UNIVERSITY COLLEGE LONDON

MASTERS THESIS

Impact of Resolution on Car Detection Accuracy in Satellite Imagery Using Convolutional Neural Networks

Author:
Heidi M HURST

Supervisor:
Dr. Santosh BHATTARAI

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in

Geographic Information Science

in the

Department of Civil, Environmental and Geomatic Engineering

September 11, 2018

UNIVERSITY COLLEGE LONDON

Declaration of Ownership

Student Name:	Heidi M Hurst
Programme:	MSc Geographic Information Science
Supervisor Name:	*Dr. Santosh Bhattarai*
Dissertation Title:	Impact of Resolution on Car Detection Accuracy in Satellite Imagery Using Convolutional Neural Networks

- I confirm that I have read and understood the guidelines on plagiarism, that I understand the meaning of plagiarism and that I may be penalised for submitting work that has been plagiarised.
- I declare that all material presented in the accompanying work is entirely my own work except where explicitly and individually indicated and that all sources used in its preparation and all quotations are clearly cited.
- I have submitted an electronic copy of the project report through Moodle/Turnitin.

Should this statement prove to be untrue, I recognise the right of the Board of Examiners to recommend what action should be taken in line with UCL regulations.

Signed: _____

Date: _____

UNIVERSITY COLLEGE LONDON

Abstract

Department of Civil, Environmental and Geomatic Engineering

Master of Science

Impact of Resolution on Car Detection Accuracy in Satellite Imagery Using Convolutional Neural Networks

by Heidi M HURST

Identification of small cars in satellite imagery has a wide variety of applications. However, it is unclear what role resolution plays in small object detection accuracy, measured as average precision (AP). This study investigates the impact of spatial and object resolution (pixels per car) on car detection accuracy in satellite imagery using convolutional neural networks (CNNs). Two trials were conducted using the publicly available xView database. In the first trial, Inception-Single Shot Multibox Detector (I-SSD) CNN models were trained on images that had been downsampled, simulating low resolution images. In the second, I-SSD models were trained on images that had been upsampled (simulating low resolution images that had been enlarged). Subsequently, AP was modelled as a linear function of object size and downsample factor. Experimental results suggest that object resolution is a greater determinant of model accuracy than spatial resolution. Additionally, model accuracy may be increased by increasing overall image size.

Acknowledgements

This body of work would not have been possible without the support, generosity, and humour of a great many individuals to whom I owe my gratitude:

The scientists and dreamers that came before me, including all the women without whom these technologies and my place in this discipline would not have been possible.

My advisor Dr. Santosh Bhattarai, who humoured my tangents, encouraged my curiosity far beyond the scope of my initial research question, and was willing to join my adventure into a new field with no guarantee of meaningful results.

Dave Griffiths, who dissuaded me from creating my own model from scratch, gave me a tutorial on TensorFlow basics, and helped me de-stress with laughter and nerdy banter. This project would not have been possible without his generosity in granting me access to the DeepLearning computing cluster (or without Dr. Jan Boem's permission, for which I am also deeply grateful).

My undergraduate advisers and mentors, including Dr. Huth, Dr. Brenner, Dr. Levine, Professor Stilgoe, and Dr. Srinivasan who always believed me that there was something beautiful to be gained from the intersection of geography, mathematics, and design.

My kind (and sometimes clueless and definitely obligated) readers Paul Hurst, Patty Mayer, Hilary Hurst, Duncan McElfresh, and Achim Harzheim who gave thoughtful feedback on my research, final report, and poster, who gave me pep talks, and who reminded me that we must continue to pursue science and truth, as JFK said, "not because it is easy, but because it is hard."

All those whose kindness helped me feel at home here in London though baked goods, climbing, a couch constantly occupied with loving visitors, Sundays at Dishoom and the flower market ...

...and Rob and Richard, who taught me British spelling and introduced me to the best of British banter.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Definitions

So, imagine if you can say, this is a ship, this is a tree, this is a car, this is a road, this is a building, this is a truck. And if you could do that for all of the millions of images coming down per day, then you basically create a database of all the sizeable objects on the planet, every day.

Will Marshall, Planet CEO (2018)

1 Introduction

1.1 Motivation

In November 2017, private space technology company Planet announced that they had achieved a remote sensing milestone: the ability to "image the entire Earth's landmass every day" (Marshall, 2018). Today, with over 200 satellites in operation, Planet processes more than six terabytes of data every day.

Extracting meaningful information from such a large volume of data previously required tremendous human resources. Recent advances in machine learning, the science of creating systems that can "learn" to accomplish tasks for which they were not explicitly programmed through ingesting large numbers of examples, as well as the increased computing power available in graphics processing units (GPUs) and cloud resources have made it possible to automate and accelerate this information extraction. The availability of high resolution (~80cm) data coupled with these technological advances has made it possible to detect smaller objects than ever before, such as small cars, across large spatial and temporal ranges.

Automated vehicle detection has a wide variety of civilian and military applications, including monitoring the impact of traffic restrictions, identifying build up of forces along a border, or estimating the population in an area recovering from a natural disaster (Larsen, Koren, and Solberg, 2009; Wolfinbarger, Drake, and Ashcroft, 2014).

Despite the clear utility of automated vehicle detection in satellite imagery, the impact of resolution on performance is not well quantified. This study addresses this research gap by investigating the impact of (a) spatial resolution and (b) object resolution (pixels per car) on small car detection accuracy.

1.2 Aims and Objectives

This research aims to answer the following question:

> How do (a) spatial resolution and (b) object resolution (pixels per car) impact small car detection accuracy in satellite imagery using convolutional neural networks (CNNs)?

This central aim is supported by the following objectives:

1. Develop a broad understanding of object detection techniques.
2. Prototype a basic neural network in a remote computing environment.
3. Train a number of models to identify small cars using the xView dataset, each at a different spatial resolution.
4. Evaluate and compare model quality.

1.3 Overview

This inquiry begins with an overview of the recent and seminal literature in CNNs for small object and vehicle detection in Chapter 2. The data, software, and hardware resources utilised are described in Chapter 3. Next, Chapter 4 describes the methodology for each trial, including specific source code modifications. The results of both trials are described and analysed in Chapter 5. Chapter 6 discusses these results and suggests avenues for future research. Finally, Chapter 7 provides a conclusion of the primary findings.

How is it possible for a slow, tiny brain, whether biological or electronic, to perceive, understand, predict, and manipulate a world far larger and more complicated than itself?

Stuart Russell and Peter Norvig (2016)

2 Literature Review

This chapter explores relevant literature, including small object detection and car detection in aerial imagery. First, traditional (non neural network) methods for car identification are discussed. Next, an overview of convolutional neural networks (CNNs) and their applications to general object detection are explored. Finally, existing literature regarding CNNs for small object and small car detection is reviewed. This chapter concludes with a discussion about the gap present in this literature that this study aims to address.

2.1 Traditional Car Detection Methods

Traditional car detection methods relied on the identification of features meaningful to a human observer's understanding of a car, such as shape, size, windscreen, and profile from shadow (if present). Many of these methods tried to replicate the human process for car identification, for example by selecting features identified through psychological tests to be important for humans recognition of cars (Zhao and Nevatia, 2003) or restricting the search area to road networks (Shi et al., 2012).

Despite significant research, these methods offered low accuracy and occasionally failed in the presence of strong shadows or trees (Zhao and Nevatia, 2003). In addition they primarily applied models to the highest resolution imagery, with the assumption that this would yield the best results.

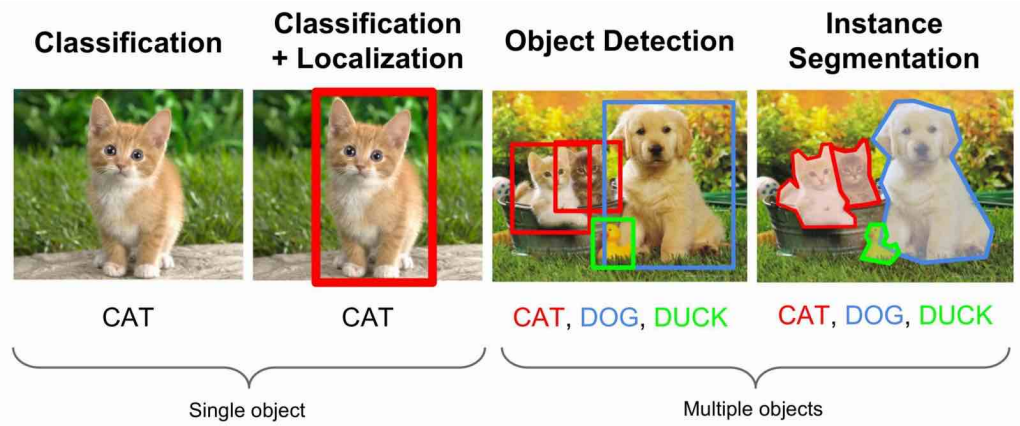


FIGURE 2.1: Examples of classification, classification and localisation, object detection, and image segmentation (Ouaknine, 2018).

2.2 Convolutional Neural Networks (CNNs) for Small Car Detection

Over the last 20 years, many algorithms such as the traditional methods for car detection discussed in Section 2.1 have been rapidly outperformed by machine learning (LeCun, Bengio, and Hinton, 2015). This section provides an introduction to neural networks (NNs)[1], a review of relevant literature about CNNs, and an overview of research into small object and small car detection.

2.2.1 Introduction to Neural Networks (NNs)

In contrast to traditional algorithms, machine learning systems "learn" to accomplish tasks for which they were not explicitly programmed. Mitchell (1997) offers the following definition of learning:

Definition 2.1 (Machine Learning). *"A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

For supervised machine learning techniques, the experience E typically consists of a labelled dataset such as satellite images with annotated bounding boxes for cars. Applying the above definition to machine learning for small car detection, accuracy (P) at detecting all small cars in an image (T) improves as it is exposed to more and more annotated example images (E).

While many machine learning algorithms exist, such as support vector machines, clustering methods, etc, CNNs have shown the best performance on image related tasks (LeCun, Bengio, and Hinton, 2015). They are widely used in both research and production for image classification, classification and localization, object detection, and image segmentation (Karpathy, 2018). Examples of these various tasks are

[1]Neural networks are often referred to as deep learning, deep networks, or artificial neural networks (ANNs). They are a subset of machine learning algorithms.

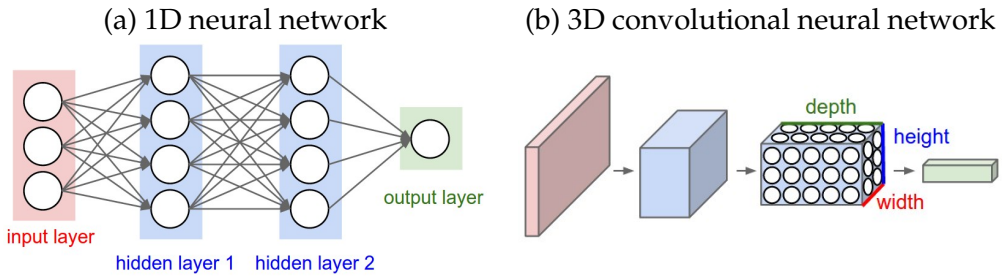


FIGURE 2.2: Simplified depiction of (a) a 1D neural network and (b) a 3D convolutional neural network (Karpathy, 2018).

shown in Figure 2.1. This study focuses exclusively on object detection, generating bounding boxes and labels for many objects in an image.

Structure

NNs, modelled after the human brain, consist of *nodes* that perform simple processing. One node may be connected to many other nodes via unidirectional pathways that receive or send data. These nodes are typically arranged in layers, as in Figure 2.2 (a), with layers in the middle referred to as *hidden layers* (LeCun, Bengio, and Hinton, 2015). In a *feed forward* network, all connections between nodes move in the same direction (that is, information never goes backward to a previous node or layer in the model).

Each node assigns a *weight* to each of its incoming connections. When in use, the node receives a value from each incoming connection and multiplies this by the respective weight, then sums over all connections. The resulting number is compared to a *threshold* value. If this number is larger than the threshold value, the node *activates* and passes this value along to other nodes in the network (Goodfellow, Bengio, and Courville, 2016).

This process of activation can be mathematically modelled as a step function whereby all inputs are mapped either to 1 or 0 (telling the node to activate or not activate). In practice this means that small perturbations in network inputs can lead to large perturbations in network outputs. To mitigate this, other activation functions such as the sigmoid function (Barron, 1993) or rectified linear unit (Nair and Hinton, 2010) are more commonly used.

Optimization (Training)

When a NN is first created, all weights are initialized to random numbers. The network then uses these weights to compute an output for a given piece of training data, for example generating a number of small car bounding boxes for a given input image. To determine the quality of this output, the generated bounding boxes are compared against the correct output (provided as part of the training data) using

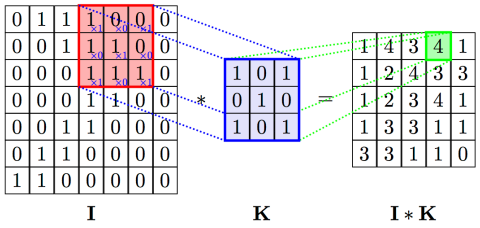


FIGURE 2.3: Example of a 2D convolutional filter K applied to an array or image I. The coloured boxes show how K (blue) is convolved with a neighbourhood in I (red) to yield a single pixel in the output image $I * K$ (green) (Velikovi, 2016).

an objective function. This is often called the *loss function* (Goodfellow, Bengio, and Courville, 2016).

The ultimate objective of training a NN is to minimize the value of this loss function without compromising model quality (LeCun et al., 1998). This can be visualized as finding the lowest spot on an n-dimensional surface riddled with hills and valleys, where n is the number of weights in the network.

To minimize this loss function, weights for each node of the NN are iteratively improved using a process called *gradient descent*. Much like rolling a ball down a hill, gradient descent aims to tweak the n-dimensional weight vector by moving it in the direction of steepest decrease. In practice, most NNs are optimized using stochastic gradient descent, a method that uses gradient descent over random subsets of examples to minimize training time and avoid local minima (LeCun et al., 1998).

In addition to the model parameters (weights), each model has *hyperparameters* that optimize how the model learns, such as how quickly it learns, how frequently to evaluate performance, when to stop training, etc. These are often adjusted by hand after evaluating model performance on a set of *evaluation* data (Goodfellow, Bengio, and Courville, 2016). This data is the same format as the training data, but has not been used in the training task.

2.2.2 Introduction to Convolutional Neural Networks (CNNs)

The general architecture described in Section 2.2.1 has been very successfully adapted to process image input using convolutional neural networks (CNNs). CNNs mimic the structure of images by organizing neurons into 3D volumes with magnitudes equal to the height, width, and channels (e.g. red, green, blue) of the input image as in Figure 2.2 (b) (Goodfellow, Bengio, and Courville, 2016).

CNNs contain convolutional layers which consist of several learnable filters. Each filter consists of a kernel (or mask) in the form of a $n \times n$ matrix that is convolved with each pixel of the input image, resulting in a localized weighted sum. This process is shown in Figure 2.3. In addition, pooling layers reduce the size along the spatial dimensions (height, width) with the purpose of combining similar features (LeCun, Bengio, and Hinton, 2015). Any number of convolutional and pooling layers may be connected. Like other NNs, CNNs end with a *fully connected* layer that

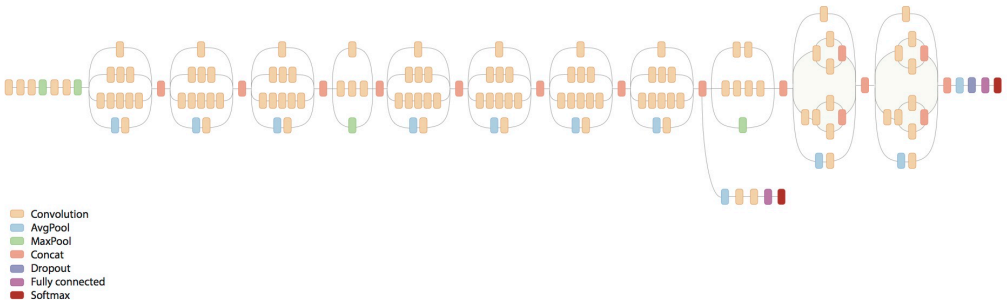


FIGURE 2.4: CNN architecture for Google's Inception v3 network showing convolution (tan), pooling (green and blue), and fully connected (purple) layers. Sections with parallel layers are called *inception modules* (*Inception Model* 2018).

maps the results to the desired output form, such as image labels or small car bounding boxes (Goodfellow, Bengio, and Courville, 2016). Figure 2.4 shows an example of the Inception v3 network (input at left, output at right), demonstrating that networks can consist of both "deep" and "wide" arrangements of modules. Segments with parallel grey lines in this model are called *inception modules* (Szegedy et al., 2015).

