(Vellore Campus)

School of Computer Science and Engineering (SCOPE)

Comparison & Optimization of federated learning algorithms in edge computing

Project Under

Dr. UMADEVI K S

Team Details

19BCT0081	A N Loganathan
19BCT0149	B Kushwanth Sai
19BCT0158	Dhanesh Shetty

Winter Semester 2021-22 Fundamentals of Fog and Edge Computing G1/TG1

TABLE OF CONTENTS

Fundamentals of Fog and Edge Computing

Abstract:

Federated Learning (FL) is a new method for training Deep Neural Networks (DNNs) cooperatively on mobile devices without exposing private user data. Previous research has demonstrated that non-Independent and Identically Distributed (non-IID) user data slows down the FL algorithms' convergence speed. Furthermore, the majority of extant FL research evaluates global-model performance. In many circumstances, such as user content recommendation, the primary goal is to improve individual User model Accuracy (UA).

To address these issues, we propose a FL algorithm that introduces non-federated Batch-Normalisation (BN) layers into the federated DNN. This benefits UA and convergence speed by allowing users to train models personalised to their own data. This is compatible with popular iterative FL optimisation algorithms such as Federated Averaging (FedAvg), and we show empirically that a distributed form of Adam optimisation (FedAvg-Adam) benefits convergence speed even further when used as the optimisation strategy within the algorithm.

Finally, we evaluate the performance of fed avg, fedadam, fedavg-adam algorithms for edge computing scenarios.

Introduction:

Multi access Edge Computing is used to move the cloud services to network edge enabling low latency and real time processing of applications using computational offloading.

Deep Neural Networks (DNNs) for Machine Learning (ML) are gaining popularity due to their broad range of applications, ease of implementation, and cutting-edge performance. However, training DNNs in supervised learning can be computationally expensive and need a large quantity of training data, especially as DNN sizes grow.
DNNs have traditionally been used in MEC to collect data from mobile phones/IoT devices/SNs, train the model in the cloud, and then deploy the model to the edge. Users are becoming increasingly hesitant to upload potentially sensitive data due to privacy concerns, posing the question of how these models will be trained.

Federated learning (also known as collaborative learning) is a machine learning approach that involves training an algorithm across numerous decentralised edge devices or servers that keep local data samples private without transferring them to servers. In Federated learning the clients do not reveal their private data and train on their local datasets and push their new model to the server.The server averages these models together before sending the new aggregated model to clients for next round.

Based on the optimization strategy used, the federated learning algorithm can be classified into FedAvg,FedAdam and FedAvg-Adam.The FedAvg algorithm uses SGD for local update and averaging for global model update. FedAdam uses SGD for local update and Adam for global update.FedAvg-Adam uses Adam for local updates while it uses Averaging for global model update.

Literature survey:

Multi-Task Federated Learning for Personalised Deep Neural Networks in Edge Computing[Jed Mills, Jia Hu, Geyong Min] :

Introducing private patch layers into the global model, They proposed a Customised Federated Learning algorithm that builds on iterative FL algorithms. Users can have personalised models with private layers, which improves the average accuracy of user models significantly (UA). They looked into the benefits of using BN layers as patches in FL. They proposed the FedAvg-Adam optimisation scheme, which uses Adam on clients, because this is a general algorithm that requires a specific FL optimisation strategy.

Experiments with MNIST show that FL with FedAvg reduces the number of rounds required to reach a target average UA by up to 5x when compared to FL. Further tests show that FL combined with FedAvg-Adam reduces this number even more, by up to three times.

These experiments also show that using private BN trainable parameters (γ, β) in model patches rather than statistics (μ, σ) improves convergence speed. When compared to other cutting-edge personalised FL algorithms, FL achieves the highest average UA with the fewest communication rounds.

Finally, They demonstrated that the communication overhead of FL with FedAvg-Adam is outweighed by its significant benefits over FL with FedAvg in terms of UA and convergence speed in experiments using a MEC-like testbed.

