Federated Learning in Healthcare: A roadmap and resource for clinicians

Supplemental Playbook

2022

1 Introduction

This document is the companion to our recent paper on federated learning (FL) for clinical research collaboration. We suggest reading our publication for additional background information regarding FL (and its biomedical application) in general, we hope to inspire and guide others through practical research using federated learning through this easy-to-follow playbook written for clinicians and other individuals not formally trained in computer science.

1.1 What is federated learning?

Briefly, federated learning is a novel approach within machine learning that does not require the traditional aggregation of data into a central repository. Instead, FL distributes the model (such a neural network) to various participating sites ('clients'). The model then learns from data within each client's local repository through any number k epochs of local training, after which the *learned weights* of the model are returned to the central site ('server'). The server then aggregates the weights from each client, commonly through averaging, into one global model and redistributes the updated weights to the clients for any number n of additional training rounds (see Figure 1).

1.2 Why is FL useful for clinical research?

Healthcare data traditionally exist in institutional silos, which is in part due to the sensitive nature of patient data that demands high standards of privacy and oversight. Any collaborative project requiring lots of data therefore requires both complicated medicolegal data sharing agreements and the commitment of one institution to manage (and pay) for a central repository. Moreover, data can be withheld for business reasons, thereby disincentivizing collaboration between potential competitors. FL promises to lower the barrier-to-entry for multi-center projects and facilitate privacy-preserving collaboration (without requiring data sharing).

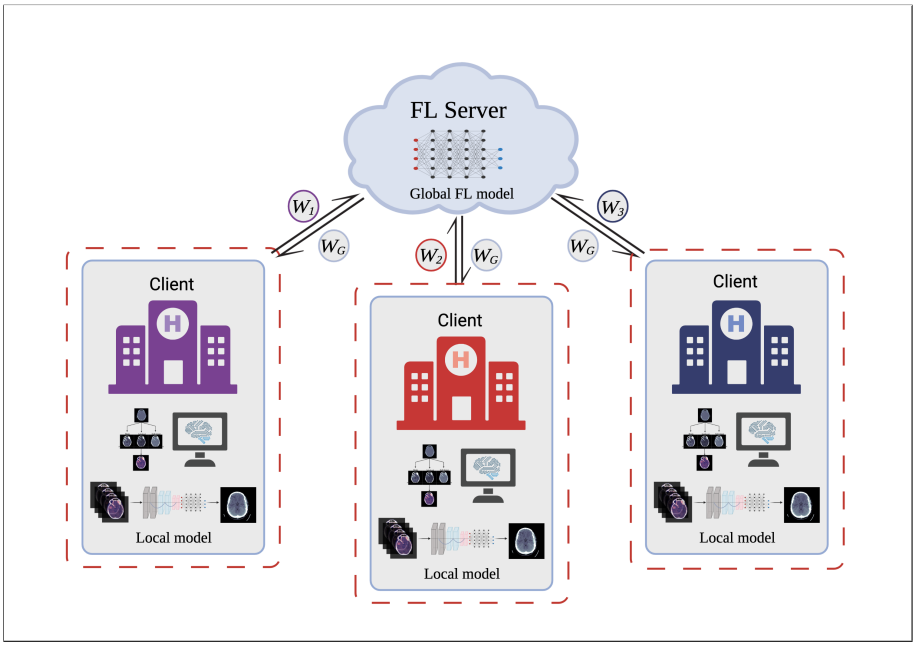

Figure 1: Schematic of FL with three clients and one aggregating server

1.3 How to use this document

This document is intended to provide a roadmap and practical guide for those wishing to set up a FL network for scientific pursuits. While many published FL solutions require sophisticated computational teams working with proprietary software, we have used an open-source software development toolkit *nvflare* created by NVIDIA (NVIDIA Corporation) and created a publicly available GitHub (GitHub, Inc.) repository for intracranial hemorrhage detection that can serve as a modifiable template for others to build upon. Each section of this document outlines the requisite skills, hardware, and steps needed to successfully establish a federated network. Section 2 describes the specific hardware considerations prior to conducting any machine learning experiments. Section 3 offers a basic tutorial and suggested readings for working with Linux-based systems and the command line, of which minimal working knowledge is necessary. Section 4 discusses the structure of *nvflare* for open-source FL and how to practically set up a federated network. Finally, Section 5 describes the steps that went into setting up the FL network for intracranial hemorrhage detection for reproducibility and inspiration for additional projects. While we offer specific instructions and example code where necessary, we also refer to external online (and free!) resources or documentation for certain topics.

2 Hardware and GPU Decisions

Deep learning as a field has blossomed over the past decade due to the development of graphics processing units (GPUs) that allow for massively parallel computing of the matrix multiplications that underpin most deep neural networks. While deep learning frameworks, including PyTorch (Meta Platforms, Inc.), can utilize central processing units (CPUs) or application-specific integrated circuits (ASICs) like Google's (Alphabet Inc.) Tensor Processing Units (TPUs), the most common solution is GPUs running on top of NVIDIA's CUDA computing platform to make GPU-accelerated training for deep learning models easily accessible.

NVIDIA's consumer-grade GPUs serve as a relatively low-cost entry point for deep learning projects. As we learned in the survey of our end-users in this guides companion manuscript, most of our end-users elected to purchase consumer grade GPUs in local workstations running on their organizational networks. When considering investing in purchasing GPU and local workstations, one of the most important factors is GPU memory and local memory. More GPU memory will enable training larger models, with larger inputs (think MRI images), and training with larger mini-batches (thereby training faster). The addition of multiple GPUs in a single workstation, particularly when connected with nVlink which provides a high throughput alternative to PCIe, can allow for additional parallelization and drastically cut down on training times.

