Final Project: Prediction of Active Tuberculosis on Smear Images

Team members: Dream Lopez, Meilin Yen, Akchar Bhagat, Coco Sheng, Meriem Cherif

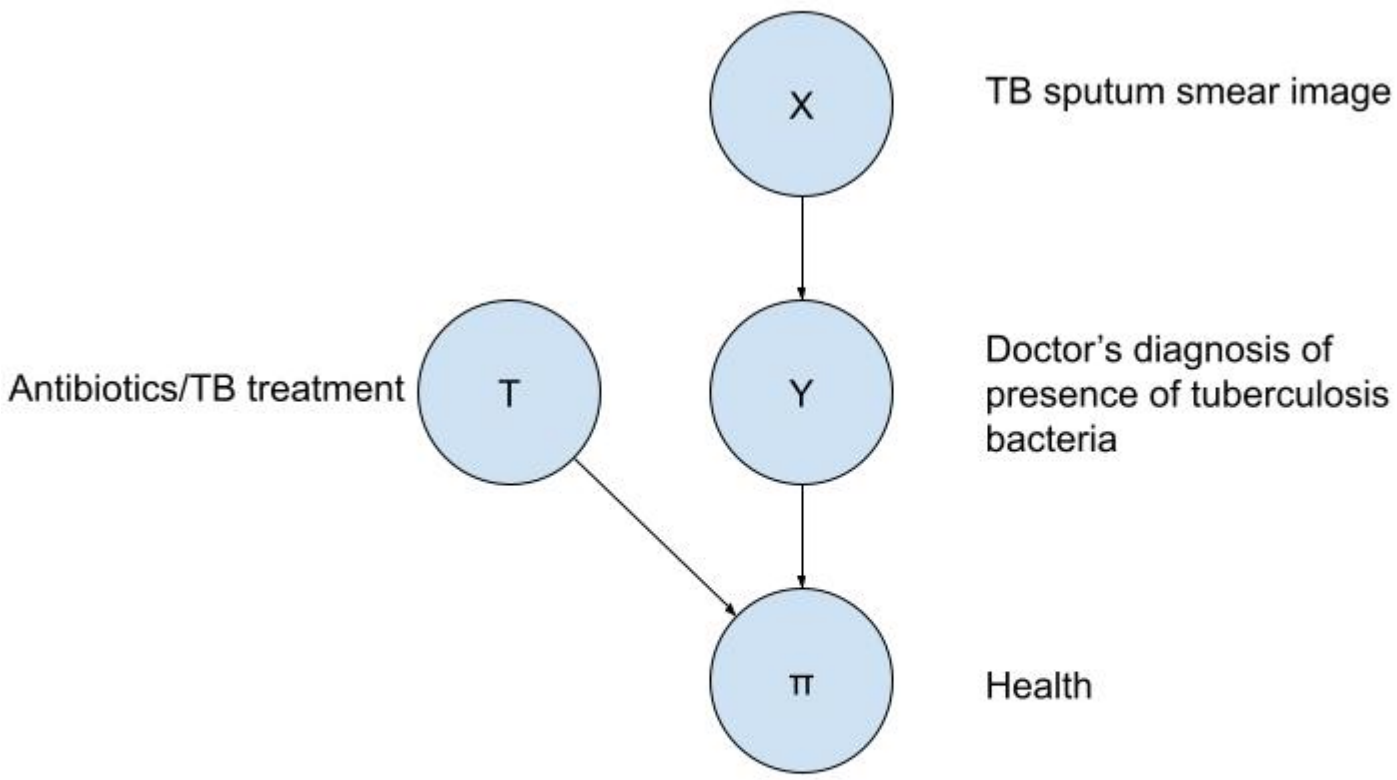
Welcome to Public Health 132 AI for Health and Healthcare Group 5's final project. In this project, we are working with Tuberculosis(TB) Smear Images and aim to predict the probability that a doctor will see active TB on a sputum slide. The goal of this project is to develop an algorithm that improves the identification of TB and helps patients minimize costs in the process, hence tailoring better treatment and optimizing healthcare resources.

