LONDON'S GLOBAL UNIVERSITY

Generating Novel Motions Based on a Single Training Example

DXZB7[1]

BSc Computer Science

Dr. Yuzuko Nakamura

Submission date: 23 April 2024

[1]**Disclaimer:** This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text.The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This research addresses the challenge of motion generation in computer graphics, traditionally reliant on costly motion capture systems and extensive datasets predominantly featuring human subjects. A novel approach is proposed, utilizing a diffusion model capable of generating diverse, variable-length motions from a single training example. This model is particularly advantageous for generating motions for arbitrary skeleton structures, including animals and imaginary creatures, which significantly deviates from traditional methods that require large, specific datasets. By leveraging the principles of noising and denoising inherent to diffusion processes, the model, employing a CovNext-based architecture and time-step embedding layers, effectively learns to reproduce and diversify motions from minimal input. This approach not only accelerates the training and generation processes but also enhances adaptability to various applications in animation and video games, reducing dependency on manual rigging. The performance of the model is rigorously evaluated through innovative metrics such as reproduction and diversity scores, demonstrating its efficacy in generating coherent and varied motions. This study contributes to the ongoing shift from generative adversarial networks to diffusion models, highlighting their potential in motion generation with limited data input.

Contents

Chapter 1

Introduction

Motion generation is a long-standing problem in computer graphics, with several applications in video games, film-making and robotics. The most common way of acquiring new motion data involves expensive motion capture systems, which require trained experts to operate rigs and perform post-processing on the collected data. Recent advances in the field have focused on neural network based approaches to rectify these issues but come with problems of their own. They often require massive amounts of data to train on and are limited to motions of humans since they are the main subject of motion capture datasets.

This project aims to address these issues by creating a model that can generate diverse motions of a variable length given a single training example. The use of a single training example means that the model loses out on the sheer variety offered by more data-driven models, but offers advantages when it comes to training speed, generation speed and adaptability to arbitrary skeleton structures. For example, the model can be fed a single motion with an arbitrary skeletal structure, such as an animal or imaginary creature, and learn to generate new motions with the same skeletal topology.

This approach could have several applications in the computer animation domain. For example, an animator could create a single animation for a complicated skeleton, which would then be fed into the proposed diffusion model to generate novel motions. These motions could be used for tasks like crowd animation, where multiple identical skeletons would be able display distinct motions even though they were only trained on a single input motion.

The concept of learning from a single training example has been previously explored in domains such as image generation and motion generation using generative ad-

versarial networks [Li+22]. Recently however, they have been shown to perform worse compared to newer diffusion-based models in regards to generative tasks. Diffusion models work by sequentially noising the input data according to some fixed schedule and then using a neural network to denoise from pure noise to generate a result [HJA20]. These models are already replacing GANs in image generation tasks, evident by rise of projects like DALL-E [Ram+21], and their suitability for motion generation has been explored in the last couple of years. However, these projects still rely on large datasets and are limited to human motions. Thus, the novelty of this project lies in investigating whether diffusion models are suitable for single-input based motion generation, a task previously dominated by generative adversarial networks (GANs).

Chapter 2

Related work

2.1 Early Work on Motion Synthesis

2.1.1 Motion texture: a two-level statistical model for character motion synthesis

This paper [LWS02] introduces a method for creating complex human-figure motions by using "motion textons." Each one is represented by a Linear Dynamic System (LDS) that models the local dynamics of motion segments. The global dynamics, which describe how these segments change and flow into each other throughout a sequence, are handled by a transition matrix. This matrix outlines the likelihood of moving from one texton to another, allowing for both the repetition and variability seen in complex motions.

The method generates new motion sequences by first learning these textons and their distributions from original data. It then uses this information to create new motions that statistically resemble the originals. The authors use a maximum likelihood algorithm to identify and understand the motion textons and their relationships within captured dance motions.

This approach is used for motion synthesis, where new animations can be automatically created or existing ones can be interactively modified to make new dance routines. This involves defining motion textons with LDS and using a transition matrix to choreograph new sequences from what has been learned.

2.1.2 Motion graphs

This paper [KGP02] introduces a new way to create realistic and controllable motion using a collection of motion capture data. The main contribution of this study is the creation of a directed graph, called a "motion graph," which connects different pieces of motion capture clips. This graph combines original motion data and automatically generated transitions, allowing for the creation of new motion sequences by following paths through the graph.

Building a motion graph involves cutting motion capture data into clips and then finding possible points where these clips can be smoothly joined. This process includes checking for transitions by comparing the similarity of motion poses, creating seamless connections using linear blending techniques, and trimming the graph to avoid any dead ends. The selection of transition points pays special attention to the dynamic qualities of human movement, ensuring that the transitions are smooth and maintain the realism of the original data.

A crucial feature of the motion graph technique is its ability to represent a broad range of movements and transitions within a single framework. By showing transitions as edges in the graph, this method provides a structured way to explore the motion database, creating continuous motion sequences that can be adjusted and controlled based on broad user instructions.

2.2 Deep Learning Methods

Recent developments in motion synthesis have moved away from old methods that rely heavily on large datasets. Instead, modern techniques focus on learning from smaller data sources like a single image, motion sequence, or video. This change helps overcome the challenge of producing realistic and varied animations without needing vast amounts of motion capture data. This is especially useful in areas where collecting large-scale data is difficult or impractical such as for arbitrary skeletons.

2.2.1 GANimator: neural motion synthesis from a single sequence

This paper [Li+22] uses several layers of GANs to build motion from random noise. These layers are structured to handle different frame rates, giving precise control over the motion's appearance at various levels of detail. Unlike traditional methods that need large sets of motion data, the model only requires one motion sequence

to learn.

The GANs include both generators and discriminators, which are trained in stages to capture motion from coarse to fine details. Starting with random noise, the first level generator produces a basic motion sequence. Following generators improve this motion by adding upscaled data from the earlier stage and new noise, leading to a more detailed motion sequence.

In the model, motion is encoded through the movement of the root joint and the rotation of other joints over time. It also uses binary labels for foot contacts to prevent common errors like foot sliding. This approach uses 6D rotation features, which have been proven effective in a previous study [Zho+20].

The training of the model involves a loss function that includes adversarial, reconstruction, and contact consistency loss. The contact consistency loss ensures that the foot contact labels match the actual foot positions, addressing foot sliding issues. This method performs better than others like Motion texture in producing varied motion sequences that closely resemble the original motion.

2.2.2 Diffusion Models

Recent advancements have led to the development of diffusion models, which offer a robust alternative to traditional GANs for generating content. These models operate by introducing and subsequently removing noise, allowing them to produce high-quality outputs from complex data. Their use in single-instance training and motion synthesis is a particularly new area, and combining these two is the focus of this paper.

2.2.3 SinFusion: Training Diffusion Models on a Single Image or Video

In the paper [NHI23], the authors explore how diffusion models, which are typically used to generate high-quality images and videos from large datasets, can be adapted to work with just one input image or video. This method can learn from a single image or video and use this learning to perform various manipulation tasks.

The paper modifies existing diffusion model structures to train with single images or videos by working with large random crops of the input. It also alleviates the issue of overfitting to the single training input by using the recently studied CovNext model [Liu+22], which shrinks the overall receptive field [ANS19] of the model. This allows it to create varied yet consistent outputs that stay true to the original style

and dynamics.

The authors also modify the typical noise prediction approach of diffusion models. Instead of predicting the noise added at each step, it directly predicts the clean image from the noisy input during training. This adjustment is shown to improve both the quality of results and training efficiency when dealing with the less complex data distribution of a single input. The training utilizes a mean squared error loss function to minimize the difference between the actual clean image and the model's prediction, further tailored to better suit single-input scenarios.

To evaluate the performance of generated results, the paper also introduces new metrics such as nearest neighbor field diversity. This metric measures the diversity of the generated samples by examining the spatial and temporal uniqueness of the patches, penalizing mere translations or repetitions found in simpler models.

The model's ability to learn from minimal data is a notable advancement in the field of diffusion models and will be useful for designing and evaluating a diffusion model that uses single-instance training on motion data for motion synthesis.

2.2.4 Human Motion Diffusion Model

This paper [Tev+22] introduces a new approach to human motion generation using a diffusion-based generative model. The model leverages the diffusion process to gradually transform random noise into a structured sequence of human joint movements over time.

A key innovation of the model is its use of a transformer-based architecture [Vas+23] rather than the commonly used U-net structure [RFB15]. This choice reflects the temporal and non-spatial nature of motion data, optimizing the model for sequences of data rather than static images. By predicting the sample directly at each diffusion step rather than predicting the noise, the model can apply geometric losses like foot-contact loss on the predicted motions. This directly improves the physical accuracy and realism of the outputs.

