Increased Cyclist Safety Using an Embedded System

Matthew Ryan Heydorn
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

BYU ScholarsArchive Citation
Heydorn, Matthew Ryan, "Increased Cyclist Safety Using an Embedded System" (2019). All Theses and Dissertations. 7391.
https://scholarsarchive.byu.edu/etd/7391

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen amatangelo@byu.edu.
Increased Cyclist Safety Using an Embedded System

Matthew Ryan Heydorn

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Dah Jye Lee, Chair
Ryan Farrell
James Archibald

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2019 Matthew Ryan Heydorn
All Rights Reserved
Increased Cyclist Safety Using an Embedded System

Matthew Ryan Heydorn
Department of Electrical and Computer Engineering, BYU
Master of Science

In order to reduce bicycle-vehicle collisions, we design and implement a cost effective embedded system to warn cyclists of approaching vehicles. The system uses an Odroid C2 single board computer (SBC) to do vehicle and lane detection in real time using only vision. The system warns cyclist are warned of approaching cars using both a smartphone app and an LED indicator.

Due to the limited performance of the Odroid C2 and other low power and low cost SBCs, we found that existing detection algorithms run either too slowly or do not have sufficient accuracy to be practical. Our solution to these limitations is to create a custom fully convolutional network (FCN) which is small enough to run at real time speeds on the Odroid C2 but robust enough to have decent accuracy. We show that this FCN runs significantly faster than Tiny YOLOv3 and MobileNetv2 while getting similar accuracy when all are trained on a limited dataset.

Since no dataset exists that separates the fronts of vehicles from other poses and is in the context of city and country roads, we create our own. Creating a dataset to train any detector has traditionally been time consuming. We present and implement a way to efficiently do this using minimal hand annotation by generating semi-synthetic images by cropping relatively few positive images into many background images. This creates a wider background class variance than would otherwise be possible.

Keywords: detection, real-time, machine learning, embedded
## TABLE OF CONTENTS

**LIST OF TABLES**  .................................................................  v

**LIST OF FIGURES** ............................................................... vi

Chapter 1  Introduction ....................................................... 1

Chapter 2  Background Information ........................................ 2
  2.1  Multilayer Perception ...................................................... 2
  2.1.1  Structure ................................................................. 2
  2.1.2  Loss ..................................................................... 3
  2.1.3  Training an MLP ....................................................... 4
  2.1.4  When to Use an MLP .................................................. 5
  2.2  Convolutional Neural Networks ........................................ 5
  2.2.1  2D Convolution ......................................................... 6
  2.2.2  Convolutional Layers .................................................. 7
  2.2.3  Max Pooling ............................................................... 7
  2.2.4  Fully Connected Layers ............................................... 8
  2.2.5  Data Flow Through a CNN ............................................ 8
  2.2.6  Training a CNN .......................................................... 8
  2.2.7  When to Use a CNN ..................................................... 9
  2.2.8  Single Shot Detection CNNs ......................................... 9
  2.3  Fully Convolutional Networks ......................................... 10
  2.3.1  Transpose Convolution ................................................. 10
  2.3.2  Transpose Convolution Layers ....................................... 11
  2.3.3  Data Flow Through an FCN ............................................ 11
  2.4  Haar Cascade Classifiers ............................................. 11
  2.4.1  Haar Features .......................................................... 11
  2.4.2  Adaboost ................................................................. 12
  2.4.3  Cascade Elimination .................................................... 13
  2.4.4  Multiple Scales ........................................................ 13
  2.4.5  Training a Cascade Classifier ..................................... 13
  2.5  Hough Line Transform .................................................. 13
  2.6  Precision and Recall ..................................................... 14
  2.6.1  Precision Recall Curves .............................................. 15
  2.7  Color Spaces ............................................................... 15
  2.7.1  RGB ................................................................. 15
  2.7.2  HLS ................................................................. 16

Chapter 3  Related Work .................................................... 18
  3.1  Cascade Classifiers ....................................................... 18
  3.2  R-CNN ................................................................. 18
  3.3  Fast R-CNN ............................................................... 19
LIST OF TABLES

6.1 Network Architecture ................................................. 33
6.2 Detector Resource Comparison ...................................... 35
8.1 Price Breakdown .............................................................. 41
## LIST OF FIGURES

2.1 MLP with two hidden layers and an output layer size of two. ........................................... 2
2.2 Offset from a bounding box prior. The red box is the offset and the black box is the prior. 9
2.3 Illustration of dilation performed as part of a transpose convolution .................................... 10
2.4 Examples of Haar features .................................................................................................. 12
2.5 Hough line transform. Each point in image space corresponds to a sinusoid in parameter space. The areas of highest intersection in parameter space define a line in image space. ................................................................. 14

4.1 100 Random images from our detection dataset ................................................................. 25

5.1 Detection from a Haar cascade classifier with the min number of neighbors set to 0 (left) and the same classifier with the min number of neighbors set to 2 (right). .............. 28
5.2 Cascade classifier performance on our validation task. Each color represents a stage count and scale factor combination. Multiple points are created for each classifier by adjusting the minimum neighbor threshold. ................................................................. 29
5.3 16 stage cascade classifier with and without a CNN ............................................................ 30

6.1 FCN architecture .................................................................................................................... 32
6.2 PR curve comparing the small FCN to MobileNetv2 0.35 224 and Tiny YOLOv3 on our detection task ................................................................................................................................. 34

7.1 Lane boundary marked in red. .............................................................................................. 38

8.1 Early prototype ...................................................................................................................... 39
8.2 Later prototype ..................................................................................................................... 40
8.3 Final prototype ...................................................................................................................... 40
8.4 Final prototype internal ........................................................................................................ 41
8.5 Smartphone app showing danger indicator ......................................................................... 42
CHAPTER 1. INTRODUCTION

Approximately 78% of fatal bicycle-vehicle collisions involve the cyclist being hit by the front of a vehicle [1]. We believe that in some cases these accidents could be deterred with the use of a device that detects the fronts of approaching vehicles and alerts the cyclist of danger. While such a device would be no replacement for standard safety equipment, its ability to constantly watch for approaching vehicles could make a difference.

The main technical challenge of creating such a device is detecting vehicles using hardware that is both low enough power to yield reasonable battery life and sufficiently cost effective to make the device financially practical. These hardware limitations present a challenge since deep learning models often have too many parameters for real-time inference speeds to be possible on a GPU-denied low power microprocessor. While forgoing deep learning in favor of a traditional sliding window detector is a valid solution, we show that a small deep learning model is capable of achieving both high frame rates and good accuracy on inexpensive hardware. We demonstrate this with a custom FCN running on an Odroid C2 single board computer.

Another technical challenge is that the creation and updating of a dataset required to train a detector is costly, due to the time needed to hand annotate images. We address this challenge by generating a semi-synthetic dataset using a small set of positive images and a set of background images extracted from un-annotated video.

We use these techniques to create and train an embedded system to warn cyclists of danger. Vehicle detection, lane detection, and danger evaluation are done on an Odroid C2 and danger indications are conveyed to the cyclist via a smartphone app and LED indicator.
CHAPTER 2. BACKGROUND INFORMATION

This section discusses some of the fundamental concepts of deep learning and computer vision that are needed to understand the design decisions made in this work.

2.1 Multilayer Perception

The multilayer perception (MLP) is generally considered to be the most basic class of neural network. An understanding of the MLP is important for an understanding of many other types of neural networks since they share many key concepts.

2.1.1 Structure

An MLP consists of an input layer, an output layer, and at least one hidden layer. All nodes in each layer are densely connected by weighted edges to nodes in the following layer. Figure 2.1 shows an MLP with 2 hidden layers and an output layer size of two. This MLP would be used to discriminate between two classes of data in which each observation is described by four data points.

