Udacity Machine Learning Engineer Capstone Project

Face Mask Detection

Hiep Doan

Mar 26, 2022

1. Definition

1.1. Project Overview

Since the beginning of 2020, Covid-19 has brought the world to a halt. Starting in Wuhan in December 2019, the coronavirus spread worldwide rapidly. At the time of writing, 281 million people had gotten Covid-19, and 5.4 million people died. In the fight against the novel virus, many preventative measures, including social distancing, masking, and lockdowns, were set in place. Wearing a face mask remains one of the most effective ways to prevent the spread of Covid-19 in public areas. In addition, face mask detection is an excellent application for facial attribute classification problems.

Using the Face Mask Detection dataset from Kaggle, the project aims to explore Deep Learning techniques for detecting face masks.

1.2. Problem Statement

The project's goal is to build a classifier that helps to detect whether a mask is being worn. Given an image of a person in JPEG/PNG format, the system tells whether a person is wearing a mask.

1.3. Metrics

1.3.1 Confusion Matrix

For a binary classification problem, a confusion matrix is useful in summarizing prediction results.

	Actual

Predict		with_mask	without_mask
	with_mask	True Positive (TP)	False Positive (FP)
	without_mask	False Negative (FN)	True Negative (TN)

1.3.2. Relevant Metrics

Metric	Formula	Definition
Precision	$\frac{TP}{TP + FP}$	Proportion of positive identifications was actually correct
Recall	$\frac{TP}{TP + FN}$	Proportion of actual positives was identified correctly
Specificity	$\frac{TN}{TN + FP}$	Proportion of actual negatives was identified correctly
Negative Rate	$\frac{TN}{TN + FN}$	Proportion of negative identifications was actually correct
F1-score	$2 * \frac{Precision * Recall}{Precision + Recall}$	Harmonic mean of precision and recall

Since the cost of misclassifying someone without masks is high, we want to reduce our False Positive rate. **F1-score** is a better metric than accuracy, especially when our data distribution is skewed towards the *with_mask* class (80% of the data). The data distribution is explored in detail in chapter 2.1.1. Data Distribution.

Due to data skew, even if we predict everyone is wearing masks (no true negative), we will achieve decent precision, recall, and f1-score thanks to a high true positive rate. We will also look at **specificity** and **negative rate** metrics to avoid this scenario.

2. Analysis

2.1. Data Exploration And Visualization

The face mask dataset contains 853 images in PNG format with three classes: with a mask, without a mask, and mask worn incorrectly.

The images are in PNG format with the following convention.

```
# id from 0 to 852
maksssksksss{id}.png
```

The annotations are in XML format with the following convention.

```
# id from 0 to 852
maksssksksss{id}.xml
```

An example of annotation objects.

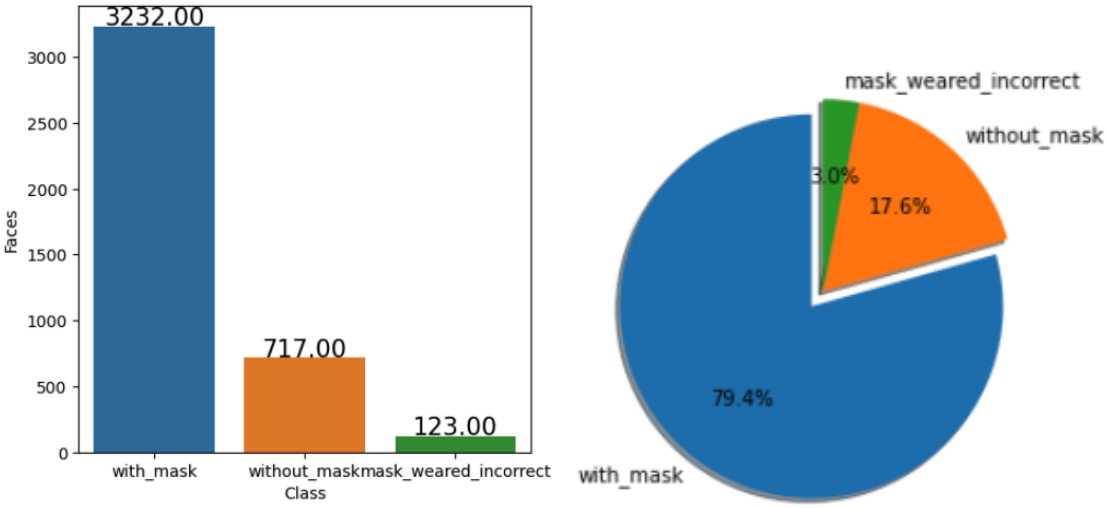
```
<annotation>
    <folder>images</folder>
    <filename>maksssksksss9.png</filename>
    <size>
        <width>267</width>
        <height>400</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>mask_weared_incorrect</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <occluded>0</occluded>
        <difficult>0</difficult>
        <bndbox>
            <xmin>148</xmin>
            <ymin>75</ymin>
            <xmax>201</xmax>
            <ymax>133</ymax>
        </bndbox>
    </object>
    <object>
        <name>without_mask</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
```

```
        <occluded>0</occluded>
        <difficult>0</difficult>
        <bndbox>
            <xmin>27</xmin>
            <ymin>78</ymin>
            <xmax>56</xmax>
            <ymax>106</ymax>
        </bndbox>
    </object>
</annotation>
```