While CNNs have been used at least since the late 1990s (Lawrence et al., 1997), their use increased dramatically following the 2012 success of the "AlexNet" CNN at classifying images in the ImageNet 2010 contest (Krizhevsky, Sutskever, and Hinton, 2012). Since then, a tremendous amount of work has been done to increase CNN accuracy and versatility. However, most of this work assumes high image resolution and is not concerned about performance across a range of resolutions.

2.2.3 CNN Evaluation

New models are typically tested against existing benchmark datasets to demonstrate improvement, such as the Pascal Visual Objects Challenge (VOC) (Everingham et al., 2010) or Common Objects in Context (COCO) (Lin et al., 2014). Typically, the Intersection over Union (IoU) metric is computed to determine true detections. The following definitions explain IoU and its use in identifying positive and negative matches:

Definition 2.2 (IoU: Intersection over Union)**.** *The Intersection over Union, known also as IoU or Jaccard index, is equal to the area of overlap between a predicted bounding box and a ground truth bounding box divided by the union of those two bounding boxes (see Figure 2.5). For most applications, IOU* ≥ 0.5 *is considered a true positive (Lam et al., 2018; Henderson and Ferrari, 2016; Everingham et al., 2010).*

Definition 2.3 (True Positive)**.** *A predicted box is considered a true positive if the IoU between the predicted box and ground truth box exceeds 0.5 (Lam et al., 2018; Henderson and Ferrari, 2016; Everingham et al., 2010). A true positive indicates a successful detection.*

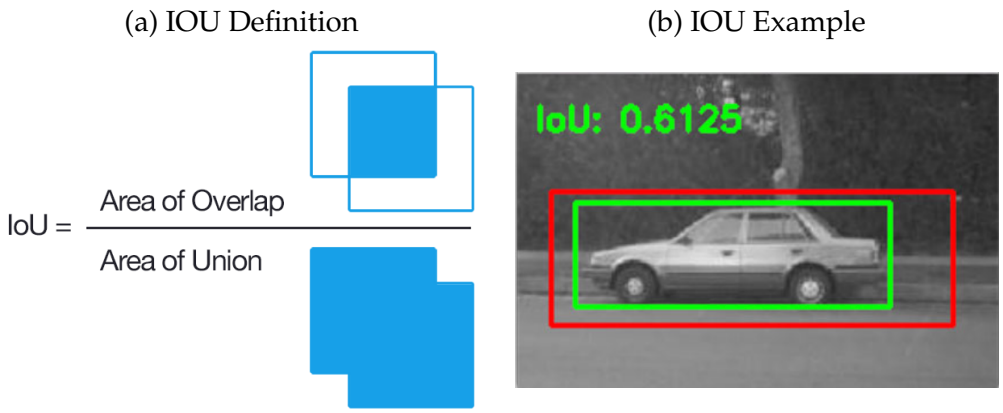


FIGURE 2.5: Visualization of (a) definition of IoU and (b) TP detection (red) for a car in profile with ground truth (green) where IoU=0.6125 (Rosebrock, 2016).

Definition 2.4 (False Positive). *A predicted box is considered a false positive if the IoU between the predicted box and ground truth box is below 0.5. Multiple detections of the same object are considered false positives; only the first detection counts as a true positive (Lam et al., 2018).*

Definition 2.5 (False Negative). *A ground truth box for which there is no corresponding true positive predicted bounding box is a false negative. A false negative indicates a missed detection.*

As calculating the number of true positives alone does not give a full picture of model quality, the number of true positive, false positive, and false negative detections are used to define the following evaluation metrics for CNN performance:

Definition 2.6 (Precision). *Precision is defined as*

$$\frac{TP}{TP + FP} \tag{2.1}$$

where TP is the number of true positives and FP is the number of false positives. Each false positive decreases precision. Generally, precision quantifies what proportion of detected objects were relevant (DIUx xView, 2018).

Definition 2.7 (Recall). *Recall is defined as*

$$\frac{TP}{TP + FN} \tag{2.2}$$

where TP is the number of true positives and FN is the number of false negatives. Each true positive increases recall. Generally, recall quantifies what proportion of the ground truth objects were detected (DIUx xView, 2018).

Definition 2.8 (Average Precision). *For each additional detection included in the model evaluation, both precision and recall can be calculated and plotted. This yields a stepwise plot*

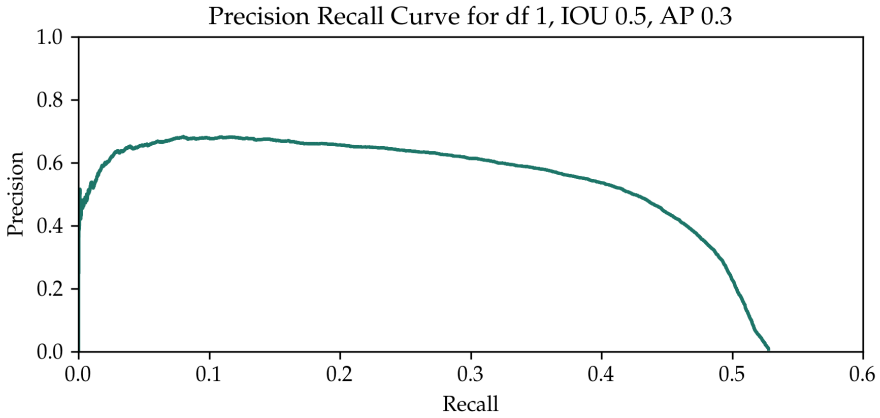


FIGURE 2.6: Precision as a function of recall for one trial in this study. Here AP (area under the curve) is 0.3.

of $P(r)$, precision as a function of recall as in Figure 2.6. Average precision is the integral of $P(r)$ with respect to r, or for simplicity the approximate area under this curve (Henderson and Ferrari, 2016). Average precision is calculated at a specific IoU value and often reported as AP@0.5IoU (Everingham et al., 2010). Values range from 0 to 1, where 1 is perfect. If more than one class is present, the mean of the AP (mAP) across classes quantifies model quality.

Average precision is the primary metric used to evaluate model quality as it balances the importance of detecting all objects in a scene with detecting objects well (Everingham et al., 2010; Lam et al., 2018; Lin et al., 2014; Henderson and Ferrari, 2016).

2.2.4 CNN Model Architectures

Several CNN architectures have been developed specifically for object detection. This section discusses two popular general architectures, RCNN and SSD, and their application to small object detection. (Here, *small objects* is generally defined as those occupying less than 5% of the pixels of an image regardless of their real world dimensions (Everingham and Winn, 2007). A single car of the type detected in this study occupies only ~0.17% of an image.)

The Regions with Convolutional Neural Network features (RCNN) architecture proposed by Girshick et al. (2013) combines a region proposal network with the basic CNN architecture described in Section 2.2.2. Essentially, the network first determines which regions of the image are of interest, then computes CNN features within those regions. Finally, those features are used to determine object classifications. This process flow is shown in Figure 2.7.

This method offered a significant increase in mean Average Precision (mAP) over existing state of the art. Further improvements were implemented by Fast-RCNN (Girshick, 2015) and Faster-RCNN (Ren et al., 2015).

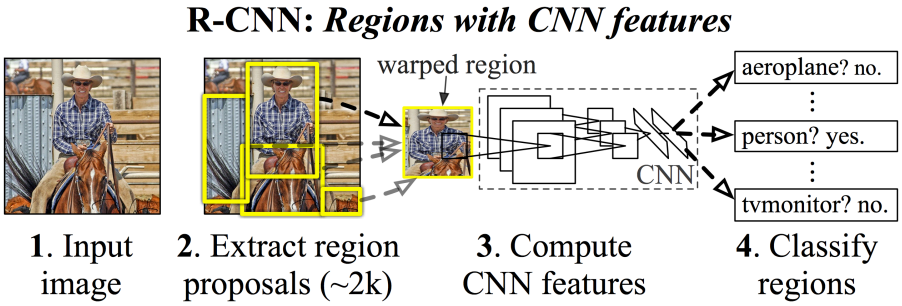


FIGURE 2.7: Processing steps for RCNN. From the (1) input image, (2) region proposals are generated and (3) CNN features are computed on each region. Finally (4) support vector machines are used to classify each region (Girshick et al., 2013).

The Single-Shot Multibox Detector (SSD) architecture proposed by Liu et al. (2016) combines creating the bounding box and generating a classification into the same step. It performs inference much faster than comparable algorithms without sacrificing accuracy; SSD achieved 74.3% mAP on a variant of the Pascal dataset, compared to Faster-RCNN with 73.2% mAP (Liu et al., 2016). The general SSD architecture is shown in Figure 2.8 (a).

However, SSD authors Liu et al. (2016) and others have noted that this general architecture is fundamentally poorly suited for detecting small objects (Zhang et al., 2017). While SSD performs on a par with other methods for overall object detection and out performs Faster-RCNN for large object detection, it performs much worse at detecting small objects. This is due to the stride (size) of the lowest level convolutional filters, leaving small objects with few or no identifiable features (Zhang et al., 2017).

2.2.5 Small Object Detection

In addition to general object detection, small object detection has been of particular interest lately as the development of safe autonomous vehicles depends on the ability to identify small objects (such as pedestrians, traffic lights, and traffic signs) at a distance with great accuracy (Cheng et al., 2018; Li et al., 2017). This section discusses some of the research has been done to adapt the architectures described in Section 2.2.4 to detect objects that occupy a small portion of the image.

As previously mentioned, SSD has poor performance at small object detection compared to other state of the art architectures. To address this, Zhang et al. have worked to develop a variant of SSD suitable for detecting small faces in images (2017). They modified the general SSD architecture to handle faces at various scales, reduce the rate of false positives, and increase the recall rate.

Efforts have also been made to adapt Faster-RCNN for small object detection. For example, Ren, Zhu, and Xiao (2018) increase small object detection accuracy through a number of modifications, such as upsampling the feature maps using bilinear interpolation and decreasing the stride of the convolutional layers.

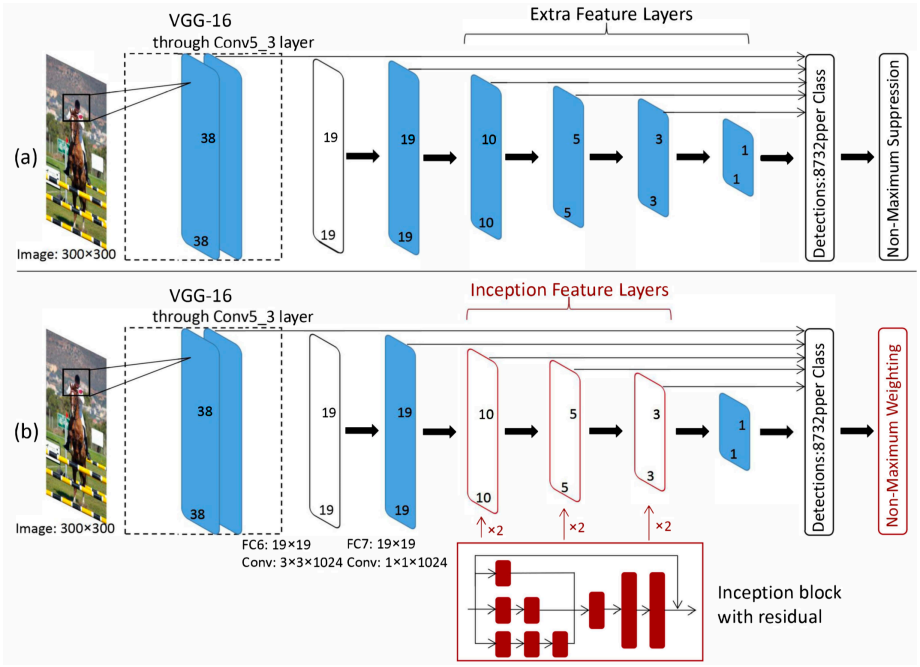


FIGURE 2.8: Architecture for (a) SSD object detection network and (b) Inception-SSD object detection network which replaces feature layers with inception modules (Ning et al., 2017).

Generative Adversarial Networks (GANs) show much promise for small object detection. Like police rushing to develop tools to catch an increasingly sophisticated criminal, GANs play one NN off another to generate sophisticated networks for purposes ranging from detecting "deepfake" videos (Knight, 2018) to generating photorealistic images from text descriptions (Zhang, Xu, and Li, 2017).

For example, Li et al. (2017) propose a system called Perceptual GAN (P-GAN) based on the Faster R-CNN architecture. One portion of this network creates "super-resolved" images of small objects that it tries to pass off as large objects, while the other (adversarial) portion tries to identify the small objects while ensuring that all images are useful for object detection. P-GAN has achieved state of the art accuracy at detecting pedestrians and small traffic signs, and could in the future be adapted to identify other types of small objects as well.

In addition to the above proposed architectures, several authors have noted that resizing the image so that small objects consist of more pixels increases detector accuracy, regardless of detector type (Liu et al., 2016; Huang et al., 2016; Chen et al., 2015). Increasing the number of pixels per object decreases the likelihood that any object will be rendered featureless by pooling or convolutional layers.

2.2.6 Car Detection in Aerial Imagery: Datasets and Benchmarks

To aid in the development of car detection algorithms, the following open source training sets and benchmark algorithms have been released.

The Vehicle Detection in Aerial Imagery (VEDAI) dataset was created to support automatic target recognition and military reconnaissance (Razakarivony and Jurie, 2015). The multi-class dataset contains 1,210 1042x1042 satellite RGB and infrared images collected at 12.5cm resolution with 2950 "small land vehicles" annotated using vehicle oriented bounding boxes. However, the architects of this dataset intentionally excluded images with "too many vehicles" such as parking lots and congested city streets which prove vital to many car counting applications such as urban traffic monitoring (Larsen, Koren, and Solberg, 2009). Additionally, all images are from Utah, USA, making models trained on this dataset unlikely to be effective in vastly different geographies. The benchmark algorithm provided with this dataset does not use CNNs.

The Cars Overhead with Context (COWC) dataset contains 32,716 positive examples of cars and 58,247 negative examples (such as car-shaped shrubs, AC units, etc) (Mundhenk et al., 2016). Greyscale or RGB imagery was collected in six international locations from a variety of aerial sensors and resampled to a uniform 15cm resolution. Annotations consist of a single coloured pixel in the centre of each object (i.e. no bounding box is provided). While the benchmark algorithms provided with this dataset do utilize CNNs, their objective is only to count the number of cars in an image.

The xView dataset contains 1,400km^2 of 30cm resolution RGB imagery with geojson formatted *ground truth* annotations across 60 distinct classes, including over 210,000 "small car" instances (Lam et al., 2018). The ground truth file contains a bounding box for each identified feature. Imagery was collected by satellite for a wide range of scenes, including ports, rural areas, and cities on five continents. This dataset was made available for a public machine learning challenge ending in August 2018 along with pre-trained model checkpoints and additional software utilities (see Section 3.3.1).

Three benchmark models trained to detect all 60 classes were released with the xView dataset: vanilla, multi-resolution, and multi-resolution augmented. As noted in the xView configuration files, all three models used the Inception-SSD architecture suggested by Ning et al. (2017) which incorporates inception modules shown in Figure 2.4. This architecture, shown in Figure 2.8 (b), outperforms SSD on Pascal VOC2007 test without increasing runtime (Ning et al., 2017).

