

Phase 4 Final Project Submission

Image Classification with Deep Learning

Student Name: Jason Lombino

Business Application

The Bentley Health Foundation is a non-profit organization with the goal of preventing infectious diseases. Childhood pneumonia is currently one of the main focuses of the Bentley Health Foundation because it kills more children than any other infectious disease worldwide according to [UNICEF](https://data.unicef.org/topic/child-health/pneumonia/) (<https://data.unicef.org/topic/child-health/pneumonia/>). Identifying cases of pneumonia quickly and accurately is a vital step in preventing deaths.

While chest x-rays are simple to take and commonly used to diagnose cases of pneumonia, they need to be interpreted by specialists. Unfortunately, these specialists are frequently not available in rural areas and developing countries. In order to combat this issue, the Bentley Health Foundation is exploring using machine learning to detect signs of pneumonia in chest x-ray images. If successful, the Bentley Health Foundation's model can assist in diagnosing cases of pneumonia so patients can receive the life-saving treatment they need without having to risk waiting for or traveling to a specialist.

Provided with a patient's chest x-ray image, the model should return the probability that the patient has pneumonia.

Import Libraries

```
In [1]: import os
import warnings
import matplotlib.pyplot as plt
import numpy as np
from lime import lime_image
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import metrics
from tensorflow.keras import preprocessing
from tensorflow.keras import callbacks
from tensorflow.keras import optimizers
from tensorflow.keras import applications
from tensorflow.keras import utils
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay
from skimage.segmentation import mark_boundaries

warnings.filterwarnings("ignore")
%matplotlib inline
```

Data

The following dataset (<https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia>) was provided for modeling. It contains thousands of chest x-rays from children with and without pneumonia. The data provided is separated into train, test, and validation folders. However, the provided validation dataset was too small to be of any use. Prior to performing any modeling I manually moved 20% of the training data into the validation folder. The remaining images in the training folder were used to train models, and the test data was used as provided. The following code loads the image data.

```
In [2]: # Procedure for moving files to validation may vary by OS.
# Procedure for linux is as follows:

# Run this command in a terminal in chest_xray/train/NORMAL
# 268 Images is 20% of the normal train data
# mv `ls | head -268` ../../val/NORMAL

# Run this command in a terminal in chest_xray/train/PNEUMONIA
# 775 Images is 20% of the pneumonia train data
# mv `ls | head -775` ../../val/PNEUMONIA
```

```
In [3]: # Data should be in three folders (train, test, and val)
train_path = 'chest_xray/train'
test_path = 'chest_xray/test'
val_path = 'chest_xray/val'
```

```
In [4]: # Data generator with some preprocessing steps for training data
train_generator = preprocessing.image.ImageDataGenerator(
    rescale=1./255,
    horizontal_flip=True,
    rotation_range=45,
    zoom_range=0.3
)

train_ds = train_generator.flow_from_directory(
    train_path,
    target_size=(256,256),
    class_mode='binary',
)
# Class sizes for pie chart
train_norm = len(os.listdir(train_path + '/NORMAL'))
train_sick = len(os.listdir(train_path + '/PNEUMONIA'))
```

Found 4175 images belonging to 2 classes.

```
In [5]: # Data generator without preprocessing steps for validation data
val_generator = preprocessing.image.ImageDataGenerator(
    rescale=1./255
)

# Shuffle is not needed for test and validation data
val_ds = val_generator.flow_from_directory(
    val_path,
    target_size=(256,256),
    class_mode='binary',
    shuffle=False
)
# Class sizes for pie chart
val_norm = len(os.listdir(val_path + '/NORMAL'))
val_sick = len(os.listdir(val_path + '/PNEUMONIA'))
```

Found 1057 images belonging to 2 classes.

```
In [6]: # Data generator without preprocessing steps for test data
test_generator = preprocessing.image.ImageDataGenerator(
    rescale=1./255
)

# Shuffle is not needed for test and validation data
test_ds = test_generator.flow_from_directory(
    test_path,
    target_size=(256,256),
    class_mode='binary',
    shuffle=False
)
# Class sizes for pie chart
test_norm = len(os.listdir(test_path + '/NORMAL'))
test_sick = len(os.listdir(test_path + '/PNEUMONIA'))
```

Found 624 images belonging to 2 classes.

Data Visualizations

There is approximately a 3:1 class imbalance in the train and validation datasets, and approximately a 2:1 class imbalance in the test dataset. While this is not optimal, It should not skew the evaluation metrics too much.

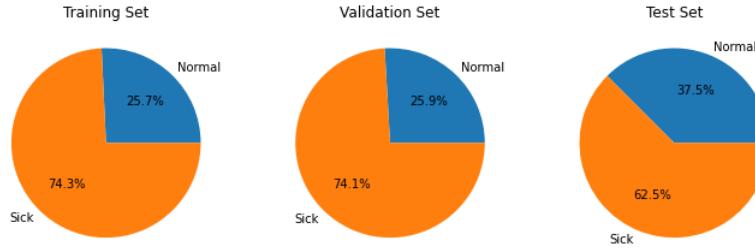
```
In [7]: data_fig, data_ax = plt.subplots(1,3,figsize=(12,4))

# Training class sizes
data_ax[0].pie([train_norm,train_sick],labels=['Normal','Sick'],autopct='%1.1f%%');
data_ax[0].set_title('Training Set');

# Validation class sizes
data_ax[1].pie([val_norm,val_sick],labels=['Normal','Sick'],autopct='%1.1f%%');
data_ax[1].set_title('Validation Set');

# Test class sizes
data_ax[2].pie([test_norm,test_sick],labels=['Normal','Sick'],autopct='%1.1f%%');
data_ax[2].set_title('Test Set');

data_fig.set_facecolor('#FFFFFF')
```



Looking at some examples of normal x-rays.

```
In [8]: normal_images = []

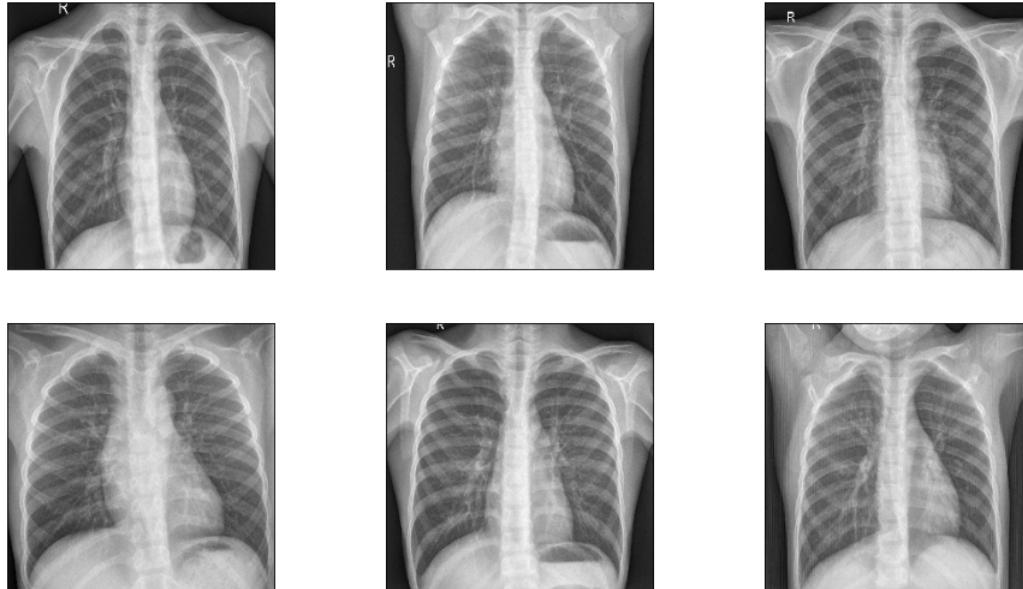
# Pull 6 normal images from the test set
# Test set is ordered so normal images are at the beginning
for image in test_ds[0][0][:6]:
    normal_images.append(image)

normal_fig,normal_ax = plt.subplots(2,3,figsize=(16,9))

# Plot the 6 normal images pulled from the test set
for idx,image in enumerate(normal_images[:6]):
    active_axis = normal_ax[idx//3][idx%3]
    active_axis.imshow(image)
    active_axis.set_xticks([])
    active_axis.set_yticks([])

normal_fig.set_facecolor('#FFFFFF')
normal_fig.suptitle('Normal X-Rays', fontsize=40);
```

Normal X-Rays



Looking at some examples of Pneumonia x-rays.

