PaddleFL

Release 0.1.0.beta

PaddlePaddle

May 25, 2020

QUICK START

CHAPTER

ONE

INSTALLATION

We **highly recommend** to run PaddleFL in Docker

```
#Pull and run the docker
docker pull hub.baidubce.com/paddlefl/paddle_fl:latest
docker run --name <docker_name> --net=host -it -v $PWD:/root <image id> /bin/bash

#Install paddle_fl
pip install paddle_fl
```

We also prepare a stable redis package for you to download and install, which will be used in tasks with MPC.

```
wget --no-check-certificate https://paddlefl.bj.bcebos.com/redis-stable.tar
tar -xf redis-stable.tar
cd redis-stable &&  make
```

CHAPTER

TWO

COMPILE FROM SOURCE CODE

2.1 A. Environment preparation

- CentOS 6 or CentOS 7 (64 bit)
- Python 2.7.15+/3.5.1+/3.6/3.7 (64 bit) or above
- pip or pip3 9.0.1+ (64 bit)
- PaddlePaddle release 1.8
- Redis 5.0.8 (64 bit)
- GCC or G++ 4.8.3+
- cmake 3.15+

2.2 B. Clone the source code, compile and install

Fetch the source code and checkout stable release

```
git clone https://github.com/PaddlePaddle/PaddleFL
cd /path/to/PaddleFL

# Checkout stable release
mkdir build && cd build
```

Execute compile commands, where `PYTHON_EXECUTABLE` is path to the python binary where the PaddlePaddle is installed, `CMAKE_CXX_COMPILER` is the path of G++ and `PYTHON_INCLUDE_DIRS` is the corresponding python include directory. You can get the `PYTHON_INCLUDE_DIRS` via the following command:

```
${PYTHON_EXECUTABLE} -c "from distutils.sysconfig import get_python_inc;print(get_
↪python_inc())"
```

Then you can put the directory in the following command and make:

```
cmake ../ -DPYTHON_EXECUTABLE=${PYTHON_EXECUTABLE} -DPYTHON_INCLUDE_DIRS=${python_
↪include_dir} -DCMAKE_CXX_COMPILER=${g++_path}
make -j$(nproc)
```

Install the package:

```
make install
cd /path/to/PaddleFL/python
${PYTHON_EXECUTABLE} setup.py sdist bdist_wheel
pip or pip3 install dist/***.whl -U
```

CHAPTER

THREE

INSTRUCTIONS FOR DATA PARALLEL

3.1 Step 1: Define Federated Learning Compile-Time

We define very simple multiple layer perceptron for demonstration. When multiple organizations agree to share data knowledge through PaddleFL, a model can be defined with agreement from these organizations. A FLJob can be generated and saved. Programs needed to be run each node will be generated separately in FLJob.

```
import paddle.fluid as fluid
import paddle_fl as fl
from paddle_fl.core.master.job_generator import JobGenerator
from paddle_fl.core.strategy.fl_strategy_base import FLStrategyFactory

class Model(object):
    def __init__(self):
        pass

    def mlp(self, inputs, label, hidden_size=128):
        self.concat = fluid.layers.concat(inputs, axis=1)
        self.fc1 = fluid.layers.fc(input=self.concat, size=256, act='relu')
        self.fc2 = fluid.layers.fc(input=self.fc1, size=128, act='relu')
    self.predict = fluid.layers.fc(input=self.fc2, size=2, act='softmax')
        self.sum_cost = fluid.layers.cross_entropy(input=self.predict, label=label)
        self.accuracy = fluid.layers.accuracy(input=self.predict, label=label)
        self.loss = fluid.layers.reduce_mean(self.sum_cost)
        self.startup_program = fluid.default_startup_program()

inputs = [fluid.layers.data( \
            name=str(slot_id), shape=[5],
        dtype="float32")
      for slot_id in range(3)]
label = fluid.layers.data( \
            name="label",
            shape=[1],
            dtype='int64')

model = Model()
model.mlp(inputs, label)

job_generator = JobGenerator()
optimizer = fluid.optimizer.SGD(learning_rate=0.1)
job_generator.set_optimizer(optimizer)
job_generator.set_losses([model.loss])
job_generator.set_startup_program(model.startup_program)
job_generator.set_infer_feed_and_target_names(
```

(continues on next page)

(continued from previous page)

```
    [x.name for x in inputs], [model.predict.name])

build_strategy = FLStrategyFactory()
build_strategy.fed_avg = True
build_strategy.inner_step = 1
strategy = build_strategy.create_fl_strategy()

endpoints = ["127.0.0.1:8181"]
output = "fl_job_config"
job_generator.generate_fl_job(
    strategy, server_endpoints=endpoints, worker_num=2, output=output)
```

3.2 Step 2: Issue FL Job to Organizations

We can define a secure service to send programs to each node in FLJob. There are two types of nodes in distributed federated learning job. One is FL Server, the other is FL Trainer. A FL Trainer is owned by individual organization and an organization can have multiple FL Trainers given different amount of data knowledge the organization is willing to share. A FL Server is owned by a secure distributed training cluster. By means of security of the cluster, all organizations participated in the Federated Training Job should agree to trust the cluster is secure.

