PySyft

一个深度学习隐私保护的框架

Introduction

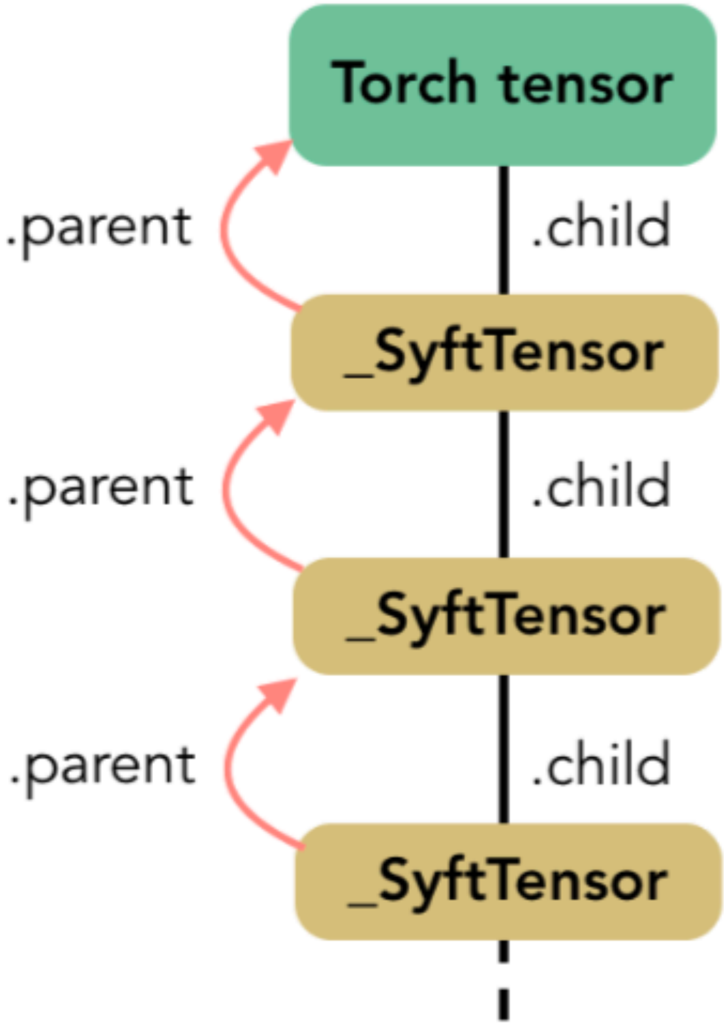
1. 首先建立一个通用的协议，方便workers之间进行交流，使得联合学习成为可能
2. 在tensors上开发链式抽象模型，以有效的覆盖操作或者编码新操作，比如在workers间发送或共享tensor
3. 提供使用这个新框架来实现差分隐私和多方计算的元素

