Image Preprocessing and the Convolution Neural Network Model for Traffic Light Recognition

Chujun (Jerry) Xu
Piedmont Hills High School, San Jose, CA, USA
Email: xuchujun2672@gmail.com

***Abstract* – Machine learning is, as Arthur Samuel defined in 1959, a "field of study that gives computers the ability to learn without being explicitly programmed." Convolutional Neural Networks (CNN) are one way that machine learning can be applied to real life problems. They are used in everyday life in a variety of different situations, from clustering to classifying data. Though they began as inferior to humans, CNN's have recently achieved accuracy for image classifying higher than those of a human. By training a network on training data from the Bosch dataset, we were able to train a model to predict the colors of traffic lights to around 92% accuracy.**

I. Introduction

In general, machine learning can be split into two categories: supervised and unsupervised learning. Supervised learning is when the training occurs with data that has been labeled with the correct answer. In contrast, unsupervised learning's training data has no labels, and the model works on its own to discover information and patterns. This paper will focus on supervised learning and how we can train models to recognize traffic light colors based on annotated training data.

Most of machine learning is done through the programming language Python because of its concise and easily readable code. In addition, it has numerous libraries that can be used to build off of, such as Numpy, Matplotlib, to name a few. We will be using Pytorch as a framework for deep learning training and inference, since many of the convolution layers have been already defined in Pytorch.

We can write code for Convolutional Neural Networks, which consists of an input, output, and multiple hidden layers in between. Its name comes from the method we form a fully connected neural network: by "convolving" filters around the image, the program can find hyperparameter weights and bias for matrices such that it produces the desired results. These layers are called convolution layers. Each hidden layer is a different matrix function that filters the image, which is inputted as a tensor of RGB values.

The convolution layers include an activation function, which is usually the Rectified Linear Unit, or ReLU (Equation 1). This function is a piecewise function defined as this:

$$f(x) = \max(0, x) = \begin{cases} 0 & \text{if x} \leq 0 \\ x & \text{if x} > 0 \end{cases} \tag{1}$$

ReLU is usually used as an activation function instead of other ones such as the sigmoid function, $y = 1/(1+e^{-x})$ because it will not become "saturated" once the x-values are too large or too small. In addition, ReLU is much more computationally efficient, since it is a simple, linear piecewise function instead of an exponential function.

Usually following the convolution layers are pooling layers, which help cut down on the size of the information. By having layers such as max pooling layers, the speed of the network is maintained. Groups of convolution layers and pooling layers are called modules.

At the end of the CNN are usually fully connected layers, which compile all the information that the convolution layers have calculated and condense the tensors of numbers into the output, a matrix of the same size as the number of labels. Each number represents the probability that the network thinks the input is each of the labels. The highest number's location is thus the predicted label.

During the training process, the weights of the convolution layers are intialized at random. The data is split into three parts: training, validation, and test. In my project, there were 4898 images in training, 1225 in validation, and 2941 in test. The dataset I downloaded it from had test and training, so I split the training data into 80% and 20% for training and validation. Both training and validation should be annotated so as to show the correct labels, also known as the ground truth. After the training data is passed in, the "loss" is calculated, which shows how accurate the model is. The loss function that I used was the cross entropy loss, also known as softmax loss. This normalizes the end outputs to probabilities that are all positive and add up to one. To calculate loss (Equation 2), this is the function used, where s_{y_i} is the probability of the correct label, and j is the total number of categories:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\Sigma_j e^{s_j}}\right) \tag{2}$$

In this function, the negative log of the probability of the correct label is found, which is higher for lower probabilities and lower for higher probabilities. Since at intialization, all the weights will approximately equal, the loss should be around ln 3, or 1.099. To decrease the loss and thus increase the accuracy, backpropagation must be performance. By marking the algebraic functions that the data goes through in the convolution layers, we can use the chain rule to find the effects of each of the weights, which is expressed as gradients (partial derivative of loss) with respect to each of the weights. The average of all the losses is known as the cost function.

Though it is very tedious to calculate by hand, Pytorch, a Python library, has built-in functions that can easily calculate and track the gradients to perform backpropagations, the opposite of forward pass. We can just call one of the functions to calculate all the partial derivatives and use it to update the weights, using the learning rate, □ (Equation 3).

$$W_j = W_j - \alpha \frac{\partial}{\partial W_j} J(W_0, W_i, ..., W_n) \tag{3}$$

All the weights are updated simultaneously, and the CNN is ready for another image to be inputted in again. The learning rate can vary, leading to different speeds of the loss converging. Usually, the input images are loaded in batches (Figure 1) limited by hardware memory so that the training process can speed up. In addition, it is better than single updates since it is less noisy.

