Chapter 5
PySyft: A Library for Easy Federated Learning

Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, Théo Ryffel, Zarreen Naowal Reza, and Georgios Kaissis

Abstract PySyft is an open-source multi-language library enabling secure and private machine learning by wrapping and extending popular deep learning frameworks such as PyTorch in a transparent, lightweight, and user-friendly manner. Its aim is

We thank the OpenMined community and contributors for their work making PySyft possible. For more information about OpenMined, find us on GitHub or slack. https://www.openmined.org/.

A. Ziller
Technical University of Munich, Munich, Germany

A. Trask · E. Bluemke
University of Oxford, Oxford, UK

A. Lopardo
ETH Zurich, Zurich, Switzerland

A. Ziller · A. Trask · A. Lopardo · B. Szymkow · B. Wagner · E. Bluemke · J.-M. Nounahon · J. Passerat-Palmbach · K. Prakash · N. Rose · T. Ryffel · Z. N. Reza · G. Kaissis
OpenMined, Oxford, UK

J.-M. Nounahon
De Vinci Research Centre, Paris, France

J. Passerat-Palmbach
Imperial College London, Consensys Health, London, UK

K. Prakash
IIIT Hyderabad, Hyderabad, India

T. Ryffel
INRIA, ENS, PSL University Paris, Paris, France

Z. N. Reza
Thales Canada Inc., Quebec, Canada

G. Kaissis (✉)
Technical University of Munich, Imperial College London, Munich, Germany
e-mail: g.kaissis@tum.de

M. H. ur. Rehman and M. M. Gaber (eds.), *Federated Learning Systems*,
Studies in Computational Intelligence 965,
https://doi.org/10.1007/978-3-030-70604-3_5

to both help popularize privacy-preserving techniques in machine learning by making them as accessible as possible via Python bindings and common tools familiar to researchers and data scientists, as well as to be extensible such that new Federated Learning (FL), Multi-Party Computation, or Differential Privacy methods can be flexibly and simply implemented and integrated. This chapter will introduce the methods available within the PySyft library and describe their implementations. We will then provide a proof-of-concept demonstration of a FL workflow using an example of how to train a convolutional neural network. Next, we review the use of PySyft in academic literature to date and discuss future use-cases and development plans. Most importantly, we introduce Duet: our tool for easier FL for scientists and data owners.

Keywords Privacy · Federated learning · Differential privacy · Multi-party computation

5.1 Introduction to PySyft

Modern machine learning requires large datasets to achieve state-of-the-art performance. Furthermore, validating that the trained models are fair, unbiased, and robust often require even more data. However, many of the most useful datasets include confidential or private data that needs to be protected for regulatory, contractual, or ethical reasons. Additionally, much of this data is generated and stored in a decentralized fashion, for example on edge computing hardware, such as mobile phones or wearable health-tracking devices.

Research using large, private, decentralized datasets requires novel technical solutions so that this data can be used securely. These technical solutions must enable training on data that is neither locally available nor directly accessible without leaking that data.

Decentralized computing techniques collectively referred to as Federated Learning (FL) allow training on non-local data. In FL, training is performed at the location where the data resides, and only the machine learning (ML) algorithm (or updates to it) are being transferred. Secure machine learning, on the other hand, represents a collection of techniques that allow ML models to be trained without direct access to the data while protecting the models themselves from theft, manipulation, or misuse. Some examples of secure computation are encryption techniques such as homomorphic encryption (HE), by performing computation on fragments of data distributed over a computational network (secure multi-party computation, SMPC) Differentially Private (DP) machine learning aims to prevent trained models from inadvertently storing personally identifiable information about the dataset in the model itself. One common way of preventing this memorization is by perturbing the dataset (for example by noise injection) in a way that allows statistical reasoning about the entire dataset while protecting the identity of any single individual contained in it.

For secure and private ML to attain widespread acceptance, the availability of well-maintained, high quality, easily deployed open-source code libraries incorporating state-of-the-art techniques of the above-mentioned fields is essential. OpenMined, an open-source community, built the PySyft library to make these techniques as accessible and easy to implement as possible.

5.1.1 The PySyft Library

PySyft is an open-source, multi-language library that enables secure and private machine learning. PySyft aims to popularize privacy-preserving techniques in machine learning by making them as accessible as possible via Python bindings and an interface reminiscent of common machine learning which is familiar to researchers and data scientists. To ease future development in the field of privacy-preserving machine learning, PySyft aims to be extensible such that new FL, Multi-Party Computation, or Differential Privacy methods can be flexibly and simply implemented and integrated.

This chapter will introduce the methods available within the PySyft library and describe their implementations. We will then provide a proof-of-concept demonstration of a FL workflow using an example of how to train a convolutional neural network. Next, we review the use of PySyft in academic literature to date and discuss future use-cases and development plans. Most importantly, we introduce Duet: our tool for easier FL for scientists and data owners. Duet is the way to use PySyft for data science in a seamless, intuitive way.

5.1.2 Privacy and FL

FL is a collection of computational techniques allowing the distributed training of algorithms on remote datasets. It relies on distributing copies of the learning algorithm to where the data is instead of centrally collecting the data itself. For example, in a healthcare setting, patient data can remain on the hospital's servers thus retaining data ownership while still allowing the training of algorithms on the data. FL can be set up in several ways, for example, a common way is having decentralized nodes for training which send their updates back to a central server for aggregation.

FL itself is not always a sufficient privacy mechanism. Deep learning models tend to unintentionally memorize aspects of their training data, i.e. storing dataset attributes in their parameters [11]. This unintentional memorization makes FL settings without additional privacy-preserving techniques sensitive to various attacks, such as attribute or membership inference attacks. In addition to the various attack vectors, FL typically has substantial network transfer requirements as well as usually reduced algorithm performance compared to centralized training. Techniques for efficient training and for reducing network transfer requirements are being actively

researched [23]. These concerns aside, FL is an important component of many privacy-preserving machine learning systems.

5.1.3 Differential Privacy

Differential privacy is a randomized system for publicly sharing meaningful information about a dataset by describing the patterns within the dataset while withholding information about individuals in the dataset. It is a process of perturbing the data with minimal noise to preserve the privacy of the users while minimizing the loss in the utility of the data and the computational complexity of operating on it. Uncertainty in the process means uncertainty for the attacker, which means better privacy.

Consider the example of a simple private database that stores the names and ages of the citizens of a small town. This dataset could be useful for various applications and help better the lives of the citizens of that town, but it is important that the database remains private and does not get into the wrong hands. The private data could be misused!

If we wanted to obtain statistics on the database publicly, such as the average, minimum, and maximum age of the citizens, we need to be careful. If we report the true average, minimum, or maximum, we might compromise the privacy of some specific citizens. So, we report a slightly noisy average, minimum, and maximum publicly, so that no one single citizen's privacy is breached. This enables all citizens a certain degree of plausible deniability as to whether they were a part of the database.

ϵ-Differential Privacy provides us with a universal privacy guarantee—given a small privacy budget ϵ, we can ensure that the output of the process does not change by more than e^{ϵ} by the change or removal of a single person from the database. Differential privacy is more robust than naive anonymization in that it quantifies the risk of data leakage and provides theoretical privacy guarantees within its scope of application.

The amount of noise that needs to be added to the query result depends on:

- Sensitivity of the query
- Distribution of the noise source
- Privacy budget.