A survey on federated Learning Systems:Vision,Hype and Reality for Data Privacy and Protection[Qinbin Li, Zeyi Wen , Zhaomin Wu , Sixu Hu , Naibo Wang , Yuan Li , Xu Liu , Bingsheng He] :

Federated learning has been a hot research topic in allowing different organisations to collaborate on machine learning model training while maintaining privacy. As researchers work to support more machine learning models with various privacy-preserving approaches, systems and infrastructures are needed to make the development of various federated learning algorithms easier. Federated learning systems (FLSs) are important in the same way that deep learning systems like PyTorch and TensorFlow help to advance deep learning. However, FLSs face challenges in terms of effectiveness, efficiency, and privacy. They conducted a comprehensive review of federated learning systems in this survey. They introduced the definition of federated learning systems and analysed the system components to ensure a smooth flow and to guide future research.

Furthermore, they categorised federated learning systems based on six different factors, including data distribution, machine learning model, privacy mechanism, communication architecture, federation scale, and federation motivation. As shown in our case studies, categorization can aid in the design of federated learning systems. They presented design factors, case studies, and future research opportunities by systematically summarising existing federated learning systems.

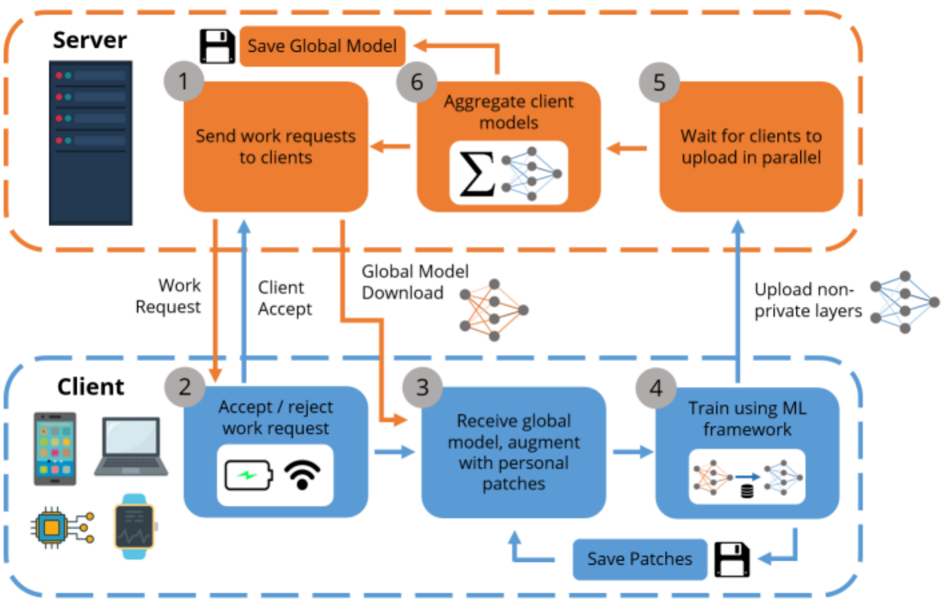
Federated Learning with Non-IID Data
Yue Zhao*, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, Vikas Chandra

Federated learning allows edge computing devices with limited resources, such as mobile phones and IoT devices, to learn a common model for prediction while keeping the training data local. This decentralised method of training models offers advantages in terms of privacy, security, regulatory compliance, and cost. The statistical problem of federated learning when local data is non-IID is the subject of this research. They first show that for neural networks trained for highly skewed non-IID data, where each client device trains just on a single class of data, the accuracy of federated learning decreases dramatically, by up to 55%.

They also show that the weight divergence, which can be measured by the earth mover's distance (EMD) between the distribution over classes on each device and the population distribution, can explain the loss in accuracy. As a solution, They propose establishing a small sample of data that is universally shared across all edge devices in order to improve training on non-IID data. Experiments suggest that using only 5% globally shared data, accuracy for the CIFAR-10 dataset can be boosted by 30%.

