DTU Space
National Space Institute

Computer Vision for Autonomous Aerial Navigation

Project report for a special course in association with Aerial Tools

Ask Espensønn Øren (s217109)

Kongens Lyngby 2022

DTU

DTU Space
The National Space Institute at the Technical University of Denmark
Technical University of Denmark

Elektrovej
Bygning 327
2800 Kongens Lyngby, Denmark
Phone +45 4525 9500
mga@space.dtu.dk
www.space.dtu.dk

Acknowledgements

First and foremost I would like to extend my thanks to Aerial Tools for entrusting me with this task, as well as for their endless positivity, curiosity and wish to help. I would also like to thank my supervisors Daniel Haugård Olesen and Per Knudsen for keeping me on the right track and passing me good references. Lastly thanks to Ashutosh Dhaka for giving me an overview of ArduPilot and MAVLink, and to Artur Coelho Fabrício Rodrigues for model debugging advice.

Contents

CHAPTER 1

Introduction

This report will concern the design, implementation and future work concerning a computer vision algorithm for autonomous aerial navigation. Specifically, it aims to create a model capable of identifying and triangulating the position of photovoltaic panels (PVs) given an input picture from the camera module. Implementation and training of the module will be done through the PyTorch machine-learning framework, the specifics of which will be discussed in chapter 3 & 4.

The project was done as a student project for the DTU startup Aerial Tools during the 13-week autumn semester of 2022. All code can be found on the github. Questions regarding the contents of this report, implementation or design decisions can be sent to the author Ask Espensønn Øren over email *ask.e.oren@gmail.com*.

CHAPTER 2

Dataset

In order to construct a deep-learning model capable of identifying the positions of PVs, one first needs a good data set to work with. For time-saving purposes, the data set should ideally already be *annotated*. Annotated here means that bounding boxes are supplied together with the imagery, however, the way they are supplied is of little consequence. The reason we need annotated data is that in order for the model to learn it needs to be given feedback on when it is doing things right or wrong. Annotation - or ground truth as it is often called - serve as this feedback mechanism. Simply put, whenever our model makes a guess, it can be told how close it is, which will allow it to modify its guess accordingly. Annotating data can be quite a tedious affair, which is why preferably the dataset should come pre-annotated.

While there is an abundance of available data sets from satellite imagery, the same cannot be said for images taken from drones, bi-copters or similar low-flying vehicles. Unfortunately, satellite imagery would not be appropriate for this implementation, as the resolution and size of the plants would appear far smaller and pixelated. Consequently, identifying singular PVs would be close to impossible. Seeing as this is imperative to the mission of Aerial Tools, the idea of using satellite imagery was quickly discarded.

A potentially useful dataset, however, was published in association with the paper "Using geospatial data for assessing energy security: Mapping small solar home systems using unmanned aerial vehicles" [Ren+22]. In this paper, the authors acknowledge the lack of similar (annotated) data. In fact, they proclaim their dataset to be the first publicly available of its kind. The dataset was gathered in Duke Forest and consists of multiple categories of images:

1. Stationary data at altitudes ranging from 30 - 120m
2. Frames clipped from videos taken while flying in normal(N) mode
3. Frames clipped from videos taken while flying in sports(S) mode

It is split such that there are 322 stationary images, 54 N-mode and 83 S-mode images. These already come split into training and test sets in case you want to recreate their studies. Annotations are given as binary images, giving a pairing of RGB-binary as seen below.

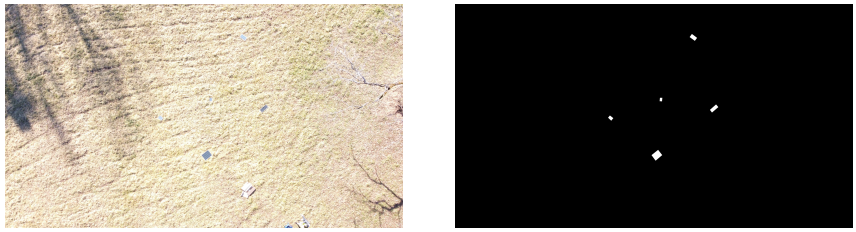


Figure 2.1: Example of a picture with its label from the dataset

While it is far closer to the ideal for our use case, it is not without flaws. One such flaw relates to the bias of the dataset. Considering all the images are taken from a top-down view, the network might not be able to recognise PVs taken from other angles, meaning that the camera for the drone on which it will be deployed will also have to have a similar setup. In the current iteration of the Aerial Tools drone, however, this is fortunately not an issue.

Perhaps the biggest issue relates to the size of each panel in the pictures. Most of the dataset uses an image resolution of either 2250x4000 or 3000x4000. Even with such a high resolution, the panels are quite small and difficult to spot. When scaled down, this could result in a large loss of important information, meaning that training the model might be both slower and more computationally heavy to account for this. Moreover, such small panels vastly differs from what a more realistic dataset would look like, for example, this picture, which was taken at Risø[1]:

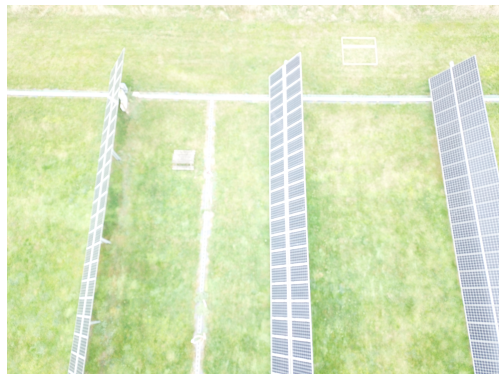


Figure 2.2: Realistic depiction of an image captured from Aerial Tools' drone

Another matter of concern pertains to clustering. As will be further discussed in chapter 4, the entire dataset consists of very spread-out PVs. While this makes the labelling and masking process easier, it does mean that the model might ultimately

[1]It should be noted that the picture was taken by an older drone, so the picture quality is not as representable as the content

not be very good at identifying large clusters of panels. I believe this to be perhaps the biggest issue with the dataset as part of the main objective will be to inspect large plants of panels. Consequently, future iterations of the model should consider looking into the acquisition of a clustered dataset to continue training.