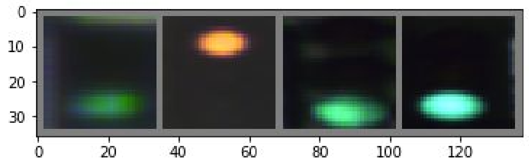


Figure 1: Example of a normalized batch

Each epoch runs through all the data and processes it. Most CNN's train on multiple epochs, to fine tune the categorization.

II. Materials and Methodology

For this project, I used Python 3.7 as the programming language, mainly due to the ease of writing readable and clear code, as well as the plethora of libraries and references already built. I utilized the following libraries and packages:

- Matplotlib
- PIL
- Numpy
- Pytorch

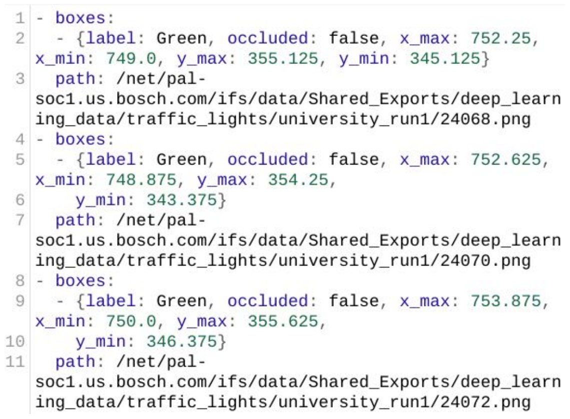
All lines of code can be found on my Github [1].

A. *Data Visualization*

For my project, I used data from the online Bosch dataset [2]. This dataset is a public compilation of annotated traffic lights. I used the "rgb train" and "rgb test" images. Due to the training images having errors in labeling, I decided to swap them, using the test images to train my CNN and the train images to test my CNN. I used 8334 images to train (Figure 2) and 5093 images to test. All of the images had a resolution of 1280 x 720 pixels.

Figure 2: A sample image from the Bosch dataset, training data

After the data was downloaded in separate zip files, I unzipped them using the file archiver 7-Zip Utility. The data was annotated with a YAML file (Figure 3), which has the boxes, whether it is occluded, the label, and the file name:



```
- boxes:
  - {label: Green, occluded: false, x_max: 752.25, x_min: 749.0, y_max: 355.125, y_min: 345.125}
  path: /net/pal-soc1.us.bosch.com/ifs/data/Shared_Exports/deep_learning_data/traffic_lights/university_run1/24068.png
- boxes:
  - {label: Green, occluded: false, x_max: 752.625, x_min: 748.875, y_max: 354.25,
    y_min: 343.375}
  path: /net/pal-soc1.us.bosch.com/ifs/data/Shared_Exports/deep_learning_data/traffic_lights/university_run1/24070.png
- boxes:
  - {label: Green, occluded: false, x_max: 753.875, x_min: 750.0, y_max: 355.625,
    y_min: 346.375}
  path: /net/pal-soc1.us.bosch.com/ifs/data/Shared_Exports/deep_learning_data/traffic_lights/university_run1/24072.png
```

Figure 3: First few entries in train.yaml

To visualize where the lights were, the original images were annotated (Figure 4) so that the traffic lights were boxed with the color that was labeled. We will use the same technique later to show the test results.

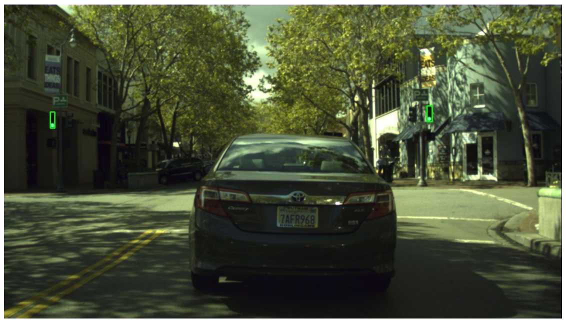


Figure 4: Annotated version of Figure 2

B. *Preprocessing*

The traffic lights have to be cropped (Figure 5) from the example images using the bounding boxes specified in the YAML file, in addition to being separated into folders for each of the labels: red, green, and yellow. Since some of the traffic lights were so small that a human eye wouldn't be able to identify the color, I filtered out any that were smaller than 10 pixels tall and 5 pixels wide. In addition, many of the tougher cases such as off or occluded were removed to make the CNN simpler.

Figure 5: A cropped green light (the left one) from Figure 2

Using the processed train and test data, graphs were compiled (Figures 6 and 7). The test data had very similar distributions as the train data. As expected, green had the most cases, followed by red and finally yellow. The sizes of the lights were also very different, from less than 10 to over 1000 pixels.