Proposed Work:

We are comparing three customised federated learning algorithms to achieve user model accuracy locally in all the edge nodes as well as global accuracy without compromising user privacy. User data would never be sent to the cloud for training the model. The level of privacy and the data sent to the server is completely under user control. We are comparing fed-avg, fed-adam, fed-avg-adam algorithms and comparing the performance of those algorithms in achieving the required accuracy with less number of communication rounds between server and edge nodes.



Architecture

Step-1:-
To begin, the server picks all or a subset of clients from its database and sends them a Work Request message inviting them to join the FL round.

Step-2:-
Clients will then accept or reject the Work Request based on their choices or conditions, such as battery capacity and internet access. The server receives an Accept message from all accepting clients.

Step-3:-
The server transmits the global model to all Request approved clients and adds private patches to their copy of the global model.

Step-4:-
Clients then use their own data to do local training and generate a new model. Clients save their new model's patch layers locally before uploading their non-private model parameters to the server.

Step-5:-
The server will patiently waits for clients to complete their training and submit their models.

Step-6:-
After that, the server will compile all of the incoming models into a single global model, which will be saved on the server before a new round begins.

Implementation:

The simulation of federated learning was implemented using Python and the deep learning was implemented using pytorch and the data obtained was visualised using matplotlib in a jupyter notebook to compare the number of rounds required to achieve high accuracy of local model. We used a mnist dataset for this project,it is a dataset of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples The models FedAvg,FedAvg-Adam and FedAdam were trained on the dataset and the accuracy of the model was plotted.

Code :

Required parameters and configurations for Federated Learning through command line arguments.

```python
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-dset', required=True, choices=['mnist'],
                        help='Federated dataset')
    parser.add_argument('-alg', required=True, help='Federated optimiser',
                        choices=[   'fedavg', 'fedavg-adam', 'fedadam',
                                    'pfedme', 'perfedavg'])
    parser.add_argument('-C', required=True, type=float,
                        help='Fraction of clients selected per round')
    parser.add_argument('-B', required=True, type=int, help='Client batch size')
    parser.add_argument('-T', required=True, type=int, help='Total rounds')
    parser.add_argument('-E', required=True, type=int, help='Client num epochs')
    parser.add_argument('-device', required=True, choices=['cpu', 'gpu'],
                        help='Training occurs on this device')
    parser.add_argument('-W', required=True, type=int,
                        help='Total workers to split data across')
    parser.add_argument('-seed', required=True, type=int, help='Random seed')
    parser.add_argument('-lr', required=True, type=float,
                        help='Client learning rate')
    parser.add_argument('-noisy_frac', required=True, type=float,
                        help='Fraction of noisy clients')

    # specific arguments for different FL algorithms
    if any_in_list(['fedavg', 'fedavg-adam', 'fedadam'], argv):
        parser.add_argument('-bn_private', choices=['usyb', 'us', 'yb', 'none'],
                            required=True,
                            help='Patch parameters to keep private')

    if any_in_list(['fedadam'], argv):
        parser.add_argument('-server_lr', required=True, type=float,
                            help='Server learning rate')

    if any_in_list(['fedavg-adam', 'fedadam'], argv):
        parser.add_argument('-beta1', required=True, type=float,
                            help='Only required for FedAdam, 0 <= beta1 < 1')
        parser.add_argument('-beta2', required=True, type=float,
                            help='Only required for FedAdam, 0 <= beta2 < 1')
        parser.add_argument('-epsilon', required=True, type=float,
                            help='Only required for FedAdam, 0 < epsilon << 1')

    args = parser.parse_args()

    return args
```

