ML-SDK MainNet Requirement

Author: *dev-goodata*

Summary

In the main net, the ML-SDK will mainly achieve three goals

1. Support vertical machine learning use case.
2. Add homomorphic encryption for updating models.
3. Add secret sharing MPC protocol.

Private Set Intersection

Currently there are four popular PSI Protocols:

1. Hash-based PSI: It is a naive solution in which both parties apply a cryptographic hash function to their inputs and compare these hashes.
2. Public-Key-Based PSI:
    a. A PSI protocol based on Diffie-Hellmann (DH) key agreement.
    b. A PSI protocol based on public-key cryptography (more specifically, blind-RSA operations), and scales linearly in the number of elements.
    c. A PSI protocol based on Bloom filter. These protocols also use public-key operations, linear in the number of elements.
    d. A PSI protocol based on oblivious pseudo-random.
    e. A PSI protocol based on polynomial interpolation and differentiation for finding intersections between multi-sets.
3. Circuit-Based PSI
    a. A PSI protocol based on GoldreichMicali-Wigderson.
    b. A PSI protocol based on Yao's garbled circuits protocol.
4. OT-Based PSI
    a. A Bloom-filter based protocol PSI based on OT extension
    b. A Oblivious pseudo-random function (OPRF) protocol based on the OT extension.

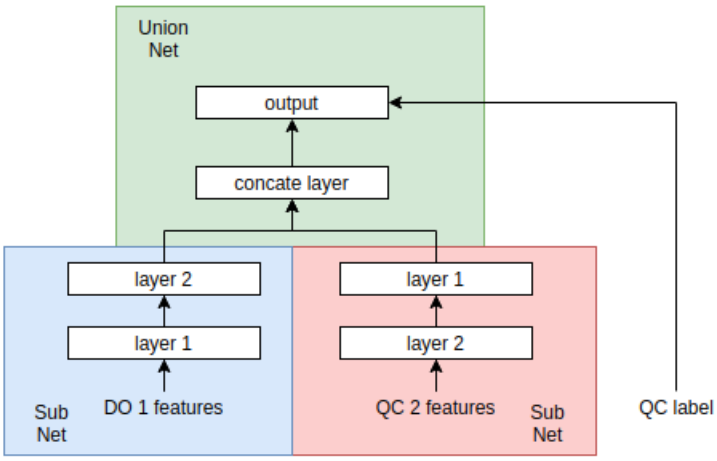
Setting	LAN					WAN			
Set Size n	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^8	2^{12}	2^{16}	2^{20}
Limited Security PSI									
Naive Hashing	**1**	**3**	**38**	**665**	**12,368**	**51**	**119**	**886**	**7,277**
Server-aided [38]	**1**	5	78	1,250	20,053	124	248	1,987	15,578
Public-Key-based PSI									
DH FFC [47]	386	5,846	88,790	1,418,772	22,681,907	3,577	56,786	880,075	11,557,061
DH ECC [47]	**231**	**3,238**	**51,380**	**818,318**	**13,065,904**	**1,949**	**28,686**	**466,606**	**5,007,681**
RSA [17]	779	12,546	203,036	3,193,920	50,713,668	10,508	166,453	1,356,757	21,094,586
Circuit-based PSI									
Yao SCS [34] ($\sigma = 32$)	**320**	3,593	74,548	-	-	**2,763**	20,826	518,136	-
GMW SCS [34] ($\sigma = 32$)	361	1,954	40,872	-	-	5,929	14,415	187,750	-
Yao PWC (§4.2, $\sigma = 32$)	428	2,294	28,491	-	-	4,248	17,897	178,522	-
GMW PWC (§4.2, $\sigma = 32$)	460	**1,324**	**10,656**	**124,732**	-	2,872	**7,644**	**59,572**	**472,687**
Yao OPRF (§4.3)	968	12,518	-	-	-	6,001	65,156	-	-
GMW OPRF (§4.3)	690	6,672	101,231	-	-	6,939	27,660	386,243	-
OT-based PSI									
Bloom Filter [23]	105	448	4,179	65,218	-	1,248	5,424	31,581	345,484
Random Bloom Filter [64]	**95**	**346**	2,991	49,171	-	**968**	3,863	22,031	220,570
OT (§5) + Hashing (§3)	311	362	**702**	**5,847**	**86,278**	2,278	**2,915**	**8,215**	**58,418**

Comparing different protocol computing time('-' means out of memory)