An appealing alternative to investing in local workstation with GPUs however is utilizing cloud-computing platforms such as Amazon's (Amazon.com, Inc.) Amazon Web Services (AWS) or Google Cloud Platform. These services allow for quick set up and scaling of compute resources. For example, an Amazon EC2 P3 instance can be set up with up to eight V100 GPUs (a data center-grade GPU with 32GB of GPU memory) and terabytes of solid-state memory storage for quick read and write speeds in a manner of minutes. It should be noted, however, that in the medical use case institutional review boards (IRBs) and institutional policies typically require that all health data stored on cloud-computing platforms are de-identified to remain HIPAA complaint. Furthermore, the cost of cloud-computing platforms is often based on the amount and time of resource allocation rather than direct utilization (although spot instances alleviate this concern somewhat) therefore for long running projects these resources can be substantially more expensive than on-premises machines in the long run.

2.1 Basic Setup

Installation instructions for the most recent version of CUDA Toolkit can be found on NVIDIA Developer's website. Here you can select your operating system, architecture, distribution, and version, and get a series of commands to install the CUDA Toolkit. For example, a Linux sytsem running Ubuntu 18.04 with x86_64 architecture can be installed with the following commands:

```
wget https://developer.download.NVIDIA.com/compute/cuda/
    repos/ubuntu1804/x86_64/cuda-ubuntu1804.pin

sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-
    repository-pin-600

wget https://developer.download.NVIDIA.com/compute/cuda
    /11.7.0/local_installers/cuda-repo-ubuntu1804-11-7-
    local_11.7.0-515.43.04-1_amd64.deb

sudo dpkg -i cuda-repo-ubuntu1804-11-7-local_11
    .7.0-515.43.04-1_amd64.deb

sudo cp /var/cuda-repo-ubuntu1804-11-7-local/cuda-*-
    keyring.gpg /usr/share/keyrings/

sudo apt-get update

sudo apt-get -y install cuda
```

3 Basics of the command line

Bash is a Unix shell, which serves as a command-line interpreter for kernel-like commands. If you are not familiar with Bash, please launch terminal and try out the following commands for starters:

```
pwd # print current directory
mkdir # create a new directory
ls # list contents of directory
cd # change to directory
rm # remove file or remove directory with -r option
cat # list contents of file
less # scroll through contents of file
head # start of file
tail # end of file
nano # edit file
mv # move file
cp # copy file
scp # secure copy or directory with -r option
NVIDIA-smi # check GPU memory available, if using GPU
```

For a nice reference and more inclusive list, please check out this repository: The Art of Command Line. Most Linux distributions come with Python installed, and PyTorch installation instructions can be found on PyTorch's website.

4 Utilizing *nvflare* to set up a federated network

Now that we have established the prerequisite skills and technical capabilities, let's next discuss the practical elements of setting up a federated learning network using NVIDIA's Federated Learning Application Runtime Environment (*nvflare*). Please see the linked documentation from NVIDIA for more in-depth information.

4.1 Nodes of a federated network

There are essentially three types of nodes of a FL network when using *nvflare*: **server**, **client**, and **admin**. The **server**, as mentioned in the introduction, allows communication across the network and performs model weight aggregation. A **client** is a single site that contains its own dataset and, once connected, frequently pings the **server** for any 'tasks' to perform, such as local training. After k epochs of local training, the **client** sends its local model's weights to the **server** for aggregation into the global FL model. The **admin** is usually run by one coordinating institution and can check on the status of **clients** and initiate or abort training (Figure 2). The **admin** is responsible for uploading a customized codebase containing the model and validation scripts.

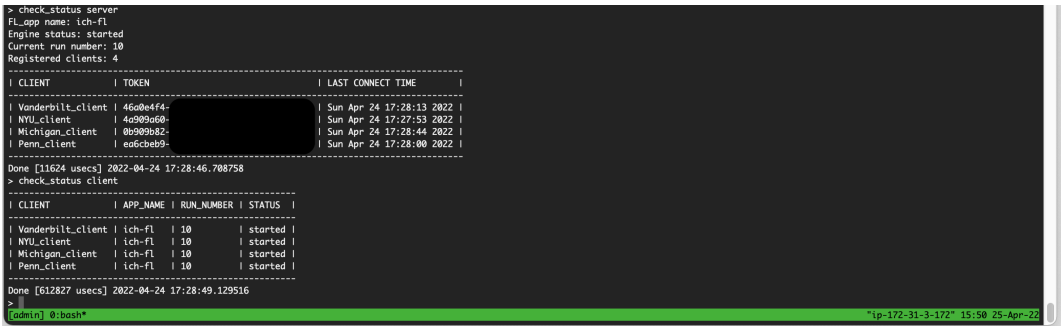

Figure 2: Sample screenshot of the **admin** checking on the status of the **server** and **clients**

4.2 Installing *nvflare* in a virtual environment

Relevant links for this section:
`https://NVIDIA.github.io/NVFlare/installation.html`
`https://docs.python.org/3.8/library/venv.html`

A virtual environment is essentially a partitioning of a computer for (in this case) Python to run in, so that various packages, libraries, and software can be isolated. This prevents issues that can arise from using different versions

of software and provides a tidy, project-specific place to conduct computer science experiments. It is important to first set up a virtual environment before installing packages like *nvflare* and other machine learning libraries. A virtual environment (and installation of *nvflare*) is necessary for each **client**, the **server**, and **admin** and should be set up on each respective computer. (Note: see Section 1 regarding Python 3 installation if this has not already been done.)

