Pedestrian Assistant: Computer Vision to Help the Visually Impaired

STEM THESIS

Darshan Krishnaswamy

Table of Contents

Acknowledgements

I would like to acknowledge Mass Academy teachers Mrs. Taricco, Dr. Crowthers, and

Mrs. Cochran, Mass Academy seniors Grant Perkins and Lucas Lanzendorf, and Computer

Vision research engineer David Mascharka, for their support with this project.

Abstract

Visually impaired pedestrians are generally more susceptible to accidents at street intersections because they lack the ability to see oncoming cars and other obstructions. A study conducted by the American Council for the Blind showed that 8 percent of visually impaired pedestrians reported being hit by a car and 28 percent reported that their canes had been run over. Pedestrian intersections often have audible cues to assist visually impaired pedestrians; however, these are not always effective because many pedestrian intersections, especially in rural areas, do not contain audible cues. Additionally, many pedestrians have difficulty memorizing the different sounds that lights facing different directions play, which may cause them to cross at an unsafe time. Because of this, a more effective system is needed to help visually impaired pedestrians know when it is safe to cross the street.

For this project, an image classification model and an object detection model were created to locate and classify pedestrian traffic lights in real-time in order to help visually impaired pedestrians know whether it is safe to cross the street. The classification model used a convolutional neural network and had a final accuracy of approximately 81.82 percent. The object detection model was created by retraining an existing object detection model called SSD Mobilenet, and ended with a final precision of about 81.8 percent, a final recall of about 90.0 percent, and a final F1 accuracy of about 85.7 percent. These models have the potential to greatly reduce accident rates for visually impaired pedestrians.

Literature Review

Introduction

Visually impaired pedestrians are generally at a high risk for getting into accidents because they lack the ability to see cars or other obstructions in front of them. Between 2007 and 2016, total pedestrian deaths increased by 27 percent. In 2017, the estimated number of pedestrian fatalities was close to 6,000 (Retting, 2017). In comparison, there were closer to 4,000 such deaths in 2009, with 24 percent of these accidents occurring at intersections (Pharr, Coughenhour, & Bungum, 2013). These statistics demonstrate that although precautions are in place to protect pedestrians, they are still ineffective at preventing many deaths. Such accidents are especially common among blind and visually impaired pedestrians. In a survey conducted on visually impaired pedestrians by the American Council of the Blind, 8 percent of respondents reported that they had been hit by a car at an intersection, and 28 percent of respondents reported that their canes had been run over (Al-Fuqaha, Oh, & Kwigizile, 2018). Visually impaired pedestrians have such a high rate of accidents due to their decreased ability to detect pedestrian traffic lights and obstructions that may be in the way, such as people, poles, and street signs. Currently, audible signals are used at some pedestrian traffic lights to assist the visually impaired; however, this system comes with many problems and inconsistencies. A study done in 2012 demonstrated that blind pedestrians performed significantly worse than the normally sighted, even with access to auditory information (Hassan & Massof, 2012).

Pedestrian Crossings

Two common types of pedestrian crossings in the United States are the zebra crossing and the signal intersection. The zebra crossing consists of striped lines on a street where pedestrians may safely cross. The second type of crossing is the type that involves a special

traffic signal designated for pedestrians, which are referred to as signal intersections. Pedestrians may cross zebra crossings at any time, as they always have the right of way at these crossings, and cars are required to yield to anyone waiting to cross at these intersections. At signal intersections, pedestrians only have the right of way when their light is green. These types of crossings are generally safer than zebra crossings, since the pedestrians only have a green light when cars have a red light. At zebra crossings, however, cars may still be traveling while the pedestrian is trying to cross, which can be dangerous (Federal Highway Administration, 1999).

Jaywalking

One especially common cause of pedestrian injuries is jaywalking. Jaywalking refers to any time that a pedestrian walks into the path of a motor vehicle when he or she does not have the right of way. Jaywalking can occur either at signal intersections, when the pedestrian light is red or at other points along the road that are not designated pedestrian crossings. Jaywalking is the cause of about 6,000 driving accidents every year (Team, 2016).

Blind pedestrians often jaywalk on accident, since they may be unaware that they are at an intersection or that the pedestrian traffic light is red. Because of this, many states and towns have White Cane Laws, which require all cars to yield their right of way to any blind pedestrian with either a white cane or a guide dog (Murphy, 1965). Although these laws help to protect blind pedestrians who are unaware of their surroundings, the system is not perfect, and, as mentioned earlier, there are still cases every year where blind people are hit by moving cars while the blind pedestrians are supposed to have the right of way (Al-Fuqaha, Oh, & Kwigizile, 2018).

Pedestrian Traffic Lights

Most common in large cities, some intersections contain special traffic lights for

pedestrians. These are mainly used in areas where drivers may find it difficult to see pedestrians,

as the drivers do not need to actively search for pedestrians and stop for them. Instead,

pedestrians can cross at designated times and are required to wait during other times. Unlike

normal traffic lights, pedestrian traffic lights can vary between different locations. The standard

pedestrian traffic light displays a solid red hand when it is not safe to cross, a walking person

when it is safe to cross, and a blinking red hand when it is not safe to begin crossing, but safe to

continue crossing if the person has already started walking. However, some pedestrian lights

appear differently. For example, some display a red person standing when it is not safe to cross,

and a green person walking when it is safe to cross. Others display the words "WALK" in white

and "DONT WALK" in red when it is safe and not safe to cross, respectively. **Figure 1** displays

the different types of pedestrian traffic lights.

Figure 1. The three common types of pedestrian traffic light. Retrieved from ("New York 'Walk /

Don't Walk' Sign", n,d,), (Vani, 2017), ("300mm Green Walk Man Pedestrian Crossing Light

For Road Safety", n.d.)

Audible Pedestrian Signals

Currently, the most common way for pedestrian traffic signals to be accessible for pedestrians is the use of audible signals that play during the WALK interval. They generally play a chirping sound, similar to the sound that birds make, and they have distinct sounds for North-South and East-West intervals, which allow visually impaired pedestrians to know which direction has the green light (Szeto, Valerio, & Novak, 1991). Not all audible pedestrian signals play the same sound. Along with the chirping sound, which is commonly referred to as a "cuckoo" sound, audible signals may also play sounds such as a "rapid tick" sound, where a ticking noise is played multiple times per second. Another possible sound is a noise referred to as a "peep-peep noise." These audible signals are not legally required at pedestrian intersections. Instead, they are often added as requested by residents of the city. In the city of Columbus, for example, only about 10.5 percent of pedestrian crossings with traffic signals have audible crosswalk signals (Kistler, 2015).

While audible signals generally help visually impaired pedestrians know when it is safe to cross the street, the system is not perfect. At times, pedestrians may mistake the sound of a real bird chirping for the audible pedestrian signal and think it is safe to cross. The different tones that play at different intersections also often confuse pedestrians. Pedestrians also often have trouble remembering which noise corresponded with each direction, and also sometimes did not know which direction they were facing (Kistler, 2015).

Machine Learning

One possible way to resolve the issue of allowing signals to be accessible to visually impaired people is through a system that uses machine learning to assist the user. Machine learning is a way of creating a model that is able to learn from data. There are three major types

of machine learning: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the model is given data and corresponding "labels," which are the desired output. The model is trained to take in data and return the correct output. In unsupervised learning, there are no labels; instead, the model looks for patterns in the data that it is given and learns how to sort the data. Reinforcement learning is mainly used in robotics and game development. It works by giving the model a "reward" for a successful action and the model learns how to maximize its reward (Thrun, 1992). A potential solution to the pedestrian traffic light problem for visually impaired pedestrians would be to use supervised learning to create a model that would be able to detect pedestrian traffic lights and alert the user of the light's status (whether it is red or green).

Neural Networks

In order to classify the different types of pedestrian traffic lights, a neural network must be created to perform the classification. A neural network is a machine learning model. Neural networks are comprised of layers, each connected to the previous layer. The first layer of a neural network is the input layer, which contains the original data. Each value in the next layer is produced by taking each value in the previous layer, multiplying each value by a distinct number, known as a weight, adding up all the resultant values, and adding another term known as a bias. The values of this layer are then passed to the next layer, and the process continues throughout all of the layers until the final layer, known as the output layer.

Layers are comprised of neurons, each holding a single number. The number of neurons in a layer generally decreases as one approaches the output layer. There are many possibilities for the number of neurons in the output layer. If the model's purpose is to predict a value, for example, the output layer could have a single neuron containing its prediction (Lapedes &

Farber, 1988). If the model's purpose is to classify an image between 10 categories, it could

have 10 neurons in the final layer, each containing the probability that it is an image of each

category. **Figure 2** illustrates how the values in one layer of a neural network correspond to the
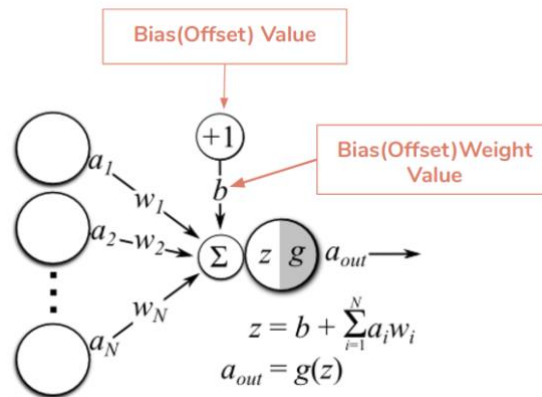
values in the next layer.

Figure 2. A visual representation of how a neural network layer works. Retrieved from

("Everything you need to know about Neural Networks", 2017).

Activation and Loss Functions

       There are two important functions used by neural networks: the activation function and

the loss function. The activation function is applied to each neuron, and it can serve many

different purposes. One major purpose of the activation function is to create an upper and lower

bound for the neurons' values. Activation functions such as the sigmoid and hyperbolic tangent

functions, two common activation functions which are based on the exponential function, have

fixed output ranges, which prevents the neurons' values from getting too large or too

small. Another purpose of an activation function is to eliminate the linearity of the model.

Without an activation function, each layer would be calculated as a linear function of the

previous layer, so the entire neural network would just be a linear function of the input

layer. Activation functions, which are usually nonlinear functions, prevent this and increase the

complexity of the model, allowing it to reach a higher accuracy than it would without an activation function (Sharma, 2017). The loss function represents the neural network's error. There are many different loss functions. One is called mean-squared error, in which the difference between the values in the output layer are subtracted from the truth values, those values are squared, and the average of the resultant values is calculated. Other loss functions include softmax cross-entropy, which is commonly used in classification, and mean-absolute error, which is similar to mean-squared error, but instead of squaring the values, the absolute value is taken (Parmar, 2018). The loss function is used in gradient descent, a process used to optimize the parameters which will be discussed further in the next section.

Gradient Descent

Once the traffic light neural network has been initialized with the appropriate number of layers and size of each layer, the model must be trained in order to have an accurate output. When a neural network is initialized, all of its parameters (weights and biases) are randomly initialized. The neural network then needs to go through a training process, in which these parameters are tuned so that they create accurate outputs. This process of training a neural network is made possible through what is known as gradient descent. The process involves finding the gradient vector of the loss function, and then adjusting all of the parameters by subtracting the gradient multiplied by another parameter known as the learning rate, which must be manually inputted before training. The gradient vector points in the direction of greatest ascent (increase) for the loss function, so the negative gradient points in the direction of steepest descent. Thus, by subtracting the gradient multiplied by some value, the model decreases the value of the loss function, making the model more accurate (Pandey, 2019). Eventually, the loss function will taper off at some value, at which the accuracy of the model is maximized, and can

only be affected by tuning other parameters such as the number of layers, the size of each layer, and the amount of data used to train the model. The python libraries Tensorflow, PyTorch, and Keras are commonly used for machine learning projects because they have optimized gradient calculation abilities, which are useful when training neural network models.