Based on the input from the command line argument, the appropriate algorithm and function is selected.
In case of fedavg algorithm the client uses SGD as optimization function.This optimization function is assigned to the client model.The server takes weighted average of the models from the client. In fedadam the server uses Adam as an optimisation function whereas the client uses SGD. In Fed Avg-adam, the client uses Adam optimization function and the weighted average of all models is taken in the server. The model,optimization function,batch size, noise, batch normalisation and other parameters are sent to the function where the actual simulation of the project takes place.

```python
    # run experiment
    print('Starting experiment...')
    if args.alg == 'fedavg':
        client_optim = ClientSGD(model.parameters(), lr=args.lr)
        model.set_optim(client_optim)
        data = run_fedavg(  feeders, test_data, model, client_optim, args.T, M,
                            K, args.B, bn_setting=bn_setting,
                            noisy_idxs=noisy_idxs)
    elif args.alg == 'fedadam':
        client_optim = ClientSGD(model.parameters(), lr=args.lr)
        model.set_optim(client_optim)
        server_optim = ServerAdam(  model.get_params(), args.server_lr,
                                    args.beta1, args.beta2, args.epsilon)
        data         = run_fedavg_adam(   feeders, test_data, model,
                                            server_optim, args.T, M,
                                            K, args.B,
                                            bn_setting=bn_setting,
                                            noisy_idxs=noisy_idxs)

    elif args.alg == 'fedavg-adam':
        client_optim = ClientAdam(  model.parameters(), lr=args.lr,
                                    betas=(args.beta1, args.beta2),
                                    eps=args.epsilon)
        model.set_optim(client_optim)
        data = run_fedavg(  feeders, test_data, model, client_optim, args.T, M,
                            K, args.B, bn_setting=bn_setting,
                            noisy_idxs=noisy_idxs)
```

In the run fed avg function, the train/test accuracy and errors array are initialised with empty values.The user private model and optimiser Batch normalisation values are extracted into a variable. A variable is created to store the global model and optimiser,which will be updated at the end of each round. The round_agg is used to accumulate client models in each round. The for loop simulates the global rounds.There are T global rounds.In each round, random user ids are selected and assigned weights.This simulates the selection of users by work request in real world. In each of the selected users, the global model is downloaded and the batch normalisation layer is applied and the model is trained using SGD locally. The models are aggregated into round_agg and the batch normalisation used is stored in an array. The global model is then updated to round_agg and train and test accuracy and errors are calculated.The train error, train accuracy,test error,test accuracy are returned which is then stored as pkl file.

```python
fl_algs.py > run_fedavg_adam
import numpy as np
import pickle
import torch
from progressbar import progressbar
from models import NumpyModel


def init_stats_arrays(T):
    return tuple(np.zeros(T, dtype=np.float32) for i in range(4))

def run_fedavg( data_feeders, test_data, model, client_opt,
                T, M, K, B, test_freq=1, bn_setting=0, noisy_idxs=[]):

    W = len(data_feeders)

    train_errs, train_accs, test_errs, test_accs = init_stats_arrays(T)

    # contains private model and optimiser BN vals (if bn_setting != 3)
    user_bn_model_vals = [model.get_bn_vals(setting=bn_setting) for w in range(W)]
    user_bn_optim_vals = [client_opt.get_bn_params(model) for w in range(W)]

    # global model/optimiser updated at the end of each round
    round_model = model.get_params()
    round_optim = client_opt.get_params()

    # stores accumulated client models/optimisers each round
    round_agg    = model.get_params()
    round_opt_agg = client_opt.get_params()

```

```python
    # stores accumulated client models/optimisers each round
    round_agg    = model.get_params()
    round_opt_agg = client_opt.get_params()

    for t in progressbar(range(T)):
        round_agg = round_agg.zeros_like()
        round_opt_agg = round_opt_agg.zeros_like()

        # select round clients and compute their weights for later sum
        user_idxs = np.random.choice(W, M, replace=False)
        weights   = np.array([data_feeders[u].n_samples for u in user_idxs])
        weights   = weights.astype(np.float32)
        weights   /= np.sum(weights)

        round_n_test_users = 0

        for (w, user_idx) in zip(weights, user_idxs):
            # download global model/optim, update with private BN params
            model.set_params(round_model)
            client_opt.set_params(round_optim)
            model.set_bn_vals(user_bn_model_vals[user_idx], setting=bn_setting)
            client_opt.set_bn_params(user_bn_optim_vals[user_idx],
                                    model, setting=bn_setting)

            # test local model if not a noisy client
            if (t % test_freq == 0) and (user_idx not in noisy_idxs):
                err, acc = model.test(  test_data[0][user_idx],
                                        test_data[1][user_idx], 128)
                test_errs[t] += err
                test_accs[t] += acc
                round_n_test_users += 1

