Diseases Classification and Detection Using Jetson Nano Board

Submitted by

Neeraj Namani

ABSTRACT

Lung Disease is any problem in the lungs that prevents the lungs from working properly. Lung diseases are some of the most common medical conditions in the world and it can happen in any part of the system. Lung diseases affects the Air Sacs (Alveoli) that defines about the airways branch into tiny tubes (bronchioles) that end in clusters of air sacs called alveoli. These air sacs make up most of your lung tissues and the diseases affecting your alveoli include Pneumonia, Tuberculosis and Cancer.

Statistics shows that all over the world about 235,760 new cases of lung related diseases had occurred and every year doctors diagnosis lung cancer in almost 2 million people. Early detection of lung related diseases helps in curing them with appropriate medications. Many researchers developed lung diseases classification model with good results. We developed a deep learning classification model for lung diseases using one of the most efficient classification algorithm which is ResNET. The dataset has been imported from Kaggle website which consists of 7135 images of 4 classes related to lung diseases. The Convolutional Neural Network (CNN) for the dataset is performed by using the ImageNet pretrained model called Residual Network (ResNET).

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

Lung diseases refers to disorders that affect the lungs, the organs that allow us to breathe. Lung disease is a major concern for women. The number of U.S. women diagnosed with lung disease is on the rise. More women are also dying from lung disease. Breathing problems caused by lung disease may prevent the body from getting enough oxygen. The examples of lung diseases are Asthma, chronic bronchitis and emphysema, Infections such as influenza and pneumonia, Lung cancer, Sarcoidosis and pulmonary fibrosis.

Respiratory diseases are among the leading causes of death worldwide. Lung infections (mostly pneumonia and tuberculosis), Lung cancer and chronic obstructive pulmonary disease (COPD) together accounted for 9.5 million deaths worldwide. There are several types of lung diseases, Here we mainly focus on few lung diseases like pneumonia, Tuberculosis and Covid 19. Lung cancer growth has turned out to be a standout amongst the most widely recognized reasons for disease in the two people. Countless bite the dust each year because of lung malignancy. The illness has diverse stages whereby it begins from the little tissue and spreads all through the distinctive territories of the lungs by a procedure called metastasis. It is the uncontrolled development of undesirable cells in the lungs. It is assessed that around 12,203 people had lung disease in 2016, 7130 males and 5073 females; passing from lung malignant growth in 2016 were 8839.

There are many different lung diseases, some of which are caused by bacterial, viral or fungal infections. Other lung diseases are associated with environmental factors, including asthma, mesothelioma and lung cancer. Chronic lower respiratory diseases is a set of conditions that includes chronic obstructive pulmonary disease (COPD), emphysema, and chronic bronchitis. Together, chronic lower respiratory diseases are a leading cause of death. Respiratory diseases such as asthma and COPD involve a narrowing or blockage of airways that reduce air flow. In other lung conditions – such as pulmonary fibrosis, a lung tissue scarring that can be caused by different factors, and pneumonia, a bacterial or viral infection in which air sacs fill with fluid – the lungs have reduced ability to hold air. Lung cancer is a disease caused by the abnormal growth of cells.

Though most lung cancer starts in the lungs, some cases start in other parts of the body and spread to the lungs. The two main types of lung cancer are small cell and non-small cell – grow and spread in different ways, and each type may be treated differently. Cigarette

smoking is the overall leading cause of lung cancer. Breathing second hand smoke can also increase a person's chance of developing the disease.

Figure 1.1: Formation of Lung Cancer

1.2 TYPES OF LUNG DISEASES

The most common lung diseases include

1.2.1 Lung cancer

Lung cancer is the cancer that starts in the lungs. The lungs are located in the chest. When you breathe, air goes through your nose, down your windpipe (trachea), and into the lungs, where it flows through tubes called bronchi. Most lung cancer begins in the cells that line these tubes.

There are two main types of lung cancer:

i. Non-small cell lung cancer: It is the most common type of lung cancer.

ii. Small cell lung cancer: It makes up about 20% of all lung cancer cases.

If the cancer started somewhere else in the body types, it is called mixed small cell or large cell cancer. If the cancer started somewhere else in the body and spreads to the lungs, it is called metastatic cancer to the lung.

Lung cancer is the deadliest type of cancer for both men and women. Each year, more people die for lung cancer than of breast, colon, and prostate cancers combined. Lung cancer is more common in older adults. It is rare in people under age 45. Cigarette smoking is the leading cause of lung cancer close to 90%. Certain types of lung cancer can also affect people who have never smoked. Secondhand smoke increases your risk for lung cancer. The following may also increase your risk for lung cancer:

- Exposure to asbestos
- Exposure to cancer causing chemicals such as uranium.

- Exposure to radon gas
- Family history of lung cancer
- High levels of air pollution
- High levels of arsenic in drinking water
- Radiation therapy to the lungs.

Figure 1.2.1 (a): Small Cell Lung Cancer (SCLC)

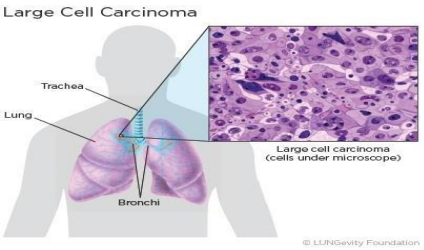

Figure 1.2.1 (b): Large Cell Lung Cancer (LCLC)

1.2.2 Pneumonia

Pneumonia is an infection that inflames the air sacs in one or both lungs. The air sacs may fill with fluid or pus (purulent material), causing cough with phlegm, fever and difficulty breathing. A variety of organisms including bacteria, viruses and fungi can cause pneumonia.

Pneumonia can range in seriousness from mild to life-threatening. It is most serious for infants and young children, people older than 65 and people with health problems or weakened immune systems. The signs and symptoms of pneumonia vary from mild to severe, depending on factors such as the type of germ causing the infection and your age and overall health. Mild signs and symptoms often are similar to those of a cold or flu, but they last longer.

- Chest pain when you breathe or cough
- Confusion or changes in mental awareness
- Cough
- Fever, Sweating
- Lower than normal body temperature
- Nausea
- Shortness of breath

Many germs can cause pneumonia. The most common are bacteria and viruses in the air we breathe. Your body usually prevents these germs from infecting

your lungs. But sometimes these germs can overpower your immune system, even if your health is generally good.

To help prevent pneumonia:

- Get Vaccinated

  Vaccines are available to prevent some types of pneumonia and the flu. Talk with your doctor about getting these shots. The vaccination guidelines have changed over time so make sure to review your vaccination status with your doctor even if you recall previously receiving a pneumonia vaccine.

- Practice good hygiene

  To protect yourself against respiratory infections that sometimes lead to pneumonia, wash your hands regularly or use an alcohol-based hand sanitizer.

- Don't Smoke
- Keep your immune system strong

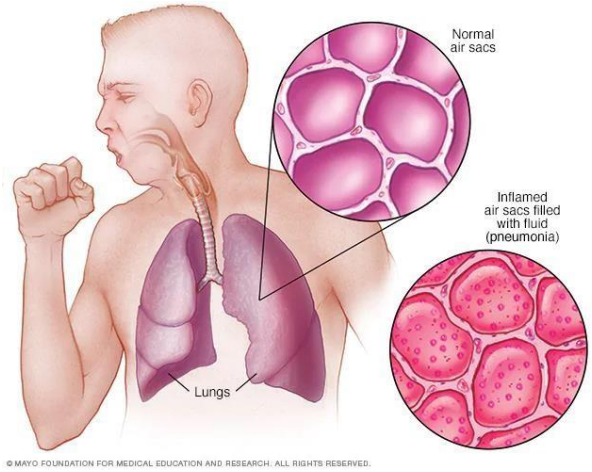

Figure 1.2.2: Pneumonia

1.2.3 Tuberculosis

Tuberculosis (TB) is a bacterial infection spread through inhaling tiny droplets from the coughs or sneezes of an infected person. It mainly affects the lungs, but it can affect any part the body, including the tummy(abdomen), glands, bones and nervous system. TB is a potentially serious condition, but it can be cured if it's treated with the right antibiotics.

Symptoms of TB include

- A persistent cough

- Weight loss
- Night sweats
- High temperature
- Loss of appetite
- Swellings in the neck

TB is a bacterial infection. TB that affects the lungs (pulmonary TB) is the most contagious type, but it usually only spreads after prolonged exposure to someone with the illness. In most healthy people, the body's natural defence against infection and illness (the immune system) kills the bacteria and there are no symptoms. If the immune system fails to kill or contain the infection, it can spread within the lungs or other parts of the body and symptoms will develop within a few weeks or months. This is known as active TB.

Tuberculosis (TB) is caused by a bacterium called Mycobacterium tuberculosis. The bacteria usually attack the lungs, but TB bacteria can attack any part of the body such as the kidney, spine, and brain.

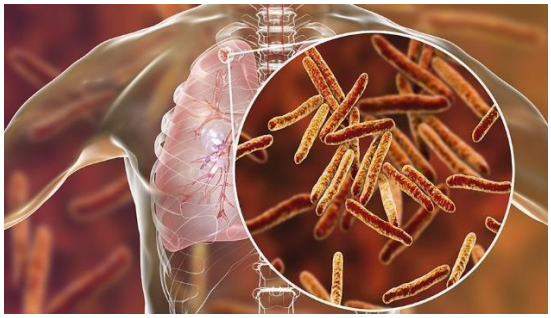

Figure 1.2.3: Tuberculosis

1.2.4 Covid-19

A coronavirus is a kind of common virus that causes an infection in your nose, sinuses, or upper throat. Most coronaviruses aren't dangerous. In early 2020, after a December 2019 outbreak in China, the World Health Organization identified SARS-CoV-2 as a new type of coronavirus. The outbreak quickly spread around the world.

COVID-19 is a disease caused by SARS-CoV-2 that can trigger what doctors call a respiratory tract infection. It can affect your upper respiratory tract (sinuses, nose, and throat) or lower respiratory tract (windpipe and lungs).

It spreads the same way other coronaviruses do, mainly through person-to-person contact. Infections range from mild to deadly. SARS-CoV-2 is one of the seven types of coronavirus, including the ones that cause severe diseases like Middle East respiratory syndrome (MERS) and sudden acute respiratory syndrome (SARS). The other coronaviruses cause most of the colds that affect us during the year but aren't a serious threat for otherwise healthy people.

There's no way to tell how long the pandemic will continue. There are many factors, including the public's efforts to slow the spread, researchers work to learn more about the virus, their search for a treatment, and the success of the vaccines.

The symptoms of Covid are

- Fever
- Coughing
- Shortness of breath
- Trouble breathing
- Fatigue
- Body aches
- Headache and sore throat
- Congestion
- Nausea

The virus can lead to respiratory failure, heart problems, liver problems, septic shock and death. Many COVID-19 complications may be caused by a condition known as cytokine release syndrome or a cytokine storm. This is when an infection triggers your immune system to flood your bloodstream with inflammatory proteins called cytokines. They can kill tissue and damage your organs. In some cases, lung transplants have been needed.

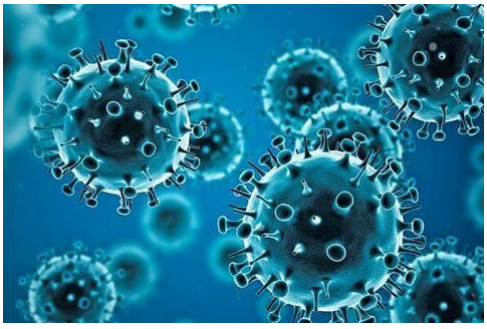

Figure: Formation of COVID-19

1.3 AIM OF THE PROJECT

This project aims to classify the lung diseases using the chest X-ray images, applying convolutional neural networks in an efficient approach. This project aims to evaluate the performance of the ResNet 50 on this classification task using metrics such as precision, Recall and Accuracy.

