X-RayNet: Automated Pneumonia Detection from Chest X-rays using Machine Learning

Konstantina-Vasiliki Tompra
Filippos Iakovidis
MSc in Data Science
International University Of Thessaloniki

Abstract

The detection of pneumonia from chest X-rays is a critical task in medical imaging. In this project, we propose a deep learning-based approach for pneumonia classification into three distinct classes: bacterial pneumonia, viral pneumonia, and non-pneumonia. Leveraging the power of deep learning algorithms, we aim to revolutionise the accuracy and efficiency of pneumonia diagnosis, enabling faster and more reliable decision-making for medical professionals. The project emphasises the construction of diverse deep learning architectures, with a specific focus on the utilisation of transfer learning. Transfer learning involves leveraging pre-trained convolutional neural network (CNN) models as a foundation and subsequently fine-tuning them using the pneumonia dataset. This approach, with the contribution of specific callbacks that were employed, seemed to enhance the classification performance of all the models used. Following, the research highlights the impact of different architectural choices, hyperparameter settings, and optimization techniques on the classification results and discusses potential challenges and limitations associated with pneumonia detection using deep learning, including class imbalance, and generalisation to diverse patient populations. Based on our findings, employing the EfficientNetB0 model, initialised with weights pre-trained on the ImageNet dataset, resulted in the highest accuracy achieved at 85.2%. The high accuracy obtained by this model showcases its effectiveness and suggests its practicality as a valuable tool in the medical field.

Keywords

Pneumonia, X-rays, Transfer learning, Deep learning, Image classification, EfficientNetB0

Data & problem description

The dataset used are mostly taken from the Chest X-Ray Images(Pneumonia) released by Zhang et al. (2018) also publicly available on the Kaggle platform which consists of 5,840 frontal chest X-ray images out of which 4,672 are used for training our model and 1,168 are used in order to get classified by the model. Each one of the training images is labelled with one of the three categories non-pneumonia(class 0), bacterial pneumonia(class 1) and viral pneumonia(class 2, and is considered as ground truth. Each class contains a variable number of images, with a sufficient representation of different pneumonia types to ensure comprehensive training and evaluation. Out of the 4,672 training images, 1,227 correspond to a subject without disease (normal), 2,238 correspond to a subject with bacterial pneumonia, and 1,207 correspond to a subject with viral pneumonia. Last, there is also a csv file that contains the class labels of the training images in the form of pairs (file_name, class_id).

Screening algorithms used for detecting lung nodules in chest X-rays can also be utilised for diagnosing various other illnesses, including pneumonia, effusion, and cardiomegaly. Pneumonia, in

particular, is a highly infectious and life-threatening disease that affects millions of individuals, especially those aged 65 and above and those with underlying chronic conditions such as asthma or diabetes [11]. Chest X-rays are considered the most effective diagnostic tool for assessing the extent and location of the infected regions in the lungs during pneumonia diagnosis. However, interpreting chest radiographs is a challenging task for radiologists. Pneumonia can appear indistinctly in chest X-ray images, leading to misinterpretation and confusion with other diagnoses. The assessment of chest X-rays, specifically for pneumonia, can be misleading since other conditions like congestive heart failure or lung scarring can mimic pneumonia. Therefore, developing an algorithm for accurately detecting thoracic diseases like pneumonia would not only aid in accurate diagnosis but also improve accessibility to clinical settings in remote areas.
The primary objective of this project is to develop an accurate and efficient deep learning-based system for pneumonia detection from chest X-ray images. The system aims to classify the X-ray images into one of three classes: pneumonia, virus pneumonia, and non-pneumonia. This classification will assist medical professionals in diagnosing and managing pneumonia cases effectively. The challenge lies in accurately distinguishing between the different pneumonia subtypes and identifying cases where pneumonia is absent. Pneumonia can have various causes, including viral infections, bacterial infections, and non-infectious factors. Therefore, the deep learning model needs to learn distinctive patterns and features associated with each pneumonia subtype while being able to differentiate normal cases from pneumonia. Additionally, the system must address potential complexities in the X-ray images, such as varying image quality, overlapping anatomical structures, and potential class imbalances within the dataset. Robustness and generalisation are key requirements to ensure accurate predictions on unseen data. The successful development of an accurate pneumonia detection system has the potential to enhance the efficiency and effectiveness of pneumonia diagnosis, leading to timely and appropriate treatment decisions. Moreover, it can alleviate the burden on medical professionals by providing an automated tool for triage and initial assessment, ultimately improving patient outcomes and healthcare management.

