
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INFORMATION SYSTEMS**
**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**


# GENERATING ANIMATIONS WITH NEURAL NETWORKS

**GENEROVÁNÍ ANIMACÍ NEURONOVÝMI SÍTĚMI**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **FILIP DRÁBER**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. MICHAL HRADIŠ, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2021**

**Brno University of Technology**
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

# Bachelor's Thesis Specification


Student: **Dráber Filip**
Programme: Information Technology
Title: **Generating Animations with Neural Networks**
Category: Image Processing

Assignment:

1. Familiarize yourself with neural networks and skeletal animation.
2. Study existing methods for generating skeletal animations which use neural networks.
3. Propose your own method based on the existing approaches.
4. Prepare a data set suitable for experiments.
5. Implement the method and experiment on the data set.
6. Analyze and interpret the results and suggest possible future extensions and modifications of the method.
7. Prepare a short video demonstrating the method and results.

Recommended literature:

- He Zhang, Sebastian Starke, Taku Komura, and Jun Saito.: Mode-adaptive neural networks for quadruped motion control. SIGGRAPH. 2018.
- Sebastian Starke, He Zhang, Taku Komura, and Jun Saito.: Neural state machine for character-scene interactions. SIGGRAPH Asia. 2019.
- Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne.: DeepMimic: example-guided deep reinforcement learning of physics-based character skills. SIGGRAPH, 2018.
- Sebastian Starke, Yiwei Zhao, Taku Komura, and Kazi Zaman.: Local motion phases for learning multi-contact character movements. SIGGRAPH, 2020.

Requirements for the first semester:

- Items 1 to 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Hradiš Michal, Ing., Ph.D.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: May 5, 2021

# Abstract

While motion capture serves as a mean for animators to circumvent some of the most arduous aspects of creating realistic animation, there is still a lot of work hiding in annotating and structuring the data. I solve this problem by designing a neural network which can be trained on a motion capture data file to reproduce human locomotion visualized in an application which allows for the user to control the character's direction. I also subject various methods of training an autoregressive model to experiments and find which method trades training times for performance the best. Additionally, I remark how the addition of certain control features to frame-by-frame generations impacts the use of recurrent neural networks for this task.



# Abstrakt

Ačkoli je snímání pohybu už tak nástrojem, který má animátorům pomoci zjednodušit ty nejsložitější aspekty tvorby realistických animací, spousta námahy je stále ukrytá v anotování a strukturalizaci těchto dat. Tento problém řeším návrhem neuronové sítě, která může být natrénována na datovém souboru nasnímaného pohybu tak, aby reprodukovala lidský pohyb, který je vizualizován v aplikaci, které umožňuje uživateli tento pohyb ovládat. Také experimentuji s různými metodami trénování autoregresivního modelu, a na základě toho určuji, která metoda nejlépe vyvažuje dobu trénování a výkon. Dalším postřehem je, jak přidání ovládacích hodnot do vlastností generovaných snímků ovlivňuje použití rekurentních neuronových sítí pro tento úkol.



# Keywords

animation, motion capture, BVH, machine learning, neural networks, LSTM, discriminative models, autoregressive models

# Klíčová slova

animace, snímání pohybu, BVH, strojové učení, neuronové sítě, LSTM, diskriminativní modely, autoregresivní modely


# Reference

DRÁBER, Filip. *Generating Animations with Neural Networks.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Hradiš, Ph.D.

# Rozšířený abstrakt

Snímání pohybu (motion capture, mocap) je proces převádění pohybu subjektu (lidského, ale i např. zvířecího) do digitální formy. Jedno z jeho primárních využití lze najít v animačním průmyslu, protože tvoření realistické animace postav je nesrovnatelně nákladné. I tak ale není zdaleka vyhráno. Data vznikající touto metodou jsou často neanotovaná, a jeden soubor může popisovat několik různých pohybů, a tak se část úsilí pouze přesouvá do zpracovávání těchto dat. Strojové učení se nabízí jako jeden ze způsobů, jak tyto procesy automatizovat.

Tento problém se dá rozdělit na dvě hlavní části – generování pohybu a jeho ovládání. Generování pohybu není ve světě strojového učení zrovna obskurní problém. Převládá v něm však generování řízené fyzikou, ve kterém je pohyb utvářen tak, aby modely respektovaly zákonitosti fyziky jako hmotnosti, elasticitu, točivé momenty a další. Generování řízené daty je ve srovnání zastoupené méně.

Neuronové sítě jako druh modelu strojového učení své uplatnění v tomto odvětví nacházejí. Dokonce i přes podstatu řešeného problému jakožto generování takřka nekonečné série dat lze hledat úspěšné příklady takových modelů, které lze popsat jako feed-forward – zpracování vstupu ve formě vektoru příznaků a generace výstupu není nijak ovlivněno předchozími vektory, a ani nijak neovlivňuje budoucí generování. Většina těchto modelů se však snaží nějakým způsobem charakterizovat časovou dimenzi generovaného pohybu, například navázáním parametrů neuronové sítě na cyklickou funkci, jejímž argumentem je hodnota, která popisuje, v jaké fázi kroku levé či pravé nohy se postava zrovna nachází.

Tato práce se však zaměřuje na rekurentní sítě, tedy takové, jejichž zpracování a výstup ovlivní zpracování budoucích vstupů. V případě časové řady snímků tedy v modelu zůstává kvantifikovaný dojem o tom, jaký pohyb aktuálně zpracovává. Místo klasické rekurence, kdy se buď celý generovaná výstup sítě nebo jeho část vrací do modelu jako vstup, lze však využít tzv. vrstvu LSTM – Long Short-Term Memory. Kromě zpracovávání svého předchozího výstupu má totiž tato vrstva ještě skrytý stav reprezentovaný vektorem, jehož hodnoty se síť učí opravovat na základě přicházejících vstupů. Některé hodnoty tohoto skrytého stavu může síť měnit vyjímečně, a tak zůstanou v modelu, aniž by byly vytlačeny nově příchozími vstupy.

Jeden ze způsobů ovládání pohybu je pomocí trajektorie. Ta je pro každý snímek reprezentována minulými a budoucími pozicemi postavy v pohybu. Zavedením trajektorie do vektoru příznaků daného snímku umožňuje před modelu charakteristiku pohybu, a pokud bychom tuto trajektorii za běhu změnili např. na základě uživatelského vstupu, model by měl generovat snímky odpovídající této změně pohybu. Protože ale požadovaná trajektorie pravděpodobně nebude přesně odpovídat žádné trajektorii v originální datové sadě, tak se trajektorie do vektoru příznaků generuje smícháním mezi trajektorií vygenerovanou modelem pro daný snímek a naší požadovanou trajektorií.

Dále je součástí práce také obecný popis datových formátů pro snímání pohybu, ale také podrobný popis formátu BVH, jehož transformaci na sadu příznakových vektorů jednotlivých snímků jsem v rámci práce implementoval. Pro práci byla využita datová sada SAUCE Project Motion Library, která obsahuje několik záznamů lidské chůze v různých náladách nebo při různých aktiviách.

Následně popisuji ředzpracování dat a veškeré transformace, které je třeba provést k vytvoření příznakových vektorů podle datové sady, pak i samotné trénování. Právě u trénování popisuji, že při trénování modelu pouze na hodnotách z datové sady může dojít k problému za běhu, kdy na vstup modelu naopak přichází pouze hodnoty, které model sám vygeneroval, a může se v nich nacházet chyba, která je od původních hodnot

odlišuje natolik, že model vygeneruje snímek s ještě větši chybou. Navrhuji toto řešit už při trénování buď postupným přidáváním šumu na vstup nebo navracením generovaného snímku zpátky na vstup modelu místo použití příznakového vektoru z datové řady.

Kromě implementačních detailů procesu trénování, generování a vizualizace pohybu také podrobuji jednotlivé modely (rozlišené právě trénovacími strategiemi popsanými výše) experimentům, které umožňují se vyjádřit ke kvalitě jednotlivých modelů jinak, než od oka. Měřením stráveného času a postupných ztrát zjišťuji, že poslední strategie může být až o šedesát procent pomalejší, než ostatní dvě. Dalšími metrikami pak jsou: míra, do jaké animovaným postavám kloužou po podlaze nohy, pak míra, do jaké otočení postavy následuje očekávanou trajektorii, a nakonec srovnání spekter různých příznaků charakterizovaných jako signálů.

Nakonec diskutuji tyto výsledky, jak by s modelem mohlo být naloženo dále, ale také vypozorovaný fakt, pokud postava náhodou zamrzne v jedné póze, tak právě ovlivnění trajektorie může mít za následek, že se postava z této pózy vytrhne a bude pokračovat v chůzi. Toto pozorování považuju za indikaci, že přidáním trajektorie lze vynutit dlouhodobost pohybu, se kterou mají jinak modely na podobných principech problém.

# Generating Animations with Neural Networks

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Michal Hradiš, Ing., PhD. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Dráber
May 17, 2021


## Acknowledgements

I would like to thank Mr. Michal Hradiš, Ing., PhD., for his supervision and guidance in this relatively complex field of machine learning. I would also like to thank Mr. Marek Šolony, Ing., PhD., for helping to provide the motion capture data set utilized in this Bachelor's thesis.

# Contents

# Chapter 1

# Introduction

In this thesis, I implement and present an application featuring a recurrent neural network model which generates human character locomotion based on an unstructured motion capture file, visualize it, and enable the user to control the character's movement. This solution is motivated by the difficulty that still arises when supplementing motion capture for animation to make the animating process easier. The issue stems from the fact that most motion capture data is unstructured and needs complex pre-processing. I aim to streamline this process for human locomotion by training a neural network model to transform a single motion capture file into locomotion. The finished result parses a motion capture file in the BVH format (also described in this thesis) and generates locomotion which reflects the one described in the file. In terms of contributions, the experiments indicate how various strategies for training autoregressive models impact the performance, and how the presence of a trajectory as an added set of control features impacts the problem of these models with long-term maintenance of periodic motion

First, the problem at hand is broken down into motion generation and motion control, and various concepts utilized in this field are described before several existing designs which try to solve these two problems are presented. Certain solutions far more complex than the one proposed by this thesis are mentioned, should the reader be interested in how deep learning can be mixed up and experimented with to find unconventional solutions.

In brief, the existing data sets for motion capture are discussed. While the data sets are not primarily meant for experimentation in machine learning and need some processing and interpretation, they include some of those most often reached for when one tries to solve the problem at hand. Then, the data set utilized in this thesis is described in greater depth, and the choice of this data set is then defended.

I then present a new model, the Trajectory-guided Locomotion Recurrent Neural Network, cross-implementing elements from other solutions. The preprocessing of the data is described in detail, and so is the training process as well as the model's runtime, the use of which is paid attention to. Details of the application involving the model are explored, and the additional implementation of the supporting visual application is described.

Finally, the model is subjected to experimentation with the explicit goal of looking for training process differences between various training methods, as well as differences in runtime results, visible to the naked eye or not. The results and the future of both the model alone and the application as a whole are all discussed.

# Chapter 2

# Motion Generation

Making a character move does, in fact, make for a relatively common machine learning problem. Two forms of generation are prevalent. First, there is physics-based generation, which assigns physicality to the characters and generates motion that respects the character's weight, elasticity, torques and such, as mentioned by Zhang, Starke et al. [22]. Then, however there is the data-driven approach, which instead processes data descripting motion, definitely including motion capture data.

## 2.1 Motion Capture

Motion capture (mocap) is a popular method utilized to create rigs for animators which have been under more and more pressure to produce realistic animations, which are otherwise rather difficult to put together using traditional animation methods. This method involves transforming natural human motion and locomotion via digital capture and thus transforming it into a format that can be utilized by animators afterwards.

Nowadays, there are numerous methods to perform motion capture [14]. The most common dividing line is whether they utilize markers, as in, small objects or devices attached directly to the actor, or not. Markers can come in various forms, from passive or active optical points that a camera can detect, to magnetic, acoustic or mechanical devices that allow to locate the actor's individual joints in the three-dimensional space.

Furthermore, methods can be distinguished by whether they calculate the relevant data immediately during capture, or they infer it from, for example, camera recordings. Either way, the animators who make use of mocap data have not won just yet. The nature of motion capture recording sessions generally result in long, unstructured data files which contain all sorts of different motions. While a key may be provided which describes the series of actions taken by the actor, it can still be a lengthy task to manually annotate segments of the data, and may still pose an issue for the animators of certain motions not blending together too well. One solution offered for these problems is to utilize machine learning and train models on these data files which would select and blend the necessary motions as the animation requires.