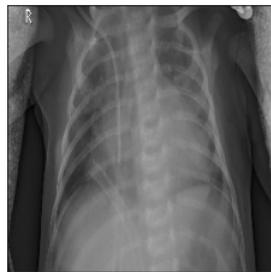
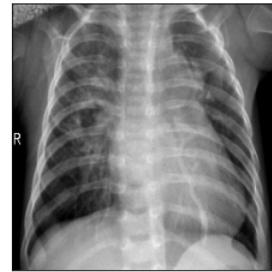
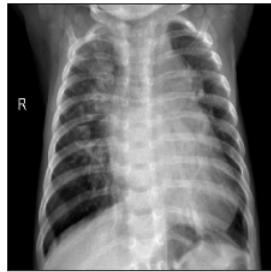
```
In [9]: sick_images = []

# Pull 6 pneumonia images from the test set
# Test set is ordered so sick images are at the end
for image in test_ds[len(test_ds)-1][0][:6]:
    sick_images.append(image)

sick_fig,sick_ax = plt.subplots(2,3,figsize=(16,9))

# Plot the 6 pneumonia images pulled from the test set
for idx,image in enumerate(sick_images[:6]):
    active_axis = sick_ax[idx//3][idx%3]
    active_axis.imshow(image)
    active_axis.set_xticks([])
    active_axis.set_yticks([])
sick_fig.set_facecolor('#FFFFFF')
sick_fig.suptitle('Pneumonia X-Rays', fontsize=40);
```

Pneumonia X-Rays



The training set is preprocessed before it is used to train the model. They can be horizontally flipped, zoomed in or out up to 30%, or rotated up to 45 degrees in either direction.

```
In [10]: # Pull the first 6 images from the training set
train_prep = train_ds[0][0][:6]

train_fig,train_ax = plt.subplots(2,3,figsize=(16,9))

# Plot the 6 transformed images pulled from the training set
for idx,image in enumerate(train_prep[:6]):
    active_axis = train_ax[idx//3][idx%3]
    active_axis.imshow(image)
    active_axis.set_xticks([])
    active_axis.set_yticks([])
train_fig.set_facecolor('#FFFFFF')
train_fig.suptitle('Training Image Transformations', fontsize=40);
```

Training Image Transformations



Putting it all together to make a useful visualization for the README.

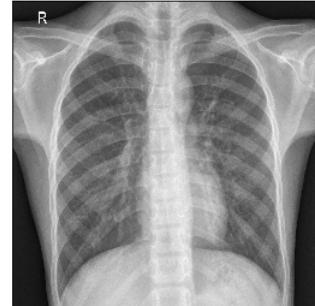
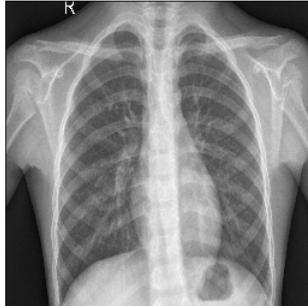
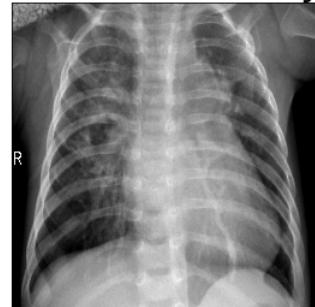
```
In [11]: comb_fig, comb_ax = plt.subplots(3,3,figsize=(16,16))

# Plot the 3 normal images pulled from the test set
for idx,image in enumerate(normal_images[:3]):
    active_axis = comb_ax[0][idx]
    active_axis.imshow(image)
    active_axis.set_xticks([])
    active_axis.set_yticks([])

# Plot the 3 pneumonia images pulled from the test set
for idx,image in enumerate(sick_images[:3]):
    active_axis = comb_ax[1][idx]
    active_axis.imshow(image)
    active_axis.set_xticks([])
    active_axis.set_yticks([])

# Plot the 3 transformed images pulled from the training set
for idx,image in enumerate(train_prep[:3]):
    active_axis = comb_ax[2][idx]
    active_axis.imshow(image)
    active_axis.set_xticks([])
    active_axis.set_yticks([])

comb_ax[0][1].set_title('Normal X-Rays',fontsize=32)
comb_ax[1][1].set_title('Pneumonia X-Rays',fontsize=32)
comb_ax[2][1].set_title('Training Image Transformations',fontsize=32)
comb_fig.set_facecolor('#FFFFFF')
```

Normal X-Rays**Pneumonia X-Rays****Training Image Transformations**

Iterative Modeling

Despite the class imbalance in both the training and test data, **Accuracy** was the primary metric used to evaluate models. The model should make as many correct decisions as possible. While false negatives are more severe than false positives in this case, favoring recall too heavily will likely result in many patients undergoing treatment for a disease they do not have wasting valuable resources.

I made several models while trying to determine the best convolutional neural network (CNN) architecture and hyperparameters for classifying the x-rays. The training and validation results for each of the models can be seen in the table below. Each model has a corresponding jupyter notebook located [here \(notebooks\)](#).

01 (notebooks/01_basic_model.ipynb) Basic CNN

This is a very basic model that used a small neural network and did not preprocess images. This was used as a starting point for the rest of my models, but was not considered when selecting a final model.

02 (notebooks/02_image_preprocessing.ipynb) Basic CNN Image Preprocessing

This model attempts to improve on the basic CNN by adding image preprocessing to reduce overfitting.

03 (notebooks/03_larger_network.ipynb) Larger CNN

This model attempts to improve on the basic CNN by increasing the size of the neural network. Both the number of layers and the size of each layer is increased.

04 (notebooks/04_tanh_activation.ipynb) Larger CNN tanh Activation

This model attempts to improve on the basic CNN by using a tanh activation function for each layer rather than relu. It performs much worse overall in testing and on validation data.

05 (notebooks/05_dropout.ipynb) Larger CNN Dropout

This model attempts to improve on the basic CNN by adding dropout layers to reduce overfitting.

06 (notebooks/06_larger_stride.ipynb) Larger CNN Larger Stride

This model attempts to improve on the basic CNN by increasing the stride of the filter in each convolutional layer to reduce overfitting.

07 (notebooks/07_double_conv.ipynb) Larger CNN Double Convolutional Layers

This model attempts to improve on the basic CNN by adding additional convolutional layers.

08 (notebooks/08_even_larger_network.ipynb) Even Larger CNN

This model attempts to improve on the basic CNN by adding additional convolutional and dense layers. Both the number of layers and the size of each layer is increased.

09 (notebooks/09_transfer_xc.ipynb) Transfer Learning Xception

This model attempts to improve on the basic CNN by using the [Xception](https://keras.io/api/applications/xception/) (<https://keras.io/api/applications/xception/>) network as its convolutional base.

10 (notebooks/10_transfer_vgg.ipynb) Transfer Learning VGG16

This model attempts to improve on the basic CNN by using the [VGG16](https://keras.io/api/applications/vgg/) (<https://keras.io/api/applications/vgg/>) network as its convolutional base.

11 (notebooks/11_vgg_decay.ipynb) Transfer Learning VGG16 Decaying Learning Rate

This model attempts to improve on the basic CNN by using the [VGG16](https://keras.io/api/applications/vgg/) (<https://keras.io/api/applications/vgg/>) network as its convolutional base. It also decreases the learning rate each epoch.

Comparison of Models

Model Name	% Train Accuracy	% Val Accuracy
01 (Not Considered) Basic CNN	98.8	96.7
02 Basic CNN Image Preprocessing	93.5	91.2
03 Larger CNN	93.5	94.0
-4 Larger CNN tanh Activation	74.3	74.1
05 Larger CNN Dropout	93.3	94.0
06 Larger CNN Larger Stride	95.3	93.4
07 Larger CNN Double Convolutional Layers	95.5	94.0
08 Even Larger CNN	94.8	94.5
09 Transfer Learning Xception	96.6	94.0
10 Transfer Learning VGG16	95.0	95.7
11 Transfer Learning VGG16 Decaying Learning Rate	94.7	94.2

Second Best Model - 08 Even Larger CNN

The second best performing model on validation data was 08 Even Larger CNN. The following code creates the model.

```
In [12]: nt_model = models.Sequential()

# First convolutional block
nt_model.add(layers.Conv2D(32, (1, 1), activation='relu', padding='same',
                         input_shape=(256, 256, 3)))
nt_model.add(layers.Conv2D(32, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.MaxPooling2D((2, 2)))

# Second convolutional block
nt_model.add(layers.Conv2D(64, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.Conv2D(64, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.MaxPooling2D((2, 2)))

# Third convolutional block
nt_model.add(layers.Conv2D(128, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.Conv2D(128, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.MaxPooling2D((2, 2)))

# Fourth convolutional block
nt_model.add(layers.Conv2D(256, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.Conv2D(256, (1, 1), activation='relu', padding='same'))
nt_model.add(layers.MaxPooling2D((2, 2)))

# Densely connected block with dropout
nt_model.add(layers.Flatten())
nt_model.add(layers.Dense(512, activation='relu'))
nt_model.add(layers.Dropout(rate=0.25))
nt_model.add(layers.Dense(512, activation='relu'))
nt_model.add(layers.Dropout(rate=0.25))
nt_model.add(layers.Dense(256, activation='relu'))
nt_model.add(layers.Dropout(rate=0.25))
nt_model.add(layers.Dense(128, activation='relu'))
nt_model.add(layers.Dropout(rate=0.25))
nt_model.add(layers.Dense(64, activation='relu'))
nt_model.add(layers.Dropout(rate=0.25))
nt_model.add(layers.Dense(32, activation='relu'))
nt_model.add(layers.Dense(1, activation='sigmoid'))
```