5.1.3.1 Differentially Private Machine Learning

In the context of data-driven approaches like machine learning, Differential Privacy is very useful, as their objectives align. Both focus on recognizing general meaningful patterns instead of individual user data. However, it is quite challenging to generalize various DP techniques to the broad set of techniques in Machine Learning. Consider an artificial neural network—it is a complex function involving a large number of parameters and inputs. Finding the query sensitivity of this neural network is quite

tricky, as it involves a lot of non-linear computations. This acts as a barrier to applying Differential Privacy more generally to machine learning.

However, despite the challenges, there have been some interesting and useful research work in this area. Various works focusing on Differential Privacy applied to learning algorithms consider the following key factors:

- The scale of the noise added relative to the data (norm).
- The right stage in the learning process for adding noise. This often is strongly related to the model of computation and data flow. In the case of data coming in from multiple (distributed) sources, we might consider adding noise early on.
- Pre-processing to have an upper bound on the sensitivity of the data.
- Adversarial attacks are specific to learning methods, such as model inversion attack, linkage attack, and data reconstruction attacks. We are interested to see how the behavior changes when Differential Privacy is used.
- The utility-privacy-computation speed trade-off in the learning process. Since Differential Privacy provides us a quantitative method to measure privacy, it helps in a strong analysis of the decrease in utility, which we achieve at the cost of data privacy.
- The effect of Differential Privacy on the learning model's ability to generalize. It has been a surprising result that adding noise to ensure privacy improves the utility of the learning model, as it can generalize better on larger unseen data.
- A similar effect has been observed in improving the stability of the learning algorithm.

5.1.3.2 Differential Privacy in PySyft

While the use of FL enables us to perform distributed computation by enabling access to remote data, it does not guarantee the privacy of users, as inferences about the users can be made from the trained model, thus risking the exposure of sensitive information about the users. PySyft creates and uses automatic Differential Privacy to provide the data users with strong privacy guarantees, independent of the Machine Learning architecture and the data itself.

PySyft can achieve this is by adding automatic query sensitivity tracking and privacy budgeting to the private tensor. The tensor is made up of a huge matrix of private scalar values, which are all bounded. This way, we can dynamically track the sensitivity of the query function as well as the amount of privacy budget we have spent. This helps us in employing the various techniques on making Machine Learning automatically differentially private with ease. Thus, PySyft is general enough to support any kind of deep learning architecture, as the key components of Differential Privacy are a part of PySyft's building blocks.

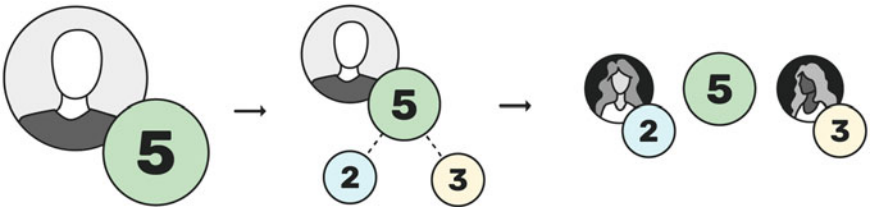

Fig. 5.1 Illustration of splitting a single integer in multiple shares, which can then be held by other parties

5.1.4 Secure Multi-party Computation

FL and differential privacy are not sufficient to prevent attacks against machine learning models or datasets by the participants in the training. For example, it is very easy for a data owner to steal a model in classic FL as they are sent the model in plain text. Methods [34] need to be developed to safe-keep both the data and the algorithms, while still permitting training and inference. Secure multi-party computation (SMPC) provides a framework allowing multiple parties to jointly perform computations over a set of inputs and to receive the resulting outputs without exposing any party's sensitive input. It thus allows models to be trained or applied to data without disclosing the training data items or the model's weights. SMPC relies on splitting data into 'shares', as seen in Fig. 5.1, which, when summed, yield the original value. Evaluating or training a model can be decomposed in basic computations for which SMPC protocols exist, which allows for end-to-end secure procedures. In the case of ML, during training, the model's gradients can be shared, while in inference, the entire model function can be shared.

SMPC incurs a significant communication overhead but has the advantage that unless a majority of the parties are malicious and coordinate to reveal an input, the data will remain private even if sought after for unlimited time and resources. SMPC can protect both models' parameters and training/inference data.

One of the conceptually simplest implementations of secret sharing in SMPC is 'additive secret sharing'. In this paradigm, a number, for example, $x = 5$, can be split into several shares (in this example two shares), $share_1 = 2$ and $share_2 = 3$, managed independently by two participants. At this point, the application of any number of additional operations on these shares individually and the sum of the results would be the same as applying the same additions individually on $x = 5$. In practice, the shares are often taken randomly in a large finite field, which implies that having access to one share doesn't reveal anything about the secret value.

```
from random import randint # import library to
    generate random integers

Q = 12163945178128104340259 3 # a large prime number

def encrypt(x, n_shares = 2):
```

```
    shares = list()
    for i in range(n_shares-1):
        shares.append(randint(0,Q)) # randint will return any random number between 0 to Q
    final_share = Q - (sum(shares) % Q) + x
    shares.append(final_share)
    return tuple(shares)

def decrypt(shares):
    return sum(shares) % Q

def add(a, b):
  assert(len(a) == len(b)) # check if length of variables is equal - if condition is False, AssertionError is raised
    c = list()
    for i in range(len(a)):
        c.append((a[i] + b[i]) % Q)
    return tuple(c)
```

For multiplication between encrypted numbers, PySyft implements the SPDZ protocol (pronounced "Speedz") that is an extension of additive secret sharing; encrypt, decrypt and add are the same, but it enables more complex operations than addition.

Multiplication in SPDZ uses externally generated triples of numbers to maintain privacy during the computation. Within the examples below we use a crypto provider that is not otherwise involved in the computation to generate these triples but they could also be generated with HE.

```
def generate_mul_triple():
    a = random.randrange(Q)
    b = random.randrange(Q)
    a_mul_b = (a * b) % Q
    return encrypt(a), encrypt(b), encrypt(a_mul_b)

# we also assume that the crypto provider distributes the shares

def mul(x, y):
    a, b, a_mul_b = generate_mul_triple()

    alpha = decrypt(x - a) # x remains hidden because a is random
    beta  = decrypt(y - b) # y remains hidden because b is random

    # local re-combination
    return alpha.mul(beta) + alpha.mul(b) + a.mul(beta) + a_mul_b
```

Taking a closer look at the operations we can see that since $alpha * beta == xy - xb - ay + ab, b * alpha == bx - ab,$ and $a * beta == ay - ab$, if we add them all together and then sum a*b we will effectively return a privately shared version of xy.

This schema appears quite simple, but it already permits all operations, as combinations of additions and multiplications, between two secretly shared numbers. Similar to the more complex homomorphic encryption schemes that work with a single party, SPDZ allows computation on ciphertexts generating an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the plaintext.

In this case, splitting the data into shares is the encryption, adding the shares back together is the decryption, while the shares are the ciphertext on which to operate.

This technique is adequate for integers, covering the encryption of things like the values of the pixels in images or the counts of entries in a database. The parameters of many machine learning models like neural networks, however, are floats, so how can we use additive secret sharing in machine learning? We need to introduce a new ingredient, Fixed Precision Encoding, an intuitive technique that enables computation to be performed on floats encoded in integers values. In base 10, the encoding is as simple as removing the decimal point while keeping as many decimal places as indicated by the precision. In PySyft, you can take any floating-point number and turn it into fixed precision by calling $my\_tensor.fix\_precision()$.

```
BASE=10
PRECISION=4

def encode(x):
    return int((x * (BASE ** PRECISION)) % Q)


def decode(x):
    return (x if x <= Q/2 else x - Q) / BASE**
    PRECISION

encode(3.5) <-- 35000
decode(35000) <-- 3.5
```

5.1.5 Homomorphic Encryption

Input data privacy is among the most important topics in secure and private ML. One solution to the issue presented above is Secure Multi-Party Computation. However, if secret sharing is not a viable option due to a limited number of participants or very high communication overhead, homomorphic encryption (HE) can provide a framework for computations on encrypted data. In brief, HE ensures that performing operations on encrypted data and decryption of the result is equivalent to performing the analogous operations without any encryption. Therefore, HE can be used to achieve input data privacy, however only requires a single party to encrypt and decrypt the data (Fig. 5.2).

