**Traffic Prediction Project**

Note: Instructions on how to run the project is down below

Github repository: <https://github.com/jacksondelametter/Traffic_Prediction.git>

**Topic**

My project’s function is to predict traffic levels from a vehicle’s perspective. I do this using a CNN architecture on a set of images that predict if there is low or medium traffic. The GUI for this project displays images from a vehicle’s perspective of what is in front of it. The trained model then predicts the traffic level given the image.

**Intuition and Design**

Most traffic notifying technologies today use GPS to determine traffic level. What is not used is a neural network to handle this function. An interesting idea would be for a vehicle to use computer vision in order to determine the traffic level. That vehicle could then send that information to notify other vehicles of the current traffic level.

My goal for this project was to use a vehicle’s vision to determine if the traffic is low, medium, or high. I also aimed to use metrics such as vehicle count, position, and closeness to determine the traffic level. I was successful in implementing all of the metrics, but ran into a few issues along my research.

An important find that I realized was that it was difficult to categorize the traffic level into the three categories stated above. The reason for this was the vehicle’s range of view (A vehicle can only see so much). Because of this, I found that getting rid of the high traffic category was sufficient.

Each vehicle only looks at other vehicles that are relatively close and in the same drivable area. This ensures that vehicles that are far away do not get incorporated in the current traffic level. It also ensures that vehicles parked or going in the opposite direction also do not get incorporated in the current traffic level. This labeling scheme though is not perfect. It cannot always determine if a car parked on the side of the street is actually parked or in traffic.

Although the labeling of images is very good, it is not perfect. A mode has been added into the GUI to label images yourself rather than relying on the automated labeling process of the program. The model is trained using these labels so it may predict a traffic level when the label is clearly wrong. In some cases, though, the trained model outperforms the labeling system. I urge you to try this mode to see its full potential.

**Dataset**

The dataset I am using is from the Berkley Deep Drive. It consists of over 100,000 images of vehicle windshield views. The link to the dataset is given below.

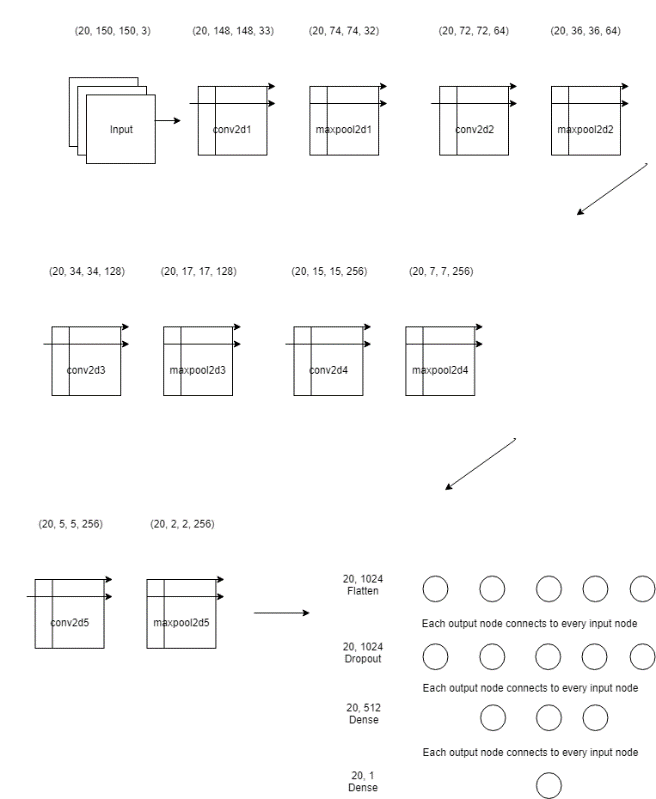
Link: <https://bdd-data.berkeley.edu/>

Each image is also labeled. Labeled information includes, vehicle count, position, and drivable areas.

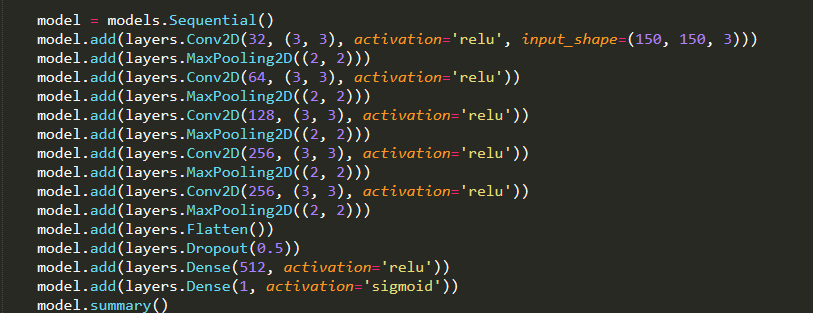


**Network Design**

My design makes use of the CNN architecture. The network architecture is given below. On top of each layer represents the output tensor.



