**PHASE-4**

**Github link:**

<https://github.com/uteegull/DM_Car.git>

Dataset and project code with details have been uploaded on the github site.

Sample Source code for training model:

class ModelConfig:

* ModelConfig is a utility class that stores important configuration option about our model

def \_\_init\_\_(self, model, name, input\_img\_dimensions, conv\_layers\_config, fc\_output\_dims, output\_classes, dropout\_keep\_pct):

self.model = model

self.name=name

self.input\_img\_dimensions = input\_img\_dimensions

# Determines the wxh dimension of filters, the starting depth (increases by x2 at every layer)

# and how many convolutional layers the network has

self.conv\_filter\_size = conv\_layers\_config[0]

self.conv\_depth\_start = conv\_layers\_config[1]

self.conv\_layers\_count = conv\_layers\_config[2]

self.fc\_output\_dims = fc\_output\_dims

self.output\_classes = output\_classes

# Try with different values for drop out at convolutional and fully connected layers

self.dropout\_conv\_keep\_pct = dropout\_keep\_pct[0]

self.dropout\_fc\_keep\_pct = dropout\_keep\_pct[1]

class ModelExecutor:

"""

ModelExecutor is responsible for executing the supplied model

"""

def \_\_init\_\_(self, model\_config, learning\_rate=0.001):

self.model\_config = model\_config

self.learning\_rate = learning\_rate

self.graph = tf.Graph()

with self.graph.as\_default() as g:

with g.name\_scope( self.model\_config.name ) as scope:

# Create Model operations

self.create\_model\_operations()

# Create a saver to persist the results of execution

self.saver = tf.train.Saver()

def create\_placeholders(self):

"""

Defining our placeholder variables:

- x, y

- one\_hot\_y

- dropout placeholders

"""

# e.g. 32 \* 32 \* 3

input\_dims = self.model\_config.input\_img\_dimensions

self.x = tf.placeholder(tf.float32, (None, input\_dims[0], input\_dims[1], input\_dims[2]), name="{0}\_x".format(self.model\_config.name))

self.y = tf.placeholder(tf.int32, (None), name="{0}\_y".format(self.model\_config.name))

self.one\_hot\_y = tf.one\_hot(self.y, self.model\_config.output\_classes)

self.dropout\_placeholder\_conv = tf.placeholder(tf.float32)

self.dropout\_placeholder\_fc = tf.placeholder(tf.float32)

def create\_model\_operations(self):

"""

Sets up all operations needed to execute run deep learning pipeline

"""

# First step is to set our x, y, etc

self.create\_placeholders()

cnn = self.model\_config.model

# Build the network - TODO: pass the configuration in the future

self.logits = cnn(self.x, self.model\_config, self.dropout\_placeholder\_conv, self.dropout\_placeholder\_fc)

# Obviously, using softmax as the activation function for final layer

self.cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(labels=self.one\_hot\_y, logits=self.logits)

# Combined all the losses across batches

self.loss\_operation = tf.reduce\_mean(self.cross\_entropy)

# method do we use to reduce our loss

self.optimizer = tf.train.AdamOptimizer(learning\_rate=self.learning\_rate)

# What do we really do in a training operation then? Answer: we attempt to reduce the loss using our chosen optimizer

self.training\_operation = self.optimizer.minimize(self.loss\_operation)

# Get the top prediction for model against labels and check whether they match

self.correct\_prediction = tf.equal(tf.argmax(self.logits, 1), tf.argmax(self.one\_hot\_y, 1))

# Compute accuracy at batch level

self.accuracy\_operation = tf.reduce\_mean(tf.cast(self.correct\_prediction, tf.float32))

# compute what the prediction would be, when we don't have matching label

self.prediction = tf.argmax(self.logits, 1)

# Registering our top 5 predictions

self.top5\_predictions = tf.nn.top\_k(tf.nn.softmax(self.logits), k=5, sorted=True, name=None)

def evaluate\_model(self, X\_data, Y\_data, batch\_size):

"""

Evaluates the model's accuracy and loss for the supplied dataset.

