DetectorShop: Democratizing Deep Learning Object Detectors

Tan Li

CMU-CS-19-121

Aug 2019

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mahadev Satyanarayanan (Chair)
Padmanabhan Pillai (Intel Lab)

Submitted in partial fulfillment of the requirements for the Degree of Master of Science.

Keywords: Computer System, Computer Vision, Deep Learning, Object Detection,

You were born with potential.
You were born with goodness and trust.
You were born with ideals and dreams.
You were born with greatness.
You were born with wings.
You are not meant for crawling, so don't.
You have wings.
Learn to use them and fly.
∼Rumi

Abstract

Deep learning object detectors have been demonstrated to be critical components in emerging artificial intelligent applications such as autonomous vehicles, cognitive assistants, and search & rescue drones. Due to the complexity of building deep learning object detectors, a toolchain for lowering the barrier has significant value. This thesis presents an end-to-end system to lower the barrier of building deep learning object detectors. The major contribution of this thesis is the design and implementation of DetectorShop, a desktop-based system that works as a PhotoShop for building object detectors.

Acknowledgments

I would like to first thank my advisor, Prof. Mahadev Satyanarayanan (Satya), for giving me the opportunity to work on this project. Studying computer systems to do cool stuffs is always what I am after for at CMU. It was extremely lucky to be able to work on this cool project and get guidance from the world-class computer system expert like Satya. I would also like to thank my mentor, Junjue Wang. To me, Junjue is like a big brother and always gives me the most sincere advice as if this was his project. Every time I had an argument with Junjue, most likely I figured that I was on the wrong side. Although frustrated, this experience is invaluable to me as I get the opportunity to deeply examine my ideas and improve them. Besides my advisor and mentor, I also need to thank Padmanabhan (Babu) Pillai for being my committee member. Outside of this thesis, Babu also gave me tremendous help, especially guided me and another PhD student through building an AR zombie gesture shooting game.

My fifth-year master journey could not happen without my undergrad academic advisor, professor Mark Stehlik, encouraging my to apply and professor Mor Harchol-Balter writing me the recommendation letter. I was lucky to get advices from people like Mark and Mor during the critical time, especially when deciding whether to directly go to industry or grad school. Looking back at the last year, especially what I've learned, I am confident that declining industry offers and went for the fifth-year program is definitely a correct decision. I also need to thank Zhuo Chen for guiding me into computer systems research in my senior year, especially taught me how to build something that no one has ever built before. The methodology Zhuo taught me is invaluable.

Finally, I want to thank my parents and my younger brother. It was my family gave me the courage and ability to pursue the life that really excites me. This thesis denotes the end of my student life at CMU, but the life of studying technologies to let this world a little better is just getting started.

Contents

List of Figures

Chapter 1

Introduction

Using a deep learning object detector to detect objects of interest is a critical step in emerging artificial intelligence(AI) applications. For example, autonomous vehicles need to detect pedestrians and traffic signs on the road; cognitive assistants need to detect whether users complete tasks correctly according to instructions; search & rescue drones need to detect people while searching over water.

Although deep learning detectors have been widely adopted, building them is still challenging for two reasons. First, building a deep learning detector is very time-consuming. Developers need to set up the coding environment, collect and label datasets, and implement object detection models. Additionally, the training and testing of a object detector are necessary. Second, the detector building process requires a deep field knowledge of object detection. This is especially true when developers need to implement object detection models by themselves. Developers need to know the usage of deep learning frameworks, detection models' details, and subtleties in tweaking models' performance. This thesis aims to lower the barrier of building deep learning object detectors.

The core contribution of this thesis is the design and implementation of DetectorShop, an end-to-end system that works as a PhotoShop for building object detectors. This system has an innovative system architecture design that efficiently manages datasets storage and can flexibly adopt specialized hardware to accelerate deep learning computation. It also provides an abstraction layer to enable users to easily make tradeoffs between speed and accuracy without a deep understanding of object detection algorithms. Finally, DetectorShop has a user workflow that enables users to build object detectors through a user-friendly GUI, entirely without coding.

1.1 Thesis Statement

In this thesis, I present a solution of lowering the barrier of building customized deep learning object detectors with private datasets. In particular, I claim that

> Lowering the barrier of building deep learning object detectors can be achieved by factoring out low-level implementations and common operations to an end-to-end specialized tool. This tool enables users to easily make tradeoffs between speed and accuracy in object detection models and flexibly adopt specialized hardware to accelerate deep learning computation. Additionally, this tool efficiently manages datasets storage.

The main contributions of this thesis are as the following:

- I design and implement a user workflow, which enables users to build object detectors through a user-friendly GUI, entirely without coding.
- I design and implement a system architecture, which efficiently manages storage for datasets and is able to flexibly take advantage of specialized hardware.
- I design and implement an abstraction layer, which enables users to easily make tradeoffs between speed and accuracy in object detection models.

1.2 Role of End-To-End Tools

There are many ongoing efforts in building the toolchain for building deep learning object detectors. TPU by Google and ASICs from many startups could be thought as efforts on the hardware level. These specialized hardware speed up the iteration speed by accelerating the deep learning computation. Additionally, Tensorflow by Google and Pytorch by Facebook represent efforts on the framework level. These frameworks provide users with specialized programming models for deep learning. End-to-end tools are the kind of tools that built on top of the previous mentioned efforts and factor out low-level implementations and common operations, just like PhotoShop. This is also the rationale behind the naming of DetectorShop, the PhotoShop for building object detectors.

1.3 Thesis Overview

Chapter 2 offers the background of the thesis, including the applications of object detection, object detection algorithms, and deep learning transfer learning. Chapter 3 focuses on the details of the user workflow, and chapter 4 covers the system architecture design of DetectorShop. Chapter 5 provides details and the rationale of the abstraction layer for making tradeoffs between speed/accuracy in object detection models. Details about the implementation cloud be found in chapter 6. Chapter 7 ends the thesis with a conclusion and a list of future works.

Chapter 2

Background

2.1 Object Detection Applications

Deep learning has enabled many exciting applications, for examples, autonomous vehicles, cognitive assistants[Chen, 2018], and intelligent search & rescue drones. Among all these applications, a reliable and accurate object detector is a critical component. By definition, object detection is a task of detecting objects of interest in the image or video stream.

For example, an autonomous vehicle requires an object detector to detect traffic signs and pedestrians to drive safely (figure 2.1).

Figure 2.1: Autonomous vehicle using object detector to detect traffic signs and pedestrians on the road

Similarly, a cognitive assistant needs an object detector to detect whether the task is completed correctly and then give corresponding instructions to the user [Chen, 2018]

(figure 2.2).

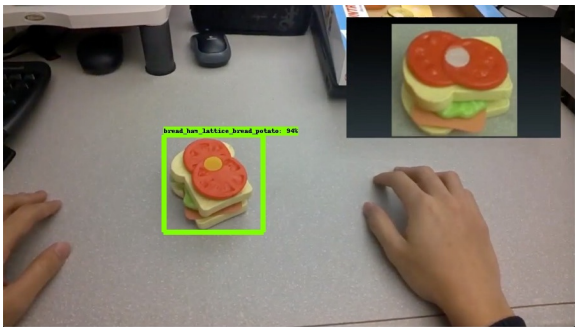

Figure 2.2: Cognitive Assistant using object detector to check whether the user has assembled the sandwich correctly according to the top right instruction

In the search & rescue drone use case, the drone requires an object detector to search for people over the water (figure 2.3).

Figure 2.3: Drone using object detector to search for people over water

2.2 Object Detection Algorithms

In general, any object detection task could be divided into two subtasks: localization and classification. In other words, to do object detection on an image, we need first to locate the object in the image and then classify what that object is, i.e., cat, dog, or a human (figure 2.4).

Figure 2.4: Object Detection = Localization + Classification

For an image containing multiple objects of interest, an object detection model should be able to localize and classify all the objects of interest in the image. (figure 2.5)

Figure 2.5: Object Detection on an image containing multiple objects of interest, i.e. Person, Car, Traffic Signs...

2.2.1 Traditional Object Detection Algorithms

Traditional object detection algorithms normally contain three major steps: region proposal, feature extraction, and classification. Region proposal is responsible for proposing all the possible regions that contain objects of interest. Since the object of interest could appear at any location of the image with any arbitrary size, most algorithms adopt a sliding

window strategy. After the regional proposal, most algorithms extract the features of the potential regions and feed the extracted features into the classification step. In the feature extraction stage, SIFT [Lowe, 2004] and HOG [Dalal and Triggs, 2005] are the popular adopted feature extraction algorithms. As for classification, SVM [Cortes and Vapnik, 1995] and AdaBoost [Schapire, 1999] are widely adopted.

However, the performance of traditional object detection algorithms is not satisfying in terms of both speed and accuracy. For speed, the region proposal step is the bottleneck. The reason is that the sliding window strategy needs to slide through the whole image with all the possible window sizes, which is very time consuming and produces a lot of redundant potential regions. The redundant potential regions further slow down the algorithm due to the unnecessary feature extraction and classification of redundant regions. As for accuracy, the feature extractor is the bottleneck. Most feature extractors are sensitive to the circumstance, i.e., lighting condition. The instability of feature extractors leads to a poor accuracy in the classification stage.

2.2.2 Deep Learning based Object Detection Algorithms

In the past several years, deep learning has dominated the research of computer vision. In almost all the computer vision tasks, deep learning based approaches dramatically outperform the traditional approaches. Take the ImageNet ILSVRC classification challenge for example, starting 2012, all the winners' models are based on deep learning (figure 2.6).

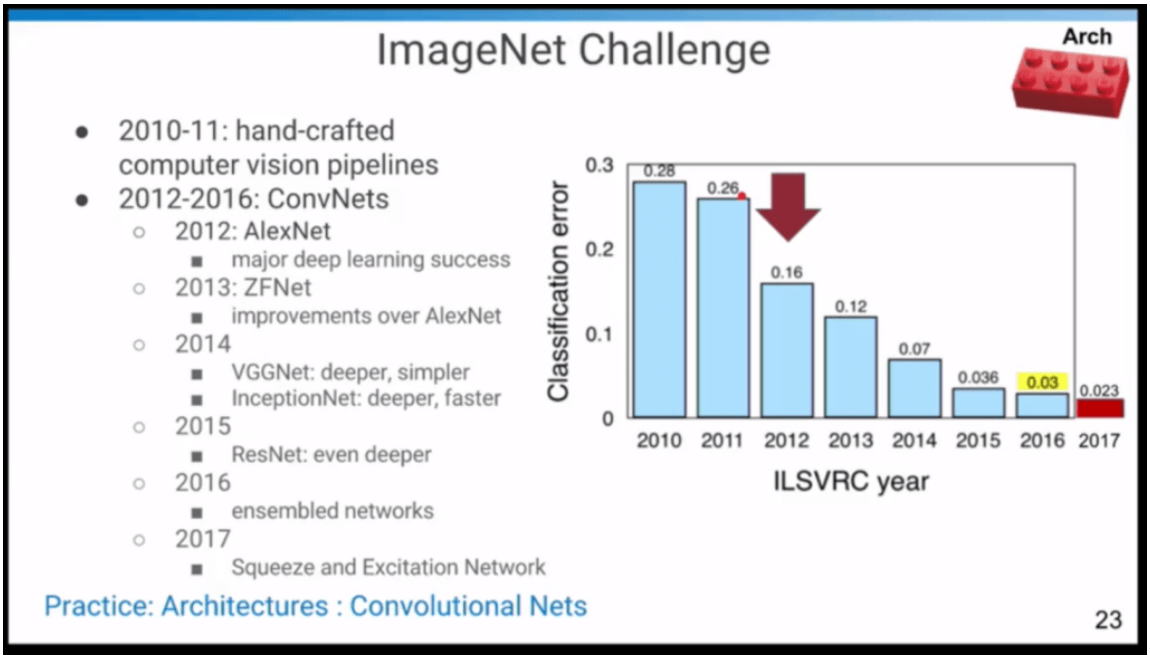

Figure 2.6: ImageNet challenge winners and their corresponding architecture and performance. (source:[nip])

At high-level, many of the leading state of the art models have converged on common deep learning architectures. For example, YOLO [Redmon et al., 2015], SSD [Liu et al., 2016], CornerNet [Law and Deng, 2018] represent models with one-stage architecture, while R-CNN [Girshick et al., 2013], FCN [Long et al., 2014], Faster R-CNN [Ren et al., 2015] represent models with two-stages architecture. We call these architectures meta-architectures. These two common meta-architectures have different tradeoffs between speed and accuracy. Typically, one-stage models have simpler neural network structures and get rid of the region proposal stage. On the other side, two-stages models first generate region proposals and then classify the regions along with modifying the regions to fit the object of interest better. Due to the more complex structures, two-stages models usually have a lower speed but higher accuracy than one-stage models. Each meta-architecture

adopts a image classification model as the feature extractor, e.g., Resnet, Mobilenet. Such combinations are often referred as feature extractor + meta-architecture, such as Resnet + Faster-RCNN, Mobilenet + SSD.

