Master's Thesis
ICA-6030769

Neural Network Based Motion Synthesis for Close Combat in Games

Claudio Freda,
under the supervision of
A. Frank van der Stappen,
Zerrin Yumak

August 29, 2018

Abstract

In this thesis we present a comparison of different neural network based motion synthesis techniques applied to the animation of a sword-fighting character. We record a set of motions in our motion capture lab, documenting the gathering and processing of data. We compare two data-driven models, Encoder-Recurrent-Decoder and Phase-Functioned Neural Networks, by training them to our data set.

Contents

1 Introduction

While there have been several developments in the field of character animation in the recent years, the majority of video game developers prefers to rely on hand-crafted finite state graphs [Clavet, 2016a,b] which are used to select between a number of pre-recorded clips depending on the current state of the character. As the fidelity expected in game animation increases, these graphs have become more detailed and expensive to maintain, while offering diminishing returns in quality the larger and more complex they become, and the limitation of only being able to play pre-recorded clips [Clavet, 2016a,b].

Research in online data-driven motion synthesis has offered an alternative to traditional methods, by making use of directed graphs and machine learning techniques, trained on large sets of motion-captured data. These systems have seen limited adoption for now, due to the strict real-time processing requirements of games and the necessity for animated characters to reliably and quickly respond to user input, two areas in which data-driven motion synthesis still struggles.

One of the first methods to try and solve the problem of hand-crafted graphs is Motion Graphs [Kovar et al., 2002], where unordered clips are sliced and organized into a directed graph through an algorithm. Using a loss function one can navigate the graph to find the optimal sequence of motions for the current action performed by the character. Motion graphs are limited by the fact that they can only combine pre-determined motions, and thus they suffer from slow and inconsistent response to real-time control.

Motion Graphs spawned a series of data-driven methods based on the "bag-of-clips" approach [Arikan et al., 2003; Gleicher et al., 2003; J. Lee et al., 2002; Li et al., 2002]. An improvement upon these methods came with Motion Graphs [Y. Lee et al., 2010]Y. Lee et al. [2010] place the poses in the database on a manifold, blending between the current pose and poses close to it on the manifold. This allows to create motion that did not exist in the source database, which greatly improves the ability of the method to respond quickly to changes.

However, Motion Graphs has been proven to be impractical to implement in a production environment [Clavet, 2016a]. Research into similar, more streamlined, data-driven motion synthesis methods for character actions has been going on internally in EA [UFC on FOX, 2014] and Ubisoft [Clavet, 2016a,b], proving that it remains an important problem to solve. However, while the results are visible in shipped games, the details of the techniques

used have not been made public, with the notable exception of Ubisoft's research on Motion Matching.

Recent developments in machine learning have led to promising results in applying neural networks to motion synthesis. Convolutional Neural Networks [Holden et al., 2016, 2015] have been used to solve this problem, but they suffer from high latency, making them unsuitable for real-time application. Encoder-Recurrent-Decoder RNNs [Fragkiadaki et al., 2015] and Phase-Functioned NNs [Holden et al., 2017] have also successfully been applied to generate the motion of a moving character, but these techniques have not yet been successfully applied to other kinds of animation[1].

While character locomotion is a challenging problem, especially when considering adaptation to uneven terrain and different gaits of motion, it is still relatively constrained in that it is a clearly cyclical action with a limited range of motion. The current neural-network based solutions are highly optimized for character locomotion and the extent to which this transfers to other kinds of animation has not been verified. Holden et al. suggest that their method could be applied to actions such as punching.

We can conclude that there is significant progress to be made in this field. This thesis explores the possible application of different neural network based techniques to the problem of animating a character trying to hit a target with a sword.

[1]While RNNs have been applied to general animation, they have been only used for classification or motion prediction in the short term rather than motion synthesis.

2 Thesis Statement

The following section will present the rationale and research goals for this project.

2.1 Research Problem

Animation in computer games is an expensive and time-consuming process. Current groundbreaking research into the application of Neural Networks to the problem of character locomotion [Fragkiadaki et al., 2015; Holden et al., 2017] shows great promise of future adoption in the field of video game development. Holden et al.'s method, Phase-Functioned Neural Networks, is currently restricted to idle, walking, jumping and running cycles.

Machine learning based methods allow for the automated generation of an animation controller from an unordered set of motion data, usually obtained through motion capture. This approach allows to improve animation quality by taking a larger amount of motions into account, and to reduce the cost of designing the animation system.

Adapting neural network based online motion synthesis to the animation of close combat animation can show the applicability of neural networks to other kinds of animations other than character locomotion. Generalizing the neural network approach can prove useful to research attempting to synthesize other types of character actions, such as ball stopping and kicking in football, throwing objects, catching objects. We have chosen sword-fighting as it is one of the most common forms of motion in action games, while presenting several challenges in synthesizing natural motion.

2.2 Research Question

In this project we attempt to answer the following research question:

Can machine learning techniques be used to synthesize natural-looking close-combat animation for games?

We would like to adapt current neural network based solutions to the problem of animation a character swinging a sword. The aim of the research is to discover whether these techniques are feasible for this kind of application.

We have divided this main question into several sub-questions:

- Is the training time practical for a development studio?
- Can our method be executed in real time in a game engine loop?
- Can our method satisfy the control requirements for interactive games?
- Can our method generate natural-looking motion?

The first question expresses the practical need for training time to be fast. Neural networks take a long time to train. A development studio that might want to apply these methods and get results in a reasonable time frame. To ensure that training is reasonably fast, we will measure training times. PFNN trains in 30 hours on a modern GPU, we are aiming for similar or shorter training times.

The second question expresses the need of fast execution of the method. Modern videogames usually run at either 60 or 30 frames per second. Our method would have to be able to be executed fast enough not to affect frame times significantly. We will measure the per-frame run-time of the network.

The third question expresses the need for the method to react well to user input. The player has to feel in control of the character, this means that response times need to be small and predictable [Y. Lee et al., 2010].

The fourth question expresses the need for the method to generate natural-looking motion (see section 3.4). This is, ultimately, a subjective assessment, but we will focus on observing two qualities of motion: physical realism (how closely the animation respects real world constraints) and variability (variations in motions repeated many times with the same parameters). We will also be considering whether the model is able to produce new motion or merely copy the motions included in the data set.

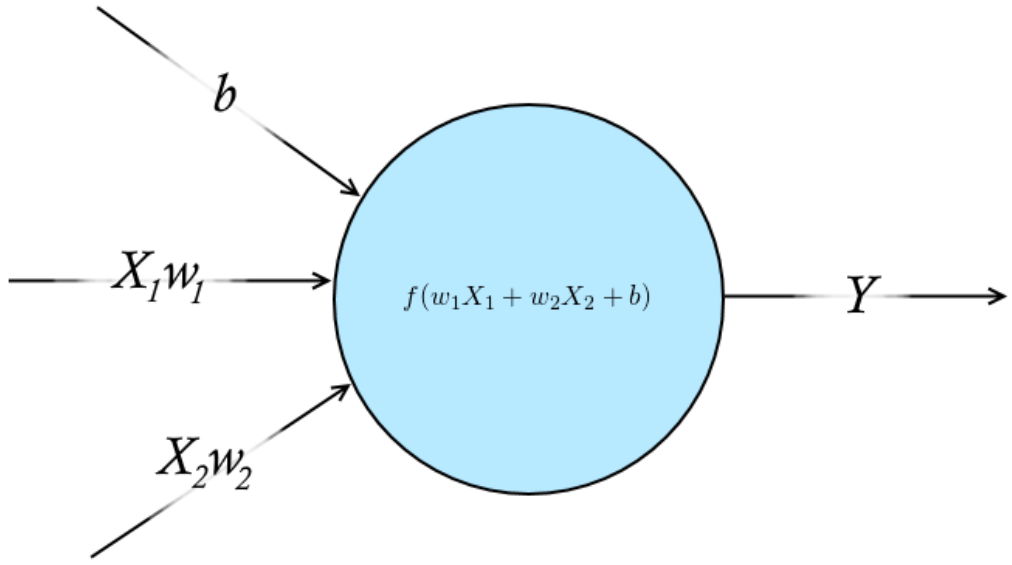

Figure 1: A neuron.

3 Related Work

This section will offer a review of existing work related to our domain of application, neural networks and motion synthesis.

3.1 Neural Networks

A neural network is a kind of statistical model that consists of many simple, connected processors called neurons, each producing a sequence of real-valued activations [Schmidhuber, 2014]. Each neuron has a number of inputs $X_1, X_2, ...X_n$, each X_i having a specific weight value w_i, a bias b, and a single output Y; the output being defined as:

$$Y = f(w_1X_1 + w_2X_2 + ... + w_nX_n + b)$$

where f is a non-linear function called the *activation function*.

Neural networks are usually organized in *layers*, each layer's neurons only being connected to the previous layer. The first layer is called the *input layer*, and its neurons are activated by the model's inputs. The last layer is called

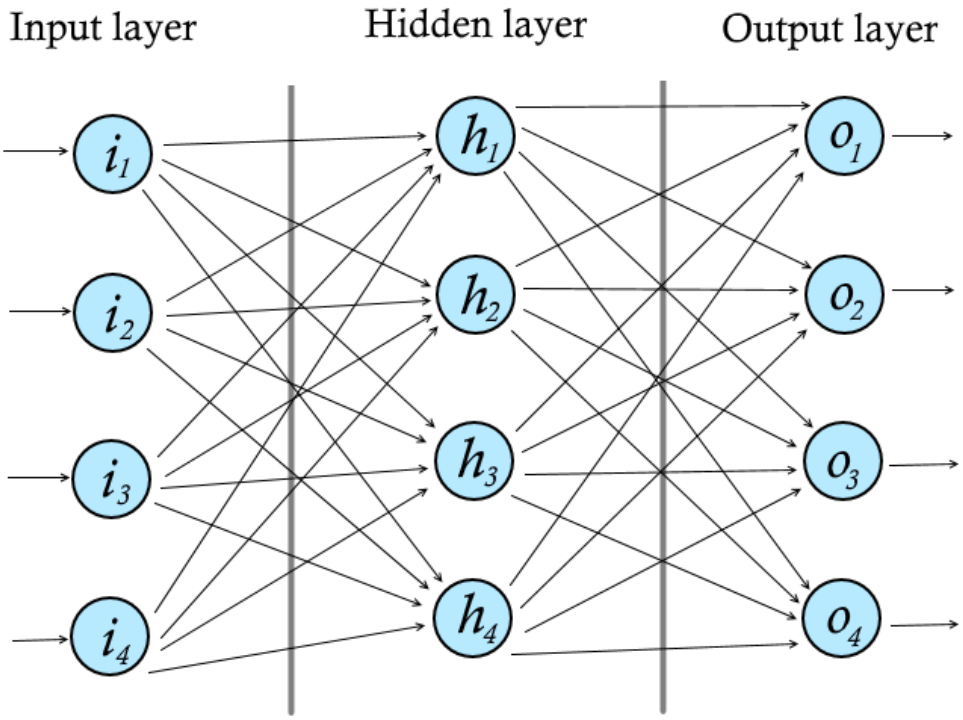

Figure 2: A simple neural network with a single hidden layer.

the *output layer*, and its neurons' outputs are the model's outputs. Finally, the layers in the middle (if any) are called *hidden layers*.

3.1.1 Training

Neural networks are able to model highly complex behaviours, provided they are trained properly. The values of the weights and biases of each neuron can be found using numerous methods, of which the most common is an iterative algorithm called *stochastic gradient descent* (SGD) [Schmidhuber, 2014].

With examples from a training set and an appropriately chosen *loss function*, we can insert the example inputs into the model, and calculate the current output *average error* of the network. With a method called *back-propagation* we can work our way back through each layer and calculate the gradient of the error compared to each weight. Subtracting the gradient multiplied by a parameter λ called *learning speed* from each weight brings us

closer to a minimum of the error.

The *stochastic* part of the algorithm involves approximating the gradient of the average error with the gradient of the error of a single example. The algorithm is then performed on each example in shuffled order, doing multiple passes if necessary until the algorithm converges.

3.1.2 Recurrent Neural Networks

Standard neural networks (also called *feed-forward* neural networks) are acyclic, i. e. the connections between their nodes do not form any cycles. Neural networks containing cycles are called Recurrent Neural Networks.

Recurrent Neural Networks are particularly good at processing sequential data, usually in an autoregressive model (i. e. where the outputs depend only on the inputs and past values of themselves). RNNs possess a form of memory and have been shown to be highly effective at tasks in the domain of natural language processing.

