Swiss Institute of Bioinformatics

Dalle Molle Institute for Artificial Intelligence USI–SUPSI

Federated Learning in Bioinformatics

SIB Training Course

Practical Sessions (Handout)

Instructors

Dr. Daniele Malpetti, *Lecturer–Researcher*
Dr. Laura Azzimonti, *Senior Lecturer–Researcher*
Dr. Sandra Mitrović, *Researcher*
Dr. Lea Multerer, *Researcher*
Dr. Francesco Gualdi, *Researcher*
Dr. Francesca Mangili, *Senior Researcher*

Dalle Molle Institute for Artificial Intelligence (IDSIA), SUPSI

SIB Group: Machine Learning for Bioinformatics and Personalised Medicine

29 August 2025 — Lugano

`https://github.com/sib-swiss/federated-learning-training`

Overview

These practicals emulate, in simplified form, a realistic collaborative workflow. Seven institutions worldwide each hold pancreas single-cell datasets, with each dataset generated using a different sequencing technology. They collaborate to train an open-source model for technology-related batch-effect removal, using an architecture provided by **scVI-tools**, and plan to disseminate it through a journal publication. Since data cannot be shared across sites, the partners adopt **Federated Learning (FL)** with the **Flower** framework, allowing training to occur locally while only model updates are exchanged.
A separate institution, which also has pancreas single-cell data, later reads the publication and wishes to use the released model. This is particularly useful for them because their study contains measurements from two different technologies (one machine failed mid-study and was replaced), and technology-related batch effects are present. They therefore need to remove these batch effects to perform several downstream tasks.

Plan for the practical

[Morning] Exercise 1. We will perform a minimal FL workflow to become familiar with Flower and the mechanics of a federated process. Concretely, we will compute a *weighted mean of the per-site gene variances* to identify the most variable genes. This is a didactic simplification (not the exact method used in practice), chosen to make the federated pipeline transparent. We will demonstrate that running this computation on centralized data and running it in a federated manner produce the same result.

[Afternoon] Exercise 2. We will train a deep-learning scVI model in a federated setting, and also train the same model in a centralized manner. Finally, we will compare the effectiveness of the federated and centralized approaches on data from a new (previously unseen) institution.

[Afternoon] Exercise 3. We will conduct a live federated training in small groups using the code you have developed: one participant will act as the server, and the others as clients.

Exercise 1 — Federated variance: a warm-up

Goal

We will familiarize ourselves with Flower and the mechanics of a federated workflow by implementing a minimal end-to-end project. Instead of training a deep model, we will compute a *weighted mean of per-site gene variances* across institutions, which is a didactic simplification of highly variable gene (HVG) selection.

Prerequisites

- The conda environment `fl-course-env` created before the course.
- The Flower app in `exercise-variance/` folder.

Steps

1. **Conda environment.** Activate the conda environment `fl-course-env` with `conda activate fl-course-env`.
2. **Centralized analysis.** Run the `centralized_analysis.ipynb` notebook to familiarize yourself with the `AnnData` object type. This script downloads the necessary dataset, creates training/validation/test splits, and saves a list of all genes (non-sensitive information), that is a necessary prerequisite for the subsequent federated learning task. In this exercise we focus on the training set, which will later be distributed across clients. In the notebook, you find seven questions to answer.
3. **Inspect the Flower app.** Review the structure of the `exercise-variance` folder:
   - `pyproject.toml`: configures the Flower app.
   - `app/server_app.py`: defines the server-side logic.
     - Reads configuration values (e.g., number of rounds, gene list file).
     - Loads a JSON list of gene names.
     - Instantiates the `FedAvg` aggregation strategy (performing weighted average), using dummy initial parameters.
     - Builds a `ServerApp` object that orchestrates the training rounds.
   - `app/client_app.py`: defines the client-side logic.
     - Implements a custom `NumPyClient`.
     - For this toy example, the client returns a random integer and a random weight in the `fit` method.
     - Wraps the client in a `ClientApp`.
   - `app/task.py`: contains helper functions such as `get_dummy_start` (dummy initial parameters), `get_random_integer`, and data-loading utilities to be used later.

In this minimal example, no real model is trained. Instead, clients generate random integer and random weights, and the server aggregates them using FedAvg, calculating a weighted average. This setup provides a simple, working skeleton that you will gradually adapt to compute gene variances.

4. **Run your first FL project.** In a terminal, navigate to `exercise-variance/` and execute:

   ```
   flwr run .
   ```

   Observe the log output of the federated process.

5. **Save results with a custom strategy.** Flower provides several aggregation strategies for the parameters received by the server, such as the widely used FedAvg, implemented in the Python class `FedAvg`. By default, `FedAvg` cannot save aggregated parameters to disk, but Flower allows you to define custom strategies.

