Advanced settings

Adjusting autoencoder architecture/training

to use a different autoencoder, a different architecture can be chosen. Change the encoder model in client.py to your desired encoder model:

```
class Encode(nn.Module):
    def __init__(self):
        super(Encode, self).__init__()
        self.conva = nn.Conv2d(32, 16, 3, padding=1)
        self.convb = nn.Conv2d(16, 8, 3, padding=1)
        self.convc = nn.Conv2d(8, 4, 3, padding=1)

    def forward(self, x):
        x = self.conva(x)
        x = self.convb(x)
        x = self.convc(x)
        return x
```

and the decoder model in server.py to your desired decoder model:

```
class Decode(nn.Module):
    def __init__(self):
        super(Decode, self).__init__()
        self.t_convx = nn.ConvTranspose2d(4, 8, 1, stride=1)
        self.t_conva = nn.ConvTranspose2d(8, 16, 1, stride=1)
        self.t_convb = nn.ConvTranspose2d(16, 32, 1, stride=1)

    def forward(self, x):
        x = self.t_convx(x)
        x = self.t_conva(x)
        x= self.t_convb(x)
        return x
```

To pretrain the models, use the pretrainer.py. you can use any dataset/hyperparameter to pretrain the autoencoder in the pretrainer.py.

Attention: client model, encoder model and decoder model in the pretrainer.py have to be the same as in the client.py and server.py. Also, the data-shape at the cut layer hast to correspond to the input/output shape of the autoencoder.

The pretrainer will create files named: convencoder.pth and convdecoder.pth. these files contain the weights for the encoder model/ decoder model for the actual training process. By copying these files into the client.py / srever.py directory, they will be load autonomously by these scripts in their main function before starting the actual training process:

client.py:

```
global encode
encode = Encode()
encode.load_state_dict(torch.load("./convencoder.pth"))
encode.eval()
encode.to(device)
```

server.py

```
global decode
decode = Decode()
decode.load_state_dict(torch.load("./convdecoder.pth"))
decode.eval()
decode.to(device)
```

Adjusting model architecture:

in order to use a different model architecture, simply adjust the model architecture at the client int the client.py:

```
class Client(nn.Module):
    def __init__(self):
        super(Client, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        self.norm1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.norm2 = nn.BatchNorm2d(32)
        self.drop1 = nn.Dropout2d(0.2)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.norm1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.norm2(x)
        x = self.drop1(x)
        return x
```

or at the server in the server.py:

```
class Server(nn.Module):
    def __init__(self):
        super(Server, self).__init__()
        self.conv3 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
        self.relu3 = nn.ReLU()
        self.norm3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        self.relu4 = nn.ReLU()
        self.norm4 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        self.drop2 = nn.Dropout2d(0.3)
        self.conv5 = nn.Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
        self.relu5 = nn.ReLU()
        self.norm5 = nn.BatchNorm2d(128)
        self.conv6 = nn.Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))
        self.relu6 = nn.ReLU()
        self.norm6 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        self.drop3 = nn.Dropout2d(0.4)
        self.linear1 = nn.Linear(in_features=128, out_features=43, bias=True)

    def forward(self, x):
        x = self.conv3(x)
        x = self.relu3(x)
        x = self.norm3(x)
        x = self.conv4(x)
        x = self.relu4(x)
        x = self.norm4(x)
        x = self.pool2(x)
        x = self.drop2(x)
        x = self.conv5(x)
        x = self.relu5(x)
        x = self.norm5(x)
        x = self.conv6(x)
        x = self.relu6(x)
        x = self.norm6(x)
        x = self.pool3(x)
        x = self.drop3(x)
        x = x.view(x.size(0), -1)
        x = nn.functional.log_softmax(self.linear1(x), dim=1)
        return x
```

Attention: The data-shape at the cut layer hast to correspond to the input/output shape of the autoencoder.

Changing Optimizer/ Loss/ Training device

Optimizer, loss and the device (as well as the models for encoder, decoder, client and server) are globally defined in the main function of the client.py/ server.py and can be changed.

client.py:

```
global device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

global client
client = Client()
client.to(device)

global encode
encode = Encode()
encode.load_state_dict(torch.load("./convencoderlate.pth"))
encode.eval()
encode.to(device)

global optimizer
optimizer = SGD(client.parameters(), lr=lr, momentum=0.9)

global error
error = nn.CrossEntropyLoss()
```

server.py:

```
global device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

global server
server = Server()
server.to(device)

global decode
decode = Decode()
decode.load_state_dict(torch.load("./convdecoder.pth"))
decode.eval()
decode.to(device)

global optimizer
optimizer = SGD(server.parameters(), lr=lr, momentum=0.9)

global error
error = nn.CrossEntropyLoss()
```

Customized Pretraining

The pretraining can also be customized. The pretrainer of the autoencoder can be found in the /ae_pretrainer/ae_pretrainer.py directory.