One-Stage Meta-Architecture Models

One-stage models take the ground truth (labeled data) and encode the objects' classification and localization information into a single target. Based on different encoding methods, different models have different loss functions for quantifying the difference between models' predications and the ground truth. The design of the loss function is a critical part of the object detection model design. In general, the smaller the loss is, the better the model performs. The training of one-stage models is done through a regression task to reduce the loss between models' output and the target (figure 2.7). In this thesis's implementation, to achieve the real-time speed performance, we focus on SSD as the representative of one-stage architecture models.

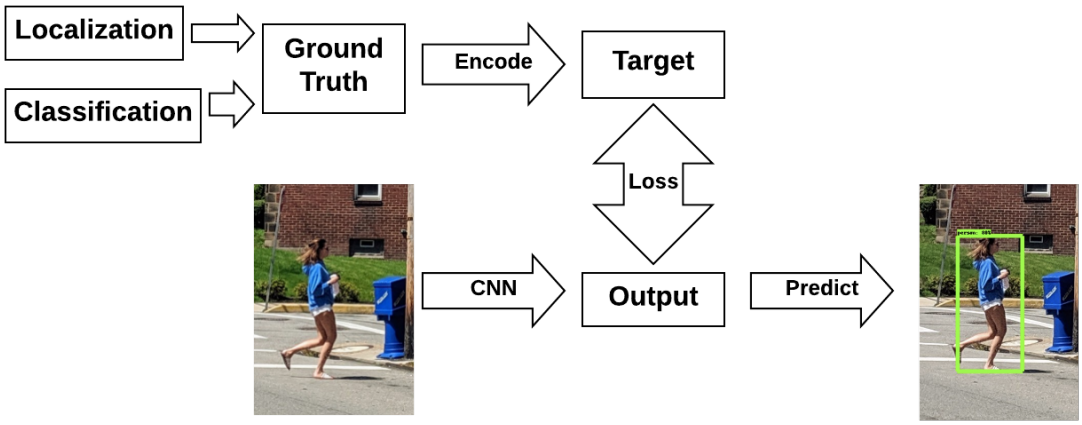

Figure 2.7: High-level abstraction for One-stage Architecture Models

Two-Stages Meta-Architecture Models

Two-stages models could be perceived as the composition of two one-stage models. Two-stages models first adopt a similar process as the one-stage model to train a region proposal network (RPN), instead of the final model. With the result from the RPN, two-stages models apply another one-stage model to each of the regions of interest (ROI). In this

thesis's implementation, to achieve the state of the art accuracy, we adopt Faster-RCNN as the representative of two-stages architecture models. We provides combinations of meta-architectures and feature extractors like Faster-RCNN + Resnet, Faster-RCNN + Inception Net, and Faster-RCNN + NAS.

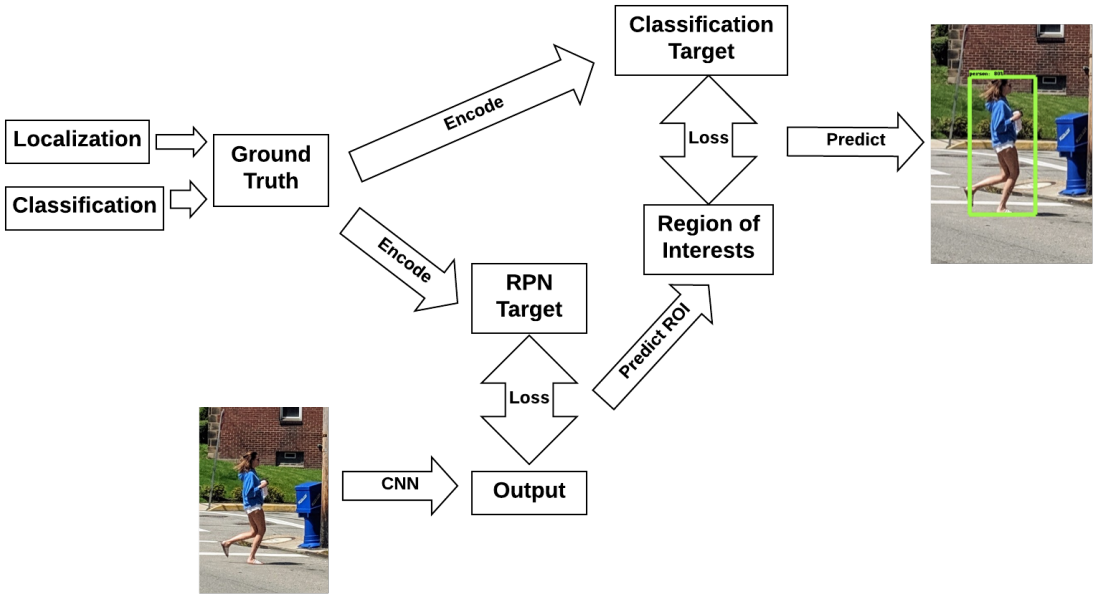

Figure 2.8: High-level abstraction for Two-stages Architecture Models

2.3 Transfer Learning

Transfer learning is a technique to reduce the amount of necessary data for deep learning training. In practice, training a deep learning model from the ground up requires a lot of data. Most of the state of the art accuracy models are trained on super large datasets like COCO (Common Objects in Context). COCO contains 330k images and over 1.5 million object instances. It is unlikely for individual users to have such a large dataset. On the other side, the backbone network layer of a model doesn't change a lot at the later period of the training, for examples, the RPN or the Feature Extractor. Therefore, to build customized object detectors, instead of training the model from ground up, people usually use a pre-trained model as the base model and train the critical layers to make sure the model performs well on their personal datasets. This technique is also adopted in the

implementation of DetectorShop.

Chapter 3

User Workflow

The user workflow is designed for factoring out low-level code implementations and common operations. Users focus on making tradeoffs between speed and accuracy in object detection models, instead of implementing models and setting up environments. Additionally, this user workflow takes advantage of labeled datasets' reusability. The same labeled dataset could be used for generating different object detectors with different tradeoffs.

3.1 Workflow Overview

Although DetectorShop is a distributed system, users can install DetectorShop as a desktop application. To access this system, all the user needs to do is downloading the application from the internet. Once the desktop application and docker-supervisors are installed, everything is self-contained, no extra installation is required. If the user wants to take advantage of a remote machine, for training for example, the user needs to make sure that the remote machine is installed with docker-supervisor and the docker deamon is running appropriately beforehand.

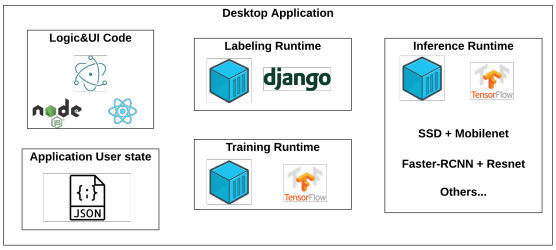

Figure 3.1: Desktop application containing multiple containers as feature runtimes

The high-level user workflow can be described as follows:

- Specify datasets, either on the local machine or on a remote server.
- Label a dataset using the data labeling module.
- Pick a labeled dataset, make model tradeoffs, and start model training.
- Monitor the progress of model training and manually decide when to stop training
- Test the performance of object detector by a live camera stream or a recorded video.

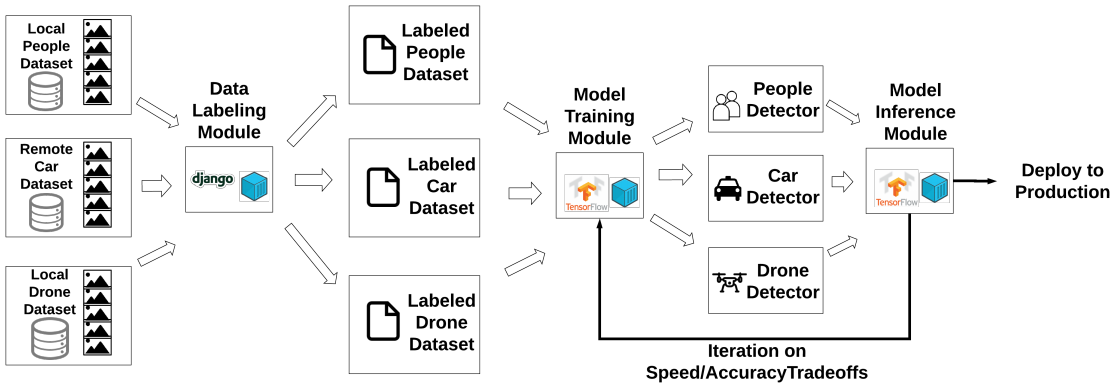

Figure 3.2: High-level user workflow

3.2 Specify and Label Datasets

To specify a dataset, users need to first indicate whether this dataset is local or remote. For local datasets, users just need to specify the folder path containing all the data. For remote

datasets, users need to specify the following information:

- URL for accessing the remote machine through SSH
  - e.g. cloudlet002.elijah.cs.cmu.edu
- Path of the folder containing all the data
  - e.g. `~/datasets/cognitive_assistant/`
- SSH credentials for accessing remote machine
  - i.e. username & password or ssh private key

After users specify the dataset, users are able to label the dataset using the labeling module. This module will open a new window and provide a set of rich UI labeling tools, e.g. drawing bounding boxs on images (figure 3.3). No matter where the dataset is located, the user experiences are the same. The design of the system architecture achieving this is provided in the section 4.3.1.

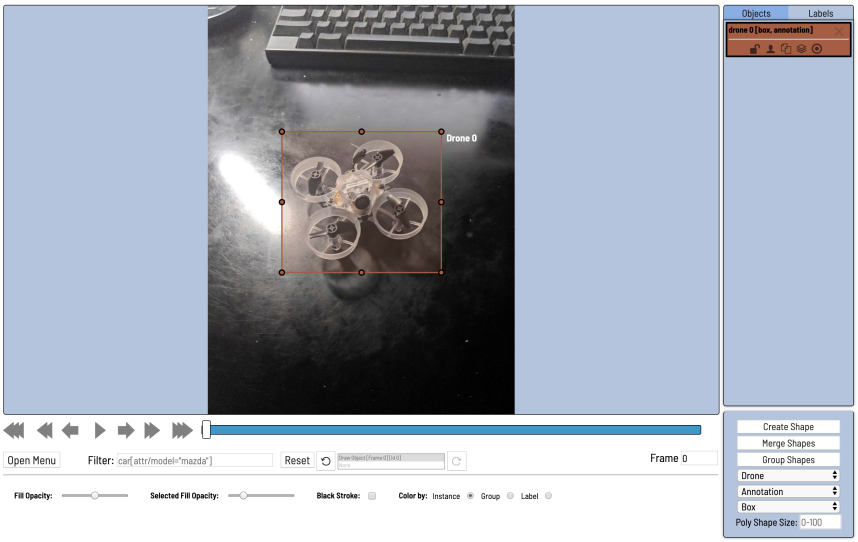

Figure 3.3: Labeling Screenshot in DetectorShop

3.3 Choose the Model, Make tradeoffs and Start Training

With a labeled dataset, users are able to generate a customized object detector. When users click the new object detector button, the system will ask users for the following information:

- The labeled dataset to build object detector
- Choice of tradeoffs between speed and accuracy
  - Feature Extractor + Meta Architecture
    - e.g. mobilenet + ssd, resnet + faster-rcnn...
  - Grid Size (One-Stage Models)
    - e.g. small, medium, large
  - Number of Region Proposals (Two-Stage Models)
    - e.g. 100, 300, 500...
- The machine to train the model
  - SSH credentials and URL are required if the machine is remote
  - e.g. a remote cluster equipped with Nvidia GPU

Once users finish the configuration and start training the model, users are able to keep track of the training progress through a training dashboard. This dashboard is a plugin in Tensorflow which provides critical information such as loss rate, learning rate to users. Based on these information, users are responsible for deciding when to stop the training (figure: 3.4).