CHAPTER 2
LITERATURE SURVEY

Lung cancer is proving to be a catastrophic threat to the mankind and is the main cause of deaths among other cancer related casualities. The presence of solitary pulmonary nodules in human lungs in the form of benign or malignant determines the gravity of lung alignment. This survey focuses on different techniques used to detect and classify the lung nodules which in turn will assist the domain experts for better diagnosis. Among many imaging modalities Computed Tomography (CT) being the most sought after because of its high resolution, isotropic acquisition which helps in locating the lung lesions. Since the volume of CT scans are very large, Computer aided detection / Diagnosis (CAD/x) has more advantages in addition to manual interpretation with respect to speed and accuracy. The CNN compared with different techniques to differentiate lung cancer nodules from non – nodules. To reduce / eliminate the false detections, they divided the nodules into four groups according to their size and they have used four different sizes of 3D CNN. They combined all those 4 classifiers to get better results.

For several decades the active area of research has been detection and diagnosis of diseases, several researchers carried out detection and diagnosis of pneumonia, using a multitude of modalities, they proposed an automated algorithm using cough sounds to diagnose pneumonia. For the development of the proposed method, a database of cough sound from 91 patients was made using bedside microphones. Depending on the position, distance between the patient's mouth and the microphone was varying between 40 cm to 70 cm. Mathematical features were collected from the cough sounds, and it was used to train a logistic regression classifier. Finally, a comparative study was performed between the three classification techniques, namely, reasonable clinical diagnosis, the WHO algorithm, and the proposed logistic regression method. The method has achieved a specificity and sensitivity of 75% and 84% respectively.

After the success of Deep learning and Computer vision in image recognition tasks, many researchers have used CXR for pneumonia detection and diagnosis. In 2017, CheXNet was developed, a convolutional neural network with 121 layers, which was trained on Chest X-ray 14 dataset. The performance of the network was assessed

with that of expert radiologist, and it was found that F1 score of CheXNet was 0.435 compared to the radiologist average of 0.387.

Raheel Siddiqi [24] presented a model trained on the dataset provided by Kermanyet al. The model developed was an 18-layer deep sequential convolutional neural network. The proposed model performed the classification task with an accuracy of 94.39%. Also, the model achieved high sensitivity of 0.99. However, the specificity of the model was quite low.

Chakraborthy S.et al. [23] also developed a model by leveraging convolutional neural networks. The network consisted of 17-layers with 3 convolutional layers, which was followed by 5 dense layers and the output layer. For the dimensionality reduction, a series of max-pooling layers have been introduced in the architecture. The model developed was able to achieve an accuracy of 95.62% with recall and precision of 95% and 96% respectively.

In this paper, we investigate the classification performance of the ResNet 50 on the dataset of Chest X-ray images. The dataset consists of 7315 images belonging to three lung diseases. We used ImageNet pretrained model for classification purpose and also calculated Precision, Recall and Accuracy.

CHAPTER 3
METHODOLOGY

The methodology presents techniques that were used in developing a Convolutional Neural Network (ResNet-50) model to detect the various lung diseases. This CNN was trained using lungs chest X-ray images from a publicly available dataset called Kaggle. Here, a CNN model with high accuracy was designed by choosing most appropriate layers.

3.1 SCHEMATIC FLOW:

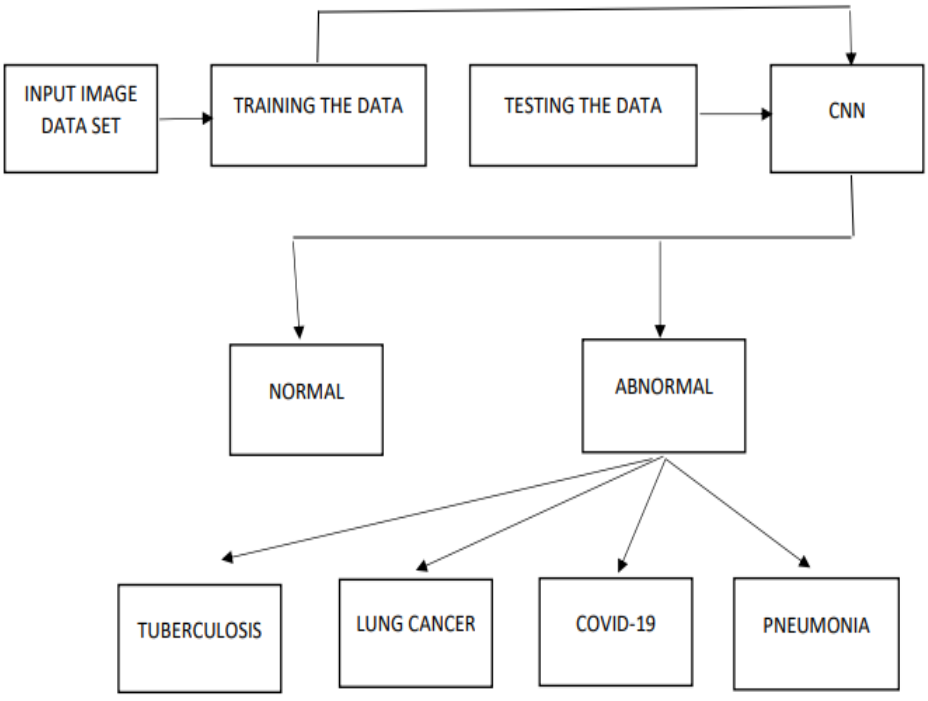

Figure 3.1: Schematic flow of Lung diseases classification using ResNet.

- Input Image: The Chest image from the dataset is the input. The shape of the input image is (224,224,3).

- Balancing the Dataset: Dataset is highly out of balance limit maximum samples

per class to 500 samples to help balance it and used to expand the training dataset in order to improve the performance and ability of the model to generalize.

- Data splitting: The dataset is splitted into three categories i.e. Train Dataset, Test Dataset, Validation Dataset.
  The length of Train Dataset : 6788
  The length of Test Dataset: 491
  The length of Validation Dataset: 36

- Model Creation: A model is a file that has been trained to recognize certain types of patterns. You train a model over a set of data, providing it an algorithm that it can use to reason over and learn from those data. Here the model name is ResNet. The activation functions used are ReLU and Softmax.

- Model Generation: To train a language model that generates text using learning tools, which will be used for the particular task. The total number of epochs used are 35.

- Model Evaluation: The model evaluation is done by calculating Accuracy, Loss, validation accuracy, validation loss.

- Outputs: The classification of various lung diseases are classified by performing various calculations.

- Classification Report: A classification report is a performance evaluation metric in machine learning. It is used to show the precision, recall and support of your trained classification model.

3.2 THE DATASET

This section describes the chest x-ray dataset and its distribution for training, validation and testing.

3.2.1 Chest X-ray Dataset

The standard X-ray image dataset consists of chest x-ray images. The final dataset consists of 7315 images. In the chest X-ray dataset, the classes are covid19,

normal, pneumonia and tuberculosis. It is quite evident that there is a high-class imbalance in the dataset, with more than two-thirds of the imageries belonging to the pneumonia class.

Figure 3.2.1: classification of Dataset

3.2.2 Data Distribution

The Chest X-ray dataset is divided into three parts: training (72 percent), validation (8 percent), and testing (20 percent). The testing set helped assess the effectiveness of our trained models. The dataset does contain any kind of duplicate images in the validation and Testing sets.

TABLE 3.2.2 (a): Data Distribution of 3 Lung classes

Lung classes	Number of images
Covid 19	1530
Pneumonia	4273
Tuberculosis	1512

TABLE 3.2.2 (b): Distribution of each class into three divisions

Lung classes	Training	Validation	Testing
COVID 19	1460	10	60
Pneumonia	3875	8	390
Tuberculosis	1453	18	41
Total	6788	36	491

3.3 RESNET ALGORITHM

Residual Network is a convolutional neural network and is one of the famous deep learning models that was introduced by shaoqing Ren in 2015. The ResNet model is one of the popular and most successful deep learning models so far.

3.3.1 **Residual Blocks**

The problem of training very deep networks has been relieved with the

introduction of these Residual blocks and the ResNet model is made up of these blocks.

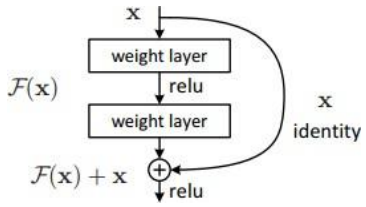

Figure 3.3.1 (a): Deep learning based Residual Network

The very first thing we can notice is that there is a direct connection that skips some layers of the model. This connection is called skip connection and is the heart of residual blocks. The output will not be the same due to this Skip connection. Without the skip connection, input X gets multiplied by the weights of the layer followed by adding a bias term.

The activation function, f() and we get the output as H(x).

$$H(x) = f(wx+b) \text{ or } H(x) = f(x)$$

But the dimension of the input may be varying from that of the output which might happen with a convolutional layer or pooling layers. Hence, this problem can be handled with these two approaches:

- Zero is padded with the skip connection to increase its dimensions.
- 1x1 convolutional layers are added to the input to match the dimensions. In such a case, the output is : $H(x) = f(x) + w1.x$

Here an additional parameter w1 is added whereas no additional parameter is added when using the first approach.

These skip connections technique in ResNet solves the problem of vanishing gradient in deep CNNs by allowing alternate shortcut path for the gradient to flow through. Also, the skip connection helps if any layer hurts the performance of architecture, then it will be skipped by regularization.

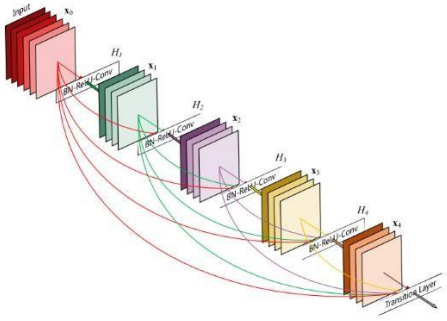

Figure 3.3.1 (b): Architecture of Residual Network

Let's dive deep into the architectural details of all the Residual Net models and find out how they differ from each other.

Generally, the models are made too wide, deep, or with a very high resolution. Increasing these characteristics helps the model initially but it quickly saturates and the model just has more parameters and is therefore not efficient. In Residual Net they are scaled in a more principled way i.e. error rate is around only 3.57%

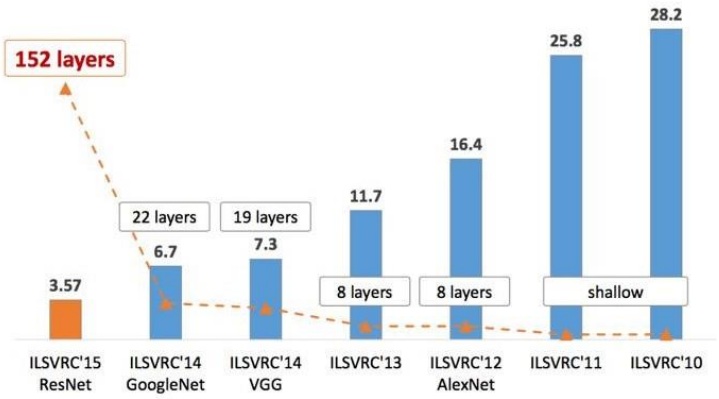

Figure 3.3.1 (c): Top Error-rate for Residual Net – 3.57%

With considerably fewer number of parameters, the family of models are efficient and also provide better results. By observing the above image, it is come to know that these might become the standard pre-trained model. Residual Net contains 8 number of models. We have used Resnet 50 pre trained model architecture which are described briefly.

3.4 RESIDUAL NETWORK (ResNet-50)

ResNet, short for Residual Netwrks is a classic neural network used as a backbone for many computer vision tasks. This model was the winner of ImageNet challenge in 2015. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+ layers successfully. Prior to Resnet

training very deep neural networks was difficult due to the problem of vanishing gradients. However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem – as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient extremely small. As a result, as network goes deeper, its performance gets saturated or even starts degrading rapidly.