Figure 2.1: MLP with two hidden layers and an output layer size of two.
The output of each node in the hidden layers is calculated by first taking the dot product of the activations of the nodes in the previous layer with the weights of the corresponding edges. The product is then added to a bias variable and passed through an activation function to get the output for each node. The sigmoid and ReLU functions are two of the most common activation functions. Equations 2.1 and 2.2 show the sigmoid and ReLU functions respectively.

\[ S(x) = \frac{1}{1 + e^{-x}} \] (2.1)

\[ ReLU(x) = \max(x, 0) \] (2.2)

The purpose of the activation functions is to introduce nonlinearity into the networks. Without the activation functions, the output layer would be a linear combination of the input layer and thus make a network incapable of learning a nonlinear discrimination function. This would not be ideal since many functions do not have a good approximation that can be expressed as a linear combination of the inputs.

A layer of an MLP without an activation function is known as a fully connected layer (FC layer). This is because all of the nodes from the previous layer are connected to all of the nodes of the current layer, making the nodes "fully connected" between layers. FC layers are relevant to many types of neural networks.

2.1.2 Loss

The loss of an MLP is a measure of the disparity between the network’s predictions and a set of ground truth values. Cross entropy loss, also known as log loss, is commonly used in MLPs. Cross entropy loss is the negative log of the difference between the ground truth and the network’s class probability prediction. Equation 2.3 defines cross entropy loss for binary classification.

\[
\text{loss}(y_p, y_t) = \begin{cases} 
-\log(y_p) & : y_t = 1 \\
-\log(1 - y_p) & : otherwise
\end{cases}
\] (2.3)
Cross entropy loss can be calculated for more than two classes by summing the negative log difference for each class, as shown in Equation 2.4.

\[
loss(y_p, y_t) = \sum_{c=0}^{M} y_{t,c} \log(y_{p,c})
\]  

(2.4)

where \( M \) is the number of classes. \( y_{t,c} \) is a binary indicator which is 1 if the observation is labeled as class \( c \) and 0 otherwise. \( y_{p,c} \) is the predicted probability that the observation belongs to class \( c \). \( y_{truth} \) is typically represented as an array of probabilities, where the index of each entry corresponds to the class number for the probability stored in the entry. In these cases, \( c \) would be the index into the array. \( y_{truth} \) is also typically an array of probabilities with a 1 in the index corresponding to the labeled correct class, and a 0 in every other entry. This is called one hot encoding.

In order to use cross entropy loss, the network must give a conditional probability for each class. The softmax function is used for this. Softmax takes a vector of real numbers and creates a normalized probability distribution by exponentiating each value in the vector then dividing by the sum of the exponentiated vector. This is shown in Equation 2.5, where \( \vec{z} \) is a vector containing the activations for each class.

\[
softmax(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}
\]  

(2.5)

### 2.1.3 Training an MLP

Training an MLP refers to finding weight and bias values which will lower the loss. The first step when training is to calculate the derivative of the loss function with respect to the weights. This allows an optimizer to see how the loss will change with a change to any of the weights and thus be able to update the weights intelligently. The process of calculating this derivative is called backpropagation because we calculate how the derivative propagates backwards from the output layer to the input layer. Because the calculation goes from the last layer to the first, backpropagation is often called a backward pass. Backpropagation is commonly used in deep learning.
Backpropagation uses the chain rule to calculate the derivative of the loss function with respect to the weights in a neural network. Thus for backpropagation to be possible, every part of the loss function must be differentiable. We won’t go into detail about how this is done in this work, but each element of the MLP that we have discussed is differentiable.

2.1.4 When to Use an MLP

The MLP as described in this work is useful in situations with the following properties. Note that the MLP will still work on data that is linearly separable but may be excessively complicated for the task.

1. Data is not linearly separable
2. Many annotated inputs are available
3. Annotations are categorical (though MLPs can be used with quantitative annotations when a different loss function is used)
4. Sufficient hardware resources are available

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a variation of the MLP which use discrete 2D convolution to encode an image into a latent space and then use a series of fully connected layers to make class predictions. CNNs can be thought of as having two parts; (1) the encode portion which uses 2D convolutions and (2) the classifier which uses FC layers to make predictions from the flattened output of the encode portion.

The same activations functions (ReLU and sigmoid) used in an MLP are used in CNNs. The activation functions are typically inserted between each convolutional layer and between each FC layer. The purpose of activation functions the CNN is the same as in the MLP, which is to add nonlinearity.
2.2.1 2D Convolution

CNNs use discrete 2D convolution to generate feature maps at each layer. A feature map (also called an activation map) is a matrix which describes how well a kernel fits at each point in an image or layer. A kernel (also called a filter) is a tiny image, usually only 2x2, 3x3, 5x5, or 7x7 in size. The values inside the kernel can be thought of as describing a feature. The process of generating a feature map from a kernel and input image is 2D convolution. Since images are stored using discrete numbers, the convolution is discrete.

To better understand 2D discrete convolution, we will compare it to simple 1D continuous convolution, which is shown in Equation 2.6.

\[ f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau \] (2.6)

This can be thought of as summing the product of \( f() \) and \( g() \) at every point as \( g() \) is slides through \( f() \). The discrete case makes this clear, as shown in Equation 2.7. The only difference between Equation 2.6 and Equation 2.7 is that the latter is discrete and thus uses summation rather then integration.

\[ f(x) * g(x) = \sum_{n=-\infty}^{\infty} f(n)g(x-n) \] (2.7)

Equation 2.7 can be extended to a second dimension when \( f() \) and \( g() \) take in two independent variables by extending the sum across the second dimension, as shown in Equation 2.8. Just as 1D convolution yields a 1D output, 2D convolution yields a 2D output.

\[ f(x,y) * g(x,y) = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f(n_1,n_2) \cdot g(x-n_1,y-n_2) \] (2.8)

In reality we can’t integrate from \(-\infty \) to \( \infty \) so we just integrate over the domain of \( f(x) \). In the case of an image, \( f() \) would be the image function and \( g() \) would be the kernel. Equation 2.9 shows the bounds where \( Img_x \) is the width of the image and \( Img_y \) is the height.

\[ f(x,y) * g(x,y) = \sum_{n_1=0}^{Img_x} \sum_{n_2=0}^{Img_y} f(n_1,n_2) \cdot g(x-n_1,y-n_2) \] (2.9)
Equation 2.9 is discrete 2D convolution and is equivalent to taking the dot product of the kernel \( g() \) and \( f() \) at every point in \( f() \). This creates the feature map of any given kernel and image combination. Equation 2.10 is a concrete example of calculating one activation pixel from a 2x2x1 kernel at location \( i, j \) in an image.

\[
h_{i,j} = W_{j,i}X_{j,i} + W_{j+1,i}X_{j+1,i} + W_{j,i+1}X_{j,i+1} + W_{j+1,i+1}X_{j+1,i+1},
\]

(2.10)

where \( h_{i,j} \) is the feature map pixel being calculated, \( W \) is the kernel, and \( X \) is the input image.

### 2.2.2 Convolutional Layers

A convolutional layer in a CNN takes in \( N \) stacked feature maps and uses \( K \) kernels to produce \( K \) stacked feature maps using Equation 2.9. \( K \), which is the number of kernels or filters, is often referred to as the depth of the network after a convolutional layer. The values of the kernel (also called weights) are learned.

