ECML PKDD 2024

UNIVERSITÀ DI TORINO

`fluke`: **f**ederated **l**earning **u**tility framewor**k** for **e**xperimentation and research

Mirko Polato, PhD

Department of Computer Science, University of Turin, Italy

mirko.polato@unito.it

SCAN ME

2nd **W**orkshop on **A**dvancement in **F**ederated **L**earning @ ECML PKDD 2024

Yet another FL framework?

Not really!

from the idea

through the implementation

to testing

Yet another FL framework?

Not really!

fluke

from the idea

through the implementation

to testing

✔ simulated federation

✘ deploy

Main features of `fluke`

Designed for fast prototyping and testing

https://github.com/makgyver/fluke

- **Open source**: `fluke` is an open-source Python package;
- **Easy to use**: `fluke` is designed to be extremely easy to use out of the box;
- **Easy to extend**: fluke is designed to minimize the overhead of adding new algorithms;
- **Up-to-date**: `fluke` comes with several (30+) state-of-the-art federated learning algorithms and datasets and it is regularly updated to include the latest affirmed techniques;
- **Easy to read**: the source code of the algorithms is written to mimic as close as possible the description in the reference papers.

SCAN ME

`fluke` is on PyPi!

Install it using a single command

```
$ pip install fluke-fl
```

… or by cloning the repo

```
$ git clone https://github.com/makgyver/fluke.git
$ cd fluke
$ pip install -r requirements.txt
```

fluke CLI

You can run your experiment outright with a single command!

path to the experiment configuration file

path to the algorithm configuration file

```
$ fluke --config exp.yaml federation algorithm.yaml
```

*

fluke command

required option to set the configuration file for the experiment

type of experiment sub-command

(*) If you cloned the repo, the command (launched from the fluke folder) is `$ python -m fluke.run –config exp.yaml federation algorithm.yaml`

Configuration files

Two YAML files for everything you want to configure

exp.yaml

```yaml
data:
  dataset:
    name: mnist
    path: ./data
  distribution:
    name: iid
  ...

exp:
  device: cpu
  seed: 42

eval:
  ...

logger:
  name: Log

protocol:
  eligible_perc: 0.2
  n_clients: 10
  n_rounds: 10
```

- data: dataset and data distribution configuration
- exp: general settings
- eval: evaluation settings
- logger: logger setting
- protocol: federated protocol configuration

algorithm.yaml

```yaml
hyperparameters:

  client:
    batch_size: 32
    local_epochs: 5
    loss: CrossEntropyLoss
    optimizer:
      lr: 0.1
    scheduler:
      gamma: 1
      step_size: 1

  server:
    weighted: true

  model: MyNeuralNet

name: fluke.algorithms.fedavg.FedAVG
```

- client: client-side hyper-parameters
- server: server-side hyper-parameters
- model: model to be learned
- name: algorithm name

Default configuration files can be downloaded using the command: `fluke-get`

Configuration files

Two YAML files for everything you want to configure

exp.yaml

```yaml
data:
  dataset:
    name: mnist
    path: ./data
  distribution:
    name: iid
  ...

exp:
  device: cpu
  seed: 42

eval:
  ...

logger:
  name: Log

protocol:
  eligible_perc: 0.2
  n_clients: 10
  n_rounds: 10
```

- data: dataset and data distribution configuration
- exp: general settings
- eval: evaluation settings
- logger: logger setting
- protocol: federated protocol configuration

algorithm.yaml

```yaml
hyperparameters:
  client:
    batch_size: 32
    local_epochs: 5
    loss: CrossEntropyLoss
    optimizer:
      lr: 0.1
    scheduler:
      gamma: 1
      step_size: 1
  server:
    weighted: true
  model: MyNeuralNet
name: fluke.algorithms.fedavg.FedAVG
```

- client: client-side hyper-parameters
- server: server-side hyper-parameters
- model: model to be learned
- name: algorithm name

Default configuration files can be downloaded using the command: `fluke-get`

`fluke` CLI – not only federation

You can run the "same" experiment without the federation for comparison

same experiment but without the federation -
the number of epochs client-side are calculated*
as `n_rounds * eligible_perc * local_epochs`

```
$ fluke --config exp.yaml clients-only algorithm.yaml
```

```
$ fluke --config exp.yaml centralized algorithm.yaml
```

same type of experiment but with all the dataset centralised

(*) The number of epochs can be set by the user via an option of the command, e.g., `–epochs=100`

Example: FedAVG on MNIST

Fluke comes with many downloadable configuration files ready to be used/modified

downloads the default experiment configuration file (named `exp.yaml`) to `./config`

downloads the default fedavg configuration file (named `fedavg.yaml`) to `./config`

```
$ fluke-get config exp
$ fluke-get config fedavg
$ fluke --config ./config/exp.yaml federation ./config/fedavg.yaml
```

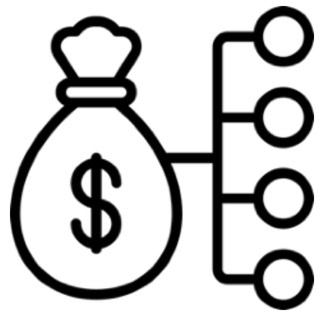
*

runs the federated algorithm specified in `fedavg.yaml` on the dataset specified in `exp.yaml`

