Capstone Project

Machine Learning Engineer Nanodegree

VIVA Hand Detection Challenge With RetinaNet

Kyle Sejin Yoon
Jan. 14, 2018

Definition

Project Overview

Deep Learning has become a significant field for Human-Computer Interaction (HCI). The interface to a computer has changed from just keyboards and mice to various touch gestures on a mobile device and motion wands for augmented reality. The inspiration for this project is from the interaction between Tony Stark and Jarvis, in the movie *Ironman*. An example of such HCI can be seen in this video. [8]

In this project, RetinaNet, a one-stage object detection system developed by Lin et al at Facebook AI Research [1], is used for the VIVA Hand Detection Challenge. A Keras implementation of RetinaNet by Fizyr is used as the implementation. Additionally, the model is evaluated against the other challenge entries. Finally, the trained convolutional neural network (CNN) is tested on video recordings of hand motions. While a practical HCI application would detect hands in real-time, this project limits the scope to recorded videos.

Problem Statement

The goal of this project is to create a hand detector that can be utilized for HCI. The steps to accomplishing this goal are:

1. Download and preprocess the data from the VIVA Hand Detection Challenge
2. Train a RetinaNet that can draw bounding boxes around detected hands
3. Compare performance using the challenge evaluation kit with Octave
4. Repeat steps 2-3 until a satisfactory evaluation is reached
5. Record videos with hand movements
6. Predict bounding boxes on each frame of the video
7. Output a video with bounding boxes drawn on each frame

Metrics

The area under the prevision-recall curve (AP) and average recall (AR) are commonly used metrics for assessing image object detection performance. These metrics are not only used for this particular challenge evaluation, but also for evaluations of other popular challenges, such as the PASCAL VOC and MS COCO.

Precision and **recall** are a measure of relevance within the detection results. They are values calculated using number of true positives (TP), false positives (FP), and false negatives (FN).

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN}$$

In this project, true positives are detection results that have an **Intersection over Union (IoU)** of 50%.

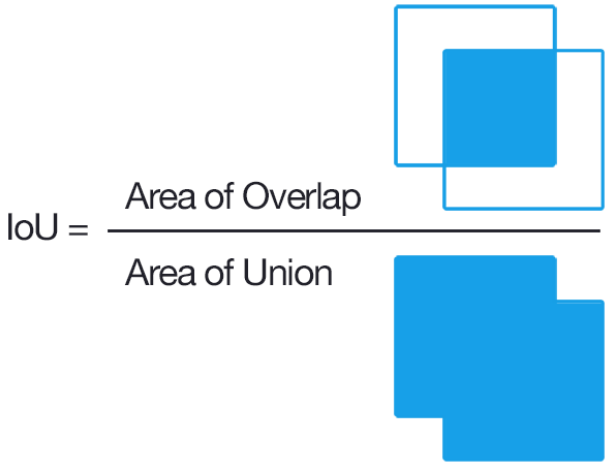

Fig. 1 Explanation of how the IoU is calculated with the detection and ground truth boxes [4]

False positives are detection results that are lower than 50% or redundant detections for the same ground truth object. False negatives are the cases when there are no detections for a given ground truth annotation.

The precision-recall curve shows the tradeoff between precision and recall. The AP summarizes the precision and recall, so that a high value indicates high precision and recall.

The evaluation toolkit provided by the challenge calculates the AP and AR. In detail, the evaluation kit for the VIVA Challenge obtains the AP by using the trapezoidal rule to approximate the area under the precision-recall curve. The metrics reported for this

challenge is on two levels. Level 1 (L1) is for hand instances with a minimum height of 70 pixels, with images only from over the shoulder (back) camera view. Level 2 (L2) is the more difficult evaluation, with hand instances of minimum height of 25 pixels and from all camera views.

Analysis

Data Exploration

The dataset for the VIVA Hand Detection Challenge has 5500 training images and 5500 testing images. The images are of the front interior space of cars with a driver, and sometimes a passenger, in a natural setting with varying illumination and motions. Bounding boxes of the hands are provided for each image.