ResNet first introduced the concept of skip connection. The diagram below illustrates skip connection. The figure on the left is stacking convolution layers together one after the other. On the right we still stack convolution layers as before but we now also add the original input to the output of the convolution block. This is called skip connection.

Figure 3.4 (a): Skip Connection

Deep Residual Network is almost similar to the networks which have convolution, pooling, activation and fully-connected layers stacked one over the other. The only construction to the simple network to make it a residual network is the identity connection between the layers. The screenshot below shows the residual block used in the network. You can see the identity connection as the curved arrow originationg from the input and sinking to the end of the residual block.

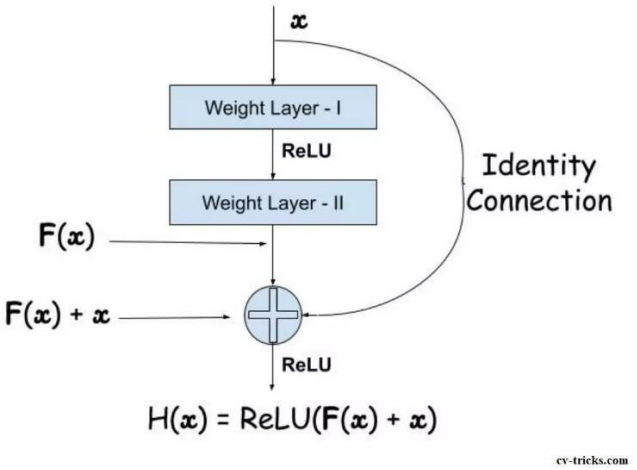

Figure 3.4 (b): Residual Block

The intuition behind the residual block is as we have learned that increasing the number of layers in the network abruptly degrades the accuracy. The deep learning community wanted a deeper network architecture that can either perform well or at least the same as the shallower networks. Now, try to imagine a deep network with convolution, pooling etc layers stacked one over the other. Let us assume that, the actual function that we are trying to learn after every layer is given by Ai where A is the output function of the i-th layer for the given input x. You can refer to the next screenshot to understand the context. You can see that the output functions after every layer are A1,A2,A3….

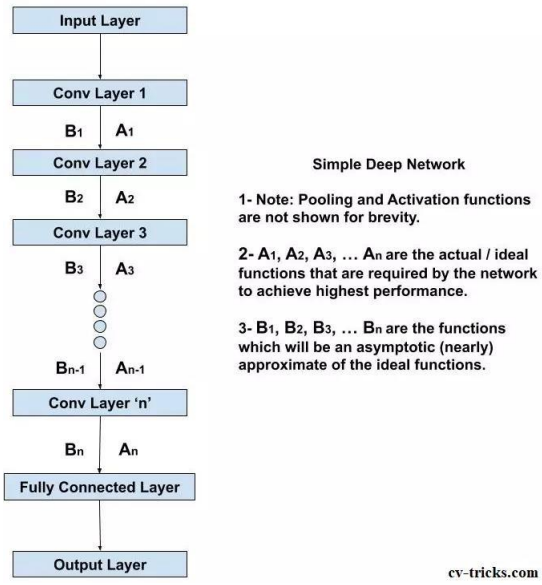

Figure 3.4 (c): Simple Deep Network

In this way of learning, the network is directly trying to learn these output functions i.e. without any extra support. Practically, it is not possible for the network to learn these ideal functions ($A_1, A_2, A_3, \ldots A_n$). The network can only learn the functions say $B_1, B_2, B_3, \ldots B_n$ that are closer to $A_1, A_2, A_3, \ldots A_n$. However, our assumed deep network is so much worse that even it can't learn $B_1, B_2, B_3, \ldots B_n$ which will be closer to $A_1, A_2, A_3, \ldots A_n$ because of the vanishing gradient effect and also due to the unsupported way of training.

The support for the training will be given by the identity mapping addition to the residual output. Firstly, let us see what is the meaning of identity mapping? In a nutshell, applying identity mapping to the input will give you the output which is the same as the input ($AI = A$: where A is input matrix and I is Identity Mapping).

Traditional networks such as AlexNet, VGG-16, etc try to learn the $A_1, A_2, A_3, \ldots A_n$ directly as shown in the diagram of the simple deep network. In the forward pass, the input (image) is passed to the network to get the output. The error is calculated and gradients are determined and backpropagation helps the network to approximate the functions $A_1, A_2, A_3, \ldots A_n$ in the form of $B_1, B_2, B_3, \ldots B_n$. The creators of ResNet thought that:

"If the multiple non-linear layers can asymptotically approximate complicated functions, then it is equivalent to hypothesize that they can asymptotically approximate the Residual function".

3.4.1 **Residual Function**

The Residual function (also known as residual mapping) is the difference between the input and output of the residual block under question. In other words, residual mapping is the value that will be added to the input to approximate the final function of the block. You can also assume that the residual mapping is the amount of error which can be added to input so as to reach the final destination i.e. to approximate the final function of the block. You can visualize the Residual Mapping as shown in the next figure. You can see that the Residual Mapping is acting as a bridge between the input and output of the block. Note that the weight layers and activation function are not shown in the diagram but they are actually present in the network.

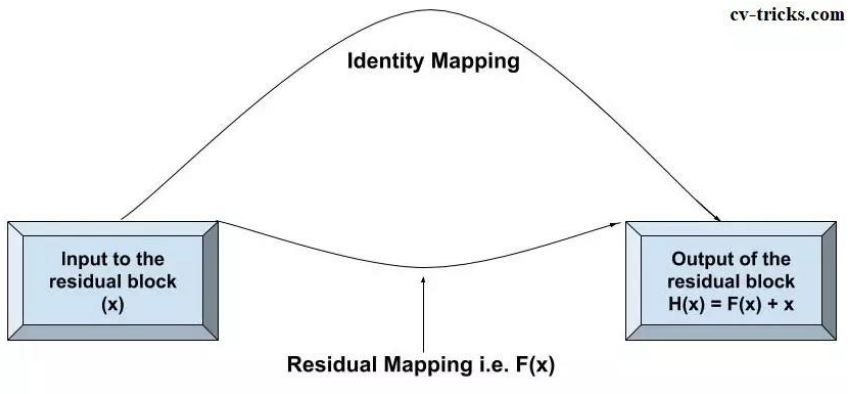

Figure 3.4.1 (a): Residual Function

During the time of backpropagation, there are two pathways for the gradients to transit back to the input layer while traversing a residual block. In the next diagram, you can see that there are two pathways:

- Pathway-1 is the identity mapping way
- Pathway-2 is the residual mapping way

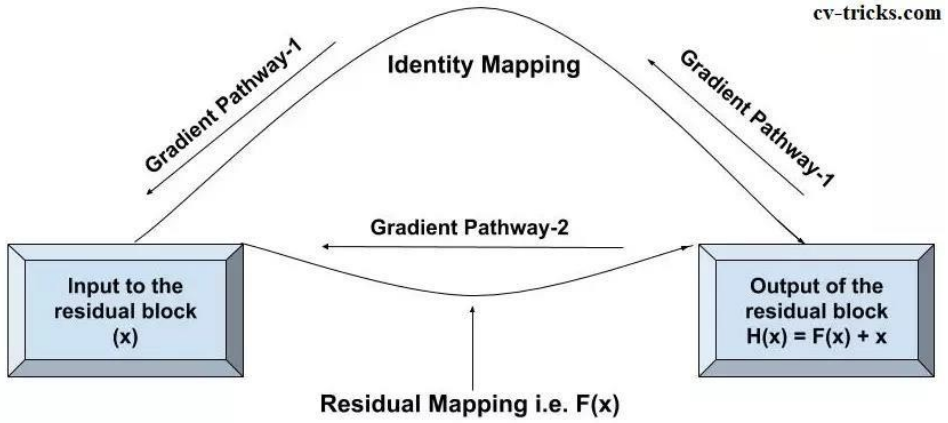

Figure 3.4.1 (b): Gradient Pathways in ResNet

We have already discussed the vanishing gradients problem in the simple deep network. Now we will try to explain the solution for the same keeping the identity connection in mind. Firstly, we will see how to represent the residual block ***F(x)*** mathematically?

$$y = F(x, \{W_i\}) + x$$

where ***y*** is the output function, ***x*** is the input to the residual block and ***F(x, {Wi})*** is the residual block. Note that the residual block contains weight layers which are represented

as ***Wi*** where **1 ≤ i ≤ number of layers in a residual block**. Also, the term ***F(x, {Wi})*** for 2 weight layers in a residual block can be simplified and can be written as follows:

$$F(x, \{W_i\}) = W_2 \sigma(W_1 x)$$

where σ is the ReLU activation function and the second non-linearity is added after the addition with identity mapping i.e. $H(x) = \sigma(y)$.

When the computed gradients pass from the ***Gradient Pathway-2,*** two weight layers are encountered which are W_1 and W_2 in our residual function ***F(x).*** The weights or the kernels in the weight layers W_1 and W_2 are updated and new gradient values are calculated. In the case of initial layers, the newly computed values will either become small or eventually vanish. To save the gradient values from vanishing, the shortcut connection (identity mapping) will come into the picture. The gradients can directly pass through the ***Gradient Pathway-1*** shown in the previous diagram. In ***Gradient Pathway-1,*** the gradients don't have to encounter any weight layer, hence, there won't be any change in the value of computed gradients. The residual block will be skipped at once and the gradients can reach the initial layers which will help them to learn the correct weights. Also, ResNet version 1 has ReLU function after the addition operation, therefore, gradient values will be changed as soon as they are getting inside the residual block.

3.5 INNER ARCHITECTURE OF RESNET

The architecture of ResNet50 has four stages as shown in the diagram below**.** The network can take the input image having height, width as multiples of 32 and 3 as channel width. For the sake of explanation, we will consider the input size as 224 x 224 x 3. Every ResNet architecture performs the initial convolution and max-pooling using 7×7 and 3×3 kernel sizes respectively. Afterward, Stage 1 of the network starts and it has 3 Residual blocks containing 3 layers each. The size of kernels used to perform the convolution operation in all 3 layers of the block of stage 1 are 64, 64 and 128 respectively. The curved arrows refer to the identity connection. The dashed connected arrow represents that the convolution operation in the Residual Block is performed with stride 2, hence, the size of input will be reduced to half in terms of height and width but the channel width will be doubled. As we progress from one stage to another, the channel width is doubled and the size of the input is reduced to half.

For deeper networks like ResNet50, ResNet152, etc, bottleneck design is used. For each residual function ***F,*** 3 layers are stacked one over the other. The three layers are **1×1, 3×3, 1×1** convolutions. The **1×1** convolution layers are responsible for reducing and then

restoring the dimensions. The **3×3** layer is left as a bottleneck with smaller input/output dimensions.

Finally, the network has an Average Pooling layer followed by a fully connected layer having 1000 neurons (ImageNet class output).