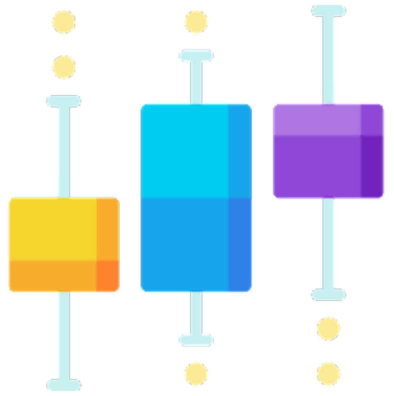
(*) If you want to know all the available default configuration files use the command `$ fluke-get list`

fluke logging

Performance can be logged on your preferred tool



communication cost*



classification performance**
(e.g., accuracy, precision, recall, F1 – marco and micro)

system performance***

TensorBoard

W&B

CLEAR|ML

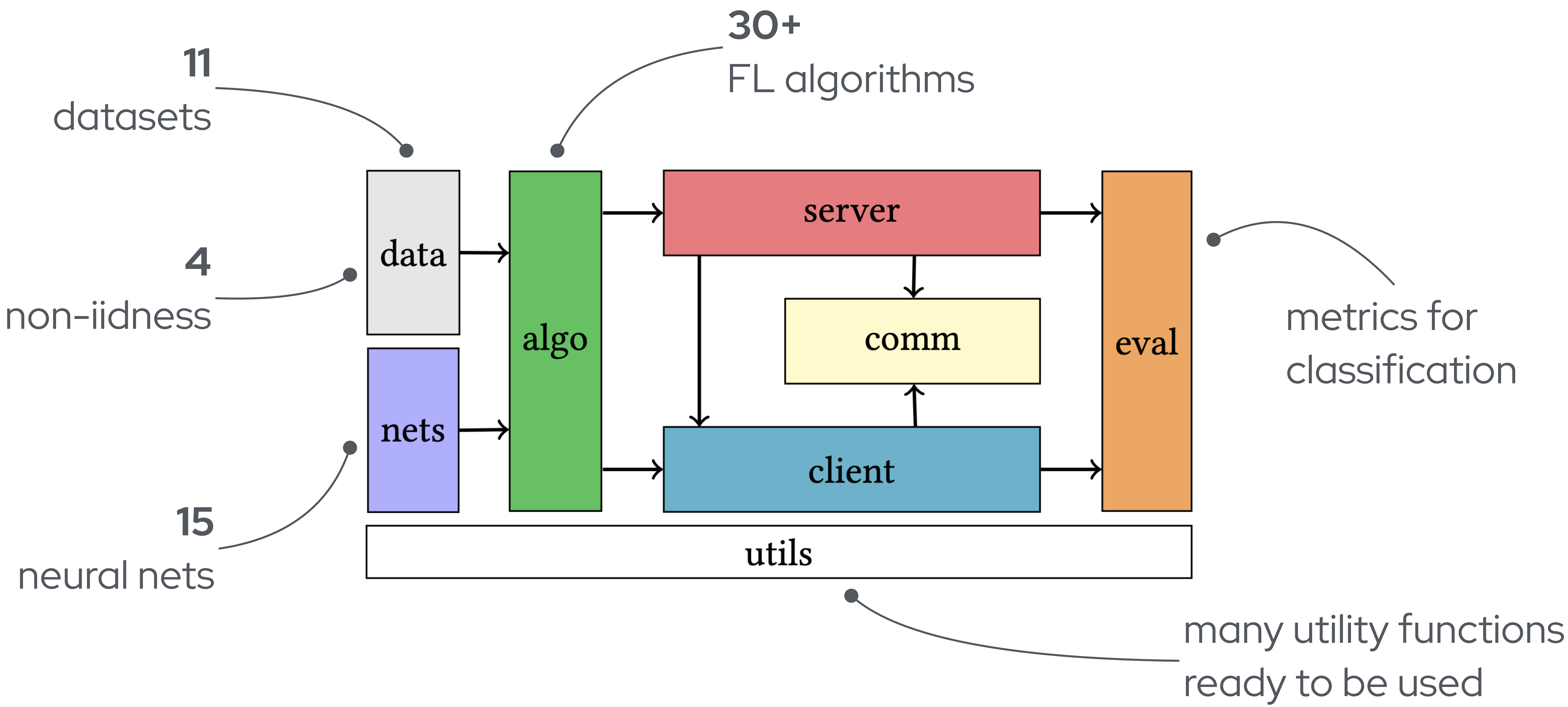
(*) The communication cost is estimated as the number fo floating points numbers exchanged by the entire federation.
(**) Currently, fluke supports only classification but it is straightforward to extend to other tasks.
(***) System performance are automatically logged by W&B and ClearML.