1. Run the following code in the command line (in whichever folder/working directory desired by using the relevant 'cd' and 'mkdir' commands from Section 3) to install Python's venv package:

```
$ sudo apt update
$ sudo apt-get install python3-venv
$ python3 -m venv nvflare-env
```

2. The folder **nvflare-env** should appear in the current working directory. To activate the virtual environment (which basically redirects any subsequent installations of packages, libraries, or software into *nvflare-env* rather than another location on your computer), run the following in the command line:

```
$ source nvflare-env/bin/activate
```

You will then see (nvflare-env) at the start of each command line prompt, indicating you are working within the virtual environment. Nice work!

```
(nvflare-env) $ ls
```

3. Finally, it is necessary to install *nvflare* using the following:

```
(nvflare-env) $ python3 -m pip install nvflare
```

(optional) 4. While not necessary for the client sites, one can also clone the GitHub repository containing source code for *nvflare*.

```
(nvflare-env) $ git clone https://github.com/NVIDIA/
    NVFlare.git
```

This will download example code that one can run using *nvflare*'s proof-of-concept ('POC') function that allows researchers to prototype new code on a local machine, essentially simulating a federated learning network with any number of "clients" on one's own singular machine (Figure 3). To run, simply follow the instructions on this link or follow the steps below.

```
(nvflare-env) $ poc -n 2
```

Once the POC network has been set up, one can open several Terminal (command line) windows to 'simulate' the steps necessary to connect to the FL network and 'see' from the perspective of each node. For example, by running each of the following lines (in four separate, respective terminal windows running within the nvflare-env virtual environment) one can see the status of each client as they connect to the server (Figure 4):

Figure 3: Sample screenshot after running the poc command creating a local FL network with two client sites

```
(nvflare-env) $ ./poc/server/startup/start.sh
(nvflare-env) $ ./poc/site-1/startup/start.sh
(nvflare-env) $ ./poc/site-2/startup/start.sh
(nvflare-env) $ ./poc/admin/startup/fl_admin.sh
```

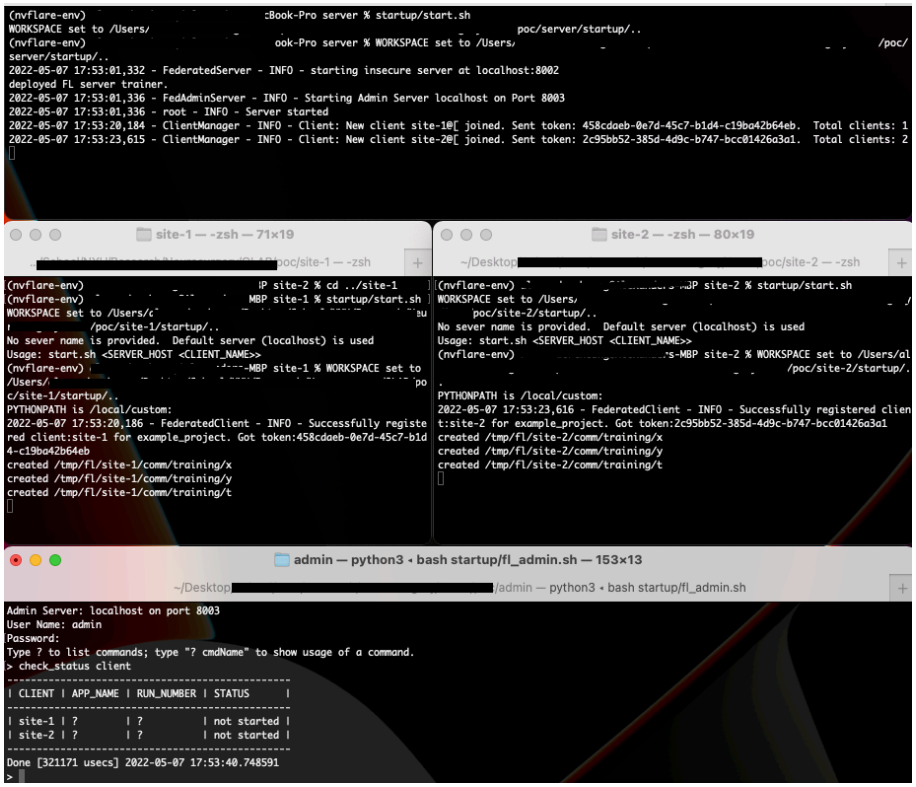

Figure 4: Sample screenshot of the POC running in four terminal windows simulating a FL network. Moving clockwise from the top: **server**, **site-2 client**, **admin**, and **site-1 client**

Please refer to `https://NVIDIA.github.io/NVFlare/example_applications.html` for example applications using the *nvflare* POC functionality.

4.3 Provisioning *nvflare* to collaborators

Relevant links for this section: `https://NVIDIA.github.io/NVFlare/user_guide/provisioning_tool.html`
For the coordinating site (eg, the institution or group usually responsible for running the **admin** and **server** as well as writing the main codebase for training/validation), *nvflare* has an incredibly useful provisioning tool that comes with a straightforward user interface (UI). The provisioning tool generates a .yml file, which is used to then create several .zip files containing the necessary scripts to startup each FL node and ultimately connect to the federated network.

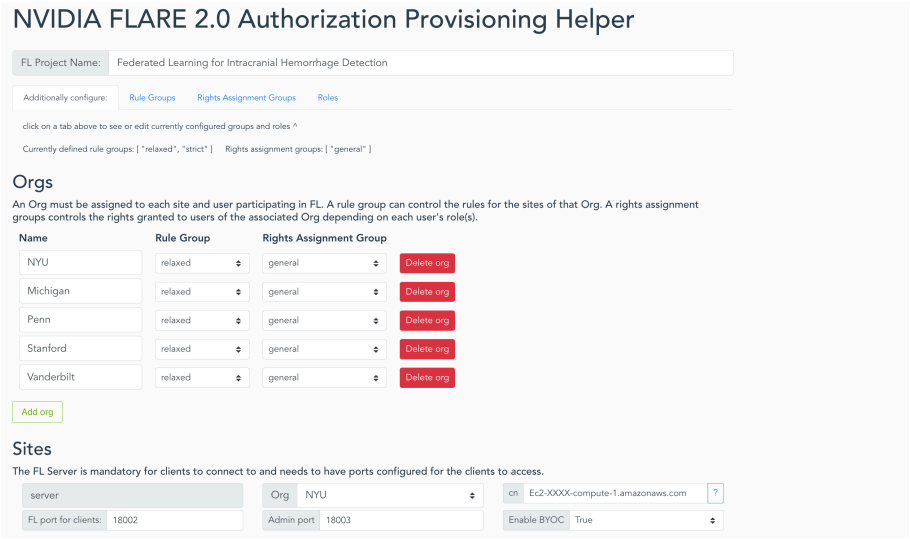

Figure 5: Sample screenshot of the provisioning tool UI

The coordinating site (after activating the nvflare-env virtual environment – see section above) can use the command line to open the provisioning UI in the user's default browser (Figure 5).

```
(nvflare-env) $ provision -u
```

Within the provisioning tool UI, one can specify the organizations participating in the FL. In this example, there were five organizations participating as **clients**. One **client** (NYU) also served as the coordinating center, and therefore managed the **admin** and **server**. The server was hosted in an AWS EC2 p3.2x large instance. We specified the ports as 18002 and 18003 for the clients and admin, respectively, and specified the IP address (eg, EC2-XXXX-comput-1.amazonaws.com) of the EC2 instance (Figure 6). These ensure the clients and admin are able to locate and connect with the central server.