The vanilla model was trained using 300x300pixel crops of the original images resulting in 0.3607mAP (Lam et al., 2018). While this model did not focus only on identifying small cars, it represents a strong benchmark for small car detection at 30cm resolution using CNNs.

2.3 Gap Analysis

As this chapter demonstrates, CNNs and small object detection in remotely sensed imagery are incredibly active areas of research. However, the bulk of this research

is focused on new and improved architectures and to achieving higher and higher mAP scores on standard benchmark datasets. In a sense this research is focused on questions for developers, not practitioners, and assumes an access to high resolution imagery that simply isn't affordable for many users.

Identifying meaningful relationships between resolution and accuracy could help practitioners appropriately allocate resources. Additionally, it could provide an initial inquiry into understanding the relationship between spatial resolution, object resolution, and detection accuracy more generally.

On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage (1864)

3 Resources

This chapter describes the resources required for this project. First, specifications of the hardware used is detailed. Next, the data set utilized is discussed, including motivation for its selection over similar resources. Finally, an overview of software used is provided.

3.1 Hardware

Due to the complexity of CNNs, significant computational power was required to train each model. Training and evaluation were automatically parallelized to take advantage of all available CPUs or GPUs (or in the future Google's custom 'Tensor Processing Units' (Abadi et al., 2016)), to drastically reduce the training time required (Ciresan et al., 2011). For example, models utilized in this study that took approximately 8.5 hours to train on a GPU enabled remote machine would have taken nearly straight three months to train on the author's personal laptop.

For convenience prototyping, code development, and data analysis were done on a personal laptop, while model training and inference were conducted on a GPU enabled remote machine.

3.1.1 Prototyping

Prototyping of models, including data exploration, pipeline development, and initial model testing, as well as code development and data analysis was conducted on a 2013 MacBook Pro running macOS High Sierra 10.13.6 with a 2.8 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM.

3.1.2 Training and Inference

Training and inference of the models was completed on the remote DeepLearning machine in the UCL Civil, Environmental, and Geomatic Engineering Department. This machine had three NVIDIA GeForce GTX 1080Ti GPUs and 12 Intel Core i7-6850K CPUs and was running Ubuntu 16.04. This system was accessed remotely via SSH while connected to the UCL internal network via Cisco AnyConnect Secure Mobility Client.

3.2 Data

The quality of an object detector relies in large part on the quality of training data available (Goodfellow, Bengio, and Courville, 2016). First, a high quality dataset must contain accurate, appropriately typed labels. In addition, a training dataset must contain a wide variety of different scenes (such as rural scenes, mountainous scenes, and occluded/cluttered urban scenes) in order to train a broadly applicable model. Finally, a large number of objects (and ideally a large number of training images) must be present for the model to account for the wide variability in object colour, pose, and lighting conditions (Razakarivony and Jurie, 2015).

The creation of such a dataset from scratch is often impractical as the number of objects required to provide sufficient training may exceed tens or even hundreds of thousands. However, many labelled object detection datasets have been made available via open source licenses or public machine learning competitions (Van Etten, Lindenbaum, and Bacastow, 2018; Razakarivony and Jurie, 2015; Mundhenk et al., 2016; Lin et al., 2014; Everingham et al., 2010). A number of datasets containing labelled cars in aerial imagery were identified in Section 2.2.6 and considered for this study.

Of the options surveyed, xView was selected due to the large number of annotated small cars, considerable geographic spread of images, available documentation, data processing utilities, scoring utilities, and active public challenge during the research period of this study. Unfortunately due to the Challenge, annotations were only released for the "train" portion of the dataset consisting of 378 images (less than 300km^2). Despite this limitation the dataset was determined to contain sufficient examples for analysis.

The 378 "train" images were downloaded to the DeepLearning remote machine described in Section 3.1.2 along with a ground truth file containing bounding boxes for all identified objects.

3.3 Software

This project utilized open source software, xView Challenge scripts, and self-developed code. Version numbers or specific Git commit IDs have been provided in Table 3.1 to

Name	Version	Reference
Ubuntu	16.0.4	https://www.ubuntu.com/
Bash	4.3.48	https://www.gnu.org/software/bash/
GDAL	1.11.3	(GDAL/OGR contributors, 2018)
Python	2.7.10	https://www.python.org/
TensorFlow	1.7.0	(Abadi et al., 2015)
ObjectDetection API	03/07/18; 7367d49	(Huang et al., 2016)
xView Utils	07/04/18; 71f6192	(Lam et al., 2018)
xView Baseline	30/04/18; bda0dae	(Lam et al., 2018)

TABLE 3.1: Version information (number or git commit date and ID) for software, APIs, and libraries used for training and inference on the remote DeepLearning machine.

encourage replication of results. Any modifications of external source code is noted in the appropriate Methodology section.

3.3.1 xView Code

The xView Challenge resources included the open source release of two sets of Python scripts. The first, xView Utils, contained scripts for modifying and preprocessing labelled imagery into a format that can be ingested by TensorFlow. The second, xView Baseline, contained scripts to generate object detections for an image using a saved neural network and compare the result to an existing set of ground truth boxes. These scripts were downloaded from the xView Github repository (DIUx xView, 2018).

3.3.2 TensorFlow and Object Detection API

While a number of popular open source programs exist for training NNs (e.g. Caffe, Theano, Torch (Jia et al., 2014; Bergstra et al., 2011; Collobert, Bengio, and Mariéthoz, 2002)) this project utilized TensorFlow for both training and inference. TensorFlow was chosen for a number of reasons.

First, utilization of TensorFlow for object detection is well supported and documented. In June 2017, Google released an open source Object Detection API for TensorFlow. This library functions with an existing TensorFlow installation to provide a straightforward mechanism to train an object detector using a single command (*TF Object Detection API* 2018).

Second, many pre-trained open source models are aggregated in the TensorFlow Object Detection Model Zoo, a one stop shop for model selection (*Model Zoo* 2016). Many models are available pre-trained on a standard object detection dataset, such as COCO, Kitti, or Open Images. While pre-trained models may be trained on a wide variety of objects not relevant to the current application, such as surfboards (Krasin et al., 2017) or spoons (Lin et al., 2014), training a novel object detector from

these pre-trained checkpoints can reduce overall training time without compromising model accuracy (Hoo-Chang et al., 2016).

Finally, the xView Challenge release included several fully trained model checkpoints created using the TensorFlow Object Detection API (Lam et al., 2018). Utilization of the TensorFlow Object Detection API in this dissertation allowed for meaningful comparison between the results of this study and the released xView benchmarks.

TensorFlow was installed in the prototype hardware environment (see Section 3.1.1) in a virtual environment using `pip install` for Python 2 along with all dependencies according to the standard installation instructions. It was installed on DeepLearning using a similar method. The TensorFlow Object Detection API was downloaded from the TensorFlow Github repository (*TF Installation* 2017).

3.3.3 Self-Developed Code

Self-developed code was used to automate data set generation, training tasks, and results processing. Code was developed in Python (v 2.7.10) and Bash (v 3.2.57) and is available in brief in Appendix A and in full on the author's personal Github repository (Hurst, 2018). Standard libraries used, such as `numpy`, `scipy`, and `pandas` for Python and `sed` for Bash are listed in the headers or imports of the relevant script files.

Perhaps . . . we should stop acting as if our goal is to author extremely elegant theories, and instead embrace complexity and make use of the best ally we have: the unreasonable effectiveness of data.

Alon Halevy, Peter Norvig, and Fernando Pereira (2009)

4 Methodology

Methodology for this project was broken into the following five steps, as shown in Figure 4.1:

1. *Prepare data*: xView training data was re-sampled to the appropriate spatial resolution and converted to a TensorFlow ready format.
2. *Train and evaluate*[1] *model*: Parameters for each node of the CNN were *trained* over thousands of steps. At the same time, a separate evaluation job was run to periodically evaluate the quality of the trained model's predictions on data it was *not* trained on.
3. *Export model*: The trained model was "frozen" at the optimal step and exported for inference.
4. *Run inference*: The frozen model was used to predict bounding boxes of cars on reserved test images at the appropriate resolution.
5. *Assess model quality*: The predicted vehicle bounding boxes for each test image were compared to ground truth data to assess model accuracy.

Two trials (`t1` and `t2`) were conducted. In Trial `t1` the above process was run for seven different sets of images downsampled to resolutions ranging from ~30cm to ~120cm to simulate object detection in low resolution images.

In Trial `t2` each downsampled set of images was upsampled back to the original resolution to simulate a low-resolution image that had been enlarged, and the five

[1] Some literature refers to evaluation as validation; in this paper, this step is referred to only as evaluation.

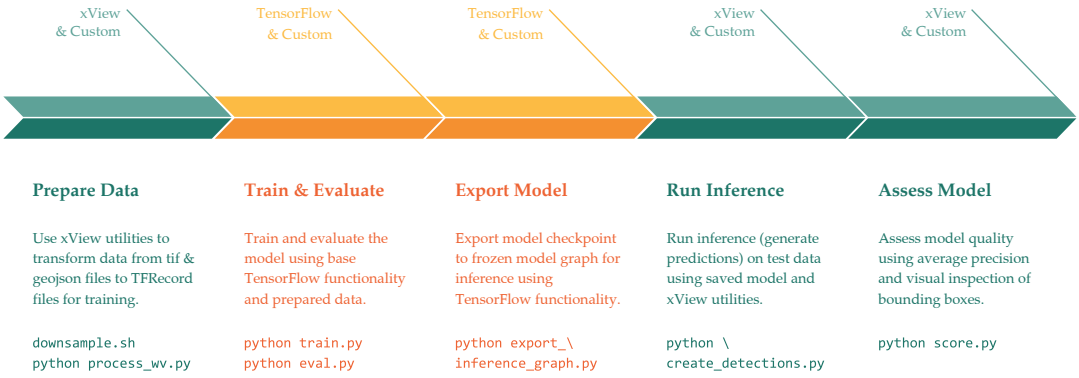


FIGURE 4.1: Methodology pipeline showing the script run for each step. Segments are coloured by the software used (in addition to self developed scripts) where orange and green indicate the use of TensorFlow Object Detection API and xView respectively.

processing steps were repeated. In total, fourteen separate models were trained, evaluated, and tested on modified versions of the xView dataset.

4.1 Data Preparation

Prior to model training, images were split into training, evaluation, and test sets. Both images and the ground truth file were resampled to the appropriate resolution for each trial. Finally, the training and evaluation sets were cropped into 300x300px *chips* and converted in to a TensorFlow ready format.

4.1.1 Data Splitting

The 387 annotated images were split for training, evaluation, and testing. Training data was used to set the parameters of the CNN (the value of each node in the network). Evaluation data was used to set the hyperparameters of the network, such as what the learning rate should be, how long the model should be run for, etc. Evaluation data was also used to ensure that the model did not overfit the training set. Finally, testing data was used *only once training and evaluation had both been completed* to determine model accuracy.

Suggested division of data between train, evaluation, and test tasks varies from 60/20/20% to 90/5/5% respectively (Ng, n.d.). Based on the size of the xView dataset, the approximate split 80/10/10% was selected. The exact proportion for each run is shown in Table 4.1.

Selection of the 38 image test set was done randomly via Code Snippet 4.1. Images were randomly sorted, then the last 38 moved to a separate test image folder.

```
ls |sort -R|tail -38|while read file; do mv $file ../test_images/; done
```

CODE SNIPPET 4.1: [Bash] Random selection of 38 test images.

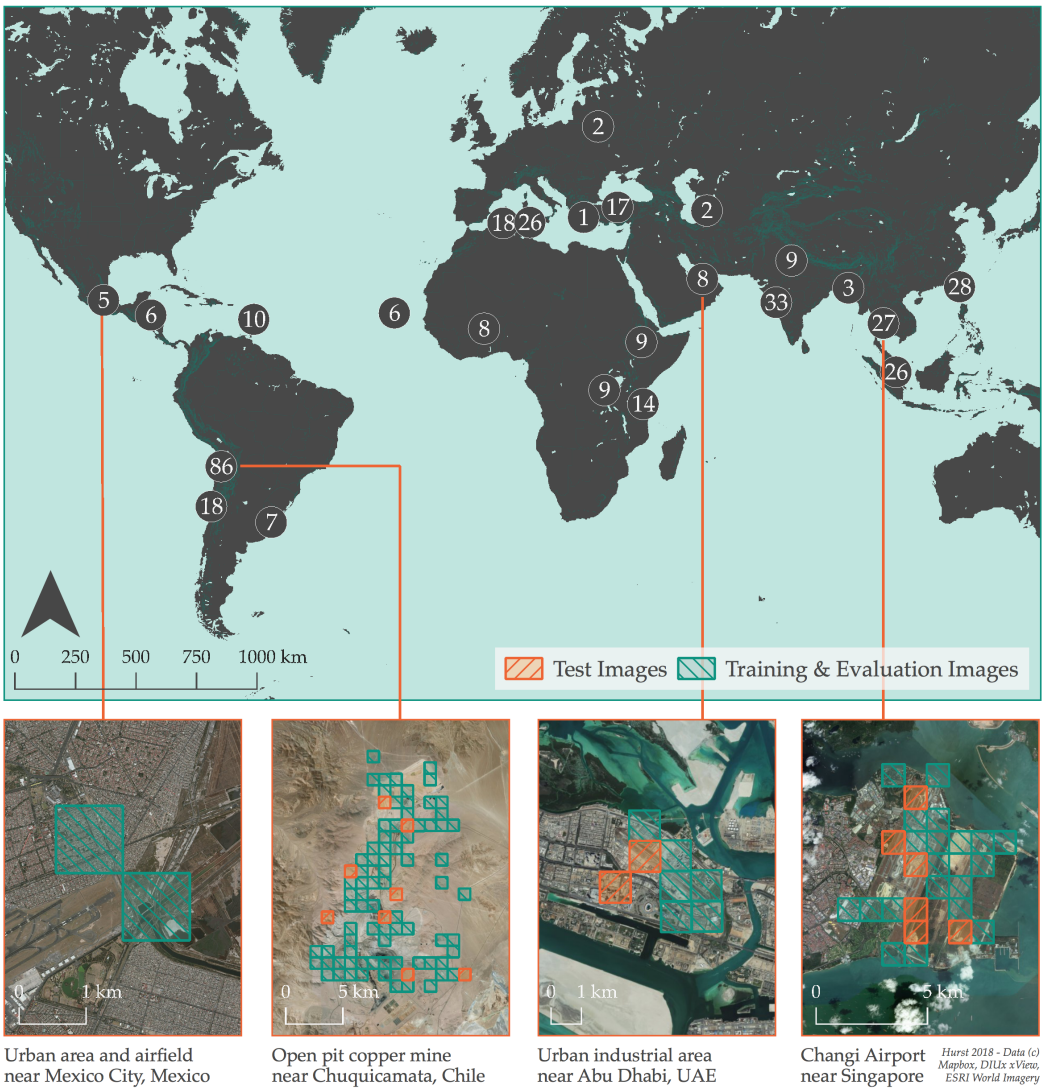


FIGURE 4.2: Locations and number of images in xView dataset utilized. Closeups showcase the variety of environments represented in the dataset as well as examples of images utilized for testing (orange).

Separation of the training and evaluation datasets was done automatically at the chip level by the xView data processing utility (see Section 4.1.4). The location of xView images and closeups of a few image clusters are shown in Figure 4.2.

4.1.2 Downsampling

To assess the impact of resolution on vehicle detection, the process of training, evaluating, and testing a model was required to be repeated on datasets of varying resolutions. However, no such dataset exists. To best mimic this labelled multi-resolution dataset, the ~30cm resolution xView dataset was downsampled several times to create full training, evaluation, and test datasets to simulate lower resolution satellite data for trial `t1`.

FIGURE 4.3: Close crop of a congested intersection in northern Tunis at original resolution (a) vs 2x downsampled (b)-(f). Note the blur effect present in (d)-(f), and the pixelated/jagged quality of (c).