### General Format

While there are several formats that can describe the data generated during motion capture, similarities can be drawn between how they approach the issue. Generally, each defines their described character as a skeleton, comprising of bones (or joints), which are (generally)

structured hierarchically. Each of these objects has a given amount of channels, or degrees of freedom. These parameters represent values such as position, rotation, or scale. Finally, each motion is broken down into frames, their amount dictated by the motion duration and the rate at which the frames are generated. Each frame carries with itself the values of each bone's channels, and it is these values which, when played back, generate the desired motion [13].

## 2.2 Data-driven Generation

Recreation of character locomotion from motion capture data is not represented among common machine learning problems as much as certain other tasks, for example facial recognition, text prediction or even pose prediction from video data. This problem consists of two main aspects – understanding and parsing the data created during motion capture, and constructing a machine learning model that would learn to predict and generate new data based on the existing ones. In addition to these two problems and their individual challenges, another one presents itself in the form of mapping the user inputs to character control, resulting in a free-running animation of a character whose locomotion should respond to the user's wishes.

### Recurrent Neural Networks

Since we are effectively processing and predicting a series of frames, it is in due course to utilize a machine learning model that can utilize information regarding the processed vectors. While there are still existing solutions detailed in Chapter 2.3 which avoid utilizing recurrent elements to a degree of success, this thesis and the implemented application make use of layers which makes use of storing and processing values generated "on the side" when processing the previous feature vector.

### Long Short-Term Memory Layers

To avoid certain issues that arise when implementing classic recurrent neural network models, a new type of a recurrent cell has been developed – the Long Short-Term Memory cell, or LSTM cell [4]. While considerably more complex (see Figure 2.1), it surpasses its counterparts for tasks which involve the learning of long-term dynamics and dependencies. For example, when inferring from text and predicting how it continues, an element in the cell state can track the gender of the most recent subject to select proper pronouns even after considerable delays [15]. It maintains its own hidden cell state, parts of which are then, via gating functions, erased or replaced by new values, and then exported onto the output.

The mathematical representation of the LSTM cells used in this thesis is described in the PyTorch documentation [18] approximately as follows:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ii}\mathbf{x}_t + \mathbf{b}_{ii} + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_{hi}) \tag{2.1}$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_{if}\mathbf{x}_t + \mathbf{b}_{if} + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_{hf}) \tag{2.2}$$
$$\mathbf{g}_t = \phi(\mathbf{W}_{ig}\mathbf{x}_t + \mathbf{b}_{ig} + \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_{hg}) \tag{2.3}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_{io}\mathbf{x}_t + \mathbf{b}_{io} + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_{ho}) \tag{2.4}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \tag{2.5}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \phi(\mathbf{c}_t) \tag{2.6}$$

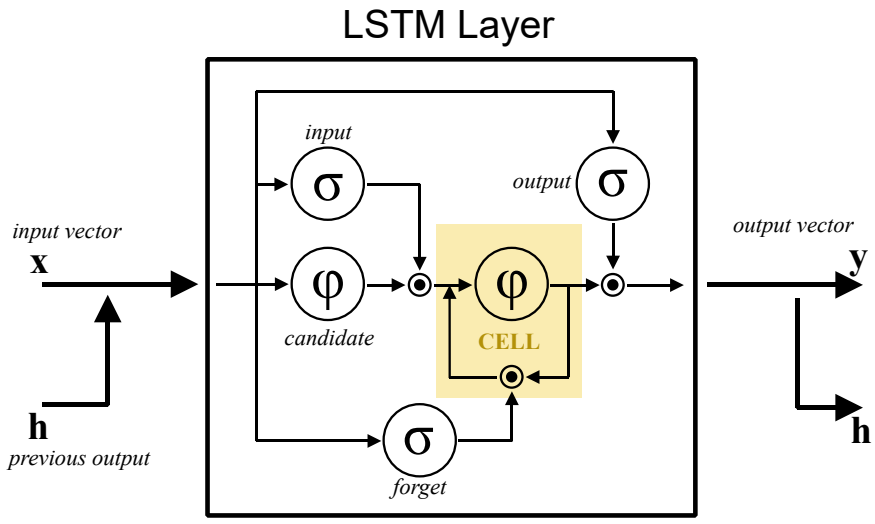


Figure 2.1: A diagram of an LSTM layer, visualizing how the current and previous input affect the cell state and generate the output out of updated cell state. Notice the sigmoid functions $\sigma$ preceding element-wise products, serving as a form of "selection" of individual values from the vector that is the other factor.

The input $\mathbf{x}$ at a given step $t$ is, along with the cell's previous output $\mathbf{h}_{t-1}$ passed through four nonlinear gates with their specific weights $\mathbf{W}$ and biases $\mathbf{b}$, as well as these two functions: The sigmoid, $\sigma(x) = \frac{1}{1+e^{-x}}$, and the hyperbolic tangent $\phi$. Each of these four vectors then comes into play when updating the cell state $\mathbf{c}_t$ and producing an output. First, the forget vector $\mathbf{f}_t$, consisting of values from the interval $[0, 1]$ determines to what degree are the elements of the old cell state $\mathbf{c}_{t-1}$ retained using the element-wise product. Candidate values $\mathbf{g}_t$ to be added to the cell state are then modulated in a similar fashion using the input gate vector $\mathbf{i}_t$ and added to the cell state. Finally, the output gate vector $\mathbf{o}_t$ dictates which values from the new cell state $\mathbf{c}_t$ are selected and passed onto the output of the LSTM cell (as well as its own input at step $t + 1$).

### Teacher Forcing

One possible and fairly straightforward strategy for training recurrent neural networks is to repeatedly provide an input $\mathbf{x}_i$), which is taken from the ground truth for the network to then find the likeliest output $\mathbf{y}_i$ based on the input and the recurrent units' memory state. The next input from the sequence of inputs, $\mathbf{x}_{i+1}$, is, again, taken from the ground truth. During runtime, however, the ground truth is no longer available, and thus the previous output, $\mathbf{y}_i$ is provided. Small discrepancies between the ground truth and the network's output, especially when introduced early into the sequence, can quickly push the outputs out of the state space the network has been trained on. This approach is known as teacher forcing [21] [1]. One proposed method to make the model less vulnerable to small deviations is to begin applying noise to the input vectors. Ideally, the intensity of the noise should be dictated by the variances of the individual features. As a result, the model effectively learns to fix corrupted data, which should enable it to mitigate minor discrepancies it produces on its output. This method is utilized by Fragkiadaki et al. in their proposed recurrent neural network model for motion generation [5].

Another method is to provide the network's own outputs as the following inputs. Bengio et al. [1] first suggest to, throughout training, randomly determine whether the next input should be taken from the ground truth or the neural network's previous output. They then propose, however, to skew the probability of performing this swap. Specifically, it should not happen too often early into the training, as the outputs will still be relatively random, but should rather become more common towards the end of the training, when the network's behavior much more closely resembles its expected behavior during runtime.

Lamb, Goyal et al. [7] instead propose to utilize the generative adversarial network framework and train a second model which forces the recurrent model to act the same whether it is receiving ground truth inputs, or sampling from its own outputs.

## 2.3 Motion Generation Solutions

Despite not being a renowned machine learning problem, models which generate animations trained on motion capture data have been explored and succsefully implemented by various teams. This section describes a selection of attempts at recreating animations and the various techniques they invented and employed to patch various issues which arise in this problem.

### Non-recurrent Neural Network Models

**PFNN** To avoid the common pitfall of the animated character slipping into a neutral pose after an arbitrary amount of time has passed, neural network models have been expanded with various mechanisms. Holden et al. [9], inspired by various works that attempt to map certain parameters directly onto the latent space of the model [11], or outright directly parametrize the network's weight [12], present the Phase-Functioned Neural Network (PFNN). The model's core is relatively simple, consisting only of a neural network of three layers of nonlinear activation functions. However, instead of training the network by altering the parameters of these layers, the weights are generated by what the article calls a "phase function" = a cyclic function controlled by a single variable, the phase $p$. Training the model instead involves finding the parameters for this function that would result in the best rectified linear unit layer weights.

The phase $p$ is a parameter within the interval $\langle 0; 2\pi)$, and, during pre-processing, is automatically assigned to each frame. Key values such as 0 and $\pi$ are automatically assigned to foot contacts, and the rest is interpolated. The phase is a part of the feature vector, and the network is trained to predict the phase of the next frame.

As a result, the animated character is constantly driven by the cyclic function, or, rather, the changes in phase, preventing slipping into a frozen frame for a prolonged amount of time.

**MANN** Zhang, Starke et al. [22] go a step beyond human locomotion and instead opt to try and predict the locomotion of quadrupeds. This is made considerably more complex by the sheer amount of gaits that they perform at various paces (rendering the phase function approach proposed by Holden et al. [9] moot). On top of that, the fact that quadruped motion cannot be as easily directed during motion capture as that of humans results in more chaotic, unstructured mocap data. The MANN – Mode-Adaptive Neural Network – produces fluid motion responsive to user control. Its weights are dynamically altered by a gating network, which in turn is controlled by "expert weights" inferred from motion features such as the gait.

**Local Motion Phases** Starke et al. [17] tackle the issue of complex motion which involves multiple asynchronous motions of various body parts. Such motion cannot be easily described by a single phase variable, and thus a model is proposed which assigns a local phase to each bone based on the bone's contacts with the environment. This approach is demonstrated on the irregular, complex and contact heavy action of playing basketball.

## A Recurrent Neural Network Model

Due to the nature of animation being a linear sequence, it is only natural that recurrent neural networks ended up being utilized for the task of recreating animation of human locomotion.

**ERD** Fragkiadaki et al. [5] propose the ERD – Encoder-Recurrent-Decoder model. The model utilizes two LSTM layers, but expands upon other existing LSTM-based models by "wrapping" these layers between non-linear encoder layers and non-linear decoder layers. This model is then both utilized to predict human locomotion from motion capture data as well as from video capture. It appears to be the first attempt to utilize a recurrent neural network model on human motion, because previous recurrent neural network models have been utilized as language models for processes such as handwriting generation and image captioning.

It is mentioned by Zhang, Starke et al. in their Mode-Adaptive Neural Network paper [22] that recurrent models such as the ERD exist, but tend to slip into a frozen pose (citing the solution by Fragkiadaki et al. as an example). Utilizing a form of character control in this thesis's solution is also an attempt at combatting this phenomenon.

# Chapter 3

# Character Control

This chapter first offers an overview of different existing solutions for individual smaller aspects of the overarching problem that is controlling the character. The largest gap to bridge is the fact that the unstructured motion capture data set does not come with predefined user inputs that the model could refer to, leaving it with no way of determining if any motion in the data set corresponds to a given user input.

## 3.1 Trajectory Blending

One method utilized in the solutions listed in this chapter is to create a trajectory of the animated character's motion. In its most basic form, a trajectory for a given frame consists of the character's positions in the surrounding frames. In these solutions as well as the one implemented in this thesis, the relevant information is only collected from a subsampled set of the surrounding frames.

Characterizing the locomotion using a trajectory additionally enables a form of user control in neural network models. However, simply generating a trajectory and forcing it into the feature vector on the model's input may present the model with values it has not encountered when training on the ground truth, and has indeed proven ineffective during the implementation of the model for this thesis.

One method to combat this, presented by Holden et al. [9] (mentioned below) and utilized in this solution is to blend the model's predicted future trajectory with the trajectory expected based on user control. More specifically, a given point $\mathbf{t}_b \in \mathbb{R}^3$ of the result trajectory is generated by mixing the corresponding points of the predicted trajectory $\mathbf{t}_p \in \mathbb{R}^3$ and the user control trajectory $\mathbf{t}_c \in \mathbb{R}^3$, weighing closer towards the point from the user control trajectory the further into the future the points are.

$$\mathbf{t}_b^n = (1 - t^\tau)\mathbf{t}_p^n + t^\tau \mathbf{t}_c^n \tag{3.1}$$

The value $t$ on the interval $(0; 1)$ corresponds to how far into the future the trajectory's point is, and the value $\tau$ is a parameter that affects the bias of the blended trajectory towards the originally predicted one.

## 3.2 Character Control Solutions

The following works, on top of finding ways of generating locomotion, also explore the various methods of solving this issue. The solutions either detail the evolution of the

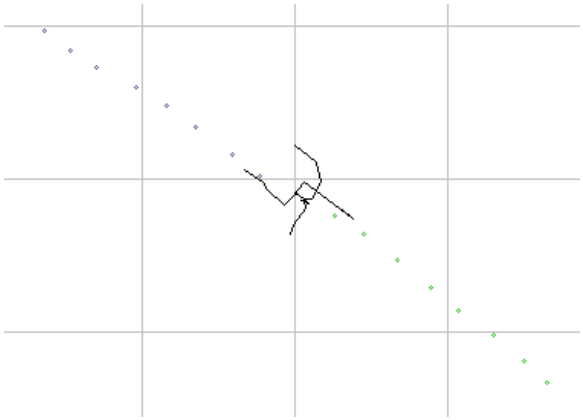

Figure 3.1: A representation of the trajectory – sixteen character positions from the past and in the future – in the application implemented in this thesis (edited).

approach to this problem, offer technologies and elements that have been adopted by the solution proposed in this thesis, or involve components too complex to be implemented within the scope of this thesis.