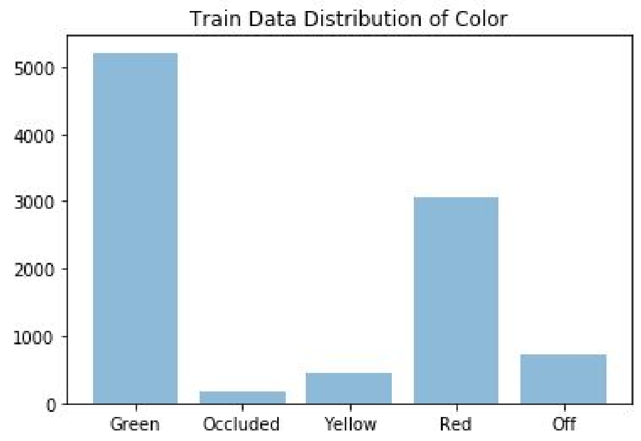


Figure 6: Uneven distribution in training data

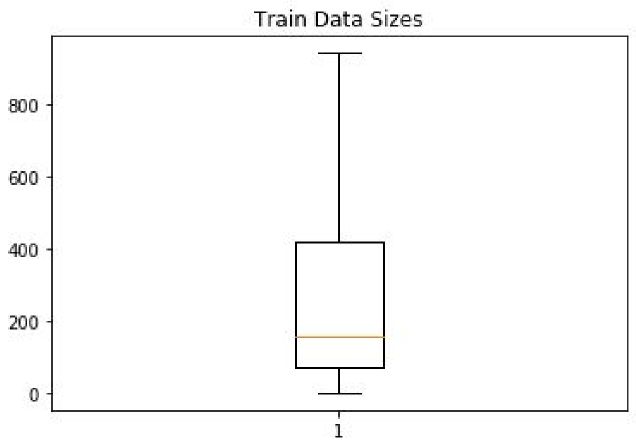


Figure 7: Range of sizes (in pixels) of training data

The cropped images are read into the model using the function torchvision.datasets.ImageFolder() and normalized and resized so that they are all 32 by 32 pixels with the range between 0 and 1.

Every time the batch is read in, the original input list gets shuffled around to ensure the order of the original input data is random. This is why running the model over and over again could produce slightly different losses and different accuracies, but eventually should converge to the same global minimum.

C. *Training*

The Convolutional Neural Network model architecture used (Figure 10) was similar to a CIFAR-10 tutorial [3], with 8 layers in total. The last fully connected layer was modified to produce three numbers instead of 10:

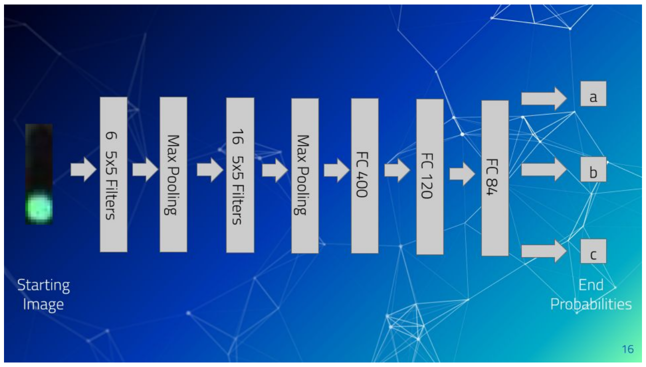


Figure 10: The CNN architecture used

Validation data was used to prevent overfitting and fine tune the hyperparameters, when our model fits the training data too well but does not do well on any other data. The loss was calculated with cross entropy loss. Also, the model used a decay learning rate, starting at 0.001 and decaying by ⅛ every time the change in loss after an epoch is less than 5 times the learning rate. Figure 11 shows the training versus validation loss, where we can see that the model was not overfitting, since the validation was always below the training loss. Note the log scaled y-axis.

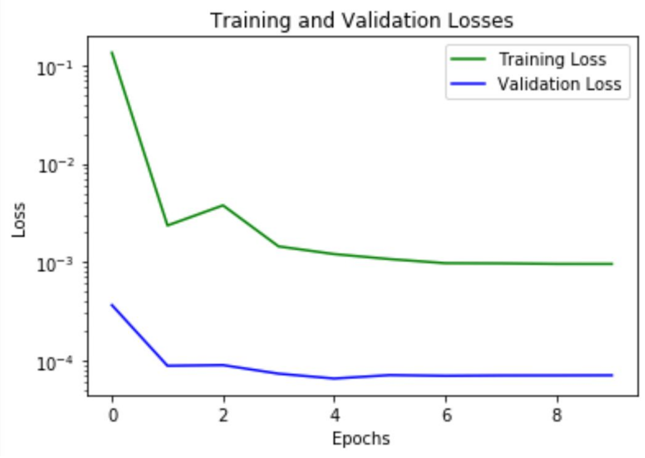


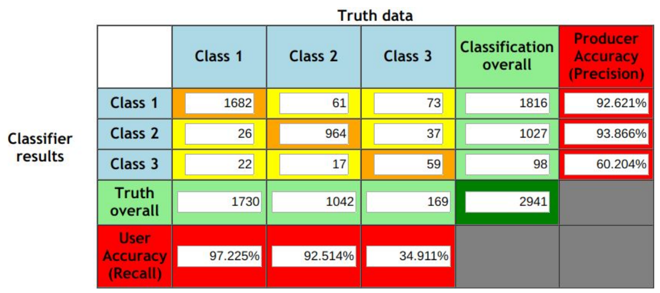
Figure 11: Training versus validation loss after 10 epochs

