TRAFFIC SIGN RECOGNITION IN THE WILD

A PREPRINT

December 11, 2020

ABSTRACT

We trained a Convolutional Neural Network to classify traffic signs into the wild. Our project work with any type of images, high or low resolutions. For training, we have used 1640 train images taken from Google Maps. These images are labeled with a program called "labelImg" [1]. In this project, we use Faster-RCNN-Inception-V2-COCO model from TensorFlow. TensorFlow provides several object detection models. Some models (such as the SSD-MobileNet model) provide an interface that allows for quicker detection but with less precision, whereas some models (such as the Faster-RCNN model) have slower detection but with more accuracy. At the end of our model training the classification loss was near 0.03. The localization loss is 0.06. These results are available after 37.000 training steps. After training the network with both models we reached the conclusion that Faster-RCNN-Inception-V2 can be trained faster than the SSD-MobileNet-V2, and gives more accurate results.

***K*eywords** Deep learning · R-CNN · Faster R-CNN · Visual recognition

1 Introduction

Deep learning is one of the hottest and most trending topics in computer science nowadays. Several breakthrough research works are available year by year. There is an incredible amount of data (e.g. text, visual, audio) what generated and gathered by companies every day. This enormous scale of the information cannot be processed and interpreted by humans. Therefore we were trying to utilize computers to solve these tasks faster and more efficiently.

In simple words, deep learning is the subset of machine learning in which multilayered neural networks learn from a vast amount of data. We use deep learning for visual recognition more specifically detect and classify object on images i.e. read images in a certain form, train a convolution neural network (CNN) which will

[1]https://github.com/tzutalin/labelImg

output the target classes what are recognized on the image. (Classification is the task when there are many defined target classes and we assign a class label for the images.)

In these days, there are numerous existing applications which use artificial intelligence for visual recognition in different scenarios, such as face recognition, revolutionize of medical diagnosis and detect severe diseases, driverless automobile technology and so on. However, these applications are quite new and require more time to improve, thus there are rarely some flaws in them. For example, an issue happened with one of the major electric vehicle company's autopilot when the car was mistaking a fast food restaurant sign for a stop sign and the vehicle was slowed down. This event indicated our motivation towards this topic because we were curious about how to solve the traffic sign recognition problem in the wild.

To address the problem we trained an object detection classifier which uses Faster Region-based Convolutional Neural Network (Faster R-CNN) and SSD-MobileNet.

2 Related work

In this section, we briefly discuss related work and we introduce the necessary and good to know background knowledge about R-CNN, Fast R-CNN, Faster R-CNN and SSD-MobileNet.

2.1 Collect and label traffic sign images

Our dataset, both for training and for test, is taken from: [2]. These images are taken from Google Maps and labeled using labelImg[3]. We have 1640 images used for training and 413 test images.

2.2 Configure training data

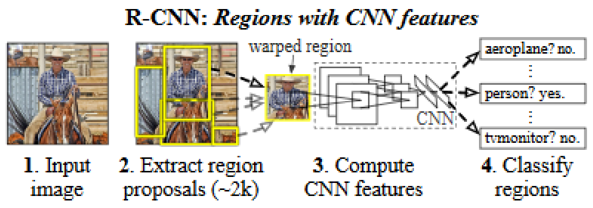
The last thing to do before training is to create a label map and edit the training configuration file. The label map tells the trainer what each object is by defining a mapping of class names to class ID numbers. Our example is the following for traffic signs:

```
item {
  id: 1
  name: 'no_stopping_or_parking'
}

item {
  id: 2
  name: 'no_entry'
}

item {
  id: 3
  name: 'crosswalk'
}

item {
  id: 4
  name: 'give_way'
}
```

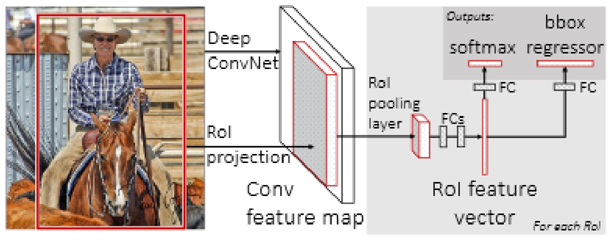
We could add more labels for our neural network but this would increase the epochs needed for training. For simplicity, we chose to work with only these four classes.