Long Short-Term Memory Standard RNNs are not very good at acting upon data more than a few executions in the past. The standard behaviour for a cyclic node is to recall the value it had at the last execution, but if at this execution the output is different that value will be effectively forgotten.

LSTM units [Hochreiter and Schmidhuber, 1997] provide a substructure that makes sure that the default behaviour is to *remember the value*, which can only be modified if the network actively learns to do so. This dramatically increases the time period in which RNNs are able to remember information.

Encoder-Recurrent-Decoder Fragkiadaki et al. [2015] propose a new architecture for a RNN using LSTM units. They stack three networks in order: an Encoder, a Recurrent and a Decoder. The Encoder and the Decoder are standard multi-layer neural networks, while the middle one is a RNN based on LSTM units. The concept of this architecture is that the Encoder would learn to transform the input data in a representation that is more easy to learn temporal behaviour from; the Decoder then transforms the data again into desirable output.

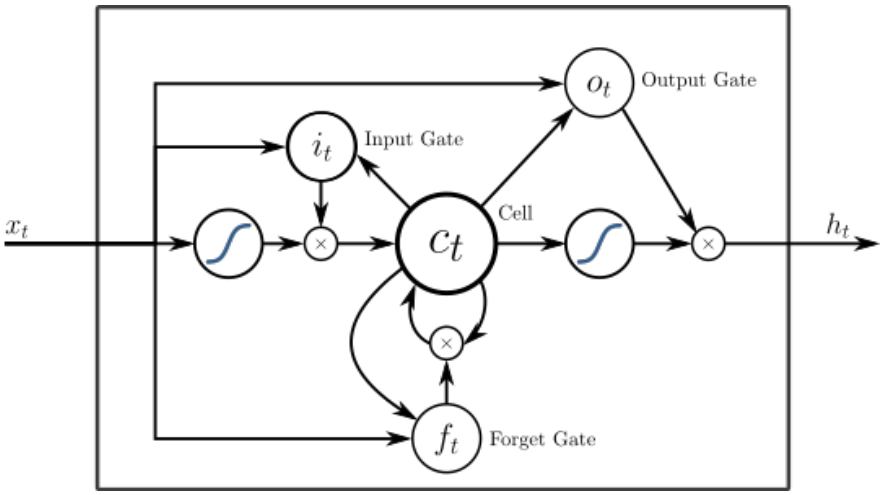

Figure 3: Architecture of an LSTM unit. [Graves et al., 2013]

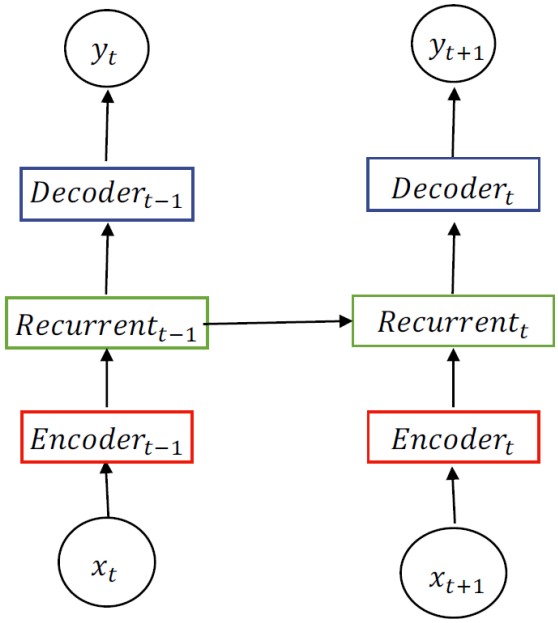

Figure 4: Architecture of the Encoder-Recurrent-Decoder model [Fragkiadaki et al., 2015].

3.2 Representing Characters and Motions

It seems useful to summarize the mathematical concepts involved in the animation of virtual characters. In doing so, we will also establish some definitions for the rest of the document.

Characters In 3D game engines, characters are represented as a polyhedral mesh [Van Welbergen et al., 2010]. Animating the polygons individually is impractical, so they are usually mapped with weights onto a skeleton, i. e. a hierarchy of segments connected by joints. Each joint can be rotated and/or translated, affecting all the joints lower in the hierarchy; depending on the joint's constraints, it can have up to 6 degrees of freedom, 3 for rotation and 3 for translation, which we will call *joint parameters*.

The complete set of joint parameters for all the joints will be called the *joint parameter vector*. A value of this vector defines a *pose*, i. e. a single configuration of the skeleton.

Motions A motion is very simply a continuous sequence of poses. Mathematically, it can be represented as a continuous function $\mathcal{M}(t) : \mathbb{R} \to \mathbb{R}^d$, where d is the cardinality of the joint parameter vector. To represent continuous movement in computer animation, $\mathcal{M}(t)$ is sampled at discrete intervals and interpolated.

With motion capture, motions are sampled at the update rate of the motion capture system, usually 60 Hz or faster; interpolation between samples usually linear.

With artificially created sequences, motions are usually sampled with more granularity. At specific time instants (keyframes) a pose is recorded, including information on the kind of interpolation method to use with adjacent keyframes.

3.3 Motion Synthesis Overview

What is motion synthesis? Although the term is often used, no-one gives a strict definition. Kovar et al. [2002] describe "synthesizing streams of motions". Pejsa and Pandzic [2010] talk about "dynamically synthesizing new, controllable motion". Van Welbergen et al. [2010] try to avoid the term, referring generally to *animation techniques* instead. We will define motion synthesis as the generation through algorithms of new motion sequences, i. e. motion data that did not exist before.

The main objectives in developing a motion synthesis technique are *naturalness*, i. e. how real the animation looks, and *control*, how effectively and quickly it can be changed. Every technique is the result of a compromise between these two qualities [Van Welbergen et al., 2010], and they can be classified based on how much they favour one over the other.

The two main application fields for motion synthesis are offline animation (e. g. films) and online animation (e. g. games). Although the techniques discussed can be applied to both, they will be presented in the context of online animation, as it is what this project focuses on. Online applications favour control over naturalness [Clavet, 2016a; Pejsa and Pandzic, 2010], therefore the methods that will be presented in detail in later sections will be ones that attempt to reach a higher standard of naturalness without sacrificing control.

Control of animation techniques is usually achieved through the combination of parametrisation and concatenation, to achieve specific control requirements (e. g. moving at the specified speed or looking at the specified direction). Parametrisation is the selection of the appropriate parameters of the animation technique that satisfy the control requirements. Concatenation is the selection of a sequence of motions that, when chained together, satisfy the control requirements.

3.3.1 Simulation

One approach to create new motions is to use mathematical models. Van Welbergen et al. [2010] call this Simulation. We can distinguish between procedural simulation, which uses mathematical formulas to describe either the joint variables or the path of specific points called effectors (e. g. hands or feet), and physical simulation, which uses forces and torques applied to the joints together with numeric integration and a dynamic model of the

character.

Simulation techniques tend to have very high control but a low degree of naturalness [Pejsa and Pandzic, 2010; Van Welbergen et al., 2010]: even when using physical simulation, the model is still simplified compared to reality. Even in games, which usually strive for high control over naturalness, they are used only in specific situations, e. g. ragdolls for unconscious or dead characters.

Despite this, simulation-based techniques are making significant strides in recent years thanks to muscle-based models [Geijtenbeek et al., 2013], and the Euphoria engine by NaturalMotion has been used in several high-profile game releases, most importantly the Grand Theft Auto series, as an improvement over ragdolls for the animation of characters that are falling or out of balance.

Simulation-based techniques will not be expanded upon in this document, as it is not the approach chosen for this project.

3.3.2 Example-Based Motion Synthesis

The second approach to motion synthesis is Example-Based or Data-Driven Motion Synthesis [Pejsa and Pandzic, 2010], also called Motion Editing [2] [Van Welbergen et al., 2010]. This includes every technique that generates motion data by deriving it from one or more motions from an existing database. The existing database can be obtained by motion capture or manual keyframing.

We can distinguish between Motion Modification, where motions are generated based on one example motion primitive, and Motion Combination, where motions are generated based on one or more motion primitives. Modification techniques are usually corrections to a motion, and they can be applied to motion generated by motion combination. For example, both Clavet [2016a] and Holden et al. [2017] use inverse kinematics after a motion combination technique to correct feet positions on uneven terrain.

Combination can be further divided in blending and statistical models [Van Welbergen et al., 2010]. Blending is when a set of motions is interpolated, with the weights as parameters. Usually the animations in the set are similar in time and joint variables at key events (e. g. foot contacts); if they are not, visual artifacts such as foot skating will occur. The other approach is

[2] Van Welbergen et al. consider some of the techniques listed, such as Motion Graphs, as more general concatenation techniques that use Motion Editing to generate transitions. For the sake of simplicity, the distinction has been avoided here.

to train a statistical model on a large set of example motions; in this case, a complex model will be able to handle a more diverse set of training examples.

It has to be noted that Motion Editing and Simulation can be combined. Motion Editing can be used for standard character locomotion, switching to Simulation when more interactivity with the environment is needed. Another example is when different sub-trees of the joint hierarchy are be animated by different methods [Van Welbergen et al., 2010].

We will now summarize the most important example-based animation techniques.

Finite State Machines (FSMs) are the most basic of Example-Based methods, often used in commercial games [Clavet, 2016a]: characters are manually organized into states (e. g. running, walking, jumping), and animations are concatenated and blended when switching from one state to the other. Plain FSMs are the baseline with which other methods need to compare themselves.

Motion Graphs [Kovar et al., 2002] employ concatenation and blending like FSMs, but act on a much larger motion database ("bag-of-clips"). To navigate this database, a graph is automatically constructed, linking together similar poses through *transitions*. Poses are determined to be similar with the help of a distance function. A motion can then be constructed by walking the graph, selecting the path that best meets the control requirements through a search algorithm.

Motion Fields [Y. Lee et al., 2010] are an improvement upon "bag-of-clips" approaches such as motion graphs. They build a motion database made of motion states, containing both the position and velocity of joint parameters. They define a motion field as the interpolation between the k nearest motion states to the current state; new frames are produced by choosing a set of weights for the interpolation and integrating accordingly. Weights are chosen using a Markov Decision Process with an appropriate reward function.

Inspired by Motion Fields, Clavet [2016b] proposes a simpler, more optimized, method that leads to similar results. At every frame, the next pose is chosen by iterating over all possible poses in the motion database and choosing the one with the least cost, using a cost function that includes both joint variables and control parameters.

Statistical Methods involve training a statistical model over a motion set to output animation data. Methods that have been tried for this include Principal Component Analysis [Glardon et al., 2004; Troje, 2002; Urtasun et al., 2004], Gaussian Processes [Wang et al., 2008; Zhou et al., 2014], Boltzmann Machines [G. W. Taylor and Hinton, 2009], and Recurrent Neural Networks [Fan et al., 2015; Fragkiadaki et al., 2015; S. Taylor et al., 2017].

More recently, Holden et al. [2017] train a neural network specifically designed for the animation of walking and running characters. The network's weights change based on a *phase function*, that assumes a value from 0 to 1 depending on the current phase of the walking cycle. The inputs are current joint variables and control parameters, while the outputs are updated joint variables and new value of the phase function.

3.4 Naturalness vs. Control

When developing a new animation technique, it is important to understand how to measure its effectiveness.

Van Welbergen et al. [2010] investigate the relationship between two properties of interactive real-time animation they call *naturalness* and *control*. Specifically, they find that as one increases the other decreases, and thus animation techniques can be evaluated based on how much they privilege one over the other.

Naturalness is the perceived realism of the animation, divided in:

physical and biological realism, how closely the animation respects real-world constraints;

whole body involvement, how well the body coherently accompanies the motion;

style, the particular way in which a motion is performed, representing personal characteristics of the subject, e. g. age, gender, personality;

variability, the differences in the motion when repeated many times.

Control is the effectiveness with which the animation can be interactively influenced through control parameters. It is divided in five aspects:

responsiveness, how quickly the desired change is enacted;

precision, how close the animation is to the control parameters;

coverage, how much of the control parameter space is covered, i. e. results in valid animations;

expressiveness, the amount of control parameters available;

intuitiveness, how much the animation parameters describe aspects of the motion, and can be directly used as control parameters.

These definitions are helpful in describing the virtues and flaws of existing animation techniques, and in setting appropriate goals when developing new ones.

3.5 Motion Blending

Section 3.3.2 described different categories of motion synthesis. We would like to focus on *motion combination*, generating new motions starting from more than one example motion. In particular, *motion blending* is when the new motion is generated as a weighted interpolation of existing motions.

3.5.1 Finite State Machines

The most used animation technique in video games at the moment is state machines [Clavet, 2016a]. The character is organized in a set of states (e.g. walking, running, turning, standing, stopping movement, attacking) and one motion is associated to each state: when moving from one state to the other, the motion changes as well.

