FIT5124 Assignment 2 – Neural Network Inference Scenario

Xiaohui Ding 31291252

3.1 Plaintext Algorithm

The plaintext algorithm describes the process of using the Inference of DNN (an already trained database/model) to classify the data in some dataset.

For simplicity, we will use Fashion-MNIST train set and test set, and we will only consider forward propaganda. The NN model is 3-layers fully connected.

Referring to week 7 lecture note page 88, the forward propaganda contains 2 computational parts. I will use 1 picture to explain the plaintext algorithm below. The dataset works as the same.

1) Linear functions

a) Multiplications

- User sends a MNIST picture to Server. The picture has 28 * 28 = 784 pixels, represented as $\vec{x} = [x_1, x_2, \dots, x_{784}]$

- Server has a trained model that has a set of weights and biases. In our case, the NN is 3 layers fully connected. So, the weights will be divided into 3 matrices: $[\#L1 * \#L2], [\#L2 * \#L3], [\#L3 * \#Label]$. I will represent each layer's weights as $\overrightarrow{w_1} = [784 * 300], \overrightarrow{w_2} = [300 * 100], \overrightarrow{w_3} = [300 * 10]$. The user input $\vec{x} = [x_1, x_2, \dots, x_{784}]$ will firstly do inner product with $\overrightarrow{w_1} = [784 * 300]$, and this will output a vector [300 * 1], which will be used in inner product of layer 2. Layer 2, 3 is also similar to this.

b) Additions

- Pre-activation: for each layer, do additions: $z = \sum_{i=1}^{n} w_i x_i + b$

- In our case, we don't have backward propagation, so we don't have to update the weight. But when training the model, it really need the update to obtain the weights and bias.

2) Non-linear functions (ReLU)

a) ReLU: $max(0, x)$

- For each layer, the pre-activation will go through ReLU to generate next layer inputs.

- Layer output: $a = \sigma(z) = \max(0, z)$

We can summarize the process in the diagram below.

--layer 1--

Input 1 pic, 28*28: $\vec{x} = [x_1, x_2, \dots, x_{784}]$

Layer1:

$W_{784*300} = [[w_1, w_2, \dots, w_{784}]_1^1, \dots, [w_1, w_2, \dots, w_{784}]_{300}^1] = \left[\overrightarrow{w_1^1}, \dots, \overrightarrow{w_{300}^1}\right]$

$\overrightarrow{b_1} = [b_{1,1}, \dots, b_{1,300}]$

Matrix multiplication:

$z_{1,1} = \vec{x} \cdot \overrightarrow{w_1^1} + b_{1,1}, \dots, z_{1,300} = \vec{x} \cdot \overrightarrow{w_{300}^1} + b_{1,300}$

$\overrightarrow{z_1} = [z_{1,1}, z_{1,2}, \dots, z_{1,300}]$

ReLU:

$\overrightarrow{a_1} = \max(0, \overrightarrow{z_1}) = [a_{1,1}, \dots, a_{1,300}]$

--layer 2--

Input: $\overrightarrow{a_1} = [a_{1,1}, \dots, a_{1,300}]$

Layer2:

$W_{300*100} = [[w_1, w_2, \dots, w_{300}]_1^2, [w_1, w_2, \dots, w_{300}]_2^2, \dots, [w_1, w_2, \dots, w_{300}]_{100}^2] = \left[\overrightarrow{w_1^2}, \dots, \overrightarrow{w_{100}^2}\right]$

$\overrightarrow{b_2} = [b_{2,1}, \dots, b_{2,100}]$

Matrix multiplication:

$z_{2,1} = \overrightarrow{a_1} \cdot \overrightarrow{w_1^2} + b_{2,1}, \dots, z_{2,100} = \overrightarrow{a_1} \cdot \overrightarrow{w_{100}^2} + b_{2,100}$

$\overrightarrow{z_2} = [z_{2,1}, z_{2,2}, \dots, z_{2,100}]$

ReLU:

$\overrightarrow{a_2} = \max(0, \overrightarrow{z_2}) = [a_{2,1}, \dots, a_{2,100}]$

--layer 3--

Input: $\overrightarrow{a_2} = [a_{2,1}, \dots, a_{2,100}]$

Layer3:

$W_{100*10} = [[w_1, w_2, \dots, w_{100}]_1^3, [w_1, w_2, \dots, w_{100}]_2^3, \dots, [w_1, w_2, \dots, w_{100}]_{10}^3] = \left[\overrightarrow{w_1^3}, \dots, \overrightarrow{w_{10}^3}\right]$

