Master Thesis

Character Motion Synthesis using Deep Neural Networks

Author(s):
Lüdi, Marcel

Publication Date:
2016

Permanent Link:

https://doi.org/10.3929/ethz-a-010862039 →

Character Motion Synthesis using Deep Neural Networks

Marcel Lüdi

Master Thesis
December 2016

Prof. Dr. Robert Sumner

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
computer graphics laboratory

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Character Motion Synthesis Using Deep Neural Networks

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Lüdi	Marcel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Oberburg, 18.12.2016	M. Lüdi

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Abstract

I present two methods to synthesize character movements. The first approach learns a mapping from sparse input poses to a full animation from a large motion capture database using a deep convolutional neural network. This approach can produce smooth natural looking walking motions without any manual pre-processing of the training data. The second technique automatically learns a low dimensional representation of human walking motions. By modeling the latent space of the variational autoencoder as a multi-variate Gaussian distribution with high standard deviation the system learns an efficient encoding with disentangled variables. This prevents a fine grained representation where small changes in latent space can result in large differences in the resulting animation. As a result the latent components can be controlled directly by a human to synthesize character motions.

Zusammenfassung

In dieser Arbeit präsentiere ich zwei Systeme zur Herstellung von animierten menschlichen Bewegungen. Der erste Ansatz lernt eine Abbildung von wenigen input Posen auf eine vollständige Animation mit Hilfe einer grossen Motion Capture Datenbank. Das System beruht auf einem Convolutional Neural Network welches realistische Gehbewegungen erzeugen kann ohne dass die Trainingsdaten manuell vorbearbeitet werden müssen. Die zweite Methode lernt eine niedrig dimensionale Darstellung von menschlichen Gehbewegungen. Da der Latente Raum als multi-variate Normalverteilung mit grosser standardabweichung modeliert wird, lernt das System eine effiziente kodierung mit unabhängigen Variablen. Dies verhindert eine sensible Darstellung bei welcher schon eine kleine Änderung im latenten Raum eine grosse Auswirkung auf die produzierte Animation haben kann. Dadurch können die latenten Komponenten direkt von einem Menschen kontrolliert werden um menschliche Bewegungen herzustellen.

Contents

List of Figures

List of Tables

1

Introduction

Controlling the motion of a digital character is a challenging task as the dimensionality of the character's digital representation is generally high. A long lasting goal of computer animation is to provide intuitive high-level controls for the character's shape and motion. Data-driven techniques allow the user to generate various new motions by providing high level parameters. Such applications greatly reduce the work for animators as they do not have to create low level details but rather provide high level instructions from which the animation is created.

However, building these models often requires a significant amount of manual preprocessing. Motion clips must either be aligned temporally or they must be manually organized with respect to parametric control nodes. While automatic labeling systems exist, at this stage a mistake can easily make the produced animation unusable. The preprocessing is therefore often done with human intervention to ensure the resulting motions look natural.

In this Thesis I propose two different models of animation synthesis. The first system uses deep learning to produce a full animation from sparse input poses. I train a deep convolutional neural network on a large set of human motion data such that it can recreate the animation given only few poses of the original clip. This learning process can be trained on arbitrary motion clips without requiring any segmentation or alignment making it a much easier process than previous approaches.

The second model addresses the goal of automatically extracting a meaningful parameterization of human motions from a disorganized collection of captured data. Previous attempts such as linear dimensionality reduction require aligning the motion clips and the reduced space is not designed to be manipulated by humans. Deep convolutional neural networks, and more specifically autoencoders, can be used to automatically learn low dimensional representations from disorganized motion clips [HSK16]. The learned latent space is however generally not suited to be manipulated directly by humans and is rather used to automatically satisfy some constraints. There are a few reasons why this latent space is unsuited for human manipulation. Firstly the dimensionality required for successful reconstruction remains quite high. Secondly

the latent variables are trained to contain as much useful data as possible. This allows the network to store more details in latent space but the recreation is very sensitive to the values of the latent variables. Even small changes in this space can have a large impact on the produced animation. Another problem is the entanglement of the variables. They do not change one single aspect independently but cause different results based on the values of the other dimensions.

In this work I automatically learn a disentangled latent space with low granularity that is meaningful enough for human use. I draw upon the concept of variational autoencoders [KW13] and represent the latent space as a multi-variate Gaussian distribution. This is achieved by penalizing deviations from a canonical multi-variate normal distribution, i.e. $H \sim N(0, I)$, in latent space. This approach forces the network to be very efficient with its variables and results in few mostly independent dimensions of the multi-variate Gaussian distribution. The large coverage of each latent variable prevents fine granularity and the independence between dimensions favors disentanglement. As a result each latent variable encodes a consistent portion of the motion space and the network preserves this consistency when the user manipulates other variables.

While variational autoencoders have been applied to other data sets before such as 2D images and sentences [WDGH16, BVV+15] the application to 3D human motion is new and lead to challenges of its own. There is a tradeoff between the quality of the encoding and the meaningfulness provided by the disentanglement. This is controlled online during the optimization process which makes it hard to train these networks. I train an intermediate latent space of 20 dimensions and perform a final linear dimensionality reduction on those to further reduce down to 3 dimensions. In short I reduce the space of one second motion clip of 61 frames for a skeleton with 66 degrees of freedom totaling 4026 dimensions down to 20 dimensions with the variational autencoder and furthermore to 3 using principle component analysis.

2

Related Work

This chapter reviews some previous approaches to motion synthesis. Here various techniques are discussed and their advantages and disadvantages are presented. The second part gives an introduction to neural networks which are the main focus of this work. In particular the variational autoencoder is presented to give an intuition on how it achieves its goals.

2.1 Animation Synthesis

In this section I review different approaches to motion synthesis. The first part focuses on kernel-based methods which have been the main technique for blending motion capture data to produce novel motions. I next show some approaches of interactive character control where the user can input some instructions which are applied to generate new animations using a motion database. The next part shows current applications of deep learning in character animation. Finally I show the use of variational autoencoders in various dimension reduction tasks.

2.1.1 Kernel-Based Methods For Motion Blending

An often used tool to produce novel motions is to blend multiple existing animations of the same class using radial-basis functions (RBF). This approach is used by Rose et al. [RBC98] who call those classes "verbs" and interpolate between them using RBFs. The verbs are chosen based on constraints for the character movement. Another usage of RBFs is for inverse kinematics [RISC01, YKH04] where the joint angles are chosen to be biased towards a natural looking pose extracted from a database. For those blended motions to appear plausible a large body of animations needs to be categorized and aligned. This process often has to be done manually to achieve good results.

Some automatic categorization and alignment algorithms have been proposed. One approach is to use similarity between the movements [KG04]. The clips are aligned using dynamic time warping to find the best alignment. However this approach can easily overfit to the data since there is no process to deal with noise and variance. To overcome this issue Mukai and Kuriyama [MK05] use Gaussian Processes where a model is fit to the data by optimizing metaparameters. A similar approach using a Gaussian Process Latent Variable Model maps the motion data to a low dimensional space [GMHP04]. The user can then impose constraints on the character such as style which are then optimized for in this reduced subspace called "style-IK" by the author. Wang et al. [WHB05] apply the same technique for time series motion data. Their method learns the pose in the next frame given data for the previous frames. Levine et al. [LWH+12] use the reduced subspace from the Gaussian Process Latent Variable Model to compute optimal motions for various tasks using reinforcement learning.

The drawback of kernel-based methods is that they require a large database of motions to take the samples from. This database needs to be present when applying those techniques which results in either a large memory cost or in reduced number of possible motions, limiting their usability in real world applications. My approach eliminates this cost and can be trained on an arbitrary large set of training data.

2.1.2 Interactive Character Control

To achieve interactive character control a system needs to produce a continuous motion based on high level input from the user. An effective way to achieve that is the use of motion graphs [KGP02]. This approach uses the methods described above to create rich models for different motion classes and generating transitions between them [HG07].

To find the best action to take after user interaction, many techniques rely on reinforcement learning [LPY16]. With this method the model learns by rewarding beneficial actions and penalizing bad motions [SB98]. However this technique relies heavily on precomputation which increases exponentially with the number of possible actions. Because of this, methods relying on this such as motion fields [LWB+10] quickly become too complex as the number of possible actions increases. To reduce this complexity the dimensionality within the classes can be reduced as shown by Levine et al. [LWH+12].

My approach does not rely on reinforcement learning and thus does not suffer from the high dimensionality in possible actions.

2.1.3 Deep Learning

Deep learning has proven to be a very useful tool to find intricate structure in high-dimensional data found in many fields [LBH15]. Techniques applying deep learning are the current state-of-the art in object recognition [KSH12, CMS12] and have been successfully used in many other tasks such as video classification [KTS+14, JXYY13] and speech recognition [GMH13]. Recently there has been an interest in using deep learning techniques to produce novel data from the learned model [GPAM+14, VLL+10]. The strength of a deep learning approach is that they automatically find appropriate features in the dataset. In image recognition the first layers often

produce filters similar to known edge detection filters while in later layers more complex filters corresponding to different objects appear [ZF14].

Deep learning has been used in physically based animation to learn control graphs for articulated characters. Levine et al. [LK14] use a neural network to learn optimal control policies and apply it to bipedal characters. Peng et al. [PBvdP16] use deep reinforcement learning to find terrain-adaptive dynamic locomotion skills.

Recently application of deep learning in character animation has gained some interest. Du et al. [DWW15] use a hierarchical recurrent neural network trained on a large animation set to classify different motions. This approach achieves a state-of-the-art recognition rate showing the potential of these methods. Holden et al. [HSPK15] use a convolutional autoencoder to produce a latent representation of some input animation. They show that like in image recognition the filters learned by the neural network show strong correlations between joints in the first layer.

Taylor et al. [TH09, THR11] use conditional Restricted Boltzmann Machines to learn a time-series predictor, which can predict the next pose of a motion given several previous frames. The spike-and-slab version of the recurrent temporal Restricted Boltzmann Machine can be used to improve the reconstruction of this approach [MKSL14]. The drawback of those time series approaches is that an edit in early frames can propagate far into the future which is not desirable for animators.

