EPFL

Decentralized Learning Made Easy With DecentralizePy

Akash Dhasade   Anne-Marie Kermarrec   Rafael Pires   **Rishi Sharma**   Milos Vujasinovic

Scalable Computing Systems Laboratory
EPFL

08.05.2023

DecentralizePy

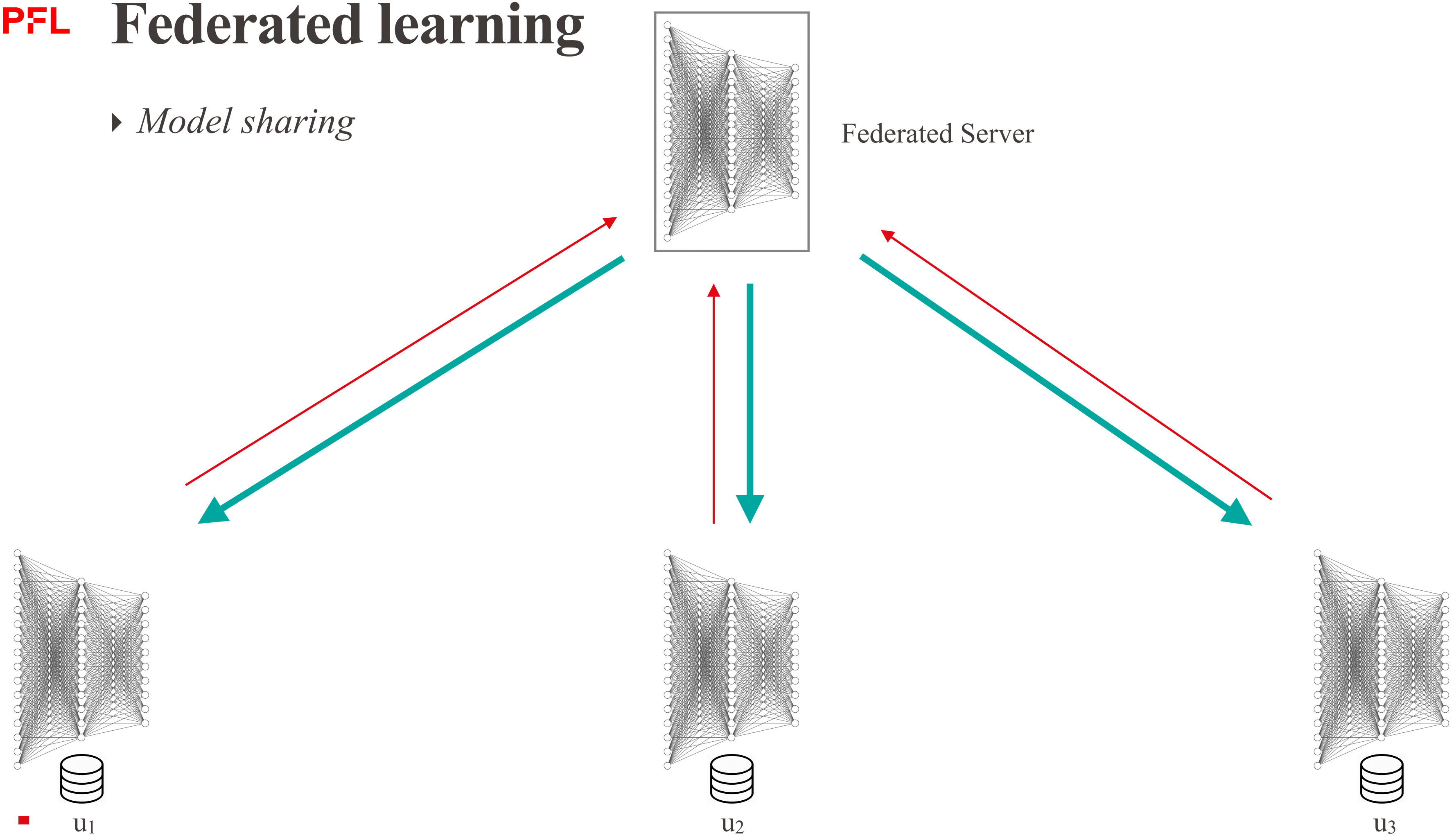
A framework for designing and studying decentralized learning systems.

