Predicting Covid-19 Positive Cases from Chest X-Ray Images

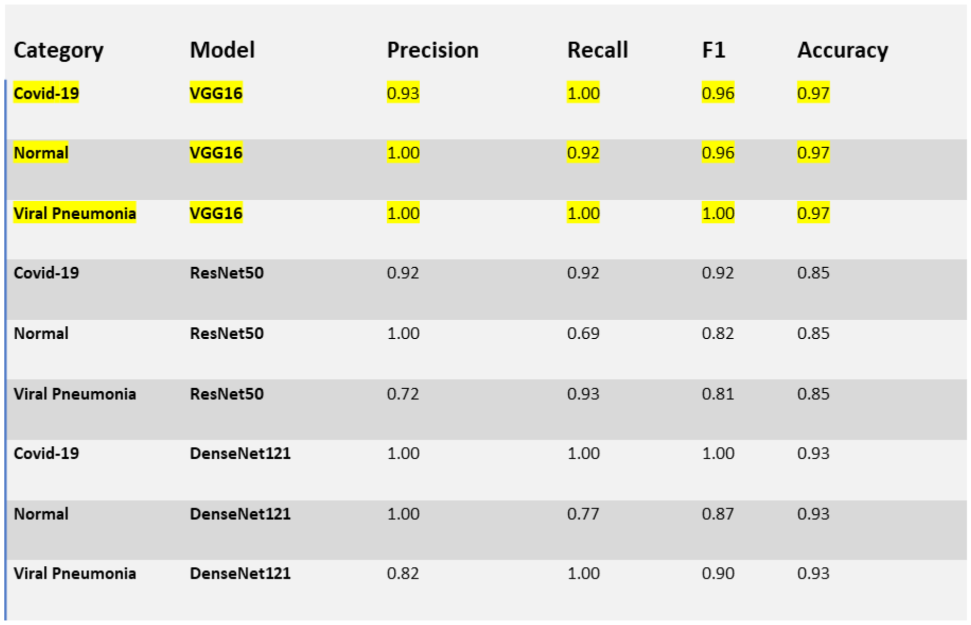
George Pinto 01/2021

Executive Summary

This document presents a useful convolutional neural network classification method leveraging transfer learning for feature extraction to assist clinicians, radiologists, and those in the need of interpreting X-ray images for diagnostic purposes. A pressing, current situation in which the use of artificial intelligence could prove particularly useful is classification of the novel disease Covid-19. This is the challenge attempted here. Data processing was performed on chest X-Rays consisting of 219 Covid-19 images, 1341 normal chest images, and 1345 normal pneumonia images made available from a team of Researchers from Qatar University and the University of Dhaka in Bangladesh. various transfer learning models were customized, trained, and tested after exploring the data, calculating summary and descriptive statistics, creating visualizations of the data, and performing random under-sampling, normalization and scaling of the data.

It was determined that a VGG16 transfer learning model could be created to significantly enhance the tasks of interpretation and classification of Covid-19 positive chest X-Ray images. The need for data processing is eliminated by incorporating this task into the data pipeline as well as scaling the images for classification (images are normally sized 1024 by 1024 pixels, making working with them a challenging and time consuming to process). Images can be uploaded to the model through an API and predictions can be served from the cloud using any device. Furthermore, model results could be further explored for deeper medical analysis and interpretability by use of the GradCAM algorithm.

The following comparative results between customized models aided in the selection of the VGG16 model as our model of choice for classification of Covid-19:



Category	Model	Precision	Recall	F1	Accuracy
Covid-19	**VGG16**	0.93	1.00	0.96	0.97
Normal	**VGG16**	1.00	0.92	0.96	0.97
Viral Pneumonia	**VGG16**	1.00	1.00	1.00	0.97
Covid-19	**ResNet50**	0.92	0.92	0.92	0.85
Normal	**ResNet50**	1.00	0.69	0.82	0.85
Viral Pneumonia	**ResNet50**	0.72	0.93	0.81	0.85
Covid-19	**DenseNet121**	1.00	1.00	1.00	0.93
Normal	**DenseNet121**	1.00	0.77	0.87	0.93
Viral Pneumonia	**DenseNet121**	0.82	1.00	0.90	0.93

Data Exploration and Processing

The initial exploration of the data began with some summary and descriptive statistics.

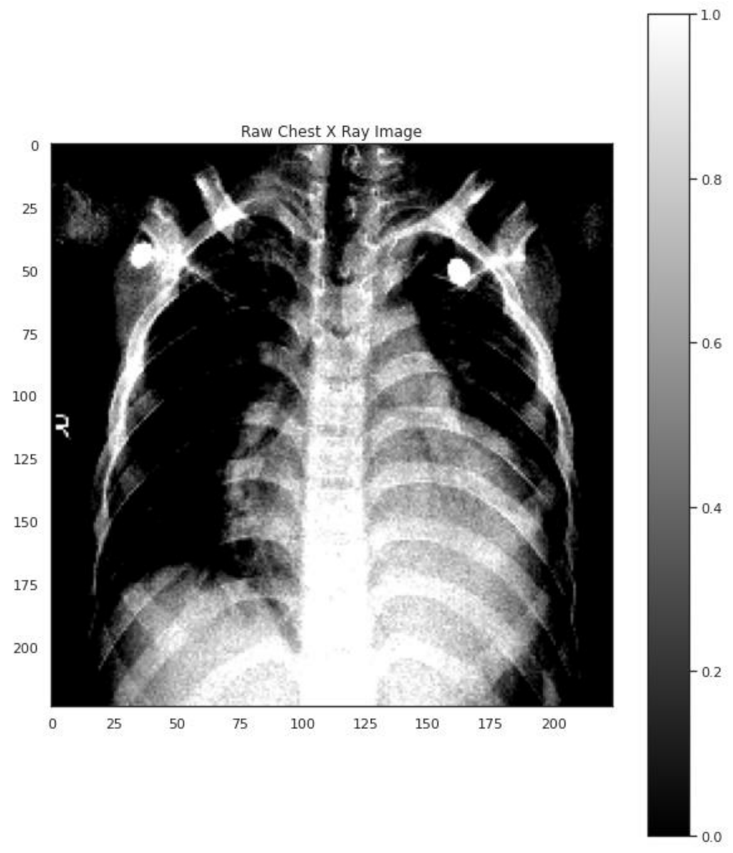
Image dimensions were 1024 x 1024 for all three classes.

