

CNNs for Identifying Pneumonia in Chest X-ray Images

By Keshav Ganesh

Introduction

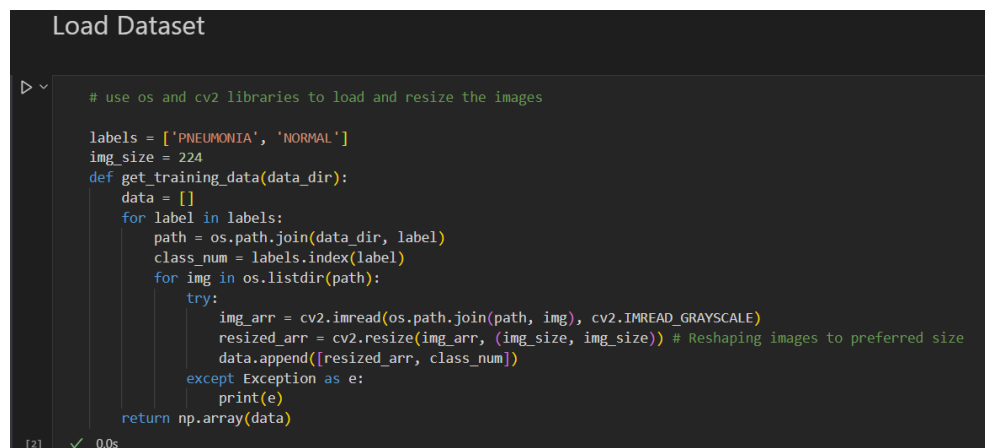
In the realm of medical imaging, machine learning and advanced image analysis tools have led to significant advances. This report aims to use convolutional neural networks (CNN) to improve the accuracy of detecting pneumonia using chest X-ray images. Here, we consider two approaches. Firstly, we implement pre-trained models, which have been trained extensively on large datasets. Alternatively, we use a custom-designed CNN model with fine-tuned hyperparameters. Finally, we compare the performances of these two models and discuss the implications.

The Dataset

Before we delve into any CNN models, we will first talk about the structure of our data. Our dataset is a [Kaggle](#) dataset. The dataset is organized into three folders (train, test, validation) and contains subfolders for each category (Pneumonia/Normal). There are 5,863 X-Ray images.

Loading the dataset

The first step is loading the dataset from the appropriate directories. The images in the dataset possessed different resolutions. We resized the images to 224 x 224 resolution. Convolutional neural networks only function when the images used to train them share the same dimensions.



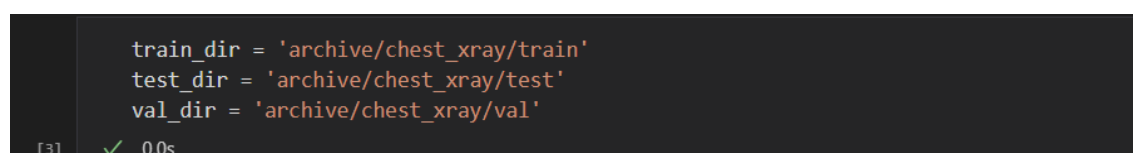
```
Load Dataset

# use os and cv2 libraries to load and resize the images

labels = ['PNEUMONIA', 'NORMAL']
img_size = 224
def get_training_data(data_dir):
    data = []
    for label in labels:
        path = os.path.join(data_dir, label)
        class_num = labels.index(label)
        for img in os.listdir(path):
            try:
                img_arr = cv2.imread(os.path.join(path, img), cv2.IMREAD_GRAYSCALE)
                resized_arr = cv2.resize(img_arr, (img_size, img_size)) # Reshaping images to preferred size
                data.append([resized_arr, class_num])
            except Exception as e:
                print(e)
    return np.array(data)
```

[2] ✓ 0.0s

Figure 1: Function to load dataset using file paths



```
train_dir = 'archive/chest_xray/train'
test_dir = 'archive/chest_xray/test'
val_dir = 'archive/chest_xray/val'
```

[3] ✓ 0.0s

Figure 2: Calling the function from Figure 1

Each item in the dataset contains the image (in 2d array representation) and its corresponding label.

```
[5] train.shape, test.shape, val.shape
... ((5216, 2), (624, 2), (16, 2))
```

