Crown of Thorns Starfish Detection

OLA ANDRÈ FLOTVE*, University of Stavanger, Norway
HÅVARD MOE JACOBSEN, University of Stavanger, Norway
HANNAH HÅLAND, University of Stavanger, Norway

The Great Barrier Reef is the largest reef in the world, and functions as a home to a countless number of species. Lately its been under a lot of distress, and one of the reasons behind this is the increasing population of coral-eating crown-of-thorns starfish (COTS) [23]. To assess this problem TensorFlow [25] created a competition in Kaggle [7] where they released a dataset containing video footage from the Great Barrier Reef. In this paper we want to use this dataset to detect COTS by using different models from the Tensorflow Object Detection API [6], and present their results with common metrics used for object detection.

CCS Concepts: • **Computing methodologies** → **Machine learning algorithms**; • **Supervised Learning**; • **Neural Networks**; • **Object Detection** → Transfer Learning;

Additional Key Words and Phrases: COTS, Crown of thorns starfish, detection, Tensorflow Object Detection API, Weights & Biases, Google Colab, Faster R-CNN Resnet50, EfficientDet, CenterNet, Hourglass

ACM Reference Format:
Ola Andrè Flotve, Håvard Moe Jacobsen, and Hannah Håland. 2022. Crown of Thorns Starfish Detection. *ACM Trans. Graph.* 37, 4, Article 111 (August 2022), 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The rising number of COTS is a real and present danger to the Great Barrier Reef. Protecting the reef is essential to make sure that the existing biodiversity persists. Detection of the COTS are conventionally carried out by snorkel divers where they use a method called "Manta Tow" [23]. The procedure consists of a diver being towed by a boat and simultaneously observing the reef while taking occasional recordings [23]. Even though the "Manta Tow" is a good working method, it has its limitations, such as performing surveys over larger areas [23].

This brings us to object detection. Object detection is when computers are assigned to find objects within a image or a video and classify these objects by giving them a label such as dog, cat or in this case COTS. Today the most commonly used method for object detection is deep learning [3].

Many different models for object detection have been developed with varying performance. In this project we want to see if we are able to use these models for detecting COTS.

We test these models:

- Faster R-CNN Resnet50
- EfficientDet-d0
- CenterNet Hourglass104

We also want to compare the performance of the models in terms of mean average precision, average recall and training time. We want to see if Weights & Biases can be a effective tool for hyper parameter tuning and for visualization of model performance.

2 CONTRIBUTIONS

All authors contributed equally to the project. Ola pre-processed the data and set up github environment. Hannah created models using TensorFlow Object Detection API in Google Colab. Håvard integrated Google Colab with Weights & Biases and performed training and testing. All contributed to the final report write-up.

3 DATASET

As mentioned earlier the research conducted in this paper will be using a dataset [23] provided by TensorFlow from a Kaggle competition for detection of COTS. The size of the dataset is at around 15 GB, and the data is provided as images which are sorted into their respective folder depending on which video the image belongs to. It also includes a file called train.csv that contains metadata for the images, and which has the following information:

- **video_id**:
  A integer representing the id of the video which the image belongs to.
- **sequence**:
  Gap free sequence id from a specific video.
- **video_frame**:
  The frame number of the image inside a video (exists gaps because of divers surfacing).
- **sequence_frame**:
  The number for a frame which is in a sequence.
- **image_id**:
  An id for the image consisting of video id and video frame.
- **annotations**:
  A list of bounding boxes marking the position of the COTS. The boxes are in COCO format.

[24] **Note:** We also decided to leave out the images without annotations.

4 THEORY

In this chapter we will introduce the theory required to understand key concepts in the paper.

*All authors contributed equally to this research.

Authors' addresses: Ola Andrè Flotve, oa.flotve@stud.uis.no, University of Stavanger, Kjell Arholms Street 41, Stavanger, Rogaland, Norway, 4021; Håvard Moe Jacobsen, ham.jacobsen@stud.uis.no, University of Stavanger, Kjell Arholms Street 41, Stavanger, Rogaland, Norway, 4021; Hannah Håland, han.haland@stud.uis.no, University of Stavanger, Kjell Arholms Street 41, Stavanger, Rogaland, Norway, 4021.

0730-0301/2022/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

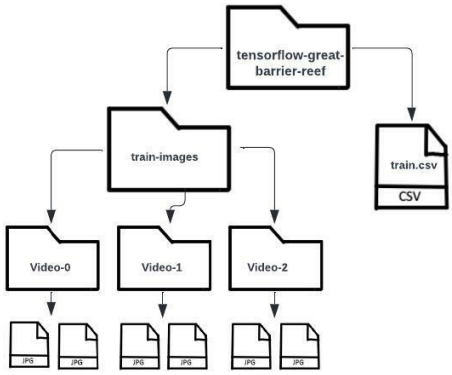

Fig. 3.1. Dataset file structure

4.1 Object Detection

In this project we are performing object detection - which is localising an object on an image and drawing a bounding box around it. This is different from image classification which labels the whole image as being about a particular object, e.g. is it a cat, dog, hat, etc; and image segmentation which is able to tell the shape of the object, not just draw a bounding box around it.

4.2 Bounding Boxes

A bounding box is used to mark an object typically inside a image. The shape of the box is either rectangular or square depending on the object in question. The two most common formats used are the COCO format and the Pascal format.

