Investigating manifold-valued data in neural networks

Adam Smith

August 12, 2019

Contents

1 Objectives

The objective of this project is to study and evaluate methods for training neural networks on manifold-valued data. Current methods often neglect the manifold structure, and lead to ineffective and/or inefficient networks. Preprocessing techniques and various network structures will be studied for data on various manifolds with the goal of obtaining an efficient and effective neural network architecture.

2 Generating Data in $\mathbb{R}$

The Van der Pol oscillator is a non-conservative oscillator with non-linear damping evolving according to the second-order chaotic differential equation

$$\frac{d^2x}{dt^2} - \alpha(1 - x^2)\frac{dx}{dt} + \beta x = f(t) \tag{1}$$

where x is the position coordinate, α and β are scalar parameters, and $f : \mathbb{R} \to \mathbb{R}$ is a periodic forcing function. This equation was solved using a numerical ODE solver under various conditions to produce real-valued data.

3 Standard Neural Network

Data was generated for 600 random values of α and β in $[0, 1]$ with $f(t) = \cos(\omega t)$ for a fixed ω and the initial condition randomized. Using *PyTorch*, an open source machine learning library for Python, a feedforward neural network (nn_param.py) was created taking numerical solutions of equation (1) as input and outputting predictions for the parameters α and β. As seen in Figure 1, moderate performance of the neural network required a high density of data points over a small time window, and performance significantly decreased with a larger time step.

4 Convolutional Neural Network

Data was generated as in section 2. The data was then plotted on a two dimensional histogram of $x(t)$ vs. $t \mod T$, where $T = 2\pi/\omega$ is the period of f. A convolutional neural network (nn_hist.py) was created taking a 2D histogram as input. The network consists of two convolutional layers each with maximum pooling operations, followed by four fully connected linear layers with a sigmoid activation function interlaced dropout layers for regularization. The forcing function $f = \cos(\omega t+\phi)$ was then given a random phase $\phi \in [0, 2\pi)$ and the network was modified slightly to include the phase into training of the network. The results are summarized in Table 1.

Forcing function	$f_1 = \cos(\omega t)$	$f_2 = \cos(\omega t + \phi)$	$f_2 = \cos(\omega t + \phi)$
Phase included in training	No	No	Yes
Mean percent error	$6.8 \pm 1.2\%$	$16.0 \pm 1.1\%$	$14.1 \pm 1.3\%$

Table 1: Mean percent error in 10 runs of the convolutional neural network. Each run consisted of 50 epochs and trained on 560 data points.

