is a Kaggle competition, so this implies that there are many teams that have tried to solve it. At the beginning, to try to understand better the problem, I looked at the project of the team with the highest score. I found out that most of the team used very high level library, like fastai, and basically they build,train and compute the model with few line of code. This is a problem, especially for me who am not used to using pytorch, because the system is more like a black box where it is very difficult to fully understand what each command is doing. For this reason I decided to take a different approach and try to improve the result of the notebooks where most of the code is implemented inside it.

3. Dataset

First of all, in order to achieve good results, we need to understand what kind of images we will work with. The images were downloaded from NASA Worldview. Three regions, spanning 21 degrees longitude and 14 degrees latitude, were chosen. The true-color images were taken from two polar-orbiting satellites, TERRA and AQUA, each of which pass a specific region once a day. Due to the small footprint of the imager (MODIS) on board these satellites, an image might be stitched together from two orbits. The remaining area, which has not been covered by two succeeding orbits, is marked black. Below are reported 15 random images from the training set with their label.

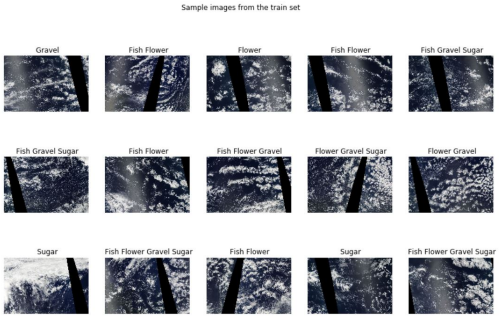

Figure 1. Sample images with their label

As said previously there are four possible type of clouds

(Fish, Flower, Gravel, Sugar) and are described as follows.

- Sugar: Dustin of very fine clouds, little evidence of self-organization
- Flower: Large-scale stratiform cloud features appearing in bouquets, well separated from each other.
- Fish: Large-scale skeletal networks of clouds separated from other clouds forms.
- Gravel: Meso-beta lines or arcs defining randomly interacting cells with intermediate granularity.

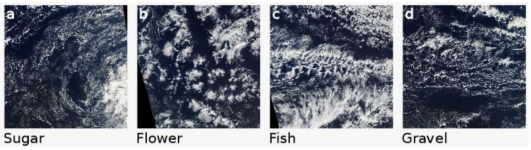

Figure 2.

A team of 68 scientists identified areas of cloud patterns in each image, and each images was labeled by approximately 3 different scientists. Ground truth was determined by the union of the areas marked by all labelers for that image.

3.1. Mask

At each images correspond four different masks, one for each type of cloud. In order to reduce the dimension of input and submission file, Kaggle uses run-length encoding on the pixel values. In practice, instead of having an exhaustive list of indexes for segmentation, a list of value pairs is used that contain a starting position and the length of consecutive pixels in the mask. E.g. '1 5' implies starting at pixel 1 and running a total of 5 pixels. The pixels are numbered from top to bottom, then left to right. If there is no area of a certain cloud type for an image, the corresponding EncodedPixels is left blank. To understand better we have reported below, in Figure 3, a random image with the relative masks. The first thing we can notice is that the masks are quite large and can overlap. Another important point is that masks can have apparently empty areas, in other words masks do not follow the shape of the clouds but roughly define the area with the same type of patterns.

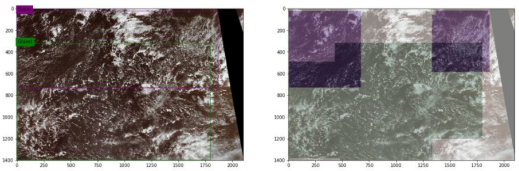

Figure 3. Bounding box and relative mask for a random image

3.2. Exploratory data analysis

To better understand the dataset we are working with, we can look at the training set in detail. There are in total 5546 images of the dimension of 1400 x 2100 px. In the barplots below the frequency of the number of clouds for each image is shown in blue while the frequency of each type of cloud is shown in red. It seems that most of the time there are 2-3 types of cloud formation in one image, while 4 types of cloud formation in one image is very rare. Furthermore, the data looks very evenly distributed for all four types of cloud formation. It is important to highlight the fact that each image has at least one label. I personally think that this choice does not reflect a real life scenario. In fact is very difficult to find a training set where all the data are useful for the prediction, there could be corrupted file or useless images, and also this makes the model less able to generalize in case of empty masks (images without these types of clouds).