A third potential problem is regarding the lighting. All footage was taken approximately at the same time of day, under similar conditions. This means that there's less variety in the environment for the model to train on. This might end up being an issue as if the model performs far worse during dimmer lighting it could heavily restrict when the drone could be used. It might be possible to remedy this however with some image pre-processing in-flight, which would perhaps be worth looking into for future iterations should this prove problematic.

CHAPTER 3

Model Selection

For model selection, the main consideration was pertaining to simply solving the task at hand. Specifically, in the typical trade-off of accuracy, speed, and required processing power, speed and accuracy were preferred. Speed, because in order to provide useful control signals the resulting system must support real-time, and accuracy to ensure the output is trustworthy for maneuvering. Moreover, the model should be suitable for image-based analysis and have an ample amount of supporting literature. This requirement was imposed due to time constraints in order to make implementation easier. Lastly, the resulting network should not be huge in size as it will be implemented on a small platform with limited space.

Image analysis is typically done through a convolutional neural network (CNN). In essence, a convolutional neural network attempts to mimic the way humans perceive the world around them. It does so by splitting an image into a multitude of smaller sub-images, before performing analysis on each one individually. At its core there are three primary layers to this process;

1. Convolution Layer: defines the step size between pixels and determines the size of sub-images.

2. Pooling Layer: computationally the same as the convolution layer, but instead of a weighted kernel calculating the outputs it simply selects either the highest pixel value found or the average of all pixels - depending on what kind of noise you have, as one the main jobs of the pooling layer is to remove image noise.

3. Fully-Connected Layer: akin to a standard neural network, it links sub-images and carries out classification.

Realistically, a complete CNN consists of many of the same types of layers stacked on top of each other, but to further exemplify consider this simple case of a 4x4x3 RGB image:

$$\mathrm{R} = \begin{pmatrix} 7 & 91 & \cdot & \cdot \\ 87 & 221 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \mathrm{G} = \begin{pmatrix} 99 & 15 & \cdot & \cdot \\ 51 & 114 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \mathrm{B} = \begin{pmatrix} 187 & 234 & \cdot & \cdot \\ 88 & 29 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

With the colour filters:

$$R = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, G = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, B = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Applying the filters would yield:

$$\begin{gathered} R = 0*8 + 1*91 + 1*87 + 0*221 = 178 \\ G = 0*99 + 1*15 + 1*51 + 1*114 = 180 \\ B = 0*187 + 0*234 + 1*88 + 1*29 = 117 \\ R + G + B = 475 \end{gathered}$$

With 475 being the first value in our new 3x3 matrix. Moving our filters across the channels one pixel at a time, and substituting the dotted values with arbitrary numbers could for example yield the following complete new 3x3 matrix:

$$PL = \begin{bmatrix} 475 & 567 & 621 \\ 627 & 515 & 472 \\ 279 & 567 & 567 \end{bmatrix}$$

The pooling layer will now attempt to further decrease the dimensionality by, for example, looking at all possible 2x2 matrices within the 3x3, and taking either the average or max values from them:

$$\begin{pmatrix} 475 & 567 & 621 \\ 627 & 515 & 472 \\ 279 & 567 & 567 \end{pmatrix} \rightarrow \begin{pmatrix} 475 & 567 \\ 627 & 515 \end{pmatrix} \xrightarrow{\max} \begin{pmatrix} 627 & 621 \\ 627 & 567 \end{pmatrix}$$

The example above is using the max operation for the pooling layer, with the blue sections being the selected parts of each matrix and where it ends up. Once the pooling layer is done, the 2x2 matrices are finally passed to the fully-connected layer, which associates them with a neuron in the neural network.

CNNs are however quite slow for large images. A much smarter approach is using a region-based convolutional neural network (R-CNN). R-CNNs introduce a pre-emptive set of region-proposals for the CNN to do classification on[Gad20]. This speeds up the process quite significantly as it means far fewer operations on large pictures. However, it has the drawback that it's a multi-staged model where each stage is independent. Independence here means it is not possible to fully train the network end-to-end. Moreover, the selective search algorithm used for region proposals is quite slow in itself. Lastly, and perhaps most importantly for this project; because each region is fed into the CNN independently, it is impossible to run in real-time.

Unfortunately, despite being able to improve the speed further, as was done with the Fast R-CNN[Gir15], Faster R-CNN[Ren+15], and Mask R-CNN[He+17], the R-CNN would simply require too much processing power to be able to guaranteed run in real-time.

Introducing real-time therefore means taking a completely different approach to image detection. The You-Only-Look-Once(YOLO) algorithm does exactly this. Rather than segmenting images into regions and calculating predictions based on the

highest scoring once, YOLO sees the entire image during both testing and training - and uses this to predict all bounding boxes at once[RF18]. As such, it implicitly encodes contextual data, which other methods are traditionally poor at. Moreover, with this approach, it is able to use a singular neural network for the entire image, as opposed to the staged approach of the R-CNN. This makes end-to-end training quite easy.