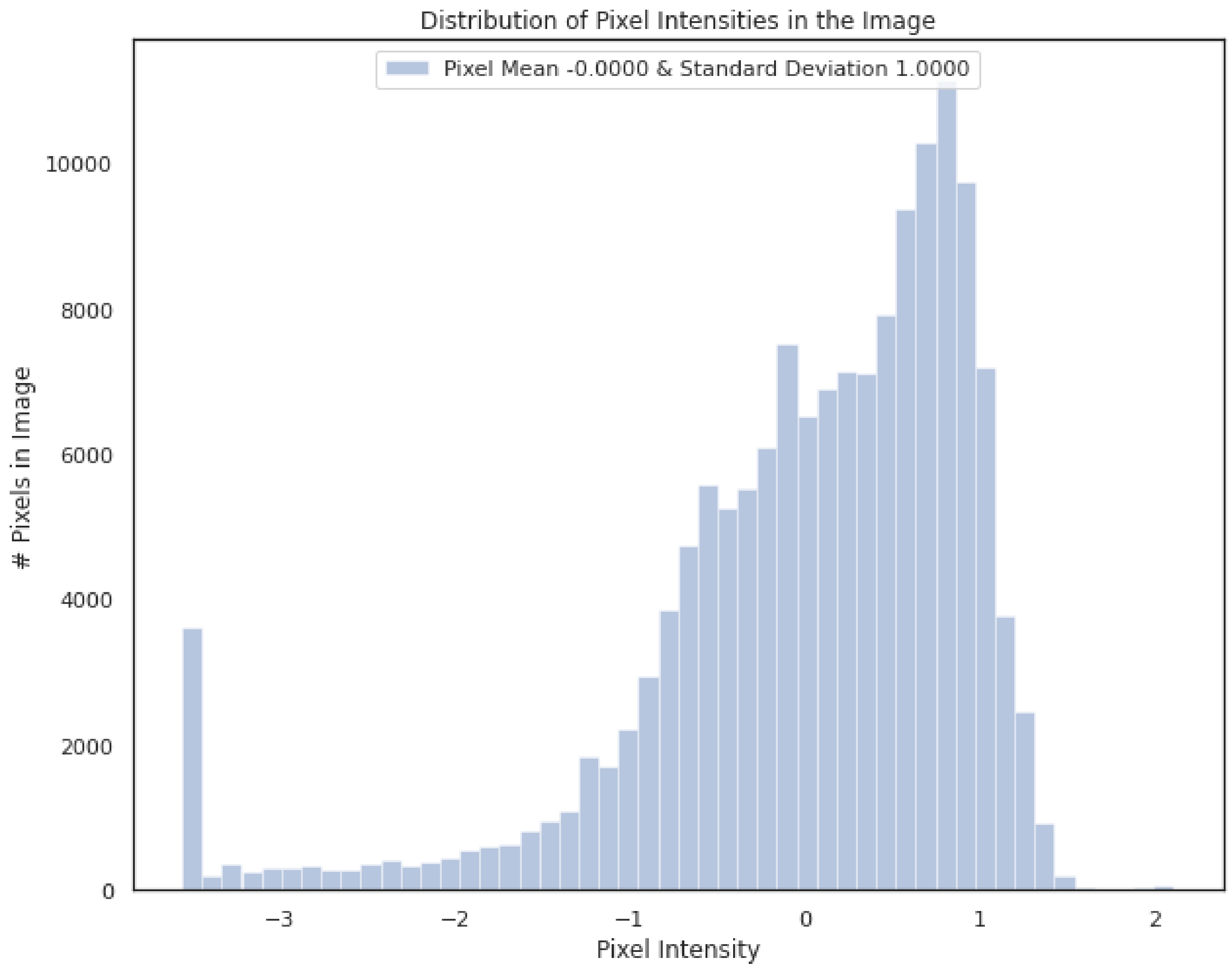
The images needed be scaled down to 224 x 224 to fit the transfer learning model input requirements. TensorFlow can automatically scale down the images if this is specified when creating generators for batch data, it is also helpful to use this feature throughout to save on computation and overall efficiency while running experiments and iterations.

The mean and standard deviation of pixel values were normalized to 0 and 1 respectively by utilizing training data statistics. This provided centered data with a good range of pixel intensities for learning.

The number of color channels was not reduced to 1 due to transfer learning (224, 224, 3) shape requirements for the TensorFlow models that were downloaded, this last dimension being the color channels, so the last dimension was left at 3 channels for the input layer. Scaling and normalization can be configured when creating the generators using TensorFlow. Here is one sample of a scaled and normalized image:

```
The dimensions of the image are 224 pixels width and 224 pixels height. The
maximum pixel value is 2.1088 and the minimum is -3.5569 The mean value of the
pixels is -0.0000 and the standard deviation is 1.0000
```





Note that the shape of the normalized pixel distributions can benefit and accelerate learning.

Since the goal was Covid-19 classification, it was noted that there was a significant class imbalance, note the number of Covid-19 images vs the other classes:

- 219 Covid-19 images
- 1341 normal chest images
- 1345 normal pneumonia images

Data was randomly under-sampled to at least the fraction of Covid-19 images prior to creating training, validation and testing subsets. This is demonstrated in Python code using Pandas on a viral pneumonia data example:

```
pneumonia_new = 219/1345
pneumonia = pneumonia.sample(frac=pneumonia_new).reset_index(drop=True)
pneumonia.shape[0]
```

Data Wrangling

The dataset included excel files for each of the separate classes containing indexed filenames along with class labels and it was noted that for downstream neural network training, we would be using the Keras flow_from_dataframe method. The chosen method would use the filenames to upload the images on the fly into the batch generators. The filenames on the excel sheets were originally not matching the image names in matching image folders, Pandas DataFrames were created making sure everything matched. The following code examples show corrections to the issues and are clearly described in the comments on the code (comments start with # on the python code and are in blue):

```
# the images in the folders have a space before the parenthesis, add space

covid['FILE NAME'] = covid['FILE NAME'].str.replace("(","  (")
```

```
# remove dash and add parenthesis to match the image files in the folders

pneumonia['FILE NAME'] = pneumonia['FILE NAME'].str.replace('-',' (') + ")"
```

```
# remove dash and add parenthesis to match the image files in the folders

normal['FILE NAME'] = normal['FILE NAME'].str.replace('-',' (') + ")"
```

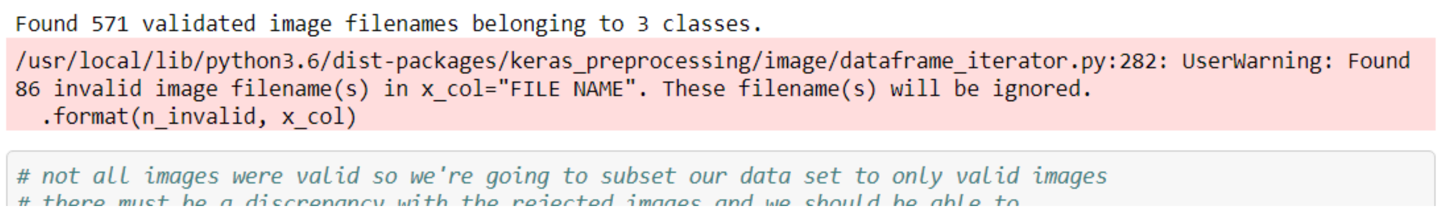
Re-sorting and re-combining the classes required adding the file format string to the file names and then the file names would be ready as a column in the combined DataFrame:

```
# add .png to all the class names to match the image names

allclasses['FILE NAME'] = allclasses['FILE NAME'].astype(str) + '.png'
```

Generators

Even after data wrangling there were some issues regarding Covid-19 file names (other class filenames had no further issues). Some of the Covid-19 images were invalid for generator input, note the error beneath the generator code given by TensorFlow:

```
generator = image_generator.flow_from_dataframe(
        dataframe=df,
        directory='/gdrive/My Drive/XRaysCombo',
        x_col="FILE NAME", # features
        y_col= 'Class', # labels
        class_mode='categorical', # 3 classes
        batch_size= 1, # images per batch
        shuffle=False, # shuffle the rows or not
        target_size=(224,224) # width and height of output image
)
```



```
Found 571 validated image filenames belonging to 3 classes.
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/dataframe_iterator.py:282: UserWarning: Found
86 invalid image filename(s) in x_col="FILE NAME". These filename(s) will be ignored.
  .format(n_invalid, x_col)
```

```
# not all images were valid so we're going to subset our data set to only valid images
# there must be a discrepancy with the rejected images and we should be able to
```