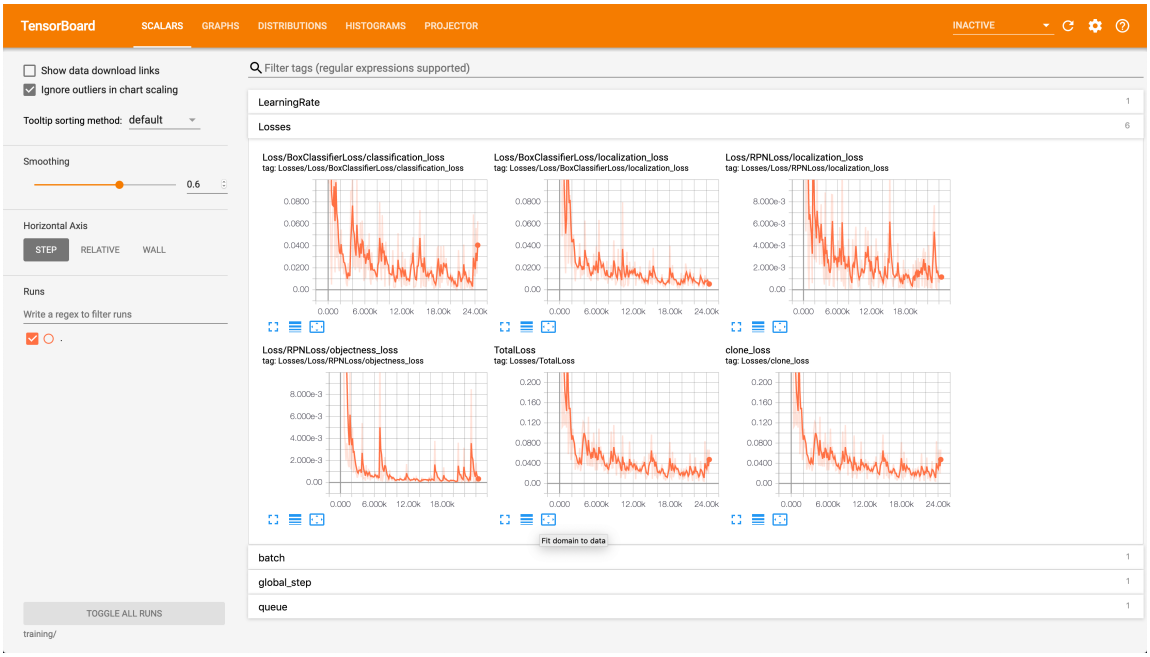

Figure 3.4: The dashboard showing the current loss value of the training
(Users can configure the dashboard to show other information through the GUI)

The rationales behind the configurable options for tradeoffs, such as the meaning of grid size and number of region proposals are provided in section 4.4. With the ability of choosing different machines to train models, usres can flexibly adopt newly released specialized hardware. Otherwise users have to physically purchase the new hardware and install it to the system. The system architecture design achieving flexible adoption of new hardware is provided in section 4.3.2.

3.4 Experiment with the Detector and Check the Performance

With a freshly built object detector, getting an intuitive impression of the detector performance is desirable. In DetectorShop, users are able to experiment with the object detector by specifying the following:

- The object detector to experiment with
- The machine to run the object detector
  - SSH credentials and URL are required if the machine is remote
  - e.g. a remote cluster equipped with TPU Edge
- The testing data to run the detector on
  - e.g. a pre-recorded video or the live camera video stream

With the feedback from the experiments, users are able to iterate on their choice of tradeoffs between speed and accuracy.

Figure 3.5: Experiment the Object Detector with Real-Time Camera Stream
(Lag for detecting the moving object, i.e., person. Need more speed)

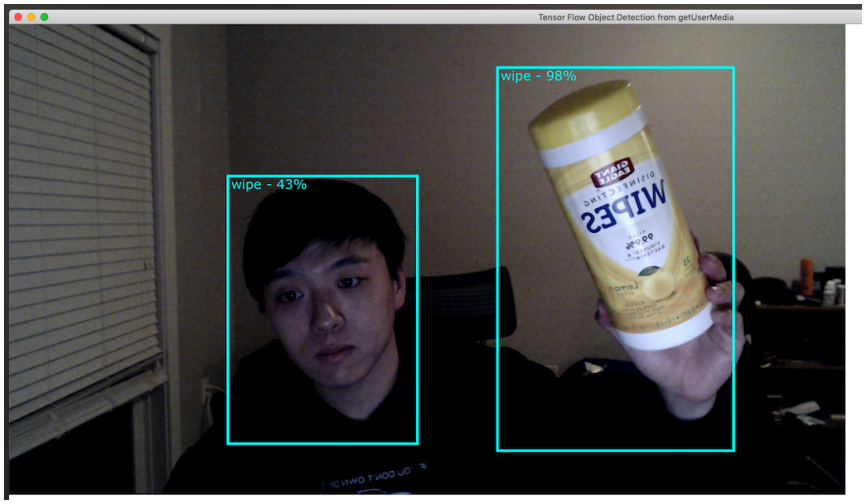

Figure 3.6: Experiment the Object Detector with Real-Time Camera Stream
(False detection of wipe, detects person instead. Need more accuracy)

Additionally, with the ability to choose different hardware platforms, users can easily see the difference between running the same object detector on two different platforms, i.e. CPU machine vs TPU-Edge machine. Finally, once the detector is acceptable in terms of both speed and accuracy, users can export the detector and use it in final applications, such as autonomous vehicles, cognitive assistants, search & rescue drones.

Chapter 4

System Architecture

4.1 Design Constraints

4.1.1 Large Datasets

Although there isn't a clear rule indicating how much data is sufficient to train a deep learning object detector, a large dataset is always necessary. Most state of the art object detection models are trained on COCO dataset, which contains 330k images and over 1.5 million object instances. Currently, almost all others end-to-end systems require users to upload the dataset into the system. Uploading datasets will cause unnecessary duplications. Meanwhile, the bandwidth consumption is significant to transmit large datasets. DetectorShop avoids unnecessary datasets duplication by pushing computation to the dataset instead of pulling the data into the system.

4.1.2 Flexible Adoption of Specialized Hardware

As an end-to-end system, one goal is to enable users faster iteration in building deep learning object detectors. However, the deep learning training is very time consuming. Take Faster-RCNN for example. Faster-RCNN with 10000 images could easily take weeks to converge on a CPU-only machine.

To accelerate the iteration of building object detectors, adopting specialized hardware to accelerate the training is necessary. There are many ongoing efforts in building specialized hardware for deep learning, TPU for an example. The state of the art hardware keeps

rapidly evolving. Always buying the latest hardware and plugging it into the system is not feasible. Meanwhile, a lot of specialized hardware are exclusive. For example, TPU is only accessible through Google Cloud Platform(GCP). DetectorShop solves this challenge by pushing deep learning computation to computing platforms with specialized hardware.

4.1.3 Cost of System Operation

To fully democratize the power of object detectors, the system should also take care of the system operation costs. Most of others similar systems are web-based tools. In other words, there is an server running 7*24 in such systems. Keeping a server running 7*24 is very costly, normally requires a dedicated operation team. This is the major reason that most similar systems only get deployed in big tech companies and research labs. DetectorShop reduces the operation cost by packing all the functionalities into a desktop application and make the setups on remote machines minimal, i.e. only need to install docker-supervisor.

4.2 Resource-Centric System Architecture Design

This section first illustrates the current approach adopted in most similar systems, and then discusses the new approach adopted in DetectorShop. Most similar systems are web-based tools. In those systems, users need to upload their datasets into the system. To adopt the newly released specialized hardware, users need to purchase the hardware and plug it into the server.(figure 4.1)

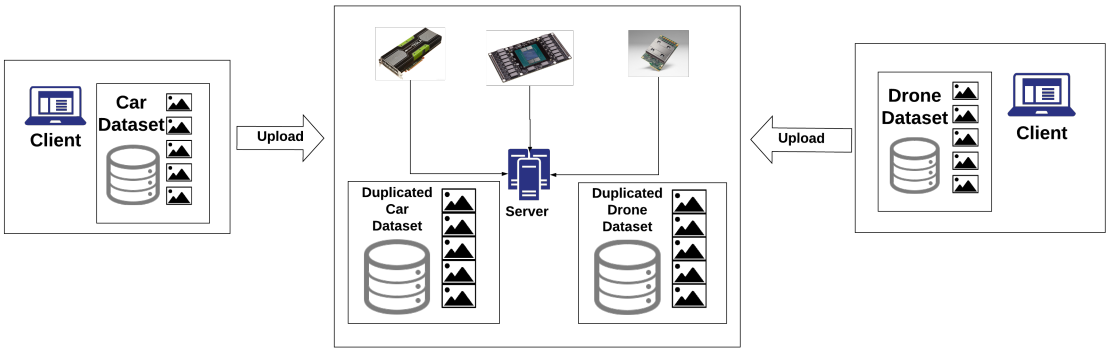

Figure 4.1: Normal approach in similar end-to-end systems

There are many interpretations of the current approach's prevalence. One interpretation is that implementation(code) is always the biggest cost in developing such systems. In other words, the codebase is the most valuable resource in the system. Keep the code running and manage the codebase are challenging. However, as there are more and more frameworks and development tools emerging these days, implementation cost is reduced dramatically. Sometime, the data storage or specialized hardware become the biggest cost in building such systems. Based on this idea, this thesis proposes a resource-centric system design, that is the system should be built around the most valuable resource in the system, i.e., datasets, specialized hardware.

This thesis practices the resource-centric system design in two specific examples, data-centric and hardware-centric. Data-centric design is adopted in the data labeling module. The goal is to eliminate the step of uploading data. Therefore, this system avoids large bandwidth consumption and unnecessary dataset duplication. The hardware-centric design is adopted in the model training and model inference modules. With the hardware-centric design, the system can flexibly adopt newly released hardware to accelerate deep learning computation. Such flexibility has significant meaning since the state of the art hardware keeps evolving rapidly.

4.2.1 Data-Centric Design

Instead of pulling data from the data server and then upload to the labeling module (figure 4.2), the data-centric design pushes the labeling runtime to the remote data server (figure 4.3). All users' labeling operations are done through HTTP and the pushed labeling runtime can access the dataset with local machine performance. Therefore, there is no need to upload the dataset anymore. Such mechanism is implemented by container and file system mapping. Implementation details are in chapter 6.

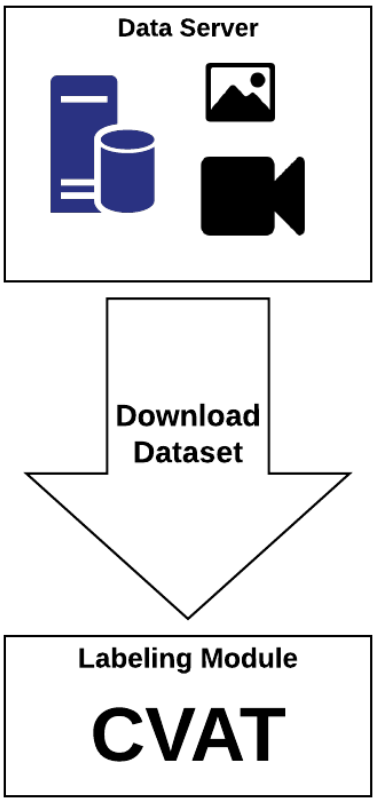

Figure 4.2: Previous Labeling Module Design

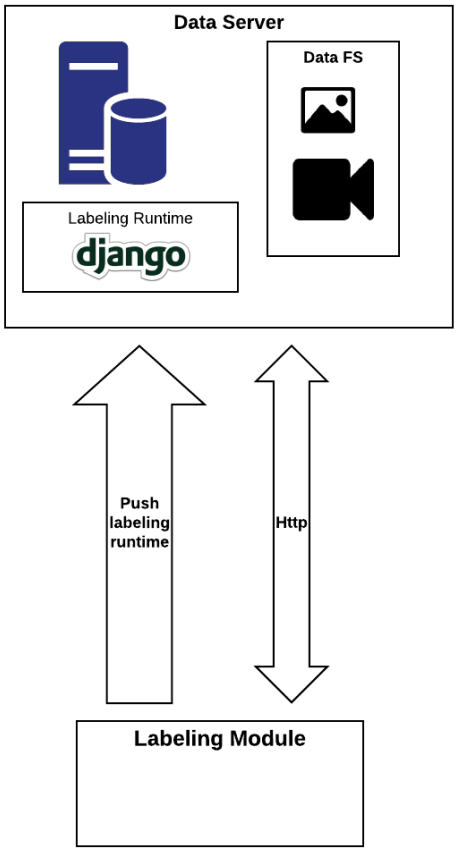

Figure 4.3: Data-centric Labeling Module Design

4.2.2 Hardware-Centric Design

Instead of plugging more and more hardware to the server running the system codebase (figure 4.4), hardware-centric design pushes the deep learning computation to remote machines with specialized hardware (figure4.5). Therefore users could flexibly adopt newly released hardware without actually purchase it and plug into the system. All users need is the access to a machine with the latest specialized hardware. Implementation details are shown in the chapter 6.

