Generating Human Motion Animation Using Echo State Networks and Conceptors

by

Rubin Deliallisi

Bachelor Thesis Proposal in Computer Science

Prof. Dr. Herbert Jaeger
Name and title of the supervisor

Date of Submission: May 12, 2017

Jacobs University — School of Engineering and Science

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Signature Place, Date

Abstract

Human motion animation generation (HMAG) refers to the problem of training a system to independently generate motion patterns similar to already recorded ones. It has been seen in a variety of applications like movies, education, computer simulations and video games. Many approaches to this problem are available including key-framing, physics based methods and space time control. However, many problems persist in the current methods because the required reality or complexity of the human motion may be rather different according to its application.

Echo State Networks (ESNs) are one of the key reservoir computing methods. They are a highly practical approach to Recurrent Neural Networks (RNNs) because of their computational efficiency. On the other hand, conceptors enable RNNs to learn/load multiple patterns in the same reservoir. They act as filters that force the reservoir to produce specific patterns. The patterns can be organized and further combined to produce new ones, for instance through morphing.

This guided research does not provide an alternative to state of the art HMAG methods. It aims to implement an ESN-based and a conceptor-based motion generation algorithm and to compare them on the basis of several carefully picked criteria.

Contents

1 Introduction **1**

2 Statement and Motivation of Research **2**
2.1 Human Motion Animation Generation 2
2.2 ESN 3
2.3 Conceptors 5
2.4 Research Objectives 7

3 Documentation of Methods **7**
3.1 Data Description and Pre-processing 8
3.2 Normalized Root Mean Square Error 9
3.3 ESN Pattern Generation 9
3.3.1 State Update Equation 9
3.3.2 Training the Network 11
3.3.3 Generating Test Data and Morphing Patterns 11
3.4 Conceptor Pattern Generation 13
3.4.1 State Update Equation 13
3.4.2 Training the Network 13
3.4.3 Generating Test Data and Morphing Patterns 15

4 Comparison **16**
4.1 Robustness 16
4.2 Stability 18
4.3 Space and Time Complexity 20
4.4 Motion Quality 21

5 Conclusion and Future Work **22**

A Dataset Summary **24**

1 Introduction

Motion is one of the most important ingredients of computer graphics (CG) movies and computer games. Obtaining realistic motion usually involves key-framing, physically based modeling or motion capture. Creating natural looking motions with key-framing requires lots of effort and expertise. Although physically based modeling can be applied to simple systems successfully, generating realistic motion on a computer is difficult, particularly for human motion [1]. Conceptors and ESNs have already been used in motion generation applications and have produced reliable and stable results [2, 3]. This guided research implements and compares the two approaches to generating human motion animation, conceptors as presented by Herbert Jaeger in [4, 2] and ESNs.

An Echo State Network is a recurrent neural network. RNNs represent a large and varied class of computational models that are designed by analogy with biological brain modules. In an RNN numerous abstract neurons are interconnected by likewise abstracted synaptic connections, which enable activations to propagate through the network [5]. RNNs are characterized by feedback (recurrent) loops in their synaptic connection pathways. They can maintain an ongoing activation even in the absence of input and thus exhibit dynamic memory. RNNs have been shown to be Turing equivalent for common activation functions. This means that they can approximate arbitrary finite state automata and are universal approximators [6].

Several learning algorithms are known that incrementally adapt the synaptic weights of an RNN in order to tune it toward the target system. With ”Deep Learning” techniques, RNN training by gradient descent has become highly practical, though not trivial. An overview of state of the art deep learning algorithms is given in [7]. The ESN approach differs from these methods in that a large RNN is used (order of 50 to 1000 neurons) and only the synaptic connections from the RNN to the output readout neurons are modified by learning. Because there are no cyclic dependencies between the trained readout connections, training an ESN becomes a simple linear regression task [8].

In addition to ESNs, I will use the mechanism of conceptors, by which the dynamics of an RNN can be governed in a variety of ways. Mathematically, conceptors are linear operators which characterize classes of signals/patterns that are being processed in the RNN. Starting from an operating RNN, they can be learned and stored, or quickly generated on-the-fly, using a simple adaptation rule: learning a regularized identity map. Conceptors can be represented as matrices or as neural subnetworks and allow the reservoir to learn, store, abstract, focus, morph, generalize, de-noise and recognize a large number of dynamical patterns within the reservoir [4].

The rest of this document is structured as follows. Section 2 describes the researched problem in more detail by providing an in-depth look into ESNs, conceptors and HMAG as described in other articles. Furthermore it states the research objectives and the motivations behind them. Section 3 documents the methods used to solve the HMAG problem using ESNs and conceptors. It also provides theoretical insights and practical hints to further optimize the used methods. Section 4 provides a comparison of the 2 methods used in this project on the basis of various criteria. Section 5 is the final section where a summary of all the results obtained in this project is given and useful starting points for further investigation are indicated. Appendix A provides details on the dataset used throughout this project.

2 Statement and Motivation of Research

2.1 Human Motion Animation Generation

In this section I provide an overview of the human motion animation generation (HMAG) problem, explain the approaches that have been used in solving this problem and elaborate on the approach that serves our purpose, that is generating human motion animations using motion capture (MoCap) data.

HMAG, also known as motion synthesis, is an important part of media like video games and movies. More lifelike characters make for more immersive environments and more believable special effects. At the same time, realistic animation of human motion is a challenging task, as people have proven to be adept at discerning the subtleties of human movement and identifying inaccuracies. There are three natural stages of motion synthesis [9]:

1. **Obtaining motion demands** involves specifying constraints on the motion, such as the length of the motion, where the body or individual joints should be or what the body needs to be doing at particular times.
2. **Generating motion** involves obtaining a rough motion that satisfies the demands. This is the main objective of this project.
3. **Post processing** involves fixing small scale offensive artifacts. An example would involve fixing the feet so that they do not penetrate or slide on the ground, lengthening or shortening strides and fixing constraint violations [10] .

In this project MoCap data is used to obtain motion demands, ESNs and conceptors are used to generate motion and post processing is not done. A comparison of a few publicly available character engines is given in [1].

Generating motion largely follows two threads: using examples and using controllers. Example based motion generation draws on an analogy with texture synthesis where a new texture (or motion) that looks like an example texture (or motion example) needs to be synthesized [11]. A road-map of all the motion examples can be constructed and searched to obtain a desired motion graph [12, 13]. Motion graphs transform the motion synthesis problem into one of selecting sequences of nodes, or graph walks. The clips in this road-map can also be parameterized for randomly sampling different motion sequences. The resulting Markov chain can be searched using dynamic programming to find a motion that connects two key-frames [14]. The results of a motion graph algorithm can be observed in Figure 2.1.

Controller based approaches use physical models of systems and controllers that produce outputs usually in the form of forces and torques as a function of the state of the body. These controllers can be designed specifically to accomplish particular tasks [15] or they can be learned automatically using statistical tools [16]. Types of probabilistic search algorithms have also been used in physically based animation synthesis [17].

Some of the standard solutions to HMAG involve motion capture: motion data for an approximate skeletal hierarchy of the subject is recorded and then used to drive a reconstruction on the computer. Most motion capture systems are very expensive, for this reason I use the CMU Graphics Lab Motion Capture Database [18]. Raw motion capture

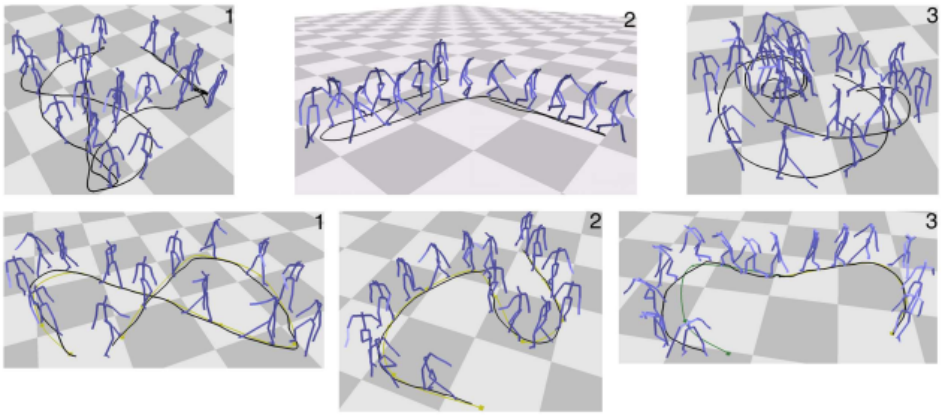

Figure 2.1: Top images show original motion capture data; two are walking motions and one is a sneaking motion. The bottom images show new motion generated by a motion graph built out of these examples as described in [13].

data contain spatiotemporal traces of markers attached to the performing person. By preprocessing this data a bundle of motion signals can be obtained. Each signal represents a sequence of sampled values for each degree of freedom. These signals are sampled at a sequence of discrete time instances with a uniform interval to form a motion clip that consists of a sequence of frames. In each frame, the sampled values from the signals determine the configuration of an articulated figure at that frame, and thus they are related to each other by kinematic constraints [12].

While motion capture is a reliable way of acquiring realistic human motion, by itself it is not a technique for reproducing motion. The recorded data has to first be cleaned and then normalized. Cleaning the data is in itself an intrinsic signal processing task, but the CMU Graphics Lab takes care of providing already clean and de-noised data. Normalization is application dependent and in most of the cases it has to be done manually for every motion pattern. Only after completing these 2 non trivial steps can the data be used to train any motion generating system. The last step is rendering the generated data for which I use the MoCap Toolbox [19]. This toolbox also contains various methods that help in preprocessing raw MoCap data.