A possible point of confusion is that the convolutional layer does 2D convolution even though its input and output are both 3D. While it is true that the dot product in Equation 2.9 is done between two 3D matrices, the convolution is still only 2D because the kernel only slides in 2 dimensions. This is possible because the kernel and input have the same depth. The output is 3D because it is a stack of 2D feature maps created from separate 2D convolutions. Each 2D convolution takes in the same input but has a different output because each has a different kernel.

### 2.2.3 Max Pooling

Max pooling is applied to a feature map by saving only the maximum activation value in a sliding window. This divides the resolution by the width or height of the sliding window. A 2x2 max pooling, which is commonly used, halves the resolution. Max pooling is applied to individual feature maps after a convolutional layer in order to reduce the size of the data being propagated.
2.2.4 Fully Connected Layers

Fully connected layers in CNNs have the same structure as the fully connected layers in MLPs. The difference is that there are typically fewer FC layers in a CNN since a lot of the actual data separation is done by the learned weights of the kernels in the convolutional layers. CNNs use FC layers more as a tool to interpret class predictions from the convolutional layers’ output rather then relying on them entirely to produce distinguishing features.

2.2.5 Data Flow Through a CNN

The image itself is used as input to the first convolutional layer. Several convolutional layers are then applied back to back with the output of each layer being used as input to the following layer. Max pooling and sigmoid/ReLU activation is done periodically between each convolutional layer. This has the effect of shrinking the input spatially but growing it depth wise.

The activation of the last convolutional layer (which will be deep but spatially coarse) is flattened into a 1D vector to form the input to a series of fully connected layers in order to get the final class predictions.

2.2.6 Training a CNN

Training a CNN is similar to training an MLP. Just as with an MLP, CNNs typically use cross entropy loss with softmax, which is possible because the final layers in a CNN have the same structure as those of an MLP. The difference is that the backpropagation must be performed on the convolutional layers in order to determine the derivative of the loss with respect to the weights of the kernels. This is necessary in order to update the kernel weights. For backpropagation to be possible, we must be able to formulate the derivative of the loss with respect to the weights of the kernels in a convolutional layer. Again, we will not go into detail about how backpropagation works but we will give an example of how the derivative is calculated for a single weight in a single layer of a CNN.
Equation 2.11 shows how the derivative is calculated for a single weight (top left) in a 2x2 kernel at position $i,j$.

$$
\partial W_{i,j} = X_{i,j} \partial h_{i,j} + X_{i+1,j} \partial h_{i+1,j} + X_{i,j+1} \partial h_{i,j+1} + X_{i+1,j+1} \partial h_{i+1,j+1},
$$

(2.11)

where $W_{i,j}$ is the kernel weight value at $i,j$, $X_{i,j}$ is the input feature map at index $i,j$, and $h_{j,i}$ is the input for the backward pass of the previous layer. Compare to Equation 2.10. This is a simple example to give an idea of how backpropagation works in a CNN. Further details are outside the scope of this work.

### 2.2.7 When to Use a CNN

CNNs are useful in situations in which the input data is spatially or temporally dependent. Image classification is the most common use case for CNNs since the meaning of each pixel depends on the pixels surrounding it (spatial dependence). Audio processing is another example of when a CNN may be useful since the data is temporally dependent. In the case of audio, 1D convolution would replace 2D convolution in the convolutional layers.

### 2.2.8 Single Shot Detection CNNs

Single shot detectors (SSDs) are an adaptation of traditional CNNs which can do bounding box prediction in addition to class prediction. This is done by interpreting the output of the fully connected layers as offsets from bounding box priors as well as class probabilities. The prior offsets usually include two translation parameters ($O_x - P_x$ and $O_y - P_y$) and two scale parameters ($\frac{O_w}{P_w}$ and $\frac{O_h}{P_h}$). Figure 2.2 illustrates a bounding box offset from a prior.

![Figure 2.2: Offset from a bounding box prior. The red box is the offset and the black box is the prior.](image-url)
The output size of the final fully connected layer can be quite large. It is the product of the number of priors, the number of predictions per prior, and the number of parameters per prior. A minimum of five parameters per prior are typically needed (four for translation/scale, and one for class).

2.3 Fully Convolutional Networks

Fully convolutional networks (FCNs) are a variation of CNNs which do segmentation rather than classification. FCNs replace the fully connected layers used in CNNs with transpose convolutions, thus having only convolutional layers. FCNs get their name from this fact. There are two parts to an FCN: (1) the encoder which uses convolutional layers to map the input into a latent space and (2) the decoder which uses transpose convolutions to map the latent image representation into the output space. The output space for an FCN is an image in which each channel represents the segmentation mask for a different class. The encoder part is the same as a CNN, but the decoder is different.

2.3.1 Transpose Convolution

A transpose convolution takes as input an image and outputs an image at a higher resolution. It is essentially learned up-sampling. Conceptually this is done by dilating the input image and performing a normal 2D convolution on it. The dilation (not to be confused with morphological dilation) is visualized in Figure 2.3. The spaces in between the original pixels in the dilated image are set to zero.

Figure 2.3: Illustration of dilation performed as part of a transpose convolution
2.3.2 Transpose Convolution Layers

The transpose convolution layer in an FCN takes as input $N$ stacked feature maps and uses $K$ kernels to produce $K$ stacked feature maps. Unlike standard convolutional layers, transpose convolutional layers usually have fewer kernels than the depth of the input ($K < N$). This is done in order to shrink the depth of the image. However, the resolution increases due to the dilation done in the transpose convolutions. The result is that transpose convolution layers decrease the depth but increase the resolution using learned weights.

2.3.3 Data Flow Through an FCN

Convolutional layers, max pooling, and activation functions are used to encode the input into a deep but spatially coarse latent space. Transpose convolution layers are then used to increase the resolution and lower the depth to get the image into the output space. The output space has a separate channel for the segmentation mask prediction of each class. The output space also typically has the same resolution as the input image.

2.4 Haar Cascade Classifiers

Haar cascade classifier works by using a series (or cascade) of filters described by Haar features in a sliding window to progressively eliminate regions of the image that do not contain the object.

2.4.1 Haar Features

Haar features are represented as rectangles that contain black and white rectangular sub regions. Haar features were inspired by the use of Haar wavelets. Haar wavelets are square shaped 1D waves that can represent an orthonormal basis which can be used to approximate any continuous real function. Thus it is intuitive that images should be describable by a linear combination of Haar features since they are essentially 2D Haar wavelets. A Haar basis can be thought of as a Fourier basis but with squares instead of sinusoids. Figure 2.4 shows examples of Haar features.
To calculate the descriptor for a Haar feature, the Haar rectangle is overlaid on a region of the image and the pixels of the image which are inside the white region of the Haar rectangle are subtracted from those inside the black region. The scalar difference is the descriptor.

2.4.2 Adaboost

Adaptive Boosting, or Adaboost, is used to select good Haar features for a cascade classifier. Adaboost works by selecting a combination, or ensemble, of weak classifiers to create a strong classifier. A weak classifier is one that does just barely better than guessing and a strong classifier is one that does much better than guessing. The weak classifiers are generated using synthesized HAAR features.

The advantage of using Adaboost to select features is that it selects only a small subset of the possible features. Since only the selected features must be evaluated at runtime, this allows for fast inference speeds. This approach is fundamentally different from neural networks, which must evaluate all parameters at runtime whether they are important or not. In short, Adaboost seeks to extract only the useful features while neural networks seek to change the features in order to make them all useful.
2.4.3 Cascade Elimination

Each stage of a cascade classifier contains a combination of Haar features. These HAAR features are used to calculate a descriptor at a fixed array of points in the image using a strided sliding window. These descriptors indicate the presence or absence of the object at each region of the image. The locations whose descriptors indicate that they do not contain an object are eliminated and are not searched by later stages so as to not waste time calculating their descriptions. The regions which make it through all of the stages of the classifier are the output of the classifier.