**Motion Matching** While this may not be work in the field of neural networks (or machine learning in general), Simon Clavet's Motion Matching presentation at the Game Developers Conference 2016 [3] showcases an approach to select proper motion capture data for animations and to blend animations together to create realistic movement guided by user control. Specifically, at a given point, the best animation to select for motion is found by a brute-force greedy search across all the animation clips and finding one with the lowest cost – the sum of various errors and mismatches that a given animation would result in. Its trajectory and cost-based approach has laid the groundwork for various future endeavors in motion generation.

**PFNN** In the attempt of Holden et al. [9] to generate character motion using a phase variable, a method is explored to match user input to motion via blending the trajectory predicted by the neural network with the desired trajectory. In this case, the trajectory is a complex combination of values, ranging from the displacement of the character's root to its individual velocities and directions at a given point in time in relation to the current frame. Moreover, since this model also deals in generating motion across terrain, the trajectory tracks the terrain height at the root positions as well as at points offset to the side of the root positions, resulting in the feature vector containing information about the character's surroundings.

**Local Motion Phases** Starke et. al [17] follow up on the groundwork provided in Motion Matching [3], but instead propose a generative control model, trained adversarially to create a low-dimensional latent space of user control. More importantly, however, it circumvents part of the issue by already matchingly appointing the user control inputs with the data set.

# Chapter 4

# Motion Capture Data Sets

The existing data sets for motion generation generally come from general motion capture data sets. Several are available on-line, and this chapter describes the two most common sets as well as the one utilized in the development and implementation of the model presented by this thesis. Specifically Ubisoft Montréal's data set can serve as a good basis for more mocap-driven research, if the supporting technologies are, from the get go, implemented to parse it properly.

## 4.1 Other Data Sets

Out of the various motion capture libraries, these two are either new or often utilized in similar problems, be it motion reconstruction, or computer vision.

### LaFAN 1

This relatively new data set has been shot in 2017, but has been released in 2020 along with the SIGGRAPH 2020 paper Robust Motion In-betweening [8]. It has been created by the Ubisoft Montréal's La Forge studio. It contains 496,672 motion frames, which, at the rate of 30 FPS, amount to about 4.6 hours of content. For the purposes of this thesis, however, only certain sequences could be used. While the files are in the BVH format, the accompanying parsing algorithms transform the rotation values into quaternions as opposed to the Euler angles utilized in this thesis. Moreover, as mentioned in Chapter 4.3, the axes are different from the other motion capture data set, and would not be parsed by the implemented BVH data file parser.

### Human 3.6 Million

This data set, commonly abbreviated as H3.6M [10], is considered to be one of the largest human motion capture data sets ever created, with the statistic of containing over 3.6 million individual poses (hence the name) often cited. Its content often involves standing motion as opposed to locomotion (smoking, talking on the phone and such), but also include various walking activities, such as walking the dog. While the walking animations could prove useful to be explored, the dataset could not be obtained for the purpose of this thesis.

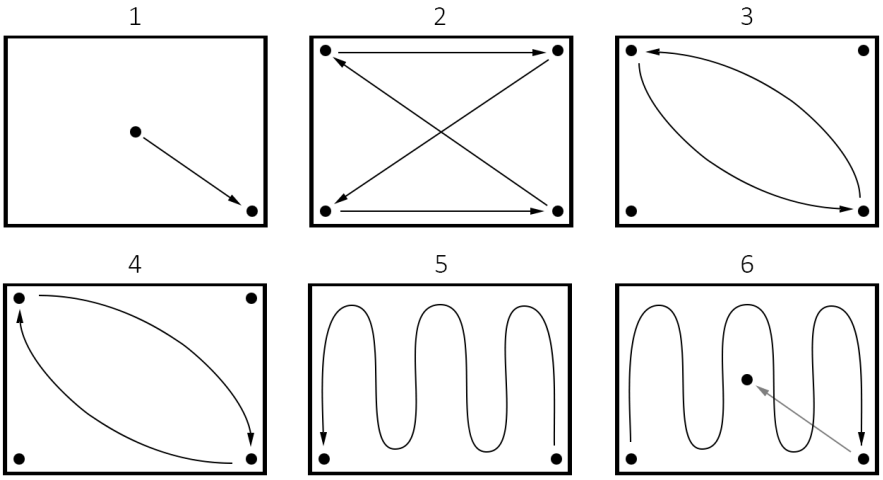


Figure 4.1: The path taken by the actors in each motion capture file of the SAUCE Project motion library, exhibiting various turning radiuses. Taken from the data set itself.

## 4.2 The SAUCE Project Motion Library

SAUCE, standing for Smart Assets for re-Use in Creative Environments [16], was a collaboration between various European academic institutions, including the Brno University of Technology, as well as industry entities such as the Disney Research Studios, based in Switzerland. The goal was to create re-usable assets and invent tools for digital production. Amongst these is the PHS Motion Library, a collection of over an hour's worth of data containing animations of a human character walking with various emotions or actions, including running, pompous sway, tiptoeing or even texting.

The data set contains 38 different motion capture recordings, as well as accompanying videos. Each motion follows the same path, although due to different exhibited paces, the total duration of each motion varies. Each recording session is provided in multiple formats – there is the BVH format utilized in this thesis and described below, recorded at 30 frames per second, but also files in FBX, another motion capture format, with recordings available at rates of both 30 and 120 frames per second. Marker data for the FBX format is also available. Finally, besides files that describe the dataset, a diagram (Figure 4.1) is also provided that details the paths taken by the actor during each session. This one is utilized in this thesis to check the functionality of BVH parsing.

The skeleton that the motion capture data describes consists of 21 joints in total, and each joint boasts six degrees of freedom, including the rotation denoted in Euler angles. While the units for the offset are not defined, it can be estimated that the individual offsets are provided in centimeters. The Euler angles are listed in degrees as opposed to radians. From the data set, the file utilized for parsing and training was a simple recording of a neutral walk, containing 2,106 frames, or a little over a minute in terms of duration. Ultimately, the size of this data set may have negatively affected the results of this model's implementation. However, merging multiple files into a greater data set to train on could conversely produce even worse results, considering the fact that the files contain various gaits and walks, and this disctinction is not accounted for in the feature vector that describes each frame. The model could end up blending these various emotions.

## 4.3 The Biovision Hierarchical Format

Introduced by the company Biovision as an extension of their existing formats to describe motion capture data, the Biovision Hierarchical format is a commonly used file format utilized to define the joint structure of the animated character as well as the motion itself [13]. It consists of two parts: a header section that defines the skeleton as a tree structure, starting with the keyword `HIERARCHY`. The keyword `ROOT` defines the joint which forms the root of the tree, and the other joints are defined recursively. Each joint is defined by an `OFFSET` keyword, which dictates its spatial displacement from its parent joint in a neutral pose (generally zero for the actual root joint), and a `CHANNELS` keyword, which is followed by the definition of the individual degrees of freedom that a given joint has. Finally, completing the recursive nature, the definition of a joint can contain the definitions of other joints, its children in the tree structure, headed by the keyword `JOINT`. Furthermore, an `End Site` keyword can be used to signify the end of a joint series, which only contains the definition of its offset. For an example of the BVH file structure, see Figure 4.2.

The second part of the file describes the motion itself via the values of each channel defined in the `HIERARCHY` section. It is headed by the `MOTION` keyword, followed by a line describing the total amount of frames described by the file, and a line containing the duration of a single frame in seconds. From then onwards, each line consists of a series of values. These values represent the channel values of each joint, with the joints being ordered accordingly to a Preorder traversal of the hierarchical structure (or, alternatively, in the order in which they are defined in the `HIERARCHY` section). For example, in the data set I have chosen to train the model on, each joint has six degrees of freedom. Thus, the first six values correspond to the six degrees of freedom of the root joint, the next six values correspond to the six degrees of freedom of the first child of the root (the spine joint in this case), the next six of the first child joint of the spine and so on. An example of what the `MOTION` part can look like is provided in Figure 4.3.

### Reconstructing the Motion

In order to recover the individual positions and motions of each joint, the data needs to be processed. The process in general is detailed in an article by Meredith and Maddock at the University of Sheffield [13]. For each joint at any given frame, its local transformation $M$ has to be calculated from its rotation and offset. Since scaling is not featured in this format, the order of multiplications $M = TRS$ is simplified to $M = TR$. The compound rotation matrix is calculated from three individual rotation matrices for each axis, their order ideally defined in the order of the channels in the `HIERARCHY` section. As with the example provided in Figure 4.2, the compound rotation matrix would thus be calculated as

$$\mathbf{R} = \mathbf{R}_z\mathbf{R}_x\mathbf{R}_y. \tag{4.1}$$

Once the local transformation for a given joint as well as for every preceding joint in the hierarchy, the joint's global transformation can be computed as their product:

$$\mathbf{M}_G^n = \mathbf{M}_L^0\mathbf{M}_L^1 \dots \mathbf{M}_L^{n-1}\mathbf{M}_L^n, \tag{4.2}$$

Where the index $n = 0$ refers to the root and increments with each joint towards the desired one. Alternatively, the global transformations can be stored for each joint and stand in for the sequence of preceding local transformations when calculating the global transformation of its children. It should be further noted that while animation will most

```
HIERARCHY
ROOT Hips
{
    OFFSET 0 0 0
    CHANNELS 6 Xpos Ypos Zpos Zrot Xrot Yrot
    JOINT Right_Leg
    {
        OFFSET -10 0 0
        CHANNELS 3 Zrot Xrot Yrot
        End Site
        {
            OFFSET 0 -90 0
        }
    }
    JOINT Left_Leg
    {
        OFFSET 10 0 0
        CHANNELS 3 Zrot Xrot Yrot
        End Site
        {
            OFFSET 0 -90 0
        }
    }
}
```

Figure 4.2: An example of a five-joint skeleton defined by a BVH file with arbitrary values.

likely require the global positions of each joint, it is sometimes more useful to find a given joint's displacement and rotation in relation to the root (and its facing direction) instead of globally. Specifically when constructing feature vectors for a machine learning model, it is in due course to avoid global positions, as discussed in Chapter 5.1.

### Irregularities in the Format

Despite boasting a defined file structure, the contents of a given BVH file do not necessarily follow any standard. This includes, but is not limited to, the amount of channels for a given joint, the units in which the values are provided, or the orientations of the axes. A popular guide for BVH files [2] implies in its order of matrix multiplications that the vertical axis is the Y axis, or at least that the order of rotation channels of each joint is dictated by which axis is the vertical one. Both apply for the SAUCE Project data set used in this thesis, but in Ubisoft Montréal's LaFAN1 data set, the axes are swapped around, immediately breaking mechanisms which assume the Y axis to be the vertical one. Likewise, both data sets utilize different units for angles. Finally, the SAUCE Project BVH files feature six channels for every single joint, not only for the root. The guide's author claims, however, that "[they] have never encountered a BVH file that did not have 6 channels for the root object and 3 channels for every other object in the hierarchy". Moreover, due to the lack of any comments within the file itself, it is not well-defined what

```
MOTION
Frames: 3
Frame Time: 0.033333
0 90 0 0.5 0 0 0.01 0 -1.5707 0.01 0 1.5707 0 0 0 0 0 0
0 91 1 0.5 0 0 0.01 0 -1.6707 0.01 0 1.4707 0 0 0 0 0 0
0 92 2 0.5 0 0 0.01 0 -1.7707 0.01 0 1.3707 0 0 0 0 0 0
```

Figure 4.3: An example motion description of the skeleton defined in Figure 4.2 with arbitrary values.

the three additional channels of each joint represent – "position" can refer to their absolute positions within the space, their relative positions to the root joint, their relative positions to their parent joint and so on. Finally, if one uses an on-line application that plays BVH files back as animations[1], they may find that the generated model is far larger than the example clips provided within the application. Such is the case with the SAUCE Project motion capture data, further showcasing the lack of unit standardization across this file format.

[1] An example BVH player: http://lo-th.github.io/olympe/BVH_player.html

# Chapter 5

# Trajectory-guided Locomotion Recurrent Neural Network

As a solution to the problem of generating character animation and locomotion, I have designed and implemented TGLRNN – a Trajectory-guided Locomotion Recurrent Neural Network. This model, trained on a single BVH file, produces motion which can be controlled during runtime in a similar fashion to how one would control a video game character. The model is implemented within an application which also features algorithms to parse the respective BVH file, as well as let the model run freely, visualized in a window, and let the user perform inputs that control the character's motion.

