# Assignment 3: Federated Learning

## 1 Coding Task

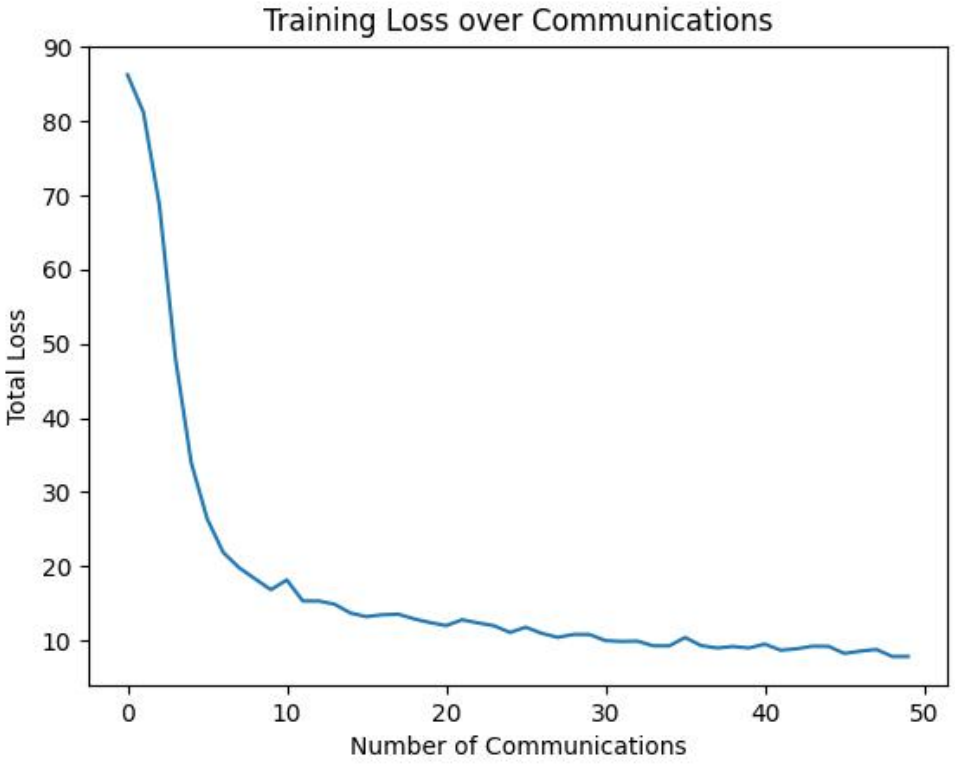
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from torch.multiprocessing import Pool
from torch.utils.data import RandomSampler
import torch.multiprocessing as mp
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# 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
# 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
# Function to duplicate a model
def duplicate_model(model, num_duplicates):
    model_dicts = [model.state_dict() for _ in range(num_duplicates)]
    duplicated_models = [Net().to(device) for _ in range(num_duplicates)]
    for i, model_dict in enumerate(model_dicts):
        duplicated_models[i].load_state_dict(model_dict)
    return duplicated_models
# Function to train a model
def train_model(args):
    model, optimizer, criterion, random_sampler, train_dataset = args
    model.to(device)
    # 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
    model.train()
    for i, (images, labels) in enumerate(train_loader_random):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    model.cpu()
    return model, total_loss
# Random sampler for train loader
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
random_sampler = RandomSampler(train_dataset, replacement=True,
                               num_samples=num_local_steps * len(train_loader))
# Training loop
losses_array = []
model = Net().to(device)
if __name__ == '__main__':
    mp.set_start_method('spawn')
    for _ in range(num_communications):
        models = duplicate_model(model, num_workers)
```

```
        optimizers = [optim.SGD(model.parameters(), lr=learning_rate) for model in models]
        args_list = [(models[i], optimizers[i], criterion, random_sampler, train_dataset) for i in r
ange(num_workers)]
        ctx = mp.get_context("spawn")
        with ctx.Pool(processes=num_workers) as pool:
            results = pool.map(train_model, args_list)
        models = [result[0].to(device) for result in results]
        total_losses = [result[1] for result in results]
        total_loss = sum(total_losses)
        # 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.xlabel('Number of Communications')
    plt.ylabel('Total Loss')
    plt.title('Training Loss over Communications')
    plt.savefig('CodingTask.png')
    plt.show()
    # Testing the Model
    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:
            images, labels = images.to(device), labels.to(device)
            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}%')
```

The screenshots of the output results are as follows.

