Applying Machine Learning Algorithms for Face Mask Detections

Mackenzie Frato
mbf20@uakron.edu

Honors Research Project

Applying Machine Learning Algorithms for Face Mask Detection

Mackenzie Frato

The University of Akron

Spring 2022

Table of Contents

Abstract

In this paper, I will be comparing Machine Learning methods – focusing on Convolution Neural Networks – to predict Face Mask Detection in static images. This is done by testing multiple packages on the same image dataset and comparing the accuracy of these models to correctly classify if the Individual in the image is wearing a Face Mask or Not. The models were cross validated then compared visually and analytically based on accuracy and loss measures. The models all had strong accuracy once cross-validated and run over a maximum of 15 epochs, but the strongest results came from the MobileNet and InceptionV3 [Keras Team] packages. The code was written in Python, so only Python packages were considered. This suggests that multiple models may work on the same dataset, so testing different methodology is useful if there is computational time allowable.

Introduction

This project focuses on Image Detection using Machine Learning Methodology. To remain timely, the images are focused on Face Mask Detection. Some of the images contain Individual(s) with Face Masks and other contain Individual(s) without Face Masks. Masks are useful for mitigating the spread of COVID-19 and its variants [Larxel]. Thus, it has become an emerging issue of how mask usage is tracked or verified. Some machines use real-time facial recognition to detect then classify if a user is wearing a mask or not wearing a mask. This project explores how Machine Learning can be used to do facial recognition in this case. While real-time facial recognition is interesting, it is much more difficult to explain how data is classified in real time. The project will discuss how data is classified and compare Machine Learning methodologies for doing so for non-real-time Facial Detection. This type of image detection can be applied to other user cases as well.

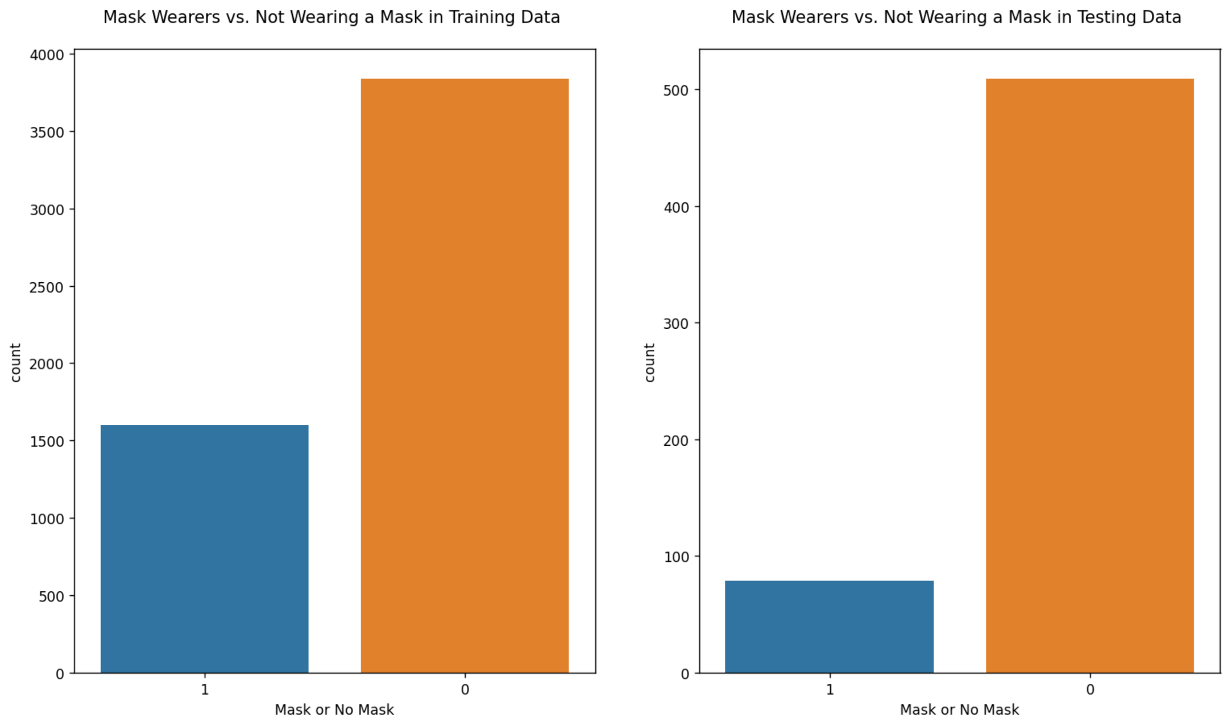

Figure 1. The Distribution of Images in Each Class (0 or 1) is shown here for the Training set and the Testing dataset. There are many more images in the “Individual(s) without a Mask” in both sets than in the “Individual(s) with a Mask” set.

The dataset used in this project is a large dataset of images in two classes: Individual(s) wearing a Mask (1) and Individual(s) not wearing a Mask (0). These images were presorted into a Training and Testing set. The training set has 3,839 images in class 0 - Individual(s) not wearing a Mask - and 747 images in class 1 - Individual(s) wearing a Mask. The testing set has 509 images in class 0 - Individual(s) not wearing a Mask - and 79 images in class 1 - Individual(s) wearing a Mask. The total size of the dataset is 5,174 images with most of them falling in the training set with Individual(s) not wearing a Mask. Due to the nature of using images as your data, it is difficult to create Descriptive Statistics around them, however, it is easy to look at the Distribution of the data as seen in Figure 1. The samples are independent from each other, as none of the images contain more than one person and there are no “in-between” cases used in this set where a person may be incorrectly wearing their mask.

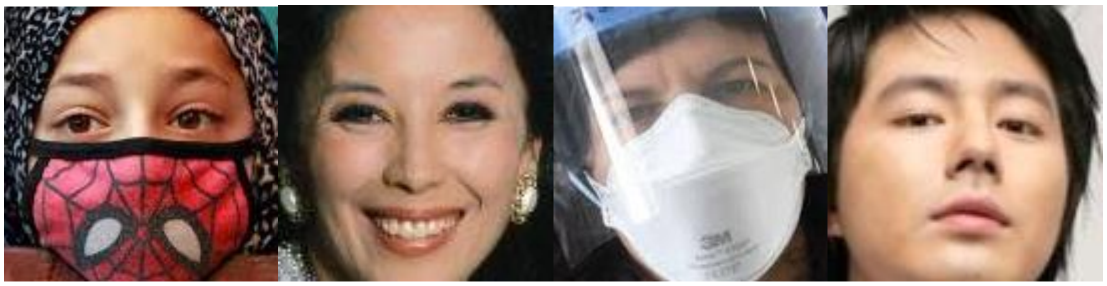

Figure 2. An example of some of the images within this dataset. Two individuals are wearing a mask and two individuals are not wearing a mask [Larxel].