Rapid development

Scalability

Federated learning

- *Model sharing*



EPFL

So many frameworks…

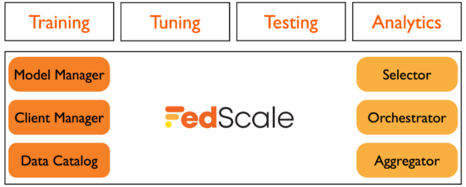
FedScale

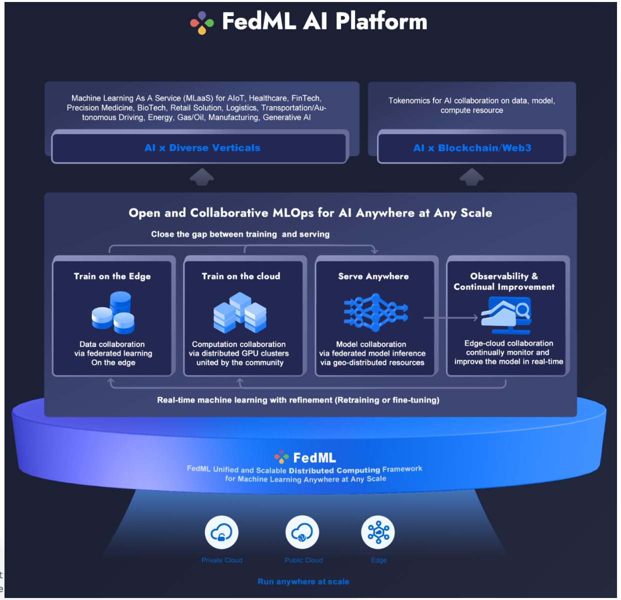
A scalable and extensible federated learning engine and benchmark

GET STARTED

FedScale is a scalable and extensible open-source federated learning (FL) engine. It provides high-level APIs to implement FL algorithms, deploy and evaluate them at scale across diverse hardware and software backends. FedScale also includes the largest FL benchmark that contains FL tasks ranging from image classification and object detection to language modeling and speech recognition. Moreover, it includes datasets to faithfully emulate FL runtime environments where FL solutions will realistically be deployed.

We are actively developing FedScale, and welcome contributions fror community. **Join our slack to keep up to date.**





Flower

What's new? Flower Next Pilot Program >

Flower A Friendly Federated Learning Framework

A unified approach to federated learning, analytics, and evaluation. Federate any workload, any ML framework, and any programming language.

Take the tutorial — to learn federated learning

TensorFlow Federated: Machine Learning on Decentralized Data

TensorFlow Federated (TFF) is an open-source framework for machine learning and other computations on decentralized data. TFF has been developed to facilitate open research and experimentation with Federated Learning (FL), an approach to machine learning where a shared global model is trained across many participating clients that keep their training data locally. For example, FL has been used to train prediction models for mobile keyboards without uploading sensitive typing data to servers.

TFF enables developers to simulate the included federated learning algorithms on their models and data, as well as to experiment with novel algorithms. Researchers will find starting points and complete examples for many kinds of research. The building blocks provided by TFF can also be used to implement non-learning computations, such as federated analytics. TFF's interfaces are organized in two main layers:

Federated Learning (FL) API
This layer offers a set of high-level interfaces that allow developers to apply the included implementations of federated training and evaluation to their existing TensorFlow models.