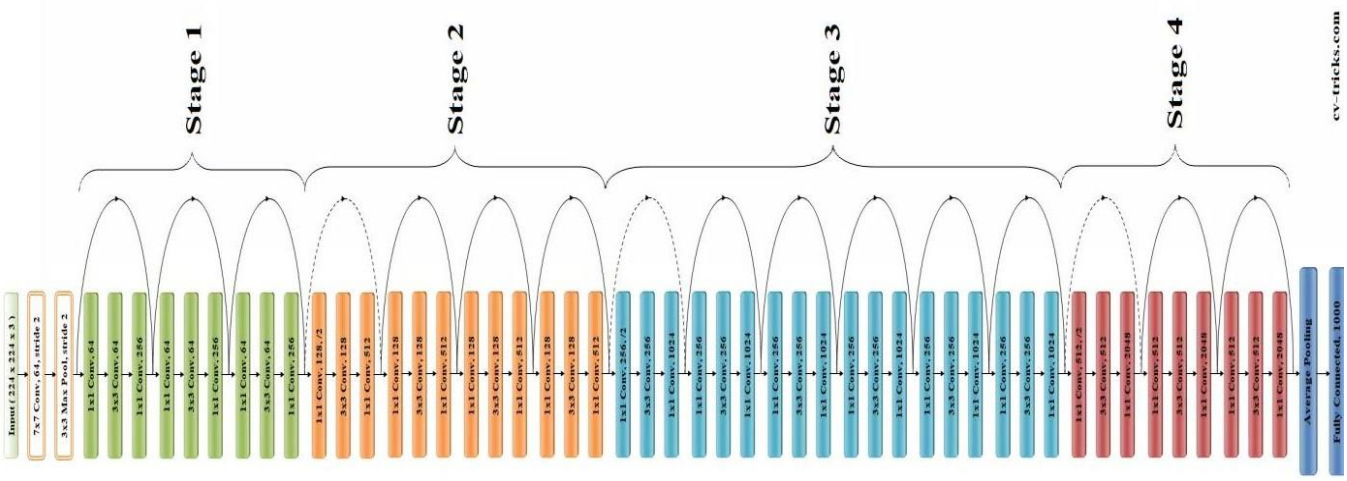

Figure 3.5 (a): Architecture of ResNet50

- Normalization: Normalization is a pre-processing technique used to standardize data. In other words, having different sources of data inside the same range and not normalizing the data before training can cause problems in our network, making it drastically harder to train and decrease its learning speed.
- Conv 2D: This layer creates a convolution kernel that is converted with the layer input to produce a tensor of outputs. If use bias is True, a bias vector is created and added to the outputs.
- Batch Normalization: Batch Norm is a normalization technique done between the layers of a neural network instead of in the raw data. It is done along mini-batches instead of the full data set. It serves to speed up training and use higher learning rates, making learning easier.
- Activation: The activation function is a node that is put at the end of or in between Neural networks. They help to decide if the neuron would fire or not. We have different types of activation functions. The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning.
- Max pooling: Max pooling is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map. The results are down sampled or pooled

feature maps that highlight the most present feature in the patch, not the average presence of the feature in the case of average pooling.

- Average pooling: Average pooling involves calculating the average for each patch of the feature map. This means that each 2x2 square of the feature map us down sampled to the average value in the square.
- Fully connected layer: Residual network has one FC layer with 1000 neurons which again outputs the class probabilities. FC layers are simply, feed forward neural networks. Fully connected layers form the last few layers in the network. The input to the fully connected layer is the output from the final pooling or convolutional layer, which is flattened and then fed into the fully connected layer.

  Let's go layer by layer about the inner architecture of the Residual Network

  Convolution 1:

  The first step on the Resnet before entering the common layer behaviour is a block called here Conv1 consisting on a convolution + batch normalization + max pooling operation.

  So, first there is a convolution operation. In the figure we can see that they use a kernel size of 7, and a feature map size of 64. You need to infer that they have padded with zeroes 3 times on each dimension – and check it on the PyTorch documentation. Taken this into account, it can be seen in the Figure that the ouput size of that operation will be a (112x122) volume. Since each convolution filter (of the 64) is providing one channel in the output volume, we end up with a (112x112x64) output volume.

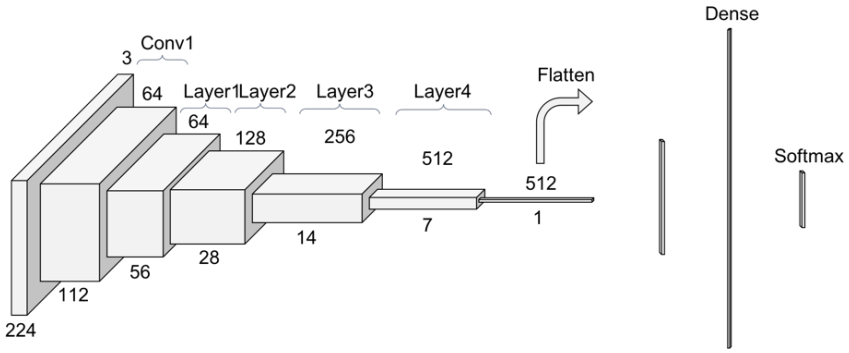

Figure 3.5 (b): Conv1

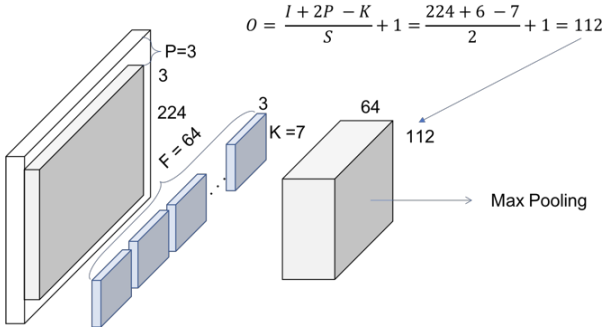

Figure 3.5 (c): Conv1 - Convolution

The next step is the batch normalization, which is an element-wise operation and therefore, it does not change the size of our volume. Finally, we have the (3x3) Max pooling operation with a stride of 2. We can also infer that they first pad the input volume, so the final volume has the desired dimensions.

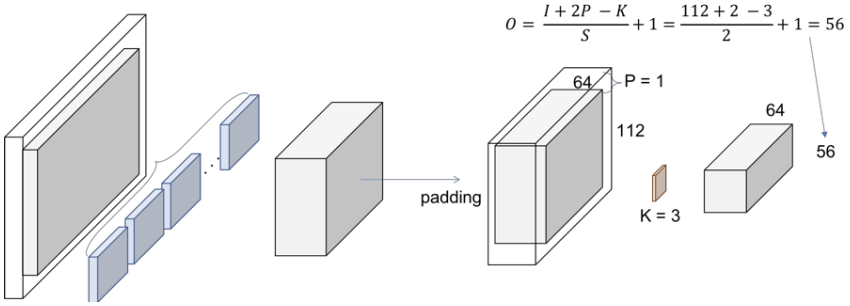

Figure 3.5 (d): Conv1 – Max pooling

Every layer of a ResNet is composed of several blocks. This is because when ResNets go deeper, they normally do it by increasing the number of operations within a block, but the number of total layers remain same. An operation here refers to a convolution a batch normalization and a ReLU activation to an input, except the last operation of a block, that does not have the ReLU.

Therefore, in the PyTorch implementation they distinguish between the blocks that includes 2 operations – Basic block and the blocks that include 3 operations – Bottleneck Bottleneck Block. Note that normally each of these operations is called layer, but we are using layer already for a group of blocks. We are facing a basic one now. The input volume is the last output volume from Conv1. Let's see Figure to figure out what's happening inside this block.

Block 1

1 Convolution

We are replicating the simplified operation for every layer on the paper.

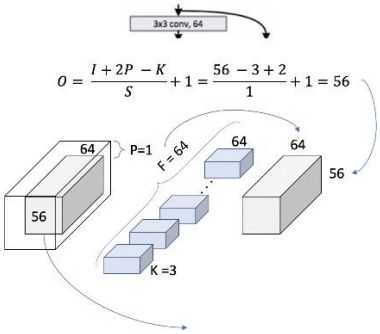

Figure 3.5 (e): Layer1, block1, operation1

We can double check now in the table from the paper we are using [3x3,64] kernel and the output size is [56x56]. We can see how, as we mentioned previously, the size of the volume does not change within a block. This is because a padding = 1 is used and a stride of also 1. Let's see how this extends to an entire block, to cover the 2 [3x3,64] that appears in the table.

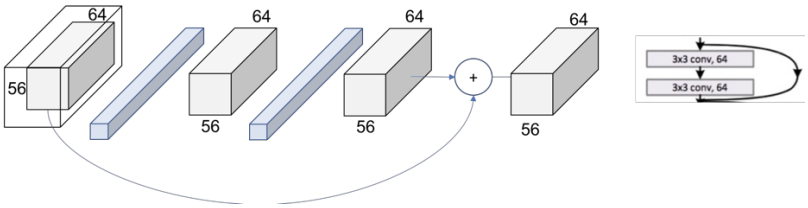

Figure 3.5 (f): Layer 1, Block 1

The same procedure can be expanded to the entire layer then as in Figure. Now, we can completely read the whole cell of the table (just recap we are in the 50 layers ResNet at Conv2_x layer.

We can see how we have the [3x3,64]x3 times within the layer.

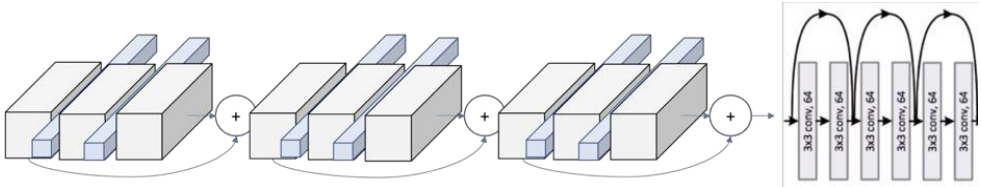

Figure 3.5 (g): Layer 1

Patterns

The next step is to escalate from the entire block to the entire layer. In the Figure 1 we can see how the layers are differentiable by colors. However, if we look at the first operation of each layer, we see that the stride used at that first one is 2, instead of 1 like for the rest of them.

This means that the down sampling of the volume though the network is achieved by increasing the stride instead of a pooling operation like normally CNNs do. In fact, only one max pooling operation is performed in our Conv1 layer, and one average pooling layer at the end of the ResNet, right before the fully connected dense layer in Figure 1.

We can also see another repeating pattern over the layers of the ResNet, the dot layer representing the change of the dimensionality. This agrees with what we just said. The first operation of each layer is reducing the dimension, so we also need to resize the volume that goes through the skip connection, so we could add them like we did in Figure 7.

This difference on the skip connections are the so called in the paper as Identity Shortcut and Projection Shortcut. The identity shortcut is the one we have already discussed, simply bypassing the input volume to the addition operator. The projection shortcut performs a convolution operation to ensure the volumes at this addition operationare the same size. From the paper we can see that there are 2 options for matching the outputsize. Either padding the input volume or perform 1x1 convolutions.

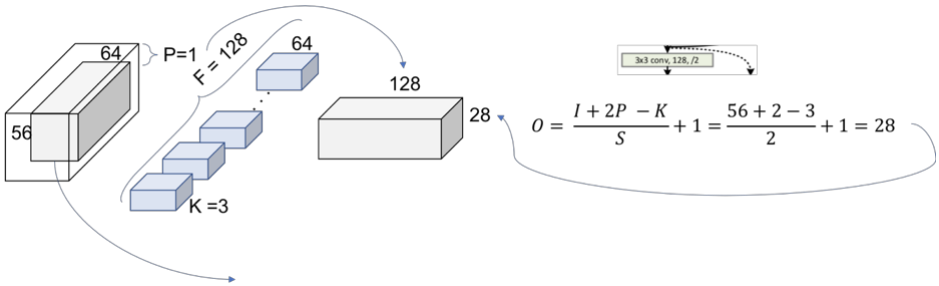

Figure 3.5 (h): Layer 2, Block 1, Operation 1

Figure 9 represents this down sampling performed by increasing the stride to 2. The number of filters is duplicated in an attempt to preserve the time complexity for every operation (56*64 = 28*128). Also, note that now the addition operation cannot be performed since

the volume got modified. In the shortcut we need to apply one of our down sampling strategies. The 1x1 convolution approach is shown in Figure 10.