There can be up to 4 bounding boxes for each image. The bounding boxes can have labels of the driver's left or right hands or the passengers left or right hands. An example of an annotation is as follows:

```
% bbGt version=3
leftHand_driver 63 306 89 53 0 0 0 0 0 0 0
rightHand_driver 102 257 170 106 0 0 0 0 0 0 0
leftHand_passenger 489 357 93 73 0 0 0 0 0 0 0
rightHand_passenger 568 316 66 66 0 0 0 0 0 0 0
```

There are a total of **13229 annotations** of hands in the training dataset. The annotations by class:

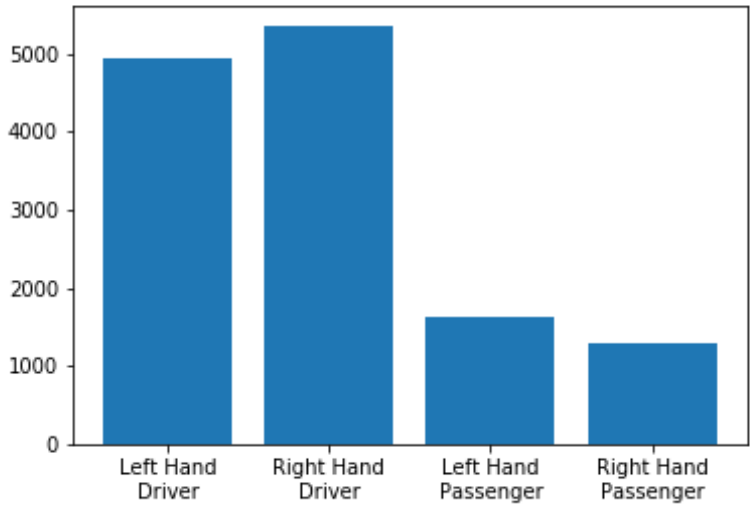

Fig. 2 A plot of the distribution of classes in the training dataset

In regards to the test data for evaluation, there are 6607 annotations that fall in the Level 1 evaluation category and 12080 that fall in the Level 2 evaluation category.

Algorithms & Techniques

One-Stage Detectors

Many of the current state-of-the-art object detectors are two-stage detectors, such as the popular R-CNN [5] and its successors [6]. They are two-stage because of the separation in detection by region proposal and classification. A one-stage system directly predicts the bounding boxes and class probabilities in a single classification process. This simplifies the model greatly, removing the region proposal step and post-processing refinements, as well as encapsulating the computation into a single network. [3, 4] Popular examples of a one-stage CNN are the YOLO [3] and SSD [4] models. Analysis of these models allow understanding of how one-stage detectors work.

The one-stage detectors mentioned above both use grid cells or default boxes that accomplish object detection. The terms "grid cell" and "default box" are more or less synonymous to the term "**anchor**" from RetinaNet [1, 3, 4]. These one-stage detectors in common apply a **dense** sampling of object locations on the input image. The sampling factors vary vastly, such as manipulating the scale or aspect ratio of the location. Ultimately, the model outputs bounding boxes and class probabilities at the anchors.

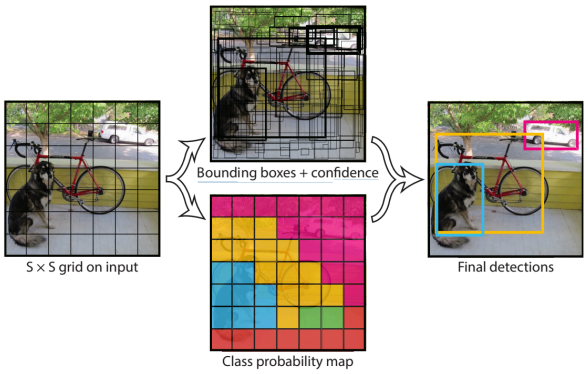

Fig. 3 YOLO grid cell system [2]

One-stage detectors sample a large set of locations from the input image, which leads to a significant the imbalance in background versus target locations. Approximately a 100k locations are sampled. [1] There are two problems with this:

1. In efficient training as most locations are easy negatives that do not contribute much learning [1]
2. The high volume of easy negatives can degenerate models [1]

RetinaNet

The classifier, RetinaNet, is a one-stage CNN that uses a novel *Focal Loss* for it's loss function. This focal loss algorithm focuses training on hard negatives, while reducing the importance of easy negatives. This compensates for the dense sampling problems mentioned above. [1] The formula for the focal loss function is as follows:

$$FL(p_t) = -\alpha_t(1-p_t)^{\gamma}\log(p_t)$$

The architecture of RetinaNet is made up of a Feature Pyramid Network backbone on top of a ResNet architecture. Then each of these layers of the backbone are connected to two subnets, one for classification and the other for regressing bounding boxes.