The COCO format uses x-top left, y-top left, width and height. The top left x and y coordinates represent the top left corner of the box, and width and height represent the width and the height of the box [8].

Pascal format uses x-top left, y-top left, x-bottom right, and y-bottom right. Similar to COCO format, top left x and y represents the top left corner of the box, however we also represent the bottom right corner with the bottom right x and y coordinates [8].

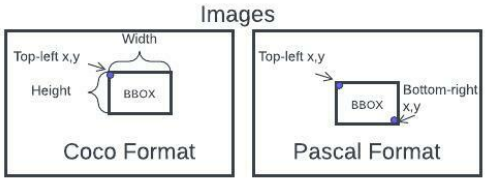

Fig. 4.1. Illustration of COCO and Pascal formats.

4.3 Models

4.3.1 Faster R-CNN ResNet 50. To understand the Faster R-CNN ResNet 50 model we first need to take a look at the convolutional neural network (CNN).

ResNet 50 stands for residual network and the number indicates the amount of layers in the network. When going deeper with the classic CNN architecture certain problems started occurring such as vanishing and exploding gradients. To solve this problem residual blocks where introduced. In simple terms the residual blocks takes the output value of the preceding layer and forward it to the next layer. This lays the foundation for utilizing deeper networks, but CNN's still can only classify one object at a time which brings us to R-CNN's.[2]

R-CNN's opens the door to the classification of multiple objects at a time. The idea behind is to use an algorithm called selective search which chooses 2000 region proposals. The proposed regions are then transformed into a square, which then again is given to the CNN. The job of the CNN is then to extract the features which again are sent into a support vector machine (SVM). The SVM will then decide if the object is within the region or not. It also produces four offset values which are used to improve the accuracy of the bounding boxes. While the R-CNN works, its use of the selective search algorithm makes it slow, which led to the rise of the faster R-CNN. [5]

The Faster R-CNN got rid of the selective search algorithm. Different to the R-CNN version the image is now fed directly to the CNN which returns a convolutional feature map. The feature map is then put into a region proposal network which produces predictions for the regions. In the last step the predicted regions are given to a Region of Interest pool layer, which again is used to produce a classification for the regions and a prediction for offset values used in the bounding boxes [5].

The combination of the residual network with 50 layers merged together with the faster version of the R-CNN then gives us the Faster R-CNN Resnet 50 model.

4.3.2 EfficientDet-d0. EfficientDet is a relatively new network that got published by Tan et al. working for the Google Brain team in 2020 [20]. In order to understand how it is built, we first study EfficientNet which acts as the backbone of its architecture.

EfficientNet is a CNN that was also first presented by the Tan et al. in 2019 [19]. They mentioned that CNNs were usually built with a fixed size, which then could be scaled up for better performance if you have the computation required for this to be done. As a way to make this process more efficient, they introduced a new scaling method which scales the network in both width, depth and input-resolution. This type of scaling introduced a lot of possibilities; you can make the layers wider, you can add more layers (scale depth), as well as tune the resolution of the images passed into the network [17]. By combining all of these hyperparameters, one gets a lot of different combinations to try in order to find the best tuning for your dataset.

The reason we mentioned EfficientNet is because it works at the input part of the EfficientDet. The input gets passed into a EfficientNet, and gets passed upwards from layer to layer until the end of the network. The output from the last 'n' layers is not just given to the

next EfficientNet layer, but also gets passed horizontally to a new part of the EfficientDet network, namely the Bi-directional Feature Pyramid Network (BiFPN). As the first BiFPN layer is connected to the EfficientNet backbone, it will receive input of decreasing size (from bottom to top), illustrations can be found at a Google blog written by the model inventor; Tan [18]. Using input of multiple resolutions of the same image in a bi-directional layer, makes it possible to see patterns which could be hard to find on a single image. At the very end there is a class/box network that interprets the output from the BiFPN network, and predicts bounding boxes for the image. **EfficientDet-d0** is the smallest version of these scaled networks which exist in range from 0-7.

4.3.3 CenterNet Hourglass104. According to the model's configuration file this model has a CenterNet [31] meta-architecture built on on the Hourglass [11] backbone.

The **Hourglass network** is a fully convolutional network with and encoder-decoder structure. The idea behind the network is to find prominent features and localize them in one forward pass. The network is made of convolution layers, max pooling layers, residual layers, bottleneck layers and up-sampling layers. Originally, (stacked) hourglass networks were used with great success for human pose detection [4].

The **CenterNet** model was introduced in 2019 as an alternative to the prevalent bounding box based detectors (such as faster RCNN). The model represents objects as a single point at the bounding box center and regresses other properties such as size, location, orientation, and even pose, depending on the application at hand [31]. At the time of its release, Centernet out-performed a range of state-of-the-art algorithms at that time (such as FasterRCNN, YOLOv3, RetinaNet) in terms of accuracy and speed [31].

How it works is that an input image is fed into a fully convolutional network (such as the Hourglass Network) that generates a heat map. Inference is performed with a single forward pass without using non-maximal suppression (NMS) for post processing [31]. The center point on the heatmap, offset and dimensions are inferred. By removing computationally demanding NMS, CenterNet achieves a large improvement in speed. The authors actually tested Centernet with Hourglass104 backbone and achieved the best accuracy out of all existing one-stage detectors at that time. Centernet Hourglass104 did not outperform sophisticated two-stage detectors in terms of accuracy, but it was faster [31].

