```
import torch
import torch . nn . functional as F
from torch.utils. data import DataLoader
import torch.optim as optim
from torch vision import datasets , transforms

import copy

def FedAvg (w ) :
     w_avg = copy . deepcopy (w[ 0 ] )
     for k in w_avg . keys ( ) :
       for i in range ( 1 , len(w ) ) :
             w_avg [ k ] += w[ i ] [ k ]
        w_avg [ k ] = t o rch . div ( w_avg [ k ] , len(w) )
return w_avg

def fedAvg ( self , client_ weights ) :
   average_weight = list ( )
     for i in range(len( client_ weights [ 0 ] ) ) :
     layer_weights = list ( )
       if len( cli e n t_ w ei g h t s ) > 1 :
        for w in client_ weights :
          try :
            layer_weights . append (w[ i ] . astype ( np
. float 64 ) )
          except:
             try :
w_ = np . array (w[ i ] , dtype=object)
layer_weights . append (w_. astype ( np . float64 ) )
             except:
 layer_weights . append (w[ i ] )
         try :
            # print ( layer_weights )
             average_weight . append ( keras . layers . Average ( )
                ( layer_weights ) )
             # print ( " calculated average weight of layer " , i )
         except:
              average_weigh t . append ( client_
weights [ 0 ] [ i ] )
         else :
            average_weight . append ( client_ weights [ 0 ] [ i ] )

    for i in range(len( self . model . layers ) ) :
        self . model . layers [ i ] . set_weights ( average_weight [
 i ] )

return average_weight

```

```
%reload_ext autoreload
%autoreload 2
%matplotlib inline

from fastai import *
from fastai.vision import *
from fastai.metrics import accuracy

path = untar_data(URLs.MNIST)
path

path.ls()

(path/'training').ls()

filenames = get_image_files(path/'training/5')
len(filenames), filenames[:10]

batchsize = 64
np.random.seed(0)
transform = get_transforms(do_flip=False)

databunch = ImageDataBunch.from_folder(path, train='training', valid_pct=0.2, size=28,
                                       ds_tfms=transform, bs=batchsize, num_workers=8).normalize()

databunch.show_batch(rows=3, figsize=(10, 10))

databunch.classes

learner = cnn_learner(databunch, models.resnet18, metrics=accuracy)

learner.fit_one_cycle(4)

learner.unfreeze()
learner.lr_find()

learner.fit_one_cycle(3, max_lr=slice(1e-6, 3e-3))

path_test = Path()/'data'

path_test.ls()

test_image = open_image(path_test/'a007.png')
test_image

learner.predict(test_image)
```

```
92
93 databunch.classes
94
95 test_image = open_image(path_test/'a003.png')
96 test_image
97
98 learner.predict(test_image)
99
100   test_image = open_image(path_test/'a005.png')
101   test_image
102
103   learner.predict(test_image)
104
105   class CNNMnist ( nn . Module ) :
106       def __init__ ( self , args ) :
107         super(CNNMnist , self ) . __init__ ( )
108         self . conv = nn . Conv2d ( args
   . num_channels , 28 , padding =1 ,
109           kernel_ size =3)
110          self.pool = nn . MaxPool2d ( 2 )
111          self.fc1 = nn . Line ar (28*14*14 , 128)
112          self.drop = nn . Dropout ( 0 . 2 )
113          self.fc2 = nn . Linear ( 128 , a r g s . num_classes )
114          self.act = nn . ReLU ( )
115     def forward ( s e l f , x ) :
116       x = self . act ( self . conv ( x ) ) # [ batch_size ,28 ,28
   ,28 ]
117       x = self . pool ( x ) # [ batch_size , 2 8 , 1 4 , 1 4 ]
118       x = x . view ( x . size ( 0 ) , -1) # [ batch_size
   ,28*14*14=5488]
119       x = self . act ( self . fc1 ( x ) ) # [ batch_size , 1 2 8 ]
120       x = self . drop ( x )
121       x = self . fc2 ( x ) # [ batch_size , 1 0 ]
122   return x
123
124   print(Model received )
125   print(Training model , please wait . . .)
126   print(Sent updated model to the server)
127
128   soc.close ()
129   print(Socket is closed.)
```