All the below code has been run and tested on Kaggle GPU.

Import Libraries

In [1]:

```
import torch
import torchvision
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.data.sampler import SubsetRandomSampler
import numpy as np
import math
import os
import cv2
import time
import pandas as pd
from matplotlib import pyplot as plt
from matplotlib import style
from numpy import genfromtxt
# import torchvision.transforms
from torchvision import transforms
```

Detect if we have a GPU available

In [2]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

Define Hyper-parameters

In [3]:

```
MODEL_SAVE_PATH = './self_driving_car.pth'



learning_rate = 1e-4
num_epochs = 5
batch_size = 256
num_workers=8
```

Creating and preprocessing dataset

In [4]:

```
class CarDataset(Dataset):

    def __init__(self):
        """Initialize Dataset class. """

        ### default directory where data is loaded ###
        self.filepath = '/kaggle/input/car-steering-angle-prediction/driving_dataset/'
        self.xs = []
        self.ys = []

        ### read data.txt ###
        with open("/kaggle/input/car-steering-angle-prediction/driving_dataset/angles.txt")
            for line in f:
                self.xs.append(line.split()[0])

                 ### The paper by Nvidia uses the inverse of the turning radius, but steeri
                   # So the steering wheel angle in radians is used as the output. ###

                self.ys.append(float(line.split()[1]) * 3.14159265 / 180)




    def __len__(self):
        """we can call len(dataset) to return the size"""
        return len(self.xs)


    def __getitem__(self, index):
        """support indexing such that dataset[i] can be used to get i-th sample"""
        filename = self.xs[index]

        full_path_img = self.filepath + filename
        img = cv2.imread(full_path_img)

        ### Resizing and normalizing images ###
#         img_resize = cv2.resize(img, (200,66), interpolation = cv2.INTER_AREA)/ 255.0
        img_resize = cv2.resize(img, (200,66), interpolation = cv2.INTER_AREA)

        ### Returned the image converted to a numpy array with its corresponding steering a
        X_img = torch.from_numpy(img_resize).float()
        Y_label = torch.tensor(self.ys[index])
        return X_img,Y_label
```

Class that implements NVIDIA model

In [16]:

```
### The model includes ELU layers after each convolutional or fully connected layer to intr
### Standard RELU's as activation function can turn "dead", which means that they are never
### So ELU is used after each convolutional layer

class ConvNet(nn.Module):

    def __init__(self):
        """Initialize ConvNet class to implement NVIDIA model. """

        super(ConvNet, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 24, kernel_size=5, stride=2),
            nn.ELU(),
            nn.Conv2d(24, 36, kernel_size=5, stride=2),
            nn.ELU(),
            nn.Conv2d(36, 48, kernel_size=5, stride=2),
            nn.ELU(),
            nn.Conv2d(48, 64, kernel_size=3, stride=1),
            nn.ELU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ELU(),
            nn.Dropout(p=0.5)
        )

        self.linear_layers = nn.Sequential(
            nn.Linear(in_features=64*1*18, out_features=100),
            nn.ELU(),
            nn.Dropout(p=0.5),
            nn.Linear(in_features=100, out_features=50),
            nn.ELU(),
            nn.Linear(in_features=50, out_features=10),
            nn.ELU(),
            nn.Linear(in_features=10, out_features=1)
        )


    def forward(self, x):
        """Forward pass."""

        x = x.view(x.size(0), 3, 66, 200)
        output = self.conv_layers(x)
        output = output.view(output.size(0), -1)
        output = self.linear_layers(output)
        return output
```