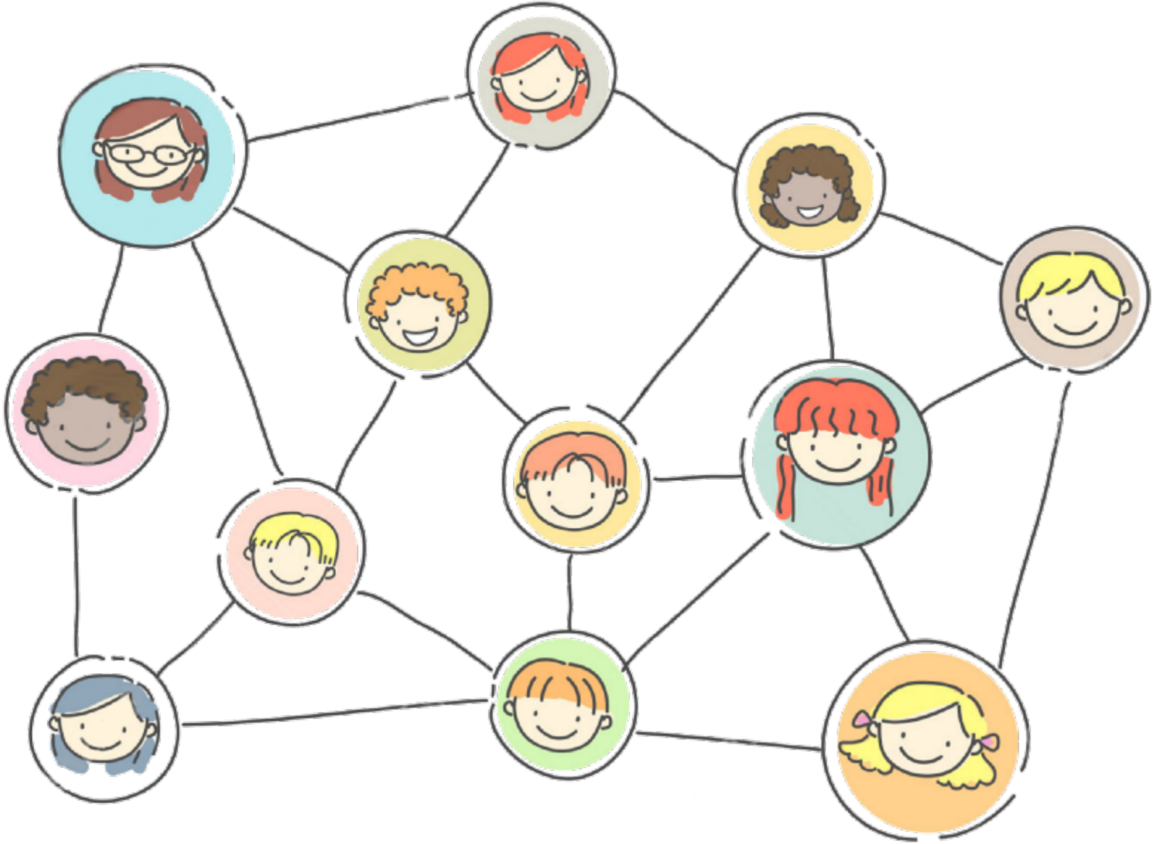
Federated Core (FC) API
At the core of the system is a set of lower-level interfaces for concisely expressing novel federated algorithms by combining TensorFlow with distributed communication operators within a strongly-typed functional programming environment. This layer also serves as the foundation upon which we've built Federated Learning.

TFF enables developers to declaratively express federated computations, so they could be deployed to diverse runtime environments. Included with TFF is a performant multi-machine simulation runtime for experiments. Please visit the tutorials and try it out yourself!

For questions and support, find us at the tensorflow-federated tag on StackOverflow.

```
import tensorflow as t
import tensorflow_fede

# Load simulation data.
source, _ = tff.simulation.datasets.emnist.load_data()
def client_data(n):
  return source.create_tf_dataset_for_client(source.client_ids[n]).map(
      lambda e: (tf.reshape(e['pixels'], [-1]), e['label'])
  ).repeat(10).batch(20)

# Pick a subset of client devices to participate in training.
train_data = [client_data(n) for n in range(3)]

# Wrap a Keras model for use with TFF.
def model_fn():
  model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(10, tf.nn.softmax, input_shape=(784,),
                            kernel_initializer='zeros')
  ])
  return tff.learning.models.from_keras_model(
      model,
      input_spec=train_data[0].element_spec,
      loss=tf.keras.losses.SparseCategoricalCrossentropy(),
      metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

# Simulate a few rounds of training with the selected client devices.
trainer = tff.learning.algorithms.build_weighted_fed_avg(
  model_fn,
  client_optimizer_fn=lambda: tf.keras.optimizers.SGD(0.1))
state = trainer.initialize()
for _ in range(5):
```