Figure 3: Shape of the datasets

Data Visualization

```
[6] plt.figure(figsize = (5,5))
    plt.imshow(train[0][0], cmap='gray')
    plt.title(labels[train[0][1]])

    plt.figure(figsize = (5,5))
    plt.imshow(train[-1][0], cmap='gray')
    plt.title(labels[train[-1][1]])
```

Figure 4: Display Images

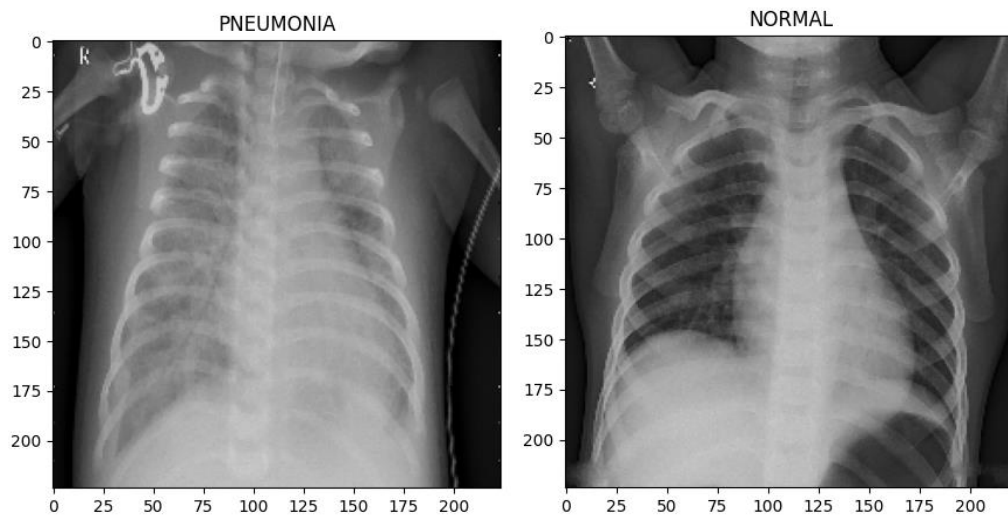


Figure 5: Sample of Pneumonia and Normal Chest X-rays

Pneumonia is an inflammatory lung condition affecting primarily the small air sacs known as alveoli. When interpreting the X-ray, the radiologist looks for white spots in the lungs (called infiltrates) that identify an infection. Fig 6 provides a diagrammatical representation of the inflammation, resulting in the opaque white spots in the chest X-ray in Fig 5.

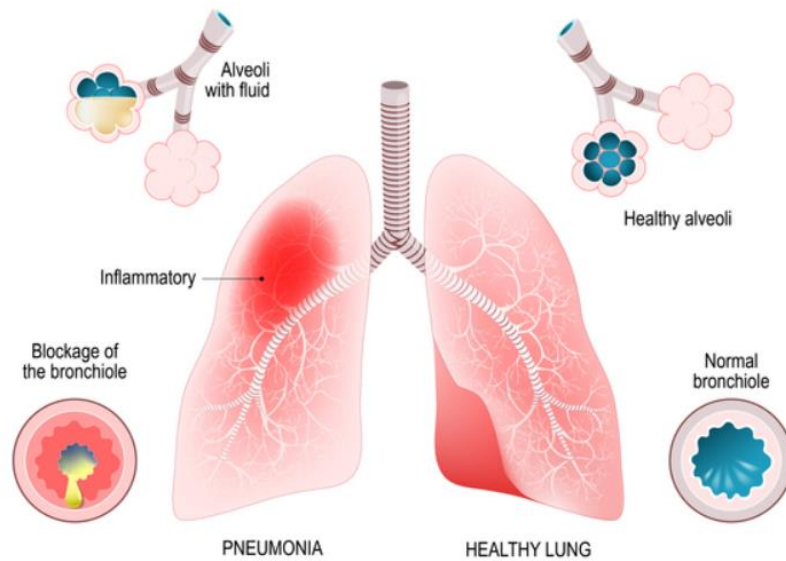


Figure 6: Normal Lung and Lung with Pneumonia

Data Preprocessing

Using numpy, we create the feature and label arrays. Then, we normalize the values in the arrays and resize the arrays to prepare them for model learning.