Each downsampled dataset was created by reducing the image *in both the x and y direction* by a so-called downsample factor (df). The overall image size is reduced by the square of the df. Df 1.0 left the image unchanged (e.g. at ~30cm resolution), while df 2.0 reduced the image to ~60cm resolution. This study examined seven downsample factors ($df \in [1.0 : 0.5 : 4.0]$).

Consideration of potential downsampling methods was restricted to five methods available in the Geospatial Data Abstraction Library (GDAL) via the `gdalwarp` utility (GDAL/OGR contributors, 2018).

As shown in Figure 4.3 (d)-(f) some methods such as cubic spline, average value, or bilinear resampling tended to wash out important details making downsampled images increasingly blurry. Nearest neighbour (c) lead to very jagged edges. In contrast, Lanczos (b) offered "the best compromise" between these two extremes, thus meeting the study objective of simulating a low resolution image (Turkowski, 1990).

In addition to image downsampling, the ground truth file was modified as the TF Record utilities (see Section 4.1.4) relied on the pixel-location bounding box coordinates. These pixel bounding box coordinates were adjusted by simply dividing each corner coordinate by the downsample factor. Figure 4.4 compares (a) an original image with annotation with (b) a downsampled image with downsampled annotation, providing visual confirmation that the process described in this section preserves ground truth labels. Additionally, Figure 4.5 (a) shows the impact of downsampling on a yellow car.

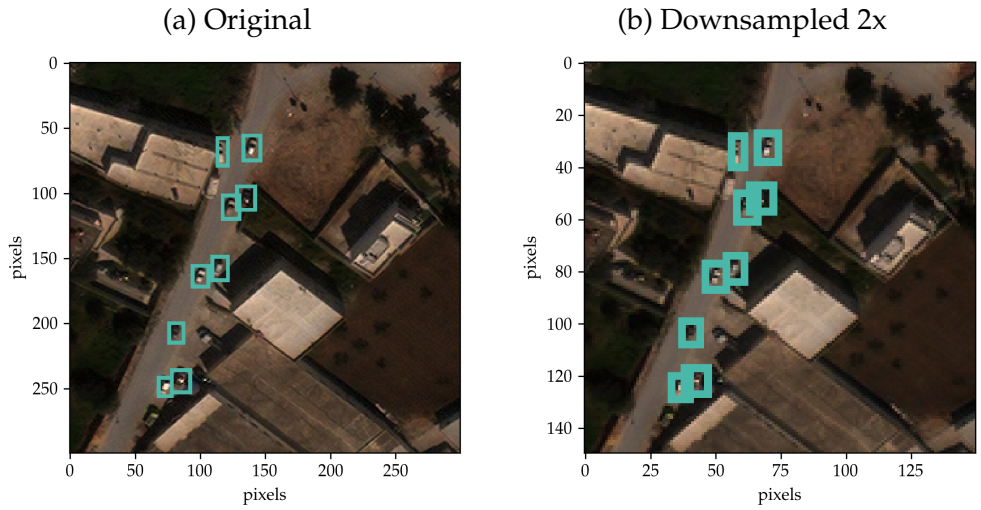


FIGURE 4.4: Comparison of (a) an annotated 300x300px image chip at the original resolution and (b) an annotated 150x150px segment of the same area with df 2.0.

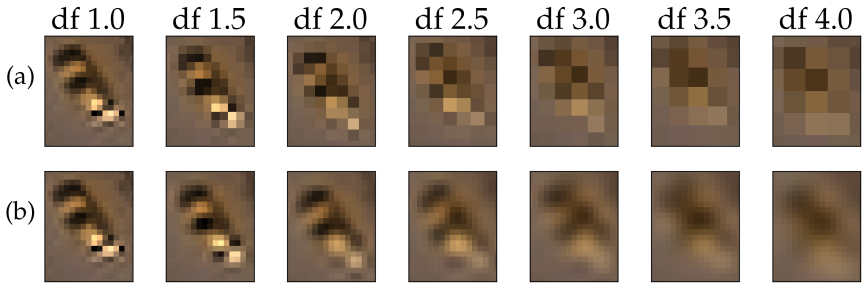


FIGURE 4.5: Example of a yellow car (a) downsampled and (b) subsequently upsampled at each of seven dfs.

4.1.3 Upsampling

In the second part of the experiment, Trial `t2`, low resolution images from Trial `t1` were upsampled back to the original ~30cm resolution using Lanczos upsampling to simulate a low resolution satellite image that had been enlarged. This was completed as described in Section 4.1.2 with df less than one (e.g. to upsample an image that had previously been downsampled by df 2.0, upsample df was set to 0.5).

This image was then combined with the original (full-resolution) annotation file to create a TF Record file as described below in Section 4.1.4. (As images had been returned to the original resolution, this is equivalent to upsampling downsampled annotation.)

The impact of upsampling on a yellow car is shown in Figure 4.5 (b). Note how the upsampled cars are more recognizable than the downsampled counterparts in (a) from which they were generated.

Downsample	Trial t1			Trial t2		
Factor	Train	Eval	Total	Train	Eval	Total
1.0	6437	691	7128	6389	739	7128
1.5	3819	337	4156	6398	730	7128
2.0	2356	241	2597	6450	678	7128
2.5	1683	109	1792	6426	702	7128
3.0	1367	24	1391	6442	686	7128
3.5	958	45	1003	6420	708	7128
4.0	742	26	768	6409	719	7128

TABLE 4.1: Number of 300x300px chips assigned to the test and evaluation splits. Due to the xView chipping utility not all proportions precisely align with the 80/10/10% goal distribution.

4.1.4 Data Formatting for TensorFlow

Before data were passed into TensorFlow for training, all images and their corresponding annotations were formatted.

First, the ground truth file was stripped of all objects except small cars. As only one class was present in this dataset, the `typeID` field was set to 1 for each bounding box (see Appendix A.1).

Second, images were broken into 300x300px non-overlapping "chips" consistent with Lam et al. (2018). Note that chip size was the same regardless of df; as Table 4.1 shows, this means that some trials had far fewer chips. Figure 4.4 (a) shows an example of one 300x300px chip with ground truth annotation.

Next, image chips were randomly shuffled then assigned to train or evaluation groups[2]. The specific number of chips used for training and evaluation for each trial and df is shown in Table 4.1.

The xView Utilities script `process_wv.py` was used to convert TIFF images and ground truth files for this dataset into two TF Record files, one for training and one for evaluation (see Code Snippet 4.2). This file format, unique to TensorFlow, combined images and annotations in the same binary file.

```
python ${xview_util_path}/process_wv.py \
    -t ${test_percent} -s ${trial}_${factor} \
    ${data_path}/train_images_${factor}/ \
    ${project_data_path}/${prefix}_${trial}_${factor}.geojson
```

CODE SNIPPET 4.2: [Bash] TF Record creation, where `factor` is downsample factor (e.g. 1.0), `trial` is trial number (e.g. `t1`), and `prefix` is object type (car). Test percent was set at 10%.

[2]The xView utilities describe these groups as train and test respectively; this nomenclature was avoided to prevent confusion between the chips used for evaluation and the 38 full sized images used for final model testing.

4.1.5 Automation of Data Preparation

Data preparation was automated through script `downsample.sh` (see Appendix A.2). For each desired resolution, the appropriate project file structure was created. Training images and the ground truth file were downsampled then processed into TF Record train and evaluation files. Finally, the output of the TF Record creation command was processed by `parse_utils.py` to set the necessary fields in the relevant model configuration file (see Appendix A.3 and Section 4.2.2).

4.2 Model Training

Once created, these TF Records were used to train each model. The TensorFlow Object Detection API streamlines much of this training process, abstracting both model creation and training minutia so that researchers can focus on selecting appropriate hyperparameters and evaluating model performance.

4.2.1 Model Selection

The TensorFlow Model Detection Zoo contains a wide variety of pre-trained models that can easily be adapted for a variety of detection tasks. For this application, the Inception Single Shot Multibox Detector (I-SSD) proposed by Ning et al. (2017) and pre-trained on the COCO dataset (Lin et al., 2014) was selected.

This model was selected for a number of reasons. First, I-SSD has low training and inference times, making it efficient for training many networks in a short time. Second, I-SSD was able to be implemented on the DeepLearning machine without any major complications. Finally, as I-SSD was used as the benchmark model by Lam et al. (2018) its use in this study enables direct performance comparison.

This model was downloaded as a `tar` file (`ssd_inception_v2 _coco_2018_01_28`) from the TensorFlow Object Detection API Model Zoo.

4.2.2 Configuration File and Hyperparameter Determination

Each model had a unique configuration file containing hyperparameters specifying how the model should be trained, such as the rate at which the model learns and the sizes of initial bounding boxes, as well as hyperparameters for model evaluation, such as the IoU threshold for a true positive. To evaluate the effectiveness of the provided I-SSD architecture, as few hyperparameters were modified as possible. Modifications to the default I-SSD configuration file are shown in Table 4.2.

4.2.3 Running Training

The TensorFlow Object Detection API simplified running model training to the execution of a single command, shown in Code Snippet 4.3. This command executed a

Parameter	Modified Value	Reason
num_classes	1	detecting only cars
fixed_shape_resizer	height/width: 300	used by Lam et al. (2018)
batch_size	32	used by Ning et al. (2017)
num_examples	see Table 4.1	equal to number of test chips
max_evals	removed	continuous evaluation

TABLE 4.2: Modifications to configuration files.

script which trained the model by iteratively improving the node weights over thousands of steps using gradient descent (Simonyan and Zisserman, 2014). Training and evaluation were automated together (see Section 4.3.4).

Each model was trained for 100,000 steps, which was empirically determined to be sufficient as most models began to overfit to their respective training data by around 40,000 steps.

During training, *checkpoints* reflecting the precise state of the model at a given step were saved periodically by storing all node weights. By default, only the last five checkpoints were retained by TensorFlow. To save all model checkpoints, the source code of `object_detection/trainer.py` in the TF Object Detection API was modified by adding `max_to_keep=0` on line 372. This allowed a model to be exported from any saved checkpoint.

```
CUDA_VISIBLE_DEVICES='0,1' python object_detection/train.py \
    --num_clones=2 --logtostderr \
    --pipeline_config_path=${config_path} \
    --train_dir=${project_path}/train
```

CODE SNIPPET 4.3: [Bash] Training job, where `config_path` points to the configuration file detailed in Section 4.2.2 and `project_path` points to the appropriate directory. Environment variable `CUDA_VISIBLE_DEVICES` was set immediately before the command to specify which GPU(s) should be used.

4.3 Model Evaluation

Evaluation and training were run concurrently to provide feedback on model quality and training completion. Evaluation helps prevent the model parameters from being overfitted to the training dataset and therefore less able to accurately identify cars in unseen images. Figure 4.7 shows a simple example of an underfitted, good fit, and overfitted model for a parabolic scatterplot. While CNNs are far more complicated, the same principles apply.

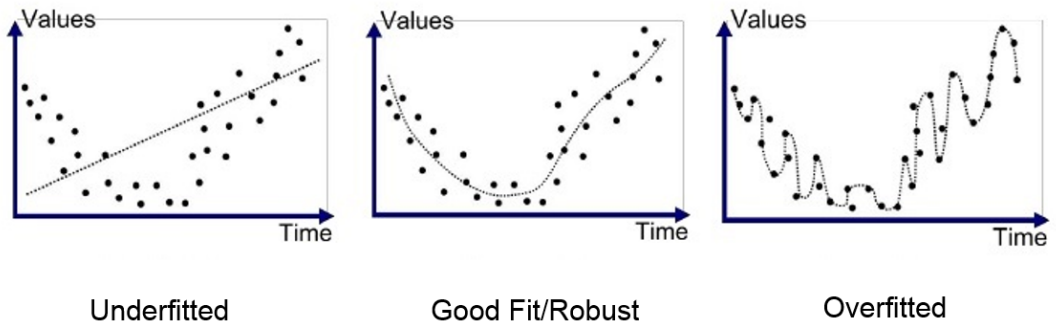


FIGURE 4.6: Example of overfitting for a simple 2-D scatterplot (Bhande, 2018). The same principle generalises to the higher dimensional NN weights.

4.3.1 Running Evaluation

Evaluation was run using a single line command shown in Code Snippet 4.4. Outputs were saved to an evaluation directory and visualized using TensorBoard (see Section 4.3.2).

```
CUDA_VISIBLE_DEVICES='2' python object_detection/eval.py \
    --logtostderr --pipeline_config_path=${config_path} \
    --checkpoint_dir=${project_path}/train \
    --eval_dir=${project_path}/eval
```

CODE SNIPPET 4.4: [Bash] Evaluation job, where `config_path` points to the configuration file detailed in Section 4.2.2 and `project_path` points to the appropriate directory. Environment variable `CUDA_VISIBLE_DEVICES` was set immediately before the command to specify which GPU(s) should be used.

4.3.2 Visualizing Training and Evaluation

TensorBoard, a utility included with TensorFlow, was used to visualize the progress of training and evaluation jobs as in Figure 4.7. Two values are visualized in Figure 4.7 (a): classification loss and localization loss. The former quantified how well the model performed at assigning the correct category to the image (in this case, "car" or "background") while the latter quantified how well model-generated bounding boxes aligned with true bounding boxes. Both of these values were computed for the evaluation data set. Figure 4.7 (b) shows the model's identification of cars in an evaluation image (at step 97899).

(a) TensorBoard Loss (b) TensorBoard Image Evaluation

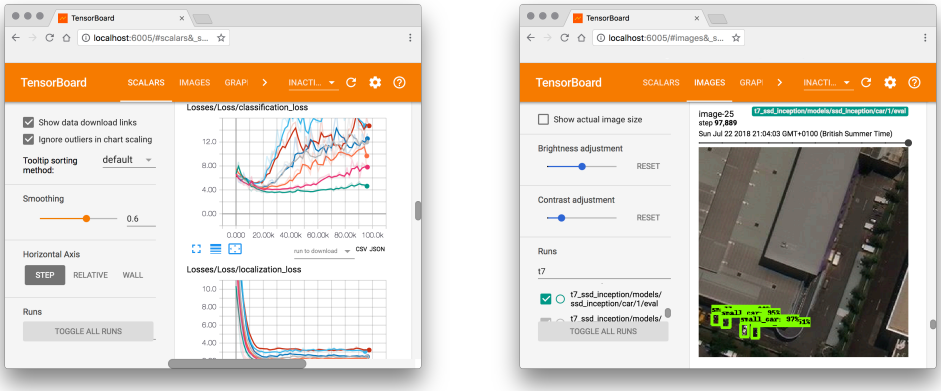


FIGURE 4.7: Screenshots of TensorBoard running remotely showing (a) classification and evaluation loss for `t1` models and (b) detections at training step 97899 by model `t1` with df 1.0.

```
tensorboard --logdir=${project_path}
```

CODE SNIPPET 4.5: [Bash] Code used to start TensorBoard. All trials located in `${project_path}` or any subfolders will be visualized.

4.3.3 Using Evaluation to Estimate Training Completion

The optimal checkpoint to use for inference was characterized by low total training loss and low localization and classification losses. Generally, total training loss decreased the longer a training job ran. While the localization loss seemed to plateau or generally decrease over time, the classification loss reached a minimum value then increased (see Figure 4.8). To avoid overfitting the model to the training dataset, the model was exported at the step corresponding to the minimum classification loss for the evaluation job.

4.3.4 Automation of Training and Evaluation

To minimize downtime on the DeepLearning machine, training and evaluation jobs were automated using the script `train_all.sh` executed in a Linux screen (see Appendix A.4). For each desired scale factor, an evaluation job was initialized in a separate detached Linux screen. The training job was then run in the current screen. After 100,000 steps, the training job completed and terminated the corresponding evaluation job.