fluke python API

https://github.com/makgyver/fluke

SCAN ME

The python API offers all you need to implement and test your FL ideas



`fluke` server

Code readability is a key feature of `fluke`

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:

initialize w_0

for each round $t = 1, 2, \dots$ **do**

$m \leftarrow \max(C \cdot K, 1)$

$S_t \leftarrow$ (random set of m clients)

for each client $k \in S_t$ **in parallel do**

$w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$

$m_t \leftarrow \sum_{k \in S_t} n_k$

$w_{t+1} \leftarrow \sum_{k \in S_t} \frac{n_k}{m_t} w_{t+1}^k$

Server *

```python
def fit(self, n_rounds: int, eligible_perc: float) -> None:

    for round in range(n_rounds):
        eligible = self.get_eligible_clients(eligible_perc)
        self.broadcast_model(eligible)

        for c, client in enumerate(eligible):
            client.local_update(round + 1)

        self.aggregate(eligible)
```

McMahan et al. Communication-Efficient Learning of Deep Networks from Decentralized Data. AISTATS 2017.

(*) This is a polish version of the fluke source code.

fluke client

Code readability is a key feature of `fluke`

ClientUpdate(k, w)**:** *// Run on client k*

$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)

for each local epoch i from 1 to E **do**

  for batch $b \in \mathcal{B}$ **do**

    $w \leftarrow w - \eta \nabla \ell(w; b)$

return w to server

Client *

```python
def local_update(self, current_round: int):
    self.receive_model()
    self.fit()
    self.send_model()

def fit(self, override_local_epochs: int = 0):
    self.model.train()

    if self.optimizer is None:
        self.optimizer, self.scheduler = self.optimizer_cfg(self.model)

    for _ in range(epochs):
        for _, (X, y) in enumerate(self.train_set):
            X, y = X.to(self.device), y.to(self.device)
            self.optimizer.zero_grad()
            y_hat = self.model(X)
            loss = self.hyper_params.loss_fn(y_hat, y)
            loss.backward()
            self.optimizer.step()
        self.scheduler.step()

def receive_model(self):
    msg = self.channel.receive(self, self.server, msg_type="model")
    self.model.load_state_dict(msg.payload.state_dict())

def send_model(self):
    self.channel.send(Message(self.model, "model", self), self.server)
```

McMahan et al. Communication-Efficient Learning of Deep Networks from Decentralized Data. AISTATS 2017.

(*) This is a slightly polish version of the fluke source code.

fluke python API - Dataset loading & splitting

Loading the dataset

```python
from fluke.data.datasets import Datasets
dataset = Datasets.get("mnist", path="./data")
```

dataset to (down)load*

folder where the dataset will be stored/loaded

Splitting the dataset

```python
from fluke.data import DataSplitter
splitter = DataSplitter(dataset=dataset, distribution="iid")
```

the dataset to split

type of non-iidness

(*) fluke currently supports the following built-in datasets: MNIST, MNIST-M, SVHN, FEMNIST, EMNIST, CIFAR10, CIFAR100, Tiny Imagenet, Shakespeare, Fashion MNIST, and CINIC10.

fluke python API – Federated algorithm

Preparing the algorithm

```python
from fluke.algorithms.fedavg import FedAVG
algorithm = FedAVG(100, splitter, hyperparams)
```

the federated learning algorithm*

number of clients

the data splitter which will distribute the data among clients

the hyper-parameters of the **algorithm**

client server model

(*) fluke currently includes 31 different federated algorithms that you can use off-the-shelf.

fluke python API - Hyper-parameters

You can load the hyper-parameters from file or hard-coding them

Hyper-parameters

```python
from fluke import DDict

client_hp = DDict(
    batch_size=10,
    local_epochs=5,
    loss="CrossEntropyLoss",
    optimizer=DDict(name="SGD", lr=0.01)
)

server_hp = DDict(weighted=True)

# we put together the hyperparameters
hyperparams = DDict(client=client_hp,
                    server=server_hp,
                    model="MNIST_2NN")
```

Hyper-parameters from file

```python
import yaml

with open("myconfig.yaml") as f:
    config_alg = yaml.safe_load(f)

hyperparams = DDict(**config_alg)
```

client specific hyper-parameters, including the optimizer and the scheduler

server specific hyper-parameters

the shared model class*

(*) fluke can also handle cases where clients and server own different models (like sub-network of the overall model)

fluke python API - Logging

Logging is handled using callbacks (i.e., design pattern Observer)

this is the default logger (on console) but you can also log on ***W&B***, ***Tensorboard*** or ***ClearML***

the loggers in fluke only log evaluation results

Logging

```python
from fluke.utils.log import Log
logger = Log()
algorithm.set_callbacks(logger)
```

you can add all the observers you like.
you can observer the clients, the server and the communication channel

fluke python API - Start the training

Run the algorithm

```python
algorithm.run(n_rounds=2, eligible_perc=0.5)
```

Log on console

```
Round: 1
{
    'global': {
        'accuracy': 0.8872,
        'micro_precision': 0.8872,
        'micro_recall': 0.8872,
        'micro_f1': 0.8872,
        'macro_precision': 0.88576,
        'macro_recall': 0.88476,
        'macro_f1': 0.88447
    },
    'comm_cost': 17811000
}
```

Log on your preferred tool

TensorBoard

CLEAR | ML

W&B

You can save and restore the FL training

fluke - Adding a new FL algorithm

You just need to implement the core part of your algorithm as described in the paper

Define your client

```python
from fluke.client import Client

class MyClient(Client):

    def __init__(index: int,
                 train_set: FastDataLoader,
                 test_set: FastDataLoader,
                 optimizer_cfg: OptimizerConfigurator,
                 loss_fn: Module,
                 local_epochs: int = 3,
                 **kwargs: dict[str, Any]):
        ...

    def receive_model(self) -> None:
        ...

    def send_model(self) -> None:
        ...

    def fit(self, override_local_epochs: int = 0) -> float:
        ...