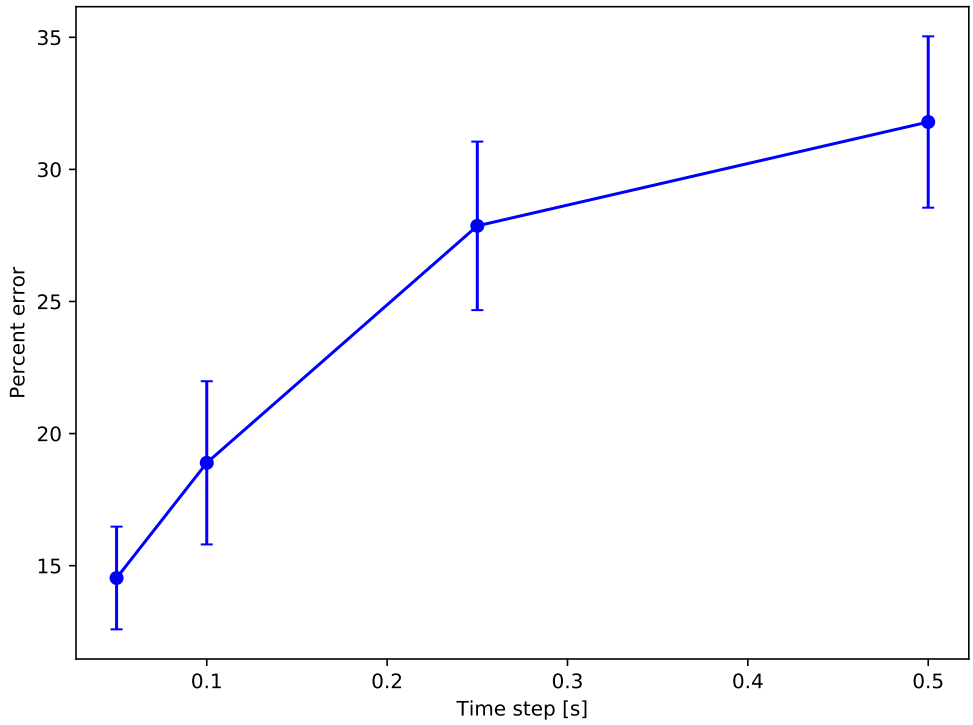

Figure 1: Standard neural network performance after 50 epochs training on 600 data points, each with 1000 time steps.

5 Fourier Transformed Data

600 data points, each consisting of $N = 1024$ time steps, were generated using the forcing functions

$$f_1 = \cos(\omega_1 t + \phi_1), \tag{2}$$

$$f_2 = C_1 \cos(\omega_1 t + \phi_1) + C_2 \cos(\omega_2 t + \phi_2), \text{ and} \tag{3}$$

$$f_3 = C_1 \cos(\omega_1 t + \phi_1) + C_2 \cos(\omega_2 t + \phi_2) + C_3 \cos(\omega_3 t + \phi_3). \tag{4}$$

Here, each C_i and ω_i was fixed, and each $\phi_i \in [0, 2\pi)$ was randomly selected for each solution of equation (1). A standard feedforward neural network (nn_ft.py) consisting of three fully connected linear layers interlaced with dropout layers was created. The fast Fourier transform (FFT) of the data points was then taken, and the network was trained on the absolute value of the FFT, the real part, the imaginary part, the peaks in the absolute value, and combinations thereof. The results are summarized in Table 2.

As is evident from Table 2, the most consistently effective and fastest method was to train the network on the highest peaks in the absolute value of the FFT. However, the number of peaks required for good performance increased with the complexity of the forcing function. Figure 2 summarizes the performance of the neural network for different numbers of peaks chosen.

Part of FFT used in training	Forcing function			Mean training time [seconds]
	f_1	f_2	f_3	
Real part	$31.8 \pm 2.1\%$	$26.3 \pm 1.4\%$	$31.2 \pm 1.5\%$	136.5
Imaginary part	$30.8 \pm 1.8\%$	$27.5 \pm 1.6\%$	$30.2 \pm 2.1\%$	135.2
Real and Imaginary parts	$29.7 \pm 1.5\%$	$26.5 \pm 1.3\%$	$29.3 \pm 1.6\%$	210.1
Absolute value	$5.6 \pm 0.5\%$	$8.8 \pm 1.1\%$	$10.7 \pm 1.2\%$	139.9
Highest peaks in absolute value	$7.5 \pm 0.7\%$	$7.0 \pm 0.6\%$	$10.2 \pm 0.8\%$	70.7

Table 2: Mean percent errors in 10 runs of the neural network. Each run consisted of 150 epochs and trained on 600 data points. The number of peaks to take was chosen to optimize performance on the particular data set. The highest 3 peaks were used for data corresponding to f_1, 6 peaks for f_2, and 17 peaks for f_3.