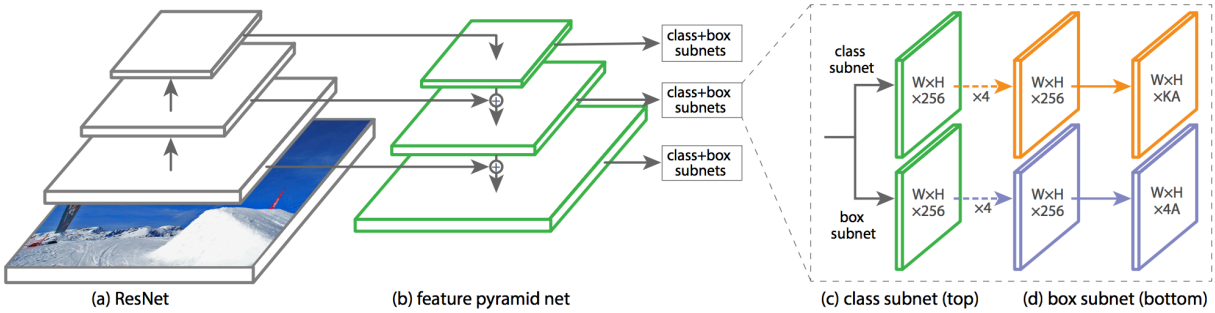

Fig. 4 RetinaNet architecture [1]

Feature Pyramid Network

A Feature Pyramid Network (FPN) [7] augments the standard CNN so that multiple scales of features are created from a single resolution image. Each of these multi-scale layers are connected to two subnets, one for classification and the other for regression.

This FPN is built on a ResNet architecture. In other words, the ResNet backbone output is fed into multiple scales of the pyramid where classification and regression features are generated at each scale.

Classification Subnet

One of the subnet types that are connected to the FPN is the classification subnet. The function of the classification subnet is to predict the probability of the labels at each of the anchors. The subnet architecture starts with four 3X3 convolution layers followed by a ReLU activation layer. Then there is another 3X3 convolution layer that is followed by sigmoid activation:

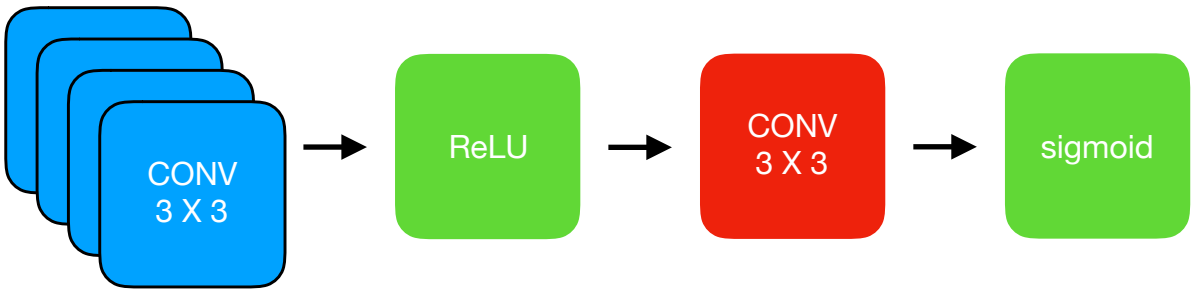

Fig. 5 A diagram of the classification subnet layers

Regression Subnet

The other subnet type that is attached in parallel to the classification subnet is the regression subnet. This subnet regresses the offset of the anchor box to the nearby ground-truth bounding box. The regression subnet uses the same architecture as the classification subnet above.

Parameters

The following parameters can be tuned to optimize the RetinaNet:

- Iteration
  - Number of epochs
  - Batch size
  - Steps per epoch
- Loss functions
  - Classification
  - Box Regression
- Optimizer
  - Learning rate
  - Weight decay
  - Momentum
- Detection threshold (minimum score for detections to use for evaluation)

Benchmark

The benchmark model for this project is from *Focal Loss for Dense Object Detection* by Lin, Tsung-Yi et al, which introduces the architecture and it's application on MS COCO. The parameters detailed in the research will serve as the benchmark to experiment against.

