Reducing Animator Keyframes

Daniel Holden

Doctor of Philosophy

Institute of Perception, Action and Behaviour

School of Informatics

University of Edinburgh

2017

Abstract

The aim of this doctoral thesis is to present a body of work aimed at reducing the time spent by animators manually constructing keyframed animation. To this end we present a number of state of the art machine learning techniques applied to the domain of character animation.

Data-driven tools for the synthesis and production of character animation have a good track record of success. In particular, they have been adopted thoroughly in the games industry as they allow designers as well as animators to simply specify the high-level descriptions of the animations to be created, and the rest is produced automatically. Even so, these techniques have not been thoroughly adopted in the film industry in the production of keyframe based animation [Planet, 2012]. Due to this, the cost of producing high quality keyframed animation remains very high, and the time of professional animators is increasingly precious.

We present our work in four main chapters. We first tackle the key problem in the adoption of data-driven tools for key framed animation - a problem called the inversion of the *rig function*. Secondly, we show the construction of a new tool for data-driven character animation called the *motion manifold* - a representation of motion constructed using deep learning that has a number of properties useful for animation research. Thirdly, we show how the motion manifold can be extended as a general tool for performing data-driven animation synthesis and editing. Finally, we show how these techniques developed for keyframed animation can also be adapted to advance the state of the art in the games industry.

Acknowledgements

I would like to thank my supervisor Taku Komura for his help, support and guidance throughout my PhD. I would also like to thank Jun Saito for all of his support and help throughout my studies. Finally I would like to thank all of the other academics, reviewers, and researchers which I have met during my studies and who have selflessly shared their ideas, helped me learn new concepts, and given their time extremely generously.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*author*)

Table of Contents

Chapter 1

Introduction

In the production of animated films, high quality character animation is one of the most important aspects of the production. In animated films, animators play the role of the actors - producing movements for the virtual characters. Just like poor acting, poor animation can disengage the audience and make the intentions and feelings of the characters unclear, while good animation can make the audience empathise with the characters, tell the story, and portray the narrative.

It is for these reasons that, in the production of animated films, the time of skilled animators is incredibly important and precious. Each scene needs to be carefully crafted and developed by hand in a process called *keyframing*, where by the animator poses the character at important times in the scene and the in between poses are computed using interpolation. This process can be incredibly time-consuming as the character may need to be manually posed over a long period of time at a high frame rate. An animator will often specify several poses *per second* for the final production. For a feature length film of several hours this is an incredible amount of manual work.

Many of the largest animation studios have supported this process, as manually keyframing and extensive artist involvement ensures the highest quality stylistic animation is always produced [Planet, 2012]. Yet this philosophy is also very slow and expensive, as it requires the studio to employ many highly trained animators. Due to this, for smaller animation studios, manually keyframing all animation might simply be impossible and interest is starting to grow in research that can reduce the total time spent manually producing keyframed animation.

Previously, data-driven research into automatic animation synthesis has provided extremely successful in the fields of robotics and game development, with many existing techniques having wide adoption. Yet, much of this research has not seen direct deployment in keyframed animation studios, even in those smaller studios who cannot afford laboriously hand crafted keyframed animation in all of their production.

Additionally, many of the tools currently in use have issues with scalability and complexity. They often have poor computational complexity making them difficult to scale to large data sets of motion. Others may require too many manual processes to be performed by artists and technical developers which make them complex and difficult to maintain and scale. How these issues can be overcome using new machine learning techniques is another area of interest in the field.

This thesis therefore has two aims. Firstly, to investigate the obstacles in deploying state of the art animation research technology in keyframed animation environments - developing new tools which can overcome these obstacles. And secondly, to use the recent advances in machine learning to develop new, powerful data-driven animation tools that reduce the total time taken keyframing by animators. This thesis is structured as follows:

The initial third of this thesis is dedicated to discovering and overcoming the main issues with using existing animation techniques in keyframed animation environments (see Chapter 2). To do this we identify the main thing stopping the use of data-driven tools in keyframed animation - a concept called the *inverse rig function*. We then present several techniques that can be used to solve this problem and demonstrate their practicality by applying them to many modern motion synthesis and editing techniques, finally comparing them to the previous state of the art.

The next third of this thesis is about developing new data-driven techniques for the automatic production of character animation (see Chapters 3 and 4). For this we use modern machine learning tools which have seen great success in other fields - namely deep learning and convolutional neural networks. We build tools based around something called the *motion manifold* (see Chapter 3) and, using this as a base, we develop a number of tools under the same framework which can perform many of the common tasks required by animation researchers and animators (see Chapter 4). We show the applications of these tools and present their advantages over the state of the art. Finally, we show how our techniques have been developed in a way which is particularly applicable to keyframed animation environments.

The last third of this thesis covers how the tools developed for keyframed animation can also be used in game development - further advancing the state of the art in that field (see Chapter 5). We present a new neural network structure called the phase-functioned neural network which can produce a character controller that is extremely compact and fast to compute, yet incredibly expressive with a huge capacity for data. Our final controller can react well to complex situations and produces high quality motion in a number of difficult situations such as climbing and jumping over rough terrain. As this controller is entirely automatic and data driven it can also reduce the time required by technical developers and animators to produce character controllers in game studios.

Chapter 2

Literature Review

2.1 Background

Since the advent of virtually animated characters for video games and films researchers in academia and the industry have been interested in finding ways to ease the process of manual keyframing. Most of this work has been in some way directed toward the automatic generation of animation from high level goals. This includes, for example, methods which allow an animator to intuitively specify the edits they wish to apply to an animation, or methods which generate animation with some specific property such as locomotion with a desired speed or turning angle.

Toward this aim there have been many publications from the early 90s to the current day. In general work has taken place in two different directions. In one direction researchers have attempted to generate animation automatically by better understanding the physical world. These methods use our understanding of the physical properties of the world, along with the physical properties of a virtual character, and a physics simulation to automatically produce animation. As well as an understanding of physics these methods often focus on other aspects of human motion which are key to the specific issue of generating animation in this way. This includes balance, control, sensing, feedback and biology. In essence, these approaches attempt to *simulate* a physically plausible character in a virtual, physically plausible environment by controlling the joint torques or other virtual forces.

The second direction which researchers have looked to for the automatic generation of animation is a *data-driven* direction. In this direction tools and algorithms used for processing and understanding data are applied to animation data in a way that aides artistic interaction. These tools in some sense construct a *model* of animation data which artists can interact with via intuitive controls. For example a system may model animation data using the style or emotion present in the data. Now, by adjusting these variables it may be possible for animators to generate new motion with a desired style or emotion. The research in this thesis falls firmly in this category and as such in the rest of this chapter we will only review work related to this research direction as apposed to any research centered around physically based animation. Additionally, each chapter will include a small literature review covering the publications and previous work which are particularly applicable to that chapter.

There has been a vast amount of research over the years dedicated to the development of new data-driven tools with the aim of automatically producing character animation and reducing the time spent by animators performing manual keyframing. To help in the understanding of this large amount of previous work we have identified five main categories under which most of the research in this direction falls. These categories are given as follows:

- *Motion Signals* - Techniques for processing motion using the one-dimensional signals of each component of rotation around each joint.
- *Motion Blending* - Techniques for generating motions via the blending of two or more motions in a motion database.
- *Spacetime Constraints* - Techniques for editing motions using custom constraints specified in both space and time.
- *Motion Graphs* - Techniques based around constructing a graph data structure to model transitions between motion clips.
- *Statistical Approaches* - Techniques which use Machine Learning and statistics to build generative or interactive models of motion data.

2.2 Motion Signals

The earliest work in the automatic generation of new motion data involved *motion signals*. A motion signal is a one-dimensional signal which represents the movement of single component of a three dimensional rotational or translational joint over time. In an articulated virtual character, depending on the number of degrees of freedom of each joint, there are a certain number of such signals associated to each joint, and by using signal processing techniques many tasks manually performed by animators can be automated.

In *Motion Signal Processing* [Bruderlin and Williams, 1995] motion signals are decomposed into a multi-resolution hierarchy and a number of operations defined over this structure. These operations include *time-warping*, *blending*, and *offsetting*. Time-warping allows an animator to intuitively adjust the timing of an animation on a large scale by scaling the motion signal on the temporal axis. Blending allows an animator to blend two or more motion signals using a weighted average to mix between different motions. Offsetting allows an animator to add an offset or “layer” to a motion signal which can be toggled on and off and scaled, allowing for a more interactive editing procedure. Following research [Lee and Shin, 2001,Lee and Shin, 1999] defined many more operations on motion signals, which are now considered a fundamental tool used by artists in almost all 3D packages geared toward animation.

A landmark work that makes use of motion signals is *Motion Warping* [Witkin and Popovic, 1995]. In *Motion Warping* Cardinal Splines are used to create a mapping between two motion signals with user specified control points. This has the effect of warping the whole motion in the way desired by the animator simply by specifying a few differing poses. By using motion signals, this warping effect can naturally encode warping in both time as well as space.

Motion signals provide a toolbox by which researchers have looked toward other ways to produce novel animations. Perlin [Perlin, 1995] used the concept of *texture* to produce animations with a desired emotion, providing user-controlled noise functions which add some noise to motion signals in an attempt to simulate the personality and emotion of the target motion. Unuma et al. [Unuma et al., 1995] performed a Fourier transform to extract the periodic components of motion signals. These components can be edited, combined and adjusted to change motion in a more abstract way, for example adjusting the emotion of an animation by changing the frequency of some component of the signal.

Motion signals provide a powerful framework for editing motion data but still exist at a relatively low level of abstraction. Animators must still essentially describe the edits they require in the one-dimensional signal space which can often require training to understand and work with intuitively. For this reason, following research has looked toward more high level abstractions that can achieve similar results with simpler interaction.

2.3 Motion Blending

One higher level abstraction which has been subject to a large amount of research is *motion blending*. Motion blending describes a category of techniques where by two or more motions are interpolated to produce a new motion with some given desired properties.

In *Interpolation synthesis for articulated figure motion* [Wiley and Hahn, 1997] Wiley et al. used linear interpolation of exemplar motions to produce new motions with desired properties, for example interpolating reaching motions to produce a motion with the character reaching a new target location, or writing on a whiteboard. Since this generation technique is limited to linear interpolation of the exemplar motions it was incapable of generating precise motion which precisely satisfied the constraints when non-linear interpolation was required.

In *Verbs and Adverbs* [Rose et al., 1998], "Verbs" are used to describe types of motion while "Adverbs" are used to describe their style. Building on the previous work using linear interpolation, this reseach adds an additional non-linear interpolation to generate motion which precisely achieves the desired objectives. First, a linear interpolation is fitted to a set of "Verbs", using the "Adverb" as an input/control variable. Secondly, a set of Radial Basis functions are fitted to the residuals of this linear interpolation. This allows for a non-linear set of "Adverb" values to be used to produce motions ("Verbs") with a new user specified style ("Adverb"). Since non-linear regression is used this process can be performed in any continuous, high dimensional, control space.

Park et al. [Park et al., 2002] apply this same process to the generation of locomotion. Rather than using style as an input they characterize the different motions in the data using the speed and turning angle of the character. Additionally, a slightly different formulation of the regression process is used. Rather than learning a regression from the control parameters to the pose of the character they instead learn a regression from the control parameters to a set of weights for each motion in the exemplar motions. This process produces similar results to the previous method but is more efficient when the dimensionality of the output space is high, and can additionally handle correctly an output space that requires custom forms of interpolation such as with the spherical interpolation of multiple quaternions.

The use of Radial Basis Functions, Gaussian Processes, and other kernel based multi-dimensional scattered data interpolation methods has been a popular component in many works that perform motion blending. Mukai and Kuriyama [Mukai and Kuriyama, 2005] present a detailed statistical approach to the problem of motion blending. Like other works in the field, they adopting a scattered data interpolation method called *Kriging*, a method that is effectively similar to Radial Basis Functions yet they additionally optimise the hyper-parameters of the model to ensure there is a higher quality of motion interpolation in the resulting user control.

Motion blending has been hugely effective in the automatic synthesis of new motion since it remains relatively simple and produces natural, high quality results in many cases. Even so, these approaches have several downsides. One is that most motion blending techniques do not have good computational complexity. Non-linear, kernel based methods scale in memory with the order $O(n^2)$ and in computational cost in the order $O(n^3)$ with respect to the number of data points. Researchers have previously dealt with this using acceleration structures and local regression methods [Wang et al., 2008], but maintaining these structures can be difficult or require manual labelling and segmentation of data. An additional downside is the inability to specify exact constraints the motion must maintain, as often there is some small error in the result of the blending/regression. To achieve motion which exactly satisfies constraints, most often some form of mathematical optimisation must be performed.

2.4 Spacetime Constraints

One way to achieve motions which perfectly satisfy constraints is via a technique called *spacetime constraints*. Using spacetime constraints a user can specify at a very high level spatial and temporal constraints they wish for the motion to satisfy and the motion is produced automatically via non-linear optimisation such that those constraints are satisfied.

In *Spacetime Constraints* [Witkin and Kass, 1988] these spacetime constraints are formulated in a physically accurate way - for example describing the amount of force that a joint is allowed to exhibit as well as the desired position and velocity of the character at a given time. The motion which describes such constraints is then found automatically via a process of non-linear optimisation. Cohen et al. [Cohen, 1992] provide a user interface by which animators can specify these such constraints and expand on the method for solving the non-linear optimisation problem which represents these constraints. Lui et al. [Liu et al., 1994] present a method to decompose these spacetime constraints into a hierarchical structure of B-splines which allows a coarse-to-fine resolution of the spacetime constraint problem and reduces the computational complexity.

While all of these works act in the physical space, the same spacetime constraints can equally be formulated in the kinematic space - an approach more appropriate for data-driven tools. In *Retargetting motion to new characters* [Gleicher, 1998] Gleicher et al. present how spacetime constraints can be used to solve the problem of *retargetting* motion from one character onto another. Rather than finding joint forces that drive a character to solve certain constraints, motion signal offsets are found that result in the given constraints being satisfied. This involves the positioning of end effectors and pose of the character. These offsets are found using Jacobian-based gradient descent and the spacetime paradigm is used to ensure movements remain smooth and natural with the correct time-warping and spatial offsets applied to make the motion appropriate for the target character.

An even more sophisticated form of spacetime constraint editing is applied in *Synchronized Multi-character Motion Editing* [Kim et al., 2009] where by spacetime constraints are specified for multiple characters and their interactions. These constraints are solved both in a discrete and continuous space where by different motion clips can be selected for characters to perform or motions themselves can be edited to ensure that the spatio-temporal constraints are satisfied exactly.

Spacetime constraints provide an intuitive way for animators to edit motion simply by specifying high level properties they wish for the motion to have in the form of a cost function. On the other hand, they suffer from two major issues. Firstly, the optimisation required to solve the spacetime constraints problem is often very slow. This makes these approaches largely impractical in real-time applications as they become difficult to integrate into the interactive feedback loop which is an important part of artists-driven animation. Secondly, producing animation in this way can be difficult as it requires some intuition in how exactly to balance the different hard and soft terms of the cost function such that the desired motion is produced.

2.5 Motion Graphs

One of the most popular approaches to data-driven animation that has seen wide adoption in the games and film industry is a set of techniques called *motion graphs*. These techniques are based around the idea of creating a graph data structure which defines the transitions between different motion clips.

Early work in this direction, *Interactive Motion Generation from Examples* [Arikan and Forsyth, 2002] assumed that a fixed motion graph was given ahead of time and defined operations for searching this graph such that certain user constraints were satisfied such as positioning of limbs, or moving along a given path. This kind of manual construction of a motion graph provides good flexibility and overall control of the output of the system but can be a laborious process, in particular when there are a large number of motions and states which the character can be in. These manually constructed motion graphs are sometimes called *move trees* [Mizuguchi et al., 2001].

In *Motion Graphs* [Kovar et al., 2002] a technique is presented to allow for the automatic construction of this graphical structure from a database. First, a distance matrix is computed between frames in the motion database using a custom distance function and then local minima are extracted to be the transition points. Smooth transitions can then be automatically constructed at the transition points. As in other motion graphs, this graph can then be walked along, searched, or processed using any existing graph based algorithms to achieve the desired user goals. Examples are shown of the character following a desired path as well as walking the tree randomly. Lee et al. [Lee et al., 2002] show various interactive control techniques for motion graph based synthesis including drawing a path through an environment and character control from recorded human motion. This work shows the flexibility and power of the motion graph technique as it can quickly find motion which satisfy a number of user constraints in complex environments.

Due to their flexibility and simplicity, many extensions to Motion Graphs have been proposed. One of the costs of motion graphs is that involved in searching the graph. For a graph with a large branching factor this search cost can quickly become exponential. For this reason researchers have found new ways to reduce the computational complexity of this task. Safonova et al. [Safonova et al., 2004] present a method of searching a motion graph in an optimal way using a combination of A-star search and an additional pruning technique that does not visit states that are non-optimal. Additionally motions with the same contact patterns can be interpolated to produce a continuous variety of motions and a character which is able to perform tasks with respect to specific locations such as jumping on stepping stones and reaching for objects. Another method to aid in the computation of searching motion graphs is precomputation. Lee et al. [Lee and Lee, 2004] use dynamic programming to precompute the utility of performing a certain transition on a motion graph given a particular user goal. For discrete user goals, or user goals with a low dimensionality this can effectively remove the cost of searching a motion graph, but does not scale when the space of user goals is large. Alternatively, Lo et al. [Lo and Zwicker, 2008] use a tree-based regression algorithm instead of a full tabulation of user parameters to solve the same problem in a way that scales better in regards to memory usage.

Another way in which researchers have extended motion graphs is by changing the data which is present at the nodes and edges of the graph structure, for example having multiple motion clips present at a single node or edge. These kinds of motion graphs are often described as “fat” motion graphs. Shin et al. [Shin and Oh, 2006] build a motion graph with poses at the nodes, and groups of motions at the edges - allowing for multiple ways for the character to transition between different poses. In *Parametric Motion Graphs* [Heck and Gleicher, 2007] Heck and Gleicher construct a kind of motion graph with multiple motions both at the nodes and the edges of each graph. Having these multiple motions at a node or along an edge allows for graphs which do not just play back existing motion, but which can produce new continuous motions via motion blending and additional user input. This can allow for several soft constraints to be satisfied more directly such as an exact speed or turning angle, and therefore provides better control over the character. Since motion graphs with multiple motions at nodes and edges are not longer discrete, they can be far more difficult to search than conventional graphs. In this case a search technique based on random sampling is required which can find a path through the parametric motion graph which satisfies given user constraints. In *Motion Graphs++* [Min and Chai, 2012] Min et al. adopt this style of approach. Multiple motions are captured and labelled and the variation in style between each class used to parameterise the exact behaviour of the motion. These motions are then connected together to produce a motion graph which can be searched using statistical methods to find the optimum motion which satisfies some semantic user inputs. Constructing these kinds of motion graphs can require a large amount of manual work. Each motion clip needs to be manually segmented, labelled, and time aligned. Any mistake or failure at this stage can be difficult to diagnose further down the line. To help this issue Kovar and Gleicher [Kovar and Gleicher, 2004] present an automatic technique to help the construction of these “fat” motion graphs. In this work a novel distance metric is used to find motions which are numerically similar and as such may allow motion blending. Since motions which can be blended are found dynamically a new blending function is required which can be computed online. Such a blending function is presented which computes a weighted sum based on the dissimilarity between the motions in the set and resembles a weighted k-nearest neighbours interpolation.

The graphical nature of motion graphs has been applied in other novel and creative ways. In *Motion Patches* [Hyun et al., 2013, Lee et al., 2006] motion clips which involve interactions between multiple characters and the environment are created in a way which allows them to connect spatially. More specifically, each clip (called a “patch”) contains multiple entry and exit points for each character. These patches are then connected such that these entry and exit points connect spatially, allowing a character to seamlessly transition from one “patch” to another. These connections, as in motion graphs, can be represented as a graph structure and the issue of connecting them solved via optimisation.

An interesting extension to motion graphs is explored by Hyun et al. in *Motion Grammars* [Hyun et al., 2016]. Hyun et al. observe that the output of a motion graph - a series of connected motion clips - can be considered similar to a stream of lexical tokens. As in language, there exist some semantic constraints that can exist in motions - for example assuming an initial state of standing, to *stop running* one must have previously *stated running*. These semantic constraints can be encoded via a *grammar* and the same algorithms used for parsing and processing formal languages can be used in the generation of animation. Now, given a semantic description of a motion, and a grammar connected to this description, it is possible to generate “sentences” (streams of connected motion clips) which satisfy this description. Additional controls are added to enforce softer constraints as well such as to deform motions to exactly perform the required motion.

Overall, motion graphs have proved to be hugely successful and have seen widespread adoption in games and films with large numbers of extensions and additions proposed due to their flexibility and simplicity. Yet, in most production strength applications of motion graphs, the graph structure itself is still produced by hand and the overall process controlled heavily by animators and technical developers. One reason for this is that in the automatic motion graph construction the quality and responsiveness of the resulting character can be very difficult to control and edit. This results in a large amount of manual work performed by animators including labelling, segmentation, and the manual insertion of clips into the graph structure. This, combined with the fact that all motion data must be kept in memory limits the scalability of motion graph based techniques. For these reasons, researchers have been interested in approaches which require less human interaction and scale to larger datasets.

2.6 Statistical Approaches

The first approaches to automatic animation synthesis using data were based around signal processing. These were followed by techniques that modelled animation via some discrete structure (such as motion graphs), or by the individual segmentation and classification of different motion alongside user specified control parameters. In more recent work there has been an overall trend toward the formalisation of such techniques using statistical models and machine learning. In this way the vocabulary used and techniques of looking at motion data have changed and researchers have borrowed technology from other fields to better describe, analyse, and generate motion data.

One of the very early works on statistical analysis of motion data was done by Bowden [Bowden, 2000] who applied Principle Component Analysis (PCA) to character poses to learn and visualise the basis of deformation which the character could undergo. To model the temporal aspect of motion Bowden presented a Markov chain based approach which was capable of learning the transition function between poses in the data. This model was used to visualise and generate motion data and showed the expected “looping” structure present in cyclic motions.

Since then many researchers have applied statistical methods such as PCA to understand and generate motion data. Chai et al. [Chai and Hodgins, 2005] used local PCA along with a database of human motion to generate full body motion from low dimensional control signals. In this way they model the latent variables that represent human motion using the low dimensional control signals, and use a learned transformation operation to recover full body motion in the full kinematic space given the latent variable values.

A more popular technique to model human motion data due to its ability to model non-linear data distributions is the Gaussian Process Latent Variable Model (GPLVM). Grochow et al. [Grochow et al., 2004] construct a GPLVM model of human motion data and use this model to better solve motion editing and optimisation problems such as IK. To do this they first use GPLVM to extract the low-dimensional latent variables representing motion and then perform optimisation on these variables with respect to some given user constraints. There are multiple advantages to optimising the values of latent variables representing the motion rather than the full set of values in the kinematic space. Firstly, since motion data lies on a low dimensional manifold the number of degrees of freedom which need to be optimised is often lower, resulting in less computation being required. Secondly, performing optimisation on the latent variables stops the optimisation from exploring poses which are unnatural (or statistically unlikely) which often results in higher quality motion. A similar idea is adopted by Chai et al. [Chai and Hodgins, 2007] who combine the learning of a statistical model of human motion with the classical idea of spacetime constraints. First PCA is applied to poses in the motion database to extract the low-dimensional latent variables representing the motion. Secondly spacetime optimisation is performed in the latent space including priors representing the dynamic behaviour of the latent variables as well as their absolute value. The final result is natural, full body animation, constructed from minimal user input such as sparse keyframes. Safonova et al. [Safonova et al., 2004] present a similar technique where by motion is decomposed into a smaller subspace using PCA and optimisation is performed on the control points of cubic B-Splines which are embedded in the latent variable space. While these methods prove effective in many cases since PCA is a linear subspace reduction technique it cannot always capture the non-linear correlations in the data as in GPLVM.

Levine et al. [Levine et al., 2012] use GPLVM to build a statistical model with the goal of fast and responsive continuous character control. They introduce a novel connectivity prior to their model which allows it to easily discover natural transitions between motion clips. Dynamic programming is then used to precompute policies of traversing this low dimensional space which can be used to enact user control. Using a low dimensional embedding space allows for generation of continuous motions not seen in the training data in a way that is responsive, dynamic, and yet does not appear unnatural. A variant of the GPLVM was used by Wang et al. [Wang et al., 2008] called *Gaussian Process Dynamic Models* (GPDM). In this work a time-series recurrent model of motion data was built which learned the recurrent latent variables of human motion where by the latent variables of the next frame could be predicted from the latent variables of the previous frames. This was used to model, generate and visualise motion data.

An ability for continuous, responsive, high quality character control with minimal manual processing has been a common theme among many works in the area. In *Motion Fields* [Lee et al., 2010] Lee et al. used reinforcement learning and model human motion as a transition function which selects the best blending weights for the ten nearest neighbouring poses in the dataset for a given user control parameter. This results in an optimal character controller with extremely fast responsive times, but since the process takes place in the space of joint angles the character can drift off the manifold of human motion and produce incorrect poses. To solve this an additional force is added to pull the resulting motion toward the k-nearest neighbours.

Linear models of human motion such as PCA are often limited in their expressiveness and cannot capture the full range of human motion. Meanwhile, kernel based methods such as GPLVM have poor scalability as they scale in the order $O(n^2)$ in memory and $O(n^3)$ in computational cost with respect to the number of data points. Local methods can be used to help both of these issues by clustering or segmenting data using spatial acceleration structures and learning separate local models for each part of the database. Yet maintaining these structures can be difficult and cumbersome, while additionally they can take a long time to construct or query. Finally, even with these structures in place all of the motion database must still be stored in memory. For these reasons researchers have looked toward Neural Networks as a potential solution. Neural Networks have a remarkable computational complexity one trained ($O(1)$), can capture non-linear data distributions, and have an almost unlimited capacity for training data.

Some of the first researchers to apply modern neural network techniques to motion data are Taylor et al. who explored a number of methods for modelling motion data using (along with other methods) a kind of neural network called a Conditional Restricted Boltzmann Machine [Taylor and Hinton, 2009, Taylor et al., 2011, Taylor et al., 2006]. The Conditional Restricted Boltzmann Machine is essentially a recurrent model of human motion similar to that of a Hidden Markov Model where by the next pose of the character is predicted from the hidden state of the character in the previous frames. To generate motion, the newly predicted state of the character is again fed back into the network to allow for prediction of the following state. In their research Taylor et al. identified some of the key issues in modelling human motion data. One was that of ambiguity - that the state of the character in the next frame is somewhat ambiguous and multiple solutions might exist in the data. For most machine learning frameworks this results in the next pose of the character becoming an average of the outputs, which can produce over-smoothed motion or the character appearing to float. To solve this issue Taylor proposed to sample the next pose of the character from a probability distribution. While this removes the issue of ambiguity, because sampling is performed each frame the output motion could appear noisy. Another proposed solution was to factor out various variables that could introduce the ambiguity such as the style of the walk - allowing these to be controlled by the user. This approach successfully increased the quality of the motion and allowed the user to control the style but did not entirely remove the tendency of the approach to either produce noisy motion or tend to the average and appear to float.

Other researchers such as Fragkiadaki et al. [Fragkiadaki et al., 2015] also used recurrent models such as the *Encoder Recurrent Decoder* model which performs recurrent generation of motion on the manifold of human motion data. In this model one or more recurrent LSTM layers are placed in between decoding and encoding layers and the whole model is trained end-to-end. In this structure the encoding and decoding layers are capable of learning an intermediate representation of motion data which represents a manifold over the space of motion - I.E. these layers are trained such that they do not encoder invalid motion and as such the recurrent model does not produce odd poses or motions which do not appear in the training data. This model can therefore generate longer sequences before "dying out" - up to several seconds. While a large improvement on previous models of the same style the quality and length of the motion generated from these approaches is still too low for most practical applications and so further research is required before they can be used in production.

Statistical models and machine learning have shown great promise in their ability to efficiently and effectively produce generative models from data sets of human motion. There are several issues with motion data that make this a challenging task and have been a struggle to address in all previous work. The first issue is that of the temporal aspect of motion data - it has not always been clear how this can be incorporated into a statistical model and the commonly used recurrent models often struggle due to the long-term temporal dependences present in motion data. The second issue is that of representation - in the kinematic space many of the potential configurations represent invalid or impossible poses. This space of valid motion gets even smaller when temporal constraints are included. Building a statistical model which produces motion only in this subspace can be challenging - in particular when there is not a lot of motion data available.

The work in the later chapters of this thesis can be considered an attempt to tackle these exact issues. Using the newest tools in machine learning - deep learning and neural networks we approach these issues from a new standpoint. We explore new ways of representing the temporal aspect of motion. This includes *spatializing* it and performing convolution on the temporal dimension. It also includes*factorizing* it using the concept of the phase of the motion - using a unique neural network structure where the weights of the network are generated by another function. To address the problem of motion representation we present a technique for learning a manifold of human motion using a denoising convolutional autoencoder and define a number of operations which can be performed on such a manifold. Overall the work contained in this thesis lies firmly alongside the previous work presented in this sub-category - building on the state of the art and contributions of others in the field.

2.7 Conclusion

The field of automatic motion synthesis with the aim of reducing the manual involvement of animators has a long history with a clear trajectory. Initial methods involved data processing techniques which processed the one-dimensional controls signals of each joint representing the motion. These techniques saw great success in motion editing and as such have become the main tools integrated and used in most modern 3d packages.

Following work saw the introduction of *Motion Blending* and *Motion Graphs* - techniques which were used for interpolation of motion data and the automatic stitching together of motion data into longer sequences. Both approaches proved highly popular due to their simplicity and extensibility and have seen huge adoption across the games and film industry. Yet, both approaches have started to show their limitations in scalability and complexity as they require deep and direct involvement from animators and technical developers to maintain, develop, and extend.

For this reason a number of researchers started looking into more formal statistical models for the generation of animation data, in the hope that they would require fewer manual processes and therefore provide a greater scalability. From this works such as *Motion Fields* [Lee et al., 2010] presented techniques to produce automatic character controllers which were responsive and produced high quality motion with minimal manual processing. Yet many of these techniques had poor computational complexity, resulting in poor scalability with respect to either the amount of data [Levine et al., 2012] or the number of control parameters [Lee et al., 2010].

Our work represents the next step in this process. Building on previous work we use deep learning and new techniques in machine learning to produce techniques for automatic animation generation without these limitations while maintaining the quality required to make them practical for use in production.

Chapter 3

Inverting the Rig Function

3.1 Preface

Research into character animation has seen great success in game development and robotics but so far has not seen wide deployment in keyframed animation environments. In this chapter we identify the core reason for this and present a number of solutions to this obstacle which allow for the seamless integration of existing character animation synthesis and editing tools into keyframed animation.

In character animation research and technology the most common representation of a character's pose is that of an articulated skeleton which is used to drive the underlying skin deformation via a process called *skinning*.

Researchers and software developers therefore most frequency work in this space. They adjust the *joint angles* of the skeleton - the rotations of each skeletal joint relative to its parent. This representation is extremely common in game development and robotics because it accurately encodes how to actually produce poses for an articulated character in the real world.

Yet, this representation is not easily adjusted by hand, and therefore in keyframed animation it is never edited directly by animators. Instead, animators use an interface called an *animation rig* - a system developed in the 3D package by dedicated specalists called *riggers*. Unlike the skeletal representation the rig is designed to be as intuitive to use and as expressive as possible. The *animation rig* is usually built utilising the 3D package's node graph system - a system which allows for the control of certain attributes via a series of connected expressions and computations. In this way the rig can be said to *drive* the underlying skeleton. Animators interface with the rig using *rig controls* - special objects in the 3D scene or on a 2D layout which act as inputs to the rig. Rather than producing keyframes for the joint angles, animators produce keyframes for the *rig controls* and allow the rig to drive the skeleton in turn. Animation data in keyframed animation environments is therefore always stored, edited and used in the space of these rig controls, rather than in the space of the joint angles.

This is the core reason why animators have previously been reluctant to use tools which only work in the space of the joint angles - the results of these tools cannot be edited by the animators after they are produced in any intuitive way. Sometimes rig specific scripts are developed to map the results from these tools back onto the rig controls, but these scripts are specific to certain characters, very complex and difficult to maintain, and may not produce keyframes in the same style as animators.

This chapter presents a series of techniques for mapping results produced in the space of the articulated skeleton back into the space of the rig controls so that animators can make use of existing animation research and technology that works in the space of the joint angles or joint positions. Our techniques are all real-time techniques with minimal overhead, which means they can be used to seamlessly perform this mapping inside of the 3D package, rather than as an offline process. The rest of this chapter consists of a publication in TVCG 2016, which itself is an extension of a previous publication in SCA 2015.

3.2 Introduction

Professional animators design character movements through an *animation rig*. This is a system in the 3D tool that drives the mechanics of the character, e.g. joints, constraints, and deformers, through control parameters. In the production pipeline, animation rigs are designed by specialists called *riggers*, who are responsible for building a rig that is as productive and expressive as possible, so that it intuitively covers all the poses and expressions the animators may want to create. For a complex rig there may be hundreds of rig parameters. For example, our quadruped rig in the examples has six hundred degrees of freedom.

Yet, most character animation research and technologies use raw, low-level structures such as articulated skeletons and 3D polygon meshes as the representation. This makes them difficult to be adopted in the pipeline for the production of animated films. After data such as motion data or deformable surfaces are captured, synthesized or edited in the raw representation, the motion has to be mapped to the animation rig for the animators to edit the results. However, there are often no clear correspondences between the rig controls and the skeletal representation. Previously, complex rig-specific scripts have been created individually for each character and rig. However, these are not general, and require revisions every time new characters and/or rigs are introduced.

The objective of this research is to bridge this gap between character animation research and 3D film production. More specifically, we propose frameworks to map the state of the character's kinematics or geometry to the state of some character rig. Given a set of animator-constructed examples, the raw, low-level data such as the joint positions or geometry of the mesh surfaces as well as the corresponding rig parameters can be extracted. Our system then learns the mapping from the 3D motion data to the rig parameters in an offline stage, employing nonlinear regression techniques.

For mapping the 3D motion data to the rig parameters, we examine and compare two types of nonlinear regression techniques: In addition to the Gaussian process regression [Rasmussen and Williams, 2005] (GPR), we proposed in the earlier version of this paper [Holden et al., 2015a], we also present the results when using feedforward neural networks. The appropriate technique depends on the nature of the rig and the amount of training data available. Gaussian processes are more suitable when there is not much training data and the frame-rate does not need to be high. The neural networks can be more suitable if more training data is available and higher precision and

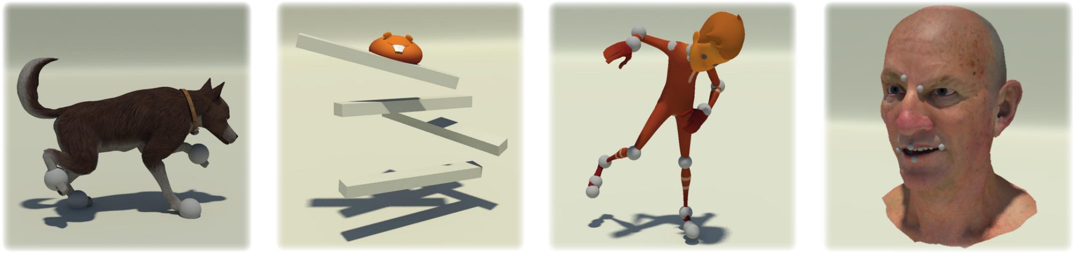

Figure 3.1: Results of our method: animation is generated in the rig space for several different character rigs including a quadruped character, a deformable mesh character, a biped character, and a facial rig. This animation is generated via some external process, yet because it is mapped to the rig space, remains editable by animators.

frame-rate are needed.