4.4 Transfer Learning

In this project we use transfer learning to perform COTS detection. Transfer learning is the process where we use a model pre-trained on a certain dataset then train it for a new specific problem. In principle, knowledge is transferred from the previous task to the new task and this leads to improved performance, potentially not needing as large of a dataset when training the new model. Refer to [14] for a unified definition of transfer learning and comprehensive review of transfer learning.

4.5 Performance Metrics

As with all types of models we need metrics to be able to evaluate them. In this section we will give a brief introduction to the ones used in this paper.

Note: We have used the COCO format which may have slight variation for some of the metrics. The most important information is covered below, but for deeper understanding we recommend taking a look at this blog [12].

4.5.1 Intersection over Union. Intersection over Union(IoU) also known as the Jaccard index lays the foundation for the calculation of some of the most popular metrics used to measure the performance of object detectors. With the use of terms such as the True Positive (TP) (correct detection), the False Positive (FP) (incorrect detection), the False Negative (FN) (object not detected), and the True Negative (TN) (correctly detected background) [9].

The actual formula for the IoU is given by the following equation:

$$IoU = \frac{Area\ of\ Intersection}{Area\ of\ Union} \tag{1}$$

where the intersection and union is calculated between the box of the actual position of the object and the predicted box position of the object. The resulting score is a number from 0-1. The more overlap the higher the score, the less overlap the smaller the score.

IoU is typically used together with a threshold alpha. The threshold is used to determine the TP, FP, FN, and TN. A IoU score higher or equal to alpha corresponds to TP, smaller than alpha FP, and FN is the part of the actual box which has not been detected by the predicted box for which the IoU is smaller than the alpha. This is beautifully illustrated in figure 4 in Koech article [9].

4.5.2 Precision. Precision is a metric used to measure the correctness of the model from all the objects it has detected [9]. The formula is given below:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

From precision we can calculate the average precision(AP). The AP is the area under the Precision-Recall curve evaluated at alpha as IoU threshold [9]. The formula is given by:

$$AP@alpha = \int_0^1 p(r)dr \tag{3}$$

The next metric is the mean Average Precision(mAP) which is the average over all the average precision values from each specific class [9]. The formula is given by:

$$mAP@alpha = \frac{1}{m}\sum_{i=1}^{m} AP_i \tag{4}$$

where m is the number of classes.

4.5.3 Recall. The recall is how good the model is at detecting the true positive in regards to all the correct predictions [9]. The formula is given below:

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

Another metric that is calculated from recall is the average recall (AR). The AR is given as two times the area under the recall-IoU

curve where the IoU is between 0.5 and 1 [12]. The original formula is given below:

$$AR = \int_{0.5}^{1} recall(o)do \tag{6}$$

where o is the IoU.

5 IMPLEMENTATION

This chapter will go through the process of implementation for the different stages of the pipeline.

5.1 Data Preparation

The first part of data preparation consisted of downloading the dataset, the creation of the workspace, and moving the images from the dataset to the workspace. The acquisition of data was done with a custom made function which used the Kaggle API. The function takes in a path that specifies where to unzip the data, and also a dataset name which the Kaggle API uses to find the dataset of the competition. The function used can be seen below:

```
def download_dataset(data_dir, dataset_name):
    competition_dir = os.path.join(data_dir, dataset_name
    )
    if not os.path.exists(os.path.join(data_dir,
    dataset_name)):
        api = KaggleApi()
        api.authenticate()
        print("Dataset is around 15 GB may take a long
    time...")
        api.competition_download_files(dataset_name, path
    =data_dir)

        os.mkdir(competition_dir)
        with zipfile.ZipFile(
            os.path.join(data_dir, f"{dataset_name}.zip")
        ) as zip_file:
            zip_file.extractall(competition_dir)
        os.remove(os.path.join(data_dir, f"{dataset_name
    }.zip"))
        print("Download complete and fils unziped!")
    else:
        print("Dataset folder already present!")
    return competition_dir
```

Listing 1. Function used for acquisition of data from prep_data_functions.py

Another function named **create_workspace** was used to build the workspace where the data preparation was performed. We also moved all the images from the dataset to a new folder called images in the workspace with a simple function called **move_images**.

In the next step we had to change the bounding box annotations from COCO to Pascal format, as well as adding labels. This was needed for the creation of the TFRecords which will be described more in detail later. The annotations for the images where stored in a train.csv file which we loaded into a pandas dataframe [15]. From here we used pandas apply function together with a lambda function as well as our custom made functions. Below you can see one of the transformations performed with one of the custom made functions.

```
train_meta\_df["annotations\_pascal"] = train\_meta\_df.
    apply(
    lambda row: to\_pascal(row), axis=1
)
```

Listing 2. Coco to Pascal from prep_data.ipynb

```
def to_pascal(row):
    coco_list = row["annotations_coco"]
    pascal_list = []
    for coco in coco_list:
        x_left = coco["x"]
        y_top = coco["y"]
        x_right = coco["x"] + coco["width"]
        y_bottom = coco["y"] + coco["height"]
        pascal_dict = {
            "x_left": x_left,
            "y_top": y_top,
            "x_right": x_right,
            "y_bottom": y_bottom,
        }
        pascal_list.append(pascal_dict)
    return pascal_list
```

Listing 3. Function used for transforming COCO to Pascal taken from prep_data_functions.py

Afterwards a label map was created which is used to specify the id and label of the class. This was performed with the **create_label_map** function.