Split dataset into training, validation and test sets

In [6]:

```
class DataSplit:

    def __init__(self, dataset, test_train_split=0.8, val_train_split=0.2, shuffle=True):
        """Initialize dataSplit class"""

        self.dataset = dataset
        dataset_size = len(dataset)

        self.indices = list(range(dataset_size))
        test_split = int(np.floor(test_train_split * dataset_size))

        if shuffle:
            np.random.seed(3116)
            np.random.shuffle(self.indices)

        train_indices, self.test_indices = self.indices[:test_split], self.indices[test_spl
        train_size = len(train_indices)
        validation_split = int(np.floor((1 - val_train_split) * train_size))

        self.train_indices = train_indices[ : validation_split]
        self.val_indices = train_indices[validation_split:]

        self.train_sampler = SubsetRandomSampler(self.train_indices)
        self.val_sampler = SubsetRandomSampler(self.val_indices)
        self.test_sampler = SubsetRandomSampler(self.test_indices)

    def get_split(self, batch_size=64, num_workers=8):
        self.train_loader = torch.utils.data.DataLoader(self.dataset, batch_size=batch_size
        self.val_loader = torch.utils.data.DataLoader(self.dataset, batch_size=batch_size,
        self.test_loader = torch.utils.data.DataLoader(self.dataset, batch_size=batch_size,
        return self.train_loader, self.val_loader, self.test_loader
```

Helper function to calculate RMSE

In [7]:

```
class RMSELoss(nn.Module):
    def __init__(self):
        super().__init__()
        self.mse = nn.MSELoss()

    def forward(self,yhat,y):
        return torch.sqrt(self.mse(yhat,y))
```

Helper function to train model on dataset

In [8]:

```
def train_model(model, train_loader, val_loader , criterion, optimizer, num_epochs=25):

    """Train model on train set and validate it on validation set"""

    train_losses, val_losses,  = [], []
    best_loss = 999999.0
    start = time.time()


    ### Each epoch has a training and validation phase ###
    for epoch in range(num_epochs):
        print('-' * 10)
        print('Epoch {}/{}'.format(epoch+1, num_epochs))

        ### Training phase ###
        train_loss = 0.0
        model.train()
        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)

            loss = criterion(outputs, labels.view(-1,1))

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        ### Validation phase ###
        val_loss = 0.0
        model.eval()
        with torch.no_grad():
            for i, (images, labels) in enumerate(val_loader):
                images = images.to(device)
                labels = labels.to(device)

                # Forward pass
                outputs = model(images)
                loss = criterion(outputs, labels.view(-1,1))

                val_loss += loss.item()

        # Average validation loss
        train_loss = train_loss / len(train_loader)
        val_loss = val_loss / len(val_loader)

        train_losses.append(train_loss)
        val_losses.append(val_loss)

        print('Train Loss: {:.2f}'.format(train_loss))
        print('Val Loss: {:.2f}'.format(val_loss))

        ### If the validation loss is at a minimum ###
        if val_loss < best_loss:
```

```
            torch.save(model,MODEL_SAVE_PATH)
            best_loss = val_loss

    time_elapsed = time.time() - start
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed %

    return model
```

Create dataset and split it into train, validate and test splits!

In [9]:

```
print("==> Preparing dataset ...")

# create dataset
dataset = CarDataset()

# Creating data indices for training, validation splits and testing splits:
split = DataSplit(dataset, shuffle=True)

train_loader, val_loader, test_loader = split.get_split(batch_size=batch_size, num_workers=

print("... Preparation of dataset done <==")
```

```
==> Preparing dataset ...
... Preparation of dataset done <==
```

Define model

In [10]:

```
print("==> Initialize model and transfer it to GPU if available ...")
model = ConvNet().to(device)
print("... Initialization of model done <==")
```

```
==> Initialize model and transfer it to GPU if available ...
... Initialization of model done <==
```

Define optimizer and criterion

In [11]:

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()
```

Train and evaluate model on validate set

In [12]:

```
model_ft = train_model(model, train_loader, val_loader,  criterion, optimizer, num_epochs=n
```