Our method can be used to apply any animation techniques that represent a character using joint angles, joint positions or a mesh structure, to characters driven by animation rigs. This includes, but is not limited to, full body motion editing and synthesis, facial animation and 3D shape deformation. In the paper we show some of these applications as results.

The rest of the paper is structured as follows. After describing about the related work, we discuss in detail about the nature of animation rigs, and show how the problem of retargeting some joint positions or angles can be equivalent to the inversion of some *rig function*. Next, we will demonstrate this rig function, its behaviours, and present the technique we use for approximating the inverse of it. Finally, we evaluate our method, present a number of applications of our method, and explain our results.

Our contribution is a method to invert any character *rig function* and generate accurate rig attributes from joint positions in real-time, as well as evaluating various regression frameworks that are suitable for the inverse mapping.

3.3 Related Work

In this section, we first briefly review research related to data-driven animation where mesh surfaces are produced by controlling blending weights of some example data. We then review techniques that learn the mapping between parameters in the task space (i.e. joint positions, landmark positions) and the control parameters. Finally, we review about the work related to animation rigs, which is a professional pipeline for animating characters.

Animation by Blending Example Data: Data-driven approaches are known to be effective for controlling the fine details of characters, which are difficult to produce by simple analytical approaches. Facial animation is one of the main areas that makes use of data-driven approaches, where the degrees of freedom of the system are too high to be entirely modelled by the animators [Pighin et al., 1998, Zhang et al., 2007]. Traditionally, the desired expressions are produced by blending the geometry of different expressions which are either captured by optical cameras or are manually designed by animators. In this case, the blending weights become the control parameters. Such data-driven approaches are also applied for other purposes such as skinning; Pose-space deformation [Lewis et al., 2000] maps the joint angles to the vertex positions using radial basis functions. Sloan et al. [Sloan et al., 2001] extend such an approach for arbitrary applications of mesh deformation. These methods are for conducting a forward mapping from the control parameters to the surfaces, while we attempt the inverse mapping.

Inverse Mapping to Control Parameters: As directly providing the control parameters can be inconvenient in many situations, there is a continuing interest in the inverse mapping. Here the control parameters are estimated from some output parameters, such as the joint positions or the vertex positions of the mesh. One example is inverse kinematics. Required are the control parameters (joint angles) that realizes the task, such as moving the hand to the target location. Classic methods include techniques such as task priority methods [Choi and Ko, 1999], singularity robust inverse [Yamane and Nakamura, 2003], and damped least squares [Buss and Kim, 2004], which originally come from robotics research [Nakamura et al., 1987, Nakamura and Hanafusa, 1986, Chan and Lawrence, 1988].

Researchers in computer graphics propose to directly map the joint positions to the joint angles, using radial basis functions [Kovar and Gleicher, 2004, Rose III et al., 2001], Gaussian processes [Mukai and Kuriyama, 2005] and GPLVM [Grochow et al., 2004]. Similarly in facial animation, researchers compute the blending weights of different expressions from a number of landmark positions, which allows animators to control the face in an inverse kinematics fashion [Zhang et al., 2007, Bickel et al., 2008, Lewis and Anjyo, 2010, Seol and Lewis, 2014]. Xian et al. [Xian et al., 2006] proposed an optimisation based method for the inverse mapping specific to Example Based Skinning. The previous studies assume certain articulation or deformation models such as articulated joint skeletons or blend shapes. Our method is agnostic to the underlying rig mechanism.

Animation Rig: *Character rigging* is the process in a professional animation pipeline where the static geometry of a character is embedded with various animation mechanisms, such as skeletal structure, constraints, and deformers, and then wrapped with intuitive controls for animators. Controls exposed to animators often drive underlying mechanics with custom expressions and chains of graph-structured computation nodes. This makes the rig's behaviour non-linear and difficult to formulate in general. In this paper, we refer to this general mapping of the user-exposed control parameters to the result of the underlying animation mechanics (more specifically, joint positions) as the *rig function*, and the space defined by it as *rig space*. The rig functions includes all the parameters involved in the control of the character, including but not limited to those of forward kinematics, inverse kinematics, blend shape weights and etc.

Only a few papers treat the production animation rig as a system with complex controls and layers of arbitrary underlying driving mechanisms. Hahn et al. [Hahn et al., 2012, Hahn et al., 2013] introduced the idea of the *rig function*, which is a black-box mapping from user-defined controls to mesh vertex positions. Where black-box means that there is only a forward mapping provided by the system, and there is no analytical inverse mapping available for computing the rig parameters. The major bottleneck in inverse mapping of such black-box rig function, as discussed in [Hahn et al., 2012,Hahn et al., 2013], is computing the Jacobian by finite difference, which involves thousands of calls to evaluate a complex rig customized on a 3D software package. For arbitrary and complex rigs this becomes intractable. Seol et al. [Seol and Lewis, 2014] is one of the few papers inversely mapping the face landmarks while treating the face rig as a black-box. Their objective, however, is on retargeting plausible human expressions to virtual characters, not inversely satisfying positional constraints. Our work is motivated by speeding up such computations such that the inverse mapping that satisfies constraints are obtained at interactive rates.

In summary, we propose an approach to produce an inverse mapping from the output of the animation pipeline to the rig parameters. Although there are methods to produce such inverse mapping for rigs consisting of simple skeletons or blendshapes, there has not been a framework that handles arbitrary types of rig functions that treats them

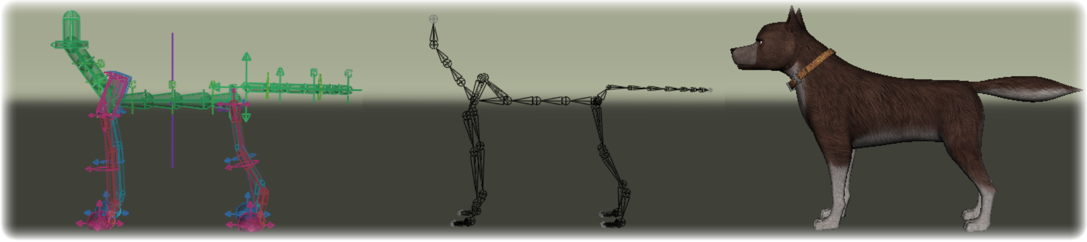

Figure 3.2: Typical setup of rigged character, showing animation rig, underlying skeletal structure, and mesh.

as black-boxes and can compute the inverse at an interactive rate. Our framework increases the precision of such an inverse mapping by learning the Jacobian for fine tuning.

3.4 Rig Function

In this section, we first explain about how the rig is used to determine the posture of a character, and then describe about the requirements of the inverse of the rig function.

3.4.1 Rig Description

Although our approach does not rely on a specific rig, or 3D tool, to give more specific details we describe our experimental set up with an example character, a dog character as set-up in *Maya*.

Fig. 3.2 shows the rig of a character, the underlying skeletal structure, and the mesh. This character's rig consists of *manipulators*. These are the colourful controls, which animators can translate, rotate, or scale in 3D space. The *manipulators* move the skeletal structure, which in turn deforms the mesh. The skeleton itself cannot be moved manually by the animators, nor can the mesh.

Whenever a rig attribute is changed, *Maya* propagates the values to connected components in the scene. This causes *Maya* to recalculate a new configuration for the character skeleton. After this skeletal configuration is found, the character mesh is deformed. In this sense the setup is like a one way function going from rig attributes, to skeletal joints, and finally to the character mesh.

3.4.2 Rig Function & Inversion

Now we describe about the mathematical characteristics of the rig function, and the requirements of its inversion.

Given a vector representing a rig configuration $\mathbf{y}$ and a vector representing the corresponding skeletal structure configuration $\mathbf{x}$, the rig computation, performed internally inside *Maya* for each frame of the animation, can be represented as the function $\mathbf{x} = f(\mathbf{y})$.

We represent the skeletal configuration of the character using a vector of the global joint positions, relative to the character's centre of gravity $\mathbf{x} \in \mathbb{R}^{3j}$ where j is the number of joints. It is worth noting that it is also possible to construct $\mathbf{x}$ using the local joint angles of a skeletal configuration. For simplicity's sake we will only discuss the construction using global joint positions.

Our interest in this research is in the inverse computation $\mathbf{y} = f^{-1}(\mathbf{x})$, where we compute the rig values given the skeletal posture. This is rather difficult due to the following characteristics of f, and the requirements that need to be satisfied as a tool-kit for animation purposes.

The function f is not one-to-one. For any skeletal pose there are several possible rig configurations that could create it. This is intuitively apparent from the fact that IK and FK controls can be used in conjunction on the same section of character. Some user-defined controls can manipulate multiple joints and constraints at the same time through custom expressions and chains of computational nodes. When inverting f we should not just pick a correct $\mathbf{y}$, but also the $\mathbf{y}$ which an animator would naturally specify.

The function f is relatively slow to compute. Evaluation of f in our setup requires interaction with *Maya* which has a fairly large fixed overhead associated [Hahn et al., 2012]. But in any 3D package, a complex rig will also contain non negligible computation in its evaluation. It may contain several complex systems working in conjunction, which may be computationally intensive.

The solutions to the inversion of f must be accurate. If the result requires too much manual correction by animators it may be discarded. In a film environment even small errors in the final product are unacceptable. Any inversion should be able to find an accurate solution that satisfies the equation.

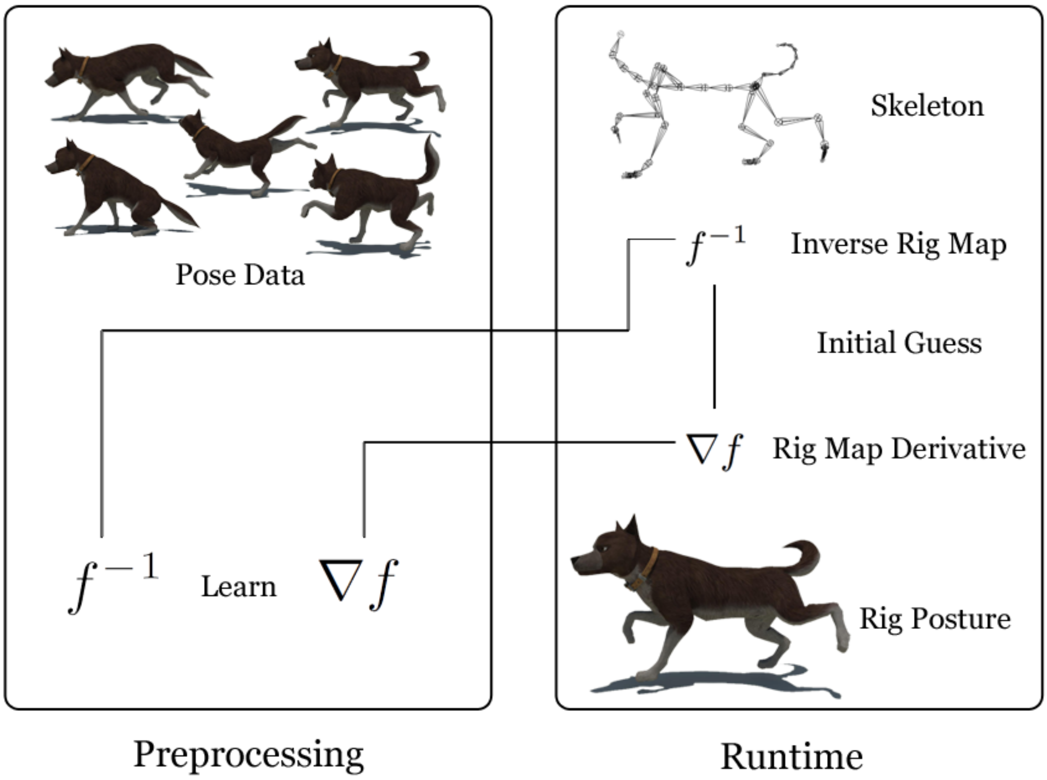

Figure 3.3: Method Overview. We learn an approximation of the inverse of the rig function and its derivative and use this to accurately find rig attributes that match some corresponding joint positions.

The function f must be invertible at interactive rates. Animation is an interactive task which requires a feedback loop between the tools and the animators. Any synthesis tools that rely on this system should have its parameters editable in real-time, so animators can view and edit the results in conjunction with the rest of the scene and make appropriate changes.

3.5 Inverse Rig Mapping by Gaussian Processes

In this section, we review our original technique [Holden et al., 2015a] that applies Gaussian processes regression (GPR) to the inverse rig problem. We first describe how to learn the inverse rig function and its derivative by GPR. We then describe how to refine the mapping using the learned values and derivatives during run-time. The summary of our method is shown in Algorithm 1.

3.5.1 Gaussian Processes Regression

Here we describe the mathematical framework of GPR from the viewpoint of applying it to the inverse rig mapping. A good introduction of Gaussian processes can be found in Rasmussen and Williams [Rasmussen and Williams, 2005].

Given a dataset of rig configurations denoted as $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_n\}$ and the corresponding joint positions denoted as $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$, we are interested in predicting the rig parameters $\mathbf{y}_*$ at arbitrary configuration of joint positions $\mathbf{x}_*$.

We start by defining the covariance function, $k(\mathbf{x}, \mathbf{x}')$ using the following multiquadric kernel (see Discussion), where θ_0 is the “length scale” parameter is found via optimisation (see Section 3.5.1.1):

$$k(\mathbf{x}, \mathbf{x}') = \sqrt{||\mathbf{x} - \mathbf{x}'||^2 + {\theta_0}^2} \tag{3.1}$$

Using the covariance function, we can define the following covariance matrix:

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \ldots & k(\mathbf{x}_1, \mathbf{x}_n) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \ldots & k(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & k(\mathbf{x}_n, \mathbf{x}_2) & \ldots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}, \tag{3.2}$$

$$\mathbf{K}_* = [k(\mathbf{x}_*, \mathbf{x}_1) k(\mathbf{x}_*, \mathbf{x}_2) \ldots k(\mathbf{x}_*, \mathbf{x}_n)], \mathbf{K}_{**} = k(\mathbf{x}_*, \mathbf{x}_*). \tag{3.3}$$

It is then possible to represent each dimension i of the output $\mathbf{y}_*$ as a sample from a multivariate Gaussian distribution N:

$$\begin{bmatrix} \mathbf{Y}^i \\ y_*^i \end{bmatrix} \sim N\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{K}_*^\intercal \\ \mathbf{K}_* & \mathbf{K}_{**} \end{bmatrix}\right), \tag{3.4}$$

where $\mathbf{Y}^i$ is a vector of the i-th dimension of the data points in $\mathbf{Y}$, and y_*^i is the i-th dimension of $\mathbf{y}_*$. The likelihood of some prediction for y_*^i is then given by the following distribution:

$$y_*^i | \mathbf{Y}^i \sim N(\mathbf{K}_* \mathbf{K}^{-1} \mathbf{Y}^i, \mathbf{K}_{**} - \mathbf{K}_* \mathbf{K}^{-1} \mathbf{K}_*^\intercal) \tag{3.5}$$

To compute our final prediction of y_*^i, we take the mean of this distribution subject to Tikhonov regularization.

$$y_*^i = \mathbf{K}_* (\mathbf{K} + \theta_1 \mathbf{I})^{-1} \mathbf{Y}^i \tag{3.6}$$

Where θ_1 is the "smoothing" parameter and can be set to some very small value such as 1×10^{-5} as our data is noiseless.

3.5.1.1 Length Scale Optimisation

The "length scale" parameter θ_0 needs to be set effectively to ensure good interpolation by the Gaussian Process. Because this is a single scalar value we perform a simple line search to find it's optimum value. We regularly take values from from the range $[1 \times 10^{-4}, 1 \times 10^{2}]$ and perform cross validation on the model. For 10 iterations we randomly remove half of the samples from the full data set, train on the remaining data and validate against the samples removed. We take the average error over the iterations to decide which value of θ_0 is best. In our case, for the quadruped character shown in the evaluation, we found a value of 0.0225 was optimum.

3.5.2 Subsampling

In general, the more data supplied to GPR, the more accurately it will perform. But memory usage increases quadratically with the number of data points, so we perform a greedy active learning-based algorithm to subsample the data if it grows too large.

Given the full data set $\mathbf{X}, \mathbf{Y}$ we aim to construct a subsampled data set $\hat{\mathbf{X}}, \hat{\mathbf{Y}}$. We start by including the rest post $\hat{\mathbf{X}} = \{\mathbf{x}_0\}, \hat{\mathbf{Y}} = \{\mathbf{y}_0\}$ and then heuristically picking several points to include in our subsampled data set. We iteratively pick the sample in the full data set furthest from all the included samples in the subsampled data set, and move it from the full data set to the subsampled data set. After some small number of iterations we terminate.

$$\mathbf{x}_i = \arg\max(\min(\|\mathbf{x}_j - \mathbf{x}_i\|) \mid \mathbf{x}_i \in \mathbf{X}, \mathbf{x}_j \in \hat{\mathbf{X}}) \tag{3.7}$$

$$\hat{\mathbf{X}} := \hat{\mathbf{X}} \cup \{\mathbf{x}_i\}, \mathbf{X} := \mathbf{X} \setminus \{\mathbf{x}_i\} \tag{3.8}$$

$$\hat{\mathbf{Y}} := \hat{\mathbf{Y}} \cup \{\mathbf{y}_i\}, \mathbf{Y} := \mathbf{Y} \setminus \{\mathbf{y}_i\} \tag{3.9}$$

We then construct a Gaussian Process conditioned on our subsampled data. We regress each of the remaining data points in the full data set and look at the error of the result.

The data point with the highest error is then moved from the full data set to the subsampled data set.

$$\mathbf{y}_i = \arg\max(\|\mathbf{y}_i - \mathbf{y}_{i*}\| \mid \mathbf{y}_i \in \mathbf{Y}, \mathbf{y}_{i*} \in GPR(\mathbf{X}|\hat{\mathbf{X}})) \tag{3.10}$$

$$\hat{\mathbf{X}} := \hat{\mathbf{X}} \cup \{\mathbf{x}_i\}, \mathbf{X} := \mathbf{X} \setminus \{\mathbf{x}_i\} \tag{3.11}$$

$$\hat{\mathbf{Y}} := \hat{\mathbf{Y}} \cup \{\mathbf{y}_i\}, \mathbf{Y} := \mathbf{Y} \setminus \{\mathbf{y}_i\} \tag{3.12}$$

This is step is repeated until we've reached the required number of samples.

3.5.3 Learning the Derivative

The derivative of the rig function (denoted here as $\mathbf{J}$, where $\mathbf{J} = \frac{\partial \mathbf{x}}{\partial \mathbf{y}}$) can be used in conjunction with gradient descent to further improve the precision of the mapping.

At runtime, given a new target posture $\mathbf{x}_*$, a corresponding Jacobian $\mathbf{J}$ (calculated as explained in Section 3.5.4), and the rig values $\mathbf{y}_*$ computed using GPR, we can apply the rig values to the scene and evaluate the rig function to get the actual posture of the character $\mathbf{x} = f(\mathbf{y}_*)$. There might be some error in this prediction in which case $\mathbf{x} \neq \mathbf{x}_*$, and we then find the difference between the target posture $\mathbf{x}_*$ and the actual posture of the character $\mathbf{x}$ given by $\Delta\mathbf{x} = \mathbf{x}_* - \mathbf{x}$. This difference can be used to compute a change in rig parameters that should be applied to minimize the error in positioning:

$$\Delta\mathbf{y} = (\mathbf{J}^\top\mathbf{J} + \lambda^2\mathbf{I})^{-1}\mathbf{J}^\top\Delta\mathbf{x} \tag{3.13}$$

To ensure stability around singularities we use some damping constant λ. This can be tuned by hand, or automatically selected by examining the error using the SVD of the pseudo-inverse. For more information see [Chan and Lawrence, 1988] [Chiaverini, 1993]. This process is repeated until $\Delta\mathbf{x}$ is below a threshold or some maximum number of iterations is reached.

3.5.4 Learning the Jacobian

We treat the rig function as a black-box, so no analytical form of the Jacobian is available. Therefore, we use finite differences to compute an approximation of the Jacobian

Algorithm 1 Inverse Rig Mapping

```
procedure PRE-PROCESSING
    Sub-sample the given animation data.
    Calculate Jacobian for each pose in the sub-sampled data
    Learn $f^{-1}$ using GPR.
    Learn $\nabla f$ using GPR.
end procedure

procedure RUNTIME
    Predict Rig Attributes $\mathbf{y}_*$.
    Predict Jacobian $\mathbf{J}_*$.
    Initialise rig attributes with predicted values $\mathbf{y}_*$.
    Repeatedly calculate $\Delta\mathbf{x}$ and integrate $\mathbf{y}$.
end procedure
```

of the rig function at some given pose.

Due to the large number of interactions with Maya required, the calculation of the Jacobian in this way is extremely slow, particularly when there are a large number of rig parameters [Hahn et al., 2012]. This process becomes intractable for large sequences of animation. Therefore, we additionally learn a function to predict the Jacobian alongside the rig values, again using GPR.

The formulation of learning the Jacobian is almost exactly the same as learning the rig values. After computing the Jacobian at each example pose, we flatten each Jacobian matrix $\mathbf{J}_i$ to create a single vector $\mathbf{j}_i$, and substitute it in the place of $\mathbf{y}_i$.

Some of the rig attributes are not used by the animators, yet taking the pseudo inverse may lead to these attributes being modified. Instead, we removed any rig attributes not used by the animators from the Jacobian matrix. This results in a gradient descent during the refinement only changing controls which are present in the data.

3.6 Inverse Rig Mapping by Feedforward Neural Networks

In this section, we propose to use a feedforward neural network for the inverse rig function. In the rest of this section, we first discuss about the motivation of using the neural network for the inverse rig mapping, and then describe about the methodology that we adopt.

3.6.1 Motivation

Although GPR is suitable for the inverse rig mapping when there is a small size of the training data, in many situations the amount of the training data required to cover the rig space becomes large. The order of memory usage in GPR grows in the square order of the number of samples which can cause issues with these large data sets. Also, as mentioned above, the precision of the initial guess can be low, requiring the use of iteration using the predicted Jacobian. Because this involved interacting with the maya scene this considerably slows down the process of the inverse rig mapping.

The neural network is a framework that can overcome these issues. The system can learn from a very large set of training data and the precision of the computed results can be improved by increasing the training samples using supersampling. We describe about the process of increasing the training samples in the next section.

3.6.2 Supersampling

Without a lot of data neural networks are susceptible to overfitting and poor generalisation. Due to this, before training, we generate an expanded data set by sampling many rig control configurations from the rig space and evaluating the Rig Function at these points. This allows us to gather more data than just that provided by the animators.

For a small rig function it may be possible to exhaustively cover the rig space with this technique, effectively fully computing the rig function, but most often the rig space is high dimensional, which makes the sampling process very challenging. For a small rig with just 5 parameters, a regular sampling of 10 samples on each axis would require 100000 samples. For a rig vector of 200 degrees of freedom this jumps to 10^{250} -

Algorithm 2 Sampling Points by PCA for NN

```
$\mathbf{M} \leftarrow PCA(\mathbf{Y})$
$\mathbf{X}' \leftarrow \{\}$
$\mathbf{Y}' \leftarrow \{\}$
while $|\mathbf{X}'| < n$ do
    $\mathbf{g} \sim \mathcal{N}$
    $\mathbf{y}' \leftarrow \mathbf{M}^{-1}(\mathbf{M}\, choice(\mathbf{Y}) + \mathbf{g})$
    $\mathbf{x}' \leftarrow f(\mathbf{y}')$
    $\mathbf{Y}' \leftarrow \mathbf{Y}' \cup \{\mathbf{y}'\}$
    $\mathbf{X}' \leftarrow \mathbf{X}' \cup \{\mathbf{x}'\}$
end while
$\mathbf{Y} \leftarrow \mathbf{Y} \cup \mathbf{Y}'$
$\mathbf{X} \leftarrow \mathbf{X} \cup \mathbf{X}'$
```

which clearly makes a regular sampling intractable. For this reason it is important to adapt a smart sampling method which samples rig control configurations the animators are likely to use in future.

To do this we perform PCA on the data given by artists and sample around the given data points in the PCA manifold space. This allows us to sample more frequently on axes of the data with the largest variance, and therefore gather more natural samples which might be poses the animators would themselves produce in the future. The algorithm for this is shown in Algorithm 2.

Using this algorithm we sample an extra 100000 data points from the rig function for training the neural network. This process takes approximately 1 hour but greatly inceases the performance of the neural network regression.

In Section 3.7.2 we evaluate our sampling method against several other sampling methods including Uniform, Univariate Gaussian, Multivariate Gaussian, and Gaussian Mixture Model. Our evaluation both is both qualitative and quantative by visualising the rig poses produced by the sampling and comparing the relative errors.

3.6.3 Training a Feedforward Neural Network

Given the training data produced by sampling, which is denoted by $\mathbf{X}$ as in the previous sections, we want to produce a regression that maps it to $\mathbf{Y}$, which are the corresponding rig values. Here we construct a neural network with parameters θ defined as $\mathbf{Y} = \mathbf{\Phi}(\mathbf{X}; \theta)$. Here $\mathbf{\Phi}$ is the forward function of a small neural network with a single hidden layer defined as follows:

$$\mathbf{\Phi}(\mathbf{x}) = \mathbf{W}_1 ReLU(\mathbf{W}_0\mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1, \tag{3.14}$$

, where the weights and biases are defined as $\theta = \{\mathbf{W}_0 \in \mathbb{R}^{h \times 3j}, \mathbf{b}_0 \in \mathbb{R}^h, \mathbf{W}_1 \in \mathbb{R}^{c \times h}, \mathbf{b}_1 \in \mathbb{R}^c\}$, j is the number of joints, c is the number of rig controls, and h is the number of hidden units in the network (in our work set to 2048).

The activation function used is the rectified linear activation $ReLU(x) = max(x, 0)$ and is used to introduce non-linearity into the mapping. This network is trained using stochastic gradient descent with respect to the following cost function.

$$Cost(\mathbf{x}, \mathbf{y}, \theta) = ||\mathbf{y} - \mathbf{\Phi}(\mathbf{x}))||_2^2 + \alpha\ ||\theta||_1 \tag{3.15}$$

In this equation $||\mathbf{y} - \mathbf{\Phi}(\mathbf{x})||_2^2$ measures the squared regression error and $||\theta||_1$ is a sparsity term that ensures the minimum number of network parameters are used to perform the regression. This term is controlled by some small scalar α, which in this work we set to 0.1.

Before training all data is normalized by subtracting the mean and dividing by the standard deviation. We train the network by taking random minibatches of size 8 from the enlarged data set $\mathbf{X}$, and using automatic derivatives calculated via Theano [Bergstra et al., 2010] we adjust the network parameters θ. To increase the speed and quality of the training we use the adaptive gradient descent algorithm *Adam* [Kingma and Ba, 2014]. Training is performed for 20 epochs and takes approximately 4 hours and 20 minutes on a 4 core Intel i7-4600U 2.1Ghz CPU.

3.7 Evaluation

In this section, we evaluate our method by comparing the error (described below) and the performance with other existing solutions. We then show results of our technique using animation of quadruped, biped, mesh and facial characters.

3.7.1 Performance

In this section we compare our technique to existing methods of approaching this problem. Presented methods include a rig specific script constructed in Maya, and several variations of our own method. For more qualitative results please see supplementary video.

Comparisons are done using the quadruped character, and biped characters shown in the results. The quadruped character has 78 joints, resulting in 234 degrees of freedom in the joint space. It has 642 degrees of freedom in animation rig, but only 218 are used in the animation data so we limit our system to only consider these. The biped character has 48 joints, resulting in 144 degrees of freedom in the joint space. It has 804 degrees of freedom in the animation rig, but only 204 are used in the animation data so we only consider these.

A set of animation sequences of the quadruped and biped character produced by an animator, each using 200 keyframes are provided as the training data. We train the Gaussian Process using 750 subsampled data points extracted from the data set including frames that are produced by interpolating the keyframes, and train the neural network on the full dataset with additional samples found via supersampling as explained in Section 3.6.2.

The rig-specific Maya script, to which we compare our method, is constructed using several of Maya's built-in tools. This script is specific to the quadruped character and is intended to represent existing approaches that have been used for rig retargeting. A script has to be written manually for every new rig and therefore is labour intensive. Primarily it makes use of positional and rotational constraints to place the main rig controls at corresponding joint positions, oriented in the correct directions. These constraints work by traversing the scene hierarchy to calculate transforms for the controls such that they are either placed in a specified location, or oriented in a specified direction. Many of these approaches work in a similar fashion.

We apply each method to three short animation clips. These clips are from a different data set to the training data, chosen such that the character is making large, fast, or extreme motions. This ensures there exists some poses not found in the training data, and that the retargeting task is difficult. For the quadruped these clips include a motion where the dog is running around, jumping and playing, a motion where the dog is galloping and making various sharp turns, and a motion where the dog is begging in an excited way. For the biped these clips involved a short motion of the character making a variety of random movements, a swingdance motion, and a tai chi motion. To evaluate the performance of each approach on each clip we make two comparisons. First we compare the resulting rig attributes found by the approach to those set by the animators. This we call the Ground Truth Error. Secondly we apply these rig attributes to the rig function to get joint positions and compare these to the target joint positions given as the original input. This we call the Joint Error.

Joint Error shows the mean squared error of the difference between the method's resulting joint positions, and those of the target. This is the *visual error* of the method, and also represents the amount of manual correction an animator may have to perform on the result. We regard this error as the most important as it says how closely the character follows the target positions.

Ground Truth Error shows the mean squared error of the difference between the rig attributes found by the method, and those set by the animators. For rotational rig attributes this is measured in radians and normalized to a similar range as the translational controls by dividing by the standard deviation. This error represents the *naturalness* of the key-frames produced by the method, and shows how comfortable the animators might be to use the results. But because there are multiple ways to configure a rig, and because doing comparisons in the rig space may be unreliable, this error may not be indicative of a bad result - it can be considered a secondary objective of the mapping.

We now explain the results shown in Table 3.1.

Maya Script - This method has the largest joint error. Because the script is a heuristic method, it does not try to find an exact solution, and so small errors accumulate over frames, even if the general shape of the character is accurate.

GPR - Using the approximate inverse rig function is very fast because there is no interaction with the Maya scene but has some residual joint error because the mapping is not exact.

GPR & Learned Jacobian - Using the approximate inverse rig function and then additionally learning an approximation of the Jacobian performs two to three times as fast as computing the Jacobian at each frame, and results in significantly less joint error than just using the approximate inverse rig function. But this approach is still somewhat slow compared to GPR alone, as each iteration of gradient descent requires

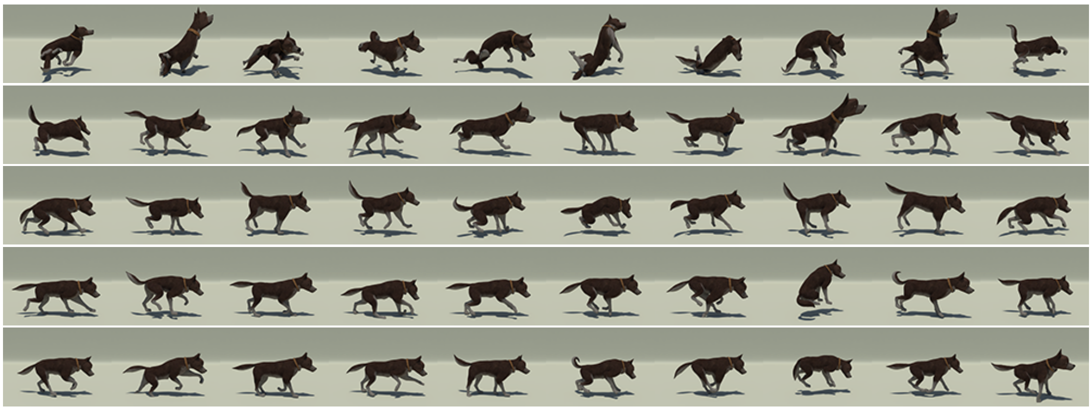

Figure 3.4: Examples of samples from various methods. From Top to Bottom: Uniform, Univariate Gaussian, Multivariate Gaussian, Gaussian Mixture Model, PCA + Unit Gaussian.

evaluation of the difference in joint positions, which means interaction with the Maya scene.

GPR & Computed Jacobian - Using the approximate inverse rig function, and calculating the Jacobian manually at each frame is the most accurate approach using GPR, with the smallest joint error. But this approach is also the slowest variation of GPR, resulting in only one to two frames per second.

NN - Using neural networks to approximate the inverse rig function is very fast because no interaction with the Maya scene is required. It is more accurate than GPR, even with iterations using the Learned Jacobian. On average using neural networks also produces less Ground Truth Error than GPR.

NN & Computed Jacobian - Using neural networks as an initial guess for the Jacobian iterations can increase the accuracy of the result further as this approach consistently gets the lowest joint error of all of the approaches.

These results were collected on a Windows 7 Laptop using a Intel Core i7 2.7Ghz CPU with 16GB of RAM.

3.7.2 Sampling Comparison

In this section we compare different sampling techniques used for training the neural network. We compare our approach of PCA & Unit Gaussian sampling against Uniform sampling, Univariate Gaussian sampling, Mutlivariate Gaussian sampling, and

Gaussian Mixture Model. The joint error and ground truth error when using each sampling method for training the neural network are shown in Table 3.2. For a visual comparison of the samples please see Fig. 3.4.

For simple models such as Uniform sampling and Univariate Gaussian sampling, the correlation between the rig controls are not considered, and therefore the generated samples appear extreme and unnatural (see Fig. 3.4 first and second row). Despite such visual appearance, the Joint and Ground Truth Error of the sequences produced by the neural network trained by the samples are rather low. We can assume this is due to the samples covering a wide range of rig space, allowing the system to produce postures very different from the original example data provided by the animators.

We find more complex models such as Multivariate Gaussian sampling and Gaussian Mixture Model sampling (25 multivariate Gaussians) actually perform worse than the simple models. Although they produce visually more natural samples (see Fig. 3.4 third and fourth row), we assume they overfit to the example data provided by the animators, and as a result produce samples that do not generalize well.

In our experiments, PCA + Normal Gaussian sampling performed the best. Here we first perform PCA on the example data set and then sample around each of the data points given by the artists using a sample from the Unit Gaussian distribution. As with the Multivariate Gaussian approach, this captures the covariance between rig controls, but also ensures dense sampling around all of the artist given data. The samples produced for training the neural network appear visually natural (see Fig. 3.4 bottom row), and the joint error is also low, despite the fact the Ground Truth Error is relatively large.

3.7.3 Results

In this section, we present results of applying our system to character models such as quadrupeds, bipeds, deformable models and facial models. The readers are referred to the supplementary video for the details. Numerical values of the models and the training data are shown in Table 3.3.

