Privacy-preserving Machine Learning using Secure Multiparty Computation for Medical Image classification

Report submitted in fulfillment of the requirements for the B.Tech Project of

Fourth Year B.Tech.

by

Shreyansh Singh, 16075052

Under the guidance of

Prof. K.K. Shukla

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY (BHU) VARANASI

Varanasi 221005, India

November 2019

Dedicated to

My parents, teachers

Declaration

We certify that

1. The work contained in this report is original and has been done by our team and the general supervision of my supervisor.
2. The work has not been submitted for any project.
3. Whenever we have used materials (data, theoretical analysis, results) from other sources, we have given due credit to them by citing them in the text of the thesis and giving their details in the references.
4. Whenever we have quoted written materials from other sources, we have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT (BHU) Varanasi
Date:

Shreyansh Singh, B.Tech.
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Certificate

*This is to certify that the work contained in this report entitled "**Privacy-preserving Machine Learning using Secure Multiparty Computation for Medical Image classification**" being submitted by **Shreyansh Singh (Roll No. 16075052)** and carried out in the Department of Computer Science and Engineering, Indian Institute of Technology (BHU) Varanasi, is a bona fide work of my supervision.*

Place: IIT (BHU) Varanasi
Date:

Prof. K.K. Shukla
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and our institute. I would like to extend my sincere thanks to all of them. I am highly indebted to my project supervisor Prof. K.K. Shukla for his guidance and constant supervision as well as for providing necessary information regarding the project and also for his support in completing the project.

Place: IIT (BHU) Varanasi
Date: **Shreyansh Singh**

Contents

List of Figures

List of Tables

Abstract

With the rising use of Machine Learning and Deep Learning in various industries, the Medical industry is also not far behind. A very simple yet extremely important use case of Machine Learning in this industry is for image classification. Detecting certain diseases timely i.e in the earlier stages, is something which doctors can miss and in many cases actually do. In such cases, Deep Learning can assist doctors in determining if the patient actually has the disease. However when using automated system like these, there is a privacy concern as well. For example if we consider that a group of hospitals use a centralised ML model, attackers have the capability to infer whether or not an individual is a patient at the hospital and is suffering from that disease or not, basically violating their right to privacy. Other than that, we also want the model to be secure, and the data that is sent to the model and the predictions that are received both should not be revealed to the model in clear text.

In this study, we aim to solve these problems in the context of a medical image classification problem, the problem we have considered is detection of pneumonia by examining chest x-ray images. We first train the model on a public dataset, secure and serve the machine learning model as a server and query the secured model to receive private predictions. We have trained three different models and all of them have a very high accuracy, the best one having around 95% accuracy.

Introduction

The use of automation in the form of Machine Learning is becoming more visible in the medical industry by the day. However, letting the computers handle confidential medical information of the patients comes with the danger of attackers getting access to those and probably misusing them. Hence, security and privacy should be an important aspect when setting up such a system.
To address this, privacy-preserving machine learning (or private machine learning) was introduced. Private machine learning is a combination of cryptography, machine learning and distributed systems. This requires a deep understanding of cryptography concepts like Secure Computation, Differential Privacy and most importantly how these can be integrated with machine learning to perform privacy-preserving predictions.

Secure Multiparty Computation

Secure multi-party computation (SMPC) also known as secure computation is a sub-field of cryptography where the goal is to create a provision for parties to jointly compute a function over their inputs which are kept private i.e. not shared with the other parties. This model is different from traditional cryptography because of the fact that here the information is to be protected from the other participants instead of from an adversary who is outside the system. A basic understanding of SMPC can be obtained from Shamir's Secret Sharing Scheme ([Shamir 1979]). The purpose of that scheme is to divide and distribute one secret value among the participants. A subset of the participants must pool their data to retrieve the secet value. Shamir's scheme can also be used on a secret shared value to perform some computation. The result of every participant's computation (on their own data) can be grouped together to get the required outcome without ever revelaing the secret inputs.

The definition of an MPC task involves defining -

- **Functionality** - What is needed to be computed?
- **Security type** - How strong of a protection is required?
- **Adversarial model** - What do we want to protect against?
- **Network model** - In which setting will it be done?

The functionality is the code of the trusted party. Security type is of 3 types - *Computational, Statistical* and *Perfect*. The Adversarial model can be described in different ways -

- **Adversarial behavior**
  - *Semi honest* - honest-but-curious. corrupted parties follow the protocol honestly, External adversary A tries to learn more information. Models inadvertent leakage
  - *Fail stop* - same as semi honest, but corrupted parties can prematurely halt. Models crash failures
  - *Malicious* - corrupted parties can deviate from the protocol in an arbitrary way
- **Adversarial power**
  - *Polynomial time* - computational security, normally requires cryptographic assumptions, e.g., encryption, signatures, oblivious transfer
  - *Computationally unbounded* - an all-powerful adversary, information-theoretic security
- **Adversarial corruption**
  - *Static* - the set of corrupted parties is defined before the execution of the protocol begins. Honest parties are always honest, corrupted parties are always corrupted
  - *Adaptive* - Adversary A can decide which parties to corrupt during the course of the protocol, based on information it dynamically learns
  - *Mobile* - Adversary A can "jump" between parties Honest parties can become corrupted, corrupted parties can become honest again
- **Number of corrupted parties**
  - Denote by $t \leq n$ an upper bound on corruptions
    * No honest majority, e.g., two-party computation
    * Honest majority, i.e., $t < n/2$
    * Two-thirds majority, i.e., $t < n/3$
  - *General adversary structure* - Protection against specific subsets of parties

The communication/network model can of the following types -

- **Point-to-point**: fully connected network of pairwise channels.
  - Unauthenticated channels
  - *Authenticated channels*: in the computational setting
  - *Private channels*: in the IT setting
- **Broadcast** - additional broadcast channel

SMPC gives a combination of encryption, distribution and distributed combination and this has a big impact on data security and data privacy.

Differential Privacy

Differential Privacy (DP) is a rigorous mathematical framework which allows sharing information about a dataset publicly by describing the patterns of groups of the dataset without revealing information about the individuals in the dataset. An algorithm is said to be differentially private if and only if the inclusion of any one instance in the training dataset causes only statistically minor changes to the output of the algorithm. This is required in situations where, for example, the identity of a patient (in the medical context) is to be kept private. If not for DP, using just the trained ML model, attackers would have been capable of finding out the hospital a specific patient belonged to which would violate their right to privacy. The role of DP here is to limit the attacker's ability to infer such membership by putting a theoretical limit on the influence that a single individual can have.

However using DP means that there will be a tradeoff between accuracy and security. Although the aim of DP is to minimise the "information leak" from a single query, but keeping this value small enough when multiple queries are made can become a challenge as for every query, the total "information leak" will increase. As a solution, more noise has to be injected in the data to minimise the privacy leakage but that would mean the accuracy of the model will go down. This can be a big problem when training complex ML models.

Motivation of the Research Work

Although the use of Deep Learning in the medical industry can aid doctors as well as patients in getting faster diagnosis, it must also be secured against attackers. It becomes even more critical as the information at stake is the confidential medical and personal information of the patients. The telemedicine/telehealth field, which involves distribution of health-related services and information via electronic information and telecommunication technologies will also benefit from our aim of providing private predictions since in these scenarios, the patients are not physically present with the doctor and private medical data (the images) will have to be sent to these ML models remotely. This research aims to solve this problem by providing private predictions when querying the model (which is also be encrypted) and adding an even stronger layer of privacy through the use of differential privacy. Prediction of pneumonia from the chest x-ray images is the problem that this research targets specifically.

Organization of the Report

The organization of the report is as follows:
Chapter 1 gives a description of the past work that has been done in the domain of secure multiparty computation, differential privacy, privacy-preserving machine learning and medical image classification and the research papers we have gone through as a prerequisite for our study.
Chapter 2 focuses on the dataset we have used and the preprocessing steps.

Chapter 3 provides the implementation details.
Chapter 4 discusses the results we obtained for every step in our study.
Chapter 5 gives an analysis of our results at every step.
Finally, we conclude our report in Chapter 6, and specify our future work.

Chapter 1

Literature Review

1.1 Introduction

We followed the framework proposed by [Vom Brocke 2009] for the literature review process. The first step involved defining the scope and creating a rough outline of the task to perform. This was followed by thorough literature survey and subsequent analysis of the work already done this field. Following these steps helped us to identify the research gaps that existed, and helped to formulate the research questions that we will attempt to answer through our work.

The literature review was conducted using exhaustive search over the following terms: "secure multiparty computation", "differential privacy", "privacy preserving machine learning" and "secure deep learning". Apart from keyword search and relevance, other selection criteria were the chronology of the papers and the quality of sources (peer reviewed journals and conferences).

The search engines utilized for this search were mainly the LTU library search and Google scholar search engines which aggregate results over a number of databases. The majority of the references comes from well known databases such as ACM, IEEE, Springer and Elsevier.

1.2 Secure Computation

Secure Multiparty computation (SMC) can be divided into two broad classes - Two-party computation and Multi-party computation.

1.2.1 Two-party computation

[Yao 1986] first introduced the idea of two-party computation (2PC). The idea of Yao's garbled circuits were introduced in [Goldreich 1987] although it was heavily based on [Yao 1986]. Yao's garbled circuits facilitates two-party secure computation in which two mistrusting parties can jointly evaluate a function over their private inputs without the presence of a trusted third party. Yao's basic protocol is secure against semi-honest adversaries. 2PC protocols in a malicious setting (secure against active adversaries) were proposed a bit later in [Lindell 2007], [Ishai 2008] and [Nielsen 2009]. A solution which works with committed inputs explicitly was given by [Jarecki 2007].