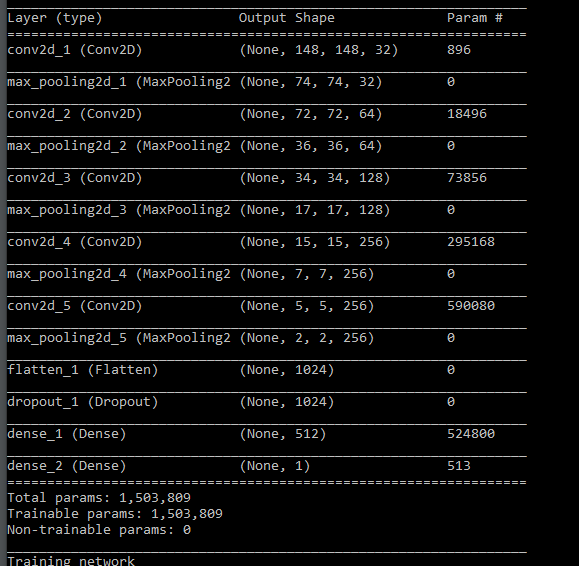
The code of the network is also given below.



My network consists of five sets of Conv2D and MaxPooling2D layers. A flatten layer is then used to flatten the output of these layers and then fed into a Dense layer. Dropout is applied to the end to fight overfitting.

**Tensor Shapes**

Below is a summary of the tensor shapes of each layer



The input to the network is a tensor of shape (20, 150, 150, 3) where the first dimension is the batch size. The figure above gives the shape of the layers between the first and the last layers. The Conv2D layers reduces each input image size by roughly two pixels (from the padding). The MaxPooling2D layers reduce each input image size by roughly half. The number of extracted features increase as the model gets deeper. This number goes from 32 to 256. After the flattening of the batch of images. The output of the network is a tensor of shape (20, 1). The single output for each image is the probability that the traffic from the car’s perspective is low or medium.

**Hyperparameters**

The hyperparameters chosen to tune included batch size, epoch number, and dropout rate. For batch size, I just happened to pick the optimum number (which is 20). I tried values ranging from 15-50 but most of these numbers resulted in worse test results.

For epoch number, I initially chose an epoch of 30. Although this is a relatively large number, it allowed me to see the overfitting clearly anytime I changed another hyperparameter or pre-processing attribute. After I was done changing other factors, I found that the optimum epoch number came to be 16.

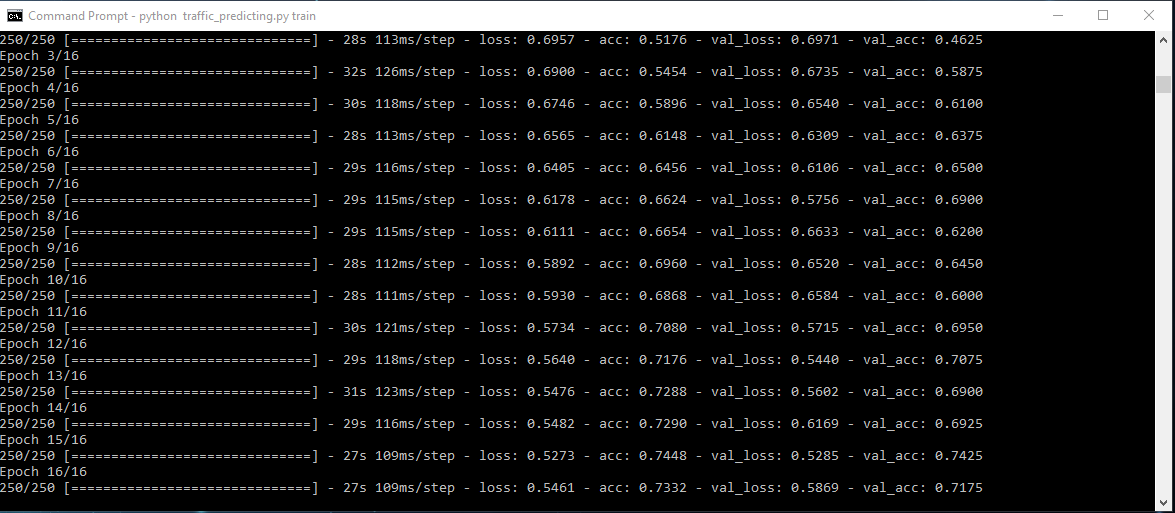
For dropout rate, I found the optimum number to be 0.5. The only other number I tried was 0.2. I found that dropping the rate resulted in more overfitting.