In Fig. 3.5 we apply a full body inverse kinematics system to a character using our technique. Given some user-positioned end effectors we move a copy of the characters underlying skeleton using Jacobian full body inverse kinematics toward the end effectors. We extract the global joint positions from the full body IK system and in-

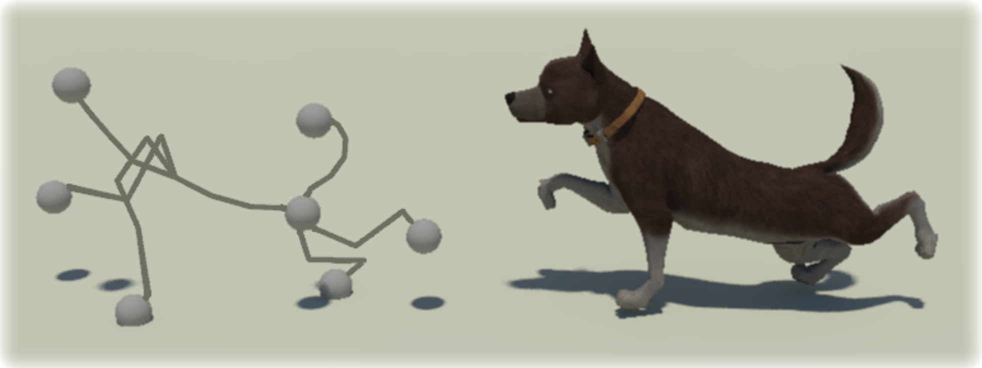

Figure 3.5: Result of Rig-space Full Body IK. From seven end effectors placed at four feet, head, hip, and tail, the optimal rig attributes are approximated by GPR and the solution is further refined by gradient descent. The animators can interactively pose the character using seven end effectors while the rig attributes are updated in real time.

put them into our method to generate corresponding rig parameters. The generated rig parameters accurately follow the skeleton state.

In Fig. 3.7 we show an example of using an inverse mapping where the inputs are only the foot positions. Instead of learning the mapping using all of the joint positions we simply learn it from the four foot positions. A trajectory of the foot positions is then generated from a dog dancing sequence and is fed into the system to compute the rig parameters. It can be observed that our system produces sensible prediction of the full body motion compared to the ground truth motion designed by the animator.

In Fig. 3.6 we show the application of motion editing using our technique. We synthesize some locomotion using animator supplied data and use Spatial Descriptors [Al-Asqhar et al., 2013] to edit the result. This advanced motion editing technique expresses an animation in terms of its environment which allows an animation to be naturally deformed to follow some terrain. First we generate a long locomotion clip in a straight line. Then we generate Spatial Descriptors on the floor, which we bend into a curve and project onto some terrain. After projecting the descriptors, we integrate them to get the new joint positions deformed to fit the terrain. This is performed for each frame. Once we have the final joint positions our method accurately updates the character rig attributes to match these new joint positions.

In Fig. 3.8, we show an example of importing motion capture data and mapping it to a biped character by our method. The character follows the joint positions well, while

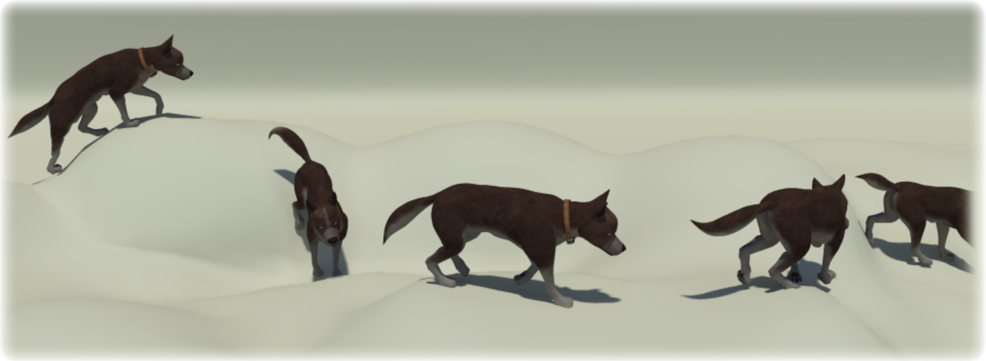

Figure 3.6: Result of Motion Editing. Joint positions are synthesized by generating a locomotion animation, bending it along a curve, and projecting it onto terrain. For all the edited poses, rig attributes are found that match the new joint positions accurately.

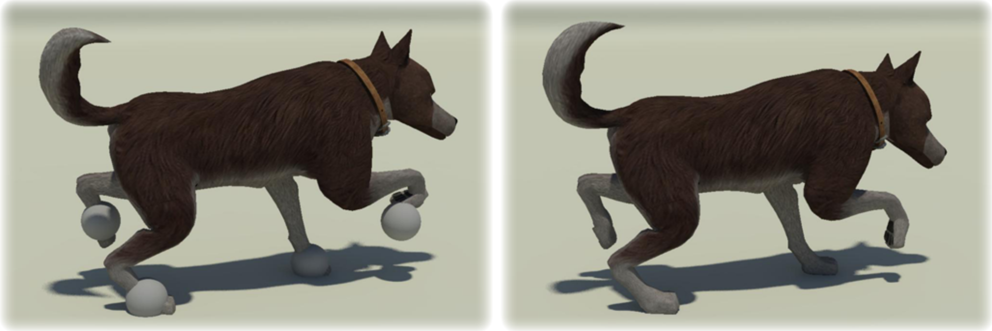

Figure 3.7: The posture of a dog predicted from the foot positions (left) and the ground truth designed by the animator (right).

the resulting trajectories of the rig controllers are smooth and continuous, and as such they are ready for further edits by the animator.

In Fig. 3.9, we show an example of applying our system to a deformable mesh character. A specialized rig that deforms the entire surface of a squirrel character is prepared in this example. Using the rig, the posture of the squirrel can be adjusted and the entire shape of the squirrel is deformed in a cartoonish fashion. Also, the rig is designed such that collisions are avoided when the deformation happens, i.e., the neck part sinks when the neck bends forward so that the teeth do not penetrate the body. Various example poses of the squirrel are designed by the rig and are used as examples for the training. Next, a deformable model of the squirrel is automatically produced from the default pose of the squirrel using the Maya nCloth functionality and its deformation is simulated. We place joints at each mesh vertex, pass their positions to our system and

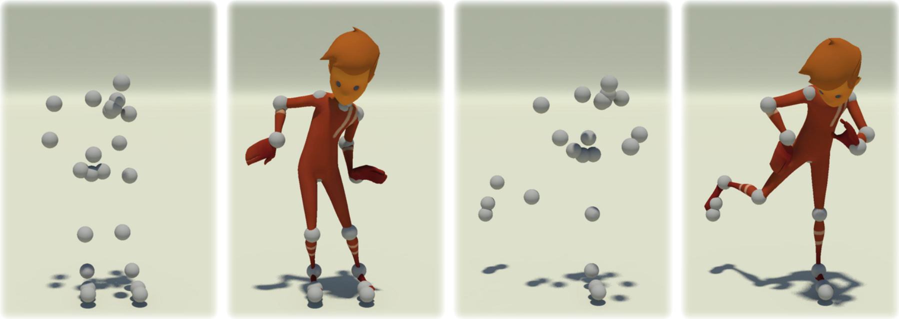

Figure 3.8: Application to Biped. Our approach is generalizable across all rig types. Given data, it immediately works on a character with a different rig, and unique controls.

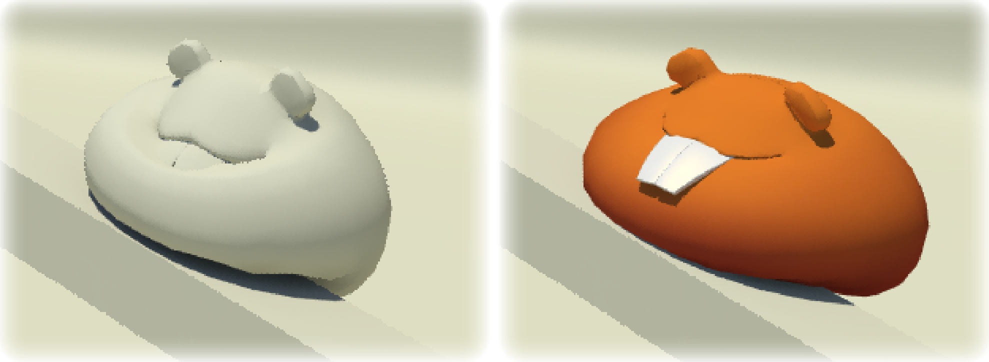

Figure 3.9: A snapshot of a deformable character whose movements are computed by physics simulation (left). Some represented vertices are used as the input for the inverse mapping and the rig animation is produced (right). The rig is carefully designed such that the teeth does not penetrate the body; this effect can be observed in the animation produced by the rig.

the rig parameters are computed using the learned inverse mapping. The poses of the squirrel match the deformation, but also express characteristic features of the rig. As can be observed in Fig. 3.9, the shape of the rigged character is deformed such that the teeth does not penetrate the body when it is squashed.

Finally, an example of applying our method to facial animation is shown in Fig. 3.10. Facial rigs have very complex structures that are composed of multiple controllers including shape deformers and blend shapes. Here, we present the results in a form of FaceIK. This is comparable to [Zhang et al., 2007,Lewis and Anjyo, 2010] except that it is not limited facial rigs using the blendshapes deformation model. Joints are placed at a few facial feature points and the user specifies the desired location of these joints.

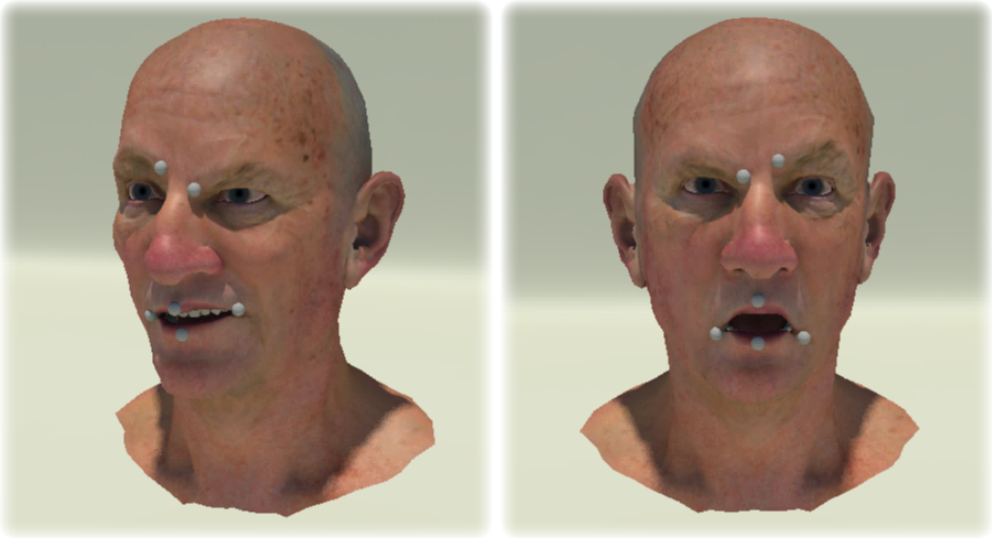

Figure 3.10: Application to facial model in a FaceIK fashion. The user moves the control points and the rig parameters of the face are computed to satisfy the constraints.

The facial expression that satisfies these constraints is automatically computed.

The strength of our method is that it is highly general, as it can be applied to various types of characters controlled by different types of rigs. Specialized scripts for each type of character can be written, but they are not applicable once the rig structure changes. In contrast, our method can be applied under the same principle, irrespective of degrees of freedom of the model.

3.8 Discussion

In this section, we first discuss about the framework that we have chosen. We then briefly discuss the applications of our system.

3.8.1 Framework

Regression Methods for Inverse Mapping

Among the methods that we have tested, the neural networks produces the highest precision for the Joint error when no fine tuning is applied using the Jacobian iteration. The output is in fact precise enough for practical usage and it may not be necessary to

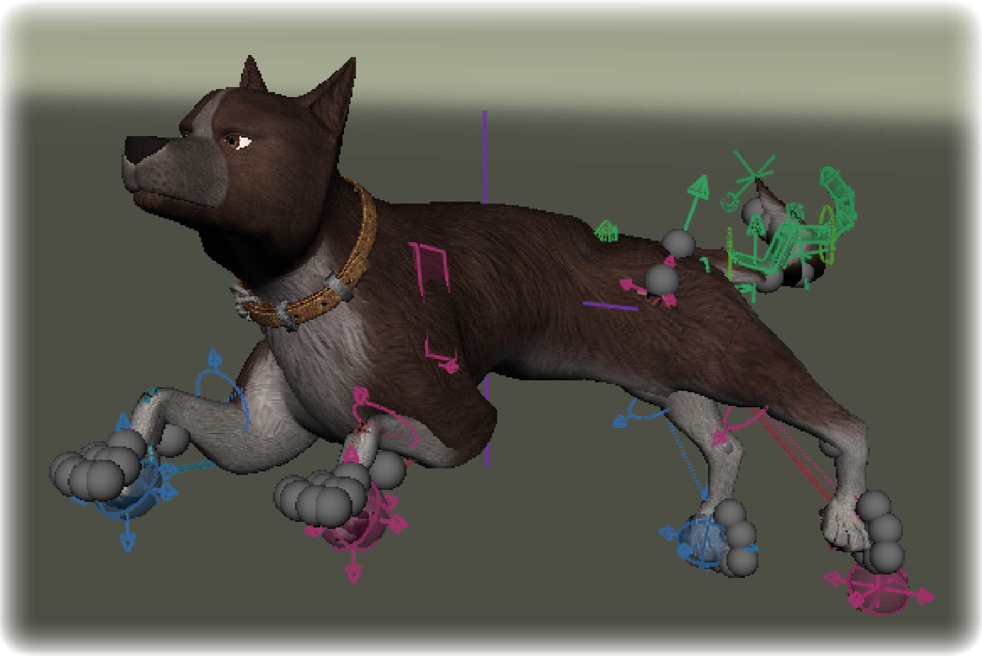

Figure 3.11: Purely optimisation based techniques can result in the joints technically ending up near their targets, but the results are unusable due to rig controls drifting along manifolds in the rig space, away from valid values the animators might set.

further fine tune the results. The ground truth error is also smaller compared to GPR. This comes with the price of producing more training data by sampling, and the system requires much more training time compared to GPR.

Sampling Methods for the Neural Network

In our experiments the proposed sampling method (using PCA with a Normal distribution) produced visually meaningful poses and the lowest numerical error. Although complex models such as multivariate Gaussians and GMMs consider the covariance of the rig controls, the precision of the results produced by the neural networks are rather low. This can potentially be due to overfitting. One possibility to improve the results from complex models is to optimize the meta parameters such as the number of Gaussians for GMM. However, such parameters will vary according to training examples and the nature of the character rig. We find our solution here based on PCA + Uniform Gaussian is a good compromise, which is simple and requires no complex parameter tuning.

Multiquadric Kernel for GPR

We find the performance of the multiquadric kernel that we have adopted is better than other kernels including Gaussian, Linear and Polynomial kernels. Other kernels perform poorer in terms of interpolation and extrapolation, often resulting in instability and large error, even after optimizing kernel parameters. The reason that the multiquadric kernel performs the best can be considered as follows: When interpolating with lots of data present, the multiquadric kernel is smooth, and so approximates the popular squared exponent kernel, which has proven effective for many machine learning tasks. Yet, when there are large gaps between data points, or the function is extrapolating, the squared exponent kernel results in the interpolated value tending toward zero. The multiquadric kernel on the other hand begins to approximate a linear kernel, and approximates a piecewise linear interpolation of the data points. Due to the high dimensionality, and sparsity of our data, there are often irregular gaps and spacing. Using a squared exponent kernel would therefore result in a landscape with large peaks and troughs where it irregularly drops to zero. The multiquadric kernel instead results in a stiffer landscape, and therefore extrapolates more accurately. The multiquadric function is conditionally positive definite rather than positive semi-definite, and thus is not strictly speaking a valid covariance - but the increasing nature of the kernel is what results in the approximation of piecewise linear interpolation instead of dropping to zero.

Inverse Rig Function Derivative

When using GPR for the inverse mapping, we approximate the inverse rig function, and the rig function derivative separately, before taking the pseudo-inverse of the predicted Jacobian. It can be observed that it should be possible to take the first derivative of the approximate inverse rig function instead.

We predict the Jacobian separately because kernel based methods such as GPR or RBF are known to approximate the 0th order derivative (the actual function values) much more accurately than they approximate 1st, 2nd, or following derivatives [Mai-Duy and Tran-Cong, 2003]. This is something we confirmed in our initial experiments.

One common way to reduce this error is to incorporate gradient constraints into the GP formulation. This results in having to solve a matrix which is of the order $O(N^2D^2)$ where N is the number of samples and D is the dimensionality of the input space. In our problem, where D can be as large as 250 this quickly becomes intractable. In this sense, interpolating the Jacobian independently has remarkable performance, because although (when flattened) it can be 50000 dimensions in size, the matrix to solve is only proportional to the number of data samples. Even so, this is an area we are very interested in and wish to do more research to try to find an appropriate technique which can be used under realistic memory constraints.

Rig Function Ambiguity

There are many possible ways to set the rig controls to construct the same pose, but lots of these configurations are undesirable because they are not how an animator would naturally animate. Using purely optimisation-based approaches results in the rig controls drifting along manifolds which technically result in accurate joint positions, but have terrible rig values. This is shown in Fig. 3.11. Using machine learning can solve this implicitly by using the animator supplied data to additionally learn what are "valid" or "sensible" rig control settings. Even if we perform a small amount of gradient descent, using an initial guess generated from data ensures the error in the rig does not get too high to make the result unusable.

3.8.2 Applications

Our work has a large number of applications in key-framed animation environments, as it allows for the better use of character animation research and technology on rigged characters. Primarily it means that data-driven animation techniques can be effectively combined with animator artistry to save time and cost in the production of animated entertainment.

Our work allows animators to use Motion Capture, Motion Warping, and Motion Editing techniques on the motions they construct for characters. For example our work could be used in conjunction with Motion Layers [Lee and Shin, 2001], Motion Warping [Witkin and Popovic, 1995], Full-Body Inverse Kinematics [Yamane and Nakamura, 2003], and Relationship Descriptors [Al-Asqhar et al., 2013]. Because our approach is in real-time it allows for a tight feedback loop between these motion tools and animator edits. This greatly increases the speed and efficiency at which animators can work.

3.9 Conclusion

We present a link between a character rig, and its underlying skeleton in the form of a rig function f. We show our method for inversion of this rig function, and evaluate it against potential alternatives. The resulting ability to quickly and effectively invert this rig function has broad applications in key-framed animation environments as it allows for a tight feedback loop between animators, and animation tools that work in the space of joint positions.

Acknowledgement

We would like to thank Marza Animation Planet [Mar,] for their support of this research as well as their permission to use the Dog character rig and animation data. We thank Faceware Technologies [Fac,] for permission to use their facial rig, as well as their facial animation data for training and demonstrating our system. We also thank Animation Mentor [Ani,] for permission to use their two character rigs Stewart and Squirrel in our research.

Dog character rig and animation (© Marza Animation Planet, Inc. 2014) Stewart character used with permission (© Animation Mentor 2013). No endorsement of sponsorship by Animation Mentor. Stewart can be downloaded at http://www.animationmentor.com/free-maya-rig. Squirrels character used with permission (© Animation Mentor 2013). No endorsement of sponsorship by Animation Mentor. Squirrels can be downloaded at http://www.animationmentor.com/free-maya-rig. Old Man facial rig and animation (© Faceware Technologies, Inc. 2012).

Character/Motion	Method	Joint Error	Ground Truth Error	Total Time (s)	Frames per Sec
Quadruped - *Jumping & Playing*	Maya Script	3.566	1.406	2.79	25.04
	GPR	0.153	3.877	0.21	315.06
	GPR & Learned Jacobian	0.147	3.877	10.73	6.42
	GPR & Computed Jacobian	0.071	3.864	41.24	1.67
	NN	0.087	2.604	0.21	319.44
	NN & Computed Jacobian	0.050	2.597	40.63	1.69
Quadruped - *Begging Excitedly*	Maya Script	0.161	0.424	2.68	30.95
	GPR	0.022	0.565	0.24	340.24
	GPR & Learned Jacobian	0.020	0.566	11.51	7.12
	GPR & Computed Jacobian	0.007	0.56	32.75	2.50
	NN	0.011	1.005	0.28	283.73
	NN & Computed Jacobian	0.005	1.002	33.02	2.48
Quadruped - *Turning & Galloping*	Maya Script	1.023	0.972	1.27	29.13
	GPR	0.041	1.547	0.18	195.65
	GPR & Learned Jacobian	0.039	1.547	6.29	5.71
	GPR & Computed Jacobian	0.023	1.537	23.86	1.50
	NN	0.036	0.839	0.17	208.09
	NN & Computed Jacobian	0.020	0.836	24.29	1.48
Quadruped - **Average**	Maya Script	1.583	0.934	2.24	28.38
	GPR	0.072	1.996	0.21	283.65
	GPR & Learned Jacobian	0.068	2.000	9.51	6.41
	GPR & Computed Jacobian	0.033	1.987	32.61	1.89
	NN	0.044	1.482	0.22	270.42
	NN & Computed Jacobian	0.025	1.478	32.64	1.88
Biped - *Range of Motion*	Maya Script	10.833	3.143	9.40	14.46
	GPR	0.460	0.214	0.22	608.10
	GPR & Learned Jacobian	0.326	0.215	22.75	5.93
	GPR & Computed Jacobian	0.121	0.206	53.74	2.51
	NN	0.339	0.157	0.25	527.34
	NN & Computed Jacobian	0.052	0.151	60.58	2.22
Biped - *Swing Dance*	Maya Script	8.871	2.852	4.56	14.23
	GPR	0.133	0.051	0.15	426.66
	GPR & Learned Jacobian	0.089	0.051	11.56	5.53
	GPR & Computed Jacobian	0.037	0.048	24.45	2.61
	NN	0.198	0.076	0.17	357.54
	NN & Computed Jacobian	0.044	0.073	29.18	2.19
Biped - *Tai chi*	Maya Script	8.198	3.091	13.94	14.33
	GPR	0.361	0.233	0.30	656.76
	GPR & Learned Jacobian	0.282	0.233	37.26	5.34
	GPR & Computed Jacobian	0.083	0.227	85.71	2.32
	NN	0.183	0.125	0.30	646.10
	NN & Computed Jacobian	0.031	0.122	89.88	2.21
Biped - **Average**	Maya Script	9.300	3.028	9.3	14.34
	GPR	0.318	0.166	0.22	563.80
	GPR & Learned Jacobian	0.232	0.166	23.85	5.6
	GPR & Computed Jacobian	0.080	0.160	54.63	2.48
	NN	0.240	0.119	0.240	510.32
	NN & Computed Jacobian	0.042	0.115	59.88	2.20

Table 3.1: Comparison of different methods.

Method	Joint Error	Ground Truth Error
No Extra Samples	0.235	1.196
Uniform	0.057	0.915
Univariate Gaussian	0.049	0.517
Multivariate Gaussian	0.085	1.366
Gaussian Mixture Model	0.077	0.992
PCA & Unit Gaussian	0.044	1.482

Table 3.2: Comparison of sampling methods.

	Training Frames	Rig DOFs	Joint DOFs
Quadruped	750	642	234
Biped	750	697	144
Facial	300	49	18
Squirrel	500	36	3375

Table 3.3: Numerical data of the models used in the experiments.

3.10 Postscript

By providing a general, accurate, and real-time approach to inverting the rig function we have allowed almost all existing animation tools which use joint angles or positions to be frictionlessly integrated into keyframed animation environments with very little overhead.

This represents the first step toward reducing animator keyframes, allowing for both the use of current tools, and the development of new tools with the hope that they can be applied to keyframed animation. This work is the first work in the field specifically targetting this issue and we hope it will lead to future improvements and a much larger adoption of many of the existing animation synthesis techniques in the production of animated films.

Chapter 4

The Motion Manifold

4.1 Preface

In this chapter we present a new tool developed for data-driven character animation called the *motion manifold*. Given the solution to the inverse rig function presented in the previous chapter we continue with the development of new data-driven tools using the most recent developments in machine learning. Such tools can reduce animator keyframes by automatically producing animation in a wide number of situations from high level controls. Here we present a technique which can be used as a component in many of these such tools.

The motion manifold is a new representation of animation data which has many uses in animation research and technology. As explained in the previous chapter, animation data is typically represented as a time-series of joint angles or joint positions, where each frame consists of a high dimensional vector representing the pose of the character. This representation is excellent for data processing but natural motion data only exists in a small subspace of this representation. For example, it is easy to produce motion that is unnatural, either by creating large differences between frames such that joints move too fast, or by adjusting the joint angles or positions to produce poses which are physically impossible to achieve.

The subspace of this represenation which constitutes natural motion is what we call the motion manifold, and in this chapter we show techniques for operating on such a manifold. By operating on the motion manifold we can redefine many operations with respect to only what is natural motion. For example we can interpolate two motions passing only through the space of natural motion, or we can take the distance between two motions along the manifold to get a more natural measure of dissimilarity.

This chapter presents a technique for the construction and operation on this motion manifold using deep learning and convolutional neural networks. It consists of a publication in SIGGRAPH Asia 2015 Technical Briefs.

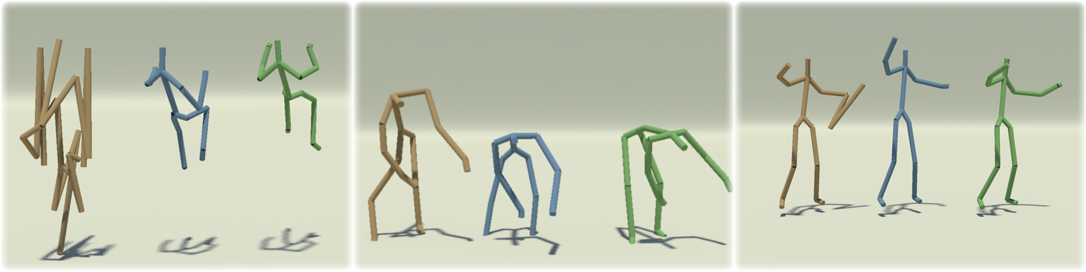

Figure 4.1: A set of corrupted motions (orange) fixed by projection onto the motion manifold (blue) and compared to the ground truth (green). Motion corrupted with artificial noise that randomly sets values to zero with a probability of 0.5 (left), motion data from Kinect motion capture system (center), and motion data with missing the wrist marker (right).

4.2 Introduction

Motion data is typically represented as a time-series where each frame represents some pose of a character. Poses of a character are usually parametrized by the character joint angles, or joint positions. This representation is excellent for data processing, but valid human motion only exists in a small subspace of this representation. It is easy to find a configuration in this space which does not represent valid human motion - either by setting the joint angles to extreme configurations and making poses that are biologically impossible, or by creating large differences between frames of the time-series such that the motion is too fast for humans to achieve.

It is of great interest to researchers to find good techniques for finding this subspace of valid motion, called the motion manifold. This is because it allows for data processing tasks to be defined with respect to it - such as interpolating motion only through the space of valid motions - or taking distances which do not pass through the space of impossible motion.

Our contribution is the following:

- Unsupervised learning of a human motion manifold by a Convolutional Autoencoder
- Application of our manifold in various areas of animation research.

4.3 Related Work

In this section we will briefly discuss works aimed at learning a manifold over the space of motion data and then works related to Convolutional Neural Networks.

4.3.0.0.1 Learning the motion manifold It is of interest to researchers in machine learning and computer graphics to produce motion manifolds from human motion data. PCA is an effective approach for modelling the manifold of human gait, although it cannot handle many different types of motion altogether. Chai et al. [Chai and Hodgins, 2005] apply local PCA to produce a motion manifold that includes of various types of human motion data, and apply it for synthesizing movements from low dimensional signals. Lee et al. [Lee et al., 2010] propose a data structure called motion fields where the user can interactively control the characters in the motion manifold. These methods require a lot of preprocessing, including computing the distance between frames and producing structures such as KD trees for handling large amounts of data. Rose et al. map the hand position to the full body motion using Radial Basis Functions (RBF). Mukai and Kuriyama [Mukai and Kuriyama, 2005] apply Gaussian Processes (GP) to produce a smoother mapping. Lawrence [Lawrence, 2003] proposes to apply Gaussian Process Latent Variable Model (GPLVM) to produce a mapping between low dimensional latent space and high dimensional data. Grochow et al. [Grochow et al., 2004] apply GPLVM to human motion synthesis. Wang et al. [Wang et al., 2008] apply GPLVM to develop dynamic models of human motion. Non-linear kernel based methods such as RBF/GP cannot scale as the covariance matrix grows as the square of the number of data points used. Additionally these works suffer from the same issue as other works of only encoding the temporal aspect of motion once a manifold has been built for human poses.

4.3.0.0.2 Neural networks and its application to motion data Convolutional Neural Networks (CNNs) have been used effectively in many machine learning tasks. They have achieved particular success in the domain of image classification [Krizhevsky et al., 2012, Ciresan et al., 2012] but they have additionally been used for many other tasks including video classification [Karpathy et al., 2014], facial recognition [Nasse et al., 2009], action recognition [Ji et al., 2013], tracking [Fan et al., 2010] and speech recognition [Abdel-Hamid et al., 2014, Ji et al., 2013]. Our work applies these techniques to the domain of learning a motion manifold on human motion data.

Taylor et al. [Taylor et al., 2011] use the notion of a Conditional Restricted Boltzmann machine to learn a time-series predictor which can predict the next pose of a motion given several previous frames. While this approach does not rely on clustering or processing of pose data its capacity is still limited because with lots of data there exists large ambiguity in generation of the next pose which can lead to either the motion ”dying out” and reaching the average pose, or high frequency noise introduced by sampling.

The difference in our approach is to use convolution to explicitly encode the temporal aspect of motion with equal importance to the pose subspace.

4.4 Notations

Given a time-series of human poses $\mathbf{X} \in \mathbb{R}^{nm}$ where n is the number of frames and m is the number of degrees of freedom in the representation of the pose, we wish to learn some manifold projection operator $\mathbf{Y} = \mathbf{\Phi}(\mathbf{X})$, $\mathbf{Y} \in [-1, 1]^{ik}$ and some inverse projection operator $\hat{\mathbf{X}} = \mathbf{\Phi}^{\dagger}(\mathbf{Y})$, $\hat{\mathbf{X}} \in \mathbb{R}^{nm}$ such that the application of the inverse of the projection operation $\mathbf{\Phi}^{\dagger}$ always produces a $\hat{\mathbf{X}}$ which lies within the subspace of valid human motion. Here i, k are the number of frames and degrees of freedom on the motion manifold, respectively. The space $[-1, 1]^{ik}$ therefore represents the parametrisation of the manifold. As this space is bounded by -1 and 1, so is the manifold, and in areas outside this bound the inverse projection operation is undefined. Additionally, the uniform distribution of values of $\hat{\mathbf{X}}$ drawn from $[-1, 1]^{ik}$ represents a prior probability distribution over the space of human motion which can be augmented with other beliefs for use in machine learning tasks.

In this paper we learn the projection operation $\mathbf{\Phi}$ and inverse projection operation $\mathbf{\Phi}^{\dagger}$ using a Deep Convolutional Autoencoder [Vincent et al., 2010]. The *Visible Units* correspond to $\mathbf{X}$ and the values of the *Hidden Units* at the deepest layer corresponds to $\mathbf{Y}$. Propagating the input through the network is therefore the projection operator $\mathbf{\Phi}$, and propagating the *Hidden Units* values backward to reconstruct *Visible Units* is the inverse operation $\mathbf{\Phi}^{\dagger}$.

4.5 Data Preprocessing

In this section we explain our construction of values of $\mathbf{X}$ using time-series data from the CMU motion capture database.

We use the full CMU Motion Capture Database [Cmu,] which consists of 2605 recordings of roughly 10 hours of human motion, captured with an optical motion capture system. We sub-sample this data to 30 frames per second and separate it out into overlapping windows of 160 frames (overlapped by 80 frames). This results in 9673 windows. We choose 160 frames as it roughly covers the period of most distinct human actions (around 5 seconds) - but in our framework this dimension is variable in size.

We normalize the joint lengths, and use the joint positions of 20 of the most important joints. Global translation around the XZ plane is removed, and global rotation around the Y axis removed.

The rotational velocity around the Y axis and the translational velocity in the XZ plane relative to the character's forward direction are included in the input vector. These can be integrated over time to recover the global position and rotation of the character in an animation. Finally we subtract the mean pose from each pose vector before input to the network. This results in a final input vector with a window size of 160 and 63 degrees of freedom. $\mathbf{X} \in \mathbb{R}^{160\times 63}$.

4.6 Convolutional Neural Networks for Learning Motion Data

In this section we will explain the structure of the Convolutional Autoencoder. Readers are referred to tutorials such [DeepLearning,] for the basics of Convolutional Neural Networks. We construct and train a three-layer Convolutional Autoencoder. An overview of our network structure is given by Fig. 4.2 and an overview of the transformations is given by Fig. 4.3.

Each layer of the CNN performs a one-dimension convolution over the temporal domain independently for a number of filters. The output of the convolution is called a *feature map*, and represents the presence of that particular filter at a certain point in

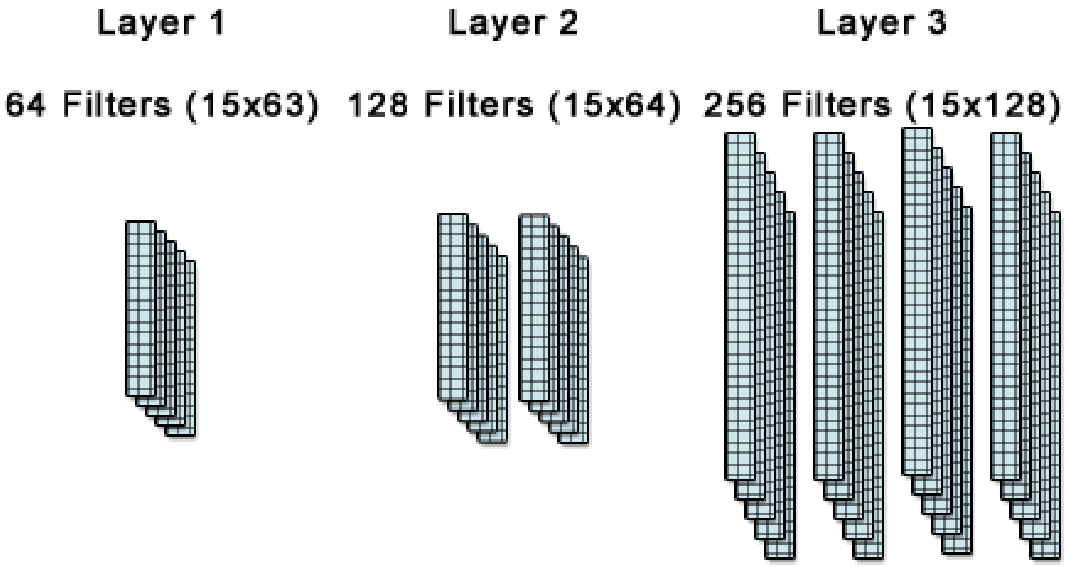

Figure 4.2: Structure of the Convolutional Autoencoder. Layer 1 contains 64 filters of size 15x63. Layer 2 contains 128 filters of size 15x64. Layer 3 contains 256 filters of size 15x128. The first dimension of the filter corresponds to a temporal window, while the second dimension corresponds to the number of features/filters on the layer below.

time. A filter size of 15 is chosen as it roughly corresponds to small actions on the lower layers (half a second), and larger actions on the deeper layers (several seconds). Filter values **W** are initialised to some small random values found using the "fan-in" and "fan-out" criteria [Hinton, 2012], while biases **b** are initialised to zero.