More concretely, YOLO starts off by splitting the image into an SxS grid. Then, it calculated the bounding boxes and confidence scores for each grid, the confidence score is how certain it is that there is an object within the box, as well as how accurate it believes the box to be. Despite its differences, however, both YOLO and R-CNN utilise the same cornerstone; the generalised layers of CNN discussed at the start of this section.

YOLO is not without fault, however, because while it is far faster than Fast R-CNN[RF18][1], it is also more prone to localization errors, meaning that the intersection-over-union (IoU) is lower than the R-CNNs. This could prove problematic when trying to predict smaller objects, but I am currently unsure whether this could be an issue for AerialTools. Another limitation of YOLO is that because of how it predicts objects, it can struggle with large clusters of small objects. One example here could be that in a plant of PVs, it might struggle to find individual ones, but should do fairly well at identifying the general grid structure.

A third consideration is the U-Net[RFB15]. U-Net was developed for biomedical purposes as a way to decrease the required size of training data while maintaining high accuracy and speed. It achieves this by manipulating and distorting the images in its training set, such that it may better learn variances and such. For our purpose, U-Net could represent a middle-ground between the Mask R-CNN and YOLO, as it sits in the middle of them both in terms of accuracy and speed[Lu+21]. Yet, U-Net is still not great for real-time and therefore does not solve the main issue presented by the Mask R-CNN, it simply comes closer to fulfilling it.

After conferring with Aerial Tools, a YOLO algorithm was chosen as it has decent results when it comes to object detection while being possible to run in real-time - and its shortcomings are believed not to be of major concern for our use case. Additionally, it has a good support network of tutorials and such, which is of great assistance. It should be noted, however, that in choosing YOLO we are sacrificing accuracy in favour of real-time capabilities. Thus, for future iterations, depending on hardware upgrades or other alterations it is perhaps worth checking back in on the R-CNN approach at a later time. Consequently, this report also includes a setup for training a Mask R-CNN on the dataset, but no test results will be provided.

[1] Note that this is for yolov3, the current iteration; yolov7 is even further ahead

CHAPTER 4

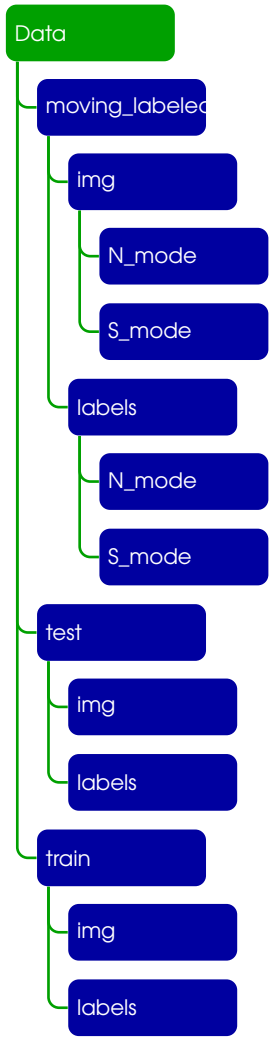
Pre-Processing the Dataset

4.1 Mask R-CNN

In order for the dataset to fit the input values expected by PyTorch some pre-processing is required. For this, a baseline from the example given in the TorchVision instance segmentation tutorial was used. In it, a few key points are discussed w.r.t the pre-processing step: creating a dataset class and applying transforms. This, however, assumes correct structure within the data - which was not the case initially for our dataset. Expressly, PyTorch datasets assume a division between the labels and images. An easy way to achieve this is through a folder structure in which labels and images are divided but correlated by being in the same order i.e. image number i corresponds to mask i. Combined with some extra steps connected to differences between the example dataset and ours, the pre-processing consists of the following:

1. Sort data and labels into corresponding folders
2. Create a dataset class
3. Label the binary mask
4. Apply transforms

For ease of implementation and reproducibility, let's have a look at each one in order. The sorting was achieved easily by noting that the dataset has different file types for images and labels. Images are all jpgs, while labels are pngs. As such, the only real step here was creating an appropriate folder structure to sort the images into. The one used here looks like this:



Sorting done, a dataset class was created, its code can be found in appendix A.1, however in short it is required to include a couple of things. Firstly, it must have an internal initialisation function in which it loads both the images and their corresponding masks. Additionally, all transformations are passed as arguments such that they may be applied individually to the images later.

Secondly, it should have a function that returns the length of the data set. Even if you do not use this yourself it is still a requirement as Pytorch's internal functions will call upon it during training and validation. Lastly, it needs a *getitem* function. The *getitem* functionality is the main part of the DataSet class. In fact, steps 3 & 4 of the previously mentioned pre-processing happen through this function. To label the binary image, it is first converted to grayscale using Pillow, and then into a NumPy

array. This step is simply to make the remaining process easier. Then, to differentiate between different instances of PVs the mask is labeled, the background removed, and then it is split into multiple sub-masks - one for each PV in the image. Finally, the masks are converted to be on the expected format (Channel, Depth, Height, Width), and the bounding boxes extracted from the labelled image.

Pytorch expects the output of the *getitem* function to be both the next transformed image, as well as a dictionary containing various pieces of information, such as; where the bounding boxes are found, all the labels and masks, area of the PV, etc. Thus, the final thing the function does is to create this dictionary, as well as loop through the transforms and apply them one at a time. It is worth noting that while our model does scaling on its own, some scaling during pre-processing was also required in order to avoid reducing away an entire dimension - consequently crashing the program.

