

# **Portable Embedded Chest X-Ray Classifier for Quick Tuberculosis (TB) Detection**

Final Year Project

Report By

**Mohammad Nouman Bin Faheem  
Muhammad Bilal Elahi  
Safwan Ullah Khan**

In Partial Fulfillment

Of the Requirements for the degree

Bachelor of Electrical Engineering

(BEE)

School of Electrical Engineering and Computer  
Science National University of Sciences and Tech-  
nology Islamabad, Pakistan

(2022)

## Declaration

We hereby declare that this project report entitled “**Portable Embedded Chest X-Ray Classifier for Tuberculosis Detection**” submitted to the “**SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES**” is a record of an original work done by us under the guidance of Supervisor “**Dr. USMAN ZABIT**” and that no part has been plagiarized without citations. Also, this project work is submitted in the partial fulfillment of the requirements for the degree of Bachelor of Electrical Engineering.

### Team Members

### Signature

**Muhammad Nouman Bin Faheem**

---

**Muhammad Bilal Elahi**

---

**Safwan Ullah Khan**

---

### Supervisor:

### Signature

Dr. Usman Zabit

---

Date:

---

Place:

---

## **Dedication**

This project is wholeheartedly dedicated to Almighty Allah, our creator. He has been source of our strength throughout this project. We would also like to dedicate it to our beloved parents for their unconditional moral, spiritual, emotional, and financial support.

## **Acknowledgements**

We would like to express our sincere gratitude to our advisor, Dr. Usman Zabit and co-advisor, Dr. Wajahat Hussain for their guidance, enthusiastic support, and insightful comments. We would also like to extend our thanks to the evaluating committee members for their useful critique that helped us move forward in right direction.

## Table of Contents

Declaration .....	ii
Dedication .....	iii
Acknowledgements .....	iv
List Of Figures: .....	vi
Abstract .....	1
Introduction .....	2
Literature Review .....	4
Lightweight Neural Network for COVID-19 Detection from Chest X-Ray[] .....	4
Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN [] .....	5
Problem Definition .....	8
Solution Methodology: .....	9
Solution Breakdown: .....	10
1-Prepping Jetson Nano for Deep Learning: .....	10
2- Testing Jetson Nano's Deep Learning readiness: .....	11
3- Initial TB Classification Model: .....	14
4-Retraining the model: .....	15
5- Implementation of the saved model on Jetson Nano: .....	18
6- Creation of a new, improved and optimized model: .....	20
The model summary: .....	23
7- Comparison of the two Models: .....	25
8- Implementation of the optimized model on Jetson Nano: .....	25
9- Creation of a new model that solves the memory issue: .....	28
10- Implementation of VGG16 model on Jetson Nano: .....	34
11- Testing the performance of the device on real X-Rays: .....	35
Conclusion and Future Applications .....	38
References .....	40

## List Of Figures:

Figure 1: Training/Validation Accuracy .....	5
Figure 2: Comparisons Between Embedded Devices .....	7
Figure 3: NVIDIA Jetson Nano .....	9
Figure 4: Work Flow Diagram.....	10
Figure 5:Python script used to load the saved model and make predictions.....	12
Figure 6:Terminal commands used to call the script .....	13
Figure 7:Output screenshot .....	14
Figure 8: Summary of initial model .....	15
Figure 9: Model Summary .....	16
Figure 10: Training and validation accuracy .....	17
Figure 11: Inference timing .....	18
Figure 12: OOM Killer Error.....	19
Figure 13: No memory left in RAM .....	19
Figure 14:The dependencies used are shown.....	20
Figure 15: Data set.....	21
Figure 16:The model.....	22
Figure 17: The Model Summary .....	23
Figure 18: Loss Function and Optimizer .....	23
Figure 19:Using Image Generator to Train the Model.....	24
Figure 20:Training Parameters.....	24
Figure 21: Accuracy.....	24
Figure 22: Inference Timing .....	25
Figure 23: Comparison of the Two Models .....	25
Figure 24:Linux Terminal Commands to call The Script .....	26
Figure 25: Command Running.....	27
Figure 26: Output (Positive) .....	27
Figure 27: OOM Killer Error.....	28
Figure 28: VGG16 Model.....	29
Figure 29: Packages Imported .....	30
Figure 30: Base Model.....	30
Figure 31:Top Layer .....	31
Figure 32: Loss Function and Optimizer of VGG16 Model .....	31
Figure 33: Model Summary .....	32
Figure 34: Model Summary (Continued).....	33
Figure 35: Training Parameters.....	33
Figure 36: Output (Negative).....	34
Figure 37: Output (Positive) .....	35
Figure 38: Confusion Matrix .....	35
Figure 39: Performance Metrics .....	36
Figure 40: Comparison Of The Three Models.....	37
Figure 41: Future Applications .....	39

## Abstract

Tuberculosis is a communicable lung disease that is designated among the top 10 leading causes of death worldwide. Pakistan is ranked fifth among high-burden countries facing tuberculosis. Accurate and early diagnosis of tuberculosis is important, otherwise, it could be fatal. Manual detection of tuberculosis from chest X-ray by radiologists is subjective to human error, slow and tedious. Computer Aided Detection (CAD) can improve screening efficiency.

In this project, we have detected tuberculosis from chest X-ray images using deep-learning classification techniques implemented on an embedded platform. An already developed neural network that has been trained on a large data set is used. A database of 7000 chest X-ray images from several data bases with 3500 normal and 3500 tuberculosis infected images was used for training this algorithm. The algorithm was further optimized for better compatibility with the device selected.

The device selected for the implementation of the model is Nvidia's Jetson Nano. Nvidia Jetson Nano is one of the most widely used accelerators for the inference phase of machine learning. It can be quickly programmed to accelerate complex machine learning algorithms. With a low power consumption and low weight, it is a perfect fit for weight/power constrained scenario. With our project our focus is portability, so this device is best suited to address the portability and power constraint of our project.

The already trained model had an accuracy of 99.8% but it had a whopping size of 1.6 GB. The size was too big to be implemented on the Nvidia Jetson Nano. When the model was loaded on device, it required much more memory to perform the inference which was not possible to run on a run memory portable device like Jetson Nano. These memory constraints were solved by creating a new model that was inspired by the original model, in which the size was reduced, but at a cost of decreased accuracy.

With our project it will be easier to detect tuberculosis in remote areas where the professional radiologists lack and the disease would be detected without any human error.

## *Chapter 1*

### **Introduction**

Tuberculosis (TB) is a communicable lung disease caused by the bacillus *Mycobacterium tuberculosis*. It is one of the top 10 causes of death worldwide and leading cause from single infectious agent worldwide ranking above human immunodeficiency virus (HIV) / acquired immunodeficiency syndrome (AIDS). According to the latest World Health Organization (WHO) global TB report 2021 [i], an estimated 10 million people fell ill with tuberculosis (TB) worldwide. 5.6 million men, 3.3 million women and 1.1 million children. A total of 1.5 million people died from TB in 2020. In 2020, the 30 high TB burden countries accounted for 86% of new TB cases. Eight countries account for two thirds of the total, with India leading the count, followed by China, Indonesia, the Philippines, Pakistan, Nigeria, Bangladesh and South Africa.[ii]

Early diagnosis and treatment can cure TB. About 85% of people who develop TB disease can be successfully treated with a 6-month drug regimen. Moreover, WHO has recommended CXR as a diagnostic tool for TB detection. While, CXR is commonly used for pulmonary TB detection, it is slow and also prone to error for human subjectivity. Subjective inconsistency can occur due to similar patterns of other CXR related disease. There is also lack of expert radiologists in low resource countries (LRC) especially rural areas which harm efficiency and accuracy of mass screening. In that perspective, low-cost computer-aided solution is vital for mass screening in developing countries. The target of CAD is to further develop proficiency for mass screening an exactness to overcome subjectivity errors among radiologists. This CAD framework would help radiologists in featuring anomalies of CXR and assisting them with making choice in regard to TB identification.

There have been a number of reports that the researchers have proposed different mechanisms to classify TB and non-TB CXR by implementing machine learning based algorithms on embedded devices.

Embedded systems are small factor computers that are used to perform application-based tasks. They are useful in applications with constraints of size, weight powers and cost. Training a machine learning algorithm on dataset of chest X ray and then implementing on a low-cost embedded system would prove to be much more efficient in the identification of Tuberculosis. In the recent covid-19 pandemic, many researchers used CNN implemented on embedded systems to detect COVID-19 from CXRs.

The main objective of this project is to deploy a machine learning based, classification



algorithm on an embedded system, which can assist radiologists in the detection of tuberculosis from CXRs. The major contributions of our work are as follows:

1. Finding an efficient and cost effective portable embedded system, that is suitable to implement the models.
2. Implementing already trained TB detection Machine learning (ML) algorithm on that embedded system.
3. Comparing the results and optimizing the algorithm to overcome memory constraints.
4. Making the embedded device portable and easy to use.

## *Chapter 2*

### **Literature Review**

In this Section, we explore a number of TB diagnosis schemes based on machine-learning and deep learning.

#### **Lightweight Neural Network for COVID-19 Detection from Chest X-Ray<sup>[iii]</sup>**

At the end of 2019, a severe public health threat named coronavirus disease (COVID-19) spread rapidly worldwide. After two years, this coronavirus still spreads at a fast rate. Due to its rapid spread, the immediate and rapid diagnosis of COVID-19 is of utmost importance. In the global fight against this virus, chest X-rays are essential in evaluating infected patients. This study proposes a robust, lightweight network where excellent classification results can diagnose COVID-19 by evaluating chest X-rays. The experimental results showed that the modified architecture of the model proposed achieved extremely high classification performance in terms of accuracy, precision, recall, and f1-score for four classes (COVID-19, normal, viral pneumonia and lung opacity) of 21.165 chest X-ray images, and at the same time meeting real-time constraints, in a low-power embedded system.

Theodora Sanida et al. proposed a modified neural network structure of the MobileNetV2 model, to maximize the learning ability for classification of chest X-rays. The modified version of their architecture requires significant less training time than other existing DL architectures due to the small number of network parameters. The dataset used in these experiments is the COVID-19 Radiography Database with 21.165 chest X-ray images and includes 3.616 COVID-19 positive cases, 10.192 normal, 6.012 lung opacity (non-COVID lung infection) and 1.345 viral pneumonia images.

Argyrios Sideris et al. split the dataset into training 70% (14.813 images), validation 20% (4.232 images), and testing 10% (2.120 images). The training set consisted of 14.813 images, in which there were 2.530 COVID-19 images, 7.134 normal images, 4.208 lung opacity images and 941 viral pneumonia images. The validation set consisted of 4.232 images, in which there were 723 COVID-19 images, 2.038 normal images, 1.202 lung opacity images and 269 viral pneumonia images. The test set consisted of 2.120 images, in which there were 363 COVID-19 images, 1.020 normal images, 602 lung opacity images and 135 viral pneumonia images. The images used for testing were not used in the training process.

The Modified MobileNetV2 architecture achieved 95.80% testing accuracy, while MobileNetV2 standard 90.47% for 4 classes of the public COVID-19 Radiography Database. The performance of modified MobileNetV2 model is much better than the standard MobileNetV2. Additionally, the modified MobileNetV2 model is more stable, while the standard MobileNetV2 shows more oscillation.

