Open Federated Learning is a Python3 library for federated learning. It enables organizations to collaborately train a model without sharing sensitive information with each other.

There are basically two components in the library: the collaborator which uses local sensitive dataset to fine-tune the aggregated model and the aggregator which receives model updates from collaborators and distribute the aggregated models.

The aggregator is framework-anostic, while the collaborator can use any deep learning frameworks, such as Tensorflow or Pytorch.

Open Federated Learning is developed by Intel Labs and Intel Internet of Things Group.
1.1 Overview

1.1.1 What is Federated Learning?

Federated learning is a distributed machine learning approach that enables organizations to collaborate on machine learning projects without sharing sensitive data, such as, patient records, financial data, or classified secrets (McMahan, 2016; Sheller, Bakas et al., 2020; Yang, Liu, Chen, & Tong, 2019). The basic premise behind federated learning is that the model moves to meet the data rather than the data moving to meet the model. Therefore, the minimum data movement needed across the federation is solely the model parameters and their updates.

Fig. 1: Federated Learning
1.1.2 Design Philosophy

The overall design is that all of the scripts are built off of the federation plan. The plan is just a YAML file that defines the collaborators, aggregator, connections, models, data, and any other parameters that describes how the training will evolve. In the “Hello Federation” demos, the plan will be located in the YAML file: bin/federations/plans/keras_cnn_mnist_2.yaml. As you modify the demo to meet your needs, you’ll effectively just be modifying the plan along with the Python code defining the model and the data loader in order to meet your requirements. Otherwise, the same scripts will apply. When in doubt, look at the FL plan’s YAML file.

1.1.3 Digital Certificates and PKI Requirements

By default, Open Federated Learning uses the TLS protocol to secure all network connections.

NOTE: Proper consideration needs to be taken when deploying Open Federated Learning. The scripts that create the custom PKI for the project do not have a mechanism to sign any of the certificates by a proper Certificate Authority. The CA certificate that is created will only be self-signed. Trust in this certificate therefore will be based on your ability to properly secure the private key for that certificate, as well as securely distribute the CA Certificate to your endpoints.

Use at your own risk.

For the TLS connection between the collaborators and the aggregator, each node (aggregator/collaborator) in the federation requires:

1. A cert_chain.crt file that holds the certificate/public key for the trusted signer.
2. A .crt file that holds the public key and common name for the node, signed by the trusted signer.
3. A .key file that holds the private key that goes with the .crt file.

Additionally, aggregator node certificates should be server certificates, while collaborator node certificates should be client certificates.

Note: The common name (CN) of the aggregator certificate must either be the FQDN of the aggregator or the IP address.

Note: The common name (CN) of the collaborator certificate must match the name used by the collaborator. The aggregator has a list of approved common names and will reject connections from collaborators whose cert CNs are not in that list.

1.1.3.1 Single Collaborator Cert Common Name Mode for testing

Open Federated Learning provides a convenience option called “Single Collaborator Cert Common Name” (SCN) mode that allows developers/testers to re-use the same collaborator certificate for each collaborator. This should only be use in fully trusted test environments on the same trusted network, and should never be used if any nodes are not under direct control of the tester/developer! Normally, the aggregator checks the cert to ensure that the collaborator name matches the common name in the certificate. This mode allows a collaborator node to masquerade as any collaborator by instead instructing the aggregator to check that the common name in the cert matches a specific name given to it when launched. Therefore, the collaborator process can claim any name it wishes, so long as it presents a certificate with that specific common name. This is especially useful for a test environment where collaborator nodes may run on different machines at different times. (Note that in SCN mode, the collaborator name must be in the approved list. The collaborator name just doesn’t have to match the CN in the cert it uses).
To enable SCN mode, pass `-scn <common name>` to each process in the federation.

**Note:** Nodes that have been launched with different SCN settings will refuse to connect.

Finally, it is possible to disable TLS entirely. **Do this at your own risk.** In the *Setting Defaults for Network settings* section, you will see a “disable_tls” configuration option.

**Note:** Some IT departments configure networks to drop unencrypted RPC traffic like gRPC. In such cases, disabling TLS could prevent the nodes from connecting.