2.2 ESN

The following section presents the general structure of an Echo State Network (ESN) as introduced in [20, 21, 22], describing the reservoir update equations and the readout training process. The explanations are focused on gradually deriving a pattern generator ESN.

ESNs provide an architecture and supervised learning principle for Recurrent Neural Networks (RNNs). The main idea is to drive a random, large, fixed RNN with the input signal, thereby inducing in each neuron within the reservoir a nonlinear response signal, and combine a desired output signal by a trainable linear combination of all of these response signals [23].

The central part of an ESN is the reservoir, a randomly generated, recurrent neural network. It can be seen as a nonlinear high-dimensional expansion of the input signal and

at the same time it serves as a memory, providing temporal context. The reservoir, being an input-driven dynamical system, should provide a rich and relevant enough signal space, such that the desired target can be obtained by a linear combination from it [24]. I consider discrete-time neural networks where the reservoir consists of K input units, N internal network units and L output units. The general architecture of a ESN can be seen in Figure 2.2.

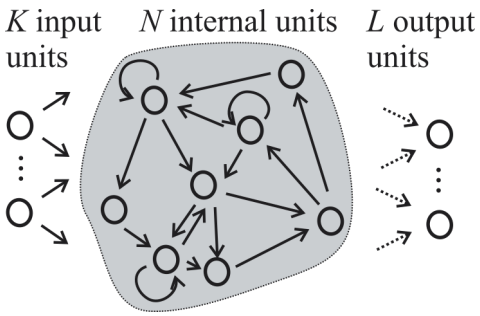

Figure 2.2: Graphical representation of ESNs taken from [20]. Dashed arrows indicate connections that are possible to train.

Each of the units at time step n has an activation (numerical value). Activations of input units at time step n are $\mathbf{u}(n) = (u_1(n), ..., u_K(n))$, of internal units are $\mathbf{x}(n) = (x_1(n), ..., x_N(n))$ and of output units $\mathbf{y}(n) = (y_1(n), ..., y_L(n))$. Real-valued connection weights are collected in matrix $\mathbf{W}^{in} \in \mathbb{R}^{N \times K}$ for the input weights, in matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$ for the internal connections, in matrix $\mathbf{W}^{out} \in \mathbb{R}^{L \times (K+N+L)}$ for the connections to the output units, and in matrix $\mathbf{W}^{back} \in \mathbb{R}^{N \times L}$ for the connections that project back from the output to the internal units [20]. Also connections from input to output and output to output are allowed but are not of particular use in our application. In theory no further constrains exist on the reservoir weights $\mathbf{W}$, $\mathbf{W}^{in}$ and $\mathbf{W}^{out}$. Since these weights are not changed during the training phase, but are precomputed according to the guidelines in [21], the size of the reservoir can easily be increased with little additional computational costs.

In this project I do not use plain ESNs as described in [20]. Instead, ESNs with leaky integrator neurons, introduced in [20, 21] and further examined in [22], are used. The activation of internal units for this particular type of ESN follows the equation:

$$\mathbf{x}(n+1) = (1-\alpha)\,\mathbf{x}(n) + \alpha\,\mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n) + \mathbf{b}) \tag{2.1}$$

where $\mathbf{x}(n)$, $\mathbf{u}(n)$, $\mathbf{y}(n)$ and $\mathbf{W}$, $\mathbf{W}^{in}$, $\mathbf{W}^{back}$ have the above described meaning, $\mathbf{f}$ denotes the component-wise application of the individual unit's transfer function, $\mathbf{b} \in \mathbb{R}^{N \times 1}$ is a bias vector. α is the leaking rate, a time constant which can be used to slow down the network [21]. For the purpose of pattern generation the term $\mathbf{W}^{in}\mathbf{u}(n+1)$ will not be needed. The sigmoid function $\mathbf{f} = \mathbf{tanh}$ is used as the transfer function.

During the training phase, the to be generated pattern is served to the network as a teacher forced output $\mathbf{y}(n) = \mathbf{d}(n)$, where $\mathbf{d}(n)$ is the training signal value at time n. When the ESN is in pattern generating mode, not driven anymore by the teacher forced output, the output is computed according to:

$$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n))) \tag{2.2}$$

where $(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n))$ denotes the concatenated vector made from input, internal, and output activation vectors. $\mathbf{u}(n)$ is not needed since the input is feed to the

network as a teacher forced output, while usage or non usage of $\mathbf{y}(n)$ is optional and in our task we can do without. The output transfer function used is $\mathbf{f}^{out} = \mathbf{id}$ [21].

The learning task is to compute the output weights $\mathbf{W}^{out}$, by minimizing the mean squared error:

$$\mathbf{MSE}_{train} = \frac{1}{T - n_0} \sum_{n=n_0+1}^{T} ||\mathbf{d}(n) - \mathbf{W}^{out}\mathbf{x}(n)||^2 \tag{2.3}$$

where T is the length of the training data and n_0 is the washout time. Minimizing $\mathbf{MSE}$ is a simple linear regression task. Ridge regression (aka Tikhonov regularization [25]) is used instead of textbook linear regression because of its higher numerical stability, ability to reduce sensitivity to noise and potential to reduce overfitting [5]. Computing $\mathbf{W}^{out}$ using ridge regression leads to:

$$(\mathbf{W}^{out})' = (\mathbf{X}\mathbf{X}' + \gamma\mathbf{I})^{-1}\mathbf{X}\mathbf{D}' \tag{2.4}$$

where $\mathbf{I}$ is the identity matrix, γ is the regularization coefficient and $'$ is the matrix transpose operator. $\mathbf{D} \in \mathbb{R}^{L\times(T-n_0)}$ and $\mathbf{X} \in \mathbb{R}^{N\times(T-n_0)}$ are both recorded during the training phase. For each time step n after the washout time n_0 the state of the internal units $\mathbf{x}(n)$ is stored in the $(n - n_0)$-th column of $\mathbf{X}$ and the teacher forced output $\mathbf{d}(n)$ is stored in the $(n - n_0)$-th column of $\mathbf{D}$.

2.3 Conceptors

This section provides a full mathematical introduction of conceptors and describes how they are computed. Conceptors were proposed not long ago by Herbert Jaeger in [4, 2] on which this section relies heavily.

Conceptors are neural filters that characterize dynamical neural activation patterns within a RNN. When a RNN is actively generating, or is passively being driven by different dynamical patterns (say a, b, c, . . .), its neural states populate different regions R_a, R_b, R_c, . . . of neural state space. For these regions, neural filters C_a, C_b, C_c, . . . (the conceptors) can be incrementally learned. A conceptor C_x, representing a pattern x, can then be invoked after learning to constrain the neural dynamics to the state region R_x, and the network will select and re-generate pattern x.

The dynamics of the neural model system (RNN in this case) are mathematically described by the state update equations:

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^*\mathbf{x}(n) + \mathbf{W}^{in}\mathbf{p}(n) + \mathbf{b}) \tag{2.5}$$

$$\mathbf{y}(n) = \mathbf{f}^{out}(\mathbf{W}^{out}\mathbf{x}(n)) \tag{2.6}$$

Time here progresses in unit steps $n = 1, 2, \ldots$. The network consists of N neurons, whose activations $\mathbf{x}_1(n), \ldots, \mathbf{x}_N(n)$ at time n are collected in an N-dimensional state vector $\mathbf{x}(n)$. The neurons are linked by random synaptic connections, whose strengths are collected in a weight matrix $\mathbf{W}^* \in \mathbb{R}^{N\times N}$. $\mathbf{W}^*$ must be scaled small enough to ensure that the RNN has the echo state property with respect to the input signals that are fed to it. An input signal $\mathbf{p}(n)$ is fed to the network through synaptic input connections assembled in the input weight matrix $\mathbf{W}^{in} \in \mathbb{R}^{N\times L}$ where L is the dimension of the pattern. $\mathbf{b} \in \mathbb{R}^{N\times 1}$

is a bias vector. As in the case of ESNs, I use $\mathbf{f} = \mathbf{tanh}$ and $\mathbf{f}^{out} = \mathbf{id}$. Equation 2.6 specifies that an output signal $\mathbf{y}(n)$ can be read from the network activation state $\mathbf{x}(n)$ by means of output weights $\mathbf{W}^{out} \in \mathbb{R}^{L \times N}$. These weights are pre-computed such that the output signal $\mathbf{y}(n)$ just repeats the input signal $\mathbf{p}(n)$. The output signal plays no functional role, but it serves as a convenient observer of the high-dimensional network dynamics.