For a HE scheme to be able to support arbitrary computations on a hidden ciphertext (thus being Fully Homomorphic), it must be able to support unbounded

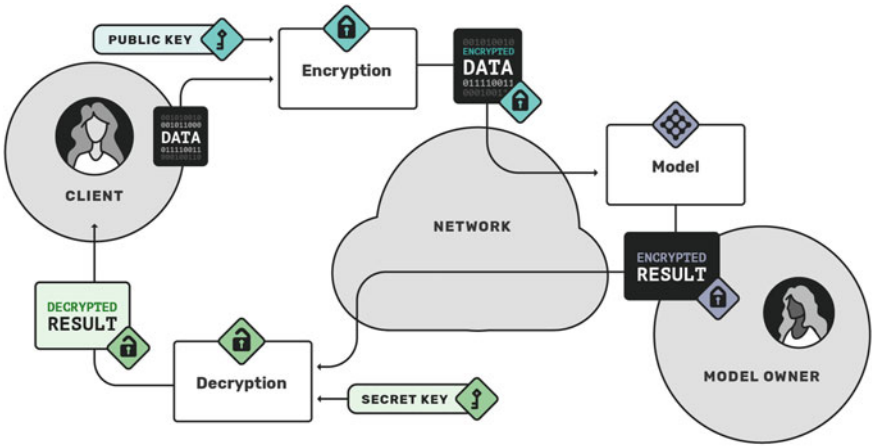

Fig. 5.2 Machine learning as a service using HE

sequences of additions and multiplications. However, a significant challenge inherent to HE, not present in SMPC, is supporting functions that, to be computed on ciphertext, require long sequences of base operations. This issue is a direct consequence of how security is ensured in schemes of this kind. In FHE, it's common to rely on complex computational problems that are conjectured to be hard to solve, to secure the plaintext. A common characteristic of these problems is the use of noise or error to make simple tasks, like calculating the GCD of a set of integers or the eigenvalues of a matrix, computationally intractable in their approximate form. Yet, the error that proves so effective at securing these schemes, after many operations on the same ciphertext without decryption, can grow too large and corrupt the plaintext.

So is there a way to decrease the error? Yes, bootstrapping, a technique that involves running the decryption procedure homomorphically using encryption of the secret key, without revealing the message. Homomorphic decryption eliminates the error produced by previous operations but injects an error of its own just like any other function so the plaintext stays secure. Nevertheless, bootstrapping and its variants are quite computationally intensive and so refreshing the ciphertext, i.e, reducing the error, remains the primary issue in FHE.

Pysyft supports the CKKS fully homomorphic encryption scheme and the Paillier scheme which is limited to addition but is much faster. Here we show how to use CKKS which is integrated into PySyft with a python wrapper around Microsoft's C++ TenSEAL library (Fig. 5.3).

```
import syft as sy
import torch as th
import syft.frameworks.tenseal as ts

torch tensors
hook = sy.TorchHook(th)

# public and private key generation
```

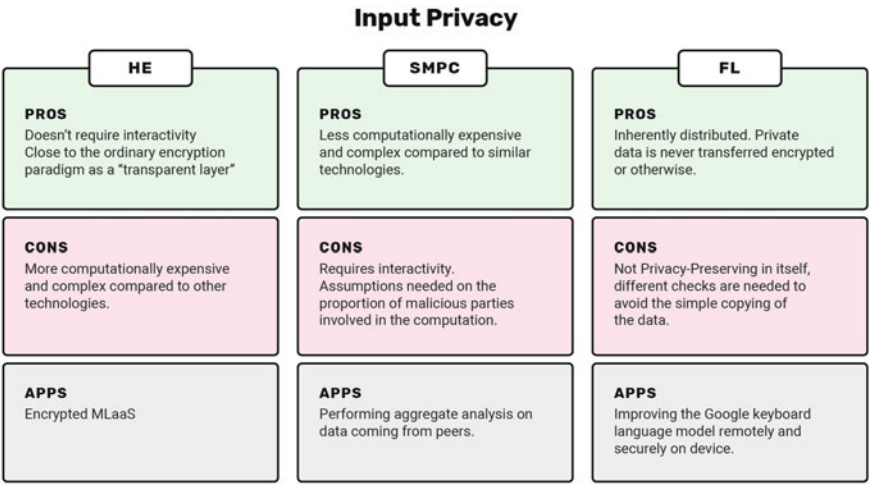

Fig. 5.3 Comparative view of pros, cons and sample application of Homomorphic Encryption, Secure Multi-Party Computation and FL

```
pub, pri = ts.generate_ckks_keys()

# encrypting with CKKS
x = th.tensor([1.2, 2.2, 3.2]).encrypt("ckks", public_key = pub)
y = th.tensor([2.0, 2.0, 2.0]).encrypt("ckks", public_key = pub)

# encrypted division and multiplication
div = x / y
mul = x * y

# decryption with private key
print(div.decrypt(pri))
print(mul.decrypt(pri))
```

```
tensor([0.6, 1.1, 1.6])
tensor([2.4, 4.4, 6.4])
```

5.2 PySyft Implementation

Since not all machine learning practitioners hold expertise in privacy and cryptography techniques, PySyft was built to make them easily available via the most popular machine learning tools that researchers and data scientists work with daily.

The central component in PySyft is the *Tensor*. Tensors usually follow standard APIs like the PyTorch or TensorFlow APIs, but also provide extra functionalities like the ability to send a tensor to a remote worker or to encrypt it. This is done

using *chains of tensors* where each tensor in the chain adds a specific capability to the chain, like the conversion of float values to fixed precision. This modularity is the key to implement FL, differential privacy, homomorphic encryption, or secure multi-party computation protocols within the same framework. Another important concept is the concept of *Worker*: workers own tensors and can perform computation on them. They can communicate using *Messages* with other workers to send or get tensors, or to perform remote computations. Workers are composed of a client and a server: an interaction with a remote worker is always made through its client which defines the set of possible actions. Messages are serialized using Protobuf (Protocol Buffers) which allows for cross-platform distributed execution: a worker in Python can communicate with a worker in JavaScript, Swift, Kotlin, etc. All remote computations are made through *Pointers*. The most popular pointer is the *PointerTensor* but there exist other types of pointers like the PointerDataset.

5.2.1 A Standardized Framework to Abstract Operations on Tensors

5.2.1.1 The Chain Structure

Performing transformations or sending tensors to other workers can be represented as a chain of operations, and each operation is embodied by a special class. To achieve this, we created an abstraction called the AbstractTensor. It gives default behaviors to Syft tensors and implements actions that are independent of the underlying framework (PyTorch, Tensorflow, …). All tensors living under the AbstractTensor are meant to represent a state or transformation of the data and can be chained together. The chain structure allows the transformations or states embodied by the different Syft tensors to be accessed downward using the `child` attribute and upward using the `parent` attribute.

Figure 5.4 presents the general structure of a tensor chain, where SyftTensors are a generic term for any concrete instance of AbstractTensor. All operations are first applied to a Wrapper tensor at the top of the chain which makes it possible to have the native Torch interface, and they are then transmitted through the chain by being forwarded to the child attribute.