Using checkpoints to save the epoch that does best on the validation data.

```
In [13]: # Checkpoint the model after each epoch
nt_checkpoint = callbacks.ModelCheckpoint(
    monitor='val_loss',
    filepath='save_data/nt_model.hdf5',
    save_best_only=True,
    verbose=0
)

nt_callbacks = [nt_checkpoint]

# Metrics are Accuracy, Precision, and Recall
use_metrics = [metrics.BinaryAccuracy(),
               metrics.Precision(),
               metrics.Recall()]
```

```
In [14]: # Compile the model with the Adam optimizer
nt_model.compile(
    loss='binary_crossentropy',
    optimizer=optimizers.Adam(),
    metrics=[use_metrics]
)

# Print overview of the model
print(nt_model.summary())
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 32)	128
conv2d_1 (Conv2D)	(None, 256, 256, 32)	1056
max_pooling2d (MaxPooling2D)	(None, 128, 128, 32)	0
conv2d_2 (Conv2D)	(None, 128, 128, 64)	2112
conv2d_3 (Conv2D)	(None, 128, 128, 64)	4160
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 64)	0
conv2d_4 (Conv2D)	(None, 64, 64, 128)	8320
conv2d_5 (Conv2D)	(None, 64, 64, 128)	16512
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 128)	0
conv2d_6 (Conv2D)	(None, 32, 32, 256)	33024
conv2d_7 (Conv2D)	(None, 32, 32, 256)	65792
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 256)	0
flatten (Flatten)	(None, 65536)	0
dense (Dense)	(None, 512)	33554944
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 512)	262656
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131328
dropout_2 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 128)	32896
dropout_3 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 64)	8256
dropout_4 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 32)	2080
dense_6 (Dense)	(None, 1)	33

```
=====
Total params: 34,123,297
Trainable params: 34,123,297
Non-trainable params: 0
```

None

```
In [15]: utils.plot_model(model=nt_model,to_file='images/nt_model.png',show_shapes=True,dpi=64)
```

conv2d_input	input:	[(None, 256, 256, 3)]	[(None, 256, 256, 3)]
<hr/>			
Conv2D	output:	(None, 256, 256, 3)	(None, 256, 256, 32)
<hr/>			
conv2d_1	input:	(None, 256, 256, 32)	(None, 256, 256, 32)
Conv2D	output:	(None, 256, 256, 32)	(None, 256, 256, 32)
<hr/>			
max_pooling2d	input:	(None, 256, 256, 32)	(None, 128, 128, 32)
MaxPooling2D	output:	(None, 256, 256, 32)	(None, 128, 128, 32)
<hr/>			
conv2d_2	input:	(None, 128, 128, 32)	(None, 128, 128, 64)
Conv2D	output:	(None, 128, 128, 32)	(None, 128, 128, 64)

It may be necessary to change this cell to get the notebook to run on your computer. If you do not have a working GPU change '/GPU:0' to '/CPU:0'. This cell fits the model and saves the results to history. It then loads the best epoch from the checkpoint.

```
In [16]: # Fit the model and save the results to history
with tf.device('/GPU:0'):
    nt_history = nt_model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=20,
        callbacks=nt_callbacks
    )

# Load the best model based on validation loss
nt_model.load_weights('save_data/nt_model.hdf5')

Epoch 1/20
131/131 [=====] - 48s 323ms/step - loss: 0.5673 - binary_accuracy: 0.7351 - precision: 0.7421 - recall: 0.9858 - val_loss: 0.5321 - val_binary_accuracy: 0.7408 - val_precision: 0.7408 - val_recall: 1.0000
Epoch 2/20
131/131 [=====] - 42s 317ms/step - loss: 0.4544 - binary_accuracy: 0.7451 - precision: 0.7460 - recall: 0.9958 - val_loss: 0.4186 - val_binary_accuracy: 0.7947 - val_precision: 0.7973 - val_recall: 0.9693
Epoch 3/20
131/131 [=====] - 41s 315ms/step - loss: 0.3774 - binary_accuracy: 0.8347 - precision: 0.8696 - recall: 0.9145 - val_loss: 0.3625 - val_binary_accuracy: 0.8439 - val_precision: 0.8480 - val_recall: 0.9617
Epoch 4/20
131/131 [=====] - 42s 317ms/step - loss: 0.3141 - binary_accuracy: 0.8731 - precision: 0.9018 - recall: 0.9303 - val_loss: 0.2467 - val_binary_accuracy: 0.8978 - val_precision: 0.9801 - val_recall: 0.8799
Epoch 5/20
131/131 [=====] - 42s 318ms/step - loss: 0.2654 - binary_accuracy: 0.9030 - precision: 0.9255 - recall: 0.9455 - val_loss: 0.1856 - val_binary_accuracy: 0.9224 - val_precision: 0.9630 - val_recall: 0.9310
Epoch 6/20
131/131 [=====] - 41s 312ms/step - loss: 0.2437 - binary_accuracy: 0.9075 - precision: 0.9392 - recall: 0.9361 - val_loss: 0.1878 - val_binary_accuracy: 0.9347 - val_precision: 0.9773 - val_recall: 0.9336
Epoch 7/20
131/131 [=====] - 42s 316ms/step - loss: 0.2162 - binary_accuracy: 0.9190 - precision: 0.9446 - recall: 0.9465 - val_loss: 0.1790 - val_binary_accuracy: 0.9272 - val_precision: 0.9836 - val_recall: 0.9170
Epoch 8/20
131/131 [=====] - 42s 320ms/step - loss: 0.2168 - binary_accuracy: 0.9176 - precision: 0.9431 - recall: 0.9461 - val_loss: 0.1717 - val_binary_accuracy: 0.9376 - val_precision: 0.9825 - val_recall: 0.9323
Epoch 9/20
131/131 [=====] - 42s 316ms/step - loss: 0.2072 - binary_accuracy: 0.9224 - precision: 0.9483 - recall: 0.9471 - val_loss: 0.1818 - val_binary_accuracy: 0.9357 - val_precision: 0.9710 - val_recall: 0.9413
Epoch 10/20
131/131 [=====] - 42s 317ms/step - loss: 0.2068 - binary_accuracy: 0.9231 - precision: 0.9481 - recall: 0.9484 - val_loss: 0.1622 - val_binary_accuracy: 0.9338 - val_precision: 0.9917 - val_recall: 0.9183
Epoch 11/20
131/131 [=====] - 41s 314ms/step - loss: 0.2058 - binary_accuracy: 0.9200 - precision: 0.9499 - recall: 0.9419 - val_loss: 0.2523 - val_binary_accuracy: 0.9063 - val_precision: 0.9453 - val_recall: 0.9272
Epoch 12/20
131/131 [=====] - 42s 317ms/step - loss: 0.2150 - binary_accuracy: 0.9224 - precision: 0.9457 - recall: 0.9500 - val_loss: 0.1999 - val_binary_accuracy: 0.9357 - val_precision: 0.9917 - val_recall: 0.9208
Epoch 13/20
131/131 [=====] - 42s 316ms/step - loss: 0.1889 - binary_accuracy: 0.9293 - precision: 0.9462 - recall: 0.9594 - val_loss: 0.1440 - val_binary_accuracy: 0.9451 - val_precision: 0.9739 - val_recall: 0.9515
Epoch 14/20
131/131 [=====] - 41s 312ms/step - loss: 0.1884 - binary_accuracy: 0.9301 - precision: 0.9529 - recall: 0.9529 - val_loss: 0.1924 - val_binary_accuracy: 0.9224 - val_precision: 0.9930 - val_recall: 0.9017
Epoch 15/20
131/131 [=====] - 41s 314ms/step - loss: 0.1863 - binary_accuracy: 0.9269 - precision: 0.9478 - recall: 0.9542 - val_loss: 0.1522 - val_binary_accuracy: 0.9404 - val_precision: 0.9826 - val_recall: 0.9361
Epoch 16/20
131/131 [=====] - 42s 318ms/step - loss: 0.1791 - binary_accuracy: 0.9337 - precision: 0.9528 - recall: 0.9581 - val_loss: 0.1790 - val_binary_accuracy: 0.9205 - val_precision: 0.9888 - val_recall: 0.9029
Epoch 17/20
131/131 [=====] - 42s 319ms/step - loss: 0.1725 - binary_accuracy: 0.9325 - precision: 0.9587 - recall: 0.9500 - val_loss: 0.1751 - val_binary_accuracy: 0.9234 - val_precision: 0.9930 - val_recall: 0.9029
Epoch 18/20
131/131 [=====] - 42s 317ms/step - loss: 0.1872 - binary_accuracy: 0.9301 - precision: 0.9558 - recall: 0.9497 - val_loss: 0.1759 - val_binary_accuracy: 0.9262 - val_precision: 0.9757 - val_recall: 0.9234
Epoch 19/20
131/131 [=====] - 41s 315ms/step - loss: 0.1840 - binary_accuracy: 0.9298 - precision: 0.9541 - recall: 0.9513 - val_loss: 0.1794 - val_binary_accuracy: 0.9319 - val_precision: 0.9671 - val_recall: 0.9400
Epoch 20/20
131/131 [=====] - 41s 315ms/step - loss: 0.1680 - binary_accuracy: 0.9363 - precision: 0.9577 - recall: 0.9565 - val_loss: 0.1657 - val_binary_accuracy: 0.9376 - val_precision: 0.9799 - val_recall: 0.9349
```

The following cell shows the training and validation accuracy and loss for each epoch. Accuracy should trend upwards and loss should trend downwards with more epochs. Validation accuracy will generally be lower than training accuracy, and validation loss will generally be higher than training loss.