[2]https://github.com/arthas009/Traffic-Sign-Detection-using-Faster-R-CNN?fbclid=IwAR2FU4R7rJk5N8Ob1l9h0BjaOYTknxHkaqN8YuGxGNld3240J-_Mgps7G7I

[3]https://github.com/tzutalin/labelImg



(a) R-CNN [13]



(b) Fast R-CNN [14]

Figure 1: Difference between the structure of R-CNN and Fast R-CNN

2.3 R-CNN, Fast R-CNN, Faster R-CNN, SSD-MobileNet

The difference between algorithms for object detection and classification algorithms is that we try to draw a bounding box around the object of interest in the detection algorithms to locate it inside the image. It can easily happen that the image will not only have one bounding box drawn, and we cannot know how many beforehand. The key explanation why the design of a regular convolutionary network followed by a fully connected layer will not continue with this problem is that the length of the output layer is variable, not constant, since the number of occurrences of the objects of interest is not fixed. [17][16]

To tackle this issue, a naive solution would be to take different regions of interest from the image and utilize a CNN to find out the object's presence within that part of the image. The difficulty with this technique is that there might be different spatial positions within the picture and different aspect ratios for the objects of interest.[17][16] As a result we will need to pick an immense amount of regions and this probably blow up computationally. Hence, several algorithms like R-CNN and others have been developed to find the previously mentioned regions fastly.

2.3.1 R-CNN

To tackle with this previous problem to select a huge number of regions, Ross Girshick et al.published a solution where they use selective search to extract around 2000 region proposals. Therefore we need to manage and classify only 2000 regions.[13][16]

The selective search algorithm operate the following way:

1. Generate initial sub-segmentation of input image
2. Use greedy algorithm to recursively combine the smaller similar regions into larger ones
3. Use the generated regions to produce the final candidate region proposals

However, there are several disadvantages of this method because it takes a lot of time to train the network as we have to classify around 2000 region proposals per image. Moreover, the selective search algorithm is fixed and there is not happening any learning.[13][16]

2.3.2 Fast R-CNN

To solve the issues with the previous approach the Fast R-CNN method was introduced by the same author. The main modification in algorithm design is instead of the region proposals, the input image is fed to the CNN to generate a convolutional feature map. From this convolutional feature map the algorithm identify the region proposals and wrap them with a bounding box.[14][16]

The fundamental cause why this new approach is faster than the R-CNN because we do not need to run the CNN around 2000 times for every image. On the contrary, we should run the CNN for every image once and it outputs the feature map i.e. region proposals.[14][16] The difference between the current and previous approach are visualized on Figure 1. The subfigures were taken over from the corresponding research papers.

2.3.3 Faster R-CNN

Both previous algorithms use selective search to find the region proposals. However, selective search is a computationally very expensive approach, thus it has a significant impact on the CNN performance.[15]

To solve the problem above they made an object detection system which has two parts. The first part is a convolution network which proposes the regions instead of using selective search. The second part is a Fast R-CNN detector that uses the previously proposed regions.[15][16] The architecture of this solution can be seen on Figure 2, which was taken over from the Shaoqing Ren et al. (2016): Faster R-CNN research paper.