```

Define your server

```python
from fluke.server import Server

class Myserver(Server):

    def __init__(self,
                 model: torch.nn.Module,
                 test_set: FastDataLoader,
                 clients: Iterable[Client],
                 weighted: bool = False,
                 **kwargs: dict[str, Any]):
        ...

    def aggregate(self, eligible: Iterable[Client]) -> None:
        ...
```

Define your algorithm

```python
class MyFLAlgo(CentralizedFL):

    def get_server_class(self):
        return MyServer

    def get_client_class(self):
        return MyClient
```

You just need to implement what characterise your FL algorithm

Implementing Kafè in `fluke`

Kafè has been presented at ECML PKDD 2024!

typical client selection process, that is already implemented in `fluke.server.Server`*

this must be implemented !!

this is the standard behaviour of a FedAVG client, that is already implemented in `fluke.client.Client`

Algorithm 1 KAFÈ

Require: T communication rounds, E local epochs, B local batchsize, h bandwith of KDE, m number of clients participating in aggregation.

Server execute:
  Initialize model w^0
  for $t = 1, ...T$ **do**
    $m \leftarrow max([C] \times K, 1)$
    $S_m \leftarrow$ random selection of m clients
    Send $w^{(t-1)}$ to all clients.
    for chosen client $k \in S_m$ in parallel **do**
      $w_k^{f,(t)}, w_k^{c,(t)} \leftarrow$ **LocalUpdating**$(w^{(t-1)})$

    Model aggregation:
      $w_g^{f,(t)} \leftarrow \sum_{k \in S_m} \frac{n_k}{n} w_k^{f,(t)}$.
      $w_g^{c,(t)} \leftarrow$ **KAFÈ**$(h, \frac{n_k}{n} w_k^{c,(t)})$.
    Update $w_g^{(t)} = (w_g^{f,(t)}, w_g^{c,(t)})$
    end for
  end for

LocalUpdating$(w^{(t-1)})$:
  for $e = 1, 2, ...E - 1$ **do**
    for each batch B **do**
      $w_k^{(t)} \leftarrow w^{(t-1)} - \eta \nabla \ell(w^{(t-1)}, b)$.
    end for
  end for
  return $w_k^{(t)} = (w_k^{f,(t)}, w_k^{c,(t)})$

Pian Qi , et al. KAFÈ: Kernel Aggregation for FEderated. In ECML-PKDD 2024

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

S_m

receive the client models $(w_k^{f,(t)}, w_k^{c,(t)})$

Model aggregation:

$$w_g^{f,(t)} \leftarrow \sum_{k \in S_m} \frac{n_k}{n} w_k^{f,(t)}.$$

$$w_g^{c,(t)} \leftarrow \textbf{KAFÈ}\left(h, \frac{n_k}{n} w_k^{c,(t)}\right).$$

Update $w_g^{(t)} = (w_g^{f,(t)}, w_g^{c,(t)})$

Kafè Server

```python
class KafeServer(Server):

    def aggregate(self, eligible: Iterable[Client]) -> None:
        avg_model_sd = OrderedDict()
        clients_sd = self.get_client_models(eligible)
        weights = self._get_client_weights(eligible)

        # get last layer of m clients' weights
        last_layer_weight_name = list(clients_sd[0].keys())[-2]
        last_layer_bias_name = list(clients_sd[0].keys())[-1]

        for key in self.model.state_dict().keys():
            if key in (last_layer_weight_name, last_layer_bias_name):
                continue
            for i, client_sd in enumerate(clients_sd):
                if key not in avg_model_sd:
                    avg_model_sd[key] = weights[i] * client_sd[key]
                else:
                    avg_model_sd[key] = avg_model_sd[key] + weights[i] * client_sd[key]
```

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

$$w_g^{c,(t)} \leftarrow \mathbf{KAF\grave{E}}(h, \frac{n_k}{n} w_k^{c,(t)}).$$

Classification layers aggregation. The m classification layers of the m local models can be denoted as $\{w_1^{c,(t)}, w_2^{c,(t)}, ..., w_k^{c,(t)}, ...\}$. To evaluate the probability density function $\hat{f}(\cdot)$ for these classification layers w^c, we employ KDE as follows:

$$\hat{f}(w^c) = \frac{1}{mh^d} \sum_{k \in S_m^{(t)}} K\Big(\frac{w_k^{c,(t)} - w^c}{h}\Big) \tag{5}$$

where d denotes the dimension of the samples, similar to Eq. (1). In this context, d corresponds to the dimension of $w_k^{c,(t)}$ after flattening. To account for the varying contributions of the classification layers from different local models, we introduce weighting factors $\frac{n_k}{n}$ to the KDE formulation. Consequently, Eq. (5) is transformed into Eq. (6).

