Secure Collaborative Training of Machine Learning Model using MPC

By Souhail Meftah, Ruomu Hou

October 22, 2019

1 Background and Motivation

The advance in machine learning that was driven by the wide application of neural networks has brought automation to a new level. Neural networks are deployed to replace human in making financial decisions, medical image processing and many tasks that were once believed to be 'of higher intelligent requirement' and hence not solvable by machines. However, the seemingly magical power of machine learning is not a magic but often relies on carefully crafted models and massive labelled training data to function. Under the hood of 'artificial intelligence', neural networks are essentially fitting massively parameterized generic models to specific functions. Hence, a huge amount of data which were previously studied and labelled is used to 'teach' neural network models to 'learn' the decision functions.

As the decisions made by neural network getting more complicated and increasingly mission-critical, it poses a higher requirement not only on the scale of the data (e.g. it is not uncommon to find PB sized database being used in training AI nowadays) but also on the specificity of the data. For instance, even prior to the emerge of neural network, banks have been using generic data, such as age and credit records to make decisions for loans, but nowadays banks are using more complicated data, such as social network posts, consumption behaviour and even facial expressions to make much more accurate decisions with the help of machine learning.

Now it is natural to ask one question: *where to get the data?* Indeed, few institutions, not to mention individuals, independently hold data of many people with specificity high enough to train the complicated models, so data from different companies or institutions need to be combined to make it possible. On the other hand, due to the high specificity, institutions are reluctant in sharing their data because of the concerns for business interest or legal liability, and this poses a need for a technology to enable parties to use their data to jointly train a neural network with protection from the potential risk of data leakage.

Our work creates a prototype of privacy-preserving federated learning applications by combining the state of the art multi-party computation framework with a real machine learning use case. In our work, we illustrated the techniques for building a secure federated learning app, analyzed the security guarantees of our implementation and tested the performance of our prototype using experiments.

1.1 Multi-party Computation

Multi-party computation is a technique created by cryptography and distributed computing research community to enable multiple mutually distrust parties to perform computation on jointly held data. Formally, N parties $p_1, p_2, ..., p_N$ each has an input $x_1, x_2, ..., x_N$ respectively. They want to collaborately compute a publicly known function f's output on the inputs, i.e. $\text{out} = f(x_1, x_2, ..., x_N)$, without exposing any information about their own input other than that exposed by out

In general, MPC is achieved with either of two techniques, or a combination of them: garbled circuit or secret sharing. The ability to do MPC does not come for free. Instead, it often poses a substantially higher communication and computational cost than their trusted counterpart where privacy is not a concern. Many works have been done to reduce the cost of MPC for evaluating specific function or with additional assumptions.

In our setting, we assume that the parties are honest-but-curious. In our model, all the parties behave honestly but they will attempt to discover other's input by examine the transcription of the execution passively, and we achieves privacy in the cryptographic sense even under $n-1$ corruption of the parties. In practice, this model provides data privacy guarantee against after-fact corruption, which covers majority of the security breaches. Moreover, this model reduces the burden of providing liveness and validating secret shares, and permits an efficient protocol. Notice that our prototype could be non-trivially extended to accomodate active attacks by incoporating SPDZ verified secret sharing techniques with another set of assumptions.

1.2 iDash Privacy and Security Workshop 2019: Secure genome analytics competition

iDash Privacy and Security Workshop is an annually workshop ...

Our work contributes to the topic 'Secure Collaborative Training of Machine Learning Model' of the competition. We choose to work with the topics provided by the competition because it provides us with real data to start with and a clear metric to judge the result. The data used in this example consist of two sets of genome sketch labelled with cancer indicator. 12k(18k) genes were analyzed with 400(200) samples respectively.

2 Methodology and Technologies

3 Implementation

3.1 Requirement and Setup

Notice: It requires at least a setup with 12GB memory to run the code.

```
# prepare dependency
sudo apt install -y git python3 python3-pip jupyter
# install the python dependencies
pip3 install jupyter syft torch torchvision pandas
# due to syft compatibility issue, we need to downgrade torch
pip3 install --upgrade torch==1.1.0
# clone the repository
git clone https://github.com/Souhail-MEFTAH/i-dash2019.git
# launch the project
cd i-dash2019/
jupyter notebook
```

Listing 1: Setup the runtime environment

Our experiment runs on 1 machine on the Tembusu Cluster with Intel E5-2620V3, 256GB DDR3 RAM, and CentOS 7.x.x. The code has been tested on Ubuntu 18.04 as well. You could use the code listed in Listing 1 to run in similar environments.

Alternatively, you may run the prepared docker image with
`sudo docker run -it --net=host houruomu/cs6203 jupyter notebook --allow-root`

3.2 Load the Data

```
import pandas as pd
import numpy as np

# read the data from the input files
def getSamples(filename):
    data = pd.read_csv(filename, sep='\t')
    return data.values[:, 1:].transpose()

data1 = getSamples("GSE2034-Normal-train.txt")
data2 = getSamples("GSE2034-Tumor-train.txt")

# code for formatting the data to numpy arrays

# partition the data into training data and test data
x_train = x[:n_train_items]
y_train = y[:n_train_items]

x_test = x[n_train_items:]
y_test = y[n_train_items:]
```

Listing 2: Load the data

Our code snippet in Listing 2 are the instructions that we used to load the data from the text document. In the text file, the first row and first column are data labels.

Each column of the file corresponds to a data sample with rows being the SNPs values (i.e. the features). The code prepares the data into 2d numpy arrays with each row corresponding to a sample.

3.3 Model Creation

```python
# The class defining our sub-network
class Res1d(nn.Module):
    def __init__(self, inSize, outSize, kernel=(3,), strides=1,):
        # code for defining the layers

    def forward(self, x):
        # code for defining how the layers are composed

# The class defining the overall network
class Net(nn.Module):
    def __init__(self):
        # code for defining the layers

    def forward(self, x):
        # code for defining how the layers are composed
```

Listing 3: Define the model

Our model is adapted from a well-estabilished convolutional model for analyzing temporal data. The neural network has two data flows, on one flow there is only a fully connected layer with 64 outputs, and on the other flow a convolutional sub-network is repeatedly applied to generate a rich feature space. Then the two data flows are concatenated and 2 fully connected layers are used to get the binary output.

The code snippet in Listing 3 illustrates our model definition. In `Pytorch`, the networks are defined as classes where in the `__init__` method the layers are defined and in `forward` method the operations to compose the layers are defined. The subnet class `Res1d` is defined with parameters `inSize, outSize, kernel, strides` to be used by the `Net` class to instatiate subnets in the layers definition. Notice that the `Net` class has a hidden parameter `dim` declared as a global variable, which corresponds to the feature space dimension of the input data.

3.4 Model Training

```python
net = Net()
criterion = nn.BCELoss()  # Binary Cross Entropy
# SGD optimizer with learning rate 0.001 and momentum 0.9
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for batch in range(1000):
    # get mini-batch
    indices = np.random.choice(len(x_train), size=(30))
    inputs = x_train[indices]
    labels = y_train[indices]

    # format input into [#sample, #channel, #feature]
    inputs = torch.from_numpy(inputs).view([-1, 1, dim]).float()
    labels = torch.from_numpy(labels).view([-1, 1]).float()

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs).view([-1, 1])
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

Listing 4: Train the model

We use mini-batched stochastic gradient descent with binary cross entropy loss function to train the parameters. Based on experimental results, we choose a batch size of 30, learning rate of 0.001 and momentum of 0.9. The code for training is in Listing 4.