As a final note on this subject, following the comments in the code, it proclaims that there is only one class. Class in this sense refers to the type of object to be labeled. In this case, as we are only looking to label PVs, the number of classes turns out to be one (or two counting the background, as is done later).

4.2 YOLOv7

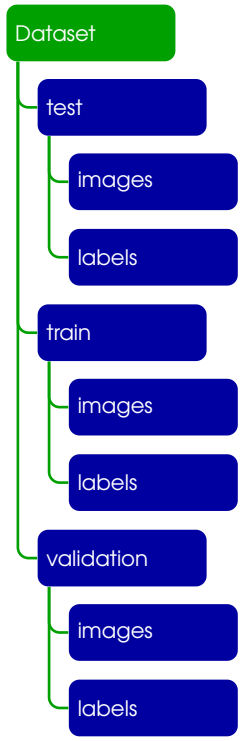
For YOLOv7, the setup is slightly different, following [Rat22]:

1. Split into test, train (and optionally, validation)
2. Extract bounding boxes from masks
3. Convert into a .txt file in the format <class num, x_center, y_center, width, height>
4. Sort into correct folder structure
5. Create a .yaml file containing the paths to images and labels
6. Create a .yaml file corresponding to the specific type of yolo model you want to use

When splitting into testing, training, and validation, as a rule of thumb, the training set should be approximately 4x as large as the test, and a bit over twice as large as the validation one.

Starting with the bounding boxes & text conversion, this is done using the script provided in appendix A.2. It starts with labeling, using this to extract the coordinates of each box. These are subsequently normalised to the range [0, 1] by dividing by the size of the image. Finally, the information is appended to a .txt file. This same algorithm is repeated for images pertaining to both the train and validation folders as well.

Once all the files were generated, they were sorted into the following folder structure:



Please note that in order to use the YOLO framework out-of-the-box, these folders should be named exactly as shown.

4.3 Setting up the Training Script

Google Colab was used to train the model using the following training script[1]. This section will function as a tutorial for how it works and which parts should be changed for further training.

The script starts with accessing the root of the Colab session, downloading the yolov7 framework and accessing the directory. This step is mostly done for ease of use for later paths and commands.

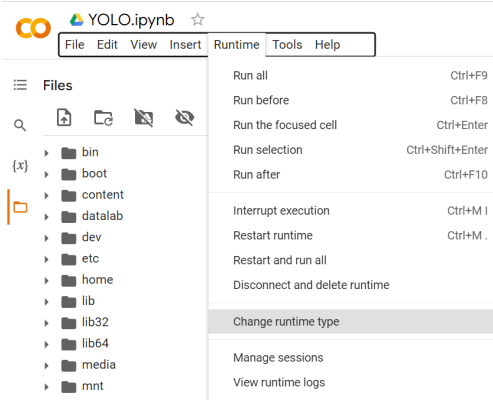
```
%cd ../..
%rm -rf sample_data
!git clone https://github.com/WongKinYiu/yolov7.git
%cd yolov7
```

[1]The access is only viewer as it would be better to copy the script into your own Colab for the sake of always having one functional backup.

Then, as I have my data saved on my drive - and would like to move results there after training - the script mounts my drive to the session. This step is voluntary if you do not have anything of interest on your drive, nor want to keep the final results there. Please note, however, Google Colab does not save generated folders, plots etc. between sessions. Thus, if you run a training session you should make sure to extract everything you need immediately after completion such as weights and plots (if you wish).

After mounting the drive, some other important setup is done. First, a new file *requirements_gpu.txt* is created, specifying which version of PyTorch and TorchVision should be downloaded. By default, the requirements file that yolov7 comes with has a version which does not support CUDA - the package used to train using a GPU - which would slow down the training process significantly. Thus, we install the correct version before acquiring the rest of the requirements. Then, we simply import our essential packages and check whether we are running on a GPU or not. Make sure that the output says something along the lines of:

If it says ”CPU” you might be using the wrong runtime environment. To change this, go to ”Runtime” $\longrightarrow$ ”Change runtime type” and select GPU.



With this, everything is ready for step 5 - creating the .yaml file with paths to images and labels. This is done quite simply with the command shown below:

```
%%writefile data/pv.yaml
train: ../../content/drive/MyDrive/Colab/YOLO/Data/train
val: ../../content/drive/MyDrive/Colab/YOLO/Data/validation
```

```
test: ../../content/drive/MyDrive/Colab/YOLO/Data/test

# Classes
nc: 1  # number of classes
names: ['solar panel']  # class names
```

The paths shown are the relative paths from the data directory to where the folders containing images and labels are, that is, test, train and validation are the folders shown in the folder structure for yolov7. For Mask R-CNN we had 2 classed; panels and background, yolo on the other hand does not require you to specify background as a class. Thus the final two lines we input just a single class with its respective name. Should you wish to train the network for multiple classes at the same time, this would be where to change it.

Yolov7 is pre-trained for a fair amount of classes, so the next step is to fetch the initial weights and build the model. For the model building, the setup provided is for the standard yolov7 provided by the creators - only the number of classes is changed to fit our case. You could also change the anchors, as more optimal versions are provided at the start of a training run, but it is not essential. There are however other potential models you could use depending on need, the tiny builds, for example, are significantly faster but less accurate. You can change this model file to fit your needs, but it is recommended to use one of the provided models - these can be found under ”yolov7/cfg/training/”.