Decentralized learning

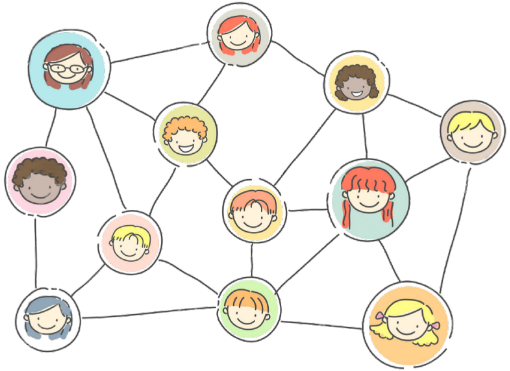


- Train → Send | Receive → Aggregate

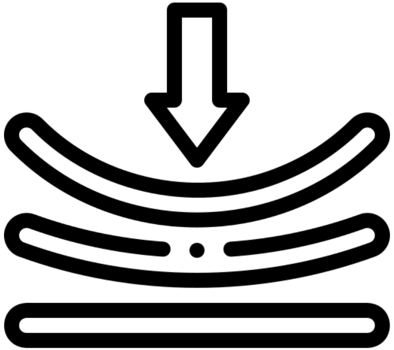
Decentralized learning

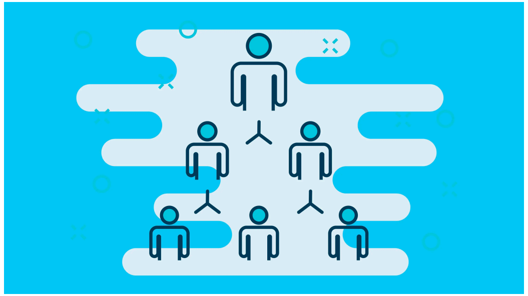
Simulations + **No Re-usability**!

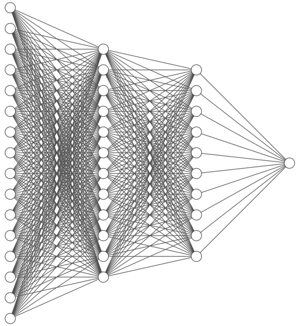
Building decentralized learning systems

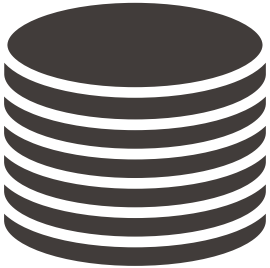

Topology


Communication


Compression

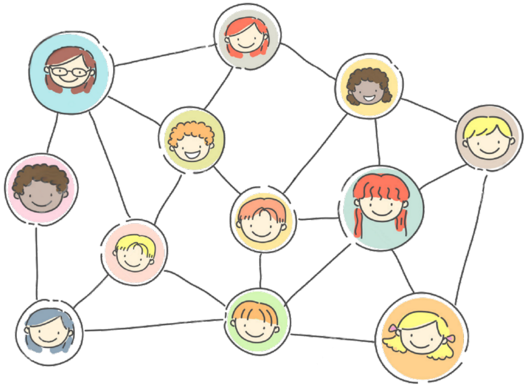

Roles

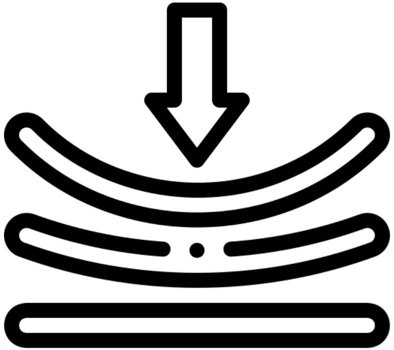

Models


Datasets

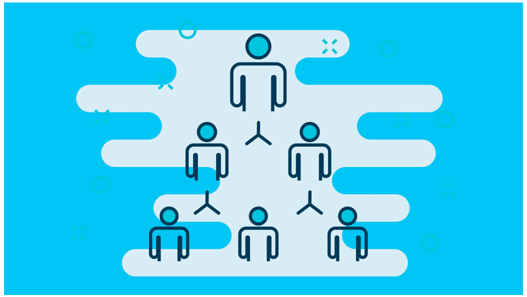
flaticon.com / freecodecamp.org / studiousguy.org