1.2.2 Multi-party computation

Secret sharing forms the fundamentals of multi-party computation (MPC). The two most commonly used methods are Shamir's secret sharing ([Shamir 1979]) and additive secret sharing. There has been a lot of work on using MPC with secret sharing schemes. One of the most popular is SPDZ ([Damgård 2012]). This uses additive secret shares and is secure against active adversaries (malicious, dishonest majority). Some other implementations of secure MPC protocols exist like [Demmler 2015], [Zahur 2015], [SCALE-MAMBA] and [FRESCO]. These however are independent frameworks that do not help much in Machine Learning in terms of integration with current ML platforms and that they simply provide implementations of the SMC protocols rather than focus on private machine learning. [Wagh 2019] is an SMC framework which provides efficient 3-party protocols tailored for state-of-the-art neural networks. Other such frameworks include SecureML ([Mohassel 2017]), GAZELLE ([Juvekar 2018]) and ABY3 ([Mohassel 2018]). These frameworks focus on adapting secure computation protocols to private machine learning. Crypten ([Facebook Research]) is a recent framework developed by Facebook Research for privacy preserving machine learning on Pytorch but is still quite limited in terms of the features it offers from the deep learning perspective. TF encrypted ([Dahl 2018]) is a framework which provides secure multi-party computation directly in TensorFlow. We use TF encrypted as the framework for SMC in our research.

1.3 Differential Privacy

Differential Privacy (DP) is a rigorous mathematical framework which allows sharing information about a dataset publicly by describing the patterns of groups of the dataset without revealing information about the individuals in the dataset. In [Shokri 2016] the authors showed that if, for example, an attacker gets access to an ML model being used in a hospital dealing with private medical information of patients, the attackers can infer whether an individual was a patient at the hospital or not, thus vioalting their right to privacy. DP can be formally stated as in [Dwork 2008] -

Definition 1.1. A randomized mechanism K provides (ϵ, δ) - differential privacy if for any two neighboring database D_1 and D_2 that differ in only a single entry, $\forall S \subseteq Range(K)$,

$$\Pr(K(D_1) \in S) \leq e^{\epsilon}\Pr(K(D_2) \in S) + \delta \tag{1.1}$$

If $\delta = 0$, K is said to satisfy ϵ-differential privacy.

The idea is that to achieve DP, noise is added to the algorithm's output. This noise is depends on the sensitivity of the output, where sensitivity is the measure of the maximum change of output due to the inclusion of a single data instance [Truex 2018].

Two popular mechanisms for achieving DP are the Laplacian and Gaussian mechanisms. When an algorithm requires multiple additive noise mechanisms,the evaluation of the privacy guarantee follows from the basic composition theorem [Dwork 2006], [Dwork 2009] or from advanced composition theorems and their extensions [Bun 2016], [Dwork 2016].

As a tool that could integerate the use of Differential Privacy with Machine Learning, Tensorflow Privacy ([McMahan 2018]) was introduced. Tensorflow Privacy is a library that includes implementations of TensorFlow optimizers for training machine learning models with differential privacy.

Chapter 2

Data Collection and Analysis

Medical image classification includes a vast array of problems to work on. This research takes up the task of detecting pneumonia in patients by analysing their chest X-ray images. The dataset is obtained from [Kermany 2018], a research published in the scientific journal *Cell*, where the authors have collected such medical images and aim to apply image-based deep learning to detect such diseases. A copy of the dataset is also available on Kaggle [Paul Mooney].

Figure 2.1: Illustrative Examples of Chest X-Rays in Patients with Pneumonia, [Kermany 2018]

Figure 2.1 shows the variation in the x-rays for the different kinds of pneumonia. According to [Kermany 2018], the normal chest X-ray (left panel) depicts clear lungs without any areas of abnormal opacification in the image. Bacterial pneumonia (middle) typically exhibits a focal lobar consolidation, in this case in the right upper lobe (white arrows), whereas viral pneumonia (right) manifests with a more diffuse "interstitial" pattern in both lungs.

2.1 Description of the dataset

The dataset is organised into 3 folders (train, test, val) and contains sub folders for each image category (Pneumonia/Normal). There are 5862 X-ray images (all in JPEG format) and 2 categories (Pneumonia/Normal).

Chest X-ray images (anterior-posterior) were selected from retrospective cohorts of pediatric patients of one to five years old from Guangzhou Women and Children's Medical Center, Guangzhou. All chest X-ray imaging was performed as part of patients' routine clinical care.

For the analysis of chest X-ray images, all chest radiographs were initially screened for quality control by removing all low quality or unreadable scans. The diagnoses for the images were then graded by two expert physicians before being cleared for training any AI system. In order to account for any grading errors, the evaluation set was also checked by a third expert. [Kermany 2018]

The type of the image, i.e. Pneumonia or Normal can be identified from the filename as well. The Normal images have the "NORMAL" keyword in them. The ones indicating Pneumonia have the keywords "virus" or "bacteria" in them indicating the type. In our experiments however, we haven't considered the type of Pneumonia in the classification. It is just Normal X-ray images or the ones with Pneumonia.

Breakdown of the data for classification -

- **Train data** - 1341 normal images and 3882 pneumonia images
- **Test data** - We merge the images in the test and val folders because there were only 18 images in the test folder. After merging, there were 234 normal images and 390 pneumonia images.

2.2 Data Preprocessing

We have done two different types of preprocessing of the images in order to perform different experiments.

2.2.1 Preprocessing Technique 1

For each image in the training and testing data, we perform the following steps -

1. Resize the image into size (125, 150) using *resize* function of the Pillow library
2. Convert the image into greyscale
3. Divide each pixel value by 255
4. Save each image as a numpy array along with their labels (0 or 1 for Normal or Pneumonia respectively)

2.2.2 Preprocessing Technique 2

For each image in the training and testing data, we perform the following steps -

1. Resize the image into size (224, 224) using *cv2.INTER_ CUBIC*, i.e. cubic interpolation using the OpenCV library in Python
2. Convert the image into a numpy array and also store the corresponding labels

We save both these arrays as pickle files to be later used with our deep learning models.

Chapter 3

Implementation Details

This section will highlight the libraries used and the implementation details. The code is written in Python language.

3.1 Libraries Used

Following are the major libraries used in the project -

- **Keras**: Deep Learning library for creating and training the model
- **tf-encrypted**: For encrypted machine learning using TensorFlow. It uses SMC to provide private predictions using the trained Keras model.
- **TensorFlow Privacy**: To train models with Differential Privacy
- **Scikit-learn**: For data preprocessing functions like shuffle
- **Numpy**: For storing the train and test data
- **cv2**: For image pre-processing
- **PIL**: For image pre-processing
- **Matplotlib**: For visualisation in the form of plots and graphs
- **Pickle**: For saving the processed images as numpy arrays

3.2 Description of functions

3.2.1 Model Training

We use Keras to train deep learning models. A code snippet showing the model definition (a sample model) , model compilation, model training and model evaluation is shown below

```
# Define model
model = tf.keras.Sequential([
          tf.keras.layers.Conv2D(16, 8,
                                 strides=2,
                                 padding='same',
                                 activation='relu',
                                 input_shape=(150, 125, 1)),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(32, activation='relu'),
          tf.keras.layers.Dense(2, activation='softmax')
  ])


# Define constants
batch_size = 32
epochs = 40


# Save model checkpoint for best model yet in terms of highest validation accuracy
from keras.callbacks import ModelCheckpoint
mcp = ModelCheckpoint(filepath='./models/model_simple.h5',monitor="val_acc", save_best_only=True, save_weights_only=False)

# Compile the model
model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              metrics=['accuracy'])

# Train the model
hist = model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=1,
           callbacks=[mcp],
           validation_data=(x_test, y_test))

# Evaluate model on test data
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

3.2.2 Model Training with Differential Privacy

Along with Keras, we also have to use TensorFlow Privacy to facilitate training with differential privacy. The code snippet for that is shown below

```
from privacy.analysis.rdp_accountant import compute_rdp
from privacy.analysis.rdp_accountant import get_privacy_spent
from privacy.optimizers.dp_optimizer import DPGradientDescentGaussianOptimizer

dpsgd = True            # If True, train with DP-SGD
learning_rate = 0.15    # Learning rate for training
noise_multiplier = 1.1  # Ratio of the standard deviation to the clipping norm
l2_norm_clip = 1.0      # Clipping norm
batch_size = 250        # Batch size
epochs = 20             # Number of epochs
microbatches = 50       # Number of microbatches
```

```python


def compute_epsilon(steps):
    """Computes epsilon value for given hyperparameters."""
    if noise_multiplier == 0.0:
        return float('inf')
    orders = [1 + x / 10. for x in range(1, 100)] + list(range(12, 64))
    sampling_probability = batch_size / 60000
    rdp = compute_rdp(q=sampling_probability,
                      noise_multiplier=noise_multiplier,
                      steps=steps,
                      orders=orders)
    return get_privacy_spent(orders, rdp, target_delta=1e-5)[0]


tf.logging.set_verbosity(tf.logging.INFO)
if dpsgd and batch_size % microbatches != 0:
    raise ValueError('Number of microbatches should divide evenly batch_size')

model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(16, 8,
                             strides=2,
                             padding='same',
                             activation='relu',
                             input_shape=(28, 28, 1)),
      tf.keras.layers.AveragePooling2D(2, 1),
      tf.keras.layers.Conv2D(32, 4,
                             strides=2,
                             padding='valid',
                             activation='relu'),
      tf.keras.layers.AveragePooling2D(2, 1),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(32, activation='relu'),
      tf.keras.layers.Dense(10)
  ])

if dpsgd:
    optimizer = DPGradientDescentGaussianOptimizer(
        l2_norm_clip=l2_norm_clip,
        noise_multiplier=noise_multiplier,
        num_microbatches=microbatches,
        learning_rate=learning_rate)
    # Compute vector of per-example loss rather than its mean over a minibatch.
    loss = tf.keras.losses.CategoricalCrossentropy(
        from_logits=True, reduction=tf.losses.Reduction.NONE)
else:
    optimizer = tf.optimizers.SGD(learning_rate=learning_rate)
    loss = tf.keras.losses.CategoricalCrossentropy(from_logits=True)