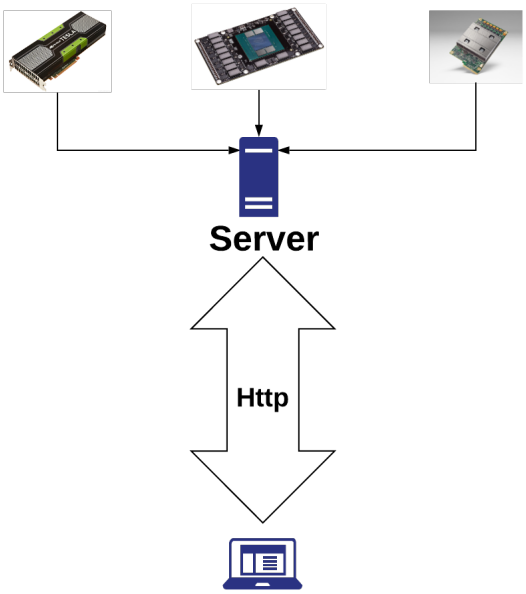

Figure 4.4: Previous Model Training/Inference Module Design

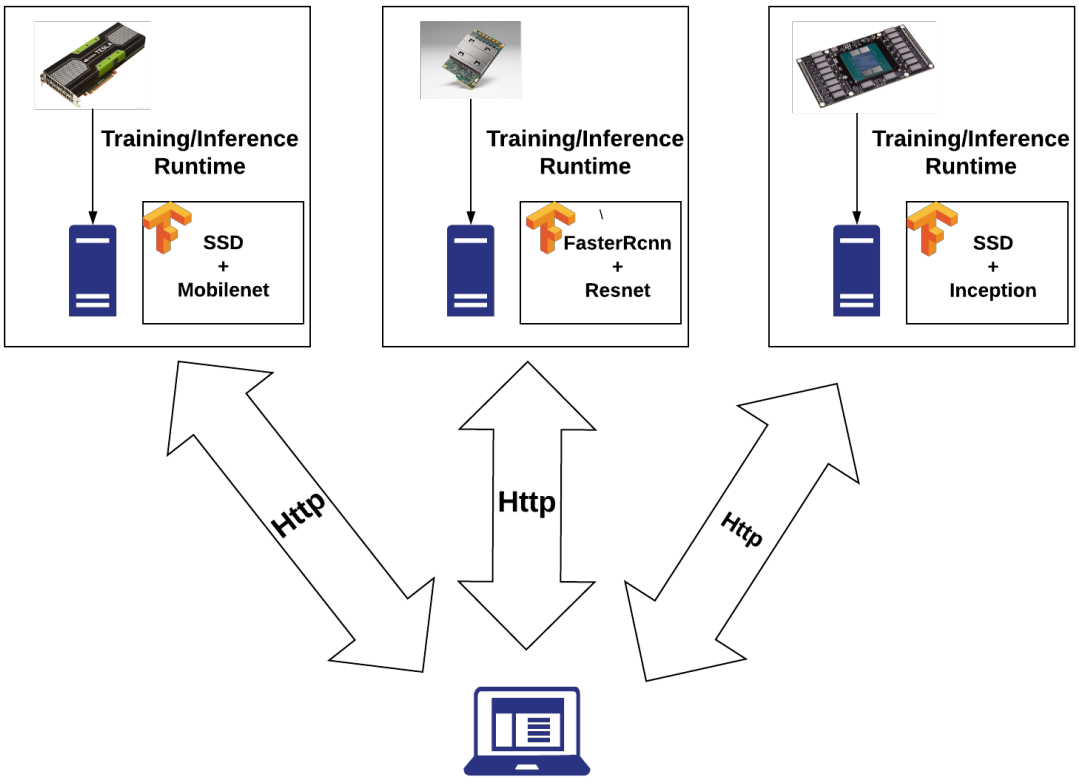

Figure 4.5: Hardware-centric Model Training/Inference Module Design

4.2.3 Software Distribution

DetectorShop is delivered as a cross-platform desktop application. All the major functionalities, such as labeling module, training module and inference module are encapsulated as containers. DetectorShop will expand itself into a distributed system when user need to access remote datasets or adopt remote hardware accelerators. Once users finish the remote task, DetectorShop will automatically cleanup itself, i.e. release the remote hardware accelerator, deleted temporary files, close the http connection. Therefore, there's no need to maintain a server that keeps running at all time.

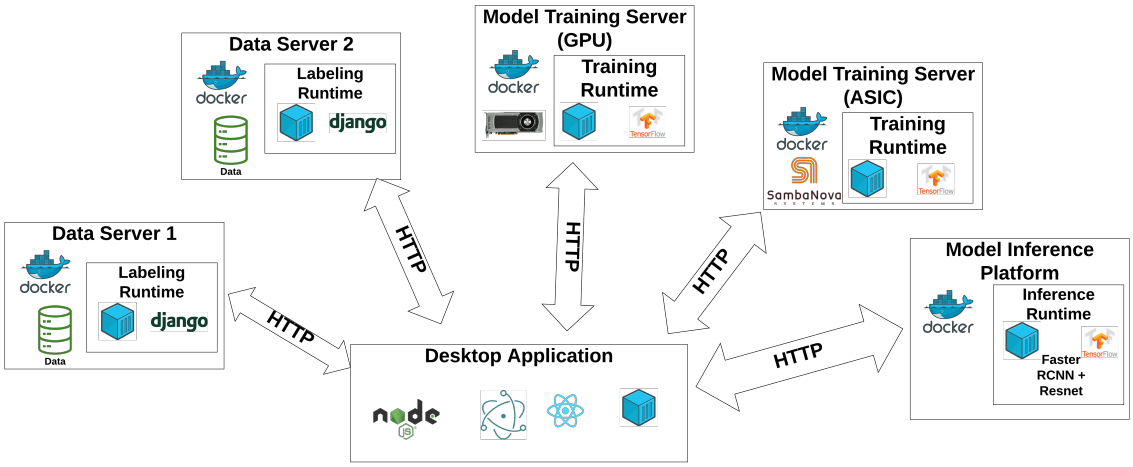

Figure 4.6: Transform from a desktop application into a distributed system

4.3 Remarks and Basic Evaluations

One thing need to mention is that the hardware-centric design depreciates the benefit of data-centric design to some extent. The reason is that in the hardware-centric design, we push the training files to a remote machine which is equipped with specialized hardware. Those training files include bytes of original data. In other words, the training files include a duplication of the original dataset. However, if the data server happens to be equipped with specialized hardware, then this is not a problem anymore. Although DetectorShop doesn't eliminate data duplication completely, DetectorShop provides the flexibility to users to eliminate data duplication entirely when it is possible.

To get an understanding of the benefits brought by the data-centric design, we consider the following usecase: A user need to label three datasets which locate on a single remote data server. Each dataset is 10 Gigabytes. Then the bandwidth consumption by DetectorShop versus traditional data uploading approach is as followings:

Bandwidth Consumption for Labeling Datasets		
Operations	DetectorShop	Data Uploading
Label the First Dataset	2.4 GB	10 GB
Label the Second Dataset	1.4 KB	10 GB
Label the Third Dataset	1.4 KB	10 GB

The reason for the relative large bandwidth consumption for labeling the first dataset is that we need to ship the labeling module docker image to the remote machine. Once that image

is shipped, the docker supervisor will cache the image and there's no need for retransmission in the following labeling operations.

It is trivial that specialized hardware will accelerate the deep learning training. To verify this idea, we compared running deep learning training on different platforms, i.e. Nvidia Tesla K40C GPU and Intel Core I7 CPU. The result is as following:

Time to achieve loss <= 0.02 with Transfer Learning on 2000 images		
Models	Tesla K40C (GPU)	Core I7 4 cores (CPU)
MobileNet + SSD	6 hours	18 hours
ResnetNet + FasterRCNN	8 hours	26 hours

4.4 Similar Ideas in Industry

During the development of the thesis project, Microsoft released the remote development module for VScode[vsc]. To adopt specialized hardware, such as mutli-core processors for accelerating compilation, users sometime want to develop their code remotely. However, most current approaches, such as network file system mapping, only fetch the actively editing files to users' machine and sync those files with the remote server. Such approaches don't have the knowledge of the whole codebase, therefore users are not able to use some advanced functionalities, such as go-to jumping. In VSCode, instead of fetching the actively files to users' laptop, VSCode pushes a server to the remote machine and let the server operate on the codebase in behalf of users (figure 4.7). Although this usecase is completely different from DetectorShop, the technical challenges and solutions are quite similar.

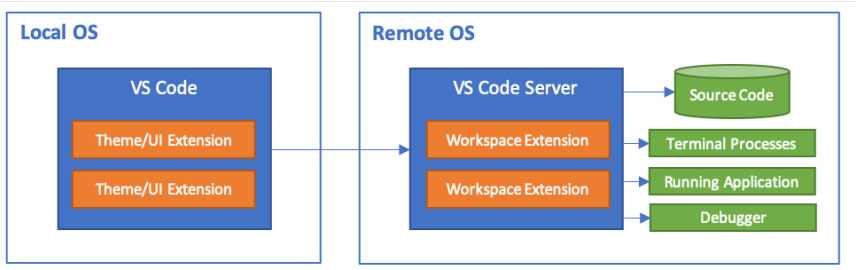

Figure 4.7: VSCode Remote Development Module (source:[vsc])

Chapter 5

Abstraction Layer for Making Speed/Accuracy Tradeoffs

Different from most computer vision research which targets only at achieving state of the art accuracy, object detectors produced by DetectorShop should also address speed. On the one hand, DetectorShop should be able to produce state of the art accuracy. On the other hand, DetectorShop should also be able to generate object detectors with realtime speed. Users need to make tradeoffs between speed and accuracy based on the specific use case. DetectorShop enables users to easily make tradeoffs between speed and accuracy by providing an abstraction layer.

5.1 Appropriate Abstraction Level

An appropriate level of abstraction is the fundamental factor in the design. That is what the lowest level of abstraction exposed to the users should be? Should it be model, neural network layer, or even tensor?

The most naive approach is providing a model level abstraction, that is treating models as black boxes. There are many open-sourced implementations of different models, and most codebases provide some callable scripts to run the model. Ideally, we could plug the system with different model implementations and produce object detectors with the corresponding performance. However, relying on the model level abstraction is not enough. Even for the same model, different implementations could generate object detectors with completely different performance. On extreme cases, some poorly implemented accuracy oriented models, e.g., Faster-RCNN have lower accuracy than a speed oriented

model, e.g, Yolo, and vice versa. Fortunately, at least at high-level, many of the leading state of the art models have converged on common deep learning architectures. For example, SSD, YOLO represent models with one-stage architecture, while R-CNN, Faster R-CNN, FCN represent models with two-stages architecture. Meanwhile, there are some high-quality open-sourced toolboxes for object detection released by big tech companies, such as [Huang et al., 2016] and [Chen et al., 2019].

Taking advantage of the converged architectures and the high-quality toolboxes, DetectorShop fixes the model implementation and provide users ability to make tradeoffs between speed and accuracy. In other words, users cannot plugin their own models but are able to make tradeoffs between speed and accuracy through a model abstraction proposed by DetectorShop.

Inherently, there's a tradeoff between one-stage and two-stage models, that is one-stage models have higher speed but lower accuracy while two-stages models have higher accuracy but lower speed. Therefore, different from other similar works [Huang et al., 2016] [Chen et al., 2019], focusing on modularizing object detection models, the focus of this model abstraction is enabling users to trade accuracy for speed in two-stage models and trade speed for accuracy in one-stage models.

