Configurable Vertically Split Neural Network

Han Liang Wee Eric (A0065517A)
Lok You Tan (A0139248X)

AY2020/21 S1
Week 13

National University of Singapore

Split Neural Networks Learning in VFL

This project aims to implement a split learning approach of neural networks (NN) in VFL. Some descriptions are as follows:

1. Code mainly in **Python**, following Google Python style. Code **sanity checks are needed**.
2. Use Pytorch, Singa, or TensorFlow as local training framework. Define local training framework **abstract** such that anyone can change it easily through configuration.
3. Make each party's local NN architecture **configurable** such that the parties' local model architectures can be different.
4. Apply partially homomorphic encryption for **secure aggregation** on the cut layer in the split learning approach.

Motivation

Premise

- Privacy laws (i.e. GDPR, CCPA) restrict entities from sharing data

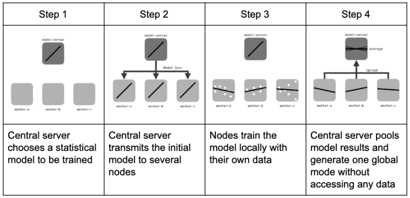
Problem

- No longer able to use data from other entities. Model performance suffers

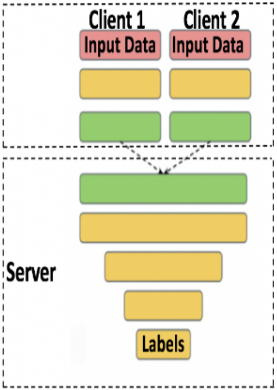
Need

- Distributed Secure Training - Training the AI models across ‘isolated islands’
- Data Privacy - Privacy must be preserved across nodes

Federated Learning and Split Learning



Step 1	Step 2	Step 3	Step 4
Central server chooses a statistical model to be trained	Central server transmits the initial model to several nodes	Nodes train the model locally with their own data	Central server pools model results and generate one global mode without accessing any data



[1]Diagram from https://en.wikipedia.org/wiki/Federated_learning
[2]Diagram from https://arxiv.org/pdf/1812.00564.pdf

Data Leakage and Defenses

- Recent works showed that sharing of gradients can result in data leakage
  - Deep leakage from gradients (Zhu et al.)

Cryptographic defense measures

- Homomorphic encryption (slow)
- Secure aggregation (no need trusted third-party)

Desirable to have system that is configurable to cater to different needs and network architectures

Our contributions

Key aspects:

- Implement a distributed VFL framework to train split NN
  - Emphasis on configurability and flexibility
  - Different parties can have different local model architectures
  - User alignment using RSA blind signature-based private set intersection
- Abstraction of underlying training framework
  - Use of Pytorch
  - Configuration file to unify different parties and their models
- NN submodel configurable
  - Pytorch load / save model state dictionary
  - Expandable, with interface to load model via code
- Secure aggregation on the cut layer
  - Choice of Threshold-Paillier or SMPC

Assumptions

- Semi-honest model (each party follows specified protocol exactly) that assumes there is a trusted party (Intel SGX)
- Labels Y held by one party (gradient computer)
- Rest are submodels
- No sharing of raw data or labels
- Not all parties share same sample IDs (require user alignment)

No assumptions w.r.t network architecture / privacy techniques (e.g. homomorphic encryption, secure multi-party computation, differential privacy)
System configurable and adaptable to different scenarios

Local party training frameworks

Want flexibility for each party to use their own preferred ML framework

- Considered Tensorflow and Pytorch (two most popular frameworks)

Differences in gradient computation and backpropagation

- Tensorflow uses tf.GradientTape
- Pytorch uses backward()

Implementation for Tensorflow would be a lot more complicated (to be able to compute gradients, need to explicitly track outputs after they are sent to other parties)

- We picked Pytorch

Wrapper

For the split / networking / encryption / related functionality, instead of implementing a custom layer, we use a wrapper because:

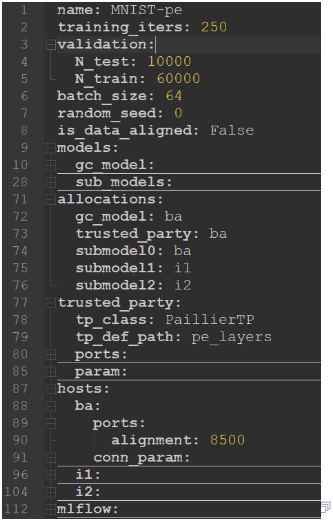
- Flexibility
- Ease of implementation
- Easily modified to accommodate additional ML frameworks / encryption / network protocols in the future

Configurability through YAML config

Configuration file to setup the entire network

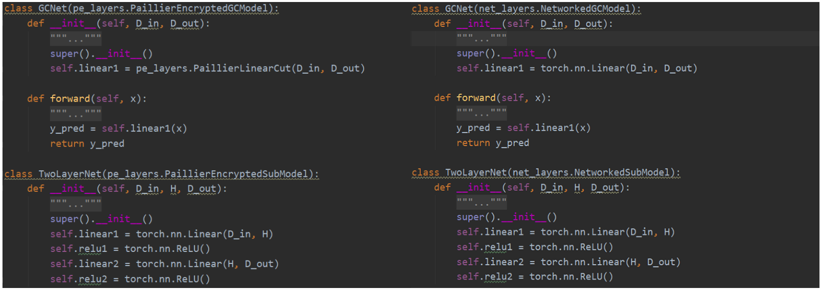
- Trusted third party setups parties and starts training process using agreed-upon config file
- Each party free to have different local architectures as long as the aggregation is done properly
- Open and transparent contract

YAML config



```yaml
name: MNIST-pe
training_iters: 250
validation:
  N_test: 10000
  N_train: 60000
batch_size: 64
random_seed: 0
is_data_aligned: False
models:
  gc_model:
  sub_models:
allocations:
  gc_model: ba
  trusted_party: ba
  submodel0: ba
  submodel1: i1
  submodel2: i2
trusted_party:
  tp_class: PaillierTP
  tp_def_path: pe_layers
  ports:
  param:
hosts:
  ba:
    ports:
      alignment: 8500
    conn_param:
  i1:
  i2:
mlflow:
```

Model definition



```
class GCNet(pe_layers.PaillierEncryptedGCModel):
    def __init__(self, D_in, D_out):
        """..."""
        super().__init__()
        self.linear1 = pe_layers.PaillierLinearCut(D_in, D_out)

    def forward(self, x):
        """..."""
        y_pred = self.linear1(x)
        return y_pred

class TwoLayerNet(pe_layers.PaillierEncryptedSubModel):
    def __init__(self, D_in, H, D_out):
        """..."""
        super().__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.relu1 = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(H, D_out)
        self.relu2 = torch.nn.ReLU()
```

```
class GCNet(net_layers.NetworkedGCModel):
    def __init__(self, D_in, D_out):
        """..."""
        super().__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)

    def forward(self, x):
        """..."""
        y_pred = self.linear1(x)
        return y_pred

class TwoLayerNet(net_layers.NetworkedSubModel):
    def __init__(self, D_in, H, D_out):
        """..."""
        super().__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.relu1 = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(H, D_out)
        self.relu2 = torch.nn.ReLU()
```

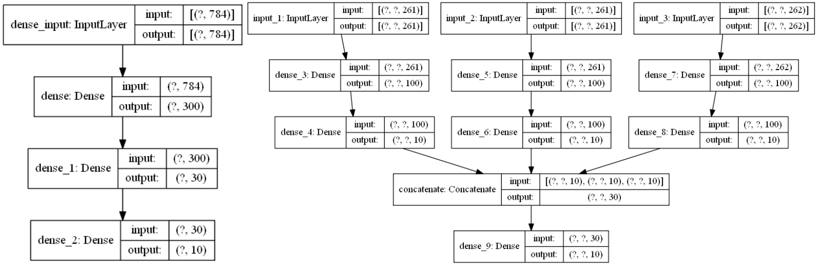
Architecture

- **GC**: Responsible for the labels and gradient propagation
- **TP**: Defined in conjuction with the scenario
- **Submodel**: Responsible for the sub neural network