```

```python

            # test local model if not a noisy client
            if (t % test_freq == 0) and (user_idx not in noisy_idxs):
                err, acc = model.test(  test_data[0][user_idx],
                                        test_data[1][user_idx], 128)
                test_errs[t] += err
                test_accs[t] += acc
                round_n_test_users += 1

            # perform local SGD
            for k in range(K):
                x, y = data_feeders[user_idx].next_batch(B)
                err, acc = model.train_step(x, y)
                train_errs[t] += err
                train_accs[t] += acc

            # upload local model/optim to server, store private BN params
            round_agg = round_agg + (model.get_params() * w)
            round_opt_agg = round_opt_agg + (client_opt.get_params() * w)
            user_bn_model_vals[user_idx] = model.get_bn_vals(setting=bn_setting)
            user_bn_optim_vals[user_idx] = client_opt.get_bn_params(model,
                                                    setting=bn_setting)

        # new global model is weighted sum of client models
        round_model = round_agg.copy()
        round_optim = round_opt_agg.copy()

        if t % test_freq == 0:
            test_errs[t] /= round_n_test_users
            test_accs[t] /= round_n_test_users

    train_errs /= M * K
    train_accs /= M * K

    return train_errs, train_accs, test_errs, test_accs
```

In run fedavg adam algorithm, local model in client nodes uses either SDG or Optimised Adam algorithm for training and Server uses SDG for finding the optimised model and distributes them to client nodes. And this happens many times until required accuracy is achieved.

The train/test accuracy and errors array are initialised with empty values.The user private model and optimiser Batch normalisation values are extracted into a variable. A variable is created to store the global model and optimiser, which will be updated at the end of each round.

There are T global rounds.In each round, random user ids are selected and assigned weights.This simulates the selection of users by work request in the real world.

The global model is then updated to round_agg and train and test accuracy and errors are calculated.The train error, train accuracy,test error,test accuracy are returned which is then stored as a pkl file.

fl_algs.py > run_fedavg

```python

def run_fedavg_adam( data_feeders, test_data, model, server_opt, T, M,
                        K, B, test_freq=1, bn_setting=0, noisy_idxs=[]):
    W = len(data_feeders)

    train_errs, train_accs, test_errs, test_accs = init_stats_arrays(T)

    round_model = NumpyModel(model.get_params())
    round_grads = NumpyModel(model.get_params())

    # contains private BN vals (if bn_setting != 3)
    user_bn_model_vals = [model.get_bn_vals(bn_setting) for w in range(W)]

    for t in progressbar(range(T)):
        round_grads = round_grads.zeros_like()  # round psuedogradient

        # select round clients and compute their weights for later sum
        user_idxs = np.random.choice(W, M, replace=False)
        weights   = np.array([data_feeders[u].n_samples for u in user_idxs])
        weights   = weights.astype(np.float32)
        weights   /= np.sum(weights)

        round_n_test_users = 0

        for (w, user_idx) in zip(weights, user_idxs):
            # download global model, update local model with private BN params
            model.set_params(round_model)
            model.set_bn_vals(user_bn_model_vals[user_idx], bn_setting)

```

```python

        for (w, user_idx) in zip(weights, user_idxs):
            # download global model, update local model with private BN params
            model.set_params(round_model)
            model.set_bn_vals(user_bn_model_vals[user_idx], bn_setting)

            # test local model if not a noisy client
            if (t % test_freq == 0) and (user_idx not in noisy_idxs):
                err, acc = model.test(  test_data[0][user_idx],
                                        test_data[1][user_idx], 128)
                test_errs[t] += err
                test_accs[t] += acc
                round_n_test_users += 1

            # perform local SGD
            for k in range(K):
                x, y = data_feeders[user_idx].next_batch(B)
                loss, acc = model.train_step(x, y)
                train_errs[t] += loss
                train_accs[t] += acc

            # upload local model to server, store private BN params
            round_grads = round_grads + ((round_model - model.get_params()) * w)
            user_bn_model_vals[user_idx] = model.get_bn_vals(bn_setting)

        # update global model using psuedogradient
        round_model = server_opt.apply_gradients(round_model, round_grads)

        if t % test_freq == 0:
            test_errs[t] /= round_n_test_users
            test_accs[t] /= round_n_test_users

    train_errs /= M * K
    train_accs /= M * K

    return train_errs, train_accs, test_errs, test_accs