Furthermore, the model's flexibility in handling various conditioning modes, such as text-to-motion and action-to-motion, allows it to be versatile across different motion generation scenarios. It uses a classifier-free training approach [HS22], which helps balance between the diversity of generated motions and the fidelity to the given input conditions. Despite some challenges with increased inference times, the benefits in terms of motion quality and versatility make it a promising model for both research and practical applications, and the results are an encouraging sign for the use of diffusion models in the generation of arbitrary skeleton motions.

Chapter 3

Preliminaries

3.1 Motion representation

While industry standards for motion data representation are varied, the Biovision Hierarchy (BVH) file format is the format of choice for most open motion datasets, and thus will be the format expected by the model. BVH files are a common format for storing motion capture data, and they represent motion as a hierarchy of joints and their rotations over time. Each joint also has a fixed offset from its parent in XYZ coordinates, which specifies the fixed bone-length between the two joints.

Each frame in a BVH file contains the rotations of each joint in the coordinate space relative to their parent in the kinematic chain, which can be used to encode the animation of an arbitrary skeleton in a 3D environment. In the datasets chosen for the model, the rotations are specified by Euler angles.

These features need to be encoded into a tensor representation that can be passed into the diffusion model to generate new motions. However, some of these features can be omitted from the model. For example, the hierarchy of joints can be implicitly represented in the tensor by making the nth row represent the nth joint encountered during the BVH parsing step. Using similar optimizations and the work of papers such as GANimator [Li+22], the tensor representation of the motion sequence solely focuses on the dynamic features that change with each frame, namely the rotations of the joints relative to their parent. The constant features such as offsets from parents are fixed and can be reconstructed at the end based on the input BVH file.

Let T represent the total number of frames in the animation, F represent the total number of features required to specify joint rotations, and J represent the total

number of joints. There are several possible representations of the same rotation, including Euler angles ($F = 3$), quaternions ($F = 4$), and the 6D [Zho+20] rotation features ($F = 6$). The model is able to operate on all 3 of these representations and can convert between them regardless of the initial input. However, the 6D representation is likely to perform the best since it avoids issues like the gimbal locks and double-cover of the $SO(3)$ rotation group which interfere with model convergence.

A frame is represented as $f \in \mathbb{R}^{J \times F}$, with each row specifying the rotation features of a joint in canonical order, dependant on the rotation representation used. The order of the joints in the matrix mirrors the order of joints encountered during preorder traversal of the BVH file hierarchy. Overall, the tensor representing the motion sequence across all frames is:

$$M \in \mathbb{R}^{J \times F \times T} \tag{3.1}$$

3.2 Diffusion Models

As discussed briefly, diffusion models are a new type of generative model that can take pure noise and turn them into samples from a learned distribution. They do this by repeatedly removing a small amount of Gaussian noise until the desired sample is achieved. Since the model heavily relies on this process for motion generation, a brief overview of the process is detailed here.

To train a diffusion model on an input motion sequence x_0, a small amount of Gaussian noise is added to it at each step in a Markov noising process, leading to a noisy motion sequence x_t after t iterations. The final noisy sample x_T is the result after T steps of this process. The distribution of x_t depends on the previous motion sequence x_{t-1} as follows:

$$q(x_t|x_{t-1}) = \mathcal{N}(\sqrt{\alpha_t}x_{t-1}, (1 - \alpha_t)I), \tag{3.2}$$

where $\alpha_t \in (0, 1)$ is a constant hyperparameter that controls the mean and variance of the generated sample and monotonically approaches 0 as t approaches T according to a fixed schedule. When α_t becomes very small, the distribution of the sample x_t approximates a standard normal distribution, with 0 mean and identity covariance. Thus, as the noising process approach time-step T, α_t gets smaller and the sample generated approaches a standard normal distribution [HJA20].

Additionally, while 3.2 describes the distribution that needs to be achieved, it does not explain how to derive it from the previous time-step. To achieve the required sample, note that noise sampled from a standard normal distribution can be used to generate a sample z from a normal distribution with arbitrary mean μ and variance σ^2 as follows [Ben]:

$$z \sim \mathcal{N}(\mu, \sigma^2) \implies z = \mu + \epsilon\sigma, \tag{3.3}$$

where $\epsilon \sim \mathcal{N}(0, 1)$. Thus, to generate the required sample x_t from x_{t-1} according to 3.2, some noise ϵ is sampled from a standard normal distribution and used as follows:

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1 - \alpha_t}\epsilon, \tag{3.4}$$

where $\epsilon \sim \mathcal{N}(0, I)$. As a final note, while 3.4 describes how x_t is derived using the sample from the previous time-step x_{t-1}, the fact that α_t are fixed hyperparamaters means that the following cumulative product can be calculated:

$$\bar{\alpha_t} = \prod_{n=1}^{t} \alpha_n \tag{3.5}$$

Since the Markov noising process depends solely on the α_t, this cumulative product can be used to derive a sample at any time-step t directly from the input motion sequence x_0 as follows [ND21]:

$$x_t = \sqrt{\bar{\alpha_t}} x_0 + \sqrt{1 - \bar{\alpha_t}}\epsilon, \tag{3.6}$$

where $\epsilon \sim \mathcal{N}(0, I)$.

Then, a neural network P is used to gradually denoise from x_T conditioned on the time-step of the sample. While most diffusion models choose to predict only the noise difference between samples for simplicity, works such as SinFusion [NHI23] show that predicting the input $\hat{x_0}$ directly works better for single-instance training purposes. Thus, the neural network P is used to predict the input motion:

$$\hat{x_0} = P(x_t, t), \tag{3.7}$$

for arbitrary time-steps t, where x_t is derived using 3.4

The loss function used to train the neural network is simply the L_2 loss between

the actual input and predicted input for each time-step $t \in (1, T)$, expressed in expectation notation as:

$$E_{t\sim[1,T]}[||x_0 - P(x_t, t)||_2^2] \tag{3.8}$$

Once the model is trained, it has learnt to denoise from the final sample x_T to an approximation of the training input $\hat{x_0}$. Furthermore, due to the way the Markov noising process is defined in 3.2, the final sample x_T is also an approximation of a standard normal distribution $\mathcal{N}(0, I)$. Thus, the neural network P can be instead fed a sample from a standard normal distribution and will generate a novel motion that is similar to the training motion input. For better results, the procedure described by Tevet et al. [Tev+22] is followed to iteratively sample rather than doing it in one step:

Algorithm 1 Generating novel motions using trained neural network P

$x_T \sim \mathcal{N}(0, I)$
for $t = T, ..., 1$ **do**
    $\hat{x_0} = P(x_t, t)$
    $\epsilon \sim \mathcal{N}(0, I)$
    $x_{t-1} = \sqrt{\bar{\alpha_{t-1}}}\hat{x_0} + \sqrt{1 - \bar{\alpha_{t-1}}}\epsilon$
end for
return $\hat{x_0}$

Chapter 4

Implementation

The diffusion process outlined above discusses how to generate noised images at specific time-steps and specifies the loss function for the denoising neural network P. There are still some details to discuss, namely the variance schedule and the architecture of the neural network P.

4.1 Variance schedule

As explained in equation 3.2, the variance of the noised sample x_t at time-step t is controlled by a hyperparameter $\alpha_t \in (0, 1)$ that monotonically approaches 0 as t approaches T. The function used to model the variance based on the time-step t is called the variance schedule. There are many different variance schedules including linear, quadratic and cosine functions. The cosine schedule is chosen for the model since it maintains a more detail across later time-steps, giving the model more information to denoise from. The equation for the schedule is derived by simply taking the cosine of the current timestep t divided by the final timestep T, and scaling the x-axis by $\frac{\pi}{2}$ to constrain the output of α_t to $(0, 1)$ as required [ND21]:

$$\alpha_t = \cos(\frac{\pi t}{2T}) \tag{4.1}$$

In the image below by Nichol et Al [ND21], the top half shows the progressively noised images with a linear schedule, while the bottom half follows a cosine schedule:

As noted, the cosine schedule leads to a more gradual noising process.