---

### 1.2 Installation and Setup

#### 1.2.1 Installation

Open Federated Learning consists of a core package set and 3 optional package sets:

1. The core OpenFL packages require no machine learning frameworks (all numpy-based). This includes the logic for the aggregator, collaborator, network, and model/data interfaces.

2. The PyTorch packages for model and data baseclasses to simplify porting PyTorch models to OpenFL. (Optional)

3. The TensorFlow packages for model and data baseclasses to simplify porting TensorFlow models to OpenFL. (Optional)
Open Federated Learning

4. The FeTS-AI packages, as a submodule, that contains the FeTS-AI model and data classes. (Optional, requires submodule init)

Our scripts create a Python 3 virtual environment at ./venv which we use to run our python scripts. You can use the make file to either install these packages in this virtual-environment, or to create wheel files for you to install in another environment.

1.2.1.1 Requirements

On each machine in your federation, you will need:

1. Python 3.5+
2. Python virtual environments

Note: You can install virtual environment support in your Python3 installation via:

```
$ python3 -m pip install --user virtualenv
```

If you have trouble installing the virtual environment, make sure you have Python 3 installed on your OS. For example, on Ubuntu:

```
$ sudo apt-get install python3-pip
```

See the official Python website for more details.

1.2.1.2 Installing In The OpenFL Virtual environment

To install the core OpenFL package in ./venv, navigate to the root OpenFL directory and run:

```
$ make install_openfl
```

This will create the virtual environment install the core OpenFL packages.

Note: The Python version used will be the same Python version referenced as Python3 by your system.

For the optional PyTorch packages, run:

```
$ make install_openfl_pytorch
```

Note: You will need to install pytorch and torchvision as detailed here: Pytorch website. To install in the virtual environment, use pip as: venv/bin/pip

For the optional TensorFlow packages, run:

```
$ make install_openfl_tensorflow
```

Finally, to download the FeTS-AI algorithms, need to initialize the submodule and run the make recipe:

```
$ git submodule update --init --recursive
$ make install_fets
```
### 1.2.1.3 (Optional) Building Wheel Files

If you want to install OpenFL and related optional packages in another Python3 environment, you can build the wheel files with the make commands:

```bash
$ make openfl_whl
$ make openfl_pytorch_whl
$ make openfl_tensorflow_whl
$ make fets_whl
```

**Note:** Running OpenFL in containers (e.g. Docker, Singularity) is a natural solution to simplify deployment, and fairly straightforward. We welcome contributions towards such a solution.

### 1.2.2 Configuring a Development/Testing Setup

This section walks you through configuring one or more machines to run test federations for testing/development.

**Prerequisites:**

1. Clone the OpenFL repository, and navigate to the root directory.
2. Determine the fully-qualified domain name or IP address of the machine you want to use as your aggregator.
   
   For example, you can use the following linux/cygwin command:

   ```bash
   $ hostname --fqdn
   ```

#### 1.2.2.1 Setting Defaults for Network settings

FL Plans support default configurations to make it easier to share settings between FL Plans. This also makes it easier to customize another’s plan without changing the main plan file, so you can share plans with other developers.

The sample plans included with OpenFL all depend on a defaults file for their network configurations. This allows a developer a single location in which to set their network configuration for their various test plans.

We include an example under `bin/federations/plans/defaults/network.yaml.example` that you can copy into the default filename for this configuration, then edit to your specification. Let’s do this now:

1. First, we need to copy the example file that we’re going to change. Copy it and take a look at it:

   ```bash
   $ cp bin/federations/plans/defaults/network.yaml.example bin/federations/plans/...defaults/network.yaml
   $ cat bin/federations/plans/defaults/network.yaml
   ...
   auto_port : True
   cert_folder : pki
   hash_salt : Anything you want. Make it unique
   init_kwargs :
   agg_addr : FQDN of aggregator machine
   disable_tls : False
   disable_client_auth : False
   ...
   ```

2. Open your new file in an editor:
Open Federated Learning

3. First, we need to set the aggregator address to the FQDN or IP of the aggregator machine, such as:

```yaml
agg_addr : msheller-aggregator.intel.com
```