In regard to performance, the VIVA Hand Detection Challenge leaderboard will serve as the benchmark. Two-stage detectors dominate the top ranks of the challenge. However, **YOLO** comes in at 10th with L1 scores of **76.4/46.0 (AP/AR) and L2 scores of 69.5/39.1**. The detection evaluation is done using the PASCAL overlap requirement of 50%. The environment that was used to produce these results is 6 cores@3.5GHz, 16GB RAM, Titan X GPU. Accomplishing a comparable result is the goal of this project.

Methodology

Data Preprocessing

The preprocessing requires downloading the VIVA dataset and it's evaluation kit. The rest of the preprocessing is done in `preprocessing.ipynb`. In this file, the image and annotation data is converted to train, validation and test CSV files for the generators to use. Additional analysis of the data is done here as well.

The steps are as follows:

1. Download and extract the VIVA Hand Detection Challenge dataset.
2. Download and extract the evaluation kit. (Not only is this necessary for evaluation, but also test image annotations are here.)
3. Extract the bounding boxes from the annotation text files. The class labels are also extracted from the annotation files.
4. Split the training data into train and validation datasets. A split of 80/20 is used for this project.
5. Write CSV files using the extracted data into `train.csv`, `validation.csv` and `test.csv`.

The preprocessing script goes through each annotation text file, extracts the bounding boxes, and outputs each bounding box with the associated image for each row. A training and testing CSV file is outputted. This results in **10585 training and 2646 validation annotations**.

Additional preprocessing for training include:

- Random horizontal flip

- Resizing the image while maintaining aspect ratio so that the long side is no bigger than 1024 px while making the short side 600 px, if possible.
- Ordering the images by aspect ratio
- Random shuffling of batches

For testing, the same preprocessing is applied minus the horizontal flip augmentation.

Implementation

The implementation process has three parts:

1. Model Training
2. Loss & AP/AR Evaluation
3. Video Detection

Model Training

The training environment is an Amazon Deep Learning AMI (Ubuntu) with the p2.xlarge configuration.

The CSV training data created during the preprocessing is used to train the RetinaNet model. The training is done in the `train.py` script provided by the `keras-retinanet` framework. For our training, the command used is:

```
python keras-retinanet-master/keras_retinanet/bin/train.py --epochs 30 --
steps 5000 csv ./data/train.csv ./data/classes.csv --val-annotations ./data/
validation.csv --val-steps 500
```

This executes the following:

1. Loads the `train.csv` and `validation.csv` data into memory.
2. Creates `CSVGenerator` that serves batches of data for training and validation
3. Initialize a ResNet50 with ImageNet weights to use as the backbone architecture.
4. Creates a Feature Pyramid Network and subnetworks that make up the rest of the RetinaNet.
5. Compiles the model with the focal loss for classification, standard smooth L1 loss for box regression, Adam optimizer.
6. Adds two Keras callbacks `ModelCheckpoint` and `ReduceLROnPlateau`.
7. Uses the `CSVGenerator` to batch train the RetinaNet.
8. Uses `pickle` Python module to save the training history.

Loss & AP/AR Evaluation

Octave or MATLAB is required to run the evaluation toolkit. For this project, Octave was used.

The steps to produce the AP and AR are below:

1. Load the history pickle file.
2. Output plots to compare the training and validation history.
3. Load the trained model.
4. Create a `CSVGenerator` with the preprocessed test data from the preprocessing stage.
5. Iterate through all the images and predict with the loaded model.
6. Record these detections in the following format:

   ```
   image_name x y w h score -1 -1 -1
   ```

   This is generated through the `evaluation.ipynb` file and is saved as `submission.txt`.

5. Download Piotr's Computer Vision Matlab Toolbox as it is used in the evaluation toolkit.
6. Make path changes in the `demo.m` file to point to the downloaded toolbox as well as the submission text file.
7. Run `demo.m` with Octave.

Video Detection

The video editing code can be seen in the Jupyter notebook `video.ipynb`.