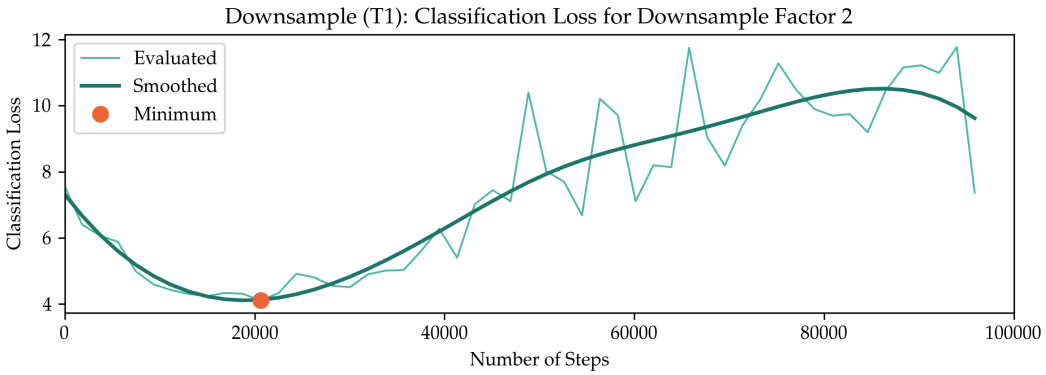


FIGURE 4.8: Determination of model completion through examination of minimum classification loss (orange). Lighter line shows observations, darker line shows the concave up trend.

4.4 Model Testing

Once each model had been trained and evaluated as described in Section 4.2 and Section 4.3, it was exported to an inference-ready format at the checkpoint corresponding to the specific step identified as described in Section 4.3.3. This was done automatically for each model via self-developed script `model_export.sh` (see Appendix A.5).

Once the model was exported, inference was run on each of the 38 images via self-developed script `run_inference.sh` (see Appendix A.6). This generated a `.txt` file for each evaluated image containing class, confidence, and coordinates for each bounding box generated by the model.

Next, these generated bounding boxes were evaluated against the ground truth and scored for a variety of IoU values. For each model and IoU value, the AP and other metrics were saved to a CSV file using self-developed script `mega_score.py` (see Appendix A.7). This script used functionality from the xView Baseline script `score.py`, wherein the `score` function was modified to return a dictionary including the following values for further analysis: map, map_score, mar_score, f1, precision, and recall. In addition, full information on precision and recall was extracted at 0.5IoU via `save_pr.py` (see Appendix A.8). One such curve is shown in Figure 2.6.

4.5 Analysis Methods

Finally, to analyse the impact of resolution on vehicle detection AP@0.5IoU was plotted against downsample factor and median car size (in pixels). Ordinary least squares was used to fit a simple 3D linear regression model (plane) to the dataset. To visualize the quality of this fit, a contour of this plane along different subsets of

the data was plotted in 2D. In addition, the correlation matrix was computed to understand the relationship between spatial resolution, object resolution, and average precision.

Machines take me by surprise with great frequency.

Alan Turing (1950)

5 Results and Analysis

This chapter presents the results of testing fourteen different CNNs. An analysis of these results, including correlation measures between the various explanatory variables and a 3D linear regression model, is also presented.

5.1 Model Inference Results

Average precision (AP) for each of the 14 models trained is shown in Table 5.1, along with the percent change in AP between different trials.

In the downsampled models trial (`t1`), AP for the model trained on the original image set (df 1.0) was 0.3029. AP for the model trained on the smallest image set (df

	Downsample (`t1`)		**Upsample (`t2`)**		**AP % Change**	
df	**AP**	**Size**	**AP**	**Size**	**↑ `t2` vs `t1`**	**↓ `t2` vs `t2`@df1.0**
1.0	0.3029	153.0	0.3198	153.0	5.6%	n/a
1.5	0.1396	80.0	0.2794	153.0	100.2%	-12.6%
2.0	0.0475	45.0	0.2653	153.0	458.2%	-17.1%
2.5	0.0239	35.0	0.2214	153.0	824.9%	-30.8%
3.0	0.0104	24.0	0.1997	153.0	1826.7%	-37.6%
3.5	0.0035	20.0	0.1635	153.0	4522.1%	-48.9%
4.0	0.0023	15.0	0.1310	153.0	5536.2%	-59.0%

TABLE 5.1: Average precision (AP), downsample factor (df), and median car size (in pixels) for each model. AP % Change shows the percent improvement of `t2` over `t1` and the precent decrease between `t2`@1.0df and all other dfs.

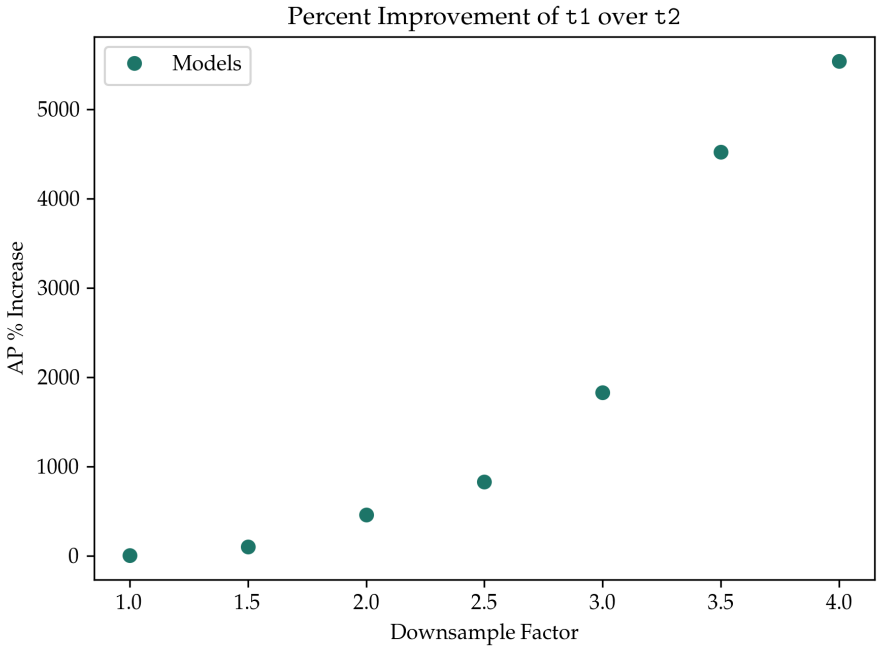


FIGURE 5.1: Percent increase in median precision in t1 vs t2.

4.0) was over 125x smaller at 0.0023. Without exception average precision decreased within t1 models by at least half as df increased by 0.5 unit steps.

In the upsampled models trial (t2), the AP for the model trained on the original image set (df 1.0) was 0.3198. AP for the model trained on images at df 4.0 was 0.1311. Similar to t1, without exception average precision decreased within t2 models as df increased.

When comparing between t1 and t2 models at the same downsample factor, the t2 models performed significantly better. As shown in the AP % Change ↑ t2 vs t1 column of Table 5.1, the improvement of t2 over t1 increased exponentially as df increased. This is also shown in Figure 5.1.

5.2 Analysis

To determine the relationship between average precision and downsample factor and object size (median pixels per car), an ordinary least squares regression was used to fit a plane to the combined dataset of all trials (see Figure 5.2). The R^2 value was 0.96.

To visually inspect how this model captured the relationship between AP and df or size for each trial, the contour of the plane was extracted along the appropriate df and size points for each trial and plotted against a scatterplot in 2D in Figure 5.3. As Figure 5.3 (a) shows, the relationship between between AP and median size for t1 is approximately linear. In contrast, the relationship between AP and df in (b) is approximated by exponential decrease. In comparison, Figure 5.3 (d) shows a linear

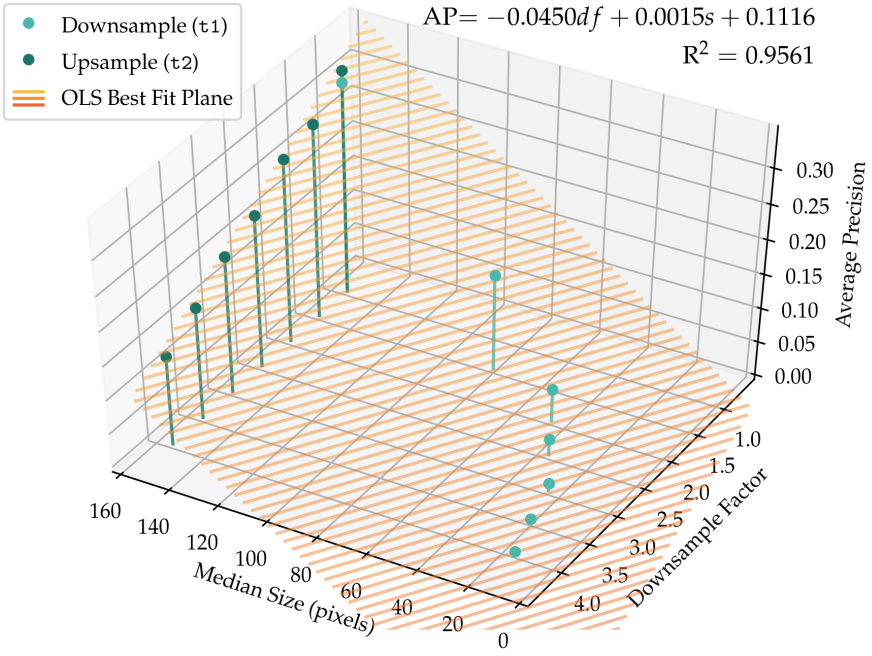


FIGURE 5.2: 3D linear regression model (orange plane) showing average precision (AP) as a function of downsample factor (df) and median car size in pixels (s).

	df	AP	Size
df	1.0000	-0.6542	-0.3337
AP	-	1.0000	0.9034
Size	-	-	1.0000

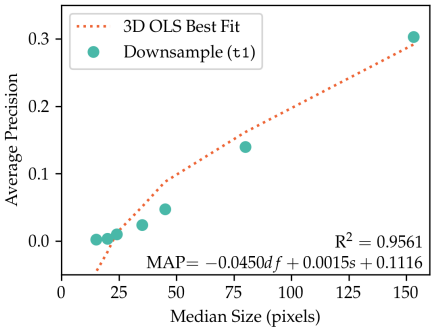
TABLE 5.2: Correlation matrix for the linear model. Note the magnitude of correlation between MAP and median size (teal) vs MAP and df (orange).

relationship between AP and df for `t2`. (Since median size did not vary between trials, all variance in model quality is attributed solely to df, as shown by (c).)

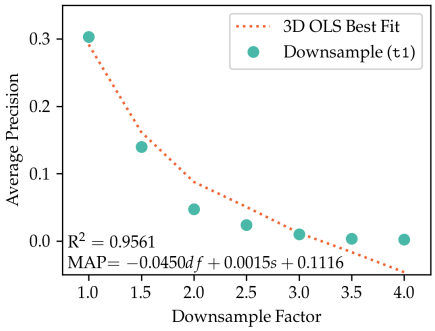
The correlation matrix in Table 5.2 quantifies the strength of the relationship between df, AP, and median size. AP is more strongly correlated with median size (0.9034) than with df (-0.6542)[1].

[1]To ensure that this correlation was not due to multicollinearity effects, the variance inflation factor (VIF) between df and size was calculated to be 1.13. As this value is less than 5, multicollinearity concerns can safely be ignored (Sheather, 2009).

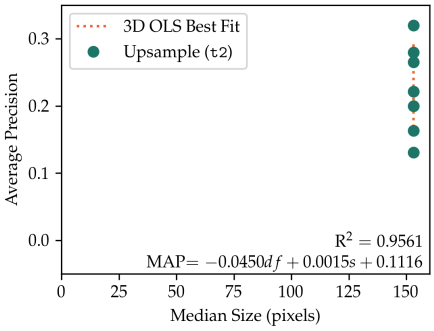
(a) Downsample (t1): AP by Size



(b) Downsample (t1): AP by df



(c) Upsample (t2): AP by Size



(d) Upsample (t2): AP by df

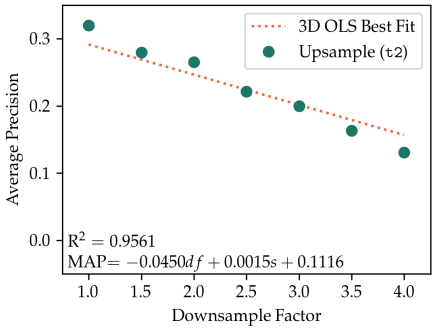


FIGURE 5.3: 3D OLS model performance. Orange dotted line shows AP value along the contour of the plane corresponding to the appropriate df and size. Figure 5.3 (a) and (c) show AP by median size for t1 and t2 respectively, while (b) and (d) show AP by df. Note how the model compares to the exponential nature of (b), the vertical line of (c), and the linear nature of (d).

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing (1950)

6 Discussion

This study has examined the impact of spatial resolution and object resolution on vehicle detection accuracy. Through the two trials described in Chapter 4 and the results and analysis detailed in Chapter 5, it has been observed that object resolution has a greater impact on car detection accuracy than spatial resolution.

This chapter discusses this primary result. First, explanations for model performance and the relationship between accuracy and resolution are considered. Second, the successes and limitations of this approach are explored. Next, the implications of this research for practitioners, developers, and those in other fields are discussed. Finally, suggestions for future work are presented.

6.1 Explanation of Results

6.1.1 Spatial Resolution

For both trials, decreasing spatial resolution led to a decrease in model quality.

This is readily explained by fundamentals of computer vision. As a CNN learns to identify objects, low level features are developed to identify fundamental components of the image, such as edges and colour gradients. As Zeiler and Fergus (2014) beautifully visualize, these filters are combined to create more complex representations of objects, such as dogs or people. Yet they fundamentally rely on the ability to detect edges and colour gradients, much as humans and birds do (Shapley and Tolhurst, 1973; Bhagavatula et al., 2009). Texture and defining features of cars are lost as even the upsampled images appear more and more blurred at higher dfs. As Figure 6.1 shows, the distinctive stripe created by a windshield becomes difficult to

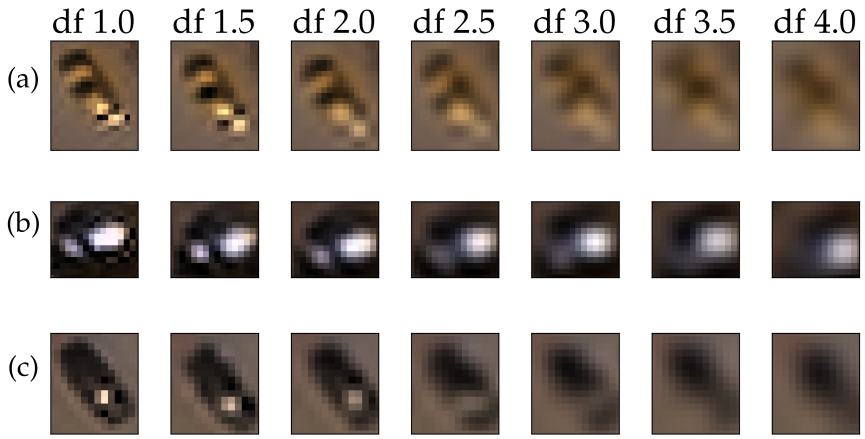


FIGURE 6.1: Upsampled images of a (a) yellow car on a neutral background, (b) white car on dark background, and (c) black car on neutral background at all dfs showing blur effect from information loss.

see as the image becomes burred, making it challenging to differentiate between vehicles and similarly shaped blobs. The importance of texture is precisely quantified by the final column of Table 5.1 (↓ `t2` vs `t2`@df1.0). The result that spatial resolution matters, then, is not surprising.

6.1.2 Object Resolution

As shown in Chapter 5, upsampled images at a given df perform better than their downsampled counterparts, implying that while spatial resolution is important, object resolution plays a crucial role in determining model quality. This is likely due to the structure of the I-SSD CNN model chosen for this study.

The first layer for object detection in I-SSD reduces the object from a spatial dimension of 300x300px to only 38x38 (Ning et al., 2017). This means that objects too small to map to a single pixel in this layer (that is, objects less than 8x8px) cannot be detected by the model as subsequent layers in the network are designed to detect larger and larger objects. Figure 6.2 shows the proportion of ground truth bounding boxes for downsampled images that are smaller than this 8 pixel threshold at each df. As this figure shows, the vast majority of bounding boxes at df 2.0 or above are simply too small to be detected.