In the proposed model abstraction, a model is specified with the following components:

- Feature Extractor (e.g., Mobilenet, Resnet, VGG...)
- Meta Architecture (e.g., SSD, Yolo, Faster-RCNN...)
- Grid Size (One-Stage Models) (e.g., small, medium, large)
- Number of Proposal (Two-Stage Models) (e.g., 100, 300, 500,...)

The configuration space can be summarized as the following:

Different Configurations for Object Detection Models			
Feature Extractor	Meta Architecture	Grid Size	Number of Proposals
Mobilenet	SSD	Small	100
Resnet	Yolo	Medium	300
VGG	Faster RCNN	Large	500
			800
			...

5.2 Feature Extractor + Meta Architecture

As demonstrated in the section 2.2, both one-stage models and two-stage models use a image-classification model as a feature extractor to extract a feature map of the input image. All the following steps in the algorithm are applied on the feature map instead of the original input image. By intuition, the better the performance of image-classification model is, the better the performance of the object detection model is. [Huang et al., 2016] has shown that there's indeed an overall positive correlation between the classification performance and the object detection performance. Since the better feature extractor performance requires more complicated image classification models, the speed of the overall object detection model become lower. In this sense, making feature extractor + meta architecture as an configurable option is a natural decision in the abstraction design.

5.3 Grid Size (One-Stage Models)

Although [Huang et al., 2016] has shown that there's indeed an overall positive correlation between the classification performance and the object detection performance, [Huang et al., 2016] also points out that such correlation is only significant in two-stage models. For one-stage models, especially SSD, such positive correlation is not significant. Therefore, the options provided in section 5.3 are not sufficient to enable users to trade speed for accuracy in one-stage models.

Based on the knowledge of the current one-stage models, grid size is a good option to enable users to trade speed for accuracy in one-stage models. This is because all the one-stage models have a common step, that is partitioning the input image into grids. These grids are used as default regions of interests. The bigger the grids are, the faster the speed is but the lower the accuracy becomes. Because it is harder for the models to detect small objects. On the other hand, the smaller the grids are, the better the accuracy of detecting small objects, but lower the speed. This is because there are more grid cells in the image and models require more computation.

Different models have different approaches of partitioning the image into grids. In the Yolo's family, the models explicitly partitions the image into S*S grids (figure 5.1). On the other side, SSD partition the image by producing feature maps with different resolutions (figure 5.2).

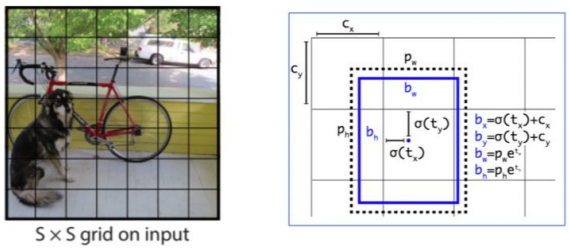

Figure 5.1: Yolo partitions the image into S*S grids (source: [Redmon et al., 2015])

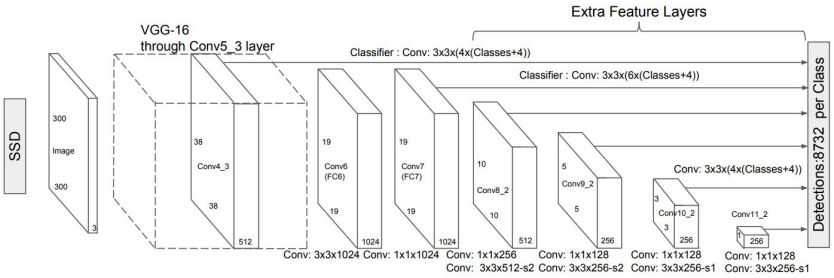

Figure 5.2: SSD uses multiple feature maps with different resolutions (source: [Liu et al., 2016])

Therefore, in the proposed abstraction, instead of asking users for the specific grid size, we only ask users for the relative info of the grid size, for example, small, medium or large. For the medium option, we use the default value specified by the original models' paper. The small and large options correspond to the 50% and 150% of the default value respectively.

5.4 Number of Region Proposals (Two-Stages Models)

As described in section 2.2, all the two-stage models have a common step for proposing regions of interest. As pointed out in [Ren et al., 2015], region proposal is one of the most expensive steps in two-stage models. A smaller number of proposed regions could accelerate the step of region proposal. Also, fewer region proposals reduce the computation for the following detection network. Therefore, reducing the number of region proposals provides the possibility of trading accuracy for speed in two-stage models.

According to the empirical work of [Huang et al., 2016], the number of proposals could be dramatically reduced without hurting accuracy excessively. Since each region only contains one object in the setup of most models, we ask users to specify the exact number of region proposals based on users' knowledge about the final use case. For example, if a user wants to deploy the object detector in an autonomous vehicle to detect traffic signs on empty streets, then he might want to specify 300 proposals at the beginning, since he is confident that it is extremely unlikely that there will be over 300 traffic signs on the street simultaneously. After several iterations, the user might be able to reduce the original 300 proposals to 30, with acceptable accuracy.

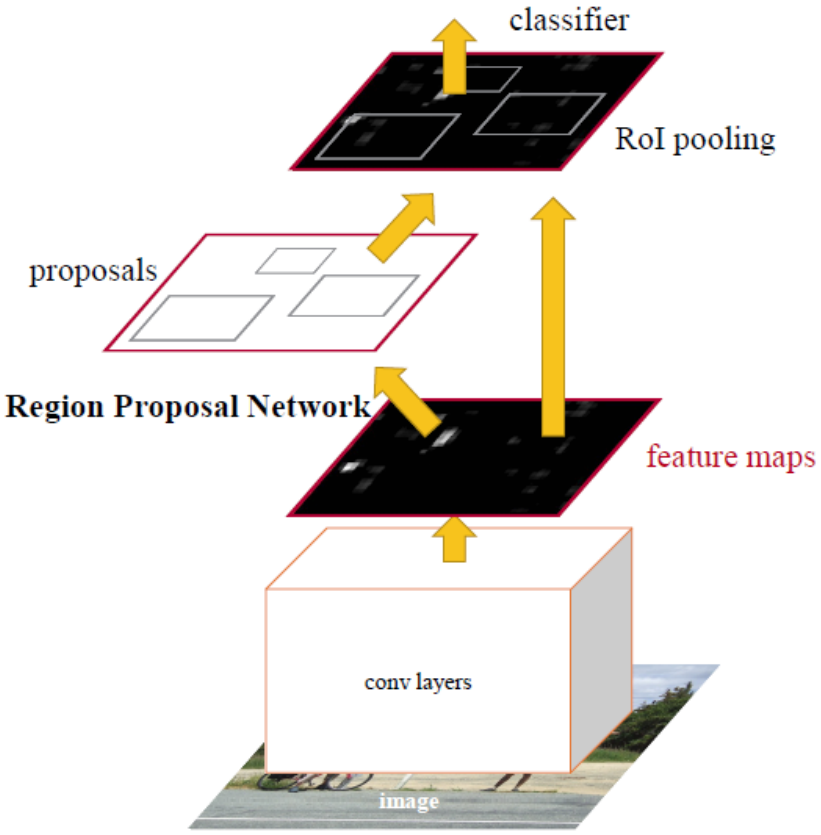

Figure 5.3: Faster-RCNN has a regional proposal stage (source: [Ren et al., 2015])

Chapter 6

Implementation

6.1 Implementation Overview

DetectorShop is built with Electron.js, a framework for building cross-platfrom desktop applications. DetectorShop uses React.js to build the UI and Node.js to access user's desktop OS resource. DetectorShop contains several docker container images as binary files. Specifically, DetectorShop contains three container images, which are labeling image, training image and inference image. These three images correspond to three major modules: labeling, training and inference. Each module is responsible for a critical aspect of DetectorShop.

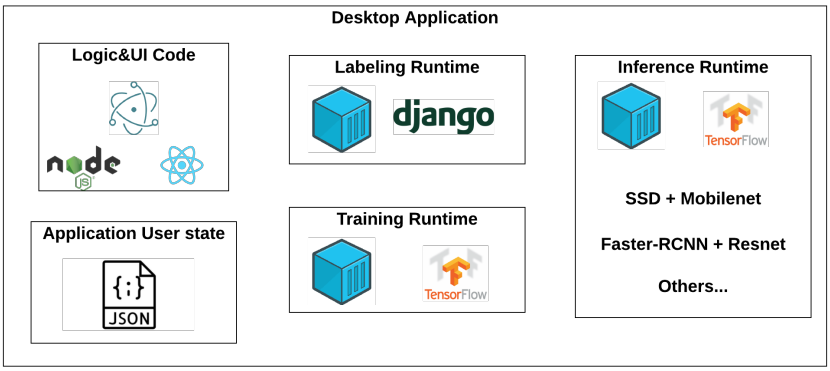

Figure 6.1: DetectorShop contains multiple containers as feature runtimes

If users need to access resources on remote machines, e.g., remote datasets, remote

hardware accelerators, DetectorShop will use SSH to transmit the docker container images to the remote machine and expand into a distributed system.

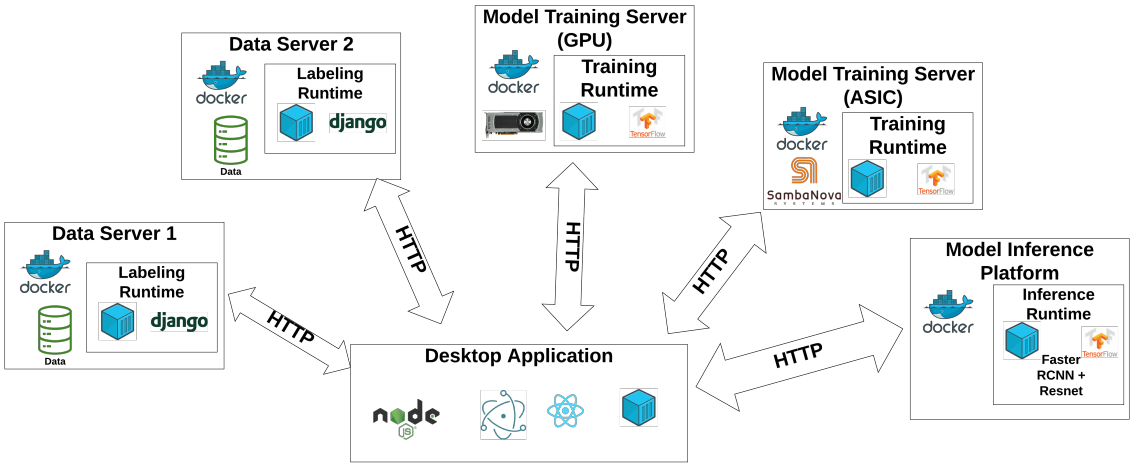

Figure 6.2: Transform from a desktop application into a distributed system

Within each module, there's a web server responsible for users' interaction. Once the system expands into a distributed system, all the interactions between the desktop client and remote module runtimes are through HTTP.

6.2 System Pipeline

The different modules are connected as follows:

1. Users use the labeling module to label the data, and get back a zip file containing the necessary files for training. We call this zip file a data record. In the current implementation, these files are the TFRecord files which contain actual bits of images and the labeling. Because TFRecord is a binary file format designed for compression, the file generated by the labeling module is way smaller than the original data.

2. DetectorShop automatically ships the data record generated from the labeling module along with the training container image to the training machine, a GPU cluster for example, and start the training. During the training, users are able to monitor the progress through a Tensorboard dashboard.

3. Once the training is ended by the user, DetectorShop will clean up the training runtime and get back a zip file containing the trained model (object detector). Users could either use the model file with Tensorflow in the final applications, or could ask DetectorShop to ship the zip file along with the inference image to an inference machine for testing the performance of the object detector.