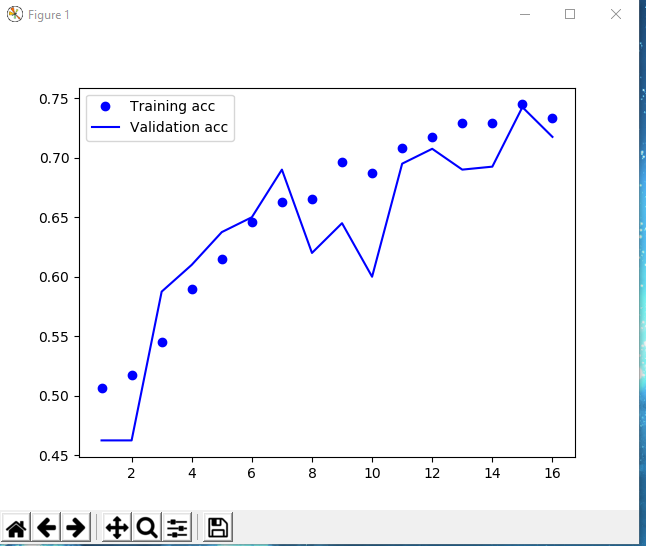
**Pre-Processing**

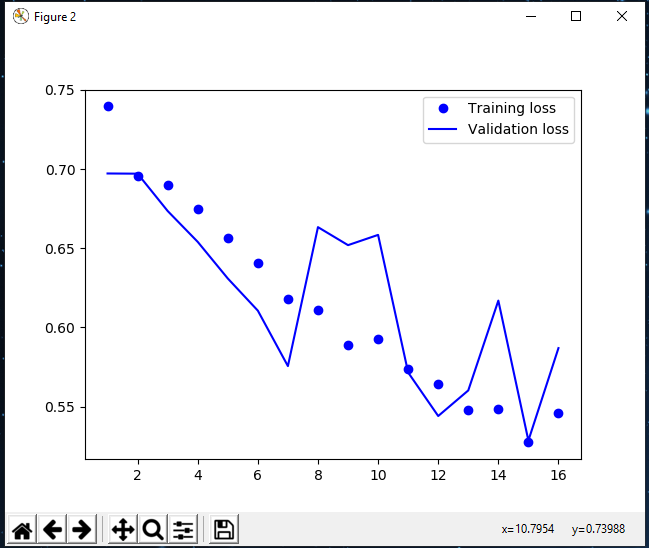
Although the model architecture and tuning of the hyperparameters were important to increase performance, I found that tuning the way images were pre-processed gave the biggest performance increases and gave more correctness to the labeling of images. Some tuning attributes included vehicle closeness and drivable area locations. I found that by tuning the vehicle closeness, I was able to achieve over a five percent increase. This is because the model was getting confused about vehicles being far in the distance. By determine parking areas on an image, I was able to modify drivable areas on an image. This also gave roughly a 5-10 percent increase. These modifications not only improved the model’s accuracy, but gave more correctness to each image being processed. Adding more data also increased the performance. I did however find that after training on 10,000 images, the performance did not increase by much.