In modern applications, FSMs are augmented with other techniques: hard transitions can be softened by blending, blend spaces can be used to parameterize motion for a small amount of variables (e.g. the aim direction of a firearm), and masking can be used to separately animate different bone hierarchies.

While FSMs are usually classified as a high-level control mechanism rather than a motion synthesis technique, in practice the presence of transition blending and blend spaces, combined with minutely designed state machines, makes this kind of usage comparable to other motion blending techniques. However, one must take notice that FSMs can also simply be used as high-level "glue" to switch between different animation techniques.

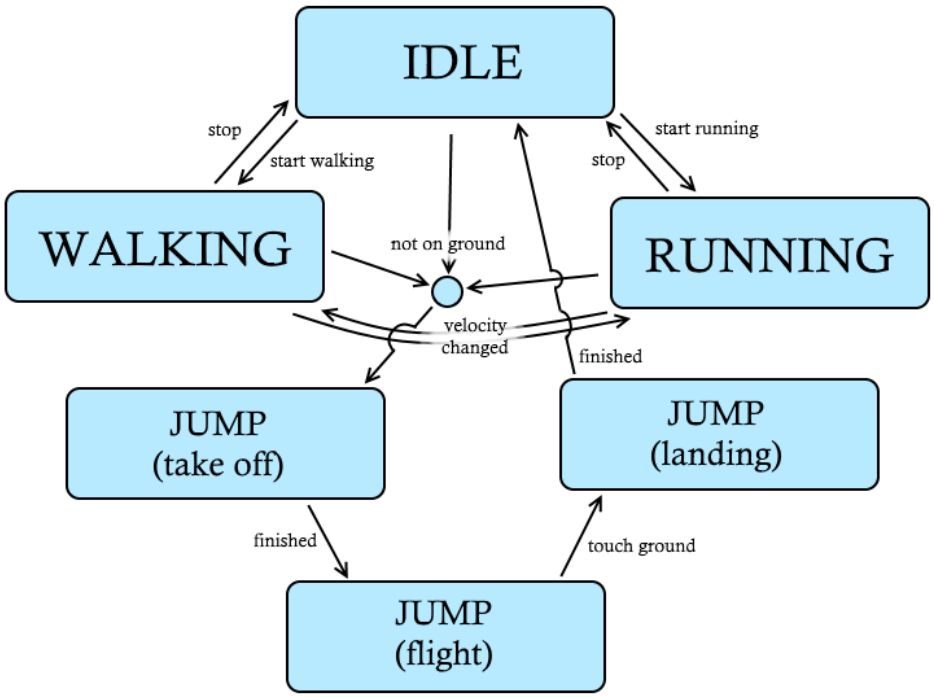

Figure 5: Example finite-state machine for the animation of a simple character.

State machines can be organized in hierarchies and accurately tuned to achieve an acceptable level of naturalness, but the process is excruciatingly time-consuming [Clavet, 2016a]. In addition, state machines are not able to produce natural motion when the character encounters something that the developer did not expect.

When tight control is an issue, the root node is moved procedurally while the character is animated in place [Pejsa and Pandzic, 2010]: this is a crude compromise that ensures that gameplay control constraints are met without sacrificing too much naturalness. Unfortunately, this brings about some artifacts and unnatural looking animations, especially during state transitions, where the animation does not fully match the procedural movement. This effect can be seen in any of the Dark Souls series games: when changing direction quickly while running, the character will simply be rotated while performing the forward run motion.

While state machines are easy to implement, they must also be prepared by hand. A state machine with few nodes does not offer much in terms of naturalness, as jerky transitions between states will be quickly noticeable. To obtain better results, numerous micro-states and transition-states have to be created, each with its own associated motion, which is a hard and time-consuming process with diminishing returns [Clavet, 2016a,b].

3.5.2 Motion Graphs

One of the first effective methods for interactive motion synthesis was Motion Graphs [Kovar et al., 2002]. Motion graphs work by organizing an unordered set of motions into a flat directed graph, made up of clips of motion and transitions connecting them. Generating a proper motion becomes a problem of finding an appropriate *graph walk*, which can be selected by borrowing search algorithms from graph theory.

Motion Graphs is inspired by the work of Arikan and Forsyth [2002], introducing the idea of example-based motion synthesis and building a graph-based structure to simplify searches. Kovar et al. spawned a broader family of techniques based on the generation of a data structure around an unordered set of motions.

Around the same year, several related works were published: J. Lee et al. [2002] use a similar approach in building the database, but employ statistical models to select the motions, Li et al. [2002] use a probabilistic model of future frames of animation called *motion texture*, Arikan et al. [2003] intro-

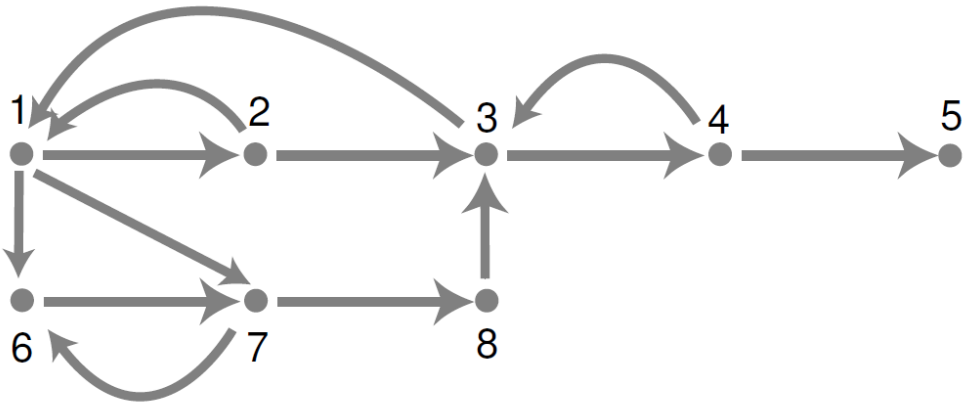

Figure 6: An example motion graph. [Kovar et al., 2002]

duce the idea of annotating the motion database to allow for more precise directing of the synthesized motion, and Gleicher et al. [2003] improve the automatic generation of transitions and introduce multi-way transitions.

Graph Building The motion graph is defined as a set of directed edges, associated with motion clips, and nodes, choice points associated with single animation frames. By following any path in the graph it is possible to synthesize a continuous motion. The edges are either parts of the original unordered set of motions, or transitions created by motion blending.

To generate the graph, every pair of frames in the database is compared using a *frame distance metric*, forming a 2D error function that is finally searched for local minima. The minima lower than a specified threshold are selected to form transitions. Transitions are generated by incrementally blending the preceding k frames of the first frame with the following k frames of the second frame. Finally, the resulting graph is pruned by removing sinks and dead ends.

Motion Search While some methods perform a global search [Arikan and Forsyth, 2002; Arikan et al., 2003], Kovar et al. [2002] only search locally, using branch and bound limited to a specific graph depth of a number of frames, then advances a single frame and executes the search again. This

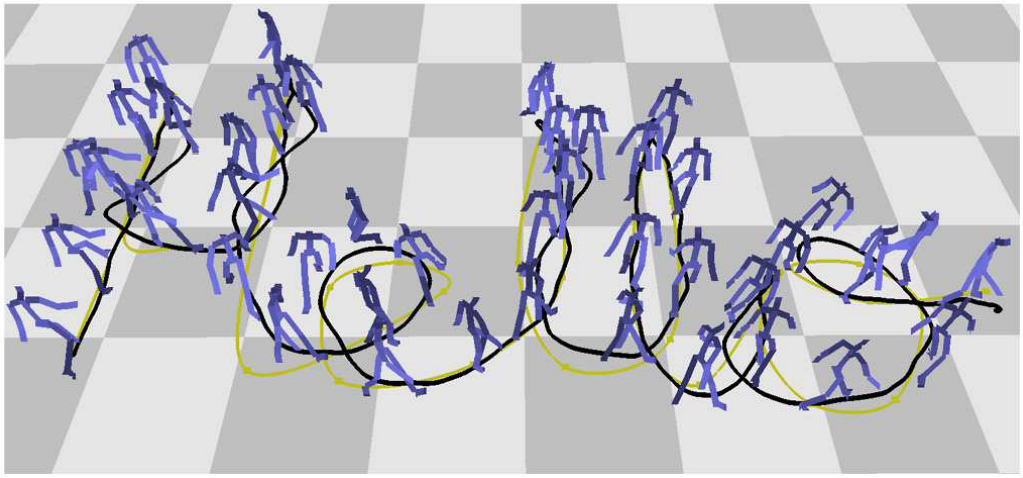

Figure 7: An example motion following a "hello" trajectory computed using Motion Graphs. [Kovar et al., 2002]

prevents search times from growing exponentially and maintains the algorithm responsive and interactive.

Results Motion Graphs is an often-cited method in the field of motion synthesis, as with a proper starting motion set it is able to produce motion that looks very realistic, but recently it has received some criticism. Y. Lee et al. [2010] document the high variance of the time elapsed between an abrupt change in input and the appropriate response. Slow, unreliable response times are due to the lack of guarantees that a fast walk between the current pose and the target pose is always possible.

3.5.3 Motion Fields

According to Y. Lee et al. [2010], methods based on unordered sets of motions such as motion graphs tend to give very high quality animations, but to be quite unresponsive. Their method, based on *motion fields*, generates motion frame-by-frame instead of relying on limited clips.

Motion Fields are a structure that approximates the possible natural motion as an interpolation between the poses from a motion set that are most similar to the current pose, the motion set being obtained through motion

capture. This way, Y. Lee et al. are able to synthesize new animation that looks natural but is also different than the source data.

This section will present how the motion database is built and how the choice is made to navigate the motion field.

The Motion Database First of all, a *motion database* is built from *motion states*, which are pairs (x, v) of the pose and velocity vector of all joints. Motion states are constructed from a pair of successive frames of the motion captured data. A *neighbourhood* of a state as the k nearest neighbours from the motion database, and the *similarity weights* as the normalized distances of each neighbour from m.

The space of available motion, or rather the states that a starting motion state m can move to during the length of a frame is defined as a weighted interpolation of the states in the neighbourhood of m. This is what they call the *motion field*. At each frame, by choosing an appropriate *action* (i. e. a set of weights) for the motion field, we can integrate to get a new motion state for our next frame of animation. This will be an original frame, created as the result of integration over the weighted interpolation of the velocities of the k neighbours. A small corrective factor is added to prevent the character from straying too far from the recorded data.

Choosing Actions Weights are chosen using a Markov Decision Process. A *finite* set of k actions is sampled. Each action's weights will be the similarity weights, slightly skewed towards one neighbour, resulting in each one of k actions approaching one of k neighbours. A reward function is used to choose the best action.

The basic reward increases in value the closer the character is to meeting the control requirements. To account for the long-term effects of choosing an action, a *value function* is added, defined as the sum of all future rewards obtained by acting optimally after choosing that action. The value function is not computable, but it can be approximated if one defines the function recursively as the sum of the reward function and the value function of the next state.

By iteratively calculating the value functions over all states we can have an estimate of the value functions for each state. Value functions have to be computed for each state for each control variable, so to prevent extraordinary memory usage Y. Lee et al. use time compression: recognizing that

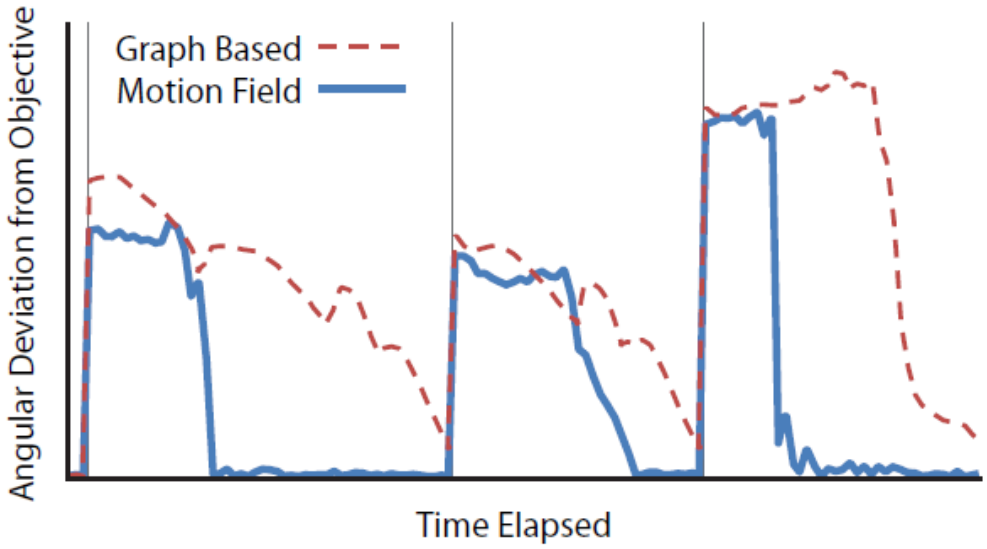

Figure 8: The improvement in responsiveness of Motion Fields compared to Motion Graphs. [Y. Lee et al., 2010]

the motion states were originally taken from continuous motions, they select specific frames in those motions as *anchor states*. Value functions will only be computed for anchor states, and interpolated for the rest.