The steps are as follows:

1. Record a video of hand gestures that simulate HCI.
2. Load the RetinaNet model with the trained weights from the first implementation step.
3. Create a CSVGenerator loaded with the test.csv data.
4. Create a video processor that holds the CSVGenerator that preprocesses the input images from the video. The prediction function is defined here.
5. Pass the processor to the `VideoFileClip` instance which inputs images to the processor. The processor draws bounding boxes on the detections and returns the edit image.
6. Output a new video with the edited frames.

Refinement

The initial approach for this project was to use a LeNet architecture CNN in conjunction with a sliding search window approach. Several issues made this first choice in architecture difficult to improve:

- LeNet architecture did not accept bounding boxes as input. So images of hands were cropped on-the-fly during batch generation for training, using the annotations.
- Custom non-hand “negative” dataset was needed. This was generated by cropping a random area of the image outside of the bounding box, introducing more parameters that were difficult to control.
- Sliding window size, step length, overlap ratio as well as heat mapping thresholds all introduced additional complexity to improve the LeNet model.

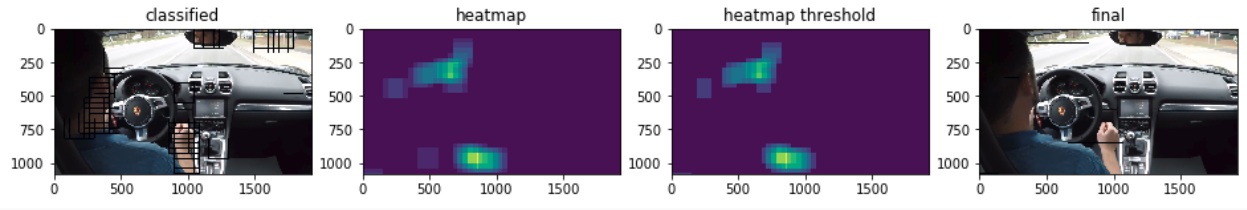

Fig. 6 Sliding window and heat mapped results using the LeNet model predictions. The detections show other parts of the driver detected up just as much as the hands.

The LeNet model did not achieve comparable AP/AR results for the VIVA Hand Detection Challenge.

The RetinaNet architecture proved to score much higher for the challenge. A significant factor that made the `keras-retinanet` framework a better choice was it’s ability to do detection rather than recognition. The first attempt achieved L1 AP/AR of 64.3/38.8 and L2 of 54.8/28.9.

Loss, AP and AR were analyzed to improve the model. The following refinements were made to improve these metrics:

- Number of epochs: Snapshots of the trained model were saved at each epoch. The benchmark RetinaNet trained at 90k iterations. The model performance was assessed at different epochs.
- Optimizer Selection: The keras-retinanet implementation uses an Adam optimizer, while the benchmark used a SGD optimizer.
- Detection Threshold: Before feeding the submissions to the evaluation kit, a detection score threshold was used to filter the detections.

Results

Model Evaluation & Validation

The AP, AR and validation loss were used to evaluate the model performance. The following parameters resulted in the highest scoring model:

- Iteration
  - Number of epochs: 21
  - Batch size: 1
  - Steps per epoch: 5000
- Loss functions
  - Classification: Focal Loss
  - Box Regression: Smooth L1 Loss
- Optimizer: Adam
  - Learning rate: 0.00001
  - `clipnorm`: 0.001
- Detection threshold: > 0.25

Scores with various parameters:

	L1 (AP/AR)	L2 (AP/AR)
Epoch 12	91.0/85.6	80.2/70.2
Epoch 18 (b)	**92.6**/90.3	82.3/71.1
Epoch 20	90.5/89.5	79.2/71.2
Epoch 21	91.0/**90.7**	78.9/**73.9**
Epoch 22	90.1/84.9	79.8/69.7
Epoch 25	83.1/80.2	71.2/63.4
Epoch 30	78.8/78.5	65.9/63.8
YOLO	76.4/46.0	69.5/39.1

Fig. 7 AP/AR scores at different epoch snapshots.
The (b) indicates parameters of the benchmark RetinaNet model.