Holden et al. [HSK16] propose a procedural approach to motion synthesis. They first train a convolutional autoencoder to form a motion manifold [HSPK15]. This is then used to map high level parameters on the latent variables to produce a novel animation. This approach however requires the user to eliminate any ambiguity in the input by providing additional data if necessary. My system uses a similar structure but takes character poses as input which reduces the ambiguity enough to produce believable results.

2.1.4 Variational Autoencoders

A large body of work has used statistical dimensionality reduction, both at the individual pose and animation clip level. The generated subspace can be used for penalizing poses in posing tasks [GMHP04, WC11] or help with convergence in space-time optimization settings [CH07, MCC09a]. The spaces produced by these methods can rarely be manipulated directly by humans and are instead used to automatically enforce some constraints. Other approaches use a database of motions and interpolate between them to create a continuous motion [RBC98]. While automatic methods to build motion graphs exist [KGP02] they are generally built manually in practice.

In contrast variational autoencoders (VAE) condition the network to find an efficient encoding based on assumptions of the distribution in latent space [KW13]. An application of this technique was shown by Bowman et al. [BVV+15] who use the continuous space of the VAE to produce sentences with different properties. Another application is in image generation. [KW13] show how the latent space of the autoencoder can be used to produce realistic hand written digits. Walker et al. [WDGH16] use a conditional variational autoencoder to predict future events in a variety of scenes and produce multiple different predictions for an ambiguous future. One well known problem with VAEs is that the latent variables are often uninterpretable.

Recently Higgins et al. [HMG+16] showed that by scaling the KL divergence by a factor β the independence between latent variables is encouraged. This results in a disentangled representation which allows the latent variables to be directly interpreted. In this work I apply the idea of optimizing for independence in latent space to generate novel character motions with a small number of parameters which can be directly manipulated by humans.

2.2 Neural Networks

This section gives an introduction to neural networks since they are the central part of this work. First a short overview of the general structure of such networks is given and how they are applied to learn meaningful behavior. The following sections focus on specific kind of networks used in this thesis. The first network architecture presented are convolutional neural networks which apply a convolution between the layers to find localized features. The second type is a variational autoencoder which can encode and decode an input sample by predicting a low dimensional hidden representation.

2.2.1 Introduction To Neural Networks

In general an artificial neural network is simply a function $f : X \mapsto Y$ which takes samples from the input set X and produces an output in the set Y. The mapping is produced by connecting a number of so called neurons in a meaningful way. The value of each neuron is typically determined by a weighted sum over all inputs followed by an activation function. So for each neuron j in the network, its output o_j is defined as:

$$o_j = \phi(\sum_{k=1}^{n} w_{kj} o_k) \tag{2.1}$$

where ϕ is the activation function and w_{kj} is the weight between neurons k and j. A commonly used activation function is the rectified linear function $ReLU(z) = max(z, 0)$.

Neural networks are often structured into many layers where successive layers use the output of the previous layer as input. A simple multilayer network can be seen in figure 2.1.This structure allows for learning of feature representation in each layer with a hierarchy from low-level to high-level features [LD14].

Convolutional neural networks (ConvNet) take advantage of spatial structure in the input data. By only considering a window of neurons in the previous layer a ConvNet can reduce the total number of parameters through shared weights and exploits local correlation in the input data. Neurons of a convolutional layer are only connected to their receptive field using a set of weights which are reused for other neurons in the same layer but on a different input window. This lets the network detect localized features.

Units that are not in the input or output layer of the network are typically called hidden units. Those hidden layers process the input in such a way that the final output layer can use them to produce meaningful results. Such networks are usually trained using stochastic gradient

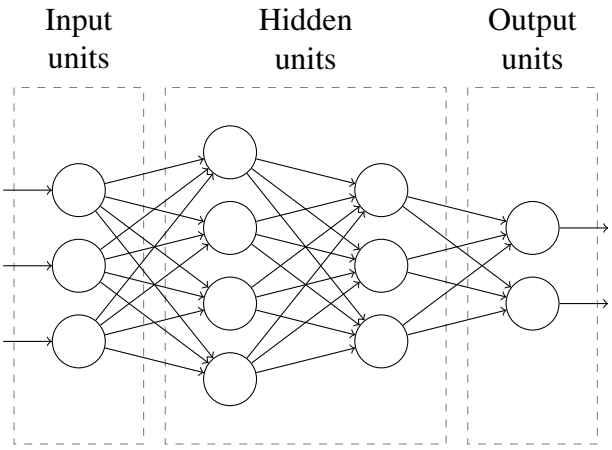


Figure 2.1: *Structure of a simple multi layer neural network.*

descent in which few random samples from the training set are used to compute an average gradient. The weights in the network are then adjusted using a method called *backpropagation* [RHW88] discussed in the next part. This process is repeated for many small sets until the function converges to a minimum.

Backpropagation

The *backpropagation* procedure for gradient computation is essentially just an application of the chain rule for derivatives [RHW88]. The key factor is that the gradient for the objective function with respect to the weights of the model can be computed by working through the layers in reverse order. To find the partial derivative with respect to a weight w_{ij}, the chain rule is applied twice:

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta o_j} \frac{\delta o_j}{\delta \text{net}_j} \frac{\delta \text{net}_j}{\delta w_{ij}} \tag{2.2}$$

where E is the objective function and net_j is the weighted sum of outputs o_k of the previous neurons.

Now only the last term depends directly on w_{ij}. This term can easily be computed using the fact that it is just the derivative of a weighted sum. Only the term containing the actual weight remains giving the output of neuron j as the final value for this term. If the neuron is in the input layer of the network this value becomes the input x_i.

The second term is just the partial derivative of the activation function, which is the reason why this function needs to be differentiable for the algorithm to work.

Finally the first term is straight forward if the neuron is in the output layer. In that case the derivative can easily be computed from the objective function. However if the neuron is in an inner layer the derivative needs to be calculated recursively depending on the neurons receiving the output o_j as input:

$$\frac{\delta E}{\delta o_j} = \sum_{l \in L_j} \left(\frac{\delta E}{\delta o_l} \frac{\delta o_l}{\delta \text{net}_l} w_{jl} \right) \tag{2.3}$$

where L_j is the set of neurons depending directly on j. It is therefore possible to compute the derivative if the derivatives of the next layer are known.

Putting it all together results in a recursive function which starts at the output layer and goes through all other layers in reverse order. The resulting gradient can then be used to update the model by applying some learning rate α to the gradient and adding it to the weights.

2.2.2 Variational Autoencoders

The aim of an autoencoder is to learn an encoding of the input data in some latent space. They consist of an encoder which produces a representation in latent space for some input and a decoder which reconstructs the original input from some sample in latent space. Training an autoencoder is usually done by giving the output of the encoder to the decoder and minimizing the reconstruction error. By imposing additional costs on the latent representation the autoencoder can be trained to produce specific types of encodings.

A variational autoencoder constrains the encoding network to generate latent vectors which roughly follow a unit Gaussian distribution, meaning that a sample in visible space does not rely on a single value as representation, but a distribution over an interval [KW13]. To achieve this the encoder network does not produce simple latent values, but a mean and standard deviation for the distribution of the latent variables. This greatly reduces the amount of information that can be stored in the latent vector itself. The network is thus encouraged to find information rich latent variables to encode the input. While this means that the reconstruction from latent space is potentially less accurate than by using a normal autoencoder, the single dimensions are much more meaningful. In general this means that each dimension gets assigned a specific task in reconstruction making it easier for humans to understand the effect when changing the latent values.

This behavior is enforced by adding an additional objective term to the reconstruction error which measures how close the produced distribution is to a unit Gaussian. the distance is computed using the Kullback-Leibler divergence [KL51] between two multivariate Gaussian distributions which can be computed in closed form as:

$$KL[\mathcal{N}(\mu_0, \Sigma_0)||\mathcal{N}(\mu_1, \Sigma_1)] = \frac{1}{2}\left(\text{tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1 - \mu_0)^\intercal \Sigma_1^{-1}(\mu_1 - \mu_0) - c + \log\left(\frac{\det \Sigma_1}{\det \Sigma_0}\right)\right) \quad (2.4)$$

where c is the dimensionality of the distribution. Using a unit Gaussian as target this simplifies to:

$$KL[\mathcal{N}(\mu(X), \Sigma(X))||\mathcal{N}(0, I)] = \frac{1}{2}\left(\text{tr}\left(\Sigma(X)\right) + (\mu(X))^\intercal(\mu(X)) - c - \log\left(\det \Sigma(X)\right)\right) \quad (2.5)$$

Here $\mu(X)$ and $\Sigma(X)$ are the mean and standard deviation obtained from the encoder network. By scaling the weight of this loss a more disentangled representation can be encouraged [HMG+16].

During training the input data is given to the encoder network to produce the mean and standard deviation in latent space. From this distribution a sample is drawn as input to the decoder network which produces a result in the original visible space. This approach however poses the problem that sampling from a distribution is not differentiable and thus the backpropagation algorithm stops working. To mitigate this problem a *reparameterization-trick* [KW13] is applied which shifts the sampling to an input layer. Instead of directly drawing a sample from the distribution $\mathcal{N}(\mu(X), \Sigma(X))$ a sample $\epsilon \sim \mathcal{N}(0, I)$ is drawn and $z = \mu(X) + \Sigma^{1/2}(X) * \epsilon$ is produced as the latent representation. Thus a gradient can now be computed and the backpropagation algorithm will work for stochastic gradient descent.

For testing the latent values are no longer produced by the encoder network but are either directly sampled from a unit Gaussian distribution or selected manually. Due to the assumption of a unit gaussian distribution in the latent components, the variables form a continuous space from which individual samples can be drawn.

3

Data Acquisition

This chapter describes the process of constructing the animation database I use to train the models. Since motion capture clips often feature different joint structures and bone length I first describe how to convert different skeletons into a common uniform skeleton. The motion representation fed to the network can have an impact on learning performance. I wish to learn a mapping that is invariant to global translation and rotation. Hence the second part describes how I convert the motions into a local body frame coordinate system with origin located on the ground where the root position is projected onto. By representing each joint position relative to the body frame it is easy to determine the dissimilarity of two poses by taking the Euclidean distance between the joint positions.

