```python
#==============================================================================
#Copyright (c) 2018 by Georgia Tech Research Corporation.
#All rights reserved.
#
#The files contain code and data associated with the paper titled
#"A Deep Learning Approach to Estimate Stress Distribution: A Fast and
#Accurate Surrogate of Finite Element Analysis".
#
#The paper is authored by Liang Liang, Minliang Liu, Caitlin Martin,
#and Wei Sun, and published at Journal of The Royal Society Interface, 2018.
#
#The file list: ShapeData.mat, StressData.mat, DLStress.py, im2patch.m,
#UnsupervisedLearning.m, ReadMeshFromVTKFile.m, ReadPolygonMeshFromVTKFile.m,
#WritePolygonMeshAsVTKFile.m, Visualization.m, TemplateMesh3D.vtk, TemplateMesh2D.vtk.
#Note: *.m and *.py files were converted to pdf files for documentation purpose.
#
#THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES,
#INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
#FOR A PARTICULAR PURPOSE.
#==============================================================================

#%%
# go to cmd and follow the instructions to setup matlab python engine
#https://www.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html
#%%
from keras.models import Sequential
from keras.layers import Dense, Conv2DTranspose
import numpy
from numpy.random import RandomState
import scipy.io as sio
import time
import matlab.engine
#%% Unsupervised Learning is done in Matlab
def UnsupervisedLearning(DataFile, ShapeDataFile, StressDataFile, IdxList_train, IdxList_test):
    idx_train_mat=matlab.double(list(IdxList_train+1)) #+1 to matlab index
    idx_test_mat=matlab.double(list(IdxList_test+1))   #+1 to matlab index
    DataFlag = eng.UnsupervisedLearning(DataFile, ShapeDataFile, StressDataFile, idx_train_mat,
    idx_test_mat)
    MatData=sio.loadmat(DataFile)
    X=MatData['ShapeCode_train']
    X=numpy.asmatrix(X)
    X=X.transpose()
    S=MatData['Stress_train']
    S=numpy.asmatrix(S)
    Y_= MatData['Y2n_train']
    Y = numpy.array([])
    Y.resize((len(IdxList_train),64))
    for k in range(0, len(IdxList_train)):
        Y[k,:,]=Y_[:,:,:,k]
    #end
    Y=numpy.asmatrix(Y)
    #
    X_t=MatData['ShapeCode_test']
```

```
    X_t=numpy.asmatrix(X_t)
    X_t=X_t.transpose()
    S_t=MatData['Stress_test']
    S_t=numpy.asmatrix(S_t)
    Y_t_= MatData['Y2n_test']
    Y_t = numpy.array([])
    Y_t.resize((len(IdxList_test),64))
    for k in range(0, len(IdxList_test)):
        Y_t[k,:,]=Y_t_[:,:,:,k]
    #end
    Y_t=numpy.asmatrix(Y_t)
    #
    L=MatData['L2']
    W1=MatData['W1']
    W2=MatData['W2']       Stress encoding and decoding weights
    #
    Proj=MatData['Proj']     shape encoding weights
    MeanShape=MatData['MeanShape']
    #
    return X, X_t, Y, Y_t, S, S_t, L, W1, W2, Proj, MeanShape
#end
#------------------------------------
#%% S11/S22/S12: 5000xN, S_all=[S11; S22; S12]
def ComputeVonMisesStress_all(S_all):
    N=S_all.shape[1]
    VM_all=numpy.array([])
    VM_all.resize(5000, N)
    S11_all=S_all[0:5000,:]
    S22_all=S_all[5000:10000,:]
    S12_all=S_all[10000:15000,:]
    for n in range(0,N):
        for k in range(0, 5000):
            S11=S11_all[k,n]
            S22=S22_all[k,n]
            S12=S12_all[k,n]
            VM=S11*S11+S22*S22-S11*S22+3*S12*S12
            VM=numpy.sqrt(VM)
            VM_all[k,n]=VM
        #end
    #end
    return VM_all;
#end
#------------ A is ground truth, A[:,n] is vector of stress values; B is the reonstructed 
version of A
def ComputeError(A, B):
    MAE=numpy.zeros(A.shape[1])    
    NMAE=numpy.zeros(A.shape[1])
    for n in range(0, A.shape[1]):
        a=A[:,n]
        b=B[:,n]
        c=numpy.absolute(a-b)        
        a_abs=numpy.absolute(a)        