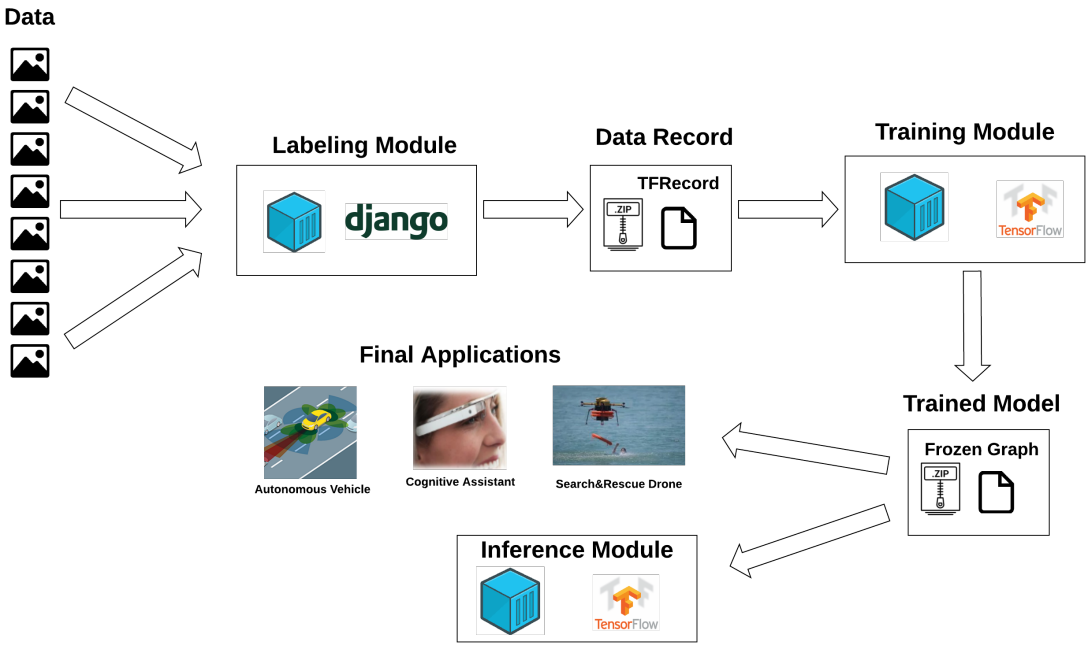

Figure 6.3: System Pipeline

6.3 Labeling Module

6.3.1 Computer Vision Annotation Tool (CVAT)

The labeling module is built on top of an open-sourced labeling tool called Computer Vision Annotation tool(CVAT). CVAT is a Django web application developed and open-sourced by Intel [cva].

6.3.2 File System Mapping Between Host and Container

The original CVAT requires users to upload the data. In our system, we eliminate the step of uploading data by following these steps:

1. Create a empty folder in the host file system
2. Use Docker's volume mechanism to map the folder into the labeling runtime
3. For each source image, create a corresponding dummy file in the newly created folder.
4. Create symbolic links between all the source images and their corresponding dummy files
5. Upload the dummy files into the labeling tool.

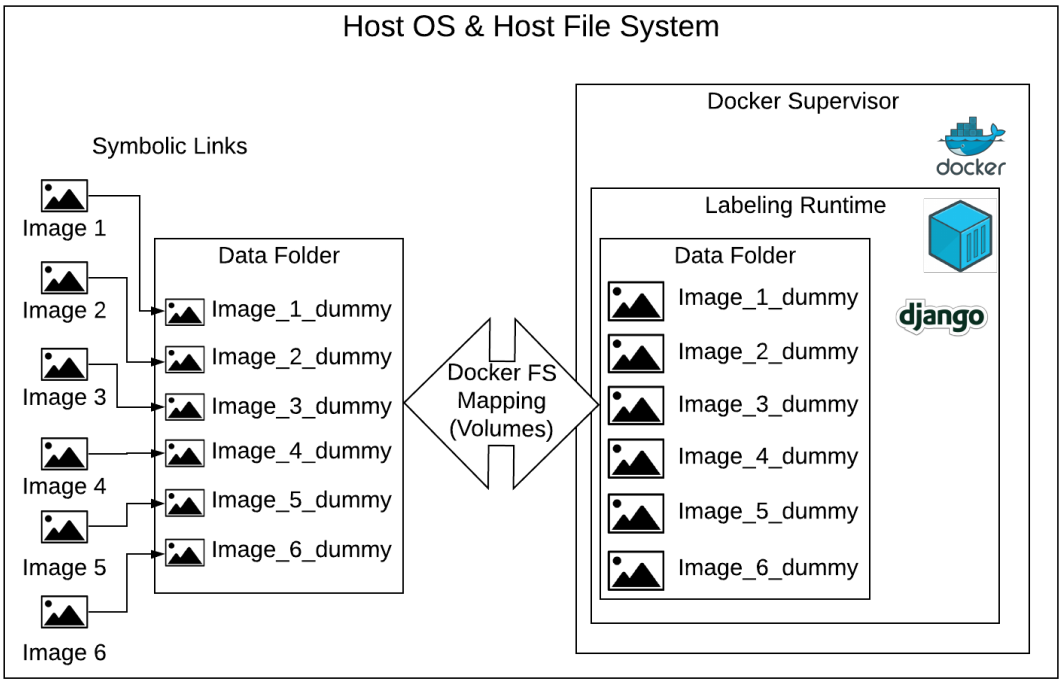

Figure 6.4: Push computation using SSH

After the aforementioned steps, users can use the CVAT just like uploading the original images.

6.4 Training Module

6.4.1 Tensorflow Object Detection API

The training module is built on top of the Tensorflow Object Detection API, an open-sourced tool box for object detection by Google. Specifically, we adopt the implemeation of Faster-RCNN, SSD, Resnet, Mobilenet, Inception Net.

6.4.2 Transfer Learning

We build the object detector with the pre-trained model by the Tensorflow Object Detection API. The pre-trained models are trained on Coco and Kitti datasets.

6.4.3 Adjustable Grid Size

To make the grid size adjustable, we modified the source code of the Tensorflow Object Detection API. We prepared three copies of the implementation of SSD. We ship the corresponding implementation to the training platform based on user's choice. More details are provided in the code base.

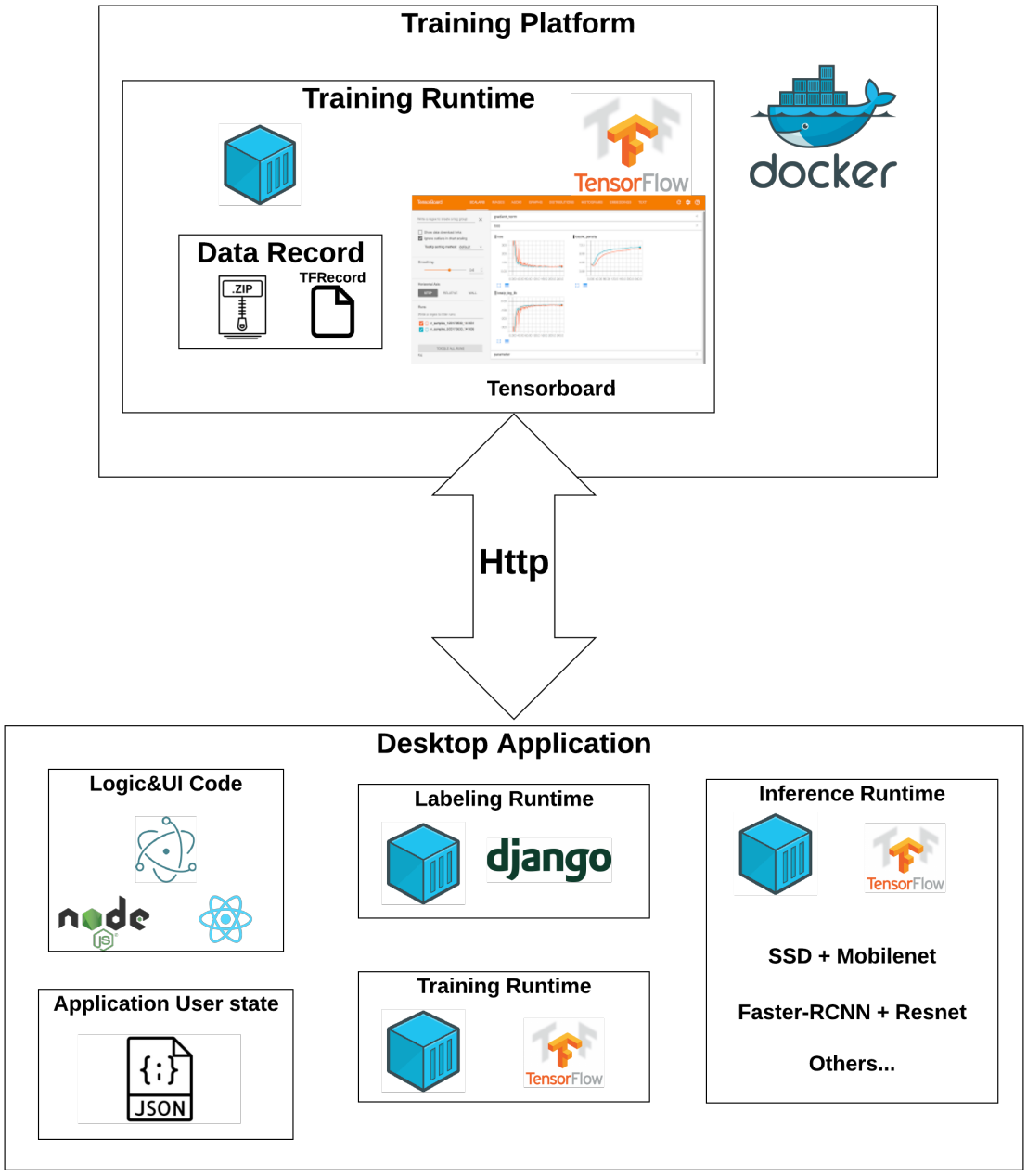

Figure 6.5: Training Module Implementation

6.5 Inference Module

6.5.1 High Frame Rate Video Streaming

One challenge in building an inference module is the video streaming frame rate. The system is able to produce object detector with realtime speed. For example the SSD+Mobilenet model's process latency is 30ms. In other words, to fully test the performance of the SSD+Mobilenet model, the inference module should support at least 30 frames per second. DetectorShop adopts webRTC as the streaming pipeline. The webRTC pipeline can easily stream video over 120fps with 1080P resolution.

6.6 Miscellaneous

6.6.1 State Management

In DetectorShop, users' states are stored as a JSON file in the desktop application (figure: 6.7). Each feature module has its own state and is stored within the feature container image. States across feature modules are synchronized by an async framework.

6.6.2 Async Framework

One challenge in the implementation is coordinating different remote module runtimes. For example, after DetectorShop pushes the labeling module image to the remote machine, it usually takes a while for the remote docker supervisor to load and run the docker image. It is critical for the desktop application to know when to open a new window for users to label the data. Network error would occur if open a window before the labeling module runtime is ready, and users will get confused. To solve this problem, we built a naive async framework. This async framework use a while loop to check the status of remote module runtimes.

6.6.3 Pushing Computation

DetectorShop first sets up a SSH connection with the remote machine. Once the SSH connection is established, DetectorShop will transmit the binary docker image to the remote

machine. Once the transmission is done, DetectorShop will invoke the remote docker supervisor to start and run the container image through SSH. All the runtime images contain a web server for handling user interactions. Once the docker container is ready and up, the users could interact with the remote runtime through the web server.