Our Dataset

This dataset is a subset of tuberculosis smear images that Wellgen Medical collected from across Asia to train and validate their smear microscopy automated system. The slides mostly come from Taiwan, and partially from China, India, and Japan. The slides are digitized using a microscopic scanner and are subdivided into images with the resolution of 2448 x 2048. All positive smear images in the dataset come from sputum slides with culture-proven tuberculosis. For more specific details please refer to https://www.ngsci.org/updates/detecting-tuberculosis-using-ai.

Problem Description

Tuberculosis is a preventable infectious disease that 25% of the world has in their bodies. In 2022, 10.6 million people became sick with the disease's conversion to symptomatic active TB. This disease can be diagnosed by a doctor or a physician looking at a patient's sputum under a microscope. The presence of a rod-like bacilli indicates Active TB, and leads to a diagnosis. Using the Nightingale Open Science Dataset on detecting Active Tuberculosis (TB), our algorithm aims to streamline the Active TB detection process, by passing through sputum images and returning a diagnosis prediction.



As shown in the diagram above, the X in our prediction problem is the TB sputum smear images, the Y is the doctor's diagnosis of presence of active tuberculosis Bacilli present, the T is the antibiotics or TB treatment, and the payoff π is health.

Potential Pitfalls of Algorithm

One pitfall of our algorithm is that the label (Y) in our dataset is based on whether a doctor sees active TB on the slide or not. This is different from Y*, which is whether the patient actually has TB or not. In creating the dataset, the doctor needs to grade the X-Rays; hence, the objective data (the X-ray) is subject to the doctor's judgment. A doctor could see an X-Ray from someone who has active TB (Y* = 1)* but not notice the active TB, giving that X-Ray a label of Y = 0 for no active TB. Alternatively, a doctor could see an X-Ray from someone who doesn't have active TB (Y* = 0) but mistake something else for active TB, giving that X-Ray a label of Y = 1 for active TB.

Another possible pitfall in building our algorithm is that the training data is sourced predominantly from a specific geographic region. When the training data originates from a specific geographic region, it may hinder the algorithm's generalizability to other areas. The resulting algorithm could exhibit biased decision-making, inaccurate predictions, and vulnerability to changes in data distribution. Scaling the algorithm to new regions could become a challenge in this scenario.

Import Key Packages

In [2]:
```
import os
import random
import datetime, time
from tqdm import tqdm

import pandas as pd
import numpy as np
import pickle

import matplotlib.pyplot as plt
from PIL import Image

from concurrent.futures import ProcessPoolExecutor

from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score, roc_curve, confusion_matrix, cl

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms as transforms
from torchvision.models import vgg19_bn, resnet50, densenet121, vit_b_16
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter

log_dir = "~/logs"
writer = SummaryWriter(log_dir)
device = "cuda:0" if torch.cuda.is_available() else "cpu"
```

```
2023-12-16 00:09:15.527753: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized
to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F FMA, in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2023-12-16 00:09:16.394081: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find T
ensorRT
```

Obtain Data

In [3]:
```
def get_filepaths(directory):
    filepaths = []
    for root, dirs, files in tqdm(os.walk(directory)):
        for name in files:
            filepath = os.path.join(root, name)
            filepaths.append(filepath)
    return filepaths

# Directory to scan
directory_to_scan = "/home/ngsci/datasets/tb-wellgen-smear/images/"

# Get list of top-level filepaths
data_filepaths = get_filepaths(directory_to_scan)

# Print out the list, or do something else with it
print(len(data_filepaths))
```

```
302it [00:00, 354.74it/s]
75087
```

In [4]:
```
labels = pd.read_csv("/home/ngsci/datasets/tb-wellgen-smear/v1/tb-labels.csv")
labels
```

Out[4]:

	image	tb_positive	file_path
0	tb00000001.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
1	tb00000002.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
2	tb00000003.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
3	tb00000004.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
4	tb00000005.jpg	1	/home/ngsci/datasets/tb-wellgen-smear/images/0...
...	...	...	...
75082	tb00075083.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
75083	tb00075084.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
75084	tb00075085.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
75085	tb00075086.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...
75086	tb00075087.jpg	0	/home/ngsci/datasets/tb-wellgen-smear/images/0...