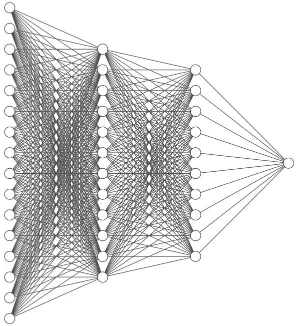
DecentralizePy Modules

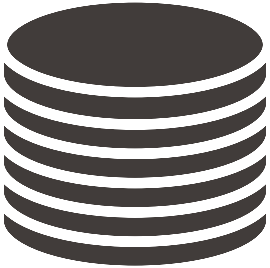




(Flexibility)







flaticon.com / freecodecamp.org / studiousguy.org

EPFL

As easy as ABC …


```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
            sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```

DecentralizePy already contains reference implementations of well-known algorithms.

Development Phase: Single machine

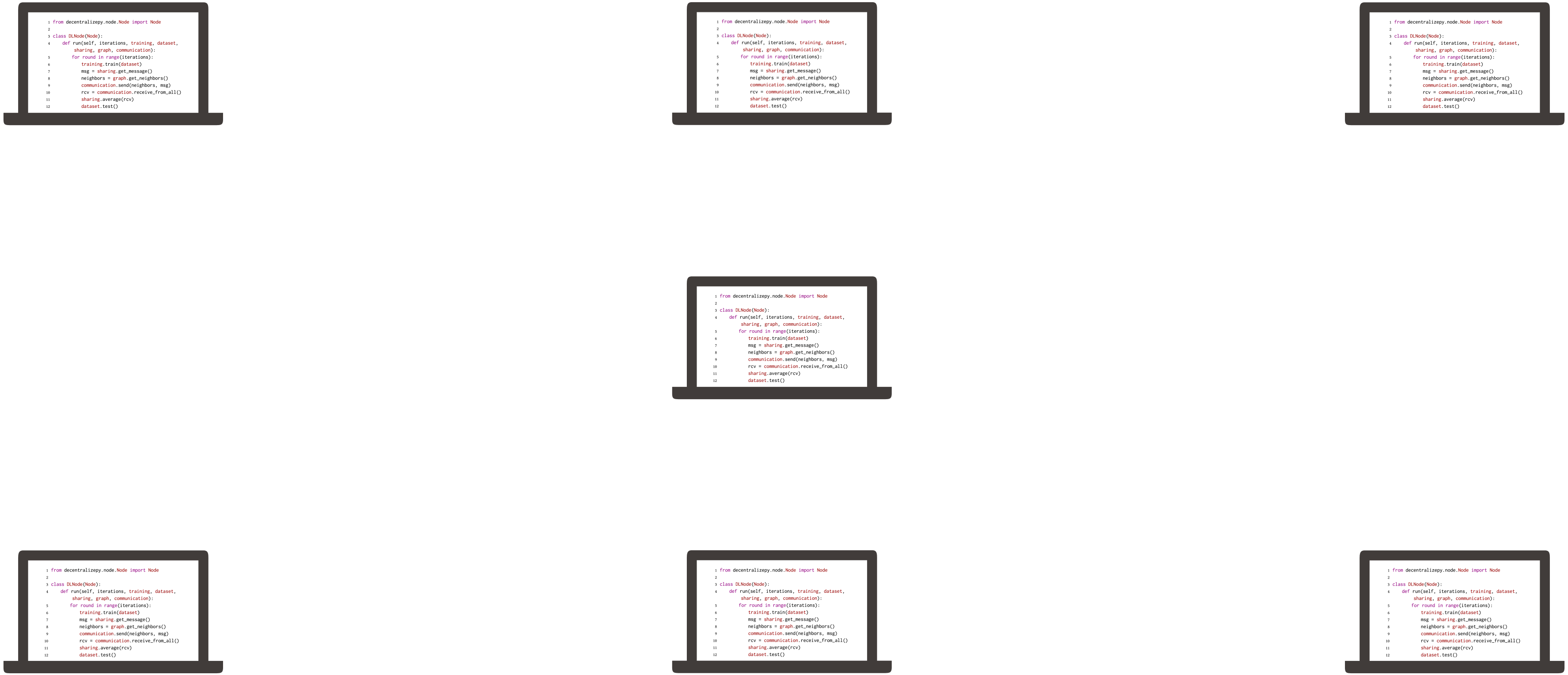
```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
         sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```


```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
         sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```