Storing patterns $\mathbf{p}^1, ..., \mathbf{p}^P$ into the reservoir amounts to re-computing the initial random reservoir weights $\mathbf{W}^*$, giving a new set of network weights $\mathbf{W}$, such that the new reservoir can mimic the impact of drivers $\mathbf{p}^j$ in the absence of them. $\mathbf{W}$ is computed as follows. In a separate run, each pattern is fed into the initial reservoir according to:

$$\mathbf{x}^j(n+1) = \mathbf{f}(\mathbf{W}^*\mathbf{x}^j(n) + \mathbf{W}^{in}\mathbf{p}^j(n) + \mathbf{b}), \quad j = 1, ..., P, \quad n = 0, ..., T \tag{2.7}$$

Then $\mathbf{W}$ is computed to minimize the quadratic loss:

$$\sum_{j=1}^{P} \sum_{n=n_0+1}^{T} ||\mathbf{W}^*\mathbf{x}^j(n) + \mathbf{W}^{in}\mathbf{p}^j(n) - \mathbf{W}\mathbf{x}^j(n)||^2 \tag{2.8}$$

where only network states for times after n_0 are used, in order to allow for washing out the arbitrary initial state $\mathbf{x}^j(0)$ according to the echo state property . This again is a linear regression task and can be efficiently solved using ridge regression. Network updates from states $\mathbf{x}^j(n)$ with either the original input-driven update rule, or with an input-free update rule that employs $\mathbf{W}$ instead of $\mathbf{W}^*$ are similar [2].

$$\mathbf{f}(\mathbf{W}^*\mathbf{x}^j(n) + \mathbf{W}^{in}\mathbf{p}^j(n) + \mathbf{b}) \approx \mathbf{f}(\mathbf{W}\mathbf{x}^j(n) + \mathbf{b}) \tag{2.9}$$

The output weights $\mathbf{W}^{out}$ are the same for all patterns and can be computed by minimizing the following loss function using ridge regression over all patterns:

$$\frac{1}{P}\sum_{j=1}^{P} \frac{1}{T - n_0} \sum_{n=n_0+1}^{T} ||\mathbf{p}^j(n) - \mathbf{W}^{out}\mathbf{x}^j(n)||^2 \tag{2.10}$$

Considering 2.9, the loaded reservoir should be able to generate approximate versions of the original patterns. However, if the network were run just by iterating $\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}\mathbf{x}(n) + \mathbf{b})$, the resulting input-free reservoir dynamics is entirely unpredictable. Here conceptors enter the stage. Each loaded pattern $\mathbf{p}^j$ is associated with conceptor matrix $\mathbf{C}^j \in \mathbb{R}^{N \times N}$ which is inserted into the state update loop via:

$$\mathbf{x}(n+1) = \mathbf{C}^j\ \mathbf{f}(\mathbf{W}\mathbf{x}(n) + \mathbf{b}) \tag{2.11}$$

The matrix $\mathbf{C}^j$ acts as a filter that leaves states $\mathbf{x}^j(n)$ from the state pattern associated with pattern $\mathbf{p}^j$ essentially unchanged, but suppresses state components of states $\mathbf{x}^i(n)$ associated with patterns $\mathbf{p}^i \neq \mathbf{p}^j$. The conceptor matrices are computed by minimizing the following quadratic loss function:

$$L(\mathbf{C}^j) = E[||\mathbf{C}^j\mathbf{x}^j(n) - \mathbf{x}^j(n)||^2] - (\alpha^j)^{-2}||\mathbf{C}^j||^2_{fro} \tag{2.12}$$

where the expectation E is taken over all states $\mathbf{x}^j(n)$ that arise in $\mathbf{p}^j$-driven runs according to Equation 2.7. The first component $E[||\mathbf{C}^j\mathbf{x}^j(n) - \mathbf{x}^j(n)||^2]$ is minimal when $\mathbf{C}^j$ is the

identity matrix. This reflects the objective that the j-th conceptor should leave the states $\mathbf{x}^j$ unchanged. The second component $||\mathbf{C}^j||^2_{fro}$ becomes minimal for $\mathbf{C}^j = 0$. This takes care of the objective that the j-th conceptor should suppress state components which are untypical of states $\mathbf{x}^j(n)$. The parameter α^j is called aperture and it strikes a balance between the two objectives.

Minimizing the loss $L(\mathbf{C}^j)$ leads to the solution:

$$\mathbf{C}^j = \mathbf{R}^j(\mathbf{R}^j + (\alpha^j)^{-2}\mathbf{I})^{-1} \tag{2.13}$$

where $\mathbf{R}^j = E[\mathbf{x}^j(n)\mathbf{x}^j(n)']$ is the $N \times N$ correlation matrix of states $\mathbf{x}^j(n)$ obtained in the state dynamics driven by pattern $\mathbf{p}^j$. $\mathbf{C}^j$ is positive semi-definite with N eigenvectors being the same as the eigenvectors of $\mathbf{R}^j$. These eigenvectors are the same as the principal component vectors obtained from a principal component analysis of state sets $\mathbf{x}^j(n)$.

2.4 Research Objectives

Firstly, I state that this project does not aim to provide a state of the art solution to the HMAG problem, nor does it try to compare the results to the current state of the art algorithms. The main objective of the current research is to train and develop an ESN motion generator, a conceptor motion generator and to compare the two approaches.

One of the goals of this research is to test the potential of ESNs in a complex motion generation task. The other goal is to use conceptors to solve the same problem, already done by Herbert Jaeger in [2] with a different focus, and to compare the two approaches. Taking into account these two goals here are the main objectives of this project:

- Develop a leaky-integrator neuron ESN for each of the patterns downloaded from the CMU Graphics Lab Database and find the optimal parameters for generating smooth and stable patterns.
- Morph the developed ESNs two by two to obtain natural looking transitions between different patterns.
- Compute conceptors for each pattern and use them to generate and transition between motion patterns.
- Compare the two approaches based on features like stability, robustness, computational and storage efficiency and motion similarity to real humans.

3 Documentation of Methods

The following section presents each component of the developed models essential to address the research objectives listed in Section 2.4. It provides theoretical insights and details on the practical implementation for a concrete understanding of the system.

3.1 Data Description and Pre-processing

The first step in tackling the HMAG task was to get familiar and to normalize the MoCap data from CMU Graphics Lab Database. The raw MoCap data comes in heterogeneous formatting, specifically the joint numbering differs across individual MoCap trace files. One has to manually unify the formats and create versions that are inputs for the neural network. This in itself would be a demanding task, but the patterns used in this project have been already preprocessed by Herbert Jaeger in [2]. The resulting patterns are 61-dimensional signals (body pose and joint angles). With 61 dimensions the curse of dimensionality would usually be a problem, but this is not the case with our data. The signals are highly interdependent, so the effective manifold dimension is much lower.

After preprocessing, the signals cannot be directly fed to the reservoir because they are not normalized. Different signals of the same pattern differ in scalings and shifts from the origin. If fed to the network in this form, the signals which have the largest absolute value overexcite the reservoir and drive the neurons towards the edge -1/+1 values. Therefore each signal is normalized according to:

$$S' = 2\,\frac{S\,.-\,\mathbf{min}(S)}{\mathbf{max}(S)-\mathbf{min}(S)} - 1 \tag{3.1}$$

where S is the signal vector, $\mathbf{min}(S)$ is the minimum of S and $\mathbf{max}(S)$ is the maximum of S. The $.-$ sign is the MATLAB notation for the element wise subtraction of a scalar from a vector. Equation 3.1 can be rewritten in the following from:

$$S' = (S + \mathbf{shift}) * \mathbf{scaling} \tag{3.2}$$

where $\mathbf{shift} = -\mathbf{min}(S) - \frac{\mathbf{max}(S)-\mathbf{min}(S)}{2}$ and $\mathbf{scaling} = \frac{2}{\mathbf{max}(S)-\mathbf{min}(S)}$. The $\mathbf{shift}$ and $\mathbf{scaling}$ is recorded and used for the reconstruction of the original pattern form the reservoir generated result. Different signals of a pattern alongside their normalized versions can be seen in Figure 3.1.

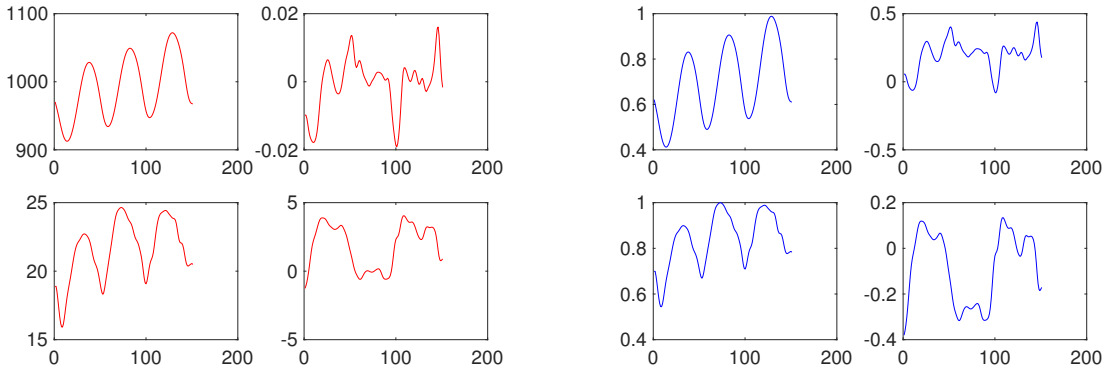

Figure 3.1: Red signals are the first four raw signals of the Jog pattern. Blue signals represent the first four normalized signals of the Jog pattern. The horizontal axis indicates the time n while the vertical axis the signal value.

Another important property of the data is the stationary/non-stationary nature. The stationary patterns are relatively easy to learn by the network. The non-stationary patterns

are a bit trickier to deal with. One has to make sure that the reservoir is already in a suitable state when it starts to generate a non-stationary pattern and to keep the reservoir in a stable state after the pattern generation has finished. How this can be achieved with ESNs and conceptors will be explained in Section 3.3.3 and 3.4.3 respectively. In our dataset there are 4 non-stationary patterns, namely:

- Standup From Chair : Getting up from a sitting position into a standing position.
- Get Seated : Sitting down from a standing position.
- Get Down : Getting down into a crawling position from a standing position.
- Standup : Getting up from a crawling position in a standing position.