3.3 Step 3: Start Federated Learning Run-Time

On FL Scheduler Node, number of servers and workers are defined. Besides, the number of workers that participate in each upating cycle is also determined. Finally, the FL Scheduler waits servers and workers to initialize.

```
from paddle_fl.core.scheduler.agent_master import FLScheduler

worker_num = 2
server_num = 1
# Define the number of worker/server and the port for scheduler
scheduler = FLScheduler(worker_num,server_num,port=9091)
scheduler.set_sample_worker_num(worker_num)
scheduler.init_env()
print("init env done.")
scheduler.start_fl_training()
```

On FL Trainer Node, a training script is defined as follows:

```
from paddle_fl.core.trainer.fl_trainer import FLTrainerFactory
from paddle_fl.core.master.fl_job import FLRunTimeJob
import numpy as np
import sys

def reader():
    for i in range(1000):
        data_dict = {}
        for i in range(3):
            data_dict[str(i)] = np.random.rand(1, 5).astype('float32')
        data_dict["label"] = np.random.randint(2, size=(1, 1)).astype('int64')
        yield data_dict
```

(continues on next page)

(continued from previous page)

```
trainer_id = int(sys.argv[1]) # trainer id for each guest
job_path = "fl_job_config"
job = FLRunTimeJob()
job.load_trainer_job(job_path, trainer_id)
job._scheduler_ep = "127.0.0.1:9091" # Inform the scheduler IP to trainer
trainer = FLTrainerFactory().create_fl_trainer(job)
trainer.start()

output_folder = "fl_model"
step_i = 0
while not trainer.stop():
    step_i += 1
    print("batch %d start train" % (step_i))
    trainer.run(feed=data, fetch=[])
    if trainer_id == 0:
        print("start saving model")
        trainer.save_inference_program(output_folder)
    if step_i >= 100:
       break
```

On FL Server Node, a training script is defined as follows:

```
import paddle_fl as fl
import paddle.fluid as fluid
from paddle_fl.core.server.fl_server import FLServer
from paddle_fl.core.master.fl_job import FLRunTimeJob
server = FLServer()
server_id = 0
job_path = "fl_job_config"
job = FLRunTimeJob()
job.load_server_job(job_path, server_id)
job._scheduler_ep = "127.0.0.1:9091" # IP address for scheduler
server.set_server_job(job)
server._current_ep = "127.0.0.1:8181" # IP address for server
server.start()
```

CHAPTER

FOUR

INSTRUCTIONS FOR FEDERATED LEARNING WITH MPC

A PFM program is exactly a PaddlePaddle program, and will be executed as normal PaddlePaddle programs. Before training/inference, user needs to choose a MPC protocol, define a machine learning model and their training strategies. Typical machine learning operators are provided in paddle_fl.mpc over encrypted data, of which the instances are created and run in order by Executor during run-time.

Below is an example for complish an vertial Federated Learning with MPC

```
import sys
import numpy as np
import time

import paddle
import paddle.fluid as fluid
import paddle_fl.mpc as pfl_mpc
import paddle_fl.mpc.data_utils.aby3 as aby3

# define your role number(0, 1, 2) and the address of redis server
role, server, port = sys.argv[1], sys.argv[2], sys.argv[3]

# specify the protocal and initialize the environment
pfl_mpc.init("aby3", int(role), "localhost", server, int(port))
role = int(role)

# data preprocessing
BATCH_SIZE = 10

feature_reader = aby3.load_aby3_shares("/tmp/house_feature", id=role, shape=(13, ))
label_reader = aby3.load_aby3_shares("/tmp/house_label", id=role, shape=(1, ))
batch_feature = aby3.batch(feature_reader, BATCH_SIZE, drop_last=True)
batch_label = aby3.batch(label_reader, BATCH_SIZE, drop_last=True)

x = pfl_mpc.data(name='x', shape=[BATCH_SIZE, 13], dtype='int64')
y = pfl_mpc.data(name='y', shape=[BATCH_SIZE, 1], dtype='int64')

# async data loader
loader = fluid.io.DataLoader.from_generator(feed_list=[x, y], capacity=BATCH_SIZE)
batch_sample = paddle.reader.compose(batch_feature, batch_label)
place = fluid.CPUPlace()
loader.set_batch_generator(batch_sample, places=place)

# define your model
y_pre = pfl_mpc.layers.fc(input=x, size=1)

infer_program = fluid.default_main_program().clone(for_test=False)
```

(continues on next page)

(continued from previous page)

```
cost = pfl_mpc.layers.square_error_cost(input=y_pre, label=y)
avg_loss = pfl_mpc.layers.mean(cost)
optimizer = pfl_mpc.optimizer.SGD(learning_rate=0.001)
optimizer.minimize(avg_loss)

# give the path to store training loss
loss_file = "/tmp/uci_loss.part{}".format(role)

# train
exe = fluid.Executor(place)
exe.run(fluid.default_startup_program())
epoch_num = 20

start_time = time.time()
for epoch_id in range(epoch_num):
    step = 0

    # Method 1: feed data directly
    # for feature, label in zip(batch_feature(), batch_label()):
    #     mpc_loss = exe.run(feed={"x": feature, "y": label}, fetch_list=[avg_loss])

    # Method 2: feed data via loader
    for sample in loader():
        mpc_loss = exe.run(feed=sample, fetch_list=[avg_loss])

        if step % 50 == 0:
            print('Epoch={}, Step={}, Loss={}'.format(epoch_id, step,
                                                      mpc_loss))
            with open(loss_file, 'ab') as f:
                f.write(np.array(mpc_loss).tostring())
            step += 1

end_time = time.time()
print('Mpc Training of Epoch={} Batch_size={}, cost time in seconds:{}'
      .format(epoch_num, BATCH_SIZE, (end_time - start_time)))
```

CHAPTER

FIVE

PADDLEFL

PaddleFL is an open source federated learning framework based on PaddlePaddle. Researchers can easily replicate and compare different federated learning algorithms with PaddleFL. Developers can also benefit from PaddleFL in that it is easy to deploy a federated learning system in large scale distributed clusters. In PaddleFL, serveral federated learning strategies will be provided with application in computer vision, natural language processing, recommendation and so on. Application of traditional machine learning training strategies such as Multi-task learning, Transfer Learning in Federated Learning settings will be provided. Based on PaddlePaddle's large scale distributed training and elastic scheduling of training job on Kubernetes, PaddleFL can be easily deployed based on full-stack open sourced software.