3.1 Animation Data

The database I use consists of freely available motion capture clips from the *CMU* dataset [CMU] as well as some data from internal captures which are recorded using a *Perception Neuron* system [Noi]. From these sources only clips showing a walking or running motion are added to the database. The final data set contains about 1.5 million frames of captured motions, sampled at 120 frames per second. Most clips feature a walking motion over flat terrain with various left and right turns. There are however some backwards and sideways walks, runs, and walks over uneven terrain.

The motion clips are converted to a reduced skeleton to decrease the degrees of freedom for the neural network and provide a uniform input. This skeleton features 21 joints and can be seen in figure 3.1. First I manually assign any correspondence of joints between the source and the common skeleton. Given a joint correspondence, I transfer the angle from the source to the target. To find any missing angles and adjust for other discrepancies the original skeleton is scaled to the same size as the reduced skeleton and the joint positions are computed with forward

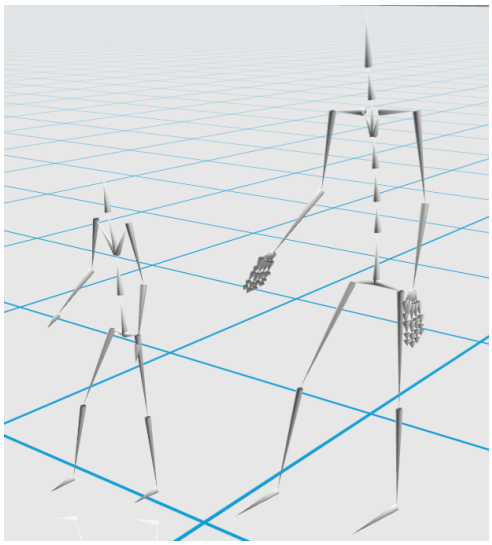


Figure 3.1: *Left: The reduced skeleton used in training the model. Right: The full skeleton produced by the* Perception Neuron *motion capture system.*

kinematics. These global positions are then used as target for a full-body inverse kinematics solve [Bus04] performed on the reduced skeleton. The joint angles are determined using a gradient descent algorithm minimizing the dissimilarity.

3.2 Data Formating

To convert the skeleton into a body-local coordinate system suitable for training, I start by subsampling the motion to 60 frames per second, interpolating the joint angles for any missing poses. The global 3D joint positions of the character are then computed using forward kinematics. From these positions the forward direction of the character is determined using the vectors between left and right shoulder, as well as left and right hips, averaging them and taking the cross product with the vertical axis $\mathbf{y}$. All joint positions are then defined in this coordinate system using the projection of the root position onto the ground as origin. I compute the relative horizontal and rotational velocity of the root for every frame and add them to the input representation. These velocities are defined in the body-local coordinate system such that any part of the animation is independent of the initial position and rotation. The velocities can be integrated over time to recover the global translation and rotation of the skeleton.

I compute the mean pose and standard deviation over the whole data set and normalize each frame by removing the mean and dividing by the standard deviation. The same process is applied to the horizontal and rotational velocities.

Finally the clips are divided into equal length segments to produce the training sets for the synthesis network and variational autoencoder. For the former I use a length of 121 frames which corresponds to two seconds of motion, overlapped by 61 frames. This produces a training set containing about ten thousand samples. The autoencoder on the other hand is trained on shorter clips of 61 frames or one second, overlapped by 31 frames. This set contains about 20 thousand training samples.

The described format is useful for several reasons. It allows to compute the dissimilarity of two poses using simple Euclidean distance. Additionally Through the use of relative velocities any part of the animation is independent of previous segments which supports the convolutional neural network in finding independent features. The clip length is chosen such that a single produced sample in the synthesis network only influences a limited window of the whole animation while still being able to plan for future events. The autoencoder uses shorter clips to make changing parameters more intuitive for the user.

4

Neural Network

4.1 Motion from Poses

To generate Animations from input poses I build a deep feed forward convolutional neural network. It consists of several layers, each performing a convolution followed by an inverse max pooling operation. To note here is that the convolution is only conducted in the time domain contrary to the 2D convolution used in image processing. Since each joint of the skeleton depends on the others it is not helpful to do a convolution over the input feature space. The details of the network are described below.

4.1.1 Network Structure

The input poses go through four convolutional layers. Each layer performs a convolution over the temporal domain for each filter separately and adds a bias to the neurons. The neurons are then activated using a rectified linear operation, which was shown to be effective by Nair and Hinton [NH10]. The values of weights and biases are variables which define the behavior of the network and are learned in the training phase.

The first three convolutional layers are followed by an inverse pooling operation. These increase the temporal resolution of the output by spreading the value of an input neuron over several output neurons. The number of neurons produced by each input value is different for each layer and is chosen in such a way that the final animation has the desired number of frames.

As such the feed forward network can be described as a function $\Pi(\boldsymbol{X})$ receiving an input vector $\boldsymbol{X} \in \mathbb{R}^{k \times d}$ consisting of k input poses with d degrees of freedom and generating an

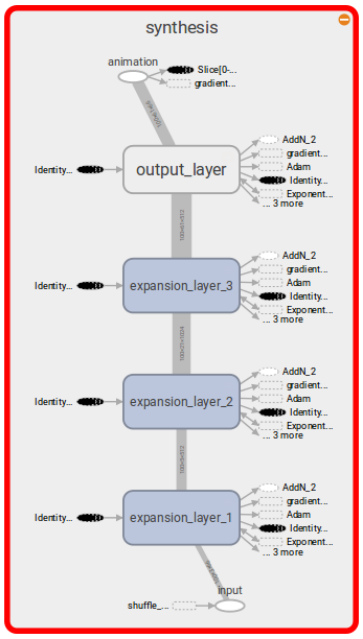


Figure 4.1: *An overview of the synthesis network produced by* Tensorboard.

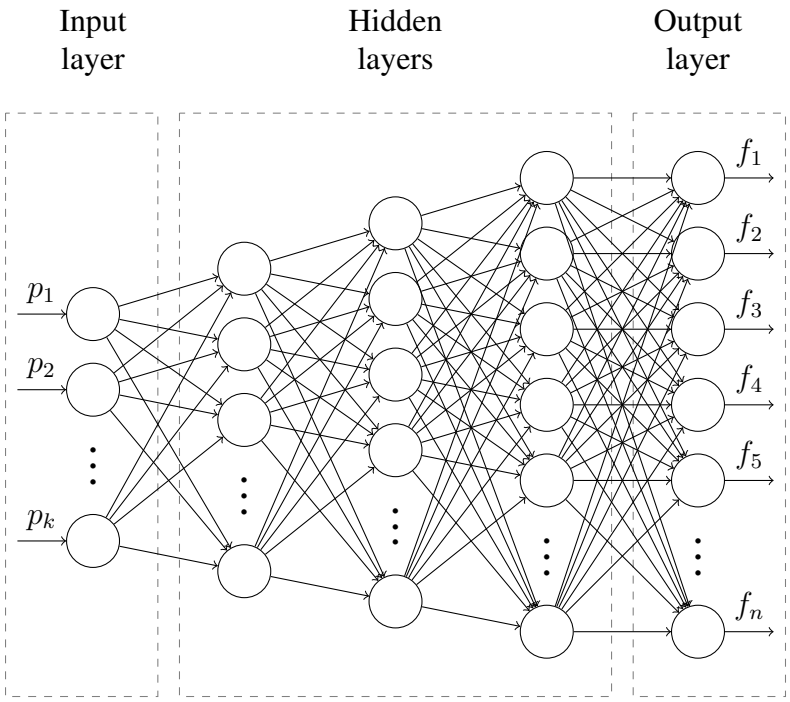


Figure 4.2: *General structure of the synthesis network. Each layer performs a convolution and expands the input in the temporal domain through an inverse max pooling operation.*

output vector $\hat{\boldsymbol{Y}} \in \mathbb{R}^{n \times d}$ containing n frames of animation using the same skeleton as the input:

$$\Pi(\boldsymbol{X}) = \Psi_3(ReLU(\Psi_2(ReLU(\Psi_1(ReLU(\boldsymbol{X} * \boldsymbol{W}_1 + \boldsymbol{b}_1)) * \boldsymbol{W}_2 + \boldsymbol{b}_2)) * \boldsymbol{W}_3 + \boldsymbol{b}_3)) * \boldsymbol{W}_4 + \boldsymbol{b}_4 \quad (4.1)$$

In this formula $*$ denotes a convolution using weight matrices $\boldsymbol{W}_i \in \mathbb{R}^{m_{i-1} \times m_i \times w_i}$ where m_{i-1} is the number of units in the previous layer, m_i the number of units in the next layer and w_i the convolution filter width. m_0 and m_4 are therefore just the degrees of freedom d of the skeleton. Accordingly the bias $\boldsymbol{b}_i$ is a vector in $\mathbb{R}^{m_i}$. Ψ_i is an inverse max pooling operation in the temporal axis with an expansion factor of e_i and $ReLU(\boldsymbol{x})$ is the nonlinear rectifying operation and is defined as $\max(\boldsymbol{x}, 0)$. The parameters used in this work can be found in table 4.1.

4.1.2 Training

During training the network is presented with a collection of input poses extracted from an original animation $\boldsymbol{Y}$ which is in the format described in chapter 3. From these poses the network generates an animation $\hat{\boldsymbol{Y}}$ which is compared to the original using simple Euclidean distance. Additionally I penalize any deviation from the input poses using a similar metric. Training is therefore given as an optimization problem where the following cost function is minimized with respect to the network parameters $\theta = \{\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3, \boldsymbol{W}_4, \boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3, \boldsymbol{b}_4\}$:

$$Cost(\boldsymbol{X}, \boldsymbol{Y}, \theta) = \alpha||\boldsymbol{Y} - \Pi(\boldsymbol{X})||_2^2 + \beta||\boldsymbol{X} - \Pi(\boldsymbol{X})_p||_2^2 + \gamma||\theta||_1 \quad (4.2)$$

d	66	w_1	3	e_1	2
m_1	256	w_2	7	e_2	5
m_2	512	w_3	13	e_3	3
m_3	256	w_4	21		

Table 4.1: *Parameters used in the synthesis network.*

The first term measures the squared reproduction error using the Euclidean distance to the original animation. The second term penalizes any deviation from the input poses by extracting the poses $\Pi(\boldsymbol{X})_p$ from the output and comparing them to the input. Finally the third term represents an additional sparsity term that encourages the use of a minimal number of network parameters. In my implementation α is set to 1 while β and γ are set to a small constant of 0.1.