# Compile model with Keras
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

# Train model with Keras
model.fit(train_data, train_labels,
        epochs=epochs,
        validation_data=(test_data, test_labels),
        batch_size=batch_size)

# Compute the privacy budget expended.
if dpsgd:
    eps = compute_epsilon(epochs * 60000 // batch_size)
    print('For delta=1e-5, the current epsilon is: %.2f' % eps)
else:
    print('Trained with vanilla non-private SGD optimizer')
```

3.2.3 Model accuracy and Model loss

A visualisation of the model accuracy and model loss with the epochs can be generated from the following snippet

```
# Model Accuracy

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['acc'], color='red')
ax.plot(hist.history['val_acc'], color ='green')
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()


# Model Loss

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['loss'], color='red')
ax.plot(hist.history['val_loss'], color ='green')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```

3.2.4 Model Serving

After training the model with normal Keras, we use TF Encrypted (TFE) to provide private predictions. To secure and serve this model, we will need three TFE servers. This is because TF Encrypted under the hood uses an encryption technique called multi-party computation (MPC). Secure-NN ([Wagh 2019]) is used as the underlying protocol. The idea is to split the model weights and input data into shares, then send a share of each value to the different servers. The key property is that if we look at the share on one server, it reveals nothing about the original value (input data or model weights). After the model is encrypted and the weights are shared we have to set up a QueueServer to serve our model.

```
import tf_encrypted as tfe
import tf_encrypted.keras.backend as KE

# Load model weights trained from normal Keras
pre_trained_weights = 'model_simple.h5'
model.load_weights(pre_trained_weights)

# Configure the protocol
players = OrderedDict([
    ('server0', 'localhost:4000'),
    ('server1', 'localhost:4001'),
    ('server2', 'localhost:4002'),
])

```

```python
config = tfe.RemoteConfig(players)
config.save('/tmp/tfe.config')

# Define the protocol to use - SecureNN
tfe.set_config(config)
tfe.set_protocol(tfe.protocol.SecureNN())

# Run in separate terminals to launch TFE servers
for player_name in players.keys():
    print("python3.6 -m tf_encrypted.player --config /tmp/tfe.config {}".format(player_name))


# Clone the Keras model to a TFE model
tf.reset_default_graph()
with tfe.protocol.SecureNN():
    tfe_model = tfe.keras.models.clone_model(model)


# Set up a new tfe.serving.QueueServer for the shared TFE model
q_input_shape = (1, 150, 125, 1)
q_output_shape = (1, 2)

server = tfe.serving.QueueServer(
    input_shape=q_input_shape, output_shape=q_output_shape, computation_fn=tfe_model
)


sess = KE.get_session()

# Wait for incoming requests for predictions and provide predictions when they arrive
request_ix = 1

def step_fn():
    global request_ix
    print("Served encrypted prediction {i} to client.".format(i=request_ix))
    request_ix += 1

# num_steps=3

server.run(
    sess,
    step_fn=step_fn)
```

3.2.5 Private Prediction for Client

We can now request private predictions. The following snippet shows it.

```python
# Load the saved config file
config = tfe.RemoteConfig.load("/tmp/tfe.config")

# Set the config for TFE
tfe.set_config(config)
tfe.set_protocol(tfe.protocol.SecureNN())

# Set up tfe.serving.QueueClient
input_shape = (1, 150, 125, 1)
output_shape = (1, 2)

client = tfe.serving.QueueClient(
    input_shape=input_shape,
    output_shape=output_shape)

```

```
# Set the config for the session
sess = tfe.Session(config=config)


# Query the model
# User inputs
num_tests = 25
images, expected_labels = x_test[100:num_tests+100], y_test[100:num_tests+100]

for image, expected_label in zip(images, expected_labels):
    # Get predictions form the model
    res = client.run(
        sess,
        image.reshape(1, 150, 125, 1))

    predicted_label = np.argmax(res)
    # Display the results
    print("The image had label {} and was {} classified as {}".format(
        expected_label,
        "correctly" if expected_label == predicted_label else "incorrectly",
        predicted_label))
```

Chapter 4

Results

4.1 Model Performance

In all, we used four models for our experiments. The first two are custom made image classification models. The first of the two used *AveragePooling2D* while the other uses *MaxPooling2D* layer. Both of these accept images preprocessed using the first preprocessing method. The third and fourth models are based on the VGG16 architecture, on of which accepts images preprocessed using the first preprocessing method while the other accepts the images preprocessed using the second preprocessing technique. In order in which the models were described, we will name them DNN1, DNN2, VGG16-1, VGG16-2.

The architectures of the models are shown below -

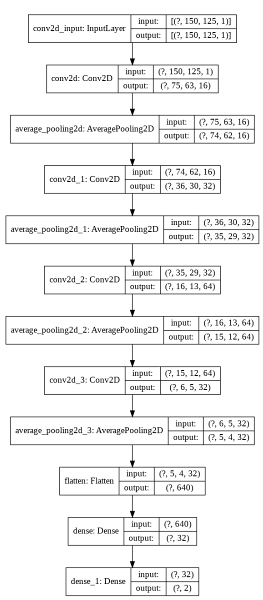

Figure 4.1: DNN with AveragePooling2D

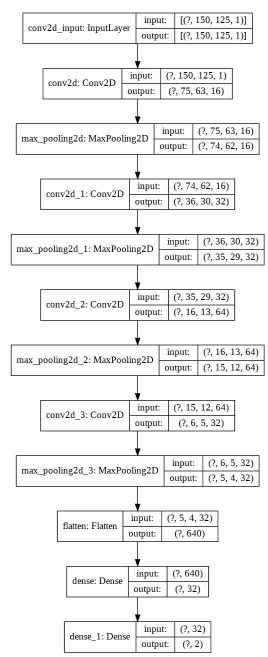

Figure 4.2: DNN with MaxPooling2D

The architecture of the VGG16 based models are also shown below -

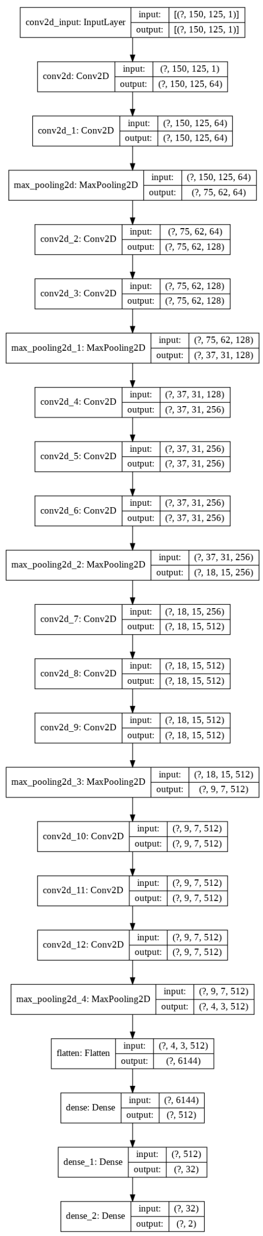

Figure 4.3: VGG16 (Preprocessing - 1)

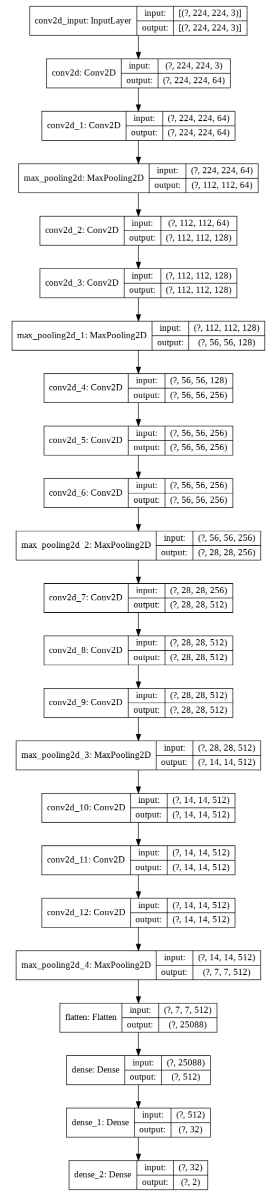

Figure 4.4: VGG16 (Preprocessing - 2)

The accuracy and loss of the models on the dataset is shown below -

Model	Accuracy	Loss
DNN1	0.852	0.33
DNN2	0.873	0.29
VGG16-1	0.809	0.53
VGG16-2	0.964	0.993

Table 4.1: Test set results for 2-class classification

When trained with Differential Privacy, the model accuracies are shown below -

Model	Accuracy
DNN1	0.721
DNN2	0.748
VGG16-1	0.705
VGG16-2	0.853

Table 4.2: Test set results for 2-class classification when training with Differential Privacy

The plot of accuracy-vs-epoch and loss-vs-epoch of each of the models are also shown below -