## 5.1 The Discriminative Model

The TGLRNN is a recurrent neural network model consisting of three modules, following the Encoder-Recurrent-Decoder structure presented by Fragkiadaki et al. [5]. First, there is the Encoder module, which transforms the input feature vector into the frame's representation in the latent space. It consists of one non-linear activation function layer. For this implementation, I have selected the element-wise Rectified Linear Unit activation function, defined as

$$\mathrm{ReLU}(x) = \max(0, x). \tag{5.1}$$

The non-linear activation function is inserted between two linear layers, and the Encoder module can thus be defined as

$$\mathcal{E}(\mathbf{x}; \mathbf{P}) = \mathbf{W}_1 \ \mathrm{ReLU}(\mathbf{W}_0 \ \mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1, \tag{5.2}$$

where the parameters $\mathbf{P}$ of the module can be described as $\mathbf{P} = \{\mathbf{W}_0 \in \mathbb{R}^{h_0 \times in}, \mathbf{W}_1 \in \mathbb{R}^{h_1 \times h_0}, \mathbf{b}_0 \in \mathbb{R}^{h_0}, \mathbf{b}_1 \in \mathbb{R}^{h_1}\}$ for an input $\mathbf{x} \in \mathbb{R}^{in}$, feature vector of size $in$, encoder hidden layer of size $h_0$ and latent space of size $h_1$.

Then, there is the Recurrent module. This module consists of two LSTM layers, described in Figure 2.2. Then, the Recurrent module can be described as

$$\mathcal{R}(\mathbf{x}, \mathbf{h}_0, \mathbf{h}_1, \mathbf{c}_0, \mathbf{c}_1; \mathbf{P}) = \mathrm{LSTM}(\mathrm{LSTM}(\mathbf{x}, \mathbf{h}_0, \mathbf{c}_0), \mathbf{h}_1, \mathbf{c}_1), \tag{5.3}$$

with the latent-space input $\mathbf{x} \in \mathbb{R}^{h_1}$ and the hidden states $\mathbf{h}_0 \in \mathbb{R}^{h_1}$, $\mathbf{h}_1 \in \mathbb{R}^{h_1}$ and cell states $\mathbf{c}_0 \in \mathbb{R}^{h_1}$, $\mathbf{c}_1 \in \mathbb{R}^{h_1}$ as the arguments, and the parameters $\mathbf{P}$ described in 2.2, all satisfying the constant latent space vector length of $h_1$.

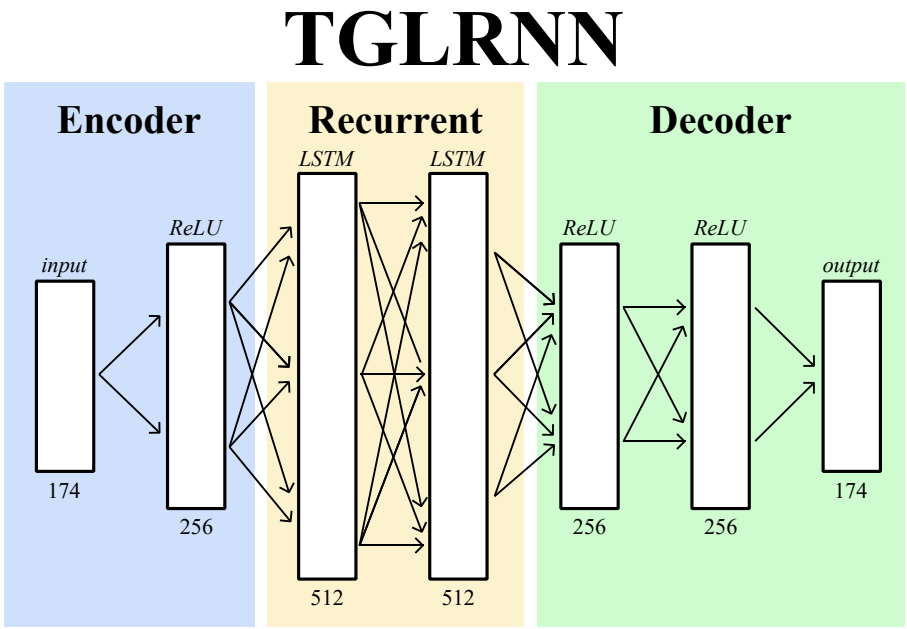


Figure 5.1: A schematic of the TGLRNN's modules and their individual layers. The vector size below each layer indicates its input size, with 174 for the input and output vectors being determined by the character's skeleton as well as the trajectory parameters.

Finally, the Decoder module consists of two non-linear activation function layers and outputs a feature vector that represents the next predicted frame:

$$\mathcal{D}(\mathbf{x};\mathbf{P}) = \mathbf{W}_2\ \mathrm{ReLU}(\mathbf{W}_1\ \mathrm{ReLU}(\mathbf{W}_0\ \mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1) + \mathbf{b}_2 \tag{5.4}$$

The model is parametrized by $\mathbf{P} = \{\mathbf{W}_0 \in \mathbb{R}^{h_0 \times h_1}, \mathbf{W}_1 \in \mathbb{R}^{h_0 \times h_0}, \mathbf{W}_2 \in \mathbb{R}^{out \times h_0}, \mathbf{b}_0 \in \mathbb{R}^{h_0}, \mathbf{b}_1 \in \mathbb{R}^{h_0}, \mathbf{b}_2 \in \mathbb{R}^{out}\}$ with the transformed latent space input $\mathbf{x} \in \mathbb{R}^{h_1}$, latent space size of $h_1$, decoder hidden layer size of $h_0$ and the predicted vector of size $out$. In this model, I have chosen the Decoder's hidden layer size to be the same as the Encoder's, and the output vector to be the same size (and composition) as its input: $out = in$.

Thus, the entire neural network model as visualized in Figure 5.1 can be described using the following function:

$$\mathrm{TGLRNN}(\mathbf{x}, \mathbf{z}; \mathbf{P}) = \mathcal{D}(\mathcal{R}(\mathcal{E}(\mathbf{x}), \mathbf{z})), \tag{5.5}$$

where $\mathbf{z}$ represents the hidden and cell states of the two recurrent layers: $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{c}_0$ and $\mathbf{c}_1$ and $\mathbf{P}$, in turn, the parameters of the individual modules.

The sizes of the individual layers are as follows:

- 174 for the size of the feature vector (based on the skeleton defined in the BVH file)
- 256 for the hidden sizes within the Encoder and Decoder modules
- 512 for the recurrent layers

### The Feature Vector

The feature vector which describes a single frame of motion consists of several values. First, there are the positions of each individual non-root joint relative to the root's position

and transformation. These values are utilized instead of, for example, the joints' global positions, as then, two sequences of the character walking would, while describing the same motion, be described by wildly different feature vectors. Especially during runtime, once there are no boundaries for the free-running character, leaving the coordinates of the boundaries of the motion capture set would result in values that have not been provided to the network during training.

Similar to positions, the velocities of each joint (including the root) are utilized, again in relation to the root's facing direction.

The input feature vector is the same as the output vector. It can be parametrized as $\mathbf{x}_i = \{\mathbf{j}_R, \mathbf{v}_R, \Delta\alpha^0, \beta^0, \gamma^0, \mathbf{v}_R^0, \mathbf{t}\}$, where $\mathbf{j}_R \in \mathbb{R}^{3(j-1)}$ are the individual non-root joint positions in relation to the root's position and rotation, $\mathbf{v}_R \in \mathbb{R}^{3(j-1)}$ are the individual non-root joint velocities in relation to the root's position and rotation, $\Delta\alpha^0 \in \mathbb{R}$ is the difference in the root's rotation around the vertical axis, $\beta^0 \in \mathbb{R}$ and $\gamma^0 \in \mathbb{R}$ are the root's rotation angles around the two non-vertical axes, $\mathbf{v}_R^0 \in \mathbb{R}^3$ is the velocity of the root in relation to its rotation and $\mathbf{t} \in \mathbb{R}^{3T}$ are the positions of trajectory points relative to the root's position and rotation. Spaces are described using $n$, which represents the number of joints in the animated charcater, and $T$, which represents the number of trajectory points.

The root's local rotational transformation, however, is broken down into specific angles. Specifically, while the root's rotation angles around the X and Z axes, $\beta^0$ and $\gamma^0$, are passed into the feature vector without any change, the rotation around the vertical Y axis, $\alpha^0$, is only tracked in its difference to the previous position's rotation, ensuring, again, that the neural network model does not distinguish between the exact same motions simply based on whether the character is, for example, facing North or South-West.

Finally, there is the trajectory. A frame's trajectory consists of the root positions and facing directions in subsampled surrounding frames, covering approximately one second into the past and the future, all relative to the current root's position and facing direction. A trajectory's goal is to link certain current motions to differences in rotation in the future, but it also allows to match user control onto the expected motion.

## 5.2 Data Preprocessing

Before the model can be trained, the provided BVH data needs to be processed and formed into a sequence of feature vectors that the model will be trained on. That, however, requires the knowledge of the hierarchical structure of the character's joints. So first, such structure is created, mimicking the structure defined in the BVH file. Then, for each frame, the channel values are used to form each joint's local transforms $\mathbf{M}_L$. Those enable finding the relative and absolute positions of each joint, as described in Chapter 4.3, by combining the joint's local offset $\mathbf{T}$ and its rotational angles around the three axes, $\mathbf{R}_z$, $\mathbf{R}_x$ and $\mathbf{R}_y$.

$$\mathbf{M}_L = \mathbf{T}\mathbf{R}_z\mathbf{R}_x\mathbf{R}_y \tag{5.6}$$

$$\mathbf{j}_G^n = \mathbf{M}_L^0\mathbf{M}_L^1 \dots \mathbf{M}_L^{n-1}\mathbf{M}_L^n[0,0,0,1] \tag{5.7}$$

represent the process of reconstructing the global position $j$ of a joint indexed $n$, with local transforms of its parent joints with decreasing indices, down to the root with index 0.

The relevant values, however, are the joints' individual positions relative to the root position and rotation. Effectively, this means that one has to remove the root's transformation

matrix from the process of finding the given joint's global transform described in (5.7):

$$\mathbf{j}_R^n = \mathbf{M}_L^1 \mathbf{M}_L^2 \dots \mathbf{M}_L^{n-1} \mathbf{M}_L^n [0, 0, 0, 1] \tag{5.8}$$

(Notice the different index of the first local transform matrix, as the root's local transform $\mathbf{M}_L^0$ is omitted.)

Another two values that can be inferred right from a given frame are the root joint's rotation angles $\beta$ and $\gamma$ around the nonvertical axes, X and Z in this case.

Afterwards, the values of joint velocities as well as the change in the root's facing direction are all inferred from comparing each two subsequent frames. The velocity $\mathbf{v}^n$ of a joint $\mathbf{j}^n$ in a given frame $t$ is found as the difference between the joint's root transforms in the subsequent frames:

$$\mathbf{v}_t^n = \mathbf{j}_{R,t}^n - \mathbf{j}_{R,t-1}^n \tag{5.9}$$

The root's turn $\Delta\alpha$ is simply just the difference in the root's facing directions represented by its rotation around the Y axis $\alpha$ between the two subsequent frames:

$$\Delta\alpha_t = \alpha_t - \alpha_{t-1} \tag{5.10}$$

Finally, the trajectory for each frame is constructed by taking several surrounding frames (with their amount and spacing selected to cover roughly one second of motion forwards and backwards from the current frame), extracting their root positions and finding their transforms to the current root's position and rotation.

$$n \in \mathbb{Z} \cap \langle -T; T \rangle - 0 \tag{5.11}$$

$$\mathbf{t}_t^n = \mathbf{T}\ \mathbf{j}_{G,t-n*S}^0\ \mathbf{R} \tag{5.12}$$

The resulting vector $\mathbf{t}_t$ consists of $2T$ three-dimensional vectors. The value $S$ represents the spacing between the regarded frames (resulting in subsampling). $\mathbf{T}$ is a 3D translation matrix constructed using the non-vertical displacement values of the global root position at frame $t$ and $\mathbf{R}$ is a rotation matrix only involving said root joint's facing direction around the vertical axis, $\alpha$. The values used to construct these matrices and the order of operations effectively perform a reverse transformation, resulting in the displacement of a past or future position of the root joint in the transform of the current frame's root position and direction.

For this implementation, I have chosen to have the application look at eight frames both in the past and in the future, spaced six frames apart from each other.

Holden et al. [9], following the ideas presented in Clavet's Motion Matching [3], also include the past and future root facing directions and velocities in the trajectory. For this model, however, I have found during testing that attaching more features related to the trajectory causes the network to neglect the periodic motion defined by the joints' relative positions, and results in the animated character sliding into a frozen pose.

