evaluation_test

March 8, 2024

1 Loading Data

This is the pre-test part of the project that consists of replicating Mariel's code to properly load and preprocess the provided data. In here you can find code to load data, put everything in a very handable data structure and format, preprocess the joint positions so that they belong to the same unit cube (since we are interested in relative motion instead of absolute motion), and finally compute the edges.

[1]:
```
import torch
from torch_geometric.data import Data
import numpy as np
from glob import glob
import os
```

[2]:
```
point_labels =␣
↪['ARIEL','C7','CLAV','LANK','LBHD','LBSH','LBWT','LELB','LFHD','LFRM','LFSH','LFWT','LHEL',

reduced_joint_names =␣
↪['ARIEL','CLAV','RFSH','LFSH','RIEL','LIEL','RIWR','LIWR','RKNE','LKNE','RTOE','LTOE','LHEL

skeleton_lines = [
#     ( (start group), (end group) ),
    (('LHEL',), ('LTOE',)), # toe to heel
    (('RHEL',), ('RTOE',)),
    (('LMT1',), ('LMT5',)), # horizontal line across foot
    (('RMT1',), ('RMT5',)),
    (('LHEL',), ('LMT1',)), # heel to sides of feet
    (('LHEL',), ('LMT5',)),
    (('RHEL',), ('RMT1',)),
    (('RHEL',), ('RMT5',)),
    (('LTOE',), ('LMT1',)), # toe to sides of feet
    (('LTOE',), ('LMT5',)),
    (('RTOE',), ('RMT1',)),
    (('RTOE',), ('RMT5',)),
    (('LKNE',), ('LHEL',)), # heel to knee
    (('RKNE',), ('RHEL',)),
    (('LFWT',), ('RBWT',)), # connect pelvis
    (('RFWT',), ('LBWT',)),
```

```
    (('LFWT',), ('RFWT',)),
    (('LBWT',), ('RBWT',)),
    (('LFWT',), ('LBWT',)),
    (('RFWT',), ('RBWT',)),
    (('LFWT',), ('LTHI',)), # pelvis to thighs
    (('RFWT',), ('RTHI',)),
    (('LBWT',), ('LTHI',)),
    (('RBWT',), ('RTHI',)),
    (('LKNE',), ('LTHI',)),
    (('RKNE',), ('RTHI',)),
    (('CLAV',), ('LFSH',)), # clavicle to shoulders
    (('CLAV',), ('RFSH',)),
    (('STRN',), ('LFSH',)), # sternum & T10 (back sternum) to shoulders
    (('STRN',), ('RFSH',)),
    (('T10',), ('LFSH',)),
    (('T10',), ('RFSH',)),
    (('C7',), ('LBSH',)), # back clavicle to back shoulders
    (('C7',), ('RBSH',)),
    (('LFSH',), ('LBSH',)), # front shoulders to back shoulders
    (('RFSH',), ('RBSH',)),
    (('LFSH',), ('RBSH',)),
    (('RFSH',), ('LBSH',)),
    (('LFSH',), ('LUPA',),), # shoulders to upper arms
    (('RFSH',), ('RUPA',),),
    (('LBSH',), ('LUPA',),),
    (('RBSH',), ('RUPA',),),
    (('LIWR',), ('LIHAND',),), # wrist to hand
    (('RIWR',), ('RIHAND',),),
    (('LOWR',), ('LOHAND',),),
    (('ROWR',), ('ROHAND',),),
    (('LIWR',), ('LOWR',),), # across the wrist
    (('RIWR',), ('ROWR',),),
    (('LIHAND',), ('LOHAND',),), # across the palm
    (('RIHAND',), ('ROHAND',),),
    (('LFHD',), ('LBHD',)), # draw lines around circumference of the head
    (('LBHD',), ('RBHD',)),
    (('RBHD',), ('RFHD',)),
    (('RFHD',), ('LFHD',)),
    (('LFHD',), ('ARIEL',)), # connect circumference points to top of head
    (('LBHD',), ('ARIEL',)),
    (('RBHD',), ('ARIEL',)),
    (('RFHD',), ('ARIEL',)),
]
```