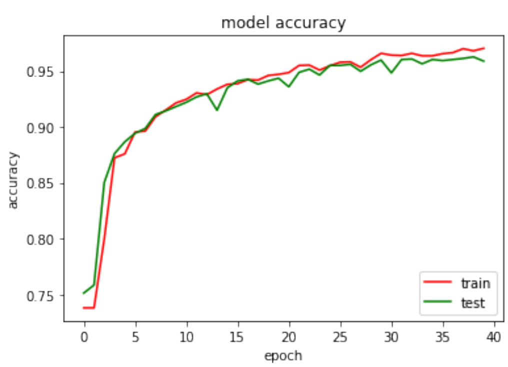

Figure 4.5: DNN1 - Accuracies

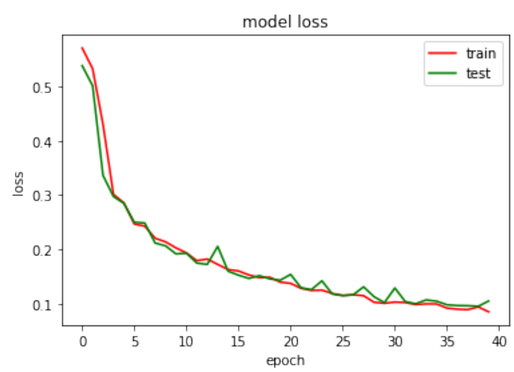

Figure 4.6: DNN1 - Losses

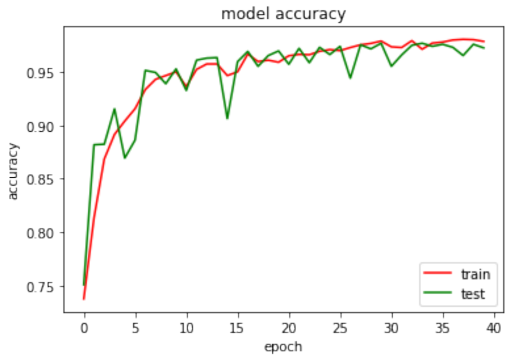

Figure 4.7: DNN2 - Accuracy

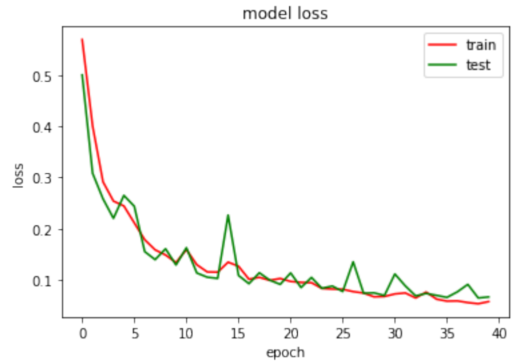

Figure 4.8: DNN2 - Loss

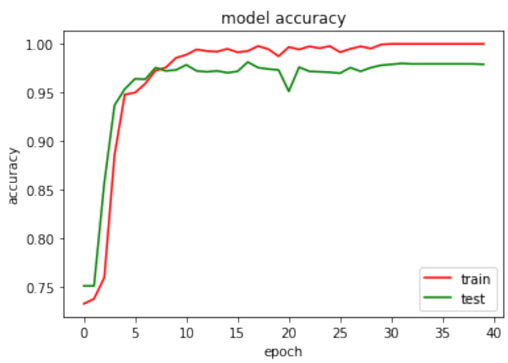

Figure 4.9: VGG16-1 - Accuracy

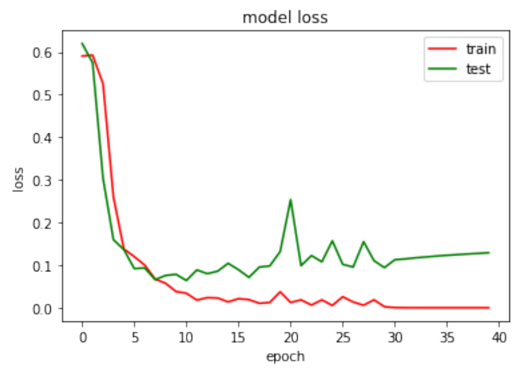

Figure 4.10: VGG16-1 - Loss

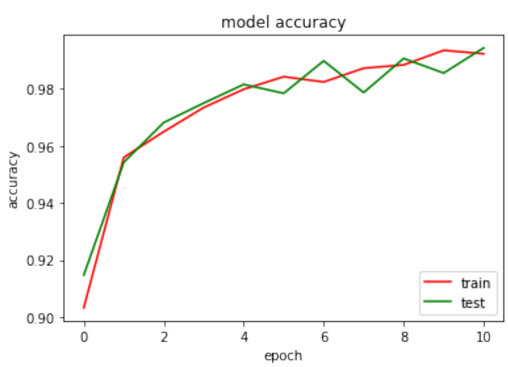

Figure 4.11: VGG16-2 - Accuracy

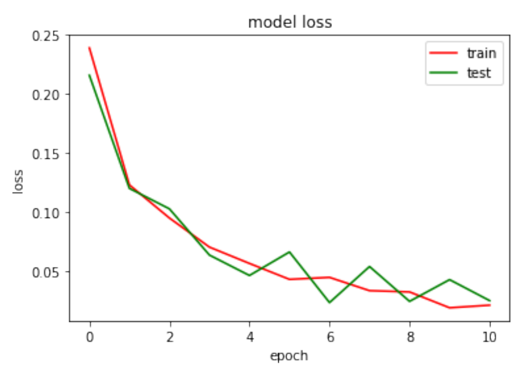

Figure 4.12: VGG16-2 - Loss

For the models trained with DP, the accuracies and losses vary a lot with epochs. We trained DNN1, DNN2 and VGG16-1 with DP, and the plots are shown below -

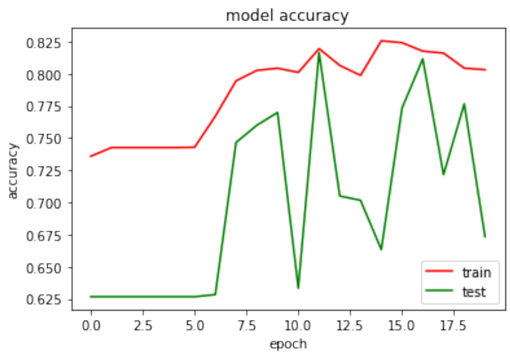

Figure 4.13: DNN1 (with DP) - Accuracies

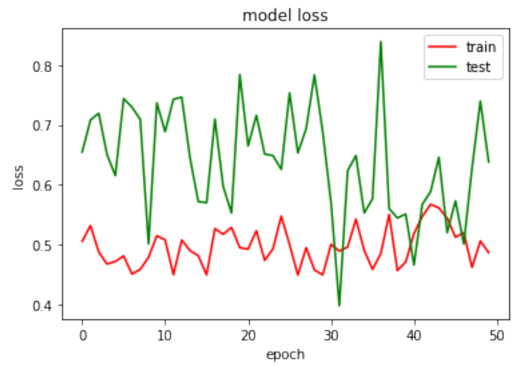

Figure 4.14: DNN1 (with DP) - Losses

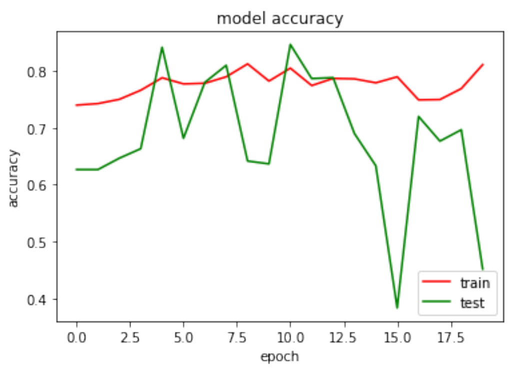

Figure 4.15: DNN2 (with DP) - Accuracy

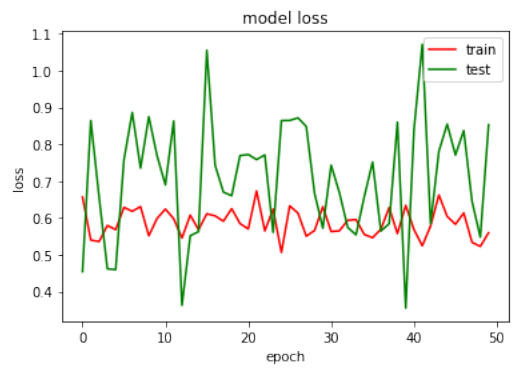

Figure 4.16: DNN2 (with DP) - Loss

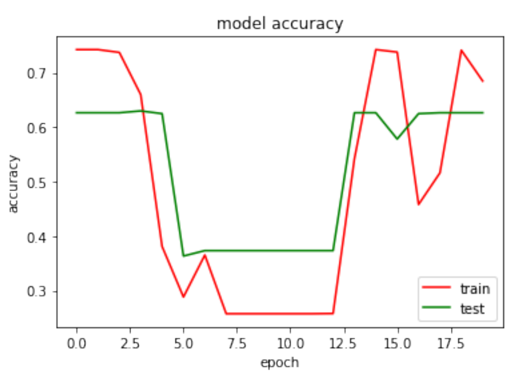

Figure 4.17: VGG16-1 (with DP) - Accuracy

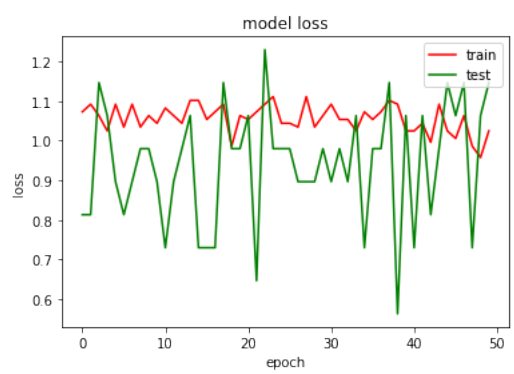

Figure 4.18: VGG16-1 (with DP) - Loss

Chapter 5

Discussion

5.1 Model Evaluation on the data

The VGG16-based model when trained with the images generated using second preprocessing method is the best model in terms of both accuracy (96.41%) and loss (cross-entropy loss = 0.1021) among all the other models. However, the VGG16 model trained on the data generated using first preprocessing method performs worse than the other two classes of models. It achieves an accuracy of 80.9% and a cross entropy loss of 0.53. In case of the other two models, the one with MaxPooling (accuracy = 87.33%, loss = 0.29) performs better than the model with AveragePooling (accuracy = 85.2%, loss = 0.33). All these models were trained for 40 epochs with a ModelCheckpoint (in Keras) to save the best model in terms of validation accuracy and EarlyStopping (in Keras) to stop the training when the model performance tends to decrease. Using Differential Privacy (DP) while training the models, results in a reduced accuracy (84.89%) which should also happen in theory as noise is introduced in the data while training. The accuracy (on an average) reduced by around 12% when training with DP. The DNN1 and DNN2 models were the most affected with reduction in accuracy of 13% where the VGG16 models saw an accuracy drop of 10% and 11% respectively.