```

The mode of Batch Normalisation is chosen from command line argument, accordingly either normalisation would be done on mean or on mean variance considering their weighted bias.

```
class FLModel(torch.nn.Module):
    def __init__(self, device):
        super(FLModel, self).__init__()
        self.optim      = None
        self.device     = device
        self.loss_fn    = None
        self.bn_layers  = []        # any model BN layers must be added to this

    def set_optim(self, optim, init_optim=True):
        self.optim = optim
        if init_optim:
            self.empty_step()

    def empty_step(self):
        raise NotImplementedError()

    def get_params(self):

        ps = [np.copy(p.data.cpu().numpy()) for p in list(self.parameters())]
        for bn in self.bn_layers:
            ps.append(np.copy(bn.running_mean.cpu().numpy()))
            ps.append(np.copy(bn.running_var.cpu().numpy()))

        return NumpyModel(ps)

    def get_bn_vals(self, setting=0):

        if setting not in [0, 1, 2, 3]:
            raise ValueError('Setting must be in: {0, 1, 2, 3}')

        vals = []

        if setting == 3:
            return vals

        with torch.no_grad():
            # add gamma, beta
            if setting in [0, 1]:
                for bn in self.bn_layers:
                    vals.append(np.copy(bn.weight.cpu().numpy()))
                    vals.append(np.copy(bn.bias.cpu().numpy()))

            # add mu, sigma
            if setting in [0, 2]:
                for bn in self.bn_layers:
                    vals.append(np.copy(bn.running_mean.cpu().numpy()))
                    vals.append(np.copy(bn.running_var.cpu().numpy()))
        return vals

```

Code Link :

https://github.com/LOGANATHANAN/fog-edge-computing

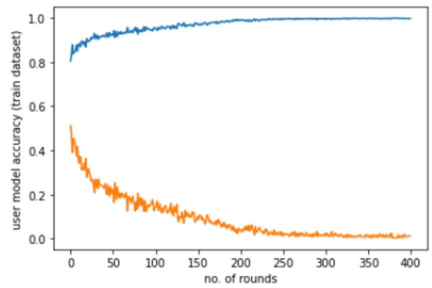
Results & Discussion:

Output Screenshots and Analysis

python3 main.py -dset mnist -C 0.5 -W 200 -T 400 -E 1 -alg fedavg-adam -seed 2 -lr 0.5 -noisy_frac 0.2 -device cpu -B 20 -bn_private usyb -beta1 0.5 -beta2 0.5 -epsilon 0.5

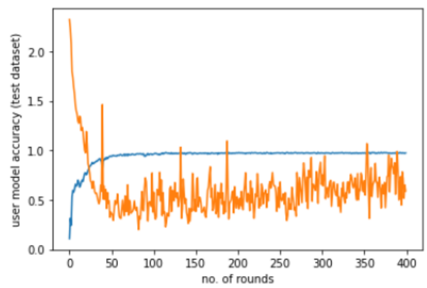
Fed Avg-adam Algorithm:

In [12]:
```
import pandas as pd
import pickle
import numpy as np
import matplotlib.pyplot as plt
my_data=pd.read_pickle('dset-mnist_alg-fedavg-adam_C-0.5_B-20_T-400_E-1_device-cpu_W-200_seed-2_lr-0.5_noisy_frac-0.2_bn_private-
#print(my_data)

train_err=my_data[0]
train_acc=my_data[1]
test_err=my_data[2]
test_acc=my_data[3]

r=range(0,400)
plt.plot(r,train_acc,r,train_err)
plt.xlabel('no. of rounds')
plt.ylabel('user model accuracy (train dataset) and error')
plt.show()
```



Above graph is on a training dataset which shows that when the communication rounds increase between server and client, accuracy increases and error decreases. Below graph is accuracy and error plot of test dataset.