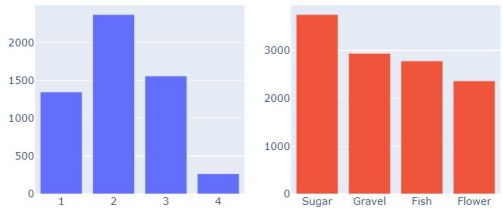

Figure 4.

3.3. Data augmentation

Certain augmentation techniques were applied to the dataset. Augmentation is a process to artificially expand the size of the dataset by creating modified data which improves the performance of the model to generalise. To augment the data I decided to use the Albumentations library. This library efficiently implements an abundant variations in image transformation that are performance optimized. I used the following image augmentation: HorizontalFlip, VerticalFlip, ElasticTransform, RandomRotate90. Below are reported all the possible augmentation for a single image and its masks (for space problems it is not reported the RandomRotate90 augmentation). Finally I normalized the image data using the mean and standard deviation of ImagNet, mean=(0.485, 0.456, 0.406) and std=(0.229, 0.224, 0.225). Data normalization is an important step which ensures that each input parameter (pixel, in this case) has a similar data distribution. This makes convergence faster while training the network.

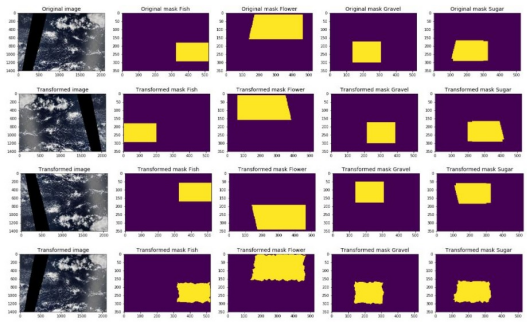

Figure 5. Different types of augmentation of the images

3.4. Data preprocessing

Finally as last preprocessing step I decided to resize all the images, and the relative masks, to 350x525 px. I decided to do this because the original size of the images is 1400 x 2100 px, but the expected masks should be 350 x 525. So, if the dataset is resized beforehand, there is no need to resize it repeatedly for each epoch. Basically this operation allows us to save a lot of time because it avoids doing the same operation several times.

4. Methods

There are many traditional way to solve a segmentation task, like region-based approaches or pixel-based clustering, but I decided to focus to a particular architecture, the U-Net. I chose this approach for two main reasons. The first is because it is easy to implement completely from scratch and the second is that achieve good segmentation results even if is trained with few images.

4.1. Architecture

I decided to implement from scratch the original UNET, that was developed by Olaf Ronneberger et al. [1] for Bio Medical Image Segmentation.

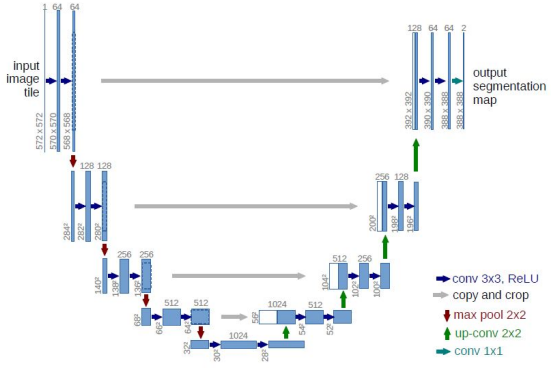

Figure 6. Unet Architecture developed by Olaf Ronneberger et al.