3. Next, you can choose a specific port, or if you intend to run multiple aggregator processes for testing, leave it as 'auto'. 'auto' simply uses federation UUID (which is a hash of the FL Plan files, including the defaults files) to pick a random port. This way the collaborators and aggregator will compute the same “random” port.

```yaml
agg_port : auto # I am keeping it auto because I run lots of federations...
```

4. Finally, in development teams with shared machines, it is possible for FL Plans to be exactly identical. This leads to identical FL Plan UUIDs (hashes). For this reason, we give our plans a silly salt. It can be anything, so long as it is unique among your team:

```yaml
hash_salt : micah.j.sheller@intel.com # your email isn't a bad choice
```

Now your FL Plans will use your aggregator machine, and if it is shared, you shouldn’t likely run into port choice conflicts.

**Note:** Here is where you can do things like disable tls or change which directory you use for your certs. **We don't condone disabling TLS.**

### 1.2.2.2 Creating Collaborator Lists

When an aggregator executes an FL Plan, it also requires a list of collaborator names that are allowed to participate. In a production setting, these names are meaningful and are tightly coupled with each client’s digital certificate used in the TLS connection. However, for test environments, you can name them whatever you wish (you will be passing these on the collaborator commandlines). You can find existing test lists under:

```bash
$ ls -l bin/federations/collaborator_lists
```

```
-rw-r--r-- 1 msheller intelall 46 Jul 6 15:01 col_one_big.yaml
-rw-r--r-- 1 msheller intelall 147 Jul 6 15:01 cols_10.yaml
-rw-r--r-- 1 msheller intelall 52 Jul 6 15:01 cols_2.yaml
-rw-r--r-- 1 msheller intelall 432 Jul 6 15:01 cols_32.yaml
-rw-r--r-- 1 msheller intelall 40 Jul 6 15:01 only_col_2.yaml
-rw-r--r-- 1 msheller intelall 52 Jul 6 15:01 only_cols_2_and_3.yaml
```

And you’ll see that they have very exciting contents, such as:

```bash
$ cat bin/federations/collaborator_lists/cols_10.yaml
```

```
collaborator_common_names :
- 'col_0'
- 'col_1'
- 'col_2'
- 'col_3'
- 'col_4'
- 'col_5'
- 'col_6'
- 'col_7'
```

(continues on next page)
In a real setting, these lists would hold the common names in the certificates of the collaborators (one per cert). In a development/test environment, feel free to use any naming-convention. You will need these names later, so we recommend keeping them simple. Note that you may want to run multiple collaborators on a single machine, so you may not want to use machine names here. (TODO: Add reference to auto-lists when we implement that convenience feature).

1.2.2.3 Configuring Collaborator Local Data Directories

When a collaborator executes an FL Plan, the FL Plan will contain a data_name entry such as “brats” or “mnist_shard” or similar. This name serves as a key in a dictionary of paths or shards on the collaborator (we use “shards” to refer to tests where a single dataset is split among collaborators at runtime, i.e. “sharded”). We store these mappings in .yaml files of a structure:

```
collaborator_common_name:
  data_name: <path or shard>
```

This way, we can configure the data-paths for multiple collaborators in a single file. In production, such a file would only have the information for a single collaborator.

You’ll find one such file in the repository that looks like this:

```
$ cat bin/federations/local_data_config.yaml
 collaborators:
   col_one_big:
     brats: '/raid/datasets/BraTS17/by_institution_NIfTY/0-9'
   col_0:
     brats: '/raid/datasets/BraTS17/by_institution_NIfTY/0'
     mnist_shard: 0
     cifar10_shard: 0
   col_1:
     brats: '/raid/datasets/BraTS17/by_institution_NIfTY/1'
     mnist_shard: 1
     cifar10_shard: 1
   ...
```

For the shards, you’ll usually just need an index. For datasets that are already separated, you need to set the paths for each collaborator/dataset pair here. Note that in our case, we have a shared /raid volume that each of our development nodes can access. This makes life easy, and also ensures we can run any collaborator on any machine. Highly recommended for testing and development! We even go so far as using softlinks to allow various collaborator assignments (e.g. moving data around to increase collaborator-specific biases).
1.2.3 Creating the Test PKI (Optional)

See *Digital Certificates and PKI Requirements* for an overview of how OpenFL uses TLS with digital certificates.