Process:

1. Each party starts process
2. Perform user alignment if necessary
3. Setup individual networks on each party using YAML config
4. Start training for # of training iterations
   - 4.1 Each **submodel** computes forward pass and passes to **GC**
   - 4.2 **GC** assembles received values from **submodel**s, computes forward pass and loss
   - 4.3 **GC** backpropagates back to the **submodel**s
   - 4.4 **Submodels** update their weights

Our MNIST neural network

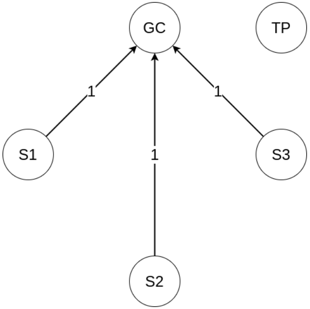

dense_input: InputLayer | input: [(?, 784)] | output: [(?, 784)]
dense: Dense | input: (?, 784) | output: (?, 300)
dense_1: Dense | input: (?, 300) | output: (?, 30)
dense_2: Dense | input: (?, 30) | output: (?, 10)
input_1: InputLayer | input: [(?, ?, 261)] | output: [(?, ?, 261)]
input_2: InputLayer | input: [(?, ?, 261)] | output: [(?, ?, 261)]
input_3: InputLayer | input: [(?, ?, 262)] | output: [(?, ?, 262)]
dense_3: Dense | input: (?, ?, 261) | output: (?, ?, 100)
dense_5: Dense | input: (?, ?, 261) | output: (?, ?, 100)
dense_7: Dense | input: (?, ?, 262) | output: (?, ?, 100)
dense_4: Dense | input: (?, ?, 100) | output: (?, ?, 10)
dense_6: Dense | input: (?, ?, 100) | output: (?, ?, 10)
dense_8: Dense | input: (?, ?, 100) | output: (?, ?, 10)
concatenate: Concatenate | input: [(?, ?, 10), (?, ?, 10), (?, ?, 10)] | output: (?, ?, 30)
dense_9: Dense | input: (?, ?, 30) | output: (?, ?, 10)

- Our neural network is exactly as defined by pytorch
- Custom defined cut layers can be defined (or not!)

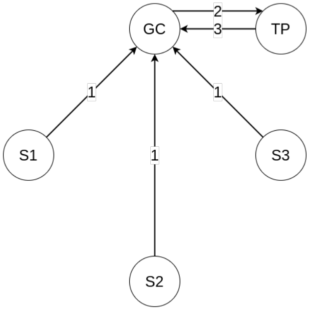
Different network architectures

Depending on the scenario, the cut layer will be different.

Unencrypted



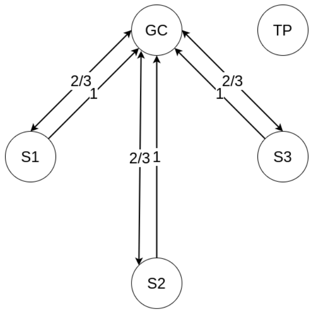
Paillier



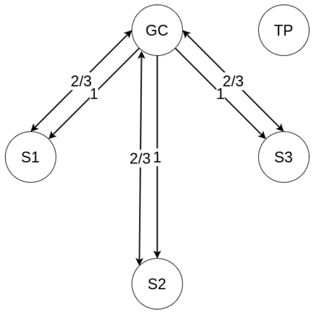
Different network architectures

Depending on the scenario, the cut layer will be different.