2.4.4 Multiple Scales

Since Haar features are not scale invariant, they cannot inherently detect objects of multiple sizes. A solution to this problem is to create scaled copies of the image (called a scale space) and to calculate Haar descriptions across the entire scale space. While this does give Haar cascade classifiers some scale invariance, it is also costly since the amount of computation is multiplied by the depth of the scale space.

2.4.5 Training a Cascade Classifier

In order to train a cascade classifier, you need a set of images that contain the object of interest (called positives) as well as a set of images that do not contain the object of interest (called negatives). Ideally the positives contain just the image and little else. OpenCV contains an implementation of AdaBoost that can be used to generate the filters from a set of positives and negatives, which is used to train the cascade classifiers in this work.

2.5 Hough Line Transform

The Hough line transformation works on a binarized image by mapping each positive pixel to a sinusoidal curve in a 2D parameter space with r and \( \theta \) as the two axes. The regions of highest intersection in parameter space define detected lines in image space. This r-theta Hough line transform technique was introduced in [2].
In order to perform the Hough line transform, a set of intersecting lines are drawn through each positive pixel in image space. For each of these lines a new line is drawn from the origin to the closest point on the original line. The length \((r)\) and angle from the origin \((\theta)\) of each of these new lines is plotted as a point in parameter space. This process is repeated for each point, yielding a discrete sinusoidal curve in parameter space for each point in image space, as shown in Figure 2.5. The regions of the highest overlap in parameter space give us the \(r\) and \(\theta\) corresponding to the line in image space. This line is drawn and a second line orthogonal to it which intersects the first line’s non origin endpoint is drawn. The second line is the detected line in image space.

![Figure 2.5: Hough line transform. Each point in image space corresponds to a sinusoid in parameter space. The areas of highest intersection in parameter space define a line in image space.](image)

### 2.6 Precision and Recall

Predicted or real occurrences of an object are classified as being one of three types. True negatives (no object existing and no object detected) are sometimes used as well, but not in this work.

1. False Positive (Detector found object when there was no object)
2. False Negative (Detector missed the object)
3. True Positive (Detector found an object and there was an object)
The quantity of these three types of occurrences is used to determine the precision and recall of a detector. Precision quantifies how accurate the predictions of a detector are. Recall quantifies how likely a detector is to detect an object. Equations 2.12 and 2.13 show how recall and precision are calculated respectively where TP is the quantity of true positives, FN the quantity of false negatives, and FP the quantity of false positives.

\[
\text{Recall} = \frac{TP}{TP + FN} \tag{2.12}
\]

\[
\text{Precision} = \frac{TP}{TP + FP} \tag{2.13}
\]

2.6.1 Precision Recall Curves

In a probabilistic detector a threshold must be set in order to determine if an occurrence counts as a detection. As this threshold is increased the precision will increase, since only the occurrences that the detector is more confident about will be counted as detections, but the recall will drop since the detector will be more likely to miss detections with the higher threshold.

A PR curve can be created by spanning the threshold values and plotting the precision and recall of the detector at each threshold. The PR curves gives a visualization of the tradeoffs a detector must make between precision and recall. A higher area under the PR curve indicates a better detector. An ideal detector would have an area under the curve of 1.0. See Figure 6.2 for an example of a PR curve.

2.7 Color Spaces

2.7.1 RGB

RGB is the simplest color space. Each pixel is represented by three scalars: one for each primary additive color (red, green, and blue). This color space is common since the pixels in color TV screens and computer monitors have red, green, and blue components and therefore an image signal is easily conveyed with red, green, and blue intensity values.
2.7.2 HLS

HLS (hue, luminance, saturation) is a color space where each pixel is represented by three scalars which correspond to the hue, luminance, and saturation of each pixel. Hue is the color. Luminance is the brightness of the color. Saturation is how easily the color can be separated from other colors. HLS is conceptually similar to HSV (hue, saturation, value) but is calculated slightly differently. Equations 2.14 through 2.19 show how the HLS value of a pixel is calculated from normalized RGB values.

\[ C_{\text{max}} = \max(R, G, B) \]  \hspace{1cm} (2.14)

\[ C_{\text{min}} = \min(R, G, B) \]  \hspace{1cm} (2.15)

\[ \Delta = C_{\text{max}} - C_{\text{min}} \]  \hspace{1cm} (2.16)

\[ H = \begin{cases} 
60 \times \left( \frac{G-B}{\Delta} \mod 6 \right) : & C_{\text{max}} = R \\
60 \times \left( \frac{B-R}{\Delta} + 2 \right) : & C_{\text{max}} = G \\
60 \times \left( \frac{R-G}{\Delta} + 4 \right) : & C_{\text{max}} = B 
\end{cases} \]  \hspace{1cm} (2.17)

\[ L = \frac{C_{\text{max}} + C_{\text{min}}}{2} \]  \hspace{1cm} (2.18)

\[ S = \begin{cases} 
0 : & L = 0 \\
\frac{\Delta}{1 - |2L-1|} : & \text{otherwise}, 
\end{cases} \]  \hspace{1cm} (2.19)

where R, G, and B are the normalized red, green, and blue values of the pixel respectively. H is the hue in degrees ranging from 0 to 360. L is the luminance ranging from 0 to 1. S is the saturation ranging from 0 to 1. The final H, L, and S values are typically normalized to be between 0 and 255 for 8 bit encoding.
The advantage to the HLS color space over RGB is that it more easily allows for invariance to luminance by specifying a range of luminance directly. The same range would correspond to many ranges of RGB values which would be more complicated to compute.
CHAPTER 3. RELATED WORK

Cascade classifiers are common place for real time object detection on embedded devices. Deep neural networks (DNNs) have also started gaining popularity in recent years. Such networks include R-CNN, Fast-RCNN, Faster-RCNN, YOLO, SSD, and MobileNet. We also discuss work related to lane detection.

3.1 Cascade Classifiers

Cascade classifiers have been used for real-time object detection on systems with limited resources ever since they were introduced in 2001 [3]. We were unable to get good inference speeds while also getting good accuracy with a cascade classifier.

3.2 R-CNN

Region-Convolutional Neural Network (R-CNN) was one of the first CNNs to do object detection. It achieved this by first running a selective search algorithm to split the image into regions. A classification CNN was then run on each of these regions to extract features. These features were passed to an SVM which learned to predict a class for each region [4].

R-CNN’s design was reminiscent of traditional detection pipelines which used hand-crafted features extracted at multiple regions in the image in order to classify each region. The main difference with R-CNN was that it used a pretrained CNN to extract these features.

R-CNN is too slow to run in real-time on embedded device because it has to run the entire feature extraction CNN over many regions, which is costly. Even with GPU acceleration, R-CNN takes about 47 seconds to produce predictions on a single test image [4].
3.3 Fast R-CNN

Fast R-CNN runs faster than R-CNN and has comparable results. Fast R-CNN is faster than R-CNN because it only uses a full CNN to extract features once per image [5]. Selective search is still used to produce regions, but the features describing each region are reused from the initial pass of the base CNN rather than being regenerated by the base CNN per region as with R-CNN.

Another difference between Fast R-CNN and R-CNN is that Fast R-CNN uses a fully connected layer to predict a class for each region rather than the SVM used in R-CNN. Fast-RCNN runs at about 0.2 fps with GPU acceleration [5], which is still too slow for most embedded applications.

3.4 Faster R-CNN

Faster R-CNN does away with R-CNN’s and Fast R-CNN’s selective search approach for region proposals and replaces it with a separate region proposal network (RPN) [6].