Split into features and labels

```
x_train = []
y_train = []

x_val = []
y_val = []

x_test = []
y_test = []

for feature, label in train:
    x_train.append(feature)
    y_train.append(label)

for feature, label in test:
    x_test.append(feature)
    y_test.append(label)

for feature, label in val:
    x_val.append(feature)
    y_val.append(label)
```

[11]

Figure 7: Split items into features and labels

Normalization

```
#Normalization changes the values in the columns to a common scale.  
#Pixel normalization technique is often used to speed up model learning.  
#Dividing by 255 normalizes RGB values to be between 0 and 1.
```

```
x_train = np.array(x_train) / 255  
x_val = np.array(x_val) / 255  
x_test = np.array(x_test) / 255
```

[10] ✓ 8.3s

Resize data for learning

```
#Numpy arrays are reshaped to prepare them to be fed into the model.
```

```
x_train = x_train.reshape(-1, img_size, img_size, 1)  
y_train = np.array(y_train)
```

```
x_val = x_val.reshape(-1, img_size, img_size, 1)  
y_val = np.array(y_val)
```

```
x_test = x_test.reshape(-1, img_size, img_size, 1)  
y_test = np.array(y_test)
```

```
x_train.shape, x_val.shape, x_test.shape
```

[11]

... ((5216, 224, 224, 1), (16, 224, 224, 1), (624, 224, 224, 1))

Figure 8: Normalization and Restructuring Data

However, there is an additional issue to consider. Pre-trained models use RGB images. We duplicate the entries along the last axis to obtain the correct model input shape.

```
x_train = np.repeat(x_train,3,-1)  
x_test = np.repeat(x_test,3,-1)  
x_val = np.repeat(x_val,3,-1)
```

```
x_train.shape, x_val.shape, x_test.shape
```

[12] ✓ 35.1s

... ((5216, 224, 224, 3), (16, 224, 224, 3), (624, 224, 224, 3))

Figure 9: Convert Grayscale to RGB

Data Augmentation and Class Imbalance

```
# Visualise class distribution

l = []
for i in train:
    if(i[1] == 0):
        l.append("Pneumonia")
    else:
        l.append("Normal")

l = pd.DataFrame(l)
sns.countplot(x = 0, data = l)
```

[10]

Figure 10: Code to visualise Class Distribution

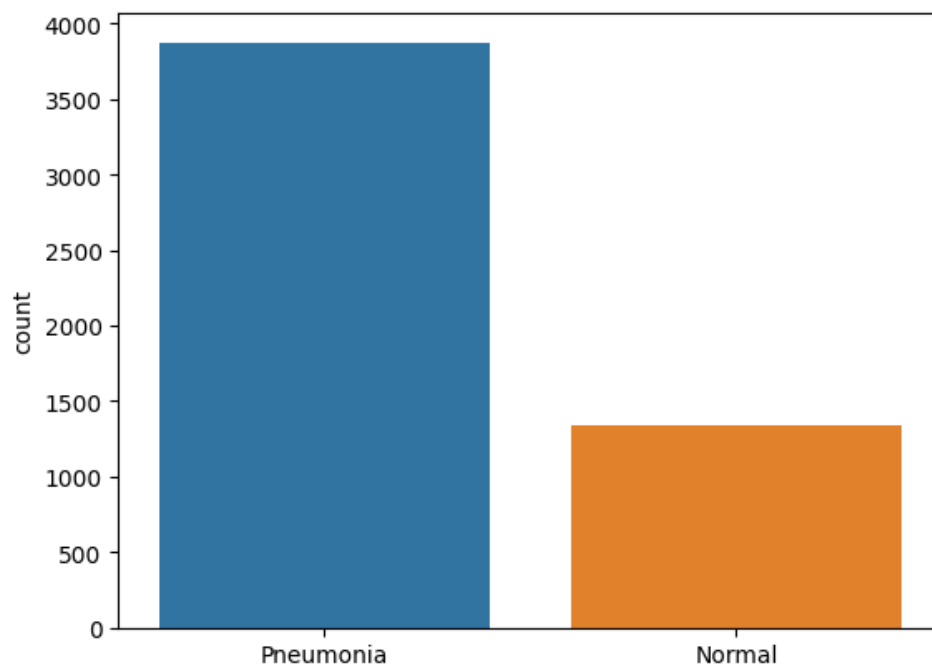


Figure 11: Class Distribution Graph

To ensure effective model training, we need to tackle the issue of class imbalance.

We have significantly more Pneumonia scans than Normal scans, leading to biased model performance, as the model may become more inclined to predict the majority class, ignoring the minority class. The result is poor predictive accuracy and underrepresentation of the minority class.

