FACE MASK DETECTION (CODING) :

IMPORTING PACKAGES :

```
import torch
from torch.utils.data import Dataset
import pandas as pd
from torchvision.io import read_image
from torchvision.transforms import Resize
from torch import nn
from torch.utils.data import DataLoader
from torch.optim import Optimizer
from torch.optim import Adam
from torch import Tensor
from sklearn.model_selection import train_test_split
import numpy as np
import time
import os
import matplotlib.pyplot as plt
```

OBTAINING OF DATASET

```
class MaskImgDataset(Dataset):
    def __init__(self, data_frame):
        self.data_path = "/kaggle/input/face-mask-detection-dataset/images/"
        self.data_frame = data_frame
        self.transformations = Resize((100, 100))

    def __len__(self):
        return len(self.data_frame["File"])

    def __getitem__(self, index):
        img = read_image(self.data_path + self.data_frame["File"][index])
        label = self.data_frame["Label"][index]

        img = self.transformations(img).float()

        return img, label
```

DEFINE THE STRUCTURE AND THE STEPS OF THE MODEL

```
class MaskDetector(nn.Module):
    def __init__(self, loss_function):
        super(MaskDetector, self).__init__()

        self.loss_function = loss_function

        self.conv2d_1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=(3,3), padding=(1,1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2,2))
        )

        self.conv2d_2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=(3,3), padding=(1,1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2,2))
        )

        self.conv2d_3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=(3,3), padding=(1,1), stride=(3,3)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2,2))
        )

        self.linearLayers = nn.Sequential(
            nn.Linear(in_features=2048, out_features=1024),
            nn.ReLU(),
            nn.Linear(in_features=1024, out_features=2)
        )

        for sequential in [self.conv2d_1, self.conv2d_2, self.conv2d_3, self.linearLayers]:
            for layer in sequential.children():
                if isinstance(layer, (nn.Linear, nn.Conv2d)):
                    nn.init.xavier_uniform_(layer.weight)
```

```
    def forward(self, x):
        out = self.conv2d_1(x)
        out = self.conv2d_2(out)
        out = self.conv2d_3(out)
        out = out.view(-1, 2048)
        out = self.linearLayers(out)

        return out

    def add_optimizer(self, optimizer):
        self.optimizer = optimizer
def plot_losses(loss_db):
    plt.plot(loss_db)
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
```

ONE EPOCH OF TRAINING

```
   def model_train_loop(model, train_DL):
    total_loss = 0
    total_correct = 0

    for inputs, labels in train_DL:  # iterate over batches
        # move inputs to GPU if used
        inputs, labels = inputs.to(device), labels.to(device)

        # forward pass
        pred = model(inputs)
        labels = labels.flatten()
        loss = model.loss_function(pred, labels)

        total_loss += loss.item()
        total_correct += (pred.argmax(dim = 1) == labels).sum()   # count correct
predictions

        # backward pass and update parameters
        model.optimizer.zero_grad()
```

```
        loss.backward()
        model.optimizer.step()

    return total_loss, total_correct
```

TRAIN MODEL FOR NUM_EPOCHS

```
def model_train(model, train_DL, num_epochs):
    loss_db = []
    print("Started training")
    start = time.time()

    for i in range(num_epochs):
        loss, correct = model_train_loop(model, train_DL)
        loss_db.append(loss)

        print(f"Epoch {i}: Loss {loss:.4f} Accuracy {correct * 100 / len(train_DL.dataset) :.2f}%")

    print(f"Training took {time.time() - start:.4f}s")
    plot_losses(loss_db)
```

PREDICT FOR VALIDATION DATASET

```
def validate_model(model, val_DL, limit = float('inf')):
    total_correct = 0
    total = 0
    total_masked = 0
    total_masked_correct = 0

    for inputs, labels in val_DL:
        inputs, labels = inputs.to(device), labels.to(device)
        pred = model(inputs)
        labels = labels.flatten()

        # Count total correct predictions
        total_correct += (pred.argmax(dim = 1) == labels).sum()
```

```
        total += len(labels)

        # Count correctly predicted masked images (true positives)
        total_masked += labels.sum()
        total_masked_correct += ((pred.argmax(dim=1) == 1) & (labels == 1)).sum()

        # If limit is given, only predict for images upto limit
        if total >= limit: break

    print(f"Validation accuracy: {total_correct * 100 / total :.2f}%")
    print(f"Accuracy of identifying masked images: {total_masked_correct * 100 / total_masked :.2f}%")
```