Wrapper tensor is one of the two important subclasses of Tensor types. It wraps the new Syft tensor types with native type so that the new tensor is compatible with the rest of the PyTorch/TensorFlow API. This construction primarily addresses the lack of support in PyTorch and TensorFlow for creating arbitrary Tensor types (via subclassing torch. Tensor for instance in PyTorch). Ensuring this compatibility is critical for the new tensors to be usable with the existing ecosystem of layers and loss functions shipping with the native frameworks. The wrapper tensor is an empty torch tensor with a child argument which is a PointerTensor.

PointerTensors are the second important class deriving from AbstractTensor. They mimic the entire API of a normal tensor, but instead of computing a tensor function locally (such as addition, subtraction, etc.) they forward the computation to another PySyft worker.

As such, PointerTensors should never exist by themselves: their role is to proxy API calls to an actual tensor located on a different worker (virtual or remote) that it points to. From a data communication point of view, PointerTensors fully rely on the corresponding worker owning the target tensor to communicate the API commands it receives down the chain of tensors as shown in Fig. 5.5.

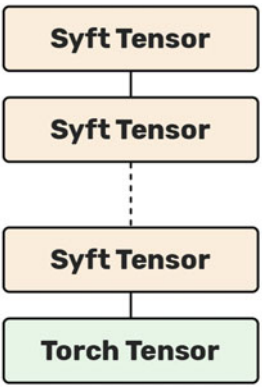

Fig. 5.4 General structure of a Tensor chain. SyftTensors are a generic term for any concrete instance of AbstractTensor

```
x = th.tensor([[1,2],
[3,-4]])
x = syft.LoggingTensor().
on(x)
x = x.send(bob)
x.abs_()
x = x.get()

```

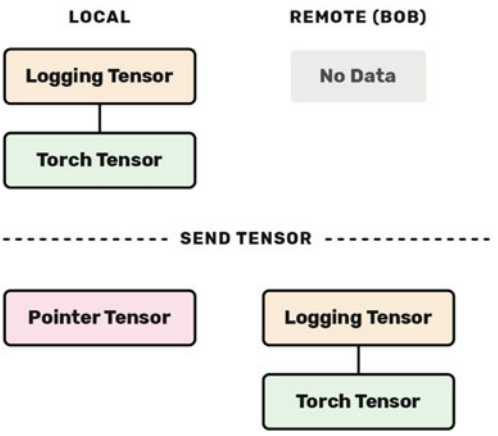

Fig. 5.5 Impact of sending a tensor on the local and remote chains. Local worker adds a `LoggingTensor` before sending the chain to `bob`. The call to `x.abs_()` is done remotely on bob's chain before the Local worker retrieves the result by calling `x.get()`

5.2.1.2 From Virtual to Real Context Execution of FL

To simplify debugging complex chains of operations, PySyft develops the notion of Virtual Workers. Virtual Workers all live on the same machine and do not communicate over the network. They simply replicate the chain of commands, serialization operations, and expose the very same interface as the actual workers to communicate with each other.

Network-enabled workers in the context of FL have two implementations in the framework as of now: WebSocket workers and GridNodes. They both leverage WebSockets as their communication medium, thus ensuring a broad range of devices from IoT to web browsers to servers can participate in a PySyft network.

WebSocket workers allow multiple workers to be instantiated from within a browser, each within its tab. This gives us another level of granularity when building FL applications before actually addressing remote workers which are not on the same machine. Web Socket workers are also a very good fit for the data science ecosystem revolving around browser-based notebooks.

The PyGrid project, also developed under the OpenMined umbrella, introduces its Worker known as GridNode. GridNodes communicate with their peers and clients over web sockets, but contrary to Web Socket workers, they expose a traditional web socket server interface that then passes input commands and messages down to an embedded VirtualWorker. GridNodes can be assembled to jointly compute under a Grid Gateway to provide horizontal scalability to compute larger workloads.

```
# on your terminal, launch the Grid nodes
cd PyGrid/apps/node
./run.sh --id alice --port 7600 --host localhost --
start_local_db
./run.sh --id bob --port 7601 --host localhost --
start_local_db
```

```
# on your notebook, connect to them and build a Grid Network
import syft as sy
from syft.grid.clients.data_centric_fl_client import
DataCentricFLClient

hook = sy.TorchHook(th)
alice = DataCentricFLClient(hook, "ws://localhost:7600")
bob = DataCentricFLClient(hook, "ws://localhost:7601")

my_grid = sy.PrivateGridNetwork(alice, bob)
```

Fig. 5.6 Code example of a Grid Network configuration with two Grid Nodes

```
x = th.tensor([1.2,
-3.4])
x = x.fix_precision()
x = x.share(alice, bob)
x = x * 2
x = x.get().
float_precision()
```

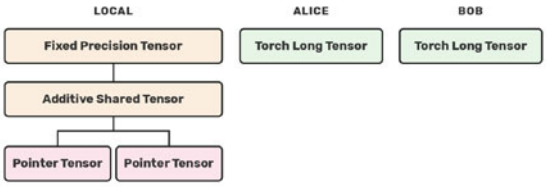

Fig. 5.7 Chain structures of an additively shared tensor. Local worker shares its tensor with `alice` and `bob`. The call to `x = x * 2` is done remotely on alice's and bob's chain before the Local worker retrieves the result by calling `x.get().float_precision()`

5.2.2 Building MPC-Aware Tensors

The elements introduced in Sect. 5.2.1.1 form the building bricks necessary to create more advanced privacy-preserving tensors. The SMPC protocol detailed in Sect. 5.1.4 involves splitting and sending shares of an original tensor. These elements can be managed using a list of PointerTensors as described in Fig. 5.6. This section will describe the MPC toolbox proposed in our PySyft. It provides convenient abstractions over both the SPDZ [8, 9] and SecureNN [41] protocols (Fig. 5.7).

The implementation of the SPDZ protocol in the MPC toolbox includes basic operations such as addition and multiplication but also preprocessing tools to generate for instance Beaver triples necessary for multiplying to private tensors, and more specific operations to neural networks including matrix multiplication. Since the SPDZ protocol assumes that the data is given as integers, we added into the chain a node called the FixedPrecisionTensor that converts float numbers into fixed precision numbers. This node encodes the value into an integer and stores the position of the radix point.

A toolbox relying solely on SPDZ would require some adjustments to be made to the traditional elements of a convolutional network due to the specificities of the protocol. As described in [9], SPDZ-based implementations use average pooling instead of max pooling and approximate higher-degree sigmoid instead of ReLU as an activation function.

This was addressed by implementing the second protocol, SecureNN, in the MPC toolbox. SecureNN enables computing layers like ReLU and MaxPool in the secret-sharing setting. The version available in PySyft was modified to handle N parties instead of the original 3 in the canonical version.

Unlike the MPC protocol proposed by [9], players are not equal in our framework since one is the owner of the model (called the local worker). He acts as a leader by controlling the training procedure on all the other players (the remote workers). To mitigate this centralization bias when dealing with data, the local worker can create remote shared tensors on data he doesn't own and can't see.

Indeed, we expect remote workers to hold some data of their own in a general setting, for instance when hospitals are contributing medical images to train a model.

Multiple players are then interested in seeing the execution performing correctly, which is particularly crucial during the inference phase where many factors could lead to corrupted predictions [14].

So far, the current implementation does not come with a mechanism to ensure that every player behaves honestly. An interesting improvement would be to implement MAC authentication of the secret shared value, as suggested by [9].