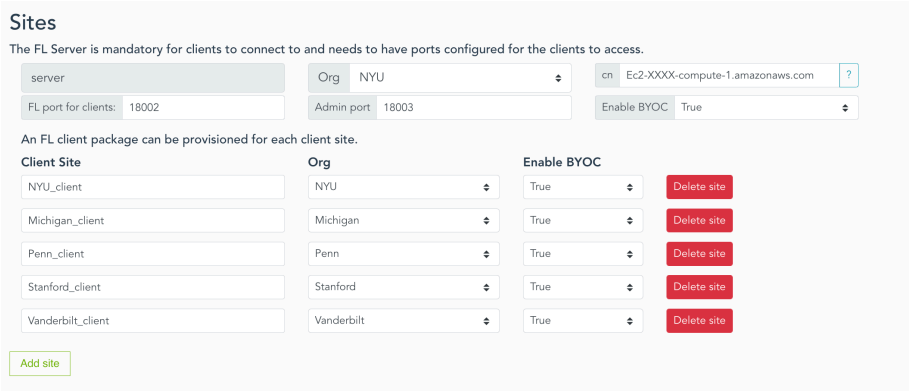

Figure 6: Provisioning tool UI specifying the client names and server location.

Lastly, the 'Users' section specifies who has admin privileges (which in this case was NYU). We recommend using the institutional email addresses for the admins as they will need this to eventually to access and control the federated network. Once all client, server, and admin information has been filled out, click 'Generate configuration file' and 'Download project.yml' (Figure 7). This will download the .yml file necessary for *nvflare*'s provision command to read and generate the necessary files (Figure 8).

Once you have the project.yml file downloaded and moved to your command line's current working directory, run the command:

```
(nvflare-env) $ provision -p project.yml
```

This will generate a folder titled 'workspace' in the same directory as the project.yml file that contains the .zip files necessary to startup the FL client, admin, and server (Figure 9). The provision command will also print out a list of passwords necessary for each user to unzip the package (Figure 10). We recommend copying and saving the password lists into a separate, secure text file. Each password and zip file can then be emailed (or otherwise shared) to each respective user.

4.4 Setting up the server

After provisioning, each participant is required to start up their own respective node of the FL network, whether that be the server, admin, or a client. Since the server is the central hub for the network, we recommend having the coordinating site set up the server immediately so clients do not run into errors when trying to connect.