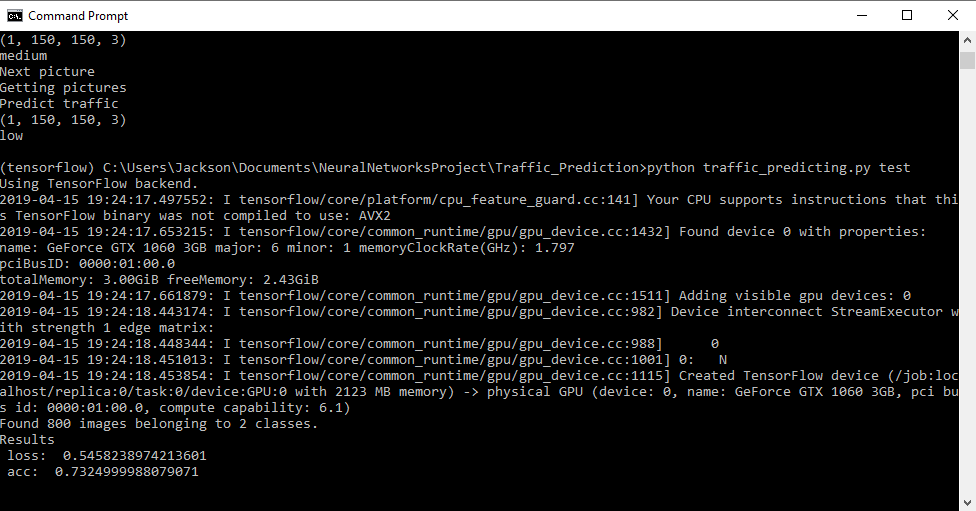
**Training and Testing Performance**

I started off with a baseline model with a test accuracy of around 50%. After tuning the model, hyperparameters, and pre-processing attributes, I was able to achieve an test accuracy of nearly 73%.

****







**Installation Instructions**

I have tried to make the installation process as painless as possible (Unfortunately there is only so much I can do). If you have any issues with the installation process, email me at delamet@tamu.edu

**Downloading the Dataset**

Unfortunately, I am not able to include a portion of my dataset in the github repository. I have detailed instructions on how to download the dataset (I have talked to Dr. Jiang about this and said to leave instructions on how to download it).

1. Follow the link <https://bdd-data.berkeley.edu/>
2. Scroll to the bottom and click on the “Download Dataset” button
3. Login using the following credentials (Although this is my tamu email, the password is not the same as my netid password). Email: [delamet@tamu.edu](mailto:delamet@tamu.edu) Password: Password1234. Alternatively, you can create an account yourself and login.
4. Click on the Downloads tab on the left
5. Under the BDD100K dropdown, click on the “Images” and “Labels” button to start downloading. (Note: these are both large downloads)

**Environment Setup**

I developed and tested this project using both windows and ubuntu while in an Anaconda virtual environment. Although it may work for other types of virtual environments, it is recommended to use Anaconda.

1. Create a new folder and change to that directory– mkdir traffic\_pred, cd traffic\_pred
2. Create a new folder – mkdir images (Note: you can put any vehicle images you like in this folder, do not change into this directory)
3. Unzip the images and labels download into the traffic\_pred directory. Depending on the unzipping tool, you may get different unzipped paths, make sure the paths of both the images and labels start with the following directory structures, images: \bdd100k\_images\bdd100k\images\...., labels: \bdd100k\_labels\_release\bdd100k\labels\...
4. Download the project from the github repository – git clone <https://github.com/jacksondelametter/Traffic_Prediction.git> (Note: this should put the project into a folder called Traffic\_Prediction)
5. Install Anaconda, issue the following commands to create the virtual environment and install all dependencies
6. conda create -n vm python=3.6.6
7. conda activate vm
8. pip install tensorflow
9. pip install keras
10. pip install Pillow
11. pip install matplotlib

**Pre-Processing Images**

Before you can run the GUI on the pre-trained model or train a new model, you first must process the downloaded images.

1. Make sure you are in the directory traffic\_pred
2. Change to project src code – cd Traffic\_Prediction
3. Run the pre-processing – python traffic\_predicting.py process
4. Let the program run, in the previous directory. Once completed you should see new directories train, test, val, and gui with images inside of them. (Note: If you get an error when the program is starting stating you don’t have permission to create the directories, try running it again)
5. Note: If you encounter any other type of error, then you did not unzip the images and labels correctly

**GUI Use**

The directory Traffic\_Prediction comes with a model and weights in order to run the GUI without training the model. My GUI shows the traffic prediction for each image. The image is shown to the left. Pressing the “Predict” button displays the prediction results from the trained model. You can go the next image by hitting the “next” button.