There must have been a discrepancy not resolved by the data wrangling (which took care of most of the issues). In the interest of preserving image to label relationships intact, along with the knowledge that enough files would still be available for testing if we removed these images, and knowing that a transfer learning model was going to be chosen to deal with the small number of images per class, it was opted to randomly under-sample all classes to the number of valid Covid-19 images, done as follows for each class, then all classes were combined randomly as we had originally done, shown here in the code:

```
# Take a sample of the normal and pneumonia datasets for which the fractions
# match the number of covid samples

normal_new = 133/219

normal = normal.sample(frac=normal_new).reset_index(drop=True)

normal.shape[0]
```

The next generator test was successful as shown in the code here with 133 images per class and this would be our chosen number of random images per class for creating the training, validation and testing data sets:

```
# Flow from dataframe with specified batch size and target image size
# Recall that the image generator will normalize the data

generator = image_generator.flow_from_dataframe( dataframe=allclasses,
directory='/gdrive/My Drive/XRaysCombo',
x_col="FILE NAME", # features
y_col= 'Class', # labels class_mode='categorical', # 3 classes
batch_size= 1, # images per batch
shuffle=False, # shuffle the rows or not
target_size=(224,224) # width and height of output image
)
```

```
Found 399 validated image filenames belonging to 3 classes.
```

Train Test Validation Split

After verifying 399 image filenames were available the images were split into train, test and validation splits using sci-kit learn, it was required to put a small number of images aside for testing and a slightly larger amount for validation. This is roughly 10% for test and 13.5% for validation. The Python/Pandas code in this case does a fantastic job of clearly showing the process:

```
# Split at 10% test, we will use the training split for validation as well
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
test_size=0.10)
```

```
# Check sizes

print('X_train: ',X_train.shape)
print('X_test: ',X_test.shape)
print('y_train: ',y_train.shape)
print('y_test: ',y_test.shape)
```

```
X_train: (359,)
X_test: (40,)
y_train: (359,)
y_test: (40,)
```

```
# Split train for 15% val
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
stratify=y_train, test_size=0.15)
```

```
# Check sizes

print('X_train: ',X_train.shape)
print('X_val: ',X_val.shape)
print('y_train: ',y_train.shape)
print('y_val: ',y_val.shape)
```

```
X_train: (305,)
X_val: (54,)
y_train: (305,)
y_val: (54,)
```

```
# Concatenate dataframes

training =  pd.concat([X_train, y_train], axis=1)
validation = pd.concat([X_val, y_val], axis=1)
```

```
testing = pd.concat([X_test, y_test], axis=1)
```

GPUs

Before running several deep learning models to compare their performance and certainly before doing multiple iterations after architectural and hyperparameter changes, it is of utmost importance to mention that running on GPUs is an absolute necessity. The notebooks were run on Google Colab, which has the option of running on GPUs. Even with GPUs the experiments will take some time to run. An important consideration here is statistical robustness. It is not unusual to run the models once, after finding the proper architectures and tuning. However, in an ideal situation, where multiple GPUs are available and high-level computing with plenty of RAM, etc., it would be recommended to run cross validation studies to further verify the robustness of the results.

Deep Learning Architectures

Since only a small number of images was available for each class it was determined that the best type of deep learning architecture choice was transfer learning. The following models were selected as the base models to choose from, obtained from TensorFlow applications:

- VGG16
- Resnet50
- Densenet121

The first model architecture attempt used the following TensorFlow 2 code, the VGG16 model was used as the base and the head was custom built using an AveragePooling2D layer, a Flatten layer, a Dense layer, a Dropout layer, and a final Dense layer with a Softmax activation. Note that the priority here is to utilize the features learned by the model using the Imagenet data set, and thus all base layers were frozen (note the for loop at the bottom of the script) to only learn the weights of the head layers:

```
model = tf.keras.Sequential()
# load the VGG16 network, ensuring the head FC layer sets are left
# off
base = tf.keras.applications.VGG16(weights="imagenet", include_top=False,
       input_tensor=tf.keras.Input(shape=(224,224,3)))

# construct custom head of the model that will be placed on top of the
# the base model
head = base.output
head = tf.keras.layers.AveragePooling2D(pool_size=(4, 4))(head)
head = tf.keras.layers.Flatten()(head)
head = tf.keras.layers.Dense(64, activation="relu")(head)
head = tf.keras.layers.Dropout(0.5)(head)
head = tf.keras.layers.Dense(3, activation="softmax")(head)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = tf.keras.Model(inputs=base.input, outputs=head)

# loop over all layers in the base model and freeze them
for layer in base.layers:
       layer.trainable = False
```

Note that the same basic concept of utilizing the base layer as a pre-trained feature extractor was used for the ResNet50 and DenseNet121 architectures with small variations on the head layer configurations, which depend on trial and error when running through model iterations, here is the TensorFlow 2 code for the other two models, observe the structural differences:

ResNet50

```
def create_model():
  """
  This function returns a base ResNet50 transfer learning model trained on
  imagenet connected to a custom head adapted to a 3 class softmax probability
  """
  # we will start by building a ResNet50 model
  # Build model freezing top layers and adding a dense layer
  # with the number of classes
  model = tf.keras.Sequential()
  # load the ResNet50 network, ensuring the head FC layer sets are left
  # off
  base = tf.keras.applications.ResNet50(weights="imagenet", include_top=False,
         input_tensor=tf.keras.Input(shape=(224,224,3)))

  # construct custom head of the model that will be placed on top of the
  # the base model
  head = base.output
  head = tf.keras.layers.AveragePooling2D(pool_size=(4, 4))(head)
  head = tf.keras.layers.Flatten()(head)
  head = tf.keras.layers.Dense(32, activation="relu")(head) # reduced number of
# units for ResNet50
  head = tf.keras.layers.Dropout(0.5)(head)
  head = tf.keras.layers.Dense(3, activation="softmax")(head)

  # place the head FC model on top of the base model (this will become
  # the actual model we will train)
  model = tf.keras.Model(inputs=base.input, outputs=head)

  # loop over all layers in the base model and freeze them
  for layer in base.layers:
         layer.trainable = False
```