CHAPTER

SIX

FEDERATED LEARNING

Data is becoming more and more expensive nowadays, and sharing of raw data is very hard across organizations. Federated Learning aims to solve the problem of data isolation and secure sharing of data knowledge among organizations. The concept of federated learning is proposed by researchers in Google [1, 2, 3].

6.1 Overview of PaddleFL

In PaddleFL, horizontal and vertical federated learning strategies will be implemented according to the categorization given in [4]. Application demonstrations in natural language processing, computer vision and recommendation will be provided in PaddleFL.

6.1.1 A. Federated Learning Strategy

- **Vertical Federated Learning**: Logistic Regression with PrivC[5], Neural Network with MPC [11]
- **Horizontal Federated Learning**: Federated Averaging [2], Differential Privacy [6], Secure Aggregation

6.1.2 B. Training Strategy

- **Multi Task Learning** [7]
- **Transfer Learning** [8]
- **Active Learning**

There are mainly two components in PaddleFL: **Data Parallel** and **Federated Learning with MPC (PFM)**.

With Data Parallel, distributed data holders can finish their Federated Learning tasks based on common horizontal federated strategies, such as FedAvg, DPSGD, etc.

Besides, PFM is implemented based on secure multi-party computation (MPC) to enable secure training and prediction. As a key product of PaddleFL, PFM intrinsically supports federated learning well, including horizontal, vertical and transfer learning scenarios. Users with little cryptography expertise can also train models or conduct prediction on encrypted data.

6.2 Framework design of PaddleFL

In Data Parallel, components for defining a federated learning task and training a federated learning job are as follows:

6.2.1 A. Compile Time

- **FL-Strategy**: a user can define federated learning strategies with FL-Strategy such as Fed-Avg[2]
- **User-Defined-Program**: PaddlePaddle's program that defines the machine learning model structure and training strategies such as multi-task learning.
- **Distributed-Config**: In federated learning, a system should be deployed in distributed settings. Distributed Training Config defines distributed training node information.
- **FL-Job-Generator**: Given FL-Strategy, User-Defined Program and Distributed Training Config, FL-Job for federated server and worker will be generated through FL Job Generator. FL-Jobs will be sent to organizations and federated parameter server for run-time execution.

6.2.2 B. Run Time

- **FL-Server**: federated parameter server that usually runs in cloud or third-party clusters.
- **FL-Worker**: Each organization participates in federated learning will have one or more federated workers that will communicate with the federated parameter server.
- **FL-scheduler**: Decide which set of trainers can join the training before each updating cycle.

Federated Learning with MPC

Paddle FL MPC implements secure training and inference tasks based on the underlying MPC protocol like ABY3[11], which is a high efficient three-party computing model.

In ABY3, participants can be classified into roles of Input Party (IP), Computing Party (CP) and Result Party (RP). Input Parties (e.g., the training data/model owners) encrypt and distribute data or models to Computing Parties. Computing Parties (e.g., the VM on the cloud) conduct training or inference tasks based on specific MPC protocols, being restricted to see only the encrypted data or models, and thus guarantee the data privacy. When the computation is completed, one or more Result Parties (e.g., data owners or specified third-party) receive the encrypted results from Computing Parties, and reconstruct the plaintext results. Roles can be overlapped, e.g., a data owner can also act as a computing party.

A full training or inference process in PFM consists of mainly three phases: data preparation, training/inference, and result reconstruction.

6.2.3 A. Data preparation

- **Private data alignment**: PFM enables data owners (IPs) to find out records with identical keys (like UUID) without revealing private data to each other. This is especially useful in the vertical learning cases where segmented features with same keys need to be identified and aligned from all owners in a private manner before training.
- **Encryption and distribution**: In PFM, data and models from IPs will be encrypted using Secret-Sharing[10], and then be sent to CPs, via directly transmission or distributed storage like HDFS. Each CP can only obtain one share of each piece of data, and thus is unable to recover the original value in the Semi-honest model.

6.2.4 B. Training/inference

A PFM program is exactly a PaddlePaddle program, and will be executed as normal PaddlePaddle programs. Before training/inference, user needs to choose a MPC protocol, define a machine learning model and their training strategies.

Typical machine learning operators are provided in `paddle_fl.mpc` over encrypted data, of which the instances are created and run in order by Executor during run-time.

6.2.5 C. Result reconstruction

Upon completion of the secure training (or inference) job, the models (or prediction results) will be output by CPs in encrypted form. Result Parties can collect the encrypted results, decrypt them using the tools in PFM, and deliver the plaintext results to users.