The RPN works in three steps. First, it uses a pretrained CNN to extract convolutional features from the input image. Second, it applies a sliding window to the convolutional features. For each spatial location of the sliding window, 9 anchors (predefined bounding boxes) are generated. Each of these 9 anchors share the same spatial location, but have different scales and aspect ratios. The intersection over union (IOU) is then calculated for each anchor by dividing the size of the overlapping region by the total size of the anchor and prediction. Third, the anchors and their corresponding convolutional features are fed into another network which predicts a final bounding box for each window location as well as a confidence of objectness. Since the RPN used in Faster R-CNN is faster than selective search for region proposals, Faster R-CNN runs at about 0.2 fps with GPU acceleration [6].

3.5 SSD

Single Shot Detector (SSD) is one of the first network architectures to do object detection without external RPN. This fact enables SSD to predict images with a single pass through a single network, which is why its called single shot detector.
SSD works by pairing a traditional CNN with a multibox detection algorithm which predicts size and position offsets from bounding box priors. The main difference between this approach and an RPN is that in this approach the features used to predict the bounding boxes are the same features used to do classification. This enables the optimizer to learn a correlation between bounding box shape and object class. It also means that the network can be trained end to end [7].

SSD boosts its sensitivity to small features by predicting bounding boxes and classes across a scale space [7]. The scale space is created automatically by the pooling layers of the base network that SSD uses, so this doesn’t slow SSD down significantly.

### 3.6 YOLO

YOLO is another single shot detector which works similar to SSD. The main difference is that YOLO concatenates features across various scales to generate a single set of bounding boxes rather than producing many bounding box and class predictions at different scales. Original YOLO also predicted bounding boxes directly using FC layers without priors. [8].

YOLO9000 improves on YOLO by adding priors (which they call anchors), a class hierarchy system which allows for classification of 9000 classes, and other changes such as batch normalization layers and the ability to run at multiple resolutions [9]. YOLOV3 is the latest iteration of YOLO which makes minor tweaks to improve accuracy without affecting the runtime significantly [10]. We use only Tiny YOLOV3 (a smaller version of YOLOV3) for our benchmark comparison.

### 3.7 Mobilenet

Mobilenet uses depth-wise separable convolutions to create a classification CNN with fewer parameters. This enables Mobilenet to have real time classification speeds on Google Pixel smartphones [11]. Mobilenet can also do object detection when paired with an object detector such as the multibox algorithm used in SSD [11].

MobilenetV2 improves on Mobilenet by introducing, which replaces all regular convolutions in the detector with separable convolutions. This enables MobilenetV2 to achieve about 20 percent faster inference speeds over Mobilenet [12]. We use MobilenetV2 for our benchmarks.
3.8 Fully Convolutional Networks

FCNs have been shown to be effective tools for learned segmentation in [13]. One reason FCNs are not typically used for detection tasks is that the ground truth required to train an FCN needs to be a per pixel segmentation map for each class. These segmentation maps are more costly to create than simple bounding boxes. As a result, networks that train on bounding boxes are generally more practical for tasks that only require detection. For our application we circumvent this problem by generating masks automatically while creating a dataset.

3.9 Lane Detection

Works such as [14] and [15] demonstrate that the Hough transform is useful for lane detection. We found that a basic hough line transform with color segmentation and a Canny edge binarization work well for our application.
CHAPTER 4. CREATING A DATASET AND BENCHMARK TASK

Before experimenting we needed a dataset of images which distinguished the fronts of cars from all other views. As no such dataset existed to our knowledge, we created our own. The process we used to do this is described in this section.

4.1 Train and Test Dataset

4.1.1 Gathering Images

We collected images of stationary vehicles in parking lots using a bicycle equipped with two cameras - one pointed at 90 degrees from the front of the bicycle and the other pointed at 45 degrees. Each camera recorded video as the bicycle was ridden up and down parking lot aisles.

4.1.2 Processing Positives

A pretrained deep neural network was run on a full resolution copy of each of the video frames we captured with the action cameras and the bounding box locations were stored. The bounding boxes were used to crop out the cars from each image at their original size. We could then throw away all images whose largest bounding box was not within a specified X threshold. This eliminated most images of the sides of cars.

Next we used the correlation distance between the histograms of the car samples to eliminate similar samples. Our goal was to keep only the first frame and the last frame chronologically for each car. To do this we started by iterating through the frames and measuring the similarity between each sample and its predecessor. Whenever the similarity was greater than a fixed threshold, both the sample and its predecessor would be marked to be saved. We then deleted all images which were not marked to be saved.
We found that the ideal correlation distance and bounding box X threshold were slightly different for each video. To find the best threshold, we created a program which would show the user which samples would be saved and which were eliminated given a threshold. The user could then adjust the threshold such that only two images at the desired orientation of each car were marked to be saved.

The next step was to separate the images of the fronts of cars from the images of the backs of cars. To do this we first manually selected about 100 images of the fronts of cars which we used to train a cascade classifier. This classifier then evaluated all of the car samples to separate the fronts from the backs. We then manually selected 100 images which were incorrectly classified (both front and back) and added these to our set of positive and negative images respectively and retrained the classifier. We repeated this process through 9 iterations, at which point the classifier did an adequate job of separating the front images from the back images. We used this same classifier for all future videos.

The final step was to manually remove low quality or erroneous samples. We found that about 1 in 3 of the samples that made it through the process up to this point were removed manually. This process was significantly easier than manually selecting images from every frame since only about 1 in 100 images made it this far. This process enables us to extract images taken at consistent angles of a variety of cars with little manual input.

4.1.3 Adding Negatives

For the negatives or background images, we extracted frames from footage of road bike races. We choose this footage because it is widely available on the internet, rarely has fronts of cars in it, and contains relevant background. We extracted one frame from every ten seconds of video to avoid having too many frames of the same background.

4.1.4 Combining Positives and Negatives

At this point we had a collection of positive and negative images. This worked well for training cascade classifiers but was not suitable for training detectors which require localization data.
Our first attempt to fix this problem was to use the video frames that the samples in the dataset were extracted from. This did not work well. The main problem was that the locations of the cars were similar in every image. This caused detection algorithms to learn to always label that region of the image as positive, regardless of the content of the image. Another problem with this approach was that the background was always a parking lot, which did not represent our use case.

We made several attempts to just create a new dataset with cars and their backgrounds. Our first attempt to make such a dataset was to use footage taken from a stationary camera along the side of the road. This did not work well since the background was static and the cars were never directly facing the camera.

Our next attempt was to put a camera on the back of a bicycle and ride it around on the streets. This turned out to be impractical since relatively few cars overtake a cyclist in the same lane that the cyclist is using. The exception is during heavy traffic when there is not enough room for motorists to change lanes, but in such cases the traffic would be moving slowly such that few cars would overtake the cyclist and consequently few useful images could be extracted. Furthermore, the cars that did overtake the cyclist in the same lane were never cars that were on a trajectory to hit the cyclist. We consider this a good thing, but it also made this approach an impractical way to create the dataset we needed.

Our solution was to create a synthetic dataset using the images that we had already collected. We created a program to generate the synthetic images by cropping a randomly chosen positive sample into a randomly chosen background image. The positive image was randomly rotated in a range of -15 to 15 degrees and randomly resized to be in a range between 0.10 to 0.45 the size of the background image. Next, the corners of the positive samples were cut to approximate the shape of a car. Lastly the border between the background and the positive sample was Gaussian blurred.