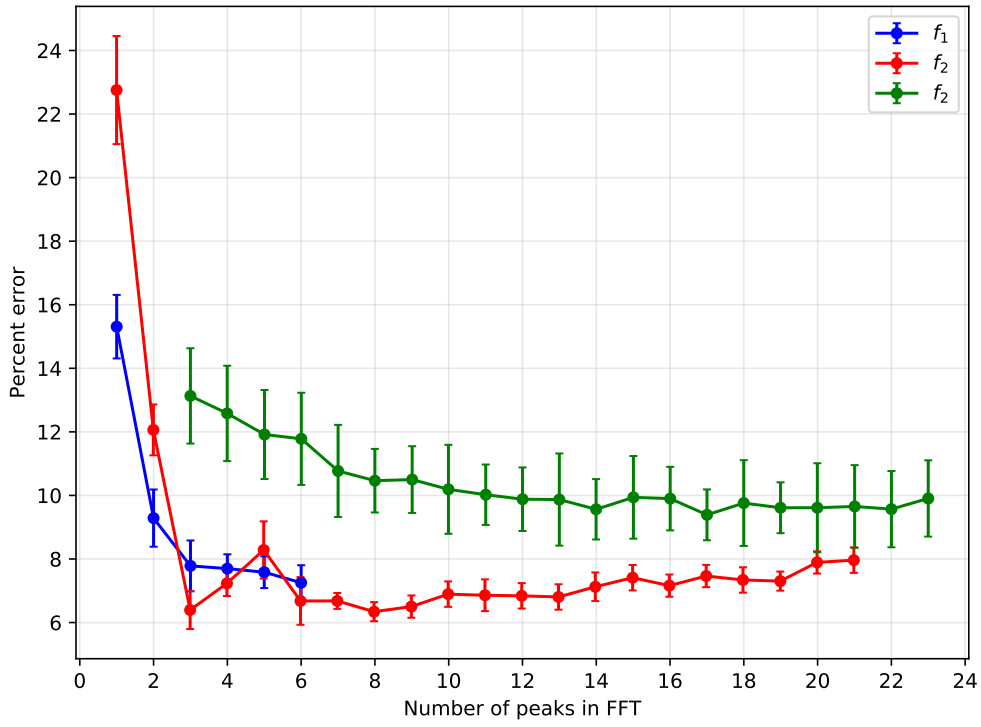

Figure 2: Mean percent errors in 10 runs of the neural network for datasets corresponding to the three forcing functions f_1, f_2, and f_3. Each run consisted of 150 epochs and trained on 600 data points.

6 Wavelet Analysis

A wavelet is a complex-valued function $\Psi : \mathbb{R} \to \mathbb{C}$ satisfying the following conditions:

$$\int_{-\infty}^{\infty} |\Psi(t)|^2 dt < \infty \tag{5}$$

$$c_\Psi := \int_{-\infty}^{\infty} \frac{|\Psi(\omega)|^2}{|\omega|} d\omega < \infty. \tag{6}$$

Intuitively, a wavelet is a wave-like oscillation with an amplitude that begins at zero, increases, then decreases back to zero. Given a wavelet Ψ, the continuous wavelet transform $S(a, b)$ of a real signal $s(t)$ is defined as

$$S(a, b) := \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} \Psi^*\Big(\frac{t-b}{a}\Big) s(t) dt \tag{7}$$

for $a > 0$ and $b \in \mathbb{R}$. The wavelet transform allows us to break up a signal into different frequency components, and then study the evolution of these components in time.
600 data points were generated using the forcing functions defined in equations (2), (3), and (4), and the continuous wavelet transform was taken to produce a 2-dimensional plot. Figure 3 gives an example of a wavelet transform of a data point and the computed local extrema.