Results Y. Lee et al. compare motion fields with their own graph-based state-of-the-art controller [Y. Lee et al., 2009], with impressive results, improving significantly response times both in average and in variance, without significant loss of naturalness.

3.5.4 Motion Matching

Motion Matching [Clavet, 2016a,b; Zadziuk, 2016] is a method inspired by motion fields, designed to provide a simpler implementation while producing similar results. Firstly, the motion database is sampled at a discrete rate to improve performance. Secondly, at every frame, the next pose is chosen with a search among all possible poses in the compressed motion database. Zadziuk [2016] also contributes some improvements in the efficient recording of motion data using *dance cards* and in using the system for making the

character react to dynamic stimuli.

Preparation Firstly, poses are sampled in advance from the unordered motion set at an interval of 0.1s, under the assumption that poses that are close together in time will be very similar, to improve search speed.

Similarly to [Arikan et al., 2003], Motion Matching requires the animations to be manually marked up and cleaned: important events are marked up throughout the animations so that they can be searched for later.

Pose search At every iteration, the current pose is compared to the sampled pose set, evaluating the one with the lowest total cost, the cost being the sum of a term based on control variables and a pose distance function. If any pose has a lower cost than the current pose, the character switches to the animation clip containing that pose with a quick blend. The pose distance function only takes into account the most important joints to improve speed.

More specifically, the cost function is made up of the current cost, i. e. how far is the candidate pose to the current pose in terms of velocity and position of both the whole character and the joints, and the future cost, i. e. how well it approximates the estimated trajectory and pose that the character will have in the future according to the control parameters. Future costs are calculated with the assumption that the character will not change animation again once switched.

Dance cards Zadziuk [2016] replicated the Motion Field approach, but working with longer, 10-15 minutes clips of motion data.

Zadziuk introduced an approach for efficiently recording a great quantity of motion captured data that can then be matched in diverse movement situations. He develops a *dance card*, a cheat sheet for the actor to show all of the motions that he has to perform. Through experimentation they arrive at a very extensive set of motions including strafing, dodging, jumping, sneaking, running, jogging, and walking.

Dynamic reactions By using impulse forces as control variables, and recording shoving and falling animations, Zadziuk [2016] discovered that by adding an impulse force as a control variable he was able to have the character react realistically to receiving dynamic impulses. This goes one step

further than Motion Fields, as Y. Lee et al. only managed to model the recovery from the impulse. Zadziuk manages to animate the character naturally even at the immediate impact, achieving a similar effect to physically-based methods such as [Geijtenbeek et al., 2013].

Results The method results in a high degree of naturalness and control. Questions remain open on the scalability of motion matching with a larger motion set, and on the possible edge cases ignored by the simplified approach. Ubisoft was very confident in the quality of this method, which is why an animation system based on motion matching was shipped in their game "For Honor".

3.6 Statistical Motion Combination

Where motion blending combines motion by selecting a blend function, a number of starting poses, and appropriate weights, statistical methods deal with training a statistical model to generate motion on the fly. Deciding what motion to perform is thus reduced to determining the input variables to the model.

We will provide a brief overview of the various statistical models that have been used for motion synthesis.

Principal Component Analysis Principal Component Analysis can be used either to generate motion online [Glardon et al., 2004; Troje, 2002] or to transfer the style of an actor over an already existing set of motions [Urtasun et al., 2004]. While PCA is extremely useful for data analysis and interpretation, it does not offer the same quality as other methods when used to generate online motion, nor can it learn to synthesize motions in more complex scenarios that involve avoiding obstacles.

Gaussian Processes Methods based on Gaussian Processes [Rasmussen and Williams, 2006] have been successfully used for generating online motion in a number of different contexts, using semi-parametric latent factor models [Zhou et al., 2014] and GP latent variable model [Wang et al., 2008]. GP-based methods tend to have a high memory and execution cost, over-fit easily and can not learn complex motion scenarios [Holden et al., 2017].

Boltzmann Machines and Neural Networks Autoregressive models based on Restricted Boltzmann Machines [G. W. Taylor and Hinton, 2009] and Encoder-Recurrent-Decoder models (see section 3.1.2) [Fragkiadaki et al., 2015] have been attempted, but while the execution, memory cost, and scalability is better than either GP-based [Wang et al., 2008] or linear autoregressors [Xia et al., 2015], they still suffer from the so-called "dying out" and "exploding" artefacts [3] [Holden et al., 2017].

Another place where RNNs have had a lot of success is facial animation: Long Short-Term Memory (see section 3.1.2) based models have given impressive results [Fan et al., 2015; S. Taylor et al., 2017], although these models are designed for facial animation and not well-proven in full-body character animation. In addition, the usage of a sliding window technique prevents their usage in situations where it is required to react to control variables in a time smaller than the window length, e. g. responding to user input.

The most recent developments in the field of statistical motion combination are Convolutional Autoencoders and Phase-Functioned Neural Networks; they will be explained in more detail in the next section.

3.6.1 Convolutional Autoencoders

Holden et al. [2015] present the concept of a *motion manifold*, a low-dimensional mapping of the high-dimensional space of joint variables. This is something already done, albeit only on a limited set of motions, by Chai and Hodgins [2005] and Y. Lee et al. [2010].

Instead, Holden et al. train a convolutional autoencoder to learn the manifold of all possible human motion, based on the CMU database. Based on this work, Holden et al. [2016] develop a motion synthesis framework based on the motion manifold by stacking a feedforward neural network onto the manifold and training it to map high-level control parameters to the manifold space.

According to Holden et al. [2017], Convolutional Neural Networks lend themselves more to applications where the entire input is given at once and the entire output is calculated at once. While this is fine for offline applications, it is undesirable for online applications such as video games.

[3] "Dying out" is what happens when motion from different frames is erroneously blended, causing the output to be an average of poses which are supposed to be at different points in time, which tends to look unnatural. "Exploding" is what happens when output high-frequency noise is fed back into the system, causing a positive feedback loop.

3.6.2 Phase-Functioned Neural Networks

Another way to approach the use of neural networks for character animation is to use a normal feed-forward regression network that takes a character pose as input and outputs the character pose for the next frame of animation. Standard neural networks are ill-suited for this approach [Holden et al., 2016], as they average together different phases of the animation, giving the character a floating appearance.

Holden et al. [2017] propose a customized feed-forward network with a single recurrent element called the *phase*, a variable representing the timing of the motion cycle. The PFNN works by generating the weights of the network at each frame as a function of the phase. This allows to keep the network compact, fast to train, and fast to execute, while providing the expressiveness needed to properly represent motion cycles. The network is explicitly tailored towards character locomotion and walking/running cycles.

Data Collection and Processing Initially, the motion capture data is recorded in studio, making sure to cover a variety of situations. The recorded data is automatically phase labelled by using the foot contact points as reference, and then gait labelled (e. g. walking, running, jogging) by hand.

The trajectory for each motion is computed. The root position is calculated as the projection of the hip on the ground, while the direction is taken by averaging the vector of the hip joints with the vector of the shoulder joints, and cross multiplying it with the up unit vector. There being no terrain data, movement is fitted to the best match from a database of height-maps. The terrain height at the root point of the character is then extracted for every motion.

Network structure and training The phase-functioned neural network is a three-layer neural network whose weights change according to a phase function. The phase function chosen by Holden et al. [2017] is a cyclic spline with four points. This means that the network will be trained on four sets of weights, and it will be smoothly interpolated depending on the phase value.

The input variables are the phase, the control variables, the joints' positions, orientations and velocities, the desired trajectory as sampled at equally spaced intervals, and the terrain height at two additional points 25 cm to the left and to the right of every trajectory point. The output variables are the joints' positions, orientations and velocities, and the new value of the phase.

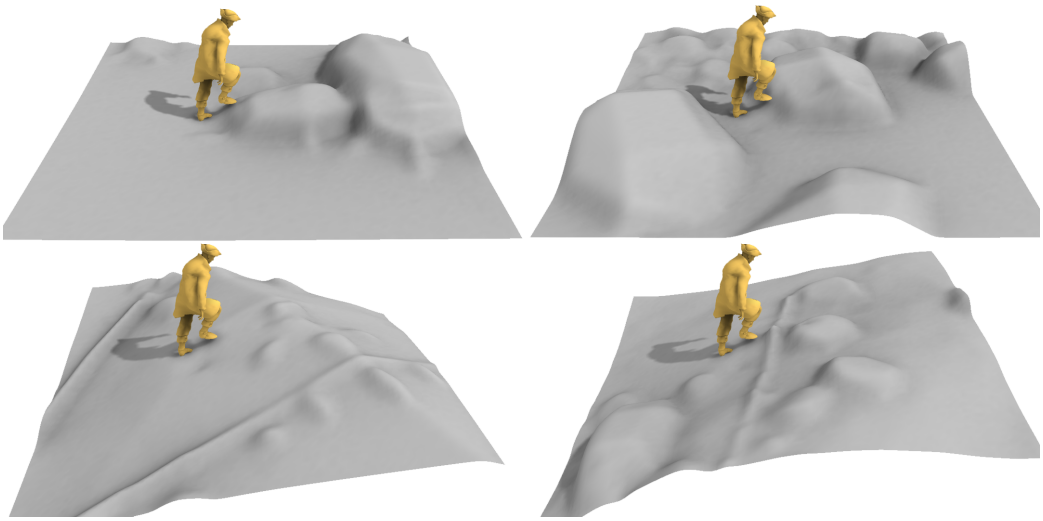

Figure 9: Fitting the motion to a database of terrains. [Holden et al., 2017]

The number of hidden units per layer is 512 and the activation function used is a ReLU (Rectified Linear Unit), which has been proven to significantly decrease training times [Holden et al., 2015]. The ReLU is defined as:

$$f(x) = \max(0, x)$$

To train the network, a stochastic gradient descent algorithm is used. The input, output, and phases are stacked into X, Y, and P matrices, and normalized. Then the problem can be expressed as optimizing this cost function with respect to β.

$$Cost(X, Y, P; \beta) = |Y - \Phi(X; \Theta(P; \beta))| + \gamma|\beta|$$

where Φ is the neural network, Θ is the phase function, β is the set of weights for each of the four phase points, and γ is a small bias to prevent the weights from becoming too large. Training takes 30 hours on a NVidia GTX 660 GPU.

Trajectory extraction Before talking about the neural network actually running, one must ask themselves how the desired trajectory will be calculated from user input. Holden et al. [2017] blend the straight trajectory generated by the direction of the control stick with the trajectory estimated

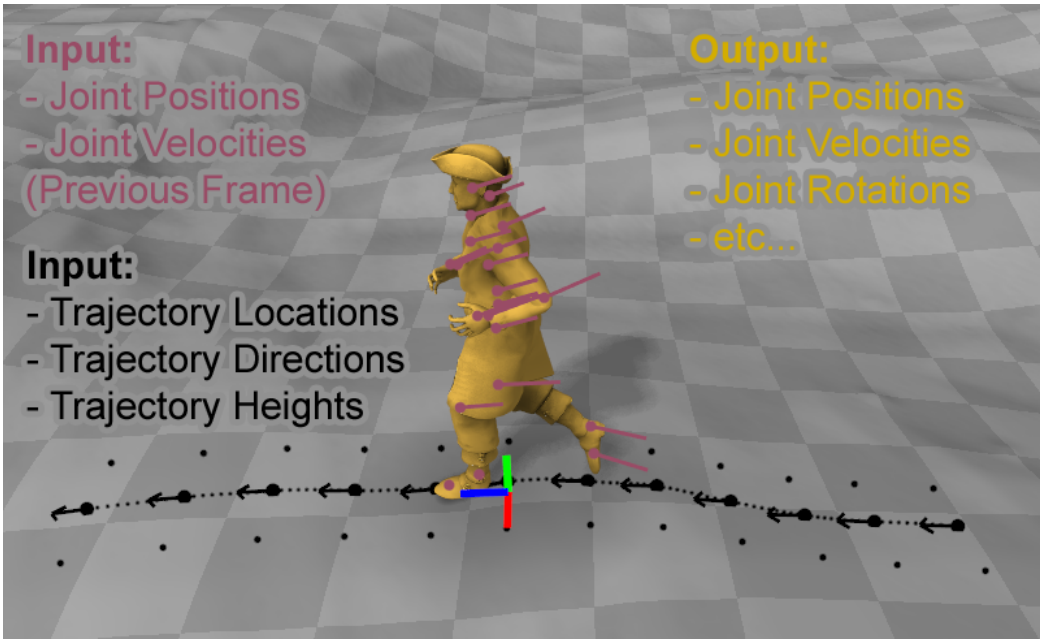

Figure 10: Input and Output parameters in PFNN. [Holden et al., 2017]

in the previous frame. However, the method should work independently of the trajectory.