A standardized framework to abstract operations on Tensors

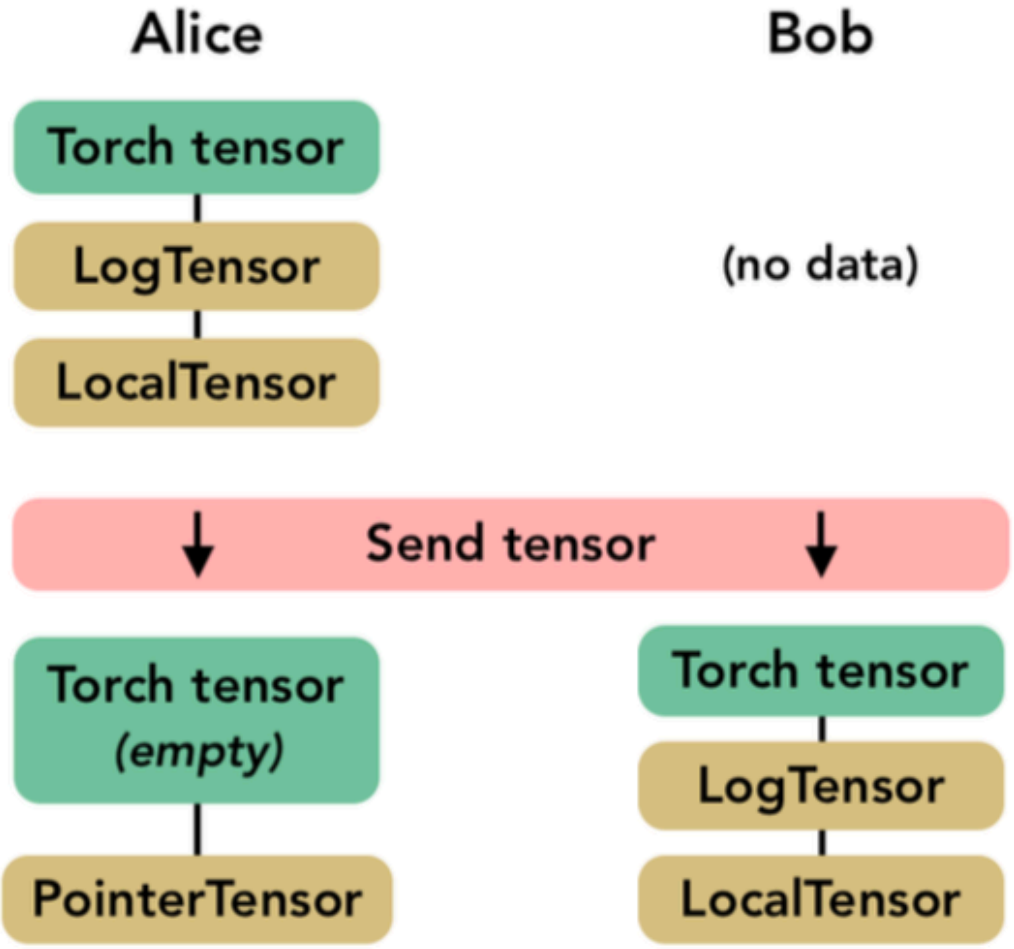
The chain structure

> SyftTensors表示数据的状态或者转换，并且可以链接在一起

1. Tensor chain的一般结构
    - 头部有Pytorch tensor，所有的操作都会首先应用于Torch tensor，使得其有本地的Torch接口，可通过访问子属性.child在链中进行传递
    - SyftTensors被子类的实例替换，每个实例具有不同的角色。



2. 有两个重要的子类
    - LocalTensor：在Torch tensor实例化时自动创建。作用是在Torch tensor上执行与重载操作相对应的本机操作
    - PointerTensor：当tensor发送给远方worker时被创建。可用`send(worker)`和`get()`方法来发送和获取一个tensor。
        - 如下图，当执行send tensor时，Alice的整个chain发送给Bob，自身的链结构替换为两个节点：一个空的Torch Tensor和记录着谁拥有数据及远程存储位置的PointerTensor



