Assignment 3: Federated Learning

Deadline: 11:59PM CST(China Standard Time), June 7th, 2025

1 Homework Grade Policy

Submission Format: Please format your homework **in English** using one of the two options below. Failing to do so will result in a minimum deduction of **10 points** from your homework grade:

1. **One PDF** generated by Jupyter Notebook that includes your code, complete output results, plotted figures, and written answers to analytical questions.
2. **One PDF + One PY file.** The `.py` file must contain all your code and be fully runnable. The PDF should include screenshots of your output results, plotted figures, along with your written answers to analytical questions.

Late Policy: Homework submitted within 72 hours after the deadline will receive 80% of the earned score. Submissions made more than three days after the deadline without a prior extension request will receive a score of 0. The request must be made before the deadline of the homework.

2 Introduction

This assignment expands upon the work you completed in HW2, where you created a Neural Network for MNIST classification. Now, in this homework, you will enhance the model to support Federated Learning.

3 Code from Assignment 2

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from multiprocessing import Pool
from torch.utils.data import RandomSampler
# Hyperparameters
learning_rate = 0.1
batch_size = 128
num_communications = 50
num_workers = 2 # Number of models to train in parallel
num_local_steps = 5

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()

# MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True,
    transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./data', train=False,
    transform=transforms.ToTensor())

# Neural network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

4 Utility Functions

The following code defines two utility functions:

1. `average_models`: This function takes a list of models as input and returns the averaged model parameters. It calculates the mean of the model parameters across all the models to create an ensemble model.

```
# Function to average the model weights
def average_models(models):
    model_params = [model.state_dict() for model in models]
    averaged_params = {}

    for param_name in model_params[0]:
        params = torch.stack([model_params[i][param_name] for
             i in range(len(models))])
        averaged_params[param_name] = torch.mean(params, dim
            =0)

    return averaged_params
```

2. `duplicate_model`: This function takes a model and the number of duplicates as input. It creates multiple copies of the model by cloning its state dictionary and returns a list of duplicated models.

```
     def duplicate_model(model, num_duplicates):
    model_dicts = [model.state_dict() for _ in range(
        num_duplicates)]
    duplicated_models = [Net() for _ in range(num_duplicates)
        ]
    for i, model_dict in enumerate(model_dicts):
        duplicated_models[i].load_state_dict(model_dict)
    return duplicated_models
```

5 Coding Task

5.1 Finish the Training Function for Each Worker (25pts)

This function to take a model, optimizer, loss criterion, random sampler, and the training dataset as input. It trains the model on the local data subset defined by the random sampler and returns the trained model and the total loss during training.

```
# Function to train a model
def train_model(args):
    model, optimizer, criterion, random_sampler, train_dataset =
        args

    # Create a new data loader with the random sampler
    train_loader_random = torch.utils.data.DataLoader(dataset=
        train_dataset,batch_size=batch_size,sampler=random_sampler
        )

    total_loss = 0
    for i, (images, labels) in enumerate(train_loader_random):
        #TODO: Write your own code to train the model

    return model, total_loss
```

5.2 Finish the Training Loop (30pts)

In each communication round, the following steps are performed:

1. A number of models equal to `num_models` are created by duplicating the initial model.
2. Separate optimizers are created for each model with Stochastic Gradient Descent (SGD) as the optimization algorithm.
3. The models are trained in parallel using the `train_model` function. Each model is trained on a different random subset of the training data using the `random_sampler`.
4. The trained models and their total loss during training are stored.
5. The model weights are averaged using the `average_models` function to create an ensemble model.
6. The total loss of all models in this communication round is stored in the `losses_array`.

```
# Random sampler for train loader
random_sampler = RandomSampler(train_dataset, replacement = True,
     num_samples = num_local_steps*len(train_loader))

# Training loop
losses_array = []
model = Net()

for _ in range(num_communications):
    models = duplicate_model(model, num_workers)
    optimizers = [optim.SGD(model.parameters(), lr=learning_rate)
        for model in models]

    #TODO: Design your own parallel process to train all models
        with train_model function. (It might be helpful to refer
        previous exercises.)

    # Average the models' weights to create the ensemble model
    ensemble_model_params = average_models(models)
    model.load_state_dict(ensemble_model_params)

    losses_array.append(total_loss)

plt.plot(losses_array)
plt.show()
```

5.3 Testing the Model

After training, we switch the model to evaluation mode (`model.eval()`). We then iterate over the test dataset and calculate the accuracy of the model's predictions. The accuracy is printed as a percentage.

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
    batch_size=batch_size, shuffle=False)
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f'Test Accuracy: {100 * correct / total}%')
```

6 Questions

1. Initialize and train a new model using each of the specified hyperparameter configurations. For each run, **plot the training loss versus the number of epochs, and report the training and test accuracies.** Each run is expected to complete within 5 to 30 minutes, depending on your computer's specifications.

   (a) **15pts:** `batch_size = 32`, `num_communications = 50`, `learning_rate = 0.1`, `num_workers = 2`, `num_local_steps = 5`.

   (b) **15pts:** `batch_size = 32`, `num_communications = 50`, `learning_rate = 0.1`, `num_workers = 4`, `num_local_steps = 5`.

   (c) **15pts:** `batch_size = 32`, `num_communications = 50`, `learning_rate = 0.1`, `num_workers = 8`, `num_local_steps = 5`.

   (d) **15pts:** `batch_size = 32`, `num_communications = 50`, `learning_rate = 0.1`, `num_workers = 4`, `num_local_steps = 20`.