The network is an end-to-end fully convolutional network (FCN), i.e. it only contains Convolutional layers and does not contain any Dense layer because of which it can accept image of any size.

The architecture is divided in two main paths. The first one is a contraction path (also called encoder) which is used to capture the context in the image. The encoder is a traditional stack of convolutional and max pooling layers. The second path instead is a symmetric expanding path (also called decoder) which is used to enable precise localization using transposed convolutions. In detail the encoder consists on repeatedly application of two convolution layers, each followed by a ReLU activation function, and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. On the other hand, each step in the decoder consists of an upsampling of the feature map done by a 2x2 convolution ('transpose convolution'), that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two convolution layers, each followed by a ReLU. In few words a transpose convolution is a technique to perform up sampling of an image. The difference between other methods, like Nearest neighbor interpolation or Bi-linear interpolation, is that transpose convolution has learnable parameters and does not use a predefined interpolation method, in other words the networks learns itself how to up-sample optimally the features map.

At the final layer a 1x1 convolution is used to map each 64- component feature vector to the desired number of classes, in this case 4.

4.2. Optimizer

As optimizer I decided to use Rectified Adam, or RAdam. It is a variant of the Adam stochastic optimizer that introduces a term to rectify the variance of the adaptive learning rate. It seeks to tackle the bad convergence problem suffered by Adam. Using this methods we have a decaying learning rate throught learning.

4.3. Metric

The evaluation metric used in this paper is the Dice coefficient. It was applied to compare the pixel-wise agreement within a predicted segmentation and corresponding ground truth, using the following equation:

$$Dice = 2 * \frac{|X \cap Y|}{|X| + |Y|}$$

Where X is the predicted set of pixels and Y is the ground truth. The Dice coefficient is defined to be 1 when both X and Y are empty. The final score provides the mean of Dice coefficients for each (Image, Label) pair in the test data.

This metric is similar to the Intersection over Union (IoU) that we have seen during the course.

$$IoU = \frac{target \cap prediction}{target \cup prediction}$$

4.4. Alternative architecture

When using only a U-Net architecture the predictions tend to lack of fine detail, to help address this problem we can try to use more complex networks. The main properties of the architecture will remain the same but now we can try to change the encoder part with more complex model, like ResNet or EfficentNet, that can be pre-trained from ImageNet. Using pre-trained model could help because the learning does not start from random weights, like the previous case where the architecture has been implemented from scratch, but has a starting knowledge of the kind of features that need to be detected and could improve it through learning. If we decided to implement, for example, a ResNet as encoder, we have to replace the convolutions in the U-Net on each level with a ResBlock. Instead too built the decoder we can use the fastai library that automatically implement the decoder given the encoder architecture or otherwise we can build it manually, but is important to implement correctly the cross connections between encoder and decoder.

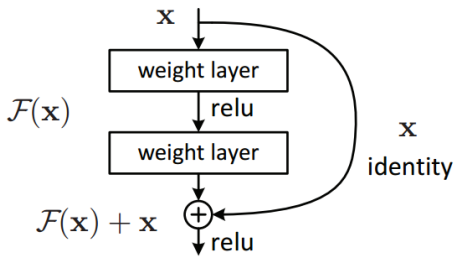

Figure 7. A ResBlock within a ResNet.

The advantage of using a Resblobk is that there is a skip connection between the input and output of the block. So now along with the long skip connection between every level of contracting and expanding paths, we have local skip connection between convolutions on each level. Skip connection helps in getting a smooth loss curve and also helps to avoid gradient disappearance and explosion, this means that is possible to construct deeper convolutional neural networks in a more efficient way.

5. Experiments

In this part of the project I focused more on finding the best parameters for the architecture implemented from scratch. I used a *greedy* approach in fact, since the training phase took a long time, I decided to find the best hyperparameter by training the network for a few epochs and then training the final model for a larger number of them. To find the best hyperparameters I need a validation set, so I split the dataset into ten folds and used nine for training and one for validation. To divide the dataset I used a 'StratifiedKFold' method in order to preserve the percentage of samples in each class.