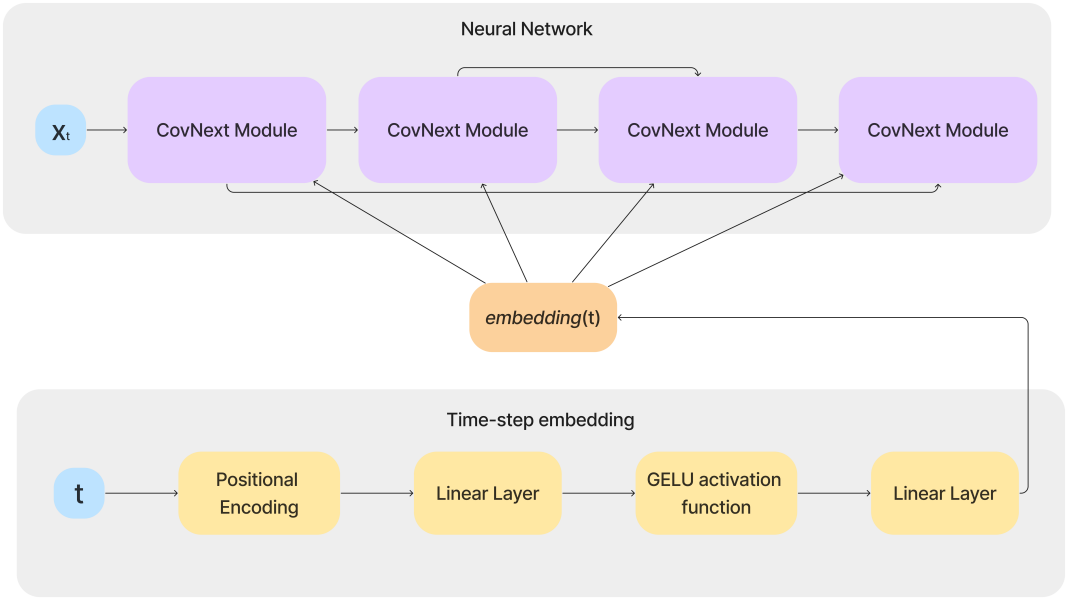
4.2 Network Model

There are several different choices for the network architecture of P, ranging from UNets [HJA20] to transformers [Tev+22]. Unfortunately, these models are unsuitable for most single-instance training due to their large receptive fields and global attention layers. Dealing with only a single training example can lead to overfitting to the training input, which can impact the diversity of generated results. However, recent work has shown that it is possible to replicate the performance of transformer-based networks using a more lightweight convolutional neural network architecture called CovNext [Liu+22].

4.2.1 Network architecture of P

These CovNext modules form the backbone of the neural network P, with residual connections to mitigate the vanishing gradient problem. The number of CovNext layers is a hyperparameter that can be adjusted depending on the use case. Increasing the depth of the network this way leads to more coherent motions at the cost of output diversity. The model uses a CovNext backbone depth of 16 after some empirical testing since it strikes a good balance between both goals.

The network also includes a time-step embedding layer, using a standard positional encoding scheme [NHI23], that is added into the input of each CovNext module. This is key because the time-step in the diffusion process directly controls how much the input motion is noised, and thus the neural network needs access to this time-step information in order to denoise accordingly. Thus, the overall model for the denoising neural network P looks like the following:



4.2.2 Dataset

Since the model is trained on a single training example, the exact dataset used doesn't matter too much, and motions from different datasets could be feasibly combined. However, the model solely uses the Truebones Zoo BVH dataset [Tru20] since all the motions within adhere to a fixed format, making the parser design simpler and less prone to error. The dataset contains motions of many different animals, allowing the model's performance to be tested on a variety of skeletons

To parse, a simple top-down parser is used, and the properties of the BVH file are loaded into an intermediate class that specifies the joints, joint parents, frame information, rotation information for each frame and joint, and other related information. All the relevant motion data is transformed into the input tensor according to 3.1. Namely, a tensor is constructed by iterating over all joints in the order they were encountered and populating the rows with the appropriate rotation features using the 6D rotation representation. Then, the resulting 2D matrices are concatenated over all the frames in the motion, yielding a 3D tensor. The model also implements a coordinate change module to freely change between Euler angle, quaternion and 6D representations of rotations during the tensor construction step. This allows comparisons on how different rotation representations can affect model convergence and performance.

4.2.3 Training

To train the model, a time-step t is uniformly sampled between 1 and T. This is used to generate a noised version of the initial motion x_t according to 3.6. This is implemented by a Diffusion class in the code that implements the diffusion algorithms to noise a given tensor x_0 to a randomly generated time-step t, yielding the noised version x_t. The CovNext-based denoising network P receives x_t and t, and gives a prediction of the initial motion $\hat{x_0}$. Then, the L_2 loss defined in equation 3.8 is used to take a gradient descent step on the weights of P. We repeat this process, randomly sampling a new time-step each time until the weights of the neural network P converge:

Algorithm 2 Training neural network P

1: **repeat**
2: $t \sim \mathcal{U}(1, T)$
3: $\epsilon \sim \mathcal{N}(0, I)$
4: $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$
5: $\hat{x_0} = P(x_t, t)$
6: Gradient Descent on: $\nabla ||x_0 - \hat{x_0}||^2$
7: **until** weights of P converged

The model is implemented in the PyTorch library and trained on an Nvidia 3060 12GB graphics card. In the model, the number of diffusion steps T is set to 1000, using a cosine scheduler for the variance as previously explained. The model is trained on a batch size of 64, using the Adam optimizer with the default learning rate of 0.001.

Since only a single training example is used, each element of the batch is the same initial input but undergoes an independent noising process, leading to more robust model convergence. Finally, to avoid the training process taking too long, the number of iterations of algorithm 2 is capped at 50,000 iterations, which empirically leads to acceptably small losses at the end.

Chapter 5

Evaluation

5.1 Evaluation Metrics

The fields of single-instance training and motion generation are still relatively new and don't have standard evaluation metrics such as in the image generation domain. However, there are broadly two goals of the model, namely, generating motions that are coherent and generating motions that are diverse.

These goals are somewhat contradictory since coherence involves evaluating how plausible the generated motion is given the input motion, while diversity involves making the outputs as different from the training input as possible. This trade-off between metrics is a common problem in many ML papers and is usually solved by calculating the harmonic mean of the all metrics. For example, the popular F1 score used in classification task is a specific type of harmonic mean [Hic+22]. Generalizing this idea, the model is evaluated based on certain metrics and the harmonic mean score of these metrics is calculated as follows:

$$HarmonicMeanScore = \frac{M}{\sum_{n=1}^{M} \frac{1}{s_n}} \tag{5.1}$$

,

where M is the total number of metrics and s_n is the score of the nth metric.

The metrics used to evaluate the model are derived from similar works in the field [Li+22] and aim to quantitatively measure the two goals of the model:

5.1.1 Reproduction Score

To evaluate coherence of generated results, it is useful to measure how much of the input motion is reproduced by the generated motion. The more of the input is reproduced by the generated result, the more coherent it is as it more closely follows the intended data distribution. However, this is difficult to measure in the realm of animation, where the data not only varies spatially in terms of the joint positions but also varies temporally across frames, where the number of all possible windows across frames is extremely large.

Using the convention from other papers dealing with motion generation, a fixed length s is chosen for the the length of the frame windows to make the problem more tractable [Li+22]. To evaluate, a length of $s = 24$ is chosen, which corresponds to 1 second of continuous animation in the Truebones Zoo dataset, since it has a frame time of 24 fps. Using this, all possible 24-frame windows in our inputs can be computed. For example, for a motion sequence M with a length of l frames, the set of all possible 24-frame windows $\mathcal{W}(M)$ is:

$$\mathcal{W}(M) = \{M_1 : M_{24}, M_2 : M_{25}, \ldots, M_{l-22} : M_{l-1}, M_{l-23} : M_l\}, \tag{5.2}$$

where M_i specifies the ith frame in the motion sequence M, and $M_i : M_j$ specifies the collection of frames between the ith and jth frame of M inclusive.

For the model, the set of all possible 24-frame windows of the initial motion sequence $\mathcal{W}(x_0)$ and the generated output $\mathcal{W}(\hat{x_0})$ is calculated using a simple sliding window algorithm that iterates over the motion sequences linearly. Once these sets are constructed, it is determined what what percentage of all the windows in $\mathcal{W}(x_0)$ are reproduced by the windows in $\mathcal{W}(\hat{x_0})$.

To determine whether a window $w \in \mathcal{W}(x_0)$ is reproduced, its nearest neighbour $w_{nn} \in \mathcal{W}(\hat{x_0})$ is found to calculate if the distance between them is less than a certain pre-determined threshold ϵ. For the model, the threshold is set to $\epsilon = 2$ based on empirical testing.

An important consideration is how the distance between two windows is calculated, which is required for determining the nearest-neighbour, via a simple linear scan, and calculating if the nearest-neighbour is within the threshold. The algorithm used to calculate distances between windows is simply based on the Euclidean distance between joint positions across all frames:

Algorithm 3 Calculating distance between windows a and b

F = number of frames in a
J = number of joints in skeleton
sum = 0
for $f = 1, \ldots, F$ **do**
    a_f = fth frame of a
    b_f = fth frame of b
    for $j = 1, \ldots, J$ **do**
        a_{fj} = 3D coordinate of jth joint in a_f
        b_{fj} = 3D coordinate of jth joint in b_f
        sum += $||a_{fj} - b_{fj}||^2$
    end for
end for
return (sum / (F * J))

Finally, the percentage of windows that are reproduced in the set $\mathcal{W}(x_0)$ is calculated to give a reproduction score, where a higher score corresponds to more coherent results.