```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
         sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```

```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
         sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```

Real-world deployment



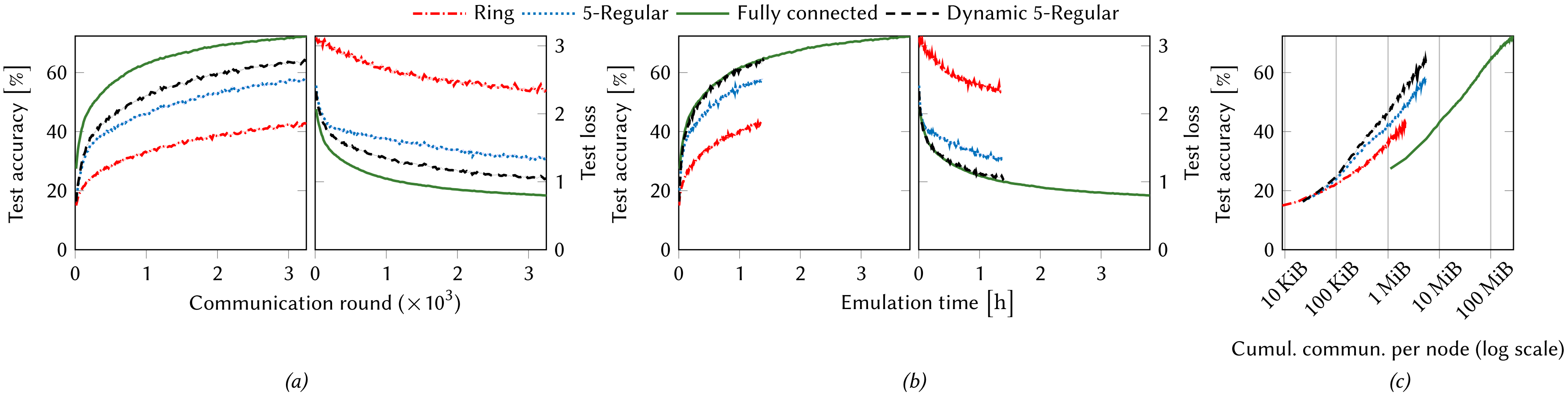
We use DecentralizePy as a **catalyst** for DL research in our lab.

Experimental Setup

- CIFAR-10 (Non-IID) with GN-LeNet
- 256 and 1024 DL nodes
- Emulation on 16 machines
- D-PSGD with Metropolis Hastings

Topologies

(256-nodes)

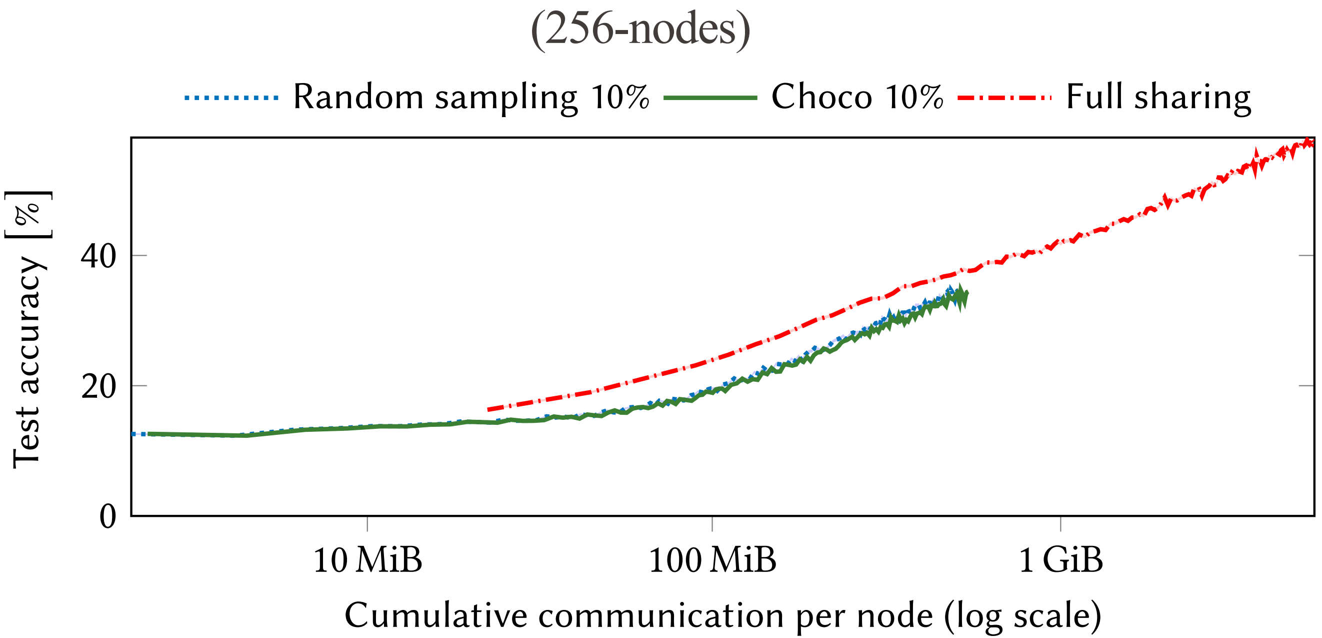


Information spreads faster through the network with dynamic topologies.

Topologies

```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
            sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```

Communication Compression



The loss of information due to compression dramatically affects the convergence in non-IID settings at scale.