```
In [17]: # Grab model results from history dict for plotting
nt_acc = nt_history.history['binary_accuracy']
nt_val_acc = nt_history.history['val_binary_accuracy']
nt_loss = nt_history.history['loss']
nt_val_loss = nt_history.history['val_loss']
nt_epochs = range(len(nt_acc))

# Plot train and val accuracy for each epoch
nt_fig, nt_ax = plt.subplots(1,2,figsize=(16,8))
nt_ax[0].plot(nt_epochs, nt_acc, label='Training accuracy');
nt_ax[0].plot(nt_epochs, nt_val_acc, label='Validation accuracy');
nt_ax[0].set_title('Training and validation accuracy', fontsize=24);
nt_ax[0].legend();

# Plot train and val loss for each epoch
nt_ax[1].plot(nt_epochs, nt_loss, label='Training loss');
nt_ax[1].plot(nt_epochs, nt_val_loss, label='Validation loss');
nt_ax[1].set_title('Training and validation loss', fontsize=24);
nt_ax[1].legend();
```



Model Results

```
In [18]: # Evaluate the model on training set
nt_train_results = nt_model.evaluate(train_ds)

131/131 [=====] - 37s 286ms/step - loss: 0.1690 - binary_accuracy: 0.9408 - precision: 0.9484 - recall: 0.9732

In [19]: # Evaluate the model on validation set
nt_val_res = nt_model.evaluate(val_ds)

34/34 [=====] - 3s 92ms/step - loss: 0.1440 - binary_accuracy: 0.9451 - precision: 0.9739 - recall: 0.9515

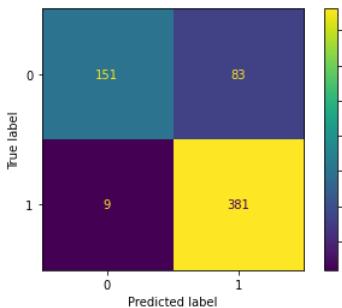
In [20]: # Evaluate the model on test set
nt_test_res = nt_model.evaluate(test_ds)

20/20 [=====] - 2s 76ms/step - loss: 0.4751 - binary_accuracy: 0.8526 - precision: 0.8211 - recall: 0.9769
```

```
In [21]: # Get predictions for test set and actual values
nt_test_predictions = np.round(nt_model.predict(test_ds),0)
test_actual = test_ds.labels

# Generate confusion matrix for test set
nt_c_matrix = confusion_matrix(test_actual,nt_test_predictions)
ConfusionMatrixDisplay(nt_c_matrix).plot()
```

```
Out[21]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f59305b1b80>
```



Lime (<https://github.com/marcotcr/lime>) Explanation of Results

```
In [22]: explainer = lime_image.LimeImageExplainer()
```

A helper function to extract certain images from the test dataset. For a given model the function can be used to extract:

- True Positive Examples
- True Negative Examples
- False Positive Examples
- False Negative Examples

```
In [23]: def extract_selected(model,true,predicted):
    selected = []

    #Loop through each image in each batch of the test dataset
    for batch in range(len(test_ds)):
        for idx,image in enumerate(test_ds[batch][0]):

            # Stop when 6 images are selected and return them
            if len(selected) >= 6:
                return(selected)

            # Select image if true and predicted classes match what was specified
            else:
                if np.round(model.predict(np.expand_dims(image, axis=0)),0) == predicted \
                    and test_ds[batch][1][idx] == true:
                    selected.append(image)

            # Stop when 6 images are selected and return them
            if len(selected) >= 6:
                return(selected)

    # Return all images if less than 6 were selected
    return(selected)
```

A helper function that uses lime to explain the selected images. It returns a formatted figure with the explanations.

```
In [24]: def lime_explain(model,selected_images,title):
    fig,ax = plt.subplots(2,3,figsize=(12,6))

    # Loop through each of up to 6 provided image
    for idx, image in enumerate(selected_images):
        active_axis = ax[idx//3][idx%3]
        active_axis.set_xticks([])
        active_axis.set_yticks([])

        # Generate explanation for prediction using Lime image explainer
        explanation = explainer.explain_instance(
            image, model.predict, top_labels=1, hide_color=0, num_samples=1000)

        # Generate image mask for explanation
        temp, mask = explanation.get_image_and_mask(explanation.top_labels[0],\
            positive_only=False, num_features=8, hide_rest=False)

        # Show the image and explanation
        active_axis.imshow(mark_boundaries(temp, mask))
    fig.set_facecolor('#FFFFFF')
    fig.suptitle(title,fontsize=32);
    return(fig)
```

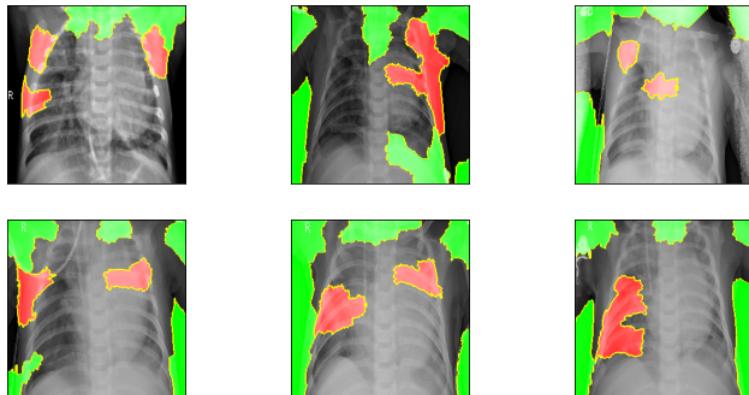
True Positives

X-rays that contain pneumonia that the model correctly identifies as containing pneumonia.

```
In [25]: # Extract true positives from test set and explain with lime
nt_tp = extract_selected(nt_model,1,1)
nt_tp_expl = lime_explain(nt_model,nt_tp,"True Positives")

0% | 0/1000 [00:00<?, ?it/s]
```

True Positives



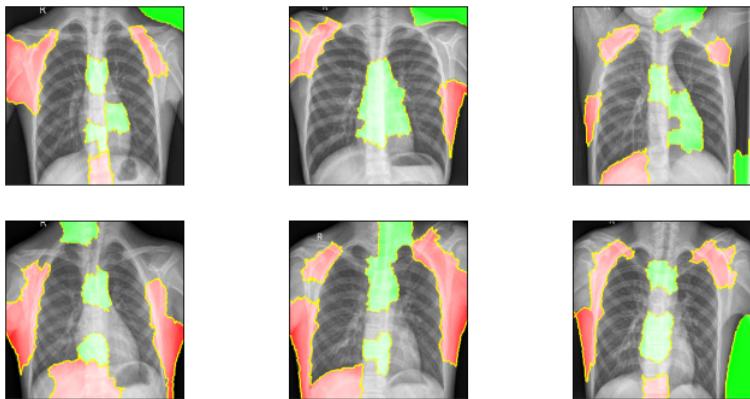
True Negatives

X-rays that are normal that the model correctly identifies as normal.

```
In [26]: # Extract true negatives from test set and explain with lime
nt_tn = extract_selected(nt_model,0,0)
nt_tn_expl = lime_explain(nt_model,nt_tn,"True Negatives")
```

0%	0/1000 [00:00<?, ?it/s]

True Negatives



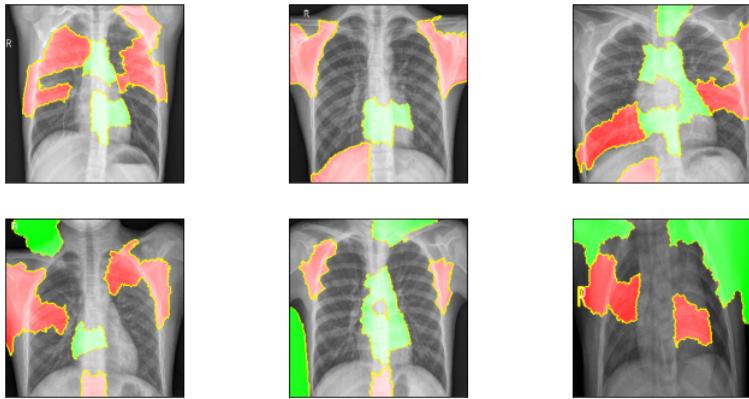
False Positives

X-rays that are normal that the model incorrectly identifies as having pneumonia.

```
In [27]: # Extract false positives from test set and explain with lime
nt_fp = extract_selected(nt_model,0,1)
nt_fp_expl = lime_explain(nt_model,nt_fp,"False Positives")
```

```
0%|          | 0/1000 [00:00<?, ?it/s]
```

False Positives



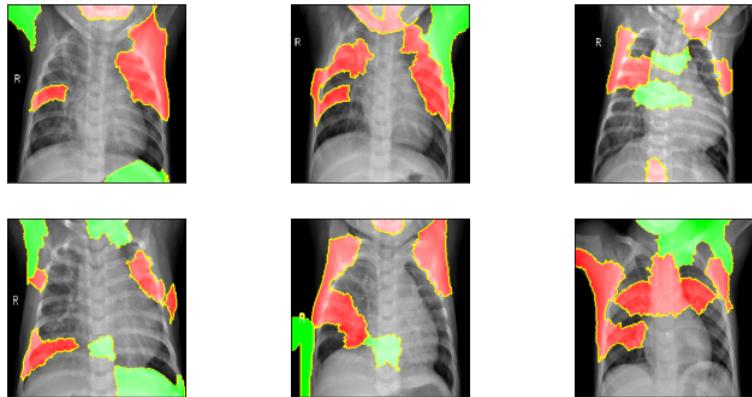
False Negatives

X-rays that have pneumonia that the model incorrectly identifies as normal.