5.2.3 Actions, Plans and Protocols

All operations in PySyft are called actions. Actions are categorized into two main classes. On the one hand, computation actions are actions that act on tensors and produce new tensors. For example, framework-specific commands like $torch.add(x, y)$ is a computation action, but syft specific command like $x.fix\_precision()$ is also one. On the other hand, communication actions are actions that involve moving tensors across workers. The command $pointer = x.send(alice)$ is typically a communication action and $pointer.get()$ as well. When a local worker wants to execute computation actions but on remote tensors, it will send a specific message to have the action computed remotely. If it wants to run n actions it will therefore need to send n messages, which can be burdensome. Therefore, PySyft provides an object called a Plan which allows batching together several computation actions. This Plan can be sent to any worker with a single message and be executed. One thing special about Plans is that they need to be built i.e. they need to be run once on dummy data to select and store the computation actions they will contain. On our OpenMined GitHub, you can find PySyft tutorials that give more details about Plans and how to use them. However, Plans fall short when someone wants to include communication actions to ship them to other workers. That's why we introduced Protocols.

Protocols contain an arbitrary set of computation and communication actions that should involve a fixed number of workers, modeled by Roles. Like Plans, Protocols need to be built and then are deployed across workers which all receive the actions in which they are involved. This Protocol abstraction allows designing complex, distributed, and asynchronous computation graphs which are extremely suited for FL and secure multi-party computation.

For more information, please see our documentation which can be found at our OpenMined GitHub (github.com/OpenMined/). You can find all of our educational tools at https://github.com/OpenMined/OM-Welcome-Package/blob/master/Educational-Tools.md, which has links to introductory videos, the Udacity course, how to join Slack, the mentorship program, OM bootcamps, links to Good First Issues and Tutorials.

5.3 The Road-Map for PySyft

5.3.1 Communications Across Nodes and Heterogeneous Languages

Python is an extremely popular language for many of the tasks associated with privacy-preserving machine learning, but it does not provide sufficient coverage to support the entire community or the planned use cases detailed below.

FL scenarios require learning to be performed across a variety of contexts including mobile phones running the iOS or Android operating systems, embedded devices, or web applications. Participants in the data science ecosystem have also been using platforms like R-Studio, Matlab, Sage, and languages like Julia, C++, and Clojure and should be included in OpenMined's plans for PySyft.

A standardized set of protocols will be developed using *protocol buffers*, Google's language-neutral and platform-neutral extensible mechanism for serializing structured data. This will enable researchers to collaborate regardless of their platform or language preference.

Specifically, the future Syft direction is to support *language-agnostic and secure remote-execution* of arbitrary code from supported frameworks and curated PPML capabilities.

5.3.2 Improved Secure Multi-party Computation Implementations

Planned improvements for the secure multi-party compute implementation include:

- Implementation of the FALCON framework which builds on and improves SecureNN
- Implementation of Function Secret Sharing, an SMPC protocol with fewer interactions
- Publicly auditable MAC authentication for SPDZ
- Improved protocol efficiency by adding optional support for the *Unconditional MPC Protocol* suggested by [18].

5.4 Introducing 'Duet': Easier FL for Scientists and Data Owners

OpenMined seeks to unlock the power of the broader scientific community by making it easy for a scientist to perform statistics, make inferences, train models, and evaluate model efficacy on data that they do not own or control. We call our first open-source

product in this sphere Duet. A Duet can be initiated between two individuals, data scientist and data owner. This is a way to use PySyft for data science in a seamless, intuitive way.

After connecting on a shared video call, a data owner starts a local Jupyter notebook, creates a "Duet" server in the notebook, and loads their data into it. This server returns a unique URL which is passed to the Data Scientist who, in their Jupyter notebook, can initiate a connection to the data owner's Duet server. This connection enables the data scientist to execute operations at the tensor level on the data owner's data without the data leaving the data owner's machine. This will allow hospitals and medical researchers, financial institutions and fraud investigators, satellite imagery companies and governments, and many other "duet" pairs of data owners and data scientists to understand insights, build models, and predict outcomes on valuable data that remains private.

Duet is rapidly evolving and improving, so any demonstration in this chapter would become quickly outdated. To try it out, please visit OpenMined.org to find the demonstration, tutorials, and materials.

5.5 PySyft Demonstration—Federated Learning on MNIST Using a CNN

In this example, we will use the classic example [37] of training a CNN on MNIST using PyTorch to demonstrate how to implement Federated Learning using the PySyft library.

5.5.1 Imports and Model Specifications

First, import the Pytorch dependencies.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

In particular we define the remote workers **alice** and **bob**, which will hold the remote data while a local worker (or client) will orchestrate the learning task. Note that we use **virtual workers**: these workers behave exactly like normal remote workers except that they live in the same Python program.

Hence, we still serialize the commands to be exchanged between the workers but we don't really send them over the network. This way, we avoid all network issues and can focus on the core logic of the project.

```
import syft as sy
```

```
hook = sy.TorchHook(torch)  # hook PyTorch ie add extra
    functionalities to support Federated Learning
bob = sy.VirtualWorker(hook, id="bob")  # define remote worker bob
alice = sy.VirtualWorker(hook, id="alice")  # and alice
```

Next, define the settings of the learning task:

```
class Arguments():
    def __init__(self):
        self.batch_size = 64
        self.test_batch_size = 1000
        self.epochs = epochs
        self.lr = 0.01
        self.momentum = 0.5
        self.no_cuda = False
        self.seed = 1
        self.log_interval = 30
        self.save_model = False

args = Arguments()
use_cuda = not args.no_cuda and torch.cuda.is_available() # GPU if
     needed unless CPU
torch.manual_seed(args.seed)
device = torch.device("cuda" if use_cuda else "cpu")
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else
    {}
```

5.5.2 Data Loading and Sending to Workers

Consider the generic situation in which the data is loaded and transformed from a local training dataset into a federated dataset using the **.federate method**: the dataset is split in two parts and sent to the workers **alice** and **bob**. This federated dataset is now handed over to a **federated DataLoader** which will iterate over remote batches.

The test dataset remains unchanged as the local client will perform the test evaluation.

```
federated_train_loader = sy.FederatedDataLoader(
    datasets.MNIST(
        "../data", train=True, download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize
    ((0.1307,), (0.3081,))]
        ),
    ).federate((bob, alice)),  # we distribute the dataset
    across all the workers
    batch_size=args.batch_size,
    shuffle=True,
    **kwargs
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "../data", train=False,
```

```
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize
    ((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=args.test_batch_size,
    shuffle=True,
    **kwargs
)
```

5.5.3 Convolutional Neural Network Specification

Here, the same CNN as in the official example is used.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

5.5.4 Define the Training and Test Functions

For the training, as the data batches are distributed across **alice** and **bob**, the model must be sent to the right location for each batch. Then, we perform all the operations remotely with the same syntax like we are doing in a local PyTorch environment. When we are done, we get back the model updates and the loss to look for improvement using the **.get()** method.

```
def train(args, model, device, train_loader, optimizer, epoch
    ):
    model.train()
    for batch_idx, (data, target) in enumerate(
    federated_train_loader): # <-- now it is a distributed
    dataset
```

```
        model.send(data.location) # send the model to the right location
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        model.get() # get the model back
        if batch_idx % args.log_interval == 0:
            loss = loss.get() # <-- NEW: get the loss back
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * args.batch_size, len(train_loader) * args.batch_size, # batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

The test function does not change as it is run locally.