## 5.3 Training

The training of the model is relatively straightforward, but certain mechanisms can be utilized to improve the model's stability. I have chosen to train it with a simple stochastic gradient descent, updating the model's parameters after sequences of 32 frames. The network is provided with the feature vectors describing each frame in a single BVH motion. The training values can be normalized – the mean value subtracted from each, and then

divided by their variance, resulting in a vector of values centered around zero and varying within similar intervals. However, while the mean subtraction is retained, the division by variances is abstained from, because some features tend to have zero variance. While this should be a red flag and the values could probably be removed from the feature vectors, their presence appears to be dictated by the individual BVH file, and could vary across various files. Since after mean subtraction, these values will be zero, dividing them by their variance (which is also zero) can be skipped. During testing, however, it was found that the higher-varying values made the model generate periodic walking motion more consistently. This is in accord with the work of Holden et al. [9], where it is mentioned that, opposingly, multiplying every non-trajectory feature vector by a coefficient of 0.1, thus diminishing their values in relation to the features containing motion control information, resulted in a more responsive character.

Since the model outputs values defining a frame of animation rather than a probability model, the output can be compared to the expected frame from the ground truth. That is why I have selected the Mean squared error function [19] as the loss metric for training the model:

$$\text{Cost}(\mathbf{X}, \mathbf{P}, \mathbf{z}) = \frac{1}{32} \sum_{n=1}^{32} (\text{TGLRNN}(\mathbf{X}_n, \mathbf{z}; \mathbf{P}) - \mathbf{X}_{n+1})^2 \tag{5.13}$$

$\mathbf{X}$ represents the sequence of 33 feature vectors lifted from the ground truth, $\mathbf{P}$ represents the trained parameters and $\mathbf{z}$ represents the LSTM layers' hidden and cell states (which are set to zero before a sequence is processed).

I utilize the Adam optimizer for training the network. The learning rate is set to 0.0005, and each model is trained for 1000 epochs, which are described below.

### Training Cycles

The training of the model can be divided into (and controlled by the number of) individual epochs. During a single epoch, a total of $4 * \frac{S}{32}$ sequences are selected from the ground truth, where $S$ represents the total number of frames in the ground truth. Each sequence is selected randomly by randomizing an index between 0 and $S - 32 - 1$ (the decrement by one at the end is to ensure that there is a vector of expected values after the last input vector from the input sequence is passed in case of the maximum possible index).

### Training Methods

The aforementioned strategy, known as teacher forcing, which has already been described in Chapter 2.2, is relatively straightforward, but suffers from the fact that once the animation is started and the network's output begin looping back into it as inputs, any errors in the network will begin to deviate from the ground truth and possibly result in a larger, cumulative error, as remarked by Fragkiadaki et al. when describing the training of their model [5]. That is why, optionally, two training strategies can be employed to combat this problem. If either strategy is selected, a probability is calculated for each sequence:

| progress | 0–20 % | 20 %–40 % | 40 %–60 % | 60 %–80 % | 80 %–100 % |
|---|---|---|---|---|---|
| $P$ | 0 | 0.1 | 0.2 | 0.4 | 0.8 |

Table 5.1: Alternate training method probabilities based on training progress.

With the given probability $P$, the selected method will be performed instead of the teacher forcing method of inputting straight from the ground truth. The probabilities are slanted towards the end of the training process, when the model more closely resembles its form during runtime.

One method is adding noise to the input feature vectors, resulting in "corrupted" data and teaching the encoding layer to sanitize the inputs and thus enable the model to fix its own deviations from the ground truth manifold that it may output. I opt to simply add a value generated by a normal distribution with the mean value of zero and standard deviation of one, multiplied by the given feature's variance.

Alternatively, instead of parsing the whole sequence of 32 vectors, the other method sees that only 31 vectors are forwarded through the network, resulting in 31 predicted frames. These 31 frames are then used as inputs again, and it is the networks' predictions with these inputs that are then compared against the ground truth to find the error:

$$\text{Cost}(\mathbf{X}, \mathbf{P}, \mathbf{z}) = \frac{1}{31} \sum_{n=1}^{31} (\text{TGLRNN}(\text{TGLRNN}(\mathbf{X}_n, \mathbf{z}; \mathbf{P}), \mathbf{z}; \mathbf{P}) - \mathbf{X}_{n+2})^2 \tag{5.14}$$

(The notation remains the same as for (5.13).)

I was planning to utilize a simpler form of returning the output frame back to input, simply taking the first frame from the ground truth replacing the other frames from the sequence with the predictions immediately, but this algorithm tended to cause the script to freeze, and while the CUDA platform mentioned in Chapter 6.2 provided limited information as to what had gone wrong, I estimate it was an issue of running out of memory. The nature of the script interacting with the GPU also prevented the interpretation from being stopped via an outside signal.

## 5.4 Runtime

The neural network model generates animation autoregressively – once it has predicted an output vector determining the next pose, it is this vector that will be utilized as the model's input for the next frame. This means, however, that the first feature vector to be fed forward through the network needs to be taken from elsewhere. Or, rather, to properly set up the network's LSTM layers, a sequence is provided instead. For the purposes of this thesis, the sequence is lifted directly from the ground truth, picked via observation of the original BVH file.

As suggested in Chapter 5.2, the transfer from the model's output back to its input is not direct and certain values can be altered to enable for user control. Specifically, the trajectory on the model's input is synthetized by blending the original predicted trajectory with one matching the user input. I have utilized the same function to calculate the mixture of trajectories as did Holden et al. [9], described in Chapter 3.1

### Locomotion

The character's movement across the 2D plane is defined by these specific values from the feature vector: the difference in the root's facing direction $\Delta\alpha$, the Euler angles defining the root's rotations around the other axes – $\beta$ and $\gamma$ – and the root's velocity in its own transform $\mathbf{v}_R^0$.

$$\alpha_t = \alpha_{t-1} + \Delta\alpha_t \tag{5.15}$$

The root velocity, however, needs to be transformed before it can be applied to the global space, using the rotation matrix $\mathbf{R}$ constructed with the Euler angles defining the root orientation $\alpha$, $\beta$ and $\gamma$, resulting in the displacement $\mathbf{v}_G^0$ in the global space.

$$\mathbf{v}_G^0 = \mathbf{R}\mathbf{v}_L^0 \tag{5.16}$$

While not visible in the visual application, it is possible for the model to predict such frames that while the character remains upright, the height of their root begins veering away from its original value of roughly 90 units without any correction, resulting in the character effectively beginning to float or sinking into the floor. I have decided to forcefully correct the root's global height for each frame to enable the measurement of foot skating, a metric determining the model's performance, which considers the foot joints' global positions and velocities.

### Animation

The animation aspect of the predicted motion is defined by the relative positions of the character's joints $\mathbf{j}_L^n$ in relation to the root joint. In fact, visualizing them as they are predicted in the feature vector can produce the walking animation on its own. To recreate the character walking on a plane, then each joint's local transform needs to be transformed into the global transform using the root's position and rotation. For the purposes of the visualization application, however, the joints are only rotated, much like how the root's velocity in the global space is calculated in (5.16) and the root position is only added to their position by the application if necessary.

# Chapter 6

# Application implementation

I have chosen to implement the model and supporting application in the Python 3.8 language, and I utilize the Numpy[1] library across various aspects of the implementation. The central script, `control.py`, handles both training a model as well as letting the model run freely and generate the desired animation.

The usage of the control.py script is detailed in Appendix C.

## 6.1 BVH Parsing

Parsing the BVH file is handled using functions provided in the `bvh_parser.py` script. Outside using the Pickle library that handles saving certain data structures into files for future use, it also utilizes the bvh-python library[2], created by the 20tab product team and distributed under the MIT license. implemented to parse the content of a BVH file. However, it by no means supplements the entire process of creating a hierarchical structure, let alone reconstructing the animation from a file. Specifically, the `Bvh` object is utilized that parses the file and provides methods to access the parsed information, of which I utilize these:

- `Bvh.get_joint_names()`, which lists the names of all joints of the character, starting with the root joint.
- `Bvh.joint_direct_children(name)`, which lists the child joints to the joint identified by `name`.
- `Bvh.joint_offset(name)`, which returns the initial offset values of a given joint identified by `name`.
- `Bvh.joint_channels(name)`, which lists the names of channels (degrees of freedom) that a given joint identified by `name` has.

The `get_joint_names` function is only utilized to get the first value, the name of the root. Then, the `joint_direct_children` function is utilized specifically to recursively build a custom tree structure, defined using the `Node` class. Each node representing a joint in this structure is assigned an offset and has declared space for channels via the aforementioned function, but from then onwards, I no longer utilize the Bvh library.

[1] available at https://numpy.org
[2] available at https://github.com/20tab/bvh-python

The greatest advantage of transferring the `Node` class is the ability to perform a Preorder traversal, implemented in the `Node.PreOrder()` method. Immediately after the tree is formed, channel values can be assigned to each node, taken from the BVH file's `MOTION` section.

Furthermore, when finding the positions of each joint in the root or global transform, recall the matrix multiplication from Chapter 4.3:

$$\mathbf{M}_G^n = \mathbf{M}_L^0 \mathbf{M}_L^1 \dots \mathbf{M}_L^{n-1} \mathbf{M}_L^n \tag{6.1}$$

If we try to find the global transform of a joint whose parent's global transform is characterized as $M_G^n$, we find it as:

$$\mathbf{M}_G^{n+1} = \mathbf{M}_L^0 \mathbf{M}_L^1 \dots \mathbf{M}_L^n \mathbf{M}_L^{n+1} \tag{6.2}$$

Then, from these two equations, we can find:

$$\mathbf{M}_G^{n+1} = \mathbf{M}_G^n \mathbf{M}_L^{n+1} \tag{6.3}$$

The same applies for the root transform, which simply omits the root's local transform from the multiplication sequence.

If we perform the Preorder traversal of the joint hierarchy, we ensure that if we want to find the global and root transform of a given joint, we have already found that of its parent joint and can avoid excess calculations by supplementing it instead of the matrix multiplication sequence.

In the end, a dictionary is created that contains all the relevant values to a single frame, organized with the following keys:

- `"absXYZ"`: each non-root joint's position in the global space
- `"relXYZ"`: each non-root joint's position in the root transform
- `"rootRot"`: the root's rotation in the global space
- `"rootPos"`: the root's position in the global space
- `"velXYZ"`: each non-root joint's velocity in the root transform
- `"rootTurn"`: the root's change in rotation in the global space
- `"rootVel"`: the root's displacement in the current root transform

If an entry contains values of multiple joints, they are sorted in another dictionary, with the keys being their

Finally, there is the trajectory, compiled together in the manner described in Chapter 5.2. Each frame is iterated through, from which other surrounding frames are considered. First, the surrounding subsampled frames in the past are iterated through backwards, and their root's absolute position is extracted, the same then happens for frames in the future, iterated through forwards. If the algorithm looks for a frame that does not exist (either at the start of the animation or at the end), the best root position in that direction (past or future) so far is supplemented instead. Afterwards, the past and future global root positions are transformed into the current frame's root transform as described in (5.12), Chapter 5.2.

The series of root positions in the current frame's root transform is then stored in the frame dictionary:

- `"trajectory"`: the root positions of surrounding subsampled frames in the current frame's root transform

Once the necessary values of each frame are found, there is a pair of functions I have implemented to transform the dictionary into a one-dimensional array of values and back in order to create and then parse the feature vectors for working with the neural network:

- `GetFlatValues(frame, root)`: This function returns the feature vector of a given frame, using the dictionary `frame`. Optional keyword arguments enable printing out a detailed description of the features and their count, making it easier to adapt the other function to any changes in the constituency of the feature vector, and also adding the trajectory point directions to the vector.
- `ParseNewValues(root, vals)`: This function returns a dictionary of values describing a given frame by iterating through `vals`, the frame's feature vector, a one-dimensional array. Optional keyword argument specifies whether trajectory directions are also expected to be a part of the feature vector.

Both functions unfortunately still require the argument `root`, which references the joint hierarchy, represented by a `Node` object. Specifically, its `Node.PreOrder()` method is utilized to find the proper assignment between each joint's channel values and their indices in the feature vector. This means that even during runtime, when these two functions are utilized back and forth, the application will still need to have parsed a BVH file, requiring the ground truth dataset to be loaded during runtime.

Running the `bvh_parser.py` script on its own parses a BVH file of a name defined within the script (`locs/loc_0001.bvh`) and replays it using the visualizing application.