Furthermore, there is an additional problem due to the size of the dataset. A small training set can lead to low model accuracy and overfitting, leading to poor performance on unseen data.

To deal with these issues, we will implement a two-pronged strategy.

Firstly, we make use of class weights. Class weighting adjusts the cost function of the model so that misclassifying an observation from the minority class is more heavily penalized than misclassifying from the majority class.

Secondly, we implement data augmentation. Data augmentation artificially increases the training set size by creating modified copies of existing data, which handles our issue of limited dataset size.

```
# create class weight dictionary

# Calculate class weights
total_samples = len(y_train)
class_samples_A = y_train.count(0)
class_samples_B = y_train.count(1)
class_weight_A = total_samples / (2 * class_samples_A)
class_weight_B = total_samples / (2 * class_samples_B)

class_weights_dict = {0: class_weight_A, 1: class_weight_B}

print(class_weights_dict)

[ ]
... {0: 0.6730322580645162, 1: 1.9448173005219984}
```

Figure 12: Class Weighting (Class 0 is Pneumonia and Class 1 is Normal)

```
datagen = keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False, # set input mean to 0 over the dataset
    samplewise_center=False, # set each sample mean to 0
    featurewise_std_normalization=False, # divide inputs by std of the dataset
    samplewise_std_normalization=False, # divide each input by its std
    zca_whitening=False, # apply ZCA whitening
    rotation_range = 30, # randomly rotate images in the range (degrees, 0 to 180)
    zoom_range = 0.2, # Randomly zoom image
    width_shift_range=0.1, # randomly shift images horizontally (fraction of total width)
    height_shift_range=0.1, # randomly shift images vertically (fraction of total height)
    horizontal_flip = True, # randomly flip images
    vertical_flip=False
)

datagen.fit(x_train)

[12] ✓ 11.1s
```

Figure 13: Data Augmentation (Only some transformations have been implemented)

```
for x_batch, y_batch in datagen.flow(x_train,y_train, batch_size=6):  
    for i in range(0, 6):  
        plt.subplot(2,3,i+1)  
        plt.imshow(x_batch[i], cmap='gray')  
        plt.axis('off')  
    break
```

[13] ✓ 4.8s

...

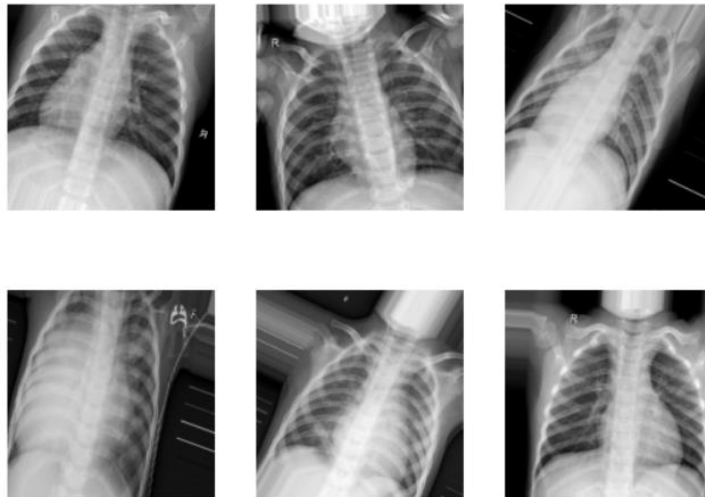


Figure 14: Shows the effect of data augmentation, we have more data to train on

Building the CNN models

A brief description of CNNs:

A CNN, like any other neural network, contains an input layer, an output layer and many hidden layers. These layers carry out operations that change the data to uncover particular properties. Convolution, activation or ReLU, and pooling are three of the most used layers.

- **Convolution:** The input images are subjected to a series of convolutional filters during convolution, each of which activates different aspects of the input images.
- **Rectified linear unit:** Maintains positive values while translating negative values to zero, enabling quicker and more efficient training. The nonlinearity of the image's pixel data is enhanced by the ReLU layer.
- **Pooling:** Reduces the number of parameters the network needs to learn, which simplifies the output.

Each layer learns to recognize various traits as these procedures are repeated across tens or hundreds of levels. Once feature learning is completed (over several layers), the architecture shifts to classification.

The second-last layer is a fully connected layer, that outputs a node with the sigmoid activation function. This is used instead of Softmax because we have two classes (Softmax is used for three or more classes).