These images are very close to the subject’s face to focus on whether they are wearing a Face Mask. Although these images are simple, due to the large quantity of images in the dataset and the processing power required to handle this type of Regression, the images must be augmented – or simplified – to be used within the following Machine Learning methods. There were 400 Megabytes of images within the dataset so running multiple models over all this data was computationally challenging. This is discussed more in the Augmentation section of the report (6).

The Machine Learning packages used in this paper are: VGG19 [“Keras Documentation: VGG16 and VGG19.”], ConvNet [Torch Contributors], InceptionV3 [Keras Team], MobileNet [Keras Team], and DenseNet [Keras Team]. Most of these packages are based on the concept of Convolution Neural Networks. These are an extension of Neural Networks that are focused on differentiating images from one another – they work especially well in our case of classification of images. The main benefit of CNN is that it can capture Spatial and Temporal dependencies in an image that are not detectable in other types of Neural Networks.

The algorithms themselves are interesting, but the main purpose of using all these different methods is to compare them. The later sections of the paper will discuss which of these methods is best suited to this type of data and why that is such. These methods may not be most optimized for every use-case, but once we identify the best method for this data, it could be expanded into real-life applications like real-time Facial Detection or surveillance reviews.

Data Preparation and Augmentation

The data preparation for this project was a lot of trial and error. The dataset was originally uploaded to Kaggle in the form of nested folders. As I have never worked with images before, this was a very different experience for me. There were many methods that I tried to use to properly import the data before arriving at my final method. Finally, I ended up learning the os package in Python to import the data effectively. The Appendix includes the code for this importing process, but I found it very interesting. First, the paths to the folders were defined separately, as there are 4 final file folders and a directory that holds all of them. Then, data frames were filled with the images and their separate classes. This process required the testing and training data to be in separate data frames, but the classes were preserved inside. For the class, each element of the separate folders for each class was multiplied by the value of its class; then for the images, they were imported within a loop of the length of each folder for the separate classes [Paialunga]. The data was correctly imported with the proper class assigned to each as tested within the code and seen in the next section with Descriptive Statistics.

Once the descriptive statistics were taken for the data, then before they were augmented, I sent the data into catalogues. To do this, I created new paths to the directories of the training and testing data. From there, the data catalogues were created by copying the data from each class while considering their location within the directory. The data catalogue creates metadata for each of the images like its type, location and in this example, class. The augmentation is then done on this catalogue to avoid causing irreversible damage to the original images and create new augmented images separately. It functions similarly to passing by reference.

The augmentation itself is simple in code, but more complex in structure. The code simply pulls the data from the directory and defines new sizes for the image to take – often scaling the image down. The main consideration to take in this step is that because we created a directory to create new images from, if the code is run multiple times, the augmented images may return in the new directory and be considered part of the testing and training data when they should not be [Tran].

There are many reasons to augment your images before running the algorithms on the data. Previously, I discussed the processing time/space saving benefits of augmentation, but augmentation itself provides benefits to the model. Establishing the base size for the images is the first step, but then those images can be augmented by creating images in new orientations. This aids in generating more data from a small sample size and helps reduce the risk of overfitting the model. There are many techniques of augmenting images like: greyscaling, rotating, Gaussian blurring, and reflection [Dennanni]. These augmentation techniques are very helpful before running many algorithms but are not always required for algorithms like Convolution Neural Networks. The main reason I choose to augment this data was to create a larger dataset from the smaller one and to save on processing time for my computer.

Convolution Neural Networks

The main way that images are processed with Machine Learning packages are through Convolution Neural Networks. With images, there is a lot more information stored within them rather than just a matrix of numbers representing variables. Images are generally made of three-color planes – RBG or other types of image formats like Greyscale, HSV, CMYK – which can be difficult to process. The goal of the Convolution Neural Network is to make the images easier to process while not losing important aspects of the image.

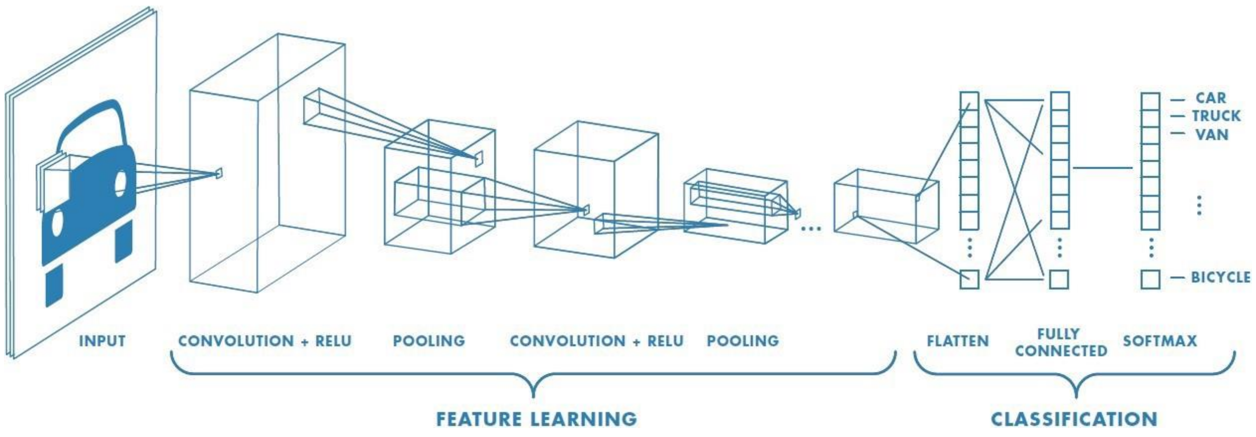

Figure 3. The image above shows the general steps of a Convolution Neural Network. The main process is split into two steps – Feature Learning and Classification. The Feature Learning process is made of Convolution + Relu and Polling repeated n times. Then Classification is made of the Flattened Layer, then Connects the layers, and then gives the Output [Saha].