Now we describe about the forward process in each layer. First the input data is convolved, and then a one-dimensional max pooling step is used to shrink the temporal domain - taking the maximum of each consecutive pair of frames. Max pooling provides a level of temporal invariance and encourages filters to learn separate distinct features of the input. Finally the output is put through the tanh function. This places it in the range $[-1, 1]$, which compresses the input, effectively clipping it to the subspace, as well as allowing the neural network to construct a non-linear manifold.

For a single layer k, given the convolution operator $*$, max pooling operator $\mathbf{\Psi}$, filter weights $\mathbf{W}_k$ and biases $\mathbf{b}_k$, the projection operation is given by the following.

$$\mathbf{\Phi}_k(\mathbf{X}) = \tanh(\mathbf{\Psi}(\mathbf{X} * \mathbf{W}_k + \mathbf{b}_k)) \tag{4.1}$$

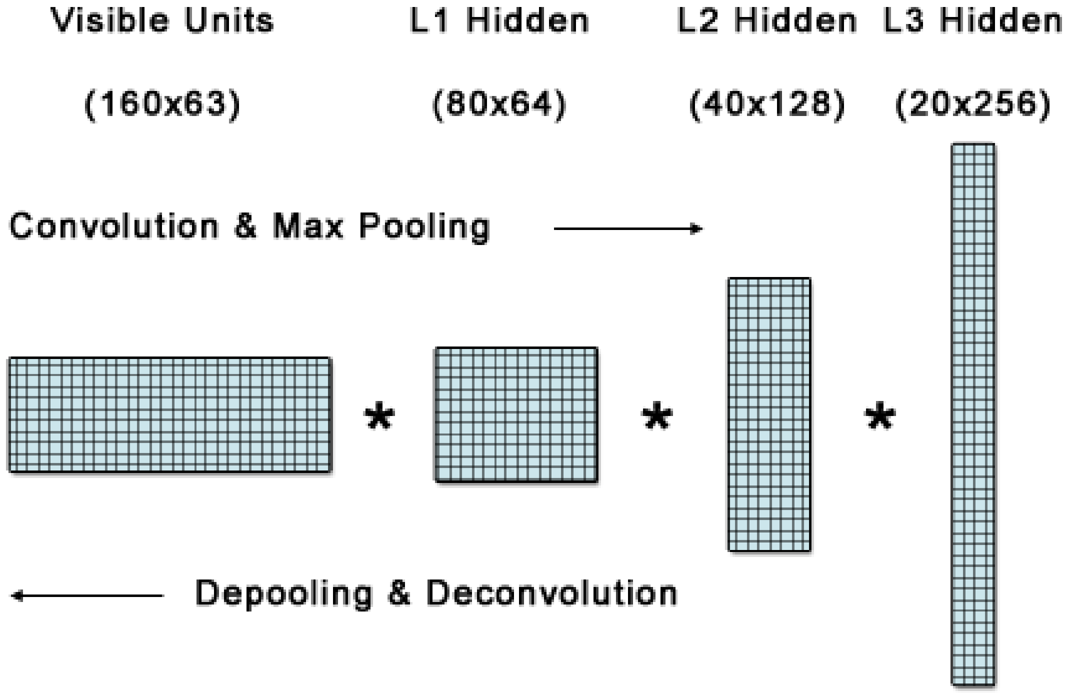

Figure 4.3: Units of the Convolutional Autoencoder. The input to layer 1 is a window of 160 frames of 63 degrees of freedom. After the first convolution and max pooling this becomes a window of 80 with 64 degrees of freedom. After layer 2 it becomes 40 by 128, and after layer 3 it becomes 20 by 256.

Now we describe about the backward process in each layer. This is done by simply inverting each component of the network, taking special care to invert the pooling operator $\mathbf{\Psi}$. In general this operator is non-invertible, so instead we formulate two approximate inverses that can be used in different cases. When training the network, which we describe shortly in Section 6.6.3, a single individual pooling location is picked randomly and the pooled value is placed there, setting the other pooling location to zero. This is a good approximation for the inverse of max pooling process, although it results in some noise in the produced animation. When training in the fine tuning stage, which is the second stage of the training stage that we describe in Section 6.6.3, or when performing projection to generate the animations shown in the results, the pooled value is instead distributed evenly across both pooling locations. This is not an accurate approximation of the inverse of the maximum operation, but it does result in smooth animation with no high frequency noise.

The inverse projection for some layer k is then given as the following. Where $\tilde{\mathbf{W}}_k$ represents the weights matrix flipped on the temporal axis and transposed on the axes representing each filter on this layer and the filters on the layer below. The $\mathbf{\Psi}^{\dagger}$ function represents the inverse pooling operation described above.

$$\mathbf{\Phi}_k^{\dagger}(\mathbf{Y}) = (\mathbf{\Psi}^{\dagger}(\tanh^{-1}(\mathbf{Y})) - \mathbf{b}_k) * \tilde{\mathbf{W}}_k \tag{4.2}$$

4.7 Training

We train the network using Denoising Autoencoding. The input $\mathbf{X}$ is corrupted to some new value $\mathbf{X}_c$ and the network is trained to reproduce the original input. The corrupted input $\mathbf{X}_c$ is produced by randomly setting values of the input $\mathbf{X}$ to zero with a probability of 0.1.

Training is first done individually for each layer. Once each layer is trained, the whole network is trained in a stage called *fine tuning*. Training is posed as an optimisation problem. We find weights $\mathbf{W}$ and biases $\mathbf{b}$ such that the following loss function is minimised.

$$Loss(\mathbf{X}) = ||\mathbf{X} - \mathbf{\Phi}^{\dagger}(\mathbf{\Phi}(\mathbf{X}_c))||_2^2 + \alpha\ ||\mathbf{\Phi}(\mathbf{X}_c)||_1 \tag{4.3}$$

Where $||\mathbf{X} - \mathbf{\Phi}^{\dagger}(\mathbf{\Phi}(\mathbf{X}_c))||_2^2$ measures the squared reproduction error and $||\mathbf{\Phi}(\mathbf{X}_c)||_1$ represents an additional sparsity constraint that ensures the minimum number of hidden units are active at a time, so that independent local features are learned. This is scaled by some small constant α which in our case is set to 0.01.

We make use of automatic derivatives calculated by Theano [Bergstra et al., 2010] and iteratively input each data point while updating the filters in the direction that minimises the loss function. We use some learning rate initially set to the value of 0.5 and slowly decayed by a factor of 0.9 at each epoch. Each layer is trained for 25 epochs.

In the fine tuning stage a different learning rate of 0.01 is used and the constant α is set to zero. Additionally, as mentioned in Section 4.6, the alternative method of max pooling is used, where values are distributed evenly.

Once training is complete the filters express strong temporal and inter-joint correspondences. Each filter expresses the movement of several joints over a period of time.

Each of these filters therefore should correspond to natural, independent components of motion. Filters on deeper layers express the presence of filters on lower layers. They therefore represent higher level combinations of the different components of motion.

4.8 Results

In this section we demonstrate some of the results of our method, including fixing corrupted motion data, filling in missing motion data, interpolation of two motions, and distance comparisons between motions. The readers are referred to the supplementary video for additional results.

Fixing Corrupt Motion Data Projection and inverse projection can be used to fix corrupted motion data. In Fig. 5.1, left, we artificially corrupt motion data by randomly setting values to zero with a probability of 0.5, removing half of the information. After projection onto the manifold the motion data is effectively recovered and matches closely to the ground truth.

We record motion data by simultaneously capturing motion with the Microsoft Kinect and a separate inertia based motion capture system. The Kinect capture contains many errors, but after projection onto the motion manifold it far more closely matches the ground truth data found from the inertia based motion capture system (see Fig. 5.1, center).

Filling in Missing Data Projection can be used to fill in missing motion data. We use our technique to automatically infer the marker position of a character's wrist (see Fig. 5.1, right). We can also produce some stepped animation data using only every 15 frames and project this onto the manifold. Our method inserts smooth transitions between the stepped keyframes, corresponding to motion found in the training data.

Motion Interpolation We interpolate two distinctly separate motions of *doing press ups* and *walking forward*. Compared to a linear interpolation, which creates an impossible walk, interpolation along the manifold creates motion which remains valid.

Motion Comparison Using Naive Euclidean distance a motion of a confident walk with a turn is found to be similar to some odd motions, such as a sidestepping motion and an old man's walk. By taking the min and max over the temporal dimension i of the Hidden Units and consequently taking Euclidean distance, we get a distance measure G which is temporally invariant and correctly matches high level features. It only finds motions with confident walks and turns to be similar.

$$\begin{aligned} G(\mathbf{X}_0, \mathbf{X}_1) = {} & \| \max_i(\mathbf{\Phi}(\mathbf{X}_0)) - \max_i(\mathbf{\Phi}(\mathbf{X}_1)) \|_2^2 + \\ & \| \min_i(\mathbf{\Phi}(\mathbf{X}_0)) - \min_i(\mathbf{\Phi}(\mathbf{X}_1)) \|_2^2 \end{aligned} \tag{4.4}$$

Computational Time Training is done using a Quadro K2000 graphics card with 2GB of memory. Training the full network takes 5 hours and 4 minutes. Where Layer 1 takes 43 Minutes, Layer 2 takes 51 Minutes, Layer 3 takes 1 hour and 17 Minutes and fine tuning takes 2 hours and 13 Minutes. At runtime the system only requires the multiplication of several medium sized matrices and so projection can be done in a fraction of a millisecond.

Comparison with Other Approaches Our method scales well compared to other approaches. Many methods that can handle large amounts of data require preprocessing that data using data structures such as KD trees. This is a very computationally expensive process, especially for high dimensional data. Our approach can handle a very large amount of data in a reasonable amount of time. The only methods that requires little data preprocessing and are scalable are classic methods such as PCA. Although PCA is good in many aspects, it does not perform well when non-linearity plays an important role such as when interpolating motions of different style or computing distances between motion data considering synchronization. The readers are referred to the supplementary video for visual comparison.

Limitations & Future Work Currently the manifold projection cannot guarantee constraints such as ground contact and joint lengths. In future we wish to explicitly incorporate these into our system using a constrained projection operator. We also wish to thoroughly evaluate our work against alternative methods and justify the effectiveness of using a deep architecture.

4.9 Conclusion

Learning a motion manifold has many useful applications in character animation. We present an approach appropriate for learning a manifold on the whole CMU motion capture database and present a number of operations that can be performed using the manifold.

The strength of our approach is in the explicit formulation of the temporal domain of the manifold. The position of a single joint is not just linked to the position of other joints in the same pose - but to the position of other joints in separate frames of the animation. This makes our system more powerful and easier to use than approaches, which first attempt to first learn a manifold over the pose space and then augment it with a statistical model of transitions.

4.10 Postscript

In this chapter we presented a technique for the construction of a motion manifold. We also include definitions of a number of useful operations on this manifold including projection, distance, and interpolation. This manifold is useful for many applications in computer animation because it represents a strong prior belief about motion. In this way it provides a basis for further work to replace many heuristic methods of data-driven character animation with counterparts that instead learn as much as possible from the data provided.

In the next chapter we present a number of extensions to this work, and show new results which make use of the motion manifold. More specifically, we show how it can be applied to the synthesis of new motion and the natural editing of existing motion.

Chapter 5

Synthesis and Editing

5.1 Preface

In this chapter we build on previous work utilising the motion manifold by developing a deep learning framework which can perform synthesis and editing of motion data in the space of the motion manifold. Automatic motion synthesis and editing can save animators from manually producing keyframed animation by providing a strong initial motion which can be further edited. We present a number of state of the art results which, unlike previous techniques, require no manual preprocessing of data or heuristics, and instead learn entirely from the database. As an example of our motion editing method we present results for a technique of style transfer which requires no alignment between motions.

Deep learning has seen great success in tasks such as classification and segmentation but it is only recently that researchers are finding ways to apply it to synthesis with results that are high quality enough to be used in production. Here we present a technique for motion synthesis and editing that produces high quality results, is extremely fast to run, and fits naturally into the design of tools used for keyframed animation. The rest of this chapter consists of a publication in SIGGRAPH 2016.

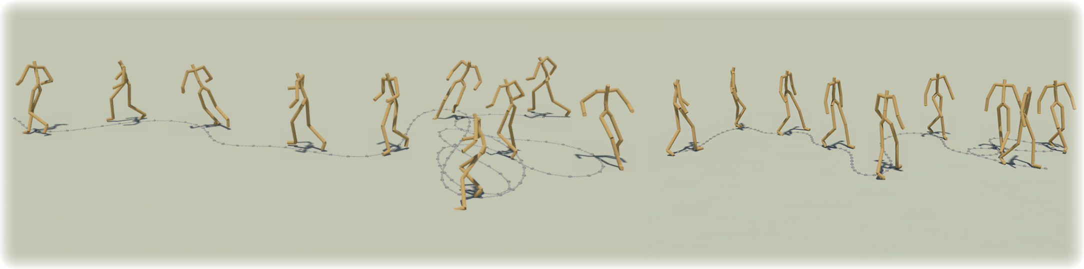

Figure 5.1: Our framework allows the animator to synthesize character movements automatically from given trajectories.

5.2 Introduction

Data driven motion synthesis allows animators to produce convincing character movements from high-level parameters. Such approaches greatly help animation production, as the animators only need to provide high level instructions rather than low level details through keyframes. Various techniques that make use of large motion capture dataset and machine learning to parameterize the motion are proposed in the area of computer animation.

The data-driven approaches currently available mostly require a significant amount of manual data preprocessing, including motion segmentation, alignment, labelling and parameterization. This makes full automation difficult and so often these systems require dedicated technical developers. A mistake in each stage can easily result in a failure of the final animation. Such preprocessing is therefore usually carefully done through a significant amount of human intervention, making sure the output movements appear smooth and natural.

In this paper, we propose a model of animation synthesis and editing based on a deep learning framework, which can automatically learn the embedding in a non-linear manifold from a large set of human motion data with no manual data preprocessing or human intervention. We train a convolutional autoencoder on a large motion database such that it can not only reproduce the motion data given as its input, but also synthesize novel motion that interpolates them. This unsupervised non-linear manifold learning process does not require any motion segmentation or alignment, which makes the process significantly easier than previous approaches. On top of this autoencoder, we stack another feedforward neural network that maps the high-level parameters to the low level human motion represented by hidden units of the autoencoder. With this, users can easily produce realistic human motion sequences from intuitive inputs such as a curve over a terrain that the body should follow, or the trajectory of the end effectors for punching and kicking. As the feedforward control network and the motion manifold are trained independently, users can easily swap and re-train the feedforward network according to the desired interface.

We also propose techniques to edit the motion data in the space of the motion manifold by constraining arbitrary body parts or applying a different style to the motion. The hidden units of the convolutional autoencoder represent the motion in a meaningful and natural fashion, such that adjusting the data in this space preserves the naturalness and smoothness of the motion, while still allowing complex movements of the body to be reproduced. One demonstrative example of this editing is shown by combining the style motion with the timing of another by minimizing the error in the Gram matrices of the hidden units of the synthesized motion and the style motion [Gatys et al., 2015].

In summary, our contribution is the following:

- A deep learning framework for synthesizing character animation from high-level parameters.
- A method of editing motion using the learned representation for satisfying user constraints and transforming motion style.

5.3 Related Work

We first review kernel-based methods for motion synthesis, which has been the main stream for synthesizing motions by blending motion capture data. We next review methods of interactive character control, where learned motions are applied for synthesizing novel motion data based on user instructions. Finally, we review methods in deep learning and how it is applied to character animation.

5.3.0.0.1 Kernel-based Methods for Motion Blending Radial-basis functions (RBFs) are effective for blending multiple motions of the same class. Rose et al. [Rose et al., 1998] call the motions of the same class as “verbs” and interpolate them using RBFs according to the direction that the character needs to move toward, or reach out. Rose et al. [Rose III et al., 2001] show an inverse kinematics usage of RBFs by mapping the joint positions to the posture of the character. For making the blended motion to appear plausible, the motions need to be categorized and aligned along the timeline. Kovar and Gleicher [Kovar and Gleicher, 2004] automates this process by computing the similarity of the movements and aligning the movements by dynamic time warping. However, RBF methods can easily overfit to the data due to the lack of mechanism to handle noise and variance. Mukai and Kuriyama [Mukai and Kuriyama, 2005] overcome this issue by using Gaussian Processes (GP), where the hyper-parameters are optimized to fit the model to the data. Grochow et al. [Grochow et al., 2004] applies Gaussian Process Latent Variable Model (GPLVM) to map the motion data to low dimensional space such that animators can intuitively control the characters. Wang et al. [Wang et al., 2008] applies GPLVM for time-series motion data for learning the posture in the next frame given the previous frames. Levine et al. [Levine et al., 2012] applies reinforcement learning in the space reduced by GPLVM to compute the optimal motion for tasks such as locomotion, kicking and punching.

Kernel-based methods such as RBF and GP suffer from large memory cost. Therefore, the number of motions that can be blended are limited. Our approach does not have such a limitation and can scale to huge sets of training data.

5.3.0.0.2 Interactive Character Control For interactive character control, a mechanism to produce a continuous motion based on the user input is needed. Motion graphs [Arikan and Forsyth, 2002, Lee et al., 2002, Kovar et al., 2002] is an effective data structure for such a purpose. Motion graphs are automatically computed from a large amount of motion capture data. Methods based on dynamic programming and reinforcement learning [Lee and Lee, 2004] are proposed to allow users to interactively control the characters. As motion graphs only replay captured motion data, techniques to interpolate motions in the same class are proposed to enrich the dataset [Min and Chai, 2012, Heck and Gleicher, 2007, Safonova and Hodgins, 2007, Shin and Oh, 2006, Levine et al., 2012]. Min et al. [Min and Chai, 2012] model the motions in the same class by PCA and the transitions by GP. Levine et al. [Levine et al., 2012] classifies the motion dataset to different classes in advance and reduces the dimensionality within each motion class. These methods require the motions to be classified, segmented and aligned for producing a rich model of the each motion class. Although Kovar and Gleicher [Kovar and Gleicher, 2004] tries to automate this process, the choice of the distance metric between motions and segmentation criteria of the motion sequence can significantly affect the performance of the accuracy. The requirement of explicitly conducting these steps can be a bottleneck of their performance. On the contrary, our unsupervised framework automatically blends motions in the proximity for synthesizing realistic movements, without any requirement of motion segmentation and classification.

5.3.0.0.3 Deep learning for Motion Data Techniques based on deep learning are currently the state-of-the-art in the area of image and speech recognition [Krizhevsky et al., 2012, Graves et al., 2013]. Recently, there is a huge interest in applying deep learning techniques for synthesizing novel data from the learned model [Vincent et al., 2010, Goodfellow et al., 2014]. One important feature of frameworks based on deep learning is that they automatically learn the features from the dataset. For example, when convolutional neural networks (CNN) are applied to image recognition, filters similar to Gabor filters appear at the bottom layers, and more complex filters that correspond to different objects appear in the higher level layers [Zeiler and Fergus, 2014]. One of our main interests is to make use of such a feature for character animation.

Deep learning and neural networks are also attracting interests of the control community for applications such robotics and physically-based animation. Allen and Faloutsos [Allen and Faloutsos, 2009] uses the NEAT algorithm, that evolves the topology of the neural network for controlling bipedal characters. Tan et al. [Tan et al., 2014] applies this technique for control of bike stunts. Levine and Vladlen [Levine and Koltun, 2014] learns the optimal control policies using neural networks and apply it for bipedal gait control. Mordatch et al. [Mordatch et al., 2015] applies the recurrent neural network to learn a near-optimal feedback controller for articulated characters. While these approaches learn the dynamics for controlling characters in physical environments, our focus is on learning features from captured human motion data and applying them for animation production.

There has been approaches to apply deep learning to human motion capture data. Du et al. [Du et al., 2015] construct a hierarchical RNN using a large motion dataset from various sources and show their method achieves the state-of-the-art recognition rate. Holden et al. [Holden et al., 2015b] applies the convolutional autoencoder to the CMU motion capture database and shows the learned representation shows good performance for retrieving similar motions in the database. These research show that neural networks can learn a good motion model from a large dataset of motion.

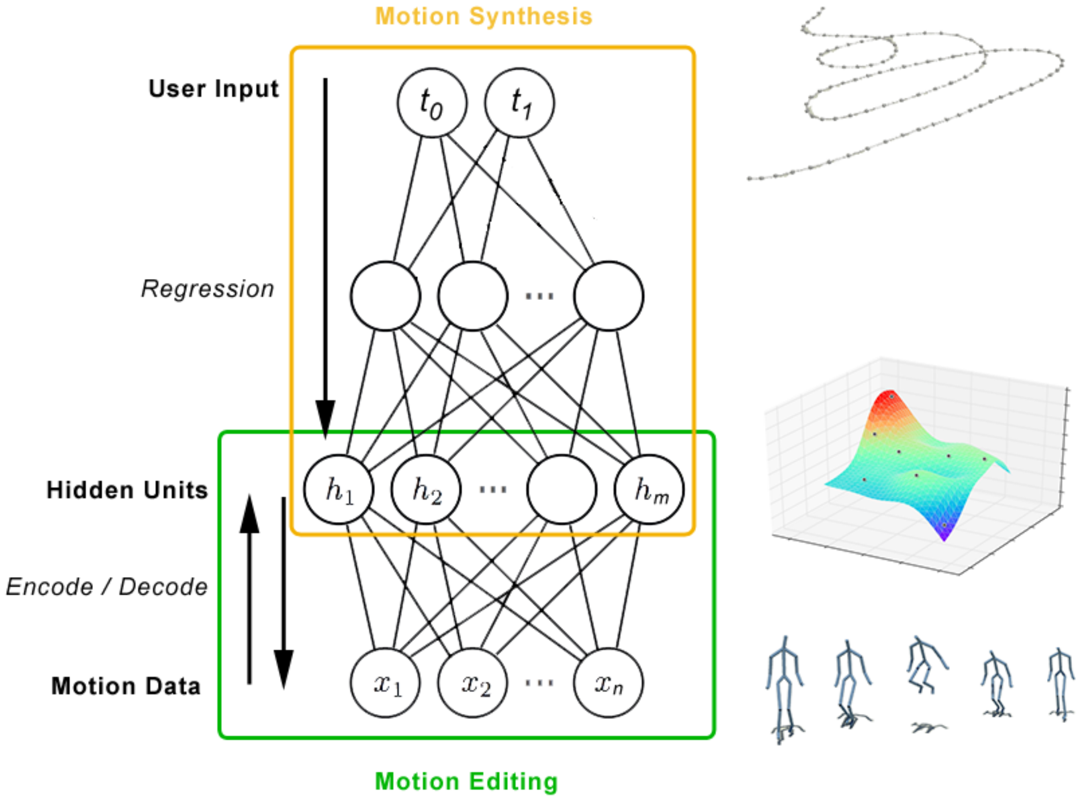

Figure 5.2: The outline of our method.

In terms of motion synthesis, Taylor et al. [Taylor and Hinton, 2009, Taylor et al., 2011] applies Restricted Boltzmann Machines (RBM) for synthesizing gait animation. Mittelman et al. [Mittelman et al., 2014] uses the spike-and-slab version of the recurrent temporal RBM and improves the reconstruction. These are timeseries approaches, where computing the entire motion requires integration from the first frame. We find this framework is not very suitable for the purpose of animation production as animators wish to see the entire motion at once and then sequentially apply minor edits while revising the motion. They do not wish to see edits happening at some frames to be propagated to the future, which will be the case when approaches based on time series is adopted. For this reason, in this paper we adopt and improve on the convolutional autoencoder representation [Holden et al., 2015b] which can produce motion at once, without performing some integration process.

5.4 System Overview

The outline of the system is shown in Fig. 6.3. Using data from the motion database (see Section 5.5), a convolutional autoencoder is trained and a motion manifold is learned (green box in Fig. 6.3, see Section 5.6). After this training, the motion is represented by the hidden units of the network. Given this representation, a mapping is produced between the user input and hidden units using a feedforward neural network stacked on top of the convolutional autoencoder (orange box in Fig. 6.3, see Section 5.7). Using this mapping, the user can produce an animation of the character walking and running by drawing a curve on the terrain, or the user can let the character punch and kick by specifying the target location in the 3D space. Once motion has been generated it can be edited in the space of hidden space, such that the resulting motion satisfies constraints and remains natural (see Section 5.8.1). Using this technique we describe a method to transform the style of the character motion using a short stylized clip as a reference (Section 5.8.2).

5.5 Data Acquisition

In this section we describe our construction of a large motion database suitable for deep learning.

5.5.1 The Motion Dataset for Deep Learning

We construct a motion dataset for our deep learning purpose by collecting many freely available large online databases of motion capture [Cmu, , Müller et al., 2007, Ofli et al., 2013, Xia et al., 2015], as well as adding data from our internal captures, and retargeting them to a uniform skeleton structure with a single scale and the same bone lengths. The retargeting is done by first copying any corresponding joint angles in the source skeleton structure to the target skeleton structure, then scaling the source skeleton to the same size as the target skeleton, and finally performing a Full Body Inverse Kinematics scheme [Buss, 2004] to move the joints of the target skeleton to match any corresponding joint positions in the source skeleton. Once constructed, the final data set is about twice the size of the CMU motion capture database and contains around six million frames of high quality motion capture data sampled at 120 frames per second.[1]

5.5.2 Data Format for Training

We convert the dataset into a format that is suitable for training the neural network. We subsample all of the motion in the database to 60 frames per second and convert the data into the 3D joint position format from the joint angle representation in the original dataset. The joint positions are defined in the body's local coordinate system whose origin is on the ground where root position is projected onto. The forward direction of the body (Z-axis) is computed by computing the vectors across the left and right shoulders and hips, averaging them and computing the cross product with the vertical axis (Y-axis). The global velocity in the XZ-plane, and rotational velocity of the body around the vertical axis (Y-axis) in every frame is appended to the input representation. These can be integrated over time to recover the global translation and rotation of the motion. Foot contact labels are found by detecting when either the toe or heel of the character goes below a certain height and velocity [Lee et al., 2002], and are also appended to input representation. The mean pose is subtracted from the data and the joint positions are divided by the standard deviation to normalize the scale of the character. The velocities, and contact labels are also divided by their own standard deviations.

In general, our model does not require motion clips to have a fixed length, but having a fixed window size can improve training speed, so for this purpose we separate the motion into overlapping windows of N frames (overlapped by $N/2$ frames), where $N = 240$ in our experiments. This results in a final input vector, representing a single sample from the database, as $\mathbf{X} \in \mathbb{R}^{N \times d}$ with N being the window size and d the degrees of freedom of the body model, which is 70 in our experiments.

[1] This dataset as well as the script for preprocessing the data will be made available for other researchers interested in deep learning research, given the permission from the providers of the original datasets.

5.6 Building the Motion Manifold

To construct a manifold over the space of human motion we build an auto-encoding convolutional neural network and train it on the complete motion database. We follow the approach by Holden et al. [Holden et al., 2015b], but adopt a different setup for optimizing the network for motion synthesis. First, we only use a single layer for encoding the motion, as multiple layers of pooling/de-pooling of stacked auto-encoders can result in blurred motion after reconstruction. Using only one layer lacks the power to further abstract the low level features so this task is performed by the deep feedforward network described in Section 5.7. We also change several components to improve the performance of the network training and basis quality. The details are described below.

5.6.1 Network Structure

In our framework, the convolutional auto-encoder performs a one-dimension convolution over the temporal domain, independently for a number of filters. As the network is bi-directional, it provides a *forward operation* $\mathbf{\Phi}$ and a *backward operation* $\mathbf{\Phi}^{\dagger}$.

The *forward operation* consists of a convolution (denoted $*$) using weights matrix $\mathbf{W}_0 \in \mathbb{R}^{h_a \times d \times w_a}$, addition of a bias $\mathbf{b}_0 \in \mathbb{R}^{h_a}$, a max pooling operation $\mathbf{\Psi}$, and the nonlinear operation $ReLU(\mathbf{X}) = \max(\mathbf{X}, 0)$, where w_a is the temporal filter width and h_a is the number of hidden units in the auto-encoding layer, each set to 25 and 256 in this work:

$$\mathbf{\Phi}(\mathbf{X}) = ReLU(\mathbf{\Psi}(\mathbf{X} * \mathbf{W}_0 + \mathbf{b}_0)). \tag{5.1}$$

The max pooling operation $\mathbf{\Psi}$ returns the maximum value between each pair of consecutive hidden units on the temporal axis. This ensures that the learned basis focus on representative features, as well as expressing a degree of temporal invariance. We use the rectified linear operation *ReLU* instead of the common *tanh* operation, since its performance as an activation function was demonstrated by Nair and Hinton [Nair and Hinton, 2010].

The *backward operation* takes hidden units $\mathbf{H}$ as input, and consists of a subtraction of a bias $\mathbf{b}_0$ and convolution using the weights matrix $\tilde{\mathbf{W}}_0$, which is the weights matrix $\mathbf{W}_0$ flipped on the temporal axis and transposed on the axes the input degrees of freedom

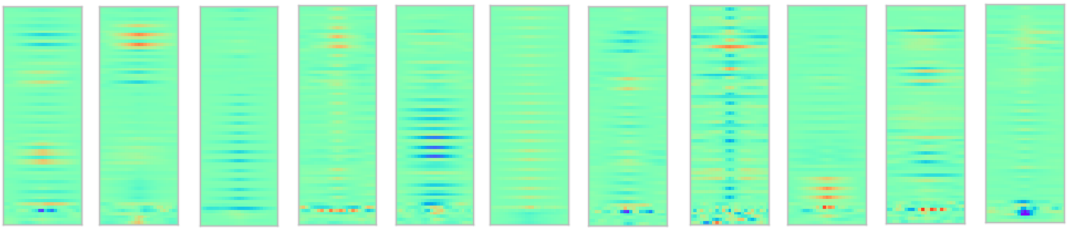

Figure 5.3: Some convolutional filters from the neural network. The horizontal axis represents time, while vertical axis represents the degrees of freedom.

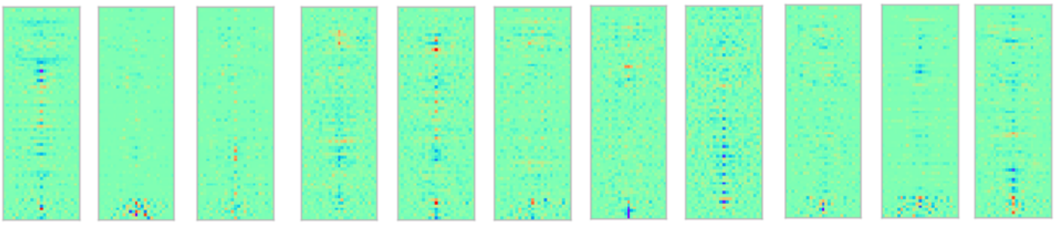

Figure 5.4: Convolutional filters learned by a network without dropout or max pooling. Compared to the filters in Fig. 5.3 these filters are noisy and have not learned any strong temporal correlations as most of their activations are located around the central frame of the filter.

and the number of hidden units, and an inverse-pooling operation $\mathbf{\Psi}^{\dagger}$:

$$\mathbf{\Phi}^{\dagger}(\mathbf{H}) = \mathbf{\Psi}^{\dagger}((\mathbf{H} - \mathbf{b}_0) * \tilde{\mathbf{W}}_0). \tag{5.2}$$

The inverse pooling operation $\mathbf{\Psi}^{\dagger}$ has two modes of operation. During training it randomly picks between the corresponding hidden units and assign the complete value of the input to that unit, leaving the other unit blank. This represents a good approximation of the inverse of the maximum operation but introduces noise into the result. Therefore when performing synthesis $\mathbf{\Psi}^{\dagger}$ acts like an average pooling operation and spreads the pooled value evenly across both hidden units.

The filter values $\mathbf{W}_0$ are initialised to some small random values found using the "fan-in" and "fan-out" criteria [Hinton, 2012], while the bias $\mathbf{b}_0$ are initialised to zero.

5.6.2 Training the Auto-Encoder

We train the network to reproduce some input $\mathbf{X}$ following both the *forward* and *backward* operations. Training is therefore given as an optimisation problem. We minimise the following cost function With respect to the parameters of the network. $\theta = \{\mathbf{W}_0, \mathbf{b}_0\}$.

$$Cost(\mathbf{X}, \theta) = ||\mathbf{X} - \mathbf{\Phi}^{\dagger}(\mathbf{\Phi}(\mathbf{X}))||_2^2 + \alpha \ ||\theta||_1 \tag{5.3}$$

In this equation $||\mathbf{X} - \mathbf{\Phi}^{\dagger}(\mathbf{\Phi}(\mathbf{X}))||_2^2$ measures the squared reproduction error and $||\theta||_1$ represents an additional sparsity term that ensures the minimum number of network parameters are used to reproduce the input. This is scaled by some small constant α which in our case is set to 0.01.

To minimize this function we perform stochastic gradient descent. We input random elements $\mathbf{X}$ from the database, and using automatic derivatives calculated via Theano [Bergstra et al., 2010], we update the network parameters θ. We make use of the adaptive gradient descent algorithm *Adam* [Kingma and Ba, 2014] to improve the training speed and quality of the final basis. To avoid overfitting we use a *Dropout* [Srivastava et al., 2014] of 0.2. Training is performed for 15 epochs and takes around 6 hours on a NVIDIA GeForce GTX 660 GPU.

Once training is complete the filters express strong temporal and inter-joint correspondences. A visualisation of the resulting weights can be seen in Fig. 5.3. Here it is shown that each filter expresses the movement of several joints over a period of time. In this way represent natural, independent components of motion.

5.7 Mapping User Inputs to Human Motions

We learn a regression between the user input and the character motion using a feedforward convolutional neural network. This is a general framework that can be applied for various types of user inputs and animation outputs. Producing a mapping between the low dimensional user inputs to the full body motion is a difficult task due to the huge amount of ambiguity and multi-modality in the output. There can be many different valid motions that could be performed to follow the input. For example, when the user wants to control a character to walk along a straight line, the timing of the motion is completely invariant: a character may walk with different step sizes or walk out of sync with another motion performed on the same trajectory. Naively mixing these out of sync motions results in an averaging of the output, making the character appear floating along the path. Unfortunately, there is no universal solution for solving such an ambiguity problem, and each problem must be solved individually based on the nature of the user input and the class of the output motion. Here we provide a solution for a locomotion task, which is a general problem with high demand.

In the rest of this section, we first describe about the structure of the feedforward network and how it can be trained, and then about the details of the user input system.