We choose this approach for three reasons. First, it enabled us to create a new dataset without spending time collecting additional images. Second, it made it possible to generate many background/positive combinations since background images were plentiful. Third, it enabled us to more directly compare the performance of detectors that require localization data to those that don’t by making a dataset that can do both.
The disadvantage to this approach is that the context of the positives is often unrealistic. We see this in Figure 4.1, which shows 100 randomly chosen images from the dataset. Many of the images contain cars in the clouds and in the trees. This makes any detector trained on this dataset context independent. While a context independent detector may be more versatile, it will probably not perform as well.

![Figure 4.1: 100 Random images from our detection dataset](image)

### 4.2 Validation Dataset and Benchmark

We created a validation task to quantify each detector’s performance. The results of this task were unknown to us during training.
4.2.1 Validation Dataset

The images we used for the task consist of 600 hand chosen frames taken from a video feed of a camera on the back of a bicycle on streets in Provo. The locations of all cars in each video frame were detected using a pretrained DNN and then each detection was marked as either a threat or non-threat manually. We deemed threat cars as those which were directly behind the cyclist, pointing towards the cyclist, and within about 20 feet of lateral distance from the cyclist. All other cars were marked as non-threats.

4.2.2 Validation Benchmark

We created a benchmark task to rate each detector’s performance using the images we annotated. For each annotated image, we ran the detector and compared its predictions to the annotated car locations. True positives were calculated by adding up the number of times the detector predicted a car close to an annotated threat car. A maximum of one true positive could be accumulated per annotated threat car. False negatives were calculated in a similar fashion - each annotated threat car which did not have a prediction from the detector within a static pixel distance was counted as a false negative. False positives were calculated by adding up the number of predictions which where far from any annotated car. We used these statistics to calculate precision and recall.

We measure precision and recall this way for two reasons. First, it made the results of the task more resilient to inconsistency in the annotator’s idea of which cars are a threat and which are not. To illustrate this, imagine we had only annotated threat cars. This would mean that the cars detected which were just outside being considered a threat by the human annotator would have been marked as false positives while those just inside the annotator’s idea of a threat car would be marked as true positives. This means that any variation in the annotator’s decision boundary would cause any given detection to change between a false positive and a false negative, which would cause the recall to vary widely due to the annotator’s likely inconsistent decision boundary due to human nature. This problem is mitigated by simply ignoring cars just outside the annotators idea of a threat car instead of counting them as false positives.
The second advantage is that it gives us a metric which better measures what we care about. If we had taken the other option and annotated all cars as threat cars then this would have meant that a detector’s prediction of a threatening car (one we care a lot about) would have counted the same as the detection of the side of a car passing in a far lane (one we care less about). Doing so would have yielded a task that measures a detector’s ability to find non-threat cars since the majority of the cars in the images were non-threat cars. This problem is avoided in our approach by ignoring the detection of non-threat cars.
CHAPTER 5. EXPERIMENTS WITH CASCADE CLASSIFIERS

Due to their speed and simplicity, cascade classifiers seemed like a good starting point for this project. In the end we found that cascade classifiers are useful, but we were unable to get the needed accuracy while maintaining sufficient prediction speeds. This section details our findings.

5.1 Traditional Cascade Classifier

We trained several cascade classifiers and benchmarked them on a subset of our validation task. The results for the entire validation tasks were not calculated until after the training process was finished so as to avoid indirectly overfitting the detectors to the validation dataset by our design decisions when creating the cascade classifiers. We discovered that cascade classifiers struggle to reliably detect cars without also producing many false positives. Figure 5.1 demonstrates a typical failure case.

Figure 5.1: Detection from a Haar cascade classifier with the min number of neighbors set to 0 (left) and the same classifier with the min number of neighbors set to 2 (right).

Figure 5.1 shows that the classifier with the minimum neighbor threshold set to 0 yields many false positives and that the same classifier with the threshold set to 2 yields fewer false positives, but the car is missed. This problem was common to all classifiers we created. This is likely due to the wide variance of vehicles.
We created cascade classifiers with various scale densities, min number of neighbors thresholds, and number of stages. Figure 5.2 shows how a few of these classifiers perform on our validation task.

Figure 5.2: Cascade classifier performance on our validation task. Each color represents a stage count and scale factor combination. Multiple points are created for each classifier by adjusting the minimum neighbor threshold.

There are various trade-offs made when creating a cascade classifier. The classifiers with a lower scale factor perform better and can reach a higher recall than the classifiers with a higher scale factor because a lower scale factor yields a denser scale space which can handle changes in object size better. The downside is that a denser scale space requires more computation at runtime.

Another trade-off is between maximum recall and precision, which is effected by the number of stages. Figure 5.2 shows that the 21 stage classifiers have higher precision at low recalls compared to the 16 stage classifiers, but have a lower maximum recall. This is because the 21 stage classifiers tend to produce fewer predictions since each region must pass through 5 more stages than the 16 stage classifiers, but the regions that do make it through the extra 5 stages are more likely to be accurate. We trained classifiers with as many as 32 stages and found a similar trend.
5.2 Cascade CNN Hybrid

We created a CNN to classify each detection from the cascade classifiers as either a car or background in an effort to reduce the false positive rate. All of the regions proposed by the cascade classifier at the final stage were resized and run through this CNN. The regions classified as background were eliminated.

The CNN we created for this task is small. It only has two convolutional layers and two fully connected layers, with 32 filters in the final convolutional layer. We choose this architecture over a deeper one so that it could be run multiple times per image in a reasonable amount of time. The CNN was trained using the same positives and negatives that the cascade classifiers were trained with. The positives were first randomly rotated between 0 and 14 degrees and randomly flipped across the Y axis. Subsections of the negatives were randomly cropped, resized, and fed to the CNN as well for the background class.

Figure 5.3: 16 stage cascade classifier with and without a CNN
Figure 5.3 shows that the CNN does not significantly improve accuracy. We experimented with CNNs between 1 and 4 layers with a maximum depth between 32 and 256 filters and got similar results. The smallest 32 depth CNN required about 0.1 seconds per detection, meaning that there was about 0.2 seconds of latency introduced assuming two detections were made per frame on average.

The hybrid approach has four main drawbacks. The first is that it is slower than a cascade classifier since it requires an addition to the cascade classifier. The second drawback is that the runtime is unpredictable. The cascade classifier outputs any number of predictions for any given image, meaning that the CNN has to run an unpredictable number of times. Since running the CNN is costly, images on which the cascade classifier produces many regions require more time than images on which only a few regions are produced by the cascade classifier. This is particularly undesirable since the images which produce more predictions from the cascade classifier are more likely to contain a car and are thus more urgently needed. Since cascade classifiers do not give probabilistic results, we couldn’t simply test the top \( n \) predictions. The third drawback to the hybrid approach is that its recall can only be as good as the cascade classifier's recall. This can be seen in Figure 5.3. The maximum recall for the hybrid detector is just over 0.8 while the cascade classifier can reach nearly 1.0. The fourth drawback is that this method is not trained end to end since the cascade classifier and the CNN are trained separately.
CHAPTER 6. FULLY CONVOLUTIONAL NETWORK FOR DETECTION

An FCN solves three of the drawbacks of the cascade-CNN hybrid method. First, an FCN only runs once per image with a fixed number of parameters and therefore has a predictable run-time. Second, an FCN’s recall is not limited by an external region proposal system. Third, an FCN can be trained end to end.

6.1 Network Architecture

We created a custom FCN with the objective of getting a minimum of 5 fps on the Odroid C2 while sacrificing as little accuracy as possible. We experimented with many layer counts, filter counts, layer resolutions, and kernel sizes and found a good compromise in the form of an FCN architecture consisting of 5 convolutional layers and 5 transpose convolution layers. All convolutional kernels are 3x3 and all transpose convolution kernels are 4x4. ReLU activation is used after each convolutional layer. Table 6.1 details the architecture.