DenseNet121

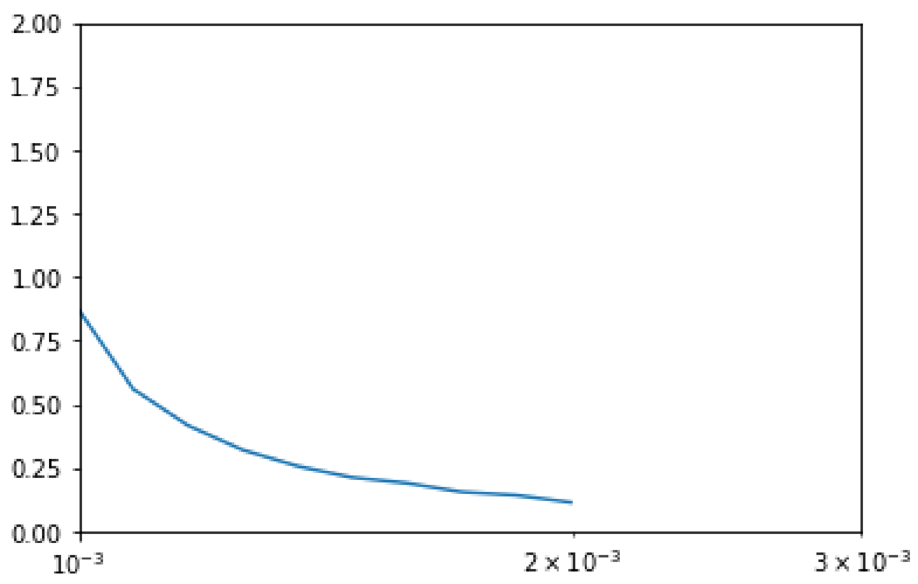
```
def create_model():
  """
  This function returns a base DenseNet121 transfer learning model trained on
  imagenet connected to a head adapted to a 3 class softmax probability
  """
  # we will start by building a DenseNet model
  # Build model freezing top layers and adding a dense layer
  # with the number of classes
  model = tf.keras.Sequential()
  # load the DenseNet121 network, ensuring the head FC layer sets are left
  # off
  base =tf.keras.applications.DenseNet121(weights="imagenet",include_top=False,
          input_tensor=tf.keras.Input(shape=(224,224,3)))

  # construct custom head of the model that will be placed on top of the
  # the base model
  head = base.output
  head = tf.keras.layers.AveragePooling2D(pool_size=(4, 4))(head)
  head = tf.keras.layers.Flatten()(head)
  head = tf.keras.layers.Dense(528, activation="relu")(head) # Increased
# complexity
  head = tf.keras.layers.Dropout(0.2)(head) # Decreased dropout
  head = tf.keras.layers.Dense(3, activation="softmax")(head)

  # place the head FC model on top of the base model (this will become
  # the actual model we will train)
  model = tf.keras.Model(inputs=base.input, outputs=head)

  # loop over all layers in the base model and freeze them
  for layer in base.layers:
          layer.trainable = False
```

Optimizers and learning rate

The Adam optimizer was chosen for this project as historically it is highly effective with image data.

Since we have a multiclass problem and we are using a softmax activation on the last layer, Categorical cross entropy was chosen as the loss function.

The TensorFlow LearningRateScheduler can be used through some trial and error to figure out, first, a good learning rate range and after that through visualization, a good starting point for the learning rate for each model, the models were remarkably similar in range in this case and a learning rate of 0.001 seemed to be right for all three, the desired learning rate is the one that shows the best loss on a decaying schedule (rates are on the x axis and loss on the y axis):



Callbacks

Callbacks were used to tune the models for optimal performance note the comments on the functions below (using python and TensorFlow code) to review what these callbacks were doing; it is important to know that parameters may need to be tweaked depending on the model and what we are trying to achieve. The TensorFlow documentation is helpful regarding recommended settings and default settings particularly for the optimizer of choice but arriving at optimal settings often depends on experimentation through iterations of training and validation for each individual model, using the parameters below worked well for VGG16, only small changes were made for the other models:

```
def get_early_stopping():
    """
    This function should return an EarlyStopping callback that stops training
when the validation (testing) accuracy has not improved in the last 3 epochs.

    """
    return tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', mode="max",
patience=3)
```

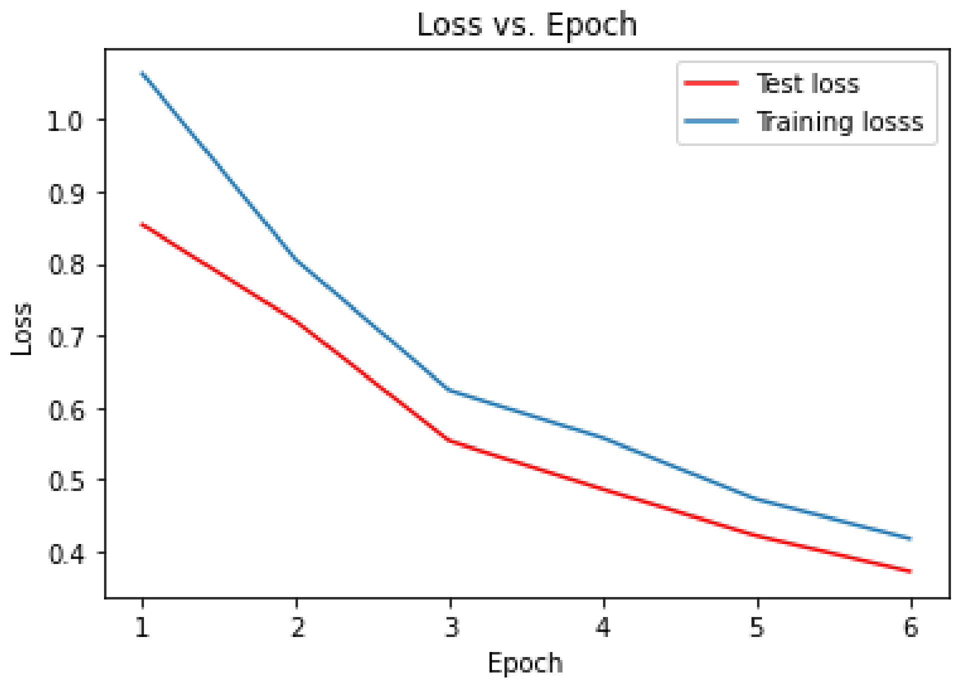
```
def get_checkpoint_best_only_conv():
    """
    This function returns a ModelCheckpoint object that:
    - saves only the weights that generate the highest validation (testing)
accuracy
    """
    checkpoint_path = '/gdrive/My Drive/covid_vgg16_no_reg_grad.pb'
    model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath=checkpoint_path,
        save_weights_only=False,
        monitor='val_accuracy',
        mode='max',
        save_best_only=True
        )

    return model_checkpoint_callback
```