CHAPTER

SEVEN

ON GOING AND FUTURE WORK

- Vertial Federated Learning will support more algorithms.
- Add K8S deployment scheme for Paddle Encrypted.
- FL mobile simulator will be open sourced in following versions.

CHAPTER

EIGHT

PADDLEFL

PaddleFL is an open source federated learning framework based on PaddlePaddle. Researchers can easily replicate and compare different federated learning algorithms with PaddleFL. Developers can also benefit from PaddleFL in that it is easy to deploy a federated learning system in large scale distributed clusters. In PaddleFL, serveral federated learning strategies will be provided with application in computer vision, natural language processing, recommendation and so on. Application of traditional machine learning training strategies such as Multi-task learning, Transfer Learning in Federated Learning settings will be provided. Based on PaddlePaddle's large scale distributed training and elastic scheduling of training job on Kubernetes, PaddleFL can be easily deployed based on full-stack open sourced software.

CHAPTER

NINE

FEDERATED LEARNING

Data is becoming more and more expensive nowadays, and sharing of raw data is very hard across organizations. Federated Learning aims to solve the problem of data isolation and secure sharing of data knowledge among organizations. The concept of federated learning is proposed by researchers in Google [1, 2, 3].

9.1 Overview of PaddleFL

In PaddleFL, horizontal and vertical federated learning strategies will be implemented according to the categorization given in [4]. Application demonstrations in natural language processing, computer vision and recommendation will be provided in PaddleFL.

9.1.1 A. Federated Learning Strategy

- **Vertical Federated Learning**: Logistic Regression with PrivC[5], Neural Network with MPC [11]
- **Horizontal Federated Learning**: Federated Averaging [2], Differential Privacy [6], Secure Aggregation

9.1.2 B. Training Strategy

- **Multi Task Learning** [7]
- **Transfer Learning** [8]
- **Active Learning**

There are mainly two components in PaddleFL: **Data Parallel** and **Federated Learning with MPC (PFM)**.

With Data Parallel, distributed data holders can finish their Federated Learning tasks based on common horizontal federated strategies, such as FedAvg, DPSGD, etc.

Besides, PFM is implemented based on secure multi-party computation (MPC) to enable secure training and prediction. As a key product of PaddleFL, PFM intrinsically supports federated learning well, including horizontal, vertical and transfer learning scenarios. Users with little cryptography expertise can also train models or conduct prediction on encrypted data.

9.2 Framework design of PaddleFL

In Data Parallel, components for defining a federated learning task and training a federated learning job are as follows:

9.2.1 A. Compile Time

- **FL-Strategy**: a user can define federated learning strategies with FL-Strategy such as Fed-Avg[2]
- **User-Defined-Program**: PaddlePaddle's program that defines the machine learning model structure and training strategies such as multi-task learning.
- **Distributed-Config**: In federated learning, a system should be deployed in distributed settings. Distributed Training Config defines distributed training node information.
- **FL-Job-Generator**: Given FL-Strategy, User-Defined Program and Distributed Training Config, FL-Job for federated server and worker will be generated through FL Job Generator. FL-Jobs will be sent to organizations and federated parameter server for run-time execution.

9.2.2 B. Run Time

- **FL-Server**: federated parameter server that usually runs in cloud or third-party clusters.
- **FL-Worker**: Each organization participates in federated learning will have one or more federated workers that will communicate with the federated parameter server.
- **FL-scheduler**: Decide which set of trainers can join the training before each updating cycle.

Federated Learning with MPC

Paddle FL MPC implements secure training and inference tasks based on the underlying MPC protocol like ABY3[11], which is a high efficient three-party computing model.

In ABY3, participants can be classified into roles of Input Party (IP), Computing Party (CP) and Result Party (RP). Input Parties (e.g., the training data/model owners) encrypt and distribute data or models to Computing Parties. Computing Parties (e.g., the VM on the cloud) conduct training or inference tasks based on specific MPC protocols, being restricted to see only the encrypted data or models, and thus guarantee the data privacy. When the computation is completed, one or more Result Parties (e.g., data owners or specified third-party) receive the encrypted results from Computing Parties, and reconstruct the plaintext results. Roles can be overlapped, e.g., a data owner can also act as a computing party.

A full training or inference process in PFM consists of mainly three phases: data preparation, training/inference, and result reconstruction.

9.2.3 A. Data preparation

- **Private data alignment**: PFM enables data owners (IPs) to find out records with identical keys (like UUID) without revealing private data to each other. This is especially useful in the vertical learning cases where segmented features with same keys need to be identified and aligned from all owners in a private manner before training.
- **Encryption and distribution**: In PFM, data and models from IPs will be encrypted using Secret-Sharing[10], and then be sent to CPs, via directly transmission or distributed storage like HDFS. Each CP can only obtain one share of each piece of data, and thus is unable to recover the original value in the Semi-honest model.

9.2.4 B. Training/inference

A PFM program is exactly a PaddlePaddle program, and will be executed as normal PaddlePaddle programs. Before training/inference, user needs to choose a MPC protocol, define a machine learning model and their training strategies.

Typical machine learning operators are provided in `paddle_fl.mpc` over encrypted data, of which the instances are created and run in order by Executor during run-time.

9.2.5 C. Result reconstruction

Upon completion of the secure training (or inference) job, the models (or prediction results) will be output by CPs in encrypted form. Result Parties can collect the encrypted results, decrypt them using the tools in PFM, and deliver the plaintext results to users.