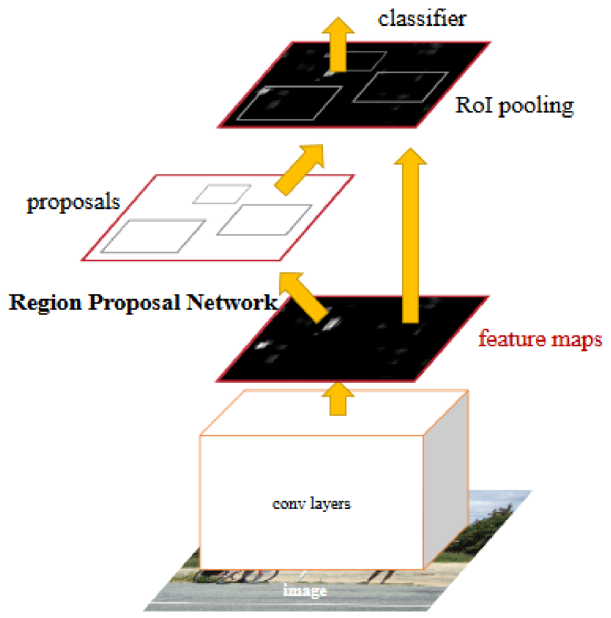


Figure 2: Structure of Faster R-CNN [15]

2.3.4 SSD-MobileNet

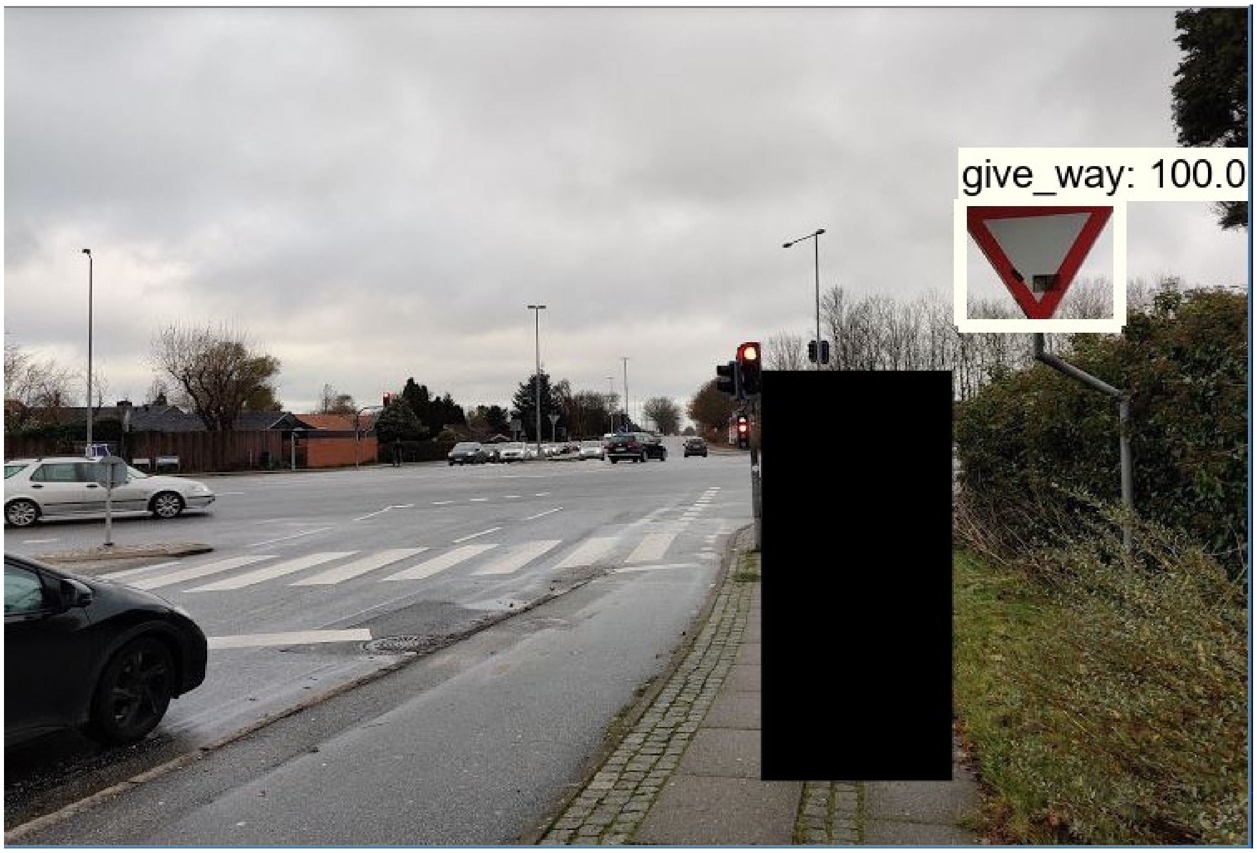
We can obtain a consistency between precision and speed by using SSD. SSD can only run a convolutionary neural network on the given input image once it calculates the map of a function. In SSD, a small 3×3-sized convolution kernel is operating on this feature map to estimate bounding boxes and classification probabilities. It also uses anchor boxes at different aspect ratios, such as Faster-RCNN, and learns off-set rather than boxing.