My GUI has two modes of operation, labeled\_images mode displays the pre-processed images in the gui folder (Note: these are not the same images used for training, validating, or testing). Hitting the predict button predicts the traffic level, shows the labeled answer, and shows the accuracy of the predicted images, non\_labeled\_images displays non pre-processed images. Hitting the predict button only predicts the traffic level. Below show instructions on how to run the GUI

1. Make sure you are in the directory traffic\_pred
2. Change to the directory Traffic\_Prediction – cd Traffic\_Prediction
3. For labeled\_images mode – python GUI.py labeled\_images (You should see the GUI appear)
4. For non\_labeled\_images mode – python GUI.py non\_labeled\_images images (Note: the second argument is the directory with the images you want to display relative to the previous directory. In the previous directory you should have created a directory called images and placed images in that directory)
5. Video for how to use the GUI is included in the github repository

**Training and Testing**

If you want to train and test a new model, follow the instructions below. (Note: you must have pre-processed the data first)

1. Make sure you are in the directory traffic\_pred
2. Change to the directory Traffic\_Prediction – cd Traffic\_Prediction
3. To train the model – python traffic\_predicting train (I trained using a GPU, your training time could be significantly longer)
4. To test the model – python traffic\_prediction test

**Annotated Code**

# traffic\_predicting.py

from keras import backend as K

import shutil

from tensorflow.python.client import device\_lib

from keras.preprocessing.image import ImageDataGenerator

import os

import json

from keras import layers

from keras import models

import matplotlib.pyplot as plt

from random import sample

from keras import regularizers

from keras.models import model\_from\_json

import sys

# Sets up all the directory paths that will be created in procesing

os.chdir('..')

current\_dir = os.getcwd()

images\_path = os.path.join(current\_dir, 'bdd100k\_images')

images\_path = os.path.join(images\_path, 'bdd100k')

images\_path = os.path.join(images\_path, 'images')

images\_path = os.path.join(images\_path, '100k')

train\_images\_path = os.path.join(images\_path, 'train')

test\_images\_path = os.path.join(images\_path, 'test')

val\_images\_path = os.path.join(images\_path, 'val')

labels\_path = os.path.join(current\_dir, 'bdd100k\_labels\_release')

labels\_path = os.path.join(labels\_path, 'bdd100k')

labels\_path = os.path.join(labels\_path, 'labels')

train\_labels\_path = os.path.join(labels\_path, 'bdd100k\_labels\_images\_train.json')

val\_labels\_path = os.path.join(labels\_path, 'bdd100k\_labels\_images\_val.json')

train\_dir = os.path.join(current\_dir, 'train')

test\_dir = os.path.join(current\_dir, 'test')

val\_dir = os.path.join(current\_dir, 'val')

gui\_dir = os.path.join(current\_dir, 'gui')

src\_path = os.path.join(current\_dir, 'Traffic\_Prediction')

# Variables used to give the number of images the train, test, val and gui directories

train\_dir\_size = 50000

val\_dir\_size = 4500

test\_dir\_size = 4500

gui\_dir\_size = 1000

train\_size = 5000

val\_size = 400

test\_size = 400

gui\_size = 100

# Used to indicate the batch number for taining and testing

batch\_no = 20;

def preprocess():

'''

preprocesses images into low and medium categories in directories train, test, val, and gui

'''

# Removes and creates these directories

print("Making train, test, and val directories")

make\_category\_dirs(train\_dir)

make\_category\_dirs(test\_dir)

make\_category\_dirs(val\_dir)

make\_category\_dirs(gui\_dir)

# Gets the images names in the labels files for adding them into the directories stated above

# Note the variables at the top of the file determine the sizes for each directory

train\_labels = get\_labels\_file(train\_labels\_path)

temp\_val\_labels = get\_labels\_file(val\_labels\_path)

val\_labels = temp\_val\_labels[0:val\_dir\_size]

test\_labels\_limit = val\_dir\_size + test\_dir\_size

test\_labels = temp\_val\_labels[val\_dir\_size:test\_labels\_limit]

gui\_labels\_limit = test\_labels\_limit + gui\_dir\_size

gui\_labels = temp\_val\_labels[test\_labels\_limit:gui\_labels\_limit]

# Categorizes pictures and adds then into the specified directory

print("Categorizing train, test, val, and gui directories");

categorize\_images(train\_labels, train\_dir, train\_size, train\_images\_path, train\_dir\_size)

categorize\_images(test\_labels, test\_dir, test\_size, val\_images\_path, test\_dir\_size)

categorize\_images(val\_labels, val\_dir, val\_size, val\_images\_path, val\_dir\_size)

categorize\_images(gui\_labels, gui\_dir, gui\_size, val\_images\_path, gui\_dir\_size)

# Used to get a label file from the specified directory

def get\_labels\_file(labels\_path):

file\_object = open(labels\_path)

if(file\_object == None):

print("Could not find labels file")

print('Found labels file')

print('Loading json files')

return json.load(file\_object)