As a last note about the dataset, signal number 5 and 17 were found to function primarily as noise. Their inclusion in the training phase induced instability in the resulting network, therefore these 2 signals were suppressed (set to 0).

3.2 Normalized Root Mean Square Error

In order to evaluate how well a signal has been learned by the network, the output pattern $\mathbf{y}(n)$ will be matched against the original pattern $\mathbf{p}(n)$, using normalized root mean square error:

$$\mathbf{NRMSE} = \sqrt{\frac{\sum_{i=1}^{N}(\mathbf{y}(i) - \mathbf{p}(i))^2}{N\hat{\sigma}(\mathbf{y} - \mathbf{p})}} \tag{3.3}$$

where N denotes the number of data points, $\hat{\sigma}(\mathbf{y} - \mathbf{p})$ denotes the estimated variance of the difference of $\mathbf{y}(n)$ and $\mathbf{p}(n)$ for all $n \in 1, ..., N$.

The lack of test data does not allow the computation of a testing $\mathbf{NRMSE}$ but we can get some insight into the quality of our pattern generator by looking at the training value of this entity. A value of $\mathbf{NRMSE}_{train}$ close to 1 means that the compared signals are completely unrelated to each other, which means that the training phase has almost certainly failed. A value close to 0 indicates that the trained network will perform very well on the training data but will most likely overfit on the testing data. A low $\mathbf{NRMSE}_{train}$ is desired but this cannot be used as an indicator of the final performance of the network.

3.3 ESN Pattern Generation

The following section focuses on the ESN component of the system, including the parameters used for tuning the network, details on the implementation of the readout training and the method chosen to morph the generated patterns.

3.3.1 State Update Equation

The network update equation, with the appropriate modifications discussed in Section 2.2, is repeated below for convenience.

$$\mathbf{x}(n+1) = (1 - \alpha)\,\mathbf{x}(n) + \alpha\,\mathbf{tanh}(\mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n) + \mathbf{b}) \tag{3.4}$$

Assume we have a reservoir with N internal neurons. $\mathbf{W} \in \mathbb{R}^{N \times N}$ and $\mathbf{W}^{back} \in \mathbb{R}^{N \times 61}$, the internal connection matrix and the feedback matrix, are initialized at the beginning with random values from a $[-1, 1]$ uniform distribution. $\mathbf{W}$ is further scaled by setting its spectral radius $|\lambda_{max}| = w_{scale}$. $\mathbf{W}^{back}$ is just scaled by a constant factor w^{back}_{scale}. The bias vector $\mathbf{b} \in \mathbb{R}^{N \times 1}$ contains values from a $[-1, 1]$ uniform distribution and is scaled by a constant factor b_{scale}.

During experimentation, the parameters that influenced the performance of the network the most were w_{scale}, the size N of the reservoir and the leaking rate α. It was observed that an increase in the reservoir size contributes to an overall increase in performance and a decrease in the sensitivity of each reservoir parameter. This is due to the fact that a larger reservoir has better memory capabilities and it can achieve a higher nonlinear dimensional expansion of the training signal than a smaller reservoir [24]. The smallest reservoir size that delivered very good results was $N = 1000$, but satisfactory results could already be obtained with a reservoir size as low as $N = 600$.

For w_{scale} the values that worked better were between 1 and 1.6. For the ESN approach this particular parameter was found to be very sensitive. Very small changes (of the order of 0.05) cause some patterns to stop functioning while fixing the generation of others. The most influenced patterns are:

- Waltz.
- Box 1.
- Box 2.

The Waltz patterns is a slow pattern and as a result bigger values of w_{scale} lead to better results. This is because the short-term memory that the network exhibits, which is necessary to produce slow patterns, is enhanced by a high information exchange between inner neurons. On the other hand, a high scaling of $\mathbf{W}$ easily lead the other two patterns in an undesired state. The two boxing patterns are highly stochastic and keeping old information flowing through the network does not help them, on the contrary it hinders their generation. For values of w_{scale} above 1.6 the reservoir stabilizes in a periodic and highly oscillating state, while for values smaller than 1 it generally dies out (stops oscillating completely).

Another important factor that determined the overall performance of the network was the leaking rate α. The optimal value of this parameter was found to be $\alpha \in [0.5, 0.6]$. Higher values usually cause difficulties in learning the slow patterns, while lower values cause difficulties in learning fast patterns. In our dataset we have both slow and fast patterns. Because of this, very small changes of α would improve the generation of some patterns and worsen the generation of others. This behavior was observed for any other value outside $[0.5, 0.6]$, which seems to strike a balance between patterns of different speeds.

The 2 parameters not mentioned up to now, w^{back}_{scale} and b_{scale}, lead only to small differences in performance when their values are kept within a reasonable interval. For w^{back}_{scale} this interval was $[0.8, 1.2]$, while $[0.1, 0.8]$ seems to work better for b_{scale}.

3.3.2 Training the Network

As stated in Section 2.2, the goal of the training algorithm is to compute the output weights $\mathbf{W}^{out}$, by minimizing the mean squared error in Equation 2.3. The general procedure used to achieve this is detailed in Equation 2.4.

Before starting to feed data to the network, the washout time n_0 has to be determined. Our dataset contains patterns of different lengths, ranging from 151 to 900 data points. A large washout size is desired because it increases the chance to remove any undesired effect left over from the initial network configuration. For patterns of very small size, as is the case of the Jog pattern, a large washout size might wash out data points that are essential in learning the pattern. During experimentation the optimal washout time was $n_0 = 50$, which allows the network to be trained on at least 100 data points for each pattern. Washout times in the interval $[30, 70]$ in general work well for all patterns.

Having decided the washout size, the output weights matrix $\mathbf{W}^{out} \in \mathbb{R}^{61 \times N}$ for each pattern is computed as follows:

- The reservoir is initialized with the state $\mathbf{x}(0) = 0$ and an initial teacher forced output $\mathbf{d}(0) = 0$.
- The network is run for n_0 steps to washout the initial state. The last state $\mathbf{s} = \mathbf{x}(n_0)$ and the last teacher forced output $\mathbf{o} = \mathbf{d}(n_0)$ are recorded. These states are crucial in the generation of testing sequences for non-stationary patterns.
- The network is run on the remaining data points and for each time point n the reservoir state $\mathbf{x}(n)$ and the teacher forced output $\mathbf{d}(n)$ are recorded and used to compute $\mathbf{W}^{out}$ in accordance with Equation 2.4. The mean absolute value ($\mathbf{MABS}$) of $\mathbf{W}^{out}$ is computed. The definition of $\mathbf{MABS}$ is the following:

$$\mathbf{MABS}(\mathbf{M}) = \frac{1}{K \times L} \sum_{i=1}^{K} \sum_{j=1}^{L} |(\mathbf{M})_{ij}| \tag{3.5}$$

  where $\mathbf{M} \in \mathbb{R}^{K \times L}$. During the computation of $\mathbf{W}^{out}$ a very important parameter is the ridge regression regularizer γ. The confidence interval for this parameter is $[0.1, 1]$. Higher values lead to underfitting while lower ones to overfitting.
- The training normalized root mean squared error $\mathbf{NRMSE}_{train}$ is computed according to Equation 3.3. This value is a good indicator of under-fitting if it is close to 1, but it does not lead us towards the right parameter selection, for it can be made arbitrarily small by increasing the network size.

One important indicator of the network performance that comes out of the training procedure is the maximum $\mathbf{MABS}(\mathbf{W}^{out})$ across all patterns. Experiments show that the smaller this value the better the performance of the network, the bigger this value the closer to overfitting we get. The ESN delivered the best results when $\mathbf{MABS}(\mathbf{W}^{out})$ across all patterns was within the interval $[0.001, 0.008]$.

3.3.3 Generating Test Data and Morphing Patterns

There are two ways of generating motion patterns after computing $\mathbf{W}^{out}$:

1. Generate patterns one by one without transitions between them
2. Morph (transition from one to another) patterns together.

To generate specific patterns Equation 2.2 is used. This equation, alongside the appropriate modifications discussed in Section 2.2, is repeated here for convenience.

$$\mathbf{y}(n) = \mathbf{W}^{out}\mathbf{x}(n) \tag{3.6}$$

When generating test data for a single pattern, at each time step n the network is first run through Equation 3.4 to compute $\mathbf{x}(n)$ and then the generated pattern value at this time $\mathbf{y}(n)$ is computed. Special attention must be put in non-stationary pattern generation. The initial testing network state and the initial output value matter. They cannot be random but must be chosen from network states and values obtained during training. In this project, I use as initial activations the values recorded after the washout phase during training, respectively $\mathbf{x}(0) = \mathbf{s}$ and $\mathbf{y}(0) = \mathbf{o}$. This initialization is further investigated in Section 4.1. In Figure 3.2 an example where the initial reservoir state and network output initializations lead to completely different results is given.