```
def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up batch loss
            pred = output.argmax(1, keepdim=True) # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

5.5.5 *Launch the Training*

The training process remain unchanged.

```
model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=args.lr)

for epoch in range(1, args.epochs + 1):
    train(args, model, device, federated_train_loader, optimizer, epoch)
```

```
    test(args, model, device, test_loader)

if (args.save_model):
    torch.save(model.state_dict(), "mnist_cnn.pt")
```

```
Train Epoch: 1 [0/60032 (0%)]   Loss: 2.305134
Train Epoch: 1 [640/60032 (1%)] Loss: 2.273475
Train Epoch: 1 [1280/60032 (2%)]    Loss: 2.216173
Train Epoch: 1 [1920/60032 (3%)]    Loss: 2.156802
Train Epoch: 1 [2560/60032 (4%)]    Loss: 2.139428
Train Epoch: 1 [3200/60032 (5%)]    Loss: 2.053060
...
Train Epoch: 10 [56960/60032 (95%)] Loss: 0.006612
Train Epoch: 10 [57600/60032 (96%)] Loss: 0.010964
Train Epoch: 10 [58240/60032 (97%)] Loss: 0.036587
Train Epoch: 10 [58880/60032 (98%)] Loss: 0.134881
Train Epoch: 10 [59520/60032 (99%)] Loss: 0.011405

Test set: Average loss: 0.0000, Accuracy: 9894/10000 (99%)
```

The CNN has been successfully trained on remote data using federated learning. This simple demonstration shows the ease of using PySyft for federated learning: only 10 lines are changed from the original demonstration and the compute overhead remains low.

5.6 PySyft Literature Review and Future Use Cases

As stated in FL—Synthesis Lectures on Artificial Intelligence and Machine Learning, PySyft is the first open-source FL framework for building secure and scalable ML models [49]. In recent review articles, PySyft has been described as a framework that has "made SMPC and FL intuitive and accessible to machine learning developer" [1], and allows data scientists to "focus purely on building up the machine learning model without being burdened with how it will be deployed in local data repositories" [20]. In two recent reviews, [22, 35], authors have mentioned the PySyft project as an influential open-source library for homomorphic encryption.

To date, PySyft has been discussed as a novel and convenient approach to FL in several comprehensive surveys of existing literature on FL frameworks in healthcare, mobile edge devices, and more [1, 3, 27, 28, 38, 43, 49]. In [33], PySyft was mentioned for its functionality of applying differential privacy methods to FL and preventing the extraction of sensitive information from trained models. In this section, we summarize the use of PySyft in different literature to date and conclude by discussing future use cases.

5.6.1 Comparisons with Other Frameworks

In 2019, Li et al. [27] noted the advantages of PySyft over other contemporary FL frameworks like TFF, PaddleFL, and others. The authors explain that while PaddleFL provides algorithm level APIs for users to use directly, PySyft provides more detailed building blocks so that the developers can easily implement their FL processes. It is also mentioned that PySyft provides more privacy mechanisms compared to TFF while covering all the listed features that TFF supports. Asad et al. [3] presented a detailed comparison between different existing FL frameworks such as PySyft, TFF, and LEAF based on their architecture, features, and communication-efficiency. On the contrary, in [4] Beutel et al. proposed an FL framework named Flower that supports heterogeneous client environments and mobile and edge device SDK along with other state-of-the-art features. Authors compared Flower with PySyft and other existing FL frameworks like Tensorflow and LEAF where Flower reportedly has an edge over others for providing these additional FL tools. On a similar note, authors in [48] developed a heterogeneity-aware FL platform that takes into consideration the challenges coming from diverse user behavior and different device hardware capacity. Their experiment showed that the heterogeneity-aware platform was able to provide more practical input configurations and output metrics compared to other existing frameworks including PySyft. Also, Boemer et al. [5] came up with MP2ML, a machine learning framework that integrates nGraph-HE (Homomorphic Encryption) and the secure two-party computation framework to overcome the limitations of leaking the intermediate feature maps to the data owner while model training. In a comparison of MP2ML to PySyft, they reported that in terms of model privacy PySyft hides model weights but not the activation functions and model architecture which is a key feature of MP2ML.

5.6.2 Use Case: Benchmarking and Standardizing FL Systems

Along with frameworks with cutting-edge tools, there are also new frameworks being introduced for benchmarking FL algorithms applied in different use-cases. As an example, in [19] Hu et al. presented a benchmark suite for federated machine learning systems equipped with diverse data partitioning methods, cutting edge applications in image, text, and structured data, and use-cases with various learning complexity. They also implemented a few available tools using existing FL frameworks as references to their benchmarking. They used PySyft for model training and implementing the encryption process for secure multi-party computation. On the other hand, He et al. in [16] introduced an open research library called FedML for the development and benchmarking of new FL algorithms in different system environments including distributed training, mobile on-device training, and standalone simulation. Authors reported that compared to PySyft, FedML provides additional features like topol-

ogy customization, algorithm implementation of decentralized FL, FedNAS [15] and vertical FL, and standardized benchmarking of Deep Neural Networks. In an attempt to standardize FL-based applications, Rodriguez-Barroso et al. in [39] proposed an open-source unified framework for FL and Differential Privacy including the standard methodology for adapting the machine learning paradigms and developing AI-based services like classification and regression using FL and differential privacy. While analyzing different existing frameworks, they mentioned PySyft as a low-level FL framework intended to be used by advanced developers with the lack of beginner-friendly support and mechanism and algorithms for Differential Privacy. In [40], Ryffle et al. proposed a low-interaction framework for private training and inference of standard deep neural networks on sensitive data using a recent cryptographic protocol called function secret sharing. The framework offers a wide range of functions including ReLU, MaxPool, and BatchNorm, and allows to use of very deep neural networks like AlexNet and ResNet18. They relied on PySyft for its communication layer for FL and support for fixed precision model parameters.

5.6.3 Use Case: FL on Edge Devices

In a comprehensive survey on implementing FL on mobile edge networks in terms of the background, challenges and existing solutions [28], PySyft is mentioned as an existing open-source framework for performing encrypted, privacy-preserving deep learning and implementations of techniques like SMPC and differential privacy in untrusted environments while protecting data. In a research article published in 2020, PySyft was used for FL in IoT Edge devices for training machine learning models in edge devices in a secure and resource-efficient way. The authors explain their use of PySyft as the FL framework as "it provides the Network Worker structure that enables the remote communication of the model and uses the Web Socket protocol to lower overhead, and facilitates real-time data transfer from and to the Server" [12]. The PySyft library is also used by Chaulwar et al in a privacy-preserving FedCollabNN framework for training machine learning models at edge, which is computationally efficient and robust against adversarial attacks [6]. PySyft has also been discussed as a tool for training deep neural networks (DNN) on the Raspberry Pi 4 boards as edge devices [10]. Finally, in [25], Kang et al. proposed a blockchain-empowered secure, scalable, and communication-efficient decentralized Federated Edge Learning (FEL) system with a hierarchical blockchain framework consisting of a main chain and subchains. Their proposed framework was implemented using Pytorch, PySyft, and a blockchain platform named EOSIO.