$\overrightarrow{b_3} = [b_{3,1}, \dots, b_{3,10}]$

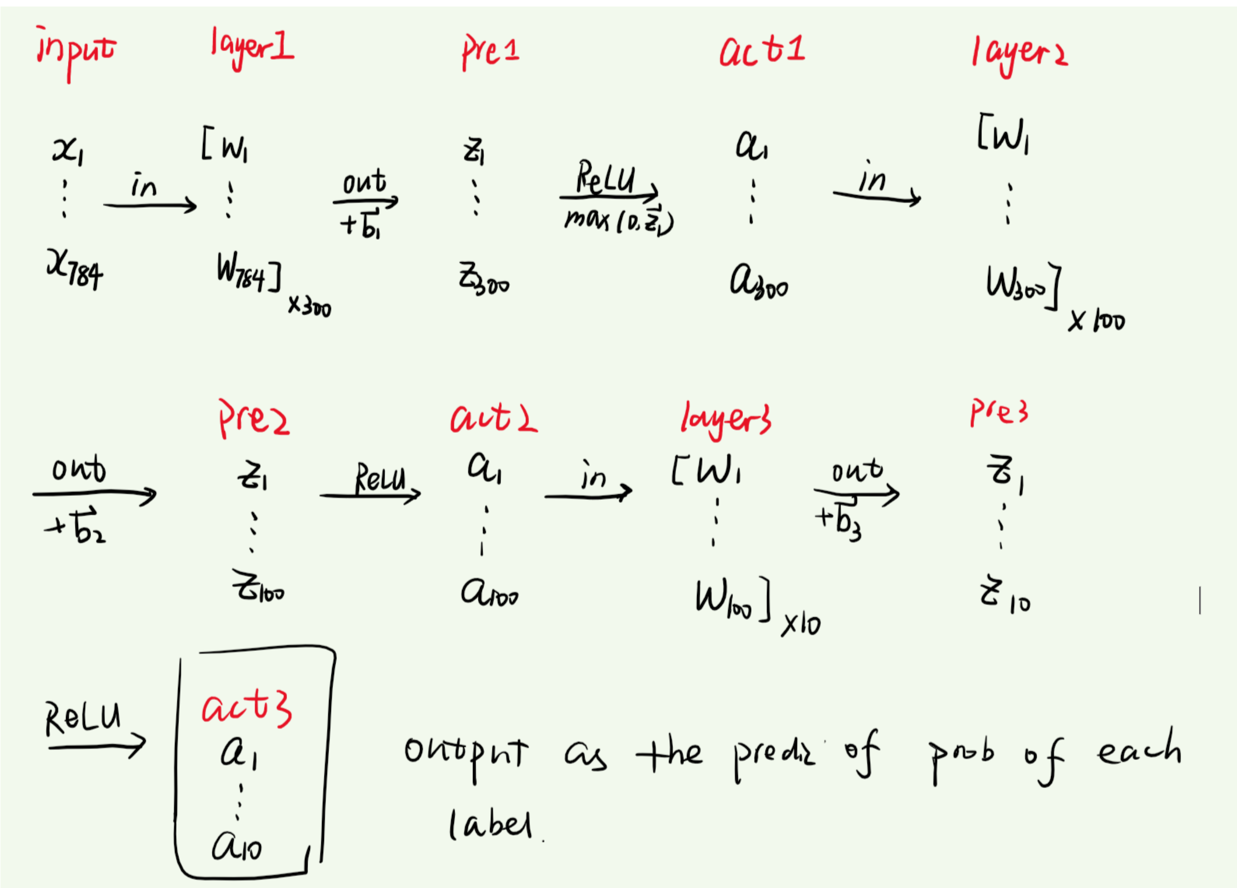
Matrix multiplication:

$z_{3,1} = \overrightarrow{a_2} \cdot \overrightarrow{w_1^3} + b_{3,1}, \dots, z_{3,10} = \overrightarrow{a_2} \cdot \overrightarrow{w_{10}^3} + b_{3,10}$

$\overrightarrow{z_3} = [z_{3,1}, z_{3,2}, \dots, z_{3,10}]$

ReLU:

$\overrightarrow{a_3} = \max(0, \overrightarrow{z_3}) = [a_{3,1}, \dots, a_{3,10}]$



3.2 Protocol Overview

We will also use 1 picture to discuss the whole situation. Referring to week 7 lecture note page 89, privacy preserving DNN Inference scenario is as below:

1) concrete application scenario

Data owner (client) will split the input image to 2 non-colluding servers, the 2 servers do multiplication MPC and addition MPC with the known and split weights and the known bias (both from the trained model). After layer 3, by A2Y, the 2 servers convert their adding results to Yao's share as one of the Yao's labels. Another label will be 0 because the non-linear function is $max(0, x)$. After GC, the output will also be converted back to Arithmetic shares by Y2A.