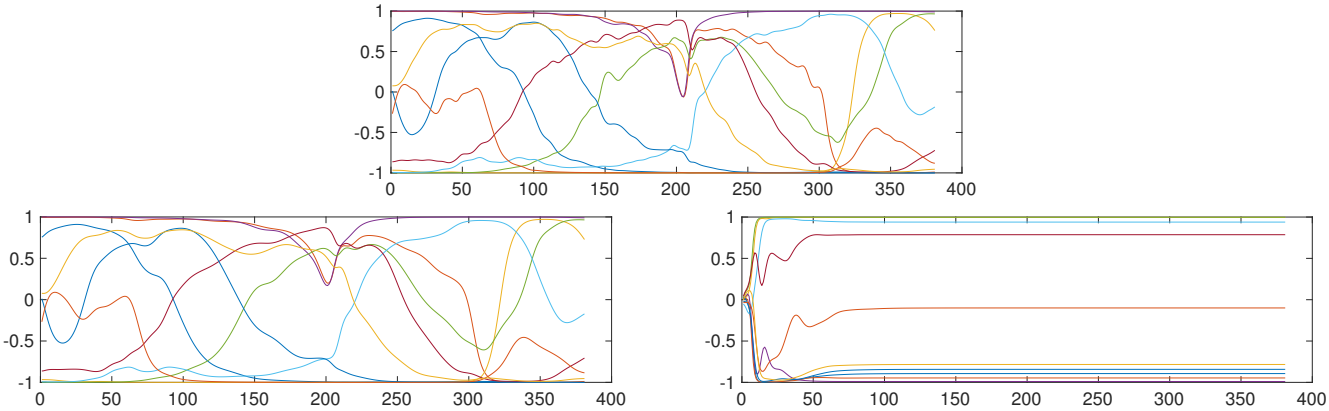

Figure 3.2: 10 random internal state activations generated for the Get Seated pattern. The horizontal axis indicates the time n while the vertical axis the signal value. Top image are the internal activations during training. The bottom left image shows the internal activations during testing when started with $\mathbf{x}(0) = \mathbf{s}$ and $\mathbf{y}(0) = \mathbf{o}$. The bottom right image shows the internal activations during testing when started with $\mathbf{x}(0) = 0$ and $\mathbf{y}(0) = 0$. The reservoir parameters are $N = 1000$, $w_{scale} = 1$, $w^{out}_{scale} = 1$, $\alpha = 0.6$, $b_{scale} = 0.1$ and $\gamma = 0.1$.

When morphing two patterns the state update Equation 3.4 is the same as in the previous case but the output computation during the morphing phase is the following:

$$\mathbf{y}(n) = (\mu\mathbf{W}^{out}_i + (1-\mu)\mathbf{W}^{out}_j)\,\mathbf{x}(n) \tag{3.7}$$

where $\mathbf{W}^{out}_i$ and $\mathbf{W}^{out}_j$ are the computed output weights of the i-th and j-th pattern respectively. $\mu \in [0, 1]$ is the morphing constant, which determines how similar the morphed pattern is to any of the initial ones. For $\mu \in [0, 1]$, the process is called interpolation between the two patterns, while outside of this range we are dealing with pattern extrapolation. When not in a morphing phase, the output is computed by using Equation 3.6.

In a morphing process we do not have the freedom of choosing the initial state or initial output of the network. This causes a problem when dealing with non-stationary patterns. The way this problem is handled in this project is to morph patterns where the end state of the previous pattern is sufficiently similar to an appropriate starting state of the next pattern. By using this strategy, one has to pay particular attention to the pattern generation sequence.

3.4 Conceptor Pattern Generation

The following section focuses on the conceptor component of the system including the parameters used for tuning the network, details on the computation of the conceptors and the method chosen to morph the generated patterns.

3.4.1 State Update Equation

The network update equation, with the appropriate modifications discussed in Section 2.3, is repeated below for convenience.

$$\mathbf{x}(n+1) = \mathbf{tanh}(\mathbf{W}^*\mathbf{x}(n) + \mathbf{W}^{in}\mathbf{p}(n) + \mathbf{b}) \tag{3.8}$$

$\mathbf{W}^* \in \mathbb{R}^{N\times N}$ and $\mathbf{W}^{in} \in \mathbb{R}^{N\times 61}$ are initialized with values form a $[-1, 1]$ uniform distribution. $\mathbf{W}^*$ is then scaled by setting its spectral radius $|\lambda_{max}| = w^*_{scale}$. This scaling is not as important as for ESNs, because the internal weights matrix $\mathbf{W}$, that will be used for pattern generation, is recomputed. $\mathbf{W}^{in}$ is scaled by a constant factor w^{in}_{scale}. The bias vector $\mathbf{b} \in \mathbb{R}^{N\times 1}$ contains values from a $[-1, 1]$ uniform distribution and is scaled by a constant factor b_{scale}. The conceptor approach can also be coupled with leaky-integrator neurons. In this project this is not done because the difference in results with and without leaky-integrator neurons was very small. In general the different speeds of the patterns can be handled by adapting the pattern specific apertures α.

The most important parameter in determining the performance of the network is the reservoir size N. In contrast with the ESN approach, where very good results can be obtained starting from $N = 1000$, a conceptor network of size $N = 600$ already generates accurate patterns and smooth pattern transitions. Higher reservoir sizes generate even better results but the computational cost of increasing the size of a conceptor driven network increases very quickly. The other parameters that affect Equation 3.8, w^*_{scale}, w^{in}_{scale} and b_{scale}, do not affect the network performance considerably if kept within reasonable intervals. These intervals are $[1, 1.8]$ for w^*_{scale}, $[0.7, 1.3]$ for w^{in}_{scale} and $[0.1, 1]$ for b_{scale}. Smaller networks have more restricted intervals, but almost always the parameter sensitivity was smaller than for ESNs.

3.4.2 Training the Network

The entities that must be computed during the training phase of a conceptor based network are:

1. The conceptor $\mathbf{C}^j \in \mathbb{R}^{N\times N}$ for each pattern $\mathbf{p}^j$.

2. A new set of reservoir weights $\mathbf{W} \in \mathbb{R}^{N \times N}$, which is the same across all pattern.
3. A set of output weights $\mathbf{W}^{out} \in \mathbb{R}^{61 \times N}$, which is the same across all patterns.

As for the ESNs, before proceeding with the training algorithm the washout size n_0 has to be determined. It was observed that for a conceptor network the same n_0 values as for the ESN approach (see Section 3.3.2) deliver optimal results.

Having decided the washout size, we can compute the conceptor for the j-th pattern as follows:

- The reservoir state is initialized with $\mathbf{x}^j(0) = 0$.
- The network is run for n_0 time steps to washout the initial state. The last state $\mathbf{s}^j = \mathbf{x}^j(n_0)$ is recorded. As for ESNs, $\mathbf{s}^j$ is essential in generating test sequences for non-stationary patterns.
- The network is run on the remaining data points and for each time step n the reservoir state $\mathbf{x}^j(n)$ is stored in a matrix $\mathbf{X}^j$ column wise and the current pattern data point $\mathbf{p}^j(n)$ is stored in a matrix $\mathbf{P}^j$ column wise. $\mathbf{X}^j$ is then used to compute the pattern specific conceptor $\mathbf{C}^j$ according to:

$$\mathbf{R}^j = \frac{1}{N}\mathbf{X}^j\mathbf{X}^{j\prime} \tag{3.9}$$

$$\mathbf{C}^j = \mathbf{R}^j(\mathbf{R}^j + (\alpha^j)^{-2}\mathbf{I})^{-1} \tag{3.10}$$

  where N is the size of the reservoir and α^j is the aperture of pattern $\mathbf{p}^j$. The aperture was not sensitive for most of the patterns. A value around 10 for 12 of the 15 patterns works very well. However, 3 patterns were found to be particularly sensitive to the aperture value. The Walk pattern works well with aperture values of 3 or 4, the Box 1 and Box 2 patterns need aperture values of 30 and 20 respectively in order to deliver optimal results. These values worked well across all tested network sizes, but they were not tested on networks with $N > 2000$.
- $\mathbf{X}^j$ and $\mathbf{P}^j$ contain data needed to compute $\mathbf{W}$ and $\mathbf{W}^{out}$. Therefore $\mathbf{X}^j$ is appended horizontally to a global matrix $\mathbf{X}$ and $\mathbf{P}^j$ is appended horizontally to a global matrix $\mathbf{P}$. Both $\mathbf{X}$ and $\mathbf{P}$ are empty matrices at the beginning of the training procedure.

After running the above procedure for every pattern, we can compute $\mathbf{W}$ and $\mathbf{W}^{out}$. $\mathbf{W}$ is computed using ridge regression according to:

$$\mathbf{D} = \mathbf{W}^*\mathbf{X} + \mathbf{W}^{in}\mathbf{P} \tag{3.11}$$

$$\mathbf{W}' = (\mathbf{X}\mathbf{X}' + \gamma_{\mathbf{W}}\mathbf{I})^{-1}\mathbf{X}\mathbf{D}' \tag{3.12}$$

where $\gamma_{\mathbf{W}}$ is the ridge regression regularizer for computing $\mathbf{W}$. Similarly for $\mathbf{W}^{out}$ we have:

$$(\mathbf{W}^{out})' = (\mathbf{X}\mathbf{X}' + \gamma_{\mathbf{W}^{out}}\mathbf{I})^{-1}\mathbf{X}\mathbf{P}' \tag{3.13}$$

where the $\gamma_{\mathbf{W}^{out}}$ is the ridge regression regularizer for computing $\mathbf{W}^{out}$. Both regularizers are very important parameters when training a conceptor based network. Usually assigning the same value from the interval $[0, 1]$ to both of them will lead to very good results. The sensitivity of the regularizers on a conceptor based approach is usually lower than for the ESN, as is the sensitivity of all other parameters.

The $\mathbf{MABS}$ as defined in Section 3.3.2 is computed for both $\mathbf{W}$ and $\mathbf{W}^{out}$. As for ESNs smaller values of this parameter deliver better results. The value intervals that delivered the best results were $[0.03, 0.12]$ for $\mathbf{W}$ and $[0.004, 0.01]$ for $\mathbf{W}^{out}$.