5.2 Limitations

TF Encrypted is a fairly recent framework developed in 2018. When cloning Keras models, TF Encrypted usually works really well, however it has some limitations. Some advanced Keras layers are still not implemented in that library. There were some issues with Batch Normalization and Dropout layers as well, as they are not correctly implemented in TFE and gives error while conversion from Keras models. Also, serving predictions from the encrypted models, is much slower than getting predictions from the unencrypted model. Although, this is expected, but for VGG16 it becomes very slow, which becomes very evident if having to be used as a service.

Chapter 6

Conclusion

Our research shows that it is possible to build a system which can help ensure privacy of the users in a very critical setting where confidentiality of the information is of utmost importance. Private medical data is usually sensitive information which the patient can not afford to lose. We developed a system which provides private predictions on a Deep Learning model in which the both the model and the data is encrypted and shared (using MPC), for an image classification problem. At no stage of the prediction process, the user is sharing raw data which can be sniffed by the attacker. Furthermore, if we also use Differential Privacy, we ensure that the model does not memorize sensitive information about the training set. This means that it should not be possible for the attackers to reveal some private information by just querying the deployed model. If the model is not trained using DP, the model is vulnerable to attacks such as [Shokri 2017] and [Fredrikson 2015], which can help attackers get information about the dataset and in this case, medical information about the patient.

6.1 Future Work

There were certain limitations of using TF Encrypted for building more complex models as it currently does not support all the layers implemented in Keras. However, TF Encrypted is also open source which means that we can contribute to adding/correcting those layers ourselves. There was a problem in the Batch Normalisation layer in TFE due to which model weights were not being transferred correctly when cloning the model. We have raised an issue regarding this on GitHub and the authors of the library are currently working on it. Later, we can also start diving into the code to improve the library. Similar improvements can be done for Crypten ([Facebook Research]) which is also at a very nascent stage in terms of development.

Apart form this, we can also experiment with more models provided some of the basic errors in the library are rectified.

Furthermore, we can extend this to a general purpose medical image classification framework as well. Then it can be used to classify other such medical images as well.

Bibliography

[Bun 2016] Mark Bun et Thomas Steinke. *Concentrated Differential Privacy: Simplifications, Extensions, and Lower Bounds*, 2016.

[Dahl 2018] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin et Gavin Uhma. *Private Machine Learning in TensorFlow using Secure Computation*, 2018.

[Damgård 2012] Ivan Damgård, Valerio Pastro, Nigel Smart et Sarah Zakarias. *Multiparty Computation from Somewhat Homomorphic Encryption.* In Reihaneh Safavi-Naini et Ran Canetti, editeurs, Advances in Cryptology – CRYPTO 2012, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Demmler 2015] Daniel Demmler, Thomas Schneider et Michael Zohner. *ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.* In NDSS, 2015.

[Dwork 2006] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov et Moni Naor. *Our Data, Ourselves: Privacy Via Distributed Noise Generation.* In Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, volume 4004 of *Lecture Notes in Computer Science*, pages 486–503. Springer, 2006.

[Dwork 2008] Cynthia Dwork. *Differential Privacy: A Survey of Results.* In Manindra Agrawal, Dingzhu Du, Zhenhua Duan et Angsheng Li, editeurs, Theory and Applications of Models of Computation, pages 1–19, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Dwork 2009] Cynthia Dwork et Jing Lei. *Differential privacy and robust statistics.* pages 371–380, 2009.

[Dwork 2016] Cynthia Dwork et Guy N. Rothblum. *Concentrated Differential Privacy*, 2016.

[Facebook Research] Facebook Research. *CrypTen - CrypTen.* `https://github.com/facebookresearch/CrypTen`.

[Fredrikson 2015] Matt Fredrikson, Somesh Jha et Thomas Ristenpart. *Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures.* In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pages 1322–1333, New York, NY, USA, 2015. ACM.

[FRESCO] FRESCO. *FRESCO - A FRamework for Efficient Secure COmputation.* https://github.com/aicis/fresco.

[Goldreich 1987] O. Goldreich, S. Micali et A. Wigderson. *How to Play ANY Mental Game.* In Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[Ishai 2008] Yuval Ishai, Manoj Prabhakaran et Amit Sahai. *Founding Cryptography on Oblivious Transfer – Efficiently.* In David Wagner, editeur, Advances in Cryptology – CRYPTO 2008, pages 572–591, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Jarecki 2007] Stanisław Jarecki et Vitaly Shmatikov. *Efficient two-party secure computation on committed inputs.* In Advances in Cryptology - EUROCRYPT 2007 - 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 97–114, 12 2007.

[Juvekar 2018] Chiraag Juvekar, Vinod Vaikuntanathan et Anantha Chandrakasan. *GAZELLE: A Low Latency Framework for Secure Neural Network Inference.* Cryptology ePrint Archive, Report 2018/073, 2018. https://eprint.iacr.org/2018/073.

[Kermany 2018] Daniel S. Kermany, Michael Goldbaum, Wenjia Cai, Carolina C.S. Valentim, Huiying Liang, Sally L. Baxter, Alex McKeown, Ge Yang, Xiaokang Wu, Fangbing Yan, Justin Dong, Made K. Prasadha, Jacqueline Pei, Magdalene Y.L. Ting, Jie Zhu, Christina Li, Sierra Hewett, Jason Dong, Ian Ziyar, Alexander Shi, Runze Zhang, Lianghong Zheng, Rui Hou, William Shi, Xin Fu, Yaou Duan, Viet A.N. Huu, Cindy Wen, Edward D. Zhang, Charlotte L. Zhang, Oulan Li, Xiaobo Wang, Michael A. Singer, Xiaodong Sun, Jie Xu, Ali Tafreshi, M. Anthony Lewis, Huimin Xia et Kang Zhang. *Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning.* Cell, vol. 172, no. 5, pages 1122 – 1131.e9, 2018.

[Lindell 2007] Yehuda Lindell et Benny Pinkas. *An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries.* In Moni Naor, editeur, Advances in Cryptology - EUROCRYPT 2007, pages 52–78, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[McMahan 2018] H. Brendan McMahan, Galen Andrew, Ulfar Erlingsson, Steve Chien, Ilya Mironov, Nicolas Papernot et Peter Kairouz. *A General Approach to Adding Differential Privacy to Iterative Training Procedures*, 2018.

[Mohassel 2017] Payman Mohassel et Yupeng Zhang. *SecureML: A System for Scalable Privacy-Preserving Machine Learning.* Cryptology ePrint Archive, Report 2017/396, 2017. https://eprint.iacr.org/2017/396.

[Mohassel 2018] Payman Mohassel et Peter Rindal. *ABY3: A Mixed Protocol Framework for Machine Learning.* Cryptology ePrint Archive, Report 2018/403, 2018. https://eprint.iacr.org/2018/403.

[Nielsen 2009] Jesper Buus Nielsen et Claudio Orlandi. *LEGO for Two-Party Secure Computation.* In Omer Reingold, editeur, Theory of Cryptography, pages 368–386, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Paul Mooney] Paul Mooney. *Chest X-Ray Images.* `https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia`.

[SCALE-MAMBA] SCALE-MAMBA. *Secure Computation Agorithms from LEuven - Multiparty AlgorithMs Basic Argot.* `https://github.com/KULeuven-COSIC/SCALE-MAMBA`.

[Shamir 1979] Adi Shamir. *How to Share a Secret.* Commun. ACM, vol. 22, no. 11, pages 612–613, Novembre 1979.

[Shokri 2016] Reza Shokri, Marco Stronati, Congzheng Song et Vitaly Shmatikov. *Membership Inference Attacks against Machine Learning Models*, 2016.

[Shokri 2017] R. Shokri, M. Stronati, C. Song et V. Shmatikov. *Membership Inference Attacks Against Machine Learning Models.* In 2017 IEEE Symposium on Security and Privacy (SP), pages 3–18, May 2017.

[Truex 2018] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang et Yi Zhou. *A Hybrid Approach to Privacy-Preserving Federated Learning*, 2018.

[Vom Brocke 2009] Simons A. Niehaves B. Riemer K. Plattfaut R. Cleven A. et al. Vom Brocke J. *Reconstructing the giant: On the importance of rigour in documenting the literature search process.* ECIS, Vol. 9, pp. 2206–2217, 2009.

[Wagh 2019] Sameer Wagh, Divya Gupta et Nishanth Chandran. *SecureNN: 3-Party Secure Computation for Neural Network Training.* Proceedings on Privacy Enhancing Technologies, 2019.

[Yao 1986] Andrew Chi-Chih Yao. *How to Generate and Exchange Secrets.* In Proceedings of the 27th Annual Symposium on Foundations of Computer Science, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.

[Zahur 2015] Samee Zahur et David Evans. *Obliv-C: A Language for Extensible Data-Oblivious Computation.* Cryptology ePrint Archive, Report 2015/1153, 2015. `https://eprint.iacr.org/2015/1153`.