Models' description

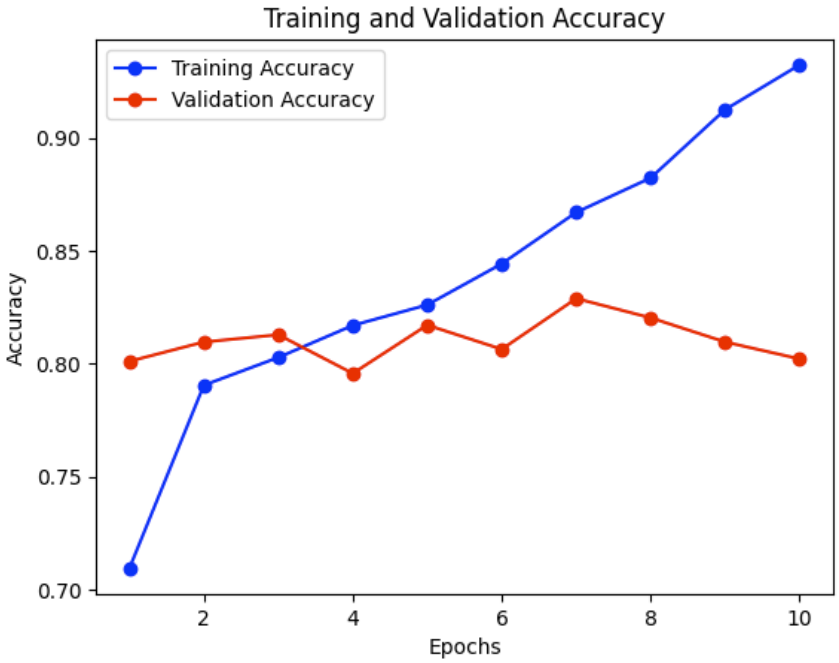
We will start with presenting the main part of our code, which was used in almost all of our submissions.

After mounting our google drive and importing our basic libraries, we used a for loop in order to read the entire train_images folder. We created a list with images and labels and appended every image with its corresponding label inside the for loop. We did the same thing with the test_images, obtaining the test image filename as well. After that, we turned these lists into numpy arrays and proceeded on normalising the images by dividing 1.0 / 255.0 and subtracting the means. After building each of our models we made the predictions on the test_images and finally saved the results in a csv. Next, we will be presenting our models while giving them a corresponding name since we are going to be presenting a comparative table with the performance of each model.

In our first try we built a sequential 9 layer simple CNN which we will call **9-Layer**.

```
model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(3, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
```

Note that in this tryout, we tried converting our train_images into grayscale hence the 1 channel in our initial input shape = (224, 224, 1). We ran the training for 10 epochs with a batch size of 32. The results weren't the best, as we came across overfitting.



In order to fix that, we tried adding a Dropout layer of 0.5 and L2 regularisation techniques. The improvement was not the desired one.

Moving on we realised that our train images should be augmented, so that our model would have more train data and therefore be able to perform better training, leading to better results. We tried defining a function which would resize, normalise and perform random rotation, horizontal and vertical flip and add random brightness on our images. We called this function in our for loop (which was reading the train images from train_images) and used it in our images. Even though the idea was right, the execution was poor. The method that we chose to go with, was actually taking all the images from our train files, one by one and practically distorting them.

```
resize_and_rescale = tf.keras.Sequential([
  Resizing(224, 224),
  Rescaling(1./255),
  RandomFlip("horizontal_and_vertical"),
  RandomRotation(0.2),
  RandomBrightness(factor=0.2)

])
```

This is the function that we created and we then trained our model using the new training data. Let's call this model the 9-**Layer-Aug.** The results were similar to our first.