3.4.3 Generating Test Data and Morphing Patterns

There are 2 ways of generating patterns using conceptors:

1. Generate patterns one by one without transitions between them.
2. Morph (transition from one to another) patterns together.

The generate specific patterns $\mathbf{p^j}$ the following equations are used:

$$\mathbf{x}^j(n+1) = \mathbf{C}^j \, \mathbf{tanh}(\mathbf{W}\mathbf{x}^j(n) + \mathbf{b}) \tag{3.14}$$

$$\mathbf{y}^j(n+1) = \mathbf{W}^{out}\mathbf{x}^j(n) \tag{3.15}$$

where $\mathbf{W}^{out}$ and $\mathbf{W}$ are the same across all patterns. The term that leads the reservoir towards generating the j-th pattern is the conceptor $\mathbf{C}^j$. The same attention as for ESNs has to be paid to non-stationary patterns. When generating such patterns the initial reservoir state has to be one of the states obtained during training. For this reason I choose $\mathbf{x}^j(0) = \mathbf{s}^j$, which has been recorded during training. In general conceptors proved to be more robust than ESNs when they are started in an undesired state. Adapting the aperture of the pattern also helped in solving this issue. The improved robustness of the network is demonstrated in Figure 3.3.

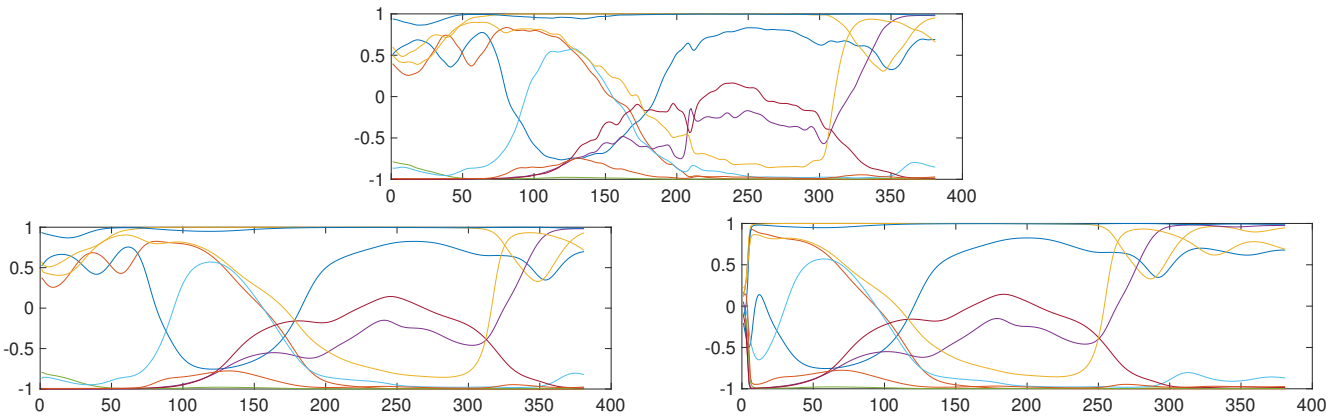

Figure 3.3: 10 random internal state activations generated for the Get Seated pattern. The horizontal axis indicates the time n while the vertical axis the signal value. Top image are the internal activations during training. The bottom left image shows the internal activations during testing when started with $\mathbf{x}(0) = \mathbf{s}$. The bottom right image shows the internal activations during testing when started with $\mathbf{x}(0) = 0$. The reservoir parameters are $N = 1000$, $w_{scale} = 1$, $w^{in}_{scale} = 1$, $b_{scale} = 0.3$, $\gamma_{\mathbf{W}} = 0.5$, $\gamma_{\mathbf{W}^{out}} = 0.5$ and $\alpha^j = 10$.

During morphing the network state and the network output are computed according to:

$$\mathbf{x}(n+1) = (\mu\mathbf{C}^j + (1-\mu)\mathbf{C}^i) \, \mathbf{tanh}(\mathbf{W}\mathbf{x}(n) + \mathbf{b}) \tag{3.16}$$

$$\mathbf{y}(n+1) = \mathbf{W}^{out}\mathbf{x}(n) \tag{3.17}$$

where $\mathbf{C}^j$ and $\mathbf{C}^i$ are the conceptors of the i-th and j-th pattern respectively. $\mu \in [0, 1]$ is the morphing constant, which determines how similar the morphed pattern is to any of the initial ones. If not in a morphing phase, when generating pattern $\mathbf{p}^j$ the network is again driven by Equation 3.14 and 3.15.

Morphing two completely unrelated patterns, for example the Cartwheel pattern with the Standup pattern, does not work in most cases even with robust conceptors. The pattern generation sequence is again a very important factor. Patterns whose end state is not sufficiently similar to the next pattern's starting state cannot be morphed together.

4 Comparison

This section provides a comparison of ESNs and conceptor networks on the basis of robustness, stability, time and space complexity and generated motion quality.

4.1 Robustness

Robustness refers to the ability of the network to produce the correct pattern independently of the starting state $\mathbf{x}(0)$. To test this an ESN and a conceptor network that deliver similar optimal results were considered. The specific parameters are given in Table 4.1.

Parameter	Value
N	1000
w_{scale}	1
w^{back}_{scale}	1
b_{scale}	0.3
α	0.6
γ	0.5

Parameter	Value
N	1000
w_{scale}	1
w^{in}_{scale}	1
b_{scale}	0.8
$\gamma_{\mathbf{W}}$	0.5
$\gamma_{\mathbf{W}^{out}}$	0.5

Table 4.1: The left table refers to the ESN parameters which resulted in $\mathbf{MABS}(\mathbf{W}^{out}) = 0.0016$ and $\mathbf{NRMSE}_{train} = 0.0275$. The right table refers to the conceptor parameters which resulted in $\mathbf{MABS}(\mathbf{W}^{out}) = 0.0066$, $\mathbf{NRMSE}(\mathbf{W}^{out}) = 0.0435$, $\mathbf{MABS}(\mathbf{W}) = 0.0312$ and $\mathbf{NRMSE}(\mathbf{W}) = 0.0057$.

After training all networks on all patterns, their task was to generate 5 selected patterns starting from different initial states $\mathbf{x}(0)$. The $\mathbf{x}(0)$ were constructed as follows:

$$\mathbf{x}(0) = \mathbf{s} + \eta \tag{4.1}$$

where $\mathbf{s}$ has been introduced in Section 3.3.2 and 3.4.2, for ESNs and conceptors respectively. $\eta \in \mathbb{R}^{N \times 1}$ is a random noise vector sampled from a $[-50, 50]$ uniform distribution. For ESNs, the starting teacher forced input was set to $\mathbf{y}(0) = \mathbf{o}$ as done in Section 3.3.3. Each of the 5 patterns was generated 100 times from each network, each time with a different η and the number of times the network would fail to converge to the desired pattern was recorded. The result of this runs are given in Table 4.2.

The chosen patterns varied in learning difficulty, speed and stationary nature. The results show that the conceptors are more robust than ESNs across all patterns. For more

Pattern	Failure Rate
Jog	0%
Cartwheel	16%
Waltz	35%
Box 3	84%
Get Seated	87%
Average	44.4%

Pattern	Failure Rate
Jog	0%
Cartwheel	1%
Waltz	30%
Box 3	31%
Get Seated	57%
Average	23.8%

Table 4.2: The left table shows the results of the ESN from Table 4.1, while the right table shows the results of the conceptor from Table 4.1.

regular and periodic patterns (Waltz and Jog) the results do not differ very much, but for stochastic (Box 3) patterns, patterns with only a few training points (Cartwheel) and non-stationary patterns (Chair Sitting) the conceptor approach clearly outperforms the ESNs in terms of robustness.

More experiments were run to test the hypothesis that conceptors are more robust than ESNs. The networks used for the first of these experiments are specified in Table 4.3. Both of them delivered optimal graphical results on all 15 patterns.

Parameter	Value
N	1000
w_{scale}	1.6
w^{back}_{scale}	0.8
b_{scale}	0.4
α	0.5
γ	1

Parameter	Value
N	1000
w_{scale}	1.6
w^{in}_{scale}	0.8
b_{scale}	0.4
$\gamma_{\mathbf{W}}$	1
$\gamma_{\mathbf{W}^{out}}$	1

Table 4.3: The left table refers to the ESN parameters which resulted in $\mathbf{MABS}(\mathbf{W}^{out}) = 0.0015$ and $\mathbf{NRMSE}_{train} = 0.0328$. The right table refers to the conceptor parameters which resulted in $\mathbf{MABS}(\mathbf{W}^{out}) = 0.0065$, $\mathbf{NRMSE}(\mathbf{W}^{out}) = 0.0454$, $\mathbf{MABS}(\mathbf{W}) = 0.0328$ and $\mathbf{NRMSE}(\mathbf{W}) = 0.0072$.

Pattern	Failure Rate
Jog	9%
Cartwheel	32%
Waltz	23%
Box 3	23%
Get Seated	74%
Average	30.4%

Pattern	Failure Rate
Jog	0%
Cartwheel	3%
Waltz	27%
Box 3	12%
Get Seated	60%
Average	20.4%

Table 4.4: The left table shows the results of the ESN from Table 4.3, while the right table shows the results of the conceptor from Table 4.3.