As mentioned previously, we hosted the server in an AWS EC2 instance. Simply

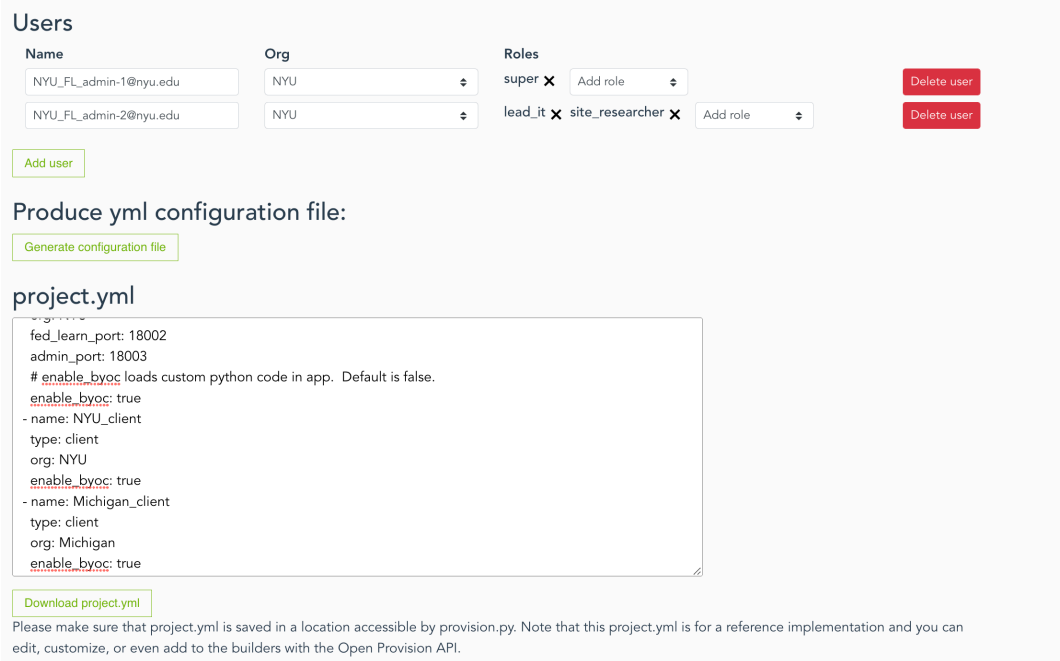

Figure 7: Provisioning tool UI specifying the admins

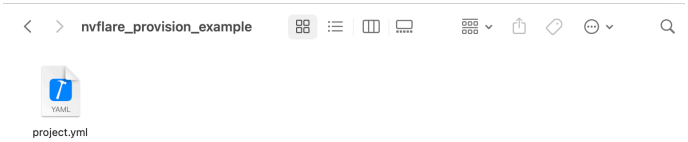

Figure 8: project.yml file in the working directory before provision command

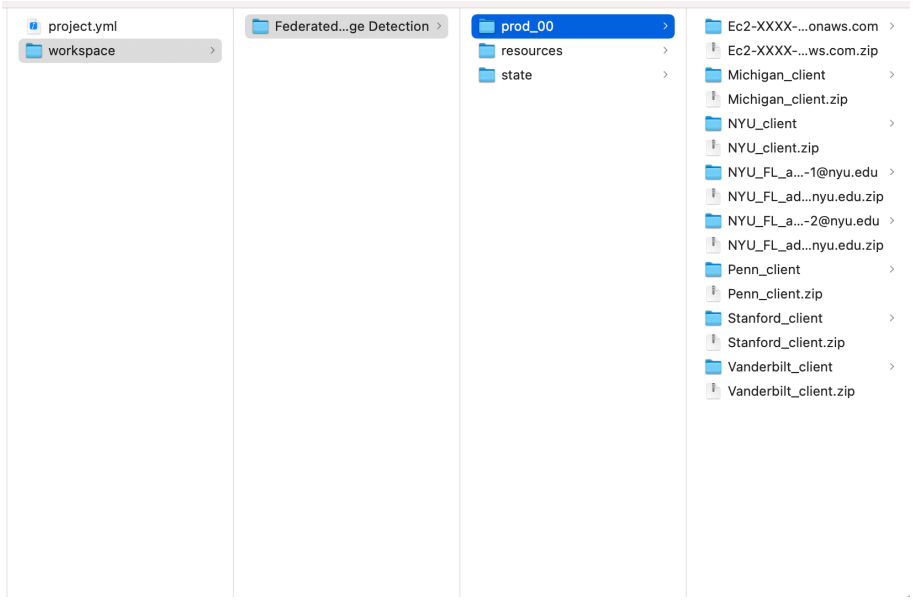

Figure 9: Newly generated files in the working directory after provision command

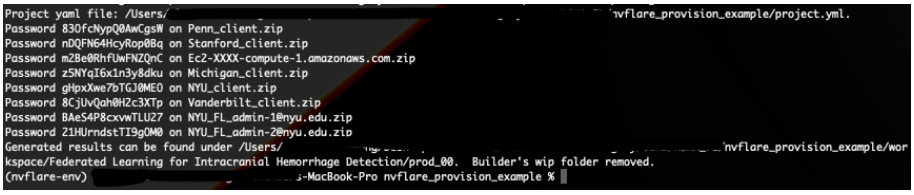

Figure 10: Passwords generated for each file after provision command

upload the .zip file for the server (eg, Ec2-XXXX-compute-1.amazonaws.com.zip) into the desired directory on the instance. Unzip the file and input the password when prompted on the command line (Figure 11) which will then unzip and produce the files necessary for the FL server in a folder titled startup (Figure 12).

```
(nvflare-env) $ unzip Ec2-XXXX-compute-1.amazonaws.com.
    zip
```

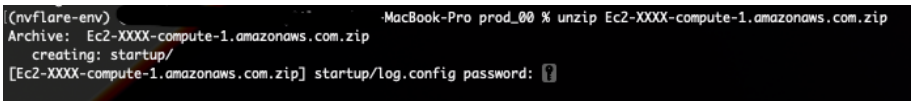

Figure 11: Password prompt when unzipping the server file on AWS

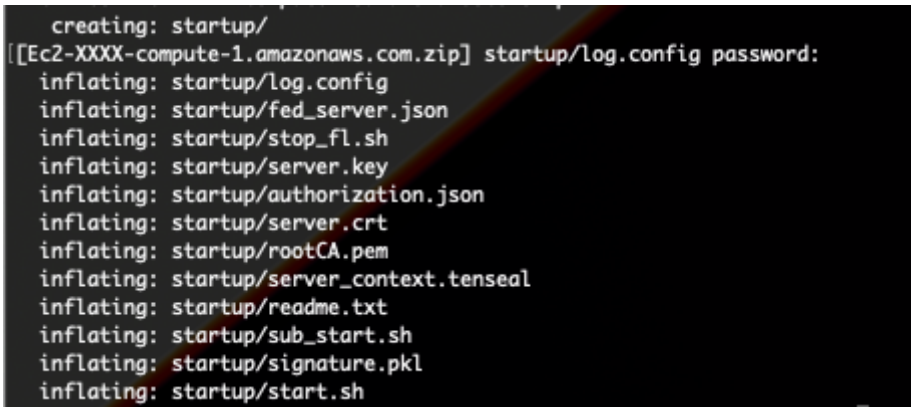

Figure 12: Unzipping decompresses files necessary to run the FL server in a folder titled startup/

Finally, to start the server, simply run the start.sh file. (It is important to remember to include the file path before the script itself in order to successfully run a bash script – so if you are within the startup folder as your current working directory, simply run ‘./start.sh‘. Otherwise, specify the path to the script as well, such as below.) It is also important to NOT MOVE OR RENAME THE FOLDERS/FILES PRODUCED AFTER UNZIPPING as this may cause errors in the future.

```
(nvflare-env) $ startup/start.sh
```

Voila! The server should now be up and running in the AWS EC2 instance and ready for clients to connect (via the 18002 port) or the admin to check in and begin training (via the 18003 port). No additional steps are needed at

this point to maintain the server, but one can access the log.txt file to monitor activity as necessary.

