

The Impact of Federated Learning on Distributed Remote Sensing Archives

The repository consists of the code for Federated Learning Experiments for Remote Sensing image data using convolution neural networks. It contains the implementation of three Federated Learning models:

- [FedAVG](#)
- [FedProx](#)
- [BSP](#)

The implementation is specifically made for the multi-label *UCMerced Landuse* [dataset](#). To apply other datasets it requires some modification.

Table of Contents

- Setup Details
- Description of the files & Usage
- Data Preparation and Splitting
- Implementation of the Federated Learning Models
 - FedAVG
 - FedProx
 - BSP

Setup Details

Option 1 : using Anaconda Distribution

Python version 3.8.5

It is recommended to use conda environments when using deep learning frameworks especially GPU supported libraries as there can be clash of versions if not. A `environment.yml` file is provided in the main directory which can be used to install the environment directly using conda. On linux follow these steps:

```
wget https://repo.anaconda.com/archive/Anaconda3-2020.11-Linux-x86_64.sh
chmod +x Anaconda3-2020.11-Linux-x86_64.sh
bash Anaconda3-2020.11-Linux-x86_64.sh
```

This will install Anaconda on the machine. Next create a virtual environment using the `environment.yml` file provided by

```
conda env create -f environment.yml
```

This creates the virtual environment `cv4rs` on the machine. Depending on the shell used the environment can be activated using

```
conda activate cv4rs
```

This environment should be able to run all the scripts and notebooks provided with the project.

Option 2 : Use pip

The packages used are provided in `requirements.txt` file which can be used to setup all the dependencies on a machine from scratch. This can be done using pip by

```
pip install -r requirements.txt
```

Description of the files & Usage

- `main.py` : The file to run to start training based on the required experiment setup. Required Arguments are data path and the CNN model to use. For more details on how to use the arguments use `python main.py -h`. E.g. `python main.py -c resnet34 -d ./UCMerced_LandUse/Images` trains FedAVG using resnet34 with the images in the provided path. The following parameters can be chosen for training:
 - CNN Model
 - Number of clients
 - Federated Learning Model / Centralized
 - Percentage of label skewness
 - Number of Epochs
 - Number of local Epochs (FedAVG and FedProx)
 - Learning Rate
 - Small Skew
 - C Fraction (For FedAvg and FedProx)
 - Batch Size
 - Validation Split
 - Data directory and multilabel excel file pathThe details are given below.
- `visualize.py` : This file is used to plot the results from training. When training the FL models using `main.py` a `csv` file is generated containing *loss*, *accuracy* and *F1-Score*. For more details on how to use the arguments use `python visualize.py -h`.
- `legacy_notebooks` : Code before combining all notebooks to a single project. Initial work was done almost individually.
- `multilabels` : Contains the multilabel excel files for the UCMerced_LandUse dataset.
- `cnn_nets.py` : Contains the CNN architectures that can be used: `ResNet34`, `LeNet` and `AlexNet`
 - `LeNet` : This model is copied from the original, with some minor adjustments for our dataset; the Kernel size is set to 5x5, additionally we adjust the output layer to our dataset.
 - `ResNet34` : We use the `ResNet34` model provided by PyTorch and adjust the output layer to our dataset. The `pretrained` flag is set to `false`.
 - `AlexNet` : We use the `AlexNet` model provided by PyTorch and adjust the output layer to our dataset. The `pretrained` flag is set to `false`.
- `custom_dataloader.py` : Functions used in the class are used to generate a dataloader for an arbitrary number of clients and split the data in both IID and non-IID distributions. Furthermore it checks which classes are least correlated.
- `custom_loss_fns.py` : Custom loss functions can be found here. One loss function is specifically for the FedProx Federated Learning algorithm. The other one is a wrapper to the Pytorch loss function. This was included to have a generic train function that supports custom loss functions.
- `CustomDataSet.py` : Inherits the abstract class `torch.utils.data.Dataset` and overrides `__len__` and `__getitem__` method. This custom dataset class supports multilabel for each image.
- `federated_train_algorithms.py` : Includes the implemented Federated Learning models. Supported FL algorithms:

- FedAvg (Federated Averaging) ([Paper](#))
- FedProx ([Paper](#))
- Bulk Synchronous Processing (BSP) ([Paper](#))

More information on the implementation of each algorithm can be found below.