```
(.conda) lumos@lumoslc:~/Desktop/python/program/week2/assignment$ python CodingTask.py
Test Accuracy: 93.36%
(.conda) lumos@lumoslc:~/Desktop/python/program/week2/assignment$
```




The original Coding Task was not designed to use the GPU. However, in order to accelerate the training process and ensure that each test set runs within approximately five minutes in my computer, the GPU was utilized. By leveraging the parallel processing capabilities of the GPU, the computational time for training the neural network models was significantly reduced.

# 2 Questions

In addition to the basic implementation of Federated Learning in the Coding Task, four new models with specified hyperparameter configurations were trained using the same code framework.

```
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 torch.utils.data import DataLoader, RandomSampler
import torch.multiprocessing as mp



device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# Define a simple neural network model for MNIST classification
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

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

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

# Function to train a single model on the local dataset
def train_model(args):
    model, optimizer, criterion, random_sampler, train_dataset, batch_size = args
    model.to(device)  # Ensure the model is on the correct device (GPU if available)
    train_loader_random = DataLoader(dataset=train_dataset, batch_size=batch_size, sampler=random_sa
mpler)
    total_loss = 0
    model.train()
    for i, (images, labels) in enumerate(train_loader_random):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
```

```
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    model.cpu()
    return model, total_loss

def test_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            model.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        model.cpu()
    return correct / total

def federated_training(batch_size, num_communications, learning_rate, num_workers, num_local_steps):
    train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), downl
oad=True)
    test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
    random_sampler = RandomSampler(train_dataset, replacement=True, num_samples=num_local_steps * le
n(train_loader))
    criterion = nn.CrossEntropyLoss()
    model = Net().to(device)  # Ensure the model starts on the correct device
    losses = []
    train_accuracies = []
    test_accuracies = []
    for _ in range(num_communications):
        models = duplicate_model(model, num_workers)
        optimizers = [optim.SGD(model.parameters(), lr=learning_rate) for model in models]
        args_list = [(models[i], optimizers[i], criterion, random_sampler, train_dataset, batch_size)

for i in
                    range(num_workers)]
        # 使用 spawn 上下文创建 Pool
        ctx = mp.get_context("spawn")
        with ctx.Pool(processes=num_workers) as pool:
```

```
            results = pool.map(train_model, args_list)
        models = [result[0].to(device) for result in results]
        total_losses = [result[1] for result in results]
        total_loss = sum(total_losses)
        losses.append(total_loss)
        ensemble_model_params = average_models(models)
        model.load_state_dict(ensemble_model_params)
        train_accuracy = test_model(model, train_loader)
        test_accuracy = test_model(model, test_loader)
        train_accuracies.append(train_accuracy)
        test_accuracies.append(test_accuracy)
    return losses, train_accuracies, test_accuracies