75087 rows × 3 columns

Table 1. Dataset Summary

In [5]:
```
tb_counts = labels['tb_positive'].value_counts()
total_images = len(labels)
percentage_tb_positive = (tb_counts[1] / total_images) * 100
percentage_tb_negative = (tb_counts[0] / total_images) * 100

df = pd.DataFrame({
    'Count': [tb_counts[1], tb_counts[0], total_images],
    'Percentage': [f'{percentage_tb_positive:.1f}%', f'{percentage_tb_negative:.1f}%', '']},
    index=pd.Index(['Images with TB Bacilli Present (TB-Positive)', 'Images with no TB Bacilli Present (TB-Negative
                    'Total Images in Dataset'])
)
df.style
```

Out[5]:

	Count	Percentage
Images with TB Bacilli Present (TB-Positive)	3976	5.3%
Images with no TB Bacilli Present (TB-Negative)	71111	94.7%
Total Images in Dataset	75087	

Here, we see that the dataset has only around 5% positive TB smears - this makes it more likely that the algorithm would result in false negatives. This could be a potential problem because a false positive can be quickly corrected by a doctor visit, but a false negative makes it more difficult for a patient to get treatment. The opportunity cost of a false positive is lower than the opportunint cost of a false negative.

Data Schema

In [6]:
```
import pandas as pd
import plotly.graph_objects as go

# Import data
data = {
    'source': [0, 0],
    'target': [1, 2],
    'value': [tb_counts[1], tb_counts[0]]
}

df = pd.DataFrame(data)

# Create a Sankey diagram
fig = go.Figure(data=[go.Sankey(
node=dict(
pad=15,
thickness=20,
line=dict(color='black', width=0.5),
label=['Total X-ray Images', 'Positive TB', 'Negative TB']
),
link=dict(
source=df['source'],
target=df['target'],
value=df['value']
)
)])

# Customize layout
fig.update_layout(
```

```
title_text="X-ray Images for TB Detection",
font_size=10,
autosize=False,
width=800,
height=400
)

# Display the diagram
fig.show()
```

Key Variables

BACILLI:

Bacilli are bacteria that are characterized by their rod-like or cylindrical shape. Wellgen Medical has identified images that contain tuberculosis bacilli and those that do not. This dataset contains both those types of images, and those labels can be found in the tb-labels.csv table in the dataset.

Building and Training the Algorithm

In [6]:
```
cropped_dataset = labels[['file_path', 'tb_positive']]
cropped_dataset
```

Out[6]:

	file_path	tb_positive
0	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
1	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
2	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
3	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
4	/home/ngsci/datasets/tb-wellgen-smear/images/0...	1
...	...	...
75082	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
75083	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
75084	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
75085	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0
75086	/home/ngsci/datasets/tb-wellgen-smear/images/0...	0

75087 rows × 2 columns

In [7]:
```
cropped_dataset.to_csv('tb_csv.csv',index=True)
```

In [8]:
```
import torchvision
from torchvision import datasets, transforms
```

The transform is used to get the data into a format that's easier to work with, and the raw_tb_pos is used to map the labels of each datapoint to the data in PyTorch.