CHAPTER

TEN

ON GOING AND FUTURE WORK

- Vertial Federated Learning will support more algorithms.
- Add K8S deployment scheme for Paddle Encrypted.
- FL mobile simulator will be open sourced in following versions.

10.1 Example in recommendation with FedAvg

This document introduces how to use PaddleFL to train a model with Fl Strategy.

10.1.1 Dependencies

- paddlepaddle>=1.8

10.1.2 How to install PaddleFL

Please use pip which has paddlepaddle installed

```
pip install paddle_fl
```

10.1.3 Model

Gru4rec is a classical session-based recommendation model. Detailed implementations with paddlepaddle is here.

10.1.4 Datasets

Public Dataset Rsc15

```
#download data
sh download.sh
```

10.1.5 How to work in PaddleFL

PaddleFL has two phases , CompileTime and RunTime. In CompileTime, a federated learning task is defined by fl_master. In RunTime, a federated learning job is executed on fl_server and fl_trainer in distributed clusters.

```
sh run.sh
```

10.1.6 How to work in CompileTime

In this example, we implement compile time programs in fl_master.py

```
# please run fl_master to generate fl_job
python fl_master.py
```

In fl_master.py, we first define FL-Strategy, User-Defined-Program and Distributed-Config. Then FL-Job-Generator generate FL-Job for federated server and worker.

```
import paddle.fluid as fluid
import paddle_fl.paddle_fl as fl
from paddle_fl.paddle_fl.core.master.job_generator import JobGenerator
from paddle_fl.paddle_fl.core.strategy.fl_strategy_base import FLStrategyFactory

# define model
model = Model()
model.gru4rec_network()

# define JobGenerator and set model config
# feed_name and target_name are config for save model.
job_generator = JobGenerator()
optimizer = fluid.optimizer.SGD(learning_rate=2.0)
job_generator.set_optimizer(optimizer)
job_generator.set_losses([model.loss])
job_generator.set_startup_program(model.startup_program)
job_generator.set_infer_feed_and_target_names(
    [x.name for x in model.inputs], [model.loss.name, model.recall.name])

# define FL-Strategy , we now support two flstrategy, fed_avg and dpsgd. Inner_step␣
↪means fl_trainer locally train inner_step mini-batch.
build_strategy = FLStrategyFactory()
build_strategy.fed_avg = True
build_strategy.inner_step = 1
strategy = build_strategy.create_fl_strategy()

# define Distributed-Config and generate fl_job
endpoints = ["127.0.0.1:8181"]
output = "fl_job_config"
job_generator.generate_fl_job(
    strategy, server_endpoints=endpoints, worker_num=4, output=output)
```

10.1.7 How to work in RunTime

```
python -u fl_scheduler.py >scheduler.log &
python -u fl_server.py >server0.log &
python -u fl_trainer.py 0 >trainer0.log &
```

(continues on next page)

(continued from previous page)

```
python -u fl_trainer.py 1 >trainer1.log &
python -u fl_trainer.py 2 >trainer2.log &
python -u fl_trainer.py 3 >trainer3.log &
```

fl_trainer.py can define own reader according to data.

```
r = Gru4rec_Reader()
train_reader = r.reader(train_file_dir, place, batch_size=10)
```

10.1.8 Simulated experiments on real world dataset

To show the concept and effectiveness of horizontal federated learning with PaddleFL, a simulated experiment is conducted on an open source dataset with a real world task. In horizontal federated learning, a group of organizations are doing similar tasks based on private dataset and they are willing to collaborate on a certain task. The goal of the collaboration is to improve the task accuracy with federated learning.

The simulated experiment suppose all organizations have homogeneous dataset and homogeneous task which is an ideal case. The whole dataset is from small part of [Rsc15] and each organization has a subset as a private dataset. To show the performanc e improvement under federated learning, models based on each organization's private dataset are trained and a model under distributed federated learning is trained. A model based on traditional parameter server training is also trained where the whole dataset is owned by a single organization.

From the table and the figure given below, model evaluation results are similar between federated learning and traditional parameter server training. It is clear that compare with models trained with only private dataset, models' performance for each organization get significant improvement with federated learning.

```
# download code and readme
wget https://paddle-zwh.bj.bcebos.com/gru4rec_paddlefl_benchmark/gru4rec_benchmark.tar
```

Dataset	training methods	FL Strategy	recall@20
the whole dataset	private training	•	0.504
the whole dataset	federated learning	FedAvg	0.504
1/4 of the whole dataset	private training	•	0.286
1/4 of the whole dataset	private training	•	0.277
1/4 of the whole dataset	private training	•	0.269
1/4 of the whole dataset	private training	•	0.282

10.2 Example in Recognize Digits with DPSGD

This document introduces how to use PaddleFL to train a model with Fl Strategy: DPSGD.

10.2.1 Dependencies

- paddlepaddle>=1.8

10.2.2 How to install PaddleFL

Please use pip which has paddlepaddle installed

```
pip install paddle_fl
```

10.2.3 Model

The simplest Softmax regression model is to get features with input layer passing through a fully connected layer and then compute and ouput probabilities of multiple classifications directly via Softmax function [PaddlePaddle tutorial: recognize digits].