5.1.2 Diversity score

To evaluate diversity, it is key to measure how different the generated motions are to the input training motion. As mentioned in chapter 5.1.1, this is difficult for data that operates on both a temporal and spatial level, and so the same strategy of slicing the motions into 24-frame windows is used. Once the set of windows $\mathcal{W}(x_0)$ and $\mathcal{W}(\hat{x_0})$ is determined, the following metric is defined for evaluating diversity. For each window in the generated motion $\hat{w} \in \mathcal{W}(\hat{x_0})$, its nearest neighbour in the training motion $w_{nn} \in \mathcal{W}(x_0)$ is determined using the distance metric defined in algorithm 3 via a linear scan process. Then, the following formula is used to find the average distance between windows in the generated motion and its nearest neighbour in the training motion for all windows in the set:

$$\frac{\Sigma_{\hat{w} \in \mathcal{W}(\hat{x_0})} distance(\hat{w}, NearestNeighbour(\hat{w}))}{|\mathcal{W}(\hat{x_0})|} \tag{5.3}$$

If the average distance is high, it means that the windows in the generated motions are quite different compared to windows in the training motion, which indicates a high diversity in the output. If the average distance is low, it means that the generated motion closely matches some windows in the training motion, which means

it either overfits the training motion or it collapses to a static pose in the training motion. Thus, the result of equation 5.3 is used as the diversity score, where a higher score corresponds to more diverse results.

5.2 Evaluation Results

To evaluate our results, both qualitative and quantitative findings are presented, with additional ablation studies to evaluate the model architecture choices.

5.2.1 Qualitative Results

For the qualitative evaluation, the model is trained on a motion of a spider attacking. The model is fully trained once and used to generate 5 motion samples of equal frame length as the training input. Then, the generated BVH file is imported into Blender [Ble24] to render a 3D model of the motion from a fixed perspective. The number of frames in the animations is too high to present properly, so the figures below are limited to the first 3 and last 3 frames of the entire motion sequence:

Figure 5.1: Training motion sequence

Figure 5.2: Generated motion sequence 1

Figure 5.3: Generated motion sequence 2

Figure 5.4: Generated motion sequence 3

Figure 5.5: Generated motion sequence 4

Figure 5.6: Generated motion sequence 5

From the figures above, it can be seen that generated motions are all distinct from each other but still follow the same general motions as the initial training input. Furthermore, the transitions between poses are coherent and do not exhibit jittering or unnatural cuts, leading to a more plausible motion.

Another interesting thing to note is that the first 3 frames of all the motions are quite similar to each other, while the last 3 frames differ quite significantly as the random changes to the rotations and positions accumulate over time. This behaviour is ideal for the model, since the results are initially quite coherent and similar to the input motion but naturally transition into diverse motions by the end of the sequence.

Furthermore, unlike the majority of motion generation models, the model is able to work on arbitrary non-humanoid skeletons such as spiders. The wide applicability of the model can be seen in the next example, where it is instead trained on a motion of a centipede. The model is able to generate diverse motions for this skeleton as well, but the total number of frames for each generated motion is also varied. This is particularly useful for crowd animation, where motions with the same frequencies would seem unnaturally in sync.

By varying the motion lengths, the motions look more random and natural, while still following the general theme of the motion they were trained on. Furthermore, for crowd animation, it is undesirable for the initial poses to match each other too closely and the depth of the CovNext-based denoising network is reduced by half to decrease the generated motion's coherence to the initial input:

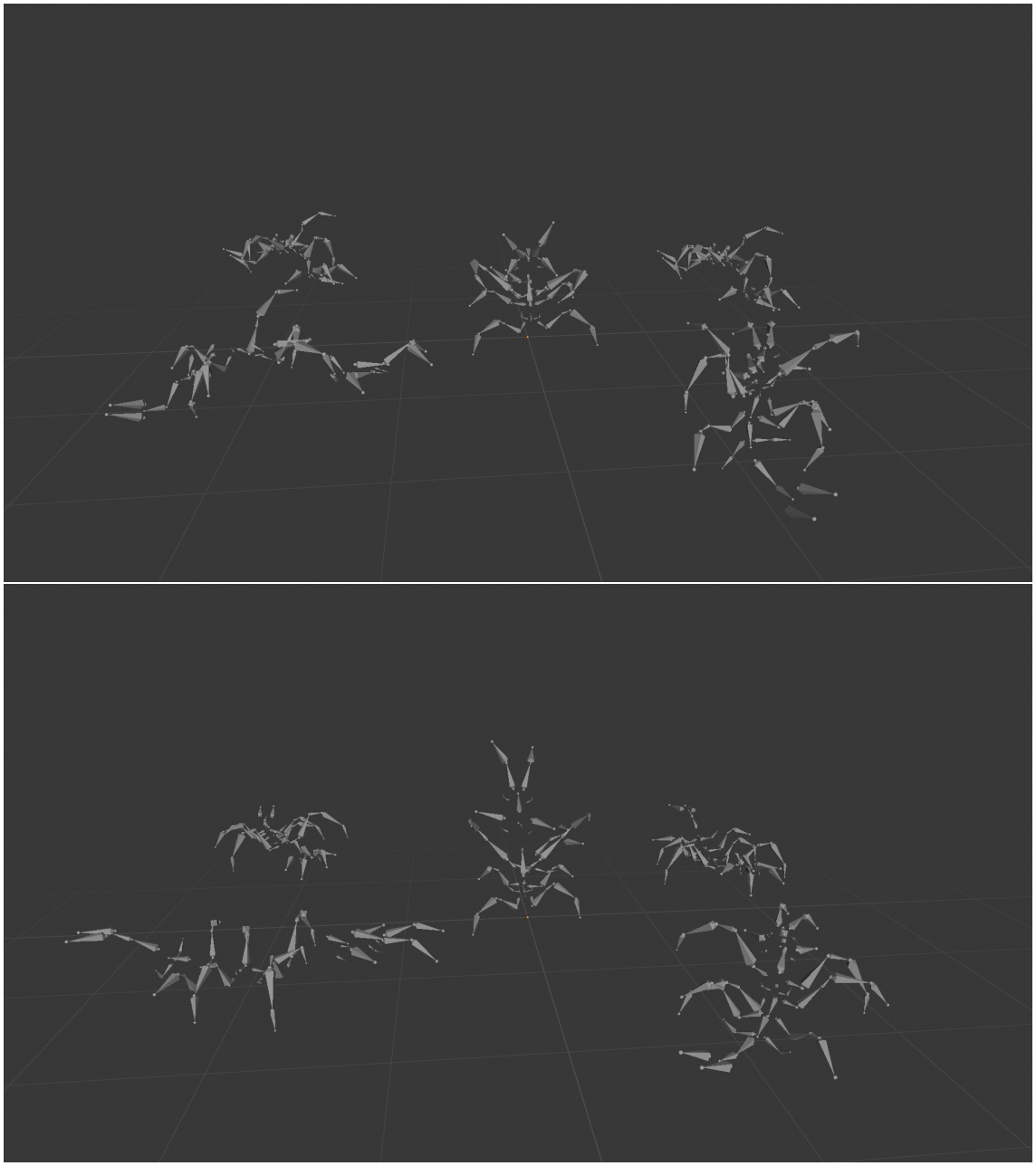


Figure 5.7: Crowd animation from single training example

The motions are all generated from the same training example, and imported into Blender with random root positions and rotations, yielding a convincing crowd animation. The first image displays the first frame of the animation, while the second image corresponds to the last frame of the animation. Note how all the poses are distinct from each other in the first and last frames, owing to the low denoising network depth.

5.2.2 Quantitative Results

For the quantitative results, the metrics defined in section 5.1 are used. To calculate a more representative score, the model is trained separately on 5 distinct training motions with different skeletons, using the model choices and hyperparameters discussed throughout chapter 4.

For each training motion, 5 motions from the trained model are generated. Then, the arithmetic mean of the reproduction and diversity scores of the motions are calculated. Finally, the reproduction and diversity scores across all training motions are averaged to yield a representative result of how well the model scores on a range of different skeletons.

To balance the metrics of reproduction score and diversity score, the harmonic mean score defined in equation 5.1 is used to calculate how well the model balances coherence and diversity. A high harmonic mean score indicates a good balance of both goals. The result on the model as defined is as follows:

Training Motion	Reproduction Score	Diversity Score	Harmonic Mean Score
1	88.4	1.06	2.09
2	90.2	1.03	2.03
3	89.7	1.09	2.15
4	87.9	1.11	2.19
5	90.5	1.05	2.07
Average	**89.34**	**1.07**	**2.11**

The high reproduction score indicate that the model is very coherent and learns the themes of the training motion well. However, it also has a positive diversity score and harmonic mean score, indicating that it does not overfit to the training motion or collapse to a static pose and balances the two goals of the model well.