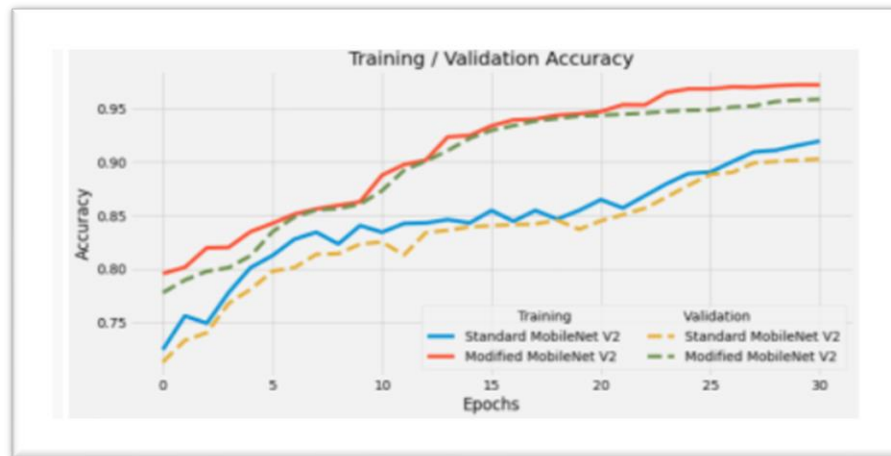


Figure 1: Training/Validation Accuracy

Training and testing are performed on the Nvidia Jetson AGX Xavier with 512 CUDA cores, an 8 core ARM processor, 32 GB RAM, 32 GB eMMC and various other peripherals. The study shows that Nvidia Jetson CPU-GPU heterogeneous architecture achieves high-performance computing and where can be quickly programmed to accelerate complex DL tasks. The embedded GPU has shown great acceleration potential and involves low-power, high accuracy and efficiency in point-of-care medical applications. At the same time, they provide local processing, eliminating security and privacy issues where required in biomedical systems. The device proves to be a portable tool to facilitate medical diagnosis with low-weight, low-cost devices with accuracy, speed and power efficiency.

## Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN [iv]

In this study, performances of single-board computers in NVIDIA Jetson Nano, NVIDIA Jetson TX2 and Raspberry PI4 through CNN algorithm created by using fashion product images dataset were compared. Parameters for performance analysis were defined as consumption (GPU, CPU, RAM, Power), accuracy and cost. Data set divided into parts of 5K, 10K, 20K, 30K and 45K in training and test of the model to expand on the differences of single-board computers. Eventually, performance of the embedded system boards in different data set in CNN algorithm was an-

alyzed. Low power consumption, high accuracy and performance are crucial factors for deep learning applications.

Raspberry Pi is a single credit board-sized PC printed circuit board that can be used for many things your computer does, such as games, word processing, spreadsheets, and also for playing HD video. The Raspberry Pi 4 Model B is the latest in the popular Raspberry Pi computer series. While maintaining backward compatibility and similar power consumption, compared to the previous generation Raspberry Pi 3 Model B+, it offers significant increases in processor speed, multimedia performance, memory and connectivity. Raspberry Pi 4 Model B provides desktop performance comparable to entry-level x86 personal computer (PC) systems.

NVIDIA Jetson Nano is a small, powerful single-board computer that allows parallel operation of multiple neural networks for applications such as image classification, object detection, segmentation, and speech processing. It has a comprehensive development environment (JetPack SDK) and libraries developed for embedded applications, deep learning, IoT, computer vision, graphics, multimedia and more. Using Jetson Nano with a GeForce-enabled graphics processor (GPU) using the same CUDA cores creates a very powerful development environment for applications. In addition, Jetson Nano has a CPU-GPU heterogeneous architecture in which the operating system can be booted by the CPU and can be programmed to speed up complex machine learning tasks of the CUDA capable GPU. Artificial intelligence reveals low power consumption in running algorithms.

The medium sized board of the Nvidia Jetson ecosystem, Jetson TX2, is higher in specifications than Jetson Nano but is much more expensive. It is very useful for computer vision and deep learning.

The comparison of CPU performance, GPU, memory, networking, display, USB, video encode/decode, camera, storage, power and performance of Jetson Nano, Tx2 and Raspberry Pi4 is made in the table below.

	<b>Raspb. PI 4</b>	<b>Jetson Nano</b>	<b>Jetson TX2</b>
<b>Performance</b>	13.5 GFLOPS	472 GFLOPS	1.3 TFLOPS
<b>CPU</b>	Quad-core ARM Cortex-A72 64-bit @ 1.5 GHz	Quad-Core ARM Cortex-A57 64-bit @ 1.42 GHz	Quad-Core ARM Cortex-A57 @ 2GHz + Dual-Core NVIDIA Denver2 @ 2GHz
<b>GPU</b>	Broadcom Video Core VI (32-bit)	NVIDIA Maxwell w/ 128 CUDA cores @ 921 MHz	NVIDIA Pascal 256 CUDA cores @ 1300MHz
<b>Memory</b>	8 GB LPDDR4	4 GB LPDDR4 @ 1600MHz, 25.6 GB/s	8GB 128-bit LPDDR4 @ 1866Mhz, 59.7 GB/s
<b>Networking</b>	Gigabit Ethernet / Wi-Fi 802.11ac	Gigabit Ethernet / M.2 Key E	Gigabit Ethernet, 802.11ac WLAN
<b>Display</b>	2x micro-HDMI (up to 4Kp60)	HDMI 2.0 and eDP 1.4	2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4
<b>USB</b>	2x USB 3.0, 2x USB 2.0	4x USB 3.0, USB 2.0 Micro-B	USB 3.0 + USB 2.0
<b>Other</b>	40-pin GPIO	40-pin GPIO	40-pin GPIO
<b>Video Encode</b>	H264(1080p30)	H.264/H.265 (4Kp30)	H.264/H.265(4Kp60)
<b>Video Decode</b>	H.265(4Kp60), H.264(1080p60)	H.264/H.265 (4Kp60, 2x 4Kp30)	H.264/H.265 (4Kp60)
<b>Camera</b>	MIPI CSI port	MIPI CSI port	MIPI CSI port
<b>Storage</b>	Micro-SD	16 GB eMMC	32GB eMMC
<b>Power under load</b>	2.56W-7.30W	5W-10W	7.5W-15W
<b>Price</b>	\$35	\$89	\$399

Figure 2: Comparisons Between Embedded Devices

The cost of the Jetson TX2, which has high hardware capabilities, is higher than the others. Although Raspberry PI without NVIDIA GPU support is the most cost-effective hardware, it is not the right prefer for deep learning applications. The performance of Jetson Nano against the low cost of the device is suitable for deep learning applications with a much better portability aspect.

## *Chapter 3*

### **Problem Definition**

Pakistan is ranked fifth among the high burdening countries worldwide. Each year 510,000 new TB cases are reported and of which 15,000 are drug resistant TB cases. The major reason for emergence includes: delays in diagnosis, unsupervised, inappropriate and inadequate drug regimens, poor follow-up and lack of social support program.

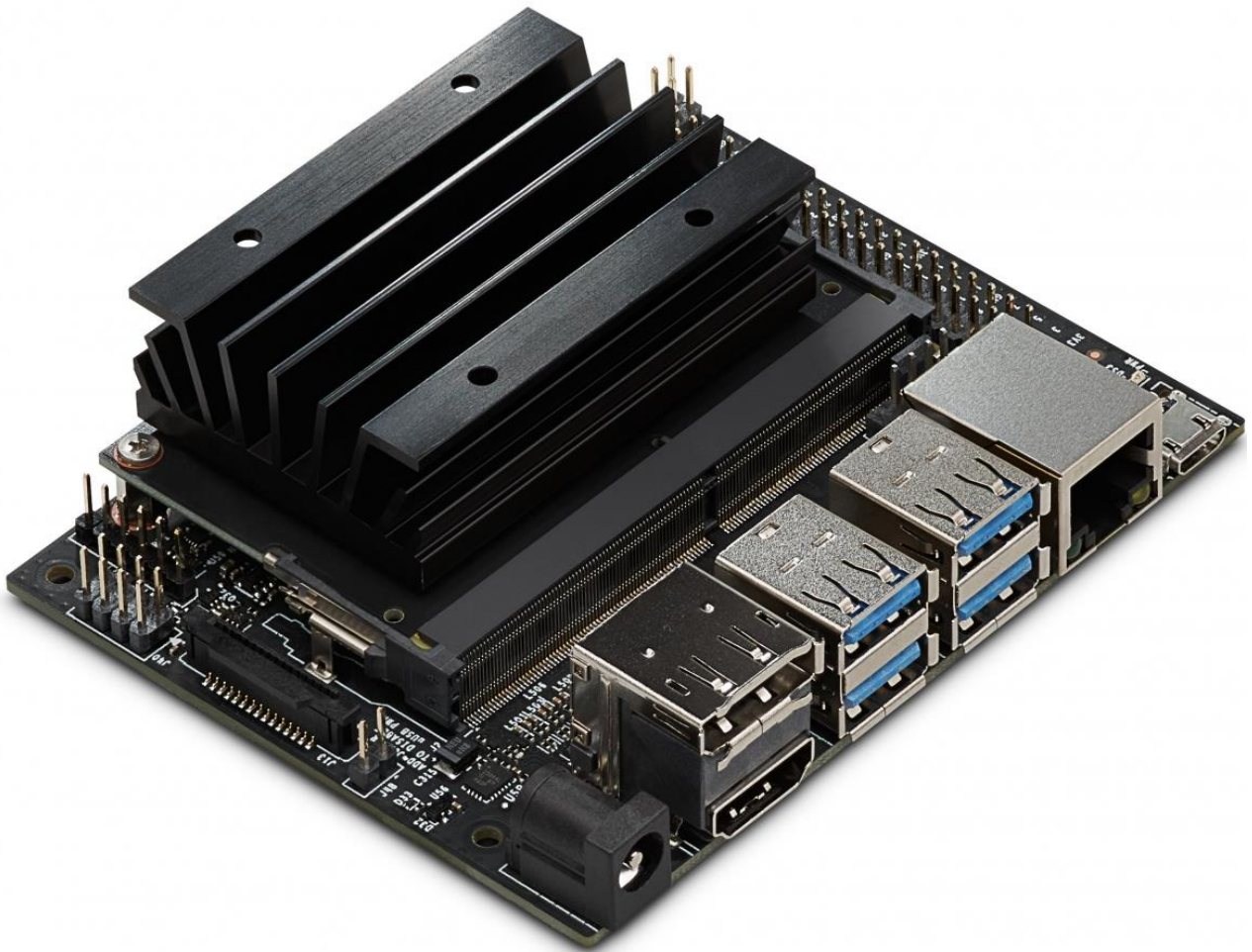
The clinical laboratory tests to detect TB include Mantoux Tuberculin Skin Test (TST) and Interferon-Gamma Release Assays (IGRAs). The problem with these tests are that they are expensive and time consuming and also these technologies are unavailable in rural areas of Pakistan which further leads to undiagnosed TB cases in the country. TB is detected commonly through CXRs and is prone to human error. The low number of expert radiologists in the developing countries further reduces the overall efficiency. In this regard a computer aided portable system is required to be designed which eradicates the above-mentioned problems.

There are a number of lung diseases such as pneumonia, cancer, emphysema and many others require CXRs for initial diagnosis and each disease requires an identification of specific pattern of cavities, air space consolidation, endobronchial spread and pleural effusion. Therefore, the diseases can be easily miss classified by a radiologist. And to overcomes this difficulty, requires a computer aided technique that classifies the CXRs with great precision and accuracy using the concepts of machine learning.

Moreover, it's impossible to detect TB in remote areas of the world where proper hospital infrastructure is scarce. So, the problem required a portable embedded hardware which rectifies all of the above-mentioned problems.

### Solution Methodology:

To Prototype this device an already developed TB detection Neural Network was trained on a large data set. The training phase of the project was done on a general purpose computer using Google Colab servers. The ML framework in use was Tensor flow. Tensor flow provides a feature where after training the neural network one can save the trained model file called an .h5 extension that contains all the input output freezed graphs and weights. This file was used to deploy the model on Jetson Nano.