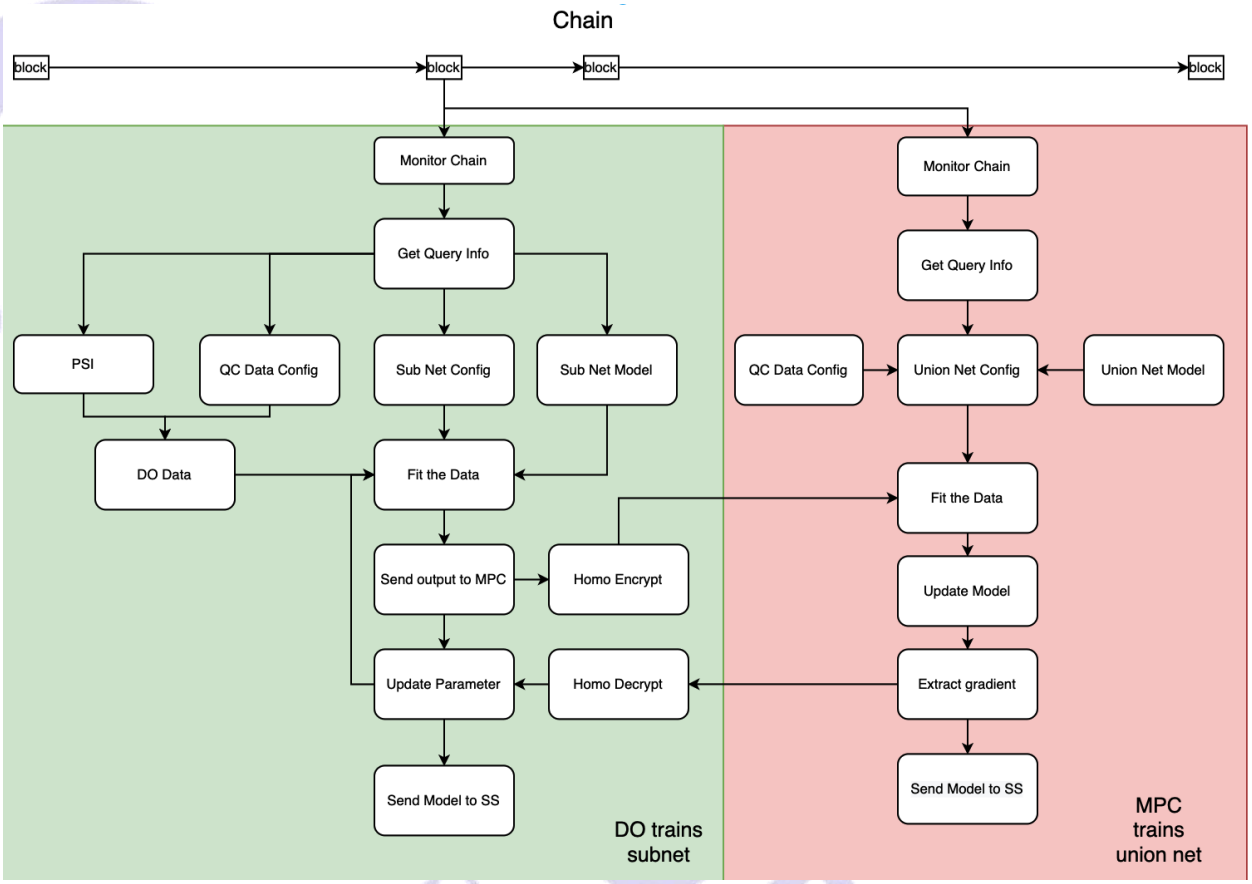
In the above choices, we prefer option 2 and 4. Because option 1 is obviously too simple and not secured. The option 3 requires to design a circuit in the underlying code to achieve the PSI. OT-Based PSI is the most popular protocol for PSI, and it has the fastest speed compared to the rest protocol, but in the MainNet 1.0, we decided to implement RSA Blind Signature-based PSI and In MainNet 2.0, we will start to implement OT-Based PSI to achieve a better performance. The reason we choose to do the RSA based protocol is because it is easier to understand and learn, and also OpenMined has a public implementation for this algorithm. In MainNet 1.0 we will introduce several difficult algorithms, so choosing the one we are more confident to achieve is the reason we decided to implement RSA based protocol.

Vertical deep learning model training

In the vertical learning, the features are splitted into different parts and the label may only hold one data owner. It is impossible to send one common model to all DOs and training parallelly. To train a neural network under this senarial, we can utilize the structure of the neural network. To be more precise:



1. Each DO can train a sub neural net in their local, this sub neural net can be treated as a feature extraction net, which extracts and summarises the features information in the DO.
2. DO sends the output of the subnet to a central server
3. Central server trains a new neural net which takes the DO outputs as input features with the label from QC
4. The central server sends the gradients back to each DO
5. DO update the parameters with the gradients from the central server.