The following sample images show how bounding boxes look like on original images when they are annotated with green for with_mask class, orange for mask_weared_incorrect class, and red for without_mask class.

2.1.1. Data distribution

We iterate through all XML annotation files and count the number of faces for each class.

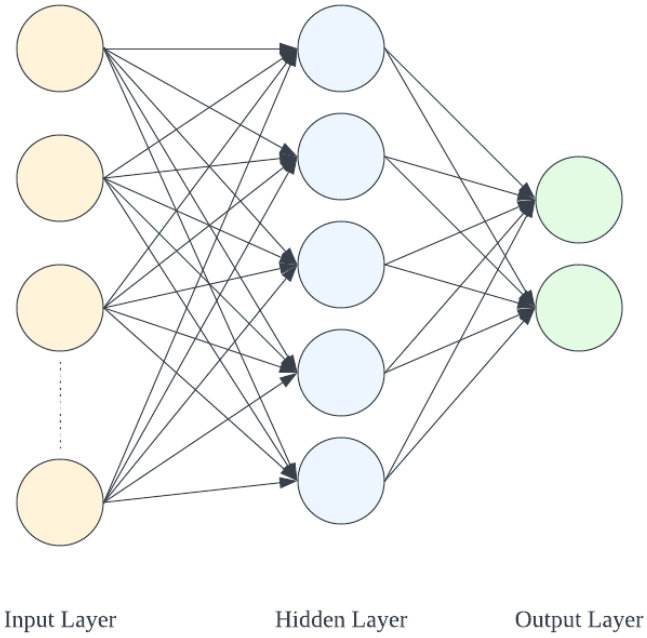


The pie chart and bar chart shows the distribution for this dataset. There are 4072 annotated faces in these images. Out of which, 3232 (79.4%) are with masks, 717 (17.6%) are without masks, and 123 (3%) are with incorrectly worn masks.

2.2. Algorithms and Techniques

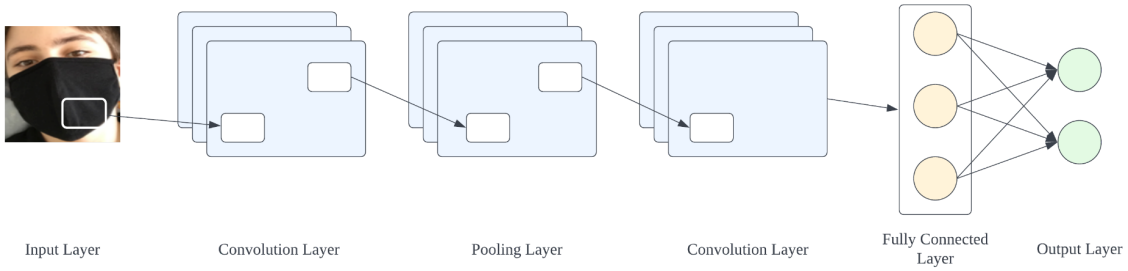
With the number of features in the image classification problem, feature extraction is challenging and time-consuming for most Machine Learning algorithms. Easing feature extraction makes Deep Learning and its class of neural networks an excellent choice for this problem.

Artificial Neural Network (ANN) is limited in image data due to the number of trainable parameters. With our image size of 224*224 with 3 color channels, there are 752640 trainable parameters being passed to a hidden layer of just 5 neurons. Training such a neural network is extremely time-consuming.



Artificial Neural Network Architecture

Convolutional Neural Network (CNN) solves this problem with Convolution Layers. While all layers in ANN are fully connected, CNN uses Convolution Layers to extract relevant features from image data before passing these features to fully connected layers. The secret behind Convolution Layers is kernel or filter, which helps us retain important information while reducing the number of features.



Convolutional Neural Network

Transfer Learning allows us to leverage pre-trained CNN models and use their weights trained on larger datasets. It saves us time and resources from training the Convolution Layers again and often gives promising results.