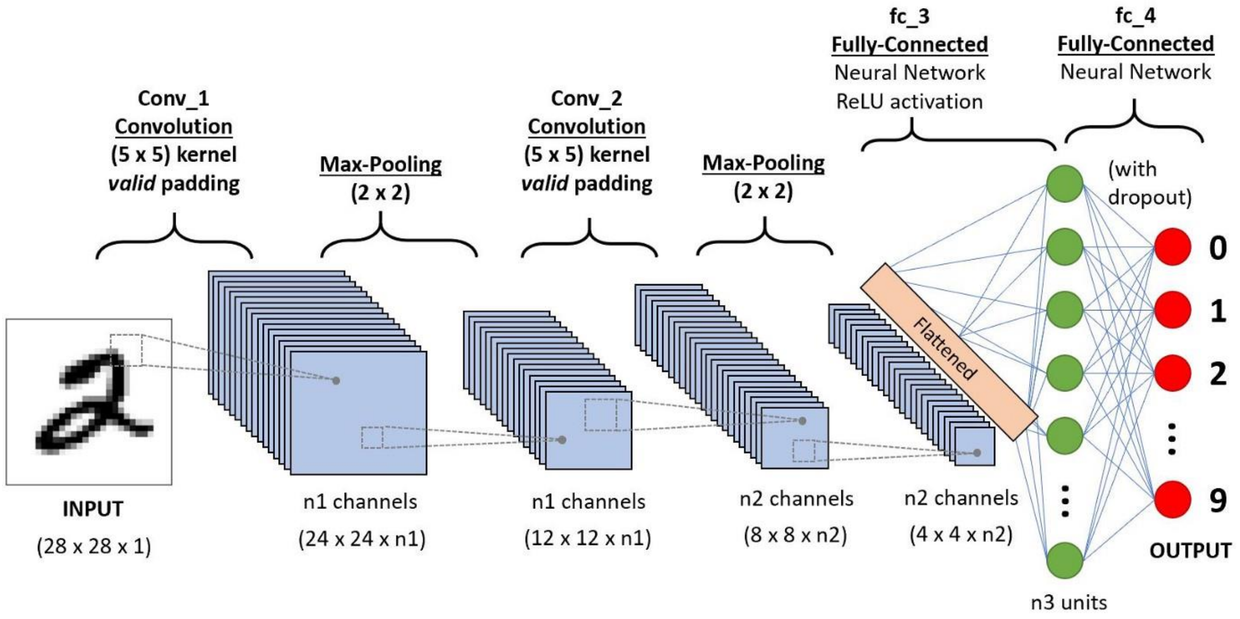

Figure 4. Another view of the stages of Convolution Neural Networks [Saha].

The first layer is the Convolution and Relu layer. This layer is very complicated. The main process is reducing the image to three matrices for each of the color layers. The algorithm traverses over sections of the color layer multiple times to reduce the matrix to the high-level features of the image. Some low-level features are the edges, color, gradients, etc, these are generally gathered in the first Convolution layer. The combination of low-level and high-level features gives the computer an understanding of an image. This process of traversing the image is called the Stride.

From here, the output can either be reduced, increased, or stay constant in dimensionality. This depends on the padding hyperparameter and the package you are using; some have a slightly different approach to how the Convolution layer changes – this will be discussed in the following section. Same Padding is when the Kernel output stays the same size and Valid Padding is when the Kernal output has the same size as the Kernel rather than the original image size or a one-dimension reduction from the original image or previous Kernel layer.

The Stride and Padding work together to create the Convolution + Relu layer. Once they are done, the Pooling layer comes into play. While the previous layer's goal to gather information from the image and possibly reduce its dimensionality, pooling is primarily a process to reduce the computational power required to process the data by further reducing the dimensions of the data. In that way it is similar to the Convolution layer or Padding, but it is also similar to Stride in the way that it extracts different dominant features.

The types of pooling are: Max Pooling – returns the maximum value of the portion – and Average Pooling – returns the average value of the portion in the Kernel. Max Pooling has an extra step to reduce the noise where Average Pooling treats the dimension reduction procedure as if it is a noise reduction mechanism itself. Generally, Max Pooling outperforms Average Pooling.

Notably, this process of stride, padding, then polling repeats as many times as there are layers for the model. This process repeats 2-20 times within this project alone for certain models. The increased number of layers means that some of these models take a significant amount of time to run. This was not a major issue in this project, but it should be considered when applying these methods to larger datasets or images that are not augmented to reduce computational load.

Once these layers have been computed for the set number of layers, the Feature Learning process is done, and the Classification process can begin. The steps of Classification are Flattening, Fully Connected, and Softmax. The Flattening layer is just when the matrix of each color layer is flattened to a nx1 matrix. The Fully Connected layer adds non-linear functionality if there is non-linearity present in the data. With the flattened output, it is refed into the Neural Network and each level of training has backpropagation applied. This is where the series of Epochs, seen below, comes into play. Softmax Classification can distinguish between dominating and low-level features to properly classify the images. Here it is Binary Classification, so the data is sorted in to 0-1 categories.

There are many other aspects that come in to play with these algorithms, mostly based on the specific package being used. A special thank you in this section to Sumit Saha, referenced below, for his in-depth discussion of CNN. The next section will discuss the differences between the packages used in this report in depth.

Comparison of Packages

Now that we have discussed the algorithm used in this paper in depth, I wanted to touch on the difference between the specific packages used later in the analysis. While there are 5 models compared at the end, six models were considered in this paper. I will give a short summary of the strengths and weaknesses of the six packages in the following paragraphs.

ConvNet is one of the most basic models here. This package is based on the PyTorch API [Google reseachers]. This is very simple because it applies a 2D Convolution over the input, as opposed to other models that can have 10-30 layers. Some of the key hyperparameters in this package are stride – which controls cross-correlation, padding, dilation – space between kernel points, and groups – the connections between inputs and outputs. Most CNN models have a weight variable, but this package also includes a learnable bias variable. Since this variable is not in the other packages, I did not include it in this analysis. ConvNet was outperformed by the following models due to higher complexity of the models.

The older version of VGG19, VGG16, is the first model considered and it is the only model considered by itself [“Keras Documentation: VGG16 and VGG19.”]. This model has a weaker performance overall than VGG19, as it is a newer version of the model. Like many of these models, it is based on the Keras API [Keras Team]. The 16 in VGG16 is the number of layers in the Neural Network. It’s later version, VGG19, has 19 layers to its Neural Network. It is a very large network, having ~138 million parameters, despite this large number, VGG16 is focused more on having static patterns of same size convolution layers, padding layers, and maxpool layers consistently. The number of hyperparameters in this model is smaller than most. This is not as important in this analysis as the five models evaluated together have the same hyperparameters inputted.

VGG19 is a newer version of VGG16. It is even larger than VGG16 but has even smaller networks filters and is deeper. In more use-cases, there is not much of a difference in VGG16 and VGG19 depending on how much computational power you want to use in your analysis. VGG16 is slightly faster than VGG19 but they are both very large networks and take a considerable amount of time to run. On my system, all the models took a long time to run as you will see later in the Model Results section if you look at the run time of each Epoch.

InceptionV3 came soon after the VGG models were created, this model increases the depth and width of the network but keeps computations constant – similar to the VGG16/19 models [Keras Team]. This is done by using 1x1 convolution layers to do dimension reduction by reducing the output of each convolution block. This model uses different kernel sizes to catch details on different scales.

The MobileNet package is based in the Tensorflow [Google researchers], Keras API [Keras Team]. This model is unique because it was developed with mobile applications in mind. It is careful of the limited resources available to a Mobile user and even has functionality for embedded applications. This was a choice that was useful for my system with its limited computational speed and performed quite well as you will see later. This model often sees more lag before it is well trained, but it works better with latency, size, and accuracy than the InceptionV3 might.