Threshold-Paillier



SMPC



User alignment by private set intersection

Parties assumed not to share the same sample IDs (require user alignment)
Private set intersection computes the intersection of sets of data without exposing them.

- Naive hashing (insecure against guessing attacks)
- Circuit-based (high comms, does not scale well)
- Public-key based (**RSA blind - linear complexity**)
    - widely available implementations
- Oblivious transfer (current SOTA)

2 kinds of PSI

- 2-party (Algorithm 1)
- multi-party

User alignment - general 2-party algorithm

Algorithm 1: User-Align

```
input  : Node ID of the current node $n$
         Number of nodes in the network $N$
         Current node's address $h = H[n]$
         Next node's address $h' = H[(n+1)\%N]$
         Current node's database $D$
output: User aligned database $D$
1  for $i \in [0, N-1]$ ;                          // Propagate $N-1$ times
2   do
3      if $n == 0$ then
4          Update $D$ = Align-To($h'$, D);              // Update using $h'$
5          Align-From($D$);                   // Wait for the previous node
6      else
7          Align-From($D$);
8          Update $D$ = Align-To($h'$, D);
9      end
10 end
11 return D;
```

Different privacy techniques

We are aiming to secure matrix multiplication (Linear layer $X \times W$) inside our cut layer. Using our framework, we can use:

- **Partially Homomorphic Encryption**
  - **Paillier (server-client) [pe]**
  - **Threshold-Paillier (distributed) [tpe]**
  - Other additively homomorphic encryption schemes
- **Secure multi-party computation [smpc]**

We implemented the items in **bold** and tested in our experiments. We also define **TP** here together with the privacy technique. We also implemented **Unsecure Cut Layer [une]** as a baseline.

Partially homomorphic encryption - Paillier

Implementation from *phe* library

1. Each **submodel** encrypts forward values using Paillier and sends results to **GC**
2. Within the cut layer, **GC** does matrix multiplication before sending to **TP**
3. **TP** decrypts, sends back to **GC** and training continues
4. Equivalent process happens in the backward pass

TP: Distribute keys, performs decryption

Partially homomorphic encryption - Threshold-Paillier

Modified by us to work with Pytorch (threshold we use is all **submodel**s)

1. Each **submodel** encrypts forward values using Paillier and sends results to **GC**
2. Within the cut layer, **GC** does matrix multiplication and sends parts to **submodel**s
3. Submodels decrypt and send each part back to **GC**
4. **GC** assembles results and continues training
5. Equivalent process happens in the backward pass.

TP: Distribute keys

Secure multi-party computation

Implementation from *PySyft*. Currently computed with only 2 **submodel**s (limitation of *PySyft*).

1. Matrix multiplication done in distributed manner
2. Secret shares distributed
3. **GC** sends weights to **submodel**s
4. Submodels do matrix multiplication and sends results to **GC**
5. **GC** assembles them, and continues training

TP: Not needed

Other Engineering Details

Some additional features:

- SSL encryption is used between any communication
- Websockets are used
- Large data is batched when sent over the network
- User alignment can be done seprately
- Implemented a facility to checkpoint the training (but not fully implemented)
- We can easily convert our model to a pytorch model **[une-pytorch]**
- We can also run our model locally without network **[une-framework]**

Experiments and datasets

Experiment Setup:

- Hardware: GCP VMs (e2-standard-4, e2-medium, e2-medium)
- Software: Conda, Python 3, Mlflow
- Expt: Take the average of 3 runs, splits are done as per dataset
- Aim 1: Understand system behavior and performance behavior
  - User alignment overhead
  - System overhead
- Aim 2: Show real-world application of our system

Datasets:

- MNIST handwritten digits - multi-class (0 to 9)
- Credit card fraud detection - binary classification of anonymized credit card transactions as genuine or fraudulent

Metrics

System overhead and real-world application

- Cross-entropy loss
- Test Accuracy / AUC-ROC (TPR against FPR. For unbalanced datasets - cc fraud. Higher is better)