2) parties involved

	Party 1	Party 2	Output (Secured result)
Multiplication Share	Data Owner ($\vec{x} \to \overrightarrow{x_a}$)	Server Alice ($\overrightarrow{w_a}$, b)	$\overrightarrow{x_a} \cdot \overrightarrow{w_a}$
	Data Owner ($\vec{x} \to \overrightarrow{x_b}$)	Server Bob ($\overrightarrow{w_b}$, b)	$\overrightarrow{x_b} \cdot \overrightarrow{w_b}$
Addition Share	Server Alice	Server Bob	Alice: z0, Bob: z1
Garbled Circuit	Server Alice	Server Bob	z ← A2Y(z0, z1, ADD) a ←GC(0, z, max(0, z)) output ← Y2A

3) security and privacy goals

To secure that in each phase of computation (multiplication, addition, and GC), the private input will not be leaked. The only leakage should be the final model/activation output.

3.3 Protocol Design (40 Marks)

1) How to use SMC techniques to realise the plaintext algorithm in a privacy-preserving manner

I drew a diagram to present the whole design of this protocol in details:

User data	$\vec{\boldsymbol{x}} = [x_1, x_2, \dots, x_{784}] = \overrightarrow{x_A}^{(392)} + \overrightarrow{x_B}^{(392)}$	
Server	Alice	Bob
Initial knows	$\overrightarrow{w_{a1}} = [w_1, w_2, \dots, w_{392}]_{300}, \overrightarrow{b_1}$	$\overrightarrow{w_{b1}} = [w'_1, w'_2, \dots, w'_{392}]_{300}, \overrightarrow{b_1}'$
	$\overrightarrow{w_{a2}} = [w_1, w_2, \dots, w_{150}]_{100}, \overrightarrow{b_2}$	$\overrightarrow{w_{b2}} = [w'_1, w'_2, \dots, w'_{150}]_{100}, \overrightarrow{b_2}'$
	$\overrightarrow{w_{a3}} = [w_1, w_2, \dots, w_{50}]_{10}, \overrightarrow{b_3}$	$\overrightarrow{w_{b3}} = [w'_1, w'_2, \dots, w'_{50}]_{10}, \overrightarrow{b_3}'$

---Layer 1---

Private inputs	User: $\overrightarrow{x_A}^{(392)}$	User: $\overrightarrow{x_B}^{(392)}$
	Alice: $\overrightarrow{w_{a1}}$	Bob: $\overrightarrow{w_{b1}}$
Mul shares	User: $\left(\overrightarrow{x_{A0}}^{(392)}, \overrightarrow{x_{A1}}^{(392)}\right)$	User: $\left(\overrightarrow{x_{B0}}^{(392)}, \overrightarrow{x_{B1}}^{(392)}\right)$
	Alice: $\left(\overrightarrow{w_{a10}}, \overrightarrow{w_{a11}}\right)$	Bob: $\left(\overrightarrow{w_{b10}}, \overrightarrow{w_{b11}}\right)$
After Mul	$\overrightarrow{z_1^A} = \overrightarrow{x_A}^{(392)} * \overrightarrow{w_{a1}} + \overrightarrow{b_1}$	$\overrightarrow{z_1^B} = \overrightarrow{x_B}^{(392)} * \overrightarrow{w_{b1}} + \overrightarrow{b_1}'$

* here represents matrix multiplication. $\overrightarrow{x_A}^{(392)}$ is 1D vector, but $\overrightarrow{w_{a1}}$ is a 2D vector (matrix). So, the multiplication should be matrix multiplication rather than inner product. The inner product will happen between $\overrightarrow{x_A}^{(392)}$ and each vector inside $\overrightarrow{w_{a1}}$. Same for Bob.