ASSIGNING GPU (IF AVAILABLE)

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

PREPARING THE DATA FOR TRAINING AND VALIDATION

```
maskDF = pd.read_csv("mask_df.csv")
# Split data into training and validation sets
train_maskDF, val_maskDF = train_test_split(maskDF, train_size=0.7, random_state=0, stratify=maskDF["Label"])
train_maskDF.reset_index(inplace = True, drop = True)
val_maskDF.reset_index(inplace = True, drop = True)

train_DS = MaskImgDataset(train_maskDF)
val_DS = MaskImgDataset(val_maskDF)

train_DL = DataLoader(train_DS, batch_size=32, shuffle=True, num_workers=2)
val_DL = DataLoader(val_DS, batch_size=32, num_workers=2)
```

CONFIGURING THE LOSS FUNCTION

```
num_non_mask_images = maskDF[maskDF["Label"] == 0].shape[0]
num_mask_images = maskDF[maskDF["Label"] == 1].shape[0]
total_images = num_non_mask_images + num_mask_images

normed_weights = [1 - num_non_mask_images/total_images, 1 - num_mask_images/total_images]
print("Weights for [unmasked, masked]:", normed_weights)

loss_function = nn.CrossEntropyLoss(weight = torch.tensor(normed_weights))

# initializing the model
model = MaskDetector(loss_function)
optimizer = Adam(model.parameters(), lr=0.00001)
model.add_optimizer(optimizer)
model.to(device)

model_train(model, train_DL, 10)
validate_model(model, val_DL, 500)
```

SAVING TRAINED MODEL FOR FUTURE USE

```
torch.save(model.state_dict(), '/kaggle/working/model.pt')
```

LOAD PRE-TRAINED MODEL

```
new_model = MaskDetector(loss_function)
new_model.to(device)
m_state_dict = torch.load('model.pt', map_location=device)
new_model.load_state_dict(m_state_dict)
```

USE MODEL TO PREDICT FOR IMAGES OUTSIDE THE DATASET

```
labels = ["No mask", "Masked"]
def predict_image(model, img_path, ax = None):
    img = read_image(img_path)
    re_img = Resize((100, 100))(img).reshape((1, 3, 100, 100)).float()
    re_img = re_img.to(device)
    pred = model(re_img)
    res = pred.argmax(dim = 1)

    ax.imshow(img.permute((1, 2, 0)))
    ax.set_title(labels[res], fontdict = {'fontsize' : 40})
    ax.axis('off')
    from math import ceil
def predict_real_images(model, img_dir):
    images = os.listdir(img_dir)
    fig, axs = plt.subplots(ceil(len(images)/5), 5, figsize = (50, 10))
    row=0
    col=0
    for i, img in enumerate(images):
        if i%5==0 and i!=0:
            col=0
            row+=1
        predict_image(model, os.path.join(img_dir, img), axs[row][col])
        col+=1

  predict_real_images(new_model, "RealMaskedImages")
  from math import ceil
  def predict_real_images1(model, img_dir):
    images = os.listdir(img_dir)
    fig, axs = plt.subplots(1, 5, figsize = (50, 10))
    for i, img in enumerate(images):
        print(os.path.join(img_dir, img))
        predict_image(model, os.path.join(img_dir, img), axs[i])
```

LOAD THE CASCADE

```
import cv2
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_alt2.xml')

for i in range(5):
   cap = cv2.VideoCapture(0)#starting camera
   ret, frame = cap.read()#reading the frame/capturing
   fname='mask\webcamphoto'+str(i+1)+".jpg"
   cv2.imwrite(fname,frame)#storing the frame
   img = cv2.imread('mask\webcamphoto1.jpg')
```

CONVERT INTO GRAYSCALE

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

DETECT FACES

```
faces = face_cascade.detectMultiScale(gray, 1.1, 4)
```

DRAW RECTANGLE AROUND THE FACES AND CROP THE FACES

```
for (x, y, w, h) in faces:
      cv2.rectangle(img, (x, y), (x+w, y+h), (0, 0, 255), 2)
      faces = img[y:y + h, x:x + w]
      #cv2.imshow("face",faces)
      cv2.imwrite('mask\webcamphoto'+str(i+1)+".jpg", faces)
   cap.release()#disconnecting the cam

predict_real_images1(new_model, "mask")
```