4.5 Setting up the client(s)

At this point, the server should be up and running and each client site should have their respective .zip file and separately emailed password. Following similar steps above for setting up the server, each client site should place the .zip file in the desired folder (named something like 'FL-client') on a GPU-enabled machine. Unzipping the file (eg, NYU's client file in this case) will again prompt users for the password, which can be copy-pasted into the command line. This will create the files necessary to start and run the FL client (Figure 13).

```
(nvflare-env) $ unzip NYU_client.zip
```

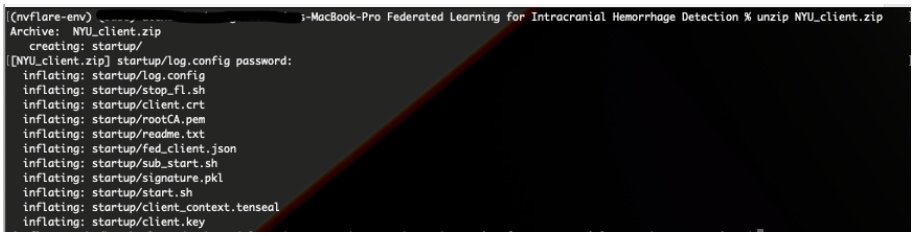

Figure 13: Unzipping decompresses files necessary to run the FL client in a folder titled startup/

Finally, to start the client and connect to the server, simply run the **start.sh** file. (It is important to remember to include the file path before the script itself in order to successfully run a bash script – so if you are within the startup folder as your current working directory, simply run './start.sh'. Otherwise, specify the path to the script as well, such as below.) It is also important to NOT MOVE OR RENAME THE FOLDERS/FILES PRODUCED AFTER UNZIPPING as this may cause errors in the future.

```
(nvflare-env) $ startup/start.sh
```

The client should now connect with the server, which will generate a unique token for that client site. No additional steps are required on the client side in terms of network management once it has been connected to the server, as long as the connection is not closed or disrupted in any way. To disconnect from the server, simply run the stop.sh script, which should stop/kill the process responsible for constantly pinging the server for tasks. If the client does disconnect for some reason (power outages, etc.), simply run the stop.sh script to ensure the prior connection/process/token has been removed from the local system, and

re-run the start.sh script to re-connect as before.

The only additional step that may be required after connection is installation of packages relevant to the specific project in the virtual environment. We recommend that the coordinating site email a **requirements.txt** package to each client site, which can be used with the following command (when in the nvflare-env virtual environment) to install the necessary packages (Figure 14). In this particular project, several routine Python packages (eg, pandas, numpy) were required, as well as torch and torchvision for the specific neural network used.

Lastly, the client should ensure that their data are in a folder location that has been standardized across the FL network. The coordinating site should communicate directly with all clients regarding this location, so that the FL application 'knows' where to look each time it starts training. We suggest creating an 'input' folder in the same directory level as the startup/ folder, since this defaults as the working directory when training through *nvflare*, which makes it straightforward to standardize reading in data when writing the custom training scripts.

```
(nvflare-env) $ pip install -r requirements
```

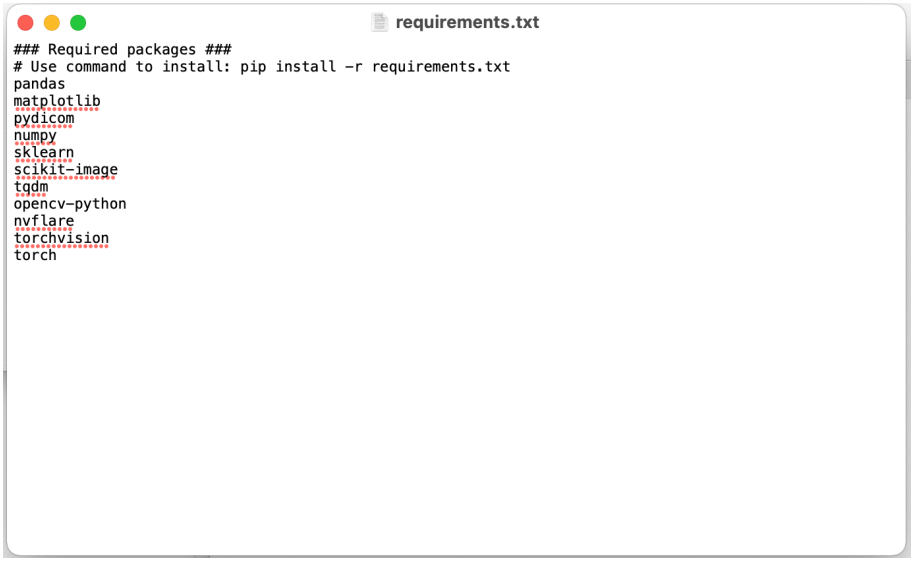

Figure 14: requirements.txt package are useful to ensure all clients have the packages necessary for the training scripts

4.6 Setting up the admin

Similar to the client and server steps above, setting up the admin requires moving the .zip file to a specified directory and unzipping it, again copy-pasting the password into the command line when prompted. This will decompress the files necessary to run the admin (Figure 15). Once unzipped, run the **fl_admin.sh** bash script to connect to the server. Unlike the server and clients, in this case, the command line will then show a prompt requesting the User Name. This will be the email address used for that particular user (eg, NYU_FL_admin-1@nyu.edu). Once connected, the admin can check on the status of the server and the clients to see who has connected or determine if training is continuing or has stopped (Figure 16).

It should be noted as well that, while the server and clients do not require additional maintenance after initial setup, the admin does disconnect after a period of time. When this occurs, simply run the command **bye** in the command line and then reconnect using the same **startup/fl_admin.sh** command as before.

```
(nvflare-env) $ startup/fl_admin.sh
```

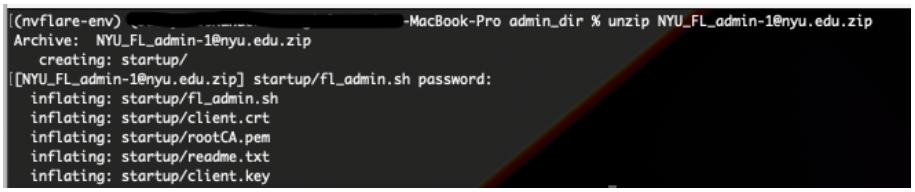

Figure 15: Unzipping creates a startup/ folder containing the files necessary to start the admin

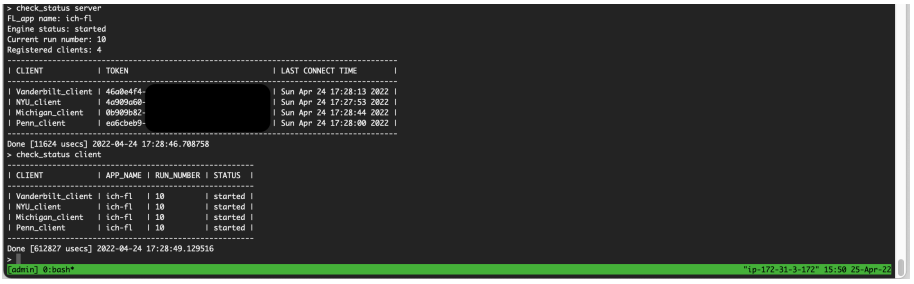

Figure 16: The admin, once connected to the server, can check in on the netowrk using various commands

Finally, create a folder titled **transfer/** in the admin folder at the same directory level as the **startup/** folder using the 'mkdir' command (Figure 17).