The model scored the best AP when trained to 18 epochs (at 5000 steps per epoch). The benchmark RetinaNet also trained 90k iterations. [1] However the ranking for the VIVA Hand Detection Challenge is based on the L2 AR. This makes the model at epoch 21 the best model.

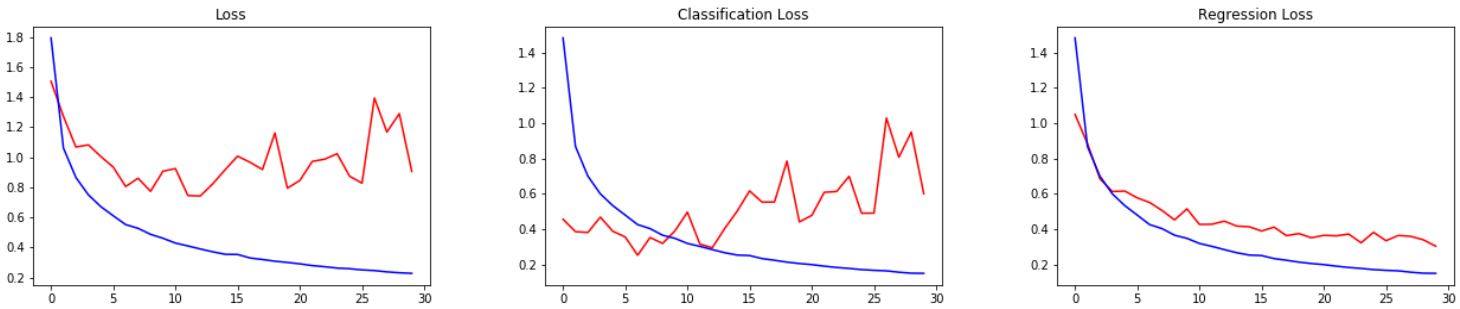

Fig. 8 Training vs Validation Loss for training up to 30 epochs.
Blue is training, red is validation.

The snapshots weights used for evaluation were based off the 'Loss' from Fig. 8. Unlike the regression loss, the classification loss does not have a predictable curve due to the focal loss algorithm. Therefore, low points in the 'Loss' history were chosen to assess the AP/AR for snapshot weights.

Justification

The AP/AR score for the RetinaNet surpassed the YOLO evaluation score in the VIVA Hand Detection Challenge leaderboard. When sorting by the AR for the L2 criteria, the model ranks 5th in the leaderboard.

When the model is applied to recorded videos outside of the challenge dataset, the model performance is not as successful.

Conclusion

Visualization

The RetinaNet's performance can be seen in the test dataset samples from Fig. 9. The detection scores are well over the threshold and the detection boxes closely mirror the ground truth annotations.