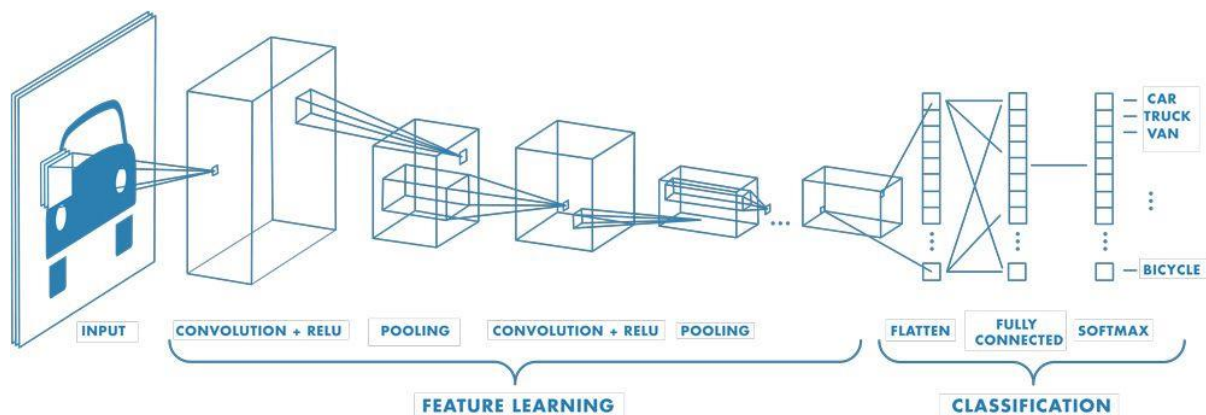


Figure 15: Example of a CNN for Image Classification

Pre-trained Models

A pre-trained model is a model that was created by someone else and was used to solve a similar classification problem. We can use a pre-trained model as a foundation, instead of building a model from scratch. This is called transfer learning, which allows us to customize the model to cater to our task.

We will use the pre-trained models on the *TensorFlow Hub* and *TensorFlow.keras.applications* module. The models we will use are the following:

- [EfficientNetB0](#)
- [DenseNet121](#)
- [CheXNet](#) (Uses Dense121 with CheXNet weights)
- [ResNet](#)
- [MobileNet](#)

We will customize each model by adding Dense layers, Average Pooling layers and making the last few layers of the network trainable. During the fitting process, the optimizer algorithm will learn the optimal weights of these layers. The following code snippets demonstrate the functions used to implement the pre-trained model.

```
Efficient Net

# use pretrained model efficient net link

pretrained_model = "https://tfhub.dev/tensorflow/efficientnet/b0/classification/1"

model = tf.keras.Sequential([
    hub.KerasLayer(pretrained_model, trainable=False, input_shape=(img_size, img_size, 3)),
    tf.keras.layers.Dense(512, activation = 'relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.summary()
```

[18]

Figure 16: Defining the Model

```
... Model: "sequential"

Layer (type)                 Output Shape              Param #
=====
keras_layer (KerasLayer)     (None, 1000)              5330564

dense (Dense)                (None, 512)               512512

dense_1 (Dense)              (None, 1)                 513
=====
Total params: 5,843,589
Trainable params: 513,025
Non-trainable params: 5,330,564
```

Figure 17: Summary of the Model

Next, we compile it and begin training using the class weights and the augmented dataset. Furthermore, we use the function *ReduceLROnPlateau* (Fig 16). It decreases the learning rate when the specified metric stops improving longer than the patience number allows. The learning rate is unchanged until the results run into stagnation.

```

from tensorflow.keras.callbacks import ReduceLROnPlateau

learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience = 2, verbose=1, factor=0.3, min_lr=0.000001)

```

[20]

Figure 18: ReduceLROnPlateau

```

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

history = model.fit(datagen.flow(x_train,y_train,batch_size=16), epochs = 12,
                    validation_data=datagen.flow(x_val,y_val),
                    class_weight=class_weights_dict, callbacks=[learning_rate_reduction])

```

[19]

[20]