After revisiting our initial **9-Layer** model, excluding the unsuccessful augmentation attempts, we conducted further experimentation. In an effort to address overfitting, we decided to reduce the learning rate to lr = 0.0005 and implemented an early stopping mechanism based on the validation loss. This modified version of the model will be referred to as "**9-Layer-ES**".

Upon analysing the subsequent graph, it became evident that the reduction in overfitting was achieved as intended. However, despite these improvements, the results obtained from the 9-Layer-ES model continued to fall short of our expectations. Despite the efforts made to mitigate overfitting, the model's performance remained inadequate in meeting our desired objectives.

After careful consideration, we reached the realisation that our current model may not be the most optimal choice for our task. As a result, we embarked on a new phase of exploration by experimenting with different pre-trained models.

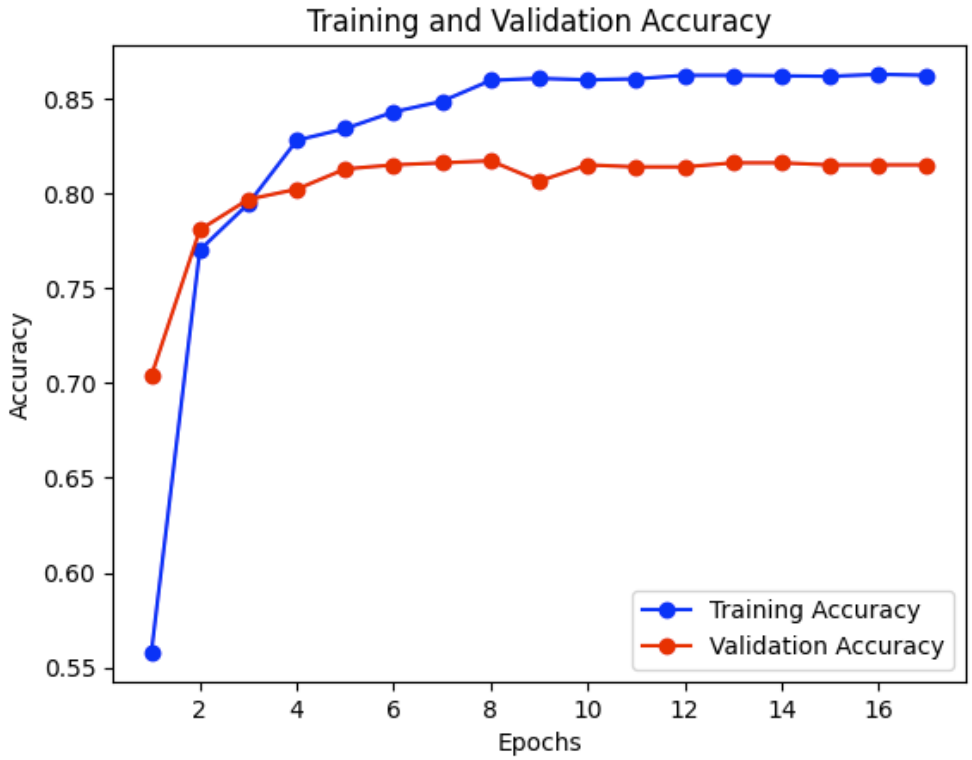
On our first attempts we loaded the **ResNet50-1Channel** model, the **Xception** model and the **DenseNet121** model, all initialised with weights pre-trained on the ImageNet dataset and applying averagePooling instead of just flattening before feeding it to the last dense layer.
We also froze various layers of the models' depending on the model. We chose to freeze the first 4 or 8 layers so that they will not be updated during training which was done to preserve the pre-trained weights in the base model and prevent them from being modified during fine-tuning.
The first two model achieved approximately the same accuracy on the unseen data, about 80,5%, while DenseNet121 had a better performance of 83% and the validation loss was noticeably decreased which was quite promising to consider keep using transfer learning.
Below we typically demonstrate the code snippet for the ResNet model, along with its performance on training and validation sets.

```python
# Load the pre-trained ResNet50 model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Add custom classification layers on top of the base model
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation='relu')(x)
predictions = Dense(3, activation='softmax')(x)

# Create the final model
model = Model(inputs=base_model.input, outputs=predictions)

for layer in model.layers[:-8]:
    layer.trainable=False

for layer in model.layers[-8:]:
    layer.trainable=True
```