5.7.1 Structure of the Feedforward Network

We now describe about the feedforward convolutional network that maps the low dimensional user inputs $\mathbf{T}$ to the hidden layer of the auto-encoding network constructed in the previous section, such that eventually the system outputs a motion of the character $\mathbf{X} \in \mathbb{R}^{n \times d}$.

The feedforward convolutional network uses a similar *forward operation* as defined in Eq. (5.1) but contains three layers, and an additional operation $\boldsymbol{\Upsilon}$, which is a task-specific operation to resolve the ambiguity problem. We will discuss more about it in Section 5.7.3. The feedforward operation is given by the following.

$$\begin{aligned} \boldsymbol{\Pi}(\mathbf{T}) = ReLU(&\boldsymbol{\Psi}(ReLU(ReLU(\boldsymbol{\Upsilon}(\mathbf{T}) \\ &* \mathbf{W}_1 + \mathbf{b}_1) * \mathbf{W}_2 + \mathbf{b}_2) * \mathbf{W}_3 + \mathbf{b}_3)) \end{aligned} \tag{5.4}$$

where $\mathbf{W}_1 \in \mathbb{R}^{h_1 \times u_i \times w_1}$, $\mathbf{b}_1 \in \mathbb{R}^{h_1}$, $\mathbf{W}_2 \in \mathbb{R}^{h_2 \times h_1 \times w_2}$, $\mathbf{b}_2 \in \mathbb{R}^{h_2}$, $\mathbf{W}_3 \in \mathbb{R}^{h_a \times h_2 \times w_3}$, $\mathbf{b}_3 \in \mathbb{R}^{h_a}$, h_1, h_2 are the number of hidden units in the two hidden layers of the feedforward network, w_1, w_2, w_3 are the filter width of the three convolutional operators, and u_i is the DOF of the user input.

The parameters of this feed forward network used for regression are therefore given by $\phi = \{\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$.

5.7.2 Training the Feedforward Network

To train the regression between the user inputs and the output motion, we minimize a cost function using stochastic gradient descent in the same way as explained in Section 5.6.2, but this time with respect to the parameters of the feed forward network, keeping the parameters of the auto-encoding network fixed. The cost function is defined as the following and consists of two terms.

$$Cost(\mathbf{T}, \mathbf{X}, \phi) = ||\mathbf{X} - \boldsymbol{\Phi}^{\dagger}(\boldsymbol{\Pi}(\mathbf{T}))||_2^2 + \alpha\ ||\phi||_1 \tag{5.5}$$

The first term $||\mathbf{X} - \mathbf{\Phi}^{\dagger}(\mathbf{\Pi}(\mathbf{T}))||_2^2$ computes the mean squared error of the regression and the second term $\alpha\ ||\phi||_1$ is a sparsity term to ensure the minimum number of hidden units are used to perform the regression. As before α is set to 0.01.

When training this network we only use data relevant to the task. For example, during the locomotion task we only use data in the database that is locomotion data. Training therefore takes less time than the autoencoder. For the locomotion task training is done for 200 epochs and takes roughly 1 hour.

5.7.3 Disambiguation for Locomotion

In this section, we describe about our solution to disambiguate the locomotion, given a curve drawn on the terrain. A curve on a terrain alone does not give enough information to fully describe the motion that should be produced due to the ambiguity problem mentioned above. We examined various types of inputs, and discovered that providing the timing when the feet is in contact with the ground can greatly disambiguate the locomotion. Indeed, the contact timing distinguishes walking and running (there is always a double support phase in walking, and there is a flying phase in running). We therefore train a statistical model of foot contact information which can be used to automatically predict foot contacts from a given trajectory. We include this in the input to the feedforward network to resolve the ambiguity.

The input to this network is the trajectory in the form of translational velocities on the XZ plane and rotational velocity around the Y axis, given for each timestep and relative to the forward facing direction of the path $\mathbf{T} \in \mathbb{R}^{n \times 3}$, where n is the number of frames for the trajectory. The character height is considered constant. This input is passed to the function $\mathbf{\Upsilon}$ in Eq. (5.4), which adds foot contact information to the trajectory input:

$$\mathbf{\Upsilon}(\mathbf{T}) = [\mathbf{T}\ \ \mathbf{F}], \tag{5.6}$$

where $\mathbf{F} \in \{-1, 1\}^{n \times 4}$ is a matrix that represents the contact states of left heel, left toe, right heel, and right toe at each frame, and whose values are 1 when in contact with the ground, and -1 otherwise.

5.7.3.0.1 Modeling Contact States by Square Waves: We model the states of the four contacts using four square waves and learn the parameters of these waves from

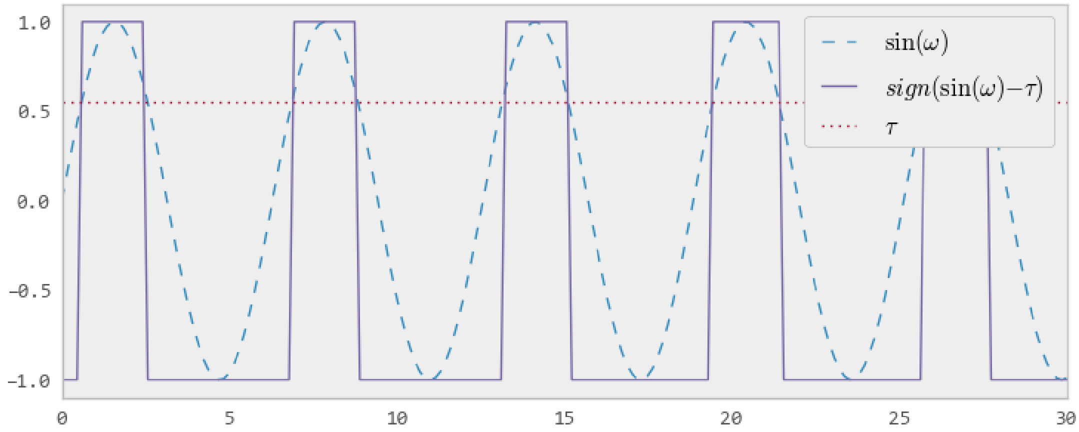

Figure 5.5: A square wave representing the foot contact computed from a sin wave. The foot is in contact with the ground when the value is 1.

the data in a way that allows us to compute them from the trajectory **T**. The four square waves that model the four contacts, and produce **F**, are defined as follows:

$$\mathbf{F}(\omega,\tau) = \begin{bmatrix} sign(\sin(\omega + a^h) - b^h - \tau^{lh}) \\ sign(\sin(\omega + a^t) - b^t - \tau^{lt}) \\ sign(\sin(\omega + a^h + \pi) - b^h - \tau^{rh}) \\ sign(\sin(\omega + a^t + \pi) - b^t - \tau^{rt}) \end{bmatrix}^T, \tag{5.7}$$

where a^h, a^t are user given constants that control the phases of the heels and toes, b^h, b^t are user given constants that can be used to bias the step size of the heels and toes - therefore forcing the character to walk or run, and ω, τ control the *frequency* and *step duration* at each frame of motion (see Fig. 5.5), and are the parameters we are interested in modelling. In our experiments, we set $a^h = -0.1, a^t = 0$, and b^h, b^t are set to zero. Below, we describe how to extract these parameters from the locomotion data.

5.7.3.0.2 Extracting Wave Parameters from Data: To produce a regression between the input curve **T** and parameters ω, τ, we need to first compute these parameter values from the foot contact information in the dataset. For each frame i the angle ω_i can be computed by summing the differential $\omega_i = \Delta\omega_i + \Delta\omega_{i-1} + ... + \Delta\omega_0$. We therefore calculate $\Delta\omega_i$ for each frame i in the data instead. From the dataset, $\Delta\omega_i$ can be computed by $\Delta\omega_i = \frac{\pi}{L_i}$ where L_i is the wavelength of the steps, and is computed by subtracting the timings of adjacent off-to-on frames, and averaging them for the four contact points (left/right, heel/toe). Learning $\mathbf{\Delta\omega}$ from the data instead of ω also allows for the footstep frequency to change during the locomotion, for example to allow the character to take more steps during a turn. We extract τ_i by looking at the foot contact information in the gait cycle that includes frame i and taking the ratio of the number of frames with the foot up u_i over the number of frames with the foot down d_i. This ratio can be converted to the square wave threshold value τ_i using the following.

$$\tau_i = \cos \frac{\pi d}{u + d} \tag{5.8}$$

For each heel and toe we learn separate τ variables, while $\boldsymbol{\Delta}\omega$ is the same between all contacts. This avoids feet going out of sync. These parameters are packed into a matrix $\boldsymbol{\Gamma} = \{\tau^{lh}, \tau^{lt}, \tau^{rh}, \tau^{rt}, \boldsymbol{\Delta}\omega\}$.

5.7.3.0.3 Regressing the Locomotion Path and Contact Information: Now we describe how we produce a regression between the input curve $\mathbf{T}$ and the contact square wave parameters $\boldsymbol{\Gamma}$. Using the locomotion data from the motion capture dataset, we compute the locomotion path $\mathbf{T}$ by projecting the root motion onto the ground. We also extract the corresponding $\boldsymbol{\Gamma}$, the foot contact parameters of the square waves, by the method described above. We then regress $\mathbf{T}$ to $\boldsymbol{\Gamma}$ using a small two layer convolutional neural network.

$$\boldsymbol{\Gamma}(\mathbf{T}) = \mathbf{W}_5 * ReLU(\mathbf{T} * \mathbf{W}_4 + \mathbf{b}_4) + \mathbf{b}_5 \tag{5.9}$$

Here $\mathbf{W}_4 \in \mathbb{R}^{64 \times 3 \times 65}$, $\mathbf{b}_4 \in \mathbb{R}^{64}$, $\mathbf{W}_5 \in \mathbb{R}^{5 \times 64 \times 45}$, $\mathbf{b}_5 \in \mathbb{R}^{5}$ are the parameters of this network. As with our other networks these parameters are found and this network is trained using stochastic gradient descent as explained in Section 5.6.2.

Once trained this network can, given some trajectory $\mathbf{T}$, predict values for $\tau, \boldsymbol{\Delta}\omega$, which can be used to calculate $\mathbf{F}$ using Eq. (5.7), therefore producing foot contact information for the trajectory.

5.8 Motion Editing in Hidden Unit Space

In this section, we describe how to edit or transform the style of the motion in the space of hidden units, which is the abstract representation of the motion data learned by the autoencoder. Because the basis learned by the convolutional auto-encoder are natural, when the motion is edited in the space of the motion manifold, it preserves its naturalness and smoothness. We represent constraints as costs with respect to the values of hidden units. This formulation of motion editing as a minimization problem is often convenient and powerful as it specifies the desired result of the edit without inferring any technique of achieving it. We first describe an approach to apply kinematic constraints (see Section 5.8.1) and then about adjusting the style of the motion in the space of hidden units (see Section 5.8.2).

5.8.1 Applying Constraints in Hidden Unit Space

Because the scope of motion editing is very large we start by describing how to apply constraints in the hidden space using two common constraints often found in character animation as examples: positional constraints and bone length constraints, but our approach is applicable to other types of constraints as well providing the constraint can be described using a the cost function.

5.8.1.0.1 Positional Constraints: Constraining the joint positions is essential for fixing foot sliding artifacts or guiding the end effector of the character such as the hand to grasp objects. Given an initial input motion in the hidden unit space $\mathbf{H}$, its cost in terms of penalty for violating the positional constraints is computed as follows:

$$Pos(\mathbf{H}) = \sum_j ||\mathbf{v}_r^{\mathbf{H}} + \omega^{\mathbf{H}} \times \mathbf{p}_j^{\mathbf{H}} + \mathbf{v}_j^{\mathbf{H}} - \mathbf{T}^{-1}\mathbf{v}'_j||_2^2. \tag{5.10}$$

where $\mathbf{v}'_j$ is the target velocity of joint j in the world coordinate system and $\mathbf{T}^{-1}$ is a transformation matrix to bring it to the body coordinate, and $\mathbf{v}_r^{\mathbf{H}}, \omega^{\mathbf{H}}, \mathbf{p}_j^{\mathbf{H}}, \mathbf{v}_j^{\mathbf{H}}$ are the root velocity, the body's angular velocity around the Y axis, joint j's position and velocity, respectively, computed from the hidden unit values $\mathbf{H}$ by the decoding operation $\mathbf{\Phi}^{\dagger}(\mathbf{H})$ in Eq. (5.2). For example, in order to avoid foot sliding, the heel and toe velocity must be zero when they are in contact with the ground.

5.8.1.0.2 Bone Length Constraints: As we use the joint positions as the representation in our framework, we need to impose the bone length constraint between the adjacent joints to preserve the rigidity of the body. The cost for such a constraint can be written as follows:

$$Bonelength(\mathbf{H}) \quad = \quad \sum_{b} ||||\mathbf{p}^{\mathbf{H}}_{b_{j1}} - \mathbf{p}^{\mathbf{H}}_{b_{j2}}|| - l_b|| \tag{5.11}$$

where b is the index for the set of bones in the body, l_b is the length of bone b, and $\mathbf{p}^{\mathbf{H}}_{b_{j1}}, \mathbf{p}^{\mathbf{H}}_{b_{j2}}$ are the reconstructed 3D positions of the two end joints of b by the decoding operation $\mathbf{\Phi}^{\dagger}(\mathbf{H})$ in Eq. (5.2).

5.8.1.0.3 Projection to Null Space of Constraints: The motion is optimized in the space of hidden units by stochastic gradient descent such that the total cost converges within a threshold:

$$\mathbf{H}' = arg \min_{\mathbf{H}} \; Pos(\mathbf{H}) + Bonelength(\mathbf{H}). \tag{5.12}$$

By minimising Eq. (5.12) and projecting the found $\mathbf{H}'$ back into the visual unit space we can constrain the joints to the desired position while keeping the rigidity of each bone.

5.8.2 Motion Stylization in Hidden Unit Space

Our framework for editing the motion in the hidden space can also be applied to transform the style of the motion using an example motion clip as a reference for the style. Gatys et al. [Gatys et al., 2015] describes that the *artistic style* of an image is encoded in the *Gram Matrix* of the hidden layers of a neural network and presents examples of combining the content of a photograph and the style of an artwork. By finding hidden unit values which produce a Gram matrix similar to the reference data, the input image can be adjusted to some different style, while retaining the original content. We can use our framework to apply this technique to motion data and produce a motion that has the timing and content of one input, with the style of another.

The cost function in this case is defined by two terms relating to the *content* and *style* of the output. Given some motion data $\mathbf{C}$ which defines the content of the produced output, and another motion data $\mathbf{S}$ which defines the style of the produced output, the cost function over hidden units $\mathbf{H}$ is given as the following,

$$Style(\mathbf{H}) = s||G(\mathbf{\Phi}(\mathbf{S})) - G(\mathbf{H})||_2^2 + c||\mathbf{\Phi}(\mathbf{C}) - \mathbf{H}||_2^2 \tag{5.13}$$

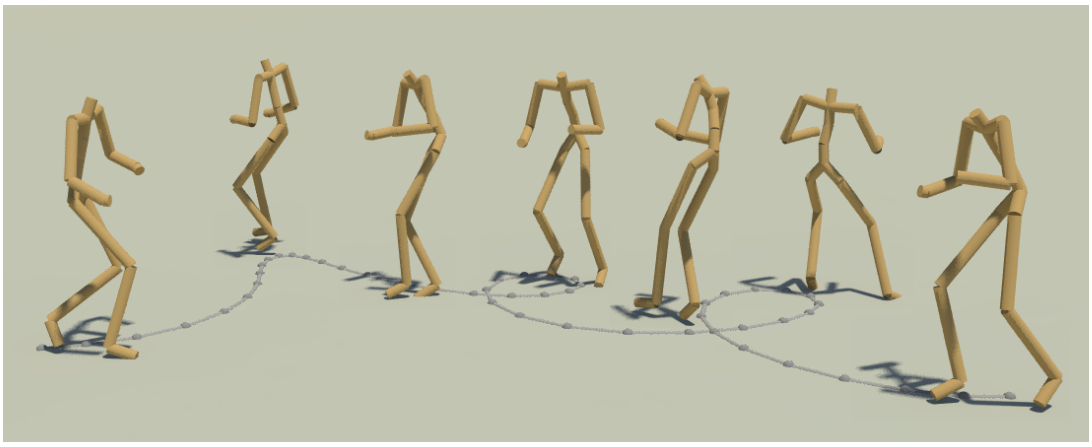

Figure 5.6: A side stepping motion produced from a velocity profile from a test data and an angular velocity profile drawn by Maya.

where c and s dictate the relative importance given to *content* and *style*, which are set to 1.0 and 0.01, respectively in our experiments, and the function G computes the Gram matrix, which is the mean of the inner product of the hidden unit values across the temporal domain i, and roughly corresponds to the average *similarity* or *co-activation* of the hidden units:

$$G(\mathbf{X}) = \frac{\sum_i^n \mathbf{X}_i \mathbf{X}_i^T}{n}. \tag{5.14}$$

To avoid a bias toward either *content* or *style*, $\mathbf{H}$ is initialised from white noise and a stylized motion is found by optimizing Eq. (5.13) until convergence using adaptive gradient descent with automatic derivatives calculated via Theano. The computed motion is then fixed by solving Eq. (5.12) to satisfy kinematic constraints.

5.9 Experimental Results

We now show some experimental results of training and synthesizing character movements. We first show examples of animating the character movements by the user input using the framework described in Section 5.7, with projection to the null space of constraints as described in Section 5.8.1. We next show examples of applying the stylization to the motions using the framework described in Section 5.8.2. As our system has a fast execution at runtime it is suitable for creating animation of large crowds. We therefore show such an example. We then evaluate the autoencoder representation by comparing its performance with other network structures. Finally we present

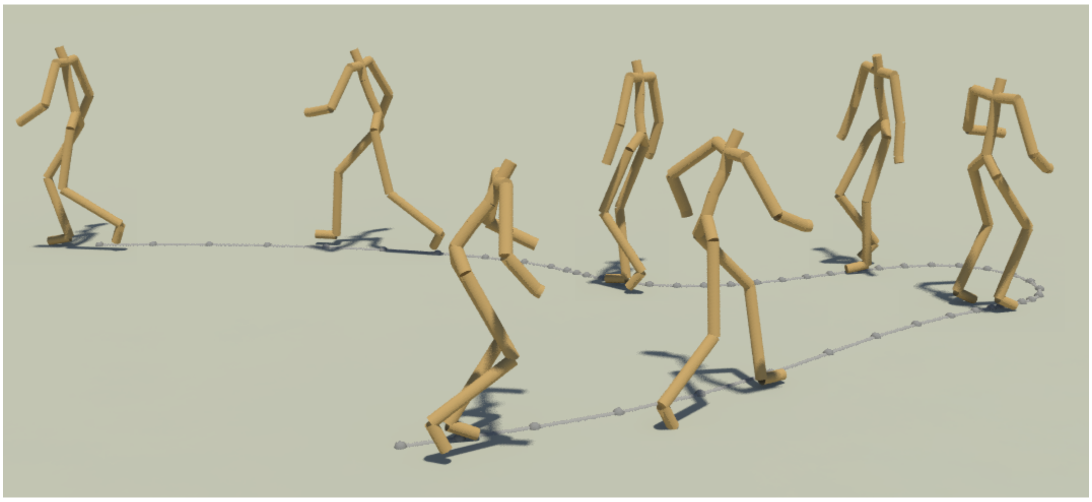

Figure 5.7: A locomotion including transition from walk to stop and run.

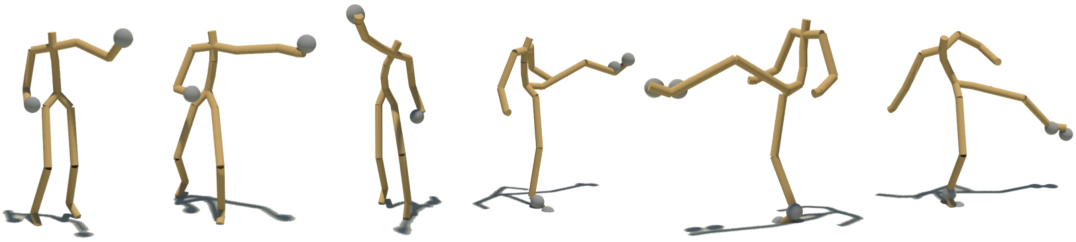

Figure 5.8: The character performs punching and kicking to attack the given targets.

a breakdown of the computation at the end of this section. The readers are referred to the supplementary video for the details of the animation.

5.9.0.0.1 Locomotion on the Terrain The feedforward network is trained such that a curve drawn on the terrain is used to generate the actual locomotion of the character. Among the data in the database, various types of locomotion data with different speed and stepping patterns are used to train the system. Using the training data, the trajectory of the root is projected onto the ground to produce a terrain curve to be used as an input. The high frequency components of the curve are removed by average filters such that the system does not overfit to the fine details of the curve.

During runtime, curves drawn by Maya are first used to produce walking and running animation. In the first two examples, the speed of the chracter is set constant, and the local angular velocity is set to 0. The timing that the heels and toes are in contact with the ground are automatically generated from the curve and used as the input for the

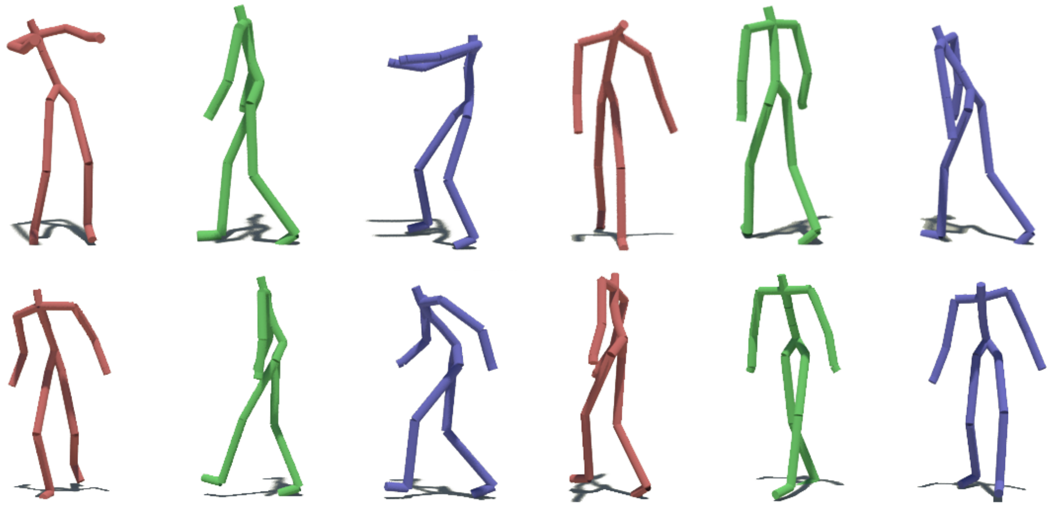

Figure 5.9: Several animations are generated with the timing from one clip and the style of. Red: input style motions. Green: input timing motions. Blue: output motion. In a clockwise order the styles used are zombie, depressed, old man, injured.

feedforward network. The character walks when the velocity is low and runs when the velocity is high (see Fig. 5.1). In the next example, we take the body velocity profile from some test set not used in the training. We also add some turning motion by drawing the angular velocity profile by Maya (see Fig. 5.6). In the final example, we use a speed profile from a test set where the character accelerates and decelerates. This is applied to a terrain curve drawn by Maya. A transition from walking to stopping and running appears as a result (see Fig. 5.7).

5.9.0.0.2 Punching and Kicking Control We show another example where the feedforward network is set up such that the character punches and kicks to follow end effector trajectories provided by the user. We tested the system using a test set not included in the training set. The character generates full body movements that follow the trajectories of the end effectors. Some snapshots of the animation are shown in Fig. 5.8.

5.9.0.0.3 Transforming the Style of the Motion We next show an example of transforming the style of the character's locomotion using a separate motion clip. Style data where the character walks in a (1) zombie style (2) depressed style (3) old man style and (4) injured style are given, and they are passed through the autoencoder to

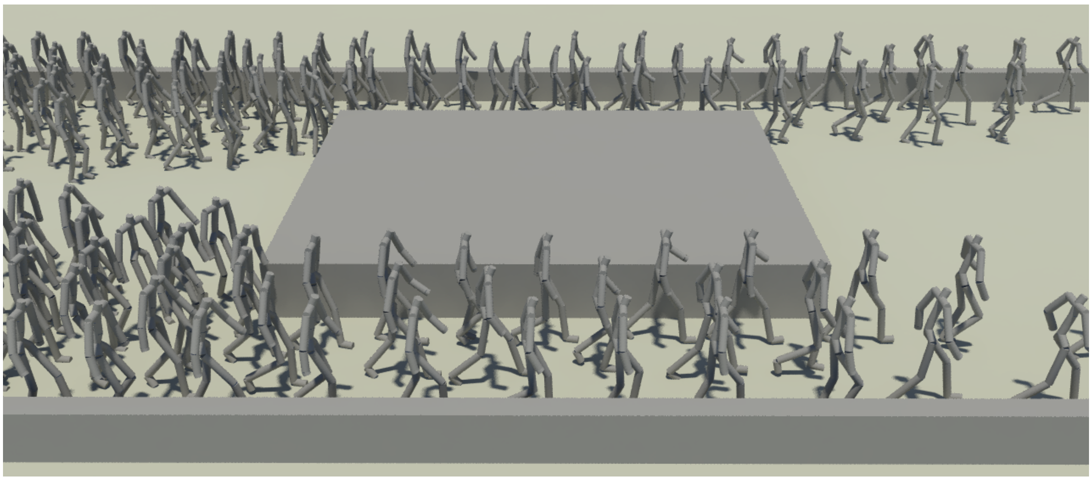

Figure 5.10: Crowd motion for 200 characters is generated in parallel using the GPU.

compute the Gram matrices. These Gram matrices are used for converting the style of the some given locomotion data using the optimization method described in Section 5.8.2. Locomotion data is taken from the data set, while style data is a combination of internal captures, and captures from [Xia et al., 2015]. The snapshots of the animation are shown in Fig. 5.9.

5.9.0.0.4 Crowd Animation Our approach allows for parallel computation across the timeline using the GPU. This allows us to create motion for many characters at once. We make use of this feature to apply our system for synthesizing an animation of a large set of characters using the terrain curve framework described in Section 5.7. Results of this are shown in Fig. 5.10.

5.9.0.0.5 Comparing the Autoencoder with Other Network Structures Here we evaluate the representation by the autoencoder. We compare it's performance to a naive construction of a neural network without max pooling or dropout. If trained without dropout or max pooling the network does not bother to learn strong temporal correspondences, as can be seen in Fig. 5.4. This motion manifold can therefore be thought of as fairly similar to a per-pose PCA. When applied to the style transfer task, because there is no temporal smoothness encoded in this model, the output is extremely noisy as shown in Fig. 5.11. In the motion synthesis task the neural network without max pooling or dropout actually had a slightly lower mean error, but similar noise was present in results. This can be seen in Fig. 5.12.

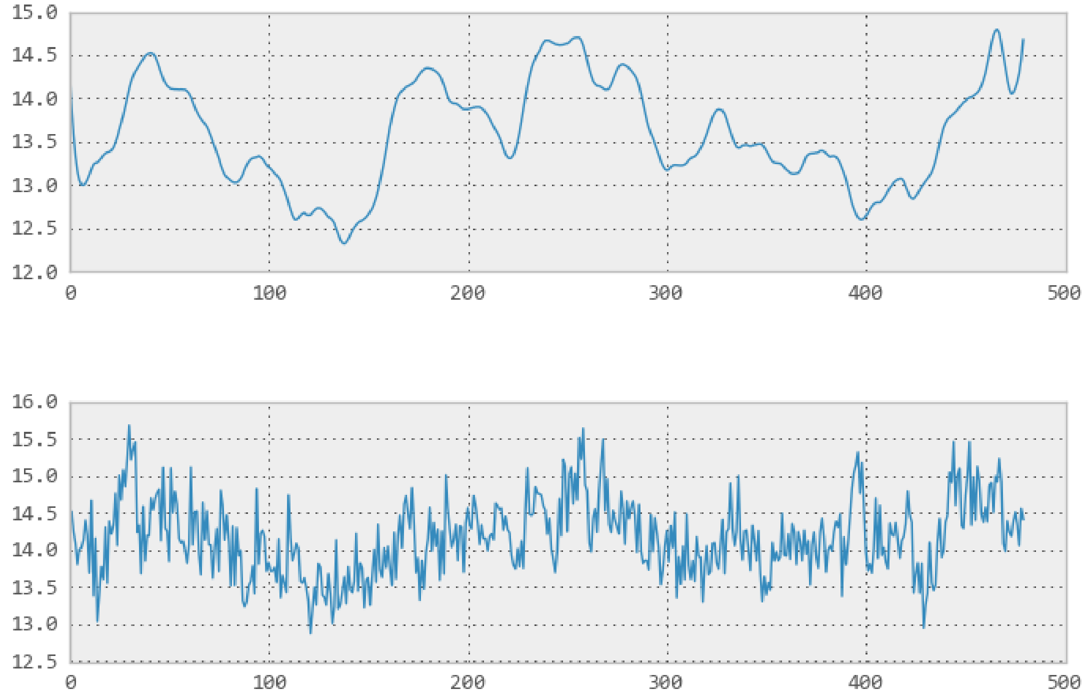

Figure 5.11: Two graphs of the vertical movement of the hand joint in motion generated in the style transfer task. Top: movement when the neural network uses dropout and max pooling - the movement is nice and smooth. Bottom: movement when the neural network does not use these operations - the movement is very noisy due to the lack of temporal correlation encoded in the motion manifold.

5.9.0.0.6 Breakdown of the Computation In this section we give a breakdown of the various timings of the computation for each result presented. This is shown in Fig. 5.13. All examples of generate animation were generated at a sample rate of 60fps. For the crowd scene the frame rate is given by finding the total number of frames generated across all 200 characters. All results were produced using a NVIDIA GeForce GTX 660 and *Theano*. Our technique clearly scales well as the total time required to generate results remains similar, even with large durations of animation or many characters. All times are given in seconds.

5.10 Discussions

Many other approaches to motion synthesis are time-series approaches [Taylor and Hinton, 2009, Taylor et al., 2011, Mittelman et al., 2014, Xia et al., 2015], but our approach to motion synthesis is a *procedural* approach as it does not require step-by-

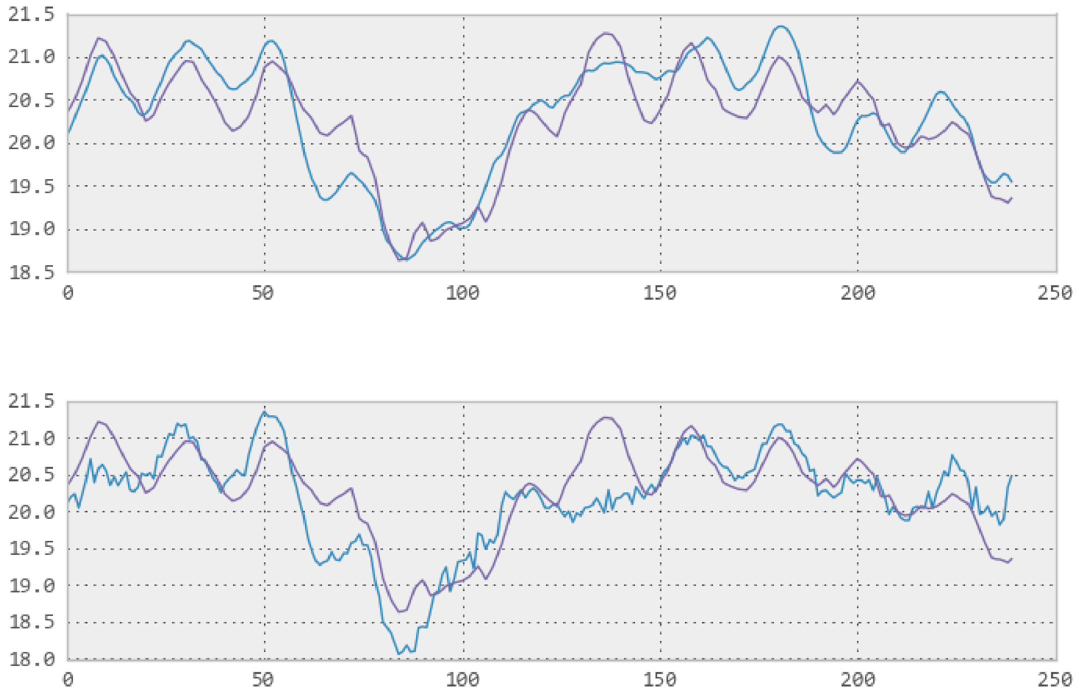

Figure 5.12: Two graphs of the vertical movement of the spine joint in motion generated in the motion synthesis task. Purple Line: Ground Truth. Blue Line: Generated Movement. Top: movement when the neural network uses dropout and max pooling - although the movement does not follow exactly, the signal is smooth. Bottom: movement when the neural network does not use these operations - the movement fits more closely to the ground truth but has high frequency noise.

step calculation, and individual frames at arbitrary times can be generated on demand. This makes it a good fit for animation production software such as *Maya* which allows animators to jump to arbitrary points in the timeline. Animators also do not need to worry about changes affecting the generated animation outside of the local convolution window. This also makes the system highly parallelisable, as motion for all frames can be computed independently. As a result, as presented in the experimental results, the trajectories of many characters can be generated during runtime on the GPU at fast rates.

Although procedural approaches are not new in character animation [Lee and Shin, 1999, Kim et al., 2009, Min et al., 2009], previous methods require the motion to be segmented, aligned and labelled before the data can be included into the model. On the contrary, our model automatically learns the model from a large set of motion data without manual labelling or segmentation. This makes the system highly practical as

Task	Duration	Foot Contacts	Synthesis	Editing	Total	FPS
Walking	60s	0.025s	0.067s	1.096s	1.188s	3030
Running	60s	0.031s	0.073s	1.110s	1.214s	2965
Punching	4s	-	0.019s	0.259s	0.278s	863
Kicking	4s	-	0.020s	0.302s	0.322s	745
Style Transfer	8s	-	-	2.234s	2.234s	214
Crowd Scene	10s	0.557s	1.335s	2.252s	4.144s	28957

Figure 5.13: Performance breakdown.

the users can easily add the new motion data into the training set to enrich the model.

Our convolutional filter only shifts along the temporal axis; a natural question to ask is if this convolution can also be used spatially, for example, over the graph structure of the character. The idea of using a temporal convolutional model is to ensure the learned basis of the autoencoder are local and invariant - that their influence is limited to a few frames, and that they can appear anywhere in the timeline. These assumptions of locality and invariance do not generalize well in the spatial domain. There are strong correlations between separate parts of the body according to the motion (for example, arms and legs synchronized during walking), and it is difficult to confine the influence along the graph structure. Also, the basis such as those for the arm are in general not applicable to other parts of the body, which shows the structure is not invariant. It is to be noted that our filters does capture the correlation of different joints; the signals of different DOFs are summed in the convolution operation in Eq. (5.1), and thus the filters are optimized to discover correlated movements.