$$\hat{f}(w^c) = \frac{1}{h^d} \sum_{k \in S_m^{(t)}} \frac{n_k}{n} K\Big(\frac{w_k^{c,(t)} - w^c}{h}\Big) \tag{6}$$

Next, by sampling m new samples from the KDE and averaging them, we obtain the new classification layer $w_g^{c,(t)}$ for the global model. Lastly, the global

Kafè Server

```python
class KafeServer(Server):

    def __init__(self,
                 model: torch.nn.Module,
                 test_set: FastDataLoader,
                 clients: Iterable[Client],
                 weighted: bool = False,
                 bandwidth: float = 1.0):
        super().__init__(model=model, test_set=test_set,
                         clients=clients, weighted=weighted)
        self.hyper_params.update(bandwidth=bandwidth)
```

```python
        w_last_layer = []
        b_last_layer = []

        for csd in clients_sd:
            w_last_layer.append(np.array(csd[last_layer_weight_name]))
            b_last_layer.append(np.array(csd[last_layer_bias_name]))

        w_last_layer = np.array(w_last_layer).reshape(len(w_last_layer), -1)
        b_last_layer = np.array(b_last_layer).reshape(len(b_last_layer), -1)

        # using KDE get the kernel density of last layers
        kde_w = KernelDensity(kernel='gaussian',
                              bandwidth=self.hyper_params.bandwidth).fit(w_last_layer,
                                                                         sample_weight=weights)
        kde_b = KernelDensity(kernel='gaussian',
                              bandwidth=self.hyper_params.bandwidth).fit(b_last_layer,
                                                                         sample_weight=weights)

        # sample m samples and average, then obtain a new last layer for the global model
        w_last_layer_new = np.mean(kde_w.sample(len(w_last_layer)), axis=0)
        b_last_layer_new = np.mean(kde_b.sample(len(b_last_layer)), axis=0)

        # update last layer
        avg_model_sd[last_layer_weight_name] = torch.tensor(w_last_layer_new.reshape(
            clients_sd[0][last_layer_weight_name].shape))
        avg_model_sd[last_layer_bias_name] = torch.tensor(b_last_layer_new.reshape(
            clients_sd[0][last_layer_bias_name].shape))

        self.model.load_state_dict(avg_model_sd)
```

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

Kafè Server

```python
class KafeServer(Server):

    def __init__(self,
                 model: torch.nn.Module,
                 test_set: FastDataLoader,
                 clients: Iterable[Client],
                 weighted: bool = False,
                 bandwidth: float = 1.0):
        super().__init__(model=model, test_set=test_set,
                         clients=clients, weighted=weighted)
        self.hyper_params.update(bandwidth=bandwidth)
```

$$w_g^{c,(t)} \leftarrow \textbf{KAF\`E}(h, \tfrac{n_k}{n} w_k^{c,(t)}).$$

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

$$w_g^{c,(t)} \leftarrow \textbf{KAFÈ}(h, \frac{n_k}{n} w_k^{c,(t)}).$$

Classification layers aggregation. The m classification layers of the m local models can be denoted as $\{w_1^{c,(t)}, w_2^{c,(t)}, ..., w_k^{c,(t)}, ...\}$. To evaluate the probability density function $\hat{f}(\cdot)$ for these classification layers w^c, we employ KDE as follows:

$$\hat{f}(w^c) = \frac{1}{mh^d} \sum_{k \in S_m^{(t)}} K\left(\frac{w_k^{c,(t)} - w^c}{h}\right) \tag{5}$$

where d denotes the dimension of the samples, similar to Eq. (1). In this context, d corresponds to the dimension of $w_k^{c,(t)}$ after flattening. To account for the varying contributions of the classification layers from different local models, we introduce weighting factors $\frac{n_k}{n}$ to the KDE formulation. Consequently, Eq. (5) is transformed into Eq. (6).

$$\hat{f}(w^c) = \frac{1}{h^d} \sum_{k \in S_m^{(t)}} \frac{n_k}{n} K\left(\frac{w_k^{c,(t)} - w^c}{h}\right) \tag{6}$$

Next, by sampling m new samples from the KDE and averaging them, we obtain the new classification layer $w_g^{c,(t)}$ for the global model. Lastly, the global