   To see how this works, open `custom_strategy.py`, where the class `FedAvgWithModelSaving` is defined. This new class inherits from `FedAvg` and extends it with methods that save the aggregated parameters as a JSON file. Import `FedAvgWithModelSaving` in `server_app.py`. Inside the function `server_fn`, uncomment the line retrieving the results file name (as `results_file_path`) from the run configuration, and replace `strategy = FedAvg(...)` by `strategy = FedAvgWithModelSaving(..., results_file_path=results_file_path)`.

   Re-run the project and verify that a `top2000_genes_federated.json` file is created containing the aggregated results from the federated process. (Note: the file name is already the final one, but at this stage it does not yet contain the list of genes.)

6. **Partition the dataset by technology.** We can now move to the use of the actual single-cell data. In simulation mode, multiple clients run on the same machine. We read the full dataset and partition it so that each client receives cells from only one sequencing technology. Uncomment the relevant code in `client_app.py` during client initialization, then re-run the project. In the terminal, you should see a client reporting its ID and the sample size (this message appears only once, since Flower suppresses repeated outputs).

7. **Compute variances.** Extend the `FlowerClient` class in `client_app.py` so that each client loads its own local data during initialization (for example by calling `load_local_data` inside `__init__`). Next, modify the `fit` function: instead of returning random integers, compute the variance of the client's local data (one variance value per gene) and store the result in a NumPy array. Return this array (wrapped in a list, as in the provided example) together with the client's sample size as the weight, so that `FedAvg` aggregates the local variances in proportion to dataset size. Re-run the simulation and check that the results file now contains the weighted average variance for all genes.

8. **Filter to the top 2000 genes.** Extend the custom strategy so that it saves a list with the names of the 2000 most variable genes. The necessary code is already included in `custom_strategy.py`; uncomment it, and adjust the `evaluate` function as needed. Re-run the project to generate the final version of the results file.

9. **Compare centralized and federated results.** Now that you have both the centralized and federated versions of the high-variance genes list, open and run the notebook `results_comparison.ipynb`. Verify that the two outputs are identical.

Exercise 2 — Federated scVI: training, logging, and model release

Goal

We will train scVI in a federated simulation using Flower, save the final aggregated model, log the losses per round and compare against a centralized baseline.

Prerequisites

- The conda environment `fl-course-env` created before the course.
- The course repository with the `exercise-scvi` project.

Steps

1. **Check the inputs.** In the `data_input` folder there are two files: one contains a list of highly variable genes (similar to the one you generated in the morning, but produced with the proper method), and the other contains the list of technologies used in the training. Both of these inputs are required for the process.

2. **Run the centralized baseline.** Open the `centralized_training.ipynb` notebook and run it. This recreates the morning's datasets (train, validation, test) and trains an scVI model on the centralized training set, saving the model. Skim the code to familiarize yourself with scVI syntax and conventions.

3. **Inspect and run the federated project.** Navigate to `exercise-scvi` and launch the Flower simulation:

   ```
   flwr run .
   ```

   Review the code structure:

   - `server_app.py`: is similar to the morning project. A *dummy* `AnnData` (based on HVGs and batches) is created to initialize shapes and registry, then an scVI model is built from it. The model's initial weights are passed to Flower's `FedAvg` as the starting point for training.
   - `client_app.py`: check `client_fn` to see how it loads both training and validation data, restricting them to the highly variable genes, and creates an scVI model.
   - `custom_strategy.py`: provides you with a custom strategy which saves the federated model, analogous to the one that you used in the morning, together with other more complicated custom strategies to be used during this exercise.

     In the client class, read `fit` (loads global parameters each round, trains locally) and `evaluate` (computes losses each round). Note how the functions `get_weights` and `set_weights` are used.

   At the end of the process, the console prints the round losses (by default, a single value taken from the first output of the `evaluate` function in the `ScviClient` class).

4. **Store and plot the loss (strategy).** We introduce a custom strategy that, in addition to saving the model, performs additional tasks. In the file `custom_strategy.py`, use the class `StoreLossSaveOnFinalFedAvg` to record per-round loss values and optionally plot or export them. Import this strategy in `server_app.py` and follow the inline instructions to enable loss tracking. This will create a plot with the loss across rounds. Inspect the plot.

5. **Log both train and validation losses.** As mentioned, by default, FedAvg consumes only the primary scalar returned by `evaluate` (which is usually the loss on the validation set). To track both training and validation losses, switch to the provided `StoreBothLossesSaveOnFinalFedAvg` strategy and update `server_app.py` accordingly. This will use both metrics to produce the loss plot. The aim of this exercise is to show how to deal with custom metrics in addition to the default one.