This understanding of the I-SSD architecture could lead to potential modifications of the architecture itself for the case of small object detection, such as decreasing the size of the initial feature map to tailor the for a specific type of object as suggested by Ren, Zhu, and Xiao (2018). Additionally, this suggests that other architectures such as R-CNN might be a better fit for small object detection, as discussed in Section 2.2.5.

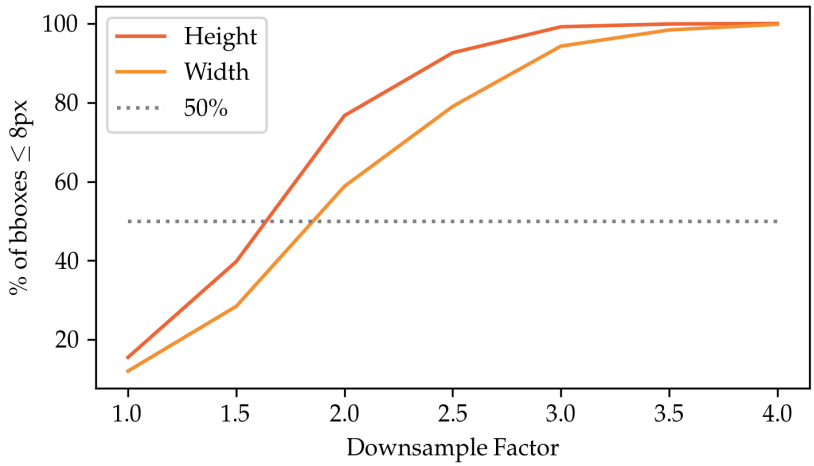


FIGURE 6.2: Approximate proportion of ground truth bounding boxes for downsampled images where height (top; dark orange) or width (bottom; yellow) is less than the 8 pixel threshold required to be detected by the smallest SSD feature map.

6.1.3 Overall Model Quality

Despite the variation in quality between different models trained for this study, even the highest quality models boasted AP no greater than 0.32, far below the level required for confident automation of small car detection.

Interestingly, all trials in this study achieved lower AP for small cars than was achieved by the vanilla xView model, despite having the same underlying architecture, with the best trial in this study achieving 0.3198AP compared to 0.3607 for the vanilla xView model (Lam et al., 2018). There are a number of explanations for this discrepancy. First, models from this study were trained on a small portion of the xView dataset as many images had not been released by DIUx with annotation. Second, hyperparameters for the vanilla xView model were carefully tweaked to optimize performance, while for this study they were not modified at all. Finally, the vanilla model may have performed better because the larger number of object classes may have led to less imbalance between the number of foreground vs background pixels (Lin et al., 2017).

Additionally, it is clear from visual inspection that models trained in this study fail to output bounding boxes with significant overlap, as in the parking lot shown in Figure 6.3. This is likely due to a layer which performs non-maximum suppression, a process whereby overlapping bounding boxes are *suppressed* from the final solution to prevent more than one box being returned for the same object (Liu et al., 2016). Suppression is defined by an IoU hyperparameter value set in the configuration file; bounding boxes that exceed this IoU value are assumed to represent the same object in the image. In this study and the xView baseline models, this value is set at 0.6. Increasing this value would allow for greater overlap between boxes, perhaps allowing for the models to better account for areas with many vehicles side

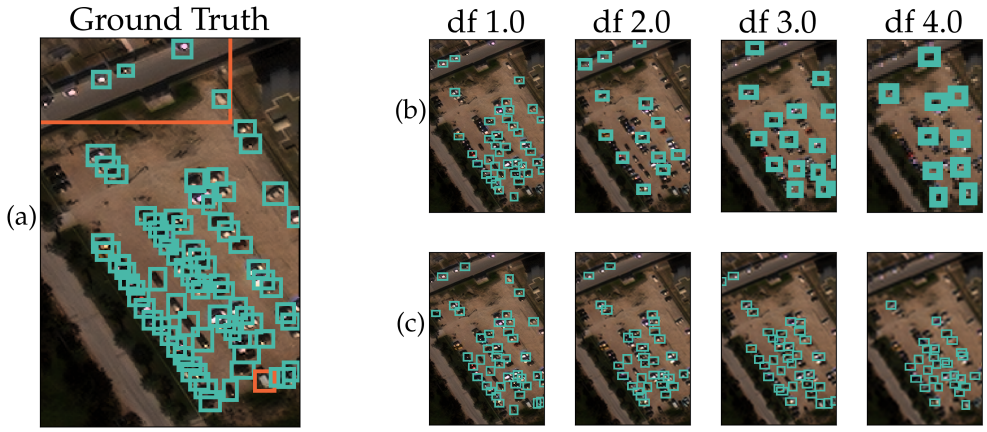


FIGURE 6.3: Comparison of (a) ground truth bounding boxes vs output for models trained on (b) downsampled and (c) upsampled imagery for a parking lot. Car bounding boxes are shown in teal, other classes shown in orange.

by side as in Figure 6.3.

6.2 Successes and Limitations

6.2.1 Successes

This study resulted in two primary successes. First, a clear trend between resolution and accuracy was identified, thus meeting the objective of this study.

Second, some models trained in this study identified vehicles that were not correctly annotated in the original dataset. For example, Figure 6.4 shows a white vehicle parked alongside a road that was not included in the original dataset but was identified by several of models trained in this study. While a thorough examination of such cases was not undertaken, this result indicates that in some cases these models correctly identified small cars that slipped past human experts undetected. (However, this does call into question the quality of the dataset. This is discussed below in Section 6.2.2.)

6.2.2 Limitations

Two primary limitations are noted regarding these results: model simplicity and dataset concerns.

Figure 5.3 shows data for each trial plotted against the relevant contour of the model plane, encouraging examination of residuals. A visual inspection indicates that the residuals for `t1` in particular do not seem independent and identically distributed, suggesting that a linear model may not be not a strong fit for the data. The dataset appears to exhibit trends (perhaps exponential, in the case of Figure 5.3 (b)) that are not accurately captured by the model. This indicates that the model is likely

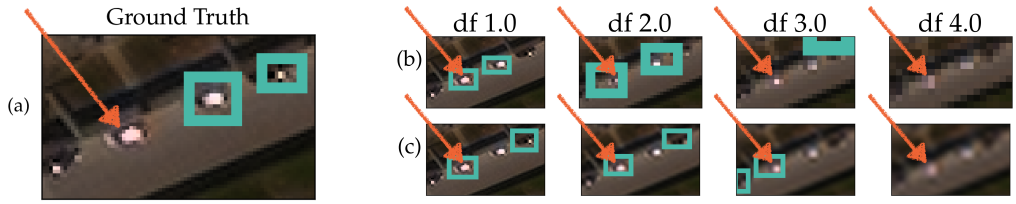


FIGURE 6.4: A car (orange arrow) not included in ground truth bounding boxes (teal) in (a) the original dataset but identified by several models trained on (b) downsampled or (c) upsampled images.

too simple for this use case. This could be mitigated by obtaining more data from other trials and attempting to fit more sophisticated functions, such as polynomial surfaces, to these data.

Some limitations also arose from the xView dataset. First, as mentioned in Section 6.2.1 the dataset does contain errors where cars were not appropriately labelled. While Lam et al. (2018) describe in detail the "high quality annotation pipeline" used to create this dataset, human error is nevertheless present. This impacted the model twice, first by decreasing available small car examples and second by suppressing model accuracy by counting detections like the one shown in Figure 6.4 as false positives, despite the clear presence of a vehicle. Second, as mentioned in Section 6.1.3 these models were trained on a subset of the full dataset. Training on the full xView dataset (to be released October 2018 or later) could increase accuracy.

6.3 Implications

This research has direct implications for practitioners. First, these findings quantify the suggestion made by many authors that increasing image resolution leads to increased accuracy for small object detection (Chen et al., 2015; Huang et al., 2016; Liu et al., 2016; Zhang et al., 2017). These increases are not modest; as Table 5.1 shows, increasing image dimensions just 1.5x doubled detector accuracy, and increasing 4x increased accuracy by over 5,000%. Moreover, implementing this finding in existing systems requires no modification to model structure and minimal modification to any data pipelines, essentially requiring only the upsampling process itself.

Next, this research has laid the groundwork for defining a multidimensional surface that models the relationship between AP and the factors that help determine it, such as object resolution, image resolution, number of examples, etc. While this relationship was modelled in 3D in this study, this could be extended to higher dimensions to account for additional variables. Creating a multidimensional surface or even a simple rule of thumb would enable practitioners to make better informed decisions about how to allocate resources. For example, given limited resources could a better model be obtained by prioritizing obtaining better imagery or laboriously creating a larger training set?

In addition, a rule of thumb would provide benefit to developers as well as practitioners as it would invite identification of underserved use cases, such as small object detection, for which niche models could be developed.

Finally, there are a variety of applications besides car detection in satellite imagery that benefit from enhanced small object detection. For example, systems to detect spiders on agricultural products like grapes and broccoli using CNNs could reduce human farm labour and the spread of pests (Menikdiwela et al., 2017). Among medical applications, detection of mitosis in breast cancer remains a difficult and time intensive task in part due to the small size of objects to be detected (Cirean et al., 2013; Araújo et al., 2017). Accurate automation of small object detection could have a significant impact on human health and safety across many industries.

6.4 Future Work

Many important questions regarding the impact of resolution on small car detection accuracy remain.

First, a number of confounding variables present were controlled during this study. Modifying these variables (i.e. reducing chip size for downsampled images) might help illuminate any unintended consequences that arose from leaving these variables constant. Optimizing hyperparameters may lead to better results as well, especially those related to allowed overlap in bounding box suggestion through non-maximum suppression (see Section 6.1.3).

Second, future research should investigate if this result holds for other types of small objects, such as ships, malignant breast cancer mitosis (Cirean et al., 2013), or spiders (Menikdiwela et al., 2017) to allow for generalisation across disciplines.

Next, the relationship between accuracy and other variables (training data, chip size, training steps, etc) should be explored. Though not specifically tied to small car detection or resolution, addressing these relationships will nevertheless help generate a more comprehensive multidimensional surface or general rule of thumb to support practitioners to allocate resources appropriately.

Finally, additional inquiries should address the generalisation of these findings to different models. Is Faster R-CNN, for example, as responsive to object resolution for small object detection as I-SSD? In addition to the research detailed in Section 2.2.5 and the suggestions made in Section 6.1.2 such as increasing the size of the initial feature map, significant additional research remains into designing CNN architectures for low resolution and small object detection.

I would like to leave you with is the following question: If you had access to imagery of the whole planet every single day, ... what problems would you solve?

Will Marshall, Planet CEO (2014)

7 Conclusion

This study has shown that resolution has a large impact on car detection accuracy in satellite imagery. This was done using Inception-Single Shot MultiBox Detector (I-SSD) convolutional neural networks (CNNs) to detect small cars in the open source xView Challenge dataset.

Two primary conclusions were drawn. First, decreased spatial resolution lead to fewer observable features in images and a reduction in texture. This in turn decreased detection accuracy, as CNNs rely on these features. This result was consistent regardless of object resolution (pixels per object).

Second, object resolution had a much larger impact on detection accuracy than spatial resolution. While this result has been suggested elsewhere, this study provides quantitative confirmation that enlarging low resolution images can dramatically improve detection accuracy without relying on any outside data.

In addition, ordinary least squares was used to model detection accuracy as a function of both spatial and object resolution. This model provides a starting point for future research into the relationship between detection accuracy and an array of other variables, such as resolution, chip size, and number of examples.

This research is useful for a variety of car detection applications, such as monitoring traffic changes, identifying a buildup of military forces, or estimating population changes following a natural disaster. In addition, these results may be generalised to detecting small objects in other disciplines, such as oncology and agriculture.

Bibliography

Abadi, Martín et al. (2016). "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16, pp. 265–283.

Abadi, Martn et al. (2015). "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". In: URL: https://www.tensorflow.org/.

Araújo, Teresa et al. (2017). "Classification of breast cancer histology images using Convolutional Neural Networks". en. In: *PLOS ONE* 12.6, e0177544. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177544. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177544.

Babbage, Charles (1864). *Passages from the Life of a Philosopher*. Longman, Green, Longman, Roberts, & Green. URL: https://books.google.co.uk/books?id=Fa1JAAAAMAAJ.

Barron, Andrew R (1993). "Universal approximation bounds for superpositions of a sigmoidal function". In: *IEEE Transactions on Information theory* 39.3, pp. 930–945.

Bergstra, James et al. (2011). "Theano: Deep learning on gpus with python". In: *NIPS 2011, BigLearning Workshop, Granada, Spain*. Vol. 3, pp. 1–48.

Bhagavatula, Partha et al. (2009). "Edge Detection in Landing Budgerigars (Melopsittacus undulatus)". en. In: *PLOS ONE* 4.10, e7301. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0007301. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0007301.

Bhande, Anup (2018). *What is underfitting and overfitting in machine learning and how to deal with it*. URL: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76.

Chen, Xiaozhi et al. (2015). "3D Object Proposals for Accurate Object Class Detection". en. In: p. 9.

Cheng, Peng et al. (2018). "LOCO: Local Context Based Faster R-CNN for Small Traffic Sign Detection". en. In: *MultiMedia Modeling*. Ed. by Klaus Schoeffmann et al. Lecture Notes in Computer Science. Springer International Publishing, pp. 329–341. ISBN: 978-3-319-73603-7.

Ciresan, Dan C et al. (2011). "Flexible, high performance convolutional neural networks for image classification". In: *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. Vol. 22, p. 1237.

Cirean, Dan C. et al. (2013). "Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks". en. In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013*. Ed. by David Hutchison et al. Vol. 8150.

Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 411–418. ISBN: 978-3-642-40762-8 978-3-642-40763-5. DOI: 10.1007/978-3-642-40763-5_51. URL: http://link.springer.com/10.1007/978-3-642-40763-5_51.

Collobert, Ronan, Samy Bengio, and Johnny Mariéthoz (2002). *Torch: a modular machine learning software library*. Tech. rep. Idiap.

DIUx xView (2018). *xView*. URL: https://challenge.xviewdataset.org/challenge-evaluation.

Everingham, Mark and John Winn (2007). "The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Development Kit". en. In: p. 23.

Everingham, Mark et al. (2010). "The Pascal Visual Object Classes (VOC) Challenge". en. In: *International Journal of Computer Vision* 88.2, pp. 303–338. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-009-0275-4. URL: https://link.springer.com/article/10.1007/s11263-009-0275-4.

GDAL/OGR contributors (2018). *GDAL/OGR Geospatial Data Abstraction software Library*. URL: http://gdal.org.

Geiger, Andreas et al. (2013). "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)*. URL: http://www.cvlibs.net/datasets/kitti/.

Girshick, Ross (2015). "Fast R-CNN". In: *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448.

Girshick, Ross et al. (2013). "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *arXiv:1311.2524 [cs]*. arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

Halevy, Alon, Peter Norvig, and Fernando Pereira (2009). "The Unreasonable Effectiveness of Data". en. In: *IEEE Intelligent Systems* 24.2, pp. 8–12. ISSN: 1541-1672. DOI: 10.1109/MIS.2009.36. URL: http://ieeexplore.ieee.org/document/4804817/.

Henderson, Paul and Vittorio Ferrari (2016). "End-to-End Training of Object Class Detectors for Mean Average Precision". en. In: *Computer Vision ACCV 2016*. Lecture Notes in Computer Science. Springer, Cham, pp. 198–213. ISBN: 978-3-319-54192-1 978-3-319-54193-8. DOI: 10.1007/978-3-319-54193-8_13. URL: https://link.springer.com/chapter/10.1007/978-3-319-54193-8_13.

Hoo-Chang, Shin et al. (2016). "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning". In: *IEEE transactions on medical imaging* 35.5, pp. 1285–1298. ISSN: 0278-0062. DOI: 10.1109/TMI.2016.2528162. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4890616/.

Huang, Jonathan et al. (2016). "Speed/accuracy trade-offs for modern convolutional object detectors". In: *arXiv:1611.10012 [cs]*. arXiv: 1611.10012. URL: http://arxiv.org/abs/1611.10012.

Hurst, Heidi (2018). *Dissertation Code Repository*. URL: https://github.com/heidimhurst/dissertation.