```
In [28]: # Extract false negatives from test set and explain with lime
nt_fn = extract_selected(nt_model,1,0)
nt_fn_expl = lime_explain(nt_model,nt_fn,"False Negatives")

0%|          | 0/1000 [00:00<?, ?it/s]
```

False Negatives



Intermediate Activations

The following attempts to show what each of the intermediate layers of the model "sees".

```
In [29]: # Print overview of the model
nt_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 256, 256, 32)	128
conv2d_1 (Conv2D)	(None, 256, 256, 32)	1056
max_pooling2d (MaxPooling2D)	(None, 128, 128, 32)	0
)		
conv2d_2 (Conv2D)	(None, 128, 128, 64)	2112
conv2d_3 (Conv2D)	(None, 128, 128, 64)	4160
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 64)	0
)		
conv2d_4 (Conv2D)	(None, 64, 64, 128)	8320
conv2d_5 (Conv2D)	(None, 64, 64, 128)	16512
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 128)	0
)		
conv2d_6 (Conv2D)	(None, 32, 32, 256)	33024
conv2d_7 (Conv2D)	(None, 32, 32, 256)	65792
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 256)	0
)		
flatten (Flatten)	(None, 65536)	0
dense (Dense)	(None, 512)	33554944
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 512)	262656
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131328
dropout_2 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 128)	32896
dropout_3 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 64)	8256
dropout_4 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 32)	2080
dense_6 (Dense)	(None, 1)	33
<hr/>		
Total params:	34,123,297	
Trainable params:	34,123,297	
Non-trainable params:	0	

The following cell returns the activation for each node in the network when a single image is passed through it.

```
In [30]: # Collect outputs for the first 12 layers of model
nt_layer_outputs = [layer.output for layer in nt_model.layers[:12]]

activation_model = models.Model(
    inputs = nt_model.input,
    outputs = nt_layer_outputs
)

# Predict the first image in test dataset and save intermediate activations
nt_activations = activation_model.predict(np.expand_dims(test_ds[0][0][0],axis=0))
```

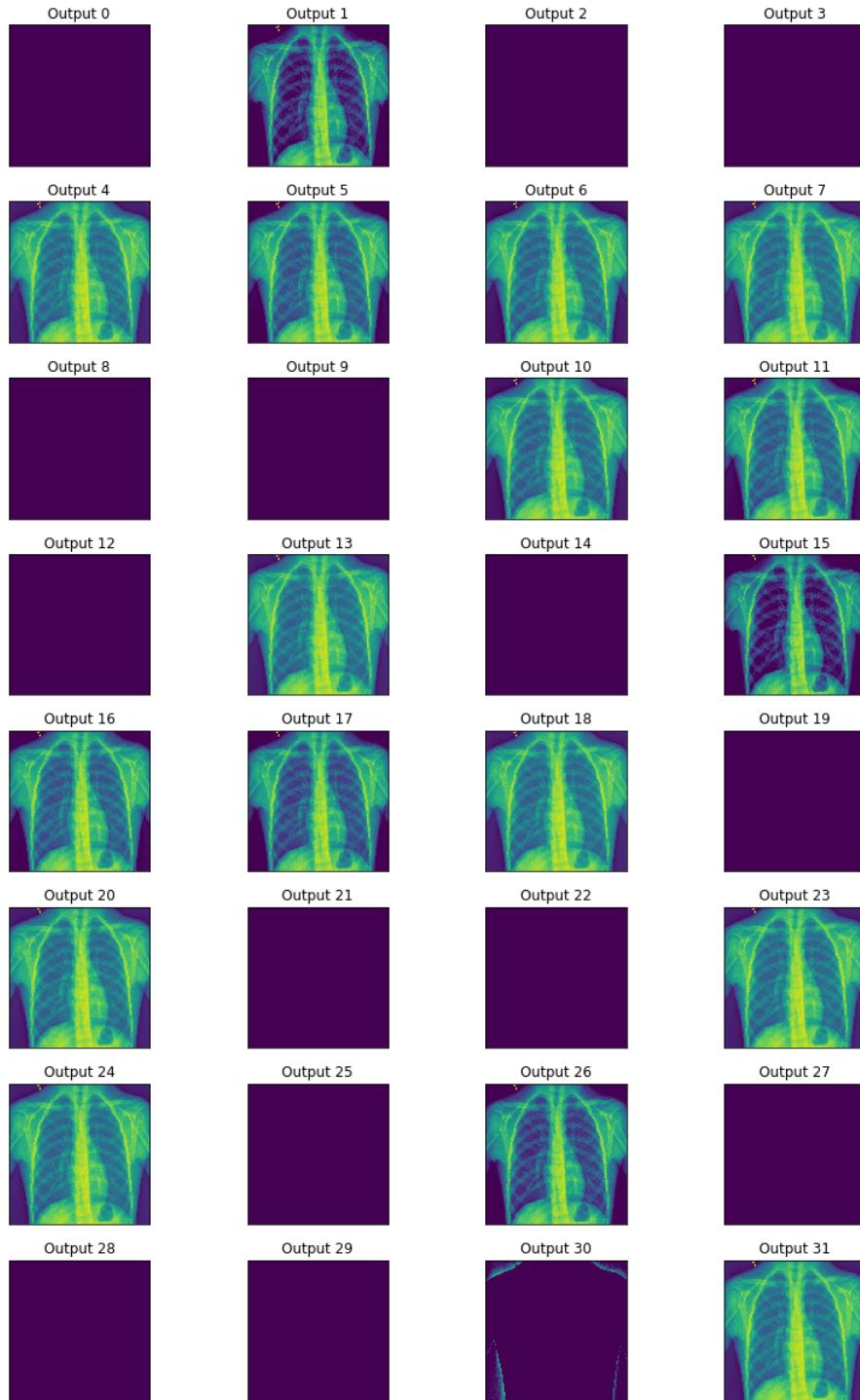
This figure shows the activation of each node in the first convolutional layer of the network.

```
In [31]: nt_l0_fig, nt_l0_ax = plt.subplots(8, 4, figsize=(12,18))

# Loop through each node in the first layer size 32:
for idx in range(32):
    active_axis = nt_l0_ax[idx//4, idx%4]

    # Extract and show node's activation
    first_layer_activation = nt_activations[0]
    active_axis.matshow(first_layer_activation[0, :, :, idx])
    active_axis.set_xticks([])
    active_axis.set_yticks([])
    active_axis.set_title('Output ' + str(idx))
nt_l0_fig.set_facecolor('#FFFFFF')
nt_l0_fig.suptitle('First Layer Activations\n', fontsize=32);
nt_l0_fig.tight_layout()
```

First Layer Activations



This figure shows the activation of the first node in each convolutional and pooling layer of the network.

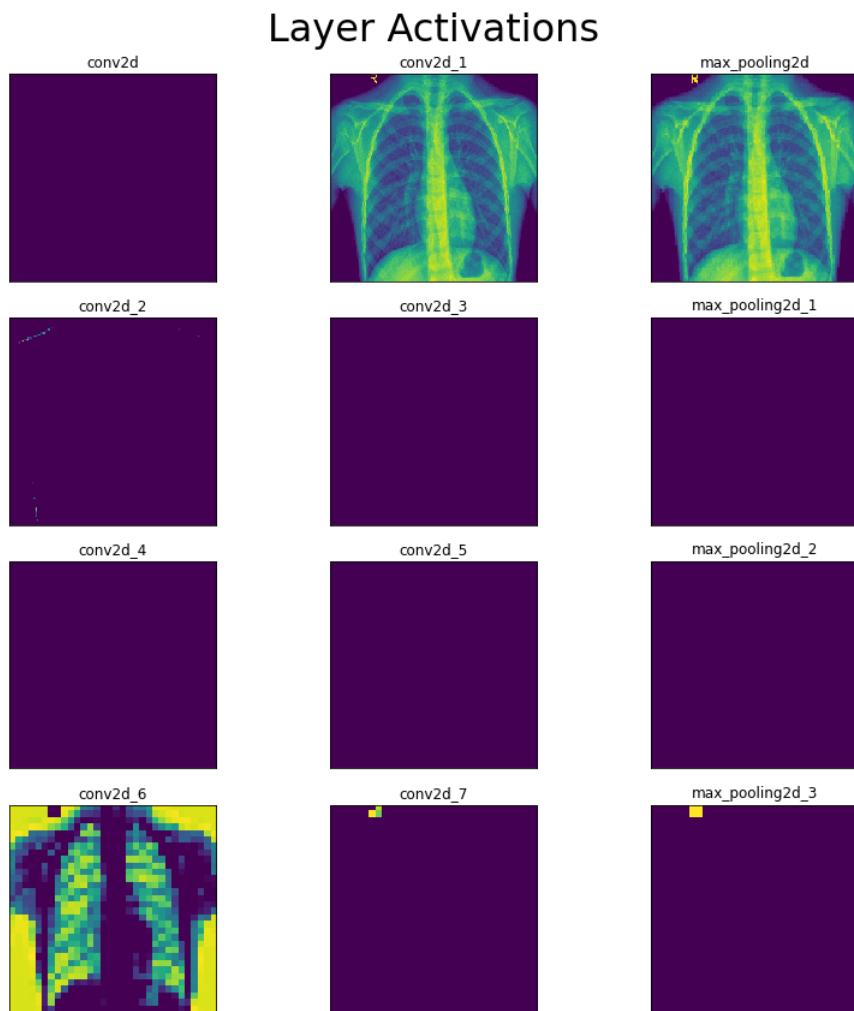
```
In [32]: nt_act_fig,nt_act_ax = plt.subplots(4,3,figsize=(12,12))

# Get name for each of the first 12 layers
nt_layer_names = []
for layer in nt_model.layers[:12]:
    nt_layer_names.append(layer.name)

# Loop through each of the first 12 layers
for idx in range(12):
    active_axis = nt_act_ax[idx//3][idx%3]

    # Show the activation for the first node in that layer
    cur_layer = nt_activations[idx]
    active_axis.imshow(cur_layer[0,:,:,:])
    active_axis.set_title(nt_layer_names[idx])
    active_axis.set_xticks([])
    active_axis.set_yticks([])

nt_act_fig.set_facecolor('#FFFFFF')
nt_act_fig.suptitle('Layer Activations',fontsize=32);
nt_act_fig.tight_layout()
```