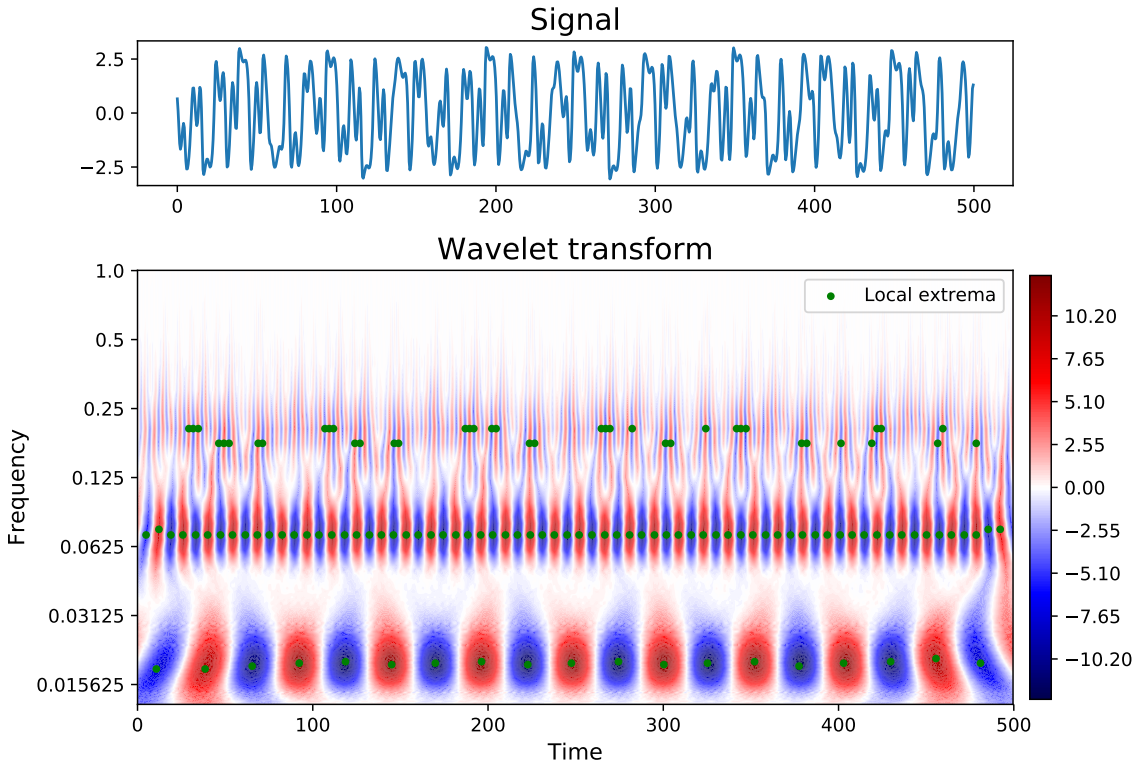

Figure 3: Example of a wavelet transform of a solution of equation (1) using the Morlet wavelet.

A feedforward neural network with dropout (nn_wavelet.py) consisting of three fully connected linear layers was created, and the time, frequency, and value of the extrema were fed into the network as inputs. The network performed poorly, achieving no more better than 30% error, and often worse on more complex datasets. A convolutional neural network similar to the one described in section 3 was then developed. However, the size of the matrix representing the wavelet transform made training slow and impractical, and this network was not pursued further.

7 Data on a Sphere

A multi-dimensional adaption of the Van der Pol equation

$$\frac{d^2\mathbf{x}}{dt^2} - \alpha \cdot (1 - |\mathbf{x}|^2)\frac{d\mathbf{x}}{dt} + \beta \cdot \mathbf{x} = \mathbf{g}(t) \tag{8}$$

where $\mathbf{x} \in \mathbb{R}^2$ is the position vector, $\alpha, \beta \in \mathbb{R}^2$ are parameters, and $\mathbf{g} : \mathbb{R} \to \mathbb{R}^2$ is a periodic forcing function was numerically solved 600 times for 1024 points on the interval $t \in [0, 500]$. This was done for the three forcing functions

$$\mathbf{g}_1 = (C_1 \cos \omega_1 t, C_2 \cos \omega_2 t), \tag{9}$$

$$\mathbf{g}_2 = (C_1 \cos(\omega_1 t + \phi_1), C_2 \cos \omega_2 t), \text{ and} \tag{10}$$

$$\mathbf{g}_3 = (C_1 \cos(\omega_1 t + \phi_1), C_2 \cos(\omega_2 t + \phi_2)) \tag{11}$$

for fixed values of C_1, C_2, ω_1, and ω_2, and random values of $\phi_1, \phi_2 \in [0, 2\pi)$. The data was the mapped to the sphere $\mathbb{S}^2$ via the mapping

$$\mathbf{x} = (x_1, x_2) \mapsto (\sin x_1 \cos x_2, \sin x_1 \sin x_2, \cos x_1). \tag{12}$$

A feedforward neural network with dropout (nn_2sphere.py) was created consisting of three fully connected linear layers. For data generated with the forcing function $\mathbf{g}_1$, training the network directly on the data points yields a network with mean percent error of $7.1 \pm 0.4\%$ after 40 epochs. For data generated with forcing functions $\mathbf{g}_2$ or $\mathbf{g}_3$, a more sophisticated network is required to obtain a good accuracy. The most effective method was found to be first projecting the data onto $\mathbb{R}^2$ via the stereographic projection

$$(x, y, z) \mapsto \Big(\frac{x}{1-z}, \frac{y}{1-z}\Big), \tag{13}$$

and then training on the highest peaks of the Fourier transform of each component. Using this method, a mean percent error of $25.3 \pm 1.1\%$ is achieved for data corresponding to $\mathbf{g}_2$ and $25.3 \pm 1.8\%$ for data corresponding to $\mathbf{g}_3$.