Inception Model (2018). URL: https://github.com/tensorflow/models/tree/master/research/inception.

Jia, Yangqing et al. (2014). "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv:1408.5093 [cs]*. arXiv: 1408.5093. URL: http://arxiv.org/abs/1408.5093.

Karpathy, Andrej (2018). *CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/convolutional-networks/.

Knight, Will (2018). *The US military is funding an effort to catch deepfakes and other AI trickery*. en. URL: https://www.technologyreview.com/s/611146/the-us-military-is-funding-an-effort-to-catch-deepfakes-and-other-ai-trickery/.

Krasin, Ivan et al. (2017). "OpenImages: A public dataset for large-scale multi-label and multi-class image classification." In: URL: https://storage.googleapis.com/openimages/web/index.html.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Lam, Darius et al. (2018). "xView: Objects in Context in Overhead Imagery". In: *arXiv:1802.07856 [cs]*. arXiv: 1802.07856. URL: http://arxiv.org/abs/1802.07856.

Larsen, Siri Øyen, Hans Koren, and Rune Solberg (2009). "Traffic monitoring using very high resolution satellite imagery". In: *Photogrammetric Engineering & Remote Sensing* 75.7, pp. 859–869.

Lawrence, S. et al. (1997). "Face recognition: a convolutional neural-network approach". en. In: *IEEE Transactions on Neural Networks* 8.1, pp. 98–113. ISSN: 10459227. DOI: 10.1109/72.554195. URL: http://ieeexplore.ieee.org/document/554195/.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". en. In: *Nature* 521.7553, pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539. URL: http://www.nature.com/articles/nature14539.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Li, Jianan et al. (2017). "Perceptual Generative Adversarial Networks for Small Object Detection". en. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, pp. 1951–1959. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.211. URL: http://ieeexplore.ieee.org/document/8099694/.

Lin, Tsung-Yi et al. (2014). "Microsoft COCO: Common Objects in Context". In: *arXiv:1405.0312 [cs]*. arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

Lin, Tsung-Yi et al. (2017). "Focal Loss for Dense Object Detection". In: *arXiv:1708.02002 [cs]*. arXiv: 1708.02002. URL: http://arxiv.org/abs/1708.02002.

Liu, Wei et al. (2016). "SSD: Single Shot MultiBox Detector". In: *arXiv:1512.02325 [cs]* 9905. arXiv: 1512.02325, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2. URL: http://arxiv.org/abs/1512.02325.

Marshall, Will (2014). *Tiny satellites show us the Earth as it changes in near-real-time*. en. URL: https://www.ted.com/talks/will_marshall_teeny_tiny_satellites_that_photograph_the_entire_planet_every_day.

— (2018). *The mission to create a searchable database of Earth's surface*. en. URL: https://www.ted.com/talks/will_marshall_the_mission_to_create_a_searchable_database_of_earth_s_surface.

Menikdiwela, M. et al. (2017). "CNN-based small object detection and visualization with feature activation mapping". In: *2017 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pp. 1–5. DOI: 10.1109/IVCNZ.2017.8402455.

Mitchell, Tom M. (1997). *Machine Learning*. en. McGraw-Hill series in computer science. New York: McGraw-Hill. ISBN: 978-0-07-042807-2.

Mundhenk, T. Nathan et al. (2016). "A Large Contextual Dataset for Classification, Detection and Counting of Cars with Deep Learning". en. In: *Computer Vision ECCV 2016*. Ed. by Bastian Leibe et al. Vol. 9907. Cham: Springer International Publishing, pp. 785–800. ISBN: 978-3-319-46486-2 978-3-319-46487-9. DOI: 10.1007/978-3-319-46487-9_48. URL: http://link.springer.com/10.1007/978-3-319-46487-9_48.

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814.

Ng, Andrew. *Train / Dev / Test sets - Practical aspects of Deep Learning*. en. URL: https://www.coursera.org/lecture/deep-neural-network/train-dev-test-sets-cxG1s.

Ning, Chengcheng et al. (2017). "Inception Single Shot MultiBox Detector for object detection". In: *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, pp. 549–554. DOI: 10.1109/ICMEW.2017.8026312.

Ouaknine, Arthur (2018). *Review of Deep Learning Algorithms for Object Detection*. URL: https://medium.com/comet-app/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852.

Razakarivony, Sébastien and Frédéric Jurie (2015). "Vehicle Detection in Aerial Imagery : A small target detection benchmark". In: *Journal of Visual Communication and Image Representation, Elsevier*. URL: https://hal.archives-ouvertes.fr/hal-01122605.

Ren, Shaoqing et al. (2015). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *arXiv:1506.01497 [cs]*. arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

Ren, Yun, Changren Zhu, and Shunping Xiao (2018). "Small Object Detection in Optical Remote Sensing Images via Modified Faster R-CNN". en. In: *Applied Sciences* 8.5, p. 813. ISSN: 2076-3417. DOI: 10.3390/app8050813. URL: http://www.mdpi.com/2076-3417/8/5/813.

Rosebrock, Adrian (2016). *Intersection over Union (IoU) for object detection*. en-US. URL: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.

Russell, Stuart J and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited.

Shapley, R. M. and D. J. Tolhurst (1973). "Edge detectors in human vision". In: *The Journal of Physiology* 229.1, pp. 165–183. ISSN: 0022-3751. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1350218/.

Sheather, Simon (2009). *A modern approach to regression with R*. Springer Science & Business Media.

Shi, Xinchu et al. (2012). "Context-driven moving vehicle detection in wide area motion imagery". In: *Pattern Recognition (ICPR), 2012 21st International Conference on*, pp. 2512–2515.

Simonyan, Karen and Andrew Zisserman (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv:1409.1556 [cs]*. arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

Szegedy, Christian et al. (2015). "Rethinking the Inception Architecture for Computer Vision". In: *arXiv:1512.00567 [cs]*. arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

Model Zoo (2016). *TensorFlow Detection Model Zoo*. URL: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.

TF Installation (2017). *Tensorflow Installation*. URL: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/installation.md.

TF Object Detection API (2018). *TensorFlow Object Detection API*. URL: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.

Turing, A. M. (1950). "Computing Machinery and Intelligence". In: *Mind, New Series* 59.236, pp. 433–460. URL: http://www.jstor.org/stable/2251299.

Turkowski, Ken (1990). "Filters for Common Resampling Tasks". en. In: *Graphics Gems*. Elsevier, pp. 147–165. ISBN: 978-0-08-050753-8. DOI: 10.1016/B978-0-08-050753-8.50042-5. URL: http://linkinghub.elsevier.com/retrieve/pii/B9780080507538500425.

Van Etten, Adam, Dave Lindenbaum, and Todd M. Bacastow (2018). "SpaceNet: A Remote Sensing Dataset and Challenge Series". In: *arXiv:1807.01232 [cs]*. arXiv: 1807.01232. URL: http://arxiv.org/abs/1807.01232.

Velikovi, Petar (2016). *Complete collection of my PGF/TikZ figures*. URL: https://github.com/PetarV-/TikZ.

Wolfinbarger, Susan, Jonathan Drake, and Eric Ashcroft (2014). *Satellite Imagery Assessment of the Crisis in Ukraine Part Two: Border Deployments*. URL: https://www.aaas.org/sites/default/files/Ukraine_Border_Deployments.pdf.

Zeiler, Matthew D. and Rob Fergus (2014). "Visualizing and Understanding Convolutional Networks". en. In: *Computer Vision ECCV 2014*. Ed. by David Fleet et al. Vol. 8689. Cham: Springer International Publishing, pp. 818–833. ISBN: 978-3-319-10589-5 978-3-319-10590-1. DOI: 10.1007/978-3-319-10590-1_53. URL: http://link.springer.com/10.1007/978-3-319-10590-1_53.

Zhang, Han, Tao Xu, and Hongsheng Li (2017). "StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks". en. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. Venice: IEEE, pp. 5908–5916. ISBN: 978-1-5386-1032-9. DOI: 10.1109/ICCV.2017.629. URL: http://ieeexplore.ieee.org/document/8237891/.

Zhang, Shifeng et al. (2017). "S^{3}FD: Single Shot Scale-invariant Face Detector". In: *arXiv:1708.05237 [cs]*. arXiv: 1708.05237. URL: http://arxiv.org/abs/1708.05237.

Zhao, Tao and Ram Nevatia (2003). "Car detection in low resolution aerial images". In: *Image and Vision Computing* 21.8, pp. 693–703. ISSN: 0262-8856. DOI: 10.1016/S0262-8856(03)00064-7. URL: http://www.sciencedirect.com/science/article/pii/S0262885603000647.

A Code Appendix

A.1 JSON Annotation Utilities (Python)

```python
# Original work 2018 copyright HMHurst, heidimhurst.github.io
# Available under Apache 2.0 License

import json
import math
import argparse
from tqdm import tqdm

'''
Helper functions to read in xView dataset geojson label file and
process appropriately, including extracting just one type of object
and downsampling the bounding boxes for objects.
'''

def clean_types(input_dict):
    """
    Sets all type_ids to 1, for simplicity.  Do not use with multiple type IDs!

    Args:
        input_dict: dictionary of geojson content in xView schema

    Output:
        returns dictionary
    """

    for item in input_dict['features']:
        item['properties']['type_id'] = 1

    return input_dict

def parsetype_dict(input_dict, type_id=[18]):
    """
```

```
    Extracts features with specific type ids from dictionary.

    Args:
        input_dict: dictionary of geojson content in xView schema
        type_id: list of integers of categories of interest (see
 ↪  https://github.com/DIUx-xView/baseline/blob/master/xview_class_labels.txt)

    Output:
        returns dictionary
    """

    # Filter python objects with list comprehensions
    input_dict['features'] = [x for x in input_dict['features'] if
     ↪  x['properties']['type_id'] in type_id]

    return input_dict

def parsetype(input_json="../data/xView_train.geojson",
              output_file='output_json_test.geojson',
              type_id=[18]):
    """
    Extracts entries in xview_train geojson corresponding to given type id and
 ↪  resaves
    as new geojson.

    Args:
        input_json: filepath to original xView training geojson
        output_file: filepath (including extension) to desired output geojson file
        type_id: list of integers of categories of interest
                 (see
 ↪  https://github.com/DIUx-xView/baseline/blob/master/xview_class_labels.txt)

    Output:
        saves new .geojson file to disk
    """

    # read input file
    with open(input_json) as f:
        output_dict = json.load(f)

    # Filter python objects with list comprehensions
    output_dict = parsetype_dict(output_dict, type_id)

    # Transform python object back into json
    output_json = json.dumps(output_dict)

    # write output_json to file for future use
    with open(output_file, 'w') as outfile:
        json.dump(output_dict, outfile)


def downsample_bbox_dict(input_dict, scale_factor=2.0):
    """
    Performs downsampling functionality for geojson format.

    Args:
        input_dict: dictionary of geojson in xView structure (must contain
 ↪  coordinates
                    in input_dict['features'][i]['properties']['bounds_imcoords'])
        scale_factor: amount that the image is reduced by in EACH DIRECTION.
                      e.g. scale factor of 2 reduces the overall image size by 1/4

```

```
    Output:
        Returns same dictionary with coordinates divided by scale_factor in each direction.
    """

    # need to modify bounds_imcoords for all features
    for i in tqdm(range(len(input_dict['features']))):
        # get coordinate string & parse (format is xmin,ymin,xmax,ymax)
        coords = [int(x) for x in input_dict['features'][i]['properties']['bounds_imcoords'].split(',')]

        # divide all coords by scale factor, taking floor of mins and ceil of maxes
        out_coords = str(int(math.floor(coords[0]/scale_factor))) + "," + \
                str(int(math.floor(coords[1]/scale_factor))) + "," + \
                str(int(math.ceil(coords[2]/scale_factor))) + "," + \
                str(int(math.ceil(coords[3]/scale_factor)))

        # modify dictionary
        input_dict['features'][i]['properties']['bounds_imcoords'] = out_coords

    return input_dict


def downsample_bbox(input_json="../data/xView_train.geojson",
                    output_file='output_json_test.geojson',
                    scale_factor=2.0):
    """
    Downsamples the bounding boxes present in training geojson file to match downsampled image.
    Saves as new geojson.

    Args:
        input_json: filepath to original xView training geojson
        output_file: filepath (including extension) to desired output geojson file
        scale_factor: amount that the image is reduced by in EACH DIRECTION.
                    e.g. scale factor of 2 reduces the image size by 1/4

    Output:
        saves new .geojson file to disk
    """

    # read input file
    with open(input_json) as f:
        output_dict = json.load(f)

    # perform downsampling
    output_dict = downsample_bbox_dict(output_dict, scale_factor)

    # Transform python object back into json
    output_json = json.dumps(output_dict)

    # write output_json to file for future use
    with open(output_file, 'w') as outfile:
        json.dump(output_dict, outfile)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("input", help="Path to original json")
    parser.add_argument("-s", "--scale_factor", type=float, help="Factor by which JSON should be reduced in each direction")
    parser.add_argument("-o","--output", default="modified_bounds.geojson", help="Filepath of desired output")
```

```
147     parser.add_argument("-t","--type_id", nargs='+', type=int, help="List of type
        ↪  ids to be selected")
148     parser.add_argument("-c","--clean_types", action='store_true', help="Set all
        ↪  type ids to 1")
149
150     args = parser.parse_args()
151
152     # read input file
153     with open(args.input) as f:
154         output_dict = json.load(f)
155
156     # filter if required
157     if args.type_id:
158         output_dict = parsetype_dict(output_dict, args.type_id)
159
160     # set all type values to 1 if required
161     if args.clean_types:
162         print "WARNING: Removing all type ids, setting type to 1"
163         output_dict = clean_types(output_dict)
164
165     # downsample if required
166     if args.scale_factor:
167         output_dict = downsample_bbox_dict(output_dict, float(args.scale_factor))
168
169     # export
170     if args.output:
171         # Transform python object back into json
172         output_json = json.dumps(output_dict)
173
174         # write output_json to file for future use
175         with open(args.output, 'w') as outfile:
176             json.dump(output_dict, outfile)
```