This project explores CNN and Transfer Learning to detect face masks.

2.3. Benchmark

The project uses a benchmark of **80% for f-score and 60% for specificity and negative rate**.

3. Methodology

3.1. Data Preprocessing

3.1.1. Image Processing (Cropping and Resizing)

Annotated faces in the raw input images are cropped and resized to *224*224* pixels. For each image, we iterate through bounding boxes in the corresponding annotation XML file and crop out faces from those bounding boxes. They are resized and saved under a directory for processed images with *{id}.png* as the format for file names. *{id}* is produced by keeping a counter when processing input images.

```
from PIL import Image
from torchvision import transforms

class DataPreprocess:

    def __init__(self, processed_dir, image_size):
        self.count = 0
        self.labels = []
        self.processed_dir = processed_dir
        self.transform = transforms.Compose([transforms.Resize((image_size,
image_size))])
        Path(self.processed_dir).mkdir(parents=True, exist_ok=True)

    def _preprocess_image(self, image, processed_path: str, bndbox: dict) -> None:
        xmin, ymin, xmax, ymax = int(bndbox['xmin']), int(bndbox['ymin']),
int(bndbox['xmax']), int(bndbox['ymax'])
        # Crop image
        crop = image.crop((xmin, ymin, xmax, ymax))
        crop = self.transform(crop)
        # Save cropped image
        crop.save(processed_path)
```

The following example shows how faces are cropped from the original input image and resized.

Raw input image

Processed facial images

3.1.2. Saving Labels

As discussed, an annotation object contains a name and a bounding box. Names are with_mask, without_mask, and mask_weared_incorrect and are used for labeling.

```
    <object>
        <name>without_mask</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <occluded>0</occluded>
        <difficult>0</difficult>
        <bndbox>
            <xmin>27</xmin>
            <ymin>78</ymin>
            <xmax>56</xmax>
            <ymax>106</ymax>
        </bndbox>
    </object>
```

During image processing, bounding boxes are extracted into individual facial images. The corresponding names are extracted from annotation objects to labels using the following mapping.

```
label_map = {
    'with_mask': 1, # wearing mask
    'without_mask': 0, # not wearing mask
    'mask_weared_incorrect': -1, # skip this label
}
```

The dataset provides 3 classes (*with_mask*, *without_mask*, and *mask_weared_incorrect*) that are possible to use as labels. Due to data skewness and the fact that only 3% of the data are *mask_weared_incorrect*, we skip *mask_weared_incorrect* class and use two classes: *wearing_mask* (label 1) and *not_wearing_mask* (label 0).

Extracted and mapped labels are stored in CSV format. The row numbers are the ids of processed images and the single column values are the labels.

```
0
1
0
1
1
1
1
1
1
1
...
```

3.2. Implementation

3.2.1. Load Processed Images

Each processed facial image of size 224*224 is read into a NumPy array with a shape of (224, 224, 4). After dropping the alpha channel and reordering, the array has a shape of (3, 224, 224), which presents 224*224 pixels and *3* color channels. Since each pixel value has a range of (0, 255), they are normalized to a continuous range of (0, 1) by dividing by 255.

```
import numpy as np
from PIL import Image

x = []

for index in range(3949):
    # defining the image path
    image_path = f'{preprocessed_path}/images/{index}.png'
    # reading the image
    img = Image.open(image_path)
    img = np.array(img)
    # dropping alpha channel
    img = img[:, :, :3]
    # change image channel ordering
    img = np.moveaxis(img, -1, 0)
    # converting the type of pixel to float 32
    img = img.astype('float32')
    # normalizing the pixel values
    img /= 255.0
    # appending the image into the list
    x.append(img)
```

3.2.2. Load Labels

Labels are read from the preprocessed annotation CSV file.

```
labels = []
with open(f'{preprocessed_path}/annotation.csv', 'r', newline='') as f:
    csv_reader = csv.reader(f, delimiter=',')
    for row in csv_reader:
        labels.append(int(row[0]))
```

3.2.3. Split Training and Testing Data

Training and testing data is split with a 70:30 ratio using *sklearn* library, which results in 2850 images for training and 1222 images for testing.

```
from sklearn.model_selection import train_test_split


# splitting training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

The data is also transformed from NumPy arrays to Tensors.