[3]:
```
class MarielDataset(torch.utils.data.Dataset):
    'Characterizes a dataset for PyTorch'
```

```python
    def __init__(self, reduced_joints=False, xy_centering=True, seq_len=128, predicted_timesteps=1, file_path="data/mariel_*.npy", no_overlap=False):
        'Initialization'
        self.file_path      = file_path
        self.seq_len        = seq_len
        self.no_overlap     = no_overlap
        self.reduced_joints = reduced_joints # use a meaningful subset of joints
        self.data           = load_data(pattern=file_path)
        self.xy_centering   = xy_centering
        self.n_joints       = 53
        self.n_dim          = 6
        self.predicted_timesteps = predicted_timesteps

        print("")

        if self.no_overlap == True:
            print("Generating non-overlapping sequences...")
        else:
            print("Generating overlapping sequences...")

        if self.xy_centering == True:
            print("Using (x,y)-centering...")
        else:
            print("Not using (x,y)-centering...")

        if self.reduced_joints == True:
            print("Reducing joints...")
        else:
            print("Using all joints...")

    def __len__(self):
        'Denotes the total number of samples'
        if self.xy_centering:
            data = self.data[1] # choose index 1, for the (x,y)-centered phrases
        else:
            data = self.data[0] # choose index 0, for data without (x,y)-centering

        if self.no_overlap == True:
             # number of complete non-overlapping phrases
            return int(len(data)/self.seq_len)
        else:
            # number of overlapping phrases up until the final complete phrase
            return len(data)-self.seq_len

    def __getitem__(self, index):
        'Generates one sample of data'
```

```python
        edge_index, is_skeleton_edge, reduced_joint_indices = edges(reduced_joints=self.reduced_joints, seq_len=self.seq_len)

        if self.xy_centering == True:
            data = self.data[1] # choose index 1, for the (x,y)-centered phrases
        else:
            data = self.data[0] # choose index 0, for data without (x,y)-centering

        if self.reduced_joints == True:
            data = data[:,reduced_joint_indices,:] # reduce number of joints if desired

        if self.no_overlap == True:
            # non-overlapping phrases
            index = index*self.seq_len
            sequence = data[index:index+self.seq_len]
            prediction_target = data[index:index+self.seq_len+self.predicted_timesteps]
        else:
            # overlapping phrases
            sequence = data[index:index+self.seq_len]
            prediction_target = data[index:index+self.seq_len+self.predicted_timesteps]

        sequence = np.transpose(sequence, [1,0,2]) # put n_joints first
        sequence = sequence.reshape((data.shape[1],self.n_dim*self.seq_len)) # flatten n_dim*seq_len into one dimension (i.e. node feature)
        prediction_target = np.transpose(prediction_target, [1,0,2]) # put n_joints first
        prediction_target = prediction_target.reshape((data.shape[1],self.n_dim*(self.seq_len+self.predicted_timesteps)))

        # Convert to torch objects
        sequence = torch.Tensor(sequence)
        prediction_target = torch.Tensor(prediction_target)
        edge_attr = torch.Tensor(is_skeleton_edge)

        return Data(x=sequence, y=prediction_target, edge_index=edge_index.t().contiguous(), edge_attr=edge_attr)

def load_data(pattern="data/mariel_*.npy"):
    # load up the six datasets, performing some minimal preprocessing beforehand
    datasets = {}
    ds_all = []
```

```
exclude_points = [26,53]
point_mask = np.ones(55, dtype=bool)
point_mask[exclude_points] = 0

for f in sorted(glob(pattern)):
    ds_name = os.path.basename(f)[7:-4]
    ds = np.load(f).transpose((1,0,2))
    ds = ds[500:-500, point_mask]
    ds[:,:,2] *= -1
    datasets[ds_name] = ds
    ds_all.append(ds)

ds_counts = np.array([ds.shape[0] for ds in ds_all])
ds_offsets = np.zeros_like(ds_counts)
ds_offsets[1:] = np.cumsum(ds_counts[:-1])

ds_all = np.concatenate(ds_all)
print("Original numpy dataset contains {:,} timesteps of {} joints with {}␣
↪dimensions each.".format(ds_all.shape[0], ds_all.shape[1], ds_all.shape[2]))

low,hi = np.quantile(ds_all, [0.01,0.99], axis=(0,1))
xy_min = min(low[:2])
xy_max = max(hi[:2])
xy_range = xy_max-xy_min
ds_all[:,:,:2] -= xy_min
ds_all *= 2/xy_range
ds_all[:,:,:2] -= 1.0

### It's also useful to have these datasets centered, i.e. with the x and y␣
↪offsets subtracted from each individual frame:
ds_all_centered = ds_all.copy()
ds_all_centered[:,:,:2] -= ds_all_centered[:,:,:2].
↪mean(axis=1,keepdims=True)

datasets_centered = {}
for ds in datasets:
    datasets[ds][:,:,:2] -= xy_min
    datasets[ds] *= 2/xy_range
    datasets[ds][:,:,:2] -= 1.0
    datasets_centered[ds] = datasets[ds].copy()
    datasets_centered[ds][:,:,:2] -= datasets[ds][:,:,:2].
↪mean(axis=1,keepdims=True)

### Calculate velocities (first velocity is always 0)
velocities = np.vstack([np.zeros((1,53,3)),np.array([35*(ds_all[t+1,:,:] -␣
↪ds_all[t,:,:]) for t in range(len(ds_all)-1)])]) # (delta_x/y/z per frame) *␣
↪(35 frames/sec)
```

```
    ### Stack positions above velocities
    ds_all = np.dstack([ds_all,velocities]) # stack along the 3rd dimension, i.
 ↪e. "depth-wise"
    ds_all_centered = np.dstack([ds_all_centered,velocities]) # stack along the␣
 ↪3rd dimension, i.e. "depth-wise"

    for data in [ds_all, ds_all_centered]:
        # Normalize locations & velocities (separately) to [-1, 1]
        loc_min = np.min(data[:,:,:3])
        loc_max = np.max(data[:,:,:3])
        vel_min = np.min(data[:,:,3:])
        vel_max = np.max(data[:,:,3:])
        print("loc_min:",loc_min,"loc_max:",loc_max)
        print("vel_min:",vel_min,"vel_max:",vel_max)
        data[:,:,:3] = (data[:,:,:3] - loc_min) * 2 / (loc_max - loc_min) - 1
        data[:,:,3:] = (data[:,:,3:] - vel_min) * 2 / (vel_max - vel_min) - 1

    return ds_all, ds_all_centered, datasets, datasets_centered, ds_counts

def edges(reduced_joints, seq_len):
    ### Define a subset of joints if we want to train on fewer joints that␣
 ↪still capture meaningful body movement:
    if reduced_joints == True:
        reduced_joint_indices = [point_labels.index(joint_name) for joint_name␣
 ↪in reduced_joint_names]
        edge_index = np.array([(i,j) for i in reduced_joint_indices for j in␣
 ↪reduced_joint_indices if i!=j])
    else:
        reduced_joint_indices = None
        edge_index = np.array([(i,j) for i in range(53) for j in range(53) if i!
 ↪=j]) # note: no self-loops!

    skeleton_idxs = []
    for g1,g2 in skeleton_lines:
        entry = []
        entry.append([point_labels.index(l) for l in g1][0])
        entry.append([point_labels.index(l) for l in g2][0])
        skeleton_idxs.append(entry)

    is_skeleton_edge = []
    for edge in np.arange(edge_index.shape[0]):
        if [edge_index[edge][0],edge_index[edge][1]] in skeleton_idxs:
            is_skeleton_edge.append(torch.tensor(1.0))
        else:
            is_skeleton_edge.append(torch.tensor(0.0))
```

```
    is_skeleton_edge = np.array(is_skeleton_edge)
    copies = np.tile(is_skeleton_edge, (seq_len,1)) # create copies of the 1D␣
↪array for every timestep
    skeleton_edges_over_time = torch.tensor(np.transpose(copies))

    if reduced_joints == True:
        ### Need to remake these lists to include only nodes 0-18 now
        edge_index = np.array([(i,j) for i in np.
↪arange(len(reduced_joint_indices)) for j in np.
↪arange(len(reduced_joint_indices)) if i!=j])

    return torch.tensor(edge_index, dtype=torch.long),␣
↪skeleton_edges_over_time, reduced_joint_indices
```

2 Visualizing Dance

This is the first part of the test, in which I effectively started developing. In here I instantiated the `MarielDataset` class, experimented for quite a while with the data to understand what each part actually represented, then built up a static visualization scheme to make sure everything was in order and finally animated a sequence from the original dataset.