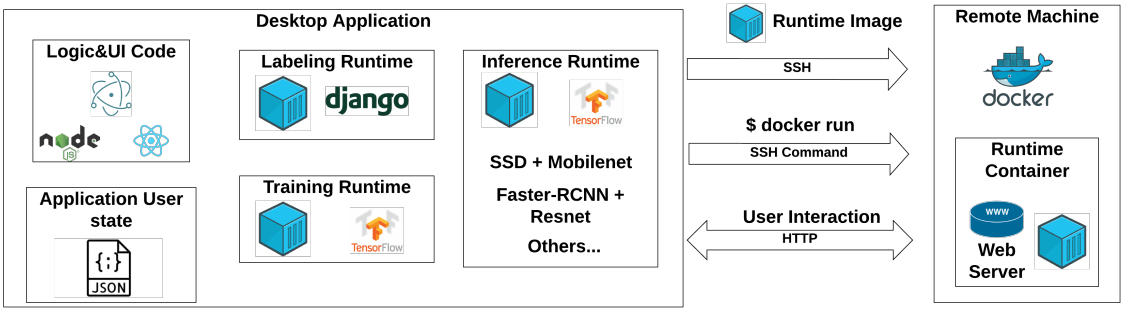

Figure 6.8: Push computation using SSH

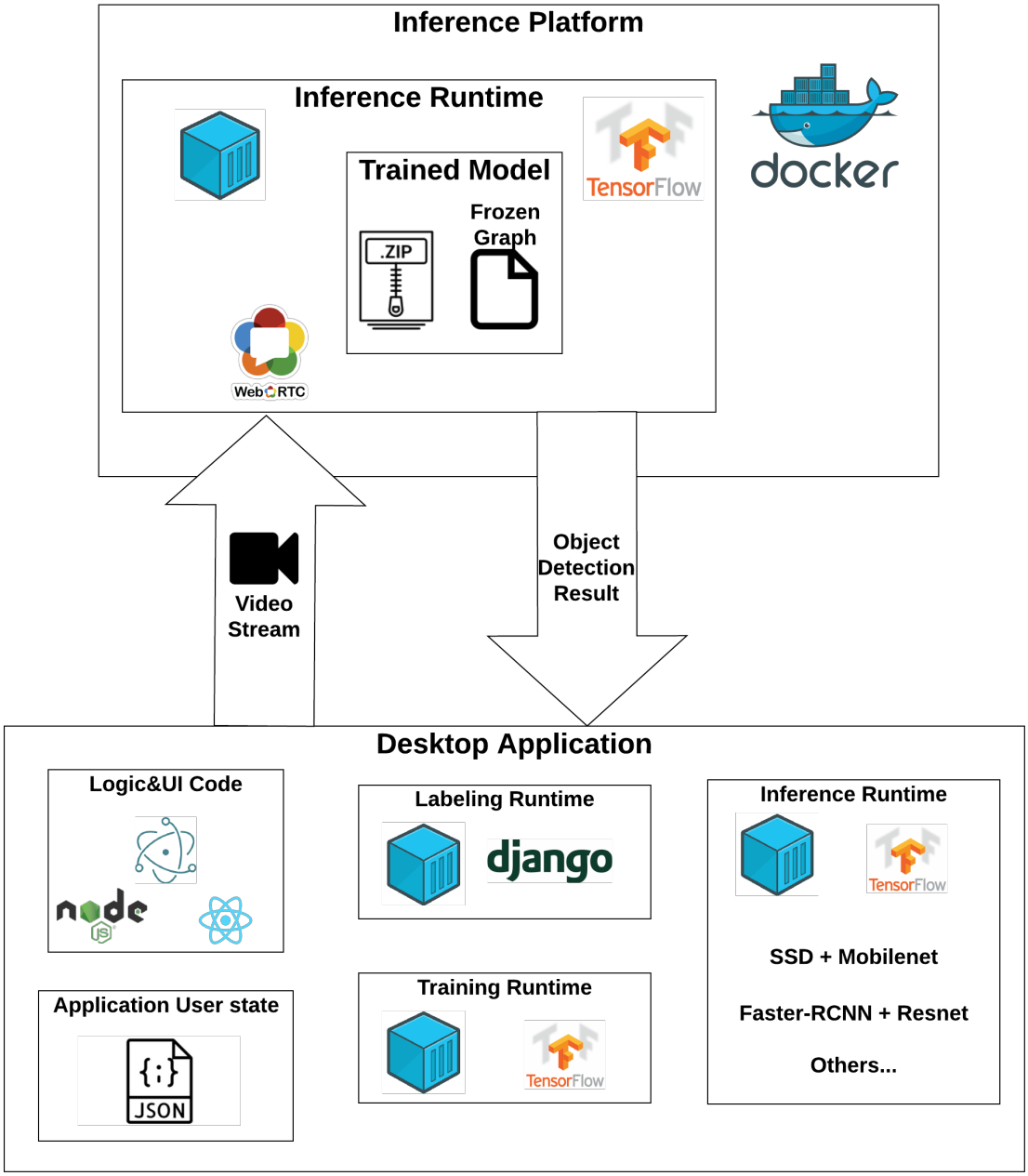

Figure 6.6: Inference Module Implementation

Figure 6.7: A Json file recording the user state

Chapter 7

Conclusion and Future Work

This thesis aims at lowering the barrier of building object detectors. It proposes DetectorShop, an end-to-end system that works as a PhotoShop for building object detectors. Through a careful system architectural design, DetectorShop efficiently manages datasets storage and is able to flexibly take advantage of specialized hardware. With a newly proposed abstraction layer, users are able to easily make tradeoffs between speed and accuracy in object detection models. Finally, DetectorShop provides a user workflow that enables users to build object detectors through a user-friendly GUI without any coding

7.1 Contributions

This thesis claims that

> Lowering the barrier of building deep learning object detectors can be achieved by factoring out low-level implementations and common operations to an end-to-end specialized tool. This tool enables users to easily make tradeoffs between speed and accuracy in object detection models and flexibly adopt specialized hardware to accelerate deep learning computation. Additionally, this tool efficiently manages datasets storage.

In this thesis, I have validated this statement from various aspects. Specifically this thesis has the following contributions:

- Design and implementation of a user workflow, which enables users to build object

detectors through a user-friendly GUI, entirely without coding.

- Design and implementation of a system architecture, which efficiently manages storage for datasets and is able to flexibly take advantage of specialized hardware.
- Design and implementation of an abstraction layer, which enables users to easily make tradeoffs between speed and accuracy in object detection models.

7.2 Roadmap for Evaluations

In this section, we discuss the future evaluation of DetectorShop that can be done. Specifically, we discuss plans with different time budgets.

7.2.1 One-Week Evaluation Plan

With one week, the goal of the evaluation will be evaluating the usability of DetectorShop, i.e, whether DetectorShop can actually help a user without much background knowledge to build an object detector. The specific steps are as follows:

- Recruit two computer science major students without much background in building deep learning object detectors.
- Provide both students a dataset of 10K images, and ask them to individually build an object detector based on this dataset.
- Provide one of the students a script of steps in building deep learning objectors, but ask the student to find corresponding tools by himself or herself.
- For the second student, provide him or her the DetectorShop.
- After one week, compare two students whether he or she is able to successfully create an object detector.

The reason we design the dataset to be 10K images large is that labeling and training 10K images using DetectorShop with a GPU suppose to take roughly half a week. Therefore, the user will have about half a week to study how to use DetectorShop.

7.2.2 One-Month Evaluation Plan

With one month, the goal of the evaluation will be evaluating whether DetectorShop can help users to create detectors that meet different speed/accuracy constraints. Similar to the one-week evaluation plan, we plan to recruit two students with similar backgrounds. But this time, we require them to have the knowledge of building deep learning object detection models. For the purpose of the evaluation, we provide 3 datasets and each dataset contains 10K images for a specific use scenario. For each of the use scenarios, we ask them to individually build an object detector that meets the speed/accuracy constraint. The 3 use scenarios are as follows:

- Ping-Pong ball detector used in cognitive assistants
  - accuracy requirement: low
  - speed requirement: around 100ms
- Person detector used in auto following drone
  - accuracy requirement: medium
  - speed requirement: around 500ms
- Defect detector used in factory quality control inspector
  - accuracy requirement: high
  - speed requirement: around 1000ms

For each use scenario, we ask the first student to implement his own detection model and ask the second student to tweak the provided models in DetectorShop through the Speed/Accuracy tradeoffs abstraction layer. After one month, we compare the completion rate of whether students are able to create detectors that meet different speed/accuracy constraints. The reason we design each dataset containing 10K images is that 10K images are enough for training an accuracy-oriented model such as Faster-RCNN.

7.2.3 One-Year Evaluation Plan

With one year, the goal of the evaluation will be extensive evaluations on all the components and critical design decisions in DetectorShop, i.e., labeling module, training module, inference module, data-centric design, and hardware-centric design. For each module

and critical design decision, we can conduct rigorous user studies for users with different backgrounds, instead of recruiting two computer science major students. The potential evaluation goals of each module and design decisions are as follows:

- Labeling Module & Data-Centric Design
  - How much user time can the labeling module save comparing to using PhotoShop for data labeling?
  - How much bandwith and disk storage can data-centric design save comparing to the traditional data-uploading approach?
- Training Module & Hardware-Centric Design
  - How much training time can the training module save by adopting specialized hardware comparing to running the deep learning training on normal hardware, such as a multi-core CPU?
  - How many new hardware that the training module can take advantage of by adopting hardware-centric design in the coming year?
- Inference Module & Hardware-Centric Design
  - How much user time can the inference module save comparing to inference models manually on images and video stream?
  - How many new hardware that the inference module can explore by adopting hardware-centric design in the coming year?

7.3 Future Work

Object detection is one of the most rapidly evolving research areas today. To match the advancement in computer vision research, many potential areas need to be explored for an end-to-end system. This section discusses future improvements toward fast iteration and testing in building object detectors. For each improvement, this thesis will discuss benefits it brings and provide a possible approach for implementations.

7.3.1 Configurable Training Strategy

Training heuristics greatly improve the accuracy of deep learning models. For object detection specifically, a good training strategy can improve the absolution precision up to

5% compared to the state of the art baselines [Zhang et al., 2019]. Enabling users to configure the object detection model training is desirable, for instance through dynamic learning rate. This functionality could be achieved by utilizing the checkpoint mechanism in Tensorflow.

7.3.2 Automatic Data Augmentation

A recent work by Google demonstrates that simply by data augmentation, the current state of the art models could improve their accuracy by more than +2.3 mAP [Zoph et al., 2019]. This functionality could be added in the labeling module by implementing the data augmentation in Tensorflow.

7.3.3 More Advanced Labeling Functionalities

The current labeling functionality in DetectorShop only supports simple data labeling, i.e. drawing bounding boxes. DetectorShop could integrate other complicated labeling tools, such as Eureka at CMU [Feng et al., 2019]. This integration could be achieved by creating a web-service between DetectorShop and Eureka.

7.3.4 More Object Detection Models & Implementations

DetectorShop's object detection models are built on top of Tensorflow Object Detection API which was developed in 2016 and only implements meta-architectures such as SSD, Faster-RCNN, FCN. Recently, there are emerging new toolboxes, e.g., MMDetection [Chen et al., 2019] . These new toolboxes support more models and claim to have better implementations. To take advantage of these new toolboxes, we can replace the Tensorflow Object Detection API with these new toolboxes in the model training module.

7.3.5 More Comprehensive Testing

DetectorShop only supports object detector testing in term of human inspection. To provide users a more complete developing experience, DetectorShop needs to provide a more comprehensive testing mechanism. One feasible approach is coming up a set of benchmarks for different applications, e.g., [Chen, 2018] proposes a set of benchmarks for cognitive assistant applications. For each of those benchmarks, DetectorShop could automate the testing process to enable users to test the object detector in the final application context.

7.3.6 Integrate Deep Learning Expert Heuristic Knowledge

As a high-level tool, DetectorShop can also integrate some deep learning expert heuristic knowledge to prevent users from making common mistakes, such as training overfitting, insufficient dataset for specific models, and datasets with too similar data. For handling training overfitting, DetectorShop can partition the dataset into the training dataset and the testing dataset. During the training, DetectorShop can create snapshots of the detection model. Once DetectorShop detects overfitting, such as the training loss keeps reducing but the testing loss starts increasing, DetectorShop can stop the training and export the latest previous snapshot as the detection model. To handle insufficient datasets, if the dataset is too small for some specific models, e.g., only 500 images for Faster-RCNN + Resnet, DetectorShop will notify the user and prevent users from start training the model. To prevent users from working on datasets that contain too similar data, DetectorShop can run perceptual hashing on the datasets. If the data are too similar to each, DetectorShop will warn the user that detectors built on such dataset is not likely to have a good performance.