5.10.0.0.1 Limitation In our framework, the input parameters of the feedforward network need to be carefully selected such that there is little ambiguity between the user input and the output motion. Ambiguity is a common issue in machine learning where the outputs of the regressor are averaged out when multiple outputs correspond to the same input in the training data. In some cases additional data that resolves the ambiguity may be required. This can either be supplied by the user directly, or must be sampled from an additional statistical model such is done with our foot contact model.

5.11 Conclusion

We propose a deep learning framework to map a user input to an output motion by first learning the motion prior using a large motion database and then producing a mapping between the user input to the output motion. We also propose approaches to edit and transform the styles of the motions under the same framework.

Currently, our autoencoder has only a single layer as deep stacked autoencoders suffer from blurriness during the depooling process. In our system, the role of combining and abstracting the low level features is covered by the feedforward network stacked on top of it. However, a more simple feedforward network, which is easier to train, can be used if a stacked deep autoencoder is used for learning a motion prior. It will be interesting to look into newly emerging depooling techniques such as hypercolumns [Hariharan et al., 2014] that cope with the blurring effect to produce a deep network for the motion prior.

5.12 Postscript

The framework shown in this chapter can be used to perform a large number of tasks required in character animation and requires no manual preprocessing of data. Additionally, a number of extensions may be possible to this work, including the deepening of the network, and more complex synthesis that includes interaction with the environment.

While the tool we developed in this chapter is a general framework, its design is particularly appropriate in keyframed animation enviroments because it is entirely time independant - animators can jump to random frames on the timeline and our system only needs to perform computation for a single frame of motion. Additionally the result of the synthesis is predictable and local - only relying on a small window over the input parameters. Finally our tool is extremely fast as it can be accelerated in parallel on the GPU.

Chapter 6

Character Control

6.1 Preface

Character control is the problem of generating realistic movement of a virtual character which follows some given user control. This problem is extremely common in video games, where the virtual character is often controlled by the movement of a gamepad joystick or some other form of input. In this case the difficulty is in producing motion which is responsive, accurate, high quality, and closely follows the given user input.

In this chapter we present a new method of character control for games influenced by work in previous chapters. Using a new neural network structure called a phase-functioned neural network we produce a character controller which can generate high quality motion in a number of complex situations such as climbing over rough terrain, crouching, and jumping over obstacles. As well as being able to learn from vast amounts of data, our controller is extremely compact once trained, requiring only a few megabytes of memory and milliseconds of execution time. The rest of this chapter consists of a publication in SIGGRAPH 2017.

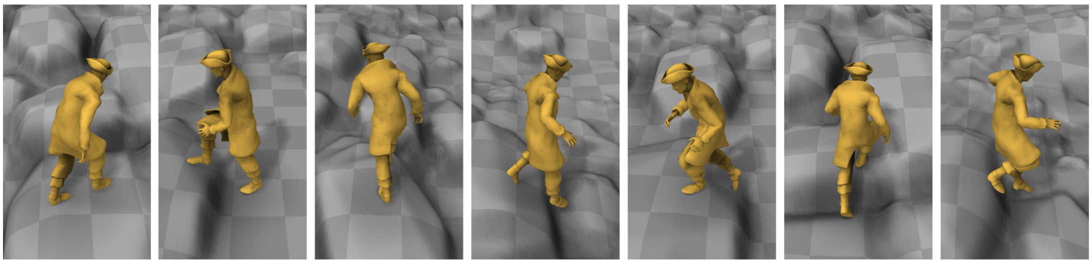

Figure 6.1: A selection of results using our method of character control to traverse rough terrain: the character automatically produces appropriate and expressive locomotion according to the real-time user control and the geometry of the environment.

6.2 Introduction

Producing real-time data-driven controllers for virtual characters has been a challenging task even with the large amounts of readily available high quality motion capture data. Partially this is because character controllers have many difficult requirements which must be satisfied for them to be useful - they must be able to learn from large amounts of data, they must not require much manual preprocessing of data, and they must be extremely fast to execute at runtime with low memory requirements. While a lot of progress has been made in this field almost all existing approaches struggle with one or more of these requirements which has slowed their general adoption.

The problem can be even more challenging when the environment is composed of uneven terrain and large obstacles which require the character to perform various stepping, climbing, jumping, or avoidance motions in order to follow the instruction of the user. In this scenario a framework which can learn from a very large amount of high dimensional motion data is required since there are a large combination of different motion trajectories and corresponding geometries which can exist.

Recent developments in deep learning and neural networks have shown some promise in potentially resolving these issues. Neural networks are capable of learning from very large, high dimensional datasets and once trained have a low memory footprint and fast execution time. The question now remains of exactly how neural networks are best applied to motion data in a way that can produce high quality output in real time with minimal data processing.

Previously some success has been achieved using convolutional models such as CNNs [Holden et al., 2016], autoregressive models such as RBMs [Taylor and Hinton, 2009], and RNNs [Fragkiadaki et al., 2015]. CNN models perform a temporally local transformation on each layer, progressively transforming the input signal until the desired output signal is produced. This structure naturally lends itself to an offline, parallel style setup where the whole input is given at once and the whole output is generated at once. In some situations such as video games this is undesirable as future inputs may be affected by the player's actions. RNNs and other autoregressive models [Taylor and Hinton, 2009, Fragkiadaki et al., 2015] are more appropriate for video games and online motion generation as they only require a single frame of future input, yet they tend to fail when generating long sequences of motion as the errors in their prediction are fed back into the input and accumulate. In this way autoregressive models tend to "die out" when frames of different phases are erroneously blended together or "explode" when high frequency noise is fed back into the system [Fragkiadaki et al., 2015]. Such artifacts are difficult to avoid without strong forms of normalization such as blending the output with the nearest known data point in the training data [Lee et al., 2010] - a process which badly affects the scalability of the execution time and memory usage.

We propose a novel neural network structure called a Phase-Functioned Neural Network (PFNN). The PFNN works by generating the weights of a regression network at each frame as a function of the phase - a variable representing timing of the motion cycle. Once generated, the network weights are used to perform a regression from the control parameters at that frame to the corresponding pose of the character. The design of the PFNN avoids explicitly mixing data from several phases - instead constructing a regression function which evolves smoothly over time with respect to the phase. Unlike CNN models, this network structure is suitable for online, real-time locomotion generation, and unlike RNN models we find it to be exceptionally stable and capable of generating high quality motion continuously in complex environments with detailed user interaction. The PFNN is fast and compact requiring only milliseconds of execution time and a few megabytes of memory, even when trained on gigabytes of motion capture data. Some of this compactness can additionally be traded for runtime speed via precomputation of the phase function, allowing for a customized trade off between memory and computational resources.

Dynamically changing the network weights as a function of the phase instead of keeping them static as in standard neural networks significantly increases the expressiveness of the regression while retaining the compact structure. This allows it to learn

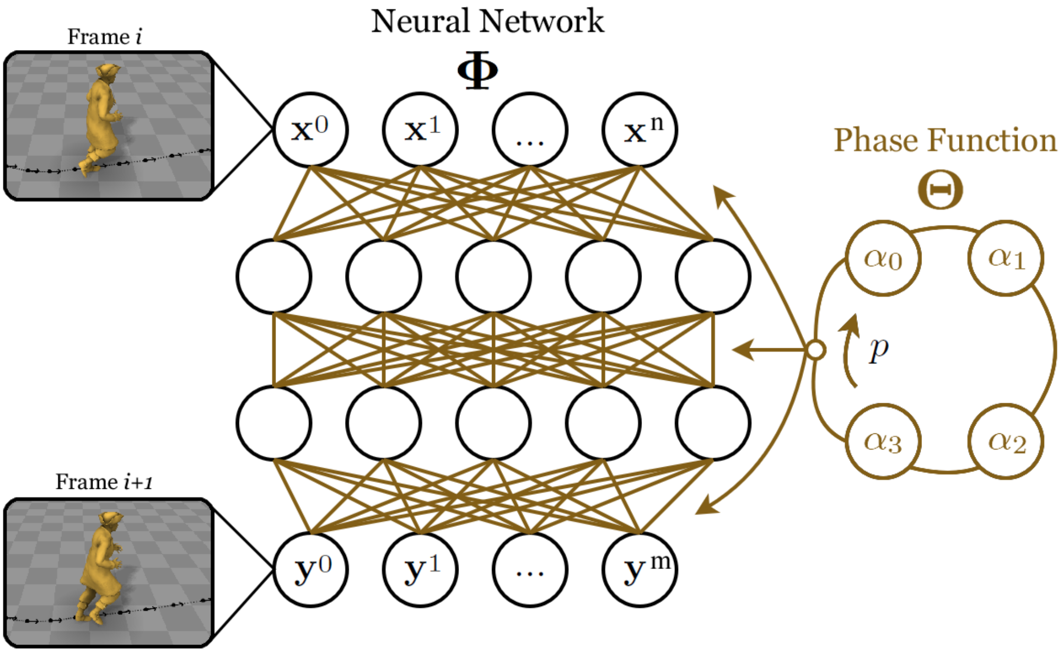

Figure 6.2: Visual diagram of Phase Functioned Neural Network. Shown in yellow is the cyclic *Phase Function* - the function which generates the weights of the regression network which performs the control task.

from a large, high dimensional dataset where environmental geometry and human motion data are coupled. Once trained, the system can automatically generate appropriate and expressive locomotion for a character moving over rough terrain and jumping, and avoiding obstacles - both in natural and urban environments (see Fig. 6.1 and Fig. 6.9). When preparing the training set we also present a process to fit motion capture data into a large database of artificial heightmaps extracted from video game environments.

In summary, the contribution of the paper is as follows:

- a novel real-time motion synthesis framework that we call the Phase-Functioned Neural Network (PFNN) that can perform character control using a large set of motion data including interactions with the environment, and

- a process to prepare training data for the PFNN by fitting locomotion data to geometry extracted from virtual environments.

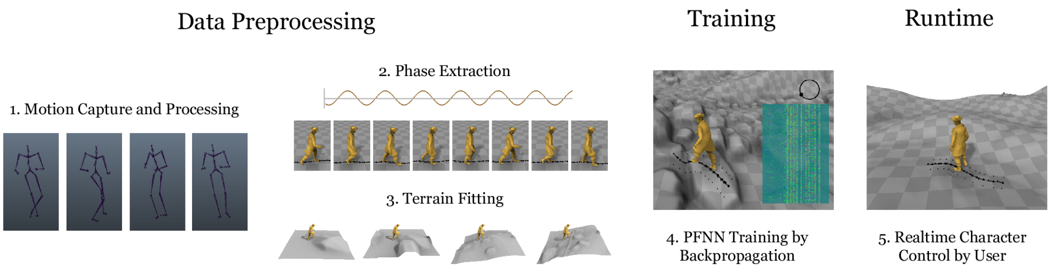
Data Preprocessing
Training
Runtime
1. Motion Capture and Processing
2. Phase Extraction
3. Terrain Fitting
4. PFNN Training by Backpropagation
5. Realtime Character Control by User

Figure 6.3: The three stages of the proposed system: In the data preprocessing stage (left), the captured motion is processed and control parameters are extracted. Next this data is fitted to heightmap data from a game engine. The PFNN is then trained by back propagation such that the output parameters can be generated from the input parameters (middle). Finally, during runtime, the character motion is computed on-the-fly given the user control and the environmental geometry (right).

6.3 Related Work

In this section, we first review data-driven approaches for generating locomotion. Next, we review methods for synthesizing character movements that interact with the environment. Finally, we review methods based on neural networks that focus on mapping latent variables to some parameters of the user's interest.

6.3.0.0.1 Data-driven locomotion synthesis: Data-driven locomotion synthesis is a topic that has attracted many researchers in the computer animation and machine learning community. Frameworks based on linear bases, kernel-based techniques, and neural networks have all been successfully applied for such a purpose.

Techniques based on linear bases such as principal component analysis (PCA) are widely adopted for reducing the dimensionality of motion data and also for predicting full body motion from a smaller number of inputs [Howe et al., 1999, Safonova et al., 2004]. As global PCA can have issues representing a wide range of movements in the low dimensional latent space, local PCA is adopted for handling arbitrary types of movements [Chai and Hodgins, 2005,Tautges et al., 2011]. Chai and Hodgins [Chai and Hodgins, 2005] apply local PCA for synthesizing full body motion with sparse marker sets. Tautges et al. [Tautges et al., 2011] produce similar local structures for predicting full body motion from sparse inertia sensors. Such structures require a significant amount of data preprocessing and computation both for training (i.e., motion segmentation, classification and alignment) and during run-time (i.e., nearest neighbor search).

Kernel-based approaches are proposed to overcome the limitations of linear methods and consider the nonlinearity of the data. Radial Basis Functions (RBF) and Gaussian Processes (GP) are common approaches for blending different types of locomotion [Rose et al., 1998, Park et al., 2002, Mukai and Kuriyama, 2005, Mukai, 2011]. Gaussian Process Latent Variable Models (GPLVM) are applied for computing a low dimensional latent space of the data to help solve the redundancy of inverse kinematics problems [Grochow et al., 2004] or to improve the efficiency for planning movements [Levine et al., 2012]. Wang et al. [Wang et al., 2008] propose a Gaussian Process Dynamic Model (GPDM) that learns the dynamics in the latent space and projects the motion to the full space using another GP. Kernel-based approaches suffer from the high memory cost of computing and inverting the covariance matrix, which scales in the square and cube order of the number of data points, respectively. Local GP approaches that limit the number of samples for interpolation are proposed to overcome this issue [Rasmussen and Ghahramani, 2002], but require $k-$nearest neighbor search which has a large memory usage and a high cost for precomputation and run-time when used with high dimensional data such as human movements.

Data-driven motion synthesis using neural networks is attracting researchers in both the computer animation and machine learning communities thanks to its high scalability and runtime efficiency. Taylor et al. [Taylor and Hinton, 2009] propose to use the conditional Restricted Boltzmann Machine (cRBM) for predicting the next pose of the body during locomotion. Fragkiadaki et al. [Fragkiadaki et al., 2015] propose an Encoder-Recurrent-Decoder (ERD) network that applies an LSTM model in the latent space for predicting the next pose of the body. These methods can be classified as autoregressive models, where the pose of the character is predicted based on the previous poses of the character during locomotion. Autoregressive models are suitable for real-time applications such as computer games as they update the pose of the character every frame. The cRBM and RNN models are more scalable and runtime-efficient than their classic linear [Xia et al., 2015] or kernel-based counterparts [Wang et al., 2008]. Despite such advantages, they suffer from drifting issues, where the motion gradually comes off the motion manifold due to noise and under-fitting, eventually converging to an average pose. Holden et al. [Holden et al., 2016] instead applies a CNN framework along the time domain to map low dimensional user signals to the full body motion. This is an offline framework that requires the full control signal along the time-line to be specified ahead of time for synthesizing the motion. Our framework in this paper is a time-series approach that can predict the pose of the character given the user inputs and the previous state of the character.

6.3.0.0.2 Interaction with the environment: Automatic character controllers in virtual environments that allow the character to avoid obstacles and adapt to terrains are useful for real-time applications such as computer games: these approaches can be classified into methods based on optimization and shape matching.

Methods based on optimization [Lau and Kuffner, 2005, Safonova and Hodgins, 2007], sampling-based approaches [Coros et al., 2008, Liu et al., 2010], maximum a posteriori probability (MAP) estimates [Min and Chai, 2012], and reinforcement learning techniques [Lee and Lee, 2004, Lo and Zwicker, 2008, Lee et al., 2010, Levine et al., 2012, Peng et al., 2016], predict the action of the character given the current state of the character (including the pose) and its relation to the geometry of the environment. They require cost/value functions that evaluate each action under different circumstances. Although it is shown that these methods can generate realistic movements, in some cases the computational cost is exponential with respect to the number of actions and thus not very scalable. More importantly, when using kinematic data as the representation, it is necessary to conduct $k-$nearest neighbour search within the samples [Lee et al., 2010, Clavet, 2016] to pull the motion onto the motion manifold. This can be a limiting factor in terms of scalability, especially in high dimensional space. Levine et al. [Levine et al., 2012] cope with such an issue by conducting reinforcement learning in the latent space computed by GPLVM but they require classification of the motion into categories and limit the search within each category. Peng et al. [Peng et al., 2016] apply deep reinforcement learning in the control space of physically-based animation in a way which can handle high dimensional state spaces. This is a very promising direction of research, but the system is only tested in relatively simple 2D environments. Our objective is to control characters in the full 3D kinematic space with complex geometric environments where previous learning based approaches have not been very successful.

Another set of approaches for controlling characters in a given environment is to conduct geometric analysis of the environments and adapt the pose or motion to the novel geometry. Lee et al. [Lee et al., 2006] conduct rigid shape matching to fit contact-rich motions such as sitting on chairs or lying down on different geometries. Grabner et al. [Grabner et al., 2011] conduct a brute-force search in order to discover locations in the scene geometry where a character can conduct a specific action. Gupta et al. [Gupta et al., 2011] produce a volume that occupies the character in various poses and fits this into a virtual Manhattan world constructed from a photograph. Kim et al. [Kim et al., 2014] propose to make use of various geometric features of objects to fit character poses to different geometries. Kang et al. [Kang and Lee, 2014] analyze the open volume and the physical comfort of the body to predict where the character can be statically located with each pose. Savva et al. [Savva et al., 2016] capture human interactions with objects using the Microsoft Kinect and produce statistical models which can be used for synthesizing novel scenes. These approaches only handle static poses and do not handle highly dynamic interactions. For animation purposes, the smoothness of the motions and the connection of the actions must be considered. Kapadia et al. [Kapadia et al., 2016] estimate the contacts between the body and the environment during dynamic movements and make use of them for fitting human movements into virtual environments. Their method requires an in-depth analysis of the geometry in advance which does not allow real-time interaction in new environments the character may face for the first time. Our technique is based on regression of the geometry to the motion and so can overcome these limitations of previous methods.

6.3.0.0.3 Mapping User Parameters to Latent Variables: In many situations people prefer to map scene parameters, such as viewpoints or lighting conditions in images, to latent variables such that users have control over them during synthesis. Kulkarni et al. [Kulkarni et al., 2015] propose a technique to map the viewpoint and lighting conditions of face images to the hidden units of a variational autoencoder. Memisavic [Memisevic, 2013] proposes a multiplicative network where the latent variables (viewpoint) directly parameterize the weights of the neural network. Such a network is especially effective when the latent parameter has a global effect on the entire output. We find this style of architecture is applicable for locomotion and use the phase as a common parameter between all types of locomotion, thus adopting a similar concept for our system.

6.4 System Overview

A visual diagram of the PFNN is shown in Fig. 6.2: it is a neural network structure (denoted by $\mathbf{\Phi}$) where the weights are computed by a periodic function of the phase p called the *phase function* (denoted by $\mathbf{\Theta}$). The inputs $\mathbf{x}$ to the network include both the previous pose of the character and the user control, while the outputs $\mathbf{y}$ include the change in the phase, the character's current pose, and some additional parameters descibed later.

There are three stages to our system: the preprocessing stage, training stage and the runtime stage. During the **preprocessing stage**, we first prepare the training data and automatically extract the control parameters that will later be supplied by the user (see Section 6.5). This process includes fitting terrain data to the captured motion data using a separate database of heightmaps (see Section 6.5.2). During the **training stage**, the PFNN is trained using this data such that it produces the motion of the character in each frame given the control parameters (see Section 6.6 for the setup of the PFNN and its training) . During the **runtime stage**, the input parameters to the PFNN are collected from the user input as well as the environment, and input into the system to determine the motion of the character (see Section 6.7).

6.5 Data Acquisition & Processing

In this section, we first describe about how we do the motion capture and extraction of control parameters which are used for training the system (see Section 6.5.1). We then describe about how we fit terrain to the motion capture data (see Section 6.5.2). Finally we summarize the parameterization of the system (see Section 6.5.3).

6.5.1 Motion Capture and Control Parameters

Once the motion data is captured, the control parameters, which include the phase of the gait, the semantic labels of the gait, the trajectory of the body, and the height information of the terrain along the trajectory, are computed or manually labeled. These processes are described below.

6.5.1.0.1 Motion Capture We start by capturing several long sequences of locomotion in a variety of gaits and facing directions. We also place obstacles, ramps and platforms in the capture studio and capture further locomotion walking, jogging and running over obstacles at a variety of speeds and in different ways. We additionally capture other varieties of locomotion such as crouching and jumping at different heights and with different step sizes. Once finished we have around 1 hour of raw motion capture data captured at 60 fps which constitutes around 1.5 GB of data. An articulated body model the same as the BVH version of the CMU motion capture data with 30 rotational joints is used with an additional root transformation added on the ground under the hips.

6.5.1.0.2 Gait Phase Next the *phase* must be labeled in the data, as this will be an input parameter for the PFNN. This can be performed by a semi-automatic procedure. Firstly, foot contact times are automatically labeled by computing the magnitude of the velocity of the heel and toe joints and observing when these velocities go below some threshold [Lee et al., 2002]. Since this heuristic can occasionally fail, the results are manually checked and corrected by hand. Once these contact times are acquired the phase can be automatically computed by observing the frames at which the right foot comes in contact with the ground and assigning a phase of 0, observing the frames when the left foot comes in contact with the ground and assigning a phase of π, and observing when the next right foot contact happens and assigning a phase of 2π. These are then interpolated for the inbetween frames. For standing motions some minimum cycle duration ($\sim 0.25s$) is used and the phase is allowed to cycle continuously.

6.5.1.0.3 Gait Labels We also provide semantic labels of the gait of the locomotion to the system represented as a binary vector. This is done for two reasons. Firstly they provide a method of removing ambiguity, since a fast walk and slow jog can often have the same trajectory, and secondly there are often times when the game designer or the user wishes to observe specific motions in specific scenarios (walking, jogging, jumping etc). This process is performed manually but can be greatly simplified by ensuring during capture that the actor does not change the gait type frequently and instead long captures are made containing only a single type of gait.

6.5.1.0.4 Body Trajectory and Terrain Height The root transformation of the character is extracted by projecting center of the hip joints onto the ground below. The facing direction is computed by averaging the vector between the hip joints and the vector between the shoulder joints, and taking the cross product with the upward direction. This direction is smoothed over time to remove any small, high-frequency movements. The trajectory of this root transformation is then extracted.

Once the trajectory of the root transformation has been extracted, and terrain has been fitted to the motion (see Section 6.5.2), the height of the terrain is computed at locations under the trajectory, as well as at locations either side of the trajectory, perpendicular to the facing direction and ~25cm away from the center point (see Fig. 6.5). More details about this process are described in Section 6.5.3.

Finally, once this process is completed, we create mirrored versions of all captures to double the amount of data.

6.5.2 Terrain Fitting

In order to produce a system where the character automatically adapts to the geometry of the environment during runtime, we need to prepare training data which includes the character moving over different terrains. Since simultaneous capture of geometry and motion in a motion capture studio is difficult, we instead present an offline process that fits a database of heightmaps gathered from video games or other virtual environments to separately captured motion data. Once fitted, these terrains allow us to use parameters relating to the geometry of the terrain as input to the control system.

Our database of heightmaps is extracted from several scenes built using the Source Engine. We trace rays into these scenes from above to capture the full geometric information in heightmap form with a resolution of one pixel per inch. From these heightmaps we randomly sample orientations and locations for ~20000 patches 3x3 meters in area. These patches are used in the fitting process.

The fitting process takes place in two stages. Firstly, for each locomotion cycle in the motion data we find the 10 best fitting patches in the database by searching in a brute-force way - attempting to fit every patch and selecting the patches which minimize a given error function. Secondly, we use a simple Radial Basis Function (RBF) mesh editing technique to refine the result, editing the terrain such that the feet of the

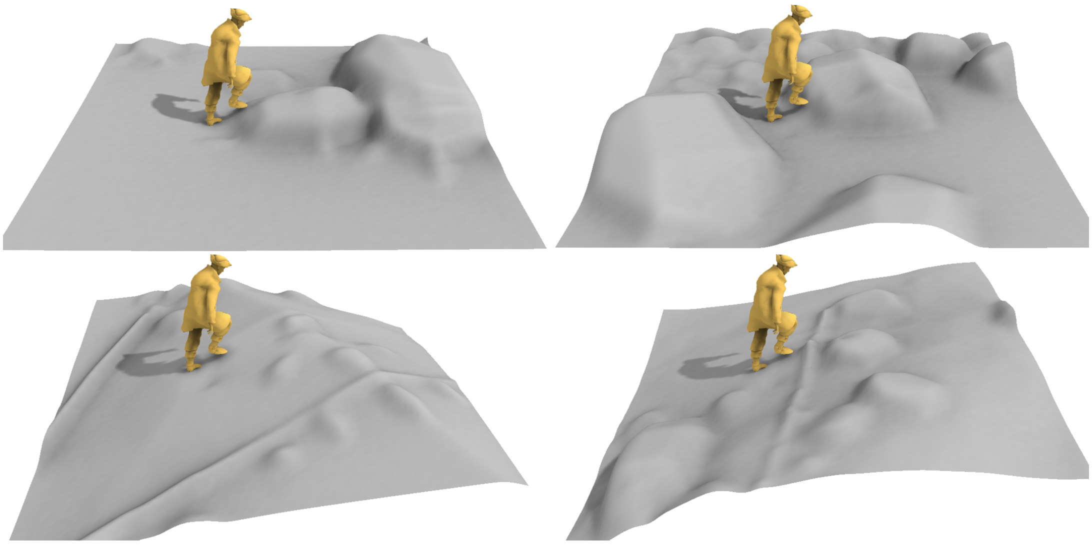

Figure 6.4: Results of fitting terrain to motion data. A variety of patches from the separately acquired terrain database are fitted to the same motion.

character are exactly on the ground during contact times.

Let us now describe the details of the fitting process. For each motion cycle in the database, given the left/right heel and toe joint indices $J \in \{lh\ rh\ lt\ rt\}$, we first compute for every frame i their heights $f_i^{lh}, f_i^{rh}, f_i^{lt}, f_i^{rt}$, and contact labels $c_i^{lh}, c_i^{rh}, c_i^{lt}, c_i^{rt}$ (a binary variable indicating if the joint is considered in contact with the floor or not). We find the average position of these joints for all the times they are in contact with the floor and center each patch in the database at this location. For each patch, we then compute the heights of the terrain under each of these joints $h_i^{lh}, h_i^{rh}, h_i^{lt}, h_i^{rt}$.

The fitting error E_{fit} is then given as follows:

$$E_{fit} = E_{down} + E_{up} + E_{over} \tag{6.1}$$

where E_{down} ensures the height of the terrain matches that of the feet when the feet are in contact with the ground,

$$E_{down} = \sum_i \sum_{j \in J} c_i^j (h_i^j - f_i^j)^2, \tag{6.2}$$

E_{up} ensures the feet are always above the terrain when not in contact with the ground (preventing intersections),

$$E_{up} = \sum_i \sum_{j \in J} (1 - c_i^j) \max(h_i^j - f_i^j, 0)^2, \tag{6.3}$$

and E_{over}, which is only activated when the character is jumping (indicated by the variable g^{jump}), ensures the height of the terrain is no more than l in distance below the feet (in our case l is set to ~30cm). This ensures large jumps are fitted to terrains with large obstacles, while small jumps are fitted to terrains with small obstacles.

$$E_{over} = g^{jump} \sum_{i} \sum_{j \in J} (1 - c_i^j) \max((f_i^j - l) - h_i^j, 0)^2. \tag{6.4}$$

Once we have computed the fitting error E_{fit} for every patch in the database we pick the 10 with the smallest fitting error and perform the second stage of the fitting process. In this stage we edit the heightmap such that the feet touch the floor when they are in contact. For this we simply deform the heightmap using a simplified version of the work of Botsch and Kobbelt [Botsch and Kobbelt, 2005]. We apply a 2D RBF to the residuals of the terrain fit using a linear kernel. Although we use this method, any other mesh editing techniques should also be appropriate as the editing required in almost all cases is quite minor.

Total data processing and fitting time for the whole motion database is around three hours on an Intel i7-6700 3.4GHz CPU running single threaded. Fig. 6.4 visualizes the results of this fitting process.

6.5.3 System Input/Output Parameters

In this section, we describe about the input/output parameters of our system. For each frame i our system requires the phase p for computing the network weights. Once these are computed the system requires neural network input $\mathbf{x}_i$ which includes the user control parameters, the state of the character in the previous frame, and parameters of the environment. From this it computes $\mathbf{y}_i$ which includes the state of the character in the current frame, the change in phase, the movement of the root transform, a prediction of the trajectory in the next frame, and contact labels for the feet joints for use in IK post-processing.

Now let us describe the details of the input parameters $\mathbf{x}_i$. Our parameterization is similar to that used in Motion Matching [Clavet, 2016] and consists of two parts. For the state of the character we take the positions and velocities of the character's joints local to the character root transform. For the user control we look at a local window centered at frame i and examine every tenth surrounding frame which, in our case,

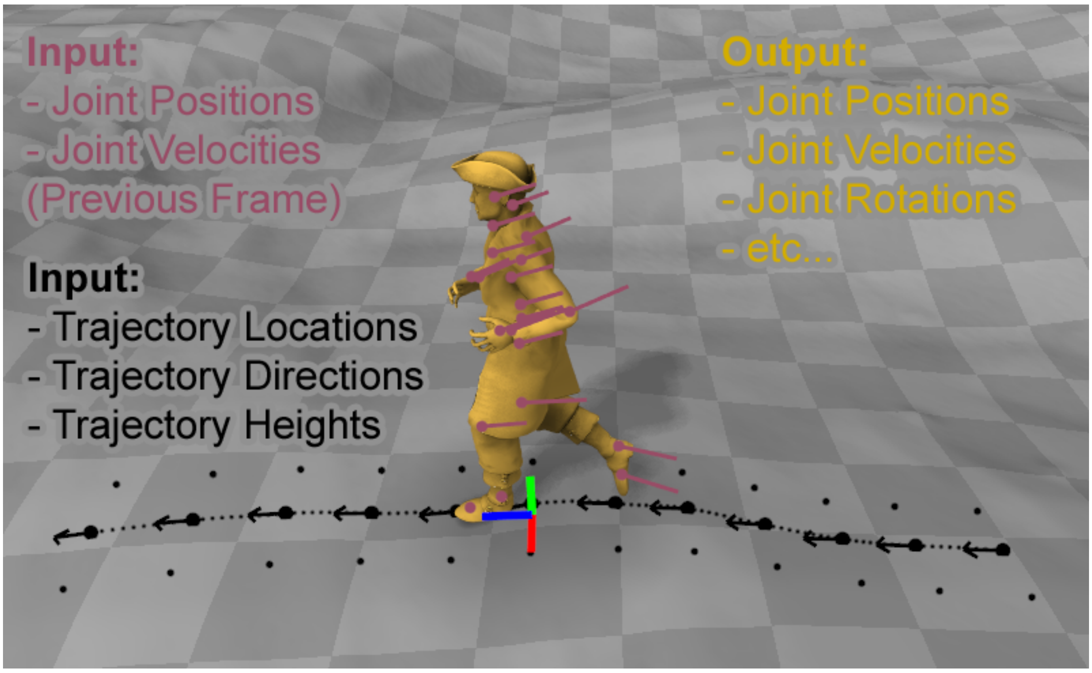

Figure 6.5: A visualization of the input parameterization of our system. In pink are the positions and velocities of character's joints from the previous frame. In black are the subsampled trajectory positions, directions and heights. In yellow is the mesh of the character, deformed using the joint positions and rotations output from our system.

produces $t = 12$ sampled surrounding frames covering 1 second of motion in the past and 0.9 seconds of motion in the future. From each surrounding sampled frame we extract a number of features including the character trajectory position and trajectory direction local to the character root transform at frame i, the character's gait represented as a binary vector, and the heights of the terrain under the trajectory and at two additional points 25cm away to the left and right of the trajectory. See Fig. 6.5 for a visual demonstration of this parameterization.

The full parameterization of the input control variables for a single frame i consists of a vector $\mathbf{x}_i = \{\ \mathbf{t}^p_i\ \mathbf{t}^d_i\ \mathbf{t}^h_i\ \mathbf{t}^g_i\ \mathbf{j}^p_{i-1}\ \mathbf{j}^v_{i-1}\ \} \in \mathbb{R}^n$ where $\mathbf{t}^p_i \in \mathbb{R}^{2t}$ are the subsampled window of trajectory positions in the 2D horizontal plane relative to frame i, $\mathbf{t}^d_i \in \mathbb{R}^{2t}$ are the trajectory directions in the 2D horizontal plane relative to frame i, $\mathbf{t}^h_i \in \mathbb{R}^{3t}$ are the trajectory heights of the three left/right/center sample points relative to frame i with the mean height subtracted, $\mathbf{t}^g_i \in \mathbb{R}^{5t}$ are the trajectory semantic variables indicating the gait of the character and other information (represented as a 5D binary vector), $\mathbf{j}^p_{i-1} \in \mathbb{R}^{3j}$ are the local joint positions of the previous frame, and $\mathbf{j}^v_{i-1} \in \mathbb{R}^{3j}$ are the local joint velocities of the previous frame, where j is the number of joints (in our case 31). In this work, the individual components of the additional semantic variables $\mathbf{t}_i^g$ are active when the character is in the following situations: 1. standing, 2. walking, 3. jogging, 4. jumping, 5. crouching (also used to represent the ceiling height).

The full parameterization of the output variables for a frame i consists of a vector $\mathbf{y}_i = \{\ \mathbf{t}_{i+1}^p\ \mathbf{t}_{i+1}^d\ \mathbf{j}_i^p\ \mathbf{j}_i^v\ \mathbf{j}_i^a\ \dot{\mathbf{r}}_i^x\ \dot{\mathbf{r}}_i^z\ \dot{\mathbf{r}}_i^a\ \dot{p}_i\ \mathbf{c}_i\ \} \in \mathbb{R}^m$ where $\mathbf{t}_{i+1}^p \in \mathbb{R}^{2t}$ are the predicted trajectory positions in the next frame, $\mathbf{t}_{i+1}^d \in \mathbb{R}^{2t}$ are the predicted trajectory directions in the next frame, $\mathbf{j}_i^p \in \mathbb{R}^{3j}$ are the joint positions local to the character root transform, $\mathbf{j}_i^v \in \mathbb{R}^{3j}$ are the joint velocities local to the character root transform, $\mathbf{j}_i^a \in \mathbb{R}^{3j}$ are the joint angles local to the character root transform expressed using the exponential map [Grassia, 1998], $\dot{\mathbf{r}}_i^x \in \mathbb{R}$ is the root transform translational x velocity relative to the forward facing direction, $\dot{\mathbf{r}}_i^z \in \mathbb{R}$ is the root transform translational z velocity relative to the forward facing direction, $\dot{\mathbf{r}}_i^a \in \mathbb{R}$ is the root transform angular velocity around the upward direction, $\dot{p}_i \in \mathbb{R}$ is the change in phase, and $\mathbf{c}_i \in \mathbb{R}^4$ are the foot contact labels (binary variables indicating if each heel and toe joint is in contact with the floor).

6.6 Phase-Functioned Neural Network

In this section we discuss the construction and training of the Phase-Functioned Neural Network (PFNN). The PFNN (see Fig. 6.2) is a neural network with weights that cyclically change according to the *phase* value. We call the function which generates the network weights the *phase function*, which in this work is defined as a cubic Catmull-Rom spline for reasons outlined in Section 6.6.2. To begin we first describe the chosen neural network structure (see Section 6.6.1), followed by the phase function used to generate the weights for this network structure (see Section 6.6.2). Finally, we describe about the training procedure (see Section 6.6.3).

