Code uses Keras to implement CNN for training the Vehicle model

*Keras is a high-level neural networks API, written in Python and capable of running on top TensorFlow, CNTK, or Theano.*

Our code uses TensorFlow as backend.

The core data structure of Keras is a **model**, a way to organize layers. The simplest type of model is the Sequential model, a linear stack of layers.

# model = Sequential()

.add() is used for stacking the layers

The first layer in our design will be the Lambda layer to center and normalize the data (pixel values) from the input image.

model.add(Lambda(lambda x: x / 255., input\_shape=inputShape, output\_shape=inputShape))

where the input shape is set to (64, 64, 3) since RGB images

The second block consists of the con

# Block 0

model.add(Conv2D(filters=16, kernel\_size= volution layer which consists of filters (16 filters) and a kernel size of (4,4)

The Activation function used here is ‘relu’

f ( x ) = max ( 0 , x ) ,

(4, 4), activation='relu', name='cv0',

input\_shape=inputShape, padding="same"))

Dropout can be applied to the input to prevent overfitting

Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.

# model.add(Dropout(0.5)) #Dropout to reduce overfitting

Multiple Layers of Convolution filters can be added

# Block 1

#model.add(Conv2D(filters=32, kernel\_size=(3, 3), activation='relu', name='cv1', padding="same"))

#model.add(Dropout(0.5))

# block 2

#model.add(Conv2D(filters=64, kernel\_size=(3, 3), activation='relu', name='cv2', padding="same"))

One Max pooling Layer that performs Max Pooling on the resultant activation maps

The Pool size being (8,8)

'''Max Pooling'''

model.add(MaxPooling2D(pool\_size=(8, 8)))

# model.add(Dropout(0.5))

The resultant final matrix is flattened and fed to the hidden layers

model.add(Flatten())

One hidden dense layer of size 128

model.add(Dense(output\_dim=128,activation='relu')) #alternative1

One output layer of one node and activation Sigmoid

model.add(Dense(output\_dim=1,activation='sigmoid')) #alternative2

Instead of the Dense layer, One final Conv2D filter of size (8,8) is used with the sigmoid activation to classify the images (binary classifier)

# binary 'classifier' Fully connected layer

#model.add(Conv2D(filters=1, kernel\_size=(8, 8), name='fcn', activation="sigmoid"))

Adam optimizer used as stochastic gradient descent , Mean Sqaure Error as loss function, Performance metric is accuracy

Binary Cross entropy function can also be used as a loss function in binary classification problems

#model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])

Images obtained from the GTI Vehicle Images Data base and also from the Kitti Vision Bench Mark Suite.

Data preprocsseing is done using the glob and sklearn modules to access the images in the folders, assign labels and shuffle them. Splitting them into the Testset, Validation set and Training set

Total Number for Vehicle Samples = 8792

Total Number for Non Vehicle Samples = 8968

y corresponds to the labels of these images

20% of total samples is split to form the Test set (3552) [xTest, yTest]

Reamining 80 % forms the traning set (14208)

The remaining 80% is split to Training and Validation

Traning (11366)

Validation (2842)

All are stored in a dictionary data type “data {…}”

createSamples creates tuples for trainSamples and validationSamples

model.fit\_generator runs along with model.compile and fits the model on data generated batch-by-batch by a Python generator.

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

The fit generator requires 5 arguments to fit the data and run the model

1. Traning data generator
2. Validation data generator

Steps\_per\_epoch [stepsPerEpoch = len(trainSamples) \* inflateFactor / batchSize]

1. Validation steps [ validationSteps = len(validationSamples) \* inflateFactor / batchSize]
2. Epoch count

For the Training and Validation data generator, a python generator is created

Generator arguments is the list containing the tuple of file names of the positive and negative samples and their labels. (x, y), and batch size whose default value is set to 32

Inside the generator,

1. The samples are first shuffled
2. Two nested for loop where the images are provided in batches
3. Images are provided individually in batches and number of batches read depends on steps per epoch

stepsPerEpoch = len(trainSamples) \* inflateFactor / batchSize

InflateFactor =2 when using flips on the input images (steps per epoch)

* **Accuracy:** Overall, how often is the classifier correct?
  + (TP+TN)/total = (100+50)/165 = 0.91
* **Misclassification Rate:** Overall, how often is it wrong?
  + (FP+FN)/total = (10+5)/165 = 0.09
  + equivalent to 1 minus Accuracy
  + also known as "Error Rate"
* **True Positive Rate:** When it's actually yes, how often does it predict yes?
  + TP/actual yes = 100/105 = 0.95
  + also known as "Sensitivity" or "Recall"
* **False Positive Rate:** When it's actually no, how often does it predict yes?
  + FP/actual no = 10/60 = 0.17
* **Specificity:** When it's actually no, how often does it predict no?
  + TN/actual no = 50/60 = 0.83
  + equivalent to 1 minus False Positive Rate
* **Precision:** When it predicts yes, how often is it correct?
  + TP/predicted yes = 100/110 = 0.91
* **Prevalence:** How often does the yes condition actually occur in our sample?
  + actual yes/total = 105/165 = 0.64