The last and most important step was the creation of the TFRecords. TFRecords are used by Tensorflow to store data in a sequenced way as binary records [27]. In the first step we split our data into three parts train, validation, and test with the **split_train_val_test** function. TFRecords were created for each of these sets. They were created by first storing the data in a so called TF Example which is a message type that creates a mapping of the data [27]. The TF Examples were then written into the actual TFRecord file.

To carry out these operations there was first created a function called **create_tf_example** that takes in a row with metadata information from the train.csv, and a path to the data directory. The function then extracts the needed information which then again is transformed to the right format and fed into TensorFlows TF Example function. A snippet from the function can be seen below:

```
    tf_example = tf.train.Example(
        features=tf.train.Features(
            feature={
                "image/height": dataset_util.
    int64_feature(image.size[1]),
                "image/width": dataset_util.int64_feature
    (image.size[0]),
                "image/filename": dataset_util.
    bytes_feature(
                    image_id.encode("utf-8")
                ),
                "image/source_id": dataset_util.
    bytes_feature(
                    image_id.encode("utf-8")
                ),
                "image/encoded": dataset_util.
    bytes_feature(encoded_img),
                "image/format": dataset_util.
    bytes_feature(
                    file_format.encode("utf-8")
                ),
                "image/object/bbox/xmin": dataset_util.
    float_list_feature(
                    x_left
```

```
                ),
                "image/object/bbox/ymin": dataset_util.
    float_list_feature(
                    y_top
                ),
                "image/object/bbox/xmax": dataset_util.
    float_list_feature(
                    x_right
                ),
                "image/object/bbox/ymax": dataset_util.
    float_list_feature(
                    y_bottom
                ),
                "image/object/class/text": dataset_util.
    bytes_list_feature(
                    class_names
                ),
                "image/object/class/label": dataset_util.
    int64_list_feature(
                    class_ids
```

Listing 4. Snippet from the data storing into TF Example taken from the function create_tf_example in prep_data_functions.py

The second step is the actual creation of the TFRecord file. For this part we implemented a function called **create_tfrecord**. The function takes in a dataframe which includes metadata such as image ids, a output path(where to store the data), and information about where to look for the data. It also takes in a Boolean called play which if true creates a play dataset that is considerably smaller than the actual dataset. Inside the function a TFRecordWriter [26] from TensorFlow is created. Then we iterate through each row in the dataframe where each row is sent into the **create_tf_example** function and the result is written to the TFRecord file. The function can be seen below:

```
def create_tfrecod(df, output_path, data_dir, play=False)
    :
    if play:
        n_samples = round(len(df)*0.1) #10%
        df = df.head(n_samples)

    writer = tf.io.TFRecordWriter(output_path)
    for _, row in df.iterrows():
        tf_example = create_tf_example(row=row, data_path
    =data_dir)
        writer.write(tf_example.SerializeToString())
    writer.close()
```

Listing 5. Function used for creating TFRecords taken from the function create_tf_example in prep_data_functions.py

Note: There was also created pytests for these functions.

5.2 Training the Models

We decided to use Google Colab for our project which is essentially an online Jupyter Notebook that allows users to execute Python code through a virtual machine. In this project we have used Colab **Pro** which allows access to faster GPUs and TPUs than the basic Colab version. There are limits with Colab (such as run-time limits, resource limits, idle-time limits) that can be prohibitive to progress for a project like ours, where there is high demand for protracted computing power. For the sake of training object detection models, we needed an environment with a GPU engine. That as well as the collaboration benefits it gave us, led us to run training on the cloud.

In order to start the model evaluation, we needed to prepare the runtime environment.

We downloaded the Tensorflow Object Detection API (TODA) which is an open-source framework built on top of TensorFlow that makes it easy to build, train and deploy object detection models [22]. A range of object detection models pre-trained on the COCO2017 dataset are available in the Tensorflow 2 Detection Model Zoo which is a part of the TensorFlow Model Garden [29], this includes the three models we are to evaluate.

The TODA comes with tutorials that guides the user step-by-step on preparation of the environment for object detection, implementing the models and performing the detection tasks. Many examples of the TODA applied to novel object detection can be found online. In this project we have referred to [21] and [30]. The whole API got downloaded into our Google Drive which we had full access to from our Colab notebook.

In order to log and visualize the training, we used a tool called Weights & Biases (WandB) [1]. Despite Tensorboard already working seamlessly with Tensorflow, we decided to use WandB because of its sweep-functionality and ease-of-use when collaborating as a team. The sweep function in WandB lets you choose a combination of hyper parameters that you want to evaluate and start training the models one by one with different combinations of the given parameters.

```
sweep_config = {
    'method': 'grid',
    'parameters': {
        'model': {'value': 'efficientdet-d0'},
        'batch_size': {'value': 3},
        'num_steps': {'values': [10000, 20000, 30000]},
        'learning_rate': {'values': [0.1, 0.03, 0.01]}
    }
}
```

Listing 6. Example of an sweep config

By passing {'method': 'grid'} we specify that we want to try every possible combination of parameters inside the sweep config. This combined with a could GPU environment give us the possibility of training models using multiple parameter combinations overnight.

To automate the training as much as possible, we created a dictionary containing information about the models we were going to use for the project. The dictionary holds information on where to find the belonging config file, as well as where to download the pretrained checkpoint from where you want to resume training from (see transfer learning for more details).