A2Y shares	$(\overrightarrow{z_{10}^A}, \overrightarrow{z_{11}^A})$		$(\overrightarrow{z_{10}^B}, \overrightarrow{z_{11}^B})$
GC - A2Y	$\overrightarrow{z_{10}^A}$ → | ADD | ← $\overrightarrow{z_{11}^A}$		$\overrightarrow{z_{10}^B}$ → | ADD | ← $\overrightarrow{z_{11}^B}$
A2Y output		→ $\overrightarrow{z_1}$ ←	
GC - ReLU	0 →	| max(0, $\overrightarrow{z_1}$) |	← $\overrightarrow{z_1}$
Activation output		→ $\overrightarrow{a_1} = max(0, \overrightarrow{z_1})$ ←	
Y2A	$\overrightarrow{a_{10}}$	$\overrightarrow{a_1} = \overrightarrow{a_{10}} + \overrightarrow{a_{11}}$	$\overrightarrow{a_{11}}$

---Layer 2---

Similar approach as Layer1

---Layer 3---

Similar approach as Layer1

Activation output → $\overrightarrow{a_3} = max(0, \overrightarrow{z_3})$ ←

This is our wanted result.

2) How the algorithm can be computed securely among the parties involved

Notice: all the arithmetic shares are calculated under module calculation, i.e., $mod\ 2^l$.

The following explanations are based on Week6 lecture. The protocol will adapt the general methods for all these secure computations.

- Multiplication shares

In the Offline phase, a trusted party generates $([a], [b], [c]) < -RandMul()$ for each round of computation.

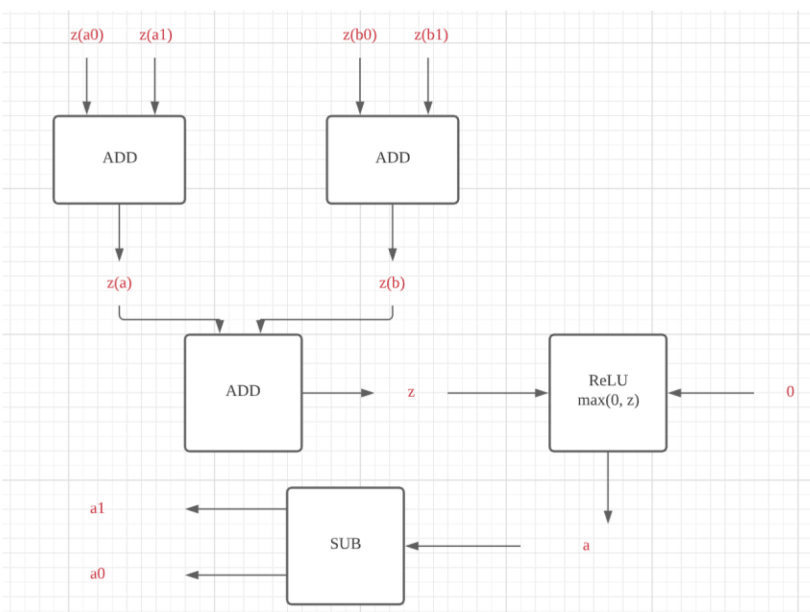
In Online phase, the 2 parties will jointly recover: $e = [a] + [x], j = [b] + [w]$. Finally, they can compute $[z] = [c] + e[w] + f[x] - ef$.

This will achieve the security and privacy goal: Alice will know $\overrightarrow{x_A} \cdot \vec{w}$, but not know $\overrightarrow{x_A}$ itself. Same for Bob.

- Addition shares & A2Y

After each pre-activation in each layer, Alice and Bob generate their add share. Since the non-linear part will use GC, they also convert their pairs of shares to a single Yao's share, respectively.

- Garbled circuit with mix protocol



1) A2Y

Alice and Bob will use A2Y to convert and add their addition shares. Their add shares will go through an ADD GC gate, so the outputs will be Alice's and Bob's private inputs.

2) ReLU GC

Let Alice be generator and Bob be evaluator. The circuit will be max(0,z).

Alice will generate random seed for herself and Bob, also encrypt the result 0 and z by using random seeds. She then sends the GC encrypted lookup table and her key to Bob.

The input z and 2 seeds of Bob will go through the Oblivious Transfer together and return to Bob his chosen seed.