5.6.4 Use Case: Healthcare and Medical Research

The benefits of PySyft as an adaptable secure and private machine learning framework have been leveraged in several healthcare-related research works. The recent work by Kaissis et al. [24] describes the availability of frameworks like PySyft as instrumental to the widespread application of secure, privacy-preserving, and federated machine learning techniques in medical imaging and medicine in general, and some current review articles reference PySyft concerning medical image workflows, amongst others Quayyum et al. [38], Suzen et al. [43] and Li et al. [27]. Furthermore, several concrete healthcare-related use-cases have been proposed utilizing PySyft. The 2019 work by Passerat-Palmbach et al. presents a combination of FL and Blockchains for use by healthcare consortia. Their proposed system architecture leverages FL building blocks from the PySyft library [36]. In a 2019 research paper by Gao et al. on FL on heterogeneous EEG data for human behavior and emotion recognition tasks, the PySyft framework was used for Horizontal Federated Learning (HFL) [13]. PySyft has also been used in a prototype platform of federated-learning-based quantitative structure-activity relationships (QSAR) modeling for collaborative drug discovery in a work by Chen et al. [7]. Lastly, the recently announced PriMIA (Private Medical Imaging Analysis) library, a collaborative development between the Technical University of Munich, Imperial College London, and OpenMined, builds upon PySyft and PyGrid to offer a single interface for federated training of algorithms and for their provision in an encrypted *inference-as-a-service* scenario, which is showcased in a large-scale, multi-institutional case study on pediatric X-ray image data.

5.6.5 Use Case: Text and Language Processing

Both PySyft and its related NLP library, SyferText, are mentioned by Qayyum et al. in the context of clinical natural language processing [38]. In [50], authors have applied FL frameworks to build a deep convolutional neural network-based unsegmented text recognition model to recognize texts from on a large corpus of Chinese financial documents, which often contain confidential and critical personal information. They used PySyft as one of the prevalent FL frameworks and compared its result with another FL framework called Tensorflow Federated (TFF). Their experiments showed that the use of a FL framework effectively improves the aggregated performance of the text recognition model, without sharing confidential raw image data.

5.6.6 Use Case: Finance, Business or Industry

In a research paper published in 2020, an Industrial Federated Learning (IFL) system has been introduced by Hiessl et al. that supports knowledge exchange in continu-

ously evaluating and updating FL cohorts of learning tasks with sufficient data similarity to enable optimal collaboration of business partners in common ML problems. As future work, they would like to evaluate the incorporation of FL open-source frameworks including PySyft concerning production readiness and support for concurrent communication and computation required by FL cohorts [17]. In another paper published in 2019, Tang et al. designed and implemented a general scheme for privacy-preserving and fair deep learning inference service in a three-worker model under the setting of publicly verifiable covert security [44]. The implementation for secure three-party computation is completely based on PySyft and PyTorch. They also noted a major advantage of PySyft over MiniONN based on PySyft's ability to split and share data and model using PointerTensors as opposed to MiniONN's requirement of transforming an entire existing neural network to an oblivious neural network [44]. Also, a potential use-case of FL and privacy-preserving collaborative learning has been discussed by Kawa et al. in [26] for credit risk assessment where multiple participating institutes will be able to collaboratively learn a shared prediction model while keeping all the data within themselves. Furthermore, in his master's thesis [21] Madeleine presented a framework for training a credit card fraud detection model using Federated Averaging implemented in PySyft.

5.6.7 Use Case: Anomaly Detection

PySyft has also been relied on for building FL-based anomaly detection models. In [42], Singh et al. implemented unsupervised anomaly detection using FL. They used a special type of neural network called autoencoder and used its reconstruction loss as the basis of classifying anomalies. The autoencoder part was obtained from PyTorch library, whereas the FL and aggregation of models were implemented using PySyft tools. Authors reported attaining comparable results to the baseline model which was implemented without FL. Apart from that, as mentioned in [31] by Liu et al., a communication-efficient on-device FL based deep anomaly detection framework for sensing time-series data in Industrial IoT was implemented entirely using PySyft libraries.

5.6.8 Other Use Cases

In [2], a k-Medoids based data partitioning technique in PATE framework has been suggested by Arora et al. to improve the privacy and accuracy of the student model in PATE. For model evaluation, the moment accountant's method in PySyft framework has been used [2]. In another paper published in 2019, Liu et al. have proposed a fingerprinting-based indoor localization system that updates the localization model via a FL framework. As reported, the localization model training and testing are implemented using PyTorch and PySyft [29]. PyTorch and PySyft have also been

used in implementing two novel federated watermarking approaches for embedding watermark into federated DNN models with high accuracy while keeping the functionality of the model [46].

5.6.9 Comparisons and Adaptations to PySyft

For the popular demonstration of training a CNN on MNIST data, PySyft has been compared against other existing frameworks, including the authors own approach for privacy-preserving FL with SMPC protocol based on functional encryption, called HybridAlpha [47]. In this undergraduate thesis paper [45], the authors presented an extension to PyTorch to facilitate differentially private optimization, which they claim is similar to PySyft. Other adaptations of PySyft have been reported in [30, 32] where authors proposed a FL-based Gated Recurrent Unit neural network framework (FedGRU) for traffic flow prediction (TFP) with the consideration of data privacy and security. Secure parameter aggregation mechanism, one of the core components of their proposed model for parameters encryption is adopted from PySyft [32].

5.7 Conclusion

PySyft is a federated learning (FL) library built and maintained by the OpenMined community. OpenMined is an open-source community whose goal is to make the world more privacy-preserving by lowering the barrier-to-entry to technologies for privacy-preserving data science. Our aim is to both help popularize privacy-preserving techniques in machine learning by making them as accessible as possible. We hope this chapter encourages you to try the PySyft library, and for a friendlier experience, try Duet: our tool for easier FL for scientists and data owners. We provide extensive PySyft and Duet tutorials on our GitHub and have a responsive community on slack to help answer your questions.

If you have any questions, join us at slack.openmined.org. If you would like to contribute, get involved with the community, or if you have any questions at all, please see our Welcome Package at https://github.com/OpenMined/OM-Welcome-Package/, we look forward to meeting you.

References

1. S. Ahmed, R.S. Mula, S.S. Dhavala, A framework for democratizing ai (2020). arXiv:2001.00818
2. H. Arora, Guided pate for scalable learning. https://www2.isye.gatech.edu/~fferdinando3/cfp/PPAI20/papers/paper_20.pdf. Accessed 29 May 2020

