# Convolutional Neural Network Applied to X-Ray Medical Imagery for Pneumonia Identification

Denis Manolescu[1,2], Neil Buckley[1], and Emanuele Lindo Secco[2] [0000-0002-3269-6749]

[1]AI Lab, School of Mathematics, Computer Science & Engineering, Liverpool Hope University
[2]Robotics Lab, School of Mathematics, Computer Science & Engineering, Liverpool Hope University
20203547@hope.ac.uk, bucklen@hope.ac.uk, seccoe@hope.ac.uk

**Abstract.** Convolutional Neural Networks (CNN) are one of the most popular and effective approaches for image recognition. They have been widely used in various medical imaging applications, such as identifying tumors, detecting lesions, and organ segmentation. The ability of CNNs to automatically learn and extract meaningful features from raw visual data makes them particularly well-suited for medical imaging tasks, where accuracy and reliability are critical. This paper presents an experimental study focused on analyzing the structure and functionality of convolutional neural networks by building an operational model capable of identifying cases of pneumonia from X-ray scans. The CNN model is trained, validated and tested on a dataset of over 5000 images, and the final results show 99% precision and 98% accuracy, with a recall value of 98%.

**Keywords:** convolutional neural networks; decision support system; intelligent system; medical image analysis; diagnostics.

## 1 Introduction

Centuries of collective learning and research have brought a solid knowledge of how the human brain works. Still, given its complexity, shaped by millions of years of natural evolution, this self-understanding endeavor is far from over, and it continues with the help of more powerful and advanced computers and algorithm models. When reduced to numbers, the vastness of the brain's structure can be beautiful but overwhelmingly convoluted, formed by an estimated 86 billion neurons connected by around one quadrillion synapses, plus other less documented brain cells [1]. With its remarkable capabilities of processing and transmitting information, this multi-specialized and self-organized neural system is used as a structural template for building artificial intelligence computing systems that can enhance our collective knowledge.

For the human mind, the ability to identify objects or collect information from images is an effortless and trivial process, while for a computer, this task can be troublesome and hard to implement. Machine learning algorithms, particularly deep learning models such as Convolutional Neural Networks (CNNs or ConvNets), focus on emulating the works of the brain, enabling them to detect patterns and make predictions based on visual data. Being detached from human biases and exhaustion, these supervised learning algorithms, trained on vast amounts of data, can achieve performance levels similar to or even superior to any individual, making them ideal for quick and consistent tasks. CNNs have shown promising results in several medical imaging challenges, such as the ISIC Skin Lesion Classification Challenge with the work of Gouda and colleagues [2], Dalmau et al [3] or the RSNA Pneumonia Detection Challenges.

This research demonstrates the performance of a convolutional neural network model in the process of medical analysis and disease diagnosis based on X-ray imagery. The study aims to support the potential use of this technology in conjunction with doctors' expertise to validate and facilitate medical assessments in a responsible and transparent manner. In practice, this work presents an implementation of a CNN model trained to identify and ultimately automate and speed up the process of diagnosing pneumonia infection from X-ray images.

The paper is structured as it follows: in the following section, the CNNs Concept and Structure paragraph will analyze in detail the basic structure of the convolutional neural network and the working mechanism behind each layer. Furthermore, the discussion will corroborate the results of other relevant studies from the literature to homogenize the view related to the performance of different algorithm techniques. Then, the Implementation paragraph will examine the execution and building of the CNN model and present each dataset processing stage, including cleaning, loading, normalization and scaling, and partitioning. The Results section elaborates on the training process of the model and its performance and concludes by evaluating the trained network on unseen data. The final Conclusion reflects on the perspective and potential of using this technology, with advantages and limitations, while proposing some improvements.

## 2 CNNs – Concept and Structure Review

### 2.1 What are CNNs

*Convolutional Neural Networks* (CNN) is a class of supervised Deep Learning architecture emulating the visual cortex in the human brain, designed to analyze and process high-dimensional visual data, predominantly images and videos. Typically, this type of neural network is composed of several sequential layers, including *Input*, *Convolutional*, *Activation*, *Pooling*, *Flatten* and *Fully-Connected Layers*, which work together to learn and identify hierarchical patterns, features, and relationships within a dataset.

The input layer in a CNN consists of a series of operations that prepares the dataset for processing by the feature detection layers. In this stage, the input data is normalization and standardized into specific formats and parameters, ensuring network stability and convergence.

### 2.2 Convolutional Layer and Activation Function

A fundamental component in any CNN is the *Convolutional Layer*, which is specialized in detecting local features like edges, texture or shapes within an input image. This process starts with a user-defined number of kernels or filters, initially operating with random weights values. While the CNN is training, it uses backpropagation and loss optimization algorithms to update and refine the weights according to the labels within the dataset.

In essence, the features refinement is achieved by performing a mathematical operation called convolution which takes place during each forward pass cycle of training. This procedure consists of sequentially sliding each kernel over the input image at a predefined stride value of pixels to produce a feature map. At each kernel spatial position on the input data, a dot product is computed between the filter weights and the corresponding input values, forming an activation map which represents the spatial arrangement of that learned feature.