## 6.2 The TGLRNN Model

The entirety of the neural network model is implemented in the provided `tglrnn.py` script. It is implemented using the PyTorch library, which has been developed to simplify implementing machine learning models in the Python 3 language and improve their performance by performing calculations in the considerably faster C++ language. CUDA, a platform and interface to perform computations on a graphics device, is utilized to further improve performance, but it needs to be enabled for the model to function. Checking if the interface is available can be done with a simple script:

```
>>> import torch
>>> torch.cuda.is_available()
True
```

PyTorch provides a special data structure, the `Tensor`, which is a multidimensional matrix of values of a given data type [20]. Its utility lies in the fact, however, that it attaches a computational graph to itself and effectively tracks the matrix's computation history. This becomes useful for computing the gradients of a model and adjusting its parameters during training.

The model, defined as a custom `TGLRNN` class, inherits from PyTorch's `Module` class, which implements most methods regarding training and using a neural network. Defined within this class specifically is the structure of the model, described in Chapter 5.1.

Each layer is implemented as an instance of a PyTorch object. As an example, here is a simple Linear layer, used in `tglrnn.py`, line 63, to define the first linear layer of the Encoder module:

```
self.e1 =
    nn.Linear(in_features=self.frame_features, out_features=HIDDEN_F1)
```

This line defines the model's `e1` object to be an instance of the `torch.nn` (here imported as `nn`) package's `Linear` class, defined to accept vectors of length `frame_features` (provided to the model when instantiated) and outputting hidden space vectors of size `HIDDEN_F1`, which is set to 256.

I have then implemented the `TGLRNN.forward(input)` function, which is called whenever we want to feed a feature vector labeled as `input` through the model. It is within this function where the PyTorch implementation of the layers shines, as to feed the input vector through it, I only need to do this:

```
res_e1 = self.e1(input)
```

This roughly corresponds to line 89 of `tglrnn.py`, albeit the script also reshapes the input vector to match the function's specifications. This line then performs the following calculation:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{6.4}$$

$\mathbf{x}$ is the vector passed in the argument of size 174, $\mathbf{y}$ is the output vector of size 256, saved to the `res_e1` variable, and $\mathbf{W} \in \mathbb{R}^{256\times174}$ and $\mathbf{b} \in \mathbb{R}^{256}$ are the learnable parameters of the layer.

Including the Rectified linear unit function is considerably simpler. Since it operates element-wise, it does not require a predefined size parameter and can be defined simply as:

```
self.e2 = nn.ReLU()
```

Then, it is applied with:

```
res_e2 = self.e2(res_e1)
```

When it comes to the LSTM layers, a remark has to be made that despite them having learnable parameters of their own, I also had to implement the model in such a way to keep track of their hidden and cell states, which are provided alongside the output of calling the LSTM object, but are in turn required on the input again, as the layers do not retain this information themselves. I solve this by turning the hidden and cell states into model variables:

```
self.r1 =
    nn.LSTM(input_size=HIDDEN_F2, hidden_size=HIDDEN_F2,
    num_layers=1, batch_first=False)
self.r1_hc = None
```

Then, the application of the recurrent layer is as follows:

```
res_r1, hc1 = self.r1(res_e3, self.r1_hc)
```

The variable `hc1` can be used to overwrite `TGLRNN.r1_hc` afterwards.

I have implemented two methods which encompass feeding the input vector forward through a model. `GetNextFrame(model,frame)` simply takes the feature vector `frame` and lets the model referenced by the `model` parameter predict a new frame, which the function returns. This function wraps the process with a `torch.no_grad()` flag, though, which prevents the feature vector from accumulating a computational graph (which we do not need outside training). The other function, `PassSeed(model, sequence)` instead has a model referenced by `model` predicts a frame following a series of feature vectors (`sequence`).

Additionally, however, a special model function, `TGLRNN.ResetRecurrent()` is called before the frame is predicted, which resets the hidden and cell states of the LSTM layers. This effectively allows to begin motion generation anew, and ensure that no previous predicted frames affect the new predictions.

```
def ResetRecurrent(self):
    self.r1_hc = None
    self.r2_hc = None
```

## 6.3 Training

I have implemented a single function, `Train(...)`, to handle the training of a single model in any of the proposed fashions. In fact, due to the initializations of certain aspects of the neural network model within this function, it needs to be called even when the model is only utilized to predict new frames, for example for the visualization aspect of the application.

```
Train(model, data, iter, enable_curr=False,
    use_oi=False, model_name=None, load_model=True,
    savetime=False, saveloss=False)
```

The function is provided with a `model` that should be trained, and a series of vectors referenced to via `data` as the ground truth to train on. The amount of iterations is specified via the `iter` parameter, which is set to zero if we only want to prepare the model for generation instead of training it. The optional arguments consist of `enable_curr` and `use_oi` which determine whether the curricular learning involving methods other than teacher forcing should be invoked, and whether it should utilize the "predicted vector back to input" method rather than denoising, respectively. The `model_name` parameter determines the name of the file into which the PyTorch library functions save the model. If this parameter is not set, the script switches to a default name based on which training method is used: `tglrnn_tf` for standard teacher forcing, `tglrnn_dn` for denoising and `tglrnn_oi` for returning predicted frames to input. If a model of that name already exists, setting `load_model` to `True` will load the existing model and train it further. Finally, setting `savetime` or `saveloss` to `True` will save either the training durations or the losses in each epoch to a file named after the model name.

For training, two more PyTorch objects are instantiated:

```
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

After each sequence of vectors is passed through the model and a sequence of predicted frames is generated that can be compared against the ground truth, the error can be computed with the loss function:

```
loss = loss_function(micro_out, micro_exp)
```

In this case, `loss` is an object which retains its computational history. Its value can be accessed via `loss.item()`. However, to perform the backpropagation and update the network parameters, the following two functions need to be called:

```
loss.backward()
optimizer.step()
```

Throughout the training, after each epoch, information about the losses is printed to the standard output, containing both the total error accumulated throughout the epoch, as well as how it is split between the normal teacher forcing method and an alternate one, if it is enabled. These losses can also be exported to a log file if the `saveloss` parameter is present. The target path for the file will be as follows:

```
out/MODEL_NAME_losses.out
```

Additionally, the `savetime` parameter can be set to `True`, and the script will then export the time spent the interpreter has spent on each epoch in seconds to a log file:

```
out/MODEL_NAME_times.out
```

In both cases, `MODEL_NAME` is the name specified with the `model_name` parameter, or default.

## 6.4 Runtime Visualization

Once a model has been trained, the `control.py` script can be additionally used to make it generate character locomotion, and visualize it using a simple user interface. This interface is implemented in the `window.py` script and utilizes the OpenCV library for Python 3[3]. However, for this application to function, OpenCV needs to be able to open a new window, which, for example with the Windows Subsystem for Linux that I have used, can be achieved by running a display window server such as Xserver which provides the application with a window output.

To test if a window is available, the script `window.py` script can be interpreted on its own, whereupon it attempts to create a such window. If it succeeds, it exits immediately.

This script contains the definition of a `DisplayWindow` class, which is instantiated by whichever script requires visualization. The visualization app offers three different views of the animated skeleton, and the current view is tracked by the instance attribute `DisplayWindow.currView`.

The three views, depicted in Figure 6.1, are as follows:

- View 0: Displays front and side view of the moving character, as well as an arrow indicating the character's direction. Movement across the global space is irrelevant to this view.
- View 1: Displays a top-down view of the character's motion, centered over the $[0, 0, 0]$ coordinate.
- View 2: Displays a top-down view of the character's motion, centered on the character. Background lines imitating the floor help indicate how the character moves around the space.

The `W` key iterates through each of these views. The `Q` key closes the window and stops the script that started it.

OpenCV handles visualization by representing the picture as a multi-dimensional array of floating point values and providing several methods that change these values based on what shape we want to visualize, and this array is one of the class instance's attributes.

The `DisplayWindow` class provides the following method, which renders an animation:

```
PutLines(self, lines, rootRot, rootPos, trajectory, arrows=[])
```

[3] Available at https://pypi.org/project/opencv-python/

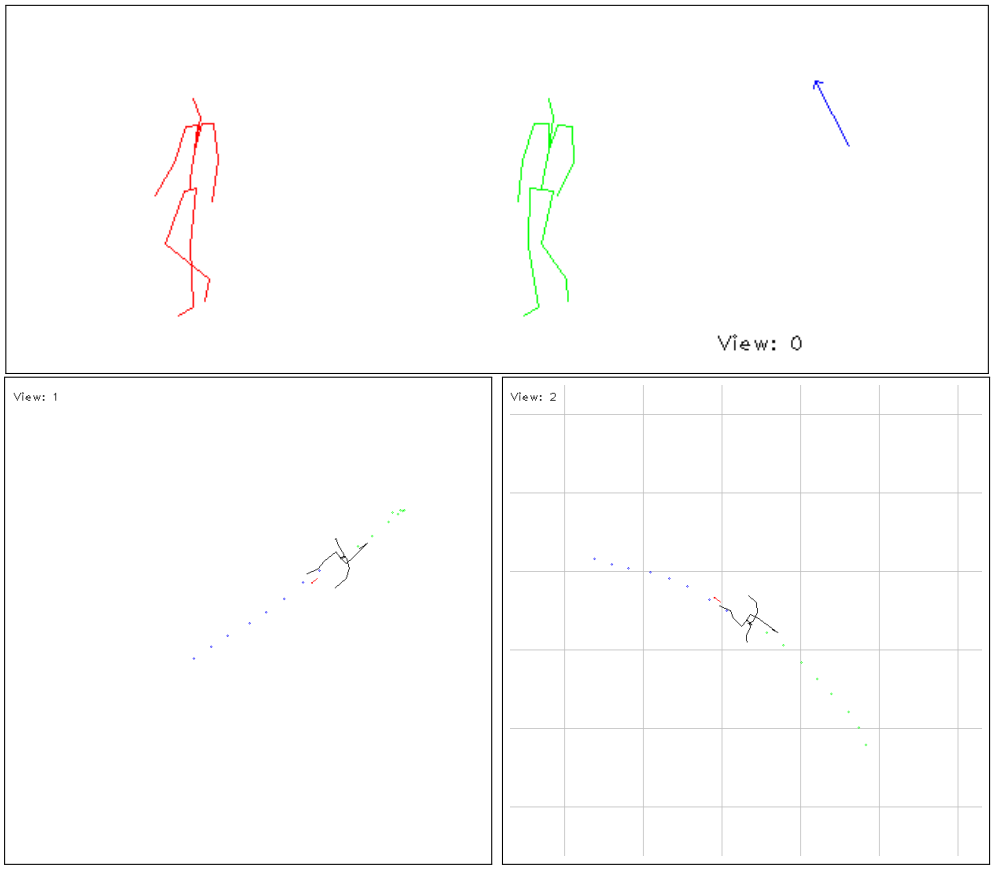


Figure 6.1: The three views (edited) of the generated motion provided by the visualization application. View 0 (top) depicts the animation from two views, each centered on the character. View 1 (bottom left) depicts the character's trajectory and locomotion from above, centered over the center of the global space. View 2 (bottom right) is instead centered over the character, but represents locomotion by representing the static floor via squares beneath the character.

In this case, `rootRot` and `rootPos` represent the character's root joint rotation and position in the global space, respectively, and `trajectory` represents the points making up the frame's current trajectory. However, `lines` is a special object that requires preprocessing on the caller's side. Specifically, it is expected to be an array of pairs, where each pair is the start and end point of a bone in root transform, but already rotated by the root's rotation in global space. Finally, the `arrows` argument provides an array for rotations expressed by three Euler angles, which are then represented by a small arrow circling the visualized character.

If character animation from a model is visualized, its trajectory can be controlled via keyboard inputs. The trajectory is blended using the function described in (3.1) in Chapter 5.4, which is implemented as a `BlendTrajectories` in the `bvh_parser.py` file. The parameter $\tau$ is left at the value of 2. The user control trajectory is generated by either the `GetStraightTrajectory()` or the `GetCircleTrajectory(radius, cwise)` functions. The former always returns the same trajectory, while the latter returns a trajectory that would see the character trace a circle of a given radius. The flag `cwise` determines whether the character is turning left (`False`) or right (`True`).

The user control trajectory is thus selected from the following array of values:

$$80 \mid 160 \mid 320 \mid \mathbf{0} \mid -320 \mid -160 \mid -80$$

Table 6.1: The turn radiuses which can be generated by user input. The default radius of 0 instead swaps out the circular trajectory for a straight line. Positive values correspond to turning left, negative to turning right.

The default trajectory is straight ahead, but the array can be iterated through using the arrow keys. Pressing the Left Arrow key selects a sharper turn to the left (the radius at a lower index than the current one, down to the radius of 80), pressing the Right Arrow key instead selects a sharper turn to the right (the radius at a higher index, up to $-80$). Additionally, for quick turns, the `A` and `D` keys immediately set the turning radius to 80 and $-80$, respectively.

# Chapter 7

# Experiments And Discussion

To measure the performance and quality of the individual neural network models, I have decided to, on top of visual addressing of the result, measure the following values:

- the time taken to train each model
- the progressive loss throughout the training process
- foot skating: the degree to which feet move in the global space despite being supposed to be planted on the ground
- angular error: the degree to which the root's turn does not correspond to the projected trajectory
- NPSS – Normalized Power Spectrum Similarity: the degree to which the powers of various predicted feature values as signals resemble those of the ground truth, as presented in Gopalakrishnan et al. [6]

The three models have only been trained once, and the value of NPSS is also calculated for each model only once. Both foot skating and angular errors are calculated on a prediction sequence of 2400 frames following a random seed from the ground truth, with the direction switched every 800 frames. These test runs have been conducted five times for each model.

Due to the how I have implemented the parsing of the BVH files, however, it is impossible to compare the TGLRNN to other existing solutions. Thus, I have decided to mostly compare three individual TGLRNN models against each other, differing in the training strategies presented in Chapter 5.3, thus providing input on the advantages and disadvantages of these methods.

## 7.1 Training Results

After each epoch of the training processed is completed, three values are saved into an output file: the amount of sequences processed using teacher forcing, the amount of sequences processed using a different strategy, and the total time spent processing the epoch. The `perf_counter()` function from Python's `time` library is used to calculate delays.

Regarding the time taken to train the individual models, the following has been found on a single session of training each model for 1000 epochs: not using any strategy to combat teacher forcing results in about an hour's worth of training the model on an Nvidia GeForce GTX 1060 graphics card. Utilizing the denoising has such little impact on the computational

| Model type | Total time | Avg. epoch time | Strategy slowdown |
|---|---|---|---|
| TF | 1.00 h | 3.59 s | – |
| DN | 0.99 h | 3.55 s | −2.5 % |
| OI | 1.19 h | 4.30 s | 60.7 % |

Table 7.1: Performance statistics for training each model: the total time taken, the average time per epoch, and how faster or slower the training is if an alternate strategy is used

time, its training session has actually ended up being faster by 2.5 %. Adding the extra computation in returning the output values back to input, however, is roughly 61 % slower. These values were found by first computing the average time spent resolving a sequence using teacher forcing before the other strategy is introduced, then subtracting their expected contribution to the delays in epochs involving the other strategy, and finding the time spent processing a sequence with the other strategy from the remainder.

I have decided to track the progression of losses throughout the training process mostly just to compare how the losses differ with the introductions of different learning strategies. An interesting find is that the denoising strategy does not actually demonstrate any improvement in terms of learning to deal with the added noise, and the total loss per epoch only increases as the chance of adding noise grows. On the other hand, letting the network receive its own predictions on input demonstrates convergence towards the teacher forcing loss values. The progression of losses and how introducing strategies to the training process affects the training itself is visualized in Figure 7.1.

## 7.2 Motion Prediction

Besides the "sinking into the floor" issue described at the end of Chapter 5.4, several remarks can also be made about the model's prediction quality based on observation in the visualization app.

First and foremost, the network appears to succeed in maintaining a periodic motion. It can still slide into a frozen pose, although this is mostly restricted to changes in trajectory. Where it fails, however, is maintaining movement in a given direction. Specifically, movement tends to come with the artifact of spinning on the spot every few frames, resulting in diverging away from the predicted straight trajectory despite appearing to steer in the desired direction otherwise. This artifact remains for various directions.

The effect of the trajectory used as a guiding tool to keep the character moving can be observed, however, in those cases when the character gets stuck in a single pose. Changing up the trajectory at that point can snap the character back into periodic motion, despite the prediction loop having started many frames ago.

### Foot Skating

Using the phenomenon known as foot skating, when a foot joint slides across the floor despite being supposed to remain rooted on the spot, is not unknown in the realm of motion prediction [22]. I have attempted to implement it in various ways. For each, the crucial task is to identify the foot's height above the floor. First, the lowest five heights of each foot joint were considered and averaged. However, due to the model possibly fluctuating in terms of feet height even when a foot is supposed to be on the floor, I have instead decided to

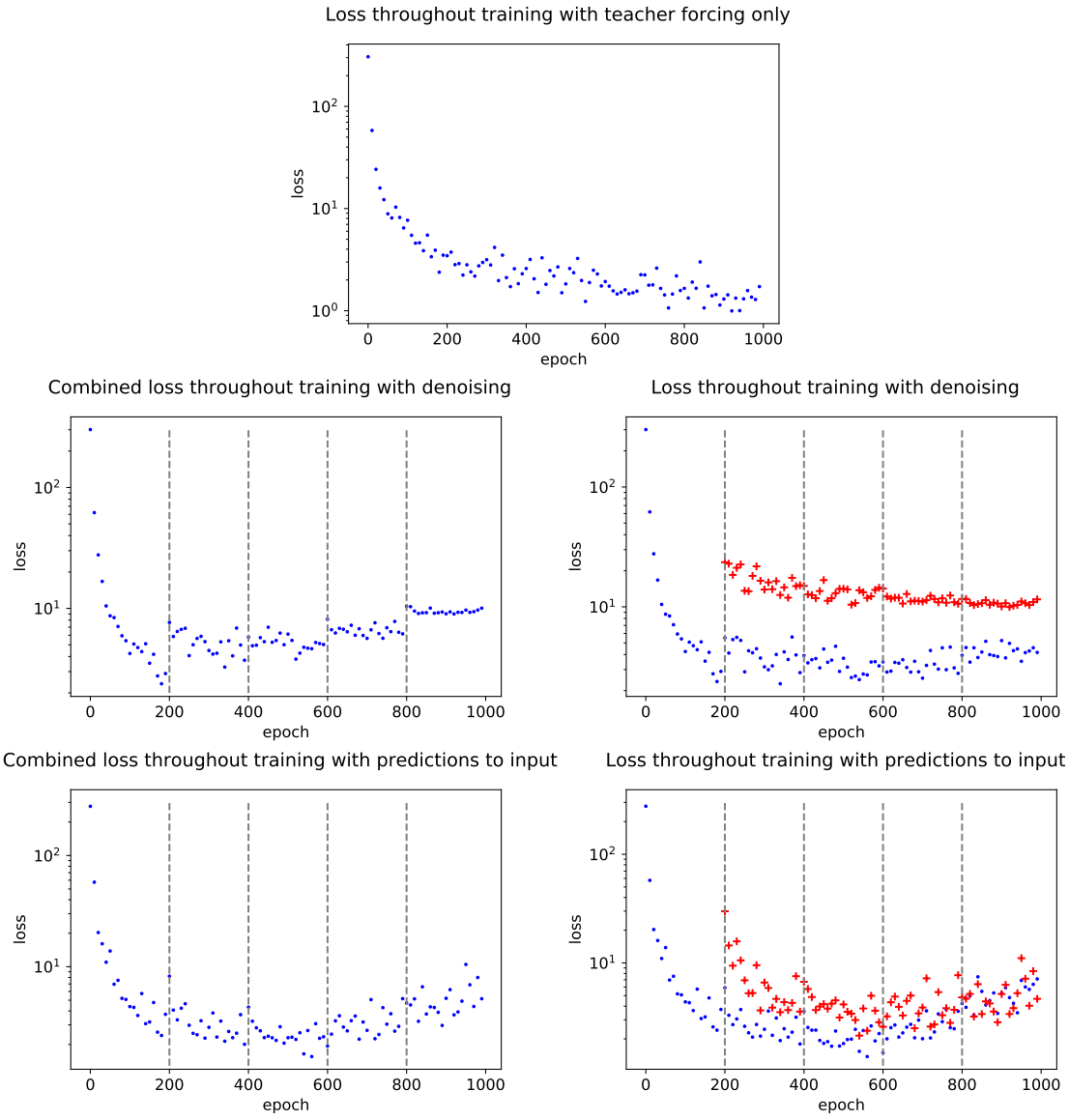


Figure 7.1: Subsampled progression of losses throughout the training of each TGLRNN model using teacher forcing only (top), denoising (middle) and predictions to input (bottom) strategies, visualized on a logarithmic scale. Left column represents total loss per epoch, right column in turn average loss per teacher forcing (blue) or different strategy (red). Vertical lines indicate increase of different strategy probability.

| Model type | Left foot score | Right foot score |
|---|---|---|
| Ground truth | 0.031 | 0.030 |
| TF | 0.214 | 0.251 |
| DN | 0.246 | 0.223 |
| **OI** | **0.238** | **0.228** |

Table 7.2: Foot skating penalty scores calculated using (7.1) of the ground truth sequence and the three TGLRNN models: trained with teacher forcing only (TF), denoising (DN) and predictions to input (OI).

find the minimum height for smaller intervals that cover the entire measured sequence, and get a minimum for a given frame by interpolating between the closest two. Unfortunately, even that proved ineffective (mostly by the ground truth reporting higher proponence to foot skating than the predicted motion). Finally, I have implemented an algorithm which calculates the height for each step independently, delimiting them by finding when the foot crosses a threshold computed as the mean foot height across the sequence.

When applicable, the foot skating penalty $p$ for a given frame is then calculated as follows:

$$p = |\Delta \mathbf{x}| * e^{\sigma * h} \tag{7.1}$$

The $|\Delta \mathbf{x}|$ represents the distance of displacement of the joint, $\sigma$ is a negative parameter controlling the degree of penalization which I have set to $-3$, and $h$ is the joint's height above the floor. The function does penalize displacement even if the foot is being lifted or is about to be planted on the ground, but since this calculation is performed on the ground truth sequence as well, what really matters in the end is how the models compare to the foot skating score of the ground truth sequence.

From the average foot skating error presented in Table 7.2, we can see that all models suffer from this phenomenon to a similar degree, although the prediction to input strategy appears to have the lowest error scores on average.

### Angular Error

The metric referred to as Angular error reflects how well different predictions' root turns correspond to the trajectory the character is supposed to follow. I have decided to calculate it by finding the angle by which the root should be turning. The angle is found by replacing the trajectory in a given frame by an arc, its shape determined by the character's position and the furtherst future point in the trajectory and then, it was expected to find the matching angle based on the arc's central angle, divided by how many frames the last trajectory point is away from the current frame, as seen in Figure 7.2

While looking for an expression of this desired estimated angle, however, I have been found that it is simply a fraction of the angle to which the target trajectory point is offset to the side.

$$\gamma = \frac{2 * \arctan\left(\frac{\Delta x}{\Delta z}\right)}{N} \tag{7.2}$$

The estimated angle $\gamma$ is equal to the angular result of the inverse tangential function of the displacement $[\Delta x, \Delta z]$ of the furthest trajectory point in root transform, divided by $N$, the amount of frames. This amount of frames, however, is not the length of the trajectory

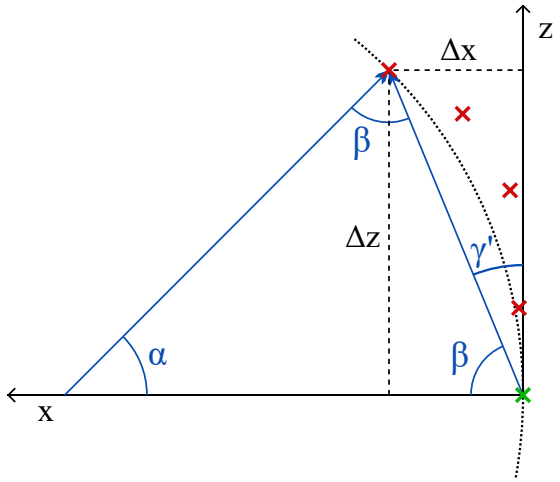


Figure 7.2: A visualization of how replacing the future trajectory (red) with an arc connecting the current position (green) to the furthest point helps find the angle $\gamma'$, which is divided by the amount of frames between the current frame and the furthest one to find the desired angle $\gamma$.

| Model type | Angular error |
|---|---|
| TF | 1.012 |
| **DN** | **0.862** |
| OI | 1.053 |

Table 7.3: Average angular errors calcuated using (7.2) of the three TGLRNN models: trained with teacher forcing only (TF), denoising (DN) and predictions to input (OI).

in the future, however, because the trajectory is expected to be calculated from subsampled frames.

### Normalized Power Spectrum Similarity

Introduced by Gopalakrishnan et al. [6], the NPSS – Normalized Power Spectrum Similarity – is a metric that evaluates long-term motion. It addresses the frequencies that characterize the channel values of various character joints. Their paper lists slow walking as an example of how the NPSS metric outshines the Mean square error loss in comparison – if the motion is exactly the same, but misaligned, the MSE will report high error values, whereas the NPSS will only at most penalize it as a phase shift.

Effectively, the metric compares several sequences from the ground truth and their corresponding predictions. For each feature in each sequence, its squared magnitude spectrum is calculated, normalized with regard to the frequencies. The difference is calculated between the ground truth spectrum and predicted sequence's spectrum, and a power weighted average of these differences across all features and sequences yields the desired scalar evaluation metric. The lower the score is, the more similar the motion is.

| Model type | NPSS score |
|---|---|
| **TF** | **0.549** |
| DN | 0.639 |
| OI | 0.563 |

Table 7.4: The Normalized Power Spectrum Similarity scores of predicted sequences compared to sequences from the ground truth of three TGLRNN models: trained with teacher forcing only (TF), denoising (DN) and predictions to input (OI).

The results, presented in Table 7.4, show that the most similar motions are, in fact, predicted only using the teacher forcing strategy, although returning predicted frames to input is not too far behind. The denoising strategy, in turn, reports a worse score than the other two.

## 7.3 Discussion

The TGLRNN models predict periodic motion to a degree of success, and there is demonstrable influence of the trajectory changing according to user control to the resulting locomotion.

The individual models report mixed scores in various metrics, ensuring that no strategy comes off as the optimal one, or the worst option. Applying noise to input during training appears to yield the worst performance in terms of losses as well as the NPSS evaluation, but outperforms the other two models with the model's ability to produce motion that follows a given direction. Combatting teacher forcing with returning predicted frames back to network input, in turn, significantly slows down the training process, and yields only marginal improvements in the foot skating metric. Ultimately, it appears that not employing any additional strategy for this task at the very least saves the extra time spent on returning predictions back to input without sacrificing too much quality during runtime.

Visually, however, the model does not appear to hold up to other state-of-the-art models, which have long emarked on solving more complex problems involving motion generation.

I believe that the greatest issue holding the model back is the BVH parsing itself. The way it has been implemented limits scaleability outside the SAUCE Project motion capture data set, and requires the ground truth to be present even during runtime. The model's issues that may arrive with following a user-controlled trajectory may come with the limited size of the data set. Future work with the TGLRNN model would likely require a step back and reworking the parsing and feature vector generation, perhaps even swapping the Euler angles for a different unit, for example quaternions.

# Chapter 8

# Conclusion

I have studied various methods of motion generation and implemented a recurrent neural network designed to produce human locomotion with parameters that allow the user to control the character's motion. I have implemented this model as part of an application that allows to train a model on a single BVH file containing motion capture data, and visualize the process of the model repeatedly predicting new and new frames of an animation while letting the user input directions to alter the character's motion.

I have experimented with various methods of training the individual models to find see if there exists an optimal solution to the issue of an autoregressive model accumulating errors by receiving its own predictions on input instead of values from the ground truth. I have found that while there are measurable differences between the performance of some of these strategies (Table 7.1 for training times, Figure 7.1 for the convergences of losses), most of the resulting models manage to fare equally well in terms of predicting motion, even when subjected to metrics other than observation.

The finished result abridges the gap between unstructured motion capture data and a responsive character in motion. Artifacts still appear in the finished result, but their presence only offers more insight into the problematic. For example, while every now and then the character does slip into a frozen pose, a possible pitfall with recurrent models generating over a longer period of time, it can get back on track via user input and its change to the character's trajectory, showing that the trajectory can work as a set of control features that maintain the periodic motion over prolonged periods of times.

Ultimately, one can find better-looking solutions regarding this problem, no further than mentioned in this thesis, no less. However, alongside the various findings during experimentation, I consider the fact that one can complete a training process and see the difference with their own eyes a success, wrapping this thesis up as a form of insight into the complex world of motion generation.

# Bibliography


[1] Bengio, S., Vinyals, O., Jaitly, N. and Shazeer, N. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In: Cortes, C., Lee, D. D., Sugiyama, M. and Garnett, R., ed. *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. Cambridge, MA, USA: MIT Press, 2015, p. 1171–1179. NIPS'15. DOI: 10.5555/2969239.

[2] *Biovision BVH* [online]. 1999 [cit. 2021-05-01]. Available at: https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html.

[3] Clavet, S. *Motion Matching and The Road to Next-Gen Animation* [Presentation at a conference]. March 2016. Game Developers Conference 2016. Available at: https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road.

[4] Donahue, J., Hendricks, L. A., Rohrbach, M., Venugopalan, S., Guadarrama, S. et al. Long-Term Recurrent Convolutional Networks for Visual Recognition and Description. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1st ed. 2017, vol. 39, no. 4, p. 677–691. DOI: 10.1109/TPAMI.2016.2599174.

[5] Fragkiadaki, K., Levine, S., Felsen, P. and Malik, J. Recurrent Network Models for Human Dynamics. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec 2015, p. 4346–4354. DOI: 10.1109/ICCV.2015.494. ISSN 2380-7504. Available at: https://doi.ieeecomputersociety.org/10.1109/ICCV.2015.494.

[6] Gopalakrishnan, A., Mali, A., Kifer, D., Giles, L. and Ororbia, A. G. A Neural Temporal Model for Human Motion Prediction. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[7] Goyal, A., Lamb, A., Zhang, Y., Zhang, S., Courville, A. et al. Professor Forcing: A New Algorithm for Training Recurrent Networks. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 4608–4616. NIPS'16. ISBN 9781510838819.

[8] Harvey, F. G., Yurick, M., Nowrouzezahrai, D. and Pal, C. Robust Motion In-Betweening. ACM. 2020, vol. 39, no. 4.

[9] Holden, D., Komura, T. and Saito, J. Phase-Functioned Neural Networks for Character Control. *ACM Trans. Graph.* 1st ed. New York, NY, USA: Association for

Computing Machinery. july 2017, vol. 36, no. 4. DOI: 10.1145/3072959.3073663. ISSN 0730-0301. Available at: https://doi.org/10.1145/3072959.3073663.

[10] Ionescu, C., Papava, D., Olaru, V. and Sminchisescu, C. Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. IEEE Computer Society. jul 2014, vol. 36, no. 7, p. 1325–1339.

[11] Kulkarni, T. D., Whitney, W. F., Kohli, P. and Tenenbaum, J. Deep Convolutional Inverse Graphics Network. In: Cortes, C., Lawrence, N., Lee, D., Sugiyama, M. and Garnett, R., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2015, vol. 28. Available at: https://proceedings.neurips.cc/paper/2015/file/ced556cd9f9c0c8315cfbe0744a3baf0-Paper.pdf.

[12] Memisevic, R. Learning to Relate Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2013, vol. 35, no. 8, p. 1829–1846. DOI: 10.1109/TPAMI.2013.53.

[13] Meredith, M. and Maddock, S. Motion Capture File Formats Explained. *Production*. january 2001.

[14] Nogueira, P. Motion Capture Fundamentals A Critical and Comparative Analysis on Real-World Applications. In: Oliveira, E., David, G. and Sousa, A. A., ed. *4th International Conference on Information Society and Technology* [online]. 1st ed., 1. Porto: FEUP, January 2012, p. 303–314 [cit. 2021-05-02]. DOI: 10.24840/978-972-752-141-8. Available at: https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_7.pdf.

[15] Olah, C. *Understanding LSTM Networks*, 27. august 2015. [cit. 2021-05-02].

[16] SAUCE Project, F. B.-W. for the. *PHS Motion capture data*. 2019. Available at: https://animationsinstitut.de/en/phs-ml.

[17] Starke, S., Zhao, Y., Komura, T. and Zaman, K. Local Motion Phases for Learning Multi-Contact Character Movements. *ACM Trans. Graph.* 1st ed. New York, NY, USA: Association for Computing Machinery. july 2020, vol. 39, no. 4. DOI: 10.1145/3386569.3392450. ISSN 0730-0301. Available at: https://doi.org/10.1145/3386569.3392450.

[18] Torch Contributors. *LSTM* [online]. 2019. 2021 [cit. 2021-05-02]. Available at: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html#torch.nn.LSTM.

[19] Torch Contributors. *MSELoss*. 2019 [cit. 2021-05-11]. Available at: https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html.

[20] Torch Contributors. *Torch.Tensor*. 2019 [cit. 2021-05-15]. Available at: https://pytorch.org/docs/stable/tensors.html.

[21] Williams, R. J. and Zipser, D. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*. 1st ed. 1989, vol. 1, no. 2, p. 270–280. DOI: 10.1162/neco.1989.1.2.270.

[22] Zhang, H., Starke, S., Komura, T. and Saito, J. Mode-Adaptive Neural Networks for Quadruped Motion Control. *ACM Trans. Graph.* 1st ed. New York, NY, USA: Association for Computing Machinery. july 2018, vol. 37, no. 4. DOI: 10.1145/3197517.3201366. ISSN 0730-0301. Available at: https://doi.org/10.1145/3197517.3201366.

# Appendix A

# Downloading the Data Set

Even though the data set is shared under the Attribution-NonCommercial-ShareAlike Creative Commons license, the Terms of Use that need to be agreed with in order to download the data set prohibits further passing of the data from the set, which is why none of the motion capture data is present within this thesis.

If a user wishes to obtain the motion capture data, the download is, at time of writing, publicly accessible nevertheless. As the data has been created for the SAUCE Project, it can be traced from the project's website at https://www.sauceproject.eu. From there, under the 'Downloads' tab, one can find a reference to the PHS Motion Library, even referred to as "available to download". This reference takes the user to the website of the Animations Institut of Filmakademie Baden-Wärtenburg. Specifically, the user is taken directly to a page detailing the institute's various contributions to the project, including the motion library, which can be downloaded after the user has agreed to the restrictive Terms of Use.

Ideally, for this thesis, the motion capture data in the BVH format should be placed in the empty `locs` folder, as the default file names are `locs/loc_0001.bvh`.

# Appendix B

# Application Requirements

A Python 3 language interpreter is required to run the various scripts provided with this thesis. While most of the required libraries and packages usually come with general Python 3 installations, there are some that may require manual installation for the application to work:

- bvh-parser: The library utilized to parse the contents of a BVH file and create data structures to generate feature vectors with. Available at https://github.com/20tab/bvh-python.
- Numpy: A package for scientific computations. Can be installed via `pip install numpy`.
- PyTorch: A framework providing simple and efficient solutions for implementing machine learning models. A guide to installation based on the target system's disposition is available at https://pytorch.org/get-started/locally/.
- CUDA: An interface for computational acceleration on a graphics device, distributed by Nvidia. Installation based on the target system is available at https://developer.nvidia.com/cuda-downloads.
- OpenCV: A computer vision library utilized to visualize the motion. Also requires a window for output, for example an Xserver. Can be installed via `pip install opencv-python`.

The accesibility of the CUDA platform can be checked with:

```
>>> import torch
>>> torch.cuda.is_available()
True
```

To see if there is a window available for OpenCV, running the `window.py` script on its own will attempt to connect to one. If it can find any, the script finishes immediately.

Finally, a BVH file is required to train the model and visualize its runtime. See Appendix A to see why its absent from the provided files and what can be done to fix that.

# Appendix C

# Running the Main Script

All interactions with the implementation of the TGLRNN are done via the `control.py` script. Here is an overview of the script's arguments.

- `mode`: A mandatory positional argument which determines whether the script trains a new TGLRNN model (`0`), or loads an existing one for motion generation (`1`).
- `-name`: This argument specifies the name of the TGLRNN model. If no name is specified, a default one is supplemented instead, generated from the training strategy.
- `-f`, `-filename`: The name of the BVH file for the model to be trained on (in case of training) or that a selected model has been trained on (in case of generation).
- (Training) `-overwrite`: Forces the script to avoid loading an existing model of the same name and instead to create a new one, overwriting the old one in the end.
- (Training) `-type`: Determines what kind of training strategy should be used for this model – teacher forcing (`0`), denoising (`1`) or returning predictions to model input (`2`). Default value is `0`.
- (Training) `-i`, `-iterations`: The amount of epochs the training lasts for. Default value is 200.
- (Training) `-s, -stats`: Determines if the statistics regarding the training time and losses should be printed to an output file in the end.
- (Generation) `-run-tests`: Enabling this prevents the script from opening a visualization window, and instead runs a series of tests on the provided model.

For example, if one then wants to train a new model called "mymodel" using the denoising method for 1000 epochs on the `locs/loc_0022.bvh` file which contains the mocap data for a rather specific type of walk, the command to do so would be as follows:

```
python3.8 control.py 0 --name=mymodel --filename=locs/loc_0022.bvh
    -type=1 --iterations=1000 --overwrite
```

Then, letting the model freely generate frames of motion would be done with the following command:

```
python3.8 control.py 1 --name=mymodel --filename=locs/loc_0022.bvh
```

# Appendix D

# Video

A video containing a brief description of the problem, the methods utilized to solve it as well as a visualization of the results is available at https://youtu.be/6LH4zf1HyPE.