- `FL_with_pytorch_only.ipynb` : The final notebook version from which the modular code was written from.
- `train.py` : The file has the training loop. This training loop is used by all federated algorithms.
- `requirements.txt` : Can be used directly with pip/conda to setup the required packages.

Data Preparation and Splitting

The augmented data set is available in the subfolder `UCMerced_LandUse_augment` with a link to the zip file to download from [tubcloud](#). It also has the multilabel file for the augmented version of the dataset.

the function `load_split_train_test` in `custom_data_loader.py` is used to create training dataloaders for the clients and a test (validation) dataloader for the server. the dataloaders are `torch.utils.data.DataLoader`.

but first:

1. a transformation is defined that rescale the images, add a random horizontal flip augmentation, then turn the image into Pytorch tensor and then normalize it.
2. a train and test `CustomDataSet` objects (defined in `CustomDataSet.py`) are created, they store all the images from the dataset path and their labels, those custom datasets have the functions `getitem` that returns the image and its label as Pytorch tensors of a given index, and the `getlabel` that returns the label as a NumPy array.
3. the images indices are first shuffled then split (based on the `Validation Split` parameter)
4. for centralized learning the train and test dataloaders are created from the split from **3**, but for federated, the train indices are split into multiple lists of indices using the function `sampler_split_for_client` based on the `number of clients` parameter, the dataset label distribution and the `percentage of label skewness` parameter. Then a train dataloader is created for each of the indices list that represents a client.

The function `sampler_split_for_client` starts by calling the function `uncor_selector`, which returns a list of classes indices (with the size of `number of clients`) that are least correlated. A given image index is assigned to a client based on the `percentage of label skewness` if the label for that index has one of the labels returned from the `uncor_selector`, otherwise it is randomly assigned to one of the clients.

the `uncor_selector` selects as many uncorrelated classes as the `number of clients`, based on the labels following score: the sum of xor - the sum of and between 2 labels. the parameter `Small Skew` signals to the function whether to calculate the correlation and choose the class indices among the more or less common labels.

Implementation of the Federated Learning Models

All clients, given to the the FL algorithms as parameter, are `torch.utils.data.DataLoader` containing the respective data partition. The communication between *server* and *client* is simulated for all three algorithms.

FedAVG (Federated Averaging)

1. For each round, FedAVG first chooses a random fraction of clients `client_subset` to be trained. The number of chosen clients is bounded by the given parameter `c_fraction`.
2. In the next step, we iterate through `client_subset` to train each chosen client. For each client, the current model (still untrained at first round) is sent out for training using the respective data partition. To avoid issues we need to initialize a new

optimizer and scheduler. Moreover, we have to create a new model and (deep-) copy the parameters from the original model to it. The training is carried out through the `train` function from the `train.py` file which returns the trained model. Each trained model is stored in the array `model_client_list` which is later used in step **3** for averaging.

3. Once we trained a model for each client from `client_subset` we proceed with the averaging: for each client we iterate through all layers and average the weights. Each client (not only from `client_subset`) is weighted depending on the size of its data partition. E.g. let $n=1000$ be the size of the whole dataset and $m=100$ is the size of a client's data partition. Its model will be weighted $m/n=1/10$. Finally, we receive an averaged model.
4. Start again from step **1** for the next round using the averaged model.

FedProx

The algorithm for `FedProx` is with exception of the used loss function exactly like for `FedAVG`. `FedProx` adds a proximal term to the standard local loss function that penalizes models that deviate too much from the global model. The loss function is defined in `custom_loss_fns.py` under the class `FedProxLoss`. This class takes in basic loss function `base_criterion` which can again be same as the one used for FedProx and a hyper parameter `mu` which impact how much the proximal term affects the overall loss. The proximal term is a penalty on the local client weights for deviating too much from the central weights.

BSP

The implementation of `BSP` is similar to `FedAVG` with a few exceptions. First, it skips step **1** of choosing a fraction of clients. Furthermore, `BSP` doesn't average the models of all clients but rather passes them from one client to the next. A round is finished once the model has been passed to all clients (and has been trained by them).