With that, all 6 pre-processing steps are complete, which means the final thing to do is run the training algorithm.

```
!python train.py --epochs 60 --workers 4 --device 0 --batch-size 3
--data data/pv.yaml \
--img 1504 1504 --cfg cfg/training/yolov7_pv.yaml
--weights 'yolov7_training.pt' \
--name yolov7_pv_fixed_res --hyp data/hyp.scratch.custom.yaml
```

There are a lot of inputs here, but here is what each of them does:

- epochs: How many rounds will the training be going on - changing this parameter will make training take longer but potentially yield more accurate results.
- device: which device should be used, 0 is the gpu
- workers: How many cores should be used - essentially speeds up training by allowing multiple cores to work in parallel.
- batch-size: How many images will be loaded in memory at once - in combination with epochs tweaking this parameter will increase/decrease GPU memory requirements.
- data: path for the .yaml file containing the paths to images and labels

- img <num> <num>: Decides what the images will be resized to for training and validation respectively.
- cfg: path for the model .yaml
- weights: path to weights file
- name: name of the output folder in which all the data and weights will be saved in
- hyp: path for the file containing essential model parameters such as learning rates, momentum etc. Custom is its own setting meant as a generally solid baseline for custom datasets.

It should be noted that ideally, yolov7 should run with the previously mentioned image size of 640, i.e. ”–img 640 640”. However, that was not possible for the current dataset. Even still, for runs with other datasets, I would recommend increasing epochs and batch size while decreasing image size. What they should be changed to will depend fully on the dataset though.

CHAPTER 5

Results & Discussion

Apart from the weights themselves, the network supplies a variety of different statistics useful for measuring the performance of the network. This chapter will present these parameters and discuss both their general meaning and how they apply specifically to our network. The results after training for 60 epochs are shown in figure 5.1.

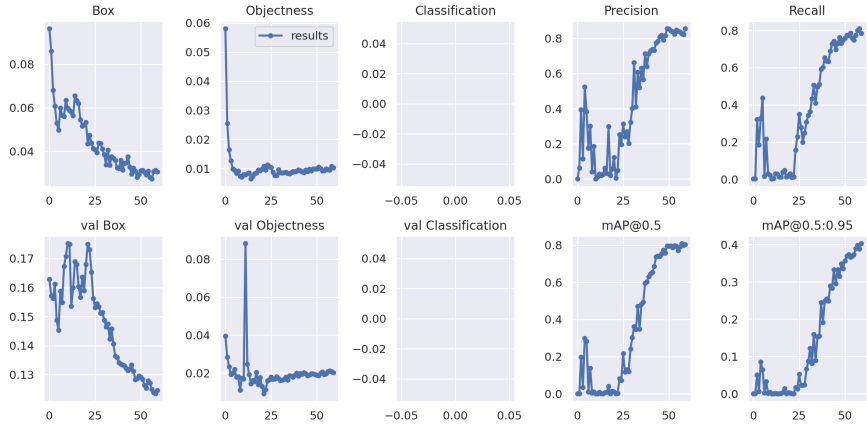


Figure 5.1: Training results for our dataset

Breaking this down - while omitting classification as this is only relevant in multi-class datasets - the most important statistics are precision and recall. Precision is defined as the ratio between true positives, that is, the network identified something correctly, and the overall amount of positives, that is, everything the network thought was an object. Mathematically it is thus defined as $\frac{TP}{TP+FP}$. Thus, we want the value to be as close to 1 as possible as that would mean the network always classifies correctly. Recall on the other hand is a measure of the ability of a network to correctly identify actual samples. So, given n amount of objects in a picture, how many of them were detected, i.e. $\frac{TP}{TP+FN}$. For both precision and recall the network reaches around 85% by the end of the training, which is very good considering both the size of the objects and the limitations posed by lowering batch size and epochs.

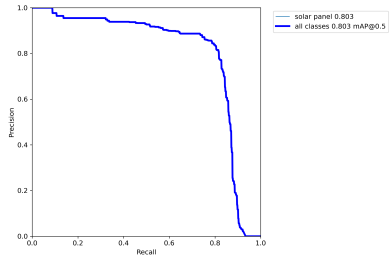
mAP stands for mean average precision, and is defined as the average IoU given

a lower limit for how high the IoU needs to be for it to be considered a match. mAP @ 0.5 refers to the minimum IoU being 0.5, while mAP @ 0.5:0.95 means the average precision given the range of IoU from 0.5 to 0.95 with a step size of 0.05. Obviously, we want both of these to be as high as possible, but above 80% for 0.5 and over 0.4 for the range is quite high.

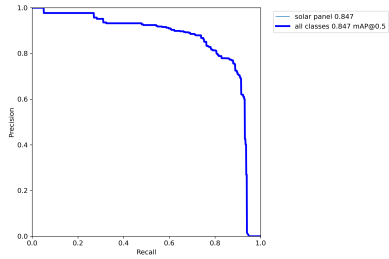
Objectness on the other hand refers to how well the model is able to identify the locations of objects. Specifically, it should increase as the distance between the proposed locations and the actual locations increases. As such, in an ideal scenario, the objectness score should tend towards zero. While it is low, for our model it is not quite zero. Moreover, we can note that it starts increasing slightly again towards the tail-end of the epochs. This could hint at the fact that ideally the model should have been trained for longer. Yet, overall the score is good enough to not warrant major concern. Finally, the box refers to the bounding boxes and how encompassing they are compared to their size. Basically, we want a smaller number as this would imply a tighter fit within the allocated boxes - which implies a smaller localization error.