Convolutional Neural Networks (CNNs)

The solution to the pedestrian traffic light situation using neural networks is through image classification, the process of classifying an image as a member of a certain category. For example, in the pedestrian traffic light case, the categories would be red light and green light. Convolutional neural networks (CNNs) are a specific type of neural network that are commonly used for image processing and classification. The input image is a matrix with three dimensions: one dimension for the width of the image, one dimension for the height of the image, and one dimension for the image's colors (this dimension has a size of three, one for each of the colors red, green, and blue, which are known as the color channels). This matrix can be imagined as three two-dimensional matrices stacked on top of each other, one for each color channel. CNNs have two special types of layers: convolutional layers and max pooling layers. In convolutional layers, the CNN uses a window, known as a kernel, which is a two-dimensional matrix. This matrix "slides" over the image matrix, one color channel at a time, and each value of the image matrix is multiplied by the value in the kernel to obtain the layer's output value. In a max pooling layer, a new kernel slides over the image. This kernel contains no values; as it slides over the matrix, the maximum value covered by the kernel replaces the covered area (the output from this layer is usually smaller than the input). This process repeats multiple times, and the output is a classification layer, which generally contains one neuron for each category that the neural network is classifying between. For example, if a neural network was classifying images

between dogs, televisions, and shoes, there would be three neurons in the final layer, each containing the probability that the input is an image of each respective category. The predicted category is the category with the highest probability from the output layer (Ciresan, Meier, Masci, Gambardella, & Schmidhuber, 2011).

CNN Activation and Loss Functions

An important step in the process of creating the image classification model is choosing appropriate activation and loss functions. The most commonly used activation and loss function for CNNs are softmax and cross-entropy, respectively. Softmax is a function which takes in a set of values for each category and returns the probability that the image should be classified as each category. The cross-entropy function is a loss function that is very similar to mean absolute or mean squared error, in that it also calculates the distance between the output value and the expected value. It utilizes logarithms, which allows it to pair well with the softmax function, which uses exponential functions. Because of this, the two functions are frequently used together in convolutional neural networks (Liu, Wen, Yu, & Yang, 2016)

Datasets and Partitioning

Neural networks "learn" how to classify between different categories by looking at data and finding patterns that allow it to differentiate between the categories. In image classification problems, an example of supervised learning, the data is accompanied by corresponding "truth values," which represent the labels for each image. For the pedestrian light neural network, this dataset will contain images of different types of pedestrian traffic lights, with labels containing the status (red or green) of each light.

As neural networks are trained on a set of data, they will eventually learn to find patterns that are specific to the data that they are trained on, which will cause it to have high accuracy in

classifying the data that it has already seen, but low accuracy on data that it has not yet seen, a phenomenon known as "overfitting." To prevent this, the dataset is often partitioned into two sets: the train set and the validation set. The train set is the data that the model sees and is trained on, while the validation set is used to obtain an unbiased value of the accuracy of the model and to prevent overfitting the model (Moore, 2001). This is especially important in the pedestrian light model, as it will be classifying images that it has never seen before, so it needs to have high accuracy in predicting the status of images that it has not been trained on.

Tensorflow and Android Implementation

The portability of cellular devices allows them to be a convenient way for users to run machine learning projects, such as the pedestrian traffic light neural network. Applications can be created for Android devices using the Android Studio Development Environment, a free software created by Google. Android applications are created using the Java and Extensible Markup Language (XML) languages. While many other Android application development programs exist, Android Studio is the official development environment for Android devices and is the most commonly used due to its versatility and convenient features such as built-in device emulators (Technoaddict, 2018).

A common library used in machine learning projects is Tensorflow, an open-source deep learning library created by Google which can be used with both the Python and C++ programming languages. The library also has a framework called Tensorflow Lite, which allows for deployment on mobile devices, including both Android and IOS devices (Tensorflow Lite, 2019), which makes it a useful software to use for creating the pedestrian traffic light neural network.

A common problem faced in deploying a neural network on mobile devices is that the neural networks are unable to run quickly or efficiently enough on the device due to the low Central Processing Unit (CPU) power. When this occurs, the models are often "quantized," meaning that the precision of the numbers used in the neural network is decreased (Jacob, Klygis, Chen, Zhu, Tang, Howard, & Kalenichenko, 2018). This slightly decreases the accuracy of the model, as the parameter values are not as exact; however, it can lead to large boosts in CPU usage and memory, greatly increasing the model's efficiency.

Conclusion

Although there have been many attempts to make pedestrian traffic lights accessible for the visually impaired, the current systems are still ineffective at fully ensuring the safety of the pedestrians. Thousands of accidents occur in which pedestrians are killed each year, and a large percentage of blind pedestrians have been involved in accidents (Al-Fuqaha, Oh, & Kwigizile, 2018). Because of this, there is a clear need for a more effective solution to this problem, and the answer may lie in using machine learning techniques to ensure the safety of these visually impaired pedestrians.

A convolutional neural network that is able to accurately detect pedestrian traffic lights and detect their statuses could greatly assist visually impaired pedestrians, as they will receive direct input about whether or not it is safe to cross, rather than be forced to rely on audible signals, which often confuse the pedestrians (Kistler, 2015). The model would need to have high accuracy on the validation sets, meaning that it would need to be trained on a large dataset of images and tested on diverse images in order to ensure its accuracy with data that it has not previously been exposed to. A potential effective implementation of the neural network would be on cellular devices, as a large percentage of people carry these around wherever they go. This

would allow for a convenient and easy to use method for visually impaired pedestrians to utilize

the neural network.

Introduction

Visually impaired pedestrians often get into accidents while crossing the street, because they are not able to see if there are any oncoming cars. In a survey conducted on visually impaired pedestrians by the American Council of the Blind, 8 percent of the respondents reported that they had been hit by a car at an intersection, and 28 percent of the respondents reported that their canes had been run over (Al-Fuqaha, Oh, & Kwigizile, 2018). A study conducted in 2012 demonstrated that blind and visually impaired pedestrians performed significantly worse in judging whether it was safe to cross than the normally sighted, even with access to auditory information (Hassan & Massof, 2012).

Pedestrian injury has been increasing at a growing rate in recent years. Between 2007 and 2016, total pedestrian deaths increased by 27 percent. In 2017, the estimated number of pedestrian fatalities was close to 6,000 (Retting, 2017). In comparison, there were closer to 4,000 such deaths in 2009 (Pharr, Coughenhour, & Bungum, 2013). This shows that there is an increase in the number of pedestrian injuries and fatalities over the years.

Many attempts have been made to assist visually impaired pedestrians, but for the most part, they have not been very successful. One such attempt has been the implementation of auditory signals at pedestrian traffic lights to alert pedestrians when it is safe to cross (Kistler, 2015). These audible signals are usually similar to a "chirping" sound or a "cuckoo" sound. However, this is not an effective overall solution. One issue is that these signals are not required by law; many cities only implement them at citizens' request. As an example, in the city of Columbus, Ohio, only about 10.5 percent of pedestrian traffic lights have audible signals (Kistler, 2015). Another common issue with these audible signals is that the sounds can vary between different cities. While some cities use the "chirping" sound during green lights, other cities may use "cuckoo" sounds, "rapid tick" sounds, or "peep-peep" noises. Because there is no

standard noise for pedestrian traffic lights, pedestrians may get confused when they hear a different sound than the one that they are used to hearing. In addition, they sometimes mistake birds chirping for the sound of the pedestrian light, which may cause them to attempt to cross the street at an unsafe time (Kistler, 2015).

This project attempts to assist pedestrians and to help them know when it is safe to cross the street using machine learning and neural networks. A live object detection model was created to detect pedestrian traffic lights in real-time and alert the pedestrian if there are any red or green traffic lights ahead. This would resolve the problem of different sounds for different traffic lights because this would always alert the pedestrian in the same way. It would also solve the problem of not all traffic lights having signals to help the visually impaired, because the object detection model would work at all traffic lights, regardless of whether or not they were created with any extra features.

Methods and Materials

Materials used in this project included a MacBook Pro Laptop and a Samsung Galaxy S9

cell phone. Software used in the project included the Python 3.7 language

(https://www.python.org/), the TensorFlow machine learning library

(https://www.tensorflow.org/), the OpenCV python library (https://opencv.org/), the NumPy

python library (https://numpy.org/), the ElemertTree Python library

(https://docs.python.org/2/library/xml.etree.elementtree.html), the image labeling service

LabelImg (https://github.com/tzutalin/labelImg), and the Android Studio

(https://developer.android.com/studio) environment for mobile application development.

The preliminary step in the creation of the neural networks was obtaining a dataset. For

this, a Google Images API (https://github.com/fastai/course-v3/blob/master/nbs/dl1/lesson2-

download.ipynb) was used. A JavaScript command was run in the console while Google Images

was opened to automatically download the images into a folder. This was done for images of

both green traffic lights and red traffic lights. The data was then filtered to remove any images

that did not contain either type of pedestrian traffic light. The final sizes of each dataset were 128

images of red traffic lights and 149 images of green traffic lights.

Next, the data was used to train an image classification neural network. The machine

learning tensor library TensorFlow 2.0 was used for training the model. The images were loaded

into the python file using the OpenCV library and were converted into 4-dimensional arrays of

integers using the NumPy library. TensorFlow was used to create a 6-layer Convolutional Neural

Network with 3 convolutional layers and 3 dense layers, with a 2-dimensional output (containing

the probabilities that the image contained a green traffic light and a red traffic light,

respectively). The two datasets were combined into a single dataset containing 277 images, and

this was split at random into two datasets: a training dataset with 200 images and a validation

dataset with 77 images. The model was trained over 50 epochs of the data using the softmax

activation function and cross-entropy loss.

After the image classification model was created, the model was converted to an object

detection model. The object detection model was created using the TensorFlow object detection

library by retraining the SSD MobileNet model. First, the set of images was annotated with

rectangles notating where the objects (traffic lights) were located within the image. The images

in the previous dataset had been shrunken to the size of 100 pixels by 100 pixels, but for object

detection, larger images were required to produce an accurate model. Additionally, images that

did not contain a pedestrian traffic light could be used in an object detection model. As a result,

the image dataset was redownloaded using the google images API (with about 400 images in

each of the two datasets) and was resized so that each image was 300 pixels by 300 pixels. The

images were annotated with boxes denoting the locations of the objects within each image using

the image-labeling service LabelImg, and this data was parsed and converted into python NumPy

arrays using Python's XML ElementTree library.

These images were again loaded into a Python file using the OpenCV and NumPy

libraries to be used in training an object detection model. The object detection model was trained

using the TensorFlow Object Detection Library. The model was trained over 5,000 epochs of the

training dataset, which had a size of about 600 images.

The object detection model and the image classification model were then implemented

for Android mobile devices to allow for easier access to the neural network by users. The

classification model and object detection models were implemented using TensorFlow Lite, a

library which allows TensorFlow neural networks to be implemented for mobile applications.