This section explains how to use some convenience open-ssl scripts to generate a test PKI.

**NOTE:** Proper consideration needs to be taken when deploying Open Federated Learning. The scripts that create the custom PKI for the project do not have a mechanism to sign any of the certificates by a proper Certificate Authority. The CA certificate that is created will only be self-signed. Trust in this certificate therefore will be based on your ability to properly secure the private key for that certificate, as well as securely distribute the CA Certificate to your endpoints.

Use these scripts at your own risk.

**Note:** You do not need to create a PKI to use the single process simulation mode, as there is no network involved in simulation, and thus no TLS.

**Note:** This tutorial relies on *Single Collaborator Cert Common Name Mode for testing*, which is a convenience feature for testing and development and should never be used in production.

### 1.2.3.1 Create the Certificate Authority and Signing Key

1. Change the directory to `bin/federations/pki`:

   ```bash
   $ cd bin/federations/pki
   ```

2. Run the Certificate Authority script. This will create a Certificate Authority for the Federation on this node. All certificates will be signed by this signing key. Follow the command-line instructions and enter in the information as prompted. The script will create a simple database file to keep track of all issued certificates.

   ```bash
   $ bash setup_ca.sh
   ```

**Note:** This is not a proper way to manage a real CA. We are not giving guidance on how to protect keys!

This will create a root key, signing key, and public cert_chain. You should find the ‘cert_chain.crt’ under `bin/federations/pki`:

```bash
$ ls -l .
drwxr-xr-x 4 msheller intelall 4096 Jul 1 12:52 ca
-rw-r--r-- 1 msheller intelall 9079 Aug 10 08:58 cert_chain.crt
drwxr-xr-x 2 msheller intelall 4096 Aug 19 16:23 client
drwxr-xr-x 2 msheller intelall 4096 Jun 10 15:13 config
-rw-r--r-- 1 msheller intelall 1684 Aug 18 12:02 create-aggregator.sh
-rw-r--r-- 1 msheller intelall 1660 Aug 18 12:02 create-and-sign-aggregator.sh
-rw-r--r-- 1 msheller intelall 1035 Aug 18 12:02 create-and-sign-collaborator.sh
-rw-r--r-- 1 msheller intelall 1083 Aug 18 12:02 create-collaborator.sh
-rw-r--r-- 1 msheller intelall 99 Jun 10 15:13 README.md
drwxr-xr-x 2 msheller intelall 4096 Aug 19 16:10 server
-rw-r--r-- 1 msheller intelall 1394 Jun 10 15:13 setup_ca.sh
-rw-r--r-- 1 msheller intelall 629 Aug 13 13:56 sign-csr.sh
```
Open Federated Learning

Every aggregator or collaborator machine will need this cert_chain.crt file in this same location. This is how that node can verify signatures made by your signing key. Without this file on each machine, the TLS connections will fail.

3. Run the aggregator cert script, replacing AGG.FQDN with the actual fully qualified domain name (FQDN) for the aggregator machine. You may optionally include the IP address for the aggregator, replacing [IP_ADDRESS].

```bash
$ bash create-and-sign-aggregator.sh AGG.FQDN
```

**Note:** You can discover the FQDN with the Linux command:

```bash
$ hostname --all-fqdns | awk '{print $1}'
```

After creating this certificate, you should see the following files under bin/federations/pki/server:

```bash
$ ls -l ./server
-rw-r--r-- 1 msheller intelall 4704 Aug 19 16:10 AGG.FQDN.crt
-rw------- 1 msheller intelall 1708 Aug 19 16:10 AGG.FQDN.key
```

You will need to move these files to the same location on aggregator node.