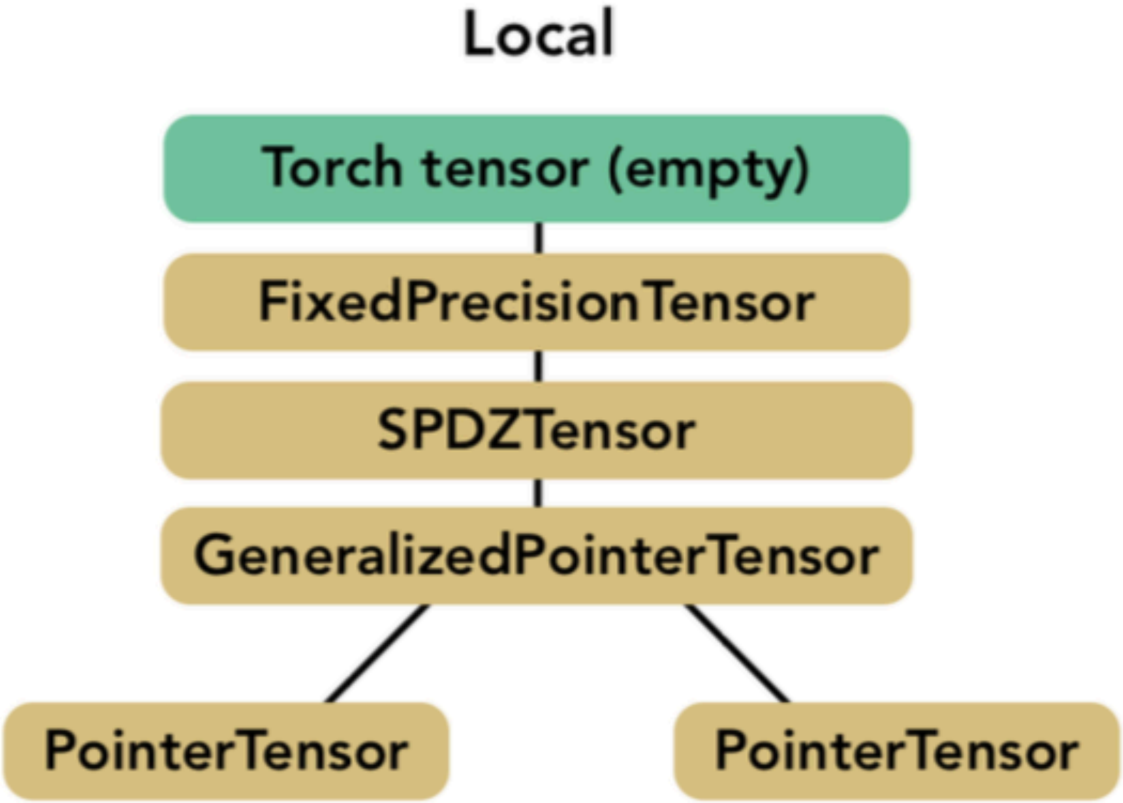
From virtual to real context execution of federated learning

1. 为了简化操作链调试起来很复杂，该框架开发了虚拟工作者（Virtual Workers）的概念
2. Virtual Workers都在同一台计算机上，不通过网络进行通信
3. Virtual Workers的作用：复制chain of commands并公开与actual workers相同的接口以便相互通信。

Towards a Secure MPC framework

Building an MPCTensor

1. 我们的框架中提出的MPC工具箱实现了SPDZ协议
2. MPC工具箱包括基本操作，如加法和乘法，还包括预处理工具，例如生成用于乘法的三元组，以及包括矩阵乘法在内的神经网络的更具体的操作。
3. SPDZ协议假定数据都是以整数的形式给出，因此在链中增加了FixedPrecisionTensor结点，此节点将值编码为整数并存储小数点的位置
4. 实现SPDZ的tensor完整结构如下所示：



5. 目前没有机制可以确保player诚实 有个方法：实现秘密共享值的MAC认证

pysyft运行机制（github）

```
安装pysyft的方法：
Install Python 3.5 or higher
Install PyTorch 0.3.1 (it MUST be this version)
Clone PySyft (git clone https://github.com/OpenMined/PySyft.git)
cd PySyft
pip install -r requirements.txt
python setup.py install
```

1. The Basic Tools of Private, Decentralized Data Science

1. 通常我们在保存数据的机器上进行深度学习，现在我们想在其他机器上执行这种计算，比如我们假设数据在其他机器上而不是我们自己的计算机
2. 对数据的操作采用pointer to tensor 而不是tensor。

2. pysyft 运用于 Federated Learning

> Federated Learning : Instead of bringing training data to the model (a central server), you bring the model to the training data (wherever it may live).