Best Model - 11 Transfer Learning with VGG16 Decaying LR

The best performing model on validation data was 10 Transfer Learning with VGG16. The following code creates the model and sets the VGG16 layers to be not trainable.

```
In [33]: # Download and load VGG16 pre-trained model
vgg_base = applications.VGG16(
    weights='imagenet',
    input_shape=(256, 256, 3),
    include_top=False
)

# Don't allow vgg16 weights to be changed
vgg_base.trainable = False
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [=====] - 1s 0us/step
58900480/58889256 [=====] - 1s 0us/step

```
In [34]: tx_model = models.Sequential()

# Add vgg16 base to transfer model
tx_model.add(vgg_base)

# Densely connected block without dropout
tx_model.add(layers.Flatten())
tx_model.add(layers.Dense(512, activation='relu'))
tx_model.add(layers.Dense(256, activation='relu'))
tx_model.add(layers.Dense(128, activation='relu'))
tx_model.add(layers.Dense(64, activation='relu'))
tx_model.add(layers.Dense(1, activation='sigmoid'))

for layer in tx_model.layers:
    print(layer.name, layer.trainable)

vgg16 False
flatten_1 True
dense_7 True
dense_8 True
dense_9 True
dense_10 True
dense_11 True
```

Using checkpoints to save the epoch that does best on the validation data.

```
In [35]: # Checkpoint the model after each epoch
tx_checkpoint = callbacks.ModelCheckpoint(
    monitor='val_loss',
    filepath='save_data/tx_model.hdf5',
    save_best_only=True,
    verbose=0
)

tx_callbacks = [tx_checkpoint]

# Metrics are Accuracy, Precision, and Recall
use_metrics = [metrics.BinaryAccuracy(),\
               metrics.Precision(),\
               metrics.Recall()]
```

Adding a scheduler to decrease the learning rate for each epoch.

```
In [36]: # Make scheduler to add learning rate decay to the model
scheduler = optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.001,
    decay_rate=0.96,
    decay_steps=2500
)
```

```
In [37]: # Compile the model with the Adam optimizer and decaying learning rate
tx_model.compile(
    loss='binary_crossentropy',
    optimizer=optimizers.Adam(learning_rate=scheduler),
    metrics=[use_metrics]
)

# Print overview of the model
print(tx_model.summary())
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 8, 8, 512)	14714688
flatten_1 (Flatten)	(None, 32768)	0
dense_7 (Dense)	(None, 512)	16777728
dense_8 (Dense)	(None, 256)	131328
dense_9 (Dense)	(None, 128)	32896
dense_10 (Dense)	(None, 64)	8256
dense_11 (Dense)	(None, 1)	65

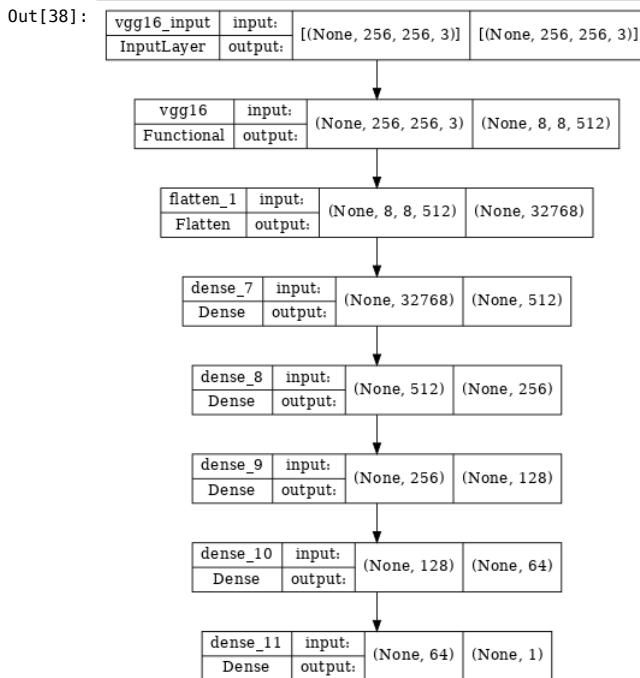
Total params: 31,664,961

Trainable params: 16,950,273

Non-trainable params: 14,714,688

None

```
In [38]: utils.plot_model(model=tx_model,to_file='images/tx_model.png',show_shapes=True,dpi=64)
```



It may be necessary to change this cell to get the notebook to run on your computer. If you do not have a working GPU change '/GPU:0' to '/CPU:0'. This cell fits the model and saves the results to history. It then loads the best epoch from the checkpoint.

```
In [39]: # Fit the model and save results to history
with tf.device('/GPU:0'):
    tx_history = tx_model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=10,
        callbacks=tx_callbacks
    )

# Load the best model based on validation loss
tx_model = models.load_model('save_data/tx_model.hdf5')

Epoch 1/10
131/131 [=====] - 64s 333ms/step - loss: 0.3013 - binary_accuracy: 0.8716 - precision_1: 0.9117 - recall_1: 0.9158 - val_loss: 0.1634 - val_binary_accuracy: 0.9357 - val_precision_1: 0.9735 - val_recall_1: 0.9387
Epoch 2/10
131/131 [=====] - 41s 315ms/step - loss: 0.1482 - binary_accuracy: 0.9447 - precision_1: 0.9635 - recall_1: 0.9619 - val_loss: 0.3779 - val_binary_accuracy: 0.8278 - val_precision_1: 1.0000 - val_recall_1: 0.7676
Epoch 3/10
131/131 [=====] - 41s 316ms/step - loss: 0.1431 - binary_accuracy: 0.9471 - precision_1: 0.9657 - recall_1: 0.9629 - val_loss: 0.3333 - val_binary_accuracy: 0.8600 - val_precision_1: 1.0000 - val_recall_1: 0.8110
Epoch 4/10
131/131 [=====] - 42s 320ms/step - loss: 0.1385 - binary_accuracy: 0.9468 - precision_1: 0.9663 - recall_1: 0.9619 - val_loss: 0.2641 - val_binary_accuracy: 0.8732 - val_precision_1: 1.0000 - val_recall_1: 0.8289
Epoch 5/10
131/131 [=====] - 41s 316ms/step - loss: 0.1354 - binary_accuracy: 0.9521 - precision_1: 0.9717 - recall_1: 0.9635 - val_loss: 0.2414 - val_binary_accuracy: 0.9007 - val_precision_1: 0.9985 - val_recall_1: 0.8672
Epoch 6/10
131/131 [=====] - 42s 322ms/step - loss: 0.1221 - binary_accuracy: 0.9499 - precision_1: 0.9689 - recall_1: 0.9635 - val_loss: 0.3011 - val_binary_accuracy: 0.8817 - val_precision_1: 0.9985 - val_recall_1: 0.8416
Epoch 7/10
131/131 [=====] - 42s 317ms/step - loss: 0.1142 - binary_accuracy: 0.9571 - precision_1: 0.9743 - recall_1: 0.9677 - val_loss: 0.1806 - val_binary_accuracy: 0.9243 - val_precision_1: 0.9986 - val_recall_1: 0.8991
Epoch 8/10
131/131 [=====] - 42s 324ms/step - loss: 0.0942 - binary_accuracy: 0.9648 - precision_1: 0.9774 - recall_1: 0.9752 - val_loss: 0.3565 - val_binary_accuracy: 0.8647 - val_precision_1: 1.0000 - val_recall_1: 0.8174
Epoch 9/10
131/131 [=====] - 42s 320ms/step - loss: 0.0990 - binary_accuracy: 0.9646 - precision_1: 0.9780 - recall_1: 0.9742 - val_loss: 0.1214 - val_binary_accuracy: 0.9451 - val_precision_1: 0.9973 - val_recall_1: 0.9285
Epoch 10/10
131/131 [=====] - 42s 323ms/step - loss: 0.1009 - binary_accuracy: 0.9610 - precision_1: 0.9757 - recall_1: 0.9716 - val_loss: 0.2561 - val_binary_accuracy: 0.8940 - val_precision_1: 0.9985 - val_recall_1: 0.8582
```