The convolutional layer operations continue by applying a non-linear transfer function to the *activation maps*, having the essential purpose of integrating non-linearity into the model and allowing it to learn complex patterns by regulating the neurons output.

The *Rectified Linear Unit* (ReLU) is one of the most widely used activation functions in CNNs development and it operates by setting all negative values within the feature maps to zero.

Although other techniques, such as Sigmoid, Mish, Swish or other derived hybrid algorithms, can produce good results, as shown in the work of Dasgupta et al [4] and in the benchmark studies carried out by Dubey et al [5], ReLU algorithm remains the preferred option for image classification models due to simplicity and high yield accuracy-time results. However, the choice of the transfer function really depends on the application, the dataset and the intended properties of the CNN model.

### 2.3 Pooling Layers

When developing a convolutional neural network, the amount of data is a paramount aspect of generating performance, consequentially leading to the potential of overgrown and overfitting models that can exhaust the computation powers of the system. These spatial dimensional issues can be controlled and reduced by using pooling layers. The modular advantage of these techniques is their capacity to operate independently on each feature map, where they can perform non-linear down-sampling while preserving most of the relevant features, essentially producing a newly condensed feature map. The two most commonly used pooling operations are Max Pooling and Average Pooling.

Although various CNN experiments focused on generalizing and trying to benchmark the performance of different traditional pooling techniques, like the work of Galanis et al, Nirthika et al, Zafar et al [6-8], the general conclusion seems to be that the pooling method is just another refinement option determined by the scope and priority of the model. The choice depends on the specific problem and the desired trade-offs, and it is often determined through experimentation.

Max pooling is preferred in the context of preserving the most prominent information of the local regions, which often corresponds to the presence of specific patterns or features. These abilities favor the mapping of edges, corners and textures and are particularly useful in image classification, object detection, facial and gesture recognition, autonomous driving or image segmentation applications. In contrast, the average pooling retains a uniform local representation of the input feature, resulting in a smoother activation map that preserves more information and is less sensitive to noise. The average pooling may be preferred in the applications that need more contextual space, like satellite image analysis, handwriting recognition, artistic style transfer or medical image segmentation. Additionally, max pooling generates fewer learning parameters than average pooling, resulting a faster training and fewer computational resources.

## 2.4 Flatten Layer

The Flatten layer is an essential component in the classification process of a standard CNN model that bridges the transition from the convolution-activation-pooling stage, responsible for the feature detection tasks, to the Fully-Connected Layers. In essence, a *Flatten Layer* performs a multidimensional-to-unidimensional transformation by taking the output of the feature detection block, which is a stack of feature maps, and converting it into a one-dimension linear array format. In this process, the linearization operation preserves the spatial relationship within the activation maps, which eventually become spatial context used by the Fully-Connected Layers.

The scope of this linearization process is to simplify the data representation to allow the Fully-Connected Layers to process the information and perform high-level predictions for classification or regression tasks based on the identified features.

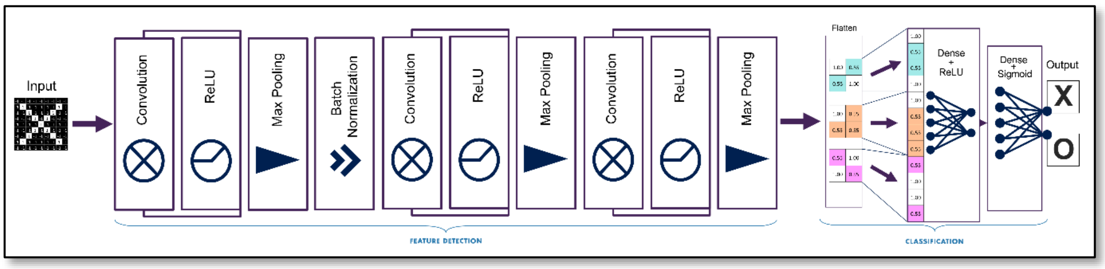


**Fig. 1.** The CNN *layer by layer* final structure design

## 2.5 Fully-Connected layers and SoftMax

After the Flatten Layer linearizes the data, it is then passed through one or more Fully-Connected or Dense Layers. These are sequentially organized and specialized in combining the high-level features extracted in the convolutional-pooling block to learn how to recognize the patterns and their relationship. This mechanism determines the combination of features and their weights to generate the final prediction, which can either be a class probability for classification tasks or continuous numerical values in the case of regression tasks.

Often, a CNN model ends with a Fully-Connected Layer, but when dealing with multi-class classification challenges, the final layer can be a *SoftMax Activation Function* that uses the output values of the last fully-connected layer to calculate the multi-class probability distribution that sums up to one.

### 2.6 Additional CNN Layers

Because of their powerful abilities and the capacity to handle visual variations like changes in orientation, scale, illumination, and distortion, CNNs have become widely used in various computer vision tasks. This is also the case of Naveen et al [9] with their work in training a CNN image recognition model to accurately identify cherries for a machine picking system. Similar is the work of Sun et al [10] in developing an image segmentation model to identify and recognise human palm veins or the work of Jinchuan et al [11] and Yona et al [12] to accurately identify dangerous objects from X-ray security screening.