Note: I did not include the experimentation parts to this section of the notebook because I didn't want to make it even longer.

[4]:
```
# Importing required libraries
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
%matplotlib inline
from IPython.display import HTML
```

[5]:
```
# Creating dancer dataset and computing the number of used joints
#dancer_data = MarielDataset(reduced_joints=False, file_path='../data/
↪mariel_beyond.npy')
dancer_data = MarielDataset(reduced_joints=False)
n_joints = len(reduced_joint_names) if dancer_data.reduced_joints else␣
↪dancer_data.n_joints
```

```
Original numpy dataset contains 38,309 timesteps of 53 joints with 3 dimensions
each.
loc_min: -1.8967371874141707 loc_max: 1.5558704656286815
vel_min: -45.57506836403084 vel_max: 33.951220235113276
loc_min: -0.4843721412027978 loc_max: 0.9283637015363149
vel_min: -45.57506836403084 vel_max: 33.951220235113276

Generating overlapping sequences…
```

```
Using (x,y)-centering…
Using all joints…
```

[6]:
```python
# Animation function for a given sequence start and sequence length. You can␣
↪also pass your own sequence to animate it,
# although this part of the function is only becoming useful afterwards.
def animate_segments(start_segment, sequences, title, color='b',␣
↪given_sequence=None):

    # Animating initial dataset
    if given_sequence is None:
        segment_list = [start_segment + i*dancer_data.seq_len for i in␣
↪range(sequences)]
        edge_indexes = dancer_data[segment_list[0]].edge_index.numpy()
        joint_positions = []
        edge_attributes = []
        for segment in segment_list:
                try:
                    joint_positions = np.concatenate((joint_positions,␣
↪dancer_data[segment].x.view(n_joints, -1, 6)[:, :, :3].numpy()), axis=1)
                except:
                    joint_positions = dancer_data[segment].x.view(n_joints, -1,␣
↪6)[:, :, :3].numpy()

                try:
                    edge_attributes = np.concatenate((edge_attributes,␣
↪dancer_data[segment].edge_attr.numpy()), axis=1)
                except:
                    edge_attributes = dancer_data[segment].edge_attr.numpy()

    # Animating generated sequence
    else:
        edge_indexes = dancer_data[0].edge_index.numpy()
        edge_attributes = dancer_data[0].edge_attr.numpy()
        joint_positions = given_sequence

    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.grid(False)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_zticks([])
    ax.set_xlabel('')
    ax.set_ylabel('')
    ax.set_zlabel('')
    ax.set_xlim3d(-1, 0.5)
    ax.set_ylim3d(-1, 0.5)
```

```
    ax.set_ylim3d(-1, 0.5)
    ax.set_title(title)

    points = [ax.plot(joint_position[0, 0], joint_position[0, 1], \
                      joint_position[0, 2], 'o', color=color)[0] for 
joint_position in joint_positions]
    edge_lines = []

    # Update function to get new joint positions, clear edges and redraw tehm
    def update(num, joint_positions, points, ax, edge_indexes, edge_attributes, 
edge_lines):
        for joint_position, point in zip(joint_positions, points):
            new_x, new_y = joint_position[num, :2].tolist()
            point.set_data(new_x, new_y)
            point.set_3d_properties(joint_position[num, 2])

        for line in edge_lines:
            line.remove()
        edge_lines.clear()

        active_edges = np.where(edge_attributes.transpose()[num] == 1)
        for start_joint, end_joint in edge_indexes.transpose()[active_edges]:
            x = [joint_positions[start_joint][num, 0], 
joint_positions[end_joint][num, 0]]
            y = [joint_positions[start_joint][num, 1], 
joint_positions[end_joint][num, 1]]
            z = [joint_positions[start_joint][num, 2], 
joint_positions[end_joint][num, 2]]
            edge_line = ax.plot(x, y, z, color='black', alpha=0.5)[0]
            edge_lines.append(edge_line)


    animation = FuncAnimation(fig, update, frames=joint_positions.shape[1], 
fargs=(joint_positions, points, ax, \
                                                                edge_indexes, 
edge_attributes, edge_lines), interval=50)

    plt.close(fig)

    return animation

# Choosing the sequence start and the sequence length and making sure we are 
computing a sequence within bounds
start_segment = 3500
sequences = 2
```

```
while sequences > 0 and start_segment + (sequences-1)*dancer_data.seq_len >␣
↪len(dancer_data):
    sequences -= 1

if sequences > 0:
    animation = animate_segments(start_segment, sequences, "Visualizing an␣
↪Original Sequence")
else:
    print('Using previous sequence, because the current sequence is invalid.')

HTML(animation.to_jshtml())
```

```
/tmp/ipykernel_15487/1008845002.py:50: MatplotlibDeprecationWarning: Setting
data with a non sequence type is deprecated since 3.7 and will be remove two
minor releases later
  point.set_data(new_x, new_y)
```

```
[6]: <IPython.core.display.HTML object>
```

3 Training Generative Model

This is the second part of the test and the most difficult one. To make all the descriptions more clear, I separate them into different sections:

Implementation

I decided to go for the LSTM-VAE model. This decision was based in two main reasons:

- I wanted to replicate the ideas used in the provided paper. I understood that they resulted in a good model as described in the paper and also that I could try to use the given hyperparameters, making the search space for optimization much easier. Since optimizing NNs can often prove to be quite a challenge, I thought it would be a good idea considering the time schedule.
- I have much more experience with LSTM than with GNNs, so I thought I should stick to models I'm more familiar with because of time limitations. I figured I could explore more about GNN models within the development of the real project if I get accepted.

3.0.1 Architecture and Optimization

- One encoder with 2 LSTM layers (384 nodes) and 2 separated branches of linear layers (256 nodes each for the latent space), one for the mean and another one for log-variance.
- One decoder with 1 linear layer (384 nodes) with ReLU activation function for the latent-space sampled data and 2 LSTM layers (159 nodes for the output).

The model was trained with Adam optimizer for 200 epochs with early stopping at 3 validation losses higher than the best validation loss at that point. I also used the KL-divergence weight provided in the paper (0.0001). Finally, I added 0.2 dropout for the LSTM layers for some more regularization.

I expanded the dataset using data noise augmentation. I got around 10000 random sequences out of the almost 40000 provided sequences, and added 0.01 scaled Gaussian noise to the joint coordinates to try and make the model a bit more robust. I wanted to do even more augmentation, but due to my GPU limitations, this was the best I could do. Finally, I used 90% of the data for training and 10% for validation, both randomly sampled from the dataset and shuffled afterwards, with a batch sizes of 64.