6.6.1 Neural Network Structure

Given input parameters $\mathbf{x} \in \mathbb{R}^n$, output parameters $\mathbf{y} \in \mathbb{R}^m$, and a single phase parameter $p \in \mathbb{R}$ described in Section 6.5.3 we start by building a simple three layer neural network $\mathbf{\Phi}$ as follows:

$$\mathbf{\Phi}(\mathbf{x};\boldsymbol{\alpha}) = \mathbf{W}_2\ \mathrm{ELU}(\ \mathbf{W}_1\ \mathrm{ELU}(\ \mathbf{W}_0\ \mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1) + \mathbf{b}_2, \tag{6.5}$$

where the parameters of the network $\boldsymbol{\alpha}$ are defined by $\boldsymbol{\alpha} = \{\mathbf{W_0} \in \mathbb{R}^{h\times n}, \mathbf{W_1} \in \mathbb{R}^{h\times h}, \mathbf{W_2} \in \mathbb{R}^{m\times h}, \mathbf{b_0} \in \mathbb{R}^{h}, \mathbf{b_1} \in \mathbb{R}^{h}, \mathbf{b_2} \in \mathbb{R}^{m}\}$. Here h is the number of hidden units used on each layer which in our work is set to 512 and the activation function used is the exponential rectified linear function [Clevert et al., 2015] defined by

$$\mathrm{ELU}(x) = \max(x,0) + \exp(\min(x,0)) - 1. \tag{6.6}$$

6.6.2 Phase Function

In the PFNN, the network weights $\boldsymbol{\alpha}$ are computed each frame by a separate function called the *phase function*, which takes as input the phase p and parameters $\boldsymbol{\beta}$ as follows: $\boldsymbol{\alpha} = \boldsymbol{\Theta}(p;\boldsymbol{\beta})$. Theoretically, there are many potential choices for $\boldsymbol{\Theta}$. For example $\boldsymbol{\Theta}$ could be a another neural network, or a Gaussian Process, but in this work we choose $\boldsymbol{\Theta}$ to be a cubic Catmull-Rom spline.

Using a cubic Catmull-Rom spline is good for several reasons - it is easily made cyclic by letting the start and end control points be the same, the number of parameters is proportional to the number of control points, and it varies smoothly with respect to the input parameter p. Choosing a cubic Catmull-Rom spline to represent the phase function means each control point $\boldsymbol{\alpha}_k$ represents a certain configuration of weights for the neural network $\boldsymbol{\alpha}$, and the function $\boldsymbol{\Theta}$ performs a smooth interpolation between these neural network weight configurations. Another way to imagine $\boldsymbol{\Theta}$ is as a one-dimensional cyclic manifold in the (high-dimensional) weight space of the neural network. This manifold is then parameterized by the phase, and in this sense training the network is finding an appropriate cyclic manifold in the space of neural network weights which performs the regression from input parameters to output parameters successfully.

We found a cyclic spline of only four control points was enough to express the regression required by this system. Given four control points (consisting of neural network weight configurations) $\boldsymbol{\beta} = \{\boldsymbol{\alpha}_0\ \boldsymbol{\alpha}_1\ \boldsymbol{\alpha}_2\ \boldsymbol{\alpha}_3\}$, the cubic Catmull-Rom spline function $\boldsymbol{\Theta}$ which produces network weights for arbitrary phase p can be defined as follows:

$$
\begin{aligned}
\boldsymbol{\Theta}(p;\boldsymbol{\beta}) = \quad & \boldsymbol{\alpha}_{k_1} \\
& + w\,(\tfrac{1}{2}\boldsymbol{\alpha}_{k_2} - \tfrac{1}{2}\boldsymbol{\alpha}_{k_0}) \\
& + w^2(\ \boldsymbol{\alpha}_{k_0} - \tfrac{5}{2}\boldsymbol{\alpha}_{k_1} +\ 2\boldsymbol{\alpha}_{k_2} - \tfrac{1}{2}\boldsymbol{\alpha}_{k_3}) \\
& + w^3(\tfrac{3}{2}\boldsymbol{\alpha}_{k_1} - \tfrac{3}{2}\boldsymbol{\alpha}_{k_2} + \tfrac{1}{2}\boldsymbol{\alpha}_{k_3} - \tfrac{1}{2}\boldsymbol{\alpha}_{k_0}) \\
w = & \frac{4p}{2\pi} \quad (\mathrm{mod}\ 1) \\
k_n = & \left\lfloor \frac{4p}{2\pi} \right\rfloor + n - 1 \quad (\mathrm{mod}\ 4).
\end{aligned}
\tag{6.7}
$$

6.6.3 Training

For each frame i the variables $\mathbf{x}_i$, $\mathbf{y}_i$, and the phases p_i are stacked into matrices $\mathbf{X} = [\ \mathbf{x}_0\ \mathbf{x}_1\ ...\ ]$, $\mathbf{Y} = [\ \mathbf{y}_0\ \mathbf{y}_1\ ...\ ]$ and $\mathbf{P} = [\ p_0\ p_1\ ...\ ]$. The mean values $\mathbf{x}_\mu$ $\mathbf{y}_\mu$ and standard deviations $\mathbf{x}_\sigma$ $\mathbf{y}_\sigma$ are used to normalize the data which is then additionally scaled by weights $\mathbf{x}_w$ $\mathbf{y}_w$ which give the relative importances of each dimension: in our experiments the best results are achieved by scaling all input variables relating to the joints by a value of 0.1 to shrink their importance. This increases the influence of the trajectory in the regression, resulting in a more responsive character. The binary variables included in the input are normalized as usual and don't recieve any special treatment. After the terrain fitting is complete and the final parameterization is constructed for each of the 10 different associated patches, we have a final dataset that contains around 4 million data points.

To train the network we must ensure that for a given set of control parameters $\mathbf{X}$ and phase parameters $\mathbf{P}$, we can produce the corresponding output variables $\mathbf{Y}$ as a function of the neural network $\boldsymbol{\Phi}$. Training is therefore an optimization problem with respect to the phase function parameters $\boldsymbol{\beta} = \{\boldsymbol{\alpha}_0\ \boldsymbol{\alpha}_1\ \boldsymbol{\alpha}_2\ \boldsymbol{\alpha}_3\}$ and the following cost function:

$$
\mathrm{Cost}(\mathbf{X},\mathbf{Y},\mathbf{P};\boldsymbol{\beta}) = ||\mathbf{Y} - \boldsymbol{\Phi}(\mathbf{X};\boldsymbol{\Theta}(\mathbf{P};\boldsymbol{\beta}))||_2^2 + \gamma\ |\boldsymbol{\beta}|. \tag{6.8}
$$

In this equation the first term represents the mean squared error of the regression result, while the second term represents a small regularization which ensures the weights do not get too large. It also introduces a small amount of sparsity to the weights. This term is controlled by the constant γ, which in this work is set to 0.01.

We use the stochastic gradient descent algorithm Adam [Kingma and Ba, 2014] with a model implemented in Theano [Bergstra et al., 2010] which automatically calculates the derivatives of the cost function with respect to $\boldsymbol{\beta}$. Dropout [Srivastava et al., 2014] is applied with a retention probability of 0.7 and the model is trained in mini-batches of size 32. Full training is performed for 20 epochs which takes around 30 hours on a NVIDIA GeForce GTX 660 GPU.

6.7 Runtime

During runtime the PFNN must be supplied at each frame with the phase p and neural network input $\mathbf{x}$. The phase p can be stored and incremented over time using the computed change in phase $\dot{p}$, modulated to loop in the range $0 \leq p \leq 2\pi$. For the neural network input $\mathbf{x}$, the variables relating to the joint positions and velocities are used in an autoregressive manner, using the computed result from the previous frame as input to the next. Our system also uses past/future trajectories $\mathbf{t}^p$ $\mathbf{t}^d$ as input $\mathbf{x}$ (see Section 6.5.3): the elements related to the *past* are simply recorded, while some care is required for those of the *future*, which we discuss next.

6.7.0.0.1 Inputs relating to Future Trajectories When preparing the runtime input $\mathbf{x}$ for PFNN, the future elements of the trajectory $\mathbf{t}^p$ $\mathbf{t}^d$ are computed by blending the trajectory estimated from the game-pad control stick and those generated by the PFNN in the previous frame.

In Motion Matching Clavet [Clavet, 2016] uses the position of the stick to describe the desired velocity and facing direction of the character. In our method this desired velocity and facing direction is then blended at each future frame with the velocity and facing direction predicted by the PFNN in the previous frame ($\mathbf{t}^p_{i+1}$ $\mathbf{t}^d_{i+1}$). To do this we use the blending function specified below:

$$\text{TrajectoryBlend}(\mathbf{a}_0, \mathbf{a}_1, t, \tau) = (1 - t^{\tau})\ \mathbf{a}_0 + t^{\tau}\ \mathbf{a}_1, \tag{6.9}$$

where t ranges from 0 to 1 as the trajectory gets further into the future, and τ represents an additional bias that controls the responsiveness of the character. In the results shown in this paper we set the bias for blending velocities τ_v to 0.5, which results in blending function which biases toward the PFNN predicted velocities, and the bias for blending facing directions τ_d to 2.0, which results in a bias toward the facing direction of the game-pad stick. This produces a character which looks natural yet remains responsive, as most perceived responsiveness comes from when the character is responding quickly to changes in the desired facing direction.

Also included in the future trajectory are variables related to the semantic information of the motion $\mathbf{t}^g$. This includes the desired gait of the character (represented as a binary vector) as well as other information such as the height of the ceiling and if the character is required to jump instead of climb. These are all set either by user interaction (e.g. we use the right shoulder button of the game-pad to indicate the gait should switch to a jog), or by checking the location of the trajectory against elements of the environment (e.g. when the trajectory passes over certain areas the variable indicating a jumping motion is activated).

Once the variables relating to the future trajectory have been found, the final step is to project the trajectory locations vertically onto the scene geometry and extract the heights to prepare $\mathbf{t}^h$. This constitutes all the required input variables for the PFNN, at which point the output $\mathbf{y}$ can be computed.

Given the output $\mathbf{y}$ the final joint transformations are computed from the predicted joint positions and angles $\mathbf{j}^p, \mathbf{j}^a$. These joint transforms are then edited to avoid foot sliding using a simple two-joint IK along with the contact labels $\mathbf{c}$. The root transform position and rotation is updated using the predicted root translational and rotational velocities $\dot{\mathbf{r}}^x, \dot{\mathbf{r}}^z, \dot{\mathbf{r}}^a$. This completes the runtime process for an individual frame.

6.7.0.0.2 Precomputation of the Phase Function: Once trained, the PFNN is extremely compact, requiring $\sim$ 10 megabytes to store the variables $\boldsymbol{\beta}$. Yet the phase function $\boldsymbol{\Theta}$ may require almost a millisecond of time to compute, which in some cases can be too slow. Because the phase function $\boldsymbol{\Theta}$ is a one-dimensional function over a fixed domain $0 \leq p \leq 2\pi$ it is possible to avoid computation of $\boldsymbol{\Theta}$ at runtime by precomputing $\boldsymbol{\Theta}$ offline for a number of fixed intervals in the range $0 \leq p \leq 2\pi$ and interpolating the results of this precomputation at runtime.

Several options of precomputation are available offering different trade offs between speed and memory consumption (See Table 6.1). The *constant* method is to precompute $\boldsymbol{\Theta}$ for $n = 50$ locations along the phase, and at runtime simply use the neural network weights at the nearest precomputed phase location. This increases the memory consumption by $\frac{n}{4}$ times, but effectively removes the computation of $\boldsymbol{\Theta}$ entirely. Alternately, $n = 10$ samples can be taken and a piecewise *linear* interpolation performed of these samples. This approach requires less memory and may be more accurate but the piecewise linear interpolation also requires more computation time. Alternately the full *cubic* Catmull-Rom spline can be evaluated at runtime.

6.8 Results

In this section we show the results of our method in a number of different situations. For a more detailed demonstration the readers are referred to the supplementary material. All results are shown using the *constant* approximation of the phase function.

In Fig. 6.6 we show our method applied to a character navigating a planar environment performing many tight turns, changes in speed, and facing directions. Our system remains responsive and adapts well to the user input, producing natural, high quality motion for a range of inputs.

In Fig. 6.1 we show the results of our method applied to a character navigating over rough terrain. Our character produces natural motion in a number of challenging situations, stepping, climbing and jumping where required.

In Fig. 6.7 we show our method applied in an environment where the ceiling is low such that the character must crouch to proceed. By adjusting a semantic variable in $\mathbf{t}^g$ relating to the height of the ceiling, either using the environment or the game-pad, the character will crouch at different heights.

Sometimes the game designer wants the character to traverse the environment in a different way. By adjusting a different semantic variable in $\mathbf{t}^g$ we can get the character to jump over obstacles instead of climb over them, as shown in Fig. 6.8.

By colliding the future trajectory with walls or other untraversable objects we can get the character to slow down and avoid obstacles. Alternately the character can be forced into certain environments to create specific movements such as balancing on a beam, as demonstrated in the urban environment shown in Fig. 6.9.

6.9 Evaluation

In this section we compare our method to a number of other techniques including a standard neural network where the phase is not given as an input (see Fig. 6.10(a)), a standard neural network where the phase is given as an additional input variable (see Fig. 6.10(b)), a Encoder-Recurrent-Decoder (ERD) network [Fragkiadaki et al., 2015] (see Fig. 6.10(c),(d)), an autoregressive Gaussian Process trained on a subset of the data (see Fig. 6.10(e)) and a similarly structured Gaussian Process approach which builds separate regressors for 10 different locations along the phase space (see Fig. 6.10(f)). All neural network approaches are trained until convergence and the number of weights in each network adjusted such that the memory usage is equal to our method to provide a fairer comparison.

We show that each of these models fails to produce the same high quality motion seen in our technique and additionally some have fundamental issues which make them difficult to train and use effectively on motion data. We also include a performance comparison detailing training time, runtime cost and memory usage. We then evaluate our data-driven terrain fitting model - comparing the results to a simple procedural model that fits a basic surface to the footstep locations. Finally, we evaluate the responsiveness and following ability of our system by adjusting the blending method of the future trajectory and measuring how these changes affect the results.

6.9.0.0.1 Standard Neural Network When using a neural network that does not explicitly provide the phase as an input (see Fig. 6.10, (a)), the system will blend inputs of different phases which results in poses from different phases being averaged and the character appearing to float [Holden et al., 2016]. As our system only blends data at the same phase it does not suffer from such issues.

In an attempt to avoid such erroneous blending it is possible to explicitly provide the phase as an additional input variable to the neural network (see Fig. 6.10, (b)). During training the influence of the phase is in some cases ignored and the other variables end up dominating the prediction, causing the motion to appear stiff and unnatural (see Fig. 6.11 and supplementary video). This effect can be partially explained by the usage of dropout [Srivastava et al., 2014], which is an essential procedure for regularization, where input nodes are randomly disabled to make the system robust against over-fitting and noise. As the input variable relating to the phase is often disabled during dropout, the system attempts to dilute its influence where possible, instead erroneously learning to predict the pose from the other input variables. To verify this theory we measure the change in output $\mathbf{y}$ with respect to the change in phase $||\frac{\delta \mathbf{y}}{\delta p}||$ in each network. In the standard neural network with the phase given as an additional input this value is relatively small (~ 0.001), while in the PFNN, as the phase is allowed to change all of the network weights simultaneously, it is around fifty times larger (~ 0.05). While it is possible to rectify this somewhat by reducing dropout just for the phase variable, or including multiple copies of the phase variable in the input, these techniques provide a less elegant solution and weaker practical guarantee compared to the full factorization performed by our method.

6.9.0.0.2 Encoder-Recurrent-Decoder Network Next, we compare our system against the Encoder-Recurrent-Decoder (ERD) Network [Fragkiadaki et al., 2015] - one of the current state-of-the-art neural network techniques which is a variant of the RNN/LSTM network structure (see Fig. 6.10 (c)). The ERD network is produced by taking Φ and making the middle transformation into an LSTM recurrent transformation. Although this structure is not identical to previous versions of the ERD network, the nature of the construction is the same.

As the ERD network has a memory, even when the phase is not given directly, it can learn some concept of phase from the observable data and use this to avoid the issue of ambiguity in the input. The ERD network conducts the recurrent process on the manifold of the motion (middle layer), thus significantly delaying the time that the "dying out" process starts [Fragkiadaki et al., 2015].

Unfortunately some pathological cases still exist - for example if the character is standing still and the user indicates for the character to walk, as it is impossible to observe the phase when the character is stationary, the ERD network cannot know if the user intends for the character to lead with their left foot or their right, resulting in an averaging of the two, and a floating effect appearing. In this sense the phase cannot be learned in all cases, and is a *hidden* latent variable which must be supplied independently as in our method.

Providing the phase as an additional input to this network (see Fig. 6.10 (d)) can improve the generation performance significantly but still does not entirely remove the floating artifacts (see supplementary video).

6.9.0.0.3 Autoregressive Gaussian Processes Gaussian Processes (GP) have been applied to autoregressive problems with some success in previous work [Wang et al., 2008]: here we compare our approach with respect to two architectures based on GP.

Firstly, we use a GP to perform the regression from $\mathbf{x}$ to $\mathbf{y}$ with phase given as an additional input (see Fig. 6.10 (e)): this can be considered similar to a Gaussian Process Dynamic Model (GPDM) [Wang et al., 2008] if you consider the input control parameters to be the latent variables, which in this case are hand-tuned instead of automatically learned. Since it is difficult to avoid the GP over-fitting on the small amount of data it is provided, the system becomes unstable and jittery (see supplementary video). Additionally, the GP cannot adapt to many complex situations since the cost of construction grows in the square order for memory and cubic order for computational cost. Thus, we could only test this system for motions on a planar surface, limiting the training samples to 3,000. Still, it has bad runtime performance and memory costs on large data sets.

Next, we build several independent GPs for 10 different locations along the phase space, selecting the nearest two GPs to perform the regression at runtime and interpolating the result (see Fig. 6.10(f)). Practically this achieves an effect similar to the PFNN but we find the quality of this regression as well as the memory and runtime performance to be much worse as it cannot be trained on nearly as much data (see Fig. 6.12).

6.9.0.0.4 Performance In Table 6.1 we compare the performances of the various methods shown above including memory usage and runtime. GP based techniques are limited by their data capacity, memory usage, and runtime performance. When using the *constant* approximation our method has comparable runtime performance to other neural network based techniques but with a higher memory usage. When using the full *cubic* Catmull-Rom spline interpolation of the phase function it has similar memory usage but with a longer runtime. One downside of our method is that it requires longer training times than other methods. All runtime measurements were made using an Intel i7-6700 3.4GHz CPU running single threaded.

6.9.0.0.5 Terrain Fitting In Fig. 6.13 we show a comparison to a different terrain fitting process. In this process we start with a flat plane and use the mesh editing technique described in Section 6.5.2 to deform the surface to touch the footstep locations. Terrain synthesized in such a way are smooth and without much variation. As a result of over-fitting to such smooth terrains the system produces odd motions during runtime when the character encounters environments such as those with large rocks. The readers are referred to the supplementary material for further details.

6.9.0.0.6 Responsiveness In Fig. 6.14 we show an evaluation of the responsiveness and following ability of our method. We create several predefined paths and instruct the character to follow them. We then measure the difference between the desired trajectory and the actual trajectory of the character (see Table 6.2). By increasing the variable τ (described in Section 6.7) responsiveness can be improved at the cost of a small loss in animation quality. The readers are referred to the supplementary material for further details.

6.10 Discussions

In this section, we first discuss about the network architecture that we adopt, and also about the input/output parameters of the system. Finally we discuss about the limitations of our system.

6.10.0.0.1 Network Architecture As shown in our experiments, our system can compute realistic motions in a time-series manner, without suffering from issues such as dying out or instability which often occur in autoregressive models. This is made possible through several system designs: By using the phase as a global parameter for the weights of the neural network we can avoid mixing motions at different phases which notoriously results in the “dying out” effect. This also ensures the influence of the phase is strongly taken into account during training and runtime, and thus there is little chance that its influence is diluted and ignored, something we observed when giving the phase as an additional input parameter to other network structures.

Our network is composed of one input layer, one output layer, and two hidden layers with 512 hidden units. This design is motivated by the wish to make the network as simple as possible to ensure the desired motion can be computed easily for our real-time animation purpose. We find the current network structure is a good compromise between computational efficiency and the richness of the movements produced.

While we only demonstrate the PFNN applied to cyclic motions, it can just as easily by applied to non-cyclic motions by adopting a non-cyclic phase function. In non-cyclic data the phase can be specified as 0 at the start of the motion 0.5 half way through and 1.0 at the end. If trained with a non-cyclic phase function and appropriate control parameters the PFNN could easily be used on other data or tasks such as punching and kicking.

Conceptually, our system is similar to training separate networks for each phase. This is something we tried in early iterations of our research, but when doing so each network learned a slightly different regression due to the random weight initialization and other factors. There was therefore no continuity or cyclic nature preserved and the motion looked jittery, with a noticeable "seam" when the phase looped around. The PFNN provides an elegant way of achieving a similar effect but without the stated problems.

6.10.0.0.2 Control Parameters for Real-time Character Control Our system is designed specifically for real-time character control in arbitrary environments that can be learned from motion capture data: for this purpose we use an input parameterization similar to one that has proven to be effective in Motion Matching [Clavet, 2016]. The future trajectory is predicted from the user inputs while the input related to the environment is found by sampling the heights of the terrain under the trajectory of the character. Using a window of past and future positions, directions, and geometry reduces the potential for ambiguity and produces higher quality motion. For additional tasks it should be possible to adapt this input parameterisation, for example to include extra input variables related to the style of the motion.

6.10.0.0.3 Other Potential Parameterization Deep Learning has found that it is possible to use input parameterizations which are not hand crafted in this way but use more neural network layers to perform the abstraction. For example - it is possible for our purposes to use a depth or RGB image of the surrounding terrain as the input relating to the environment and use convolutional layers for abstracting this input. This is an interesting approach and may be a good solution especially for robotics, where the actions are limited and the details of the ground are needed for keeping balance. In the early stages of our research we tested an approach similar to this, but noticed a few issues. Firstly, using a CNN required a lot more training data as we found that the character does not interact with any terrain along the sides of the images, meaning we needed a large variety of different images to prevent over-fitting in these places. We also found that when using convolutional layers or more complex neural network structures to abstract the input the processing time increased to a point where it became unsuitable for real-time applications such as computer games.

6.10.0.0.4 Limitations & Future Work In order to achieve the real-time performance we only coarsely sample points along the trajectory. This results in the system missing some high resolution details such as sharp small obstacles along the terrain, which need to be avoided in actual applications. One simple solution is to have an additional layer of control on top of the current system to respond to labeled obstacles, while using our system for low frequency geometry of the environment.

Like many methods in this field our technique cannot deal well with complex interactions with the environment - in particular if they include precise hand movements such as climbing up walls or interacting with other objects in the scene. In the future labeling hand contacts and performing IK on the hands may help this issue partially. Alternately, performing the regression in a space more naturally suited to interactions such as that defined by relationship descriptors [Al-Asqhar et al., 2013] may be interesting and produce compelling results.

The PFNN is relatively slow to train as each element in the mini-batch produces different network weights meaning the computation during training is much more expensive than usual. The PFNN can produce acceptable results for testing after just a couple of hours of training, but performing the full 30 hour training each time new data is added is not desirable. For this reason we are interested in ways to speed up the training of the PFNN or ways of performing incremental training.

If the user supplies an input trajectory which is unachievable or invalid in the given context (e.g. the terrain is too steep) our system will extrapolate which may produce undesirable results (see Fig. 6.15). Additionally the results of our method may be difficult to predict, and therefore hard for artists to fix or edit. A dedicated method for editing and controlling the results of techniques such as ours is therefore desirable.

Finally, it is also interesting to apply our technique for other modalities, such as videos of periodic data e.g. fMRI images of heartbeats. Using a periodic model such as the PFNN can potentially make the learning process more efficient for such kinds of data.

6.11 Conclusion

We propose a novel learning framework called a Phase-Functioned Neural Network (PFNN) that is suitable for generating cyclic behavior such as human locomotion. We also design the input and output parameters of the network for real-time data-driven character control in complex environments with detailed user interaction. Despite its compact structure, the network can learn from a large, high dimensional dataset thanks to the factorization of the key phase parameter, and a phase function that varies smoothly over time to produce a large variation of network configurations. We also propose a framework to produce additional data for training the PFNN where the human locomotion and the environmental geometry are coupled. Once trained our system is fast, requires little memory, and produces high quality motion without exhibiting any of the common artifacts found in existing methods.

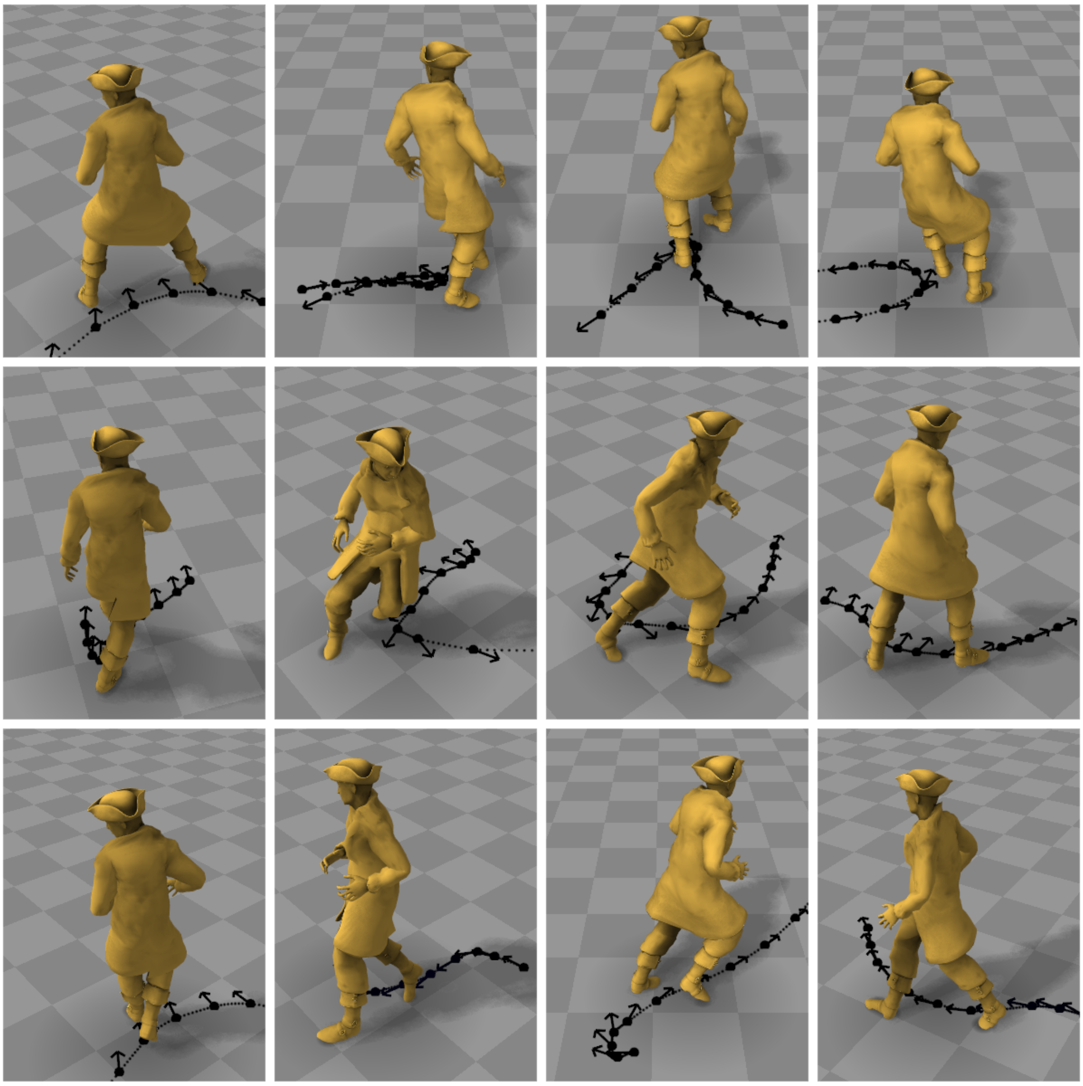

Figure 6.6: Results showing the character traversing a planar environment. By adjusting the future trajectory position and direction the user can make the character turn and sidestep.

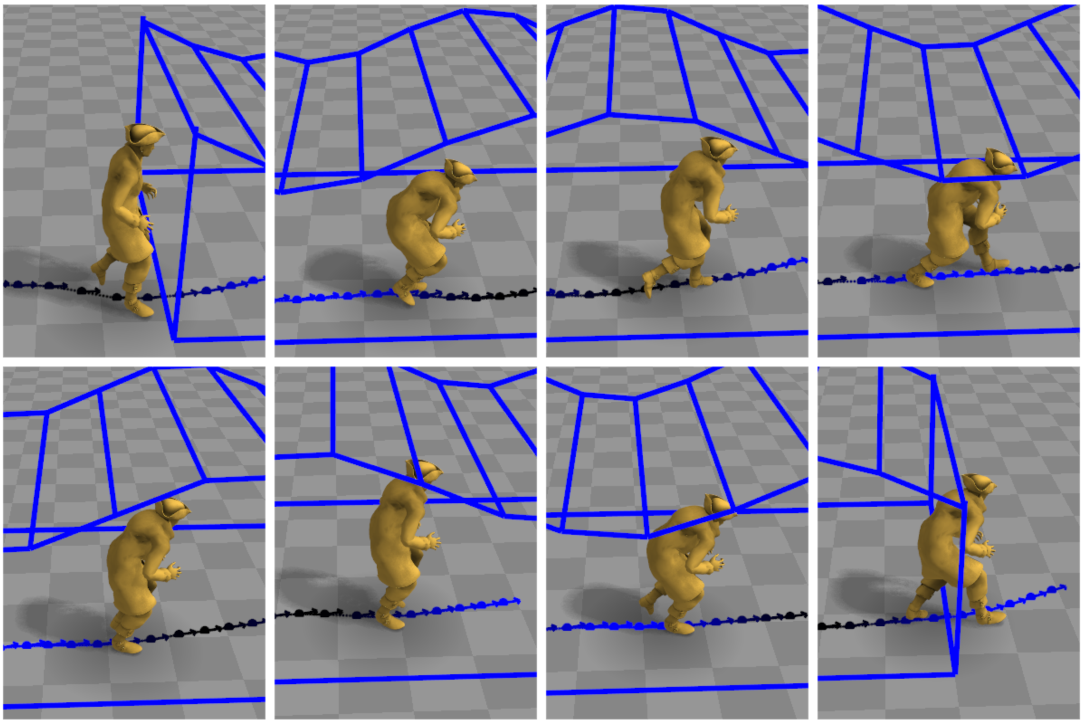

Figure 6.7: Results showing the character crouching as the variable indicating the ceiling height is adjusted by the environment.

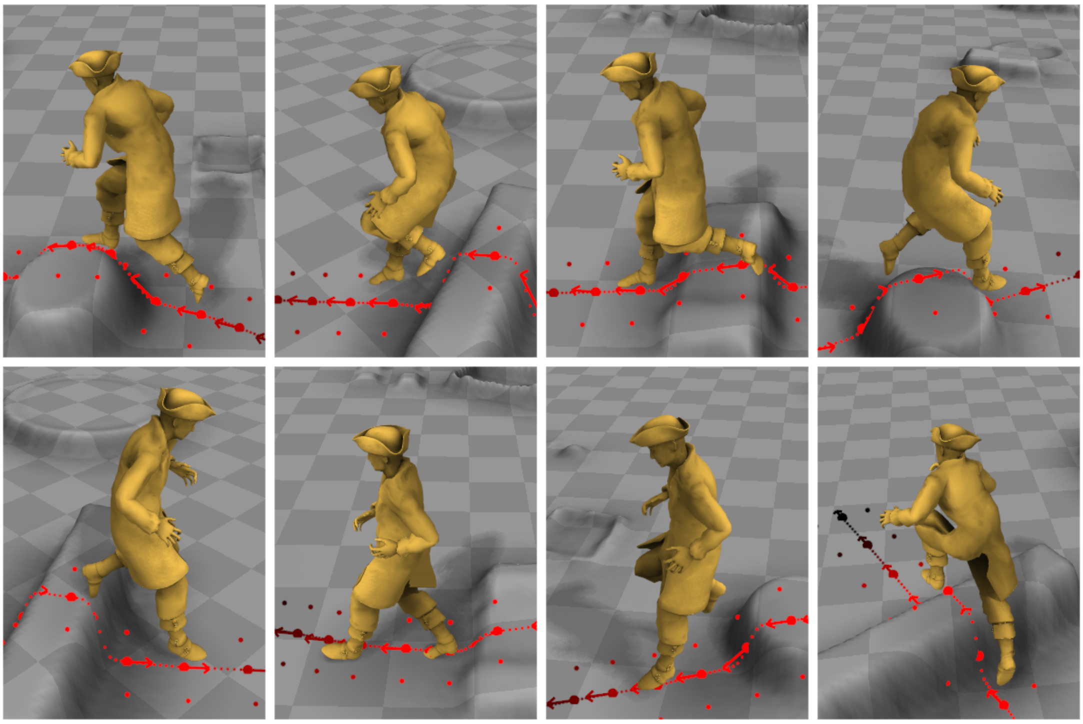

Figure 6.8: Results showing the character performing jumping motions over obstacles that have been labeled to indicate jumping should be performed.

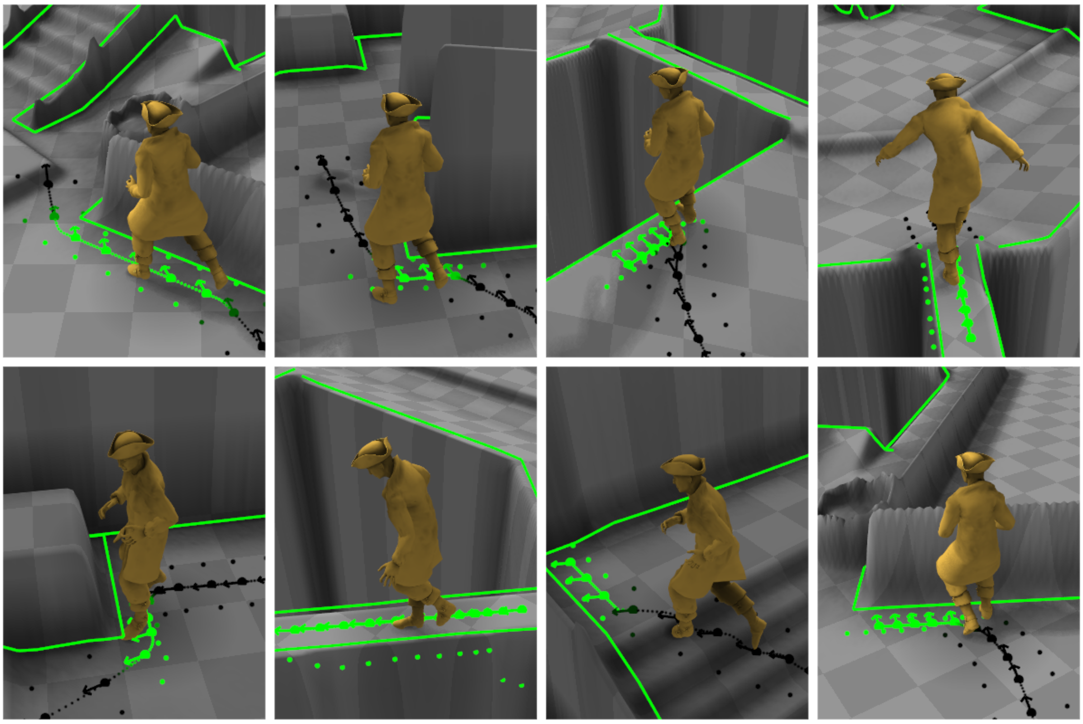

Figure 6.9: Result of the character where the future trajectory has been collided with walls, pits and other objects in the environment. By colliding the future trajectory with non-traversable objects the character will slow down or avoid such obstacles. When walking along the beam, since the measured heights on either side of the character are significantly lower than in the center, a balancing motion is naturally produced.