For training the convolution weights are initialized to small random values using a Xavier initialization [GB10]. This method chooses the weights according to a "fan-in" and "fan-out" criteria representing a truncated normal distribution with zero mean and a standard deviation based on the number of incoming and outgoing neurons. The biases are initialized to zero.

To minimize the cost function I perform stochastic gradient descent. The samples are randomly drawn from the database and using derivatives computed via *Tensorflow* [AAB+15] I update the parameters θ. Here I make use of the adaptive gradient descent algorithm *Adam* [KB14] to improve training speed and quality. To further ensure convergence I use an exponentially decaying learning rate which starts at a step size of 0.001 and gets decreased by a factor of 0.1 every 50 epochs. This prevents that the optimization process gets stuck in a local minimum too early but allows it to reach closer to the minimum in later epochs.

To prevent overfitting to the training data I use a *dropout* [SHK+14] of 0.2. This technique decides with the given probability to drop a neuron from the network which can be viewed as training an exponential number of different networks. For the actual synthesis process those networks are averaged by scaling the output of each neuron with the dropout probability.

Training is performed for 200 epochs and takes around six hours on two NVIDIA GeForce Titan X GPUs. Both GPUs receive a batch of data to compute the gradients separately. The individual gradients are then averaged on the CPU and applied to the model.

4.1.3 Combining Sequences

Since the network is trained to generate animations of 121 frames, a special technique is required to produce longer sequences. For this purpose I split the input poses into overlapping segments and produce a reduced animation for each of them. Those resulting motion segments are then blended together using the frame-wise mean to produce a smooth transition. This approach is similar to the framework proposed by Taehwan et al. [KYTM15] who successfully apply it to facial animations.

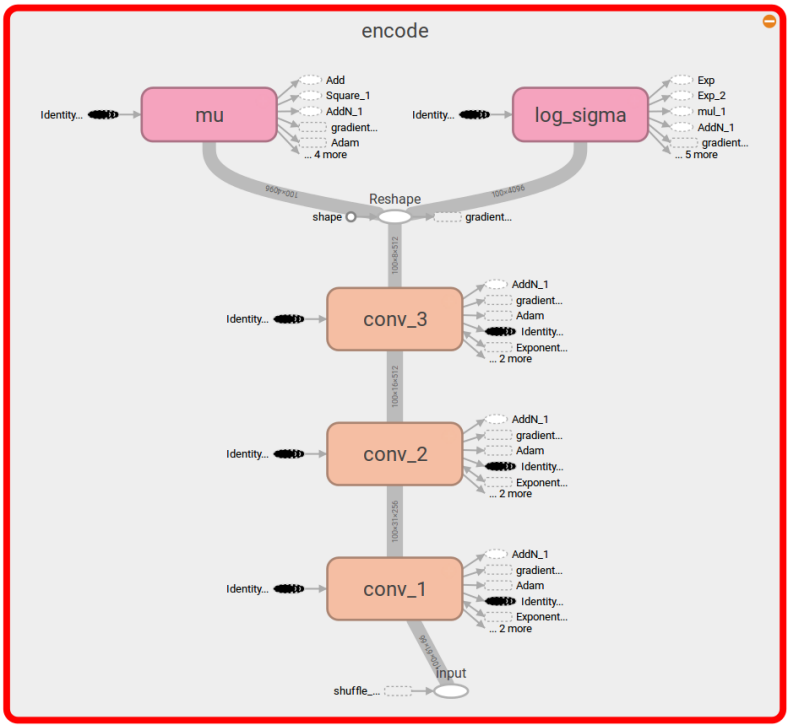


Figure 4.3: *The structure of the encoder as seen in* Tensorboard.

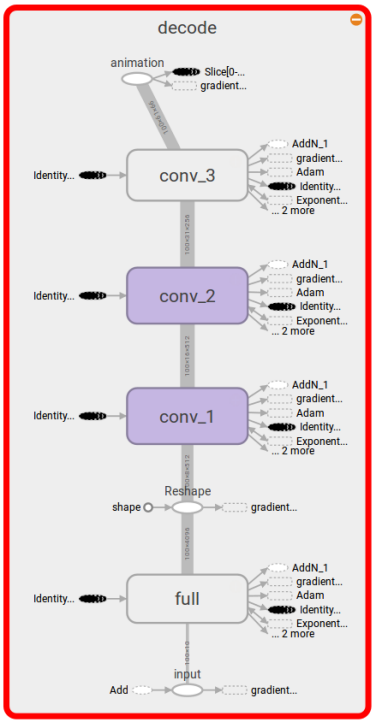


Figure 4.4: *The structure of the decoder as seen in* Tensorboard.

4.2 Variational Autoencoder

The variational autoencoder uses a similar structure as the synthesis network described in the previous section. However during training it uses two separate networks to encode and decode the animation clips. The encoding network takes as input a motion clip and produces a mean and standard deviation for the multi-variate Gaussian distribution describing the latent space. From this distribution a sample is drawn and given as input to the decoder which reconstructs a full animation. For the actual motion synthesis task only the decoder is used by feeding it samples from latent space directly. The structure of the autoencoder during training can be seen in figure 4.5. In this section I first describe the encoding and decoding neural networks and then explain how it is trained.

4.2.1 Structure

encoder

The encoding network takes as input an animation $\boldsymbol{Y} \in \mathbb{R}^{n \times d}$ and produces an output that is divided into two parts: The predicted mean $\boldsymbol{\mu} \in \mathbb{R}^c$ and the predicted standard deviation $\boldsymbol{\sigma} \in \mathbb{R}^c$ of a multi-variate Gaussian distribution with c components describing the encoded latent variables for the given input. The animation first goes through three convolutional layers, each followed by a max pooling operation to reduce the temporal resolution. After those layers

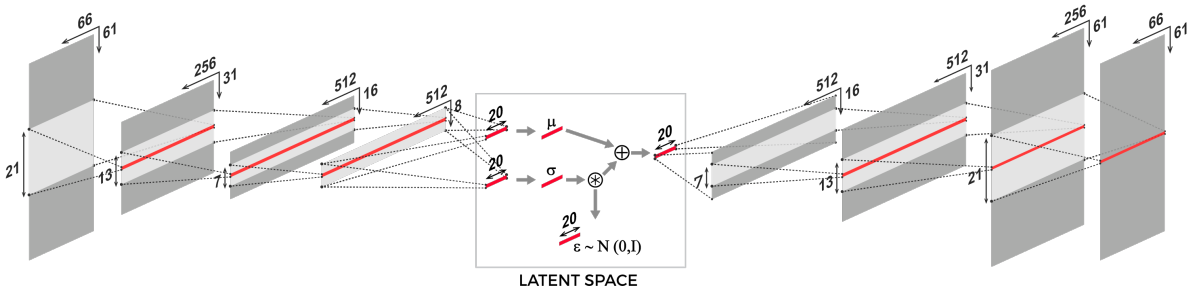


Figure 4.5: *The structure of the variational autoencoder at training time. The μ and σ generated by the encoder are combined with a random normal distributed sample to generate the input for the decoder.*

I use two separate fully connected networks to compute the mean and standard deviation.

In summary I have a convolutional network $\Phi(\boldsymbol{Y})$ to generate an intermediate hidden state $\boldsymbol{H}'$ which is then fed into two independent fully connected networks $\Phi_\mu(\boldsymbol{H}')$ and $\Phi_\sigma(\boldsymbol{H}')$. Those networks yield the mean $\boldsymbol{H}_\mu \in \mathbb{R}^c$ and standard deviation $\boldsymbol{H}_\sigma \in \mathbb{R}^c$ of the distribution in latent space. Because the standard deviation needs to be positive I first generate the logarithm which can be exponentiated to receive a positive value.

The convolutional network can be described ast he following function:

$$\Phi(\boldsymbol{Y}) = ReLU(\Psi(ReLU(\Psi(ReLU(\Psi(\boldsymbol{Y} * \boldsymbol{W}_1 + \boldsymbol{b}_1)) * \boldsymbol{W}_2 + \boldsymbol{b}_2)) * \boldsymbol{W}_3 + \boldsymbol{b}_3)) \tag{4.3}$$

where Ψ is a max pooling layer in the temporal axis with a reduction factor of 2 and the operator $*$ again denotes a convolution in the temporal domain. $\boldsymbol{W}_i \in \mathbb{R}^{m_{i-1} \times m_i \times w_i}$ and $\boldsymbol{b}_i \in \mathbb{R}^{m_i}$ are the weights and biases used in the convolution layers and are the parameters to be learned for the network. As before m_i denotes the number of neurons in a specific layer and w_i is the convolutional filter width.