1. 联合学习的Training Steps，如part1.py所示

- send model to correct worker
- train on the data located there
- get the model back and repeat with next worker

```
# Run this cell to see if things work
import syft as sy
#Step 1: Initializing a Hook
#PySyft, at it's most basic level, is a set of overloaded operations
on major deep learning frameworks. (it takes a framework and modifies
its default behavior). The object we use to do this is called a Hook,
and we must "run a hook" in order to get PySyft's functionality.
hook = sy.TorchHook() # always run this when you import syft
from torch import nn, optim

# A Toy Dataset
data = sy.Var(sy.FloatTensor([[0,0],[0,1],[1,0],[1,1]]))
target = sy.Var(sy.FloatTensor([[0],[0],[1],[1]]))

# create a couple workers
bob = sy.VirtualWorker(id="bob")
alice = sy.VirtualWorker(id="alice")

# get pointers to training data on each worker by
# sending some training data to bob and alice
data_bob = data[0:2].send(bob)
target_bob = target[0:2].send(bob)

data_alice = data[2:].send(alice)
target_alice = target[2:].send(alice)

# organize pointers into a list
datasets = [(data_bob,target_bob),(data_alice,target_alice)]

# Initialize a Toy Model
model = nn.Linear(2,1)

def train():
    # Training Logic
    opt = optim.SGD(params=model.parameters(),lr=0.1)
    for iter in range(20):

        # NEW) iterate through each worker's dataset
        for data,target in datasets:

            # NEW) send model to correct worker
            model.send(data.location)

            # 1) erase previous gradients (if they exist)
            opt.zero_grad()

            # 2) make a prediction
            pred = model(data)

            # 3) calculate how much we missed
            loss = ((pred - target)**2).sum()

            # 4) figure out which weights caused us to miss
            loss.backward()

            # NEW) get model (with gradients)
            model.get()

            # 5) change those weights
            opt.step()

            # 6) print our progress
            print(loss.get().data[0]) # NEW) slight edit... need to call .get() on loss

train()
```

2. 我们将模型发送给每个worker，生成一个新的梯度，然后将梯度取回本地服务器，更新全局模型。我们从未在此过程中看到或请求访问training data。

- 缺点：当我们调用`model.get()`并从Bob或Alice接收更新的模型时，我们实际上可以通过查看它们的梯度来了解Bob和Alice的训练数据。在某些情况下，我们可以完美地恢复他们的训练数据。
- 一种解决方法：average the gradient across multiple individuals before uploading it to the central server，we want to average the gradients BEFORE calling `.get()`.That way, we won't ever see anyone's exact gradient