10.2.4 Datasets

Public Dataset MNIST

The dataset will downloaded automatically in the API and will be located under `/home/username/.cache/paddle/dataset/mnist`:

filename	note
train-images-idx3-ubyte	train data picture, 60,000 data
train-labels-idx1-ubyte	train data label, 60,000 data
t10k-images-idx3-ubyte	test data picture, 10,000 data
t10k-labels-idx1-ubyte	test data label, 10,000 data

10.2.5 How to work in PaddleFL

PaddleFL has two phases , CompileTime and RunTime. In CompileTime, a federated learning task is defined by fl_master. In RunTime, a federated learning job is executed on fl_server and fl_trainer in distributed clusters.

```
sh run.sh
```

How to work in CompileTime

In this example, we implement compile time programs in fl_master.py

```
python fl_master.py
```

In fl_master.py, we first define FL-Strategy, User-Defined-Program and Distributed-Config. Then FL-Job-Generator generate FL-Job for federated server and worker.

```
import paddle.fluid as fluid
import paddle_fl.paddle_fl as fl
from paddle_fl.paddle_fl.core.master.job_generator import JobGenerator
from paddle_fl.paddle_fl.core.strategy.fl_strategy_base import FLStrategyFactory
```

(continues on next page)

(continued from previous page)

```
import math

class Model(object):
    def __init__(self):
        pass

    def lr_network(self):
        self.inputs = fluid.layers.data(name='img', shape=[1, 28, 28], dtype="float32
↪")
        self.label = fluid.layers.data(name='label', shape=[1],dtype='int64')
        self.predict = fluid.layers.fc(input=self.inputs, size=10, act='softmax')
        self.sum_cost = fluid.layers.cross_entropy(input=self.predict, label=self.
↪label)
        self.accuracy = fluid.layers.accuracy(input=self.predict, label=self.label)
        self.loss = fluid.layers.mean(self.sum_cost)
        self.startup_program = fluid.default_startup_program()


model = Model()
model.lr_network()

STEP_EPSILON = 0.1
DELTA = 0.00001
SIGMA = math.sqrt(2.0 * math.log(1.25/DELTA)) / STEP_EPSILON
CLIP = 4.0
batch_size = 64

job_generator = JobGenerator()
optimizer = fluid.optimizer.SGD(learning_rate=0.1)
job_generator.set_optimizer(optimizer)
job_generator.set_losses([model.loss])
job_generator.set_startup_program(model.startup_program)
job_generator.set_infer_feed_and_target_names(
    [model.inputs.name, model.label.name], [model.loss.name, model.accuracy.name])

build_strategy = FLStrategyFactory()
build_strategy.dpsgd = True
build_strategy.inner_step = 1
strategy = build_strategy.create_fl_strategy()
strategy.learning_rate = 0.1
strategy.clip = CLIP
strategy.batch_size = float(batch_size)
strategy.sigma = CLIP * SIGMA

# endpoints will be collected through the cluster
# in this example, we suppose endpoints have been collected
endpoints = ["127.0.0.1:8181"]
output = "fl_job_config"
job_generator.generate_fl_job(
    strategy, server_endpoints=endpoints, worker_num=2, output=output)
```

How to work in RunTime

```
python -u fl_scheduler.py >scheduler.log &
python -u fl_server.py >server0.log &
```

(continues on next page)

(continued from previous page)

```
python -u fl_trainer.py 0 >trainer0.log &
python -u fl_trainer.py 1 >trainer1.log &
python -u fl_trainer.py 2 >trainer2.log &
python -u fl_trainer.py 3 >trainer3.log &
```

In fl_scheduler.py, we let server and trainers to do registeration.

```
from paddle_fl.paddle_fl.core.scheduler.agent_master import FLScheduler

worker_num = 4
server_num = 1
#Define number of worker/server and the port for scheduler
scheduler = FLScheduler(worker_num, server_num, port=9091)
scheduler.set_sample_worker_num(4)
scheduler.init_env()
print("init env done.")
scheduler.start_fl_training()
```

In fl_server.py, we load and run the FL server job.

```
import paddle_fl.paddle_fl as fl
import paddle.fluid as fluid
from paddle_fl.paddle_fl.core.server.fl_server import FLServer
from paddle_fl.paddle_fl.core.master.fl_job import FLRunTimeJob

server = FLServer()
server_id = 0
job_path = "fl_job_config"
job = FLRunTimeJob()
job.load_server_job(job_path, server_id)
job._scheduler_ep = "127.0.0.1:9091"  # IP address for scheduler
server.set_server_job(job)
server._current_ep = "127.0.0.1:8181"  # IP address for server
server.start()
```

In fl_trainer.py, we load and run the FL trainer job, then evaluate the accuracy with test data and compute the privacy budget.

```
import numpy
import sys
import paddle
import paddle.fluid as fluid
import logging
import math
from paddle_fl.paddle_fl.core.master.fl_job import FLRunTimeJob
from paddle_fl.paddle_fl.core.trainer.fl_trainer import FLTrainerFactory

trainer_id = int(sys.argv[1]) # trainer id for each guest
job_path = "fl_job_config"
job = FLRunTimeJob()
job.load_trainer_job(job_path, trainer_id)
trainer = FLTrainerFactory().create_fl_trainer(job)
trainer.start()

def train_test(train_test_program, train_test_feed, train_test_reader):
        acc_set = []
```

(continues on next page)