```
import torch


X_train, X_test, y_train, y_test = torch.from_numpy(X_train),
torch.from_numpy(X_test), torch.from_numpy(y_train), torch.from_numpy(y_test)
```

3.2.4. Build CNN Model

We define a CNN with 3 Convolution Layers and 1 Fully Connected Layer. The Convolution Layers increase input channels from 3 to 4 to 8 to 16. Each Convolution Layer is paired with a Batch Normalization Layer, a ReLU activation function, and a Pooling Layer. The Batch Normalization Layers are used to accelerate the speed of the model training by eliminating internal covariate shift problem. The Pooling Layers are used to reduce the dimensions of input features, which reduces the number of trainable parameters. These Pooling Layers decrease the dimension of the input features from 224*224 to 112*112 to 56*56 to 28*28.

The Fully Connected layer is a linear that takes 16*28*28=12544 input features and output 2 features for binary classification.

The forward method passes output from Convolution layers to the Fully Connected layer.

```
from torch import nn


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()


        self.cnn_layers = nn.Sequential(
            # Layer 1: (*, 3, 224, 224) (Conv2d)-> (*, 4, 224, 224) (Pool)-> (*, 4,
112, 112)
            nn.Conv2d(3, 4, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(4),
```

```
            nn.ReLU(inplace=False),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Layer 2: (*, 4, 112, 112) (Conv2d)-> (*, 8, 112, 112) (Pool)-> (*, 8,
56, 56)
            nn.Conv2d(4, 8, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(inplace=False),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Layer 3: (*, 8, 56, 56) (Conv2d)-> (*, 16, 56, 56) (Pool)-> (*, 16,
28, 28)
            nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=False),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        # Fully connected layer
        self.linear_layers = nn.Sequential(
            nn.Linear(16 * 28 * 28, 2)
        )

    # Defining the forward pass
    def forward(self, x):
        x = self.cnn_layers(x)
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        return x
```

3.2.5. Training

3.2.5.1. Initialize Model

We use the popular Adam optimizer with a learning rate of 0.07 and Cross Entry for loss function.

One thing to highlight is the weights for our classes. Since our data is skewed towards the *with_mask (1)* class (79.4%), training the model with equal weights results in favor of the dominating class. To avoid this, we give a weight of 60/100 to the *without_mask (0)* class and

40/100 to the *with_mask (1)* class. The chosen weights are explained further under the Fine Tuning section.

```
import torch
from torch import nn
from torch.optim import Adam

# defining the model
model = Net()
# defining the optimizer
optimizer = Adam(model.parameters(), lr=0.07)
# defining the loss function
criterion = nn.CrossEntropyLoss(weight=torch.tensor([60.0, 40.0]))
# checking if GPU is available
device = "cuda" if torch.cuda.is_available() else "cpu"
```

3.2.5.2. Define Training Method

The training method performs back-propagation to train our model and shows the validation loss for *epochs* times.

```
losses = []

def train(epochs):
    batch_size = 128

    for epoch in range(epochs):
        inputs, labels = Variable(X_train), Variable(y_train)

        # clearing the Gradients of the model parameters
        optimizer.zero_grad()

        # prediction for training and validation set
        outputs = model(inputs)

        # computing the training and validation loss
        loss = criterion(outputs, labels)
```

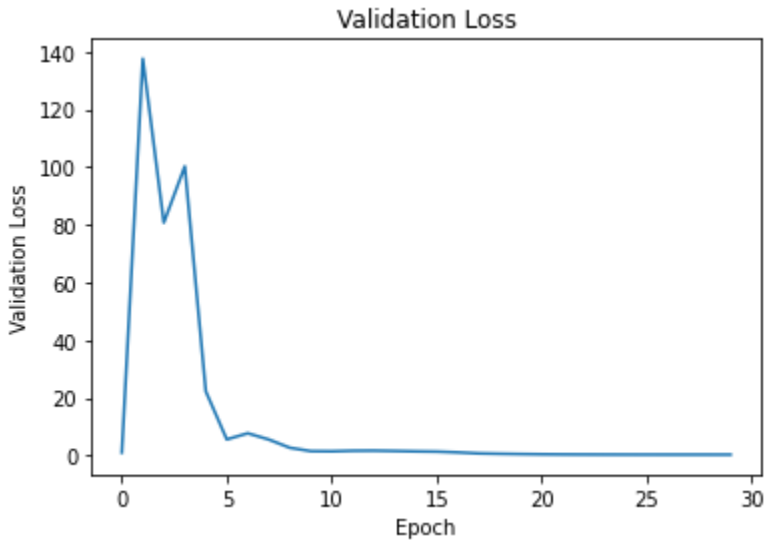
```
        # computing the updated weights of all the model parameters
        loss.backward()
        optimizer.step()

        # printing the validation loss
        print('Epoch : ', epoch+1, '\t', 'loss :', loss)

        losses.append(loss)
```

3.2.5.3. Train Model

We call the training method with 30 iterations. We observe the validation loss plateaus after the 24th iteration.