*Figure 3: NVIDIA Jetson Nano*

Jetson Nano is a SBC (Single board Computer) that has specifically been designed for ML inference on the go. To deploy the Network on Jetson Nano The freezed tensor flow file is loaded into Jetson nano using a python script. The loaded model takes images from the camera in real



time and makes predictions on the X-Ray held in front of it and classifies it as either Normal or Tuberculosis. After Successful deployment of the Network a hand held device was fashioned that can look at an X-Ray and diagnose it for TB in real time.

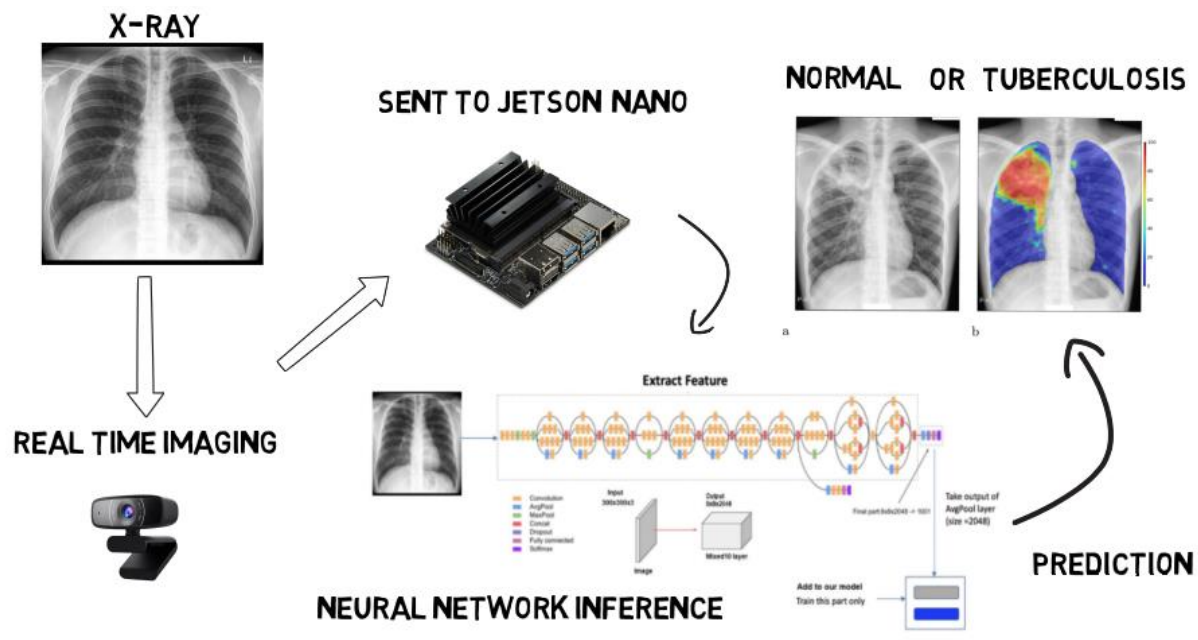


Figure 4: Work Flow Diagram

## Solution Breakdown:

### 1-Prepping Jetson Nano for Deep Learning:

The Jetson Nano Developer Kit uses a microSD card as a boot device and for main storage. First the Jetson Nano Developer Kit SD Card Image [v] was written on the memory card using the tool Belena Etcher. After boot up to save space unimportant applications were deleted. The detailed step by step setup instructions for setting up Jetson Nano for deep learning are listed here [vi]. The major steps involved are as follows:

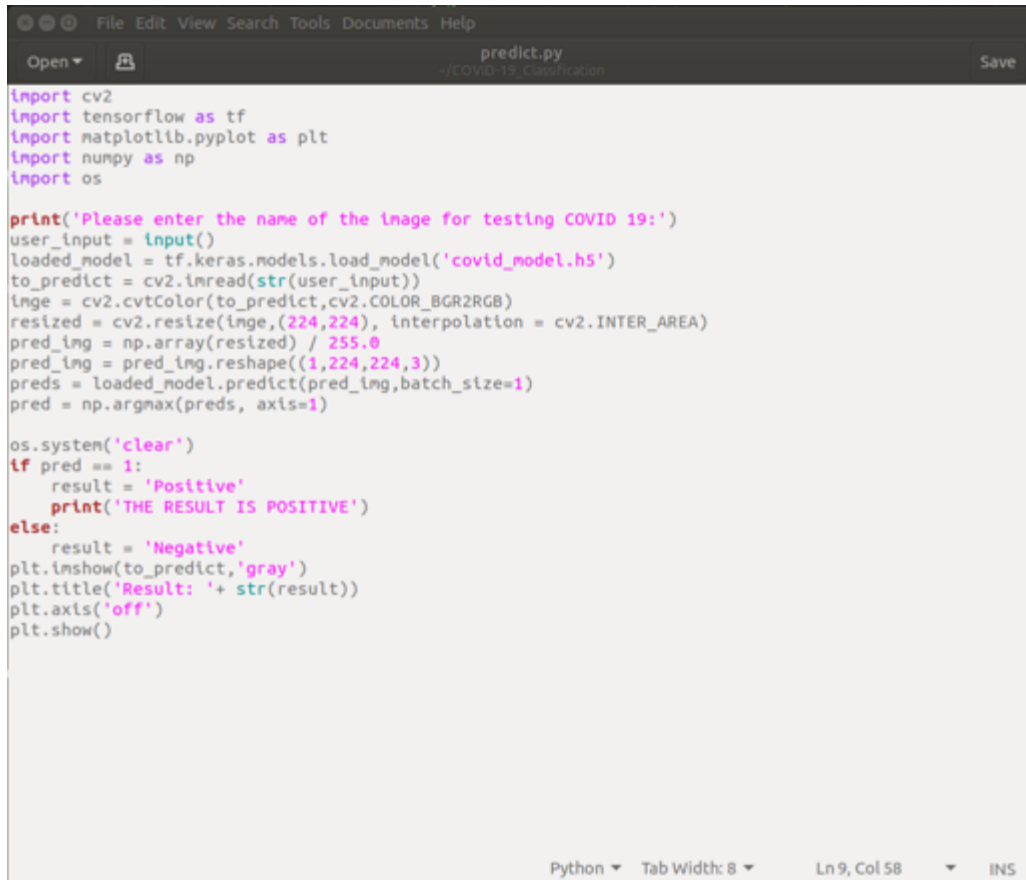
1. Installed some system level dependencies for Linux based Jetson Nano to install OPEN CV, then using these commands [vii] built the OPEN CV libraries onto Jetson Nano. Open CV is a great tool for image processing and performing computer vision tasks and is also of great importance in deep learning applications.
2. As our TB classification model is based on tensor flow framework it was imperative that we incorporated tensor flow capabilities into our Linux based Jetson Nano. So to install tensor flow we used the following steps [viii].



3. Using the same methodology in step a and b we downloaded the following system dependencies/packages to make our Jetson Nano a deep learning ready device:
  - Numpy
  - Scipy
  - Scikit-learn
  - Theano
  - PyTorch
  - Pandas
  - Matplotlib

## **2- Testing Jetson Nano's Deep Learning readiness:**

To test the jetson nano's deep learning readiness and to assess any pitfalls and fallacies we deemed it better to implement an already optimized classification model on the device. For this purpose the following Github\_repository [<sup>ix</sup>] was cloned into jetson nano. This is a Covid 19 classification model developed by hakantekgul [<sup>x</sup>]. After making a separate virtual environment [<sup>xi</sup>] for testing this model and installing the version of Open CV, Tensorflow and other dependencies that were suitable for this model we got it up and running and making real time prediction on chest X-rays for detecting covid 19 on the edge. The python script used to make inferences is appended here [<sup>xii</sup>].



```
import cv2
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os

print('Please enter the name of the image for testing COVID 19:')
user_input = input()
loaded_model = tf.keras.models.load_model('covid_model.h5')
to_predict = cv2.imread(str(user_input))
image = cv2.cvtColor(to_predict, cv2.COLOR_BGR2RGB)
resized = cv2.resize(image, (224, 224), interpolation = cv2.INTER_AREA)
pred_img = np.array(resized) / 255.0
pred_img = pred_img.reshape((1, 224, 224, 3))
preds = loaded_model.predict(pred_img, batch_size=1)
pred = np.argmax(preds, axis=1)

os.system('clear')
if pred == 1:
    result = 'Positive'
    print('THE RESULT IS POSITIVE')
else:
    result = 'Negative'
plt.imshow(to_predict, 'gray')
plt.title('Result: ' + str(result))
plt.axis('off')
plt.show()
```

Python Tab Width: 8 Ln 9, Col 58 INS

Figure 5: Python script used to load the saved model and make predictions

The image shows a terminal window on a Jetson Nano. The top part of the terminal shows the execution of a script named `predict.py` in the `COVID-19_Classification` directory. The script successfully opens dynamic libraries and prompts for an image name. The user enters `test_nocovid.jpeg`.

The bottom part of the terminal shows the output of the `nvtop` command, displaying system status and resource usage.

```
jetson@nano:~$ cd COVID-19_Classification
jetson@nano:~/COVID-19_Classification$ python3 predict.py
2022-03-12 16:30:30.657852: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dynamic library libcudart.so.10.2
2022-03-12 16:30:50.770651: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dynamic library libcudart.so.10.2
Please enter the name of the image for testing COVID 19:
test_nocovid.jpeg
```

```
NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.6 [L4T 32.6.1]
VR CPU1 [||||| Schedutil - 14%] 1.2GHz
CPU2 [||||| Schedutil - 10%] 1.2GHz
CPU3 [||||| Schedutil - 15%] 1.2GHz
CPU4 [||||| Schedutil - 11%] 1.2GHz
Jet Mem [|||||] 1.4G/4.1GB] (lfb 151x4MB)
Imm [|||||] 0.0k/252.0kB] (lfb 252kB)
Swp [|||||] 0.291GB/2.0GB] (cached 16MB)
EMC [|||||] 3%] 1.6GHz
GPU [|||||] 0%] 76 MHz
Dsk [|||||] 21.3GB/28.5GB]

[Info] [Sensor] [Temp] [Power/mW] [Cur] [Avr]
UpT: 0 days 0:10:11 AO 46.00C 5V CPU 198 598
FAN [|||||] 0%] Ta= 0% CPU 40.00C 5V GPU 39 85
Jetson Clocks: inactive CPU 38.50C ALL 2026 2553
NV Power[0]: MAXN PLL 37.50C
[HW engines] thermal 39.00C
APE: 25MHz
NVENC: [OFF] NVDEC: [OFF]
NVJPG: [OFF]
```

At the bottom of the terminal window, there is a navigation bar with the following options: `1ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit`. The name `Raffaello Bonghi` is visible in the bottom right corner.