Appendices

Appendix A

Source Code

A.1 Dataset preprocessing

```python
# -*- coding: utf-8 -*-

"""
Dataset Preprocessing
"""

from PIL import Image
import numpy as np
import pickle
import os

DATASET_PATH = './chest_xray/'
TRAIN_PATH_NORMAL = os.path.join(DATASET_PATH, 'train/', 'NORMAL/')
TRAIN_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'train/', 'PNEUMONIA/')
VALIDATION_PATH_NORMAL = os.path.join(DATASET_PATH, 'val/', 'NORMAL/')
VALIDATION_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'val/', 'PNEUMONIA/')
TEST_PATH_NORMAL = os.path.join(DATASET_PATH, 'test/', 'NORMAL/')
TEST_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'test/', 'PNEUMONIA/')

for file in os.listdir(TRAIN_PATH_NORMAL):
  print(file)
  if file == ".DS_Store":
    continue
  f = os.path.join(TRAIN_PATH_NORMAL, file)
  im = Image.open(f)
  a = np.asarray(im, dtype="int32")
  print(a)
  print(a.shape)
  newsize = (125, 150)
  im1 = im.resize(newsize)
  im1 = im1.convert('L')
  a = np.asarray(im1, dtype="int32")
  print(a)
  print(a.shape)
  break

print("**** TRAINING DATA - NORMAL ****")
normal_train = []

for file in os.listdir(TRAIN_PATH_NORMAL):
  print(file)
  if file == ".DS_Store":
```

```
        continue
    f = os.path.join(TRAIN_PATH_NORMAL, file)
    im = Image.open(f)
    newsize = (125, 150)
    im1 = im.resize(newsize)
    im1 = im1.convert('L')
    data = np.asarray(im1, dtype="int32")
    normal_train.append(data)

normal_train = np.asarray(normal_train)

with open("./normal_train.pkl", "wb") as f:
        pickle.dump(normal_train, f, protocol=4)

print("**** TRAINING DATA - PNEUMONIA ****")
pneumonia_train = []

for file in os.listdir(TRAIN_PATH_PNEUMONIA):
        print(file)
        if file == ".DS_Store":
                continue
        f = os.path.join(TRAIN_PATH_PNEUMONIA, file)
        im = Image.open(f)
        newsize = (125, 150)
        im1 = im.resize(newsize)
        im1 = im1.convert('L')
        data = np.asarray(im1, dtype="int32")
        pneumonia_train.append(data)

pneumonia_train = np.asarray(pneumonia_train)

with open("./pneumonia_train.pkl", "wb") as f:
        pickle.dump(pneumonia_train, f, protocol=4)

print("**** VALIDATION DATA - NORMAL ****")
normal_val = []

for file in os.listdir(VALIDATION_PATH_NORMAL):
        print(file)
        if file == ".DS_Store":
                continue
        f = os.path.join(VALIDATION_PATH_NORMAL, file)
        im = Image.open(f)
        newsize = (125, 150)
        im1 = im.resize(newsize)
        im1 = im1.convert('L')
        data = np.asarray(im1, dtype="int32")
        normal_val.append(data)

!ls '/content/drive/My Drive/BTP'

normal_val = np.asarray(normal_val)
with open("./normal_val.pkl", "wb") as f:
        pickle.dump(normal_val, f, protocol=4)

print("**** VALIDATION DATA - PNEUMONIA ****")
pneumonia_val = []

for file in os.listdir(VALIDATION_PATH_PNEUMONIA):
        print(file)
        if file == ".DS_Store":
                continue
        f = os.path.join(VALIDATION_PATH_PNEUMONIA, file)
        im = Image.open(f)
        newsize = (125, 150)
        im1 = im.resize(newsize)
```

```
        im1 = im1.convert('L')
        data = np.asarray(im1, dtype="int32")
        pneumonia_val.append(data)

pneumonia_val = np.asarray(pneumonia_val)
with open("./pneumonia_val.pkl", "wb") as f:
        pickle.dump(pneumonia_val, f, protocol=4)

print("**** TEST DATA - NORMAL ****")
normal_test = []

for file in os.listdir(TEST_PATH_NORMAL):
        print(file)
        if file == ".DS_Store":
                continue
        f = os.path.join(TEST_PATH_NORMAL, file)
        im = Image.open(f)
        newsize = (125, 150)
        im1 = im.resize(newsize)
        im1 = im1.convert('L')
        data = np.asarray(im1, dtype="int32")
        normal_test.append(data)

normal_test = np.asarray(normal_test)
with open("./normal_test.pkl", "wb") as f:
        pickle.dump(normal_test, f, protocol=4)

print("**** TEST DATA - PNEUMONIA ****")
pneumonia_test = []

for file in os.listdir(TEST_PATH_PNEUMONIA):
        print(file)
        if file == ".DS_Store":
                continue
        f = os.path.join(TEST_PATH_PNEUMONIA, file)
        im = Image.open(f)
        newsize = (125, 150)
        im1 = im.resize(newsize)
        im1 = im1.convert('L')
        data = np.asarray(im1, dtype="int32")
        pneumonia_test.append(data)

pneumonia_test = np.asarray(pneumonia_test)
with open("./pneumonia_test.pkl", "wb") as f:
        pickle.dump(pneumonia_test, f, protocol=4)
```

A.2 Model Training - After preprocessing 1

```
# -*- coding: utf-8 -*-
"""
Training models on images after preprocessing 1
"""

from __future__ import print_function
import tensorflow as tf

import pickle

import numpy as np


import os
DATASET_PATH = './'
TRAIN_PATH_NORMAL = os.path.join(DATASET_PATH, 'normal_train.pkl')
```

```
TRAIN_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'pneumonia_train.pkl')
VALIDATION_PATH_NORMAL = os.path.join(DATASET_PATH, 'normal_val.pkl')
VALIDATION_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'pneumonia_val.pkl')
TEST_PATH_NORMAL = os.path.join(DATASET_PATH, 'normal_test.pkl')
TEST_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'pneumonia_test.pkl')

file = open(TRAIN_PATH_NORMAL, 'rb')
# dump information to that file
train_normal = pickle.load(file)
# close the file
file.close()

file = open(TRAIN_PATH_PNEUMONIA, 'rb')
# dump information to that file
train_pneumonia = pickle.load(file)
# close the file
file.close()

file = open(VALIDATION_PATH_NORMAL, 'rb')
# dump information to that file
val_normal = pickle.load(file)
# close the file
file.close()

file = open(VALIDATION_PATH_PNEUMONIA, 'rb')
# dump information to that file
val_pneumonia = pickle.load(file)
# close the file
file.close()

file = open(TEST_PATH_NORMAL, 'rb')
# dump information to that file
test_normal = pickle.load(file)
# close the file
file.close()

file = open(TEST_PATH_PNEUMONIA, 'rb')
# dump information to that file
test_pneumonia = pickle.load(file)
# close the file
file.close()

y_normal = np.zeros((1341,), dtype=int)
y_pneumonia = np.ones((3882,), dtype=int)

x_train = np.concatenate((train_normal, train_pneumonia))
y_train = np.concatenate((y_normal, y_pneumonia))

y_normal_t = np.zeros((234,), dtype=int)
y_pneumonia_t = np.ones((390,), dtype=int)

x_test = np.concatenate((test_normal, test_pneumonia))
y_test = np.concatenate((y_normal_t, y_pneumonia_t))


from sklearn.utils import shuffle
x_train, y_train = shuffle(x_train, y_train, random_state=0)
x_test, y_test = shuffle(x_test, y_test, random_state=0)


# input image dimensions
img_rows, img_cols = 150, 125

x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
```

```

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

num_classes = 2
# convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

model = tf.keras.Sequential([
          tf.keras.layers.Conv2D(16, 8,
                                 strides=2,
                                 padding='same',
                                 activation='relu',
                                 input_shape=(150, 125, 1)),
          tf.keras.layers.MaxPooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.MaxPooling2D(2, 1),
          tf.keras.layers.Conv2D(64, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.MaxPooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.MaxPooling2D(2, 1),
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(32, activation='relu'),
          tf.keras.layers.Dense(2, activation='softmax')
  ])

model = tf.keras.Sequential([
          tf.keras.layers.Conv2D(16, 8,
                                 strides=2,
                                 padding='same',
                                 activation='relu',
                                 input_shape=(150, 125, 1)),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(64, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(32, activation='relu'),
          tf.keras.layers.Dense(2, activation='softmax')
```

```
  ])

model = tf.keras.Sequential([
          tf.keras.layers.Conv2D(16, 8,
                                  strides=2,
                                  padding='same',
                                  activation='relu',
                                  input_shape=(150, 125, 1)),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                  strides=2,
                                  padding='valid',
                                  activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(64, 4,
                                  strides=2,
                                  padding='valid',
                                  activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(256, 4,
                                  strides=2,
                                  padding='valid',
                                  activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(32, activation='relu'),
          tf.keras.layers.Dense(2, activation='softmax')
  ])

model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(64, (3, 3), input_shape=(150, 125, 1), padding='same', activation='relu'),
  tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
  tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(32, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])

batch_size = 32
epochs = 40

from keras.callbacks import ModelCheckpoint, EarlyStopping
mcp = ModelCheckpoint(filepath='./model_simple.h5',monitor="val_acc", save_best_only=True, save_weights_only=False)
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=4)

model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              metrics=['accuracy'])
```

```

hist = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          callbacks=[mcp],
          validation_split=0.4)
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

# model.save('./model_inc1.h5')

import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['acc'], color='red')
ax.plot(hist.history['val_acc'], color ='green')
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['loss'], color='red')
ax.plot(hist.history['val_loss'], color ='green')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

tf.keras.utils.plot_model(model, to_file='./ave_model.png', show_shapes=True)
```