Apart from the standard structure, Convolutional Neural Networks can also incorporate other different types of layers with a variety of benefits. Integrating Normalization layers into the CNN model can help stabilize and improve convergence during training by homogenising the data, as Zhijie et al [13] and Suzuki et al [14] proved in their studies. Additionally, Dropout layers can be used to regulate the data by randomly deactivating a portion of neurons. This process prevents overfitting and promotes generalization, advantages shown by Tingting et al [15] and Farhadi et al [16]. Furthermore, as Xiaofeng et al [17-18] illustrate, the Deconvolution Layers can also be utilised to perform image reconstruction or generative modelling of new data, which ultimately could be used for additional training of the model, although improper usage of this feature can lead to image artefacts and error propagation within the model, as Odena et al [19] and Kirchhoff [20] identified. In addition, Depthwise Separable Convolution layers can be integrated into the CNN structure to reduce the computation demand of operations, a benefit also shown in the work of Lin et al [21]. The general modular characteristics of CNNs allow for easily expanding the structure of the model with other more exotic or specialised layers.

## 3 Implementation of the CNN

### 3.1 The Data Set

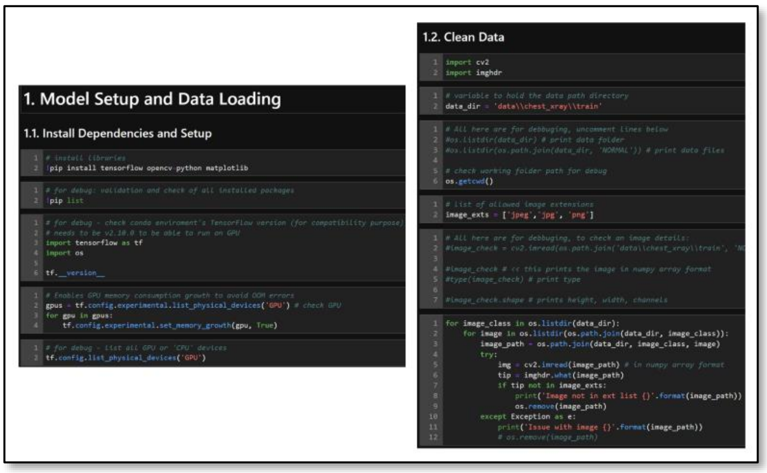
Pneumonia is a contagious viral or bacterial lung infection that can be life-threatening if not discovered early and quickly treated with antibiotics. Usually, this condition is diagnosed and evaluated with a chest *X-Ray* (WHO - Bernadeta Dadonaite, 2019 – [22]). In the effort to tackle this issue, AI technologies like CNNs trained models can offer a significant advantage to unburden any medical system and assist medical professionals in the diagnosing process.

In this study, the CNN model is trained, evaluated, and tested using a dataset comprised of approximately *5,200 X-Ray Lung Images* available for public use with the effort of Kermany and his colleagues [23]. The dataset content is divided between normal scans and pneumonia-identified scans.

### 3.2 Software Implementation

The CNN model developed in this research can be executed and retrained in *Jupyter Lab IDE* - on a local machine or using the online *Google Colab* environment. In either case, the algorithm is designed to perform on GPU runtime memory. Although CPU execution is also possible, this aspect is out of the scope of the study. Additionally, the training dataset needs to be downloaded, in both cases, by the user.

For a Google Colab implementation, the CNN model can be downloaded from GitHub. The train in Colab takes roughly 30 min, while on a local machine with 16 GB VRAM, the process requires about 3 min, although the initial setup is more complex.



1. Model Setup and Data Loading

1.1. Install Dependencies and Setup

```
# install libraries
!pip install tensorflow opencv-python matplotlib
```

```
# for debug: validation and check of all installed packages
!pip list
```

```
# for debug - check conda enviroment's TensorFlow version (for compatibility purpose)
# needs to be v2.10.0 to be able to run on GPU
import tensorflow as tf
import os

tf.__version__
```

```
# Enables GPU memory consumption growth to avoid OOM errors
gpus = tf.config.experimental.list_physical_devices('GPU') # check GPU
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

```
# for debug - list all GPU or 'CPU' devices
tf.config.list_physical_devices('GPU')
```

1.2. Clean Data

```
import cv2
import imghdr
```

```
# variable to hold the data path directory
data_dir = 'data\\chest_xray\\train'
```

```
# All here are for debbuging, uncomment lines below
#os.listdir(data_dir) # print data folder
#os.listdir(os.path.join(data_dir, 'NORMAL')) # print data files

# check working folder path for debug
os.getcwd()
```

```
# list of allowed image extensions
image_exts = ['jpeg','jpg', 'png']
```

```
# All here are for debbuging, to check an image details:
#image_check = cv2.imread(os.path.join('data\\chest_xray\\train', 'NO

#image_check # << this prints the image in numpy array format
#type(image_check) # print type