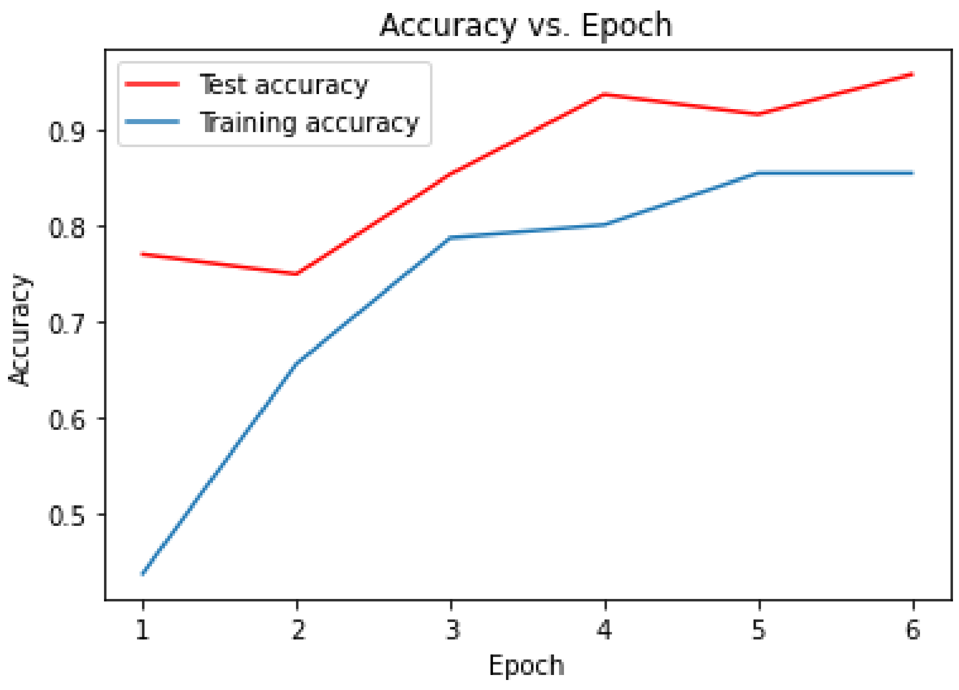
```
checkpoint_best_only = get_checkpoint_best_only_conv()
early_stopping = get_early_stopping()

# If validation loss stops improving reduce learning rate using these
# parameters

reduce_on_plateau = tf.keras.callbacks.ReduceLROnPlateau(
                              monitor='val_loss',
                              factor=1/30., patience=2,
                              verbose=1, mode='auto', min_delta=0.0001,
                              cooldown=0, min_lr=0)

# loop over all layers in the base model and freeze them

callbacks = [checkpoint_best_only, early_stopping, reduce_on_plateau]
```

Validation

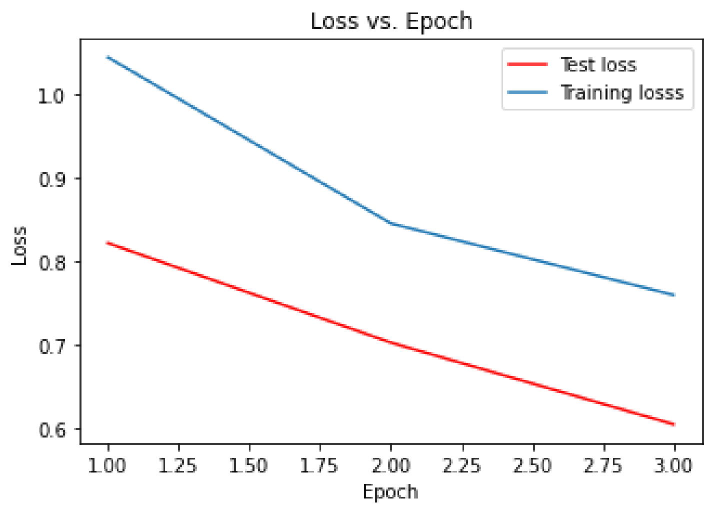
The following Loss and Accuracy plots resulted from the initial modeling efforts for each transfer learning model, note that this is validation data (test loss here refers to the validation data set as we are still trying to optimize the models):

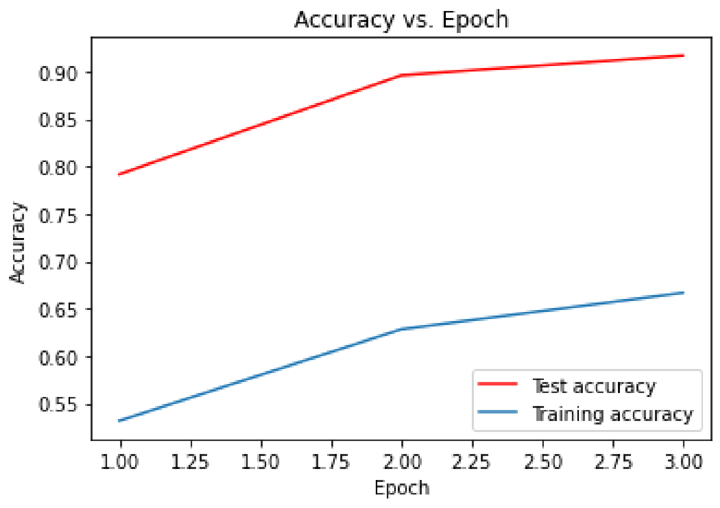
VGG16


Loss vs. Epoch
Test loss
Training losss
Loss
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
1
2
3
4
5
6

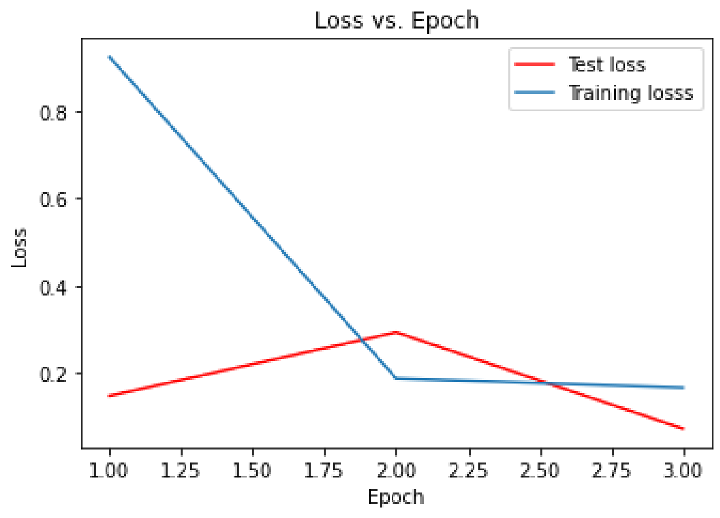

Accuracy vs. Epoch
Test accuracy
Training accuracy
Accuracy
Epoch
0.9
0.8
0.7
0.6
0.5
1
2
3
4
5
6

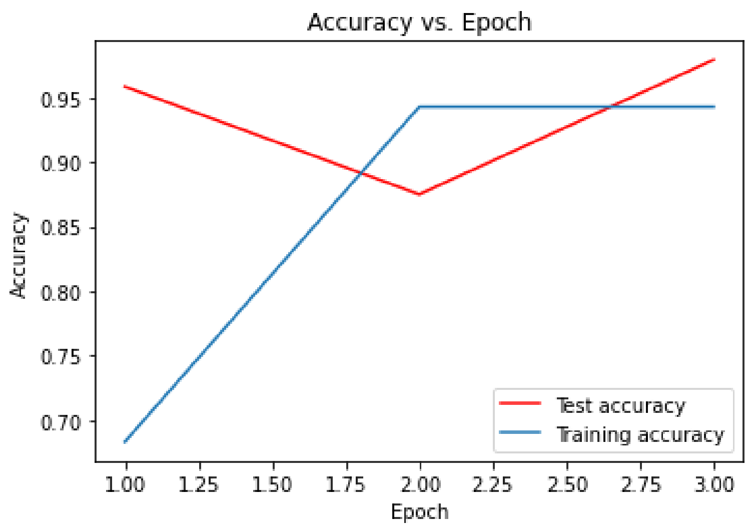
ResNet50





DenseNet121





It is important to note here that the validation data on the VGG16 and ResNet50 plots performed better than the training data on both accuracy and loss, and there is a visible gap between the blue and red lines. This can be dangerous for unseen samples since the model could be overlearning the validation data, and predicting wildly on test data, this is a phenomenon sometimes caused when there is too little data. The problem here could be that because there is too little data, and the regularization is affecting some of the learning during training such that the model is not learning enough from the data and thus we need to remove the regularization and add some model complexity if necessary, we should see the blue and red lines converge and hopefully see training data performing a little better than testing data but not the other way around. It looks like because of the additional architectural complexity that we added to the DenseNet121 and because we had a lower dropout, this effect was less of an issue for the DenseNet121 model, and we can see training and validation curves are converging.