A.3 Model Training - After preprocessing 2

```
# -*- coding: utf-8 -*-
"""
Training models on images after preprocessing 1
"""

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import os
print(os.listdir("./chest_xray"))
from glob import glob
from PIL import Image
# %matplotlib inline
import matplotlib.pyplot as plt
import cv2
import fnmatch
import keras
from time import sleep
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense,Conv2D,MaxPool2D,Dropout,Flatten,BatchNormalization,
    MaxPooling2D,Activation
```

```
from keras.optimizers import RMSprop,Adam
from tensorflow.keras.callbacks import EarlyStopping
from keras import backend as k

imagePatches = glob('./chest_xray/**/**/*.jpeg', recursive=True)
print(len(imagePatches))

"""## No need to run"""

pattern_normal = '*NORMAL*'
pattern_bacteria = '*_bacteria_*'
pattern_virus = '*_virus_*'

normal = fnmatch.filter(imagePatches, pattern_normal)
bacteria = fnmatch.filter(imagePatches, pattern_bacteria)
virus = fnmatch.filter(imagePatches, pattern_virus)
x = []
y = []
for img in imagePatches:
    full_size_image = cv2.imread(img)
    print(full_size_image.shape)
    im = cv2.resize(full_size_image, (224, 224), interpolation=cv2.INTER_CUBIC)
    print(im)
    x.append(im)
    break
    if img in normal:
        y.append(0)
    elif img in bacteria:
        y.append(1)
    elif img in virus:
        y.append(1)
    else:
        #break
        print('no class')

pattern_normal = '*NORMAL*'
pattern_bacteria = '*_bacteria_*'
pattern_virus = '*_virus_*'

normal = fnmatch.filter(imagePatches, pattern_normal)
bacteria = fnmatch.filter(imagePatches, pattern_bacteria)
virus = fnmatch.filter(imagePatches, pattern_virus)
x = []
y = []
for img in imagePatches:
    full_size_image = cv2.imread(img)
    im = cv2.resize(full_size_image, (224, 224), interpolation=cv2.INTER_CUBIC)
    x.append(im)
    if img in normal:
        y.append(0)
    elif img in bacteria:
        y.append(1)
    elif img in virus:
        y.append(1)
    else:
        #break
        print('no class')
x = np.array(x)
y = np.array(y)

from sklearn.model_selection import train_test_split
x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size = 0.2, random_state
    = 101)
y_train = to_categorical(y_train, num_classes = 2)
y_valid = to_categorical(y_valid, num_classes = 2)
del x, y
```

```python

import pickle
with open("./kernel_train_x.pkl", "wb") as f:
        pickle.dump(x_train, f, protocol=4)

with open("./kernel_train_y.pkl", "wb") as f:
        pickle.dump(x_valid, f, protocol=4)

with open("./kernel_test_x.pkl", "wb") as f:
        pickle.dump(y_train, f, protocol=4)

with open("./kernel_test_y.pkl", "wb") as f:
        pickle.dump(y_valid, f, protocol=4)

"""## Run from here"""

import pickle
with open("./kernel_train_x.pkl", "rb") as f:
        x_train = pickle.load(f)

with open("./kernel_train_y.pkl", "rb") as f:
        x_valid = pickle.load(f)

with open("./kernel_test_x.pkl", "rb") as f:
        y_train = pickle.load(f)

with open("./kernel_test_y.pkl", "rb") as f:
        y_valid = pickle.load(f)


import tensorflow as tf
model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(64, (3, 3), input_shape=(224, 224, 3), padding='same', activation='relu'),
  tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
  tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(32, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])

from keras.callbacks import ModelCheckpoint
mcp = ModelCheckpoint(filepath='./model_vgg_3.hdf5',monitor="val_acc", save_best_only=True, save_weights_only=False)
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=4)

model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              metrics=['accuracy'])

```

```
hist = model.fit(x_train, y_train,
          batch_size=32,
          epochs=40,
          verbose=1,
          callbacks=[mcp, es],
          validation_data=(x_valid, y_valid))
score = model.evaluate(x_valid, y_valid, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['acc'], color='red')
ax.plot(hist.history['val_acc'], color ='green')
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['loss'], color='red')
ax.plot(hist.history['val_loss'], color ='green')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

tf.keras.utils.plot_model(model, to_file='/content/drive/My Drive/BTP/final/vgg_3_model.png', show_shapes=True)
```

A.4 Model Training - With Differential Privacy

```
# -*- coding: utf-8 -*-
"""
Training using Differential Privacy
"""

from __future__ import print_function
import tensorflow as tf

import pickle
import numpy as np

import os
DATASET_PATH = './'
TRAIN_PATH_NORMAL = os.path.join(DATASET_PATH, 'normal_train.pkl')
TRAIN_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'pneumonia_train.pkl')
VALIDATION_PATH_NORMAL = os.path.join(DATASET_PATH, 'normal_val.pkl')
VALIDATION_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'pneumonia_val.pkl')
TEST_PATH_NORMAL = os.path.join(DATASET_PATH, 'normal_test.pkl')
TEST_PATH_PNEUMONIA = os.path.join(DATASET_PATH, 'pneumonia_test.pkl')

file = open(TRAIN_PATH_NORMAL, 'rb')
# dump information to that file
train_normal = pickle.load(file)
# close the file
```

```
file.close()

file = open(TRAIN_PATH_PNEUMONIA, 'rb')
# dump information to that file
train_pneumonia = pickle.load(file)
# close the file
file.close()

file = open(VALIDATION_PATH_NORMAL, 'rb')
# dump information to that file
val_normal = pickle.load(file)
# close the file
file.close()

file = open(VALIDATION_PATH_PNEUMONIA, 'rb')
# dump information to that file
val_pneumonia = pickle.load(file)
# close the file
file.close()

file = open(TEST_PATH_NORMAL, 'rb')
# dump information to that file
test_normal = pickle.load(file)
# close the file
file.close()

file = open(TEST_PATH_PNEUMONIA, 'rb')
# dump information to that file
test_pneumonia = pickle.load(file)
# close the file
file.close()

y_normal = np.zeros((1341,), dtype=int)
y_pneumonia = np.ones((3882,), dtype=int)

x_train = np.concatenate((train_normal, train_pneumonia))
y_train = np.concatenate((y_normal, y_pneumonia))

y_normal_t = np.zeros((234,), dtype=int)
y_pneumonia_t = np.ones((390,), dtype=int)

x_test = np.concatenate((test_normal, test_pneumonia))
y_test = np.concatenate((y_normal_t, y_pneumonia_t))

from sklearn.utils import shuffle
x_train, y_train = shuffle(x_train, y_train, random_state=0)
x_test, y_test = shuffle(x_test, y_test, random_state=0)

# input image dimensions
img_rows, img_cols = 150, 125

x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

num_classes = 2
# convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
```

```
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
print('y_train shape:', y_train.shape)

print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)

dpsgd = True             # If True, train with DP-SGD
learning_rate = 0.015     # Learning rate for training
noise_multiplier = 1.1   # Ratio of the standard deviation to the clipping norm
l2_norm_clip = 1.0       # Clipping norm
batch_size = 50         # Batch size
epochs = 20              # Number of epochs
microbatches = 5        # Number of microbatches


def compute_epsilon(steps):
    """Computes epsilon value for given hyperparameters."""
    if noise_multiplier == 0.0:
        return float('inf')
    orders = [1 + x / 10. for x in range(1, 100)] + list(range(12, 64))
    sampling_probability = batch_size / 60000
    rdp = compute_rdp(q=sampling_probability,
                      noise_multiplier=noise_multiplier,
                      steps=steps,
                      orders=orders)
    # Delta is set to 1e-5 because MNIST has 60000 training points.
    return get_privacy_spent(orders, rdp, target_delta=1e-5)[0]

tf.logging.set_verbosity(tf.logging.INFO)
if dpsgd and batch_size % microbatches != 0:
    raise ValueError('Number of microbatches should divide evenly batch_size')

# model = tf.keras.Sequential([
#       tf.keras.layers.Conv2D(16, 8,
#                              strides=2,
#                              padding='same',
#                              activation='relu',
#                              input_shape=(150, 125, 1)),
#       tf.keras.layers.AveragePooling2D(2, 1),
#       tf.keras.layers.Conv2D(32, 4,
#                              strides=2,
#                              padding='valid',
#                              activation='relu'),
#       tf.keras.layers.AveragePooling2D(2, 1),
#       tf.keras.layers.Flatten(),
#       tf.keras.layers.Dense(32, activation='relu'),
#       tf.keras.layers.Dense(2)
#   ])

# model = tf.keras.Sequential([
#           tf.keras.layers.Conv2D(16, 8,
#                                  strides=2,
#                                  padding='same',
#                                  activation='relu',
#                                  input_shape=(150, 125, 1)),
#           tf.keras.layers.MaxPooling2D(2, 1),
#           tf.keras.layers.Conv2D(32, 4,
#                                  strides=2,
#                                  padding='valid',
#                                  activation='relu'),
#           tf.keras.layers.MaxPooling2D(2, 1),
#           tf.keras.layers.Conv2D(64, 4,
#                                  strides=2,
#                                  padding='valid',
#                                  activation='relu'),
#           tf.keras.layers.MaxPooling2D(2, 1),
```

```
#           tf.keras.layers.Conv2D(32, 4,
#                                 strides=2,
#                                 padding='valid',
#                                 activation='relu'),
#           tf.keras.layers.MaxPooling2D(2, 1),
#           tf.keras.layers.Flatten(),
#           tf.keras.layers.Dense(32, activation='relu'),
#           tf.keras.layers.Dense(2)
#   ])

# model = tf.keras.Sequential([
#           tf.keras.layers.Conv2D(16, 8,
#                                 strides=2,
#                                 padding='same',
#                                 activation='relu',
#                                 input_shape=(150, 125, 1)),
#           tf.keras.layers.AveragePooling2D(2, 1),
#           tf.keras.layers.Conv2D(32, 4,
#                                 strides=2,
#                                 padding='valid',
#                                 activation='relu'),
#           tf.keras.layers.AveragePooling2D(2, 1),
#           tf.keras.layers.Conv2D(64, 4,
#                                 strides=2,
#                                 padding='valid',
#                                 activation='relu'),
#           tf.keras.layers.AveragePooling2D(2, 1),
#           tf.keras.layers.Conv2D(32, 4,
#                                 strides=2,
#                                 padding='valid',
#                                 activation='relu'),
#           tf.keras.layers.AveragePooling2D(2, 1),
#           tf.keras.layers.Flatten(),
#           tf.keras.layers.Dense(32, activation='relu'),
#           tf.keras.layers.Dense(2, activation='softmax')
#   ])

model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(64, (3, 3), input_shape=(150, 125, 1), padding='same', activation
    ='relu'),
  tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
  tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same',),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(32, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])

from tensorflow_privacy.privacy.analysis.rdp_accountant import compute_rdp
from tensorflow_privacy.privacy.analysis.rdp_accountant import get_privacy_spent
from tensorflow_privacy.privacy.optimizers.dp_optimizer import
```

```
    DPGradientDescentGaussianOptimizer

if dpsgd:
    optimizer = DPGradientDescentGaussianOptimizer(
        l2_norm_clip=l2_norm_clip,
        noise_multiplier=noise_multiplier,
        num_microbatches=microbatches,
        learning_rate=learning_rate)
    # Compute vector of per-example loss rather than its mean over a minibatch.
    loss = tf.keras.losses.CategoricalCrossentropy(
        from_logits=True, reduction=tf.losses.Reduction.NONE)
else:
    optimizer = tf.optimizers.SGD(learning_rate=learning_rate)
    loss = tf.keras.losses.CategoricalCrossentropy(from_logits=True)

# Compile model with Keras
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

rows = x_train.shape[0]
rows_req = (rows//50)*50
x_train = x_train[:rows_req,:,:,:]
y_train = y_train[:rows_req,:]

x_train.shape

rows = x_test.shape[0]
rows_req = (rows//50)*50
x_test = x_test[:rows_req,:,:,:]
y_test = y_test[:rows_req,:]

x_test.shape

y_test[:30]

from keras.callbacks import ModelCheckpoint, EarlyStopping
mcp = ModelCheckpoint(filepath='./ave_model_dp.h5',monitor="val_acc", save_best_only=True,
    save_weights_only=False)
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=4)

hist = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

# Compute the privacy budget expended.
if dpsgd:
    eps = compute_epsilon(epochs * 60000 // batch_size)
    print('For delta=1e-5, the current epsilon is: %.2f' % eps)
else:
    print('Trained with vanilla non-private SGD optimizer')

# model.save('/content/drive/My Drive/BTP/models/simple_dp.h5')

x_test = x_test[:576,:,:,:]
y_test = y_test[:576,:]

# model.load_weights('/content/drive/My Drive/BTP/final/ave_model_dp.h5')
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
# x_test[0].shape
# model.predict(x_test[0], y_test[0])

import matplotlib.pyplot as plt
fig = plt.figure()
```

```
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['acc'], color='red')
ax.plot(hist.history['val_acc'], color ='green')
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_facecolor('w')
ax.grid(b=False)
ax.plot(hist.history['loss'][10], color='red')
ax.plot(hist.history['val_loss'][10], color ='green')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```

A.5 Model Serving