It should be noted that the overall trend in the graphs is that while they are on the verge of flattening, they have not quite settled yet, as most still have an up/downwards slope. In general, this would again imply the need for either larger batch sizes or more training epochs. Both would be possible but might require an alternative platform to Colab as there is not enough memory allocated for a free user to increase batch sizes without shrinking image size. Moreover, increasing epochs could result in being thrown out prior to finishing as free users are on an internal, unknown timer, which was a recurring issue during the training of this model.

Over- or underfitting was also looked into as this could also affect how well the model generalises. One way of doing so is by comparing the Precision-Recall (PR) curves of the test and training set[McW+21]. Noteworthy differences between the two curves suggest overfitting, while graphs shifted to the right would mean underfitting.



((a)) PR curve for the training set



((b)) PR curve for the test set

Looking at the curves, however, there neither a prevalent right shift nor any big difference is observed between the test and training set. Yet, this does not decisively mean that there is no over or underfitting. In fact, despite being able to detect the solar panels within the test image with incredibly high accuracy given the size of the objects, as shown in figure 5.3.

Figure 5.3: Image with its corresponding prediction after training

Due to the problems with the dataset, such as the lack of variety within the images, as discussed in chapter 2 this result should be taken with a pinch of salt. This result proves that the network is able to find PVs in an environment similar to the one in the dataset, but it does not mean that a generalised model capable of identifying solar panels in other environments and under other conditions has been found. In fact, when applying the model on the image from Risø 2.2, it was unable to find any of the panels. Thus, the model is not generalised but rather overtrained for images of similar characters to the training set. Primarily I believe the reason behind the inability to locate the panels is the size of panels in the dataset as well as the colour. Colour especially should be important, as the dataset used for training has quite deep blue colours on the panels, while the dataset from Risø on the other hand has far more light reflection and glare, meaning that on colour alone the network could struggle to identify panels - even disregarding the other differing factors.

Nevertheless, it serves as a proof of concept for the network, meaning that once a better dataset is pre-processed allowing the network to train on it should yield a good result.

CHAPTER 6

Obtaining and Sending Waypoints

Having a way to obtain the positions of the solar panels in the images, this section will now lay out a potential control scheme, from obtaining the bounding boxes to passing the commands to the flight controller. This implementation works under the assumption that all images are geotagged. Meaning that the center of the image has a given latitude, longitude and altitude (L, L', A), and that the cardinal directions compared to the image coordinates are known.

The flight controller runs on ArduPilot, whose communication protocol is MAVLink. ArduPilot support two different kinds of GPS positions, real-time kinematics (RTK) and pure GPS based localisation. Using RTK the accuracy of the GPS is greatly increased, i.e., for the on-board GPS of the drone, the Here3+, the accuracy is increased from $\pm 2.5m$ to $\pm 0.025m$[Cub]. With such high accuracy, it should be possible to use the center of each column of panels as waypoints. For waypoint creation, MAVLink expects an input consisting of the x,y,z coordinates expressed as (L, L', A). Thus, the primary objective will be to convert the (x,y) pixel positions to longitude and latitude commands. The altitude is given by the GPS itself. Thus, the general pipeline should look something like this:

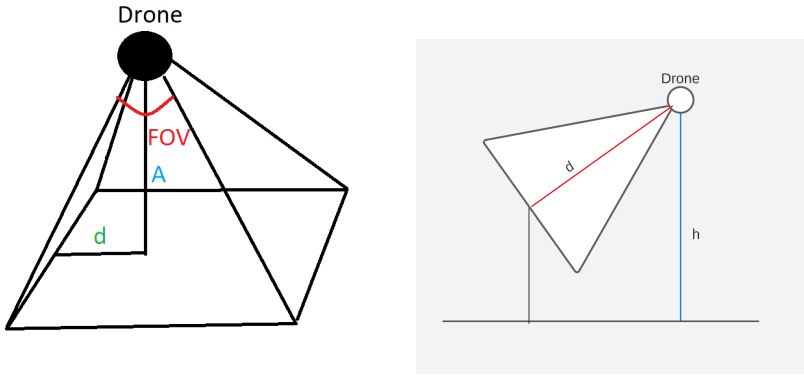
1. Take picture and identify PVs
2. Find (x,y) coordinates of bounding boxes - compute centre of panels
3. Transform (x,y) coordinates into (L, L', A)
4. Pass (L, L', A) to MAVLink as a waypoint command

Step 1 is already taken care of. Bounding boxes can be directly extracted from the output of the "detect.py" script provided by the YOLO framework. Afterwards, computing the centre of the panels is pretty straightforward:

$$(x_c, y_c) = (x_{min} + \frac{x_{max} - x_{min}}{2}, y_{min} + \frac{y_{max} - y_{min}}{2}) \tag{6.1}$$

In order to take these coordinates and transform them into (L, L', A), we first need to know the real-life distances of the photo. Initially, I will make the assumption that

the drone is always flying directly above the PVs. Considering that the Aerial Tools drone does not currently have a gimbal, this is a safe assumption to make for now. Doing so simplifies the problem to the situation crudely sketched out in figure 6.1(a).