if __name__ == '__main__':
    if mp.get_start_method(allow_none=True) is None:
        mp.set_start_method('spawn')
    hyperparam_configs = [
        (32, 50, 0.1, 2, 5),
        (32, 50, 0.1, 4, 5),
        (32, 50, 0.1, 8, 5),
        (32, 50, 0.1, 4, 20)
    ]
    for i, config in enumerate(hyperparam_configs):
        batch_size, num_communications, learning_rate, num_workers, num_local_steps = config
        print(f"Running experiment with config: batch size={batch_size}, num communications={num_com
munications}, "
              f"learning rate={learning_rate}, num workers={num_workers}, num local steps={num_local
_steps}")
        losses, train_accuracies, test_accuracies = federated_training(batch_size, num_communication
s, learning_rate,
                                                                       num_workers, num_local_steps)
        plt.figure()
        plt.plot(range(num_communications), losses)
        plt.xlabel('Number of Communications')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss vs Communications (Config {i + 1})')
        filename = f'training_loss_config_{i + 1}.png'
        plt.savefig(filename)
        plt.show()
        final_train_accuracy = train_accuracies[-1]
        final_test_accuracy = test_accuracies[-1]
        print(f"Final Training Accuracy: {final_train_accuracy * 100:.2f}%")
        print(f"Final Test Accuracy: {final_test_accuracy * 100:.2f}%")
        print("-" * 50)
```

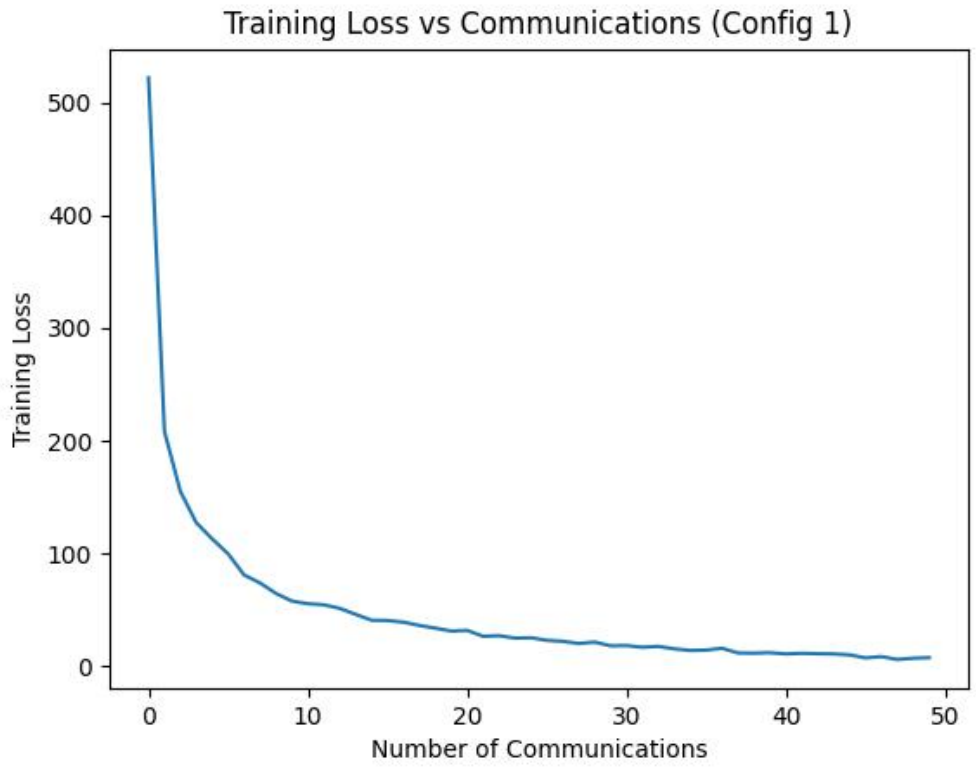
The screenshots of the output results are presented below.

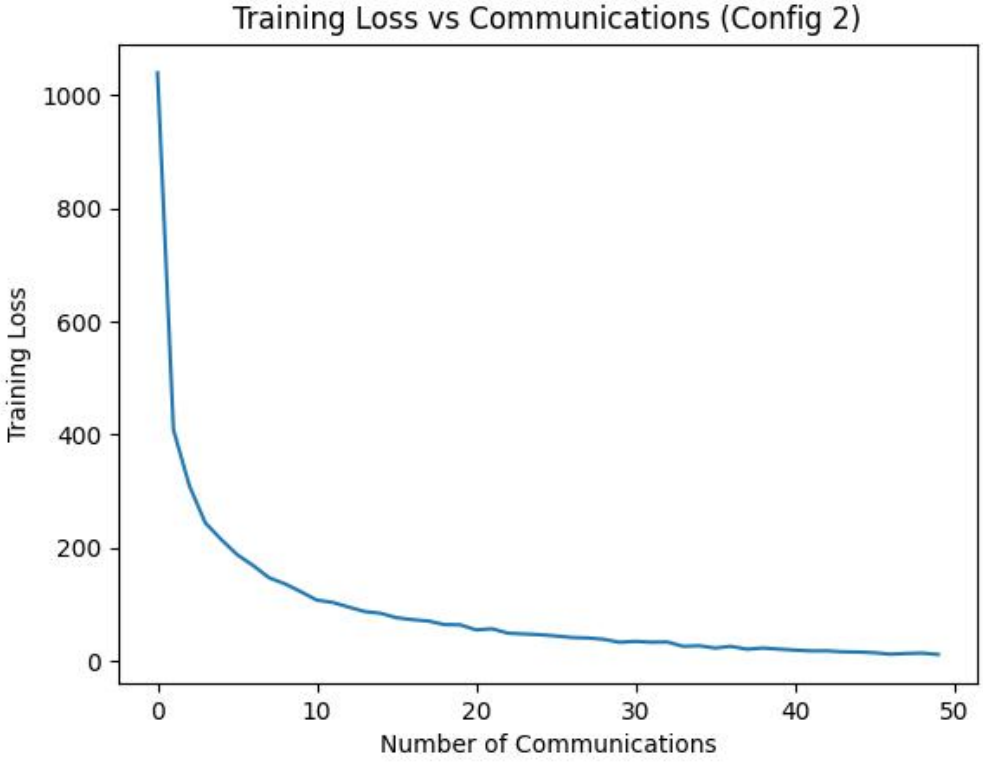
```
问题 1  输出  调试控制台  终端  端口  JUPYTER  评论