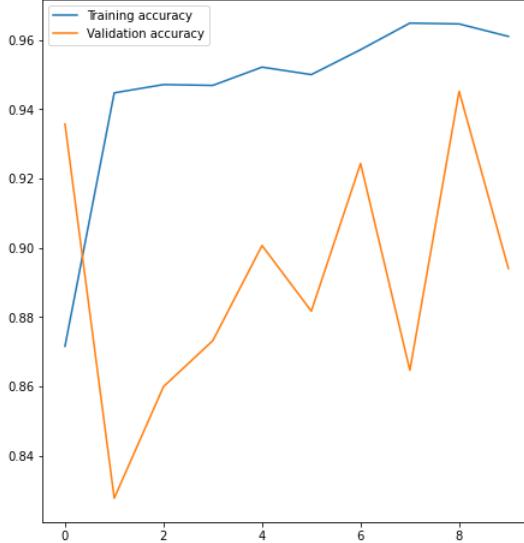
The following cell shows the training and validation accuracy and loss for each epoch. Accuracy should trend upwards and loss should trend downwards with more epochs. Validation accuracy will generally be lower than training accuracy, and validation loss will generally be higher than training loss.

```
In [40]: # Grab model results from history dict for plotting
tx_acc = tx_history.history['binary_accuracy']
tx_val_acc = tx_history.history['val_binary_accuracy']
tx_loss = tx_history.history['loss']
tx_val_loss = tx_history.history['val_loss']
tx_epochs = range(len(tx_acc))

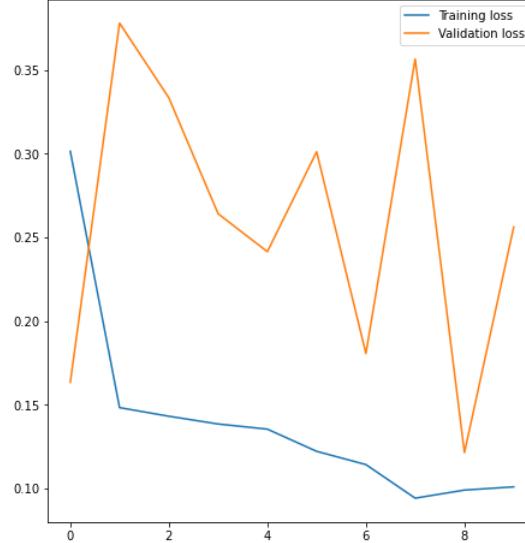
# Plot train and val accuracy for each epoch
tx_fig, tx_ax = plt.subplots(1,2,figsize=(16,8))
tx_ax[0].plot(tx_epochs, tx_acc, label='Training accuracy');
tx_ax[0].plot(tx_epochs, tx_val_acc, label='Validation accuracy');
tx_ax[0].set_title('Training and validation accuracy',fontsize=24);
tx_ax[0].legend();

# Plot train and val loss for each epoch
tx_ax[1].plot(tx_epochs, tx_loss, label='Training loss');
tx_ax[1].plot(tx_epochs, tx_val_loss, label='Validation loss');
tx_ax[1].set_title('Training and validation loss',fontsize=24);
tx_ax[1].legend();
```

Training and validation accuracy



Training and validation loss



Model Results

```
In [41]: # Evaluate the model on training set
tx_train_res = tx_model.evaluate(train_ds)

131/131 [=====] - 38s 290ms/step - loss: 0.0984 - binary_accuracy: 0.9646 - precision_1: 0.9642 - recall_1: 0.9890

In [42]: # Evaluate the model on validation set
tx_val_res = tx_model.evaluate(val_ds)

34/34 [=====] - 4s 100ms/step - loss: 0.1214 - binary_accuracy: 0.9451 - precision_1: 0.9973 - recall_1: 0.9285

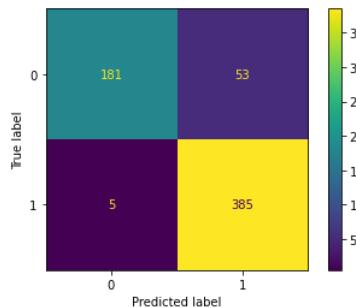
In [43]: # Evaluate the model on test set
tx_test_res = tx_model.evaluate(test_ds)

20/20 [=====] - 2s 104ms/step - loss: 0.2802 - binary_accuracy: 0.9071 - precision_1: 0.8790 - recall_1: 0.9872
```

```
In [44]: # Get predictions for test set and actual values
tx_test_predictions = np.round(tx_model.predict(test_ds),0)
test_actual = test_ds.labels

# Generate confusion matrix for test set
tx_c_matrix = confusion_matrix(test_actual,tx_test_predictions)
ConfusionMatrixDisplay(tx_c_matrix).plot()
```

```
Out[44]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f5a2deece80>
```



Lime (<https://github.com/marcotcr/lime>) Explanation of Results

```
In [45]: explainer = lime_image.LimeImageExplainer()
```

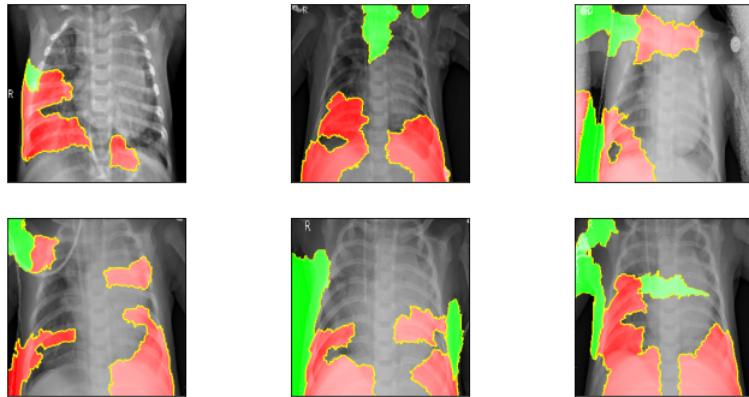
True Positives

X-rays that contain pneumonia that the model correctly identifies as containing pneumonia.

```
In [46]: # Extract true positives from test set and explain with lime
tx_tp = extract_selected(tx_model,1,1)
tx_tp_expl = lime_explain(tx_model,tx_tp,"True Positives")
```

```
0%|          | 0/1000 [00:00<?, ?it/s]
```

True Positives



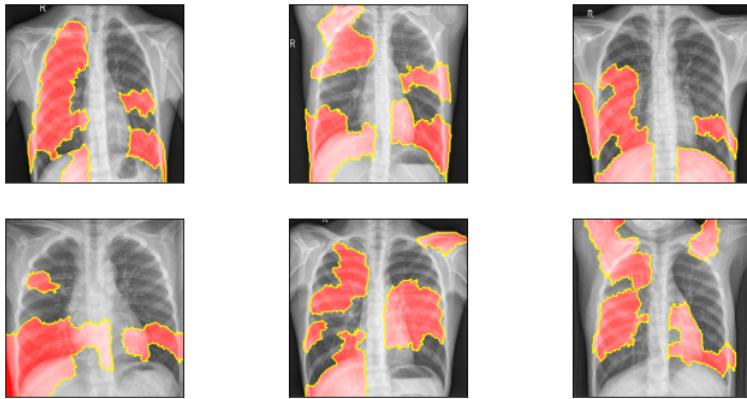
True Negatives

X-rays that are normal that the model correctly identifies as normal.

```
In [47]: # Extract true negatives from test set and explain with lime
tx_tn = extract_selected(tx_model,0,0)
tx_tn_expl = lime_explain(tx_model,tx_tn,"True Negatives")

0%|          | 0/1000 [00:00<?, ?it/s]
```

True Negatives



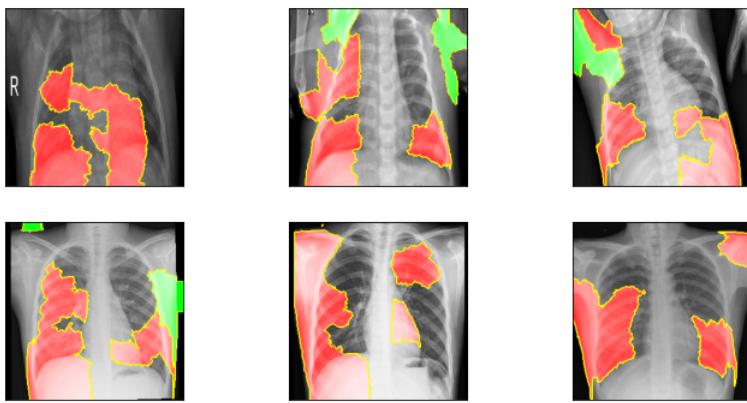
False Positives

X-rays that are normal that the model incorrectly identifies as having pneumonia.

```
In [48]: # Extract false positives from test set and explain with lime
tx_fp = extract_selected(tx_model,0,1)
tx_fp_expl = lime_explain(tx_model,tx_fp,"False Positives")

0%|          | 0/1000 [00:00<?, ?it/s]
```