In [9]:
```
transform = torchvision.transforms.Compose([
        torchvision.transforms.Resize([224,224]),
```

```
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

In [11]:
```
raw_csv = pd.read_csv("tb_csv.csv")
raw_tb_pos = list(raw_csv['tb_positive'])
```

In [12]:
```
torch_dataset = datasets.ImageFolder("/home/ngsci/datasets/tb-wellgen-smear/images/",
                                     transform = transform,
                                     target_transform = (lambda y: raw_tb_pos[y - 1]))
```

In [13]:
```
dataloader = torch.utils.data.DataLoader(torch_dataset, batch_size=32, shuffle=True)
```

These are test cells to ensure that the code so far is working the way it should be. The warning below can be ignored.

In [14]:
```
images, labels = next(iter(dataloader))
```

In [15]:
```
labels[0]
```

Out[15]:
```
tensor(0)
```

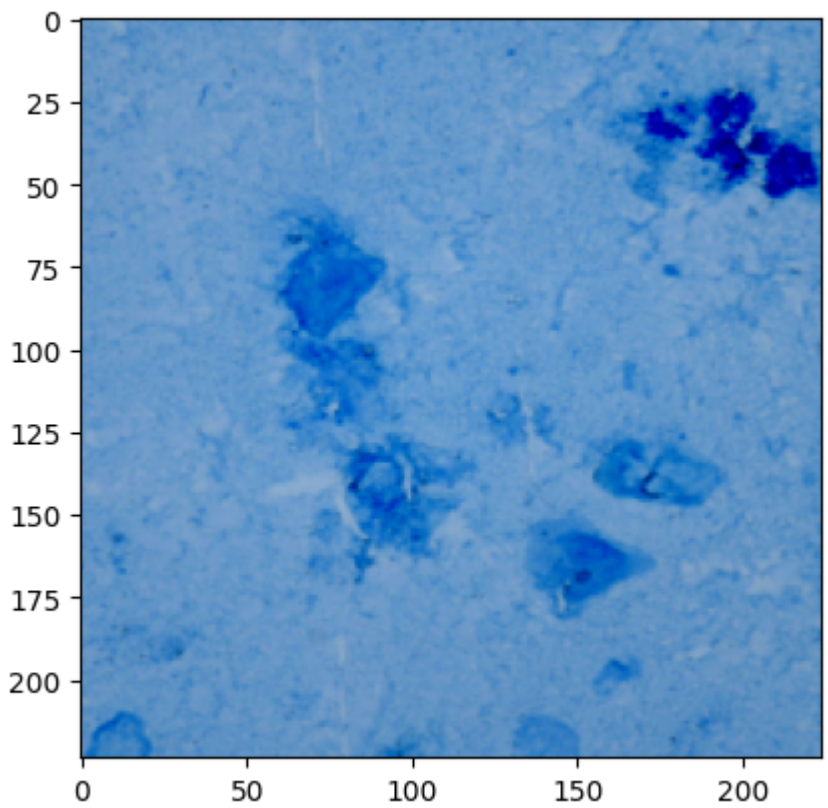
In [16]:
```
plt.imshow(images[0].T)
```

```
/tmp/ipykernel_215/1109450127.py:1: UserWarning:

The use of `x.T` on tensors of dimension other than 2 to reverse their shape is deprecated and it will throw an erro
r in a future release. Consider `x.mT` to transpose batches of matrices or `x.permute(*torch.arange(x.ndim - 1, -1,
-1))` to reverse the dimensions of a tensor. (Triggered internally at ../aten/src/ATen/native/TensorShape.cpp:3571.)

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Out[16]:
```
<matplotlib.image.AxesImage at 0x7f0121470250>
```



Note: We decided to work with a much smaller sample of the data because working with the entire dataset of 75k images took too long to run. In order to ensure reproducibility, we provided a random seed below.

In [18]:
```
torch.manual_seed(132)
np.random.seed(132)
```

In [19]:
```
small_subset = np.random.choice(len(raw_tb_pos), 1000)
```

In [20]:
```
torch_subset = torch.utils.data.Subset(torch_dataset, small_subset)
torch_subset
```

Out[20]:
```
<torch.utils.data.dataset.Subset at 0x7f012081f4f0>
```

The cells below make the train-test split for the smaller subset.

In [28]:
```
subset_dataloader = torch.utils.data.DataLoader(torch_subset, batch_size=32, shuffle=True)
```

In [29]:
```
gen = torch.Generator()
tb_train_partition = torch.utils.data.random_split(torch_subset,
                                                   [0.8, 0.2],
                                                   generator = gen)

tb_train = tb_train_partition[0]
tb_val = tb_train_partition[1]
```

In [31]:
```
len(tb_train), len(tb_val)
```

Out[31]: (800, 200)

The code below is the setup for the neural network. There are two convolutional layers for processing the image data, with adaptive max pooling after each convolutional layer. The Sigmoid activation function and the cross-entropy loss are used because a probability is being predicted as the output.