(.conda) lumos@lumoslc:~/Desktop/python/program/week2/assignment$ python main.py
Running experiment with config: batch size=32, num communications=50, learning rate=0.1, num workers=2, num local steps=5
Final Training Accuracy: 99.82%
Final Test Accuracy: 98.14%
--------------------------------------------------
Running experiment with config: batch size=32, num communications=50, learning rate=0.1, num workers=4, num local steps=5
Final Training Accuracy: 99.93%
Final Test Accuracy: 98.20%
--------------------------------------------------
Running experiment with config: batch size=32, num communications=50, learning rate=0.1, num workers=8, num local steps=5
Final Training Accuracy: 99.95%
Final Test Accuracy: 98.29%
--------------------------------------------------
Running experiment with config: batch size=32, num communications=50, learning rate=0.1, num workers=4, num local steps=20
Final Training Accuracy: 100.00%
Final Test Accuracy: 98.37%
--------------------------------------------------
(.conda) lumos@lumoslc:~/Desktop/python/program/week2/assignment$
```

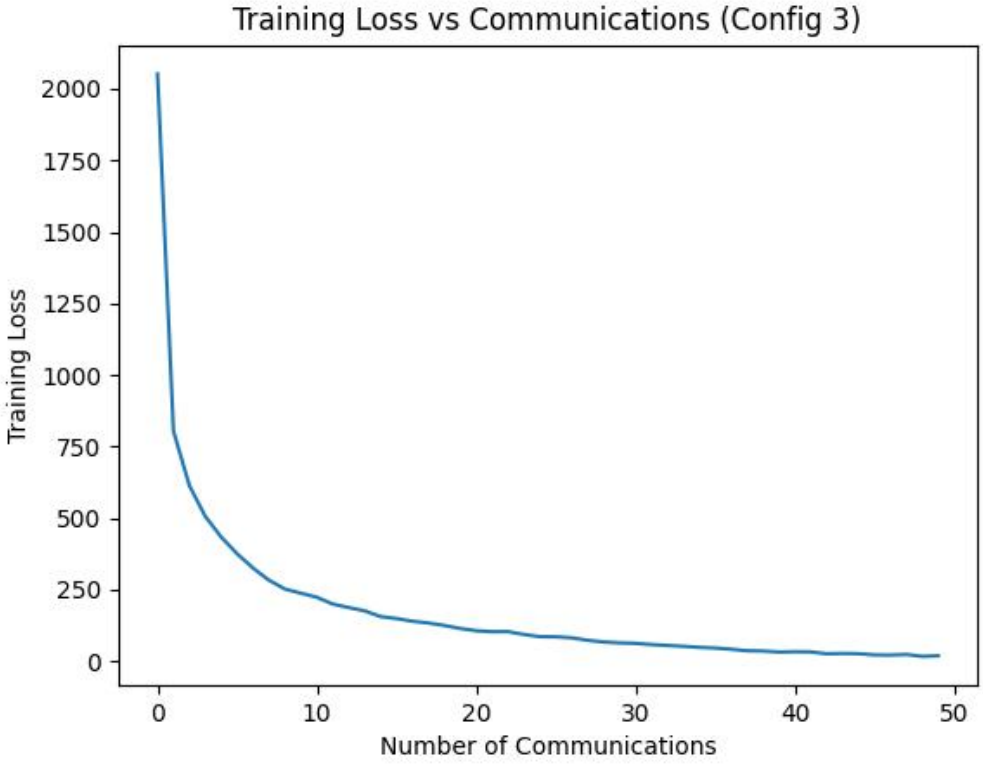
Here is the table of training and test accuracies of different hyperparameter configurations.

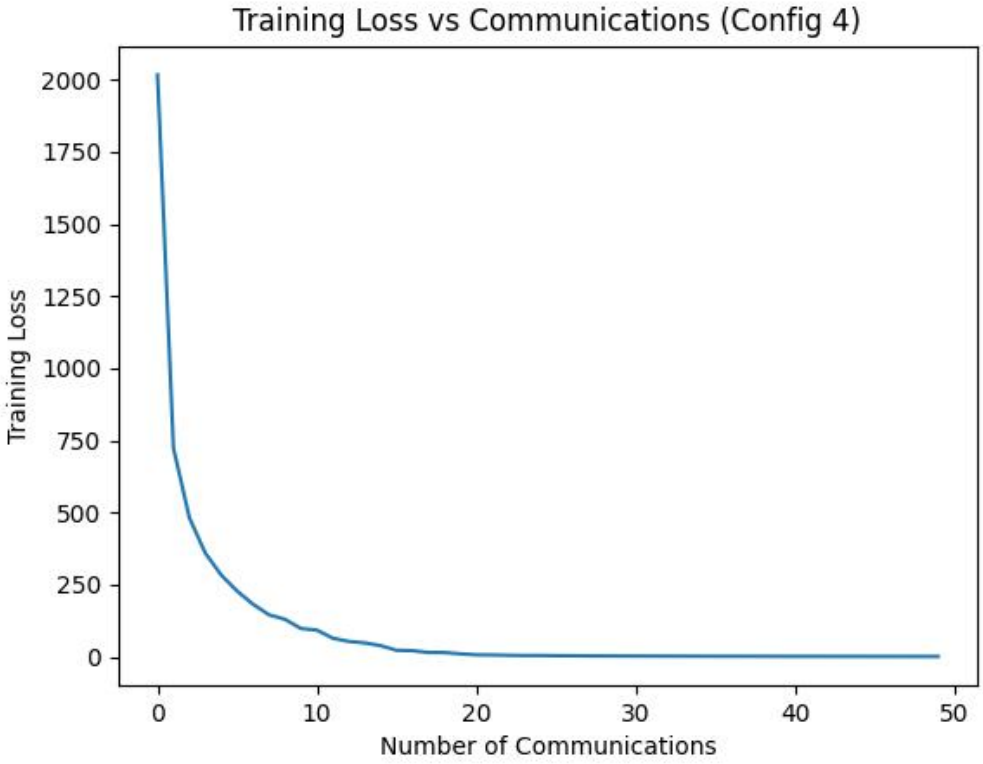
| Batch Size | Num Communications | Learning Rate | Num Workers | Num Local Steps | Final Training Accuracy | Final Test Accuracy |
|---|---|---|---|---|---|---|
| 32 | 50 | 0.1 | 2 | 5 | 99.82 | 98.14 |
| 32 | 50 | 0.1 | 4 | 5 | 99.93 | 98.2 |
| 32 | 50 | 0.1 | 8 | 5 | 99.95 | 98.29 |
| 32 | 50 | 0.1 | 4 | 20 | 100.0 | 98.37 |

The plots of different hyperparameter configurations are shown as follows.




Training Loss vs Communications (Config 2)
Training Loss
1000
800
600
400
200
0
0
10
20
30
40
50
Number of Communications


Training Loss vs Communications (Config 3)
Training Loss
2000
1750
1500
1250
1000
750
500
250
0
0
10
20
30
40
50
Number of Communications


Training Loss vs Communications (Config 4)
Training Loss
2000
1750
1500
1250
1000
750
500
250
0
0
10
20
30
40
50
Number of Communications