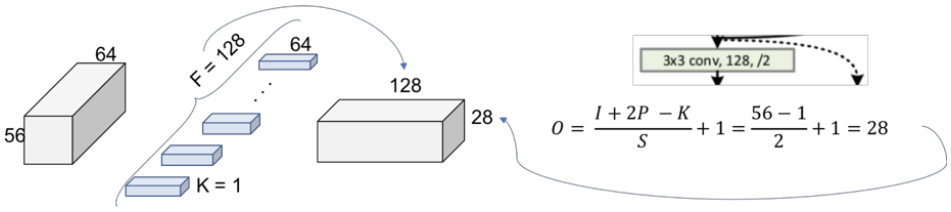

Figure 3.5 (i): Projection Shortcut

The final picture looks then like in Figure 11 where now the 2 output volumes of each thread has the same size and can be added.

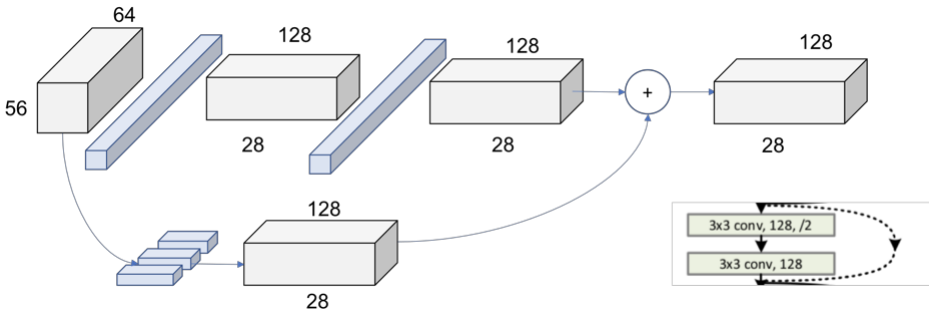

Figure 3.5 (j): Layer 2, Block 1

In Figure 12 we can see the global picture of the entire second layer. The behavior is exactly the same for the following layers 3 and 4, changing only the dimensions of the incoming volumes.

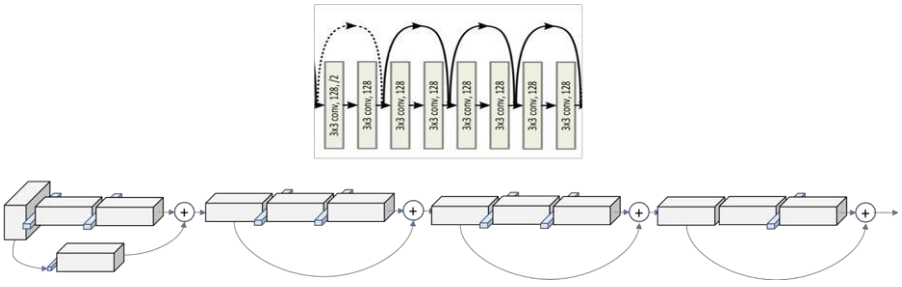

Figure 3.5 (k): Layer 2

3.6 ACTIVATION FUNCTION

Every activation function (or non-linearity) takes a single number and performs a certain fixed mathematical operation on it. There are a number of common activation functions in use with neural networks.

- **ReLU** (Rectified Linear Units)

  Sometimes, a layer is applied immediately after each convolution layer in order to introduce non-linearity to the system. After the convolutional layers, ReLU layers introduce non-linearity to the system. Researchers have recently found that ReLU performs well as other non linear functions (such as the sigmoid function) due to its faster learning ability without making a significant difference in accuracy. Further, it helps to alleviate the vanishing gradient issue, where the higher layers of the network train slowly due to a decrease of gradient exponentially through the layers. Figure 3 shows the image of ReLU activation function. The ReLU layer applies the function f(x) = max (0,x) to all the values in the input volume in order to add non linearity to the model.

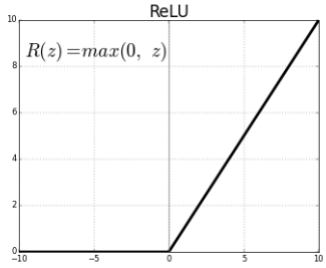

Figure 3.6 (a): Graph for ReLU Function

- **Soft Max**

  Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. The most common use of the softmax function in applied machine learning is in its use as an activation function in a neural network model. Specifically, the network is configured to output N values, one for each class in the classification task, and the softmax function is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one.

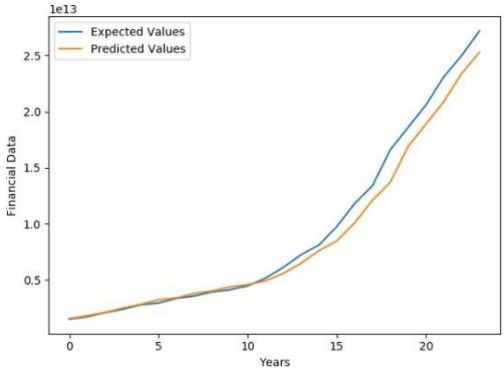

Figure 3.6 (b): Softmax Function

3.7 PERFORMANCE EVALUATION METRICS

This section describes the evaluation metrics used to assess each model's performance, namely Precision, Recall and Accuracy. There are no universal standards for evaluating the performance of any classification model. In literature, we see a standard set of performance measures that depend upon the user's requirement. When classes are highly imbalanced, the metrics Precision, Recall, Accuracy are useful.

- **Accuracy**

  The percentage of correctly predicted image classes to the total number of images is referred to as Accuracy. It is the most straightforward performance metric. Accuracy is, however, only valid when the class distribution is symmetric, that is, when the number of images in each class is roughly the same. For the model of Resnet50, we report the class with the highest probability matches the expected or the actual class.

- **Precision**

  Precision amounts to the fraction of images correctly labelled as belonging the positive class divided by the total number of images labelled as belonging to the positive class by the model.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

- **Recall**

  Recall actually calculates how many of the actual positives our model capture through labelling it as positive (True positive). Applying the same

understanding, we know that recall shall be the model metric we use to select our best model when there is a high cost associated with false Negative.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}$$

CHAPTER 4

SOFTWARE AND HARDWARE REQUIREMENTS

4.1 SOFTWARE REQUIREMENTS

The functional requirements or the overall description documents include the product perspective and features, operating system and operating environment, graphics requirements, design constraints and user documentation.

The appropriation of requirements and implementation constraints gives the general overview of the project in regards to what the areas of strength and deficit are and how to tackle them.

- Programming language: Python
- Platform used: Kaggle
- GPU Accelerator: 128 Core

4.1.1 Modules used in the project

The building of this project has involved the usage of many modules which include:

- Tensor RT
- Tensor Flow
- Keras
- OpenCV
- Numpy
- Pandas
- Torchvision
- Pytorch

Module Description:

Tensor Flow:

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

Keras

Keras is an open source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the Tensor Flow library. Keras contains

numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers and a host of tools to make working with image and text data easier to simplify the coding necessary for writing deep neural network code.

In addition to standard neural networks, keras has support for convolutional and recurrent neural networks. It supports other utility layers like drop out, batch normalization and pooling.

OpenCV:

OpenCV is a great tool for image processing and performing computer vision tasks. It is an open-source library that can be used to perform tasks like face detection, objection detection tracking, landmark detection, and much more. It supports multiple languages including python, Java, C++.

Numpy:

Numpy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with python. It contains various features including these important ones:

- A powerful N-dimensional array object
- Sophisticated (Broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform and random number capabilities. Besides its obvious scientific uses, Numpy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined using Numpy which allows Numpy to seamlessly and speedily integrate with a wide variety of databases.

Pandas:

Pandas is an open-source Python library providing high-performance data manipulation and analysis tool using its powerful data structures. Python was majorly used for data munging and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data load, prepare, manipulate, model and analyze. Python with pandas is used in a wide range of fields including academic and commercial domains including finance, economics, statistics and analytics etc.

Pytorch:

PyTorch is an open source machine learning (ML) framework based on the Python programming language and the torch library. It is one of the preferred platforms for deep learning research. The framework is built to speed up the process between research prototyping and deployment. pyTorch is similar to Numpy and computes using tensors that are accelerated by graphics processing units (GPU). Tensors are arrays, a type of multidimensional data structure, that can be operated on and manipulated with APIs. The pyTorch framework supports over 200 different mathematical operations.

Torch Vision

Torchvision is a library for computer vision that goes hand in hand with PyTorch. It has utilities for efficient image and video transformations, some commonly used pre-trained models and some datasets (Torchvision does not come bundled with PyTorch, you will have to install it separately.)

Tensor RT

Torch-TensorRT is an integration for pytorch that leverages inference optimizations of TensorRT on NVIDIA GPUs with just one line of code, it provides a simple API that gives upto 6x performance speedup on NVIDIA GPUs.

This integration takes advantage of TensorRT optimizations, such as FP16 and INT8 reduced precision, while offering a fallback to native PyTorch when TensorRT does not support the model subgraphs.

4.2 HARDWARE REQUIREMENTS

4.2.1 Jetson Nano Board:

The Jetson Nano Developer Kit is a small, powerful computer that lets you run multiple neural network in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts.

Jetson Nano delivers 472 GFLOPS for running modern AI algoritms fast, with a quad-core 64-bit ARM CPU, a 128-core integrated NVIDIA GPU, as well as 2GB LPDDR4 memory. It runs multiple neural networks in parallel and processes several high-resolution sensors simultaneously.

Jetson Nano is also supported by NVIDIA JetPack, which includes a board support package (BSP), CUDA, cuDNN, and TensorRT software libraries for deep learning, computer vision, GPU computing, multimedia processing, and much more. The SDK also includes the ability to natively install popular

open source Machine learning (ML) frameworks such as TensorFlow, PyTorch, Caffe/Caffe2, Keras, and MXNet, enables the developers to integrate their favourite AI model / AI framework into products fast and easily.

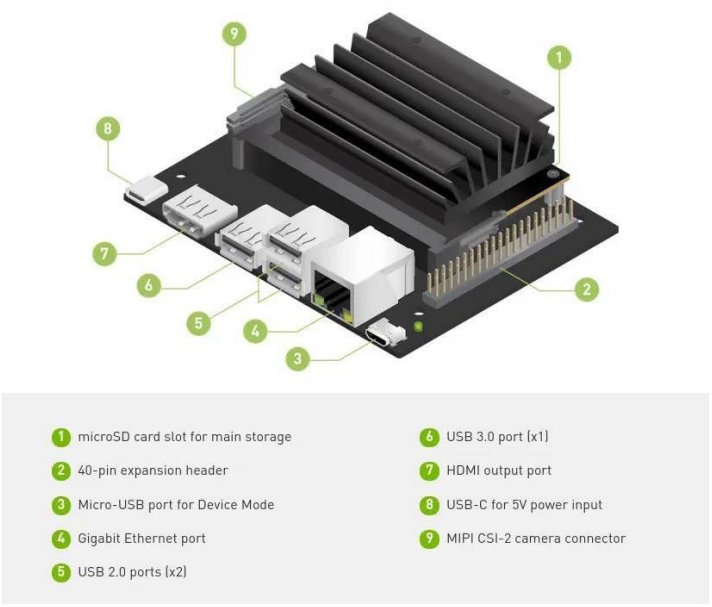

Figure 4.2: Jetson Nano Board

4.2.2 Logitech Camera:

The Logitech C20 HD 720p camera is used to capture the chest X-Ray images from a browser or a hard image. In this process, the camera after capturing the images it will able to detect the type of lung disease and also it displays the accuracy which is to be obtained – Live Detection.