Figure 6: Terminal commands used to call the script

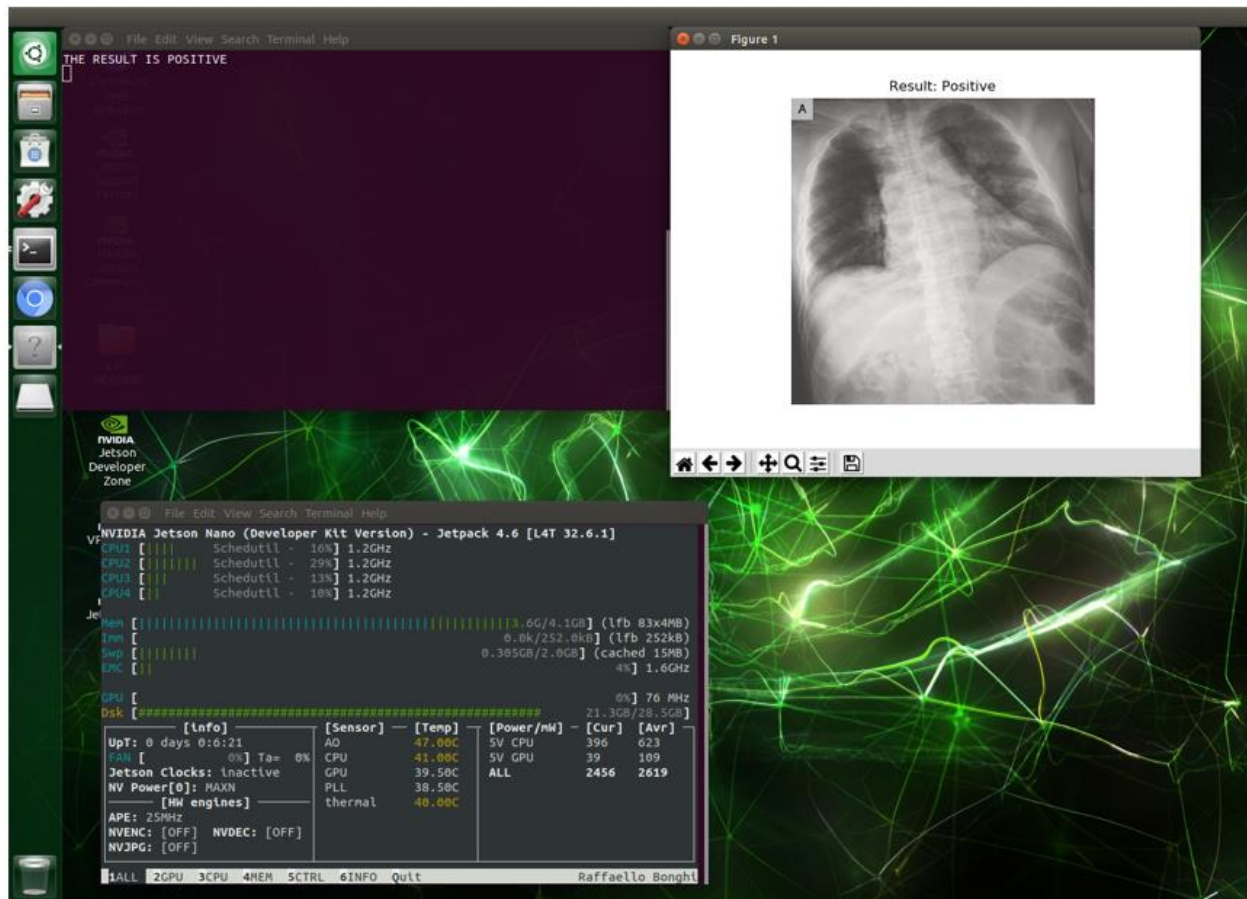


Figure 7: Output screenshot

### 3- Initial TB Classification Model:

Our project is a continuation of the previous year project where the students developed a CNN model using Inception V3 to classify tuberculosis from a chest X-Ray. They used Tensor Flow and Keras framework to implement this model. The colab code shared with us is appended below:

- 1- [HTTPS://COLAB.RE-  
SEARCH.GOOGLE.COM/DRIVE/1xBZXWVIfC\\_iwCbWKRERL2177BT2RM-  
T6?USP=SHARING](https://colab.research.google.com/drive/1xBZXWVIfC_iwCbWKRERL2177BT2RM-T6?usp=sharing) [xiii]
- 2- [HTTPS://COLAB.RE-  
SEARCH.GOOGLE.COM/DRIVE/1MVP5FX3HKCPL0MJMQNUZDQVID71HTUzi?usp=SHAR  
ING](https://colab.research.google.com/drive/1MVP5FX3HKCPL0MJMQNUZDQVID71HTUzi?usp=sharing)[xiv]

These codes are the intellectual property of Muhammad Zain Ishtiaq and Shanawar Chaudhary from batch '21.

They focused on building a CNN model for binary classification (Normal or TB) of chest X-

rays such that it reaches or outperforms the state-of-the-art accuracy. The following figure summarizes their works:

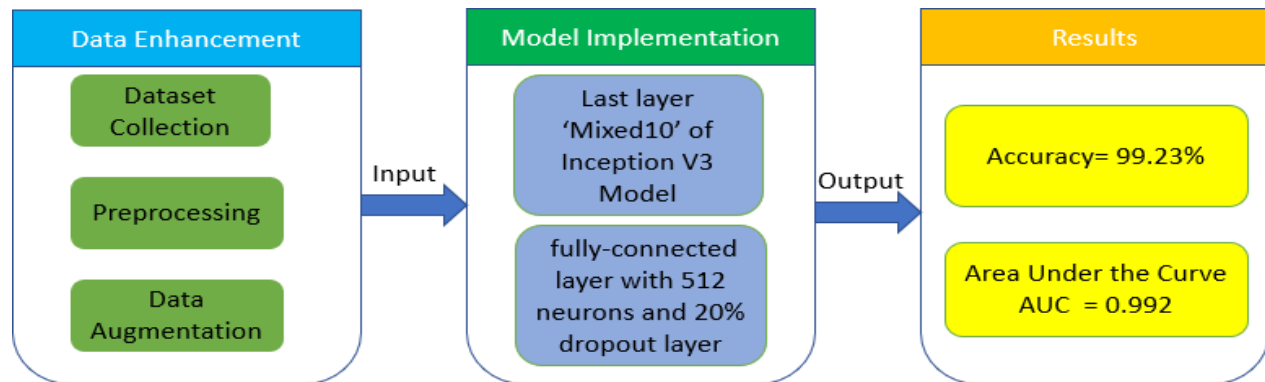


Figure 8: Summary of initial model

The code appended above was a very cluttered code where they tried and tested various model configurations and even used different models for transfer learning apart from that they also integrated a lot of image processing techniques which were not really relevant to us.

So, we isolated the parts that were useful to us and created a new colab file that had the best implementation from the original code in it. The isolated code is appended below <sup>[xv]</sup>:

- [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1VBG6QL4JSQBGoC330GG-PP8UKJWM3UR4?USP=SHARING](https://colab.research.google.com/drive/1VBG6QL4JSQBGoC330GG-PP8UKJWM3UR4?usp=sharing)

#### 4-Retraining the model:

The data set we had contained 7,178 images that were split into 5,741 images used for training the model and 1,437 images used for validation. Both these subsets contained nearly 50% normal and 50% TB infested images. Using the image generator method in colab we trained the model the results are appended below:

batch_normalization_87 (Batch Normalization)	(None, 8, 8, 384)	1152	['conv2d_87[0][0]']
batch_normalization_88 (Batch Normalization)	(None, 8, 8, 384)	1152	['conv2d_88[0][0]']
batch_normalization_91 (Batch Normalization)	(None, 8, 8, 384)	1152	['conv2d_91[0][0]']
batch_normalization_92 (Batch Normalization)	(None, 8, 8, 384)	1152	['conv2d_92[0][0]']
conv2d_93 (Conv2D)	(None, 8, 8, 192)	393216	['average_pooling2d_8[0][0]']
batch_normalization_85 (Batch Normalization)	(None, 8, 8, 320)	960	['conv2d_85[0][0]']
activation_87 (Activation)	(None, 8, 8, 384)	0	['batch_normalization_87[0][0]']
activation_88 (Activation)	(None, 8, 8, 384)	0	['batch_normalization_88[0][0]']
activation_91 (Activation)	(None, 8, 8, 384)	0	['batch_normalization_91[0][0]']
activation_92 (Activation)	(None, 8, 8, 384)	0	['batch_normalization_92[0][0]']
batch_normalization_93 (Batch Normalization)	(None, 8, 8, 192)	576	['conv2d_93[0][0]']
activation_85 (Activation)	(None, 8, 8, 320)	0	['batch_normalization_85[0][0]']
mixed9_1 (Concatenate)	(None, 8, 8, 768)	0	['activation_87[0][0]', 'activation_88[0][0]']
concatenate_1 (Concatenate)	(None, 8, 8, 768)	0	['activation_91[0][0]', 'activation_92[0][0]']
activation_93 (Activation)	(None, 8, 8, 192)	0	['batch_normalization_93[0][0]']
mixed10 (Concatenate)	(None, 8, 8, 2048)	0	['activation_85[0][0]', 'mixed9_1[0][0]', 'concatenate_1[0][0]', 'activation_93[0][0]']
flatten (Flatten)	(None, 131072)	0	['mixed10[0][0]']
dense (Dense)	(None, 1024)	134218752	['flatten[0][0]']
dropout (Dropout)	(None, 1024)	0	['dense[0][0]']
dense_1 (Dense)	(None, 1024)	1049600	['dropout[0][0]']
dense_2 (Dense)	(None, 1)	1025	['dense_1[0][0]']

=====

Total params: 157,072,161  
Trainable params: 135,269,377  
Non-trainable params: 21,802,784

Figure 9: Model Summary

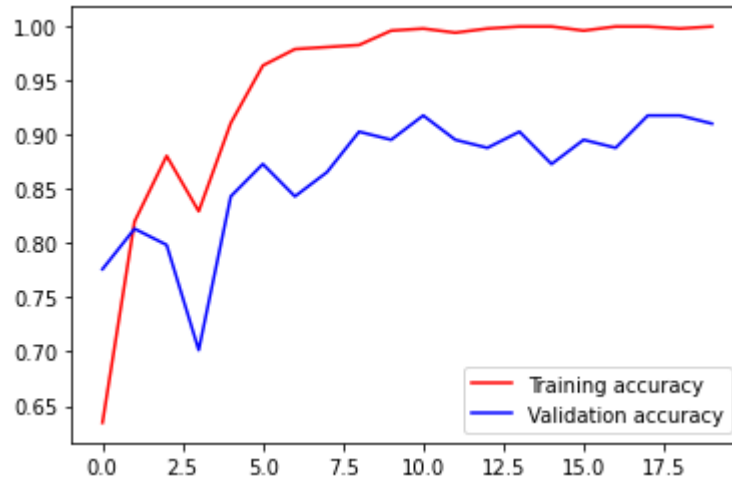


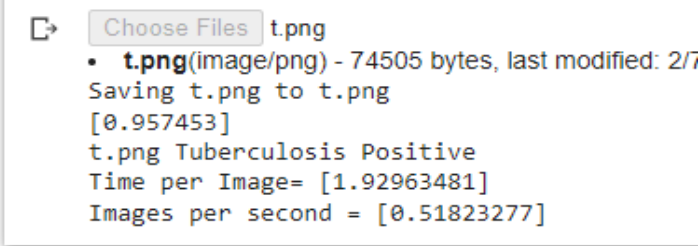
Figure 10: Training and validation accuracy

After training the model we saved the file using the command below. This commands saves the model in .h5 format. Although it's not necessary to save the model in this format it was easier to implement it on jetson nano that's why this format was given preference over .pb or .onnx formats. To convert the model into .h5 format from .pb, This [xvi] method can be used.

```
[ ] model.save('./model_1.h5')
```

To load the saved model and run prediction/inference on a single image and to measure the time it takes to make predictions we used the following python script [xvii]. The results are appended below:

```
if classes[0]>0.5:
    print(fn + " Tuberculosis Positive ")
else:
    print(fn + " Tuberculosis Negative ")
end_time = time.time()
elapsed_time = np.append(elapsed_time, end_time)
print('Time per Image=',elapsed_time)
print ('Images per second =',1/elapsed_time)
```



The terminal output shows a file upload process for 't.png' (74505 bytes) and the following inference results:

```
t.png Tuberculosis Positive
Time per Image= [1.92963481]
Images per second = [0.51823277]
```

Figure 11: Inference timing

**So, it took approximately 1.92 seconds to run inference on a single image at the rate of 0.5 images per second.**

The saved model file had an .h5 extension and its size was almost 1.6 GB which is a big constraint for ML at the edge due to memory issues.