Kafè Server

```python
class KafeServer(Server):

    def __init__(self,
                 model: torch.nn.Module,
                 test_set: FastDataLoader,
                 clients: Iterable[Client],
                 weighted: bool = False,
                 bandwidth: float = 1.0):
        super().__init__(model=model, test_set=test_set,
                         clients=clients, weighted=weighted)
        self.hyper_params.update(bandwidth=bandwidth)
```

```python
        w_last_layer = []
        b_last_layer = []

        for csd in clients_sd:
            w_last_layer.append(np.array(csd[last_layer_weight_name]))
            b_last_layer.append(np.array(csd[last_layer_bias_name]))

        w_last_layer = np.array(w_last_layer).reshape(len(w_last_layer), -1)
        b_last_layer = np.array(b_last_layer).reshape(len(b_last_layer), -1)

        # using KDE get the kernel density of last layers
        kde_w = KernelDensity(kernel='gaussian',
                              bandwidth=self.hyper_params.bandwidth).fit(w_last_layer,
                                                                          sample_weight=weights)
        kde_b = KernelDensity(kernel='gaussian',
                              bandwidth=self.hyper_params.bandwidth).fit(b_last_layer,
                                                                          sample_weight=weights)

        # sample m samples and average, then obtain a new last layer for the global model
        w_last_layer_new = np.mean(kde_w.sample(len(w_last_layer)), axis=0)
        b_last_layer_new = np.mean(kde_b.sample(len(b_last_layer)), axis=0)

        # update last layer
        avg_model_sd[last_layer_weight_name] = torch.tensor(w_last_layer_new.reshape(
            clients_sd[0][last_layer_weight_name].shape))
        avg_model_sd[last_layer_bias_name] = torch.tensor(b_last_layer_new.reshape(
            clients_sd[0][last_layer_bias_name].shape))

        self.model.load_state_dict(avg_model_sd)
```

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

context, d corresponds to the dimension of $w_k^{c,(t)}$ after flattening. To account for the varying contributions of the classification layers from different local models, we introduce weighting factors $\frac{n_k}{n}$ to the KDE formulation. Consequently, Eq. (5) is transformed into Eq. (6).

$$\hat{f}(w^c) = \frac{1}{h^d} \sum_{k \in S_m^{(t)}} \frac{n_k}{n} K\left(\frac{w_k^{c,(t)} - w^c}{h}\right) \tag{6}$$

```
kde_w = KernelDensity(kernel='gaussian',
                      bandwidth=self.hyper_params.bandwidth).fit(w_last_layer,
                                                                 sample_weight=weights)
kde_b = KernelDensity(kernel='gaussian',
                      bandwidth=self.hyper_params.bandwidth).fit(b_last_layer,
                                                                 sample_weight=weights)
```

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

$$w_g^{c,(t)} \leftarrow \textbf{KAFÈ}(h, \frac{n_k}{n} w_k^{c,(t)}).$$

Classification layers aggregation. The m classification layers of the m local models can be denoted as $\{w_1^{c,(t)}, w_2^{c,(t)}, ..., w_k^{c,(t)}, ...\}$. To evaluate the probability density function $\hat{f}(\cdot)$ for these classification layers w^c, we employ KDE as follows:

$$\hat{f}(w^c) = \frac{1}{mh^d} \sum_{k \in S_m^{(t)}} K(\frac{w_k^{c,(t)} - w^c}{h}) \tag{5}$$

where d denotes the dimension of the samples, similar to Eq. (1). In this context, d corresponds to the dimension of $w_k^{c,(t)}$ after flattening. To account for the varying contributions of the classification layers from different local models, we introduce weighting factors $\frac{n_k}{n}$ to the KDE formulation. Consequently, Eq. (5) is transformed into Eq. (6).

$$\hat{f}(w^c) = \frac{1}{h^d} \sum_{k \in S_m^{(t)}} \frac{n_k}{n} K(\frac{w_k^{c,(t)} - w^c}{h}) \tag{6}$$

Next, by sampling m new samples from the KDE and averaging them, we obtain the new classification layer $w_g^{c,(t)}$ for the global model. Lastly, the global

Kafè Server

```python
class KafeServer(Server):

    def __init__(self,
                 model: torch.nn.Module,
                 test_set: FastDataLoader,
                 clients: Iterable[Client],
                 weighted: bool = False,
                 bandwidth: float = 1.0):
        super().__init__(model=model, test_set=test_set,
                         clients=clients, weighted=weighted)
        self.hyper_params.update(bandwidth=bandwidth)
```

```python
        w_last_layer = []
        b_last_layer = []

        for csd in clients_sd:
            w_last_layer.append(np.array(csd[last_layer_weight_name]))
            b_last_layer.append(np.array(csd[last_layer_bias_name]))

        w_last_layer = np.array(w_last_layer).reshape(len(w_last_layer), -1)
        b_last_layer = np.array(b_last_layer).reshape(len(b_last_layer), -1)

        # using KDE get the kernel density of last layers
        kde_w = KernelDensity(kernel='gaussian',
                              bandwidth=self.hyper_params.bandwidth).fit(w_last_layer,
                                                                         sample_weight=weights)
        kde_b = KernelDensity(kernel='gaussian',
                              bandwidth=self.hyper_params.bandwidth).fit(b_last_layer,
                                                                         sample_weight=weights)

        # sample m samples and average, then obtain a new last layer for the global model
        w_last_layer_new = np.mean(kde_w.sample(len(w_last_layer)), axis=0)
        b_last_layer_new = np.mean(kde_b.sample(len(b_last_layer)), axis=0)

        # update last layer
        avg_model_sd[last_layer_weight_name] = torch.tensor(w_last_layer_new.reshape(
            clients_sd[0][last_layer_weight_name].shape))
        avg_model_sd[last_layer_bias_name] = torch.tensor(b_last_layer_new.reshape(
            clients_sd[0][last_layer_bias_name].shape))

        self.model.load_state_dict(avg_model_sd)
```

Implementing Kafè in `fluke`

Kafè is a FL algorithm presented at ECML PKDD 2024!