In [2]:
```
plt.plot(r,test_acc,r,test_err)
plt.xlabel('no. of rounds')
plt.ylabel('user model accuracy (test dataset) and error')
plt.show()
```

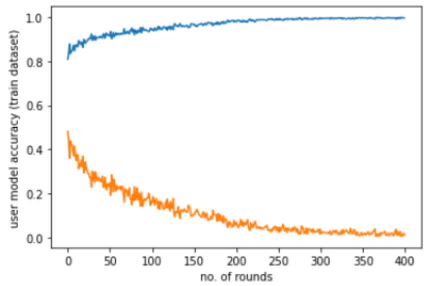


Fed avg-adam model achieves 95% user model accuracy in just 60 communication rounds on training dataset and 67 rounds on test dataset.

python3 main.py -dset mnist -C 0.5 -W 200 -T 400 -E 1 -alg fedavg -seed 2 -lr 0.5 -noisy_frac 0.2 -device cpu -B 20 -bn_private usyb

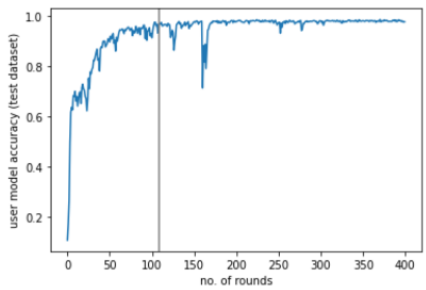
Fed avg algorithm :

In [3]:
```
my_data=pd.read_pickle('dset-mnist_alg-fedavg_C-0.5_B-20_T-400_E-1_device-cpu_W-200_seed-2_lr-0.5_noisy_frac-0.2_bn_private-usyb.
#print(my_data)

train_err=my_data[0]
train_acc=my_data[1]
test_err=my_data[2]
test_acc=my_data[3]

r=range(0,400)
plt.plot(r,train_acc,r,train_err)
plt.xlabel('no. of rounds')
plt.ylabel('user model accuracy (train dataset) and error')
plt.show()
```



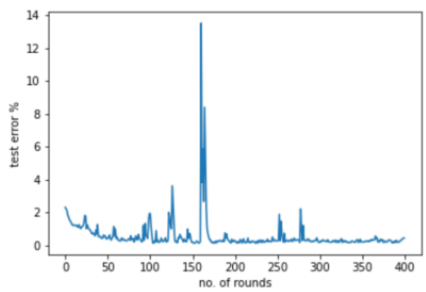
Above graph is on a training dataset which shows that when the communication rounds increase between server and client, accuracy increases and error decreases. Below graph is accuracy and error plot of test dataset.

In [32]:
```
plt.plot(r,test_acc)
plt.xlabel('no. of rounds')
plt.ylabel('user model accuracy (test dataset)')
plt.axvline(x=np.interp(0.95,train_acc,r),ymin=0.00,ymax=1.00,color='grey')
plt.show()
```



Fed avg model achieves 95% user model accuracy in just 81 communication rounds on training dataset and 107 rounds on test dataset.

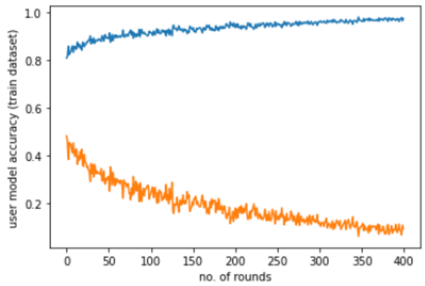
In [33]:
```
plt.plot(r,test_err)
plt.xlabel('no. of rounds')
plt.ylabel('test error %')
plt.show()
```