#image_check.shape # prints height, width, channels
```

```
for image_class in os.listdir(data_dir):
    for image in os.listdir(os.path.join(data_dir, image_class)):
        image_path = os.path.join(data_dir, image_class, image)
        try:
            img = cv2.imread(image_path) # in numpy array format
            tip = imghdr.what(image_path)
            if tip not in image_exts:
                print('Image not in ext list {}'.format(image_path))
                os.remove(image_path)
        except Exception as e:
            print('Issue with image {}'.format(image_path))
            # os.remove(image_path)
```

**Fig. 2.** Set up of the model with its dependencies (left panel) and pre-processing procedure for cleaning the data (right panel).

### 3.2.1 Dependencies Installation and GPU Setup

The initial step for building the CNN model is to install all the necessary libraries and dependencies using the python's package manager pip. This should include *TensorFlow* - which is part of the CNN training pipeline, *opencv-python* – the library that will facilitate the interaction with images, making sure the dataset is operational and compatible with python format, and *matplotlib* – used for graphical representation of the end results and performance. Training a CNN model on GPU graphics card tends to expand the VRAM available, which can lead to *Out Of Memory* errors. This issue is managed by initializing the GPU and enabling the *experimental.set_memory_growth* function, which will gradually increase the VRAM as the training requires. As debug measure, the setup stage ends with checking the TensorFlow version and also checking whether IDE recognizes GPU device (Fig. 2, left panel).

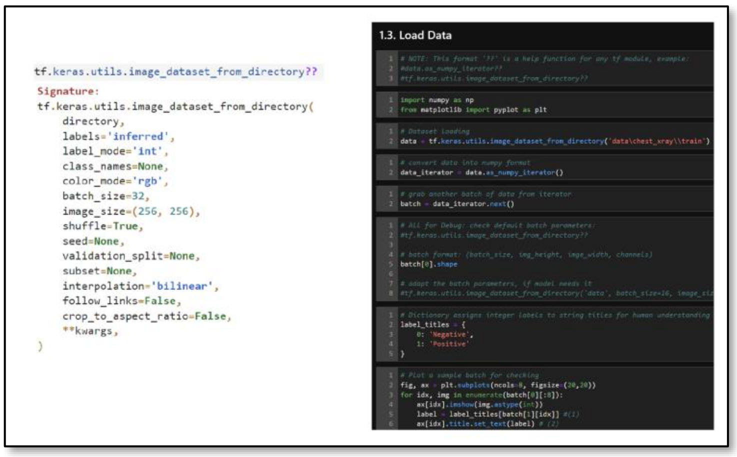
### 3.2.2 Data Cleaning

The interaction with the CNN dataset is possible by using the *cv2 Python Optimization Library* for Computer Vision that offers great support for image-based tasks like read, write, filter. Also, the *imghdr module* is used for fast file extension detection.

Most of the time, the dataset can be messy, with different unusable formats that can interfere with the CNN training, for that reason a data cleaning process is mandatory. This is implemented by first defining a *data_dir* variable to hold the files path folder, and an *image_exts* list to hold the allowed extensions. Next, for loops are used to go through folders, open each image and check its extension. If any image cannot be open or is bad format, they will be removed using the remove function from python OS module (Figure 2, right panel).

### 3.2.3 Data Loading

*TensorFlow* relies on the *Keras API* to re-format its dataset as a pipeline, and in practice the framework is implemented as a generator to pre-process and fast-load data in small batches during training and evaluation (Figure 3, right panel). Precisely, the function *tf.keras.utils.image_dataset_from_directory* returns a variable data as a labelled object of *tf.data.Dataset* (Figure 3, left panel). In this format each element of the dataset is basically a tuple of a batch of images and a batch of labels. Furthermore, images are represented as tensors of shape (i.e. the *batch_size*, *image_height*, *image_width* and *num_channels*), while the labels are represented as tensors of shape (i.e. *batch_size* and *num_classes*).



```
tf.keras.utils.image_dataset_from_directory??
Signature:
tf.keras.utils.image_dataset_from_directory(
    directory,
    labels='inferred',
    label_mode='int',
    class_names=None,
    color_mode='rgb',
    batch_size=32,
    image_size=(256, 256),
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation='bilinear',
    follow_links=False,
    crop_to_aspect_ratio=False,
    **kwargs,
)
```

1.3. Load Data

```
# NOTE: This format '??' is a help function for any tf module, example:
#data.as_numpy_iterator??
#tf.keras.utils.image_dataset_from_directory??
```

```
import numpy as np
from matplotlib import pyplot as plt
```

```
# Dataset loading
data = tf.keras.utils.image_dataset_from_directory('data\chest_xray\\train')
```

```
# convert data into numpy format
data_iterator = data.as_numpy_iterator()
```

```
# grab another batch of data from iterator
batch = data_iterator.next()
```

```
# All for Debug: check default batch parameters:
#tf.keras.utils.image_dataset_from_directory??