DenseNet is a bit different from the other models in its approach [Keras Team]. It has a much more compact model due to feature reuse. There are some issues that come up with feature reuse, these are remedied by appropriate padding in the layers and Dense Blocks – blocks that skip connectivity inside them. The growth rate of this algorithm is a much more important hyperparameter in DenseNet than other algorithms. While DenseNet is good for image classification, it is more versatile than other models.

These six models are all sufficiently unique from each other and offer a breadth of hyperparameters and approaches to image classification. I felt that these six models were appropriate in this project due to this breadth.

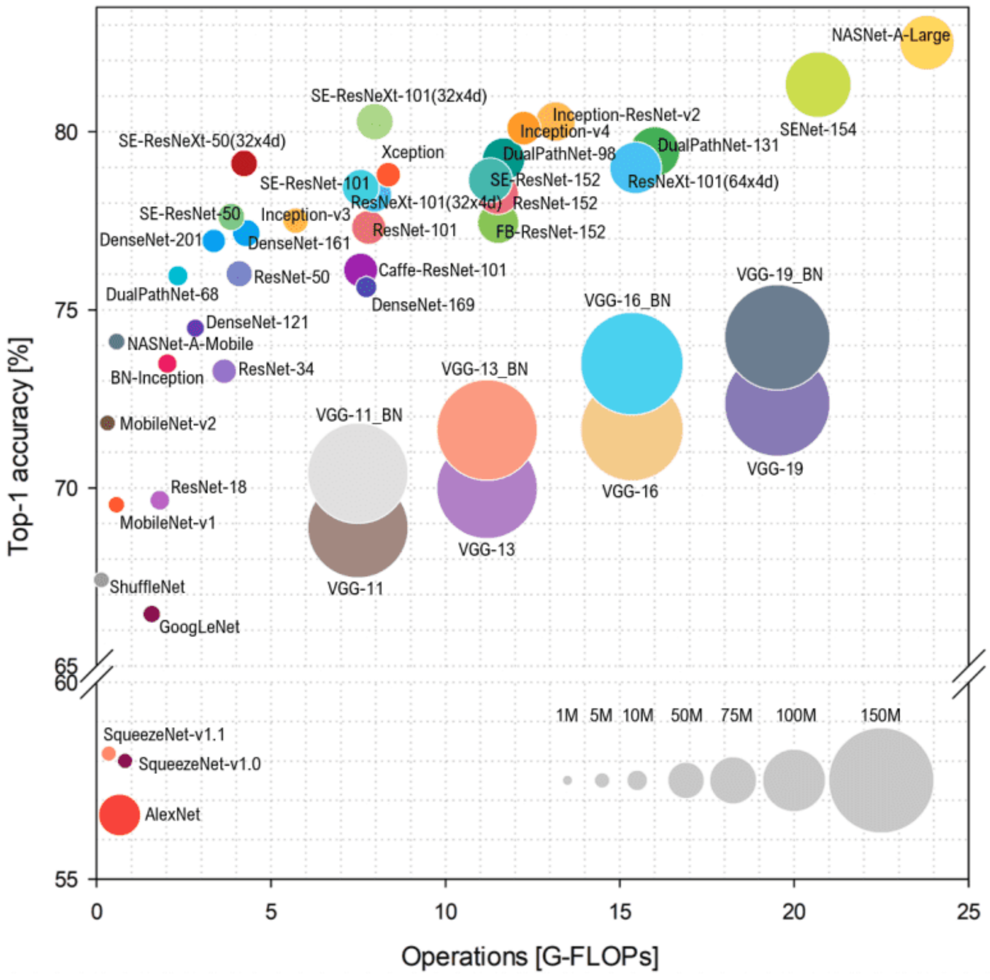

Figure 5. This figure shows many of the CNN architectures in terms of Operations vs. Accuracy. The image is here to display just how many CNN architectures there are that could have been used on this dataset. The Operations [G-FLOPs] suggests the complexity of the model and the Top-1 accuracy [%] show how well the model performs over many use-cases [Adaloglou].

While this paper only covers six models, there are many CNN architectures that you can use in a project. I choose the ones I did out of ease of modeling and personal curiosity. Some of the models chosen are less complex due to the computational limits of my personal system.

Validation

A major consideration of this project was how I was going to validate the models. The model performance must be validated to handle possible overfitting. Overfitting is when the model is trained "too well" to the training data set and the predictions are then less accurate. This is a major problem with prediction but can easily be found and remedied.

One of the easiest ways to validate your data is by using Cross-Validation. This is a process by which the original data set is split into a set that you "train" your model on and then "test". The Descriptive Statistics section (page 6) shows the way that this dataset was split. The training set has ~1,500 observations with Individual(s) wearing a Mask and ~4,000 observations with Individual(s) not wearing a Mask. The testing set has ~80 observations with Individual(s) wearing a Mask and ~500 observations with Individual(s) not wearing a Mask.

When the models were created, these training and testing sets were used to optimize the models. In the Results section (page 15), you can see that when I ran the VGG16 model, the training and the testing accuracy and loss are shown separately. When considering the strength of the models, I considered the difference between the training accuracy and the testing accuracy as a measure of model strength. For some of the models, they were run again on the initial model to tune them up more towards higher validation accuracy. I will explain this in more detail later in the paper.

With Cross-Validation, you can observe the accuracy of the training model versus the accuracy of the testing model to see if the model is overfitting or underfitting. Generally, as the accuracy of the training model increases, if the accuracy of the testing model doesn't increase along with it, then the model is being overfit. Vice-versa, when the accuracy of the testing model decreases, and the accuracy of the testing model increases greatly, then the model may be underfitting. Underfitting is usually less of a problem than overfitting, so as the Machine Learning model iterates, it works to balance these two accuracies out, but most models will have a slightly higher training model accuracy than testing model accuracy.

Accuracy, thought, is not the only consideration in validation. It can be split into Sensitivity – the ability of a model to correctly classify true cases – and Specificity – the ability of a model to correctly classify false cases. These are both extremely important to model and the combined value is considered the model accuracy. For this project, I would like to put a higher weight on Specificity to correctly identify when someone is not wearing a Mask. This is more useful for identifying people without a Mask – to either let them know that they need a mask in a building or to find later if someone was breaking policy from video surveillance. I did not have time to include this split of accuracy in this report, but if included I would have given a greater weight to Specificity over Sensitivity.

I considered the overfitting/underfitting trade off a lot more in the Results section (page 15) and used it as a measure of the model's strength. Since I had already used on a training and testing set on the data, so I know that it is more accurate, I felt comfortable stopping my Validation considerations here rather than breaking into more complicated methods. I would have dug deeper into Cross-Validation with this project if not for the computational limitations of my system.