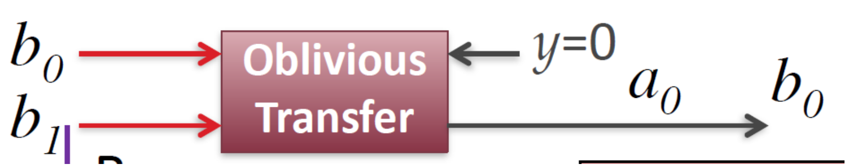
Bob then uses his and Alice's seeds to decrypt the results in the lookup table and send the results to Alice.

Alice will reveal the final result and shares it with Bob.

3) Y2A GC

After each $\overrightarrow{a_n}$ is revealed, it will go through another SUB gate and split into $\overrightarrow{a_{n0}}$ & $\overrightarrow{a_{n1}}$ arithmetic shares for the next layer's calculation.

- Oblivious Transfer



Referring to Week6 lecture note page 78, Alice will input Bob's random seeds b0, b1, and her chosen random seed a0. Bob will input his private input y and thus will obtain b0.

- Truncation

We also need to think about the truncation. So, in each of the calculation, the float inputs will be converted to integer format to do the arithmetic calculation. But the result will double the decimal places. So the paper suggested a way to simply truncate the redundant places, which will only have neglect effect on results.

3.4 Protocol Analysis (10 Marks)

This is a MPC + ML protocol, so we will analyze both security and accuracy. The complexity of this complicated protocol is important too.

- Security

There are 3 kinds of calculation in the plaintext which are multiplication, addition, and ReLU function. In the privacy-preserving scenario, these 3 phases are all protected by arithmetic sharing and garbled circuit. Nothing is leaked except the final result.

- Accuracy

The 3-layers fully connected neural network is trained previously. The training process includes forward and backward, so the obtained weights & biases are very accurate.

In the privacy-preserving scenario, we only consider forward situation, so the accuracy will purely depend on the known weights & biases. But there might be some interference in the process, such as the package loss of throughput.

- Efficiency/Complexity

The main time cost will happen in the offline phase because in this phase there are plenty of random multiplication triplets being generated. Comparing to the offline phase, the online phase will cost less time.

But the online phase will take most of communication cost and computational cost. The communication cost is caused by the two servers. There are multiplication shares, addition shares, and garbled circuit, so the communication cost should be 3 times of the plaintext.

The computational cost is twice of the plaintext in the arithmetic share phrase, as mentioned in the paper. ReLU and garbled circuit also increase some overhead in this protocol.

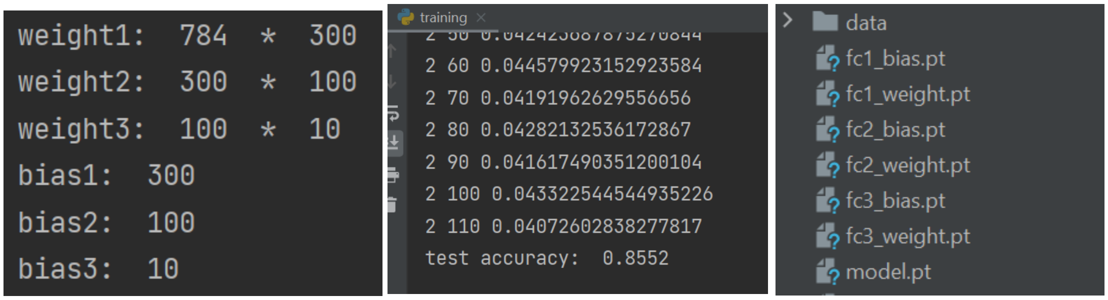
3.5 Implementation and evaluation (30 Marks)

0) Generate original weights & biases

Using a complete training algorithm(including forward/backward) to train a 3-layer model and save the weights & bias.

Code: See Appendix 0

Results:

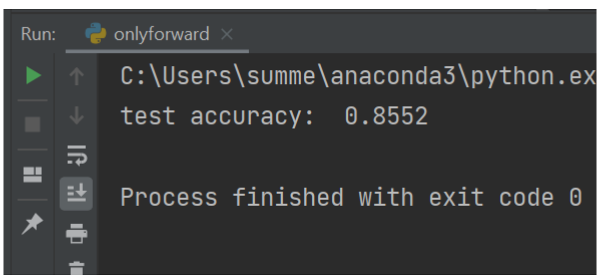


1) Plaintext training

We will now remove the backward part and use the trained 3-layers weights and biases.