## 5- Implementation of the saved model on Jetson Nano:

As we had already set up Jetson Nano with deep learning capabilities all that was left to do was to create another virtual environment and install the appropriate version of Tensor flow, Open CV and other dependencies and packages that were suitable for this model. Finally after prepping the Nano for it we re-used the Covid Python script to load this model and make predictions on TB. The results are appended below:



```

result = math_ops.add(result * (maxval - minval), minval, name=name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py", line 1164, in binary_op_wrapper
    return func(x, y, name=name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py", line 1496, in _mul_dispatch
    return multiply(x, y, name=name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/util/dispatch.py", line 201, in wrapper
    return target(*args, **kwargs)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py", line 518, in multiply
    return gen_math_ops.mul(x, y, name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/gen_math_ops.py", line 6068, in mul
    _ops.raise_from_not_ok_status(e, name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/ops.py", line 6862, in raise_from_not_ok_status
    six.raise_from(core._status_to_exception(e.code, message), None)
File "<string>", line 3, in raise_from
tensorflow.python.framework.errors_impl.ResourceExhaustedError: OOM when allocating tensor with shape[3,3,256,256] and type float on /job:localhost/replica:0/task:0/device:GPU:0 by allocator GPU_0_bfc [Op:Mul]

```

Figure 12: OOM Killer Error

```

NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.6 [L4T 32.6.1]
CPU1 [|||||] Schedutil - 16%] 1.2GHz
CPU2 [|||||] Schedutil - 29%] 1.2GHz
CPU3 [|||||] Schedutil - 13%] 1.2GHz
CPU4 [|||||] Schedutil - 10%] 1.2GHz

Mem [|||||] 3.6G/4.1GB] (1fb 83x4MB)
Inm [|||||] 0.0K/252.0KB] (1fb 252KB)
Swp [|||||] 0.305GB/2.0GB] (cached 15MB)
EMC [|||||] 4%] 1.6GHz

GPU [|||||] 0%] 76 MHz
Dsk [|||||] 21.3GB/28.5GB]

[Info] [Sensor] [Temp] [Power/mW] [Cur] [Avr]
UpT: 0 days 0:0:21 AO 47.00C 5V CPU 396 623
FAN [|||||] 0%] Ta= 0% CPU 41.00C 5V GPU 39 109
Jetson Clocks: inactive GPU 39.50C ALL 2456 2619
NV Power[0]: MAXN PLL 38.50C
[HW engines] thermal 40.00C
APE: 25MHz
NVENC: [OFF] NVDEC: [OFF]
NVJPG: [OFF]

```

Figure 13: No memory left in RAM

The error mentioned above is due to the fact that Jetson Nano is an Embedded hardware with limited main memory capacity and the model saved is a memory expensive unit that not only has a big size in bytes (1.6 GB) but also when loaded to make inference asks for a much larger memory capacity to make calculations which is just not possible for a portable low memory device.

After weeks of research and implementation of different memory saving techniques like making a SWAP memory nothing made the error go away so we concluded that the original saved model file namely model\_1.h5 <sup>[xviii]</sup> file had too big a size to be run on Jetson Nano. The following forum <sup>[xix]</sup> discussion also supports our argument.

## 6- Creation of a new, improved and optimized model:

So due to above mentioned constraints we decide to make a minimalistic yet robust model that was inspired from the original code. This model deleted the transfer learning part where inception V3 was used and instead made use of a bare Convolution Neural Network model (CNN) architecture combined with maxpooling layers for classification. The details are attached below. Full code is appended here [xx].

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import os
import shutil
import glob
import cv2
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
```

*Figure 14: The dependencies used are shown*

The ML framework used for this was also Tensorflow, Keras and Open CV.

The data set used had a total of 7,178 images divided as follows for training and validation.

```
print(f'total training normal images: {len(os.listdir(train_normal))}')  
print(f'total training tuberculosis images: {len(os.listdir(train_tuberculosis))}')  
print(f'total validation normal images: {len(os.listdir(validation_normal))}')  
print(f'total validation tuberculosis images: {len(os.listdir(validation_tuberculosis))}')
```

```
total training normal images: 2902  
total training tuberculosis images: 2840  
total validation normal images: 726  
total validation tuberculosis images: 711
```

*Figure 15: Data set*

The ‘train data generator’ method was used to preprocess the data set for training.

The optimization required the size of the model to be reduced. This was done by removing the inception V3 base model because it was a memory hungry model which made the size of the saved model file big. Instead, a bare CNN was used with its layers set manually to achieve smallest possible size. To further reduce the size of the model extra convolutional and maxpooling layers were removed to decrease the amount of computation that the hardware had to carry out.

The model contained an input layer that in turn contained three convolutional layer and three max pooling layers. After input layer a flatten, layer is used to convert the two dimensional image matrix into a single row matrix i.e. flat layer. The final densely connected layer has been implemented with 512 hidden layers and ‘ReLU’ activation. ReLU function gives the positive part of the argument as the output. Finally, the last output layer has only 1 neuron which makes use of ‘sigmoid’ activation. Sigmoid is an S shaped function that gives binary output either positive or

negative corresponding to the outputs that we have TB positive or TB negative.

```
import tensorflow as tf

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150 with 3 bytes color
    # This is the first convolution
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # # The fourth convolution (You can uncomment the 4th and 5th conv layers later to see the effect)
    # tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    # tf.keras.layers.MaxPooling2D(2,2),
    # # The fifth convolution
    # tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    # tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # Only 1 output neuron. It will contain a value from 0-1 where 0 for 1 class ('horses') and 1 for the other ('humans')
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

*Figure 16: The model*

## The model summary:

```
Model: "sequential"
Layer (type)                Output Shape                Param #
-----
conv2d (Conv2D)             (None, 298, 298, 16)       448
max_pooling2d (MaxPooling2D) (None, 149, 149, 16)       0
conv2d_1 (Conv2D)           (None, 147, 147, 32)       4640
max_pooling2d_1 (MaxPooling2D) (None, 73, 73, 32)       0
conv2d_2 (Conv2D)           (None, 71, 71, 64)        18496
max_pooling2d_2 (MaxPooling2D) (None, 35, 35, 64)       0
flatten (Flatten)           (None, 78400)              0
dense (Dense)               (None, 512)                40141312
dense_1 (Dense)             (None, 1)                  513
-----
Total params: 40,165,409
Trainable params: 40,165,409
Non-trainable params: 0
```

*Figure 17: The Model Summary*

The model was compiled using 'Binary Cross entropy' as the loss function. RMS prop optimizer was used using learning rate of 0.001.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(learning_rate=0.001),
              metrics=['accuracy'])
```

*Figure 18: Loss Function and Optimizer*

```

from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)
validation_datagen = ImageDataGenerator(rescale=1/255)

# Flow training images in batches of 128 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
    '/content/drive/MyDrive/train', # This is the source directory for training images
    target_size=(300, 300), # All images will be resized to 150x150
    batch_size=128,
    # Since you used binary_crossentropy loss, you need binary labels
    class_mode='binary')

# Flow training images in batches of 128 using train_datagen generator
validation_generator = validation_datagen.flow_from_directory(
    '/content/drive/MyDrive/val', # This is the source directory for training images
    target_size=(300, 300), # All images will be resized to 150x150
    batch_size=32,
    # Since you used binary_crossentropy loss, you need binary labels
    class_mode='binary')

Found 5741 images belonging to 2 classes.
Found 1437 images belonging to 2 classes.

```

*Figure 19: Using Image Generator to Train the Model.*

```

history = model.fit(
    train_generator,
    steps_per_epoch=8,
    epochs=18,
    verbose=1,
    validation_data = validation_generator,
    validation_steps=8)

```

*Figure 20: Training Parameters*

The accuracy of the optimized model turned out to be about 94 %