We made a main function for training and evaluation of models. Its input parameters are 'config' and 'run_id'; config is which model, and which parameters you are going to use whilst the run_id is simply the unique id for a 'train, evaluation' iteration in the sweep you are doing. Before training is initiated, the main function calls two preparation functions; 'prepare_env' and 'update_config_file'. **prepare_env** checks and potentially downloads chosen model's pretrained checkpoint and its config file. **update_config_file** does as the name suggests, it updates the model's config file. Following this, training and subsequently evaluation are performed by a calls to the model_main_tf2.py TODA script (see our training notebook in github for details [13]).

As mentioned, Tensorboard is already deeply integrated with Tensorflow. This integration includes the creation of so called 'tfevent' files (one for training and one for evaluation) which Tensorboard reads to show metrics from the training and evaluation. We created some logger-functions which are able to read the tfevent files, and log the relevant information to WandB.

6 RESULTS

In this section we will present the results from the predictions of COTS for the different models. The best performance for each model with its respective hyperparameters is stated in table 1. For all the results from different hyperparameter runs see the appendix (B).

model	steps	lr_init	lr_base	mAP	AR@1	AR@10	AR@100	R-time
resnet-50-faster	30000	NA	0.1	0.5918	0.3328	0.6223	0.6566	2h3m
efficientdet-d0	30000	NA	0.01	0.2872	0.2181	0.3790	0.4350	1h57m
centernet	10000	0.01	NA	0.1378	0.1190	0.2514	0.3042	1h55m

Table 1. Table containing stats of the best performing models

The true position of the bounding boxes for the COTS in image 2_0 and 2_2 are shown in figure 6.1 from top to bottom.

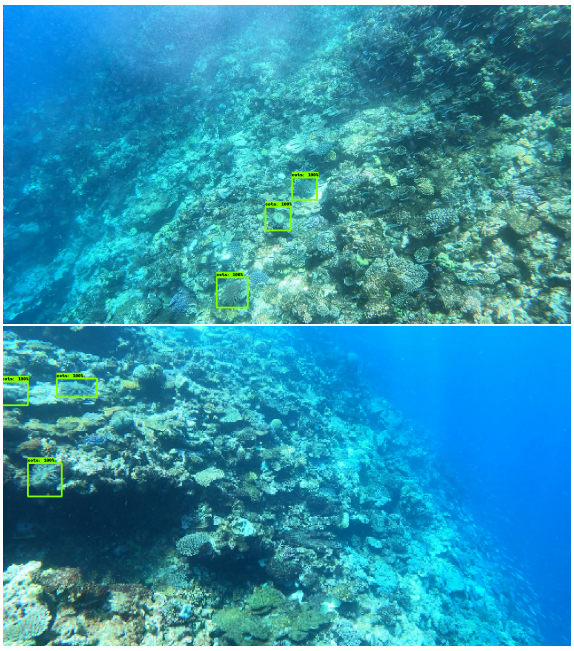

Fig. 6.1. image 2_0 and 2_2 respectively. Together with their labels

6.1 Faster R-CNN ResNet 50. Results

Faster R-CNN ResNet50's prediction results for image 2_0 and 2_2 can be seen in figure 6.2.

6.2 EfficientDet-d0 Results

The results from EfficientDet-d0 predcitons for image 2_0 and 2_2 can be found in figure 6.3.

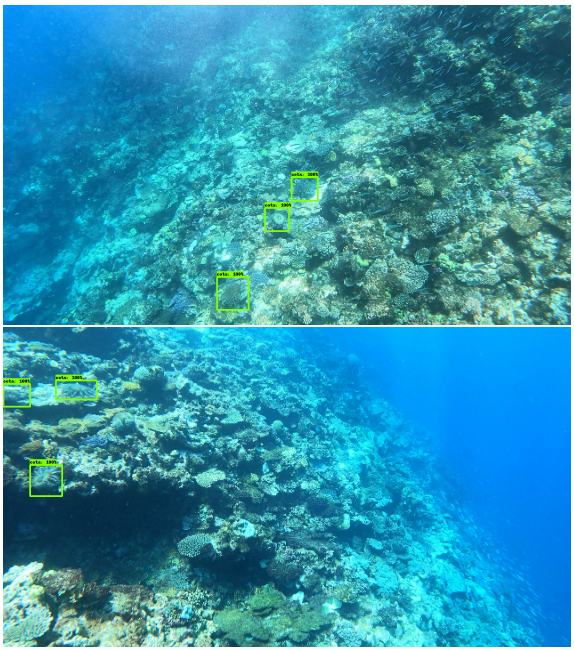

Fig. 6.2. Predictions from Faster R-CNN Resnet50

Fig. 6.3. Predictions from EfficientDet-d0

6.3 CenterNet Results

For CenterNet Hourglass104 the prediction results from image 2_0 and 2_2 can be observed in figure 6.4.

Fig. 6.4. Predictions from CenterNet Hourglass104

7 HYPERPARAMETER TUNING

Hyperparameter tuning is a essential part in the search for creating a good performing model. The parameters chosen will dictate the training phase which will have great impact on results. There are many different parameters to choose from. In this paper we have decided to work with the learning rate and the number of steps. We also had batch size as an input, but decided that the size should not be set to higher than 3 to prevent Out-Of-Memory errors from the GPU. Something that is well known is that hyperparameter tuning is a tedious task. To make this task easier we experimented with using sweeping [28] from Weights & Biases.