5.2.3 Ablation study

To justify the model choices made in chapter 4, an ablation study is performed to measuring the metrics above after making certain changes to the model. Namely, these changes include changing the variance schedule to use a linear schedule, changing the rotation representations to quaternions and Euler angles, and changing the depth of the CovNext denoising network P. The results of this ablation study are presented below:

Model type	Reproduction Score	Diversity Score	Harmonic Mean Score
Default	89.34	1.07	2.11
Linear schedule	85.09	1.00	1.98
Quaternions	87.63	1.04	2.06
Euler angles	86.28	1.04	2.06
Double CovNext depth	94.06	1.01	2.00

From the table above, it is clear that the default model choices strike the best balance between coherence and diversity, yielding the highest harmonic mean score. An interesting thing to note is that the rotation representations do not make as much of a difference as expected, yielding similar scores across metrics to the default 6D rotation representations. This might be because of the small sample size, and perhaps the change would be more obvious across more types of skeletons and longer motion sequences that give time for rotation errors to accumulate.

The biggest impact to model performance comes by changing the variance schedule from cosine to linear. This makes sense since the linear schedule is proven too be a lot more destructive in the noising process, causing the denoising model to struggle [ND21]. Interestingly, the linear schedule model scores low in both the reproduction score and diversity score, indicating that it collapses to a pose that it not even representative of the training motion. Lastly, the change to CovNext depth is as predicted by several papers, where increased depth leads to a higher receptive field, leading to overfitting to the input at the expense of diversity [Liu+22].

Chapter 6

Conclusions

6.1 Achievements

The project successfully implements a model to generate new motions given only a single training input. This allows the generation of motions of arbitrary skeletons, which is useful in fields such as animation and video games, and can significantly reduce the need for manual animation rigging. The model uses the noising and denoising diffusion processes from the state-of-the-art diffusion model that has seen tremendous success in the image domain. It uses this process to train a neural network P that can denoise from pure noise to generate plausible motions that match the input training motions.

The architecture of the neural network is carefully considered to mitigate issues with single-instance training such as large receptive fields and overfitting. This is done by utilizing a CovNext-based backbone for the network, a strategy that has seen success for other single-instance training tasks. The network also includes time-step embedding layers to be able to conditionally train on noising time-step, and thus learn how to denoise from arbitrary time-steps in the diffusion process.

The results of the model are then evaluated qualitatively and quantitatively, alongside an ablation study to justify model hyperparamaters and architecture choices. However, the task of evaluating motion data is a difficult one due to how it varies both spatially and temporally. Thus, works from similar motion generation papers are used to define bespoke metrics to evaluate model performance, namely, the reproduction score and the diversity score. These scores successfully quantify how well the model achieves its goals of generating diverse and coherent motions, and they give a deeper insight into how changes to the model affect its results.

6.2 Future Work

The promising results achieved by the current model suggest several avenues for further research and development. A potential enhancement is the creation of a conditional version of the model. Specifically, conditioning the model on external inputs, such as joystick movements or other user interactions, could significantly enhance its applicability in interactive scenarios like video games or virtual reality, where user input directly influences motion generation in real-time.

Integrating a foot contact loss function [Tev+22] could also be highly beneficial, especially for datasets that lack explicit contact information. This integration would improve the physical plausibility of the generated motions by ensuring that foot placements are realistic and appropriately grounded, thus reducing floating or sliding artifacts that often detract from visual realism.

Moreover, advancing the evaluation methods used to assess the quality of generated motions by incorporating physics-based metrics is proposed. Utilizing forward kinematics could provide a more accurate measurement of the deviation of generated motion from expected physical behavior.

Bibliography

[KGP02] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. "Motion graphs". In: *ACM Transactions on Graphics* 21.3 (July 1, 2002), pp. 473–482. ISSN: 0730-0301. DOI: 10.1145/566654.566605. URL: https://dl.acm.org/doi/10.1145/566654.566605 (visited on 04/20/2024).

[LWS02] Yan Li, Tianshu Wang, and Heung-Yeung Shum. "Motion texture: a two-level statistical model for character motion synthesis". In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '02. New York, NY, USA: Association for Computing Machinery, July 1, 2002, pp. 465–472. ISBN: 978-1-58113-521-3. DOI: 10.1145/566570.566604. URL: https://doi.org/10.1145/566570.566604 (visited on 04/20/2024).

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. May 18, 2015. DOI: 10.48550/arXiv.1505.04597. arXiv: 1505.04597[cs]. URL: http://arxiv.org/abs/1505.04597 (visited on 04/20/2024).

[ANS19] Andr\&eacute Araujo, Wade Norris, and Jack Sim. "Computing Receptive Fields of Convolutional Neural Networks". In: *Distill* 4.11 (Nov. 4, 2019), e21. ISSN: 2476-0757. DOI: 10.23915/distill.00021. URL: https://distill.pub/2019/computing-receptive-fields (visited on 04/20/2024).

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. *Denoising Diffusion Probabilistic Models*. Dec. 16, 2020. DOI: 10.48550/arXiv.2006.11239. arXiv: 2006.11239[cs,stat]. URL: http://arxiv.org/abs/2006.11239 (visited on 04/20/2024).

[Tru20] Truebones. *FREE TRUEBONES ZOO, Over 75 Animals and Animations*. 2020. URL: https://truebones.gumroad.com/p/free-truebones-zoo-over-75-animals-and-animations (visited on 04/20/2024).

[Zho+20] Yi Zhou et al. *On the Continuity of Rotation Representations in Neural Networks*. June 8, 2020. DOI: 10.48550/arXiv.1812.07035. arXiv: 1812.07035[cs,stat]. URL: http://arxiv.org/abs/1812.07035 (visited on 04/20/2024).

[ND21] Alex Nichol and Prafulla Dhariwal. *Improved Denoising Diffusion Probabilistic Models*. Feb. 18, 2021. arXiv: 2102.09672[cs,stat]. URL: http://arxiv.org/abs/2102.09672 (visited on 04/17/2024).

[Ram+21] Aditya Ramesh et al. *Zero-Shot Text-to-Image Generation*. Feb. 26, 2021. DOI: 10.48550/arXiv.2102.12092. arXiv: 2102.12092[cs]. URL: http://arxiv.org/abs/2102.12092 (visited on 04/20/2024).

[Hic+22] Steven A. Hicks et al. "On evaluation metrics for medical applications of artificial intelligence". In: *Scientific Reports* 12 (Apr. 8, 2022), p. 5979. ISSN: 2045-2322. DOI: 10.1038/s41598-022-09954-8. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8993826/ (visited on 04/20/2024).

[HS22] Jonathan Ho and Tim Salimans. *Classifier-Free Diffusion Guidance*. July 25, 2022. DOI: 10.48550/arXiv.2207.12598. arXiv: 2207.12598[cs]. URL: http://arxiv.org/abs/2207.12598 (visited on 04/20/2024).

[Li+22] Peizhuo Li et al. "GANimator: neural motion synthesis from a single sequence". In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–12. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3528223.3530157. URL: https://dl.acm.org/doi/10.1145/3528223.3530157 (visited on 12/03/2023).

[Liu+22] Zhuang Liu et al. *A ConvNet for the 2020s*. Mar. 2, 2022. DOI: 10.48550/arXiv.2201.03545. arXiv: 2201.03545[cs]. URL: http://arxiv.org/abs/2201.03545 (visited on 04/20/2024).

[Tev+22] Guy Tevet et al. *Human Motion Diffusion Model*. Oct. 3, 2022. arXiv: 2209.14916[cs]. URL: http://arxiv.org/abs/2209.14916 (visited on 01/07/2024).

[NHI23] Yaniv Nikankin, Niv Haim, and Michal Irani. *SinFusion: Training Diffusion Models on a Single Image or Video*. June 19, 2023. arXiv: 2211.11743[cs]. URL: http://arxiv.org/abs/2211.11743 (visited on 01/07/2024).

[Vas+23] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 1, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 04/20/2024).

[Ble24] Blender. *blender.org*. blender.org. 2024. URL: https://www.blender.org/ (visited on 04/20/2024).

[Ben] Emma Benjaminson. *The Reparameterization Trick*. URL: https://sassafras13.github.io/ReparamTrick/ (visited on 04/20/2024).

Appendix A

Project Plan

Project Plan

Name: DXZB7

Project Title: Diffusion Models for Novel Motion Generation

Supervisor's Name: Yuzuko Nakamura

Aims and Objectives

Aims: - Investigate the application of diffusion models for generating realistic and novel motions based on input motion sequences.