The two fully connected networks which produce the mean and standard deviation for the latent Gaussian distribution are described with the following function:

$$\Phi_\mu(\boldsymbol{Y}) = flat(\Phi(\boldsymbol{Y})) \cdot \boldsymbol{W}_\mu + \boldsymbol{b}_\mu \tag{4.4}$$

$$\Phi_\sigma(\boldsymbol{Y}) = \exp\left(flat(\Phi(\boldsymbol{Y})) \cdot \boldsymbol{W}_\sigma + \boldsymbol{b}_{\log(\sigma)}\right) \tag{4.5}$$

where $\cdot$ is a matrix multiplication with the weights $\boldsymbol{W}_\mu \in \mathbb{R}^{(s \cdot m_3) \times c}$ and $\boldsymbol{W}_\sigma \in \mathbb{R}^{(s \cdot m_3) \times c}$ representing the fully connected layer. The $flat()$ function simply reshapes the output of the convolutional network to be a one dimensional vector. In this case s is the temporal resolution after the final convolutional layer. The parameters used in the encoder can be found in table 4.2.

decoder

The decoder $\Phi^\dagger$ functions very similar to the encoder but in reverse order and reconstructs an output animation $\hat{\boldsymbol{Y}} \in \mathbb{R}^{n \times d}$ with n frames. An input sample $\boldsymbol{H} \in \mathbb{R}^c$ in latent space is first processed in a fully connected layer and then given to a convolutional network which increases

d	66	w_1	21
m_1	256	w_2	13
m_2	512	w_3	7
m_3	512		
c	20		

Table 4.2: *Parameters used in the encoder network*

c	20	w_1	7
m_0	512	w_2	13
m_1	512	w_3	21
m_2	256		
m_3	66		

Table 4.3: *Parameters used in the decoder network*

Figure 4.6: *Value of the loss functions during training of the variational autoencoder. From left to right: KL-divergence loss, regularization penalty, reconstruction error, total loss.*

the temporal resolution with inverse pooling operations. The decoding network is described as the following function:

$$\Phi^{\dagger}(\boldsymbol{H}) = \Psi^{\dagger}(ReLU(\Psi^{\dagger}(ReLU(\Psi^{\dagger}(flat^{\dagger}(\boldsymbol{H} \cdot \boldsymbol{W}_0 + \boldsymbol{b}_0)) \\ * \boldsymbol{W}_1 + \boldsymbol{b}_1)) * \boldsymbol{W}_2 + \boldsymbol{b}_2)) * \boldsymbol{W}_3 + \boldsymbol{b}_3 \quad (4.6)$$

where $flat^{\dagger}$ brings the one dimensional output of the fully connected layer into the correct shape for further processing in the convolutional layers. In this case the output of the $flat^{\dagger}$ function is a matrix $\boldsymbol{H}' \in \mathbb{R}^{s \times m_0}$ where s is again the temporal resolution after down sampling the original input and m_0 is the number of hidden units in the first layer. $\Psi^{\dagger}$ denotes an inverse pooling operation similar to the one used in the synthesis network. However this time the expansion factor is fixed to 2 for all layers to reflect the max pooling from the encoder. The exact parameters used for the decoder can be found in table 4.3

4.2.2 Training

The autoencoder is trained to encode an animation clip $\boldsymbol{Y}$ into latent space and then decode it back into full space. The difference to traditional autoencoders is that the latent variables are modeled as a multi-variate Gaussian distribution. As a result the input for the decoder is drawn from the Gaussian distribution defined by the mean and standard variation generated by the encoder. This however represents a non-continuous operation which has no gradient. To work around that I apply a ”reparameterization trick” [KW13] by moving the sampling to an input layer. The sample $\boldsymbol{s}$ given to the decoder is computed using the formula

$$S(\boldsymbol{\mu}, \boldsymbol{\sigma}) = \boldsymbol{\mu} + \boldsymbol{\epsilon} * \boldsymbol{\sigma} \quad (4.7)$$

where ϵ is a random sample generated from a standard normal distribution $\epsilon \sim \mathcal{N}(0, I)$. This trick makes it possible to train the whole autoencoder network using the backpropagation algorithm.

For each input sample I minimize the distance between the original animation and the reconstruction $\Phi^{\dagger}\left(S(\Phi_{\mu}(\boldsymbol{Y}), \Phi_{\sigma}(\boldsymbol{Y}))\right)$. An additional goal of the optimization is to achieve a standard deviation of one for each latent dimension, thereby maximizing the efficiency of the encoding and preventing high granularity in the components. As seen in section 2.2 this can be achieved through a KL-divergence term computed from the generated mean and standard deviation in the latent layer of the autoencoder, which measures how closely the distribution matches a canonical multi-variate Gaussian.

This leads to an optimization problem where I minimize the following cost function with respect to the network parameters θ for both the encoder and decoder:

$$Cost(\boldsymbol{Y}, \theta) = \alpha||\boldsymbol{Y} - \Phi^{\dagger}(S(\Phi_{\mu}(\boldsymbol{Y}), \Phi_{\log(\sigma)}(\boldsymbol{Y})))||_2^2 + \beta KL(\Phi_{\mu}(\boldsymbol{Y}), \Phi_{\sigma}(\boldsymbol{Y})) + \gamma||\theta||_1 \quad (4.8)$$

where the first term measures the squared reconstruction error and

$$KL(\boldsymbol{\mu}, \boldsymbol{\sigma}) = \frac{1}{2}\sum_{i=0}^{c}(\mu_i^2 + \sigma_i^2 - 1 - 2\log(\sigma_i)) \quad (4.9)$$

is the KL-divergence loss. As with the synthesis network there is an additional sparsity term to encourage usage of a minimal number of parameters.

At the start of training β is set to zero such that the network first learns a stable reconstruction before optimizing the distribution. Over the course of the session β gets increased using a sigmoid function which lets the network adapt to the additional cost term.

The actual training process is similar to the synthesis network. The weights are initialized using the "fan-in" and "fan-out" criteria used by [GB10] while the biases are initialized to zero. The function is minimized using stochastic gradient descent where samples are randomly drawn from the database. I use automatice derivative calculation via *Tensorflow*and the adaptive gradient descent algorithm *Adam* for this process. A dropout of 0.2 is used to avoid overfitting to the training data. The variational autoencoder is trained over 200 epochs on two NVIDIA Titan X GPUs.

Figure 4.6 shows the evolution of the error during training. The autoencoder first learns a hidden representation of the input before being exposed to the KL-divergence loss. Once it reaches a certain value the reconstruction error increases in order to adapt to this new cost.

4.2.3 Linear Reduction

To further reduce the number of dimensions in latent space, a subset of the training set is used to perform principle component analysis on their latent representation. This is found by a minimizing the reconstruction error with respect to the values of the latent variables. This minimization problem is solved by using a gradient descent algorithm to optimize the following cost function:

$$Cost(\boldsymbol{H}) = ||\boldsymbol{p}_r^{\boldsymbol{H}} - \boldsymbol{p}_r'||_2^2 + \sum_j ||\boldsymbol{p}_j^{\boldsymbol{H}} - \boldsymbol{p}_j'||_2^2 \quad (4.10)$$

where $\boldsymbol{p}_r^{\boldsymbol{H}}$ is the reconstructed global root position and $\boldsymbol{p}'_r$ the target root position. $\boldsymbol{p}_j^{\boldsymbol{H}}$ is the reconstructed local joint position and $\boldsymbol{p}'_j$ the target joint position. As an initial guess I use the mean generated through the encoding network for each clip. This process only aims to find the best latent representation without considering sparsity or any other metrics.

After this optimization step the latent values are used for principal component analysis and the first few dimensions are kept. That way the dimensionality of the latent space can be reduced from originally 20 dimensions to just 3 while keeping the most expressive features.

Since this process only allows to generate motions of one second length an interface was implemented to combine multiple sequences. The sequences are appended by overlapping them by half a second. Once the user is satisfied with the motion of a second he can add it to the complete animation. The next sequence is then chosen such as to match the ending of the previous segment to avoid cumbersome leg synchronization. That way the user can choose the step length and turn direction of the animation for every half second.

5

Experimental Results

This chapter presents the results of training the neural networks and synthesizing motions. I first show examples of walking animations created using the synthesis network described in section 4.1. Here I also compare results using different numbers of input poses. Next I show examples of synthesized motions using the variational autoencoder described in section 4.2. I also give a description of the effects associated with the different dimensions in the reduced latent space.

5.1 Motion Synthesis from Input Poses

The feedforward network is trained to generate a human walking animation given a set of input poses. For training these poses are extracted at regular intervals from clips in the motion database described in chapter 3. For testing, poses extracted from a test set are fed to the network and the result is compared to the original motion.

In a first run the network is trained using two poses per second as input. In this case the intermediate poses can be recreated with good accuracy and a realistic walking animation is produced as shown in figure 5.1. However due to the data format used the trajectory of the walked path often drifts away from the original motion which is especially noticeable in sharp turns as can be seen in figure 5.2. Another problem are motions that only rarely appear in the training set. Figure 5.3 shows the attempt of recreating a backwards walking character which appears only very sparsely in the motion database. While the poses are recreated with good accuracy the actual motion is in the opposite direction.

The second run is trained on an input of one pose per second of animation. Here the network has trouble creating novel frames and resorts to an average pose between the given input poses. Drift is an even bigger problem than before and the path of the input can generally not be

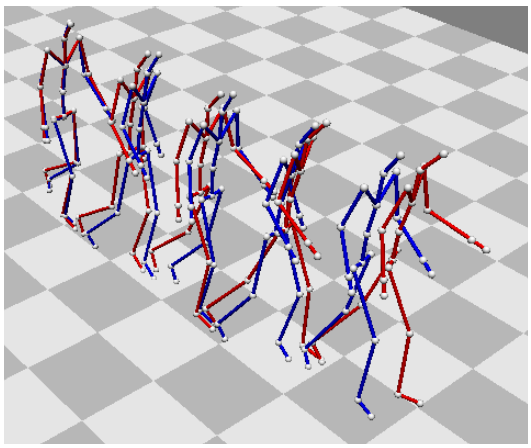


Figure 5.1: *Reconstruction (blue) with two input poses (red) per second. The poses are recreated quite faithfully but the character drifts away from the original path.*

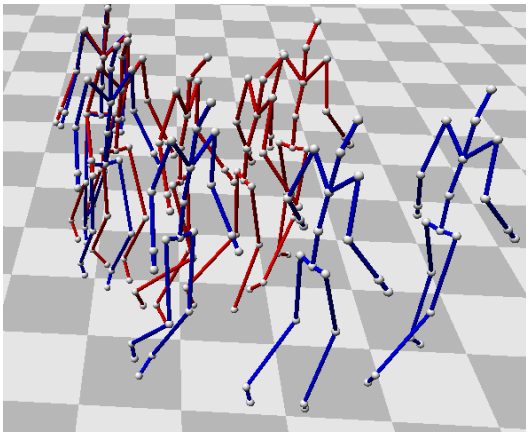


Figure 5.2: *The reconstruction (blue) drifts away from the original input (red). This error is not captured well by the network since it only sees relative positions.*