Next, by sampling m new samples from the KDE and averaging them, we obtain the new classification layer $w_g^{c,(t)}$ for the global model. Lastly, the global

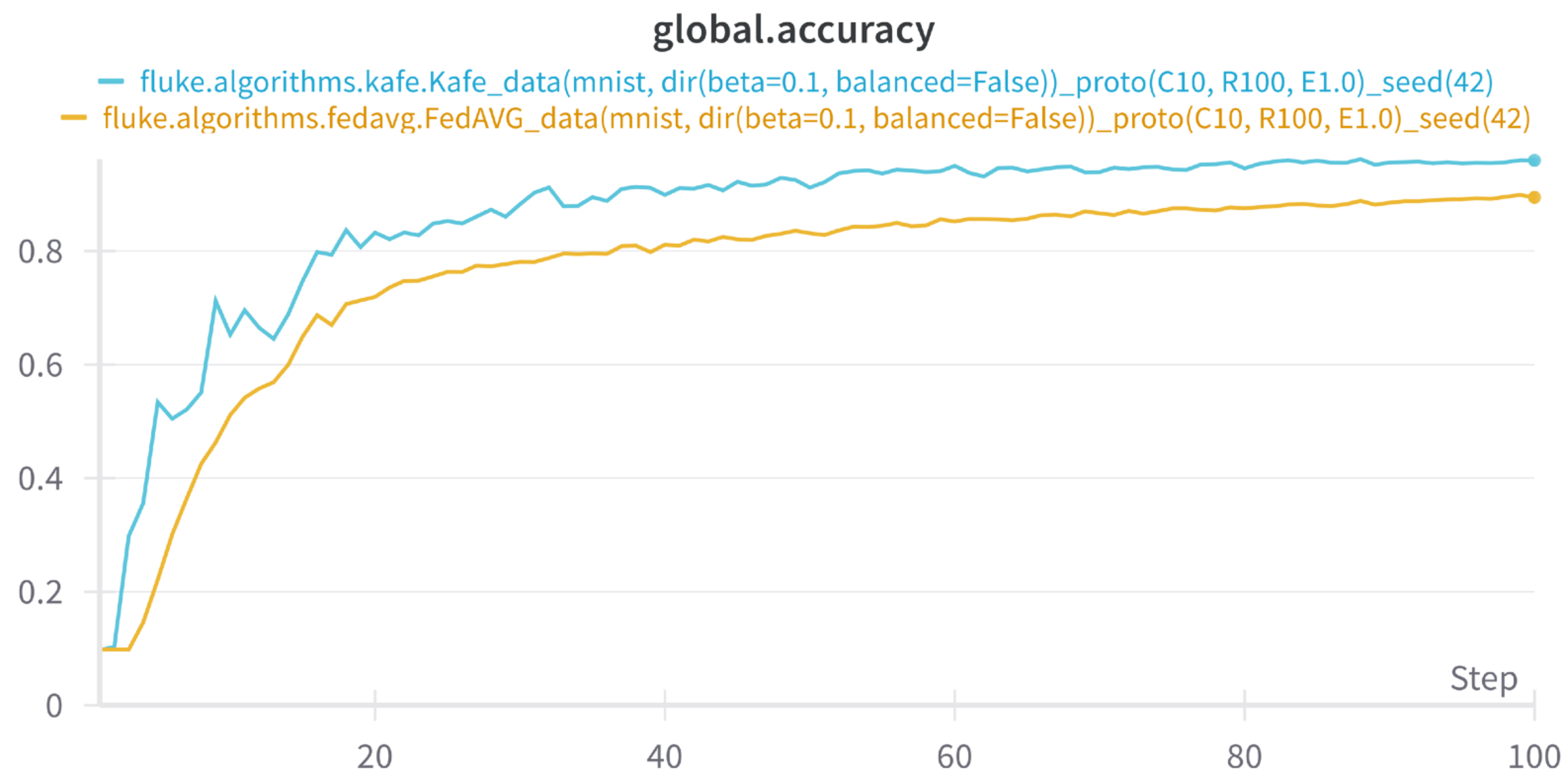
```
w_last_layer_new = np.mean(kde_w.sample(len(w_last_layer)), axis=0)
b_last_layer_new = np.mean(kde_b.sample(len(b_last_layer)), axis=0)

# update last layer
avg_model_sd[last_layer_weight_name] = torch.tensor(w_last_layer_new.reshape(
    clients_sd[0][last_layer_weight_name].shape))
avg_model_sd[last_layer_bias_name] = torch.tensor(b_last_layer_new.reshape(
    clients_sd[0][last_layer_bias_name].shape))

self.model.load_state_dict(avg_model_sd)
```

