Run Mask Detect on live video

Imports, paths, and info

In [1]:
```
import cv2
import mediapipe as mp
import os
import numpy as np
import PIL
from PIL import Image, ImageOps, ImageDraw
import torch
from torch import nn
from typing import Sequence

```

In [2]:
```
mp_face_detection = mp.solutions.face_detection
mp_drawing = mp.solutions.drawing_utils

target_h = 112
target_w = target_h  # enforce square

# detection constants
BAD_COLOR = (0, 0, 255)  # red
GOOD_COLOR = (0, 128, 0)  # green
BOX_LINE_THICKNESS = 4
```

In [3]:
```
video_fp = r'C:\Users\Andrew\Documents\2022 Summer\Data Mining\Project\Webcam\videos\surgial mask.mov'
#video_fp = r'C:\Users\Andrew\Documents\2022 Summer\Data Mining\Project\Webcam\videos\cloth mask.mov'

processed_video_fp = r'C:\Users\Andrew\Documents\2022 Summer\Data Mining\Project\Webcam\videos\processed.avi'

full_model_path = r'C:\Users\Andrew\Documents\2022 Summer\Data Mining\Project\results\upgrade\full_model_best'
debug_output_dir = r'D:\data\face_mask\webcam\debug'
```

Model and Functions

In [4]:

```python
class CNN(nn.Module):
    def __init__(
        self,
        input_size: Sequence[int] = (3, 112, 112),
        num_classes: int = 2,
        channels: Sequence[int] = (8, 16, 32),
        kernel_sizes: Sequence[int] = (10, 10, 10, 10),
        linear_units: Sequence[int] = (100, 10),
        lr: float = 0.001,
        epochs: int = 10
    ):
        super(CNN, self).__init__()

        self.input_size = input_size
        self.num_classes = num_classes
        self.channels = input_size[0:1] + channels
        self.kernel_sizes = kernel_sizes
        self.linear_units = linear_units
        self.lr = lr
        self.epochs = epochs

        self.flatten = nn.Flatten()
        self.pool = partial(nn.MaxPool2d, kernel_size=2, stride=2)  # first 2 is for 2x2 kernel, second is stride length
        self.dropout = nn.Dropout
        self.activation = nn.ReLU
        self.accuracy = torchmetrics.functional.accuracy
        self.conf_matrix = torchmetrics.functional.confusion_matrix

        # optional, define batch norm here

        # build the convolutional layers
        conv_layers = list()
        for in_channels, out_channels, kernel_size in zip(
            self.channels[:-2], self.channels[1:-1], self.kernel_sizes[:-1]
        ):
            conv_layers.append(
                nn.Conv2d(
                    in_channels=in_channels,
                    out_channels=out_channels,
                    kernel_size=kernel_size,
                    #stride=2,
                    #padding='same',
                )
            )
            conv_layers.append(self.activation())
            conv_layers.append(self.pool())
        # add final layer to convolutions
        conv_layers.append(
            nn.Conv2d(
                in_channels=self.channels[-2],
                out_channels=self.channels[-1],
                kernel_size=self.kernel_sizes[-1],
                stride=2,
                #padding='same',
            )
        )
        conv_layers.append(self.activation())
        conv_layers.append(self.pool())


        # turn list into layers
        self.conv_net = nn.Sequential(*conv_layers)

        # linear layers
        linear_layers = list()
        prev_linear_size = self.channels[-1] * 9  # const scale it correctly
        for dense_layer_size in self.linear_units:
            linear_layers.append(
                nn.Linear(
                    in_features=prev_linear_size,
                    out_features=dense_layer_size,
                )
            )
            linear_layers.append(self.activation())
            prev_linear_size=dense_layer_size

        self.penultimate_dense = nn.Sequential(*linear_layers)
        self.ultimate_dense = nn.Linear(
            in_features=self.linear_units[-1],
            out_features=self.num_classes
        )


    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.conv_net(x)
        x = self.flatten(x)
        # may need to expand dense entry since flatten
        x = self.penultimate_dense(x)
        x = self.ultimate_dense(x)
        return x


def train(dataloader, model, loss_fn, optimizer, verbose=False):
    #model = model.float()  # sometime fixes random obscure type error
    model.train()  # configures for training, grad on, dropout if there is dropout
    size = len(dataloader.dataset)

    for batch, (X, y) in enumerate(dataloader):
        optimizer.zero_grad()

        # compute prediction loss
        preds = model(X)
        loss = loss_fn(preds, y)

        # backprop
        loss.backward()
        optimizer.step()

        if batch % 5 == 0 and verbose:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
    return loss

# for evaluating on validation data too
```

```
def test(dataloader, model, loss_fn, verbose=False):
    model.eval()
    test_loss, correct = 0, 0
    size = len(dataloader.dataset)
    num_batches = len(dataloader)

    with torch.no_grad():
        for X, y in dataloader:

            pred = model(X.float())
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    if verbose:
        print(f"Results: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
    return correct, test_loss

```

In [5]:

```
def correct_crop(xl, xr, yt, yb, w, h):
    if yt < 0:
        diff = abs(yt)
        yt = 0
        expand_left = int(diff / 2)
        expand_right = diff - expand_left
        xl = xl - expand_left
        xr = xr + expand_right
    if xl < 0:
        diff = abs(xl)
        xl = 0
        expand_down = int(diff / 2)
        expand_up = diff - expand_down
        yb = yb + expand_down
        yt = yt - expand_up
    if yb > h:
        diff = yb - h
        yb = h
        expand_left = int(diff / 2)
        expand_right = diff - expand_left
        xl = xl - expand_left
        xr = xr + expand_right
    if xr > w:
        diff = xr - w
        xr = w
        expand_down = int(diff / 2)
        expand_up = diff - expand_down
        yb = yb + expand_down
        yt = yt - expand_up
    if yt < 0 or xl < 0 or yb > h or xr > w:
        print('coords error after correction')
    return xl, xr, yt, yb
```