Results

For the image classification model, the final accuracy and loss values were determined using TensorFlow's built-in accuracy and loss calculator. The model was trained over 50 epochs of the train data. The model's final train accuracy was 96.00 percent with a cross-entropy loss value of 0.0770 and a validation accuracy of 74.03 percent. However, the optimal state of the model (where the validation accuracy was the highest) was located after the 22_{nd} epoch, where its accuracy was 89.00 percent, its cross-entropy loss value was 0.2061, and its validation accuracy was 81.82 percent. Because of this, the final model was reverted back to its state when it had completed 22 epochs. **Figure 1** shows a graph of the train loss, train accuracy, and validation accuracy over the first 22 epochs.

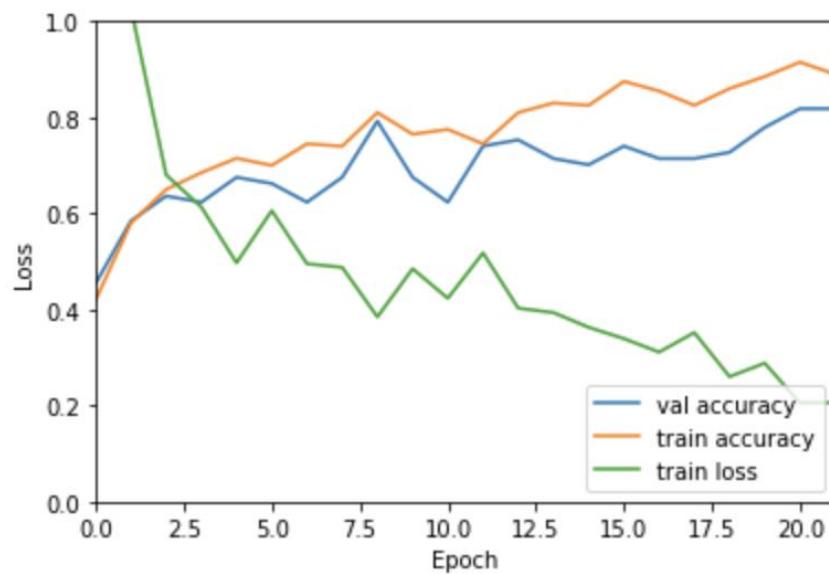*Figure 3: A graph of the train loss, train accuracy, and validation accuracy of the model over the first 22 epochs. Graph created using the Python Library MatPlotLib and data created by the Python tensor library TensorFlow.*

For the object detection model, the TensorFlow service TensorBoard was used to find the loss value of the model. The model began with a loss value of about 15.5 and finished with a

value of about 2.5. The model was trained over 5,000 epochs of the training data, which

contained 603 annotated images. Based on a manual validation of the model on the test dataset,

the model finished with a final precision value of about 81.81 percent, a final recall value of

about 90.00 percent, and a final F1 accuracy value of about 85.70 percent. **Figure 2** shows a

graph of the train loss of the model over all 5,000 epochs.

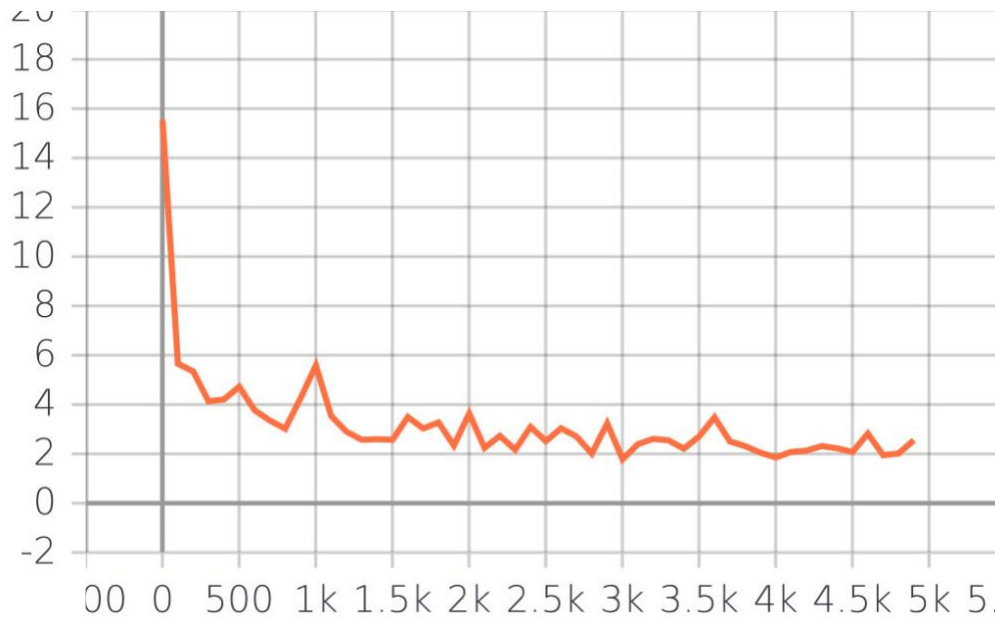
Figure 4: A graph of the train loss of the object detection model over 5000 epochs. The y-axis

represents the train loss of the model and the x-axis represents the epoch number. This graph

was created by the TensorFlow service Tensorboard

Discussion/Conclusion

The image classification and object detection neural networks were both successfully able to learn how to classify and detect the different types of pedestrian traffic lights. The image classification neural network finished with an accuracy of about 82 percent, after starting with an accuracy of about 42 percent. This high accuracy demonstrates that the model was successfully able to learn the differences between green and red pedestrian traffic lights. Although this accuracy is not as high as that of other similar object detection models, which are able to reach accuracies of close to 99 or 100 percent, these other models often require much higher computing powers and must run on a portable personal computer rather than on a mobile device (Cheng et al., 2017).

While the model was able to take an image of a pedestrian traffic light as an input and output the state of the traffic light, the model did not do well with classifying images where the traffic lights were small or distant. Rather, it worked most effectively with images with large or nearby pedestrian traffic lights, but these do not accurately represent what a pedestrian would see at an intersection. Additionally, the model always outputted a value of "green" or "red," even if there was no traffic light in the image. Because of this, the image classification model would not suffice for helping visually impaired pedestrians and the object detection model was needed to accurately locate and classify pedestrian traffic lights in real-time.

The object detection model was manually evaluated to obtain the precision, recall, and F1 values, which are three common metrics in evaluating object detection models. The precision value was found by calculating $\dfrac{c}{c + f}$, where c represents the number of correctly predicted bounding boxes, while f represents the number of false positives (boxes predicted when there was no pedestrian traffic light in the image). This was found to be about 81.82 percent. The

recall value was found by calculating $\frac{c}{c + i}$, where c represents the number of correctly

predicted bounding boxes and i represents the number of traffic lights that were not detected by

the model. This was calculated to be about 90.00 percent. The F1 accuracy value was calculated

by finding $\frac{2 * p * r}{p + r}$, where p represents the precision of the model and r represents the recall of

the model. This was found to be about 85.70 percent.

      The high recall value of the object detection model shows that when there is a traffic light

in the image, the model is successfully able to locate and classify it about 90 percent of the time.

The precision value of the model is slightly lower, which means that the model occasionally

identifies and classifies pedestrian traffic lights in images when they do not exist. The F1 value

creates a balance between the two values, providing an accuracy number that takes into account

both false positives and missed traffic lights.

      While the values for the object detection seem similar to those of the image classification

model, the object detection model was much more successful at accurately locating and

predicting the state of pedestrian traffic lights. This is because the classification model was only

able to classify the entire image as one entity, and was not able to determine whether or not the

image contained a traffic light, while the object detection model was able to determine whether

or not an image contained a traffic light as well as classify each traffic light in the image.

      There are multiple different potential causes of error in this project that limited the

maximum accuracy that the model was able to reach. One possible source of error was

inconsistency in the annotated images used for training. Because the images had to be manually

annotated and labeled by having boxes drawn around the pedestrian traffic lights, there were

likely inconsistencies in which areas of the traffic lights were included within the box. Another

likely source of error was a limited dataset size, which prevented the model from fully being able to account for the variation between different images of pedestrian traffic lights (Solegaonkar, 2019).

There are many possible extensions to this project. One possible extension would be to improve the runtime of the model, because on average, the model took a little under one second per image to classify the image, draw the bounding box, and display the image. While this would likely be sufficient to help visually impaired pedestrians, making it detect the traffic lights more quickly could help improve its real-time detection accuracy. Another possible extension would be to create a more cost-effective device on which the model can be implemented. This would allow it to be accessible to all visually impaired pedestrians. One more possible extension would be to improve the precision and recall values for the model, to reduce the model's susceptibility to errors in detection and classification. This could be accomplished by increasing the size of the dataset and by tuning the hyperparameters (values which must be manually set before training) to help the model more accurately locate and classify the different traffic lights (Solegaonkar, 2019). Overall, the image classification and object detection models have the potential to greatly reduce accident rates for visually impaired pedestrians.

References

300mm Green Walk Man Pedestrian Crossing Light For Road Safety - Buy Pedestrian Crossing Light,Pedestrian Traffic Light,Led Traffic Light Product on Alibaba.com. (n.d.). Retrieved from https://www.alibaba.com/product-detail/300mm-green-walk-man-pedestrian-crossing_60756182483.html

Al-Fuqaha, A., Oh, J. S., & Kwigizile, V. (2018). 16-06 Vehicle-to-Device (V2D) Communications: Readiness of the Technology and Potential Applications for People with Disability.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (2011, June). Flexible, high performance convolutional neural networks for image classification. In Twenty-Second International Joint Conference on Artificial Intelligence.

Cheng, Ruiqi & Wang, Kaiwei & Yang, Kailun & Long, Ningbo & Bai, Jian & Liu, Dong. (2017). Real-time pedestrian crossing lights detection algorithm for the visually impaired. Multimedia Tools and Applications. 77. 10.1007/s11042-017-5472-5.

Everything you need to know about Neural Networks. (2017). Retrieved from https://hackernoon.com/everything-you-need-to-know-about-neural-networks-8988c3ee4491

Hassan, S. E., & Massof, R. W. (2012). Measurements of street-crossing decision-making in pedestrians with low vision. Accident; analysis and prevention, 49, 410–418. doi: 10.1016/j.aap.2012.03.009

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., ... & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2704-2713).

Kistler, S. E. (2015, September 17). Why do intersections have different audible crosswalk

　　　signals? Retrieved from https://www.dispatch.com/article/20150911/NEWS/309119629

Lapedes, A., & Farber, R. (1989). How Neural Nets Work. Evolution, Learning and Cognition,

　　　331–346. doi: 10.1142/9789814434102_0012

Liu, W., Wen, Y., Yu, Z., & Yang, M. (2016, June). Large-margin softmax loss for

　　　convolutional neural networks. In ICML (Vol. 2, No. 3, p. 7).

Moore, A. W. (2001). Cross-validation for detecting and preventing overfitting. School of

　　　Computer Science Carnegie Mellon University.

Murphy, E. F. (1965). Some notes on canes and cane tips. Bulletin of Prosthetics Research - Fall

　　　1977, 28. doi: 10.1.1.599.307

New York 'Walk / Don't Walk' Sign. (n.d.). Retrieved from

　　　https://acesspeedshop.co.uk/product/new-york-walk-dont-walk-sign/

Pandey, P. (2019, March 28). Understanding the Mathematics behind Gradient

　　　Descent. Retrieved from https://towardsdatascience.com/understanding-the-

　　　mathematics-behind-gradient-descent-dde5dc9be06e

Parmar, R. (2018, September 02). Common Loss functions in machine learning. Retrieved

　　　from https://towardsdatascience.com/common-loss-functions-in-machine-learning-

　　　46af0ffc4d23

Pharr, Jennifer & Coughenour, Courtney & Bungum, Timothy. (2013). Environmental, Human

　　　and Socioeconomic Characteristics of Pedestrian Injury and Death in Las Vegas, NV..