As we can see from the results in Table 4.4 the conceptor network performs almost the same overall while the ESN has a noticeable improvement. For a network of size N, the output weight matrix $\mathbf{W}^{out}$ holds a total of $61 \times N$ weights, while the conceptor $\mathbf{C}$ holds N^2 weights. The ratio of the number of conceptor weights with the number of output weight matrix weights for a network of the given size is $N/61$. Thus the bigger

the network the more advantage the conceptor approach is expected to have in terms of robustness. Comparing conceptors with ESN of the same size will almost always lead to better conceptor results. The last performed experiment aims to compare a much bigger ($N = 2000$) ESN with the results of the conceptor from Table 4.3.

Parameter	Value
N	2000
w_{scale}	1
w^{back}_{scale}	1
b_{scale}	0.3
α	0.6
γ	0.1

Table 4.5: ESN parameters which resulted in $\mathbf{MABS}(\mathbf{W}^{out}) = 0.0013$ and $\mathbf{NRMSE}_{train} = 0.0156$.

Pattern	Failure Rate
Jog	0%
Cartwheel	0%
Waltz	27%
Box 3	12%
Get Seated	74%
Average	22.6%

Table 4.6: Results of the ESN from Table 4.5.

As we can see in Table 4.6 the ESN results improve considerably, but it takes a network of size $N = 2000$ to achieve these results. An increased network size adds to the computational complexity of generating new pattern data and does not make the network more robust than with a $N = 1000$ conceptor network. All in all, ESNs are not more robust than conceptor driven networks. Their robustness can be increased by increasing the model size but this way they lose their computational edge to the conceptors.

4.2 Stability

To test the stability of a method, the network was trained on all patterns and then it was run for 1000 time steps on each of the analyzed patterns. During the first 250 time steps noise $\eta(n)$ sampled from a $[-1, 1]$ normal distribution was added to the network. The resulting equations for driving the network during these 250 time steps are shown in Equation 4.2 and 4.3 for ESNs.

$$\mathbf{x}(n+1) = (1-\alpha)\,\mathbf{x}(n) + \alpha\,\mathbf{tanh}(\mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n) + \mathbf{b}) + \eta(n+1) \tag{4.2}$$

$$\mathbf{y}(n+1) = \mathbf{W}^{out}\mathbf{x}(n+1) \tag{4.3}$$

The conceptor network driving and output equations are given in 4.4 and 4.5.

$$\mathbf{x}(n+1) = \mathbf{C}\,(\mathbf{tanh}(\mathbf{W}\mathbf{x}(n) + \mathbf{b}) + \eta(n)) \tag{4.4}$$

$$\mathbf{y}(n+1) = \mathbf{W}^{out}\mathbf{x}(n+1) \tag{4.5}$$

During stability experiments, the same networks as in Section 4.1 are used. The pattern group has only 1 change, the Get Seated motion has been removed since it is a non-stationary pattern and after 250 time steps the network generated result is unpredictable. The above described experiment was run 100 times for each motion network combination and the number of times the network failed to continue generating the correct pattern was recorded.

Pattern	**Failure Rate**
Jog	0%
Cartwheel	19%
Waltz	65%
Box 3	0%
Average	21%

Pattern	**Failure Rate**
Jog	0%
Cartwheel	0%
Waltz	22%
Box 3	0%
Average	5.5%

Table 4.7: The left table shows the results of the ESN from Table 4.1, while the right table shows the results of the conceptor from Table 4.1.

Pattern	**Failure Rate**
Jog	0%
Cartwheel	45%
Waltz	3%
Box 3	3%
Average	12.75%

Pattern	**Failure Rate**
Jog	0%
Cartwheel	0%
Waltz	13%
Box 3	0%
Average	3.25%

Table 4.8: The left table shows the results of the ESN from Table 4.3, while the right table shows the results of the conceptor from Table 4.3.

Pattern	**Failure Rate**
Jog	0%
Cartwheel	0%
Waltz	8%
Box 3	0%
Average	2%

Table 4.9: Results of the ESN from Table 4.5.

The results in Table 4.7, 4.8 and 4.9 show that conceptors are more stable than ESNs. The conceptor networks failed less than 10% of the time while the first 2 ESNs failed in more than 10% of the trials. The last ESN, of significantly bigger size, showed results slightly better than the conceptor approach but at a much higher computational cost. It was noticed that with high reservoir sizes both methods tend to become numerically unstable. Already with a reservoir of size 2000 some of the eigenvalues of the unscaled $\mathbf{W}$ ($\mathbf{W}^*$ for conceptors) fail to converge if the built in MATLAB functions are used. The instabilities are pointed out by the MATLAB runtime system for the ridge regression step, where the numerical methods used to invert a matrix fail to converge quickly. This is one more reason to favor conceptor, which deliver very good results for small reservoir sizes, over ESNs.

4.3 Space and Time Complexity

For the purpose of this section assume matrix multiplication of 2 $n \times n$ matrices has $O(n^3)$ time complexity. For simplicity assume there are P patterns, with T_1 data points each and the size of the reservoir is N. The trained network will be used to generate T_2 data points for each pattern. The time complexity of each training and testing step for the ESNs and conceptors is given respectively in Table 4.10 and 4.11.

Stage	**Time Complexity**
Initialization ($\mathbf{W}$, $\mathbf{W}^{back}$, $\mathbf{b}$)	$O(N^3)$
Recording training data ($\mathbf{X}$, $\mathbf{D}$)	$O(PT_1N^3)$
Computing $\mathbf{W}^{out}$	$O(P(T_1+N)^3)$
Generating test patterns	$O(PT_2N^3)$
Overall	$O(P(T_1+T_2)N^3) + O(P(T_1+N)^3)$

Table 4.10: ESN training and testing asymptotic time complexity.

Stage	**Time Complexity**
Initialization ($\mathbf{W}^*$, $\mathbf{W}^{in}$, $\mathbf{b}$)	$O(N^3)$
Recording training data ($\mathbf{X}$, $\mathbf{P}$)	$O(PT_1N^3)$
Computing $\mathbf{W}^{out}$, $\mathbf{W}$	$O((PT_1+N)^3)$
Computing conceptors $\mathbf{C}$	$O(PN^3)$
Generating test patterns	$O(PT_2N^3)$
Overall	$O(P(T_1+T_2)N^3) + O((PT_1+N)^3)$

Table 4.11: Conceptor training and testing asymptotic time complexity.

The conceptor method is more computationally expensive than the ESN approach. The initialization of the reservoir is the same, with the spectral radius of the internal weight matrix having cubic complexity. Recording data is again computationally the same, but there is a difference in the way the data is used. In an ESN approach, to compute $\mathbf{W}^{out}$ only the data for one pattern is needed. With conceptors, the data from one particular pattern is used only to compute the respective conceptor, but for $\mathbf{W}$ and $\mathbf{W}^{out}$ the cumulative data from all patterns is needed, which makes the computation slower. In computing $\mathbf{W}$ and $\mathbf{W}^{out}$ what takes the most time is the multiplication of the state matrix $\mathbf{X}$ with its transpose and inverting the result. The last step, generating test patterns, is the same asymptotically but the additional multiplication of the network state with the conceptor adds another constant complexity factor to the conceptor network.

When talking about space efficiency ESNs still outperform the conceptor based networks. Asymptotic space complexity summaries can be found in Table 4.12 and 4.13.

Stage	**Space Complexity**
Storing $\mathbf{W}$, $\mathbf{W}^{back}$, $\mathbf{b}$	$O(N^2)$
Storing training data ($\mathbf{X}$, $\mathbf{D}$)	$O(T_1N)$
Storing $\mathbf{W}^{out}$	$O(PN)$
Overall	$O((P+T_1+N)N)$

Table 4.12: ESN asymptotic space complexity.

Stage	Space Complexity
Storing $\mathbf{W}$, $\mathbf{W}^*$, $\mathbf{W}^{in}$, $\mathbf{W}^{out}$, $\mathbf{b}$	$O(N^2)$
Storing training data ($\mathbf{X}$, $\mathbf{P}$)	$O(PT_1N)$
Storing conceptors $\mathbf{C}$	$O(PN^2)$
Overall	$O((T_1 + N)PN)$

Table 4.13: Conceptor asymptotic space complexity.

As we can see the conceptors are outperformed by ESNs even in terms of space complexity. The network setup phase uses the same amount of memory. The biggest difference is noticed when training the network and storing the results. As explained above, for the conceptor approach the training data from all patterns is needed to compute $\mathbf{W}$ and $\mathbf{W}^{out}$, which is not the case for ESNs. Furthermore, the conceptor $\mathbf{C}$ contains more parameters than $\mathbf{W}^{out}$, $O(N)$ more to be precise. Therefore, the conceptors are quite space inefficient in comparison to ESNs.

4.4 Motion Quality

When looking at the quality of the generated motion there are several observations that have to be made:

1. Does the pattern look human?
2. Do the pattern morphings look human?
3. How fast do the patterns converge after morphing?

Human means that the generated motion sequence does not have any irregularities (abrupt movements, freeze moments, lift from the ground, body distortions etc.). For both approaches used in this project, all patterns generated by a network with optimal parameters do look human. This is very impressive since no post-processing was done on any of the generated patterns to instruct the character to touch the ground at all times or not to rotate in place while moving. These movement constraints are learned by the network alongside the respective pattern.