8 ManifoldNet

ManifoldNet is a deep network framework for manifold-valued data presented in [1] employing the use of the weighted Fréchet mean (wFm) as a convolution operation. This network architecture was applied to the dataset generated on $\mathbb{S}^2$ to create a network (nn_wfm_s2.py) consisting of one convolutional layer followed by a wFm and geodesic distance operation and fully connected linear layer. The ManifoldNet architecture was also applied to SO(3) matrices derived from a numerical solution of Euler's equation for rigid body motion (nn_wfm_so3.py). For both cases, the performance of the network was quite poor, achieving not better than 30% error on the validation dataset. Moreover, the network takes roughly 10 minutes per epoch to run, making it much slower and less practical than previous methods.

9 References

[1] R. Chakraborty, J. Bouza, J. Manton, and B. Vemuri. ManifoldNet: A Deep Network Framework for Manifold-valued Data. arXiv:1809.06211, 2018.

10 Sample of Code

```
"""
Created on Thu Aug 1 10:13:27 2019

@author: adamreidsmith
"""

'''
ManifoldNet neural network based of an algorithm presented here:
    https://arxiv.org/abs/1809.06211
Trains on SO(3) matrices.
'''

import torch
import numpy as np
from torch import nn
from torch.nn.parameter import Parameter
from torch.utils.data import Dataset, DataLoader, random_split
import matplotlib.pyplot as plt
from torch import autograd
from so3_data_generation import generate_data

###############################################################################
'''
Inputs:
    n_epochs:           Number of epochs to train for.

    batch_size:         Batch size.

    lr:                 Learning Rate.

    weight_decay:       Weight decay factor.

    lr_factor:          Learning rate decay factor.  Learning rate is multiplied
                            by this factor every epoch.

    loss_function:      Loss function. Can be:
                            'mean square':    loss = sum((x_i - y_i)^2)
                            'log cosh':       loss = sum(log(cosh(x_i - y_i)))
                            'abs':            loss = sum(|x_i - y_i|)

    n:                  Number of time series to generate. See
                        'so3_data_generation.py'.

    tn:                 Number of SO(3) matrics in each time series. See
                        'so3_data_generation.py'.

    noise_level:        See 'so3_data_generation.py'.
'''
###############################################################################

def main(n_epochs=15,
         batch_size=5,
         lr=0.001,
         lr_factor=0.95,
         weight_decay=1e-8,
         loss_function='abs',
```

```
        n=1000,
        tn=500,
        noise_level=1e-4):

class Data(Dataset):

    def __init__(self):
        print('\nGenerating data...')

        self.I = [self.make_moments(0.1, 2) for _ in range(n)]

        self.data = [generate_data(self.I[i], 1, tn, noise_level)
                     for i in range(n)]
        self.data = torch.Tensor(self.data)
        self.len = self.data.shape[0]

        self.I = torch.Tensor(self.I)

        self.I_reduced = [[torch.div(I[0],I[2]), torch.div(I[1],I[2])]
                          for I in self.I]
        self.I_reduced = torch.Tensor(self.I_reduced)


    __getitem__ = lambda self, index: (self.data[index], self.I_reduced[index])

    __len__ = lambda self: self.len

    def make_moments(self, minimum, maximum):
        '''
        Moments of inertia (can be any non-negative real number)
        '''
        I1 = np.random.uniform(minimum,maximum)
        I2 = np.random.uniform(minimum,maximum)
        I3 = np.random.uniform(minimum,maximum)

        while I1+I2<I3 or I1+I3<I2 or I2+I3<I1:
            I3 = np.random.uniform(minimum,maximum)

        return [I1,I2,I3]

dataset = Data()

# Lengths of the training and validation datasets
train_len = int(0.75*dataset.len)
valid_len = dataset.len - train_len

#Randomly split the data into training and validation datasets
train_data, valid_data = random_split(dataset, (train_len, valid_len))

train_loader = DataLoader(dataset=train_data, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(dataset=valid_data, batch_size=batch_size, shuffle=True)

#Weighted Frechet mean