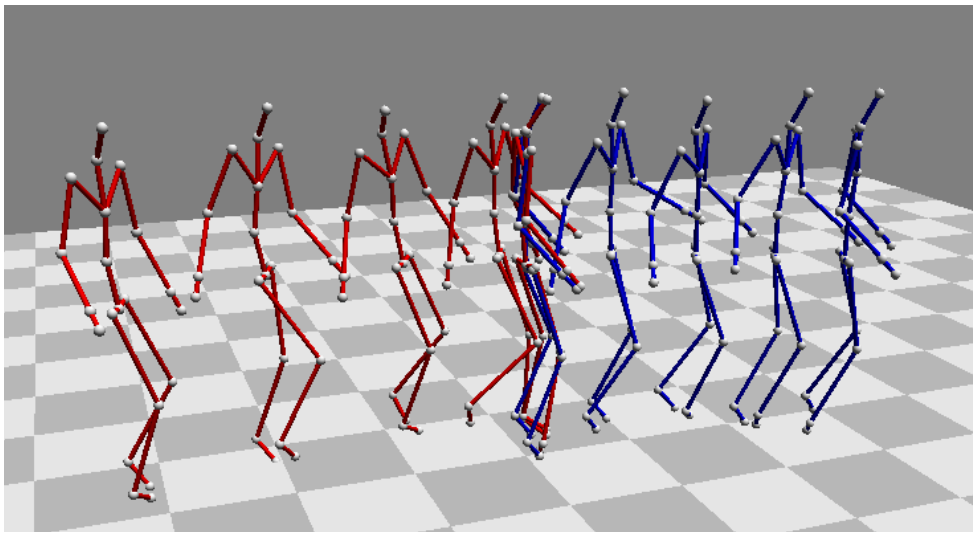


Figure 5.3: *The original backwards motion (red) can not be recreated. The network while creating correct poses still produces a forward motion.*

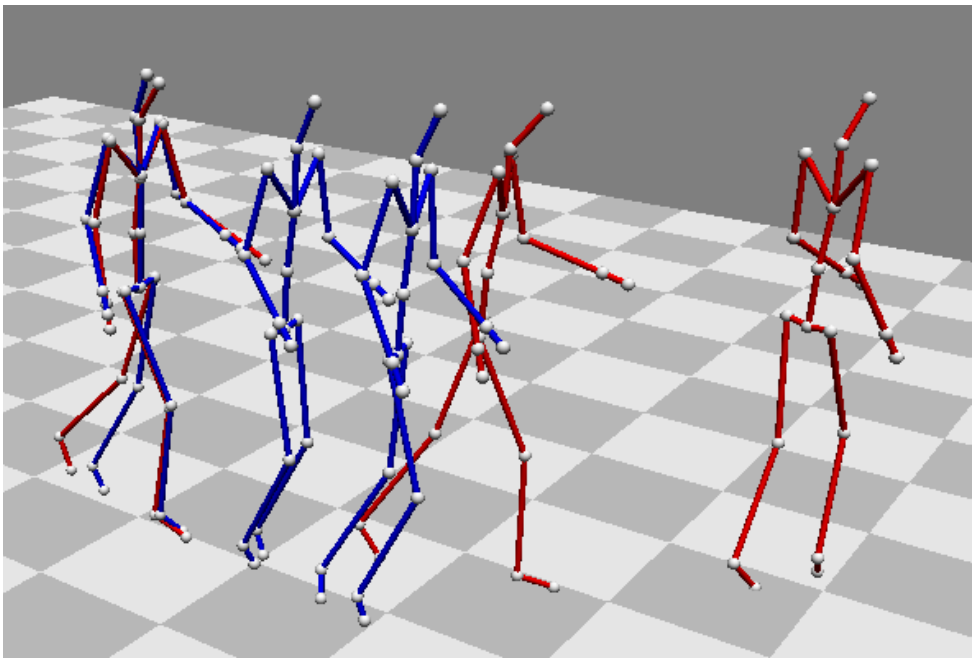


Figure 5.4: *Reconstruction (blue) with one input pose (red) per second. The motion can not be reconstructed.*

reconstructed. The network produces too short translations making the result a damped version of the original. A recreated animation can be seen in figure 5.4.

5.2 Variational Autoencoder

The variational autoencoder was trained on the walking animations of the training set. An interactive interface which is shown in figure 5.5 was implemented to test the quality of the reconstruction. The interface lets a user change the values of either the 20 latent units of the autoencoder or the final reduced 3 of the principal component analysis. Evaluating the network is fast and the reconstruction of a one second motion clip is displayed in real time.

While some dimensions produced by the variational autoencoder show a clear role such as turning left and right or increasing the length of steps, some dimensions do not appear to have a big influence on the generated motion. So to reduce the number of redundant dimensions I perform principle component analysis on the latent values computed from a subset of the training data. During this process I only retain the three most important components, leaving out others that have little influence on the result.

Using this approach the remaining variables have a clear effect on the generated animation. The first two components are associated with the left and right leg position. A negative value for the first dimension moves the starting position of the left leg to the back while a positive value moves it to the front. Similarly the second component moves the right leg to the back when set to a negative value and to the front with a positive one. The resulting animation is then dependent on the starting positions of the legs. It is obviously not possible to have both legs start in the front or back. If both values are set to negative the character starts from a neutral standing

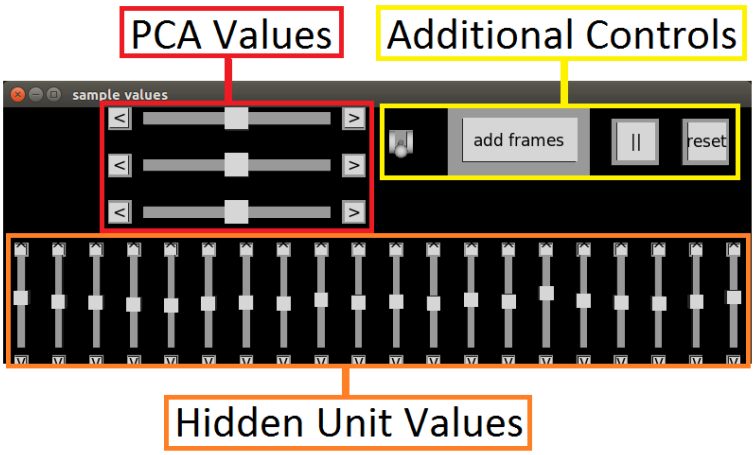


Figure 5.5: *The interface used to edit the PCA values. The hidden unit values can still be set directly if wished. The additional controls allow to toggle between the full animation clip and the currently being edited one, adding the current clip to the full animation, stopping or playing the animation and resetting the values for the current clip.*

position and takes small steps starting with the right leg. Having both be positive again creates a neutral starting position but the character starts with a step using the right leg. The distance between the two values for the first two components decides about the length of the steps the character takes. The starting positions for various combinations can be seen in figure 5.6.

The third component controls the direction in which the character turns during the walk. A negative value creates a rotation to the left while a positive value generates a curve in the other direction. The step length defined by the previous components is kept the same for all values in this component, allowing for intuitive editing. Using only those three values a wide range of walking animations can be generated.

To allow for further customization of the output additional components can be kept, however those have a significantly lower influence on the output. The fourth component can be used to make the character lean to the left or to the right while the fifth component lets the character walk hunched up or with a straight back. The effects of those two dimensions can be seen in figure 5.7. The remaining components only have a very minor influence on the outcome of the reconstruction.

To generate a longer motion the single clips are blended together. To facilitate the generation of the next sequence the starting point of the character is set such that the beginning matches the end of the previous animation as closely as possible. To this end the first two dimensions are reversed to change the starting leg of the animation.

I compare the results of principle component analysis in hidden space to the naive approach of applying the same technique to the training data directly without alignment. For this test I take the normalized training set and subtract the per clip mean. PCA is then applied using each clip as one 4026 dimensional data point. The algorithm finds some clearly separate components

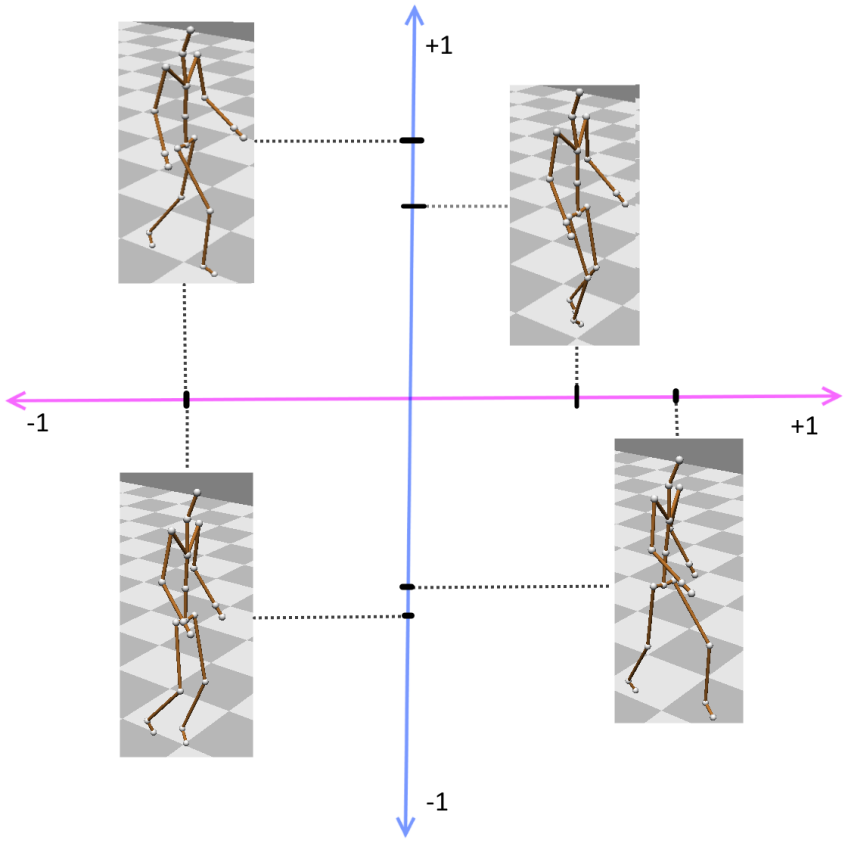
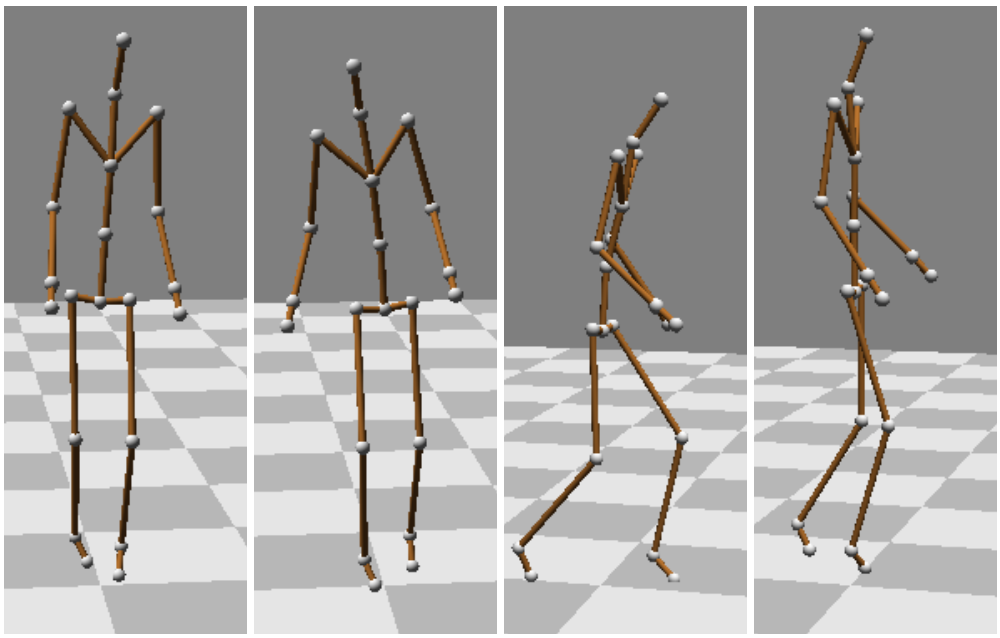