3.0.2 Comments and Results

Even though I had reduced a lot the hyperparamter space by trying to replicate the provided paper, I still ended up having to train the model multiple times to find out the best hyperparameters. Not only that, but also I had to modify the number of LSTM layers in the model (from 3 in the original paper to 2) to make it more slightly more adequate to the amount of data/time I had without completly losing the fundamental temporal structure of the model.

Furthermore, I had issues with the validation loss that made me replicate the experiments an enormous amount of times. I had a decreasing validation loss, as expected, but still orders of magnitude larger than the training loss. I think this problem is mostly related to the model being a bit to complex for the amount of data I had ($\frac{2}{3}$ of the data size from the paper with much less augmentation due to GPU limitations). I tried reducing the amount of LSTM layers even further to make the model simpler, but found out it was not really capable of capturing the complexity of dance sequences, mostly generating sequences in which the figure stands almost still.

A better solution I could think of was to reduce the sequence length drastically (64 instead of 128) to both expand the dataset and make the sequences much more simple to learn. It indeed reduced the validation loss quite a lot, but generated very bad sequences (almost random joint positions and movement). In the end I did not have the time to properly evaluate all the hyperparameters possibilities for this reduced sequence model and went back to the original sequence lengths implementation that had much better results at least.

Even with all these issues, I managed to train the model and come up with some very interesting results. Some sequences from the original dataset are accurately reconstructed by the model. Others, even if not perfectly reconstructed, still clearly show that the model was able to capture the essence of their movements. A sequence that rotates, for example, remains rotating in its reconstructed version, or a sequence that lifts its leg remains with this movement in the reconstruction as well.

When it comes to generating new sequences, the model is quite sensitive to the standard deviation used. When the latent space is sampled with a normal distribution, the model generates interesting sequences, but with fewer movements than the original sequences. When the latent space is sampled with a higher standard deviation, the sequence tends to be more creative, but it is also common to see joints getting lost in space (many points converging to the same coordinates or points moving shakily).

Finally, one behavior I did not manage to fix was the initial state of the joints. Even in the best reconstructed/generated sequences, the joints start in weird positions, making the first miliseconds of the animation almost glitch to proper positions and then start a proper sequence of movements.

In the experiments below I show some of the good and bad results obtained.

[7]:
```
# Defining the VAE with LSTMs model, reparametrization trick (to be able to␣
↪properly backpropagate even with sampling), loss function (reconstruction
```

```python
# error + distribution similarity) and weight initialization (for proper␣
 ↪gradient propagation)
import torch.nn as nn

class Encoder(nn.Module):
    def __init__(self, in_dim, hid_dim, lat_dim, num_layers):
        super(Encoder, self).__init__()
        self.lstm_enc = nn.LSTM(in_dim, hid_dim, num_layers=num_layers,␣
 ↪dropout=0.2, batch_first=True)
        self.hid_mean = nn.Linear(hid_dim, lat_dim)
        self.hid_logvar = nn.Linear(hid_dim, lat_dim)

    def forward(self, x):
        _, (final_hid_state, _) = self.lstm_enc(x)
        mean = self.hid_mean(final_hid_state[-1])
        logvar = self.hid_logvar(final_hid_state[-1])
        return mean, logvar

class Decoder(nn.Module):
    def __init__(self, lat_dim, hid_dim, out_dim, num_layers):
        super(Decoder, self).__init__()
        self.lat_to_hid = nn.Sequential(nn.Linear(lat_dim, hid_dim), nn.ReLU())
        self.lstm_dec = nn.LSTM(hid_dim, out_dim, num_layers=num_layers,␣
 ↪dropout=0.2, batch_first=True)

    def forward(self, z, seq_len):
        hid = self.lat_to_hid(z)
        hid = hid.unsqueeze(1).repeat(1, seq_len, 1)
        outputs, _ = self.lstm_dec(hid)
        return outputs

class VAE(nn.Module):
    def __init__(self, in_dim, hid_dim, lat_dim, seq_len, num_layers):
        super(VAE, self).__init__()
        self.encoder = Encoder(in_dim, hid_dim, lat_dim, num_layers)
        self.decoder = Decoder(lat_dim, hid_dim, in_dim, num_layers)
        self.seq_len = seq_len
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else␣
 ↪'cpu')

    def forward(self, x):
        mean, logvar = self.encoder(x)
        z = reparametrization_trick(mean, logvar).to(self.device)
        return self.decoder(z, self.seq_len), mean, logvar

def reparametrization_trick(mean, logvar):
    std = torch.exp(logvar/2)
```

```
    return mean + torch.randn_like(std)*std

def loss_function(x_prime, x, mean, logvar):
    BCE = nn.functional.mse_loss(x_prime, x, reduction='sum')
    KLD = -torch.sum(1 + logvar - mean.pow(2) - logvar.exp())/2
    return BCE + 0.0001*KLD

def weight_initialization(model):
    if isinstance(model, nn.Linear):
        nn.init.xavier_uniform_(model.weight)
        nn.init.constant_(model.bias, 0)

    elif isinstance(model, nn.LSTM):
        for param in model.parameters():
            if len(param.shape) >= 2:
                nn.init.orthogonal_(param.data)

            else:
                nn.init.normal_(param.data)
```

[8]:
```
# Code to generate the dataset tensor from the provided data. This cell is␣
↪commented because you only run it once, save the tensor and start
# using the local version to avoid processing data for about 10 minutes every␣
↪run
# from tqdm import tqdm

# dance_sequences = []
# try:
#     for dance_seq in tqdm(dancer_data):
#         dance_seq = dance_seq.x.view(n_joints, -1, 6)[:, :, :3]
#         dance_sequences.append(dance_seq)
# except:
#     print('Value error on the last sequence.')

# dance_sequences = np.array(dance_sequences)
# ds_shape = dance_sequences.shape
# dance_sequences = np.swapaxes(dance_sequences, 1, 2).
↪reshape(ds_shape[0],ds_shape[2], n_joints*3)
# print(dance_sequences.shape)

# dance_sequences = torch.tensor(dance_sequences)
# torch.save(dance_sequences, 'data/dance_sequences.pt')
```