The model went through the data for a total of 10 epochs, ending with a loss of around 0.001 for test data. We can save the model using torch.save, so that once we train once, we can use the same model to predict multiple datasets.

III. Results and Discussions

After running all the data back in, the model predicted 4897/4898 of the training data, 1225/1225 of the validation data, and 2705/2941 images of the test data. In terms of percentages, the model had an accuracy of 99% for training, 100% for validation, and 91% for test.

Within the test data, the green lights were classified as green 1682/1730, the red 964/1042, and yellow 59/169. This meant that green had an accuracy of 97%, red had an accuracy of 92%, and yellow had 34% (Figure 12).



		Truth data				
		Class 1	Class 2	Class 3	Classification overall	Producer Accuracy (Precision)
Classifier results	Class 1	1682	61	73	1816	92.621%
	Class 2	26	964	37	1027	93.866%
	Class 3	22	17	59	98	60.204%
	Truth overall	1730	1042	169	2941	
	User Accuracy (Recall)	97.225%	92.514%	34.911%		

Figure 12: Confusion matrix of the results
(Class 1 = green, Class 2 = red, Class 3 = yellow)

As expected, green had the highest accuracy and yellow the lowest. This is parallel to the amount of training data we gave the model. Many yellow traffic lights were classified as green, probably due to the close proximity and how close the colors seem from far away.

Despite the low accuracy for yellow, it is still higher than 33%, which is randomly guessing. This means that the CNN is actually learning what the traffic lights look light and it is highly unlikely that this result was due to random chance.

Another possible explanation for some of the inaccuracies is that the ground truth labels might have been wrong for a couple of them, because it is easy for even humans to make mistakes identifying traffic lights that are small and far away.

Using similar lines of code from when we annotated the training data with the ground truth, we also annotated the test data with the labels that the CNN predicted. Using these images, we can generate a video that shows how a CNN of this type can be used in real life to identify traffic lights, even the ones that a naked human eye might not be able to [4].

IV. Future Work

This model is far from perfect and usable on future self-driving cars or other technology. First, it can train on a much larger set of training data, with more epochs. This way, the loss can be lower and the model more accurate. Data augmentation can be used to increase the number of data for yellow, since it will most likely always have the least number of data [5]. There are many ways to augment data, such as cropping, rotating, shifting, etc. but we will not be able to change the color or orientation of the image - only flipping along a vertical axis. The architecture can also be made more complicated, like the architecture of AlexNet or even ResNet [6].

Another expansion can be to incorporate other types of traffic lights and not remove the occluded or off ones. One example is arrows. With left, forward, and right arrows of each of the three colors, the number of categories will increase dramatically and allow CNN to be used in more situations involving different traffic lights. This should not be very difficult other than finding enough training data for the model to be able to accurately predict.

This entire CNN has been based off of already cropped images. In the future, it can be modified to take in not just the already cropped images, but the entire image, like from a car's dash camera. This would add traffic light detection before the image classification problem, but would open the code to almost any image in real life, which doesn't have to be annotated.

After it incorporates the traffic light detector, hardware can be created for the project, so that there can be a monitor that can be attached to a dashcam, predicting the traffic lights ahead in real time. This has direct application on self driving cars, visually impaired people who want to drive, and even just driving under terrible visibility conditions.

V. Acknowledgement

I would like to thank Alex Cui for helping guide me throughout this project and answering all my questions about the topics. He has provided me with much guidance and support.

VI. References

[1] Github, https://github.com/xu-jerry/trafficlight-project

[2] Bosch Dataset, https://hci.iwr.uni-heidelberg.de/node/6132

[3] CIFAR-10 tutorial architecture, https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py

[4] Traffic Light Project: Stop Motion Annotation Example, https://www.youtube.com/watch?v=koKBcn6yvxY

[5] Smart Augmentation Learning an Optimal Data Augmentation Strategy, https://ieeexplore.ieee.org/abstract/document/7906545

[6] Computer Vision: A Study On Different CNN Architectures and their Applications, https://medium.com/alumnaiacademy/introduction-to-computer-vision-4fc2a2ba9dc