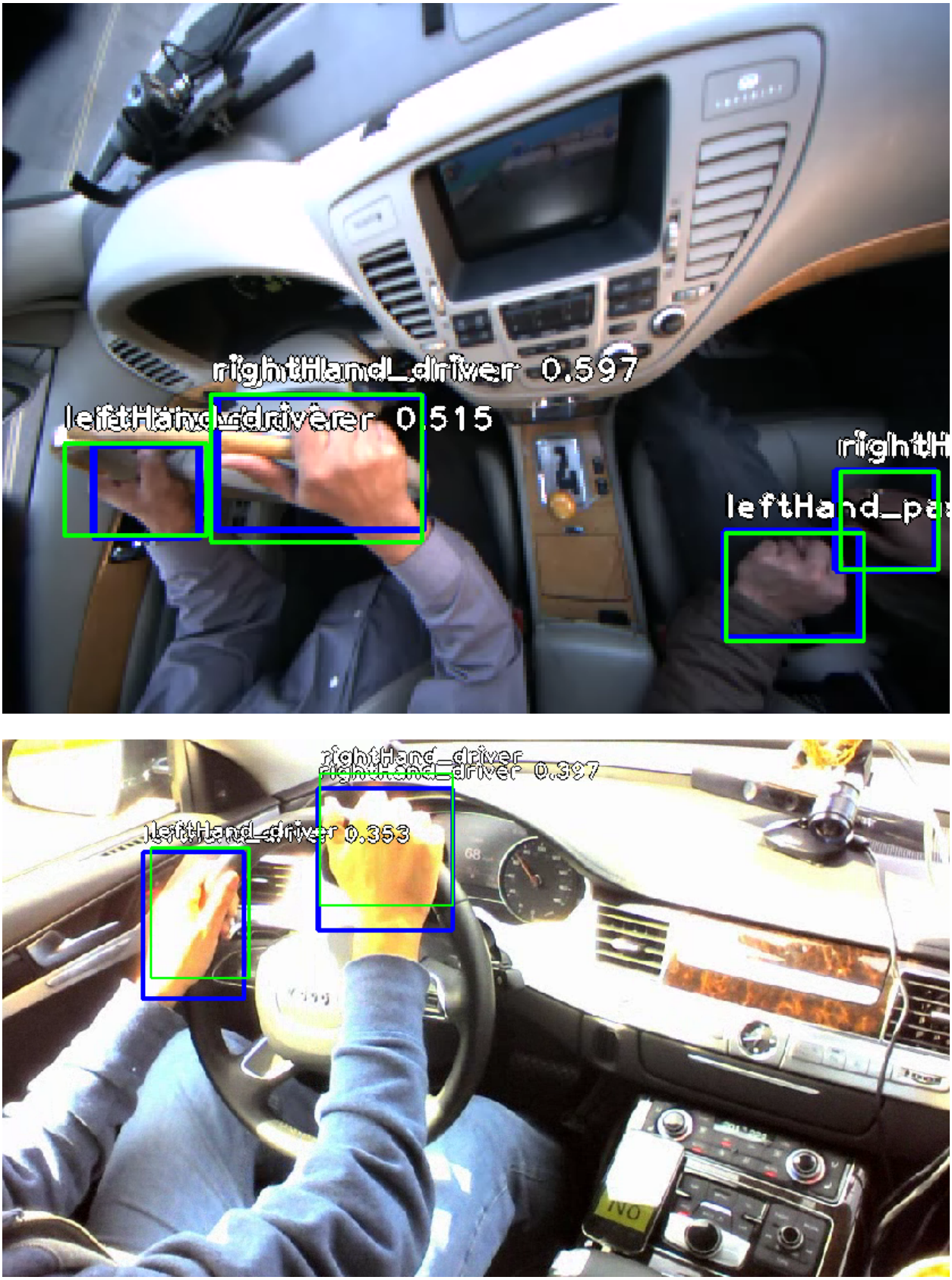

Fig. 9 Sample detection on the test dataset. The blue boxes are the detection bounding boxes and the green are the ground truth annotations.