# Creates the given directory

def make\_dir(dir):

if(os.path.exists(dir)):

shutil.rmtree(dir)

os.mkdir(dir)

print('Created director ', dir)

# Creates the given directory with the two categories in it

def make\_category\_dirs(dir):

make\_dir(dir)

low\_traffic\_dir = os.path.join(dir, 'low')

make\_dir(low\_traffic\_dir)

medium\_traffic\_dir = os.path.join(dir, 'medium')

make\_dir(medium\_traffic\_dir)

# Categorizes the given images from the labels and adds them into the given directory

def categorize\_images(labels, save\_dir, save\_dir\_size, images\_dir, image\_dir\_size):

low\_dir = os.path.join(save\_dir, 'low')

medium\_dir = os.path.join(save\_dir, 'medium')

low\_traffic\_max\_thresh = 4

low\_traffic\_num = 0

medium\_traffic\_num = 0

print('Randomizing labels')

randomozed\_labels = sample(labels, image\_dir\_size)

print('Categorizing images')

for image\_value in randomozed\_labels:

image\_name = image\_value['name']

image\_labels = image\_value['labels']

#print("image is ", image\_name)

drivable\_areas = getDrivableArea(image\_labels)

car\_count = 0;

for label in image\_labels:

if isVehicle(label, drivable\_areas):

car\_count = car\_count + 1

src = os.path.join(images\_dir, image\_name)

if low\_traffic\_num > save\_dir\_size or medium\_traffic\_num > save\_dir\_size:

break

if car\_count <= low\_traffic\_max\_thresh and low\_traffic\_num < save\_dir\_size:

shutil.copy(src, low\_dir)

low\_traffic\_num = low\_traffic\_num + 1

elif car\_count > low\_traffic\_max\_thresh and medium\_traffic\_num < save\_dir\_size:

shutil.copy(src, medium\_dir)

medium\_traffic\_num = medium\_traffic\_num + 1

print('Categorized ', save\_dir)

print('low traffic no: ', low\_traffic\_num)

print('medium traffic no: ', medium\_traffic\_num)

# Gets the driveable areas for an image

def getDrivableArea(image\_labels):

drivable\_areas = []

for label in image\_labels:

category = label['category']

if category == 'drivable area':

attr = label['attributes']

area\_type = attr['areaType']

poly2d = label['poly2d']

vertices = poly2d[0]

vertices = vertices['vertices']

drivable\_areas.append(vertices)

return drivable\_areas

# Determines if the given vehicle is in a drivable area

def inDrivableArea(drivable\_areas, box2d):

x1\_car = box2d['x1']

x2\_car = box2d['x2']

in\_left\_bounds = False

in\_right\_bounds = False

# Determines if car is in left hand lane

for area in drivable\_areas:

for vertex in area:

x\_vertex = vertex[0]

if x1\_car > x\_vertex and x2\_car > x\_vertex:

in\_left\_bounds = True

break

for area in drivable\_areas:

for vertex in area:

x\_vertex = vertex[0]

if x1\_car < x\_vertex and x2\_car < x\_vertex:

in\_right\_bounds = True

break

return in\_left\_bounds and in\_right\_bounds

# Determines if the given vehicle is close or not

def isClose(box2d):

y\_thresh = 60

y1 = box2d['y1']

y2 = box2d['y2']

y\_size = y2 - y1

if(y\_size <= y\_thresh):

return False

return True

# Determines if the given label is a vehicle or not

def isVehicle(label, drivable\_areas):

category = label['category']

close = False

drivable = False

if category == 'car' or category == 'truck' or category == 'bus':

box = label['box2d']

close = isClose(box)

drivable = inDrivableArea(drivable\_areas, box)

return close and drivable

return False

# Trains the network

def train\_network():

train\_datagen = ImageDataGenerator(rescale=1./255)

test\_datagen = ImageDataGenerator(rescale=1./255)

val\_datagen = ImageDataGenerator(rescale=1./255)

print('Creating image generators')

train\_generator = train\_datagen.flow\_from\_directory(train\_dir, target\_size=(150, 150), batch\_size=batch\_no, class\_mode='binary', shuffle=True)

val\_generator = val\_datagen.flow\_from\_directory(val\_dir, target\_size=(150, 150), batch\_size=batch\_no, class\_mode='binary', shuffle=True)

test\_generator = test\_datagen.flow\_from\_directory(test\_dir, target\_size=(150, 150), batch\_size=batch\_no, class\_mode='binary', shuffle=True)

test\_generator = test\_datagen.flow\_from\_directory(test\_dir, target\_size=(150, 150), batch\_size=batch\_no, class\_mode='binary', shuffle=True)

print('Creating network model')