3. M.U. Asad, A.A. Moustafa, T. Ito, A. Muhammad, Evaluating the communication efficiency in federated learning algorithms (2020). arXiv:2004.02738
4. J.D. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, N.D. Lane, Flower: a friendly federated learning research framework (2020)
5. F. Boemer, R. Cammarota, D. Demmler, T. Schneider, H. Yalame, Mp2ml: a mixed-protocol machine learning framework for private inference. Cryptology ePrint Archive, Report 2020/721 (2020), https://eprint.iacr.org/2020/721
6. A. Chaulwar, Private dataset generation using privacy preserving collaborative learning (2020). arXiv:2004.13598
7. S. Chen, D. Xue, G. Chuai, Q. Yang, Q. Liu, Fl-qsar: a federated learning based qsar prototype for collaborative drug discovery. bioRxiv (2020)
8. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, N.P. Smart, Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits, in *European Symposium on Research in Computer Security* (Springer, 2013), pp. 1–18
9. I. Damgård, V. Pastro, N. Smart, S. Zakarias, Multiparty computation from somewhat homomorphic encryption, in *Advances in Cryptology - CRYPTO 2012*, ed. by R. Safavi-Naini, R. Canetti (Springer, Berlin, 2012), pp. 643–662
10. A. Das, T. Brunschwiler, Privacy is what we care about: experimental investigation of federated learning on edge devices, in *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, AIChallengeIoT'19, New York, NY, USA. Association for Computing Machinery (2019), pp. 39–42
11. Wang et al., Beyond inferring class representatives: user-level privacy leakage from federated learning (2018). arXiv:1812.00535
12. A. Feraudo, P. Yadav, V. Safronov, D.A. Popescu, R. Mortier, S. Wang, P. Bellavista, J. Crowcroft, Colearn: enabling federated learning in mud-compliant iot edge networks, in *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking* (2020), pp. 25–30
13. D. Gao, C. Ju, X. Wei, Y. Liu, T. Chen, Q. Yang, Hhhfl: hierarchical heterogeneous horizontal federated learning for electroencephalography (2019). arXiv:1909.05784
14. Z. Ghodsi, T. Gu, S. Garg. Safetynets: verifiable execution of deep neural networks on an untrusted cloud, in *Advances in Neural Information Processing Systems* (2017), pp. 4672–4681
15. C. He, M. Annavaram, S. Avestimehr, Fednas: Federated deep learning via neural architecture search (2020)
16. C. He, S. Li, J. So, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, S. Avestimehr, Fedml: A research library and benchmark for federated machine learning (2020)
17. T. Hiessl, D. Schall, J. Kemnitz, S. Schulte, Industrial federated learning - requirements and system design (2020). arXiv:2005.06850
18. M. Hirt, D. Tschudi, Efficient general-adversary multi-party computation, in *International Conference on the Theory and Application of Cryptology and Information Security* (Springer, 2013), pp. 181–200
19. H. Sixu, X. Yuan Li, Q.L. Liu, W. Zhaomin, B. He, The oarf benchmark suite: Characterization and implications for federated learning systems (2020)
20. S. Ickin, K. Vandikas, M. Fiedler, Privacy preserving qoe modeling using collaborative learning, in *Proceedings of the 4th Internet-QoE Workshop on QoE-Based Analysis and Management of Data Communication Networks*, Internet-QoE'19, New York, NY, USA. Association for Computing Machinery (2019), pp. 13–18
21. M. Jansson, M. Axelsson, Federated learning used to detect credit card fraud. Master's thesis. Accessed 19 June 2020
22. D. Kahrobaei, A. Wood, K. Najarian, Homomorphic encryption for machine learning in medicine and bioinformatics. ACM Comput. Surv. (2020)
23. P. Kairouz, H.B. McMahan, B. Avent, A. Bellet, M. Bennis, A.N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R.G.L. D'Oliveira, S.E. Rouayheb, D. Evans, J. Gardner,

Z. Garrett, Adrion, B. Ghazi, P.B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konecný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S.U. Stich, Z. Sun, A.T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F.X. Yu, H. Yu, S. Zhao, Advances and Open Problems in Federated Learning (2019). arXiv:1912.04977 [cs, stat]
24. G.A. Kaissis, M.R. Makowski, D. Rückert, R.F. Braren, Secure, privacy-preserving and federated machine learning in medical imaging. Nat. Mach. Intell. 1–7 (2020)
25. J. Kang, Z. Xiong, C. Jiang, Y. Liu, S. Guo, Y. Zhang, D. Niyato, C. Leung, C. Miao, Scalable and communication-efficient decentralized federated edge learning with multi-blockchain framework (2020)
26. D. Kawa, S. Punyani, P. Nayak, A. Karkera, V. Jyotinagar, Credit risk assessment from combined bank records using federated learning. Int. Res. J. Eng. Technol. (IRJET) **6** (2019)
27. Q. Li, Z. Wen, B. He, Federated learning systems: Vision, hype and reality for data privacy and protection (2019). arXiv:1907.09693
28. W.Y.B. Lim, N.C. Luong, D.T. Hoang, Y. Jiao, Y. Liang, Q. Yang, D. Niyato, C. Miao, Federated learning in mobile edge networks: a comprehensive survey. IEEE Commun. Surv. Tutor. **22**(3), 2031–2063 (2020)
29. Y. Liu, H. Li, J. Xiao, H. Jin, Floc: fingerprint-based indoor localization system under a federated learning updating framework, in *2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)* (2019), pp. 113–118
30. Y. Liu, J.J.Q. Yu, J. Kang, D. Niyato, S. Zhang, Privacy-preserving traffic flow prediction: a federated learning approach. IEEE Internet Things J. **7**(8), 7751–7763 (2020)
31. Y. Liu, S. Garg, J. Nie, Y. Zhang, Z. Xiong, J. Kang, M.S. Hossain, Deep anomaly detection for time-series data in industrial iot: a communication-efficient on-device federated learning approach. IEEE Internet Things J. 1–1 (2020)
32. Y. Liu, S. Zhang, C. Zhang, J.J.Q. Yu, Fedgru: Privacy-preserving traffic flow prediction via federated learning (2020)
33. R. Mayer, H.-A. Jacobsen, Scalable deep learning on distributed infrastructures: challenges, techniques, and tools. ACM Comput. Surv. **53**(1) (2020)
34. C. Milani, Protecting against linkage attacks that use 'anonymous data'
35. A. Neupane, Homomorphic learning: a privacy-focused approach to machine learning. IEEE Comput. Soc. (2019)
36. J. Passerat-Palmbach, T. Farnan, R. Miller, M.S. Gross, H. Flannery, B. Gleim, A blockchain-orchestrated federated learning architecture for healthcare consortia (2019). arXiv:1910.12603
37. A. Paszke, et al., Basic MNIST example (2016). Accessed 15 Nov 2020
38. A. Qayyum, J. Qadir, M. Bilal, A. Al-Fuqaha, Secure and robust machine learning for healthcare: a survey (2020). arXiv:2001.08103
39. N. Rodríguez-Barroso, G. Stipcich, D. Jiménez-López, J.A. Ruiz-Millán, E. Martínez-Cámara, G. González-Seco, M.V. Luzón, M.A. Veganzones, F. Herrera, Federated learning and differential privacy: software tools analysis, the sherpa.ai fl framework and methodological guidelines for preserving data privacy. Inf. Fus. **64**, 270–292 (2020)
40. T. Ryffel, D. Pointcheval, F. Bach, Ariann: low-interaction privacy-preserving deep learning via function secret sharing (2020)
41. N.C. Sameer Wagh, D. Gupta, Securenn: 3-party secure computation for neural network training (2018)
42. S. Singh, S. Bhardwaj, H. Pandey, G. Beniwal, Anomaly detection using federated learning, in *Proceedings of International Conference on Artificial Intelligence and Applications*, ed. by P. Bansal, M. Tushir, V.E. Balas, R. Srivastava, vol. 1164 (Springer Singapore, Singapore, 2021), pp. 141–148
43. A. Süzen, M. Simsek, A novel approach to machine learning application to protection privacy data in healthcare: federated learning. Namik Kemal Tip Dergisi **8**, 22–30 (2020)
44. F. Tang, J. Hao, J. Liu, H. Wang, M. Xian, PFDLIS: privacy-preserving and fair deep learning inference service under publicly verifiable covert security setting. Electronics **8**(12), 1488 (2019)

45. C. Waites, PyVacy: towards practical differential privacy for deep learning. Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2019
46. Y. Xia, Watermarking federated deep neural network models. G2 pro gradu, diplomityö. Accessed 16 March 2020
47. R. Xu, N. Baracaldo, Y. Zhou, A. Anwar, H. Ludwig, Hybridalpha: an efficient approach for privacy-preserving federated learning, in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19, New York, NY, USA. Association for Computing Machinery (2019), pp. 13–23
48. C. Yang, Q.P. Wang, M. Xu, S. Wang, K. Bian, X. Liu, Heterogeneity-aware federated learning (2020)
49. Q. Yang, Y. Cheng, Y. Kang, T. Chen, H. Yu, *Federated Learning*, vol. 13, 3rd edn. (Morgan & Claypool Publishers, San Rafael, 2019)
50. X. Zhu, J. Wang, Z. Hong, T. Xia, J. Xiao, Federated learning of unsegmented chinese text recognition model, in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* (2019), pp. 1341–1345