Results

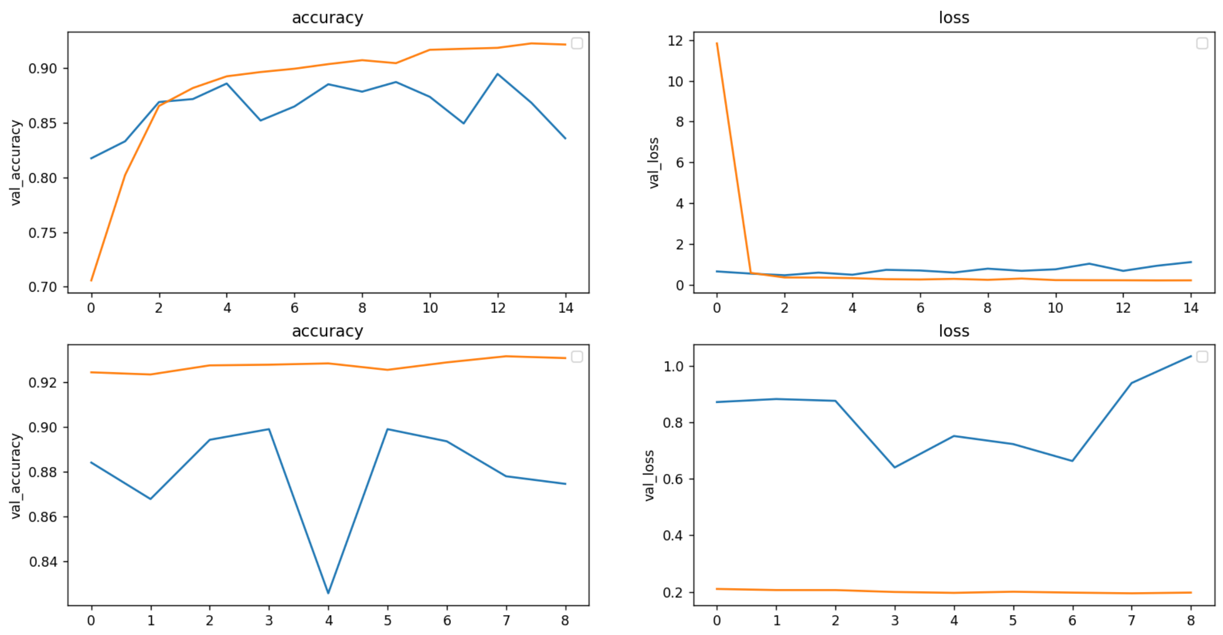

Figure 6. The results of the VGG16 model: training and trained versions, are shown here. The top two graphs show the training set, and the bottom two graphs show the validation set. The x-axis represents the number of Epochs for each model. The orange line represents the training set, and the orange represents the testing set. The accuracy of the predictions is shown in the left graphs and the loss (opposite of accuracy) is shown in the right graphs. For ease of access, I will discuss the graphs as 1 – accuracy of training (top left), 2 – loss of training (top right), 3 – accuracy of trained (bottom left), and 4 – loss of trained (bottom right).

The results shown here are for the first attempt using the VGG16 method with validation. The orange line represents the training set, and the orange represents the testing set. These were displayed separately so I could get a better idea of what the data looked like and how I should observe the results given that the models were trained and tested twice. The Epochs ran on the training model form 1-15, but with the trained model running over validation data, it often did not need to run for more than 8 Epochs.

From these graphs, in terms of accuracy, the first model took a bit longer to train than the second model as evidenced by the high slope in Graph 1 of the orange line. The final value of the first model was an accuracy of 0.9218, loss of 0.2157, validation accuracy of 0.8359 and validation loss of 1.1108. The final value of the second model was an accuracy of 0.9307, loss of 0.1973, validation accuracy of 0.8746 and validation loss of 1.0338. The training accuracy of both models ends up very strong at 92% and 93% respectively, but the issue comes in with the lower validation accuracy of 83% and 87%. While this is a big difference, the second model seems to have less of an issue with overfitting than the first model.

This overfitting is not extremely significant in the second model, but it was concerning. The validation set for both models had a much weaker accuracy and higher loss, suggesting that the model is a bit overfit to the original training set than we would want with a model. The trained model was even less overfit to the original data, there is near perfect accuracy of the training set then a much weaker accuracy of the testing set (seen in the bottom two graphs). That reason is the main reason I ended up looking into other Convolution Neural Network based models to see if there is an algorithm that will combat overfitting.

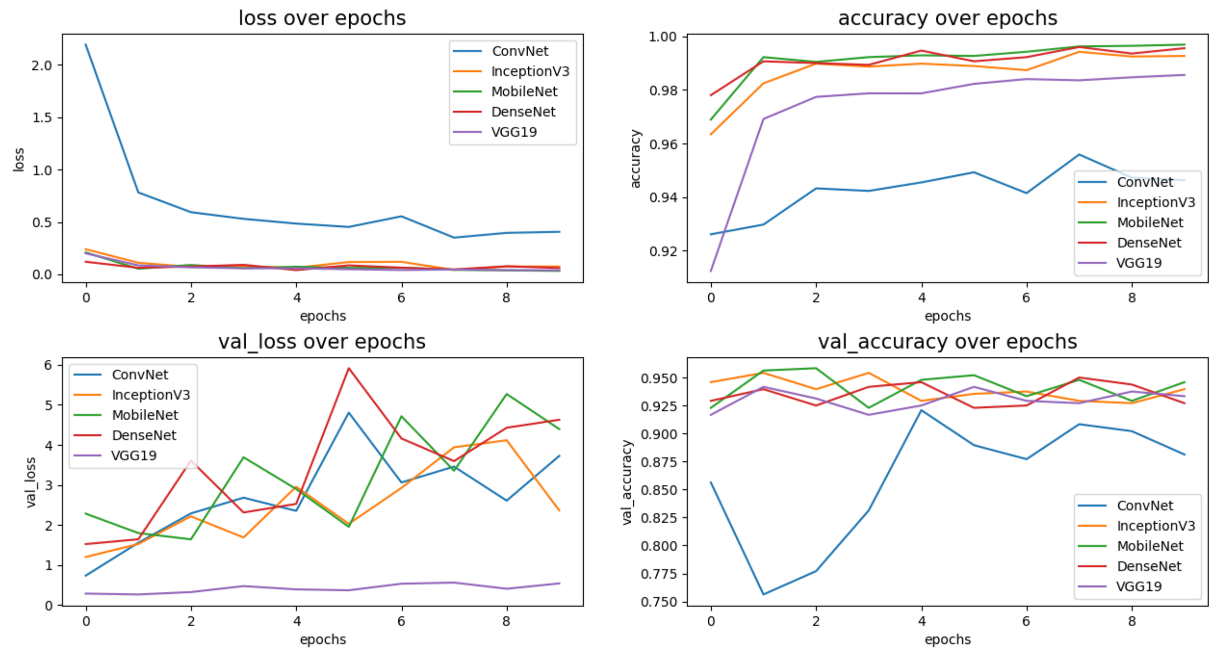

Figure 7. This chart shows the results for five different models in terms of accuracy, loss, validation accuracy, and validation loss. The five models shown are ConvNet (blue), InceptionV3 (orange), MobileNet (green), DenseNet (red), and VGG19 (purple).