4. Next we create collaborator certificates. Normally, you want to create a certificate for each collaborator. However, in testing environments, this is overly-burdensome to manage, and not necessary if you are only testing that TLS is working and all the machines are under your control and trusted. Instead, we will create a single collaborator certificate for all our test collaborator processes. **This is not appropriate for actual TLS use cases.** Pick a name for your test_collaborator certificate. You will be passing this name as an argument to every collaborator/aggregator process.

```bash
$ bash create-and-sign-collaborator.sh MICAH.TEST.COLLABORATOR.CERT
```

**Note:** I don’t advise using my name :) Pick something more meaningful.

Now you should have the following files under bin/federations/pki/client:

```bash
$ ls -l ./client
-rw-r--r-- 1 msheller intelall 4655 Aug 19 16:23 MICAH.TEST.COLLABORATOR.CERT.crt
-rw------- 1 msheller intelall 1704 Aug 19 16:23 MICAH.TEST.COLLABORATOR.CERT.key
```

Each collaborator machine will need a copy of these files in this same location.

**Note:** Beating a dead horse here: a production-worthy PKI involves some real form of identity verification. Generating keys, signing them, then giving them out is NOT proper key management. This is for testing/development purposes only!
1.2.3.2 Summary of Test PKI files

After creating and transferring files around you should have:

Table 1: Collaborator PKI Files (on each collaborator machine)

<table>
<thead>
<tr>
<th>File Type</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certificate chain</td>
<td>bin/federations/pki/cert_chain.crt</td>
</tr>
<tr>
<td>Collaborator certificate</td>
<td>bin/federations/pki/client/MICAH.TEST.COLLABORATOR.CERT.crt</td>
</tr>
<tr>
<td>Collaborator key</td>
<td>bin/federations/pki/client/MICAH.TEST.COLLABORATOR.CERT.key</td>
</tr>
</tbody>
</table>

Table 2: Aggregator PKI Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certificate chain</td>
<td>bin/federations/pki/cert_chain.crt</td>
</tr>
<tr>
<td>Aggregator certificate</td>
<td>bin/federations/pki/server/AGG.FQDN.crt</td>
</tr>
<tr>
<td>Aggregator key</td>
<td>bin/federations/pki/server/AGG.FQDN.key</td>
</tr>
</tbody>
</table>

1.3 Creating Initial Weights

At the start of federated training, the aggregator shares the initial global model with all collaborators to use for their model initializations. Collaborator model state is held locally in the form of a ‘tensor dictionary’, providing a map of parameter names to numpy array values. These tensor dictionaries are split by our framework code (using default logic, which can be changed via the flplan) into a portion for global sharing (parameters to be aggregation for example) and a portion for holding-out from sharing. The shared portion is converted into a protobuf file for serialization, in order to be sent over the network or saved to file. The initial weights file is such a file, holding the initial global state of the model to be used for a particular federation.

The initial weights file can be derived from another file produced by a native ML framework (such as PyTorch, Tensorflow, . . . ) or can be generated without such a file - using random initialization of the model to generate the state.

In both cases, the initial weights file is generated using the script, ‘/bin/create_initial_weights_file_from_flplan.py’, which is run from the bin directory using:

```bash
../venv/bin/python create_initial_weights_file_from_flplan.py -p <fplan fname> -c <collaborator list fname>
```

This script creates an instance of the model provided in the plan, populates it’s state (via a native weights file or random initialization), then pulls and saves a serialized form of the globally shared state to disk in the folder ‘bin/federations/weights/’ under the filename provided in the plan.

Because a model is instantiated, dependencies of model instantiation (such as an example feature shape and number of output classes) need to be provided. Currently such information can passed as arguments to the script directly, or inferred from the data that is defined in the plan.

Passing of the feature shape and number of classes to the script is done via the arguments -fs and -nc respectively (-fs should be followed by a list of integers separated by spaces. Allowing the script to infer this information requires that a collaborators list file name is passed to the script (using the -c argument), as well as that an entry can be found in the local_data_config corresponding to the first collaborator in the corresponding list.

Finally, if utilizing a native weights file, pass the absolute path of the file to the script using the -nmfw argument. Note: Only PyTorch native weights are currently supported.
1.4 Running a Federation Simulation

1.4.1 Running a Federation Simulation (no network, single process)

When exploring the convergence properties of federated learning for a particular use-case, it is helpful to run several federations in parallel, each of which runs the aggregator and collaborators (round-robin) in a single process avoiding the need for network communication. We describe here how to run one of these simulations.