5.1. Architecture

First of all I tried to modify the architecture of the UNet model. I built a shallow and a deeper version of it. In the first case I removed the last part so the input images will be represented with 512 channels, instead of 1024, in the *'bottom part'* of the network. In the deeper version I add two convolutional layers and a max-pooling layer in order to have a final representation with 2048 channels and a higher compression in the size of the images. From the table below we can notice that the original architecture achieve better results both in term of validation loss and Dice score.

Architecture	Val loss	Dice	Time (min)
Shallow	0.957	0.424	6.15
Normal	0.885	0.462	6.56
Deeper	0.926	0.444	7.49

Table 1.

5.2. Augmentation

As stated earlier augmentation helps the models to be more robust and makes it able to generalize better. Nevertheless we need to decide how much augmentation apply to our data. To do that we need to find the best value for the *p* parameter. This parameter controls the probability of applying the augmentation to a particular image. In the table below I reported the results considering different value of p.

p value	Train loss	Val loss	Dice
0	0.870	0.838	0.475
0.2	0.901	0.885	0.462
0.5	0.907	0.869	0.469
1	0.889	1.207	0.382

Table 2. Training for 8 epochs

This seems pretty odd. In fact the results suggest that not apply any type of augmentation, p=0, will lead to better results. In general this is not true because augmentation of the data helps to generalize better. To see if this is really the case and is not a coincidence, due to the fact I train the model for a few number of epochs, I decided to retrain the networks with more epochs, in this case 12.

p value	Train loss	Val loss	Dice
0	0.840	0.875	0.481
0.2	0.875	0.866	0.488
0.5	0.888	0.856	0.491

Table 3. Training for 12 epochs

From the table above we can see that with a p value of 0.2 or 0.5 the model achieve better results than not apply any augmentation. Finally I decided to use the value of 0.5 because is slightly better in term of Dice score compare to 0.2.

5.3. Kernel size

Another important thing to adjust is the kernel size of the convolution layers, which is basically the size of the filter. A larger filter reduces the size of the input image more than a smaller one and so, in order to preserve the dimension, we need to increase the padding value. Another thing is that using larger filters means that the model has to learn more parameters. This could be a problem because the time required for the training phase increases a lot if the number of parameters to be learned starts to increase too much. We could notice this problem from the table below where is reported the time required to train the model for one epoch at different kernel size. In the case of a kernel size of 7 is necessary over half an hour for just one epoch. Such a time is unacceptable because it does not allow the model to be trained for a greater number of epochs, needed too much time. For this reason I decided to use a kernel size of 3.

Kernel size	Epochs	Val loss	Time (min)
3	6	0.863	6.56
5	4	0.924	16.34
7	1	1.024	37.36

Table 4.

5.4. Learning rate

Finally I decided to find the best learning rate for the optimizer (RAdam). I tried three possible values : 0.05, 0.005, 0.0005. From the plot below we can notice that a learning rate of 0.0005, corresponding to the yellow lines, leads to better results both in term of validation loss and Dice score.

5.5. Final model

Before training the final model with the relative hyperparameter is necessary to tune few other parameters. To understand which ones we need to remember what the output of our network is. The task is to predict four masks

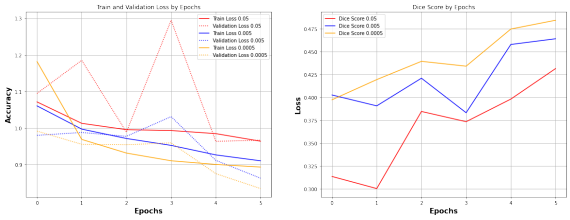

Figure 8.