Code: See Appendix 1

Result:



This accuracy is exactly the same with the trained model, which means the loading of model is successful.

2) MPC

I really don't know how to use C++, so I decided to write MPC methods in python. Unfortunately, I haven't finished implemented them all. I only implemented Add shares.

Code: See Appendix 2

- Arithmetic Shares

Add:

```
- Add Share Test -
Please enter Alice's private input: 123
Please enter Bob's private input: 456
Alice result:  579
Bob result:  579
```

Mul:

I’ve done partial of it. It’s not hard, but I don’t have time to finish it.

- GC

The implementation of GC will be like this:

Server has instant variable for private input. The generator server will have method of encrypting, decrypting, generating and sending, meanwhile the evaluator server will have method of calculating and sending back the decrypted lookup table result.

They both have OT in the method.

- A2Y & Y2A

This can be implemented by replacing the ReLU in GC.

- Truncation

After transfer the float to integer calculation, using round() function to simply round the result.

Appendix 0 – Training a model

net.py

```
# Ref:
https://blog.csdn.net/xjm850552586/article/details/109171016

from torch import nn
from torch.nn import functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # wx+b
        self.fc1 = nn.Linear(28 * 28, 300)
        self.fc2 = nn.Linear(300, 100)
        self.fc3 = nn.Linear(100, 10)

    def forward(self, x):
        # ReLU non linear
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

training.py

```
# Reference:
https://blog.csdn.net/hxxjxw/article/details/105667269

import torch
import torchvision
from torchvision import transforms
from torch.nn import functional as F
from torch import optim
from utils import one_hot
import net


batch_size = 512
# num of picture / time

transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5,), (0.5,))
])
```

```
trainset = torchvision.datasets.MNIST(
    root='./data/FashionMNIST',
    train=True,
    download=True,
    transform=transform
)

train_loader = torch.utils.data.DataLoader(
    dataset=trainset,
    batch_size=batch_size,
    shuffle=True
)

testset = torchvision.datasets.MNIST(
    root='./data/FashionMNIST',
    train=False,
    download=True,
    transform=transform
)

test_loader = torch.utils.data.DataLoader(
    dataset=testset,
    batch_size=batch_size,
    shuffle=True
)

model = net.Net()  # Create a model

# [w1, b1, w2, b2, w3, b3] initialize weights & biases
optimizer = optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)

train_loss = []

for epoch in range(3):

    for idx, data in enumerate(train_loader):
        inputs, labels = data
        inputs = inputs.view(inputs.size(0), 28 * 28)
        outputs = model(inputs)
        labels_onehot = one_hot(labels)
        loss = F.mse_loss(outputs, labels_onehot)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
        if idx % 10 == 0:
```

```
            print(epoch, idx, loss.item())


total_correct = 0

for idx, data in enumerate(test_loader):
    inputs, labels = data
    inputs = inputs.view(inputs.size(0), 28 * 28)

    outputs = model(inputs)
    pred = outputs.argmax(dim=1)
    correct = pred.eq(labels).sum().float().item()

    total_correct += correct

accuracy = total_correct / len(test_loader.dataset)
print('test accuracy: ', accuracy)



dic = model.state_dict()
key = dic.keys()
print(key)

torch.save(model.state_dict(), "./model.pt")

# obtain weights & biases
output_1_bias = model.fc1.bias.data
output_1_weight = model.fc1.weight.data
save_1_bias = torch.save(output_1_bias, "./fc1_bias.pt")
save_1_weight = torch.save(output_1_weight, "./fc1_weight.pt")

output_2_bias = model.fc2.bias.data
output_2_weight = model.fc2.weight.data
save_2_bias = torch.save(output_2_bias, "./fc2_bias.pt")
save_2_weight = torch.save(output_2_weight, "./fc2_weight.pt")

output_3_bias = model.fc3.bias.data
output_3_weight = model.fc3.weight.data
save_3_bias = torch.save(output_3_bias, "./fc3_bias.pt")
save_3_weight = torch.save(output_3_weight, "./fc3_weight.pt")
```

utils.py