In [33]:
```
class Conv_Net(nn.Module):
    def __init__(self,padding=1,stride=1,activation="Sigmoid"):
        super().__init__()
        self.padding = padding
        self.stride = stride
        # in channels, out channels, kernel size
        self.conv1 = nn.Conv2d(3, 16, 5, stride = self.stride, padding = self.padding)
        self.pool1 = nn.AdaptiveMaxPool2d(16)
        self.conv2 = nn.Conv2d(16, 8, 5, stride = self.stride, padding = self.padding)
        self.pool2 = nn.AdaptiveMaxPool2d(16)

        self.fc1 = nn.Linear(128 * 4 * 4, 128)
        self.fc2 = nn.Linear(128, 16)
        self.fc3 = nn.Linear(16, 2)
        if activation == "ReLU":
            self.activation = nn.functional.relu
        elif activation == "Sigmoid": # Default for binary tb data
            self.activation = nn.functional.sigmoid
        elif activation == "Softmax":
            self.activation = nn.functional.softmax
        else:
            raise Exception("Please input valid activation function")

    def forward(self, X):
        temp = self.pool1(self.activation(self.conv1(X)))
        temp2 = self.pool2(self.activation(self.conv2(temp)))
        # print(temp2.shape)
        flattened_data = torch.flatten(temp2,1)
        # print(flattened_data.shape)
        temp3 = self.activation(self.fc1(flattened_data))
        temp4 = self.activation(self.fc2(temp3))
        out = self.fc3(temp4)
        return out
```

In [34]:
```
tb_epochs = 10
```

In [35]:
```
cnn = Conv_Net()
tb_batch_size = 64
cnn_criterion = nn.CrossEntropyLoss()
```

In [36]:
```
tb_train_dataloader = torch.utils.data.DataLoader(tb_train,
                                                  batch_size = tb_batch_size,
                                                  shuffle = True,
                                                  num_workers = 1)

tb_val_dataloader = torch.utils.data.DataLoader(tb_val,
                                                batch_size = tb_batch_size,
                                                shuffle = True,
                                                num_workers = 1)
```

In [37]:
```
tb_lr = [0.01] # only using 1 arbitrarily chosen value for less time/ease of computation

tls_for_all_lr = []
tas_for_all_lr = []

vls_for_all_lr = []
vas_for_all_lr = []
```