```
----------
Epoch 1/5
Train Loss: 0.27
Val Loss: 0.25
----------
Epoch 2/5
Train Loss: 0.23
Val Loss: 0.20
----------
Epoch 3/5
Train Loss: 0.19
Val Loss: 0.17
----------
Epoch 4/5
Train Loss: 0.14
Val Loss: 0.12
----------
Epoch 5/5
Train Loss: 0.11
Val Loss: 0.10
Training complete in 9m 29s
```

Transfer the model to GPU to evaluate it on test set

In [13]:

```
model_load = model_ft.to(device)
```

Evaluate the saved model on test set

In [14]:

```
criterion_rmse = RMSELoss()
```

In [15]:

```
Rmse_total=0

model.eval()

with torch.no_grad():
    for i, (images, labels) in enumerate(test_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model_load(images)
        loss = criterion_rmse(outputs, labels.view(-1,1))

        Rmse_total += loss.item()

# print('RMSE on test images: {:.0f}'.format(Rmse_total))

test_loss = Rmse_total / len(test_loader)
print('RMSE on test images: {:.2f}'.format(test_loss))
```

```
RMSE on test images: 0.31
```

Bonus Exercises

1)Cut_out for regularization:

Updated helper classes

In [17]:

```
class Cutout:
    """
    Randomly mask out one or more patches from an image.
    """
    def __init__(self, n_holes, length):
        """
        n_holes (int): Number of patches to cut out of each image.
        length (int): The length (in pixels) of each square patch.
        """
        self.n_holes = n_holes
        self.length = length

    def __call__(self, img):
        h = img.size(1) #img.height
        w = img.size(2) #img.width

        mask = np.ones((h, w), np.float32)

        for n in range(self.n_holes):
            y = np.random.randint(h)
            x = np.random.randint(w)

            y1 = np.clip(y - self.length // 2, 0, h)
            y2 = np.clip(y + self.length // 2, 0, h)
            x1 = np.clip(x - self.length // 2, 0, w)
            x2 = np.clip(x + self.length // 2, 0, w)

            mask[y1: y2, x1: x2] = 0.

        mask = torch.from_numpy(mask)
        mask = mask.expand_as(img)
        img = img * mask

        return img
```

In [18]:

```
def random_flip(image, steering_angle):
    """
    Randomly flipt the image and adjust the steering angle.
    """
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 1)
        steering_angle = -steering_angle
    return image, steering_angle
```

In [19]:

```
class CarDatasetCutout(Dataset):

    def __init__(self,transform = None):
        """Initialize Dataset class. """

        ### default directory where data is loaded ###
        self.filepath = '/kaggle/input/car-steering-angle-prediction/driving_dataset/'
        self.xs = []
        self.ys = []
        self.transform = transform


        ### read data.txt ###
        with open("/kaggle/input/car-steering-angle-prediction/driving_dataset/angles.txt")
            for line in f:
                self.xs.append(line.split()[0])

                 ### The paper by Nvidia uses the inverse of the turning radius, but steeri
                   # So the steering wheel angle in radians is used as the output. ###

                self.ys.append(float(line.split()[1]) * 3.14159265 / 180)




    def __len__(self):
        """we can call len(dataset) to return the size"""
        return len(self.xs)


    def __getitem__(self, index):
        """support indexing such that dataset[i] can be used to get i-th sample"""
        filename = self.xs[index]

        full_path_img = self.filepath + filename
        img = cv2.imread(full_path_img)


        # Randomly flip images
        if np.random.rand() < 0.6:
            X_img, Y_label = random_flip(img, self.ys[index])
        else:
            X_img = img
            Y_label =  self.ys[index]

        ### Resizing images ###
        img_resize = cv2.resize(X_img, (200,66), interpolation = cv2.INTER_AREA)


        ### Returned the image tranformed to a numpy array with its corresponding steering
        X_img_transformed = self.transform(img_resize)
        Y_label = Y_label


        return X_img_transformed,Y_label
```

In [20]:

```
def train_model_cutout(model, train_loader, val_loader , criterion, optimizer, num_epochs=2

    """Train model on train set and validate it on validation set"""

    train_losses, val_losses,  = [], []
    best_loss = 10000.0
    start = time.time()


    ### Each epoch has a training and validation phase ###
    for epoch in range(num_epochs):
        print('-' * 10)

        print('Epoch {}/{}'.format(epoch+1, num_epochs))

        ### Training phase ###
        train_loss = 0.0
        model.train()
        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device,dtype=torch.float)
            labels = labels.to(device, dtype=torch.float)

            # Forward pass
            outputs = model(images)

            loss = criterion(outputs, labels.view(-1,1))

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        ### Validation phase ###
        val_loss = 0.0
        model.eval()
        with torch.no_grad():
            for i, (images, labels) in enumerate(val_loader):
                images = images.to(device, dtype=torch.float)
                labels = labels.to(device, dtype=torch.float)

                # Forward pass
                outputs = model(images)
                loss = criterion(outputs, labels.view(-1,1))

                val_loss += loss.item()

        # Average validation loss
        train_loss = train_loss / len(train_loader)
        val_loss = val_loss / len(val_loader)

        train_losses.append(train_loss)
        val_losses.append(val_loss)

        print('Train Loss: {:.2f}'.format(train_loss))
        print('Val Loss: {:.2f}'.format(val_loss))

        ### If the validation loss is at a minimum ###
```

```
        if val_loss < best_loss:
            torch.save(model,MODEL_SAVE_PATH)
            best_loss = val_loss

    time_elapsed = time.time() - start
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed %
    return model
```

Load dataset after cutout

In [21]:

```
# Load Data with the transformations including cutout for regularization

transformations = transforms.Compose([transforms.Lambda(lambda x: (x / 127.5) - 1.0), torch
transformations.transforms.append(Cutout(n_holes=2, length=32)) # Apply cutout

print("==> Preparing dataset ...")

# create dataset
dataset = CarDatasetCutout(transformations)

# Creating data indices for training, validation splits and testing splits:
split = DataSplit(dataset, shuffle=True)

train_loader_cutout, val_loader_cutout, test_loader_cutout = split.get_split(batch_size=bat

print("... Preparation of dataset done <==")
```

```
==> Preparing dataset ...
... Preparation of dataset done <==
```

Loading model

In [22]:

```
print("==> Initialize model and transfer it to GPU if available ...")
model = ConvNet().to(device)
print("... Initialization of model done <==")
```

```
==> Initialize model and transfer it to GPU if available ...
... Initialization of model done <==
```

Train and evaluate model on validate set

In [23]:

```
model_ft_cutout = train_model_cutout(model, train_loader_cutout, val_loader_cutout,  criter
```

```
----------
Epoch 1/5
Train Loss: 0.33
Val Loss: 0.34
----------
Epoch 2/5
Train Loss: 0.33
Val Loss: 0.33
----------
Epoch 3/5
Train Loss: 0.33
Val Loss: 0.33
----------
Epoch 4/5
Train Loss: 0.33
Val Loss: 0.33
----------
Epoch 5/5
Train Loss: 0.33
Val Loss: 0.34
Training complete in 11m 17s
```

Evaluate model on test set

In [24]:

```
Rmse_total=0

model.eval()

with torch.no_grad():
    for i, (images, labels) in enumerate(test_loader_cutout):
        images = images.to(device, dtype=torch.float)
        labels = labels.to(device, dtype=torch.float)

        # Forward pass
        outputs = model_ft_cutout(images)
        loss = criterion_rmse(outputs, labels.view(-1,1))

        Rmse_total += loss.item()

# print('RMSE on test images: {:.0f}'.format(Rmse_total))

test_loss = Rmse_total / len(test_loader)
print('RMSE on test images: {:.2f}'.format(test_loss))
```

```
RMSE on test images: 0.56
```

In [ ]:

```

```

2) MixUp for regularization:

Updated HyperClasses

In [25]:

```
class CarDatasetMixup(Dataset):

    def __init__(self,transform = None):
        """Initialize Dataset class. """

        ### default directory where data is loaded ###
        self.filepath = '/kaggle/input/car-steering-angle-prediction/driving_dataset/'
        self.xs = []
        self.ys = []
        self.transform = transform


        ### read data.txt ###
        with open("/kaggle/input/car-steering-angle-prediction/driving_dataset/angles.txt")
            for line in f:
                self.xs.append(line.split()[0])

                 ### The paper by Nvidia uses the inverse of the turning radius, but steeri
                   # So the steering wheel angle in radians is used as the output. ###

                self.ys.append(float(line.split()[1]) * 3.14159265 / 180)




    def __len__(self):
        """we can call len(dataset) to return the size"""
        return len(self.xs)


    def __getitem__(self, index):
        """support indexing such that dataset[i] can be used to get i-th sample"""
        filename = self.xs[index]

        full_path_img = self.filepath + filename
        img = cv2.imread(full_path_img)


        X_img = img
        Y_label =  self.ys[index]

        ### Resizing images ###

        img_resize = cv2.resize(X_img, (200,66), interpolation = cv2.INTER_AREA)


        ### Returned the image tranformed to a numpy array with its corresponding steering
        X_img_transformed = self.transform(img_resize)
        Y_label = Y_label


        return X_img_transformed,Y_label
```

In [26]:

```
# Function implementing mixup regularization. It required one hot vector for labels.
# But given dataset does not have continuos labels, so we cant convert labels to one hot ve

def mixup(data, targets, alpha):
    indices = torch.randperm(data.size(0))
    data2 = data[indices]
    targets2 = targets[indices]


    lam = torch.FloatTensor([np.random.beta(alpha, alpha)])
    data = data * lam + data2 * (1 - lam)
    targets = targets * lam + targets2 * (1 - lam)

    return data, targets
```

In [27]:

```
def train_model_mixup(model, train_loader, val_loader , criterion, optimizer, num_epochs=25

    """Train model on train set and validate it on validation set"""

    train_losses, val_losses,  = [], []
    best_loss = 99999.0
    start = time.time()


    ### Each epoch has a training and validation phase ###
    for epoch in range(num_epochs):
        print('-' * 10)
        print('Epoch {}/{}'.format(epoch+1, num_epochs))


        ### Training phase ###
        train_loss = 0.0
        model.train()
        for i, (images, labels) in enumerate(train_loader):

            images,labels  = mixup(images, labels, 1)


            images = images.to(device,dtype=torch.float)
            labels = labels.to(device, dtype=torch.float)

            # Forward pass
            outputs = model(images)

            loss = criterion(outputs, labels.view(-1,1))

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        ### Validation phase ###
        val_loss = 0.0
        model.eval()
        with torch.no_grad():
            for i, (images, labels) in enumerate(val_loader):
                images = images.to(device, dtype=torch.float)
                labels = labels.to(device, dtype=torch.float)

                # Forward pass
                outputs = model(images)
                loss = criterion(outputs, labels.view(-1,1))

                val_loss += loss.item()

        # Average validation loss
        train_loss = train_loss / len(train_loader)
        val_loss = val_loss / len(val_loader)

        train_losses.append(train_loss)
        val_losses.append(val_loss)
```

```
        print('Train Loss: {:.2f}'.format(train_loss))
        print('Val Loss: {:.2f}'.format(val_loss))

        ### If the validation loss is at a minimum ###
        if val_loss < best_loss:
            torch.save(model,MODEL_SAVE_PATH)
            best_loss = val_loss

    time_elapsed = time.time() - start
    return model
```

Load dataset

In [28]:

```
# Load Dataset

transformations = transforms.Compose([transforms.Lambda(lambda x: (x / 127.5) - 1.0), torch

print("==> Preparing dataset ...")