[9]:
```
# Creating datasets by reading the saved tensor, using augmentation on random␣
↪sequences and then random splitting data into training (90%) and
# validation (10%)
from torch.utils.data import TensorDataset, DataLoader, random_split
```

```python
# Reading tensor
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
dance_sequences = torch.load('data/dance_sequences.pt', map_location=lambda␣
↪storage, loc: storage.cuda(0))

# Data augmentation
s1, s2, s3 = dance_sequences.shape[0], dance_sequences.shape[1],␣
↪dance_sequences.shape[2]
noise = (torch.normal(mean=torch.zeros((int(s1/4), s2, s3)), std=torch.
↪rand((int(s1/4), s2, s3)))*0.01).to(device)

random_sequences = torch.randint(0, s1, (int(s1/4),))
aug_sequences = dance_sequences[random_sequences] + noise
del noise
del random_sequences
torch.cuda.empty_cache()

dance_sequences = torch.vstack((dance_sequences, aug_sequences))
del aug_sequences
torch.cuda.empty_cache()

# Experimenting with 64 seq_len instead of 128
#dance_sequences = dance_sequences.view(-1, int(s2/2), s3)

# Creating training and validation splits
dataset = TensorDataset(dance_sequences)
train_size = int(0.9*len(dataset))
train_dataset, val_dataset = random_split(dataset, [train_size, len(dataset) -␣
↪train_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=True)
```