Testing Kafè in `fluke`

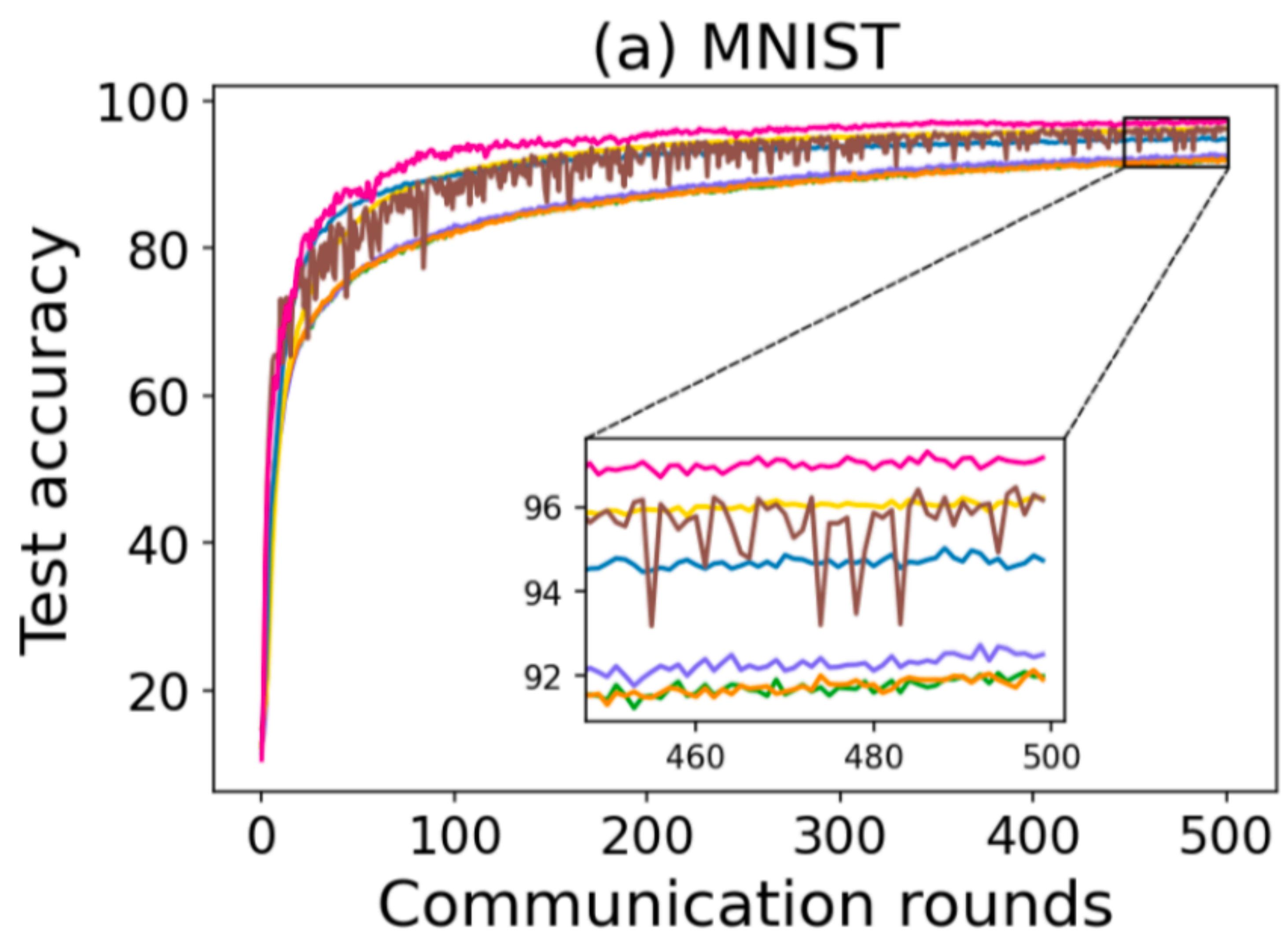
You just need to define the configuration files and use the fluke CLI

Running Kafè

```
1 $ fluke --config exp.yaml federation kafe.yaml
```



fluke results (W&B plot)



Paper's result

Give it a try, you won't regret it!

fluke 0.3.0 is now available!

30+
FL algorithms

11
datasets

15
neural networks

propose your FL algorithm
to be included in `fluke`!

SCAN ME

https://github.com/makgyver/fluke

https://makgyver.github.io/fluke/

Credits

All the icons have been downloaded from www.flaticon.com

[quick] Icon made by Cuputo from www.flaticon.com
[chart] Icon made by Freepik from www.flaticon.com
[idea] Icon made by Freepik from www.flaticon.com
[network cost] Icon made by juicy_fish from www.flaticon.com
[performance] Icon made by juicy_fish from www.flaticon.com
[programming] Icon made by juicy_fish from www.flaticon.com
[close] Icon made by Pixel Perfect from www.flaticon.com
[check-mark] Icon made by Ian June from www.flaticon.com
[pause] Icon made by bqlqn from www.flaticon.com
[jigsaw] Icon made by monkik from www.flaticon.com