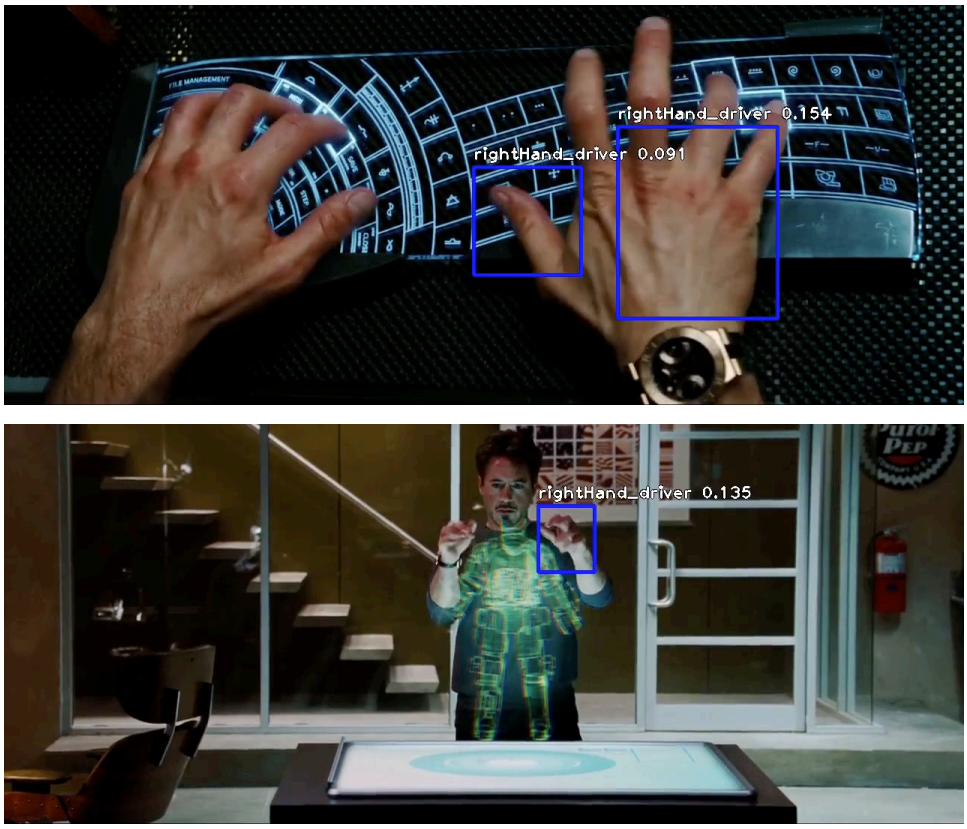

Fig. 10 Sample detections on clips from the movie *Ironman*

However, when the model is applied to images with hands that are in a different environment and resolution, the performance drops drastically. Thus, the final solution for this project is not effective for detection outside the VIVA Hand Detection Challenge.

Reflection

The process used to reach the resulting project can be summarized as below:

1. Research relevant model architectures and available technologies
2. Select model and data
3. Download and investigate data
4. Preprocess the data for training, validation and testing
5. Execute training
6. Test and evaluate trained model
7. If model architecture fails to accomplish goal, start from step 1 and repeat
8. Fine-tune parameters of the model, start from step 5 and repeat
9. Apply model to recorded video
10. Adjust detection threshold and repeat step 9

Selecting the right model architecture and evaluating the trained model were the most challenging tasks of this project. Finding a model architecture that supported object detection was a new area that required significant research outside of the course. Additionally, the VIVA Hand Detection Challenge evaluation kit did not work out-of-the-box with Octave. The code, that was written in MATLAB language, needed to be updated. Learning the new language and debugging the code was especially challenging.

However, the process of researching publications on various CNNs gave great insight into one-stage object detection systems. Furthermore, exposure to open-sourced projects for machine learning showed how to engineer such packages.

Improvement

A significant improvement to this project would be to process hand detection in real-time and interpret such gestures for a use-case. This application would have to be written in a more efficient language and be hosted on a platform with appropriate processing power.

Additionally, adding stronger image augmentation to the preprocessing stage may increase the model's effectiveness on real-world applications.

Finally, hands and gestures are much more expressive if they are not limited to a bounding box. Approaching this problem with object segmentation could significantly increase the applications for hand-based HCI.

References

1. Lin, Tsung-Yi et al. “*Focal Loss for Dense Object Detection”*. *International Conference on Computer Vision, Facebook AI Research*. 22 Oct. 2017, https://research.fb.com/publications/focal-loss-for-dense-object-detection/.
2. Redmon, Joseph et al. “You Only Look Once: Unified, Real-Time Object Detection”. *University of Washington, Allen Institute of AI, Facebook AI Research*. https://pjreddie.com/media/files/papers/yolo_1.pdf.
3. Liu, Wei et al. "SSD: Single Shot MultiBox Detector”. *UNC Chapel Hill, Zoox Inc., Google Inc., University of Michigan*. 8 Dec. 2015, https://arxiv.org/pdf/1512.02325.pdf.
4. Rosebrock, Adrian. “Intersection Over Union (IoU) for Object Detection”. *PyImageSearch*. 7 Nov. 2016, https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.
5. Girshick, Ross et al. "Rich Feature Hierarchies For Accurate Object Detection and Semantic Segmentation”. *Tech Report, UC Berkley*. 22 Oct. 2014, https://arxiv.org/pdf/1311.2524.pdf
6. Ren, Shaoqing et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. 6 Jan. 2016, https://arxiv.org/pdf/1506.01497.pdf.
7. Lin, Tsung-Yi et al. "Feature Pyramid Networks for Object Detection”. *Facebook AI Research, Cornell University, Cornell Tech*. https://arxiv.org/pdf/1612.03144.pdf.
8. Kruger, Frank. “IronMan Design”. *YouTube*. 17 Mar. 2013, https://www.youtube.com/watch?v=DZaAFADoF1M&feature=youtu.be.