For blending, they use the following function:

$$TrajectoryBlend(a_0, a_1, t, \tau) = (1 - t^\tau)a_0 + t^\tau a_1,$$

where t goes from 0 to 1 as the trajectory extends into the future, τ is a weighting parameter to control the responsiveness of the character, and a_0 and a_1 are trajectories. The parameter τ has been selected to be 0.5 for the velocities but 2 for the directions. This gives a result where a character will be highly responsive but also feel real and heavy.

The trajectory is then projected onto the world terrain to obtain the height parameters. The input variables related to the desired gait, instead, are assigned based on user input, or in the case of crouching based on the height of the ceiling.

Phase function pre-calculation The phase function takes about 1 millisecond to calculate; this is relatively slow compared to the rest of the neural

network. Holden et al. [2017] actually sampled the function in small steps for the whole of the $[0, 2\pi)$ interval and interpolated between them linearly. This makes the function very cheap.

Results The main accomplishments of this method are the exceptional run-time performance, the ability to adapt to variably-sized obstacles, the high responsiveness, and the high scalability with large data-sets and high-dimensional data. From the naturalness-realism

The performance per frame in the linear kernel implementation of this method goes as low as 0.0008 seconds, with a memory consumption of 125 MB. The training time, however, is very slow (30 hours), but this is still a good trade-off because run-time performance is the most important factor in interactive games.

The character adapts well to a wide range of terrain, climbing rough hills, balancing on narrow bridges, jumping over small obstacles, and crouching under low ceilings. It also supports strafing, as seen in many third-person shooters.

The responsiveness is measured by comparing an input trajectory with the actual resulting trajectory. An average error below 20 cm is expected, and by increasing the τ parameters it is possible to reach much lower margins of error, at the expense of animation quality.

Finally, PFNN is more scalable over high-dimensional large data-sets simply because neural networks are very scalable in that context. The limit to the amount of data that PFNN is able to process is very high, and it is only limited by the training time. An evolution of this method might very well be used in the future to dynamically generate the movement profile of an actor over a few recording sessions.

The main limitation of PFNN is that it is explicitly designed to work around the characteristics of walking and running cycles, and it is not trivially generalisable to other kinds of animations.

4 Research Method

In this thesis we present a comparison of two data-driven motion synthesis models and their application to the animation of a character using a sword. The first model is a Phase-Functioned Neural Network (PFNN) [Holden et al., 2017] and the second is an Encoder-Recurrent-Decoder (ERD) [Fragkiadaki et al., 2015] network.

In this section we will provide an overview of the general approach. In section 4.1 we will explain how the data is gathered, and after that in section 4.2 how it is processed. In section 4.3 we explain how the networks are configured, and in section 4.4 how they are trained. Finally, in section 4.5 we explain how the models work during runtime.

Both PFNN and ERD are used for generating motions online, i. e. in real time. Instead of generating a whole motion clip by itself, the models are set up to work on a frame-by-frame basis, where the data from one frame of animation is used as part of the input to the network, which then outputs the next frame of animation. This is also called an *autoregressive model*.

In particular we must note that ERD in the original paper was not used to synthesize motion, but to classify motion or predict motion in the future. However, Holden et al. implemented ERD as a motion synthesis method to compare against their own PFNN. This works because effectively, predicting motion in the future is the same as generating new motion: past and present data is used to generate future frames of animation.

Characters are represented as a hierarchical structure called a skeleton, made up of joints; the current configuration of the joints is represented by a *joint vector*, composed of all the rotations and positions of the joints that make up its skeleton. An instance of the joint vector represents a pose.

The character's motion is simply a sequence of poses, sampled at a constant time interval. We are strictly considering real-time application, so particular attention will be placed on the *current pose*. We refer back to section 3.2 for a full explanation of how characters and motions are represented.

The sword is simply represented by an additional joint, bound to the root joint. The sword is tied to the root joint instead of a hand joint because it is not physically attached to a hand, and it can be shifted from hand to hand or held with both hands.

As already said, both PFNN and ERD are used in an auto-regressive configuration, which means that they use the joint vector at time t as part of their input, and the joint vector at time $t+1$ as their output; an additional

control vector representing the current goal is appended to the input vectors. Auto-regressive models are particularly suited to real-time processing as they allow to generate motion on a frame-by-frame basis.

During execution, the models are used to generate the pose in the next frame of animation, which then is used to re-evaluate the input vector for the next frame. Repeatedly re-evaluating the input and output vectors allows to generate motion in response to the control vector.

The networks are trained over a set of motion captured data recorded in the motion capture laboratory at Utrecht University with a Vicon Blade system. The motion captured data was manually cleaned and labeled, and then processed to present it in a format more suitable for the learning process. In particular, we choose a relative frame of reference related to the projection of the character on the ground.

To prepare for training, a pair of adjacent frames at times t and $t + 1$ is thus turned into a data point, consisting of an x_t and y_t vector. At this point the *control vector* is evaluated, representing the goal at the time of recording, and appended to the x vector. The control vector is composed of the target point, the current trajectory of the character in a neighborhood of time, and an arbitrary value called *phase* representing the current state in the attack cycle.

Both networks were also trained over the locomotion data set used by Holden et al. [2017], firstly to check for the correctness of our implementation, and secondly to compare the results by using the same models to synthesize locomotion instead of sword animations.

In the following sections, we will explain how the data is gathered and processed, how the models are structured, how they are trained, and how they are executed.

4.1 Data gathering

Our models need to be trained on an unordered set of motions. There is no publicly available data set for sword animations, which makes it especially important to carefully decide how to gather our own data.

Before recording, we must define the domain of motion that we are trying to generate; this is helpful in choosing the optimal recording strategy in order to ensure maximum coverage of this domain. A proper strategy also helps our data set to be more efficient, i. e. to cover a wider range of motions with less data.

1. Firstly, the character will be holding and moving a two-handed sword with a length of 115 cm, which is a common measure for a sword. The handle is 30 cm long and the blade is 85 cm long. We considered a weapon of a fixed size to simplify our domain.

2. We are considering the character to be attacking a target point. Since we do not want to model the action of moving towards the target from a distance, we only consider targets within a range of 2.5 meters, which is approximately the length of the sword, plus the length of an arm, plus the space the actor is able to cover in a wide step.

3. The character will be performing a number of attacks in sequence, one after another; this means that for every recording take the target will move around in a small area and perform multiple attacks. We record sequences of attacks instead of isolated attacks to have enough data to represent transitions between attacks, and not just to and from a neutral pose.

4. The target will be allowed in any direction with respect to the character. It is important that the model is able to adapt and aim towards targets in any direction.

5. We will be considering a varied set of thrusts and swings, horizontal, vertical and diagonal. We are already recording multiple attacks and usually it is not practical or natural to perform the same kind attack twice in a row. Additionally, there is no much point in developing a model that can only perform one kind of attack.

6. We will only consider static targets, as there is no availability for a second actor to perform the role of a moving target. The static target will be considered a reasonable approximation of a slowly moving enemy. For the same reason, we limited ourselves to tracking the target only on the horizontal plane (in our frame of reference, the x and z axes).

7. We will consider situations where one must switch from one target to another. This is a common situation in video games and it is important to be able to reproduce it correctly.

For recording, we used the Utrecht University motion capture laboratory, equipped with Vicon Blade motion capture cameras and suits, with a single

actor performing the motions. Before every take, the actor was given a predetermined sequence of attacks to perform, towards a number of different static targets that we placed in our recording area.

The sword was represented as a wooden prop. The Vicon system is able to track rigid bodies by tracking a number of markers on the surface of a prop. Initially we placed 5 markers along the whole length of the sword, but experimentation proved that multiple markers could easily get occluded by the actor's body. In the end we moved the markers to a small patch near the endpoint of the sword prop. This patch, being farther away from the actor, had less of a chance to be hidden from the field of view of multiple cameras, and ultimately yielded better tracking for the sword.

The targets were represented by sets of markers placed on chairs and other furniture. The targets were not at the correct height to be realistic, so we instructed the actor to always attack above the target, as if it was as tall as himself, as to simulate an enemy of a similar size.

We classified the types of attacks based on whether they were thrusts or swings and based on the trajectory of the sword. We classified swings in 8 directions, based on whether they were moving horizontally, vertically, or diagonally, taking inspiration by Renaissance fencing rules; additionally, directional attacks are a common feature in games so it makes sense to proceed this way. In total we ended up with 9 types of attacks: one thrust and 8 kinds of swings.

Three targets were placed in the recording room around the actor, in a wide area. We randomly generated a sequence of attacks, randomizing which target was to be attacked and what kind of attack was to be performed. The actor was then instructed to move from one target to another, following the generated sequence. The actor moving from one target to another caused him to have to turn around several times to face each target. In addition, every couple of takes we would slightly move the targets and change the starting orientation of the actor to provide variety.

A randomly generated sequence ensures that we introduce no bias in the data set by manually choosing what the actor should perform, it is also an attempt to maximize the coverage of our problem domain. With enough recordings, random sequences are guaranteed to present a varied enough set of attacks without having to go through all the combinations of direction, target distance, and type of attack.

Every take was to be made of a sequence of attacks, with two to five attacks in a sequence. Two to five allows us to have a good balance between

having transitions between attacks and having transitions from and to a neutral stance at the beginning and end of the sequences.

Every attack's target is chosen randomly among the three available. Even if the targets are all within a range of 2.5 meters from the starting position, they are placed in different directions with respect to the actor. This, in addition to the actor having to move and turn to face each target, ensures that a variety of directions and distances of attack are generated. Finally, the type of attack is chosen randomly between a thrust and 8 kinds of swings.

We recorded 36 sequences in total. Each sequence was then manually cleaned using AutoDesk MotionBuilder, and manually tagged to mark key instants in the attack. We marked the beginning, end, and apex point of each attack, with the apex being defined as the moment where the endpoint of the sword is closest to the target. Each attack in the sequence was also tagged to specify which target was being attacked.

The Vicon system records at 120 Hz, but ultimately all the sequences were down-sampled to 60 Hz, to be closer to the frequency of a game loop, since on a game running at 60 fps or less most frames in a 120 Hz animation would be discarded. This also allows the ERD network to learn more efficiently as there is more information encoded in the time axis over the same amount of frames. Finally, a constant sample frequency allows our models to be trained using a constant time step, which means that the network does not have to learn to change its output based on the time interval.

4.2 Data processing

Before training, our data has to be converted in a format suitable for machine learning. From each pair of adjacent poses at instants t and $t+1$ in a sequence, we extract an input vector x_t and an output vector y_t. These vectors are then stacked vertically into matrices X and Y, where each row is a frame and each column is a feature of the vectors. The matrices for all the sequences are then collated together.

The input vector x_t is made of the joint vector j_t for the current frame and an additional control vector c_t. The output vector y_t is made of the joint variables for the next frame j_{t+1}. Additionally y_t has some variables that help with phase and trajectory prediction during the run-time phase (see 4.5).

When forming these vectors, some of the features have multiple dimensions. Positions are represented by a 3-dimensional vector, while rotations

are represented as unit quaternions.

Firstly, we have to make sure that all quaternions are on the same side of the quaternion hypersphere. For every quaternion in fact both the quaternion and its additive inverse represent the same rotation. In situations such as interpolation and machine learning, more stable results are yielded when similar rotations are always represented by similar values. We pick the half-sphere where the real part of the quaternion is positive, and negate any quaternion that has a negative real part.

To make the representation more compact, we consider the logarithm of the quaternion and discard the scalar part, which for the logarithm of a unit quaternion is always zero. Both positions and rotations are therefore represented by three-dimensional vectors, which are flattened and inserted into the vector element by element. The networks then learn from each element independently.

In the following sections we describe the features we settled on for the joint vector and for the control vector. These features came about after a process of experimentation, adding and removing tentative features until an optimal configuration was achieved. In the result section we document what happens when some features are added or removed from the model.