```
# coding: utf-8
"""
Private Predictions with TFE Keras
"""

from collections import OrderedDict

import numpy as np
import tensorflow as tf

import tf_encrypted as tfe
import tf_encrypted.keras.backend as KE


num_classes = 2
input_shape = (1, 150, 125, 1)

model = tf.keras.Sequential([
          tf.keras.layers.Conv2D(16, 8,
                                 strides=2,
                                 padding='same',
                                 activation='relu',
                                 batch_input_shape=input_shape),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(64, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
          tf.keras.layers.Conv2D(32, 4,
                                 strides=2,
                                 padding='valid',
                                 activation='relu'),
          tf.keras.layers.AveragePooling2D(2, 1),
```

```
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(32, activation='relu'),
            tf.keras.layers.Dense(2, name='logit')
    ])

model = tf.keras.Sequential([
            tf.keras.layers.Conv2D(16, 8,
                                   strides=2,
                                   padding='same',
                                   activation='relu',
                                   batch_input_shape=input_shape),
            tf.keras.layers.AveragePooling2D(2, 1),
            tf.keras.layers.Conv2D(32, 4,
                                   strides=2,
                                   padding='valid',
                                   activation='relu'),
            tf.keras.layers.AveragePooling2D(2, 1),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(32, activation='relu'),
            tf.keras.layers.Dense(num_classes, name='logit')
    ])


# With `load_weights` we can easily load the weights you have saved previously after training your model.

# ### Only for Differential Privacy model

dpsgd = True              # If True, train with DP-SGD
learning_rate = 0.015     # Learning rate for training
noise_multiplier = 1.1   # Ratio of the standard deviation to the clipping norm
l2_norm_clip = 1.0       # Clipping norm
batch_size = 50          # Batch size
epochs = 20              # Number of epochs
microbatches = 5         # Number of microbatches


def compute_epsilon(steps):
    """Computes epsilon value for given hyperparameters."""
    if noise_multiplier == 0.0:
        return float('inf')
    orders = [1 + x / 10. for x in range(1, 100)] + list(range(12, 64))
    sampling_probability = batch_size / 60000
    rdp = compute_rdp(q=sampling_probability,
                      noise_multiplier=noise_multiplier,
                      steps=steps,
                      orders=orders)
    # Delta is set to 1e-5 because MNIST has 60000 training points.
    return get_privacy_spent(orders, rdp, target_delta=1e-5)[0]

tf.logging.set_verbosity(tf.logging.INFO)
if dpsgd and batch_size % microbatches != 0:
    raise ValueError('Number of microbatches should divide evenly batch_size')

from privacy.analysis.rdp_accountant import compute_rdp
from privacy.analysis.rdp_accountant import get_privacy_spent
from privacy.optimizers.dp_optimizer import DPGradientDescentGaussianOptimizer

if dpsgd:
    optimizer = DPGradientDescentGaussianOptimizer(
        l2_norm_clip=l2_norm_clip,
        noise_multiplier=noise_multiplier,
        num_microbatches=microbatches,
        learning_rate=learning_rate)
    # Compute vector of per-example loss rather than its mean over a minibatch.
    loss = tf.keras.losses.CategoricalCrossentropy(
```

```
        from_logits=True, reduction=tf.losses.Reduction.NONE)
else:
    optimizer = tf.optimizers.SGD(learning_rate=learning_rate)
    loss = tf.keras.losses.CategoricalCrossentropy(from_logits=True)

# Compile model with Keras
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])


pre_trained_weights = 'model_simple_dp.h5'
model.load_weights(pre_trained_weights)


# ## Protocol
#
# We first configure the protocol we will be using, as well as the servers on which we want to run it. We will be using the SecureNN protocol to secret share the model between each of the three TFE servers. Most importantly, this will add the capability of providing predictions on encrypted data.
#
# Note that the configuration is saved to file as we will be needing it in the client as well.

players = OrderedDict([
    ('server0', 'localhost:4000'),
    ('server1', 'localhost:4001'),
    ('server2', 'localhost:4002'),
])

config = tfe.RemoteConfig(players)
config.save('/tmp/tfe.config')


tfe.set_config(config)
tfe.set_protocol(tfe.protocol.SecureNN())


# ## Launching servers
#
# Before actually serving the computation below we need to launch TFE servers in new processes. Run the following in three different terminals. You may have to allow Python to accept incoming connections.

for player_name in players.keys():
    print("python3.6 -m tf_encrypted.player --config /tmp/tfe.config {}".format(player_name))


# ## Convert TF Keras into TFE Keras
#
# `tfe.keras.models.clone_model` can convert automatically the TF Keras model into a TFE Keras model.

tf.reset_default_graph()
with tfe.protocol.SecureNN():
    tfe_model = tfe.keras.models.clone_model(model)


# ## Set up a new `tfe.serving.QueueServer`
#
# `tfe.serving.QueueServer` will launch a serving queue, so that the TFE servers can accept prediction requests on the secured model from external clients.

# Set up a new tfe.serving.QueueServer for the shared TFE model
q_input_shape = (1, 150, 125, 1)
q_output_shape = (1, 2)
```

```
server = tfe.serving.QueueServer(
    input_shape=q_input_shape, output_shape=q_output_shape, computation_fn=tfe_model
)


# ## Start Server
#
sess = KE.get_session()


request_ix = 1

def step_fn():
    global request_ix
    print("Served encrypted prediction {i} to client.".format(i=request_ix))
    request_ix += 1

server.run(
    sess,
    step_fn=step_fn)
```

A.6 Client Prediction

```
# coding: utf-8
# Private Prediction using TFE Keras - Serving (Client)
#

import numpy as np
import tensorflow as tf
import tf_encrypted as tfe

from tensorflow.keras.datasets import mnist


# ## Data

import pickle
file = open('../normal_test.pkl', 'rb')
# dump information to that file
test_normal = pickle.load(file)
# close the file
file.close()

file = open('../pneumonia_test.pkl', 'rb')
# dump information to that file
test_pneumonia = pickle.load(file)
# close the file
file.close()


y_normal_t = np.zeros((234,), dtype=int)
y_pneumonia_t = np.ones((390,), dtype=int)

x_test = np.concatenate((test_normal, test_pneumonia))
y_test = np.concatenate((y_normal_t, y_pneumonia_t))
from sklearn.utils import shuffle
x_test, y_test = shuffle(x_test, y_test, random_state=0)
x_test.shape


rows = x_test.shape[0]
rows_req = (rows//50)*50
x_test = x_test[:rows_req,:,:]
```

```python
y_test = y_test[:rows_req]


# input image dimensions
img_rows, img_cols = 150, 125

x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_test = x_test.astype('float32')
x_test /= 255


## Set up `tfe.serving.QueueClient`

config = tfe.RemoteConfig.load("/tmp/tfe.config")

tfe.set_config(config)
tfe.set_protocol(tfe.protocol.SecureNN())


input_shape = (1, 150, 125, 1)
output_shape = (1, 2)


client = tfe.serving.QueueClient(
    input_shape=input_shape,
    output_shape=output_shape)


sess = tfe.Session(config=config)


# ## Query Model

# User inputs
num_tests = 25
images, expected_labels = x_test[100:num_tests+100], y_test[100:num_tests+100]


for image, expected_label in zip(images, expected_labels):

    res = client.run(
        sess,
        image.reshape(1, 150, 125, 1))

    predicted_label = np.argmax(res)

    print("The image had label {} and was {} classified as {}".format(
        expected_label,
        "correctly" if expected_label == predicted_label else "incorrectly",
        predicted_label))
```