# spotGEO Challenge Contribution

Alexander Merdian-Tarko
August 2020

## 1. Introduction

Although I did not manage to submit a proper, competitive solution in the end, I would still like to share my thoughts on and my experience with the challenge during the last couple of weeks. The provided starter kernel and explanations on the competition website (in particular the section on scoring) were good starting points to get familiar with the data and the problem at hand. Throughout my work on the challenge I often came back to the starter kernel and the competition documentation to question my understanding at the time and improve my understanding of specific details. In the following I would like to share what challenges I faced, the approaches I tried and some of the many things I learned while working on the challenge.

## 2. Approaches

After having gained a basic understanding of the challenge and its goals, I tried to use the starter kernel to train a basic random forest classifier on ~80% of the training data (following the pixel-wise classification approach with the provided post-processing functions in the starter kernel) and classify the entire test set, consisting of 5.120 sequences à 5 frames, i.e. 25.600 images in total. While training the random forest classifier was fast and fairly straightforward, and the performance on my validation set decent, I quickly realized that classifying the entire test set on my local machine would take literally dozens of hours (given that it only has 4 CPUs).

My idea was then to access freely available compute on Kaggle or Colab. Since I have previous experience with Kaggle, but not Colab, I decided to go with Kaggle. There appear to be no major differences regarding the available computational resources and run times anyway (see here and here). I used the Kaggle API to create a private dataset and upload the training and test images along with the training annotation file. Training the basic random forest classifier from earlier using a Kaggle kernel was again fairly straightforward, but in order to classify the entire test set I had to use 2 different kernels: the first would classify the first half of test sequences (1 to 2560) and the second would classify the second half of test sequences (2561 to 5120). This is because Kaggle kernels also have only 4 CPUs (though more powerful than the ones in my local machine) and a run time limited to 9 hours. Thus, I had to split the classification process in the explained manor to stay within the run time limit, which worked fine, and I was able to make a valid submission after combining the predictions from the 2 classification kernels.

Although the performance of random forest on my validation set was decent, the performance on the test set, however, was poor. Tuning hyperparameters did not lead to a significant boost in performance. I thought random forest might be too simplistic of a model. Therefore, I tried to use more complex models. I trained and tuned an XGBoost classifier and neural nets with different numbers of hidden layers, using the sci-kit learn and Keras APIs. Using the Keras API allowed me to leverage GPU compute that Kaggle provides. There is a weekly quota of GPU time of about 30 to 40 hours per week per user. XGBoost and different neural nets achieved better performance on my validation set than random forest, but the performance on the test set remained poor. This puzzled me and I believe that there is something fundamental that I might have missed or not understood properly at this point. Besides, parallelizing the classification of the test set, for some reason, did not work with XGBoost and the neural nets. After doing research on StackOverflow and

GitHub it seems like there are some issues when using the Python multiprocessing library in combination with specific types of models and Jupyter notebooks in general. It was clear to me that I would not find a solution to this issue in a reasonable amount of time and so I decided to split the classification process across several kernels all of which would then classify different chunks of the test set each using a single CPU. This was a bit cumbersome, but worked fine. However, I had to be particularly aware of the time a model required for a single prediction in order to finish classifying a given chunk of the test set within the kernel run time limit of 9 hours. I ended up having an upper limit of about 10 seconds per prediction to satisfy the run time constraints.

Since using more complex models with the pixel-wise classification approach did not lead to improved performance on the test set, I decided to try an entirely different approach: object detection. I was not sure whether this would be feasible, particularly because the satellites in the images were so small and pre-trained models were usually trained on images very different from the ones used in the spotGEO challenge. To save time and get results as quickly as possible, I used the Python ImageAI library that allows one to work on computer vision problems quickly without too much setup. To try this approach I switched to Colab, particularly because the library's tutorial I used as a reference used Colab. Before I could apply an object detection model to the satellite images, I had to prepare the images and compute bounding boxes for the images that contained satellites. I experimented with the size of the bounding boxes (e.g. 7x7 and 21x21 pixels), but did not find a significant effect (probably there are more fundamental issues with this approach than the mere size of the bounding boxes). I used only images that contain satellites for training as including negative examples in training an object detection model with ImageAI is not straightforward (and perhaps not even necessary). To present the model a large variety of images containing satellites and limiting the amount of data to feed to the model at the same time, I used every other training sequence and only the first, third and fifth frame within a given sequence. I first tried to use a YOLOv3 model pre-trained on the COCO dataset as it is supposed to do predictions quickly. The Colab run time limit is 12 hours and one has also limited access to GPUs. Thus, I re-trained part of the pre-trained YOLOv3 model using a batch size of 8 for 20 epochs. To classify the entire test set I again had to split the process across 2 Colab instances. Unfortunately, the test set performance remained poor. I tried to train a YOLOv3 architecture from scratch again with a batch size of 8 for 20 epochs (plus 3 epochs at the beginning as a warm-up). This effort remained unsuccessful as well. I have a couple of suspicions why this might be the case. It could be that I would need more data and have to train the model for more epochs to achieve decent results. It could also be that I would have to customize the model architecture provided by ImageAI to fit better the specific characteristic of the satellite images (e.g. size of anchor boxes, disable automatic down-scaling of images to 224x224 pixels). Or perhaps this was just not the right approach.

## 3. Conclusion

The major challenges I faced during my work on the spotGEO challenge were, I believe, my lack of previous in-depth experience with image data, my lack of compute which had me bound to constraints of my local machine, Kaggle and Colab, and to some extent the fact that I was working full-time (I worked on the challenge before and after work and on weekends, but I don't think this actually was the biggest challenge). I also learned a lot during the challenge. I gained experience with a novel use case that I haven't encountered previously but was still able to apply my existing knowledge to work on the given problem. I improved my knowledge of working with image data (especially when I tried object detection). I used Colab and GPUs properly for the first time. I believe working on a team with more experienced people could also have possibly enhanced my experience and learning process. Overall, I enjoyed the challenge a lot, learned a thing or two and I'm looking forward to the next one.