Figure 5.6: *Different starting poses using different values for the first two PCA components.*



(a) ***(b)*** ***(c)*** ***(d)***

Figure 5.7: *Effects of additional PCA components on the starting position. (a) and (b) are the result of setting the fourth component to a negative or positive value respectively. (c) and (d) show the same for the fifth component.*

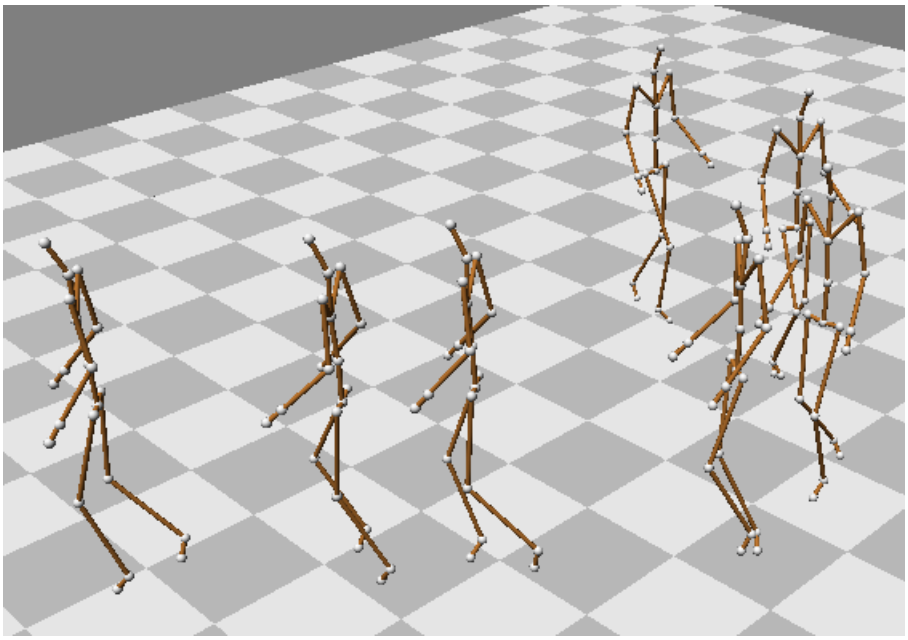


Figure 5.8: *An animation created with the variational autoencoder.*

although the difference in variation is not as high as when using the reduced subspace of the variational autoencoder. However unlike in my approach, the first few principle components do not capture any body movement and focus solely on motion in the 2D ground plane. In this case PCA can not separate meaningful dimensions because the clips are not aligned in any way and contain an arbitrary walking motion. This problem is solved using my variational autoencoder which learns the alignment of the clips during training.

6

Discussion

6.1 Motion Synthesis from Input Poses

I propose a novel approach to motion synthesis by using sparse input poses to create a full character animation. Many previous approaches use step by step calculation to produce an output animation [TH09, THR11, MKSL14]. In contrast my system uses a procedural approach which can generate frames at arbitrary times. This independence of frames means that the process can be greatly parallelized on a GPU allowing for very fast animation generation. In my tests the animation could be produced in realtime using only a CPU. In addition any changes to an input pose only have a localized effect on the produced animation, meaning animators do not have to worry about changes affecting the generated animation outside the small window of dependence. These characteristics make it a good fit for animation production software.

Procedural motion synthesis approaches have been proposed [LS99, KHKL09, MCC09b], however those methods require the motion to be segmented, aligned and labeled before it can be used. In my model this labeling is not necessary as the features of the animation are learned by the neural network. This allows for easy addition of new motions by adding the desired animation to the training set without any labeling or alignment.

A similar approach to motion synthesis using a deep neural network is proposed by Holden et al. [HSPK15]. Their system relies on a convolutional autoencoder for an intermediate representation of the animation to allow for easy editing. However to produce good results the user is required to provide additional data to reduce ambiguity in the output. In contrast my method relies solely on the input poses and can generate plausible animations without further information.

Different to current models for image synthesis [VLL+10, GPAM+14] which often feature two dimensional convolutions in both spatial domains my system only uses a one dimensional convolution over the time axis. A natural question would be if this could be enhanced by a spatial

component in the filters, for example over a hierarchical representation of the joints. This idea however poses some problems: In general the joints of a skeleton are dependent on each other such as the arm and leg movements in a walking animation. Also a learned motion of one body part is usually not applicable to other parts meaning that a transfer of filters is not useful. Using only a temporal filter allows for detection of those inter joint correlations.

The parameters in my model are determined first by intuition and are refined through experimental results. Those parameters are namely the filter widths of the convolutional layers in the generation network (w_1, w_2, w_3 and w_4 in equation 4.1) and the number of frames in a single output motion (n in equation 4.1). The filter widths are chosen to cover about the length of this output motion. The produced number of frames is chosen with respect to the desired sphere of influence. Using 120 frames gives two seconds of animation for each part of the generated motion. This allows for a single pose to influence about four seconds of animation, giving the network enough time to prepare for the input pose while still restricting the influence enough, such as to not overfit to the training data.

6.1.1 Limitations

As seen in section 5.1 the produced animations suffer from some drift. Using a motion manifold similar to the one used in [HSPK15] could allow for editing the animation in hidden space such that it follows the input poses more closely. As of now the output is generated in a single run making edits in the hidden layers impossible.

Another problem with the approach of stacked depooling layers is a blurring of the produced motion. This is especially obvious when using fewer input poses as the produced animation appears to be a heavily dampened version of the original input. A similar problem can be seen with more input poses where the produced motion tends to have a too high turning radius when it includes a rotation. The network also struggles to generate fast or sudden movements. One possible solution to this problem could be new approaches to depooling such as *hyperclumns* proposed by Hariharan et al. [HAGM15]. Another approach could again be to use a hidden representation of an autoencoder to enhance the animation manually.

The biggest drawback of this approach is the dependence on good input poses. Since the network tries to match the given poses in the output they need to be of good quality in order for a realistic motion to be generated. While the intermediate frames are synthesized automatically an artist still needs to model a considerable amount of keyframes for the animation. Additionally since the poses need to be evenly spaced, it might not be possible to time an event correctly in the generated animation. It would be interesting to see if a network can be trained on general poses as input instead of fully modeled keyframes. In that case a joint specific importance function could be used to indicate how close the resulting motion should follow the given joint.

6.2 Variational Autoencoder

The second system I propose uses a variational autoencoder to learn a low dimensional representation of human walking animations. The number of parameters is further reduced by

applying principle component analysis on representations of various motions in latent space of the autoencoder. This reduced space can be used by humans directly to edit the generated animation. The system aims at giving the user an interface with few meaningful parameters to produce believable animations. With just three dimensions a walking motion on flat terrain can be generated, including walking around corners and taking long or short steps. The effects of every dimensions after reconstruction are clearly visible to a human and are thus intuitive to control. While they are not completely independent of each other, the effect of changing one component is still consistent for any value of the other components. This is in contrast to applying principle component analysis directly to the training set where no meaningful components are learned.

Using a variational autoencoder allows for arbitrary motions to be represented in a very low dimensional subspace without any alignment or labeling of the clips. Not having to manually label and align the input clips makes it easy to add new data for training. In addition a complex motion can be generated using only three variables. The expressiveness can be enhanced by providing further parameters for editing. Since the components found by PCA are ordered after the magnitude of their effect it is easy to add new variables and gives the system a good flexibility. A novice user might be content with only three parameters while an expert may use further components to enhance the output.

An important parameter in this model is the number of dimensions in latent space of the autoencoder. In practice there is a trade-off between having as few dimensions as possible and the reconstruction quality. In my case having more dimensions only marginally improves the results. With 100 latent variables the reconstruction is still largely the same as with 20 but the complexity for the user is much higher. On the other hand having less dimensions can reduce the reconstruction quality quite substantially. With too few latent components the model can not capture enough variation in the input to generate believable results. At 5 dimensions or less the produced animations are noticeably worse. Another consideration is that the autoencoder does not order the dimensions for their importance. Hence the effect of each dimension has to be tested individually, while using PCA allows for direct sorting with respect to the highest variation. As a compromise I opted to use 20 dimensions because this allows for accurate reconstruction and performing PCA in this latent space to get an ordering of the most important dimensions.