        a_max=numpy.max(a_abs[301:4700])        
```

```
        MAE[n]=numpy.mean(c)        
        NMAE[n]=MAE[n]/a_max
    #end
    return MAE, NMAE
#end
#------------
def ComputeError_peak(A, B):
    AE=numpy.zeros(A.shape[1])
    APE=numpy.zeros(A.shape[1])
    for n in range(0, A.shape[1]):    
        a=A[:,n]
        b=B[:,n]
        a_abs=numpy.absolute(a)
        b_abs=numpy.absolute(b)
        a_max=numpy.max(a_abs[301:4700])        
        b_max=numpy.max(b_abs[301:4700])        
        AE[n]=numpy.absolute(a_max-b_max)
        APE[n]=AE[n]/a_max
    #end
    return AE, APE
#end
#%% Define the Deep Learning Model: shape_encoding, nonlinear mapping, stress_decoding
def CreateModel_ShapeEncoding(NodeCount, Proj):
    model = Sequential()
    model.add(Dense(3, input_dim=NodeCount*3, kernel_initializer='normal', activation='linear'))
    W0=model.layers[0].get_weights()
    W0[0]=Proj
    model.layers[0].set_weights(W0)
    model.compile(loss='mean_squared_error', optimizer='adamax')
    return model
#end
#-----------------------------------
def CreateModel_NonlinearMapping(Xshape, Yshape):
    model = Sequential()
    model.add(Dense(128, input_dim=Xshape[1], kernel_initializer='normal', activation='softplus'
    ))
    model.add(Dense(128, kernel_initializer='normal', activation='softplus'))        
    model.add(Dense(Yshape[1], kernel_initializer='normal', activation='linear'))    
    model.compile(loss='mean_squared_error', optimizer='adamax')
    return model
#end
#-----------------------------------
def CreateModel_StressDecoding(W1_in, W2_in):
    model = Sequential()
    model.add(Conv2DTranspose(filters=256, kernel_size=[5,5], strides=(1,1), input_shape=(1,1,64
    ),data_format='channels_last'))
    model.add(Conv2DTranspose(filters=3, kernel_size=[10,20], strides=(10,20)))
    model.compile(loss='mean_squared_error', optimizer='adamax')
    W0=model.layers[0].get_weights()
    W1=model.layers[1].get_weights()    
    W0[0]=W2_in
    W1[0]=W1_in    
    model.layers[0].set_weights(W0)
```

```
    model.layers[1].set_weights(W1)
    return model 
#end
#-----------------------------------
#%%
ShapeDataFile='ShapeData.mat'
StressDataFile='StressData.mat'
TempDataFile='TempData.mat'
ResultFile='DL_Stress_result.mat'
#
MatData_shape=sio.loadmat(ShapeDataFile)
ShapeData=MatData_shape['ShapeData']
#%%
#make sure you can see DLStress.py in the current directory of spyder
eng = matlab.engine.start_matlab()
rng=RandomState(0)
IndexList= numpy.arange(0, 729, 1)
S11MAE=[]; S11NMAE=[];
S22MAE=[]; S22NMAE=[];
S12MAE=[]; S12NMAE=[];
VMMAE=[];   VMNMAE=[];
S11AE=[]; S11APE=[];
S22AE=[]; S22APE=[];
S12AE=[]; S12APE=[];
VMAE=[];   VMAPE=[];
IndexList_test=[];
IndexList_train=[];
for k in range(0, 100):
    #specify traning set and testing set    
    rng.shuffle(IndexList)
    idx_train=IndexList[0:656]    
    idx_test=IndexList[656:729] 
    IndexList_train.append(idx_train)
    IndexList_test.append(idx_test)    
    ShapeData_train=ShapeData[:,idx_train]
    ShapeData_test=ShapeData[:,idx_test]
    
    #unsupervised learning in Matlab
    t1=time.clock()    
    [X, X_t, Y, Y_t, S, S_t, L, W1, W2, Proj, MeanShape]=UnsupervisedLearning(TempDataFile, 
    ShapeDataFile, StressDataFile, idx_train, idx_test)
    t2=time.clock()
    print(k, 'Unsupervised Learning', t2-t1)
    
    #subtract mean shape
    for n in range(0, 656):
        ShapeData_train[:,n]-=MeanShape[:,0]
    #end
    for n in range(0, 73):
        ShapeData_test[:,n]-=MeanShape[:,0]
    #end

    #alghouth encoding has been done in Matlab, redo it here
```

arrange() It creates an array by using the evenly spaced values over the given interval. The interval mentioned is half opened i.e. [Start, Stop]).