((a)) Drone sketch for acquiring real-life distances from images

((b)) Drone looking at a panel at an angle

Figure 6.1: Sketches of the two situations for observing panels

Using simple geometry the following relation appears:

$$d = A \cdot tan(\frac{FOV}{2}) \tag{6.2}$$

Where d is the real-life distance, in meters, from the centre of the image to the border, A is the relative altitude of the drone compared to the ground, and the FOV is the field of view, which is defined by the intrinsic parameters of the camera. Dividing the width and height of the image with d yields the pixels per meter(ppm). In turn, this can be used to find the distances from the centre of the image to each of the centres of the panels. Then, all that remains is to transform the coordinates. Latitude and longitude are defined as having the following relations:

$$Latitude(L): \quad 1^\circ = 111.32km \longrightarrow L(^\circ) = \frac{x}{111320}m \tag{6.3}$$

$$Longitude: \quad \frac{40,075,000cos(L)}{360} = L'(^\circ) \tag{6.4}$$

x here is the distance travelled north or south from the equator. Inserting our values for the centres gives the change in (L, L') from the centre of the geotagged picture. Thus, in order to obtain the final coordinates we sum the geotagged (L, L') coordinates with our deltas. With that, the waypoint is ready to be sent to the ArduPilot.

As mentioned, however, the situation is quite a bit more complicated if we assume that we cannot assume the panels to be straight below us. In that case, the situation is as seen in figure 6.1(b). It is not possible to obtain the distance from a single

picture without any assumptions, however, we do have some options for how to resolve the situation. For example, if you know the length of a panel, you can use the relation between that length and the bounding boxes obtained to find pixel length. Alternatively, by taking two pictures from slightly different positions in the same plane you could create a depth map:

$$x - \hat{x} = \frac{B \cdot f}{d} \tag{6.5a}$$

$$d = \frac{B \cdot f}{x - \hat{x}} \tag{6.5b}$$

Where x is the pixel value of the images, B is the distance between the two images and f is the focal length of the camera. There are some other caveats to this solution, such as accounting for image distortion and ensuring that we are in fact looking straight at the panel, but both these solutions are options at least - with the bonus that *OpenCV* would support both implementations.

CHAPTER 7

Future Work

As it became clear that the dataset would not be good enough for the resulting network to be directly useable for Aerial Tools, I immediately set to work in searching for a better dataset. In the end, a far better alternative presented itself in the form of the "Multi-resolution dataset for photovoltaic panel segmentation from satellite and aerial imagery" [Jia+21]. Unfortunately, these issues were discovered too late into the work to fully prepare a functional model using it. However, as the pipeline for the training is working, all that needs to be done is to pre-process a new dataset and leave it to train. Thus, the overall remaining workload should not pose too big of a challenge. Alternatively, an even better solution would be to compile and annotate a new dataset comprised of images taken from the drone during a variety of different runs. Regardless of the approach, the dataset should be comprised of high-quality UAV photos from various PV farms under different lighting and weather conditions. Ideally, the colours should be strong, but the larger the variety presented the better. As discussed in chapter 2, the pictures should also include PVs standing at various different angles in order to have as little bias as possible. Additionally, the dataset should be larger than the one used for training in this report to allow for optimal performance. As noted in chapter 5 the model has also only been trained for a single set of parameters. Thus, it would make sense to explore different learning rates, momentum etc. to achieve even higher-quality results.

On the notion of training, while Colab is a good tool, it does pose some constraints in the form of kicking you out after a certain amount of time unless you are using a paid plan. Seeing as DTU provides access to supercomputers, it would likely be worthwhile to spend some time looking into how to use these, as their capabilities also far exceed what Colab could provide in terms of GPU power. For this model, however, it was not done as it became obvious an alternative dataset would be needed regardless.

Implementing the main function and integrating both the model and the waypoint generation onto hardware as described in chapter 6 also remains to be done. Here, it is also wise to look into not only the simple case but also the case in which the camera is free to move.

CHAPTER 8

Conclusion

In conclusion, while the result of the project is not a fully implementable network, it is providing a framework in which one can easily create a general network given a better dataset in the future. Such a dataset could for example be the one provided in[Jia+21], or even better, a self-made one. Perhaps even a fusion of those two approaches in order to provide as much variance as possible. That being said, for the dataset used, the model is able to attain good results in precision despite extremely small object sizes - although this might unfortunately also be attributed to overtraining based on a uniform dataset. Regarding the model type, multiple different avenues have been analysed, with the corresponding trade-off readily available should a need for a different approach arise in the future. Not only that but there is also supplied python implementations for training of both the yolov7 network as well as Mask R-CNN. Constraints regarding training and how to potentially alleviate them by making use of the DTU supercomputers were also explored, although it will be up to Aerial Tools to set up the environment in the future.

Moreover, the communication setup with the flight controller has been explored, ending with a rudimentary control algorithm as well as potential ways to improve upon it once a gimbal has been acquired. With this, the communication between the main computer which will be gathering the image data and the flight controller has been laid out - providing each of the steps necessary from a picture is taken until a command is sent.