def wFM(X, w):
    '''
    Weighted frechet mean of SO3 matrices
    X can be a list of matrices or a list of lists of matrices
    '''
    if len(X.shape) == 4:
```

```
        M = X[:,0]
        for i in range(1,X.shape[1]):
            M = so3_geodesic(M, X[:,i], torch.div(w[i], w[:i+1].sum()))
        return M
    else:
        M = X[0]
        for i in range(1,X.shape[0]):
            M = so3_geodesic(M, X[i], torch.div(w[i], w[:i+1].sum()))
        return M

def powerm(A,k):
    '''
    Computes matrix power A^k where k is a non-negative integer
    A can be a square matrix or a batch of square matrices
    '''
    if k == 0 and len(A.shape) == 2:
        return torch.eye(A.shape[-1])

    if k == 0 and len(A.shape) == 3:
        inter = torch.eye(A.shape[-1]).unsqueeze(0)
        return inter.repeat(A.shape[0],1,1)

    if k == 1:
        return A

    P = torch.matmul(A,A)
    for i in range(k-2):
        P = torch.matmul(P,A)
    return P

def expm(A, tol=1e-8, maxit=20):
    '''
    Matrix exponential using power series definition
    A can be a square matrix or a batch of square matrices
    '''
    shape = A.shape
    batched = False if len(shape) == 2 else True
    eps = 10.0
    exp = powerm(A,0)
    k = 1
    while eps > tol and k < maxit:
        old_exp = exp.clone()
        exp += torch.mul(1/np.math.factorial(k), powerm(A,k))
        k += 1

        if batched:
            inter1 = exp - old_exp
            inter2 = torch.mul(inter1, inter1).view(shape[0], shape[1]*shape[2])
            eps = torch.max(torch.sqrt(inter2).sum(dim=1))
        else:
            eps = torch.norm(exp - old_exp)

    return exp

def trace(A):
    '''
    Trace of a matrix or a batch of matrices
    '''
    shape = A.shape
```

```
    if len(shape) == 2:
        return torch.trace(A)
    mask = torch.eye(shape[-1]).unsqueeze(0).repeat(shape[0],1,1)
    return torch.mul(mask, A).view(shape[0], shape[1]*shape[2]).sum(dim=1)

def so3_geodesic(A1,A2,t):
    '''
    Parameterized geodesic curve in SO(3)
    A1 and A2 can be square matrices of the same size
    or batches of square matrices of equal size
    '''
    P = torch.matmul(torch.inverse(A1), A2)
    G = logm(P)
    O = torch.matmul(A1, expm(torch.mul(t, G)))
    return O

def logm(A):
    '''
    Matrix logarithm
    A can be a square matrix or a batch of square matrices
    '''
    shape = A.shape

    theta = torch.acos(torch.div(trace(A) - 1, 2))
    C = torch.div(theta, torch.mul(2, torch.sin(theta)))

    if len(shape) == 2:
        G = torch.mul(C, A - A.transpose(0,1))
    else:
        inter = A - A.transpose(1,2)
        G = torch.mul(C.unsqueeze(1), inter.view(shape[0], shape[1]*shape[2]))
        G = G.view(shape)

    return G

def geo_dist(A,B):
    '''
    Geodesic distance between SO(3) matrices
    Either x1 and x2 are the same shape, or x1 is a single point
    and x2 is a tensor of points
    '''
    C = torch.matmul(A.t(), B)
    return torch.abs(torch.acos((trace(C) - 1)/2))


#Custom convolutional layer based on the weighted Frechet mean
class wFMLayer(nn.Module):

    def __init__(self, in_channels=1, out_channels_per_ic=1, kernel_size=3,
                 stride=1):
        super(wFMLayer, self).__init__()
        self.kernel_size = kernel_size
        self.stride = stride
        self.in_channels = in_channels
        self.out_channels_per_ic = out_channels_per_ic

        self.weights = Parameter(torch.Tensor(in_channels,
                                              out_channels_per_ic,
                                              kernel_size))
```

```
        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.kaiming_uniform_(self.weights, a=np.sqrt(5))
        self.weights.data = self.weights.data / torch.max(self.weights.data)
        self.weights.data = self.weights.data * torch.sign(self.weights.data)

    def forward(self, x):
        #x has shape (batch_size, in_channels, tn, 3, 3)
        for k in range(x.shape[0]):
            for j in range(self.in_channels):
                #Partition x into batches of size self.kernel_size
                X = [x[k][j][m:m + self.kernel_size].tolist()
                     for m in range(0, len(x[k][j]), self.stride)
                     if len(x[k][j][m:m + self.kernel_size]) ==