In [6]:

```
def rect_square_expansion(xl, xr, yt, yb, w, h):
    bbh = yb - yt
    bbw = xr - xl
    if bbh > bbw:
        diff = bbh - bbw
        expand_left = int(diff/2)
        expand_right = diff - expand_left
        xl = xl - expand_left
        xr = xr + expand_right
    elif bbw > bbh:
        diff = bbw - bbh
        expand_down = int(diff/2)
        expand_up = diff - expand_down
        yb = yb + expand_down
        yt = yt - expand_up

    return xl, xr, yt, yb
```

In [7]:

```
# for debug
counter = 0

# classification smoothing
classification = 0.5  # init to no confidence in either direction
smoothing_adaptability = .1  # 1.0 means display most recent detection,
    # 0.0 means do not update classification at all




# load classification model
model = torch.load(full_model_path)
model.eval()

#cap = cv2.VideoCapture(0)
cap = cv2.VideoCapture(video_fp)

frame_width = int(cap.get(3))
frame_height = int(cap.get(4))
size = (frame_width, frame_height)

output_vid_writer = cv2.VideoWriter(processed_video_fp,
                        cv2.VideoWriter_fourcc(*'MJPG'),
                        10, size)

output_vid_writer = cv2.VideoWriter(processed_video_fp,
                        cv2.VideoWriter_fourcc(*"MJPG"),
                        10, size)

with mp_face_detection.FaceDetection(
    model_selection=0, min_detection_confidence=0.5) as face_detection:
      while cap.isOpened():
        success, image = cap.read()
        if not success:
            print("Ignoring empty camera frame.")
            # If loading a video, use 'break' instead of 'continue'.
            break

        # To improve performance, optionally mark the image as not writeable to
        # pass by reference.
        image.flags.writeable = False
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        results = face_detection.process(image)

        # Draw the face detection annotations on the image.
        image.flags.writeable = True
        image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
        if results.detections:
            for detection in results.detections:

                # get height, width and depth of image frame
                h, w, d = image.shape

                # draw face
                #mp_drawing.draw_detection(image, detection)

                # get bounding box and transform normalized coords to pixel coords
                rbb = detection.location_data.relative_bounding_box
                rect_start_point = mp_drawing._normalized_to_pixel_coordinates(
                    rbb.xmin, rbb.ymin, w, h)
                rect_end_point = mp_drawing._normalized_to_pixel_coordinates(
                    rbb.xmin + rbb.width, rbb.ymin + rbb.height, w, h)

                if rect_start_point is not None and rect_end_point is not None:
                    # get individual coordinates from the tuples and create the square
                    xl, yt = rect_start_point
                    xr, yb = rect_end_point
                    xl, xr, yt, yb = rect_square_expansion(xl, xr, yt, yb, w, h)

                    # expand if nessisary
                    expansion = .125
                    bbh = yb - yt
                    bbw = xr - xl
                    amt_to_add = int(expansion * max(bbh, bbw))
                    yt = yt - amt_to_add
                    yb = yb + amt_to_add
                    xl = xl - amt_to_add
                    xr = xr + amt_to_add

                    xl, xr, yt, yb = correct_crop(xl, xr, yt, yb, w, h)
                    xl, xr, yt, yb = correct_crop(xl, xr, yt, yb, w, h)

                    # crop frame to face
                    pil_img = Image.fromarray(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
                    # PIL crop format:  left, top, right, bottom
                    crop = [xl, yt, xr, yb]
                    pil_crop = pil_img.crop(crop)

                    # resize
                    pil_crop = pil_crop.resize((target_h, target_w), resample=PIL.Image.Resampling.HAMMING)

                    # debug
                    #out_path = os.path.join(debug_output_dir, '{}.png'.format(counter))
                    #pil_crop.save(out_path)

                    # turn image into array
                    im_arr = np.array(pil_crop)
                    im_arr = im_arr.reshape((3, 112, 112))
                    im_arr = im_arr.reshape((1, 3, 112, 112))

                    # norm
                    im_arr = im_arr / 255

                    model_input = torch.Tensor(im_arr)
                    raw_pred = model(model_input.float())  # need to add .float()
                    mask_class = raw_pred.argmax(1).item()
                    #print(mask_class)

                    rect_start_point_classification = (xl, yt)
                    rect_end_point_classification = (xr, yb)

                    # smooth out detection
                    classification = smoothing_adaptability * mask_class + (1-smoothing_adaptability) * classification
```

```
                    if classification < 0.5:  # no mask/incorrect mask
                        cv2.rectangle(image, rect_start_point_classification, rect_end_point_classification,
                                      BAD_COLOR, BOX_LINE_THICKNESS)
                    else:
                        cv2.rectangle(image, rect_start_point_classification, rect_end_point_classification,
                                      GOOD_COLOR, BOX_LINE_THICKNESS)



        counter += 1


        output_vid_writer.write(image)

        # Flip the image horizontally for a selfie-view display.
        cv2.imshow('MediaPipe Face Detection', cv2.flip(image, 1))
        if cv2.waitKey(5) & 0xFF == 27:  # if wait 5 miliseconds and 0xFF == 00011011 (always false)??
            break
cap.release()
output_vid_writer.release()
```

```
Ignoring empty camera frame.
```

In [8]:

```
#cap.release()
output_vid_writer.release()

# Closes all the frames
cv2.destroyAllWindows()
```

In [ ]:

```

```