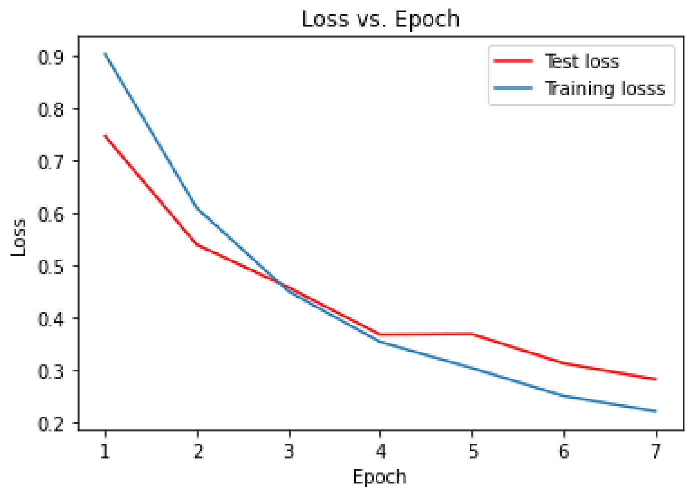
Architectural Changes and Running Validation Again

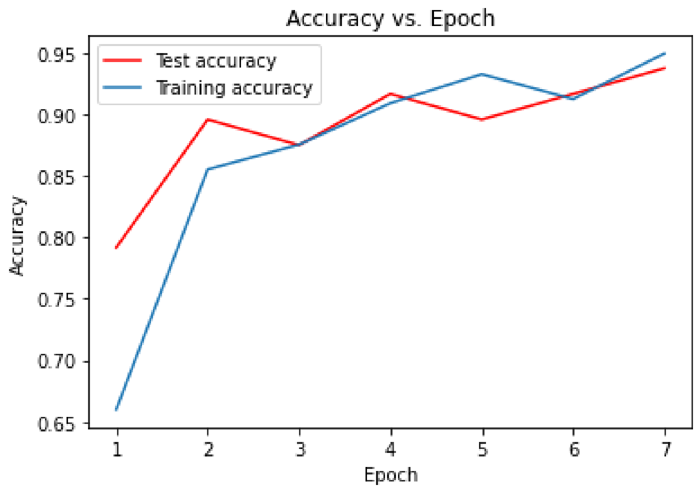
The dropout layer was removed from the VGG16 model yielding the following architecture:

```
def create_model():
  """
  This function returns a base VGG16 transfer learning model trained on
  imagenet connected to a head adapted to a 3 class softmax probability
  """
  # we will start by building a VGG16 model
  # Build model freezing top layers and adding a dense layer
  # with the number of classes
  model = tf.keras.Sequential()
  # load the VGG16 network, ensuring the head FC layer sets are left
  # off
  base = tf.keras.applications.VGG16(weights="imagenet", include_top=False,
         input_tensor=tf.keras.Input(shape=(224,224,3)))

  # construct the head of the model that will be placed on top of the
  # the base model
  head = base.output
  head = tf.keras.layers.AveragePooling2D(pool_size=(4, 4))(head)
```

```python
head = tf.keras.layers.Flatten()(head)
head = tf.keras.layers.Dense(64, activation="relu")(head)
#head = tf.keras.layers.Dropout(0.5)(head) # remove dropout
head = tf.keras.layers.Dense(3, activation="softmax")(head)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = tf.keras.Model(inputs=base.input, outputs=head)

# loop over all layers in the base model and freeze them
for layer in base.layers:
        layer.trainable = False
```

The validation test was run again with the new architecture resulting in the following plots:



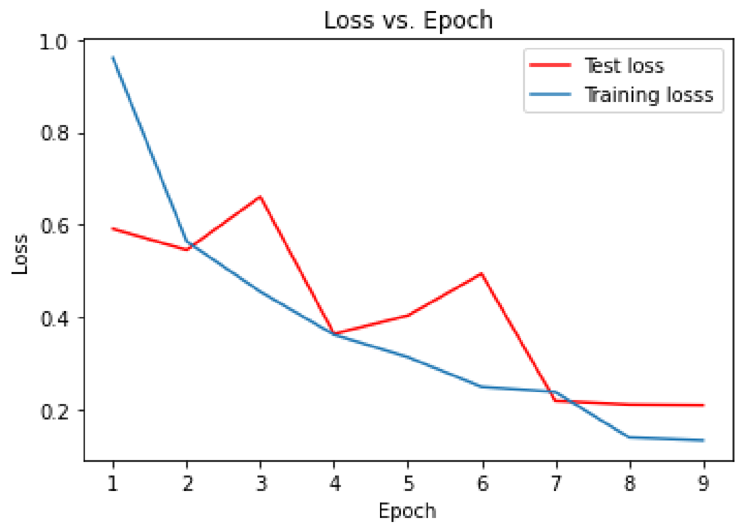


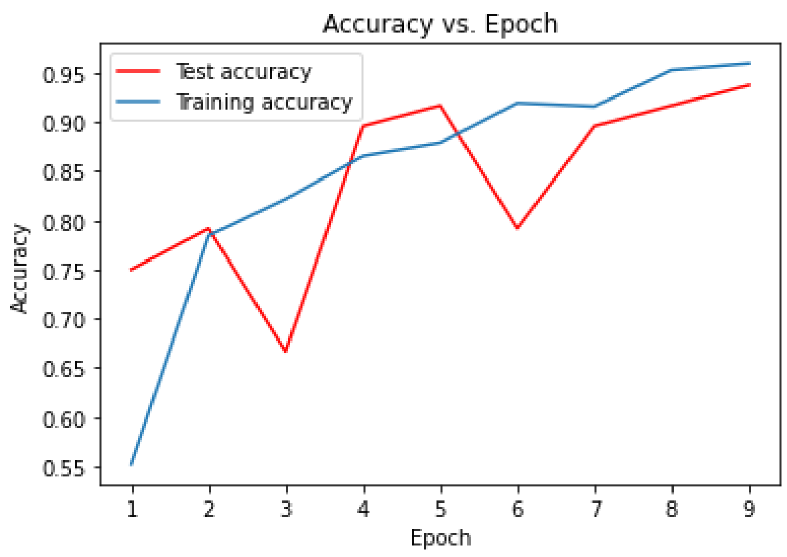
These curves do seem much better. It does appear like the dropout was affecting learning negatively. Training and testing curves are now converging and should result in better unseen sample test results.