Communication Compression

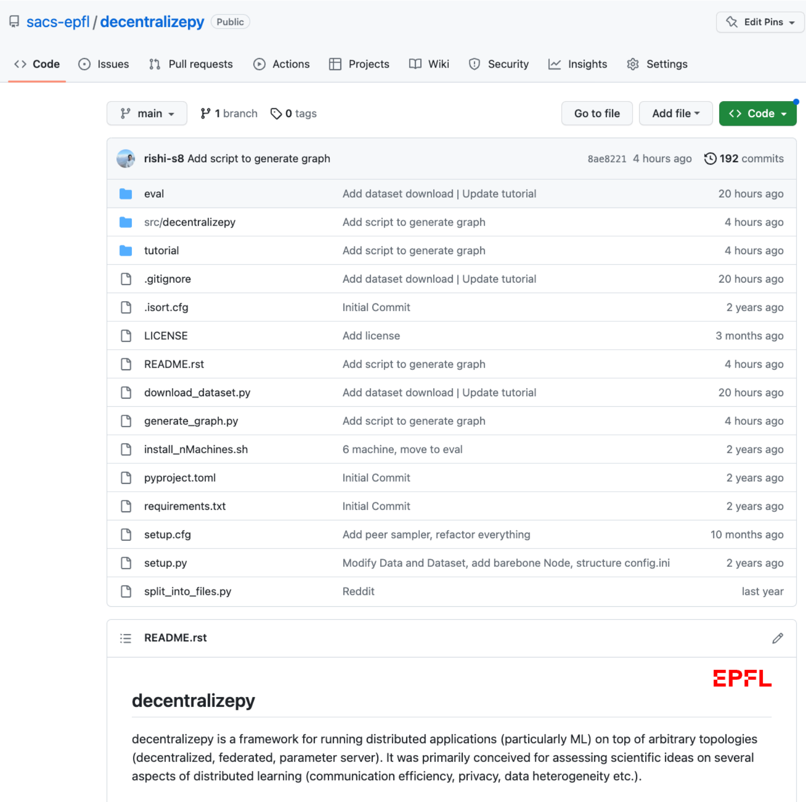
```python
from decentralizepy.node.Node import Node

class DLNode(Node):
    def run(self, iterations, training, dataset,
            sharing, graph, communication):
        for round in range(iterations):
            training.train(dataset)
            msg = sharing.get_message()
            neighbors = graph.get_neighbors()
            communication.send(neighbors, msg)
            rcv = communication.receive_from_all()
            sharing.average(rcv)
            dataset.test()
```

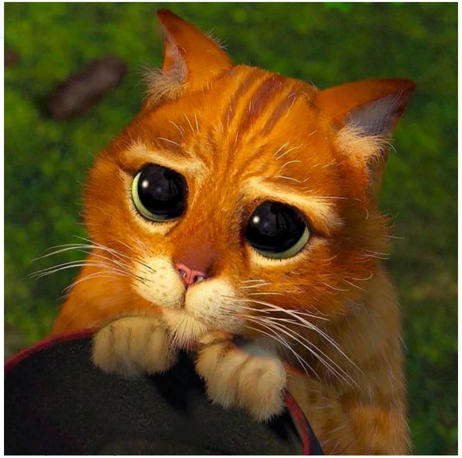
DecentralizePy is rapidly evolving

- Open source
- Already being used for a number of projects
- Adding new algorithms
- Realistic network emulations
- Peer-sampling and availability traces

https://github.com/sacs-epfl/decentralizepy

DecentralizePy is rapidly evolving





Please try DecentralizePy if you are working with DL and help us improve the framework.

EPFL

Go Decentralized!

https://github.com/sacs-epfl/decentralizepy