Naturally, Dropout is ignored in this case (i.e. we set dropout\_keep\_pct to 1.0)

"""

num\_examples = len(X\_data)

total\_accuracy = 0.0

total\_loss = 0.0

sess = tf.get\_default\_session()

for offset in range(0, num\_examples, batch\_size):

batch\_x, batch\_y = X\_data[offset:offset+batch\_size], Y\_data[offset:offset+batch\_size]

# Compute both accuracy and loss for this batch

accuracy = sess.run(self.accuracy\_operation,

feed\_dict={

self.dropout\_placeholder\_conv: 1.0,

self.dropout\_placeholder\_fc: 1.0,

self.x: batch\_x,

self.y: batch\_y

})

loss = sess.run(self.loss\_operation, feed\_dict={

self.dropout\_placeholder\_conv: 1.0,

self.dropout\_placeholder\_fc: 1.0,

self.x: batch\_x,

self.y: batch\_y

})

# Weighting accuracy by the total number of elements in batch

total\_accuracy += (accuracy \* len(batch\_x))

total\_loss += (loss \* len(batch\_x))

# To produce a true mean accuracy over whole dataset

return (total\_accuracy / num\_examples, total\_loss / num\_examples)

def train\_model(self, X\_train\_features, X\_train\_labels, X\_valid\_features, y\_valid\_labels, batch\_size=512, epochs=100, PRINT\_FREQ=10):

"""

Trains the model for the specified number of epochs supplied when creating the executor

"""

# Create our array of metrics

training\_metrics = np.zeros((epochs, 3))

validation\_metrics = np.zeros((epochs, 3))

with tf.Session(graph = self.graph, config=tf.ConfigProto(allow\_soft\_placement=True, log\_device\_placement=True)) as sess:

sess.run(tf.global\_variables\_initializer())

num\_examples = len(X\_train\_features)

print("Training {0} [epochs={1}, batch\_size={2}]...\n".format(self.model\_config.name, epochs, batch\_size))

for i in range(epochs):

start = time.time()

X\_train, Y\_train = shuffle(X\_train\_features, X\_train\_labels)

for offset in range(0, num\_examples, batch\_size):

end = offset + batch\_size

batch\_x, batch\_y = X\_train[offset:end], Y\_train[offset:end]

sess.run(self.training\_operation, feed\_dict={

self.x: batch\_x,

self.y: batch\_y,

self.dropout\_placeholder\_conv: self.model\_config.dropout\_conv\_keep\_pct,

self.dropout\_placeholder\_fc: self.model\_config.dropout\_fc\_keep\_pct,

})

end\_training\_time = time.time()

training\_duration = end\_training\_time - start

#computing training accuracy

training\_accuracy, training\_loss = self.evaluate\_model(X\_train\_features, X\_train\_labels, batch\_size)

# Computing validation accuracy

validation\_accuracy, validation\_loss = self.evaluate\_model(X\_valid\_features, y\_valid\_labels, batch\_size)

end\_epoch\_time = time.time()

validation\_duration = end\_epoch\_time - end\_training\_time

epoch\_duration = end\_epoch\_time - start

if i == 0 or (i+1) % PRINT\_FREQ == 0:

print("[{0}]\ttotal={1:.3f}s | train: time={2:.3f}s, loss={3:.4f}, acc={4:.4f} | val: time={5:.3f}s, loss={6:.4f}, acc={7:.4f}".format(

i+1, epoch\_duration, training\_duration, training\_loss, training\_accuracy,

validation\_duration, validation\_loss, validation\_accuracy))

training\_metrics[i] = [training\_duration, training\_loss, training\_accuracy]

validation\_metrics[i] = [validation\_duration, validation\_loss, validation\_accuracy]

model\_file\_name = "{0}{1}.chkpt".format(models\_path, self.model\_config.name)

# Save the model

self.saver.save(sess, model\_file\_name)

print("Model {0} saved".format(model\_file\_name))

return (training\_metrics, validation\_metrics, epoch\_duration)

def test\_model(self, test\_imgs, test\_lbs, batch\_size=512):

"""

Evaluating the model with the test dataset and test labels

Returns the tuple (test\_accuracy, test\_loss, duration)

"""

with tf.Session(graph = self.graph) as sess:

# Never forget to re-initialise the variables

tf.global\_variables\_initializer()

model\_file\_name = "{0}{1}.chkpt".format(models\_path, self.model\_config.name)

self.saver.restore(sess, model\_file\_name)

start = time.time()

(test\_accuracy, test\_loss) = self.evaluate\_model(test\_imgs, test\_lbs, batch\_size)

duration = time.time() - start

print("[{0} - Test Set]\ttime={1:.3f}s, loss={2:.4f}, acc={3:.4f}".format(self.model\_config.name, duration, test\_loss, test\_accuracy))

return (test\_accuracy, test\_loss, duration)

Image Size: 35X35( height and width)

CNN architecture: The network is comprised of 3 convolutional layers – with kernel size of 3X3 , with depth doubling at next layer — using [ReLU](https://en.wikipedia.org/wiki/Rectifier_%28neural_networks%29" \t "_blank) as the activation function, each followed by a 2x2 max pooling operation. The last 3 layers are fully connected, with the final layer producing 43 results (the total number of possible labels) computed using the [SoftMax](https://www.quora.com/Why-is-softmax-activate-function-called-softmax) activation function.