The admin will load any custom scripts used for training and federated learning into this transfer folder, which then gets uploaded to the server and ultimately deployed to the clients.

```
(nvflare-env) $ mkdir transfer
```

Figure 17: Create a transfer/ folder at the same level as startup/ to upload custom scripts for federated learning

In our experience, the most common admin commands used were the following:

check_status client (to check on the status of clients and state of training)

set_run_number N (creates a new folder titled run_N at the server and each client site for different experiments/debugging)

upload_app APP_NAME (uploads the folder titled APP_NAME containing custom training scripts from transfer/ folder to the server)

deploy_app APP_NAME (deploys the APP_NAME folder to clients)

start_app all (starts training for all clients)

abort client (stops a client training process prematurely)

abort server (stops a server training process)

See the official documentation for more information regarding the admin commands: `https://NVIDIA.github.io/NVFlare/user_guide/admin_commands.html`

4.7 Summary

Congratulations on setting up the network for federated learning through *nvflare*! To briefly summarize the above steps, we **(1)** ensured all participants (server, admin, and clients) had a Python virtual environment set up and activated with *nvflare* installed using the pip package manager. **(2)** The coordinating site then used the provisioning UI to specify the network architecture and generate the project.yml file that ultimately creates the necessary .zip files (and associated

passwords). **(3)** After distributing the .zip files to each respective site, the process is relatively consistent between clients, admin, and server: unzip the file in its permanent location, copy-paste the password when prompted, and run start.sh (or fl_admin.sh) to connect to the network. **(4)** Some additional logistical steps to consider (although distinct from setting up the actual network) are **(4a)** installing the required packages necessary for training and **(4b)** standardizing the location of data across all clients. Approximately 80% of the work above is carried out by the coordinating site (which, in most cases, will likely manage a client, admin, and the server), and once a client is connected to the server, no additional maintenance is required on their end.

As a final note, in our experience, we found that we could not run clients from a local MacBook (Apple Inc.) during testing of the network (which we do not recommend anyway during actual training unless it has GPU capabilities), and we could not run the admin from our institution's (NYU) computing cluster. We therefore suggest the coordinating site host both the server and admin in a cloud-based server such as AWS, using **tmux** or **screen** to allow continuous and parallel access to each from the command line.

5 Applying FL to a clinical problem

The following section discusses a clinically applicable example of federated learning using *nvflare* to detect intracranial hemorrhages on computed tomography (CT) scans. Using the above sections in this document, the associated GitHub repository (`https://github.com/nyuolab/detectICH/tree/main/ich-fl`), and our recent paper, one should be able to recreate and reproduce the exact experiment since all data and code are open-source and publicly available. Once the general FL network infrastructure has been established, our hope is that other groups can work upon this template to pursue other FL experiments to solve other biomedical and healthcare problems. As much of the following has been documented extensively in the methods section of the paper or commented in the codebase, we will refer to these external documents as necessary for specific details.

While we strongly believe only minimal computer science experience (largely concerning the command line) is necessary for the client portion of federated learning with *nvflare*, the coordinating site (using the following sections) should be someone with greater proficiency in the command line, Python, and machine learning.

5.1 The Dataset: Intracranial Hemorrhage Detection

Since the main purpose of this project is to help clinicians with minimal experience in computer science establish a federated learning network, we decided to use a readily available dataset that does not contain protected health in-

formation (PHI). We therefore used the Radiological Society of North America (RSNA) Intracranial Hemorrhage (ICH) Detection dataset found on the popular data science competition website Kaggle, which contains head CT scans from thousands of patients stored as DICOM files. Each DICOM image is a single slice from a patient CT scan and has been annotated (labeled) by radiologists as containing any of several hemorrhage types. Since an image can contain multiple hemorrhage subtypes, we approached this as a multi-label problem. Please also note that since each image is a single slice of a three-dimensional scan, multiple images can come from the same patient scan. Please refer to the following link for additional details:

`https://www.kaggle.com/competitions/rsna-intracranial-hemorrhage-detection/overview`

One can download the data (approx. 500 GB) either directly from the website or via the API (see `https://www.kaggle.com/docs/api` for specific instructions). We split the data into two sets: 90% for training (approx. 677,000 images) and 10% as a hold-out set for additional testing (approx. 75,000 images). Using each image's metadata, the training set was then split into five groups by patient ID and distributed to each site.

At each client site, the data were stored at the same level as the 'startup/' directory in a folder labeled 'input/'. Within the 'input/' folder is another folder labeled 'data/' containing the actual DICOM images, as well as a file 'labels.csv' that contains the image name and diagnoses associated with the image (Figure 18).

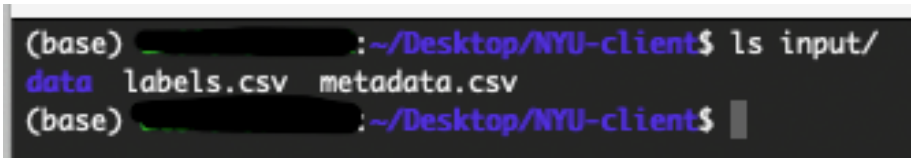

Figure 18: File structure of the data at each client

5.2 Application structure using *nvflare*

An 'application' refers to a custom set of code that an admin can deploy to the FL network (hence the commands ***upload_app***, ***deploy_app***, and ***start_app***). In this case, the application consists of scripts to train a ResNeXt-101 neural network on DICOM files for ICH detection. The app needs to be contained within one folder (eg, ich-fl). Within that folder there must be a **config/** folder and a **custom/** folder.