```
train(30)


Epoch :  1     loss : tensor(1.0368, grad_fn=<NllLossBackward0>)
Epoch :  2     loss : tensor(137.5457, grad_fn=<NllLossBackward0>)
Epoch :  3     loss : tensor(80.7127, grad_fn=<NllLossBackward0>)
Epoch :  4     loss : tensor(100.2310, grad_fn=<NllLossBackward0>)
Epoch :  5     loss : tensor(22.2846, grad_fn=<NllLossBackward0>)
Epoch :  6     loss : tensor(5.6183, grad_fn=<NllLossBackward0>)
```

```
Epoch :  7    loss : tensor(7.7409, grad_fn=<NllLossBackward0>)
Epoch :  8    loss : tensor(5.5863, grad_fn=<NllLossBackward0>)
Epoch :  9    loss : tensor(2.7331, grad_fn=<NllLossBackward0>)
Epoch :  10   loss : tensor(1.5610, grad_fn=<NllLossBackward0>)
Epoch :  11   loss : tensor(1.5031, grad_fn=<NllLossBackward0>)
Epoch :  12   loss : tensor(1.6836, grad_fn=<NllLossBackward0>)
Epoch :  13   loss : tensor(1.7230, grad_fn=<NllLossBackward0>)
Epoch :  14   loss : tensor(1.6098, grad_fn=<NllLossBackward0>)
Epoch :  15   loss : tensor(1.4746, grad_fn=<NllLossBackward0>)
Epoch :  16   loss : tensor(1.3880, grad_fn=<NllLossBackward0>)
Epoch :  17   loss : tensor(1.1198, grad_fn=<NllLossBackward0>)
Epoch :  18   loss : tensor(0.7721, grad_fn=<NllLossBackward0>)
Epoch :  19   loss : tensor(0.6440, grad_fn=<NllLossBackward0>)
Epoch :  20   loss : tensor(0.5420, grad_fn=<NllLossBackward0>)
Epoch :  21   loss : tensor(0.4645, grad_fn=<NllLossBackward0>)
Epoch :  22   loss : tensor(0.4149, grad_fn=<NllLossBackward0>)
Epoch :  23   loss : tensor(0.3866, grad_fn=<NllLossBackward0>)
Epoch :  24   loss : tensor(0.3681, grad_fn=<NllLossBackward0>)
Epoch :  25   loss : tensor(0.3569, grad_fn=<NllLossBackward0>)
Epoch :  26   loss : tensor(0.3501, grad_fn=<NllLossBackward0>)
Epoch :  27   loss : tensor(0.3461, grad_fn=<NllLossBackward0>)
Epoch :  28   loss : tensor(0.3442, grad_fn=<NllLossBackward0>)
Epoch :  29   loss : tensor(0.3428, grad_fn=<NllLossBackward0>)
Epoch :  30   loss : tensor(0.3405, grad_fn=<NllLossBackward0>)
```

3.2.6. Evaluation

We set the model to evaluation mode and get the prediction results by running the model against test data.

```
model.eval()
test_inputs, test_labels = Variable(X_test), Variable(y_test)
prediction = model(test_inputs)
```

We define a method that takes the number of true positive instances, number of false-positive instances, number of true negative instances, and number of false-negative instances.