for each image, more specifically we obtain for each pixel a value between 0 to 1, where 0 represent no clouds and 1 represent clouds. To decide whether a pixel must be 0 or 1, the output value must be below or above a certain threshold, usually 0.5. The value of this threshold could be considered as an hyperparameter in fact it is fixed, not learned from the network, and based on that we will obtain different masks. We can also consider another parameter *min.size* that could help to achieve better results. This parameter represent the minimum possible size for a mask. If the total number of pixels in the predicted mask is lower than the min.size, then the mask is forced to be empty, if it is higher, then the mask is unchanged. This helps to clean the output in the case the masks are too small, for example if the model predict as clouds a very small portion of pixel. Since each mask for each type of cloud is produced independently we can find independently the best values of threshold and *min.size* for each type of clouds. To find these parameter we don't need to train the network again every time but we just need to train once and try difference combination, with a grid search, in the prediction phase. Below in the table are reported the best combination of these two parameter for each type of clouds.

	threshold	min.size	dice
Fish	0.40	30000	0.595
Flower	0.55	20000	0.730
Gravel	0.60	20000	0.622
Sugar	0.65	10000	0.588

Finally, when we have fine-tuned all the parameters, we can predict the masks. I trained the final model for 25 epochs, about 3 hours. The final model achieve a Validation loss of 0.780 and a Dice score of 0.522 in the validation set. Below are reported the original, raw and post-processed mask for a random image in the validation set. The predicted masks are quite similar to the original ones. We can also note that in the post processed masks the Fish and Flower masks are removed, this is due to the fact that the dimensions of these masks are smaller than their respective *min.size* parameter.

5.6. Results

Finally I can use the trained model to predict the masks in test set. The model that I built achieve a Public score of

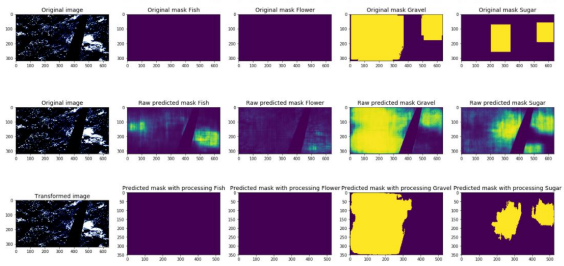

Figure 9.

0.63780. I was able to increase the results of the 'starting' notebook of around 1%, in fact the notebook I used as a reference [2] achieve a score of 0.62963. This a good improvement of the results especially considering the fact that the best team got a score of 0.67175.

5.7. Alternative model

In order to have a quick comparison I report the works of two other team. The first one is the work of Ryches [3] that used 'Resnet18' as encoder and achieved a Public score of 0.64927. The second project is made by Tashin and Noor Hossain [4]; in this case they used different version of 'EfficientNet' as encoder. In the end they managed to achieve a public score of 0.6650. They both used architecture pretrained with ImageNet and they both achieved better results than those obtained using the original UNet architecture. This suggest that using more complicated model as encoder could improve the quality of the segmentation.

6. Conclusion

In the end we can conclude that using a Unet architecture is a good strategy to solve segmentation task. It allows to reach good results even if the model is built from scratch. Moreover if a strong augmentation of the data is applied it does not need huge dataset to be trained, in fact I trained it from zero with just 5546 images. If more complex model for the encoder part are used it could achieve better results, but in this case there is not a huge difference around 2%. This may be due to the fact that the training dataset, composed of photos only of clouds in the ocean, is very different from the ImageNet dataset and so, have pretrained model on it, does not improved so much the training phase.

Future work should definitely be to directly implement a UNet with more complex encoder architectures and see, with the right fine tuning of the hyperparameter, if I am able to improve the results further.

References

[1] P. Fischer O. Ronneberger and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[2] D. Raut. Kaggle notebook: Image segmentation from scratch in pytorch, 2019.

[3] Ryches. Kaggle notebook: Turbo charging andrew's pytorch. 2019.

[4] N. H. Nuri Sabab T. Ahmed. Classification and understanding of cloud structures via satellite images with efficientunet. 2019.