False Positives



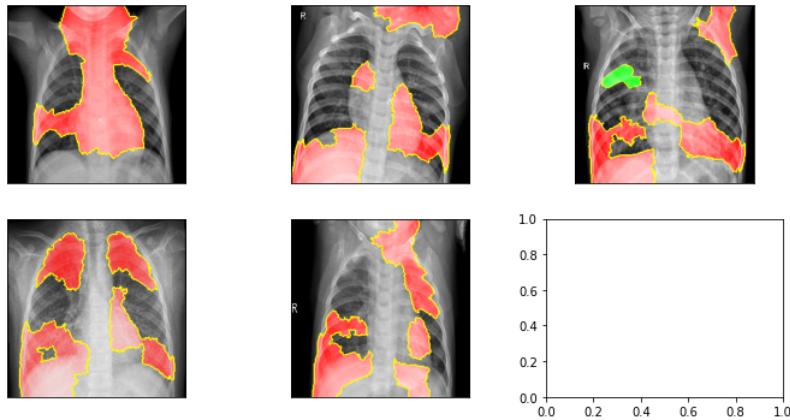
False Negatives

X-rays that have pneumonia that the model incorrectly identifies as normal.

```
In [49]: # Extract false negatives from test set and explain with lime
tx_fn = extract_selected(tx_model,1,0)
tx_fn_expl = lime_explain(tx_model,tx_fn,"False Negatives")
```

```
0%|      | 0/1000 [00:00<?, ?it/s]
```

False Negatives



Intermediate Activations

The following attempts to show what each of the intermediate layers of the model "sees".

```
In [50]: # Print overview of the model
tx_model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
<hr/>		
vgg16 (Functional)	(None, 8, 8, 512)	14714688
flatten_1 (Flatten)	(None, 32768)	0
dense_7 (Dense)	(None, 512)	16777728
dense_8 (Dense)	(None, 256)	131328
dense_9 (Dense)	(None, 128)	32896
dense_10 (Dense)	(None, 64)	8256
dense_11 (Dense)	(None, 1)	65
<hr/>		
Total params: 31,664,961		
Trainable params: 16,950,273		
Non-trainable params: 14,714,688		

```
In [51]: # Print overview of the model's vgg16 layers
tx_model.layers[0].summary()
```

Model: "vgg16"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[None, 256, 256, 3]	0
block1_conv1 (Conv2D)	(None, 256, 256, 64)	1792
block1_conv2 (Conv2D)	(None, 256, 256, 64)	36928
block1_pool (MaxPooling2D)	(None, 128, 128, 64)	0
block2_conv1 (Conv2D)	(None, 128, 128, 128)	73856
block2_conv2 (Conv2D)	(None, 128, 128, 128)	147584
block2_pool (MaxPooling2D)	(None, 64, 64, 128)	0
block3_conv1 (Conv2D)	(None, 64, 64, 256)	295168
block3_conv2 (Conv2D)	(None, 64, 64, 256)	590080
block3_conv3 (Conv2D)	(None, 64, 64, 256)	590080
block3_pool (MaxPooling2D)	(None, 32, 32, 256)	0
block4_conv1 (Conv2D)	(None, 32, 32, 512)	1180160
block4_conv2 (Conv2D)	(None, 32, 32, 512)	2359808
block4_conv3 (Conv2D)	(None, 32, 32, 512)	2359808
block4_pool (MaxPooling2D)	(None, 16, 16, 512)	0
block5_conv1 (Conv2D)	(None, 16, 16, 512)	2359808
block5_conv2 (Conv2D)	(None, 16, 16, 512)	2359808
block5_conv3 (Conv2D)	(None, 16, 16, 512)	2359808
block5_pool (MaxPooling2D)	(None, 8, 8, 512)	0
<hr/>		
Total params: 14,714,688		
Trainable params: 0		
Non-trainable params: 14,714,688		

The following cell returns the activation for each node in the network when a single image is passed through it.

```
In [52]: # Collect outputs for the vgg16 layers
tx_layer_outputs = [layer.output for layer in tx_model.layers[0].layers]

tx_activation_model = models.Model(
    inputs = tx_model.layers[0].input,
    outputs = tx_layer_outputs
)

# Predict the first image in test dataset and save intermediate activations
tx_activations = tx_activation_model.predict(np.expand_dims(test_ds[0][0], axis=0))
```

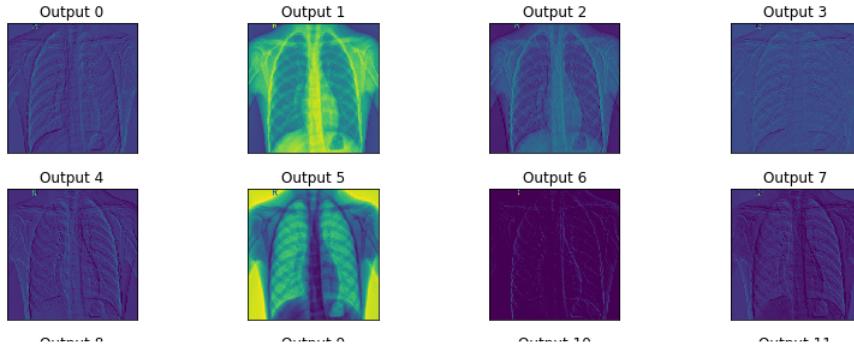
This figure shows the activation of each node in the first convolutional layer of the network.

```
In [53]: tx_l0_fig, tx_l0_ax = plt.subplots(16, 4, figsize=(12,32))

# Loop through each node in the first layer size 64
for idx in range(64):
    active_axis = tx_l0_ax[idx//4, idx%4]

    # Extract and show node's activation
    first_layer_activation = tx_activations[1]
    active_axis.matshow(first_layer_activation[0 , :, :, idx])
    active_axis.set_xticks([])
    active_axis.set_yticks([])
    active_axis.set_title('Output ' + str(idx))
tx_l0_fig.set_facecolor('#FFFFFF')
tx_l0_fig.suptitle('First Layer Activations\n', fontsize=32);
tx_l0_fig.tight_layout()
```

First Layer Activations



This figure shows the activation of the first node in each convolutional and pooling layer of the network.

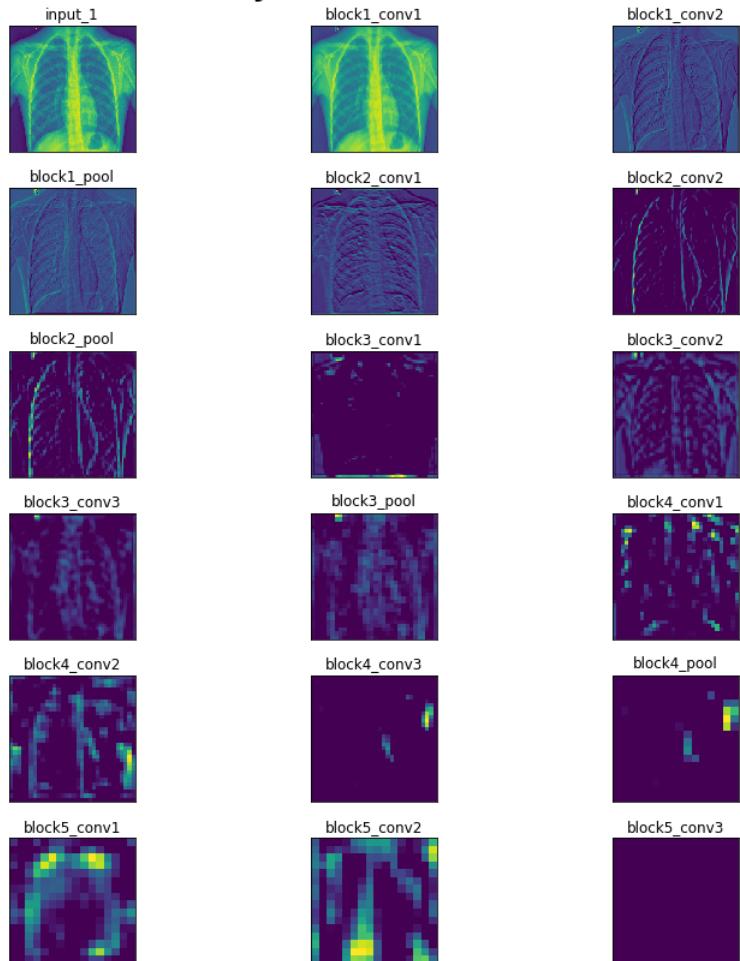
```
In [54]: tx_act_fig,tx_act_ax = plt.subplots(6,3,figsize=(12,12))

# Get name for each of the vgg16 layers
tx_layer_names = []
for layer in tx_model.layers[0].layers:
    tx_layer_names.append(layer.name)

# Loop through each of the vgg16 layers
for idx in range(18):
    active_axis = tx_act_ax[idx//3][idx%3]

    # Show the activation for the first node in that layer
    cur_layer = tx_activations[idx]
    active_axis.matshow(cur_layer[0,:,:,:1])
    active_axis.set_title(tx_layer_names[idx])
    active_axis.set_xticks([])
    active_axis.set_yticks([])
tx_act_fig.set_facecolor('#FFFFFF')
tx_act_fig.suptitle('Layer Activations',fontsize=32);
tx_act_fig.tight_layout()
```

Layer Activations



Conclusion

The final model can be used by the Bentley Health Foundation to determine whether any given chest x-ray shows signs of pneumonia. This model can be deployed in regions where pneumonia is common to assist with diagnosing cases. This will allow patients to receive the life-saving treatment they need without having to risk waiting for or traveling to a specialist.