Moving on, if we print the shape of the train images, we can see that they have 3 channels. That gave us the idea to keep the channels of the images as they are, and play with different pretrained models. Most of the models that we found were built around the fact that their images had 3 channels. We tried toying around with the first layer of the pretrained models and setting the input tensor to have 1 channel, but that ruins the entire architecture of the pretrained model. So we decided to re-read our images without changing their channel to 1, and ran the loadable known models.

For our 3-channel pretrained models we began with training the **VGG-16** model. We split our training data on 80%-20% as usual and we ran the training for 17 epochs.

```python
# Load the VGG16 model with pre-trained weights
model = VGG16(weights='imagenet', include_top=True)


from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import ModelCheckpoint, EarlyStopping
# Callbacks
checkpoint2 = ModelCheckpoint(filepath='best_weights_vgg16.hdf5', save_best_only=True, save_weights_only=True)

lr_reduce = ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=2, verbose=2, mode='max')
early_stop = EarlyStopping(monitor='val_loss', min_delta=0.1, patience=1, mode='min')

# Freezing first layers
for layer in model.layers[:-8]:
    layer.trainable=False

for layer in model.layers[-8:]:
    layer.trainable=True

# Compile the model
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
```



In the same concept, we trained various models like **ViT** and **ResNet152.** The results were similar but not the best.

Also, we tried using data augmentation in order to provide more images for the training but the results did not exceed our best score. We also tried less and simpler augmentation than the ones you see below, to prevent “confusing” our model. In particular, we tried only adding a small zoom rate, height and width shift range.

```
# Specify the path to your train images directory
train_images_dir = path

# Load the CSV file containing the file names and labels
labels_file = labels_path
labels_df = pd.read_csv(labels_file)

# Convert the "class_id" column to string
labels_df['class_id'] = labels_df['class_id'].astype(str)

# Extract the file names and labels from the DataFrame
file_names = labels_df['file_name'].values
labels = labels_df['class_id'].values

# Split the data into train and validation sets
train_files, val_files, train_labels, val_labels = train_test_split(
    file_names, labels, test_size=0.2, random_state=42)
```

```
# Create an ImageDataGenerator object with desired augmentation settings
datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=5,
    width_shift_range=0.05,
    height_shift_range=0.05,
    brightness_range=[0.9, 1.1],
    shear_range=0.05,
    zoom_range=0.05,
    fill_mode='nearest'
)



# Load and preprocess your image data using the ImageDataGenerator
train_generator = datagen.flow_from_dataframe(
    dataframe=pd.DataFrame({'file_name': train_files, 'class_id': train_labels}),
    directory=train_images_dir,
    x_col='file_name',
    y_col='class_id',
    target_size=(224, 224),
    batch_size=32,
    class_mode='sparse',
    shuffle=True
)

validation_generator = datagen.flow_from_dataframe(
    dataframe=pd.DataFrame({'file_name': val_files, 'class_id': val_labels}),
    directory=train_images_dir,
    x_col='file_name',
    y_col='class_id',
    target_size=(224, 224),
    batch_size=32,
    class_mode='sparse',
    shuffle=False
)
```

We also tried different optimizers for our training such as: Adagrad, RMSprop and SGD with no better results.

Next we experimented with the **EfficientNetB0** model, which at our very first try, achieved 83% accuracy, with the next versions of it (**EfficientNetB6, EfficientNetB7)** having poorer performance and demanded much more time and RAM resources.