Now, when looking at 5 models there is a much greater difference in performance. The VGG19 model was still included in this section so that direct comparisons could be drawn. All the models needed a little bit of work to train their initial model up to a strong level of performance, but overall, all the models performed strongly by their final epoch of accuracy > 90% and validation accuracy of > 90% except for the ConvNet model.

When looking at training model accuracy, there is a cluster of three models near the top: MobileNet, InceptionV3, and DenseNet. VGG19 appears slightly below those three with ConvNet falling last place. This makes sense when looking at the loss over epochs graph, where ConvNet has the steepest drop in loss then stays consistently higher in loss value than the rest of the models. For the training data, the three models at the top stand out as the strongest contenders. Their accuracy and losses are strong with MobileNet even reaching 0.9969 accuracy in the final Epoch.

As the most important issue with the VGG19 model was overfitting, the difference in validation accuracy for these models is very important. The following bullet points will display the accuracy of each model followed by the validation accuracy, the difference between the two at the final Epoch, and the percent difference between the two at the final Epoch.

- ConvNet: 0.9683, 0.8479, 0.1204, 13.2585%
- InceptionV3: 0.9969, 0.9458, 0.0511, 5.26072%
- MobileNet: 0.9969, 0.9458, 0.0511, 5.26072%
- DenseNet: 0.9956, 0.9271, 0.0685, 7.1254%
- VGG19: 0.9856, 0.9333, 0.0523, 5.45104

The results of all the models are very strong with the validation sets as well, with ConvNet trailing behind. The models with the lowest difference in performance are: InceptionV3, MobileNet, and VGG19. DenseNet has a higher training accuracy than VGG19, but its validation accuracy drops to be the second worst out of the five. Although, the difference is not enough for me to consider it to be a major issue with overfitting. The ConvNet model has a difference in accuracy almost twice as large as the next highest difference; out of these five models, this one appears to be overfitting the most. Due to this overfitting and its weak prediction abilities, I feel comfortable labeling ConvNet the worst model for this dataset and moving on from it.

The models with the smallest difference are InceptionV3 and MobileNet, which had the exact same performance in my analysis. On the chart, MobileNet appears to work better than the InceptionV3 model, but their results are the same. The difference between the accuracy and validation accuracy is also minimized for this model. The MobileNet seems to be the best model for this data, slightly above the InceptionV3 model due to its higher values in earlier Epochs.

From here, the DenseNet and VGG19 models come very close to each other in performance. The models both have a final accuracy around 93%, which is worse than InceptionV3 and MobileNet, but the main difference between the two is that DenseNet has a much higher training accuracy and VGG19 has a higher validation accuracy. In this report, accuracy validation is more important than training accuracy. So, due to the higher validation accuracy and smaller percent difference between the training and testing accuracies – VGG19 takes the third place in performance of these models.

On the side of some technical considerations, all the models took a fair bit of time to run on my laptop, but no model took more time than the others. So, the computational speed does not majorly factor in model performance. All of them are easy to implement once you have imported and augmented the images. I wanted to note these things as they are important to the project, but did not have a great effect on the final results of the models.

In terms of the Accuracy, Validation Accuracy, and the Overfitting of the data, the best models for Face Mask Detection in this project are:

1. MobileNet (same results as InceptionV3 but inches above on the graphs overall)

2. InceptionV3 (great results, would be a tie if this was not a ranked list)
3. VGG19 (due to slightly better final performance to DenseNet)
4. DenseNet (slightly preferring accuracy over possible overfitting)
5. ConvNet (worst performance of the five models by far)
6. VGG16 (worst performance in terms of training accuracy and validating accuracy)

I wanted to quickly note that the VGG19 model performs much better than the VGG16 model here in terms of overall accuracy. The more complex model does take more time to run, but the deeper neural networks seem to be more effective for this image classification data.

The best model for this Face Mask Detection Dataset and image processing is the MobileNet model. While the InceptionV3 model was strong, as well as the rest of the models considering that the accuracy validation was all around ~90%, the MobileNet model inches out as the best. The MobileNet model predicts whether someone in an image is wearing a Face Mask with 94.6% accuracy and the training model has a 99.7% accuracy rate. When considering projects like this, it is good to run multiple models and validate the data to get the best result. This project shows the strength of Convolution Neural Network models for Image Detection.

Discussion

Overall, I am very happy with the results of this project. The final model that worked best on this data was very well optimized for a limited resource system. All the models ended up having a strong prediction accuracy of at least 85% and the strongest got up to 94%. This is very good for validation prediction accuracy.

On my system, all the code took about 12 hours, with short breaks, to run even with the limited Epochs. My system has its own limitations, older i5 processor and general bloat, so I am not unhappy with the computational time. If I had more computational power, I would have tried some of the more Operation heavy models from Figure 5. With Machine Learning, it is good to test a lot of different models if you have time to see which one is best based on the parameters you are most interested in studying.

I was surprised by the strength of the MobileNet model in this application due to its lag and limited resource assumption. MobileNet is at about 72.5% on the Top-1% accuracy in Figure 4, but some of the other models tested were even higher in accuracy in the same figure [Adaloglou]. It had a small difference between the accuracy and the validation accuracy, which was good too, it didn't seem to have any issues with overfitting.

I would have liked to include the accuracy split into Sensitivity and Specificity. The lack of split was due to time and computational limitations. The very high accuracy could be due to a maximized Sensitivity or Specificity, but likely these values are still very high with the MobileNet at 94% overall. If I had some extra time and less issues with my computer during this project, I would have added a section to split the accuracy and then consider how the accuracy changes when a higher penalty is put on Type 2 errors.

One of the hardest parts of this project was learning how to properly import the images and augment them. I tried many different methods to import the images, as this is the first time I've handled images in code, but with the final method I learned a lot about the os package in Python. The image augmentation process taught me about data catalogues in Python and their usage. The CNN packages are very straight forward and make the programming extremely simple. Machine Learning models, in my experience, are not very challenging to code as they are mostly package based, but their interpretation/explanation can be challenging.