```
def calculate_metrics(tp: int, fp: int, tn: int, fn: int):
    precision, recall, specificity, negative_rate = tp / (tp+fp), tp / (tp+fn), tn / (tn+fp), tn / (fn+tn)
    f1 = 2 * (precision*recall) / (precision + recall)
    print("Precision: {:0.2f}%\nRecall: {:0.2f}%\nSpecificity: {:0.2f}%\nNegative Rate: {:0.2f}%\nF1: {:0.2f}%".format(precision*100, recall*100, specificity*100, negative_rate*100, f1*100))
```

The method shows the following evaluation metrics.

- Precision: 91.98%
- Recall: 92.17%
- Specificity: 66.08%
- Negative Rate: 66.67%
- F1: 92.08%

Given our benchmark of 80% for f-score and 60% for specificity and negative rate, the model performs well on f-score and meets the mark for specificity and negative rate. The evaluation shows that the model is able to make a positive prediction when a mask is being worn with a high precision.

3.3. Refinement

3.3.1. Fine-tune CNN model

We fine-tune the CNN model with the number of convolution layers and class weights for the loss function.

The initial CNN model has 2 convolution layers with equal class weights for the loss function. The final model has 3 convolution layers with 60/100 weight for the *without_mask (0)* class and 40/100 weight for the *with_mask (1)* class.

All tested configurations and their performances are reported in the following table.

The metrics are color-coded as follows:

- Red: 0% - 50%
- Yellow: 50% - 80%
- Green: 80% - 100%

Setup	**Precision**	**Recall**	**Specificity**	**Negative Rate**	**F1**
2 Convolution Layers: - Layer 1: **(*, 3, 224, 224)** (Conv2d)-> (*, 4, 224, 224) (Pool)-> (*, 4, 112, 112) - Layer 2: (*, 4, 112, 112) (Conv2d)-> (*, 8, 112, 112) (Pool)-> **(*, 8, 56, 56)** Class Weights: (50.0, 50.0)	78.70%	92.28%	41.85%	69.95%	84.95%
3 Convolution Layers: - Layer 1: **(*, 3, 224, 224)** (Conv2d)-> (*, 4, 224, 224) (Pool)-> (*, 4, 112, 112) - Layer 2: (*, 4, 112, 112) (Conv2d)-> (*, 8, 112, 112) (Pool)-> (*, 8, 56, 56) - Layer 3: (*, 8, 56, 56) (Conv2d)-> (*, 16, 56, 56) (Pool)-> **(*, 16, 28, 28)** Class Weights: (50.0, 50.0)	75.21%	97.86%	44.98%	92.49%	85.05%
3 Convolution Layers: - Layer 1: **(*, 3, 224, 224)** (Conv2d)-> (*, 4, 224, 224) (Pool)-> (*, 4, 112, 112) - Layer 2: (*, 4, 112, 112) (Conv2d)-> (*, 8, 112, 112) (Pool)-> (*, 8, 56, 56) - Layer 3: (*, 8, 56, 56) (Conv2d)-> (*, 16, 56, 56) (Pool)-> **(*, 16, 28, 28)** Class Weights: (70.0, 30.0)	52.24%	99.04%	29.43%	97.51%	68.40%
2 Convolution Layers: - Layer 1: **(*, 3, 224, 224)** (Conv2d)-> (*, 4, 224, 224) (Pool)-> (*, 4, 112, 112) - Layer 2: (*, 4, 112, 112) (Conv2d)-> (*, 8, 112, 112) (Pool)-> **(*, 8, 56, 56)** Class Weights: (60.0, 40.0)	75.41%	94.95%	42.13%	81.69%	84.06%
3 Convolution Layers: - Layer 1: **(*, 3, 224, 224)** (Conv2d)-> (*, 4, 224, 224) (Pool)-> (*, 4, 112, 112) - Layer 2: (*, 4, 112, 112) (Conv2d)-> (*, 8, 112, 112) (Pool)-> (*, 8, 56, 56) - Layer 3: (*, 8, 56, 56) (Conv2d)-> (*,	91.98%	92.17%	66.08%	66.67%	92.08%

16, 56, 56) (Pool)-> **(*, 16, 28, 28)** Class Weights: (60.0, 40.0)					

3.3.2. Transfer Learning

For image classification, Transfer Learning is a powerful method to leverage existing pre-trained models. We use MobileNetV3 and replace the last fully connected layer with our own for binary classification.

3.3.2.1. Update model

We replace the model's last fully connected layer with our own to support binary classification since the model was originally trained to classify 1000 classes. Since MobileNetV3 has about 3 million trainable parameters, we freeze the rest of the model weights to avoid expensive re-training.

```
from torch import nn
from torchvision import models


tl_model = models.mobilenet_v3_small(pretrained=True)
# freeze model weights
for param in tl_model.parameters():
    param.requires_grad = False
# redefine last layer to support binary classification
tl_model.classifier[-1] = nn.Sequential(nn.Linear(1024, 2))
for param in tl_model.classifier[-1].parameters():
    param.requires_grad = True
```

3.3.2.2. Initialize Model

Same as our CNN model, we use the Adam optimizer with a learning rate of 0.07 and Cross Entry for loss function.

```
import torch
from torch import nn
from torch.optim import Adam


```

```
# defining the optimizer
tl_optimizer = Adam(tl_model.parameters(), lr=0.07)
# defining the loss function
tl_criterion = nn.CrossEntropyLoss()
```

3.3.2.3. Define Training Method

The training method performs back-propagation to train our model and shows the validation loss for *epochs* times.

```
def train(epochs):
    batch_size = 128

    for epoch in range(epochs):
        inputs, labels = Variable(X_train), Variable(y_train)

        # clearing the Gradients of the model parameters
        tl_optimizer.zero_grad()

        # prediction for training and validation set
        outputs = tl_model(inputs)

        # computing the training and validation loss
        loss = tl_criterion(outputs, labels)

        # computing the updated weights of all the model parameters
        loss.backward()
        tl_optimizer.step()

        # printing the validation loss
        print('Epoch : ', epoch+1, '\t', 'loss :', loss)
```

3.3.2.4. Training

Same as our CNN model, we call the training method with 30 iterations.