The dropout layer was removed from the ResNet50 model and complexity was added by adding dense layers, yielding the following architecture:

```
def create_model(): """ This function returns a base ResNet50 transfer learning
model trained on imagenet connected to a custom head adapted to a 3 class
softmax probability """

# we will start by building a ResNet50 model
# Build model freezing top layers and adding a dense layer
# with the number of classes
model = tf.keras.Sequential()
# load the ResNet50 network, ensuring the head FC layer sets are left off
base = tf.keras.applications.ResNet50(weights="imagenet", include_top=False,
input_tensor=tf.keras.Input(shape=(224,224,3)))
# construct the head of the model that will be placed on top of the
# the base model
head = base.output
head = tf.keras.layers.AveragePooling2D(pool_size=(4, 4))(head)
head = tf.keras.layers.Flatten()(head)
head = tf.keras.layers.Dense(256, activation="relu")(head) # increase units
head = tf.keras.layers.Dense(512, activation="relu")(head) # increase units
head = tf.keras.layers.Dense(512, activation="relu")(head) # increase units
# head = tf.keras.layers.Dropout(0.5)(head) # remove dropout
head = tf.keras.layers.Dense(3, activation="softmax")(head)
# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = tf.keras.Model(inputs=base.input, outputs=head)
# loop over all layers in the base model and freeze them
for layer in base.layers:
      layer.trainable = False
```

The validation test was run again with the new architecture resulting in the following plots:





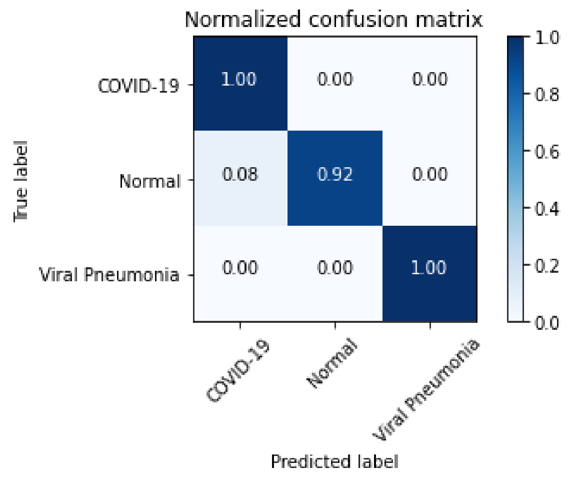
These ResNet50 curves do seem to be converging more and training data does appear to be doing better than testing data, but a lot of variance is noted on the testing data. Rather than performing further tuning or changes in the architecture (we could try batch normalization for example to see if we can get the testing curve more stable), it was decided to keep this model and have confidence that we would have solid results on at least one of our chosen architectures. This project focuses on practical solutions and if the Covid-19 model R&D needed to wrap up quickly, we could say that at this point we had reasonable evidence that at least one of the models would provide us with robust predictions and it was time to move on to testing.

No modifications were necessary for the DenseNet121 architecture.

Unseen Data Test Results

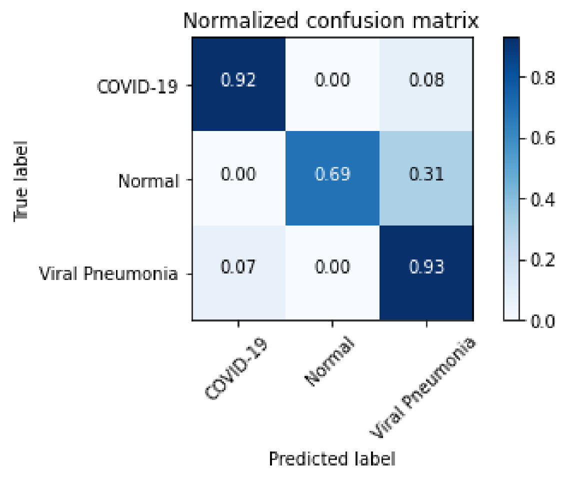
Here are the confusion matrices and classification reports on all three final models from predictions on unseen test data:

VGG16



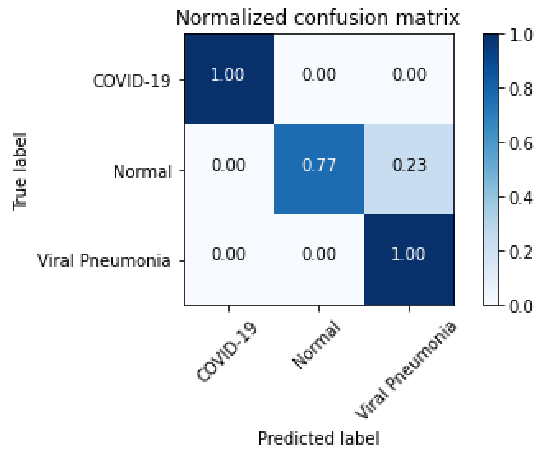
```
                 precision    recall  f1-score   support

       COVID-19       0.93      1.00      0.96        13
         Normal       1.00      0.92      0.96        13
Viral Pneumonia       1.00      1.00      1.00        14

       accuracy                           0.97        40
      macro avg       0.98      0.97      0.97        40
   weighted avg       0.98      0.97      0.97        40
```

ResNet50



```
                 precision    recall  f1-score   support

       COVID-19       0.92      0.92      0.92        13
         Normal       1.00      0.69      0.82        13
Viral Pneumonia       0.72      0.93      0.81        14

       accuracy                           0.85        40
      macro avg       0.88      0.85      0.85        40
   weighted avg       0.88      0.85      0.85        40
```

DenseNet121



```
              precision    recall  f1-score   support

       COVID-19      1.00      1.00      1.00        13
         Normal      1.00      0.77      0.87        13
Viral Pneumonia      0.82      1.00      0.90        14

       accuracy                          0.93        40
      macro avg      0.94      0.92      0.92        40
   weighted avg      0.94      0.93      0.92        40
```

Here we can see that the VGG16 has the same recall score as the DenseNet121 model on Covid-19 and Viral Pneumonia, both models scoring remarkably high on recall at 1, and they both outperform the ResNet50 model in terms of recall for the Normal class. Between the two of them the VGG16 model has the higher recall score on the normal class and beats the DenseNet21, the models score 0.92 and 0.77 respectively, and thus for prioritizing the goal (classifying positives that are actually positive, or for a more general view of it, correctly predicting classes), the VGG16 model is the best model overall.

Model Interpretability

We know that we obtained the highest scores overall on the VGG16 model to classify positives correctly, but how do we gain deeper insights on what the model was focusing on when correctly classifying those positives?