The dataset used for pretraining can be set. By default, the CIFAR10 dataset is used:

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)

# Load Datasets
train_loader = torch.utils.data.DataLoader(trainset, batch_size=batchsize,
shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(testset, batch_size=test_batches,
shuffle=False, num_workers=2)
```

To set the training parameter, learningrate, training epochs, batchsize and number of test batches can be changed.

```
batchsize = 64
test_batches = 1500
epoch = 250
lr = 0.0001
```

The lossfunction and device can be configured in the main function of the ae_pretrainer.py file

```
global device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
global error
error = nn.MSELoss()
```

Since a customized adaptive learning rate mechanism is used, the optimizer for encoder and decoder have to be set in the start_training() function.

Parameters for the customized adaptive learningrate training can be set as the following parameters:

```
prev_loss = 999
diverge_tresh = 1.1
lr_adapt = 0.5
```

The diverge_thresh variable describes, that if the thresh of an training epoch is above this threshold (e.g. 1.1 = the loss has increased by 10% in comparison to the previous training epoch), the weightchanges fron thes epoch are dropped (due to divergence, state from previous epoch is load). Additionally, the learningrate is modified by the factor lr_adapt . prev_loss is a helpvariable of the loss of the "imaginary previous epoch" of the first training epoch.

Also, encoder and decoder model can be changed in the ae_pretrain.py file.

encoder

```python
class Encode(nn.Module):
    def __init__(self):
        super(Encode, self).__init__()
        self.conva = nn.Conv2d(64, 16, 3, padding=1)
        self.convb = nn.Conv2d(16, 8, 3, padding=1)
        self.convc = nn.Conv2d(8, 4, 3, padding=1)##

    def forward(self, x):
        x = self.conva(x)
        x = self.convb(x)
        x = self.convc(x)
        return x
```

decoder

```python
class Decode(nn.Module):
    def __init__(self):
        super(Decode, self).__init__()
        self.t_convx = nn.ConvTranspose2d(4, 8, 1, stride=1)
        self.t_conva = nn.ConvTranspose2d(8, 16, 1, stride=1)
        self.t_convb = nn.ConvTranspose2d(16, 64, 1, stride=1)

    def forward(self, x):
        x = self.t_convx(x)
        x = self.t_conva(x)
        x= self.t_convb(x)
        return x
```

Attention: client model, encoder model and decoder model in the pretrainer.py have to be the same as in the client.py and server.py. Also, the data-shape at the cut layer hast to correspond to the input/output shape of the autoencoder.

In the ae_pretrainer.py file, the Clientmodel always has to match the client side model in the client/client.py

After the training has finished, two files (convencoder.pth, convdecoder.pth) are created autonomously in the /ae_pretrainer directory. The convencoder.pth file has to be copied into the /client directory, and the convdecoder.pth file into the /server directory.

Measuring Message Size and FLOPs

In order to figure out the size of a message in bytes, the following function can be added to the client.py or server.py:

```
import struct
import sys

def get_size_send_msg(msg):
"""
can be called to figure out the meassage size of a message in byte
:param msg: message
:return: string that includes the meassagesize in bytes
"""

    msg = [0, msg]  # add getid
    msg = pickle.dumps(msg)
        # add 4-byte length in network byte order
    msg = struct.pack('>I', len(msg)) + msg
    return ("sendsize: ", sys.getsizeof(msg), " bytes")
```

In order to estimate the FLOPs (Floating Point Operations) of the forward pass of a MODEL, the library thop can be used: https://github.com/Lyken17/pytorch-OpCounter.

```
import thop
flops_client, params = thop.profile(MODEL, inputs=(torch.rand(batchsize, 3, 32,
32).to(device),))
```

Recreate Environment

In the repository, an environment.yaml file is provided. The Conda environment can be reacreated easily, by running the following command:

```
conda env create -f environment.yaml
```