The **config/** folder contains JSON files for configuring the server (**config_fed_server.json**)

by customizing network parameters, such as the minimum number of clients; the number of rounds of FL to perform; and the aggregation algorithm utilized. (We simply opted for averaging the weights, but *nvflare* offers other FL algorithms as well. See the examples here for more details.) There is also a JSON file for the client (**config_fed_client.json**) that determines local training hyperparameters, such as the learning rate, number of epochs, and order of tasks (eg, train or validate) to perform. Please refer to the **ich-fl/config/** folder on the GitHub repo to see how we configured our FL network, as well as the official documentation regarding application formatting here:
`https://NVIDIA.github.io/NVFlare/user_guide/application.html`

The **custom/** folder contains all the custom code needed to train the model. This fundamentally resembles a repository for traditional centralized machine learning and contains scripts for the dataset class, model architecture, training loops, and validation steps. The main difference is that the code for training and validation are nested within an **Executor** class, which is a type of **FLComponent** for clients, that produces a **Shareable**, which is an object (such as a Python dict) that communicates between the server and client. One can read more about Executors here and Shareables here, but reading through this project's GitHub repo should provide an intuition as to how the practical implementation of these *nvflare*-specific classes work.

Beyond the training and validation scripts just mentioned, there are also **pt_model_locator.py** and **validation_json_generator.py** scripts. The prior defines the model used and is essentially boilerplate code besides the specific lines defining the model used. In our case, we defined the resnext101_32x8d model from the torchvision library, and added a fully connected layer to output six possible classes (to predict each type of intracranial hemorrhage present). **validation_json_generator.py** is another boilerplate script that creates a JSON file of the cross-site validation results to share back to the server.

We highly recommend diving into the GitHub repo to implement the **ich-fl** app and gain a practical intuition of the code. NVIDIA also provides several example applications in its *nvflare* documentation that can help to explain further some of the concepts introduced here.

`https://NVIDIA.github.io/NVFlare/example_applications.html`

5.3 Cross-site validation

Cross-site validation is the final step of a FL experiment once all rounds of training have completed. The process takes the most recent locally trained model from each client site and the most recently updated Global FL model from the central server and assesses their performance on data found at each client site. Through this process, one can get a sense of how a locally trained model or the global model performs across varying datasets. (It should be noted

that the term "locally trained" model is somewhat of a misnomer, as these models technically have weights that have been updated in the central server during prior rounds of FL.) The code for cross-site validation can be found in the **ich_validator.py** script in the **custom/** folder, and can be modified to output a dict() of whatever performance metrics desired (eg. F1 score, AUC, accuracy, etc.).

5.4 Debugging and troubleshooting

We highly recommend testing any custom code using a small prototyping dataset in the proof-of-concept (POC) mode mentioned in prior sections. This will allow rapid debugging, access to all FL network participants (eg, all client logs), and will avoid unnecessary costs for cloud-based computing if an experiment fails.

5.5 Deploying the app for federated learning

One the app has been successfully debugged and run in POC mode, the next step is to run your experiment in the actual FL network. Follow the steps above on starting up a FL network to ensure the server and clients are all connected. (You can use the admin to check the status of these.)

From the **admin** command line (after running ./fl_admin.sh), ensure the app (containing the **config/** and **custom/** folders) is in the folder named **transfer/**. Then run the following commands:

```
> upload_app APP_NAME
> set_run_number 1
> deploy_app APP_NAME all
```

This will create a folder called **run_1/** on the server and all client sites and upload the app scripts to each site as well. From here, you can check on the status of each client to ensure everyone has received the appropriate tasks (Figure 19).

```
> check_status client
```

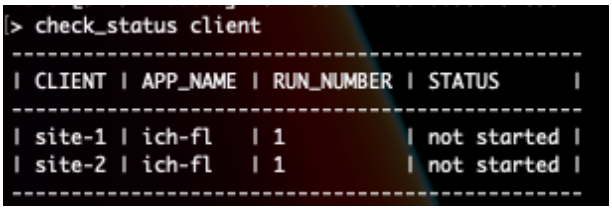

Figure 19: Using the admin to check on client sites prior to starting training (including which app, which run number, and each client's status)

The last step is to simply run the following command, which will initiate training.

```
> start_app all
```

If you follow the associated GitHub repository for ICH detection, a folder will appear at the end of training and validation in the associated run folder in the central server titled **cross_site_val/**. This folder will contain a JSON file that contains the performance metrics as specified in the validation script, where one can interrogate the performance of each model on each site's data. For example, if you have federated network consisting of three clients (and therefore three datasets at each site, plus four models including the global FL model), there should be 12 values for each metric, since each of the four models was assessed on each of the three datasets.

6 Conclusion

This document, if followed carefully alongside the associated manuscript and GitHub repository, should allow researchers to reproduce our experiment and results while also providing a scaffold and roadmap for future federated learning projects. Our hope is that this will lower the barrier to entry for individuals hoping to start a federated network of their own by utilizing open-source codebase and software. While the central coordinating site will require a more advanced computer science skill set to properly adapt machine learning libraries to a federated setting, the information and tutorials included herein should allow collaborating client sites with minimal computational abilities to easily connect to the network itself. Our vision for federated learning is to enable seamless multi-institutional collaboration to tackle medicine's most pressing and difficult problems. This is our humble contribution towards further democratizing access to its promise and capabilities.

Acknowledgements

Thanks to all the co-authors, contributors, and funding sources for this project (see associated manuscript), and to NVIDIA (NVIDIA Corporation) for creating *nvflare* and providing technical assistance. This proof-of-concept project would not be possible without the publicly available dataset from RSNA/Kaggle and the open-source commitment of Python and its associated machine learning library PyTorch (Meta Platforms, Inc.).