Tiagrajah and Mohamed [4] reached the conclusion that "Faster-RCNN inception V2 model works better than SSD MobileNet V2 model. FRCNN model performs well in both detecting the objects and quick convergence during training phase. SDD MobileNet V2 model produced higher false negatives in our experiments." We can say that our results relate to the ones that they have found since after conducting our experiments the results showed that FRCNN performs better than SSD-MobileNet.

3 Methods

3.1 Implementation

When starting the implementation, we first looked into how we can train a neural network to recognize traffic signs. We have found the GTSRB - German Traffic Sign Recognition Benchmark data set[5] that was offering more than 50.000 images of different traffic signs. Using this data set, we have implemented a small neural



[redacted]

Figure 3: Image classification of a give way sign

network using CNN & Keras that could perform image classification. Unfortunately, this data set contained only images of traffic signs alone, and our goal was to detect traffic signs in the wild.

Even though the initial project could not be used for further development, it gave us a better understanding of how Convolutional Neural Networks(CNN) work. Our next step was to find a suitable object detection API that we could use for our project. We ended up using Tensorflow[6]. It is an open-source platform for machine learning that is offering an object detection API that we can use. The TensorFlow Object Detection API is an open-source framework that is built on top of TensorFlow, making it easy to build, train, and deploy object detection models[8]. With this API, we can detect and localize traffic signs present in a given image.

We have decided to use Tensorflow v1.14. From our point of view, it was better documented, and there were more available resources on the internet. They offer multiple detection models that can be used for image recognition. Some models have an architecture that allows for faster detection but with less accuracy (such as the SSD-MobileNet model), while some models (such as the Faster-RCNN model) provide slower detection but with more accuracy. We have picked up one of the faster R-CNN models, the faster_rcnn_inception_v2_coco model[7]. As they stated on their GitHub repository, it had the best detection speed (58ms) and a good mAP[ˆ 1] (28).

We wanted to use the Tensorflow-GPU variant of this API because we had to train our model from scratch. For this to work, we needed a CUDA enabled GPU. The implementation of the project was created on an Asus laptop equipped with: Windows 10 (64 bit), CPU Brand: Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz, DirectX Card: NVIDIA GeForce GTX 950M 2Gb VRAM, RAM: 12177 Mb.

Firstly, we had to install Anaconda, Nvidia CUDA driver, and cuDNN specific to our version of TensorFlow that we were going to use. Anaconda[9] is a software toolkit that creates Python virtual environments so that Python libraries can be installed and used without having to worry about creating version conflicts with current installations.

After these requirements were met, we created an Anaconda virtual environment with Python 3.6 and cloned the TensorFlow object detection repository located at https://github.com/tensorflow/models and installed all the dependencies. This repository offers the training capabilities needed to train a neural network with the object detection API they are offering. The training Python script requires a detection model configuration that states the detection model, training and test image folders, the label map of each sign, and other relevant information needed, such as batch size.

