Advanced settings Server

Adjusting autoencoder architecture/training

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

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 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.

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()
```

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
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.

```
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
```