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(150, 150, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(256, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(256, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())

model.add(layers.Dropout(0.5))

model.add(layers.Dense(512, activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))

model.summary()

model.compile(loss='binary\_crossentropy', optimizer='rmsprop', metrics=['acc'])

train\_epoch\_steps = train\_size / batch\_no

val\_epoch\_steps = val\_size / batch\_no

print("Training network")

history = model.fit\_generator(train\_generator, steps\_per\_epoch=train\_epoch\_steps, epochs=16, validation\_data=val\_generator, validation\_steps=val\_epoch\_steps)

# Saves the model and its trained weight

os.chdir(src\_path)

model\_json = model.to\_json()

with open("model.json", "w") as json\_file:

json\_file.write(model\_json)

model.save\_weights("model.h5")

test\_model()

# Displays the training results

acc = history.history['acc']

val\_acc = history.history['val\_acc']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val\_acc, 'b', label='Validation acc')

plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val\_loss, 'b', label='Validation loss')

plt.legend()

plt.show()

# Tests the model with the test data

def test\_model():

os.chdir(src\_path)

json\_file = open('model.json')

loaded\_model\_json = json\_file.read()

json\_file.close()

model = model\_from\_json(loaded\_model\_json)

model.load\_weights('model.h5')

test\_datagen = ImageDataGenerator(rescale=1./255)

test\_generator = test\_datagen.flow\_from\_directory(test\_dir, target\_size=(150, 150), batch\_size=batch\_no, class\_mode='binary', shuffle=True)

test\_epoch\_steps = test\_size / batch\_no

model.compile(loss='binary\_crossentropy', optimizer='rmsprop', metrics=['acc'])

results = model.evaluate\_generator(generator=test\_generator, steps=test\_epoch\_steps)

print('Results\n loss: ', results[0], '\n', 'acc: ', results[1], '\n')

return results

# Program starts here

if len(sys.argv) == 0:

print('Must have mode argument: preprocess or train')

command = sys.argv[1]

if command == 'process':

preprocess()

elif command == 'train':

train\_network()

elif command == 'test':

test\_model()

else:

print('Invalid argument')

# GUI.py

import os

from tkinter import \*

import matplotlib.image as image

from PIL import ImageTk, Image

import matplotlib.pyplot as plt

import sys

from random import sample

from keras.models import model\_from\_json

import keras

import numpy as np

from keras.preprocessing import image

# Gets the model and weights

os.chdir('..')

current\_dir = os.getcwd()

os.chdir('Traffic\_Prediction')

json\_file = open('model.json')

loaded\_model\_json = json\_file.read()

json\_file.close()

model = model\_from\_json(loaded\_model\_json)

model.load\_weights('model.h5')

print('Loaded model with weights')

# Class used to make gui

class main:

def \_\_init\_\_(self, master, mode):

# Setups gui for both labeled and non labeled mode

self.master = master

self.mode = mode

if mode == 'labeled\_images':

self.get\_pictures()

else:

self.get\_non\_labeled\_pics()

self.picture\_frame = Frame(self.master, padx=5, pady=5)

self.picture\_label = Label(self.picture\_frame)

self.picture\_label.pack()

self.picture\_frame.pack(side=LEFT)

info\_frame = Frame(self.master, padx=5, pady=5)

Label(info\_frame,text="Traffic Predicting",fg="black",font=("",20,"bold")).pack(pady=10)

self.prediction\_label = Label(info\_frame,text="Traffic Prediction: None",fg="blue",font=("",20,"bold"))

self.answer\_label = Label(info\_frame,text="Traffic Answer: None",fg="blue",font=("",20,"bold"))

self.acc\_label = Label(info\_frame,text="Acc: None",fg="blue",font=("",20,"bold"))

self.setup\_acc()

self.prediction\_label.pack(pady=20)

if mode == 'labeled\_images':

# If labeled mode, includes answer and accuracy in gui

self.answer\_label.pack(pady=20)

self.acc\_label.pack(pady=20)

self.next\_picture()

arrow\_frame = Frame(info\_frame, pady=20)

self.next\_button = Button(arrow\_frame,font=("",10),fg="white",bg="red", text="Next", command=self.next\_picture)

self.next\_button.pack(side=RIGHT)

arrow\_frame.pack(side=BOTTOM)