4.2.1 Joint Variables

The joint variables, position and rotation, are by convention represented in a local frame of reference, i. e. for each joint in reference to the parent joint in the hierarchy. The root joint (the pelvis) is then represented in an absolute frame of reference with respect to the room where the motion was recorded. This is a problem for two reasons. Firstly, similar motions recorded with the actor in different locations should be represented similarly. Secondly, the translations and rotations of every joint in relation to their parent are not very significant for training, since any specific value of a rotation can have very different meanings depending on the positions and rotations of the joints higher up in the hierarchy.

Our solution, which is also the one used by Holden et al. [2017], is projecting the pelvis joint onto the floor, creating a new root joint, and representing everything using this root joint as a frame of reference. Being a projection onto a plane, this root joint has only 3 degrees of freedom, 2 of translation along the horizontal axes x and z, and 1 of rotation about the y axis.

The root joint's orientation should represent the forward direction of the

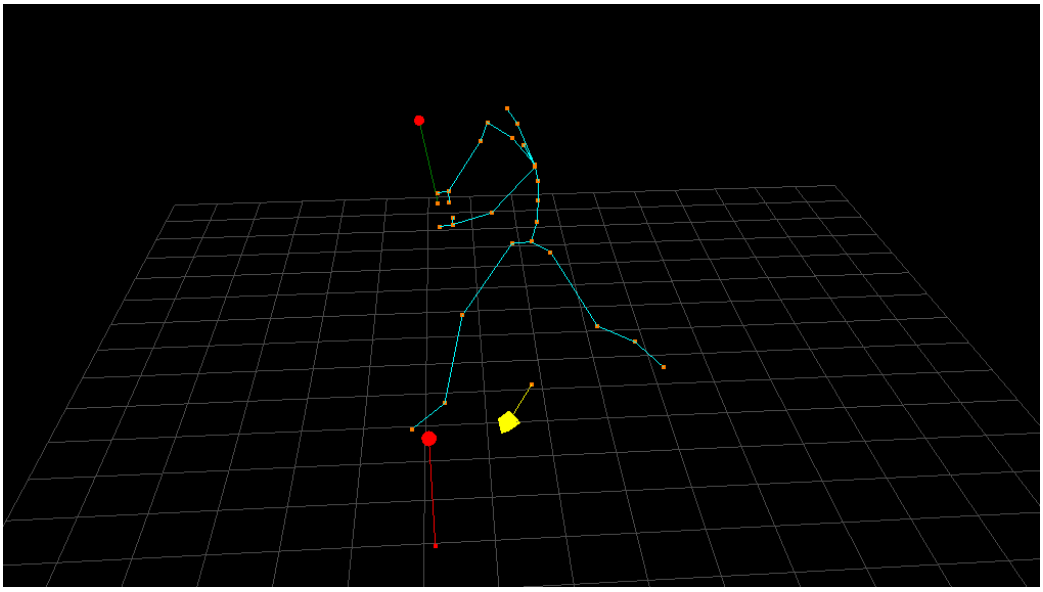

Figure 11: An example of the projection of the pelvis onto the floor, with a yellow arrow indicating the orientation.

character. This is calculated by taking as a forward direction the cross product between the up vector and the vector between the left and right hip joint. This forward direction is smoothed with a simple Gaussian filter and then converted to a rotation angle about the y axis.

Since everything is represented in relation to the root joint, in this frame of reference the joint variables of the root joint are zero at every frame, so they don't encode any information; a better choice is to measure the linear (axes x and z) and rotational (axis y) velocity at every frame of motion. The absolute position and orientation will then be integrated during run-time (see 4.5).

In addition to the positions and rotations, as an additional feature we also gathered the linear velocity of the joints, with respect to the root joint. This is helpful for the PFNN model, which does not have any recurrent memory cells and thus is less capable of learning time-based effects.

To conclude, the joint vector is formed by the following features.

1. The current linear velocity of the root joint, along the X and Z axes
2. The current rotational velocity of the root joint, about the Y axis

3. The current positions of the other joints, with respect to the root joint
4. The current orientations of the other joints, with respect to the root joint
5. The current velocities of the other joints, with respect to the root joint

4.2.2 Control Variables

Phase As an additional variable for the input vector, we considered a phase value ϕ representing the current state of the attack. The phase value is a value introduced by Holden et al. [2017] as a recurrent element to their model, where the phase represents the current state of the walking cycle. We recognized that repeated attacks have similar characteristics, and decided to add a phase value to our model as well. While it is strictly necessary to train PFNNs, it is also helpful as a feature in ERD, as documented by Holden et al.

The phase value is a partially arbitrary value that represents the current moment in the motion of an attack. We have observed that much like walking in [Holden et al., 2017], multiple attacks in sequence are a form of cyclical motion, where the attack alternates between a swing phase, where the weapon is swung against the opponent, and a back-swing phase, where the weapon just follows its momentum and slows down. These are divided by the apex points we marked while gathering our data.

We define the phase value $\phi \in [0, 1)$, where ϕ goes linearly from 0 to 0.5 between the beginning and the apex of the attack, and from 0.5 to 1 from the apex to the end of the attack. When the next attack begins, the phase value loops back to zero. Having a phase value also requires a way to update the phase value for the next frame during run-time, so we add the difference between the next and the current frame's phase is added to the output vector.

Trajectory As a second control variable, we added the current trajectory of the root joint, represented as an array of positions and orientations in space, represented with respect to the root joint at the current frame. The root joint is sampled 6 time steps into the future and 6 time steps into the past, at intervals of 10 frames.

During run-time, we can easily track the past trajectory of the root joint, but we have to find a way to predict the future trajectory. To help trajectory

prediction to adhere more closely to the data set, we also include the future trajectory, computed for time $t + 1$, to the output vector. This will be used during the run-time phase as a starting point to predict the future trajectory (see section 4.5).

Target position We want to add the position of the current target, with respect to the root joint, as a feature to our control vector. While gathering data, we recorded a number of target points. Each attack in the sequence has a target point specified in the labelling data.

Intuitively, at each frame we would take the target point associated with the current attack. Through experimentation, we noticed that anticipating the target switch to the back-swing phase helps the model to react faster to target changes. We might compare this to what a person does while "thinking ahead" while performing a sequence of tasks.

Therefore, to improve the ability of the models to learn to change targets quickly, which target we are considering as the current target is taken from the current attack in the sequence, when $\phi \leq 0.5$, and from the next attack in the sequence, when $\phi > 0.5$.

4.2.3 Summary

To summarize, the input vector x_t is made up by the joint vector at time t, the trajectory points and orientations at time, the target point, all computed at time t. The output vector y_t is made up by the joint vector at time $t + 1$, the future trajectory points and orientations at time $t + 1$, and the difference between the phase value at time $t + 1$ and at time t.

4.3 Network configuration

In this section we will describe the configuration of our ERD model and of our PFNN model. General descriptions of these models have already been given in section 3, so here we will describe the details of our implementation of these models. The models were implemented using Python and the Tensorflow framework, and were configured to train on a GPU.

PFNN A Phase-Functioned Neural network (see figure 12) is a variation of a standard feed-forward neural networks where the network weights vary based on a function of the phase value ϕ. The function used for each

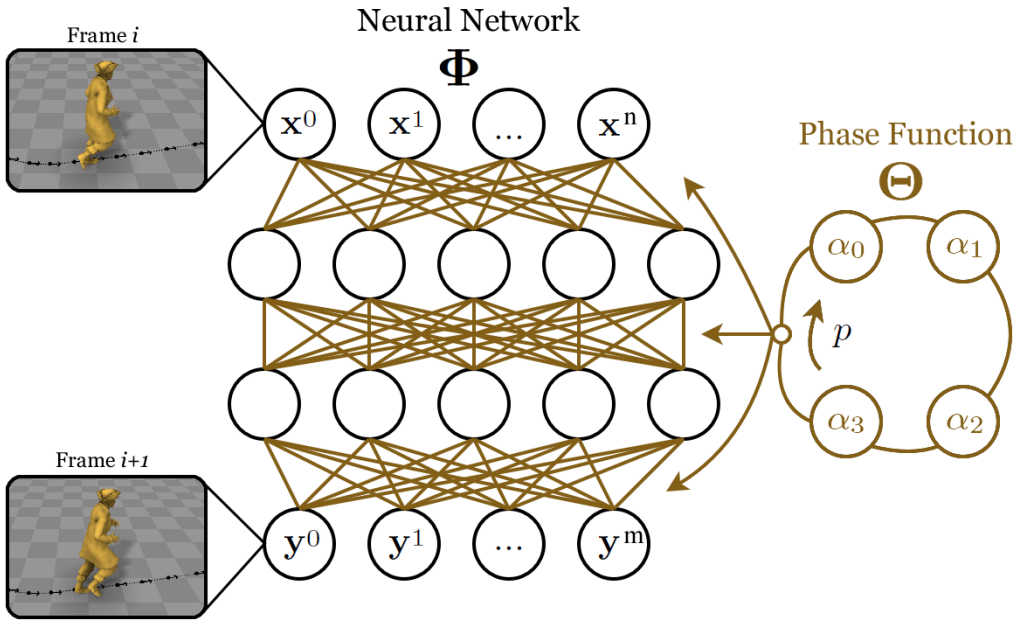

Figure 12: The network structure in PFNN. [Holden et al., 2017]

weights is a cubic spline with 4 points. Training for different values of ϕ allows the network weights to be trained for all of the 4 points over the spline. ϕ is effectively the single recurrent element in the network. In this thesis, PFNN was configured exactly as in [Holden et al., 2017], having three layers of 512 units.

The main advantage of PFNN is being able to imitate some features of recurrent networks while remaining simple. It is highly specialized for cyclic motions where the phase value is highly significant.

ERD The Encoder-Recurrent-Decoder [Fragkiadaki et al., 2015] model consists of three neural networks layered together (see figure 13), the input of one going into the other, where the one in the middle is a LSTM-based (see 3.1.2) recurrent neural network, and the other two are normal feed-forward neural networks, called the Encoder and Decoder. We settled on encoder and decoder networks with two layers of 512 single-neuron units, with a recurrent network composed of two layers of 256 LSTM cells. This was determined by using 512 units as a starting point (a common layer size for this kind of application [Holden et al., 2017]),

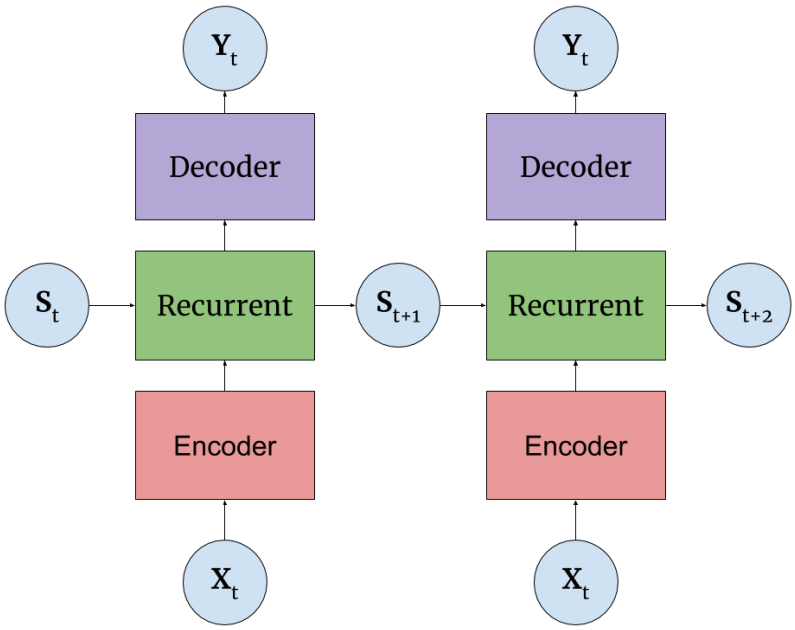

Figure 13: A representation of the Encoder-Recurrent-Decoder model.

and dividing by two for the recurrent part. We started with a single layer per step, but increased it to two as the initial network wasn't able to learn any complex motion.

The objective of the ERD model is to present the recurrent part of the network with a lower-dimensionality, encoded version of the input vector, allowing the recurrent layer to be smaller and faster to train than an equivalent standard RNN. After the recurrent step, its output is then again converted into a useful output vector through a decoder network. Of course, this process results in a partial loss of information, but this is a common practice when dealing with high-dimensional data [Fragkiadaki et al., 2015; Holden et al., 2016, 2015]. In the end, the feature space that the recurrent layer deals with ends up being an efficient representation of the network input.