Appendix: Code in Python

```
#Imports
import os
from shutil import copyfile
import cv2
import keras  # remember you have to import this separately
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import PIL
import seaborn as sns
import tensorflow as tf
from keras.callbacks import *
from keras.layers import *
from keras.layers import (BatchNormalization, Conv2D, Dense, Dropout, Flatten,
                          MaxPooling2D)
from keras.models import Sequential
from keras.preprocessing.image import *
from matplotlib.animation import PillowWriter
from PIL import Image
from tensorflow.keras.applications import (VGG19, DenseNet201, EfficientNetB1,
                                           InceptionV3, MobileNet, ResNet50)
from tensorflow.keras.optimizers import Adam
from centroid_neural_network import (centroid_neural_net, img_encoder_kmeans,
                                     remove_element)
from Generation import *

#Establishing where the images are from, then importing
#This would be different for Different Users
face_mask_detection_dir = 'C:\\Users\\clari\\OneDrive\\Documents\\Spring 
2022\\Honors Project\\Face Recognition Data\\images'

with_without_mask_train = 'C:\\Users\\clari\\OneDrive\\Documents\\Spring 
2022\\Honors Project\\Face Mask Dataset\\Train'
with_without_mask_test = 'C:\\Users\\clari\\OneDrive\\Documents\\Spring 
2022\\Honors Project\\Face Mask Dataset\\Test'

with_mask_train_dir = os.path.join(with_without_mask_train, 'WithMask')
without_mask_train_dir = os.path.join(with_without_mask_train, 'WithoutMask')

with_mask_test_dir = os.path.join(with_without_mask_test, 'WithMask')
without_mask_test_dir = os.path.join(with_without_mask_test, 'WithoutMask')

#Descriptive Statistics
```

```
#1 is "Wearing Mask" and 0 is "Not Wearing Mask"
train_wear_mask = pd.DataFrame()
train_wear_mask['Mask or No Mask'] = ['1'] *
len(os.listdir(face_mask_detection_dir)) + ['1'] *
len(os.listdir(with_mask_train_dir)) + ['0'] *
len(os.listdir(without_mask_train_dir))
train_wear_mask['id'] = [os.path.join(face_mask_detection_dir, name) for name in
os.listdir(face_mask_detection_dir)] +[os.path.join(with_mask_train_dir, name)
for name in os.listdir(with_mask_train_dir)] +
[os.path.join(without_mask_train_dir, name) for name in
os.listdir(without_mask_train_dir)]

test_wear_mask = pd.DataFrame()
test_wear_mask['Mask or No Mask'] = ['1'] * len(os.listdir(with_mask_test_dir)) +
['0'] * len(os.listdir(without_mask_test_dir))
test_wear_mask['id'] = [os.path.join(with_mask_test_dir, name) for name in
os.listdir(with_mask_test_dir)] + [os.path.join(without_mask_test_dir, name) for
name in os.listdir(without_mask_test_dir)]

train_wear = 0
train_not_wear = 0
i = 0
for i in range(0, len(train_wear_mask['Mask or No Mask'])):
    if train_wear_mask['Mask or No Mask'][i] == '1':
        train_wear += 1
        i += 1
    else:
        train_not_wear += 1
        i += 1

test_wear = 0
test_not_wear = 0
i = 0
for i in range(0, len(test_wear_mask['Mask or No Mask'])):
    if test_wear_mask['Mask or No Mask'][i] == '1':
        test_wear += 1
        i += 1
    else:
        test_not_wear += 1
        i += 1

##Quick Percentages of Wearing Mask vs. Not Wearing Mask
train_distribution_wear = (train_wear/len(train_wear_mask['Mask or No Mask'])) *
100
```

```
train_distribution_not_wear = (train_not_wear/len(train_wear_mask['Mask or No
Mask'])) * 100
#Here, there is slightly more data for the "Wearing Mask" category

test_distribution_wear = (test_wear/len(test_wear_mask['Mask or No Mask'])) * 100
test_distribution_not_wear = (test_not_wear/len(test_wear_mask['Mask or No
Mask'])) * 100
#This split is much closer to 50-50

##Plotting the Distributions
plt.figure(figsize = (15, 10))

plt.subplot(1, 2, 1)
plt.title('Mask Wearers vs. Not Wearing a Mask in Training Data \n')
sns.countplot(x = train_wear_mask['Mask or No Mask'])

plt.subplot(1, 2, 2)
plt.title('Mask Wearers vs. Not Wearing a Mask in Testing Data \n')

sns.countplot(x = test_wear_mask['Mask or No Mask'])
plt.show()

 # Train data

# Define the directories
os.mkdir('train')
os.mkdir('train/without')
os.mkdir('train/with')

train_dir = 'train'
train_with = 'train/with'
train_without = 'train/without'

# Copy the files to create catalogues
for index in range(len(train_wear_mask['id'])):
    if train_wear_mask['Mask or No Mask'][index] == '1':
        copyfile(src = train_wear_mask['id'].iloc[index], dst =
'train/with/{}.jpg'.format(index))
    else:
        copyfile(src = train_wear_mask['id'].iloc[index], dst =
'train/without/{}.jpg'.format(index))

# Test_data
# Define the directories
os.mkdir('test')
os.mkdir('test/without')
os.mkdir('test/with')

test_dir = 'test'
test_with = 'test/with_test'
test_without = 'test/without_test'

# Copy the files to create catalogues
for index in range(len(test_wear_mask['id'])):
    if test_wear_mask['Mask or No Mask'][index] == '1':
        copyfile(src = test_wear_mask['id'].iloc[index], dst =
'test/with/{}.jpg'.format(index))
    else:
        copyfile(src = test_wear_mask['id'].iloc[index], dst =
'test/without/{}.jpg'.format(index))

# Generator for Rescalling
train_datagen = ImageDataGenerator(
    rescale = 1./255,
    zoom_range = 0.2,
    horizontal_flip  = True,
    rotation_range = 40,
    fill_mode = 'nearest'
)

# Generator for training and testing sets are the same
test_datagen = ImageDataGenerator(rescale = 1./255)

# Flows from directory to dynamically use the images
train_generator = train_datagen.flow_from_directory(
    train_dir,
    batch_size = 32,
    class_mode = 'binary',
    target_size = (256, 256)
)

test_generator = train_datagen.flow_from_directory(
    test_dir,
    batch_size = 32,
    class_mode = 'binary',
    target_size = (256, 256)
)

valid_generator = test_datagen.flow_from_directory(
    test_dir,
    batch_size = 32,
    class_mode = 'binary',
    target_size = (256, 256)
)
# Make sure that you get data with multiple classes here, and you will need to
remove the new augmented pictures
# when you rerun the code.

# This allows the code to cut off the epochs on higher iterations
def scheduler(epoch, lr):
    if epoch < 15:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

callbacks = [
    EarlyStopping(monitor = 'val_accuracy', patience = 5),
    ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001),
    ModelCheckpoint(filepath='weights.h5', save_weights_only=True,
monitor='val_accuracy',mode='max', save_best_only=True),
    LearningRateScheduler(scheduler),]

# VGG16 CNN Method, by itself

VGG =  VGG16(weights = 'imagenet', include_top = False, input_shape = (256, 256,
3))

VGG.trainable = False

model = Sequential()
model.add(VGG)

model.add(Flatten())

model.add(Dropout(0.3))
model.add(Dense(64, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))

model.compile(loss = 'binary_crossentropy', metrics = ['accuracy'])

history = model.fit(
    train_generator,
    batch_size = 32,
    epochs = 15,
    validation_data = test_generator,
    callbacks = callbacks
         )

plt.figure(figsize = (10, 10))
plt.subplot(2,2, 1)
plt.title('accuracy')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'val_accuracy')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'accuracy')
plt.legend()

plt.subplot(2,2, 2)
plt.title('loss')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'val_loss')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'loss')
plt.legend()

VGG.trainable = True
set_trainable = False

for layer in VGG.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False

model.compile(loss = 'binary_crossentropy', metrics = ['accuracy'])

history = model.fit(
    train_generator,
    batch_size = 32,
    epochs = 15,
    validation_data = test_generator,
    callbacks = callbacks
         )

plt.subplot(2,2, 3)
plt.title('accuracy')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'val_accuracy')
```

```
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'accuracy')
plt.legend()

plt.subplot(2,2, 4)
plt.title('loss')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'val_loss')
sns.lineplot(data = history.history, x =
range(len(history.history['val_accuracy'])), y = 'loss')
plt.legend()
plt.show()

# Conv2D Method with Optimizer

histories = []
for i in range(3):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(256,
256, 3)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))

    if i > 0:
        model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
input_shape=(256, 256, 3)))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.2))

        if i > 1:
            model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
input_shape=(256, 256, 3)))
            model.add(BatchNormalization())
            model.add(MaxPooling2D(pool_size=(2, 2)))
            model.add(Dropout(0.2))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    model.summary()

model.compile(optimizer='adam', loss='binary_crossentropy', metrics='accuracy')
histories.append(model.fit_generator(generator=train_generator,
                                         validation_data=test_generator,
                                         steps_per_epoch=len(train_generator)//3,
                                         validation_steps=len(test_generator)//3,
                                         epochs=10))

fig, axes = plt.subplots(2, 2, figsize=(15, 10))
for metric in histories[0].history:
    index = list(histories[0].history).index(metric)
    ax = axes.flatten()[index]
    layer_num = 0
    for history in histories:
        layer_num += 1
        ax.plot(history.history[metric], label=str(layer_num)+' layer(s)')
    ax.set_title(metric)
    ax.legend()
plt.show()

# Multiple Methods in one Model (5 Models)

model_histories = []
models = [InceptionV3(include_top=False, input_shape=(256, 256, 3)),
                   MobileNet(include_top=False, input_shape=(256, 256, 3)),
                   DenseNet201(include_top=False, input_shape=(256, 256, 3)),
                   VGG19(include_top=False, input_shape=(256, 256, 3))]
names = ['ConvNet', 'InceptionV3', 'MobileNet', 'DenseNet', 'VGG19']

for layer in [Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(256,
256, 3))]:
    model = Sequential()
    model.add(layer)
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics='accuracy')
    model_histories.append(model.fit_generator(generator=train_generator,
                                         validation_data=test_generator,
                                         steps_per_epoch=len(train_generator)//3,
                                         validation_steps=len(test_generator)//3,
                                         epochs=10))

# Rerunning on the Trained model
for functional in models:
    for layer in functional.layers:
        layer.trainable = False
    model = Sequential()
    model.add(functional)
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics='accuracy')
    model_histories.append(model.fit_generator(generator=train_generator,
                                          validation_data=test_generator,
                        steps_per_epoch=len(train_generator)//3,
                        validation_steps=len(test_generator)//3, epochs=10))

fig, axes = plt.subplots(2, 2, figsize=(15, 10))
fig.subplots_adjust(hspace=0.3)
for metric in model_histories[0].history:
    index = list(model_histories[0].history).index(metric)
    ax = axes.flatten()[index]
    name_index = 0
    for history in model_histories:
        ax.plot(history.history[metric], label=names[name_index])
        name_index += 1
    ax.set_title(metric+' over epochs', size=15)
    ax.set_xlabel('epochs')
    ax.set_ylabel(metric)
    ax.legend()
plt.show()
```