DO (training sub net):

1. DO monitor chain to receive the signal for joining a training job.
2. Each DO should receive a model file, config (include data config and model config), intersection set keys, MPC ip, homomorphic encryption public/private key.
3. Prepare data for training Sub Net. def process_data(data, data_config, psi_keys): return a data loader
    a. data is the DO's data,
    b. data_config is used for DO to process the data based on the QC requirements,
    c. psi_keys is an intersection set between all DO's data.
4. Load the model sent by QC and train. We create a model class for training pytorch model class PytorchModel(BaseMmodel), it includes methods:
    a. def __init__(self, model, model_config, he_keys): initialize the PytorchModel class
        i. model: the model structure sent by QC.
        ii. Model_config: the model related setups.
        iii. he_keys: the homomorphic encryption public and private keys, are used to encrypt and decrypt the data.
    b. def fit(self, dataloader): Fit the model with training data
    c. def predict(self, data): make the prediction of the data

d. def evaluate(self, data, evaluate_metrics): calculate the evaluation metric for given data on the trained model.
e. def load_model(self, model_dict): Load the model from the SS
f. def save_model(self, model_dict): Save the model to the local path
g. def send_log_to_chain(self, chain, log_info): send the training log to the Chain

5. DO use Homomorphic Encryption to encrypt the output from Sub Net, and send to the node to train the Union Net.
    a. def encryptor(tensor, public_key): encrypt the output tensor
    b. def send_subnet_output_to_mpc(encrypted_data, mpc_node): Send the encrypted subnet output to the MPC cluster and receive the encrypted gradient from MPC.
6. DO update the the Sub Net parameters based on the gradient received from Union Net:
    a. def decryptor(union_net_gradient, sceret_key): decrypt the gradient from MPC
    b. def update_param(sub_net, union_net_gradient): update the parameters in the SubNet based on the gradient received from MPC.

MPC (training union net):

1. DO monitor chain to receive the signal for joining a training job.
2. MPC should receive model file, config (include data config and model config), DOs' ip.
3. Load the model sent by QC and data from DOs, using class PytorchModel(BaseMmodel) to train the union net.
4. Extract the first layer's gradient and send back to each DO.
    a. def get_first_layer_gradient(trained_union_net, data_config, DO_ip): extract the gradient, use data config to figure out the source DO for the first layer's input data, split and prepare the gradient for each DO.
5. Send the gradient back to each DO
6. Send the same training log as the subnet to the chain.

Interactions with SS and Chain

SS

1. Support to send a specific model to a node. Each DO may have different SubNet structure based on their feature type, so if multiple models are existing for a vertical training task, SS needs to know which model should go to which DO.
2. SS needs to send the Union Net to the MPC.
3. MPC can make prediction with the whole model (MainNet 2.0)
4. QC need to initial a pair of public and private key and send public key to each DO to do the encryption and decryption

Homomorphic encryption

Homomorphic encryption (HE) can let the mathematical operations applied on the encrypted text get the same result as the plain text, so the HE algorithms are variated based on what kind of mathematical operation is going to apply on encrypted text.

1. Pailler: Paillier encryption is a kind of addition homomorphic encryption which belongs to a probabilistic asymmetric algorithm.
2. Gentry: The first fully homomorphic encryption scheme(FHE) using ideal lattices which supports both addition and multiplication operations on ciphertexts.
3. GSW: GSW provides a new technique for building FHE schemes that avoids an expensive "relinearization" step in homomorphic multiplication.
4. CKKS: CKKS scheme supports efficient rounding operations in encrypted state. CKKS is focused as a solution for encrypted machine learning. CKKS scheme is constructed to deal efficiently with the errors arising from the approximations. The scheme is familiar to machine learning which has inherent noises in its structure.

Implementing the HE algorithm is the most challenging part, so instead of developing our own library, the efficient way is to find the existing libraries. We find Microsoft has implemented the CKKS scheme with C++ and OpenMined built a library based on that to encrypted tensor, so those two libraries will be the primary libraries we choose the use, also we will try to build a Python wrapper on the Microsoft C++ library to offer us more flexibility to use HE. The detailed implementation and algorithm review will be done by Gouxi Wu in a separated doc.

Secret sharing

For secret sharing, we choose the SPDZ protocol. SPDZ allows us to have as few as two parties computing on private values, and it is popular in the past few years with several optimisations that are known that can be used to speed up the computation. The detailed implementation and algorithm review will be done by Gouxi Wu in a separated doc.