APPENDIX A

An Appendix

A.1 Code

```
    class PVDataSet(torch.utils.data.Dataset):
    def __init__(self, root, d_transforms = None):
        self.root = root
        self.transforms = d_transforms

        # Load all image files
        self.imgs = list(os.listdir(os.path.join(root, "img/")))
        self.masks = list(os.listdir(os.path.join(root, "labels/")))

    def __getitem__(self, idx):
        # Load images and masks
        img_path = os.path.join(self.root, "img/", self.imgs[idx])
        mask_path = os.path.join(self.root, "labels/", self.masks[idx])

        # Convert the image to RGB
        img = Image.open(img_path).convert("RGB")

        # Convert to grayscale
        mask = Image.open(mask_path)
        mask = ImageOps.grayscale(mask)

        # Convert PIL to np-array
        mask = np.array(mask)

        # BLOB Analysis
        label_im = label(mask)

        # Instances are different colours
        obj_ids = np.unique(label_im)

        # First id is background - remove it
        obj_ids = obj_ids[1:]

        # Split into multiple separate mask segments
        masks = (label_im[:, None, None] == obj_ids[:, None, None])

        # Convert masks to C, D, H, W
        masks = np.transpose(masks, (2,1,0,3))

        # Loop through and get the boxes
```

```
        num_objs = len(obj_ids)
        boxes = []
        for i in range(num_objs):
            pos = np.where(label_im == obj_ids[i])
            xmin = np.min(pos[1])
            xmax = np.max(pos[1])
            ymin = np.min(pos[0])
            ymax = np.max(pos[0])
            boxes.append([xmin, ymin, xmax, ymax])

        # Convert everything into a torch.Tensor
        boxes = torch.as_tensor(boxes, dtype=torch.float32)

        # There is only one class
        labels = torch.ones((num_objs,), dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)

        image_id = torch.tensor([idx])
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])

        # Suppose all instances are individual
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target

    def __len__(self):
        return len(self.imgs)
```

Listing A.1: Dataset class for RCNN

```
for path in test_mask_paths:
    name = path.removesuffix(".png") + ".txt"
    print(name)

    # Open and convert to grayscale
    mask = Image.open(path)
    mask = ImageOps.grayscale(mask)
    x_norm, y_norm = mask.size

    # Convert PIL to np-array
    mask = np.array(mask)
    label_im = label(mask)
    vals = np.unique(label_im)
    vals = vals[1:]
```

```
    masks = (label_im[:, None, None] == vals[:, None, None])
    masks = np.transpose(masks, (2,1,0,3))

    num_objs = len(vals)
    center_x = []
    center_y = []
    w = []
    h = []
    boxes = []
    for i in range(num_objs):
        pos = np.where(label_im == vals[i])
        xmin = np.min(pos[1])/x_norm
        xmax = np.max(pos[1])/x_norm
        ymin = np.min(pos[0])/y_norm
        ymax = np.max(pos[0])/y_norm
        boxes.append([xmin, ymin, xmax, ymax])
        x = xmin + (xmax - xmin)/2
        y = ymin + (ymax - ymin)/2
        if x > 1 or y > 1:
            print("Error in normalisation")
            print(f"X: {x}, Y: {y}")
        center_x.append(x)
        center_y.append(y)
        h.append(ymax - ymin)
        w.append(xmax - xmin)

    with open(name, "a") as f:
        for i in range(num_objs):
            if i != num_objs - 1:
                content = '0 ' + str(center_x[i]) + ' ' + str(center_y[i]) +
                    ' ' + str(w[i]) + ' ' + str(h[i]) + '\n'
                #print(content)
                f.write(content)
            else:
                content = '0 ' + str(center_x[i]) + ' ' + str(center_y[i]) +
                    ' ' + str(w[i]) + ' ' + str(h[i])
                #print(content)
                f.write(content)
```

Listing A.2: Script for extracting bounding boxes and getting it to the format expected by yolov7

Bibliography

[Cub] CubePilot. *Here 3 Manual*. URL: https://docs.cubepilot.org/user-guides/here-3/here-3-manual.

[Gad20] Ahmed Fawzy Gad. "Faster R-CNN Explained for Object Detection Tasks." In: (2020). URL: https://blog.paperspace.com/faster-r-cnn-explained-object-detection/.

[Gir15] Ross Girshick. *Fast R-CNN*. 2015. DOI: 10.48550/ARXIV.1504.08083. URL: https://arxiv.org/abs/1504.08083.

[He+17] Kaiming He et al. *Mask R-CNN*. 2017. DOI: 10.48550/ARXIV.1703.06870. URL: https://arxiv.org/abs/1703.06870.

[Jia+21] H. Jiang et al. "Multi-resolution dataset for photovoltaic panel segmentation from satellite and aerial imagery." In: *Earth System Science Data* 13.11 (2021), pages 5389–5401. DOI: 10.5194/essd-13-5389-2021. URL: https://essd.copernicus.org/articles/13/5389/2021/.

[Lu+21] Tan Lu et al. *Comparison of RetinaNet, SSD, and YOLO v3 for real-time pill identification*. 2021. DOI: https://doi.org/10.1186/s12911-021-01691-8. URL: https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-021-01691-8#citeas.

[McW+21] Claire McWhite et al. "Co-fractionation/mass spectrometry to identify protein complexes." In: *STAR Protocols* 2 (March 2021), page 100370. DOI: 10.1016/j.xpro.2021.100370.

[Rat22] Sovit Rath. "Fine Tuning YOLOv7 on Custom Dataset." In: (2022). URL: https://learnopencv.com/fine-tuning-yolov7-on-custom-dataset/ (visited on November 11, 2022).

[Ren+15] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2015. DOI: 10.48550/ARXIV.1506.01497. URL: https://arxiv.org/abs/1506.01497.

[Ren+22] Simiao Ren et al. "Utilizing Geospatial Data for Assessing Energy Security: Mapping Small Solar Home Systems Using Unmanned Aerial Vehicles and Deep Learning." In: *ISPRS International Journal of Geo-Information* 11.4 (March 2022), page 222. DOI: 10.3390/ijgi11040222. URL: https://doi.org/10.3390%5C%2Fijgi11040222.

[RF18] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement.” In: *arXiv* (2018).

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