(continued from previous page)

```
        for test_data in train_test_reader():
            acc_np = trainer.exe.run(
                program=train_test_program,
                feed=train_test_feed.feed(test_data),
                fetch_list=["accuracy_0.tmp_0"])
            acc_set.append(float(acc_np[0]))
        acc_val_mean = numpy.array(acc_set).mean()
        return acc_val_mean

def compute_privacy_budget(sample_ratio, epsilon, step, delta):
    E = 2 * epsilon * math.sqrt(step * sample_ratio)
    print("({0}, {1})-DP".format(E, delta))

output_folder = "model_node%d" % trainer_id
epoch_id = 0
step = 0

while not trainer.stop():
    epoch_id += 1
    if epoch_id > 40:
        break
    print("epoch %d start train" % (epoch_id))
    for step_id, data in enumerate(train_reader()):
        acc = trainer.run(feeder.feed(data), fetch=["accuracy_0.tmp_0"])
        step += 1
    # print("acc:%.3f" % (acc[0]))

    acc_val = train_test(
        train_test_program=test_program,
        train_test_reader=test_reader,
        train_test_feed=feeder)

    print("Test with epoch %d, accuracy: %s" % (epoch_id, acc_val))
    compute_privacy_budget(sample_ratio=0.001, epsilon=0.1, step=step, delta=0.00001)

    save_dir = (output_folder + "/epoch_%d") % epoch_id
    trainer.save_inference_program(output_folder)
```

10.2.6 Simulated experiments on public dataset MNIST

To show the effectiveness of DPSGD-based federated learning with PaddleFL, a simulated experiment is conducted on an open source dataset MNIST. From the figure given below, model evaluation results are similar between DPSGD-based federated learning and traditional parameter server training when the overall privacy budget *epsilon* is 1.3 or 0.13.

10.3 Instructions for PaddleFL-MPC UCI Housing Demo

(|English)

This document introduces how to run UCI Housing demo based on Paddle-MPC, which has two ways of running, i.e., single machine and multi machines.

10.3.1 1. Running on Single Machine

(1). Prepare Data

Generate encrypted data utilizing `generate_encrypted_data()` in `process_data.py` script. For example, users can write the following code into a python script named `prepare.py`, and then run the script with command `python prepare.py`.

```
import process_data
process_data.generate_encrypted_data()
```

Encrypted data files of feature and label would be generated and saved in `/tmp` directory. Different suffix names are used for these files to indicate the ownership of different computation parties. For instance, a file named `house_feature.part0` means it is a feature file of party 0.

(2). Launch Demo with A Shell Script

Launch demo with the `run_standalone.sh` script. The concrete command is:

```
bash run_standalone.sh uci_housing_demo.py
```

The loss with cypher text format will be displayed on screen while training. At the same time, the loss data would be also save in `/tmp` directory, and the format of file name is similar to what is described in Step 1.

Besides, predictions would be made in this demo once training is finished. The predictions with cypher text format would also be save in `/tmp` directory.

Finally, using `load_decrypt_data()` in `process_data.py` script, this demo would decrypt and print the loss and predictions, which can be compared with related results of Paddle plain text model.

Note that remember to delete the loss and prediction files in `/tmp` directory generated in last running, in case of any influence on the decrypted results of current running. For simplifying users operations, we provide the following commands in `run_standalone.sh`, which can delete the files mentioned above when running this script.

```
# remove temp data generated in last time
LOSS_FILE="/tmp/uci_loss.*"
PRED_FILE="/tmp/uci_prediction.*"
if [ "$LOSS_FILE" ]; then
        rm -rf $LOSS_FILE
fi

if [ "$PRED_FILE" ]; then
        rm -rf $PRED_FILE
fi
```

10.3.2 2. Running on Multi Machines

(1). Prepare Data

Data owner encrypts data. Concrete operations are consistent with “Prepare Data” in “Running on Single Machine”.

(2). Distribute Encrypted Data

According to the suffix of file name, distribute encrypted data files to `/tmp` directories of all 3 computation parties. For example, send `house_feature.part0` and `house_label.part0` to `/tmp` of party 0 with `scp` command.

(3). Modify uci_housing_demo.py

Each computation party makes the following modifications on `uci_housing_demo.py` according to the environment of machine.

- Modify IP Information

  Modify `localhost` in the following code as the IP address of the machine.

```
pfl_mpc.init("aby3", int(role), "localhost", server, int(port))
```

- Comment Out Codes for Single Machine Running

  Comment out the following codes which are used when running on single machine.

```
import process_data
print("uci_loss:")
process_data.load_decrypt_data("/tmp/uci_loss", (1,))
print("prediction:")
process_data.load_decrypt_data("/tmp/uci_prediction", (BATCH_SIZE,))
```

(4). Launch Demo on Each Party

Note that Redis service is necessary for demo running. Remember to clear the cache of Redis server before launching demo on each computation party, in order to avoid any negative influences caused by the cached records in Redis. The following command can be used for clear Redis, where REDIS_BIN is the executable binary of redis-cli, SERVER and PORT represent the IP and port of Redis server respectively.

```
$REDIS_BIN -h $SERVER -p $PORT flushall
```

Launch demo on each computation party with the following command,

```
$PYTHON_EXECUTABLE uci_housing_demo.py $PARTY_ID $SERVER $PORT
```

where PYTHON_EXECUTABLE is the python which installs PaddleFL, PARTY_ID is the ID of computation party, which is 0, 1, or 2, SERVER and PORT represent the IP and port of Redis server respectively.

Similarly, training loss with cypher text format would be printed on the screen of each computation party. And at the same time, the loss and predictions would be saved in `/tmp` directory.

Note that remember to delete the loss and prediction files in `/tmp` directory generated in last running, in case of any influence on the decrypted results of current running.