The three layers are all trained together as if they were a single network, with the error gradient being back-propagated through all the layers. With the middle network being recurrent, the combined network is recurrent as well. A recurrent network has units that connect back to themselves. We unroll the loops in the time dimension, making each

of the recurrent units connected with a previous version of itself. In addition to an input and an output, a RNN has therefore an additional state vector s_t to keep track of, which represents the current value of the recurrent connections. A RNN takes a current state s_t as an additional input and a new state s_{t+1} as an additional output. The state variables have no predetermined meaning, as they only acquire it during the training process.

The main advantage of RNN is that they are able to carry information from one iteration to the next, effectively giving them a form of memory, and making them very suited for learning in time-based or sequence-based applications. ERD makes the learning process more efficient by limiting the size of the recurrent part of the network through the encoding-decoding process.

For both PFNN and ERD, all of the layers are provided with dropout, i. e. randomly setting neuron values to zero, a common measure to prevent overfitting [Srivastava et al., 2014]. During training, the unreliability of neuron paths forces the network to learn the same information through multiple paths in the network. During execution, dropout is disabled as it generates noise in the output. We settled on a dropout rate of 0.7 [Holden et al., 2017].

4.4 Training

For each frame t in a sequence, we gather the input and output vectors x_t and y_t and stack them into separate X and Y matrices, where rows are frames and columns are features of the vector.

For each feature, we calculate its standard deviation and mean over all frames and normalize it accordingly. Normalizing our data helps the network learn by preventing it from developing a bias towards a single feature. The phase value is excluded from normalization, as it is important that it stays between 0 and 1 for the PFNN phase function to give correct results.

The X and Y matrices are fed to the neural networks for training. The training is done using a mini-batch gradient descent method, using as a cost function the mean square error of the network output compared to ground truth. This cost function is corrected by the L_2 norm of the network weights, multiplied by a factor γ, to prevent excessively large weights, which is set to 0.01 as not to affect the training quality too much.

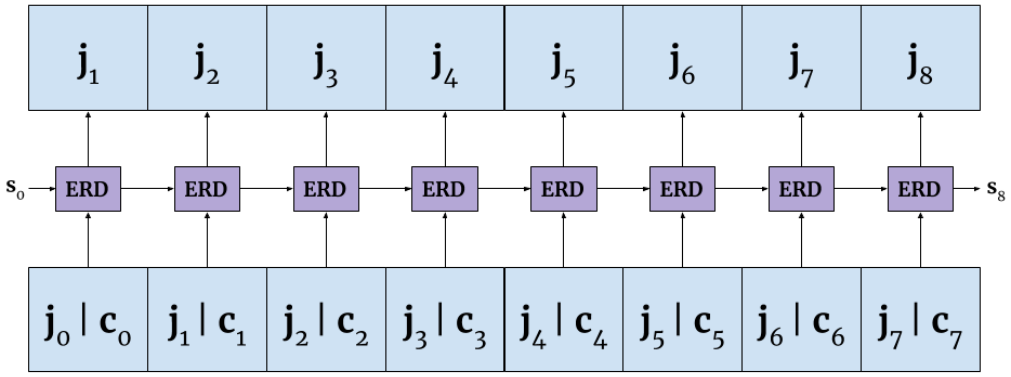

Figure 14: Unrolling the ERD model over a sequence for training.

PFNN is trained frame-by-frame on single data points consisting of matching columns of the X and Y matrices. The mini-batches are made of 32 frames [Holden et al., 2017], selected randomly across the whole data set.

ERD has to be trained over a whole sequence at the same time, as the error gradient has to be back-propagated through the recurrent connection, going back in time to the first frame of the sequence. The X and Y matrices are therefore divided by sequence, and the network is trained one sequence at a time. The network is unfolded over the whole sequence (see figure 14), being applied to each frame at the same time, and the aggregate error of all the outputs is considered.

We have to determine the starting state s_0 for the recurrent part of the network. In common RNN applications, this is not a very important issue as what matters is the output of the final iteration of the network. The starting state is usually set at zero and a few iterations at the beginning will have incorrect output while the memory develops.

In our case, we care about the output of the network throughout the whole sequence, and we want to minimize any output artifacts on the first few iterations. To do this, we turn s_0 into a trainable variable. This means that, just like with the weights of the network, the gradient of the error with respect to s_0 is calculated, and the value is changed

accordingly during training.

Since we have to work with full sequences instead of frames, the batches are also made of sequences, with a batch size of two sequences, which was the maximum batch size that our GPU could process in a single step without performance degradation.

4.5 Run-time

The models are executed in an auto-regressive configuration, i. e. every execution of the model yields the new joint vector for the next frame, which is then used as part of the new input vector. For ERD, the internal network state is also kept track of, with the output state being given as input for the next execution of the model.

The root joint linear and rotational velocity are used to move the root joint of the character, using a simple Euler integrator. Similarly, the delta phase from the output vector is used to increase the phase value, taking only the decimal part in case it is greater than 1.

As for the trajectory inputs, it is straightforward to keep a past buffer of positions and orientations to pass to the input vector as samples of the past trajectory. There is no way however to precisely know the future trajectory of the character.

To estimate the future trajectory, we take the predicted trajectory output from the y vector as a starting point. This trajectory represents a realistic future trajectory for our character. The trajectory is blended with a *goal trajectory* to influence the movement of the character and help it move and aim towards the target.

The goal trajectory consists of simply going towards the target in a straight line at a target velocity, up to a distance of 75 cm. This trajectory is then blended together with the predicted trajectory, using a blending function

$$TrajectoryBlend(a_0, a_1, t, \tau) = (1 - t^\tau)a_0 + t^\tau a_1$$

where a_0 and a_1 are the starting and ending values of the blend.

This blending function is the same as used by Holden et al. [2017]. Its main propriety is that when t is zero the function yields a_0 and when t is one it yields a_0. τ is a bias variable that controls the responsiveness of the curve. t is set so that it goes from 0 to 1 as we proceed into the future.

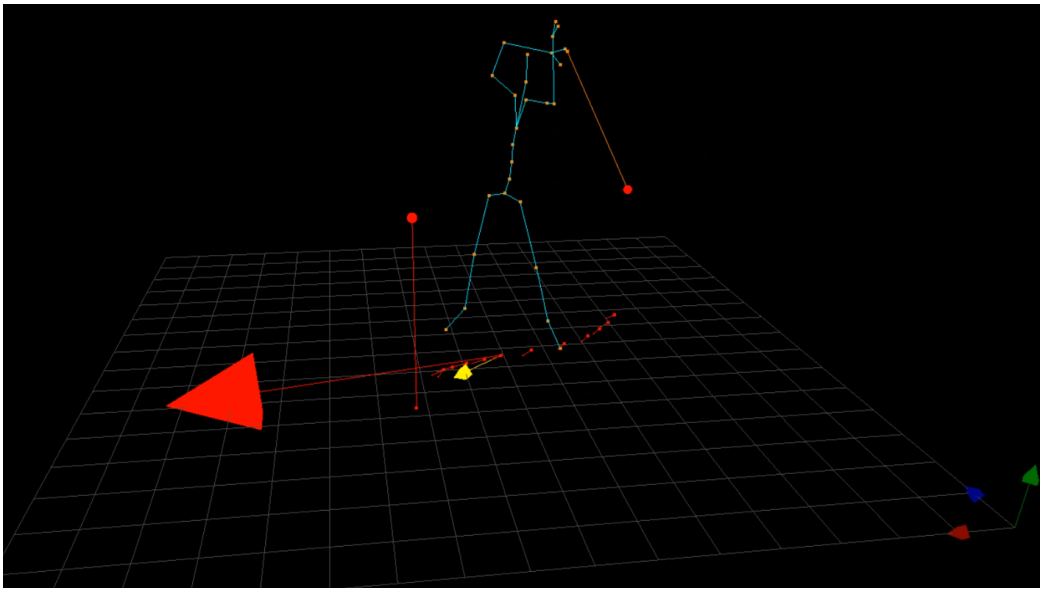

Figure 15: Our demo executing our trained models in real time.

While directions are blended directly, positions are blended by calculating the velocity at each step, blending the velocities, and then doing a quick Euler integration to generate a new trajectory. This generates a more realistic curve.

The responsiveness factor τ is 0.5 for velocities and 2 for directions, as a quick change in facing will feel more natural than a quick change in velocity. The same values are used by Holden et al., as quick changes in directions give the perception of a quick response, while slow changes in velocity make the character feel more physically grounded.

5 Results

To demonstrate the results of our methods, we have built a demonstrative application where the character's skeleton is rendered and is allowed to move around and respond to changes in the control variables (see figure 15). The user can move the target point on the horizontal plane and observe the resulting animation.

We recorded a series of videos, with various configurations of the input vector and of the dropout rate, at different stages of the training process.

These videos can be viewed on the thesis website.

As a correctness check, we have also used our implementation of PFNN and ERD to replicate Holden et al. [2017]'s results, using their data set of locomotion animations. For the sake of simplicity, we only considered the flat ground data, as our project does not deal with uneven terrain.

The results with the locomotion data set was successful and in line with Holden et al.'s findings. ERD was able to replicate character movement in any direction with good responsiveness, with limited foot sliding artifacts. PFNN gave better, more reactive results with less foot sliding.

5.1 PFNN

The first clear result from our experiments is that PFNN proved to be inadequate for sword motion. While it reacted promptly to user command and changes in target, and the motion of the root joint was correct, the actual motion of the sword and the character limbs resulted in unnatural arcs, significant noise, and motion that barely looked like the motion of a sword.

This in our opinion is to be attributed to the highly cyclical nature of PFNN and the fact that it is an optimised version of an RNN designed specifically for walking motion. While sword swinging is also a cyclical motion, it involves a physical object swinging around with momentum and complex interaction with the arms. It is possible that a single recurrent variable is not enough to model these kinds of animations.

We also observed that PFNN seems to be unable to discern between the various attack types, yielding an average of all of them at the same time, which is what makes the attack arcs look significantly different than those in the data set, and all similar to each other, with little variation between one attack and the next.

5.2 ERD

ERD gave us some better results. Firstly, the model was able to generate motion similar, but not identical, to the motions from the training set. The character was able to track and follow a target and to change directions when the target moved.

Naturalness We measure the naturalness (see subsection 3.4) of motion based on the physical realism, i. e. whether the generated motion has similar

dynamic properties to a human, on the variety, i. e. whether it produces different kinds of motion with similar input, and on whether it produces new motion rather than just learning to replicate the motion from the training set.

ERD is indeed able to generate new motion, instead of just copying the motion from the data set. We notice in several cases that attacks start looking like one attack from the data set and end looking like another attack. The character shifts its stance and type of attack based on the relative target position. The new motions are also varied, as it is very difficult to get the character to repeat something exactly.

Physical realism is roughly equivalent to that of the data set, but we can still see sliding artifacts with the feet movements, which confirm that RNN-based models have a problem with preserving foot contacts on the ground, which was documented by Holden et al. [2017] when they applied of ERD to walking motion.

Control We measure control (see subsection 3.4) based on how well and how quickly the character responds to changes in our control vector, which represents user input.

However, even with ERD, the quality of motion generated from the sword-fighting animation data set was not as good as that obtained from the walking animation data set. In particular, while the generated motions were convincing, we were not able to achieve the same level of responsiveness as with walking animations.

While in the walking demo the character promptly responds to user command, in the sword-fighting demo the character takes an inconsistent amount of time to respond and turn towards the target; at times it is immediate, but at other times it does one or two empty attacks, swinging in other directions, before turning on the target. In all cases, it eventually ends up attacking the target, but not always in the most direct way.

These issues are partially reduced when placing the target in the close vicinity of the character. However, even inside our problem domain, which considers targets within a distance of 2.5 meters from the characters, there are some areas where the responsiveness is not optimal.

As a measure of responsiveness, and to ensure the accurateness of our phase value, we measured the phase value at the points of minimum distance (apex) of the sword endpoint to the target. We limited this measurement to

the controlled situation of having a target closer than 2 meters. Ideally, we would expect the phase value to be as close to 0.5 as possible.

	Mean	Variance
Phase value at the apex	0.461	0.143
Apex distance (cm)	34.3	14.5

These results tell us that the phase variable correctly represents the attack cycle and that on average the character tends to bring the sword close to the target. In the end, this is an acceptable result, and we can say that we have successfully trained the model to attack targets.

Training It is also to be noted that when the training time is increase to over an hour (on an NVidia GTX 1070M) the results start worsening instead of improving. Instead of generating new motion, the character locks into a few sequences that highly resemble the ones from the data set. Since this is a clear case of overfitting, increasing regularization parameters should improve things, but increasing the dropout rate beyond 0.8 simply results in the model not being able to learn anything and getting locked into a static pose.