　　　International Journal of Science. 2. 31-39.

Retting, R. (2017). Pedestrian traffic fatalities by state. Governors Highway Safety

　　　Association, Washington, DC.

Sharma, A. (2017, March 30). Understanding Activation Functions in Neural

    Networks. Retrieved from https://medium.com/the-theory-of-everything/understanding-

    activation-functions-in-neural-networks-9491262884e0

Solegaonkar, V. (2019, February 6). Error Analysis in Neural Networks. Retrieved from

    https://towardsdatascience.com/error-analysis-in-neural-networks-6b0785858845

Szeto, A. Y., Valerio, N. C., & Novak, R. E. (1991). Audible pedestrian traffic signals: Part 3.

    Detectability. *Journal of rehabilitation research and development*, *28*(2), 71-78.

Team, C. (2016, June 29). Jaywalking defined and explained with examples. Retrieved from

    https://legaldictionary.net/jaywalking/

Technoaddict. (2018, August 08). 5 Most Used IDE For Developing Android Apps. Retrieved

    from https://medium.com/@technoaddict/5-most-used-ide-for-developing-android-apps-

    2aa601cb23ca

Tensorflow. (2020, March 7). tensorflow/models. Retrieved from

    https://github.com/tensorflow/models/tree/master/official

TensorFlow Lite. (2019). Retrieved from https://www.tensorflow.org/lite

Thrun, S. B. (1992). Efficient exploration in reinforcement learning. SpringerReference. doi:

    10.1007/springerreference_178956

Vani, S. T. (2017, September 21). The pedestrian signal. Retrieved from

    https://medium.com/@sumukhatv/the-pedestrian-signal-11426f4c836d

Appendix A: Limitations and Assumptions

Assumptions

One important assumption made in this project was that the most effective way to ensure that it is safe to cross a street is through pedestrian traffic lights. While tracking pedestrian traffic lights is likely an effective way to ensure safety in crossing the street, there are many other potential metrics that could also be used, such as the detection of moving vehicles or tracking other pedestrians moving in a certain direction. Another assumption made in this project is that the Single Shot Detector MobileNet neural network model was the optimal model to retrain for object detection. While the MobileNet model is considered one of the best models for fast and efficient inference, many other models and architectures exist, including Fast R-CNNs, Single Shot Detector Inception, and Single Shot Detector Resnet (TensorFlow, 2020). One more assumption made was that the final precision and recall values of 82 percent and 90 percent were high enough for the model to be implemented for use by visually impaired pedestrians.

Limitations

One limitation for this project was the available hardware that could be used for training the neural networks. The training was limited to a MacBook Pro with a 1.4 GHz processor and an Intel i5 processor. The laptop did not have a Graphics Processing Unit, which the speed of the neural network training and the number of epochs that it could be trained for. Another limitation was the amount of data that could be used for training the neural networks. The images were obtained from Google Images via a search API, and many of the images used for training did not contain pedestrian traffic lights, so they were not helpful for training the models. As a result, only about 1,000 images could be used for training the models, which limited the models' final accuracies.

Appendix B: Project Notes

Knowledge Gaps:

Knowledge Gap	Resolved By	Information is located	Date resolved
Dataset of pedestrian traffic lights	Article 10	Notes for article 10 (Bosch Dataset)	11/4/19
Train a neural network for object recognition (but not detection)	Articles 5,6,7, and 8	Terms or concepts to study further" for articles 5,6,7, and 8	10/19/19
Methods of neural network to improve accuracy	Articles 5,6,7,8, and 9 (partially resolved, more research needed on the specifics of the methods)	"Terms or concepts to study further" for articles 5,6,7,8, and 9	11/04/19
Creating an effective Object Detection Neural Network	Articles 11, 12, and 13	Notes for articles 11,12, and 13	01/20/20 and 01/26/20

Literature Search Parameters:

These searches were performed between 09/01/2019 and 03/07/2020.
List of keywords and databases used during this project.

Database/search engine	Keywords	Summary of search
Google Scholar	Pedestrian Traffic Light, Live Recognition	Obtained a few articles about live recognition used on both pedestrian and car traffic lights
IEEE	Convolutional Neural Network, Traffic Light	Got articles that contained useful methods of using neural networks for detection and recognition
Google Scholar	Traffic light, object recognition	Found some articles about traffic light recognition, mainly for cars
Google Scholar	Traffic light convolutional neural network	Obtained articles about specific types of convolutional neural networks that were used for detecting traffic lights (mainly car)

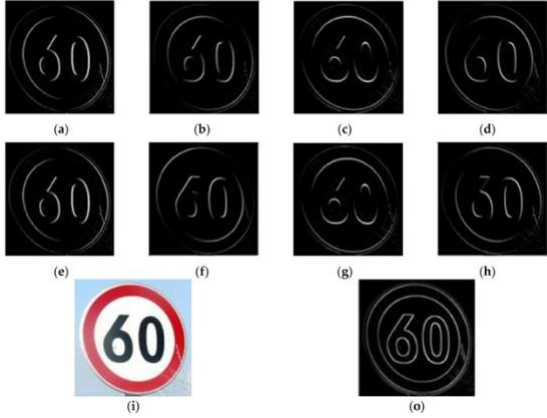| Google Scholar	Object Detection, R-CNN	Found articles describing more effective forms of object detection models and R-CNNs than the typical forms (such as LibraNet described in Article #13 Notes).

Article #1 Notes: CNN Design for Real-Time Traffic Sign Recognition

Source Title	CNN Design for Real-Time Traffic Sign Recognition
Source Author	A. Shustanov, P. Yakimov
Source citation	AlexanderShustanova. (2017, September 27). CNN Design for Real-Time Traffic Sign Recognition. Retrieved from https://www.sciencedirect.com/science/article/pii/S1877705817341231
Source type	Article
Keywords	TensorFlow Convolutional Neural Networks Traffic Sign Recognition Image Processing Computer Vision Mobile GPU
Summary of key points	Computer vision system in cars can help improve safety while driving by partially automating some procedures which commonly lead to crash or injury. One such problem is with traffic lights and street signs, and computer vision can help automatically stop the car or warn the driver during a red light or stop sign to help prevent a crash.
Important Figures	Fig. 6. Localized and recognized traffic signs.
Reason for interest	Very similar technology could be used for pedestrian traffic lights and street signs, rather than for cars.

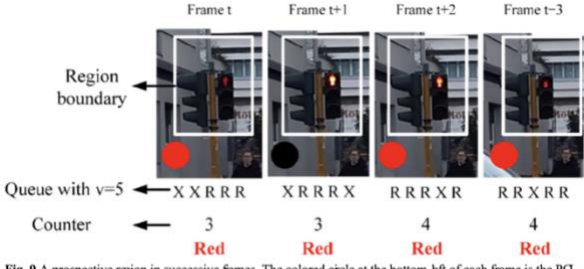|	
Notes	The device uses tensorflow and does live object recognition, which is exactly what I want to use for my project. This could be helpful in figuring out how to set up my neural network
Terms or concepts to study further:	- CUDA - German Traffic Sign Detection Benchmark - German Traffic Sign Recognition Benchmark - TensorBoard

Article #2 Notes: Real-Time Traffic Sign Detection and Recognition Method Based on Simplified Gabor Wavelets and CNNs

Source Title	Real-Time Traffic Sign Detection and Recognition Method Based on Simplified Gabor Wavelets and CNNs
Source Author	F. Shao, X. Wang, F. Meng, T. Rui, D. Wang, J. Tang
Source citation	Shao, F., Wang, X., Meng, F., Rui, T., Wang, D., & Tang, J. (2018). Real-Time Traffic Sign Detection and Recognition Method Based on Simplified Gabor Wavelets and CNNs. *Sensors (Basel, Switzerland)*, *18*(10), 3192. doi:10.3390/s18103192
Source type	Article
Keywords	simplified Gabor wavelets, MSERs, regions of interest, SVM, CNN
Summary of key points	The images were converted to grayscale, the edges of the signs were detected, and a CNN was used to recognize the type of sign. Simplified Gabor Wavelets and Support Vector Machines were used in the recognition

Important Figures	
Reason for interest	Traffic light detection and pedestrian light detection are very similar, so I can use similar methods in my project to what these researchers used.
Notes	I need to figure out what SVMs and SGRs are and how they are used in this project. I should also see if they might make my neural network more efficient/effective
Terms or concepts to study further:	- Support Vector Machines - Simplified Gabor Wavelets

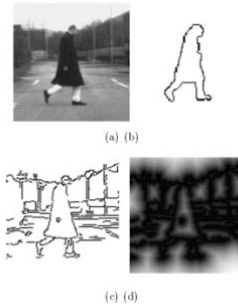**Article #3 Notes: Real-time pedestrian crossing lights detection algorithm for the visually impaired**

Source Title	Real-time pedestrian crossing lights detection algorithm for the visually impaired
Source Author	R. Cheng, K. Wang, K. Yang, N. Long, J. Bai, D. Liu
Source citation	SCheng, Ruiqi & Wang, Kaiwei & Yang, Kailun & Long, Ningbo & Bai, Jian & Liu, Dong. (2017). Real-time pedestrian crossing lights detection algorithm for the visually impaired. Multimedia Tools and Applications. 77. 10.1007/s11042-017-5472-5.
Source type	Article
Keywords	Pedestrian crossing lights detection .Real-time video processing .Candidate extraction and recognition .Temporal-spatial analysis .Visually impaired people
Summary of key points	A CNN was used to help blind people while they are crossing the street so that they can feel safe while outside.

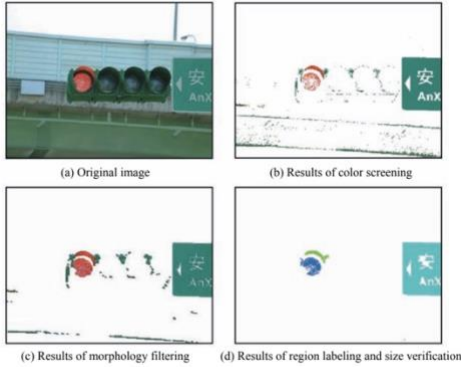| Important Figures	
	Fig. 9 A prospective region in successive frames. The colored circle at the bottom-left of each frame is the PCL recognition result of that frame. The red denotes that a red PCL is recognized, and the black denotes that no PCL is detected. The words at the bottom are the final results after temporal-spatial analysis
Reason for interest	This is very similar to a big portion of my project, so this will help me with training my model.
Notes	Similar to my idea, but the difference between this and my project is that this one is based on lights that look similar to US car traffic lights, while mine will use standard United States pedestrian traffic lights. Mine will also include human, obstruction, and street sign recognition in addition.
Terms or concepts to study further:	- Temporal-Spatial Analysis - Histograms of Oriented Gradient

Article #4 Notes: Real-time people and vehicle detection from UAV imagery

Source Title	Real-time people and vehicle detection from UAV imagery
Source Author	A. Gaszczak; T. Breckon; J. Han
Source citation	Anna Gaszczak, Toby P. Breckon, Jiwan Han, "Real-time people and vehicle detection from UAV imagery," Proc. SPIE 7878, Intelligent Robots and Computer Vision XXVIII: Algorithms and Techniques, 78780B (24 January 2011);
Source type	Article
Keywords	Cascaded Haar Classifiers, Gaussian Shape Matching, Episodic Object Detection Rate
Summary of key points	Vehicles and people were detected and classified autonomously by a neural network using aerial images taken by a drone.
Important Figures	The detection rate for people is ~70% and cars ~80% although the overall episodic object detection rate for each flight pattern exceeds 90%.
Reason for interest	My project incorporates live object detection, so this will be helpful in figuring out how to accomplish that.
Notes	The neural network had high accuracy using techniques such as cascaded haar classifiers and gaussian shape matching. I should study up on these and see if I can implement them in my project to improve the accuracy or speed of my model.
Terms or concepts to study further:	- Cascaded Haar Classifiers - Gaussian Shape Matching - Episodic Object Detection Rate