Objectives:

- **Literature Review:**
  - Conduct an in-depth review of diffusion models, motion generation techniques, and relevant works in the field.
  - Identify key challenges and opportunities in utilizing diffusion models for motion generation.
  - Explore existing research on reducing irregularities, particularly in foot contact consistency, in generated motions.
- **Data Preprocessing and Model Design:**
  - Analyze and preprocess the given motion sequence data into a suitable format for diffusion model investigation.
  - Design a diffusion model for motion generation, considering factors such as style, speed, and diversity.
  - Investigate and propose methods for improving foot contact consistency and reducing irregularities in generated motions.
- **Implementation of Investigative Framework:**
  - Implement the designed framework for investigating diffusion models in motion generation.
  - Conduct initial experiments and adjustments based on preliminary findings.
  - Establish a methodology for evaluating the effectiveness of the proposed irregularity reduction techniques.
- **Experiments and Analysis:**
  - Execute experiments using the diffusion model and the proposed irregularity reduction techniques.
  - Analyze the results, considering factors such as motion realism, diversity, and irregularity reduction.
  - Iteratively refine the investigative framework based on experimental outcomes.
- **Documentation and Paper Writing:**
  - Document the entire research process, including methodology, experimental setup, and results.

- Draft the research paper, including literature review, methodology, findings, and discussion.
- Revise and finalize the paper for submission to a peer-reviewed conference or journal.

Expected Outcomes/Deliverables

- **Reports and Documentation:**
  - Comprehensive literature survey on diffusion models in motion generation.
  - Documented investigative framework for diffusion model evaluation.
  - Research paper detailing the investigation, methodology, and findings.
  - Detailed documentation on proposed irregularity reduction techniques.
- **MoSCoW Requirements List:**
  - **Must-Have:**
    - Comprehensive literature survey on diffusion models.
    - Documented investigative framework for diffusion model evaluation.
    - Research paper detailing investigation and findings.
  - **Should-Have:**
    - Fully implemented diffusion model for motion generation.
    - Experimental results demonstrating effectiveness.
    - Analysis of irregularity reduction techniques.
  - **Could-Have:**
    - Integration with external motion capture data sources.
    - Real-time motion generation capabilities.
  - **Won't-Have:**
    - Integration with virtual reality or augmented reality platforms.

Work Plan

- **Project Start to End of October:**
  - Literature review on diffusion models and motion generation techniques.
  - Define initial project requirements and constraints.
- **Mid-October to Mid-November:**
  - Refine project requirements based on literature findings.
  - Develop a detailed investigative framework for diffusion model evaluation.
  - Conduct a risk analysis and identify potential project challenges.
- **November to Mid-January:**
  - Implement the investigative framework and conduct initial experiments.
  - Analyze preliminary results and make necessary adjustments.
  - Establish a robust methodology for evaluating irregularity reduction techniques.
- **Mid-January to Mid-March:**
  - Conduct experiments using the diffusion model and proposed irregularity reduction techniques.

- Analyze experimental results and iteratively refine the investigative framework.
- Document findings and prepare the initial draft of the research paper.

- **Mid-March to End of April:**
  - Revise and finalize the research paper based on feedback.
  - Prepare detailed documentation on proposed irregularity reduction techniques.
  - Complete the final research paper and submit it to a peer-reviewed venue.

Appendix B

Interim Report

Synthesizing motions of arbitrary skeletal topologies from a single training example using denoising diffusion probabilistic models

Author: DXZB7

Supervisor: Dr. Yuzuko Nakamura

Current Progress

1. Literature review – reviewed the following papers pertaining to state-of-the-art techniques of motion generation, as well as ways to modify generative models to only work with a single training example in cases where data is sparse:
   a) Li, Peizhuo, Kfir Aberman, Zihan Zhang, Rana Hanocka, and Olga Sorkine-Hornung. "GANimator: Neural Motion Synthesis from a Single Sequence." ACM Transactions on Graphics 41, no. 4 (July 2022): 1–12. https://doi.org/10.1145/3528223.3530157.
   b) Nikankin, Yaniv, Niv Haim, and Michal Irani. "SinFusion: Training Diffusion Models on a Single Image or Video." arXiv, June 19, 2023. http://arxiv.org/abs/2211.11743.
   c) Tevet, Guy, Sigal Raab, Brian Gordon, Yonatan Shafir, Daniel Cohen-Or, and Amit H. Bermano. "Human Motion Diffusion Model." arXiv, October 3, 2022. http://arxiv.org/abs/2209.14916.
   d) Zhang, Mingyuan, Zhongang Cai, Liang Pan, Fangzhou Hong, Xinying Guo, Lei Yang, and Ziwei Liu. "MotionDiffuse: Text-Driven Human Motion Generation with Diffusion Model." arXiv, August 31, 2022. http://arxiv.org/abs/2208.15001.
2. Used Adobe Mixamo to create several animations with non-human skeleton topology to use as training examples in the proposed model. Since animations are saved as industry standard .bvh files, I have also implemented a parser in Python to convert them into the

matrix representation suggested by papers in my literature review. Also created a parser for the reverse process to convert any matrix representation generated by the proposed model into a standard .bvh file. This is important since the .bvh files can be read by software such as Blender, which will be used for qualitative analysis of generated animations.

3. Working on initial prototype of diffusion model. Main aim of the model is to work with only a single training example, so model design considerations revolve around ensuring generation diversity. Using results from papers like SinFusion, a CovNext-based architecture is being implemented.

Remaining Work

1. Finish off prototype of CovNext-based architecture to generate results from training examples. Preliminary evaluation of model outputs can be done to measure model diversity.
2. Investigate implementation of losses such as velocity loss, position loss, and foot-contact loss to improve coherence of generated motions.
3. Investigate differences between 1D and 2D convolutions on the output quality. While 2D is more suited to spatial data such as images, it is likely that the temporal nature of animations works better with 1D convolutions.

Appendix C

Source code

```
import sys
import torch
from torch import optim
from dataloader.get_motion_data import get_motion_data
from models.diffusion import Diffusion
from models.nextnet import NextNet

BATCH_SIZE = 64
MAX_ITER = 50_000


def train(model, batch, optimizer, device):
    model.train()
    for data in batch:
        data = data.to(device)
        optimizer.zero_grad()
        loss = model(data)
        print(f'Loss: {loss}')
        loss.backward()
        optimizer.step()

def main():
    device = torch.device("cuda" if torch.cuda.is_available() else
   "cpu")

    # Load your motion data - adjust as necessary for your
   dataloader
    filename = sys.argv[1] if len(sys.argv) > 1 else "/home/
   aaryaman/Developer/Truebone_Z-OO/Spider/__Jump.bvh"
    motion_data = get_motion_data(filename)
    num_frames = motion_data.shape[2]
```

```
    batch = motion_data.repeat((BATCH_SIZE, 1, 1, 1))

    # Initialize the NextNet model (adjust parameters as needed)
    nextnet_model = NextNet(in_channels=num_frames, out_channels=
    num_frames).to(device)

    # Initialize the Diffusion model with NextNet model instance
    diffusion_model = Diffusion(nextnet_model).to(device)

    # Specify the optimizer
    optimizer = optim.Adam(diffusion_model.parameters(), lr=0.001)

    # Training loop
    for i in range(MAX_ITER):
        train(diffusion_model, batch, optimizer, device)
        if i % 1000 == 0:
            print('Iteration:', i)


if __name__ == "__main__":
    main()
```