Let us get an overview of GradCAM based on the code for a strong intuition of what this algorithm actually does. Here is our VGG16 model as a Keras summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
```

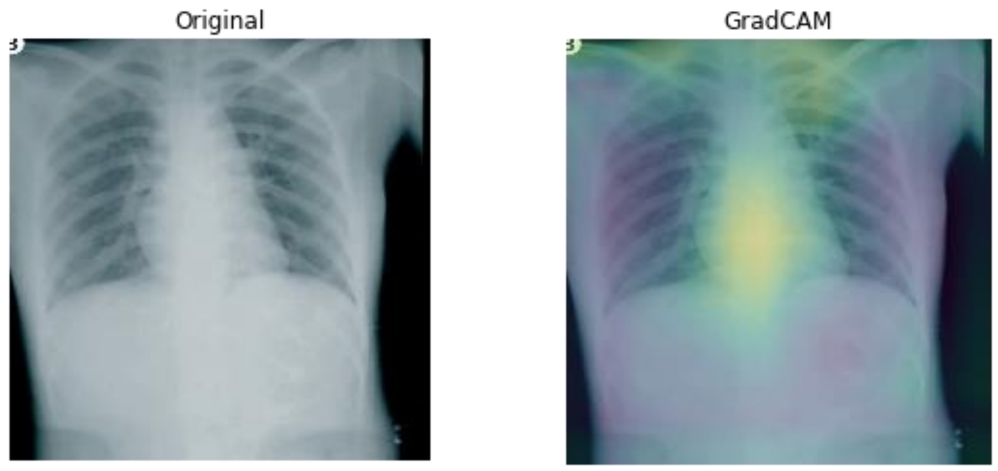
```
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
average_pooling2d (AveragePo (None, 1, 1, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 64)                32832
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 195
=================================================================
Total params: 14,747,715
Trainable params: 14,747,715
Non-trainable params: 0
```

We do not need to go over all the details of the algorithm to get an intuitive understanding. GradCam looks at the gradient activations, you can map them to their relative position in the image. If you home in on the very last convolutional layer on the model summary above, this a good one to select since it is the last convolutional layer in the network, we know it will have very well-defined features being extracted, and you would use this layer to get the activation map with help from the algorithm, this map is spatially correct, so then an activation heatmap can be created and we can overlay this on the original image. Here are some snippets of the code that give a great intuition of this process – note the use of cv2 a popular python image processing library:

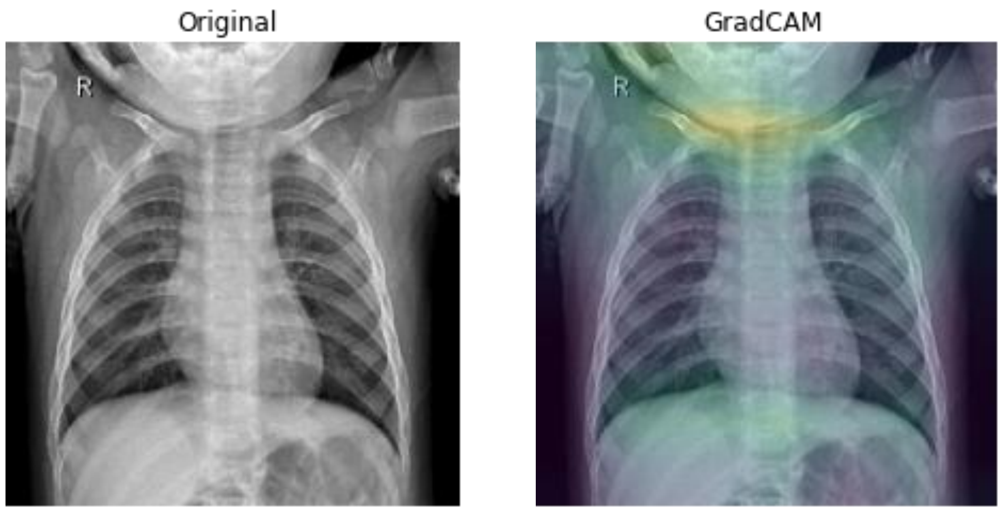
```
from tensorflow.keras import Model

im = load_image_normalize(im_path, mean, std) cam = grad_cam(model, im, 0,
'block5_conv3') # covid 19 is class 0

#Select original image
orig = cv2.imread(image_dir + '/COVID-19 (83).png')

# resize the resulting heatmap to the original input image dimensions
# and then overlay heatmap on top of the image
heatmap = cv2.resize(cam, (orig.shape[1], orig.shape[0]))
(heatmap, output) = overlay_heatmap(heatmap, orig, alpha=0.5)
```

The GradCAM algorithm was used on the VGG16 model in this way to visualize the activation maps of Covid-19 images. We can see in what regions of the images the activations were strongest. The color depicting stronger activations is bright yellow on these images. Images like these can be shared with the medical team to assess the activation regions, not only to verify that the data matches with what has been observed when dealing with Covid-19 patients but also to identify potentially other areas of concern and be able to proceed more efficiently and expeditiously. Here are some GradCAM examples of the different classes correctly identified as positive by the VGG16 model: I would put normal first.

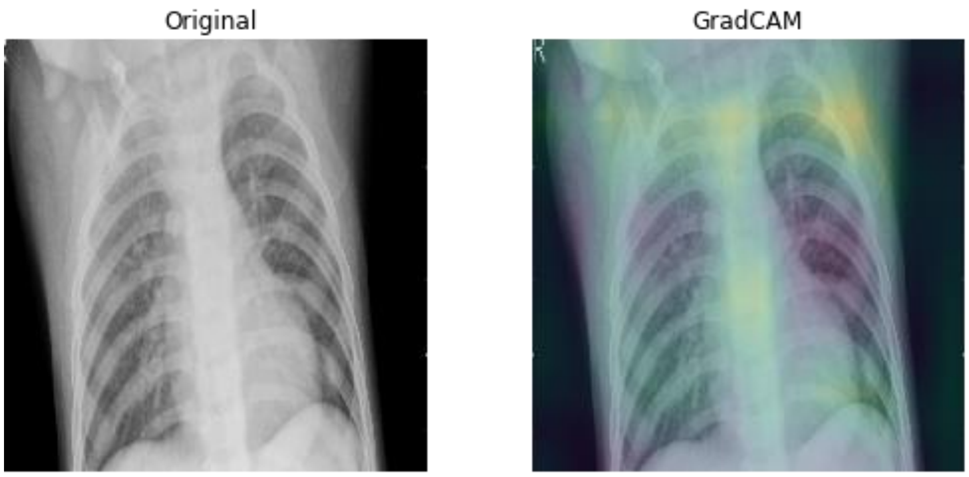
Covid-19



Normal



Viral Pneumonia



Deployment

Lastly a test deployment with TensorFlow serving was coded up to test this capability. A cloud deployment can make uploading images and getting predictions possible from any device. It is important however, to note the seriousness of the deployment protocol and the fact that a medical team and a data science team should all be in consensus prior to such an effort. However, it is impossible to ignore the flexibility that a cloud deployment would give to clinicians/radiologists. Here is some code from the post request showing how simple it can be to send the image from any device and obtain a prediction with some minor coding, of course this can be further configured to make it very straight forward for the people doing the work and the prediction is available in minutes:

```
import json

data = json.dumps({"signature_name": "serving_default", "instances": input_image.tolist()})

import requests

headers = {"content-type": "application/json"}

r = requests.post('http://localhost:8501/v1/models/covid_vgg16_no_reg_grad:predict', data=data, headers=headers

j = r.json()

pred = np.array(j['predictions'])

pred = pred.argmax(axis=1)

class_names = ['COVID-19', 'Normal', 'Viral Pneumonia']

pred = [class_names[i] for i in pred]

print('The prediction for this image is: ',pred[0])

The prediction for this image is:  COVID-19
```

Conclusion

The VGG16 deep learning model can be a useful addition to the clinician/radiologist toolkit. This model can be trained and used to classify Covid-19 on chest X-Ray images and other diseases that might be similar to Covid-19 to assist with diagnostics. Transfer learning allows the model to perform well even with limited Covid-19 data. The model can obtain the following results on unseen test data with the small number of images that are available today: Precision 0.93, Recall 1.00, F1 0.96, Accuracy 0.97. The Model offers the following: automates image processing tasks, serves predictions from the cloud using any device, provides activation regions associated with image classification for interpretation.

Potential future studies to improve interpretation could be the addition of image segmentation that can be used to identify anatomical regions. New models can be created that identify anatomical regions and the activation regions obtained using the GradCAM algorithm can be superimposed to immediately visualize the names of these regions.