Note that much of the code used for simulation (e.g., collaborator and aggregator objects) is the same as for the multiprocess solution with grpc. Since the collaborator calls the aggregator object methods via the grpc channel object, simulation is performed by simply replacing the channel object provided to each collaborator with the aggregator object.

Simulations are run from an flplan, and in fact the same flplan that is used for a multi-process federation can be used. Note: Simulations utilize a single model, with each new collaborator taking control of the model when it is their turn in the round-robin. It is therefore critical that the model, ‘set_tensor_dict’ method completely overwrites all substantive model state in order that state does not leak from the collaborator who previously held the model.

1.4.2 The steps for running a simulation

1. Go through the steps for project Installation and Setup with the following in mind. In this case, we require make install_openfl as well as installation of ML framework libraries (supported now are make install_openfl_tensorflow and make install_openfl_pytorch) as we will be running both the aggregator and all collaborators (and thus models) here. Though the network will not be used here, we still currently require that the flplan default network file is present even when running simulations, however it’s contents can be exactly the same as the example file as far as running simulation goes. Finally, the PKI creation step can be skipped as we will not be using the network here.

2. From the bin directory, run the following command (see the notes on creating_initial_weights from the flplan for further options on parameters to this script):

```
$ ../venv/bin/python create_initial_weights_file_from_flplan.py -p <flplan filename> -c <collaborators list filename>
```

3. Again from the bin directory, kick off the simulation by running the following:

```
$ ../venv/bin/python run_simulation_from_flplan.py -p <flplan filename> -c <collaborators list filename>
```

4. You’ll find the output from the aggregator in bin/logs/aggregator.log. Grep this file to see results. You can check the progress as the simulation runs, if desired. The aggregator.log is always appended to, so will include results from previous runs.

1.4.3 Running a Federation Simulation (MNIST Example)

Here we will use an example flplan (bin/federations/plans/keras_cnn_mnist_10.yaml) that will work in conjunction with an example collaborators list (bin/federations/collaborator_lists/cols_10.yaml) for which entries already exist in the example local data config (bin/federations/local_data_config.yaml) enabling a predetermined sharding of the MNIST public dataset across 10 collaborators.
Open Federated Learning

1.4.3.1 Setup and Installation

1. Clone the repository onto a Linux machine that has Python 3.5 or greater, and the virtualenv library installed.
2. Enter the project root directory, and install the project with support for Keras models.

```
$ make install_openfl install_openfl_tensorflow
```

3. Make an exact copy of the example network configuration to ensure one exists (it needs to be there, but in this case it's contents are not important).

```
$ cp bin/federations/plans/defaults/network.yaml.example bin/federations/plans/defaults/network.yaml
```

1.4.3.2 Creation of Initial Weights

4. Create the initial weights file by running the following command from the bin directory:

```
$ ../venv/bin/python create_initial_weights_file_from_flplan.py -p keras_cnn_mnist_10.yaml -c cols_10.yaml
```

1.4.3.3 Launch the Simulated Federation

5. Again from the bin directory, kick off the simulation by running the following:

```
$ ../venv/bin/python run_simulation_from_flplan.py -p keras_cnn_mnist_10.yaml -c cols_10.yaml
```

1.4.3.4 Monitor the Progress

6. You’ll find the output from the aggregator in bin/logs/aggregator.log. Grep this file to see results (one example below). You can check the progress as the simulation runs, if desired.

```
$ pwd
/home/<user>/git/openfl/bin
$ grep -A 2 "round results" logs/aggregator.log
2020-03-30 13:45:33,404 - openfl.aggregator.aggregator - INFO - round results for model id/version KerasCNN/1
2020-03-30 13:45:33,404 - openfl.aggregator.aggregator - INFO - validation: 0.446
2020-03-30 13:45:33,404 - openfl.aggregator.aggregator - INFO - loss: 1.063
--
2020-03-30 13:45:35,127 - openfl.aggregator.aggregator - INFO - round results for model id/version KerasCNN/2
2020-03-30 13:45:35,127 - openfl.aggregator.aggregator - INFO - validation: 0.863
2020-03-30 13:45:35,127 - openfl.aggregator.aggregator - INFO - loss: 0.413
```

Note that aggregator.log is always appended to, so will include results from previous runs.
1.4.3.5 Explore Modifications

7. Perform a new simulation using 32 collaborators instead of 10 (using the plan, ‘keras_cnn_mnist_32.yaml’) to see how this effects the learning curve. Explore further modifications by copying and editing existing plans.