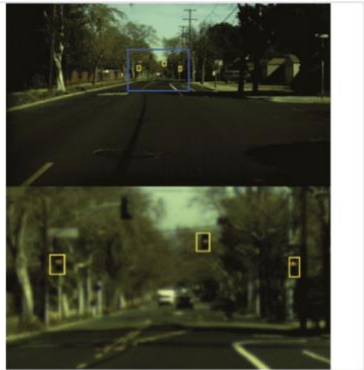**Article #5 Notes: Real-time Object Detection For "Smart" Vehicles**

Source Title	Real-time Object Detection For "Smart" Vehicles
Source Author	D.M. Gavrila, V. Philomin
Source citation	Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788).
Source type	Article
Keywords	Distance Transforms, SIMD parallelism, transformation parameters, stochastic optimization techniques
Summary of key points	Object detection was used in a moving vehicle to detect objects such as humans and street signs. It was used to alert the driver if they made errors such as speeding or making a wrong turn. A Distance Transform was used for the object detection.
Important Figures	Figure 2: (a) original image (b) template (c) edge image (d) DT image
Reason for interest	My project incorporates live object detection, so this will be helpful in figuring out how to accomplish that. This also uses humans and street signs, which are related to my project.
Notes	The neural network had high accuracy using the Distance Transform, so I should do research on what that means and how I can use it for my project. I should also research SIMD parallelism and what it can be used for.
Terms or concepts to study further:	• Distance Transforms • SIMD parallelism • transformation parameters • stochastic optimization techniques

Article #6 Notes: Traffic Light Recognition

Source Title	Traffic Light Recognition
Source Author	K. Lu, C. Wang, S. Chen
Source citation	Lu, K. H., Wang, C. M., & Chen, S. Y. (2008). Traffic light recognition. Journal of the Chinese institute of engineers, 31(6), 1069-1075.
Source type	Article
Keywords	intelligent transportation system, traffic light detection, traffic light classification, traffic light recognition, color identification, environmental adaptability, real-time implementation
Summary of key points	An automatic traffic light detection and recognition system was used to help drivers make decisions while driving. The system covers both detection and classification of traffic lights, and uses HSI color spaces for detection.
Important Figures	(a) Original image (b) Results of color screening (c) Results of morphology filtering (d) Results of region labeling and size verification Fig. 1 Experimental results of traffic light detection
Reason for interest	My project incorporates live object detection, so this will be helpful in figuring out how to accomplish that. This also uses traffic lights, which is similar to what my project does.
Notes	They separated detection and classification into two separate phases, which will probably be a good idea for my project. Detection uses HSI color spaces (I will have to do more research on this topic). Classification uses border detection to obtain features (which I can do with a Convolutional Neural Network).

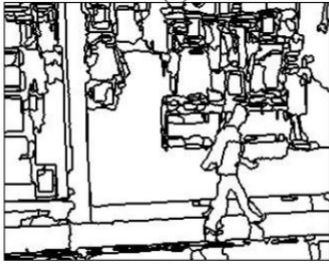| Terms or concepts to study further:	• Intelligent Transportation System • HSI color space
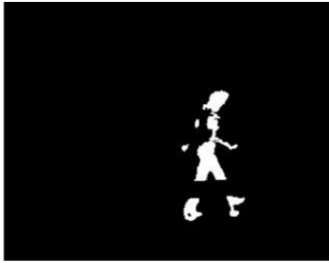
Article #7 Notes: A deep learning approach to traffic lights: Detection, tracking, and classification

Source Title	A deep learning approach to traffic lights: Detection, tracking, and classification
Source Author	K. Behrendt, L. Novak, R. Botros
Source citation	Behrendt, K., Novak, L., & Botros, R. (2017, May). A deep learning approach to traffic lights: Detection, tracking, and classification. In 2017 IEEE International Conference on Robotics and Automation (ICRA) (pp. 1370-1377). IEEE.
Source type	Article
Keywords	Training, Neural networks, Computer architecture, Agriculture, Machine learning, Microprocessors, Kernel
Summary of key points	An automatic traffic light detection system was created. It runs at 10 frames per second and was highly accurate at detecting the traffic lights. It was able to accurately identify images only 1 pixel in width and was accurate over 90% of the time overall.
Important Figures	 Fig. 1: Sample detections of small traffic lights in an image. The top image is taken at the full resolution of 1280 x 720. At the bottom, the enlarged crop shows detected traffic lights of size about 6 x 12 pixels. All lights are correctly classified as yellow.
Reason for interest	My project incorporates live object detection, so this will be helpful in figuring out how to accomplish that. This also uses traffic lights, which is similar to what my project does.

Notes	They used the Bosch Small Traffic Lights Dataset, which may contain the pedestrian traffic lights that I need for my project. I can also incorporate some of the techniques used in this, such as Stereo Vision, in making my neural network.
Terms or concepts to study further:	• Stereo vision • Bosch Small Traffic Lights Dataset
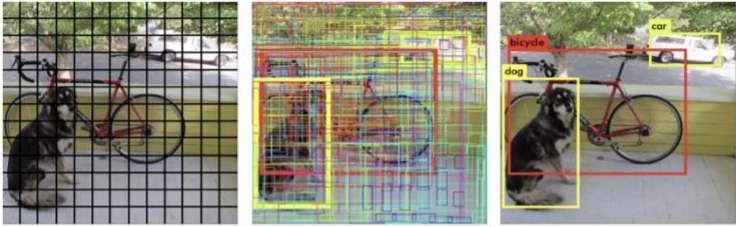
Article #8 Notes: A Moving Object Detection Algorithm Based on Color Information

Source Title	A Moving Object Detection Algorithm Based on Color Information
Source Author	X. H. Fang, W. Xiong, B. J. Hu, L. T. Wang
Source citation	X H Fang et al 2006 J. Phys.: Conf. Ser. 48 384
Source type	Article
Keywords	YUV Chrominance, Gaussians, Color Segmentation
Summary of key points	In this paper, the scientists created a model that can perform object detection on moving objects by using the colors of the different pixels. They used color segmentation to highlight the object and ignore the background (as shown in the figure below),

Important Figures	

(a) (b) (c)

(d) (e) (f)

Reason for interest	My object detection model will need to run efficiently in real-time, so if the model I create does not work as efficiently as it needs to, this can help me resolve the issue. Although my objects (traffic lights) will not be moving, the camera will be, so this will still be helpful.
Notes	The real-time capability of the model is not fully effective, as stated in the paper's conclusion. Based on a simulation, this algorithm is an efficient way to perform object detection on moving objects.
Terms or concepts to study further:	• Color Segmentation • Gaussian • Chrominance Component Pixel

Article #9 Notes: Mobile Object Detection using TensorFlow Lite and Transfer Learning

Source Title	Mobile Object Detection using TensorFlow Lite and Transfer Learning
Source Author	O. Alsing
Source citation	Alsing, O. (2018). Mobile Object Detection using TensorFlow Lite and Transfer Learning (Dissertation). Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-233775
Source type	Article
Keywords	CNN, convolutional neural networks, transfer learning, mobile object detection

Summary of key points	This paper discusses the use of deep learning models on mobile devices to perform live object detection. It describes the use of TensorFlow and TensorFlow Lite and discusses the process of creating and implementing an object detection model for mobile devices. The model described in the article detects post-it notes.
Important Figures	 Figure 2.2: The left image shows how the YOLO architecture image is split into an grid, and the middle image displays how this grid is used to evaluate multiple sub-windows of the image. The right image displays the original image with the corresponding GT boxes [40].
Reason for interest	I plan to implement my object detection model for use on Android devices, so the information in this article is helpful in understanding how to effectively implement the model for an Android application.
Notes	The YOLO Network is an effective way to perform object detection (shown in figure 2.2). TensorFlow Lite contains many optimizations such as hardware acceleration to allow MobileNets to effectively run on mobile devices.
Terms or concepts to study further:	• Mean Average Precision • Precision, Recall, F1 • MobileNet, ResNet

Article #10 Notes: Libra R-CNN: Towards Balanced Learning for Object Detection

Source Title	Libra R-CNN: Towards Balanced Learning for Object Detection
Source Author	J. Pang, K. Chen, J. Shi, H. Feng, W. Ouyang, D. Lin
Source citation	Pang, J., Chen, K., Shi, J., Feng, H., Ouyang, W., & Lin, D. (2019). Libra r-cnn: Towards balanced learning for object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 821-830).
Source type	Article
Keywords	None

Summary of key points	This paper discusses the creation of a balanced R-CNN (object detection neural network). Most R-CNNs have imbalance in the sample level, feature level, and objective level. The Libra R-CNN (the model created in this paper) achieves higher precision than standard R-CNN types (such as RetinaNet and Faster R-CNN). This R-CNN achieves greater balance due to its use of balanced sampling, balanced feature pyramids, and balanced loss functions.
Important Figures	
Reason for interest	My neural network will be an R-CNN (most likely a Faster R-CNN), so this paper can help me improve my model's accuracy by creating greater balance.
Notes	The model created in this paper is able to reach a higher accuracy on a very common large dataset (the MS COCO dataset). This means that it can likely have a major positive impact on the accuracy of my model. This is definitely worth doing some research into and figuring out how to implement it with my project (the model described in the paper is in PyTorch, which my model will also use, so that should make this easier to accomplish)
Terms or concepts to study further:	• Feature Pyramid • IoU (Intersection over Union) • RetinaNet

Appendix C: Decision Matrix and Competitor Analysis

Decision Matrix

		K-Nearest Neighbors Classifier	**Convolutional Neural Network**	Color Sensor	Use an API to locate intersections
Criteria	Weight	**A**	**B**	**C**	**D**
Accuracy	5	3	4	2	3
Runs/Evaluates Quickly	4	3	3	4	2
User-Friendly/Easy to use	4	5	5	5	3
Easy to create given data	2	4	4	4	2
Easy to obtain data	2	3	3	5	3
Works offline	3	5	5	5	1
Easy to implement	4	3	3	4	3
Total	N/A	100	**108**	98	63

Competitor Analysis

Criteria	Weight	Competitor: Audible Pedestrian Signals **A**	**Object Detection Model** **B**
Accuracy	5	5	4
Runs/Evaluates Quickly	4	5	3
User-Friendly/ Easy to use	4	3	5
Easy to create	2	2	5
Works offline	3	5	5
Easy to implement	4	2	4
Total		83	**93**