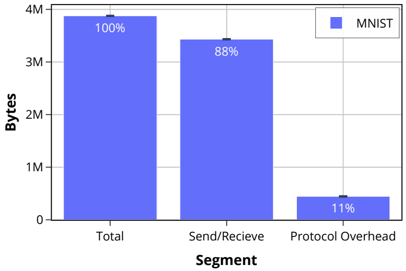
	# samples
Genuine	284315
Fraudulent	492

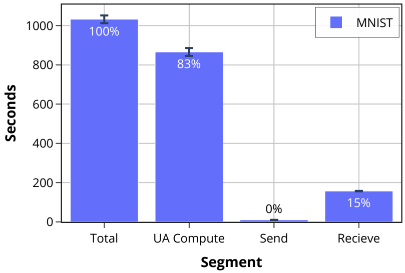
- Time taken to train

User alignment overhead

- Time
- Communication costs (data transferred / received)

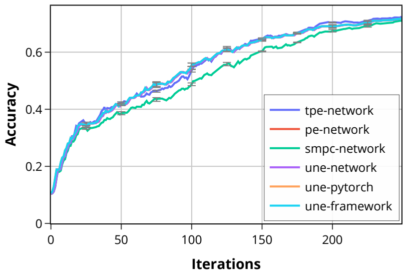
User alignment overhead (MNIST)

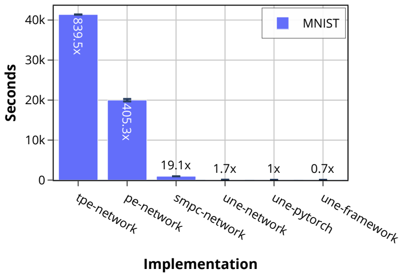




- RSA PSI algorithm takes quite some time to run
- Possible to use a multi-party algorithm to replace this to improve efficiency

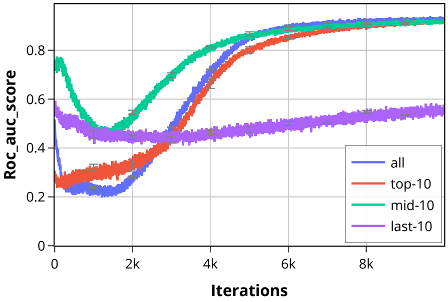
System overhead (MNIST)





- Accuracy differences are due to the noise introduced
- Differences show the various inefficiencies
- Interestingly, une-framework is faster than pytorch (parallel)

Real-world application (credit card fraud detection)



Features are PCA transformed into 30 features.

- ROC AUC, higher is better
- Motivates our VFL framework
- Works as expected on Real-world dataset
- *PS. we did not change any code.*

Our accomplishments

- Code mainly in **Python**, following Google Python style. Code sanity checks are needed.
  - ✓ Python, unit tests (Threshold-Paillier, key serialization, user alignment, expt using baselines)
- Use Pytorch, Singa, or TensorFlow as local training framework. Define local training framework abstract such that anyone can change it easily through configuration.
  - ✓ Pytorch (with ability to extend to other frameworks)
- Make each party's local NN architecture configurable such that the parties' local model architectures can be different.
  - ✓ Configurable through YAML, flexible network architectures
- Apply partially homomorphic encryption for **secure aggregation** on the cut layer in the split learning approach.
  - ✓ Supports *both* Paillier, Threshold-Paillier

Conclusion - Final Thoughts

Went above and beyond:

- Demonstrate Cut-Layer configurablity - implementing Secure multi-party computation & Threshold-Paillier (P2P)
- Demonstrate ability to swap out user alignment algorithms
- Implemented RSA-PSI algorithm
- Configurability in pytorch/yaml: load or not load statedict
- Extensive analysis of our UA and system performance

Takeaways: Homomorphic Encryption is not ready, more can be done. Generally the field of privacy has active interest (ie. PySyft).