Button(info\_frame,font=("",15),fg="white",bg="red", text="Predict", command=self.predict\_traffic).pack(side=BOTTOM)

info\_frame.pack(side=RIGHT,fill=Y)

# Setup for accuraccy metrics

def setup\_acc(self):

self.correct\_preds = 0;

self.total\_preds = 0;

# Predict button was pressed, use model and weight to predict traffic

def predict\_traffic(self):

print('Predict traffic')

label = self.picture\_dic[self.current\_pic]

if self.mode == 'labeled\_images':

pic\_path = os.path.join(pic\_dir, label)

pic\_path = os.path.join(pic\_path, self.current\_pic)

else:

pic\_path = os.path.join(pic\_dir, self.current\_pic)

pic = image.load\_img(pic\_path, target\_size=(150, 150))

pic\_array = image.img\_to\_array(pic)

pic\_array = pic\_array / 255

img = np.expand\_dims(pic\_array, axis=0)

print(img.shape)

result = model.predict\_classes(img)

prediction = result[0]

if prediction == 0:

prediction = 'low'

else:

prediction = 'medium'

print(prediction)

if prediction == label:

self.correct\_preds = self.correct\_preds + 1

self.total\_preds = self.total\_preds + 1

self.prediction\_label['text'] = 'Traffic Prediction: {}'.format(prediction)

self.answer\_label['text'] = 'Traffic Answer: ' + label

self.acc\_label['text'] = 'Acc: ' + str((self.correct\_preds / self.total\_preds \* 100)) + '%'

# Sets the next picture in line to the picture on the gui

def next\_picture(self):

print('Next picture')

if self.picture\_index < len(self.picture\_list) - 1:

self.picture\_index += 1

self.prediction\_label['text'] = "Traffic Prediction: None"

self.answer\_label['text'] = "Traffic Answer: None"

img = self.get\_next\_picture()

self.picture\_label.configure(image=img)

self.picture\_label.image = img

# Helper method used by next\_picture(), returns the next picture in line

def get\_next\_picture(self):

print('Getting pictures')

self.current\_pic = self.picture\_list[self.picture\_index]

picture\_label = self.picture\_dic[self.current\_pic]

if self.mode == 'labeled\_images':

picture\_path = os.path.join(pic\_dir, picture\_label)

picture\_path = os.path.join(picture\_path, self.current\_pic)

else:

picture\_path = os.path.join(pic\_dir, self.current\_pic)

img = ImageTk.PhotoImage(file=picture\_path)

return img

# Gets all the pictures in the gui folder, called in labeled\_mode

def get\_pictures(self):

low\_dir = os.path.join(pic\_dir, 'low')

medium\_dir = os.path.join(pic\_dir, 'medium')

pics = {}

low\_pics = self.get\_pictures\_from\_dir(low\_dir, pics, 'low')

medium\_pics = self.get\_pictures\_from\_dir(medium\_dir, pics, 'medium')

self.picture\_dic = pics

self.picture\_list = sample(pics.keys(), len(pics.keys()))

self.picture\_index = -1

# Adds all the pictures names in the given directory to the dictionary pics

def get\_pictures\_from\_dir(self, dir, pics, label):

count = 0

for pic\_name in os.listdir(dir):

if count != 100:

pics[pic\_name] = label

# Gets all non labeled pics, called in non\_labeled\_mode

def get\_non\_labeled\_pics(self):

pics = {}

self.get\_pictures\_from\_dir(pic\_dir, pics, 'none')

self.picture\_dic = pics

self.picture\_list = sample(pics.keys(), len(pics.keys()))

self.picture\_index = -1

# Program starts here

if len(sys.argv) == 1:

print('Enter mode: labeled\_images or non\_labeled\_images')

print("For labeled\_images mode, use argument labeled\_images, Demo will use processed images")

print('For non\_labeled images, use argument non\_labeled\_images followed by directory relative to previous directory')

sys.exit()

mode = sys.argv[1]

if mode == 'labeled\_images':

pic\_dir = os.path.join(current\_dir, 'gui')

elif mode == 'non\_labeled\_images':

pic\_dir = os.path.join(current\_dir, sys.argv[2])

else:

print('Enter a valid mode')

root = Tk()

main(root, mode)

root.title('Traffic detector')

root.resizable(0, 0)

root.mainloop()