A similar problem arises with the length of the input sequence. Longer clips require the model to be more expressive meaning the number of latent variables needs to be higher for an accurate reconstruction. With 61 frames I cover a range of 1 second per clip. In this time a normal walking motion takes about two steps which lends itself well as a cutting point.

The variational autoencoder features similar convolutional filters as the synthesis network. Again the filter widths are chosen such that they cover the whole clip. Choosing this value too high results in smoothed out reconstructed motions or even failure to converge. Setting the width too short will produce noisy outputs as the system does not learn the smoothness from the data.

6.2.1 Limitations

Due to the nature of variational autoencoders an animation produced from a previously encoded clip can have a larger reconstruction error than a similar sparse autoencoder. This is due to the latent space being modeled as a multi-variate Gaussian distribution meaning a single variable holds less information. On the other hand each dimension is more independent and thus better suited for human interpretation.

Having only three variables to control the motion leads to outputs which are quite similar to each other. The components allow only for high level motion description such as step length and turn direction. Here further variable components can be used to add a notion of style to the produced animation, though the effect is still limited. For this purpose the network could be trained on specific stylized motions. That way a latent variable may be used to change the style of the generated animation.

As of now the variational autoencoder is incapable of producing backward or sideway walking as well as running animations. This is due to the fact that those motions only very rarely appear in the training set and are thus not influential enough in the training phase. With a more balanced data set this issue could be overcome.

7

Future Work

My synthesis system can be enhanced by using a learned motion manifold similar to [HSPK15]. That way the drifting issue currently present can be addressed by optimizing an additional objective in latent space of the autoencoder. Using this latent optimization technique a user could specify if the network should favor matching the input poses or produce individual output. This can be achieved by varying the penalty for pose matching in this additional optimization step.

As an alternative to the standard convolutional autoencoder the variational autoencoder described in this work can offer a great alternative. This gives the artist an interface to directly alter the produced animation in latent space instead of relying on automated edits. For this purpose the variational autoencoder can be modified to take as input the original poses in addition to the sample in latent space. That way the edits from an artist are applied to the resulting motion while still keeping the original input poses as a starting point.

Another approach to reconstructing an animation from input poses can be achieved by using a recurrent neural network. This type of networks can be used to produce a sequence-to-sequence mapping and has been successfully used for language processing [SVL14]. A similar transformation can be applied to the variational autoencoder [BVV$^+$15], allowing for a variable length output with a fixed set of parameters. This approach however would violate the paradigm of having a procedural generation of the animation since changes get propagated through the network.

It will also be interesting to see how new depooling techniques affect the result of generating animations. Hariharan et al. [HAGM15] propose the use of *hypercolumns* to overcome the disparity between neurons in early and late layers of a deep neural network. This procedure can potentially be used to reduce the blurring effect of the traditional depooling process. In the application of animation synthesis this could reduce the drifting found in motions with fast or sudden movements and allow for fewer poses to be used as input. The same approach can be used to improve the variational autoencoder, possibly reducing the reconstruction error.

8

Conclusion

I propose two deep learning approaches to motion synthesis. The first technique learns a mapping from sparse input poses to a full animation through a deep convolutional neural network trained on a large set of human walking animations. The framework does not rely on aligned or labeled data and is thus easy to apply to other motions.

Currently the produced motion can only be edited in a dedicated program. An improvement could therefore be achieved by combining this system with some autoencoder similar to the approach proposed by Holden et al. [HSK16]. That way the animation could be edited directly in the reduced latent space, facilitating the removal of artifacts or allowing for some style to be imposed on the motion.

In a second approach I apply the concept of a variational autoencoder on human walking motions to learn a low dimensional representation. The autoencoder is able to produce a latent space with 20 dimensions from a misaligned collection of walking motions. I further reduce this space to 3 components through principal component analysis. Using only these 3 variables it is possible to generate various realistic human walking motions.

One of the main benefits of this approach is that the learning process automatically identifies meaningful and disentangled latent variables that remain consistent throughout editing. Hence the effects of every dimension after reconstruction are clearly visible to a human and a user can manipulate these values to create believable animations. Currently the possible motions are limited to walking with different step sizes as well as walking around corners. It will be interesting to see this concept expanded to other types of motions such as running, or for automatically learning stylistic attributes.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Bus04] Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17(1-19):16, 2004.

[BVV+15] Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

[CH07] Jinxiang Chai and Jessica K. Hodgins. Constraint-based motion optimization using a statistical dynamic model. *ACM Transactions on Graphics (TOG)*, 26, 2007.

[CMS12] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[CMU] CMU. Carnegie Mellon University Mocap Database. http://mocap.cs.cmu.edu/.

[DWW15] Yong Du, Wei Wang, and Liang Wang. Hierarchical recurrent neural network

for skeleton based action recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1110–1118, 2015.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[GMH13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[GMHP04] Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *ACM Trans. Graph.*, 23(3):522–531, 2004.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[HAGM15] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Hypercolumns for object segmentation and fine-grained localization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 447–456, 2015.

[HG07] Rachel Heck and Michael Gleicher. Parametric motion graphs. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 129–136. ACM, 2007.

[HMG+16] Irina Higgins, Loic Matthey, Xavier Glorot, Arka Pal, Benigno Uria, Charles Blundell, Shakir Mohamed, and Alexander Lerchner. Early visual concept learning with unsupervised deep learning. *arXiv preprint arXiv:1606.05579*, 2016.

[HSK16] Daniel Holden, Jun Saito, and Taku Komura. A deep learning framework for character motion synthesis and editing. *ACM Trans. Graph.*, 35(4):138:1–138:11, July 2016.

[HSPK15] Daniel Holden, Jun Saito, Marza Animation Planet, and Taku Komura. A deep learning framework for character motion synthesis and editing. *IEEE Transactions on Visualization and Computer Graphics*, 21:1, 2015.

[JXYY13] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[KG04] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 559–568. ACM, 2004.

[KGP02] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. In *ACM transactions on graphics (TOG)*, volume 21, pages 473–482. ACM, 2002.

[KHKL09] Manmyung Kim, Kyunglyul Hyun, Jongmin Kim, and Jehee Lee. Synchronized multi-character motion editing. In *ACM transactions on graphics (TOG)*, volume 28, page 79. ACM, 2009.

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KTS+14] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[KYTM15] Taehwan Kim, Yisong Yue, Sarah Taylor, and Iain Matthews. A decision tree framework for spatiotemporal sequence prediction. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 577–586, New York, NY, USA, 2015. ACM.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LD14] Dong Yu Li Deng. Deep learning: Methods and applications. Technical report, May 2014.

[LK14] Sergey Levine and Vladlen Koltun. Learning complex neural network policies with trajectory optimization. In *ICML*, pages 829–837, 2014.

[LPY16] Libin Liu, Michiel Van De Panne, and KangKang Yin. Guided learning of control graphs for physics-based characters. *ACM Transactions on Graphics (TOG)*, 35(3):29, 2016.

[LS99] 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.

[LWB+10] Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. Motion fields for interactive character locomotion. In *ACM Transactions on Graphics (TOG)*, volume 29, page 138. ACM, 2010.

[LWH+12] Sergey Levine, Jack M Wang, Alexis Haraux, Zoran Popović, and Vladlen Koltun. Continuous character control with low-dimensional embeddings. *ACM Transactions on Graphics (TOG)*, 31(4):28, 2012.

[MCC09a] Jianyuan Min, Yen-Lin Chen, and Jinxiang Chai. Interactive generation of human animation with deformable motion models. *ACM Transactions on Graphics (TOG)*, 29, 2009.

[MCC09b] Jianyuan Min, Yen-Lin Chen, and Jinxiang Chai. Interactive generation of human animation with deformable motion models. *ACM Transactions on Graphics (TOG)*, 29(1):9, 2009.

[MK05] Tomohiko Mukai and Shigeru Kuriyama. Geostatistical motion interpolation. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1062–1070. ACM, 2005.

[MKSL14] Roni Mittelman, Benjamin Kuipers, Silvio Savarese, and Honglak Lee. Structured recurrent temporal restricted boltzmann machines. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1647–1655, 2014.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[Noi] Noitom. Perception neuron motion capture system.

[PBvdP16] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2016)*, 35(4), 2016.

[RBC98] Charles Rose, Bobby Bodenheimer, and Michael F. Cohen. Verbs and adverbs: Multidimensional motion interpolation using radial basis functions. *IEEE Computer Graphics and Applications*, 18:32–40, 1998.

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[RISC01] Charles F Rose III, Peter-Pike J Sloan, and Michael F Cohen. Artist-directed inverse-kinematics using radial basis function interpolation. In *Computer Graphics Forum*, volume 20, pages 239–250. Wiley Online Library, 2001.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[TH09] Graham W Taylor and Geoffrey E Hinton. Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th annual international conference on machine learning*, pages 1025–1032. ACM, 2009.

[THR11] Graham W Taylor, Geoffrey E Hinton, and Sam T Roweis. Two distributed-state

models for generating high-dimensional time series. *Journal of Machine Learning Research*, 12(Mar):1025–1068, 2011.

[VLL+10] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[WC11] Xiaolin Wei and Jinxiang Chai. Intuitive interactive human-character posing with millions of example poses. *IEEE Comput. Graph. Appl.*, 31(4):78–88, 2011.

[WDGH16] Jacob Walker, Carl Doersch, Abhinav Gupta, and Martial Hebert. An uncertain future: Forecasting from static images using variational autoencoders. In *European Conference on Computer Vision*, pages 835–851. Springer, 2016.

[WHB05] Jack Wang, Aaron Hertzmann, and David M Blei. Gaussian process dynamical models. In *Advances in neural information processing systems*, pages 1441–1448, 2005.

[YKH04] Katsu Yamane, James J Kuffner, and Jessica K Hodgins. Synthesizing animations of human manipulation tasks. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 532–539. ACM, 2004.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.