# create dataset
dataset = CarDatasetMixup(transformations)

# Creating data indices for training, validation splits and testing splits:
split = DataSplit(dataset, shuffle=True)

train_loader_mixup, val_loader_mixup, test_loader_mixup = split.get_split(batch_size=batch_

print("... Preparation of dataset done <==")
```

```
==> Preparing dataset ...
... Preparation of dataset done <==
```

Train and evaluate model on validate set

In [ ]:

```
model_mixup = train_model_mixup(model, train_loader_mixup, val_loader_mixup,  criterion, op
```

```
----------
Epoch 1/5
Train Loss: 0.25
Val Loss: 0.34
----------
Epoch 2/5
```

Evaluate model on Test set

In [291]:

```
Rmse_total=0

model.eval()

with torch.no_grad():
    for i, (images, labels) in enumerate(test_loader_mixup):
        images = images.to(device, dtype=torch.float)
        labels = labels.to(device, dtype=torch.float)

        # Forward pass
        outputs = model_mixup(images)
        loss = criterion_rmse(outputs, labels.view(-1,1))

        Rmse_total += loss.item()

# print('RMSE on test images: {:.0f}'.format(Rmse_total))

test_loss = Rmse_total / len(test_loader)
print('RMSE on test images: {:.2f}'.format(test_loss))
```

```
RMSE on test images: 0.57
```

In [ ]:

```

```

3) HyperBand Alogrithm for regularization:

The Algorithm was taking alot of time to train. So I just used 2000 images from dataset and run the experiments on some hyper parameters.

Updated helper classes

In [316]:

```
class DataSplitHyperBand:

    def __init__(self, dataset, test_train_split=0.8, val_train_split=0.2, shuffle=True):
        """Initialize dataSplit class"""

        self.dataset = dataset
        dataset_size = 2000

        self.indices = list(range(dataset_size))
        test_split = int(np.floor(test_train_split * dataset_size))

        if shuffle:
            np.random.seed(3116)
            np.random.shuffle(self.indices)

        train_indices, self.test_indices = self.indices[:test_split], self.indices[test_spl
        train_size = len(train_indices)
        validation_split = int(np.floor((1 - val_train_split) * train_size))

        self.train_indices = train_indices[ : validation_split]
        self.val_indices = train_indices[validation_split:]

        self.train_sampler = SubsetRandomSampler(self.train_indices)
        self.val_sampler = SubsetRandomSampler(self.val_indices)
        self.test_sampler = SubsetRandomSampler(self.test_indices)

    def get_split(self, batch_size=64, num_workers=8):
        self.train_loader = torch.utils.data.DataLoader(self.dataset, batch_size=batch_size
        self.val_loader = torch.utils.data.DataLoader(self.dataset, batch_size=batch_size,
        self.test_loader = torch.utils.data.DataLoader(self.dataset, batch_size=batch_size,
        return self.train_loader, self.val_loader, self.test_loader
```

In [ ]:

```
def train_model_hyperband(model, train_loader, val_loader , criterion, num_epochs, lr, wd ):


    """Train model on train set and validate it on validation set"""

    train_losses, val_losses,  = [], []
    best_loss = 10000.0
    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=wd)


    ### Each epoch has a training and validation phase ###
    for epoch in range(num_epochs):

        ### Training phase ###
        train_loss = 0.0
        model.train()
        for i, (images, labels) in enumerate(train_loader):

            images,labels  = mixup(images, labels, 1)


            images = images.to(device,dtype=torch.float)
            labels = labels.to(device, dtype=torch.float)

            # Forward pass
            outputs = model(images)

            loss = criterion(outputs, labels.view(-1,1))

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        ### Validation phase ###
        val_loss = 0.0
        model.eval()
        with torch.no_grad():
            for i, (images, labels) in enumerate(val_loader):
                images = images.to(device, dtype=torch.float)
                labels = labels.to(device, dtype=torch.float)