As for the pattern morphings, for both approaches, they do not look human. Each method tends to break the human like flow in different ways. Conceptors tend to have very little movement during the morphing phase. Sometimes it appears as the character is staying still until the conceptor of the next pattern is activated completely. Besides the slowness of the morphing process, conceptors do not show any other deviation from real human motion. The ESNs on the other hand tend to have very lively pattern morphings. During the morphing phase the character might make abrupt movements, freeze or even elevate from the ground completely. None of the approaches has the edge over the other when it comes to pattern morphing.

Convergence speed is another issue without post-processing the generated patterns. After the respective pattern $\mathbf{W}^{out}$ for ESNs or $\mathbf{C}$ for conceptors has been fully activated the network does not start to generate the desired pattern immediately. ESN are faster in converging towards the new pattern (if convergence is possible from the current state) but the first few frames of the motion might not look human like. Figure 4.1 shows a

situation where a body distortion occurred most of the time. Conceptors did not show any unhuman like behavior but they were slightly slower at converging to the next pattern.

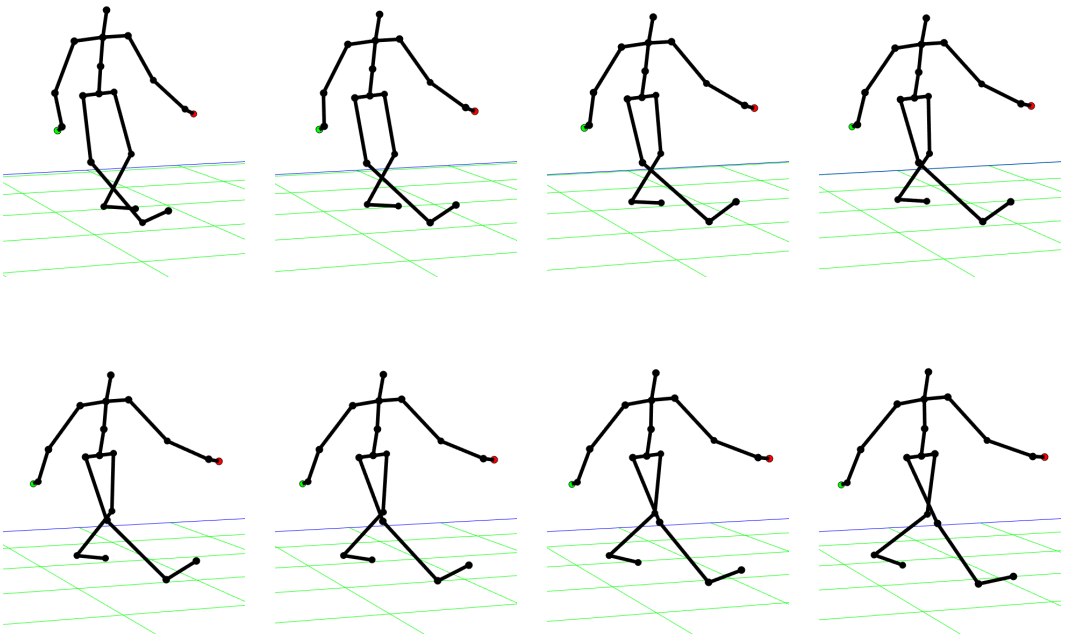

Figure 4.1: 8 frames of the Walk pattern generated after the Waltz pattern. Sequence starts from the top left frame and proceeds row by row. The right foot of the character is bent in an unnatural way.

5 Conclusion and Future Work

This section will present the conclusions of this project and discus further improvements on the current work.

This project tried to solve the problem of human motion animation generation using two methods, Echo State Networks and conceptor based Recurrent Neural Networks. The final goal was not to compete with state of the art solutions but rather to compare the two selected methods.

ESNs showed great potential since the very beginning because of their computational and space efficiency. It was possible to develop ESN networks for all of the patterns in appendix A with satisfactory graphical results, even without post-processing the generated motions. However they showed a lack of robustness and stability when stress tested in addition to unpredictable movements during pattern morphing.

Conceptors had already been tried as a solution for HMAG [2]. This project used a slightly different version without leaky integrator neurons. Conceptors introduce quite an overhead in terms of space and time complexity but are very robust and stable. Furthermore pattern transitions look more human like compared to ESN generated ones, but are still far from the desired quality.

One last approach that was tried during the end of the project, but not documented in this report, are Random Feature Conceptor (RFC) networks [4]. They offer a solution to the computational and space overhead that conceptors introduce without any loss in robustness or stability. The patterns used in this project were found to be very short to work with this approach. The task that lies ahead is to find longer patterns that allow the RFCs to be learned online during training.

A Dataset Summary

In this section I provide the name, dimension, length (number of data points) and stationary/non-stationary nature for each pattern.

Pattern	**Dimension**	**Length**	**Stationary** (True / False)
Exaggerated Stride	61	291	True
Slow Walk	61	241	True
Walk	61	320	True
Jog	61	151	True
Cartwheel	61	451	True
Waltz	61	900	True
Crawl	61	201	True
Standup	61	201	False
Get Down	61	241	False
Stay Seated	61	201	True
Get Seated	61	451	False
Standup From Chair	61	301	False
Box 1	61	451	True
Box 2	61	701	True
Box 3	61	301	True

Table A.1: Short summary of the used dataset.

Table A.1 provides the summary for the normalized version of the data. They can be found in appropriate .mat files in the project code repository [26]. The raw versions of the data, the ones provided by the CMU database, are not in this repository.

References

[1] Marco Gillies and Bernhard Spanlang. Comparing and evaluating real time character engines for virtual environments. *Teleoper. Virtual Environ.*, 19(2):95–117, apr 2010. http://dx.doi.org/10.1162/pres.19.2.95.

[2] Herbert Jaeger. Using conceptors to manage neural long-term memories for temporal patterns. *Journal of Machine Learning Research*, 18(13):1–43, 2017. http://jmlr.org/papers/v18/15-449.html.

[3] Sotirios P. Chatzis and Yiannis Demiris. A reservoir-driven non-stationary hidden Markov model. *Pattern Recognition*, 45(11):3985 – 3996, 2012. http://www.sciencedirect.com/science/article/pii/S0031320312001987.

[4] Herbert Jaeger. Controlling recurrent neural networks by conceptors. *CoRR*, 2014. http://arxiv.org/abs/1403.3369.

[5] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127 – 149, 2009. http://www.sciencedirect.com/science/article/pii/S1574013709000173.

[6] Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th European Symposium on Artificial Neural Networks*, pages 471–482, 2007. https://biblio.ugent.be/publication/416607.

[7] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015. http://www.sciencedirect.com/science/article/pii/S0893608014002135.

[8] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004. http://science.sciencemag.org/content/304/5667/78.

[9] Okan Arikan and David A. Forsyth. Interactive motion generation from examples. *ACM Trans. Graph.*, 21(3):483–490, July 2002. http://doi.acm.org/10.1145/566654.566606.

[10] Michael Gleicher. Retargetting motion to new characters. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 33–42, New York, NY, USA, 1998. ACM. http://doi.acm.org/10.1145/280814.280820.

[11] Alexei A. Efros and Thomas Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1033–1038, 1999. http://ieeexplore.ieee.org/document/790383/?arnumber=790383&tag=1.

[12] Jehee Lee and Sung Yong Shin. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 39–48, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. http://dx.doi.org/10.1145/311535.311539.

[13] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. *ACM Trans. Graph.*, 21(3):473–482, July 2002. http://doi.acm.org/10.1145/566654.566605.

[14] Luis Molina Tanco and Adrian Hilton. Realistic synthesis of novel human movements from a database of motion capture examples. In *Proceedings Workshop on Human Motion*, pages 137–142, 2000. http://ieeexplore.ieee.org/document/897383/?arnumber=897383.

[15] Ronald A. Metoyer David C. Brogan and Jessica K. Hodgins. Dynamically simulated characters in virtual environments. *IEEE Computer Graphics and Applications*, 18(5):58–69, Sep 1998. http://ieeexplore.ieee.org/document/708561/?arnumber=708561.

[16] Maja J. Mataric. Getting humanoids to move and imitate. *IEEE Intelligent Systems*, 15(4):18–24, July 2000. http://dx.doi.org/10.1109/5254.867908.

[17] Stephen Chenney and David A. Forsyth. Sampling plausible solutions to multi-body constraint problems. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 219–228, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. http://dx.doi.org/10.1145/344779.344882.

[18] CMU Graphics Lab Motion Capture Database. http://mocap.cs.cmu.edu/.

[19] MoCap Toolbox. https://www.jyu.fi/hum/laitokset/musiikki/en/research/coe/materials/mocaptoolbox/.

[20] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks - with an Erratum note. *German National Research Center for Information Technology GMD Technical Report*, 2001. http://www.faculty.jacobs-university.de/hjaeger/pubs/EchoStatesTechRep.pdf.

[21] Herbert Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach. *German National Research Center for Information Technology GMD Technical Report*, 2002. http://www.pdx.edu/sites/www.pdx.edu.sysc/files/Jaeger_TrainingRNNsTutorial.2005.pdf.

[22] Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, and Udo Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural networks*, 20(3):335–352, 2007. http://www.sciencedirect.com/science/article/pii/S089360800700041X.

[23] Herbert Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007. http://www.scholarpedia.org/article/Echo_state_network.

[24] Mantas Lukoševičius. A practical guide to applying echo state networks. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade: Second Edition*, pages 659–686. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. http://dx.doi.org/10.1007/978-3-642-35289-8_36.

[25] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer Series in Statistics Springer, Berlin, 2001. http://statweb.stanford.edu/~tibs/book/.

[26] Personal RNN Repository. https://github.com/rdeliallisi/RNN.