```
train(30)
```

```
Epoch :  1    loss : tensor(0.7144, grad_fn=<NllLossBackward0>)
Epoch :  2    loss : tensor(0.4658, grad_fn=<NllLossBackward0>)
Epoch :  3    loss : tensor(0.3442, grad_fn=<NllLossBackward0>)
Epoch :  4    loss : tensor(0.2891, grad_fn=<NllLossBackward0>)
Epoch :  5    loss : tensor(0.2442, grad_fn=<NllLossBackward0>)
Epoch :  6    loss : tensor(0.2495, grad_fn=<NllLossBackward0>)
Epoch :  7    loss : tensor(0.2399, grad_fn=<NllLossBackward0>)
Epoch :  8    loss : tensor(0.2422, grad_fn=<NllLossBackward0>)
Epoch :  9    loss : tensor(0.2173, grad_fn=<NllLossBackward0>)
Epoch :  10   loss : tensor(0.2612, grad_fn=<NllLossBackward0>)
Epoch :  11   loss : tensor(0.1964, grad_fn=<NllLossBackward0>)
Epoch :  12   loss : tensor(0.1942, grad_fn=<NllLossBackward0>)
Epoch :  13   loss : tensor(0.1510, grad_fn=<NllLossBackward0>)
Epoch :  14   loss : tensor(0.1851, grad_fn=<NllLossBackward0>)
Epoch :  15   loss : tensor(0.1848, grad_fn=<NllLossBackward0>)
Epoch :  16   loss : tensor(0.1752, grad_fn=<NllLossBackward0>)
Epoch :  17   loss : tensor(0.1656, grad_fn=<NllLossBackward0>)
Epoch :  18   loss : tensor(0.1584, grad_fn=<NllLossBackward0>)
Epoch :  19   loss : tensor(0.1497, grad_fn=<NllLossBackward0>)
Epoch :  20   loss : tensor(0.1518, grad_fn=<NllLossBackward0>)
Epoch :  21   loss : tensor(0.1532, grad_fn=<NllLossBackward0>)
Epoch :  22   loss : tensor(0.1341, grad_fn=<NllLossBackward0>)
Epoch :  23   loss : tensor(0.1311, grad_fn=<NllLossBackward0>)
Epoch :  24   loss : tensor(0.1389, grad_fn=<NllLossBackward0>)
Epoch :  25   loss : tensor(0.1178, grad_fn=<NllLossBackward0>)
Epoch :  26   loss : tensor(0.0953, grad_fn=<NllLossBackward0>)
Epoch :  27   loss : tensor(0.1173, grad_fn=<NllLossBackward0>)
Epoch :  28   loss : tensor(0.1039, grad_fn=<NllLossBackward0>)
Epoch :  29   loss : tensor(0.1023, grad_fn=<NllLossBackward0>)
Epoch :  30   loss : tensor(0.1081, grad_fn=<NllLossBackward0>)
```

3.3.2.5. Evaluation

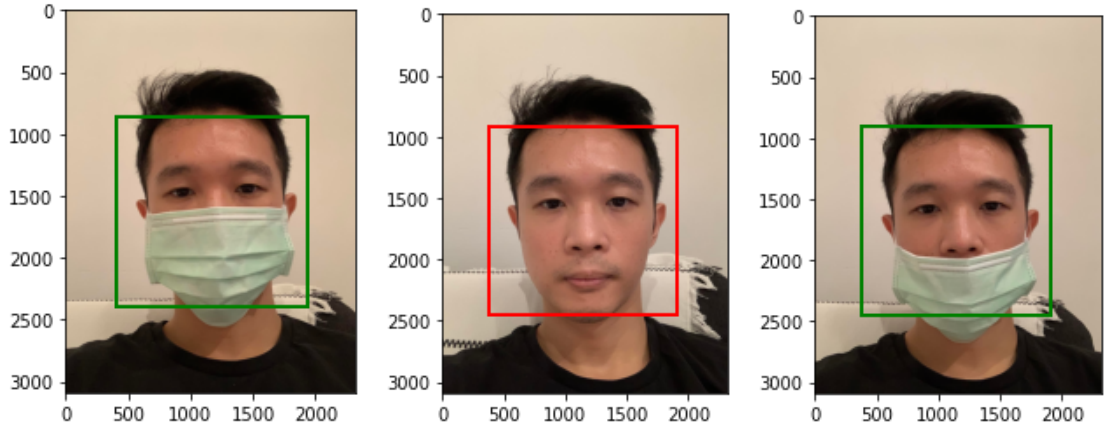
We set the model to evaluation mode and get the prediction results by running the model against test data. From the numbers in our confusion matrix, the following metrics are derived.

- Precision: 69.79%
- Recall: 96.26%
- Specificity: 40.70%
- Negative Rate: 88.44%
- F1: 80.92%

Even though our MobileNetV3 model performs well for negative rate and recall, its f1-score falls behind our CNN model. This can be due to the fact that only weights from the last fully connected layer are being trained and the model wasn't trained with mask data.

4. Application

To further evaluate our model in real-life scenarios, we deploy a Haar Cascade classifier to detect faces from real-life images. We use a pre-built frontal cascade classifier. With the faces extracted from the images, our CNN model classifies whether a mask is being worn. The result is annotated on the original images where green presents a with_mask prediction, and red presents a without_mask prediction.