```

Epoch 17/18
8/8 [=====] - 108s 13s/step - loss: 0.1932 - accuracy: 0.9209 - val_loss: 0.1257 - val_accuracy: 0.9648
Epoch 18/18
8/8 [=====] - 109s 13s/step - loss: 0.1410 - accuracy: 0.9385 - val_loss: 0.2344 - val_accuracy: 0.9102

```

*Figure 21: Accuracy*

The model was saved. Then it was loaded back and the inference timing was measured using the following python script [xxi].

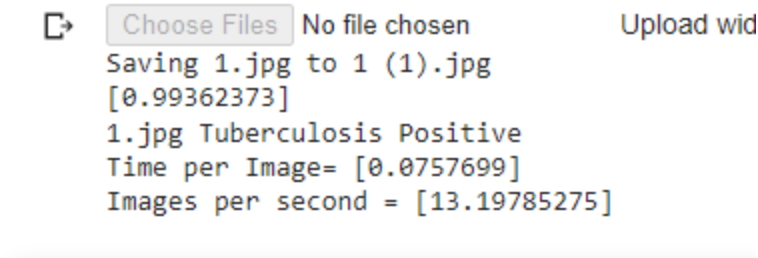


Figure 22: Inference Timing

**So, it took approximately 0.076 seconds to run inference on a single image at the rate of 13.2 images per second.**

## 7- Comparison of the two Models:

The results show that after creating the optimized model although the accuracy dropped from 98% to 94% but inference speed increased from 0.5 images per second to 13.2 images per second. Almost 26 times faster than before with a really small change in accuracy. Moreover, the number of parameters in the optimized model were reduced to almost less than a third (1/3) of the original: from 157,072,161 (157 million) to 40,165,409 (40 Million) which reflected in the file size of saved model which reduced from 1.6 GB to only 300 MBs which is a huge leap forward as far as the memory limited hardware i.e. Jetson Nano is concerned.

Performance Parameters	Original Model	Optimized Model
Accuracy	98%	93.85%
Inference Time Per Image	1.92 s	0.076 s
Inference Rate (Images/s)	0.5	13.2
Number of Parameters	157 Million	40 Million
Saved Model File Size	1.6 GB	300 MB
Able to run on Jetson Nano	NO	YES

Figure 23: Comparison of the Two Models

## 8- Implementation of the optimized model on Jetson Nano:

After the optimized model has been saved, its frozen graph is implemented on jetson nano using a camera that takes the image of an X-Ray held in front of it. That image is then passed through the loaded model to make predictions that then classifies the X-Ray as either TB positive or TB negative. The results are appended below:



After the script runs it takes live frame as inputs as soon as you press space bar it captures that particular frame from the camera input and when you press ESC it passes that saved image into the neural network for inference.

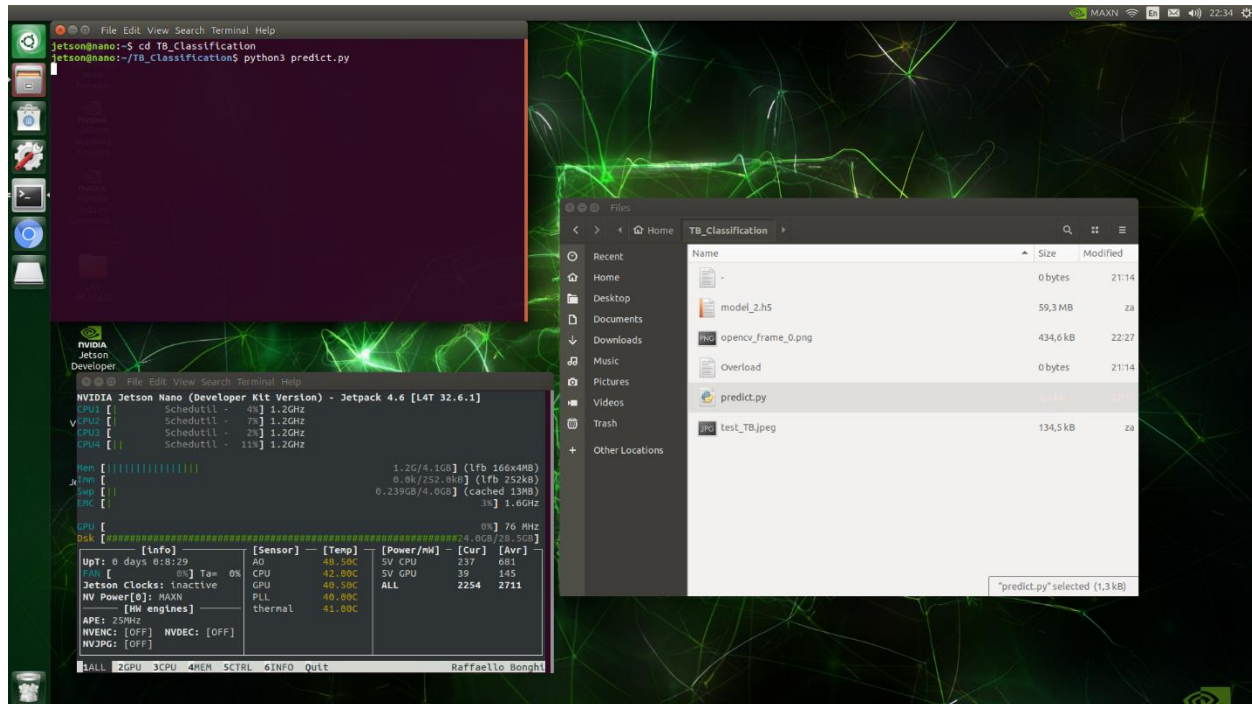


Figure 24: Linux Terminal Commands to call The Script



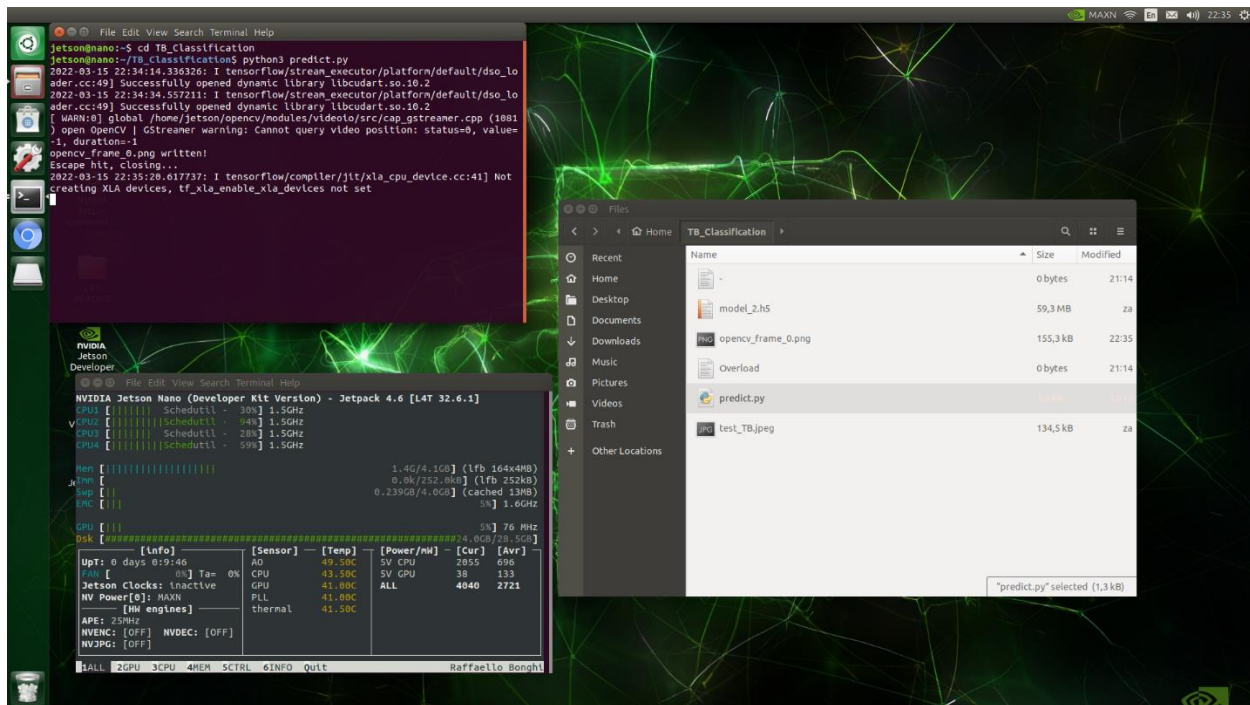


Figure 25: Command Running

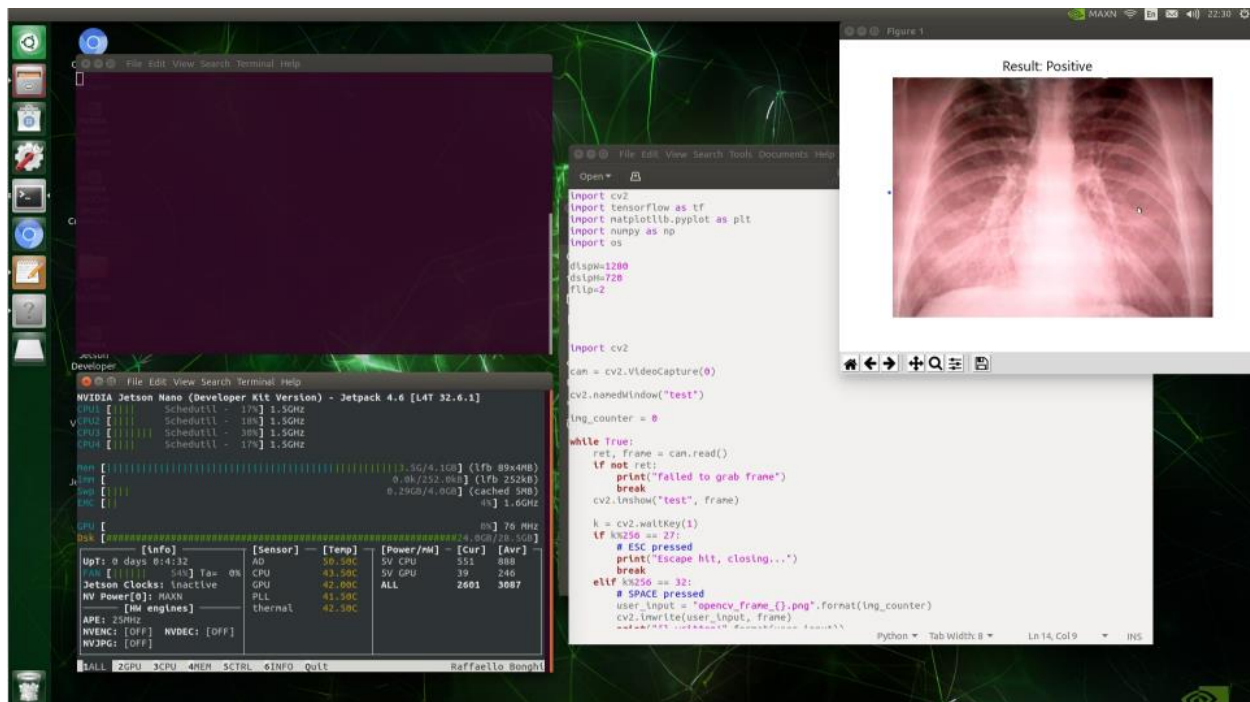
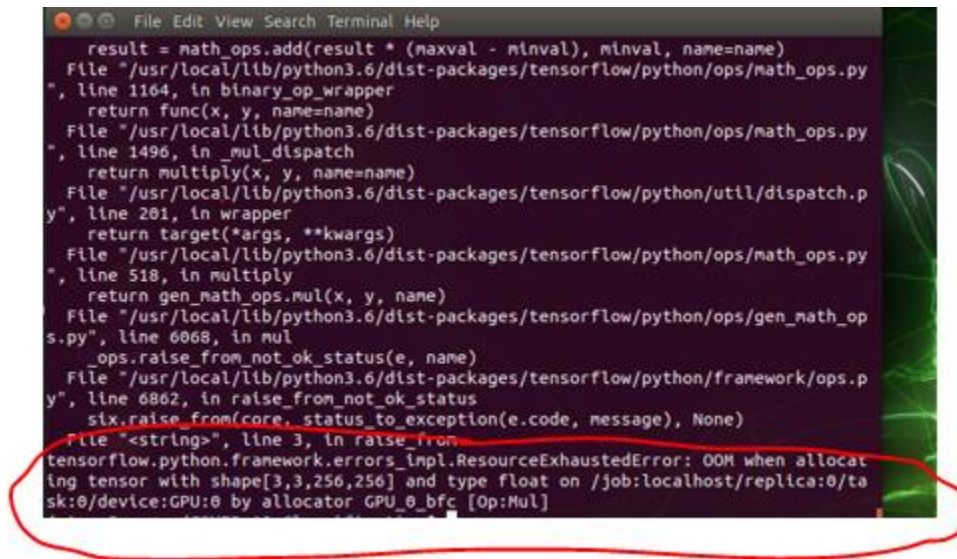


Figure 26: Output (Positive)

After implementing the optimized model on jetson nano at first the OOM killer (The Out Of Memory Killer or OOM Killer is a process that the linux kernel employs when the system is critically low on memory. This situation occurs because the linux kernel has over allocated

memory to its processes) [xxii] would kill the program even before it finishes execution.



```
result = math_ops.add(result * (maxval - minval), minval, name=name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py", line 1164, in binary_op_wrapper
    return func(x, y, name=name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py", line 1496, in _mul_dispatch
    return multiply(x, y, name=name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/util/dispatch.py", line 201, in wrapper
    return target(*args, **kwargs)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py", line 518, in multiply
    return gen_math_ops.mul(x, y, name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/gen_math_ops.py", line 6068, in mul
    _ops.raise_from_not_ok_status(e, name)
File "/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/ops.py", line 6862, in raise_from_not_ok_status
    six.raise_from(core._status_to_exception(e.code, message), None)
File "<string>", line 3, in raise_from
tensorflow.python.framework.errors_impl.ResourceExhaustedError: OOM when allocating tensor with shape[3,3,256,256] and type float on /job:localhost/replica:0/task:0/device:GPU:0 by allocator GPU_0_bfc [Op:Mul]
```

Figure 27: OOM Killer Error

To remedy this problem a swap file (*Swap space* in Linux is used when the amount of physical memory (RAM) is full. If the system needs more memory resources and the RAM is full, inactive pages in memory are moved to the swap space) [xxiii] was created. The creation of swap would allow the optimized model to make inference on the jetson nano atleast once before the swap filled up and the device required a reboot. In essence even after the creation of an alternative memory unit the device could only make a single inference at every boot up which was a performance constraint and had to be fixed.

## 9- Creation of a new model that solves the memory issue:

As the previously optimized model could only run inference a single time at every boot, a new CNN model was required that had an even smaller saved model file size. To achieve this, we implemented various different classification architectures like ResNet 18, ResNet 50, Inception V3, Inception V4 and finally VGG16.

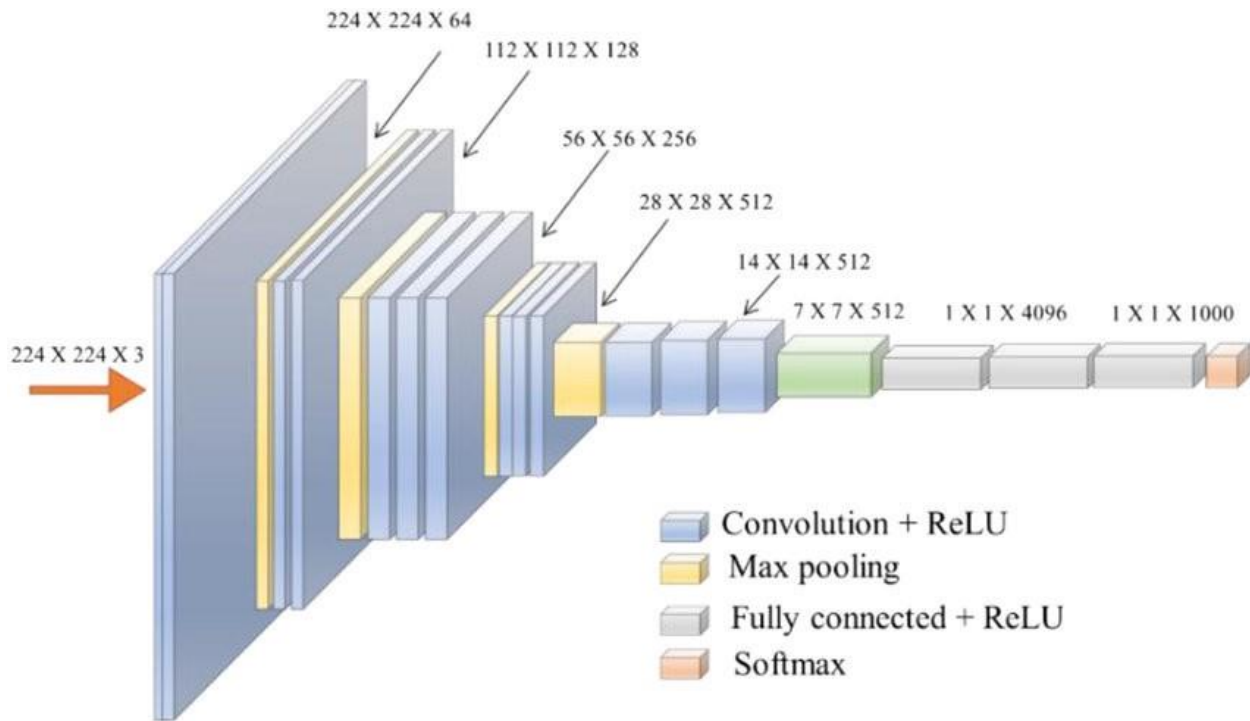


Figure 28: VGG16 Model

The figure 28 above represents the abstract working of VGG16 based models. After comparison of all of the above mentioned models we concluded that VGG16 was the least memory hungry architecture and decided to train our new model on this one. The full code of this new memory optimized model is appended here <sup>[xxiv]</sup> and the summary of the code is explained below:

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import os
import shutil
import glob
import cv2
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical

```

Figure 29: Packages Imported

The same data set as in the previous model with 7178 images was used here using image generator method.

VGG16 is trained on the google's imagenet data set which consists of millions of images. To use this model for the purpose of Tuberculosis Classification the base model was downloaded and ontop of it we implemented our own model to detect TB. The input size of the image was selected to be (224,224,3). This was reduced from (300,300,3) of the previous models to save up on memory utilizations on the edge.

```

▶ baseModel = VGG16(weights="imagenet", include_top = False, input_tensor=Input(shape=(224,224,3)))

```


 Downloading data from [https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16\\_weights\\_tf\\_dim\\_ordering\\_tf\\_kernels\\_notop.h5](https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5)  
58892288/58889256 [=====] - 0s 0us/step  
58900480/58889256 [=====] - 0s 0us/step

Figure 30: Base Model

```

headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(64, activation="relu")(headModel)
headModel = Dropout(0.4)(headModel)
headModel = Dense(1, activation="sigmoid")(headModel)

```

*Figure 31:Top Layer*

The output layer has only 1 neuron which makes use of ‘sigmoid’ activation. Sigmoid is an S shaped function that gives binary output either positive or negative corresponding to the outputs that we have TB positive or TB negative.


The optimizer function used for this model was Adam <sup>[xxv]</sup> with a learning rate of lr=1e-3 and a decay of 1e-3/10. The loss function used was binary crossentropy (Binary Cross Entropy is the negative average of the log of corrected predicted probabilities. Binary cross entropy compares each of the predicted probabilities to actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far from the actual value.) <sup>[xxvi]</sup>


```

opt = Adam(lr=1e-3, decay=1e-3 / 10)
model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["accuracy"])

```

*Figure 32: Loss Function and Optimizer of VGG16 Model*

 `model.summary()`

 Model: "model"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160

*Figure 33: Model Summary*

▶	block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
↗	block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
	block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
	block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
	block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
	block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
	block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
	average_pooling2d (AveragePooling2D)	(None, 1, 1, 512)	0
	flatten (Flatten)	(None, 512)	0
	dense (Dense)	(None, 64)	32832
	dropout (Dropout)	(None, 64)	0
	dense_1 (Dense)	(None, 1)	65
=====			
Total params: 14,747,585			
Trainable params: 32,897			
Non-trainable params: 14,714,688			

Figure 34: Model Summary (Continued)

The model is fitted and trained using the following parameters:

```
history = model.fit(
    train_generator,
    steps_per_epoch=8,
    epochs=25,
    verbose=1,
    validation_data = validation_generator,
    validation_steps=8)
```

Figure 35: Training Parameters



After training the accuracy came out to be about 92%. A 2% decrease from the previous model but the tradeoff was definitely worth it because the saved model file size was reduced from 300MBs to only 60 MBs.

## 10- Implementation of VGG16 model on Jetson Nano:

After the VGG16 model was trained on our TB dataset we saved the model file [xxvii] which had a size of only 60 MBs. After that the python script [xxviii] was used to load the saved model, get an image of X-Ray from the camera, pass it through the neural network and make a prediction as either TB positive or Negative. Note that this implementation made use of the original VGG16 model without any quantization steps involved.

The implantation yields the following results:

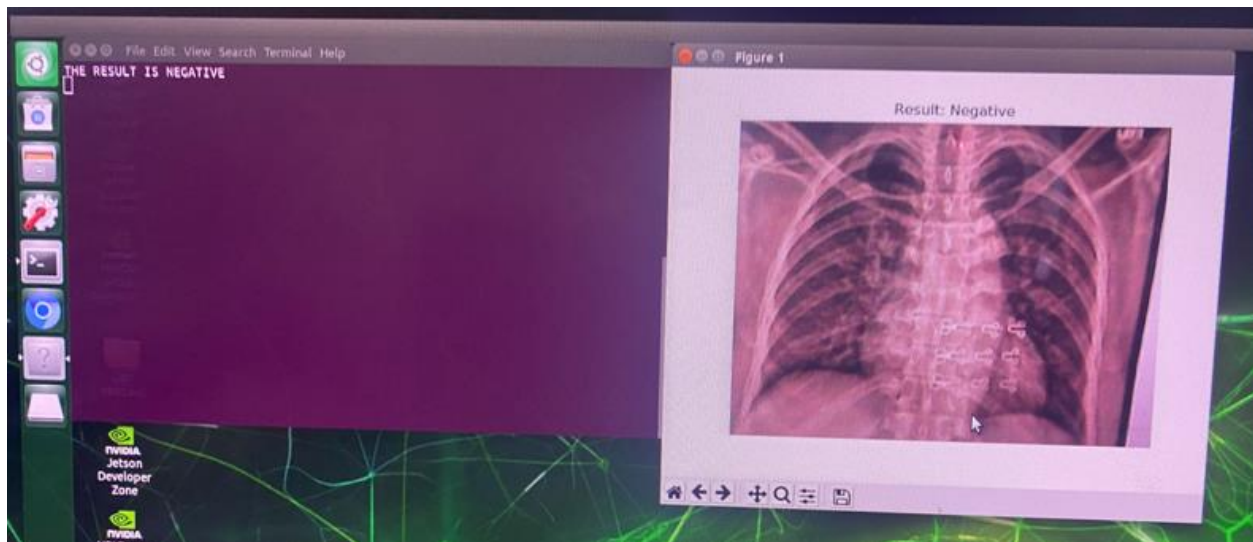


Figure 36: Output (Negative)



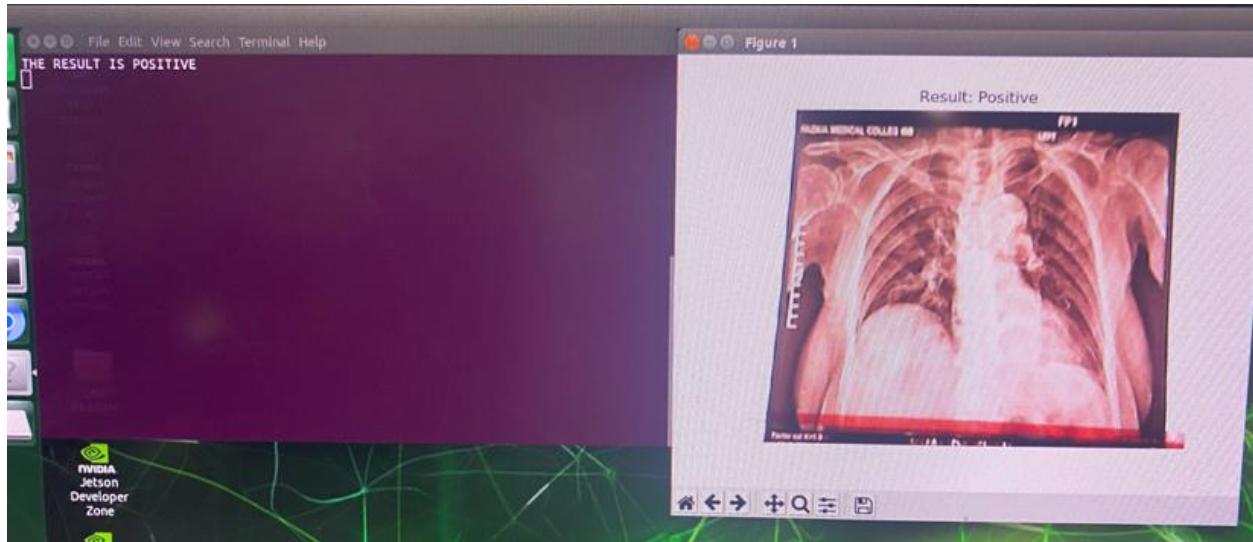


Figure 37: Output (Positive)

Moreover, this new memory optimized model also solves the OOM killer problem. Now the device can make as many inferences from a single boot as required without the OOM killer killing of the process.

## 11- Testing the performance of the device on real X-Rays:

After the final VGG16 based model was successfully deployed onto the device a portable embedded hardware for TB detection was fashioned. We used the portable X-Ray classifier to test the accuracy of it using real world TB and normal X-Rays from Fazaia Medical College E9 Islamabad.

Twenty X-Rays were tested manually over the course of a few weeks out of which 10 were TB infested and 10 were normal. The results are best summarized by the following confusion matrix table:

	Actually Positive		Actually Negative	
Predicted Positive	TP	= 8	FP	= 0
Predicted negative	FN	= 2	TN	= 10

Figure 38: Confusion Matrix

Some other performance metrics are also represented in tabular form:

Measure	Value	Derivations
<b>Sensitivity</b>	0.8000	$TPR = TP / (TP + FN)$
<b>Specificity</b>	1.0000	$SPC = TN / (FP + TN)$
<b>Precision</b>	1.0000	$PPV = TP / (TP + FP)$
<b>Negative Predictive Value</b>	0.8333	$NPV = TN / (TN + FN)$
<b>False Positive Rate</b>	0.0000	$FPR = FP / (FP + TN)$
<b>False Discovery Rate</b>	0.0000	$FDR = FP / (FP + TP)$
<b>False Negative Rate</b>	0.2000	$FNR = FN / (FN + TP)$
<b>Accuracy</b>	0.9000	$ACC = (TP + TN) / (P + N)$
<b>F1 Score</b>	0.8889	$F1 = 2TP / (2TP + FP + FN)$
<b>Matthews Correlation Coefficient</b>	0.8165	$TP*TN - FP*FN / \sqrt{((TP+FP)*(TP+FN)*(TN+FP)*(TN+FN))}$

*Figure 39: Performance Metrics*

The actual accuracy of the model turned out to be about 90% not a huge dip from the theoretical accuracy of 92.08%.