A.2 Downsample (Bash)

```
1   #!/bin/bash
2
3   # project variables
4   # define project trial in initial command call (e.g. ./downsample.sh t8) to avoid
    ↪  default being set
5   trial=${1:-t7}
6   # factor=1.5
7
8   id=18 #89 #18: small car; 89: shipping container lot
9   object_type=car
10  prefix=car #or cars or whatever
11
12  base_path=/home/uceshhu
13
14  project_path=${base_path}/runs/${trial}_ssd_inception
15  data_path=${base_path}/xview_data
16  personal_util_path=${base_path}/personal_codebase/dissertation
17  xview_util_path=${base_path}/xview_codebase/data_utilities
18  model_name=/ssd_inception
19
20  # scale factors of interest
21  # declare -a sf=("1.5" "2.5" "3" "3.5")
22  declare -a sf=(1 1.5 2 2.5 3 3.5 4)
```

```
method="mode"

echo "Processing downsampling for ${object_type}-type objects - trial ${trial} using method: ${method}"

# loop through each of these
for i in ${sf[@]}; do
        factor=$i
        config_name=${prefix}_${trial}_${factor}.config
        config_path=${project_path}/models${model_name}/${config_name}
        project_data_path=${project_path}/data/${object_type}/${factor}
        # let user know what's up
        echo ----------- Creating images for downsample factor $i -----------
        # create folder for downsampled images
        mkdir ${data_path}/train_images_${method}_${factor}
        # downsample images
        python ${personal_util_path}/downsample.py \
        -i ${data_path}/train_images \
        -o ${data_path}/train_images_${method}_${factor} \
        -r ${method} -s ${factor}
        # create file structure
        echo ----------- Creating required folders for object $object_type at factor $i -----------
        interim_path=${project_path}/models${model_name}/${object_type}/${factor}
        mkdir ${interim_path}
        mkdir ${interim_path}/eval ${interim_path}/train
        mkdir $project_data_path
        # create JSON file
        echo ----------- Creating JSON file for downsample factor $i -----------
        python ${personal_util_path}/json_annotation_utilities.py \
        ${data_path}/xView_train.geojson \
        -c -t ${id} -s ${factor} \
        -o ${project_path}/data/${object_type}/${factor}/${prefix}_${trial}_${factor}.geojson
        # create appropriate config files, modify as needed
        echo ----------- Creating project config file ${config_path} -----------
        cp ${project_path}/models${model_name}/${trial}_template.config ${config_path}
        # replace train path
        sed -i "s+{TFR_TRAIN}+\"${project_path}/data/${object_type}/${factor}/xview_train_${trial}_${factor}.record\"+g" ${config_path}
        # # replace evaluation path
        sed -i "s+{TFR_EVAL}+\"${project_path}/data/${object_type}/${factor}/xview_test_${trial}_${factor}.record\"+g" ${config_path}
        # # replace label map path
        sed -i "s+{LABEL}+\"${project_path}/data/${object_type}/map_${trial}.pbtxt\"+g" ${config_path}
        # # replace model checkpoint
        sed -i "s+{CHECKPOINT}+\"${project_path}/models${model_name}/model.ckpt\"+g" ${config_path}
        # create tfrecord files
        echo ----------- Creating tfrecord files for factor $i -----------
        # pipe output to a file for processing
        # make sure you're in the correct directory
        cd ${project_data_path}
        # create tfrecord file and pipe input to save for later processing
        python ${xview_util_path}/process_wv.py -t .1 -s ${trial}_${factor} \
        ${data_path}/train_images_${method}_${factor}/ \
        ${project_data_path}/ ${prefix}_${trial}_ ${factor}.geojson 2>&1 | tee ${project_data_path}/processing_output.txt
        # process information from tf record file
```

```
        echo ----------- Processing TF Record Output Info -----------
        # run parse utilities
        python ${personal_util_path}/parse_utils.py \
        ${project_data_path}/processing_output.txt \
        -o ${project_path}/data/${object_type}/${object_type}_processing.csv \
        -f ${factor} -c ${config_path}

done
```

A.3 Parse Utils (Python)

```
"""
Copyright 2018 Heidi M Hurst github.io/heidimhurst
All rights reserved.
"""

from subprocess import call
import csv, argparse

def parse_file(info_file):
    """
    Parses an info file created by the chipping utility to create a txt file
    with all the information we want in it yay.

    Args:
        info_file: path to file containing outputs from process_wv.py
    Returns:
        test: int number of test chips
        train: int number of training chips
        chips: int total number of chips
        boxes: int total number of objects
    """

    with open(info_file, 'r') as info:
        # read in as a string
        data = info.read().replace('\n', '')
        # find substring containing chip information
        test = int(data[data.rfind(':') + 1:data.rfind('test')].strip())
        # cut off last info section of data
        data = data[:data.rfind('INFO')]
        # find substring iwth training information
        train = int(data[data.rfind(':') + 1:data.rfind('train')].strip())
        # total number of chips
        chips = int(data[data.find("Chips:")+6:data.rfind("INFO")].strip())
        # cut off last info section of data
        data = data[:data.rfind('INFO')]
        # total number of boxes
        boxes = int(data[data.find("Tot Box:")+8:data.rfind("INFO")].strip())
            return test, train, chips, boxes


def modify_config(config_path, test):
    """
    Modifies a specified config file with the correct number of test chips.

    Args:
        config_path: path to configuration file
        test: integer number of test chips
```

```
48
49      Returns:
50          no return
51      """
52
53      call(['sed -i "s+{NUM}+'+str(test)+'+g" '+config_path], shell=True)
54
55  def write_results(factor=-1, results=(-1,-1,-1,-1), out_file='tfrecord_info.csv'):
56      """
57      Writes statistics from info file to end of outfile.
58
59      Args:
60          results: integer touple (test, train, chips, boxes) from parse_file (-1 as
     ↪  default)
61          out_file: path to csv file where info should be written
62
63      Returns:
64          none
65      """
66      # write processed results to formatted csv file in order factor test train chips
     ↪  boxes
67      with open(out_file, 'a') as o:
68              writer = csv.writer(o, delimiter=',')
69              writer.writerow([factor] + list(results))
70
71
72  def process_info_file(info_file, factor=-1, out_file=None, config_path=None):
73      """
74      Reads in info file, processes to csv outfile, and modifies requisite config
     ↪  path.
75
76      Args:
77          info_file: path to file containing outputs from process_wv.py
78          out_file: path to csv file where info should be written
79          config_path: path to configuration file to be modified
80
81      Returns:
82          no returns.
83      """
84
85      # extract information from info file
86      results = parse_file(info_file)
87
88      # write results to outfile, if desired
89      if out_file:
90          write_results(factor, results, out_file)
91
92      # process config file, if desired
93      if config_path:
94          modify_config(config_path, results[0])
95
96  if __name__ == "__main__":
97      parser = argparse.ArgumentParser()
98      parser.add_argument("input", help="Path to output file containing information")
99      parser.add_argument("-o","--output_csv", help="Desired csv output, with
     ↪  location")
100     parser.add_argument("-c","--config_path", help="Path to configuration file to
     ↪  change")
101     parser.add_argument("-f", "--factor", type=float, help="Downsample factor")
102     args = parser.parse_args()
103
104     process_info_file(args.input, args.factor, args.output_csv, args.config_path)
```

A.4 Train All (Bash)

```
#!/bin/bash

# initialize required variables
trial=t7
object_type=car
prefix=car
base_path=/home/uceshhu
project_path=${base_path}/runs/t7_ssd_inception
data_path=${base_path}/xview_data
personal_util_path=${base_path}/personal_codebase/dissertation
xview_util_path=${base_path}/xview_codebase/data_utilities
model_name=/ssd_inception
models_research_path=${base_path}/tf_api_codebase/models/research

# scale factors of interest
declare -a sf=(1 1.5 2 2.5 3 3.5 4)

# loop through each of these and train/eval
for i in ${sf[@]}; do
        # setup variables
        factor=$i
        config_name=${prefix}_${trial}_${factor}.config
        config_path=${project_path}/models${model_name}/${config_name}
        project_data_path=${project_path}/data/${object_type}/${factor}

        screen_name=${trial}_eval_${factor}

        # initialize screen
        screen -L -d -m  -S ${screen_name}

        # create eval command
        eval_command="cd ${models_research_path} && export
         ↪  PYTHONPATH=$PYTHONPATH:${models_research_path}:
         ↪  ${models_research_path}/slim && CUDA_VISIBLE_DEVICES='2' python
         ↪  object_detection/eval.py \
        --logtostderr \
        --pipeline_config_path=${config_path} \
        --checkpoint_dir=${project_path}/ models${model_name}/
 ↪  ${object_type}/${factor}/train \
        --eval_dir=${project_path}/models${model_name}/${object_type}/
 ↪  ${factor}/eval"

        # start evaluation in a different screen
        screen -S $screen_name -X stuff "${eval_command}"$(echo -ne '\015')

        # move to requisite folder
        cd ${models_research_path}
        export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim

        # run a train job IN THIS SCREEN
        CUDA_VISIBLE_DEVICES='0,1' python object_detection/train.py --num_clones=2
         ↪  --logtostderr \
        --pipeline_config_path=${config_path} \
        --train_dir=${project_path}/models${model_name}/
         ↪  ${object_type}/${factor}/train

        # kills screen of eval command, killing eval command
        screen -S ${screen_name} -X quit
done
```

A.5 Model Export (Bash)

```
#!/bin/bash

trial=t7
object_type=car
prefix=car

base_path=/home/uceshhu

project_path=${base_path}/runs/${trial}_ssd_inception
data_path=${base_path}/xview_data
personal_util_path=${base_path}/personal_codebase/dissertation
xview_util_path=${base_path}/xview_codebase/data_utilities
model_name=/ssd_inception

models_research_path=${base_path}/tf_api_codebase/models/research

declare -a factors=(1 1.5 2 2.5 3 3.5)
declare -a step_val=(33855  20816  20639  16869  13213  11311)

cd ${models_research_path}
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim

for i in {0..5}; do
^^Ifactor=${factors[i]}
^^Iconfig_name=${prefix}_${trial}_${factor}.config
^^Iconfig_path=${project_path}/models${model_name}/${config_name}

^^Istep=${step_val[i]}

^^Itrained_checkpoint_prefix=${project_path}/models${model_name}/${object_type}/
 ↪  ${factor}/train/model.ckpt-${step}
^^Iexport_dir=${project_path}/models${model_name}/${object_type}/${factor}/
 ↪  saved_model_${step}/


^^IINPUT_TYPE=image_tensor

^^Iecho $config_path
^^Iecho $trained_checkpoint_prefix
^^Iecho $export_dir
^^Ipython object_detection/export_inference_graph.py \
^^I    --input_type=${INPUT_TYPE} \
^^I    --pipeline_config_path=${config_path} \
^^I    --trained_checkpoint_prefix=${trained_checkpoint_prefix} \
^^I    --output_directory=${export_dir}
done
```

A.6 Run Inference (Bash)

```
#!/bin/bash

# setup definitions
trial=t9
object_type=car
prefix=car #or cars or whatever
```

```
7
8  base_path=/home/uceshhu
9
10 project_path=${base_path}/runs/${trial}_ssd_inception
11 data_path=${base_path}/xview_data
12 personal_util_path=${base_path}/personal_codebase/dissertation
13 xview_util_path=${base_path}/xview_codebase/baseline/inference
14 model_name=/ssd_inception
15
16 models_research_path=${base_path}/tf_api_codebase/models/research
17
18 declare -a factors=(1 1.5 2 2.5 3 3.5 4)
19
20 cd ${models_research_path}
21 export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
22
23 cd ${xview_util_path}
24
25 # i=1
26 # loop through all factors of interest
27 for i in ${factors[@]}; do
28         echo ------------------- $i --------------------
29         # set directory location of saved model, test images, etc
30         # path to frozen_model.pb
31         saved_model_path=${project_path}/models${model_name}/${object_type}/${i}/
           ↪  saved_model_*/frozen_inference_graph.pb
32         # path to images
33         test_image_path=${data_path}/test_images_upsample_${i}
34         # folder to save .txt. files to
35         output_folder=${project_path}/models${model_name}/${object_type}/${i}
           ↪  /test_detections
36
37         # make folder if it doesn't exist
38         mkdir $output_folder
39
40         echo $saved_model_path
41         echo $output_folder
42
43         cd ${test_image_path}
44         # loop through filenames
45         for filename in *.tif; do
46                 echo 'FILENAME: '$filename
47                 bname=$(basename $filename)
48                 echo 'BASENAME: '$bname
49                 output_path=${output_folder}/${bname%.*}.tif.txt
50
51                 echo 'OUTPUT PATH: '$output_path
52                 # run inference
53                 CUDA_VISIBLE_DEVICES='1,2' python
                   ↪  ${xview_util_path}/create_detections.py -c ${saved_model_path}
                   ↪  -o ${output_path} ${filename}
54
55         done
56 done
```

A.7 Mega Score (Python)

Note: this script was stored in and executed from the xView Baseline/scoring folder.

```python
# script to run inference for like everything

# import scoring functionality
from score import score
import numpy as np
from tqdm import tqdm
# TODO: turn this into function

factors = ["1", "1.5", "2", "2.5", "3", "3.5", "4"]

base_path="/home/uceshhu"
trial="t9"
object_type="car"
prefix="car"

project_path="{}/runs/{}_ssd_inception".format(base_path, trial)
data_path="{}/xview_data".format(base_path)
personal_util_path="{}/personal_codebase/dissertation".format(base_path)
xview_util_path="{}/xview_codebase/baseline/scoring".format(base_path)
model_name="/ssd_inception"

models_research_path="{}/tf_api_codebase/models/research".format(base_path)
output_csv="{}/models{}/{}/{}_{}_performance.csv".format(project_path, model_name,
 ↪  object_type, prefix, trial)

# create for loop for each downsample factor
results=np.zeros([7*19, 6])
for i in tqdm(range(len(factors))):
        factor=factors[i]

        # ----- project specific definitions
        project_data_path="{}/data/{}/{}".format(project_path, object_type, factor)
        # run python script to process json
        groundtruth="{}/{}_{}_{}.geojson".format(project_data_path, prefix, trial,
         ↪  factor)

        saved_model_path="{}/models{}/{}/{}/saved_model_*/frozen_inference_graph.pb"
         ↪  .format(project_path, model_name, object_type, factor)
        # path to images
        test_image_path="{}/test_images_upsample_{}".format(data_path, factor)
        # folder to save .txt. files to
        detection_folder="{}/models{}/{}/{}/test_detections/".format(project_path,
         ↪  model_name, object_type, factor)
        output_folder="{}/models{}/{}/{}/test_evaluation".format(project_path,
         ↪  model_name, object_type, factor)

        # create for loop for each desired IOU
        for j in range(5,100,5):
                iou_threshold=j/100.0
                print "Calculating performance for downsample factor {} at IOU
                 ↪  {}".format(factor, iou_threshold)
                # run processing
                result=score(detection_folder, groundtruth, 'test.txt',
                 ↪  iou_threshold=iou_threshold)
                # append result to numpy array
                print "Adding results to row {}".format(i*19+((j/5-1)))
                results[i*19 + ((j/5)-1),]=[float(factor),iou_threshold,
                 ↪  result['map'],result['map_score'], result['mar_score'],
                 ↪  result['f1']]

# save results to specified CSV
np.savetxt(output_csv, results, delimiter=',')
```

A.8 Save PR (Python)

Note: this script was stored in and executed from the xView Baseline/scoring folder.

```
# script to run inference for like everything

# import scoring functionality
from score import score
from tqdm import tqdm
import numpy as np

# TODO: turn this into function

factors = ["1", "1.5", "2", "2.5", "3", "3.5", "4"]

base_path="/home/uceshhu"
trial="t9"
object_type="car"
prefix="car"

project_path="{}/runs/{}_ssd_inception".format(base_path,trial)
data_path="{}/xview_data".format(base_path)
personal_util_path="{}/personal_codebase/dissertation".format(base_path)
xview_util_path="{}/xview_codebase/baseline/scoring".format(base_path)
model_name="/ssd_inception"

models_research_path="{}/tf_api_codebase/models/research".format(base_path)
output_csv_path="{}/models{}/{}/pr/".format(project_path, model_name, object_type)


# create for loop for each downsample factor
for i in tqdm(range(len(factors))):
        factor=factors[i]

        # print "Processing results for downsample factor {}".format(factor)

        # ----- project specific definitions
        project_data_path="{}/data/{}/{}".format(project_path, object_type, factor)
        # run python script to process json
        groundtruth="{}/{}_{}_{}.geojson".format(project_data_path, prefix, trial,
        ↪  factor)
        saved_model_path="{}/models{}/{}/{}/saved_model_*/frozen_inference_
        ↪  graph.pb".format(project_path, model_name, object_type, factor)
        # path to images
        test_image_path="{}/test_images_{}".format(data_path, factor)
        # folder to save .txt. files to
        detection_folder="{}/models{}/{}/{}/test_detections/".format(project_path,
        ↪  model_name, object_type, factor)
        output_folder="{}/models{}/{}/{}/test_evaluation".format(project_path,
        ↪  model_name, object_type, factor)
                # run for one IOU only
        iou_threshold=0.5
        print "Calculating performance for downsample factor {} at IOU
        ↪  {}".format(factor, iou_threshold)
        # run processing
        result=score(detection_folder, groundtruth, 'test.txt',
        ↪  iou_threshold=iou_threshold)
        # append result to files

        # recall
```

```
51          # open file
52          r_handle = file(output_csv_path + "{}_{}_{}_recall.csv".format(object_type,
             ↪  trial, factor), 'w')
53          # write content & close
54          np.savetxt(r_handle, np.asarray(result['recall']),delimiter=',')
55          r_handle.close()
56
57          p_handle = file(output_csv_path +
             ↪  "{}_{}_{}_precision.csv".format(object_type, trial, factor), 'w')
58          # write content & close
59          np.savetxt(p_handle, np.asarray(result['precision']),delimiter=',')
60          p_handle.close()
```