4.3 IMPLEMENTATION

Gather all the Accessories like hardware components such as

- Jetson Nano Board
- Power Supply (5v)
- HDMI to HDMI cable
- Mouse and Keyboard (USB Type)
- SD Card – 64GB recommended
- Ethernet Connection
- Webcam 1080p HD

4.3.1 Setup and installation:

Connect the Sd card to your personal computer and download the SD card image for the 2GB kit along with SD card Formatter and install them. After formatting the SD card,

Figure 4.3.1 (a): 64GB Memory Card

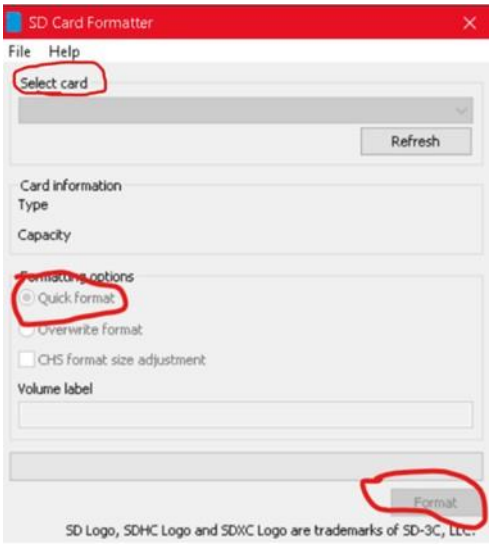

Figure 4.3.1 (b): Formatting SD card

Download and install the ETCHER and launch it by selecting the source as downloaded image and also select the target device for installation of Memory card and flash it. The process will take around 10mins.

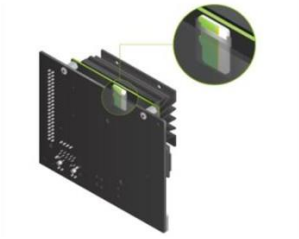

Figure 4.3.1 (c): Memory card slot location

Now, Insert the 64 GB SD card into the kit and attach the accessories like the Power supply, HDMI to HDMI cable and ethernet connection with the appropriate slots of kit as shown in the figure.

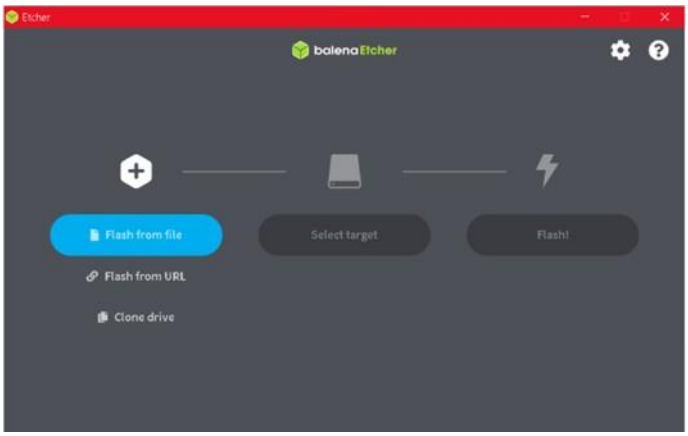

Figure 4.3.1 (d): Flashing the Memory Card

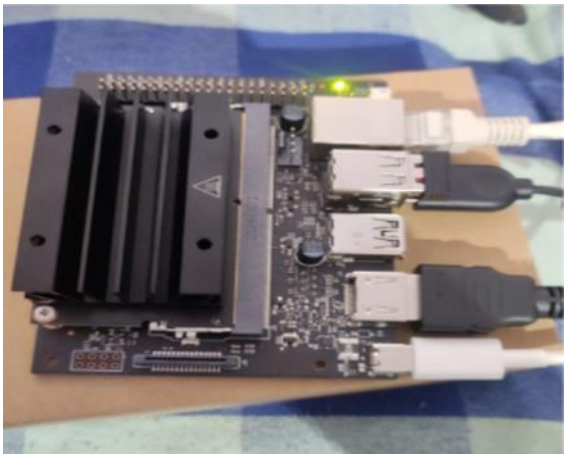

Figure 4.3.1 (e): Connections to the Jetson board

Turn on the power supply and wait for the system for the booting process, When you boot for the first time, the developer kit will take you through some initial setup, including

- Review and accept NVIDIA Jetson Software EULA.
- Select system language, Keyboard layout and time zone.
- Create username, password and computer name.
- Optionally configure wireless networking.
- Select APP partition size. It is recommended to use the max size suggested.

- Create a swap file. It is recommended to create a swap file.

After the setting up of the kit process, The welcome window of the kit as shown in the figure can be seen while launching the kit.

Figure 4.3.1 (f): Welcome Kit

4.3.2 Source:

We will be using Jetson inference for the performance of the project. In order to download jetson inference, we use docker container which contains bash files.

- Open terminal and type the following command
  git clone --recursive https://github.com/dusty-nv/jetson-inference

Wait until it downloads all the required files along with the docker files. It usually takes 10-15 mins on slow connection.

- Change the directory to jetson inference using below command
  cd jetson-inference
- Run the docker container command, it will ask for the password of your kit. Enter the password (in linux the password you tyoe is not shown) and press Enter key.
  docker/run.sh
- It may take time in First time running the docker command and ask for models that you want to download. Download default models and press ok.

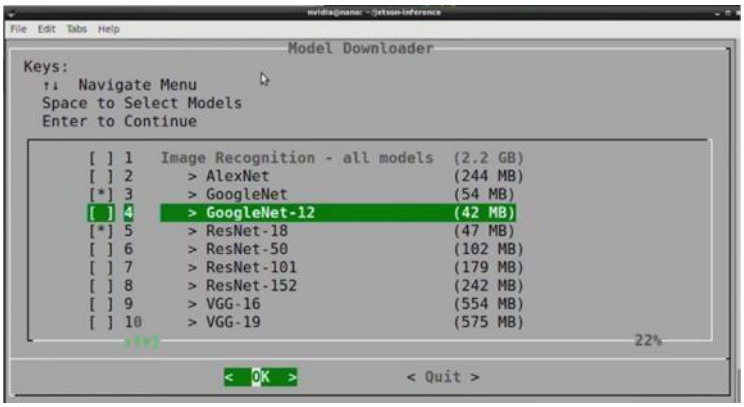

Figure 4.3.2 (a): Image Recognition models

The dataset is taken from Kaggle website and we used the following dataset to train the model using the pre-trained model called Residual Network which is 50 layers deep. We have collected chest X-ray images dataset from the below link.

https://www.kaggle.com/jtiptj/chest-xray-pneumoniacovid19tuberculosis/download

The dataset consist of Chest X-ray images of covid-19 patient, Pneumonia patient and Tuberculosis patient along with Normal X-ray images. The dataset contains a total of 7315 images with different classes.

Figure 4.3.2 (b): Dataset (Chest Xray)

- Extract the Dataset to the following location Jetson-inference/python/training/classification/data/
- Include dataset name after '/data/' in the above command. In that folder create a label.txt file and name all the things that your kit gonna detect.

In order to Train the dataset for the classification purpose of different lung diseases we need to enter the following command so that the preferred the Neural Network can be built.

python3 train.py --model-dir=models/xray --batch-size=4 --workers=1 --epoch=35 data/xray

It will take around 8-10 hours for training and kit gets hot so don't touch it. After the completion of training the neural network model, we need to export the onnx file to run the whole process.

python3 onnx_export.py --model-dir=models/xray

After executing this command in the linux terminal, it will extend its support towards the trained the model by providing or exporting its prerequisities to the trained model.

4.3.3 Execute:

In order to run the project after the training process and export the onnx file, an onnx file engine will be generated in the path classification/models/ and make sure that the onnx file has been exported successfully.

After the above process, we need to enter a command to see the performance of our pre-trained model to check the classification along with the accuracy that is obtained.

imagenet --model=models/xray/resnet50.onnx --input_blob=input_0 output_blob=output_0 --labels=data/xray/labels.txt (input location) (output location)

We have used a webcamera for live capture detection, In this step, we need to interface the camera to the system through the linux terminal by running the following command

Command 1: video-viewer /dev/video0

After entering the above command, A new window will pop-up and camera starts running to capture the video screening and when you click on the close button, the camera will shuts down its process.

Command 2:

imagenet --model=models/xray/resnet50.onnx --input_blob=input_0 --output_blob=output_0 --labels=data/xray/labels.txt (input location) (output location)

With this command, a new window will pop up again and we need to capture few hard xray copies so that it classify them into lung diseases with good accuracy.

You can Train the model once again or can be fine-tuned the pre-trained model to increase the accuracy and we can use different ImageNet existing pre-trained models for classification purposes.

CHAPTER 5
RESULTS AND DISCUSSION

5.1 OUTPUT PARAMETERS ON TRAINING AND TESTING SET

For 1 Epoch

Table 5.1: Output Parameters on Train set

Recall	100.00
Accuracy	57.97
Loss	68.63

Table 5.2: Output Parameters on Test set

Recall	100.00
Accuracy	55.84
Loss	69.72

Table 5.1 and 5.2 shows the values of parameters, after 1 epoch. It can be observed, that there is no much difference in the values of parameters on training, testing data sets.

After 35 Epochs

Table 5.3: Output parameters on Train set

Recall	92.03
Accuracy	94.42
Loss	3.18

Table 5.4: Output Parameters on Test set

Recall	92.10
Accuracy	95.58
Loss	3.10

From the above tables, it can be observed the difference in the values of parameters for 1 epoch and after 10 epochs. During training, the fluctuation in the values of parameters has been observed. The accuracy after 35 epochs on training data set is 96.42% , on testing datac set it is 95.58%.

5.2 CLASSIFICATION REPORT

Below is the classification Reports of the ResNet model. This classification Report comprises of precision, Recall and Accuracy with the respective type of disease.

Table 5.2.1: Classification Report of ResNet50 Model

Type of Disease	Precision	Recall	Accuracy
COVID-19	0.88	1.00	0.93
Pneumonia	0.67	0.70	0.74
Tuberculosis	0.94	0.88	0.91

5.3 OUTPUT

As the dataset provided to the pre-trained model for the classification purpose, the input images that taken from the Chest x-ray image dataset has been given to the pre-trained model called ResNet50 and since it is a existing trained model, the accuracy was good.

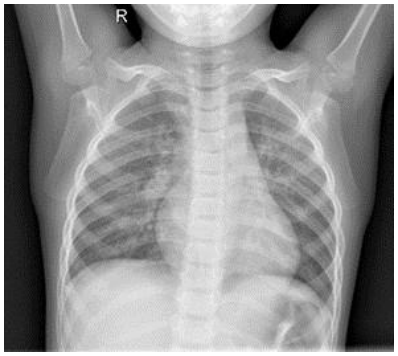

Figure 5.3.1: Normal Chest X-ray

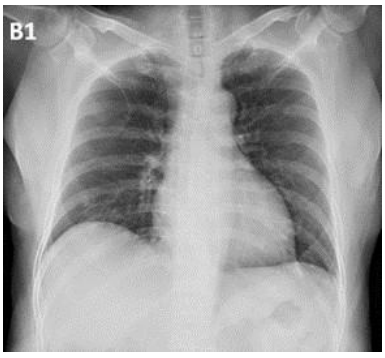

Figure 5.3.2: Abnormal Chest X-ray

As shown in the figure 5.3.1, it is a normal chest X-ray image with no abnormalities means the X-ray is in perfect condition and functioning well. But, in the figure 5.3.2 it is an abnormal chest X-ray image with some kind of abnormality due to some kind of lung malignancy which may lead to lung related diseases.