After that, we tried various ensemble methods based on **EfficientNetB0** that performed best. We created 2 EfficientNetB0 models while slightly changing some hyperparameters and then we made the predictions on the test images.

```
# Ensembles best weights from models model1, model2.
#Two efficient net b0 models - added regularization and dropout on model2. Predictions on the best weights per model
# model.load_weights('best_weights.hdf5')
model1.load_weights('best_weights1.hdf5')
model2.load_weights('best_weights2.hdf5')
```

```
# pred = model.predict(test_images)
pred1 = model1.predict(test_images)
pred2 = model2.predict(test_images)
```

```
37/37 [==============================] - 4s 75ms/step
37/37 [==============================] - 3s 57ms/step
```

```
ensemble_pred = (pred1 + pred2) / 2
```

```
ensemble_labels = np.argmax(ensemble_pred, axis=1)
```

We did the same thing with 3 EfficientNetB0 models and we also tried using ensembles on 2 or 3 different models. For example one EfficientNetB0, one ViT and one VGG16. Surprisingly none of the above results were an improvement to our single best weights B0 predictions.
Lastly, we successfully managed to perform K-fold cross validation with K=5. We practically trained the entire model 5 times, each time on a different train-validation split on our train images. We saved the best weights for each run, and used an ensemble for our final prediction. Sadly, this didn't achieve a better performance either.

```
ensemble_predictions = []

# Create a random array of integers from 0-4, one for each image
folds = np.random.randint(0, 5, len(images))

for valid_fold in range(5):

    train_indices = np.where(folds != valid_fold)[0]
    valid_indices = np.where(folds == valid_fold)[0]

    train_images = [images[k] for k in train_indices]
    train_labels = [labels[k] for k in train_indices]

    valid_images = [images[k] for k in valid_indices]
    valid_labels = [labels[k] for k in valid_indices]

    # Convert the images and labels to arrays
    train_images = np.array(train_images)
    train_labels = np.array(train_labels)
    valid_images = np.array(valid_images)
    valid_labels = np.array(valid_labels)
```

```
# Callbacks
checkpoint2 = ModelCheckpoint(filepath=f'best_weights{valid_fold}.hdf5', save_best_only=True, save_weights_only=True)

lr_reduce = ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=2, verbose=2, mode='max')
early_stop = EarlyStopping(monitor='val_loss', min_delta=0.1, patience=1, mode='min')
model.compile(loss='sparse_categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
history = model.fit(
    train_images, train_labels, epochs=17, batch_size=32,
    validation_data=(valid_images, valid_labels),
    callbacks=[checkpoint2, lr_reduce]
)

# Load the best weights of the current model
model.load_weights(f'best_weights{valid_fold}.hdf5')

# Make predictions on the test set using the current model
predictions = model.predict(test_images)

# Append the predictions to the ensemble list
ensemble_predictions.append(predictions)
```