                        self.kernel_size]

                inter = x[k][j][-self.kernel_size:].tolist()
                if inter != X[-1]:
                    X.append(x[k][j][-self.kernel_size:].tolist())
                X = torch.Tensor(X)

                #X has shape [len(X), self.kernel_size, 3, 3]
                #len(X) depends on the shape of x relative to self.kernel_size
                #X can be thought of as a list of lists of 3x3 matrices

                if k == 0 and j == 0:
                    wFM_convolutions = torch.empty(x.shape[0],
                        self.in_channels*self.out_channels_per_ic,
                        len(X), 3, 3)

                for i in range(self.out_channels_per_ic):
                    wFM_convolutions[k][j * self.out_channels_per_ic + i] =
                        wFM(X, self.weights[j][i])

        #wFM_convolitions has shape
        #(batch_size, self.in_channels*self.out_channels_per_ic, len(X), 3, 3)
        return wFM_convolutions


class LastLayer(nn.Module):

    def __init__(self, xshape, out_channels):
        super(LastLayer, self).__init__()
        self.xshape = xshape
        self.FCLayer = nn.Linear(xshape[1]*xshape[2], out_channels)

    def forward(self, x):
        O = torch.Tensor(self.xshape[0], self.xshape[1]*self.xshape[2])

        for k in range(x.shape[0]):
            #x[k] consists of all Z_i's in a batch item.
            #Each Z_i is a list of points on sphere
            X = x[k].view(x.shape[1]*x.shape[2], x.shape[3], x.shape[4])

            lenX = X.shape[0]

            M_u = wFM(X, (1/lenX)*torch.ones(lenX))
```

```
            O_i = geo_dist(M_u, X)

            O[k] = O_i

        return self.FCLayer(O)


#Model of the neural network
class Model(nn.Module):

    def __init__(self):
        super(Model, self).__init__()

        self.wfmconv1 = wFMLayer(in_channels=1,
                                 out_channels_per_ic=2,
                                 kernel_size=5,
                                 stride=2)

        self.LL = LastLayer(xshape=[batch_size,2,249,3,3], out_channels=2)

    def forward(self, x):

        x = self.wfmconv1(x)

        return self.LL(x)

    def normalize_weights(self, weights):
        weights = torch.div(weights, torch.max(weights))
        return weights * torch.sign(weights)

    def check_normalize(self, weights):
        if torch.min(weights) < 0 or torch.max(weights) > 1:
            return self.normalize_weights(weights)
        return weights

model = Model()

if loss_function == 'mean square':
    loss_func = nn.MSELoss()
elif loss_function == 'log cosh':
    loss_func = lambda x, y: torch.log(torch.cosh(2*(x - y))).sum()
elif loss_function == 'abs':
    loss_func = lambda x, y: torch.abs(x-y).sum()
else:
    raise RuntimeError('loss_function not recognized. ' +
            'Set loss_function to \'mean square\' or \'log cosh\'')

#Optimizer
optimizer = torch.optim.Adam(model.parameters(),
                             lr=lr,
                             weight_decay=weight_decay)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
                                                   gamma=lr_factor)

def evaluate():
    #Evaluation mode
    model.eval()
    for data in valid_loader:
```

```
            #Split batch into inputs and outputs
            x, y = data[0], data[1]
            #x is a batch of time series of matrices
            #y is a batch of corresponding principle moments of inertia

            #Forward propagation
            out = model(x)

            #Loss computation
            loss = loss_func(out, y)

            #Save training loss in this batch
            if not torch.isnan(loss) and not torch.isinf(loss):
                valid_loss.append(loss.item())

            #Compute the average percent error over a validation batch
            if torch.isnan(out).sum().item() == 0 and
               torch.isinf(out).sum().item() == 0:
                percent_error = 100*torch.div(torch.abs(out - y), y)
                all_percent_error.extend(percent_error.flatten().squeeze(0).tolist())

    return valid_loss

def train():
    #Training mode
    model.train()
    assert len(train_loader) % batch_size == 0,