3. 改进：Federated Learning with Model Averaging

- 在最终生成的模型发送回模型所有者（我们）之前，允许权重由信任的“secure worker”聚合。通过这种方式，只有“secure worker”才能看到谁的权重来自谁。 我们只可以知道模型的哪些部分发生了变化，但不知道哪个worker（bob或alice）做出了哪些改变，从而创造了一层隐私。
- 具体步骤在part2.py

```
import syft as sy
import copy
hook = sy.TorchHook()
from torch import nn, optim

#### Step 1: Create Data Owners
```

```
#First, we're going to create two data owners (Bob and Alice) each
with a small amount of data.
#We're also going to initialize a secure machine called
"secure_worker".

# create a couple workers
bob = sy.VirtualWorker(id="bob")
alice = sy.VirtualWorker(id="alice")
secure_worker = sy.VirtualWorker(id="secure_worker")

bob.add_workers([alice, secure_worker])
alice.add_workers([bob, secure_worker])
secure_worker.add_workers([alice, bob])

# A Toy Dataset
data = sy.Var(sy.FloatTensor([[0,0],[0,1],[1,0],[1,1]]))
target = sy.Var(sy.FloatTensor([[0],[0],[1],[1]]))

# get pointers to training data on each worker by
# sending some training data to bob and alice
bobs_data = data[0:2].send(bob)
bobs_target = target[0:2].send(bob)

alices_data = data[2:].send(alice)
alices_target = target[2:].send(alice)

#### Step 2: Create Our Model
# Iniitalize A Toy Model
model = nn.Linear(2,1)

#### Step 3: Send a Copy of the Model to Alice and Bob
#we need to send a copy of the current model to Alice and Bob,they can
perform steps of learning on their own datasets.
bobs_model = model.copy().send(bob)
alices_model = model.copy().send(alice)

bobs_opt = optim.SGD(params=bobs_model.parameters(),lr=0.1)
alices_opt = optim.SGD(params=alices_model.parameters(),lr=0.1)

#### Step 4: Train Bob's and Alice's Models (in parallel)
#each data owner first trains their model for several iterations
locally before the models are averaged together.
for i in range(10):
    # Train Bob's Model
    bobs_opt.zero_grad()
    bobs_pred = bobs_model(bobs_data)
    bobs_loss = ((bobs_pred - bobs_target)**2).sum()
    bobs_loss.backward()
    bobs_opt.step()
    bobs_loss = bobs_loss.get().data[0]

    # Train Alice's Model
    alices_opt.zero_grad()
    alices_pred = alices_model(alices_data)
    alices_loss = ((alices_pred - alices_target)**2).sum()
    alices_loss.backward()

    alices_opt.step()
    alices_loss = alices_loss.get().data[0]
    alices_loss

#### Step 5: Send Both Updated Models to a Secure Worker
alices_model.move(secure_worker)
bobs_model.move(secure_worker)

##### Step 6: Average the Models
model.weight.data.set_(((alices_model.weight.data +
bobs_model.weight.data) / 2).get())
model.bias.data.set_(((alices_model.bias.data + bobs_model.bias.data)
/ 2).get())

#### And now we just need to iterate this multiple times!
iterations = 10
worker_iters = 5

for a_iter in range(iterations):
    bobs_model = model.copy().send(bob)
    alices_model = model.copy().send(alice)

    bobs_opt = optim.SGD(params=bobs_model.parameters(),lr=0.1)
    alices_opt = optim.SGD(params=alices_model.parameters(),lr=0.1)

    for wi in range(worker_iters):
        # Train Bob's Model
        bobs_opt.zero_grad()
        bobs_pred = bobs_model(bobs_data)
        bobs_loss = ((bobs_pred - bobs_target)**2).sum()
        bobs_loss.backward()

        bobs_opt.step()
        bobs_loss = bobs_loss.get().data[0]

        # Train Alice's Model
        alices_opt.zero_grad()
        alices_pred = alices_model(alices_data)
        alices_loss = ((alices_pred - alices_target)**2).sum()
        alices_loss.backward()
        alices_opt.step()
        alices_loss = alices_loss.get().data[0]

    alices_model.move(secure_worker)
    bobs_model.move(secure_worker)

    model.weight.data.set_(((alices_model.weight.data + bobs_model.weight.data) / 2).get())
    model.bias.data.set_(((alices_model.bias.data + bobs_model.bias.data) / 2).get())

    print("Bob:" + str(bobs_loss) + " Alice:" + str(alices_loss))

preds = model(data)
loss = ((preds - target) ** 2).sum()
print(preds)
print(target)
print(loss.data[0])
```

3. Encrypted Programs

1. 安全多方计算SMPC

- Instead of using a public/private key to encrypt a variable, each value is split into multiple "shares", each of which operate like a private key. Typically, these "shares" will be distributed amongst 2 or more "owners". Thus, in order to decrypt the variable, all owners must agree to allow the decryption. In essence, everyone has a private key.
- SMPC 协议的存在可以允许加密计算用于以下操作：1.加法 2.乘法 3.比较 4.使用上述基本的底层基元执行任意操作

2. SMPC Using PySyft

- 基本的加密/解密操作：

  Encryption is as simple as taking any PySyft tensor and calling `.share()`.

  Decryption is as simple as calling `.get()` on the shared variable

3. Build an Encrypted, Decentralized Database

   Encrypted（加密）：BOTH the values in the database will be encrypted AND all queries to the database will be encrypted.