# batch format: (batch_size, img_height, imge_width, channels)
batch[0].shape

# adapt the batch parameters, if model needs it
#tf.keras.utils.image_dataset_from_directory('data', batch_size=16, image_si
```

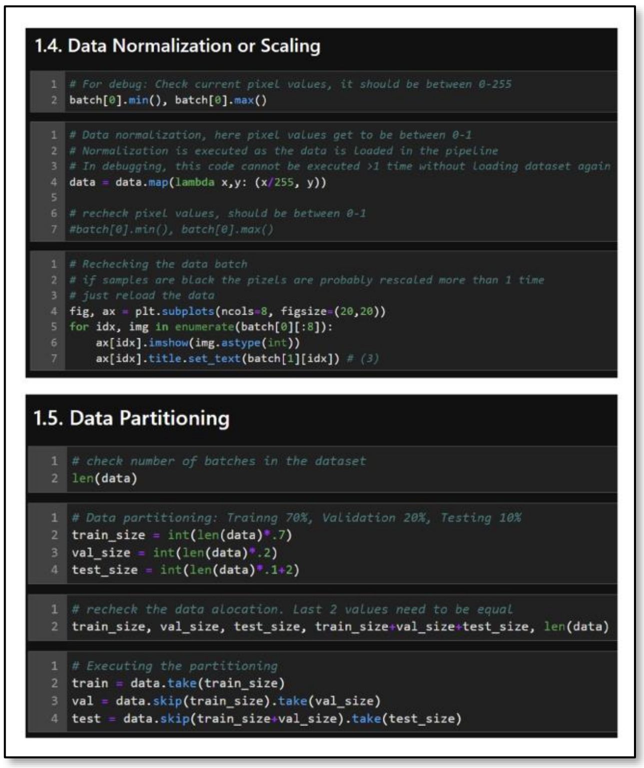
```
# Dictionary assigns integer labels to string titles for human understanding
label_titles = {
    0: 'Negative',
    1: 'Positive'
}
```

```
# Plot a sample batch for checking
fig, ax = plt.subplots(ncols=8, figsize=(20,20))
for idx, img in enumerate(batch[0][:8]):
    ax[idx].imshow(img.astype(int))
    label = label_titles[batch[1][idx]] #(1)
    ax[idx].title.set_text(label) # (2)
```

**Fig. 3.** The *Keras Utils* method attributes (left panel) and the Input Layer for *Data Loading* (right panel).

The order of the images in each batch is random and changes from batch to batch during training, in order to reduce bias and improve the performance of the CNN model. This random ordering ensures that the model does not memorize the order of the images in the dataset and instead learns to generalize new examples. Next, the *tf.data.Dataset* function *as_numpy_iterator()*, provides a sequence of batches of data and allows access to the generator by iterating over elements of the dataset and converts it into *NumPy arrays*. Then, the data batch is retrieved one by one using *batch = data_iterator.next()* which catches a batch from the iterator and advances the iterator to the next batch. This strategy can be a way to adapt the dataset's parameters like *batch_size*, *image_size* to the VRAM capabilities of the system, and then control the workflow.

Additionally, *label_titles* define a dictionary to assign the batch labels to the string titles, for a better human interpretation of the results, while subplots and a for loop function is used to generate a batch sample by visualizing the first eight images and their labels using the *Matplotlib* functions.



1.4. Data Normalization or Scaling

```
# For debug: Check current pixel values, it should be between 0-255
batch[0].min(), batch[0].max()
```

```
# Data normalization, here pixel values get to be between 0-1
# Normalization is executed as the data is loaded in the pipeline
# In debugging, this code cannot be executed >1 time without loading dataset again
data = data.map(lambda x,y: (x/255, y))

# recheck pixel values, should be between 0-1
#batch[0].min(), batch[0].max()
```

```
# Rechecking the data batch
# if samples are black the pizels are probably rescaled more than 1 time
# just reload the data
fig, ax = plt.subplots(ncols=8, figsize=(20,20))
for idx, img in enumerate(batch[0][:8]):
    ax[idx].imshow(img.astype(int))
    ax[idx].title.set_text(batch[1][idx]) # (3)
```

1.5. Data Partitioning

```
# check number of batches in the dataset
len(data)
```

```
# Data partitioning: Trainng 70%, Validation 20%, Testing 10%
train_size = int(len(data)*.7)
val_size = int(len(data)*.2)
test_size = int(len(data)*.1+2)
```

```
# recheck the data alocation. Last 2 values need to be equal
train_size, val_size, test_size, train_size+val_size+test_size, len(data)
```