Appendix D: Project Code

File: imgread.py

```python
#!/usr/bin/env python
# coding: utf-8

# This file is where the image classification model was created, trained and exported to TensorFlow Lite

import cv2
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models

"""
Used to load the train images as numpy arrays
"""
greendata = []
for i in range(1,10):
    x = 'Images/green/0000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (100,100,3)
        greendata.append(cv2.imread(x))
    except:
        pass
for i in range(10,100):
    x = 'Images/green/000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (100,100,3)
        greendata.append(cv2.imread(x))
    except:
        pass
for i in range(100,200):
    x = 'Images/green/00000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (100,100,3)
        greendata.append(cv2.imread(x))
    except:
        pass
reddata = []
for i in range(1,10):
    x = 'Images/red/0000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (100,100,3)
        reddata.append(cv2.imread(x))
    except:
        pass
```

```python
for i in range(10,100):
    x = 'Images/red/000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (100,100,3)
        reddata.append(cv2.imread(x))
    except:
        pass
for i in range(100,200):
    x = 'Images/red/00000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (100,100,3)
        reddata.append(cv2.imread(x))
    except:
        pass
reddata = np.array(reddata)
greendata = np.array(greendata)
print(reddata.shape)
print(greendata.shape)


"""
Shuffles two arrays (the images and the indices) in the same way so that corresponding values in each
array are still the same as they initially were
"""
def same_shuffle(arr1,arr2):
    size = arr2.size
    print(size)
    idxs = np.arange(size)
    np.random.shuffle(idxs)
    return arr1[idxs], arr2[idxs]



"""
Splits the data into train and validation sets
"""
greenlist = list(greendata)
redlist = list(reddata)
datalist = greenlist+redlist
data = np.array(datalist)
y_green = list(np.zeros(149))
y_red = list(np.zeros(128)+1)
y = np.array(y_green + y_red)
data, y = same_shuffle(data, y)
data_test = data[200:]
data = data[:200]
y_test = y[200:]
y = y[:200]
print(data_test.shape)
print(data.shape)


"""
```

Initializes the image classification neural network model in TensorFlow/Keras

```python
"""
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(100, 100, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(16, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dropout(0.2))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(8, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(2, activation='softmax'))
model.summary()
model.compile(optimizer=tf.optimizers.Adam(),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])


"""
```

Trains the neural network on the train data and validates it on the validation set. Also saves the model as a .h5 file after each of the 50 epochs.

```python
"""
filepath = 'weights.{epoch:02d}-{val_accuracy:.2f}.h5'
j = [tf.keras.callbacks.ModelCheckpoint(filepath, monitor='val_accuracy', verbose=0, save_best_only=False,
save_weights_only=False, mode='auto', period=1)]
history = model.fit(data, y, epochs=50,validation_data=(data_test,y_test), callbacks=j)


"""
```

Used to plot the loss and accuracy values for the model over the 50 epochs.

```python
"""
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.plot(history.history['val_accuracy'], label='val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
# plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.plot(history.history['accuracy'], label='train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
# plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.plot(history.history['loss'], label='train loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.xlim([0,21])
plt.legend(loc='lower right')
```

```python
# Saves the model
model.save('nn.h5')


converter = tf.lite.TFLiteConverter.from_keras_model(model)


tflite_model = converter.convert()


"""
Converts the model into a Float TensorFlow Lite model
"""
with open ('model.tflite', mode='wb') as file:
    file.write(tflite_model)


"""
Converts the model into a Quantized TensorFlow Lite model
"""
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_quant_model = converter.convert()
with open ('detect_model.tflite', mode='wb') as file:
    file.write(tflite_quant_model)
```

File: inference.py

```python
#!/usr/bin/env python
# coding: utf-8

# # This file was used for live inference with the image classification model


import tensorflow as tf
import cv2
import numpy as np

"""
Runs a videocapture to obtain an image from the webcam and then runs inference on the image using
the image classification model to determine
whether the image is of a green traffic light or a red traffic light
"""

# Loads a saved TensorFlow model
model = tf.keras.models.load_model('weights.50-0.74.h5')

cam = cv2.VideoCapture(0)
```

```python
cv2.namedWindow("test")

img_counter = 0

while True:
    ret, frame = cam.read()
    cv2.imshow("test", frame)
    if not ret:
        break
    k = cv2.waitKey(1)

    if k%256 == 27:
        # ESC pressed
        print("Escape hit, closing...")
        break
    elif k%256 == 32:
        # SPACE pressed
        img_name = "opencv_frame_0.png"
        cv2.imwrite(img_name, frame)
        print("{} written!".format(img_name))
        img_counter += 1
        break

cam.release()
cv2.destroyAllWindows()

array_img = cv2.imread('opencv_frame_0.png')
frame = cv2.resize(frame, (100,100))
labels = ['Green', 'Red']


labels = ['Green', 'Red']
print(labels[np.argmax(model.predict(np.array([frame], dtype=np.float32))[0])])

print(model.predict(np.array([frame], dtype=np.float32))[0])
```

File: XML_Read.py

```python
#!/usr/bin/env python
# coding: utf-8

# # This file was used to parse the XML files for Object Detection


import numpy
import numpy as np
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```python
"""
Rectangle class that allows for easier use/access to the four coordinates of a bounding box
"""
class rectangle:
    def __init__(self, xmin, ymin, xmax, ymax):
        self.xmin = xmin
        self.ymin = ymin
        self.xmax = xmax
        self.ymax = ymax


    def __repr__(self):
        return str(np.array([self.xmin, self.ymin, self.xmax, self.ymax]))



    def toArray(self):
        return np.array([self.xmin, self.ymin, self.xmax, self.ymax])

"""
Reads the XML files and appends all important data to the 'boxes' and 'colors' lists
"""
boxes = []
colors = []
for i in range(10):
    tmp_box = []
    tmp_color = []
    try:
        tree = ET.parse('images_new_resized/green_new_labels/0000000'+str(i)+'.xml')
        root = tree.getroot()
        for child in root:
            if child.tag == 'object' :
                for subchild in child:
                    if subchild.tag == 'name':
                        if subchild.text == 'green':
                            tmp_color.append(0)
                        elif subchild.text == 'red':
                            tmp_color.append(1)
                        else:
                            print("something is wrong with "+str(i))
                for subchild in child:
                    if subchild.tag == 'bndbox':
                        for subsubchild in subchild:
                            if subsubchild.tag == 'xmin':
                                xmin = int(subsubchild.text)
                            elif subsubchild.tag == 'ymin':
                                ymin = int(subsubchild.text)
                            elif subsubchild.tag == 'xmax':
                                xmax = int(subsubchild.text)
                            elif subsubchild.tag == 'ymax':
                                ymax = int(subsubchild.text)
                        tmp_box.append(rectangle(xmin, ymin, xmax, ymax).toArray())
    except:
        boxes.append([])
        colors.append([])
```

```python
                continue

        boxes.append(tmp_box)
        colors.append(tmp_color)

for i in range(10,100):
    tmp_box = []
    tmp_color = []
    try:
        tree = ET.parse('images_new_resized/green_new_labels/000000'+str(i)+'.xml')
        root = tree.getroot()
        for child in root:
            if child.tag == 'object' :
                for subchild in child:
                    if subchild.tag == 'name':
                        if subchild.text == 'green':
                            tmp_color.append(0)
                        elif subchild.text == 'red':
                            tmp_color.append(1)
                        else:
                            print("something is wrong with "+str(i))
                for subchild in child:
                    if subchild.tag == 'bndbox':
                        for subsubchild in subchild:
                            if subsubchild.tag == 'xmin':
                                xmin = int(subsubchild.text)
                            elif subsubchild.tag == 'ymin':
                                ymin = int(subsubchild.text)
                            elif subsubchild.tag == 'xmax':
                                xmax = int(subsubchild.text)
                            elif subsubchild.tag == 'ymax':
                                ymax = int(subsubchild.text)
                        tmp_box.append(rectangle(xmin, ymin, xmax, ymax).toArray())
    except:
        boxes.append([])
        colors.append([])
        continue

    boxes.append(tmp_box)
    colors.append(tmp_color)


for i in range(100,480):
    tmp_box = []
    tmp_color = []
    try:
        tree = ET.parse('images_new_resized/green_new_labels/00000'+str(i)+'.xml')
        root = tree.getroot()
        for child in root:
            if child.tag == 'object' :
                for subchild in child:
                    if subchild.tag == 'name':
                        if subchild.text == 'green':
```

```python
                    tmp_color.append(0)
                elif subchild.text == 'red':
                    tmp_color.append(1)
                else:
                    print("something is wrong with "+str(i))
            for subchild in child:
                if subchild.tag == 'bndbox':
                    for subsubchild in subchild:
                        if subsubchild.tag == 'xmin':
                            xmin = int(subsubchild.text)
                        elif subsubchild.tag == 'ymin':
                            ymin = int(subsubchild.text)
                        elif subsubchild.tag == 'xmax':
                            xmax = int(subsubchild.text)
                        elif subsubchild.tag == 'ymax':
                            ymax = int(subsubchild.text)
                    tmp_box.append(rectangle(xmin, ymin, xmax, ymax).toArray())
    except:
        boxes.append([])
        colors.append([])
        continue

    boxes.append(tmp_box)
    colors.append(tmp_color)
print(len(colors))
print(len(boxes))

for i in range(10):
    tmp_box = []
    tmp_color = []
    try:
        tree = ET.parse('images_new_resized/red_new_labels/0000000'+str(i)+'.xml')
        root = tree.getroot()
        for child in root:
            if child.tag == 'object' :
                for subchild in child:
                    if subchild.tag == 'name':
                        if subchild.text == 'green':
                            tmp_color.append(0)
                        elif subchild.text == 'red':
                            tmp_color.append(1)
                        else:
                            print("something is wrong with "+str(i))
                for subchild in child:
                    if subchild.tag == 'bndbox':
                        for subsubchild in subchild:
                            if subsubchild.tag == 'xmin':
                                xmin = int(subsubchild.text)
                            elif subsubchild.tag == 'ymin':
                                ymin = int(subsubchild.text)
                            elif subsubchild.tag == 'xmax':
                                xmax = int(subsubchild.text)
                            elif subsubchild.tag == 'ymax':
```

```python
                    ymax = int(subsubchild.text)
                tmp_box.append(rectangle(xmin, ymin, xmax, ymax).toArray())
    except:
        boxes.append([])
        colors.append([])
        continue

    boxes.append(tmp_box)
    colors.append(tmp_color)

for i in range(10,100):
    tmp_box = []
    tmp_color = []
    try:
        tree = ET.parse('images_new_resized/red_new_labels/000000'+str(i)+'.xml')
        root = tree.getroot()
        for child in root:
            if child.tag == 'object' :
                for subchild in child:
                    if subchild.tag == 'name':
                        if subchild.text == 'green':
                            tmp_color.append(0)
                        elif subchild.text == 'red':
                            tmp_color.append(1)
                        else:
                            print("something is wrong with "+str(i))
                for subchild in child:
                    if subchild.tag == 'bndbox':
                        for subsubchild in subchild:
                            if subsubchild.tag == 'xmin':
                                xmin = int(subsubchild.text)
                            elif subsubchild.tag == 'ymin':
                                ymin = int(subsubchild.text)
                            elif subsubchild.tag == 'xmax':
                                xmax = int(subsubchild.text)
                            elif subsubchild.tag == 'ymax':
                                ymax = int(subsubchild.text)
                        tmp_box.append(rectangle(xmin, ymin, xmax, ymax).toArray())
    except:
        boxes.append([])
        colors.append([])
        continue

    boxes.append(tmp_box)
    colors.append(tmp_color)


for i in range(100,500):
    tmp_box = []
    tmp_color = []
    try:
        tree = ET.parse('images_new_resized/red_new_labels/00000'+str(i)+'.xml')
        root = tree.getroot()
```

```python
    for child in root:
        if child.tag == 'object' :
            for subchild in child:
                if subchild.tag == 'name':
                    if subchild.text == 'green':
                        tmp_color.append(0)
                    elif subchild.text == 'red':
                        tmp_color.append(1)
                    else:
                        print("something is wrong with "+str(i))
            for subchild in child:
                if subchild.tag == 'bndbox':
                    for subsubchild in subchild:
                        if subsubchild.tag == 'xmin':
                            xmin = int(subsubchild.text)
                        elif subsubchild.tag == 'ymin':
                            ymin = int(subsubchild.text)
                        elif subsubchild.tag == 'xmax':
                            xmax = int(subsubchild.text)
                        elif subsubchild.tag == 'ymax':
                            ymax = int(subsubchild.text)
                    tmp_box.append(rectangle(xmin, ymin, xmax, ymax).toArray())
    except:
        boxes.append([])
        colors.append([])
        continue
    boxes.append(tmp_box)
    colors.append(tmp_color)

print(len(boxes))
print(len(colors))


"""
Removes non-array elements from all 3 data lists
"""
def remove_same(a, b, c):
    assert len(a) == len(b) and len(b) == len(c)
    for i in range(len(a)):
        try:
            if type(a[i]) != numpy.ndarray:
                a.pop(i)
                b.pop(i)
                c.pop(i)
                remove_same(a,b,c)
        except:
            pass


"""
Reads the .jpg image files and appends their data to the 'images' list
"""
```

```python
images = []
for i in range(10):
    x = 'images_new_resized/green_new/0000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (300,300,3)
        images.append(cv2.imread(x))
    except:
        images.append([])
for i in range(10,100):
    x = 'images_new_resized/green_new/000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (300,300,3)
        images.append(cv2.imread(x))
    except:
        images.append([])
for i in range(100,480):
    x = 'images_new_resized/green_new/00000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (300,300,3)
        images.append(cv2.imread(x))
    except:
        images.append([])


for i in range(10):
    x = 'images_new_resized/red_new/0000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (300,300,3)
        images.append(cv2.imread(x))
    except:
        images.append([])
for i in range(10,100):
    x = 'images_new_resized/red_new/000000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (300,300,3)
        images.append(cv2.imread(x))
    except:
        images.append([])
for i in range(100,500):
    x = 'images_new_resized/red_new/00000'+str(i)+".jpg"
    try:
        plt.imshow(cv2.imread(x));
        assert cv2.imread(x).shape == (300,300,3)
        images.append(cv2.imread(x))
    except:
        images.append([])

images = np.array(images)
```

```python
len(images)

print(len(images), len(boxes), len(colors))

images = list(images)
boxes = list(boxes)
colors = list(colors)
remove_same(images, boxes, colors)


print(len(images), len(boxes), len(colors))


"""
Converts images from BGR format to RGB format
"""
def bgr_to_rgb(img):
    return img[:, :, ::-1]


images = np.array(images)
boxes = np.array(boxes)
colors = np.array(colors)
```