[10]:
```python
# Training process: hyperparameters selection, model instantiation and training␣
↪loop
from tqdm import tqdm
import math

# Choosing hyperparameters to be the same as in the provided paper
in_dim = n_joints * 3
hid_dim = 384
lat_dim = 256
seq_len = 128
num_layers = 3
epochs = 150
```

```
# Instantiating model, moving it to device and initializing optimizer
model = VAE(in_dim, hid_dim, lat_dim, seq_len, num_layers)
model.to(device)
model.apply(weight_initialization)

optimizer = torch.optim.Adam(model.parameters())

# Training loop
best_loss = np.inf
best_model_path = 'data/best_training_model.pth'
early_stop_max = math.ceil(0.01*epochs)
early_stop_counter = 0
train_loss_series = []
val_loss_series = []
for epoch in tqdm(range(epochs)):
    model.train()
    train_loss = 0

    for batch_id, (dance_seq,) in enumerate(train_loader):
        optimizer.zero_grad()

        seq_prime, mean, logvar = model(dance_seq)

        loss = loss_function(seq_prime, dance_seq, mean, logvar)
        loss.backward()
        train_loss += loss.item()

        optimizer.step()

    avg_train_loss = train_loss/len(dancer_data)
    train_loss_series.append(avg_train_loss)
    print('Epoch: {}, Average training Loss: {}'.format(epoch, avg_train_loss))

    if epoch%5 == 0:
        model.eval()
        val_loss = 0

        with torch.no_grad():
            for (dance_seq,) in val_loader:
                seq_prime, mean, logvar = model(dance_seq)
                loss = loss_function(seq_prime, dance_seq, mean, logvar)
                val_loss += loss.item()

        avg_val_loss = val_loss/len(val_loader)
        val_loss_series.append(avg_val_loss)
        print('Epoch: {}, Average validation Loss: {}'.format(epoch,␣
 ↪avg_val_loss))
```

```
        if avg_val_loss < best_loss:
            best_loss = avg_val_loss
            early_stop_counter = 0

            torch.save(model.state_dict(), best_model_path)
            print('Saved new best model with validation loss: {}'.
↪format(best_loss))

        else:
            early_stop_counter += 1

    if early_stop_counter > early_stop_max:
        break
```

```
  0%|
| 0/150 [00:00<?, ?it/s]

Epoch: 0, Average training Loss: 662.5653920698552

  1%|
| 1/150 [00:23<58:40, 23.63s/it]

Epoch: 0, Average validation Loss: 32030.82354166667
Saved new best model with validation loss: 32030.82354166667

  1%|
| 2/150 [00:46<56:36, 22.95s/it]

Epoch: 1, Average training Loss: 519.5006268840154

  2%|
| 3/150 [01:08<55:50, 22.79s/it]

Epoch: 2, Average training Loss: 561.5282714344995

  3%|
| 4/150 [01:31<55:21, 22.75s/it]

Epoch: 3, Average training Loss: 532.9742323672933

  3%|
| 5/150 [01:54<54:55, 22.73s/it]

Epoch: 4, Average training Loss: 365.838541545175
Epoch: 5, Average training Loss: 354.56412418961247

  4%|
| 6/150 [02:17<55:27, 23.11s/it]

Epoch: 5, Average validation Loss: 20275.413346354166
Saved new best model with validation loss: 20275.413346354166
```

```
  5%|
| 7/150 [02:40<54:38, 22.93s/it]

Epoch: 6, Average training Loss: 347.10026367775777

  5%|
| 8/150 [03:02<53:52, 22.77s/it]

Epoch: 7, Average training Loss: 353.1374363371077

  6%|
| 9/150 [03:25<53:12, 22.64s/it]

Epoch: 8, Average training Loss: 349.6426158065878

  7%|
| 10/150 [03:47<52:39, 22.57s/it]

Epoch: 9, Average training Loss: 342.2437071748403
Epoch: 10, Average training Loss: 331.89507516802166

  7%|
| 11/150 [04:11<52:59, 22.87s/it]

Epoch: 10, Average validation Loss: 18289.2596484375
Saved new best model with validation loss: 18289.2596484375

  8%|
| 12/150 [04:33<52:14, 22.72s/it]

Epoch: 11, Average training Loss: 325.43011325142703

  9%|
| 13/150 [04:55<51:38, 22.62s/it]

Epoch: 12, Average training Loss: 329.1753437394622

  9%|
| 14/150 [05:18<51:06, 22.55s/it]

Epoch: 13, Average training Loss: 309.67951721989095

 10%|
| 15/150 [05:40<50:37, 22.50s/it]

Epoch: 14, Average training Loss: 293.97950964392874
Epoch: 15, Average training Loss: 286.81955967392565

 11%|
| 16/150 [06:04<50:59, 22.83s/it]

Epoch: 15, Average validation Loss: 15527.949765625
Saved new best model with validation loss: 15527.949765625

 11%|
| 17/150 [06:26<50:18, 22.70s/it]

Epoch: 16, Average training Loss: 280.1648583405052
```

```
 12%|
| 18/150 [06:49<49:42, 22.60s/it]

Epoch: 17, Average training Loss: 279.39836159970355

 13%|
| 19/150 [07:11<49:10, 22.53s/it]

Epoch: 18, Average training Loss: 274.4886077523578

 13%|
| 20/150 [07:33<48:41, 22.47s/it]

Epoch: 19, Average training Loss: 267.10513203275394
Epoch: 20, Average training Loss: 262.32625371559766

 14%|
| 21/150 [07:57<48:59, 22.79s/it]

Epoch: 20, Average validation Loss: 15807.633489583333

 15%|
| 22/150 [08:19<48:20, 22.66s/it]

Epoch: 21, Average training Loss: 265.25266692029004

 15%|
| 23/150 [08:42<47:46, 22.57s/it]

Epoch: 22, Average training Loss: 255.98077161903888

 16%|
| 24/150 [09:04<47:15, 22.51s/it]

Epoch: 23, Average training Loss: 276.05854616684445

 17%|
| 25/150 [09:26<46:47, 22.46s/it]

Epoch: 24, Average training Loss: 266.52801241434713
Epoch: 25, Average training Loss: 259.3441266752033

 17%|
| 26/150 [09:50<47:06, 22.79s/it]

Epoch: 25, Average validation Loss: 13903.291484375
Saved new best model with validation loss: 13903.291484375

 18%|
| 27/150 [10:12<46:27, 22.67s/it]

Epoch: 26, Average training Loss: 252.00863753686184

 19%|
| 28/150 [10:35<45:53, 22.57s/it]

Epoch: 27, Average training Loss: 250.38114919967452
```

```
 19%|
| 29/150 [10:57<45:23, 22.51s/it]

Epoch: 28, Average training Loss: 238.49209403137462

 20%|
| 30/150 [11:19<44:55, 22.46s/it]

Epoch: 29, Average training Loss: 230.5314934372524
Epoch: 30, Average training Loss: 223.47075595787274

 21%|
| 31/150 [11:43<45:13, 22.81s/it]

Epoch: 30, Average validation Loss: 12515.97248046875
Saved new best model with validation loss: 12515.97248046875

 21%|
| 32/150 [12:05<44:36, 22.68s/it]

Epoch: 31, Average training Loss: 226.0515244078984

 22%|
| 33/150 [12:28<44:02, 22.58s/it]

Epoch: 32, Average training Loss: 226.48270232563624

 23%|
| 34/150 [12:50<43:32, 22.52s/it]

Epoch: 33, Average training Loss: 224.293817209169

 23%|
| 35/150 [13:12<43:04, 22.48s/it]

Epoch: 34, Average training Loss: 240.43419155275612
Epoch: 35, Average training Loss: 228.1970019114342

 24%|
| 36/150 [13:36<43:18, 22.79s/it]

Epoch: 35, Average validation Loss: 13085.441640625

 25%|
| 37/150 [13:58<42:41, 22.67s/it]

Epoch: 36, Average training Loss: 222.40537922473712

 25%|
| 38/150 [14:21<42:08, 22.58s/it]

Epoch: 37, Average training Loss: 216.1139324259469

 26%|
| 39/150 [14:43<41:37, 22.50s/it]

Epoch: 38, Average training Loss: 214.06821088665257
```

```
 27%|
| 40/150 [15:05<41:10, 22.46s/it]

Epoch: 39, Average training Loss: 207.7882227434682
Epoch: 40, Average training Loss: 208.81305857374863

 27%|
| 41/150 [15:29<41:24, 22.79s/it]

Epoch: 40, Average validation Loss: 11358.200286458334
Saved new best model with validation loss: 11358.200286458334

 28%|
| 42/150 [15:51<40:48, 22.67s/it]

Epoch: 41, Average training Loss: 203.70666447636444

 29%|
| 43/150 [16:14<40:15, 22.58s/it]

Epoch: 42, Average training Loss: 199.83057990141222

 29%|
| 44/150 [16:36<39:45, 22.51s/it]

Epoch: 43, Average training Loss: 212.1995899644743

 30%|
| 45/150 [16:58<39:18, 22.46s/it]

Epoch: 44, Average training Loss: 209.5171423281336
Epoch: 45, Average training Loss: 204.31551056575609

 31%|
| 46/150 [17:22<39:28, 22.78s/it]

Epoch: 45, Average validation Loss: 12655.05931640625

 31%|
| 47/150 [17:44<38:53, 22.65s/it]

Epoch: 46, Average training Loss: 200.1678109163886

 32%|
| 48/150 [18:07<38:21, 22.56s/it]

Epoch: 47, Average training Loss: 207.61689289878618

 33%|
| 49/150 [18:29<37:52, 22.50s/it]

Epoch: 48, Average training Loss: 201.6052372125329

 33%|
| 50/150 [18:51<37:26, 22.46s/it]

Epoch: 49, Average training Loss: 208.0330824966013
Epoch: 50, Average training Loss: 197.6804667664893
```

```
 34%|
| 51/150 [19:15<37:35, 22.79s/it]

Epoch: 50, Average validation Loss: 11152.138828125
Saved new best model with validation loss: 11152.138828125

 35%|
| 52/150 [19:37<37:00, 22.66s/it]

Epoch: 51, Average training Loss: 195.91622711634835

 35%|
| 53/150 [20:00<36:29, 22.57s/it]

Epoch: 52, Average training Loss: 233.0387368960408

 36%|
| 54/150 [20:22<35:59, 22.49s/it]

Epoch: 53, Average training Loss: 212.8753385204376

 37%|
| 55/150 [20:44<35:32, 22.45s/it]

Epoch: 54, Average training Loss: 202.63611326129978
Epoch: 55, Average training Loss: 202.86581781936388

 37%|
| 56/150 [21:08<35:38, 22.75s/it]

Epoch: 55, Average validation Loss: 11813.426477864583

 38%|
| 57/150 [21:30<35:03, 22.62s/it]

Epoch: 56, Average training Loss: 195.3697715981679

 39%|
| 58/150 [21:52<34:34, 22.54s/it]

Epoch: 57, Average training Loss: 204.11805844471175

 39%|
| 59/150 [22:15<34:07, 22.50s/it]

Epoch: 58, Average training Loss: 198.8385080390627

 40%|
| 60/150 [22:37<33:42, 22.47s/it]

Epoch: 59, Average training Loss: 190.04727754972308
Epoch: 60, Average training Loss: 197.2855082847827

 41%|
| 61/150 [23:01<33:49, 22.80s/it]

Epoch: 60, Average validation Loss: 10795.025592447917
Saved new best model with validation loss: 10795.025592447917
```

```
 41%|
| 62/150 [23:23<33:14, 22.67s/it]

Epoch: 61, Average training Loss: 194.9624526764529

 42%|
| 63/150 [23:45<32:44, 22.58s/it]

Epoch: 62, Average training Loss: 189.65677333158487

 43%|
| 64/150 [24:08<32:15, 22.51s/it]

Epoch: 63, Average training Loss: 186.74133513711314

 43%|
| 65/150 [24:30<31:49, 22.46s/it]

Epoch: 64, Average training Loss: 200.7240331913595
Epoch: 65, Average training Loss: 193.67370435060755

 44%|
| 66/150 [24:54<31:52, 22.77s/it]

Epoch: 65, Average validation Loss: 10927.897604166666

 45%|
| 67/150 [25:16<31:20, 22.65s/it]

Epoch: 66, Average training Loss: 189.1080205921405

 45%|
| 68/150 [25:38<30:50, 22.57s/it]

Epoch: 67, Average training Loss: 188.53981702695714

 46%|
| 69/150 [26:01<30:23, 22.51s/it]

Epoch: 68, Average training Loss: 180.2252318373596

 47%|
| 70/150 [26:23<29:57, 22.47s/it]

Epoch: 69, Average training Loss: 179.2267280803013
Epoch: 70, Average training Loss: 180.56691584663653

 47%|
| 71/150 [26:47<30:00, 22.79s/it]

Epoch: 70, Average validation Loss: 10579.395553385417
Saved new best model with validation loss: 10579.395553385417

 48%|
| 72/150 [27:09<29:27, 22.66s/it]

Epoch: 71, Average training Loss: 183.38486327549512
```

```
 49%|
| 73/150 [27:31<28:58, 22.58s/it]

Epoch: 72, Average training Loss: 197.64734947212781

 49%|
| 74/150 [27:54<28:31, 22.52s/it]

Epoch: 73, Average training Loss: 185.20205671451748

 50%|
| 75/150 [28:16<28:06, 22.48s/it]

Epoch: 74, Average training Loss: 182.17687089562912
Epoch: 75, Average training Loss: 177.32711926991766

 51%|
| 76/150 [28:40<28:06, 22.79s/it]

Epoch: 75, Average validation Loss: 10998.801393229167

 51%|
| 77/150 [29:02<27:34, 22.66s/it]

Epoch: 76, Average training Loss: 177.41277692673958

 52%|
| 78/150 [29:24<27:05, 22.58s/it]

Epoch: 77, Average training Loss: 172.68697472454906

 53%|
| 79/150 [29:47<26:37, 22.51s/it]

Epoch: 78, Average training Loss: 169.99113326598348

 53%|
| 80/150 [30:09<26:11, 22.45s/it]

Epoch: 79, Average training Loss: 172.98190296826812
Epoch: 80, Average training Loss: 175.94723157217533

 54%|
| 81/150 [30:33<26:10, 22.77s/it]

Epoch: 80, Average validation Loss: 9650.098522135417
Saved new best model with validation loss: 9650.098522135417

 55%|
| 82/150 [30:55<25:39, 22.63s/it]

Epoch: 81, Average training Loss: 173.17847542757636

 55%|
| 83/150 [31:17<25:10, 22.55s/it]

Epoch: 82, Average training Loss: 183.85384830105085
```

```
 56%|
| 84/150 [31:40<24:43, 22.48s/it]

Epoch: 83, Average training Loss: 171.86267566538535

 57%|
| 85/150 [32:02<24:18, 22.44s/it]

Epoch: 84, Average training Loss: 175.10514348174138
Epoch: 85, Average training Loss: 168.54746458928258

 57%|
| 86/150 [32:26<24:17, 22.77s/it]

Epoch: 85, Average validation Loss: 9564.3280859375
Saved new best model with validation loss: 9564.3280859375

 58%|
| 87/150 [32:48<23:46, 22.64s/it]

Epoch: 86, Average training Loss: 207.39528266293485

 59%|
| 88/150 [33:10<23:18, 22.56s/it]

Epoch: 87, Average training Loss: 208.95376379658936

 59%|
| 89/150 [33:33<22:52, 22.50s/it]

Epoch: 88, Average training Loss: 199.54460247990247

 60%|
| 90/150 [33:55<22:27, 22.46s/it]

Epoch: 89, Average training Loss: 193.42557347373776
Epoch: 90, Average training Loss: 183.85544061829367

 61%|
| 91/150 [34:18<22:23, 22.78s/it]

Epoch: 90, Average validation Loss: 9974.208404947916

 61%|
| 92/150 [34:41<21:54, 22.66s/it]

Epoch: 91, Average training Loss: 177.87448022684097

 62%|
| 93/150 [35:03<21:26, 22.57s/it]

Epoch: 92, Average training Loss: 172.3462343247664

63%|
| 94/150 [35:26<21:00, 22.51s/it]

Epoch: 93, Average training Loss: 185.41741769594375
```

```
63%|
| 95/150 [35:48<20:35, 22.46s/it]

Epoch: 94, Average training Loss: 192.19908087938148
Epoch: 95, Average training Loss: 175.04703979060574

 64%|
                                                  | 96/150 [36:11<20:30, 22.78s/it]

Epoch: 95, Average validation Loss: 10067.47525390625

 65%|
                                                  | 97/150 [36:34<20:00, 22.64s/it]

Epoch: 96, Average training Loss: 170.5375293076174

 65%|
                                                  | 98/150 [36:56<19:32, 22.55s/it]

Epoch: 97, Average training Loss: 172.1560898548505

 66%|
                                                  | 99/150 [37:18<19:06, 22.48s/it]

Epoch: 98, Average training Loss: 207.00443846965666

 67%|
                                                  | 100/150 [37:41<18:42, 22.45s/it]

Epoch: 99, Average training Loss: 188.44016343859573
Epoch: 100, Average training Loss: 172.6855187049279

 67%|
                                                  | 100/150 [38:04<19:02, 22.85s/it]

Epoch: 100, Average validation Loss: 9684.602115885416
```