```
# Executing the partitioning
train = data.take(train_size)
val = data.skip(train_size).take(val_size)
test = data.skip(train_size+val_size).take(test_size)
```

**Fig. 4.** Data Normalization & Scaling (top panel) and Partitioning (bottom panel).

3.2.4 Data Normalization and Scaling

The dataset normalization process is executed in the dataset pipeline and consists of scaling the image values from 0-to-255 into 0-to-1. This method helps the deep learning model to generalize faster and superior results. In practice, while the data is being pre-processed in the pipeline, the *map()* function executes the scaling by dividing each pixel value by the maximum pixel value in the batch (x/255). This is a commonly used technique to improve the training stability of the CNN, improving the convergence rate of the optimization algorithm and the speed of processing the data (Figure 4, top panel).

3.2.5 Data Set Partitioning

Partitioning the dataset is an essential process in building a stable and accurate CNN model. Properly allocating non-overlapping data subsets for training, validation and testing help to assess performance and to prevent overfitting, ultimately leading to an increase ability to generalize on new data. The training subset is used for parameters optimization through back propagation and gradient descent. The validation subset is used for tunning the hyperparameters like learning rate, number of layers or regularization parameters, while the testing data subset is used for evaluation.

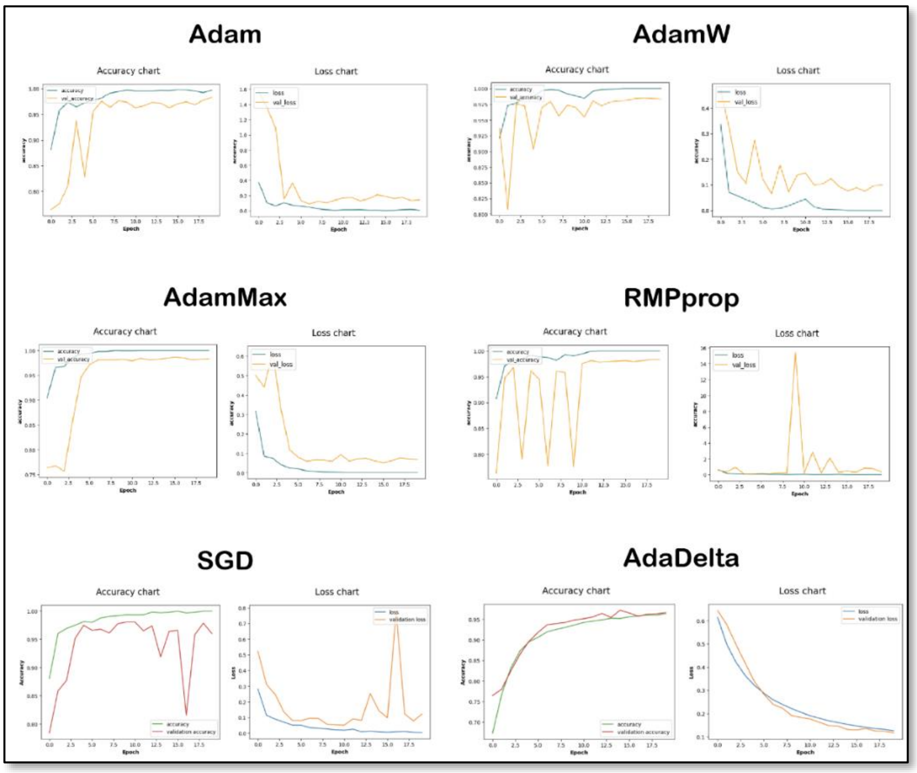


**Fig. 5.** CNN training result charts using different optimization algorithms

The CNN dataset contains 164 batches, which are divided into:

- 70% for training (*train_size*)
- 20% for validation (*val_size*)
- 10% for testing (*test_size*)

The partition is implemented using the Tensorflow *data.Dataset* pipeline methods take and skip (Figure 4, bottom panel).

## 3.3 The CNN Model

The CNN architecture is designed by means of the *Keras Sequential API*, a deep learning model builder suitable for one-input one-output data flow projects. The final architecture of the CNN model is composed of three convolutional blocks, one Flatten

Layer, one Fully-Connected Layer with ReLU Activation and another single unit Dense Layer with a Sigmoid Activation Function. Additionally, the input block integrates a Batch-Normalization Layer to minimize the loss function during training, which otherwise will result in a spike of noise during validation.

The 1st and 3rd Convolutional Layer have 16 filters, while the 2nd one has 32 filters. The rest of their hyper-parameters are the same, with a kernel size of 3x3 pixels and a one-pixel stride. All layers are sequentially stacked together using *add()* function.

In the last stage before training, the CNN model is compiled using the *Adagrad Optimizer*, the *BinaryCrossentropy()* loss function and the accuracy tracking metrics for later evaluation. In addition, the *summary()* function reviews how the model transforms the data within each block.

## 4 Results

### 4.1 CNN Training and Performance Visualization

The training data is stored in a *logs* directory, while the *tensorflow_callback* tool allows the visualisation and monitor various parameters. Next, the *fit()* method initiates the training process in which the *train* dataset is passed through the CNN model for 20 epochs. Additionally, the *hist* variable holds the training history returned by the *fit()* function, together with all the metrics information, *validation_data*, and *callback* logs.

The training performances are plotted using *Matplotlib* visualisation tools provided through TensorFlow. A comprehensive comparison table with the results of other optimizers is briefly presented in Figure 5, this includes *Adam*, *AdamW*, *AdamMax*, *PMP-prop*, *SGD* and *AdaDelta*.

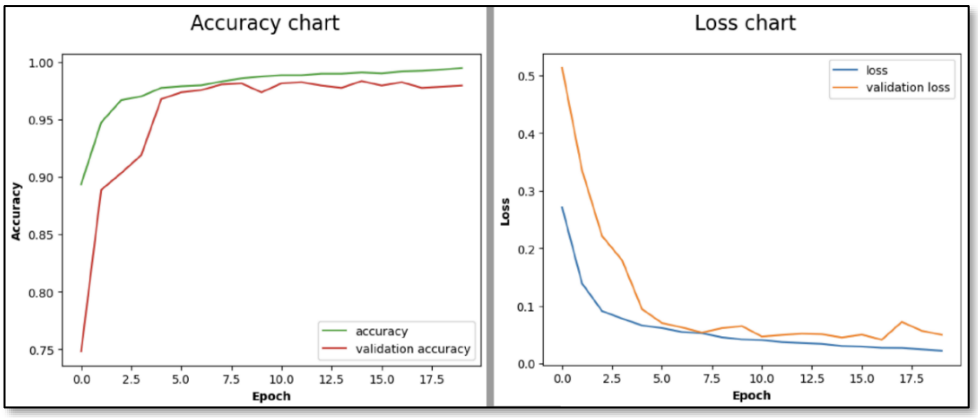


**Fig. 6.** Accuracy and loss charts from the training results.

In the final results of the research, the developed CNN model shows its best performance when integrating the *Adagrad Optimiser* (Figure 6).