Figure 19: Compiling and Fitting

The final step is evaluating the model on the test dataset. We used the confusion matrix and classification report methods from *sklearn.metrics* to interpret the quality of our results. Our final model will be the model which achieves the highest accuracy on the test set. Figures 21 and 22 show the evaluation of the EfficientNetB0 model.

```

def plot_confusion_matrix(confusion_matrix, classes=["Pneumonia", "Normal"]):
    plt.figure(figsize=(8, 6))
    im = plt.imshow(confusion_matrix, interpolation='nearest', cmap='Blues')
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted Labels")
    plt.ylabel("True Labels")

    # Add color bar
    color_bar = plt.colorbar(im)
    color_bar.set_label('Score')

    # Set tick labels
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    # Rotate tick labels if needed
    plt.xticks(rotation=45, ha="right")

    # Add text annotations
    thresh = confusion_matrix.max() / 2
    for i in range(confusion_matrix.shape[0]):
        for j in range(confusion_matrix.shape[1]):
            plt.text(j, i, format(confusion_matrix[i, j], 'd'),
                    ha="center", va="center",
                    color="white" if confusion_matrix[i, j] > thresh else "black")

    plt.tight_layout()
    plt.show()

```

[14]

Figure 20: Function to plot confusion matrix

```

from sklearn.metrics import confusion_matrix, classification_report

y_pred = model.predict(x_test)
y_pred = np.round(y_pred).astype(int)

loss, acc = model.evaluate(x_test, y_test)
print("Accuracy: ", acc*100, "%")
print("Loss: ", loss)

cm = confusion_matrix(y_test, y_pred)
plot_confusion_matrix(cm)

print(classification_report(y_test, y_pred, target_names = ['Pneumonia (Class 0)', 'Normal (Class 1)']))

```

[22]

```

... 20/20 [=====] - 27s 1s/step
20/20 [=====] - 24s 1s/step - loss: 0.4401 - accuracy: 0.8141
Accuracy: 81.41025900840759 %
Loss: 0.44007864594459534

```

Figure 21: Model evaluation

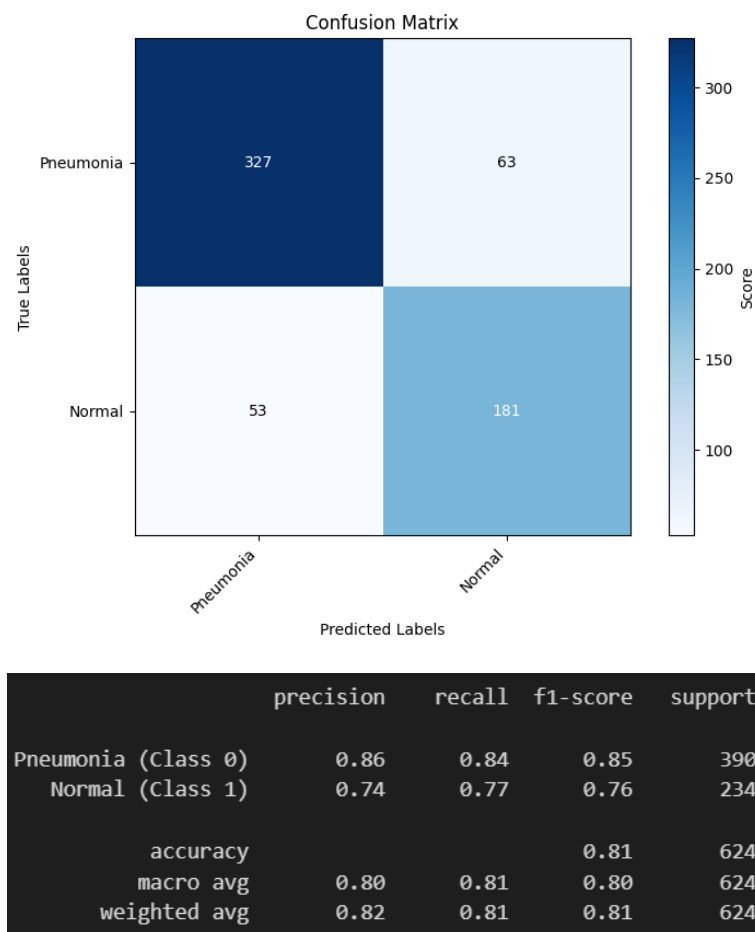


Figure 22: Confusion Matrix and Classification Report

EfficientNetB0 scored an accuracy of 81.41%. One can notice a significant bias against the minority class due to the differing F1 scores (0.85 for Pneumonia and 0.76 for Normal). We will implement the other pre-trained models and summarise their accuracies with a table. The code snippets present the structure of the pre-trained models.