File: object_detection_retraining.py

```python
# -*- coding: utf-8 -*-
"""Copy of tensorflow-object-detection-training-colab.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1AaedDJaE-SCkELO3qs3YBEZFkWO2jg8K

# Adapted from [How to train an object detection model easy for free](https://www.dlology.com/blog/how-to-
train-an-object-detection-model-easy-for-free/) | DLology Blog

## Configs and Hyperparameters

Support a variety of models, you can find more pretrained model from [Tensorflow detection model zoo:
COCO-trained
models](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_
zoo.md#coco-trained-models), as well as their pipline config files in
[object_detection/samples/configs/](https://github.com/tensorflow/models/tree/master/research/object_detecti
on/samples/configs).
"""

import os
import shutil
import glob
import urllib.request
import tarfile
```

```python
import re
import tensorflow as tf
import numpy as np
import cv2
import sys


from matplotlib import pyplot as plt
from PIL import Image
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth, files
from oauth2client.client import GoogleCredentials
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as vis_util
from object_detection.utils import ops as utils_ops


# If you forked the repository, you can replace the link.
repo_url = 'https://github.com/darshankrishnaswamy/object_detection_demo'

# Number of training steps.
num_steps = 500  # 200000

# Number of evaluation steps.
num_eval_steps = 100

MODELS_CONFIG = {
    'ssd_mobilenet_v2': {
        'model_name': 'ssd_mobilenet_v2_coco_2018_03_29',
        'pipeline_file': 'ssd_mobilenet_v2_coco.config',
        'batch_size': 10
    },
    'faster_rcnn_inception_v2': {
        'model_name': 'faster_rcnn_inception_v2_coco_2018_01_28',
        'pipeline_file': 'faster_rcnn_inception_v2_pets.config',
        'batch_size': 10
    },
    'rfcn_resnet101': {
        'model_name': 'rfcn_resnet101_coco_2018_01_28',
        'pipeline_file': 'rfcn_resnet101_pets.config',
        'batch_size': 10
    }
}

# Pick the model you want to use
# Select a model in `MODELS_CONFIG`.
selected_model = 'ssd_mobilenet_v2'

# Name of the object detection model to use.
MODEL = MODELS_CONFIG[selected_model]['model_name']

# Name of the pipline file in tensorflow object detection API.
```

```python
pipeline_file = MODELS_CONFIG[selected_model]['pipeline_file']

# Training batch size fits in Colabe's Tesla K80 GPU memory for selected model.
batch_size = MODELS_CONFIG[selected_model]['batch_size']

"""## Clone the `object_detection_demo` repository or your fork."""

# Commented out IPython magic to ensure Python compatibility.


# %cd /content

repo_dir_path = os.path.abspath(os.path.join('.', os.path.basename(repo_url)))

!git clone {repo_url}
# %cd {repo_dir_path}
!git pull

"""## Install required packages"""

# Commented out IPython magic to ensure Python compatibility.
# %cd /content
!git clone --quiet https://github.com/tensorflow/models.git

!apt-get install -qq protobuf-compiler python-pil python-lxml python-tk

!pip install -q Cython contextlib2 pillow lxml matplotlib

!pip install -q pycocotools

# %cd /content/models/research
!protoc object_detection/protos/*.proto --python_out=.

os.environ['PYTHONPATH'] += ':/content/models/research/:/content/models/research/slim/'

!python object_detection/builders/model_builder_test.py

"""## Prepare `tfrecord` files

Use the following scripts to generate the `tfrecord` files.
```bash
# Convert train folder annotation xml files to a single csv file,
# generate the `label_map.pbtxt` file to `data/` directory as well.
python xml_to_csv.py -i data/images/train -o data/annotations/train_labels.csv -l data/annotations

# Convert test folder annotation xml files to a single csv.
python xml_to_csv.py -i data/images/test -o data/annotations/test_labels.csv

# Generate `train.record`
python generate_tfrecord.py --csv_input=data/annotations/train_labels.csv --
output_path=data/annotations/train.record --img_path=data/images/train --label_map
data/annotations/label_map.pbtxt
```
```

```
# Generate `test.record`
python generate_tfrecord.py --csv_input=data/annotations/test_labels.csv --
output_path=data/annotations/test.record --img_path=data/images/test --label_map
data/annotations/label_map.pbtxt
```
"""

Commented out IPython magic to ensure Python compatibility.
%cd {repo_dir_path}

Convert train folder annotation xml files to a single csv file,
generate the `label_map.pbtxt` file to `data/` directory as well.
!python xml_to_csv.py -i data/images/train -o data/annotations/train_labels.csv -l data/annotations

Convert test folder annotation xml files to a single csv.
!python xml_to_csv.py -i data/images/test -o data/annotations/test_labels.csv

Generate `train.record`
!python generate_tfrecord.py --csv_input=data/annotations/train_labels.csv --
output_path=data/annotations/train.record --img_path=data/images/train --label_map
data/annotations/label_map.pbtxt

Generate `test.record`
!python generate_tfrecord.py --csv_input=data/annotations/test_labels.csv --
output_path=data/annotations/test.record --img_path=data/images/test --label_map
data/annotations/label_map.pbtxt

test_record_fname = '/content/object_detection_demo/data/annotations/test.record'
train_record_fname = '/content/object_detection_demo/data/annotations/train.record'
label_map_pbtxt_fname = '/content/object_detection_demo/data/annotations/label_map.pbtxt'

"""## Download base model"""

Commented out IPython magic to ensure Python compatibility.
%cd /content/models/research

MODEL_FILE = MODEL + '.tar.gz'
DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'
DEST_DIR = '/content/models/research/pretrained_model'

if not (os.path.exists(MODEL_FILE)):
    urllib.request.urlretrieve(DOWNLOAD_BASE + MODEL_FILE, MODEL_FILE)

tar = tarfile.open(MODEL_FILE)
tar.extractall()
tar.close()

os.remove(MODEL_FILE)
if (os.path.exists(DEST_DIR)):
    shutil.rmtree(DEST_DIR)
os.rename(MODEL, DEST_DIR)
```

```python
!echo {DEST_DIR}
!ls -alh {DEST_DIR}

fine_tune_checkpoint = os.path.join(DEST_DIR, "model.ckpt")
fine_tune_checkpoint

"""## Configuring a Training Pipeline"""


pipeline_fname = os.path.join('/content/models/research/object_detection/samples/configs/', pipeline_file)

assert os.path.isfile(pipeline_fname), '`{}` not exist'.format(pipeline_fname)

def get_num_classes(pbtxt_fname):

    label_map = label_map_util.load_labelmap(pbtxt_fname)
    categories = label_map_util.convert_label_map_to_categories(
        label_map, max_num_classes=2, use_display_name=True)
    category_index = label_map_util.create_category_index(categories)
    return len(category_index.keys())



num_classes = get_num_classes(label_map_pbtxt_fname)
with open(pipeline_fname) as f:
    s = f.read()
with open(pipeline_fname, 'w') as f:

    # fine_tune_checkpoint
    s = re.sub('fine_tune_checkpoint: ".*?"',
            'fine_tune_checkpoint: "{}"'.format(fine_tune_checkpoint), s)

    # tfrecord files train and test.
    s = re.sub(
        '(input_path: ".*?)(train.record)(.*?")', 'input_path: "{}"'.format(train_record_fname), s)
    s = re.sub(
        '(input_path: ".*?)(val.record)(.*?")', 'input_path: "{}"'.format(test_record_fname), s)

    # label_map_path
    s = re.sub(
        'label_map_path: ".*?"', 'label_map_path: "{}"'.format(label_map_pbtxt_fname), s)

    # Set training batch_size.
    s = re.sub('batch_size: [0-9]+',
            'batch_size: {}'.format(batch_size), s)

    # Set training steps, num_steps
    s = re.sub('num_steps: [0-9]+',
            'num_steps: {}'.format(num_steps), s)

    # Set number of classes num_classes.
    s = re.sub('num_classes: [0-9]+',
            'num_classes: {}'.format(num_classes), s)
```

```
    f.write(s)

!cat {pipeline_fname}

model_dir = 'training/'
# Optionally remove content in output model directory to fresh start.
!rm -rf {model_dir}
os.makedirs(model_dir, exist_ok=True)

"""## Run Tensorboard(Optional)"""

!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
!unzip -o ngrok-stable-linux-amd64.zip

LOG_DIR = model_dir
get_ipython().system_raw(
    'tensorboard --logdir {} --host 0.0.0.0 --port 6006 &'
    .format(LOG_DIR)
)

get_ipython().system_raw('./ngrok http 6006 &')

"""### Get Tensorboard link"""

! curl -s http://localhost:4040/api/tunnels | python3 -c \
    "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public_url'])"

"""## Train the model"""



!python /content/models/research/object_detection/model_main.py \
    --pipeline_config_path={pipeline_fname} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --num_eval_steps={num_eval_steps}