Technique	**Training**	**Runtime**	**Memory**
PFNN *cubic*	30 hours	0.0018s	10 MB
PFNN *linear*	30 hours	0.0014s	25 MB
PFNN *constant*	30 hours	0.0008s	125 MB
NN (a) (b)	3 hours	0.0008s	10 MB
ERD (c) (d)	9 hours	0.0009s	10 MB
GP (e)	10 minutes	0.0219s	100 MB
PFGP (f)	1 hour	0.0427s	1000 MB

Table 6.1: Numerical comparison between our method and other methods described in Fig. 6.10.

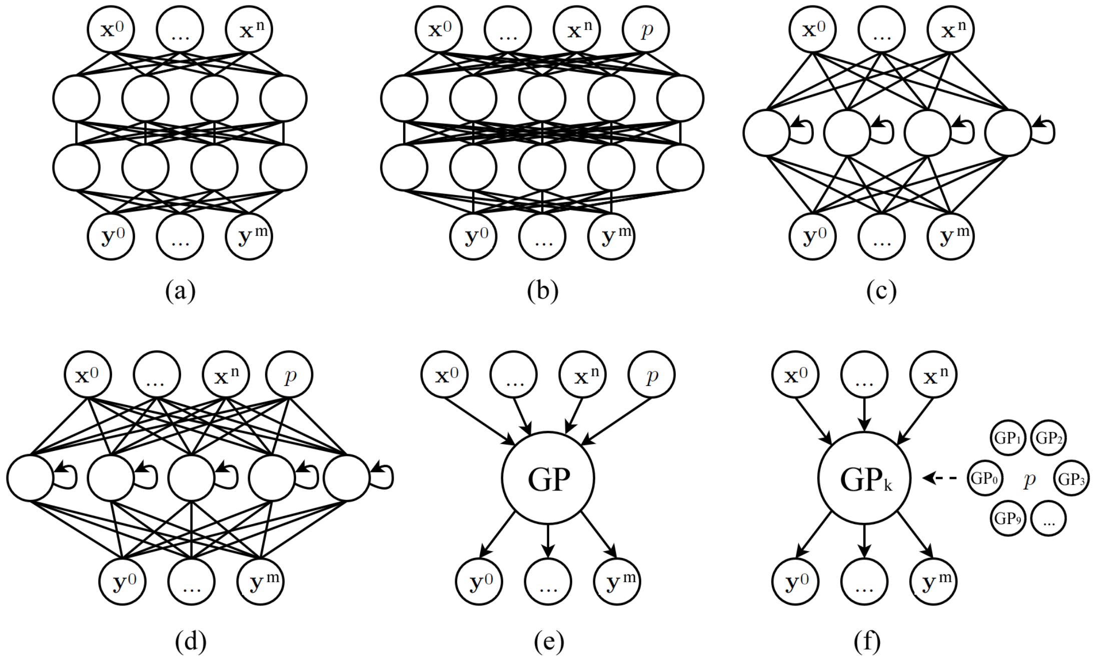

Figure 6.10: The configurations of neural network structures and Gaussian process structures evaluated in our comparison: (a) NN without the phase given as input, (b) NN with the phase given as an additional input variable, (c) an ERD network, (d) an ERD network with the phase given as an additional input variable, (e) a GP autoregressor, and (f) a GP autoregressor that is selected using the phase.

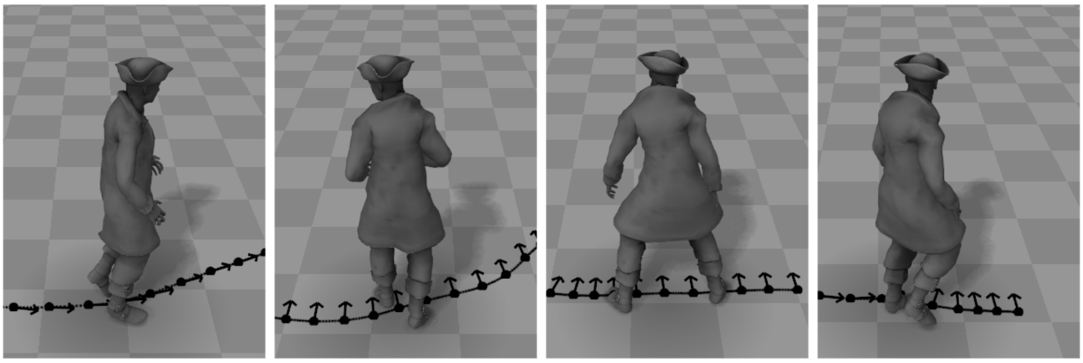

Figure 6.11: Results of using a neural network where the phase is given as an additional input. The character motion appears stiff and unnatural as the input relating to the phase is often ignored by the network and other variables used to infer the pose instead.

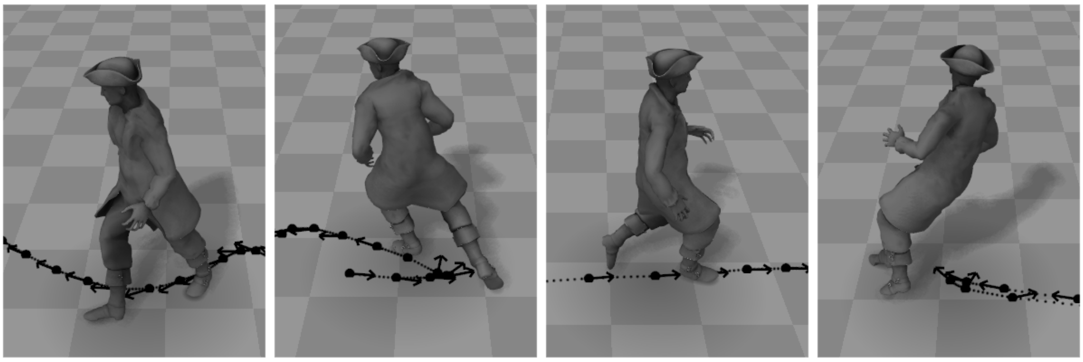

Figure 6.12: The Phase Functioned Gaussian Process (PFGP) achieves an effect similar to our method, but the quality of the produced motion and the computational costs are much worse since it cannot learn from as much data.

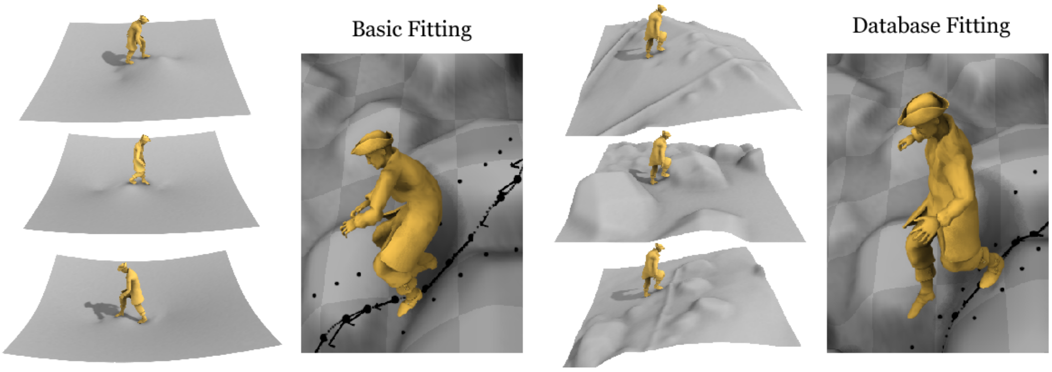

Figure 6.13: Comparison of terrain fitting approaches. Simple mesh editing (left) creates a basic surface which touches the footstep locations. In this case the system over-fits to the style of terrains produced by this technique and produces odd motions when terrains unobserved during training are encountered at runtime. Our method (right) uses similar terrains to those encountered at runtime therefore producing more natural motion.

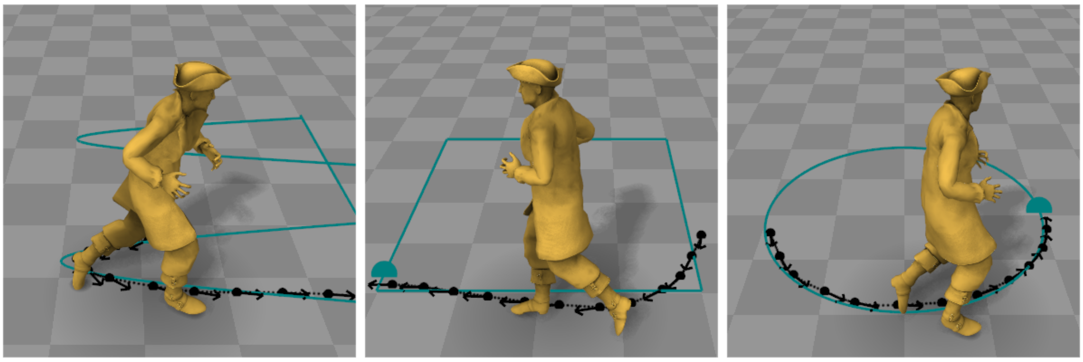

Figure 6.14: Evaluation of the responsiveness of our method. The character is directed to follow several predefined paths and the difference between the desired path and actual path made by the character is measured.

Path	$\tau_\mathbf{v}$	$\tau_\mathbf{d}$	**Avg. Error**
Wave	0.5	2.0	17.29 cm
	2.0	5.0	10.66 cm
	5.0	10.0	8.88 cm
Square	0.5	2.0	21.26 cm
	2.0	5.0	11.25 cm
	5.0	10.0	8.17 cm
Circle	0.5	2.0	13.70 cm
	2.0	5.0	8.03 cm
	5.0	10.0	5.82 cm

Table 6.2: Numerical evaluation of character responsiveness and following ability. For each scene in Fig. 6.14 we measure the average error between the desired path and that taken by the character with different biases supplied to the future trajectory blending function Eq. (6.9). Here τ_v represents the blending bias for the future trajectory velocity, and τ_d represents the blending bias for the future trajectory facing direction (see Section 6.7 for a more detailed explanation).

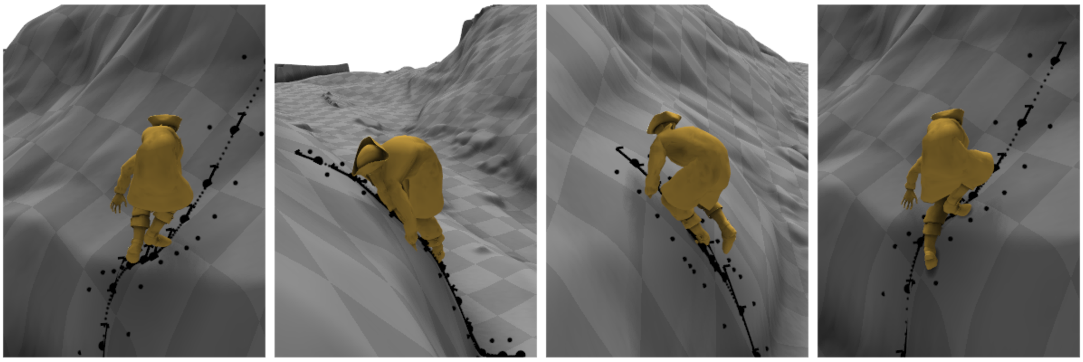

Figure 6.15: If the input trajectory is unachievable such as when the terrain is too steep our approach will extrapolate which can often produce bad looking motion.

6.12 Postscript

In this chapter we presented a new neural network structure influenced by work in previous chapters which can effectively solve the issue of character control - extending our previous work to the domain of games.

Since our method is fully automatic it not only significantly reduces the time required by technical developers, but also reduces the manual editing time performed by animators in game studios. This work also shows how the developments made in previous chapters can be widely adopted to new domains and can have a significant contribution to future research.

Chapter 7

Conclusion

In the development of this thesis there were two major goals. The first goal was to identify the reasons why existing character animation tools had not been adopted in keyframed animation environments and to develop new methods to overcome these issues. For this goal we developed a number of data-driven methods which can invert the *rig function* at interactive rates - allowing existing and future animation research tools to be seamlessly integrated into the keyframed animation pipeline. Our technique provided an effective initial solution but we also see it as a first step toward many more incremental improvements and the *bridging* of a long-standing gap in the animation production pipeline. This work has lead to publications in an esteemed conference and a journal.

The second goal of this work was to use modern machine learning tools that had seen great success in other fields in the production of new tools for character animation. In this direction we first developed the *motion manifold* - a manifold upon which human motion data exists. This itself is a powerful tool, with many applications in animation research and technology, but additionally this presented a base upon which further work could be developed and new machine learning techniques explored. This work was published in SIGGRAPH Asia Technical Briefs.

Using the motion manifold we then developed a framework for character motion synthesis and editing. This framework can be used for a large number of common character animation tasks, does not require any manual preprocessing of animation data, and fits nicely into the keyframed animation pipeline. This represents a significant contribution to the use deep learning techniques in character animation and was published in

SIGGRAPH 2016.

Finally we showed how these developments for keyframed animation could be used to advance the state of the art in game development. Influenced by these previous works we developed a new neural network structure called the phase-functioned neural network which has several features particularly suitable for character control - a large capacity for data, an ability to handle complex control situations, a low memory footprint, and a fast runtime.

Together we believe these techniques have laid a strong groundwork for the use of deep learning in the field of character animation. By applying new techniques in the automatic creation of high quality production level character animation we have contributed to reducing the number of manually constructed animator keyframes required both in the development of animated films and in games.

On a final note, while deep learning has undeniably seen great successes in many fields of computer science it has in some ways solved the *easy* problems when it comes to statistical modelling - those problems related to data capacity, computational complexity, memory usage and computation speed. Many of the problems that are in some sense *difficult* remain open and as important as ever - *how can we edit the results of these systems* - *how do we fix things when they go wrong* - *what can we do when there is not enough data* - *how can we produce stylistic or non-human motion*. In short - *where does the human fit into the loop*.

Bibliography

[Ani,] Animation mentor. http://www.animationmentor.com/.

[Fac,] Faceware technologies inc. http://facewaretech.com/.

[Mar,] Marza animation planet inc. http://www.marza.com/en/.

[Abdel-Hamid et al., 2014] Abdel-Hamid, O., Mohamed, A.-R., Jiang, H., Deng, L., Penn, G., and Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 22(10):1533–1545.

[Al-Asqhar et al., 2013] Al-Asqhar, R. A., Komura, T., and Choi, M. G. (2013). Relationship descriptors for interactive motion adaptation. In *Proc. SCA*, pages 45–53.

[Allen and Faloutsos, 2009] Allen, B. F. and Faloutsos, P. (2009). Evolved controllers for simulated locomotion. In *Motion in games*, pages 219–230. Springer.

[Arikan and Forsyth, 2002] Arikan, O. and Forsyth, D. A. (2002). Interactive motion generation from examples. *ACM Trans. Graph.*, 21(3):483–490.

[Bergstra et al., 2010] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

[Bickel et al., 2008] Bickel, B., Lang, M., Botsch, M., Otaduy, M. A., and Gross, M. (2008). Pose-space animation and transfer of facial details. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 57–66.

[Botsch and Kobbelt, 2005] Botsch, M. and Kobbelt, L. (2005). Real-Time Shape Editing using Radial Basis Functions. *Computer Graphics Forum*.

[Bowden, 2000] Bowden, R. (2000). Learning statistical models of human motion.

[Bruderlin and Williams, 1995] Bruderlin, A. and Williams, L. (1995). Motion signal processing. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 97–104, New York, NY, USA. ACM.

[Buss, 2004] Buss, S. R. (2004). Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. Technical report, IEEE Journal of Robotics and Automation.

[Buss and Kim, 2004] Buss, S. R. and Kim, J.-S. (2004). Selectively damped least squares for inverse kinematics. *Journal of Graphics Tools*, 10:37–49.

[Chai and Hodgins, 2005] Chai, J. and Hodgins, J. K. (2005). Performance animation from low-dimensional control signals. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 686–696, New York, NY, USA. ACM.

[Chai and Hodgins, 2007] Chai, J. and Hodgins, J. K. (2007). Constraint-based motion optimization using a statistical dynamic model. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA. ACM.

[Chan and Lawrence, 1988] Chan, S. and Lawrence, P. (1988). General inverse kinematics with the error damped pseudoinverse. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pages 834–839 vol.2.

[Chiaverini, 1993] Chiaverini, S. (1993). Estimate of the two smallest singular values of the jacobian matrix: Application to damped least-squares inverse kinematics. *Journal of Robotic Systems*, 10(8):991–1008.

[Choi and Ko, 1999] Choi, K.-J. and Ko, H.-S. (1999). On-line motion retargetting. *Journal of Visualization and Computer Animation*, 11:223–235.

[Ciresan et al., 2012] Ciresan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649.

[Clavet, 2016] Clavet, S. (2016). Motion matching and the road to next-gen animation. In *Proc. of GDC 2016*.

[Clevert et al., 2015] Clevert, D., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289.

[Cmu,] Cmu. Carnegie-Mellon Mocap Database. `http://mocap.cs.cmu.edu/`.

[Cohen, 1992] Cohen, M. F. (1992). Interactive spacetime control for animation. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 293–302, New York, NY, USA. ACM.

[Coros et al., 2008] Coros, S., Beaudoin, P., Yin, K. K., and van de Pann, M. (2008). Synthesis of constrained walking skills. *ACM Trans on Graph*, 27(5):113.

[DeepLearning,] DeepLearning. Deep learning convolutional neural networks. `http://deeplearning.net/tutorial/lenet.html`.

[Du et al., 2015] Du, Y., Wang, W., and Wang, L. (2015). Hierarchical recurrent neural network for skeleton based action recognition. In *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*.

[Fan et al., 2010] Fan, J., Xu, W., Wu, Y., and Gong, Y. (2010). Human tracking using convolutional neural networks. *Neural Networks, IEEE Transactions on*, 21(10):1610–1623.

[Fragkiadaki et al., 2015] Fragkiadaki, K., Levine, S., Felsen, P., and Malik, J. (2015). Recurrent network models for human dynamics. In *Proc. ICCV*, pages 4346–4354.

[Gatys et al., 2015] Gatys, L. A., Ecker, A. S., and Bethge, M. (2015). A neural algorithm of artistic style. *CoRR*, abs/1508.06576.

[Gleicher, 1998] Gleicher, M. (1998). Retargetting motion to new characters. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 33–42, New York, NY, USA. ACM.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

[Grabner et al., 2011] Grabner, H., Gall, J., and Van Gool, L. (2011). What makes a chair a chair? In *Proc. IEEE CVPR*, pages 1529–1536.

[Grassia, 1998] Grassia, F. S. (1998). Practical parameterization of rotations using the exponential map. *J. Graph. Tools*, 3(3):29–48.

[Graves et al., 2013] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE.

[Grochow et al., 2004] Grochow, K., Martin, S. L., Hertzmann, A., and Popović, Z. (2004). Style-based inverse kinematics. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 522–531, New York, NY, USA. ACM.

[Gupta et al., 2011] Gupta, A., Satkin, S., Efros, A. A., and Hebert, M. (2011). From 3d scene geometry to human workspace. In *Proc. IEEE CVPR*, pages 1961–1968.

[Hahn et al., 2012] Hahn, F., Martin, S., Thomaszewski, B., Sumner, R., Coros, S., and Gross, M. (2012). Rig-space physics. *ACM Trans. Graph.*, 31(4):72:1–72:8.

[Hahn et al., 2013] Hahn, F., Thomaszewski, B., Coros, S., Sumner, R. W., and Gross, M. (2013). Efficient simulation of secondary motion in rig-space. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

[Hariharan et al., 2014] Hariharan, B., Arbeláez, P. A., Girshick, R. B., and Malik, J. (2014). Hypercolumns for object segmentation and fine-grained localization. *CoRR*, abs/1411.5752.

[Heck and Gleicher, 2007] Heck, R. and Gleicher, M. (2007). Parametric motion graphs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 129–136, New York, NY, USA. ACM.

[Hinton, 2012] Hinton, G. (2012). A practical guide to training restricted boltzmann machines. In Montavon, G., Orr, G., and Mller, K.-R., editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer Berlin Heidelberg.

[Holden et al., 2015a] Holden, D., Saito, J., and Komura, T. (2015a). Learning an inverse rig mapping for character animation. In *Proceedings of the 14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 165–173. ACM.

[Holden et al., 2016] Holden, D., Saito, J., and Komura, T. (2016). A deep learning framework for character motion synthesis and editing. *ACM Trans on Graph*, 35(4).

[Holden et al., 2015b] Holden, D., Saito, J., Komura, T., and Joyce, T. (2015b). Learning motion manifolds with convolutional autoencoders. In *SIGGRAPH Asia 2015 Technical Briefs*, SA '15, pages 18:1–18:4, New York, NY, USA. ACM.

[Howe et al., 1999] Howe, N. R., Leventon, M. E., and Freeman, W. T. (1999). Bayesian reconstruction of 3d human motion from single-camera video. In *Proc. NIPS*.

[Hyun et al., 2013] Hyun, K., Kim, M., Hwang, Y., and Lee, J. (2013). Tiling motion patches. *IEEE Transactions on Visualization and Computer Graphics*, 19(11):1923–1934.

[Hyun et al., 2016] Hyun, K., Lee, K., and Lee, J. (2016). Motion grammars for character animation. *Comput. Graph. Forum*, 35(2):103–113.

[Ji et al., 2013] Ji, S., Xu, W., Yang, M., and Yu, K. (2013). 3d convolutional neural networks for human action recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(1):221–231.

[Kang and Lee, 2014] Kang, C. and Lee, S.-H. (2014). Environment-adaptive contact poses for virtual characters. In *Computer Graphics Forum*, volume 33, pages 1–10. Wiley Online Library.

[Kapadia et al., 2016] Kapadia, M., Xianghao, X., Nitti, M., Kallmann, M., Coros, S., Sumner, R. W., and Gross, M. (2016). Precision: precomputing environment semantics for contact-rich character animation. In *Proc. I3D*, pages 29–37.

[Karpathy et al., 2014] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1725–1732.

[Kim et al., 2009] Kim, M., Hyun, K., Kim, J., and Lee, J. (2009). Synchronized multi-character motion editing. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 79:1–79:9, New York, NY, USA. ACM.

[Kim et al., 2014] Kim, V. G., Chaudhuri, S., Guibas, L., and Funkhouser, T. (2014). Shape2pose: Human-centric shape analysis. *ACM Trans on Graph*, 33(4):120.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[Kovar and Gleicher, 2004] Kovar, L. and Gleicher, M. (2004). Automated extraction and parameterization of motions in large data sets. *ACM Trans. Graph.*, 23(3):559–568.

[Kovar et al., 2002] Kovar, L., Gleicher, M., and Pighin, F. (2002). Motion graphs. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 473–482, New York, NY, USA. ACM.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Kulkarni et al., 2015] Kulkarni, T. D., Whitney, W. F., Kohli, P., and Tenenbaum, J. (2015). Deep convolutional inverse graphics network. In *Proc. NIPS*, pages 2539–2547.

[Lau and Kuffner, 2005] Lau, M. and Kuffner, J. J. (2005). Behavior planning for character animation. In *Proc. SCA*, pages 271–280.

[Lawrence, 2003] Lawrence, N. D. (2003). Gaussian process latent variable models for visualisation of high dimensional data. In *Advances in Neural Information Processing Systems*, page None.

[Lee et al., 2002] Lee, J., Chai, J., Reitsma, P. S. A., Hodgins, J. K., and Pollard, N. S. (2002). Interactive control of avatars animated with human motion data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 491–500, New York, NY, USA. ACM.

[Lee and Lee, 2004] Lee, J. and Lee, K. H. (2004). Precomputing avatar behavior from human motion data. *Proceedings of 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 79–87.

[Lee and Shin, 1999] Lee, J. and Shin, S. Y. (1999). A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 39–48, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Lee and Shin, 2001] Lee, J. and Shin, S. Y. (2001). A coordinate-invariant approach to multiresolution motion analysis. *Graph. Models*, 63(2):87–105.

[Lee et al., 2006] Lee, K. H., Choi, M. G., and Lee, J. (2006). Motion patches: Building blocks for virtual environments annotated with motion data. *ACM Trans. Graph.*, 25(3):898–906.

[Lee et al., 2010] Lee, Y., Wampler, K., Bernstein, G., Popović, J., and Popović, Z. (2010). Motion fields for interactive character locomotion. In *ACM Trans. Graph.*, volume 29, page 138. ACM.

[Levine and Koltun, 2014] Levine, S. and Koltun, V. (2014). Learning complex neural network policies with trajectory optimization. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 829–837.

[Levine et al., 2012] Levine, S., Wang, J. M., Haraux, A., Popović, Z., and Koltun, V. (2012). Continuous character control with low-dimensional embeddings. *ACM Trans. Graph.*, 31(4):28:1–28:10.

[Lewis and Anjyo, 2010] Lewis, J. and Anjyo, K.-i. (2010). Direct manipulation blendshapes. *IEEE Computer Graphics and Applications*, 30(4):42–50.

[Lewis et al., 2000] Lewis, J. P., Cordner, M., and Fong, N. (2000). Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of SIGGRAPH*, pages 165–172.

[Liu et al., 2010] Liu, L., Yin, K., van de Panne, M., Shao, T., and Xu, W. (2010). Sampling-based contact-rich motion control. *ACM Trans on Graph*, 29(4):128.

[Liu et al., 1994] Liu, Z., Gortler, S. J., and Cohen, M. F. (1994). Hierarchical spacetime control. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 35–42, New York, NY, USA. ACM.

[Lo and Zwicker, 2008] Lo, W.-Y. and Zwicker, M. (2008). Real-time planning for parameterized human motion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 29–38, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Mai-Duy and Tran-Cong, 2003] Mai-Duy, N. and Tran-Cong, T. (2003). Approximation of function and its derivatives using radial basis function networks. *Applied Mathematical Modelling*, 27(3):197–220.

[Memisevic, 2013] Memisevic, R. (2013). Learning to relate images. *IEEE PAMI*, 35(8):1829–1846.

[Min and Chai, 2012] Min, J. and Chai, J. (2012). Motion graphs++: A compact generative model for semantic motion analysis and synthesis. *ACM Trans. Graph.*, 31(6):153:1–153:12.

[Min et al., 2009] Min, J., Chen, Y.-L., and Chai, J. (2009). Interactive generation of human animation with deformable motion models. *ACM Transactions on Graphics (TOG)*, 29(1):9.

[Mittelman et al., 2014] Mittelman, R., Kuipers, B., Savarese, S., and Lee, H. (2014). Structured recurrent temporal restricted boltzmann machines. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1647–1655.

[Mizuguchi et al., 2001] Mizuguchi, M., Buchanan, J., and Calvert, T. (2001). Data driven motion transitions for interactive games. In *Eurographics 2001 - Short Presentations*. Eurographics Association.

[Mordatch et al., 2015] Mordatch, I., Lowrey, K., Andrew, G., Popovic, Z., and Todorov, E. (2015). Interactive control of diverse complex characters with neural networks. In *Proceedings of Neural Information Processing Systems*.

[Mukai, 2011] Mukai, T. (2011). Motion rings for interactive gait synthesis. In *Proc. I3D*, pages 125–132.

[Mukai and Kuriyama, 2005] Mukai, T. and Kuriyama, S. (2005). Geostatistical motion interpolation. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1062–1070, New York, NY, USA. ACM.

[Müller et al., 2007] Müller, M., Röder, T., Clausen, M., Eberhardt, B., Krüger, B., and Weber, A. (2007). Documentation mocap database hdm05. Technical Report CG-2007-2, Universität Bonn.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814.

[Nakamura and Hanafusa, 1986] Nakamura, Y. and Hanafusa, H. (1986). Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of dynamic systems, measurement, and control*, 108(3):163–171.

[Nakamura et al., 1987] Nakamura, Y., Hanafusa, H., and Yoshikawa, T. (1987). Task-priority based redundancy control of robot manipulators. *The International Journal of Robotics Research*, 6(2):3–15.

[Nasse et al., 2009] Nasse, F., Thurau, C., and Fink, G. (2009). Face detection using gpu-based convolutional neural networks. In *Computer Analysis of Images and Patterns*, volume 5702 of *Lecture Notes in Computer Science*, pages 83–90.

[Ofli et al., 2013] Ofli, F., Chaudhry, R., Kurillo, G., Vidal, R., and Bajcsy, R. (2013). Berkeley mhad: A comprehensive multimodal human action database. In *Applications of Computer Vision (WACV), 2013 IEEE Workshop on*, pages 53–60.

[Park et al., 2002] Park, S. I., Shin, H. J., and Shin, S. Y. (2002). On-line locomotion generation based on motion blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, pages 105–111, New York, NY, USA. ACM.

[Peng et al., 2016] Peng, X. B., Berseth, G., and van de Panne, M. (2016). Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Trans on Graph*, 35(4).

[Perlin, 1995] Perlin, K. (1995). Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15.

[Pighin et al., 1998] Pighin, F. H., Hecker, J., Lischinski, D., Szeliski, R., and Salesin, D. (1998). Synthesizing realistic facial expressions from photographs. In *Proceedings of SIGGRAPH*, pages 75–84.

[Planet, 2012] Planet, C. (2012). Building character. `http://www.creativeplanetnetwork.com/news/news-articles/building-character/373633`.

[Rasmussen and Ghahramani, 2002] Rasmussen, C. E. and Ghahramani, Z. (2002). Infinite mixtures of gaussian process experts. In *Proc. NIPS*, number 2, pages 881–888.

[Rasmussen and Williams, 2005] Rasmussen, C. E. and Williams, C. K. (2005). Gaussian processes for machine learning (adaptive computation and machine learning).

[Rose et al., 1998] Rose, C., Cohen, M. F., and Bodenheimer, B. (1998). Verbs and adverbs: Multidimensional motion interpolation. *IEEE Comput. Graph. Appl.*, 18(5):32–40.

[Rose III et al., 2001] Rose III, C. F., Sloan, P.-P. J., and Cohen, M. F. (2001). Artist-directed inverse-kinematics using radial basis function interpolation. In *Computer Graphics Forum*, volume 20, pages 239–250.

[Safonova and Hodgins, 2007] Safonova, A. and Hodgins, J. K. (2007). Construction and optimal search of interpolated motion graphs. In *ACM Transactions on Graphics (TOG)*, volume 26, page 106. ACM.

[Safonova et al., 2004] Safonova, A., Hodgins, J. K., and Pollard, N. S. (2004). Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 514–521, New York, NY, USA. ACM.

[Savva et al., 2016] Savva, M., Chang, A. X., Hanrahan, P., Fisher, M., and Nießner, M. (2016). PiGraphs: Learning Interaction Snapshots from Observations. *ACM Trans on Graph*, 35(4).

[Seol and Lewis, 2014] Seol, Y. and Lewis, J. P. (2014). Tuning facial animation in a mocap pipeline. In *ACM SIGGRAPH Talks*, pages 13:1–13:1.

[Shin and Oh, 2006] Shin, H. J. and Oh, H. S. (2006). Fat graphs: Constructing an interactive character with continuous controls. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '06, pages 291–298, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Sloan et al., 2001] Sloan, P.-P. J., Rose, III, C. F., and Cohen, M. F. (2001). Shape by example. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 135–143.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

[Tan et al., 2014] Tan, J., Gu, Y., Liu, C. K., and Turk, G. (2014). Learning bicycle stunts. *ACM Transactions on Graphics (TOG)*, 33(4):50.

[Tautges et al., 2011] Tautges, J., Zinke, A., Krüger, B., Baumann, J., Weber, A., Helten, T., Müller, M., Seidel, H.-P., and Eberhardt, B. (2011). Motion reconstruction using sparse accelerometer data. *ACM Trans on Graph*, 30(3):18.

[Taylor and Hinton, 2009] Taylor, G. W. and Hinton, G. E. (2009). Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1025–1032, New York, NY, USA. ACM.

[Taylor et al., 2006] Taylor, G. W., Hinton, G. E., and Roweis, S. (2006). Modeling human motion using binary latent variables. In *Advances in Neural Information Processing Systems*, page 2007. MIT Press.

[Taylor et al., 2011] Taylor, G. W., Hinton, G. E., and Roweis, S. T. (2011). Two distributed-state models for generating high-dimensional time series. *The Journal of Machine Learning Research*, 12:1025–1068.

[Unuma et al., 1995] Unuma, M., Anjyo, K., and Takeuchi, R. (1995). Fourier principles for emotion-based human figure animation. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 91–96, New York, NY, USA. ACM.

[Vincent et al., 2010] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408.

[Wang et al., 2008] Wang, J., Fleet, D., and Hertzmann, A. (2008). Gaussian process dynamical models for human motion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2):283–298.

[Wiley and Hahn, 1997] Wiley, D. J. and Hahn, J. K. (1997). Interpolation synthesis of articulated figure motion. *IEEE Comput. Graph. Appl.*, 17(6):39–45.

[Witkin and Kass, 1988] Witkin, A. and Kass, M. (1988). Spacetime constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 159–168, New York, NY, USA. ACM.

[Witkin and Popovic, 1995] Witkin, A. and Popovic, Z. (1995). Motion warping. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive*

Techniques, SIGGRAPH ’95, pages 105–108, New York, NY, USA. ACM.

[Xia et al., 2015] Xia, S., Wang, C., Chai, J., and Hodgins, J. (2015). Realtime style transfer for unlabeled heterogeneous human motion. *ACM Trans. Graph.*, 34(4):119:1–119:10.

[Xian et al., 2006] Xian, X., Soon, S. H., Feng, T., Lewis, J. P., and Fong, N. (2006). A powell optimization approach for example-based skinning in a production animation environment. In *Computer Animation and Social Agents*.

[Yamane and Nakamura, 2003] Yamane, K. and Nakamura, Y. (2003). Natural motion animation through constraining and deconstraining at will. *Visualization and Computer Graphics, IEEE Transactions on*, 9(3):352–360.

[Zeiler and Fergus, 2014] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer.

[Zhang et al., 2007] Zhang, L., Snavely, N., Curless, B., and Seitz, S. M. (2007). Spacetime faces: High-resolution capture for˜ modeling and animation. In *Data-Driven 3D Facial Animation*, pages 248–276. Springer.