   Decentralized（分散的）：using SMPC, all values will be "shared" amongst a variety of owners, meaning that all owners must agree to allow a query to be performed. It has no central "owner".

   架构：采用键值对的形式构建数据库，其中键和值都是string类型

- Constructing a Key System

SMPC采用数字进行操作，密钥类型为字符串，因此需要将字符串编码为数字。方法：将每个可能的键映射到唯一的hash散列（整数）

- Constructing a Value Storage System

  将value的每个字符串转换为一个数字列表（将每个char与int相对应）

- Creating the Tensor Based Key-Value Store

  可以通过加法、乘法和比较来对数据库进行查询，键为数字列表，值为整数数组列表

- 根据key值查询value

  1. 检查待查询键和数据库中已有键的相等性（查看是否存在）,每行返回1或0。 我们将每行的结果称为“key_match”。
  2. 将每一行的“key_match”整数乘以其相应行中的对应值，这将使数据库中没有匹配键的相应行归零。
  3. 将数据库中的所有行相加并返回结果

```
import string

char2int = {}
int2char = {}

# 将char和int相互编码
for i, c in enumerate(' ' + string.ascii_letters + '0123456789' +
string.punctuation):
    char2int[c] = i
    int2char[i] = c
# sy.mpc.securenn.field是我们默认使用SMPC编码的最大值
def string2key(input_str):
    return sy.LongTensor([(hash(input_str)+1234) %
int(sy.mpc.securenn.field)])
# encode each string as a list of numbers
def string2values(input_str):
    values = list()
    for char in input_str:
        values.append(char2int[char])
    return sy.LongTensor(values)

def values2string(input_values):
    s = ""
    for v in input_values:
        if(int(v) in int2char):
            s += int2char[int(v)]
        else:
            s += "."
    return s

class DecentralizedDB:

    def __init__(self, *owners):
        self.owners = owners #owners是分布式的数据库拥有者
        self.keys = list()
        self.values = list()

    def add_entry(self, string_key, string_value):
        key = string2key(string_key).share(*self.owners)#通过share操作加密数据库的键值对
        value = string2values(string_value).share(*self.owners)

        self.keys.append(key)
        self.values.append(value)

    def query(self, str_query):
        # hash the query string
        qhash = sy.LongTensor([string2key(str_query)])
        qhash = qhash.share(*self.owners)#通过share操作加密查询操作

        # see if our query matches any key
        key_match = list()
        for key in self.keys:
            key_match.append((key == qhash))

        # Multiply each row's value by its corresponding keymatch
        value_match = list()
        for i, value in enumerate(self.values):
            shape = list(value.get_shape())
            km = key_match[i]
            expanded_key = km.expand(1,shape[0])[0]
            value_match.append(expanded_key * value)

        # sum the values together
        final_value = value_match[0]
        for v in value_match[1:]:
            final_value = final_value + v

        result = values2string(final_value.get())#通过get操作取回数据

        # there is a certain element of randomness
        # which can cause the database to return empty
        # so if this happens, just try again

        if(list(set(result))[0] == '.'):
            return self.query(str_query)

        # Decypher final value
        return result
```

- 提高Performance的方法
    1. 对keys进行one-hot编码：比较（如==）的计算成本非常高，这使得查询需要很长时间。因此可以使用one-hot对字符串进行编码，于是可以使用乘法对数据库进行查询
    2. 固定values的长度值：通过固定values的长度，我们可以将整个数据库编码为单个tensor，可以使得底层的硬件更快的工作