```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import cv2
import imutils
import torch
from torch.autograd import Variable
from PIL import Image
```

```
# Use prebuilt frontal face cascade classifier
frontal_detector =
cv2.CascadeClassifier('./app/haarcascade_frontalface_default.xml')
# Define a torch vision transform to resize the images
transform = transforms.Compose([transforms.Resize((image_size, image_size))])

def predict(path: str, detector) -> None:
    # run detection
    image = cv2.imread(path)
    # image = imutils.resize(image, width=500)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rects = detector.detectMultiScale(gray, scaleFactor=2, minNeighbors=9,
minSize=(30, 30), flags=cv2.CASCADE_SCALE_IMAGE)
    if len(rects) != 1:
        raise Exception(f'There are {len(rects)} face(s) instead of 1 face detected
for {path}')

    # crop and transform to tensor
    x, y, w, h = rects[0]
    img = Image.open(path)
    crop = img.crop((x, y, x+w, y+h))
    crop = transform(crop)
    img_in = np.array(crop)
    img_in = img_in[:, :, :3]
    img_in = np.moveaxis(img_in, -1, 0)
    img_in = img_in.astype('float32')
    img_in /= 255.0

    # predict with model
    inp = np.array([img_in])
    inp = torch.from_numpy(inp)
    inp = Variable(inp)
    out = model(inp)
    pred = torch.max(out, dim=1)[1][0]
```

```
    # show results
    res = 'mask' if pred else 'no mask'
    color = 'green' if pred else 'red'
    print(out, pred, res)

    # annotate
    fig, ax = plt.subplots()
    ax.imshow(img)
    patch = patches.Rectangle((x, y), w, h, linewidth=2, edgecolor=color,
facecolor='none')
    ax.add_patch(patch)
    plt.show()
```

The application shows the CNN model able to make correct predictions using real-life images.

For future improvements, we can detect whether a person is wearing masks correctly and run our prediction in real-time on a video feed.

5. Results

5.2. Justification

Precision	Recall	Specificity	Negative Rate	F1
91.98%	92.17%	66.08%	66.67%	92.08%

Our final CNN model performs well on overall scores with 92.08% f1-score and ~66% on both specificity and negative rate given a benchmark of 80% f1-score and 60% specificity and negative rate.

High precision and specificity serve our goal of avoiding misclassifying someone without masks. With high recall, we are confident of detecting those with masks.

5.3. Model Evaluation and Validation

The final CNN model has 3 Convolution + BatchNorm2d + Pooling Layers and 1 Fully Connected Layer. For loss function, class weights of 60 (label 0) and 40 (label 1) are

used. The model was chosen after the fine-tuning process reported in chapter 3.3.1. Fine-tune CNN Model. By comparing this final model with other CNN models and with Transfer Learning, it's concluded that this model works best given the dataset and the use case.

Chapter 4. Application shows how the model is tested in an application that feeds real-life images to the model.

6. References

[1] Kaiming He., Xiangyu Zhang., Shaoqing Ren., & Jian Sun. *Deep Residual Learning for Image Recognition*.
[2] Preeti Nagrath., Rachna Jain., Agam Madan., Rohan Arora., Piyush Kataria., & Jude Hemanth (2020). *SSDMNV2: A real time DNN-based face mask detection system using single shot multibox detector and MobileNetV2*.
[3] PulkitS (2020). *How to Train an Image Classification Model in PyTorch and TensorFlow*.
[4] PulkitS (2019). *Build an Image Classification Model using Convolutional Neural Networks in PyTorch*.
[5] PulkitS (2019). *Deep Learning for Everyone: Master the Powerful Art of Transfer Learning using PyTorch*.