                # Forward pass
                outputs = model(images)
                loss = criterion(outputs, labels.view(-1,1))

                val_loss += loss.item()

        # Average validation loss
        train_loss = train_loss / len(train_loader)
        val_loss = val_loss / len(val_loader)

        train_losses.append(train_loss)
        val_losses.append(val_loss)

        ### If the validation loss is at a minimum ###
        if val_loss < best_loss:
```

```
#             torch.save(model,MODEL_SAVE_PATH)
            best_loss = val_loss

    return val_loss
```

In [ ]:

```
class hyperband:

    def __init__(self,space):
        self.search_space = space
        self.myModel = ConvNet().to(device)


    def sample_space(self,n_samples):

        config = np.array([np.random.choice(self.search_space[val],n_samples,replace=True)
        return np.transpose(config)

    def model_hyperband(self, epochs, params):

        print('\n \t Running the configurations ',params,' for - ',int(epochs),' epochs')

        matrix = train_model_hyperband(self.myModel, train_loader_hyperband, val_loader_hyp
        return matrix



    def search(self,max_iter=5,eta=3,skip_last=1):

        logeta = lambda x: math.log(x) / math.log(eta)
        s_max = int(logeta(max_iter))
        B = (s_max + 1) * max_iter

        result = np.array([])
        best_config = np.array([])

        ## this loop denotes the no. of unique run of successive halving
        for s in reversed(range(s_max + 1)):

            print('\n  Current bracket number - ', s)

            if skip_last:
                if s == 0: break

            n = int(math.ceil(int(B / max_iter / (s + 1)) * eta ** s))  # number of configu
            r = max_iter * eta ** (-s)  # number of resources at starting for given bracket

            T = self.sample_space(n)  # sampling from the search space
            metric = np.array([])

            ## this loop runs the successive halving for a given bracket
            for i in range(s + 1):

                n_i = n * eta ** (-i)  # no. of configs for given successive halving
                r_i = r * eta ** (i)  # no. of resources for given successive halving
                val_metric = np.array([self.model_hyperband(r_i,t) for t in T])  # getting
                T = np.array([T[i] for i in reversed(np.argsort(val_metric)[int(n / eta):])
                metric = np.append(metric,np.max(val_metric))


                print('\n\n \t number of reduction/successive halving done - ', i)

            best_config = np.append(best_config, T[:2])  # keeping track of the best config
            result = np.append(result,metric[-1])
```

```
        best = best_config[np.argmax(result)]
        print('\n\n the Best configuration - ', best)
```

Load Dataset

In [362]:

```
print("==> Preparing dataset ...")

# create dataset
dataset = CarDataset()

# Creating data indices for training, validation splits and testing splits:
split = DataSplitHyperBand(dataset, shuffle=True)

train_loader_hyperband, val_loader_hyperband, test_loader_hyperband = split.get_split(batch

print("... Preparation of dataset done <==")
```

```
==> Preparing dataset ...
... Preparation of dataset done <==
```

Run hyper band algorithm to best optimal paramters

In [363]:

```
## defining the search space and run hyperband to get best paramaters on which model perfor

space = {'lr': np.array([1e-4, 1e-3, 1e-2]), 'weight_decay': np.array([1e-2,1e-1,0])}

hyper_band = hyperband(space)

hyper_band.search()   ## calling the search function from hyperband
```

```
 Current bracket number -  1

        running the config  [0.01 0.1 ]  for -  1  epochs

        running the config  [0.001 0.   ]  for -  1  epochs

        running the config  [0.001 0.   ]  for -  1  epochs


        number of reduction/successive halving done -  0

        running the config  [0.01 0.1 ]  for -  5  epochs

        running the config  [0.001 0.   ]  for -  5  epochs


        number of reduction/successive halving done -  1

 Current bracket number -  0


the best configuration -  0.1
```