```
    ShapeEncoder = CreateModel_ShapeEncoding(5000, Proj)
    X=ShapeEncoder.predict(ShapeData_train.transpose(), batch_size=100, verbose=0)
    X_t=ShapeEncoder.predict(ShapeData_test.transpose(), batch_size=100, verbose=0)
    t3=time.clock()
    print(k, 'Shape Encoding', t3-t2)
    
    NMapper=CreateModel_NonlinearMapping(X.shape, Y.shape)
    NMapper.fit(X, Y, epochs=5000, batch_size=100, verbose=0)
    #test
    Yp=NMapper.predict(X_t, batch_size=idx_test.size, verbose=0)
    for n in range(0, 64):
        Yp[:,n]=Yp[:,n]*L[n]
    #end
    Ypp = numpy.array([])
    Ypp.resize((idx_test.size,1,1,64))
    for n in range(0, idx_test.size):
        Ypp[n,0,0,:]=Yp[n,:]
    #end
    t4=time.clock()
    print(k, 'Nonlinear Mapping', t4-t3)
    
    StressDecoder=CreateModel_StressDecoding(W1, W2);
    Spp=StressDecoder.predict(Ypp);
    Sp = numpy.array([])
    Sp.resize((15000, idx_test.size))
    for n in range(0, idx_test.size):
        tempS11=Spp[n,:,:,0];
        tempS11=tempS11.reshape((5000), order='F')
        Sp[0:5000,n]=tempS11
        tempS22=Spp[n,:,:,1];
        tempS22=tempS22.reshape((5000), order='F')
        Sp[5000:10000,n]=tempS22
        tempS12=Spp[n,:,:,2];
        tempS12=tempS12.reshape((5000), order='F')
        Sp[10000:15000,n]=tempS12
    #end
    Sp=numpy.asmatrix(Sp)
    t5=time.clock()
    print(k, 'Stress Decoding', t5-t4)
    
    #compare ground-truth S and predicted Sp
    [S11MAE_k, S11NMAE_k]=ComputeError(S_t[0:5000,:], Sp[0:5000,:])
    S11MAE.append(S11MAE_k)
    S11NMAE.append(S11NMAE_k)
    [S22MAE_k, S22NMAE_k]=ComputeError(S_t[5000:10000,:], Sp[5000:10000,:])
    S22MAE.append(S22MAE_k)
    S22NMAE.append(S22NMAE_k)
    [S12MAE_k, S12NMAE_k]=ComputeError(S_t[10000:15000,:], Sp[10000:15000,:])
    S12MAE.append(S12MAE_k)
    S12NMAE.append(S12NMAE_k)
    #peak stress error
    [S11AE_k, S11APE_k]=ComputeError_peak(S_t[0:5000,:], Sp[0:5000,:])
    S11AE.append(S11AE_k)
```

```
    S11APE.append(S11APE_k)
    [S22AE_k, S22APE_k]=ComputeError_peak(S_t[5000:10000,:], Sp[5000:10000,:])
    S22AE.append(S22AE_k)
    S22APE.append(S22APE_k)
    [S12AE_k, S12APE_k]=ComputeError_peak(S_t[10000:15000,:], Sp[10000:15000,:])
    S12AE.append(S12AE_k)
    S12APE.append(S12APE_k)
    #compare Von Mises Stress
    VM_t=ComputeVonMisesStress_all(S_t)
    VMp=ComputeVonMisesStress_all(Sp)
    [VMMAE_k, VMNMAE_k]=ComputeError(VM_t, VMp)
    VMMAE.append(VMMAE_k)
    VMNMAE.append(VMNMAE_k)
    #peak stress error
    [VMAE_k, VMAPE_k]=ComputeError_peak(VM_t, VMp)
    VMAE.append(VMAE_k)