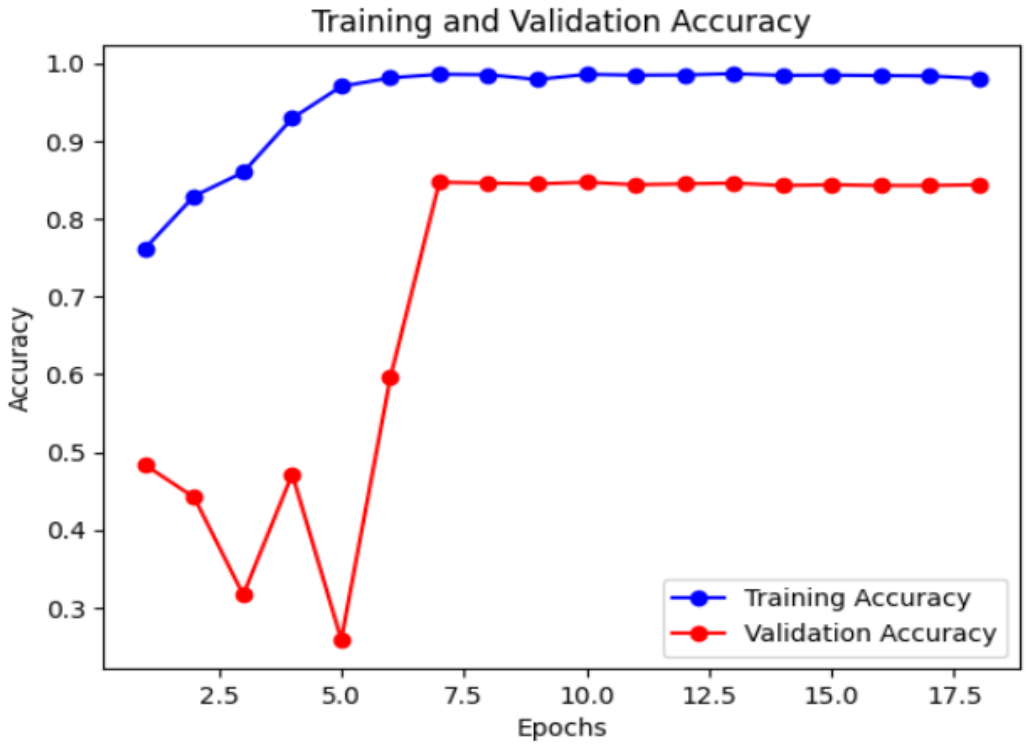
Best Model

Subsequently, we proceeded to use the **EfficientNetB0** model for further experimentation while it was quite promising.

We froze all of the pre-trained model's layers except for the last 10, this time, which we allow to be trainable, meaning that their weights will be updated during training. During the training process, we incorporated two callbacks, a learning rate reduction function and a checkpoint system to save the best weights obtained during model training in an hdf5 file. Following the completion of training, we loaded the model using the saved best weight file and utilised it to make predictions.

Throughout the experimentation phase, we changed various hyperparameters of the model in order to seek the best results. After thorough evaluation, we finalised our best submission, which we named "**B0Last10**". Notably, our training process spanned 18 epochs.

```python
# Load the EfficientNetB0 pre-trained weights without the top (classification) layer
base_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the pretrained weights
base_model.trainable = False

# Model
model = Sequential()
model.add(base_model)
model.add(GlobalAveragePooling2D())
model.add(Dense(256, activation='relu'))
model.add(Dense(3, activation='softmax'))

# Unfreeze the 10 last layers
for layer in model.layers[:-10]:
    layer.trainable=False

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

# Compile the model
optimizer = Adam(learning_rate=0.0005)
model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```



The graph might seem a bit iffy because it starts learning quite slowly and it looks like it converges. With a lower initial learning rate and fewer epochs the graph would seem better. Nevertheless, by creating a checkpoint for our best weights during training, we achieved our best accuracy.

Comparative table of achieved results

Model	***Accuracy***
9-Layer	***0.78253***
9-Layer-Aug	***0.80308***
9-Layer-ES	***0.76541***
ResNet50-1Channel	***0.80684***
VGG-16	***0.80308***
EfficientNetB0	***0.83547***
B0BestWeights	***0.84417***
EnsembleB0-2M	***0.80821***
EnsembleB0-3M	***0.83732***
K-FoldsEnsembleB0	***0.84246***
B0Last10	***0.85273***
DenseNet121	***0.83132***
AdaboostB0	***0.82876***

Note that we have trained many different variations of our mentioned models, using different augmentation, changing the hyperparameters, changing the amount of frozen layers, ensembles, Adaboost etc.

Conclusion

Having expert radiologists is crucial for accurate diagnosis of thoracic diseases. However, in many areas, the availability of radiologists is limited. This research aims to enhance medical capabilities in such regions by enabling early detection of pneumonia to prevent adverse outcomes, including fatalities. Developing algorithms in this field can greatly improve healthcare services. In this study, we evaluated the performance of various pre-trained CNN models and different classifiers while we came up to the finding that the EfficientNetB0 seems to be the best model for this particular image classification task. We also demonstrated the effectiveness of hyper-parameter optimization in improving model performance. Through a series of experiments, our objective is to identify the most effective pre-trained CNN model and classifier, providing valuable insights for future research in this area. The outcomes of our study are expected to assess the development of advanced algorithms for pneumonia detection in the near future.