The config files that came with the pretrained models already had some chosen parameters in them. Among these, the optimizer, which we decided to leave as is. Inside the optimizer, you had to specify parameters for the optimizer's learning rate.

The Centernet config file came with a adam optimizer using a manual step learning rate. This means that it uses a specified initial learning rate until step 'k', where the learning rate gets updated to the next one (usually lower than the previous one). For our project, this was done twice; the learning rate got updated when reaching 60 and 80 percent of the total steps. The initial learning rate decreased 90 percent reaching the first threshold, and 90 percent more reaching the next one.

For Resnet and EfficientDet, a momentum optimizer containing a cosine decay learning rate using warmup was implemented. Firstly, warmup means that the learning rate begins at a lower level than the base rate, and slowly works its way up towards this base rate for 'k' number of steps. We chose this 'k' to be a 25th of the total number of steps.

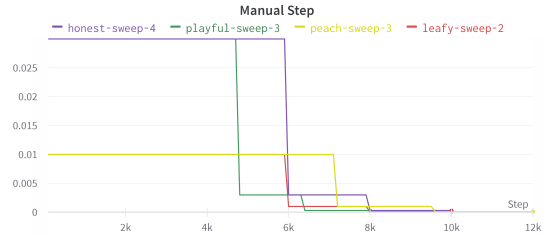

Fig. 7.1. Manual Step Learning Rate

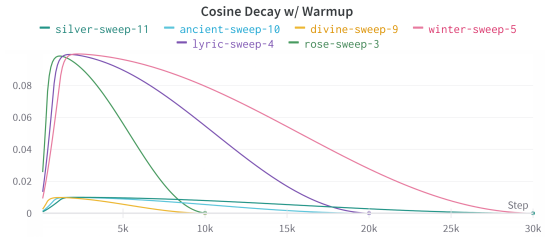

Fig. 7.2. Cosine Decay Learning Rate w/ Warmup

7.1 Faster R-CNN ResNet 50

Figure 7.3 showcases the use of different hyperparameters and how the mAP and AR@10 results are affected for Faster R-CNN ResNet 50. One can see that a higher number of steps seem to result in higher scores on the metrics. On the other hand, a larger learning rate combined with few steps has resulted in low scores which may indicate that the model has not been trained sufficiently enough. In conclusion, it looks like the most important part is to provide a high number of steps, and that the base learning rate has a smaller impact.

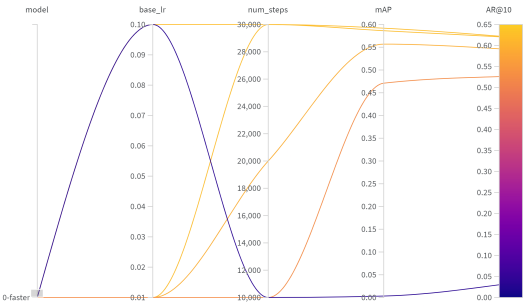

Fig. 7.3. Tuning Faster R-CNN Resnet50

7.2 EfficientDet-d0

Figure 7.4 showcases the use of different hyperparameters and how the mAP and AR@10 results are affected for EfficientDet-d0. For EfficientDet-d0 larger base learning rates look to have a bad impact on the metrics. We can see that for all of the runs that have used

a base rate of 0.1 regardless of the number of steps the resulting mAP and AR@10 are very low or zero. In contrast a smaller base learning rate of 0.01 combined with a higher number of steps seem to produce the best results.

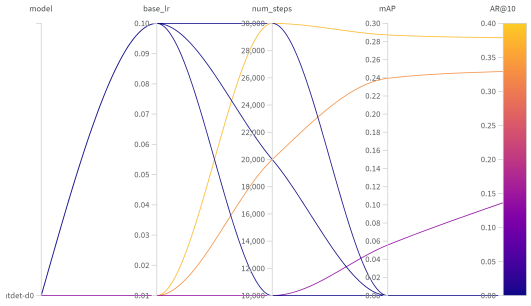

Fig. 7.4. Tuning EfficientDet-d0

7.3 CenterNet

Figure 7.5 showcases the use of different hyperparameters and how the mAP and AR@10 results are affected for CenterNet. The tuning of CenterNet was a challenging one. The network was extremely slow to train which resulted in less time to test different types of hyperparameters. The use of high learning rates looks to have a destructive effect resulting in metric results at zero. We can also see that it is not necessarily worthwhile to use to large steps, so to conclude the "best" results seem to be produced with a number of steps at around 10 000 and smaller initial learning rates such as 0.01.

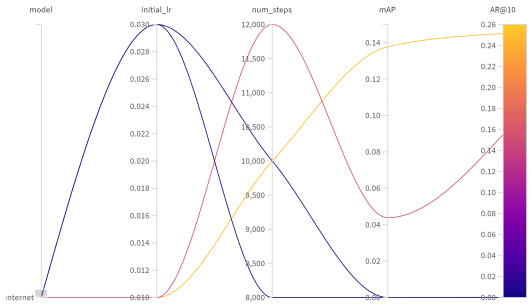

Fig. 7.5. Tuning CenterNet Hourglass104

8 DISCUSSION

After tuning hyperparamers, the best combinations of hyper parameters were used to train the final models. The results are shown in Section 6.

Overall the Faster R-CNN Resnet50 model provided the best outcome in terms precision and recall. The model required longer to train compared to EfficientDet-d0. During training it was found that CenterNet Hourglass104 was very slow to train so we did not explore increasing the number of steps.