6. **Enable privacy-preserving training.** To use secure aggregation (SecAgg+), uncomment the relevant section at the end of `server_app.py` and modify the last line of `client_app.py` as suggested in the comment there. Ensure any required backend configuration is in place.

7. **Experiment with different numbers of local epochs and federated rounds.** Keep the total number of local epochs fixed at 50, but vary how they are distributed between local training and federated rounds in the configuration file. You may want to do this exercise in groups, running a setting configuration each person, in order to save time. For example, try:

   - 1 local epoch and 50 rounds,
   - 5 local epochs and 10 rounds,
   - 10 local epochs and 5 rounds.

   What trends do you observe?

   - In terms of computational time?
   - In terms of the loss behaviour?

   Feel free to discuss your observations with an instructor.

8. **Compare centralized and federated models.** Open `model_comparison.ipynb` and visualize the embeddings/UMAPs produced by both models. You should observe a substantial reduction of batch effects in both cases. The comparison between the two models, of course, rather than a visual comparison would require a thorough quantitative comparison, but this is beyond the scope of this practical.

Exercise 3 — Live federated run

Goal

We will execute a real multi-client federated training session: one server coordinates several clients, each holding a single-technology subset.

In this exercise you will all connect to the same Wi-Fi network. In a real-world federated learning project, the server and clients would typically be connected over different networks. What we demonstrate here is still a truly federated process, but in a simplified setup.

For details on running a production deployment, please refer to the Flower documentation.

Setup

1. **Form groups.** Split into two groups of comparable size.
2. **Assign roles.** In each group, decide who will act as the *server*; all other participants will be *clients*. Assign each client a unique integer ID, `YOUR_ID`, ranging from 0 up to the number of participants (`NUM_CLIENTS`) minus one.

Server steps

a) **Prepare the project.** In `exercise-real/`, remove the `data_download.ipynb` notebook, as this is only needed on the client side. Next, copy the following items from Exercise 2 into this folder:

   - the `app/` folder,
   - the file `pyproject.toml`.

   Finally, modify the client app replacing the function `load_data_simulation` by the function `load_data_deployment` (remember to import it, note it appears twice, and note it does not take the argument `partition_id`). The new function loads the local data directly, without performing any partitioning.

b) **Configure and launch.**

   i. Connect to the Wi-Fi network provided by the instructors. We will actually be using a hotspot, since the campus Wi-Fi would block our federated process.

   ii. Edit `pyproject.toml` and add the following at the end:

   ```
   [tool.flwr.federations.local-deployment]
   address = "127.0.0.1:9093"
   insecure = true
   ```

   iii. Open a terminal and activate the course virtual environment:

   ```
   $ conda activate fl-course-env
   ```

   Then start the SuperLink service:

   ```
   $ flower-superlink --insecure
   ```

The flag `-insecure` disables encrypted communication. This is acceptable for our toy example but would not be appropriate in a real-world production setup.

iv. Ask the clients to run their connection command to join the SuperLink. Show the server logs in your terminal so participants can see when each client connects.

v. Open another terminal, navigate to `exercise-real/` and activate the course virtual environment:

```
$ conda activate fl-course-env
```

Tehn start the federated process:

```
$ flwr run . local-deployment --stream
```

While the model is training, display the loss plot updating in real time so participants can follow the process.

Client steps

a) **Prepare local data.**

i. In the `data_input` folder you will find both the HVG list file and the batch list file that you used before.

ii. Open the notebook `data_download.ipynb`, set the variable `YOUR_ID` at the top, and run it. This will create a local dataset containing only one technology. Once you are done, you may remove the notebook if you wish.

b) **Run the client.** Start the client process so it can connect to the server and participate in training.

i. Connect to the Wi-Fi network provided by the instructors. We will actually be using a hotspot, since the campus Wi-Fi would block our federated process.

ii. Compute your port as `YOUR_PORT = 9094 + YOUR_ID` (for example, if `YOUR_ID = 0`, then `YOUR_PORT = 9094`; if `YOUR_ID = 1`, then `YOUR_PORT = 9095`, and so on).

iii. Open a terminal, navigate to `exercise-real/`, and and activate the course environment:

```
$ conda activate fl-course-env
```

Launch the client using the following command (replace the placeholders with your values), one instructor will provide you with the `SUPERLINK_IP`:

```
$ flower-supernode \
    --insecure \
    --superlink <SUPERLINK_IP>:9092 \
    --clientappio-api-address 127.0.0.1:<YOUR_PORT> \
    --node-config "partition-id=<YOUR_ID> num-partitions=<NUM_CLIENTS>"
```