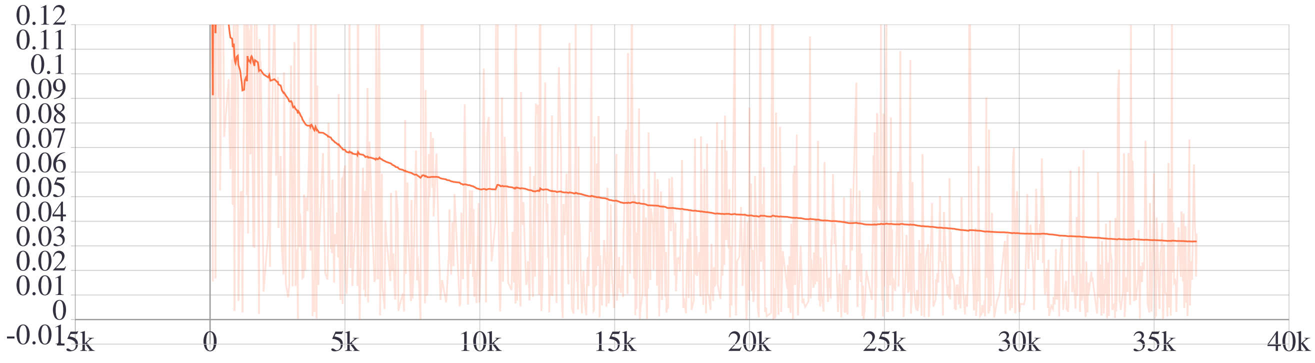


Figure 4: Classification loss

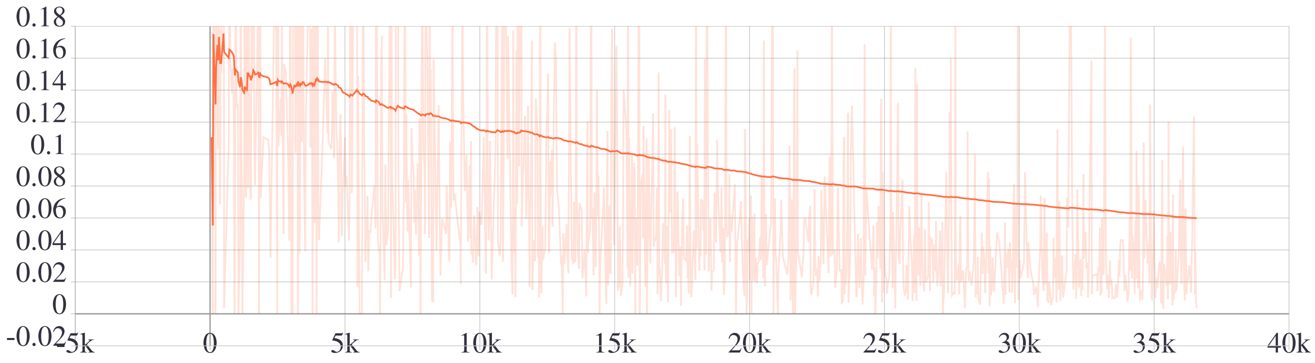


Figure 5: Localization loss

Our data set was labeled using LabelImg[10], hence each image has an XML file that states which signs are present in the image and their location. For this to be digestible by our training script, we needed to generate the TFRecords that serve as input data to the TensorFlow training model. These records were generated using xml_to_csv.py and generate_tfrecord.py scripts from Dat Tran's Raccoon Detector dataset[11], with some modifications to work with our directory structure.

First, a recursive search is performed by the xml_to_csv.py script for XML files that are attached to images. All the data will be collected inside a CSV file. This will create the CSV test and train files that the next script, the generate_tfrecord.py script, will use. This will create files for train.record and test.record and will be used to train the new classifier for object detection.

For our training process to start, we had to configure a training pipeline. This pipeline defines the model and parameters that will be used for training. With that configured, we could start training our neural network.

We have trained our neural network for about 37.000 epochs. At that point, we were receiving a pretty constant loss, and we decided to stop our training process. Figure 4 presents our classification loss graph and figure 5 presents our localization loss graph. Both of these graphs were generated using TensorFlow's TensorBoard[12], a visualization toolkit. This will create a webpage on your local machine that can be viewed through a web browser. The TensorBoard page provides information and graphs showing how the training is progressing. One important graph is the Loss graph, which shows the overall time loss of the classifier.