The mAP results were surprising and counter to the reported performance on the COCO dataset - we would expect Centernet Hourglass104 to perform well, followed by EfficientDet-d0, then Faster RCNN Resnet50. This suggests that perhaps we have not found the most optimal set of hyperparameters for the models.

Model	COCO mAP	COTS mAP
Faster RCNN Resnet50	29.3	59.2
EfficientDet-d0	33.6	28.7
CenterNet Hourglass104	41.9	11.9

Table 2. Model performance compared to the COCO dataset [29]

An interesting observation is that our mAP for Faster RCNN Resnet50 is better than the COCO mAP. We are not sure what the reason for this is. Potentially this could be that we are dealing with only 1 class, whilst the COCO dataset contains 91 classes. This would mean the model concentrates learning a set of features for the 1 class and becomes better at it.

The lower mAP for EfficientDet-d0 and CenterNet Hourglass104 could be attributable to the size of COTS being quite small compared to the full image size. This may make them harder to detect.

9 CONCLUSION

Our work showed that it is indeed possible to detect COTS using deep neural networks. We had many ideas that we wanted to try in order to create the best model to detect COTS. The single most frustrating aspect of this work is computing power. Our lack of computing power restricted us from exploring all possible hyperparameters. Training time was excessively long which meant we were not able to test as many configurations as we would like. The fact that our results were counter to that of the COCO dataset suggests that we have not sufficiently optimized our models. Our dataset is a difficult one - images were were underwater and saturated. It is hard even for humans to detect the COTS so it is expected that the neural network will have similar difficulties. Future work should look at ways to handle underwater object detection. The applicability of transferring learning to this dataset should be addressed. Apart of that, computational power, data augmentation and search for optimal hyperparameters and optimal models should be explored. In light of what we have learnt, our list of future work items is long.

9.1 Future work

The work that we have performed has only scratched the surface of what is possible in object detection. We note the following which should we considered for future work.

Computational Power. This work is computationally intensive - training typically takes several hours and batch size was limited due to limitations on RAM. Future work should secure more powerful computers and perform computations over many computers at once.

Transfer Learning. In this work we have made the assumption that transfer learning will have positive improvement for our model. However, as shown by [16] when the source domain is very

different from the target domain, transfer learning may hinder performance. Given that the COCO dataset [10] is an extensive set of labelled images of 91 different categories and none of of which have any obvious relation to COTS, we need to answer the question *if* knowledge can be transferred, *what* part of that knowledge could be transferred, and *how* it can be transferred.

Hyperparameter Tuning. Due to time limits, we were not able to try tuning as many hyperparameters as we would like. It would be interesting to try tuning different parameters, e.g. optimizer type, warm up schedule, training schedule to see how this effects the models' performance.

Data Augmentation. Data augmentation techniques can be used to generate more training data and hence lead to increased performance. We should explore different techniques to generate more data for training.

Underwater Object Detection. Detecting objects underwater has a unique set of challenges such as image saturation. Techniques should be explored to handle challenges specific to underwater object detection.

Other Models. Due to time limits, we were not able to try as many models as we would like. In particular we would like to experiment with a YOLO-class model (which is not available in TODA). The current latest YOLO model is supposedly the best in terms of speed and performance.

REFERENCES

[1] Weights & Biases. 2022. Weights & Biases. Retrieved May 13, 2022 from https://wandb.ai/site

[2] Gaudenz Boesch. 2022. Deep Residual Networks (ResNet, ResNet50) – Guide in 2022. Retrieved May 13, 2022 from https://viso.ai/deep-learning/resnet-residual-neural-network/

[3] Gaudenz Boesch. 2022. *Object Detection in 2022: The Definitive Guide.* viso. Retrieved May 11, 2022 from https://viso.ai/deep-learning/object-detection/

[4] Nushaine Ferdinand. 2020. Using Hourglass Networks To Understand Human Poses. Retrieved May 12, 2022 from https://towardsdatascience.com/using-hourglass-networks-to-understand-human-poses-1e40e349fa15

[5] Rohith Gandhi. 2018. R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms. Retrieved May 13, 2022 from https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[6] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. 2016. Speed/accuracy trade-offs for modern convolutional object detectors. https://doi.org/10.48550/ARXIV.1611.10012

[7] Kaggle. 2022. Kaggle. Retrieved May 11, 2022 from https://www.kaggle.com/

[8] Renu Khandelwal. 2019. COCO and Pascal VOC data format for Object detection. Retrieved May 12, 2022 from https://towardsdatascience.com/coco-data-format-for-object-detection-a4c5eaf518c5

[9] Kiprono Elijah Koech. 2020. Object Detection Metrics With Worked Example. Retrieved May 14, 2022 from https://towardsdatascience.com/on-object-detection-metrics-with-worked-example-216f173ed31e

[10] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2014. Microsoft COCO: Common Objects in Context. https://doi.org/10.48550/ARXIV.1405.0312

[11] Alejandro Newell, Kaiyu Yang, and Jia Deng. 2016. Stacked Hourglass Networks for Human Pose Estimation. https://doi.org/10.48550/ARXIV.1603.06937

[12] Nickzeng. 2018. An Introduction to Evaluation Metrics for Object Detection. Retrieved May 15, 2022 from https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/

[13] Hannah Håland Ola Andrè Flotve, Håvard Moe Jacobsen. 2022. COTS Detection. https://github.com/havardMoe/cots_detection.