References

Adaloglou, Nikolas. “Best Deep CNN Architectures and Their Principles: From Alexnet to EfficientNet.” *AI Summer*, Sergios Karagiannakos, 21 Jan. 2021, https://theaisummer.com/cnn-architectures/.

“Centroid Neural Network and Vector Quantization for Image Compression.” *Medium*, Towards AI, 19 Apr. 2021, https://pub.towardsai.net/centroid-neural-network-and-vector-quantization-for-image-compression-a7d30aa63167.

Dennanni, Adriano. “How to Deal with Image Resizing in Deep Learning.” *Medium*, Neuronio, 8 Oct. 2019, https://medium.com/neuronio/how-to-deal-with-image-resizing-in-deep-learning-e5177fad7d89.

Dwivedi, Harshit. “Comparing Mobilenet Models in Tensorflow.” *KDnuggets*, Mar. 2019, https://www.kdnuggets.com/2019/03/comparing-mobilenet-models-tensorflow.html.

Gilani, Rafay. “Main Challenges in Image Classification.” *Medium*, Towards Data Science, 14 June 2020, https://towardsdatascience.com/main-challenges-in-image-classification-ba24dc78b558.

Google researchers. “API Documentation: Tensorflow Core v2.8.0.” *TensorFlow*, Google, 2 Feb. 2022, https://www.tensorflow.org/api_docs.

Jain, Taru. “Basics of Machine Learning Image Classification Techniques.” *OpenGenus IQ: Computing Expertise & Legacy*, OpenGenus IQ: Computing Expertise & Legacy, 29 Aug. 2019, https://iq.opengenus.org/basics-of-machine-learning-image-classification-techniques/.

“Keras Documentation: VGG16 and VGG19.” *Keras*, Google, 17 June 2019, https://keras.io/api/applications/vgg/.

Larxel. “Face Mask Detection.” *Kaggle*, 22 May 2020, https://www.kaggle.com/andrewmvd/face-mask-detection.

Paialunga, Piero. “Image Classification Using Machine Learning, Made Simple.” *Medium*, Towards Data Science, 1 Apr. 2021, https://towardsdatascience.com/image-classification-using-machine-learning-made-simple-cf7428a85bee.

Saha, Sumit. “A Comprehensive Guide to Convolutional Neural Networks - the eli5 Way.” *Medium*, Towards Data Science, 17 Dec. 2018, https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Team, Keras. “Keras Documentation: DenseNet.” *Keras*, Google, https://keras.io/api/applications/densenet/.

Team, Keras. “Keras Documentation: Inceptionv3.” *Keras*, Google, https://keras.io/api/applications/inceptionv3/.

Team, Keras. “Keras Documentation: Mobilenet, mobilenetv2, and MobileNetV3.” *Keras*, Google, https://keras.io/api/applications/mobilenet/.

Torch Contributors. “Conv2d.” *Conv2d - PyTorch 1.11.0 Documentation*, PyTorch, 2019, https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.