Listing C.1: train.py

```
import numpy as np


# all transforms assume angles are in radians

def euler_to_quaternion(euler_angles):
    roll = euler_angles[0]
    pitch = euler_angles[1]
    yaw = euler_angles[2]

    cy = np.cos(yaw * 0.5)
    sy = np.sin(yaw * 0.5)
    cp = np.cos(pitch * 0.5)
    sp = np.sin(pitch * 0.5)
    cr = np.cos(roll * 0.5)
    sr = np.sin(roll * 0.5)

    w = cr * cp * cy + sr * sp * sy
    x = sr * cp * cy - cr * sp * sy
    y = cr * sp * cy + sr * cp * sy
    z = cr * cp * sy - sr * sp * cy

    return np.array([w, x, y, z])
```

```
24
25
26 def quaternion_to_euler(quaternion):
27     w = quaternion[0]
28     x = quaternion[1]
29     y = quaternion[2]
30     z = quaternion[3]
31     t0 = 2.0 * (w * x + y * z)
32     t1 = 1.0 - 2.0 * (x * x + y * y)
33     x1 = np.arctan2(t0, t1)
34     t2 = 2.0 * (w * y - z * x)
35     t2 = 1 if t2 > 1 else t2
36     t2 = -1 if t2 < -1 else t2
37     y1 = np.arcsin(t2)
38     t3 = 2.0 * (w * z + x * y)
39     t4 = 1.0 - 2.0 * (y * y + z * z)
40     z1 = np.arctan2(t3, t4)
41     return np.array([x1, y1, z1])
42
43
44 def quaternion_to_rot3(quaternion):
45     w = quaternion[0]
46     x = quaternion[1]
47     y = quaternion[2]
48     z = quaternion[3]
49
50     x2 = x * x
51     y2 = y * y
52     z2 = z * z
53     xy = x * y
54     xz = x * z
55     yz = y * z
56     wx = w * x
57     wy = w * y
58     wz = w * z
59
60     rot3 = np.array([[1 - 2 * (y2 + z2), 2 * (xy - wz), 2 * (xz +
    wy)],
61                      [2 * (xy + wz), 1 - 2 * (x2 + z2), 2 * (yz -
    wx)],
62                      [2 * (xz - wy), 2 * (yz + wx), 1 - 2 * (x2 +
    y2)]])
63     return rot3
64
65
66 def rot3_to_quaternion(rot3):
```

```
    trace = rot3[0, 0] + rot3[1, 1] + rot3[2, 2]
    if trace > 0:
        s = 0.5 / np.sqrt(trace + 1.0)
        w = 0.25 / s
        x = (rot3[2, 1] - rot3[1, 2]) * s
        y = (rot3[0, 2] - rot3[2, 0]) * s
        z = (rot3[1, 0] - rot3[0, 1]) * s
    else:
        if rot3[0, 0] > rot3[1, 1] and rot3[0, 0] > rot3[2, 2]:
            s = 2.0 * np.sqrt(1.0 + rot3[0, 0] - rot3[1, 1] - rot3
    [2, 2])
            w = (rot3[2, 1] - rot3[1, 2]) / s
            x = 0.25 * s
            y = (rot3[0, 1] + rot3[1, 0]) / s
            z = (rot3[0, 2] + rot3[2, 0]) / s
        elif rot3[1, 1] > rot3[2, 2]:
            s = 2.0 * np.sqrt(1.0 + rot3[1, 1] - rot3[0, 0] - rot3
    [2, 2])
            w = (rot3[0, 2] - rot3[2, 0]) / s
            x = (rot3[0, 1] + rot3[1, 0]) / s
            y = 0.25 * s
            z = (rot3[1, 2] + rot3[2, 1]) / s
        else:
            s = 2.0 * np.sqrt(1.0 + rot3[2, 2] - rot3[0, 0] - rot3
    [1, 1])
            w = (rot3[1, 0] - rot3[0, 1]) / s
            x = (rot3[0, 2] + rot3[2, 0]) / s
            y = (rot3[1, 2] + rot3[2, 1]) / s
            z = 0.25 * s
    return np.array([w, x, y, z])


def quaternion_to_6d(quaternion):
    rot3 = quaternion_to_rot3(quaternion)
    _6d = rot3[..., :2, :]
    _6d = _6d.reshape(_6d.shape[:-2] + (6,))
    return _6d


def _6d_to_quaternion(_6d):
    _6d = _6d.reshape(_6d.shape[:-1] + (2, 3))
    rot3 = np.concatenate([_6d, np.cross(_6d[..., :1, :], _6d[...,
    1:, :])], axis=-2)
    return rot3_to_quaternion(rot3)
```

Listing C.2: transforms.py

```python
import numpy as np
import torch
from torch import nn


def cosine_noise_schedule(timesteps, s=0.008):
    """
    cosine schedule
    as proposed in https://openreview.net/forum?id=-NEXDKk8gZ
    """
    steps = timesteps + 1
    x = np.linspace(0, steps, steps)
    alphas_cumprod = np.cos(((x / steps) + s) / (1 + s) * np.pi * 0.5) ** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return np.clip(betas, a_min=0, a_max=0.999)


class Diffusion(nn.Module):
    def __init__(self, model, timesteps=1000):
        super().__init__()
        self.num_timesteps = timesteps
        self.model = model
        self.betas = cosine_noise_schedule(timesteps)

        alphas = 1.0 - self.betas
        self.alphas_cumprod = np.cumprod(alphas)
        self.alphas_cumprod_prev = np.append(1.0, self.alphas_cumprod[:-1])
        self.alphas_cumprod_next = self.to_torch(np.append(self.alphas_cumprod[1:], 0.0))
        self.sqrt_alphas_cumprod = self.to_torch(np.sqrt(self.alphas_cumprod))
        self.sqrt_one_minus_alphas_cumprod = self.to_torch(np.sqrt(1.0 - self.alphas_cumprod))
        self.log_one_minus_alphas_cumprod = self.to_torch(np.log(1.0 - self.alphas_cumprod))
        self.sqrt_recip_alphas_cumprod = self.to_torch(np.sqrt(1.0 / self.alphas_cumprod))
        self.sqrt_recipm1_alphas_cumprod = self.to_torch(np.sqrt(1.0 / self.alphas_cumprod - 1))

        # calculations for posterior q(x_{t-1} | x_t, x_0)
        self.posterior_variance = self.betas * (1.0 - self.alphas_cumprod_prev) / (1.0 - self.alphas_cumprod)
```

```
38
39          # log calculation clipped because the posterior variance is
     0 at the
40          # beginning of the diffusion chain.
41          self.posterior_log_variance_clipped = self.to_torch(np.log(
42              np.append(self.posterior_variance[1], self.
    posterior_variance[1:])
43          ))
44          self.posterior_mean_coef1 = self.to_torch((
45                  self.betas * np.sqrt(self.alphas_cumprod_prev) /
    (1.0 - self.alphas_cumprod)
46          ))
47          self.posterior_mean_coef2 = self.to_torch((
48                  (1.0 - self.alphas_cumprod_prev)
49                  * np.sqrt(alphas)
50                  / (1.0 - self.alphas_cumprod)
51          ))
52
53          self.alphas_cumprod = self.to_torch(self.alphas_cumprod)
54          self.alphas_cumprod_prev = self.to_torch(self.
    alphas_cumprod_prev)
55          self.posterior_variance = self.to_torch(self.
    posterior_variance)
56
57      @staticmethod
58      def to_torch(arr, device='cuda'):
59          return torch.tensor(arr, dtype=torch.float32, device=device
    )
60
61      def q_posterior(self, x_start, x_t, t):
62          posterior_mean = self.posterior_mean_coef1[t] * x_start +
    self.posterior_mean_coef2[t] * x_t
63          posterior_variance = self.posterior_variance[t]
64          posterior_log_variance_clipped = self.
    posterior_log_variance_clipped[t]
65          return posterior_mean, posterior_variance,
    posterior_log_variance_clipped
66
67      def q_sample(self, x_start, t, noise=None):
68          if noise is None:
69              noise = torch.randn_like(x_start)
70
71          return self.sqrt_alphas_cumprod[t] * x_start + self.
    sqrt_one_minus_alphas_cumprod[t] * noise
72
73      def p_mean_variance(self, x, t, clip_denoised):
```

```
74          batch_size = x.shape[0]
75          t_tensor = torch.full((batch_size,), t, dtype=torch.int64,
    device='cuda')
76
77          x_recon = self.model(x, t_tensor)
78
79          if clip_denoised:
80              x_recon.clamp_(-1., 1.)
81
82          model_mean, posterior_variance, posterior_log_variance =
    self.q_posterior(x_start=x_recon, x_t=x, t=t)
83          return model_mean, posterior_variance,
    posterior_log_variance
84
85      @torch.no_grad()
86      def p_sample(self, x, t, clip_denoised=True):
87          model_mean, _, model_log_variance = self.p_mean_variance(x=
    x, t=t, clip_denoised=clip_denoised)
88          noise = torch.randn_like(x) if t > 0 else torch.zeros_like(
    x)  # no noise when t == 0
89          return model_mean + noise * (0.5 * model_log_variance).exp
    ()
90
91      @torch.no_grad()
92      def sample(self, batch_size, x_shape):
93          sample_shape = (batch_size, *x_shape)
94
95          timesteps = self.num_timesteps
96          res = torch.randn(sample_shape, device='cuda')
97          for t in reversed(range(0, timesteps)):
98              res = self.p_sample(res, t)
99          return res
100
101     @staticmethod
102     def kl(mean1, logvar1, mean2, logvar2):
103         kl = 0.5 * (logvar2 - logvar1) - 0.5 + (torch.exp(logvar1)
    + (mean1 - mean2) ** 2) / (2 * torch.exp(logvar2)) - 0.5
104         return kl
105
106     @staticmethod
107     def discrete_gaussian_log_likelihood(x, means, log_scales):
108         log_scales = torch.clamp(log_scales, min=1e-12)
109         inv_stdv = torch.exp(-log_scales)
110         return -0.5 * ((x - means) * inv_stdv) ** 2 - log_scales -
    0.5 * np.log(2 * np.pi)
111
```

```
    def compute_loss(self, x_start, x_t, t, clip_denoised=True):
        real_mean, real_variance, real_log_variance = self.
    q_posterior(x_start, x_t, t)
        model_mean, model_variance, model_log_variance = self.
    p_mean_variance(x_t, t, clip_denoised)
        kl = self.kl(real_mean, real_log_variance, model_mean,
    model_log_variance)
        kl = kl.mean(dim=list(range(1, len(kl.shape)))) / np.log
    (2.0)
        decoder_nll = -self.discrete_gaussian_log_likelihood(x_t,
    model_mean, 0.5 * model_log_variance)
        decoder_nll = decoder_nll.mean(dim=list(range(1, len(x_t.
    shape)))) / np.log(2.0)
        output = torch.where((t == 0), decoder_nll, kl)
        return output

    def forward(self, x):
        frames = x.shape[2]
        print(x.shape)
        t = np.random.randint(0, self.num_timesteps)
        t_tensor = torch.full((frames,), t, dtype=torch.int64,
    device='cuda')
        noise = torch.randn_like(x)
        x_noisy = self.q_sample(x_start=x, t=t, noise=noise)
        x0_recon = self.model(x_noisy, t_tensor)
        return self.compute_loss(x, x0_recon, t)
```