### 4.2 Performance Evaluation

The performance of the newly trained CNN model is finally evaluated using the unseen Test dataset partition. This process is carried out by the *Keras Metrics Functions*, namely the *Precision*, *Recall* and *BinaryAccuracy* functions. Usually, for classification problems, these are the most relevant. Once the metrics instances are stored into variables, a for loops iterates through the test dataset using *as_numpy_iterator()* method which returns batches in sequence in a *numpy format*. Furthermore, the *x* and *y* variables unpack the input features and the true labels of the current batch. The *model.predict(x)* technique uses the freshly trained CNN model to predict the input features *x* and compare it with the true labels *y*. The result is getting stored in the *yhat* variable. In the final step, all metrics variables are updated, and the evaluation result is printed:

*Precision* = 0.9907

*Recall* = 0.9839

*Accuracy* = 0.9804

In conclusion, the CNN model developed in this research shows 99% precision and 98% accuracy, with an effectiveness of 98% given by the recall value.

In the final stage, the CNN trained model is saved and ready to be deployed as an API or used on compatible edge devices, like *Nvidia Jetson Systems* or other similar. The saving process is basically a serialization or archiving in a *.h5* format, a process executed by the *model.save()* function, giving a path (*'models'* folder) and a model name ('*cnnXray_pneumonia_LHU.h5*'). Additionally, *load_model()* can be used to reload the trained network anytime and it can directly take data for prediction or inference with the help of *predict()* function.

## 5 Conclusion

The CNN model built and trained in this research has proved the speed and reliability that can be achieved by these deep learning algorithms in identifying pneumonia infections from *X-Ray Imagery*. The experimentation results are a compelling argument showing the technology is trustworthy to assist medical personnel in their diagnosing procedure, offering remarkable advantages of endurance, consistency, and accuracy. Additionally, the success of this model in detecting pneumonia infections suggests that training the network on additional datasets of other types of health issues can further expand its capabilities. These automation systems have the potential to become highly efficient diagnostic tools that can address real-world clinical needs while reducing the workload on medical professionals, allowing them to focus on different critical tasks [24-26. Furthermore, once trained, the model has the flexibility to be deployed at a minimum cost using the existing computational infrastructure or edge devices, making it an accessible solution to isolated communities worldwide.

## Acknowledgments

This work was presented in coursework form in fulfilment of the requirements for the BEng in Robotics for the student Vasile Denis Manolescu from the AI and Robotics Labs, School of Mathematics, Computer Science & Engineering, Liverpool Hope University..

## References

1. Tompa, R., 2022. Why is the human brain so difficult to understand? (Allen Institute). [Online] Available at: https://alleninstitute.org/news/why-is-the-human-brain-so-difficult-to-understand-we-asked-4-neuroscientists/ [Accessed 2023].
2. Walaa Gouda, N. U. S. G. A.-W. M. H. N. Z. J., 2022. Detection of Skin Cancer Based on Skin Lesion Images Using Deep Learning. [Online] Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9324455/ [Accessed 2023].
3. Marta Cullell Dalmau, S. N. M. O.-V. I. M. C. M., 2021. Convolutional Neural Network for Skin Lesion Classification: Understanding the Fundamentals Through Hands-On Learnin. [Online] Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7969634/ [Accessed 2023].
4. Rupshali Dasgupta, Y. S. C. S. N., 2021. Performance Comparison of Benchmark Activation Function ReLU, Swish and Mish for Facial Mask Detection Using Convolutional Neural Network. [Online] Available at: http://dx.doi.org/10.1007/978-981-16-2248-9_34 [Accessed 2023].
5. Shiv Ram Dubey, S. K. S. B. B. C., 2022. Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. [Online] Available at: https://arxiv.org/pdf/2109.14545.pdf [Accessed 2023].
6. Nikolaos-Ioannis Galanis, P. V. K.-G. M. G. A. P., 2022. Convolutional Neural Networks: A Roundup and Benchmark of Their Pooling Layer Variants. [Online] Available at: https://www.mdpi.com/1999-4893/15/11/391 [Accessed 2023].