Fundamentals of Fog and Edge Computing

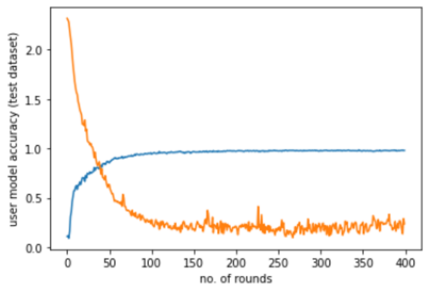
Python3 main.py -dset mnist -C 0.5 -W 200 -T 400 -E 1 -alg fedadam -seed 2 -lr 0.5 -server_lr 0.2 -noisy_frac 0.2 -device cpu -B 20 -bn_private usyb -beta1 0.5 -beta2 0.5 -epsilon 0.5

In [7]:
```
my_data=pd.read_pickle('dset-mnist_alg-fedadam_C-0.5_B-20_T-400_E-1_device-cpu_W-200_seed-2_lr-0.5_noisy_frac-0.2_bn_private-usyb
#print(my_data)

train_err=my_data[0]
train_acc=my_data[1]
test_err=my_data[2]
test_acc=my_data[3]

r=range(0,400)
plt.plot(r,train_acc,r,train_err)
plt.xlabel('no. of rounds')
plt.ylabel('user model accuracy (train dataset) and error')
plt.show()
```



Above graph is on a training dataset which shows that when the communication rounds increase between server and client, accuracy increases and error decreases. Below graph is accuracy and error plot of test dataset.

In [8]:
```
plt.plot(r,test_acc,r,test_err)
plt.xlabel('no. of rounds')
plt.ylabel('user model accuracy (test dataset) and error')
plt.show()
```



Fed avg model achieves 95% user model accuracy in just 103 communication rounds on training dataset and 194 rounds on test dataset.

Below statistics is for following parameters :
Dataset - mnist, Total no. of edge nodes -200, Edge node epochs-1,Total Communication rounds-400,noise in data=20%, fraction of active edge nodes=50%, batch normalisation - usyb, hyper tuning parameters for FL algorithm-0.5

Algorithm	Rounds taken to achieve 95% user model accuracy (train dataset)	Rounds taken to achieve 95% user model accuracy (test dataset)
Fedavg	81	107
Fedadam	103	194
Fedavg-adam	60	67

References:

[1] Y. Mao, C. You, J. Zhang, K. Huang, and K. Letaief, “A survey on mobile edge computing: The communication perspective,” IEEE Communications Surveys Tutorials, vol. 19, no. 4, pp. 2322–2358, 2017.

[2] B. Yang, X. Cao, J. Bassey, X. Li, and L. Qian, “Computation offloading in multi-access edge computing: A multi-task learning approach,” in Proc. IEEE International Conference on Communications (ICC), pp. 1– 6, 2019.

[3] Y. Li, X. Wang, X. Gan, H. Jin, L. Fu, and X. Wang, “Learning-aided computation offloading for trusted collaborative mobile edge computing,” IEEE Transactions on Mobile Computing, vol. 19, no. 12, pp. 2833–2849, 2020.

[4] T. Li, A. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions,” IEEE Signal Processing Magazine, vol. 37, no. 3, pp. 50–60, 2020.

[5] B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in Proc. 20th International Conference on Artifical Intelligence and Statistics (AISTATS), pp. 1273–1282, 2017.

[6] A. G. Roy, S. Siddiqui, S. Polsterl, N. Navab, and C. Wachinger, ¨ “Braintorrent: A peer-to-peer environment for decentralized federated learning,” arXiv preprint arXiv:1905.06731, 2019.

[7] A. Hard, K. Rao, R. Mathews, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, “Federated learning for mobile keyboard prediction,” arXiv preprint arXiv:1811.03604, 2018.

[8] M. Ammad-ud-din, E. Ivannikova, S. A. Khan, W. Oyomno, Q. Fu, K. E. Tan, and A. Flanagan, “Federated collaborative filtering for privacy-preserving personalized recommendation system,” arXiv preprint arXiv:1901.09888, 2019.

[9] J. Mills, J. Hu, and G. Min, “Multi-Task Federated Learning for Personalised Deep Neural Networks in Edge Computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 630–641, Mar. 2022, doi: 10.1109/tpds.2021.3098467.