!ls {model_dir}

CONFIG_FILE =
"/content/models/research/object_detection/samples/configs/ssd_mobilenet_v1_coco.config"
CHECKPOINT_PATH = model_dir+"model.ckpt-500"
OUTPUT_DIR = "/content/tflite"

# Legacy way of training(also works).
# !python /content/models/research/object_detection/legacy/train.py --logtostderr --train_dir={model_dir} --
pipeline_config_path={pipeline_fname}

!python /content/models/research/object_detection/export_tflite_ssd_graph.py --
pipeline_config_path={CONFIG_FILE} --trained_checkpoint_prefix={CHECKPOINT_PATH} --
```

```
output_directory={OUTPUT_DIR} --add_postprocessing_op=true

!ls /content/tflite

!ls /content/models/research/object_detection/samples/configs/

!cat {model_dir}/graph.pbtxt

!ls

"""## Exporting a Trained Inference Graph
Once your training job is complete, you need to extract the newly trained inference graph, which will be
later used to perform the object detection. This can be done as follows:
"""

!ls
output_directory = './fine_tuned_model_2_2_2_2'

lst = os.listdir(model_dir)
lst = [l for l in lst if 'model.ckpt-' in l and '.meta' in l]
steps=np.array([int(re.findall('\d+', l)[0]) for l in lst])
last_model = lst[steps.argmax()].replace('.meta', '')

last_model_path = os.path.join(model_dir, last_model)
print(last_model_path)
!python /content/models/research/object_detection/export_tflite_ssd_graph.py \
    --input_type=tf.float32 \
    --input_shape 1,300,300,3 \
    --pipeline_config_path={pipeline_fname} \
    --output_directory={output_directory} \
    --trained_checkpoint_prefix={last_model_path}

!ls ./fine_tuned_model_2_2_2_2

!tflite_convert --output_file model.tflite \
        --saved_model_dir   ./fine_tuned_model_2_2_2_2/saved_model/ \
        --output_format TFLITE \
        --inference_type FLOAT \
        --input_arrays image_tensor \
        --input_shapes 1,300,300,3 \
        --output_arrays detection_boxes,detection_classes,detection_scores,num_detections

output_directory ="./fine_tuned_model_2_2_2_2"
!ls {output_directory}

cat fine_tuned_model_2_2_2_2/checkpoint

ls fine_tuned_model_2_2_2_2/saved_model/

!python object_detection/export_tflite_ssd_graph.py --
pipeline_config_path=./fine_tuned_model_2_2_2_2/pipeline.config --trained_checkpoint_prefix= --
output_directory=./tflite !--add_postprocessing_op=true
```

```python
"""## Download the model `.pb` file"""




pb_fname = os.path.join('/content/stemdata', "frozen_inference_graph(3).pb")
assert os.path.isfile(pb_fname), '`{}` not exist'.format(pb_fname)

ls stemdata

!ls -alh {pb_fname}

"""### Option1 : upload the `.pb` file to your Google Drive
Then download it from your Google Drive to local file system.

During this step, you will be prompted to enter the token.
"""

# Install the PyDrive wrapper & import libraries.
# This only needs to be done once in a notebook.
!pip install -U -q PyDrive




# Authenticate and create the PyDrive client.
# This only needs to be done once in a notebook.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

fname = os.path.basename(pb_fname)
# Create & upload a text file.
uploaded = drive.CreateFile({'title': fname})
uploaded.SetContentFile(pb_fname)
uploaded.Upload()
print('Uploaded file with ID {}'.format(uploaded.get('id')))

"""### Option2 :  Download the `.pb` file directly to your local file system
This method may not be stable when downloading large files like the model `.pb` file. Try **option 1**
instead if not working.
"""


files.download(pb_fname)

"""### Download the `label_map.pbtxt` file"""


files.download(label_map_pbtxt_fname)

"""### Download the modified pipline file
If you plan to use OpenVINO toolkit to convert the `.pb` file to inference faster on Intel's hardware
(CPU/GPU, Movidius, etc.)
```

```
"""

files.download(pipeline_fname)

# !tar cfz fine_tuned_model.tar.gz fine_tuned_model
# from google.colab import files
# files.download('fine_tuned_model.tar.gz')

repo_dir_path = "object_detection_demo/"

ls object_detection_demo/data/images/test

!git clone https://github.com/darshankrishnaswamy/stemdata

!ls stemdata



"""## Run inference test
Test with images in repository `object_detection_demo/test` directory.
"""


# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_CKPT = '/content/stemdata/frozen_inference_graph(3).pb'

# List of the strings that is used to add correct label for each box.
PATH_TO_LABELS = '/content/stemdata/saved_model.pbtxt'

# If you want to test the code with your images, just add images files to the PATH_TO_TEST_IMAGES_DIR.
PATH_TO_TEST_IMAGES_DIR = "object_detection_demo/data/test"
# pb_fname = 'stemdata/frozen_inference_graph(3).pb'
assert os.path.isfile(pb_fname)
assert os.path.isfile(PATH_TO_LABELS)
# TEST_IMAGE_PATHS = glob.glob("object_detection_demo/*.jpg")
# assert len(TEST_IMAGE_PATHS) > 0, 'No image found in `{}`.'.format(PATH_TO_TEST_IMAGES_DIR)
TEST_IMAGE_PATHS = ["/content/object_detection_demo/data/images/test/00000966.jpg",
"/content/object_detection_demo/data/images/test/00000973.jpg",
"/content/object_detection_demo/data/images/train/00000561.jpg",
"/content/object_detection_demo/data/images/train/00000560.jpg",
"/content/object_detection_demo/data/images/train/00000391.jpg",
"/content/object_detection_demo/data/images/train/00000560.jpg",
"/content/object_detection_demo/test/00000030.jpg"]
print(TEST_IMAGE_PATHS)

!ls /content/object_detection_demo/

BAZEL_VERSION = '0.20.0'
!wget https://github.com/bazelbuild/bazel/releases/download/{BAZEL_VERSION}/bazel-
{BAZEL_VERSION}-installer-linux-x86_64.sh

!chmod +x bazel-{BAZEL_VERSION}-installer-linux-x86_64.sh
!./bazel-{BAZEL_VERSION}-installer-linux-x86_64.sh
```

```
!bazel run -c opt tensorflow/contrib/lite/toco:toco -- \
--input_file=./frozen_inference_graph2.pb \
--output_file=./tflite/detect.tflite \
--input_shapes=1,300,300,3 \
--input_arrays=normalized_input_image_tensor \
--
output_arrays='TFLite_Detection_PostProcess','TFLite_Detection_PostProcess:1','TFLite_Detection_Pos
tProcess:2','TFLite_Detection_PostProcess:3' \
--inference_type=QUANTIZED_UINT8 \
--mean_values=128 \
--std_values=128 \
--change_concat_input_ranges=false \
--allow_custom_ops
```

```
!ls /content/models/research/fine_tuned_model_2_2_2_2
```

```
!ls /content/models/research/fine_tuned_model_2_2_2_2/
```

```
!zip -r /content/file.zip /content/models/research/fine_tuned_model_2_2_2_2/
```

```
files.download("/content/file.zip")
```

```
!pwd
```

```
!tflite_convert   --saved_model_dir=./fine_tuned_model_2_2_2_2/saved_model   --
output_file=/content/tflite/foo.tflite
```

```
!ls /content
```

```
# !pip install tensorflow==1.5.0
```

```
# Commented out IPython magic to ensure Python compatibility.
```

```
# %matplotlib inline
```

```
for j in TEST_IMAGE_PATHS:
  plt.imshow(cv2.imread(j)[::, ::, ::-1])
```

```
# Commented out IPython magic to ensure Python compatibility.
# %cd /content/models/research/object_detection
```

```
# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")
```

```
# This is needed to display the images.
```

```python
# %matplotlib inline



detection_graph = tf.Graph()
with detection_graph.as_default():
   od_graph_def = tf.GraphDef()
   with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
      serialized_graph = fid.read()
      od_graph_def.ParseFromString(serialized_graph)
      tf.import_graph_def(od_graph_def, name='')


label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(
   label_map, max_num_classes=num_classes, use_display_name=True)
category_index = label_map_util.create_category_index(categories)


def load_image_into_numpy_array(image):
   (im_width, im_height) = image.size
   return np.array(image.getdata()).reshape(
      (im_height, im_width, 3)).astype(np.uint8)

# Size, in inches, of the output images.
IMAGE_SIZE = (12, 8)


def run_inference_for_single_image(image, graph):
   with graph.as_default():
      with tf.Session() as sess:
         # Get handles to input and output tensors
         ops = tf.get_default_graph().get_operations()
         all_tensor_names = {
            output.name for op in ops for output in op.outputs}
         tensor_dict = {}
         for key in [
            'num_detections', 'detection_boxes', 'detection_scores',
            'detection_classes', 'detection_masks'
         ]:
            tensor_name = key + ':0'
            if tensor_name in all_tensor_names:
               tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(
                  tensor_name)
         if 'detection_masks' in tensor_dict:
            # The following processing is only for single image
            detection_boxes = tf.squeeze(
               tensor_dict['detection_boxes'], [0])
            detection_masks = tf.squeeze(
               tensor_dict['detection_masks'], [0])
            # Reframe is required to translate mask from box coordinates to image coordinates and fit the image
size.
            real_num_detection = tf.cast(
```

```python
                    tensor_dict['num_detections'][0], tf.int32)
                detection_boxes = tf.slice(detection_boxes, [0, 0], [
                            real_num_detection, -1])
                detection_masks = tf.slice(detection_masks, [0, 0, 0], [
                            real_num_detection, -1, -1])
                detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
                    detection_masks, detection_boxes, image.shape[0], image.shape[1])
                detection_masks_reframed = tf.cast(
                    tf.greater(detection_masks_reframed, 0.5), tf.uint8)
                # Follow the convention by adding back the batch dimension
                tensor_dict['detection_masks'] = tf.expand_dims(
                    detection_masks_reframed, 0)
            image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')

            # Run inference
            output_dict = sess.run(tensor_dict,
                        feed_dict={image_tensor: np.expand_dims(image, 0)})

            # all outputs are float32 numpy arrays, so convert types as appropriate
            output_dict['num_detections'] = int(
                output_dict['num_detections'][0])
            output_dict['detection_classes'] = output_dict[
                'detection_classes'][0].astype(np.uint8)
            output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
            output_dict['detection_scores'] = output_dict['detection_scores'][0]
            if 'detection_masks' in output_dict:
                output_dict['detection_masks'] = output_dict['detection_masks'][0]
    return output_dict


for image_path in TEST_IMAGE_PATHS:
    image = Image.open(image_path)
    # the array based representation of the image will be used later in order to prepare the
    # result image with boxes and labels on it.
    image_np = load_image_into_numpy_array(image)
    # Expand dimensions since the model expects images to have shape: [1, None, None, 3]
    image_np_expanded = np.expand_dims(image_np, axis=0)
    # Actual detection.
    output_dict = run_inference_for_single_image(image_np, detection_graph)
    # Visualization of the results of a detection.
    vis_util.visualize_boxes_and_labels_on_image_array(
        image_np,
        output_dict['detection_boxes'],
        output_dict['detection_classes'],
        output_dict['detection_scores'],
        category_index,
        instance_masks=output_dict.get('detection_masks'),
        use_normalized_coordinates=True,
        line_thickness=8)
    plt.figure(figsize=IMAGE_SIZE)
    plt.imshow(image_np)
```

```
!tflite_convert --output_file=ssd2.tflite --graph_def_file=frozen_inference_graph2.pb --input_arrays=input_13
--output_arrays=predictions/concat
```