The following tables shows the comparison of different models implemented throughout the course of the project:

Performance Parameters	Original Model	Optimized Model
Accuracy	98%	93.85%
Inference Time Per Image	1.92 s	0.076 s
Inference Rate (Images/s)	0.5	13.2
Number of Parameters	157 Million	40 Million
Saved Model File Size	1.6 GB	300 MB
Able to run on Jetson Nano	NO	YES

Performance parameters:	VGG16 based Final Code:
Accuracy	92.08 %
Saved model file size	60 MB
Able to run on jetson Nano	Yes

*Figure 40: Comparison Of The Three Models*

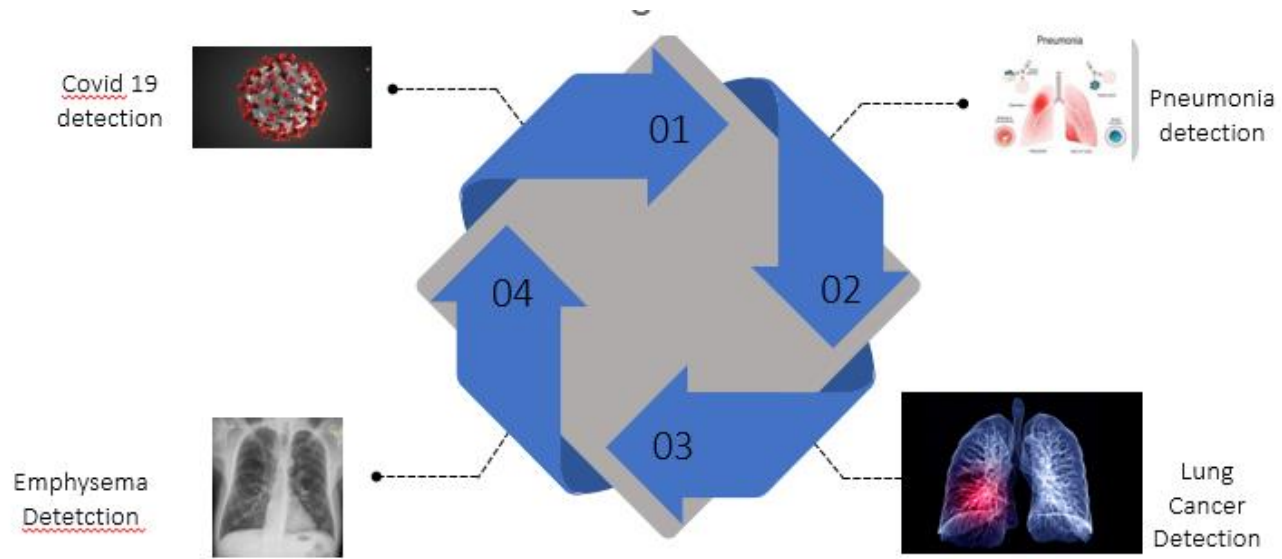
### **Conclusion and Future Applications**

Human error in interpretation of CXR and the lack of expert radiologists in underdeveloped countries make TB identification a difficult procedure. Computer-aided identification of TB has greatly increased the efficiency of detection.

In this project, we studied the technical specifications of different embedded systems in the literature review and selected the best embedded system i.e., Nvidia Jetson Nano, regarding the memory, power and cost suitable for the project. Then the Nano was prepared for deep learning by installing different system level dependencies on the embedded system. Furthermore, a memory constraint was realized so we optimized the model and reduced its size so that the new model can smoothly run on the low memory device, Jetson Nano. Efficiency was further increased by creating a model on state-of-the-art architecture and recorded the results of all three different models. Compared the results of the different models to find the pros and cons of each model. Finally, we integrated a camera on the Jetson Nano to take real life images of CXRs and attached a Bluetooth mouse-keyboard to make the device portable.

The project provides an embedded device that uses a Machine Learning based algorithm for TB detection. In the same manner, the device can be used to implement other models for the diagnosis of other lung diseases such as:

1. Lung cancer
2. Covid
3. Emphysema
4. Pneumonia



*Figure 41: Future Applications*

## References

- 
- [i] [HTTPS://WWW.WHO.INT/PUBLICATIONS/I/ITEM/9789240037021](https://www.who.int/publications/i/item/9789240037021)
- [ii] [HTTPS://WWW.WHO.INT/PUBLICATIONS/I/ITEM/9789240037021](https://www.who.int/publications/i/item/9789240037021)
- [iii] [HTTPS://WWW.MDPI.COM/2227-7080/10/2/37/HTM](https://www.mdpi.com/2227-7080/10/2/37/htm)
- [iv] [HTTPS://IEEEEXPLORE.IEEE.ORG/DOCUMENT/9152915](https://ieeexplore.ieee.org/document/9152915)
- [v] [HTTPS://DEVELOPER.NVIDIA.COM/EMBEDDED/LEARN/GET-STARTED-JETSON-NANO-DEVKIT#WRITE](https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#write)
- [vi] [HTTPS://PYIMAGESEARCH.COM/2020/03/25/HOW-TO-CONFIGURE-YOUR-NVIDIA-JETSON-NANO-FOR-COMPUTER-VISION-AND-DEEP-LEARNING/](https://pyimagesearch.com/2020/03/25/how-to-configure-your-nvidia-jetson-nano-for-computer-vision-and-deep-learning/)
- [vii] [HTTPS://AUTOMATICADDISON.COM/HOW-TO-INSTALL-OPENCV-4-5-ON-NVIDIA-JETSON-NANO/](https://automaticaddison.com/how-to-install-opencv-4-5-on-nvidia-jetson-nano/)
- [viii] [HTTPS://QENGINEERING.EU/INSTALL-TENSORFLOW-2.4.0-ON-JETSON-NANO.HTML](https://qengineering.eu/install-tensorflow-2.4.0-on-jetson-nano.html)
- [ix] [HTTPS://GITHUB.COM/HAKANTEKGUL/COVID-19\\_CLASSIFICATION](https://github.com/Hakantekgul/COVID-19_classification)
- [x] [HTTPS://GITHUB.COM/HAKANTEKGUL](https://github.com/Hakantekgul)
- [xi] [HTTPS://ABHATIKAR.MEDIUM.COM/MAKE-YOUR-NVIDIA-JETSON-NANO-DEEP-LEARNING-READY-A8F5FCD7B25C](https://abhatikar.medium.com/make-your-nvidia-jetson-nano-deep-learning-ready-a8f5fcd7b25c)
- [xii] [HTTPS://GITHUB.COM/HAKANTEKGUL/COVID-19\\_CLASSIFICATION/BLOB/MASTER/PREDICT.PY](https://github.com/Hakantekgul/COVID-19_classification/blob/master/predict.py)
- [xiii] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1VPH17U1GWQI70E-T49J9YBAMVLE1DDFz/VIEW?USP=SHARING](https://colab.research.google.com/drive/1VPH17U1gWQI70E-T49J9YBAMVLe1DDFz/view?usp=sharing)  
[HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1xBZXWVIFC\\_IWCbWKRerL2177Bt2RM-T6?USP=SHARING](https://colab.research.google.com/drive/1xBZXWVIfC_IwCbWKRerL2177Bt2RM-T6?usp=sharing)
- [xiv] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1MVP5FX3HKCPL0MJMQNUZDQVID71HTUZI?USP=SHARING](https://colab.research.google.com/drive/1MVP5Fxf3HKCPL0MjMQNUZDQvid71HtUzi?usp=sharing)
- [xv] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1VBG6QL4JSQBGOc330GG-PP8UKJWM3UR4?USP=SHARING](https://colab.research.google.com/drive/1VBG6QL4JSQBGOc330GG-PP8UKJWM3UR4?usp=sharing)
- [xvi] [HTTPS://STACKOVERFLOW.COM/QUESTIONS/59375679/TENSORFLOW-PB-FORMAT-TO-KERAS-H5](https://stackoverflow.com/questions/59375679/tensorflow-pb-format-to-keras-h5)
- [xvii] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1KTUSUCIGekkJQRd5MYRMvHOC7SmRJNHR?USP=SHARING](https://colab.research.google.com/drive/1KTUSuCIgekkJQRd5MYRMvHOC7SmRJNHR?usp=sharing)
- [xviii] [HTTPS://DRIVE.GOOGLE.COM/FILE/D/13X2TF5ZZGDQHYJ0BTONKDXRXVQ4EXTJF/VIEW](https://drive.google.com/file/d/13X2Tf5zzGDqHYJ0BtonkdXRxVq4eXtJf/view)

---

[xix] [HTTPS://FORUMS.DEVELOPER.NVIDIA.COM/T/TENSORFLOW-GPU-NOT-WORKING-IN-NANO/82171](https://forums.developer.nvidia.com/t/tensorflow-gpu-not-working-in-nano/82171)

[xx] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/19DtNOi1QV3JtN-VQHAANLhUzXAJUM4QR?usp=sharing](https://colab.research.google.com/drive/19DtNOi1QV3JtN-VQHAANLhUzXAJUM4QR?usp=sharing)

[xxi] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1ktUSuClGekkJQRD5MYrMvHOC7SmRJNHR?usp=sharing](https://colab.research.google.com/drive/1ktUSuClGekkJQRD5MYrMvHOC7SmRJNHR?usp=sharing)

[xxii] [HTTPS://NEO4J.COM/DEVELOPER/KB/LINUX-OUT-OF-MEMORY-KILLER/#:~:text=The%20out%20of%20memory%20killer,of%20memory%20from%20the%20kernel.](https://neo4j.com/developer/kb/linux-out-of-memory-killer/#:~:text=The%20out%20of%20memory%20killer,of%20memory%20from%20the%20kernel.)

[xxiii] [HTTPS://WEB.MIT.EDU/RHEL-DOC/5/RHEL-5-MANUAL/DEPLOYMENT\\_GUIDE-EN-US/CH-SWAP-SPACE.HTML#:~:text=swap%20space%20in%20linux%20is,a%20replacement%20for%20more%20ram.](https://web.mit.edu/rhel-doc/5/rhel-5-manual/deployment_guide-en-us/ch-swap-space.html#:~:text=swap%20space%20in%20linux%20is,a%20replacement%20for%20more%20ram.)

[xxiv] [HTTPS://COLAB.RESEARCH.GOOGLE.COM/DRIVE/1PAeRYkVPORyeI0rUNgQYTJgQMBcZFISE?authuser=1](https://colab.research.google.com/drive/1PAeRYkVPORyeI0rUNgQYTJgQMBcZFISE?authuser=1)

[xxv] [HTTPS://WWW.GEEKSFORGEKS.ORG/INTUITION-OF-ADAM-OPTIMIZER/#:~:text=Adam%20optimizer%20involves%20a%20combination,minima%20in%20a%20faster%20pace.](https://www.geeksforgeeks.org/intuition-of-adam-optimizer/#:~:text=Adam%20optimizer%20involves%20a%20combination,minima%20in%20a%20faster%20pace.)

[xxvi] [HTTPS://WWW.ANALYTICSVIDHYA.COM/BLOG/2021/03/BINARY-CROSS-ENTROPY-LOG-LOSS-FOR-BINARY-CLASSIFICATION/](https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/)

[xxvii] [HTTPS://DRIVE.GOOGLE.COM/FILE/D/1VPH17U1GWQI70E-T49J9yBAMVLe1DDFz/view?usp=sharing](https://drive.google.com/file/d/1VPH17U1GWQI70E-T49J9yBAMVLe1DDFz/view?usp=sharing)

[xxviii] [HTTPS://DRIVE.GOOGLE.COM/FILE/D/1E4NGH7EOEiIZb48Af-Q-UWY-JTUXKLG5/view?usp=sharing](https://drive.google.com/file/d/1E4NGH7EOEiIZb48Af-Q-UWY-JTUXKLG5/view?usp=sharing)