Lab 8 - ML Programming Task 2 + Bonus

January 15, 2022

[ ]:
```
!pip install cv2
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import cv2
import os
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
import torch.nn.functional as F
```

[ ]:
```
## Read the images from Kaggle

## Code given in lab instructions
class Dataset(Dataset): ## Inherits from torch.utils.data.Dataset
    def __init__(self):
        ## default directory where data is loaded
        self.filepath = '/kaggle/input/car-steering-angle-prediction/
↪driving_dataset/'
        self.filenames = os.listdir(self.filepath)

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, index):
        filename = self.filenames[index]
        img = cv2.imread(self.filepath + filename)
        ## Resize images to (66, 200) to match paper inputs
        img = cv2.resize(img, (66, 200), fx=0, fy=0, interpolation = cv2.
↪INTER_AREA)
        ## Return the image converted to a numpy array its corresponding␣
↪steering angle
        return torch.from_numpy(img.transpose()).float(), torch.rand(1)

data = Dataset()
```

[ ]:
```
## Divide data into corresponding train/validation/test splits
## Leave the last 10k images for testing (images are id'ed)

data_len = data.__len__()

indices = list(range(data_len))

train_split = int((data_len)* 0.3)
test_split = (data_len) - 10000
train_indices, val_indices, test_indices = indices[0:train_split],␣
↪indices[train_split:test_split], indices[test_split:]
print(f'Training data length = {train_indices.__len__()}')
print(f'Validation data length = {val_indices.__len__()}')
print(f'Test data length = {test_indices.__len__()}')

## Code given in lab instructions
train_loader = torch.utils.data.DataLoader(data, batch_size = 60, sampler =␣
↪train_indices)
val_loader = torch.utils.data.DataLoader(data, batch_size = 60, sampler =␣
↪val_indices)
test_loader = torch.utils.data.DataLoader(data, batch_size = 60, sampler =␣
↪test_indices)
```

[ ]:
```
## Implement the Convolutional Neural Network Architecture proposed in the␣
↪paper titled, "End to End Learning for Self-Driving Cars"

## Overall code structure given in lab instructions
class ConvNet(torch.nn.Module):
    def __init__(self):
        ## The network consists of 9 layers, including a normalization layer, 5␣
↪convolutional layers and 3 fully connected layers
        super(ConvNet, self).__init__()
        self.norm = nn.BatchNorm2d(3)
        ## Use strided convolutions in 1st 3 convolutional layers with a 2×2␣
↪stride and a 5×5 kernel
        ## Use non-strided convolution with a 3×3 kernel size in the last two␣
↪convolutional layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=24, kernel_size=5,␣
↪stride=(2,2))
        self.conv2 = nn.Conv2d(in_channels=24, out_channels=36, kernel_size=5,␣
↪stride=(2,2))
        self.conv3 = nn.Conv2d(in_channels=36, out_channels=48, kernel_size=5,␣
↪stride=(2,2))
        ## Non-strided convolution: stride=1
        self.conv4 = nn.Conv2d(in_channels=48, out_channels=64, kernel_size=3,␣
↪stride=1)
```

```
        self.conv5 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,␣
↪stride=1)
        self.flat = nn.Flatten()
        self.dense1 = nn.Linear(in_features=1152, out_features=1164)
        ## 3 fully connected layers
        self.dense2 = nn.Linear(in_features=1164, out_features=100)
        self.dense3 = nn.Linear(in_features=100, out_features=50)
        self.dense4 = nn.Linear(in_features=50, out_features=10)
        ## Output
        self.dense5 = nn.Linear(in_features=10,out_features=1)

    def forward(self, x):
        x = self.norm(x)
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.flat(x)
        x = self.dense1(x)
        x = self.dense2(x)
        x = self.dense3(x)
        x = self.dense4(x)
        x = self.dense5(x)
        return x

net = ConvNet()
optimizer = torch.optim.Adam(net.parameters(), lr = 1e-3)
criterion = torch.nn.MSELoss()

## Training taken from https://pytorch.org/tutorials/beginner/blitz/
↪cifar10_tutorial.html
for epoch in range(5):
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 180 == 179:
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 180:.3f}')
            running_loss = 0.0
print('Finished Training')
```

```
[ ]: ## Report one test RMSE for the test set of images

MSEsum = 0
count = 0
for i, data in enumerate(test_loader):
    yhat = net(data[0])
    MSE = criterion(yhat, torch.rand([60,1]))
    MSEsum = MSEsum + MSE
    count = count + 1
print (f'Test RMSE: {torch.sqrt(MSEsum/count)}')
```

1 BONUS

```
[ ]: ## Tune the associated hyperparameters
## (e.g. batch_size, number_of_layers, kernel_sizes, learning_rate,␣
 ↪l1_regularization, l2_regularization coefficients etc.)
## Either implement Random Search or Hyperband

????

from sklearn.model_selection import KFold
from sklearn.model_selection import RandomizedSearchCV

param_grid  = {
    'batch_size': [36,48,60],
    'number_of_layers': [6,9,12],
    'kernel_sizes': [(2,2),(3,3),(5,5)],
    'l1_regularization': ['none','l1'],
    'learning_rate': ['constant','adaptive'],
    'l2_regularization': ['none','l2']
}

k_fold = KFold(n_splits=5, shuffle=True, random_state=3116)
conv_net = ConvNet()
search = RandomizedSearchCV(conv_net, param_grid, cv=k_fold, random_state=3116,␣
 ↪n_jobs=-1).fit(data)
search.best_params_
```

```
[ ]: ## Implement regularization scheme named "Cutout"

## "randomly masking out square regions of input during training"
def Cutout(image, size=12, n_holes=1):
    h = image.size(1)
    w = image.size(2)
    mask = np.zeros((h, w), np.float32)
    for n in range(n_holes):
```

```
        y = np.random.randint(h)
        x = np.random.randint(w)
        y1 = np.clip(y - size // 2, 0, h)
        y2 = np.clip(y + size // 2, 0, h)
        x1 = np.clip(x - size // 2, 0, w)
        x2 = np.clip(x + size // 2, 0, w)
        mask[y1:y2, x1:x2] = 0
    mask = torch.from_numpy(mask)
    mask = mask.expand_as(image)
    image = image * mask
    return image
```

[ ]:
```
## Implement the regularization scheme titled, "MixUp"

## Code taken from https://www.kaggle.com/daisukelab/
 ↪mixup-cutout-or-random-erasing-to-augment
## Haven't had time to adjust algorithm according to the paper + structure of␣
 ↪our data
def mixup(data, one_hot_labels, alpha=1, debug=False):
    np.random.seed(3116)
    batch_size = len(data)
    weights = np.random.beta(alpha, alpha, batch_size)
    index = np.random.permutation(batch_size)
    x1, x2 = data, data[index]
    x = np.array([x1[i] * weights [i] + x2[i] * (1 - weights[i]) for i in␣
 ↪range(len(weights))])
    y1 = np.array(one_hot_labels).astype(np.float)
    y2 = np.array(np.array(one_hot_labels)[index]).astype(np.float)
    y = np.array([y1[i] * weights[i] + y2[i] * (1 - weights[i]) for i in␣
 ↪range(len(weights))])
    if debug:
        print('Mixup weights', weights)
    return x, y
```