[14] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

[15] Pandas. NA. pandas.DataFrame. Retrieved May 15, 2022 from https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

[16] Michael T. Rosenstein, Zvika Marx, Leslie Pack Kaelbling, and Thomas G. Dietterich. 2005. To transfer or not to transfer. In *In NIPS'05 Workshop, Inductive Transfer: 10 Years Later*. Empty, Empty, Empty pages.

[17] Jacob Solawetz. 2020. EfficientDet for Object Detection. Retrieved May 14, 2022 from https://blog.roboflow.com/breaking-down-efficientdet/

[18] Mingxing Tan. 2020. EfficientDet: Towards Scalable and Efficient Object Detection. Retrieved May 14, 2022 from https://ai.googleblog.com/2020/04/efficientdet-towards-scalable-and.html

[19] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, arxiv, NA, 6105–6114.

[20] Mingxing Tan, Ruoming Pang, and Quoc V Le. 2020. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. arxiv, NA, 10781–10790.

[21] Gilbert Tanner. 2020. Tensorflow Object Detection with Tensorflow 2: Creating a custom model. Retrieved May 03, 2022 from https://gilberttanner.com/blog/tensorflow-object-detection-with-tensorflow-2-creating-a-custom-modelf

[22] Great Learning Team. 2021. *Real-Time Object Detection Using TensorFlow.* Great Learning Team. Retrieved May 12, 2022 from https://www.mygreatlearning.com/blog/object-detection-using-tensorflow/

[23] TensorFlow. 2021. TensorFlow - Help Protect the Great Barrier Reef. Retrieved May 11, 2022 from https://www.kaggle.com/competitions/tensorflow-great-barrier-reef

[24] TensorFlow. 2021. TensorFlow - Help Protect the Great Barrier Reef. Retrieved May 11, 2022 from https://www.kaggle.com/competitions/tensorflow-great-barrier-reef/data?select=greatbarrierreef

[25] TensorFlow. 2022. TensorFlow. Retrieved May 11, 2022 from https://www.tensorflow.org/

[26] TensorFlow. NA. tf.io.TFRecordWriter. Retrieved May 15, 2022 from https://www.tensorflow.org/api_docs/python/tf/io/TFRecordWriter

[27] TensorFlow. NA. TFRecord and tf.train.Example. Retrieved May 15, 2022 from https://www.tensorflow.org/tutorials/load_data/tfrecord

[28] Weights and Biases. 2022. Hyperparameter Tuning. Retrieved May 15, 2022 from https://docs.wandb.ai/guides/sweeps

[29] Hongkun Yu, Chen Chen, Xianzhi Du, Yeqing Li, Abdullah Rashwan, Le Hou, Pengchong Jin, Fan Yang, Frederick Liu, Jaeyoun Kim, , and Jing Li. 2020. TensorFlow Model Garden. https://github.com/tensorflow/models.

[30] Hugo Zanini. 2021. Custom object detection in the browser using TensorFlow.js. Retrieved May 03, 2022 from https://blog.tensorflow.org/2021/01/custom-object-detection-in-browser.html

[31] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. 2019. Objects as Points. https://doi.org/10.48550/ARXIV.1904.07850

A GITHUB REPOSITORY

Our code is available at our GitHub repository [13].

B TABLES

model	batch_size	nr_steps	lr_rate_base	mAP	AR@1	AR@10	AR@100	Runtime (s)	steps_per_sec
resnet-50-faster	3	30000	0.1	0.591889	0.3327962	0.62227487	0.656587	7404	4.170066
resnet-50-faster	3	30000	0.01	0.586365	0.328625	0.62075829	0.654976	7425	4.190214
resnet-50-faster	3	20000	0.01	0.556876	0.319241	0.592132	0.624265	5019	4.175323
resnet-50-faster	3	10000	0.01	0.471062	0.290521	0.525876	0.555829	2611	4.150116
resnet-50-faster	3	10000	0.1	0.003158	0.010995	0.030331	0.067962	2635	4.224659

Table 3. Faser R-CNN ResNet50 results

model	batch_size	nr_steps	lr_rate_base	mAP	AR@1	AR@10	AR@100	Runtime	steps_per_sec
efficientdet-d0	3	30000	0.01	0.287249	0.218104	0.378957	0.434976	7047	4.378952
efficientdet-d0	3	20000	0.01	0.239488	0.186824	0.329289	0.397819	4795	4.407646
efficientdet-d0	3	10000	0.01	0.055742	0.065876	0.136492	0.203033	2476	4.377227
efficientdet-d0	3	30000	0.1	0	0	0	0	7023	4.395498
efficientdet-d0	3	20000	0.1	0	0	0	0	4754	4.398981
efficientdet-d0	3	10000	0.1	0	0	0	0	2466	4.384568

Table 4. Efficientdet-d0 results

model	batch_size	nr_steps	initial_lr_rate	mAP	AR@1	AR@10	AR@100	Runtime	steps_per_sec
centernet	3	10000	0.01	0.1377567	0.118957	0.251374	0.304265	6923	1.543672
centernet	3	12000	0.01	0.0438355	0.050900	0.155260	0.228909	8261	1.539879
centernet	3	8000	0.03	0.000005	0	0.000094	0.004170	5579	1.555591
centernet	3	10000	0.03	0.000001	0	0	0.000853	6941	1.539509

Table 5. CenterNet results