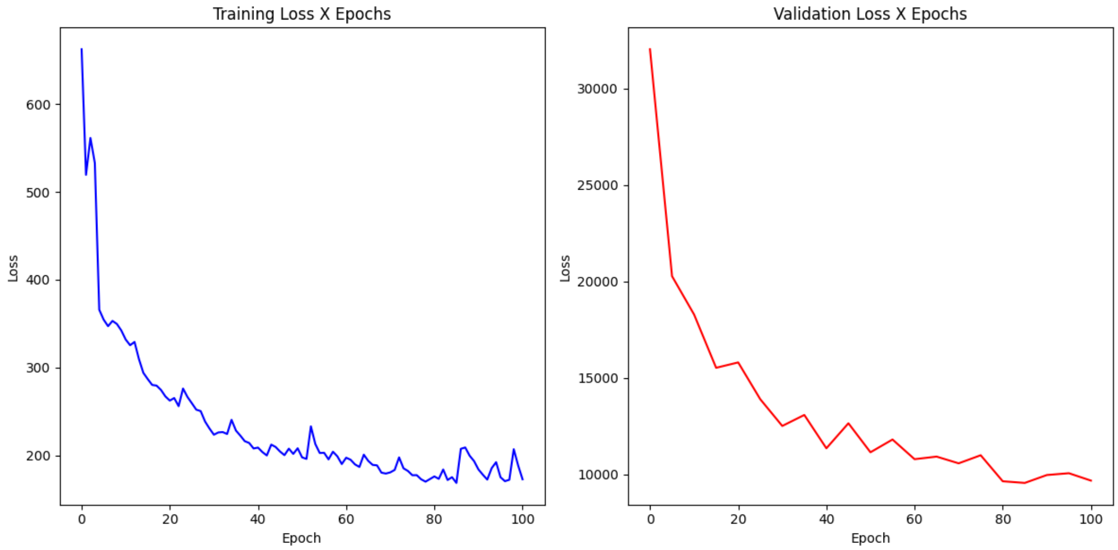
[11]:
```
# Plotting training and validation loss curves
fig, axs = plt.subplots(1, 2, figsize=(12, 6))  # 1 row, 2 columns of graphs

axs[0].plot(train_loss_series, label='Training Loss', color='blue')
axs[0].set_title('Training Loss X Epochs')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('Loss')

val_epochs = [i*5 for i in range(len(val_loss_series))]
axs[1].plot(val_epochs, val_loss_series,  label='Validation Loss', color='red')
axs[1].set_title('Validation Loss X Epochs')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('Loss')

plt.tight_layout()
```

```
plt.show()
```