(5). Decrypt Loss and Prediction Data

Each computation party sends `uci_loss.part` and `uci_prediction.part` files in `/tmp` directory to the `/tmp` directory of data owner. Data owner decrypts and gets the plain text of loss and predictions with `load_decrypt_data()` in `process_data.py`.

For example, the following code can be written into a python script to decrypt and print training loss.

```
import process_data
print("uci_loss:")
process_data.load_decrypt_data("/tmp/uci_loss", (1,))
```

And the following code can be written into a python script to decrypt and print predictions.

```
import process_data
print("prediction:")
process_data.load_decrypt_data("/tmp/uci_prediction", (BATCH_SIZE,))
```

Convergence of paddle_fl.mpc vs paddle

Below, is the result of our experiment to test the convergence of paddle_fl.mpc

A. Convergence of paddle_fl.mpc vs paddle

1. Training Parameters

- Dataset: Boston house price dataset
- Number of Epoch: 20
- Batch Size: 10

2. Experiment Results

Epoch/Step	paddle_fl.mpc	Paddle
Epoch=0, Step=0	738.39491	738.46204
Epoch=1, Step=0	630.68834	629.9071
Epoch=2, Step=0	539.54683	538.1757
Epoch=3, Step=0	462.41159	460.64722
Epoch=4, Step=0	397.11516	395.11017
Epoch=5, Step=0	341.83102	339.69815
Epoch=6, Step=0	295.01114	292.83597
Epoch=7, Step=0	255.35141	253.19429
Epoch=8, Step=0	221.74739	219.65132
Epoch=9, Step=0	193.26459	191.25981
Epoch=10, Step=0	169.11423	167.2204
Epoch=11, Step=0	148.63138	146.85835
Epoch=12, Step=0	131.25081	129.60391
Epoch=13, Step=0	116.49708	114.97599
Epoch=14, Step=0	103.96669	102.56854
Epoch=15, Step=0	93.31706	92.03858
Epoch=16, Step=0	84.26219	83.09653
Epoch=17, Step=0	76.55664	75.49785
Epoch=18, Step=0	69.99673	69.03561
Epoch=19, Step=0	64.40562	63.53539

CHAPTER

ELEVEN

THE TEAM

11.1 The Team

PaddleFL is developed by PaddlePaddle and Security team.

PaddleFL is developed by PaddlePaddle and Security team.

11.2 Reference

[1]. Jakub Konečný, H. Brendan McMahan, Daniel Ramage, Peter Richtárik. **Federated Optimization: Distributed Machine Learning for On-Device Intelligence.** 2016

[2]. H. Brendan McMahan, Eider Moore, Daniel Ramage, Blaise Agüera y Arcas. **Federated Learning of Deep Networks using Model Averaging.** 2017

[3]. Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, Dave Bacon. **Federated Learning: Strategies for Improving Communication Efficiency.** 2016

[4]. Qiang Yang, Yang Liu, Tianjian Chen, Yongxin Tong. **Federated Machine Learning: Concept and Applications.** 2019

[5]. Kai He, Liu Yang, Jue Hong, Jinghua Jiang, Jieming Wu, Xu Dong et al. **PrivC - A framework for efficient Secure Two-Party Computation.** In Proc. of SecureComm 2019

[6]. Martín Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, Li Zhang. **Deep Learning with Differential Privacy.** 2016

[7]. Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, Ameet Talwalkar. **Federated Multi-Task Learning** 2016

[8]. Yang Liu, Tianjian Chen, Qiang Yang. **Secure Federated Transfer Learning.** 2018

[9]. Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, Domonkos Tikk. **Session-based Recommendations with Recurrent Neural Networks.** 2016

[10]. https://en.wikipedia.org/wiki/Secret_sharing

[11]. Payman Mohassel and Peter Rindal. **ABY3: A Mixed Protocol Framework for Machine Learning.** In Proc. of CCS 2018

11.3 Reference

[1]. Jakub Konečný, H. Brendan McMahan, Daniel Ramage, Peter Richtárik. **Federated Optimization: Distributed Machine Learning for On-Device Intelligence.** 2016

[2]. H. Brendan McMahan, Eider Moore, Daniel Ramage, Blaise Agüera y Arcas. **Federated Learning of Deep Networks using Model Averaging.** 2017

[3]. Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, Dave Bacon. **Federated Learning: Strategies for Improving Communication Efficiency.** 2016

[4]. Qiang Yang, Yang Liu, Tianjian Chen, Yongxin Tong. **Federated Machine Learning: Concept and Applications.** 2019

[5]. Kai He, Liu Yang, Jue Hong, Jinghua Jiang, Jieming Wu, Xu Dong et al. **PrivC - A framework for efficient Secure Two-Party Computation.** In Proc. of SecureComm 2019

[6]. Martín Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, Li Zhang. **Deep Learning with Differential Privacy.** 2016

[7]. Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, Ameet Talwalkar. **Federated Multi-Task Learning** 2016

[8]. Yang Liu, Tianjian Chen, Qiang Yang. **Secure Federated Transfer Learning.** 2018

[9]. Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, Domonkos Tikk. **Session-based Recommendations with Recurrent Neural Networks.** 2016

[10]. https://en.wikipedia.org/wiki/Secret_sharing

[11]. Payman Mohassel and Peter Rindal. **ABY3: A Mixed Protocol Framework for Machine Learning.** In Proc. of CCS 2018

CHAPTER

TWELVE

LICENSE

PaddleFL uses Apache License 2.0.