Model	Trainable Params	Non-trainable Params	Total Params
EfficientNetB0	513,025	5,330,564	5,843,589
DenseNet121	167,169	6,871,360	7,038,529
CheXNet	180,509	6,871,360	7,051,869
MobileNet	1,052,673	3,226,816	4,279,489
ResNet152v2	2,103,297	58,327,552	60,430,849

Figure 23: Table which shows details of model parameters

```

from keras.applications import densenet
from keras.models import Model
from keras.layers import Dense

densenet = densenet.DenseNet121(weights=None, include_top=False,
                                input_shape=(224, 224, 3), pooling="avg")

x = densenet.output
predictions = Dense(1, activation="sigmoid", name="predictions")(x)

model = Model(inputs=densenet.input, outputs=predictions)

for layer in model.layers:
    layer.trainable = True

for layer in model.layers[:-10]:
    layer.trainable = False

model.summary()

```

[19]

Figure 24: DenseNet121 model

```

from keras.applications import densenet
from keras.models import Model
from keras.layers import Dense

# https://www.kaggle.com/datasets/theewok/chexnet-keras-weights

densenet = densenet.DenseNet121(weights=None, include_top=False,
                                input_shape=(224, 224, 3), pooling="avg")
output = tf.keras.layers.Dense(14, activation='sigmoid', name='output')(densenet.layers[-1].output)
model = tf.keras.Model(inputs=[densenet.input], outputs=[output])
model.load_weights("./CheXNet_weights.h5")

x = model.output
predictions = Dense(1, activation="sigmoid")(x)
model = Model(inputs=model.input, outputs=predictions)

for layer in model.layers:
    layer.trainable = True

for layer in model.layers[:-10]:
    layer.trainable = False

model.summary()

```

[18]

Figure 25: CheXNet model (uses the Kaggle dataset link to load the weights)

```

from tensorflow.keras.applications import mobilenet
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model

model = mobilenet.MobileNet(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

x = model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)

model = Model(inputs=model.input, outputs=predictions)

for layer in model.layers[:-5]:
    layer.trainable = False

model.summary()

```

[22]

Figure 26: MobileNet model

```

from keras.applications import ResNet152V2
from keras.layers import Dense, GlobalAveragePooling2D
from keras.models import Model

model = ResNet152V2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

x = model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)

model = Model(inputs=model.input, outputs=predictions)

for layer in model.layers:
    layer.trainable = True

for layer in model.layers[:-5]:
    layer.trainable = False

model.summary()

```

[19]

Figure 27: ResNet152v2 Model

Performance Table

Model	Accuracy	Precision		Recall		F1-Score	
		Pneumonia	Normal	Pneumonia	Normal	Pneumonia	Normal
EfficientNetB0	0.8141	0.86	0.74	0.84	0.77	0.85	0.76
Densenet121	0.8782	0.94	0.80	0.86	0.91	0.90	0.85
CheXNet	0.8910	0.96	0.81	0.86	0.94	0.91	0.87
ResNet152v2	0.9054	0.94	0.86	0.91	0.90	0.92	0.88
MobileNet	0.9278	0.94	0.90	0.94	0.91	0.94	0.90

The best model is the MobileNet model. It outscores all the other models in both accuracy and F1-Score. We conclude the section on pre-trained models.

Custom Model

Our custom CNN model serves as a blank slate that we can shape to fit the specifics of our diagnostic problem. However, depending on the number of layers and the size of our network, the training time can increase drastically. The following code snippet explains the architecture of the custom CNN model.

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, BatchNormalization

#Construct the CNN model

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(img_size, img_size, 3), padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

[18]

Figure 28: Custom CNN architecture

The architecture is simple. We have multiple convolutional layers accompanied by BatchNormalization layers and MaxPooling layers. After five sets of convolutional layers, we add two dense layers and our output layer. Fig 28 shows the details of the model parameters.

```
Total params: 980,289
Trainable params: 979,329
Non-trainable params: 960
```

Figure 29: Custom CNN model parameters

After building the model, we train the model with the augmented dataset and evaluate the model on the test set.

```

... 20/20 [=====] - 9s 380ms/step
20/20 [=====] - 8s 363ms/step - loss: 0.2925 - accuracy: 0.9215
Accuracy: 92.14743375778198 %
Loss: 0.2924772799015045