7. Rajendran Nirthika, S. M. A. R. R. W., 2022. Pooling in convolutional neural networks for medical image analysis: a survey and an empirical study. [Online] Available at: https://link.springer.com/article/10.1007/s00521-022-06953-8#Sec32 [Accessed 2023].
8. Afia Zafar, M. A. N. M. N. A. A. S. R. A. A. A. K. D. S. A., 2022. A Comparison of Pooling Methods for Convolutional Neural Networks. [Online] Available at: https://www.mdpi.com/2076-3417/12/17/8643 [Accessed 2023].
9. P Naveen, B. D., 2021. Pre-trained VGG-16 with CNN Architecture to classify X-Rays images into Normal or Pneumonia. [Online] Available at: https://ieeexplore.ieee.org/document/9396997 [Accessed 2023].
10. Sun, B., Tao, X., li, J. & Luo, X., 2020. Research on Palm Vein Recognition Algorithm Based on Improved Convolutional Neural Network. [Online] Available at: https://ieeexplore.ieee.org/document/9289736 [Accessed 2023].
11. Jinchuan Li, Y. L. Z. C., 2020. Segmentation and Attention Network for Complicated X-Ray Images. [Online] Available at: https://ieeexplore.ieee.org/document/9337635 [Accessed 2023].
12. Yona Falinie Gaus, N. B. T. P. B., 2020. On the Use of Deep Learning for the Detection of Firearms in X-ray Baggage Security Imagery. [Online] Available at: https://ieeexplore.ieee.org/document/9032917 [Accessed 2023].
13. Zhijie, Y. et al., 2022. Bactran: A Hardware Batch Normalization Implementation for CNN Training Engine. [Online] Available at: https://ieeexplore.ieee.org/document/9003257 [Accessed 2023].

14. Suzuki, Y. & Ichige, K., 2021. High Accuracy Video Foreground Segmentation Based on Feature Normalization. [Online] Available at: https://ieeexplore.ieee.org/document/9590583 [Accessed 2023].
15. Tingting, C., Jianlin, X. & Huafeng, C., 2021. Improved Convolutional Neural Network Fault Diagnosis Method Based on Dropout. [Online] Available at: https://ieeexplore.ieee.org/document/9356698 [Accessed 2023].
16. Farhadi, Z., Bevrani, H. & Feizi-Derakhshi, M.-R., 2022. Combining Regularization and Dropout Techniques for Deep Convolutional Neural Network. [Online] Available at: https://ieeexplore.ieee.org/document/9986657 [Accessed 2023].
17. Xiaofeng Gu, J. L. X. Z. P. K., 2017. Using checkerboard rendering and deconvolution to eliminate checkerboard artifacts in images generated by neural networks. [Online] Available at: https://ieeexplore.ieee.org/document/8301478 [Accessed 2023].
18. Xiaofeng Gu, J. L. X. Z. P. K., 2017. Using checkerboard rendering and deconvolution to eliminate checkerboard artifacts in images generated by neural networks. [Online] Available at: https://ieeexplore.ieee.org/document/8301478 [Accessed 2023].
19. Augustus Odena, B. V. C. O., 2016. Deconvolution and Checkerboard Artifacts. [Online] Available at: https://distill.pub/2016/deconv-checkerboard/ [Accessed 2023].
20. Kirchhoff, D., 2021. Deconvolutions and what to do about artifacts. [Online] Available at: https://www.neuralception.com/convs-deconvs-artifacts/ [Accessed 2023].
21. Lin, Y. et al., 2021. A High-speed Low-cost CNN Inference Accelerator for Depthwise Separable Convolution. [Online] Available at: https://ieeexplore.ieee.org/document/9332057 [Accessed 2023].
22. WHO - Bernadeta Dadonaite, M. R., 2019. Pneumonia. [Online] Available at: https://ourworldindata.org/pneumonia#number-of-people-dying-from-pneumonia-by-age [Accessed 2023].
23. Daniel Kermany, K. Z. ,. M. G., 2018. Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification (University of California San Diego). [Online] Available at: https://data.mendeley.com/datasets/rscbjbr9sj/2
24. D McHugh, N Buckley, EL Secco, A low-cost visual sensor for gesture recognition via AI CNNS, Intelligent Systems Conference (IntelliSys) 2020, Amsterdam, The Netherlands
25. Buckley N, Sherrett L, Secco EL, A CNN sign language recognition system with single & double-handed gestures, IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), 1250-1253, 2021 - 10.1109/COMPSAC51774.2021.00173
26. EL Secco, DD McHugh, N Buckley, A CNN-based Computer Vision Interface for Prosthetics' application, EAI MobiHealth 2021 - 10th EAI International Conference on Wireless Mobile Communication and Healthcare, 41-59, 2022, DOI: 10.1007/978-3-031-06368-8_3