The overall loss started at about 3.0 and quickly dropped below 0.9 after as little as 20 epochs. Each epoch would take 0.7 to 0.8 seconds to be performed, using TensorFlow-GPU. Using the CPU variant of TensorFlow, the training becomes way slower, taking between 5 to 6 seconds for an epoch to be executed.

As a last test, we took a photo with a traffic sign that was located to our home in order to see the efficiency of the neural network. We got an accuracy of 100% for a "give way" sign and 97 % for a "no entry" sign. Figure 3 shows the result the we received.

```
image = cv2.imread(PATH_TO_IMAGE)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_expanded = np.expand_dims(image_rgb, axis=0)
# Perform the actual detection by running the model with the image as
    input
(boxes, scores, classes, num) = sess.run(
    [detection_boxes, detection_scores, detection_classes,
        num_detections],
    feed_dict={image_tensor: image_expanded})

# Draw the results of the detection
vis_util.visualize_boxes_and_labels_on_image_array(
    image,
    np.squeeze(boxes),
    np.squeeze(classes).astype(np.int32),
    np.squeeze(scores),
    category_index,
    use_normalized_coordinates=True,
    line_thickness=8,
    min_score_thresh=0.60)
```

Listing 1: Python image object detection code

3.2 Experiments

After succeeding in training a classifier for object detection using a faster R-CNN model, we decided to do another object detection classifier, using a different model. TensorFlow offers multiple object detection models that can be used for image recognition. Most of them were either faster R-CNN models or Single Shot Detector (SSD) models pre-trained on different data sets. As we have already trained a faster R-CNN model, trying another wouldn't make much sense in our case. The benchmarks stated on TensorFlow's GitHub are pretty similar, and the differences would be too little.

We chose to try out the ssd_mobilenet_v2_coco model from their website. The SSD models are usually used on devices with lower computation power than a laptop or desktop PC, but we decided to try it out. The overall loss started at about 20. It was way higher than the one we were getting with the Faster R-CNN implementation, but each epoch was taking less time. Around 0.3 seconds using the GPU variant of TensorFlow and 1 second on the CPU.

Because we were not running the training procedure on a low powered device, we have increased the batch size to the point where going a little higher would crash our Python script due to not enough RAM or VRAM. From that batch size of 1, we could increase it to 12. This would mean that we are feeding 12 images at a time instead of only 1. After this modification, the training speed became way slower, and a little bit more effective. Each epoch would take 1 to 1.2 seconds to be performed, using TensorFlow-GPU, while the CPU variant of TensorFlow was taking between 5.5 to 6 seconds for a step to be executed.

After 300 epochs, the loss that we were achieving was around 9.00. Even the first epochs of our last neural network were achieving a smaller loss. The training speed was mostly the same as the implementation with the Faster R-CNN model, but the loss was very bad in comparison. This would mean that we needed to allocate more time into training this neural network to perform as well, but due to time limitations, we stopped the experiment.

4 Results

In this section, we present our results based on the evaluation of the Convolutional Neural Network that we've created. These results are generated by an evaluation script made by TensorFlow, a script that evaluates detection models.

We have run the evaluation against our test data-set. This data-set contained 413 images of traffic signs taken from Google Maps. On average, the image evaluation time took:

$$EvalTime = \frac{4.87}{413} \approx 0.01179sec = 11.79ms$$

Table 1 shows the Average Precision and the Average Recall on IoU(intersection over union) thresholds. These thresholds are between 0.5 and 0.95, where the maximum detections over which it is calculated is 100. Average Precision stands for #correct detections / total detections, and Average Recall stands for = #ground truth with matched detections / total ground truth.