    VMAPE.append(VMAPE_k)
    
    #report
    t6=time.clock()
    print(k, 'ComputeError', t6-t5)
    print('VM',  numpy.mean(VMMAE), numpy.std(VMMAE), numpy.mean(VMNMAE), numpy.std(VMNMAE))  
    print('S11', numpy.mean(S11MAE), numpy.std(S11MAE), numpy.mean(S11NMAE), numpy.std(S11NMAE))
    
    print('S22', numpy.mean(S22MAE), numpy.std(S22MAE), numpy.mean(S22NMAE), numpy.std(S22NMAE))
    print('S12', numpy.mean(S12MAE), numpy.std(S12MAE), numpy.mean(S12NMAE), numpy.std(S12NMAE))
    print('VMpeak', numpy.mean(VMAE), numpy.std(VMAE), numpy.mean(VMAPE), numpy.std(VMAPE))
    print('S11peak', numpy.mean(S11AE), numpy.std(S11AE), numpy.mean(S11APE), numpy.std(S11APE))
    print('S22peak', numpy.mean(S22AE), numpy.std(S22AE), numpy.mean(S22APE), numpy.std(S22APE))
    print('S12peak', numpy.mean(S12AE), numpy.std(S12AE), numpy.mean(S12APE), numpy.std(S12APE))
#end    
#%%
sio.savemat(ResultFile,
            {'DataFile':DataFile,
             'IndexList_test':IndexList_test, 'IndexList_train':IndexList_train,             
             'VMMAE':VMMAE, 'VMNMAE':VMNMAE,
             'S11MAE':S11MAE, 'S11NMAE':S11NMAE,
             'S22MAE':S22MAE, 'S22NMAE':S22NMAE,
             'S12MAE':S12MAE, 'S12NMAE':S12NMAE,
             'VMAE':VMAE,'APE':VMAPE,
             'S11AE':S11AE,'S11APE':S11APE,
             'S22AE':S22AE,'S22APE':S22APE,
             'S12AE':S12AE,'S12APE':S12APE})
#%%
# to save the predicted stress distribution during cross validation
# insert the function below to the code block of the cross validation
# sio.savemat('StressData_pred.mat', {'Sp':Sp, 'idx_test':idx_test})
#%% show the time cost on the testing set: input shape, output stress
t_start=time.clock()
X_t=ShapeEncoder.predict(ShapeData_test.transpose(), batch_size=100, verbose=0)    
Yp=NMapper.predict(X_t, batch_size=idx_test.size, verbose=0)    
for n in range(0, 64):
    Yp[:,n]=Yp[:,n]*L[n]
```

```
#end
Ypp = numpy.array([])
Ypp.resize((idx_test.size,1,1,64))
for n in range(0, idx_test.size):
    Ypp[n,0,0,:]=Yp[n,:]
#end
Spp=StressDecoder.predict(Ypp);
Sp = numpy.array([])
Sp.resize((15000, idx_test.size))
for n in range(0, idx_test.size):
    tempS11=Spp[n,:,:,0];
    tempS11=tempS11.reshape((5000), order='F')
    Sp[0:5000,n]=tempS11
    tempS22=Spp[n,:,:,1];
    tempS22=tempS22.reshape((5000), order='F')
    Sp[5000:10000,n]=tempS22
    tempS12=Spp[n,:,:,2];
    tempS12=tempS12.reshape((5000), order='F')
    Sp[10000:15000,n]=tempS12
#end
Sp=numpy.asmatrix(Sp)  
t_end=time.clock()
print('Time Cost Per Testing Sample ', (t_end-t_start)/73)
```