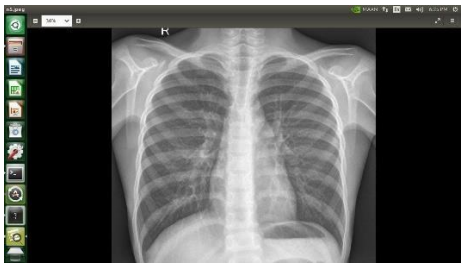

Figure 5.3.3 (a): Input Sample

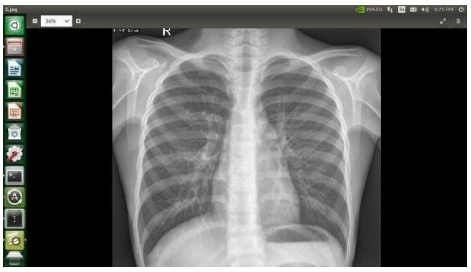

Figure 5.3.3 (b): Normal (Predicted output)

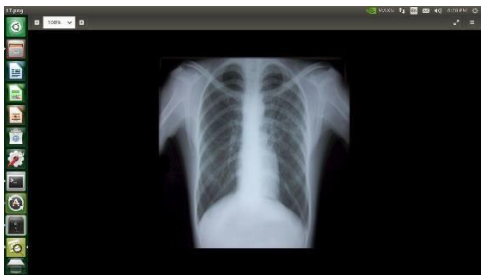

Figure 5.3.3 (c): Input Sample

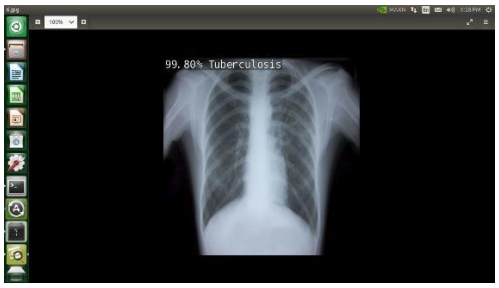

Figure 5.3.3 (d): Tuberculosis (Predicted output)

The above figures from 5.3.3 (a to d) shows us the output figures of the classification purpose for lung diseases. In our pre-trained we have obtained an accuracy of above 90 percent and it can improved by using fine-tuning the pre existing weights if required.

Figure 5.3.4 : Accuracy of Lung diseases from X-ray images

The figure 5.3.4 shows the graph about the accuracy of the CNN model by considering an epoch size of 35 in total so that an accuracy of around 94.5% was obtained. The training

statistics represents the accuracy of our model that is performed on our dataset. By observing the graph, we can see that the accuracy is being increasing for each epoch size and the loss is being decreasing gradually.

CHAPTER 6
CONCLUSION AND FUTURE SCOPE

Lung cancer is one of the most prevalent and severe cancers. This condition is primarily diagnosed by the doctors with appropriate treatments. Due to the fine-grained diversity in the look of its numerous diagnostic categories, lung diseases classification is a tough undertaking. In recent studies, on the other hand, CNNs have outperformed the treatments in lung disease classification. For this effort, we constructed a pre-trained model based on the requirement for the classification purpose. We trained Resnet50 on the Chest X-ray dataset by performing transfer-learning on pre-trained weights of ImageNet. To analyze the model, we evaluated the performance of Resnet50 on this imbalanced lung disease classification problem using measures such as Precision, Recall, Accuracy. In particular, our robust model, the ResNet50 scored an 95 percent accuracy.

As far as we know, this is the first study to examine the ResNet 50 performance on the Chest X-ray dataset as well as the lung disease classification challenge using a hardware GPU called Jetson Nano Board. We tested ResNet classifiers using criteria accounting for the high-class imbalance. Our results demonstrate that more model complexity does not necessarily equal better classification performance. We noticed the best performance with middle-level complexity models such as ResNet 50.

The future scope for this model is to evaluate the classification of lung related diseases by using different ImageNet models and can be used to compare them to look for better performed model and the model can be fine-tuned to improve its accuracy.

REFERENCES

[1] Rahane, W., Dalvi, H., Magar, Y., Kalane, A., & Jondhale, S. (2018, March). Lung cancer detection using image processing and machine learning healthcare. In 2018 International Conference on Current Trends towards Converging Technologies (ICCTCT) (pp. 1-5). IEEE.

[2] Y. Xie, J. Zhang, Y. Xia, M. Fulham, and Y. Zhang, ‘‘Fusing texture, shape and deep model-learned information at decision level for automated classification of lung nodules on chest CT,’’ *Inf. Fusion*, vol. 42, pp. 102–110, Jul. 2018.

[3] H. Xie, D. Yang, N. Sun, Z. Chen, and Y. Zhang, ‘‘Automated pulmonary nodule detection in CT images using deep convolutional neural networks,’’ *Pattern Recognit.*, vol. 85, pp. 109–119, Jan. 2019.

[4] Hripcsak, G., Knirsch, C. A., Jain, N. L., & Pablos-Mendez, A. (1997). Automated tuberculosis detection. Journal of the American Medical Informatics Association, 4(5), 376-381

[5] Anuj Rohilla RH, Mittal A. TB detection in chest radiograph using deep learning architecture. Int. J. Adv. Res. Sci. Eng. 2017;6:1073–1084. [Google Scholar]

[6] Candemir S, et al. Lung segmentation in chest radiographs using anatomical atlases with nonrigid registration. IEEE Trans. Med. Imaging. 2014;33:577–590. doi: 10.1109/TMI.2013.2290491. [PubMed] [CrossRef] [Google Scholar]

[7] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L.: ImageNet: a large-scale hierarchical image database. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009), pp. 248–255 (2009)

[8] Gordienko, Y., et al.: Deep learning with lung segmentation and bone shadow exclusion techniques for chest X-ray analysis of lung cancer. Computing Research Repository, vol. abs/1712.07632 (2017)

[9] Islam, M.T., Aowal, M.A., Minhaz, A.T., Ashraf, K.: Abnormality detection and localization in chest X-rays using deep convolutional neural networks. ArXiv, vol. abs/1705.09850 (2017)

[10] Jaeger S, Candemir S, Antani S, Wang Y-X, Lu P-X, Thoma G. Two public chest X-ray datasets for computer-aided screening of pulmonary diseases. Quant. Imaging Med. Surg. 2014;4:475–477. [PMC free article] [PubMed] [Google Scholar]

[11] Jaeger S, Karargyris A, Candemir S, Folio L, Siegelman J, Callaghan F, Xue Z, Palaniappan K, Singh RK, Antani S, Thoma G, Wang Y, Lu P, McDonald CJ. Automatic

tuberculosis screening using chest radiographs. IEEE Trans. Med. Imaging. 2014;33:233–245. doi: 10.1109/TMI.2013.2284099. [PubMed] [CrossRef] [Google Scholar]
[12] Kang Q, Lao Q, Fevens T, et al. Nuclei segmentation in histopathological images using two-stage learning. In: Shen D, et al., editors. Medical Image Computing and Computer Assisted Intervention – MICCAI 2019; Cham: Springer; 2019. pp. 703–711. [Google Scholar]
[13] Kermany KZD, Goldbaum M. Large dataset of labeled optical coherence tomography (OCT) and chest X-ray images. Cell. 2018;172:1122–1131. doi: 10.1016/j.cell.2018.02.010. [PubMed] [CrossRef] [Google Scholar]
[14] Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems (NIPS 2012), vol. 25, pp. 1097–1105 (2012)
[15] Li F, et al. Computer-aided detection of peripheral lung cancers missed at CT: ROC analyses without and with localization. Radiology. 2005;237(2):684–90. doi: 10.1148/radiol.2372041555. [PubMed] [CrossRef] [Google Scholar]
[16] Lo S-CB, Li H, Wang YJ, Kinnard L, Freedman MT. A multiple circular path convolution neural network system for detection of mammographic masses. IEEE Trans. Med. Imaging. 2002;21:150–158. doi: 10.1109/42.993133. [PubMed] [CrossRef] [Google Scholar]

APPENDIX

CNN CODE (Architecture Model)

```
import argparse
import  os
import random
import shutil
import time
import warnings
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.multiprocessing as mp
import torch.utils.data
import torch.utils.data.distributed
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models

from reshape import reshape_model
model_names = sorted(name for name in models.__dict___
        if name.islower() and not name.startswith("__")
        and callable(models._dict_[name]))
#Parse command line arguments
parser = argparse.ArgumentParser(description = "PyTorch ImageNet Training")
parser.add_argument('data', metavar = 'DIR', help = 'path to dataset')
parser.add_argument('—model-dir', type = str, default = ' ', help = 'path to desired output
directory for saving model' , 'checkpoints(default: current directory)')
parser.add_argument('-a', '--arch', metavar = 'ARCH', default = 'resnet50', choices =
```

model_names, help = ‘model architecture: ‘ + ‘.join(model_names) + ‘ | ‘(default: resnet50)’)
parser.add_argument(‘--resolution’, default = 224, type = int, metavar = ‘N’, help = ‘input NxN image resolution of model (default: 224 x 224)’ ‘note than Inception models should use 299 x 299’)
parser.add_argument(‘--j’, ‘--workers’, default = 2, type = int, metavar = ‘N’, help = ‘number of data loading workers (default: 2)’)
parser.add_argument(‘--epochs’, default = 35, type = int, metavar = ‘N’, help = ‘number of total epochs to run’)
parser.add_argument(‘--start-epoch’, default = 0, type = int, metavar = ‘N’, help = manual epoch number (useful on restarts)’)
parser.add_argument(‘-b’, ‘--batch-size’, default = 8, type = int, metavar = ‘N’, help = ‘mini-batch size (default: 8), ‘this is the total batch size of all GPUs on the current node when using Data Parallel or Distributed Data Parallel’)
parser.add_argument(‘--lr’,’—learning-rate’, default = 0.1, type = float, metavar = ‘LR’, help = ‘initial learning rate’, dest=’lr’)
parser.add_argument(‘—momentum’, default = 0.9, type = float, metavar = ‘M’, help = ‘momentum’)
parser.add_argument(‘--wd’, ‘--weight-decay’, default = 1e-4, type = float, help = ‘momentum’ (default: 1e-4)’, dest = ‘weight_decay’)
parser.add_argument(‘-p’, ‘--print-freq’, default = 10, type = int, metavar = ‘N’, help = ‘print frequency (default: 10)’)
parser.add_argument(‘--resume’, default = ‘ ‘, type = str, metavar = ‘PATH’, help = ‘path to latest checkpoint (default: none)’)
parser.add_argument(‘-e’, ‘--evaluate’, dest = ‘evaluate’, action = ‘store_true’, help = ‘evaluate model on validation set’)
parser.add_argument(‘—pretrained’, dest = ‘pretrained’, action = ‘store_true’, default = True, help = ‘use pre-trained model’)
parser.add_argument(‘—world-size’, default = -1, type = int, help = ‘number of nodes for distributed training’)
parser.add_argument(‘—dist-url’, default = ‘tcp://224.66.41.62:23456’, type = str, help = ‘url used to set up distributed training’)
parser.add_argument(‘--dist-backend’, default = ‘nccl’, type = str, help = ‘distributed backend’)
parser.add_argument(‘--seed’, default = None, type = int, help = ‘seed for initializing training.