```
# Reference:
https://blog.csdn.net/hxxjxw/article/details/105667269
```

```
import torch
from matplotlib import pyplot as plt


def plot_curve(data):
    fig = plt.figure()
    plt.plot(range(len(data)), data, color='blue')
    plt.legend(['value'], loc='upper right')
    plt.xlabel('step')
    plt.ylabel('value')
    plt.show()


def plot_image(img, label, name):
    fig = plt.figure()
    for i in range(6):
        plt.subplot(2, 3, i + 1)
        plt.tight_layout()
        plt.imshow(img[i][0] * 0.5 + 0.5, cmap='gray',
interpolation='none')
        plt.title("{}: {}".format(name, label[i].item()))
        plt.xticks([])
        plt.yticks([])
    plt.show()


def one_hot(label, depth=10):
    out = torch.zeros(label.size(0), depth)
    idx = torch.LongTensor(label).view(-1, 1)
    out.scatter_(dim=1, index=idx, value=1)
    return out
```

test.py

```
import torch

weight1 = torch.load("./fc1_weight.pt")
weight2 = torch.load("./fc2_weight.pt")
weight3 = torch.load("./fc3_weight.pt")

bias1 = torch.load("./fc1_bias.pt")
bias2 = torch.load("./fc2_bias.pt")
bias3 = torch.load("./fc3_bias.pt")

print("weight1: ", len(weight1[1]), " * ", len(weight1))
print("weight2: ", len(weight2[1]), " * ", len(weight2))
print("weight3: ", len(weight3[1]), " * ", len(weight3))
```

```
print("bias1: ", len(bias1))
print("bias2: ", len(bias2))
print("bias3: ", len(bias3))
```

Appendix 1 – Plaintext algorithm

```
# Reference:
https://blog.csdn.net/hxxjxw/article/details/105667269

import torch
import torchvision
from torchvision import transforms
import net


transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5,), (0.5,))
])

testset = torchvision.datasets.MNIST(
    root='./data/FashionMNIST',
    train=False,
    download=True,
    transform=transform
)

test_loader = torch.utils.data.DataLoader(
    dataset=testset,
    shuffle=True
)

# load model
model = net.Net()  # Create a model
model.load_state_dict(torch.load("./model.pt"))
model.eval()


total_correct = 0

for idx, data in enumerate(test_loader):
    inputs, labels = data
    inputs = inputs.view(inputs.size(0), 28 * 28)

    outputs = model(inputs)
```

```
    pred = outputs.argmax(dim=1)
    correct = pred.eq(labels).sum().float().item()

    total_correct += correct

accuracy = total_correct / len(test_loader.dataset)
print('test accuracy: ', accuracy)

# dic = model.state_dict()
# key = dic.keys()
# print(key)
```

Appendix 2 – MPC

server.py

```
# import
import random

mod_number = 2**16

# server type
class ADDServer:
    def __init__(self):
        self.private_input = None
        self.own_shares = None # x0,x1
        self.others_share = None # y0
        self.calculation_seed = None #z0
        self.result = None

    def get_private_input(self):
        return self.private_input
    def get_own_shares(self):
        return self.own_shares
    def get_others_shares(self):
        return self.others_shares
    def get_calculation_seed(self):
        return self.calculation_seed
    def get_result(self):
        return self.result

    def set_private_input(self,private_input):
        self.private_input = private_input
        return True
```

```python
    def set_own_shares(self, own_shares):
        self.own_shares = own_shares
        return True
    def set_others_shares(self, others_shares):
        self.others_shares = others_shares
        return True
    def set_calculation_seed(self, calculation_seed):
        self.calculation_seed = calculation_seed
        return True
    def set_result(self, result):
        self.result = result
        return True
class MulServer:
    def __init__(self):
        self.private_input = None
        self.own_shares = None # x0,x1
        self.others_share = None # y0
        self.random_pair = None # a0, b0
        self.calculation_seed = None # e0, f0, e1, f1, e, f, c0
        self.result = None # c

    def get_private_input(self):
        return self.private_input
    def get_own_shares(self):
        return self.own_shares
    def get_others_shares(self):
        return self.others_shares
    def get_random_pair(self):
        return self.random_pair
    def get_calculation_seed(self):
        return self.calculation_seed
    def get_result(self):
        return self.result

    def set_private_input(self,private_input):
        self.private_input = private_input
        return True
    def set_own_shares(self, own_shares):
        self.own_shares = own_shares
        return True
    def set_others_shares(self, others_shares):
        self.others_shares = others_shares
        return True
    def set_random_pair(self, random_pair):
        self.random_pair = random_pair
        return True
    def set_calculation_seed(self, calculation_seed):
```

```
        self.calculation_seed.append(calculation_seed)
        return True
    def set_result(self, result):
        self.result = result
        return True