4. Build an Encrypted, Decentralized Ledger

> 设计一个简单的原型（一个非常高性能的）数据库，你的帐户在其中有多个银行，但没有一个可以看到你的余额，交易只有在所有银行一起合作时才会发生

- Step 1: 先创建 A Simple Ledger

```
class SimpleLedger():

    def __init__(self, num_accounts=10, starting_balance=100):
        #创建一个初始化列表，每次交易都会更改这些值
        self.transactions = list()
        self.num_accounts = num_accounts

        initial_balances = sy.ones(num_accounts).long() *
starting_balance
        self.transactions.append(initial_balances)

    def transact(self, amt=10,frm=0,to=1):
        record = sy.zeros(self.num_accounts).long()
        record[frm] = -amt
        record[to] = amt
        self.transactions.append(record)
    #通过将全部的交易值相加得到余额
    def compute_balances(self):
        balance = self.transactions[0]
        for t in self.transactions[1:]:
            balance = balance + t

        return balance
```

- Step2: Checking Balance Values

  如果没有可用的余额，我们需要拒绝交易，需要做以下两件事

  1）维护一个运行的余额列表

  2）拒绝使得余额变为负数的交易

- Step 3: Encrypting / Decentralizing the Ledger

  we need to do is SMPC share all of the tensors involved.

```
class DecentralizedLedger():
    def __init__(self, *owners, num_accounts=10, starting_balance=100):

        self.transactions = list()
        self.owners = owners
        self.num_accounts = num_accounts

        initial_balances = sy.ones(num_accounts).long() * starting_balance
        initial_balances = initial_balances.share(*self.owners)
        self.transactions.append(initial_balances)

        # cache zeros for use in comparison
        self.zeros = initial_balances - initial_balances
        #维护一个运行的余额列表
        self.running_balance = self.compute_balances()

    def transact(self, amt=10,frm=0,to=1):
        record = sy.zeros(self.num_accounts).long()
        record[frm] = -amt
        record[to] = amt

        record = record.share(*self.owners)
        #每次交易前检查如果发生了这笔交易余额是否会为负数，是的话不发生交易
        candidate_balance = self.running_balance + record
        neg_check = 1 - (candidate_balance < self.zeros).sum(0).expand(self.num_accounts)

        record = record * neg_check

        self.transactions.append(record)
        self.running_balance = self.running_balance + record
    #将初始值和所有交易相加，即可得到用户余额
    def compute_balances(self):
        balance = self.transactions[0]
        for t in self.transactions[1:]:
            balance = balance + t

        return balance
```

Federated Learning with Encrypted Gradient Aggregation

> we will show how one can use SMPC to perform aggregation such that we don't need a "trusted aggregator".

详见/examples/tutorials/Part 8

Train an Encrypted NN on Encrypted Data

> we're going to use all the techniques we've learned thus far to perform neural network training (and prediction) while both the model and the data are encrypted.
>
> Note that Autograd is not yet supported for encrypted variables, thus we'll have to roll our own gradients ourselves. This functionality will be added in the next PySyft version.

1. Step 1: Create Workers and Toy Data

```
import syft as sy
hook = sy.TorchHook(verbose=True)

me = hook.local_worker
me.is_client_worker = False

bob = sy.VirtualWorker(id="bob", hook=hook, is_client_worker=False)
alice = sy.VirtualWorker(id="alice", hook=hook,
is_client_worker=False)

# create our dataset
data = sy.FloatTensor([[0,0],[0,1],[1,0],[1,1]])
target = sy.FloatTensor([[0],[0],[1],[1]])

model = sy.zeros(2,1)
```

2. Step 2: Encrypt the Model and Data

   1）由于安全多方计算仅适用于Longs类型，为了对浮点数类型进行操作（比如权重值等），需要使用Fixed Precision对所有数字进行编码，通过调用 `.fix_precision()` 来做到这一点。

   2）调用 `.share()` 通过在Alice和Bob之间分享数据来加密数据

```
data = data.fix_precision().share(alice, bob)
target = target.fix_precision().share(alice, bob)
model = model.fix_precision().share(alice, bob)
```

3. Step 3: Train

```
for i in range(10):
    pred = data.mm(model)
    grad = pred - target

    update = data.transpose(0,1).mm(grad)

    model = model - update * 0.1
    loss = grad.get().decode().abs().sum()
    print(loss)
```