')

parser.add_argument('--gpu', default = 0, type = int, help = 'GPU id to use.')

parser.add_argument('--multiprocessing-distributed', action = 'store_true', help = 'Use multi-processing distributed training to launch' 'N processes per node, which has N GPUs. This is the 'fastest way to use PyTorch for either single node or multi node data parallel training')

best_acc1 = 0

initiate worker threads (if using distributed multi – GPU)

```
def main():
    args = parser.parse_args()
    if args.seed is not None:
        random.seed(args.seed)
        torch.manual_seed(args.seed)
        cudnn.deterministic = True
        warnings.warn('You have chosen to seed training. 'This will turn on the CUDNN deterministic setting, which can slow down your training considerably!' 'You may see unexpected behavior when restarting from checkpoints.')
    #if args.gpu is not None:
    # warnings.warn('You have chosen a specific GPU. This will completely disable data parallelism.')

    If args.dist_url == "env://" and args.world_size == -1:
        args.world_size = int(os.environ["WORLD_SIZE"])
        args.distributed = args.world_size > 1 or args.multiprocessing_distributed
    ngpus_per_node = torch.cuda.device_count()
    if args.multiprocessing_distributed:
    # Since we have ngpus_per_node processes per node, the total_world_size
    # needs to be adjusted accordingly args.world_size = ngpus_per_node * args.world_size
    # Use torch.multiprocessing.spawn to launch distributed processes: the
    #main-worker processfunction
    mp.spawn(main_worker, nprocs = ngpus_per_node, args = (ngpus_per_node, args))
    else:
        # Simply call main_worker function
        main_worker(args.gpu, ngpus_per_node, args)
```

```
# Worker thread (per – GPU)
def main_worker(gpu, ngpus_per_node, args):
        global best_acc1
        args.gpu = gpu
        if args.gpu is not None:
                print("Use GPU: {} for training".format(args.gpu))
        if args.distributed:
                if args.dist_url == "env:/ /" and args.rank == -1:
                        args.rank = int(os.environ["RANK"])
                        if args.multiprocessing_distributed:
        # For multiprocessing distributed training, rank needs to be the
        # global rank among all the processes
        args.rank = args.rank * ngpus_per_node + gpu
dist.init_process_group(backend = args.dist_backend, init_method = args.dist_url, world_size
= args.world_size, rank = args.rank)
#data loading code
traindir = os.path.join(args.data, 'train')
valdir = os.path.join(args.data, 'val')
normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                                std = [0.229, 0.224, 0.225])
train_dataset = datasets.ImageFolder(
        traindir,
        transforms.Compose([
                #transorms.Resize(224),
                transforms.RandomResizedCrop(args.resolution),
                transforms.RandomHorizontalFlip(),
                transforms.ToTensor(),
                normalize,
        ]))
num_classes = len(train_dataset.classes)
print('=> dataset classes: '+ str(num_classes) + ' ' + str(train_dataset.classes))
if args.distributed:
        train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
else:
```

```
        train_sampler = None
train_loader = torch.utils.data.DataLoader(
        train_dataset, batch_size = args.batch_size, shuffle = (train_sampler is None),
num_workers = args.workers, pin_memory = True, sampler = train_sampler)
val_loader = torch.utils.data.DataLoader(
        datasets.ImageFolder(valdir, transforms.Compose([
                transforms.Resize(256),
                transforms.CenterCrop(args.resolution),
                transforms.ToTensor(),
                normalize,
        ])),
        batch_size = args.batch_size, shuffle = False,
        num_workers = args.workers, pin_memory = True)
#create or load the model if using pre – trained (the default)
if args.pretrained:
        print("=> using pre-trained model '{}'".format(args.arch))
        model = models._dict_[args.arch](pretrained = True)
else:
        print("=> creating model '{}'".format(args.arch))
        model = models._dict[args.arch]()
#reshape the model for the number of classes in the dataset
model = reshape_model(model, args.arch, num_classes)

#transfer the model to the GPU that it should be run on
if args.distributed:
        #For multiprocessing distributed, Distributed Data Parallel Constructor
        # should always set the single device scoe, otherwise,
        # DistributedDataParallel will use all available devices.
        if args.gpu is not None:
                torch.cuda.set_device(args.gpu)
                model.cuda(args.gpu)
                # when using a single GPU per process and per
        # DistributedDataParallel, we need to divide the batch size
        # Ourselves based on the total number of GPUs we have
```

```
        args.batch_size = int(args.batch_size / ngpus_per_node)
        args.workers – int(args.workers / ngpus_per_node)
        model = torch.nn.parallel.DistributedDataParallel(model, device_ids = [args.gpu]
        else:
                model.cuda()
                # DistributedDataParallel will divide and allocated batch_size to all
                # available GPUs if device_ids are not set
                Model = torch.nn.parallel.DistributedDataParallel(model)
elif args,gpu is not None:
        torch.cuda.set_device(args.gpu)
        model = model.cuda(args.gpu)
else:
        # DataParallel will divide and allocate batch_size to all available GPUs
if args.arch.startswith('alexnet') or args.arch.startswith('vgg'):
        model.featues = torch.nn.DataParallel(model.features)
        model.cuda()
else:
        model = torch.nn.DataParallel(model).cuda()
#define loss function (criterion) and optimizer
criterion = nn.CrossEntropyLoss().cuda(args.gpu)
optimizer = torch.optim.SGD(model.parameters(), args.lr,
                                        momentum = args.momentum,
                                        weight_decay = args.weight_decay)
#optionally resume from a checkpoint
if args.resume:
        if os.path.isfile(args.resume):
                print("=> loading checkpoint '{}' ".format(args.resume))
                checkpoint = torch.load(args.resume)
                args.start_epoch = checkpoint['epoch']
                best_acc1 = checkpoint['best_acc1']
                if args.gpu is not None:
                # best_acc1 may be from a checkpoint from a different GPU
                best_acc1 = best_acc1.to(args.gpu)
        model.load_state_dict(checkpoint['state_dict'])
```

```
        optimizer.load_state_dict(checkpoint['optimizer'])
        print('=> loaded checkpoint '{}' (epoch {})".format(args.resume,
checkpoint['epoch']))
else:
        print("=> no checkpoint found at '{}'".format(args.resume))
cudnn.benchmark = True

#if in evaluation mode, only run validation
if args.evaluate:
        validate(val_loader, model, criterion, num_classes, args)
        return
#train for the specified number of epochs
For epoch in range(args.start_epoch, args.epochs):
        if args.distributed:
                train_sampler.set_epoch(epoch)

        # decay the learning rate
        adjust_learning_rate(optimizer, epoch, args)
        # evaluate on validation set
        acc1 = validate(val_loader, model, criterion, num_classes, args)
        #remember best acc@1 and save checkpoint
        is_best = acc1 > best_acc1
        best_acc1 = max(acc1, best_acc1)

        if not args.multiprocessing_distributed or (args.multiprocessing_distributed and
args.rank % ngpus_per_node ==0):
        save_checkpoint({
                'epoch' : epoch + 1,
                'arch' : args.arch,
                'resolution' : args.resolution,
                'num_classes' : num_classes,
                'state_dict' : model.state_dict(),
                'best_acc1' : best_acc1,
                'optimizer' : optimizer.state_dict(),
```

```
        }, is_best, args)
#train one epoch
def train(train_loader, model, criterion, optimizer, epoch, num_classes, args):
                batch_time = AverageMeter('Time', ':6.3f')
                data_time = AverageMeter('Data', ':6.3f')
                losses = AverageMeter('Loss', ':.4e')
                top1 = AverageMeter('Acc@1', ':6.2f')
                top5 = AverageMeter('Acc@5', ':6.2f')
                progress = ProgressMeter(len(train_loader), [batch_time, data_time, losses,
top1, top5],
                prefix = "Epoch: [{}]".format(epoch))
                #switch to train mode
                model.train()

                # get the start time
                epoch_start = time.time()
                end = epoch_start

                # train over each image batch from the dataset
                For i, (images, target) in enumerate(train_loader):
                        #measure data loading time
                        data_time.update(time.time() – end)
                        if args.gpu is not None:
                                images = images.cuda(args.gpu, non_blocking = True)
                                target = target.cuda(args.gpu, non_blocking = True)
                        # compute output
                output = model(images)
                loss = criterion(output, target)
                # measure accuracy and record loss
                acc1, acc5 = accuracy(output, target, topk = (1, min(5, num_classes)))
                losses.update(loss.item(), images.size(0))
                top1.update(acc1[0], images.size(0))
                top5.update(acc5[0], images.size(0))
                # compute gradient and do SGD step
```

```
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                # measure elapsed time
                batch_time.update(time.time() – end)
                end = time.time()

                if i % args.print_freq == 0:
                        progress.display(i)
        print(“Epoch: [{:d}] completed, elapsed time {:6.3f}seconds”.format(epoch, time.time() – epoch_start))
# measure model performance across the val dataset
def validate(val_loader, model, criterion, num_classes, args):
        batch_time = AverageMeter(‘Loss’, ‘:.4e’)
        top1 = AverageMeter(‘Acc@1’,’:6.2f’)
        top5 = AverageMeter(‘Acc@5’,’:6.2f’)
        progress = ProgressMeter(len(val_loader), [batch_time, losses, top1 top5], prefix = ‘Test: ‘)

# switch to evaluate mode
model.eval()

with torch.no_grad():
        end = time.time()
        for i, (images, target) in enumerate(val_loader):
                        if args.gpu is not None:
                                images = images.cuda(args.gpu, non_blocking = True)
                                target = target.cuda(args.gpu, non_blocking = True)
                #compute output
        output = model(images)
        loss = criterion(output, target)
                # measure accuracy and record loss
        acc1, acc5 = accuracy(output, target, topk = (1, min(5, num_classes)))
```

```
        losses.update(loss.item(), images.size(0))
        top1.update(acc1[0], images.size(0))
        top5.update(acc5[0], images.size(0))
        # measure elapsed time
        batch_time.update(time.time() – end)
        end = time.time()
        if i % args.print_freq == 0:
                progress.display(i)
        #TODO: this should also be done with the ProgressMeter
        Print(‘ * Acc@1 {top1.avg: .3f} Acc@5 {top5.avg:.3f}’.format(top1 = top1, top5 = top5))
                Return top1.avg
# save model checkpoint
def save_checkpoint (state, is_best, args, filename = ‘checkpoint.pth.tar’, best_filename = ‘model_best.pth.tar’):
                “Save a model checkpoint file, along with the best-performing model if applicable”
        # if saving to an output directory make sure it exists
        if args.model_dir:
                model_path = os.path.join(model_path, filename)
                best_filename = os.path.join(model_path, best_filename)
                #save the checkpoint
                torch.save(state, filename)
                # earmark the best checkpoint
                if is_best:
                        shutil.copyfile(filename, best_filename)
                        print(“saved best model to: “ + best_filename)
                        print(“saved checkpoint to: “ + filename)

# statistic averaging
Class AverageMeter(object):
        “Computes and stores the average and current value”
        def __init__(self, name, fmt = ‘:f’);
                self.name = name
```

```
            self.fmt = fmt
            self.reset()
        def reset(self):
            self.val = 0
            self.avg = 0
            self.sum = 0
            self.count = 0
        def update(self, val, n = 1):
            self.val = val
            self.sum += val * n
            self.count += n
            self.avg = self.sum / self.count
        def__str_(self):
            fmt.str = '{name} {val' + self.fmt + '} ({avg' + self.fmt + '})'
        return fmtstr.format(**self.__dict__)

#Progress metering
Class ProgressMeter(object):
        def __init_(self, num_batches, meters, prefix = " "):
            self.batch_fmtstr = self._get_batch_fmtstr(num_batches)
            self.meters = meters
            self.prefix = prefix
        def display(self, batch):
            entries = [self.prefix + self.batch_fmtstr.format(batch)]
            entries += [str(meter) for meter in self.meters]
                print(' '.join(entries))
        def _get_batch_fmtstr(self, num_batches):
            num_digits = len(str(num_batches//1))
            fmt = '{:' + str(num_digits) + 'd}'
            return '[' + fmt + '/' + fmt.format(num_batches) + ']'
# Learning rate decay
def adjust_learning_rate(optimizer, epoch, args):
        """sets the learning rate to the initial LR decayed by 10 every 30 epochs"""
        lr = args.lr * (0.1 ** (epoch // 30))
```

```
        for param_group in optimizer.param_grous:
                param_group[‘lr’] = lr
#compute the accuracy for a given result
def accuracy (output, target, topk = (1,)):
        “””“Computes the accuracy over the k top predictions for the specified values of k”””
        with torch.no_grad():
                maxk = max(topk)
                batch_size = target.size(0)
                pred = output.topk(maxk, 1, True, True)
                pred = pred.t()
                correct = pred.eq(target.view(1, -1).expand_as(pred))
                res = []
                for k in topk:
                        correct_k = correct[:k].reshape(-1).float().sum(0, keepdim = True)
res.append(correct_k.mul_(100.0 / batch_size))
                return res
if__name__== ‘_main_’:
        main()
```