In [40]:
```
for r in tb_lr:
    cnn = Conv_Net()
    tb_optimizer = torch.optim.SGD(cnn.parameters(), lr = r)

    tb_tls_per_epoch = [] # Training losses per epoch for training set
    tb_tas_per_epoch = [] # Training accuracies per epoch for training set

    tb_vls_per_epoch = [] # Validation losses per epoch for training set
    tb_vas_per_epoch = [] # Validation accuracies per epoch for training set

    for epoch in range(tb_epochs):
        print("Learning Rate:", r, "Epoch: ",epoch)
        training_losses = []
        num = 0
        denom = 0
        for i, data in enumerate(tb_train_dataloader, 0):
            print(i)
            X, y = data
            denom += X.shape[0]
```

```
            # Zero gradient
            tb_optimizer.zero_grad()

            cnn_probs = cnn(X)
            # print(cnn_probs.float(), y)
            loss = cnn_criterion(cnn_probs, y)
            cnn_preds = torch.argmax(cnn_probs, dim = 1)

            loss.backward()
            tb_optimizer.step()
            training_losses.append(loss.item())
            num += torch.sum(torch.round(cnn_preds) == y).item()

        tb_tls_per_epoch.append(np.mean(training_losses)) # mean loss
        tb_tas_per_epoch.append(num / denom)

        print(tb_tls_per_epoch, tb_tas_per_epoch)

        val_num = 0
        val_denom = 0
        val_losses = []

        for v, data in enumerate(tb_val_dataloader, 0):
            v_X, v_y= data
            val_denom += v_X.shape[0]

            cnn.eval()

            cnn_vprobs = cnn(v_X)
            cnn_vpreds = torch.argmax(cnn_vprobs, dim = 1)

            v_loss = cnn_criterion(cnn_vprobs,v_y).item()
            val_losses.append(v_loss)
            val_num += torch.sum(torch.round(cnn_vpreds) == v_y).item()

            cnn.train()
        tb_vls_per_epoch.append(np.mean(val_losses))
        tb_vas_per_epoch.append(val_num / val_denom)

        print("Finished Epoch", epoch + 1, ", training loss:", np.mean(training_losses), "Learning Rate:",r)

    tls_for_all_lr.append(tb_tls_per_epoch)
    tas_for_all_lr.append(tb_tas_per_epoch)

    vls_for_all_lr.append(tb_vls_per_epoch)
    vas_for_all_lr.append(tb_vas_per_epoch)
```

```
Learning Rate: 0.01 Epoch:  0
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821] [0.88125]
Finished Epoch 1 , training loss: 0.4235632144487821 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  1
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852] [0.88125, 0.88125]
Finished Epoch 2 , training loss: 0.4047178396811852 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  2
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867] [0.88125, 0.88125, 0.88125]
Finished Epoch 3 , training loss: 0.38313758144011867 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  3
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846] [0.88125, 0.88125, 0.88125, 0.8812
5]
Finished Epoch 4 , training loss: 0.374054505274846 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  4
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846, 0.3742868097928854] [0.88125, 0.881
25, 0.88125, 0.88125, 0.88125]
Finished Epoch 5 , training loss: 0.3742868097928854 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  5
0
1
2
3
```

```
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846, 0.3742868097928854, 0.3690384374215
0527] [0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125]
Finished Epoch 6 , training loss: 0.36903843742150527 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  6
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846, 0.3742868097928854, 0.3690384374215
0527, 0.36902662882438075] [0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125]
Finished Epoch 7 , training loss: 0.36902662882438075 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  7
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846, 0.3742868097928854, 0.3690384374215
0527, 0.36902662882438075, 0.36336910266142625] [0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88
125]
Finished Epoch 8 , training loss: 0.36336910266142625 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  8
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846, 0.3742868097928854, 0.3690384374215
0527, 0.36902662882438075, 0.36336910266142625, 0.3667607307434082] [0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.
88125, 0.88125, 0.88125, 0.88125]
Finished Epoch 9 , training loss: 0.3667607307434082 Learning Rate: 0.01
Learning Rate: 0.01 Epoch:  9
0
1
2
3
4
5
6
7
8
9
10
11
12
[0.4235632144487821, 0.4047178396811852, 0.38313758144011867, 0.374054505274846, 0.3742868097928854, 0.3690384374215
0527, 0.36902662882438075, 0.36336910266142625, 0.3667607307434082, 0.36392112190906817] [0.88125, 0.88125, 0.88125,
0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125, 0.88125]
Finished Epoch 10 , training loss: 0.36392112190906817 Learning Rate: 0.01
```

In [44]:
```
vas_for_all_lr
```

Out[44]:
```
[[0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85]]
```

After 10 epochs on a learning rate of 0.01, on a random subset of 1000 images, the CNN above achieved a validation accuracy across 0.85. However, the algorithm could just be predicting 0 for all of the individuals in each batch (as there are very little data points in our batch with labels of 1), so examining sensitivity and/or specificity are probably better metrics of algorithm performance. The CNN above is just a rudimentary algorithm for prediction, and if our group were to further develop this algorithm, we would examine these metrics as well and try to increase them.

In [49]:
```
subset_labels = [raw_tb_pos[i] for i in small_subset]
sum(subset_labels) / len(subset_labels) # prints proportion of people in the subset with Y = 1
```

Out[49]:
```
0.049
```

Value of our Algorithm

The value of the algorithm is in how much it can reduce the time and cost it takes for a patient living in a place with limited access to healtcare to go from developing a cough to being diagnosed with TB.

There is two ways this algorithm can generate value:

1. In regions with limited access to healthcare, the algorithm can expedite the process of diagnosing TB smears, decreasing the number of doctors and cost of treating villages with TB.
2. If this algorithm is deployed in conjunction with a mobile clinic set up, it could reduce the number of visits required to diagnose someone with TB - rather than wait for the sputum sample to be sent to a lab to be analyzed, it can be processed at the clinic and the algorithm can flag samples of concern.

In summary, this algorithm, if funded and developed fully, has the potential to benefit real patients by facilitating early detection and intervention. It can potentially address inefficiencies in the healthcare system related to delayed or missed diagnoses. By automating the detection process, it helps overcome resource constraints, streamlines workflows, and ensures that healthcare professionals can focus their attention on confirmed or high-risk cases. This, in turn, can improve health outcomes, reduce the severity of the disease, and enhance the overall quality of life for individuals affected by TB, especially in areas where there are limited health resources.