[12]:
```
# Animating original sequence to be reconstructed
animation = animate_segments(10000, 1, "Visualizing Original Sequence")
HTML(animation.to_jshtml())
```

```
/tmp/ipykernel_15487/1008845002.py:50: MatplotlibDeprecationWarning: Setting
data with a non sequence type is deprecated since 3.7 and will be remove two
minor releases later
  point.set_data(new_x, new_y)
```

[12]: <IPython.core.display.HTML object>

[13]:
```
# Animating reconstructed sequence based on the above original sequence
best_model_path = 'best_model.pth'
model.load_state_dict(torch.load(best_model_path))
model.eval()
with torch.no_grad():
    sequence_prime, _, _ = model(dance_sequences[10000].unsqueeze(0))

sequence_prime = np.array(sequence_prime.squeeze(0).cpu()).reshape(seq_len,␣
  ↪n_joints, 3)
sequence_prime = np.swapaxes(sequence_prime, 0, 1)

animation = animate_segments(None, None, "Visualizing Reconstructed Sequence",␣
  ↪"r", sequence_prime)
HTML(animation.to_jshtml())
```

```
/tmp/ipykernel_15487/1008845002.py:50: MatplotlibDeprecationWarning: Setting
```

```
data with a non sequence type is deprecated since 3.7 and will be remove two
minor releases later
  point.set_data(new_x, new_y)
```

[13]: <IPython.core.display.HTML object>

[14]:
```python
# Generating and animating new sequence - either sampling from a normal␣
↪gaussian distribution or sampling with slightly modified standard
# deviations to make sequences a bit more crazy and test limits.
torch.manual_seed(0)
sample = torch.randn(1, lat_dim).to(device)
#sample = torch.normal(mean=torch.rand(lat_dim), std=(torch.rand(lat_dim))*1.2).
↪unsqueeze(0).to(device)

model.load_state_dict(torch.load(best_model_path))
model.eval()
with torch.no_grad():
    generated_sequence = model.decoder(sample, seq_len)

generated_sequence = np.array(generated_sequence.squeeze(0).cpu()).
↪reshape(seq_len, n_joints, 3)
generated_sequence = np.swapaxes(generated_sequence, 0, 1)

animation = animate_segments(None, None, "Visualizing Generated Sequence", "r",␣
↪generated_sequence)
HTML(animation.to_jshtml())
```

```
/tmp/ipykernel_15487/1008845002.py:50: MatplotlibDeprecationWarning: Setting
data with a non sequence type is deprecated since 3.7 and will be remove two
minor releases later
  point.set_data(new_x, new_y)
```

[14]: <IPython.core.display.HTML object>

4 Why this Project?

Growing up in the northeast of Brazil, art and communication were central to my life. Surrounded by the richness of Brazilian music and dance from a young age, I quickly connected to the arts. While I'm not as skilled as many dancers, I strongly believe in the power of dance to bring warmth to any environment and I always engage in it with joy. My talents, though, are more related to communication, making me a natural-born chatterbox, always wanting to learn from others, and share parts of my own journey. I think the mixture of my cultural background along with my academic and professional trajectory is exactly what connects me to this project and truly makes me want to be a part of it.

Now talking about approaches to the project, I would first focus on building the dataset. I would use the same methods employed in the provided paper to preprocess the dance sequences, but now separating the joints of the two dancers into different groups and encoding the interaction between nodes from the two. Focusing on the proposed methods, we could:

- Draw and edge between joints from different dancers that have some common properties. Very close nodes with either same or opposite velocity vectors, and nodes in symmetric positions are examples of what these properties could be. This form of encoding fits perfectly a GNN and could be used to help the model understand the sequences while capturing the relationship between dancers.
- Use the same approach for pairs of nodes computation as before, but rather than encoding pairs with an edge, encoding them into a dance sequence by a special connection token. These tokens could be used by a transformer model to capture the relationship between nodes over time, understanding a sequence as combination of interactive joint pairs.