1.5 Running a Federation (MNIST Example)

We will be training an MNIST classifier using federated learning and two collaborators. We will use an flplan (keras_cnn_mnist_2.yaml) already provided in the repo inside /bin/federations/plans, as well as a provided collaborators list file (cols_2.yaml inside /bin/federations/collaborator_lists) containing two collaborator names, ‘col_0’ and ‘col_1’. Both collaborator names are already provided in the default local_data_config file so that the framework can locate the collaborator specific data information (which in this case consists of a shard number to use for hard-coded sharding logic that is performed after grabbing MNIST from a hard coded online location).

1.5.1 Setup and Configuration

We will show you how to set up Open Federated Learning using a simple MNIST dataset and a TensorFlow/Keras CNN model as an example.

1.5.1.1 On the Aggregator

1. Go through Installation and Setup on the aggregator machine with the following in mind. In this case, we require make install_openfl and make install_openfl_tensorflow. The tensorflow piece is required, as we will have the aggregator create the initial weights using the model code. The aggregator does not require this otherwise and indeed the creation of initial weights can be done on a collaborator machine and copied to the aggregator (who needs it to start the federation) if you wish. Though the collaborator list we use here is already provided in the repository (and is already compatible with the flplan and local_data_config default file), you will also need to make sure you create a copy of the network config as part of this setup and enter in the appropriate FQDN for the machine that is running the aggregator. Finally the test PKI will also be needed, and we will set up a single cert (use the common name, ‘test’, to be shared by the two collaborators).

2. Create the initial weights file on the aggregator machine by running the following command from the bin directory:

$ ../venv/bin/python3 create_initial_weights_file_from_flplan.py -p keras_cnn_mnist_2.yaml -c cols_2.yaml

1.5.1.2 On each Collaborator Running on a Different Machine than the Aggregator

3. Go through installation and setup (see Installation and Setup), with the following in mind. In this case, we require make install_openfl and make install_openfl_tensorflow. The tensorflow piece is required, as we will be running the models here. Finally, copy over the following files from the aggregator machine to these machines:

<table>
<thead>
<tr>
<th>File Type</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>default network file</td>
<td>bin/federations/plans/defaults/network.yaml</td>
</tr>
<tr>
<td>certificate authority cert</td>
<td>bin/federations/pki/cert_chain.crt</td>
</tr>
<tr>
<td>shared client test public key</td>
<td>bin/federations/pki/col_test/col_test.crt</td>
</tr>
<tr>
<td>shared client test private key</td>
<td>bin/federations/pki/col_test/col_test.key</td>
</tr>
</tbody>
</table>
1.5.2 Running the Federation

1.5.2.1 On the Aggregator

1. To start the aggregator, run the following script from the bin directory. Note that we will need to pass in the shared single collaborator cert common name in order to specify that we are running in single cert test mode (this mode should only be used for testing purposes).

```
$ ../venv/bin/python3 ./run_aggregator_from_flplan.py -p keras_cnn_mnist_2.yaml -scn test --c cols_2.yaml
```

At this point, the aggregator is running and waiting for the collaborators to connect. When all of the collaborators connect, the aggregator starts training. When the last round of training is complete, the aggregator stores the final weights in the protobuf file that was specified in the YAML file (in this case `keras_cnn_mnist_latest.pbuf`).

1.5.2.2 On each Collaborator [col_number = ‘0’ and ‘1’]:

2. From the bin directory, run the collaborator using the following script, passing in the shared single collaborator cert common name in order to specify that we are running in single cert test mode (this mode should only be used for testing purposes).

```
$ ../venv/bin/python3 ./run_collaborator_from_flplan.py -p keras_cnn_mnist_2.yaml -col col_<col_number> -scn test
```
CHAPTER
TWO

INDICES AND TABLES

- genindex
- modindex
- search