Discussion The reason for this difference in results between the ERD model trained on the walking data set and on the sword-fighting data set could be attributed to two factors.

1. The problem of animating a character swinging a sword is more complex than that of a character moving.

2. Our sword-fighting data set might not have been as good as the walking data set, in quality or size, or both.

As for the first one, there is mild evidence to indicate it. It is true that PFNN, a simpler model than ERD, completely failed to learn the concept of a sword swing; however that could also just be due to the fact that PFNN is a highly specialized model and this does not necessarily transfer to ERD.

As for the second one, the walking data set was indeed much larger than what we were able to record on our own for the sword-fighting data set. There is indeed the possibility that this is simply a question of the necessary size of the data set.

The fact that there is a problem with the data is also corroborated by the training problems we encountered when training for a prolonged amount of time. It is possible that the data did not have enough different examples for the model to be able to generalize beyond a certain point.

The other possibility is that our method of randomly choosing the target imparted an unforeseen bias upon the data set, which prevented the model from properly learning how to turn towards the target. While this is possible we are not convinced that this is the case, as PFNN was perfectly able to learn how to properly face and move towards the target.

5.3 Input vector experimentation

We experimented with a variety features for the input vector x before settling on the ones presented in subsection 4.2. We evaluated the configurations based on the same naturalness and control principles as presented earlier.

Initially we tested a minimal setup with only the phase value, the joint positions and the trajectory positions, without any rotation or velocity data. This setup showed a clear difficulty in facing the target position, which could be explained by the fact that the only input data correlated to the target were the future trajectory points, and this did not include any rotation data.

The first addition was adding the target position (with respect to the root joint) to the input vector. We noticed that this helped the character learn how to turn towards the target, but still unreliably. The character would sometimes perform an attack in the opposite direction with respect to the target, even when the target was very close.

Adding the directions at the trajectory points finally allowed the model to learn how to turn itself reliably. This is the point where we reached the optimal configuration for ERD, which are the results described in the previous section.

The last modification was adding joint rotations and velocities to the joint vector. This did not cause any particular change in the ERD results, but PFNN improved noticeably by significantly reducing the movement noise. We can attribute this to the fact that PFNN is unable to learn time-based effects, so including the velocity as a feature helps it learn the concept of momentum and acceleration. On the contrary, ERD is able to learn the concept itself, which explains why the changes were minimal.

5.4 Run-time performance

We tested the performance of both ERD and PFNN running on an Intel Core i7-8750H. These numbers are worse than those presented by Holden et al. [2017] by a factor of two, but it is to be expected since our models were running on unoptimized Python code while Holden et al. reimplemented their model in fast C++ code for run-time.

	Frame time
ERD	2.3 ms
PFNN	3.4 ms

It is still well within the limits of a 60 fps frame cycle, and faster results can be easily obtained by translating the run-time logic to a language such as C++ and optimizing it.

5.5 Limitations and future work

The data gathering aspect of this project could be improved significantly. Future work might try using same methods with a larger data set, recorded with a more thorough approach to covering the problem domain, and document the changes in the naturalness and control indicators.

While recording we had to settle for a few approximations that future research might be able to avoid. Firstly, we represented the targets as static, as we did not have a second actor to play the target. Future work might try to track a moving target so as to adapt the animation to the motion of the adversary. A more complete data set would also include animation for the opponent being hit, and model that side of the exchange as well.

Secondly, the method we used for randomizing the target did not take into account the necessity of recording attacks from a variety of directions. While attacks from different directions happened, there was no guarantee that they happened with an even distribution. It is possible that we introduced an unneeded bias into the data set, and that another data gathering strategy might have yielded better results.

Another possible improvement is to introduce environmental obstacles into the data set, like Holden et al. [2017] do for character locomotion. This could be further improved by introducing dynamic variables such as external forces into the input vector, allowing the character to physically respond

to contact with the target. Clavet [2016b] did something similar in their Motion Matching experiments, but they decided not to ship that feature into the game as it was still in development. Additionally, the amount of data necessary for such a project would increase dramatically.

ERD-based methods still show numerous feet sliding artifacts. A possible improvement upon these methods would be to devise an IK-based solution to adjust feet positions on the fly and prevent them from sliding. A similar method has been documented partially by Holden et al., but they use IK to adjust the feet movements to uneven terrains rather than to prevent foot sliding. Clavet also presents an IK-based solution to adjust feet movements.

Finally, during the training we did not consider the possibility of manually weighting the input variables in order to bias the model towards certain outcomes. In particular, assigning a larger weight to the control vector might increase responsiveness. It is possible that this strategy yields better results, however it is not a well documented technique.

6 Conclusions

In this thesis we have presented the application of two neural network models, Encoder-Recurrent-Decoder and Phase-Functioned Neural Networks, to the problem of synthesizing motion in real time for a character swinging a sword.

In general we can say that we have been successful in developing a model able to generate sword animation in real time. We can not say that the results are at a point where they can be reliably used by game studios, but it is a step towards being able to generate generic motion online with a data-driven approach.

As for the research questions, the first asked whether the training times were practical for a development studio. The results shown after one hour of training definitely seem to indicate that, with a better data set, we would be in the order of magnitude of the hours. Further proof from Holden et al. [2017] confirms that ERD models take about 9 hours to fully train. As we have determined a practical training time to be a day or less, we can definitely say that this kind of model is within that limit.

The second research question asked whether the method could be executed in real time in an engine loop. With a large margin we can say to have proven that run-time performance is very high both for PFNN and ERD, with execution times much smaller than that of a typical update rate (30-

60Hz). Additionally, our run-time code was unoptimized Python code, in case of an optimized C++ version the performance footprint could be even smaller. Of course, our measurement are only for a single character, but we expect a system like this to be used only for important characters that are very visible on screen.

The third question asked whether the method is able to react quickly and predictably to user input. PFNN was able to react well and move according to the control variables. As for ERD, it showed an inconsistency to its response that was only partially placated by placing the target in a more predictable area in the vicinity of the character.

The fourth question asked whether the method was able to produce natural-looking motion. We can definitely say that PFNN is not suited to this domain of application due to its failure in reproducing motions similar to those in the data set. As for ERD, it was able to generate new motions similar to those in the data set with a few artifacts, such as feet sliding.

The objective of the thesis was to find out whether it was feasible to use neural network based techniques in this domain of animation. While the answer to this question is not conclusive, we can say that the results are promising in this area of research.

References

Arikan, Okan and David A Forsyth (2002). "Interactive motion generation from examples". In: *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. ACM, pp. 483–490.

Arikan, Okan, David A Forsyth, and James F O'Brien (2003). "Motion synthesis from annotations". In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM, pp. 402–408.

Chai, Jinxiang and Jessica K Hodgins (2005). "Performance animation from low-dimensional control signals". In: *ACM Transactions on Graphics (TOG)*. Vol. 24. 3. ACM, pp. 686–696.

Clavet, Simon (2016a). *Motion Matching and the Road to Next-Gen Animation*. Ubisoft. URL: `http://www.gdcvault.com/play/1022985/Motion-Matching-and-The-Road`.

— (2016b). *Motion Matching for Realistic Animation in "For Honor"*. Ubisoft. URL: `https://www.youtube.com/watch?v=4pdcA3mhe0E`.

Fan, Bo et al. (2015). "Photo-real talking head with deep bidirectional LSTM". In: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, pp. 4884–4888.

Fragkiadaki, Katerina et al. (2015). "Recurrent network models for human dynamics". In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4346–4354.

Geijtenbeek, Thomas, Michiel van de Panne, and A Frank van der Stappen (2013). "Flexible muscle-based locomotion for bipedal creatures". In: *ACM Transactions on Graphics (TOG)* 32.6, p. 206.

Glardon, Pascal, Ronan Boulic, and Daniel Thalmann (2004). "A coherent locomotion engine extrapolating beyond experimental data". In: *Proceedings of CASA* 4, pp. 73–84.

Gleicher, Michael et al. (2003). "Snap-together Motion: Assembling Run-time Animations". In: *ACM Trans. Graph.* 22.3, pp. 702–702. ISSN: 0730-0301. DOI: `10.1145/882262.882333`. URL: `http://doi.acm.org/10.1145/882262.882333`.

Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton (2013). "Speech recognition with deep recurrent neural networks". In: *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, pp. 6645–6649.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Holden, Daniel, Taku Komura, and Jun Saito (2017). "Phase-functioned neural networks for character control". In: *ACM Transactions on Graphics (TOG)* 36.4, p. 42.

Holden, Daniel, Jun Saito, and Taku Komura (2016). "A deep learning framework for character motion synthesis and editing". In: *ACM Transactions on Graphics (TOG)* 35.4, p. 138.

Holden, Daniel et al. (2015). "Learning motion manifolds with convolutional autoencoders". In: *SIGGRAPH Asia 2015 Technical Briefs*. ACM, p. 18.

Kovar, Lucas, Michael Gleicher, and Frédéric Pighin (2002). "Motion graphs". In: *ACM transactions on graphics (TOG)*. Vol. 21. 3. ACM, pp. 473–482.

Lee, Jehee et al. (2002). "Interactive control of avatars animated with human motion data". In: *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. ACM, pp. 491–500.

Lee, Yongjoon, Seong Jae Lee, and Zoran Popović (2009). "Compact character controllers". In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 5. ACM, p. 169.

Lee, Yongjoon et al. (2010). "Motion fields for interactive character locomotion". In: *ACM Transactions on Graphics (TOG)*. Vol. 29. 6. ACM, p. 138.

Li, Yan, Tianshu Wang, and Heung-Yeung Shum (2002). "Motion texture: a two-level statistical model for character motion synthesis". In: *ACM Transactions on Graphics (ToG)*. Vol. 21. 3. ACM, pp. 465–472.

Pejsa, Tomislav and Igor S Pandzic (2010). "State of the Art in Example-Based Motion Synthesis for Virtual Characters in Interactive Applications". In: *Computer Graphics Forum*. Vol. 29. 1. Wiley Online Library, pp. 202–226.

Rasmussen, Carl Edward and Christopher KI Williams (2006). *Gaussian processes for machine learning*. Vol. 1. MIT press Cambridge.

Schmidhuber, Jürgen (2014). "Deep Learning in Neural Networks: An Overview". In: *CoRR* abs/1404.7828. arXiv: `1404.7828`. URL: `http://arxiv.org/abs/1404.7828`.

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15, pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

Taylor, Graham W and Geoffrey E Hinton (2009). "Factored conditional restricted Boltzmann machines for modeling motion style". In: *Proceedings*

of the 26th annual international conference on machine learning. ACM, pp. 1025–1032.

Taylor, Sarah et al. (2017). "A deep learning approach for generalized speech animation". In: *ACM Transactions on Graphics (TOG)* 36.4, p. 93.

Troje, Nikolaus F. (2002). "Decomposing biological motion: A framework for analysis and synthesis of human gait patterns". In: *Journal of Vision* 2.5, p. 2. DOI: 10.1167/2.5.2. eprint: /data/journals/jov/933498/jov-2-5-2.pdf. URL: +%20http://dx.doi.org/10.1167/2.5.2.

UFC on FOX (2014). *EA Sports, UFC make most realistic fight game ever*. UFC. URL: https://youtu.be/0kW4U88Vc4c.

Urtasun, Raquel et al. (2004). "Style-based motion synthesis". In: *Computer Graphics Forum*. Vol. 23. 4. Wiley Online Library, pp. 799–812.

Van Welbergen, Herwin et al. (2010). "Real Time Animation of Virtual Humans: A Trade-off Between Naturalness and Control". In: *Computer Graphics Forum*. Vol. 29. 8. Wiley Online Library, pp. 2530–2554.

Wang, Jack M, David J Fleet, and Aaron Hertzmann (2008). "Gaussian process dynamical models for human motion". In: *IEEE transactions on pattern analysis and machine intelligence* 30.2, pp. 283–298.

Xia, Shihong et al. (2015). "Realtime style transfer for unlabeled heterogeneous human motion". In: *ACM Transactions on Graphics (TOG)* 34.4, p. 119.

Zadziuk, Kristjan (2016). *GDC 2016 - Motion Matching, The Future of Games Animation... Today*. Ubisoft. URL: https://www.youtube.com/watch?v=KSTn3ePDt50.

Zhou, Liuyang et al. (2014). "Human motion variation synthesis with multivariate Gaussian processes". In: *Computer Animation and Virtual Worlds* 25.3-4, pp. 301–309. ISSN: 1546-427X. DOI: 10.1002/cav.1599. URL: http://dx.doi.org/10.1002/cav.1599.