```

Figure 30: Custom CNN accuracy and loss

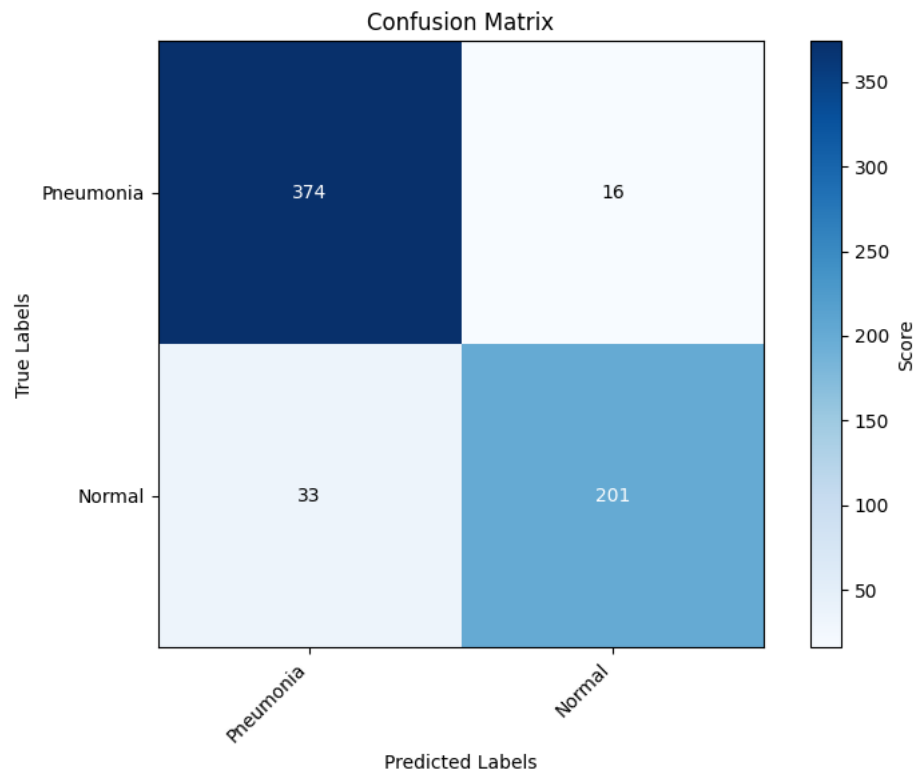


Figure 31: Confusion matrix

	precision	recall	f1-score	support
Pneumonia (Class 0)	0.92	0.96	0.94	390
Normal (Class 1)	0.93	0.86	0.89	234
accuracy			0.92	624
macro avg	0.92	0.91	0.91	624
weighted avg	0.92	0.92	0.92	624

Figure 32: Classification Report

Conclusion

The accuracy of the custom model falls just short of the accuracy obtained by the MobileNet model. We summarize the model results and model parameters in a table.

Model	Accuracy	Precision		Recall		F1-Score	
		Pneumonia	Normal	Pneumonia	Normal	Pneumonia	Normal
EfficientNetB0	0.8141	0.86	0.74	0.84	0.77	0.85	0.76
Densenet121	0.8782	0.94	0.80	0.86	0.91	0.90	0.85
CheXNet	0.8910	0.96	0.81	0.86	0.94	0.91	0.87
ResNet152v2	0.9054	0.94	0.86	0.91	0.90	0.92	0.88
Custom CNN	0.9215	0.92	0.93	0.96	0.86	0.94	0.89
MobileNet	0.9278	0.94	0.90	0.94	0.91	0.94	0.90

The accuracy of our custom CNN falls just shy of the MobileNet model.

Model	Trainable Params	Non-trainable Params	Total Params
EfficientNetB0	513,025	5,330,564	5,843,589
DenseNet121	167,169	6,871,360	7,038,529
CheXNet	180,509	6,871,360	7,051,869
MobileNet	1,052,673	3,226,816	4,279,489
ResNet152v2	2,103,297	58,327,552	60,430,849
Custom CNN	979,329	960	980,289

Figure 33: All Models' Parameters

We have saved the weights and the parameters of the MobileNet and Custom CNN models as .h5 files.

This report demonstrates the use of machine learning tools in medical diagnostics. To prepare the dataset for learning, we implemented data preprocessing and data augmentation. We delved into the domain of transfer learning by employing pre-trained models and comparing their performance against a custom-designed CNN model tailored to our diagnostic task. This comparison investigation sheds light on the skill of adapting current models' knowledge and innovating for healthcare requirements.