Table 1: AP & AR evaluation values

Average Precision	IoU=0.50:0.95	area= all	maxDets=100	= 0.541
Average Precision	IoU=0.50	area= all	maxDets=100	= 0.902
Average Precision	IoU=0.75	area= all	maxDets=100	= 0.579
Average Precision	IoU=0.50:0.95	area= small	maxDets=100	= 0.282
Average Precision	IoU=0.50:0.95	area=medium	maxDets=100	= 0.586
Average Precision	IoU=0.50:0.95	area= large	maxDets=100	= 0.797
Average Recall	IoU=0.50:0.95	area= all	maxDets= 1	= 0.551
Average Recall	IoU=0.50:0.95	area= all	maxDets= 10	= 0.620
Average Recall	IoU=0.50:0.95	area= all	maxDets=100	= 0.626
Average Recall	IoU=0.50:0.95	area= small	maxDets=100	= 0.419
Average Recall	IoU=0.50:0.95	area=medium	maxDets=100	= 0.668
Average Recall	IoU=0.50:0.95	area= large	maxDets=100	= 0.834

Our Mean Average Precision(mAP) values are:

DetectionBoxes_Precision/mAP = 0.54122233
DetectionBoxes_Precision/mAP (large) = 0.79659146
DetectionBoxes_Precision/mAP (medium) = 0.58643335
DetectionBoxes_Precision/mAP (small) = 0.28197667
DetectionBoxes_Precision/mAP@.50IOU = 0.90166384
DetectionBoxes_Precision/mAP@.75IOU = 0.5791029

5 Discussion

Our analysis proved the theory that the Faster R-CNN solution provide higher accuracy than SSD-MobileNet. However, this solution is also slower in detection than SSD-MobileNet. Furthermore, the results indicate that the Faster R-CNN approach is truly usable in real world scenarios based on the evaluation time of an image in our test dataset. In line with the outcome of Shaoqing Ren et al. (2016) research we also achieved a similar scale in detection time which can be measure in tens of milliseconds. We found that the current state-of-the-art CNNs provide excellent performance, but we need to face with downsides on the accuracy or detection time. Thus, a future research area could be to find a new model which may gives us fast detection time with high accuracy.

References

[1] George Kour and Raid Saabne. Real-time segmentation of on-line handwritten arabic script. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 417–422. IEEE, 2014.

[2] George Kour and Raid Saabne. Fast classification of handwritten on-line arabic characters. In *Soft Computing and Pattern Recognition (SoCPaR), 2014 6th International Conference of*, pages 312–318. IEEE, 2014.

[3] Guy Hadash, Einat Kermany, Boaz Carmeli, Ofer Lavi, George Kour, and Alon Jacovi. Estimate and replace: A novel approach to integrating deep neural networks with existing applications. *arXiv preprint arXiv:1804.09028*, 2018.

[4] v. Janahiraman, Tiagrajah and Mohamed Subuhan, Mohamed Shahrul. Traffic Light Detection Using Tensorflow Object Detection Framework *10.1109/ICSEngT.2019.8906486*, 2019.

[5] GTSRB - German Traffic Sign Recognition Benchmark *https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign*

[6] Tensorflow *https://www.tensorflow.org/*

[7] Tensorflow Detection models *https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md*

[8] Tensorflow GitHub repository *https://github.com/tensorflow/models*

[9] Anaconda *https://www.anaconda.com/*

[10] LabelImg *https://github.com/tzutalin/labelImg*

[11] Dat Tran's Raccoon Detector dataset *https://github.com/datitran/raccoon_dataset*

[12] Tensorboard *https://www.tensorflow.org/tensorboard*

[13] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation *arXiv preprint arXiv:1311.2524v5*, 2014.

[14] Ross Girshick. Fast R-CNN. *arXiv preprint arXiv:1504.08083v2*, 2015.

[15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object-Detection with Region Proposal Networks. *arXiv preprint arXiv:1506.01497v3*, 2016.

[16] Comparision of different R-CNNs *https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e*

[17] Image recognition in general *https://towardsdatascience.com/module-6-image-recognition-for-insurance-claim-handling-part-i-a338d16c9de0*