           '\'train_loader\' must have length divisible by \'batch_size\''
    assert len(valid_loader) % batch_size == 0,
           '\'valid_loader\' must have length divisible by \'batch_size\''

    for data in train_loader:

        #Split batch into inputs and outputs
        x, y = data[0], data[1]
        #x is a batch of time series of matrices
        #x has shape (batch_size, in_channels, tn, 3, 3)
        #y is a batch of corresponding principle moments of inertia

        #torch.set_anomaly_enabled(False)
        def closure():
            #Reset gradients to zero
            optimizer.zero_grad()

            #Forward propagation
            out = model(x)

            #Loss computation
            loss = loss_func(out, y)

            #Backpropagation
            try:
                with autograd.detect_anomaly():
                    loss.backward()
            except:
                pass

            return loss
```

```
            optimizer.step(closure)

            #Keep weights in [0,1]
            model.wfmconv1.weights.data =
                model.check_normalize(model.wfmconv1.weights.data)

            #Save training loss in this batch
            l = loss_func(model(x), y)
            if not torch.isnan(l) and not torch.isinf(l):
                train_loss.append(l.item())

    return train_loss

def plot_hist():
    '''
    Plot histograms of the error (Predicted - True) in the predicted data
    '''
    error = []
    model.eval()
    for data in valid_loader:
        #Split batch into inputs and outputs
        x, y = data[0], data[1]

        out = model(x)
        error.append((out - y).detach().numpy())

    error = np.array(error)

    params = ['I1/I3','I2/I3']
    colors = ['b','r']
    for i in range(2):
        relevant_error = np.array([error[j][k][i]
                                   for j in range(len(error))
                                   for k in range(batch_size)])

        plt.figure(figsize=(8,6))
        plt.hist(relevant_error, bins=30, color=colors[i])
        plt.title('Prediction error in parameter \'' + params[i] +
                  '\' in validation data')
        plt.xlabel('Predicted - True')

    plt.figure(figsize=(8,6))
    p_err_less_100 = [i for i in all_percent_error if i <= 100]
    n_more_100 = len(all_percent_error) - len(p_err_less_100)
    plt.hist(p_err_less_100, bins=30)
    plt.text(x=plt.xlim()[1]-35,
             y=plt.ylim()[1]-20,
             s='More than 100% error:\n'+str(n_more_100))
    plt.xlabel('Percent Error')
    plt.title('Histogram of percent errors in predictions of validation data')

    plt.show()

#Print statistics about the current run
print('\nModel Information:\n', model, sep='')
print('\nRun Start',
      '\n  Batch size:', batch_size,
      '\n  Epochs:',  n_epochs,
      '\n  Training data size:', len(train_loader)*batch_size,
```

```
        '\n  Validation data size:', len(valid_loader)*batch_size,
        '\n  Learning rate:', lr,
        '\n  LR decay factor:', lr_factor,
        '\n  Weight decay:', weight_decay,
        '\n  Loss function:', loss_function,
        '\n  Optimizer:', repr(optimizer).partition('(')[0],
        '\n  LR scheduler:', repr(scheduler)[repr(scheduler).find('er.')
                             +3:repr(scheduler).find(' obj')],
        '\n')

    #Training and evaluation loop
    for epoch in range(n_epochs):
        #An epoch is a run of the entire training dataset

        train_loss, valid_loss, all_percent_error = [], [], []

        #Train the network
        train_loss = train()

        #Evaluate the network
        valid_loss = evaluate()

        if (epoch+1) % 1 == 0:
            print('Epoch:', epoch+1,
                  '\n  Learning rate:              ', scheduler.get_lr()[0],
                  '\n  Mean epoch training loss:  ', np.mean(train_loss),
                  '\n  Mean epoch validation loss:', np.mean(valid_loss),
                  '\n  Overfitting factor:        ',
                        np.mean(valid_loss)/np.mean(train_loss),
                  '\n  Median percent error:      ',
                        np.median(np.array(all_percent_error)), '%')

        #Update the learing rate
        scheduler.step()

    if n_epochs:
        plot_hist()

main()
```