![FCN Architecture Diagram](image)

Figure 6.1: FCN architecture
Table 6.1: Network Architecture

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>Resolution (After)</th>
<th>Filters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1</td>
<td>Convolution</td>
<td>224x224</td>
<td>8</td>
</tr>
<tr>
<td>Pool1</td>
<td>Max Pool</td>
<td>112x112</td>
<td>-</td>
</tr>
<tr>
<td>Conv2</td>
<td>Convolution</td>
<td>112x112</td>
<td>16</td>
</tr>
<tr>
<td>Pool2</td>
<td>Max Pool</td>
<td>56x56</td>
<td>-</td>
</tr>
<tr>
<td>Conv3</td>
<td>Convolution</td>
<td>56x56</td>
<td>32</td>
</tr>
<tr>
<td>Pool3</td>
<td>Max Pool</td>
<td>28x28</td>
<td>-</td>
</tr>
<tr>
<td>Conv4</td>
<td>Convolution</td>
<td>28x28</td>
<td>64</td>
</tr>
<tr>
<td>Pool4</td>
<td>Max Pool</td>
<td>14x14</td>
<td>-</td>
</tr>
<tr>
<td>Conv5</td>
<td>Convolution</td>
<td>14x14</td>
<td>128</td>
</tr>
<tr>
<td>Pool5</td>
<td>Max Pool</td>
<td>7x7</td>
<td>-</td>
</tr>
<tr>
<td>TConv1</td>
<td>Transpose Convolution</td>
<td>14x14</td>
<td>64</td>
</tr>
<tr>
<td>TConv2</td>
<td>Transpose Convolution</td>
<td>28x28</td>
<td>32</td>
</tr>
<tr>
<td>TConv3</td>
<td>Transpose Convolution</td>
<td>56x56</td>
<td>16</td>
</tr>
<tr>
<td>TConv4</td>
<td>Transpose Convolution</td>
<td>112x112</td>
<td>8</td>
</tr>
<tr>
<td>TConv5</td>
<td>Transpose Convolution</td>
<td>224x224</td>
<td>2</td>
</tr>
</tbody>
</table>

In order to increase sensitivity to small objects, we implemented fuse connections as done in [13] by summing the activations from TConv4 to Pool1, TConv3 to Pool2, TConv2 to Pool3, and TConv1 to Pool4.

Figure 6.1 illustrates our FCN architecture. The convolutional layers encode the image into a deep but spatially coarse latent space. The transpose convolution layers then decode the latent image representation into the output space in which the vehicle segmentation prediction is in a separate channel from the background segmentation prediction. We have two channels in this space, one for background and one for vehicles.
6.2 Comparison with Other Detectors

6.2.1 Accuracy Comparison

We trained the FCN, Tiny YOLOv3, MobileNetv2, and a 16 stage cascade classifier on our detection dataset and benchmarked each on the task of detecting fronts of vehicles in 600 real world images. Bounding boxes were created for the FCN by doing blob detection on the softmax of the FCN’s output activations, with the average softmax value in each blob being used as the confidence score and the diameter of each blob being used as the width and height. Figure 6.2 compares the PR curve of each detector.

![PR Curve - Deep Networks](image)

Figure 6.2: PR curve comparing the small FCN to MobileNetv2 0.35 224 and Tiny YOLOv3 on our detection task
Figure 6.2 show that the FCN performs similarly to MobileNetv2 and Tiny YOLOv3 at the point right before precision starts dropping quickly, which is the best threshold to run the detectors at in our use case. While these results do not suggest that the FCN is universally better, they do suggest that the FCN’s accuracy is in the vicinity of mainstream detectors for our use case.

The results on the complete validation set were not calculated until after the networks were done training so as to avoid overfitting. We used a subset of 100 images as a test set. These images were taken from a separate video from the rest of the validation images. The benchmark results on these 100 images were available to us when training each network.

6.2.2 Resource Comparison

We compared the inference time per image, number of parameters, and model sizes of these detectors. The results are shown in Table 6.2. All inference times were measured on the same Odroid C2 running the same OS. We verified that all approaches utilized all four of the C2’s cores. The FCN runs significantly faster than all other detectors we benchmarked. It also has fewer parameters. The FCN’s model size is comparable, though the model size comparison here is not conclusive since not all of the networks’ weight files use the same format.

<table>
<thead>
<tr>
<th>Network</th>
<th>Parameters</th>
<th>Inference Time</th>
<th>Model Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD 300</td>
<td>26.29 M</td>
<td>6.01 sec</td>
<td>105.1 MB</td>
</tr>
<tr>
<td>MobileNetv2 1.0</td>
<td>5.6 M</td>
<td>1.81 sec</td>
<td>14.1 MB</td>
</tr>
<tr>
<td>MobileNetv2 0.35</td>
<td>3.8 M</td>
<td>1.12 sec</td>
<td>6.8 MB</td>
</tr>
<tr>
<td>Cascade 16 Layer</td>
<td>N/A</td>
<td>1.20 sec</td>
<td>2.0 MB</td>
</tr>
<tr>
<td>FCN</td>
<td>2.1 M</td>
<td>0.17 sec</td>
<td>15.9 MB</td>
</tr>
<tr>
<td>Tiny YOLOv3</td>
<td>6.41 M</td>
<td>11.51 sec</td>
<td>63.5 MB</td>
</tr>
</tbody>
</table>
We calculated the total number of parameters in the convolutional stages of each network by summing up the number of filters and multiplying by the filter size. The total number of trainable parameters, \( P() \), is defined below.

\[
P(S(), D_{in}(l_n), D_{out}(l_n), L) = \sum_{l_n=0}^{L} S(l_n)D_{in}(l_n)D_{out}(l_n),
\]

(6.1)

where \( L \) is the number of layers, \( S(l_n) \) is the size of the spatial dimensions of the filters at layer \( l_n \), \( D_{in}(l_n) \) is the number channels of the input to layer \( l_n \), and \( D_{out}(l_n) \) is the number of channels of the output from layer \( l_n \). Since Mobilenet uses depth-wise separable filters, equation 6.1 does not apply. Fortunately, Google lists the total number of parameters for each type of Mobilenet, which we used for Table 6.2 [11].

6.3 Discussion of Results

The primary advantage of the FCN is that it runs faster than other approaches while getting similar accuracy for the task of detecting fronts of cars when trained with a limited dataset. The faster runtime over traditional SSDs is due in part because the FCN has no fully connected layers. Fully connected layers can be costly because their nodes are densely connected, meaning the number of weight variables is equal to the input size times the output size. This tends to yield many weight variables in all but very shallow cases. Many weight variables is undesirable since each weight variable must be stored in memory and calculated at runtime, hurting memory usage and inference speed.

One disadvantage to the FCN is that you need segmentation masks to train, though this work shows that it is sufficient to synthesize these masks at least in the case of cars. This approach would likely not be as effective on objects with greater concavity such that a significant amount of the background behind the sample is included in the bounding boxes.

Another disadvantage of the FCN is that it has no notion of separate instances of objects. Figure 6.2 suggests that blob detection does a sufficient job at separating instances for our use case, but blob detection would be less effective with objects that tend to overlap. This also has the side effect of limiting the maximum recall of the FCN, which is evident in Figure 6.2 by the fact that the blue line does not make it all the way to 1.0 on the x axis.
We do lane detection to better assess the threat of detected cars by classifying roads as either having a dedicated bike lane or not. Dedicated bike lanes are detected by the presence of a white line to the left of the cyclist. When such a white lane marker is detected, cars which are detected to the left (towards the center of the road) of the lane marker are allowed to get slightly closer to the cyclist before being considered a threat. The absence of such a lane marker causes the algorithm to warn of cars at a slightly longer distance from the cyclist. We do this with the assumption that roads with no dedicated cycling lane are generally more dangerous and therefore approaching cars should be treated as greater threats.