# calculation type

class AriAdd:
    def __init__(self, alice, bob):
        self.alice = alice
        self.bob = bob

    def generate_random_shares(self, server):
        r = random.randint(1, server.private_input)
        own_shares = []
        own_shares.append(r % mod_number)
        own_shares.append((server.private_input - r) %
mod_number)
        server.set_own_shares(own_shares)

    def exchange_shares(self):
        bob_share = self.bob.get_own_shares()
        alice_share = self.alice.get_own_shares()
        self.alice.set_others_shares(bob_share[0])
        self.bob.set_others_shares(alice_share[1])

    def calculate_add(self):
        z0 = (self.alice.get_own_shares()[0] +
self.alice.get_others_shares()) % mod_number
        z1 = (self.bob.get_own_shares()[1] +
self.bob.get_others_shares()) % mod_number
        self.alice.set_calculation_seed(z0)
        self.bob.set_calculation_seed(z1)

    def generate_result(self):
        result = self.alice.get_calculation_seed() +
self.bob.get_calculation_seed()
        self.alice.set_result(result)
        self.bob.set_result(result)

    def do_calculate(self):
        self.generate_random_shares(self.alice)
        self.generate_random_shares(self.bob)
        self.exchange_shares()
```

```
        self.calculate_add()
        self.generate_result()

class AriMul:
    def __init__(self, alice, bob):
        self.alice = alice
        self.bob = bob

    def generate_random_shares(self, server):
        r = random.randint(1, server.private_input)
        own_shares = []
        own_shares.append(r % mod_number)
        own_shares.append((server.private_input - r) %
mod_number)
        server.set_own_shares(own_shares)

    def generate_random_seed(self):
        c = random.randint(1, mod_number-1)
        flag_int = False
        while not flag_int:
            a = random.randint(1, c)
            if a % c == 0:
                flag_int = True
        b = a % c


        a0 = random.randint(1, a)
        a1 = (a - a0) % mod_number
        b0 = random.randint(1, b)
        b1 = (b - b0) % mod_number

        alice_random_seed = []
        bob_random_seed = []

        alice_random_seed.append(a0)
        alice_random_seed.append(b0)
        bob_random_seed.append(a1)
        bob_random_seed.append(b1)

        self.alice.set_random_pair(alice_random_seed)
        self.bob.set_random_pair(bob_random_seed)

    def exchange_shares(self):
        bob_share = self.bob.get_own_shares()
        alice_share = self.alice.get_own_shares()
        self.alice.set_others_shares(bob_share[0])
        self.bob.set_others_shares(alice_share[1])
```

```python
    def calculate_ef(self):
        e0 = self.alice.get_own_shares()[0] -
self.alice.get_random_pair()[0]
        self.alice.set_calculation_seed(e0)
        f0 = self.alice.get_others_shares() -
self.alice.get_random_pair()[1]
        self.alice.set_calculation_seed(f0)

        e1 = self.bob.get_own_shares()[1] -
self.bob.get_random_pair()[0]
        self.bob.set_calculation_seed(e1)
        f1 = self.bob.get_others_shares() -
self.bob.get_random_pair()[1]
        self.bob.set_calculation_seed(f1)

        # exchange
        self.alice.set_calculation_seed(e1)
        self.alice.set_calculation_seed(f1)
        self.bob.set_calculation_seed(e0)
        self.bob.set_calculation_seed(f0)

        # cal
        ef_alice = self.alice.get_calculation_seed[0] +
self.alice.get_calculation_seed[1] \
                   + self.alice.get_calculation_seed[2] +
self.alice.get_calculation_seed[3]
        ef_bob = self.bob.get_calculation_seed[0] +
self.bob.get_calculation_seed[1] \
                   + self.bob.get_calculation_seed[2] +
self.bob.get_calculation_seed[3]
```

Test_scene.py

```python
import Server

# Test Add Shares
print("- Add Share Test -")
alice_add = Server.ADDServer()
bob_add = Server.ADDServer()
alice_private = int(input("Please enter Alice's private input:
"))
bob_private = int(input("Please enter Bob's private input: "))
alice_add.set_private_input(alice_private)
bob_add.set_private_input(bob_private)

Add_test = Server.AriAdd(alice_add, bob_add)
```

```
Add_test.do_calculate()
print("Alice result: ", alice_add.result)
print("Bob result: ", bob_add.result)
```