Appendix A

Screenshots of the Current Implementation

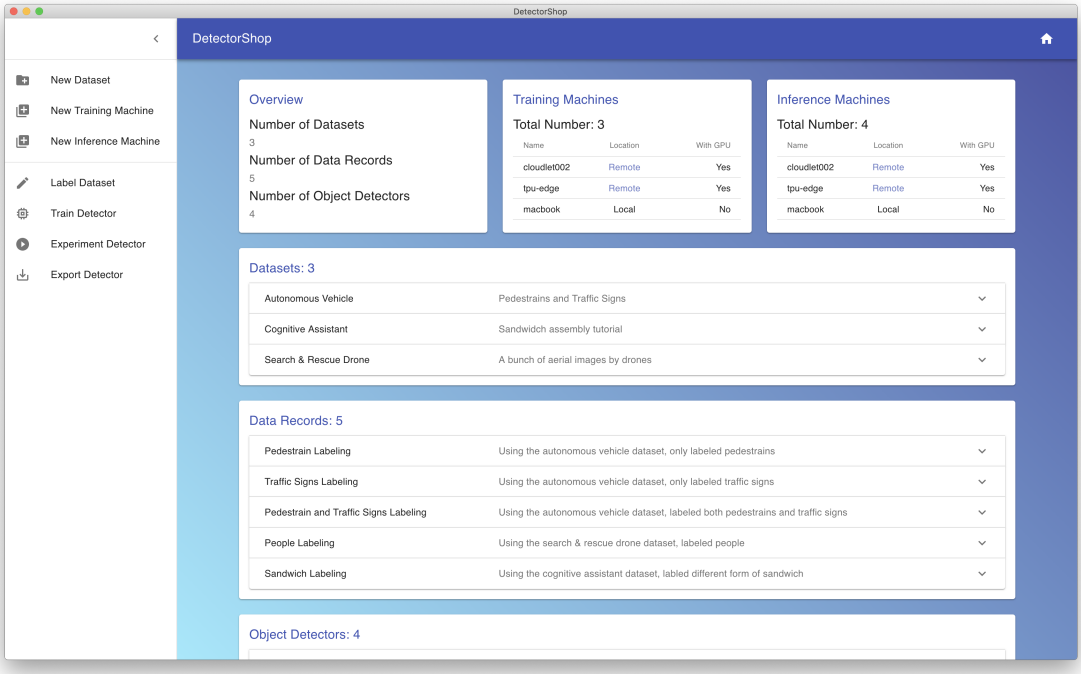

Figure A.1: A dashboard showing the overview information about datasets, data records, and object detectors

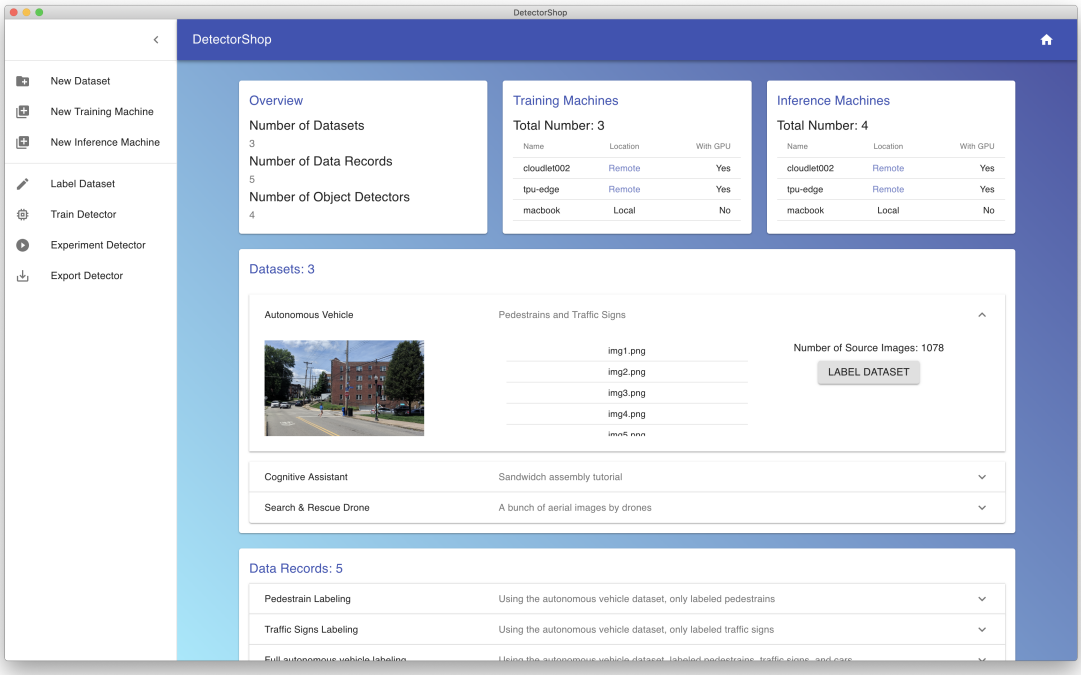

Figure A.2: Basic information about a dataset, autonomous vehicle dataset for example

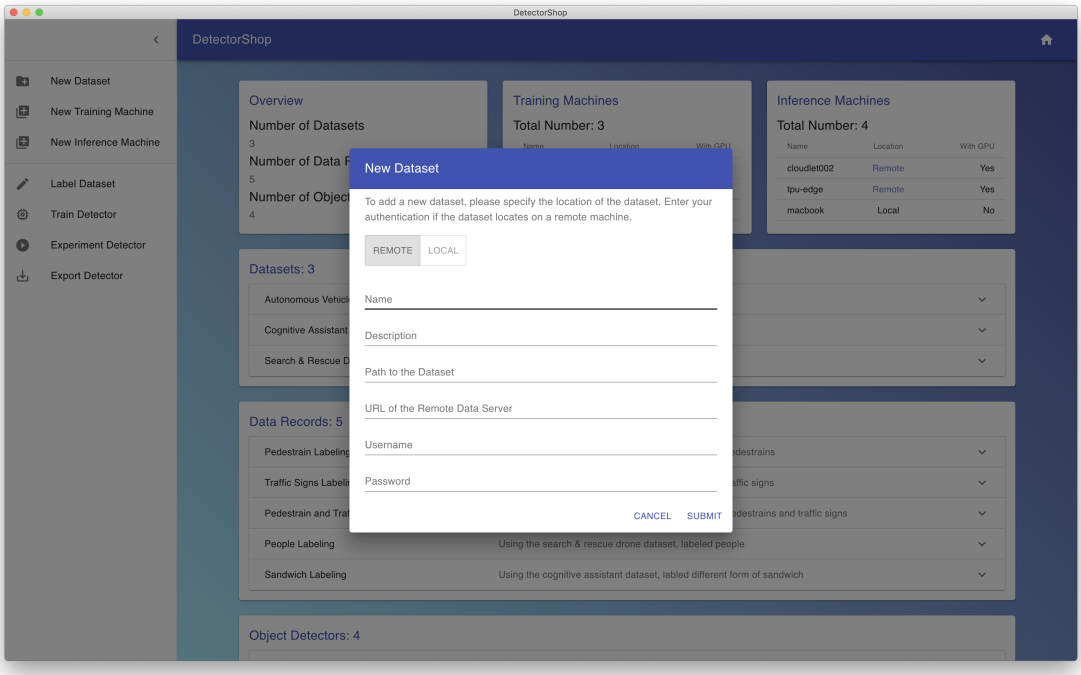

Figure A.3: Specify a new dataset

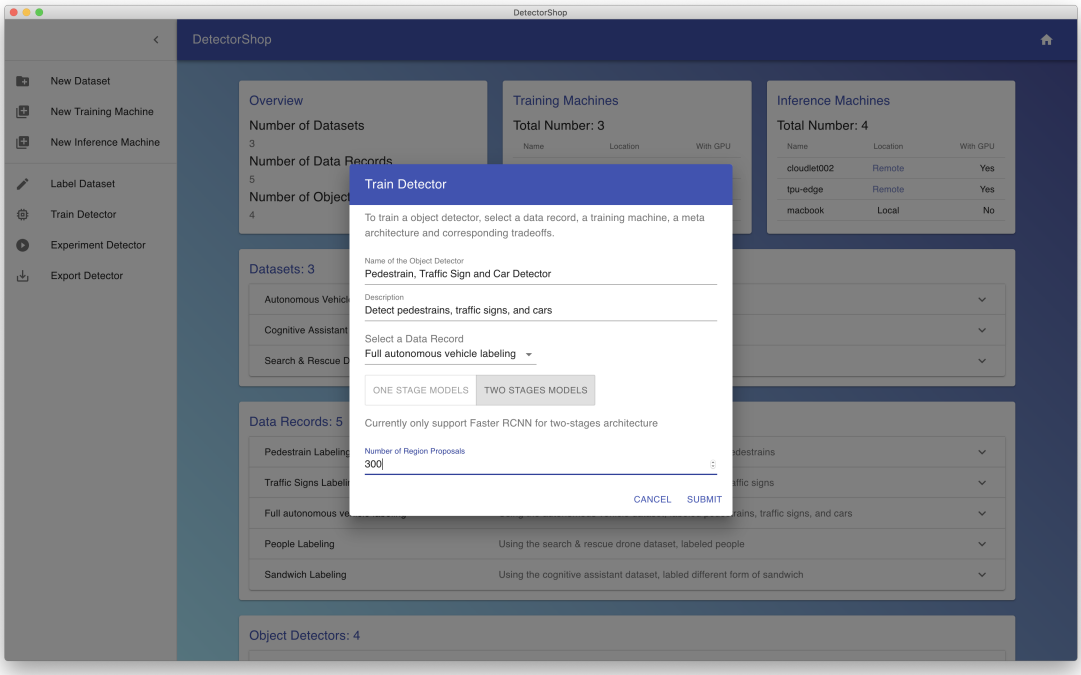

Figure A.4: Train an object detector using Faster-RCNN and autonomous vehicle dataset

Bibliography

Powerful and efficient Computer Vision Annotation Tool (CVAT): opencv/cvat. URL https://github.com/opencv/cvat. original-date: 2018-06-29T14:02:45Z. 6.3.1

Overarching trends and applications at nips 2017. URL http://copypasteprogrammers.com/overarching-trends-and-applications-at-nips-2017-with-links-fc523d3354a7/. (document), 2.6

Visual Studio Code Remote Development. URL https://code.visualstudio.com/docs/remote/remote-overview. (document), 4.4, 4.7

Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, Jifeng Dai, Jingdong Wang, Jianping Shi, Wanli Ouyang, Chen Change Loy, and Dahua Lin. MMDetection: Open MMLab Detection Toolbox and Benchmark. *arXiv:1906.07155 [cs, eess]*, June 2019. URL http://arxiv.org/abs/1906.07155. arXiv: 1906.07155. 5.1, 7.3.4

Zhuo Chen. *An Application Framework for Wearable Cognitive Assistance*. PhD thesis, Carnegie Mellon University, April 2018. 2.1, 2.1, 7.3.5

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20 (3):273–297, September 1995. ISSN 1573-0565. doi: 10.1007/BF00994018. URL https://doi.org/10.1007/BF00994018. 2.2.1

N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, June 2005. doi: 10.1109/CVPR.2005.177. 2.2.1

Ziqiang Feng, Shilpa George, Jan Harkes, Padmanabhan Pillai, Roberta Klatzky, and Mahadev Satyanarayanan. Eureka: Edge-based discovery of training data for machine learning. *IEEE Internet Computing*, 2019. 7.3.3

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv:1311.2524 [cs]*, November 2013. URL http://arxiv.org/abs/1311.2524. arXiv: 1311.2524. 2.2.2

Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *arXiv:1611.10012 [cs]*, November 2016. URL http://arxiv.org/abs/1611.10012. arXiv: 1611.10012. 5.1, 5.2, 5.3, 5.4

Hei Law and Jia Deng. CornerNet: Detecting Objects as Paired Keypoints. *arXiv:1808.01244 [cs]*, August 2018. URL http://arxiv.org/abs/1808.01244. arXiv: 1808.01244. 2.2.2

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. *arXiv:1512.02325 [cs]*, 9905:21–37, 2016. doi: 10.1007/978-3-319-46448-0_2. URL http://arxiv.org/abs/1512.02325. arXiv: 1512.02325. (document), 2.2.2, 5.2

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *arXiv:1411.4038 [cs]*, November 2014. URL http://arxiv.org/abs/1411.4038. arXiv: 1411.4038. 2.2.2

David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004. ISSN 1573-1405. doi: 10.1023/B:VISI.0000029664.99615.94. URL https://doi.org/10.1023/B:VISI.0000029664.99615.94. 2.2.1

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv:1506.02640 [cs]*, June 2015. URL http://arxiv.org/abs/1506.02640. arXiv: 1506.02640. (document), 2.2.2, 5.1

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv:1506.01497 [cs]*, June 2015. URL http://arxiv.org/abs/1506.01497. arXiv: 1506.01497. (document), 2.2.2, 5.4, 5.3

Robert E. Schapire. A Brief Introduction to Boosting. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 1401–1406, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. URL

http://dl.acm.org/citation.cfm?id=1624312.1624417. event-place: Stockholm, Sweden. 2.2.1

Zhi Zhang, Tong He, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of Freebies for Training Object Detection Neural Networks. *arXiv:1902.04103 [cs]*, February 2019. URL http://arxiv.org/abs/1902.04103. arXiv: 1902.04103. 7.3.1

Barret Zoph, Ekin D. Cubuk, Golnaz Ghiasi, Tsung-Yi Lin, Jonathon Shlens, and Quoc V. Le. Learning data augmentation strategies for object detection, 2019. 7.3.2