### 7.1 Finding Lanes

Lane detection is done by first converting the image into the HSL (Hue, Saturation and Luminance) color space using Equations 2.14 to 2.19. The HSL image is used to create yellow and white masks. The yellow and white masks are unioned together to get a single mask which is then intersected with the original HSL image to get an image with the yellow and white pixels preserved and all others set to 0. The image is then binarized using a Canny edge detector and lastly a Hough lines is run on the binarized image to get the lane locations.

### 7.2 Binning Lanes

In practice we found it best to detect many Hough lines, bin them by their start and end coordinates, and only draw one line defined by the average start and end locations of the lines in the bin. Our approach to this is basic. When a line is detected which is outside of a fixed threshold from any other bin, a new bin is created and the line is inserted. When a line is detected that is within a threshold of an existing bin, we add that line to the bin. The average lines of each bin are the final lines we use. Figure 7.1 shows a lane boundary detected by this algorithm.
Our binning approach is similar to KMeans. It differs in being that the bin centroids are not updated when new lines are added and there is only one iteration. While KMeans would be a more accurate way of binning the Hough lines, we found that the process of detecting the Hough lines would fail before our binning approach was a problem. Since KMeans would require more complexity to update the bin centroids after each iteration, we do not use it.
CHAPTER 8. IMPLEMENTATION DETAILS

8.1 Hardware

We use an Odroid C2 as the vision processor. The C2 has 2GB of RAM, a 1.5 GHz 64-bit quad-core ARM Cortex-A53 processor, and a 10 watt max power draw. We choose the C2 over other single board computers because of its combination of low power consumption, low cost, good thermals, and respectable performance compared to competitors. The low power consumption of the C2 was especially important because we found that it required a considerably more expensive voltage regulator to get more than 2A at 5V from a Li-ion battery. The C2 is the only board of the three which requires 2A or less at 5V.

We designed and 3D printed a case for the C2 with an integrated camera, connector for an external led indicator, 9.7Wh Li-ion battery, and charge controller. The internal battery was only included in the final revisions, as the earlier designs relied on an external USB power pack. Below are a few of the most significant design iterations in chronological order.

![Figure 8.1: Early prototype](image)
Figure 8.1 shows one of the first working prototypes. It required an external battery, the USB cord for the camera ran along the outside, and it used a rubber strap to mount to the bicycle. The USB battery is mounted to the chain stay of the bicycle Figure 8.1.

![Figure 8.1: First prototype](image1)

Figure 8.2: Later prototype

Figure 8.2 shows a later iteration. The rubber strap was replaced with a GoPro style mount, the case is about half as thick, and it uses a landscape orientation in order to work with shorter bicycle seat posts.

![Figure 8.2: Later prototype](image2)

Figure 8.3: Final prototype

Figure 8.3 shows the most recent iteration. The wiring for the camera is internal and a Li-ion battery was added. The case is slightly thicker than the case in Figure 8.3 due to the battery and charge controller, but is still considerably thinner than the case in Figure 8.1.

![Figure 8.3: Final prototype](image3)
Figure 8.4 shows the inside of the final version. The battery, charge controller, and camera sensor board are visible. The Odroid C2 sits behind these components. The 9.7Wh battery yields about 1 hour of battery life.

Table 8.1: Price Breakdown

<table>
<thead>
<tr>
<th>Part</th>
<th>Cost</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charge Board</td>
<td>$9</td>
<td>Contains regulators for 3.7-5.0V and 5.0V-3.7V</td>
</tr>
<tr>
<td>ABS Plastic</td>
<td>$3</td>
<td>3D printer maintenance costs not included</td>
</tr>
<tr>
<td>Li-ion Battery</td>
<td>$12</td>
<td>Replacement battery for LG L90 smartphone</td>
</tr>
<tr>
<td>Webcam</td>
<td>$10</td>
<td>720p output, USB 2.0</td>
</tr>
<tr>
<td>Odroid C2</td>
<td>$63</td>
<td></td>
</tr>
<tr>
<td>Bluetooth Dongle</td>
<td>$7</td>
<td>Bluetooth 4.0</td>
</tr>
<tr>
<td>Misc</td>
<td>$2</td>
<td>4.5mm jack for LED indicator and on/off switch</td>
</tr>
</tbody>
</table>
Table 8.1 shows how much each part costs. The total cost was $106. All parts were purchased at standard consumer pricing.

### 8.2 Danger Score Calculation

Each detection is assigned a danger score based on its location and size. Equation 8.1 shows how the score is calculated. A score of 1.0 means maximum danger and a score of 0.0 means minimum danger.

\[
Score = \frac{B_H \times B_W}{Img_A} \times (1 - \frac{|B_X - Img_W/2|}{Img_W/2})
\]  

(8.1)

Where \(B_H\) is the height of the bounding box, \(B_W\) is the width of the bounding box, \(B_X\) is the mean \(x\) value of the bounding box, \(Img_A\) is the pixel area of the frame, and \(Img_W\) is the width of the frame. If more than one bounding box is detected in a frame, the score is calculated for each box and only the maximum score is reported. The score is 0 when no bounding boxes are present. If a bicycle lane is detected and any part of the bounding box lies inside the lane boundary then the score is increased by 25% or to 1.0, whichever is lower. If the bounding box is entirely outside of the lane boundary then the score is reduced by 25%.

### 8.3 Smartphone App

We created a smartphone app which connects to the Odroid C2 via Bluetooth. The app indicates the absence or presence of a high danger vehicle using a green or red box, as shown in Figure 8.5.

Figure 8.5: Smartphone app showing danger indicator
The app works by receiving Bluetooth serial messages sent by the Odroid C2 after each frame is processed. Each message contains the danger score for that frame. The app has a timer which, when expired, will turn the indicator red and display a message saying that there is a connection error. The timer is reset every time the device sends a message with a danger score.
CHAPTER 9. CONCLUSION

9.1 Summary

We created a device that detects cars to warn cyclist of danger. Since no adequate detection algorithm existed for this particular use case, we experimented with several and discovered that a small FCN can perform comparably to mainstream object detectors and at faster speeds with the restrictions of our use case. We also showed that such an FCN can be trained on semi synthesized images in the case of cars when positive samples are available. We introduced a way to create such a dataset and demonstrated its effectiveness by creating a dataset of the fronts of cars.

9.2 Future Work

The dataset we created can be expanded using the method described in this work. With a larger dataset, it is possible that there will be more of an accuracy difference between the FCN and other detectors since it is possible that the size of the dataset is limiting accuracy rather then the robustness of the models.

Comparing performance on models trained with our synthesized dataset to the same models trained on non-synthesized datasets with similar images would make it possible to measure the loss in accuracy due to the semi-synthetic nature of our dataset. Although creating such a dataset would be difficult for the reasons explained in this work, it would likely perform better.

Further experimentation with different depths and resolutions in the FCN architecture could lead to a better FCN. While we did experiment with a range of depths and resolutions, our search was not exhaustive. In addition to tweaking the design of the FCN, creating a small detector similar to multibox and pairing it with a small CNN to create a fast detector is a similar approach which would be interesting to compare to the FCN.
The hardware of the embedded system presented here can be enhanced in several ways. Audio functionality for the app can be added. The device itself can be made smaller with a custom single board computer in which the full sized USB, HDMI, and Ethernet ports are removed to save space. The heatsink for the CPU could be integrated into the design of the case, making room for a larger battery. Lastly, a bright LED could be added to the back of the device to warn motorists.
REFERENCES