Listing C.3: diffusion.py

```
import math

import torch
from torch import nn


class SinusoidalPosEmb(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, x):
        device = 'cuda'
        half_dim = self.dim // 2
        emb = math.log(10000) / (half_dim - 1)
        emb = torch.exp(torch.arange(half_dim, device=device) * -
    emb)
        emb = x[:, None] * emb[None, :]
```

```
        emb = torch.cat((emb.sin(), emb.cos()), dim=-1)
        return emb


class LayerNorm(nn.Module):
    def __init__(self, dim, eps=1e-5):
        super().__init__()
        self.eps = eps
        self.g = nn.Parameter(torch.ones(1, dim, 1, 1))
        self.b = nn.Parameter(torch.zeros(1, dim, 1, 1))

    def forward(self, x):
        var = torch.var(x, dim=1, unbiased=False, keepdim=True)
        mean = torch.mean(x, dim=1, keepdim=True)
        return (x - mean) / (var + self.eps).sqrt() * self.g + self
    .b


class ConvNextBlock(nn.Module):
    """ https://arxiv.org/abs/2201.03545 """

    def __init__(self, dim, dim_out, *, emb_dim=None, mult=3, norm=
    True):
        super().__init__()

        self.mlp = nn.Sequential(
            nn.GELU(),
            nn.Linear(emb_dim, dim)
        )

        self.ds_conv = nn.Conv2d(dim, dim, 7, padding=3, groups=dim
    )

        self.net = nn.Sequential(
            LayerNorm(dim) if norm else nn.Identity(),
            nn.Conv2d(dim, dim_out * mult, 3, padding=1),
            nn.GELU(),
            nn.Conv2d(dim_out * mult, dim_out, 3, padding=1)
        )

        self.res_conv = nn.Conv2d(dim, dim_out, 1) if dim !=
    dim_out else nn.Identity()

    def forward(self, x, emb=None):
        h = self.ds_conv(x)
        h = self.net(h)
```

```
        return h + self.res_conv(x)


class NextNet(nn.Module):
    """
    A backbone model comprised of a chain of ConvNext blocks, with skip connections.
    The skip connections are connected similar to a "U-Net" structure (first to last, middle to middle, etc).
    """

    def __init__(self, in_channels, out_channels, depth=16, filter_size=64):
        super().__init__()

        dims = filter_size * depth

        time_dim = dims
        emb_dim = time_dim
        self.depth = depth
        self.layers = nn.ModuleList([])

        # First block doesn't have a normalization layer
        self.layers.append(ConvNextBlock(in_channels, dims, emb_dim=emb_dim, norm=False))

        for i in range(1, math.ceil(self.depth / 2)):
            self.layers.append(ConvNextBlock(dims, dims, emb_dim=emb_dim, norm=True))
        for i in range(math.ceil(self.depth / 2), depth):
            self.layers.append(ConvNextBlock(2 * dims, dims, emb_dim=emb_dim, norm=True))

        # After all blocks, do a 1x1 conv to get the required amount of output channels
        self.final_conv = nn.Conv2d(dims, out_channels, 1)

        # Encoder for positional embedding of timestep
        self.time_encoder = nn.Sequential(
            SinusoidalPosEmb(time_dim),
            nn.Linear(time_dim, time_dim * 4),
            nn.GELU(),
            nn.Linear(time_dim * 4, time_dim)
        )

    def forward(self, x, t):
```

```
        embedding = self.time_encoder(t)
        print(embedding.shape)
        residuals = []
        for layer in self.layers[0: math.ceil(self.depth / 2)]:
            x = layer(x, embedding)
            residuals.append(x)

        for layer in self.layers[math.ceil(self.depth / 2): self.depth]:
            print(x.shape)
            print(residuals[-1].shape)
            print(len(residuals))
            print("")
            x = torch.cat((x, residuals.pop()), dim=1)
            x = layer(x, embedding)

        return self.final_conv(x)
```

Listing C.4: nextnet.py

```
import numpy as np
import torch

from bvh.load import BvhLoader
from transforms import transforms


def get_motion_data(filename, motion_repr='quaternion'):
    motion_data = None
    bvh = BvhLoader(filename)
    positions, rotations = bvh.positions, np.radians(bvh.rotations)

    if motion_repr == 'quaternion':
        frames, joints = rotations.shape[:2]
        quat_rotations = np.zeros((frames, joints, 4))

        for i in range(frames):
            for j in range(joints):
                quat_rotations[i, j] = transforms.euler_to_quaternion(rotations[i, j])

        motion_data = np.concatenate([positions, quat_rotations], axis=2)

    elif motion_repr == '6d':
        frames, joints = rotations.shape[:2]
        repr_6d_rotations = np.zeros((frames, joints, 6))
```

```
26
27          for i in range(frames):
28              for j in range(joints):
29                  repr_6d_rotations[i, j] = transforms.
    quaternion_to_6d(transforms.euler_to_quaternion(rotations[i, j])
    )
30
31          motion_data = np.concatenate([positions, repr_6d_rotations
    ], axis=2)
32
33      motion_data = torch.from_numpy(motion_data).permute(1, 2, 0).to
    (torch.float32)
34      motion_data.to(torch.float32)
35      return motion_data
```

Listing C.5: getmotiondata.py

```
1  import numpy as np
2
3  from bvh.bvh import Bvh
4
5
6  class BvhLoader:
7      def __init__(self, filename):
8          self.bvh = self.load_bvh(filename)
9          self.joint_names = self.get_joint_names()
10
11         self.parent_indices = self.get_parent_indices()
12         self.offsets = self.get_offsets()
13         self.joint_channels = self.get_joint_channels()
14
15         self.rotations = self.get_rotations(canonical=True)
16         self.positions = self.get_positions()
17
18     @staticmethod
19     def load_bvh(filename):
20         with open(filename, 'r') as f:
21             return Bvh(f.read())
22
23     def get_joint_names(self):
24         return self.bvh.get_joints_names()
25
26     def get_parent_indices(self):
27         return [self.bvh.joint_parent_index(joint) for joint in
    self.joint_names]
28
29     def get_offsets(self):
```

```
30          self.offsets = np.empty((0, 3))
31          for joint in self.joint_names:
32              self.offsets = np.vstack((self.offsets, self.bvh.
    joint_offset(joint)))
33          return self.offsets
34
35      def get_joint_channels(self):
36          # assumes all joints have same channels
37          return self.bvh.joint_channels(self.joint_names[0])
38
39      def get_position(self, frame_index):
40          positions = np.empty((0, 3))
41          for joint in self.joint_names:
42              positions = np.vstack(
43                  (positions, self.bvh.frame_joint_channels(
    frame_index, joint, self.joint_channels[:3])))
44          return positions
45
46      def get_rotation(self, frame_index, canonical=False):
47          # canonical: if True, return rotations in xyz order
48
49          rotations = np.empty((0, 3))
50          for joint in self.joint_names:
51              rotations = np.vstack(
52                  (rotations, self.bvh.frame_joint_channels(
    frame_index, joint, self.joint_channels[3:])))
53          return rotations[:, [2, 1, 0] if canonical else [0, 1, 2]]
54
55      def get_positions(self):
56          positions = np.empty((0, len(self.joint_names), 3))
57          for frame_index in range(self.bvh.nframes):
58              positions = np.append(positions, np.expand_dims(self.
    get_position(frame_index), 0), axis=0)
59          return positions
60
61      def get_rotations(self, canonical=False):
62          rotations = np.empty((0, len(self.joint_names), 3))
63          for frame_index in range(self.bvh.nframes):
64              rotations = np.append(rotations, np.expand_dims(self.
    get_rotation(frame_index, canonical), 0), axis=0)
65          return rotations
```

Listing C.6: load.py